



数字集成电路设计自动化

姚恩义

2024年10月16日

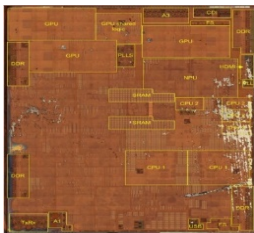
My Curriculum Vitae

- B. Eng – Harbin Institute of Technology 2007 – 2011
- PhD – Nanyang Technological University 2011 – 2016
School of Electrical and Electronic Engineering
- Post-Doc Nanyang Technological University 2015 – 2016
- Senior Engineer – Huawei Technologies 2016 – 2020
Computing Theory & Architecture
- Faculty – South China University of Technology since 2021
- Research: 1) Emerging computing architecture
2) Hardware Security
3) Hardware Acceleration for EDA
- Office: B1e303
- Email: yaoenyi@scut.edu.cn

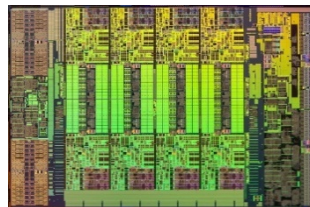
Outline

- Digital Design Flow from a IC design perspective
- Simulation
- Synthesis
- Placement
- Routing

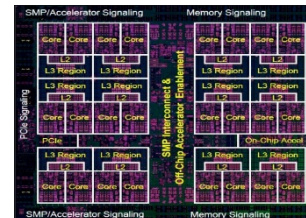
Era of Billion-Transistor Chips



Apple A11
~4B transistors (?)



Intel Haswell-EP Xeon E5
~7B transistors



IBM Power9
~8B transistors



Oracle SPARC M7
~10B transistors



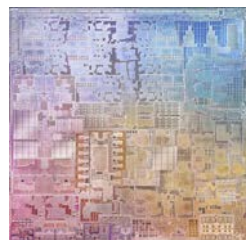
NVIDIA V100 Pascal
~21B transistors



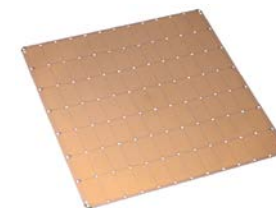
Intel/Altera Stratix 10
~30B transistors



Xilinx VU9P
~ 35B transistors



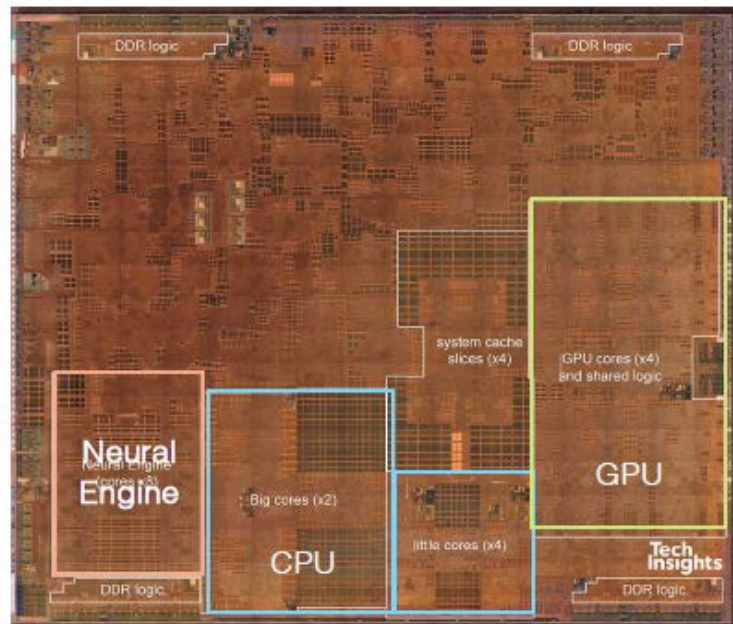
Apple M1
~ 57B transistors



Cerebras WSE-2
~ 2.6T transistors

Modern System-On-Chip (SoC)

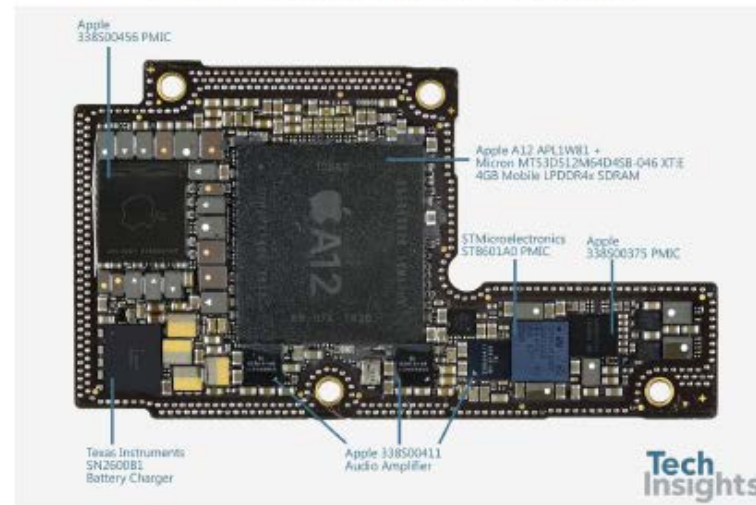
- Apple A12



TechInsights

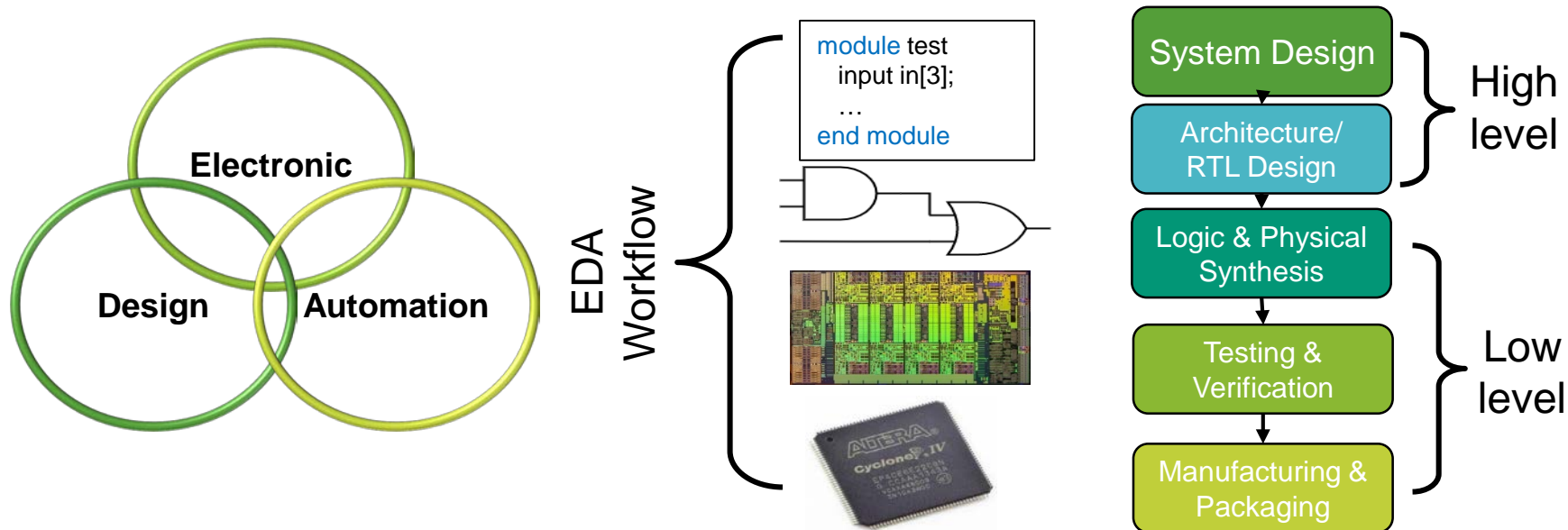
- 7nm CMOS
- Up to 2.49GHz

- 2x Large CPUs
- 4x Small CPUs
- GPUs
- Neural processing unit (NPU)
- Lots of memory
- DDR memory interfaces



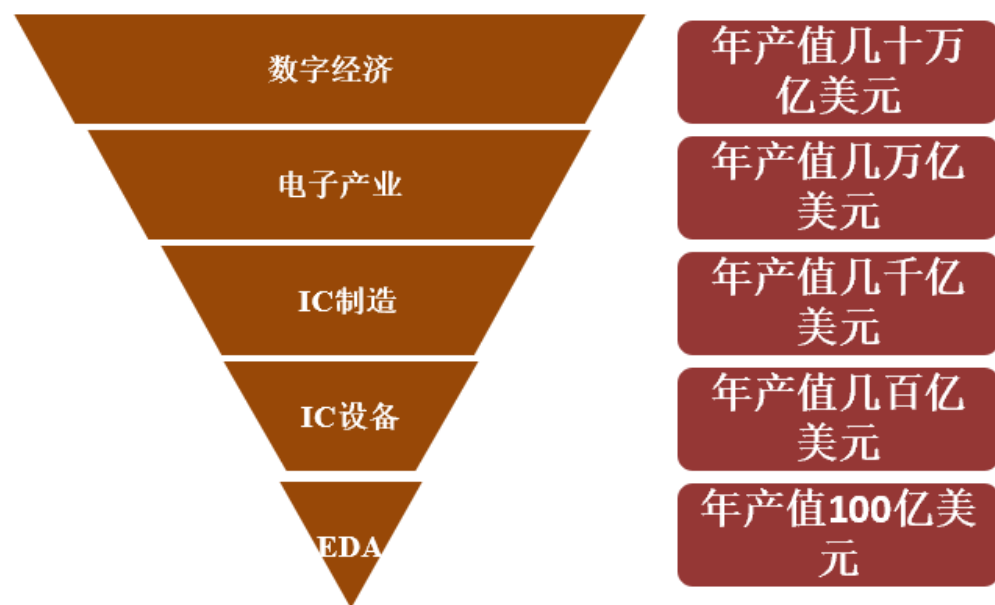
iPhone XS,
2018

Electronic Design Automation



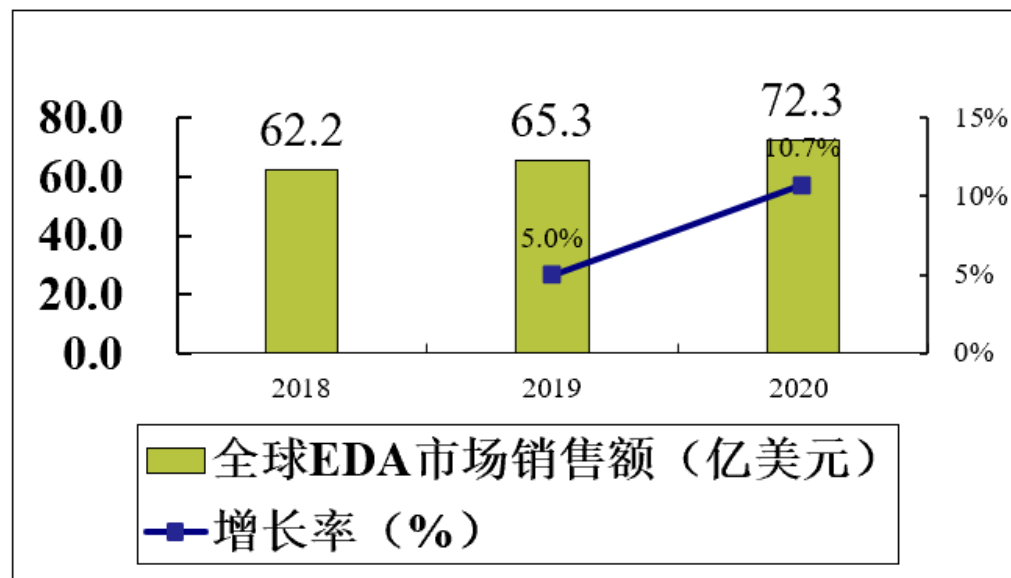
- **EDA:** a general methodology for refining a **high-level description down to a detailed physical implementation** for designs
 - integrated circuits (including system-on-chips),
 - and electronic systems
- **Modeling, synthesis, and verification** at every level of **abstraction**

全球EDA市场近期持续保持增长



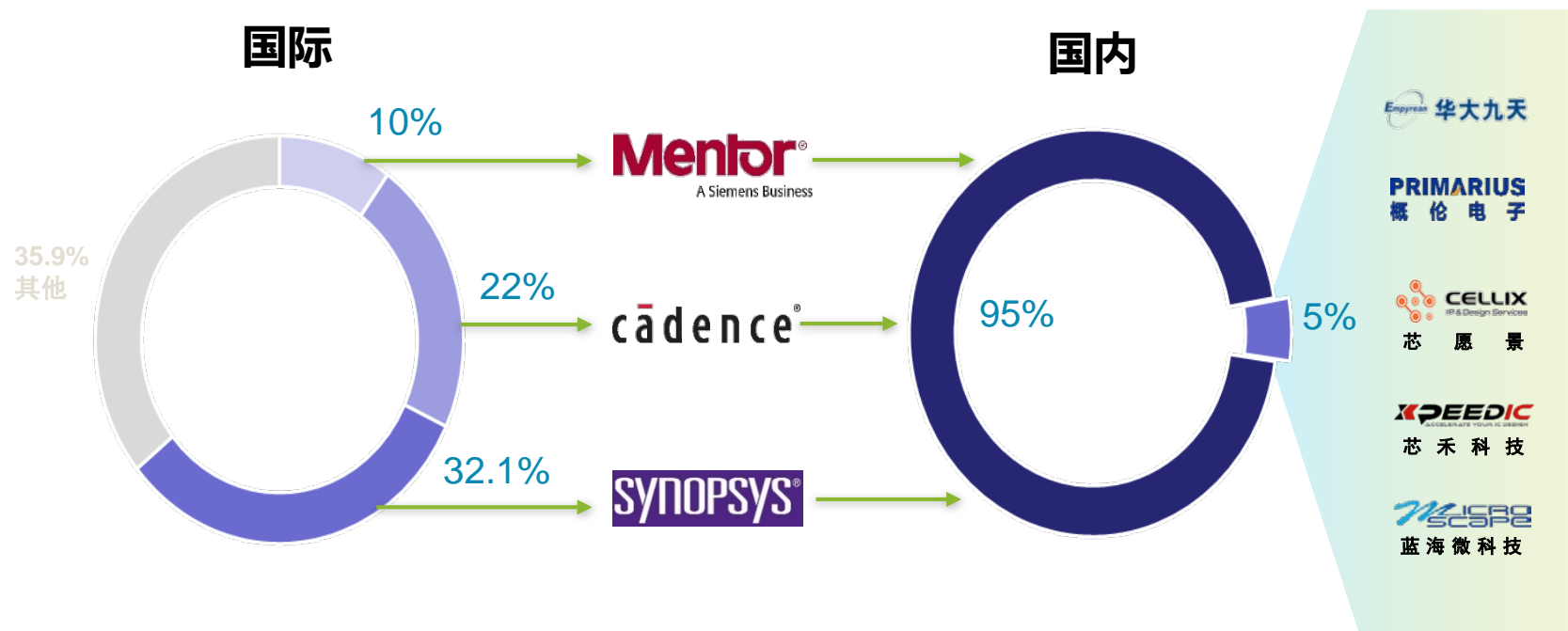
半导体产业链倒金字塔结构

2018-2020年全球EDA市场销售额



2018-2020年全球EDA工具市场竞争格局

芯片之母 --- EDA



Synopsys

- ➡ 高层次综合: Synphony
- ➡ 逻辑综合: Design Compiler
- ➡ 逻辑仿真: VCS
- ➡ 形式验证: Formality
- ➡ 测试生成: TestMax
- ➡ 时序分析: Primetime
- ➡ 物理实现: IC Compiler II
- ➡ 物理验证: IC Validator

Cadence

- ➡ 高层次综合: Stratus
- ➡ 逻辑综合: Genus
- ➡ 逻辑仿真: Xcelium
- ➡ 形式验证: Conformal Equivalence Checker
- ➡ 时序分析: Tempus
- ➡ 物理实现: Innovus
- ➡ 时序建库: Liberate
- ➡ 物理验证: PVS

Mentor

- 高层次综合: Catapult
- 仿真工具: Modelsim
- 物理验证: Calibre
- 后端工具: Olympus
- DFT: BSDArchit
- MBIST: MBIST Architect
- ATPG: Test Kompress

国产工具：前端设计验证工具

- 目前正在开发前端设计验证工具的公司，包括硬件加速器
 - 芯华章
 - 合见工软
 - 思尔芯
- 主要困难
 - 效率与规模与国外工具尚存差距

国产工具：后端工具

➤ 目前正在开发后端工具的公司

- 鸿芯微纳
- 立芯
- 南京E创
- 上海伴芯

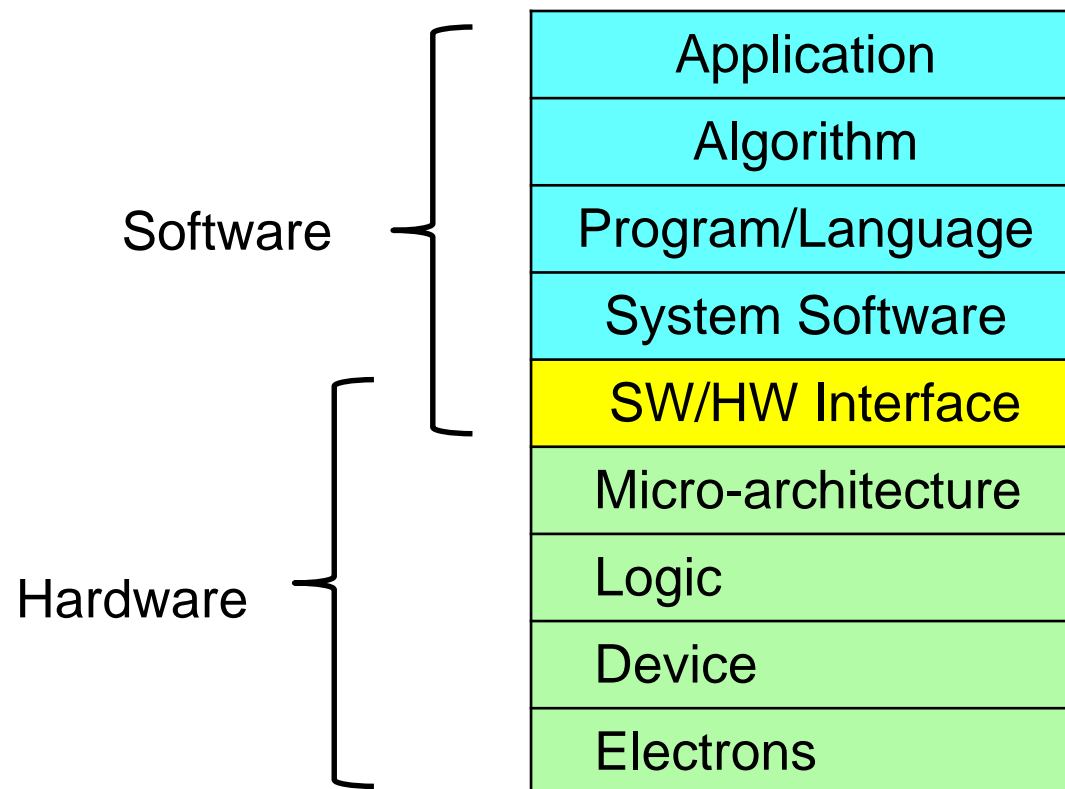
➤ 主要困难

- 模型、规模、效率、流程、先进工艺、时序收敛

国产工具：签核工具

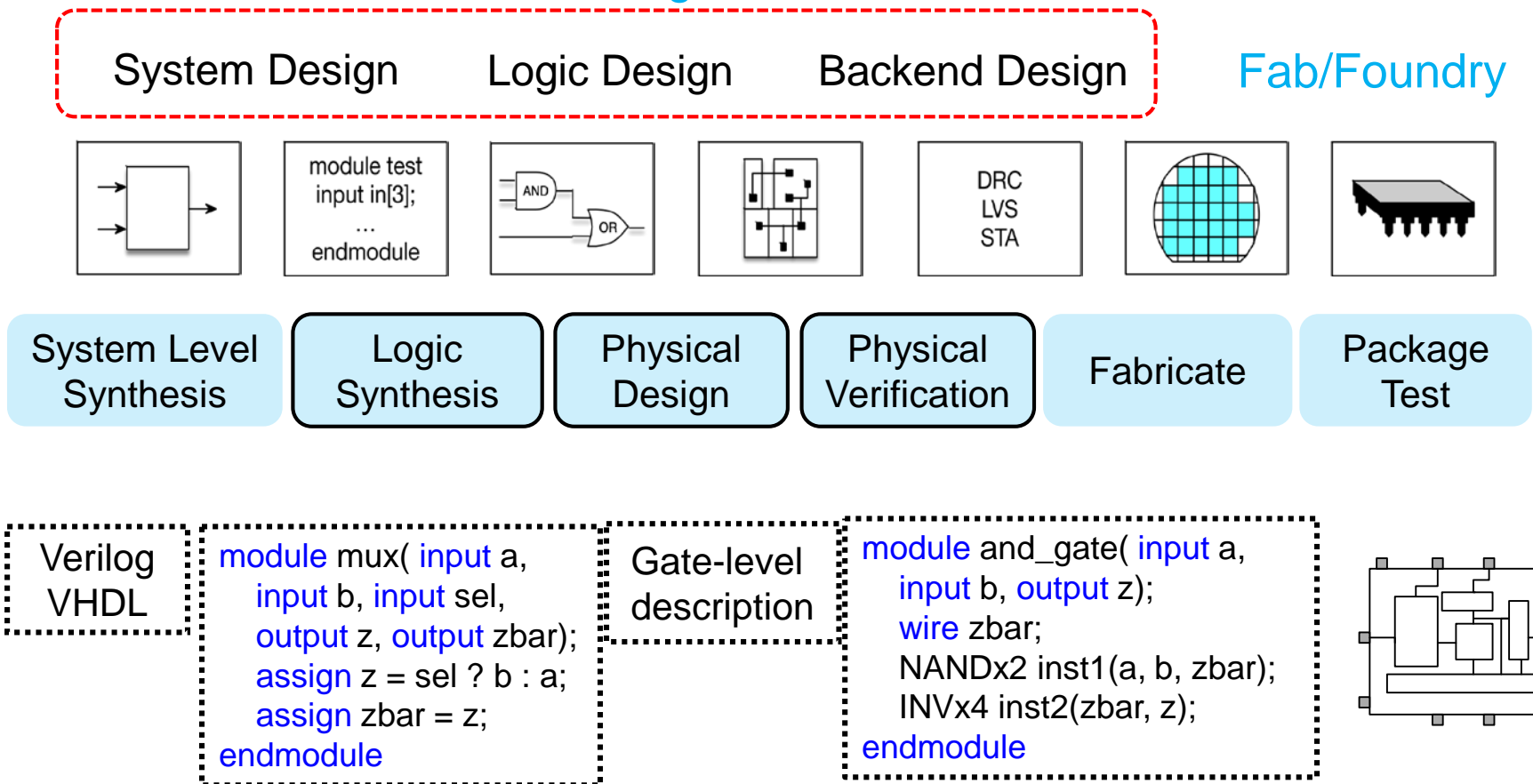
- 目前正在开发签核工具的公司
 - 华大九天(仿真、参数提取、物理验证等)
 - 行芯科技(参数提取、时序分析、可靠性分析等)
- 主要困难
 - 模型校准
 - 效率、并行与规模

