北京大学
PEKING UNIVERSITY

基于AI的数字系统设计

第四讲：Prompt Engineering for HDL

主讲：陶耀宇

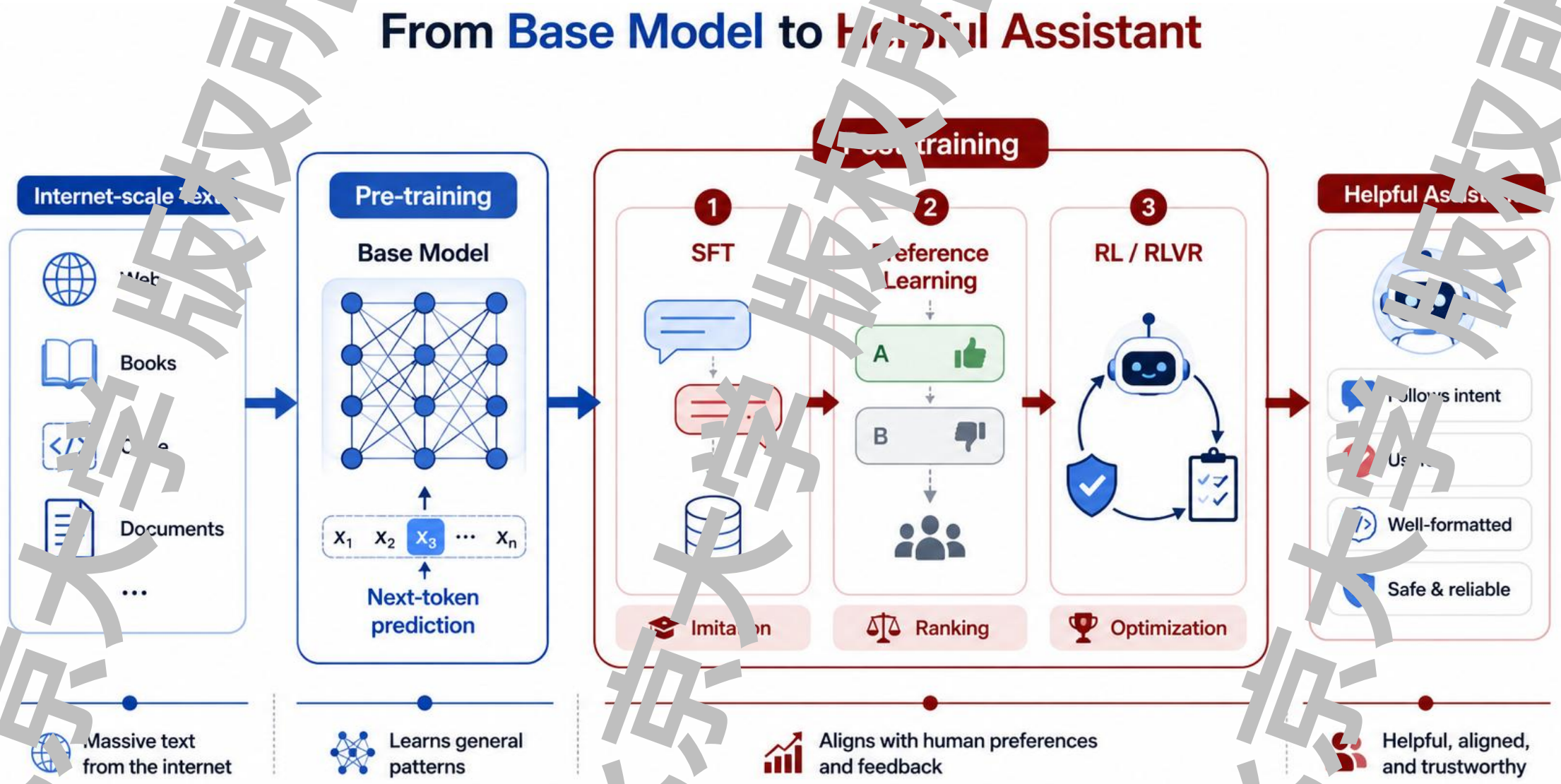
2026年秋季

北京大学
PEKING UNIVERSITY

后训练：从基座模型到助手

SFT、RLHF、GRPO 与现代强化学习共同塑造有用的助手行为。

为什么需要后训练？


From Base Model to Helpful Assistant
Internet-scale
Books
Documents
Pre-training
Base Model
x_1 x_2 x_3 ⋯ x_n
Next-token prediction
Post-training
1
SFT
2
Reference Learning
A
B
3
RL / RLVR
Imitation
Ranking
Optimization
Helpful Assistant
Follows intent
Well-formatted
Safe & reliable
Massive text from the internet
Learns general patterns
Aligns with human preferences and feedback
Helpful, aligned, and trustworthy

SFT：有监督微调

SFT 教会模型指令遵循的格式。

训练样例

指令：
Explain what a decoder-only Transformer is.

回复：
Sure! A decoder-only Transformer is ...

指令：
What directory are we currently in?

回复：
<tool_call>{"name": "pwd", "arguments": {}}
</tool_call>

SFT 教会了什么

- 回答问题，而不只是续写文本。
- 使用对话和回答的格式。
- 采用乐于助人的风格。
- 工具调用格式。

$$\mathcal{L}_{SFT} = -\sum_{t} \log p_\theta(y_t \mid x, y_{1:t-1})$$

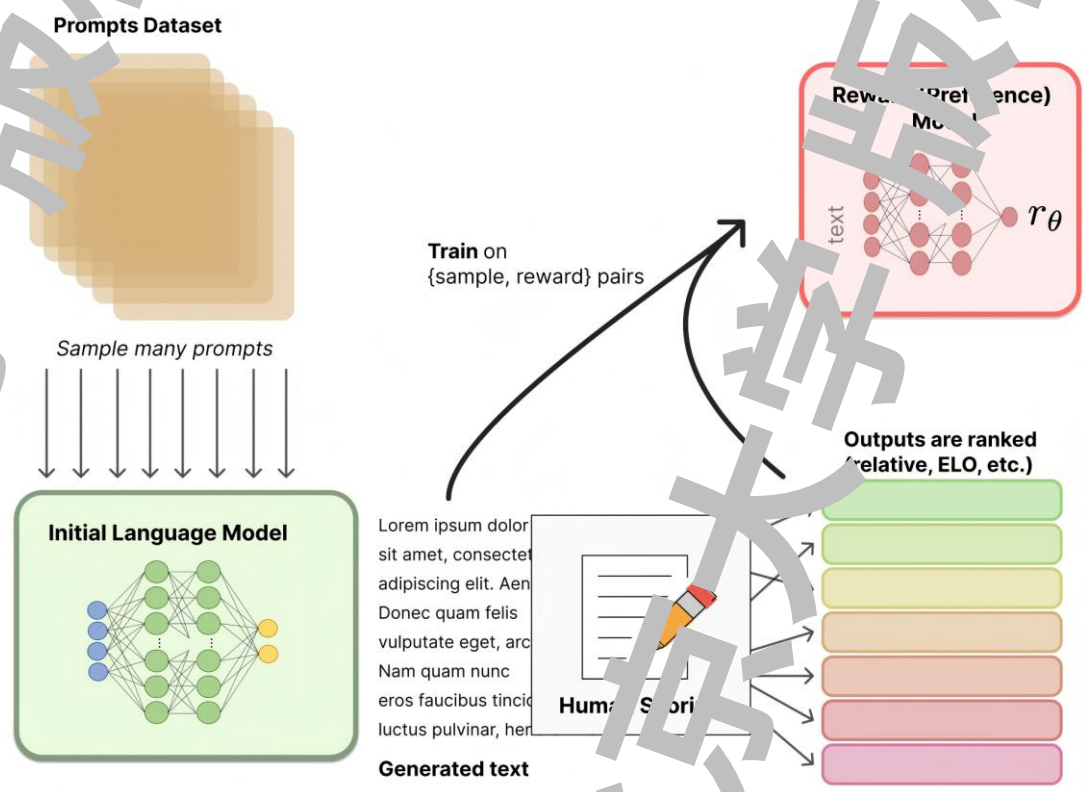
RLHF：学习人类偏好

RLHF 使模型与人类偏好对齐。

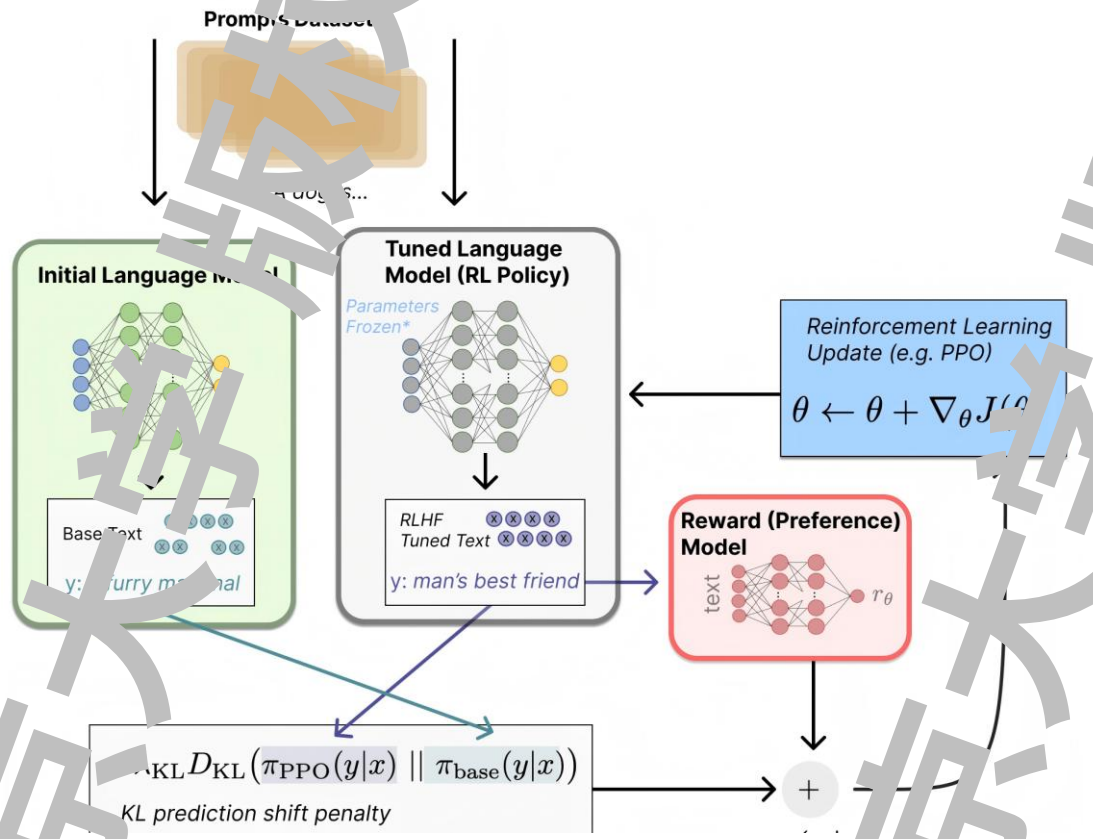
偏好数据

- 对同一提示采样多个回答。
- 由人类选出更好的回答。
- 用比较数据训练奖励模型。

奖励模型近似人类偏好。



RLHF：优化策略



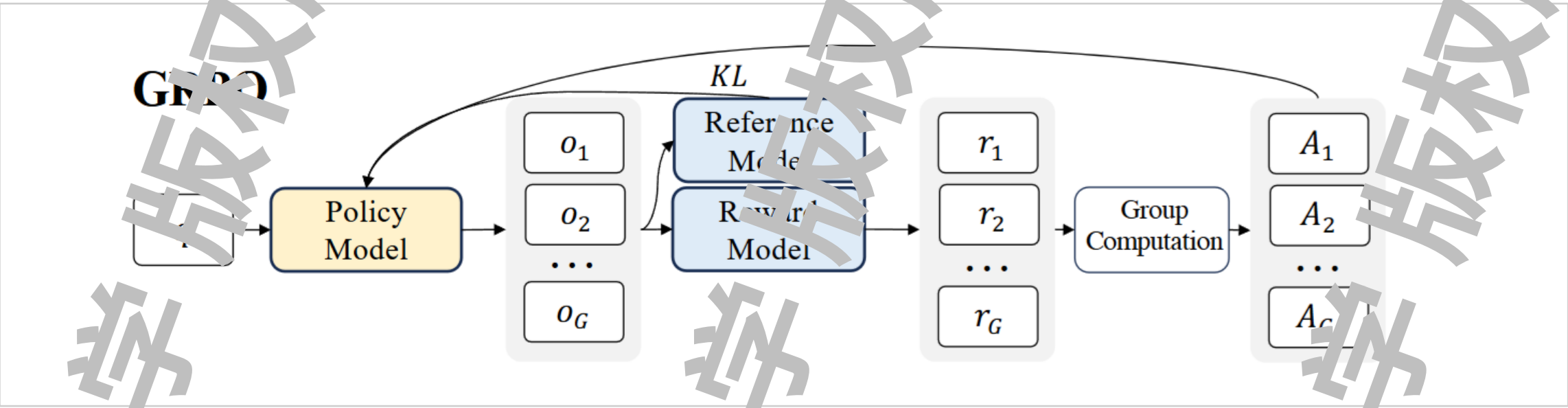
优化思路

- 策略模型生成回答。
- 奖励模型为其打分。
- 强化学习增强高奖励的行为。
- KL 惩罚让模型不偏离参考模型太远。

Kullback-Leibler Divergence Penalty

$$\max_{\theta} \mathbb{E}\left[r_{\phi}(x,y)\right] - \beta D_{KL}(\pi_{\theta} || \pi_{ref})$$

