國立清華大學
電機工程學系
一〇二學年度第二學期

EE-5265
『積體電路設計自動化』講義

Feb. – June, 2014

清華大學 EE 5265
積體電路設計自動化

單元 1
Overview of Design Automation

教育部顧問室
「超大型積體電路與系統設計」教育改進計畫
EDA聯盟 – 推廣課程

致謝

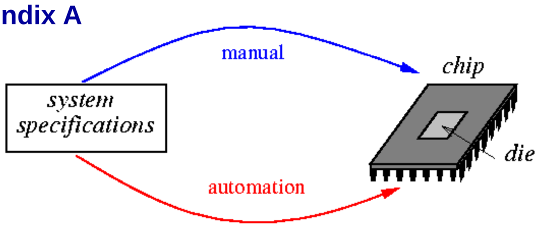
本單元之教材主要取自於
教育部
超大型積體電路與系統設計教育改進計畫
EDA聯盟之課程發展成果

(教材編纂小組成員)
台灣大學電機系 張耀文
清華大學電機系 黃錫瑜
交通大學資科系 李毅郎
中央大學電機系 劉建男
元智大學資工系 林榮彬

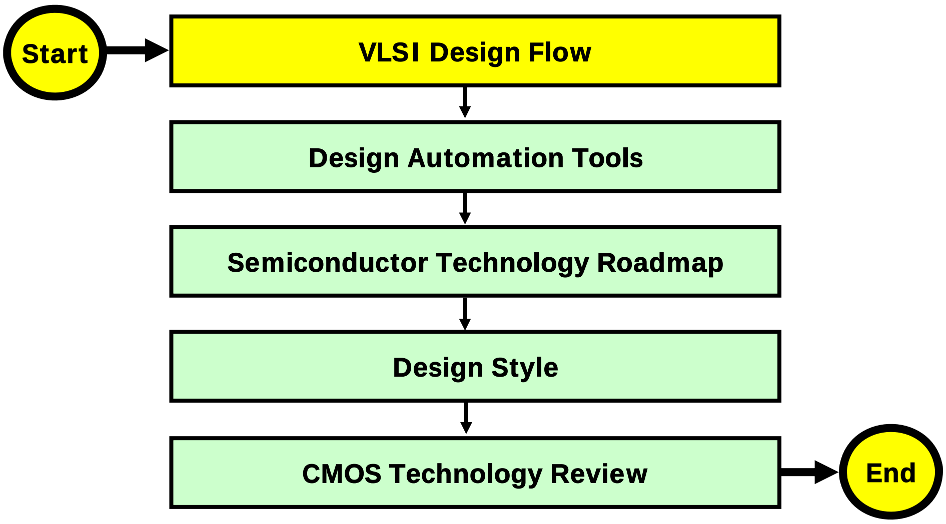
教育部顧問室
「超大型積體電路與系統設計」教育改進計畫
EDA聯盟 – 推廣課程

單元 1: Overview of Design Automation

- Course contents:
  - Introduction to VLSI design flow
  - Introduction to VLSI design automation tools
  - Semiconductor technology roadmap
  - Design styles
  - CMOS technology
- Readings
  - Chapters 1-2
  - Appendix A



Outline



Start

VLSI Design Flow

Design Automation Tools

Semiconductor Technology Roadmap

Design Style

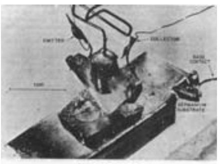
CMOS Technology Review

End

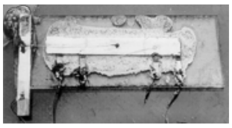
Milestones for IC Industry

- 1947: Bardeen, Brattain & Shockley invented the transistor, the foundation of the IC industry.
- 1952: SONY introduced the first transistor-based radio.
- 1958: Kilby invented integrated circuits (ICs).
- 1965: Moore's law.
- 1968: Noyce and Moore founded Intel.
- 1970: Intel introduced 1 K DRAM.

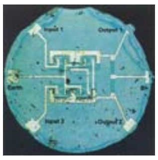
In 1956 John Bardeen, William Shockley and Walter Brattain shared the Nobel Prize in Physics for their discovery of the transistor.



First transistor



First IC by Kilby

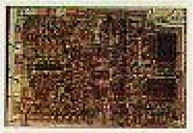


First IC by Noyce

Milestones for IC Industry

- 1971: Intel announced 4-bit 4004 microprocessors (2250 transistors).
- 1976/81: Apple II/IBM PC.
- 1985: Intel began focusing on microprocessor products.
- 1987: TSMC was founded (to support fabless IC design).
- 1991: ARM introduced its first embeddable RISC IP core (chipless IC design).

Intel founders



4004

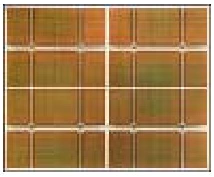
IBM PC

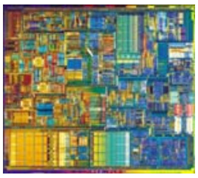
tsmc

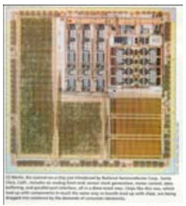
ARM
Advanced RISC Machines

Complexity Is Skyrocketing …

- 1996: Samsung introduced IG DRAM.
- 1998: IBM announced 1GHz microprocessor.
- 1999/earlier:
  - System-on-Chip (SOC) methodology gains popularity.
  - Intel P4 processor: 42 million transistors
- Productivity:
  - 30 million transistors per person today for ASIC chips
  - 1 billion/person by 2008

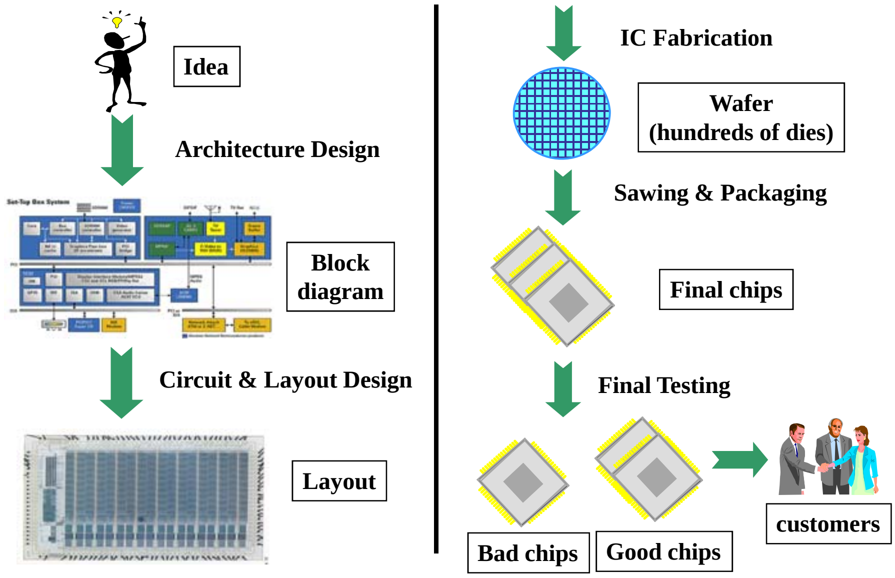

4GB DRAM (2001)


Pentium 4


Scanner-on-chip


Blue tooth technology

IC Design & Manufacturing Process



Idea

Architecture Design

Block diagram

Circuit & Layout Design

Layout

IC Fabrication

Wafer (hundreds of dies)

Sawing & Packaging

Final chips

Final Testing

Bad chips

Good chips

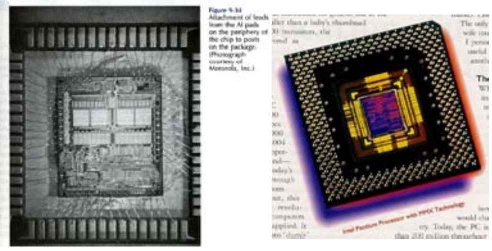
customers

From Wafer to Chip

Silicon Ingot

Lithography

Packaging



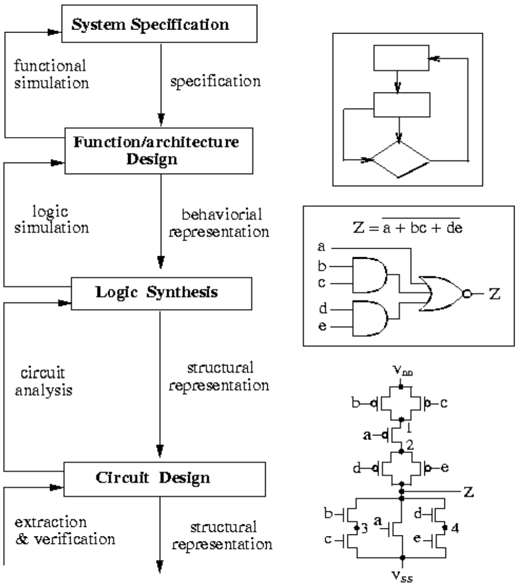
Final Chips

Traditional VLSI Design Cycle

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

- Other tasks involved: simulation, testing, etc.
- Design metrics: area, speed, power dissipation, noise, design time, testability, etc.

Traditional VLSI Design Cycle



System Specification

functional simulation

specification

Function/architecture Design

logic simulation

behaviorial representation

Logic Synthesis

circuit analysis

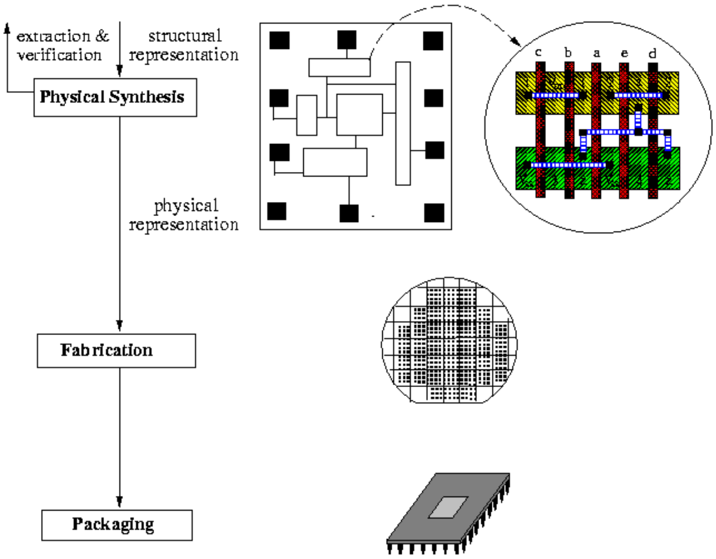
structural representation

Circuit Design

extraction & verification

structural representation

Traditional VLSI Design Flow (Cont'd)



extraction & verification

structural representation

Physical Synthesis

physical representation

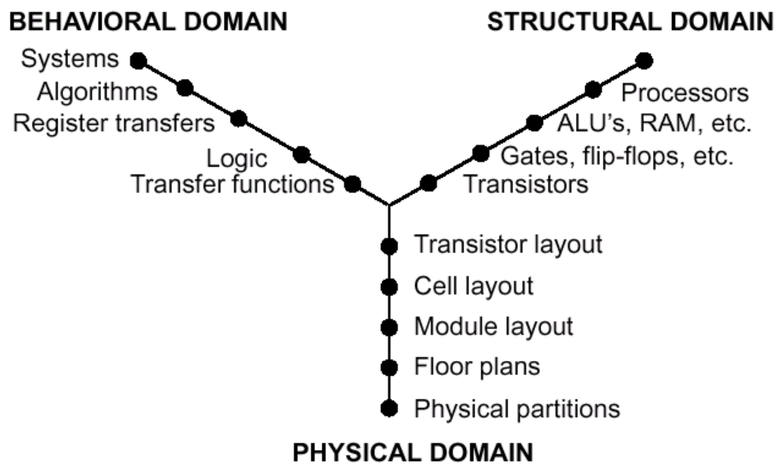
Fabrication

Packaging

Design Actions

- **Synthesis:**
  - increasing information about the design by providing more details (e.g., logic synthesis, physical synthesis)
- **Optimization:**
  - increasing the quality of the design by restructuring a given description (e.g., logic optimizer, timing optimizer).
- **Analysis:**
  - collecting information on the quality of the design (e.g., timing analysis, power analysis, etc).
- **Verification:**
  - checking whether an implementation conforms to the desired specification
  - Is what I get really what I want?
- **Design Management:**
  - storage of design data, cooperation between tools, design flow, etc. (e.g., database).

Gajski's Y-Chart



BEHAVIORAL DOMAIN

Systems
Algorithms
Register transfers
Logic
Transfer functions

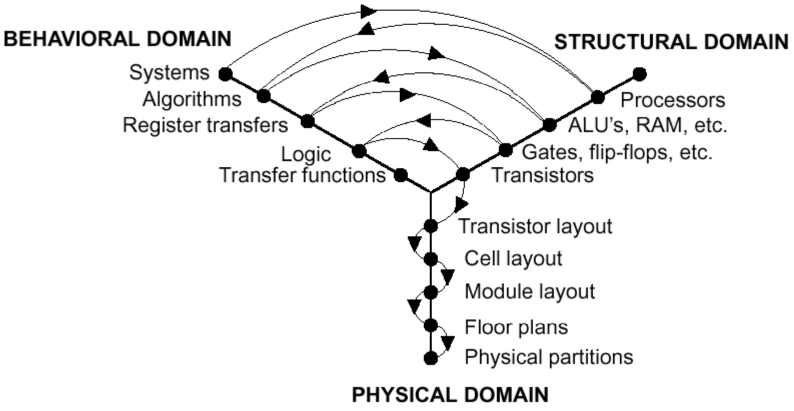
STRUCTURAL DOMAIN

Processors
ALU's, RAM, etc.
Gates, flip-flops, etc.
Transistors

Transistor layout
Cell layout
Module layout
Floor plans
Physical partitions

PHYSICAL DOMAIN

Top-Down Structural Design



Outline

Start

VLSI Design Flow

Design Automation Tools

Semiconductor Technology Roadmap

Design Style

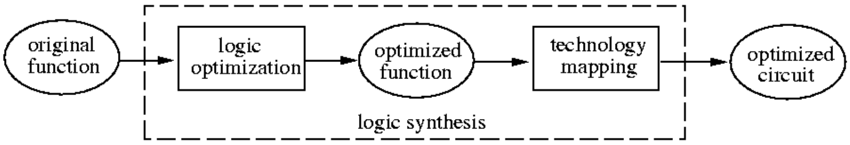
CMOS Technology Review

End

Design Issues and Tools

- System-level design (taking a C code as the input)
  - Hardware/software partitioning, co-verification
  - System-Verilog, System-C for co-simulation
  - Silicon compilation (from C to layout) ➔ rarely used…
- Architecture-level design
  - RTL simulation
  - RTL synthesis (From RTL code to Gate-Level circuit)
- Logic-level design
  - Logic optimization
  - Gate-level simulation (functionality, timing, power, etc)
  - Static timing analysis (STA), or statistical static timing analysis (SSTA)
  - Formal verification
- Transistor-Level Design
  - Schematic editor, circuit simulation (SPICE)
- Physical-level design
  - Floorplanning, Placement, Routing, Compaction
  - DRC for Design Rule Checking
  - LVS for Layout vs. Schematic Check
  - Parasitic RC extraction

Logic Synthesis



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

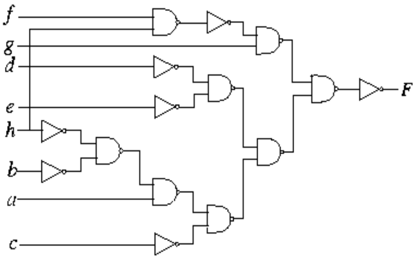
Logic Optimization Examples

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

$t1 = a + b\ \ c;$
$t2 = d + e;$
$t3 = a\ \ b + d;$
$t4 = t1\ \ t2 + f\,g;$
$t5 = t4\ \ h + t2\ \ t3;$
$F = t5';$

$\xrightarrow{\text{logic optimization}}$

$t1 = d + e;$
$t2 = b + h;$
$t3 = a\ \ t2 + c;$
$t4 = t1\ \ t3 + f\,g\,h;$



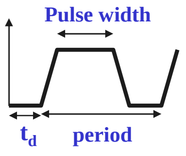
subject graph for the optimized equations

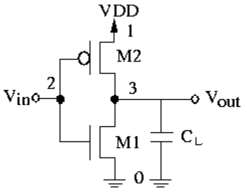
Circuit Simulation of a CMOS Inverter (0.6 μm)

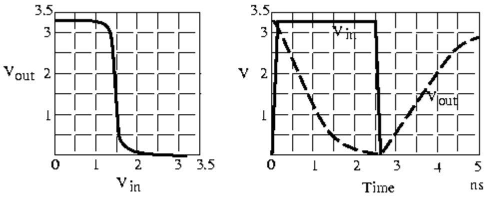
```
M1 3 2 0 0 nch W=1.2u L=0.6u AS=2.16p PS=4.8u AD=2.16p PD=4.8u
M2 3 2 1 1 pch W=1.8u L=0.6u AS=3.24p PS=5.4u AD=3.24p PD=5.4u
CL 3 0 0.2pF
```

v_L v_H t_d t_r t_f **Pulse Width** **Period**

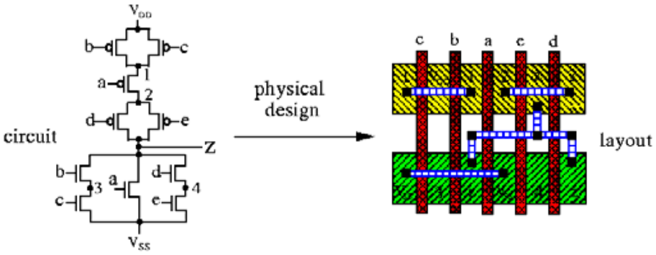
```
VDD 1 0 3.3
VIN 2 0 DC 0 PULSE (0 3.3 0ns 100ps 100ps 2.4ns 5ns)

.LIB '../mod_06' typical

.OPTION NOMOD POST INGOLD=2 NUMDGT=6 BRIEF
.DC VIN 0V 3.3V 0.001V
.PRINT DC V(3)
.TRAN 0.001N 5N
.PRINT TRAN V(2) V(3)
.END
```





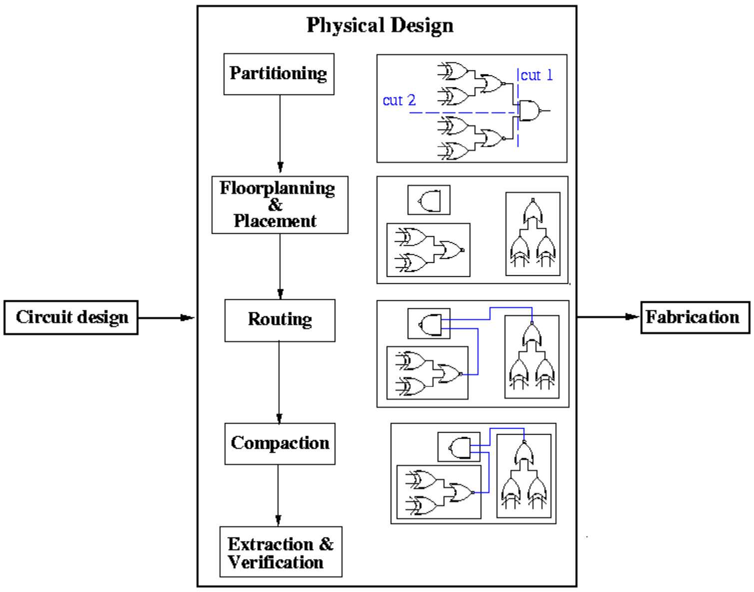


Physical Design



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

Physical Design Flow



Physical Design

Circuit design

Partitioning

Floorplanning & Placement

Routing

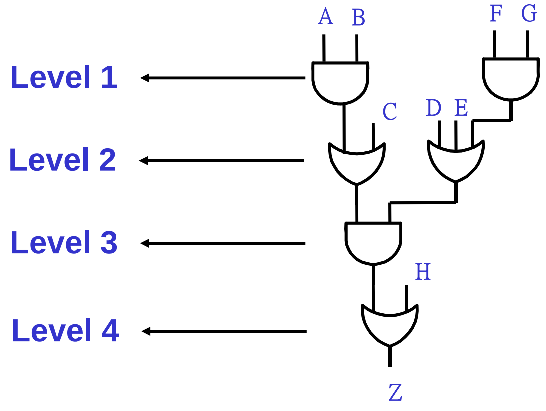
Compaction

Extraction & Verification

Fabrication

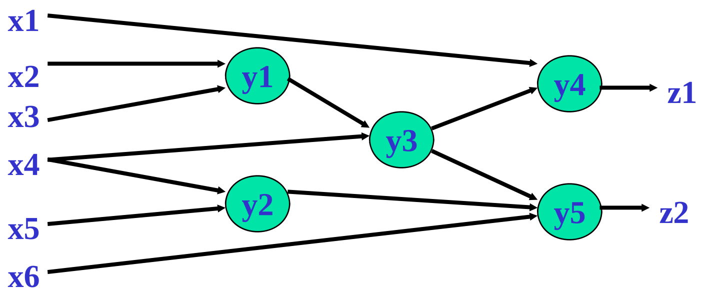
Logic Circuit (or Logic Netlist)

- Multi-level logic:
  - A set of logic equations with no cyclic dependencies
- Example: Z = (AB + C)(D + E + FG) + H
  - 4-level, 6 gates, 13 gate inputs



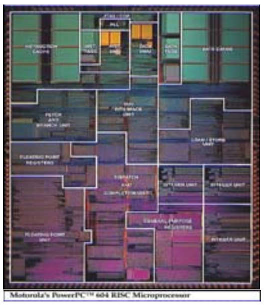
Boolean Network
(Data Structure for A Logic Netist)

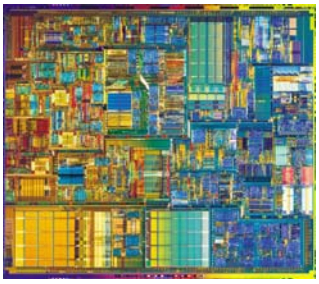
- A Boolean Network
  - Is a Directed Acyclic Graph (DAG)
  - Each source node is a primary input
  - Each sink node is a primary output
  - Each internal node represents an equation
  - Arcs represent variable dependencies



Floorplan Examples

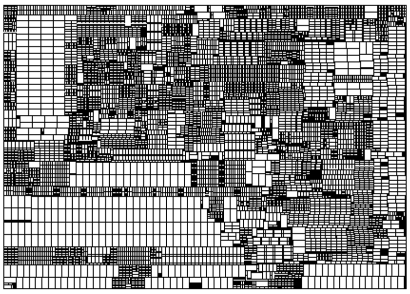
PowerPC 604





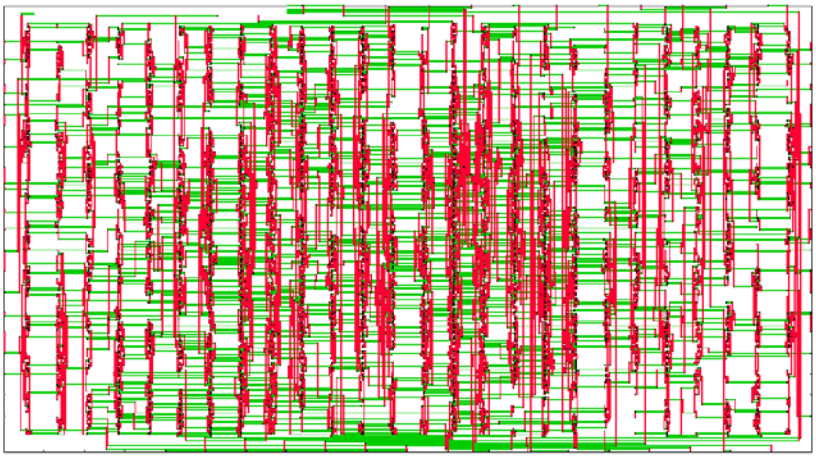
Pentium 4

A floorplan with 9800 blocks

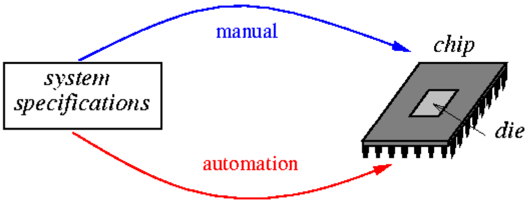


Routing Example

- 0.18um technology



IC Design Considerations



- Several conflicting considerations:
  - Design Complexity: large number of devices/transistors
  - Performance: optimization requirements for high performance
  - Time-to-market: about a 15% gain for early birds
  - Cost: die area, packaging, testing, etc.
  - Others: power, signal integrity (noise, etc), testability, reliability, manufacturability, etc.

Outline

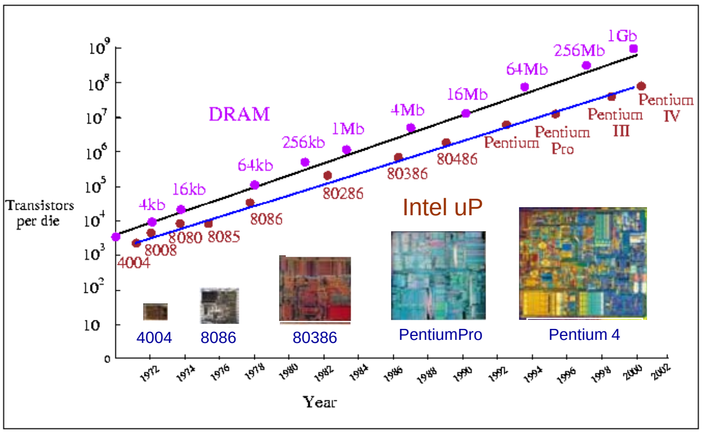
Start

- VLSI Design Flow
- Design Automation Tools
- **Semiconductor Technology Roadmap**
- Design Style
- CMOS Technology Review

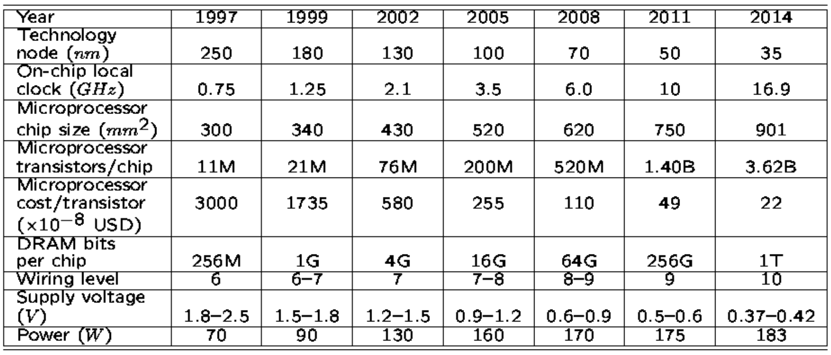
End

“Moore’s” Law: Driving Force of Technology

- **Logic capacity doubles per IC at a regular interval.**
- **Moore:**
  - **Logic capacity doubles every two years (1975).**
- **D. House:**
  - **Computer performance doubles every 18 months (1975)**



Technology Roadmap for Semiconductors

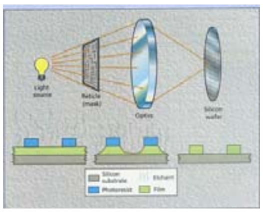


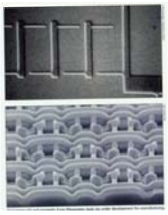
Year	1997	1999	2002	2005	2008	2011	2014
Technology node (nm)	250	180	130	100	70	50	35
On-chip local clock (GHz)	0.75	1.25	2.1	3.5	6.0	10	16.9
Microprocessor chip size (mm^2)	300	340	430	520	620	750	901
Microprocessor transistors/chip	11M	21M	76M	200M	520M	1.40B	3.62B
Microprocessor cost/transistor ($\times 10^{-8}$ USD)	3000	1735	580	255	110	49	22
DRAM bits per chip	256M	1G	4G	16G	64G	256G	1T
Wiring level	6	6–7	7	7–8	8–9	9	10
Supply voltage (V)	1.8–2.5	1.5–1.8	1.2–1.5	0.9–1.2	0.6–0.9	0.5–0.6	0.37–0.42
Power (W)	70	90	130	160	170	175	183

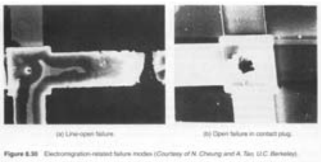
- **Source:**
  - **International Technology Roadmap for Semiconductors, Nov, 2002.**
- **Deep submicron technology: node (feature size) < 0.25 μm.**
- **Nanometer Technology: node < 0.1 μm.**

Nanometer Design Challenges

- **In 2005, feature size ≈ 0.1 μm, μP frequency ≈ 3.5 GHz, die size ≈ 520 mm^2, μP transistor count per chip ≈ 200M, wiring level ≈ 8 layers, supply voltage ≈ 1 V, power consumption ≈ 160 W.**
  - **Feature size ↓ → sub-wavelength lithography (impacts of process variation)? noise? wire coupling? reliability?**
  - **Frequency ↑ → interconnect delay? electromagnetic field effects? timing closure?**
  - **Chip complexity ↑ → large-scale system design methodology?**
  - **Supply voltage ↓ → signal integrity (noise, IR drop, etc)?**
  - **Wiring level ↑ → manufacturability? 3D layout?**
  - **Power consumption ↑ → power & thermal issues?**

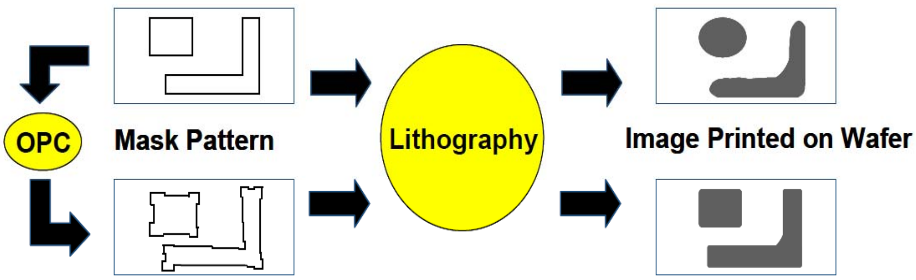






Design for Manufacturability (DfM) - Optical Proximity Correction (OPC)

OPC: An technique that modifies that layout in a way that the distortion of the lithography can be compensated



Severely Distorted!

OPC

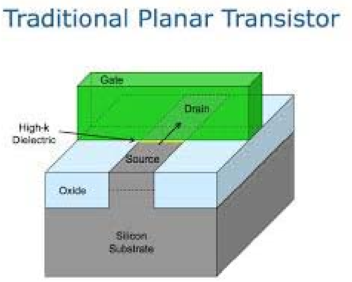
Mask Pattern

Lithography

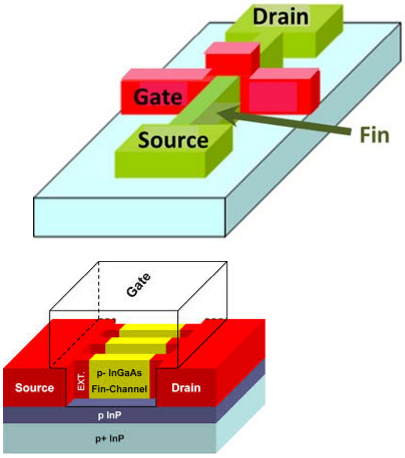
Image Printed on Wafer

Better!

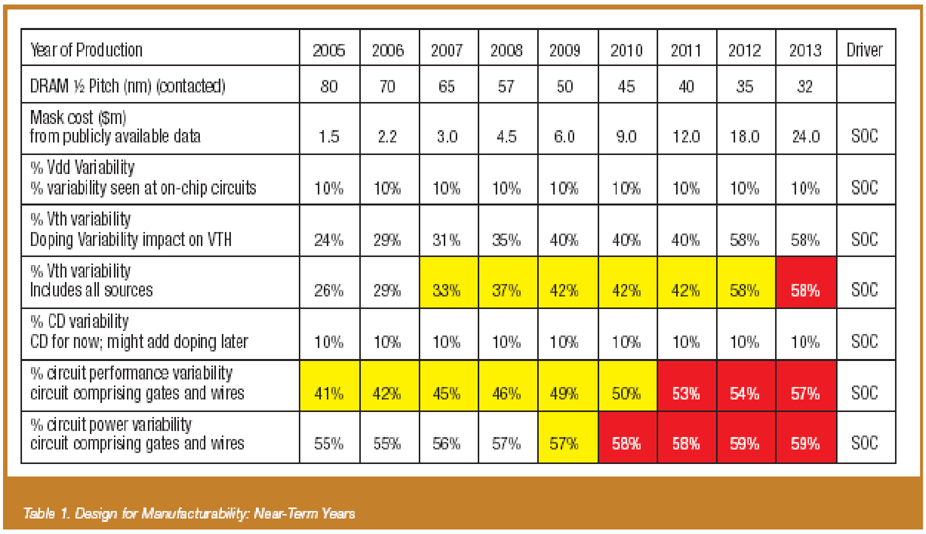
3D Transistor



FinFET Transistor



Worsening Manufacturing Variability



Year of Production	2005	2006	2007	2008	2009	2010	2011	2012	2013	Driver
DRAM ½ Pitch (nm) (contacted)	80	70	65	57	50	45	40	35	32	
Mask cost ($m) from publicly available data	1.5	2.2	3.0	4.5	6.0	9.0	12.0	18.0	24.0	SOC
% Vdd Variability % variability seen at on-chip circuits	10%	10%	10%	10%	10%	10%	10%	10%	10%	SOC
% Vth variability Doping Variability impact on VTH	24%	29%	31%	35%	40%	40%	40%	58%	58%	SOC
% Vth variability Includes all sources	26%	29%	33%	37%	42%	42%	42%	58%	58%	SOC
% CD variability CD for now; might add doping later	10%	10%	10%	10%	10%	10%	10%	10%	10%	SOC
% circuit performance variability circuit comprising gates and wires	41%	42%	45%	46%	49%	50%	53%	54%	57%	SOC
% circuit power variability circuit comprising gates and wires	55%	55%	56%	57%	57%	58%	58%	59%	59%	SOC

Table 1. Design for Manufacturability: Near-Term Years

http://www.future-fab.com/documents.asp?d_ID=3996

More Moore + More Than Moore

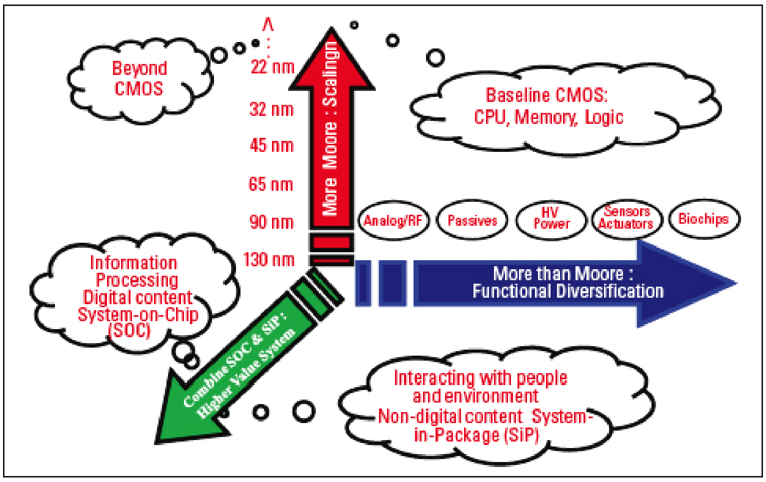


FIGURE 1. SoC and SiP technologies provide a path for continued improvement in performance, power, cost and size at the system level, exclusive of conventional CMOS scaling.

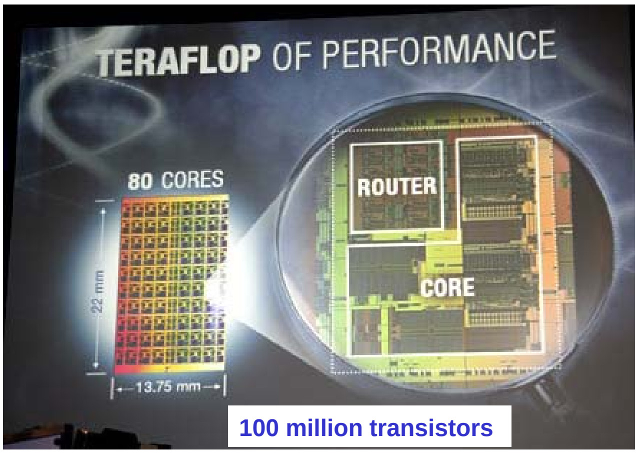
Source" http://pcdandf.com/cms/magazine/212/4495-sips-give-more-to-moore

From Multi-Core to Many-Core Era

New Elements:

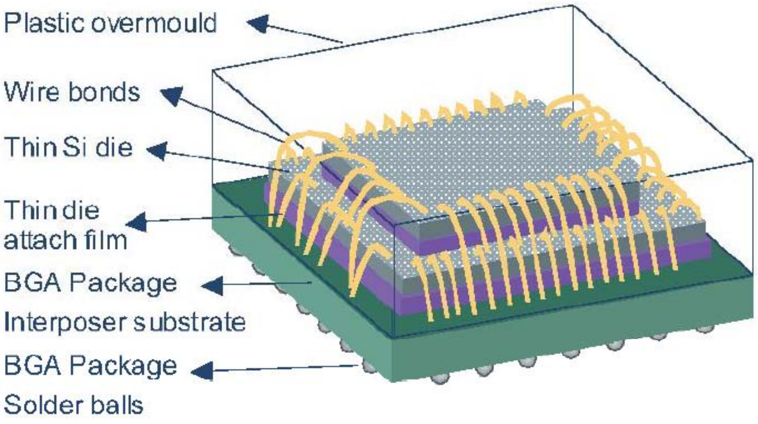
(1) (HW) Network on Chip (NoC) + Global-Local Memory Architecture

(2) (SW) Multi-thread programming



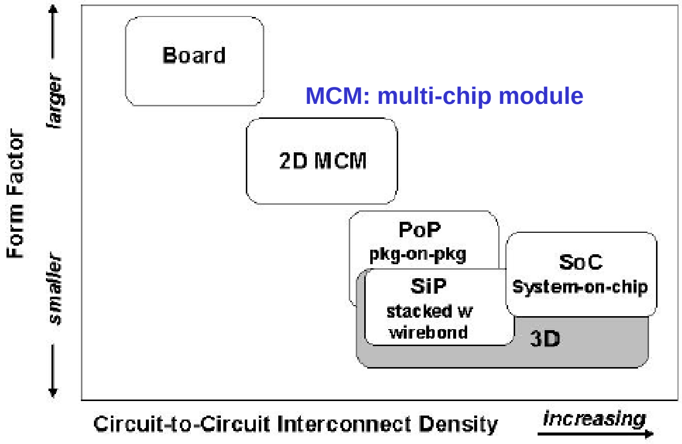
100 million transistors

SiP: Stacked Dies with Wire Bonding



Ref: E. Beyne, “3D System Integration Technologies”

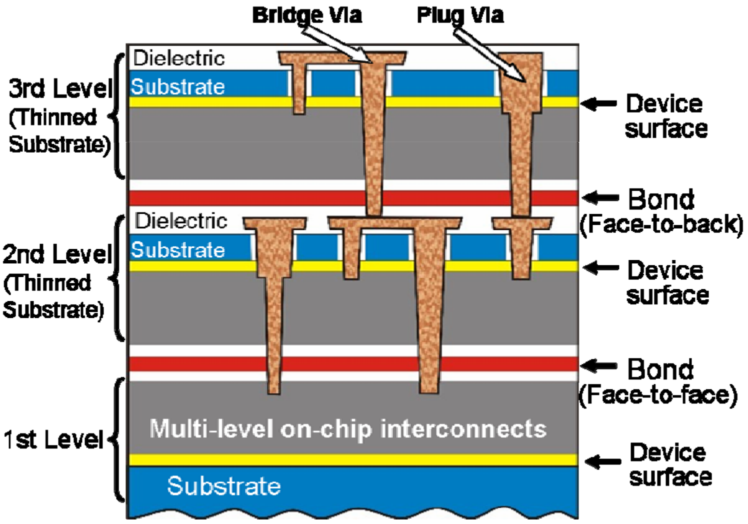
Evolution of System Interconnect Technologies



MCM: multi-chip module

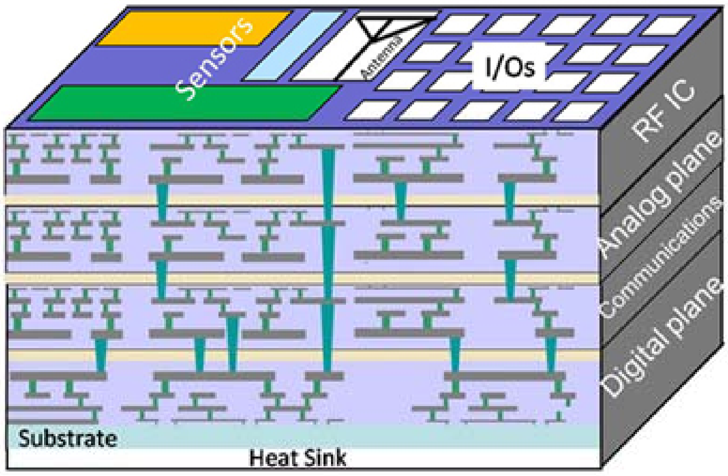
Ref: R. E. Jones, R. Chatterjee, and S. Pozder,
“Technology and Application of 3D Interconnect”

True 3D IC
(with TSV - Through Silicon Via)



http://www.process-evolution.com/3d-ics_doe.html

Illustration of a 3D IC



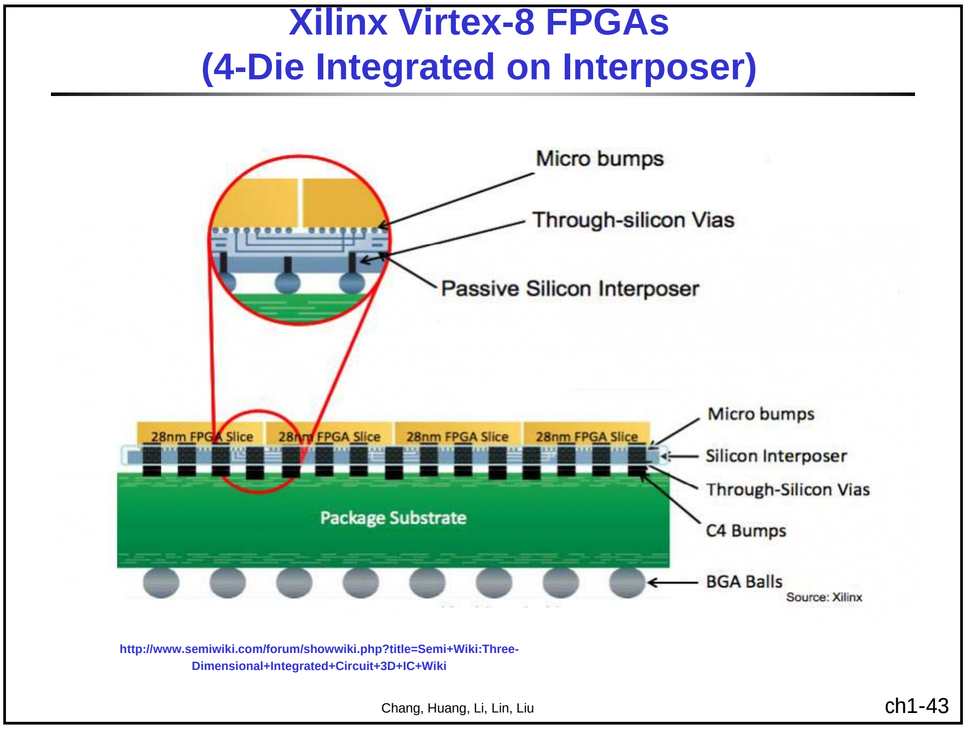
http://lsi.epfl.ch/page-13136-en.html

Benefits of 3D IC's

- Smaller form-factor
- Shorter Interconnect
- Lower Power
- Higher Yield?
- Heterogeneous Integration
- Fast and High-Bandwidth between logic and memory

Design-and-Test Challenges

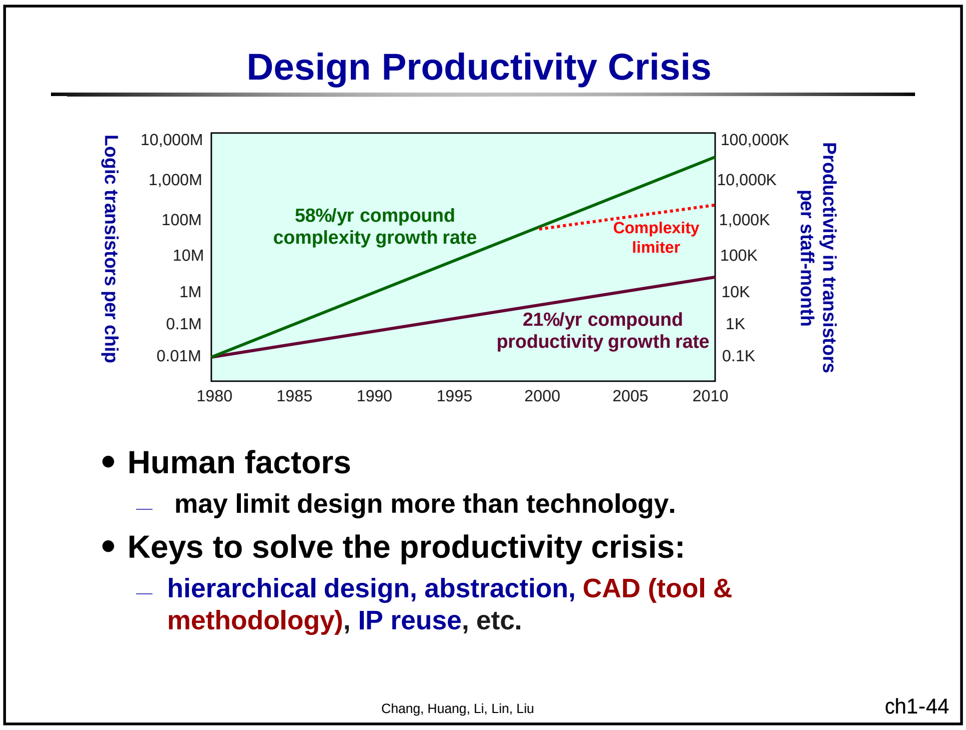
- 3D IC Process / Manufacturing
  - Aligning, stacking, thinning, TSV's
- New Process / Memory Architecture
- Power Delivery
- 3D Design Flow
- Floor-plan & Layout
- Thermal Modeling
- Yield Enhancement
  - Design for Yield & Resiliency
- Testing
  - Electrical Characterization of TSV, Boundary Scan, Known-Good-Die (KGD)



Xilinx Virtex-8 FPGAs (4-Die Integrated on Interposer)

Source: Xilinx

http://www.semiwiki.com/forum/showwiki.php?title=Semi+Wiki:Three-Dimensional+Integrated+Circuit+3D+IC+Wiki

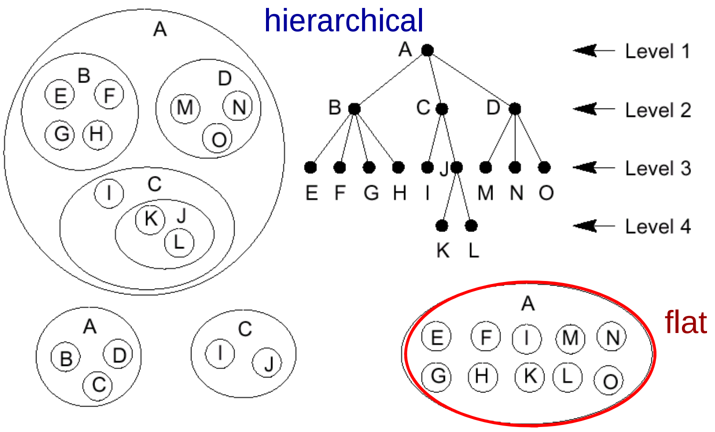


Design Productivity Crisis

- **Human factors**
  - **may limit design more than technology.**
- **Keys to solve the productivity crisis:**
  - **hierarchical design, abstraction, CAD (tool & methodology), IP reuse, etc.**

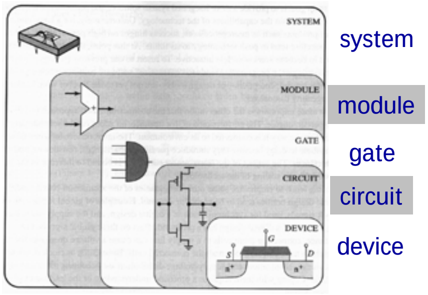
Hierarchical Design

- ***Hierarchy:*** **something is composed of simpler things.**
- **Design cannot be done in one step ⇒ partition the design hierarchically.**



Abstraction

- ***Abstraction:*** **when looking at a certain level, you don't need to know all details of the lower levels.**



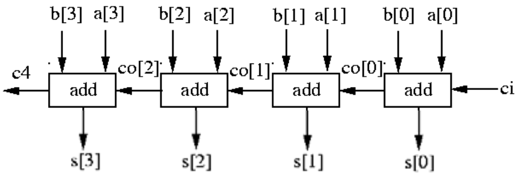
- **Design domains:**
  - **Behavioral: black box view**
  - **Structural: interconnection of subblocks**
  - **Physical: layout properties**
- **Each design domain has its own hierarchy.**

Three Design Views

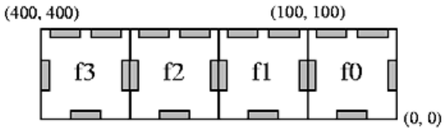
Behavior

```
module add4 (s, c4, ci, a, b);
          input [3:0] a, b;
          input ci;
          output [3:0] s;
          output c4;
          wire [2:0] co;
                 add f0 (co[0], s[0], a[0], b[0], ci);
                 add f1 (co[1], s[1], a[1], b[1], co[0]);
                 add f2 (co[2], s[2], a[2], b[2], co[1]);
                 add f3 (c4, s[3], a[3], b[3], co[2]);
endmodule
```

Structural



Physical

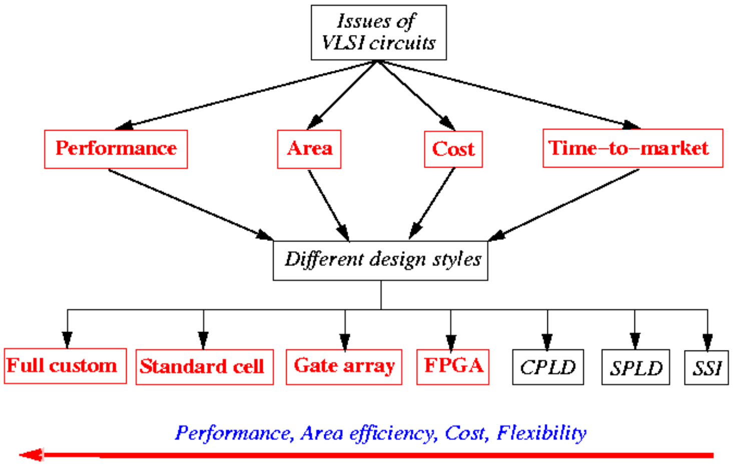


Outline

- **Start** → **VLSI Design Flow**
- **Design Automation Tools**
- **Semiconductor Technology Roadmap**
- **Design Style**
- **CMOS Technology Review** → **End**

Design Styles

- **Specific design styles shall require specific CAD tools**



Issues of VLSI circuits

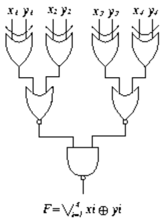
Performance **Area** **Cost** **Time–to–market**

Different design styles

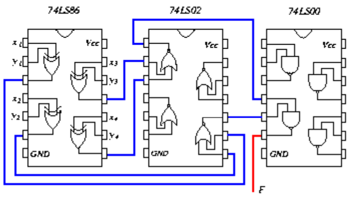
Full custom **Standard cell** **Gate array** **FPGA** *CPLD* *SPLD* *SSI*

Performance, Area efficiency, Cost, Flexibility

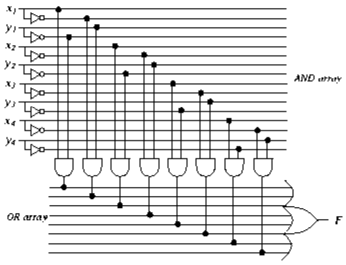
SSI/SPLD Design Style



(a) 4–bit comparator.



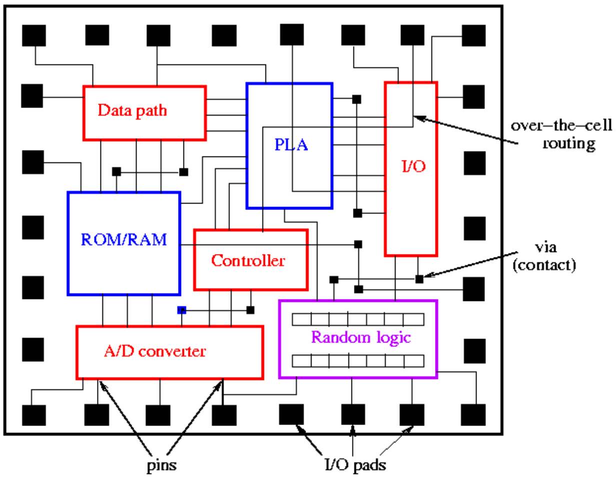
(b) SSI implementation.



(c) SPLD (PLA) implementation.

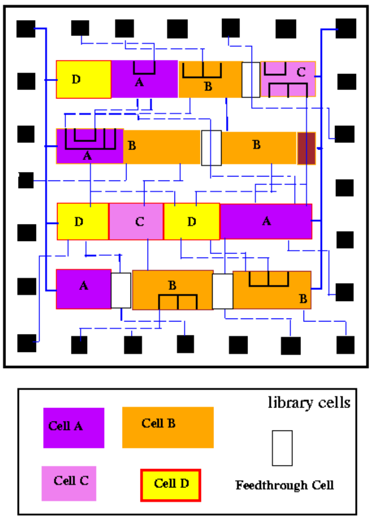
Full Custom Design Style

- Designers can control the shape of all mask patterns.
- Designers can specify the design up to the level of individual transistors.



Standard Cell Design Style

- Selects pre-designed cells (of same height) to implement logic

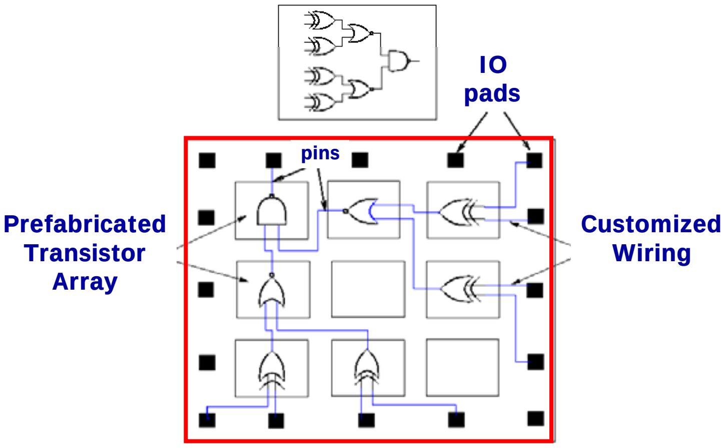


Example: Standard Cells

Interconnections are routed over the cells.

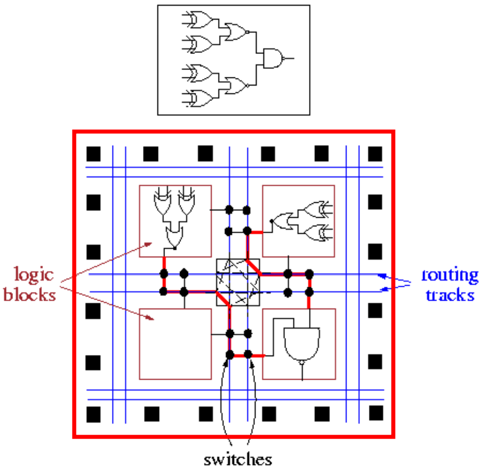
Gate Array Design Style

- **Prefabricates a transistor array**
- **Needs wiring customization to implement logic**



FPGA Design Style

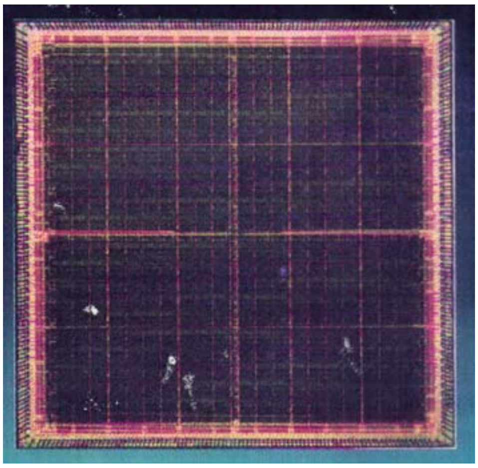
- **Logic and interconnects are both prefabricated.**
- **Illustrated by a symmetric array-based FPGA**



Prefabricated all chip components

Array-Based FPGA Example

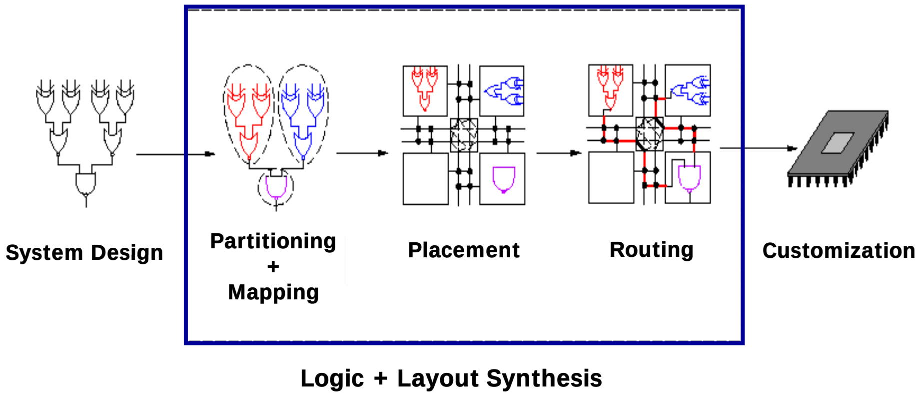
- **Lucent Technologies 15K ORCA FPGA**



- **0.5 um 3LM CMOS**
- **2.45 M Transistors**
- **1600 Flip-flops**
- **25K bit user RAM**
- **320 I/Os**

FPGA Design Process

- Illustrated by a symmetric array-based FPGA
- No fabrication is needed



System Design

Partitioning + Mapping

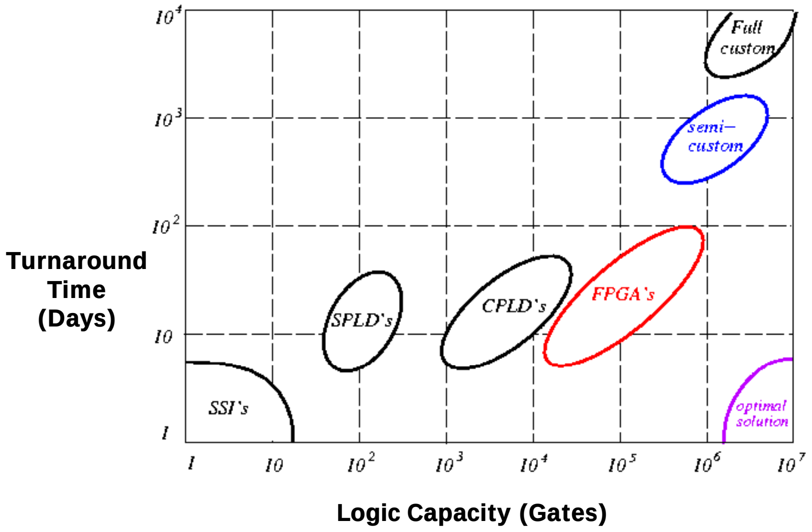
Placement

Routing

Customization

Logic + Layout Synthesis

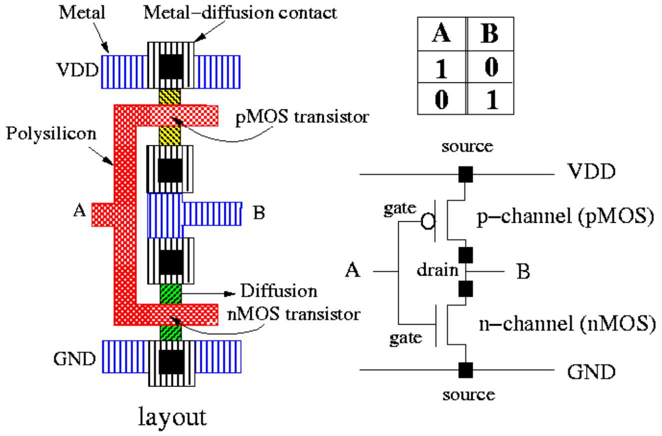
Design Style Trade-offs



Outline

Start →
- **VLSI Design Flow**
- **Design Automation Tools**
- **Semiconductor Technology Roadmap**
- **Design Style**
- **CMOS Technology Review** → **End**

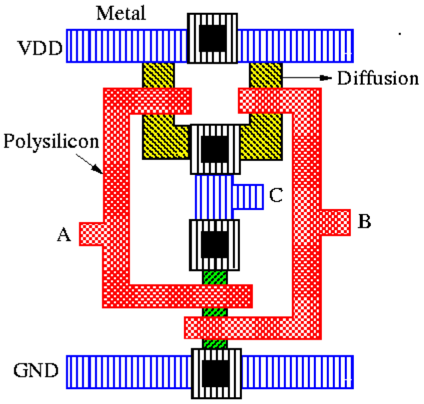
A CMOS Inverter



A	B
1	0
0	1

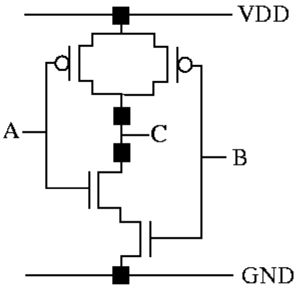
- ▥ metal 1: blue
- ▤ metal 2: brown
- ▩ polysilicon: red
- ■ contact/via: black
- ▨ p−diffusion: yellow (p−well: light yellow)
- ▨ n−diffusion: green (n−well: light green)

A CMOS NAND Gate

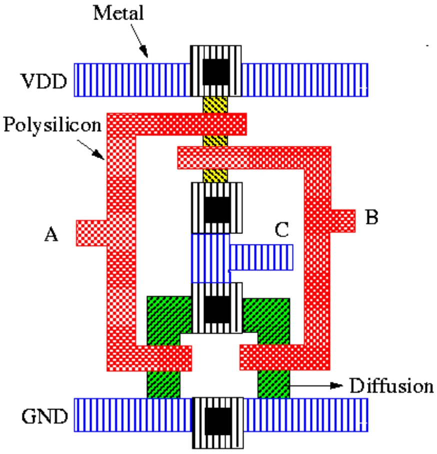


layout

in1	in2	out
A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

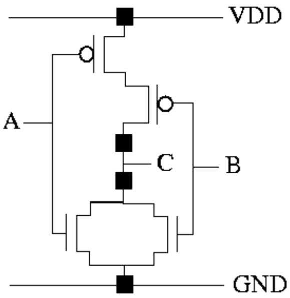


A CMOS NOR Gate

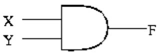
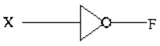
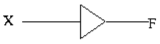
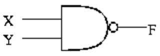


layout

A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

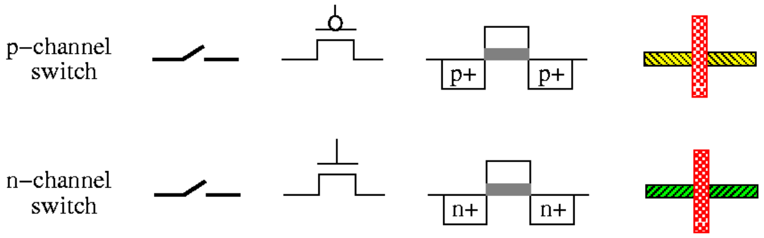


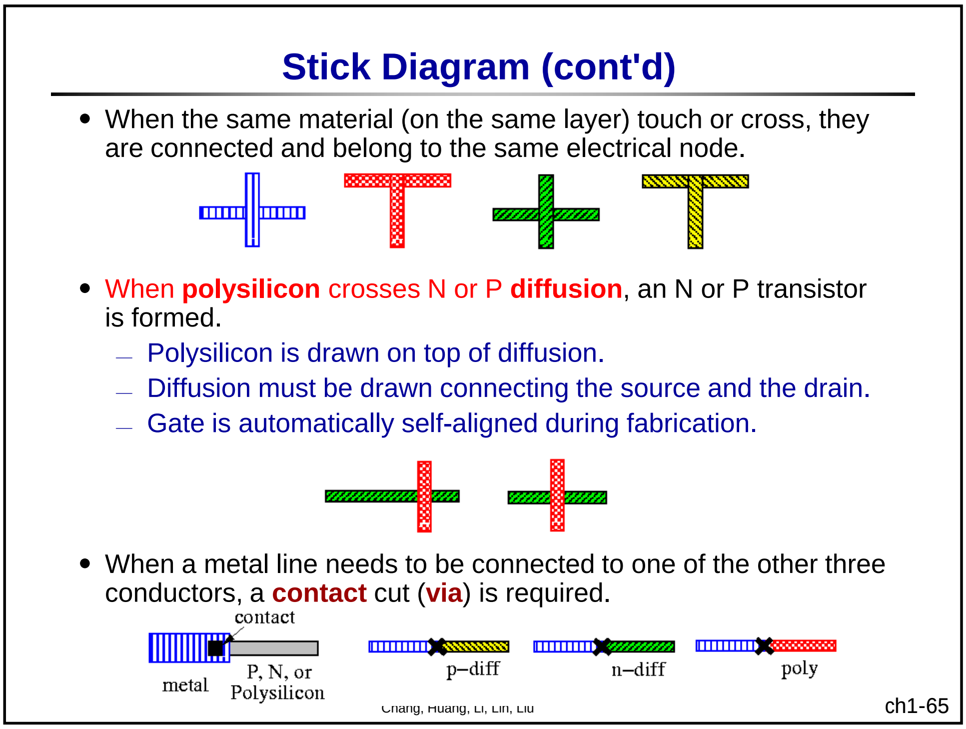
Basic CMOS Logic Library

Name	Distinctive shape	Algebraic equation	Cost (# of transistors)	Scaled gate delay (ps)
AND		F=XY	6	24
OR		F=X+Y	6	24
NOT (inverter/ repeater)		$F=\overline{X}$	2	10
Buffer (driver/ repeater)		F=X	4	20
NAND		$F=\overline{XY}$	4	14
NOR		$F=\overline{X+Y}$	4	14
Exclusive−OR (XOR)		$F=X\overline{Y}+\overline{X}Y$ $=X\oplus Y$	14	42

Stick Diagram

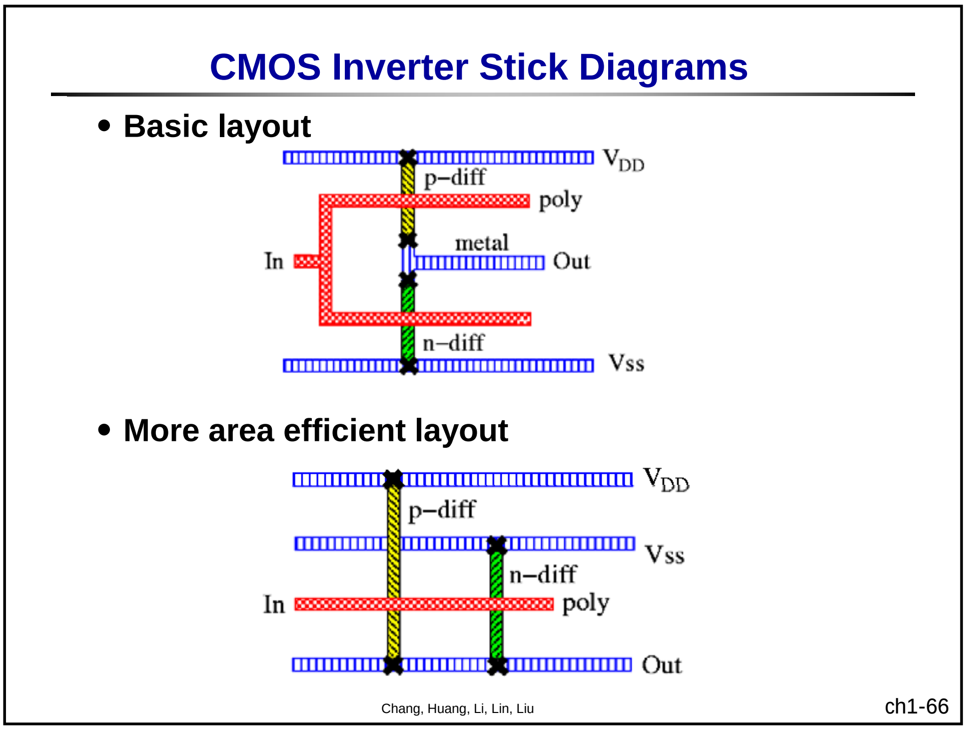
- **Intermediate representation**
  - **between the transistor level and the mask (layout) level.**
- **Gives topological information**
  - **(identifies different layers and their relationship)**
- **Assumes that wires have no width.**
- **It is possible**
  - **to translate stick diagram automatically to layout with correct design rules.**





Stick Diagram (cont'd)

- When the same material (on the same layer) touch or cross, they are connected and belong to the same electrical node.
- When **polysilicon** crosses N or P **diffusion**, an N or P transistor is formed.
  - Polysilicon is drawn on top of diffusion.
  - Diffusion must be drawn connecting the source and the drain.
  - Gate is automatically self-aligned during fabrication.
- When a metal line needs to be connected to one of the other three conductors, a **contact** cut (**via**) is required.