EDA: Cross-layer



IC Design Flow

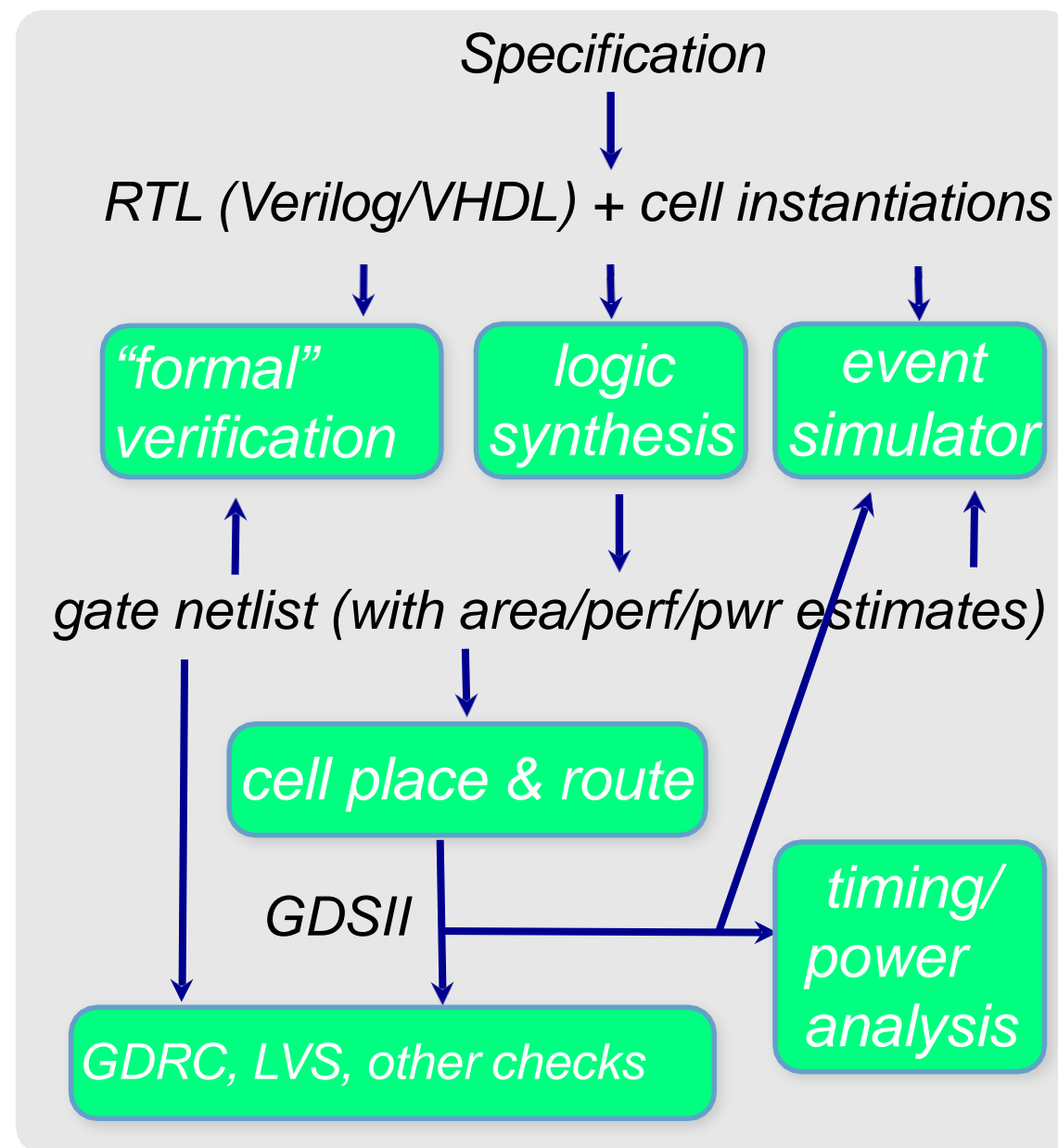
Fabless Design House



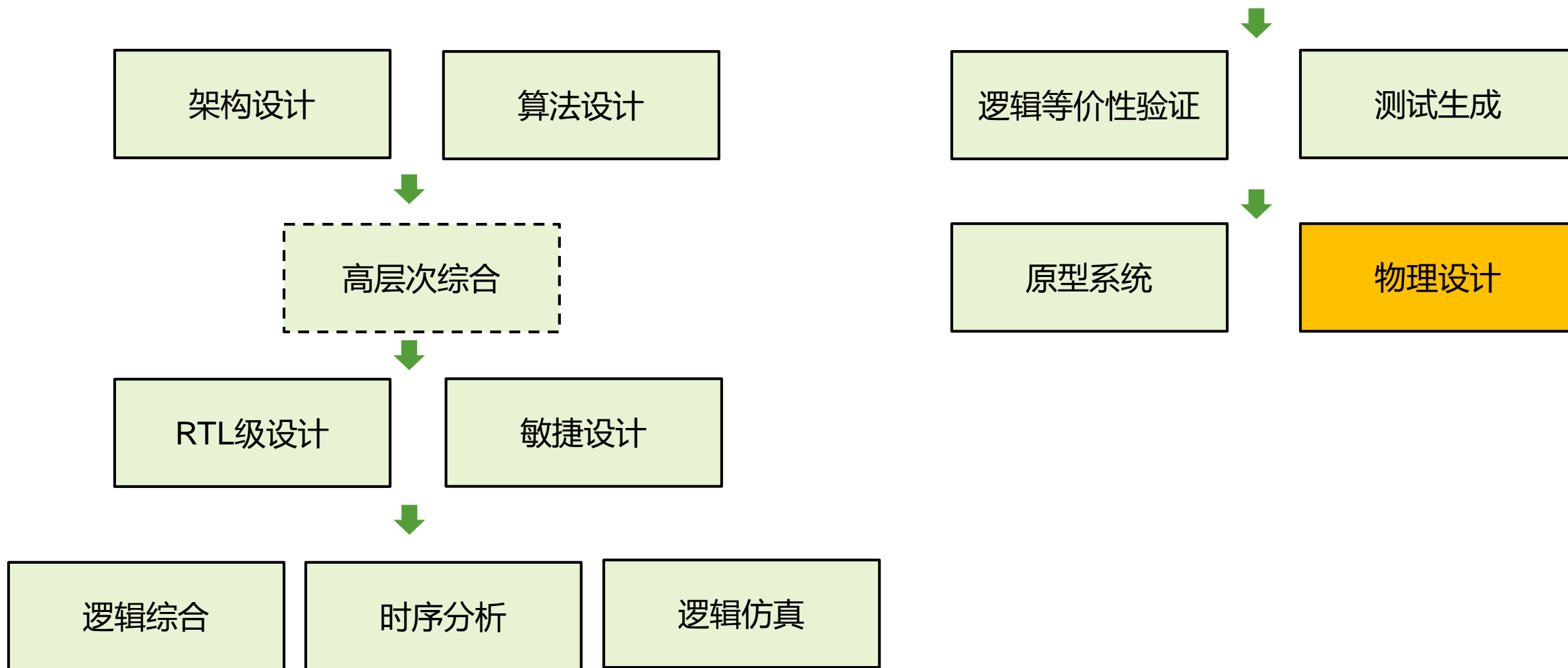
IC Design Flow

RTL Synthesis Based

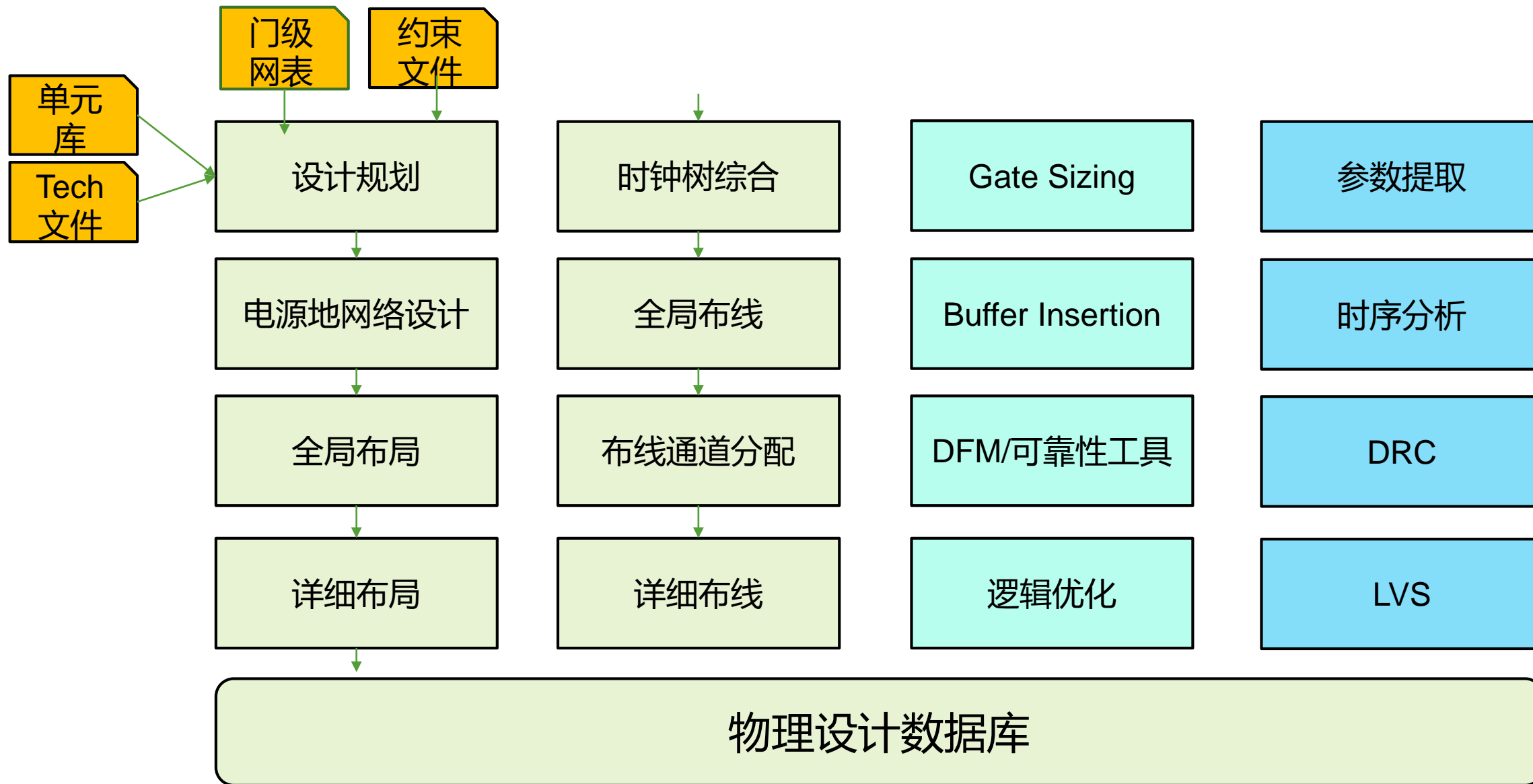
- HDL specifies design as combinational logic + state elements
- Logic Synthesis converts hardware description to gate and flip-flop implementation
Cell instantiations needed for blocks not inferred by synthesis (typically RAM)
- Event simulation verifies RTL
- “Formal” verification compares logical structure of gate netlist to RTL
- Place & route generates layout
- Timing and power checked statically
- Layout verified with LVS and GDRC



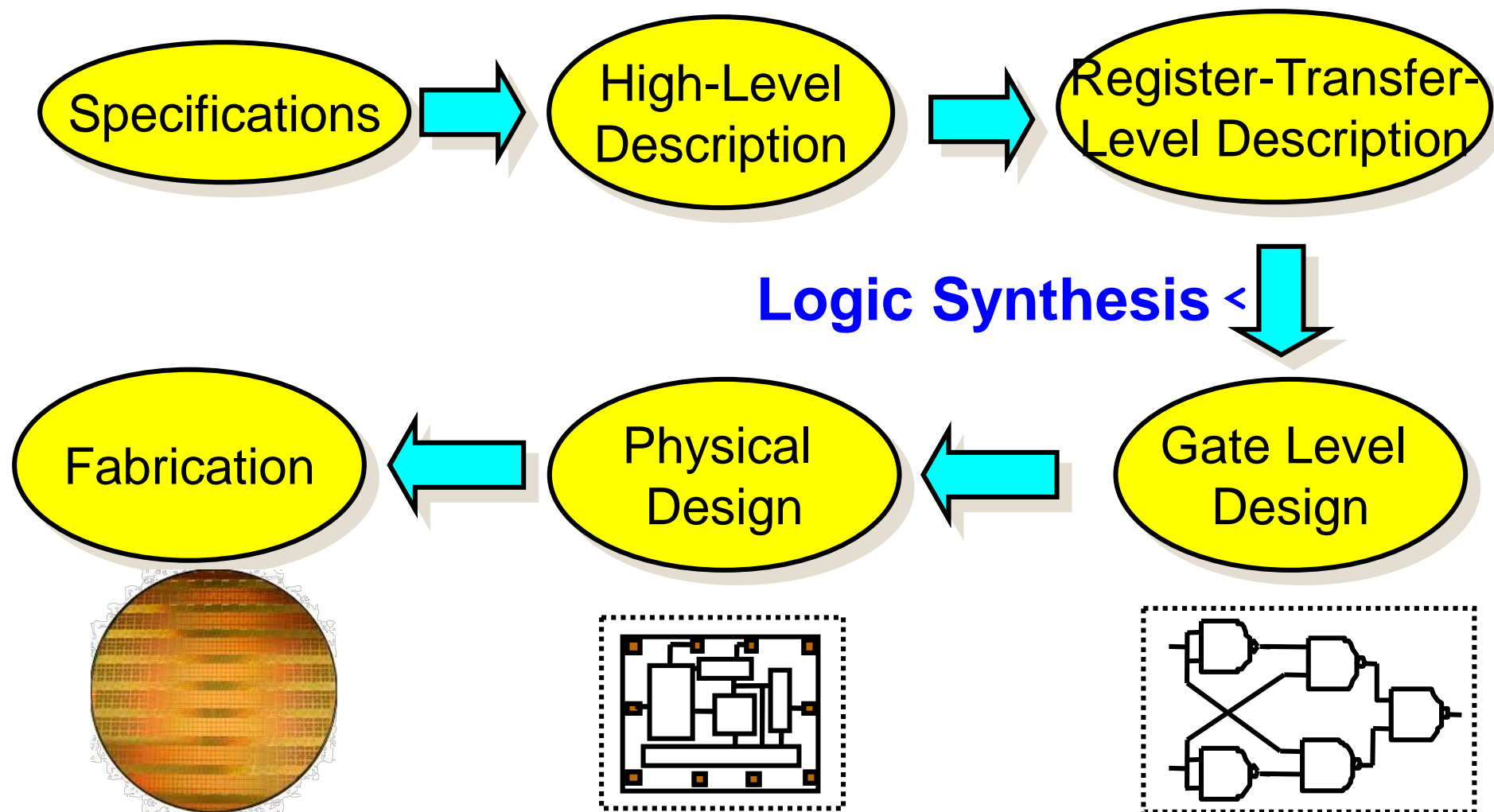
IC Design Flow - frontend



IC Design Flow - backend



IC Design Steps

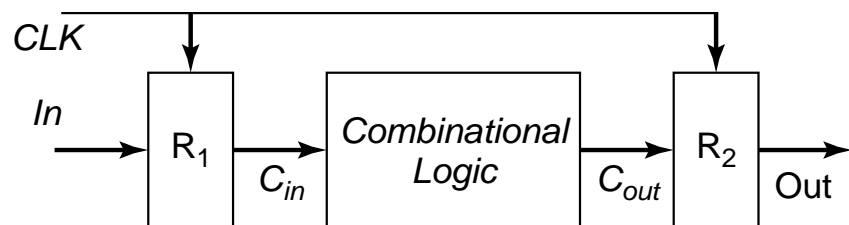


Outline

- Digital Design Flow from a IC design perspective
- Simulation
- Synthesis
- Placement
- Routing

RTL

- RTL级描述语言的发展推动了数字集成电路的自动化
- RTL级描述的是寄存器之间的逻辑功能
 - 逻辑和寄存器都可映射为电路单元
 - 可以描述并发和时序



```
always @(x, in1, in2)
begin
    case (x)
        2'b00: out = in1 * in2;           // in1 * in2
        2'b01: out = in1 << in2[3:0];    // in1 * pow(2, in2[3:0])
        2'b10: out = {in1[IN_WIDTH-2:1], in2[IN_WIDTH-1:2]};
        2'b11: out = in2[IN_WIDTH-1:0] + in1;
    endcase
end
```

Logic Simulation

- 逻辑仿真是逻辑设计验证和调试的重要步骤。逻辑仿真又可以分为RTL级仿真和门级仿真两类
 - RTL级仿真用于功能的仿真和调试，缺乏精确的时序信息
 - 门级仿真可以将门延迟和互连线延迟同时考虑进去，获得更加精确的仿真结果

Logic Simulation

➤ Event-driven

- 根据事件，更新仿真结果，生成新的事件

➤ Cycle-based

- 不考虑cycle内部延迟
- 速度更快

Emulation

- 逻辑仿真包括RTL级仿真和门级仿真，都是事件驱动或存在拓扑上的相互关系，导致仿真并行程度难以大规模提升。对于大规模电路，特别是现代集成电路中的SoC系统，逻辑仿真的效率无法满足全系统仿真验证的需求
- 基于硬件仿真器如FPGA和布尔处理器的方法直接利用硬件门或布尔处理器来仿真逻辑门，利用直接互连或存储数据交换来模拟门之间的连接关系，可以有效的提升仿真的效率

Cadence
Palladium Z1



EEChina.com

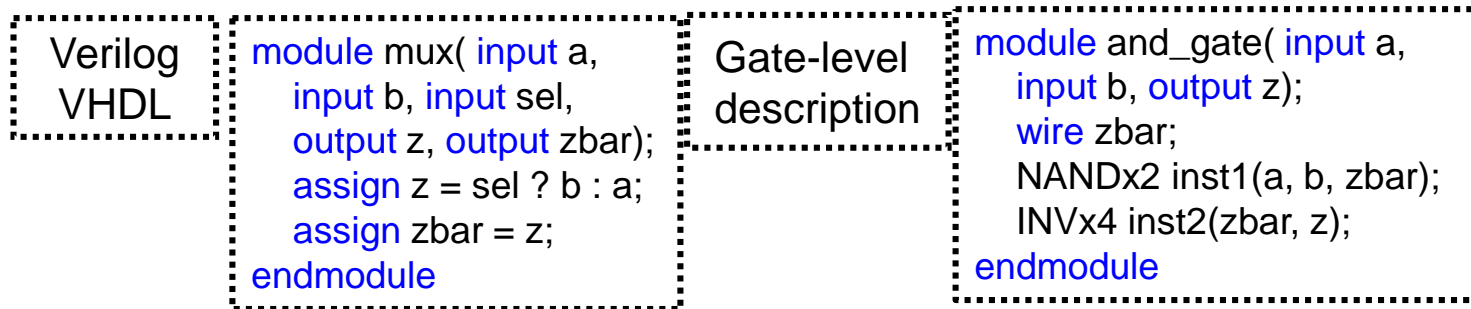
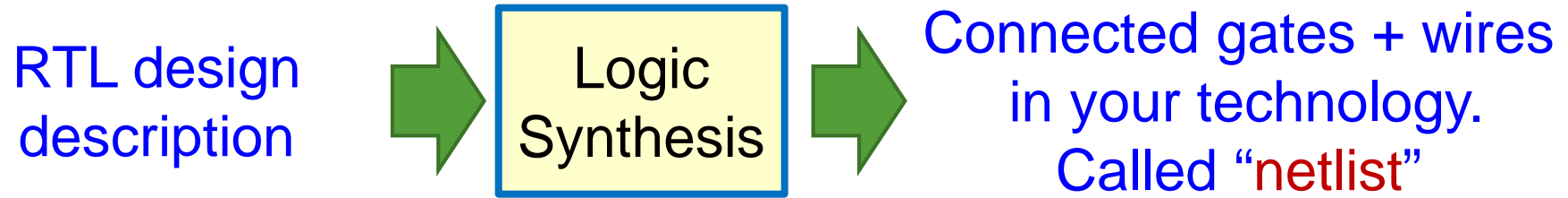


Synopsys
Zebu

Outline

- Digital Design Flow from a IC design perspective
- Simulation
- **Synthesis**
- Placement
- Routing

Logic Synthesis



约束文件 (tcl)

Tool command Language

```

#-----
#   TOP MODULE NAME
#-----
set top_module inno_ddrn_ctrl_ip

#-----
#   SYSTEM CLOCK PERIOD
#-----

set clk_period 3.3 # make even smaller
set nn_clk_period 3.3 #
set ok_clk_period 10 #
set spi_clk_period 10 #

```

```

#create_clock -name nn_clk -period $nn_clk_period [get_ports pad_nn_clk]
create_clock -name nn_clk -period $nn_clk_period [get_ports clk]
create_clock -name v_nn_clk -period $nn_clk_period

set_clock_uncertainty -setup 0.2 nn_clk
set_clock_uncertainty -hold 0.1 nn_clk