GRPO 通过强化高质量的解答来提升推理能力。



Group Relative Policy Optimization（组相对策略优化）

采样一组

对一个提示词生成多个回复。

为回复打分

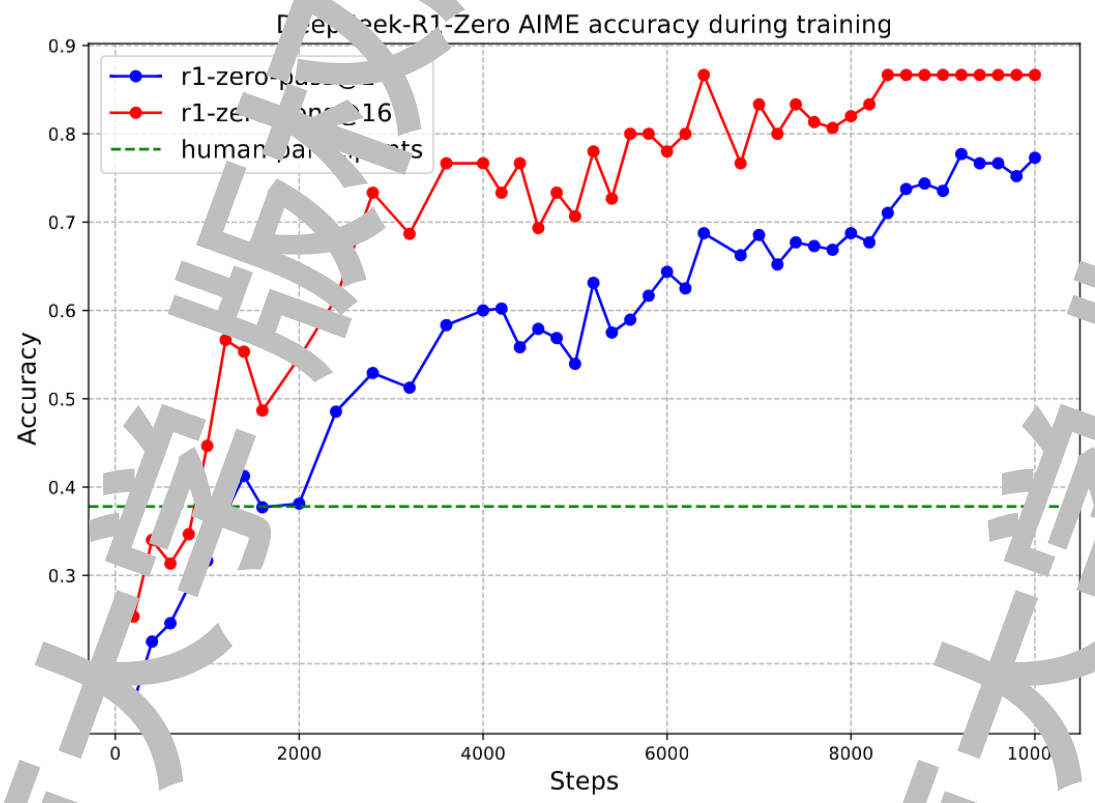
使用奖励模型、验证器或测试用例。

相对更新

提升组内更好的回复的概率。

GRPO 的创新点在于：彻底去掉了单独的价值网络（Critic），转而通过**组内相对表现**来直接估算优势。

GRPO 训练动态



GRPO 的效果

- 激活基座模型中潜在的推理能力。
- 强化学习训练过程中性能稳步提升。
- 在许多困难基准上，模型超过了人类表现。

核心信息：强化学习可以把潜在能力转化为可见的推理表现。

训练流程总结

各训练阶段逐步把**文本预测变成助手行为。**

预训练 → SFT → RLHF → GRPO / RLVR → 助手

预训练

- 语言知识
- 世界知识
- 基础代码 / 数学先验

SFT

- 指令遵循
- 对话格式
- 有帮助的回复行为

RLHF

- 使模型输出与人类偏好对齐
- 提升清晰度、有用性和回答质量

GRPO / RLVR

- 强化可验证的推理
- 数学 / 代码
- 长程问题求解

RLVR（Reinforcement Learning with Verifiable Rewards）
可验证奖励：指不需要人工主观打分，而是通过确定性规则或程序（如比对数学标准答案、运行代码单元测试、检查 JSON 格式合规性）来客观判断结果对错的奖励函数。

北京大学
PEKING UNIVERSITY

Verilog Module与接口

回顾：Spec-first 提示模板

- **把五要素固化成结构化模板，每次只替换内容**

TEMPLATE

```
【模块】<module name>, SystemVerilog-2012, 可综合
【功能】<一句话说明> + 关键行为>
【接口】
  clk, rst_n(异步低有效)
  <port> : <dir> [<W>-1:0]  // <含义>
【参数】<NAME> = <default>, 范围 <..>
【时序】输入到输出 <N> 拍; <握手协议>
【规范】
  - always_ff / always_comb, 禁止 always @*
  - 全部使用 logic; 非阻塞赋值仅用于时序块
  - 组合逻辑给默认值; 显式位宽, 禁止隐式截断
  - 不生成门控时钟 / 不用 initial 初始化
【输出】RTL + 自检 testbench (iverilog -g2012 可运行)
【验证】<用例 / 边界条件>
```

✕ 模糊提示

"帮我写一个 FIFO。"

模型必须猜：深度、位宽、同步/异步、复位极性、是否 FWFT、满/空判定、要不要 count ...每一项猜错都要返工一轮

✓ 规格化提示

"sync_fifo，DW=8 / DEPTH=16 (2 的幂)，单时钟，异步低有效复位，FWFT 读，full/empty/count 输出，always_ff 风格，附 testbench。"

一次生成即可编译仿真 → 见第 18 页

2.1 Module 声明：用“端口表”代替自然语言

- 端口名、方向、位宽、极性、优先级都写清楚，生成结果才能稳定复现

弱提示

写一个带使能的 8 位计数器模块。

强提示

模块名 `counter`，ANSI 风格端口；参数 `WIDTH = 8`。

端口：

- `clk_i`：时钟，上升沿
- `rst_ni`：异步复位，低有效
- `en_i`：计数使能
- `clr_i`：同步清零，优先级高于 `en_i`
- `cnt_o [WIDTH-1:0]`：计数值，寄存器输出
- `ovf_o`：计满且将翻转的当拍拉高，组合输出

SystemVerilog

```
module counter #(
  parameter int WIDTH = 8
)(
  input  logic             clk_i,
  input  logic             rst_ni,  // asynchronous active-low
  input  logic             en_i,
  input  logic             clr_i,   // synchronous clear
  output logic [WIDTH-1:0] cnt_o,
  output logic             ovf_o
);

  always_ff @(posedge clk_i or negedge rst_ni) begin
    if (!rst_ni)     cnt_o <= '0;
    else if (clr_i)  cnt_o <= '0;        // clr > en
    else if (en_i)   cnt_o <= cnt_o + 1'b1;
  end

  assign ovf_o = en_i & ~clr_i & (&cnt_o);

endmodule
```

2.2 时钟、复位与握手：必须显式说明的“隐含项”

项目	不说明时 LLM 的常见臆测	提示词写法示例
复位	同步/异步随机选择；高低有效不一致	“异步复位、同步释放，低有效，信号名 `rst_n`；所有寄存器初始值复位”
时钟域	默认单时钟 `posedge`，忽略跨时钟域问题	单时钟 `clk`，上升沿；来自其他时钟域的 `req_a` 先经两级同步器”
输出时序	组合输出还是寄存输出不确定，延迟拍数漂移	“所有输出由寄存器直接驱动；输入到输出固定延迟 1 拍”
握手协议	`valid` 依赖 `ready`，形成组合环路或死锁	“valid / ready 握手；`valid` 不得依赖 `ready`；握手完成前 `data` 保持不变”
控制优先级	多个控制同时有效时行为未定义	“优先级：`rst_n` > `clr` > `load` > `en`”
默认值	未选中的输出悬空或保持，出现 X 或 latch	“未命中任何分支时输出为 `'0`；不允许出现锁存器”

2.3 interface 与 modport：先冻结接口，再让 LLM 写模块

- 把一组握手信号封装为 interface，可显著减少端口连线类错误

SystemVerilog

```
interface stream_if #(parameter int DW = 32)
                    (input logic clk, rst_n);
  logic           valid, ready;
  logic [DW-1:0]  data;

  modport master (output valid, data, input  ready);
  modport slave  (input  valid, data, output ready);
endinterface

module sink (stream_if.slave s);
  assign s.ready = 1'b1;
  // ... consume s.data when s.valid & s.ready
endmodule
```

提示要点

- **先定义、后引用**：interface 单独生成并人工确认，之后在每条提示词里原样贴出
- **说明 modport 视角**：`master` 侧 `valid`/`data` 为输出，`ready` 为输入
- **禁止展开**：明确要求不要把 interface 拆回独立端口
- **交代工具链**：部分综合器对参数化 interface、interface 数组支持有限，可要求改用 `struct packed` 端口

仅使用上面给出的 `stream_if`，不要修改其定义；模块 `sink` 通过 `stream_if.slave` 接入；不要把 interface 展开成独立端口。

3.1 数据类型（一）：四态与二态，logic 优先

类型	状态	默认位宽 / 符号	在提示词中如何约定
`logic`	4 态	1 位，unsigned	RTL 信号首选，替代 `reg` 与单驱动 `wire`；可用于 `always` 与 `assign`
`wire` / `tri`	4 态	1 位，unsigned	仅用于多驱动网络与 `inout` 三态总线
`reg`	4 态	1 位，unsigned	Verilog-2001 遗留；要求 LLM 一律改写为 `logic`
`bit`	2 态	1 位，unsigned	testbench 使用；RTL 中慎用，会掩盖 X 态传播
`int` / `integer`	2 态 / 4 态	32 位，signed	`int` 用作循环变量与参数类型；避免用 `integer` 声明信号
`byte` `shortint` `longint`	2 态	8 / 16 / 64 位，signed	testbench 与 DPI；注意默认是有符号数

X 表示未知或冲突的状态，而 Z 表示高阻抗（断开或悬空）的状态

RTL 内部信号最好使用 `logic`；仅 `inout` 端口使用 `wire`；循环变量用 `int`；

赋值要求 LLM 一律写成 `assign`。

3.2 数据类型（二）：enum / struct / typedef / package

- 类型集中定义在 package 中，先贴给 LLM，再让它写模块

SystemVerilog

```
package fsm_pkg;

  typedef enum logic [1:0] {
    IDLE, LOAD, BUSY, DONE
  } state_e;

  typedef struct packed {
    logic       valid;
    logic [3:0] id;
    logic [31:0] data;
  } req_t;                    // 37 bits, MSB = valid

endpackage

// --- in the module ----
import fsm_pkg::*;

state_e state_q, state_d;
req_t   req_q;
```

1 **enum 显式底层类型**

要求"状态用 `typedef enum logic [N-1:0]`"：波形干净，lint 能查出未用状态与位宽不匹配。

2 **struct 必须 packed**

只有 `packed` 结构体才能当作向量整体赋值、作为端口与写入 FIFO；成员顺序决定位序。

3 **package 先行**

提示词里先贴 package 原文，再要求"使用其中的 `state_e` 与 `req_t`"，保证跨模块一致。

使用 `fsm_pkg` 中的 `state_e` 与 `req_t`；两段式状态机，信号名 `state_q` / `state_d`；不要用 `parameter` 或宏来定义状态。

3.3 位宽与符号：LLM 最容易写出的“静默 bug

- 不报错、能仿真，但在边界数据上出错 —— 靠提示规则 + lint 双重约束

陷阱	LLM 常见写法	正确写法	写进提示词的规则
中间结果溢出	avg = (a + b) >> 1;	avg = (9'(a) + 9'(b)) >> 1;	加法、移位之前先显式扩展位宽
有 / 无符号混算	p = a_s * b_u;	p = a_s * $signed({1'b0, b_u});	声明每个信号的符号性，跨域必须显式转换
算术右移	y = x >> 2;	y = $signed(x) >>> 2;	>>> 只对 signed 操作数做符号扩展
无位宽常量	bus = {a, 1};	bus = {a, 1'b1};	拼接中的常量必须指定位宽；填充用 '0 / '1
隐式截断	c4 = a8 + 1;	c4 = 4'(a8 + 1'b1);	有意截断用 N'() 显式表达并加注释

配套检查：Verilator `-Wall` 的 WIDTH / UNSIGNED 告警、Spyglass / Vivado lint —— 把告警原文回喂给 LLM 逐条消除

