Ninevah University

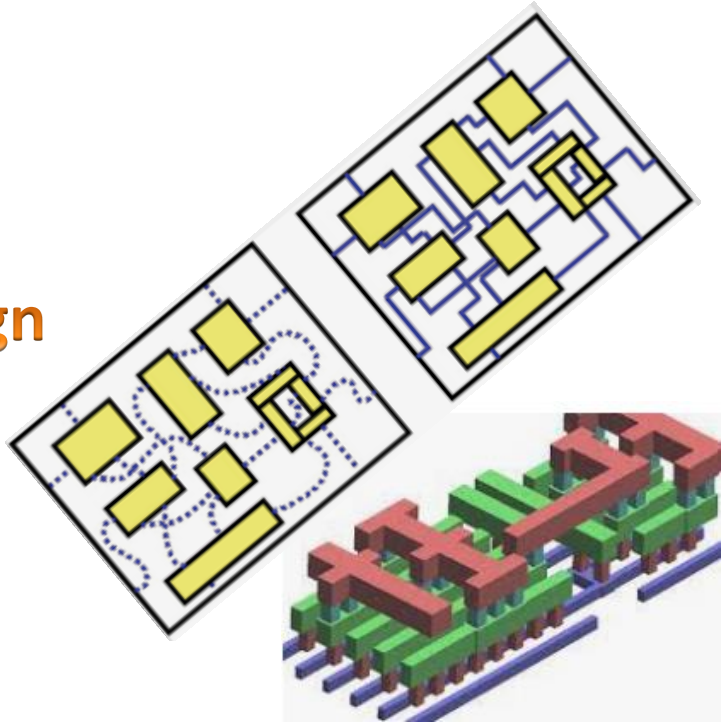
Collage of Electronics Engineering

Course:

Electronic Design Automation

Lecturer: H. M. Hussein

EDA11: Physical Design Automation – Routing



Grid Routing

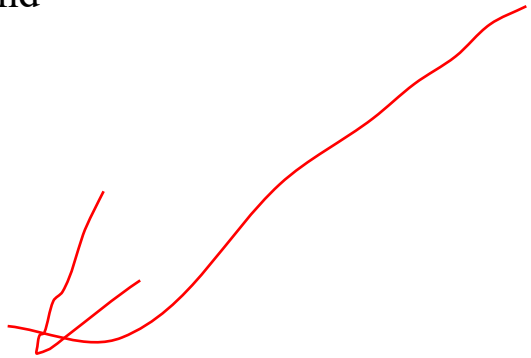
Introduction

• In the VLSI design cycle, routing follows cell **placement**.

• During routing, precise paths are defined on the layout surface, on which conductors carrying electrical signals are run.

• Routing takes up almost 30% of the design time, and a large percentage of layout area.

• We first take up the problem of **grid routing**.



What is Grid Routing?

• The layout surface is assumed to be made up of a rectangular array of grid cells.

• Some of the grid cells act as obstacles.

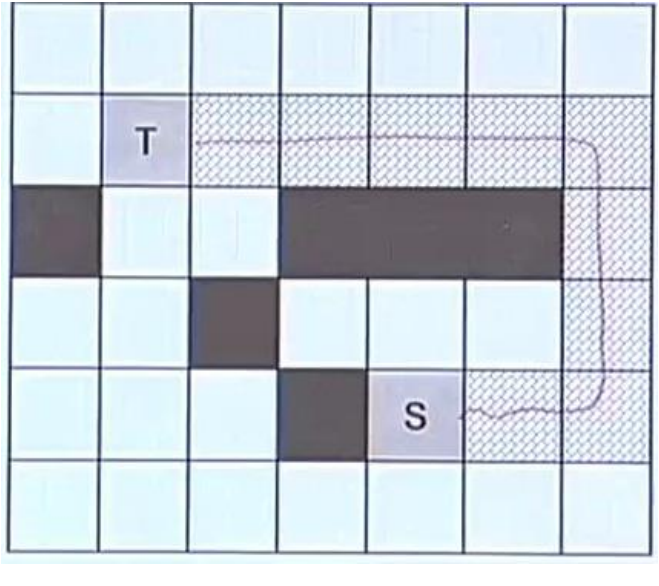
- Blocks that are placed on the surface.
- Some nets that are already laid out.

• Objective is to find out a path (sequence of grid cells) for connecting two points belonging to the same net.

• Two broad class of algorithms:

Maze routing algorithms.

Line search algorithms.



Problem Definition

• The general routing problem is defined as follows.

• Given:

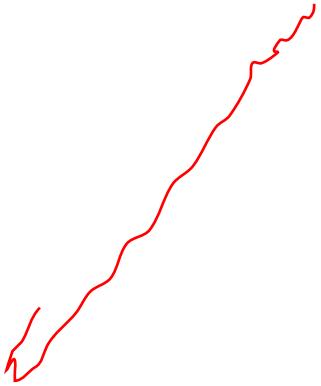
- A set of blocks with pins on the boundaries.
- A set of signal nets.
- Locations of blocks on the layout floor.

• Objective:

- Find suitable paths on the available layout space, on which wires are run to connect the desired set of pins.
- Minimize some given objective function, subject to given constraints.

• **Types of routing constraints:**

- Minimum width of routing wires.
- Minimum separation between adjacent wires.
- Number of routing layers available.
- Timing constraints.



Grid Routing Algorithms

1. Maze running algorithm
    - Lee's algorithm
    - Hadlock's algorithm
2. Line search algorithm
    - Mikami-Tabuchi's algorithm
    - Hightower's algorithm
3. Steiner tree algorithm

Maze Running Algorithms

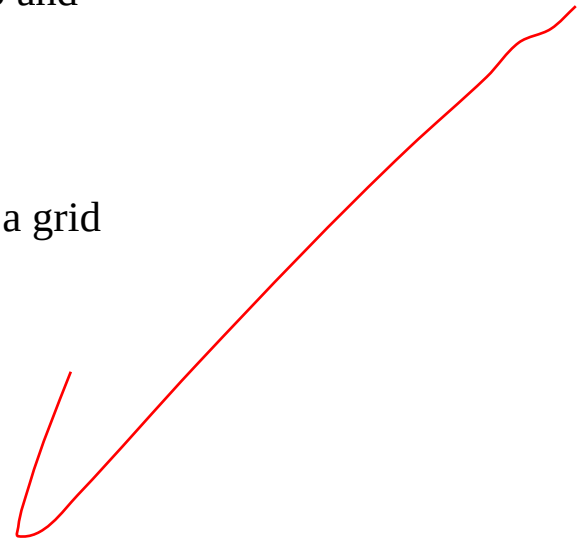
- The entire routing surface is represented by a 2-D array of grid cells.
    - All pins, wires and edges of bounding boxes that enclose the blocks are aligned with respect to the grid lines.
    - The segments on which wires run are also aligned.
    - The size of grid cells is appropriately defined.
        - Wires belonging to different nets can be routed through adjacent cells without violating the width and spacing rules.

• Maze routers connect a single pair of points at a time.

- By finding a sequence of adjacent cells from one point to the other.

Lee's Algorithm

- The most common maze routing algorithm.
- Characteristics:
  - If a path exists between a pair of points S and T, it is definitely found.
  - It always finds the shortest path.
  - Uses breadth-first search.
- Time and space complexities are O(h x w) for a grid of dimension h x w.



Phase 1 of Lee's Algorithm

• Wave propagation phase

- Iterative process.

- During step i, non-blocking grid cells at Manhattan distance of i from grid cell S are all labeled with i.

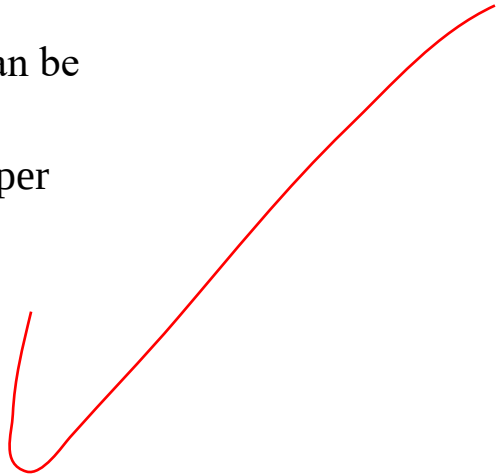
- Labeling continues until the target grid cell T is r marked in step L.

  • L is the length of the shortest path.

- The process fails if:

  • T is not reached and no new grid cells can be labeled during step i.

  • T is not reached and i equal M, some upper bound on the path length.



Phase 2 of Lee's Algorithm

• Retrace phase

- Systematically backtrack from the target cell T back towards the source cell S.

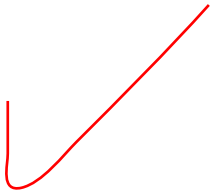
If T was reached during step i, then at least one grid cell adjacent to it will be labeled i-1, and so on.

- By tracing the numbered cells in descending order, we can reach S following the shortest path.

  • There is a choice of cells that can be made in general.

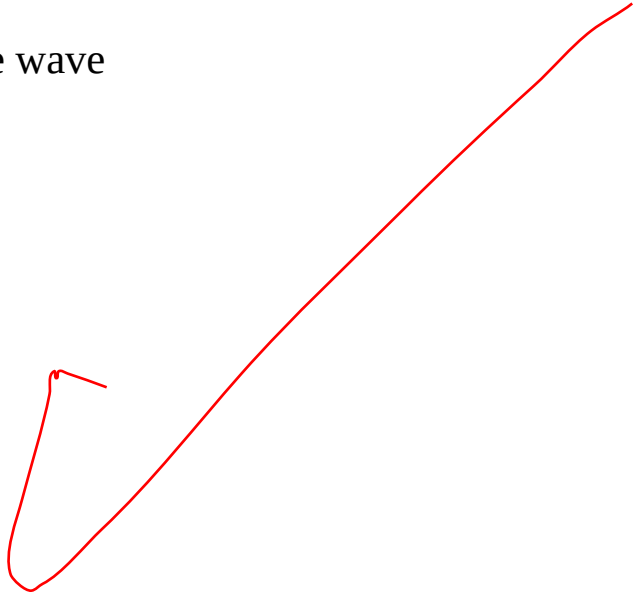
  • In practice, the rule of thumb is not to change the direction of retrace unless one has to do so.

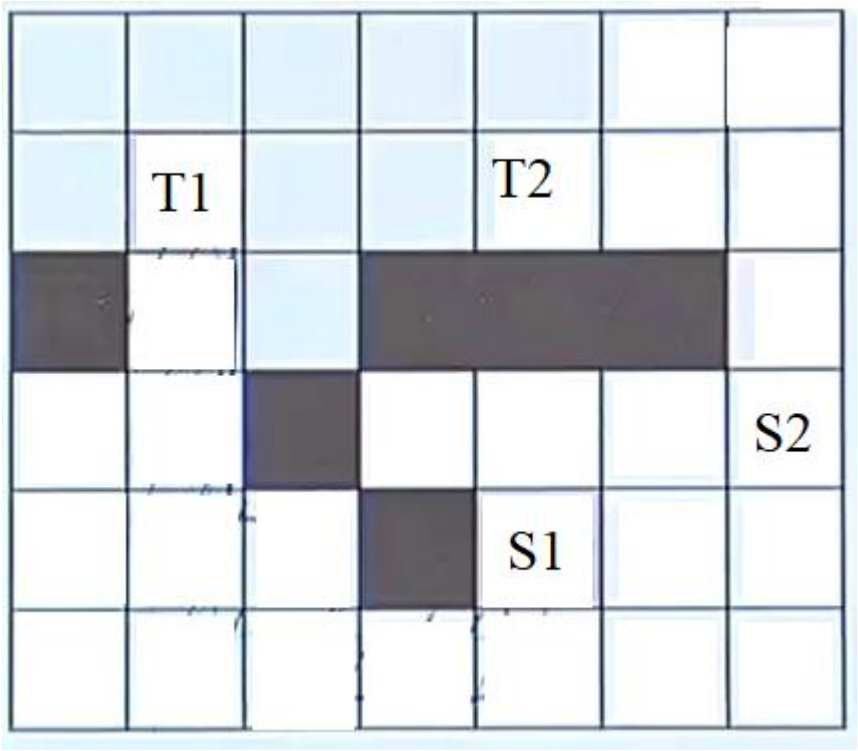
  • Minimizes number of bends.



Phase 3 of Lee's Algorithm

- Label clearance
  - All labeled cells except those corresponding to the path just found are cleared.
  - Search complexity is as involved as the wave propagation step itself.




T1
T2
S2
S1

• Memory Requirement

- Each cell needs to store a number between 1 and L, where L=2N is some bound on the maximum path length. Where N is number of cells in each row.
- One bit combination to denote empty cell.
- One bit combination to denote obstacles.

log2(L+2) bits per coll.

• Improvements:

-Instead of using the sequence 1,2,3,4,5.... For numbering the cells, the sequence 1,2,3,1,2,3,... is used.

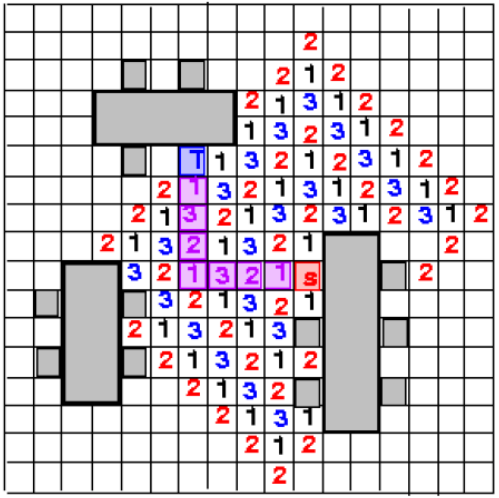
-For a cell, labels of predecessors and successors are different. So tracing back is easy.

log2(3+2) = 3 bits per cell.

-Use the sequence 0,0,1,1,0,0,1,1,…

• Predecessors and successors are again different.

log2(2+2) = 2 bits per cell.



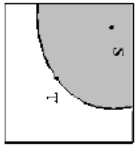
Sequence: 1, 2, 3, 1, 2, 3, ...

Reducing Running Time

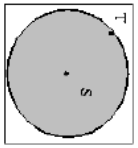
1- Starting point selection
   a. Choose the starting point as the one that is farthest from the center of the grid.

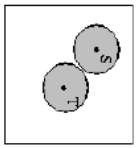
2- Double fan-out
   a. Propagate waves from both the source and the target cells.
   b. Labeling continues until the wavefronts touch.

3- Framing
   a. An artificial boundary is considered outside the terminal pairs to be connected.
   b. 10-20% larger than the smallest bounding box.

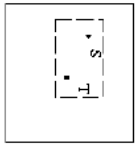


starting point selection





double fan–out



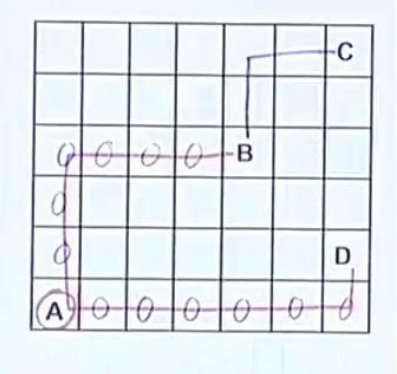
framing

Connecting Multi-point Nets

• A multi-pin net consists of three or more terminal points to be connected.

• Extension of Lee's algorithm:

- One of the terminals of the net is treated as source, and the rest as targets.
- A wave is propagated from the source until one of the targets is reached.
- All the cells in the determined path are next labeled as source cells, and the remaining unconnected terminals as targets.
- Process continues.



Hadlock's Algorithm

• Uses a new method for cell labeling called detour numbers.

- A goal directed search method.
- The detour number d(P) of a path P connecting two cells S and T is defined as the number of grid cells directed away from its target T.
- The length of the path P is given by:

$$len(P) = MD(S,T) + 2\,d(P)$$

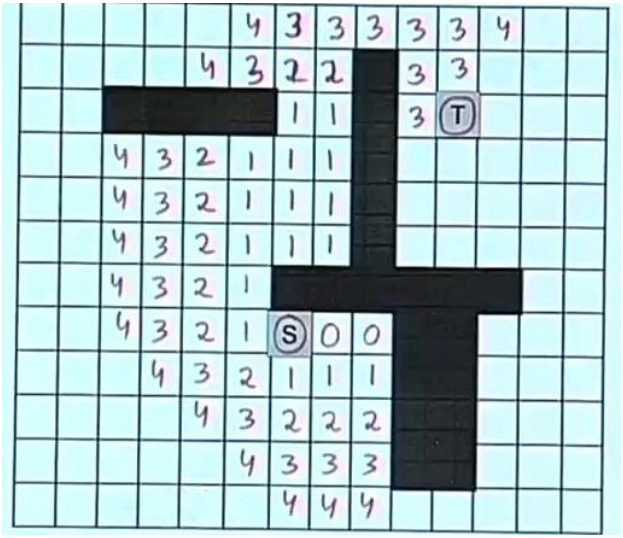
where MD (S,T) is the Manhattan distance between S and T.

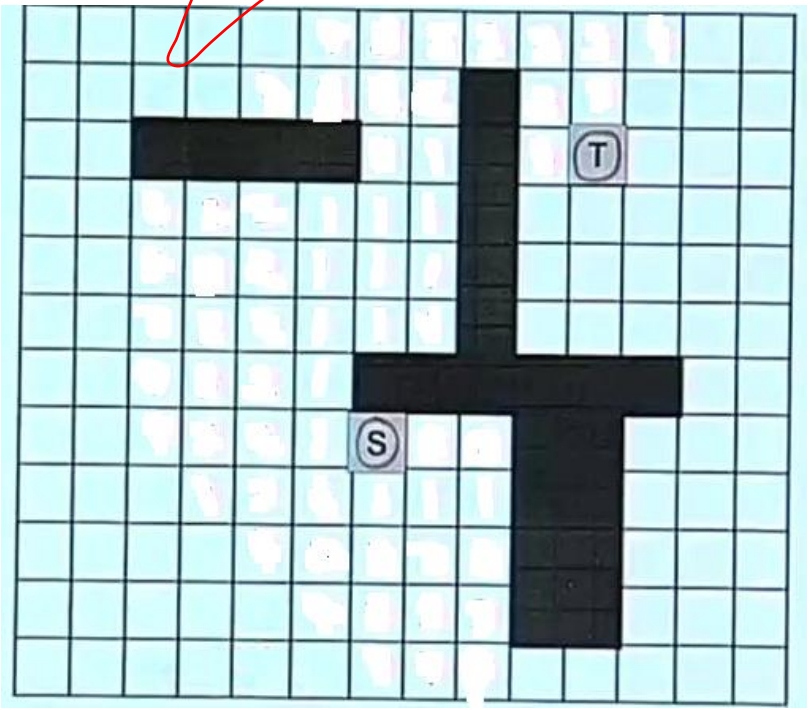
>> The cell filling phase of Lee's algorithm can be modified as follows:

- Fill a cell with the detour number with respect to a specified target T (not by its distance from source).

- Cells with smaller detour numbers are expanded with high priority.

• Path retracing is of course more complex, and requires some degree of searching.





- Advantages **Hadlock's Algorithm**:
  - Number of grid cells filled up is considerably less as compared to Lee's algorithm.
  - Running time for an NxN grid ranges from O(N) to $O(N^2)$.
    - Depends on the obstructions.
    - Also locations of S and T.

Line Search Algorithm

• In maze running algorithms, the time and space complexities are too high.

• An alternative approach is called line searching, which overcomes this drawback.

• **Basic idea**:

- Assume no obstacles for the time being.

- A vertical line drawn through S and a horizontal line passing though T will intersect.

  • Manhattan path between S and T.

- In the presence of obstacles, several such lines need to be drawn.

• Line search algorithms do not guarantee finding the optimal path.

  - May need several backtrackings.

- Running time and memory requirements are significantly less.
- Routing area and paths are represented by a set if line segments.
• Not as a matrix as in Lee's or Hadlock's algorithm.

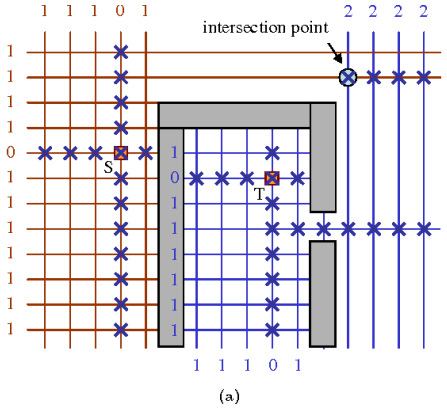
Mikami-Tabuchi's Algorithm

• Let S and T denote a pair of terminals to be connected.

• **Step 1**:

- Generate four lines (two horizontal and two vertical) passing through S and T.
- Extend these lines till they hit obstructions or the boundary of the layout.
- If a line generated from S intersects a line generated from T, then a connecting path is found.
- If they do not intersect, they are identified as trial lines of level zero.

• Stored in temporary storage for further processing.

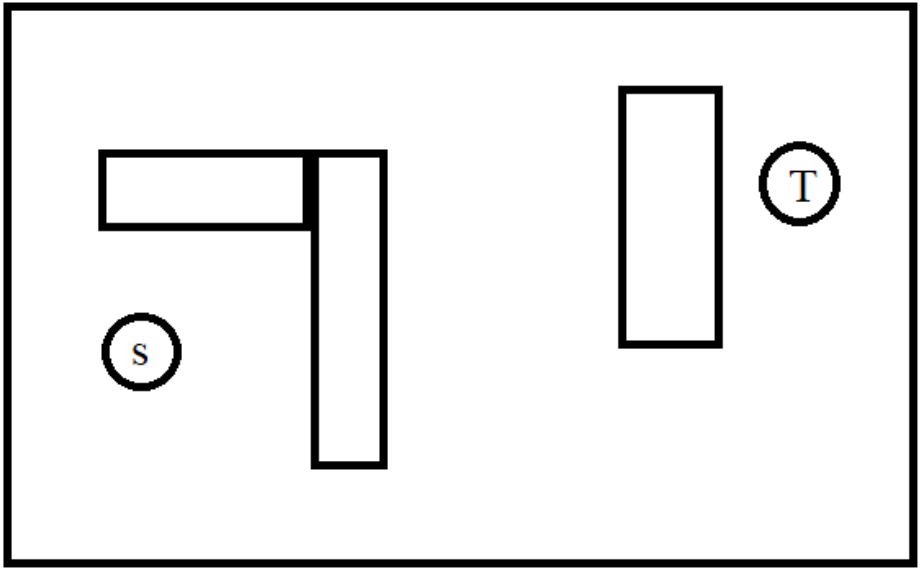


• **Step i of Iteration**:

- Pick up trial lines of level i, one at a time.
    - Along the trial line, all its grid points are traced.
    - Starting from these grid points, now trial lines (of level i+1) are generated perpendicular to the trial line of level i.

- If a trial line of level i+1 intersects a trial line (of any level) from the other terminal point, the connecting path can be found.
    - By backtracing from the intersection point to S and T.
    - Otherwise, all trial lines of level (i+1) are added to temporary storage, and the procedure repeated.

• The algorithm guarantees to find a path if it exists.


T
S

Hightower's Algorithm

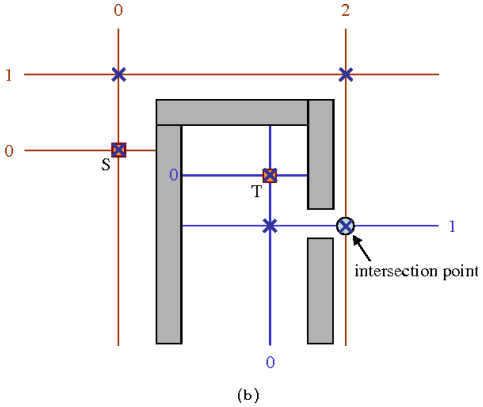
• Similar to Mikami•Tabuchi's algorithm.

- Instead of generating all line segments perpendicular to a trial line, consider only those lines that can be extended beyond the obstacle which blocked the preceding trial line.

-**Steps of the algorithm**:

- Pass a horizontal and a vertical line through source and target points (called first-level probes).
- If the source and the target lines meet, a path is found.
- Otherwise, pass a perpendicular line to the previous probe whenever it intersects an obstacle.

• Concept of escape point and escape line.



Global Routing:

Basic Idea

• The routing problem is typically solved using a two-step approach:

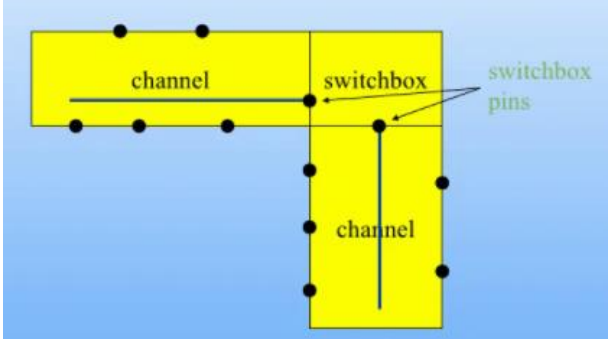
1. **Global Routing**
   - Define the routing regions.
   - Generate a tentative route for each net.
   - Each net is assigned to a set of routing regions.
   - Does not specify the actual layout of wires.
2. **Detailed Routing**
   - For each routing region, each net passing through that region is assigned particular routing tracks.
   - Actual layout of wires gots fixed.
   - Associated sub-problems: channel routing and switchbox routing.

Routing Regions

• Regions through which interconnecting wires are laid out.

• How to define these regions?

- Partition the routing area into a set of non-intersecting rectangular regions.
- Types of routing regions:
  • Horizontal channel: parallel to the x-axis with pins at their top and bottom boundaries.
  • Vertical channel: parallel to the y-axis with pins at their left and right boundaries.
  • Switchbox: rectangular regions with pins on all four sides.

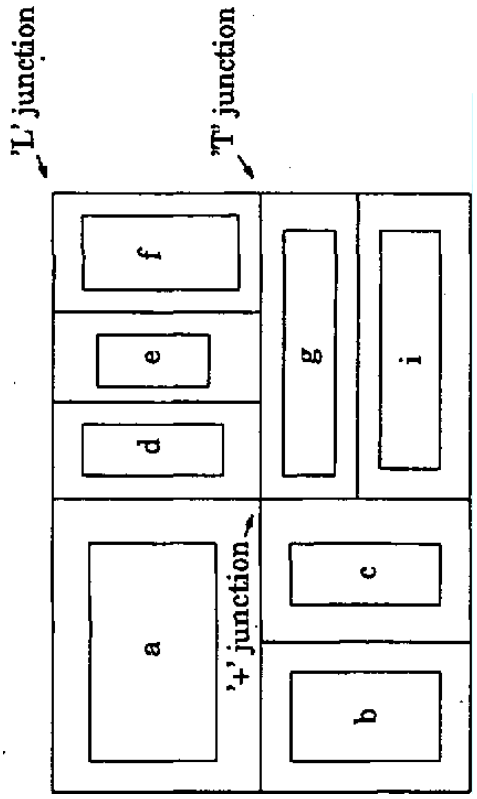


- **Points to note**:

  - Identification of routing regions is a crucial first step to global routing.
  - Routing regions often do not have pre-fixed capacities.
  - The order in which the routing regions are considered during detailed routing plays a vital part in determining overall routing quality.

• Three types of channel junctions may occur:

- L-type:
  - • Occurs at the corners of the layout surface.
  - • Ordering is not important (hiring detailed routing.
  - • Can be routed using channel routers.
- T-type:
  - • The leg of the "T" must be routed before the shoulder.
  - • Can be routed using channel routers.
- +-type:
  - • More complex and requires switchbox routers.
  - • Advantageous to convert +-junctions to T-junctions.



Design Style Specific Issues

• Full Custom

- The problem formulation is similar to the general formulation as discussed.
    - All the types of routing regions and channels junctions can occur.
- Since channels can be expanded, some violations of capacity constraints are allowed.
- Major violation in constraints are, however, not allowed.
    - May need significant changes in placement.

- **Standard Cell**

- At the end of the placement phase
    - Location of each cell In a row is fixed.
    - Capacity and location of each feed-through is fixed.
    - Feed-throughs have predetermined capacity.
- Only horizontal channels exist.
    - Channel heights are not fixed.
- Insufficient feed-throughs may lead to failure.
- Over-the-cell routing can reduce channel height, and change the global routing problem.

- **Gate Array**

  - The size and location of cells are fixed.
  - Routing channels & their capacities are also fixed.
  -- Primary objective of global routing is to guarantee routability.
  - Secondary objective may be to minimize critical path delay.

Ninevah University

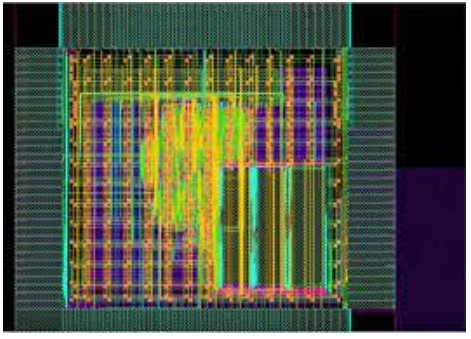
Collage of Electronics Engineering

Course:

Electronic Design Automation

Lecturer: H. M. Hussein

EDA09: Physical Design Automation – Placement



Simulated Evolution / Genetic Algorithm

• The algorithm starts with an initial set of placement configurations.

- Called the population.

• The process is iterative, where each iteration is called a **generation**.

- The individuals of a population are evaluated to measure their goodness.

• To move from one generation to the next, three genetic operators are used:

1. Crossover
2. Mutation
3. Selection

CROSSOVER Operator

• Choose a random cut point.

• Generate offsprings by combining the left segment of one parent with the right segment of the other.

— Some blocks may get repeated, while some others may get deleted.

— Various ways to deal with this problem.

• Number of times the "crossover" operator is applied is controlled by crossover rate.

MUTATION Operator

• Causes incremental random changes to an offspring produced by crossover.
• Most common is pairwise exchange.
• Number of times this is done is controlled by mutation rate.

SELECT Operator

• Select members for crossover based on their fitness value.
    - Obtained by evaluating a cost function.
• Higher the fitness value of a solution, higher will be the probability for selection for crossover.

Force Directed Placement

• Explores the similarity between placement problem and classical mechanics problem of a system of bodies attached to springs.

• The blocks connected to each other by nets are supposed to exert attractive forces on each other.

- Magnitude of this force is directly proportional to the distance between the blocks.
  - Analogous to Hooke's law in mechanics.
- Final configuration is one in which the system achieves equilibrium.

• A cell i connected to several cells j experiences a total force Fi = $\sum_j (w_{ij} * d_{ij})$ where wij is the weight of connection between i and j dij is the distance between I and J.

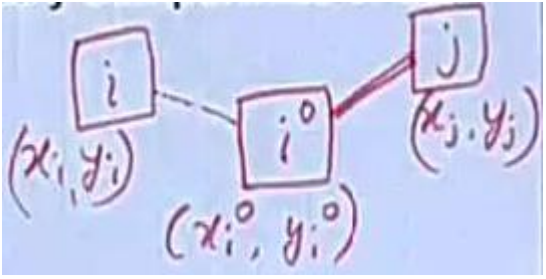
• If the cell i is free to move, it would do so in the direction of force F, until the resultant force on it is zero.

• When all cells move to their zero-force target locations, the total wire length is minimized.

• For cell i, if (xi^0, yi^0) represents the zero-force target location, by equating the x- and y-components of the force to zero, we get

$$\sum_i \left(\left(w_{ij} * \left(x_j - xi^0 \right) \right) \right) = 0$$

$$\sum_i \left(\left(w_{ij} * \left(y_j - yi^0 \right) \right) \right) = 0$$



• Solving for xi^0 and yi^0, we get

$$xi^0 = \left(\sum_j (w_{ij} * x_j) \right) / \left(\sum_j w_{ij} \right)$$

$$yi^0 = \left(\sum_j (w_{ij} * y_j) \right) / \left(\sum_j w_{ij} \right)$$

• Care are should be taken to avoid assigning more than one cell to the same location.

Breuer's Algorithm

• Partitioning technique used to generate placement.

• The given circuit is repeatedly partitioned into two sub-circuits.

— At each level of partitioning, the available layout area is partitioned into horizontal and vertical subsections alternately.

— Each of the sub-circuits is assigned to a subsection.

— Process continues till each sub-circuit consists of a single gate and has a unique place on the layout area.

• Different sequences of cut lines used:

1. Cut Oriented Min-Cut Placement
2. Quadrature Placement
3. Bisection Placement
4. Slice Bisection Placement

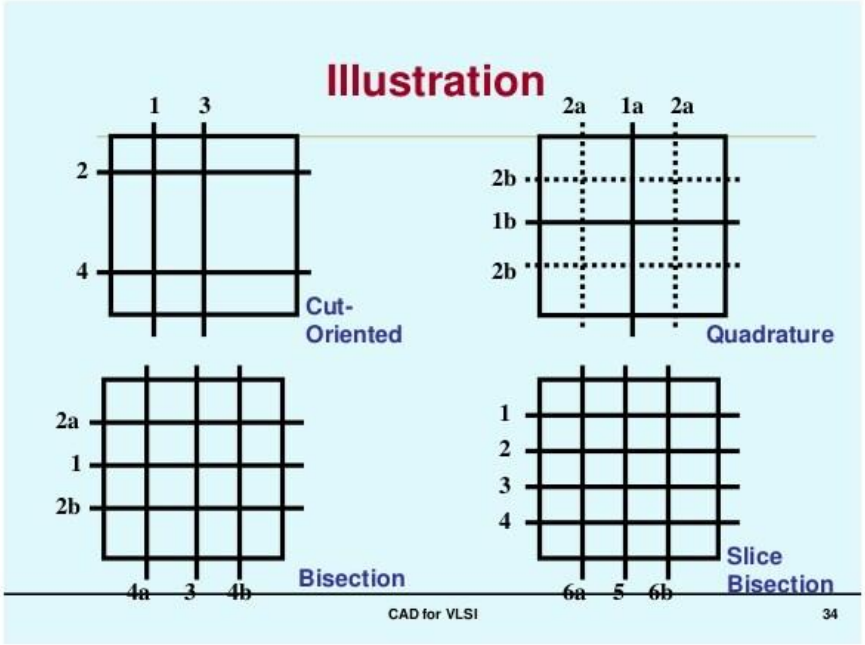


Illustration
1
3
2
4
Cut-
Oriented
2a
1a
2a
2b
1b
2b
Quadrature
2a
1
2b
4a
3
4b
Bisection
1
2
3
4
6a
5
6b
Slice
Bisection
CAD for VLSI
34

Cluster Growth

• In this constructive placement algorithm, bottom-up approach is used.

• Blocks are placed sequentially in a partially completed layout.

- The first block (seed) is usually placed by the user.
- Other blocks are selected and placed one by one.

• Selection of blocks is usually based on connectivity with placed blocks.

• Layouts produced are not usually good.

- Does not take into account the interconnections and other circuit features.

• Useful for generating initial placements.

- For iterative placement algorithms.

```
Algorithm Cluster_Growth
begin
    B = set of blocks to be placed;
    Select a seed block S from B;
    Place S in the layout;
    B = B – S;
    while (B ≠ ϕ) do
      begin
        Select a block X from B;
        Place X in the layout;
        B = B – X;
      end;
end
```

Ninevah University

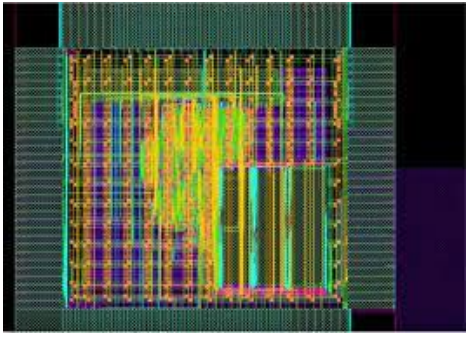
Collage of Electronics Engineering

Course:

Electronic Design Automation

Lecturer: H. M. Hussein

EDA09: Physical Design Automation – Pin Assignment



Pin Assignment

Introduction

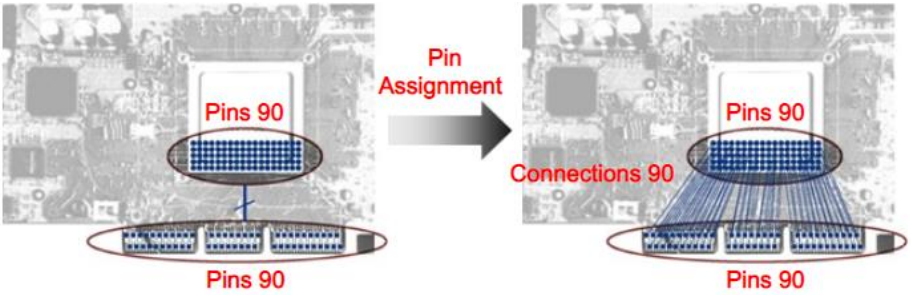
• The purpose is to define the signal that each pin will receive.

• It can be done:

- During floorplanning
- During placement
- After placement is fixed

> For undesigned blocks, a good assignment of pins improves placement.

• If the blocks are already designed, still some pins can be exchanged.



Pin Assignment

Input:

- A placement of blocks.
- Number of pins on each block, possibly an ordering.
- A netlist.

• Requirements:

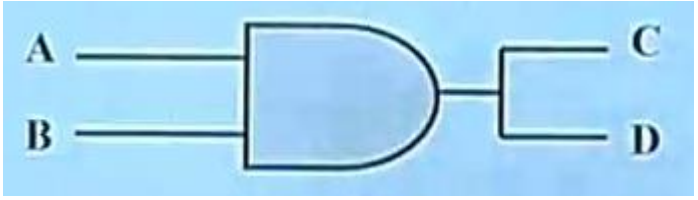
- To determine the pin locations on the blocks.

Objectives:

- To minimize net-length.

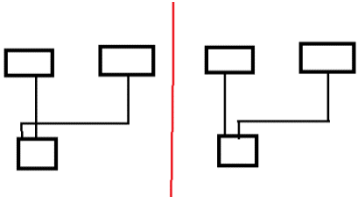
- Functionally equivalent pins:
  - Exchanging the signals does not affect the circuit.
- Equipotential pins:

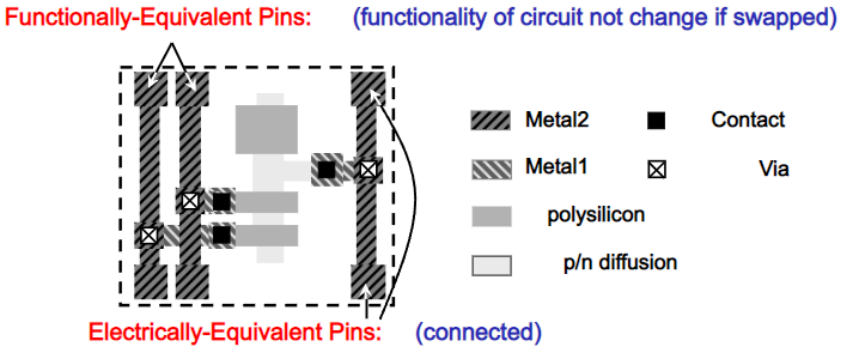
  Both are internally connected and represent the same net.



A, B :: functionally equivalent

C, D :: equipotential





Problem Formulation

• Purpose is to optimize the assignment of nets within a functionally equivalent (or equipotential) pin groups.

• Objective:

- To reduce congestion or reduce the number of crossovers.

Design Style Specific Issues

• Full Custom

- Two types of pin assignment problems:

a) During floorplanning, the pin location along the block boundary can he changed as the block is assigned a shape. <REDUCES CONGESTION>

b) During placement, simply assign nets to pins.

• Standard Cell

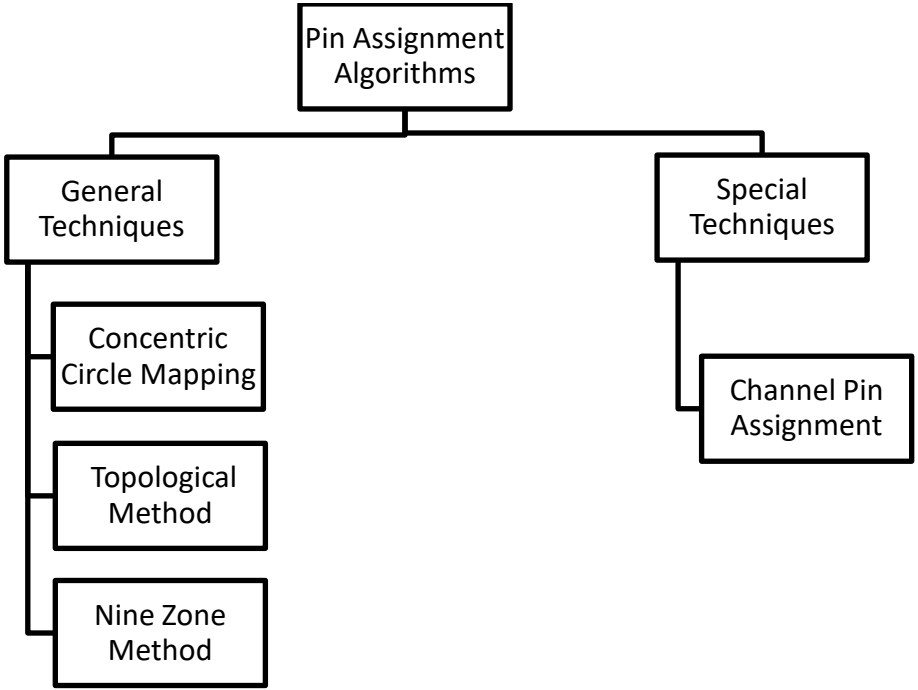
Essentially two things to be done:

a) Permuting net assignment for functionally equivalent pins.

b) Changing equipotential pins for a net.

• Gate Array - Same as that for standard cells.

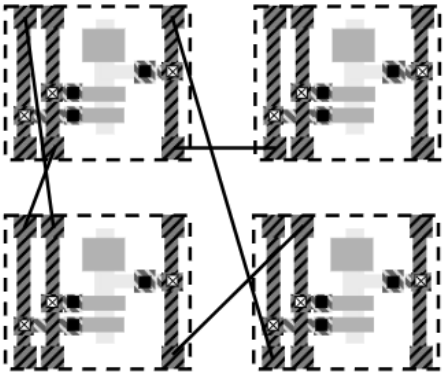
Classification of Algorithms



Concentric Circle Mapping

• Uses two concentric circles to planarize the interconnections.

- Pins on the block being considered are shown as points on the inner circle.
- Interconnections to be made with other blocks are shown as points on the outer circle.

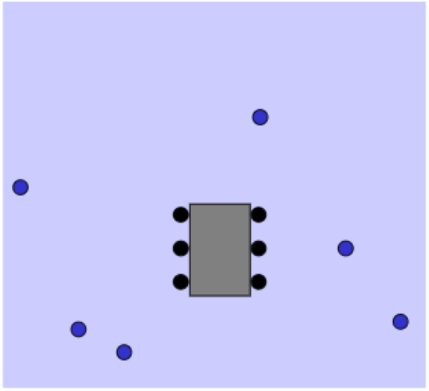


• Divides the problem into two parts:

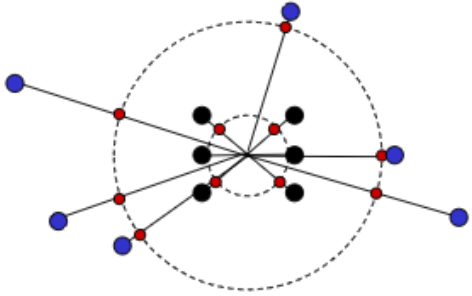
a) Assignment of pins to points of the two circles.

b) Mapping the points on the inner circle to those on the outer circle.

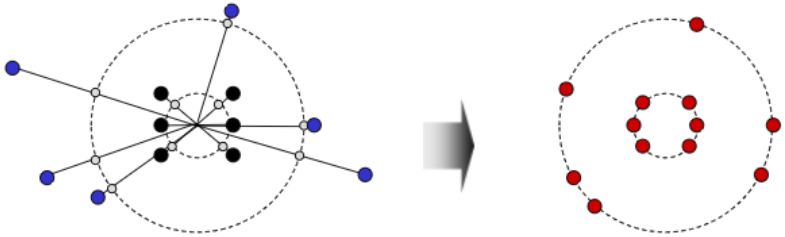
Given: Two sets of pins



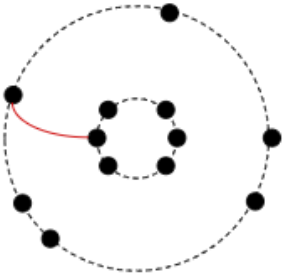
Determine the points (2)



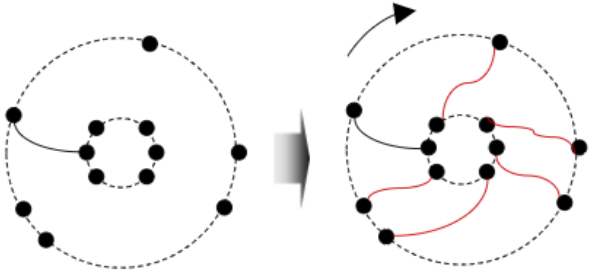
Determine the points (2)



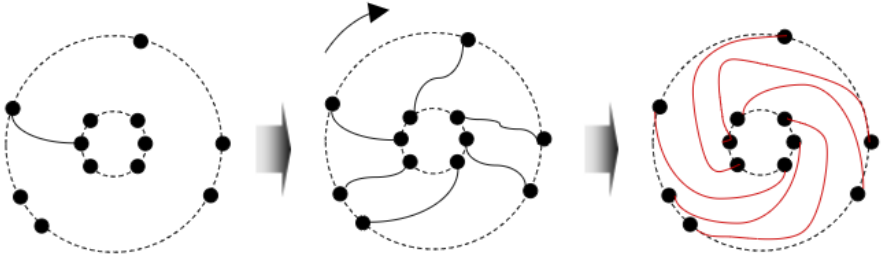
Determine initial mapping (arbitrarily) (3)



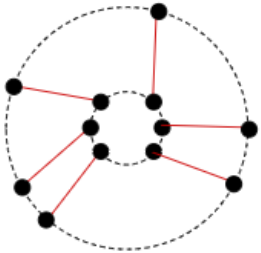
Determine initial mapping and (4) optimize the (3) mapping (complete rotation)



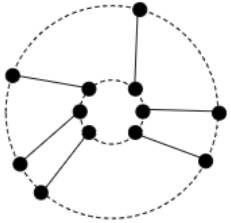
Determine initial mapping and (4) optimize the (3) mapping (complete rotation)



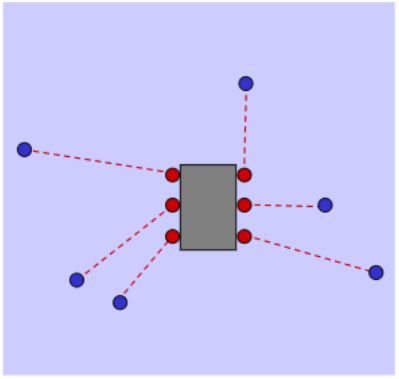
Best mapping (shortest Euclidean distance) (4)



Best mapping (4)



Final pin assignment



Topological Pin Assignment

• Similar to concentric circle mapping.

• Easier to complete pin assignment.

- When there is interference from other components and barriers.
- For nets connected to more than two pins.

• If a net has been assigned to more than two pins, then the pin closest to the center of the primary component is chosen.

• Pins of primary component are mapped onto a circle as before.

• Beginning at the bottom of the circle, and moving clockwise, the pins are assigned to nets.

Integrated Approach

• Better understanding of the different stages in physical design automation over the years.

- Attempts are being made to merge some steps of the design cycle.
- For example, floorplanning and placement are considered together.
- Sometimes, placement and routing stages can also be combined together.

• Still a problem of research.

Placement

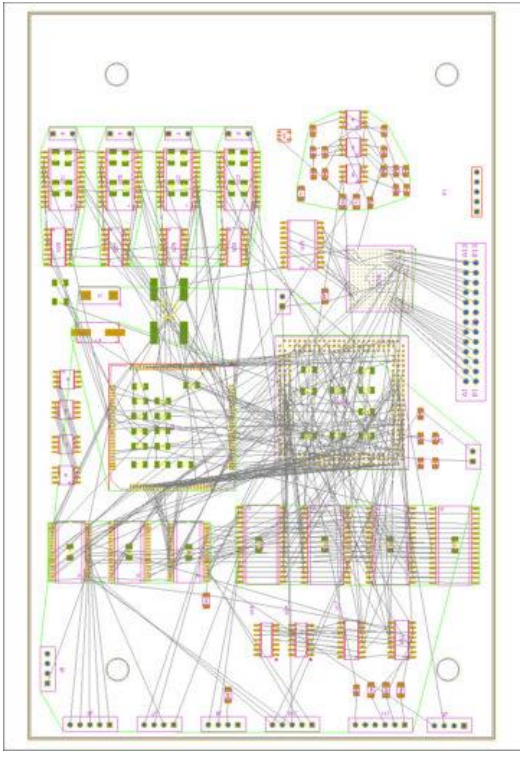
Introduction

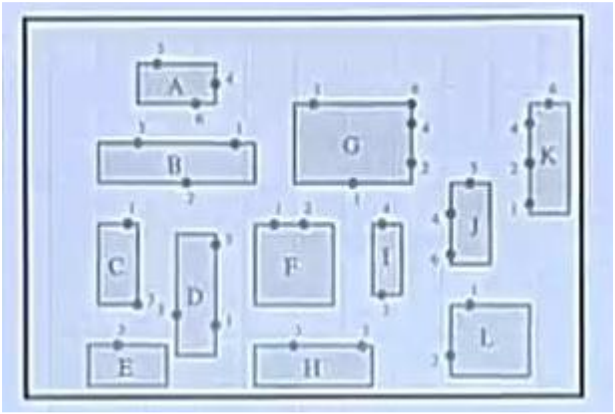
• A very important step in physical design cycle.

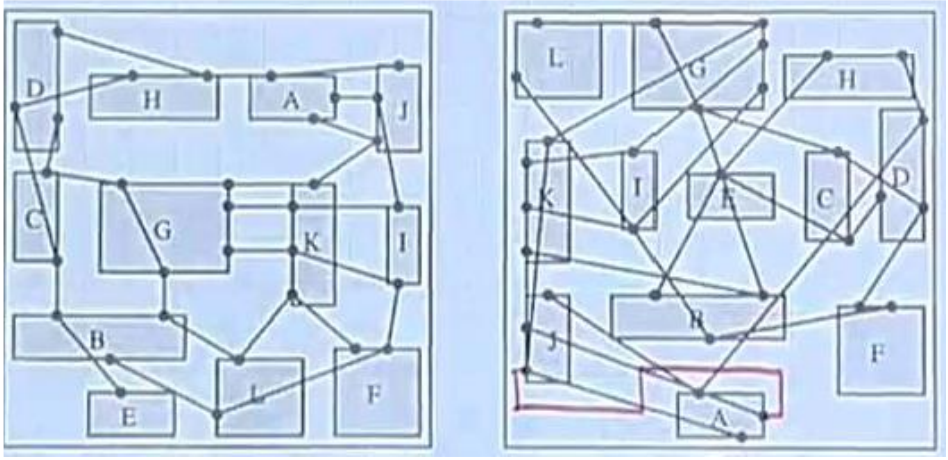
- A poor placement requires larger area.
- Also results in performance degradation.

• It is the process of arranging a set of modules on the layout surface.

- Each module has fixed shape and fixed terminal locations.
- A subset of modules may have pre-assigned positions (e.g., I/O pads).




A
B
G
K
J
C
D
F
I
L
E
H


D
H
A
J
C
G
K
I
B
E
L
F
L
G
H
K
I
E
C
D
B
J
F
A

The Placement Problem

• **Inputs**:

- A set of modules with
    - well-defined shapes
    - fixed locations of pins.
- A netlist.

• **Requirements**:

- Find locations for each module so that no two modules overlap.
- The placement is routable.

• **Objectives**:

- Minimize layout area.
- Reduce the length of critical nets.
- Completion of rotating.

Placement Problems at Different Levels

1. System-level placement
   - Place all the PCBs together such that
   • Area occupied is minimum
   • Heat dissipation is within limits.

2. Board-level placement
   - All the chips have to be placed on a PCB.
      • Area is fixed
      • All modules of rectangular shape
   - Objective is to
   • Minimize the number of routing layers
   • Meet system performance requirements.

3. Chip-level placement

- Normally, floorplanning / placement carried out along with pin assignment.
- Limited number of routing layers (2 to 4).
    - Bad placements may be unroutable.
    - Can be detected only later (during routing)
    - Costly delays in design cycle.
- Minimization of area.

Problem Formulation

• Notations:

B1,B2,..., Bn: modules/blocks to be placed

Wi,hi,: width and height of Bi 1<i<n

N={N1,N2,….Nm} : set of nets (i.e. the netlist)

Q={Q1,Q2,...Qk} : rectangular empty spaces for routing

Li : estimated length of net Ni 1<i<m

- **The problem**

Find rectangular regions R={R1,R21...Rn} for each of the blocks such that

- Block Bi can be placed in region Ri.
- No two rectangles overlap, Ri∩Rj = Φ.
- Placement is routable (Q is sufficient to route all nets).
- Total area of rectangle bounding R and Q is minimized.
- Total wire length $\sum Li$ is minimized.

For high performance circuits, max{Li| i=1,2,...,m} is minimized

. General problem is NP-complete.

. Algorithms used are heuristic in nature.

Interconnection Topologies

• The actual wiring paths are not known during placement.

-For making an estimation, a placement algorithm needs to model the topology of the interconnection nets.

- An interconnection graph structure is used.
- Vertices are terminals, and edges are interconnections.

Estimation of Wirelength

• The speed and quality of estimation has a drastic effect on the performance of placement algorithms.

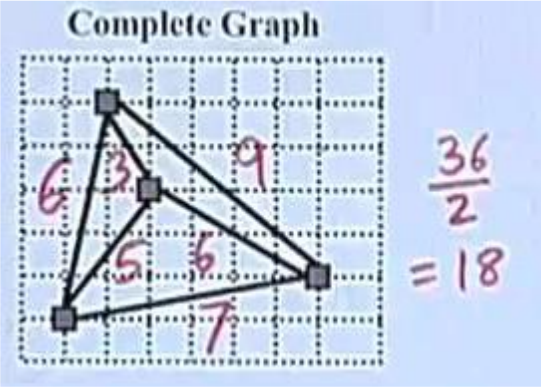
- For 2-terminal nets, we can use Manhattan distance as an estimate.
- If the end co-ordinates are (x1,y1) and (x2,y2), then the wire length L= | xl - x2 |+ | y1 - y2 |

• How to estimate length of multi-terminal nets?

Modeling of Multi-terminal Nets

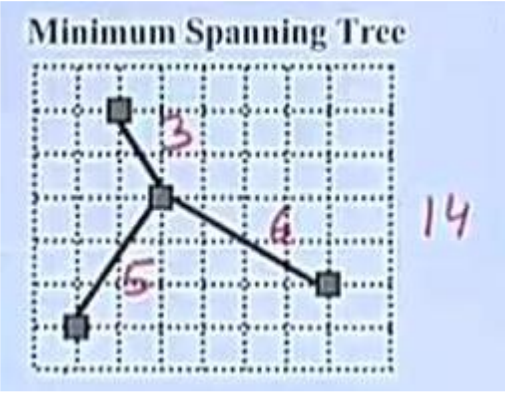
1- Complete Graph

- ${}^{n}C_2 = n(n-1)/2$ edges for a n-pin net.
- A tree has (n-1) edges which is 2/n times the number of edges of the complete graph.
- Length is estimated as 2/n times the sum of the edge weights.



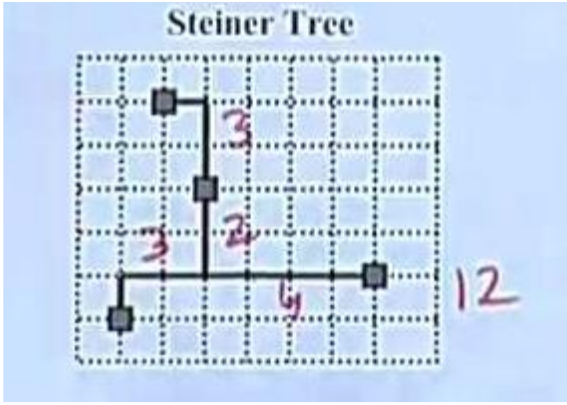
2- Minimum Spanning Tree

- Commonly used structure.
- Branching allowed only at pin locations.
- Easy to compute.



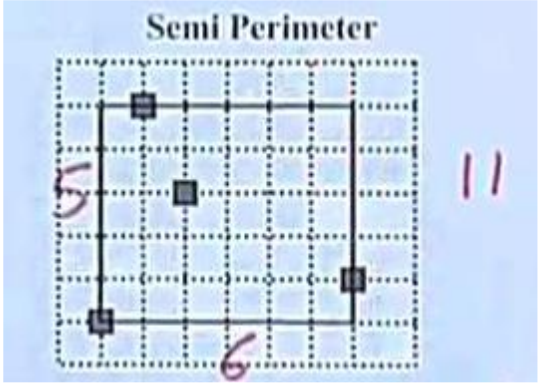
3- Rectangular Steiner Tree

- A Steiner tree is the shortest route for connecting a set of pins.
- A wire can branch from any point along its length.
- Problem of finding Steiner tree is NP-complete.



4- Semi Perimeter

- Efficient and most widely used.
- Finds the smallest bounding rectangle that encloses all the pins of the net to be connected.
- Estimated wire length is half the perimeter of this rectangle.
- Always underestimates the wire length for congested nets.



Design Style Specific Issues

• Full Custom

- Placing a number of blocks of various shapes and sizes within a rectangular region.
- Irregularity of block shapes may lead to unused areas.

• Standard Cell

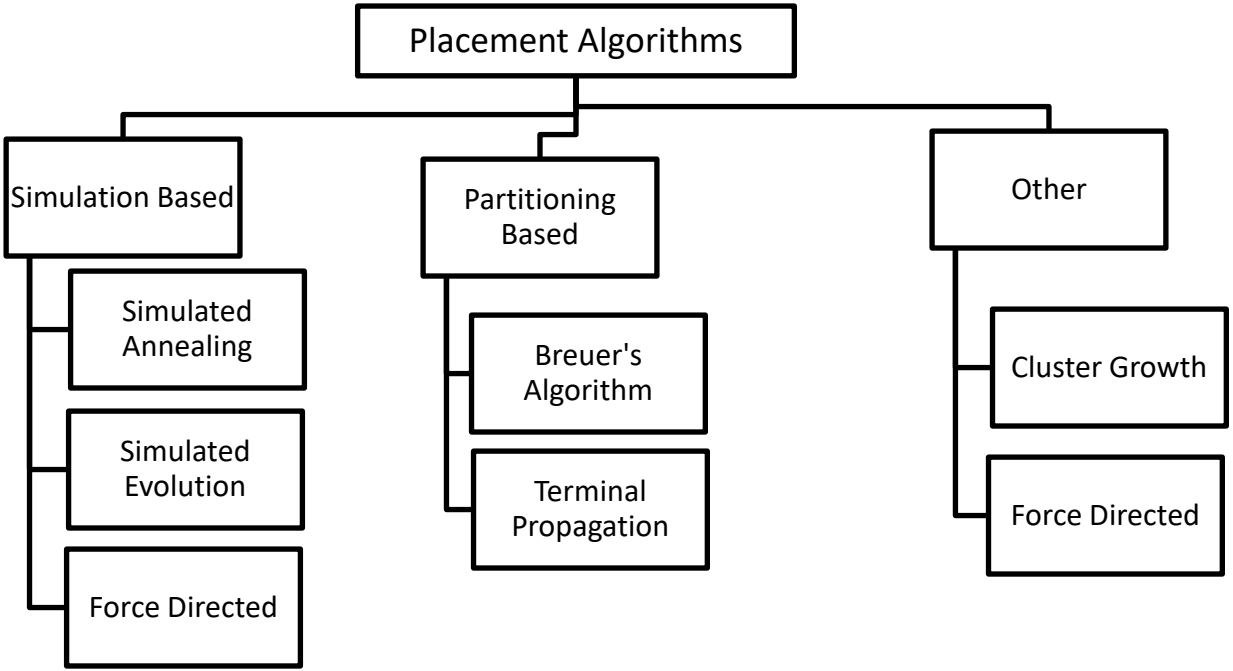
— Minimization of the layout area means:

- Minimize sum of channel heights.
- Minimize width of the widest row.
- All rows should have equal width.

— Over-the-cell routing leads to almost "channel-less" standard cell designs.

- Gate Arrays
  - The problem of partitioning and placement are the same in this design style.
  - For FPGA's, the partitioned sub-circuit may be a complex netlist.
    - Map the netlist to one or more basic blocks (placement).

Classification of Placement Algorithms



Simulated Annealing

• Simulation of the annealing process in metals or glass.

- Avoids getting trapped in local minima.
- Starts with an initial placement.
- Incremental improvements by exchanging blocks, displacing a block, etc.
- Moves which decrease cost are always accepted.
- Moves which increase cost are accepted with a probability that decreases with the number of iterations.

• Timberwolf is one of the most successful placement algorithms based on simulated annealing.

Simulated Annealing Algorithm

```
Algorithm SA_Placement
begin
        T = initial temperature;
        P = initial_placement;
        while ( T > final_temperature) do
                while (no_of_trIals_at_eachtemp not yet
                completed) do
                        new P = PERTURB (P);
                        delta_C = COST (new_P) - COST (P);
                        if (delta_C < 0) then
                                P = new_P;
                        else if (random(0,1) > exp(delta_C/T)) then
                                P = new_P;
                T = SCHEDULE (T); /** Decrease temperature
end
```

TimberWolf

• One of the most successfulplacement algorithms.

- Developed by Sechen and Sangiovanni-Vincentelli. •

Parameters used:

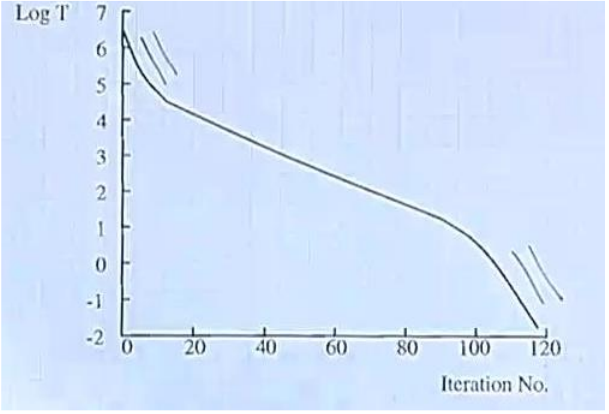
- Initial_temperature = 4,000,000
- Final_temperature = 0.1
- SCHEDULE(T) = alpha(T) x T

• alpha(T) specifies the cooling rate which depends on the current temperature.

• a(T) is 0.8 when the cooling process just starts.

• a(T) is 0.95 in the medium range of temperature.

• a(T) is 0.8 again when temperature is low.



The PERTURB Function

• New configuration is generated by making a weighted random selection from one of the following:

a) The displacement of a block to a new location.

b) The interchange of locations between two blocks.

c) An orientation change for a block.

- Mirror image of the block's x-coordinate.
- Used only when a new configuration generated using alternative (a) is rejected.

The COST Function

• The cost of a solution is computed as:

COST = cost1 + cost2 + cost3

where

cost1 : weighted sum of estimated length of all nets

cost2 : penalty cost for overlapping

cost3 : penalty cost for uneven length among standard cell rows.

- Overlap is not allowed in placement.
- Computationally complex to remove all overlaps.
- More efficient to allow overlaps during intermediate placements.

• Cost function (cost2) penalizes the overlapping.

Ninevah University

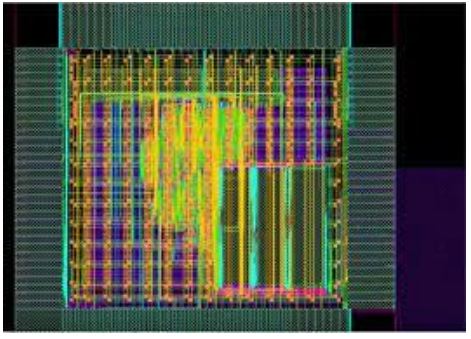
Collage of Electronics Engineering

Course:

Electronic Design Automation

Lecturer: H. M. Hussein

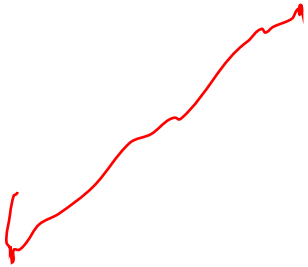
EDA07: Physical Design Automation - Floor-Planning



Floor-Planning

Problem Definition

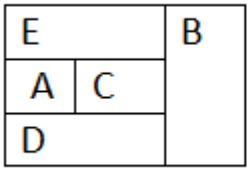
- Input:
  - A set of blocks, both fixed and flexible.
    - Area of the block $A_i = w_i \times h_i$
    - Constraint on the Shape of the block (rigid/flexible)
  - Pin locations of fixed blocks.
  - A netlist.
- Requirements:
  - Find locations for each block so that no two blocks overlap.
  - Determine shapes of flexible blocks.
- Objectives: - Minimize area.
  - Reduce wire-length for critical nets.

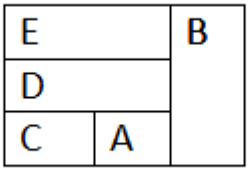


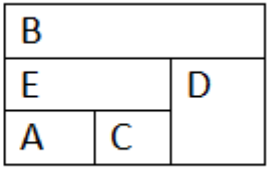
Example: Rigid Blocks

Module	Width	Height
A	1	1
B	1	3
C	1	1
D	1	2
E	2	1

Feasible Floor-plans







Design Style Specific Issues

- Full Custom
  - All the steps required for general cells.
- Standard Cell
  - Dimensions of all cells are fixed.
  - Floorplanning problem is simply the placement problem.
  - For large netlists, two steps:
    - First do global partitioning.
    - Placement for individual regions next.
- Gate Array
  - Floorplanning problem same as placement problem.

Estimating Cost of a Floorplan

• The number of feasible solutions of a floorplanning problem is very large.

— Finding the best solution is NP-hard.

• Several criteria used to measure the quality of floor-plans:

a) Minimize area

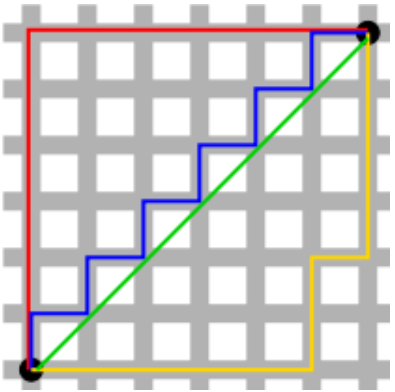
b) Minimize total length of wire

c) Maximize rout-ability

d) Minimize delays

e) Any combination of above.

- How to determine area?
  - Not difficult.
  - Can be easily estimated because the dimensions of each block is known.
  - Area A computed for each candidate floor-plan.
- How to determine wire length?
  - A coarse measure is used.
  - Based on a model where all I/O pins of the blocks are merged and assumed to reside at its center.
  - Overall wiring length L =$\sum_{i,j}(c_{ij} * d_{ij})$ where c_{ij} is the connectivity between blocks i and j d_{ij} is the Manhattan distances between the centers of rectangles of blocks i and j.



- Typical cost function used:

  Cost = w1 * A + w2 * L

  where w1 and w2 are user-specified parameters.