```

```

set_input_delay [expr $ok_clk_period * 0.6] -clock v_okClk [get_ports pad_epPipeWrite]
set_input_delay [expr $ok_clk_period * 0.6] -clock v_okClk [get_ports pad_epPipeIn[*]]
set_input_delay [expr $ok_clk_period * 0.6] -clock v_okClk [get_ports pad_epPipeRead]
set_output_delay [expr $ok_clk_period * 0.6] -clock v_okClk [get_ports pad_epPipeOut[*]]
set_output_delay [expr $ok_clk_period * 0.6] -clock v_okClk [get_ports pad_ep60trig_0]
set_output_delay [expr $ok_clk_period * 0.6] -clock v_okClk [get_ports pad_ep60trig_2]
set_output_delay [expr $ok_clk_period * 0.6] -clock v_okClk [get_ports pad_ep60trig_3]
set_output_delay [expr $ok_clk_period * 0.6] -clock v_okClk [get_ports pad_ep60trig_4]
set_output_delay [expr $ok_clk_period * 0.6] -clock v_okClk [get_ports pad_ep60trig_5]

set_input_delay [expr $spi_clk_period * 0.6] -clock v_spi_clk [get_ports pad_i_SPI_CS_n]
set_input_delay [expr $spi_clk_period * 0.6] -clock v_spi_clk [get_ports pad_i_SPI_MOSI]
set_output_delay [expr $spi_clk_period * 0.6] -clock v_spi_clk [get_ports pad_o_SPI_MISO]

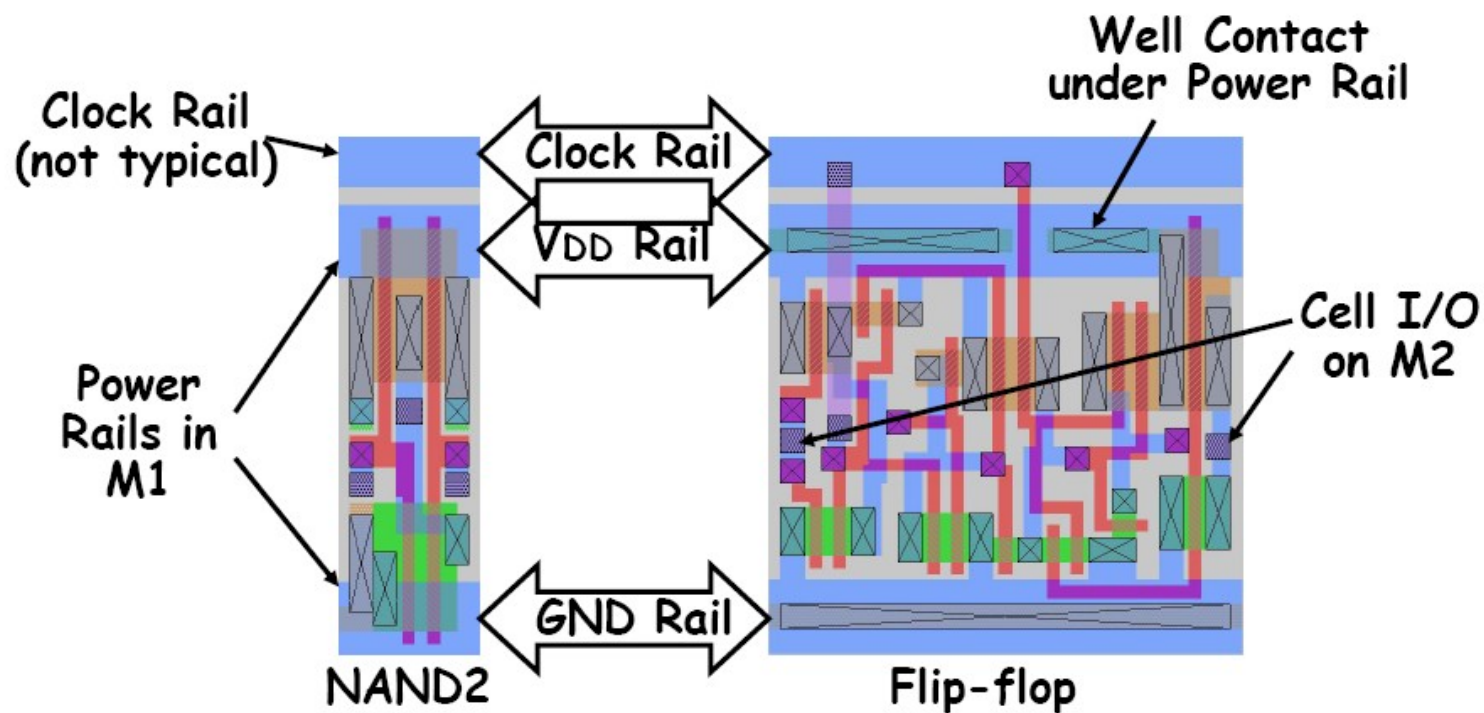
set_output_delay [expr $nn_clk_period * 0.6] -clock v_nn_clk [get_ports pad_error]

```

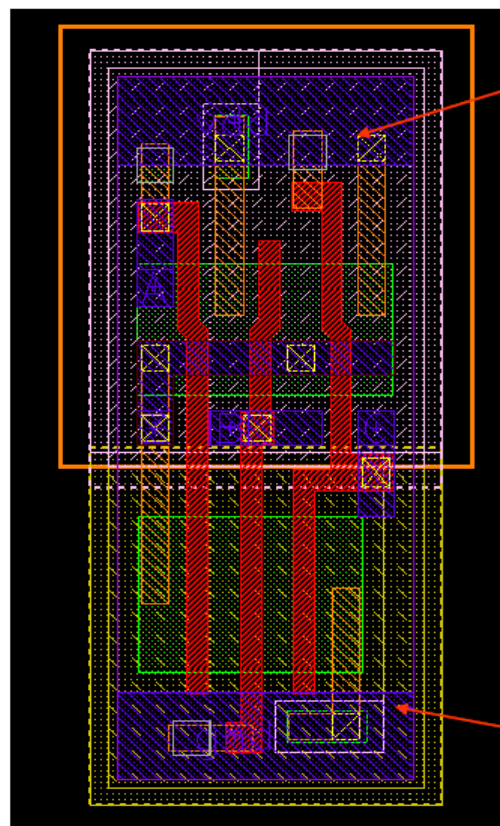
Standard Cell

Cells have standard height but vary in width

Designed to connect power, ground, and wells by abutment



Standard Cell



Power Supply Line (V_{DD}) Delay in (ns)!!

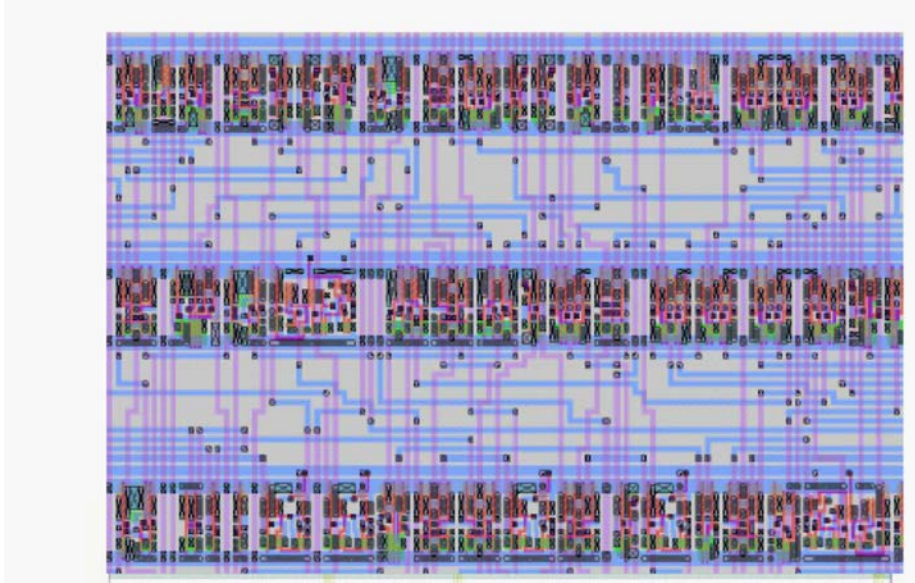
Path	1.2V - 125°C	1.6V - 40°C
$In1-t_{pLH}$	$0.073+7.98C+0.317T$	$0.020+2.73C+0.253T$
$In1-t_{pHL}$	$0.069+8.43C+0.364T$	$0.018+2.14C+0.292T$
$In2-t_{pLH}$	$0.101+7.97C+0.318T$	$0.026+2.38C+0.255T$
$In2-t_{pHL}$	$0.097+8.42C+0.325T$	$0.023+2.14C+0.269T$
$In3-t_{pLH}$	$0.120+8.00C+0.318T$	$0.031+2.37C+0.258T$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

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

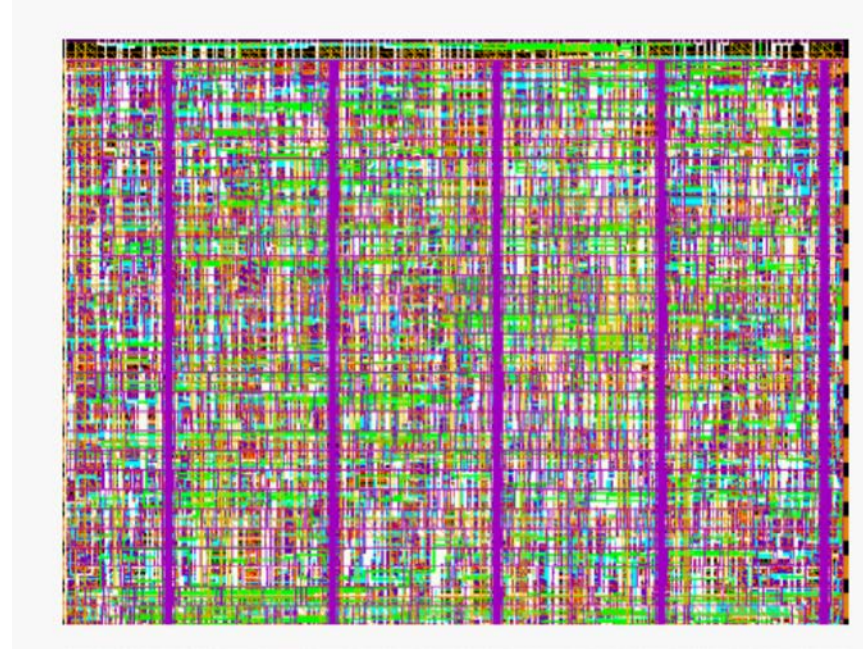
Ground Supply Line (GND)

- Each library cell (FF, NAND, NOR, INV, etc.) and the variations on size (strength of the gate) is fully characterized across temperature, loading, etc.

Standard Cell



1μm, 2-metal process

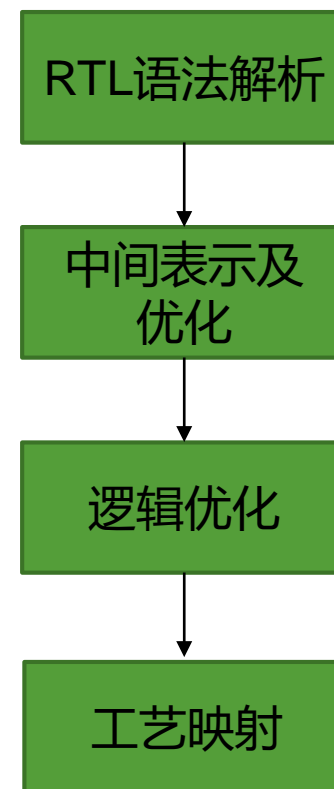


*Modern sub-100nm process
“Transistors are free things
that fit under wires”*

- With limited # metal layers, dedicated routing channels were needed
- Currently area dominated by wires

逻辑综合基本过程

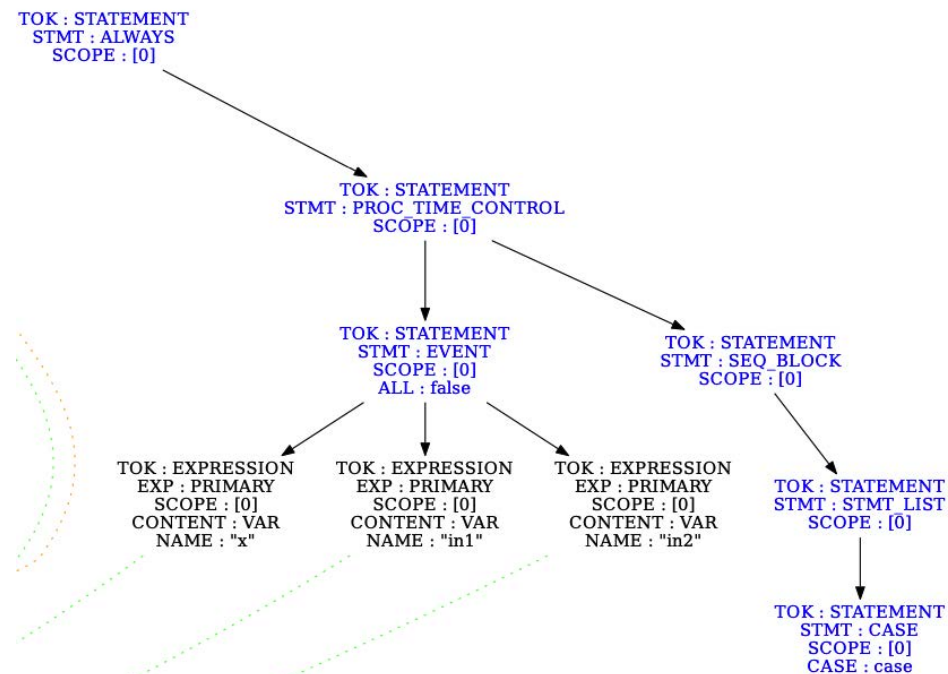
- 逻辑综合与高级语言编译类似，基于多层的中间表示和不同的pass对中间表示进行优化
- 中间表示优化与前端语言无关，与工艺无关
- 后端工艺映射再将工艺无关优化结果映射为实际工艺实现
- 一般可以包含两种级别的中间表示
 - RTL级中间表示：用于RTL级的优化及逻辑综合
 - 逻辑级中间表示：用于逻辑优化



逻辑综合 - 前端

- RTL语法解析将RTL语言转化为抽象语法树(AST)，并提供访问函数接口来遍历抽象语法树上的节点

```
always @(x, in1, in2)
begin
  case (x)
    2'b00: out = in1 * in2;           // in1 * in2
    2'b01: out = in1 << in2[3:0];    // in1 * pow(2,
in2[3:0])
    2'b10: out = {in1[IN_WIDTH-2:1],
in2[IN_WIDTH-1:2]};
    2'b11: out = in2[IN_WIDTH-1:0] + in1;
  endcase
end
```



RTL语言工艺无关的中间表示

- 在生成抽象语法树后，逻辑综合的前端一般也会维持一个类似LLVM的中间表示结构
- 逻辑综合的RTL级中间表示，
 - 基本的module/procedure等RTL级表示，
 - 基本的抽象逻辑、计算单元和原语如寄存器、与或非单元、加减乘除基本运算单元、MUX单元、存储器等

抽象语法树到RTL中间表示的转换

- 根据RTL语法进行推断转换，将一些控制、计算转换成对应抽象逻辑、计算单元和原语，这些转换包括但不限于：
 - 将if/else和switch/case等语法转化为MUX原语；
 - 将基本的与或非转化为基本抽象逻辑单元；
 - 将加减乘除等操作转化为基本的抽象计算单元；
 - 根据RTL语义，推断出寄存器单元和存储单元

```
module ff_with_en_and_async_reset(clock, reset, enable, d, q);  
input clock, reset, enable, d;  
output reg q;  
always @(posedge clock, posedge reset)  
    if (reset)  
        q <= 0;  
    else if (enable)  
        q <= d;  
endmodule
```



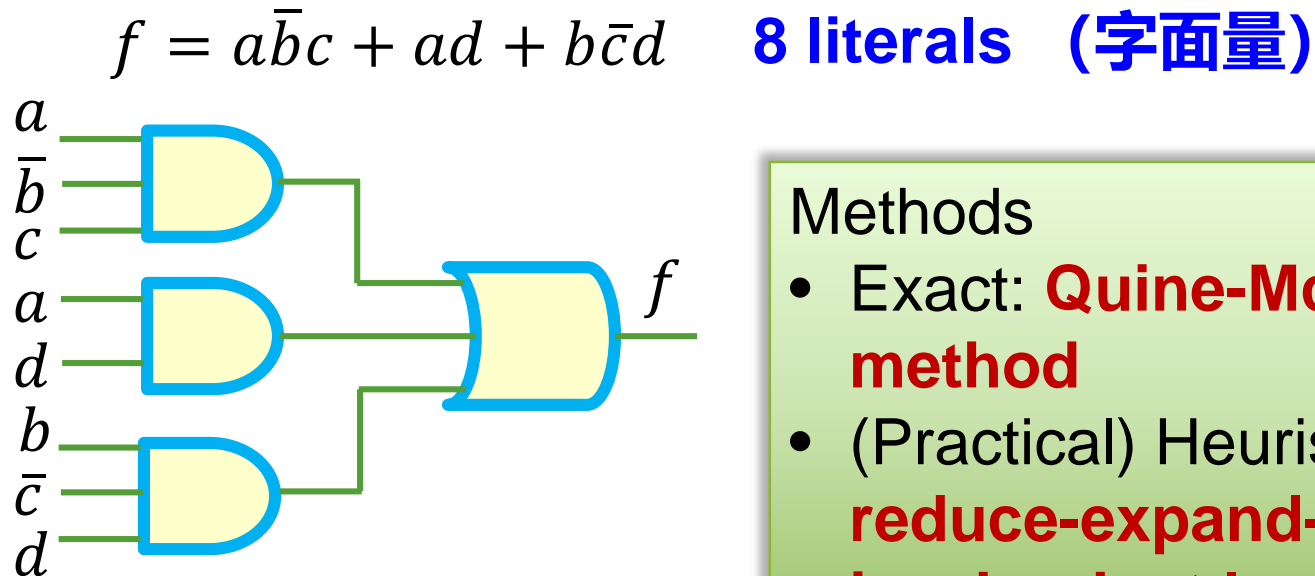
```
cell $adff $procdff$6  
    parameter \ARST_POLARITY 1'1  
    parameter \ARST_VALUE 1'0  
    parameter \CLK_POLARITY 1'1  
    parameter \WIDTH 1  
    connect \ARST \reset  
    connect \CLK \clock  
    connect \D $0\q[0:0]  
    connect \Q \q  
end  
cell $mux $procmux$3  
    parameter \WIDTH 1  
    connect \A \q  
    connect \B \d  
    connect \S \enable  
    connect \Y $0\q[0:0]  
end
```

逻辑优化

- 逻辑优化是逻辑综合的关键步骤
- 早期的逻辑优化：工艺相关的逻辑优化。基于规则，针对门级网表，构建一个优化窗口，对窗口内的门电路依据实际实现进行优化。规则多，且随着单元电路加入会快速增加
- 逻辑优化的发展：工艺无关的逻辑优化+工艺映射。中间表示与工艺无关，但可以表征最终实现后的代价。工艺映射再将中间表示面向特定工艺进行映射
- 商用工具：工艺无关优化+逻辑映射+基于规则的映射后优化
- 两层逻辑优化 - 多级逻辑电路

Two-Level Logic Design

- A logic implementation based on **sum-of-products (SOP)** expression.
 - First level: many AND gates.
 - Second level: one “big” OR gate.
- Optimization target: minimize **number of literals**

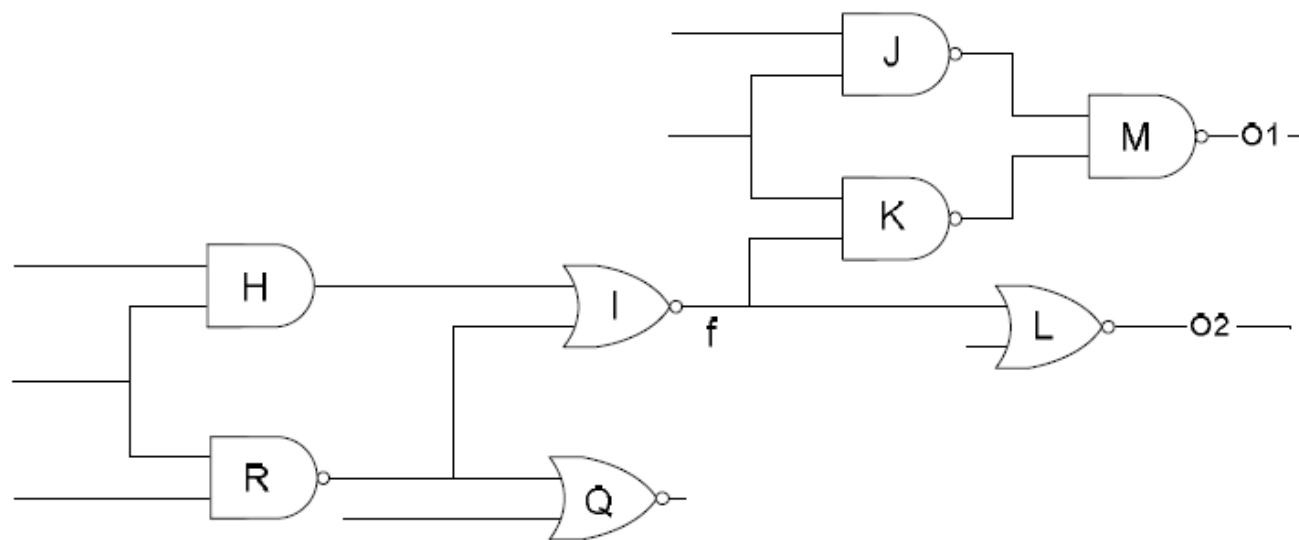


Methods

- Exact: **Quine-McCluskey method**
- (Practical) Heuristic: **reduce-expand-irredundant loop**

Multi-Level Design

- Modern design style!