3.4 运算符（一）：按位、逻辑、归约与相等

类别	运算符	结果	LLM 常见错误 → 提示词对策
按位	`&` `|` `^` `~` `~^`	与操作数等宽	多位信号写 `if (a & b)`，不同于 `&&` → “条件必须是 1 位”
逻辑	`&&` `||` `!`	1 位	用 `!vec` 对向量取反（应为 `~vec`）→ “向量取反只用 `~`”
归约	`&a` `|a` `^a`	1 位	写成逐位循环 → “全 1 / 非零 / 奇偶判断用归约运算符”
相等	`==` `!=`	1 位，含 X 则为 X	与 `===` 混用 → “RTL 只用 `==`；`===` 仅限 testbench 与断言”
通配相等	`==?` `!=?`	1 位	用 `casex` 做模式匹配 → “用 `==?` 或 `case inside`，禁止 `casex`”
条件	`? :`	取两分支较大位宽	多层嵌套难以阅读 → “超过两层改写为 `always_comb` + `case`”

`==`：只要含 X/Z，结果即为 X（不确定）　　`===`：把 X 和 Z 当作普通状态逐位精确比较（不可综合）

优先级陷阱：`a & b == c` 实际是 `a & (b == c)` —— 规则：所有混合运算一律加括号

3.5 运算符（二）：移位、拼接、流操作与 inside

- SystemVerilog 的[illegible]运算符能让代码更短 —— 但要告诉 LLM 你[illegible]实用

SystemVerilog

```
// shift: logical vs arithmetic
y_l = x >> 2;                  // zero fill
y_a = $signed(x) >>> 2;        // sign extend

// concatenation / replication
ext = {{24{b[7]}}, b};         // sign-extend 8 -> 32
bus = {hdr, payload, crc};

// streaming: byte reverse
swapped = {<<8{data}};

// set membership
if (opcode inside {4'h1, 4'h3, [4'h8:4'hB]})
  hit = 1'b1;

// indexed part-select: variable base, const width
byte o = word[idx*8 +: 8];
```

+: 定宽切片

LLM 常写出 `word[hi:lo]` 这种两端都是变量的范围（非法）。规则：可变位置切片一律用 `[base +: W]`。

Verilog 编译器在编译时必须知道切片的确切宽度

复制运算符做扩展

符号扩展 / 零扩展用 `{{N{msb}}, x}`，不要手写 `?:` 判断符号位。

inside 取代长比较

一串 `==` 加 `||` 改写为 `inside {…}`，支持区间；也可用于 `case inside`。

流操作符先确认

`{<<8{data}}` 简洁，但需先确认综合器支持；否则要求用 `for` 循环等价实现。

3.6 过程块与赋值：always_ff / always_comb 的分工

- 组合块首行给所有输出赋默认值 —— 从结构上杜绝 latch

过程块	赋值	工具会帮你检查什么
always_ff	<=	只允许可综合时序逻辑；禁止多块驱动
always_comb	=	自动敏感表；推断出 latch 即告警
always_latch	=	显式声明 latch；别处出现即为错误

状态机采用两段式：

- always_ff 只做 state_q <= state_d
- always_comb 首行写 state_d = state_q 及所有输出的默认值，再用 unique case 覆盖

不推荐使用 always @(*) 与 always @(posedge clk)

SystemVerilog

```
always_ff @(posedge clk or negedge rst_n)
  if (!rst_n) state_q <= IDLE;
  else        state_q <= state_d;

always_comb begin
  state_d = state_q;          // default: hold
  done_o  = 1'b0;             // default output

  unique case (state_q)
    IDLE: if (start_i) state_d = BUSY;
    BUSY: if (last_i)  state_d = DONE;
    DONE: begin
      done_o  = 1'b1;
      state_d = IDLE;
    end
    default: state_d = IDLE;
  endcase
end
```

4.1 参数化：parameter / localparam / $clog2

- 让 LLM 写“可复用 IP”：派生量自动推导，非法参数在 elaboration 期报错

SystemVerilog

```
module sync_fifo #(
  parameter int DEPTH  = 16,
  parameter type data_t = logic [7:0],
  localparam int AW    = $clog2(DEPTH)
)(
  input  logic clk, rst_n,
  input  logic wr_en, rd_en,
  input  data_t wr_data,
  output data_t rd_data,
  output logic  full, empty
);
  data_t mem [DEPTH];
  logic [AW:0] wr_ptr, rd_ptr;   // extra MSB: full/empty

  // elaboration-time parameter check
  if (DEPTH < 2 || (DEPTH & (DEPTH - 1)))
    $error("DEPTH must be a power of 2");
  // ...
endmodule
```

写进提示词的参数化规则

- **可配置量**用 `parameter int`，给出默认值、取值范围与典型值
- **派生量**一律 `localparam`，位宽用 `$clog2` 推导，禁止魔数
- **数据通路类型**用 `parameter type`，同一份 FIFO 可存 `req_t`
- **合法性检查**：elaboration 期 `$error` / `$fatal`
- **宏定义**只用于全局开关，不用于模块内常量

`DEPTH` 取值 4 – 1024 且为 2 的幂，默认 16；所有位宽由参数推导；非法参数在 elaboration 期报错

4.2 generate：for / if / case 三种结构化生成

- generate 在 elaboration 期展开 —— 条件只能是参数或常量，每个块都要命名

```
generate
// for-generate: N-stage pipeline
for (genvar i = 0; i < STAGES; i++) begin : g_stage
  always_ff @(posedge clk)
    pipe[i+1] <= pipe[i];
end

// if-generate: pick an implementation
if (USE_DSP) begin : g_dsp
  mult_dsp u_mul (.a, .b, .p);
end else begin : g_lut
  mult_lut u_mul (.a, .b, .p);
end

// case-generate
case (MODE)
  0:       begin : g_bypass assign y = x; end
  default: begin : g_reg
    always_ff @(posedge clk) y <= x;
  end
endcase
endgenerate
```

1 结构选择用 generate

“实现方式由参数 `USE_DSP` 决定”应写成 if-generate，而不是运行时 `if` —— 后者会把两份硬件都综合出来。

2 块必须命名

统一 `g_` 前缀：层次路径稳定（`g_stage[3]...`），方便写时序约束、看波形。

3 genvar ≠ int

`genvar` 只能出现在 generate-for；`always` 块内的循环变量用 `int`。

用 for-generate 例化 `N` 个 `u_cell`，块名 `g_cell`，不要手工展开。

4.3 各类循环一览：哪些能综合，哪些只属于 testbench

循环	语法示例	可综合性	典型用途
for	for (int i = 0; i < N; i++)	可综合（边界为常量）	位操作、优先编码、数组复位
foreach	foreach (mem[i]) mem[i] <= '0;	可综合（定长数组）	遍历数组，免写边界、避免越界
generate-for	for (genvar i = 0; …) begin : g_x	elaboration 期展开	复制模块实例或 always 块
while / do-while	while (!done) @(posedge clk);	一般不可综合	testbench 中等待条件成立
repeat	repeat (8) @(posedge clk);	含时序控制时不可综合	testbench 中等待 N 拍
forever	forever #5 clk = ~clk;	不可综合	时钟生成、监视器进程

RTL 中只允许 **for**、**foreach** 与 generate-for，且**循环边界必须是参数或常量**；**while** / **repeat** / **forever** 只能出现在 testbench 文件中。

给RTL和tb的prompt中要显式的分开提示

4.4 可综合循环的提示要点：循环 = 空间展开

- 每一次迭代都对应一份硬件；没有“执行到一半跳出”

SystemVerilog

```
// priority encoder: lowest set bit wins
always_comb begin
  idx_o   = '0;
  valid_o = 1'b0;
  for (int i = N-1; i >= 0; i--) begin
    if (req_i[i]) begin
      idx_o   = IDX_W'(i);    // last write wins
      valid_o = 1'b1;
    end
  end
end
```

弱提示

用循环找到 `req` 中第一个为 1 的位。 → 可能得到 `while` `break` 或用状态逐位扫描的多周期实现

强提示

纯组合逻辑，单周期给出结果；在 `always_comb` 内使用 `for (int i ...)`，循环边界为参数 `N`；不使用 `break` / `while`；最低位优先；无请求时 `valid_o = 0`、`idx_o = 0`。

1 边界静态可定

循环次数在 elaboration 期即可确定：**参数、常量、定长数组的维度。**

2 无时序控制

循环体内不出现 `@`、`#`、`wait`；**需要跨周期的行为改用计数器 + 状态机。**

3 关注逻辑深度

N 次迭代 = N 级串联逻辑；N 较大时提示 LLM 改为树形结构或插入流水线。

4.5 调用方式：function / task / 模块例化

- 三种“复用”手段对应三种硬件含义 —— 在提示词里指定用哪一种

function 组合逻辑封装

```
function automatic logic [7:0]
    gray2bin (input logic [7:0] g);
  for (int i = 0; i < 8; i++)
    gray2bin[i] = ^(g >> i);
endfunction
```

- 零延时，可综合，展开为组合逻辑
- 必须 `automatic`，不访问外部变量
- 不含 `@` / `#` 等时序控制

task 过程封装（TB 为主）

```
task automatic send_byte(input byte d);
  @(posedge clk);
  valid <= 1'b1; data <= d;
  @(posedge clk iff ready);
  valid <= 1'b0;
endtask
```

- 可含时序控制，可有多个输出
- 用于 testbench 的驱动与检查
- RTL 中避免使用 task

模块例化 放置一块电路

```
sync_fifo #(
  .DEPTH  (32),
  .data_t (req_t)
) u_fifo (
  .clk, .rst_n,          // .name
  .wr_data (req),
  .rd_data (req_q)
);
```

- 参数与端口一律按名称连接
- `.name` / `.*` 要求同名同位宽
- 实例名统一 `u_` 前缀

可复用的组合运算封装为 `automatic function`；例化一律按名称连接并显式覆盖参数，禁止按位置连接；不要在 RTL 中使用 `task`。

5.1 验证闭环：让 LLM 同时产出 Testbench 与 SVA

- 把提示词里的时序要求翻译成断言 —— 需求从此变成“可执行的检查”

SystemVerilog Assertions

```
// handshake: data must hold until accepted
property p_hold;
  @(posedge clk) disable iff (!rst_n)
  valid && !ready |=> valid && $stable(data);
endproperty
a_hold: assert property (p_hold);

// FIFO can never be full and empty together
a_excl: assert property (
  @(posedge clk) disable iff (!rst_n)
  !(full && empty));
```

错误模板

以下是 Verilator 的原始输出，请只修改相关行并说明根因：

```
%Warning-LATCH: fifo.sv:42: Latch inferred for signal
'rd_data'
```

要求：

- 不改动端口与其他逻辑
- 指出违反了哪条编码规范（规则编号）
- 以 diff 形式给出修改

1 自检 Testbench

激励源 + 参考模型 + 自动比对；结束时打印 PASS / FAIL，便于脚本判定。

2 断言即规格

每条时序要求对应一条 SVA；仿真与形式验证共用同一份断言。

3 工具当裁判

Verilator `--lint-only`、Icarus、Yosys 即可搭建零成本的检查流水线。

5.2 提示工程 Checklist 与总结

维度	提[illegible]词中必须写明
Module	[illegible]口表、时钟 / 复位极性、输出是否寄存、握手协议与控制优先级
数据类型	`logic` 为主；`enum` / `struct packed` 集中放入 package；每个信号的符号性
运算符	混合表达式加括号；`>>>` 配 signed；常量带位宽；`+:` 定宽[illegible]片
过程块	`always_ff` + `<=`；`always_comb` + `=` + 块首默认值；`case` 带 `default`
参数化	`parameter` / `localparam` / `$clog2`；取值范围[illegible]ation 期检查
循环	边界静态；generate 块命名；`while` / `repeat` / `forever` 仅限 testbench
调用	`automatic function`；按名称例化；RTL 不使用 `task`
验证	同步生成 testbench 与 SVA；lint / 仿真日志[illegible]回喂

1 规范先行

把 SystemVerilog 编码规范固化为带编号的 system prompt

2 接口精确

用端口表、package 与代码片段消除自然语言的歧义

3 闭环迭代

让 EDA 工具[illegible]用报错原文驱动 LLM 小步修正

北京大学
PEKING UNIVERSITY

AI辅助组合逻辑设计

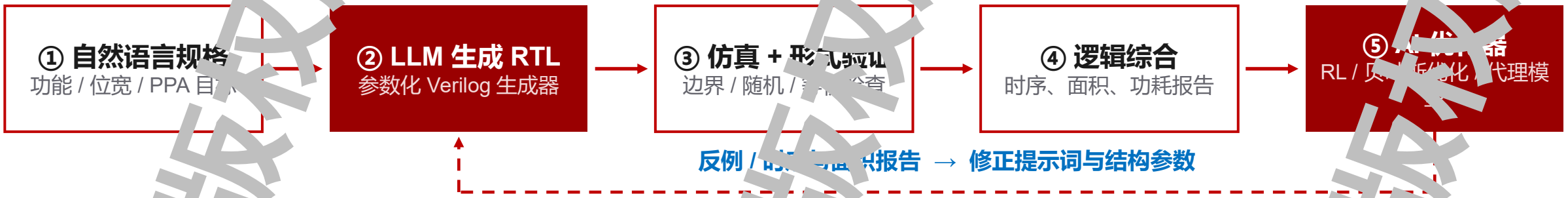
组合逻辑分类与 AI 辅助要点

- 结构决定 PPA 上限，AI 在“生成—验证—优化”三个环节提升效率

类别	典型结构	延迟	面积	AI 辅助重点
静态 CMOS 门	NAND / NOR / AOI / OAI	逻辑努力模型	晶体管数	等价检查、尺寸建议
Ripple-Carry	全加器级联	O(N)	O(N)	参数化生成、最坏向量
Carry-Select	双路预计算 + MUX	O(√N)	≈ 2× RCA	分块方案搜索
Carry-Lookahead	分组 (G, P) + LCU	O(log N)	O(N)	扇入约束下展开
PGK / 前缀	预处理-前缀-后处理	O(log N)	由前缀图决定	前缀图 RL 搜索
Kogge-Stone	跨度倍增，扇出 2	log N 级	$N \cdot \log_2 N$ 单元	布线感知、稀疏化
Sklansky	分治，高扇出	$\log_2 N$ 级	$(N/2) \cdot \log_2 N$	尺寸与扇出优化
阵列乘法器	AND 阵列 + 加法阵列	O(N)	$O(N^2)$	穷举 / 形式验证
Booth 乘法器	Radix-4 重编码	部分积减半	编码器开销	符号与 +1 纠错
Wallace 树	3:2 / 4:2 压缩树	O(log N)	$O(N^2)$	压缩器树 RL 优化
移位器	对数 / 漏斗	$\log_2 N$ 级	$N \cdot \log_2 N$ MUX	统一漏斗模板
编码 / 译码器	预译码、树形优先编码	O(log N)	O(N) / $O(2^n)$	无效态、树形化
比较器	相等 / 减法 / 树形	O(log N)	O(N)	符号与边界、资源共享