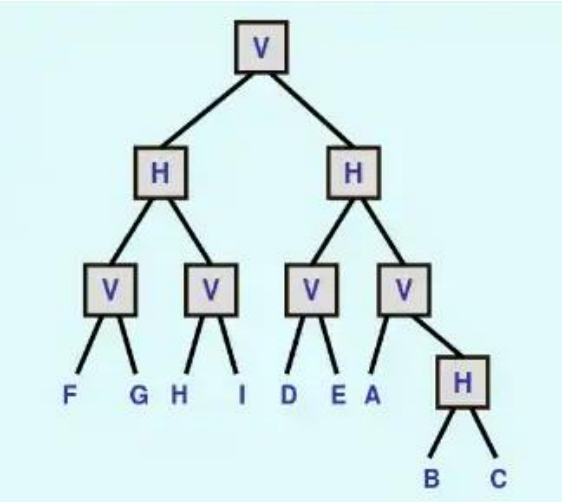
Slicing Structure

- Definition

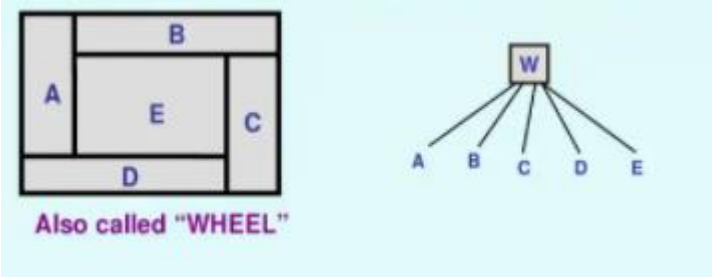
  - A rectangular dissection that can be obtained by repeatedly splitting rectangles by horizontal and vertical lines into smaller rectangles.

- Slicing Tree

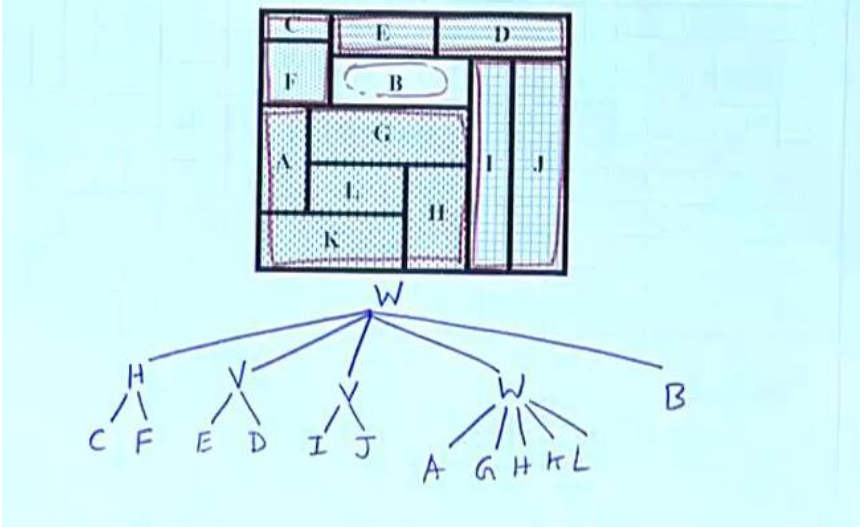
  - A binary tree that models a slicing structure.
  - Each node represents a vertical cut line (V), or a horizontal cut line (H).
    - A third kind of node called Wheel (W) appears for non-sliceable floor-plans (discussed later).
  - Each leaf is a basic block (rectangle).



Non-Slicing Floor plan



Hierarchical Floor plan:



Floor-planning Algorithms

- Several broad classes of algorithms:
    1. Integer programming based
    2. Rectangular dual graph based
    3. Hierarchical tree based
    4. Simulated annealing based
    5. Other variations

Linear programming

- **Linear programming** (LP, or **linear optimization**) is a mathematical method for determining a way to achieve the best outcome (such as maximum profit or lowest cost) in a given mathematical model for some list of requirements represented as linear relationships.
- linear programming is a technique for the optimization of a linear objective function, subject to constraints.

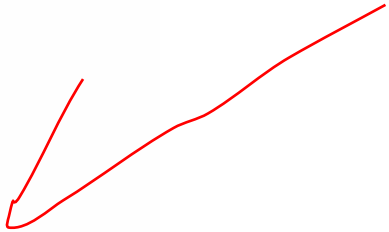
$$\begin{aligned} \text{maximize} \quad & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} \quad & A\mathbf{x} \leq \mathbf{b} \\ \text{and} \quad & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

How is ILP applied to floor planning?

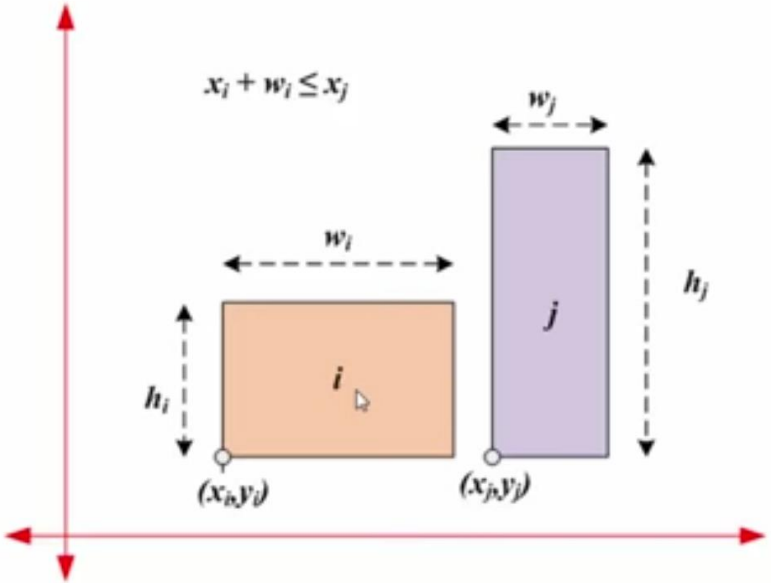
- The constraints specifying a feasible floor plan are described by a set of mathematical equations
- Solved by commercial ILP solvers.
- "An analytical approach to floor plan design and optimization" by Suphachai sutanthavibul, Eugene Shragowitz and J.B.Rosen, IEEE Transaction on CAD, 1991

Problem formulation

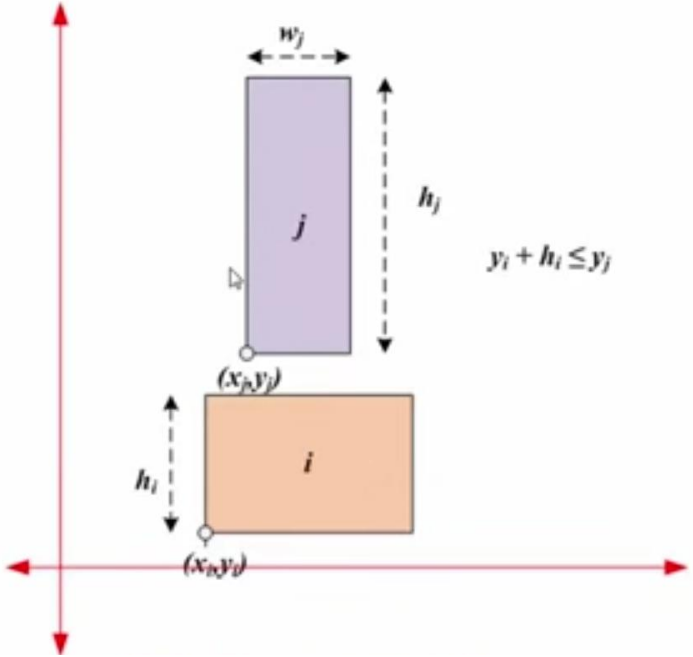
- Let S be the set of modules.
- Modules can be rigid or flexible.
- Assume rigid modules.
- Original algorithm can handle flexible modules



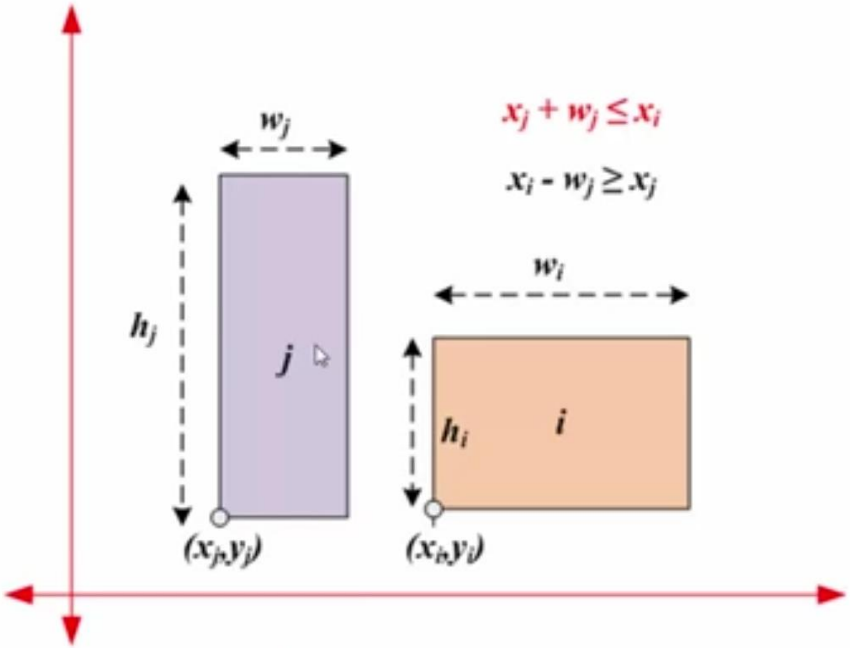
Module i is to the left of j



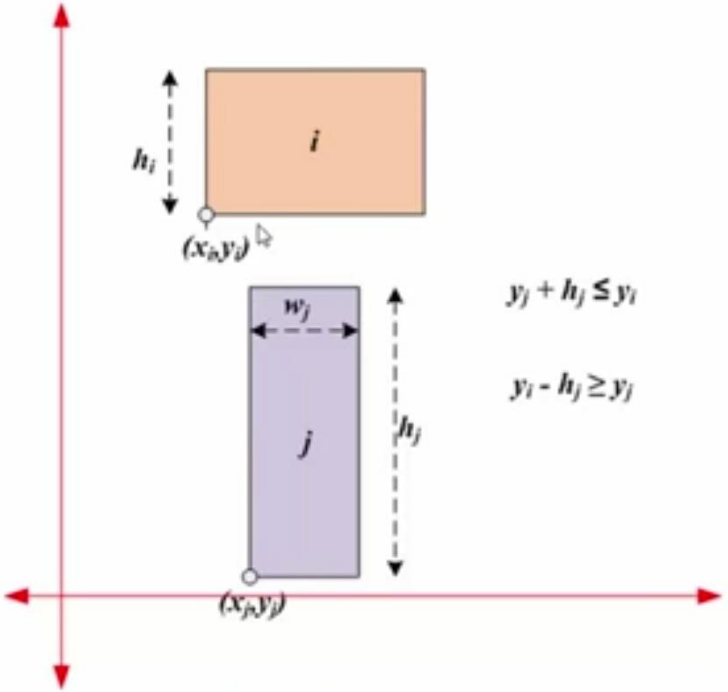
Module i is below j



Module i is to the right of j



Module i is above j



$$y_j + h_j \leq y_i$$

$$y_i - h_j \geq y_j$$

Relative positions of 2 modules i and j governed by 4 equations

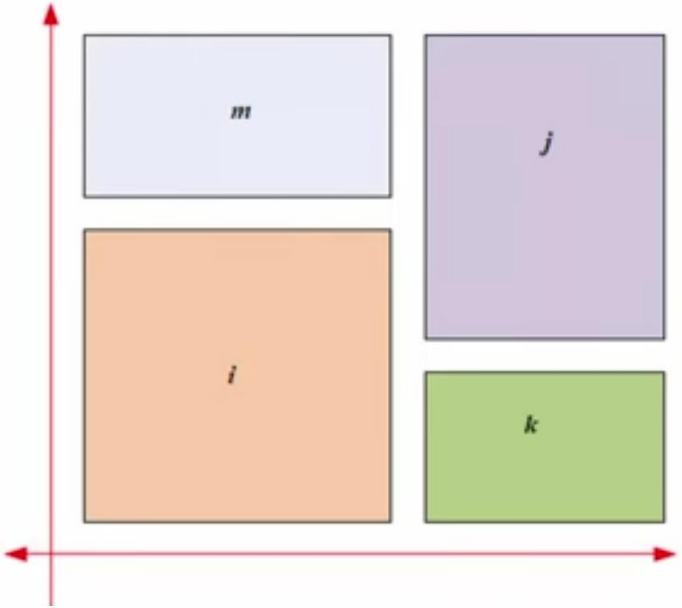
$$x_i + w_i \leq x_j$$

$$y_i + h_i \leq y_j$$

$$x_i - w_j \geq x_j$$

$$y_i - h_j \geq y_j$$

Equation to be satisfied



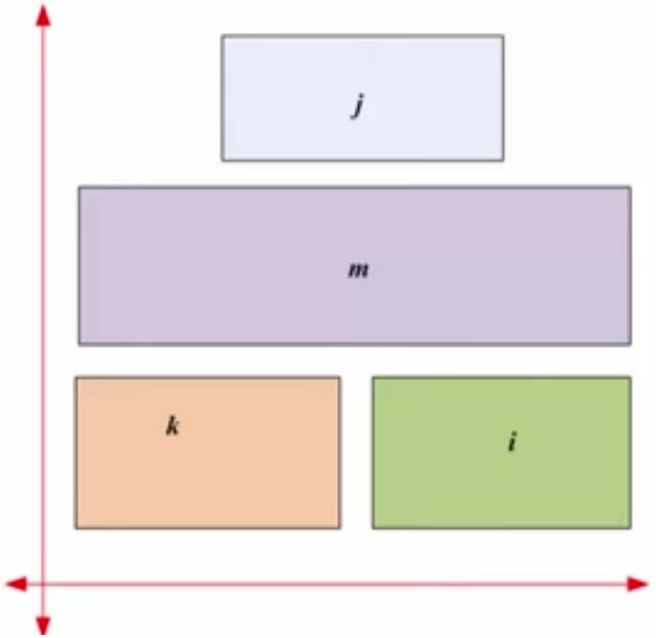
$x_i + w_i \leq x_j$

$y_i + h_i \leq y_j$

$x_i - w_j \geq x_j$

$y_i - h_j \geq y_j$

Equation to be satisfied



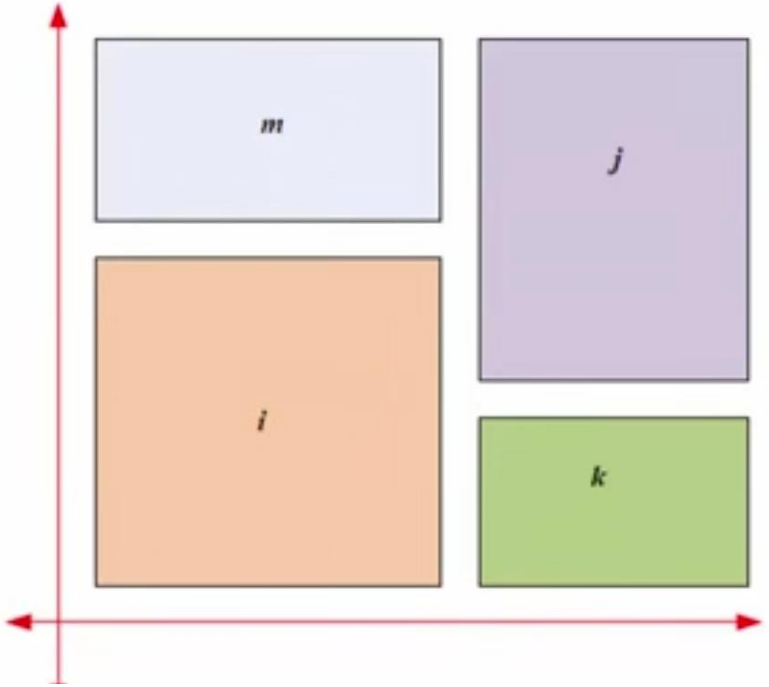
$x_i + w_i \leq x_j$

$y_i + h_i \leq y_j$

$x_i - w_j \geq x_j$

$y_i - h_j \geq y_j$

Equation to be satisfied



$$x_i + w_i \leq x_j$$

$$y_i + h_i \leq y_j$$

$$x_i - w_j \geq x_j$$

$$y_i - h_j \geq y_j$$

Notation & Problem Definition

- Let W and H be upper bounds on the floorplan width and height. Hence, $|x_i - x_j| \leq W$ and $|y_i - y_j| \leq H$
- If W and H are not given, then possible estimates of these quantities could be $W = \Sigma_i w_i$ and $H = \Sigma_i h_i$

Linear Prog. Formulation:

Assumption: one dimension of the chip (W), is fixed.

- CASE 1: All modules are rigid and have fixed orientation.

Constraints:

(1) no two modules overlap

(2) each module is enclosed within the floorplan enveloping rectangle of width W and height Y , i.e., $x_i + w_i \leq W$ and $y_i + h_i \leq Y$; $1 \leq i \leq n$;

(3) all modules are in the first quadrant, $x_i \geq 0$ and $y_i \geq 0$; $1 \leq i \leq n$.

Objective:

- Since the width W is fixed, a possible objective to minimize would be Y , the height of the floor plan.

Linear Prog. Formulation

To summarize, we end up with the following 0-1 integer linear program:

Y ← minimize

Subject to :

$$x_i + w_i \le W; \qquad 1 \le i \le n$$

$$y_i + h_i \le Y; \qquad 1 \le i \le n$$

$$x_i + w_i \le x_j + W(x_{ij} + y_{ij}); \qquad 1 \le i < j \le n$$

$$x_i - w_j \ge x_j - W(1 - x_{ij} + y_{ij}); \qquad 1 \le i < j \le n$$

$$y_i + h_i \le y_j + H(1 + x_{ij} - y_{ij}); \qquad 1 \le i < j \le n$$

$$y_i - h_j \ge y_j - H(2 - x_{ij} - y_{ij}); \qquad 1 \le i < j \le n$$

$$x_i \ge 0;\ y_i \ge 0; \qquad 1 \le i \le n$$

Linear Prog. Formulation:

CASE 2: All modules rigid and rotation allowed.

For each free-orientation module i, one 0-1 integer variable is introduced z_i.

$z_i = 0$ → module i is not rotated; $z_i = 1$ → module i is rotated.

Y ← minimize

Subject to :

$$x_i + z_i h_i + (1 - z_i) w_i \le W; \qquad 1 \le i \le n$$

$$y_i + z_i w_i + (1 - z_i) h_i \le Y; \qquad 1 \le i \le n$$

$$x_i + z_i h_i + (1 - z_i) w_i \le x_j + M(x_{ij} + y_{ij}); \qquad 1 \le i < j \le n$$

$$x_i - z_j h_j - (1 - z_j) w_j \ge x_j - M(1 - x_{ij} + y_{ij}); \qquad 1 \le i < j \le n$$

$$y_i + z_i w_i + (1 - z_i) h_i \le y_j + M(1 + x_{ij} - y_{ij}); \qquad 1 \le i < j \le n$$

$$y_i - z_j w_j - (1 - z_j) h_j \ge y_j - M(2 - x_{ij} - y_{ij}); \qquad 1 \le i < j \le n$$

$$x_i \ge 0; y_i \ge 0; \qquad 1 \le i \le n$$

- M could be set equal to max(W,H) or W + H.
- The number of equations did not change from the first formulation. However, the number of 0-1 integer variables have increased by n, which is equal to the number of modules.

Practice problem

- Formulate the ILP floorplanning for the following problem instances.
- Assume that the dimension of the fixed modules is given as (width, height).
- Four fixed modules: $m_1(4, 5)$, $m_2(3, 7)$, $m_3(6, 4)$, *and* $m_4(7, 7)$. *Rotation* is not allowed.

Practice problem

- Formulate the ILP floorplanning for the following problem instances.
- Assume that the dimension of the fixed modules is given as (width, height).
- Four fixed modules: $m_1(4, 5)$, $m_2(3, 7)$, $m_3(6, 4)$, and $m_4(7, 7)$. *Rotation* is not allowed.
- We associate 8 integer variables – $(x_1,y_1),(x_2,y_2),(x_3,y_3)$ and (x_4,y_4) to denote the co-ordianates of the lower left corner of the 4 modules
- We associate 12 binary variables for each pair of modules- (x_{12},y_{12}) for modules m_1 and m_2, (x_{13},y_{13}) for modules m_1 and m_3 etc

No. integer variables = 2*n
Number of 0-1 variables = n*(n-1)
No. of constraints = 4n+2n(n-1)

Minimize y^*
Subject to
non-overlap constraints:

$$x_1 + w_1 \leq x_2 + 20(x_{12} + y_{12})$$
$$x_1 - w_2 \geq x_2 - 20(1 - x_{12} + y_{12})$$
$$y_1 + h_1 \leq y_2 + 23(1 + x_{12} - y_{12})$$
$$y_1 - h_2 \geq y_2 - 23(2 - x_{12} - y_{12})$$

$$x_1 + w_1 \leq x_3 + 20(x_{13} + y_{13})$$
$$x_1 - w_3 \geq x_3 - 20(1 - x_{13} + y_{13})$$
$$y_1 + h_1 \leq y_3 + 23(1 + x_{13} - y_{13})$$
$$y_1 - h_3 \geq y_3 - 23(2 - x_{13} - y_{13})$$

Size of the linear program:
2 x n continuous variables, n(n - 1) integer variables, and $2n^2$ linear constraints.

$$x_1 + w_1 \leq x_4 + 20(x_{14} + y_{14})$$
$$x_1 - w_4 \geq x_4 - 20(1 - x_{14} + y_{14})$$
$$y_1 + h_1 \leq y_4 + 23(1 + x_{14} - y_{14})$$
$$y_1 - h_4 \geq y_4 - 23(2 - x_{14} - y_{14})$$

$$x_2 + w_2 \leq x_3 + 20(x_{23} + y_{23})$$
$$x_2 - w_3 \geq x_3 - 20(1 - x_{23} + y_{23})$$
$$y_2 + h_2 \leq y_3 + 23(1 + x_{23} - y_{23})$$
$$y_2 - h_3 \geq y_3 - 23(2 - x_{23} - y_{23})$$

$$x_2 + w_2 \leq x_4 + 20(x_{24} + y_{24})$$
$$x_2 - w_4 \geq x_4 - 20(1 - x_{24} + y_{24})$$
$$y_2 + h_2 \leq y_4 + 23(1 + x_{24} - y_{24})$$
$$y_2 - h_4 \geq y_4 - 23(2 - x_{24} - y_{24})$$

$$x_3 + w_3 \leq x_4 + 20(x_{34} + y_{34})$$
$$x_3 - w_4 \geq x_4 - 20(1 - x_{34} + y_{34})$$
$$y_3 + h_3 \leq y_4 + 23(1 + x_{34} - y_{34})$$
$$y_3 - h_4 \geq y_4 - 23(2 - x_{34} - y_{34})$$

variable type constraints:

$$x_1 \geq 0,\ x_2 \geq 0,\ x_3 \geq 0,\ x_4 \geq 0$$
$$y_1 \geq 0,\ y_2 \geq 0,\ y_3 \geq 0,\ y_4 \geq 0$$
$$x_{12},\ x_{13},\ x_{14},\ x_{23},\ x_{24},\ x_{34} \in \{0,1\}$$
$$y_{12},\ y_{13},\ y_{14},\ y_{23},\ y_{24},\ y_{34} \in \{0,1\}$$

chip width constraints:

$$x_1 + w_1 \leq y^*$$
$$x_2 + w_2 \leq y^*$$
$$x_3 + w_3 \leq y^*$$
$$x_4 + w_4 \leq y^*$$

chip height constraints:

$$y_1 + h_1 \leq y^*$$
$$y_2 + h_2 \leq y^*$$
$$y_3 + h_3 \leq y^*$$
$$y_4 + h_4 \leq y^*$$

Solving using GLPK, we obtain the following solution:

$y^* = 12$

$(x_1, y_1) = (7, 7), (x_2, y_2) = (9, 0), (x_3, y_3) = (0, 0), (x_4, y_4) = (0, 4)$

$(x_{12}, y_{12}) = (1, 1) : (1 \text{ is above } 2)$

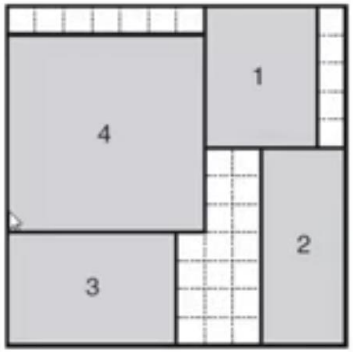
$(x_{13}, y_{13}) = (1, 1) : (1 \text{ is above } 3)$

$(x_{14}, y_{14}) = (1, 0) : (1 \text{ is to the right of } 4)$

$(x_{23}, y_{23}) = (1, 0) : (2 \text{ is to the right of } 3)$

$(x_{24}, y_{24}) = (1, 0) : (2 \text{ is to the right of } 4)$

$(x_{34}, y_{34}) = (0, 1) : (3 \text{ is below } 4)$



$y^* = 12$

$(x_1, y_1) = (7, 7), (x_2, y_2) = (9, 0), (x_3, y_3) = (0, 0), (x_4, y_4) = (0, 4)$

$(x_{12}, y_{12}) = (1, 1) : (1 \text{ is above } 2)$

$(x_{13}, y_{13}) = (1, 1) : (1 \text{ is above } 3)$

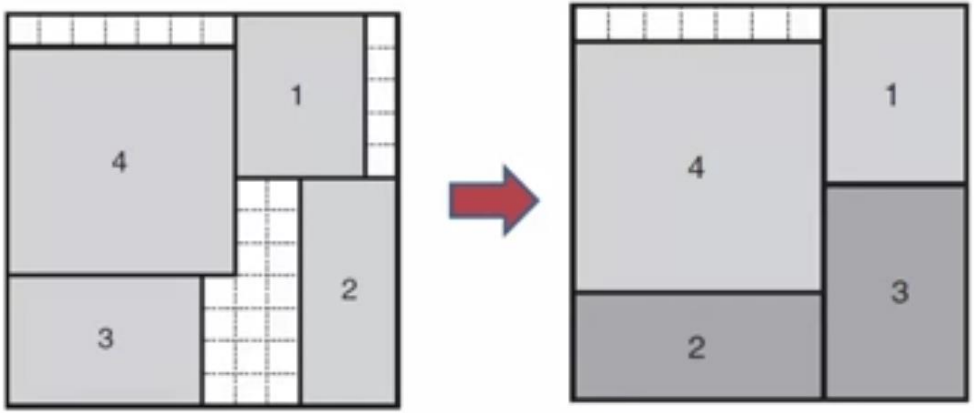
$(x_{14}, y_{14}) = (1, 0) : (1 \text{ is to the right of } 4)$

$(x_{23}, y_{23}) = (1, 0) : (2 \text{ is to the right of } 3)$

$(x_{24}, y_{24}) = (1, 0) : (2 \text{ is to the right of } 4)$

$(x_{34}, y_{34}) = (0, 1) : (3 \text{ is below } 4)$

Same problem, rotation allowed



Refer: Practical Problems in Physical design automation by Sung Kyu Lim, Springer,2008

Ninevah University

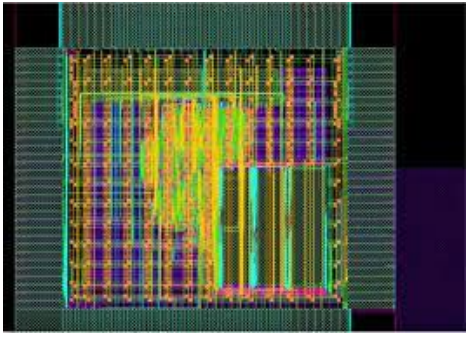
Collage of Electronics Engineering

Course:

Electronic Design Automation

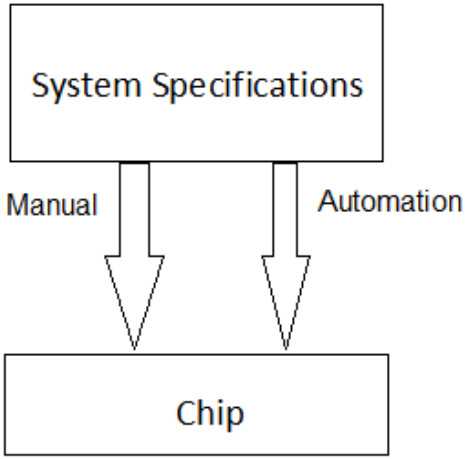
Lecturer: H. M. Hussein

EDA07: Physical Design Automation



VLSI Design Cycle

- Large number of devices
- Optimization requirements for high performance
- Time-to-market competition
- Cost



VLSI Design Cycle (contd.i

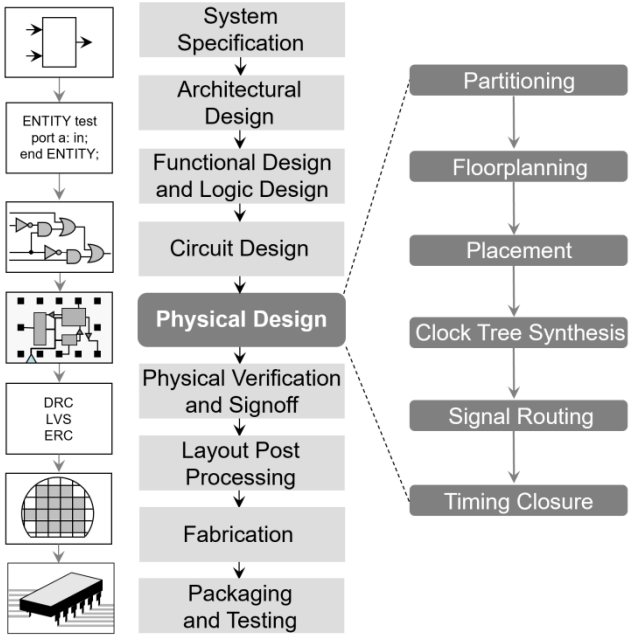
1. System specification
2. Functional design
3. Logic design
4. Circuit design
5. Physical design
6. Design verification
7. Fabrication
8. Packaging, testing, and debugging

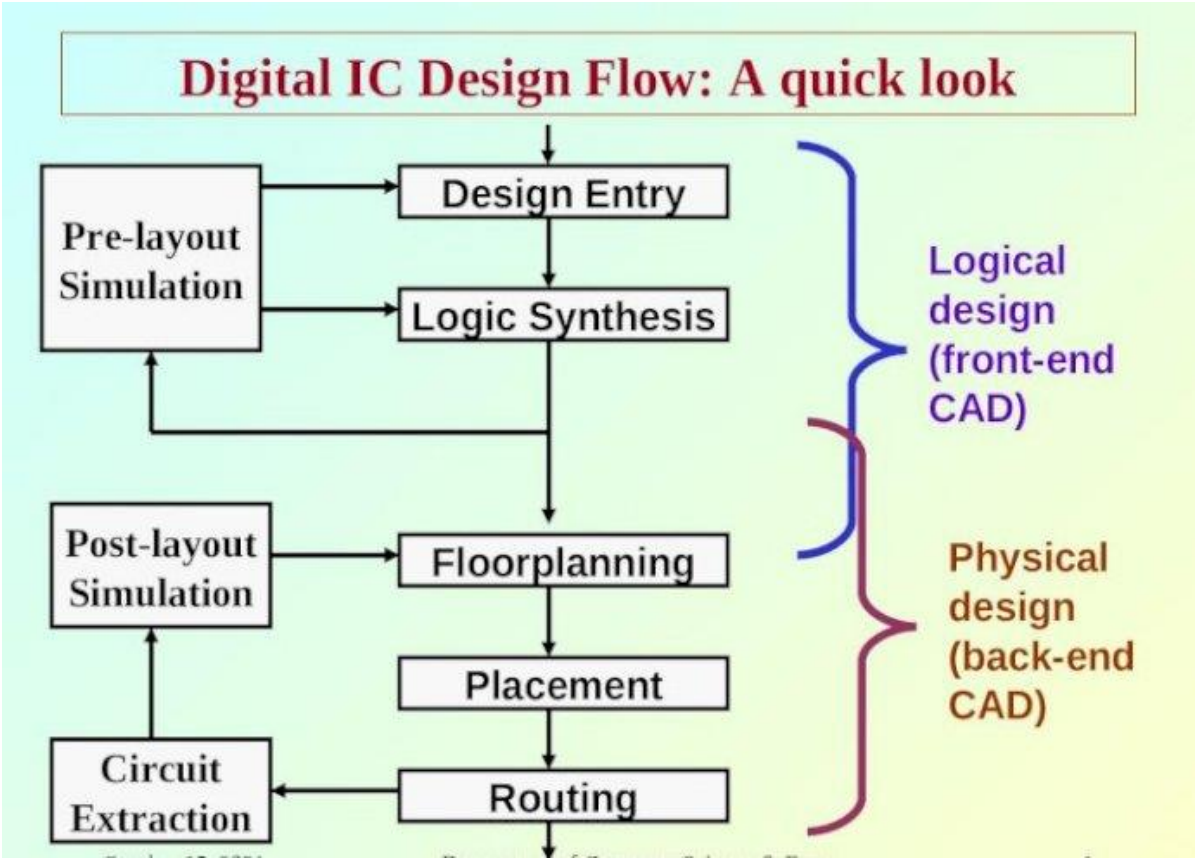
Physical Design

- Converts a circuit description into a geometric description.

  -This description is used for fabrication of the chip.

- Basic steps in the physical design cycle
  1. Partitioning
  2. Floor-planning and placement
  3. Routing
  4. Compaction




Digital IC Design Flow: A quick look
Design Entry
Pre-layout Simulation
Logic Synthesis
Logical design (front-end CAD)
Post-layout Simulation
Floorplanning
Physical design (back-end CAD)
Placement
Circuit Extraction
Routing

VLSI Design Styles

- Programmable Logic Devices
  - — Programmable Logic Device (PLD)
  - — Field Programmable Gate Array (FPGA)
  - — Gate Array

- Standard Cell (Semi-Custom Design)

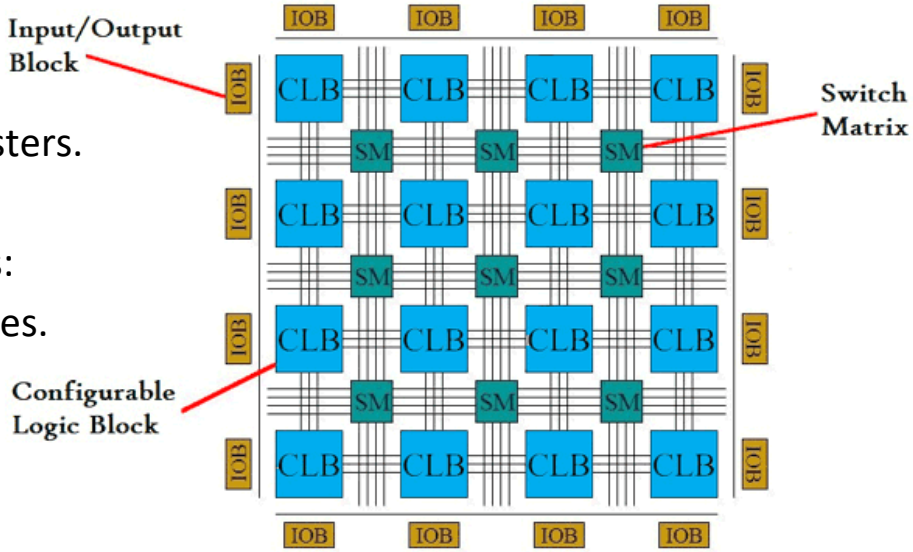
- Full-Custom Design

Field Programmable Gate Array (FPGA)

- User / Field Programmability.
- Array of logic cells connected via routing channels.

- Different types of cells:
  - Special I/O cells.
  - Logic cells.
    - Mainly lookup tables (LUT) with associated registers.

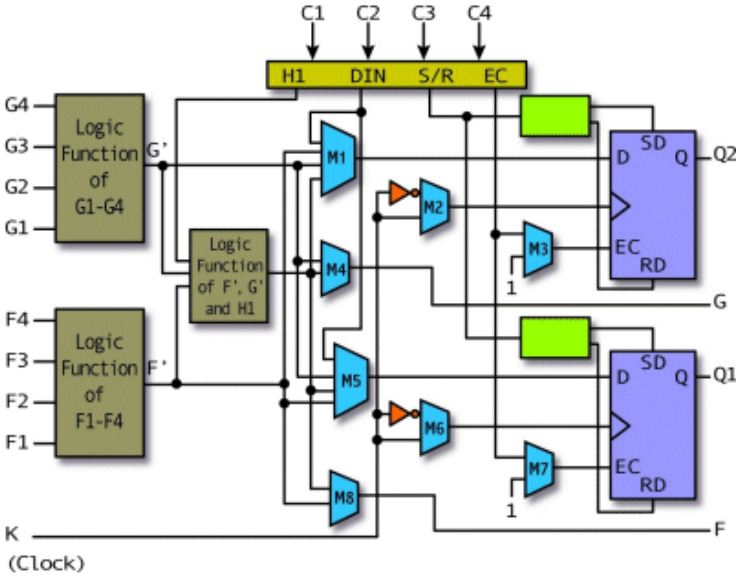
- Interconnection between cells:
  - Using SRAM based switches.
  - Using antifuse elements.



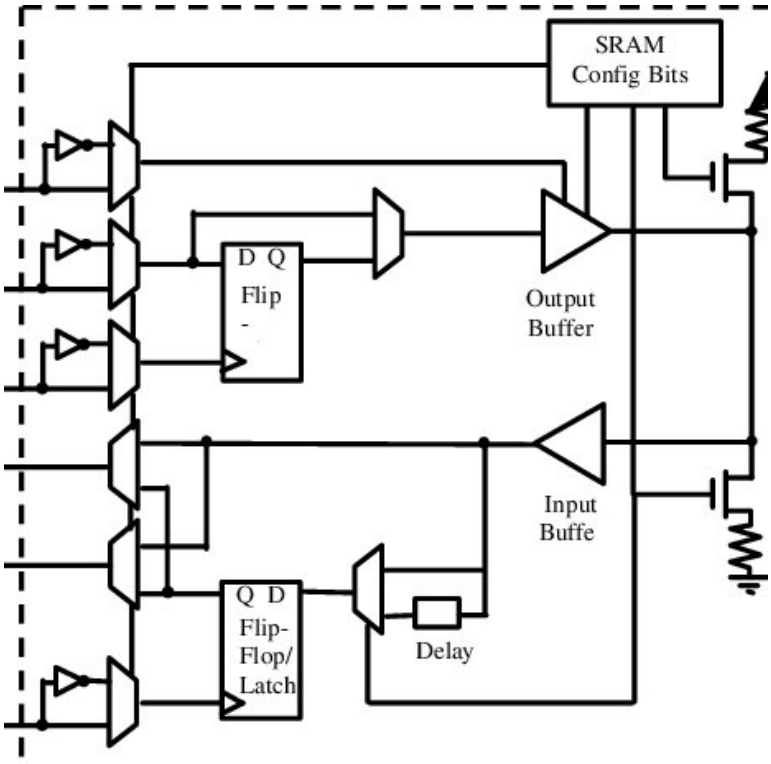
Configurable Logic Block:
CLB Functionalities

• Two 4-input function generators - Implemented using Lookup Tables using 16x1 RAM. - Can also implement 16x1 memory.

• Two Registers Each can be configured as flip-flop or latch. - Independent clock polarity. - Synchronous and asynchronous Set / Reset.



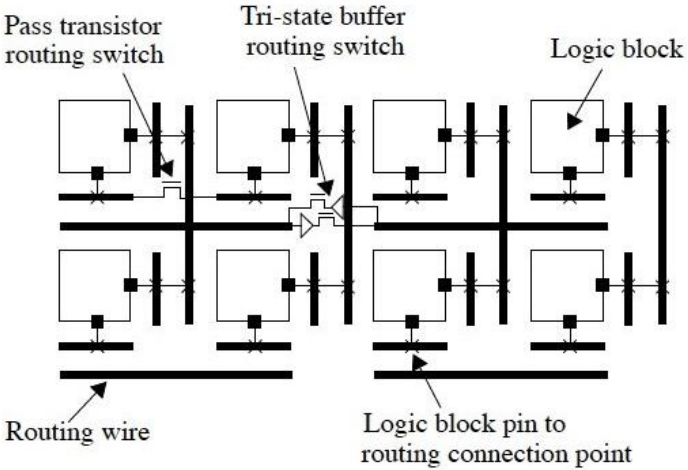
I/O Block Diagram



Xilinx FPGA Routing

1) Fast Direct Interconnect - CLB to CLB

2) General Purpose Interconnect - Uses switch matrix.



FPGA Design Flow

- Design Entry
  - In schematic, VHDL, or Verilog.
- Implementation
  - Placement & Routing
  - Bit-stream generation
  - Analyze timing, view layout, simulation, etc.
- Download
  - Directly to Xilinx hardware devices with unlimited reconfigurations.

Gate Array

• In view of the fast prototyping capability, the gate array (GA) comes after the FPGA.

- Design implementation of
  - • FPGA chip is done with user programming,
  - • Gate array is done with metal mask design and processing.

• Gate array implementation requires a two-step manufacturing process:

1. The first phase, which is based on generic (standard) masks, results in an array of uncommitted transistors on each GA chip.
2. These uncommitted chips can be customized later, which is completed by defining the metal interconnects between the transistors of the array.

Design Phases:

Phase 1:

Fabricate an array of transistors/gates:

- Diffusion
- Poly-silicon
- Oxidation

Phase 2:

Interconnect transistors/gates

- metallization.

• The GA chip utilization factor is higher than that of FPGA.

The used chip area divided by the total chip area.

• Chip speed is also higher.

- More customized design can be achieved with metal mask designs.

• Current gate array chips can implement as many as hundreds of thousands of logic gates.

Standard Cell

- One of the most prevalent custom design styles.
  - -- Also called semi-custom design style.
  - Requires developing full custom mask set.
- Basic idea:
  - All of the commonly used logic cells are developed, characterized, and stored in a standard cell library.
  - A typical library may contain a few hundred cells.
    - Inverters, NAND gates, NOR gates, complex AOI, OAI gates, D-latches, and flip-flops.

Characteristic of the Cells

Each cell is designed with **a fixed height**.

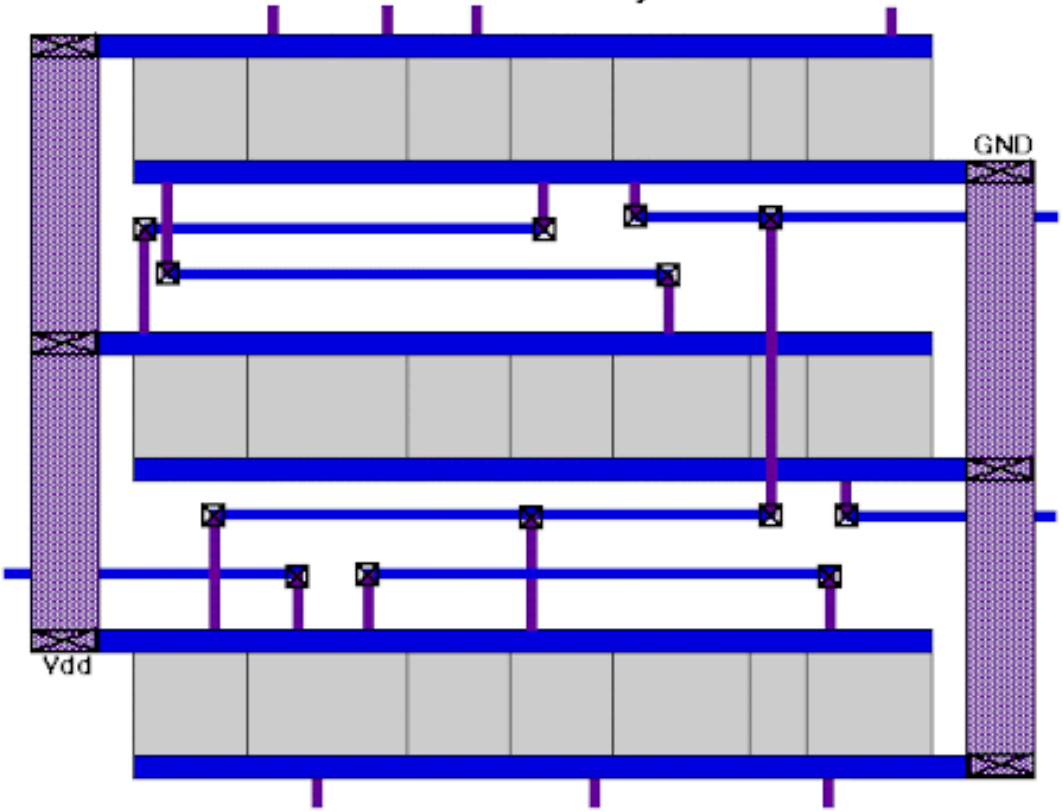
- To enable automated placement of the cells,
- Routing of inter-cell connections.
- A number of cells can be abutted side-by-side to form rows.

• The power and ground rails typically run parallel to upper and lower boundaries of cell.

- Neighboring cells share a common power and ground bus.
- nMOS transistors are located closer to the ground rail while the pMOS transistors are placed closer to the power rail.

• The input and output pins are located on the upper and lower boundaries of the cell.

5. Standard cell layout



Floor-plan for Standard Cell Design

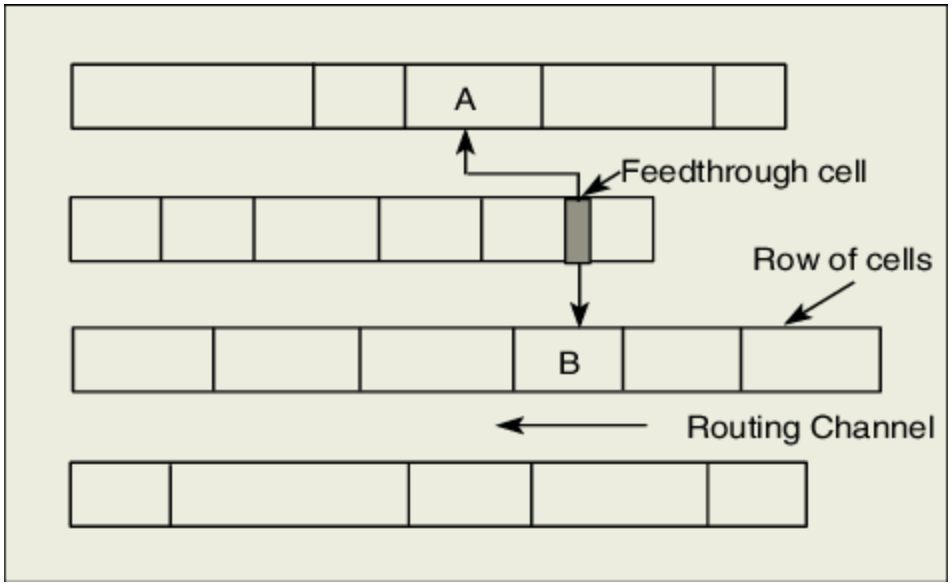
• Inside the I/O frame which is reserved for I/O cells, the chip area containsrows or columns of standard cells.

- Between cell rows are channels for dedicated inter-cell routing.
- Over-the-cell routing is also possible.

• The physical design and layout of logic cells ensure that

- When placed into rows, their heights match.
- Neighboring cells can be abutted side-by-side, which provides natural connections for power and ground lines in each row.

Feed-through Cell



Full Custom Design

• The standard-cells based design is often called semi-custom design.

- The cells are pre-designed for general use and the same cells are utilized in many different chip designs.

• In the full custom design, the entire mask design is done anew without use of any library.

- The development cost of such a design style is prohibitively high.
- The concept of design reuse is becoming popular to reduce design cycle time and cost.

• The most rigorous full custom design can be the design of a memory cell.

- Static or dynamic.
- Since the same layout design is replicated, there would not be any alternative to high density memory chip design.

• For logic chip design, a good compromise can be achieved by using a combination of different design styles on the same chip.

— Standard cells, data-path cells and PLAs.

Comparison among Various Design Styles

	Design Style			
	FPGA	Gate array	Standard cell	Full custom
Cell size	Fixed	Fixed	Fixed height	Variable
Cell typo	Programmable	Fixed	Variable	Variable
Cell placement	Fixed	Fixed	In row	Variable
Interconnect	Programmable	Variable	Variable	Variable
Design time	Very fast	Fast	Medium	Slow

Circuit Partitioning:

System Design

- Decomposition of a complex system into smaller subsystems.
- Each subsystem can be designed independently. • Decomposition scheme has to minimize the interconnections between the subsystems. • Decomposition is carried out hierarchically until each subsystem is of manageable size.

M1, M2, …, Mn, Interface Information

Example

Wires

Cut1-2: 4

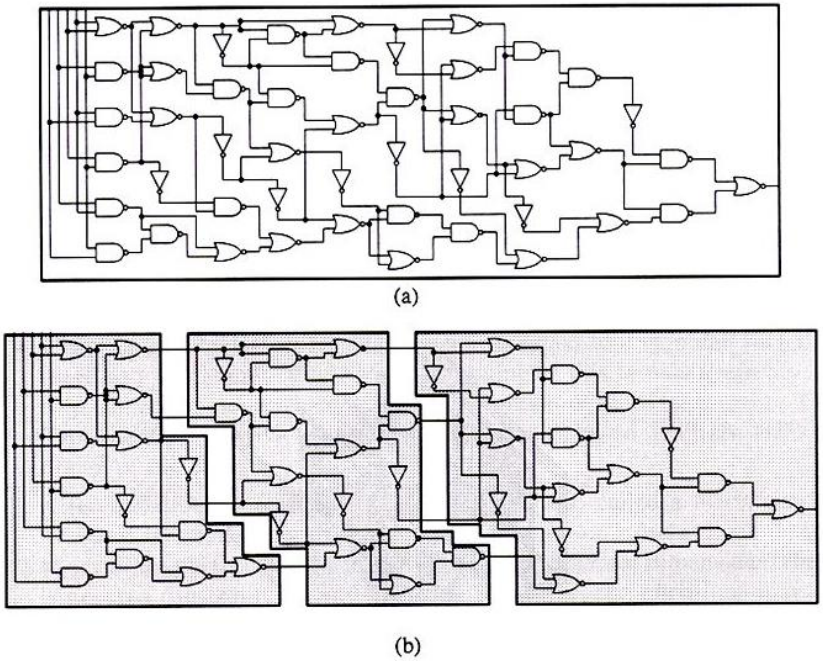
Cut2-3: 4

Size:

Cut1: 15

Cut2: 16

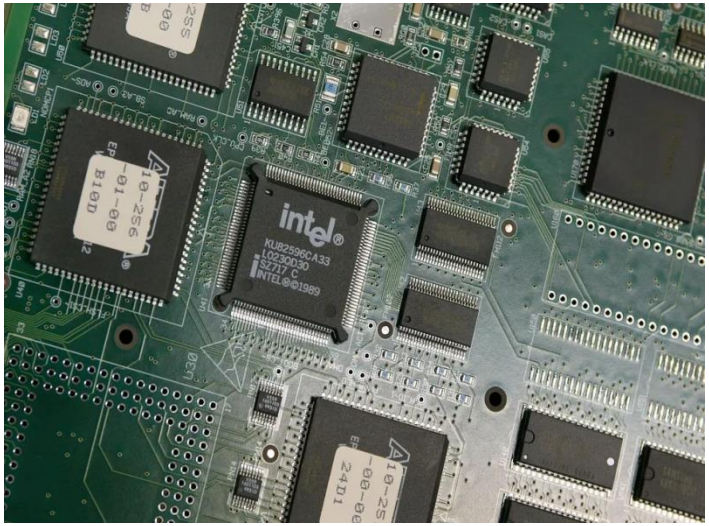
Cut3: 17

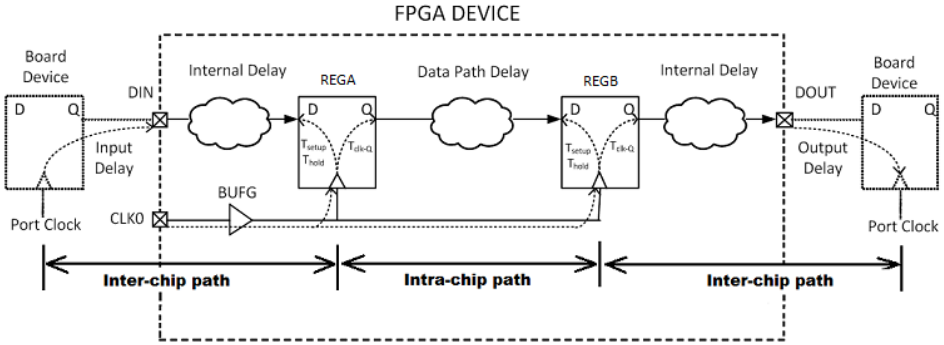


Partitioning at Different Levels

- Can be done at multiple levels:
  - — System level
  - — Board level
  - — Chip level.

- Delay implications are different:
  - — Intra-chip → X
  - — Intra-board or Inter-chip → 10X
  - — Inter-board → 20X

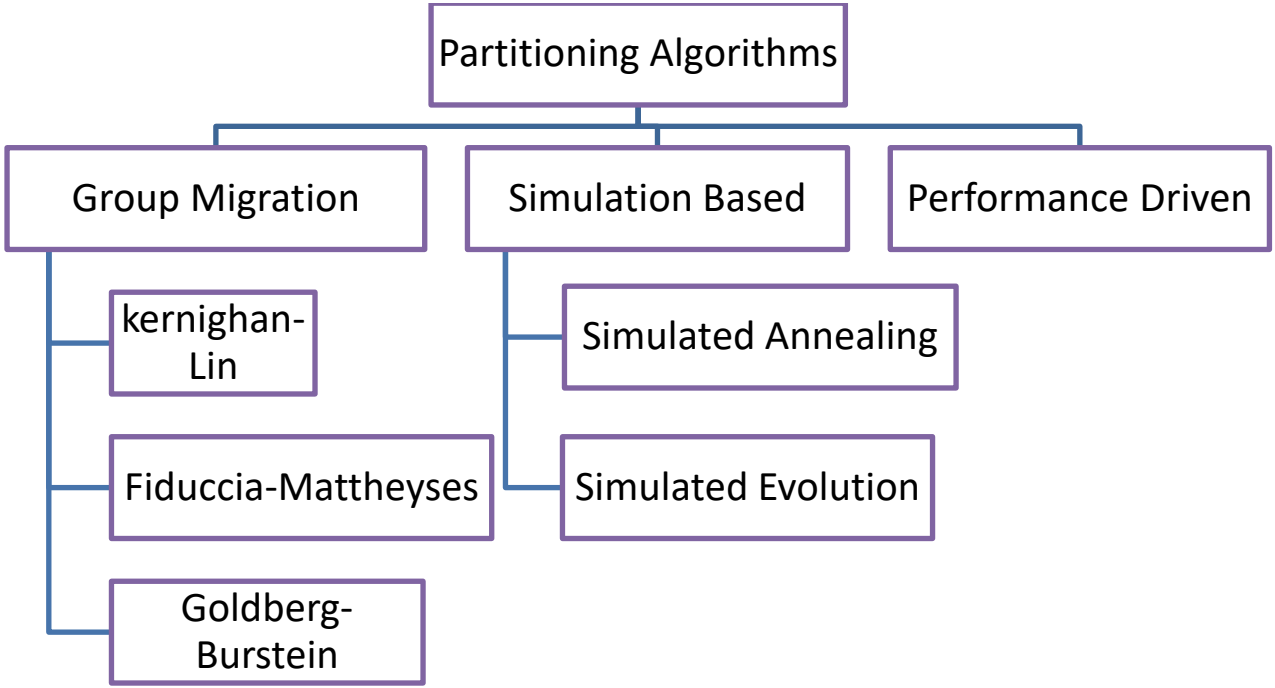




Problem Formulation

1. Interconnection between partitions is minimized.
2. Delay due to partitioning is minimized.
3. Number of terminals must be less than a predetermined maximum value.
4. The area of each partition should remain within specified bounds.
5. The number of partitions should also remain within specified bounds.

Classification of Partitioning Algorithms



Group Migration Algorithms

- Kernighan-Lin
  — An iterative improvement algorithm for balanced two-way partitioning.
- Goldberq-Burstein
  — Uses properties of graphs to improve the performance of K-L algorithm.
- Fiduccia-Mattheyses
  — Considers multi-pin nets.
  — Can generate partitions of unequal sizes.
  — Uses efficient data structure to represent nodes.

Extension of K-L Algorithm

• Unequal sized blocks

- To partition a graph with 2n vertices into two sub-graphs of unequal sizes n/ and n2:

  • Divide the nodes into two subsets A and B, containing MIN (n1, n2) and MAX (n1, n2) vertices respectively.

  • Apply K-L algorithm, but restrict the maximum number of vertices that can be interchanged in one pass to MIN (n1, n2).

Unequal sized elements

- To generate a two-way partition of a graph whose vertices have unequal sizes:

  - Assume that the smallest element has unit size.
  - Replace each element of size s with s vertices which are fully connected (s-clique) with edges of infinite weight.
  - Apply K-L algorithm to the modified graph.

Simulated Annealing and Evoluticiti

- These belong to the probabilistic and iterative class of algorithms.
- Simulated Annealing
  - Simulates the annealing process used for metals.
  - As in the actual annealing process, the value of temperature is decreased slowly till it approaches the freezing point.
- Simulated Evolution
  - Simulates the biological process of evolution.
  -Each solution (generation) is improved in each iteration by using operators which simulate the biological events in the evolution process.

Simulated Annealing

- Concept analogous to the annealing process for metals and glass.
- A random initial partition is available as input.
- A new partition is generated by exchanging some elements.
- If the quality of partition improves, the move is always accepted.
- If not, the move is accepted with a probability which decreases with the (increase) in a parameter called temperature (T).

Simulated Annealing Algorithm

```
Algorithm SA
begin
        t = to;
        cur_part =init_part;
        cur_score = SCORE(cur part);
        repeat
                repeat
                comp1 = SELECT(part1 );
                comp2 = SELECT (part2);
                trial_part = EXCHANGE (comp1, comp2, cur_part); trial_score =  SCORE (trial_part);
                delta_s = trial_score – cur_score;
                if (delta_S<0) then
                        cur_score = trial_score;
                        cur_part = MOVE (comp1, comp2);
                else
                        r = RAND (0,1);
                        if (r < exp(- delta_s/t)) then
                                cur_ score = trial score;
                                cur_part = MOVE(comp1, comp2);
                 until (equilibrium at t is reached);
                t= alpha *t;    / /* 0 < alpha <1 */
        until (freezing point is reached);
end
```

- The SCORE function

  Imbalance(A,B) = I size(A) — size(B) |

  Cutcost(A,B) = Sum of weights of cut edges

  Cost = W1 * Imbalance(A,B) + W2*Cutcost(A,B)

- The MOVE function

  — Several alternatives:

  - Pairwise exchange (W1 =0)
  - Subsets of elements exchanged
  - Select that node
    - Which is internally connected to least number of vertices.
    - Whose contribution to external cost is highest.

Performance Driven Partitioning

• Typically, on-board delay is three orders of magnitude larger than on-chip delay.

— On-chip delay is of the order of nanoseconds.

— On-board delay can be in the order of milliseconds.

• If a critical path is cut many times by the partition, the delay in the path may be too large to meet the goals of high-performance systems.

• Goal of partitioning in high-performance systems:

1. Reduce the cut-size.
2. Minimize the delay in critical paths.
3. Timing constraints have to be satisfied.

- The problem can be modeled as a graph.
  - Each vertex represents a component (gate).
  - Each edge represents a connection between two gates.
  - Each vertex has a weight specifying the component delay.
  - Each edge has a weight, which depends on the partitions to which the edges belong.

> This problem is very general and still a topic of intensive research.

Summary

- Broadly, two classes of algorithms:

1. Group migration based
    - High speed
    - Poor performance
2. Simulation based
    - Low speed.
    - High performance.

Ninevah University

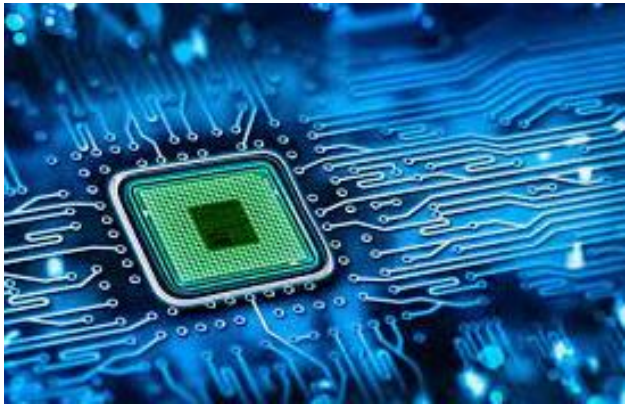
Collage of Electronics Engineering

Course:

Electronic Design Automation

Lecturer: H. M. Hussein

EDA06: High Level Synthesis



Design Representation

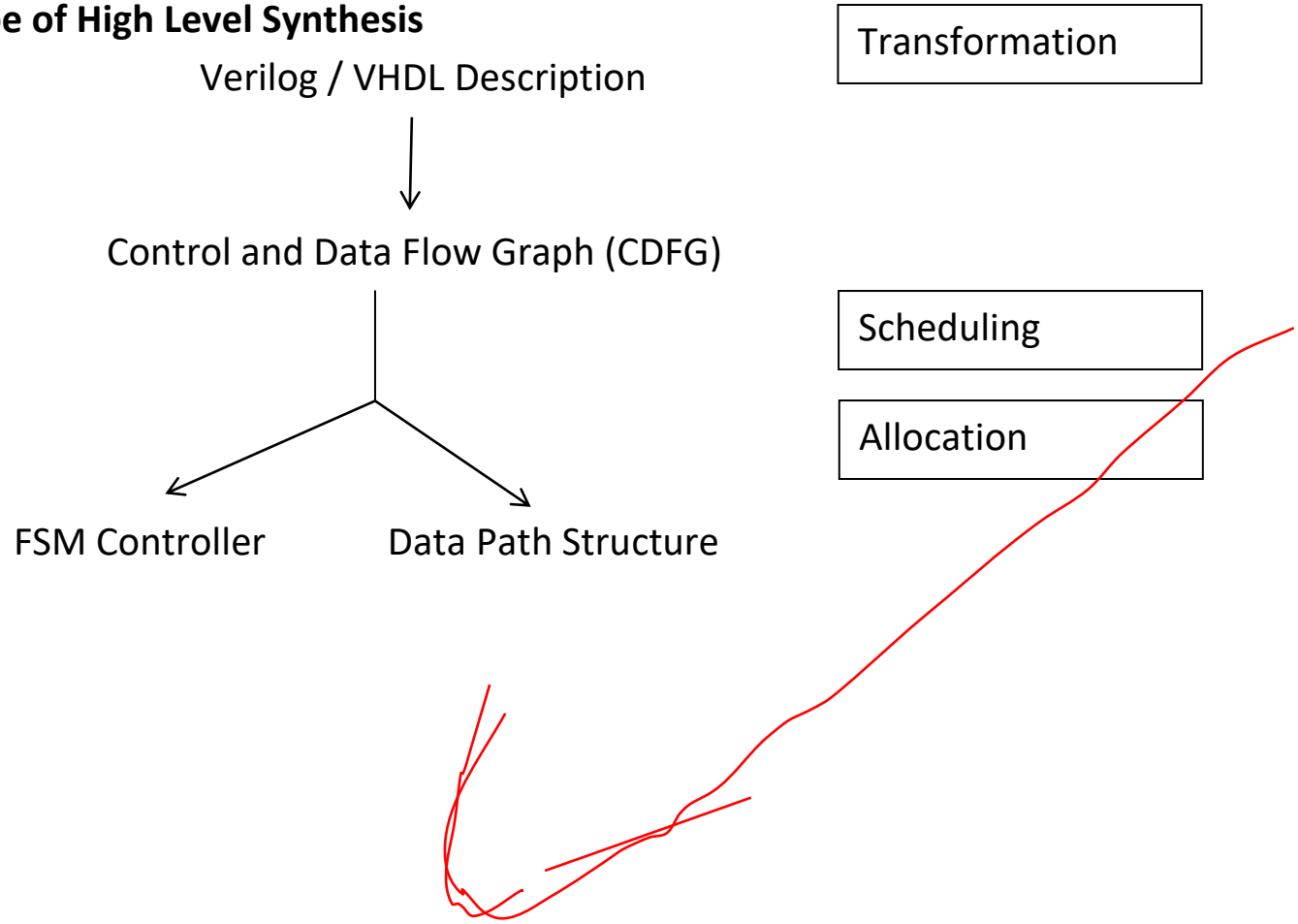
• Intermediate representation essential for efficient processing.

— Input HDL behavioral descriptions translated into some canonical intermediate representation.

- Language independent
- Uniform view across CAD tools and users

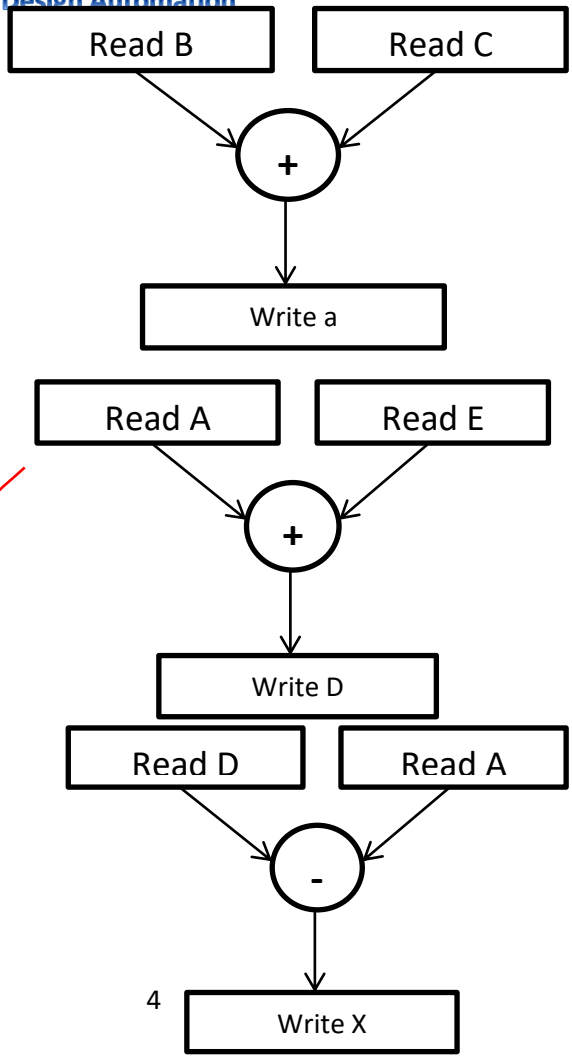
- Synthesis tools carry out transformations of the intermediate representation.