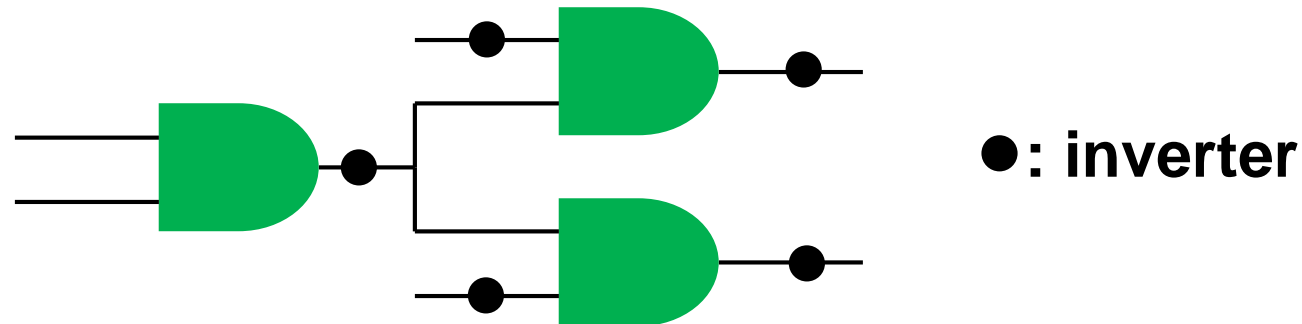


- Two phases in multi-level logic synthesis:
 - Technology-independent synthesis
 - Technology mapping

Technology-Independent Synthesis

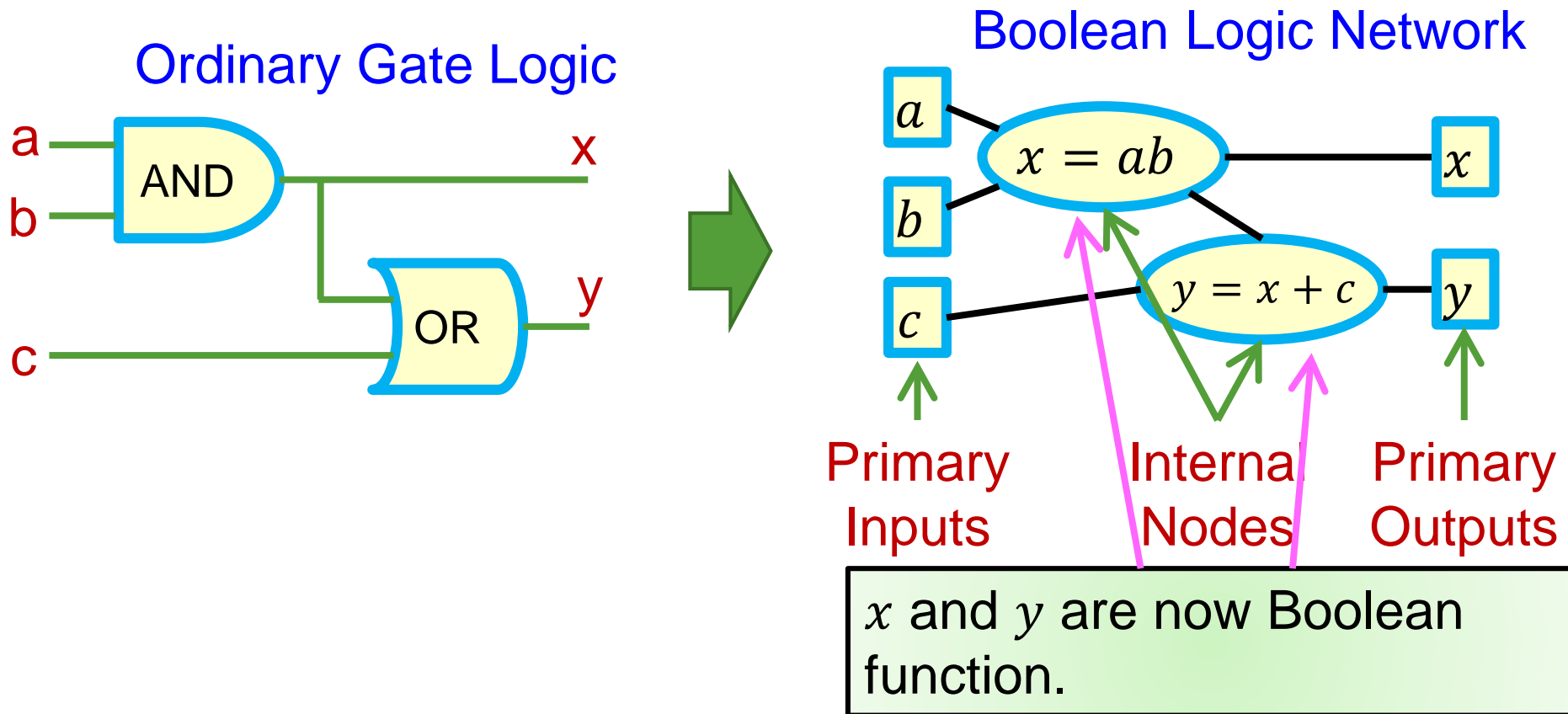
- Output: **An abstract representation of Boolean function**
 - It is **not** the actual **gate-level netlist**
 - The result is called: “**technology independent**” logic or “**uncommitted**” logic
 - Example: **AND-inverter graph (AIG)**, **Boolean logic network**, etc.

AND-inverter graph (AIG)



Boolean Logic Network Model

- A **graph** of connected blocks, like any logic diagram, but now individual component blocks are **2-level Boolean functions in SOP form**.

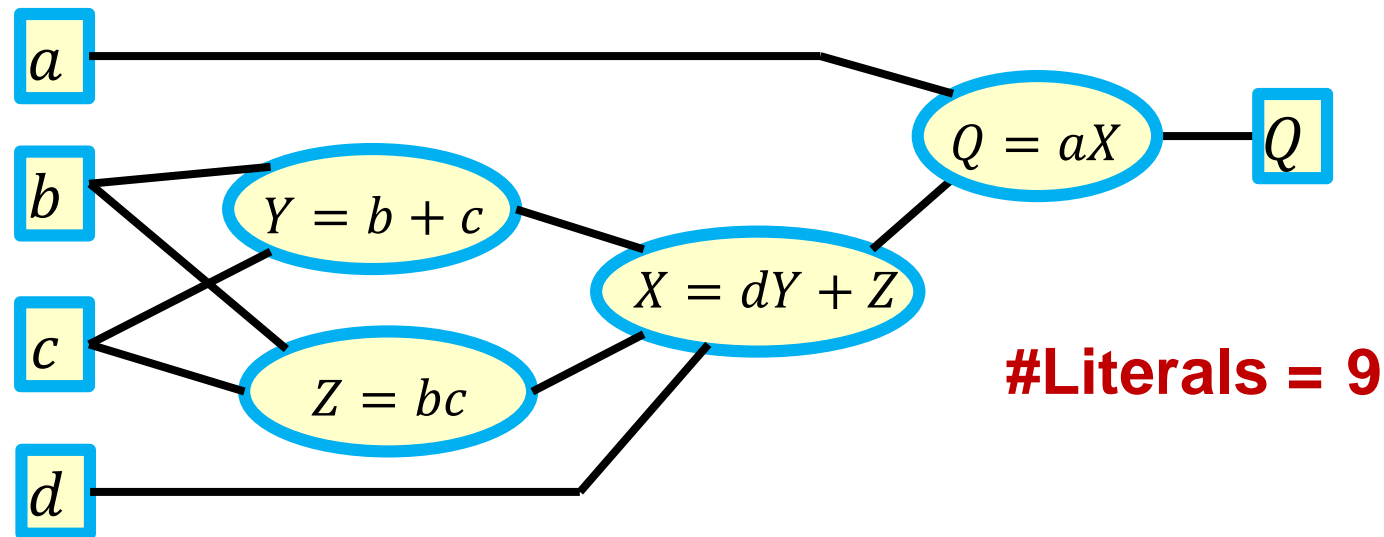


Boolean Logic Network: What to Optimize?

- A simplistic but surprisingly useful metric:

Total literal count

- Count **every** appearance of **every** variable on right hand side of “=” in **every internal node**.
- Delays also matter, but here only focus on **area**.



Optimizing Multilevel Logic: Big Ideas

- **Boolean logic network** is a **data structure**. So, what **operators** do we need?
- 3 basic kinds of operators:
 - **Simplify** network nodes: no change in # of nodes, just simplify insides, which are **SOP form**.
 - **Remove** network nodes: take “too small” nodes, **substitute them back** into fanouts.
 - This is not too hard. This is mostly manipulating the graph, simple SOP edits.
 - **Add** new network nodes: this is **factoring**. Take big nodes, split into smaller nodes.
 - This is a **big deal** and takes effort.

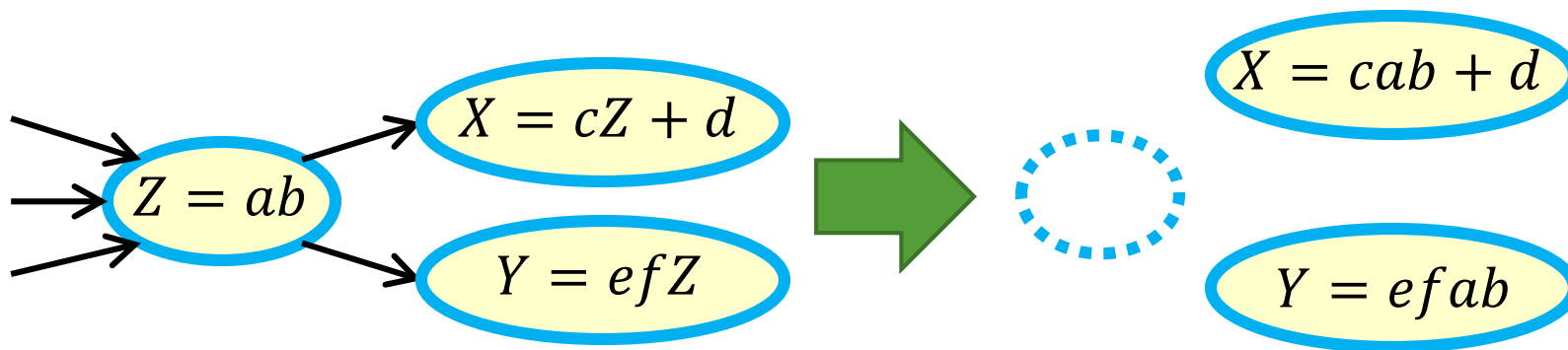
Simplifying a Node

- This is **2-level synthesis**.
- Just run ESPRESSO on 2-level form **inside** the node, to reduce # literals.
- As structural changes happen across network, “**insides**” of nodes may present opportunity to simplify.



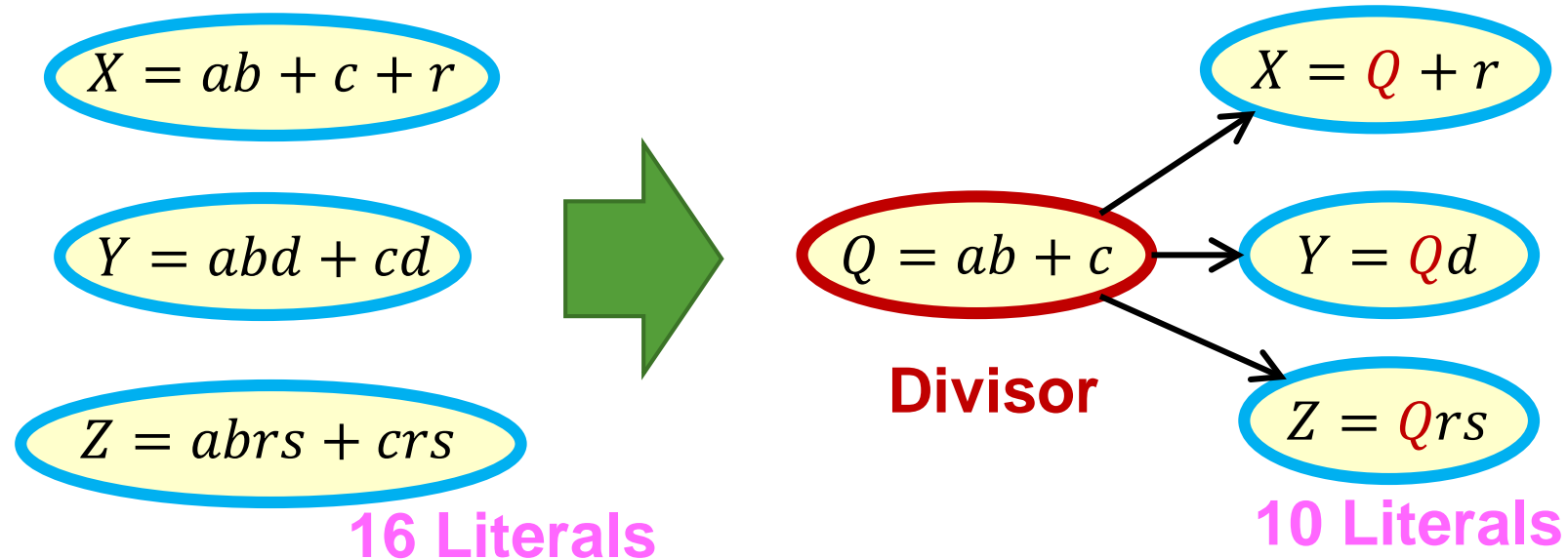
Removing a Node

- Typical case is you have a “**small**” factor which doesn’t seem to be worth making it a **separate** node.
- “**Push**” it back into its fanouts, make those nodes bigger, and hope you can use 2-level simplification on them.



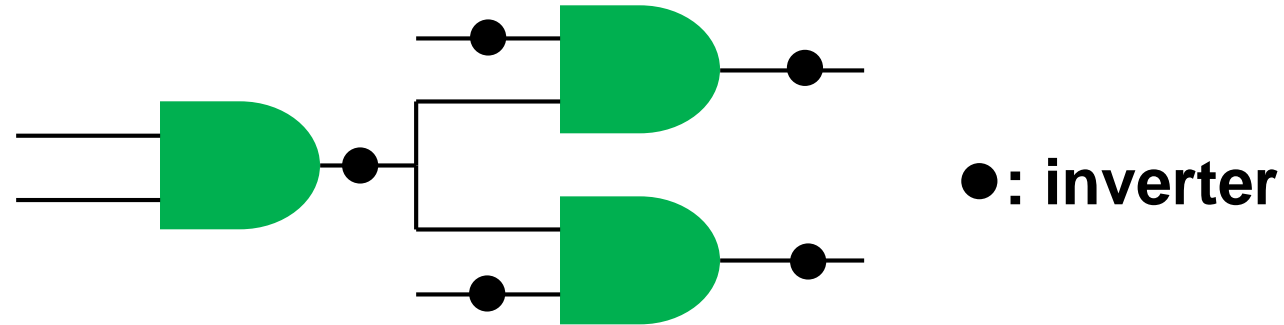
Adding new Nodes

- Look at existing nodes, identify **common divisors**, extract them, connect as fanins.
- Tradeoff: **more** delay (another level of logic), but **fewer literals** (less gate area).
- Based on **algebraic model**, **algebraic division**, **kernels**



AND-Inverter Graphs (AIGs)

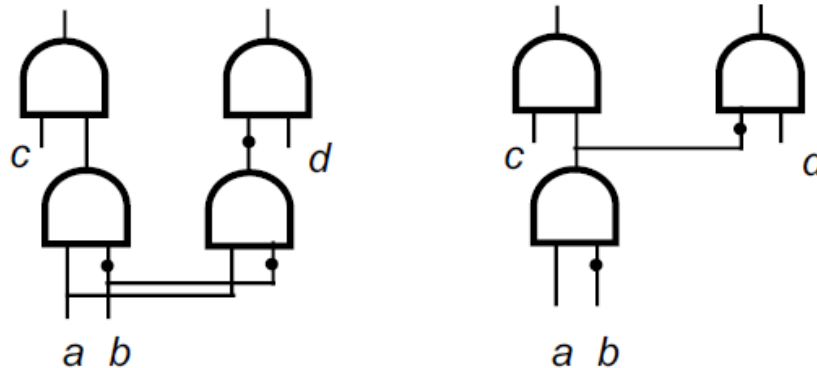
- Any Boolean function can be represented as an AIG!



- But AIG is not **canonical**
 - For the same function, many different AIG representations.
- For area reduction, minimize the **number of AND nodes**.
- Tool: ABC: A System for Sequential Synthesis and Verification
 - <http://www.eecs.berkeley.edu/~alanmi/abc>

AIG Structural Hashing (Strashing)

- When building AIGs, always add AND node
 - When an AIG is constructed without strashing, AND gates are added one at a time without checking whether AND gate with the same fanins already exists
- One-level strashing
 - When adding a new AND-node, check the hash table for a node with the same input pair (fanin)
 - If it exists, return it; otherwise, create a new node

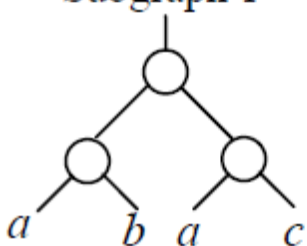
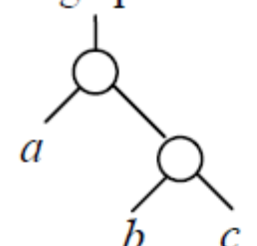
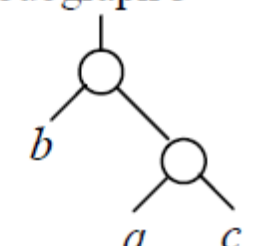


AIG Optimization Operations

- Rewriting ✓
- Resubstitution ✓
- Balancing
- Refactoring

Rewriting: Phase 1

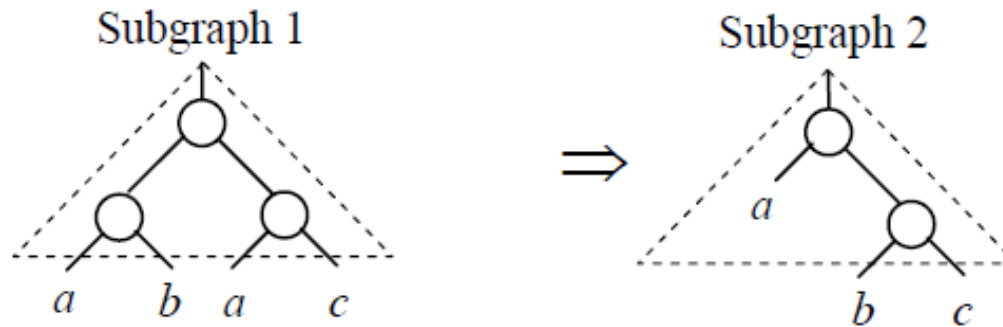
- All two-level AIG subgraphs are **pre-computed** and stored in a **hash table** by their functionality.
- The table contains all non-redundant AIG implementations of logic functions with four variables or less

Hash table	Function	All non-redundant AIG implementations		
		Subgraph 1	Subgraph 2	Subgraph 3
	abc			
	$a(b + c)$...		
		

Rewriting: Phase 2

- For each node in topological order, find its two-level AIG subgraph and compute its Boolean function.
- Use the function to access the hash table to find equivalent subgraphs.
- Try each subgraph, while taking into account logic sharing between the new subgraph nodes and the existing nodes.
- Choose the subgraph with largest save in # of nodes.

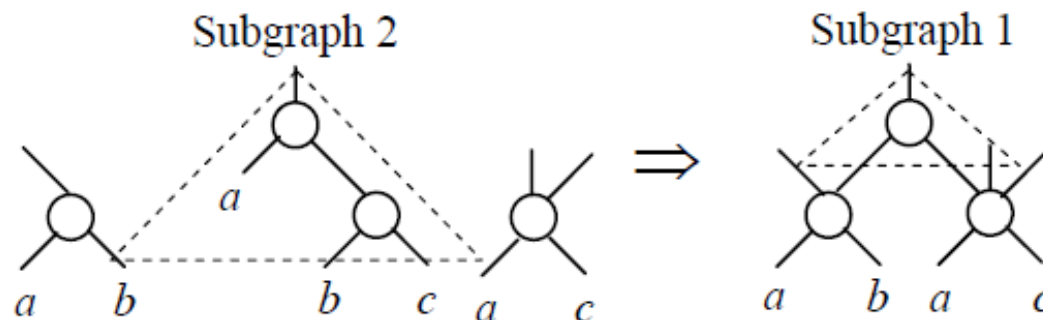
Example 1



Rewriting: Phase 2

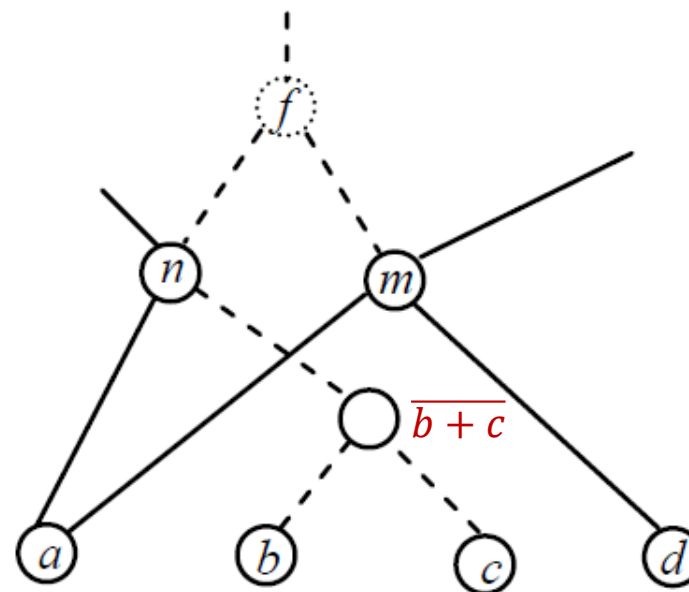
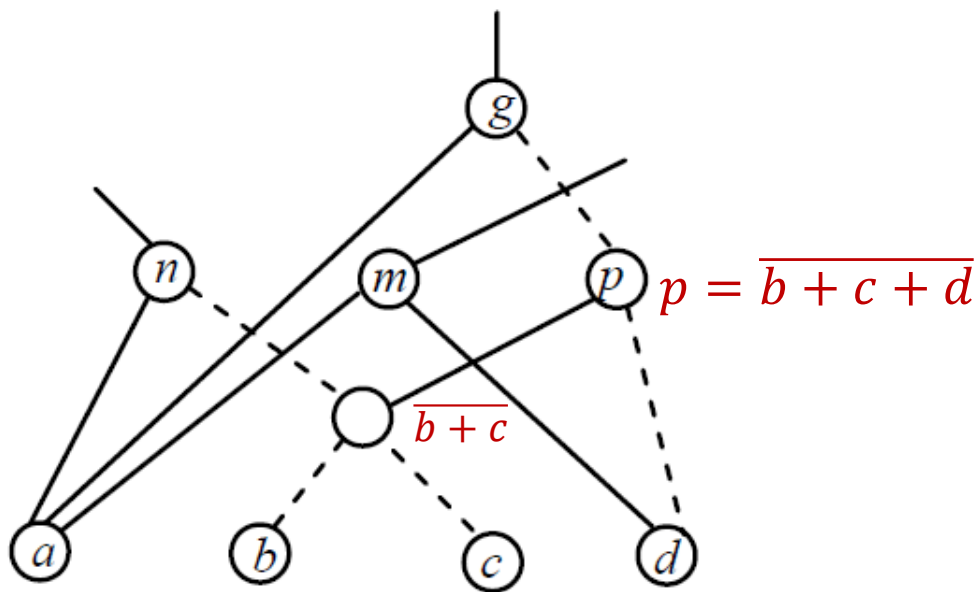
- For each node in topological order, find its two-level AIG subgraph and compute its Boolean function.
- Use the function to access the hash table to find equivalent subgraphs.
- Try each subgraph, while taking into account logic sharing between the new subgraph nodes and the existing nodes.
- Choose the subgraph with largest save in # of nodes.