1.1 AI辅助组合逻辑设计流程

- 从“人工设计”到“AI生成 + 验证 + 优化”的闭环



生成 Generation

- LLM 将规格翻译为参数化 RTL / 生成器代码
- 显式约束结构：加法器、移位器、译码
- 输出须可综合、无锁存器、位宽明确

验证 Verification

- 仿真：边界 + 随机 + 小位宽穷举
- 形式等价：与 a+b、a×b 等参考模型比对
- AI 可写测试平台与断言，结论由工具给出

优化 Optimization

- 结构参数化后形成离散设计空间
- RL / 贝叶斯优化搜索 PPA Pareto 前沿
- 代理模型预测 PPA，减少真实综合次数

1.2 组合逻辑分类总览

- 按功能把典型组合逻辑分为五大类，共 13 种典型结构

门级逻辑	加法器	乘法器	移位器	编码 · 译码 · 比较
• 静态 CMOS 反相器 • NAND / NOR • AOI / OAI 复合门 • 复杂门电路	• Ripple-Carry • Carry-Select • Carry-Lookahead • PGK 前缀框架 • Kogge–Stone • Sklansky	• 常规阵列乘法器 • Booth 编码乘法器 • Wallace / Dadda 树 • 最终快速加法器	• 逻辑 / 算术 / 循环 • 阵列（桶形）移位器 • 对数移位器 • 漏斗移位器	• 二进制编码器 / 预译码 • 编码器 / 优先编码器 • 相等比较器 • 大小 / 树形比较器

本讲结构： 每类典型组合逻辑 5 页 —— ① 原理与结构　② 性能分析　③ AI 辅助设计

1.3 AI 在组合逻辑设计中的四个作用点

- AI 负责提出候选方案，EDA 工具与形式验证负责最终裁决

1 LLM 生成 RTL

根据自然语言规格直接输出可综合 Verilog；VerilogEval、RTLLM 等基准用于评测生成正确率。鼓励结构规则、可参数化的模块。

2 结构空间搜索

前缀图、压缩器树、分块方案都是离散结构，可用强化学习或贝叶斯优化搜索面积-延迟 Pareto 前沿（如 PrefixRL）。

3 逻辑综合调优

学习综合命令序列（如 ABC 优化脚本组合），并用机器学习模型预测时序 / 面积，减少昂贵的综合与 STA 次数。

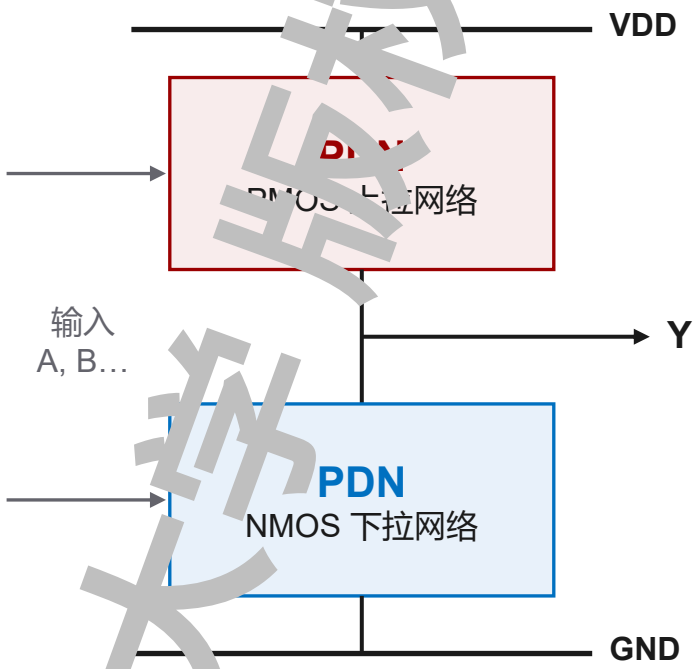
4 验证辅助

自动生成测试平台、断言与边界用例；与形式等价检查联用，保证 AI 生成的每一行 RTL 都“可验证”。

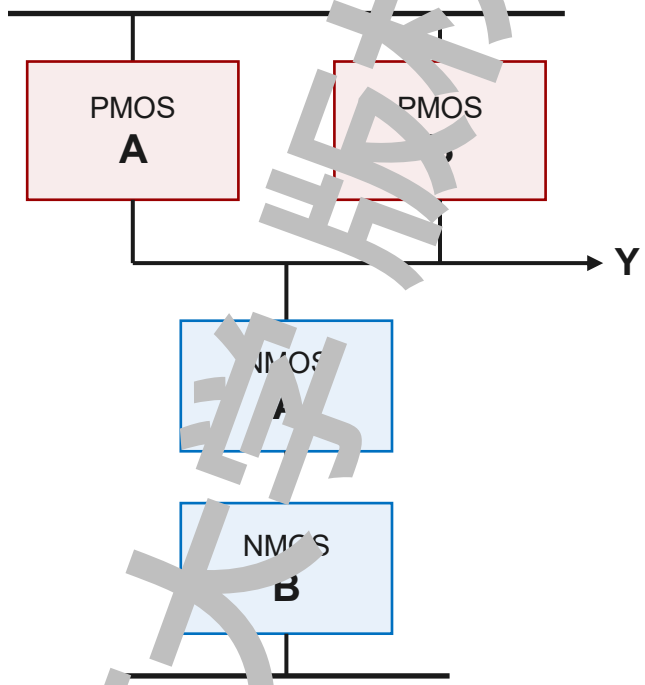
2.1 静态CMOS逻辑 – 互补结构

- 上拉网络 PUN 与下拉网络 PDN 互补：任一稳态只有一个导通

通用结构



实例：NAND2 Y = NOT(A·B)



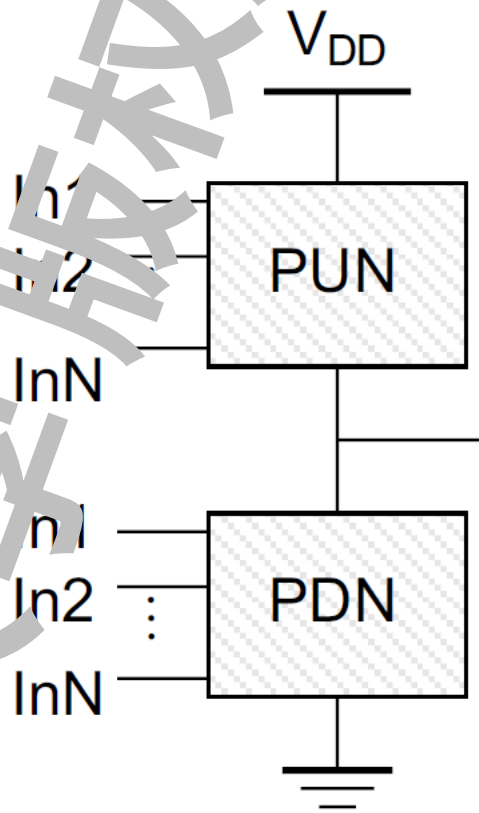
PUN：PMOS 并联 ⇄ PDN：NMOS 串联
（两者互为对偶）

核心特性

- 全摆幅输出，噪声容限大
- 稳态下无直流通路，静态功耗仅为泄漏
- 天然反相：单级只能实现非型函数
- PDN 串联 ↔ PUN 并联，互为对偶
- 串联管数一般 ≤ 4，过多堆叠过深导致变慢

经典静态逻辑电路

- NMOS、PMOS可以组成PDN和PUN



PMOS only

F(In1,In2,…InN)

NMOS only

上拉逻辑设计：

Pull-up network: make a connection from V_{dd} to F when F(In1,In2…)=1

下拉逻辑设计：

Pull-down network: make a connection from Ground to F when F(In1,In2…)=0

PUN and PDN are *dual* networks

经典静态逻辑电路

- 由PDN和PUN组成的NAND门电路

A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

Truth Table of a 2 input NAND gate

$OUT = \overline{A \bullet B}$ NAND门

PDN: Connects OUT to ground when $\overline{A} + \overline{B} = 1$

PUN: Connects OUT to V_{dd} when $\overline{A} + \overline{B} = 1$

So OUT = Complement of PDN function

Also OUT = PUN function with each input inverted

当A和B同为1时，输出为0

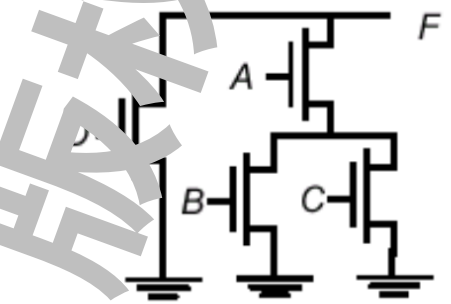
当A和B有0时，输出为1

经典静态逻辑电路

- 由PDN和PUN组成的复杂静态逻辑电路

$$F = \overline{D + (A\,(B + C))}$$

什么情况下F为0 -> D+A(B+C) = 1



忽略取非操作，直接构建PDN

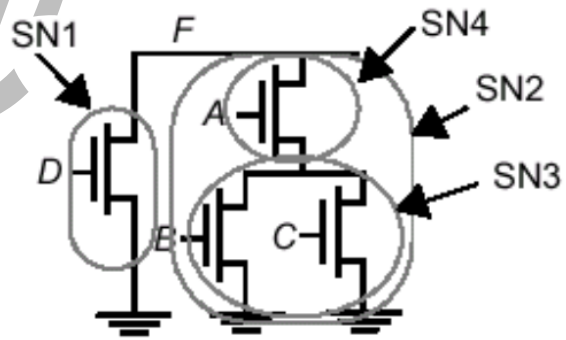
Ex：D与X意味着D与X并联

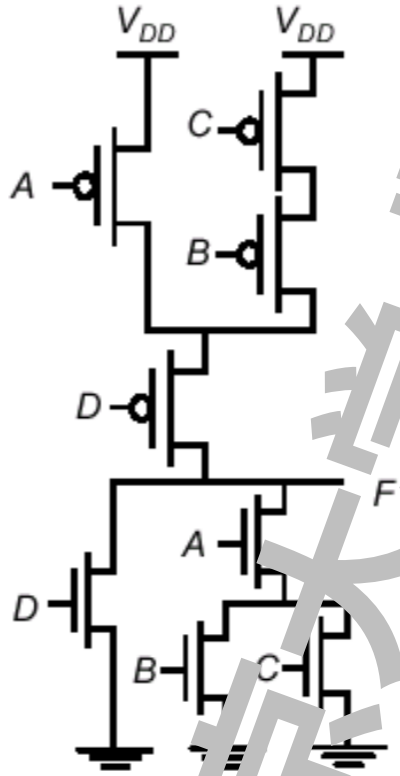
构建PDN的互补网络

在SN3内，B和C是并联的，

则在PUN中，他们串联

自下向上构建





2.3 AI辅助：静态 CMOS门级设计

- LLM 同时给出 RTL 与晶体管拓扑，工具负责确认等价与尺寸

提示词 Prompt

用静态CMOS实现 Y = ~((A & B) + C)，输出可综合 Verilog，并描述 PUN / PDN 的互
并联关系与建议尺寸。

LLM 生成的 Verilog

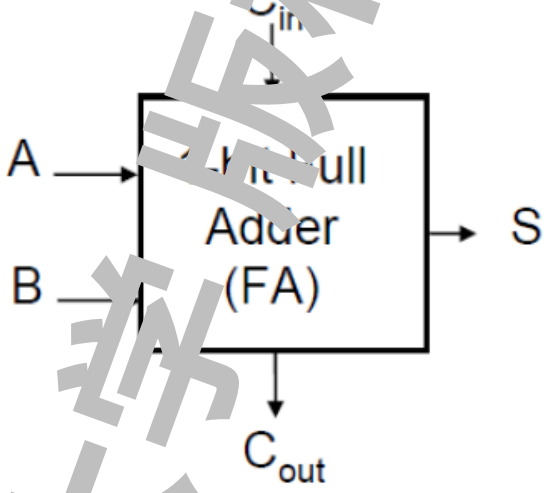
```
module aoi21 (
  input  wire a, b, c,
  output wire y
);
  // PDN: a 串 b 并 c
  // PUN: (a 并 b) 串 c   —— 与 PDN 对偶
  assign y = ~((a & b) | c);
endmodule
```