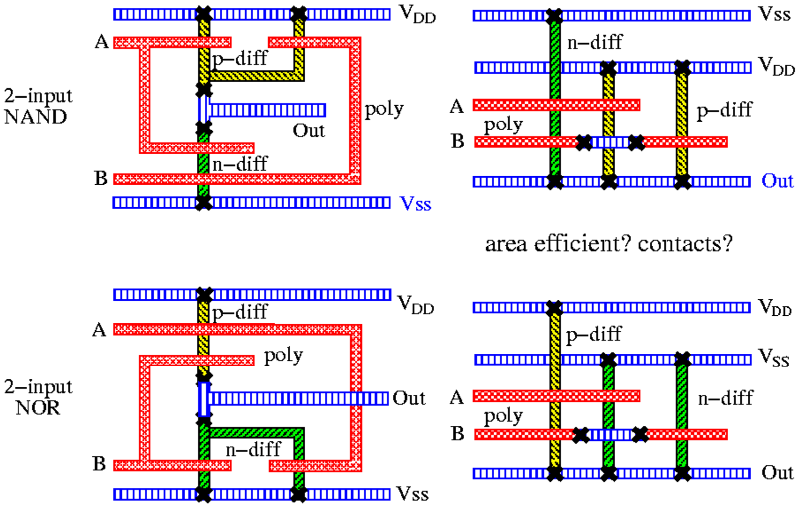


CMOS Inverter Stick Diagrams

- **Basic layout**

- **More area efficient layout**

CMOS NAND/NOR Stick Diagrams

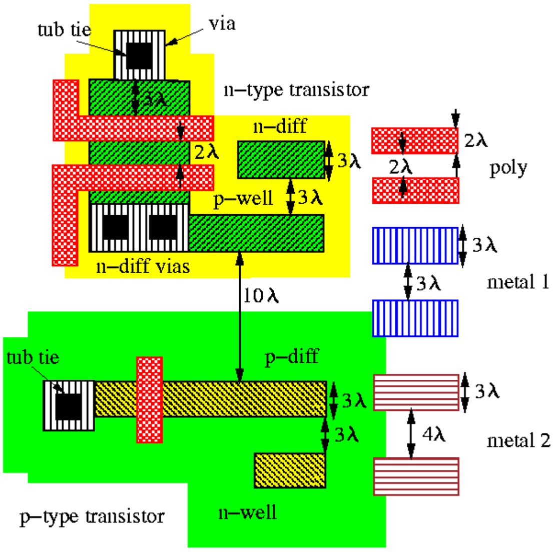


area efficient? contacts?

Design Rules

- **Layout rules are used for preparing the masks for fabrication.**
- **Fabrication processes have inherent limitations in accuracy.**
- **Design rules specify geometry of masks to optimize yield and reliability (trade-offs: area, yield, reliability).**
- **Three major rules:**
  - **Wire width: Minimum dimension associated with a given feature.**
  - **Wire separation: Allowable separation.**
  - **Contact: overlap rules.**
- **Two major approaches:**
  - **“Micron” rules: stated at micron resolution.**
  - **λ rules: simplified micron rules with limited scaling attributes.**
- **λ may be viewed as the size of minimum feature.**
- **Design rules represents a tolerance which insures very high probability of correct fabrication (not a hard boundary between correct and incorrect fabrication).**
- **Design rules are determined by experience.**

SCMOS Design Rules



MOSIS Layout Design Rules

- **MOSIS design rules (SCMOS rules) are available at http://www.mosis.org.**
- **3 basic design rules:**
  - **Wire width**
  - **Wire separation**
  - **Contact rule**
- **MOSIS design rule examples**

R1	Min active area width	3 λ
R3	Min poly width	2 λ
R4	Min poly spacing	2 λ
R5	Min gate extension of poly over active	2 λ
R8	Min metal width	3 λ
R9	Min metal spacing	3 λ
R10	Poly contact size	2 λ
R11	Min poly contact spacing	2 λ

Concluding Remarks

- Milestones technology in silicon era
  - Transistor ➔ Integrated Circuits ➔ CMOS Technology
- Key weapons in SOC era
  - Design Automation
  - Design Reuse
- Breakthrough techniques in design automation
  - Simulation (e.g., SPICE, Verilog-XL, etc.)
  - Automatic Placement and Routing (APR)
  - Logic Synthesis (e.g., Design Compiler)
  - Formal Verification
  - Test Pattern Generation

It is EDA that

pushes the IC design technology forward !

Latest Design Automation – by Synopsys

- 10M Gate Routing in Under ½ Hour
- Complete Physical Verification Solution Through 45nm
- Design Compiler® Graphical: Congestion Prediction and Removal During Synthesis
- Get to Market Early with SystemC™ TLM Virtual Platforms
- Hot Topics in Test: Power- and Timing-Aware DFT
- Low Power Verification
- Matching Moore's Law, PrimeTime® Performance, Capacity and QoR
- Mixed-Signal Circuit Design and Verification with Discovery™-AMS and Synopsys' Custom Environment
- Synopsys Eclypse™ Low Power Solution
- SystemVerilog Verification Solution with VCS®
- Transistor-Level Design Analysis and Sign-Off Using Star-RCXT®, HSIM™ and HSPICE®

CAD Related Conferences/Journals

- Important Conferences:
  - ACM/IEEE Design Automation Conference (DAC)
  - IEEE/ACM Int'l Conference on Computer-Aided Design (ICCAD)
  - ACM/IEEE Asia and South Pacific Design Automation Conf. (ASP-DAC)
  - ACM/IEEE Design, Automation, and Test in Europe (DATE)
  - IEEE Int'l Conference on Computer Design (ICCD)
  - IEEE Custom Integrated Circuits Conference (CICC)
  - IEEE Int'l Symposium on Circuits and Systems (ISCAS)
  - ACM Int'l Symposium on Physical Design (ISPD)
  - IEEE Int'l Test Conference (ITC)
  - Others: VLSI Design/CAD Symposium/Taiwan
- Important Journals:
  - IEEE Transactions on Computer-Aided Design (TCAD)
  - ACM Transactions on Design Automation of Electronic Systems (TODAES)
  - IEEE Transactions on VLSI Systems (TVLSI)
  - IEEE Transactions on Computers (TC)
  - IEE Proceedings – Circuits, Devices and Systems
  - IEE Proceedings – Digital Systems
  - INTEGRATION: The VLSI Journal

清華大學 EE 5265
積體電路設計自動化

單元 2
Generic Algorithms

教育部顧問室
「超大型積體電路與系統設計」教育改進計畫
EDA聯盟 – 推廣課程

致謝

本單元之教材主要取自於
教育部
超大型積體電路與系統設計教育改進計畫
EDA聯盟之課程發展成果

(教材編纂小組成員)
台灣大學電機系 張耀文
清華大學電機系 黃錫瑜
交通大學資科系 李毅郎
中央大學電機系 劉建男
元智大學資工系 林榮彬

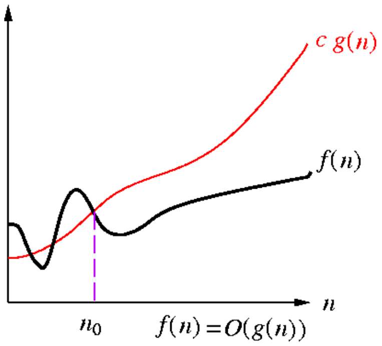
教育部顧問室
「超大型積體電路與系統設計」教育改進計畫
EDA聯盟 – 推廣課程

Outline

- **Complexity**
- **Common Problems in EDA**
  - **Optimization Problem**
  - **Decision Problem**
  - **Satisfiability Problem**
- **General-Purpose Algorithms**
  - **Exhaustive v.s. Branch-and-Bound**
  - **Greedy v.s. Dynamic Programming**
  - **Divide-and-Conquer v.s. Hierarchical**
  - **Mathematical Programming**
  - **Simulated Annealing**
  - **Tabu Search**
  - **Genetic Algorithm**

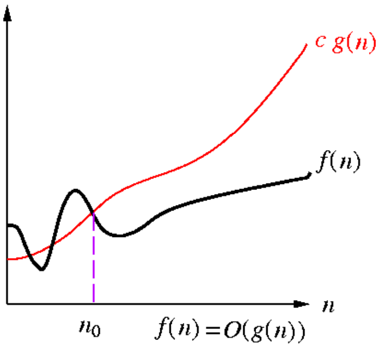
O: Upper Bounding Function

- **Def:** $f(n) = O(g(n))$ if $\exists\, c > 0$ and $n_0 > 0$ such that $0 \le \mathbf{f(n)} \le \mathbf{cg(n)}$ for all $n \ge n_0$.
  - Examples: $2n^2 + 3n = O(n^2)$, $2n^2 = O(n^3)$, $3n \log n = O(n^2)$
- Intuition: $f(n)$ "$\le$" $g(n)$ when we ignore constant multiples and small values of n.



Big-*O* Notation

- How to show *O* (Big-Oh) relationships?
  - $f(n) = O(g(n))$ iff $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$ for some **$c \geq 0$.**
- "An algorithm has worst-case running time $O(f(n))$": there is a constant c such that (s.t.) for big enough value n, **the execution** on an input of size n takes **at most** $cf(n)$ time.



Computational Complexity

- **Computational complexity:**
  - **an abstract measure of the time and space necessary to execute an algorithm as a function of its "input size".**
- **Input size examples:**
  - **(1) sort *n* words of bounded length ⇒ *n***
  - **(2) the input is a graph *G(V, E)* ⇒ |*V*| and |*E*|**
- **Time complexity**
  - **is expressed in *elementary computational steps* (e.g., an addition, multiplication, pointer indirection).**
- **Space Complexity**
  - **is expressed in *memory locations* (e.g. bits, bytes, words).**

Asymptotic Functions

- **Polynomial-time complexity:**
  - $O(n^k)$, where n is the input size and k is a constant.
- **Example polynomial functions:**
  - 999: constant
  - log n: logarithmic (sub-linear)
  - n: linear
  - n log n: log-linear
  - n^2: quadratic
  - n^3: cubic
- **Example non-polynomial functions**
  - 2^n, 3^n: exponential
  - n!: factorial

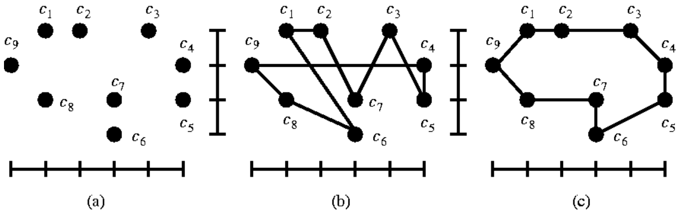
Optimization Problems

- **Optimization problems:**
  - Those finding a legal configuration such that its cost is minimum (or maximum).
- **An instance $\alpha = (F, c)$ where**
  - (1) Feasible solution space: F
    - F is also referred to as search space
  - (2) Cost function: c: F → R
    - Assigning a cost value to each feasible solution
- **Example**
  - Minimum Spanning Tree (MST)
  - Given a graph $G=(V, E)$, find the cost of a minimum spanning tree of G.

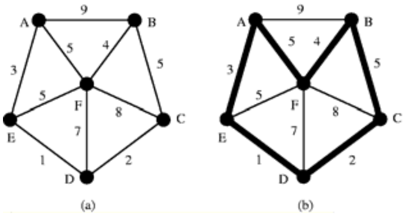
The Traveling Salesman Problem (TSP)

- **Problem Definition of TSP:**
  - **Given a set of cities and the distance between each pair of cities.**
  - **Find the distance of a "minimum tour" both starting and ending at a given city and visiting every city exactly once.**

Euclidean Space

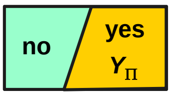


Graph Space



Decision Problem

- **Decision problems:**
  - **problem with "yes" or "no" answer**
- **Examples:**
  - **(1) MST: Given a graph *G*=(*V, E*) and a bound *K*, is there a spanning tree with a cost at most *K*?**
  - **(2) TSP: Given a set of cities, distance between each pair of cities, and a bound *B*, is there a route that starts and ends at a given city, visits every city exactly once, and has total distance at most *B*?**



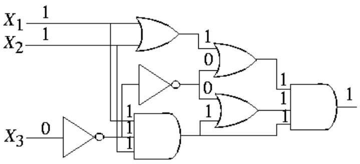
- **A decision problem Π = (*F*, *c*, *k*)**
  - **Solution Space Y_Π :**
    - **The input sub-space for which the answer is "yes"**
  - **Solution Checking: (deciding if an input point is in Y_Π)**
    - **Checking whether the cost of a solution point, $f \in F$, is less than *k*.**
- **Could apply binary search on decision problems to obtain solutions for optimization problems.**
- **NP-completeness is associated with decision problems.**

Boolean Satisfiability Problem (SAT)

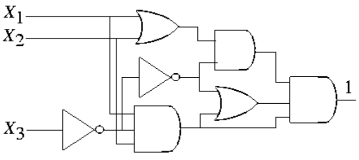
- **Given**
  - **n binary variables $\{x_1, x_2, \dots, x_n\}$**
  - **A Boolean expression in Product-of-Sum (POS) form**
- **Boolean Satisfiability Problem**
  - **Is a decision problem**
  - **Decides if there is a variable assignment such that every term evaluates to true?**
- **Example**: $\underbrace{(x1 + x3 + x4)}_{\text{A Term or Clause}}(x1 + x2 + x5)(x3 + x4 + x5)$

The Circuit-Satisfiability Problem (Circuit-SAT)

- **The Circuit-Satisfiability Problem (Circuit-SAT):**
  - **Instance: A combinational circuit *C* composed of AND, OR, and NOT gates.**
  - **Question: Is there a set of input values (called input pattern or input vector) that makes the output of *C* to 1?**
- **A circuit is satisfiable**
  - **if there exists an input pattern that makes the output of the circuit to be 1.**
  - **Circuit (a) is satisfiable since $<x_1, x_2, x_3> = <1, 1, 0>$ makes the output to be 1, while circuit (b) is not satisfiable.**

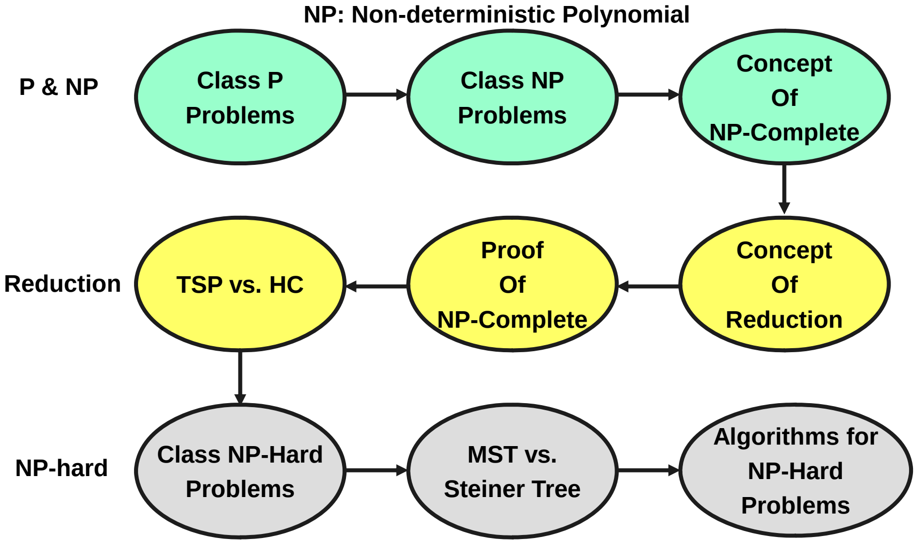


(a) satisfiable circuit



(b) unsatisfiable circuit

Discussion Flow On Problem Complexity



NP: Non-deterministic Polynomial

P & NP

Class P Problems

Class NP Problems

Concept Of NP-Complete

Reduction

TSP vs. HC

Proof Of NP-Complete

Concept Of Reduction

NP-hard

Class NP-Hard Problems

MST vs. Steiner Tree

Algorithms for NP-Hard Problems

Complexity of Class-*P* Problems

- **The Class-*P* problems**
  - **Are problems that can be solved in polynomial time in terms of input size**
  - **Problems in Class-P are considered tractable.**
- **Computational Model: *deterministic Turing machine***
  - **(1) A *Turing machine* is a mathematical model of a generic computer (any computation that needs polynomial time on a Turing machine can also be performed in polynomial time on any other machine).**
  - **(2) *"Deterministic"* means that each computational step is predictable.**
- **Example:**
  - **Minimum Spanning Tree Problem is a class-*P* problem.**

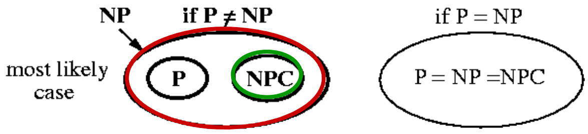
Complexity Class-NP

- Suppose that *solution checking* for a given problem can be done in polynomial time on a deterministic machine $\Rightarrow$ Then, the problem can be solved in polynomial time on a *nondeterministic Turing machine.*
  - *Nondeterministic*: in some sense the machine is able to evaluate all possibilities in parallel.
- The class-NP (Nondeterministic Polynomial):
  - (1) Is a class of problems that can be verified in polynomial time in the size of input.
  - (2) NP is also a class of problems that can be solved in polynomial time on a nondeterministic machine.
- Is TSP $\in$ NP?
  - Need to check a solution in polynomial time.
    - Guess a tour.
    - Check if the tour visits every city exactly once.
    - Check if the tour returns to the start.
    - Check if the total distance $\leq B$.
  - All can be done in $O(n)$ time, so TSP $\in$ NP.

NP-Completeness

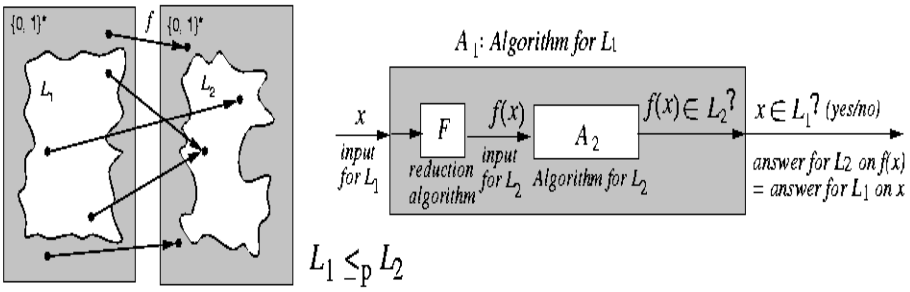
- An issue which is still unsettled:

  $P \subset NP$ or $P = NP$?
- There is a strong belief that $P \neq NP$, due to the existence of *NP*-complete problems.
- The class NP-complete (NPC):
  - Developed by S. Cook and R. Karp in early 1970.
  - All problems in NPC have the same degree of difficulty: Any NPC problem can be solved in polynomial time $\Rightarrow$ all problems in NP can be solved in polynomial time.



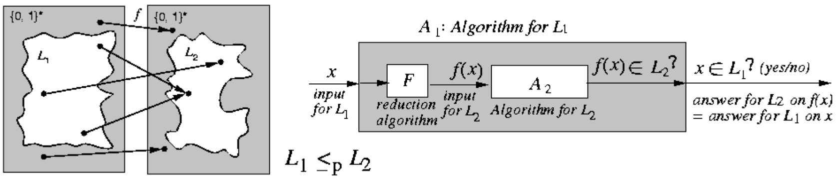
Reduction

- **Given**
  - **Two decision problems, L_1 and L_2**
- **Reduction**
  - **(1) Is a mapping function between the input spaces of L_1 and L_2**
  - **(2) The final yes/no answers are retained**



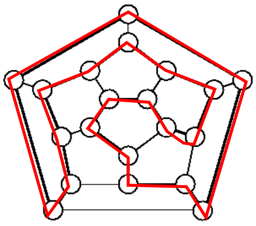
Polynomial-Time Reduction

- **Motivation:**
  - **Let L_1 and L_2 be two decision problems. Suppose algorithm A_2 can solve L_2. Can we use A_2 to solve L_1?**
- **Polynomial-time reduction *f***
  - **from L_1 to L_2: $L_1 \leq_P L_2$**
  - *f* reduces an input for L_1 into an input for L_2 s.t. the reduced input is a "yes" input for L_2 iff the original input is a "yes" input for L_1.
    - $L_1 \leq_P L_2$ if ∃ polynomial-time computable function *f*: $\{0, 1\}^* \rightarrow \{0, 1\}^*$ s.t. $x \in L_1$ iff $f(x) \in L_2$, $\forall x \in \{0, 1\}^*$.
    - **L_2 is at least as hard as L_1.**
  - *f* is computable in polynomial time.

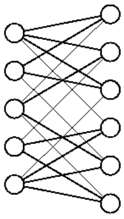


Example: Polynomial-time Reduction

- **The Hamiltonian Circuit Problem (HC)**
  - **Instance: an undirected graph $G = (V, E)$.**
  - **Question: is there a cycle in G that includes every vertex exactly once?**
- **TSP (The Decision-version Traveling Salesman Problem)**
- **How to show HC $\le_P$ TSP?**
  1. **Define a function f mapping any HC instance into a TSP instance, and show that f can be computed in polynomial time.**
  2. **Prove that G has an HC iff the reduced instance has a TSP tour with distance $\le B$ ($x \in$ HC $\Leftrightarrow f(x) \in$ TSP)**



Hamiltonian



nonhamiltonian

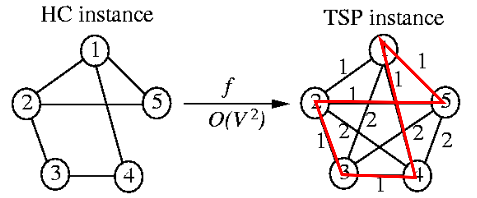
HC $\le_P$ TSP: Step 1

1. Define a reduction function f for HC $\le_P$ TSP.

- **Given an arbitrary HC instance $G = (V, E)$ with n vertices**
  - **Create a set of n cities labeled with names in V.**
  - **Assign distance between u and v**

$$d(u,v) = \begin{cases} 1, & \text{if } (u,v) \in E, \\ 2, & \text{if } (u,v) \notin E. \end{cases}$$

  - **Set bound $B = n$.**
- **f can be computed in $O(V^2)$ time.**

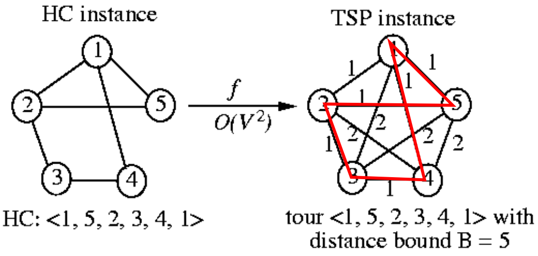


HC: <1, 5, 2, 3, 4, 1>

tour <1, 5, 2, 3, 4, 1> with distance bound B = 5

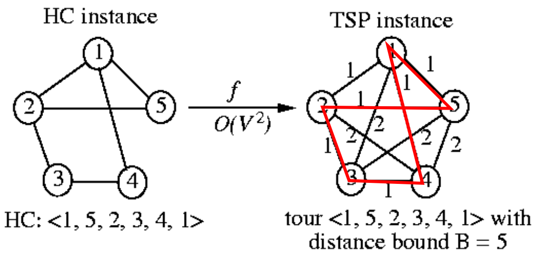
HC $\leq_P$ TSP: Step 2

2. **G has an HC iff the reduced instance has a TSP with distance $\leq B$.**
   - **$x \in$ HC $\Rightarrow f(x) \in$ TSP.**
     - **Suppose the HC is $h = <v_1, v_2, \ldots, v_n, v_1>$. Then, h is also a tour in the transformed TSP instance.**
     - **The distance of the tour h is $n = B$ since there are n consecutive edges in E, each having distance 1 in $f(x)$.**
     - **Thus, $f(x) \in$ TSP ($f(x)$ has a TSP tour with distance $\leq B$.**



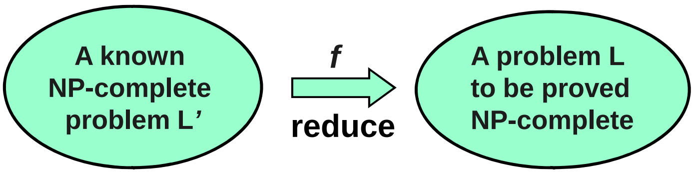
HC $\leq_P$ TSP: Step 2 (cont'd)

2. **G has an HC iff the reduced instance has a TSP with distance $\leq B$.**
   - **$f(x) \in$ TSP $\Rightarrow x \in$ HC.**
     - **Suppose there is a TSP tour with distance $\leq n = B$. Let it be $<v_1, v_2, \ldots, v_n, v_1>$..**
     - **Since distance of the tour $\leq n$ and there are n edges in the TSP tour, the tour contains only edges in E.**
     - **Thus, $<v_1, v_2, \ldots, v_n, v_1>$ is a Hamiltonian cycle ($x \in$ HC).**



Summary of Proving NP-Completeness

- **Five steps for proving that *L* is NP-complete:**
  1. **Prove $L \in$ NP.**
  2. **Select a known NP-complete problem L'.**
  3. **Construct a reduction *f* that can transform any arbitrary instance of L' to an instance of L.**
  4. **Prove that *f* is a polynomial-time transformation**
  5. **Prove that $x \in L'$ iff $f(x) \in L$ for all $x \in \{0, 1\}^*$.**
- We have shown that TSP is NP-complete, since HC is a proven NP-complete problem



A known NP-complete problem L'

$\xrightarrow{f}$ **reduce**

A problem L to be proved NP-complete

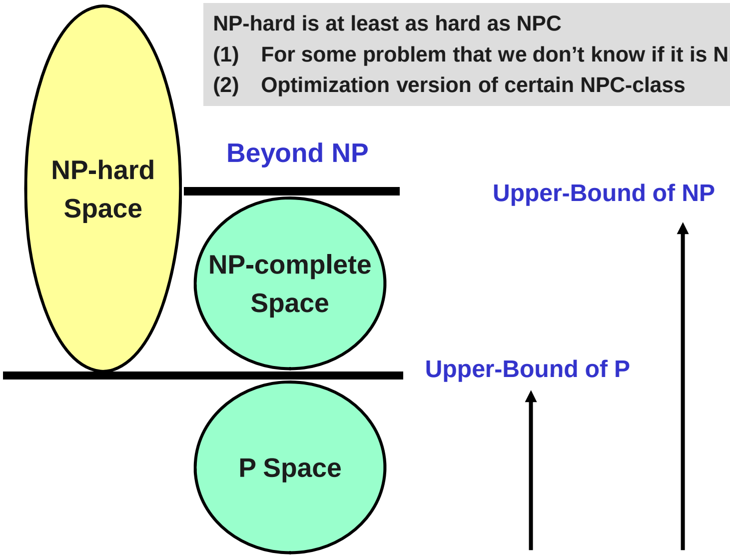
Hamiltonian Circuit problem **TSP problem**

NP-Completeness and NP-Hardness

- **L is NP-complete if**
  - **NP-Hard: $L' \le_P L$ for every $L' \in$ NPC.**
  - **$L \in$ NP**
- **NP-hard: If *L* satisfies the 1st property, but not necessarily the 2nd property, we say that *L* is NP-hard.**

NP-Hard Problems

NP-hard is at least as hard as NPC
(1) For some problem that we don't know if it is NP
(2) Optimization version of certain NPC-class



NP-hard Space

Beyond NP

NP-complete Space

P Space

Upper-Bound of NP

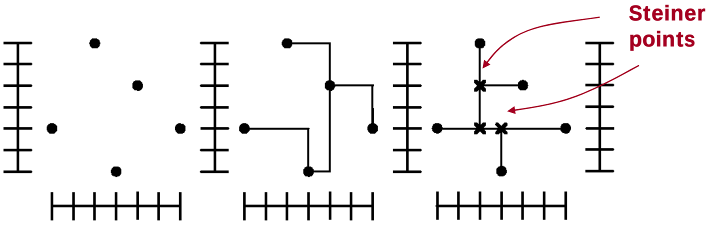
Upper-Bound of P

Coping with NP-hard problems

- Approximate algorithms
  - The solution found is guaranteed to be a fixed percentage away from the optimum.
  - E.g., MST for the minimum Steiner tree problem.
- Pseudo-polynomial time algorithms
  - Has the form of a polynomial function for the complexity, but not in terms of the problem size.
- Restriction
  - Work on some subset of the original problem.
  - E.g., the longest path problem in Directed Acyclic Graphs (DAG).
- Exhaustive search/Branch and bound
  - Is feasible only when the problem size is small.
- Local search:
  - Simulated annealing (hill climbing), genetic algorithms, etc.
- Heuristics: No guarantee of performance.

Spanning Tree v.s. Steiner Tree

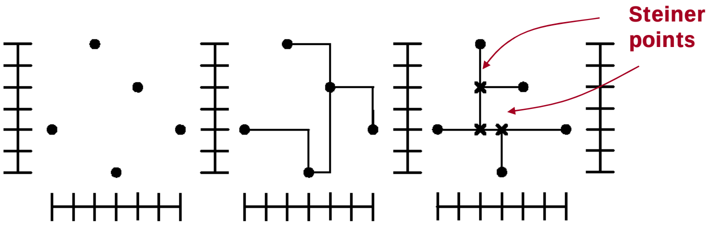
- Manhattan distance:
  - If two points (nodes) are located at coordinates (x_1, y_1) and (x_2, y_2), the Manhattan distance between them is given by $d_{12} = |x_1 - x_2| + |y_1 - y_2|$.
- Rectilinear spanning tree:
  - a spanning tree connected with Manhattan paths (Fig. (b) below).
- Steiner tree:
  - a tree that connects its nodes, and additional points (Steiner points) are permitted to be used for the connections.



(a) (b) (c)

Complexities of Spanning and Steiner Trees

- The minimum rectilinear spanning tree problem is in P
- The minimum rectilinear Steiner tree (Fig. (c)) problem is NP-complete.
  - The spanning tree algorithm can be an *approximation* for the Steiner tree problem (at most 50% away from the optimum).



(a) (b) (c)

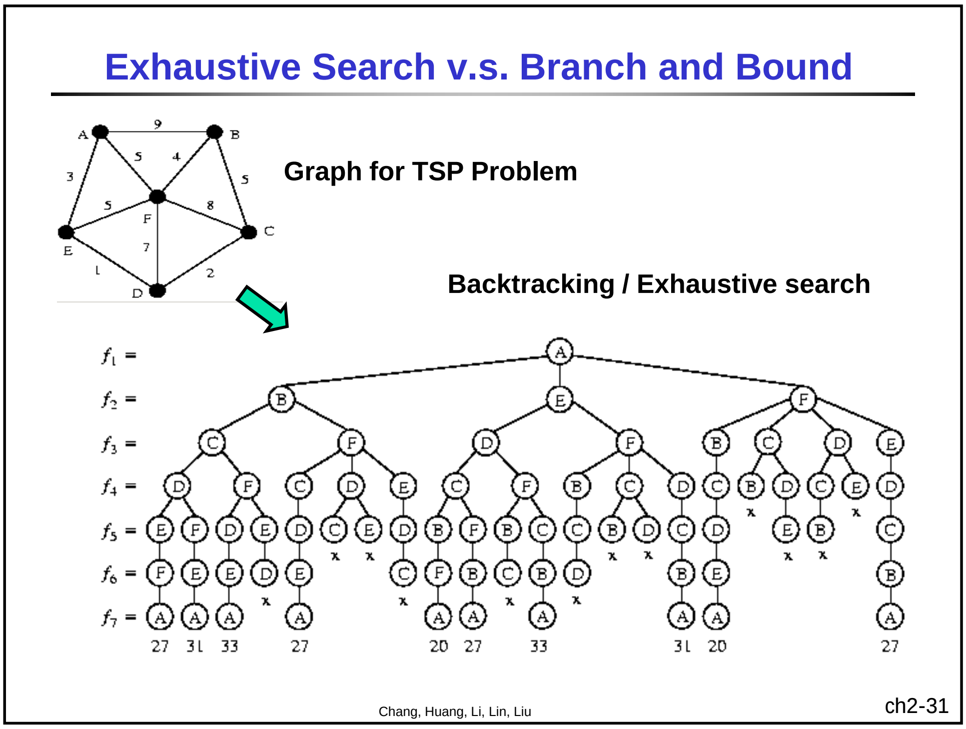
Outline

- **Complexity**
- **Common Problems in EDA**
  - **Optimization Problem**
  - **Decision Problem**
  - **Satisfiability Problem**
- **General-Purpose Algorithms**
  - **Exhaustive v.s. Branch-and-Bound**
  - **Greedy v.s. Dynamic Programming**
  - **Divide-and-Conquer (Hierarchical)**
  - **Mathematical Programming**
  - **Simulated Annealing**
  - **Tabu Search**
  - **Genetic Algorithm**

Search Paradigms

- **Exhaustive search:** Search the entire input space.
- **Branch and bound:** A search technique with pruning.
- **Greedy method:** Pick a locally optimal solution at each step.
- **Dynamic programming:** Partition a problem into a collection of sub-problems, the sub-problems are solved first, and then the original problem is solved by combining the solutions. (Applicable when the sub-problems are **NOT independent**).
- **Hierarchical approach:** Divide-and-conquer.
- **Mathematical programming:** A system of optimizing an objective function under constraints.
- **Simulated annealing:** An adaptive, iterative, non-deterministic algorithm that allows "uphill" moves to escape from local optima.
- **Tabu search:** Similar to simulated annealing, but does not decrease the chance of "uphill" moves throughout the search.
- **Genetic algorithm:** A population of solutions is stored and allowed to evolve through successive generations via mutation, crossover, etc.

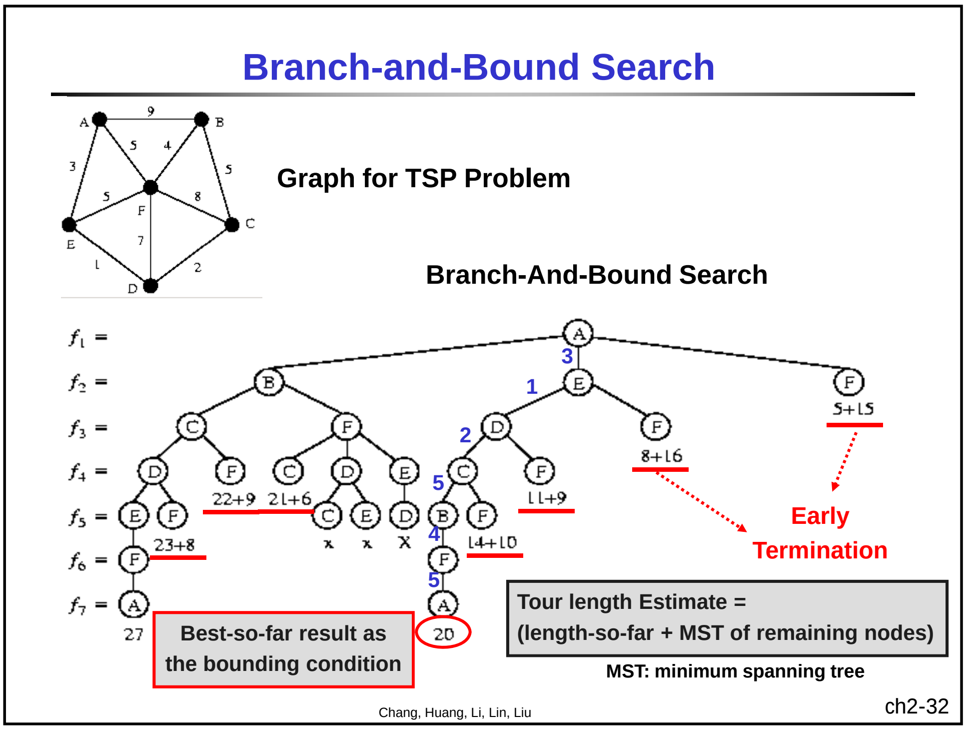
突變 交配



Exhaustive Search v.s. Branch and Bound

Graph for TSP Problem

Backtracking / Exhaustive search



Branch-and-Bound Search

Graph for TSP Problem

Branch-And-Bound Search

Early Termination

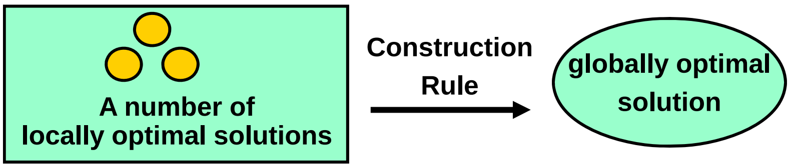
Best-so-far result as the bounding condition

Tour length Estimate =
(length-so-far + MST of remaining nodes)

MST: minimum spanning tree

Dynamic Programming (DP) v.s. Divide-and-Conquer

- Both solve problems by combining the solutions of sub-problems.
- Divide-and-conquer algorithms
  - (1) Partition a problem into independent sub-problems
  - (2) Solve the sub-problems recursively
  - (3) Combine their solutions to derive the final answer
- Dynamic programming (DP)
  - Defines optimal solutions in terms of optimal partial solutions
  - Applicable when the sub-problems are mutually dependent
- Principle of Optimality
  - Parts of the search space can be discarded without losing optimality if DP is exact



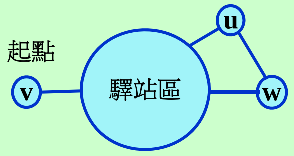
A number of locally optimal solutions

Construction Rule

globally optimal solution

DP-Example 1: Shortest-Path Problem

```
void  ShortestPath( const int n, const int v)  // Dijkstra's Algorithm
// dist[j], 0≦j<n, is set to the length of the shortest path from vertex v to vertex j
// in a graph G with n vertex and edge lengths given by length[i][j]
{
   for ( int  i=0; i<n; i++ ) {  s[i] = FALSE;  dist[i] = length[v][i]; }  // initialize
   s[v] = TRUE;
   dist[v] = 0;

    for ( i=0; i<n-2; i++) {    // problem increases incrementally
          int  u = choose(n);  // routine 'choose' returns a value u such that
                                          // dist[u] = minimum dist[w], where s[w] = FALSE
          s[u] = TRUE;
          for ( int  w=0; w<n; w++) {
                 if ( ! s[w] )
                     if ( dist[u] + length[u][w] < dist[w] )
                          dist[w] = dist[u] + length[u][w];
          }
   }
}
```

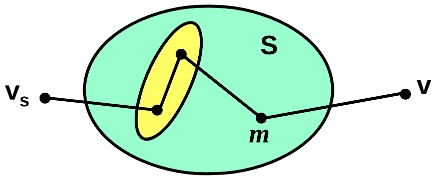


驛站區從 0 個擴大成 n-1 個

DP-Example 2: TSP Problem via DP

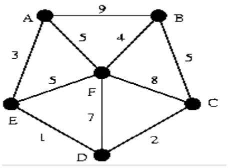
- **Given**
  - A graph $G(V, E)$ with edge weights w
- **Sub-Problem Formulation**
  - v_s: the starting vertex of the tour
  - C(S, v): the shortest tour length (v_s → v) passing through intermediate vertex set S
- **Construction Rule (problem size from k to k+1)**

$$C(S,v) = \min_{m \in S}\left[C(S-\{m\},m) + w(m,v)\right]$$



Every vertex in S
Can be the last vertex *m*

DP-Example 2: TSP Problem Via DP (Details)



$$C(S,v) = \min_{m \in S}\left[C(S-\{m\},m) + w(m,v)\right]$$

- **Problem size |S| = 0**
  - C(ϕ, B) = 9, C(ϕ, C) = ∞, C(ϕ, D) = ∞ , C(ϕ, E) = 3, C(ϕ, F) = 5
- **Problem Size |S| = 1, there are 20 computation, e.g.,**
  - C({B}, C) = C(ϕ, B) + w(B, C) = 9 + 5 = 14
  - C({B}, F) = C(ϕ, B) + w(B, F) = 9 + 4 = 13
  - C({F}, B) = C(ϕ, F) + w(F, B) = 5 + 4 = 9
- **Problem Size = 2, there are 30 computations, e.g.,**
  - C({B, F}, C) = min [$\underline{\text{C(\{B\}, F) + w(F, C)}}$, $\underline{\text{C(\{F\}, B) + w(B, C)}}$
    = min [13+8, 9+5] = 14
- **Final solution: C({ B, C, D, E, F}, A) = 18**

Linear Programming (LP)

- Given :
  - matrix $\mathbf{A}$ and vectors $\mathbf{b}$, $\mathbf{c}$
  - An unknown vector $\mathbf{x}$

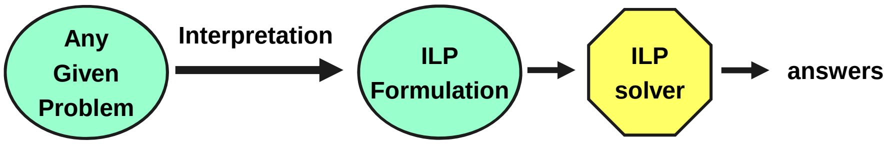
Canonical form:
Minimize or maximize: $\mathbf{c}^{\mathbf{T}}\mathbf{x}$
Subject to: $\mathbf{Ax} \leqq \mathbf{b}$ **and** $\mathbf{x} \geqq \mathbf{0}$

Standard form:
Minimize or maximize: $\mathbf{c}^{\mathbf{T}}\mathbf{x}$
Subject to: $\mathbf{Ax} = \mathbf{b}$ **and** $\mathbf{x} \geqq \mathbf{0}$

1. The two forms are interchangeably
*** (interception and/or adding dummy variables)**
2. Algorithms for solving LP
*** Polynomial, *ellipsoid***
*** Exponential in worst-case, *simplex***
*** In most cases, simplex is better than ellipsoid**

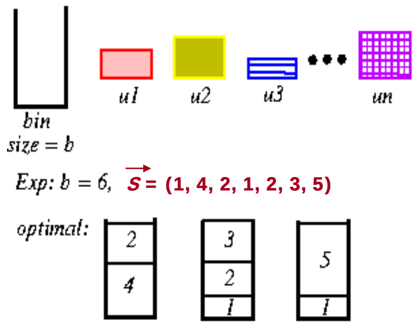
Integer Linear Programming

- **INTEGER LINEAR PROGRAMMING (ILP)**
  - **ILP is a special form of linear programming**
  - **Each variable takes on an integer value**
  - **ILP is very common for solving combinatorial optimization**
  - **ILP is NP-complete**
  - **ILP-solvers are available at public domain**
- **Applications in VLSI Design Automation**
  - **Often takes a special form: zero-one ILP**
  - **(1) Exact solutions for problems with small input sizes**
  - **(2) To know how good the other heuristics are**
  - **(3) As a source of inspiration for developing new heuristics**

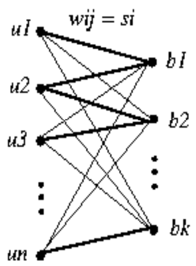


Example: Bin Packing

- **The Bin-Packing Problem :**
  - Items $U = \{u_1, u_2, \ldots, u_n\}$, where u_i has a size denoted as s_i
  - A set of bins, each with a capacity denoted as b
- **Goal:**
  - Pack all items, minimizing # of bins used. (NP-hard!)

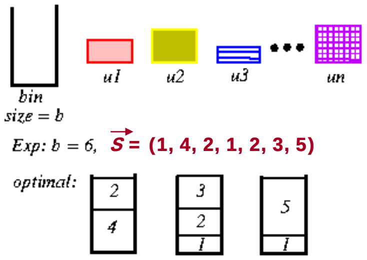


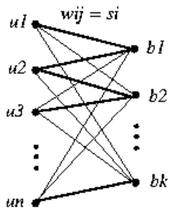
An edge means assignment



A bipartite graph

Algorithms for Bin Packing





- **Greedy approximation algorithm**
  - First-Fit Decreasing (FFD)
- **Dynamic Programming? Hierarchical Approach? Genetic Algorithm? …**
- **Mathematical Programming: Use integer linear programming (ILP) to find a solution using |*B*| bins, then search for the smallest feasible |*B*|.**

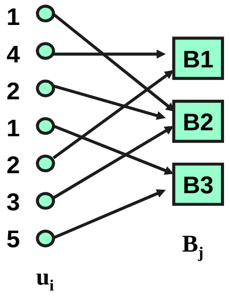
ILP Formulation

- **Variables**
  - 0-1 variable x_{ij}=1 if item u_i is placed in bin b_j, 0 otherwise
  - There are n items, $\{u_1, u_2, \ldots, u_n\}$, and the size of u_i is w_i
  - There are |B| bins, $\{B_1, B_2, \ldots, B_k\}$, each having capacity of b
- **ILP Formulation**
  - Three types of constraints for feasible solutions

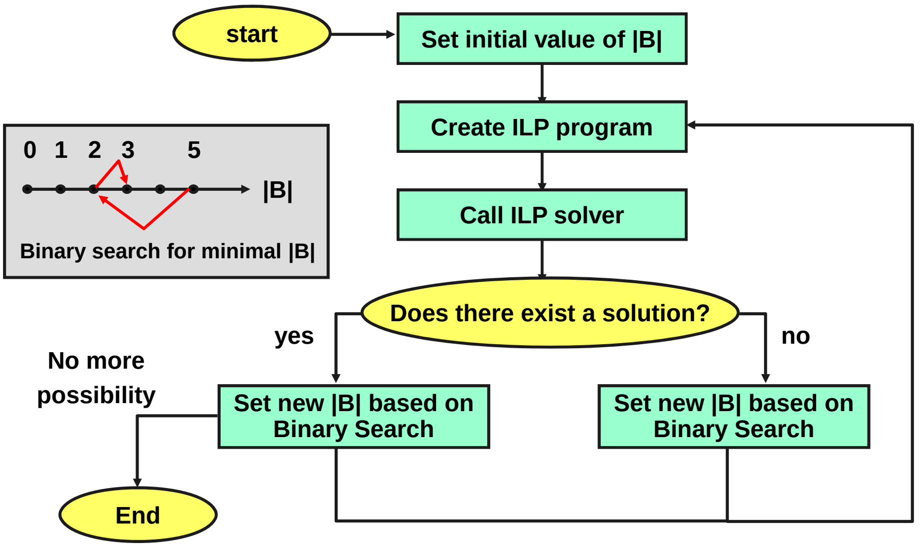
(1) 0-1 variables $x_{ij} \in \{0,1\}$

(2) Exactly once assignment $\sum_{j=1}^{B} x_{ij} = 1$

(3) Bin capacity constraint $\sum_{i=1}^{n} w_i x_{ij} \le b$



Flow of Solving Bin Packing By ILP



start

Set initial value of |B|

Create ILP program

Call ILP solver

Does there exist a solution?

yes

no

Set new |B| based on Binary Search

Set new |B| based on Binary Search

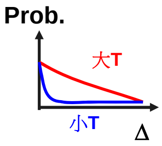
No more possibility

End

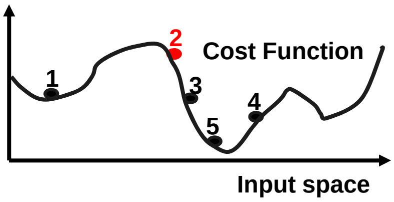
Binary search for minimal |B|

Simulated Annealing

- **Inspiration**
  - The material's slow cooling-down process
  - (1) Initially, molecules are free to move like liquid
  - (2) Gradually, they lose their energy and take a fixed position
  - Also called statistical cooling
- **Analogy**
  - Energy ➔ cost function
  - Molecule movement ➔ movement in search space
  - Temperature ➔ Control parameter T



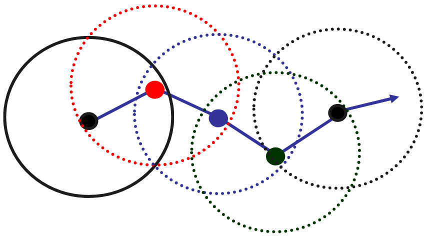
Δ : cost increase of a move



Moving Strategy:
(1) Cost Down: Always Accept
(2) Cost Up: Accept with probability $e^{\frac{-\Delta}{T}}$

Tabu Search

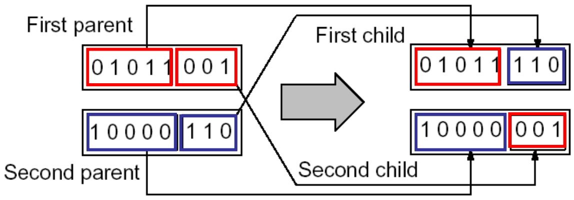
- **Principles:**
  - Moves to the cheapest neighbor, even when cost increases
- **Side Effect**
  - There will be risk of cycling
- **Cycle Prevention**
  - Store the last *k* feasible solutions in a tabu list
  - Any attempts getting into the tabu list is rejected



Genetic Algorithms

- **Principles:**
  - Based on analogy with evolution process in nature
  - Optimization based on survival-of-the-fittest principle
- **Major elements**
  - (1) A population of feasible solutions
  - (2) Encoding of each solution, called chromosome
  - (3) Crossover: to produce children (or offsprings)
  - (4) Mutation (突變): to escape from local minimum

Illustration of Crossover



Concluding Remarks

- NP-Hard problems are everywhere in EDA
  - Cannot be exactly solved
  - However, many good heuristics still lead to good results
- When it comes to search problems …
  - Numerous paradigms exist
  - Each has its own proper application domain
  - A tug-of-war between space and time

Problem ➔ Formulation ➔ Algorithms
A problem is half solved when it is clearly formulated.

清華大學 EE 5265
積體電路設計自動化

單元 3
Logic Synthesis

教育部顧問室
「超大型積體電路與系統設計」教育改進計畫
EDA聯盟 – 推廣課程

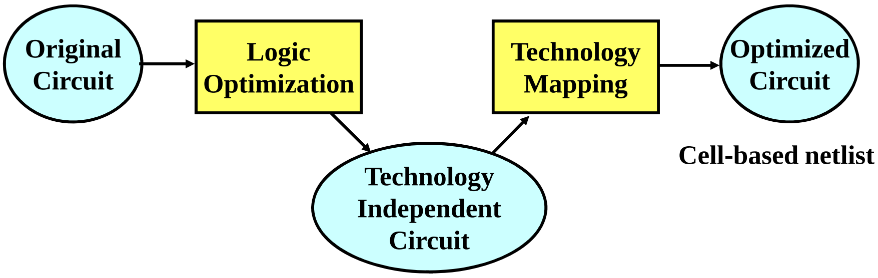
致謝

本單元之教材主要取自於
教育部
超大型積體電路與系統設計教育改進計畫
EDA聯盟之課程發展成果

- **(教材編纂小組成員)**
- 台灣大學電機系 張耀文
- 清華大學電機系 黃錫瑜
- 交通大學資科系 李毅郎
- 中央大學電機系 劉建男
- 元智大學資工系 林榮彬

Outline

- RTL Synthesis
- Logic Optimization
- Technology Mapping


Original Circuit

Logic Optimization

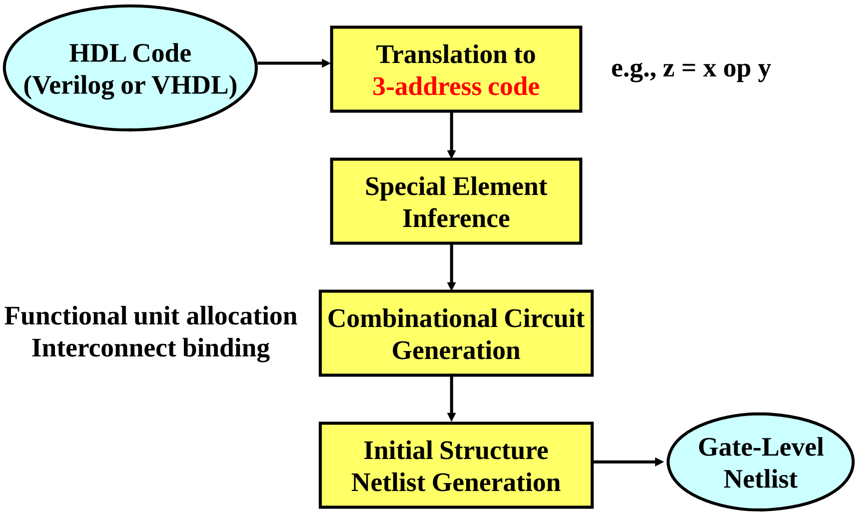
Technology Independent Circuit

Technology Mapping

Optimized Circuit

Cell-based netlist

RTL Synthesis Flow


HDL Code (Verilog or VHDL)

Translation to 3-address code

e.g., z = x op y

Special Element Inference

Functional unit allocation
Interconnect binding

Combinational Circuit Generation

Initial Structure Netlist Generation

Gate-Level Netlist

Example of RTL Synthesis

RTL code segment

```
x=c+d+e;
if(a==b) x= e-f;
y=x;
```

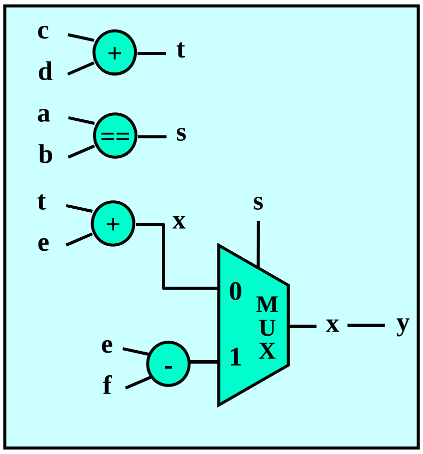
➡

3-address code

```
t=c+d;
x=t+e;
s = (a==b);
if(s) x= e-f;
y=x;
```

➡

Circuit Component



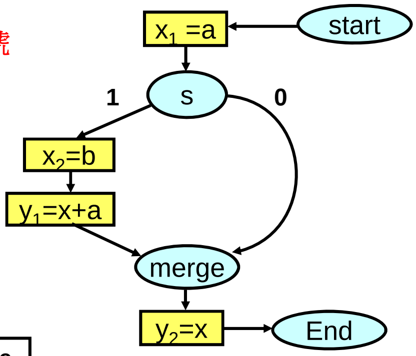
Control Data Flow Graph

- CDFG (Control/Data Flow Graph)
  - A representation for the cycle behavior of an RTL code
  - Nodes: operation, decision, or merge point
  - Edges: signal flow
  - Used to resolve the data and control dependency

一個變數可能出現多次 ➔ 需加以編號

```
/*d1*/  x = a;
         if(s) begin
/*d2*/      x = b;
/*d3*/      y = x + a;
         end
/*d4*/  y = x;
```

➡

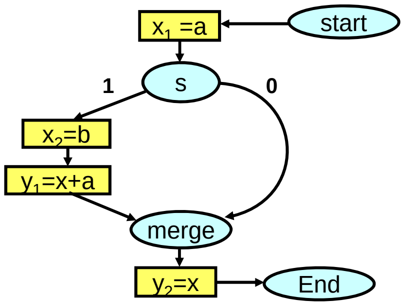


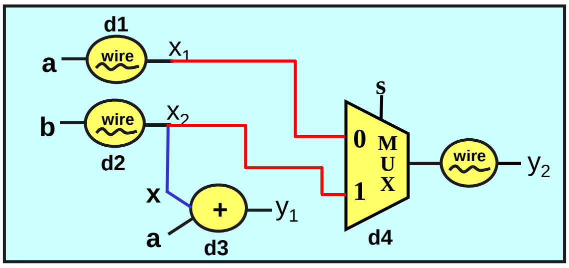
CDFG is used to decide where the input operands should come from.

y=b or a, depending on the value of s

Component Binding

```
/*d1*/  x = a;
        if(s) begin
/*d2*/      x = b;
/*d3*/      y = x + a;
        end
/*d4*/  y = x;
```





Special Element Inferences

- **Three special elements to be inferred**
  - **Latch (D-type) inference**
  - **Flip-Flop (D-type) inference**
  - **Tri-state buffer inference**

```
reg Q;
always@(D or en)
  if(en) Q = D;
```

Latch inferred!!

```
reg Q;
always@(posedge clk)
   Q = D;
```

Flip-flop inferred!!

```
reg Q;
always@(D or en)
  if(en) Q = D;
  else    Q = 1’bz;
```

Tri-state buffer inferred!!

Sequential Section vs. Combinational Section

- Sequential section
  - "Always statement" triggered by clock edges
- Combinational section
  - All signals whose values are used in the "always statement" are included in the sensitivity list

```
reg Q;
always@(posedge clk)
   Q = D;
```

Sequential section
Conduct flip-flop inference

```
reg Q;
always@(in or en)
  if(en) Q=in;
```

Combinational section
Conduct latch inference
(in dangling if-the-else)

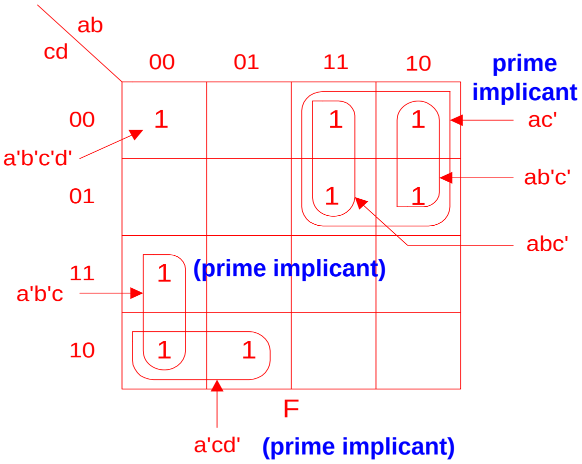
Outline

- RTL Synthesis
- **Logic Optimization**
  - Two-Level Logic Minimization
  - Multi-Level Logic Minimization
- Technology Mapping

What can we do beyond Karnaugh Map [1953] and Quine-McCluskey [1956] ?