Example 2



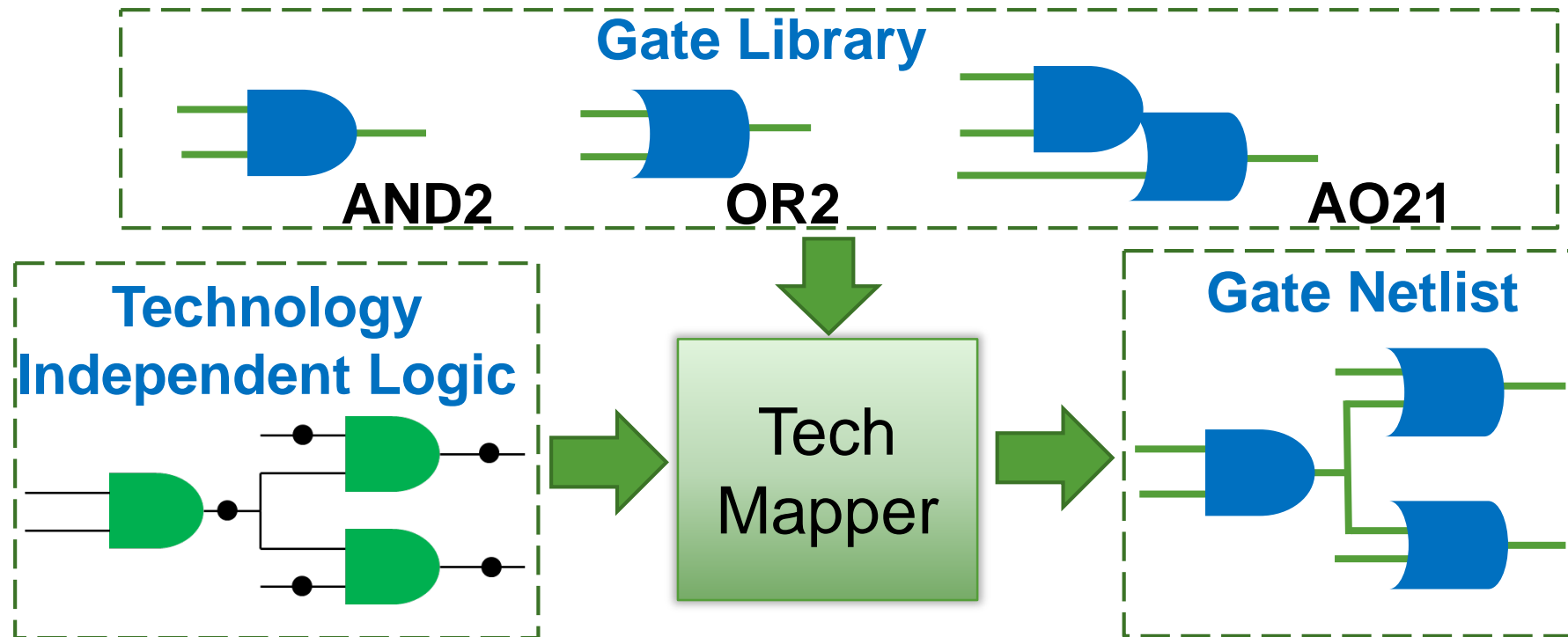
Resubstitution

- Express the function of a node using other nodes (divisors).
- Example:
 - Replace function $g = a(b + c + d)$ as $f = n + m$, where $n = a(b + c)$ and $m = ad$



Technology Mapping

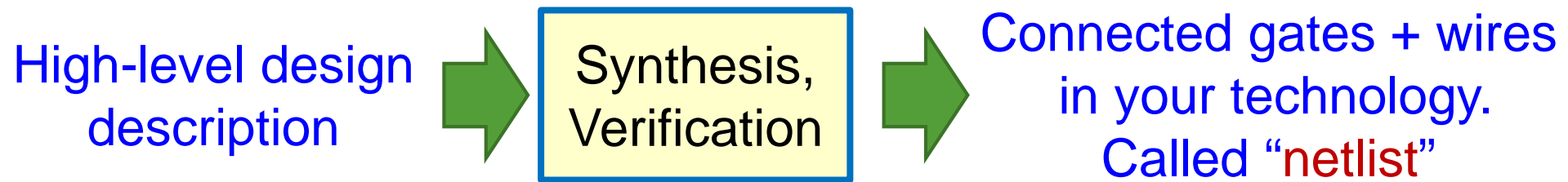
- Given a gate library, from which we can pick gates, and **technology independent** logic F , build a final **gate netlist** for F using only these gates from the given library.



From Logic... To Layout...

➤ What you know...

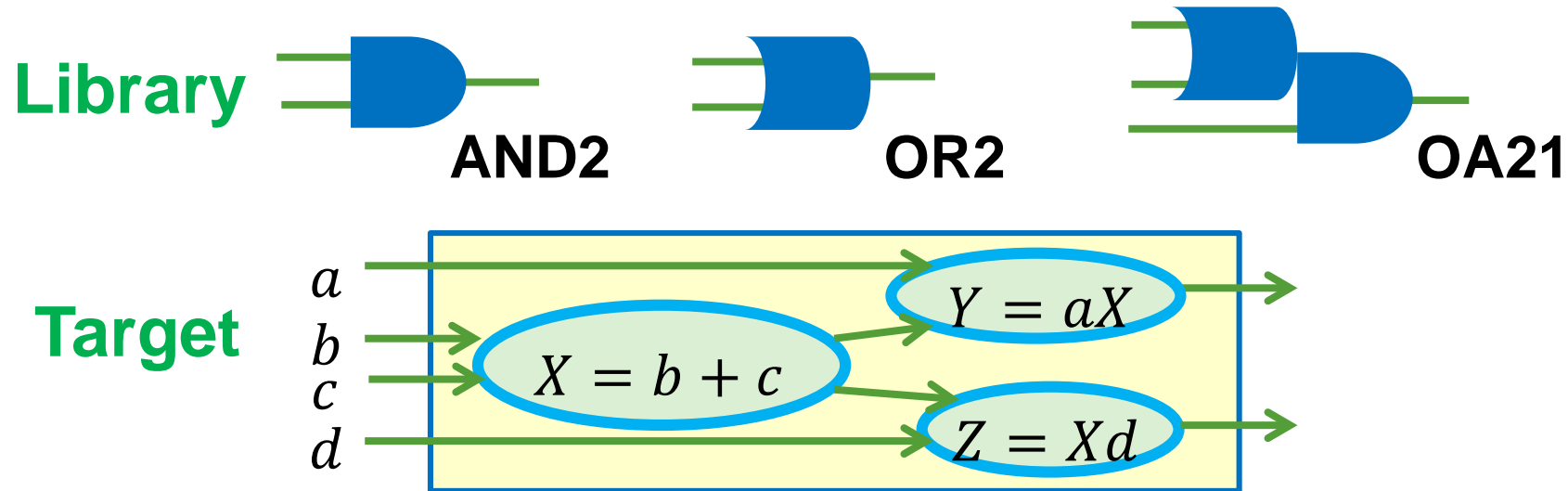
- Computational Boolean algebra, representation, some verification, some synthesis.
- This is what happens in the “**front end**” of the ASIC design process.



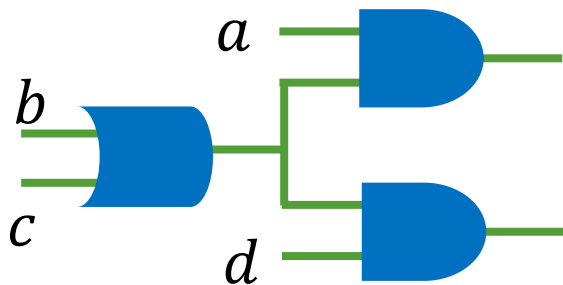
➤ One key **connecting** step:

- How to **transform** result of multi-level synthesis into real gates for layout task.
- Called: **Technology Mapping**, or **Tech Mapping**.

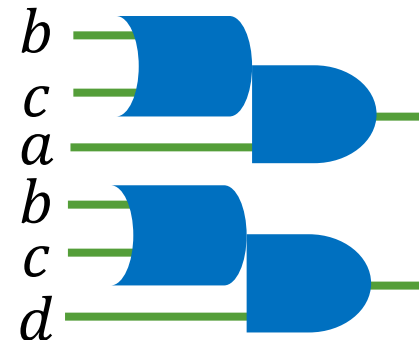
Tech Mapping: Simple Example



Obvious Mapping



Non-obvious Mapping



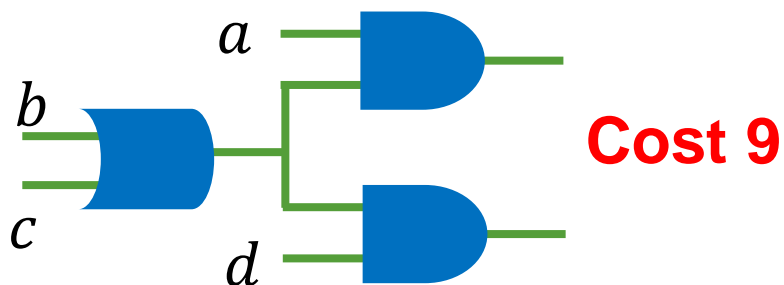
Tech Mapping

► Why choose a **non-obvious** mapping?

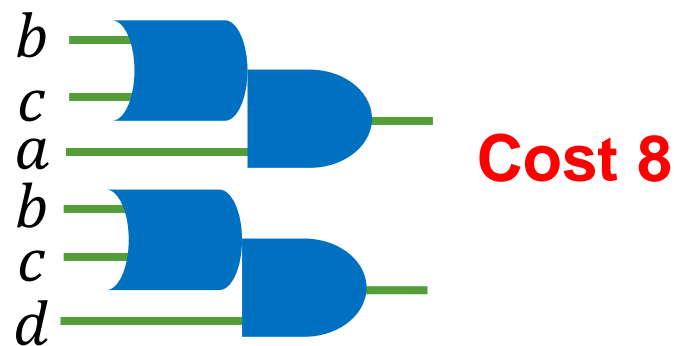
- Answer 1: **Cost**. Suppose each gate in library has a cost associated with it, e.g., the **silicon area** of the standard cell gate.



Obvious Mapping

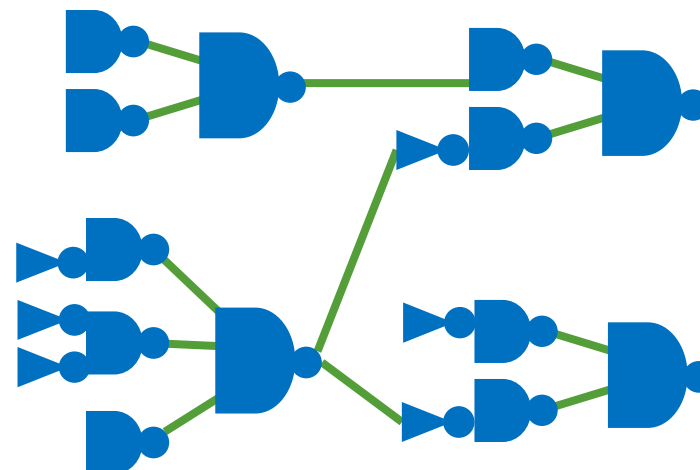
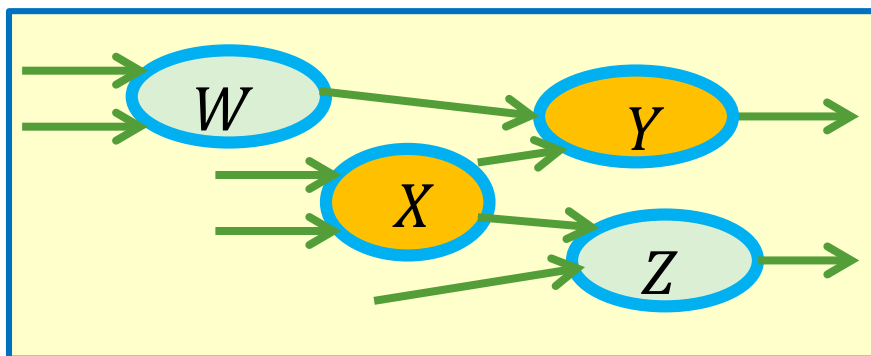


Non-obvious Mapping



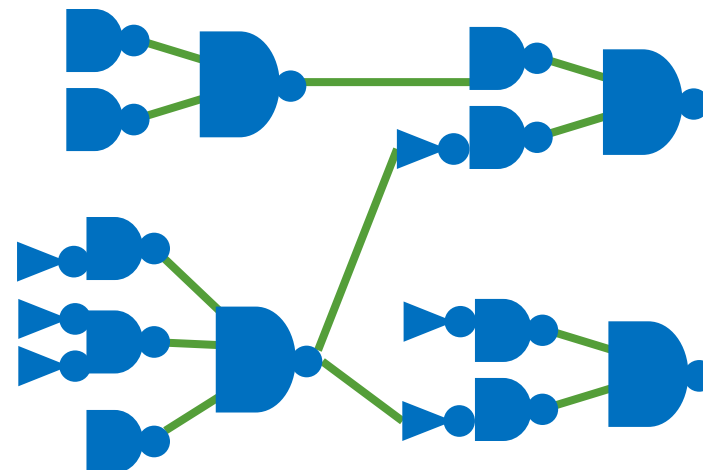
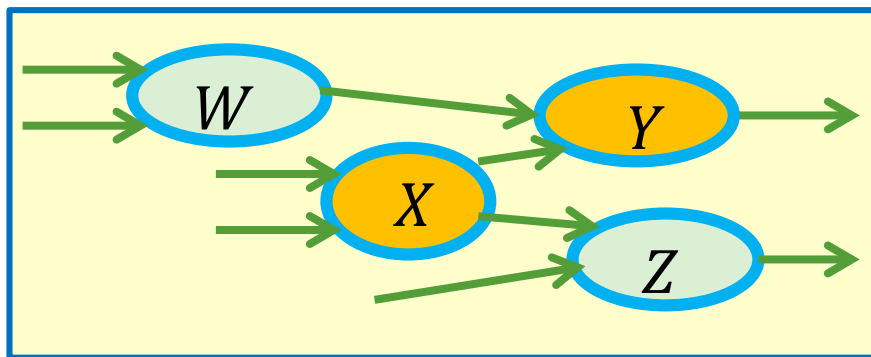
What “Technology Mapping” Does?

- By transforming every SOP form in each node into NAND & NOT gates ...
 - ... Boolean logic network **disappears**. W, X, Y, Z boundaries go away.
 - We have one BIG “**flat**” network of **NANDs** and **NOTs**. This is what we are going to map.



Technology Mapping

- ➡ Multi-level synthesis produces this big network of simple gates.
- ➡ How to transform – **map** – this onto standard cells in our library?

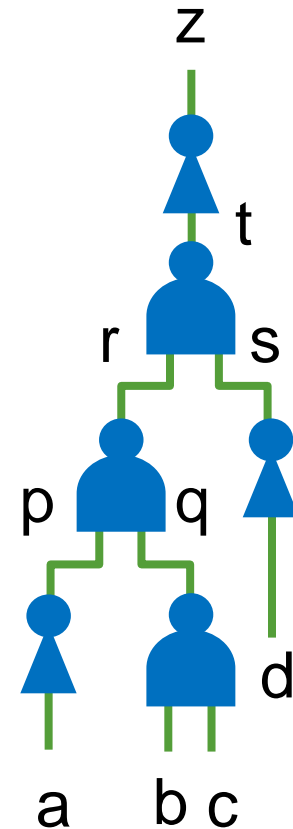


Technology Mapping as Tree Covering

- One famous, simple model of problem:
 - Your logic network to be mapped is **a tree of simple gates**.
 - We assume uncommitted form is **2-input NAND** (“**NAND2**”) and **NOT** gates, only.
 - Your library of available “real” gate types is also represented in this form.
 - Each gate is represented as a **tree** of **NAND2** and **NOT** gates, with associated **cost**.
- Method is surprisingly **simple** and **optimal**.
 - Reference: Kurt Keutzer, “DAGON: Technology Binding and Local Optimization by DAG Matching,” Proc. ACM/IEEE Design Automation Conference (DAC), 1987.

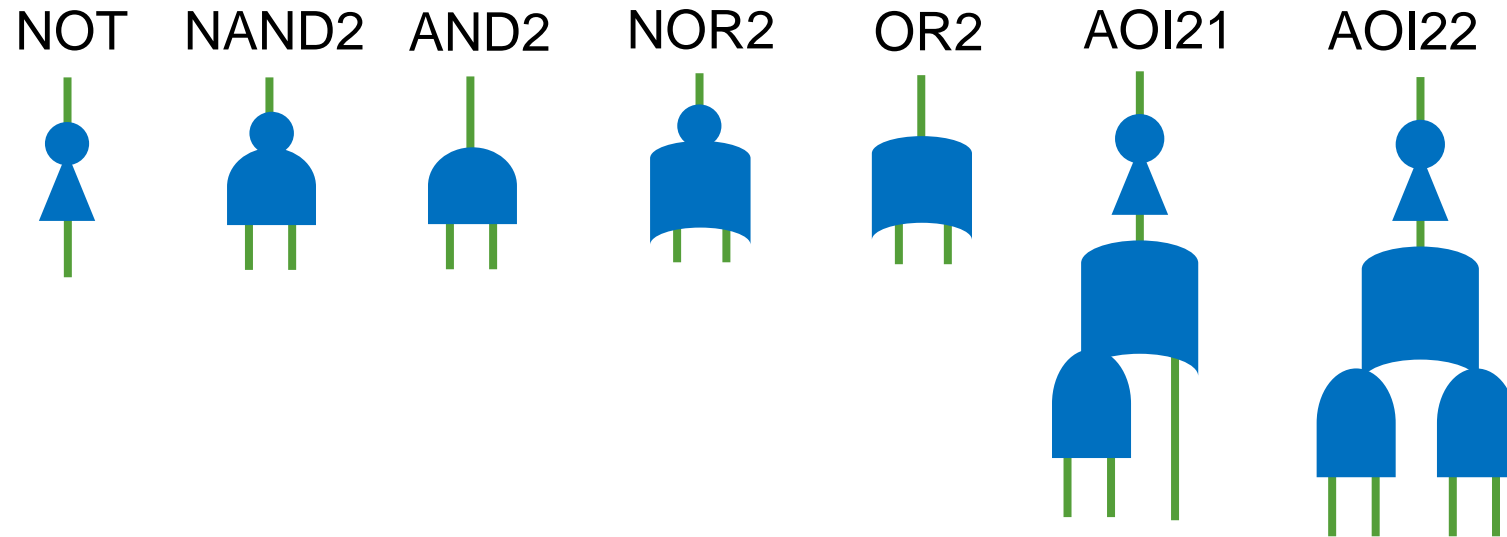
Tree Covering Example: Start with...

- Here is your uncommitted logic to be **tech mapped**.
 - This is what results from our multi-level synthesis optimization...
 - ... after replacing all SOP forms in the network nodes with **NAND2/NOT**.
 - Called the **subject tree**.
 - (Restrict to **NAND2** to keep this simple. Also label not only inputs but all internal wires too.)



Tree Covering: Your Technology Library

➤ And, here is a very simple **technology library**.

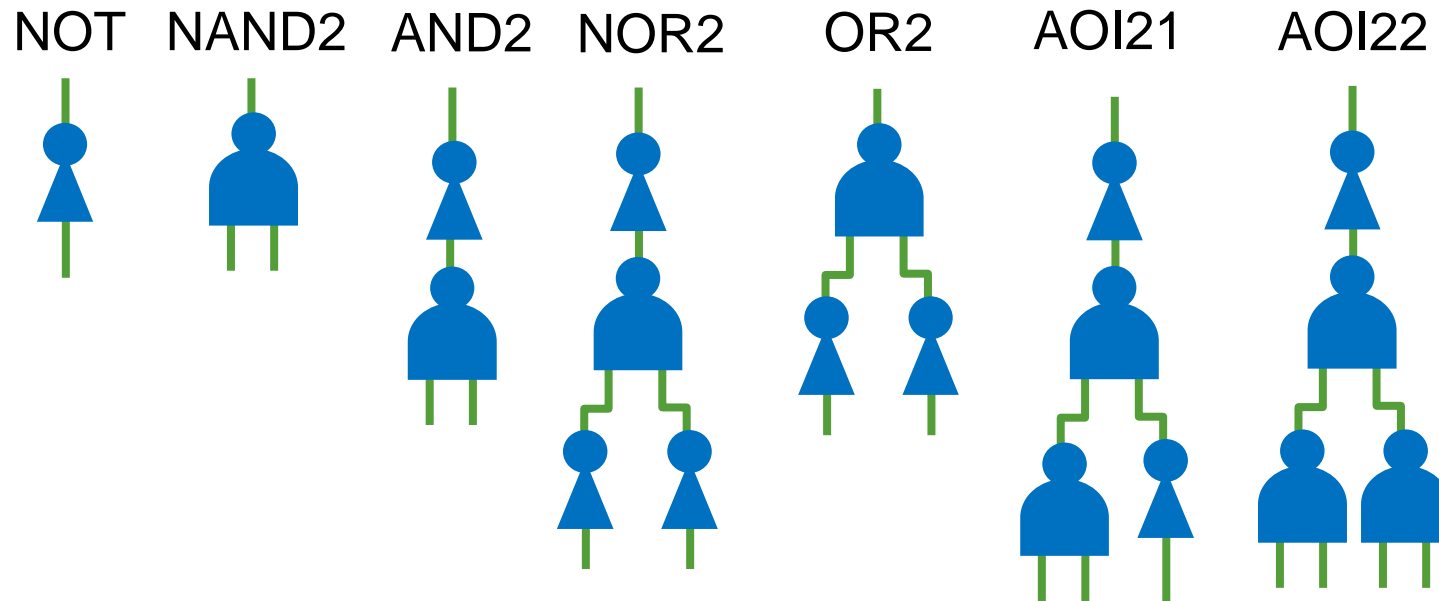


➤ First problem: this is **not** in the required **NAND2/NOT-only** form. Must transform.