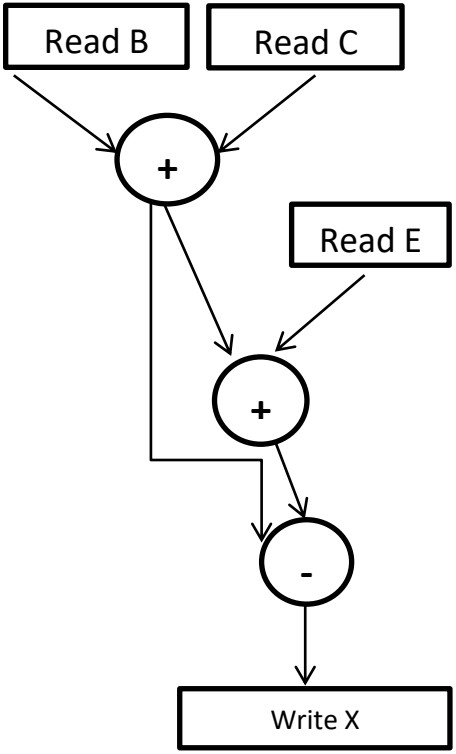
Scope of High Level Synthesis



Simple Transformation

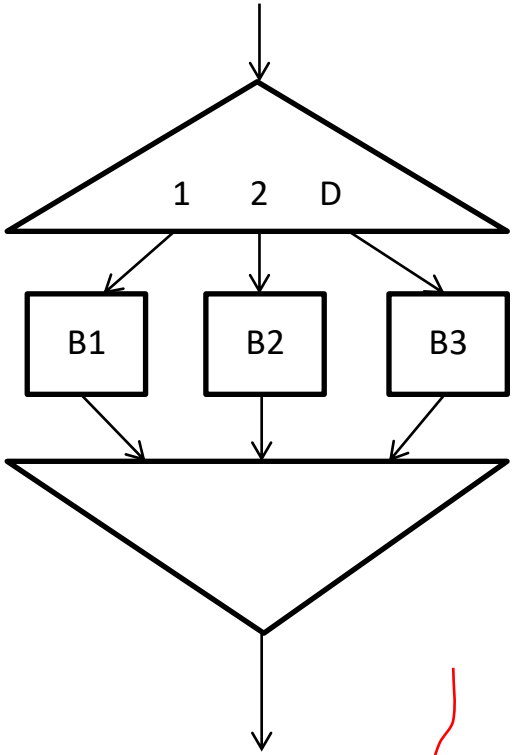
```
A <= B + C;
D <= A + E;
X <= D - A;
```



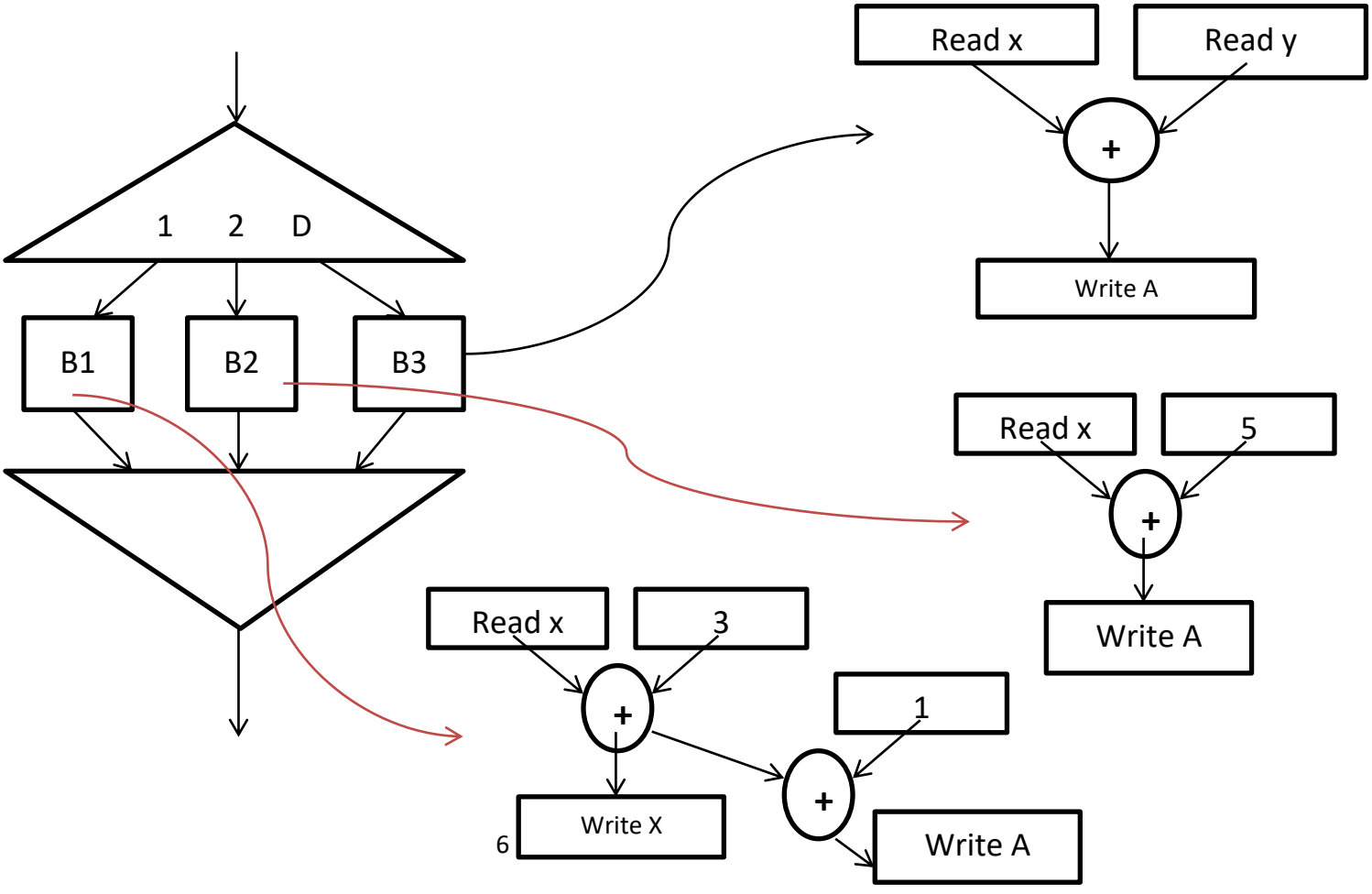


Transformation with Control/Data Flow

```
case (C)
    1: begin
           X = X + 3;
           A = X + 1;
       end
    2: A = X + 5;
    default: A = X + Y;
endcase
```

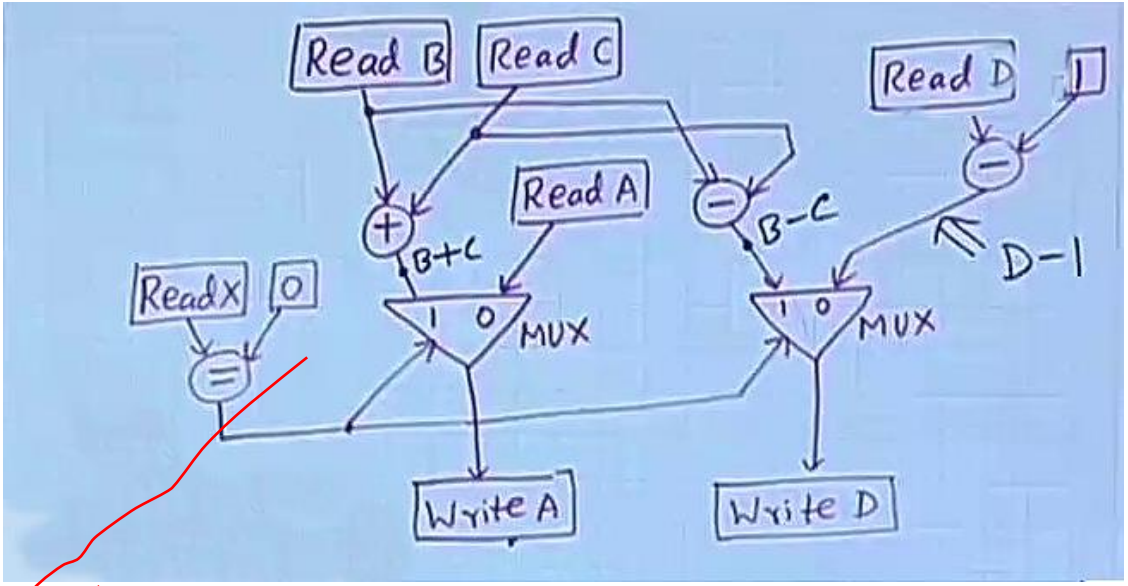


Transformation with Control/Data Flow



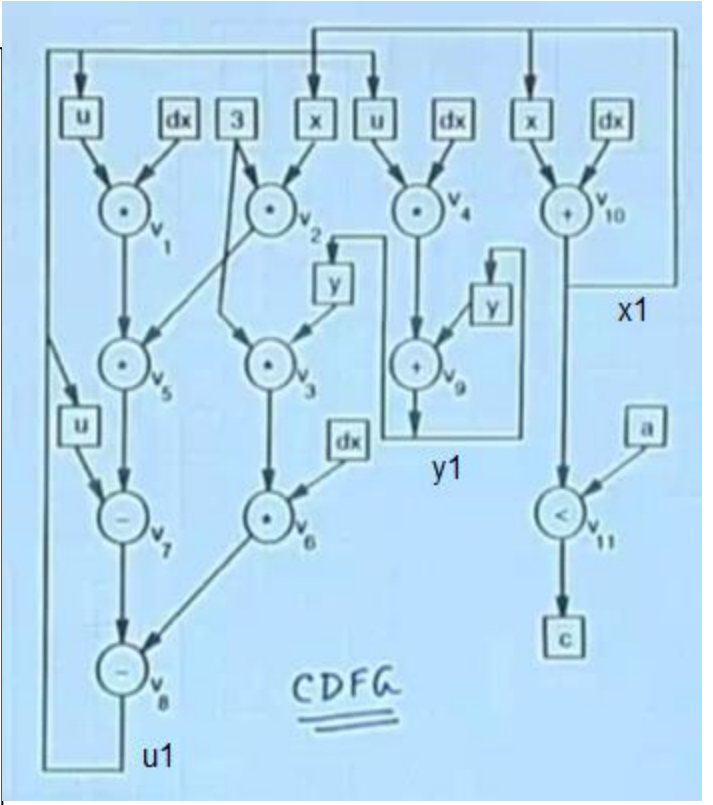
Another Example

```
If (x==0)
    A <= + C;
   D <= B — C;
else
    D = D-1;
```



- Solving 2nd order differential equations

```
module HAL (x, dx, 1.1, a, clock, y);
 input x, dx, u, a, clock;
 output y;

always @(posedge clock)
     while (x < a)
          begin
                x1 = x + dx;
                u1 = u — (3* x * u * dx) — (3* y *
          dx);
                y1 = y + (u * dx);
                x = x1;
                u = u1;
                y= y1;
            end
endmodule
```



Compiler Transformations

• Set of operations carried out on the intermediate representation.

— Constant folding

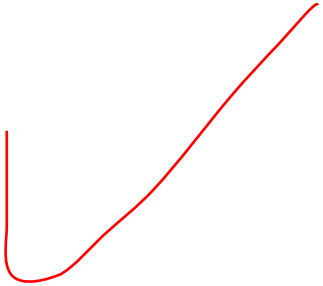
— Redundant operator elimination

— Tree height transformation

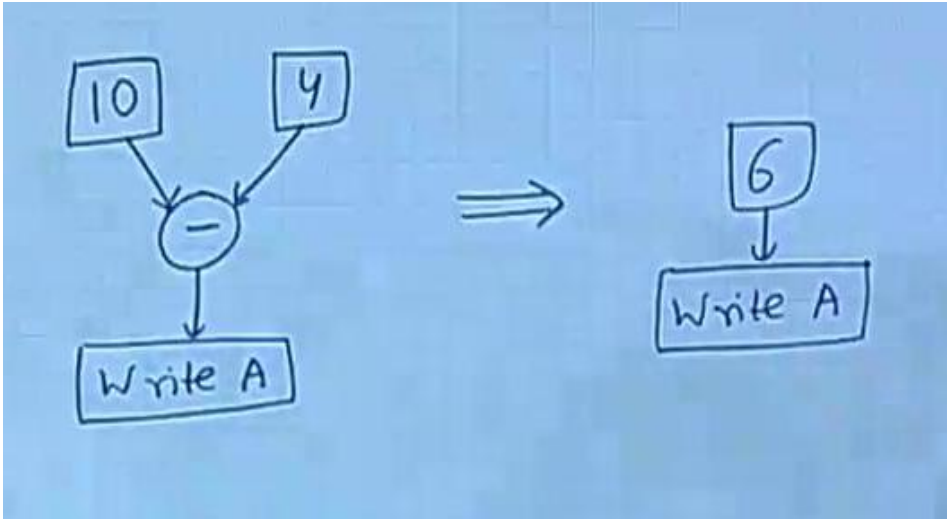
— Control flattening

— Logic level transformation

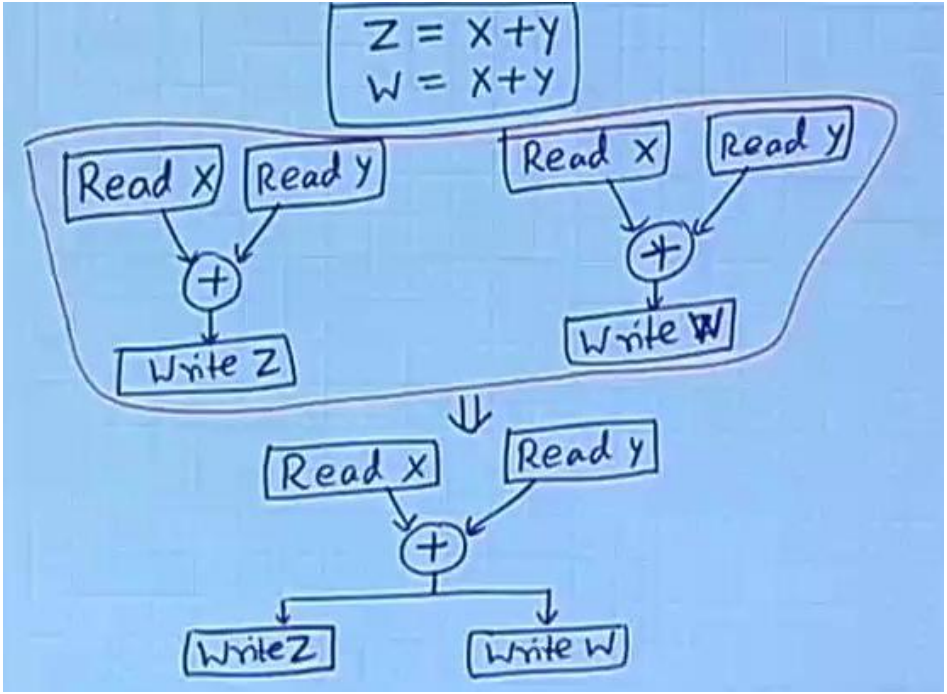
— Register-Transfer level transformation



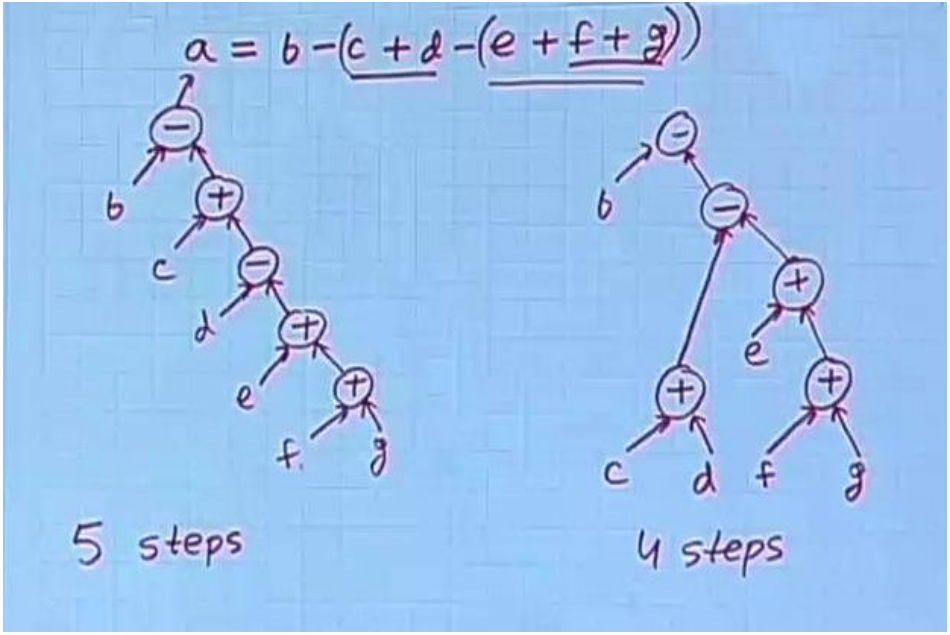
Constant Folding:



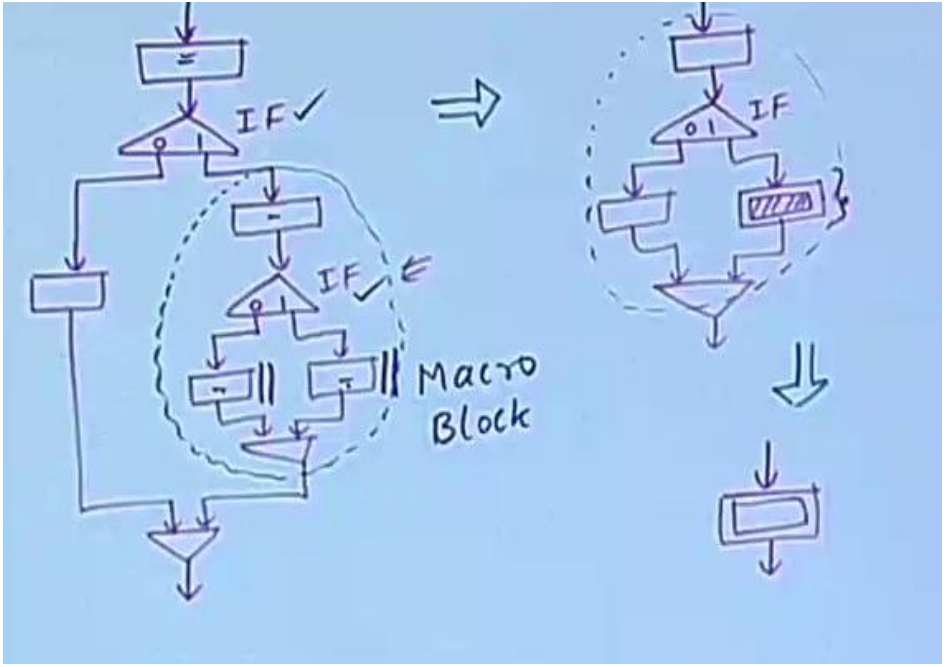
Redundant Operator Elimination:



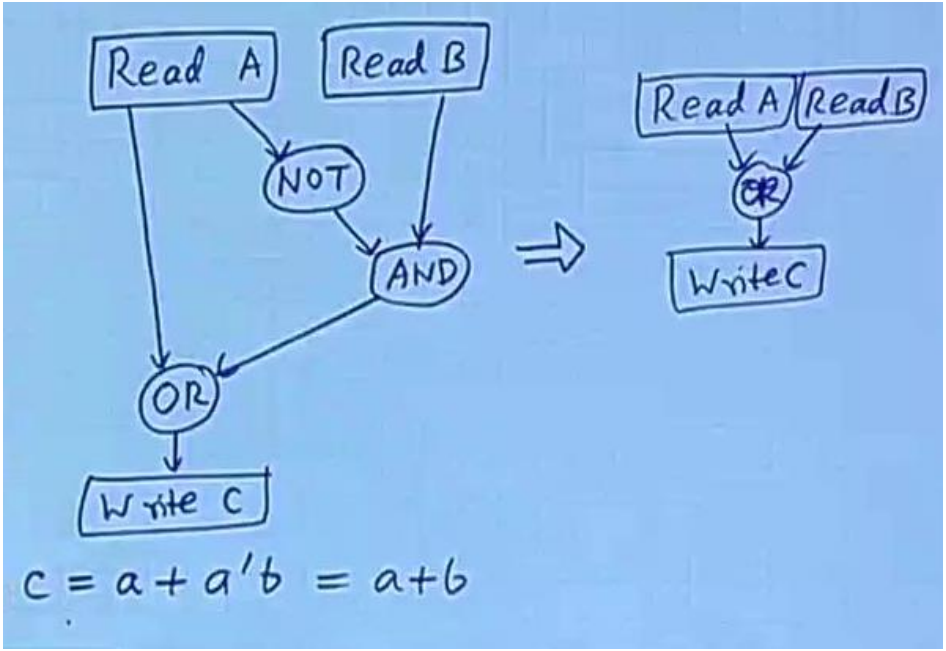
Tree height transformation



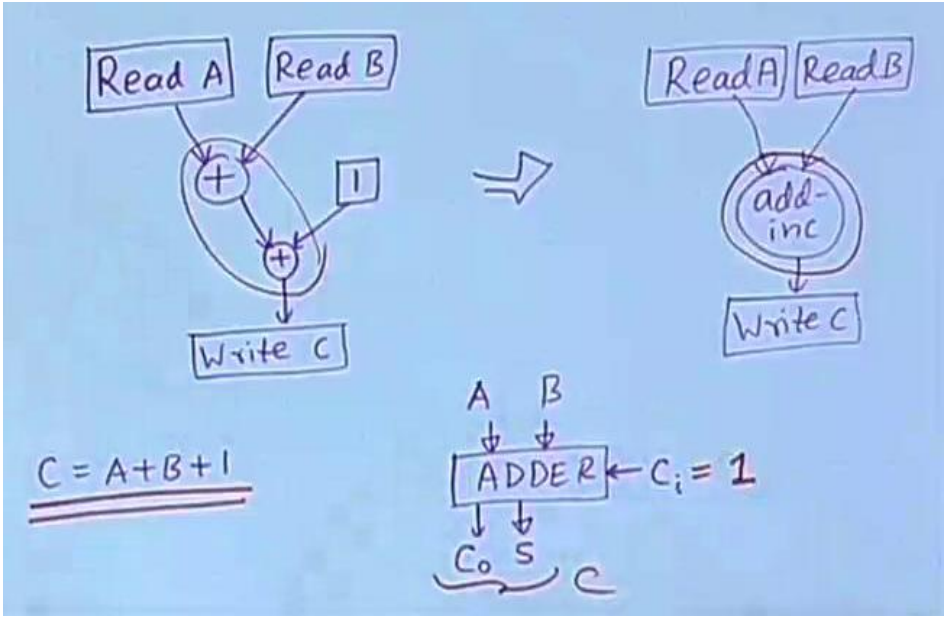
Control Flattening:



Logic Level Transformation:



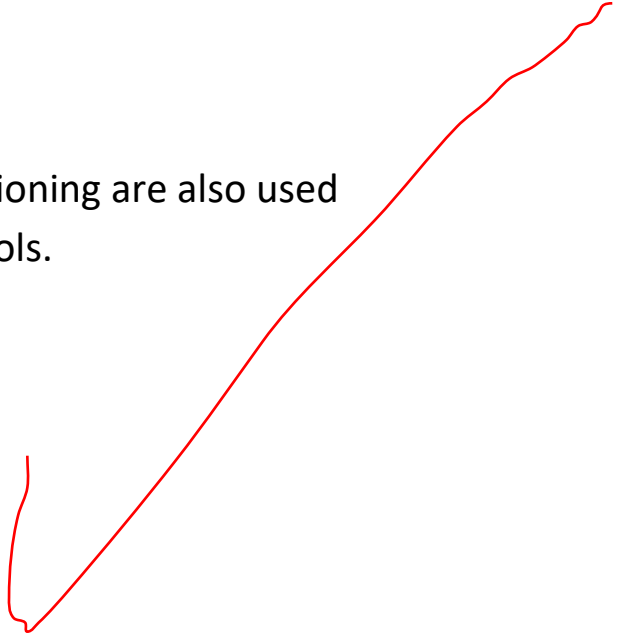
RTL Level Transformation:



Partitioning

Why Required?

- Used in various steps of High Level Synthesis:
  - Scheduling
  - Allocation
  - Unit selection
- The same techniques for partitioning are also used in physical design automation tools.

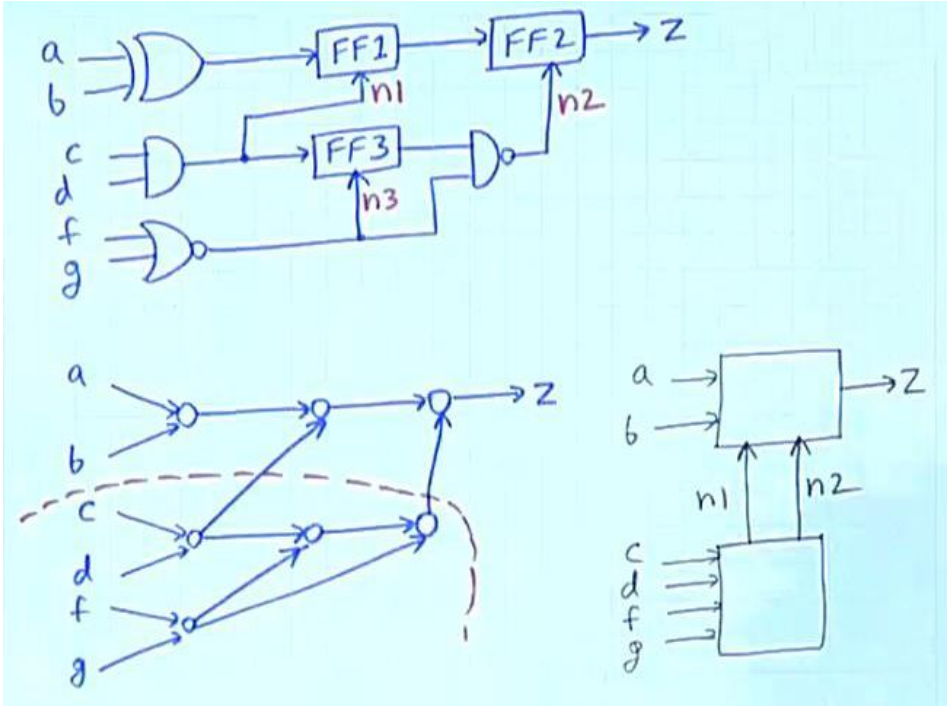


Component Partitioning

• Given a netlist, create a partition which satisfies some objective function.

- Clusters almost of equal sizes.
- Minimum interconnection strength between clusters.

- **An example to illustrate the concept.**



Behavioral Partitioning

- With respect to Verilog, can be used when:
  - — Multiple modules are instantiated in a top-level module description.
    - Each module becomes a partition.
  - — Several concurrent "always" blocks are used.
- Each "always" block becomes a partition.

Partitioning Techniques

- Broadly two classes of algorithms:
  1- Constructive
    - Random selection
    - Cluster growth
    - Hierarchical clustering
  2- Iterative-improvement
    - Min-cut
    - Simulated annealing

Random Selection

- Randomly select nodes one at a time and place them into clusters of fixed size, until the proper size is reached.
- Repeat above procedure until all the nodes have been placed.
- Quality/Performance:
  - Fast and easy to implement.
  - Generally produces poor results.
- Usually used to generate the initial partitions for iterative placement algorithms.

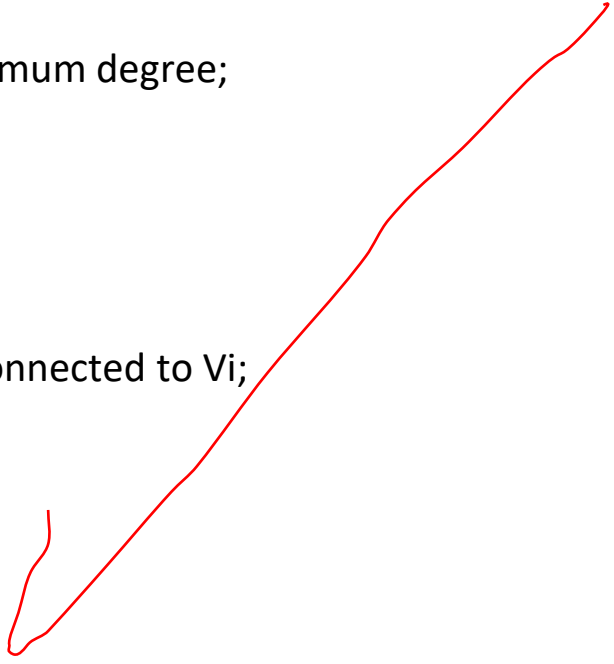
Cluster Growth

m : size of each cluster,

V : set of nodes.

n = lVI / m;

```
for (i=1; i<=n; i++)
    {   seed = vertex In V with maximum degree;
        Vi = (seed);
        V = V — (seed);
        for (J=1; j<m; J++)
        {
          t = vertex in V maximally connected to Vi;
              Vi = VI U {t};
              V = V — (t);
        }
    }
```



Hierarchical Clustering

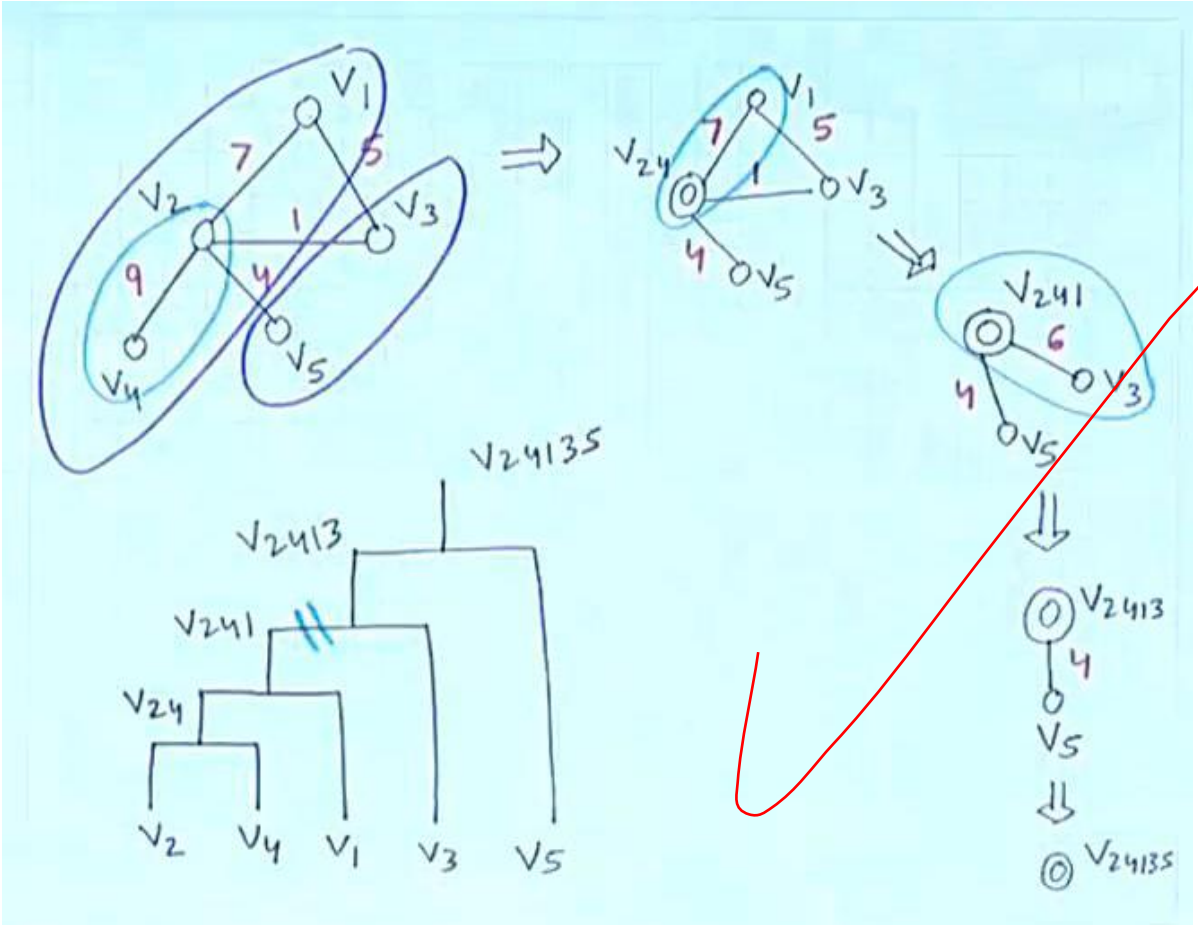
• Consider a set of objects and group them depending of some measure of closeness.

- The two closest objects are clustered first, and considered to be a single object for further partitioning.

- The process continues by grouping two individual objects, or an object or cluster with another cluster.

- We stop when a single cluster is generated and a hierarchical cluster tree has been formed.

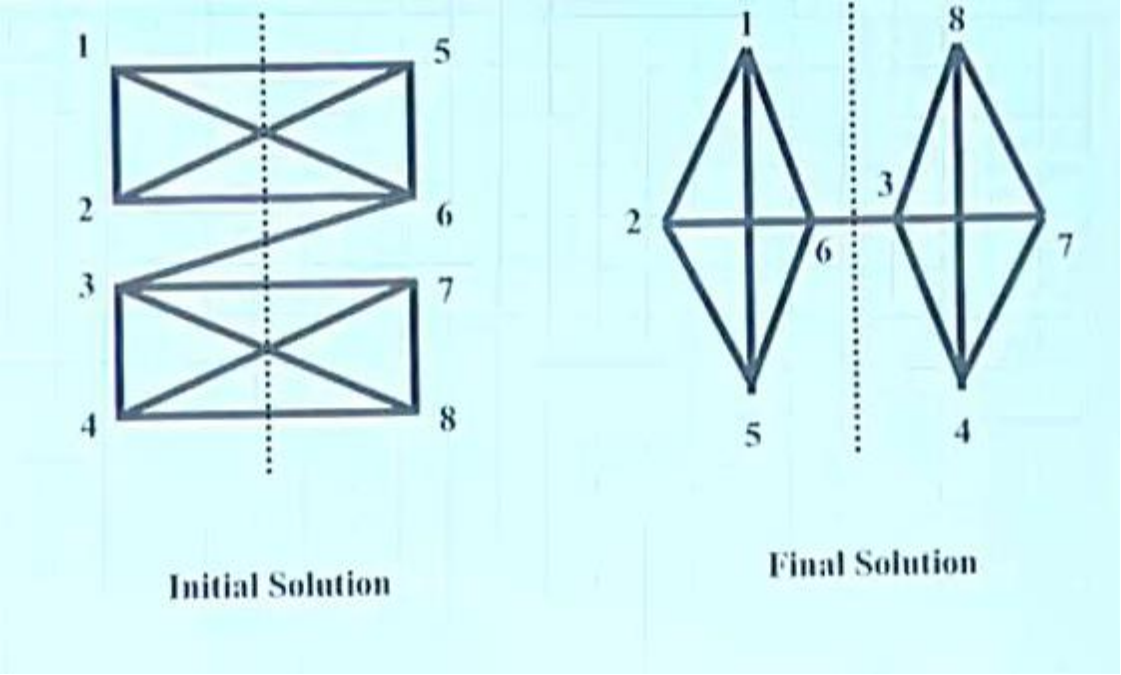
    • The tree can be cut in any way to get clusters.

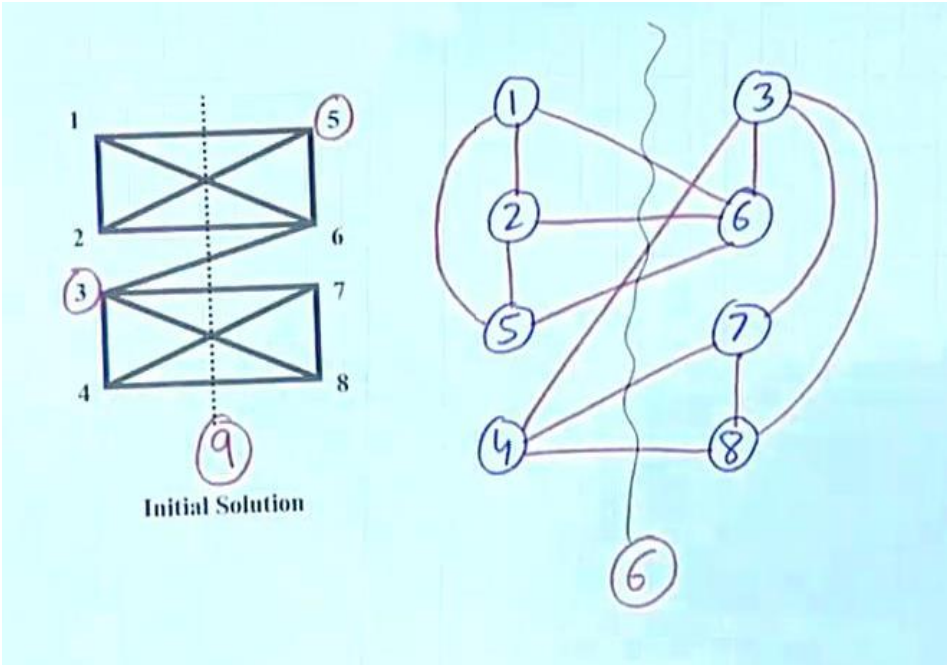

V1
V2
V3
V4
V5
7
5
1
9
4
V24
V241
V2413
V24135
6

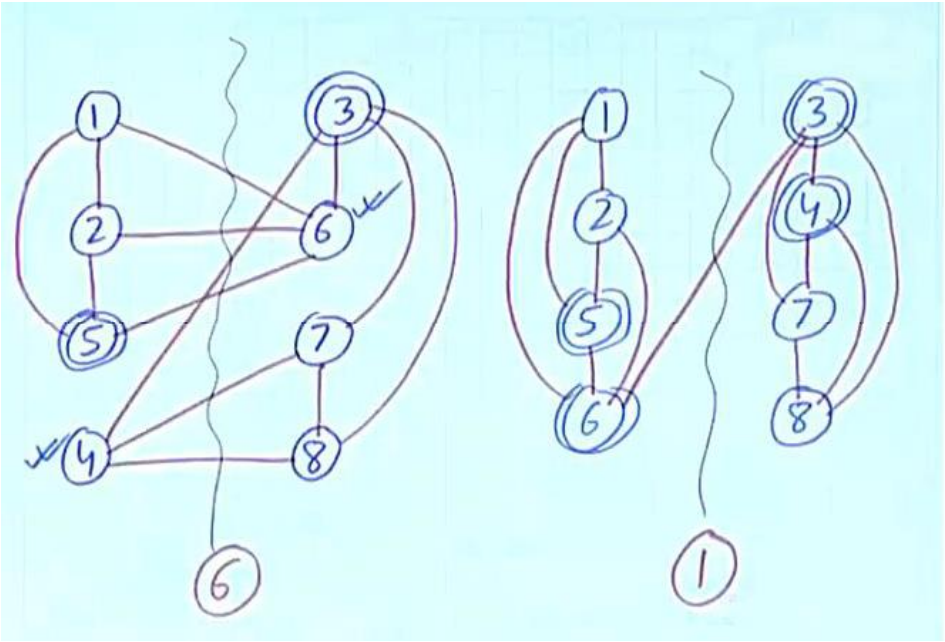
Min-Cut Algorithm (Kernighan-Lin)

- Basically a bisection algorithm.
  - The input graph is partitioned into two subsets of equal sizes.
- Till the cut-sets keep improving:
  - Vertex pairs which give the largest decrease in cut-size are exchanged.
  - These vertices are then locked.
  - If no improvement is possible and some vertices are still unlocked, the vertices which give the sm allest increase are exchanged.

Example:




1
5
2
6
3
7
4
8
9
Initial Solution
1
3
2
6
5
7
4
8
6


1
2
3
4
5
6
7
8
6
1
2
3
4
5
6
7
8
1

- **Drawbacks of K-L Algorithm**

- It is not applicable for hyper-graphs.
  - It considers edges instead of hyper-edges.
  - It cannot handle arbitrarily weighted graphs.
  - Partition sizes have to be specified a priori.

- Time complexity is high.
  - $0(n^3)$.
- It considers balanced partitions only.

Simulated Annealing

- Iterative improvement algorithm.
  - Simulates the annealing process in metals.
  - Parameters:
    - Solution representation
    - Cost function
    - Moves
    - Termination condition
    - Randomized algorithm

- To be discussed later.

What is Scheduling?

• Task of assigning behavioral operators to control steps.

- Input: • CDFG

— Output:

• Temporal ordering of individual operations (FSM states)

• Basic Objective:

- Obtain the fastest design within constraints (exploit parallelism).

Scheduling Algorithms

- Three popular algorithms:
    1. As Soon As Possible (ASAP)
    2. As Late As Possible (ALAP)
    3. Resource Constrained (List scheduling)

As Soon As Possible (ASAP)

• Generated from the DFG by a breadth-first search from the data sources to the sinks.

- Starts with the highest nodes (that have no parents) in the DFG, and assigns time steps in increasing order as it proceeds downwards.

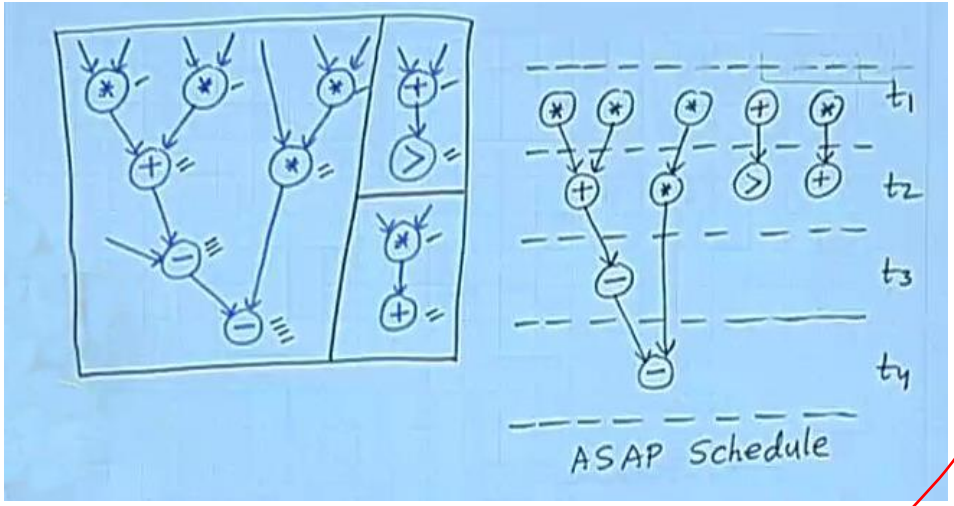
— Follows the simple rule that a successor node can execute only after its Parent has executed.

• Fastest schedule possible

— Requires least number of control steps.

— Does **not** consider resource constraints.

As Soon As Possible (ASAP)



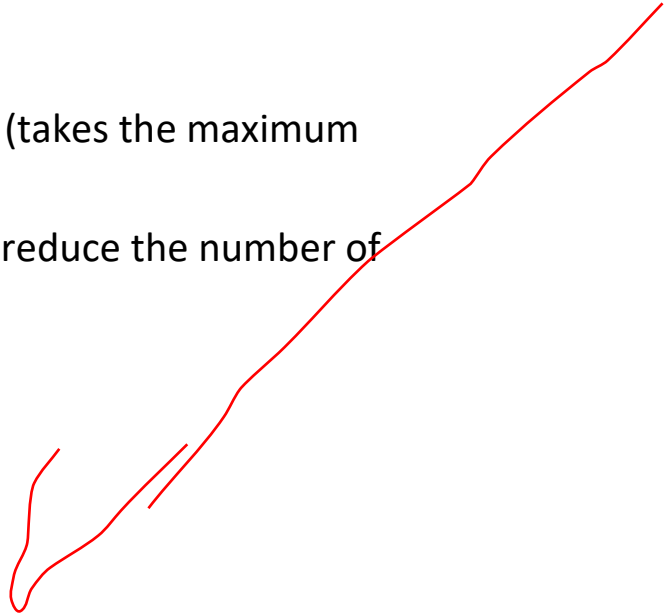
As Late As Possible (ALAP)

• Works very similar to the ASAP algorithm, except that it starts at the bottom of the DFG and proceeds upwards.

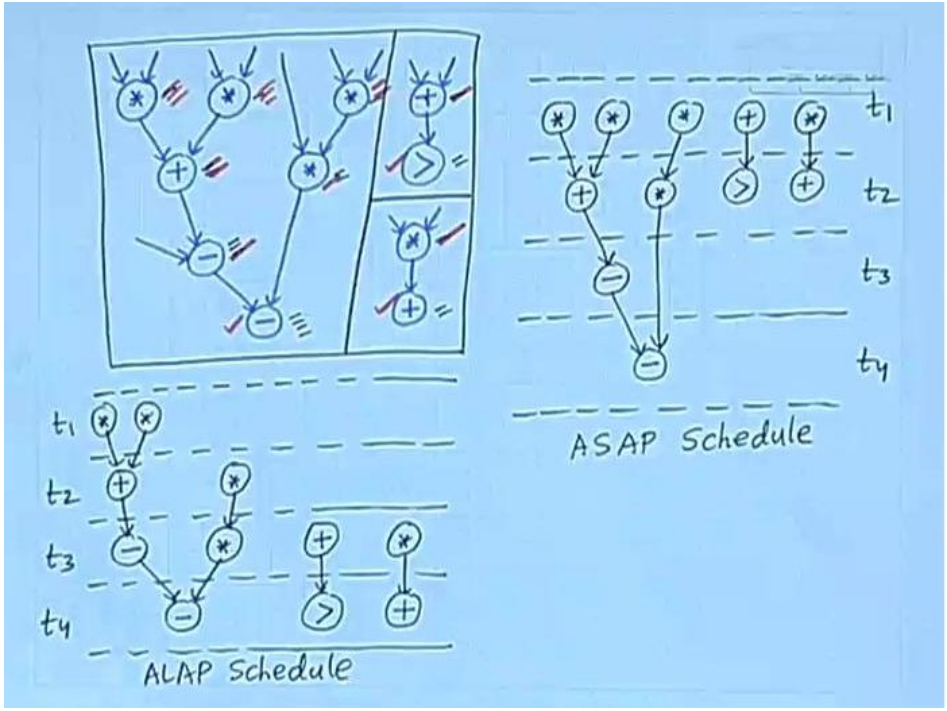
• Usually gives a bad solution:

- Slowest possible schedule (takes the maximum number of control steps).

-Also does not necessarily reduce the number of functional units needed.



As Late As Possible (ALAP)



Resource Constrained Scheduling

There is a constraint on the number of resources that can be used.

— List-Based Scheduling

- One of the most popular methods.
- Generalization of ASAP scheduling, since it produces the same result in absence of resource constraints.

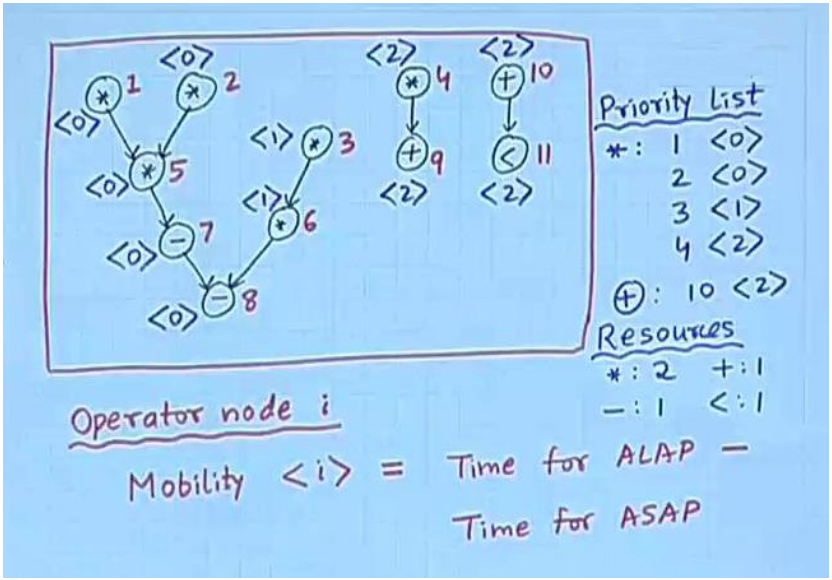
- Basic idea of List-Based Scheduling:

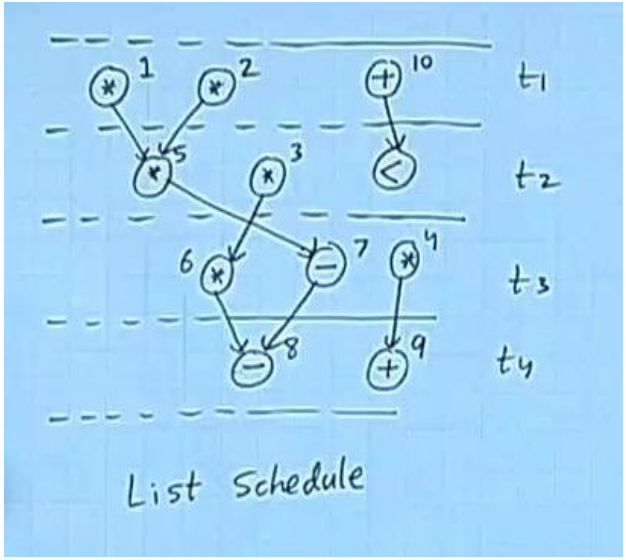
• Maintains a priority list of "ready" nodes.

• During each iteration, we try to use up all resources in that state by scheduling operations in the head of the list.

• For conflicts, the operator with higher priority will be scheduled first.

Example:


Priority list
*: 1 <0>
2 <0>
3 <1>
4 <2>
+: 10 <2>
Resources
*: 2 +: 1
−: 1 <: 1
Operator node i
Mobility <i> = Time for ALAP − Time for ASAP


t1
t2
t3
t4
List Schedule

ALLOCATION and BINDING

Basic Idea:

• Selection of components to be used in the register transfer level design.

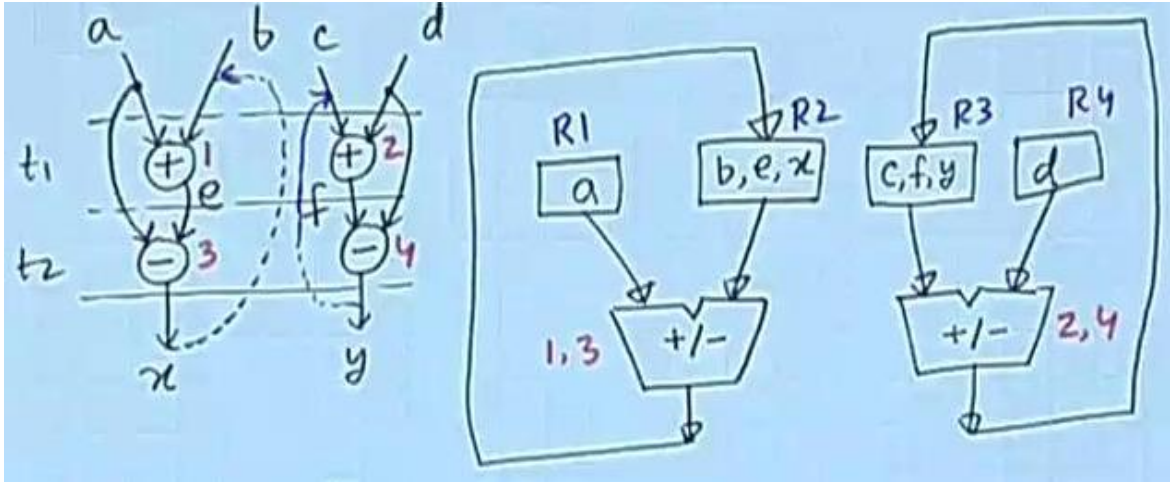
• Binding of hardware structures to behavioral operators and variables.

-Register

- ALU

- Interconnection (MUX)

Example for Binding:



Variable Life Time Analysis

Ninevah University

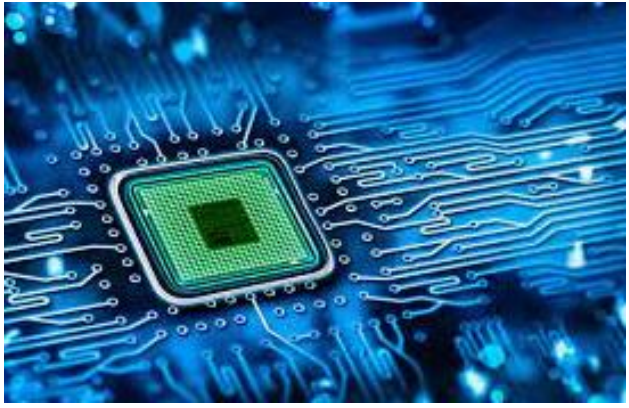
Collage of Electronics Engineering

Course:

Electronic Design Automation

Lecturer: H. M. Hussein

EDA05: Synthesis 2



Introduction

- Representation of Boolean functions

- Canonical
  - Truth table
  - Karnaugh map
  - Set of minterms

- Non-Canonical
  - Sum of products
  - Product of sums
  - Factored form
  - Binary Decision Diagram

Binary Decision Diagram (BDD)

• Proposed by Akers in 1978.

- Several variations suggested subsequently.
    - Ordered BDD (OBDD)
    - Reduced Ordered BDD (ROBDD)
    - A set of reduction rules and operators

defined for BDDs.

• Construction of a BDD is based on the Shannon expansion of a function.

Shannon Expansion

• Given a Boolean function $f(x_1,x_2,...,x_i...,x_n)$

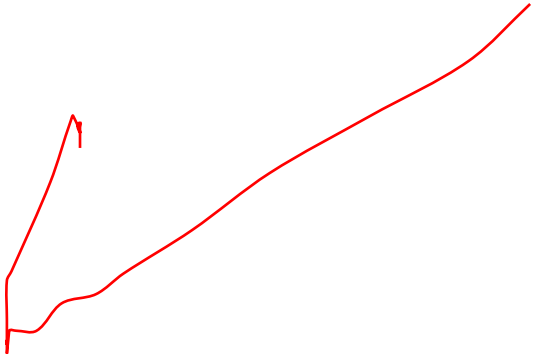
• Positive cofactor fi^1 = f(x1,x2,....,1,…xn

• Negative cofactor fi^0 = f(x1,x2,...,0,...,xn)

- Shannon's expansion theorem states that

$f = x_i' \, fi^0 + x_i \, fi^1$

$f = (x_i + fi^0)(x_i' + fi^1)$



f (a,b,c,d) = abc + b'c'd+ a'bd'

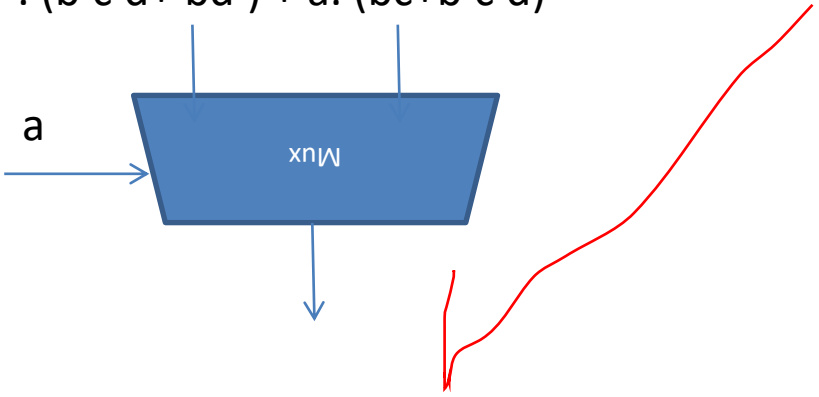
Expand with respect to a:

f= abc + a'b'c'd + ab'c'd + a'bd'

= a'(b'c'd + bd') + a (bc +b'c'd)

f= a'. f(0, b,c,d) + a. f(1,b,c,d)

= a' . (b'c'd+ bd') + a. (bc+b'c'd)

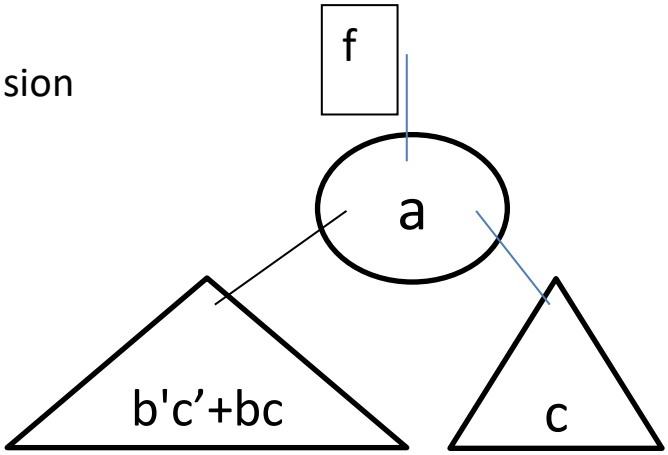


How to construct BDD?

f = ac + bc + a'b'c' with Shannon's expansion

= a' (b'c' + bc) + a (c + bc)

= a' (b'c' + bc) + a (c)



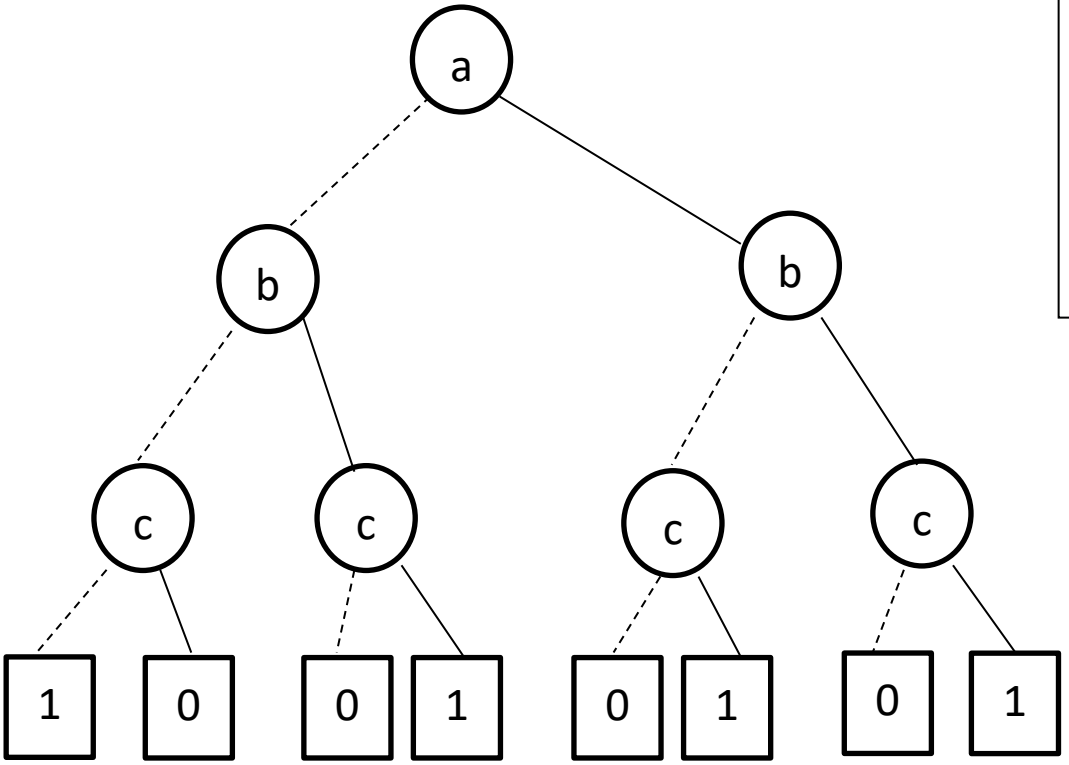
This is the first step. The process is continued for all input variables.

b'c' + bc

b'c+bc -> c

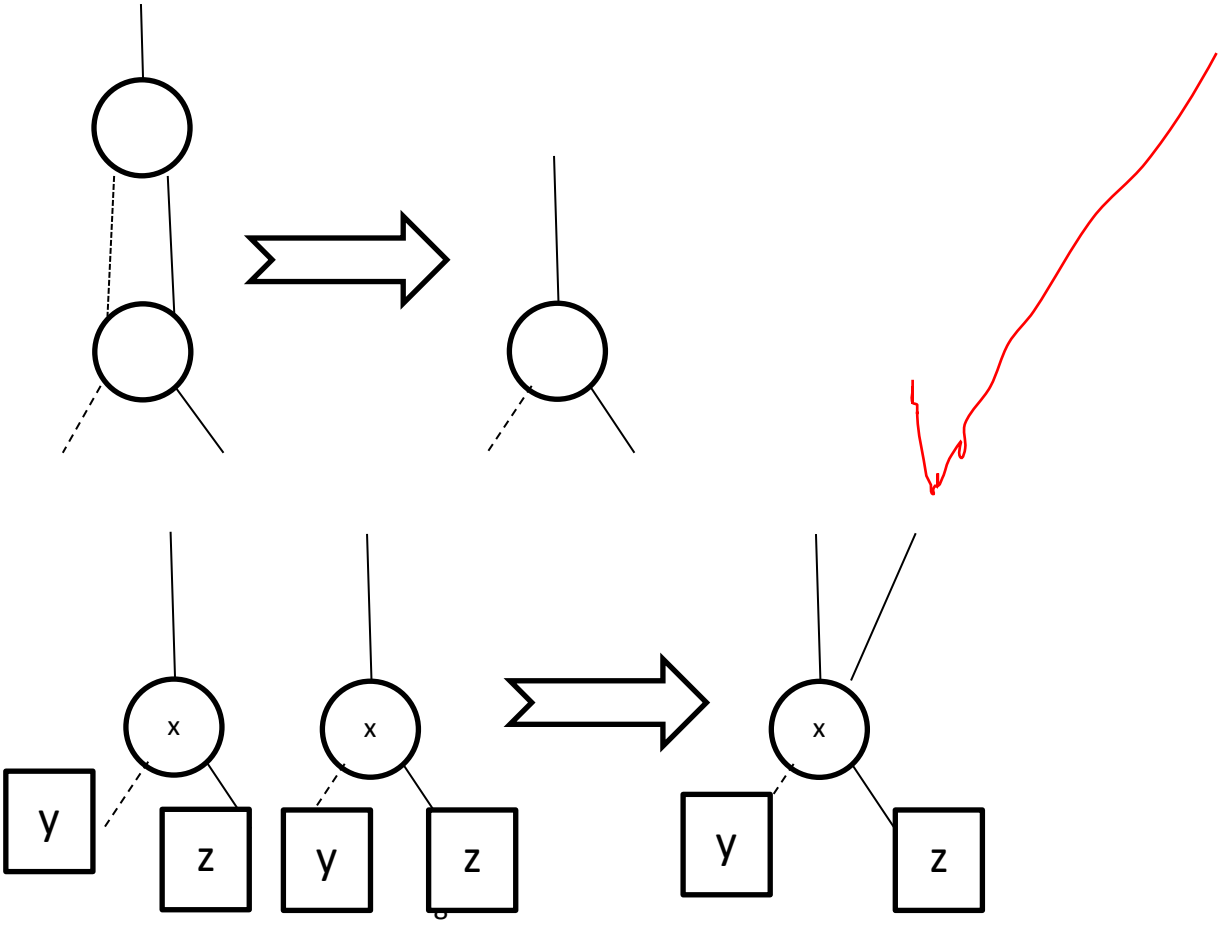
c'

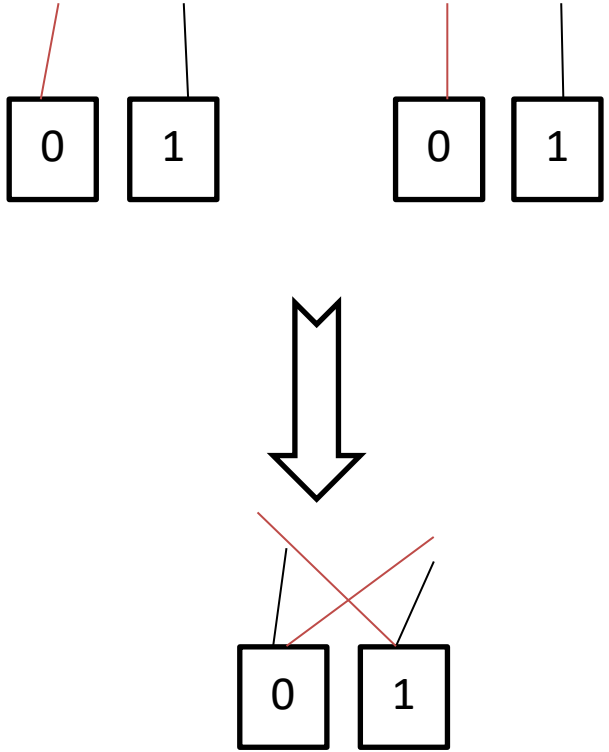
c'.1 + c.0

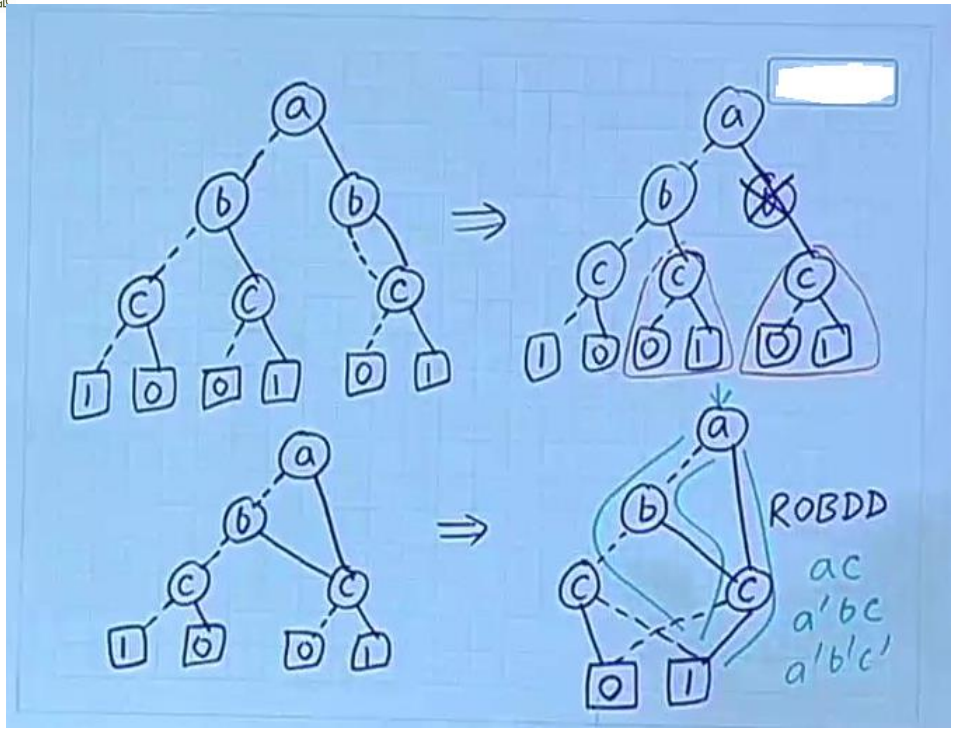


BDD depends on variable ordering OBDD

Reduction Rules:




0
1
0
1
0
1

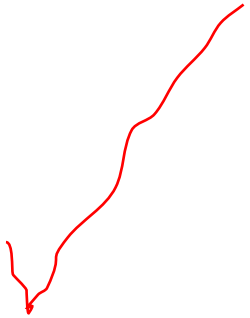

a
b
b
c
c
c
1
0
0
1
0
1
a
b
c
c
1
0
0
1
a
b
c
c
1
0
0
1
a
b
c
c
0
1
ROBDD
ac
a'bc
a'b'c'

Some Benefits of BDD

- Check for tautology is trivial.
  - BDD is a constant 1.
- Complementation.
  - Given a BDD for a function f, the BDD for f' can be obtained by interchanging the terminal nodes.
- Equivalence check.
  - Two functions f and g are equivalent if their BDDs (under the same variable ordering) are the same.