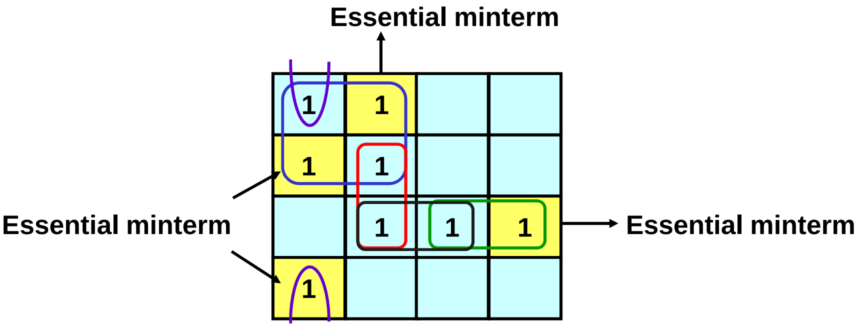
Prime Implicant

Review: implicant and prime implicant



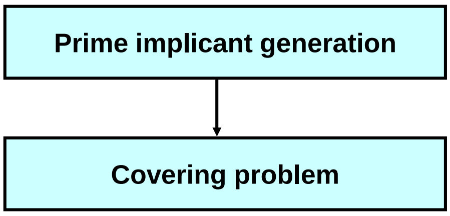
Essential Prime Implicant

- **Essential Minterm**
  - Is a minterm <u>covered by only one prime implicant</u>
- **Essential Prime Implicant**
  - Is a prime implicant that contains at least one essential minterms

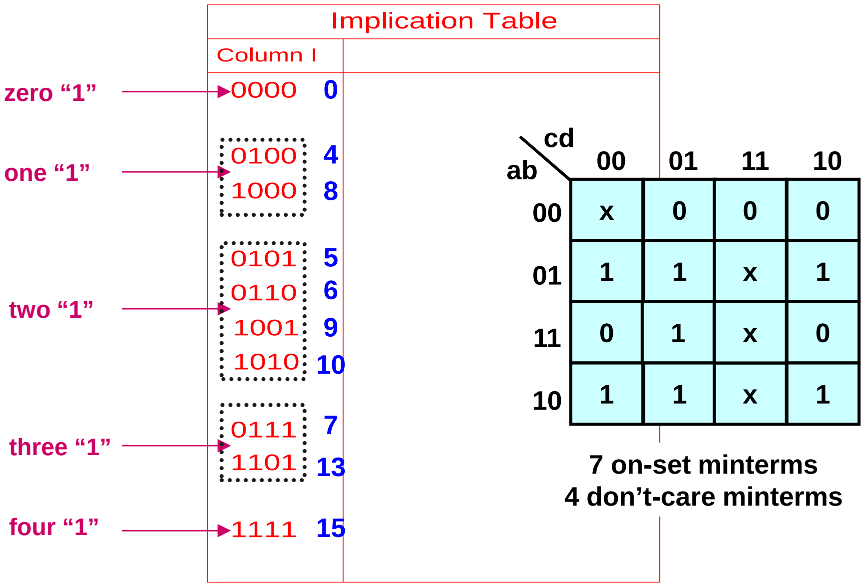


Classical Logic Minimization

- Theorem:[Quine, McCluskey]
  - There exists a minimum cover for F that is prime
  - We only need to look at just primes (to reduce the search space)
- Classical methods: using a two-step process
  1. Generate all prime implicants
  2. Find a minimum cover (covering problem)
     - A cover is a set of primes that covers every on-set minterm

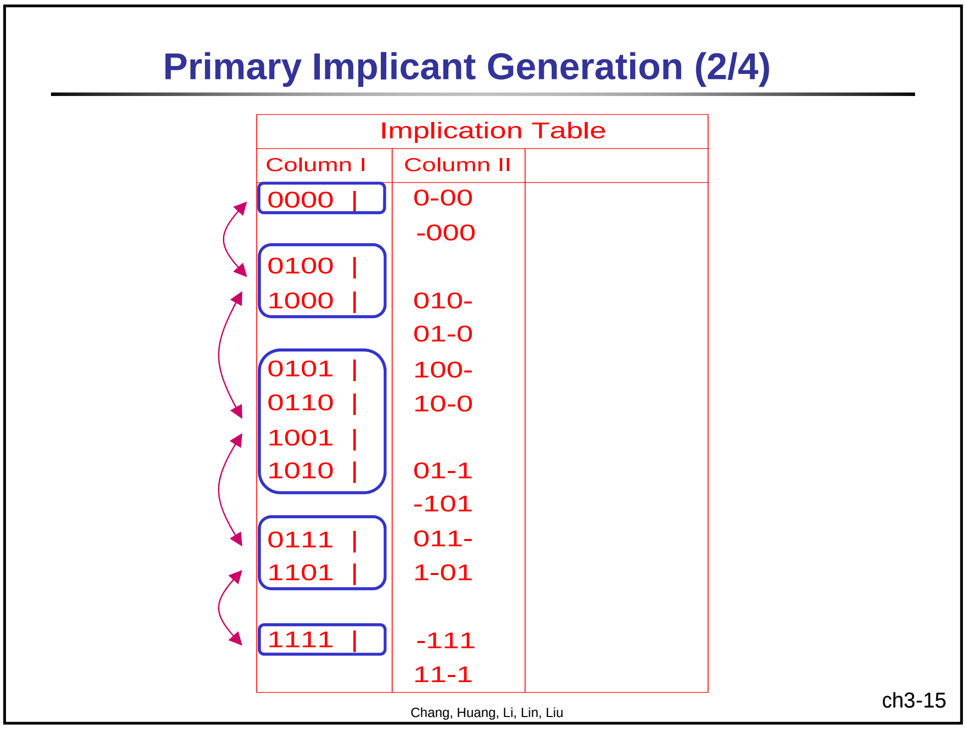


Primary Implicant Generation (1/4)



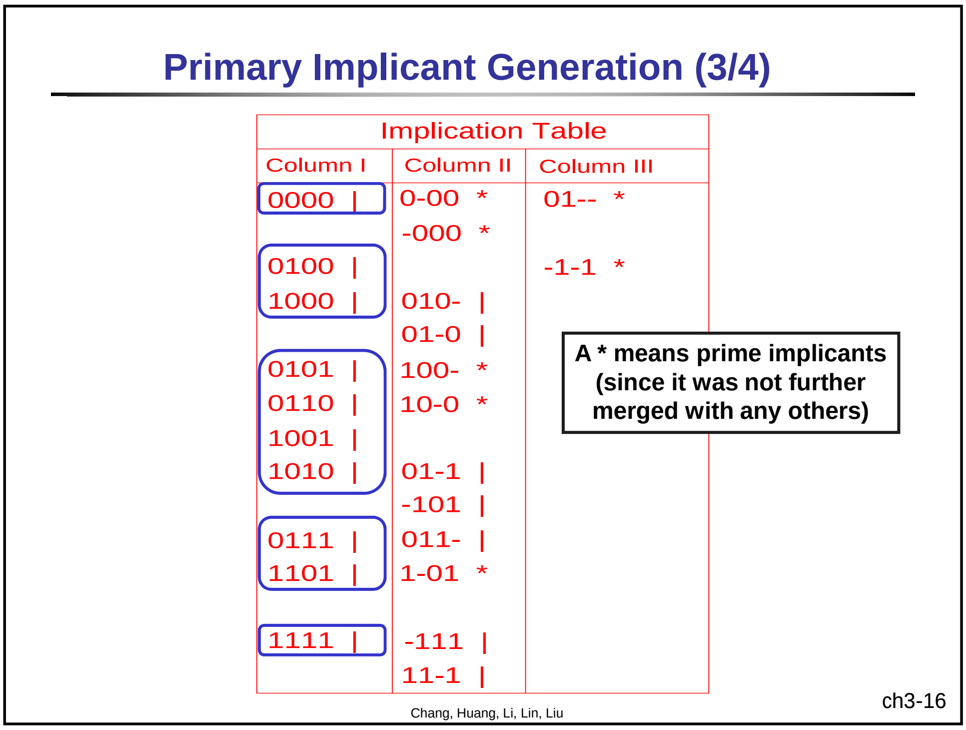
ab \ cd	00	01	11	10
00	x	0	0	0
01	1	1	x	1
11	0	1	x	0
10	1	1	x	1

7 on-set minterms
4 don't-care minterms



Primary Implicant Generation (2/4)

Implication Table		
Column I	Column II	
0000 |	0-00	
	-000	
0100 |		
1000 |	010-	
	01-0	
0101 |	100-	
0110 |	10-0	
1001 |		
1010 |	01-1	
	-101	
0111 |	011-	
1101 |	1-01	
1111 |	-111	
	11-1	



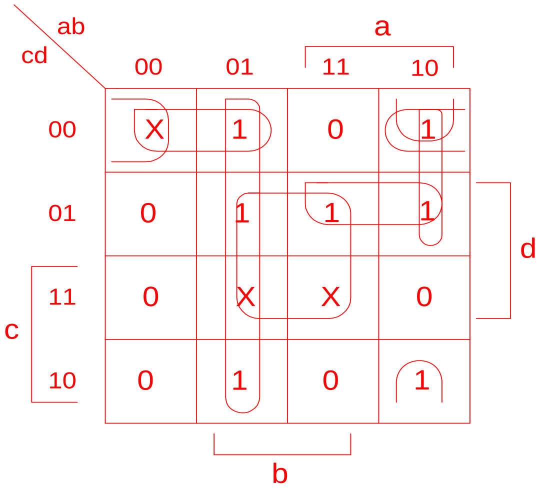
Primary Implicant Generation (3/4)

Implication Table		
Column I	Column II	Column III
0000 |	0-00 *	01-- *
	-000 *	
0100 |		-1-1 *
1000 |	010- |	
	01-0 |	
0101 |	100- *	
0110 |	10-0 *	
1001 |		
1010 |	01-1 |	
	-101 |	
0111 |	011- |	
1101 |	1-01 *	
1111 |	-111 |	
	11-1 |	

A * means prime implicants (since it was not further merged with any others)

Primary Implicant Generation (4/4)

There are seven prime implicants in this case



cd \ ab	00	01	11	10
00	X	1	0	1
01	0	1	1	1
11	0	X	X	0
10	0	1	0	1

Prime Implicants:

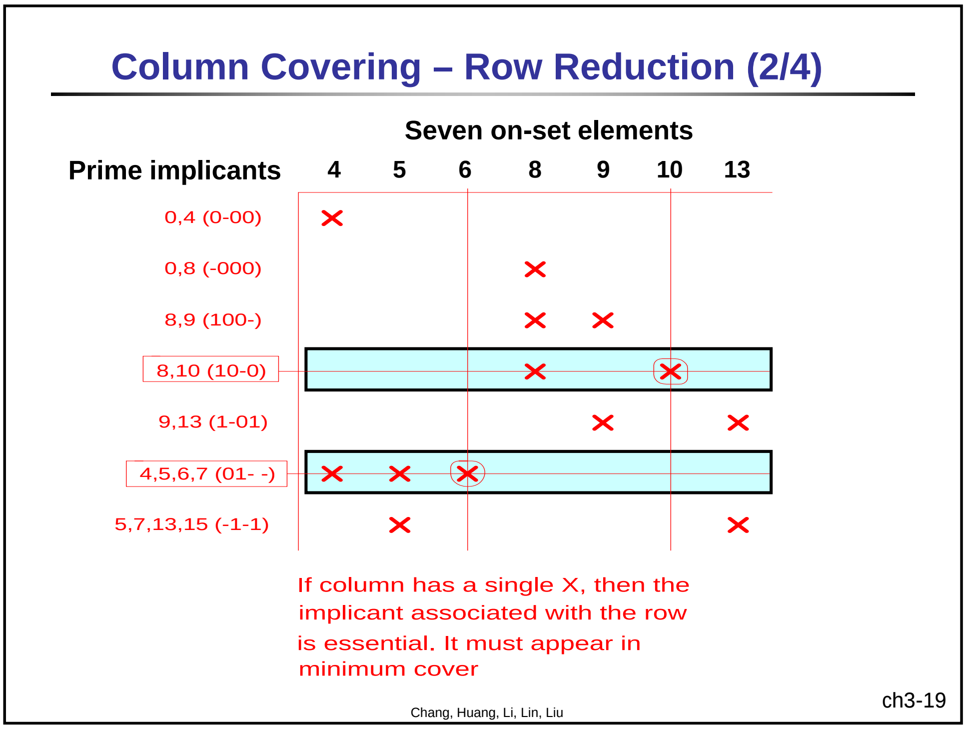
0-00 = a'c'd'
100- = ab'c'
1-01 = ac'd
-1-1 = bd
-000 = b'c'd'
10-0 = ab'd'
01-- = a'b

Column Covering (1/4)

Prime implicants	Seven on-set elements						
	4	5	6	8	9	10	13
0,4 (0-00)	×						
0,8 (-000)				×			
8,9 (100-)				×	×		
8,10 (10-0)				×		×	
9,13 (1-01)					×		×
4,5,6,7 (01- -)	×	×	×				
5,7,13,15 (-1-1)		×					×

Note: minterms 0, 7, 11, 15 are don't-care terms are thus not shown in the table.

rows = prime implicants
columns = ON-set elements
place an "X" if ON-set element is covered by the prime implicant

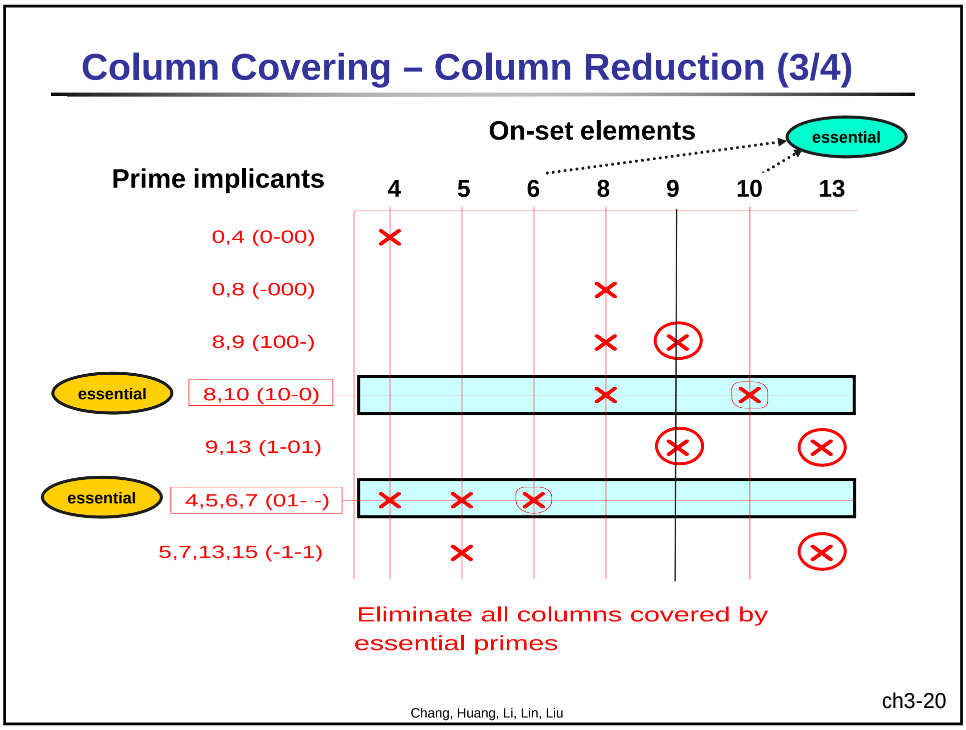


Column Covering – Row Reduction (2/4)

Seven on-set elements

Prime implicants	4	5	6	8	9	10	13
0,4 (0-00)	✕						
0,8 (-000)				✕			
8,9 (100-)				✕	✕		
8,10 (10-0)				✕		⊗	
9,13 (1-01)					✕		✕
4,5,6,7 (01- -)	✕	✕	⊗				
5,7,13,15 (-1-1)		✕					✕

If column has a single X, then the implicant associated with the row is essential. It must appear in minimum cover



Column Covering – Column Reduction (3/4)

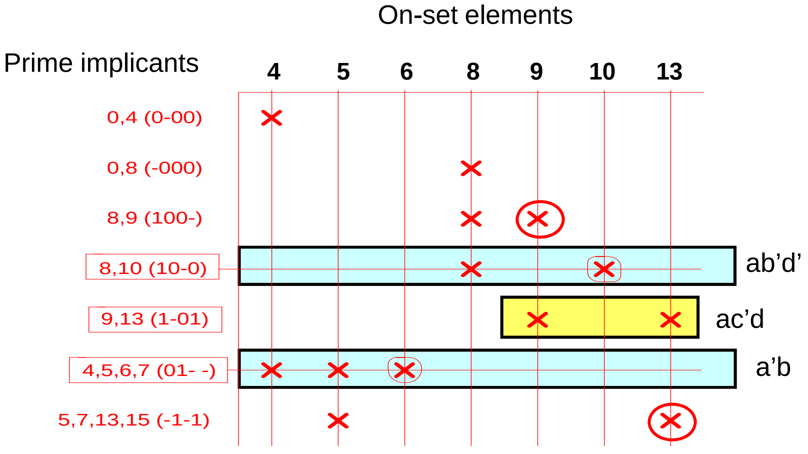
On-set elements

essential

	Prime implicants	4	5	6	8	9	10	13
	0,4 (0-00)	✕						
	0,8 (-000)				✕			
	8,9 (100-)				✕	⊗		
essential	8,10 (10-0)				✕		⊗	
	9,13 (1-01)					⊗		⊗
essential	4,5,6,7 (01- -)	✕	✕	⊗				
	5,7,13,15 (-1-1)		✕					⊗

Eliminate all columns covered by essential primes

Column Covering – Min. Cover (4/4)



On-set elements

Prime implicants	4	5	6	8	9	10	13
0,4 (0-00)	✕						
0,8 (-000)				✕			
8,9 (100-)				✕	✕		
8,10 (10-0)				✕		✕	
9,13 (1-01)					✕		✕
4,5,6,7 (01- -)	✕	✕	✕				
5,7,13,15 (-1-1)		✕					✕

Find minimum set of rows that cover the remaining columns

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

Petrick’s Method

- **Solve the Satisfiability problem of the following function**

P = (P1+P6)(P6+P7)P6(P2+P3+P4)(P3+P5)P4(P5+P7)=1

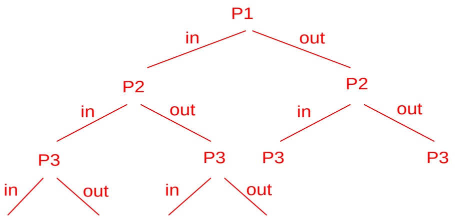
- **Each clause represents a corresponding column**
- **Each column must be chosen at least once**
- **All columns must be covered**

(prime implicants)

		4	5	6	8	9	10	13
P1	0,4 (0-00)	✕						
P2	0,8 (-000)				✕			
P3	8,9 (100-)				✕	✕		
P4	8,10 (10-0)				✕		✕	
P5	9,13 (1-01)					✕		✕
P6	4,5,6,7 (01- -)	✕	✕	✕				
P7	5,7,13,15 (-1-1)		✕					✕

Brute Force Technique

- **Brute force technique: Consider all possible elements**



- **Complete tree has $2^{|P|}$ leaves!!**
  - **Need to prune it**
- **Complexity reduction**
  - **Essential primes can be included right away**
    - **If there is a row with a singleton "1" for the column**
  - **Keep track of best solution seen so far**
    - **Classical *branch and bound***

Heuristic Optimization

- **Generation of *all* prime implicants is impractical**
- **Finding an *exact* minimum cover is NP-hard**
  - **Cannot be finished in polynomial time**
- **Expansion-Based Heuristic:**
  - **Avoid generation of all prime implicants**
- **Procedure**
  - **(Step 1): An on-set minterm is selected, and expanded until it becomes a prime implicant**
  - **(Step 2): The prime implicant is included in the final cover, and all minterms covered by this prime implicant are removed**
  - **(Step 3): Iterate until all minterms of the ON(f) are covered**
- **"ESPRESSO" developed by UC Berkeley**
  - **The kernel of synthesis tools**

Outline

- **RTL Synthesis**
- **Logic Optimization**
  - **Two-Level Logic Minimization**
  - **Multi-Level Logic Minimization**
- **Technology Mapping**

Multi-Level v.s. Two-Level

- **Two-level:**
  - **Often used in control**

    $f_1 = x_1x_2 + x_1x_3 + x_1x_4$

    $f_2 = x_1'x_2 + x_1'x_3 + x_1x_4$
  - **Only x_1x_4 shared**
  - **Sharing restricted to common cube**
- **Multi-level:**
  - **Datapath or control**
  - **Can share $x_2 + x_3$ between the two expressions**
  - **Can use complex gates**

    $g_1 = x_2 + x_3$

    $g_2 = x_1x_4$

    Common sub-functions

    $f_1 = x_1g_1 + g_2$

    $f_2 = x_1'g_1 + g_2$

如何發現 **common sub-functions** 需要某種型式的【因式分解】

Minimization via Division

- **Goal:**
  - Reduce the no. of literals in a given Boolean formula
- **Two problems:**
  - (1) Find good common sub-functions !
  - (2) How to perform division ?

(Minimization Via Division)
F: a Boolean function in SOP form
P: a good sub-function (kernel)

$$F = PQ + r$$

Quotient function

Remainder function

Example:
F = ac + ad + bc + bd + e
F = (a+b)(c+d) + e
Literal count: 9 ➔ 5

Terminology: Primary Divisors

- Cube-Free Expression
  - An expression is cube-free if no cube divides the expression
  - E.g., ab + c is cube-free
  - E.g., ab + ac = a (b + c) is not cube-free
  - A cube-free expression must have more than one cube
  - E.g., *abc* is not cube-free
- Primary Divisors
  - The set of primary divisors of an expression f is defined as:
    D(f) = {f/c | c is a cube}
  - We are more interested in finding cube-free divisors

Terminology: Kernels and Co-Kernels

- **Kernel**
  - The set of kernels of an expression f is defined as cube-free primary divisors, I.e.,

    K(f) = {g | g$\in$ D(f) and g is cube-free}
- **Co-kernel**
  - The cubes used to obtain the kernels are co-kernels, C(f)

$$f = K(f) \cdot C(f) + r$$

Kernel:
Cube-free primary divisor f/c

Co-Kernels:
Cube c

Example: Factorization by Kernels

- Example:

  $f = x_1x_2x_3 + x_1x_2x_4 + x_3'x_2$ ➔ 8 literals

  $K = \{\underline{x_1x_3 + x_1x_4 + x_3'}, \underline{x_3 + x_4}\}$
  - x_2 is the co-kernel for kernel $(x_1x_3 + x_1x_4 + x_3')$
  - x_1x_2 is the co-kernel for kernel $x_3 + x_4$
- Kernels can be used to factorize an expression

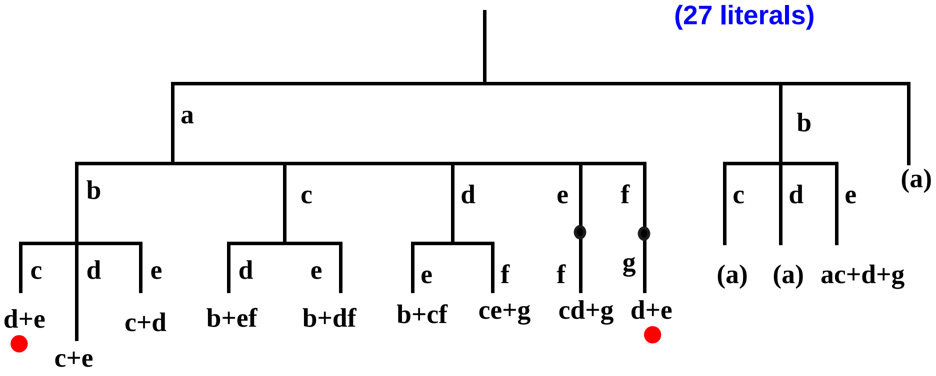
  $f = (x_3 + x_4)(x_1x_2) + x_3'x_2 = x_2(x_1(x_3 + x_4) + x_3')$

  ➔ 5 literals
- For multiple-function minimization
  - It is key to find *common divisors* between expressions

Find Out All Kernels (1/2)

abcd + abce + adfg + aefg + adbe + acdef + beg

(27 literals)



➔abc (d+e) + afg(d+e) + (adbe + acdef + beg)
➔ (abc+afg)(d+e) + (adbe + acdef + beg)
➔ a(bc+fg)(d+e) + [ade(b + cf) + beg]   (16 literals)

Find Out All Kernels (2/2)

co-kernel	kernel
1	a((bc + fg)(d + e) + de(b + cf))) + beg
a	(bc + fg)(d + e) + de(b + cf)
ab	c(d + e) + de
abc	d + e
.	.
ac	b(d + e) + def
acd	b + ef
.	.
bc	ad + ae

They can be obtained in n^2 time
where *n* is number of cubes in this expression.

Common Divisor

- **Theorem (Brayton & McMullen):**

  f **and** ***g*** **have a multiple-cube common divisor if and only if the intersection of a kernel of** ***f*** **and a kernel of** ***g*** **has more than one cube**

$f_1 = x_1(x_2x_3 + x_2'x_4) + x_5$

$f_2 = x_1(x_2x_3 + x_2'x_5) + x_4$

$K(f_1) = \{x_2x_3 + x_2'x_4,$

$\quad x_1(x_2x_3 + x_2'x_4) + x_5\}$

$K(f_2) = \{x_2x_3 + x_2'x_5,$

$\quad x_1(x_2x_3 + x_2'x_5) + x_4\}$

$K_1 \cap K_2 = \{x_2x_3, x_1x_2x_3\}$

- **f_1 and f_2 have no multiple-cube common divisor**

$f_1 = x_1x_2 + x_3x_4 + x_5$

$f_2 = x_1x_2 + x_3'x_4 + x_5$

$K(f_1) = \{ x_1x_2 + x_3x_4 + x_5\}$

$K(f_2) = \{ x_1x_2 + x_3'x_4 + x_5\}$

$K_1 \cap K_2 = \{ x_1x_2 + x_5\}$

- **f_1 and f_2 have multiple-cube common divisor**

Cube-Literal Matrix

- **Cube-literal matrix**

$f = x_1x_2x_3x_4x_7 + x_1x_2x_3x_4x_8 + x_1x_2x_3x_5 + x_1x_2x_3x_6 + x_1x_2x_9$

		Literals								
		x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
Cubes	$x_1x_2x_3x_4x_7$	1	1	1	1	0	0	1	0	0
	$x_1x_2x_3x_4x_8$	1	1	1	1	0	0	0	1	0
	$x_1x_2x_3x_5$	1	1	1	0	1	0	0	0	0
	$x_1x_2x_3x_6$	1	1	1	0	0	1	0	0	0
	$x_1x_2x_9$	1	1	0	0	0	0	0	0	1

Cube-Literal Matrix & Rectangles

- **A Rectangle (R, C)** of a matrix A
  - R is a subset of rows, and C is a subset of columns, such that

    $A_{ij} = 1$, for all $i \in R$, $j \in C$
  - Rows and columns need not be continuous
- **A Prime Rectangle**
  - Is a rectangle not contained in any other rectangle
  - A prime rectangle indicates a co-kernel kernel pair

Rectangle = {R, C} = {{1, 2, 3, 4},{1, 2, 3}}
➔ co-kernel: $x_1x_2x_3$, kernel: $x_4x_7 + x_4x_8 + x_5 + x_6$

Cubes

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
$x_1x_2x_3x_4x_7$	1	1	1	1	0	0	1	0	0
$x_1x_2x_3x_4x_8$	1	1	1	1	0	0	0	1	0
$x_1x_2x_3x_5$	1	1	1	0	1	0	0	0	0
$x_1x_2x_3x_6$	1	1	1	0	0	1	0	0	0
$x_1x_2x_9$	1	1	0	0	0	0	0	0	1

Prime Rectangles and Logic Synthesis

- **Given functions**

  F = abc + abd + eg
  G = abfg
  H = bd + ef
- **Prime Rectangles**

  ({1,2,4},{1,2}) ➔ c_1 = ab
  ({2,5},{2,4}) ➔ c_2 = bd

➡

(Co-kernels)

X = ab, Y = bd

(Minimized Functions)

F = Xc + XY + eg
G = Xfg
H = Y + ef

		a	b	c	d	e	f	g
		1	2	3	4	5	6	7
abc	1	1	1	1	0	0	0	0
abd	2	1	1	0	1	0	0	0
eg	3	0	0	0	0	1	0	1
abfg	4	1	1	0	0	0	1	1
bd	5	0	1	0	1	0	0	0
ef	6	0	0	0	0	1	1	0

Outline

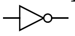
- RTL Synthesis
- Logic Optimization
- Technology Mapping

Technology Mapping

- **General approach:**
  - Choose a set of base functions for canonical representation
    - Ex: 2-input NAND and Inverter
  - Represent optimized Boolean network using base functions
    - Subject graph (for entire Boolean network)
  - Represent each library cell using base functions
    - One Pattern graph for one library cell
  - Each pattern is associated with a cost which is dependent on the optimization criteria
- **Goal:**
  - Finding a minimum-cost cover for a subject graph (I.e., a Boolean network) using pattern graphs (I.e., cells)

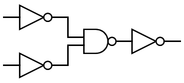
Example Pattern Graph (1/3)

inv (1)

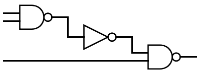


nand2 (1)

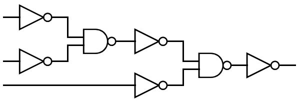
nor2 (2)



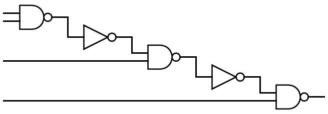
nand3 (3)



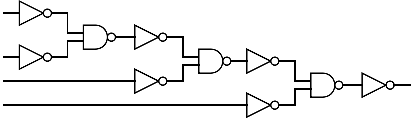
nor3 (3)



nand4 (4)

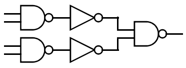


nor4 (4)

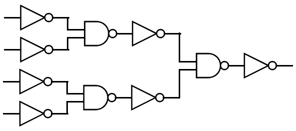


Example Pattern Graph (2/3)

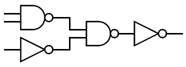
nand4 (4)



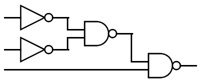
nor4 (4)



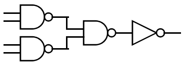
aoi21 (3)



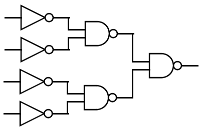
oai21 (3)



aoi22 (4)

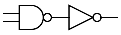


oai22 (4)

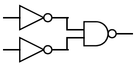


Example Pattern Graph (3/3)

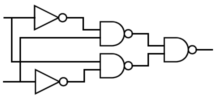
and2 (3)



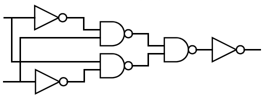
or2 (3)



xor (5)

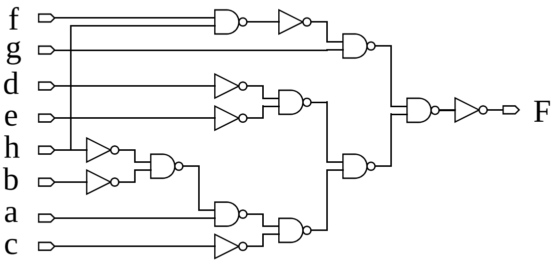


xnor (5)

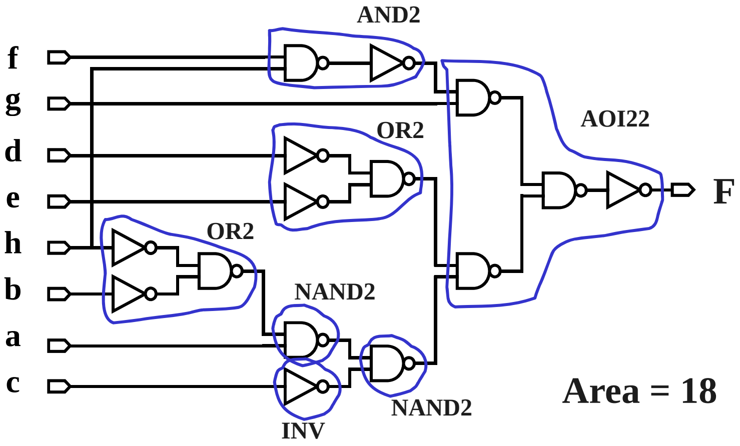


Example Subject Graph

t1 = d + e;
t2 = b + h;
t3 = a t2 + c;
t4 = t1 t3 + f g h;
F = t4';

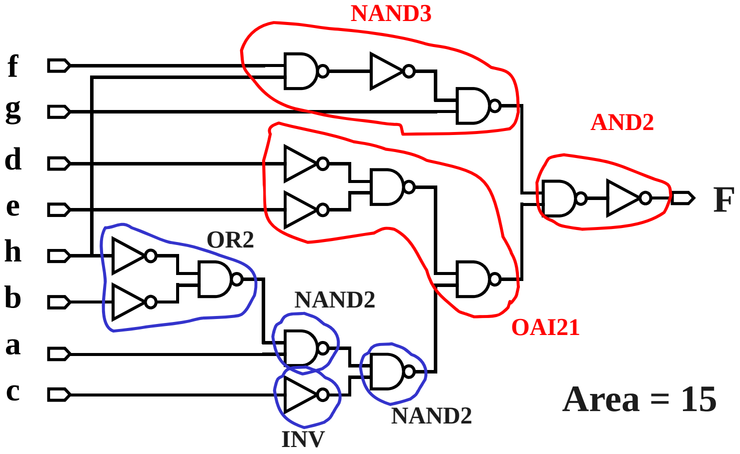


Sample Covers (1/2)



Area = 18

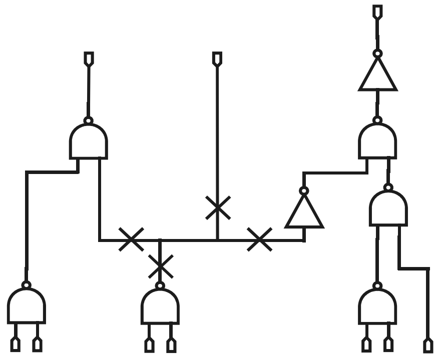
Sample Covers (2/2)



Area = 15

DAGON Approach

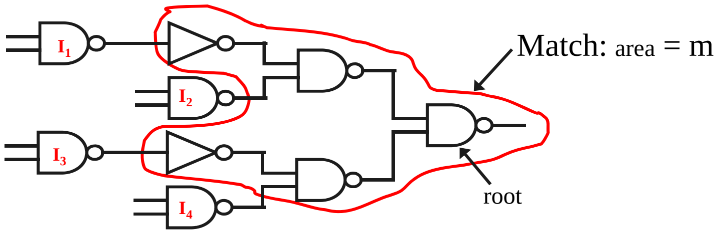
- **Partition a subject graph into trees**
  - Cut the graph at all multiple fanout points



- **Optimally cover each tree using dynamic programming approach**
- **Piece the tree-covers into a cover for the subject graph**

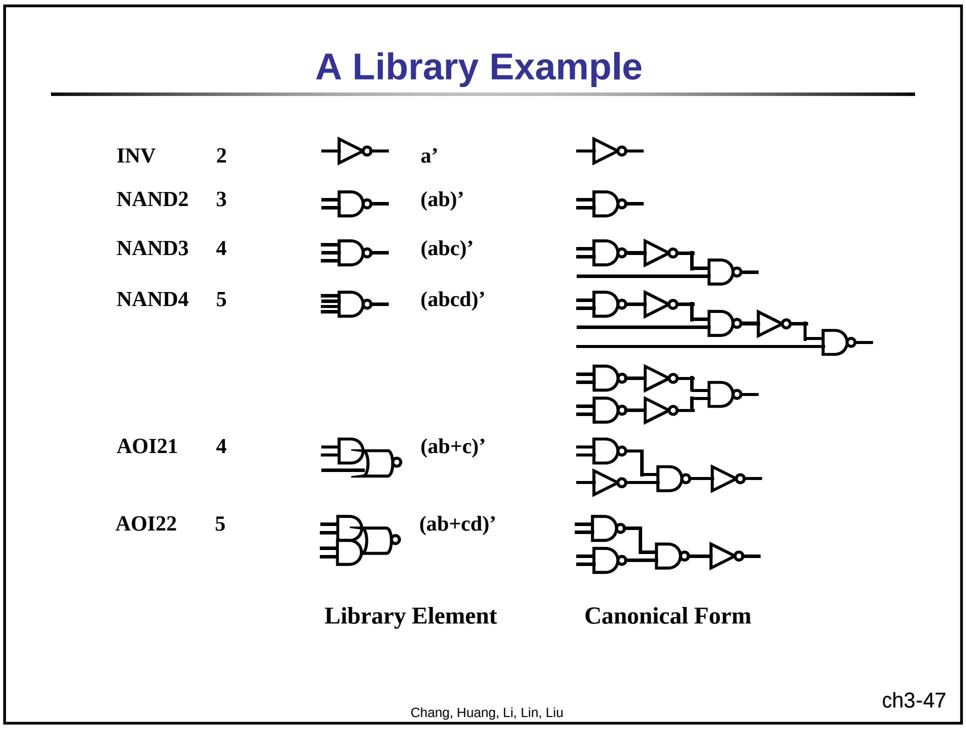
Dynamic Programming for Minimum Area

- **Principle of optimality:**
  - Optimal cover for the tree consists of a match at the root plus the optimal cover for the sub-tree starting at each input of the match



$$A(root) = m + A(I_1) + A(I_2) + A(I_3) + A(I_4)$$

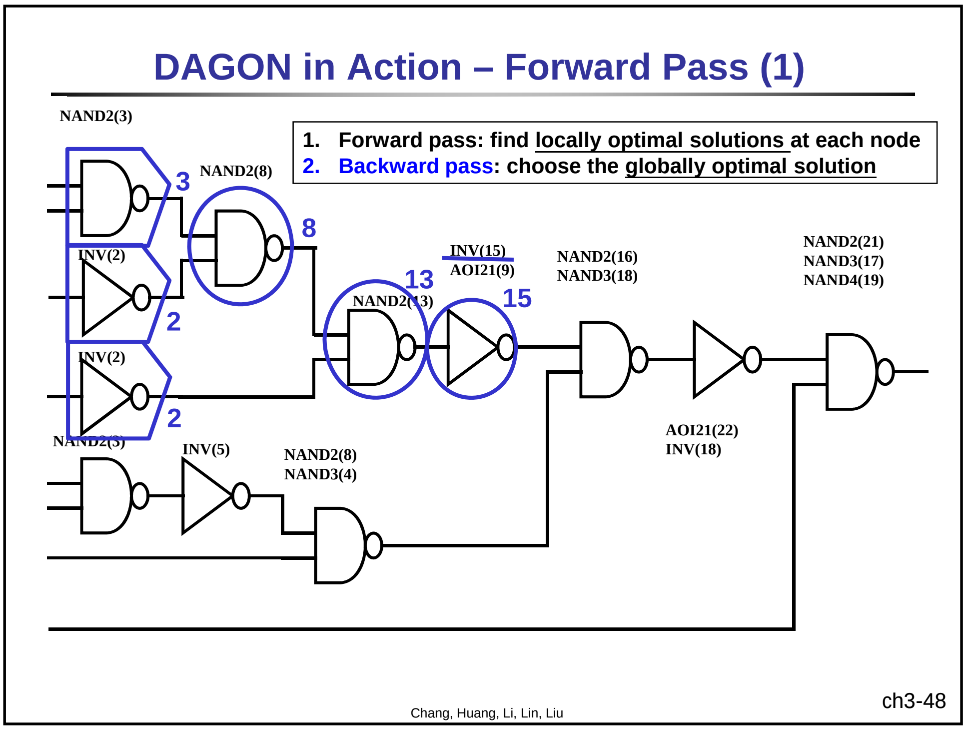
cost of a leaf = 0



A Library Example

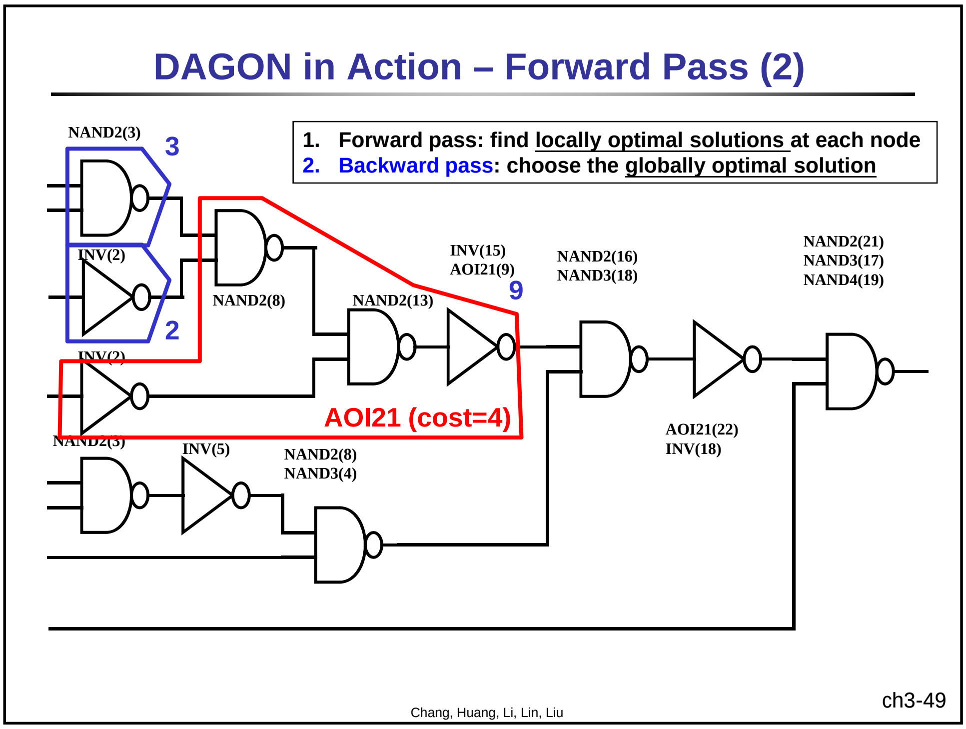
INV	2	a’	
NAND2	3	(ab)’	
NAND3	4	(abc)’	
NAND4	5	(abcd)’	
AOI21	4	(ab+c)’	
AOI22	5	(ab+cd)’	

Library Element **Canonical Form**



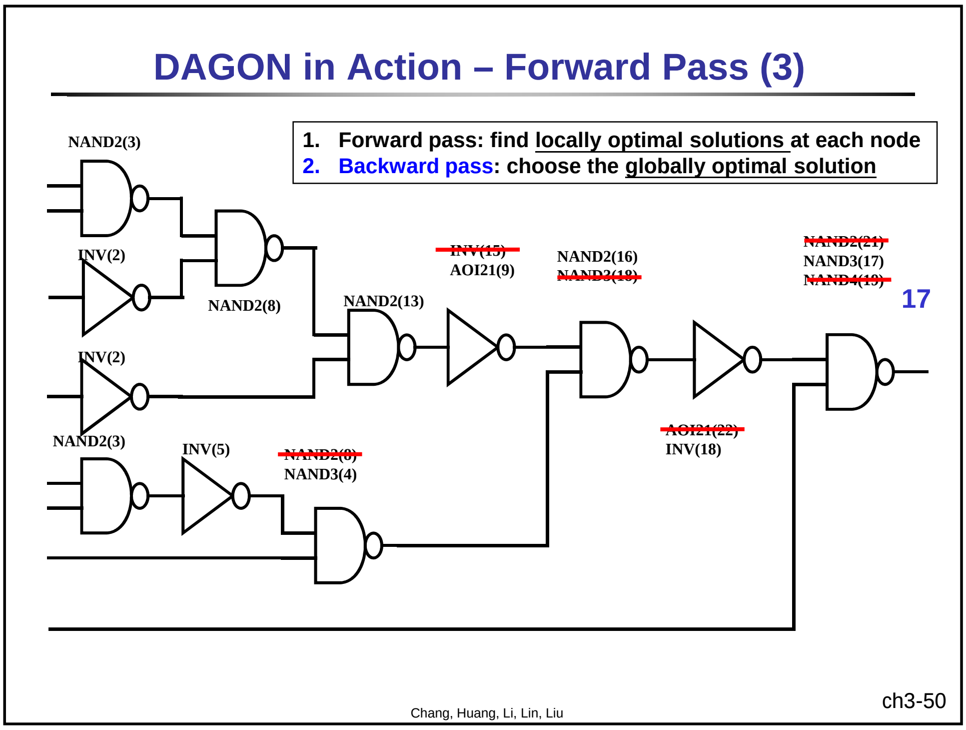
DAGON in Action – Forward Pass (1)

1. Forward pass: find locally optimal solutions at each node
2. Backward pass: choose the globally optimal solution



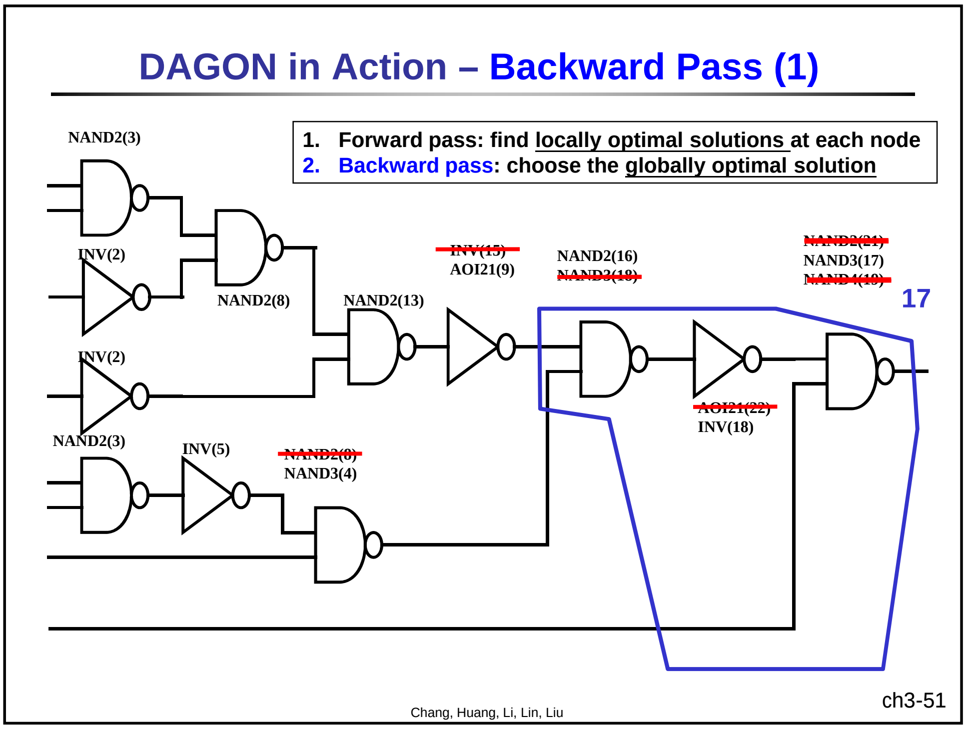
DAGON in Action – Forward Pass (2)

1. Forward pass: find locally optimal solutions at each node
2. **Backward pass**: choose the globally optimal solution



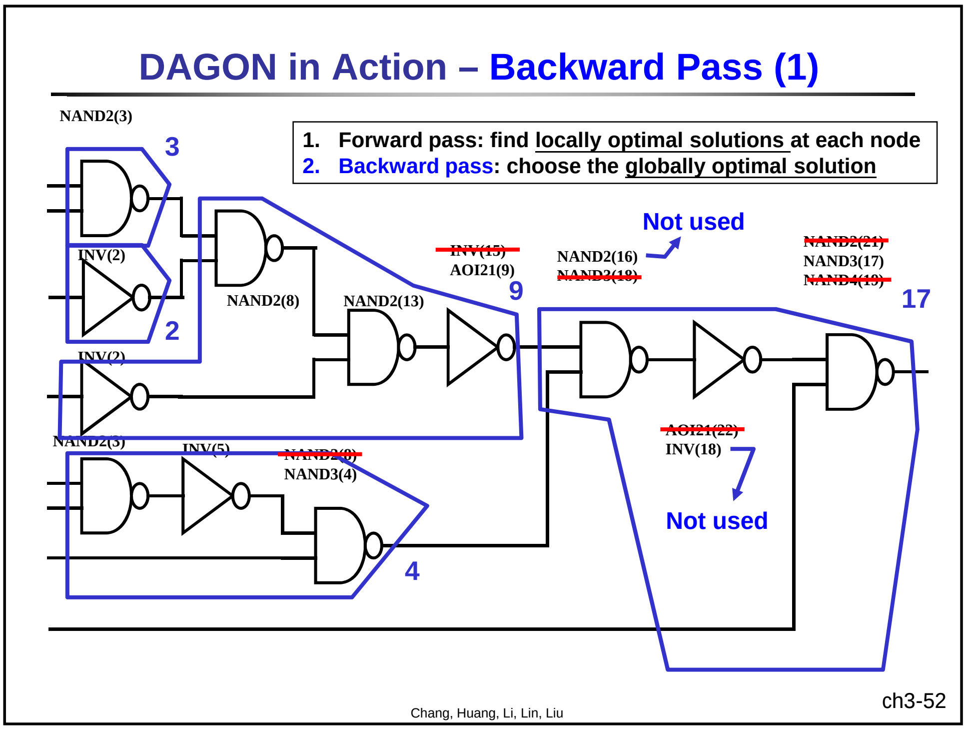
DAGON in Action – Forward Pass (3)

1. Forward pass: find locally optimal solutions at each node
2. **Backward pass**: choose the globally optimal solution



DAGON in Action – Backward Pass (1)

1. Forward pass: find locally optimal solutions at each node
2. Backward pass: choose the globally optimal solution



DAGON in Action – Backward Pass (1)

1. Forward pass: find locally optimal solutions at each node
2. Backward pass: choose the globally optimal solution

Concluding Remarks

- A Milestone Design Technology
  - RTL coding ➔ Translation & Optimization ➔ Done !
- RTL Synthesis
  - Control Data Flow Graph for component binding
  - Storage Element Inference
- Logic Synthesis
  - Division-Based Factorization
  - Kernel-Based Factorization
  - Common Sub-Expression Extraction
- Technology Mapping
  - A pattern matching problem

Being A Million-Dollar Concept,
Synthesis Quantum-Jumps The Productivity.

清華大學 EE 5265
積體電路設計自動化

單元 4
Simulation

教育部顧問室
「超大型積體電路與系統設計」教育改進計畫
EDA聯盟 – 推廣課程

致謝

本單元之教材主要取自於
教育部
超大型積體電路與系統設計教育改進計畫
EDA聯盟之課程發展成果

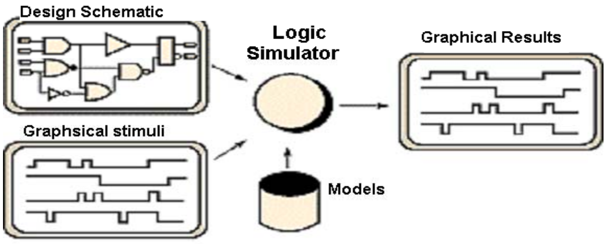
- **(教材編纂小組成員)**
- 台灣大學電機系 張耀文
- 清華大學電機系 黃錫瑜
- 交通大學資科系 李毅郎
- 中央大學電機系 劉建男
- 元智大學資工系 林榮彬

Outline

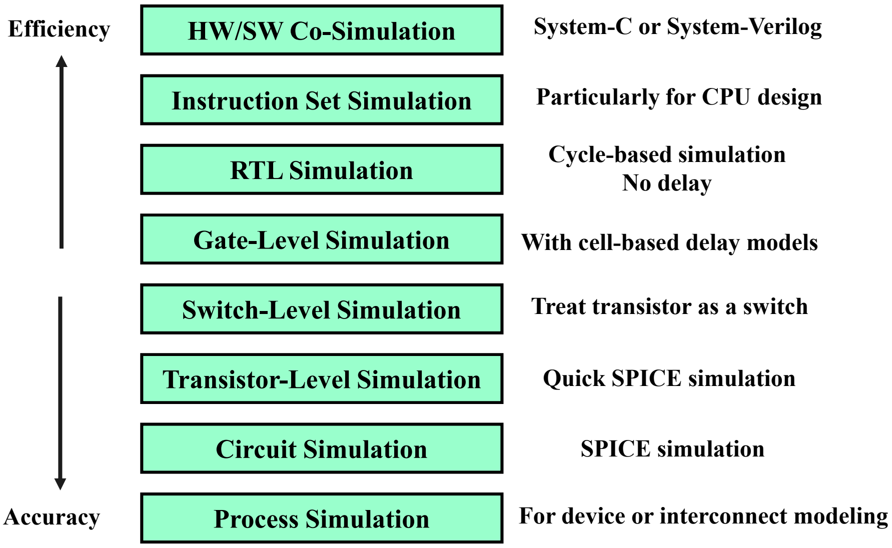
- Introduction
- Gate-level simulation
  - Compiled-Code Simulation
  - Event-Driven Simulation
- Switch-Level Simulation

Simulation

- Simulation
  - is a design validation process for checking a circuit's functionality, power dissipation and/or timing
- Inputs
  - (1) A design model (e.g., RTL code or gate-level netlist)
  - (2) A set of input signals (stimuli, patterns, or vectors)
  - (3) A pre-characterized cell models when necessary
- Outputs
  - The waveforms of output signals



Comparison of Simulation



Efficiency	HW/SW Co-Simulation	System-C or System-Verilog
	Instruction Set Simulation	Particularly for CPU design
	RTL Simulation	Cycle-based simulation No delay
	Gate-Level Simulation	With cell-based delay models
	Switch-Level Simulation	Treat transistor as a switch
	Transistor-Level Simulation	Quick SPICE simulation
	Circuit Simulation	SPICE simulation
Accuracy	Process Simulation	For device or interconnect modeling

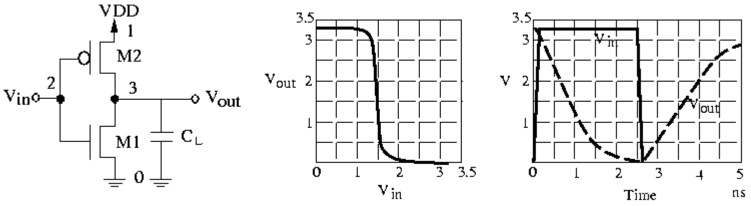
---

Circuit Simulation

- **Circuit simulators (e.g., SPICE)**
  - **Determine time-domain or frequency domain behaviors**
  - **Based on Kirchoff's voltage and current law (KVL and KCL)**
  - **Numerical methods are needed for nonlinear transistors**
  - **Need SPICE model (either functional or table) for devices**
- **DC Analysis**
  - **Finds the operating point of a circuit.**
- **AC (Small-Signal) Analysis**
  - **Finds the frequency response of a circuit.**
  - **A transistor is linearized at its DC point.**
- **Transient Analysis**
  - **Finds the time domain response for a circuit when it is excited by certain input stimuli.**
- **SPICE Home Page**
  - **http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE/**

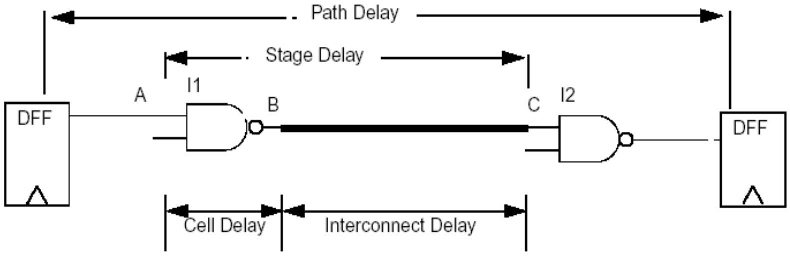
Circuit Simulation of a CMOS Inverter (0.6 μm)

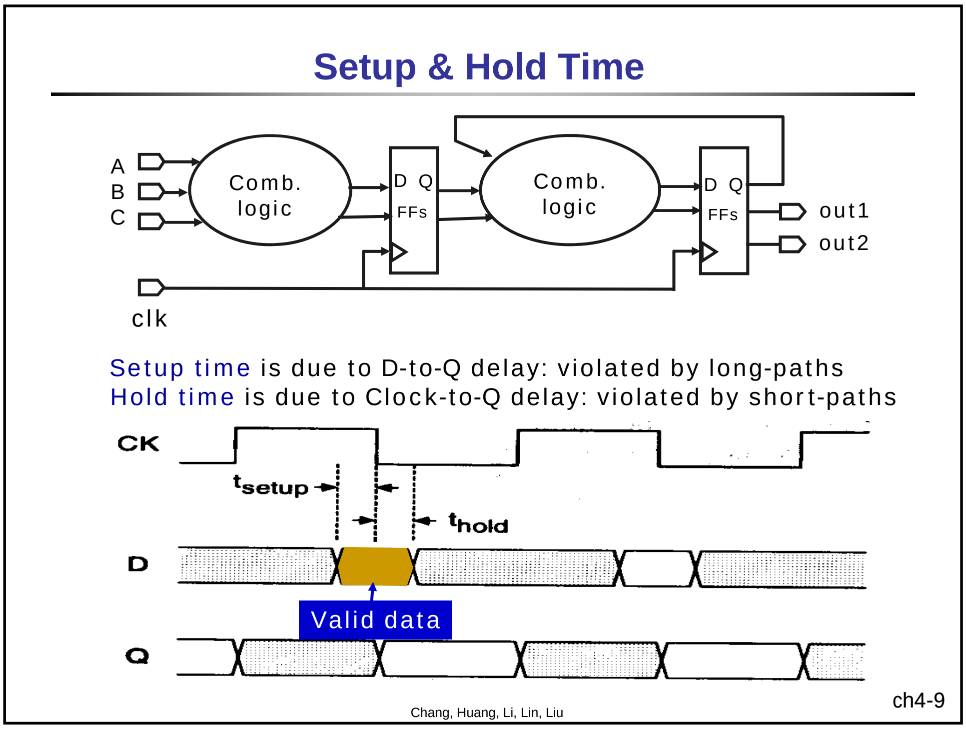
```
M1 3 2 0 0 nch W=1.2u L=0.6u AS=2.16p PS=4.8u AD=2.16p PD=4.8u
M2 3 2 1 1 pch W=1.8u L=0.6u AS=3.24p PS=5.4u AD=3.24p PD=5.4u
CL 3 0 0.2pF

VDD 1 0 3.3
VIN 2 0 DC 0 PULSE (0 3.3 0ns 100ps 100ps 2.4ns 5ns)

.LIB ’../mod_06’ typical

.OPTION NOMOD POST INGOLD=2 NUMDGT=6 BRIEF
.DC VIN 0V 3.3V 0.001V
.PRINT DC V(3)
.TRAN 0.001N 5N
.PRINT TRAN V(2) V(3)
.END
```



Cell Delay and Interconnect Delay

Stage Delay: Cell delay + Interconnect delay
Path: (PI ➔ PO, FF→PO, FF→FF)
Path delay: sum of stage delays along a path

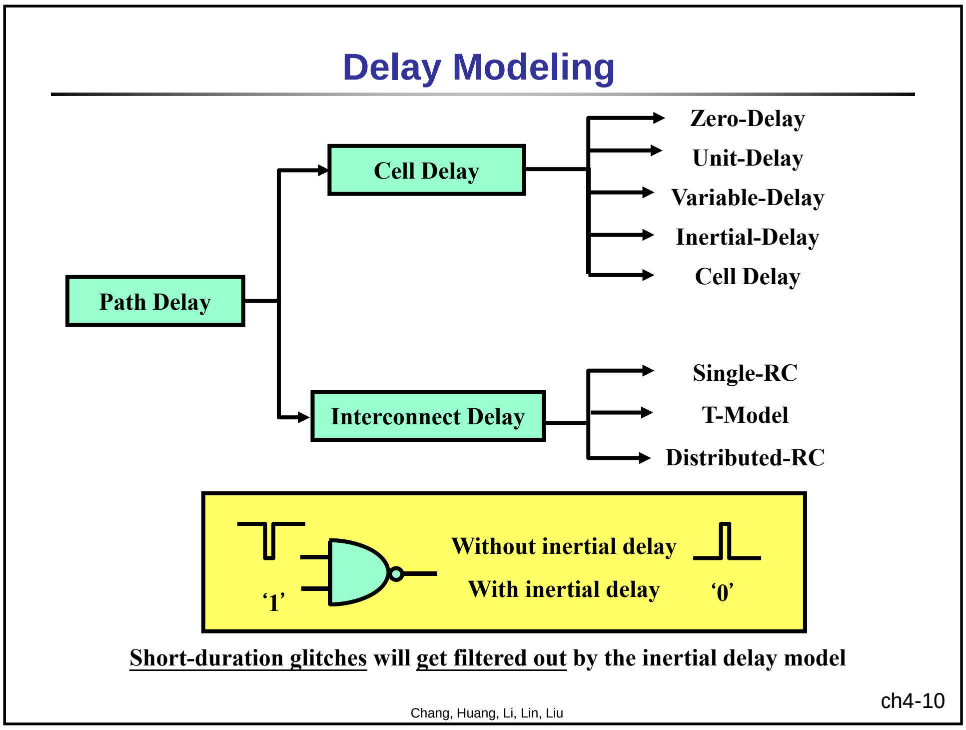




Setup & Hold Time

Setup time is due to D-to-Q delay: violated by long-paths
Hold time is due to Clock-to-Q delay: violated by short-paths

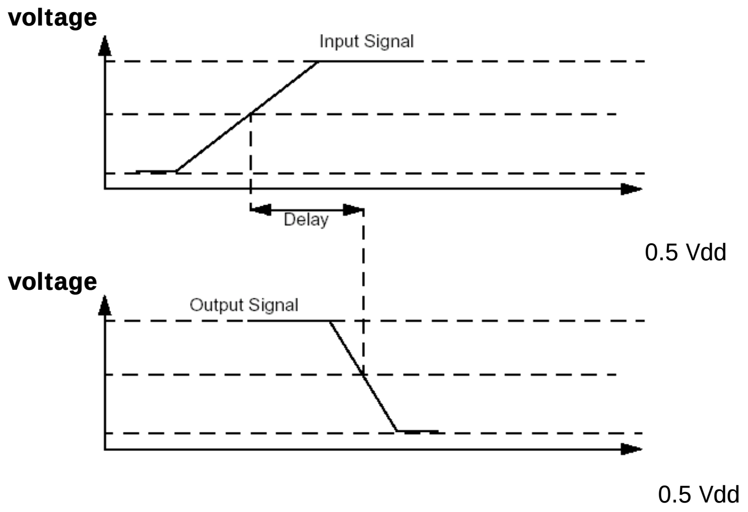
---



Delay Modeling

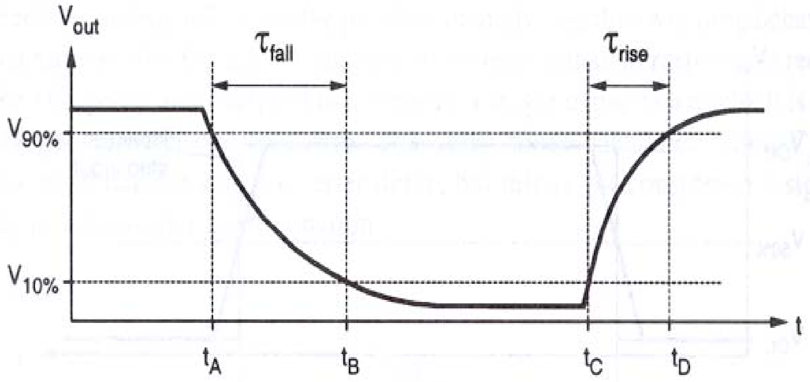
<u>Short-duration glitches</u> will <u>get filtered out</u> by the inertial delay model

Propagation Delay



Rise Time and Fall Time

Rise Time τ_{rise}: output to rise from $V_{10\%}$ to $V_{90\%}$
Fall Time τ_{fall}: output to fall from $V_{90\%}$ to $V_{10\%}$



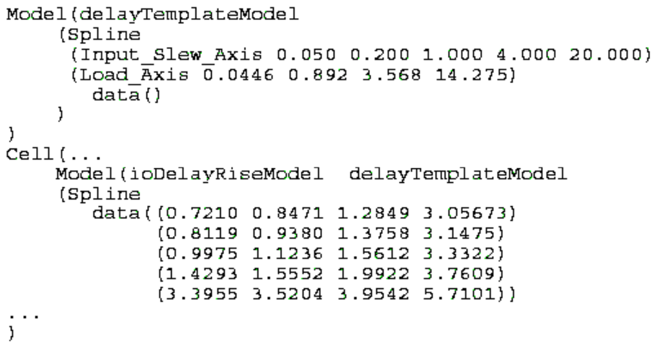
Timing Model for A Logic Cell

- Three Factors Affecting Cell Delays
  - (1) Gate size (fixed for a given cell)
  - (2) Loading capacitance
  - (3) Input slope
- Three PVT Corners (Process + Operation)
  - (1) Worst case delay
    - Using T(emperature) = 125° C, supply voltage= 90% Vdd, and worst case SPICE model for delay characterization.
  - (2) Best case delay
    - Using T = 0° C, supply voltage= 110% Vdd, and best case SPICE model for delay characterization.
  - (3) Typical case delay
    - Using T = 27° C, supply voltage= 100% Vdd, and typical case SPICE model for delay characterization.

Pre-characterized Timing Table

- A timing table for each cell is provided by library vendor
  - To look up propagation delay, rise time, fall time
  - Based on load capacitance and input slope
- For unspecified input conditions
  - Interpolation or extrapolation is used

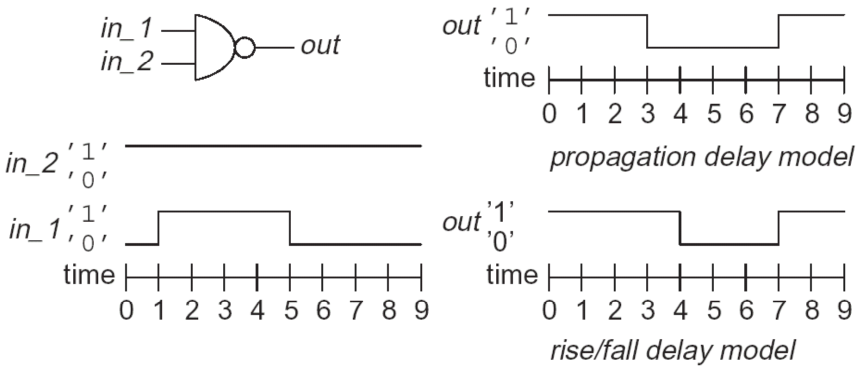
Delay table for some cell in Cadence TLF format



```
Model(delayTemplateModel
     (Spline
      (Input_Slew_Axis 0.050 0.200 1.000 4.000 20.000)
      (Load_Axis 0.0446 0.892 3.568 14.275)
        data()
     )
)
Cell(...
     Model(ioDelayRiseModel  delayTemplateModel
     (Spline
        data((0.7210 0.8471 1.2849 3.05673)
             (0.8119 0.9380 1.3758 3.1475)
             (0.9975 1.1236 1.5612 3.3322)
             (1.4293 1.5552 1.9922 3.7609)
             (3.3955 3.5204 3.9542 5.7101))
...
)
```

Delay Model Example

Signal propagation delays = 2
No rise or fall delay

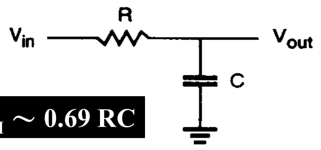


Signal propagation delays = 2
Fall delay = 1
Rise delay = 0

Interconnect Delay Models

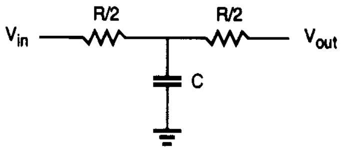
- Single RC Model
- T Model
- Distributed RC Model

RC Model

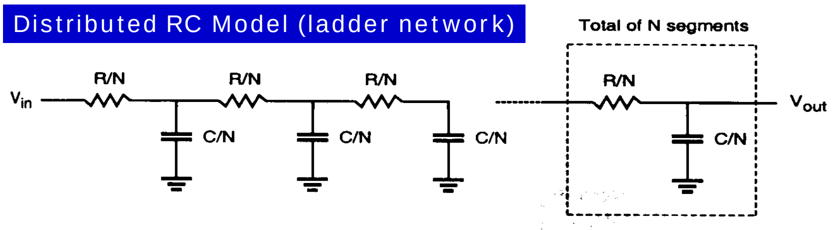


$\tau_{PLH} \sim 0.69 \text{ RC}$

T Model



Distributed RC Model (ladder network)



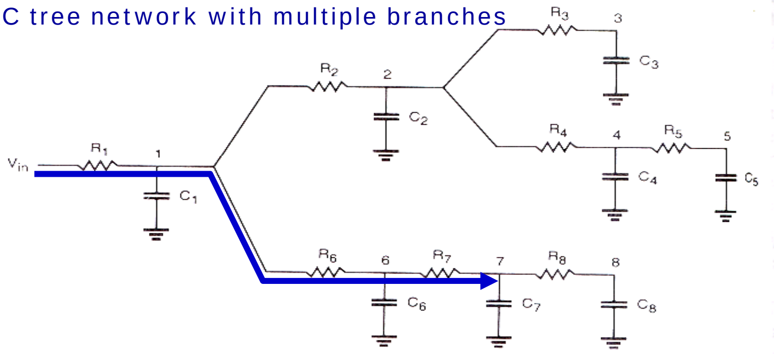
Elmore Delay Calculation

Notation: Let TC_i be the total loading capacitance seen at resistor R_i

Time constant of the delay from source to node i :

$$\boxed{\tau_{Di} = \sum_{i=0}^{N} (R_i \cdot TC_i)}$$

RC tree network with multiple branches

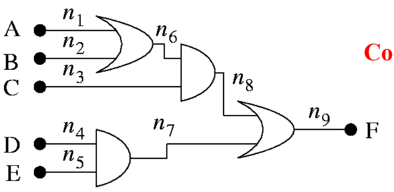


Outline

- **Introduction**
- **Gate-level simulation**
  - **Compiled-Code Simulation**
  - **Event-Driven Simulation**
- **Switch-Level Simulation**

Compiled-Code Simulation

- **Strategy of Compiled-Code Simulation**
  - Convert the circuit into a sub-routine for repeated evaluation
- **Advantages**
  - Could be computationally efficient because there is no need to process complex data structure in circuit
  - Especially suitable for zero-delay or unit-delay model
- **Drawback**
  - Cannot process complex delay model easily

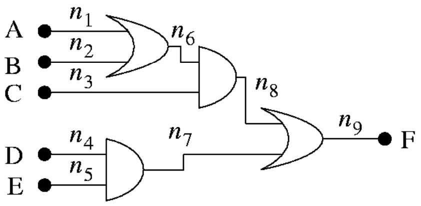


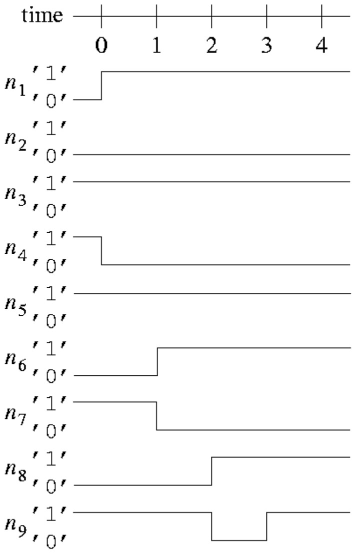
Code Generation

$n_1 \leftarrow$ A;
$n_2 \leftarrow$ B;
$n_3 \leftarrow$ C;
$n_4 \leftarrow$ D;
$n_5 \leftarrow$ E;
$n_6 \leftarrow$ OR(n_1, n_2);
$n_7 \leftarrow$ AND(n_4, n_5);
$n_8 \leftarrow$ AND(n_6, n_3);
$n_9 \leftarrow$ OR(n_7, n_8);
F $\leftarrow n_9$;

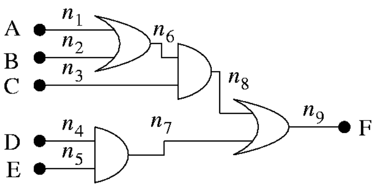
Unit-Delay Simulation

- **Assumes that all gate delays equal 1.**
- **Provides some information about signal evolution in time, especially to detect *glitches*.**





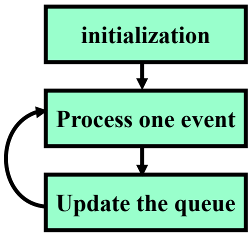
Compiled Code for Unit-Delay Simulation



```
for ($t \leftarrow t_{start}$; $t \leq t_{end}$; $t \leftarrow t + 1$) {
    new[1] ← A;
    new[2] ← B;
    new[3] ← C;
    new[4] ← D;
    new[5] ← E;
    new[6] ← OR(old[1], old[2]);
    new[7] ← AND(old[4], old[5]);
    new[8] ← AND(old[6], old[3]);
    new[9] ← OR(old[7], old[8]);
    F ← new[9];
    old ← new;
}
```

Event-Driven Simulation

- Event-driven simulation
  - is a widely-used mechanism in gate-level and switch-level simulators.
- An event
  - is a change of a signal value that may trigger new changes.
- A queue of events
  - Is needed
- Basic steps:
  - (1) Initialize the input stimuli
  - (2) Process one event from the queue
  - (3) Put newly born events into the queue
  - (4) Go back to step 2 until queue is empty



initialization

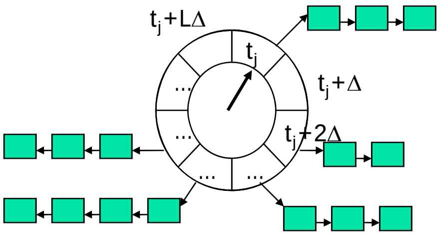
Process one event

Update the queue

Data Structures For Event Queue

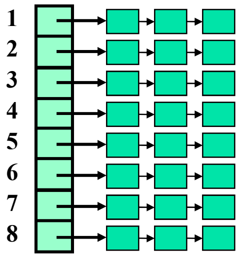
- **Potential Data Structures for Event Queue**
  - **(1) Timing wheel**
  - **(2) Array of linked list**
  - **(3) Priority queue (e.g., Heap)**

Timing Wheel



t_j is current time
Δ is the time step

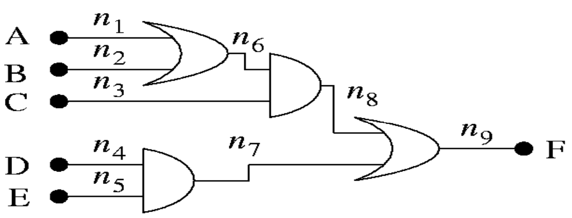
Array of linked list



Signal Modeling for Gate-Level Simulation

- **Binary Value Simulation**
  - **Each gate's output can take on a value of either '0' or '1'.**
- **Three-Valued Simulation**
  - **Signal value set is {'0', '1' and 'X'}.**
  - **'X' means ''unknown''.**
- **Nine-Valued Simulation**
  - **IEEE std_logic data type with 9 values.**
  - **mixture of level and strength.**
    - **'U' (uninitialized)**
    - **'X' (forcing unknown)**
    - **'0' (forcing 0); '1' (forcing 1)**
    - **'Z' (high impedance)**
    - **'W'(weak unknown)**
    - **'L' (weak 0), 'H', (weak 1)**
    - **'–' (don't care).**

Example (1/10)



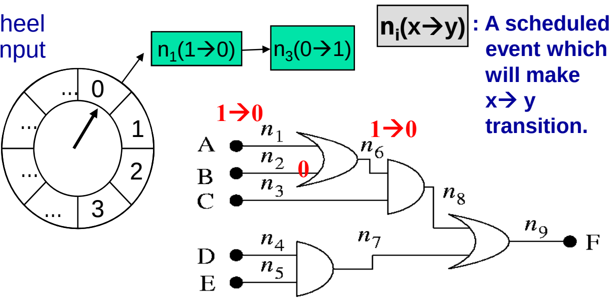
- **Assumptions**
  - **Propagation delay for two-input OR gate is 2 ns**
  - **Propagation delay for two-input AND gate is 3ns**
  - **Time resolution for simulation is 1 ns (i.e., Δ = 1ns).**
- **Input stimuli at time 0:**
  - **A: 1→ 0, B:0→0, C: 0→1, D: 0→0, E: 0→0**

Example (2/10)

- Set up the timing-wheel for the changes of input A and C.