Tree Covering: Representing Library

➤ Transforming to NAND2/NOT form is **easy**.

— Just apply **De Morgan's law**.



➤ Each library element in this form is called a **pattern tree**.

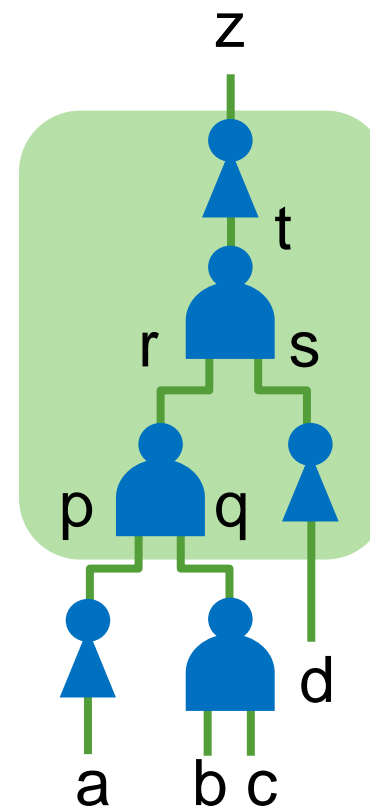
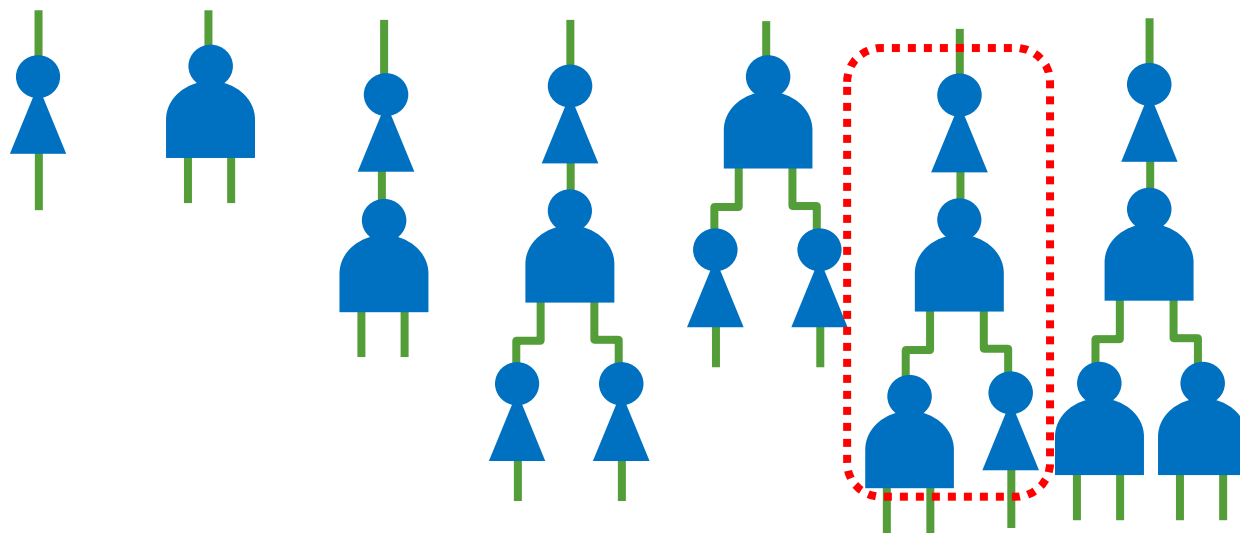
Essential Idea in “Tree Covering”

➤ **Avoid** any **Boolean algebra**!

➤ Just do “**pattern matching**”.

- Find where, in subject graph, the library pattern “matches”.
- NAND matches NAND, NOT matches NOT, etc.
- This is called: **structural mapping**.

NOT NAND2 AND2 NOR2 OR2 AOI21 AOI22

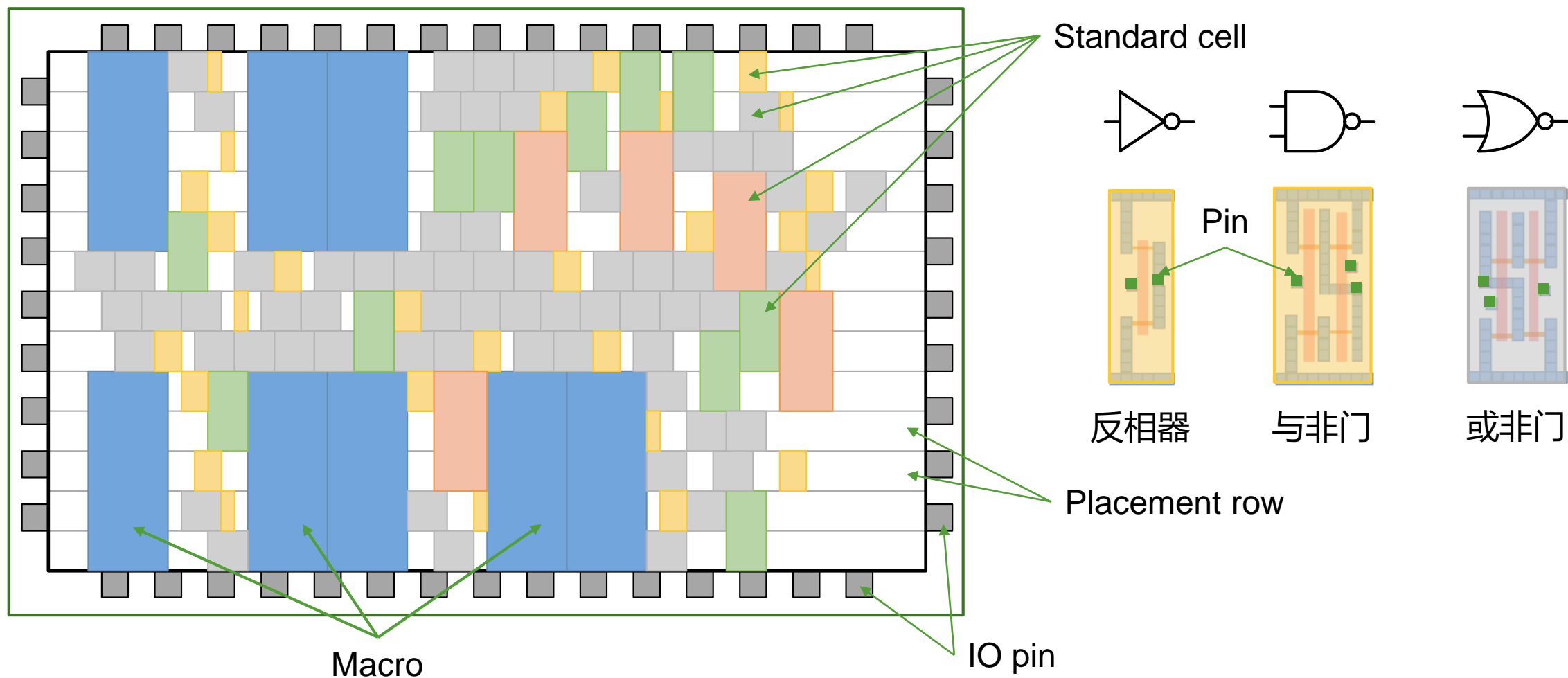


Outline

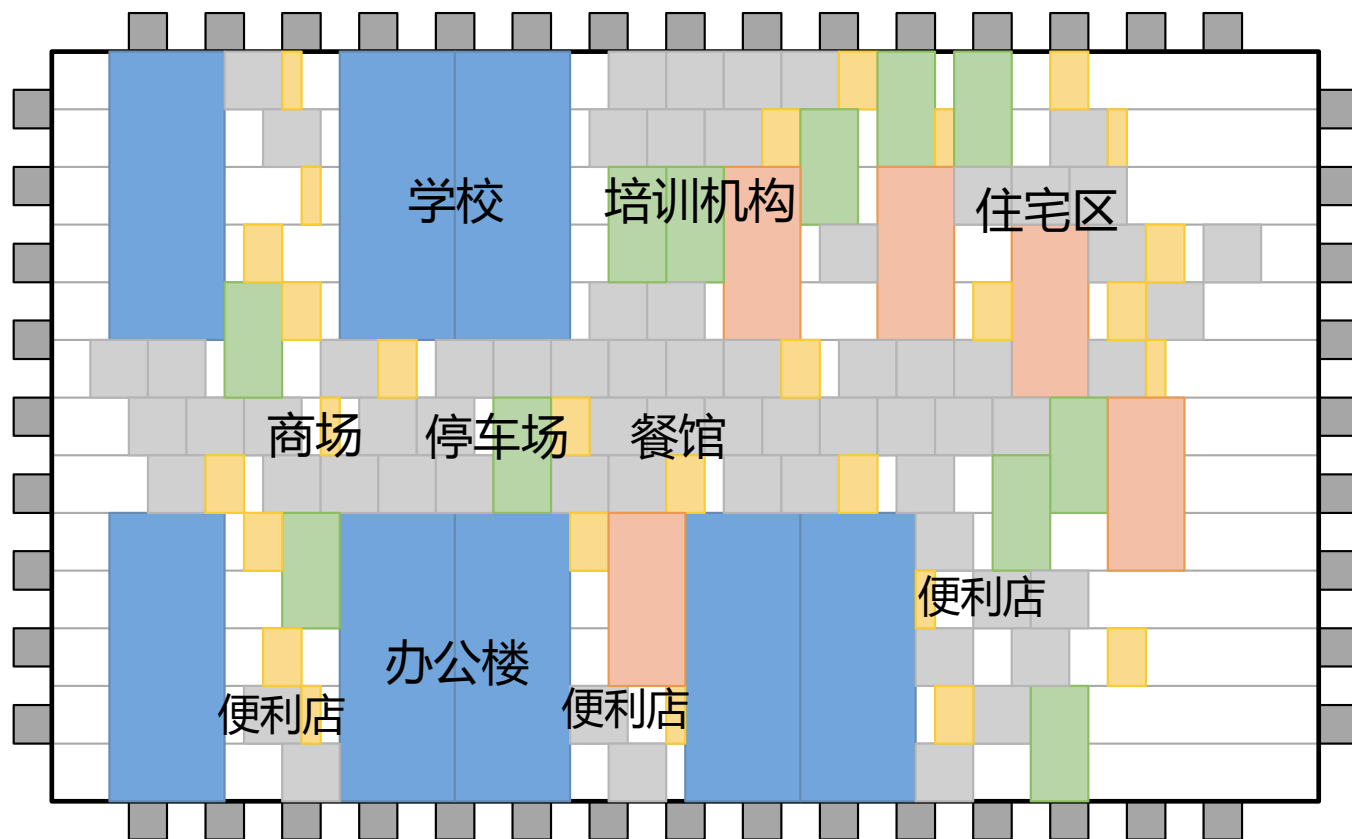
- Digital Design Flow from a IC design perspective
- Simulation
- Synthesis
- Placement
- Routing

What is Placement

Layout



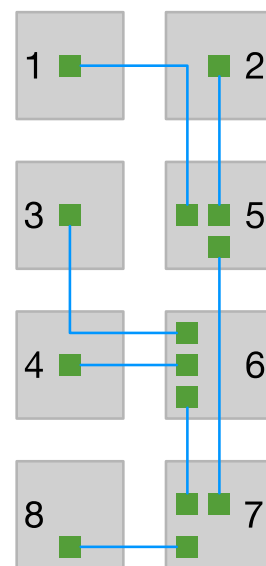
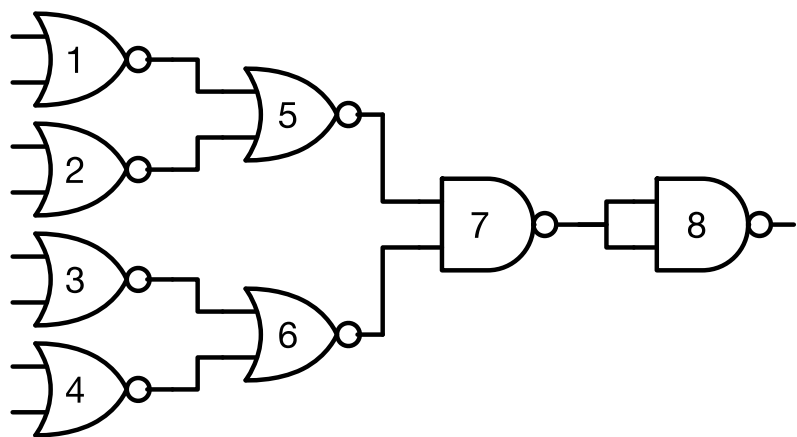
Analog to Urban Planning



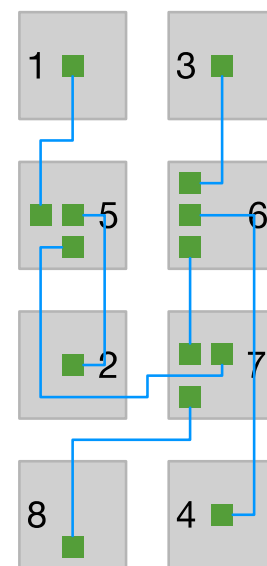
好的城市规划应使**功能关联**建筑之间的**距离**尽可能小

Metrics for Placement

- Wirelength: length of physical wires connecting cells



Wirelength=100



Wirelength=150

好的布局应使相连Cell之间的距离尽可能小

Problem Formulation for Placement

➤ Input

- Blocks (standard cells and macros) B_1, \dots, B_n
- Shapes and Pin Positions for each block B_i
- Nets N_1, \dots, N_m

➤ Output

- Coordinates (x_i, y_i) for block B_i .
- No overlaps between blocks within a fixed layout area

➤ Objective

- The total wirelength is minimized

➤ Other objectives: timing, routability, clock, buffering

How Difficult Placement is



#states: $\sim 10^{123}$



#states: $\sim 10^{360}$

Google AlphaGo
Train **40 days** using **176 GPUs**

How Difficult Placement is

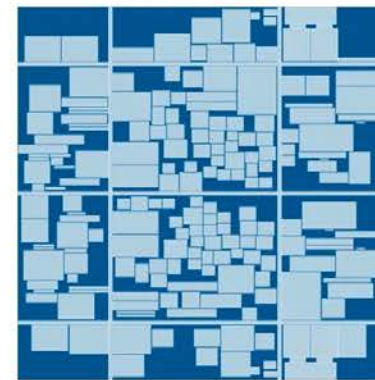
- Huge problem sizes: tens of millions of cells
- Huge solution space: larger than $1K \times 1K$ grids in a layout



#states: $\sim 10^{123}$



#states: $\sim 10^{360}$



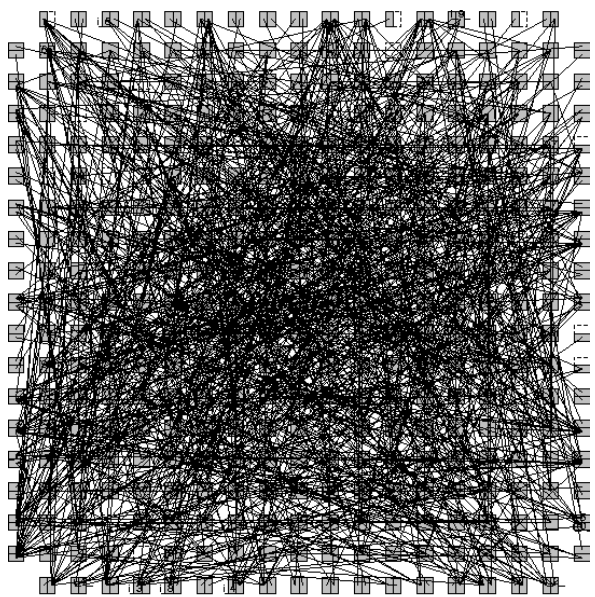
#states: $> 10^{100,000}$

Google AlphaGo
Train **40 days** using **176 GPUs**

Good Placement vs Bad Placement

- 230 cells in FPGA (design *e64* in the MCNC benchmark suite)

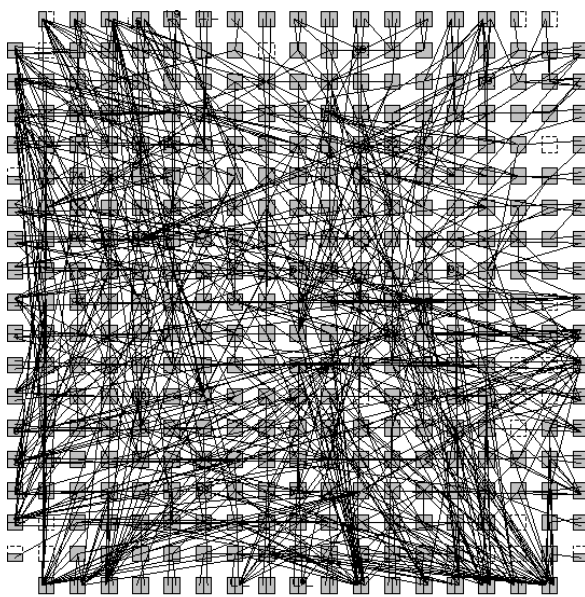
Random Initial



Initial Placement. Cost: 74.5582. Channel Factor: 100

WL = 5.47e+4

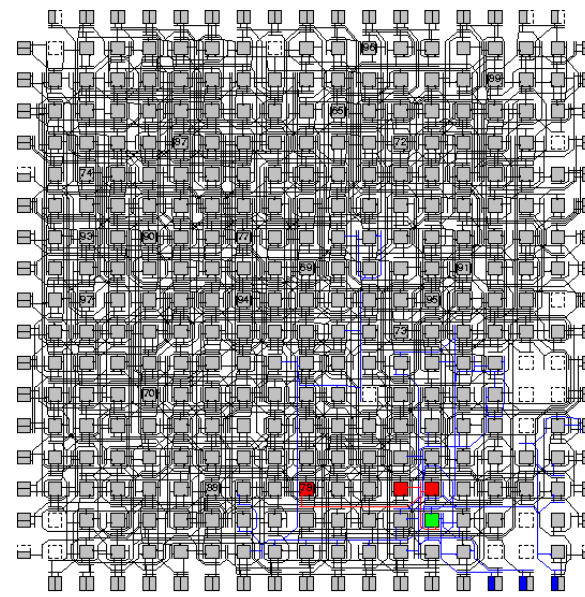
Final Solution



Final Placement. Cost: 28.5384. Channel Factor: 100

WL = 6.73e+3

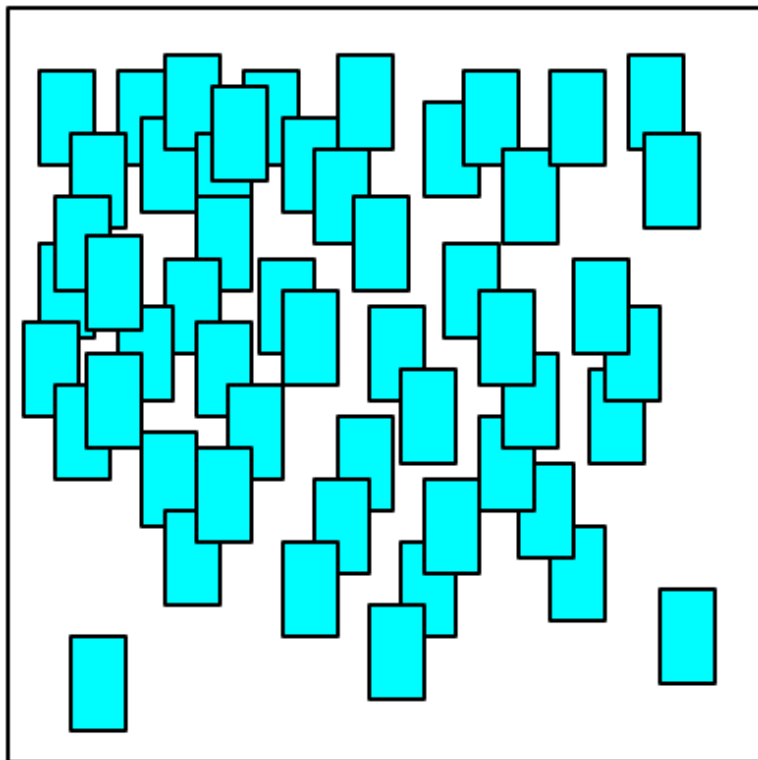
Routing Solution



Routing succeeded with a channel width factor of 7.