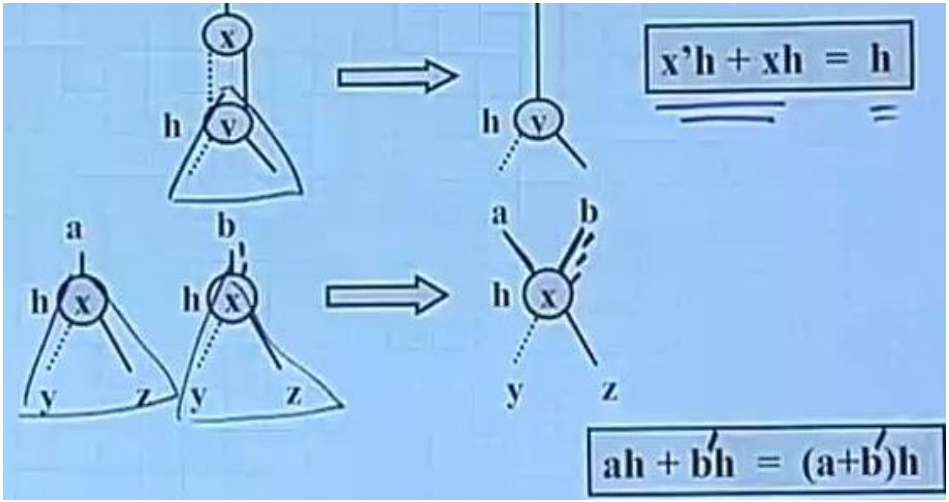
An Important Point

- The size of a BDD can vary drastically if the order in which the variables are expanded is changed.
- The number of nodes in the BDD can be exponential in the number of variables in the worst case, even after reduction.

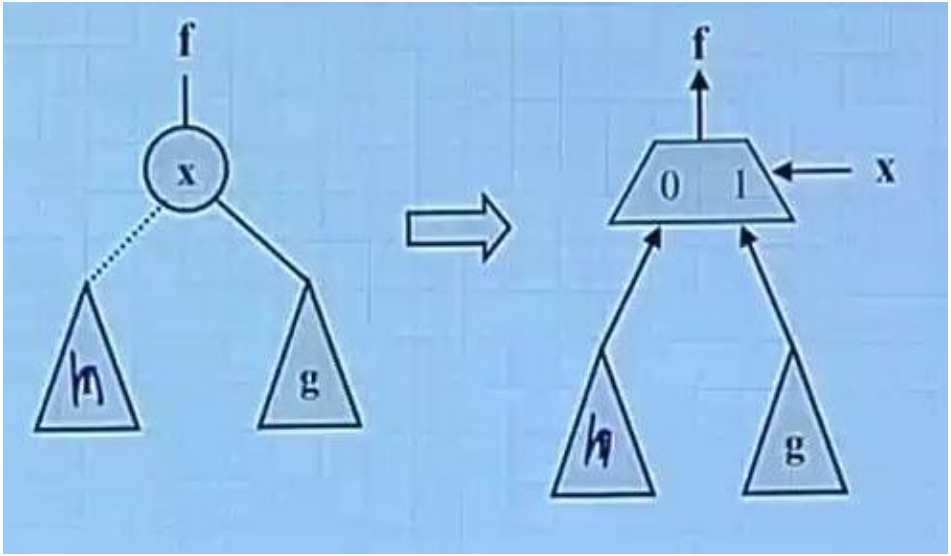


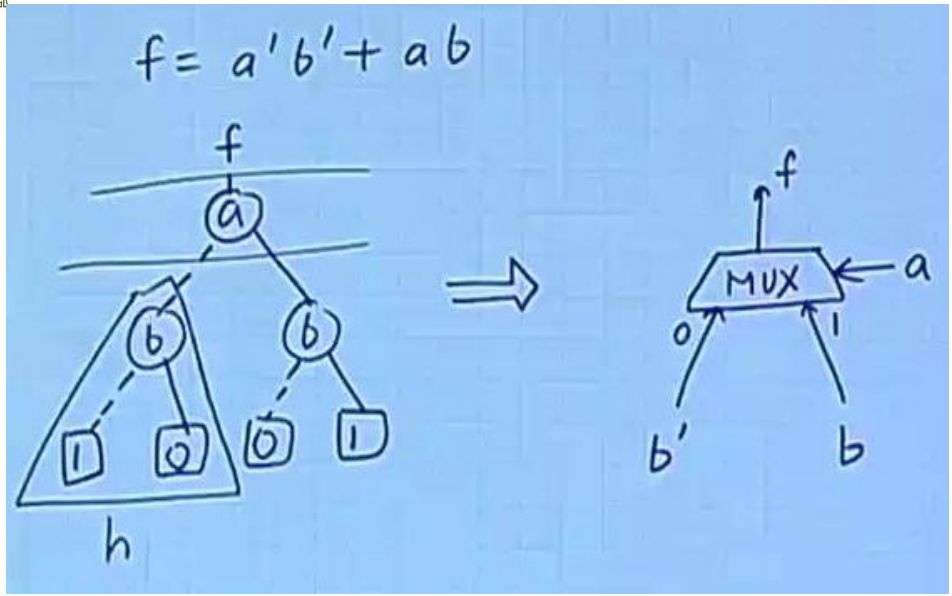
Use of BDD in Synthesis

- BDD is canonical for a given variable ordering.
- It implicitly uses factored representation:

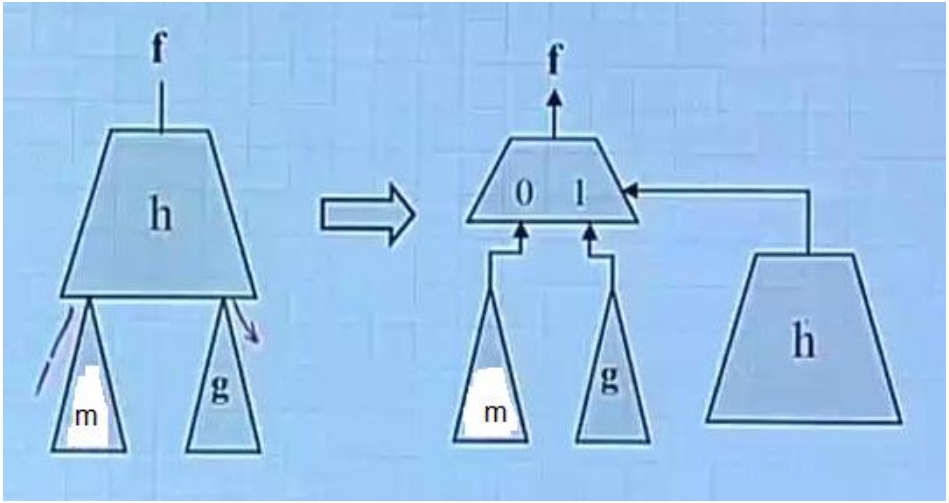


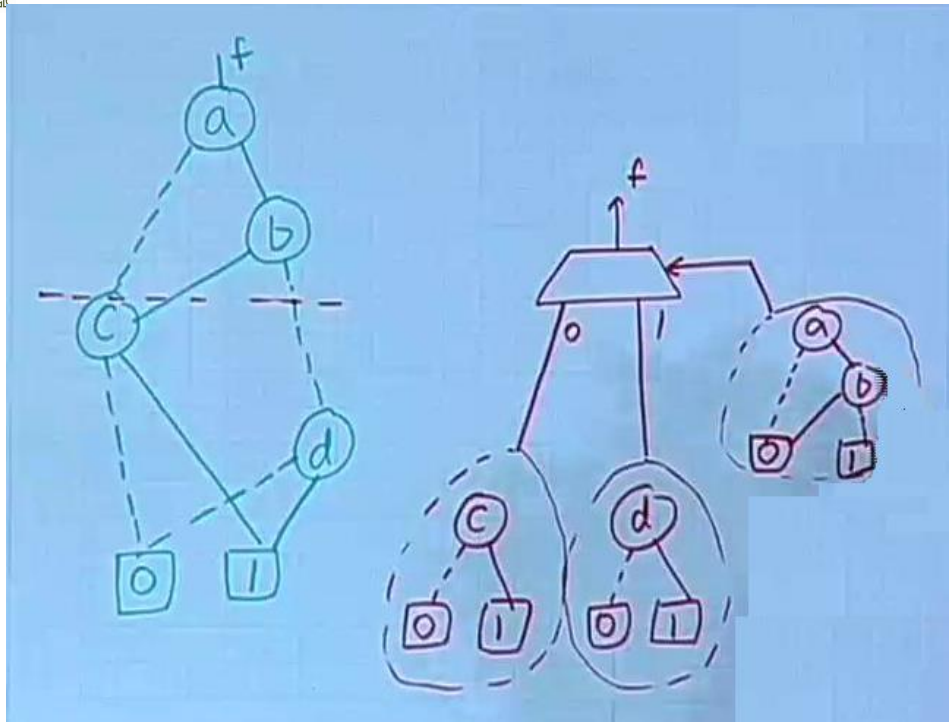
MUX realization of functions:

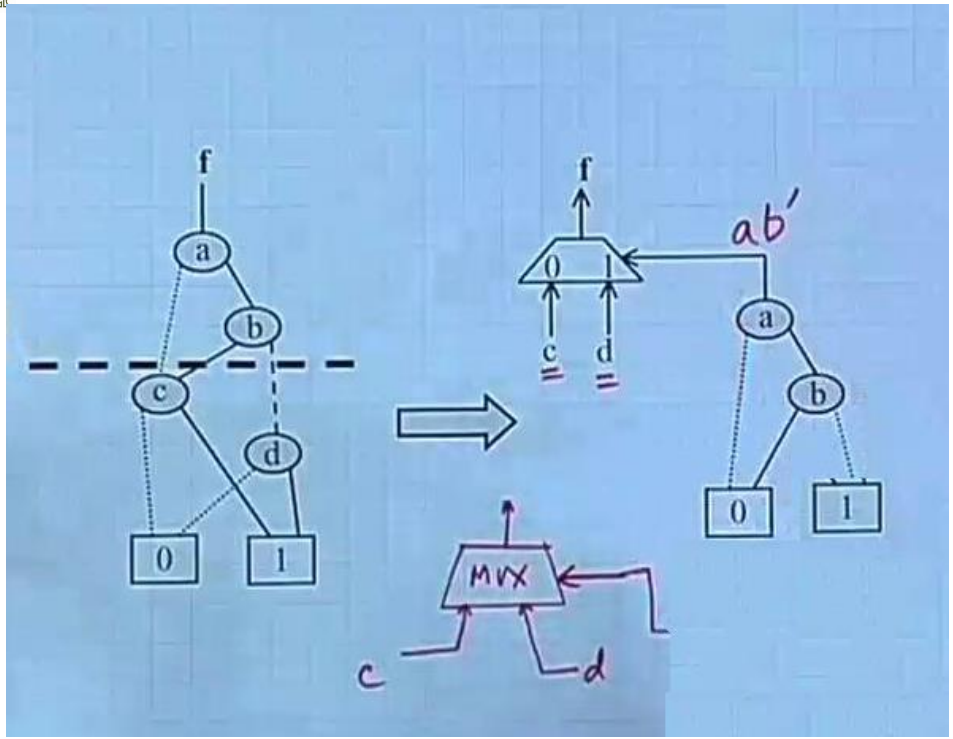



f= a'b'+ ab
f
a
b
b
1
0
0
1
h
f
MUX
a
0
1
b'
b

MUX- based Functional Decomposition:




f
a
b
c
d
0
1
f
0
1
a
b
c
d
0
1
0
1
0
1


f
a
b
c
d
0
1
f
ab′
0
1
c
d
a
b
0
1
MUX
c
d

To Summarize

• BDDs have been used traditionally to represent and manipulate Boolean functions.

- Used in synthesis systems.
- Used in formal verification tools.
- Efficient packages to manipulate BDDs are available.

Ninevah University

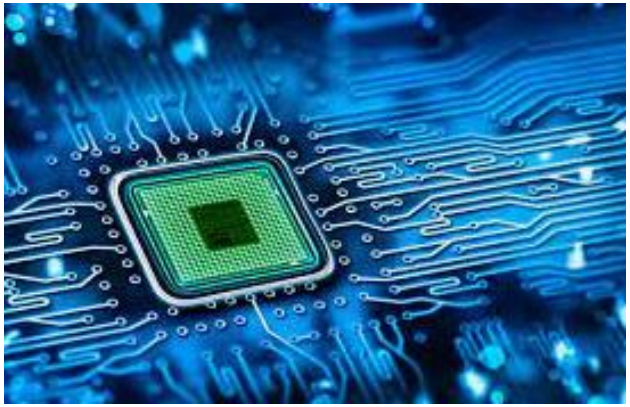
Collage of Electronics Engineering

Course:

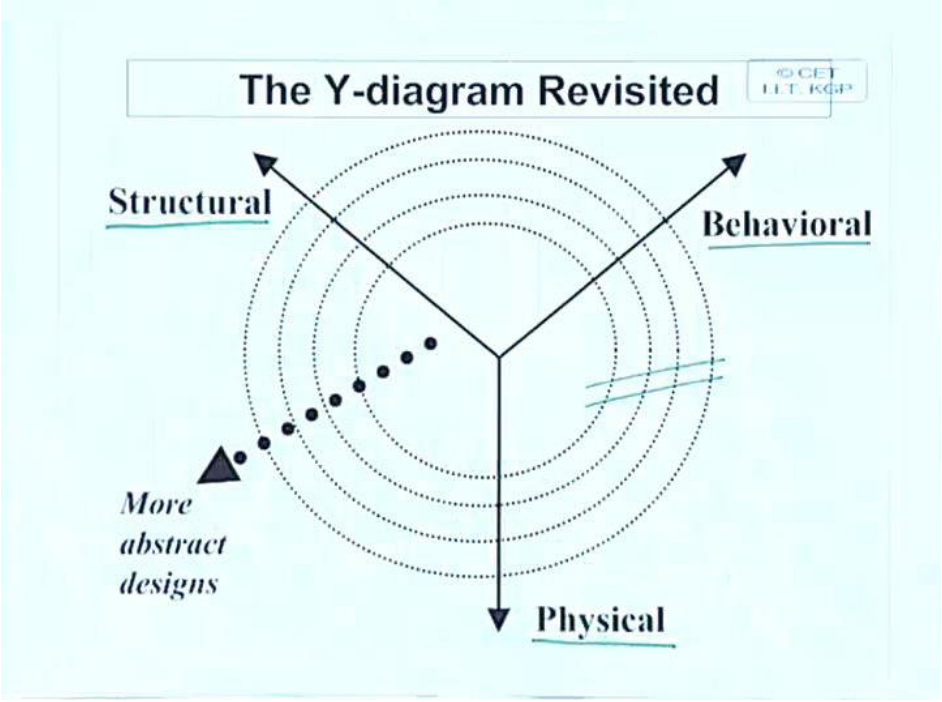
Electronic Design Automation

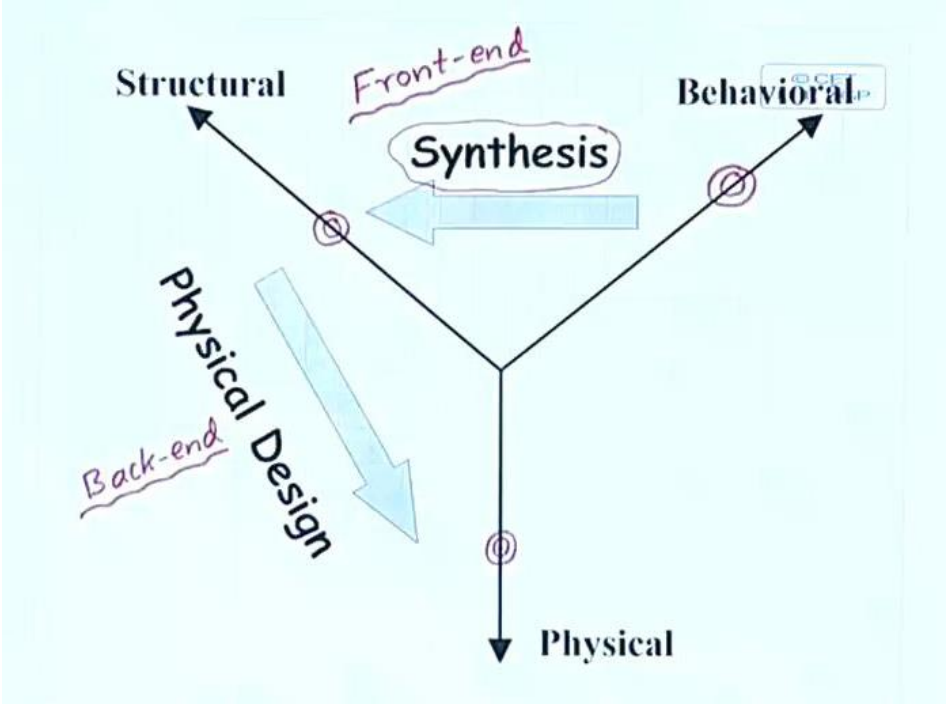
Lecturer: H. M. Hussein

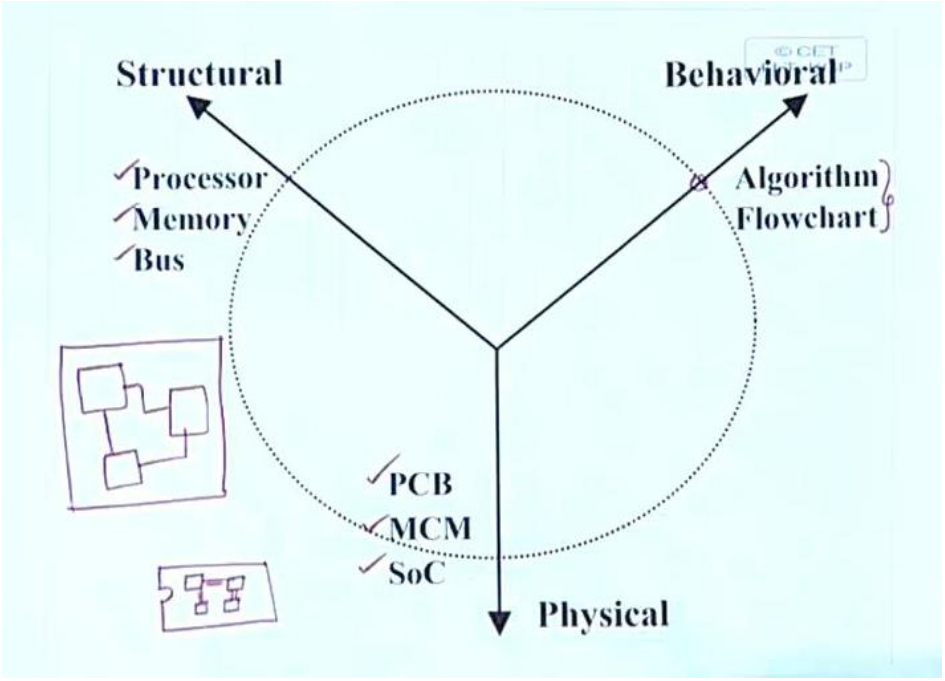
EDA02: Synthesis 1

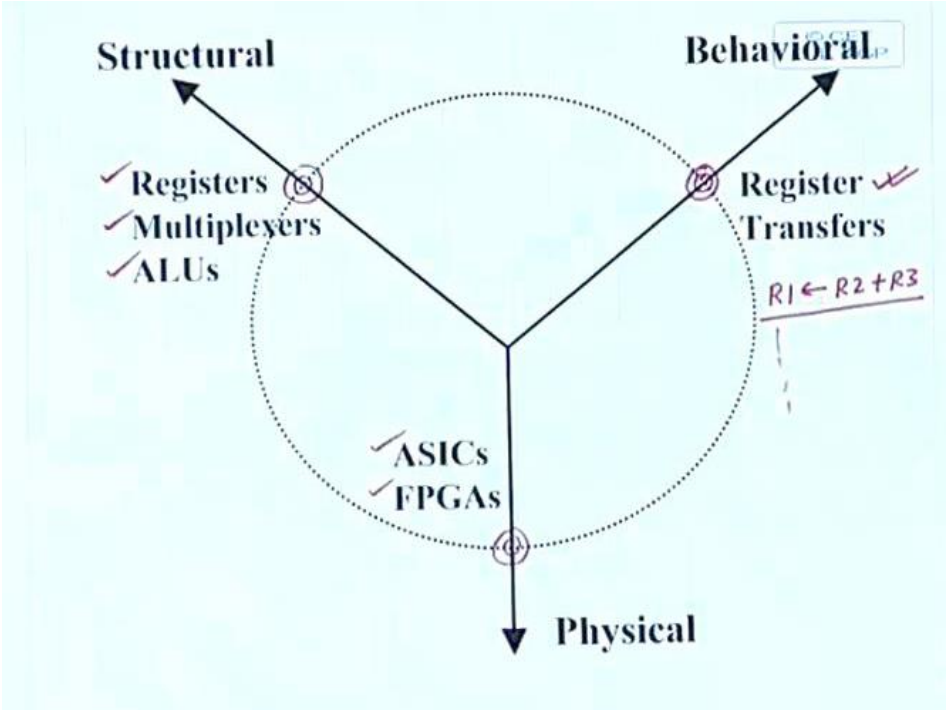


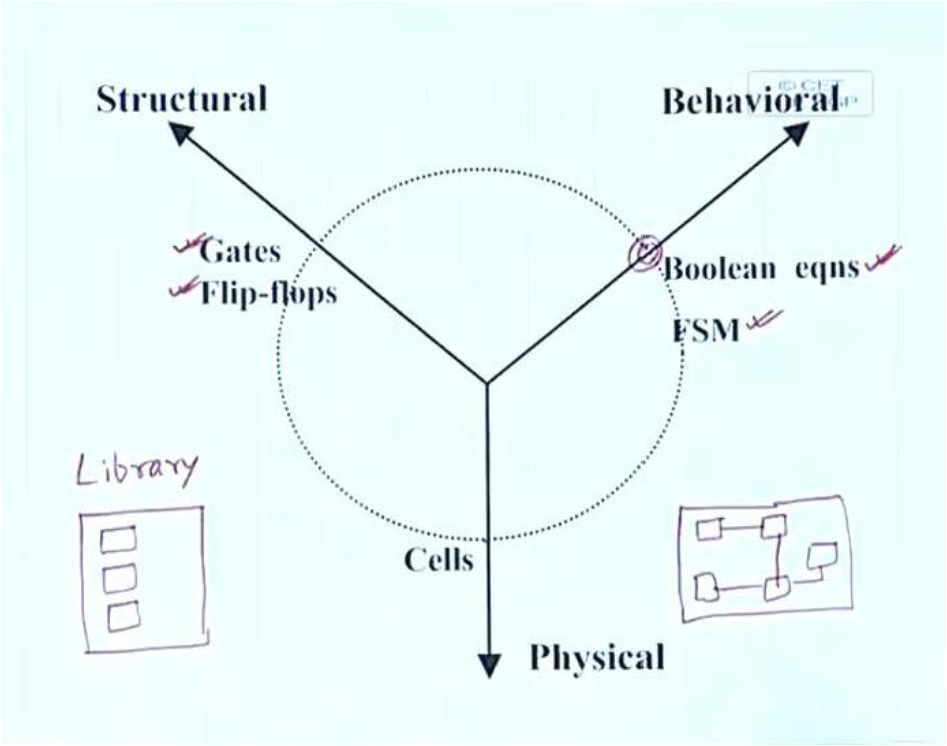
The Y-diagram Revisited

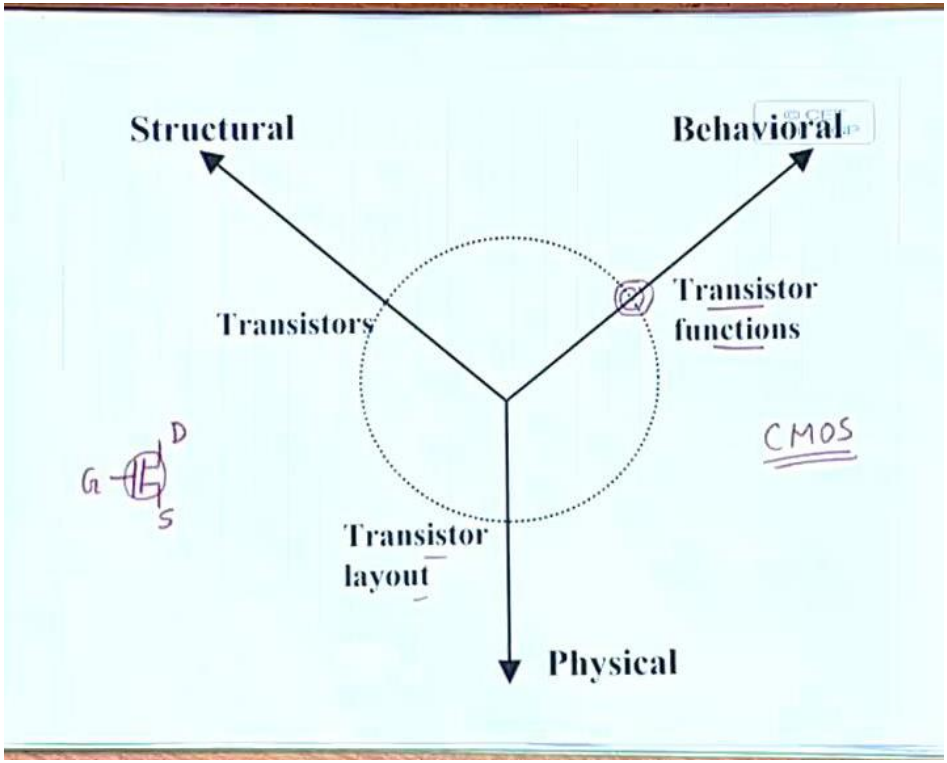


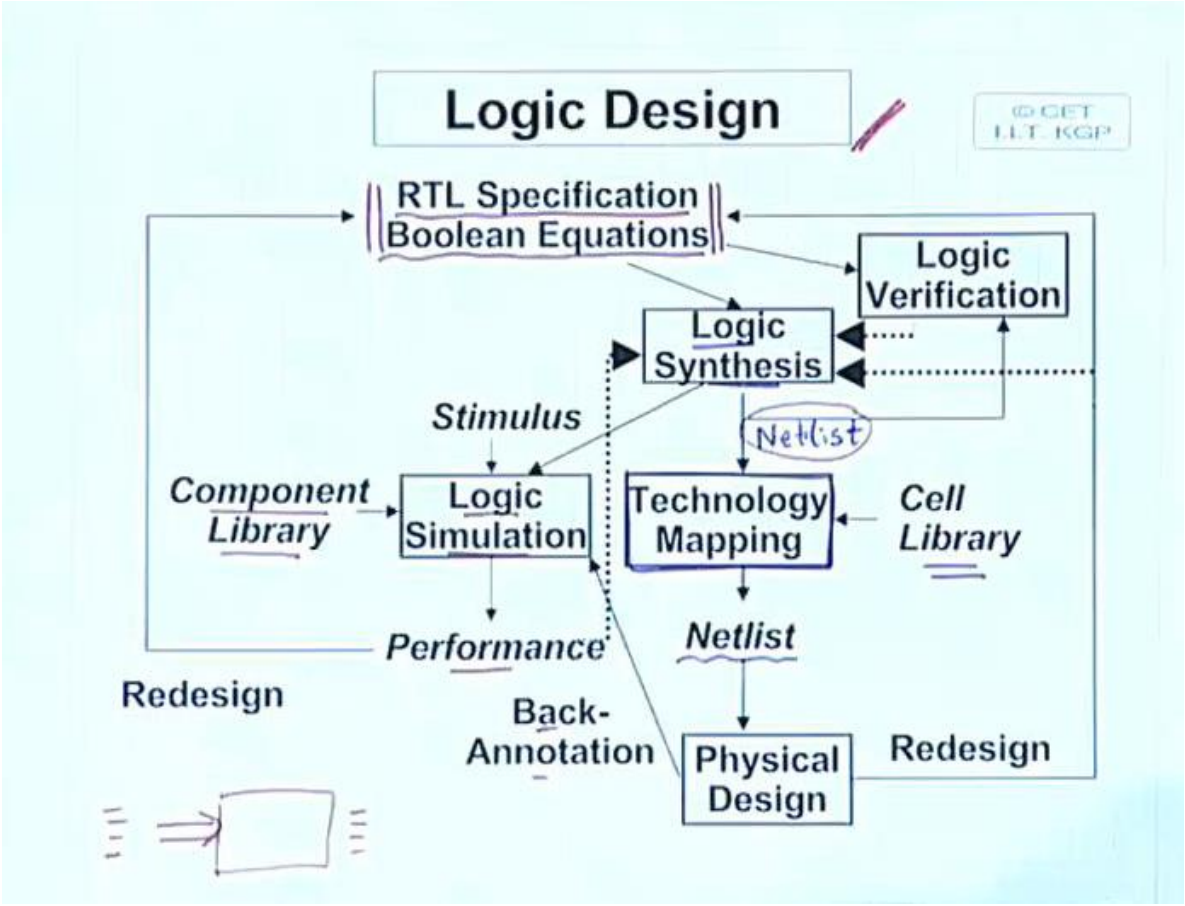

Structural
Front-end
Behavioral
Synthesis
Physical Design
Back-end
Physical


Structural
Behavioral
Processor
Memory
Bus
Algorithm
Flowchart
PCB
MCM
SoC
Physical


Structural
Behavioral
Registers
Multiplexers
ALUs
Register
Transfers
R1 ← R2 + R3
ASICs
FPGAs
Physical


Structural
Behavioral
Gates
Flip-flops
Boolean eqns
FSM
Library
Cells
Physical


Structural
Behavioral
Transistors
Transistor functions
CMOS
G
D
S
Transistor layout
Physical


Logic Design
© CET
I.I.T. KGP
RTL Specification
Boolean Equations
Logic Verification
Logic Synthesis
Stimulus
Netlist
Component Library
Logic Simulation
Technology Mapping
Cell Library
Performance
Netlist
Redesign
Back-Annotation
Physical Design
Redesign

LOGIC SIMULATION

- Takes a logic level netlist as input, and simulate functional behavior.
  - "Netlist" obtained from schematic capture or synthesis.
  - For simulation, the behavior of components is used.
    - Available from component library
    - Gates, flip-flops, MUX, registers, adder
- Ability to handle large circuits (millions of gates)
  - Should be very fast
  - Hardware accelerators.

• SIMULATION OBJECTIVES

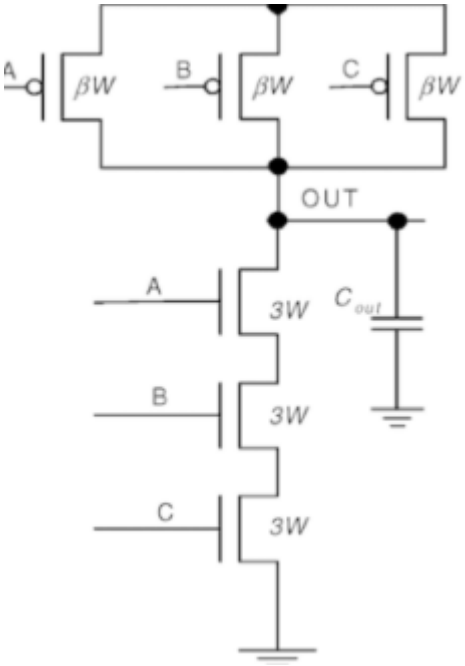
- Functional correctness of the netlist
    - Requires application of a set of test vectors → test bench
- Timing analysis
    - Estimation of delay, critical paths
    - Hazards, races, etc.
- Test generation
    - Required for manufacture test.

Logic Synthesis

- Input: Boolean equations and FSMs
- Output: A netlist of gates and flip-flops
  - Combinational circuits and sequential circuits are typically handled separately
- Design Goals:
  - Minimize number of levels (delay)
  - Minimize number of gates (area)
  - Minimize signal activity (power)
- Typical Constraints:
  - Target library (say, only NAND, NOT gates)

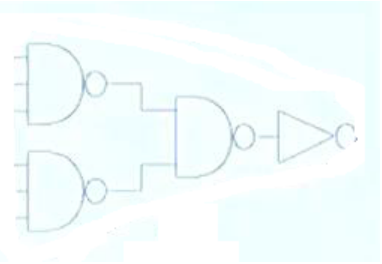
Special Considerations

- Ability to handle large circuits within a reasonable amount of time.
  - Problem is known to be NP-complete
- Ability to handle mutually conflicting requirements (area & delay)
- Typically a fully automated process
  - Algorithms/heuristics well understood
  - Do not need user intervention
- Use technology dependent considerations
  - Break a 20-input gate into smaller gates
  - Use gates available in the library

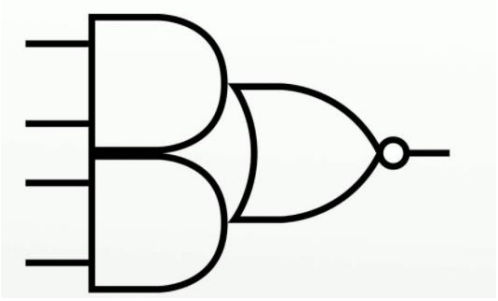


Technology Mapping

- Basic Concept:
  - During logic synthesis, map portions of the netlist to "cells" available in the cell library
    - Standard library (NAND, NOR, NOT, A01)
    - FPGA cells, standard cells
- Objectives:
  - Minimize area, delay, power
  - Should be fast
    - Able to handle large circuits, and large technology libraries.



An Example: AND_OR_INVERT



Logic Verification

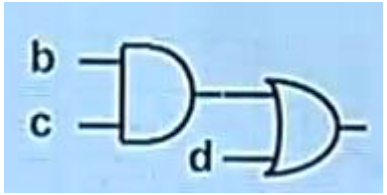
- Verify that the synthesized netlist matches the original specification
  - Detect design errors, also synthesis errors
  - Basic objective is to ensure functional correctness, and to locate errors, if any
- Broadly two approaches:
  1. Simulation
     - Fast, incremental, can handle large circuits
  2. Formal verification
     - Slow, exhaustive, for small circuits only

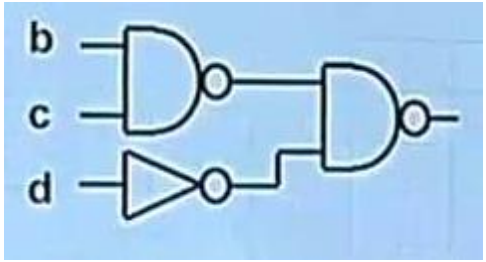
The Basic Problem

• Convert from logic equations to gate-level netlists (assume combinational logic).

- Maximize speed
- Minimize area, power

a'bc + abc + d → bc + d

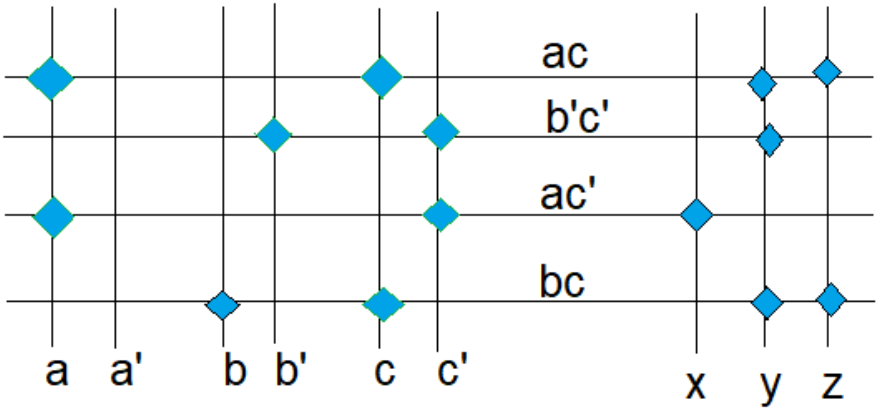




Logic Specification

- PLA Format

.i 3
.o 3
.p 4
1x1 011
x00 010
1x0 100
x11 011
.e



- Sum-of-product form

x = ac'

y = ac + b'c' + bc

z = ac + bc

Logic Synthesis Problem

1. Simplification of logic equations
    - Reduce number of literals (and operands)
2. Synthesis
    - Map logic equations to gates (AND, OR, etc)
3. Gate-level optimization
    - Replace OR-NOT by NOR, for example
    - Delay, power, area
4. Technology mapping
    - Map from gates to technology library
    - FPGA, TTL chips, standard cells, etc.

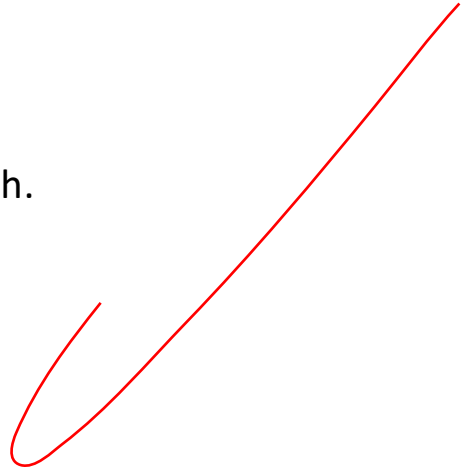
Two-level Minimization

- Karnaugh Maps

- For n inputs, the map contains 2^n entries
- Objective is to find minimum prime cover
    - Minimum → fewest terms
    - Prime → choose only maximal covers
- Don't care terms are used to advantage
- Difficult to automate
    - Minimum cover problem is NP-complete
    - Process can get into a local minima

• Problems with K-maps:

- Number of cells is exponential in the number of input variables.
  - Imagine a 50-input circuit.
- Requires efficient data structures
  - For representing the function
  - For searching for minimal prime cover
- Quine-McCluskey method
  - Easy to implement in software.
  - Computational complexity remains high.



Espresso: A 2-level logic optimizer

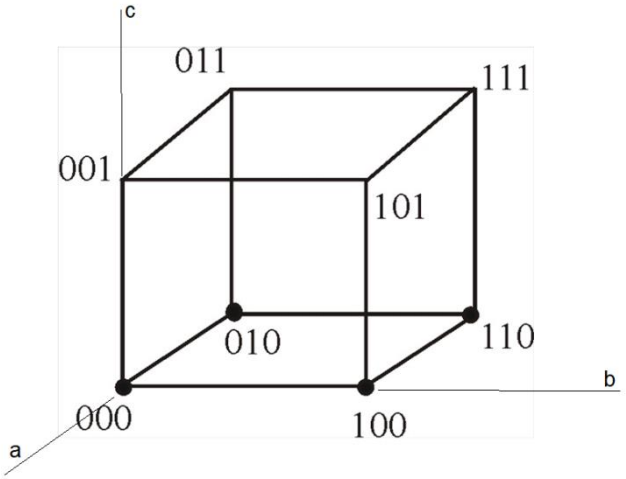
• Some notations:

— For an n-input function, n-dimensional Boolean space

- • Each point mapped to a unique combination of the n literals.
- • Entries in K-map, minterm.

— Cube:

- • Conjunction (AND) of literals in an n-dimensional space.
- • Points on the n-dimensional hypercube that are ‘’1’’.

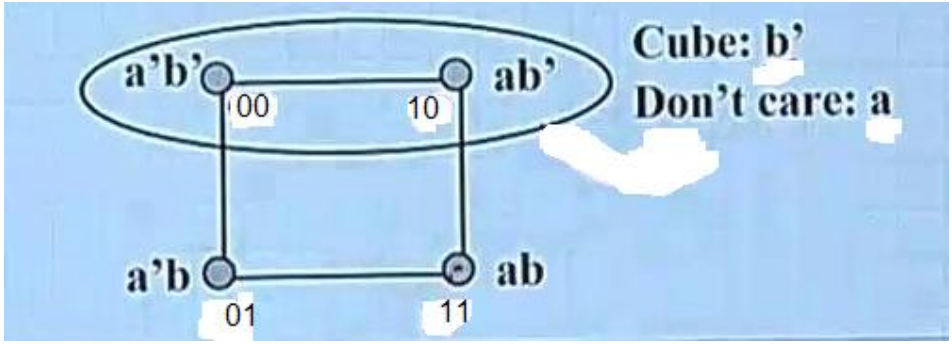


- Expression

  - Disjunction (OR) of cubes

- Don't cares

  - Literals that are missing from a cube



- Basic Approach

- Minimize cover of "ON-set" of the function
    - ON-set → set of vertices that correspond to "1" min-terms
    - Minimum set of cubes
    - Size of the cubes can be increased by exploiting don't care literals

- **The Espresso Algorithm (Outline) :**

Start with the sum-of-products form (i.e., cubes covering the ON-set)

- In an iterative loop
    - Expands
    - Remove redundancy Irredundant
    - Reduce cubes until no further improvement is possible.

- Perturb the solution, and repeat the previous iterative step, as long as the time budget permits.
    - For each cube, add a sub-cube not covered by any other cube.
    - expand sub-cubes and add them if they cover another cube.

Cube operation: expand

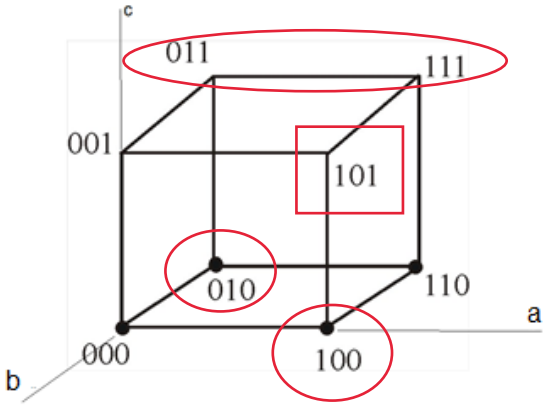
• Make each cube as large as possible without covering a point in the OFF-set.

- Increases the number of literals in the cover.
- Sets the stage for finding a new and possibly better solution.

• Example:

f = a'bc' + bc + ab'c' ;Don't care: ab'c

f = + bc + ac + ab'



Cube operation :: irredundant

- Throw out redundant cubes.
  - Points may be covered by several cubes after the `expand" step.
  - Remove smaller cubes whose points are covered by larger cubes.
  - There must be one cube for every essential vertex.
- Example:

  f = a'b+ bc + ac + ab'

  f = a'b + ac + ab'

One vertex in (bc) is covered by (a'b) & the other by (ac)

Cube operation :: reduce

- The cubes in the cover are reduced in size.
  - The number of literals in the cover is reduced.
  - Smaller cubes can expand in more directions.
  - Smaller cubes are more likely to be covered by other cubes during expansion.
- Example

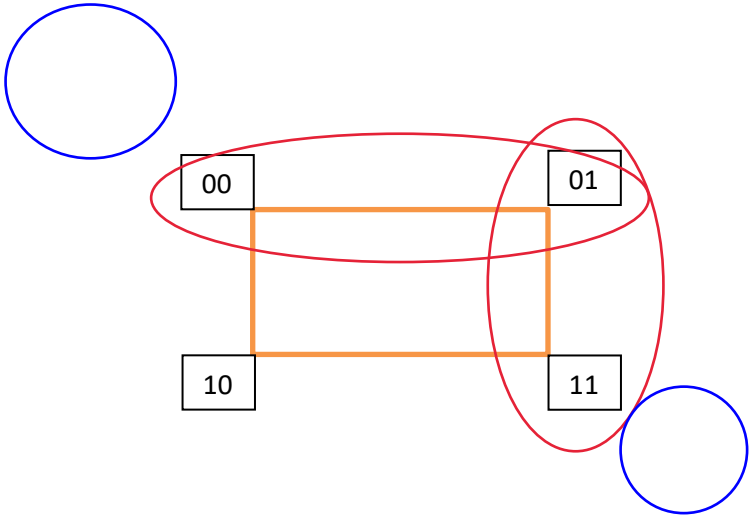
f = a'b + ac + ab'

f = a'b + abc + ab'c'

- In general, the new cover will be different from the initial cover.
    - "expand" and "irredundant" steps can possibly find out a new way to cover the points in the ON-set.
    - Hopefully, the new cover will be smaller.

Cube operation: perturbations

Example:

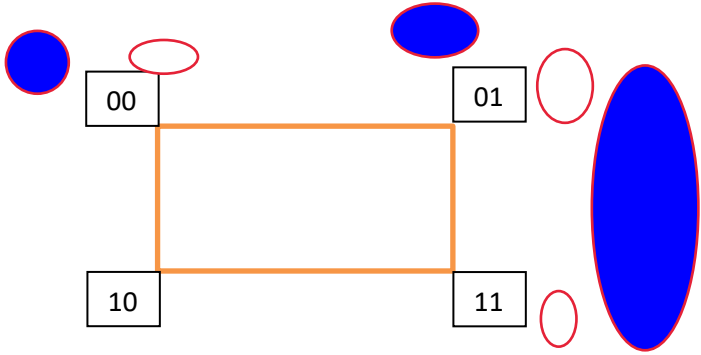


(**Reduce Gasp**)

f = a' + b → f = a' + b + a'b' + ab

(**Expand Gasp**)

f = a'b' + a'b + ab → f = a'b' + a'b + b



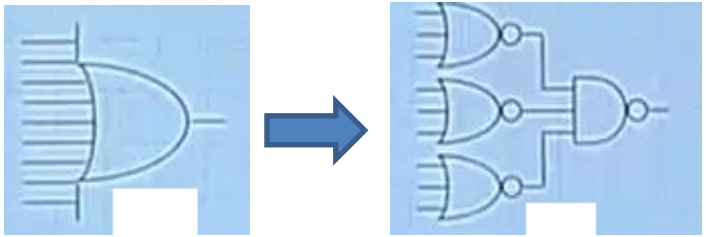
Espresso :: conclusion

- The algorithm successively generates new covers until no further improvement is possible.
- Produces near-optimal solutions.
- Used for PLA minimization, or as a sub-function in multilevel logic minimization.
- Can process very large circuits.
  - 10,000 literals, 100 inputs, 100 outputs
  - Less than 15 minutes on a high-speed workstation

Multilevel Logic Minimization

• In many applications, 2-level logic is unsuitable as compared to random (multilevel) logic.

- Gates with high fan-in are slow, and take more area.
- It makes sense to transform a 2-level logic realization to multi-level logic.



- **A classic example:: XOR function**
  - For an 8-input XOR function,
    - For 2-level NAND-NAND realization

$^8C_1 + {}^8C_3 + {}^8C_5 + {}^8C_7$ = 128 NAND8 gates

1 NAND128 gate

- For 3-level XOR realization

7 XOR2 gates

→ 28 NAND2 gates

Number of levels = 9

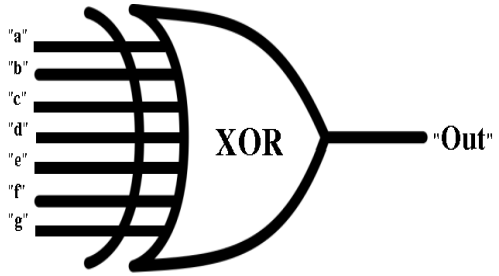
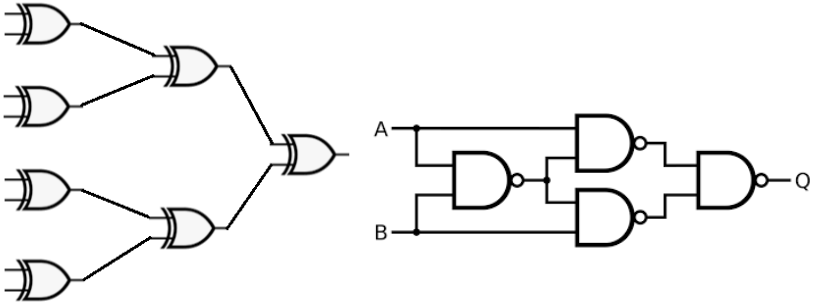


Fig. 16. Schematic of 8-input "XOR" gate.

TABLE IV. TRUTH TABLE OF 8-INPUT "XOR" GATE

a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*Out*
0	0	0	0	0	0	0	0	0
-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-
1	1	1	1	1	1	1	1	0



Multilevel logic optimization:

1. Local
   » Rule-based transformation
2. Global
   » Weak division

Local Optimization Technique

- Perform rule-based local transformations.
  - Objective → to reduce area, delay, power.
  - Developing a good set of rules is a challenge.
  - Should be comprehensive enough so as to completely explore the design space.

- Basic idea:
  - Apply a transformation which reduces cost.
  - Iterate and continue applying transformations as long as solution keeps improving.

- **AND/OR transformations**

- Reduce the size of the circuit, critical path.

-Typical transformations:

a . 1 = a

a + 1 = 1

a + a' = 1

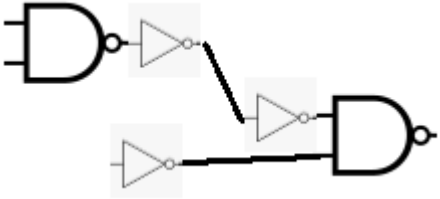
a . a' = 0

(a')' = a

a + a' . b = a + b

xor (xor(a1,a2 .....,an), b) = xor(a1,a2,….,an,b)

- Transform the AND/OR form to NAND form (or NOR form).



- NAND (NOR) transformations

- Some synthesis systems assume that all gates are of the same type (NAND or NOR).
- Does not require technology mapping.
- Rules framed that transform a NAND (NOR) network to another.

Examples:

NAND (NOT (NAND (alb)), c) = NAND (a,b,c)

NAND (NAND(a,b,c), NAND(a,b,c')) = NAND(a,b)

Global Optimization Technique

- Used in GE Socrates.
  - Looks at all the equations at one time.
- Perform weak division.
  - Divide out common sub-expressions.
  - Literal count gets reduced.
- The following iterative steps are carried out:
  - Generate the candidate sub-expressions.
  - Select a sub-expression to divide.
  - Divide functions by selected sub-expression.

F1 = ab + ac

= a(b+c) = a. F2

F2 = b+c

Example

- Original equations:

f1= a.b.c + b.c.d + b.e.g

f2 = b.c.f.h + d.g + a.b.g → No. of literals = 18

— We find literals saved for sub-expressions:

b.c →4 a.b → 2

a + d → 2 b.g → 2

Select the sub-expression bc.

- Modified equations (after iteration 1):

f1= (a + d).u + b.e.g

f2 = u.f.h + d.g + a.b.g

u = b.c → No. of literals = 14

f1 = (a + d).u + b.e.g

f2 = u.f.h + d.g + a.b.g

u = b.c

— Literals saved for the sub-expressions: b.g →2

- Modified equations (after iteration 2):

f1= (a + d).u + e.v

f2 = u.f.h + d.g + a.v

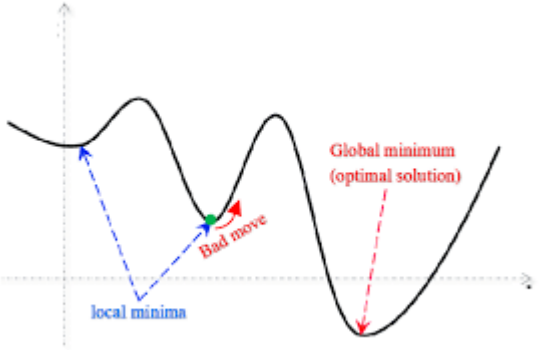
u = b.c

v = b.g → No. of literals = 12

- No common sub-expressions → STOP

About the algorithm

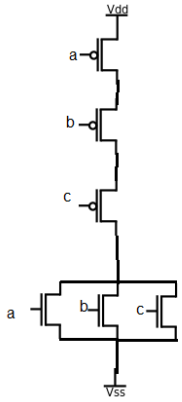
- Basically a greedy algorithm
  - Can get stuck in local minima.
  - Give a "bounce" to come out of local minima.
    - Like the "gasp" function in Espresso. •



— Generation of all candidate expressions is expensive.

- Some heuristic used.

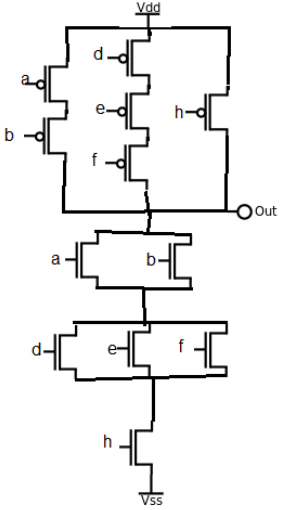
MULTILEVEL LOGIC INTERACTIVE SYNTHESIS SYSTEM (MIS)



- A very popular & widely used algorithm.
  - Uses factoring of equations.
  - Similar to weak division used in Socrates.
  - The target technology is CMOS gate.
  - Complex gates realizing any complex functions.
  - Example:

    f’ = (a + b + c)

    g' = (a + b) . (d + e + f) h



Basic Concept

- For **global** optimization,

  - Use algebraic factorization to identify common sub-expressions.
  - Avoid exponential search.

- For **local** optimization,

  - Identify 2-level sub-circuits.
  - Minimize them using Espresso, or some similar approach.

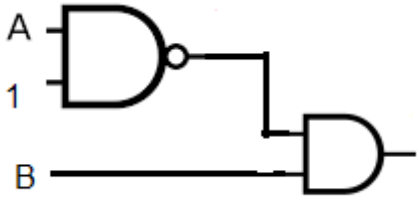
Global Optimization Approach

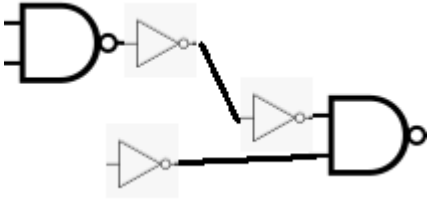
• Given a netlist of gates

- Scan the network

- Apply simple heuristics to "clean up" the netlist.
    - Constant propagation
    - Double inverter elimination
    - Espresso minimization of each equation:

- Then proceed for global optimization with a view to minimize area.





— Basically an iterative approach.

- Enumerate all common factors and identify the "best" candidate.
- Equations themselves may be common factors.
- Invert an equation if it helps.

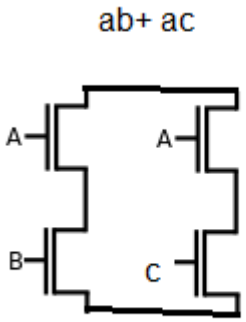
  f=a + b + c → f'=a' b' c'

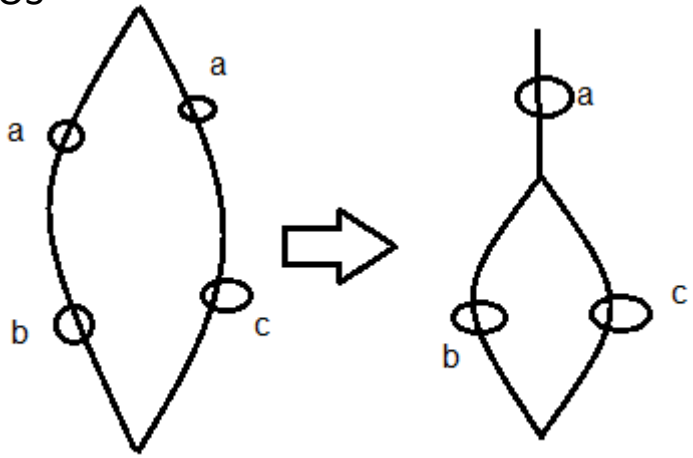
— Factors may show up in the inverted form.

- Number of literals used to estimate area.

Some Illustrative Examples

- Factoring can reduce area.
  - An equation in simple sum-of-products form can have many literals.
    - Many transistors for CMOS realization.
  - Factoring the equation reduces the number of literals.
  - Reduces number of transistors in CMOS realization.





f = a b e’ f + a b g + a c e’ f + a c g + a d e’ f + a c d g

→ 22 literals → 44 transistors

f = (a (b + c) + d) (e’ f + g (b + c))

➔ 9 literals → 18 transistors

Common sub-expressions:

f= a b c + a d g + a c’ f

g= a c’ d + a d f (15 literals, 30 t)

u= ac’

f= u b + a d g + u f

g= u d + a d f (14 literals, 28 t)

u= ac’

f= u (b + f) + a d g

g= d(u + a f) (12 literals, 24 t)

Area: Minimum no. of transistors.

Delay: Number of levels would be reduced

- No division
- Only factorization.

Power: signal activity.

Ninevah University

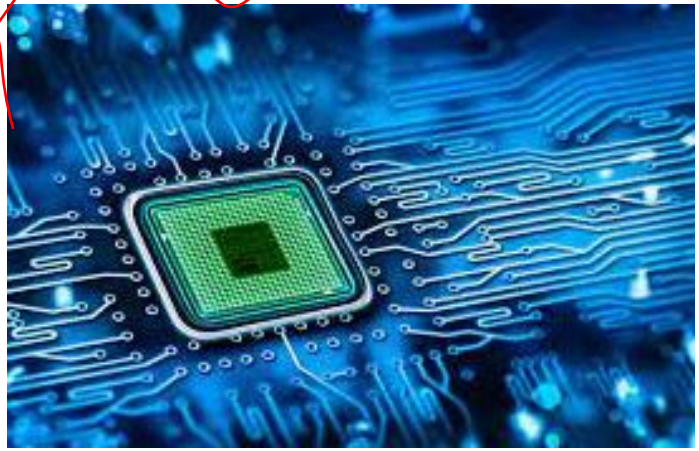
Collage of Electronics Engineering

Course:

Electronic Design Automation

Lecturer: H. M. Hussein

EDA02: Verilog 2



Blocking & Non-blocking Assignments

. Sequential statements within procedural blocks (“always” and “initial”) can use two types of assignment:

- Blocking assignment: =
- No-blocking assignment: <=

Blocking assignment: (using =)

. Most commonly used type.

• The target of assignment gets updated before the next sequential statement in the procedural block is executed.

• A statement using blocking assignment blocks the execution of the statements following it, until it gets completed.

• Recommended style for modeling combinational logic.

- Non-Blocking Assignment (using ‘<=1’)

- The assignment to the target gets scheduled for the end of the simulation cycle.

  - Normally occurs at the end of the sequential block.

  - Statements subsequent to the instruction under consideration are not blocked by the assignment.

- Recommended style for modeling sequential logic.

  - Can be used to assign several 'reg' type variables synchronously, under the control of a common clock.

- Non-Blocking Assignment (using ‘<=1’)

. The assignment to the target gets scheduled for the end of the simulation cycle.

- Normally occurs at the end of the sequential block.
- Statements subsequent to the instruction under consideration are not blocked by the assignment.

• Recommended style for modeling sequential logic.

- Can be used to assign several `reg' type variables synchronously, under the control of a common clock.

Some Rules to be followed:

• Verilog synthesizer ignores the delays specified in a procedural assignment statement.

- May lead to functional mismatch between the design model and the synthesized netlist.

• A variable cannot appear as the target of both a blocking and a non-blocking assignment.

- Following is not permissible:

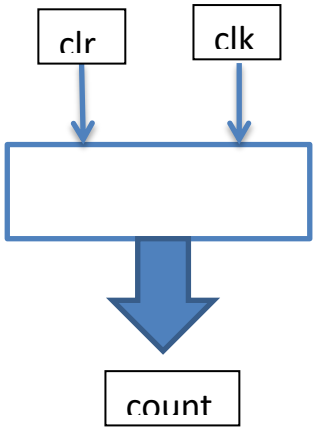
```
value = value+ '1;

value <= init;
```

Example 1:

```
// Up-don counter (synchronous clear)
module counter(mode, clr, ld, d_in, clk, count);
        input mode, clr, ld, clk;  input [0:7] d_in;
        output [0:7] count;       reg[0:7]  count;

        always @(posedge clk)
                if (ld)
                        count <= d_in;
                else if (clr)
                                count <= 0;
                        else if (mode)
                                        count <= count + 1;
                                else
                                        count <= count – 1;
endmodule
```

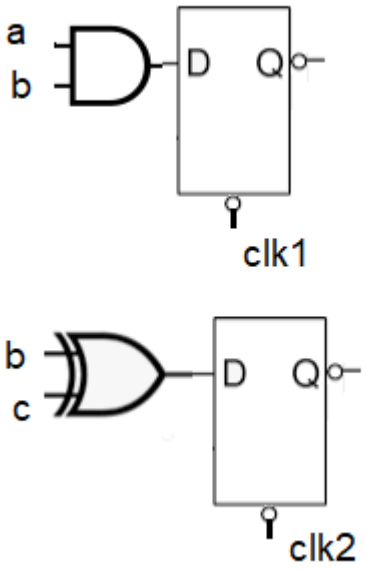
Example 2:

```
// Parameteterized design :: an N-bit counter
module counter( clr, clk, count);
        parameter N=7;
        input clr, clk;
        output [0:N] count;      reg[0:N]  count;

        always @(posedge clk)
                if (clr)
                        count <= 0;
                else
                        count <= count + 1;

endmodule
```



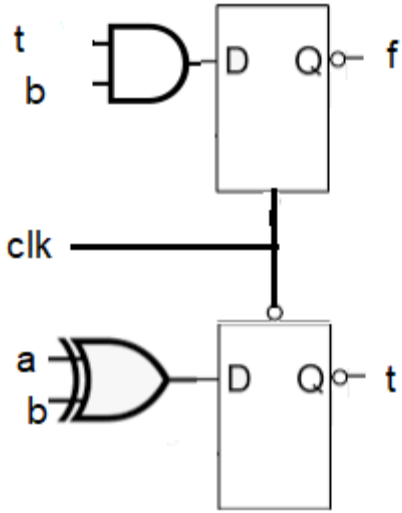
Example 3:

```
// Using more than one clocks in module
module multiple_clk( clk1, clk2, a, b, c, f1, f2);
        input clk1, clk2, a, b, c;
        output f1, f2;      reg f1, f2;

        always @(posedge clk1)
                f1 <= a&b;
        always @(posedge clk2)
                f2 <= b^c;
endmodule
```



Example 4:

```
// Using multiple edges of same clock
module multi_phase_clk( clk, a, b, f);
        input clk, a, b;
        output f;       reg f, t;

        always @(posedge clk)
                f <= t&b;
        always @(negedge clk)
                t <= a^b;
endmodule
```



Example 5: Ring counter 1

```
// A ring counter
module ring_counter( clk, init, count);
        input clk, init;
        output [7:0] count;
        reg [7:0] count;

        always @(posedge clk)
        begin
                if (init)
                        count = 8'b10000000;
                else begin
                                count = count <<1;
                                count[0] = count[7];
                        end
        end
endmodule
```

1 0 0 0 0 0 0 0

Example 6: Ring counter 2 Modified-1

```
// A ring counter
module ring_counter_M1( clk, init, count);
        input clk, init;
        output [7:0] count;
        reg [7:0] count;

        always @(posedge clk)
        begin
                if (init)
                        count = 8'b10000000;
                else begin
                                count <= count <<1;
                                count[0] <= count[7];
                        end
        end
endmodule
```

1 0 0 0 0 0 0 0

Example 7: Ring counter 3 Modified-2

```
// A ring counter
module ring_counter_M2( clk, init, count);
        input clk, init;
        output [7:0] count;
        reg [7:0] count;

        always @(posedge clk)
        begin
                if (init)
                        count = 8'b10000000;
                else
                        count = {count[6:0], count[7]};
        end
endmodule
```

1 0 0 0 0 0 0 0

About "Loop" Statements

- Verilog supports four types of loops:
  - 'while' loop
  - 'for' loop
  - 'forever' loop
  - 'repeat' loop
- Many Verilog synthesizers' supports only `for' loop for synthesis:
  - Loop bound must evaluate to a constant.
  - Implemented by unrolling the 'for' loop, and replicating the statements.

Modeling Memory

• Synthesis tools are usually not very efficient in synthesizing memory.

— Best modeled as a component.

— Instantiated in a design.

• Implementing memory as a two-dimensional register file is inefficient.

Example 8: Memory Modeling

```
// ROM
module mem_example( clk, en, adbus , dbus, rw);
        parameter N=16;
        input clk, rw, en;
        input  [N-1:0] adbus;
        output [N-1:0] dbus;

        ROM Mem1 (clk, en, rw, adbus, dbus);

endmodule
```

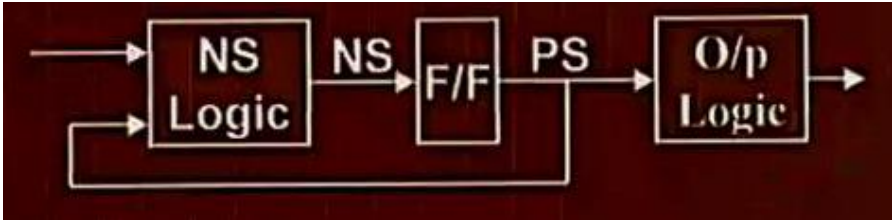
Example 9: Tri_state gates Modeling

```
// A ring counter
module bus_diver( in, out, en);
        input en;    input [0:7] in;
        output [0:7] out;
        reg [0:7] out;

        always @(en or in)

                if (en)
                        out = in;
                else
                        out = 8'bz;

endmodule
```

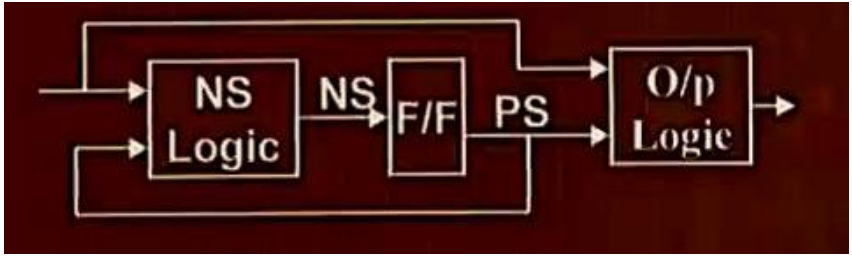
Modeling Finite State Machines

- Two types of FSMs

- Moore Machine



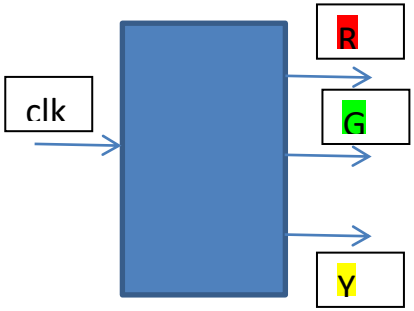
- Mealy Machine



Moore Machine: Example 1

- Traffic Light Controller

- Simplifying assumptions made
  - Three lights only (RED, GREEN, YELLOW)
- The lights glow cyclically at a fixed rate
  - -Say, 10 seconds each
  - The circuit will be driven by a clock of appropriate frequency.



```
module traffic_light (clk, light);
        input clk;
        output [0:2] light; reg [0:2] light;
        parameter S0=0, S1=1, S2=2;
        parameter RED=3'b100, GREEN=3'b010, YELLOW=3'b001;
        reg [0:1] state;
        always @ (posecige clk)
                case (state)
                        S0: begin II S0 means RED
                                light <= YELLOW;
                                state <= S1;
                            end
                         S1: begin // S1 means YELLOW
                                light <= GREEN;
                                state <= S2;
                            end
                        S2: begin // S2 means GREEN
                                light <= RED; state <= SO;
                           end
                         default: begin
                                  light <= RED; state <= SO;
                                end
                endcase
endmodule
```

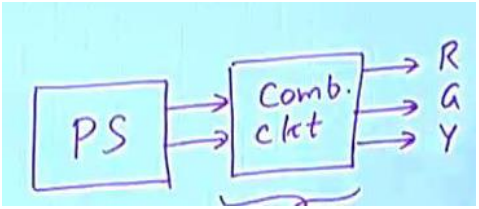
Comment on the solution

— Five flip-flops are synthesized

- Two for 'state'
- Three for `light' (outputs are also latched into flip-flops)

— If we want non-latched outputs, we have to modify the Verilog code.