AI 输出检查清单

1. **功能等价**
   8 种输入穷举真值表，与规格逐项比对
2. **对偶性检查**
   AI 给出的 PUN / PDN 必须互为对偶，串并联不能写反
3. **组合纯净**
   禁止 always 块中不完全赋值——否则会推断出锁存器
4. **尺寸建议**
   按逻辑努力给出 W_p / W_n 初值，再由 SPICE / STA 确认

经典静态逻辑电路

- 简单1bit加法器电路
  - A[n-1:0] + B[n-1:0] = S[n-1:0]



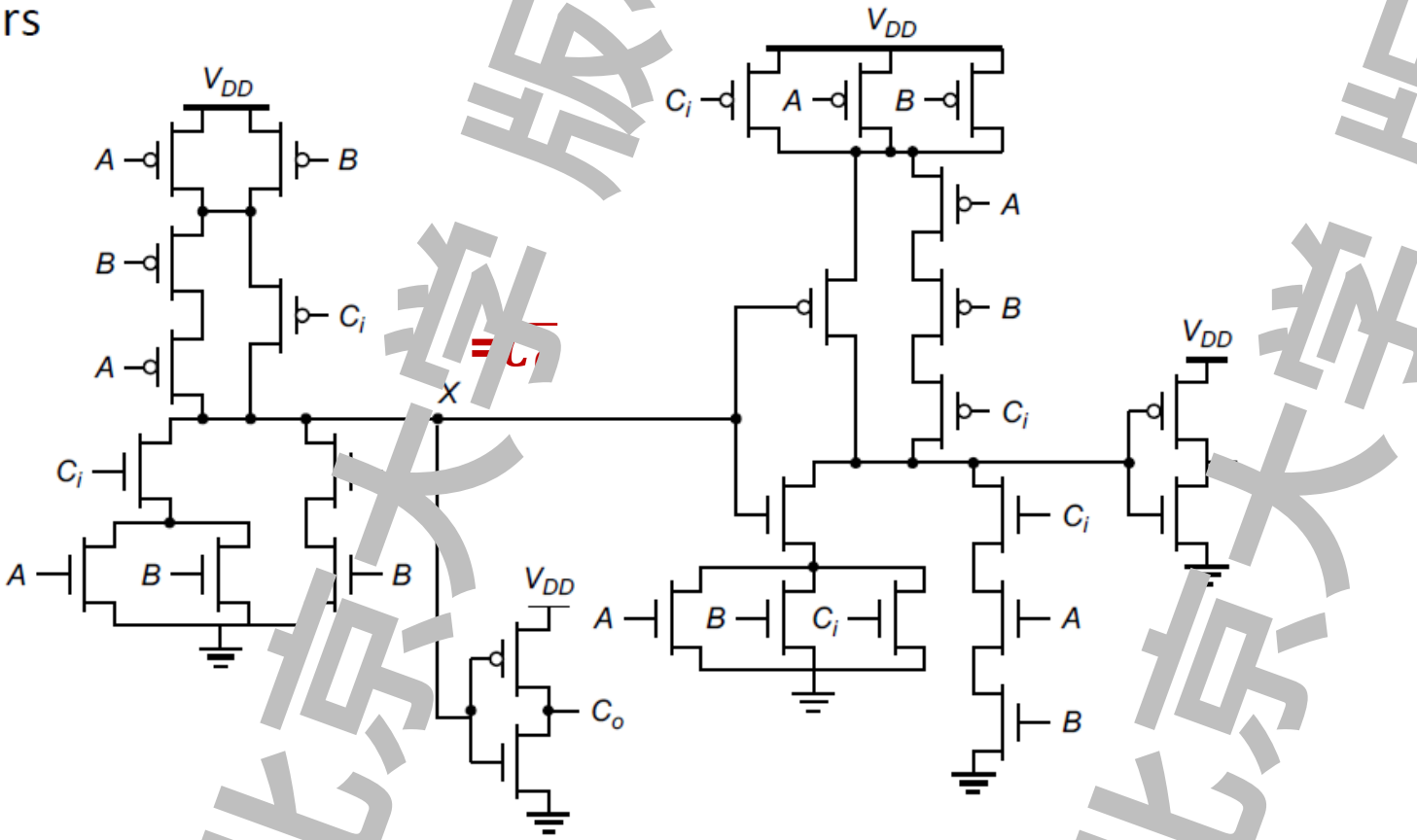
A	B	C_{in}	C_{out}	S	carry status
0	0	0	0	0	kill
0	0	1	0	1	kill
0	1	0	0	1	propagate
0	1	1	1	0	propagate
1	0	0	0	1	propagate
1	0	1	1	0	propagate
1	1	0	1	0	generate
1	1	1	1	1	generate

  - $C_o = AB + BC_i + AC_i = AB + (A + B)C_i$
  - $S = A \oplus B \oplus C_i = ABC_i + \overline{C_o}(A + B + C_i)$
  - $G = AB,\ K = \overline{A}\,\overline{B},\ P = A \oplus B$

单比特加法器逻辑设计

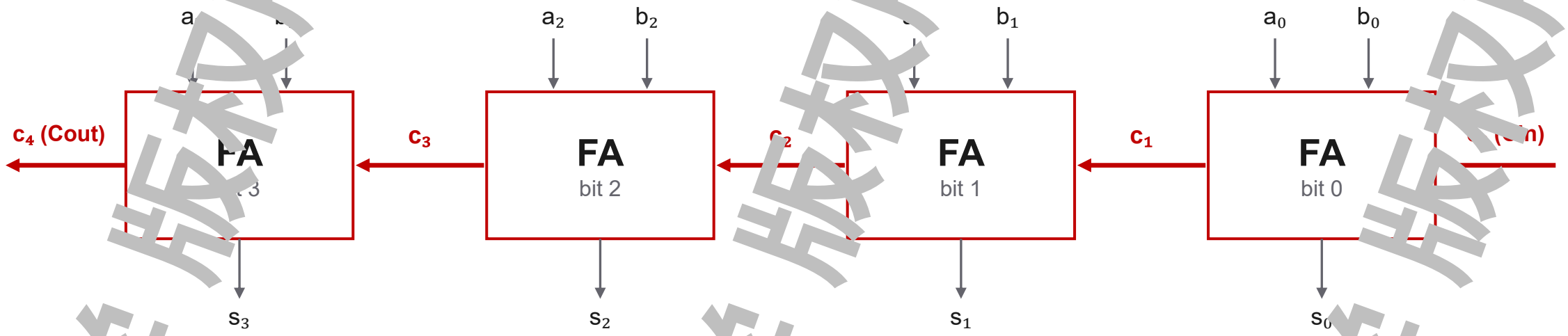
经典静态逻辑电路

- 简单1bit加法器电路
  - $C_o = AB + BC_i + AC_i = AB + (A + B)C_i$
  - $S = A \oplus B \oplus C_i = ABC_i + \overline{C_o}(A + B + C_i)$
  - 28 transistors



3.1 Ripple-Carry 加法器 – 结构

- N 个全加器首尾级联，进位逐位传播



关键路径：进位信号自 c_0 或 c_{in} 逐位串行穿过每一个全加器（红色路径）

$$s_i = a_i \oplus b_i \oplus c_i$$

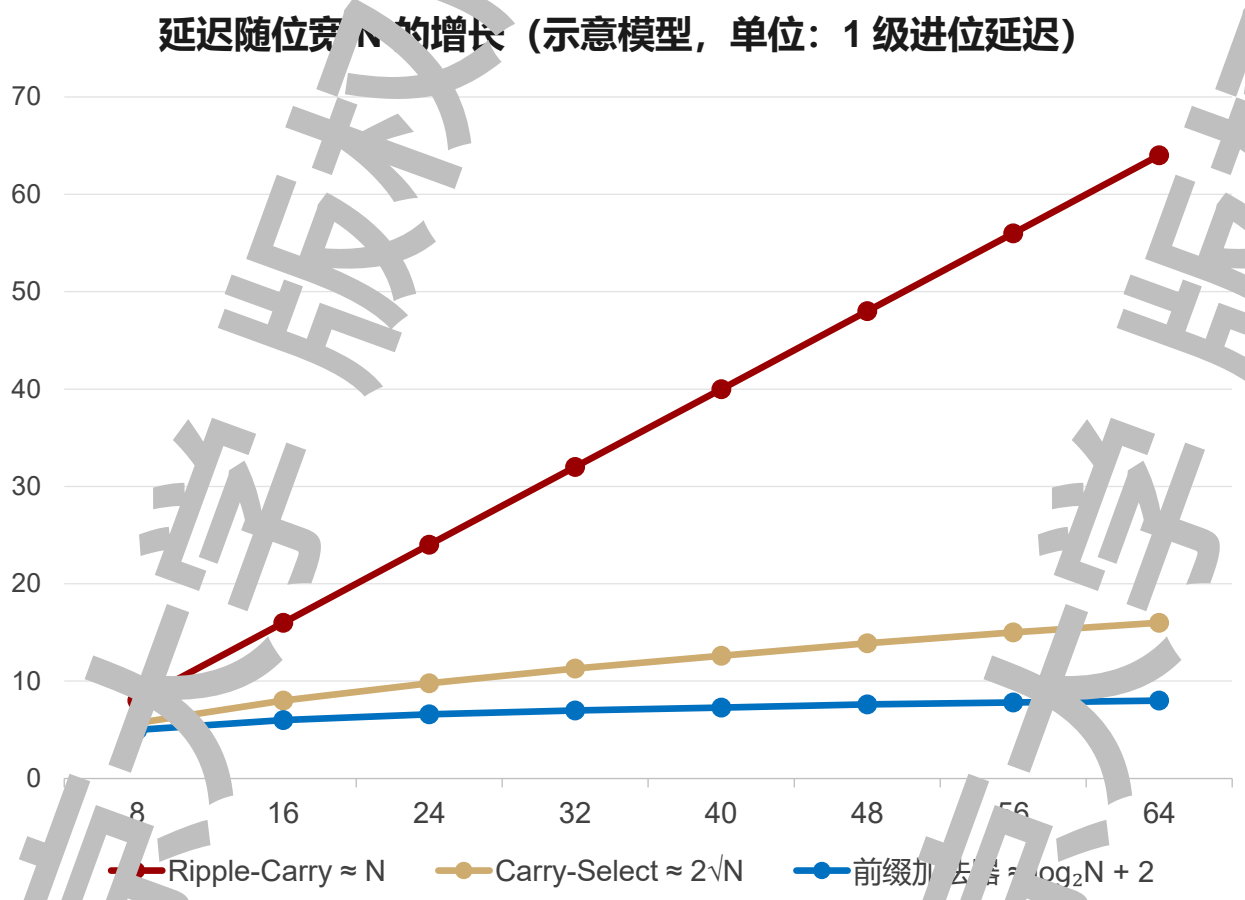
$$c_{i+1} = a_i b_i + c_i(a_i \oplus b_i)$$

$$= MAJ(a_i, b_i, c_i)$$

- 结构最简单：N 个全加器级联
- 面积 O(N)，延迟 O(N)
- 规则、低功耗，常作其他加法器的子块

3.2 Ripple-Carry 加法器 – 关键路径分析

- 延迟与位宽线性相关：进位路径是唯一需要优化的对象



$$t \approx t_{setup} + (N-1)\cdot t_{carry} + t_{sum}$$

- 进位路径决定性能：优化 t_{carry} 比 t_{sum} 更重要
- 镜像加法器：进位级只有 2 级晶体管堆叠
- 反相进位：奇数位使用反相输入 / 输出，省掉每级反相器
- N ≤ 8，或面积 / 功耗优先时仍是首选