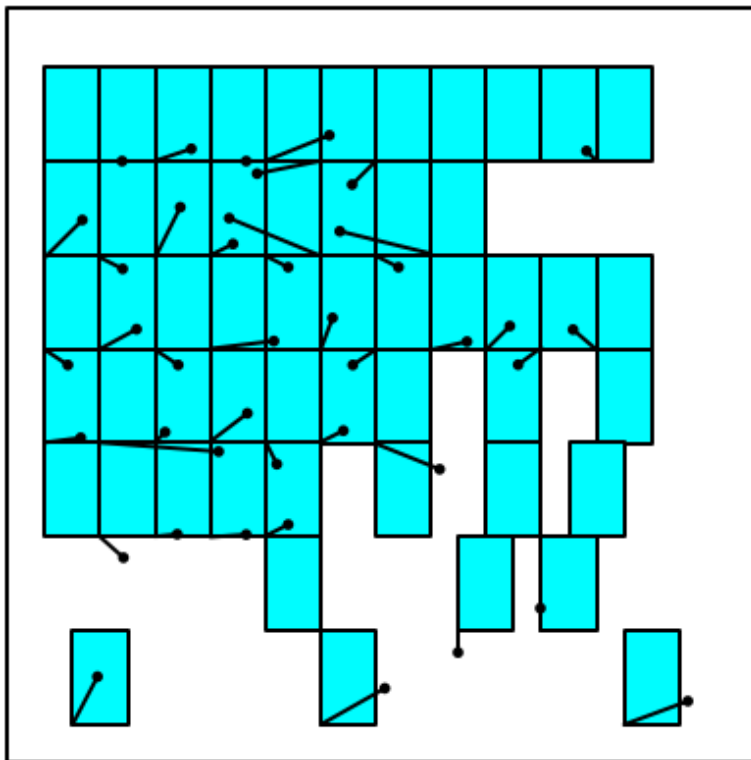
Typical Placement Flow

WL: 1.00e+6



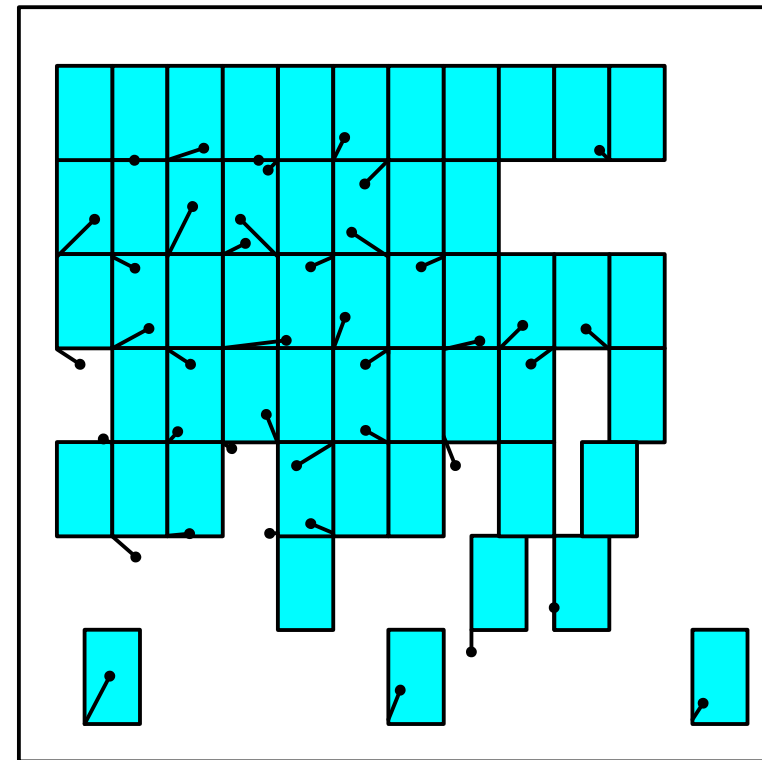
Global placement

WL: 1.05e+6



Legalization

WL: 1.02e+6



Detailed Placement

The History of Placement Algorithms

<1970-1980s	1980s-1990s
Partitioning	Simulated Annealing

Breuer

Timberwolf
VPR

Dunlop &
Kernighan

Dragon

Quadratic
Assignment

Cadence
QPlace

The History of Placement Algorithms

<1970-1980s	1980s-1990s	1990s-2010s			>2010s	
Partitioning	Simulated Annealing	Min-Cut (Multi-level)	Analytic		Analytic	
			Quadratic	Nonlinear	Quadratic	Nonlinear
Breuer	Timberwolf VPR	FengShui	GORDIAN	APlace	POLAR	ePlace RePIAce
Dunlop & Kernighan	Dragon	Capo	BonnPlace	Naylor Synopsis	SimPL ComPLx	DREAMPlace
Quadratic Assignment		Capo +Rooster	mFar	NTUplace	MAPLE	
Cadence QPlace			Kraftwerk	mPL6		
			FastPlace			
			Warp3			

Simulated Annealing

► Timberwolf package [JSSC-85, DAC-86]

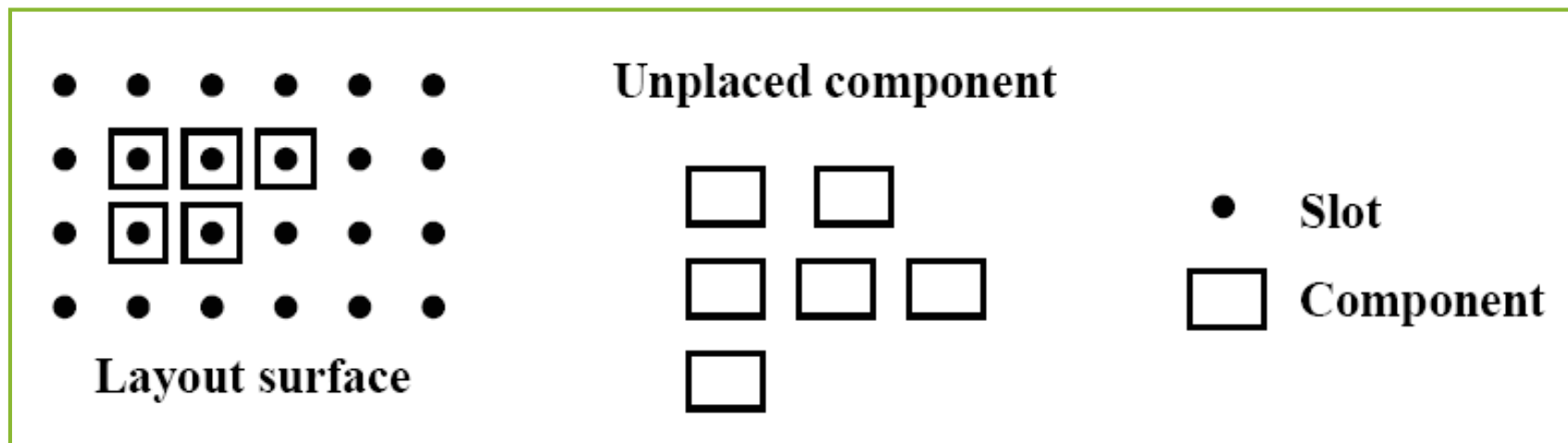
- Sechen, Carl, and Alberto Sangiovanni-Vincentelli. "The TimberWolf placement and routing package." *IEEE Journal of Solid-State Circuits* 20.2 (1985): 510-522.
- Sechen, Carl, and Alberto Sangiovanni-Vincentelli. "TimberWolf3. 2: A new standard cell placement and global routing package." *23rd ACM/IEEE Design Automation Conference*. IEEE, 1986.

► Dragon [ICCAD-00]

- Yang, Xiaojian, and Majid Sarrafzadeh. "Dragon2000: Standard-cell placement tool for large industry circuits." *IEEE/ACM International Conference on Computer Aided Design. ICCAD-2000. IEEE/ACM Digest of Technical Papers (Cat. No. 00CH37140)*. IEEE, 2000.

A Down-to-the-Earth Method

- Select unplaced components and place them in slots
- **SELECT**: choose the unplaced component that is most strongly connected to all (or any single) of the placed component
- **PLACE**: place the selected component at a slot such that a certain “cost” of the partial placement is minimized
- Simple and fast: ideal for **initial** placement



TimberWolf

➤ Stage 1

- Modules are moved between different rows as well as within the same row
- Module overlaps are allowed
- When the temperature is reduced below a certain value, stage 2 begins

➤ Stage 2

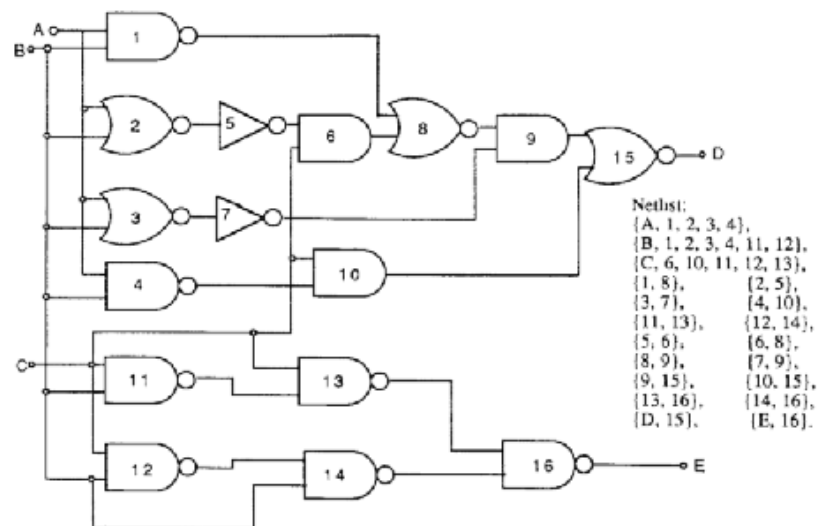
- Remove overlaps
- Annealing process continues, but only interchanges adjacent modules within the same row

Partitioning-based Approach

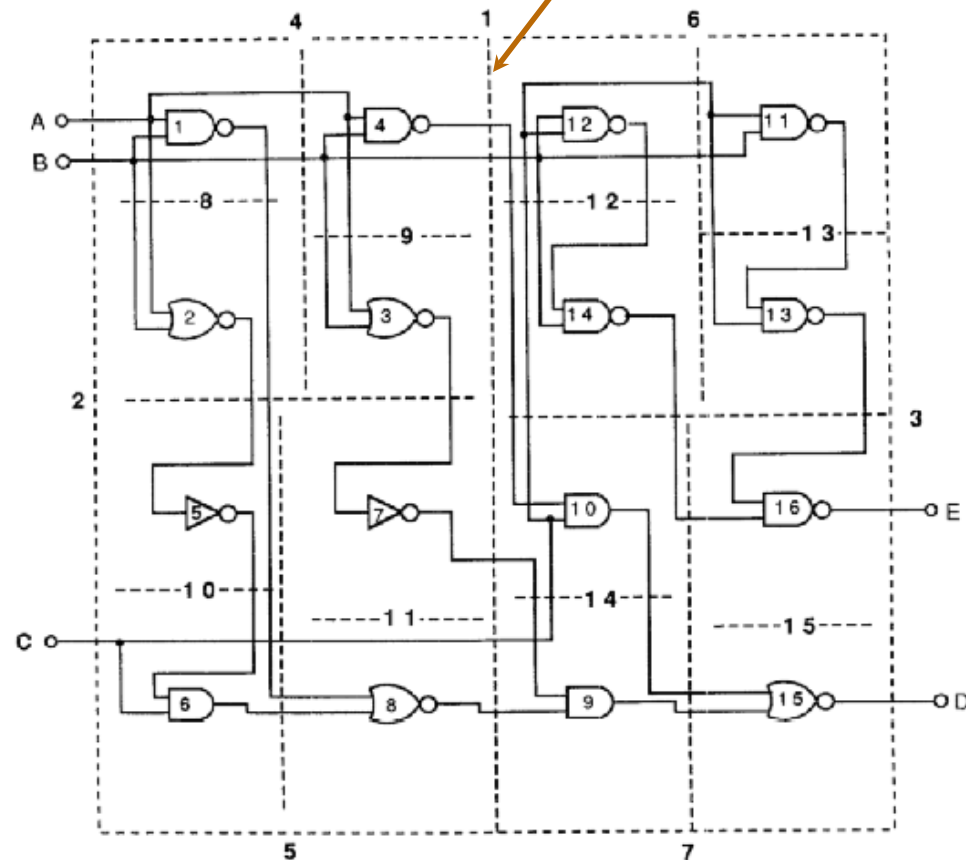
- Try to group closely connected modules together.
- Repetitively divide a circuit into sub-circuits such that the cut value is minimized.
- Also, the placement region is partitioned (by cutlines) accordingly.
- Each sub-circuit is assigned to one partition of the placement region.

Note: Also called min-cut placement approach.

An Example



Circuit

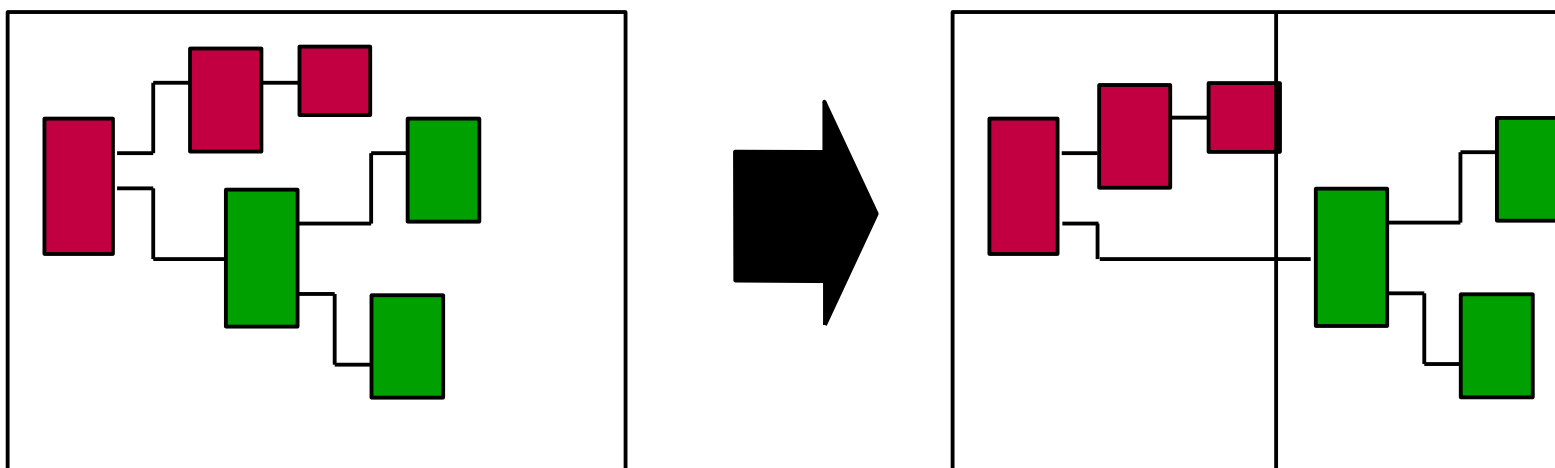


Placement

Partitioning

➤ Objective:

- Given a set of interconnected blocks, produce two sets that are of equal size, and such that the number of nets connecting the two sets is minimized.

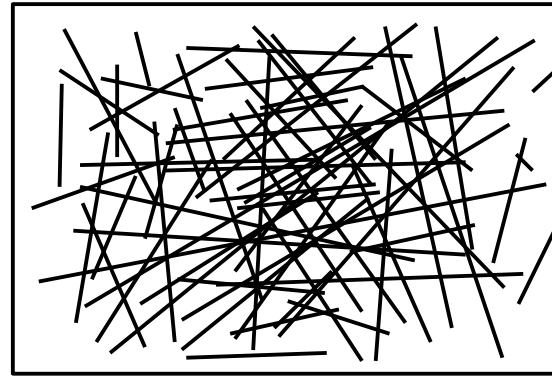


FM Partitioning

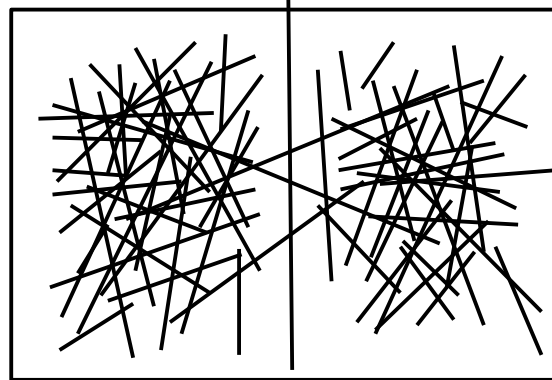
```

list_of_sets = entire_chip;
while(any_set_has_2_or_more_objects(list_of_sets))
{
    for_each_set_in(list_of_sets)
    {
        partition_it();
    }
    /* each time through this loop the number of */
    /* sets in the list doubles.                  */
}

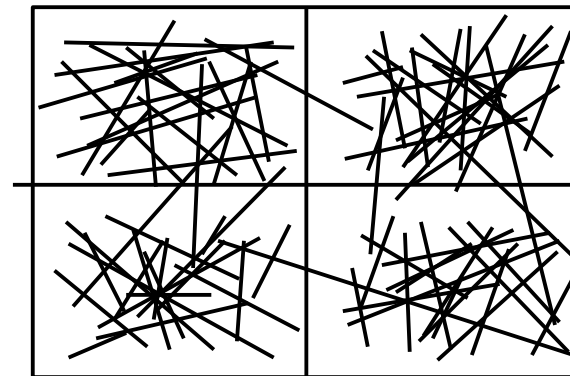
```



Initial Random Placement



After Cut 1

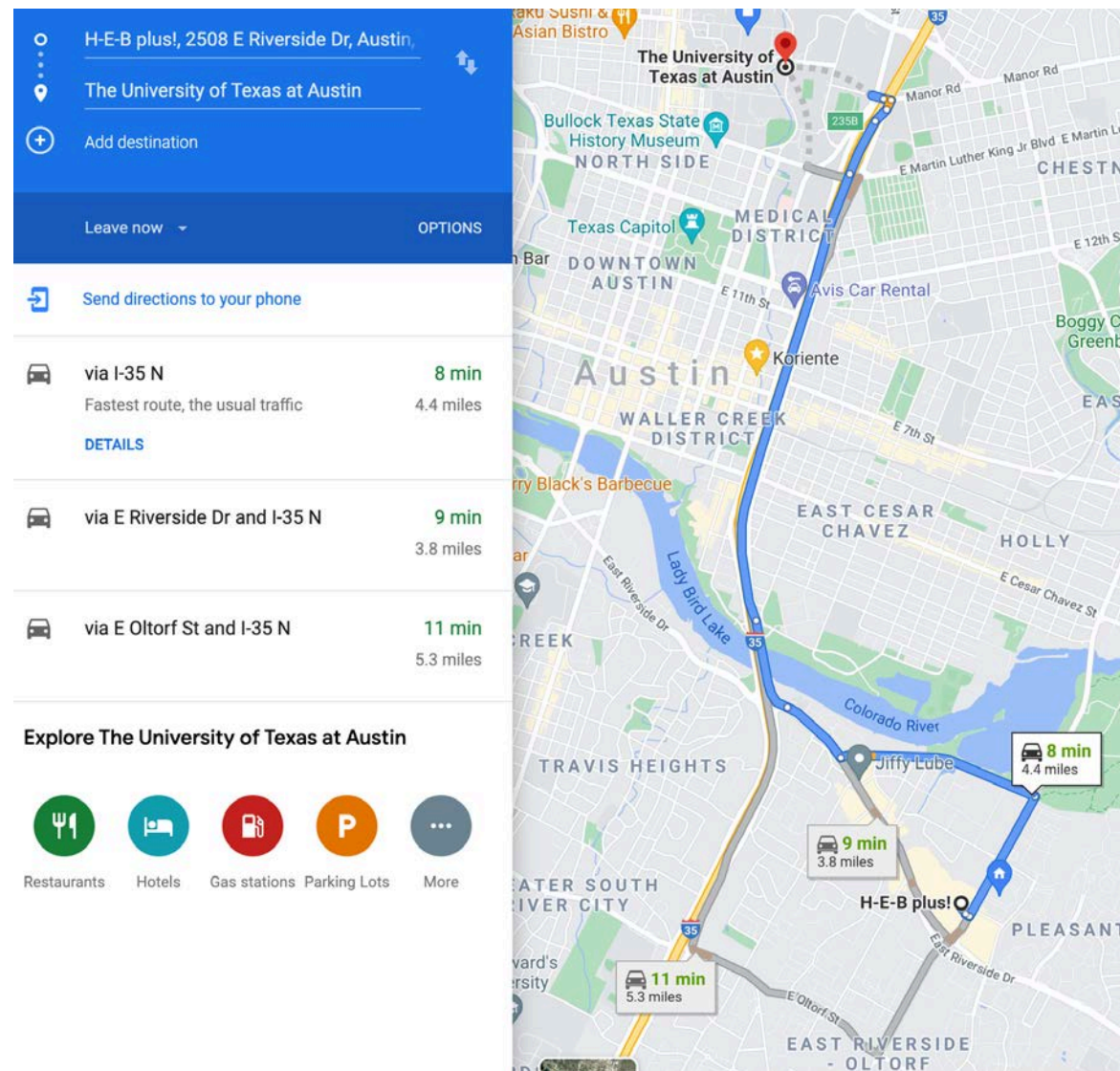


After Cut 2

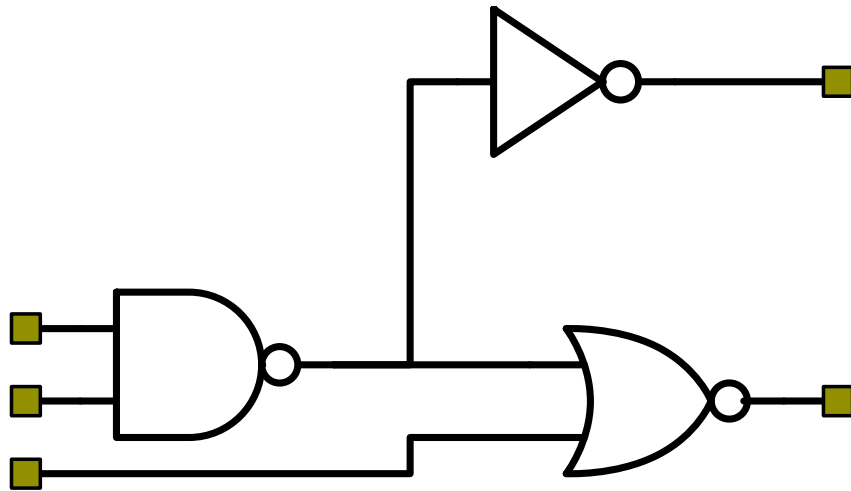
Outline

- Digital Design Flow from a IC design perspective
- Simulation
- Synthesis
- Placement
- Routing