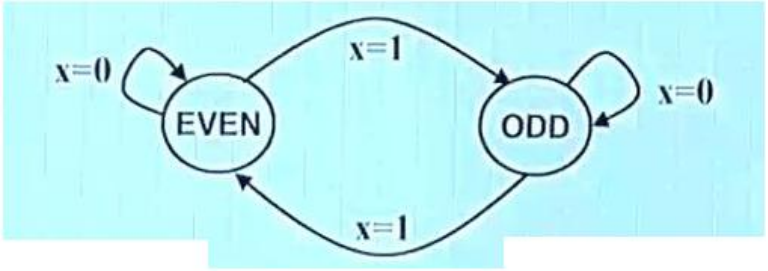
- Assignment to 'light' made in a separate 'always' block.
- Use blocking assignment.

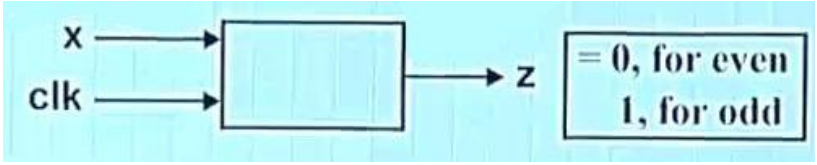


```
module traffic_light_nonlatched_op (clk, lig);
        input clk;
        output [0:2] light;     reg [0:2] light;
       parameter S0=0, S1=1, S2=2;
        parameter RED=3'b100, GREEN=3'b010, YELLOW=3’b001;
        reg [0:1] state;

       always @ (posedge clk)
              case (state)
                     S0: state <= S1;
                      S1: state <= S2;
                      S2: state <= S0;
                     default: state <=S0;
              endcase
       always @ (state)
              case (state)
                     S0: light= RED;
                     S1: light = YELLOW;
                     S2: light= GREEN;
                     default: light = RED;
              endcase
endmodule
```

Moore Machine: Example 2

Serial Parity detector

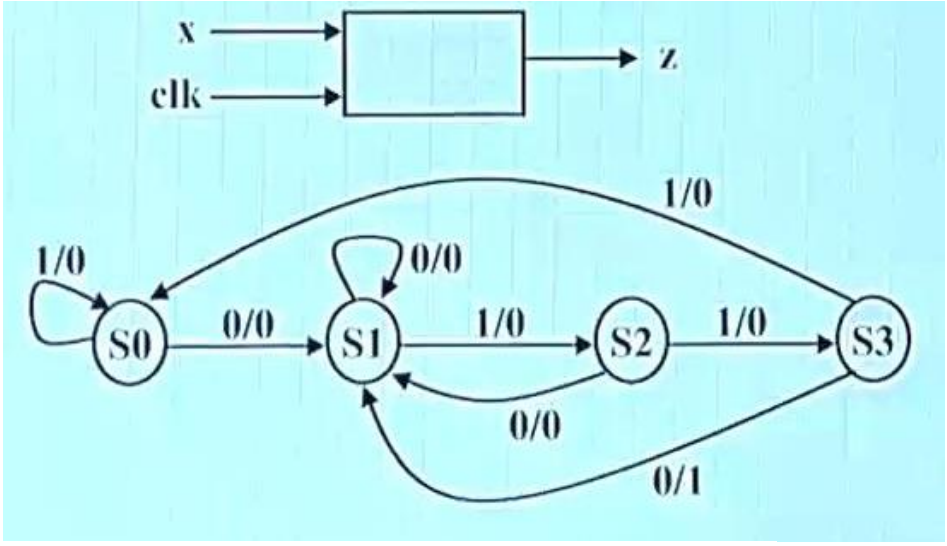




```
module parity_gen (x, clk, z);
        input x, clk;
        output z;
         reg z;
         reg even_odd; // The machine state
        parameter EVEN=0, ODD=1;

        always @ (posedge clk)
                case (even_odd)
                EVEN: begin
                                z <= x ? 1 : 0;
                                even_odd <= x ? ODD : EVEN;
                        end
                ODD: begin
                                z <= x ? 0 : 1;
                                even_odd <= x ? EVEN : ODD;
                        end
                end case
endmodule
```

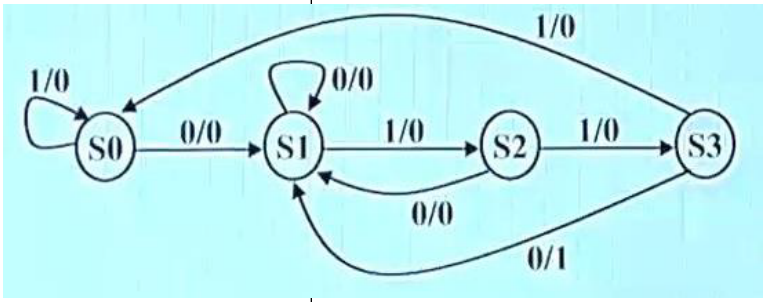
Mealy Machine: Example

Sequence detector for the pattern ‘0110’



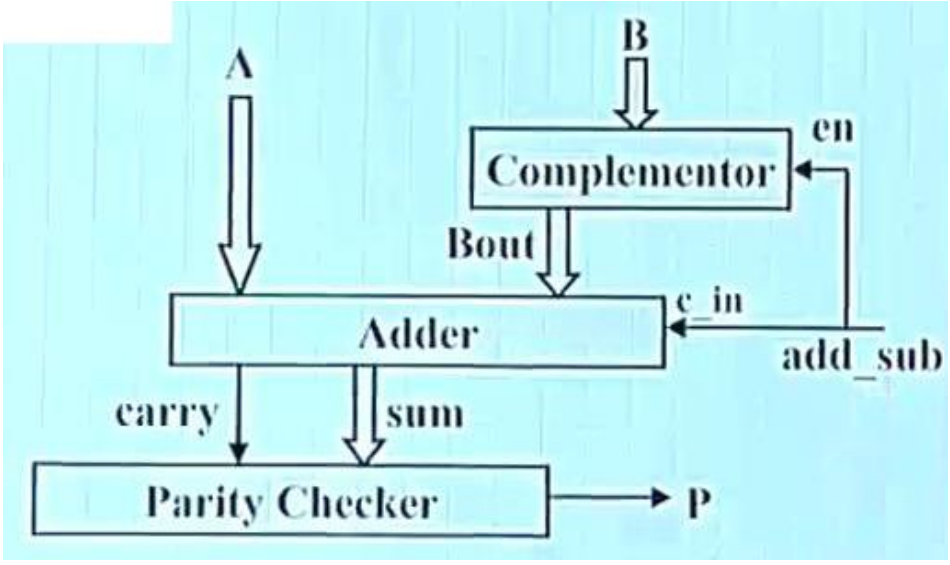
x	0 0 1 1 0 0 1 1 0 1 1 0 0
z	0 0 0 0 1 0 0 0 1 0 0 1 0

```
// Sequence detector for the pattern '0110'
module seq_detector (x, clk, z);
        input x, clk;
         output z;          reg z;
        parameter S0=0, S1=1, S2=2, S3=3;
        reg [0:1] PS, NS;
        always @ (posedge clk)
                PS <= NS;
        always @ (PS or x)
        case (PS)
        S0: begin
                z = x ? 0 : 0;
                NS = x ? S0 :S1;
             end;
        S1: begin
                 z = x ? 0 : 0;
                NS = x ? S2 : S1;
            end;
        S2: begin
                z = x ? 0 : 0;
                 NS = x ? S3 : S1;
             end;
        S2: begin
                z = x ? 0 : 1;
                 NS = x ? S0 : S1;
             end;            endmodule
```



Example with Multiple Modules

- A simple example showing multiple module definitions.



```
module complementor (Y, X, comp);
        input [7:0] X;
        input comp;
        output [7:0] Y;   reg [7:0] Y;

        always @ (X or comp)
                if (comp)
                        Y = ~X;
                else
                        Y = X;
endmodule
```

```
module adder (sum, cy_out, inl, in2, cy_in);
        input [7:0] inl, in2;
        input cy_in;
        output [7:0] sum;        reg [7:0] sum;
        output cy_out;           reg cy_out;

        always @ (inl or in2 or cy_in)
            {cy_out, sum} = inl in2 cy_in;
endmodule
```

```
module parity_checker (out_par, in_word);
        input [8:0] in_word;
        output out_par;
        always @ (in_word)
            out_par = A (in_word);
endmodule
```

```
//Top level module
module add_sub_parity (p, a, b, add_sub);
            input [7:0] a, b;
            input add_sub; // 0 for add, 1 for subtract
           output p; // parity of the result
           wire [7:0] Bout, sum;      wire carry;

           complementor M1 (Bout, B, add_sub);
            adder M2 (sum, carry, A, Bout, add_sub);
           parity_checker M3 (p, {carry, sum});
endmodule
```

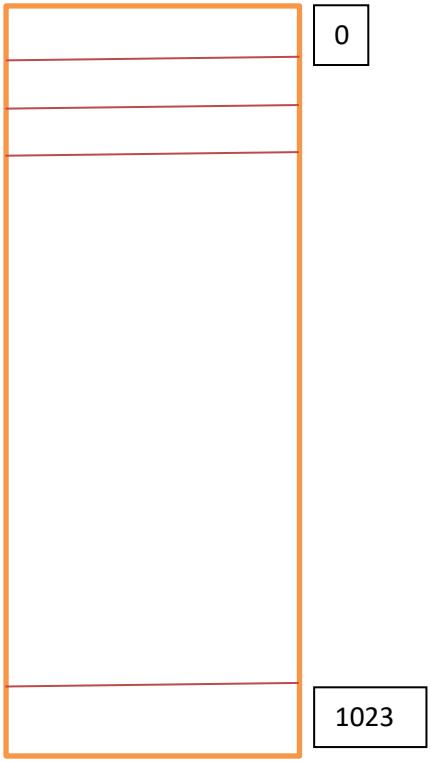
Memory Modeling Revisited

• Memory is typically included by instantiating a pre-designed module.

• Alternatively, we can model memories using two-dimensional arrays

- Array of register variables.
    - Behavioral model of memory
- Mostly used for simulation purposes.
- For small memories, even for synthesis.

```
//Memory Example
module memory_model (……..);

        reg [7:0] mem[0:1023];

endmodule
```



How to Initialize memory

• By reading memory data patterns from a specified disk file.

- Used for simulation.
- Used in test benches.

• Two Verilog functions are available:

1- $readmemb (filename, memname, startaddr, stopaddr)

Data read in binary format.

2. $readmemh (filename, memname, startaddr, stopaddr)

Data read hexadecimal format.

```
//Memory Example
module memory_model (……….);
        reg [7:0] mem[0:1023];

        begin
              $readmemh(“mem.dat”, mem);
        end

endmodule
```

A Specific Example :: Single Port RAM with Synchronous Read-Write

```
module ram_1 (addr, data, clk, rd, wr, cs);
        input [9:0] addr;
        input clk, rd, wr, cs;
        inout [7:0] data;
        reg [7:0] mem [1023:0];
        reg [7:0] d_out;

        assign data = (cs && rd) ? d_out ; 8'bz;
        always @ (posedge clk)
                if (cs && wr && !rd) mem [addr] = data;
        always @ (posedge clk)
                if (cs && rd && !wr) d_out = mem [addr];

endmodule
```

A Specific Example :: Single Port RAM with Asynchronous Read-Write

```
module ram_2(addr, data,  rd, wr, cs);
        input [9:0] addr;
        input rd, wr, cs;
        inout [7:0] data;
        reg [7:0] mem [1023:0];
        reg [7:0] d_out;

        assign data = (cs && rd) ? d_out ; 8'bz;
        always @ (addr or data or rd or wr or cs)
                if (cs && wr && !rd) mem [addr] = data;
        always @ (addr or data or rd or wr or cs)
                if (cs && rd && !wr) d_out = mem [addr];

endmodule
```

A Specific Example :: ROM/EPROM

```
module rom(addr, data,  rd_en, cs);
        input [2:0] addr;
        input rd_en, cs;
        output [7:0] data;
        reg [7:0] data;

        always @ (addr or  rd_en or cs)
                case (addr)
                        0: 22;
                        1:45;



                        7:12;
                endcase
endmodule
```

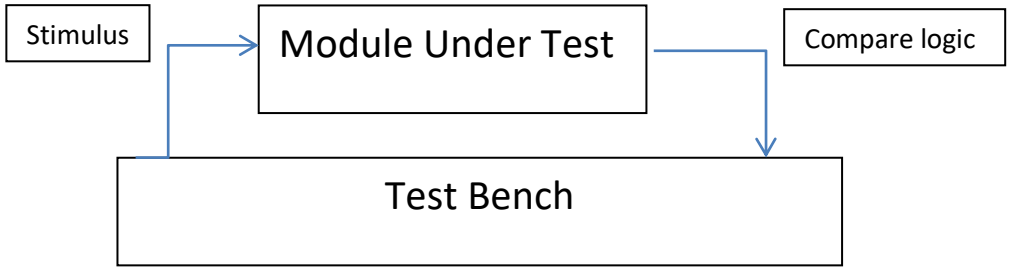
Verilog Test Bench

- What is test bench?

  - A Verilog procedural block which executes only once.

  - Used for simulation.

  — Test bench generates clock, reset, and the required test vectors.



How to Write Testbench?

- Create a dummy template
  - Declare inputs to the module-under-test (MUT) as "reg", and the outputs as "wire"
  - Instantiate the MUT.
- Initialization
  - Assign some known values to the MUT inputs.
- Clock generation logic
  - Various ways to do so.
- May include several simulator directives
  -- Like $display, $monitor, $dumpfile, $dumpvars, $finish.

- $display

  Prints text or variables to stdout.

  - Syntax same as "printf".

- $monitor

  Similar to $display, but prints the value whenever the value of some variable in the given list changes.

- $finish

  Terminates the simulation process.

- $dumpfile

  Specify the file that will be used for storing the waveform.

- $dumpvars

  Starts dumping all the signals to the specified file.

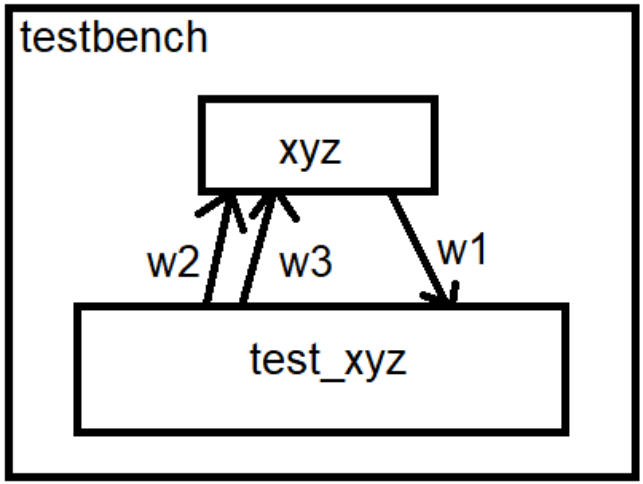
Example Testbench

```
module shifter_toplevel;

        reg clk, clear, shift;

        wire [7:0] data;

        shift_register S1 (clk, clear, shift, data);


        initial
        begin
                clk = 0; clear = 0; shift = 0;
        end
        always
                #10 clk = !clk;
```

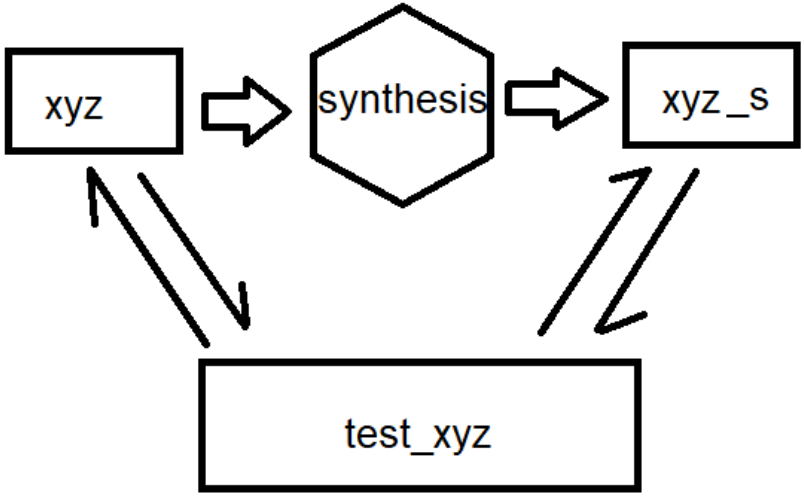
```
initial
begin
        $dumpfile ("shifter.vcd");
        $dumpvars;
end

initial
begin
        $display ("\ttime, \tclk, \tclr, \tsft, \tdata);
        $monitor ("%d, %d, %d, %d, %d, %d", $time, clk, reset, clear, shift, data);
end

initial
                #400 Sfinish;

                ***** REMAINING CODE HERE
endmodule
```

A Complete Example:

```
module testbench;
        wire wl, w2, w3;
         xyz m1(wl, w2, w3);
        test_xyz m2 (wl, w2, w3);
endmodule

module xyz (f, A, B);
        input A, B; output f;
        nor #1 (f, A, B);
endmodule
```



```
module test_xyz (f, A, B);
        input f; output A, B;
        reg A, B;
        initial
        begin
        $mointer($time, “A=%b”, B=%b”, “f=%b”, A, B, f);

        #10 A=0; B=0;
        #10 A=1; B=0;
        #10 A=1; B=1;
        #10 A=0; B=1;
        #10 $finish;
        end
endmodule
```


xyz
synthesis
xyz_s
test_xyz

Ninevah University

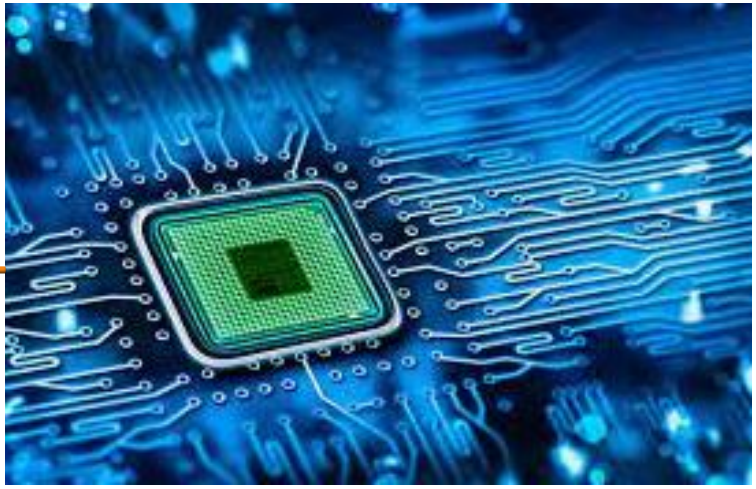
Collage of Electronics Engineering

Course:

Electronic Design Automation

Lecturer: H. M. Hussein

EDA02: Verilog 1



Parameters

. A parameter is a constant with a name.

. No size is allowed to be specified for it.

-The size gets decided from constant itself (32bit)

. Examples:

```
parameter Hi=25, Lo=5;
parameter up=b00,  down=b01, steady=b10
```

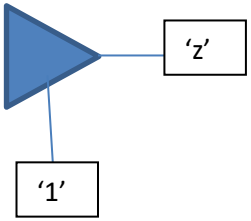
Logic values

. The common values used in modeling hardware are:

- 0 :: Logic-0 or FALSE
- 1 :: Logic-1 or TRUE
- x :: Unknown (or don't care)
- z :: High impedance

. Initialization:

- All unconnected nets set to 'z'
- All register variables set to 'x'

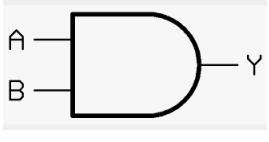


Logic Gates

- Verilog provides a set of predefined logic gates.

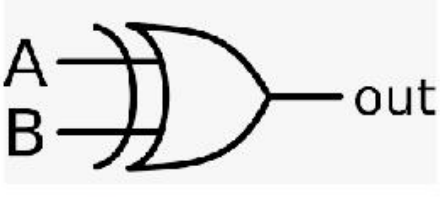
- They respond to inputs (0, 1, x, or z) in a logic way.

- Examples:: AND .



0 & 0 → 0
0 & 1 → 0
1 & 1 → 1
1 & x → x

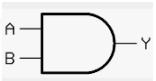
0 & x → 0
1 & z → x
z & x → x

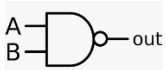


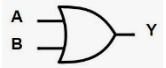
0 & 0 → 0
0 & 1 → 1
1 & 1 → 0
1 & x → x
0 & x → x
1 & z → x
0 & z → x

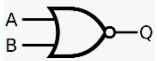
Primitive logic gates (instantiations):

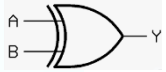
```
and   G (out, in1, in2);
nand  G (out, in1, in2);
or    G (out, in1, in2);
nor   G (out, in1, in2);
xor   G (out, in1, in2);
xnor  G (out, in1, in2);
not   G (out1, in);
buf   G (out1, in);
```

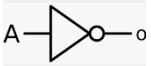


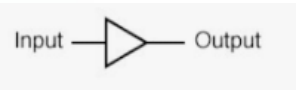






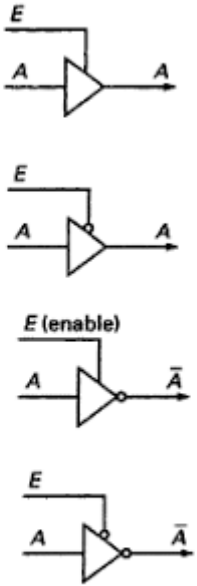






Primitive Tri-state gates (instantiations):

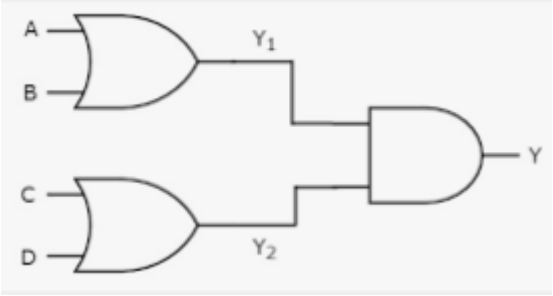
```
bufif1  G (out, in, ctrl);
bufif0  G (out, in, ctrl);
notif1  G (out, in, ctrl);
notif0  G (out, in, ctrl);
```



Some Points to Note:

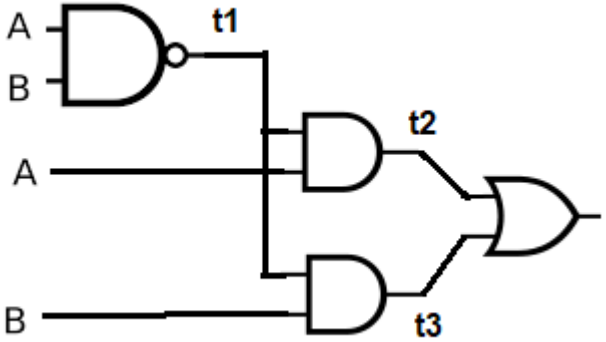
. For all primitive gates:

- The output port must be connected to a net (a wire).
- The input ports may be connected to nets or register type variables.
- They can have a single output but any number of inputs.
- An optional delay may be specified.
  -> Logic synthesis tools ignore time delays.


A
B
C
D
Y1
Y2
Y

Example 1:

```
‘timescale 1ns/1ns
module ex_or(f, a, b);
        input a, b;
        output f;
        wire t1, t2, t3;

        nand #5 m1(t1, a, b);
        and #5 m1(t2, a, t1);
        and #5 m1(t3, t1, b);
        or #5 m1(f, t2, t3);

endmodule
```



Hardware Modeling Issues:

. The values computed can be held in

- A ‘wire’
- A ‘flip-flop’ (edge-triggered storage cell)
- A ‘latch’ (level-sensitive storage cell)

. A variable can be of

-‘net’ data type:

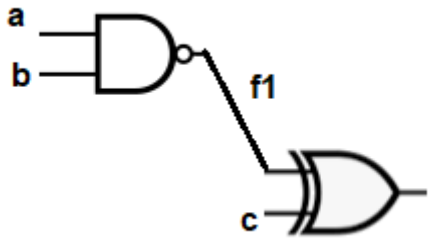
-> Maps to a wire during synthesis.

- ‘register’ data type

-> Maps either to a ‘wire’ or to a ‘storage cell’ depending on the context under which a value is assigned.

Example 2:

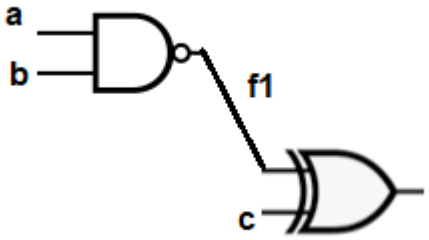
```
module carry(cy, a, b, c);
        input a, b, c;
        output f1, f2;
        wire a, b, c;
        reg f1, f2;

        always @(a or b or c)
        begin
                f1= ~(a & b);
                f2= f1 ^ c;
        end
endmodule
```



The synthesis system will generate a wire for f1

Example 3:

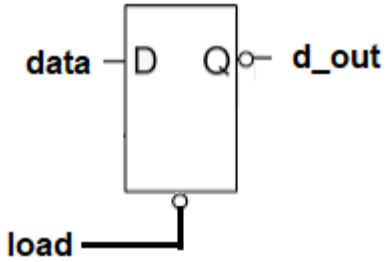
```
module carry(cy, a, b, c);
        input a, b, c;
        output f1, f2;
        wire a, b, c;
        reg f1, f2;

        always @(a or b or c)
        begin
                f2= f1 ^ c;
                f1= ~(a & b);
        end
endmodule
```



The synthesis system will not generate a storage cell for f1

Example 4:

```
// A latch gets inferred here
module simple_latch(data, load, d_out);
        input data, load;
        output d_out;
        reg t;

        always @(load or data)
        begin
                if(!load)
                        t=data;
                d_out = !t;
        end
endmodule
```



Else part missing; so latch is inferred.

Verilog Operators:

. Arithmetic operators:

*, /, +, -, %.

. Logical operators:

! :: logical negation

&& :: logical AND

|| :: logical OR

. Relational operators:

>, <, >=, <=, ==, !=

. Bitwise operators:

~, &, |, ^, ~^

Verilog Operators:

. Reduction operators (operate on all the bits within a word). Example: b = &a; //a 8bit

. Shift operators:

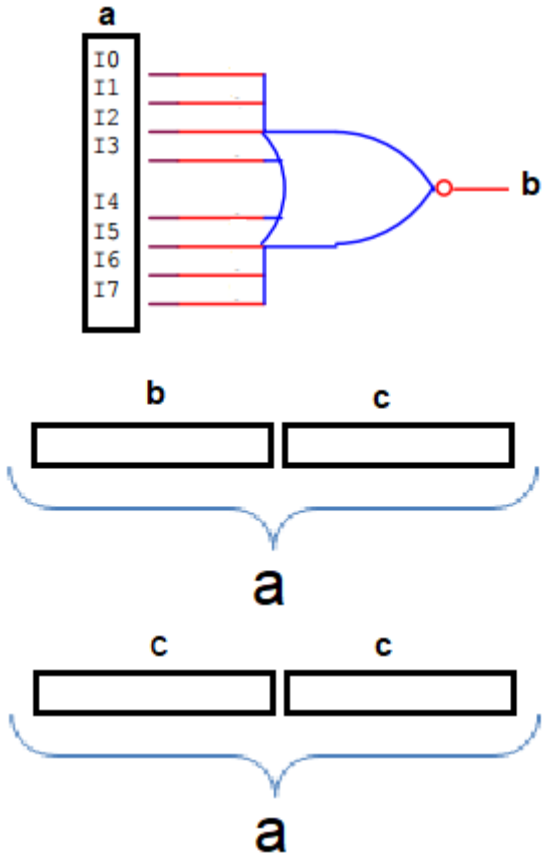
>>, <<

. Concatenation {} a={b,c}

. Replication {{}} a= {2{c}}

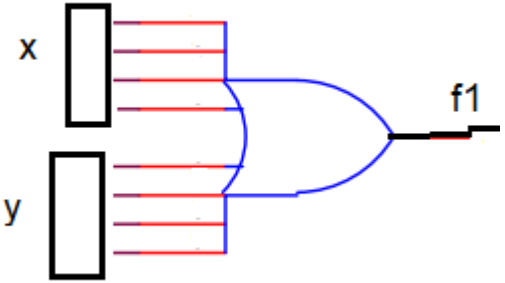
. Conditional

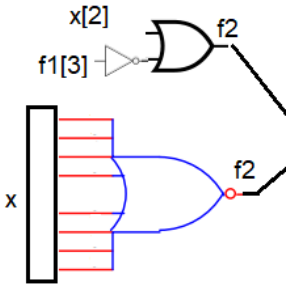
<condition> ? <expr1> : <expr2>

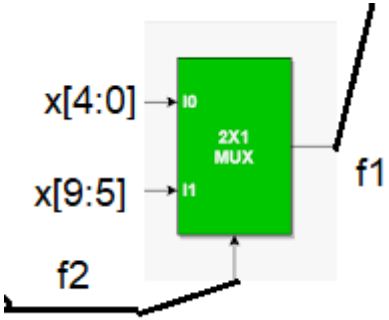


Example 5:

```
module oper_example(x, y, f1, f2);
        input x, y;
        output f1, f2;
        wire [9:0] x, y;
        wire [4:0] f1;
        wire f2;

        assign f1= x[4:0] & y[4:0];
        assign f2= x[2] | ~f1[3];
        assign f2= ~& x;
        assign f1=f2? x[9:5] : x[4:0];
endmodule
```

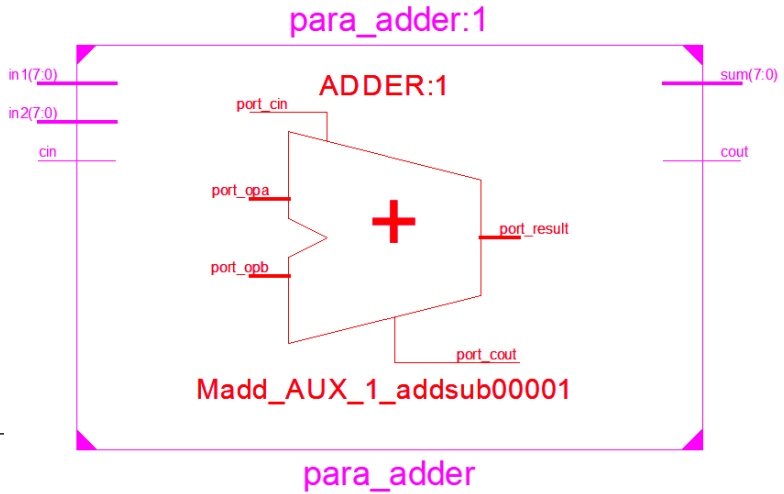






Example 6:

```
// An 8 bit adder description
module para_adder(sum, cout, in1, in2, cin);
        input [7:0] in1, in2; input cin;
        output [7:0] sum; output cout;

        assign #20 {cout, sum} =in1+in2+cin;


endmodule
```



Some points:

. The presence of a ‘z’ or ‘x’ in a reg or wire being used in arithmetic expression results in the whole expression being unknown ‘x’ **.**

. The logical operators (!, &&, ||) all evaluate to a 1-bit result (0, 1, or x).

. The relational operators (>, <, ….) also evaluate to a 1-bit result (0 or 1).

. Boolean **false** is equivalent to 1’b0

Boolean **true** is equivalent to 1’b1.

Some valid statements:

```
assign outp = (p == 4'b1111)
if (load && (select == 2'b01)) …..

assign a = b >> 1;
assign a= b << 3;

assign f= {a,b};
assign f= {a, 3'b101,b};
assign f= {x[2], y[0], a};

assign f={4{a}}
assign f={2'b10,  3{2'b01},  x};
```

Description Styles in Verilog:

. Two different styles of description:

1. Data flow
   - Continuous assignment (combinational)
2. Behavioral
   - Blocking (combinational)
   - Non-blocking (sequential)

Data flow Style: Continuous Assignment:

. Identified by the keyword ‘’assign’’:

```
assign a = b & c;
assign f[2]=c[0];
```

. Forms a static binding between:

- The net being assigned on LHS.
- The expression on the RHS.

. The assignment continuously active:

. Almost exclusively used to model combinational logic.

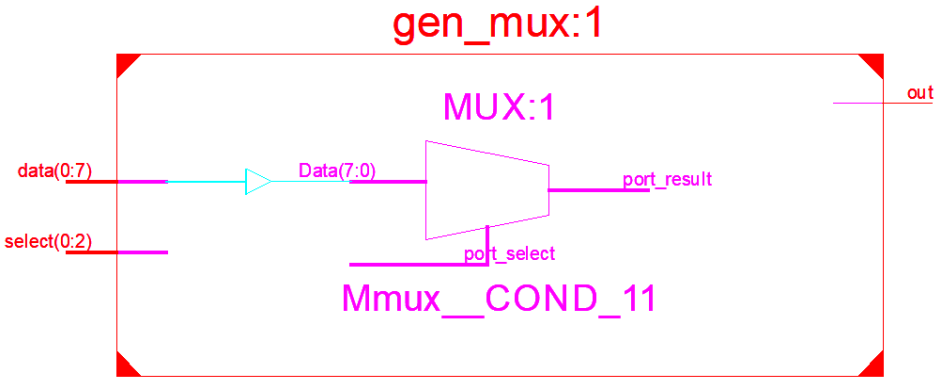
. For an assign statement:

- RHS: contain register or net.
- LHS: must be net type, like wire.

Example 7:

```
module gen_mux (data, select, out);
        input [0:7] data;
        input [0:2] select;
        output out;

        assign out= data[select];

endmodule
```

Non-constant index in expression on RHS generates a MUX



Example 8:

```
module gen_demux (in, select, out);
        input in;
        input [0:1] select;
        output [0:3] out;

        assign out[select] = in;

endmodule
```

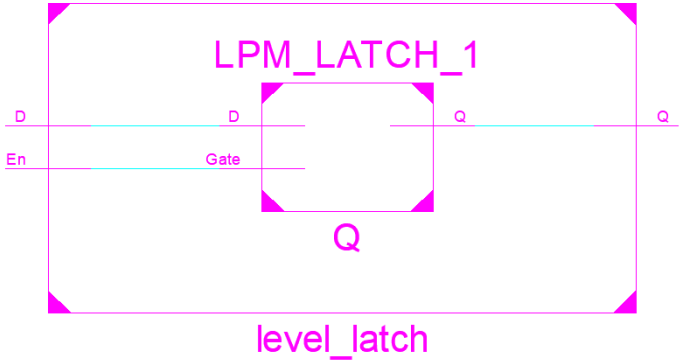
Non-constant index in expression on LHS generates a decoder

Errors

ERROR:HDLCompilers:226 - "ex1.v" line 27 Index in bit-select of vector wire 'out' must be constant
ERROR:HDLCompilers:53 - "ex1.v" line 27 Illegal left hand side of continuous assign

Example 9:

```
// level sensitive latch
//Using assign to describe
//sequential logic
module level_latch (D, Q, En);
        input D, En;
        output Q;

        assign Q = En ? D : Q;

endmodule
```



Behavioral Style: Procedural Assignment:

. The procedural block defines:

- A region containing sequential statements.
- The statements execute in they order they are written.

. Two types of procedural blocks:

- "**always**" block: A continues loop that never terminates.
- "**initial**" block: Executed once in the beginning of simulation (used in **test-benches**).

. A module can contain any number of “always” blocks, all of which execute concurrently.

. Basic syntax of “always” block:

```
module _____;
        always @(event_expression)
        begin

                sequential statements;

        end
endmodule
```

. The @(event_expression) is required for both combinational and sequential logic description.

. Only “reg” type variables can be assigned within an “always” block.

- The sequential “always” block executes only when the event expression triggers.
- At other times the block is doing nothing.
- An object being assigned to must therefore remember the last value assigned (not continuously driven)
- So, only “reg” type type variables can be assigned within an “always” block.
- Of course, any kind of variables may appear in the event expression (reg, wire, etc.)

Sequential Statements:

1-

```
begin
        sequential_statements;
end
// end not required if there is only
//one statemnet
```

2-

```
if (expression)
        sequential_statement;
else
        sequential_statement;
```

3-

```
case (expression)
        expr:  sequential_statement;
        ……..
        Default: sequential_statement;
endcase
```

4-
```
forever
        sequential_statement;
```

5-
```
repeat (expression)
        sequential_statement;
```

6-
```
while (expression)
        sequential_statement;
```

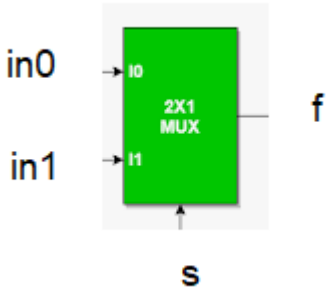
7-
```
for (expression1; expression2, expression3 )
        sequential_statement;
```

8-
```
# (time_value)    //makes a block suspend for time_value time units;
```

9-
```
@ (event_expression) //makes a block suspend until
                     //  event_expression triggers.
```

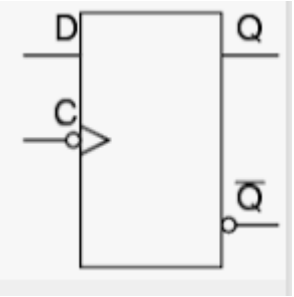
Example 10:

```
// A combinational logic example
module mux2-1 (in1, in0, s, f);
        input in1, in0, s;
        output f;
        reg f;

        always @(in1 or in0 or s)
                if (s)
                        f=in1;
                else
                        f= in0;
endmodule
```



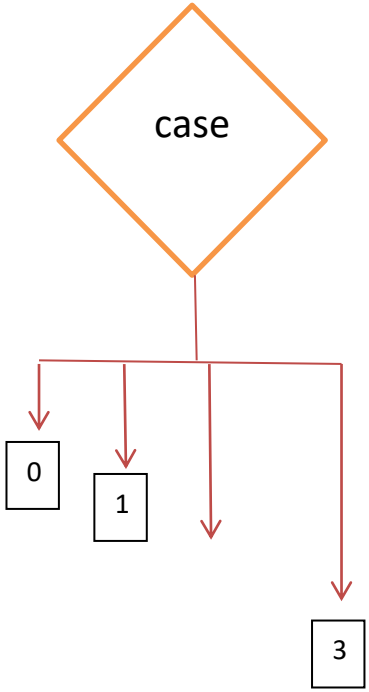
Example 11:

```
// A sequential logic example
module dff_negedge (D, clock, Q, Qbar);
        input D, clock;
        output Q, Qbar;
        reg Q, Qbar;

        always @(negedge clock)
                begin
                        Q= D;
                        Qbar= ~D;
                end
endmodule
```



Example 12:

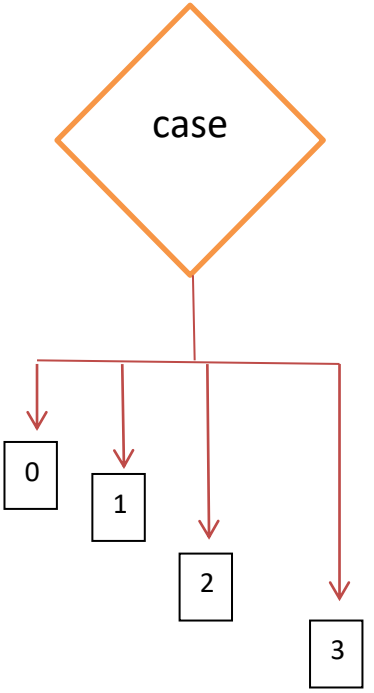
```
// Another sequential logic example
module incomplete_state_spec (curr_state, flag);
        input [0:1] curr_state;
        output [0:1] flag;
        reg [0:1] flag;

        always @( curr_state)
                case (curr_state)
                        0, 1 : flag=2;
                        3    : flag=0;
                endcase
endmodule
```



→ Latch is *inferred*

Example 12 repated:

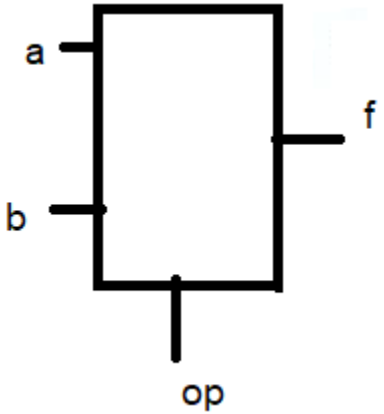
```
// Another sequential logic example
module incomplete_state_spec (curr_state, flag);
        input [0:1] curr_state;
        output [0:1] flag;
        reg [0:1] flag;

        always @( curr_state)
                begin
                        flag = 0
                        case (curr_state)
                                0, 1 : flag=2;
                                3    : flag=0;
                        endcase
                end
endmodule
```



➔ Latch is *avoided*

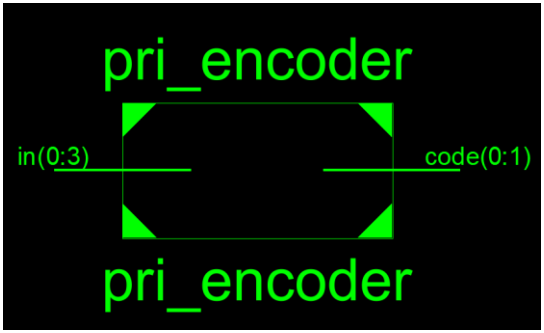
Example 13:

```
// ALU example
module ALU_4bit (f, a, b, op);
        input [1:0] op;     input [3:0] a,b;
        output [3:0] f;     reg [3:0] f;

        parameter  ADD=2'b00, SUB=2'b01,
                        MUL=2'b10, DIV=2'b11;
        always @( a or b or op)
                case (op)
                        ADD: f= a+b;
                        SUB: f=a-b;
                        MUL: f =a*b;
                        DIV: f= a/b;
                endcase
endmodule
```



Example 14:

```
// priority encoder example
module pri_encoder (in, code);
        input [0:3] in;
        output [0:1] code;
        reg [0:1] code;

        always @( in)
                case (1'b1)
                        in[0] : code = 2'b00;
                        in[1] : code = 2'b01;
                        in[2] : code = 2'b10;
                        in[3] : code = 2'b11;
                endcase
endmodule
```

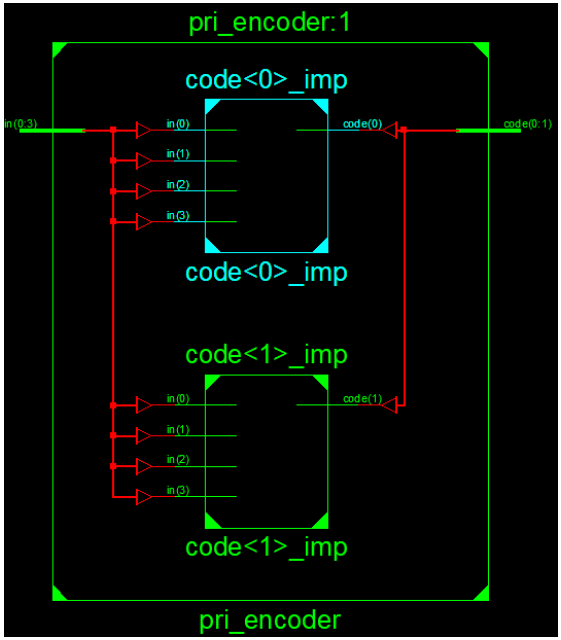


```
WARNING:Xst:737 - Found 2-bit latch for signal <code>.
```

```
Latches may be generated from incomplete case or if statements.
```

```
We do not recommend the use of latches in FPGA/CPLD designs, as they may lead to timing problems.
```

```
22  module pri_encoder (in, code);
23     input [0:3] in;
24  output [0:1] code;
25  reg [0:1] code;
26
27     always @( in)
28        case (1'b1)
29           in[0] : code = 2'b00;
30           in[1] : code = 2'b01;
31           in[2] : code = 2'b10;
32           in[3] : code = 2'b11;
33           default : code =0;
34        endcase
35  endmodule
```



Ninevah University

Collage of Electronics Engineering

CMOS Fabrication and Layout

CMOS Fabrication and Layout

- Transistors are fabricated on a thin silicon wafer that serve as both a mechanical support and electrical common point called substrate
- Fabrication process (a.k.a. Lithography) is similar to printing press
  - On each step, different materials are deposited or etched
- Easiest way to understand physical layout is to look at the wafer from two perspectives:
  - Top-section
  - Cross-section

Photo Lythography

- "Carving pictures in stone using light"

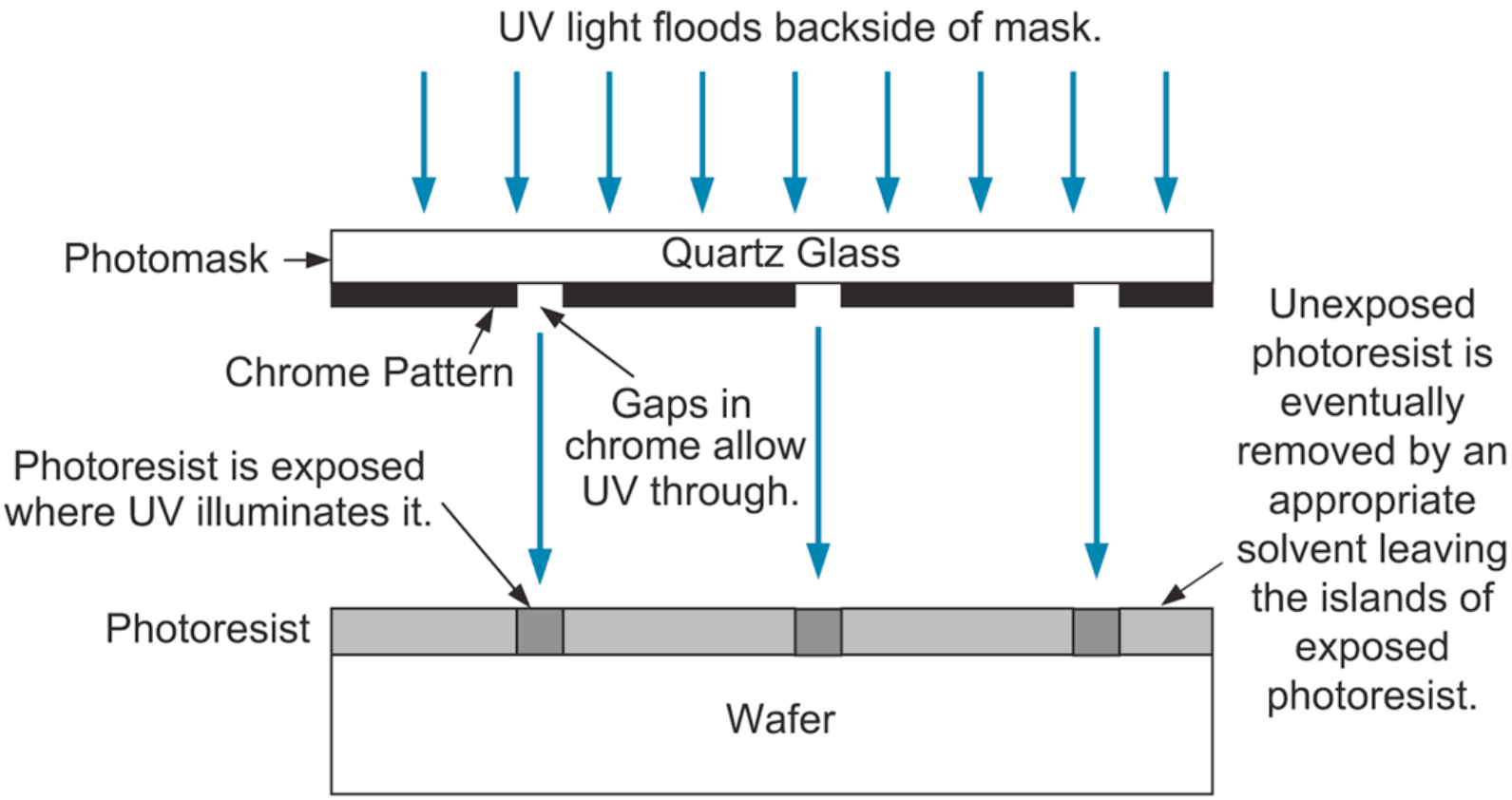


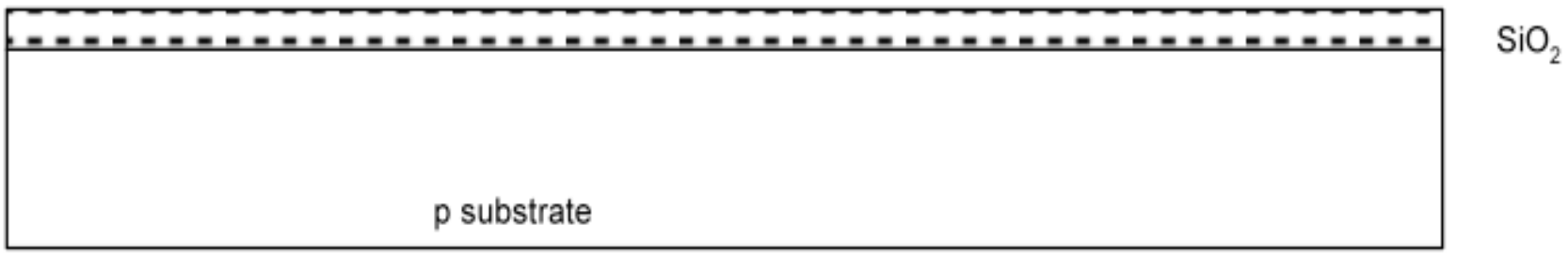
FIG 3.1 Photomasking with a negative resist (lens system between mask and wafer omitted to improve clarity and avoid diffracting the reader ☺)

Fabrication Steps

- Start with blank wafer
- Build invert from bottom up

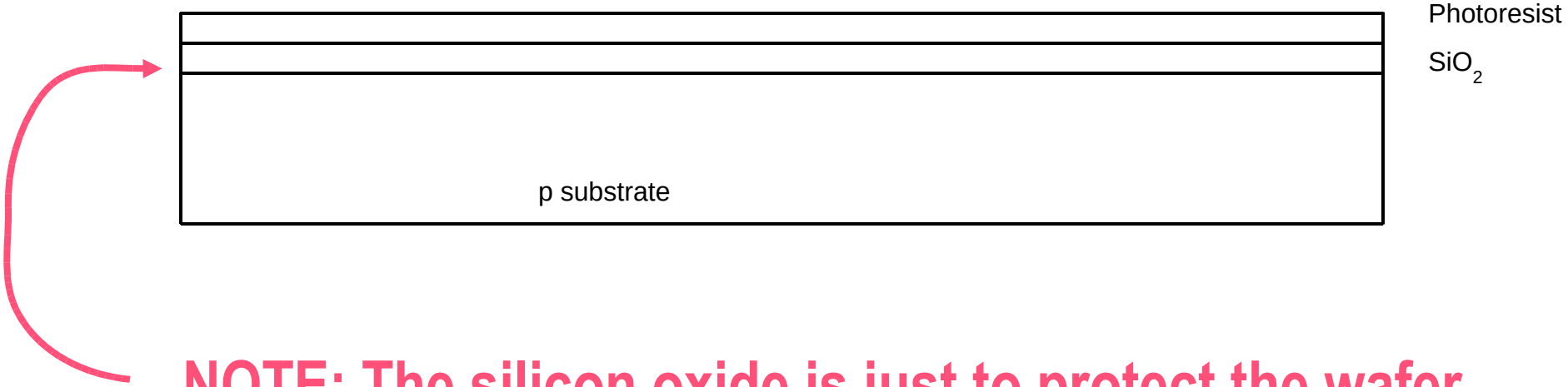
n-well Formation

- First step will be to form the n-well
  - Cover wafer with protective layer of SiO_2 (oxide)
    to grow SiO_2 on top of Si wafer put the Si with H_2O or O_2 in oxidation furnace at 900 – 1200 C
  - (Remove layer where n-well should be built)
  - (Implant or diffuse n dopants into exposed wafer)
  - (Strip off SiO_2)



Deposit silicon-oxide and photoresist

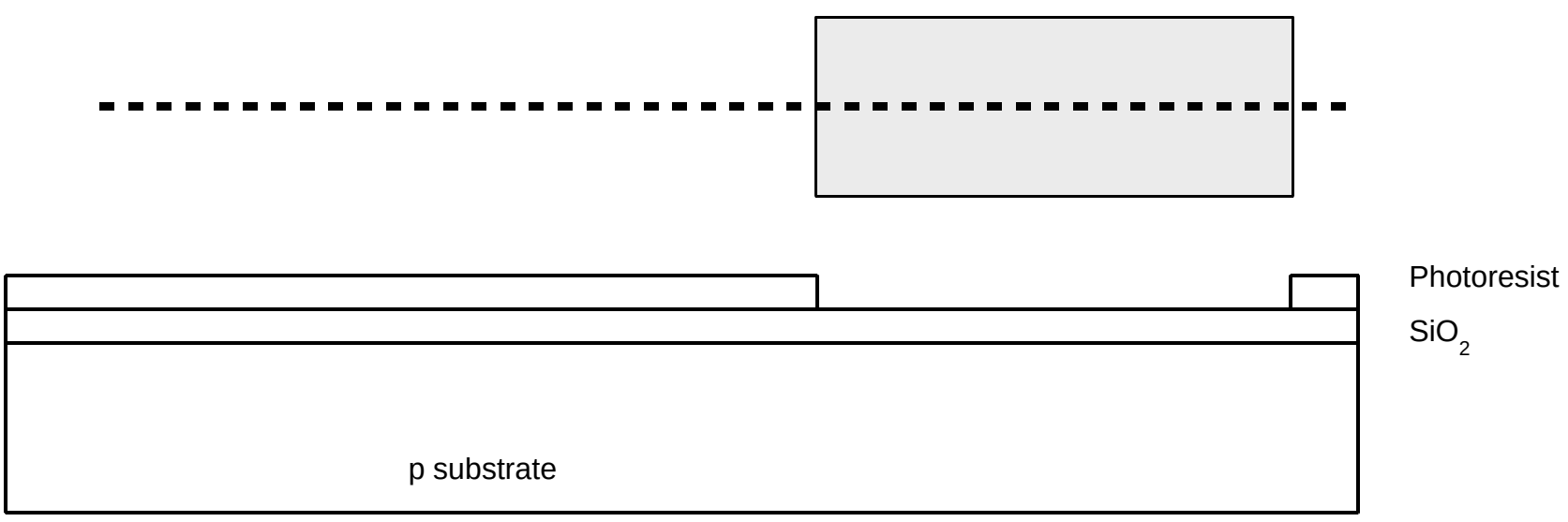
- Photoresist is a light-sensitive organic polymer
- Softens where exposed to light



NOTE: The silicon oxide is just to protect the wafer

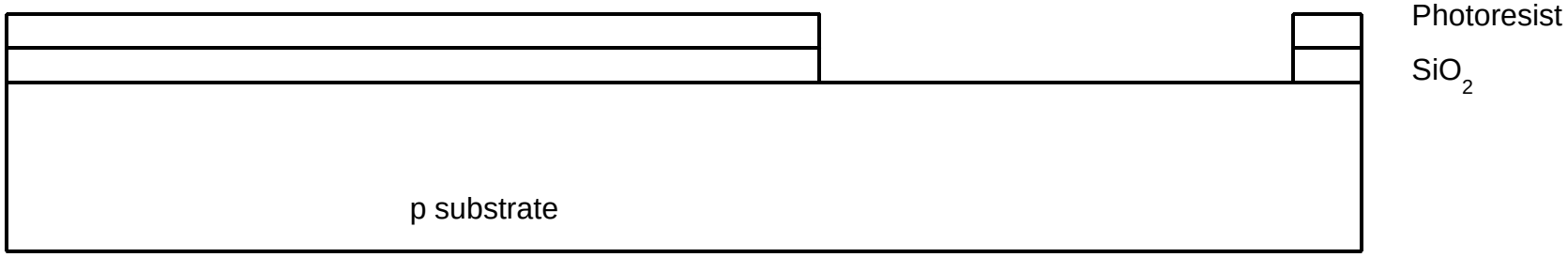
Photo-Lithography

- Expose photoresist through n-well mask
- Strip off exposed photoresist



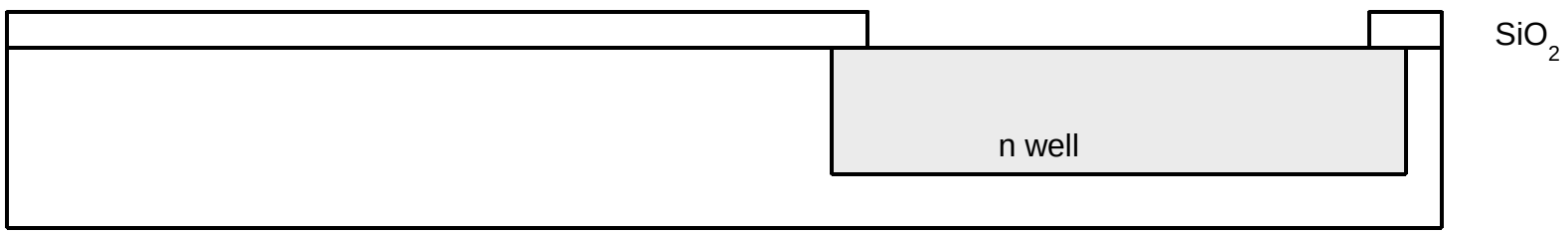
Etching

- Etch oxide with hydrofluoric acid (HF)
  - Seeps through skin and eats bone: nasty stuff!!!
- Only attacks oxide where resist has been exposed



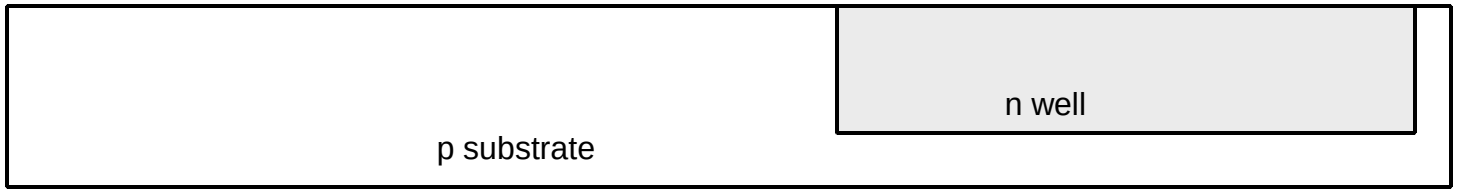
The n-well

- n-well is formed with diffusion or ion implantation
- Diffusion
  - Place wafer in furnace with arsenic gas
  - Heat until As atoms diffuse into exposed Si
- Ion Implantation
  - Blast wafer with beam of As ions
  - Ions blocked by SiO_2, only enter exposed Si



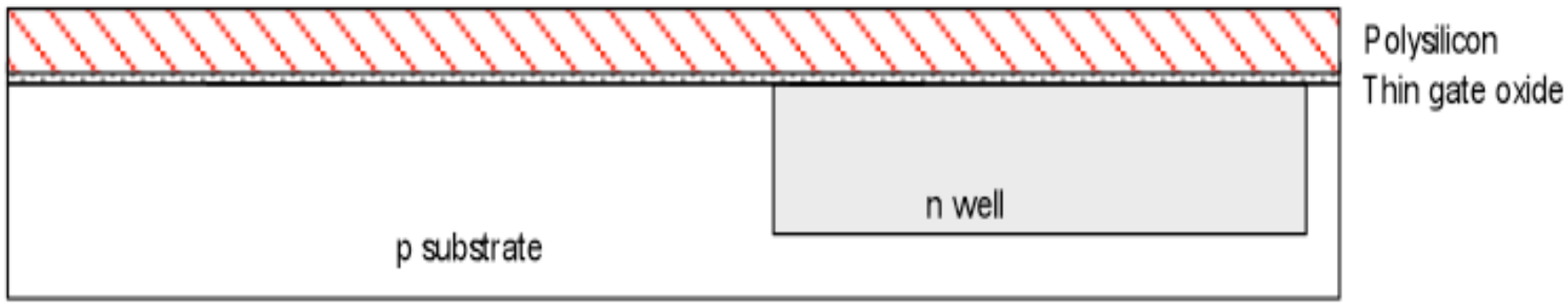
Strip protective oxide

- Strip off the remaining oxide using HF
- Back to bare wafer with n-well
- Subsequent steps involve similar series of steps



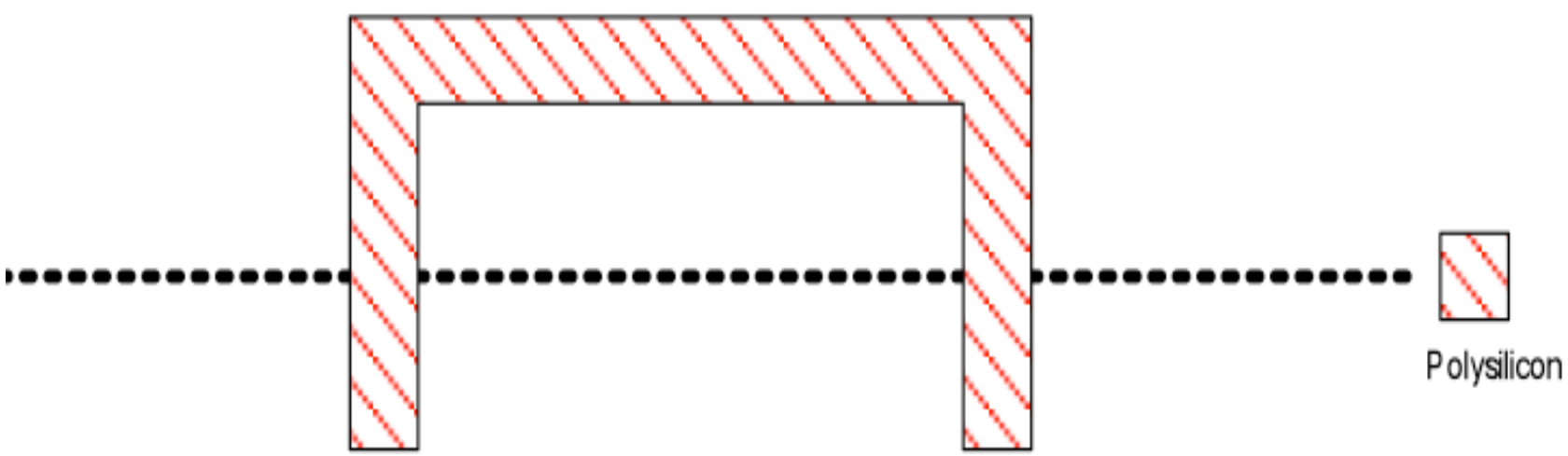
Gate oxide and Polysilicon

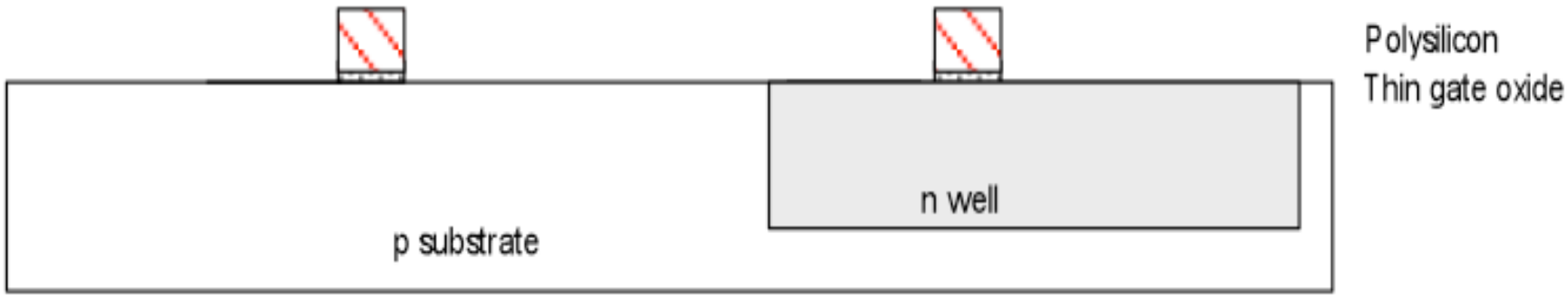
- Deposit very thin layer of gate oxide
  < 20 Å (6-7 atomic layers)
- Chemical Vapor Deposition (CVD) of silicon layer
  - Place wafer in furnace with Silane gas (SiH_4)
  - Forms many small crystals called polysilicon
  - Heavily doped to be good conductor



Polysilicon patterning

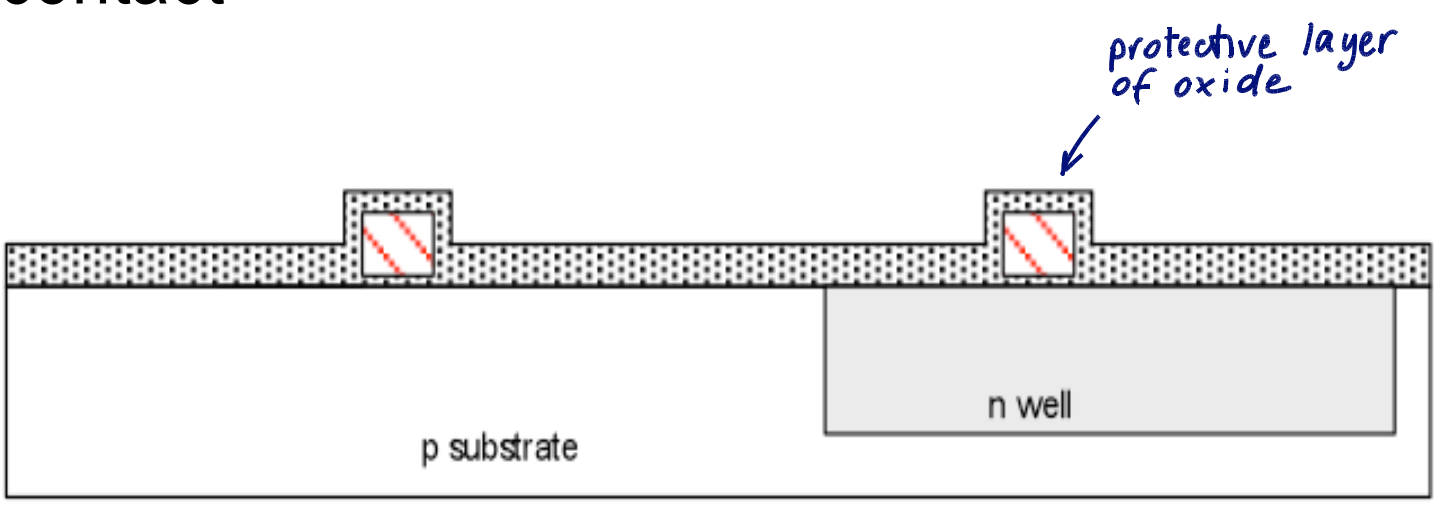
- Use same lithography process to pattern polysilicon





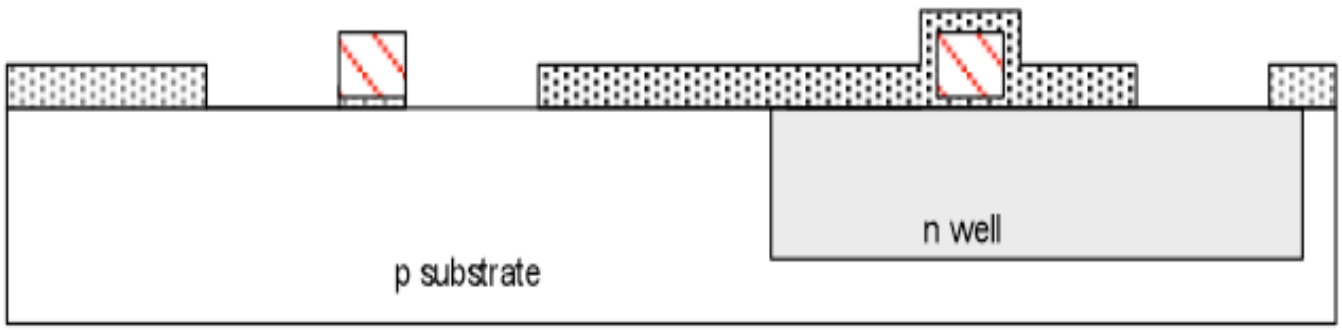
Self-aligned polysilicon gate process

- The polysilicon gate serves as a mask to allow precise alignment of the source and drain with the gate
- Use oxide and masking to expose where n+ dopants should be diffused or implanted
- n-diffusion forms nMOS source, drain, and n-well contact