Current time t=0

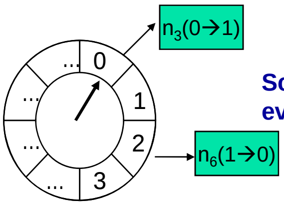
n_1=1, n_2=0, n_3=0, n_4=0, n_5=0, n_6=1, n_7=0, n_8=0, n_9=0



n_i(x→y) : A scheduled event which will make x→ y transition.

Process event n_1(1→0) at t= 0

n_1=1→0, n_2=0, n_3=0, n_4=0, n_5=0, n_6=1, n_7=0, n_8=0, n_9=0

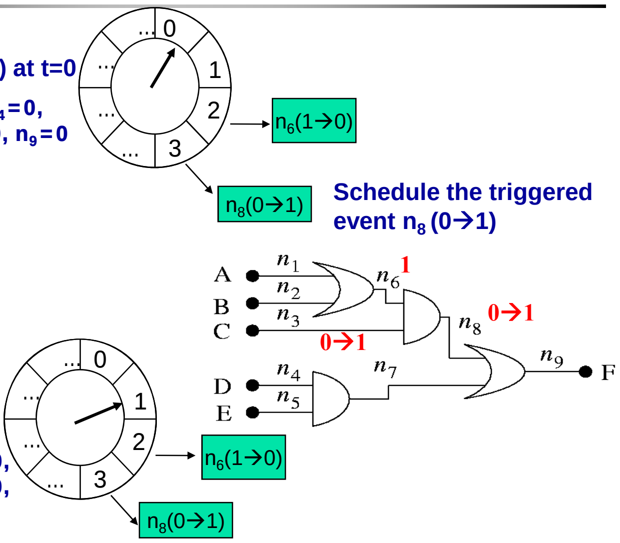


Schedule the triggered event n_6(1→0)

Example (3/10)

Process event n_3(0→1) at t=0

n_1=0, n_2=0, n_3=0→1, n_4=0, n_5=0, n_6=1, n_7=0, n_8=0, n_9=0



Schedule the triggered event n_8 (0→1)

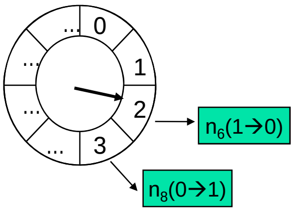
Advance current time by one resolution unit to t = 1

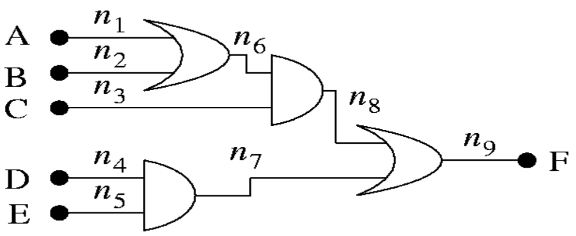
n_1=0, n_2=0, n_3=1, n_4=0, n_5=0, n_6=1, n_7=0, n_8=0, n_9=0

Example (4/10)

Advance current time by one resolution unit to t=2

n_1=0, n_2=0, n_3=1, n_4=0, n_5=0, n_6=1, n_7=0, n_8=0, n_9=0

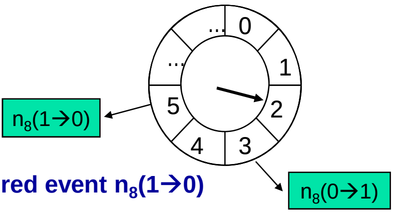




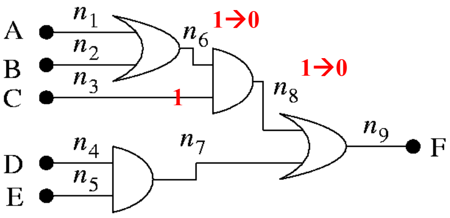
Example (5/10)

Process event $n_6(1\rightarrow 0)$ at t=2

n_1=0, n_2=0, n_3=1, n_4=0, n_5=0,
n_6=1→0, n_7=0, n_8=0, n_9=0



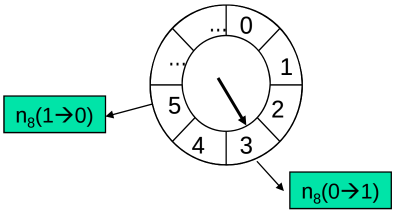
Schedule triggered event $n_8(1\rightarrow 0)$

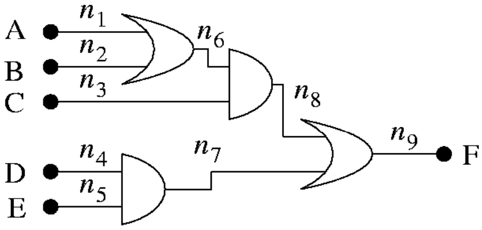


Example (6/10)

Advance time to t=3

n_1=0, n_2=0, n_3=1, n_4=0,
n_5=0, n_6=0, n_7=0, n_8=0,
n_9=0

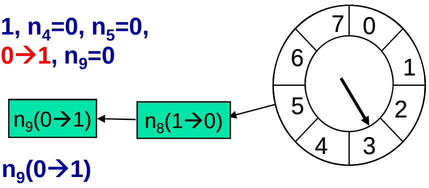




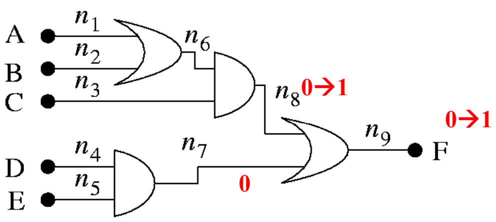
Example (7/10)

Process event $n_8(0\rightarrow1)$ at t=3

$n_1=0$, $n_2=0$, $n_3=1$, $n_4=0$, $n_5=0$,
$n_6=0$, $n_7=0$, $n_8=0\rightarrow1$, $n_9=0$



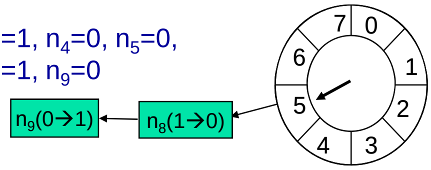
Schedule event $n_9(0\rightarrow1)$

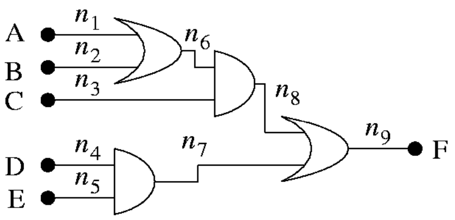


Example (8/10)

Advance time to t=4 and then to t=5

$n_1=0$, $n_2=0$, $n_3=1$, $n_4=0$, $n_5=0$,
$n_6=0$, $n_7=0$, $n_8=1$, $n_9=0$

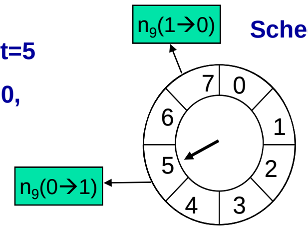




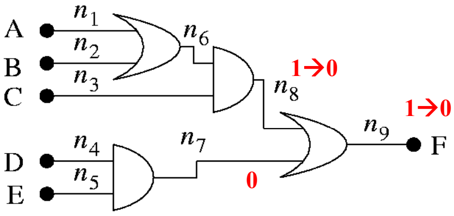
Example (8/10)

Process event $n_8(1 \rightarrow 0)$ at t=5

$n_1=0$, $n_2=0$, $n_3=1$, $n_4=0$, $n_5=0$,
$n_6=0$, $n_7=0$, $n_8=1 \rightarrow 0$, $n_9=0$



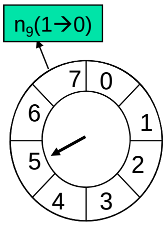
Schedule event $n_9(1 \rightarrow 0)$

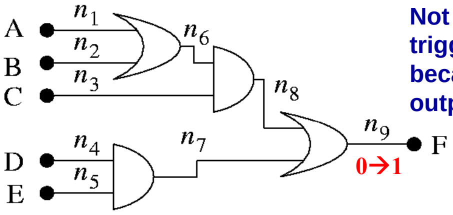


Example (9/10)

Process event $n_9(0 \rightarrow 1)$ at t=5

$n_1=0$, $n_2=0$, $n_3=1$, $n_4=0$, $n_5=0$,
$n_6=0$, $n_7=0$, $n_8=0$, $n_9=0 \rightarrow 1$



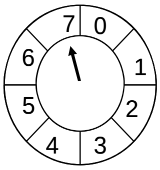


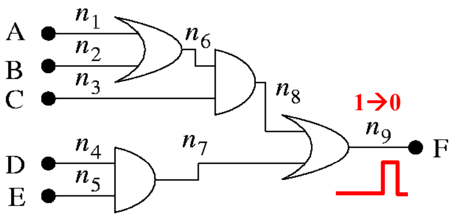
Not scheduling any triggered event because n_9 is an output.

Example (10/10)

Advance time to t=6, t=7 and then process event n_9(1→0) at t=7

n_1=0, n_2=0, n_3=1, n_4=0, n_5=0, n_6=0, n_7=0, n_8=0, n_9=1->0





Not scheduling any triggered event because n_9 is an output.

- There are hazards on n8(0→1→0) and n9 (0→1→0).
- It takes 7ns to propagate the input change to the output.

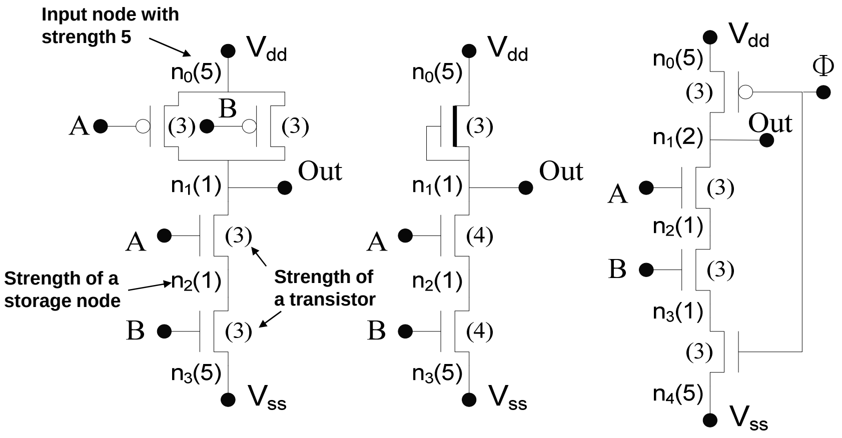
Outline

- Introduction
- Gate-level simulation
  - Compiled-Code Simulation
  - Event-Driven Simulation
- Switch-Level Simulation
  - Circuit Partitioning
  - Evaluate each channel-connected component

Basics of Switch-Level Simulation

- **Input**
  - A transistor schematic
- **Simulation Strategy**
  - (1) Treating transistors as bi-directional switches
  - (2) Label each transistor by its on-resistance
  - (3) Label each node by (strength, value) pair
  - (4) Parasitic RC can be included
- **Two types of nodes**
  - (1) input node and (2) charged node (or storage node)
- **Input node**
  - Could be Vdd, GND, strong ‘0’ or ‘1’
  - The strength of an input node is the maximum one
- **Charged node**
  - Is associated with a capacitance
  - The strength is proportional to its capacitance

Strength Model Example



Input node with strength 5

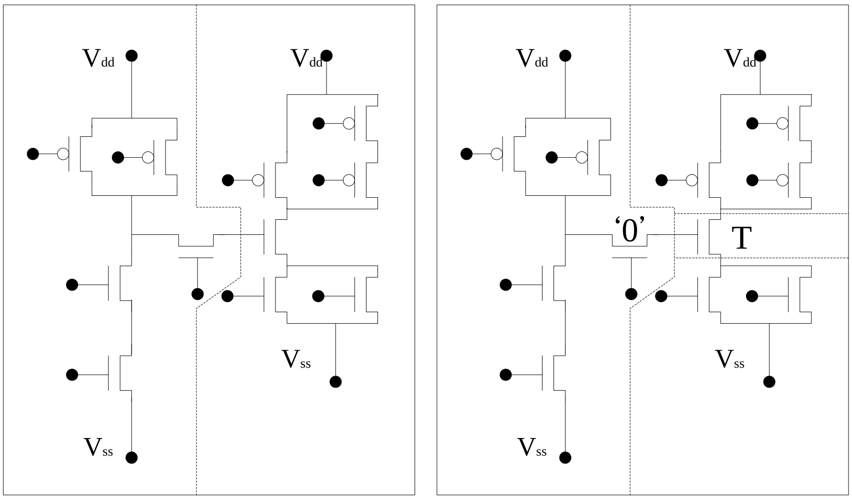
Strength of a storage node

Strength of a transistor

Switch-Level Simulation Techniques

- Partitioning the circuit into subcircuits that can be treated as unidirectional components.
  - Static partitioning: Connections to the gate of a transistor determine subcircuit boundaries irrespective of the signals carried by the nets.
  - Dynamic partitioning: Known signal values in the network are taken into account such that further partitioning of subcircuits is possible.
- Each subcircuit is then modeled as a channel-connected component or a switch graph (multigraph) G=(V, E), where
  - V is a set of vertices representing input or storage nodes labeled with node (net) names and strengths.
  - E is a set of edges representing transistors labeled with a transistor name and strength.

Static Versus Dynamic Partitioning

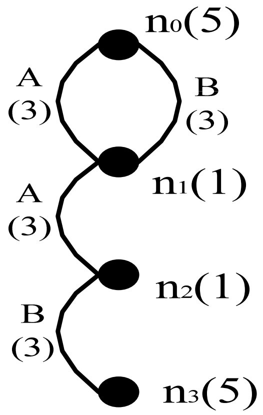


Static partitioning

Dynamic partitioning

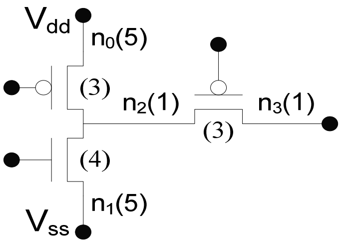
Multi-Graph

- A convenient representation for switch-level circuits is a multigraph.
- Vertices represent nets and are labeled with the net name and strength.
- Edges represent transistors and are labeled with a transistor ID and strength.



Ex:Evaluate a Channel-Connected Component

- The table below shows how input signal changes are propagated to the output.



Propagate (from → to)	State of n_2	State of n_3
"Initial state"	('X', 1)	('X', 1)
$n_o \rightarrow n_2$	('1', 3)	('X', 1)
$n_1 \rightarrow n_2$	('0', 4)	('X', 1)
$n_2 \rightarrow n_3$	('0', 4)	('0', 3)

Winner takes all

Logic value Strength

Switch-Level Timing Simulation

- Need delay models to account for
  - Transistor on-resistance and capacitance
  - Interconnect resistance and capacitance
- Delay Models
  - Lumped RC model (overestimating delay)
  - Lumped RC model + input slope (slew rate)
  - Distributed RC model + input slope

Concluding Remarks

- Trade-off in simulation
  - Behavior-level → Cycle-accurate → Timing-accurate
- Two major types of gate-level simulation
  - Compiled-code simulation
  - Event-driven simulation
- Switch-Level Simulation
  - Partitioning of circuits into channel-connected components
  - A fight-breaking scheme in terms of strength of signals

Simulation Is Not Real, It Is Just Almost Real.

清華大學 EE 5265

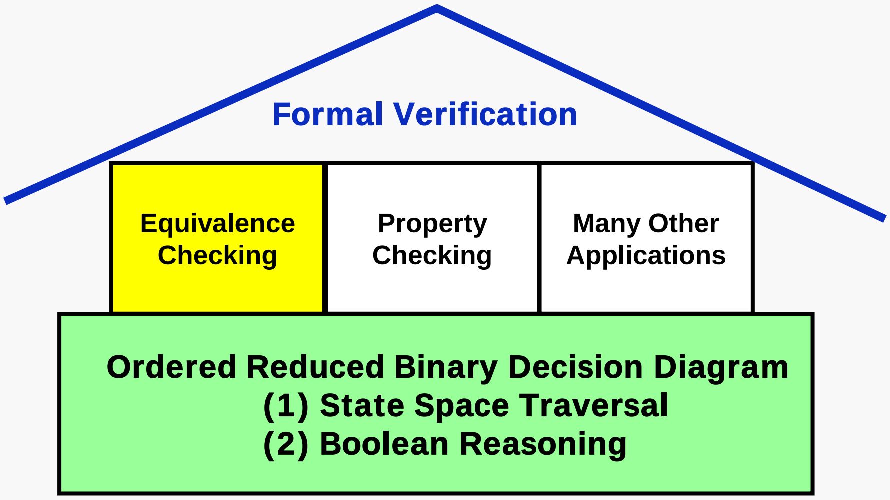
積體電路設計自動化

單元 5：Formal Verification

清華大學電機系　黃錫瑜

教育部顧問室
「超大型積體電路與系統設計」教育改進計畫
EDA聯盟 – 推廣課程

Overview

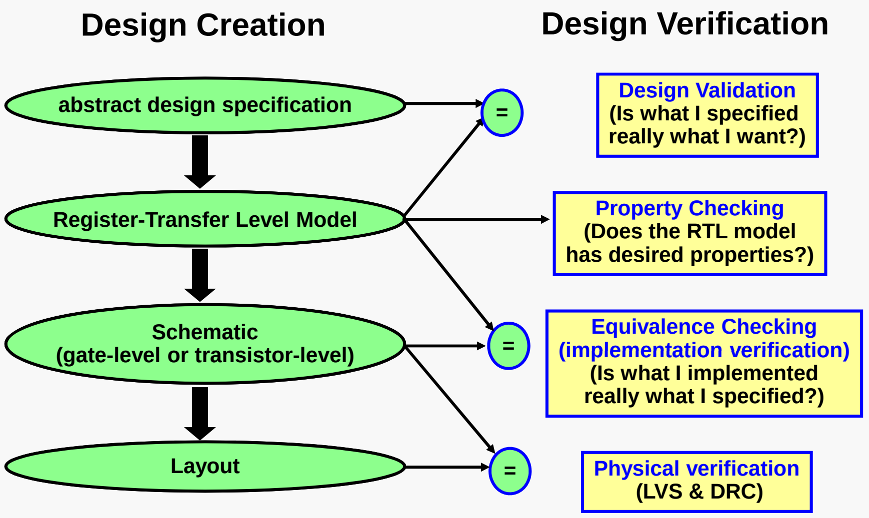


Ordered Reduced Binary Decision Diagram
(1) State Space Traversal
(2) Boolean Reasoning

Outline

- Fundamentals
  - The roles of formal verification
  - Binary Decision Diagram (BDD)
- Equivalence Checking
  - Product Machine
  - State Space Traversal
  - Implicit State Enumeration

The Roles of Functional Verification



Design Creation

Design Verification

abstract design specification

Register-Transfer Level Model

Schematic
(gate-level or transistor-level)

Layout

=

Design Validation
(Is what I specified really what I want?)

Property Checking
(Does the RTL model has desired properties?)

Equivalence Checking (implementation verification)
(Is what I implemented really what I specified?)

Physical verification
(LVS & DRC)

LVS: layout vs. schematic check, DRC: design rule check

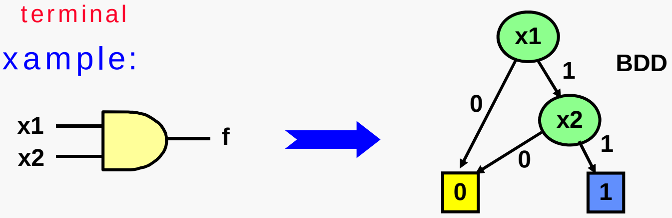
Functional Verification Paradigms

- Simulation
  - not complete (i.e., may fail to catch bugs)
  - very time-consuming, especially when at lower abstraction levels such as the gate or transistor level
  - still the most popular way for design validation
- Emulation
  - (1) based on an FPGA-based emulation system, or
  - (2) based on a massively parallel machine (e.g., with 8 boards, each having 128 processors)
  - 2 to 3 orders of magnitude faster than software simulation
  - costly and might not be very easy-to-use
- Formal verification
  - a relatively new paradigm for property checking and equivalence checking
  - requires no input stimuli
  - perform exhaustive proof through rigorous logical reasoning

---

Binary Decision Diagram (BDD)

- Basic Features
  - BDD was proposed by [R.E. Bryant] in 1986
    - "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. on Computers*, vol. C-35, Aug. 1986, pp. 677-691.
  - BDD is a Directed Acyclic Graph (DAG) used to represent a Boolean function f: $B^n \rightarrow B$
  - each non-terminal node is a decision node associated with an input variable with two branches – 0-branch and 1-branch
  - There are two terminal nodes – 0-terminal and 1-terminal
- Example:



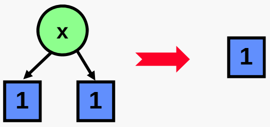
Canonicity

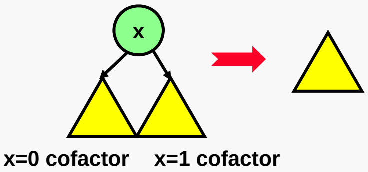
- Canonicity Requirements
  - The BDD representation is not canonical for a given Boolean function unless the following constraints are satisfied:
  - (1) Simple BDD – each variable can appear only once along each path from the root to a leaf
  - (2) Ordered BDD – Boolean variables are ordered in such a way that if the node labeled x_i has a child labeled x_k, then order(x_i) < order(x_k)
  - (3) Reduced BDD – no two nodes represent the same function, I.e., redundancies are removed by sharing isomorphic sub-graphs

---

Reduced Ordered BDD (ROBDD)

- Rules for ROBDD
  - Rule 1: merge two children with the same terminal nodes
  - Rule 2: merge two isomorphic sub-graphs





- Reduction Procedure
  - Input: An arbitrary BDD
  - Output: A canonical reduced ordered BDD
  - Traverse the graph from the terminal nodes towards to root node (I.e., in a bottom-up manner) and apply the above reduction rules whenever possible

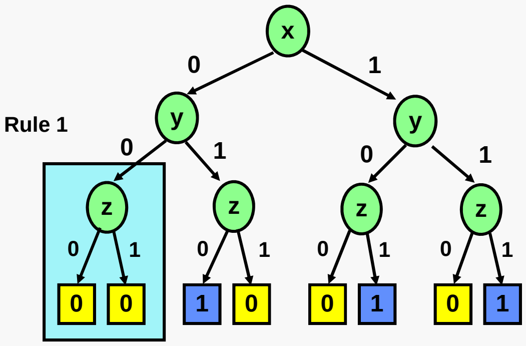
Example: BDD Reduction (1)

- f = x’yz’ + xz
- variable order: x → y → z

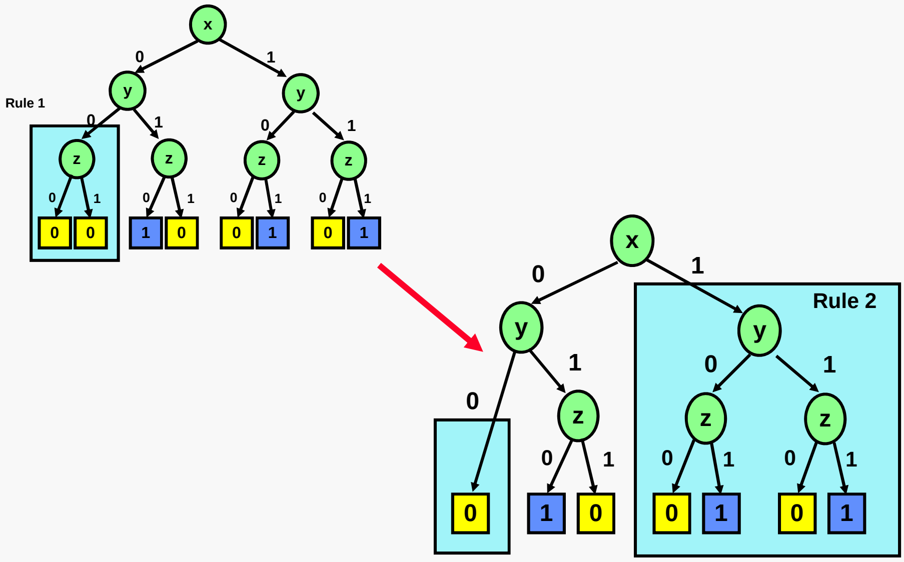
Truth table

xyz	f
000	0
001	0
010	1
011	0
100	0
101	1
110	0
111	1

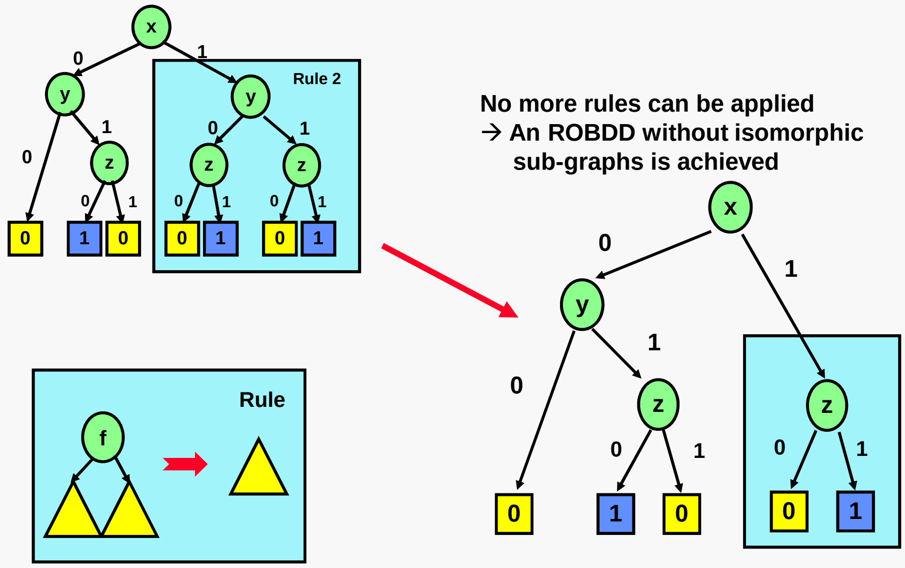
not yet reduced BDD



Example: BDD Reduction (2)



Example: BDD Reduction (3)



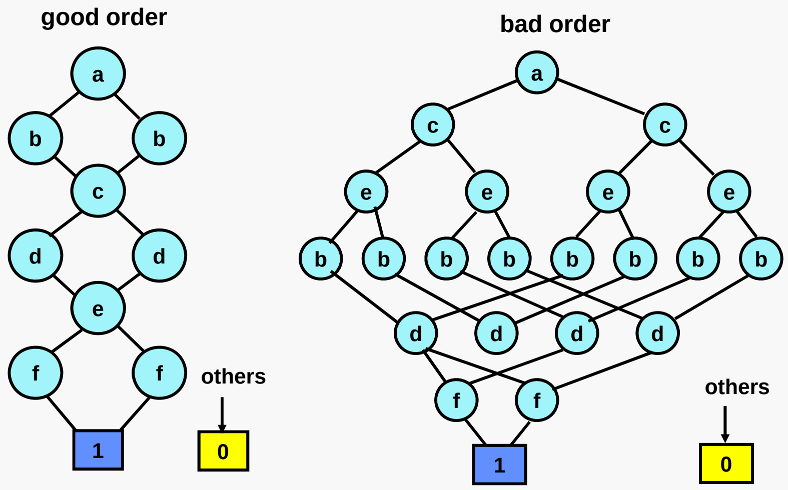
No more rules can be applied
→ An ROBDD without isomorphic sub-graphs is achieved

The Influence of Variable Ordering

- Size of BDD
  - can vary from linear to exponential in the number of the variables, depending on the variable ordering
- Hard-to-Build BDD
  - Data path components (e.g., multipliers) cannot be represented in polynomial space, regardless of the variable ordering
- Heuristics of Ordering
  - (1) Put variables that influence most on the top of BDD
  - (2) Minimize the distance between strongly related variables
  - (e.g., x1x2 + x2x3 + x3x4)
    x1→x2→x3→x4 is better than x1→x4→x2→x3

Example on Variable Ordering

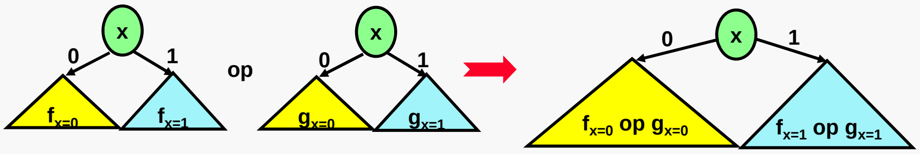
$z = (a \oplus b) \cdot (c \oplus d) \cdot (e \oplus f)$



Recursive BDD Operations

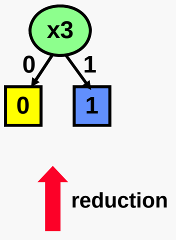
- Notations
  - f and g are two BDDs representing two functions
  - op is a Boolean operator (I.e., AND, OR, NOT, ...)
- BDD operation
  - Problem: Construct the BDD of h = f op g
  - A recursive procedure on each variable
    - $h = x \cdot h_{x=1} + x' \cdot h_{x=0}$, where x is a variable

      $= x \cdot (f \text{ op } g)_{x=1} + x' \cdot (f \text{ op } g)_{x=0}$
    - For most operations, $(f \text{ op } g)_{x=1} = (f_{x=1} \text{ op } g_{x=1})$
    - Hence $h = x \cdot (f_{x=1} \text{ op } g_{x=1}) + x' (f_{x=0} \text{ op } g_{x=0})$



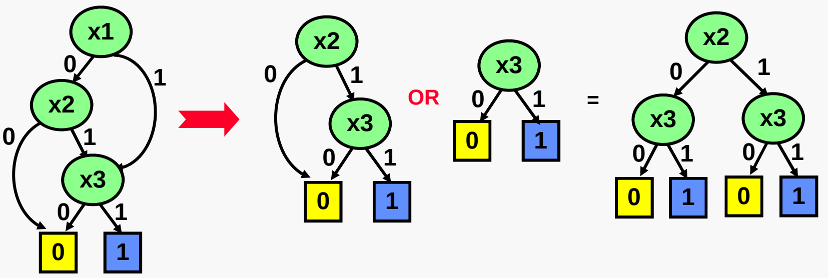
Existential Quantification

- Definition
  - $\exists x_1 [f(x_1, y_1, \ldots, y_n)] = g(y_1, \ldots, y_n)$
    such that $g(y_1, \ldots, y_n) = 1$
    iff $f(0, y_1, \ldots, y_n)=1$ or $f(1, y_1, \ldots, y_n)=1$
- Example



$f = (x1+x2) \cdot x3$

$\exists x_1\ f = f_{x1=0} + f_{x1=1}$

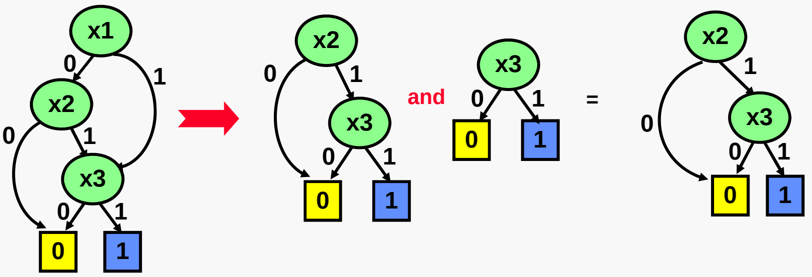


Universal Quantification

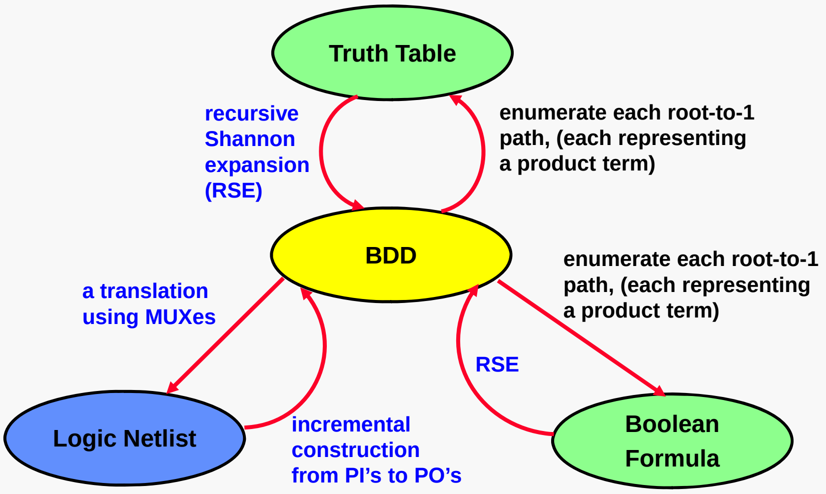
- Definition
  - $\forall x_1 [f(x_1, y_1, \ldots, y_n)] = g(y_1, \ldots, y_n)$
    such that $g(y_1, \ldots, y_n) = 1$
    iff $f(0, y_1, \ldots, y_n)=1$ and $f(1, y_1, \ldots, y_n)=1$
- Example

$f = (x1+x2) \cdot x3$

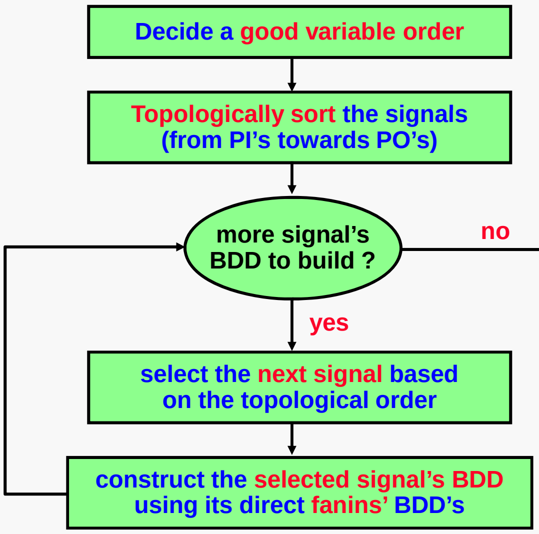
$\exists x_1\ f = f_{x1=0} + f_{x1=1}$

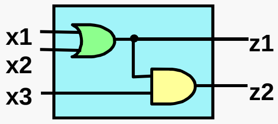


Translations Among Boolean Function Representations



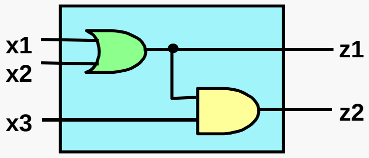
From Netlist to OBDD





Boolean network C

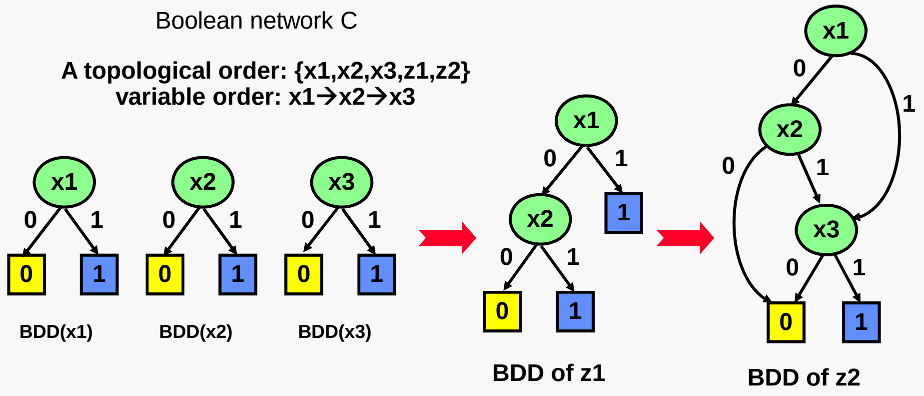
Example: Constructing BDD



Boolean network C

BDD(z2) = BDD(x3) · BDD(z1)

A topological order: {x1,x2,x3,z1,z2}
variable order: x1→x2→x3



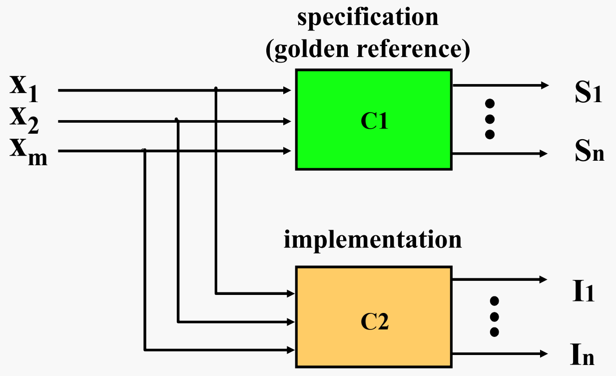
Summary of BDD

- Good Properties
  - BDD is a compact representation for Boolean functions
  - Canonical, given a fixed variable ordering
  - Polynomial time in BDD size for many Boolean operations
- Bad Properties
  - In the worst case, the size of a BDD is $O(2^n)$ for n-input Boolean functions

Outline

- Fundamentals
  - The roles of formal verification
  - Binary Decision Diagram (BDD)
- Equivalence Checking
  - Product Machine
  - State Space Traversal
  - Implicit State Enumeration

The Problem of Equivalence Checking

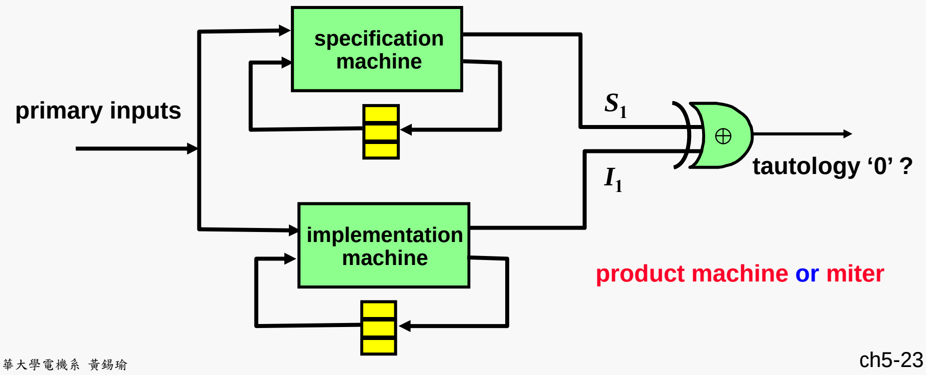


(Question): Is every primary output pair equivalent (i.e., $S_k = I_k$, $1 \leqq k \leqq n$) for all possible input sequences?

Product Machine

Assumption:
the no. of states in the specification machine: **n1**
the no. of states in the implementation machine: **n2**
Then the product machine has (n1 x n2) states

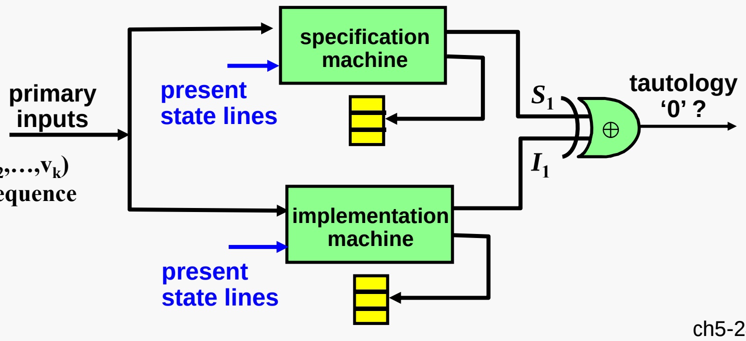
Two machines are equivalent if and only if the product machine's outputs are tautology '0' for all possible input sequences



product machine or miter

Overall Procedure for Symbolic Equivalence Checking

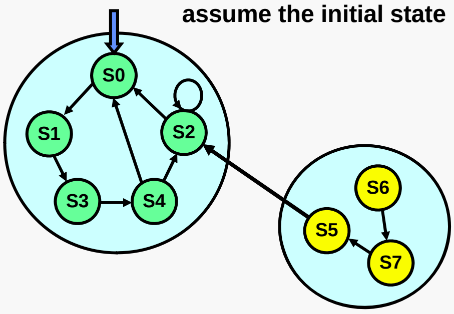
- Sequential Equivalence Checking
  - (Step1): enumerate all possible reachable states of the product machine → a process requires FSM traversal
  - (Step 2): prove every output of the product machine for any combination of primary inputs and reachable states is tautology '0' → a combinational checking problem

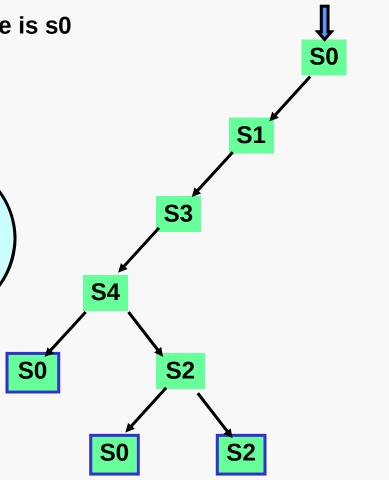


Let $\pi = (v_1, v_2, \ldots, v_k)$ be an input sequence

Reachable State Computation

assume the initial state is s0





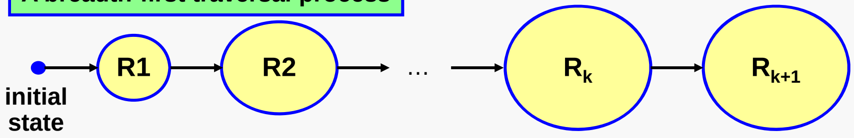
{S0, S1, S2, S3, S4} are reachable states

the tree stops at ever-visited state

Finite State Machine Traversal (FSM Traversal)

- FSM Traversal
  - a process to compute the set of reachable states
  - can be a breadth-first or depth-first traversal
  - Breadth-first traversal
    1. Initial $R_0 = \{s_0\}$
    2. $R_{j+1} = R_j \cup \{\text{next states of } R_j\}$
    3. Repeat step (2) until a fixed-point is found, i.e.,two consecutive reachable state sets R_k, R_{k+1} are the same

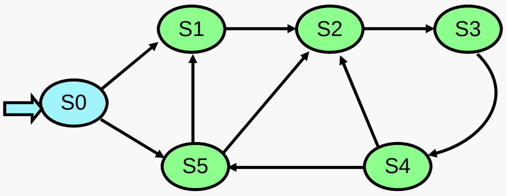
A breadth-first traversal process



R_j represents the set of states reachable in *j* steps from the initial state in the STG

fixed-point if R_k=R_{k+1}

Example: FSM-traversal



iteration j	reachable states R_j
0	{s0}
1	{s0, s1, s5}
2	{s0, s1, s2, s5}
3	{s0, s1, s2, s3, s5}
4	{s0, s1, s2, s3, s4, s5}
5	{s0, s1, s2, s3, s4, s5} (fixed point)

Implicit State Enumeration

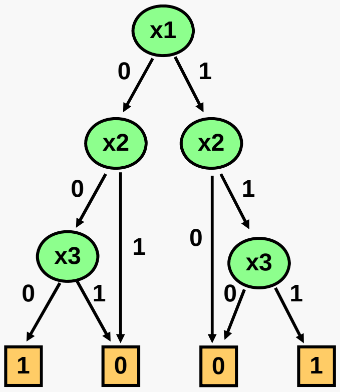
- Implicit state enumeration
  - The reachable states are computed without constructing the state transition graph explicitly
  - The state space is implicitly traversed
  - BDD is used to
    - represent a set of states
    - represent the state transition relation of a machine
  - More efficient than explicit state enumeration based on the state transition graph
  - Capable of handling larger designs (e.g., one with 10^{20} states)

BDD for Set Representation

v is now represent a set of input vectors

x1x2x3	characteristic function f_v where v = {(000), (111)}
000	1
001	0
010	0
011	0
100	0
101	0
110	0
111	1

Truth table



BDD-representation

Input/Output Relation

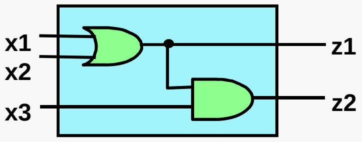
- Definition
  - Let C be a Boolean network from B^m to B^n
  - Let v be an input vector, w be an output vector
  - The I/O relation of C is a relation R_C: $B^m x B^n$, and (v,w) $\in R_C$ if C(v) = w
  - A network's I/O relation consists of every valid input/output combinations
- Characteristic formula

$$R_C(x_1, \ldots, x_m | z_1, \ldots, z_n) = (z_1 \equiv \lambda_1(X)) \bullet (z_2 \equiv \lambda_2(X)) \ldots \bullet (z_n \equiv \lambda_n(X))$$

λ_i is the Boolean function of output z_i

$$= \prod_{i=1}^{n} (z_i \equiv \lambda_i(X)) \qquad \text{where } (a \equiv b) \text{ corresponds to } (ab + \bar{a}\bar{b})$$

Example: I/O Relation

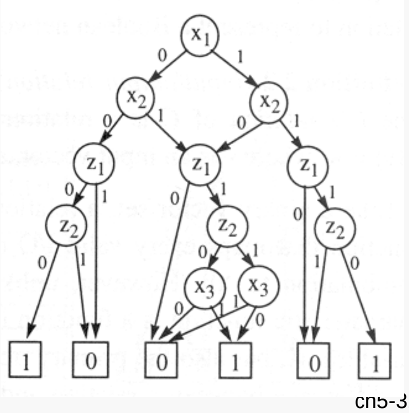


Boolean network C

$$R_C = (z_1 \equiv (x_1 + x_2)) \bullet (z_2 \equiv ((x_1 + x_2) \cdot x_3))$$

x1 x2 x3	z1 z2	input/output relation R_C	comment
000	00	1	valid combina-tions
001	00	1	
010	10	1	
011	11	1	
100	10	1	
101	11	1	
110	10	1	
111	10	1	
any other		0	invalid

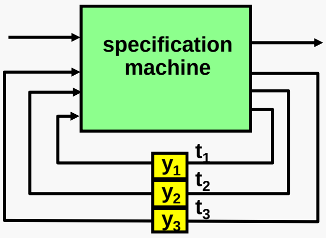
BDD representing the I/O relation



Finite State Machine

- Six-tuple Notation for a FSM
  - **M = (I, O, S, s_0, δ, λ)**
  - **I is the input space defined by input variables {x_1, x_2, …, x_m}**
  - **O is the output space defined by output variables {z_1, z_2, …, z_n}**
  - **S is the state space defined by state variables {y_1, y_1, …, y_k}**
  - **s_0 is the known initial state**
  - **δ is a set of transition functions**
  - **λ is a set of output functions**

next state line function
$t_i = \delta_i(x_1,\ldots, x_m,y_1,\ldots,y_k)$



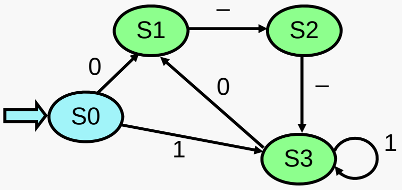
Transition Relation

- Definition
  - Let M=(I,O,S,s_0,δ,λ) be a FSM
  - The transition relation T: B^mx B^kx B^k, m and k are the dimensions of the input and state space
  - (v,p,q) ∈ T if machine M will transition from state p to state q under the input vector v
- Characteristic formula

$$T(x_1, \ldots, x_m | y_1, \ldots, y_k | t_1, \ldots t_2, t_k) = (t_1 \equiv \delta_1) \bullet (t_2 \equiv \delta_2) \ldots \bullet (t_k \equiv \delta_k)$$
$$= \prod_{i=1}^{k} (t_i \equiv \delta_i(X, Y))$$

Example: Transition Relation

Example FSM



primary inputs	current state	next state	characteristic function of transition relation T
0	S0	S1	1
1	S0	S3	1
-	S1	S2	1
-	S2	S3	1
0	S3	S1	1
1	S3	S3	1
other combinations			0

Existential Transition Relation

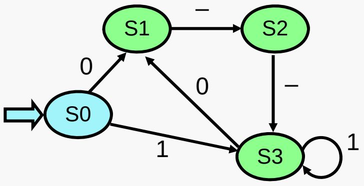
- Definition
  - Let M=(I,O,S,s_0,δ,λ) be a FSM
  - The existential transition relation T_{exist}: B^kx B^k, where k is the dimension of the state space
  - (p,q) ∈ T_{exist} if there exists an input vector that brings the machine M from state p to state q
  - Note that existential transition relation only concerns about the connectivity of the FSM's transition graph
- Characteristic formula

$$T_{exist}(y_1, \ldots, y_k | t_1, \ldots t_2, t_k) = (\exists x_1 x_2 \ldots x_m)((t_1 \equiv \delta_1) \bullet (t_2 \equiv \delta_2) \ldots \bullet (t_k \equiv \delta_k))$$

$$= (\exists x_1 x_2 \ldots x_m) \prod_{i=1}^{k} (t_i \equiv \delta_i(X, Y))$$

Example: T_{exist}

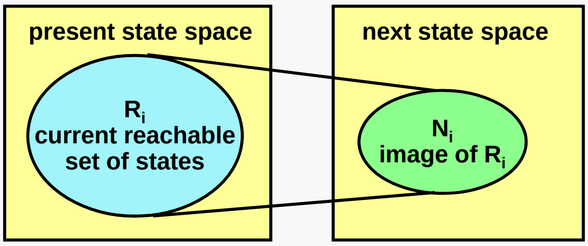
Example FSM



current state	next state	characteristic function of existential transition relation T_{exist}
S0	S1	1
S0	S3	1
S1	S2	1
S2	S3	1
S3	S1	1
S3	S3	1
others combinations		0

Reachable State Computation

- Existential Transition Relation
  - defines a projection from present state space to the next state space
  - A state could reach multiple states
  - Multiple states can reach the same next state
  - Hence, T_{exist} is a many-to-many mapping
- Reachable states R_{i+1} in the breadth-first traversal
  - $R_{i+1} = R_i \cup N_i$, where N_i is image of R_i



present state space

R_i
current reachable set of states

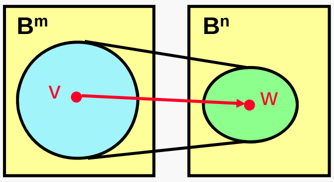
next state space

N_i
image of R_i

Symbolic Image Computation

- Definition
  - Let T be a projection, $T: B^m x B^n$
  - Let A be a set of vectors in B^m
  - The image of A is a set in B^n

    image(T, A) = $\{ w \in B^n \mid (v, w) \in T \text{ and } v \in A\}$



- Characteristic Function
  - in the application of reachable next state computation

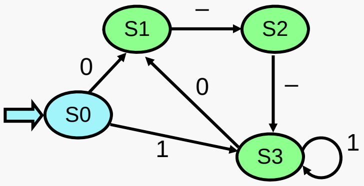
$$\textit{reachable next states } N_i = Image(T_{exist}, R)$$

$$= (\exists y_1 y_2 \ldots y_k)(R_i \bullet T_{exist})$$

$$= (\exists y_1 y_2 \ldots y_k)\left(R_i \bullet \left((\exists x_1 x_2 \ldots x_m)\prod_{i=1}^{k}(t_i \equiv \delta_i(X, Y))\right)\right)$$

Ex: Next-State Computation

Example FSM



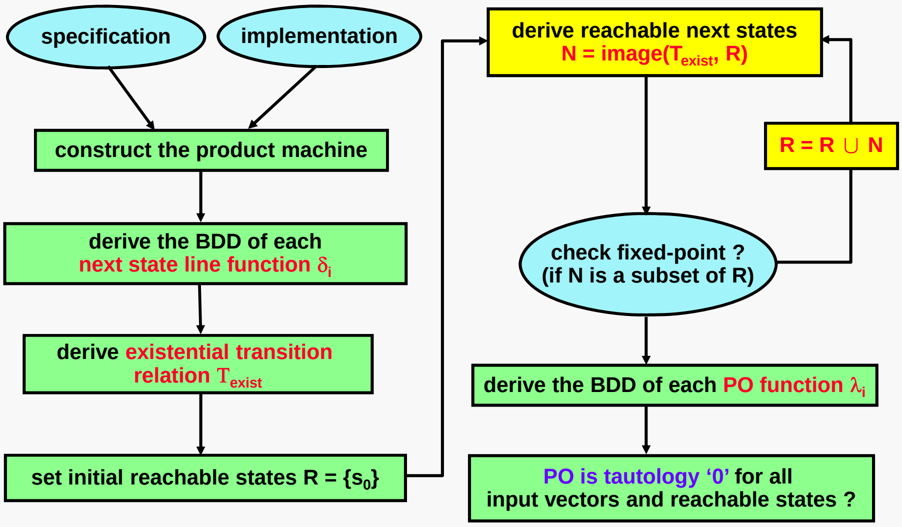
Transition Relation:

T = { (0, S_0, S_1), (1, S_0, S_3), (-, S_1, S_2), (-, S_2, S_3), (1, S_3, S_3), (0, S_3, S_1)}
T_{exist} = { (S_0, S_1), (S_0, S_3), (S_1, S_2), (S_2, S_3), (S_3, S_3), (S_3, S_1)}

What is the set of the next states of R = {S_1, S_3}?

$R \cap T_{exist}$ = {(S_1, S_2), (S_3, S_3), (S_3, S_1)}
➔ It implies that there are three transitions outgoing from {S_1, S_3}
And the destination states (i.e. the next states) include {S_2, S_3, S_1}
So, final set of reachable next states from {S_1, S_3} is {S_1, S_2, S_3}

Overall Flow of Sequential Equivalence Checking



specification

implementation

construct the product machine

derive the BDD of each next state line function δ_i

derive existential transition relation T_{exist}

set initial reachable states R = {s_0}

derive reachable next states N = image(T_{exist}, R)

check fixed-point ? (if N is a subset of R)

R = R ∪ N

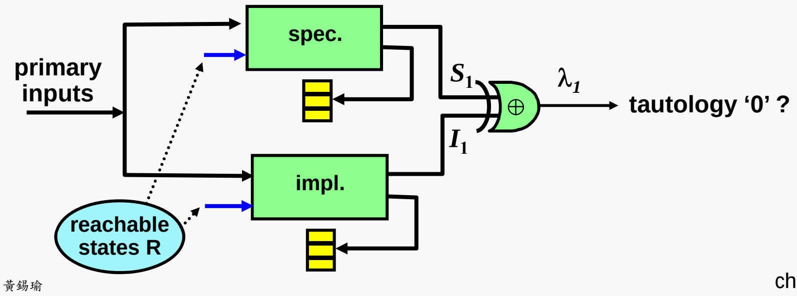
derive the BDD of each PO function λ_i

PO is tautology '0' for all input vectors and reachable states ?

Tautology Checking

- Notation
  - Let R be the reachable states derived from the FSM traversal
- Theorem
  - Two machines are equivalent if and only if

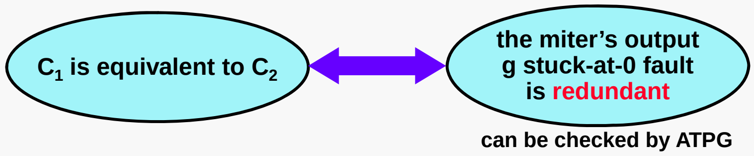
    $(\lambda_1+\lambda_2+\ldots+\lambda_n) \cdot$ **R is tautology ‘0’**



Why Incremental Verification ?

- Limitations of Symbolic Approaches
  - Could be time-consuming
  - Cannot handle larger design due to memory explosion
- In Practice
  - The two circuits under equivalence checking have a lot of structural similarity
- Idea of Incremental Verification
  - Exploring the structural similarity between the two circuits to speed up the verification process and to handle real large designs (e.g., multi-million gate-count design) [D. Brand 1993]

A Naïve ATPG-based Verification

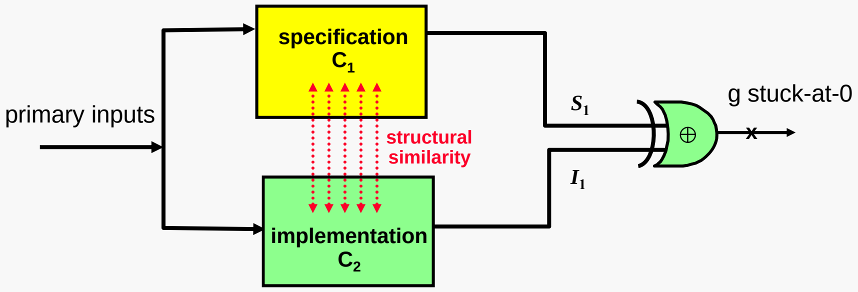


C_1 is equivalent to C_2

the miter's output g stuck-at-0 fault is redundant

can be checked by ATPG

Computational model called miter



specification C_1

implementation C_2

primary inputs

structural similarity

S_1

I_1

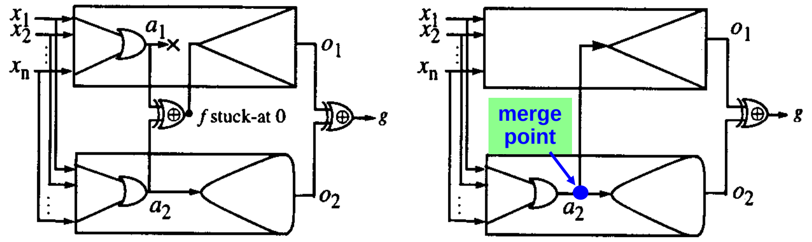
g stuck-at-0

Terminology

- Signal pair
  - (a_1, a_2) is called a signal pair if a_1 is from C_1 and a_2 is from C_2, or vice versa
- Equivalent signal pair
  - (a_1, a_2) is called an equivalent (signal) pair if the binary value of a_1 and a_2 in response to any input vector are identical
- Permissible signal pair
  - (a_1, a_2) is called a permissible (signal) pair if replacing a_1 by a_2 in the miter does not alter the output's functionality
  - Note that (a_1, a_2) is a permissible pair does not necessarily imply that (a_2, a_1) is also a permissible pair

Pruning Miter

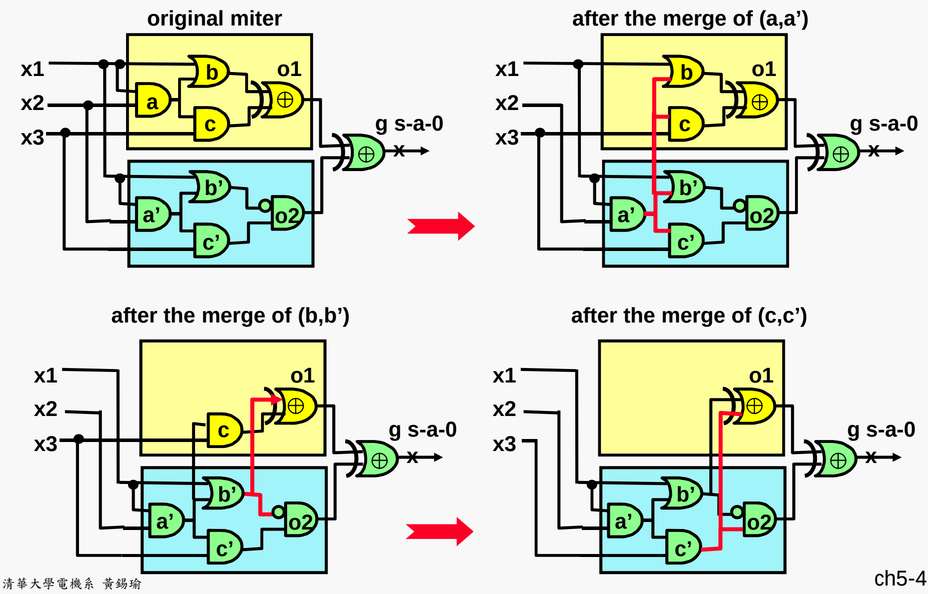
- Given a candidate permissible pair ($a_{1,}a_2$)
  - (1) check the permissibility by the model in Fig (a)
  - (2) If it sustains, replace a1 by a2
- The strategy is
  - merging internal permissible pairs first before checking the equivalence of an output pair (to improve efficiency)



(a) model for checking if ($a_{1,}$ a_2) is permissible.

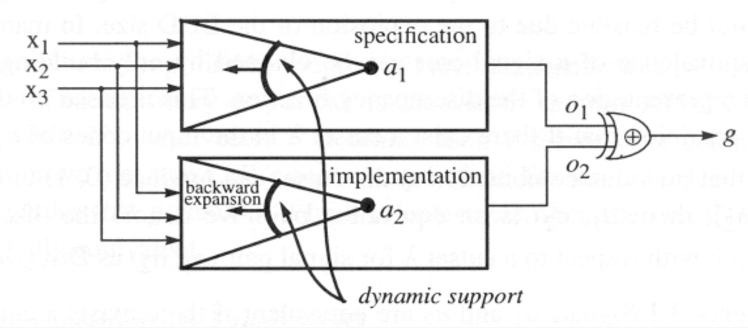
(b) replace a_1 by a_2.

Example: Incremental Verification



Enhancement by Using Local BDD

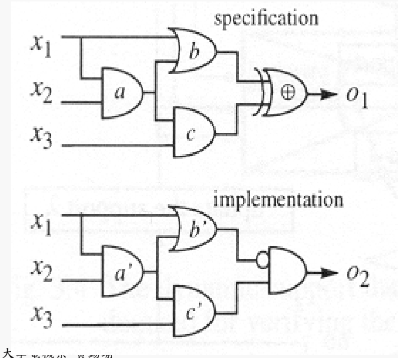
- Local BDD
  - is a BDD taking certain internal signals, instead of the primary inputs, as the supporting variables
- The concept of dynamic support

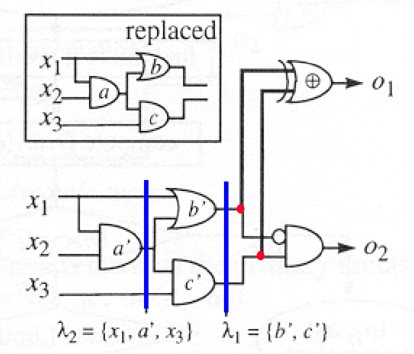


The dynamic support expands towards the PI's on demand as verifying the equivalence of (o_1,o_2)

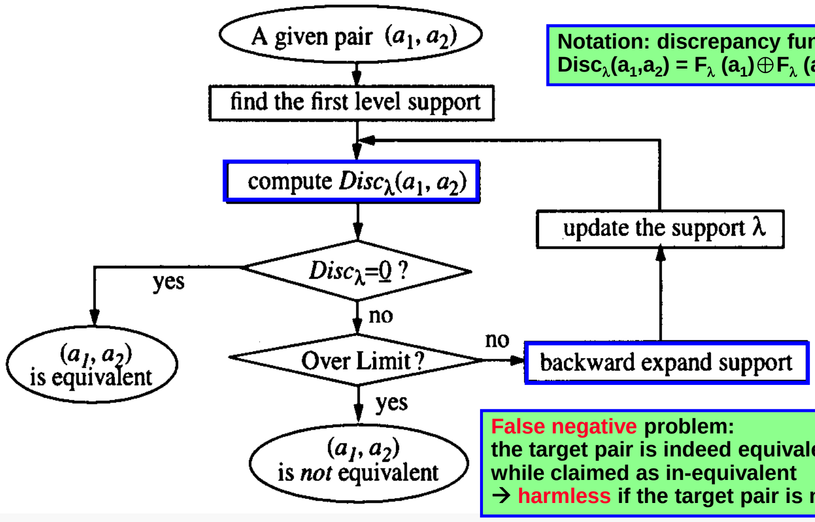
Example: Incremental Verification Using Local BDD

- First support λ_1={b',c'}
  - → The local BDDs of o_1 and o_2 in terms of λ_1 is NOT equivalent
  - → expand the support towards PI's
- Second support λ_2={x_1,a',x_3}
  - → The local BDDs of o_1 and o_2 in terms of λ_2 is equivalent
  - → Conclude that (o_1,o_2) is equivalent





Routine of Equivalence Checking Using Local BDD



Notation: discrepancy function
$Disc_\lambda(a_1,a_2) = F_\lambda(a_1) \oplus F_\lambda(a_1)$

False negative problem:
the target pair is indeed equivalent while claimed as in-equivalent
→ harmless if the target pair is not PO

Conclusions

Formal Method is fantastic when it works.