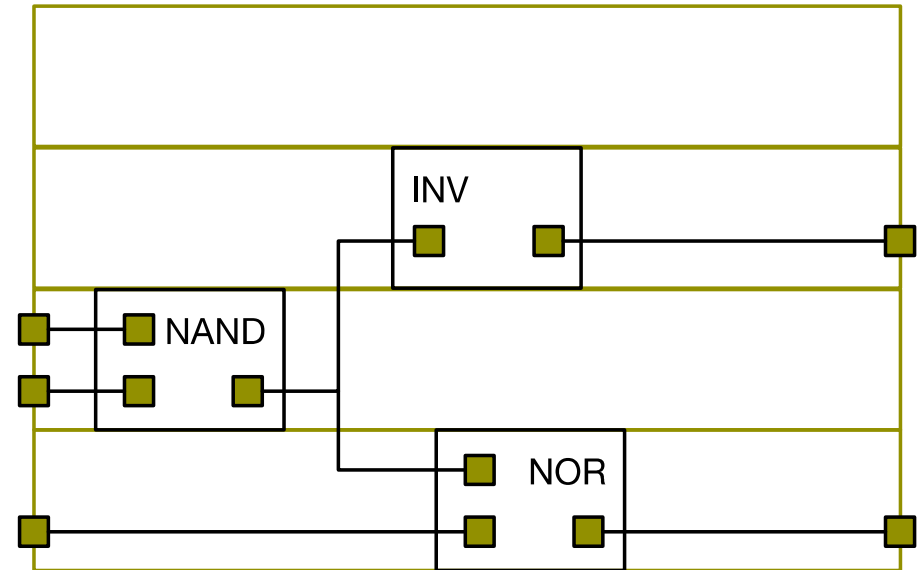
What is Routing



What is Routing

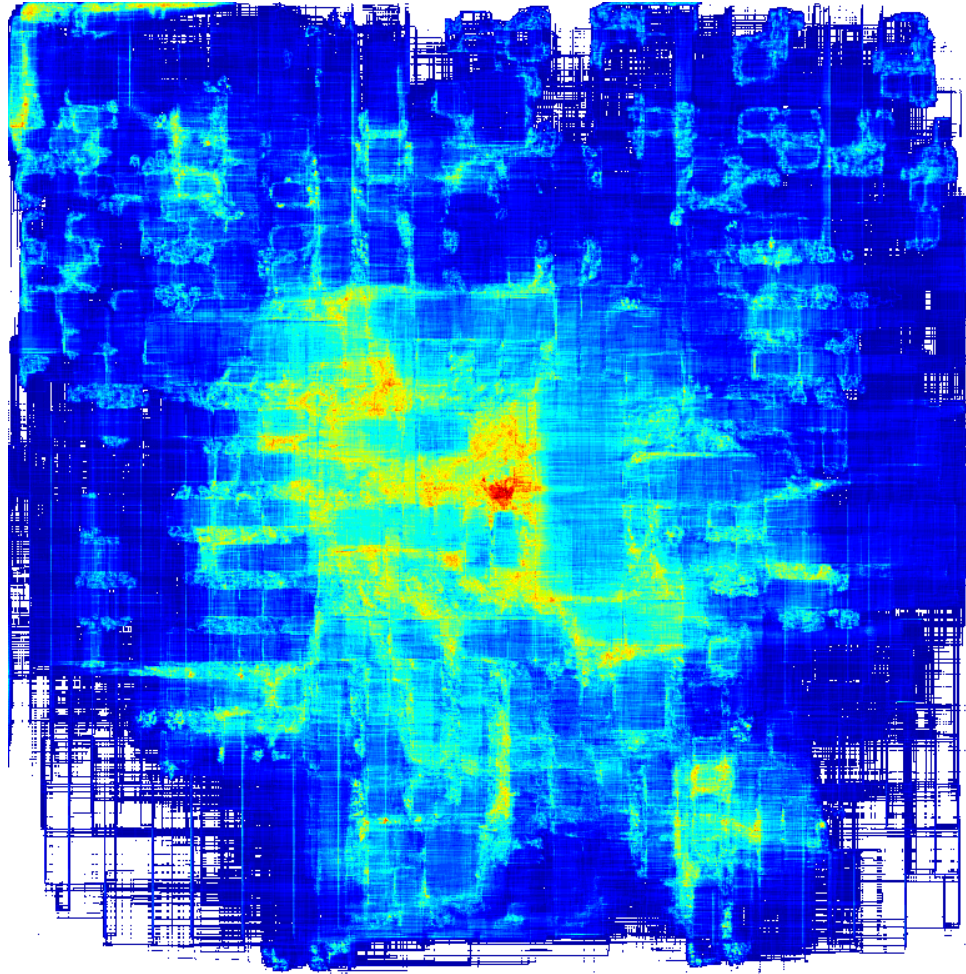


Netlist

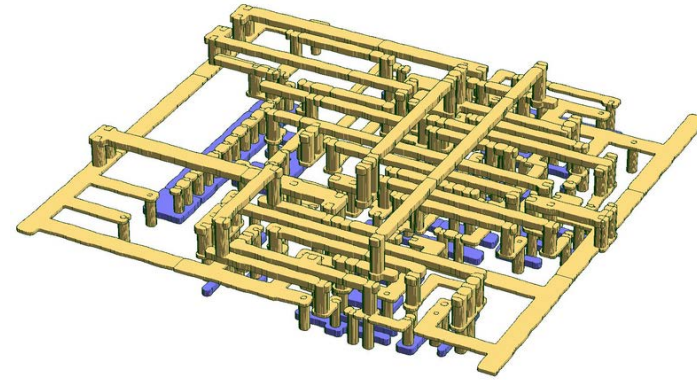


Placement and Routing

What is Routing



[Curtesy Umich]



A zoom-in 3D view

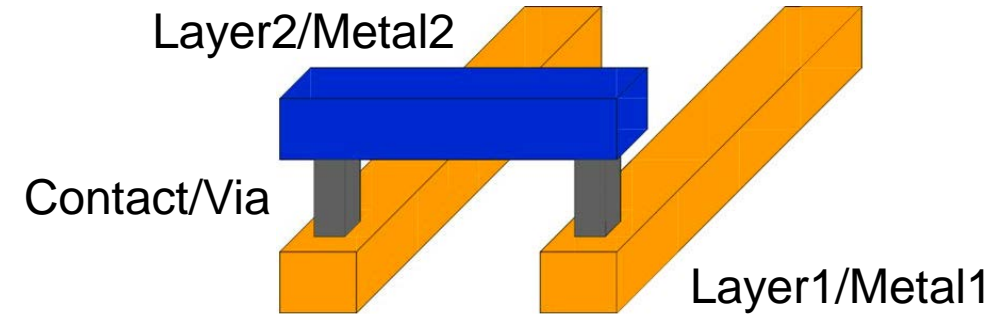
[curtesy samyzaf]

Challenging problem

- 10+ metal layers
- Millions of nets
- May be highly congested
- Minimize wirelength

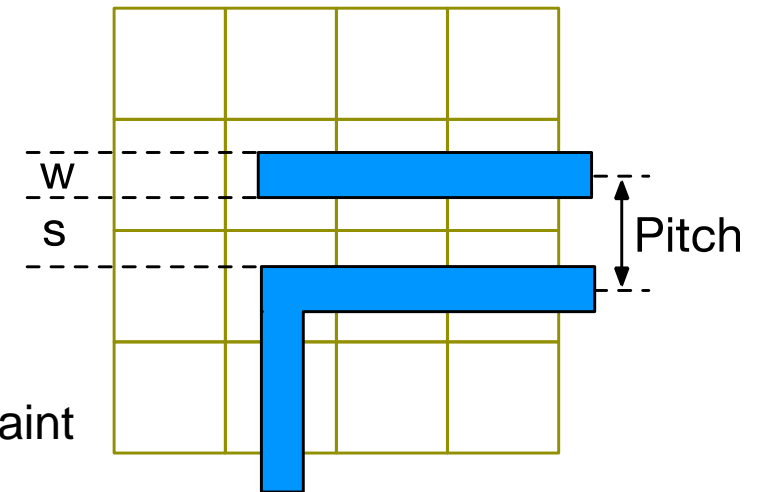
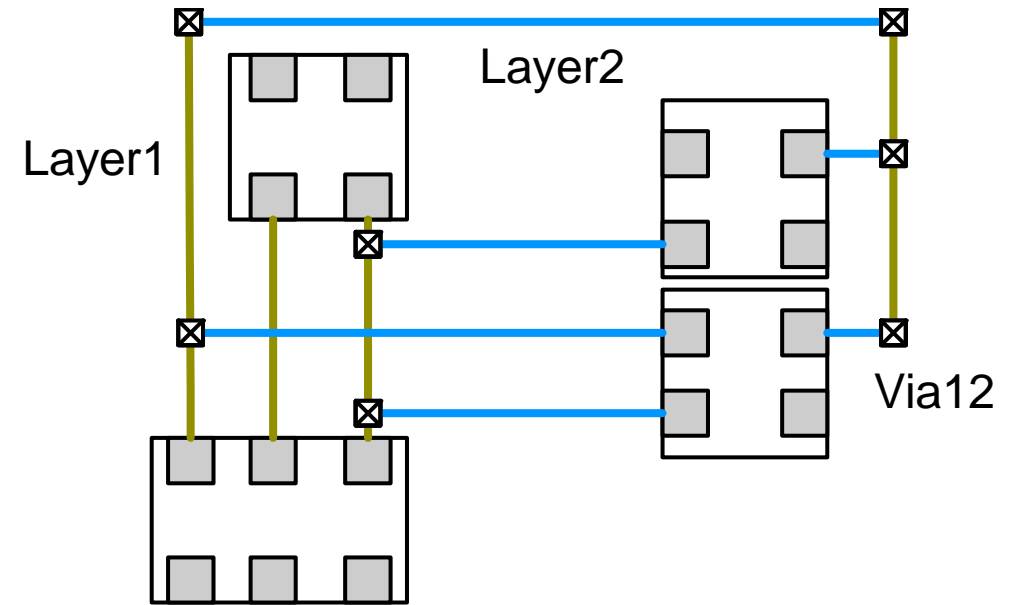
Routing Problem Formulation

- Apply it after floorplanning/placement
- Input
 - Netlist
 - Timing budget for, typically, critical nets
 - Locations of blocks and locations of pins
- Output
 - Geometric layouts of all nets
- Objective
 - Minimize the total wire length, the number of vias, or just completing all connections without increasing the chip area.
 - Each net meets its timing budget



The Routing Constraints

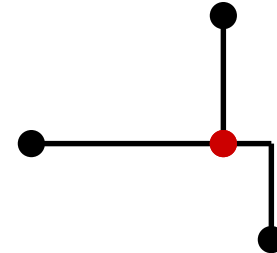
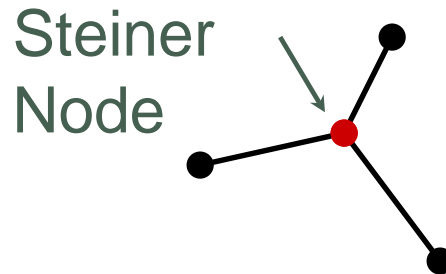
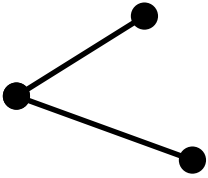
- Placement constraint
- Number of routing layers
- Delay constraint
- Meet all geometrical constraints (design rules)
- Physical/Electrical/Manufacturing constraints:
 - Crosstalk
 - Process variations, yield, or lithography issues?



Geometrical constraint

Steiner Tree

- For a multi-pin net, we can construct a spanning tree to connect all the pins together.
- But the wire length will be large.
- Better use Steiner Tree:
A tree connecting all pins and some additional nodes (Steiner nodes).
- Rectilinear Steiner Tree:
Steiner tree in which all the edges run horizontally and vertically.



Maze Routing Problem

➤ Input

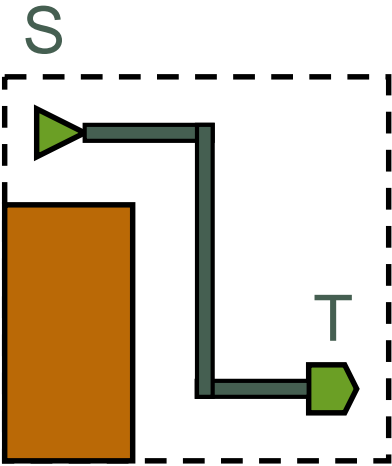
- A planar rectangular grid graph.
- Two points S and T on the graph.
- Obstacles modeled as blocked vertices.

➤ Objective

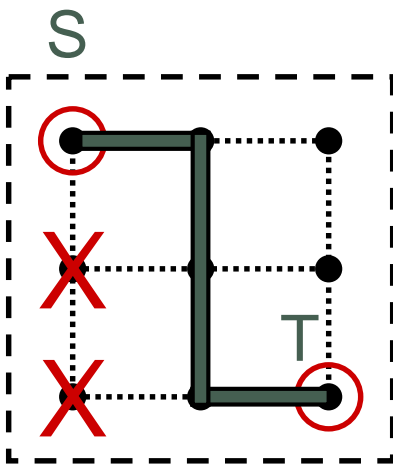
- Find the shortest path connecting S and T .

➤ This technique can be used in global or detailed routing problems.

Grid Graph



Area Routing

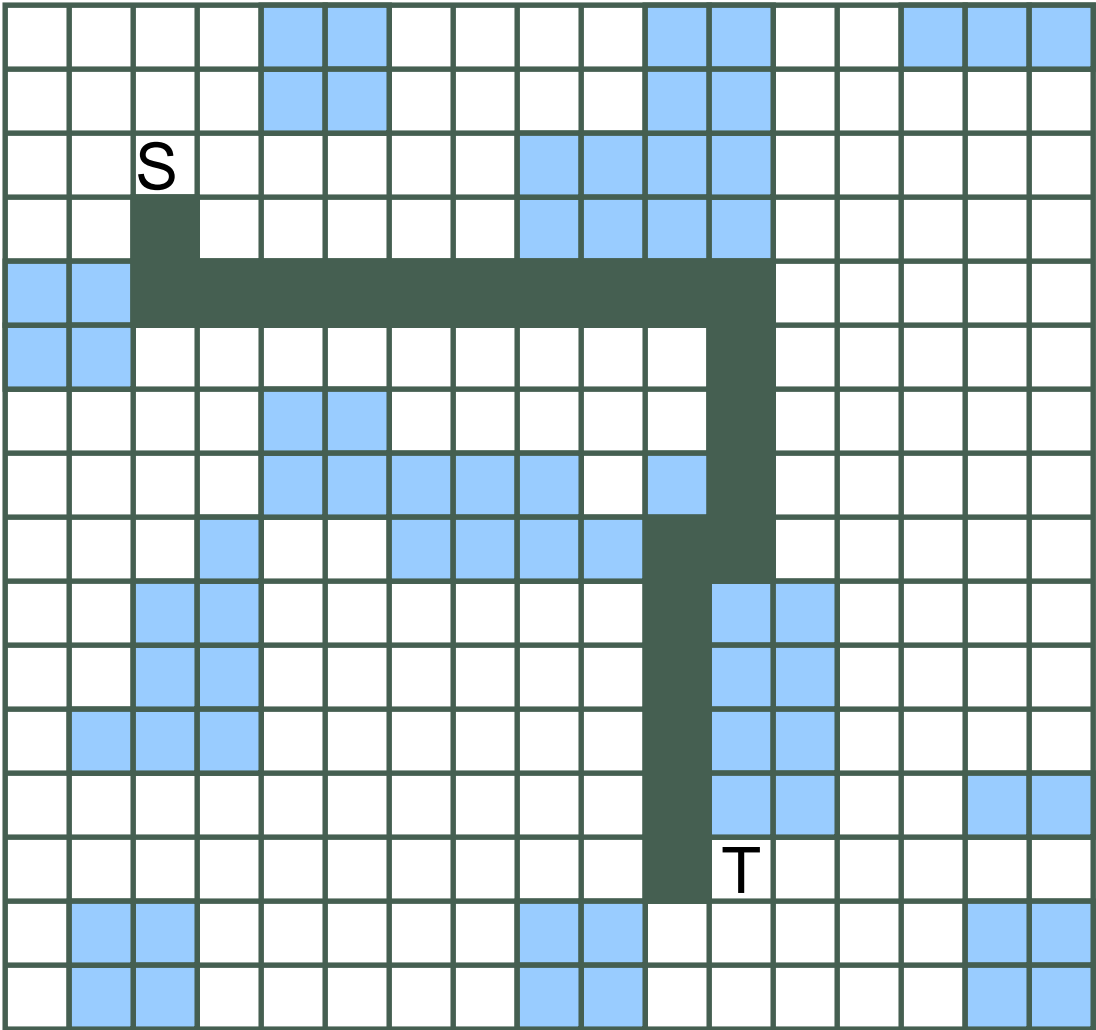


Grid Graph
(Maze)

S	✓	
X	✓	
X	✓	T

Simplified
Representation

Maze Routing



Lee's Algorithm

- Basic idea
- A Breadth-First Search (BFS) of the grid graph.
- Always find the shortest path possible.
- Consists of two phases:
 - Wave Propagation
 - Retrace

S0	1	2	3
1	2	3	
	3	4	5
5	4	5	T6

Wave Propagation

- At step k , all vertices at Manhattan-distance k from S are labeled with k .
- A Propagation List (FIFO) is used to keep track of the vertices to be considered next.

S_0			
			T

After Step 0

S_0	1	2	3
1	2	3	
	3		
			T

After Step 3

S_0	1	2	3
1	2	3	
	3	4	5
5	4	5	T_6

After Step 6

Retrace

- Trace back the actual route.
- Starting from T .
- At vertex with k , go to any vertex with label $k-1$.

S	0	1	2	3
	1	← 2	← 3	
		3	4	5
	5	4	5	← T 6

Final labeling

Concurrent Approach

- Consider all the nets simultaneously.
- Formulate as an integer program.
- Given

L_{ij} = Total wire length of T_{ij}

C_e = Capacity of edge e

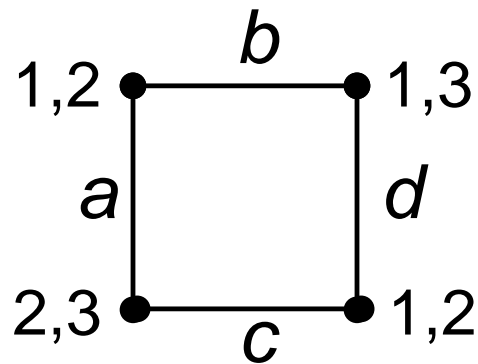
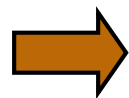
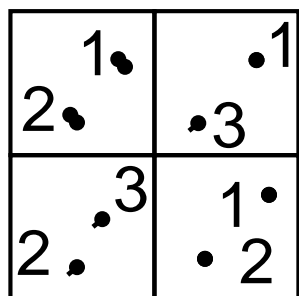
<i>Nets</i>	<i>Set of possible routing trees</i>
net 1	$T_{11}, T_{12}, \dots, T_{1k_1}$
\vdots	\vdots
net n	$T_{n1}, T_{n2}, \dots, T_{nk_n}$

- Determine variable x_{ij} s.t. $x_{ij} = 1$ if T_{ij} is used
 $x_{ij} = 0$ otherwise.

Integer Programming

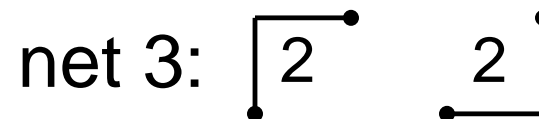
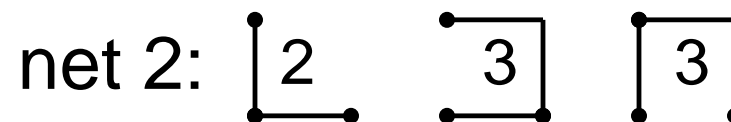
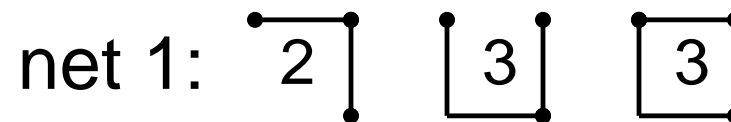
$$\begin{aligned} &\text{Min.} \sum_{i=1}^n \sum_{j=1}^{k_i} L_{ij} \times x_{ij} \\ &\text{s.t.} \quad \sum_{j=1}^{k_i} x_{ij} = 1 \quad \text{for all } i = 1, \dots, n \\ &\quad \sum_{i,j \text{ s.t. } e \in T_{ij}} x_{ij} \leq C_e \quad \text{for all edge } e \\ &\quad x_{ij} = 0 \text{ or } 1 \quad \forall i, j \end{aligned}$$

Example

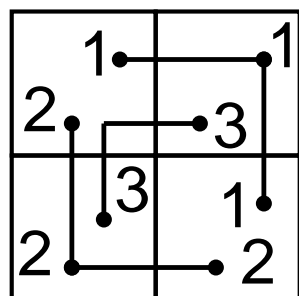


$$C_a = C_b = C_c = C_d = 2$$

Possible trees:



Solution



$$\begin{aligned} &\text{Min. } 2x_{11} + 3x_{12} + 3x_{13} + 2x_{21} + 3x_{22} + 3x_{23} + 2x_{31} + 2x_{32} \\ &\text{s.t. } \begin{cases} x_{11} + x_{12} + x_{13} = 1; \\ x_{21} + x_{22} + x_{23} = 1; \\ x_{31} + x_{32} = 1; \\ x_{ij} = 0 \text{ or } 1 \quad \forall i, j; \end{cases} \end{aligned}$$

What are the constraints for edge capacity?

$$x_{12} + x_{13} + x_{21} + x_{23} + x_{31} < C_a$$

Integer Programming Approach

- Standard techniques to solve IP.
- No net ordering. Give global optimum.
- Can be extremely slow, especially for large problems.
- To make it faster, a fewer choices of routing trees for each net can be used. May make the problem infeasible or give a bad solution.
- Determining a good set of choices of routing trees is a hard problem by itself.

Hierarchical Approach to Speed Up IP

- Large Integer Programs are difficult to solve.
- Hierarchical Approach reduces global routing to routing problems on a 2x2 grid.
- Decompose recursively in a top-down fashion.
- Those 2x2 routing problems can be solved optimally by integer programming formulation.