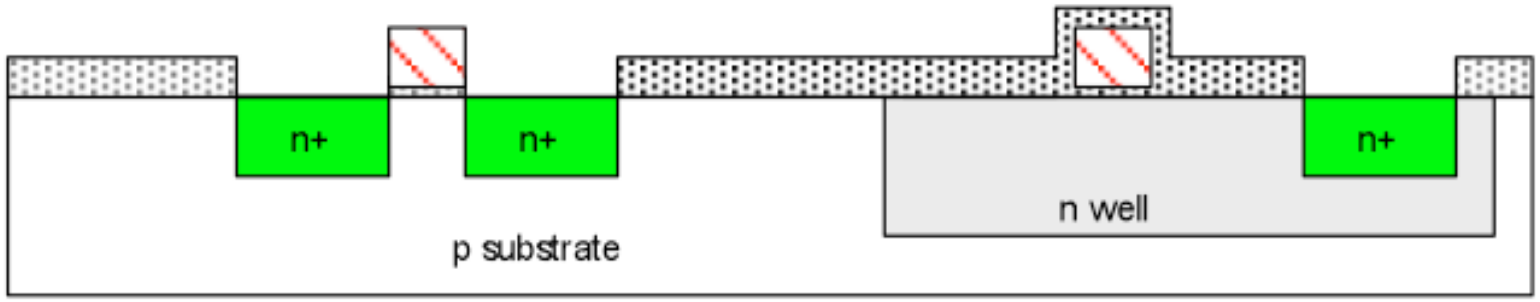
Formation of the n-diffusions

- Pattern oxide and form n+ regions
- *Self-aligned process* (poysilicon gate) “blocks” diffusion under the gate
- Polysilicon is better than metal for self-aligned gates because it doesn’t melt during later processing

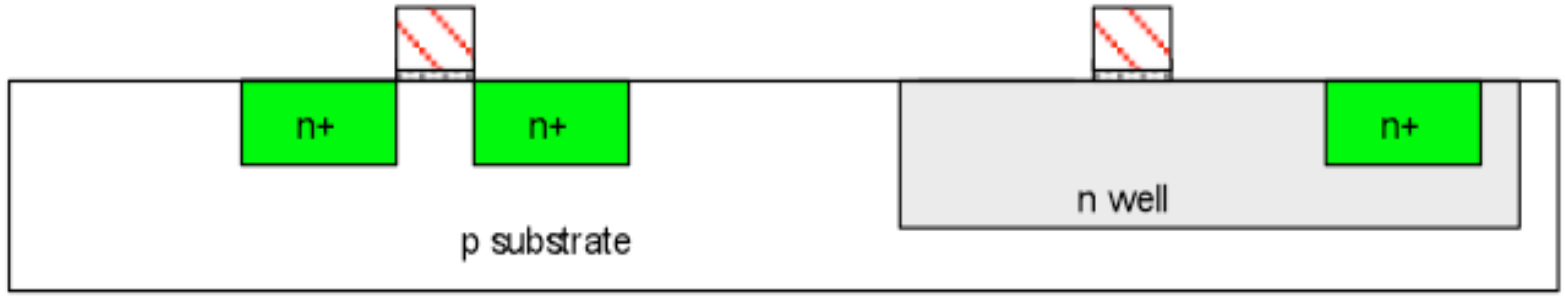


The n-diffusions

- Historically dopants were diffused
- Usually ion implantation today (but regions are still called diffusion)

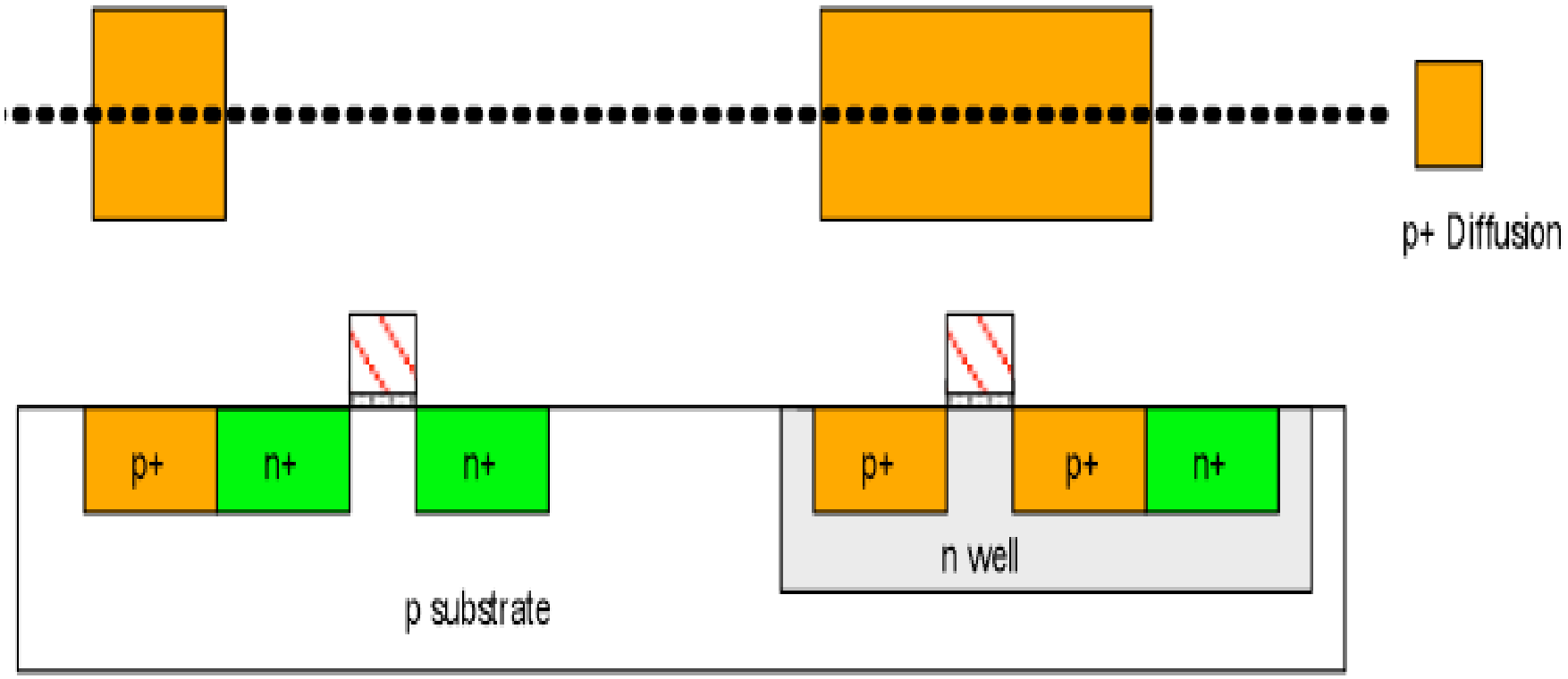


- Strip off oxide to complete patterning step



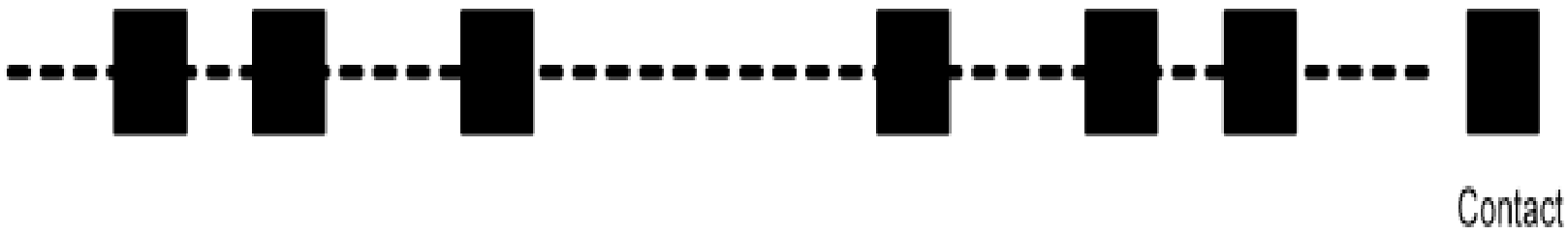
The p-diffusions

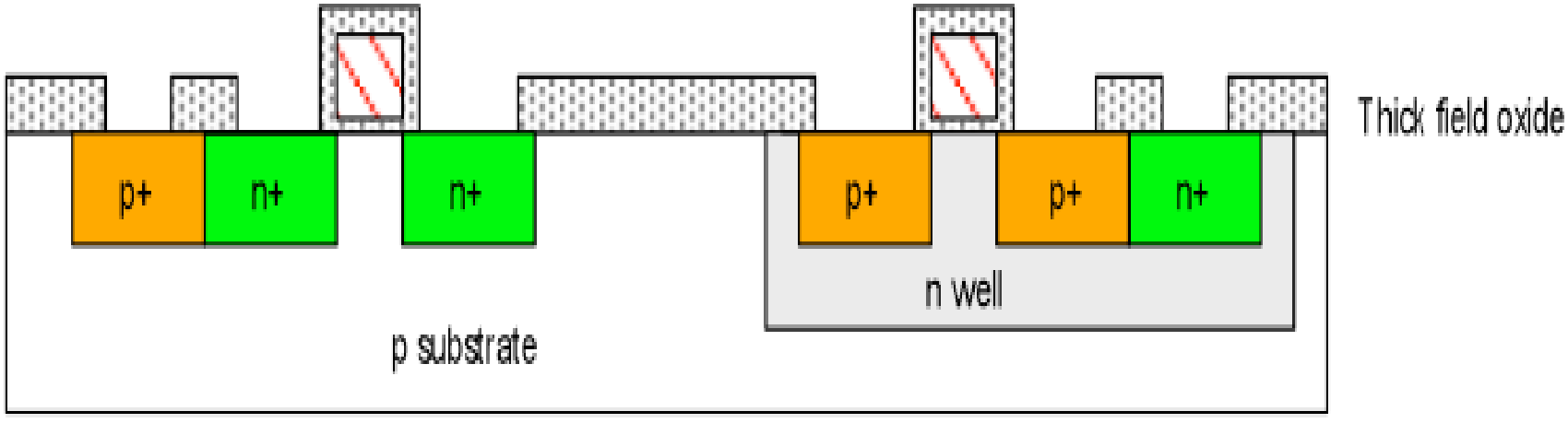
- Similar set of steps form p+ diffusion regions for pMOS source and drain and substrate contact



Contacts

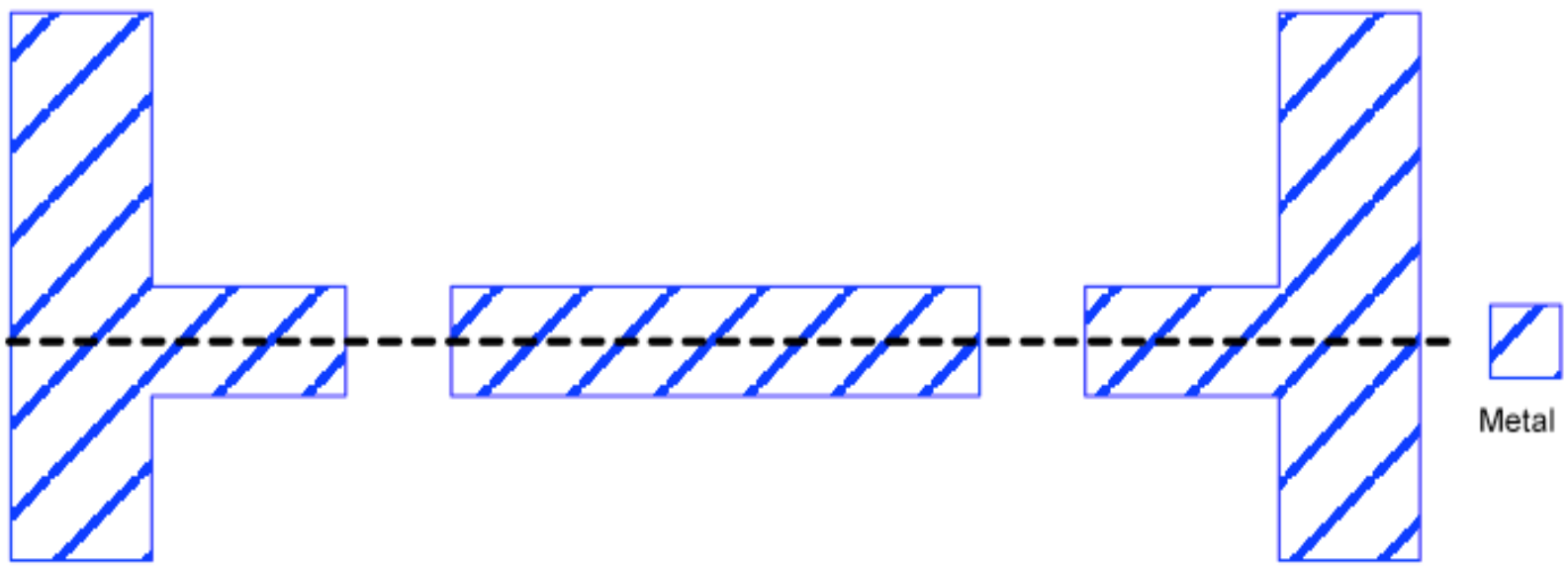
- Now we need to create the devices' terminals
- Cover chip with thick field oxide (FOX)
- Etch oxide where contact cuts are needed

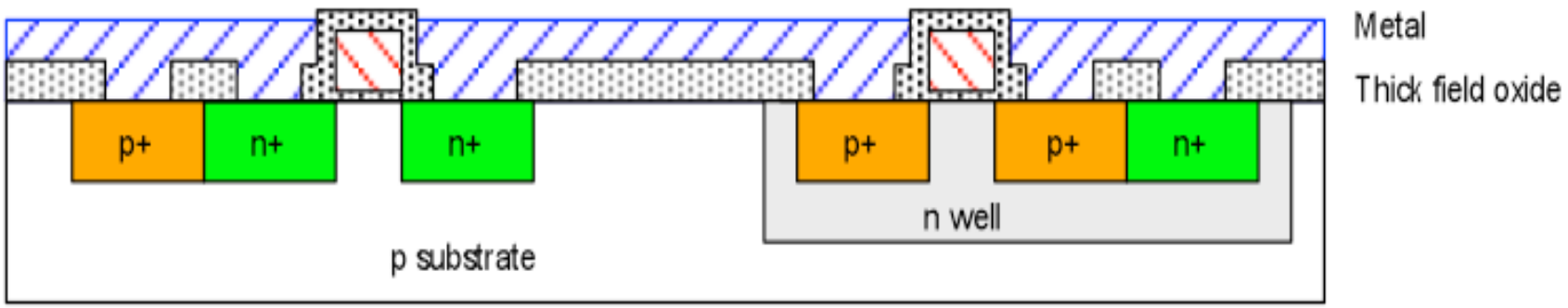




Metallization

- Sputter on aluminum over whole wafer, filling the contacts as well
- Pattern to remove excess metal, leaving wires





Fabrication Steps Summary (1/3)

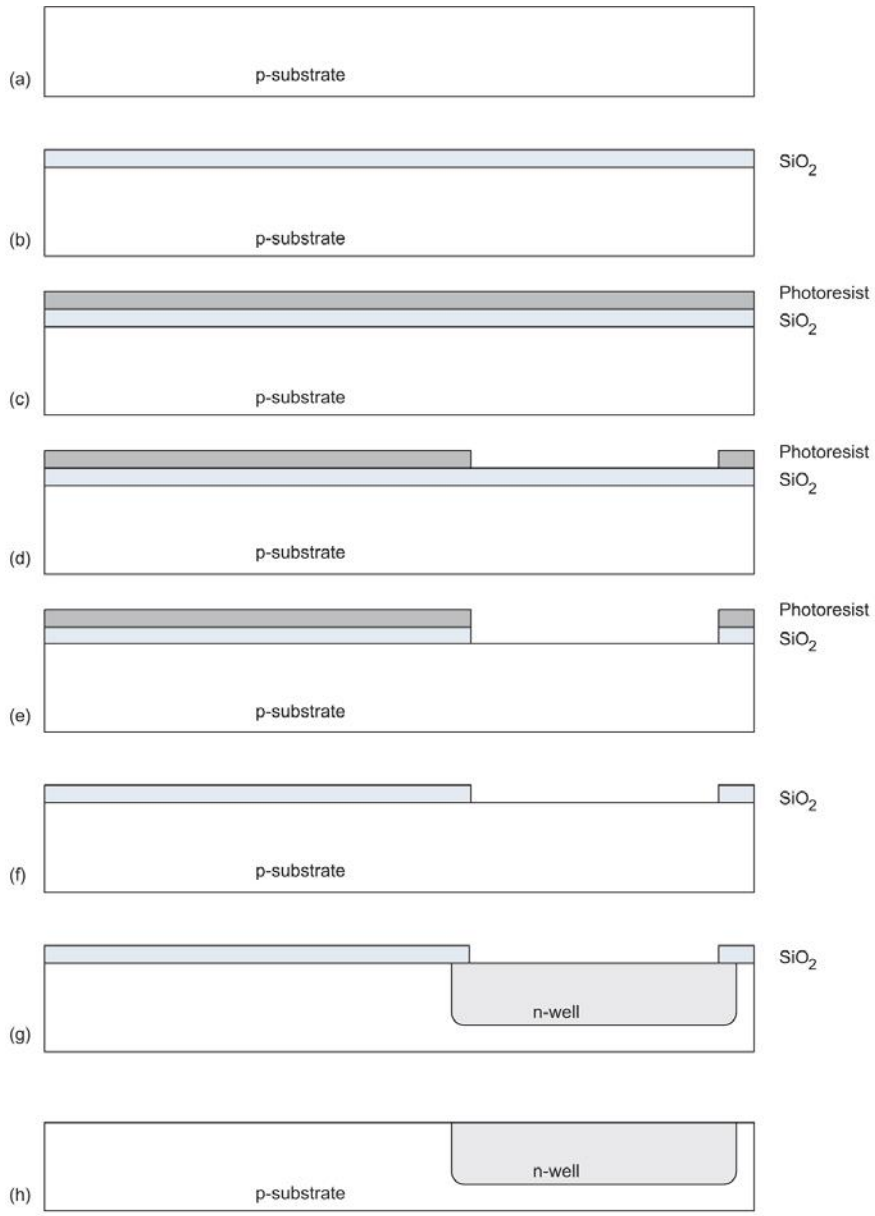


FIG 1.36 Cross-sections while manufacturing the n-well

Fabrication Steps Summary (2/3)

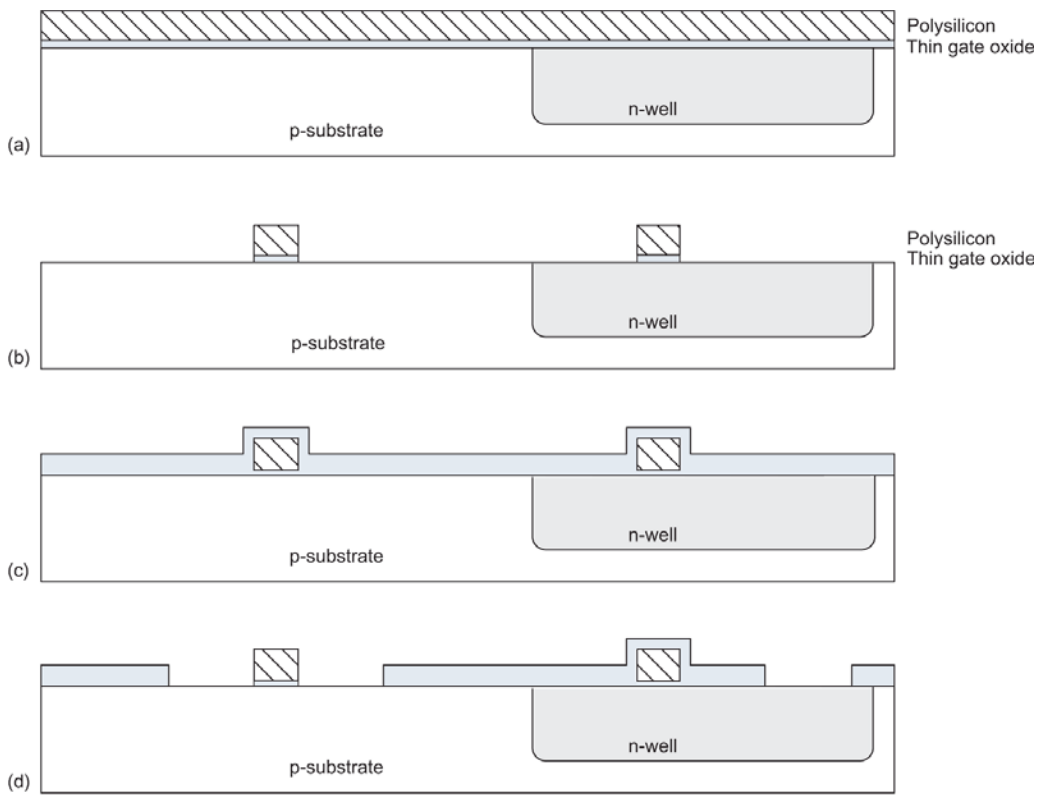


FIG 1.37 Cross-sections while manufacturing polysilicon and n-diffusion

Fabrication Steps Summary (3/3)

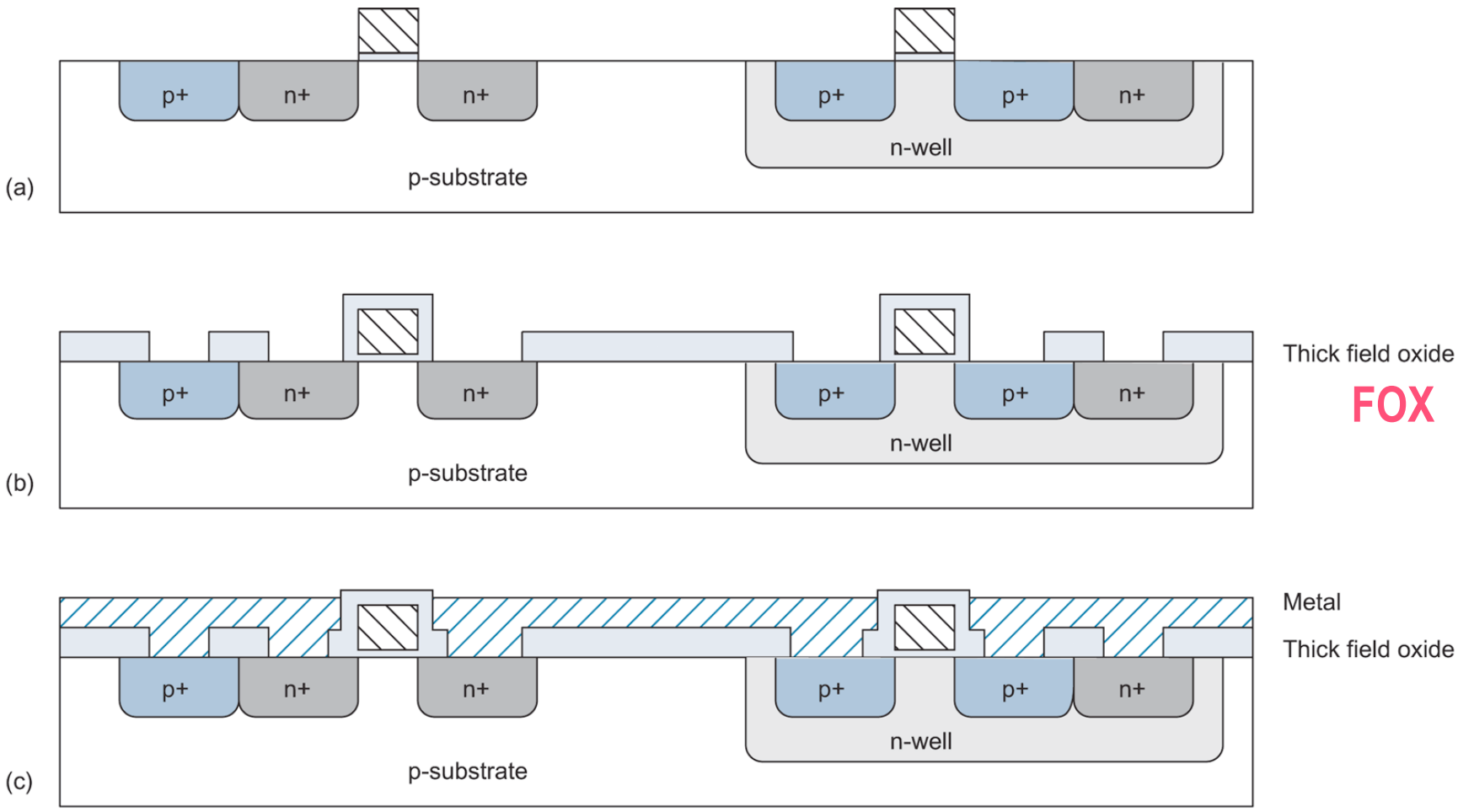


FIG 1.38 Cross-sections while manufacturing p-diffusion, contacts, and metal

Layout Design

- Chips are specified with set of masks
- Minimum dimensions of masks determine transistor size (and hence speed, cost, and power)
- Feature size *f* = distance between source and drain
  - Set by minimum width of polysilicon
- Feature size improves 30% every 3 years or so
- Normalize for feature size when describing design rules
- Express rules in terms of □ = *f*/2
  - E.g. □ = 0.3 μm in 0.6 μm process

Layout Design Rules

Conservative rules to get started !

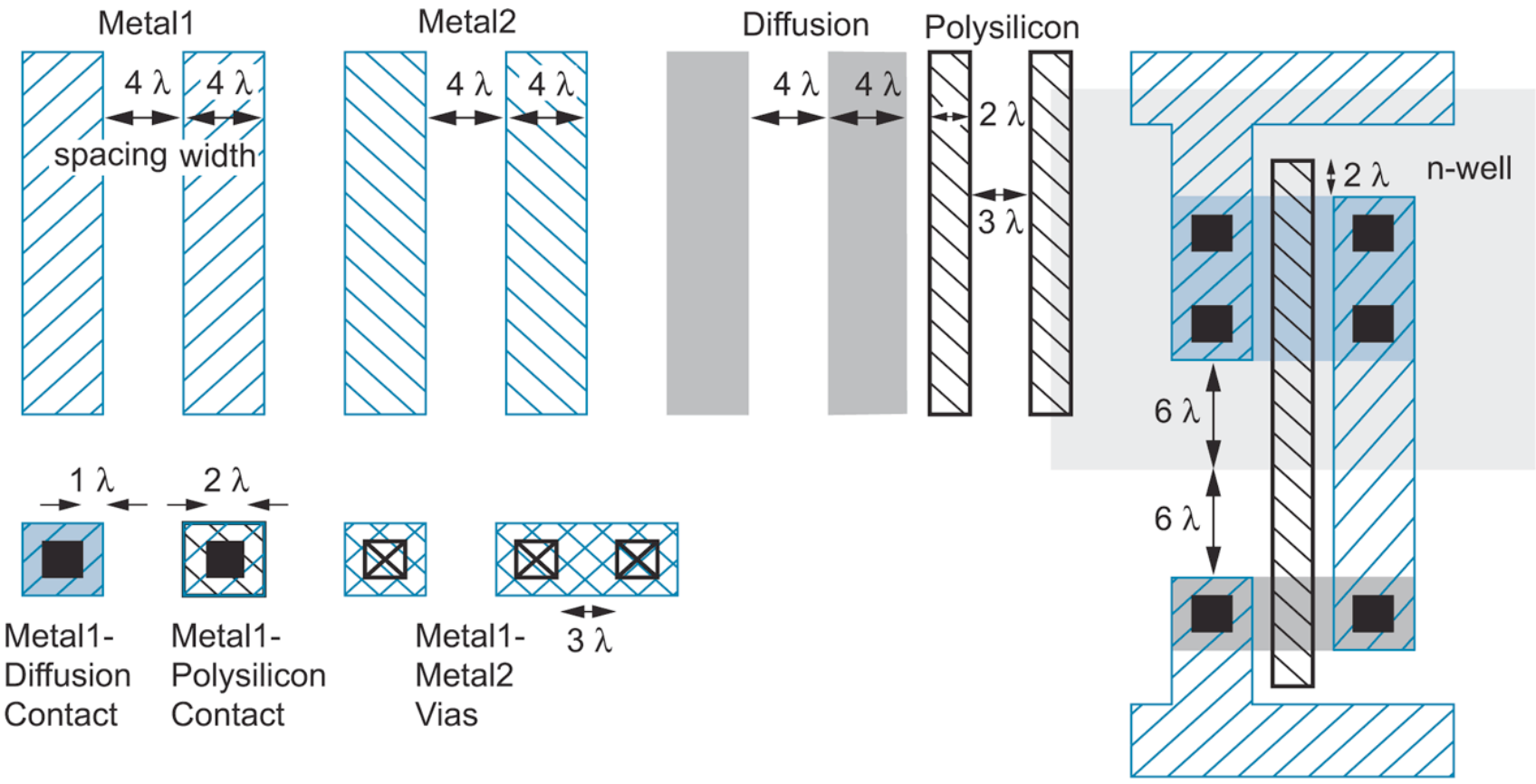


FIG 1.39 Simplified λ-based design rules for layouts with 2-metal layers (MOSIS)

Design Rules Summary

- Metal and diffusion have minimum width and spacing of 4□
- Contacts are 2□ x 2□ and must be surrounded by 1□ on the layers above and below
- Polysilicon uses a width of 2□
- Polysilicon overlaps diffusions by 2□ where a transistor is desired and has spacing or 1□ away where no transistor is desired
- Polysilicon and contacts have a spacing of 3□ from other polysilicon or contacts
- N-well surrounds pMOS transistors by 6□ and avoid nMOS transistors by 6□

Logic Gates layout

- Layout can be very time consuming
- Design gates to fit together nicely
- Build a library of standard cells

- Standard cell design methodology
  - V_{DD} and GND should abut (standard height)
  - Adjacent gates should satisfy design rules
  - nMOS at bottom and pMOS at top
  - All gates include well and substrate contacts

The power and ground lines are called supply rails

Inverter Layout

- Transistor dimensions specified as W / L ratio
- Minimum size is 4□ / 2□, sometimes called 1 unit

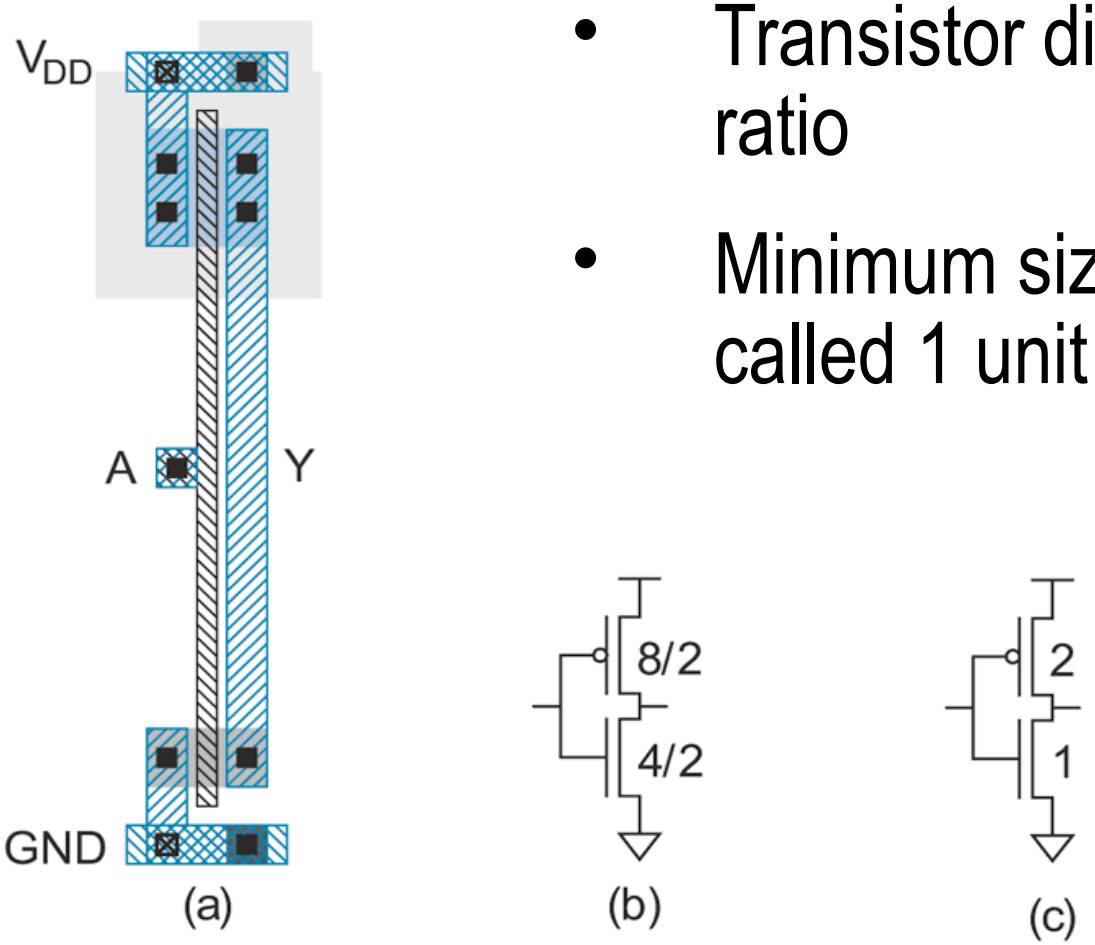
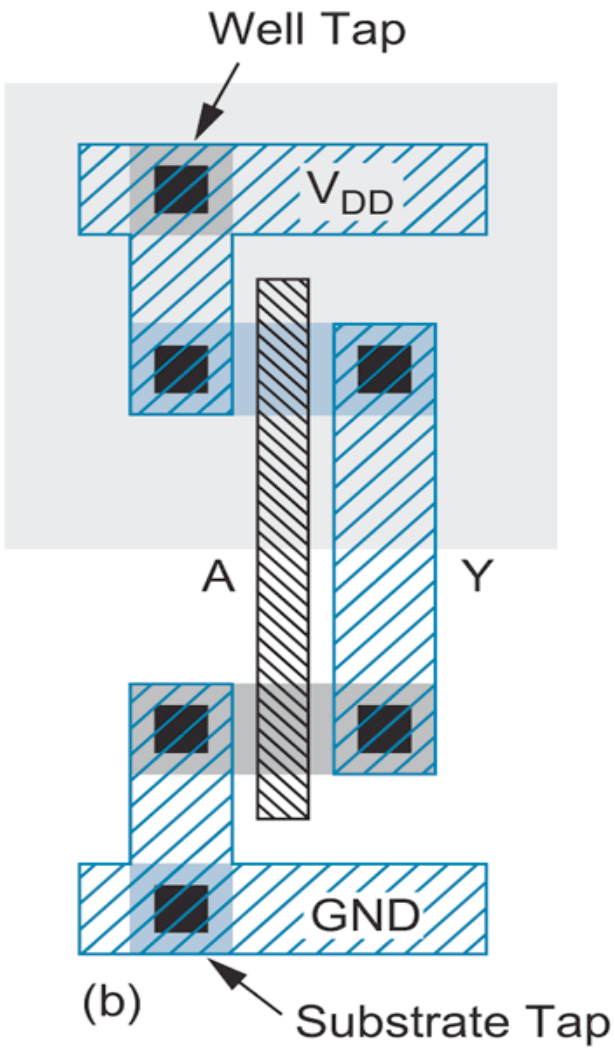


FIG 1.40 Inverter with dimensions labeled

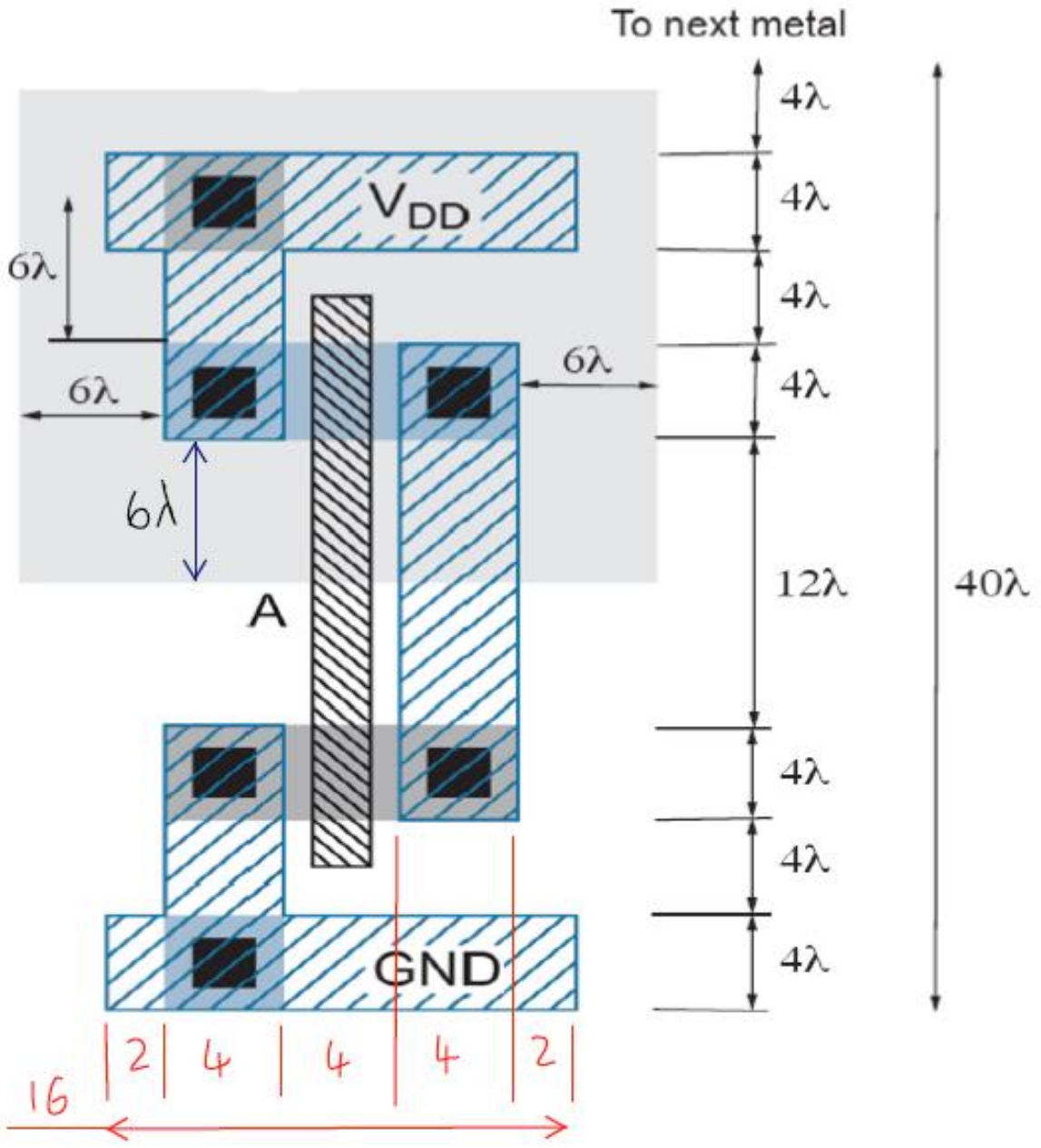
- In *f* = 0.6 μm process, this is 1.2 μm wide, 0.6 μm long

Inverter Standard Cell Layout

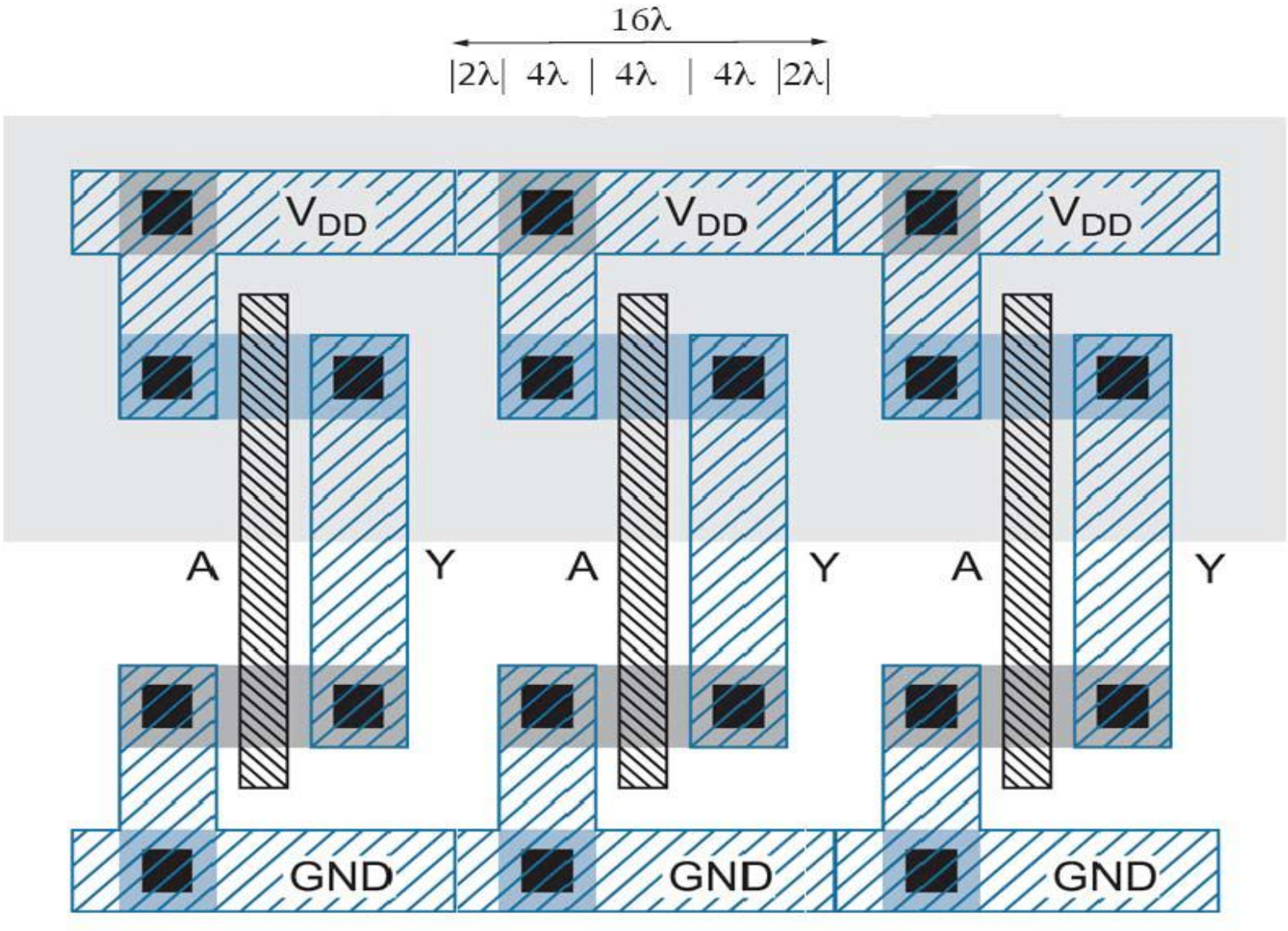


Usually the pMOS has width 2 or 3 times the width of the nMOS

Inverter Standard Cell Area (1/2)



Inverter Standard Cell Area (2)



Three abutted standard cell inverters

3-input Standard Cell NAND

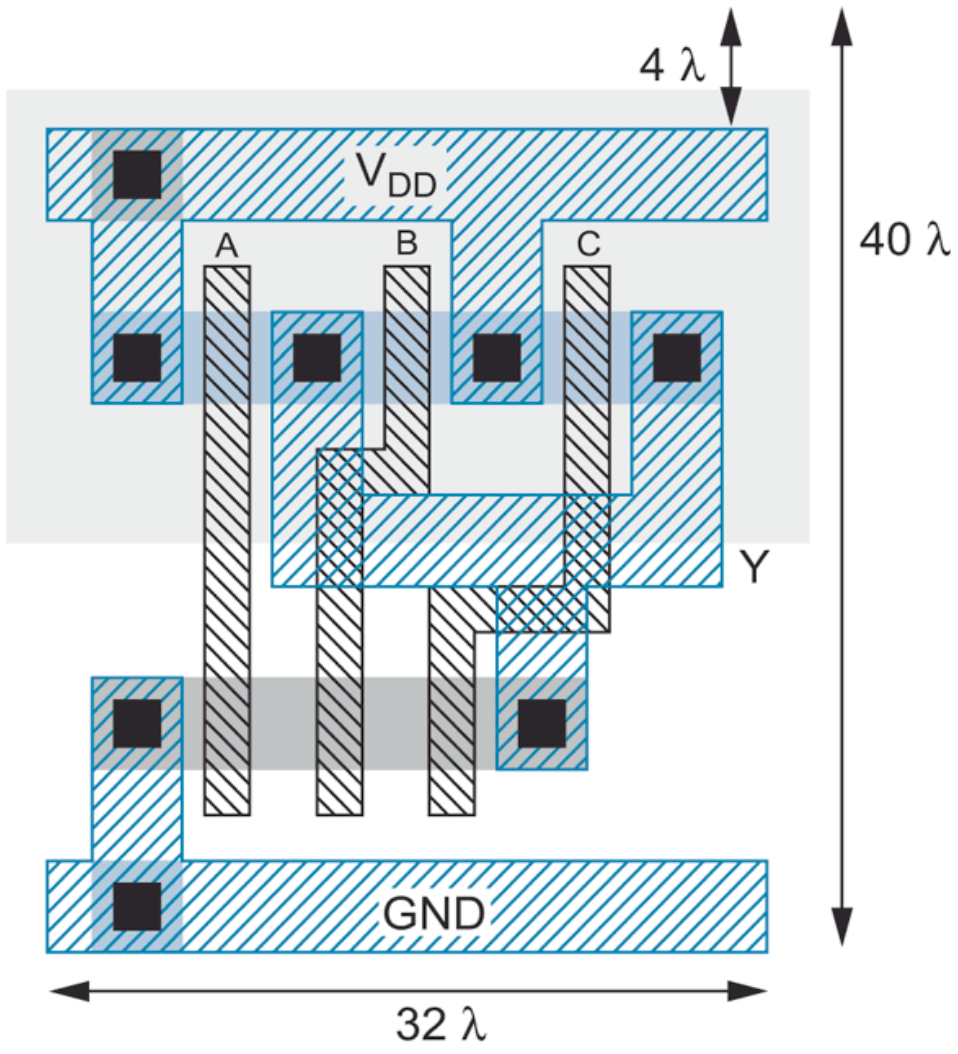


FIG 1.42 3-input NAND standard cell gate layouts

Stick Diagrams

- **Stick diagrams** help plan layout quickly
  - Need not be to scale
  - Draw with color pencils or dry-erase markers

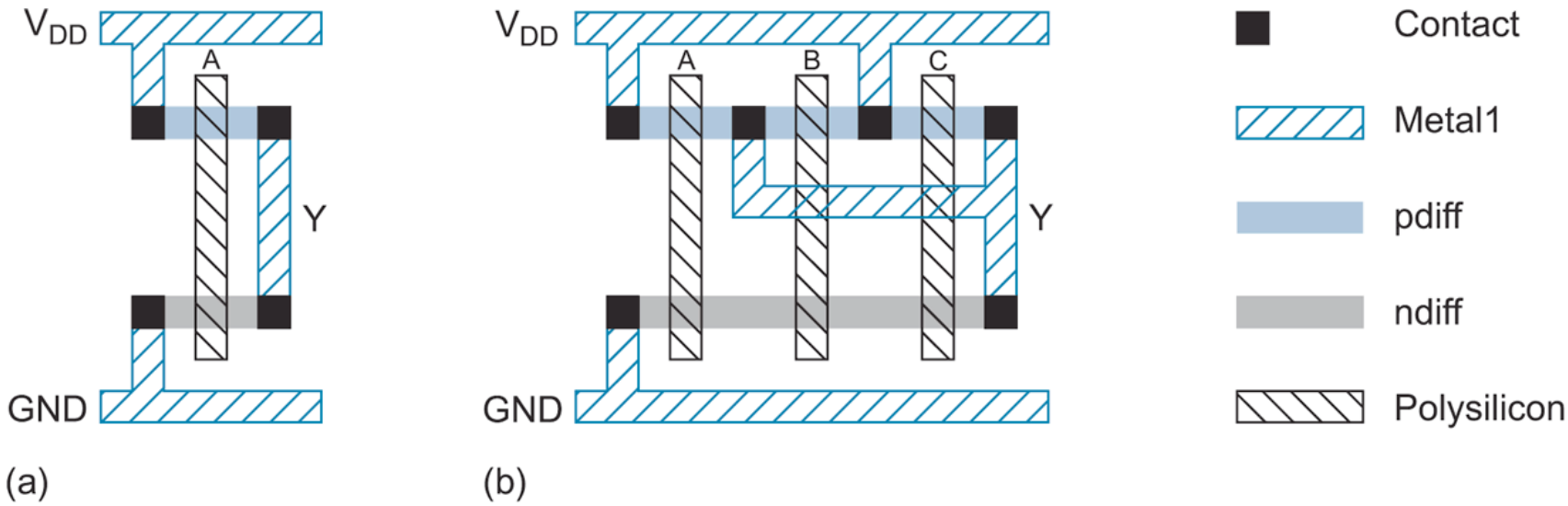


FIG 1.43 Stick diagrams of inverter and 3-input NAND gate. Color version on inside front cover.

Wiring Tracks

- A **wiring track** is the space required for a wire
  - 4□ width, 4□ spacing from neighbor = 8□ pitch
  - Transistors also consume one wiring track

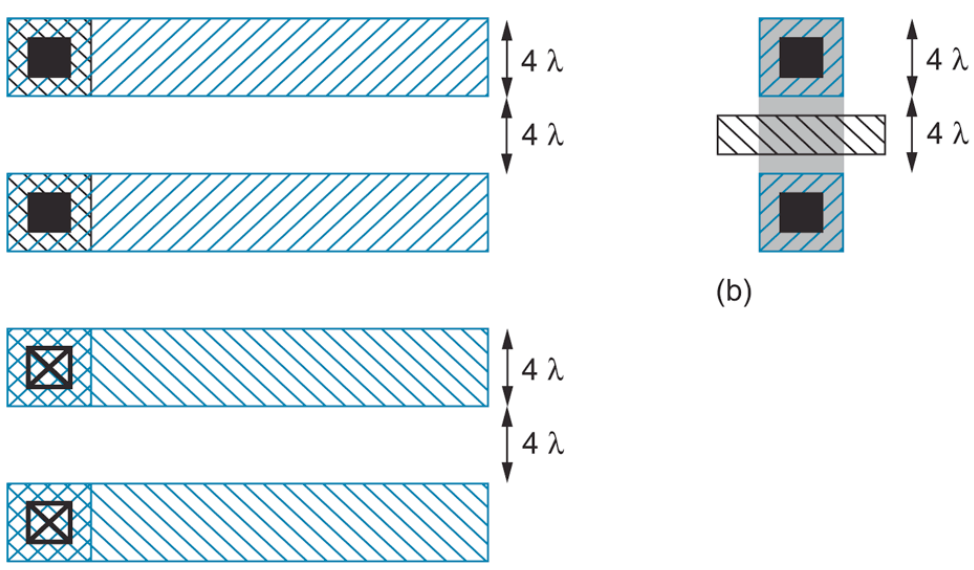


FIG 1.44 Pitch of routing tracks

Well Spacing

- Wells must surround transistors by 6□
  - Implies 12□ between opposite transistor flavors
  - Leaves room for one wire track

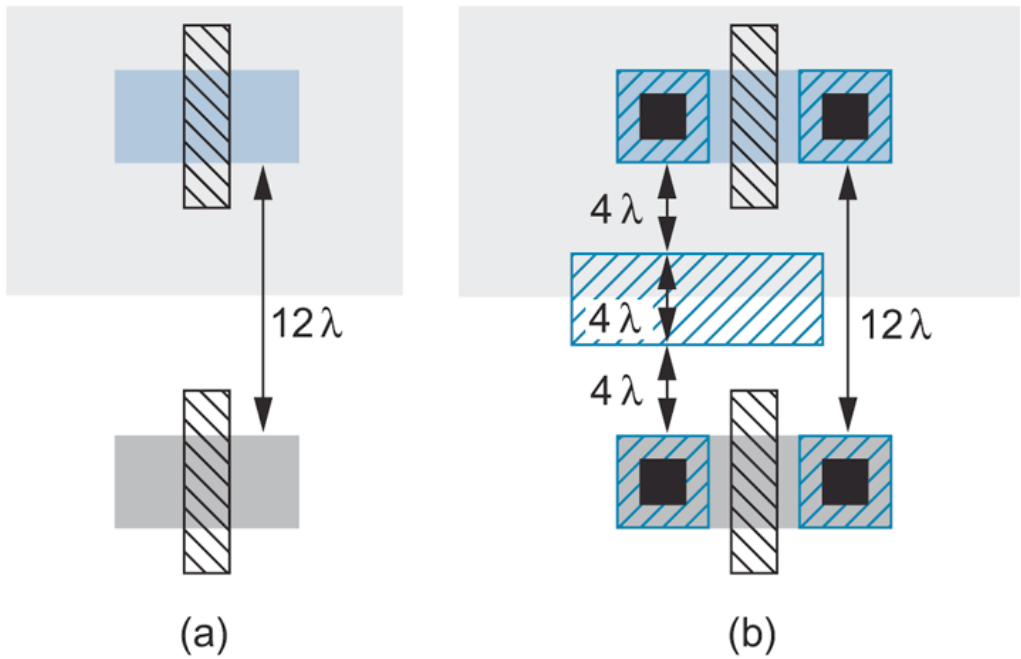


FIG 1.45 Spacing between nMOS and pMOS transistors

Area Estimation

Horizontal
4 × 8 = 32

Vertical
5 × 8 = 40

- Estimate area by counting wiring tracks
  - Multiply by 8 to express in □

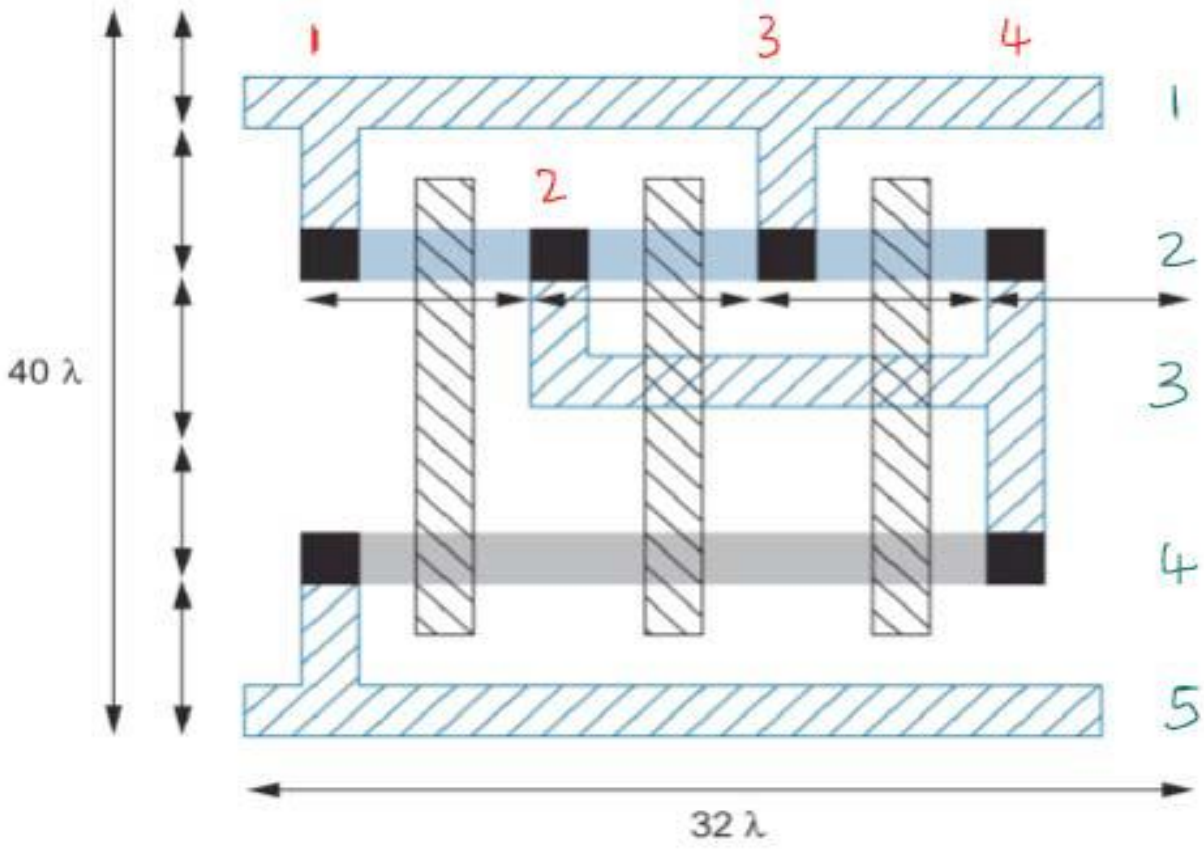
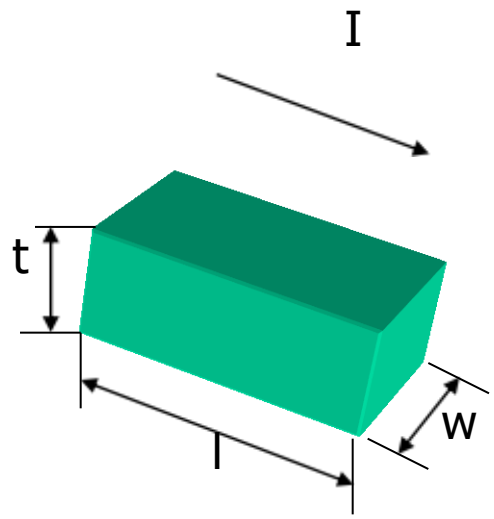


FIG 1.46 3-input NAND gate area estimation

Resistance estimation

- Resistance of uniform slab can be given as,

$$R = \frac{\rho}{t} \cdot \frac{l}{w} \quad ohms$$



Where ρ = resistivity

t= thickness

l= conductor length

w=conductor width

or,

$$R = R_s \cdot \frac{l}{w} \quad ohms$$

R_s is the sheet resistance Ω/□

Resistance estimation (cont.)

- Resistance of certain layers

Material	R_s (Ω/□)
metal	0.03
Poly	15→100
Diffusion p	80
Diffusion n	35
Silicide	2→4
N-well	1K → 5K

Resistance estimation

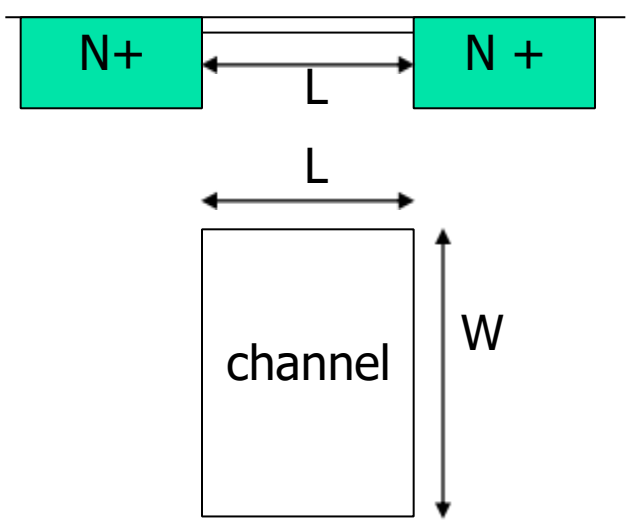
- For MOSFET channel resistance

$$R_{channel} = R_{Sheet}(L/W)$$

where $R_{sheet} = 1/\mu C_{OX}(V_{gs}-V_t)$

For P and n channels

$$R_{sheet} = 1000 \text{ ->} 30{,}000\ \Omega/\square$$



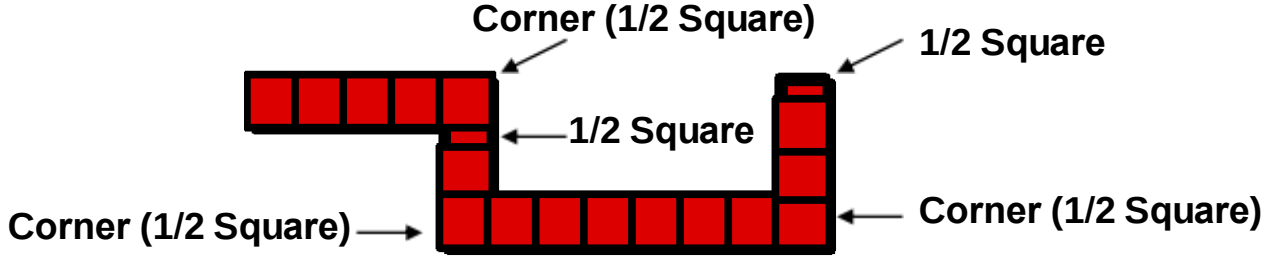
Resistance

- Depends on resistivity of material ρ (Rho)
- Sheet resistance $R_s = \rho / t$
- Resistance $R = R_s * L / W$
- Corner approximation - count a corner as half a square

Example:

$R = R_{s(poly)} * 13 + 2*(1/2) + 3*(1/2)$ squares

$R = 4\Omega/sq * 15.5$ squares $= 62\Omega$



Inverter resistance estimation

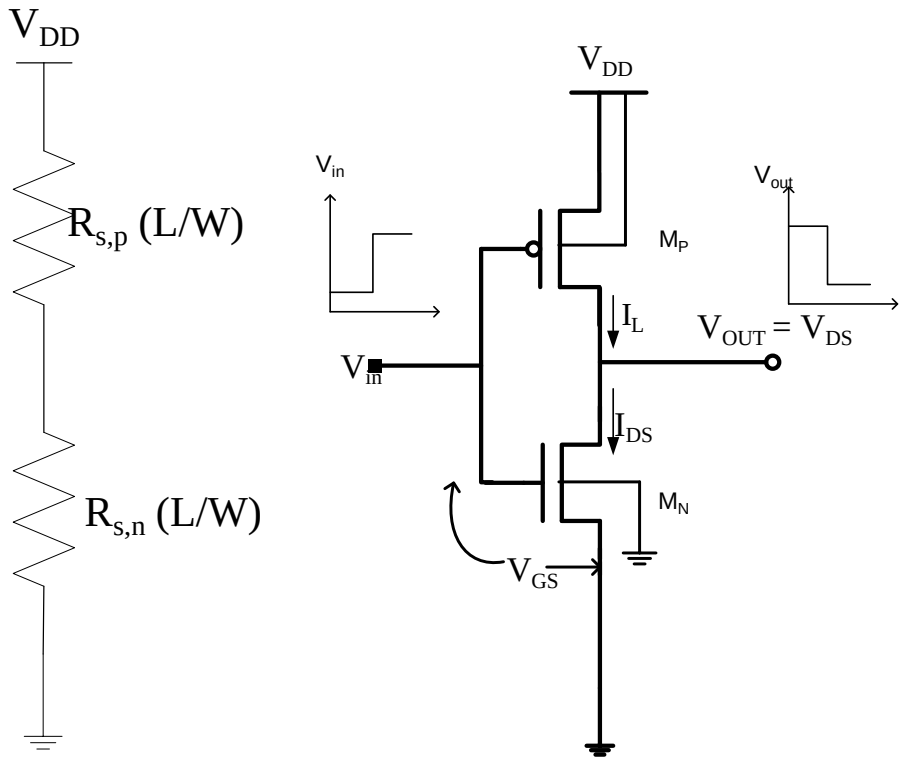
- CMOS inverter (no static current)
- Switching current

$$I_{\max} = \frac{V_{DD}}{R_{total}} = \frac{V_{DD}}{R_{s,p}\frac{L}{W} + R_{s,n}\frac{L}{W}}$$

$$for \quad L = W = 1$$

$$I_{\max} = \frac{V_{DD}}{R_{s,p} + R_{s,n}} = \frac{V_{DD}}{25 + 10} = \frac{V_{DD}}{35}$$

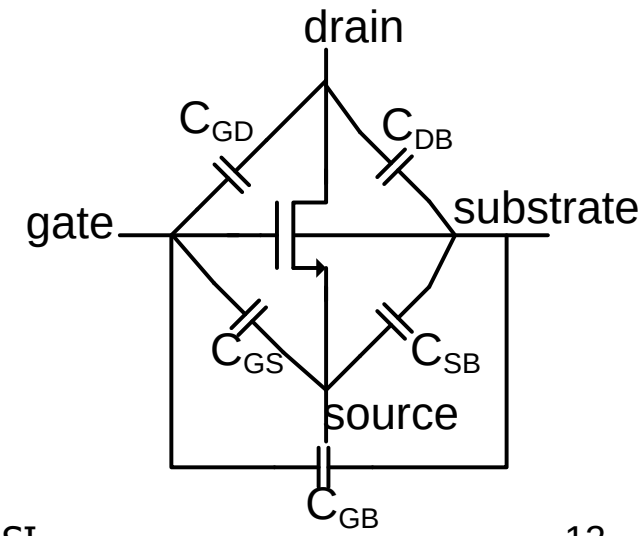
$$switching\ \ power\ \ loss = I_{\max} V_{DD} = \frac{V_{DD}^2}{35}$$



Capacitance estimation

The dynamic response of MOS systems strongly depends on the parasitic capacitances associated with the MOS device. The total load capacitance on the output of a CMOS gate is the sum of:

- gate capacitance (of other inputs connected to out)
- diffusion capacitance (of drain/source regions)
- routing capacitances (output to other inputs)



Capacitance (1/2)

- Transistors
  - Depends on area of transistor gate
  - Depends on physical materials, thickness of insulator
  - Given for a specific process as C_g
- Diffusion to substrate
  - Side-wall capacitance - capacitance from periphery
  - bottom-wall capacitance - capacitance to substrate
  - Given for a specific process as $C_{diff,bot}$, $C_{diff,side}$

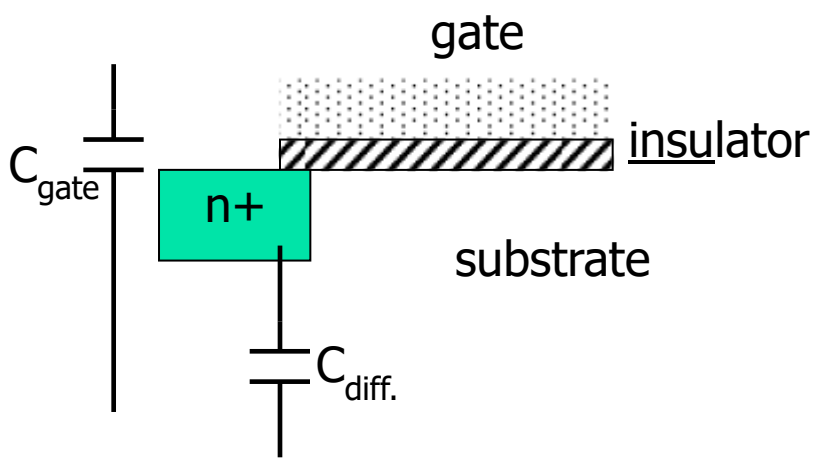
Capacitance (2/2)

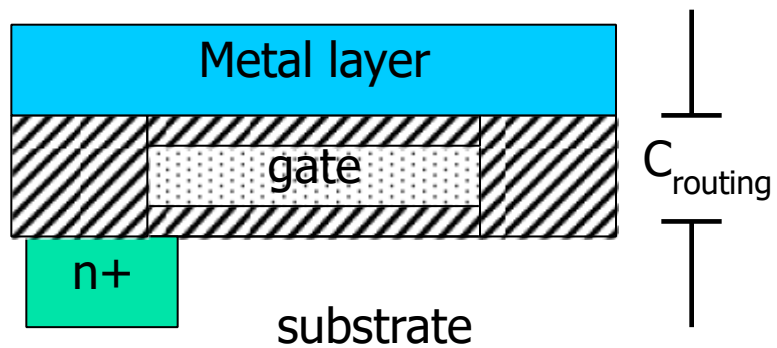
- Metal to substrate
  - Parallel plate capacitance is dominant
  - Need to account for fringing, too
- Poly to substrate
  - Parallel plate plus fringing, like metal
  - don’t confuse poly over substrate with gate capacitance
- Also important: capacitance between conductors
  - Metal1-Metal1
  - Metal1-Metal2

Capacitance estimation (cont.)