But it could fail badly when it does not.

It is all about Boolean reasoning

Good luck on finding your own applications …

References

1. **R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. on Computers*, vol. C-35, Aug. 1986, pp. 677-691.**
2. **E.M. Clark, O. Grumberg, and D. Peled, "Model Checking", 2000**
3. **S.-Y. Huang and K.-T. Cheng, "Formal Equivalence Checking and Design Debugging," Kluwer Academic Publishers, 1998.**

清華大學 EE 5265
積體電路設計自動化

單元 6
Floorplanning

教育部顧問室
「超大型積體電路與系統設計」教育改進計畫
EDA聯盟 – 推廣課程

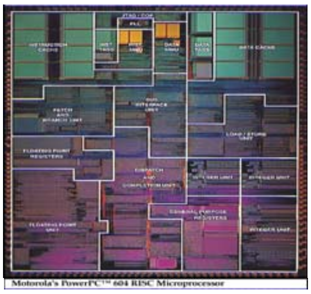
致謝

本單元之教材主要取自於
教育部
超大型積體電路與系統設計教育改進計畫
EDA聯盟之課程發展成果

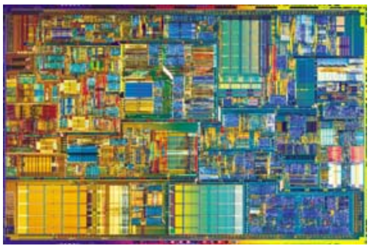
- (教材編纂小組成員)
- 台灣大學電機系 張耀文
- 清華大學電機系 黃錫瑜
- 交通大學資科系 李毅郎
- 中央大學電機系 劉建男
- 元智大學資工系 林榮彬

Outline of Floorplanning

- Contents
  - (1) Basics of Floorplanning
  - (2) Slicing Floorplanning
  - (3) Non-Slicing Floorplanning



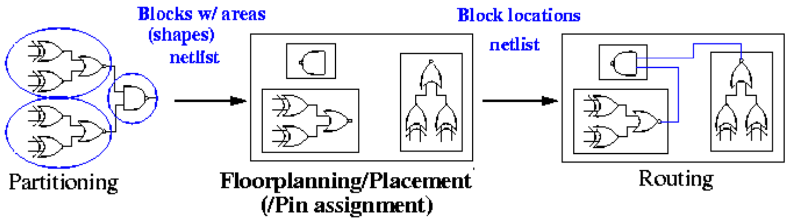
PowerPC 604



Pentium 4

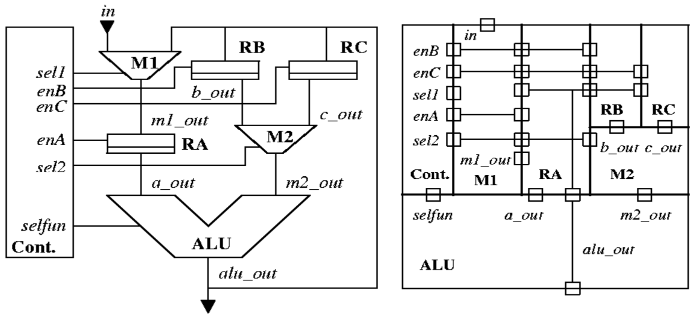
Floorplanning

- Floorplanning leads to
  - Well-defined blocks in terms of the physical structures
- Block Types
  - Hard or rigid blocks: with defined areas and shapes
  - Soft or flexible blocks: with approximate areas, undefined shapes
- Objectives
  - Find locations for all blocks
  - Report shapes of soft block and pin locations of all the blocks



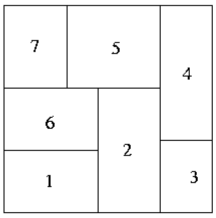
Why Floorplanning?

- Main Purpose of Floorplanning
  - (1) To implement the top-down design strategy
  - (2) To decide the shape and terminals of each soft block
  - (3) For rough estimation of the wiring delays

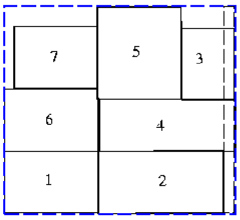


Floorplanning Problem

- Inputs to the floorplanning problem:
  - A set of blocks, hard or soft.
  - Pin locations of hard blocks.
  - A netlist.
- Objectives:
  - minimize area, reduce wire length for (critical) nets, maximize routability (minimize congestion), determine shapes of soft blocks, etc.

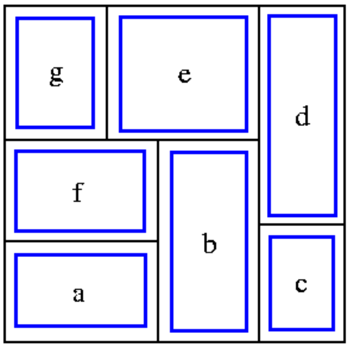


An optimal floorplan, in terms of area

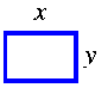


A non–optimal floorplan

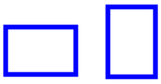
Floorplan Design



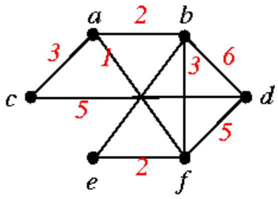
- *Modules:*



- *Area: A=xy*
- *Aspect ratio: r <= y/x <= s*
- *Rotation:*



- *Module connectivity*



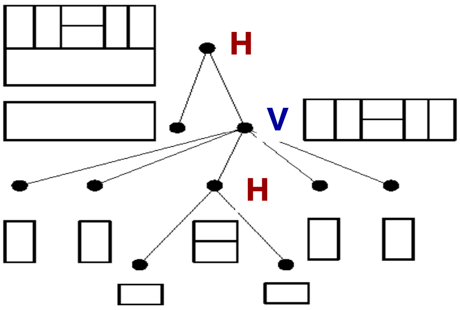
Representing Floorplan As a Tree

Three Types of Nodes

(1) H-node: horizontal cut
- **Left sub-tree is the bottom half**
- **Right sub-tree is the top half**

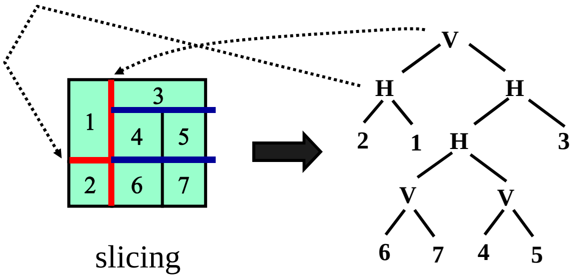
(2) V-node: vertical cut
- **Left sub-tree is the left half**
- **Right sub-tree is the right half**

(3) Leaf node: a basic block



Slicing Floorplan

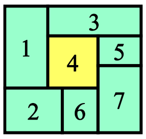
- **Slicing structure:**
  - A rectangular dissection that can be obtained by repetitively subdividing rectangles horizontally or vertically.
- **Slicing tree:**
  - A binary tree, where each internal node represents a vertical cut line or horizontal cut line, and each leaf a basic rectangle.



slicing floorplan

slicing tree

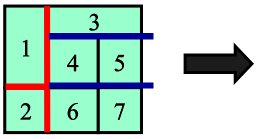
wheel structure



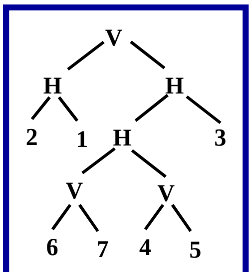
non-slicing floorplan

Skewed Slicing Tree

- **Problem**: There might be multiple trees for a floorplan !
- **Skewed slicing tree**: (Desired)
  - One in which no node and its right child are the same.

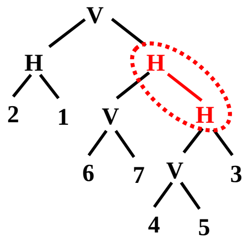


slicing floorplan



A slicing tree (Skewed)

OR



Another slicing tree (Non-Skewed)

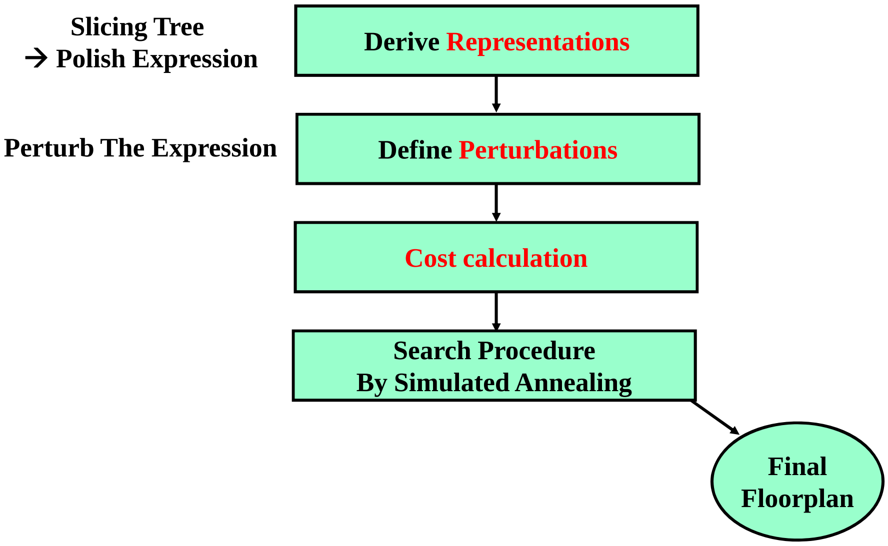
Outline

- Basics of Floorplanning
- Slicing Floorplanning
  - Normalized Polish Expression
  - Simulated Annealing Formulation
  - Block Shaping Problem
- Non-Slicing Floorplanning
  - Simulated Annealing Formulation

Slicing Floorplan Design by Simulated Annealing

- Related work
  - (1) Wong & Liu, "A new algorithm for floorplan design," DAC-86.
    - Considers slicing floorplans.
  - (2) Wong & Liu, "Floorplan design for rectangular and L-shaped modules," ICCAD'87.
    - Also considers L-shaped modules.
  - (3) Wong, Leong, Liu, *Simulated Annealing for VLSI Design*, pp. 31--71, Kluwer Academic Publishers, 1988.
- Ingredients to simulated annealing
  - solution space?
  - neighborhood structure?
  - cost function?
  - annealing schedule?

Overall Strategy



Slicing Tree
→ Polish Expression

Derive Representations

Perturb The Expression

Define Perturbations

Cost calculation

Search Procedure
By Simulated Annealing

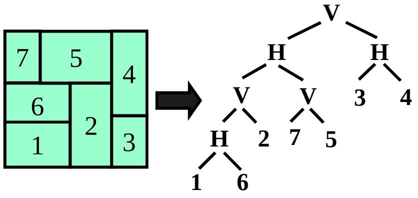
Final Floorplan

Polish Expression

- Definition of Polish Expression
  - An expression $E = e_1 e_2 \ldots e_{2n-1}$, where $e_i \in \{1, 2, \ldots, n, H, V\}$, $1 \le i \le 2n-1$
  - (1) Every operand j, $1 \le j \le n$, appears exactly once in E;
  - (2) (The Balloting Property) For every sub-expression $E_i = e_1 \ldots e_i$, $1 \le i \le 2n-1$, no. of operands > no. of operators

1 6 H 3 5 V 2 H V 7 4 H V

# of operands = 4 .... = 7
# of operators = 2 .... = 5



Post-order Traversal

E=16H2V75VH34HV
Or
E=16+2*75*+34+*

ijH*: rectangle *i* on bottom of *j
***ijV*: rectangle *i* on the left of *j*.**

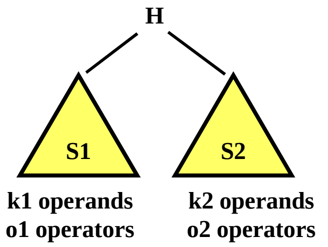
Why Balloting Property?

- Balloting property
  - Operands should outnumber operators
  - To guarantee a valid post-order traversal of a slicing tree

S has two sub-trees, S1 and S2,
Both S1 and S2 satisfy the balloting property
➔ Then, the entire tree satisfies the balloting property as well

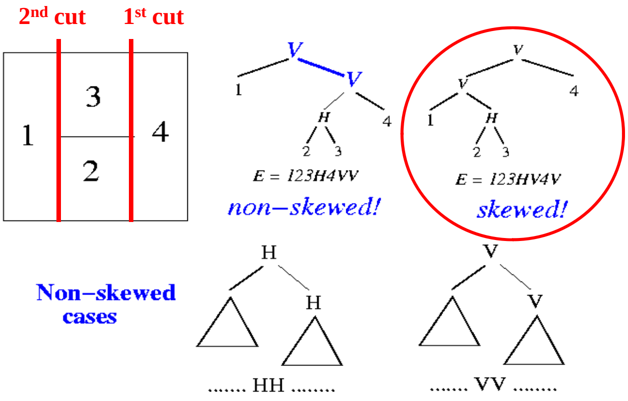
The total number of operands: (k1 + k2)
The total number of operators: (o1 + o2 + 1)

Since (k1 >= o1+1) and (k2 >= o2+1)
So, (k1 + k2) >= (o1 + o2 + 2)
➔ I.e., (k1 + k2) > (o1 + o2 + 1)
➔ Operands outnumber operators



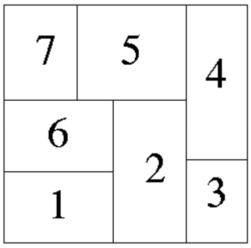
Redundant Representations

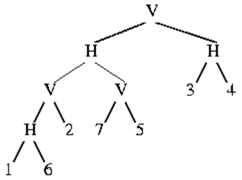
- Problem:
  - One floorplan could correspond to multiple slicing tree representations !
- Solution: Give specific orders to consecutive cuts
  - (1) Consecutive H-cuts: ordered from right to left
  - (2) Consecutive V-cuts: ordered from top to bottom



Normalized Polish Expression

- **Definition of Normalized Polish Expression**
  - A Polish expression $E = e_1\ e_2\ \ldots\ e_{2n-1}$ is called normalized iff *E* has no consecutive operators of the same type (*H* or *V*)
- **A Normalized Polish Expression**
  - Corresponds to an unique rectangular slicing structure



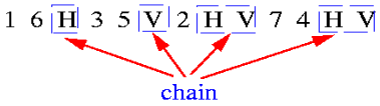


$E = 16H2V75VH34HV$

A normalized Polish expression

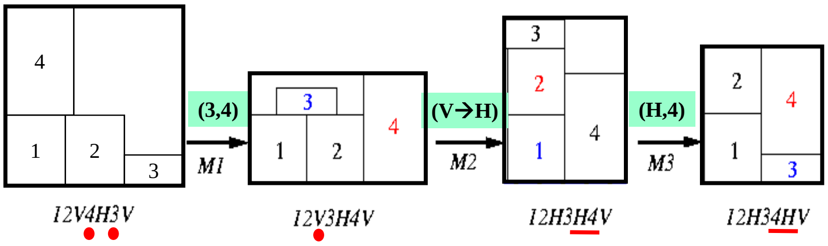
Neighborhood Structure and Perturbation

- **Chain**: *HVHVH* … or *VHVHV* …



- **Adjacency Relations**
  - 1 and 6 are adjacent operands; 2 and 7 are adjacent operands; 5 and *V* are adjacent operand and operator
- **Three Types of Perturbations**
  - **M1 (Operand Swap):**
    - Swap two adjacent operands
  - **M2 (Chain Invert):**
    - Complement some chain (V = H, H = V)
  - **M3 (Operator/Operand Swap):**
    - Swap two adjacent operand and operator

Effects of Perturbation



- **Keep The balloting property during the moves**
  - **(1) M1 and M2 moves are OK**
  - **(2) Look out for the *M3* moves!**
    - **Reject illegal *M3* moves if necessary**

Validation of Operand-Operator Swap (M3)

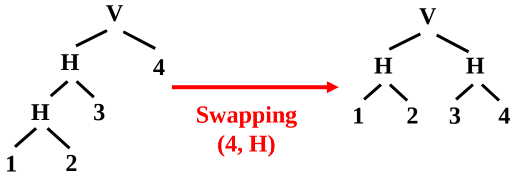
- **Validation check of M3 moves:**
  - **Assume the swapping of operand e_i with the operator e_{i+1}, $1 \le i \le k-1$**
  - **N_k is no. of operators in Polish expression $E = e_1\ e_2 \ldots e_k$, $1 \le k \le 2n-1$**
  - **Then, the swap will not violate the balloting property iff $2N_{i+1} < i$**

(Polish Expression) **12H3H4V** → **12H34HV**

Can we swap 2 & H?
➔ NO

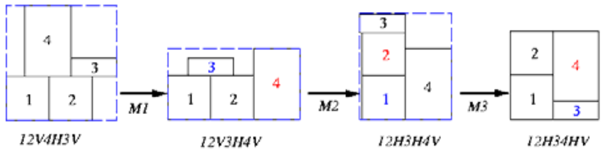
(swap e_5 and e_6) ➔ i = 5 ➔ N_{5+1} = 2 ➔ $2N_{5+1}$ = 2*2 = 4 < i ➔ legal move !

(Slicing Tree)

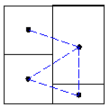


Cost Function

- $\phi = A + \lambda W$
  - *A*: area of the smallest rectangle
  - *W*: overall wiring length
  - λ : user-specified parameter

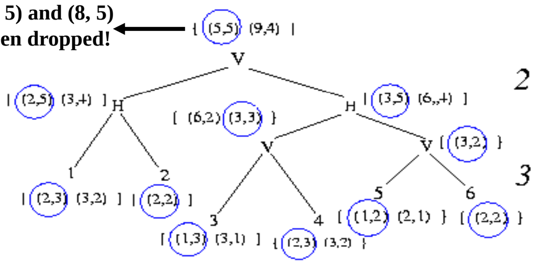


- **Wire Length Estimation**: $W= \sum_{ij} c_{ij}\, d_{ij}$
  - c_{ij}: # of connections between blocks *i* and *j*.
  - d_{ij}: center-to-center distance between basic rectangles *i* and *j*.

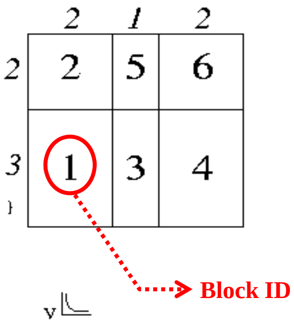


Area Computation for Hard Blocks

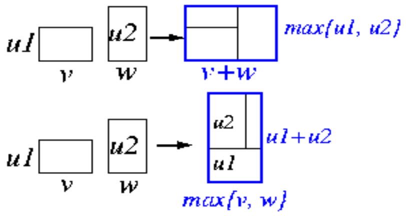
- Take rotation into consideration

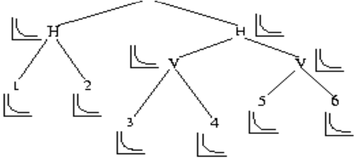


Note (6, 5) and (8, 5) have been dropped!



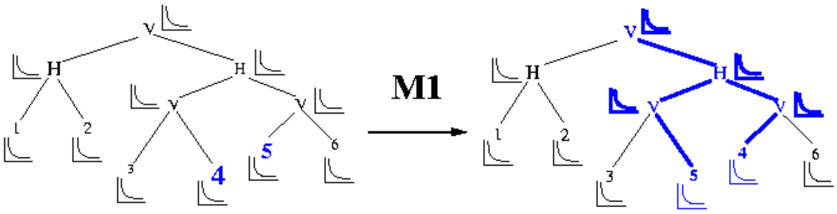
Block ID





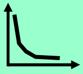
Incremental Computation of Cost Function

- The cost change due to a move
  - Can be estimated incrementally
  - By updating at most two paths of the slicing tree



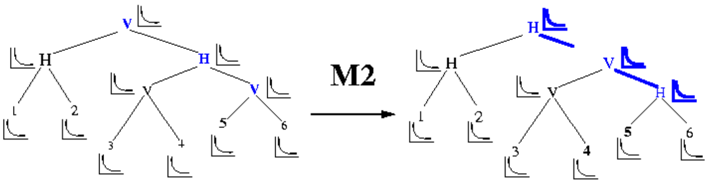
E = 12H34V56VHV

E = 12H35V46VHV



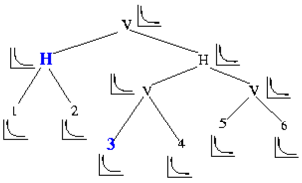
Shape function of a soft block under area constraint

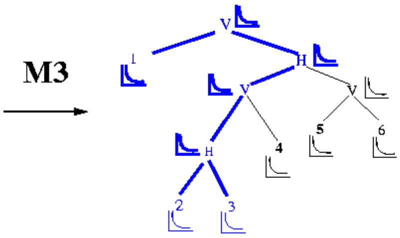
Incremental Computation of Cost Function (cont'd)



E = 12H34V56VHV

E = 12H34V56HVH



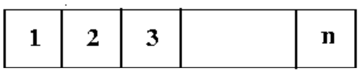


E = 12H34V56VHV

E = 123H4V56VHV

Annealing Schedule

- **Initial solution**: **12*V*3*V* … *nV*.**



- **Temperature Cooling:**
  - *T*i = r^i T_0, *i* = 1, 2, 3, …; *r* =0.85.
- **Perturbations**
  - At each temperature, try *kn* moves (*k* = 5-10).
- **Terminating Conditions**
  - (1) Number of accepted moves < 5%, or
  - (2) Temperature is low enough, or
  - (3) Run out of time

Algorithm: Wong-Liu (*P*, ε, *r*, *k*)

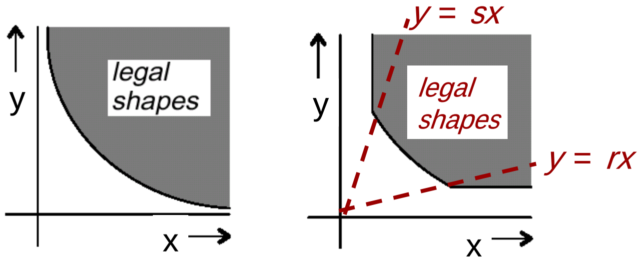
```
E ← 12V3V4V … nV; /* initial solution */
Best ← E; T_0 ← $\frac{\Delta_{avg}}{ln(P)}$; M ← MT ← uphill ← 0; N = kn;
repeat
  MT ← uphill ← reject ← 0;
  repeat
    SelectMove(M);
    Case M of
    M_1:  Select two adjacent operands e_i and e_j; NE ← Swap(E, e_i, e_j);
    M_2:  Select a nonzero length chain C; NE ← Complement(E, C);
    M_3:  done ← FALSE;
       while not (done) do
           Select two adjacent operand e_i and operator e_{i+1};
           if (e_{i-1} ≠ e_{i+1})  and (2 N_{i+1} < i) then done ← TRUE;
           Select two adjacent operator e_i and operand e_{i+1};
           if (e_i ≠ e_{i+2}) then done ← TRUE;
       NE ← Swap(E, e_i, e_{i+1});
    MT ← MT+1; Δcost ← cost(NE) - cost(E);
    if (Δcost ≤ 0) or (Random < $e^{\frac{-\Delta cost}{T}}$ )
     then
         if (Δcost > 0) then uphill  ← uphill + 1;
         E  ← NE;
         if  cost(E) < cost(best) then best ← E;
      else reject ← reject + 1;
   until (uphill > N) or (MT > 2N);
   T ← rT; /* reduce temperature */
until (reject/MT > 0.95) or (T < ε) or OutOfTime;
```

Outline

- Basics of Floorplanning
- Slicing Floorplanning
  - Normalized Polish Expression
  - Simulated Annealing Formulation
  - Block Shaping Problem
- Non-Slicing Floorplanning
  - Simulated Annealing Formulation

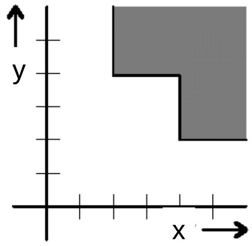
Shape Curve

- Soft blocks could have different aspect ratios.
- The shape function is a hyperbola:
  - xy=A, with the width *x* and the height *y*
- In practice,
  - Very thin blocks are often not feasible to design.
  - The shape function is a hyperbola constrained by two lines
  - Aspect ratio: *r* <= *y/x* <= *s*.



Discrete Shape Curve

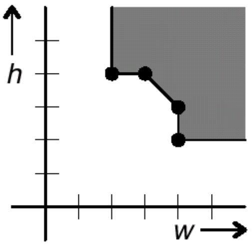
- **Leaf cells are built from discrete transistors:**
  - it is not realistic to assume that the shape function follows the hyperbola continuously.
- **In an extreme case, a cell is rigid:**
  - it can only be rotated and mirrored during floorplanning or placement.



The shape function of a 2 × 4 inset cell.

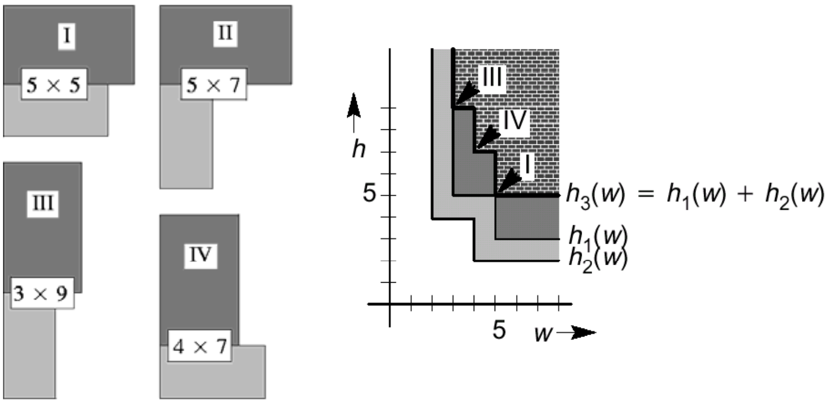
Piecewise Linear Shape Curve

- **In general, a *piecewise linear* function can be used to approximate any shape function.**
- **The points where the function changes its direction, are called the corner *(break) points* of the piecewise linear function.**



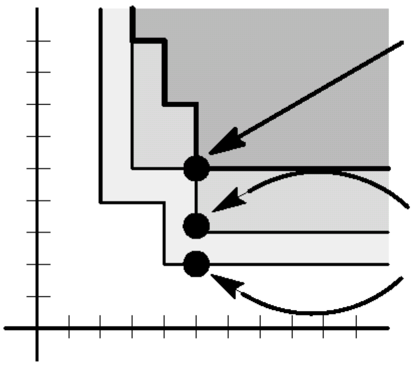
Composition Rules for Vertical Abutment

- Composition by vertical abutment ➔ the addition of shape functions.



Deriving Shapes of Children

- A choice for the minimal shape of composite cell fixes the shapes of its children cells.



minimal area of parent

consequences for children's shapes

Shaping Procedure of Slicing Floorplans

Shaping procedure is performed on a slicing tree
➔ To decide the shape of each basic block.

start →

Represent shape functions of each basic block
As piecewise linear functions

↓

To derive the shape function of each composite block
(Bottom-up traversal)

↓

Decide the shape of the top block
(only break points of shape function need to be evaluated)

↓

Propagate the consequences down to the leaf blocks
(Top-down propagation)

↓

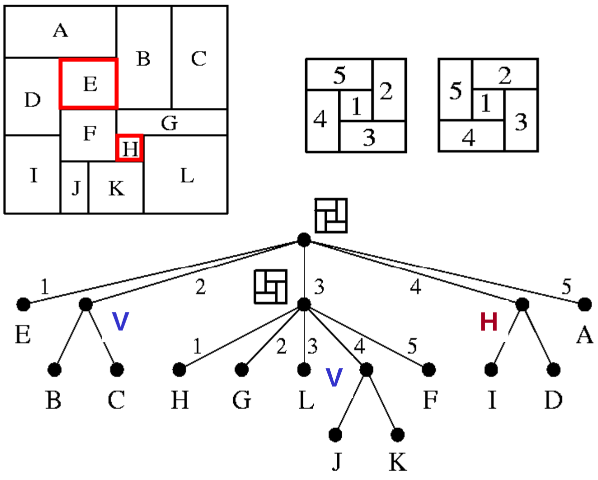
Derive the final shape of each block

Outline

- **Basics of Floorplanning**
- **Slicing Floorplanning**
  - **Normalized Polish Expression**
  - **Simulated Annealing Formulation**
  - **Block Shaping Problem**
- **Non-Slicing Floorplanning**
  - **Simulated Annealing Formulation**

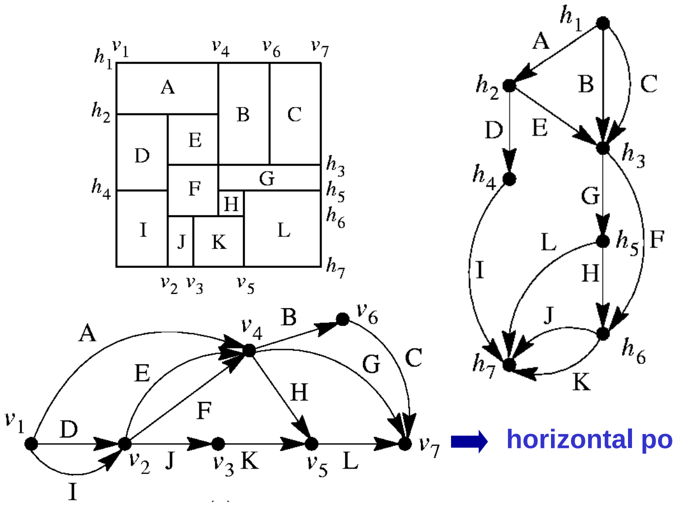
Order-of-5 Floorplan Examples

Wheel (or spiral) floorplan include a wheel structure as a basic block
➔ Could lead to an even better result than a pure slicing floorplan
➔ an order-of-5 floorplan, I.e., a node could have five children in the tree



General Floorplan Representation: Polar Graphs

- **vertex: channel segment (or boundary)**
- **edge (weight): cell/block/module (dimension)**



vertical polar graph

horizontal polar graph

Concluding Remarks

- Floorplanning Strategy
  - Representation ➔ Cost Calculation ➔ Perturbation Scheme
- Slicing Tree
  - Normalized Polish Expression
- Non-Slicing Tree
  - Polar Graph

It Is The Floorplan That
Shapes The Landscape of Your IC.

清華大學 EE 5265

積體電路設計自動化

單元 7

Placement and Partitioning

教育部顧問室
「超大型積體電路與系統設計」教育改進計畫
EDA聯盟 – 推廣課程

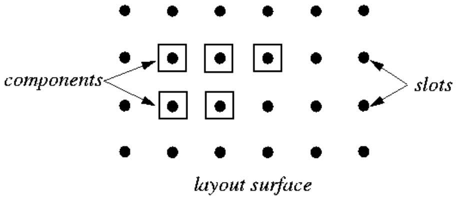
致謝

本單元之教材主要取自於
教育部
超大型積體電路與系統設計教育改進計畫
EDA聯盟之課程發展成果

- **(教材編纂小組成員)**
- 台灣大學電機系 張耀文
- 清華大學電機系 黃錫瑜
- 交通大學資科系 李毅郎
- 中央大學電機系 劉建男
- 元智大學資工系 林榮彬

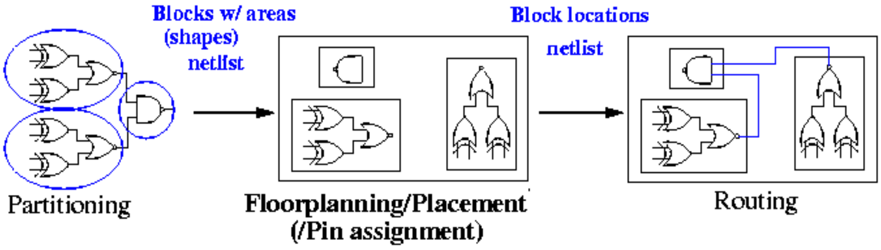
Outline of Placement

- Course contents:
  - Placement metrics
  - Placement
    - Clustering-Based, Partitioning-Based, Force-Directed, Simulated-Annealing, Genetic Algorithm
  - Partitioning
    - Kernighang-Lin Partitioning Algorithm
    - Simulated-Annealing Based Partitioning



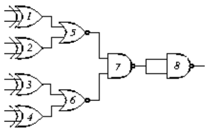
Placement

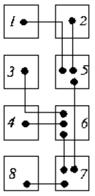
- Placement
  - is to automatically assign pre-designed cells to correct positions on the chip, so as to minimize certain criteria
- Inputs: A set of fixed cells/modules, a netlist.
- Quality metrics:
  - Routability, Channel Density, Wire-Length, cut size, performance, thermal issues, I/O pads.



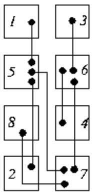
Placement Objectives and Constraints

- **What does a placement algorithm try to optimize?**
  - the total area
  - the total wire length
  - the number of horizontal/vertical wire segments crossing a line
- **Constraints:**
  - the placement should be routable (no cell overlaps; no density overflow).
  - timing constraints are met (some wires should always be shorter than a given length).

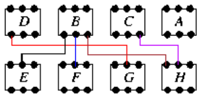




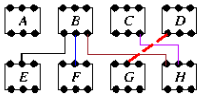
wirelength = 10



wirelength = 12



Density = 2 (2 tracks required)

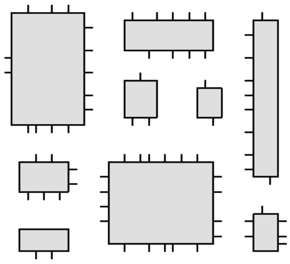


Shorter wirelength, 3 tracks required.

Placement Styles

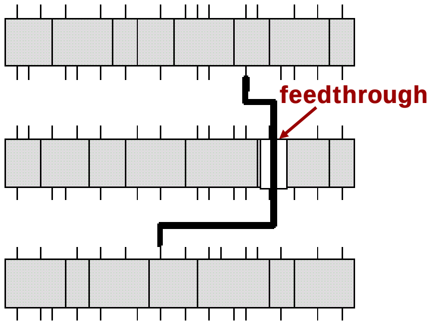
- **Building-Block Placement**
  - The cells to be placed have arbitrary shapes
- **Standard-Cell Placement**
  - Cells are to be placed in rows
- **Gate-Level Placement**
  - Cells are mapped into pre-fabricated logic blocks

Building-block placement

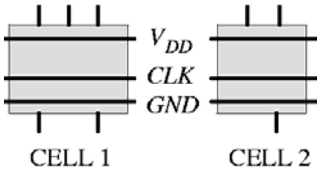


Standard-Cell Placement

- Standard cells are designed in such a way that
  - power and clock connections run horizontally through the cell and other I/O leaves the cell from the top or bottom sides.
- Sometimes **feedthrough** cells are added to ease wiring.

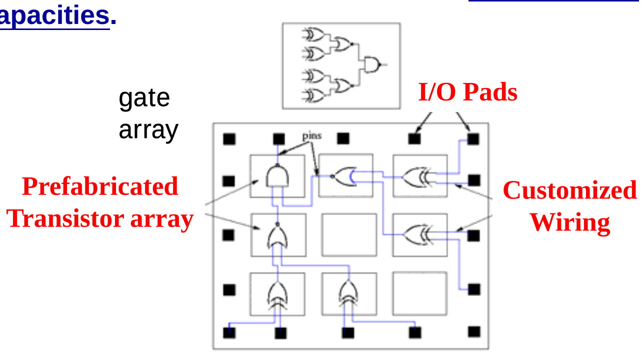


Routing of logistic signals



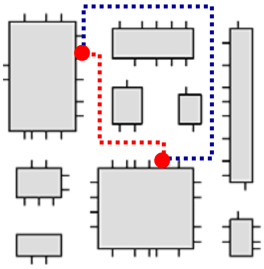
Consequences of Fabrication Method

- Full-custom fabrication (building block):
  - Free selection of aspect ratio (quotient of height and width).
  - Height of wiring channels can be adapted if necessary.
- Semi-custom fabrication (gate array, standard cell):
  - Placement has to deal with fixed carrier dimensions.
  - Placement should be able to deal with fixed channel capacities.



Relation with Routing

- **Ideally,**
  - **placement and routing should be performed simultaneously as they depend on each other's results. This is, however, too complicated.**
  - **P&R: placement and routing**
- **In practice,**
  - **placement is done prior to routing. The placement algorithm estimates the wire length of a net using some *metric*.**



(Wire Length Estimation)
Input: the multiple pins of a net
Output: estimation of the length

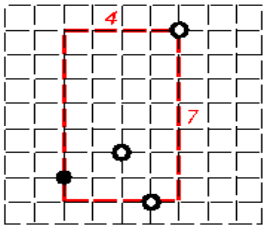
Estimation of Wirelength

- **Semi-perimeter method:**
  - **Half the perimeter of the bounding rectangle that encloses all the pins of the net to be connected. Most widely used approximation!**
- **Squared Euclidean distance:**
  - **Squares of all pairwise terminal distances in a net using a quadratic cost function**

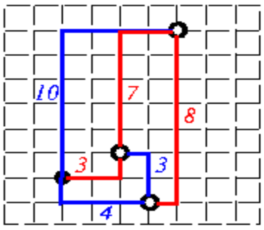
$$\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\gamma_{ij}[(x_i - x_j)^2 + (y_i - y_j)^2]$$

- **Steiner-tree approximation:**
  - **Computationally expensive.**
- **Minimum spanning tree:**
  - **Good approximation to Steiner trees.**

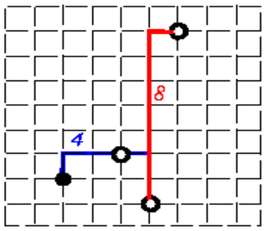
Estimation of Wirelength (cont'd)



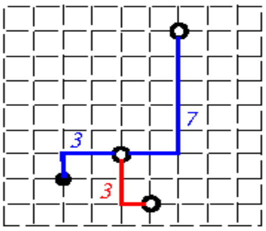
semi–perimeter len = 11



*complete graph len * 2/n = 17.5*



Steiner tree len = 12



Spanning tree len = 13

Placement Algorithms

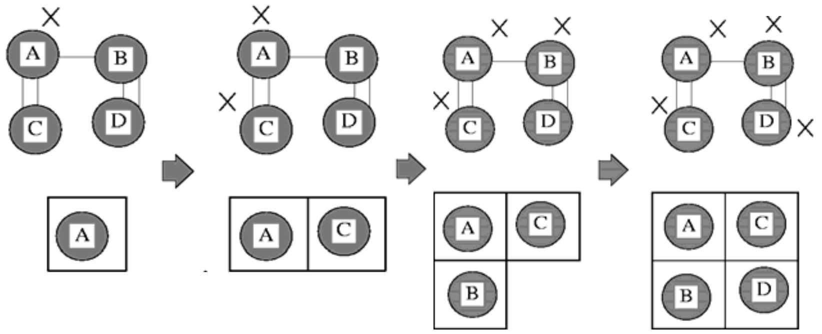
- **The placement problem is NP-complete**
- **Popular placement algorithms:**
  - **Constructive algorithms: once the position of a cell is fixed, it is not modified anymore.**
    - **Clustering-based, Partition-based**
  - **Iterative algorithms: intermediate placements are modified in an attempt to improve the cost function.**
    - **Force-directed method, etc**
  - **Non-deterministic approaches:**
    - **Simulated annealing, genetic algorithm, etc.**
- **Most approaches combine multiple elements:**
  - **(1) Constructive algorithms are used to obtain an initial placement.**
  - **(2) The initial placement is refined by an iterative improvement phase.**
  - **(3) The results can further be improved by simulated annealing.**

Outline

- Placement metrics
- Placement
  - Clustering-Based
  - Partitioning-Based
  - Force-Directed
  - Simulated-Annealing
  - Genetic Algorithm
- Partitioning
  - Kernighang-Lin Partitioning Algorithm
  - Simulated-Annealing Based Partitioning

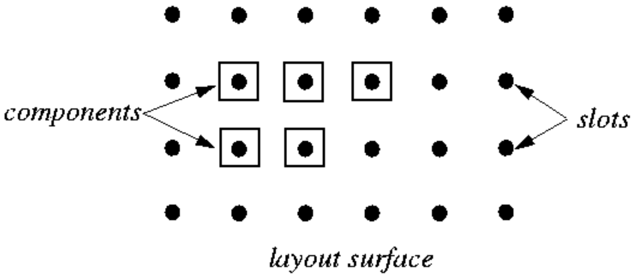
Bottom-Up Placement: Clustering

- Starts with a single cell and finds more cells that share nets with it.



Clustering-Based Placement

- Greedy method: Selects unplaced components and places them in available slots.
  - (1) SELECT: Choose the unplaced component that is most strongly connected to all of the placed components (or most strongly connected to any single placed component).
  - (2) PLACE: Place the selected component at a slot such that a certain “cost” of the partial placement is minimized.

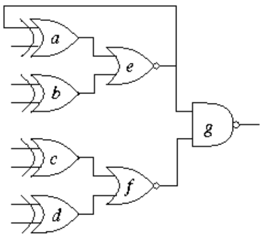


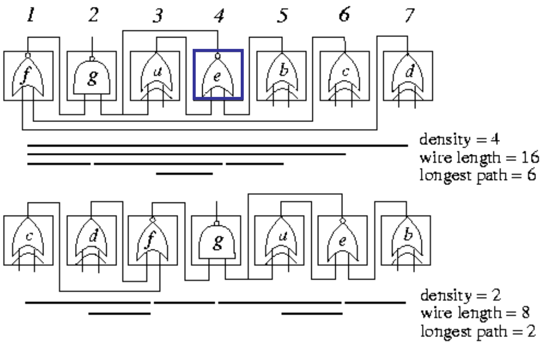
Example: Clustering-Based Placement

- Connectivity degree of each cell
  - c_a=3, c_b=1, c_c=1, c_d=1, c_e=4, c_f=3, and c_g=3
  - ➔ *e* has the most connectivity.
- Place *e* in the center, slot 4. *a, b, g* are connected to *e* ➔ Place *a* next to *e* (say, slot 3). Continue with other cells
- Further improve the placement by swapping the gates.

connectivity

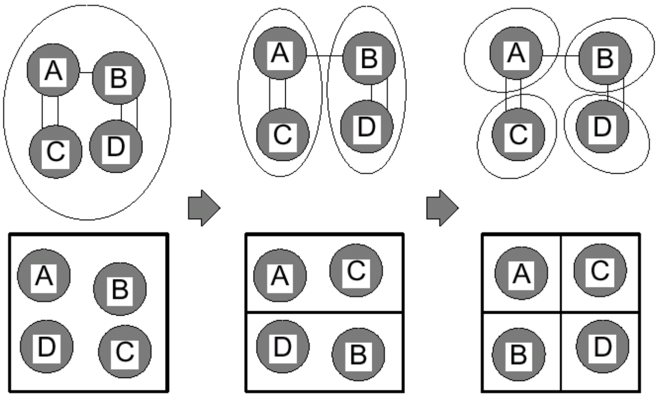
$\hat{c}_{ae} = 2, \hat{c}_{be} = \hat{c}_{eg} = 1$





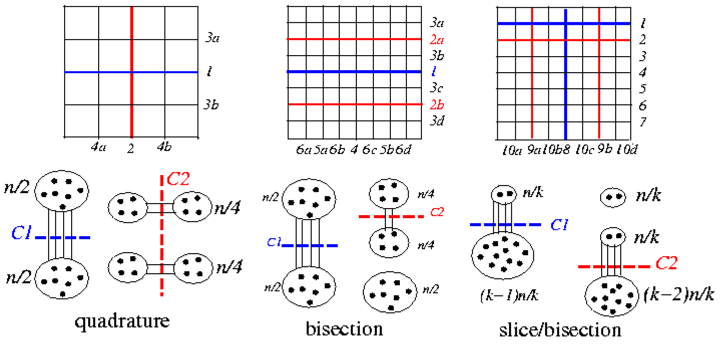
Top-down Placement: Partitioning-Based

- **Starts with the whole circuit and ends with small circuits.**
- **<u>Recursive Bi-partitioning</u> of a circuit leads to a min-cut placement.**



Partitioning-Based (or Min-Cut) Placement

- **Breuer**
  - **"A class of min-cut placement algorithms," DAC-77.**
- **Partition-Based Placement**
  - **Quadrature**
  - **Bisection**
  - **Slice / Bisection**

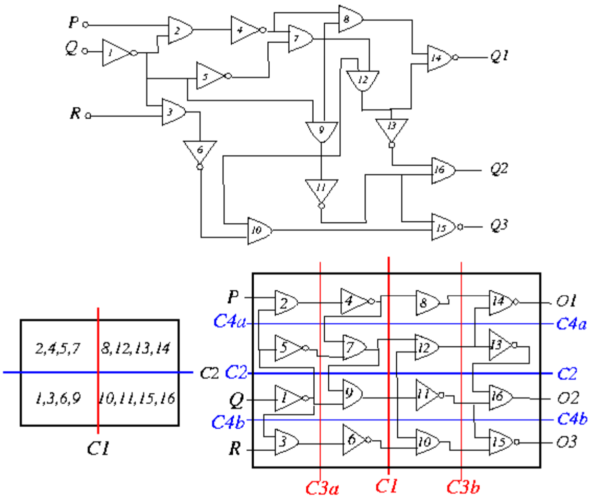


Algorithm for Min-Cut Placement

```
Algorithm: Min_Cut_Placement($N$, $n$, $C$)
/* $N$: the layout surface */
/* $n$ : no. of cells to be placed */
/* $n_0$: no. of cells in a slot */
/* $C$: the connectivity matrix */

begin
if ($n \le n_0$) then PlaceCells($N$, $n$, $C$)
else
    ($N_1$, $N_2$) ← CutSurface($N$);
    ($n_1$, $C_1$), ($n_2$, $C_2$) ← Partition($n$, $C$);
 Call Min_Cut_Placement($N_1$, $n_1$, $C_1$);
 Call Min_Cut_Placement($N_2$, $n_2$, $C_2$);
end
```

Quadrature Placement Example

- **K-L heuristic to partition + Quadrature Placement: Cost C_1 = 4, C_{2L}= C_{2R} = 2, etc.**



Outline

- **Placement metrics**
- **Placement**
  - **Clustering-Based**
  - **Partitioning-Based**
  - → **Force-Directed**
  - **Simulated-Annealing**
  - **Genetic Algorithm**
- **Partitioning**
  - **Kernighang-Lin Partitioning Algorithm**
  - **Simulated-Annealing Based Partitioning**

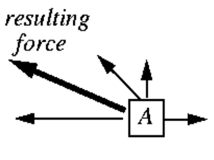
General Procedure for Iterative Improvement

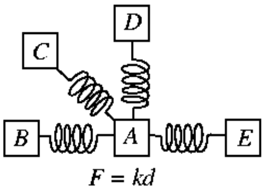
```
Algorithm: Iterative_Improvement()
begin
s ← initial_configuration();
c ← cost(s);
while (not stop()) do
    s' ← perturb(s);
    c' ← cost(s');
    if (accept(c, c'))
    then   s ← s';
end
```

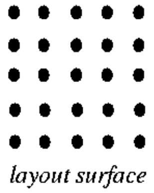
A mechanism that affect the results only slightly

Placement by the Force-Directed Method

- **Hanan & Kurtzberg**,
  - "Placement techniques," in *Design Automation of Digital Systems*, Breuer, Ed, 1972.
- **Quinn, Jr. & Breuer**,
  - "A force directed component placement procedure for printed circuit boards," *IEEE Trans. Circuits and Systems*, June 1979.
- Force-Directed Method:
  - Reduce the placement problem to solving a set of simultaneous linear equations to determine equilibrium locations for cells.
- Analogy to Hooke's law:
  - $F = kd$, F: force, k: spring constant, d: distance.







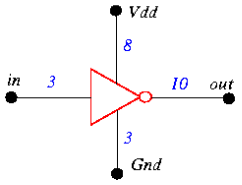
Finding the Zero-Force Location

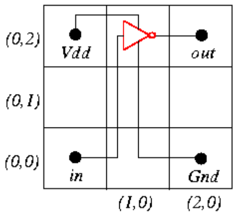
- Cell i connects to several cells j's at distances d_{ij}'s by wires of weights w_{ij}'s. Total force: $F_i = \sum_j w_{ij} d_{ij}$
- The zero-force location ($\hat{x_i}$, $\hat{y_i}$) can be determined by equating the x- and y-components of the forces to zero:

$$\sum_j w_{ij} \cdot (x_j - \hat{x_i}) = 0 \quad \Rightarrow \quad \hat{x_i} = \frac{\sum_j w_{ij} x_j}{\sum_j w_{ij}}$$

$$\sum_j w_{ij} \cdot (y_j - \hat{y_i}) = 0 \quad \Rightarrow \quad \hat{y_i} = \frac{\sum_j w_{ij} y_j}{\sum_j w_{ij}}$$

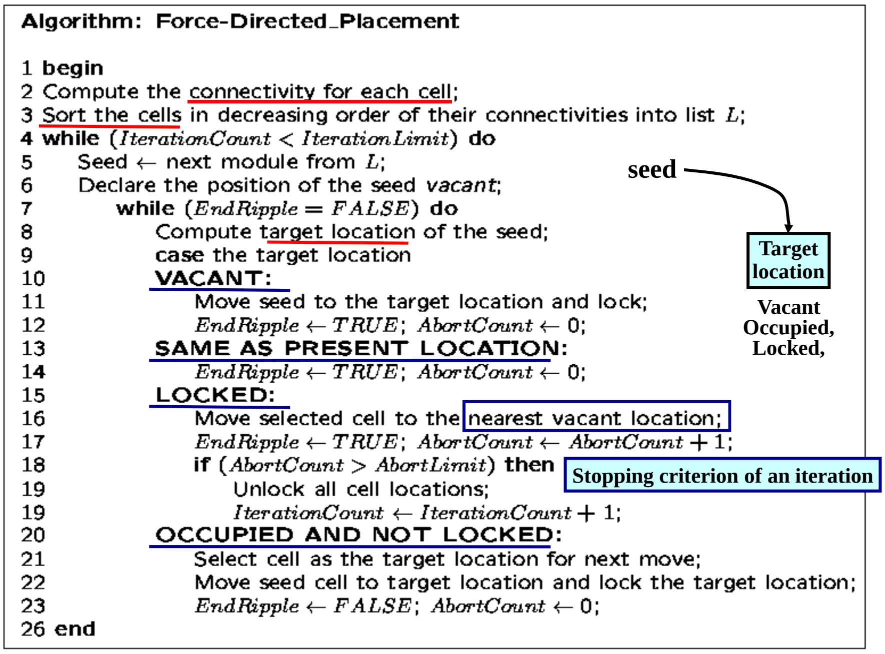
- In the example, $\hat{x_i} = \frac{8\times0+10\times2+3\times0+3\times2}{8+10+3+3} = 1.083$ and $\hat{y_i}$ = 1.50.





Force-Directed Placement

- Can be constructive or iterative:
  - Start with an initial placement.
  - Select a "most profitable" cell *p* (e.g., maximum *F*, critical cells) and place it in its zero-force location.
  - "Fix" placement if the zero-location has been occupied by another cell *q*.
- Popular options to fix:
  - Ripple move: place *p* in the occupied location, compute a new zero-force location for *q*, …
  - Chain move: place *p* in the occupied location, move *q* to an adjacent location, …
  - Proximity Move: place *p* to a free location close to *q*.



```
Algorithm: Force-Directed_Placement

begin
Compute the connectivity for each cell;
Sort the cells in decreasing order of their connectivities into list $L$;
while ($IterationCount < IterationLimit$) do
    Seed ← next module from $L$;
    Declare the position of the seed vacant;
        while ($EndRipple = FALSE$) do
            Compute target location of the seed;
            case the target location
            VACANT:
                Move seed to the target location and lock;
                $EndRipple \leftarrow TRUE$; $AbortCount \leftarrow 0$;
            SAME AS PRESENT LOCATION:
                $EndRipple \leftarrow TRUE$; $AbortCount \leftarrow 0$;
            LOCKED:
                Move selected cell to the nearest vacant location;
                $EndRipple \leftarrow TRUE$; $AbortCount \leftarrow AbortCount + 1$;
                if ($AbortCount > AbortLimit$) then
                    Unlock all cell locations;
                    $IterationCount \leftarrow IterationCount + 1$;
            OCCUPIED AND NOT LOCKED:
                Select cell as the target location for next move;
                Move seed cell to target location and lock the target location;
                $EndRipple \leftarrow FALSE$; $AbortCount \leftarrow 0$;
end
```

seed

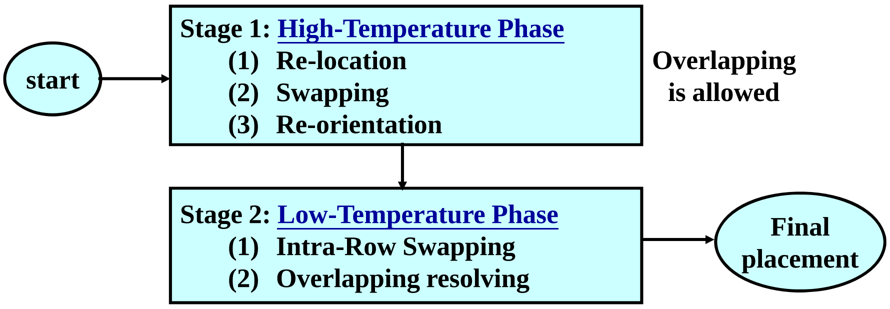
Target location

Vacant
Occupied,
Locked,

Stopping criterion of an iteration

TimberWolf: Placement by Simulated Annealing

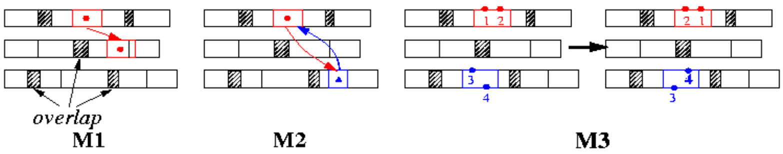
- Sechen and Sangiovanni-Vincentelli,
  - “The TimberWolf placement and routing package,” *IEEE J. Solid-State Circuits*, Feb. 1985;
  - “TimberWolf 3.2: A new standard cell placement and global routing package,” DAC-86.



Questions: What are the moving scheme and cost function?

TimberWolf: Moving Type

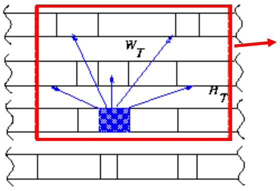
- Solution Space:
  - All possible arrangements of the modules into rows, possibly with overlaps.
- Moving Types
  - *M_1*: Displace a module to a new location.
  - *M_2*: Interchange two modules.
  - *M_3*: Change the orientation of a module.



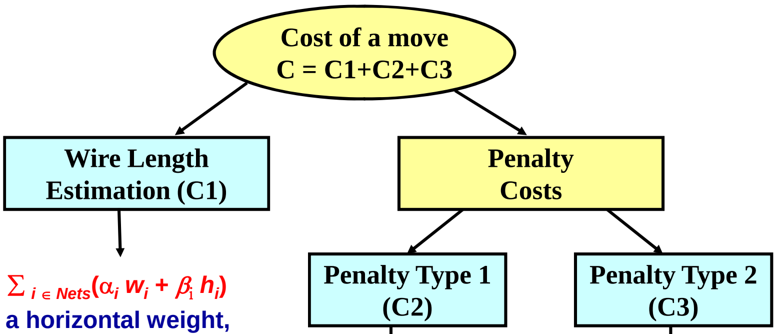
re-location **swapping** **re-orientation**

TimberWolf: Moving Scheme

- Neighborhood Window: Range Limiter
  - The neighborhood window shrinks as temperature decreases.
  - At the beginning, (W_T, H_T) is big enough to contain the whole chip.
  - Window height & width is proportional to *log*(*T*).
- Moving Scheme
  - (1) Pick an M1 or M2 type of move. The probabilities of M1 is 0.8, while that of M2 is 0.2.
  - (2) Check acceptance or rejection by cost function and temperature
  - (3) If M1 is picked while rejected ➔ Try M3 with probability of 0.1.



TimberWolf: Cost Function



Cost of a move
C = C1+C2+C3

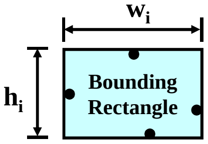
Wire Length Estimation (C1)

Penalty Costs

Penalty Type 1 (C2)

Penalty Type 2 (C3)

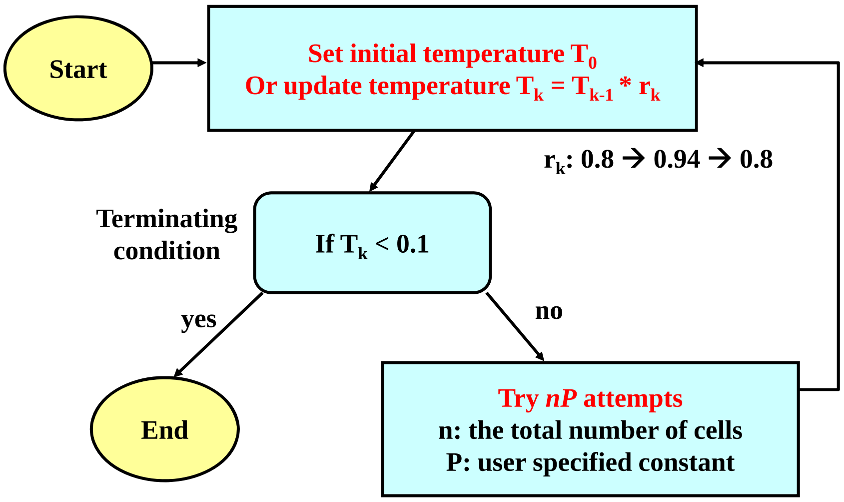
$C_1 = \Sigma_{i \in Nets}(\alpha_i w_i + \beta_i h_i)$
α_i is a horizontal weight,
β_i is a vertical weight,



$C_2 = \gamma \Sigma_{i \neq j} O^2_{ij}$
γ: penalty factor
O_{ij}: amount of overlaps in the *x*-dimension between cell *i* and *j*.

$C_3 = \delta \Sigma_{r \in Rows}|L_r - D_r|$,
δ : penalty factor
D_r: Desired row length
L_r: sum of cells' widths in row *r*

TimberWolf: Annealing Schedule



Start

Set initial temperature T_0
Or update temperature $T_k = T_{k-1} * r_k$

r_k: 0.8 → 0.94 → 0.8

Terminating condition

If $T_k < 0.1$

yes

no

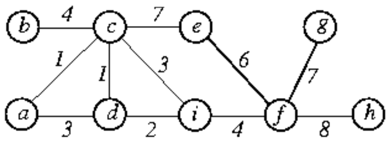
End

Try *nP* attempts
n: the total number of cells
P: user specified constant

Placement by the Genetic Algorithm

- Cohoon & Paris, "Genetic placement," ICCAD-86.
- Genetic Ingredients:
  - (1) Encoding (or Chromosome) of feasible solutions
  - (2) No. of populations in each generation: e.g., 50
  - (3) Fitness Function: for offspring selection
  - (4) Operators: Crossover, Mutation, Inversion

A connectivity graph
For a netlist



Space to be
filled in

6	7	8
3	4	5
0	1	2

Encoding

d	e	f
c	b	i
a	g	h

string: aghcbidef

Genetic Operator: Crossover

- Main genetic operator:
  - Operate on two individuals and generates an offspring.
  - $[bidef|aghc](\frac{1}{86}) + [bdefi|gcha](\frac{1}{110}) \rightarrow [bidefgcha](\frac{1}{63})$.
  - Need to avoid repeated symbols in the solution string!
- Partially mapped crossover
  - for avoiding repeated symbols:
  - $[bidef|gcha](\frac{1}{86}) + [aghcb|\underline{idef}](\frac{1}{85}) \rightarrow [bgcha|idef]$.
  - Copy *idef* to the offspring; scan [*bidef*|*gcha*] from the left, and then copy all unrepeated genes.

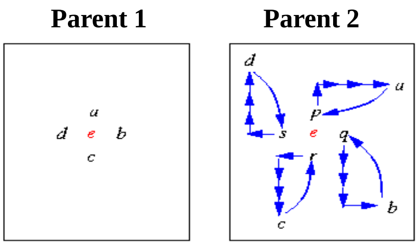
Note: Cost of each placement's encoding

$$\text{Cost} = \frac{1}{\text{(Weighted Sum of Wire Lengths)}}$$

Two More Crossover Operations

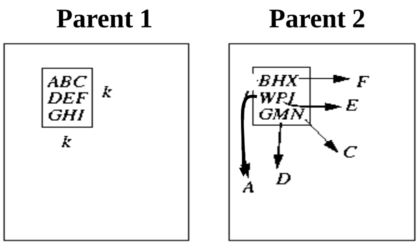
- Cut-and-paste + Chain moves
  - The cells that earlier occupied the neighboring locations in parent 2 are shifted outwards.
- Cut-and-paste + Swapping
  - Copy $k \times k$ square modules
  - Swap cells not in both square modules.

Common squares: {BHIG}


Parent 1

Parent 2

Cut-and-paste + chain moves


Parent 1

Parent 2

Cut-and-paste + Swapping

Genetic Operators: Mutation & Inversion

- **Mutation:**
  - Prevents loss of diversity by introducing new solutions.
  - A commonly used mutation: pairwise interchange.
- **Inversion: [*bid*|*efgch*|*a*] → [*bid*|*hcgfe*|*a*].**
- **Probabilities of mutation and inversion:**
  - probability P_μ and P_i respectively.

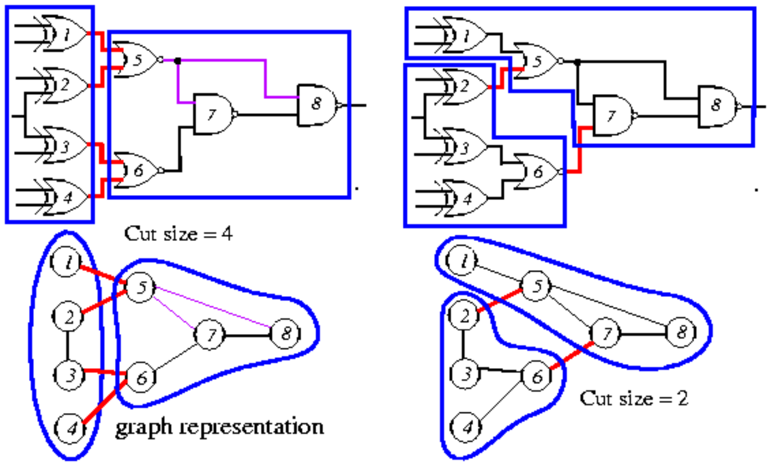
Pseudo-Code of Genetic Algorithm

```
Algorithm: Genetic_Placement($N_p$, $N_g$, $N_o$, $P_i$, $P\mu$ )
/* $N_p$: population size; */                /* $N_g$: number of generation; */
/* $N_o$: number of offspring; */
/* $Pi$ : inversion probability; */          /* $P\mu$ : mutation probability; */
begin
ConstructPopulation($N_p$);  /* randomly generate the initial population */
for $j \leftarrow 1$ to $N_p$
    Evaluate Fitness(population($N_p$));
for $i \leftarrow 1$ to $N_g$ /* produce one generation at a time */
   for $j \leftarrow 1$ to $N_o$
      ($x$, $y$) ← ChooseParents; /* choose parents with probability $\propto$ fitness value */
      offspring($j$) ← GenerateOffspring($x$, $y$); /* crossover to generate offspring */
      for $h \leftarrow 1$ to $N_p$
         With probability $P\mu$, apply Mutation(population($h$));
      for $h \leftarrow 1$ to $N_p$
         With probability $P_i$, apply Inversion(population($h$));
      Evaluate Fitness(offspring($j$));
   population ← Select(population, offspring, $N_p$);
return the highest scoring configuration in population;
```

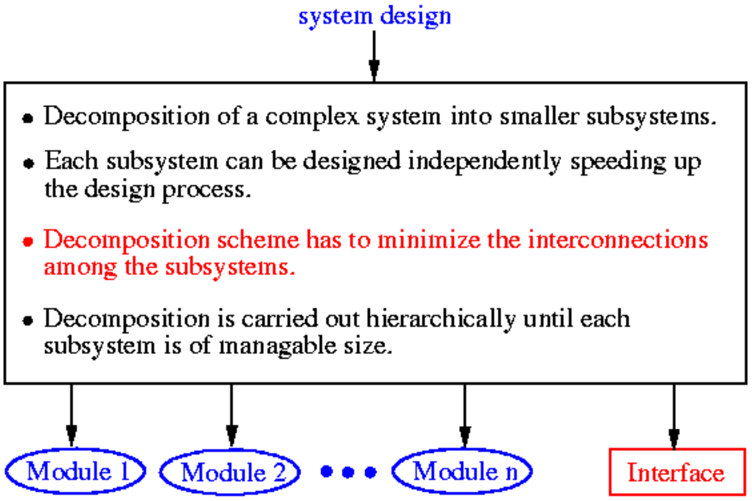
Outline

- **Placement metrics**
- **Placement**
  - **Clustering-Based**
  - **Partitioning-Based**
  - **Force-Directed**
  - **Simulated-Annealing**
  - **Genetic Algorithm**
- **Partitioning**
  - **Kernighang-Lin Partitioning Algorithm**
  - **Simulated-Annealing Based Partitioning**

Example: Circuit Partitioning



Partitioning

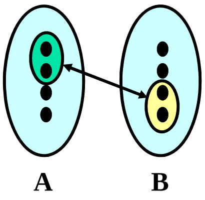


system design

- Decomposition of a complex system into smaller subsystems.
- Each subsystem can be designed independently speeding up the design process.
- Decomposition scheme has to minimize the interconnections among the subsystems.
- Decomposition is carried out hierarchically until each subsystem is of managable size.

Kernighan-Lin Algorithm

- **Kernighan and Lin,**
  - **"An efficient heuristic procedure for partitioning graphs,"** ***The Bell System Technical Journal*****, vol. 49, no. 2, Feb. 1970.**
- **Basic Strategy**
  - **An iterative, 2-way, balanced partitioning heuristic**
- **Basic Procedure**
  - **(1) Start with an initial solution S = (A | B)**
  - **(2) Find a subset from A and B for swapping**
  - **(3) Iterate until there is no gain**

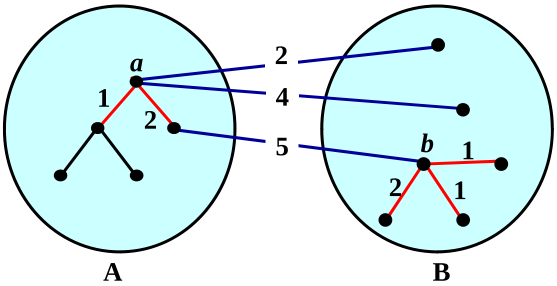


Questions:
(1) What is the cost function?
(2) How to find best swapping pairs?