3.3 AI辅助：RCA 的生成与验证

- 参数化生成器 + 最坏路径向量 + 形式等价，三步完成

提示词 Prompt

生成位宽可参数化的行波进位加法器，要求用 generate 循环逐位实例化全加器逻辑，并输出 cout。

LLM 生成的 Verilog

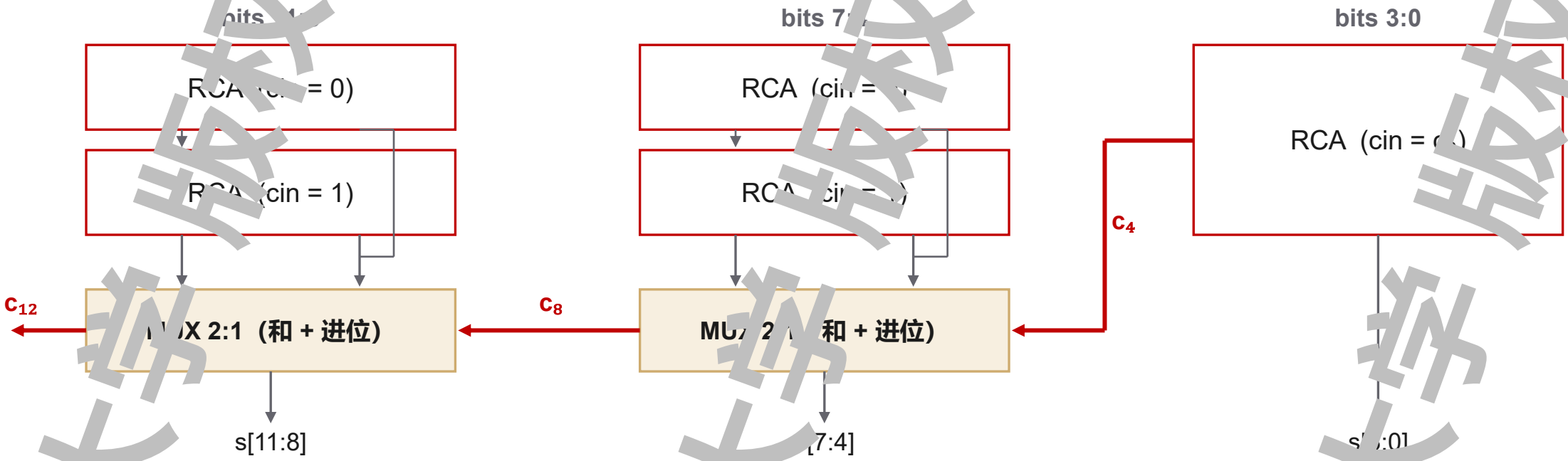
```
module rca #(parameter N = 16) (
  input  [N-1:0] a, b,
  input          cin,
  output [N-1:0] s,
  output         cout
);
  wire [N:0] c;
  assign c[0] = cin;
  genvar i;
  generate for (i = 0; i < N; i = i + 1) begin : g_fa
    assign s[i]   = a[i] ^ b[i] ^ c[i];
    assign c[i+1] = (a[i] & b[i]) | (c[i] & (a[i] ^ b[i]));
  end endgenerate
  assign cout = c[N];
endmodule
```

AI 输出检查清单

1. **参数化与位宽**
   要求使用 parameter N，并明确 c[N:0] 的位宽
2. **最坏情况向量**
   a = 全 1、b = 0、cin = 1 激活整条进位链，用于时序仿真
3. **形式等价**
   与参考模型 {cout, s} == a + b + cin 做等价检查
4. **保持结构**
   综合器会把加法重构为更快结构；需保留 RCA 时加 dont_touch 约束

3.4 Carry-Select 加法器 – 结构

- 每块预先计算 cin = 0 / 1 两种结果，真实进位到达后只需一级 MUX



- 每块两套 RCA 并行工作，与低位进位无关
- 块内进位 + 块间 MUX 选择构成两级结构
- 关键路径：首块 RCA + (块数 − 1) 级 MUX
- 以约 2 倍加法器面积换取速度

3.5 Carry-Select 加法器 – 分块策略与延迟

- 递增分块让各块结果与选择信号同时就绪，延迟降为 $O(\sqrt{N})$

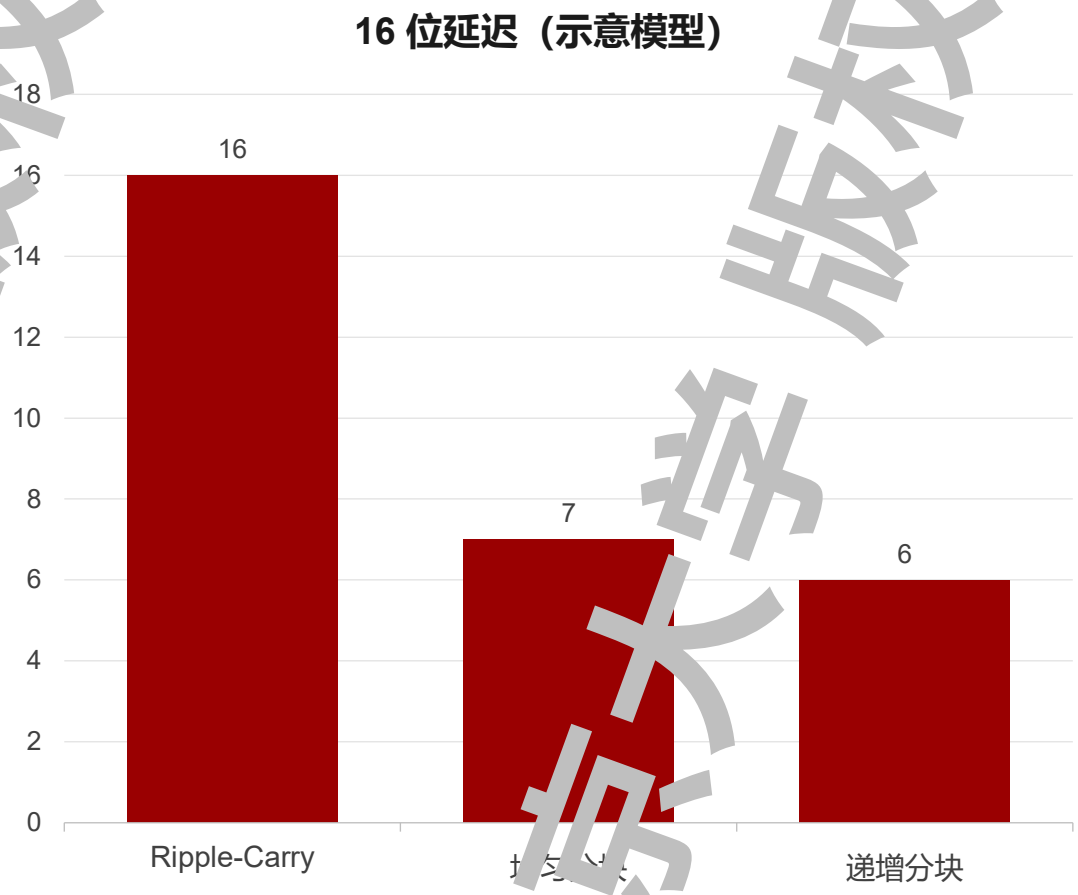
$$t \approx M \cdot t_{carry} + (N/M) \cdot t_{mux}$$

$$M_{opt} \approx \sqrt{(N \cdot t_{mux} / t_{carry})} \Rightarrow O(\sqrt{N})$$

均匀分块 4-4-4-4

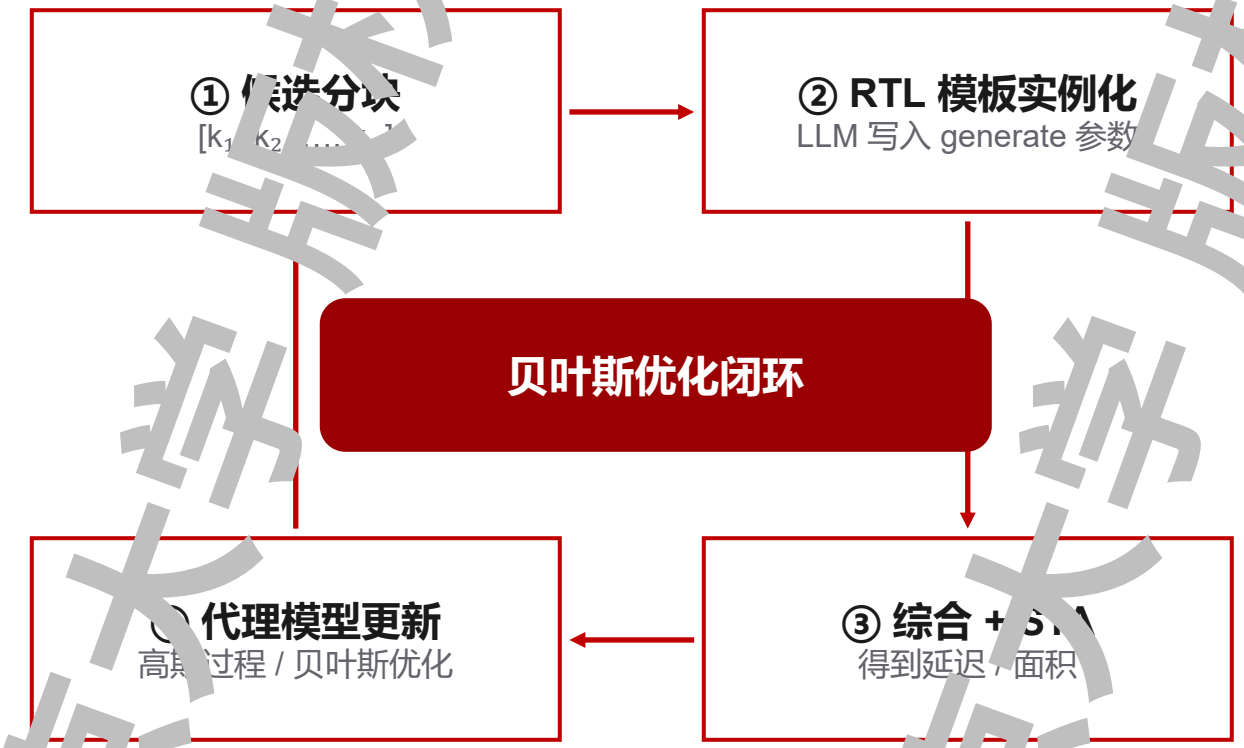
递增分块 5-4-3-2-2

- 递增块：低位块小、高位块大，结果就绪时间与 MUX 选择信号到达时间对齐
- 代价：面积约为 RCA 的 2 倍，外加每位一个 MUX
- 设计可以让：**高位块内部计算延迟 = 前级进位传输延迟**，进位信号就像通过一条平滑的斜坡一样，无缝地一路选择过去，使得最后一级输出结果的总延时达到最小。



3.6 AI辅助：Carry-Select 分块方案搜索

- 分块向量是典型离散设计空间，适合贝叶斯优化 + 代理模型



为什么需要 AI 搜索

- N 位分块组合数为 2^{N-1}，N = 16 时已达 32768 种
- 真实延迟受扇出、布线与单元库影响，解析模型不准
- 代理模型只挑最有希望的候选做真实综合

LLM 的角色

- 把分块向量写成参数，自动生成 RTL
- 解析时序报告，指出瓶颈块并建议新候选

3.7 Carry-Lookahead 加法器 – 产生与传播

- 把进位改写为 g_i、p_i 的函数，所有进位可以并行求出

基本定义

$g_i = a_i \cdot b_i$ 产生

$p_i = a_i \oplus$ 传

$c_{i+1} = g_i$ … c_i

$s_i = p_i \oplus c_i$

展开后的进位方程

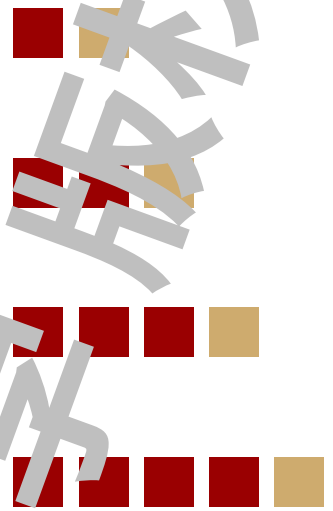
$c_1 = g_0 + p_0c_0$

$c_2 = g_1 + p_1$ … $p_1p_0c_0$

$c_3 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$

$c_4 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$

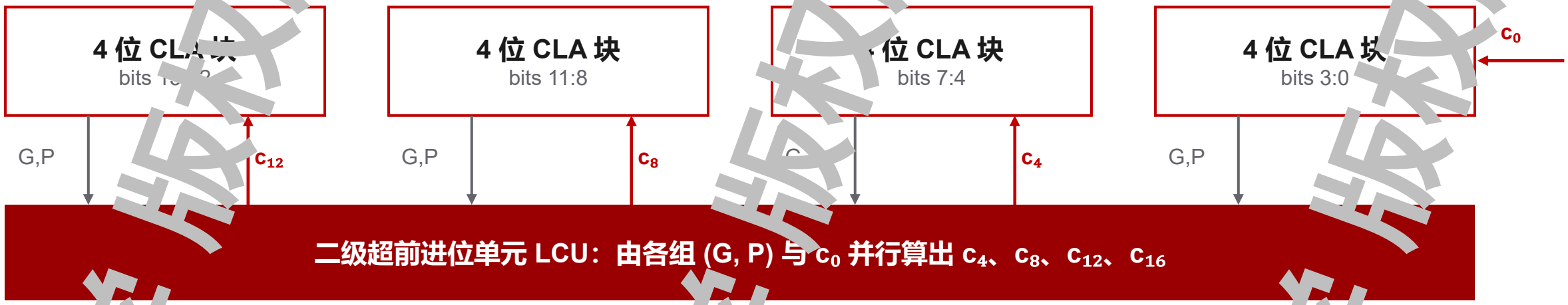
乘积项数



所有进位只需两级（与-或）即可并行求出；但乘积项数与扇入随位数线性增长 → 必须分组

3.8 Carry-Lookahead 加法器 – 分组与多级超前

- 组内求 (G, P)，组间再做一次超前进位：延迟降为 O(log N)



$$C_{\cdot} = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0$$

$$P = p_3p_2p_1p_0$$

$$c_{\cdot} = G_0 + P_0 \cdot c_0$$

- 延迟 ≈ $O(\log_k N)$，k 为每组位数（受扇入限制，常取 4）
- 组内与组间使用同一 (G, P) 形式 → 可递归构造
- “分组求 (G, P)”正是前缀加法器的出发点