- Gate capacitance
- Diffusion capacitance
- Routing capacitance

- $C_{diff} > C_{poly} > C_{m1} > C_{m2}$



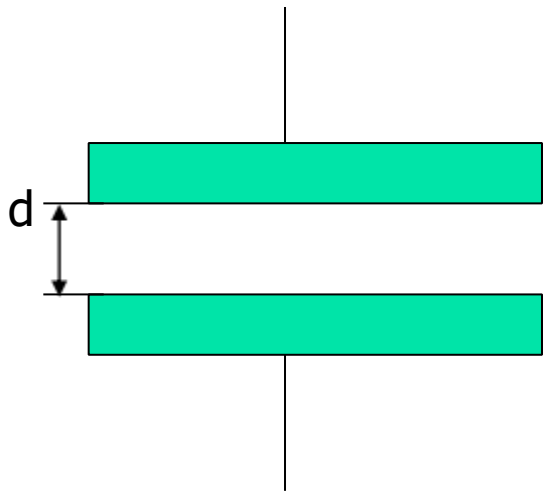


Capacitance estimation

- In general, capacitance could be calculated using

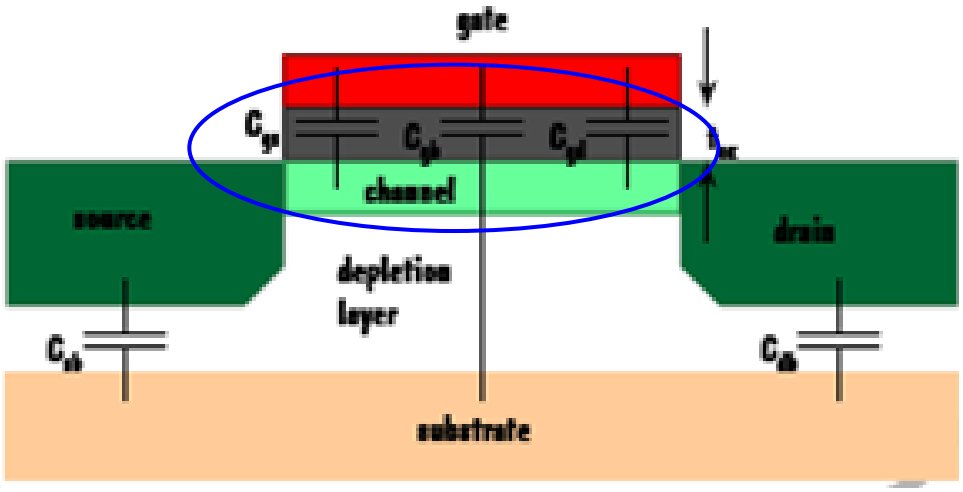
$$C = \frac{\varepsilon . A}{d} = \frac{\varepsilon_o . \varepsilon_r . A}{d}$$

$$C_{/unitarea} = \frac{\varepsilon_o . \varepsilon_r}{d} = C_{ox}$$



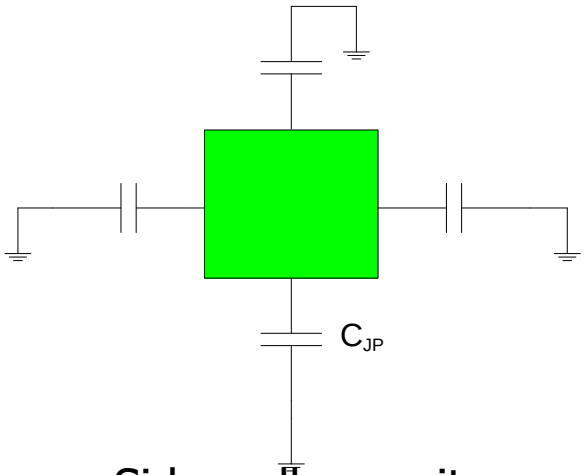
Gate Capacitance

- $C_g = C_{gs} + C_{gd} + C_{gb}$

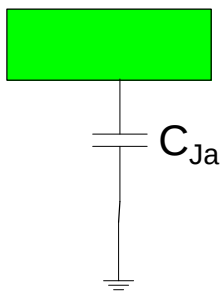


Capacitance estimation (cont.)

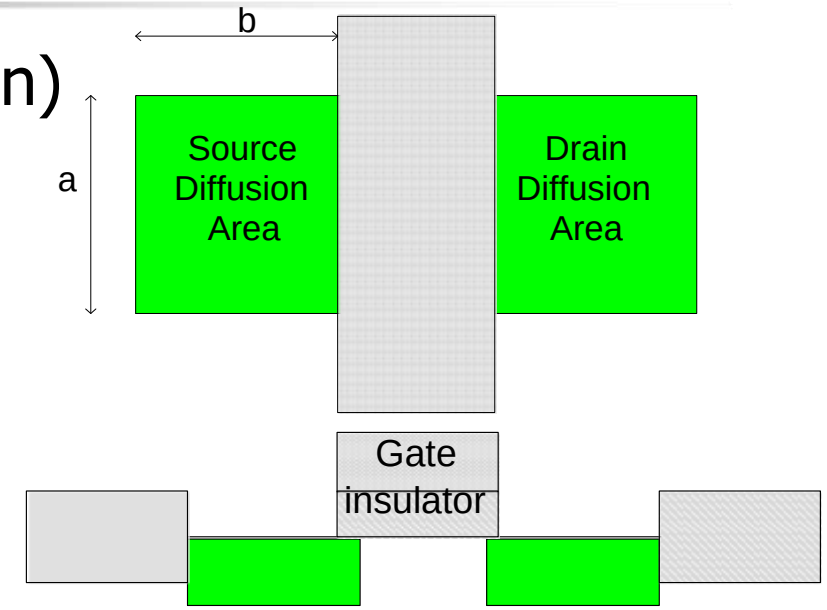
- Diffusion capacitance (source/drain)



Side wall capacitance



area capacitance



$$C_{s,diff} = C_{d,Area}.A + C_{d,sidewalls}.P$$

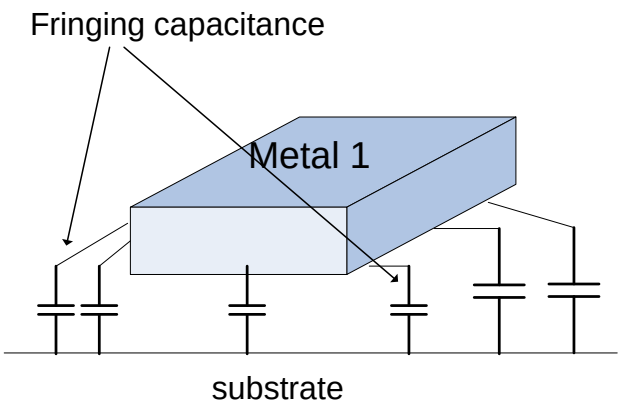
Where A = area and p = perimeters

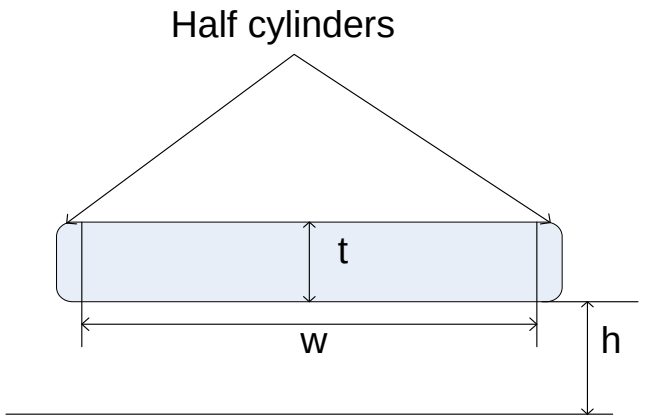
Routing capacitance

- single conductor capacitance
- multiple conductor capacitance

Capacitance estimation (cont.)

Routing capacitance: a) single conductor capacitance

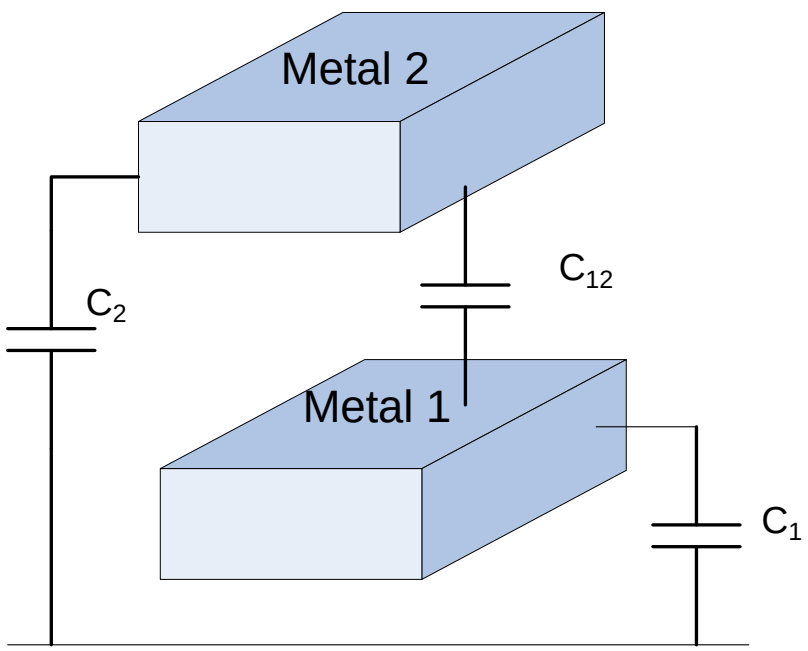


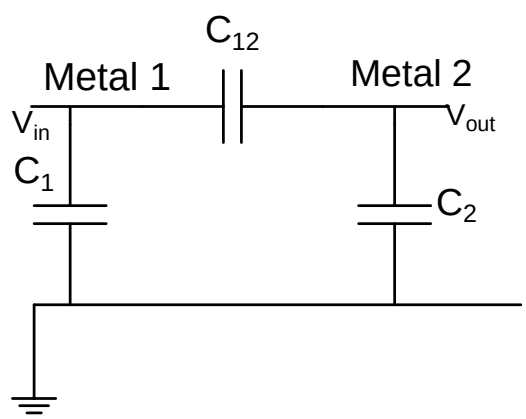


$$C_{total} \quad \varepsilon = \left[\frac{w - \frac{t}{2}}{h} + \frac{2\,\Pi}{\ln\left[1 + \frac{2}{\frac{h}{t}} + \sqrt{\left(\frac{2}{\frac{h}{t}}\right)^2 + 2}\right]} \right]$$

Capacitance estimation (cont.)

Routing capacitance: b) multiple conductor capacitance





$$\Delta V_{out} = \Delta V_{in} \cdot \frac{C_{12}}{C_2 + C_{12}}$$

Multilayer capacitance calculations

- Example: given the layout shown in the figure calculate the total capacitance at source and gate given that:

$C_{metal/Area} = 0.025\mu F/\mu m^2$

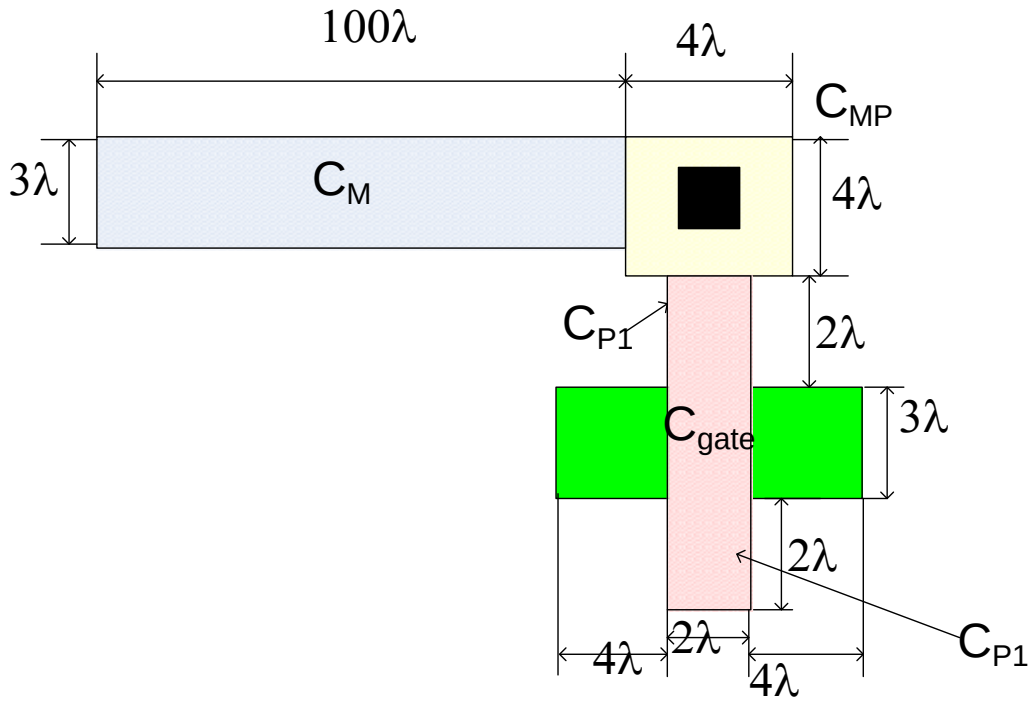
$C_{poly/Area} = 0.045\mu F/\mu m^2$

$C_{Gate/A} = 0.7\ fF/\mu m^2$

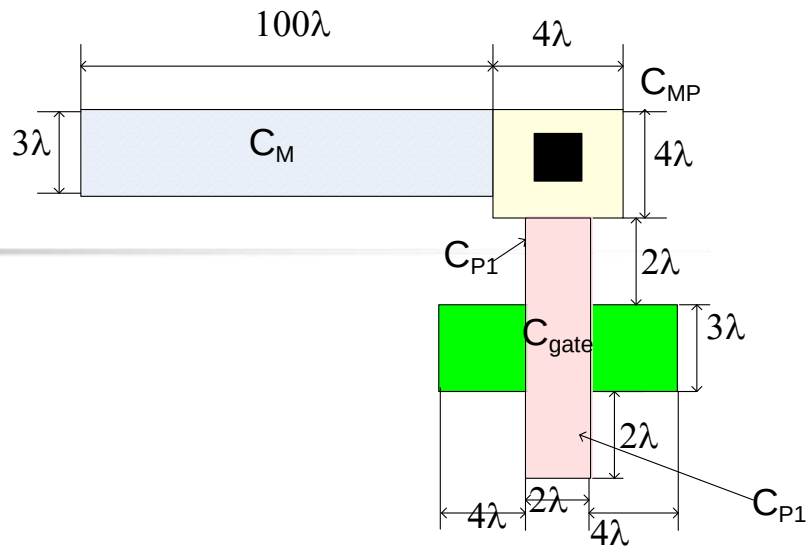
$C_{d,a/A} = 0.33fF/\mu m^2$

$C_{d,side/L} = 2.6fF/\mu m$

$\lambda = 5.1\mu m$



Solution



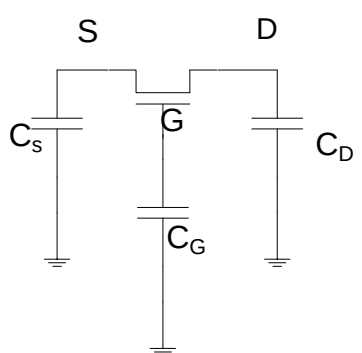
Source capacitance

$$C_{S,diff} = C_{d,A} \cdot A + C_{d,side\ walls} \cdot P$$

$$A = 4\lambda * 3\lambda = 12\ \lambda^2$$

$$P = 2*(4\lambda + 3\lambda) = 14\ \lambda$$

So, $C_{S,\ diff} = 0.33*\ 12\ \lambda^2 + 2.6*\ 14\ \lambda = 63.51fF$



Solution (cont.)

Gate capacitance

$$C_{G,total} = C_M + C_{MP} + C_{P1} + C_G + C_{P2}$$

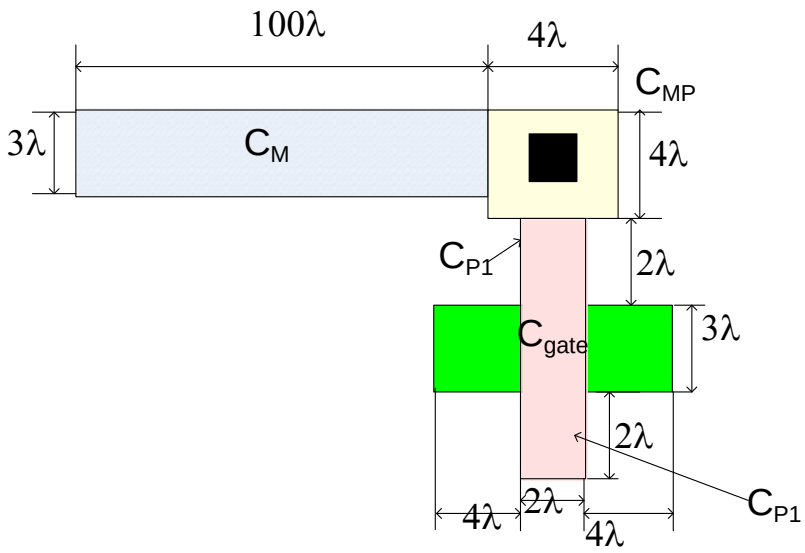
$$C_M = 0.025 * 100\lambda * 3\lambda = 7.5\lambda^2$$

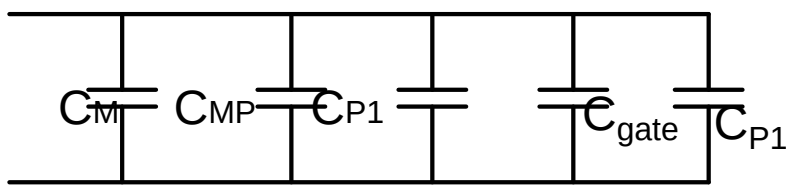
$$C_{MP} = 0.045 * 4\lambda * 4\lambda = 0.72\lambda^2$$

$$C_{P1} = 0.045 * 2\lambda * 2\lambda = 0.18\lambda^2$$

$$C_{P2} = 0.045 * 2\lambda * 2\lambda = 0.18\lambda^2$$

$$C_G = 0.7 * 2\lambda * 3\lambda = 4.2\lambda^2$$





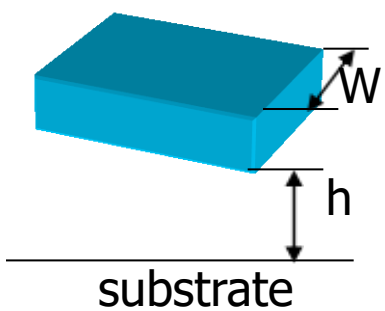
Inductance estimation

- Inductance is normally small but as the process shrink on-chip inductance must be taken into account.
- Bond-wire inductance can cause deleterious effects in large, high speed I/O buffers.
- The inductance of bonding wires and the pins on packages could be calculated by,

$$L = \frac{\mu}{2\Pi} \ln\left(\frac{8h}{w} + \frac{4}{d}\right)$$

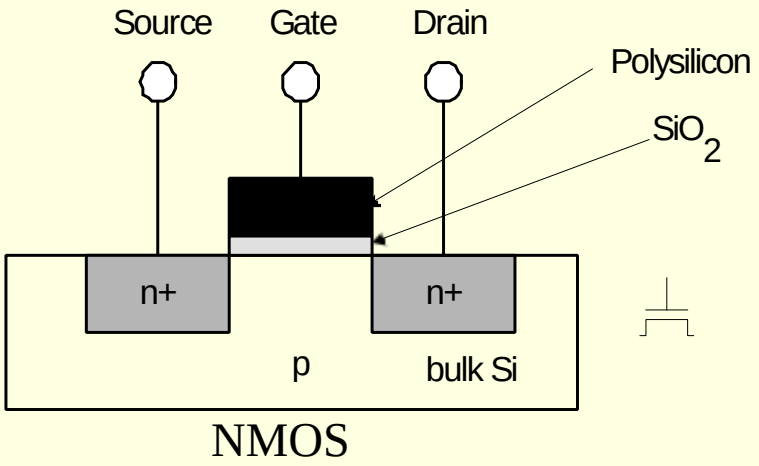
Design techniques to overcome this problem:

- ✓separate power pins for I/O pads and chip core
- ✓multiple power and ground pins
- ✓careful selection of the position of the power and ground pins on the package
- ✓adding decoupling capacitances on the board
- ✓increase the rise and fall times
- ✓use advanced package technologies (SMD, etc)

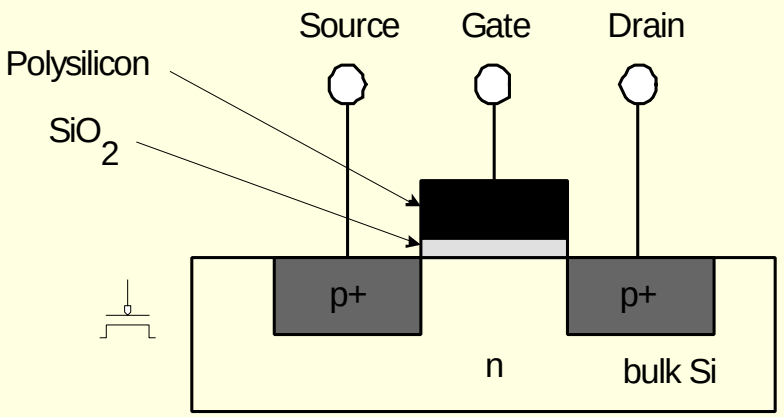


MOS Transistors

- Four terminal device: gate, source, drain, body
- Gate – oxide – body stack looks like a capacitor
  - Gate and body are conductors (body is also called the substrate)
  - SiO_2 (oxide) is a “good” insulator (separates the gate from the body
  - Called metal–oxide–semiconductor (MOS) capacitor, even though gate is mostly made of poly-crystalline silicon (polysilicon)



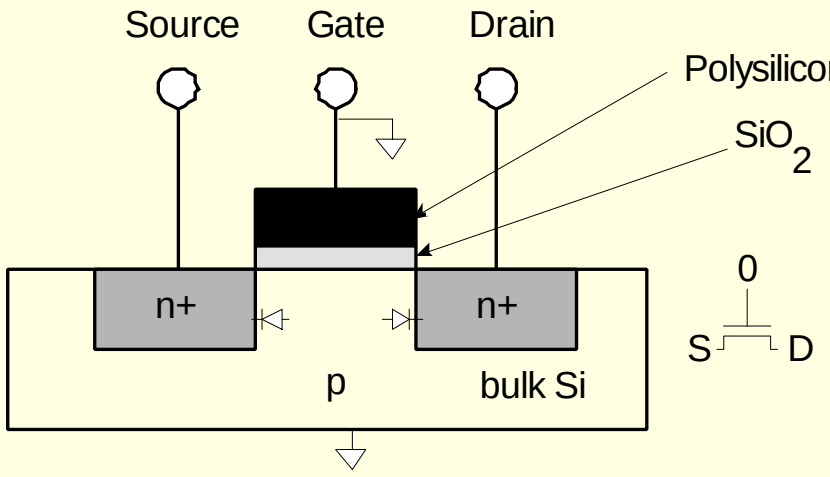
NMOS



PMOS

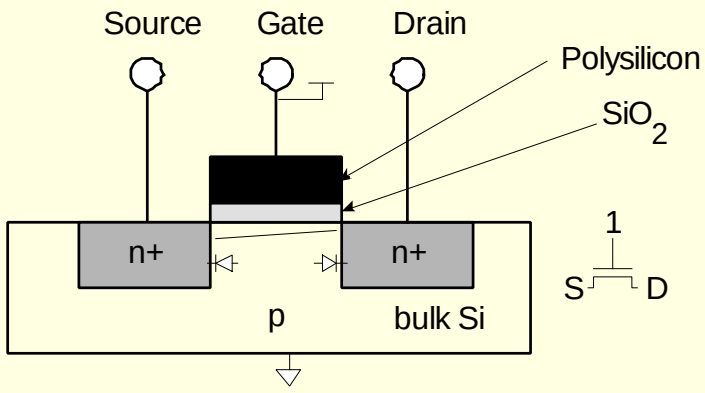
NMOS Operation

- Body is commonly tied to ground (0 V)
- Drain is at a higher voltage than Source
- When the gate is at a low voltage:
  - P-type body is at low voltage
  - Source-body and drain-body “diodes” are OFF
  - No current flows, transistor is OFF



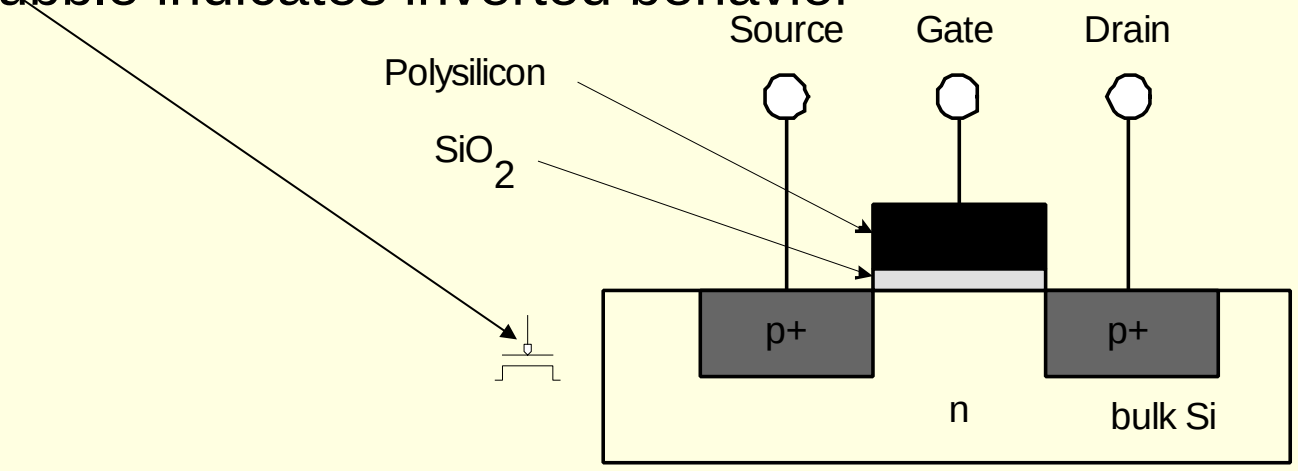
NMOS Operation Cont.

- When the gate is at a high voltage: Positive charge on gate of MOS capacitor
  - Negative charge is attracted to body under the gate
  - Inverts a channel under gate to “n-type” (N-channel, hence called the NMOS) if the gate voltage is above a threshold voltage (VT)
  - Now current can flow through “n-type” silicon from source through channel to drain, transistor is ON



PMOS Transistor

- Similar, but doping and voltages reversed
  - Body tied to high voltage (V_{DD})
  - Drain is at a lower voltage than the Source
  - Gate low: transistor ON
  - Gate high: transistor OFF
  - Bubble indicates inverted behavior



Ninevah University

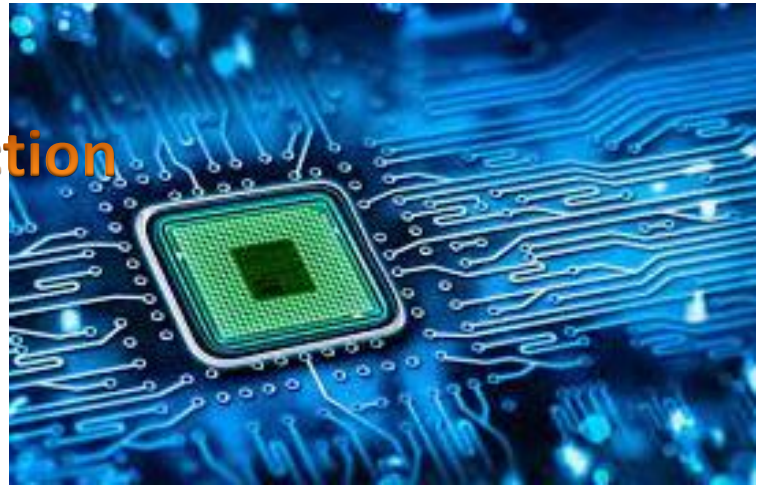
Collage of Electronics Engineering

Course:

Electronic Design Automation

Lecturer: H. M. Hussein

EDA01: Introduction



Course Outline

1- Digital circuit design flow.
2- Verilog Hardware description language
3- Logic Synthesis
    - Multilevel logic minimization.
    - Technology mapping
    - High-level synthesis
4- Testability Issues
5- Physical Design Automation
    - Floor planning, Placement, Routing.

Digital Design Flow

. Design complexity increased rapidly

- Increased size and complexity
- CAD tools are essential

. The present trend

- Standardize the design flow.

What is design flow?

. Standardized design procedure

- starting from the design idea down to the actual implementation.

. Encompasses many steps

- Specification
- Synthesis
- Simulation
- Layout
- Testability analysis
- Many **more…**

New CAD tools

- Based on Hardware Description Language (HDL)

- HDLs provide formats for representing the outputs of various design steps.

- An HDL based CAD tool transforms from its HDL input into HDL output which contains more hardware information.

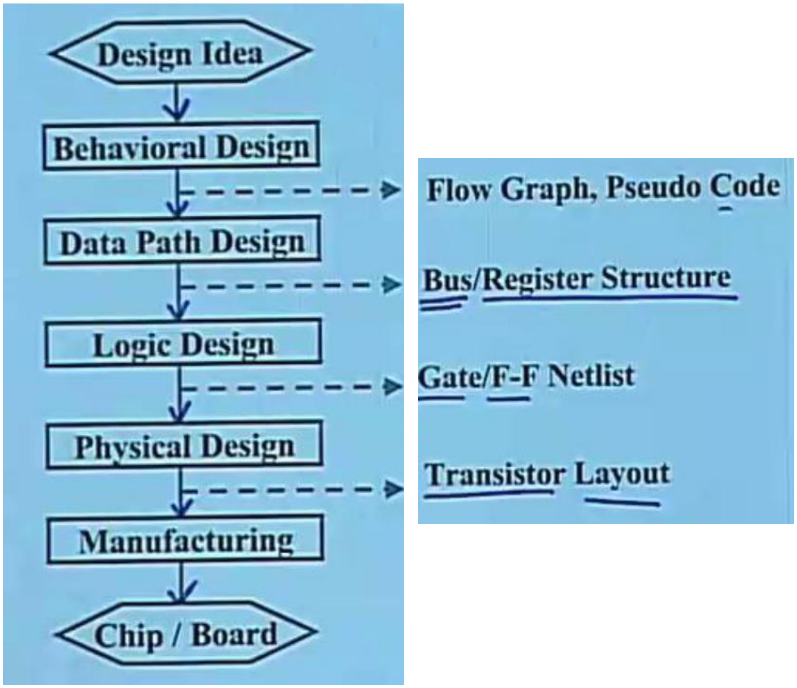
. Behavioral level to register transfer level.
. Register transfer level to gate level.
. Gate level to transistor level.

Two compting HDLs

1-VHDL

2-Verilog

Simplistic view of Design flow:

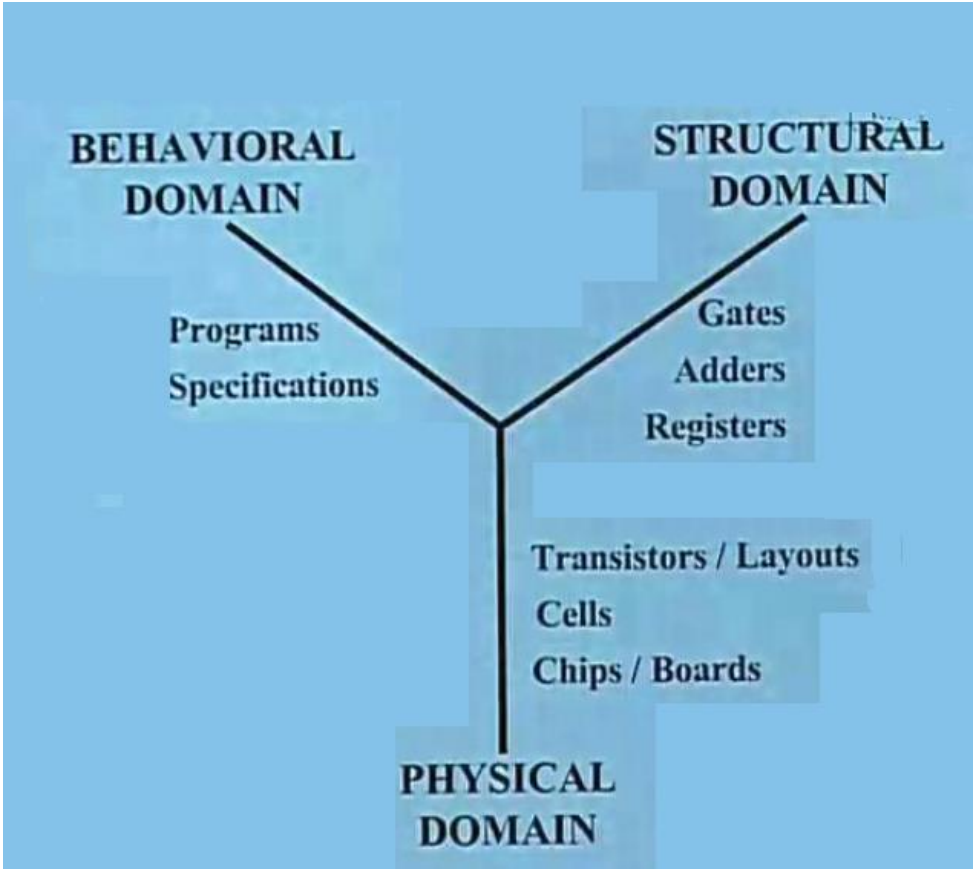


Design Representation

. A design can be represented at various levels from three different points of view:

1- Behavioral
2- Structural
3- Physical

. can be represented by **Y-diagram**


BEHAVIORAL DOMAIN
STRUCTURAL DOMAIN
Programs
Specifications
Gates
Adders
Registers
Transistors / Layouts
Cells
Chips / Boards
PHYSICAL DOMAIN

Behavioral Representation:

. Specifies how a particular design should respond to given set of inputs.

. May be specified by:

- Boolean equations.

- Tables of input and output values.

- Algorithms written in standard HLL like C.

- Algorithms written in special HDL like Verilog.

An algorithm level description:

```
module carry(cy, a,b,c);
        input a,b,c;
        output y;
        assign
           cy = (a&b)|(b&c)|(c&a)
endmodule
```

Boolean behavioral specification for cy:

```
primitive carry(cy, a,b,c);
        input a,b,c;
        output y;
        talbel
           // a  b  c  :  cy
              1  1  ? : 1
              1  ?  1 : 1
              ?  1  1 : 1
              0  0  ? : 0
              1  ?  0 : 0
              ?  0  0 : 0
        End table
endprimitive
```

Structural Representation

. Specifies how components are interconnected.

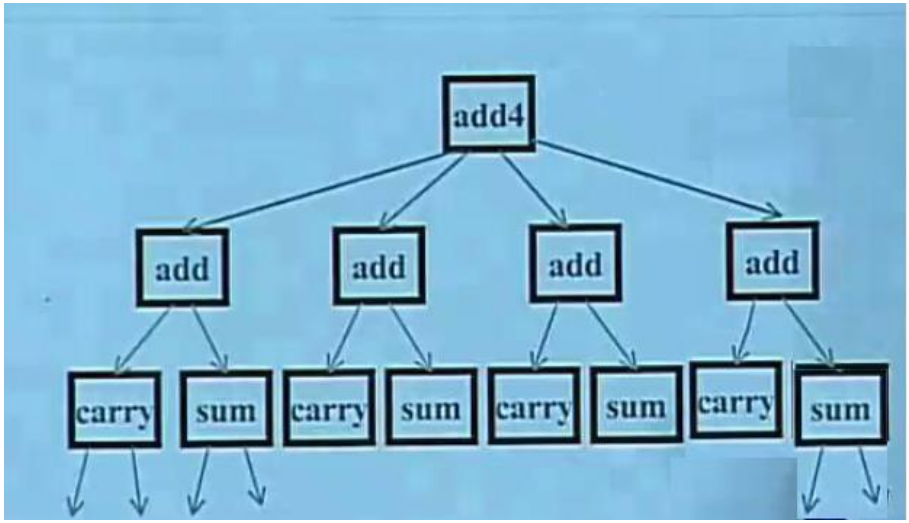
. In general, the description is a list of modules and their interconnects.

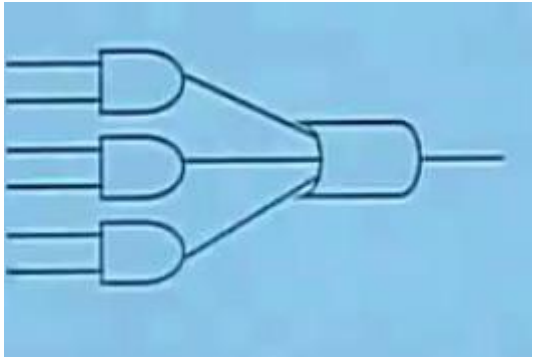
- called **netlist**.
- can be specified at various levels.

. At the structural level of, the level of abstraction are:

- the module level
- the gate level
- the switch level
- the circuit level

. In each level more details is revealed about implementation.





Structural representation : example

4- Bit adder

```
module add4(s,cy4,cy_in,x,y);
        input [3:0] x, y;
        input cy_in;
        output [3:0] s;
        output cy4;
        wire [2:0] cy_out;
                add B0 (cy_out[0],s[0],x[0],y[0],cy_in);
                add B1 (cy_out[1],s[1],x[1],y[1], cy_out[0]);
                add B2 (cy_out[2],s[2],x[2],y[2], cy_out[1]);
                add B3 (cy4,          s[3],x[3],y[3], cy_out[2]);
endmodule
```

```
module add (cy_out,sum,a,b,cy_in);
        input a,b,cy_in;
        output sum, cy_out;
                sum s1(sum,a,b,cy_in);
                carry c1(cy_out,a,b,cy_in);
endmodule
```

```
module carry (cy_out ,a,b,cy_in);
        input a,b,cy_in;
        input cy_out;
        wire t1,t2,t3;
                and g1(t1,a,b);
                and g2(t2,a,c);
                and g3(t3,b,c);
                or    g4(cy_out,t1,t2,t3);
endmodule
```

Physical Representation:

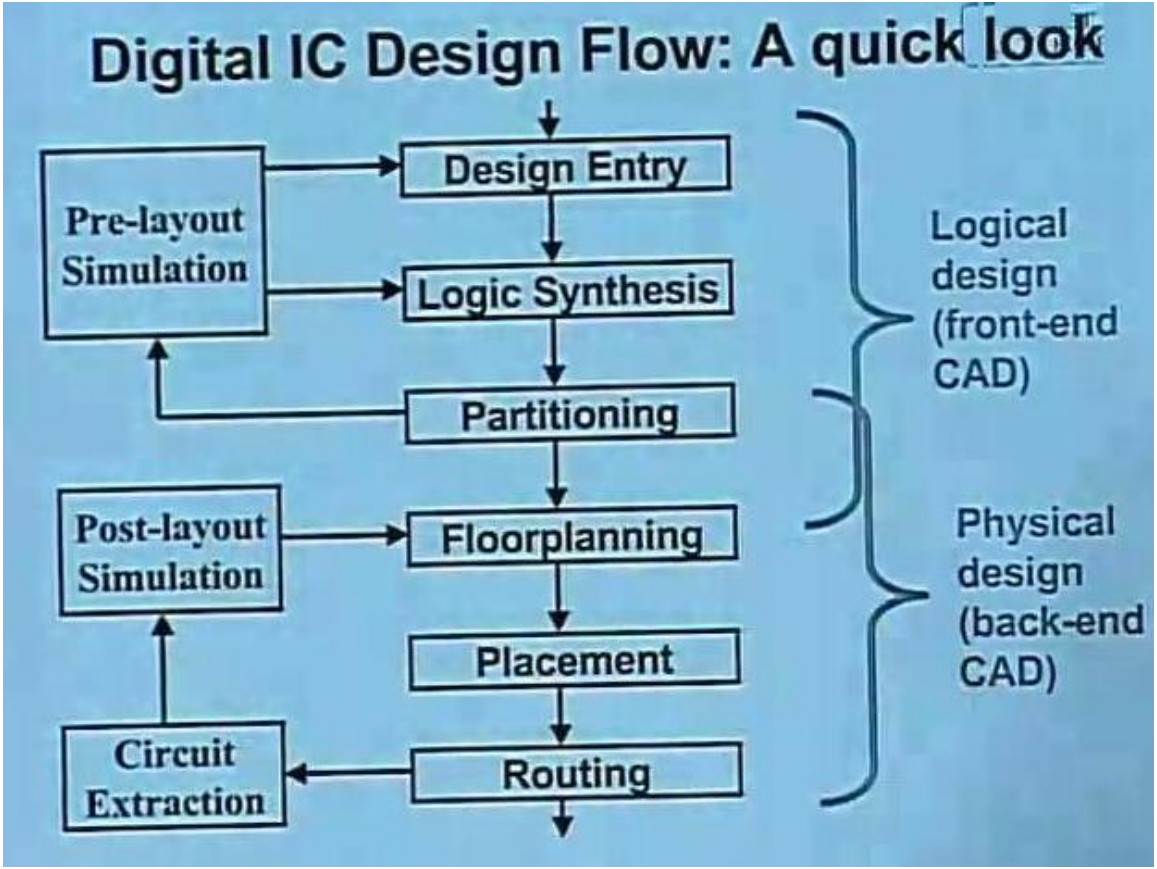
. The lowest level of physical specification:

- Photo-mask information required by the various processing steps in the fabrication process.

. At the module level, the physical layout for 4-bit adder may be defined by a rectangle or polygon, and a collection of ports.

Physical Representation: example;

```
module add4(s,cy4,cy_in,x,y);
        input x[3:0] , y[3:0];
        input cy_in;
        output [3:0] s;
        output cy4;
        boundary [0,0,130,500];
        port x[0] aluminum width=1 origin=[0,35];
        port y[0] aluminum width=1 origin=[0,85];
        port cy_in polysilicon width=2 origin=[70,0];
        port s[0] aluminum width=1 origin=[120,65];

        add a0 origin=[0,0];
        add a1 origin=[0,120];
endmodule
```


Digital IC Design Flow: A quick look
Design Entry
Pre-layout Simulation
Logic Synthesis
Partitioning
Post-layout Simulation
Floorplanning
Placement
Circuit Extraction
Routing
Logical design (front-end CAD)
Physical design (back-end CAD)

About Verilog:

. Along with VHDL, Verilog is among the most widely used HDLs.

. Main differences:

- VHDL was designed to support system-level design and specification.
- Verilog was designed primarily for digital hardware designers developing FPGAs and ASICs.

. The differences become clear if someone analyze the language features.

. VHDL

- Provides some high-level constructs not available in Verilog (user defined types, configurations, etc.).

. Verilog

- Provides comprehensive support for low-level digital design. Not available in native VHDL(type definition and called packages need to be included)

Concept of Verilog " module"

. In Verilog, the basic unit of hardware is called **module.**

- Modules cannot contain definitions of other modules.
- A module can, however, be instantiated within another module.
- Allows the creation of hierarchy in Verilog description.

```
module module_name (list_of_ports);
        input declaration;
        output declaration;
        local net declaration;
        parallel statements;
endmodule
```

Example 1: simple AND gate

```
module simpleand (f,x,y);
        input x,y;
        output f;
        assign f=x & y;
endmodule
```

Example 2: two-level circuit

```
module two-level (a,b,c,d,f);
        input a,b,c,d;
        output f;
        wire t1,t2;
        assign t1= a & b;
        assign t2= ~(c|d);
        assign f= t1 ^ t2;
endmodule
```

Example 3: a hierarchical design

```
module add3 (s,cy3,cy_in,x,y);
        input [2:0] x,y;
        input cy_in;
        output [2:0] s;
        output cy3;
        wire [1:0] cy_out;
        add B0 (cy_out[0],s[0],x[0],y[0],cy_in);
        add B1 (cy_out[1],s[1],x[1],y[1], cy_out[0]);
        add B2 (cy3         ,s[2],x[2],y[2], cy_out[1]);

endmodule
```

Specifying Connectivity:

. There are two alternate ways of specifying connectivity:

- Positional association
  - The connections are listed in the same order

    add A1(c_out,sum,a,b,c_in);
- Explicit association
  - May be listed in any order

add A1(.in1(a), .in2(b),.cin(c_in),.sum(sum),.cout(c_out));

Variable Data Types

. A variable belongs to one of two data types:

- Net
  - Must be continuously driven
  - Used to model connections between continuous assignments & instantiations.
- Register
  - Retains the last value assigned to it
  - Often used to represent storage elements

Net data type:

. Different ‘net’ types supported for synthesis:

- wire , wor, wand, tri, supply0, supply1

. ‘wire’ and ‘tri’ are equivalent; when there are multiple drivers driving them, the outputs of the drivers are shorted together.

. ‘wor’ / ‘wand’ inserts OR / AND gate at the connection.

. ‘supply0’ / supply1 model power supply connections.

Example 4:

```
module using_wire (a,b,c,d,f);
        input a,b,c,d;
        output f;
        wire f;
        assign f= a & b;
        assign f= c|d;
endmodule
```

```
module using_wired_and (a,b,c,d,f);
        input a,b,c,d;
        output f;
        wand f;   //f as wand
        assign f= a & b;
        assign f= c|d;
endmodule
```

Example 5:

```
module using_supply_wire (a,b,c,f);
        input a,b,c;
        output f;
        wire t1,t2;
        supply0 gnd;
        supply1 vdd;

        nand G1 (t1,vdd,a,b);
        xor G2 (t1,c,gnd);
        and G3(f,t1,t2);
endmodule
```

Register data type:

. Different ‘register’ types supported for synthesis:

- reg, integer

. the ‘reg’ declaration explicitly specifies the size.

- reg x,y; // single- bit register variable
- reg [15:0] bus; //16-bit bus, bus [15] MSB
- unsigned
- used to model the actual hardware register

. For ‘integer’, it takes the default size, usually 32-bit.

- Synthesizer tries to determine the size.
- Sinegd
- Used for loop counting

Example 6:

```
module simple_counter (clk,rst,count);
        input clk,rst;
        output count;
        reg [31:0] count;

        always @(posedge clk)
        begin

                if (rst)
                        count = 32'b0;
                else
                        count = count +1;
        end
endmodule
```

- When ‘integer’ is used, the synthesis system often caries out a data flow analysis of the model to determine its actual size.
- .

```
wire [1:10] a,b;
integer     c;
c = a +b;
```

- The size of c can be determined to be equal 11 (10 bits plus a carry).
- A value may be specified in either the ‘sized’ or the ‘un-sized’ form > syntax: <size>’<base><numbr>
-

Examples:

```
8'b01110011  // 8-bit binary number
12'hA2D      // 1010 0010 1101 in binary
12'hCx5      // 1100 xxxx 0101 in binary
25           // signed number, 32 bits
1'b0         // logic 0
1'b1         // logic 1
```

Silicon Manufacturing

Why silicon?

Semiconductor devices are of two forms

(i) Discrete Units

(ii) Integrated Units

Discrete Units can be diodes, transistors, etc.

Integrated Circuits uses these discrete units to make one device.

Integrated Circuits can be of two forms

(i) Monolithic-where transistors, diodes, resistors are fabricated and interconnected on the same chip.

(ii) Hybrid- in these circuits, elements are discrete form and others are connected on the chip with discrete elements externally to those formed on the chip

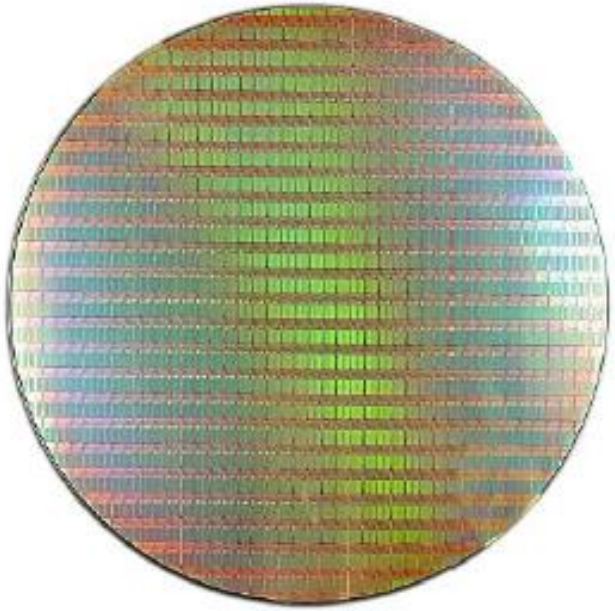
Why silicon?

- ✓ Two other semiconductors, germanium and gallium arsenide, present special problems while silicon has certain specific advantages not available with the others.
- ✓ A major advantage of silicon, in addition to its abundant availability in the form of sand, is that it is possible to form a superior stable oxide, SiO_2, which has superb insulating properties.
- ✓ Both Si and Ge do not suffer, in the processing steps, from possible decomposition that may occur in compound semiconductors such as GaAs.
- ✓ Lastly, at the present time, silicon remains the major semiconductor in the industry.

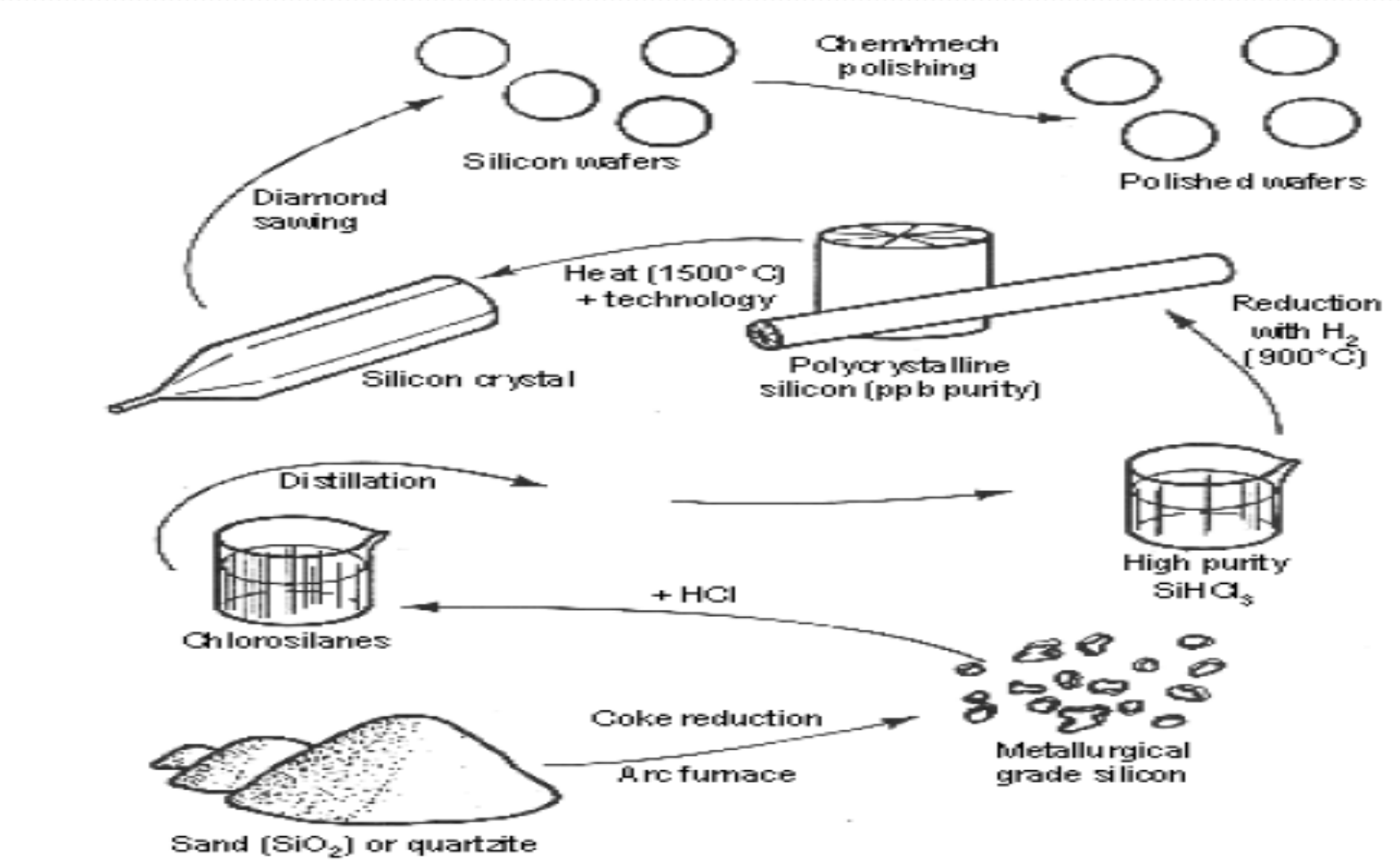
How to product pure silicon

Silicon From Sand

?

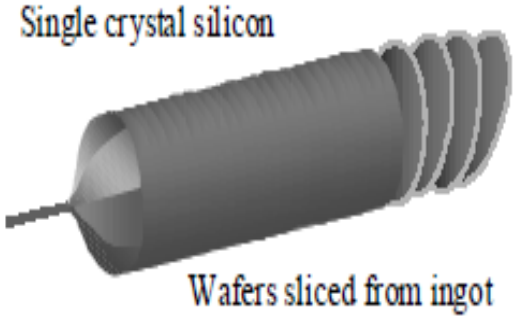


STEPS OF PRODUCTION

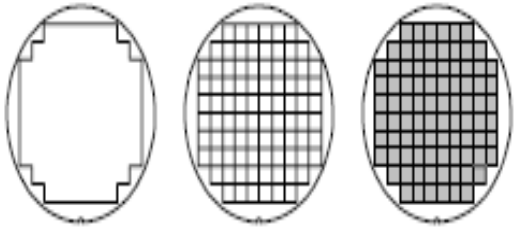


STEPS OF PRODUCTION

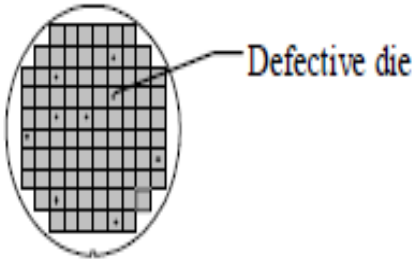
1. **Wafer Preparation** includes crystal growing, rounding, slicing and polishing.



2. **Wafer Fabrication** includes cleaning, layering, patterning, etching and doping.

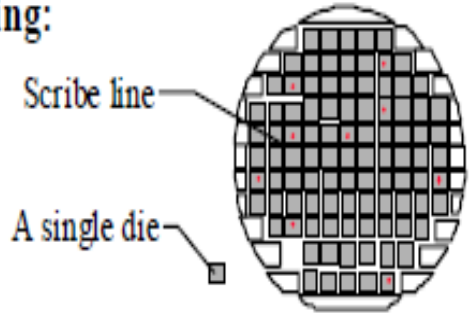


3. **Test/Sort** includes probing, testing and sorting of each die on the wafer.

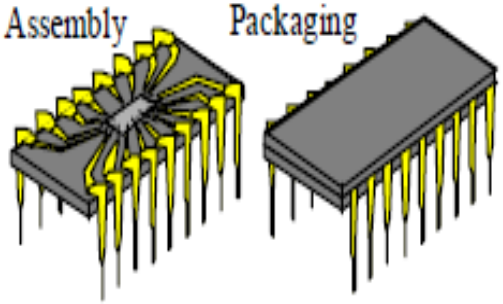


4. **Assembly and Packaging:**

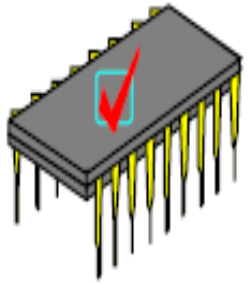
The wafer is cut along scribe lines to separate each die.



Metal connections are made and the chip is encapsulated.



5. **Final Test** ensures IC passes electrical and environmental testing.



Semiconductor Manufacturing Process

Fundamental Processing Steps

1. Silicon Manufacturing
   a) Czochralski method.
   b) Wafer Manufacturing
   c) Crystal structure

2. Photolithography
   a) Photoresists
   b) Photomask and Reticles
   c) Patterning

Semiconductor Manufacturing Process

3.Oxide Growth & Removal

- **a) Oxide Growth & Deposition**
- **b) Oxide Removal**
- **c) Other effects**
- **d) Local Oxidation**

4. Diffusion & Ion Implantation

- **a) Diffusion**
- **b) Other effects**
- **c) Ion Implantation**

Semiconductor Manufacturing Process

Oxidation

The process of oxidation consists of growing a thin film of silicon dioxide on the surface of the silicon wafer.

Diffusion

This process consists of the introduction of a few tenths to several micrometers of impurities by the solid-state diffusion of dopants into selected regions of a wafer to
form junctions.

Ion Implantation

This is a process of introducing dopants into selected areas of the surface of the wafer by bombarding the surface with high-energy ions of the particular dopant.

Semiconductor Manufacturing Process

Photolithography

In this process, the image on the reticle is transferred to the surface of the wafer.

Epitaxy

Epitaxy is the process of the *controlled growth of a crystalline doped layer of silicon on a single crystal substrate.*

Metallization and interconnections

After all semiconductor fabrication steps of a device or of an integrated circuit are completed, it becomes necessary to provide metallic interconnections for the integrated circuit and for external connections to both the device and to the IC.

Semiconductor Manufacturing Process

Etching Techniques

Etching is the process of selective removal of regions of a semiconductor ,metal, or silicon dioxide.

Diffusion

•Most of these diffusion processes occur in two steps: the *predeposition* and the *drive-in* diffusion.

Silicon Manufacturing

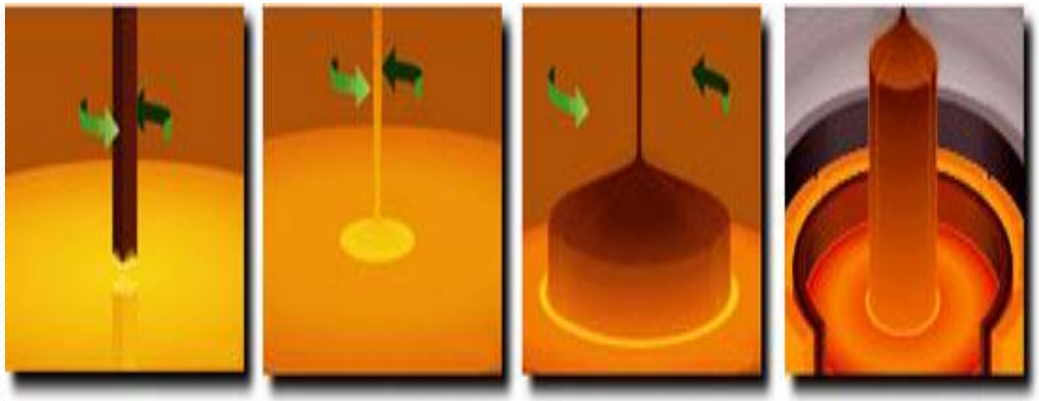
Crystal Growth and Wafer Manufacturing

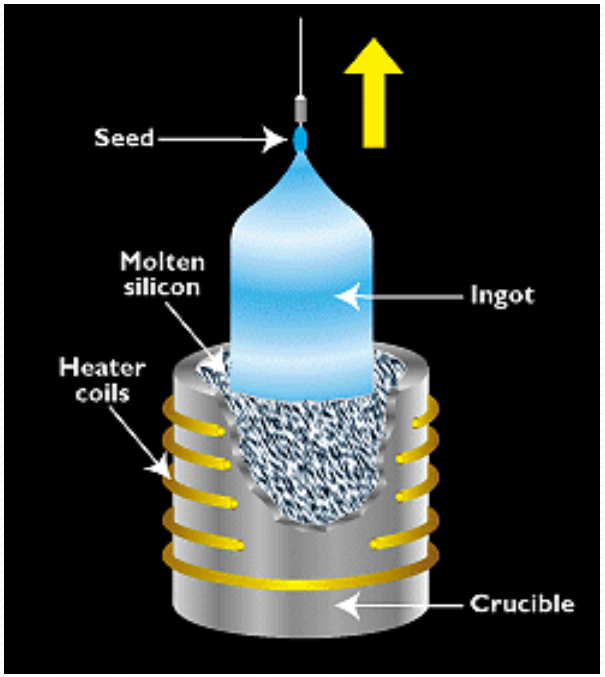
FABRICATING SILICON

- Quartz, or Silica, Consists of Silicon Dioxide
- Sand Contains Many Tiny Grains of Quartz
- Silicon Can be Artificially Produced by Combining Silica and Carbon in Electric Furnice
- Gives Polycrystalline Silicon (multitude of crystals)
- Practical Integrated Circuits Can Only be Fabricated from Single-Crystal Material

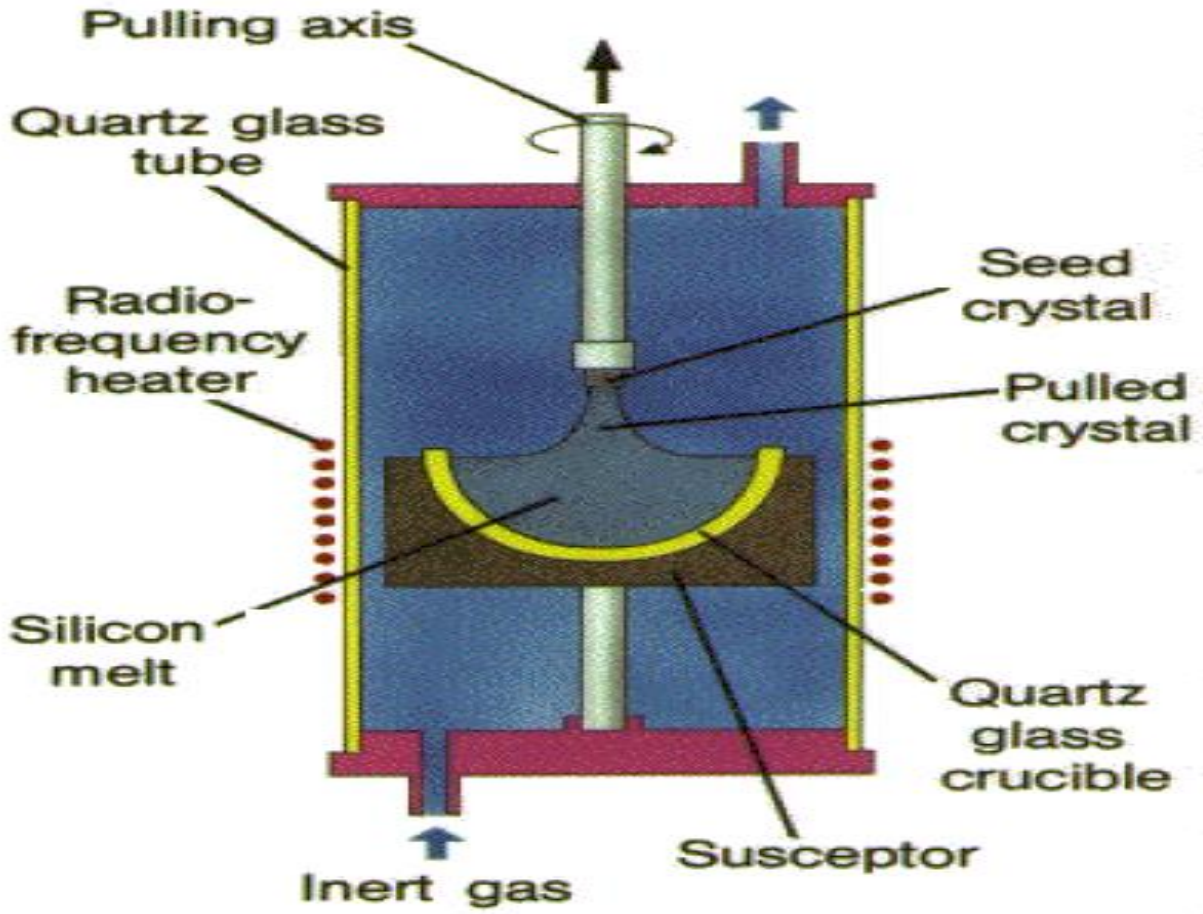
CRYSTAL GROWTH

- Czochralski Process is a Technique in Making Single-Crystal Silicon
- A Solid Seed Crystal is Rotated and Slowly Extracted from a Pool of Molten Si
- Requires Careful Control to Give Crystals Desired Purity and Dimensions





CRYSTAL GROWTH

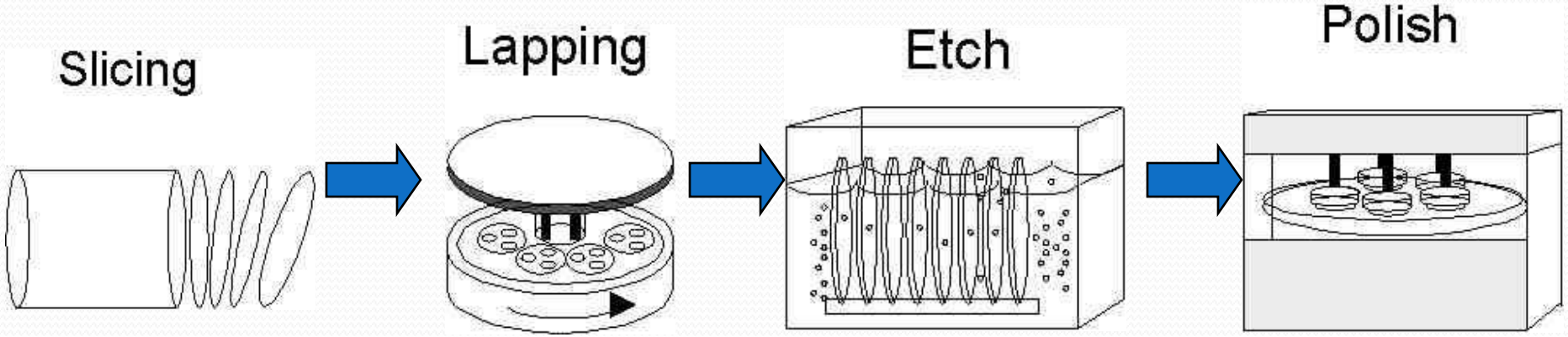


CYLINDER OF MONOCRYSTALLINE

- The Silicon Cylinder is Known as an Ingot
- Typical Ingot is About 1 or 2 Meters in Length
- Can be Sliced into Hundreds of Smaller Circular Pieces Called Wafers
- Each Wafer Yields Hundreds or Thousands of Integrated Circuits

WAFER MANUFACTURING

- The Silicon Crystal is Sliced by Using a Diamond-Tipped Saw into Thin Wafers
- Sorted by Thickness
- Damaged Wafers Removed During Lapping
- Etch Wafers in Chemical to Remove any Remaining Crystal Damage
- Polishing Smoothes Uneven Surface Left by Sawing Process



Silicon Manufacturing

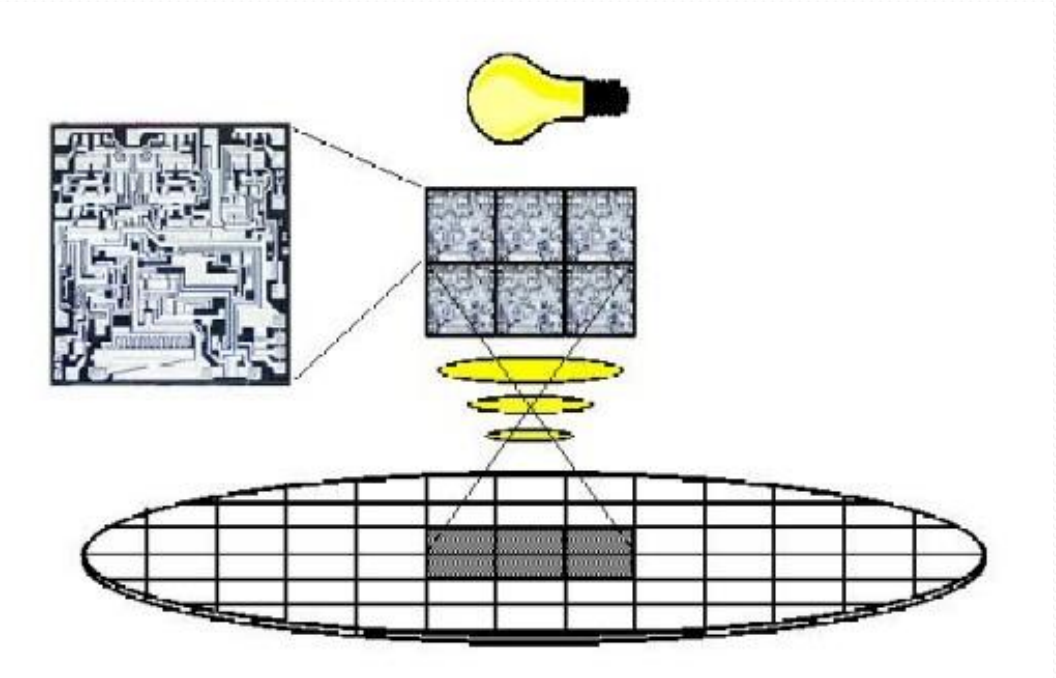
Photolithography

Photolithography

Deep UV Photolithography

Photolithography

Photolithography is a technique that is used to define the shape of micro-machined structures on a wafer.