Internal Cost vs. External Cost

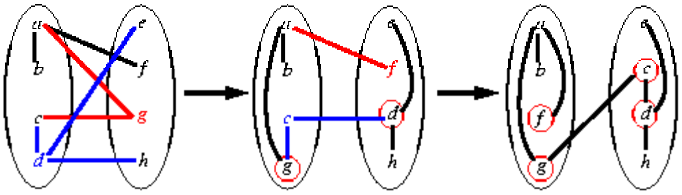
For vertex *a*:	**For vertex *b*:**
Internal cost: (1+2) = 3	**Internal cost: (1+1+2) = 4**
External cost: (2+4) = 6	**External cost: 5**

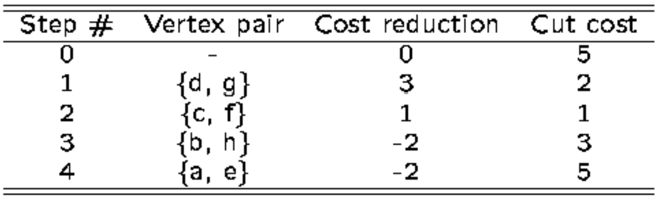
What if we swap *a* and *b*?
➔ Internal cost and external cost swaps as well.
➔ External Cost Change = (3+4) - (6+5) = -4 (reduction).
➔ Looks like a good swap.



K-L Algorithm: A Simple Example

- **Each edge has a unit weight.**



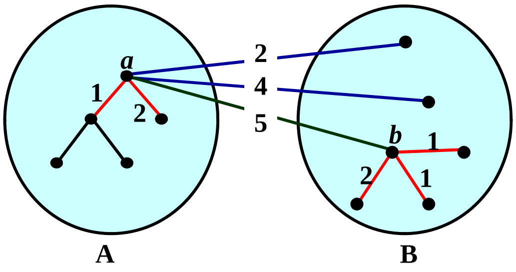


Step #	Vertex pair	Cost reduction	Cut cost
0	-	0	5
1	{d, g}	3	2
2	{c, f}	1	1
3	{b, h}	-2	3
4	{a, e}	-2	5

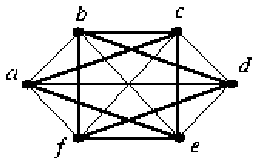
Terminology

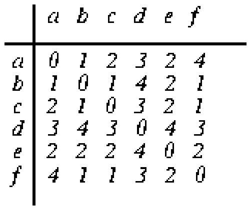
- Two sets *A* and *B* such that $|A| = n = |B|$ and $A \cap B = \emptyset$.
- External cost of $a \in A$: $E_a = \sum_{v \in B} c_{av}$.
- Internal cost of $a \in A$: $I_a = \sum_{v \in A} c_{av}$.
- *D*-value of a vertex *a*: $D_a = E_a - I_a$ (cost reduction for moving *a*).
- Cost reduction (gain) for swapping *a* and *b*:
  - $g_{ab} = D_a + D_b - 2c_{ab}$

Watch out for the edge connecting the swapping pair (*a*, *b*) !

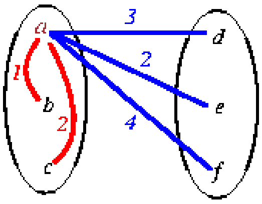


K-L Algorithm: A Weighted Example





	a	b	c	d	e	f
a	0	1	2	3	2	4
b	1	0	1	4	2	1
c	2	1	0	3	2	1
d	3	4	3	0	4	3
e	2	2	2	4	0	2
f	4	1	1	3	2	0



costs associated with *a*

Initial cut cost = (3+2+4)+(4+2+1)+(3+2+1) = 22

- **Iteration 1:**

$I_a = 1 + 2 = 3$; $E_a = 3 + 2 + 4 = 9$; $D_a = E_a - I_a = 9 - 3 = 6$

$I_b = 1 + 1 = 2$; $E_b = 4 + 2 + 1 = 7$; $D_b = E_b - I_b = 7 - 2 = 5$

$I_c = 2 + 1 = 3$; $E_c = 3 + 2 + 1 = 6$; $D_c = E_c - I_c = 6 - 3 = 3$

$I_d = 4 + 3 = 7$; $E_d = 3 + 4 + 3 = 10$; $D_d = E_d - I_d = 10 - 7 = 3$

$I_e = 4 + 2 = 6$; $E_e = 2 + 2 + 2 = 6$; $D_e = E_e - I_e = 6 - 6 = 0$

$I_f = 3 + 2 = 5$; $E_f = 4 + 1 + 1 = 6$; $D_f = E_f - I_f = 6 - 5 = 1$

(Step 1): Computing the g Value

- **Iteration 1:**

$$\begin{array}{lll} I_a = 1 + 2 = 3; & E_a = 3 + 2 + 4 = 9; & D_a = E_a - I_a = 9 - 3 = 6 \\ I_b = 1 + 1 = 2; & E_b = 4 + 2 + 1 = 7; & D_b = E_b - I_b = 7 - 2 = 5 \\ I_c = 2 + 1 = 3; & E_c = 3 + 2 + 1 = 6; & D_c = E_c - I_c = 6 - 3 = 3 \\ I_d = 4 + 3 = 7; & E_d = 3 + 4 + 3 = 10; & D_d = E_d - I_d = 10 - 7 = 3 \\ I_e = 4 + 2 = 6; & E_e = 2 + 2 + 2 = 6; & D_e = E_e - I_e = 6 - 6 = 0 \\ I_f = 3 + 2 = 5; & E_f = 4 + 1 + 1 = 6; & D_f = E_f - I_f = 6 - 5 = 1 \end{array}$$

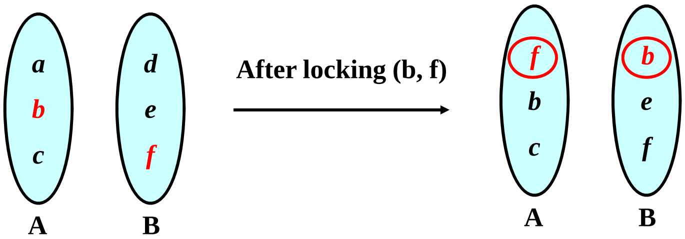
- **$g_{xy} = D_x + D_y - 2c_{xy}$.**

$$\begin{array}{lll} g_{ad} & = & D_a + D_d - 2c_{ad} = 6 + 3 - 2 \times 3 = 3 \\ g_{ae} & = & 6 + 0 - 2 \times 2 = 2 \\ g_{af} & = & 6 + 1 - 2 \times 4 = -1 \\ g_{bd} & = & 5 + 3 - 2 \times 4 = 0 \\ g_{be} & = & 5 + 0 - 2 \times 2 = 1 \\ \underline{g_{bf}} & \underline{=} & \underline{5 + 1 - 2 \times 1 = 4 \ (maximum)} \\ g_{cd} & = & 3 + 3 - 2 \times 3 = 0 \\ g_{ce} & = & 3 + 0 - 2 \times 2 = -1 \\ g_{cf} & = & 3 + 1 - 2 \times 1 = 2 \end{array}$$

Best pick

- **Swap *b* and *f*.** $(\hat{g}_1 = 4)$

(Step 2): Lock and Update



Update the D-value of each unlocked vertices:
(1) For unlocked vertex, x, in A: $D_x' = D_x + 2c_{xb} - 2c_{xf}$
(2) For unlocked vertex, y, in B: $D_y' = D_y + 2c_{yf} - 2c_{yb}$

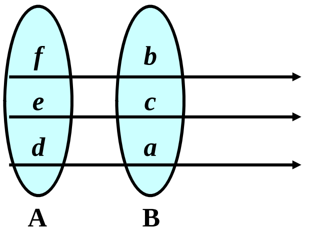
➔ Update the g-value of each unlocked vertex pairs

$$g_{xy} = D_x' + D_y' - 2c_{xy}$$

➔ Find the next candidate pair to lock …

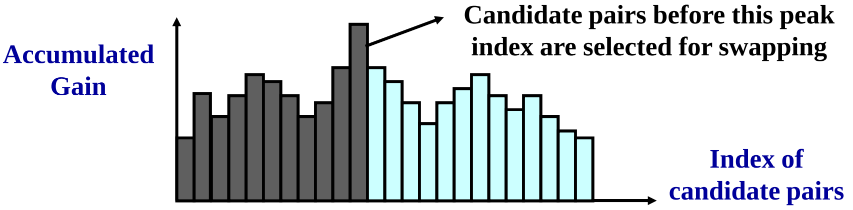
(Step 3): Determining Swapping Pairs

At the end of the locking process: each vertex is paired up with another one
Locked pairs: (b, f) ➔ (c, e) ➔ (a, d)



Gain of each pair	Accumulated gain
4	4
2	6 best
-2	4

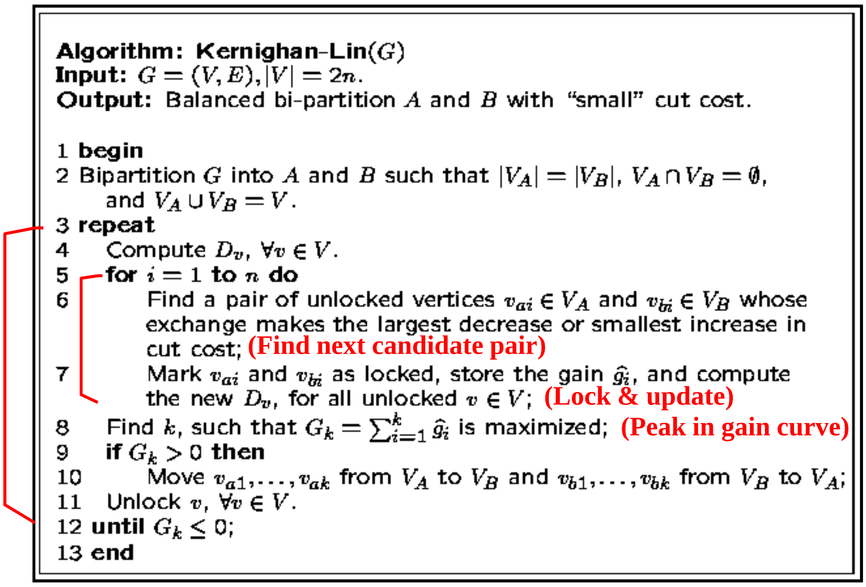
➔ Best swapping pairs {(b, f), (c, e)}



Candidate pairs before this peak index are selected for swapping

---

Pseudo-Code of Kernighan-Lin Algorithm



```
Algorithm: Kernighan-Lin($G$)
Input: $G = (V, E), |V| = 2n$.
Output: Balanced bi-partition $A$ and $B$ with "small" cut cost.

begin
    Bipartition $G$ into $A$ and $B$ such that $|V_A| = |V_B|$, $V_A \cap V_B = \emptyset$,
        and $V_A \cup V_B = V$.
repeat
    Compute $D_v$, $\forall v \in V$.
    for $i = 1$ to $n$ do
        Find a pair of unlocked vertices $v_{ai} \in V_A$ and $v_{bi} \in V_B$ whose
        exchange makes the largest decrease or smallest increase in
        cut cost; (Find next candidate pair)
        Mark $v_{ai}$ and $v_{bi}$ as locked, store the gain $\hat{g}_i$, and compute
        the new $D_v$, for all unlocked $v \in V$; (Lock & update)
    Find $k$, such that $G_k = \sum_{i=1}^{k} \hat{g}_i$ is maximized; (Peak in gain curve)
    if $G_k > 0$ then
        Move $v_{a1}, \ldots, v_{ak}$ from $V_A$ to $V_B$ and $v_{b1}, \ldots, v_{bk}$ from $V_B$ to $V_A$;
    Unlock $v$, $\forall v \in V$.
until $G_k \leq 0$;
end
```

Time Complexity

- Line 4: Initial computation of D: $O(n^2)$
- Line 5: The for-loop: $O(n)$
- The body of the loop: $O(n^2)$.
  - Lines 6--7: Step i takes $(n - i + 1)^2$ time.
- Lines 4--11: Each pass of the repeat loop: $O(n^3)$.
- Suppose the repeat loop terminates after r passes.
- The total running time: $O(rn^3)$.
  - Polynomial-time algorithm?

Extensions of K-L Algorithm

- Unequally sized subsets (assume $n_1 < n_2$)
  1. Partition: $|A| = n_1$ and $|B| = n_2$.
  2. Add $n_2 - n_1$ dummy vertices to set A. Dummy vertices have no connections to the original graph.
  3. Apply the Kernighan-Lin algorithm.
  4. Remove all dummy vertices.
- Unequally sized “vertices”
  1. Assume that the smallest “vertex" has unit size.
  2. Replace each vertex of size s with s vertices which are fully connected with edges of infinite weight.
  3. Apply the Kernighan-Lin algorithm.
- K-way partition
  1. Partition the graph into k equally sized sets.
  2. Apply the Kernighan-Lin algorithm for each pair of subsets.
  3. Time complexity? Can be reduced by recursive bi-partition.

Concluding Remarks

- **Placement**
  - (1) Top-Down, Bottom-Up, or Hybrid
  - (2) A Good Heuristic: Force-Directed Algorithm
  - (3) Application of Simulated Annealing, Genetic Algorithm
- **Partitioning**
  - An often encountered problem in EDA
  - Kernighan-Lin algorithm is a classic algorithm

For Any Search Problem,
Evolution Works, But Just Takes Time …

清華大學 EE 5265
積體電路設計自動化

單元 8
Routing

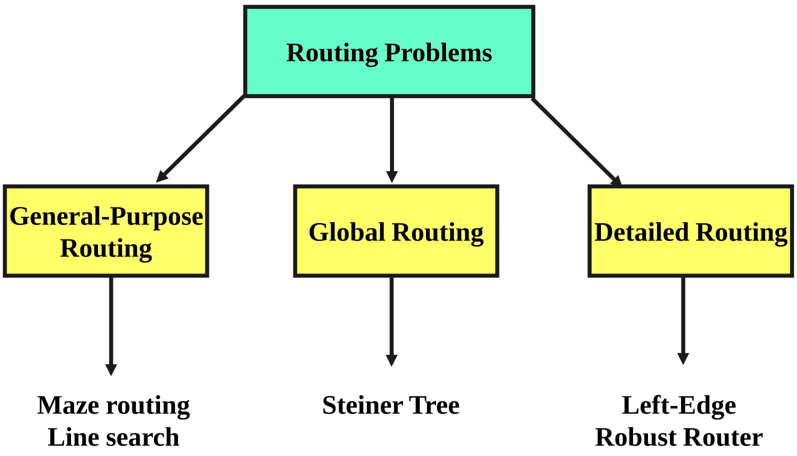
教育部顧問室
「超大型積體電路與系統設計」教育改進計畫
EDA聯盟 － 推廣課程

致謝

本單元之教材主要取自於
教育部
超大型積體電路與系統設計教育改進計畫
EDA聯盟之課程發展成果

- **(教材編纂小組成員)**
- 台灣大學電機系 張耀文
- 清華大學電機系 黃錫瑜
- 交通大學資科系 李毅郎
- 中央大學電機系 劉建男
- 元智大學資工系 林榮彬

Classification of Routing Problems

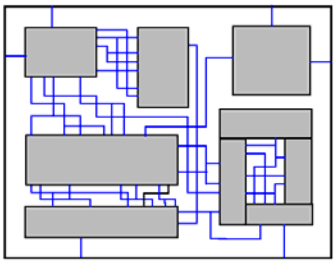


Routing Problems

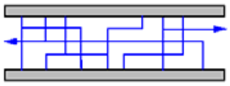
- General-Purpose Routing
  - Maze routing
  - Line search
- Global Routing
  - Steiner Tree
- Detailed Routing
  - Left-Edge
  - Robust Router

Outline

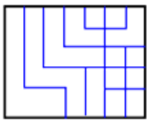
- Course contents
  - Basics of Routing
  - General-Purpose Routing (Maze Routing, Line Search Routing)
  - Global Routing
  - Detailed Routing
- Readings
  - Chapter 9



Detailed routing

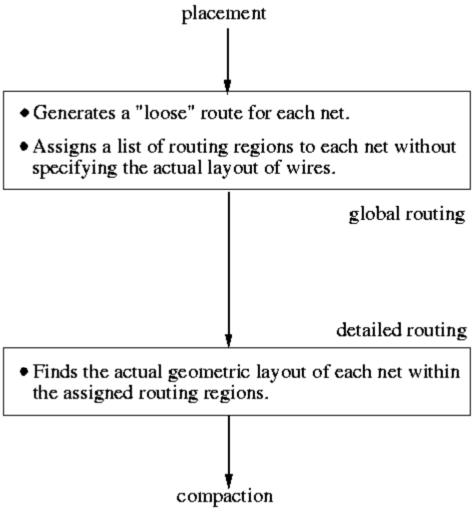


channel routing



switchbox routing

Routing



placement

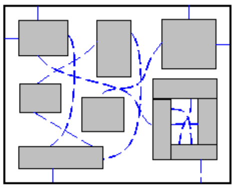
- Generates a "loose" route for each net.
- Assigns a list of routing regions to each net without specifying the actual layout of wires.

global routing

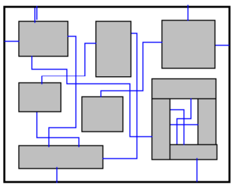
detailed routing

- Finds the actual geometric layout of each net within the assigned routing regions.

compaction



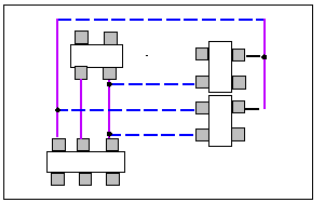
Global routing



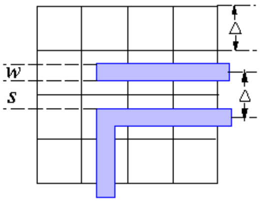
Detailed routing

Routing Constraints

- Requirements of a valid routing
  - 100% routing completion
  - 100% layout rules compliance
  - Use of assigned layers only
- Quality considerations of Routing
  - (1) Area minimization
  - (2) Performance-driven routing (critical wire length minimization)
  - (3) Crosstalk alleviation
  - (4) Resistance to process variations (Design for manufacturability)



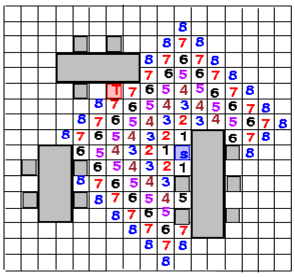
Two−layer routing



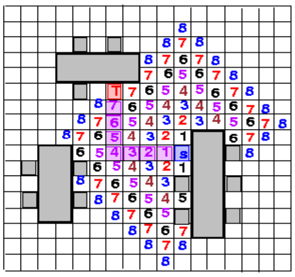
Geometrical constraint

Lee Algorithm

- **Basic Concept:**
  - Find a path from *S* to *T* by “wave propagation”.
- **Strength:**
  - Guarantee to find the best route
- **Time and space complexities**
  - O(MxN) for MxN grid ➔ Huge !



Filling

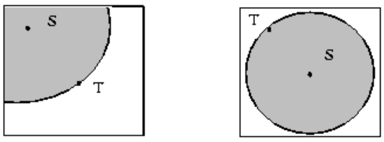


Retrace

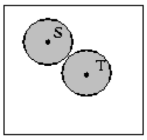
Improvements of Maze Routing

- **Starting Point Selection:**
  - Choose the point farthest from the center of the grid as the starting point.
- **Double Fan-Out:**
  - Propagate waves from both the source and the target cell.
- **Framing:**
  - Search inside a rectangle area 10-20% larger than the bounding box containing the source and target.
  - Need to enlarge the rectangle and redo the search if it fails.

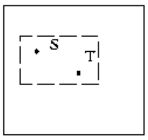
starting point selection



double fan–out

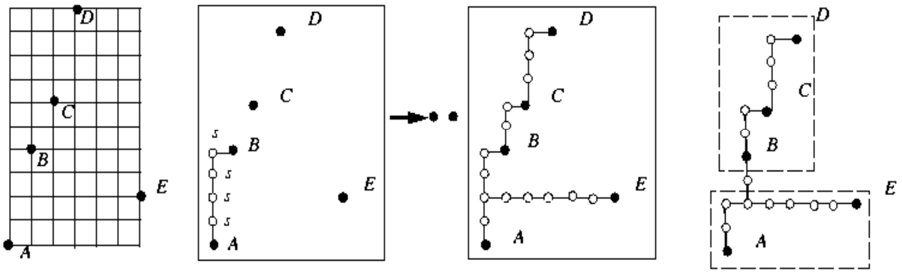


framing



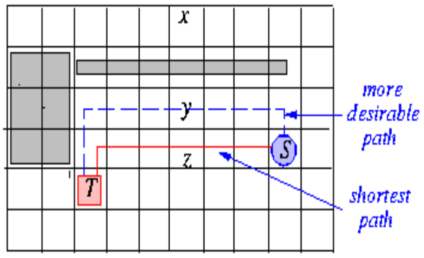
Connecting Multi-Terminal Nets

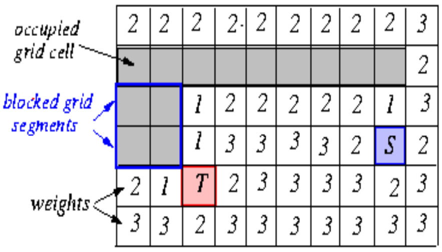
- Step 1: Propagate wave from the source *s* to the closest target.
- Step 2: Mark ALL cells on the path as *s*.
- Step 3: Propagate wave from ALL *s* cells to the other cells.
- Step 4: Continue until all cells are reached.
- Step 5: Apply heuristics to further reduce the tree cost.



Routing on a Weighted Grid

- **Motivation:**
  - To find a more desirable path (I.e., path of less weight)
  - To achieve a more balanced routing
- **Weight of a grid cell**
  - Defined as (the number of unblocked neighbor cells – 1)

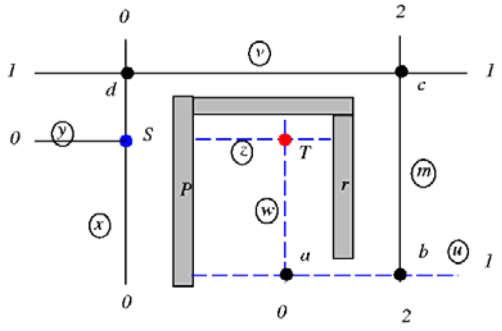




2	2	2	2	2	2	2	2	2	3
									2
		1	2	2	2	2	2	1	3
		1	3	3	3	3	2	S	2
2	1	T	2	3	3	3	3	2	3
3	3	2	3	3	3	3	3	3	3

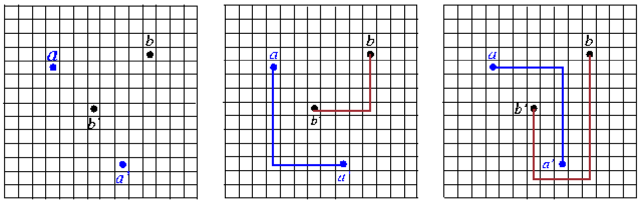
Hightower – Line-Search Algorithm

- **Hightower**
  - "A solution to line-routing problem on the continuous plane," DAC-69.
- **Basic Concept:**
  - A route is searched by moving two crossing lines.
  - One stride is determined at each step.
  - Alternate the vertical and horizontal moves.
  - Get around the obstacles and get closer to the target destination.

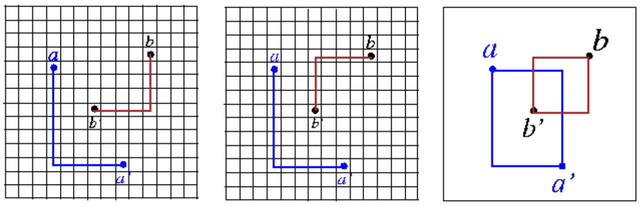


Net Ordering

- **Net ordering greatly affects routing solutions.**



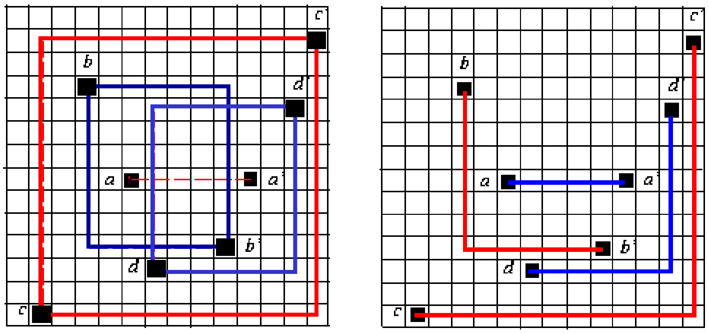
route net a before net b



route net b before net a

Net Ordering Heuristics

- Ordering Criteria
  - A net with more pins within their bounding boxes last
  - A net with large length estimation first (or last?)
  - A net with higher timing criticality first



routing ordering: a (0) –> b (1) –> d (2) –> c (6)

Rip-Up and Re-Routing

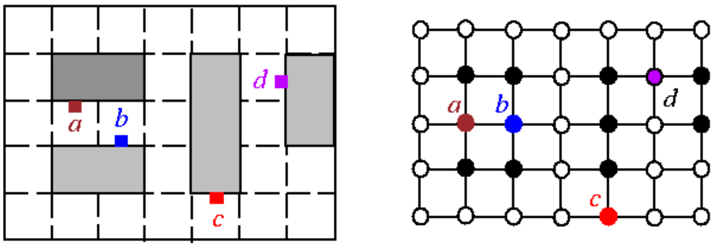
- Rip-up and re-routing
  - Is required when a router fails to connect all nets.
- Two steps in rip-up and re-routing
  - (1) Identify bottleneck regions
  - (2) Rip up some already routed nets
  - (3) Route the blocked connections
  - (4) Re-Route the ripped up connections
- Stopping criteria
  - (1) All nets are routed successfully
  - (2) Time limit is exceeded

Outline

- Basics of Routing
- General-Purpose Routing
  - Maze Routing
  - Line Search Routing
- ➡ Global Routing
  - Minimum Steiner Tree Problem
- Channel Routing

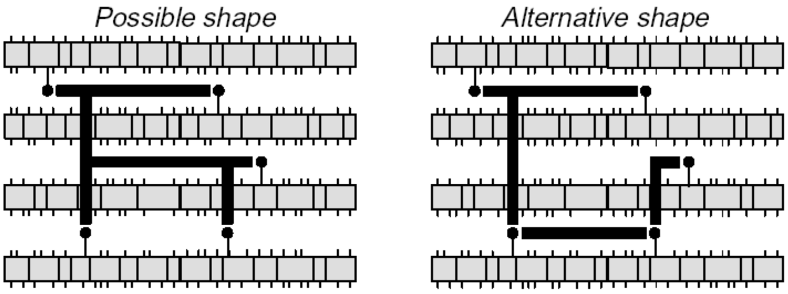
Graph Models for Global Routing: Grid Graph

- Vertex
  - Each cell is represented by a vertex.
- Edge
  - Two vertices are joined by an edge if the corresponding cells are adjacent to each other
- Occupation mark
  - The occupied cells are marked as filled circles, whereas the others are clear circles.



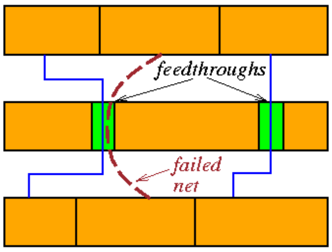
Global Routing

- **Global Routing**
  - is the process of roughly fixing the shapes of the connections for each net.
  - by distributing the wiring segments among channels.
  - Each shape is a rectilinear Steiner tree.


Possible shape *Alternative shape*

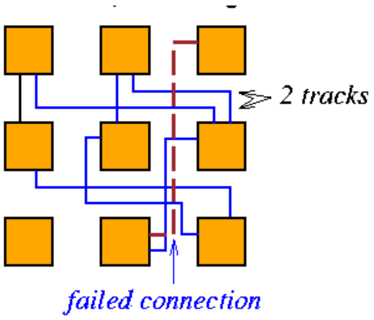
Example Global Routing

Global routing
For Standard-Cell Design



Routing could fail due to Inadequate feed-through channels

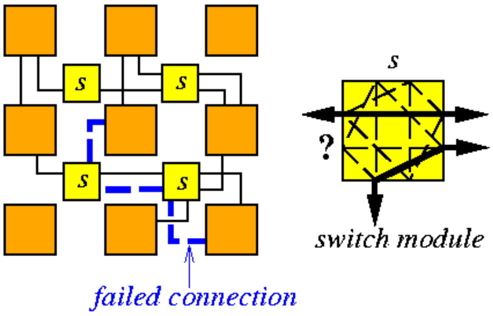
Global routing
For Gate-Array



Each channel has a capacity of 2 tracks.

Global Routing in FPGA

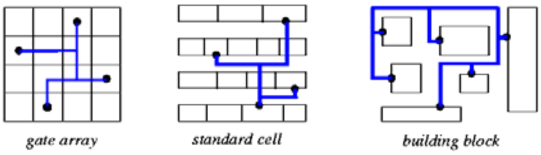
- **Routing constraints**
  - **Depends on the switch box architecture.**
- **For performance-driven routing**
  - **(1) Minimize the number of switches.**
  - **(2) Minimize the maximum of the critical wire length.**



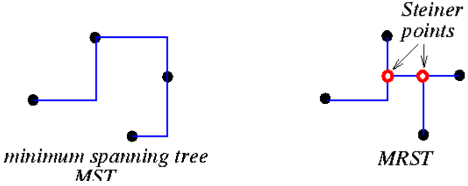
Each channel has a capacity of 2 tracks.

The Steiner Tree Problem

- **Problem:**
  - **Given a set of pins of a net, connect the pins by a routing tree.**

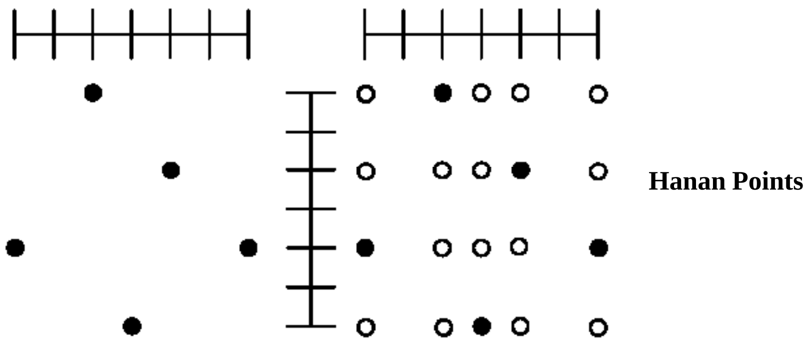


- **Minimum Rectilinear Steiner Tree (MRST) Problem:**
  - Given *n* points in the plane, find a minimum-length tree of rectilinear edges which connects the points.
  - MRST(P) = MST(P ∪ S), where *P* and *S* are the sets of original points and Steiner points, respectively.



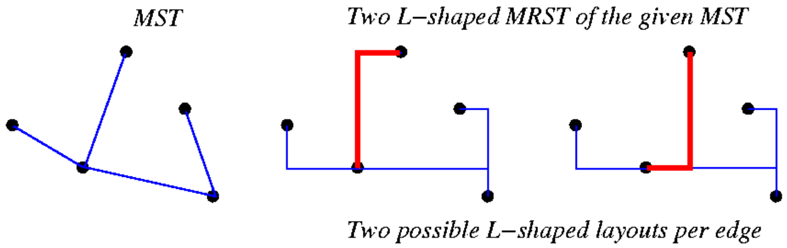
Theoretical Results for the MRST Problem

- **Hanan's Theorem:**
  - There exists an MRST with all Steiner points (set *S*) chosen from the points of horizontal and vertical lines crossing points in *P*.
- **Hwang's Theorem:** For any point set *P*, $\frac{Cost(MST(P))}{Cost(MRST(P))} \leq \frac{3}{2}$.



Coping with the MRST Problem

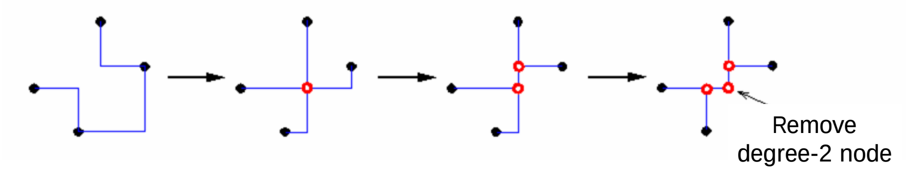
- Ho, Vijayan, Wong,
  - "New algorithms for the rectilinear Steiner problem,"
  - (1) Construct an MRST from an MST.
  - (2) Each edge is straight or L-shaped.
  - (3) Maximize overlaps by dynamic programming.
- About 8% smaller than *Cost*(*MST*).



Iterated 1-Steiner Heuristic for MRST

- **Kahng & Robins (1990)**
  - **"A new class of Steiner tree heuristics with good performance: the iterated 1-Steiner approach,"**

```
Algorithm: Iterated_1-Steiner(P)
P: set of n points to be connected
begin
$S \leftarrow \emptyset$;
    /* $H(P \cup S)$: set of Hanan points */
    /* $\Delta MST(A, B) = Cost(MST(A)) - Cost(MST(A \cup B))$ */
while ($Cand \leftarrow \{x \in H(P \cup S) | \Delta MST(P \cup S, \{x\}) > 0\} \neq \emptyset$) do
    Find $x \in C$ and which maximizes $\Delta MST(P \cup S), \{x\})$;
    $S \leftarrow S \cup \{x\}$;
    Remove points in S which have degree $\leq 2$ in $MST(P \cup S)$;
Output $MST(P \cup S)$;
end
```



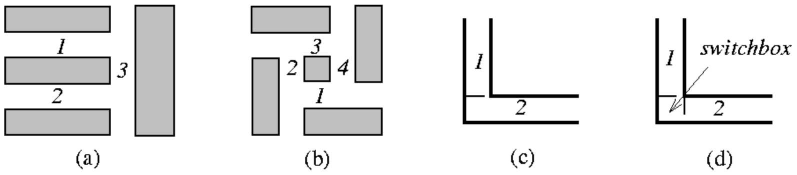
Remove degree-2 node

Outline

- **Basics of Routing**
- **General-Purpose Routing**
- **Global Routing**
- **Channel Routing**
  - **Left-Edge Algorithm**
  - **Robust Channel Router**

Routing Area Partitioning

- Routing Area
  - Is usually partitioned into smaller pieces before routing
- Types of routing area
  - (1) Normal channel
  - (2) L-shaped channel
  - (3) Switchbox
- An L-shaped channel
  - Can be divided into a normal channel + a switchbox

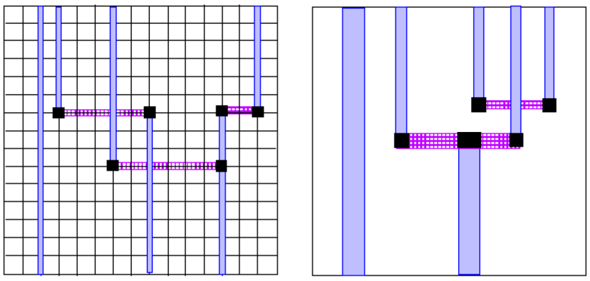


A Few Parameters When Doing Routing

- Number of terminals
  - Two-terminal nets vs. multi-terminal nets
- Net types
  - Power / ground / clock wires vs. signal wires
- Number of layers
  - Two vs. three, or more layers
- Signal types
  - Critical nets vs. non-critical nets

Routing Models

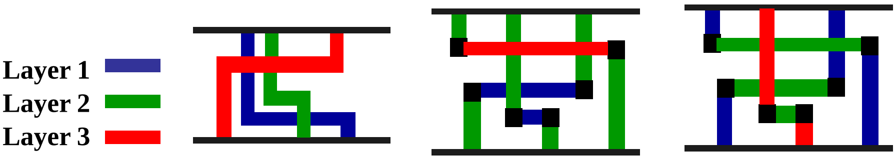
- **Grid-based model:**
  - A grid is superimposed on the routing region.
  - Wires follow paths along the grid lines.
  - Pitch: distance between two grid lines.
- **Gridless model: one without grid lines.**



grid–based gridless

Models for Multi-Layer Routing

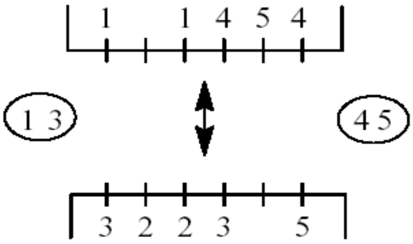
- **Unreserved layer model:**
  - Any net segment is allowed to be placed in any layer.
- **Reserved layer model:**
  - Certain type of segments are restricted to particular layer(s).
  - Two-layer:
    - (HV): Layer 1→ Horizontal, Layer 2→ Vertical
    - (VH): Layer 1→ Vertical, Layer 2→ Horizontal
  - Three-layer: HVH, VHV



Unreserved model HVH model VHV model

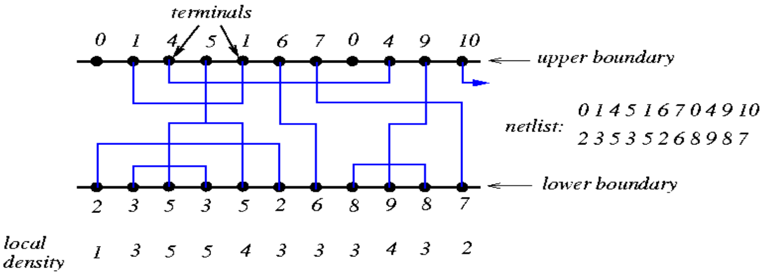
Channel Routing Problem

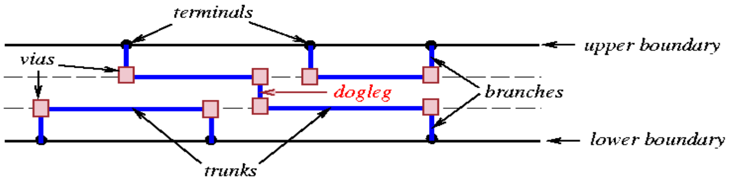
- **Inputs for channel routing**
  - (1) A rectangle routing area
  - (2) Fixed terminals at the top and bottom boundaries
  - (3) Floating terminals at left and right boundaries
- **Objective**
  - To minimize the channel height



Terminology for Channel Routing

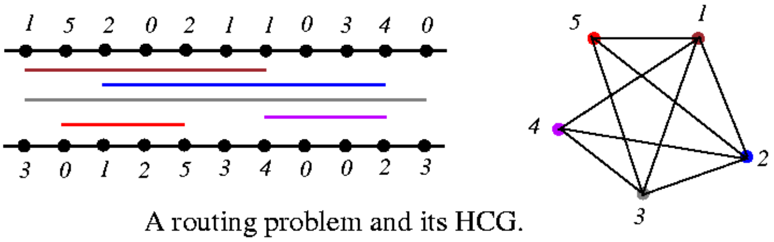
- **Channel density: maximum local density**
  - Number of horizontal tracks required ≥ channel density.





Horizontal Constraint Graph (HCG)

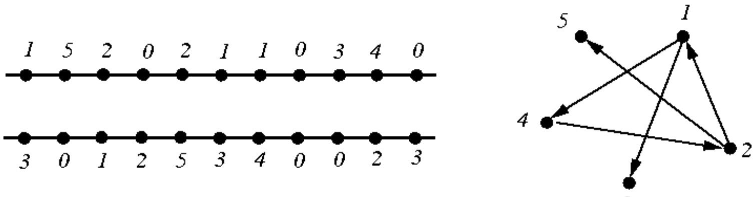
- HCG $G = (V, E)$ is an undirected graph, where
  - $V = \{ v_i \mid v_i$ represents a net $n_i\}$
  - $E = \{(v_i, v_j) \mid$ a horizontal constraint exists between n_i and $n_j\}$.
- For graph G:
  - vertices ⇔ nets; edge (i, j) ⇔ net i overlaps net j.



A routing problem and its HCG.

Vertical Constraint Graph (VCG)

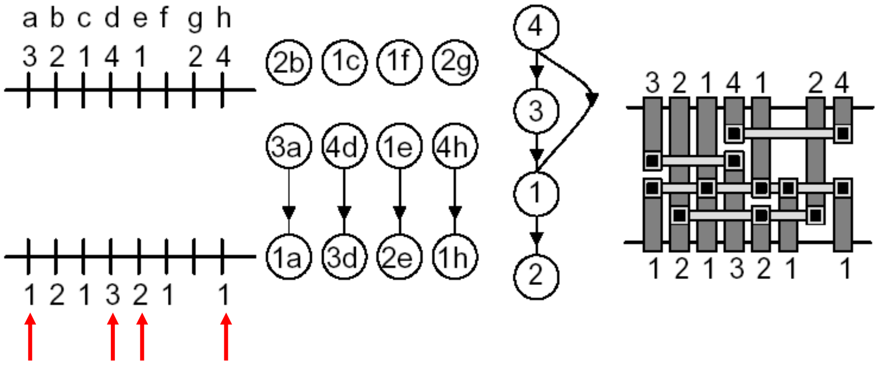
- VCG $G = (V, E)$ is a directed graph where
  - $V = \{ v_i \mid v_i$ represents a net $n_i\}$
  - $E = \{(v_i, v_j) \mid$ a vertical constraint exists between n_i and $n_j\}$.
- For graph G:
  - Vertices ⇔ nets; edge $i \rightarrow j$ ⇔ net i must be above net j.



A routing problem and its VCG.

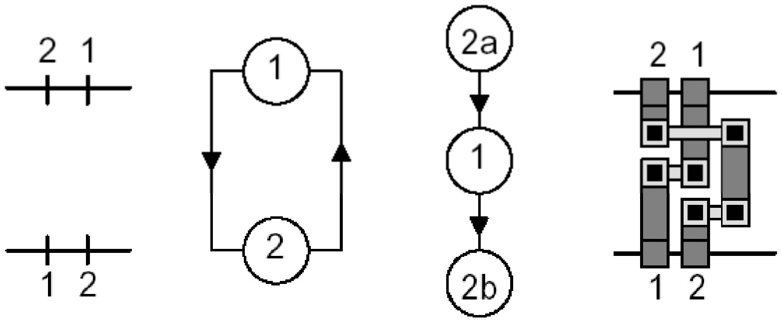
Example: Vertical Constraint Graph

- **Nets to be routed:**
  - Nets 1, 2, 3, 4
- **Columns of the channel**
  - Columns a, b, c, d, e, f, g, h



Example: Cyclic Vertical Constraints

- **Cyclic vertical constraints**
  - Needs to be resolved by splitting horizontal segments
  - That is, doglegs are necessary



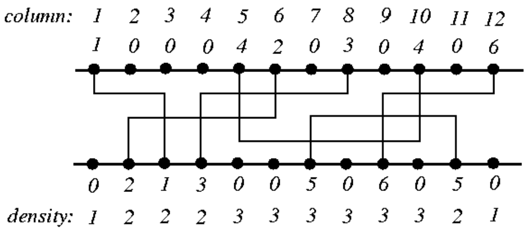
2L Channel Routing: Basic Left-Edge Algorithm

- Hashimoto & Stevens
  - "Wire Routing by Optimizing Channel Assignment within Large Apertures," DAC-71
  - For problems without vertical constraint
  - HV-layer model is used
  - No doglegs are allowed
- Major operations
  - (1) Treat each net as an interval
  - (2) Intervals are sorted based on theirs left-end x-coordinates
  - (3) Intervals are routed one at a time based on the above order
  - (4) For a net, tracks are scanned from top to bottom. First available track is assigned immediately for this net
- Results
  - Simple left-edge algorithm produces minimal number of tracks under the assumption that there are no vertical constraints

---

Basic Left-Edge Example

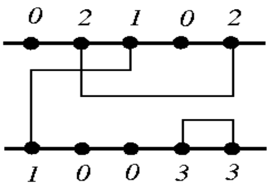
- List of nets to be routed, $U = \{I_1, I_2, \ldots, I_6\}$;
  - $I_1 = [1, 3]$, $I_2 = [2, 6]$, $I_3 = [4, 8]$, $I_4 = [5, 10]$, $I_5 = [7, 11]$, $I_6 = [9, 12]$.
- *t* =1:
  - Route I_1: *watermark* = 3;
  - Route I_3 : *watermark* = 8;
  - Route I_6: *watermark* = 12;
- *t* = 2:
  - Route I_2 : *watermark* = 6;
  - Route I_5 : *watermark* = 11;
- *t* = 3: Route I_4



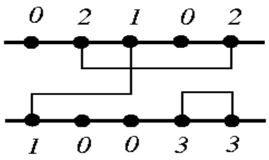
Basic Left-Edge Algorithm

```
Algorithm: Basic_Left-Edge($U$, $track[j]$)
    $U$: set of unassigned intervals (nets) $I_1, \ldots, I_n$;
    $I_j=[s_j, e_j]$: interval $j$ with left-end $x$-coordinate $s_j$ and right-end $e_j$;
    $track[j]$: track to which net $j$ is assigned.

begin
$U \leftarrow \{I_1, I_2, \ldots, I_n\}$;
$t \leftarrow 0$;
while ($U \neq \emptyset$) do
    $t \leftarrow t + 1$;
    $watermark \leftarrow 0$;
    while (there is an $I_j \in U$ s.t. $s_j > watermark$) do
        Pick the interval $I_j \in U$ with $s_j > watermark$,
        $track[j] \leftarrow t$;
      $watermark \leftarrow e_j$;
      $U \leftarrow U - \{I_j\}$;
end
```

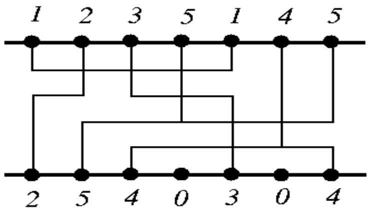
Example: Left-Edge Under Vertical Constraints

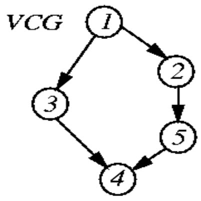


result from basic
left–edge algorithm
3 tracks



optimal routing: 2 tracks



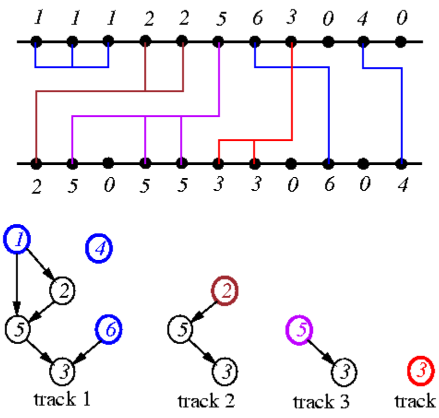


Constrained Left-Edge Algorithm

```
Algorithm: Constrained_Left-Edge($U$, $track[j]$)
    $U$: set of unassigned intervals (nets) $I_1, \ldots, I_n$;
    $I_j=[s_j, e_j]$: interval $j$ with left-end $x$-coordinate $s_j$ and right-end $e_j$;
    $track[j]$: track to which net $j$ is assigned.

begin
$U \leftarrow \{ I_1, I_2, \ldots, I_n\}$;
$t \leftarrow 0$;
while ($U \neq \emptyset$) do
    $t \leftarrow t + 1$;
    $watermark \leftarrow 0$;
    while (there is an unconstrained $I_j \in U$ s.t. $s_j > watermark$) do
        Pick the interval $I_j \in U$ that is unconstrained,
            with $s_j > watermark$
        $track[j] \leftarrow t$;
        $watermark \leftarrow e_j$;
        $U \leftarrow U - \{I_j\}$;
end
```

Constrained Left-Edge Example

- $I_1 = [1, 3]$, $I_2 = [1, 5]$, $I_3 = [6, 8]$, $I_4 = [10, 11]$, $I_5 = [2, 6]$, $I_6 = [7, 9]$.
- **Track 1: Route I_1 (cannot route I_3); Route I_6; Route I_4.**
- **Track 2: Route I_2; cannot route I_3.**
- **Track 3: Route I_5.**
- **Track 4: Route I_3.**



Dogleg Channel Router

- **Deutch**
  - "A dogleg channel router," 13rd DAC, 1976.
- **Motivation:**
  - Left-Edge algorithm cannot handle constraint cycles.
- **Solution:**
  - Doglegs are used to resolve constraint cycle.

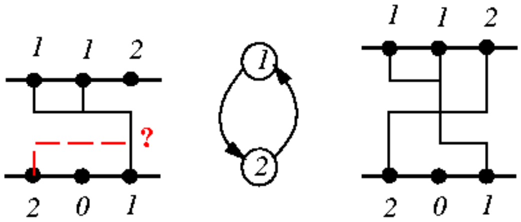
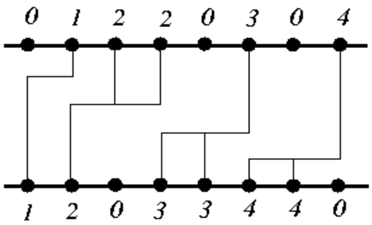


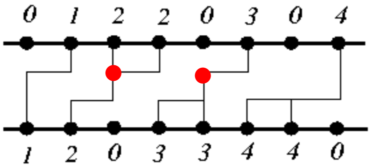
Illustration of Dogleg

- **Left-Edge:**
  - The entire net is on a single track.
- **Dogleg Strategy**
  - Doglegs are used to split the horizontal parts of a net into different tracks to minimize the channel height.
- **Penalty**
  - Additional vias might be needed.



no dogleg

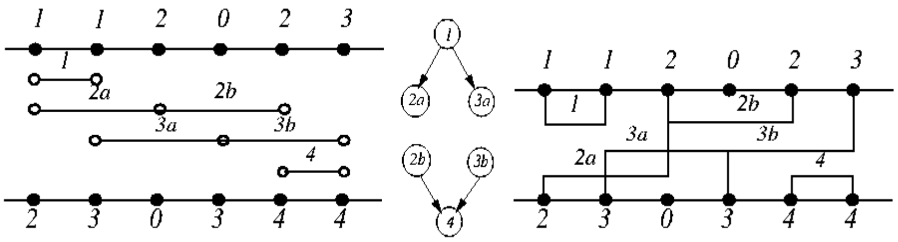
save 2 tracks, with via penalty



with dogleg

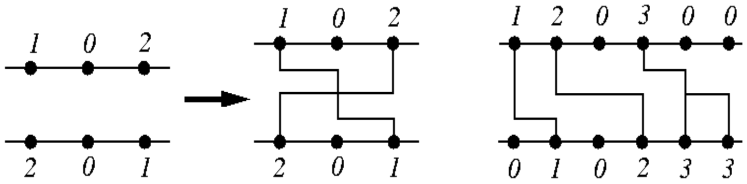
Dogleg Channel Router

- **Basic Idea:**
  - Each multi-pin net is broken down into a set of 2-pin nets.
- **Two parameters are used to control routing:**
  - (1) Range: Determine the # of consecutive 2-terminal subnets of the same net that can be placed on the same track.
  - (2) Routing sequence: Specifies the starting position and the direction of routing along the channel.
- **Modified Left-Edge Algorithm is applied to each subnet.**



Restricted vs. Unrestricted Doglegging

- **Unrestricted doglegging:**
  - Allow a dogleg even at a position where there is no pin.
- **Restricted doglegging:**
  - Allow a dogleg only at a position where there is a pin belonging to that net.
- **The dogleg channel router**
  - does not allow unrestricted doglegging.



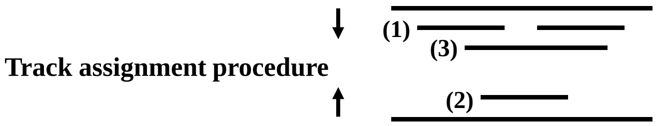
channel router will fail! *Solution exists!* *restricted doglegging splits a net into subnets.*

Channel Routing Strategies At A Glance

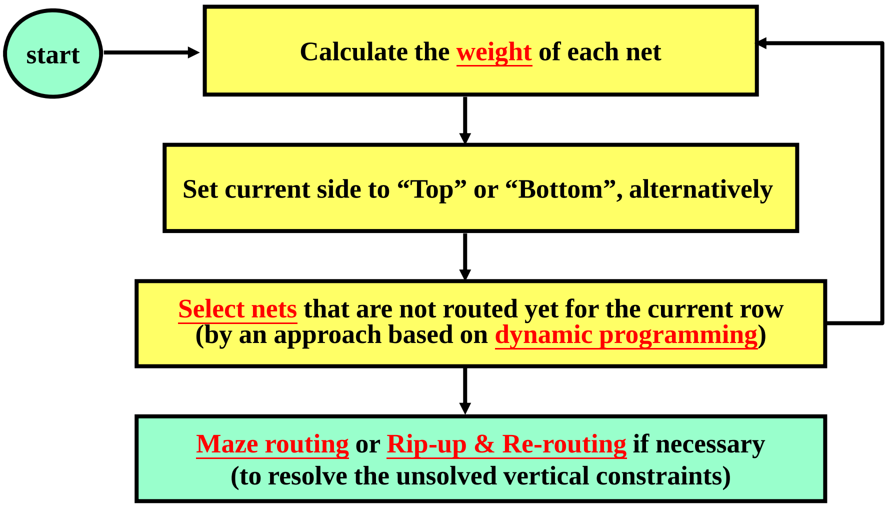
Approaches	Features
Left-Edge Algorithm	**Optimal if no vertical constraints**
Modified Left-Edge Algorithm	**Feasible for vertical constraints**
Dogleg Channel Routing	**Doglegging For cyclic V-constraints**
Robust Channel Router	**Unrestricted Doglegging**

Robust Channel Router

- **Yoeli,**
  - **"A robust channel router," IEEE TCAD, 1991.**
- **Track assigning procedure**
  - **(1) From top and bottom sides towards to the center of channel.**
  - **(2) Alternates between top and bottom tracks towards center.**
  - **The working side is called the *current side*.**
- **Weights**
  - **are used to guide the assignment of segments in a track, which**
  - **(1) favor nets that contribute to the channel density;**
  - **(2) favor nets with terminals at the current side;**
  - **(3) penalize nets whose routing at the current side would cause vertical constraint violations.**
- **Allows unrestricted doglegs by rip-up and re-route.**



Outline of Robust Channel Router



start

Calculate the weight of each net

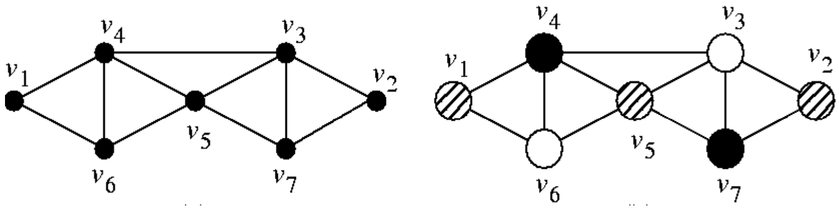
Set current side to “Top” or “Bottom”, alternatively

Select nets that are not routed yet for the current row
(by an approach based on dynamic programming)

Maze routing or Rip-up & Re-routing if necessary
(to resolve the unsolved vertical constraints)

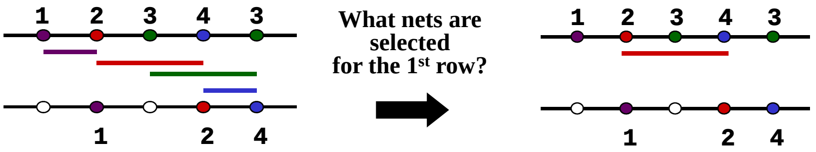
Interval Graphs

- Vertex:
  - There is a vertex for each interval.
- Edge:
  - Vertices corresponding to overlapping intervals are linked by an edge.
- The net selection problem (I.e., selecting nets to one row)
  - is equivalent to finding a minimal vertex coloring of the graph.



Net Selection For One Row

- **The nets to be put in one row of the current side**
  - Is done by selecting maximum weighted compatible set in the interval graph.
  - NP-complete for general graphs, but can be solved efficiently for interval graphs using dynamic programming.
- **Main ideas:**
  - The interval for net *i* is denoted by $[x_{i_{min}}, x_{i_{max}}]$; its weight is w_i.
  - (1) Process each channel column from left to right column;
  - The optimal benefit for position *c* is denoted by total[*c*];
  - (2) A net *n* with a rightmost terminal at position *c* is taken into the candidate set if $w_n + \text{total}[x_{n_{min}} - 1] > \text{total}[c - 1]$



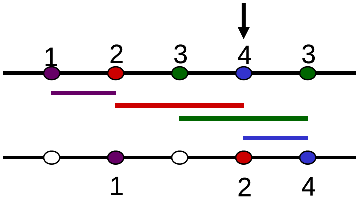
What nets are selected for the 1st row?

Candidate Selection Criterion

Net *n* is selected as a condidate if the following holds:

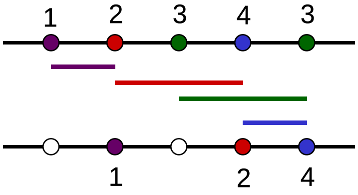
$$\underbrace{w_n}_{\text{Benefit if Selecting net } n} + \underbrace{\text{total}[x_{n_{min}} - 1]}_{\text{Total benefits up to } x_{n_{min}}} > \underbrace{\text{total}[c - 1]}_{\text{Total benefit if Not selecting net } n}$$

E.g., at current position *c* = 4



Assume:
W_2 = 987, Total [1] = 0, Total [3] = 0
➔W_2 + Total [1] > Total [3]
➔Net 2 is included as a candidate

Weight Computation

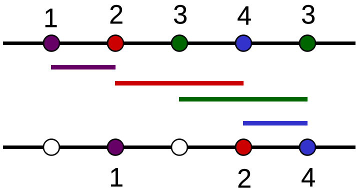


(Density at each column position)

d(1) = 1
d(2) = 2
d(3) = 2
d(4) = 3 (nets 2, 3, 4)
d(5) = 2

- **Computation of the weight w_i for net i:**
  1. **favor nets that contribute to the channel density: add a large B to w_i.**
  2. **favor nets with current side terminals at column x: add d(x) to w_i.**
  3. **penalize nets whose routing at the current side would cause vertical constraint violations: subtract Kd(x) from w_i, K = 5 ~ 10.**
     - **Assume B = 1000 and K = 5 in the 1st iteration (top side):**
       - **$w_1 = (0) + (1) + (-5 * 2) = -9$**
       - **Net 1 does not contribute to the channel density**
       - **One net 1 terminal on the top**
       - **Routing net 1 causes a vertical constraint from net 2 at column 2 whose density is 2**

Weight Computation (cont'd)

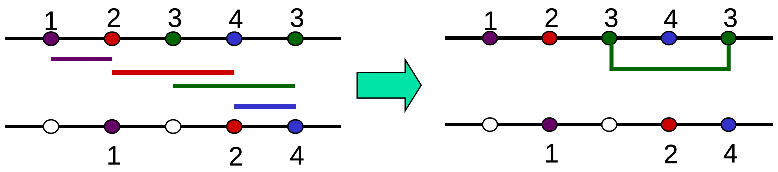


d(1) = 1
d(2) = 2
d(3) = 2
d(4) = 3 (nets 2, 3, 4)
d(5) = 2

- **Computation of the weight w_i for net i:**
  1. **favor nets that contribute to the channel density: add a large B to w_i.**
  2. **favor nets with current side terminals at column x: add d(x) to w_i.**
  3. **penalize nets whose routing at the current side would cause vertical constraint violations: subtract Kd(x) from w_i, K = 5 ~ 10.**
     - **Assume B = 1000 and K = 5 in the 1st iteration (top side):**
       - **$w_1 = (0) + (1) + (-5 * 2) = -9$**
       - **$w_2 = (1000) + (2) + (-5 * 3) = 987$**
       - **$w_3 = (1000) + (2+2) + (0) = 1004$**
       - **$w_4 = (1000) + (3) + (-5 * 2) = 993$**

兩賞一罰
+ Channel density contributor
+ Current side terminals
- Vertical constraint violator

1^{st} Iteration: Top-Row Net Selection



- $w_1 = -9$, $w_2 = 987$, $w_3 = 1004$, $w_4 = 993$.
- **A net *n* with a rightmost terminal at position *c* is taken into the candidate set if: $w_n + \text{total}[x_{n_{min}} - 1] > \text{total}[c - 1]$.**

Column ID

Column ID

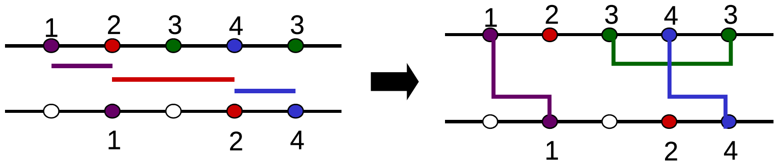
Net ID

total[1] = 0	selected_net[1] = 0
total[2] = max(0, 0-9) = 0	selected_net[2] = 0
total[3] = 0	selected_net[3] = 0
total[4] = max(0, w_2+total[1]) = 987	selected_net[4] = 2
total[5] = max(987, 0+1004, 0+993) = 1004	selected_net[5] = 3

Best net Ending col. 5

- **Select those nets not violating horizontal constraints backwards from right to left: Only net 3 is selected for the top row. (Net 2 is not selected since it overlaps with net 3.)**

2^{nd} Iteration: Bottom-Row Net Selection



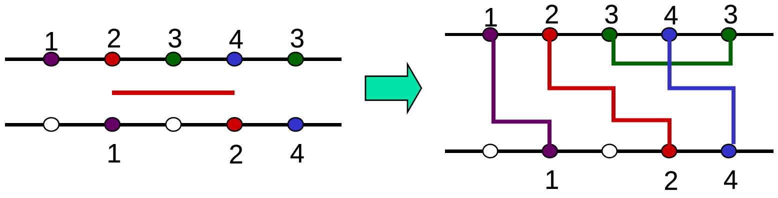
- **2^{nd} iteration: bottom-row selection**
  - **w_1 = (1000) + (2) + (0) = 1002**
  - **w_2 = (1000) + (2) + (-5 * 2) = 992**
  - **w_4 = (1000) + (1) + (-5 * 2) = 991**

(1) Forward scan
- pick candidate nets
(2) Backward scan
- decide max. compatible set

total[1] = 0	selected_net[1] = 0
total[2] = max(0, 0+1002) = 1002	selected_net[2] = 1
total[3] = 1002	selected_net[3] = 0
total[4] = max(1002, 0+992) = 1002	selected_net[4] = 0
total[5] = max(1002, 1002+991) = 1993	selected_net[5] = 4

- **Nets 4 and 1 are selected for the bottom row.**

Maze Routing + Rip-up & Re-route



- 3rd iteration
  - Routing net 2 in the middle row leads to an infeasible solution.
  - Apply maze routing and rip-up and re-route nets 2 and 4 to fix the solution.

Concluding Remarks

- Routing In One Shot
  - Maze routing or line search routing
- Routing In Stages (Divide-and-Conquer)
  - (1) Routing Area Decomposition
  - (2) Global Routing
  - (3) Detailed Routing
- Global Routing
  - To find minimum rectilinear Steiner tree
  - Good heuristics are available
- Detailed Channel Routing
  - Getting around vertical and horizontal constraints
  - Modified left-edge, Robust router, etc.

Routing In A Maze,

It Is Important That All Mice Find Their Ways Out !

Lex & Yacc Tutorial

Instructor: 清華大學　黃錫瑜

Acknowledgements:
C.C. Weng 翁嘉謙
M.C. Li 李明治

July 20, 2012
Department of Electrical Engineering
National TsingHua University, HsinChu, Taiwan

Outline

- *Overview*
- **Lex: A Lexical Analyzer Generator**
- **Yacc: Yet Another Compiler-Compiler**
- **Case Study**

History of Lex & Yacc

- **Lex & Yacc were both developed at Bell Lab. in the 1970s.**
- **Yacc was developed as the first of the two by Stephen C. Johnson.**
- **Lex was designed by Mike E. Lesk and Eric Schmidt to work with Yacc.**
- **Standard UNIX utilities**

Who Needs Lex & Yacc ?

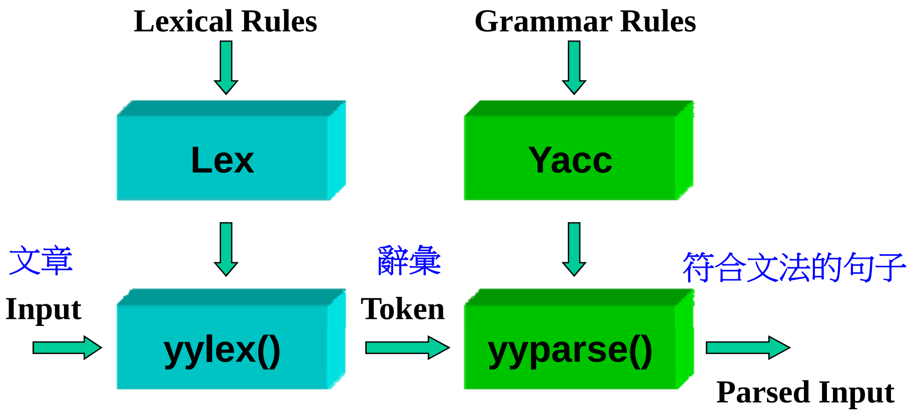
- **Lex & Yacc are programming tools designed for**
  - **Writers of compiler and interpreters**
  - **Non-compiler-writers**
- **Any application looking for patterns in its input or having an input/command language is a candidate for Lex/Yacc.**

Why Lex & Yacc ?

- **Lex & yacc help you write programs that transform structured input**
  - **Lex: generate a lexical analyzer**
    - **(將文章分成一個一個辭彙)**
      - **Divide an input stream into tokens**
      - **Pass the tokens to Yacc**
  - **Yacc: generate a parser**
    - **(將一個一個辭彙組成符合自訂文法的句子)**
      - **Grammar checking**
      - **Create an interpreter**

Ex: 德語裡，*動詞*放在受詞的*後面*

Lex with Yacc



A Simple Example

- **Build a program that recognizes different types of English words**
- **Extend it to handle multiword sentences that conform to a simple English grammar**
- **Vocabulary set:** 辭彙

Noun: Tom, Mary, apple, dog, cat
Pronoun: I, you, they, we
Verb: love, hate,

- **Valid sentence grammar: subject + verb + object**
  文法

Is "Mary hate dog" a valid sentence?