3.9 AI辅助：CLA 展开式与扇入权衡

- 让 AI 生成展开式时必须同时给出扇入约束与分组方式

提示词 Prompt

> 实现 16 位两级 CLA：先写可复用的 4 位 cla4 模块（输出组 G/P），再用同一模块作为组间 LCU，每个门扇入不超过 4。

LLM 生成的 Verilog

```
// 4 位组：组内进位 + 组 (G, P)，每个门扇入 ≤ 4
module cla4 (
  input  [3:0] g, p,
  input        c0,
  output [3:1] c,
  output       G, P
);
  assign c[1] = g[0] | (p[0] & c0);
  assign c[2] = g[1] | (p[1] & g[0]) | (p[1] & p[0] & c0);
  assign c[3] = g[2] | (p[2] & g[1]) | (p[2] & p[1] & g[0])
                     | (p[2] & p[1] & p[0] & c0);
  assign G = g[3] | (p[3] & g[2]) | (p[3] & p[2] & g[1])
                  | (p[3] & p[2] & p[1] & g[0]);
  assign P = &p;
endmodule
```

AI 输出检查清单

1. **扇入约束**
   在提示中写明每个门 ≤ 4 输入，超限时要求 AI 自动再分组
2. **递归复用**
   同一 cla4 既做组内也做组间 LCU，减少手写错误
3. **p 的两种定义**
   进位链可用 p = a | b，求和必须用 p = a ⊕ b —— AI 常混用
4. **等价验证**
   16 位整体与 a + b 做形式等价检查

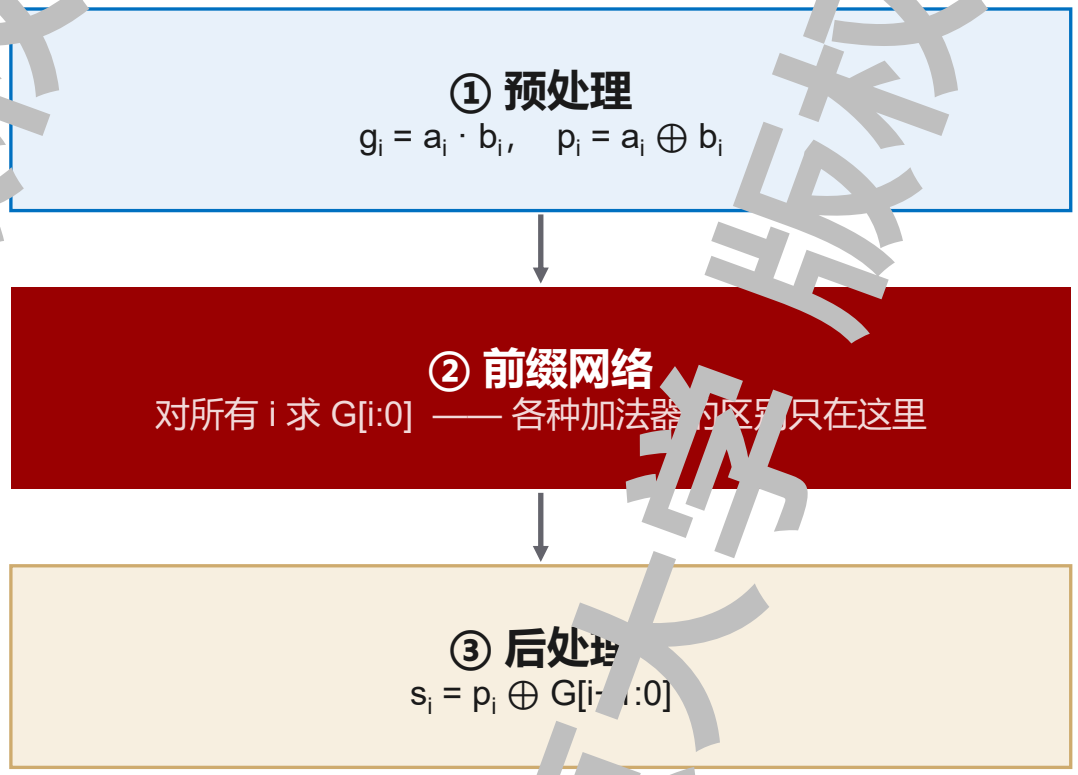
3.10 PGK 加法 – 进位的三种状态

- 每一位对进位只有三种作用：产生 Generate、传播 Propagate、消除 Kill

a_i	b_i	状态	对进位的作用
0	0	K Kill	$c_{i+1} = 0$，进位被吸收
0	1	P Propagate	$c_{i+1} = c_i$，进位穿过
1	0	P Propagate	$c_{i+1} = c_i$，进位穿过
1	1	G Generate	$c_{i+1} = 1$，本位产生进位

推广到区间 [i : j]

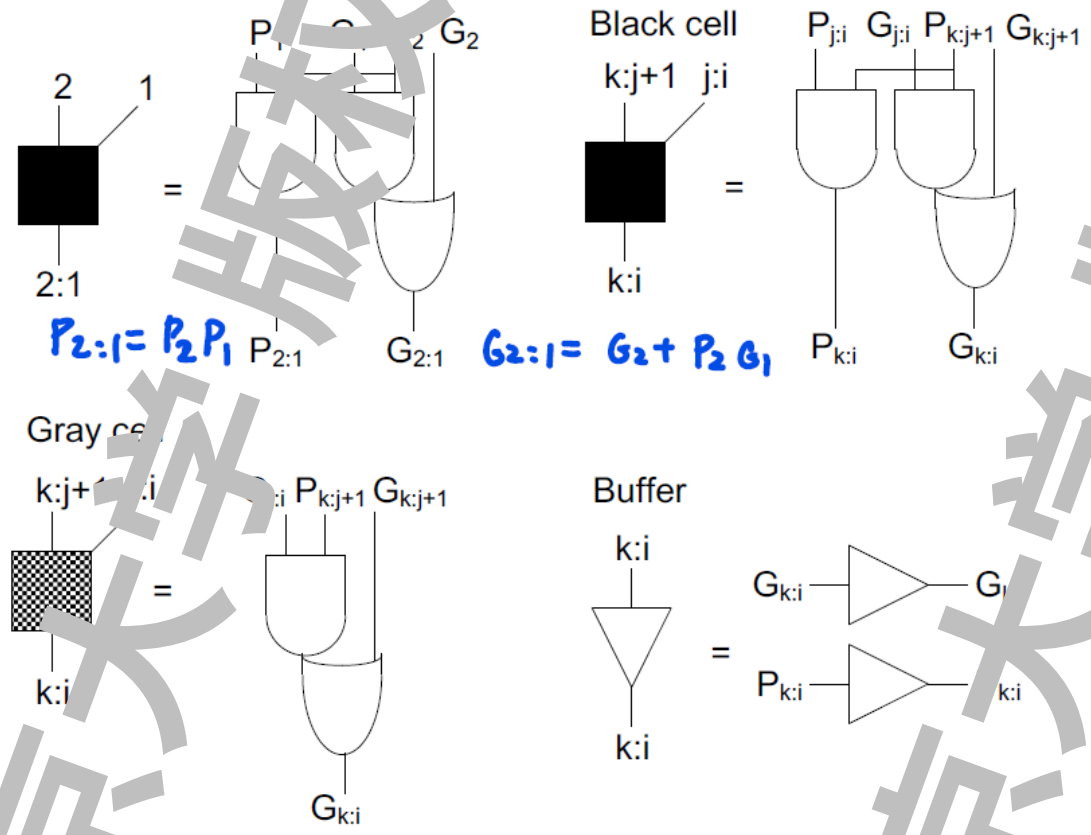
- G[i:j]：该区间自身产生进位；P[i:j] = 1：进位可穿过整个区间
- $c_{i+1} = G[i:0]$（把 c_0 视为第 −1 位的 G）
- $s_i = p_i \oplus G[i-1:0]$