Photolithography

Photoresist

The first step in the photolithography process is to develop a mask, which will be typically be a chromium pattern on a glass plate.

Next, the wafer is then coated with a polymer which is sensitive to ultraviolet light called a photoresist.

Afterward, the photoresist is then developed which transfers the pattern on the mask to the photoresist layer.

Photolithography
Photoresist

There are two basic types of Photoresists Positive and Negative.

Positive resists.

Positive resists decomposes ultraviolet light. The resist is exposed with UV light wherever the underlying material is to be removed. In these resists, exposure to the UV light changes the chemical structure of the resist so that it becomes more soluble in the developer. The exposed resist is then washed away by the developer solution, leaving windows of the bare underlying material. The mask, therefore, contains an exact copy of the pattern which is to remain on the wafer.

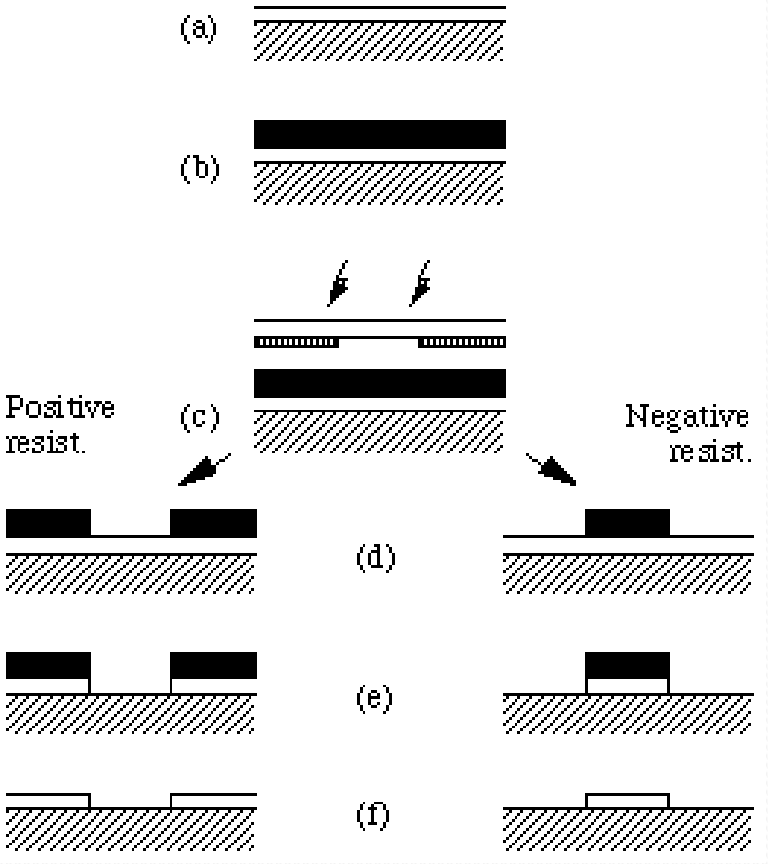
Photolithography Photoresist

Negative resists

Exposure to the UV light causes the negative resist to become polymerized, and more difficult to dissolve. Therefore, the negative resist remains on the surface wherever it is exposed, and the developer solution removes only the unexposed portions. Masks used for negative photoresists, therefore, contain the inverse (or photographic "negative") of the pattern to be transferred.

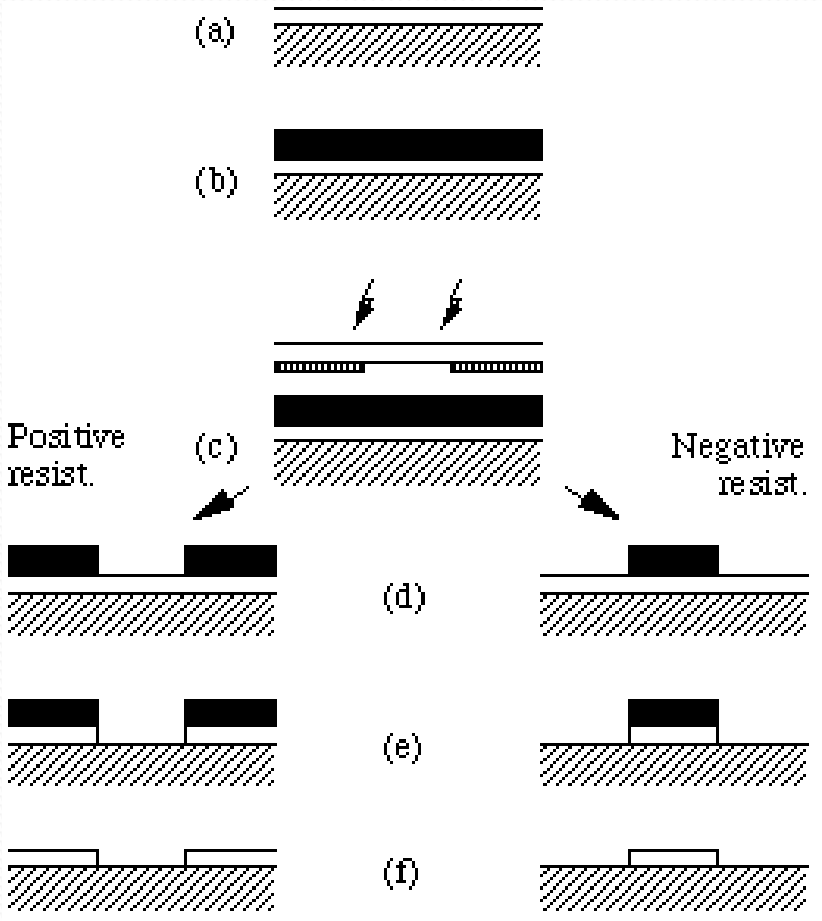
Photolithography Model

- Figure 1a shows a thin film of some material (eg, silicon dioxide) on a substrate of some other material (eg, a silicon wafer).
- Photoresist layer (Figure 1b)
- Ultraviolet light is then shone through the mask onto the photoresist (figure 1c).

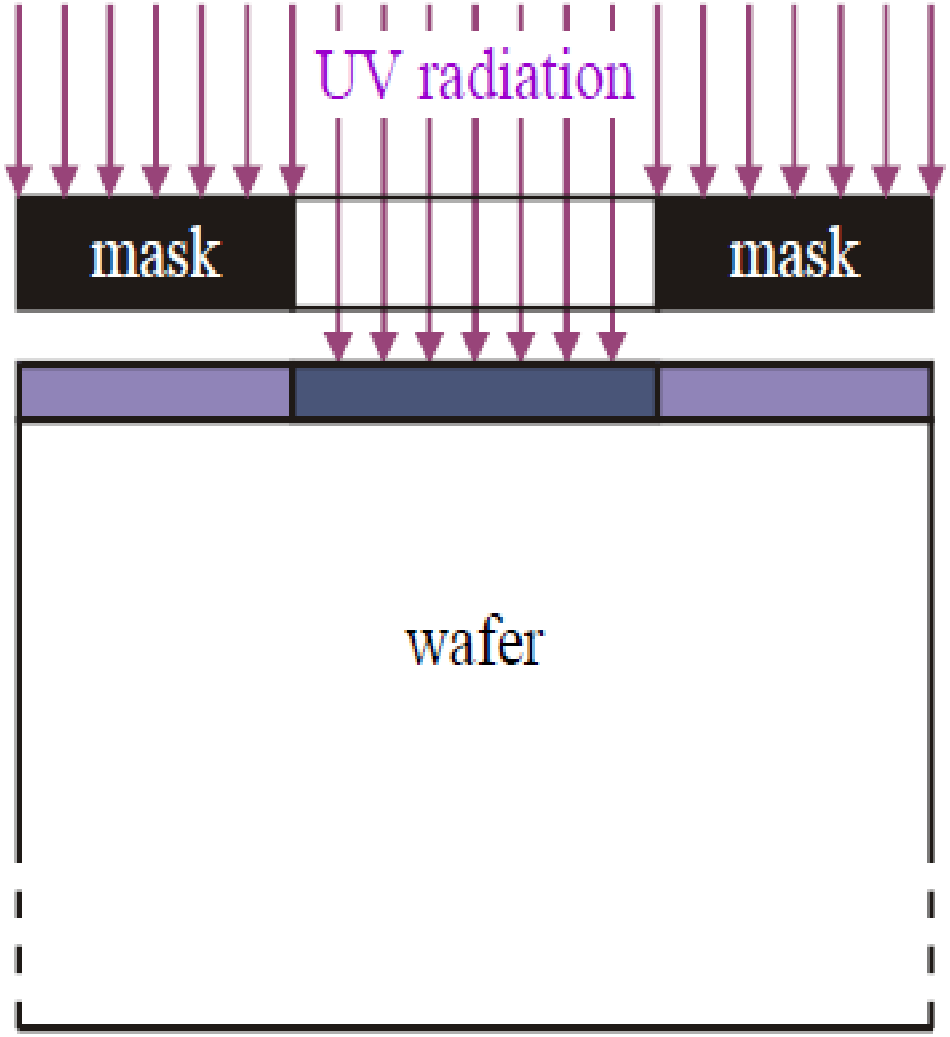


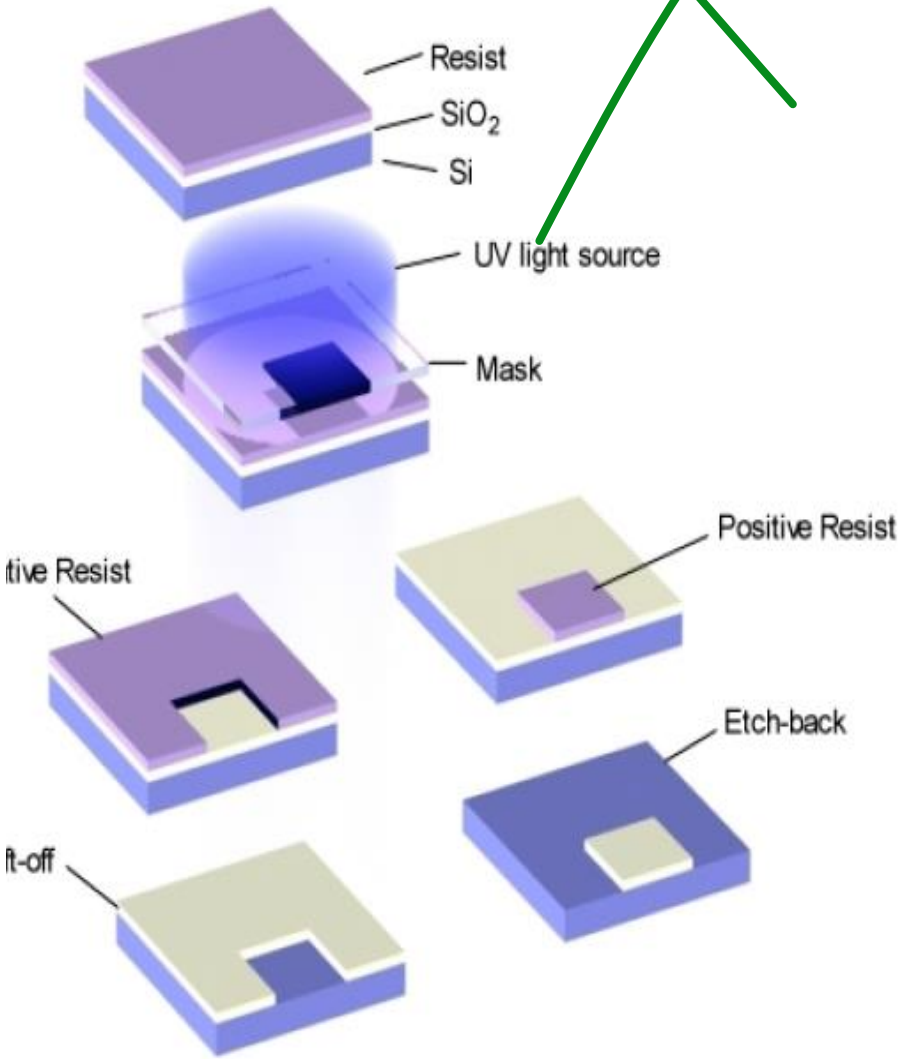
Photolithography Model

- The photoresist is then developed which transfers the pattern on the mask to the photoresist layer (figure 1d).
- A chemical (or some other method) is then used to remove the oxide where it is exposed through the openings in the resist (figure 1e).
- Finally the resist is removed leaving the patterned oxide (figure 1f).



Positive and negative resist

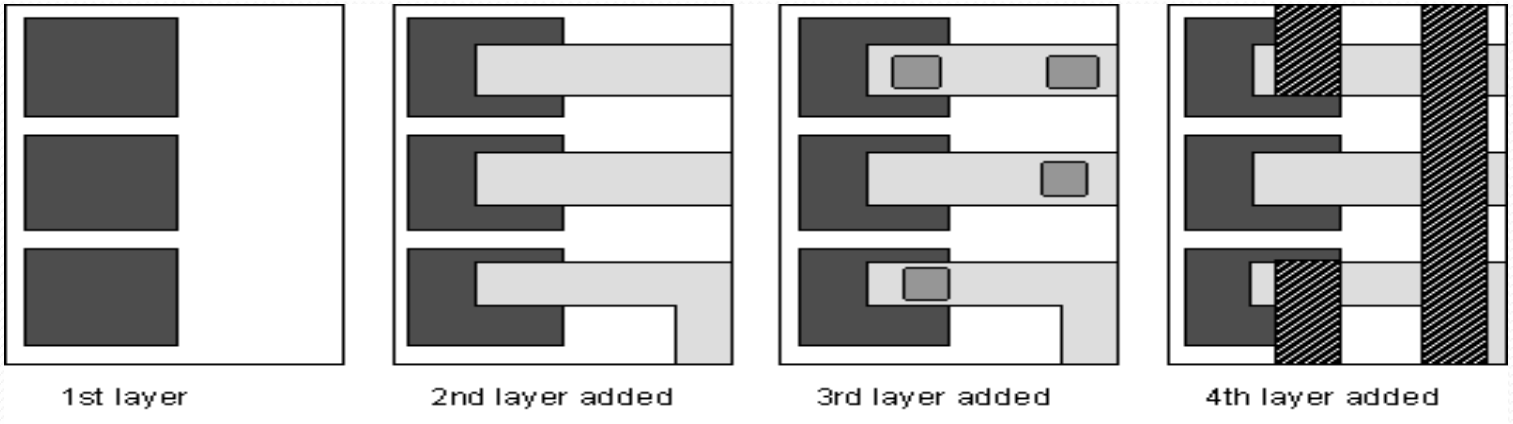




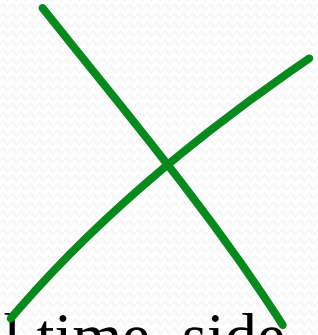
Photolithography
Photomasks and Reticles

Photomask

This is a square glass plate with a patterned emulsion of metal film on one side. The mask is aligned with the wafer, so that the pattern can be transferred onto the wafer surface. Each mask after the first one must be aligned to the previous pattern.



Photolithography
Photomasks and Reticles



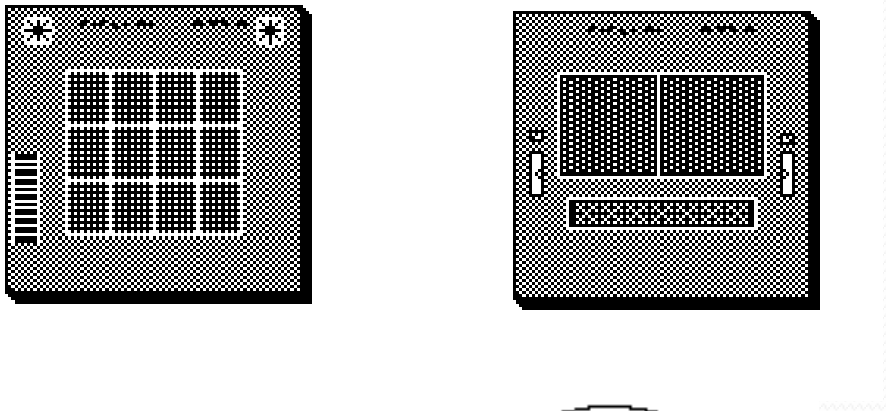
When a image on the photomask is projected several time side by side onto the wafer, this is known as **stepping** and the photomask is called a **reticle**.

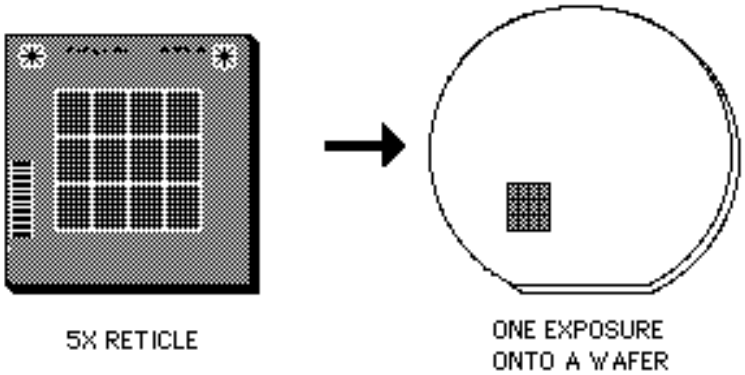
An common reticle is the 5X

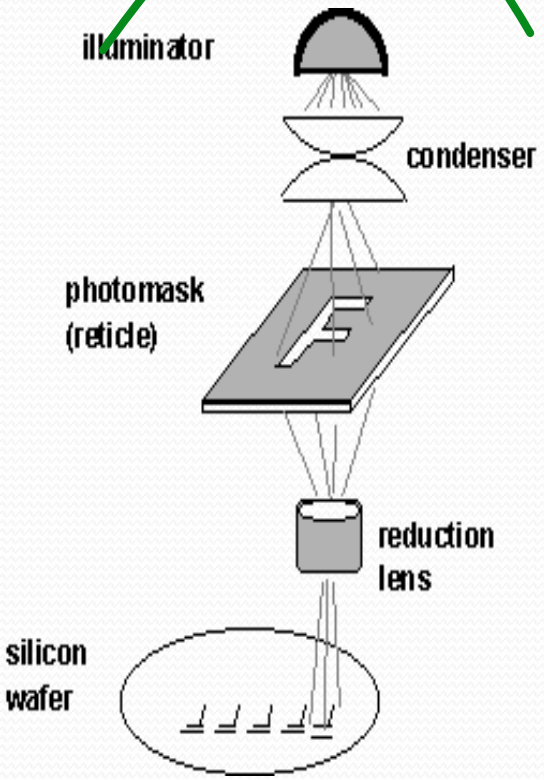
The patterns on the 5X reticle are reduced 5 times when projected onto the wafer. This means the dies on the photomask are 5 times larger than they are on the final product. There are other kinds of reduction reticles (2X, 4X, and 10X), but the 5X is the most commonly used. Reduction reticles are used on a variety of steppers, the most common being ASM, Canon, Nikon, and GCA.

Photolithography
Photomasks and Reticles

Examples of 5X Reticles:

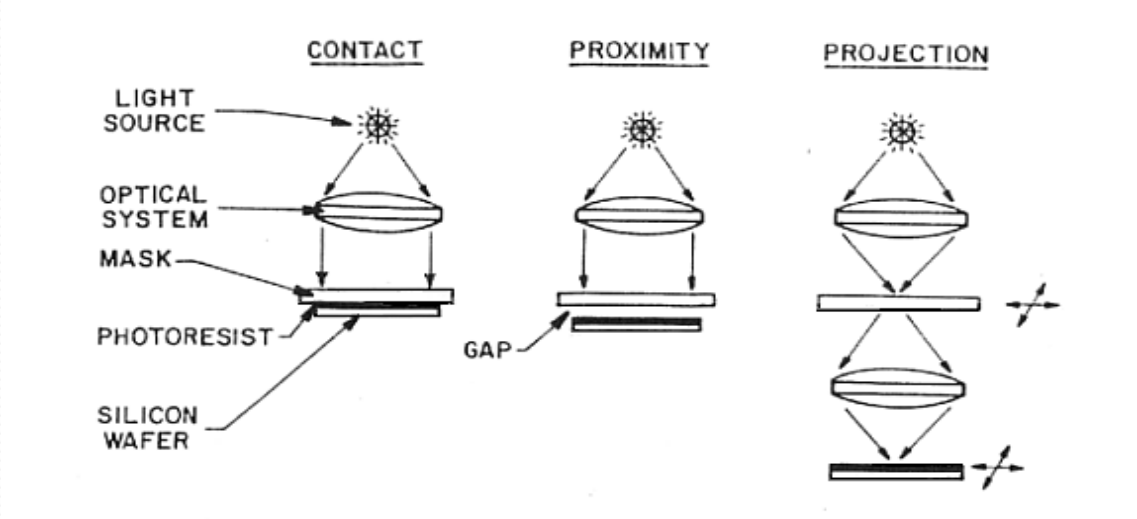




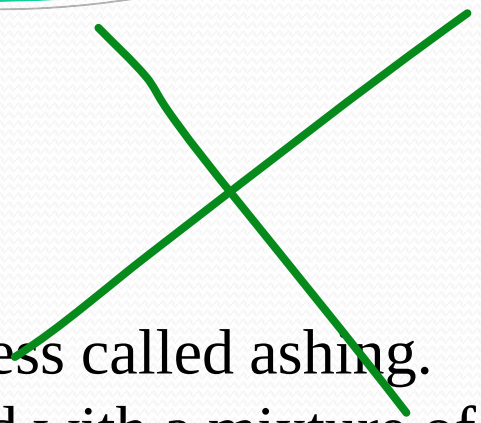


Photolithography
Photomasks and Reticles

Once the mask has been accurately aligned with the pattern on the wafer's surface, the photoresist is exposed through the pattern on the mask with a high intensity ultraviolet light. There are three primary exposure methods: contact, proximity, and projection.



Photolithography Patterning



The last stage of Photolithography is a process called ashing. This process has the exposed wafers sprayed with a mixture of organic solvents that dissolves portions of the photoresist .

Conventional methods of ashing require an oxygen-plasma ash, often in combination with halogen gases, to penetrate the crust and remove the photoresist. Usually, the plasma ashing process also requires a follow-up cleaning with wet-chemicals and acids to remove the residues and non-volatile contaminants that remain after ashing. Despite this treatment, it is not unusual to repeat the "ash plus wet-clean" cycle in order to completely remove all photoresist and residues.

Silicon Manufacturing

Oxidation of Silicon

The simplest method of producing an oxide layer consists of heating a silicon wafer in an oxidizing atmosphere.

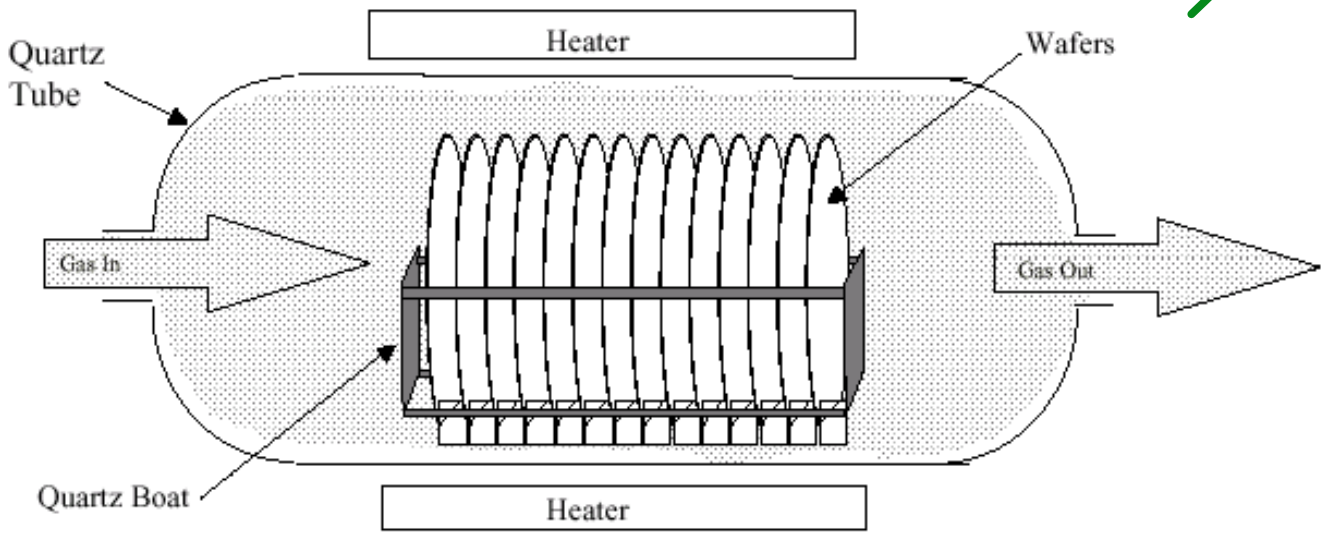
Oxidation of Silicon

- **SiO_2 growth is a key process step in manufacturing all Si devices ,Thick (1µm) oxides are used for field oxides (isolate devices from one another)**
  - **Dense and hard SiO_2 layer act as contamination barrier Hardness of the SiO_2 layer protect the surface from scratches during fabrication process**
  - **Sacrificial layers are grown and removed to clean up surfaces**
- **The stability and ease of formation of SiO_2 was one of the reasons that Si replaced Ge as the semiconductor of choice.**

Oxidation of Silicon

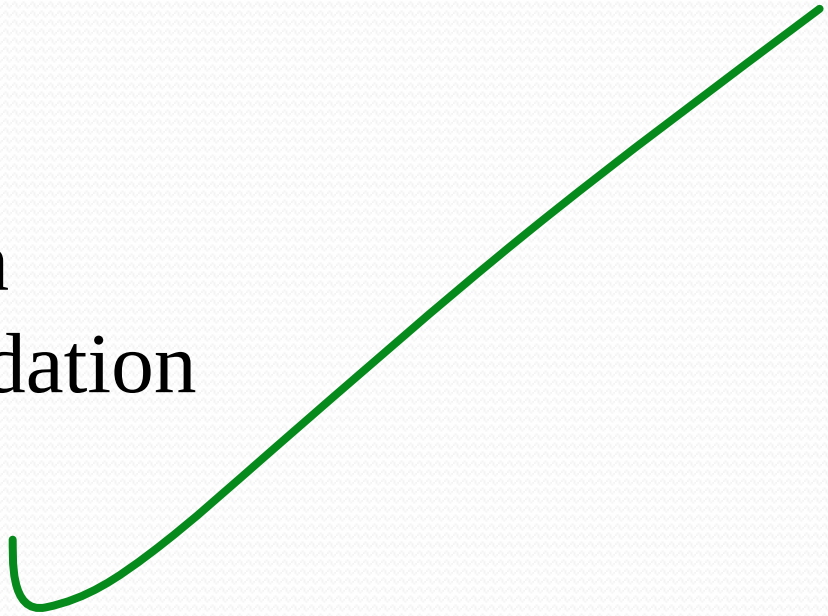
- Heat wafers in an atmosphere containing an oxidant, usually O_2, steam, or N_2O.

$$Si + O_2 \rightarrow SiO_2$$
$$Si + 2H_2O \rightarrow SiO_2 + 2H_2$$
$$Si + 2N_2O \rightarrow SiO_2 + 2N_2$$



Type of oxidation

1. Dry oxidation
2. Wet oxidation
3. Thermal oxidation
4. High pressure oxidation



- Oxidation temperature 900-1200°C
- Oxidation: Si wafer → placed in a heated chamber → exposed to oxygen gas

Oxidation of Silicon

- **Dry oxide** - Pure dry oxygen is employed

Disadvantage

- Dry oxide grows very slowly.

Advantage

- Oxide layers are very uniform.
- Relatively few defects exist at the oxide-silicon interface (These defects interfere with the proper operation of semiconductor devices)
- It has especially low surface state charges and thus make ideal dielectrics for MOS transistors.

$Si\ (solid) + O_2\ (gas) \rightarrow SiO_2\ (solid)$ (dry oxidation)

Oxidation of Silicon

- **Wet oxide -** In the same way as dry oxides, but steam is injected

  Disadvantage

- Hydrogen atoms liberated by the decomposition of the water molecules produce imperfections that may degrade the oxide quality.

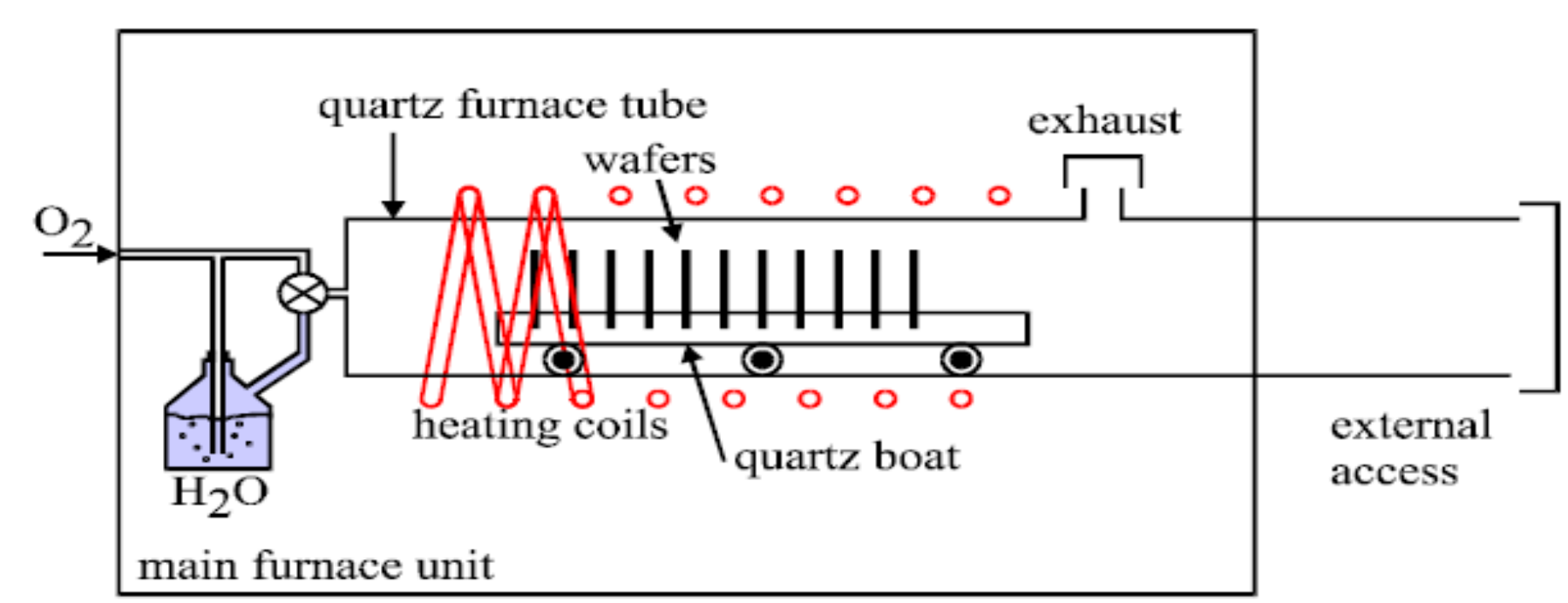
  Advantage

1. Wet oxide grows fast.
2. Useful to grow a thick layer of field oxide

Si (solid) + $2H_2O$ (gas) → SiO_2 (solid) + $2H_2$(gas) (wet oxidation)

Thermal oxidation

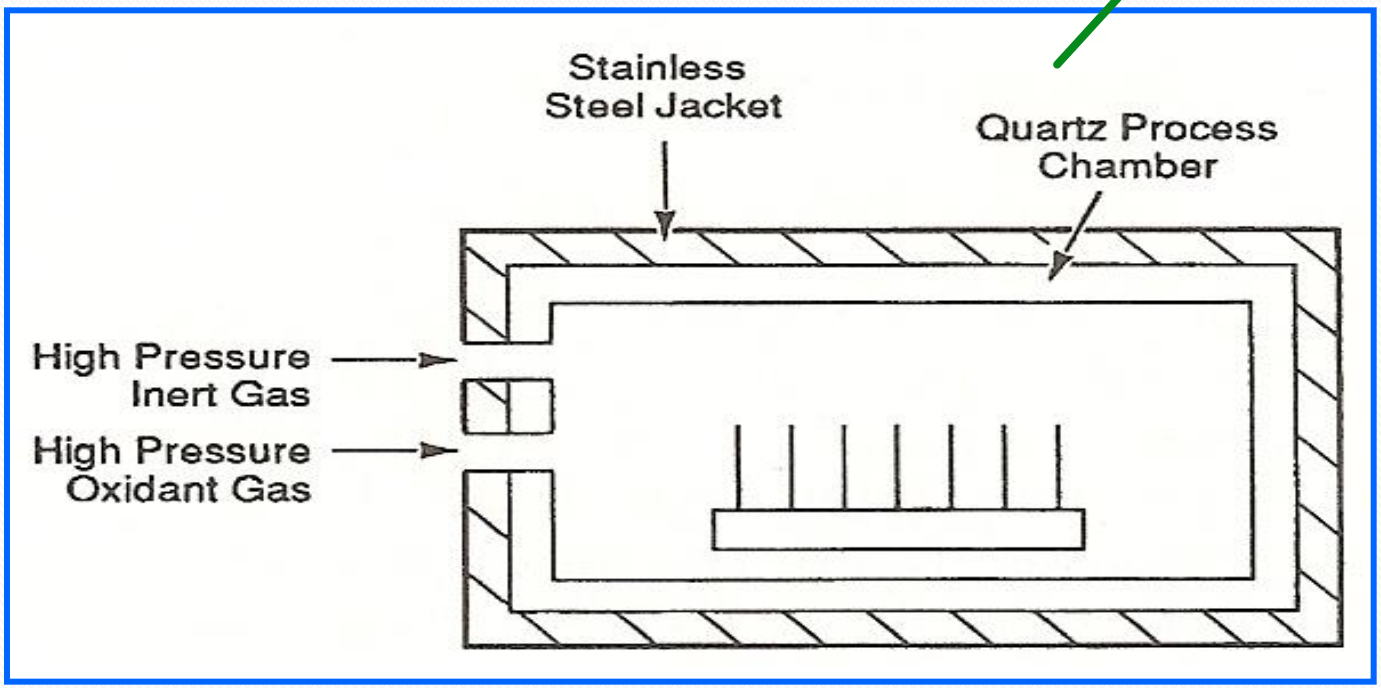
- the growth of a layer of silicon dioxide (SiO_2) on the substrate surface
- Requires only substrate heating to 900-1200 °C in a dry (O_2) or wet (H_2O steam) ambient using an oxidation furnace
- Silicon oxidizes quite readily one reason why Si is so widely used

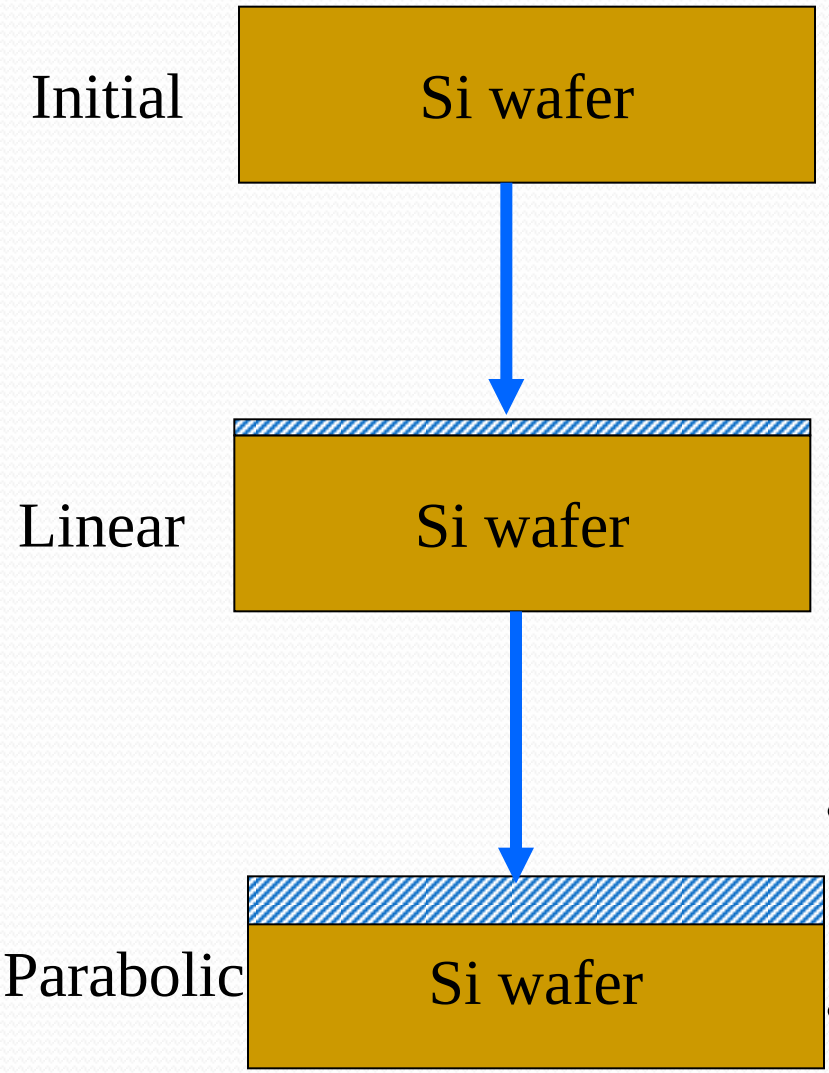


High Pressure Oxidation

- High pressure oxidation results in faster oxidation rate
- Advantage of high pressure oxidation
  - Drop the oxidation temperature
  - Reduce oxidation time

Thin oxide produced using high pressure oxidation → higher dielectric strength than oxides grown at atmospheric pressure





- In a furnace with O_2 gas environment

- Oxygen atoms combine readily with Si atoms
- Linear- oxide grows in equal amounts for each time
- Around 500Å thick

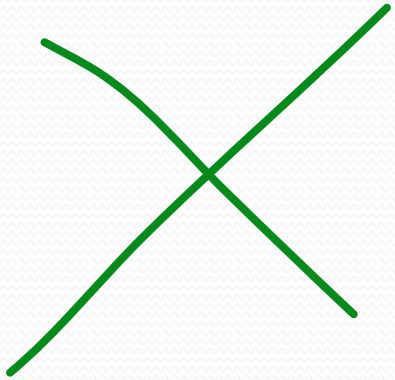
- Above 500Å, in order for oxide layer to keep growing, oxygen and Si atoms must be in contact
- SiO_2 layer separate the oxygen in the chamber from the wafer surface

Deposited Oxides

- Oxide is frequently employed as an insulator between two layers of metalization. In such cases, some form of **deposited oxide** must be used rather than the grown oxides.
- Deposited oxides can be produced by various reactions between gaseous silicon compounds and gaseous oxidizers. Deposited oxides tend to possess low densities and large numbers of defect sites. Not suitable for use as gate dielectrics for MOS transistors but still acceptable for use as insulating layers between multiple conductor layers, or as protective overcoats.

Key Variables in Oxidation

- **Temperature**
  - **- reaction rate**
  - **- solid state diffusion**
- **Oxidizing species**
  - **- wet oxidation is much faster than dry oxidation**
- **Surface cleanliness**
  - **- metallic contamination can catalyze reaction**
  - **- quality of oxide grown (interface states)**



Etching

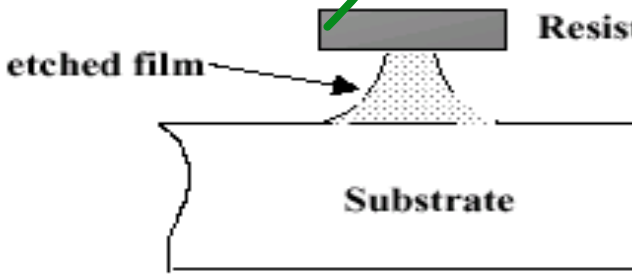
Etching is the process where unwanted areas of films are removed by either dissolving them in a wet chemical solution (**Wet Etching**) or by reacting them with gases in a plasma to form volatile products (**Dry Etching**).

Resist protects areas which are to remain. In some cases a hard mask, usually patterned layers of SiO_2 or Si_3N_4, are used when the etch selectivity to photoresist is low or the etching environment causes resist to delaminate.

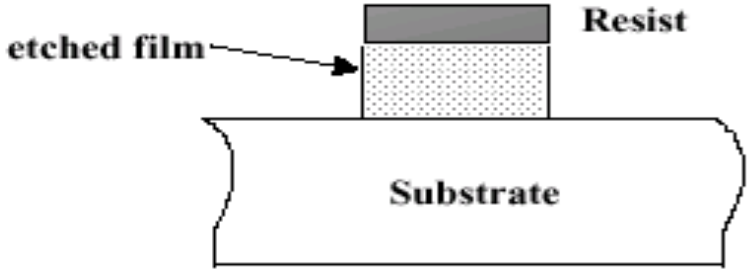
This is part of lithography - pattern transfer.

Terminology

- **Isotropic etch**
  - a process that etches at the same rate in all directions.



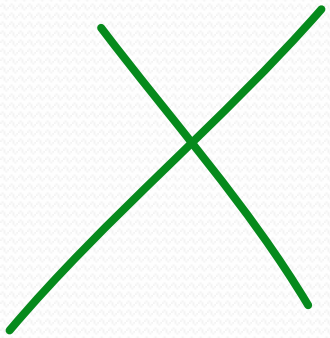
- **Anisotropic etch**
  - a process that etches only in one direction.



Wet Chemical Etching

- **Wet etches:**

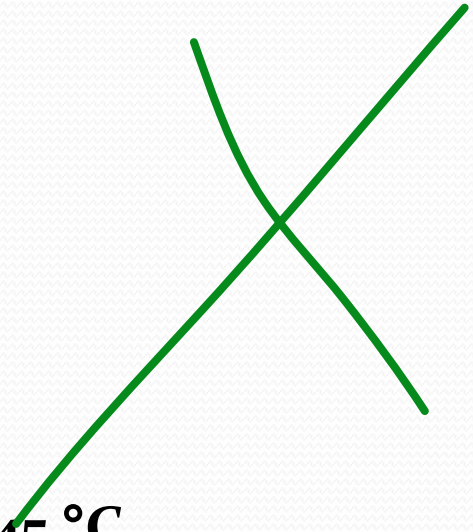
  - are in general isotropic
  (not used to etch features less than ≈ 3 μm)
  - achieve high selectivities for most film combinations
  - capable of high throughputs
  - use comparably cheap equipment
  - can have resist adhesion problems
  - can etch just about anything



-Use acid or basic solutions. For instance, hydrofluoric acid buffered with ammonium fluoride is typically used to etch SiO_2

Example Wet Processes

- For SiO_2 etching
  - $HF + NH_4F + H_2O$ (buffered oxide etch or BOE)
- For Si_3N_4
  - Hot phosphoric acid: H_3PO_4 at 180 °C
  - need to use oxide hard mask
- Silicon
  - Nitric, HF, acetic acids
  - $HNO_3 + HF + CH_3COOH + H_2O$
- Aluminum
  - Acetic, nitric, phosphoric acids at 35-45 °C
  - $CH_3COOH + HNO_3 + H_3PO_4$



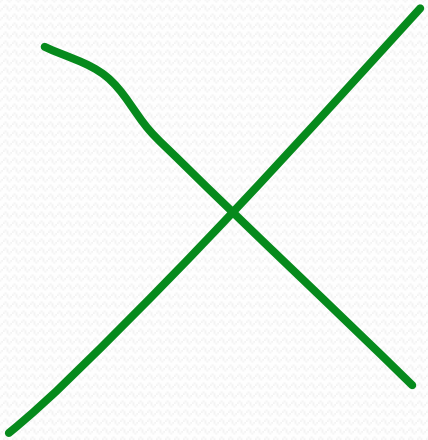
What is a plasma (glow discharge)?

- A plasma is a partially ionized gas made up of equal parts positively and negatively charged particles.
- Plasmas are generated by flowing gases through an electric or magnetic field.
- These fields remove electrons from some of the gas molecules. The liberated electrons are accelerated, or energized, by the fields.
- The energetic electrons slam into other gas molecules, liberating more electrons, which are accelerated and liberate more electrons from gas molecules, thus sustaining the plasma.

Dry or Plasma Etching

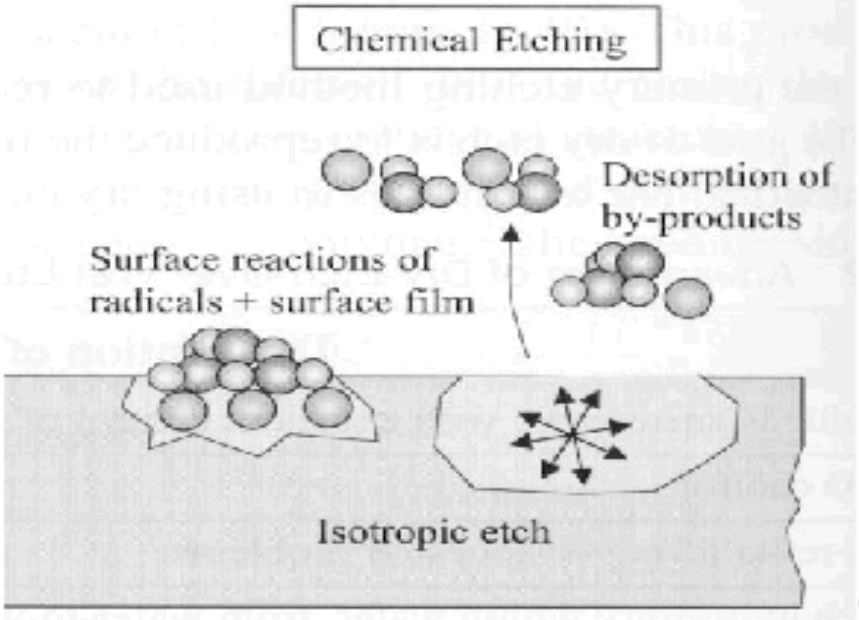
1. Purely physical (sputtering)

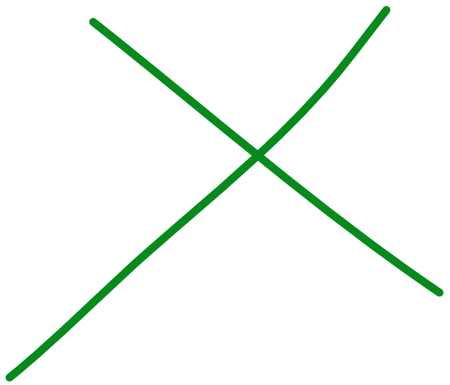
- Can be anisotropic
- All materials have sputter yields within a factor of about 3. therefore selectivities will be low
- nonvolatile species can redeposit on surfaces
- e. Ion Milling process
- In dry etching, ions of a neutral material are accelerated toward the surface and cause ejection of atoms of all materials



Dry or Plasma Etching

- **Purely chemical**
  - **isotropic**
  - **can have high selectivities**
  - **similar to wet etching**
  - **ex. High Pressure Plasma process**





Semiconductor Manufacturing Technology, Quirk and Serda, Prentice Hall,2001

Dry or Plasma Etching

Combination of chemical and physical etching – Reactive Ion Etching (RIE)

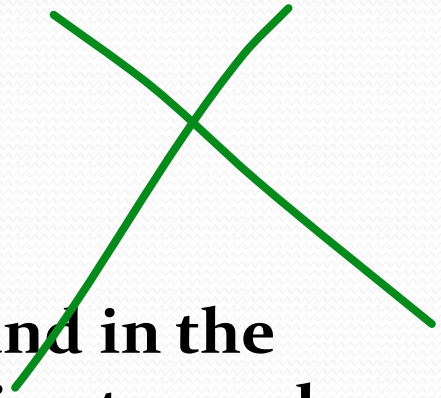
Directional etching due to ion assistance.

In RIE processes the wafers sit on the powered electrode. This placement sets up a negative bias on the wafer which accelerates positively charge ions toward the surface. These ions enhance the chemical etching mechanisms and allow **anisotropic** etching.

- **Wet etches are simpler, but dry etches provide better line width control since it is anisotropic.**

- Plasma etching has the advantage of offering a well-defined directionality to the etching action, creating patterns with sharp vertical contours.

Other Effects of Oxide Growth and Removal



- Oxide Step
  - The differences in oxide thickness and in the depths of the silicon surfaces combine to produce a characteristic surface discontinuity
- The growth of a thermal oxide affects the doping levels in the underlying silicon
- The doping of silicon affects the rate of oxide growth

Local Oxidation of Silicon (LOCOS)

- **LOCOS: localized oxidation of silicon using silicon nitride as a mask against thermal oxidation.**
- **A technique called local oxidation of silicon (LOCOS) allows the selective growth thick oxide layers**
- **CMOS and BiCMOS processes employ LOCOS to grow a thick field oxide over electrically inactive regions of the wafer**

•The presence of another material such as silicon nitride (Si_3N_4) on the surface inhibits the growth of oxide in that region

Silicon Manufacturing

Diffusion , Ion Implantation And Epitaxial processes

Diffusion

•Most of these diffusion processes occur in two steps: the predeposition and the drive-in diffusion,

•In the pre deposition step. a high concentration of dopant atoms are introduced at the silicon surface by a vapor that contains the dopant at a temperature of about 1000°C. In recent years Ion Implantation is used.

•At the temperature of l000°C.silicon atoms move out of their lattice sites creating a high density of vacancies and breaking the bond with the neighboring atoms.

•The second step is drive-in process. used to drive the impurities deeper into the surface without adding anymore impurities.

Diffusion

•Common dopants are boron for P-type layers and phosphorus. antimony and arsenic for N-type layers.

•A typical arrangement of the process of diffusion is shown in Figure.

•The wafers are placed in a quartz furnace tube that is heated by resistance heaters surrounding it. So that the wafers may be inserted and removed easily from the furnace. they are placed in a slotted quartz carrier known as a boat. To introduce a phosphorus dopant. as aim example. phosphorus oxychloride

Diffusion

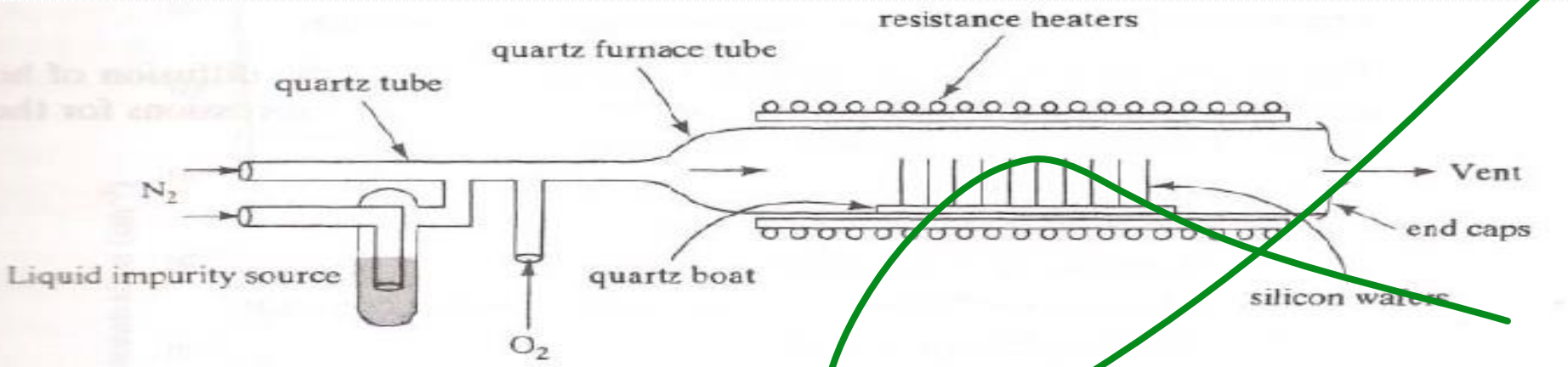
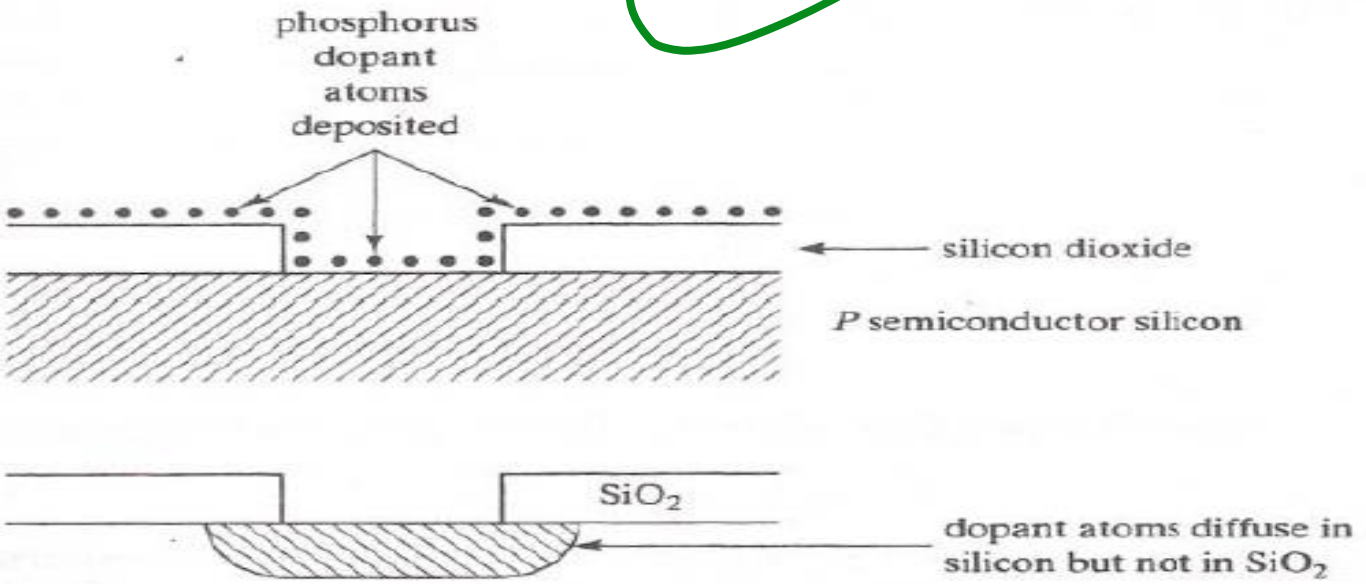


Figure 6.5 Physical layout of equipment used in diffusion.



Diffusion

- (POCI3) is placed in a container either inside the quartz tube, in a region of relatively low temperature. or in a container outside the furnace at a temperature that helps maintain its liquid form
- Nitrogen and oxygen gas are made to pass over the container. These gases carry the dopant vapor into the furnace, where the gases are deposited on the surface of the wafers. These gases react with the silicon, forming a layer on the surface of the wafer that contains silicon,oxygen, and phosphorus. At the high temperature of the furnace. phosphorus diffuses easily into the silicon.

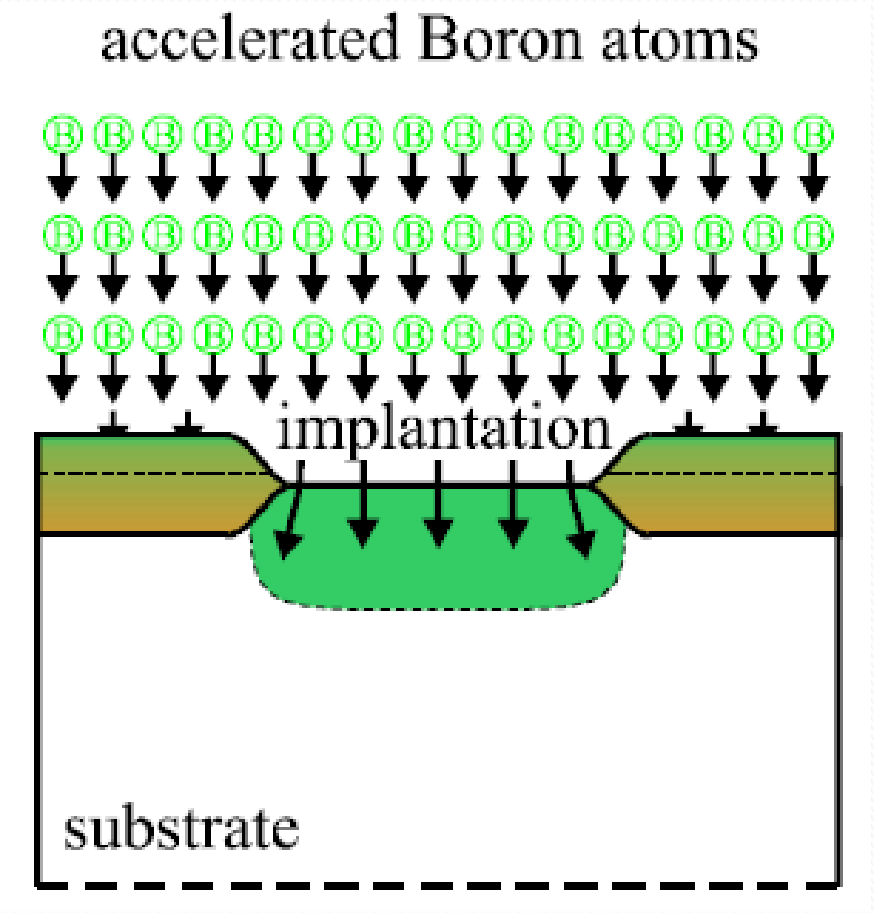
- Diffusion depth is controlled by the time and temperature of the drive-in process.
- By precise control of the time and temperature (to within 0.25°C).accurate junction depths of fraction of a micron can be obtained.

Ion Implantation

To generate ions. such as those of phosphorus. an arc discharge is made to occur in a gas. such as phosphine (PH3).that contains the dopant. The ions are then accelerated in an electric field so that they acquire an energy of about 20keV and are passed through a strong magnetic field. Because during the arc discharge unwanted impurities may have been generated. the magnetic field acts to separate these impurities from the dopant ions based on the fact that the amount of deflection of a particle in a magnetic field depends on its mass.

Following the action of the magnetic field, the ions are further accelerated so that their energy reaches several hundred keV, whereupon they are focused on and strike the surface of the silicon wafer.

- In ion implantation, dopantatoms are accelerated toward the substrate surface and enter due to their kinetic energy
- This is the preferred technique for introduction of dopantatoms since the amount of lateral diffusion is much lower



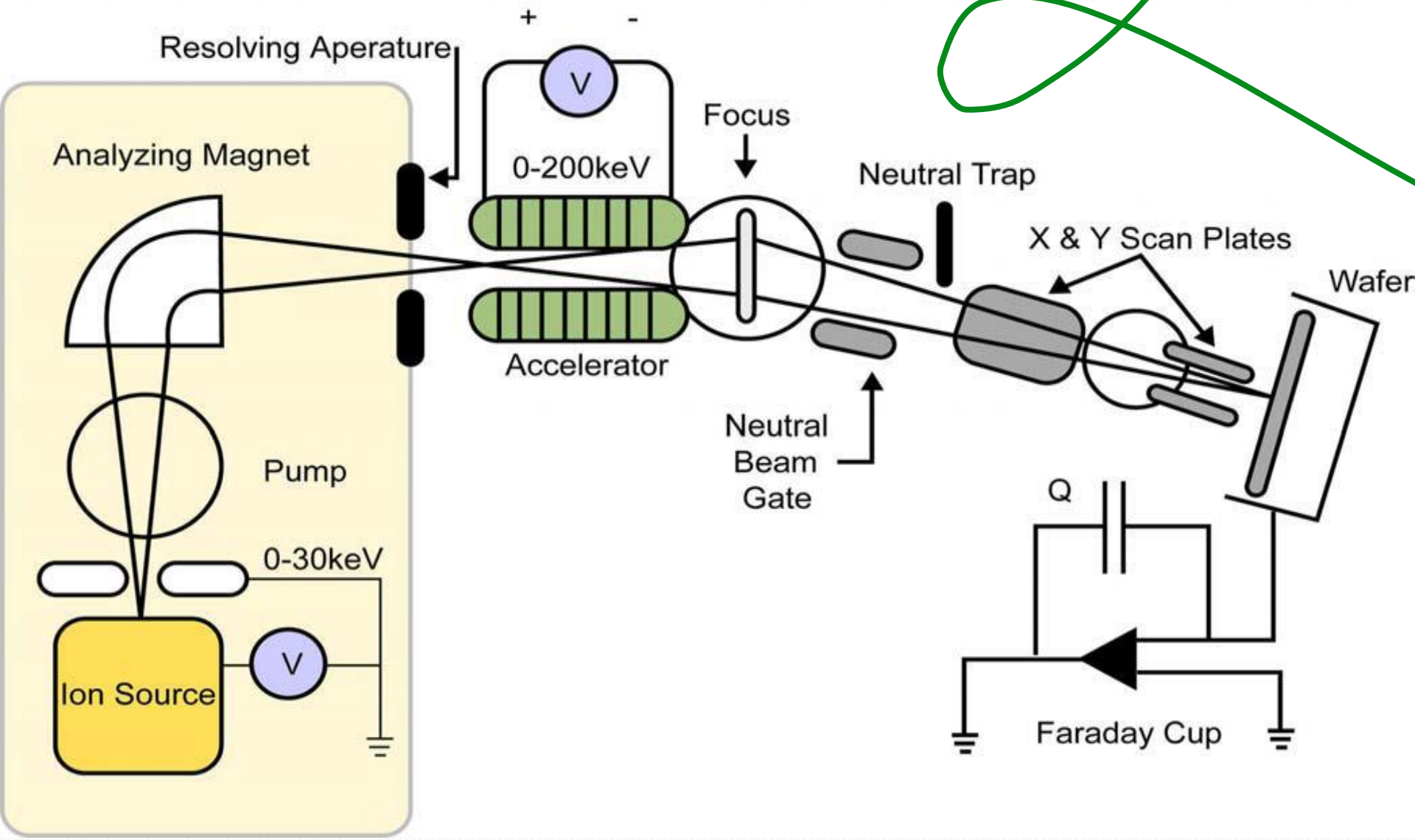
showing the ion beam hitting the 300mm wafer end-station.

Ion Implantation Equipment

Ions generated in a source (from feed gas, e.g. BF_3, AsH_3, PH_3 ... or

heated solid source, then ionized in arc chamber by electrons from hot filament) select desired species by q/m, using a magnet,

accelerated by an E-field and focused using electrostatic lenses and impact substrate (a bend removes neutrals) in raster pattern.


+
-
Resolving Aperature
V
Analyzing Magnet
0-200keV
Focus
Neutral Trap
X & Y Scan Plates
Wafer
Accelerator
Neutral
Beam
Gate
Pump
Q
0-30keV
Ion Source
V
Faraday Cup

Comparison of Diffusion and Ion Implantation

- **Diffusion is a cheaper and more simplistic method, but can only be performed from the surface of the wafers. Dopants also diffuse unevenly, and interact with each other altering the diffusion rate.**

- **Ion implantation is more expensive and complex. It does not require high temperatures and also allows for greater control of dopant concentration and profile. It is an anisotropic process and therefore does not spread the dopant implant as much as diffusion.**

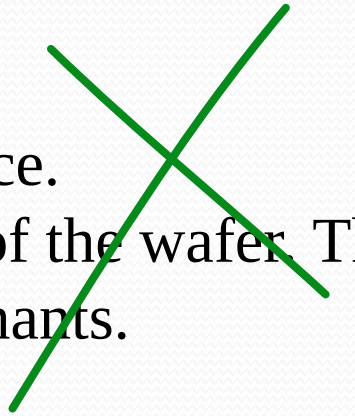
Epitaxial process

Epitaxial Growth

Epitaxial is used to deposit N on N+ silicon, which is impossible it. accomplish by diffusion. It is also used in isolation between bipolar transistors wherein N- is deposited on P.

We list below, and with reference to Figure. the sequence of operation involved in the process:

1. Heat wafer to 1200°C.
2. Turn on H2, to reduce the SiO2, on the wafer surface.
3. Turn on anhydrous HCL to vapor-etch the surface of the wafer. This removes a small amount of silicon and other contaminants.
4. Turn off HCL
5. Drop temperature to 1100°C.
6. Turn on silicon tetrachloride (SiCl4)
7. Introduce dopant.



Epitaxial process