Recognizing Word w/ Lex

```
%{
#include  "y.tab.h"
%}
%%
[ \t]+      ;
Tom |
Mary |
apple |
dog |
cat         {return (NOUN);}
I |
you |
they |
```

```
we          {return (PRONOUN);}
love |
hate        {return (VERB);}
.|\n        {ECHO;}
%%

main( )
{

  yylex();

}
```

Mary	Noun
hate	Verb
dog	Noun

Checking Grammar w/ Yacc

```
%{
#include <stdio.h>
%}

%token NOUN PRONOUN VERB

%%
sentence: subject VERB object
              {printf(“valid\n”);}
            ;
subject:   NOUN
            |  PRONOUN
            ;
```

```
object:   NOUN
            ;
%%
extern FILE *yyin;
main ()
{
    do{
        y
       }v
}
yyerro
char *s;
{ fprintf(stderr,”%s\n”,s);}
```

Mary	Noun	subject
hate	Verb	
dog	Noun	object

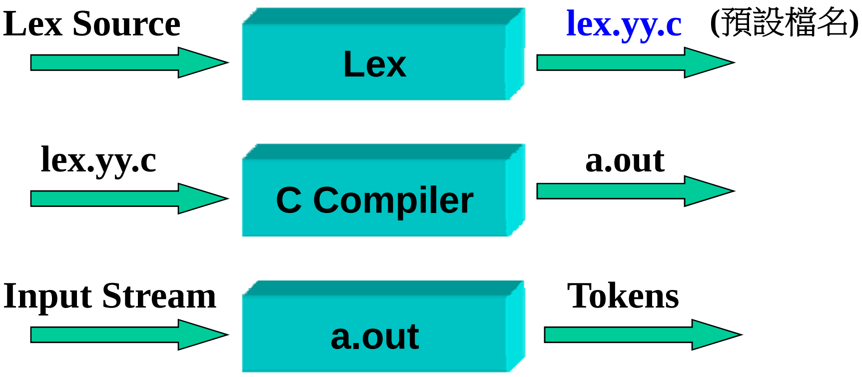
References

1. John R. Levine, “lex & yacc,” *O’REILLY*, 1992.
2. M.E. Lesk and E.Schmidt, “Lex – A Lexical Analyzer Generator,” Tech. Rep., Bell Laboratories, 1975.
3. S.C. Johnson, “Yacc: Yet Another Compiler-Compiler,” Tech. Rep, Bell Laboratories, 1975.
4. Thomas Nierman, “A Compact Guide to Lex & Yacc”, *ePaper Press.*

Outline

- **Overview**
- ***Lex: A Lexical Analyzer Generator***
  - ***Lex Source Format***
  - **Lex Regular Expressions**
  - **Lex Actions**
  - **Usage**
- **Yacc: Yet Another Compiler-Compiler**
- **Case Study**

An Overview of Lex



Format of Lex Source

- **Lex source consists of three parts:**

```
{definitions section}
%%
{rules section}
%%
{user subroutines}
```

  - **Separated by lines consisting of %%**
  - **The first two sections are required, can be empty**
  - **The absolute minimum lex program is:**

```
%%
```

→ **Copy the input to the output unchanged**

Definition Section

- **Can include the *included code*, *name translation*, *start conditions* and *changes to internal setting***
- **An Example:**

```
        int count;    /*<space>  <code> */
%{
int words_count;
int lines_count;
void foo( );
%}
W   [a-zA-Z]
D   [0-9]
%Start state1 state2
```

將這段 **code** 加到 **lex.yy.c** 的最前面

Rules Section

- **Contains *regular expression* and *actions*, program fragments to be executed when expressions are recognized**
- **An example:**

```
%%
[ \t]      ;              /*no action*/
\n         {lines_count++;}
apple      {ECHO;}    /* <regexp>   <action> */
{W}+       {printf("find a word %s\n", yytext);}
<state1>man       {printf("a man in state1\n");}
<state2>man       {printf("a man in state2\n");}
{D}+       {foo( );}
```

User Subroutines Section

- **Includes user-defined routines called from the rules, and the redefined *input(), output(), unput() or yywrap()***
- **The content in this section is copied verbatim to C file.**

```
%%
main( )
{
    yylex( );
    printf ("word count = %d\n", words_count);
}
void foo( ){
…
}
```

One Simple Example

```
%%
.|\n      ECHO; /* matches any character or a
                     new line */

%%
```

This program copies standard input to standard out!!

Another Example

A word count program

```
%{
int charCount=0, wordCount=0, lineCount=0;
%}
word  [^ \t\n]+
%%
{word}  {wordCount++; charCount+=yyleng;}
\n      {charCount++; lineCount++;}
.       charCount++;
%%
main ( ){
  yylex ( );
  printf (“%d %d %d”, charCount, wordCount, lineCount);
}
```

Outline

- **Overview**
- ***Lex: A Lexical Analyzer Generator***
  - **Lex Source Format**
  - ***Lex Regular Expressions***
  - **Lex Actions**
  - **Usage**
- **Yacc: Yet Another Compiler-Compiler**
- **Case Study**

Lex Regular Expressions

- **Specify a set of strings to be matched**
- **Contain text characters and operator characters**
  - **Operators**
  - **Character classes**
  - **Arbitrary character**
  - **Optional expressions**
  - **Repeated expressions**
  - **Alternation and Grouping**
  - **Context sensitivity**
  - **Repetitions and Definitions**

Operators

- **The set of operator characters:**

“ \ [ ] ^ - ? . * + | ( ) $ / { } % < >

- **If used as text characters, an escape should be added.**

***xyz* “+ +” = “xyz++” = xyz\+\+ → xyz++**

- **Any blank not contained within [ ] must be quoted.**
- **Every character but *blank*, *tab (\t)*, *newline (\n)*, and the list above is always a text character.**

Character Classes

- **Class of characters can be specified using the operator pair *[ ]*.**
- **Most operator meanings are ignored except**
  - **\ (turn into ASCII character),**
  - **– (indicate range),**
  - **^ (except)**

[abc] => a or b or c
[a-z] => from a to z
[-+0-9] => all the digits and the two signs
[^a-zA-Z] => any character which is not a letter
[\40-\176] => from octal 40 (blank) to octal 176 (tilde~)

Arbitrary Character

- **The operator character . matches all characters except newline.**
- **[\40-\176] matches all printable characters in the ASCII character set.**

Optional & Repeated Expressions

a? => zero or one instance of a

a* => zero, one, or more instances of a

a+ => one or more instances of a

ab?c => ac or abc
[a-z]+ => all strings of lower case letters
[A-Za-z][a-zA-Z0-9]* => all alphanumeric strings with a leading alphabetic character

d923940 ?

Alternation and Grouping

- **The operator | indicates alternation.**
- **The parentheses ( ) can be used for grouping.**

```
(ab|cd)  =   ab|cd  =>  ab or cd
(ab|cd+)?(ef)*   => abefef, efefef, cdef, or cddd
                    but not abc, abcd, or abcdef
```

cdcdef ?

Context Sensitivity

- **The operator ^ means the expression is matched from the beginning of the line.**

```
^ab  => matches the string ab, but only if ab is at the start of the line
```

- **The operator $ means the expression is matched from the end of the line.**

```
ab$  => matches the string ab, but only if ab is at the end of the line
```

- **The operator / indicates trailing context.**

```
ab/cd => matches the string ab, but only if followed by cd
ab/\n  =  ab$
```

Repetitions and Definitions

- The operators { } specify
  - Repetitions (if enclosing a number)
  - Definition expansion (if enclosing a name)

a{1,5} => 1 to 5 occurrences of a
{digit} => inserts a predefined string named *digit*
(The string is defined in *definition section*)

Regular Expression Summary

Regexp	Description
x	the character "x"
"x"	an "x", even if x is an operator
\x	an "x", even if x is an operator
[xy]	the character x or y
[x-z]	the characters x, y or z
[^x]	any character but x
.	any character but newline
^x	an x at the beginning of a line
<y>x	an x when Lex is in start condition y

Regexp	Description
x$	an x at the end of a line
x?	an optional x
x*	0, 1, 2, … instances of x
x+	1, 2, 3, … instances of x
x|y	an x or a y
(x)	an x
x/y	an x but only if followed by y
{xx}	the translation of xx from the definitions section
x{m,n}	*m* through *n* occurrences of x

Outline

- **Overview**
- ***Lex: A Lexical Analyzer Generator***
  - **Lex Source Format**
  - **Lex Regular Expressions**
  - ***Lex Actions***
  - **Usage**
- **Yacc: Yet Another Compiler-Compiler**
- **Case Study**

Lex Actions

- **When an expression is matched**
  - **Lex executes the corresponding action, i.e., a C program fragment**
- **The action character | indicates the action for this rule is the action for the next rule.**

```
%%
regexp <one or more blanks> {action (C codes)}

-?(([0-9]+)|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?)  {printf(“number\n”);}
[ \t\n]   ;        => ignore the three spacing characters
“ ” |
“\t” |
“\n” ;             => the same as [ \t\n]
%%
```

More Lex Actions

- **Lex predefined variable *yytext* is the pointer to the matched string.**
- ***yyleng* indicates the length of matched string.**
- **The action of *ECHO* is to print the matched string.**

```
[a-z]+   printf("%s", yytext);
[a-z]+   ECHO;          =>  the same
[a-zA-Z]+       {wordsCount++;charsCount+=yyleng;}
```

yytext[yyleng-1] ⇨ **The last character of the matched string**

Ambiguous Source Rules

- **When more than one expression can match the current input,**
  - **The longest match is preferred**
  - **The rule given first is preferred**

```
is | am | are       {printf("Verb\n");}
ambiguous           {printf("ADJ\n");}
[a-zA-Z]+           {printf("Unknown\n";}
```

How does lex choose the action when the input is "ambiguous"?

⇨ **ADJ**

Lex Action REJECT

- **The action REJECT means "go do the next alternative"**
  - **To override the rules**

有點像 Recycle Matched token

```
she       s1++;
he        h1++;
. |
\n        ;
```

```
she      {s2++;REJECT}
he       {h2++;REJECT}
. |
\n        ;
```

When the input is "she"

⇨ **s1=1; h1=0;**
s2=1; h2=1;

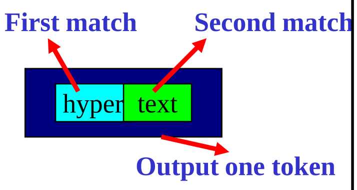
More Details – yymore

- **yymore( )**
  - **Append the next matched token to the end of the current matched token**

串連好幾個 Matched tokens 到 yytext 裡

```
%%
hyper  {yymore();}
text    {printf("Token is %s\n", yytext);}
```

Input: "hypertext"
Output: "Token is hypertext"



First match

Second match

hyper text

Output one token

More Details – yyless

- yyless(n)
  - – Push back all but the first *n* characters of the token
- Consider a string "= – a"

```
=-[a-zA-Z]  {printf("Op: =-\n");
              yyless(yyleng-1);
             …action for =- …
             }
```

➡ `=-/[A-Za-z]`

=-
a

```
=-[a-zA-Z]  {printf("Op: =\n");
              yyless(yyleng-2);
             …action for = …
             }
```

➡ `=/-[A-Za-z]`

=
-a

Start Conditions

- When only a few rules change from one environment to another
  - – The start conditions can be used to explicitly declare multiple states (in definition section)

```
%Start  state1   state2 …
```

- Different rules are applied according to the corresponding state.

```
<state1>man            printf("a man in state1\n");
<state2>man            printf("a man in state2\n");
```

Example of Start Conditions

- **Consider the following problem:**
  - **copy the input to the output**
  - **changing the word *magic* to *first* on every line which began with the letter *a*,**
  - **changing the word *magic* to *second* on every line which began with the letter *b*,**
  - **changing the word *magic* to *third* on every line which began with the letter *c*.**

```
amagic magic                 afirst first
b magic          ──>         b second
cmagicmagic                  cthirdthird
```

Example of Start Conditions (Cont.)

Using flag

```
int flag;
%%
^a          {flag='a';ECHO;}
^b          {flag='b';ECHO;}
^c          {flag='c';ECHO;}
\n          {flag= 0 ; ECHO;}
magic       {
            switch(flag)
            {
            case 'a': printf("first");break;
            case 'b': printf("second");break;
            case 'c': printf("third");break;
            default: ECHO; break;
            }
            }
```

Using Start Conditions

```
%Start AA BB CC
%%
^a          {ECHO;BEGIN AA;}
^b          {ECHO;BEGIN BB;}
^c          {ECHO;BEGIN CC;}
<AA>magic        printf("first");
<BB>magic        printf("second")
<CC>magic        printf("third")
```

Equivalent

Predefined Variables in LEX

Name	Description
char *yytext	pointer to matched string
int yyleng	length of matched string
FILE *yyin	input stream pointer
FILE *yyout	output stream pointer
int yylex(void)	call to invoke lexer, returns token
char* yymore(void)	return the next token
int yyless(int n)	retain the first n characters in yytext
int yywrap(void)	Wrap-up, return 1 if done, 0 if not done
ECHO	write matched string
REJECT	go to the next alternative rule
INITIAL	initial start condition
BEGIN condition	switch start condition

Outline

- **Overview**
- ***Lex: A Lexical Analyzer Generator***
  - **Lex Source Format**
  - **Lex Regular Expressions**
  - **Lex Actions**
  - ***Usage***
- **Yacc: Yet Another Compiler-Compiler**
- **Case Study**

How to Generate 辭彙解析器 by LEX

Step1: Turn the lex source into a C program

```
lex test.l
```

- **lex.yy.c is then produced, which is a C program for lexical analyzer.**

Step2: Compile lex.yy.c into an executable

```
gcc lex.yy.c -ll
```

Step3: Run the lexical analyzer program

```
./a.out  <  inputfile
```

Versions of Lex

- **AT&T – lex**
  - **http://www.combo.org/lex_yacc_page/lex.html**
- **GNU – flex**
  - **http://www.gnu.org/manual/flex-2.5.4/flex.html**
- **Win32 version of flex**
  - **http://www.monmouth.com/~wstreett/lex-yacc/lex-yacc.html**
- **Cygwin**
  - **http://sources.redhat.com/cygwin/**

Outline

- **Overview**
- **Lex: A Lexical Analyzer Generator**
- ***Yacc: Yet Another Compiler-Compiler***
  - ***What does Parser do***
  - **Introduction to YACC**
  - **How the Parser Works**
  - **Work with Lex**
- **Case Study**

What does Parser do ?

- **Parser invokes scanner for token processing.**
- **Parser analyzes the syntactic structure.**
- **Parser executes the semantic routines.**

Scanner 就是 **Token Recognizer** 辭彙解析器
Parser 就是 文章解析器
Syntactic: 語法的 **(**有關句子的結構**)**
Semantic: 語意的 **(**有關句子的意義**)**

Outline

- **Overview**
- **Lex: A Lexical Analyzer Generator**
- ***Yacc: Yet Another Compiler-Compiler***
  - **What does Parser do**
  - ***Introduction to YACC***
  - **How the Parser Works**
  - **Work with Lex**
- **Case Study**

Introduction of Yacc

- **Yacc source format**

Declarations
%%
Rules (Grammar)
%%
Programs

Declarations

- **C source codes, include files, etc**
- **Token definition**

Declarations

Example :

```
%{
   #include <stdio.h>
%}

%token NOUN PRONOUN VERB ADVERB
   ADJECTIVE PREPOSITION CONJUNCTION
```

Rules

Example :

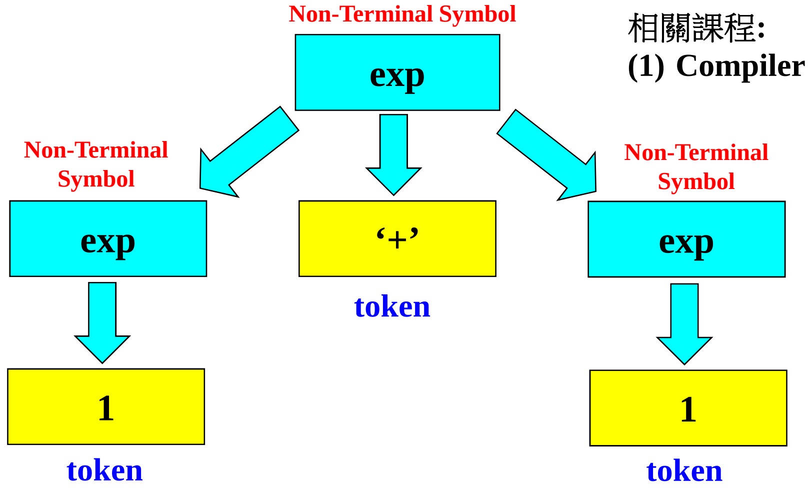
大寫或字串的是 Token
小寫的是 Non-Token
(下一頁進一步解釋…)

```
%token NAME NUMBER
%%
statement:    NAME '=' expression
         |    expression   { printf("= %d\n", $1); }
```

LHS **RHS**

```
expression:   expression '+' NUMBER  { $$ = $1 + $3; }
         |    expression '-' NUMBER  { $$ = $1  - $3; }
         |    NUMBER                  { $$ = $1; }
         ;
```

Parse Tree (如何理解一個句子)



相關課程:
(1) Compiler

Token is also called “**terminal symbol**”, 為基本辭彙

Programs

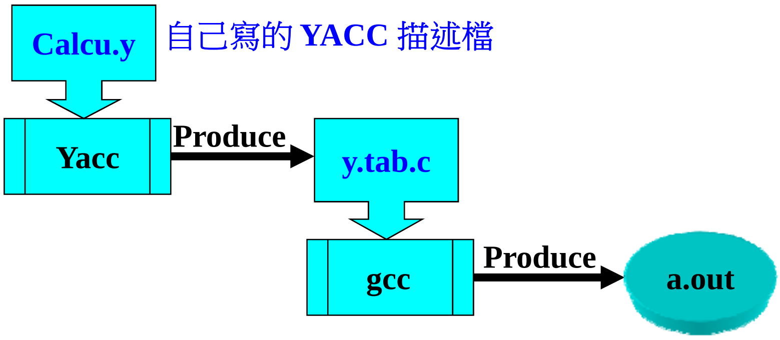
Example :

```
%%
void yyerror(char *s) {
        fprintf(stderr, “%s\n”, s);
}

int main(void) {
        yyparse();
        return 0;
}
```

Outline

- **Overview**
- **Lex: A Lexical Analyzer Generator**
- ***Yacc: Yet Another Compiler-Compiler***
  - **What does Parser do?**
  - **Introduction to YACC**
  - ***How does the Parser Work?***
  - **Work with Lex**
- **Case Study**

How Does the Parser Work?

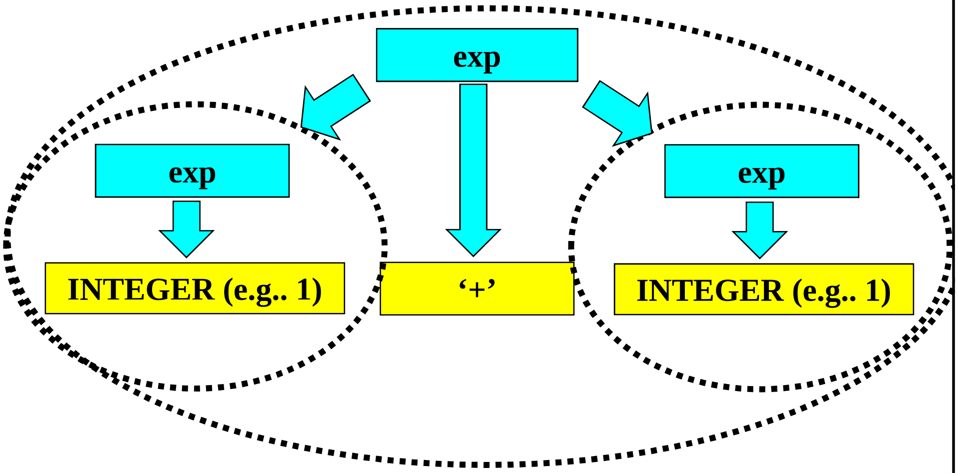


a.out :

Executable program that will parse grammar given in Calcu.y

How Does YACC Process An Article?

It can be done by scanning the tokens in the input file, performing grammar reduction recursively with the aide of a stack…



How Does YACC Work?

- **The parser produced by Yacc consists of a finite state machine with a stack.**

- **The machine has only four actions available to it .**
  - **SHIFT (move on to the next token – keep parsing…)**
  - **REDUCE (a grammar has just matched)**
  - **ACCEPT (the entire article has parsed successfully)**
  - **ERROR (the article does not conform to the grammars)**

A shift and reduce action example

Part of *Rule* section:

```
exp : INTEGER    { $$ = $1 ;}
      | exp '+' exp    { $$ = $1 + $3 ;}
      | exp '-'  exp    { $$ = $1  - $3 ;}
```

Input :
3+1

stack:
<empty>

Reduce and shift!!

A shift and reduce action example

Part of *Rule* section:

```
exp : INTEGER    { $$ = $1 ;}
     | exp '+' exp    { $$ = $1 + $3 ;}
     | exp '-'  exp    { $$ = $1  - $3 ;}
```

Input :
+1

stack:
exp

shift!!

matched no grammar..

A shift and reduce action example

Part of *Rule* section:

```
exp : INTEGER    { $$ = $1 ;}
     | exp '+' exp    { $$ = $1 + $3 ;}
     | exp '-'  exp    { $$ = $1  - $3 ;}
```

Input :
1

stack:
exp +

Reduce and Shift!!

A shift and reduce action example

Part of *Rule* section:

```
exp : INTEGER    { $$ = $1 ;}
    | exp '+' exp   { $$ = $1 + $3 ;}
    | exp '-'  exp   { $$ = $1  - $3 ;}
```

Input :
<empty>

stack:
exp + exp

Continue to Reduce

A shift and reduce action example

Part of *Rule* section:

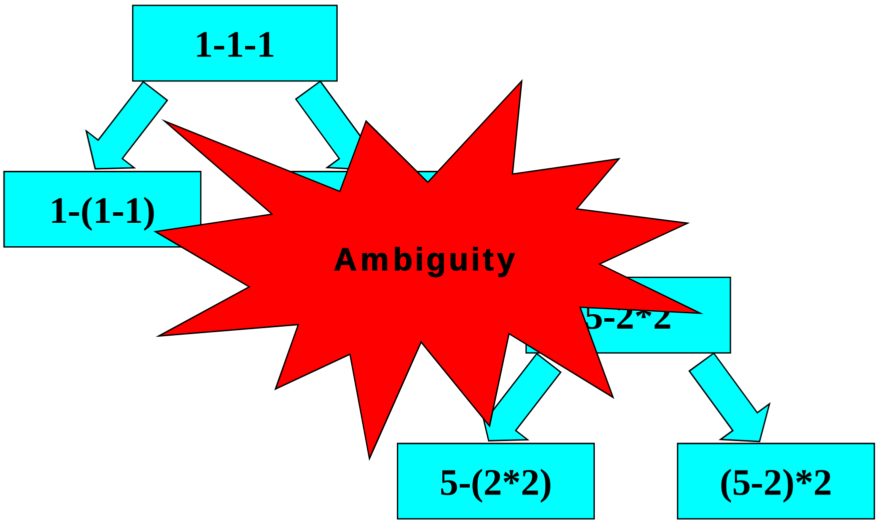
```
exp : INTEGER    { $$ = $1 ;}
    | exp '+' exp   { $$ = $1 + $3 ;}
    | exp '-'  exp   { $$ = $1  - $3 ;}
```

Input :
<empty>

stack:
exp

At the completion of parsing all the input tokens, we conclude that it is an expression

Associative – Left or Right



Precedence & Association

Under “Declaration Section” :

%left '+' '-‘
%left ‘*' ‘/‘

↓ **Higher precedence**

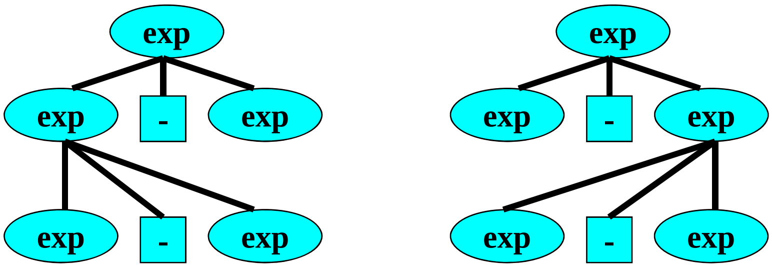
Association :

%left :
A – B – C → (A – B) – C

%right :
A – B – C → A – (B – C)

Left Association

```
exp : INTEGER    { $$ = $1 ;}
      | exp ‘+’ exp    { $$ = $1 + $3 ;}
      | exp ‘-’  exp    { $$ = $1  - $3 ;}
```



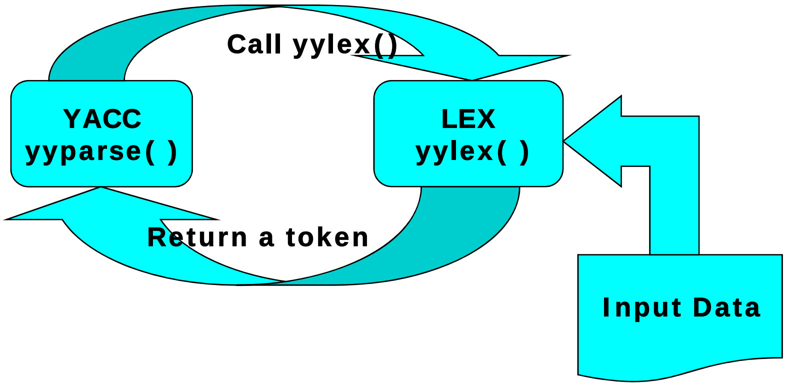
Desired!!!

Outline

- Overview
- Lex: A Lexical Analyzer Generator
- *Yacc: Yet Another Compiler-Compiler*
  - What does Parser do
  - Introduction to YACC
  - How the Parser Works
  - *Work with Lex*
- Case Study

Work with Lex

以 **YACC** 為主，以 **LEX** 為輔



Outline

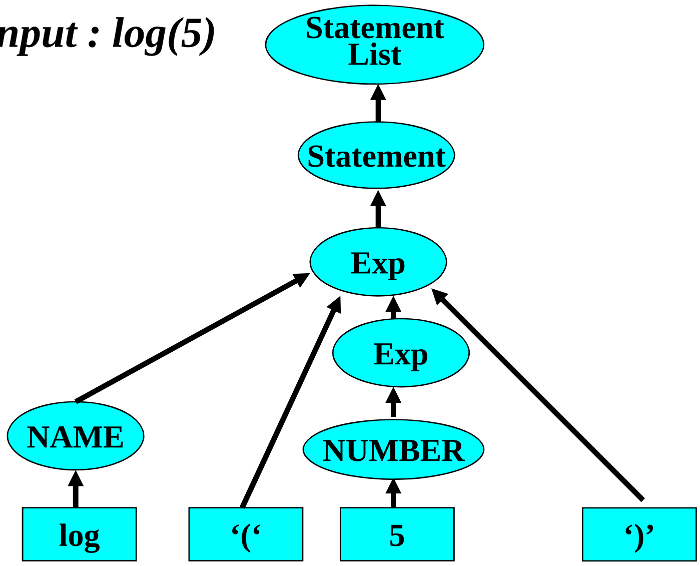
- **Overview**
- **Lex: A Lexical Analyzer Generator**
- **Yacc: Yet Another Compiler-Compiler**
- ***Case Study***
    - *A calculator*

Example

- *Try to realize a calculator by Lex & Yacc*
- *(Provided example files)*
  - ***calculator.h*** **(共同的 header file)**
  - ***calculator.l*** **(LEX 描述檔)**
  - ***calculator.y*** **(YACC 描述檔)**
  - ***exercise***

Example: Parse Tree

Input : log(5)



Exercise of an Extension

- **Please add the power function (e.g. pow(2,3)=8) into the calculator.y**

清華大學 EE 5265
積體電路設計自動化

Appendix 1: IP Design

教育部顧問室
「超大型積體電路與系統設計」教育改進計畫 DIP聯盟

Outline

- ➡ **Brief Introduction of Verilog**
  - **HDL stands for Hardware Description Language**
- **Cell-Based Design Flow**

Verilog HDL (Data-Flow)

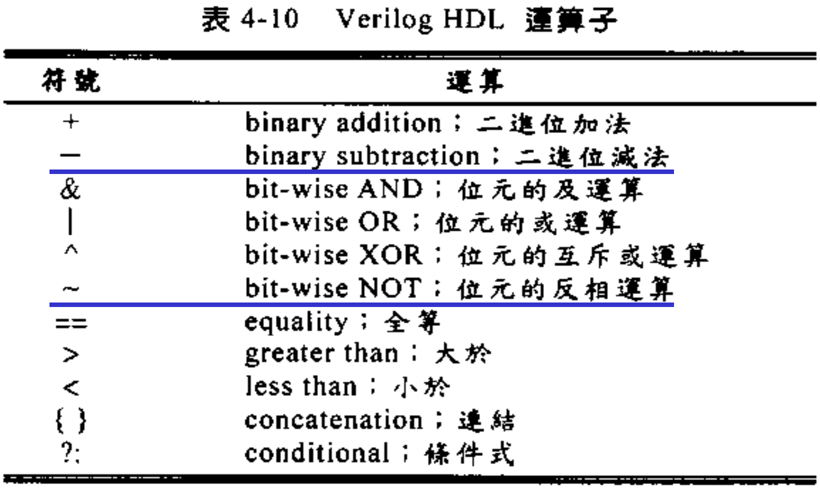


表 4-10 Verilog HDL 運算子

符號	運算
+	binary addition；二進位加法
−	binary subtraction；二進位減法
&	bit-wise AND；位元的及運算
|	bit-wise OR；位元的或運算
^	bit-wise XOR；位元的互斥或運算
~	bit-wise NOT；位元的反相運算
==	equality；全等
>	greater than；大於
<	less than；小於
{ }	concatenation；連結
?:	conditional；條件式

Data-Flow for Adder

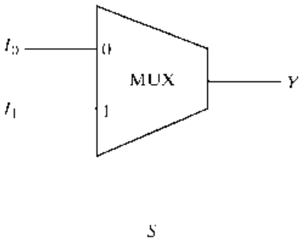
```
// Dataflow description of 4-bit adder
module  binary_adder (A, B, Cin, SUM, Cout);
  input  [3:0] A,B;
  input  Cin; // carry input
  output  [3:0] SUM;
  output  Cout; // carry output
  assign  {Cout, SUM} = A + B + Cin;
endmodule
```

Data-Flow for Comparator

```
// Dataflow description of a 4-bit comparator.
module  magcomp (A, B, ALTB, AGTB, AEQB);
   input  [3:0] A,B;
   output  ALTB,AGTB,AEQB;
   assign  ALTB = (A < B),
                AGTB = (A > B),
                AEQB = (A = = B);
endmodule
```

Data-Flow for 2-To-1 MUX

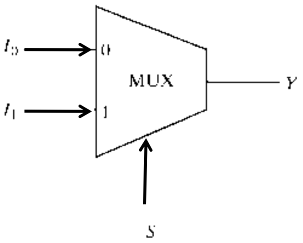
```
// Dataflow description of 2-to-1-line multiplexer
module mux_2x1 _in_data_flow (A,B,select,OUT);
   input  A, B, select;
   output  OUT;
   assign  OUT = select ? A : B;
endmodule
```



Behavior Description for MUX

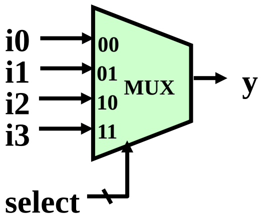
```
// Behavioral description of 2-to-1-line multiplexer
module  behavior_2x1_mux(A, B, select, OUT);
  input  A, B, select;
  output  OUT;
  reg  OUT;
  always @ (select  or  A  or  B)
    begin
      if  (select == 1)  OUT = A;
      else  OUT = B;
    end
endmodule
```

Left-Hand-Side (LHS) variables
Of assignment statements in *always* block
has to be declared as *reg* type of variables
For example, variables *OUT*

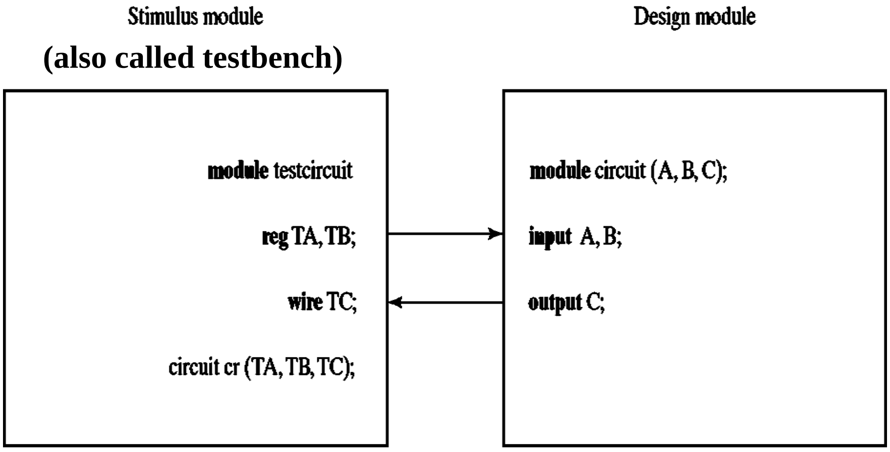


Behavior Description for MUX

```
// Behavioral description of 4-to-1- line multiplexer
module  behavior_4x1_mux (i0, i1, i2, i3, select, y);
   input  i0, i1, i2, i3;
   input  [1:0] select;
   output  y;
   reg  y;
always  @ (i0 or i1 or i2 or i3 or select)
   begin
        case  (select)
            2'b00: y = i0;
            2'b01: y = i1;
            2'b10: y = i2;
            2'b11: y = i3;
        endcase
   end
endmodule
```



Simulation Testbench



Stimulus module

(also called testbench)

```
module testcircuit
reg TA, TB;
wire TC;
circuit cr (TA, TB, TC);
```

Design module

```
module circuit (A, B, C);
input A, B;
output C;
```

Fig. 4-33 Stimulus and Design Modules Interaction

Basics of a Testbench

- *Initial* **block: executed once**
- **$display – dump variables' values with the end-of-line**
- **$write – the same as $display but without the end-of-line**
- **$monitor – dump variables' values when changed**
- **$time – dump simulation time**
- **$finish – terminate simulation**

Syntax: Task_name(format specification, argument list);
Example: $display(%d, %b, %b, C, A, B)
➔ Display C in decimal and A, B in binary

Testbench for Adder

```
// Stimulus for 3-input 2-output circuit analysis
 module  test_circuit;
   reg  [2:0] D;
   wire  F1, F2;

   analysis fig42(D[2], D[1], D[0], F1, F2);

    initial
      D = 3'b000;
       repeat(7)
          #10 D = D + 1'b1;
       end

    initial
       $monitor ("ABC = %b  F1 = %b  F2 =%b ",
                        D, F1, F2);

endmodule
```

```
Simulation log:
ABC=000  F1=0  F2=0
ABC=001  F1=1  F2=0
ABC=010  F1=1  F2=0
ABC=011  F1=0  F2=1
ABC=100  F1=1  F2=0
ABC=101  F1=0  F2=1
ABC=110  F1=0  F2=1
ABC=111  F1=1  F2=1
```

D Flip-Flop

```
//D flip-flop
module D-FF（Q, D ,CLK）；
  output  Q；
  input  D, CLK；
  reg  Q；
always @（posedge CLK）
Q=D；
endmodule
```

```
// D flip-flop with asynchronous reset.
module DFF（Q, D, CLK, RST）；
  output  Q；
  input  D, CLK, RST；
  reg  Q；
always @（posedge CLK or negedge RST）
  if（~RST）Q=1’b0;
 //same as：if（RST ==0）
else Q=D；
endmodule
```

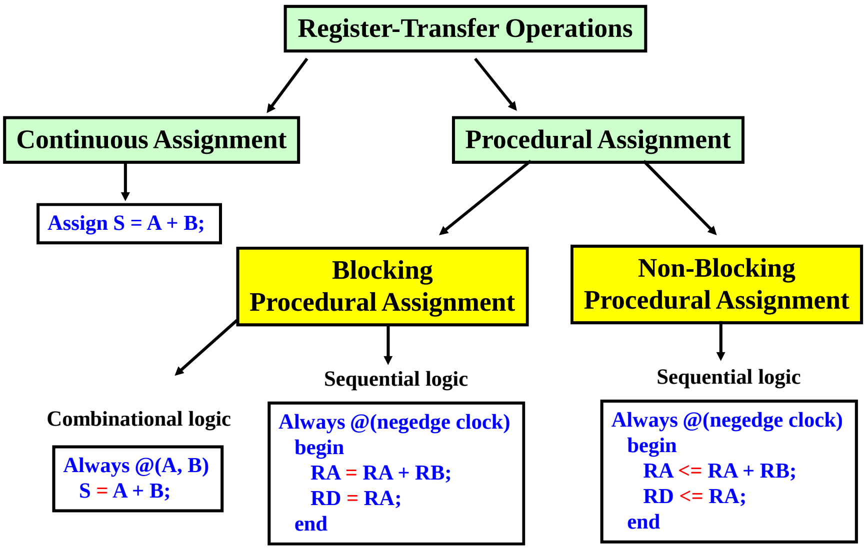
Register-Transfer-Level (RTL)

- **A digital system**
  - **is represented at RTL when it is specified by the following three components**
  - **(1) The set of registers in the system**
  - **(2) Operations performed on registers' values**
  - **(3) The control regulates the operations**

Ex (sequence of RTL operations):

```
R1 ← R1 + R2    // Add contents of R2 to R1
R3 ← R3 + 3     // Increment R3 by 1 (count upwards)
R4 ← shr R4     // Shift right R4
R5 ← 0          // Clear R5 to 0
```

Different Ways of Register-Transfer Operations in Verilog



Register-Transfer Operations

Continuous Assignment

```
Assign S = A + B;
```

Procedural Assignment

Blocking Procedural Assignment

Non-Blocking Procedural Assignment

Combinational logic

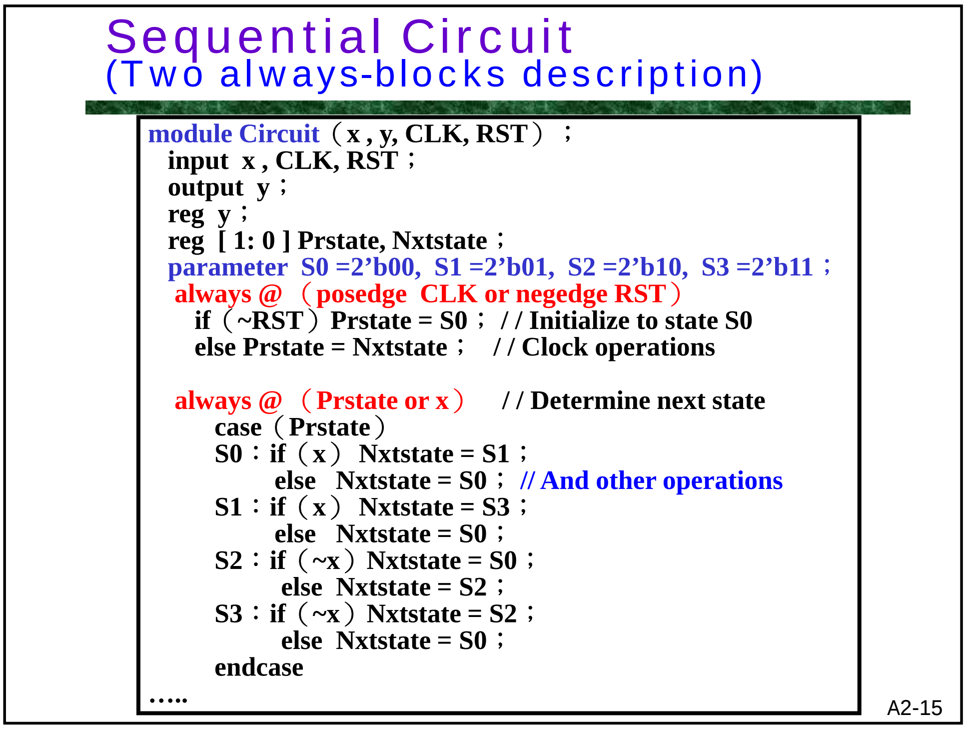
```
Always @(A, B)
  S = A + B;
```

Sequential logic

```
Always @(negedge clock)
  begin
    RA = RA + RB;
    RD = RA;
  end
```

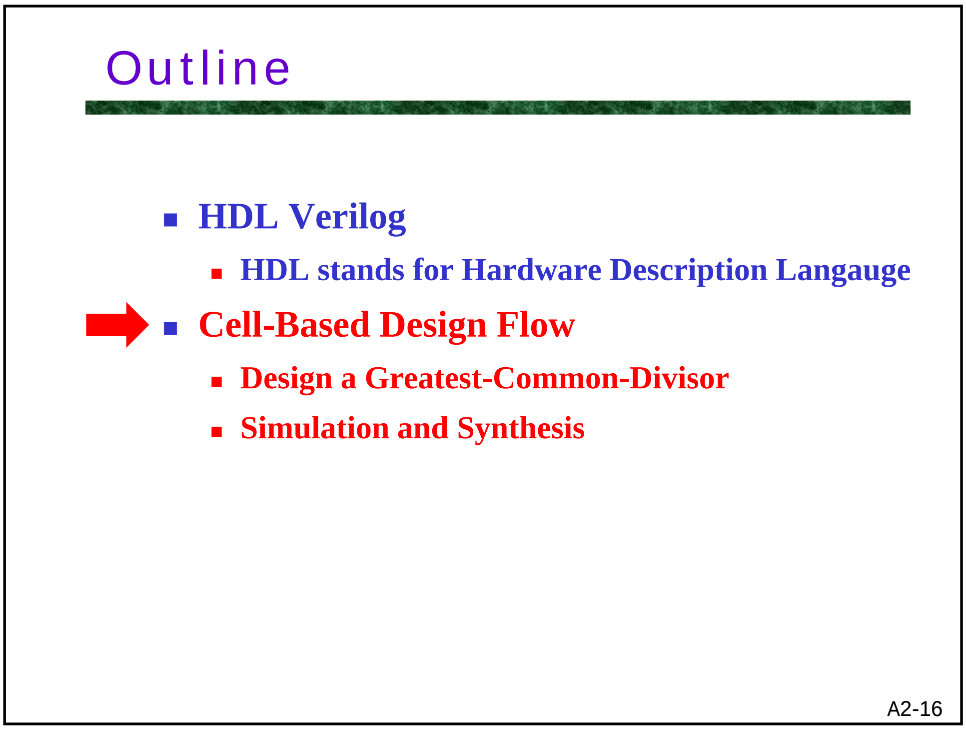
Sequential logic

```
Always @(negedge clock)
  begin
    RA <= RA + RB;
    RD <= RA;
  end
```



Sequential Circuit
(Two always-blocks description)

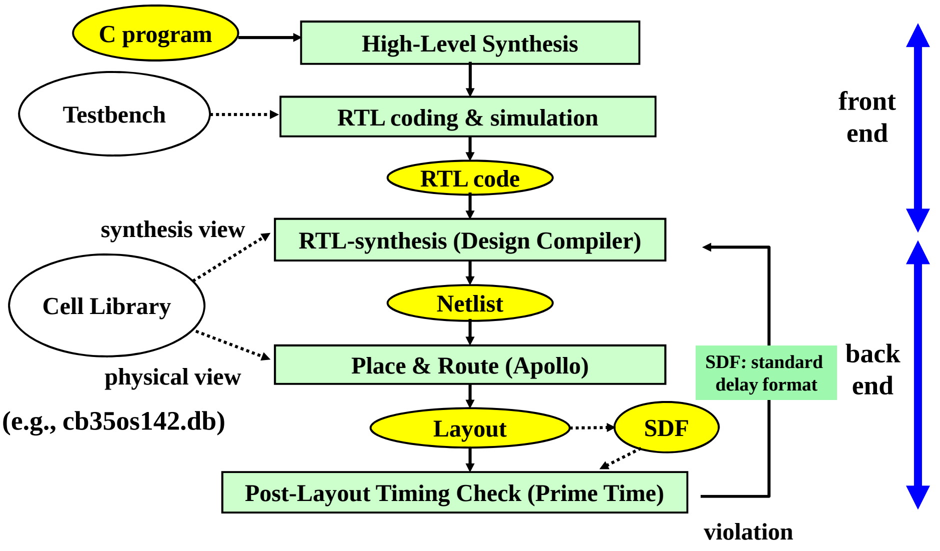
```
module Circuit（x , y, CLK, RST）；
  input  x , CLK, RST；
  output  y；
  reg  y；
  reg  [ 1: 0 ] Prstate, Nxtstate；
  parameter  S0 =2’b00,  S1 =2’b01,  S2 =2’b10,  S3 =2’b11；
   always @ （posedge  CLK or negedge RST）
     if（~RST）Prstate = S0； / / Initialize to state S0
     else Prstate = Nxtstate；   / / Clock operations

   always @ （Prstate or x）     / / Determine next state
        case（Prstate）
        S0：if（x） Nxtstate = S1；
               else   Nxtstate = S0； // And other operations
        S1：if（x） Nxtstate = S3；
               else   Nxtstate = S0；
        S2：if（~x）Nxtstate = S0；
                else  Nxtstate = S2；
        S3：if（~x）Nxtstate = S2；
                else  Nxtstate = S0；
        endcase
…..
```



Outline

- **HDL Verilog**
  - **HDL stands for Hardware Description Langauge**
- **Cell-Based Design Flow**
  - **Design a Greatest-Common-Divisor**
  - **Simulation and Synthesis**

Cell-Based Synthesis Flow



C program

High-Level Synthesis

Testbench

RTL coding & simulation

RTL code

synthesis view

RTL-synthesis (Design Compiler)

Cell Library

Netlist

physical view

Place & Route (Apollo)

(e.g., cb35os142.db)

Layout

SDF

SDF: standard delay format

Post-Layout Timing Check (Prime Time)

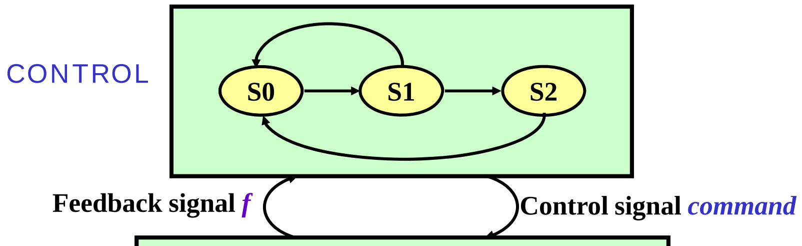
violation

front end

back end

A Design Block

Two-way interactions often exist between CONTROL and DATA



DATA

```
case (command):
  `RESET:      c = 0;
  `NOP:        c = c;
  `INCREMENT: c = c + 1;
endcase
assign f = (c==6) ? 1 : 0;
```

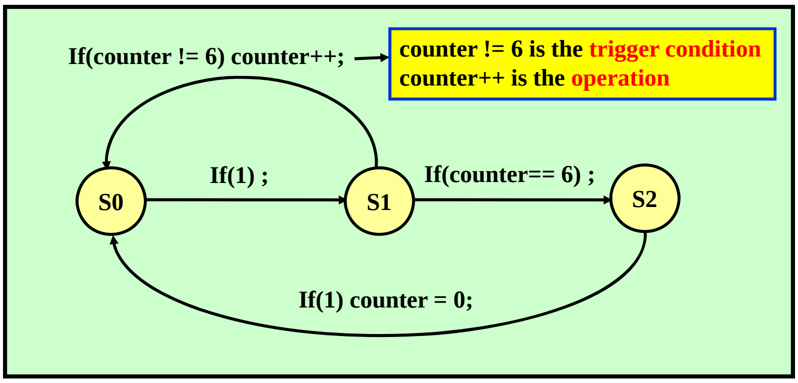
Why EFSM?

- **Extended Finite State Machine (EFSM)**
  - **is a high-level graphic representation of a design**
  - **combines the CONTROL and DATA in a single model**
  - **captures the design intensions easily**
  - **Also called Algorithmic Finite State Machine**

Example of An EFSM

FSM: a transition is associated with Boolean input conditions and a set of Boolean output operations.

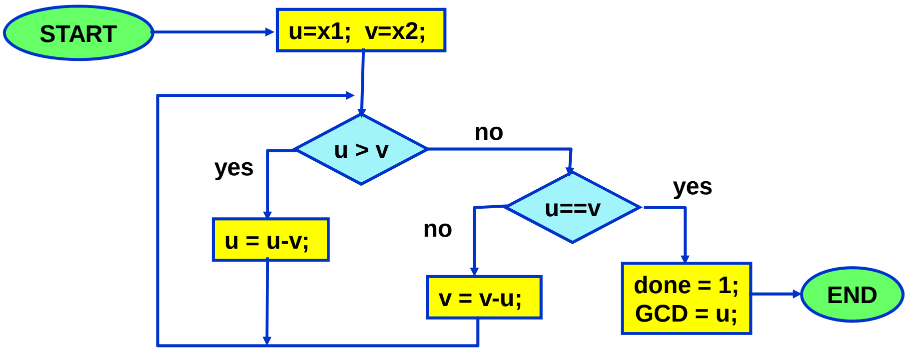
Extended FSM: a transition is modeled by an "if statement"



counter != 6 is the trigger condition
counter++ is the operation

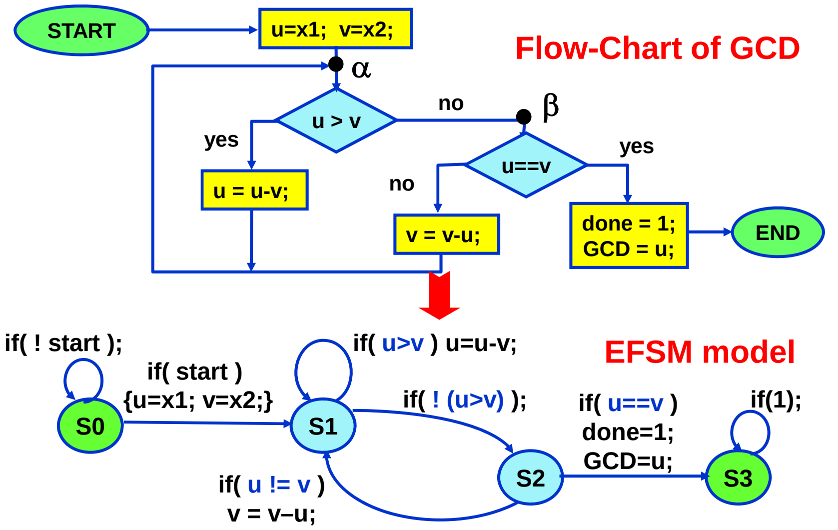
Ex1: Greatest Common Divisor

Inputs: two natural numbers x1 and x2
Output: the greatest common divisor of x1 and x2
Example: (9, 6) → (3, 6) → (3, 3) → Found GCD = 3



Flow-Chart of GCD

From Flow-Chart To EFSM

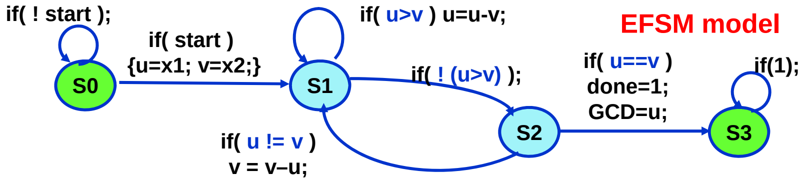


Flow-Chart of GCD

EFSM model

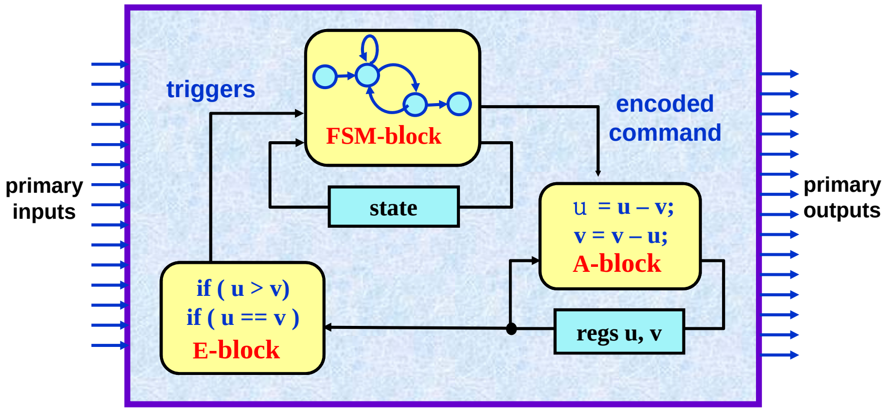
One-Module RTL Coding

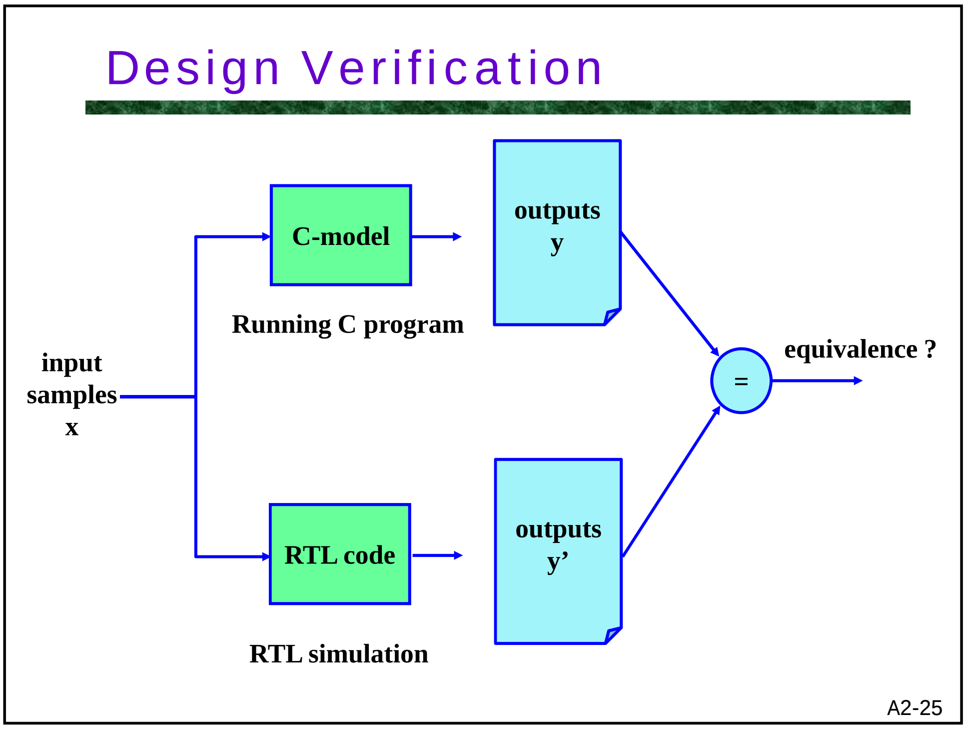
```
case(state)
      `S0:
          if(start) begin
              next_u <= x1; next_v <= x2; next_state <= `S1;
          end
      `S1:
          if(u > v) next_u <= u – v;
          else      next_state <= `S2;
      `S2:
          if(u==v) done<=1; GCD<=u; next_state <= `S3;
          else      next_v = v – u; next_state <= `S1;
      `S3: begin end
endcase
```



Three-Block Architecture

FSM-block: for controller
A-block: for data operation
E-block: for trigger evaluation





Design Verification

input samples x

C-model

Running C program

outputs y

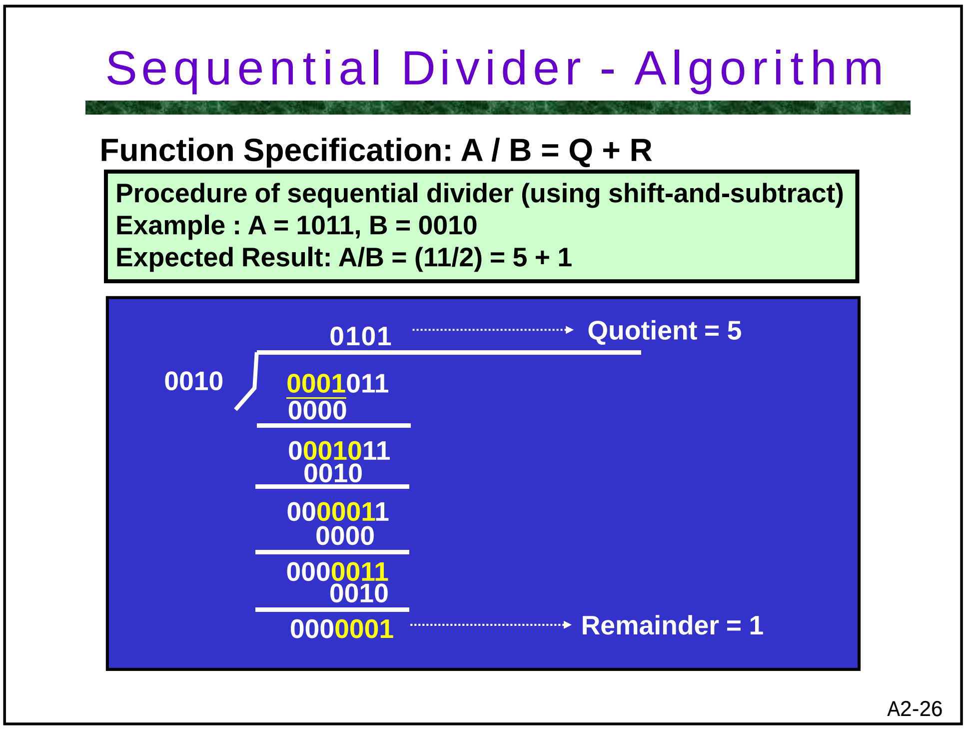
RTL code

RTL simulation

outputs y'

=

equivalence ?



Sequential Divider - Algorithm

Function Specification: A / B = Q + R

Procedure of sequential divider (using shift-and-subtract)
Example : A = 1011, B = 0010
Expected Result: A/B = (11/2) = 5 + 1

0101 → **Quotient = 5**

0010

0001011
0000

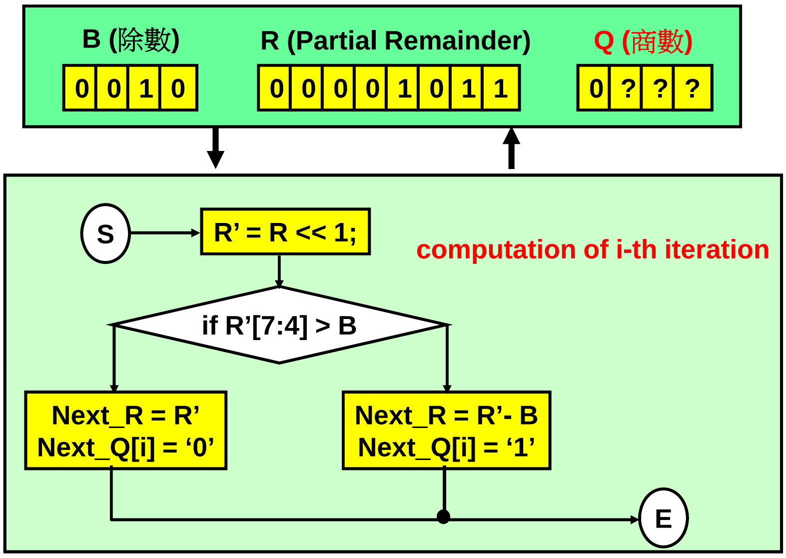
0001011
0010

0000011
0000

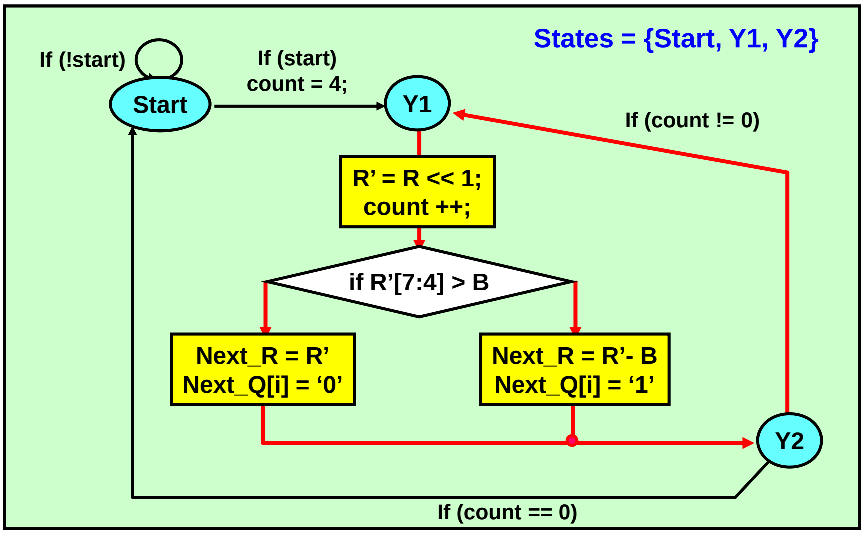
0000011
0010

0000001 → **Remainder = 1**

Flow Char – One Quotient Bit



Complete Flow Chart & States



Synthesis Script

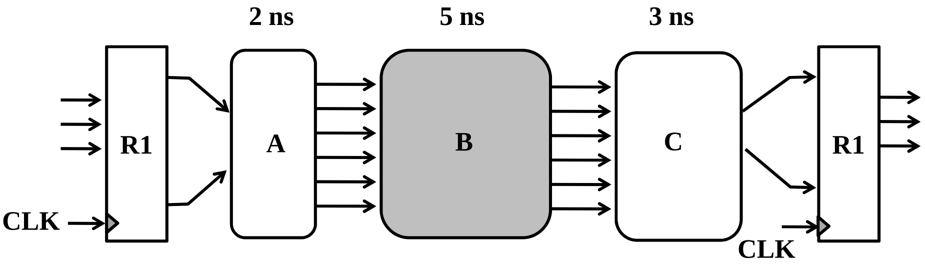
```
design_dir = ../design
io_dir = /home/users/cic/CIC_CBDK35_V3/Synopsys
lib_dir = /home/users/cic/CIC_CBDK35_V3/Synopsys
search_path =  search_path + lib_dir + io_dir + design_dir
define_design_lib Analyzed  -path analyzed_dir
/*-----  (1) Specify Target Libraries   -----*/
target_library = "cb35os142_max.db "
link_library = "cb35os142_max.db “
/*-----  (2) Read in design  -----*/
design_list = { m1.v, m2.v}
read -format verilog design_list
link
/*-----  (3) Set constraints  -----*/
create_clock  -period 2 -waveform  {0, 1}  find(port *clk)
set_input_delay -max 0.0 -clock clk all_inputs()
set_input_delay -min 0.0 -clock clk all_inputs()
set_output_delay -max 0.0 -clock clk all_outputs()
set_output_delay -min 0.0 -clock clk all_outputs()
set_load 1 all_outputs()
/*-----  (4)  Compile      ------*/
uniquify
set_structure true -timing true
compile -map_effort low -boundary_optimization
/*-----  (5)  Report results      ------*/
write -f verilog -output FIR.gate.v
report_timing -max  1 > FIR.data
report_area >> FIR.data
report_power >> FIR.data
```

Example of Time-Budgeting

Timing Constraints For Block B:

set_input_delay -max 2.0 -clock clk all_inputs()
set_output_delay -max 3.0 -clock clk all_outputs()

預計 A 用掉 1ns
留 3ns 給 C



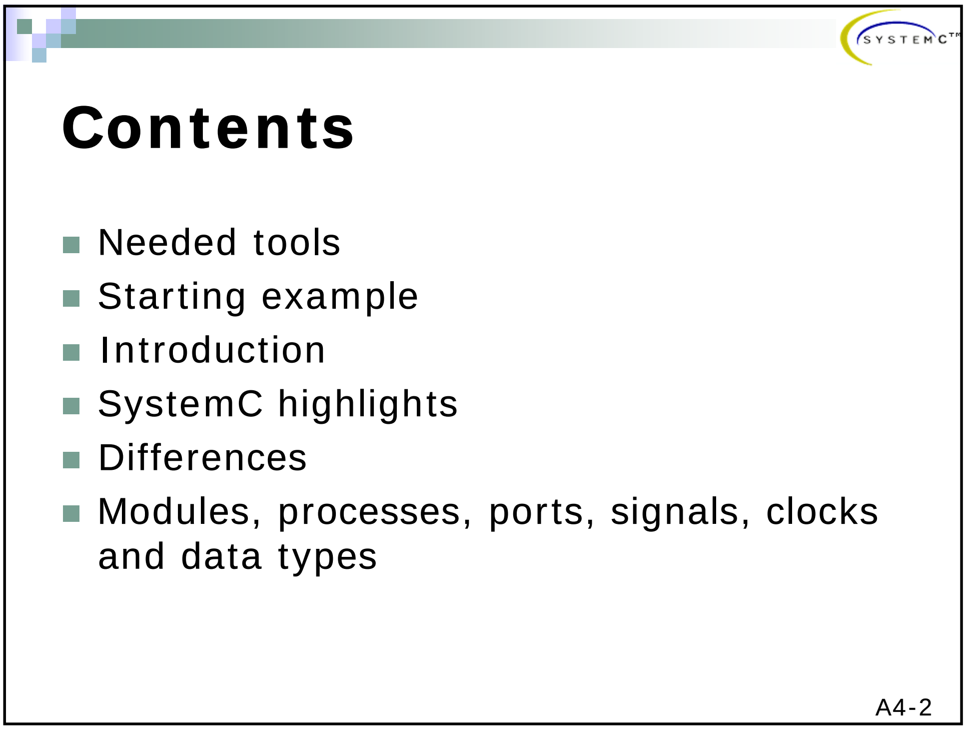
Synthesis Results

architecture criteria	Direct Form	Transposed Form
Area (gate-count)	1674 (1573, 101)	1212 (1110, 101)
Timing (ns)	12.7 ns	10.37 ns
Power (mW)	35.03 nW	37.26 mW

(1) **Gate count** is in terms of equivalent **2-input NAND gate**
(2) **Timing** is based on **static timing analysis**
(3) **Power dissipation** is only a very **rough estimation**

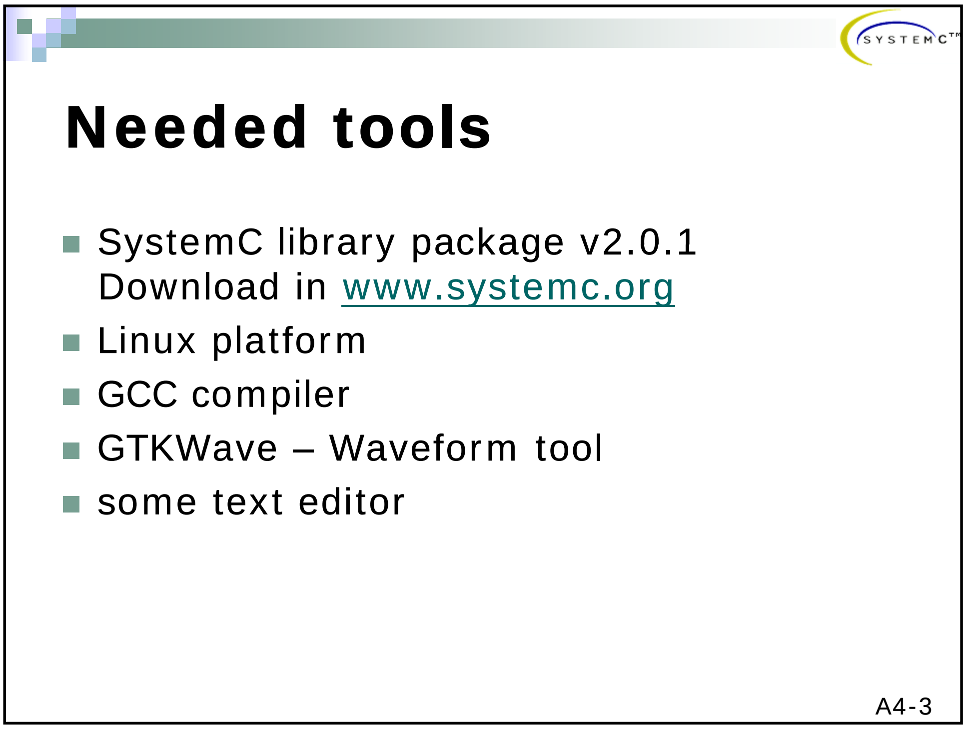
SystemC Tutorial

Author: Silvio Veloso
svfn@cin.ufpe.br



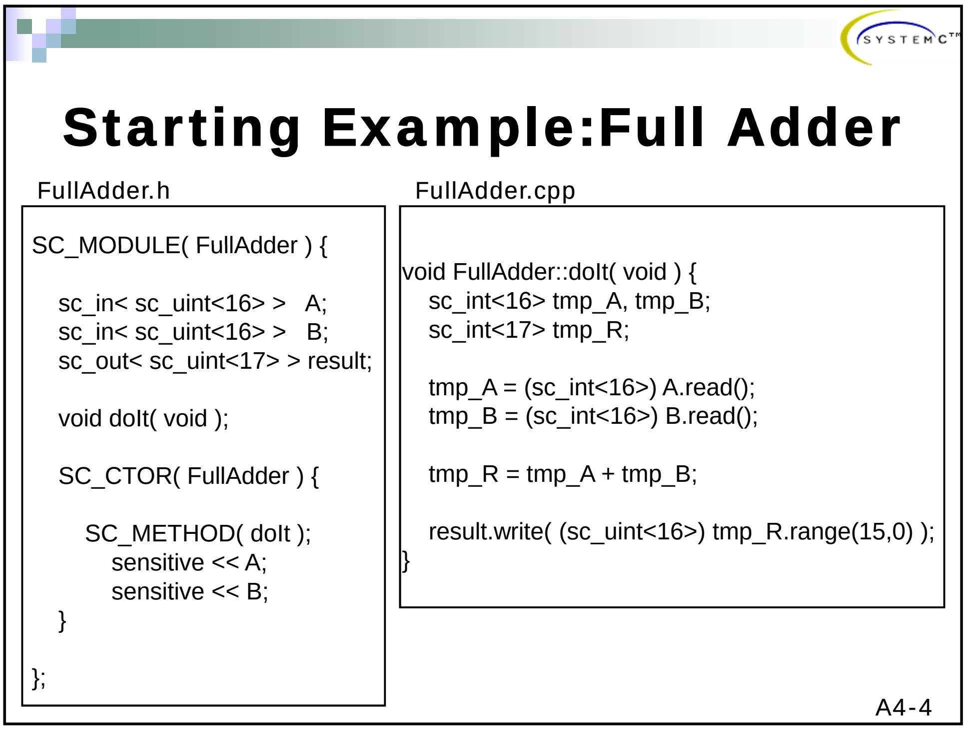
Contents

- Needed tools
- Starting example
- Introduction
- SystemC highlights
- Differences
- Modules, processes, ports, signals, clocks and data types



Needed tools

- SystemC library package v2.0.1
  Download in www.systemc.org
- Linux platform
- GCC compiler
- GTKWave – Waveform tool
- some text editor



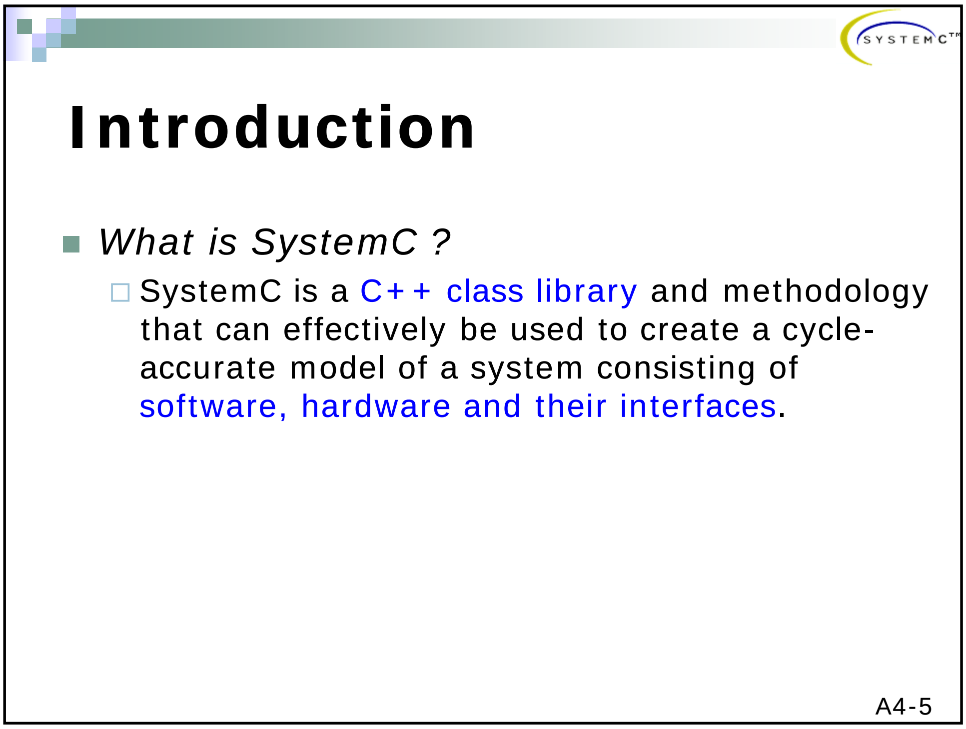
Starting Example:Full Adder

FullAdder.h

```
SC_MODULE( FullAdder ) {

   sc_in< sc_uint<16> >   A;
   sc_in< sc_uint<16> >   B;
   sc_out< sc_uint<17> > result;

   void doIt( void );

   SC_CTOR( FullAdder ) {

      SC_METHOD( doIt );
         sensitive << A;
         sensitive << B;
   }

};
```

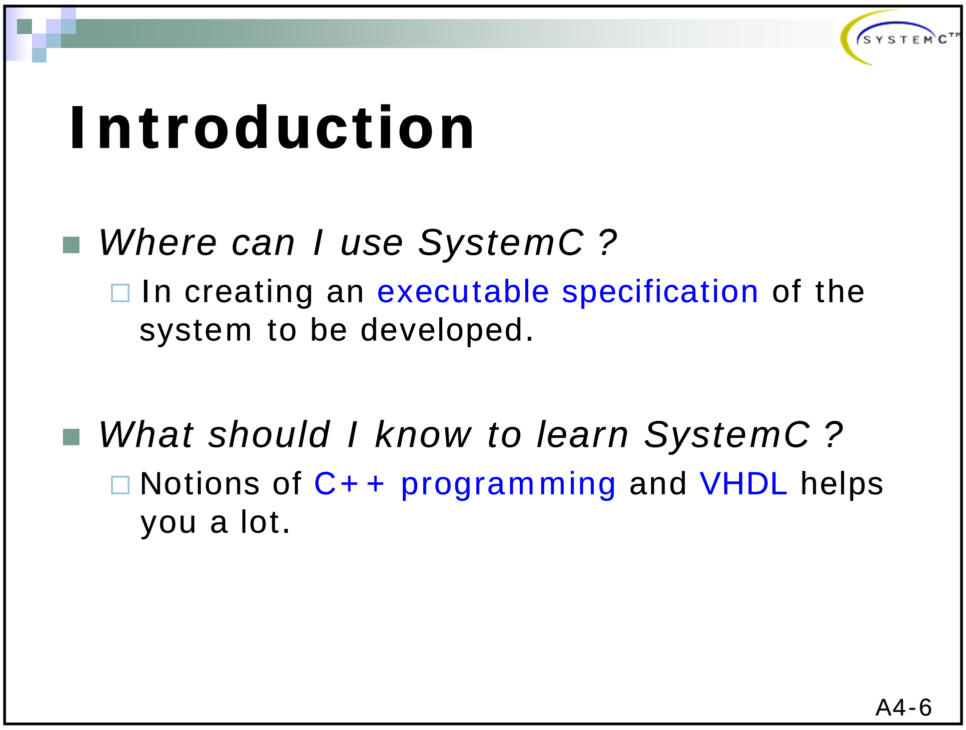
FullAdder.cpp

```
void FullAdder::doIt( void ) {
   sc_int<16> tmp_A, tmp_B;
   sc_int<17> tmp_R;

   tmp_A = (sc_int<16>) A.read();
   tmp_B = (sc_int<16>) B.read();

   tmp_R = tmp_A + tmp_B;

   result.write( (sc_uint<16>) tmp_R.range(15,0) );
}
```



Introduction

- *What is SystemC ?*
  - ☐ SystemC is a C++ class library and methodology that can effectively be used to create a cycle-accurate model of a system consisting of software, hardware and their interfaces.



Introduction

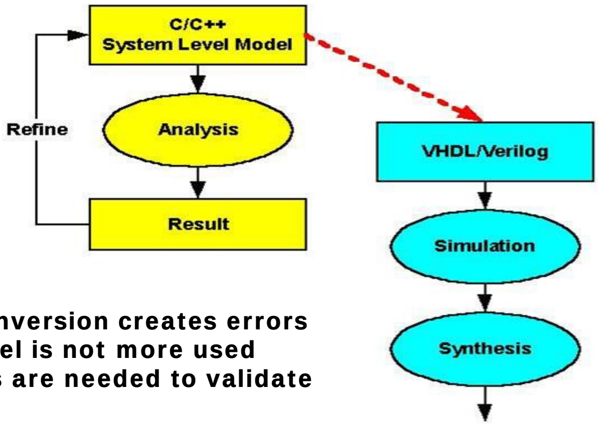
- *Where can I use SystemC ?*
  - ☐ In creating an executable specification of the system to be developed.
- *What should I know to learn SystemC ?*
  - ☐ Notions of C++ programming and VHDL helps you a lot.

SystemC highlights

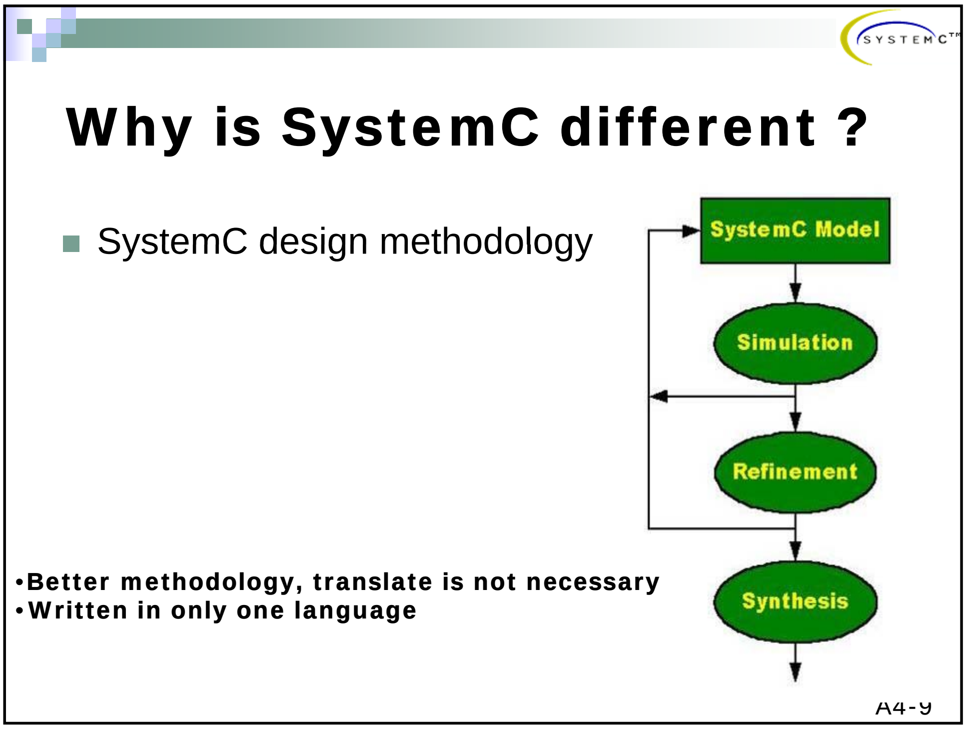
- Supports hardware and software co-design
- Developing an executable specification avoids inconsistency and errors
- Avoids wrong interpretation of the specification
- SystemC has a rich set of data types for you to model your systems
- It allows multiple abstraction levels, from high level design down to cycle-accurate RTL level

Why is SystemC different ?

- Current design methodology



- Manual conversion creates errors
- The C model is not more used
- Many tests are needed to validate



Why is SystemC different ?

- SystemC design methodology

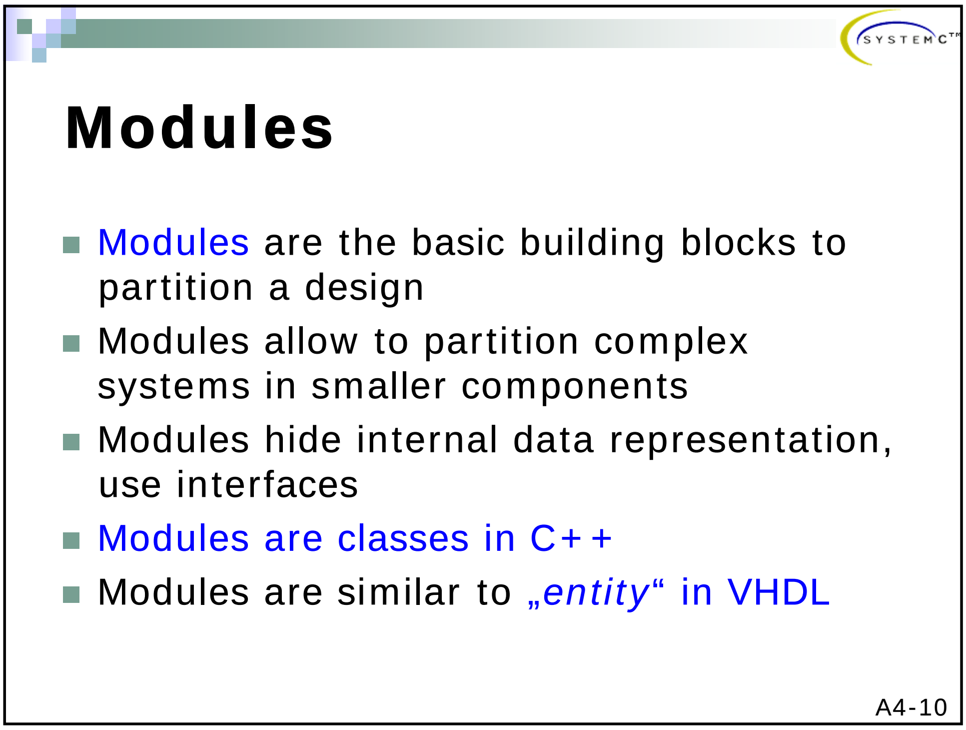
SystemC Model

Simulation

Refinement

Synthesis

•Better methodology, translate is not necessary
•Written in only one language



Modules

- Modules are the basic building blocks to partition a design
- Modules allow to partition complex systems in smaller components
- Modules hide internal data representation, use interfaces
- Modules are classes in C++
- Modules are similar to „*entity*“ in VHDL

Modules

```
SC_MODULE(module_name)
{
    // Ports declaration
    // Signals declaration
    // Module constructor : SC_CTOR
    // Process constructors and sensibility list
    //          SC_METHOD
    // Sub-Modules creation and port mappings
    // Signals initialization
}
```

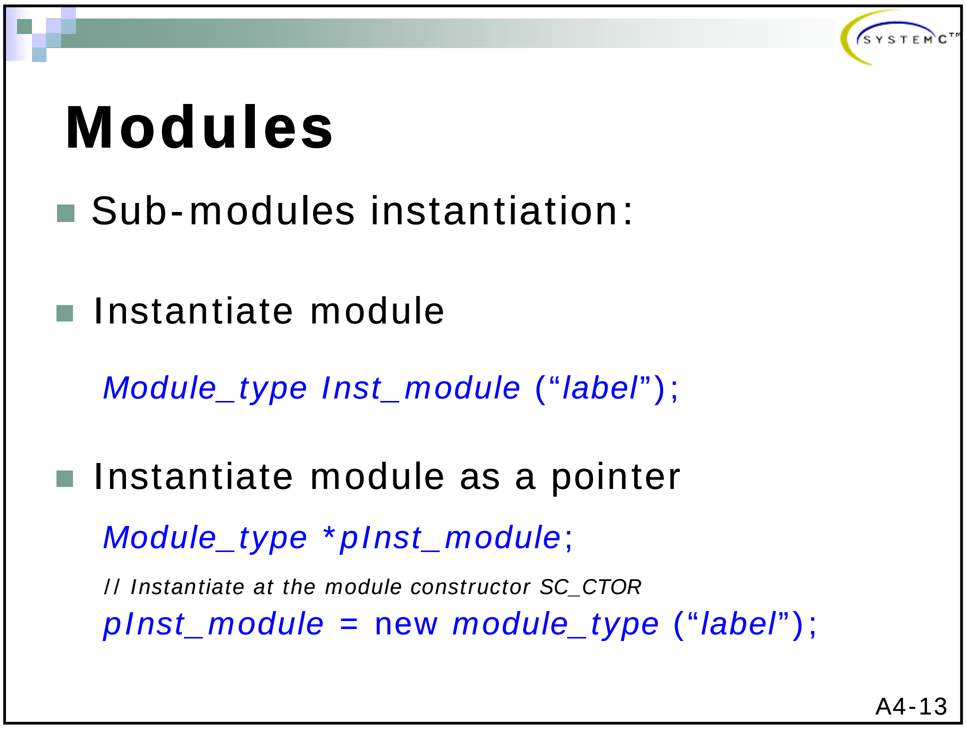
They can contain ports, signals, local data, other modules, processes and constructors.

Modules

- Module constructor
- Similar to „*architecture*“ in VHDL

Example: Full Adder constructor

```
SC_CTOR( FullAdder ) {

        SC_METHOD( doIt );
            sensitive << A;
            sensitive << B;
    }
```



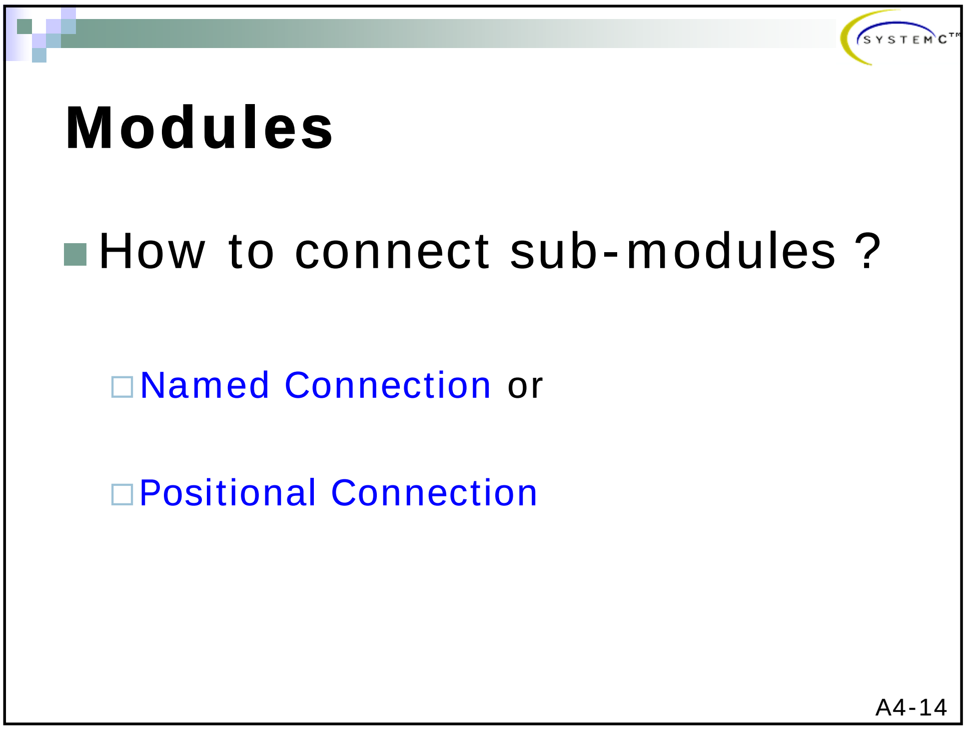
Modules

- Sub-modules instantiation:
- Instantiate module

```
Module_type Inst_module (“label”);
```

- Instantiate module as a pointer

```
Module_type *pInst_module;
// Instantiate at the module constructor SC_CTOR
pInst_module = new module_type (“label”);
```



Modules

- How to connect sub-modules ?
  - ☐ Named Connection or
  - ☐ Positional Connection

Modules

- Named Connection

```
Inst_module.a(s);
Inst_module.b(c);
Inst_module.q(q);

pInst_module -> a(s);
pInst_module -> b(c);
pInst_module -> q(q);
```

Modules

- Positional Connection

```
Inst_module << s << c << q;
(*pInst_module)(s,c,q);
```

Modules

- Internal Data Storage
- Local variables: can not be used to connect ports
- Allowed data types
  - ☐ C++ types
  - ☐ SystemC types
  - ☐ User defined types

Modules

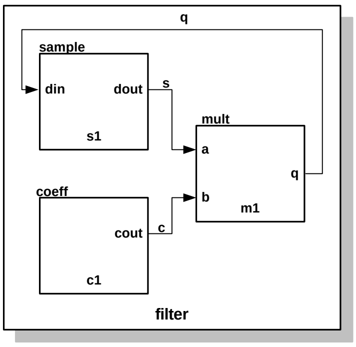
Objects of
template class *sc_in*
(8-bit unsigned integer input port)

- Example: Mux 2:1

```
SC_MODULE( Mux21 ) {

    sc_in< sc_uint<8> >   in1;
    sc_in< sc_uint<8> >   in2;
    sc_in< bool >       selection;
    sc_out< sc_uint<8> >  out;

    void doIt( void );

    SC_CTOR( Mux21 ) {

        SC_METHOD( doIt );
            sensitive << selection;
            sensitive << in1;
            sensitive << in2;

    }

};
```

Modules

- Example:



```
SC_MODULE(filter) {
        // Sub-modules : “components”
        sample *s1;
        coeff *c1;
        mult *m1;

        sc_signal<sc_uint 32> q, s, c; // Signals

        // Constructor : “architecture”
        SC_CTOR(filter) {

       // Sub-modules instantiation and mapping
                s1 = new sample (“s1”);
                s1->din(q); // named mapping
                s1->dout(s);

                c1 = new coeff(“c1”);
                c1->out(c); // named mapping

                m1 = new mult (“m1”);
                (*m1)(s, c, q); // Positional mapping

          }
}
```

Processes

- Processes are functions that are identified to the SystemC kernel. They are called if one signal of the sensitivity list changes its value.
- Processes implement the funcionality of modules
- Processes can be Methods, Threads and CThreads

Processes

- Methods

When activated, executes and returns

  - SC_METHOD(process_name)

- Threads

Can be suspended and reactivated

  - wait() -> suspends
  - one sensitivity list event -> activates
  - SC_THREAD(process_name)

- CThreads

Are activated in the clock pulse

  - SC_CTHREAD(process_name, clock value);

Processes

Type	SC_METHOD	SC_THREAD	SC_CTHREAD
Activates Exec.	Event in sensit. list	Event in sensit. List	Clock pulse
Suspends Execution	NO	YES	YES
Infinite Loop	NO	YES	YES
suspended/ reactivated by	N.D.	wait()	wait() wait_until()

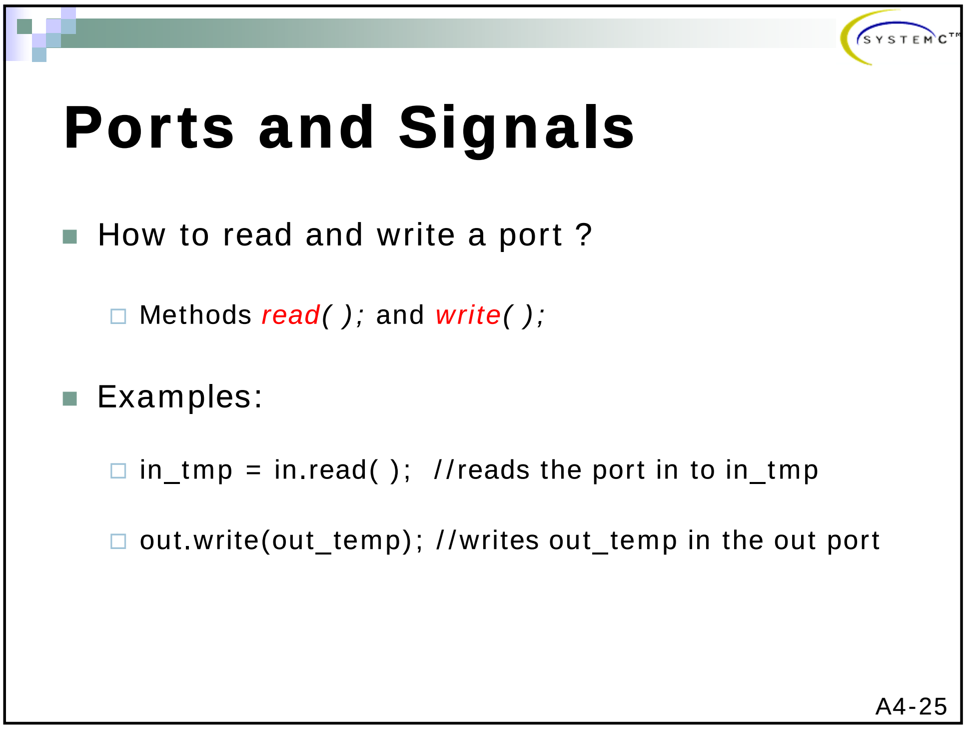
Constructor & Sensibility definition	SC_METHOD(*call_back*);
	sensitive(*signals*);
	sensitive_pos(*signals*);
	sensitive_neg(*signals*);
		SC_THREAD(*call_back*);
		sensitive(*signals*);
		sensitive_pos(*signals*);
		sensitive_neg(*signals*);
			SC_CTHREAD(*call_back*, *clock*.pos());
			SC_CTHREAD(*call_back*, *clock*.neg());

Ports and Signals

- Ports of a module are the external interfaces that pass information to and from a module
- In SystemC one port can be *IN*, *OUT* or *INOUT*
- Signals are used to connect module ports allowing modules to communicate
- Very similar to ports and signals in VHDL

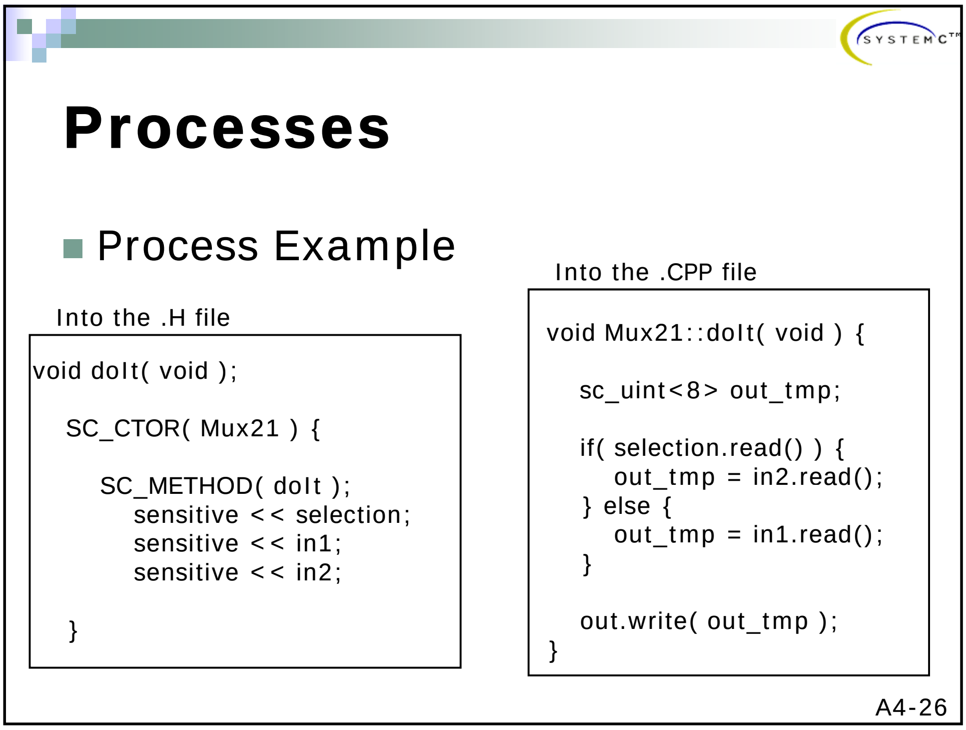
Ports and Signals

- Types of ports and signals:
  - All natives C/C++ types
  - All SystemC types
  - User defined types
- How to declare
  - IN : sc_in<***port_typ***>
  - OUT : sc_out<***port_type***>
  - Bi-Directional : sc_inout<***port_type***>



Ports and Signals

- How to read and write a port ?
  - Methods *read( );* and *write( );*
- Examples:
  - `in_tmp = in.read( );  //reads the port in to in_tmp`
  - `out.write(out_temp); //writes out_temp in the out port`



Processes

- Process Example

Into the .H file

```
void doIt( void );

   SC_CTOR( Mux21 ) {

      SC_METHOD( doIt );
         sensitive << selection;
         sensitive << in1;
         sensitive << in2;

   }
```

Into the .CPP file

```
void Mux21::doIt( void ) {

   sc_uint<8> out_tmp;

   if( selection.read() ) {
      out_tmp = in2.read();
   } else {
      out_tmp = in1.read();
   }

   out.write( out_tmp );
}
```

Clocks

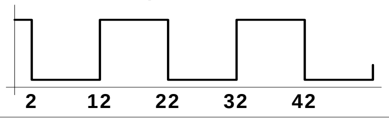
- Special object
- How to create ?

  sc_clock ***clock_name*** (

  "***clock_label***", ***period***, ***duty_ratio***, ***offset***, ***initial_value***);
- Clock connection

  f1.clk(clk_signal); //where f1 is a module
- Clock example:

sc_clock clock1 ("clock1", 20, 0.5, 2, true);



Data Types

- SystemC supports:
  - ☐ C/C++ native types
  - ☐ SystemC types
- SystemC types
  - ☐ Types for systems modelling
  - ☐ 2 values **('0','1')**
  - ☐ 4 values **('0','1','Z','X')**
  - ☐ Arbitrary size integer (Signed/Unsigned)
  - ☐ Fixed point types

SystemC types

Type	Description
sc_logic	Simple bit with 4 values(0/1/X/Z)
sc_int	Signed Integer from 1-64 bits
sc_uint	Unsigned Integer from 1-64 bits
sc_bigint	Arbitrary size signed integer
sc_biguint	Arbitrary size unsigned integer
sc_bv	Arbitrary size 2-values vector
sc_lv	Arbitrary size 4-values vector
sc_fixed	templated signed fixed point
sc_ufixed	templated unsigned fixed point
sc_fix	untemplated signed fixed point
sc_ufix	untemplated unsigned fixed point

SystemC types

- ***Simple bit type***
- Assignment similar to *char*
  - my_bit = ‘1’;
- Declaration
  - bool my_bit;

Operators

Bitwise	& (and)	| (or)	^ (xor)	~ (not)
Assignment	=	&=	|=	^=
Equality	==	!=		

SystemC types

Operators of fixed precision types

Bitwise	~	&	|	^	>>	<<			
Arithmetics	+	-	*	/	%				
Assignement	=	+=	-=	*=	/=	%=	&=	|=	^=
Equality	==	!=							
Relational	<	<=	>	>=					
Auto-Inc/Dec	++	--							
Bit selection	[*x*]		*ex)* mybit = myint[7]						
Part select	range()		*ex)* myrange = myint.range(7,4)						
Concatenation	(,)		*ex)* intc = (inta, intb);						

SystemC types

- Bit vector
  - sc_bv<***n***>
  - 2-value vector (0/1)
  - Not used in arithmetics operations
  - Faster simulation than *sc_lv*

- Logic Vector
  - sc_lv<***n***>
  - Vector to the *sc_logic* type

- Assignment operator (“=“)
  - my_vector = “XZ01”
  - Conversion between vector and integer (int or uint)
  - Assignment between *sc_bv* and *sc_lv*
  - Additional Operators

Reduction	and_reduction()	or_reduction()	xor_reduction()
Conversion	to_string()		

SystemC types

- Examples:
  - sc_bit y, sc_bv<8> x;
  - y = x[6];

  - sc_bv<16> x, sc_bv<8> y;
  - y = x.range(0,7);

  - sc_bv<64> databus, sc_logic result;
  - result = databus.or_reduce();

  - sc_lv<32> bus2;
  - cout << “bus = “ << bus2.to_string();

Ending Example:Full Adder

FullAdder.h

```
SC_MODULE( FullAdder ) {

    sc_in< sc_uint<16> >   A;
    sc_in< sc_uint<16> >   B;
    sc_out< sc_uint<17> > result;

    void doIt( void );

    SC_CTOR( FullAdder ) {

        SC_METHOD( doIt );
            sensitive << A;
            sensitive << B;
    }

};
```

FullAdder.cpp

```
void FullAdder::doIt( void ) {
    sc_int<16> tmp_A, tmp_B;
    sc_int<17> tmp_R;

    tmp_A = (sc_int<16>) A.read();
    tmp_B = (sc_int<16>) B.read();

    tmp_R = tmp_A + tmp_B;

    result.write( (sc_uint<16>) tmp_R.range(15,0) );
}
```

Appendix:
on Integrated Fan-Out Wafer-Level Packaging (InFO-WLP)

Edited by:

Shi-Yu Huang

National Tsing Hua University, Taiwan

References

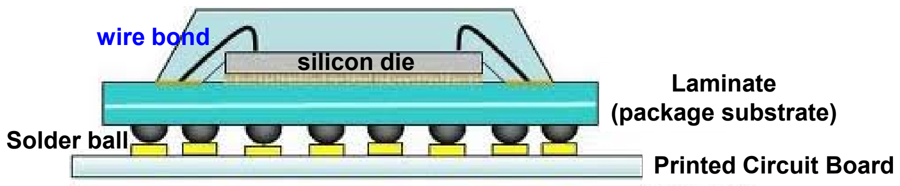
1. ***Web site at STATSChipPAC http://www.statschippac.com/***
2. ***Web site at "SEMI Networking Day: Packaging – Key for System Integration," http://www.semi.org/eu/node/8481***
3. ***T. Meyer et al., "System Integration with eWLB", Electronic System-Integration Technology Conference (ESITC), 2010.***
4. ***Seung Wook et al., "Fanout Flipchip eWLB (embedded Wafer Level Ball Grid Array) Technology as 2.5D Packaging Solutions" Electronic Components and Technology Conf., (ECTC) 2013.***
5. ***Christianto C. Liu et al., "High-Performance Integrated Fan-Out Wafer Level Packaging (InFO-WLP): Technology and System Integration", Int'l Electron Device Meetings (IEDM), 2012.***
6. ***Cedric Durand et al., "High Performance RF Inductors Integrated in Advanced Fan-Out Wafer Level Packaging Technology", Silicon Monolithic Integrated Circuits in RF Systems (SiRF), 2012.***

Outline

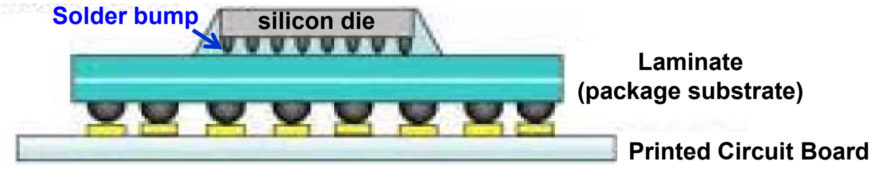
- ◆ What is FO-WLP?
  - Evolution of packaging technology
  - Processing steps
  - Advantages
- ◆ Example of application
  - An RF test chip validated by TSMC
- ◆ Discussion

Classical Single-Die Package

Wire-Bond Package

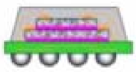


Flip-Chip Package

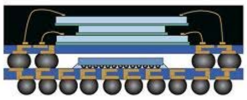


Classical SiP (System-In-Package)

(multi dies or components in a single package)

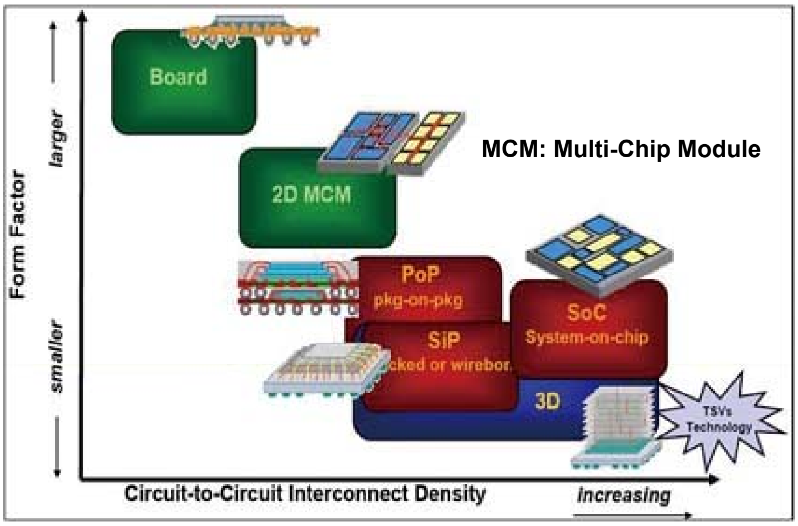
Structure		Example
Side-by-Side Structure		 Wire Bonded  Flip-Chip
Stacked Structure	Package On Package Stacking	 Wire Bonded  Wire Bonded & Flip Chip  PoP Flip-Chip Type
	Die Stacking	 Through Silicon Via

Easier to test (O)
Lengthy interconnects (X)
Substrate cost (X)



Another PoP example

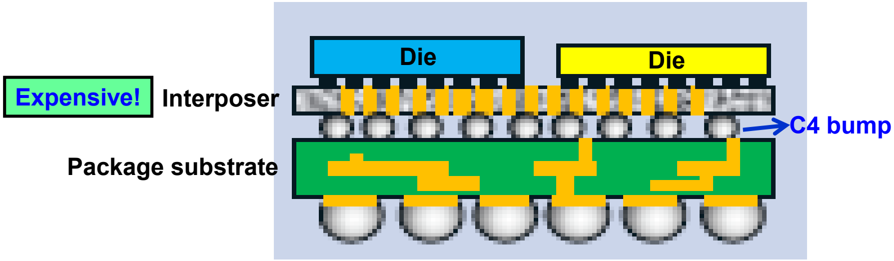
Evolution of Packaging Technologies

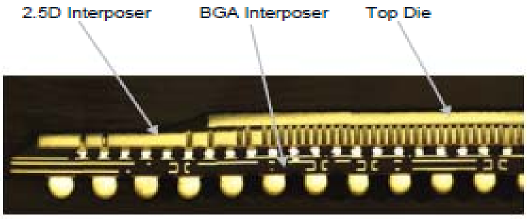


Source: Eric Mounier, Yole Development, Lyon, France, Global SMT & Packaging July 2007.

Interposer-Based 2.5D-Ics

CoWoS (Chip-on-Wafer-on-Substrate) – TSMC 2012





2.5D silicon Interposer, Cross Section
(Courtesy of ASE)

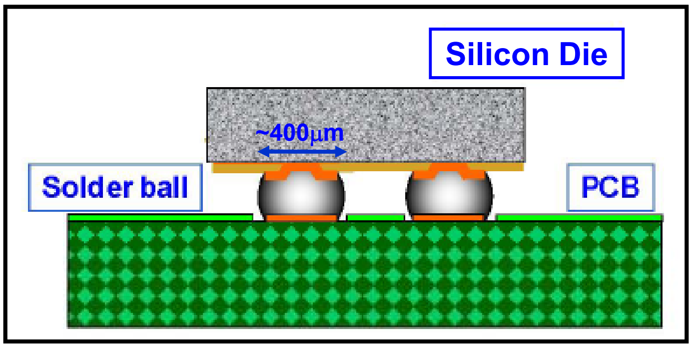
Quest:
Can we have a low-cost 2.5D IC without interposer?

➔ FO-WLP might be it (with adequate performance)

Typical Wafer-Level Package (Fan-In WLP)

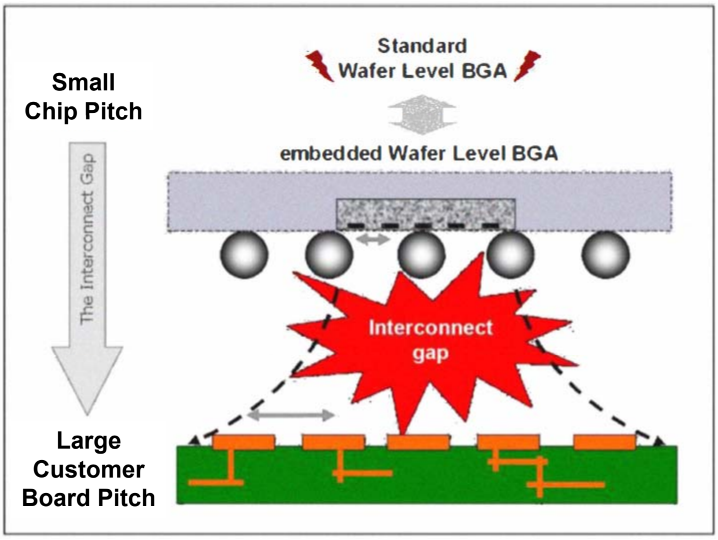
Three layers: (1) Silicon die, (2) Solder Ball, (3) PCB

Only one-layer of solder bump/ball → No Laminate, but larger IO pitch



P.S. One typical dimension of an IO pad is roughly 60µm x 60µm
P.S. ITRS roadmap: The IO pad pitch will continue to shrink…

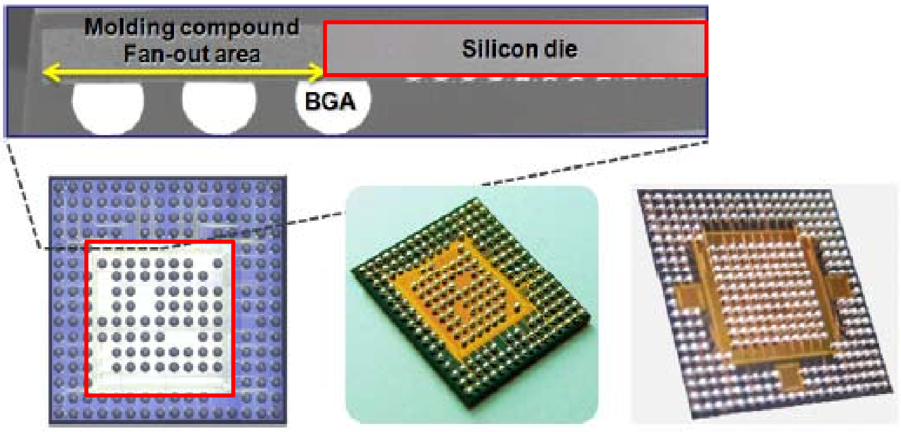
Interconnect Pitch Gap Problem



Source: Infineon

InFO-WLP
(Integrated Fan-Out Wafer-Level Packaging)

Fanout: the extra area outside the die areas
Wafer-Level Package: Interconnects and ball-dropping on a re-constituted wafer

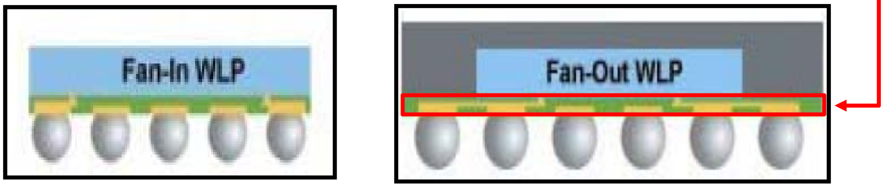


BGA: Ball-Grid Array

400 pins

RDL (Re-Distribution Layer)

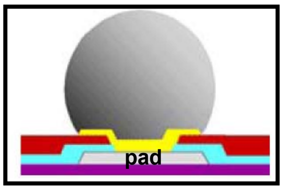
Exploit higher Flexibility provided by RDL (Re-Distribution Layer) between bare dies and solder balls
➔ (1) package foot-print > chip foot-print, (2) multi-chip package



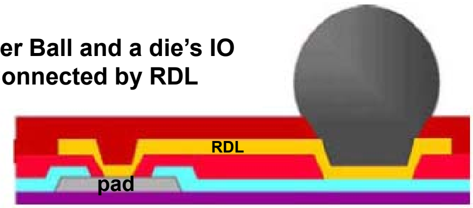
RDL: used to route the signal path from the die's IOs to desired bump locations

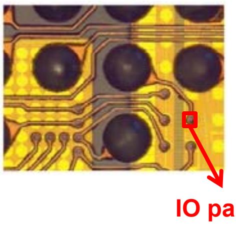
Detailed View of a Solder Joint

Solder ball sitting on top a die's IO



Solder Ball and a die's IO
Connected by RDL





Technology Steps for InFO-WLP

1: Wafer reconstruction (including wafer molding)

↓

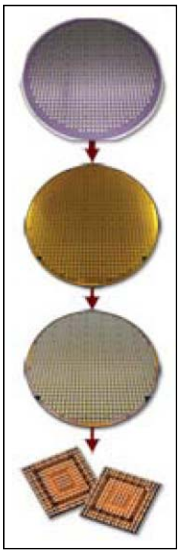
2: Re-Distribution

↓

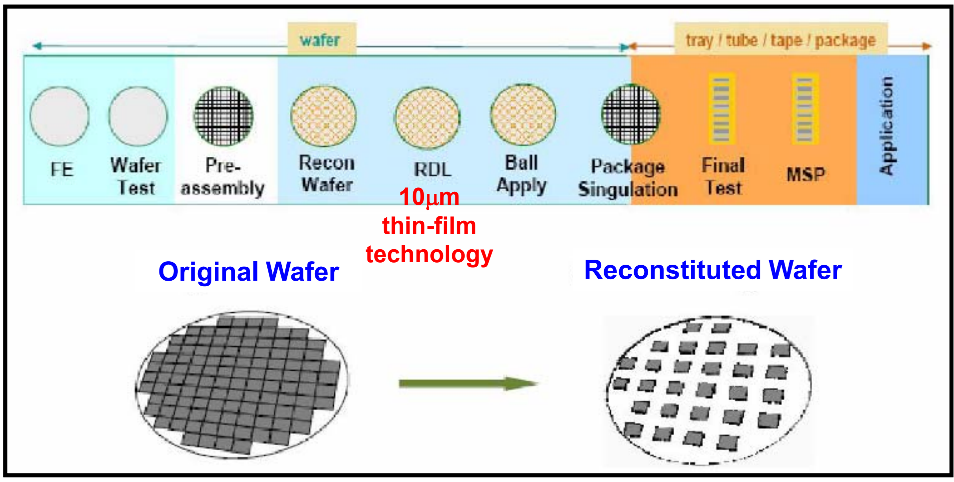
3. Ball Mount and Singulation

↓

4. Test, Mark, Scan, Pack



Another View of Processing Steps



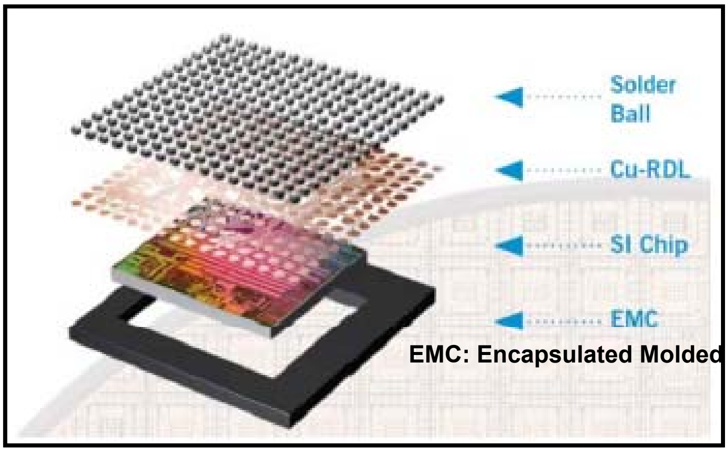
10μm thin-film technology

Original Wafer

Reconstituted Wafer

"good tested" dies of a silicon wafer are placed face-down onto a carrier with an adhesive tape.

Layers of a FO-WLP Package



EMC: Encapsulated Molded Compound

Source: STMicroelectronics, SiRF 2012.

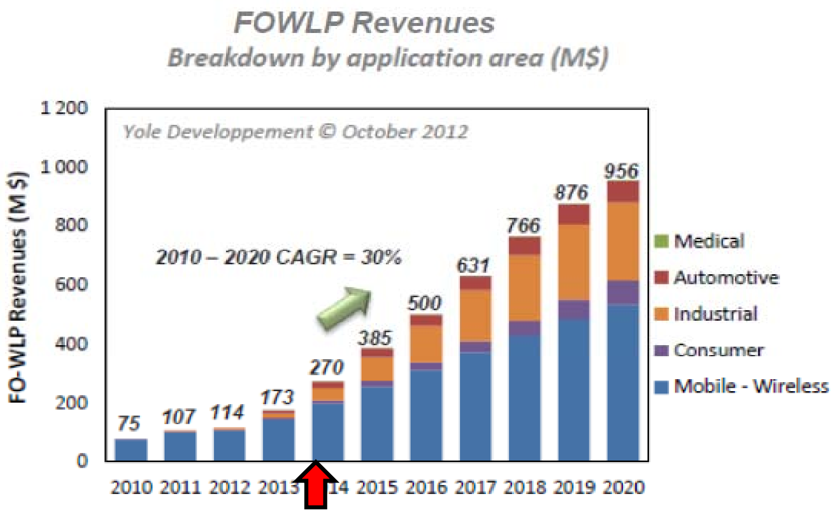
A Specific WLP Technology

- Embedded Wafer-Level Ball-Grid Array (eWLB)

- ◆ ***eWLB was a technology pioneered by Nanium***
- ◆ ***Siemens → Infineon (1999) → Qimonda (2006) → Nanium (2010)***
- ◆ ***With proven high-volume manufacturing capability, Nanium have shipped more than 300 million eWLB components, achieving industry-level yields and full JEDEC quality/reliability compliance.***

MOTIVATION:
to address the growing mismatch in interconnect gap, higher levels of integration, improved electrical performance and shorter vertical interconnects.

Business Outlook of FO-WLP



We are here!➔ 4.5X increase till 2020

Based on Yole Development, due to ramp-up of fabless wireless IC vendors

Features of eWLB

- Flexibility to integrate die from diverse processes, manufacturing sources & silicon wafer nodes for increased functionality
- 2D solutions in single & multi-die configurations, down to 0.4mm
- MCP versions with flip chip & IPD (integrated passive devices) integration capability
- 2.5D & 3D options offer lower overall cost than TSV integration with increased process simplicity
- Industry’s thinnest 3D PoP solutions (ultra thin z-height of 0.3mm with stacked thickness down to 0.8mm height)
- Ultra fine ball pitch (down to 0.3mm) & maximum I/O density
- Thin film processing enables very fine lines for X,Y routing (line-width/line-space ratios less than 10um/10um), very fine via pitches and thin dielectrics
- Bumpless thin film interconnection offers lowest cost structure over competing manufacturing approaches

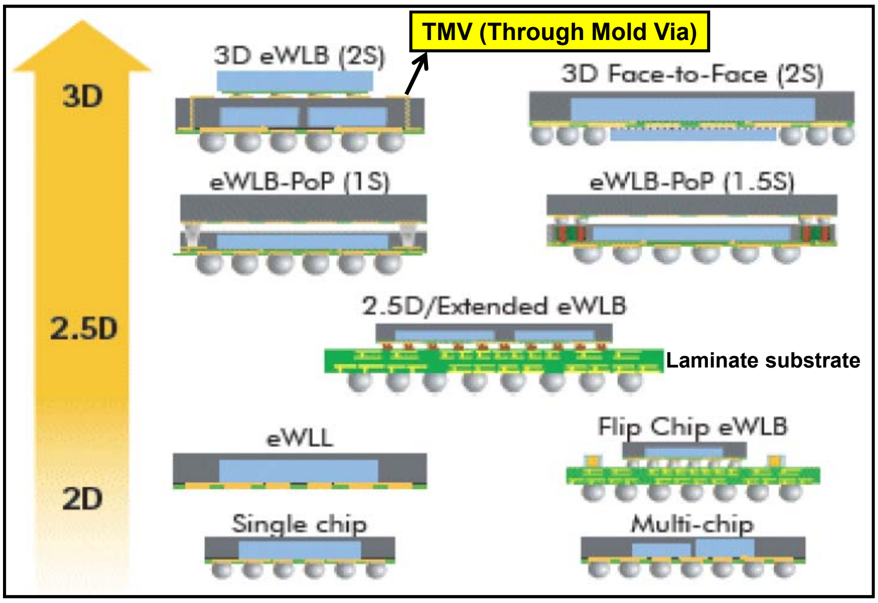
Source: web site of STATS ChipPAC

Features of eWLB - Continued

- Elimination of substrate results in a thinner package with lower warpage, simplifying supply chain & reducing costs
- Cost effective HVM batch processing (includes wafer level test)
- Advanced dielectric materials for reliable, power-efficient solutions
- Strong electrical performance (capable to beyond 60GHz)
- Effective heat dissipation supports strong thermal performance
- KGD helps achieve strong yields (99.9%)
- Cu/low-k (ELK) compatible packaging technology
- Green packaging (Pb-free and Halogen-free)

Source: web site of STATS ChipPAC

InFO-WLP for 2.5D and 3D Integration

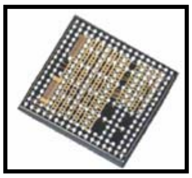


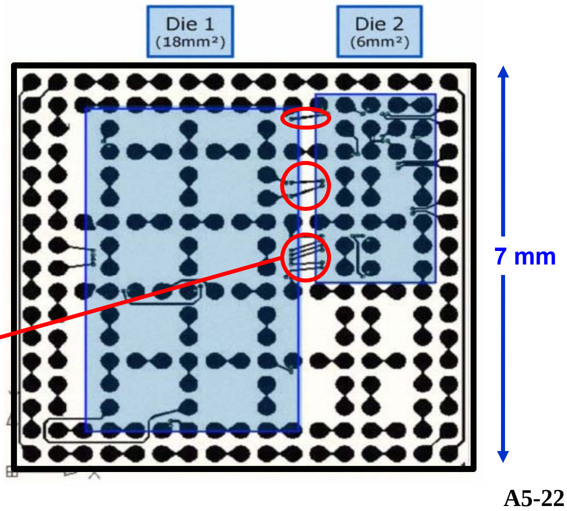
TMV (Through Mold Via)

Laminate substrate

Multi-Chip Test Vehicle

Die-to-Die distance is 300 μm in this case





Die-to-die interconnects

TMV (Through Mold Via in z-Direction)

- also called TEV (Through Encapsulant Via)

(Via-Before-Molding)

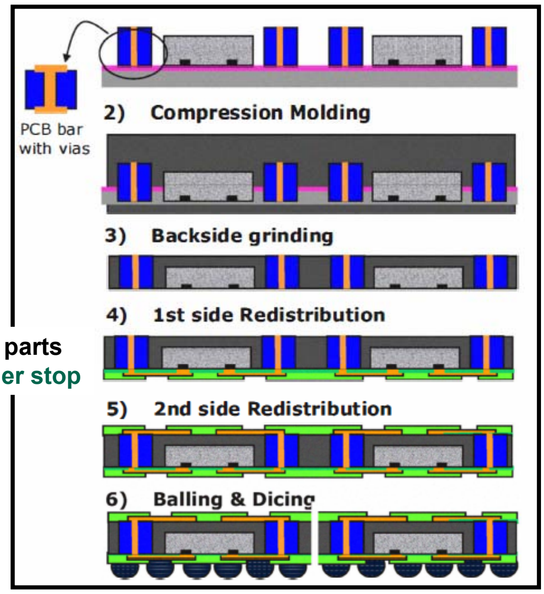
The placement of pre-fabricated via bars prior to the molding of the Reconstituted Wafer.

(Via-After-Molding)

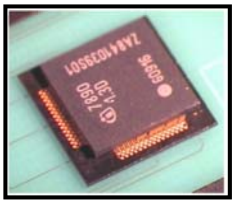
laser drilling and copper filling of vias in the mold compound.

---

ePOP Process Flow with Via Bars

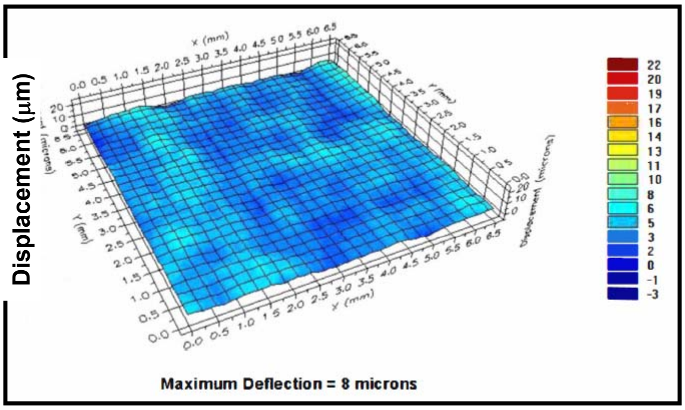


Green parts are solder stop



Defects

Potential manufacturing imperfections:
(1) Die shift (due to imprecision of pick-and-place equipment)
(2) Wafer warpage



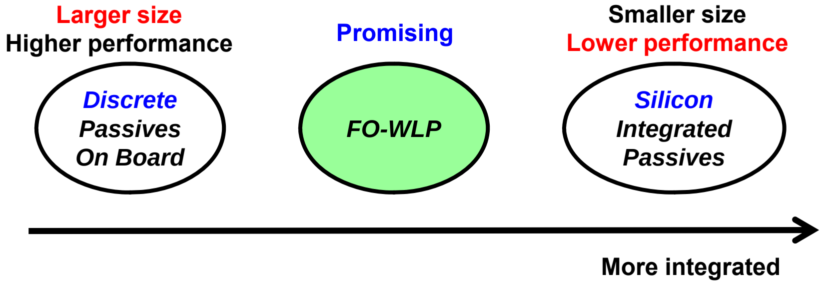
Maximum Deflection = 8 microns

Outline

- ◆ What is FO-WLP?
- ◆ **Example of application**
  - RF test vehicles
- ◆ Discussion

Integration of Passive Components

Why passive components matter:
(1) Wireless products are everywhere
(2) Passives (in particularly inductors) are key in RF circuits
  - Low-Noise Amplifiers (LNA), Voltage Controlled Oscillators (VCO),
  - Power Amplifiers (PA), filters, impedance matching networks…



Larger size
Higher performance

Discrete
Passives
On Board

Promising

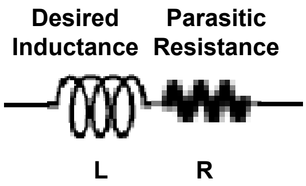
FO-WLP

Smaller size
Lower performance

Silicon
Integrated
Passives

More integrated

Q Factor of Inductance



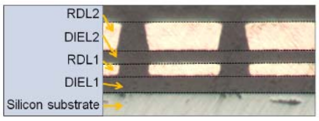
$$\boxed{Q = \frac{wL}{R}}$$

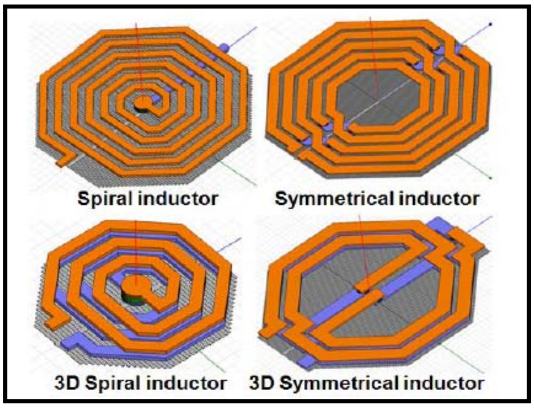
Smaller R → Higher Inductor's Q Factor
(Less Energy Loss in the Tuned Circuit)

Typical Q-factor range:
(1) Around 10 for CMOS inductors
(2) Around 25 ~ 35 for eWLB inductors

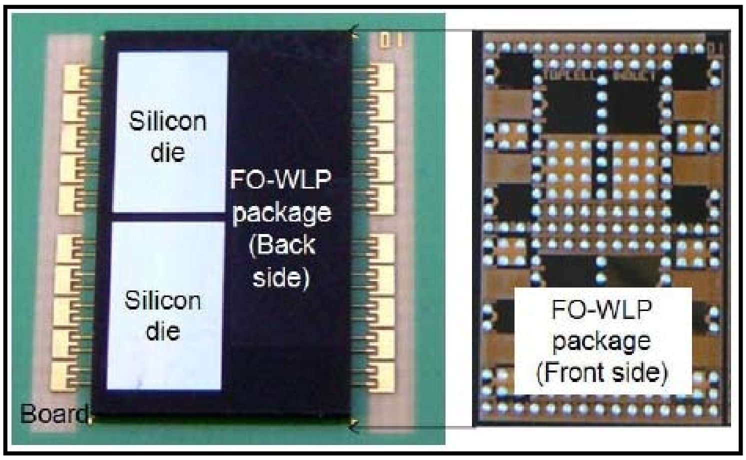
2-RDL Based Inductor in FO-WLP

Minimum RDL width and space: 10μm



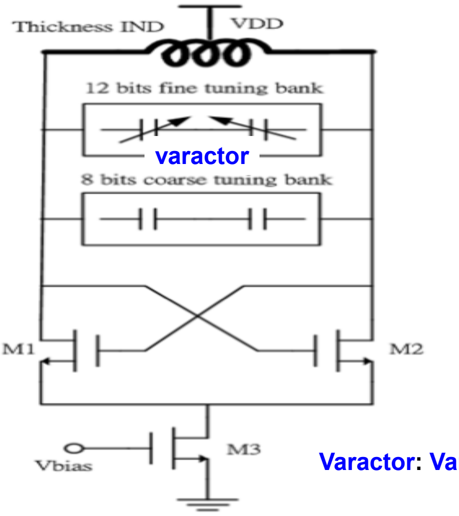


FO-WLP Package on Board



20-Bit DCO (Digitally Controlled Oscillator) - using LC Tank Oscillator

Schematic



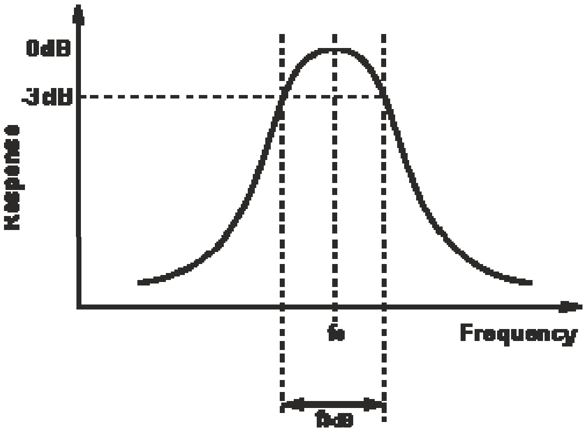
Resonant freq.

$$f_r = \frac{1}{2\pi\sqrt{LC}}$$

Varactor: **Var**iable Cap**acitor**

Q Factor of a Band-Pass Filter

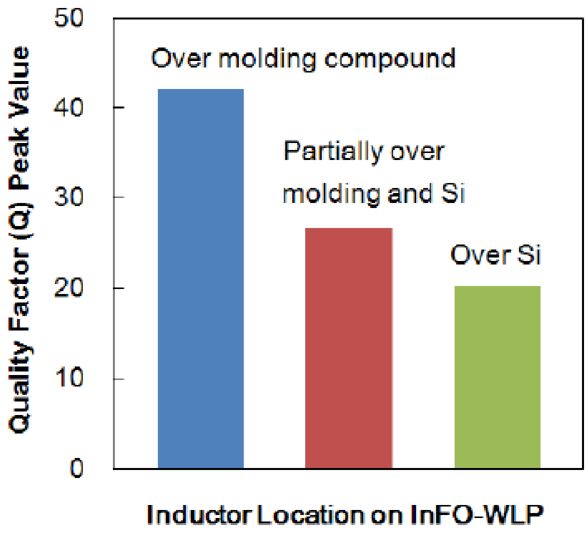
$$Q = \frac{F_0}{F_{3dB}}$$



DCO Performance Summary

	On-Chip	InFO-WLP (High Perf)	InFO-WLP (Low Power)
Center Freq (GHz)	10.6		
Technology (nm)	28		
Supply Voltage (V)	0.85		
Power (mW)	4.84	4.85	4.25
Phase Noise @ 1 MHz (dBc/Hz)	-100	-115	-107
FOM (dBc/Hz)	-174	-189	-181

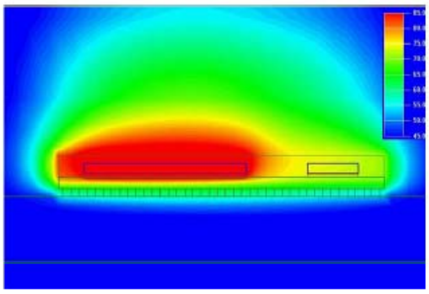
Peak Q Vale at Different Regions



Thermal Analysis

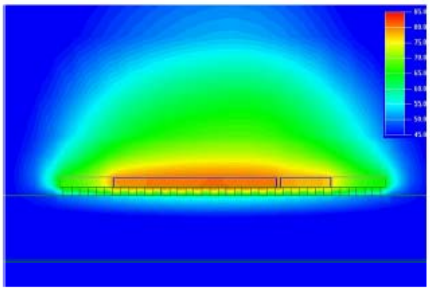
	FC-BGA/ MCM	InFO-WLP
Package Size (mm^2)	8x8	
Die Sizes (mm^2)	1 5x5 die, 2 2x1.25 dies	
Die Thickness (mm)	0.5	<0.3
Substrate Thickness (mm)	0.3	N/A
Ball Count	400	
Ball Diameter/Pitch (mm)	0.26/0.4	
Total Power (W)	2.0	
Ambient Temp (°C)	25	
Max Temp (°C)	90.5	81.5
Thermal Resistance (°C/W)	32.5	28.0

Thermal Map



Multi-Chip Module (MCM)

Thinner package
better heat dissipation



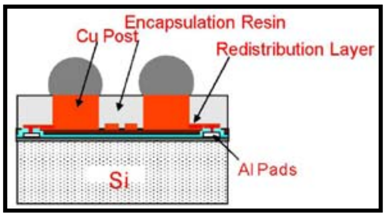
InFO-WLP

Concluding Remarks

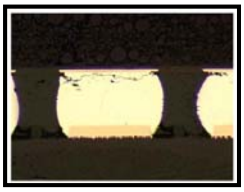
Wafer-Level Packaging (WLP)
has great potential for future multi-chip packaging.

Impacts on Testing:

(1) Yield might be highly dependent on KGD test
(2) Solder ball could be a weak point of thermal reliability
(3) Testing & Delay characterization for die-to-die interconnects



Ref: Coefficient of Thermal Expansion
(~2.6ppm/°C for silicon and 17ppm/°C for PCB)



Fatigue of solder ball after intensive Temperature Cycling