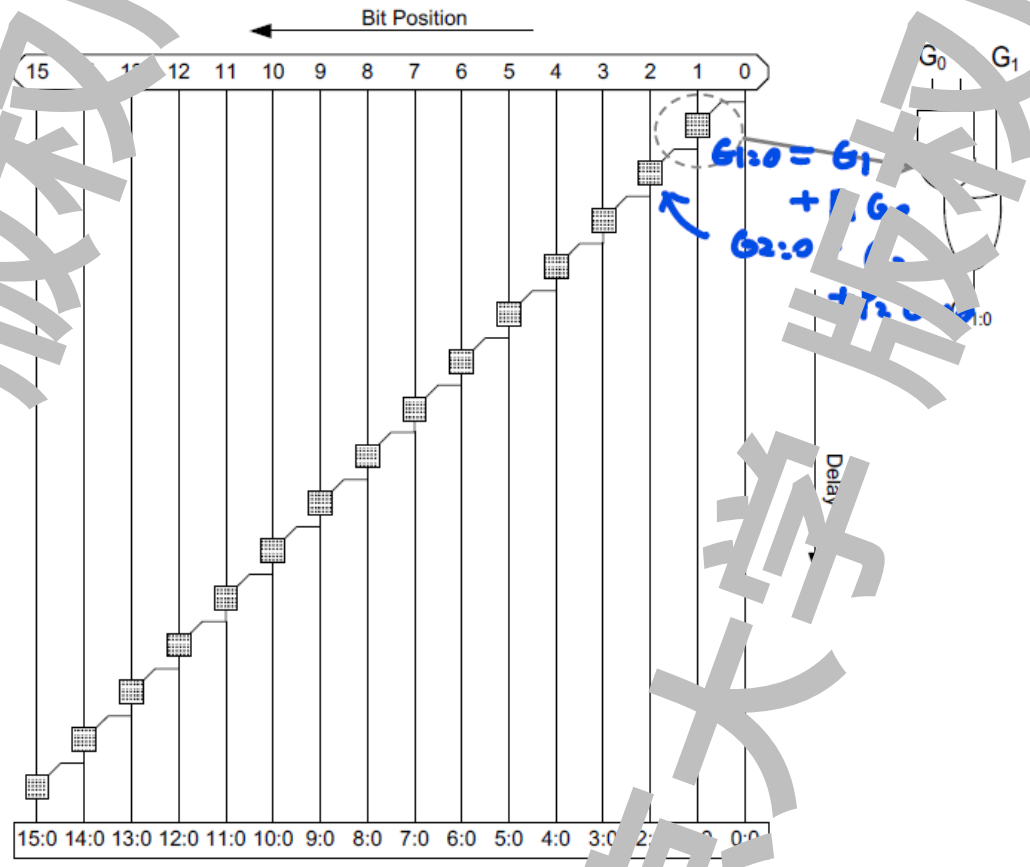
RCA、CLA、KS、Sklansky 都是 ② 的不同实现

加法器设计

- 基于PGK的加法器设计方法



PG生成逻辑



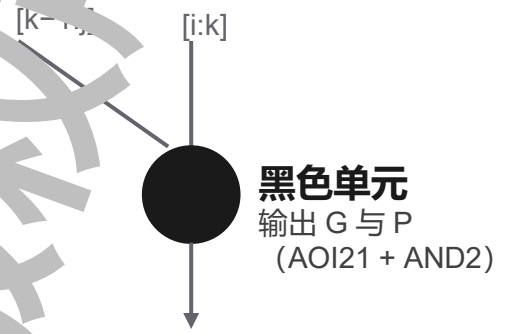
Carry Ripple的PG图

3.11 PGK 加法 – 前缀运算子

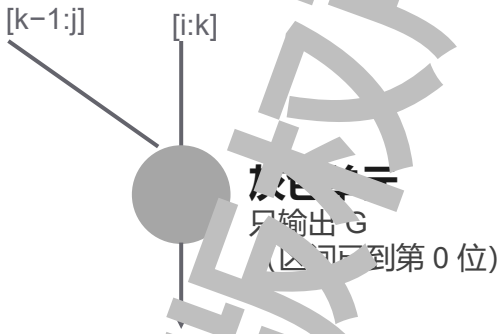
- 前缀运算满足结合律：括号怎么加都正确，不同加括号方式 = 不同前缀树

$$(G, P) \circ (G', P') = (G + P \cdot G',\ P \cdot P')$$

(G,P)[i:j] = (G,P)[i:k] ∘ (G,P)[k−1:j]　满足结合律，不满足交换律

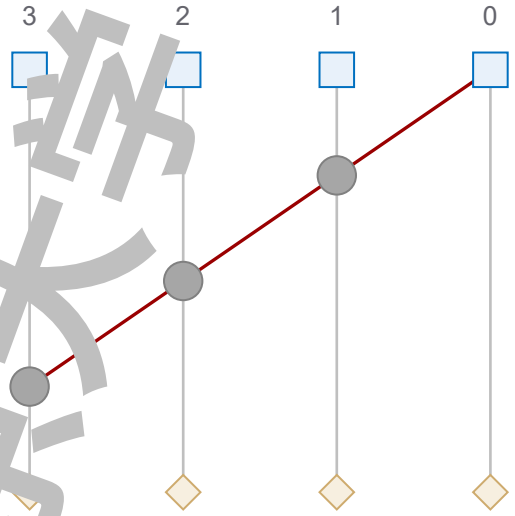


黑色单元
输出 G 与 P
（AOI21 + AND2）



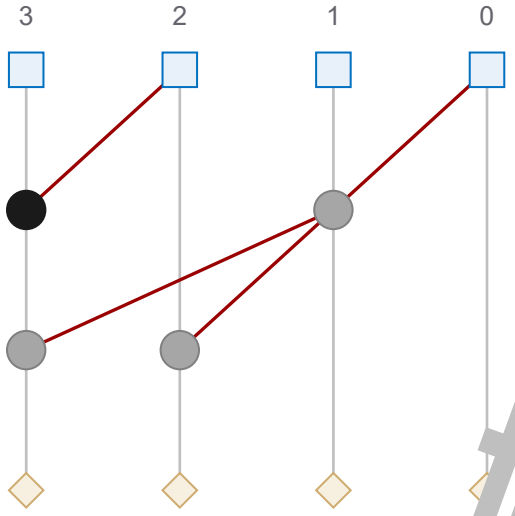
只输出 G
（区间压到第 0 位）

串行：$g_3 \circ (g_2 \circ (g_1 \circ g_0))$



3 级 —— 本质就是 RCA

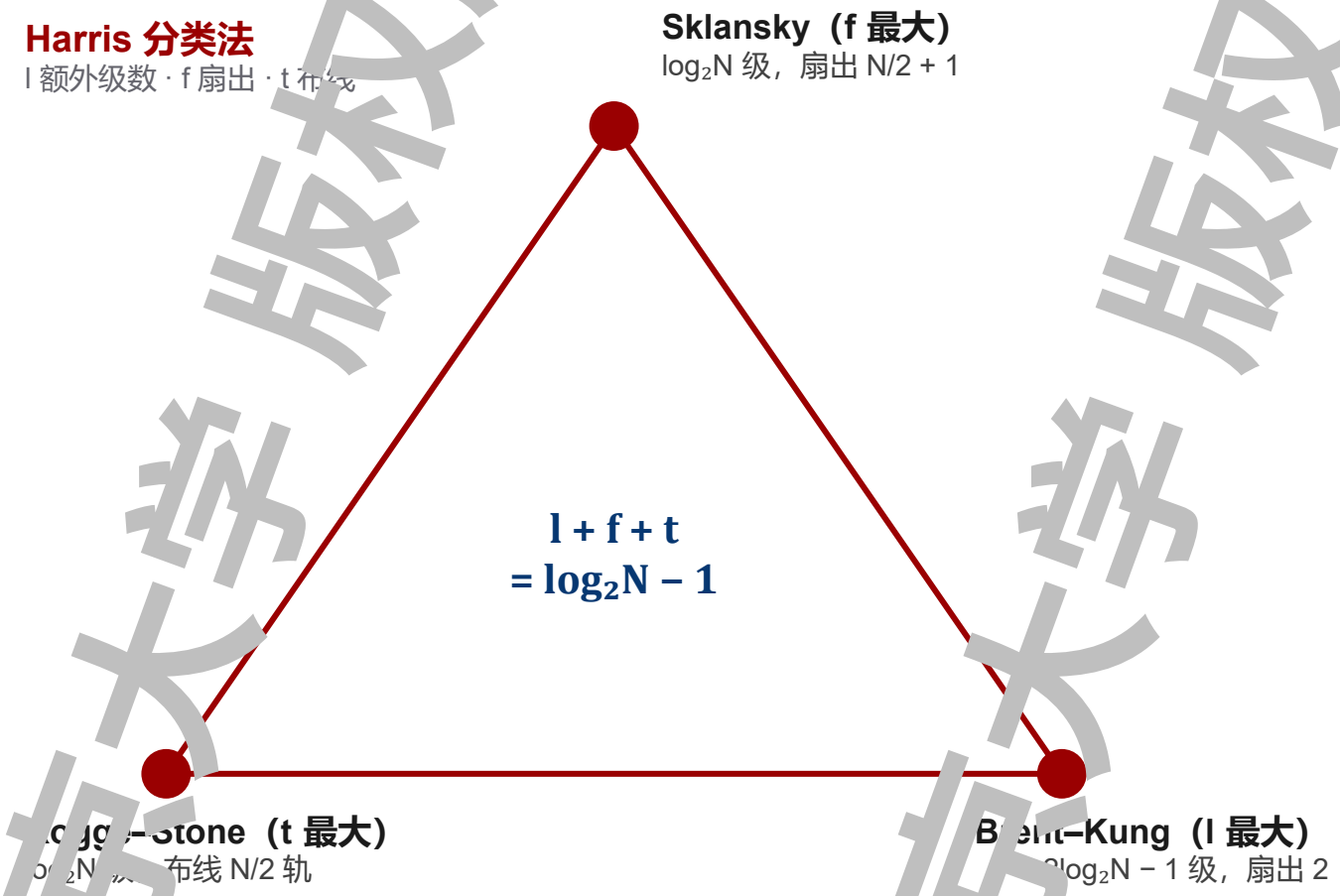
树形：$(g_3 \circ g_2) \circ (g_1 \circ g_0)$



2 级 —— 前缀树

3.12 AI辅助：前缀加法器的统一设计空间

- 把前缀图当作可搜索的对象：经典结构只是空间中的几个角点



PrefixRL 类方法：用强化学习设计前缀图

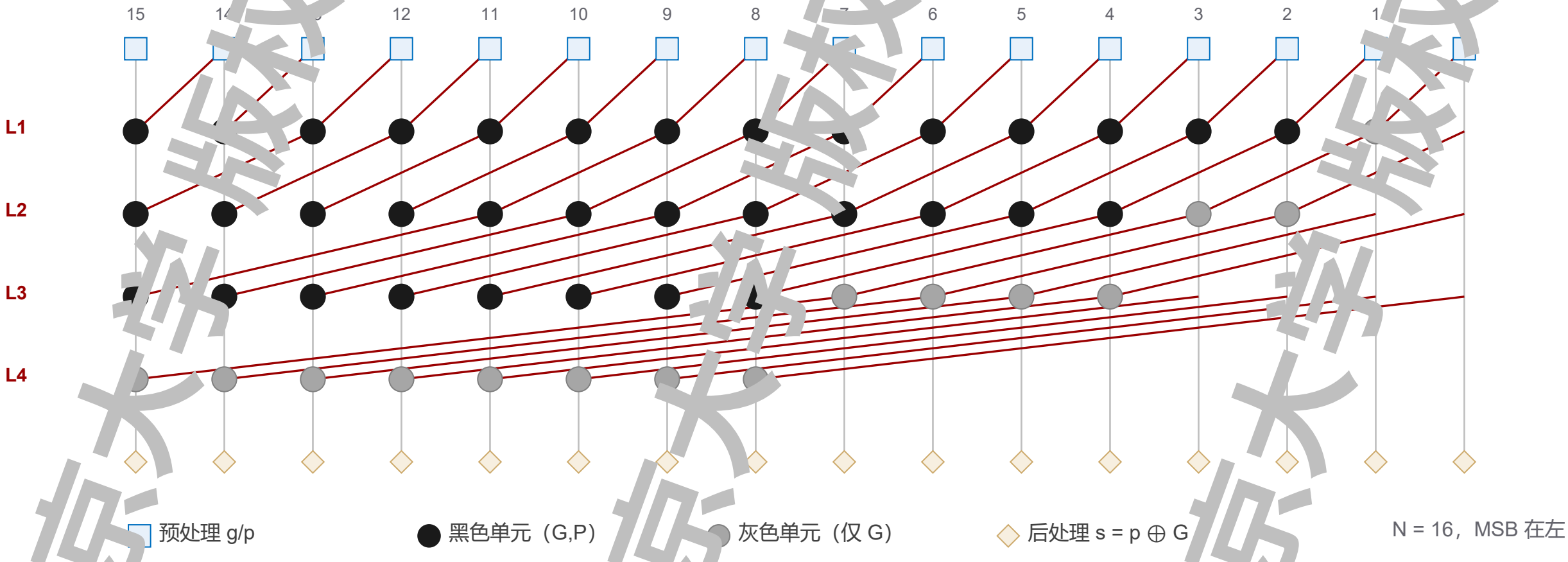
- 状态：N×N 网格，节点 (i, j) 表示计算 G[i:j]
- 动作：增加 / 删除一个节点，自动补全依赖并保持图合法
- 奖励：综合后面积与延迟的加权改进
- 已报道的结果在面积-延迟上优于传统工具生成的加法器

LLM 的角色

- 把前缀图（节点列表）自动转成 RTL / 网表
- 解释各 Pareto 点的结构差异，辅助人工取舍

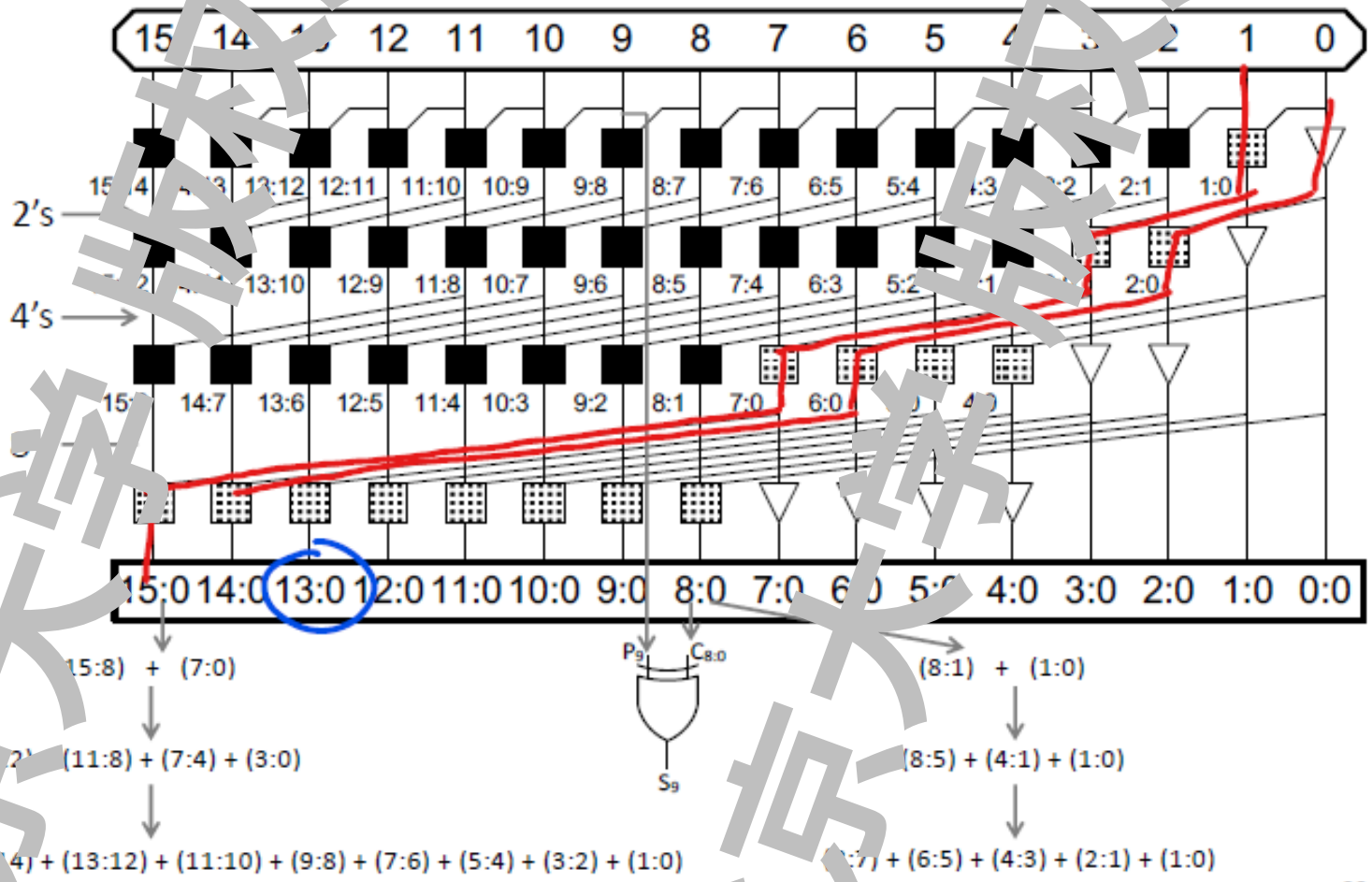
3.13 Kogge–Stone 加法器 – 结构

- 每级跨度翻倍（1, 2, 4, 8），$\log_2 N$ 级完成所有前缀，扇出恒为 2



3.13 Kogge–Stone 加法器 – 结构

- 基于PGK的加法器设计方法 – 复杂PG树加法器



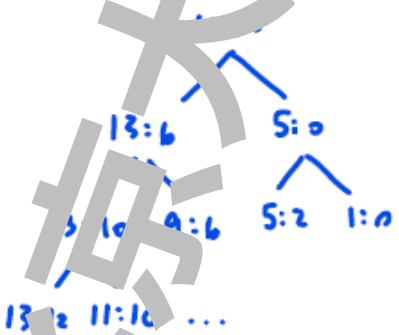
$\log_2(N)$

$G_{15:0} = G_{13:6} + P_{13:6}\,G_{5:0}$

$G_{13:6} = G_{13:10} + P_{13:10}\,G_{9:6}$

$P_{13:6} = P_{13:10}\,P_{9:6}$

$G_{5:0} = G_{5:2} + P_{5:2}$

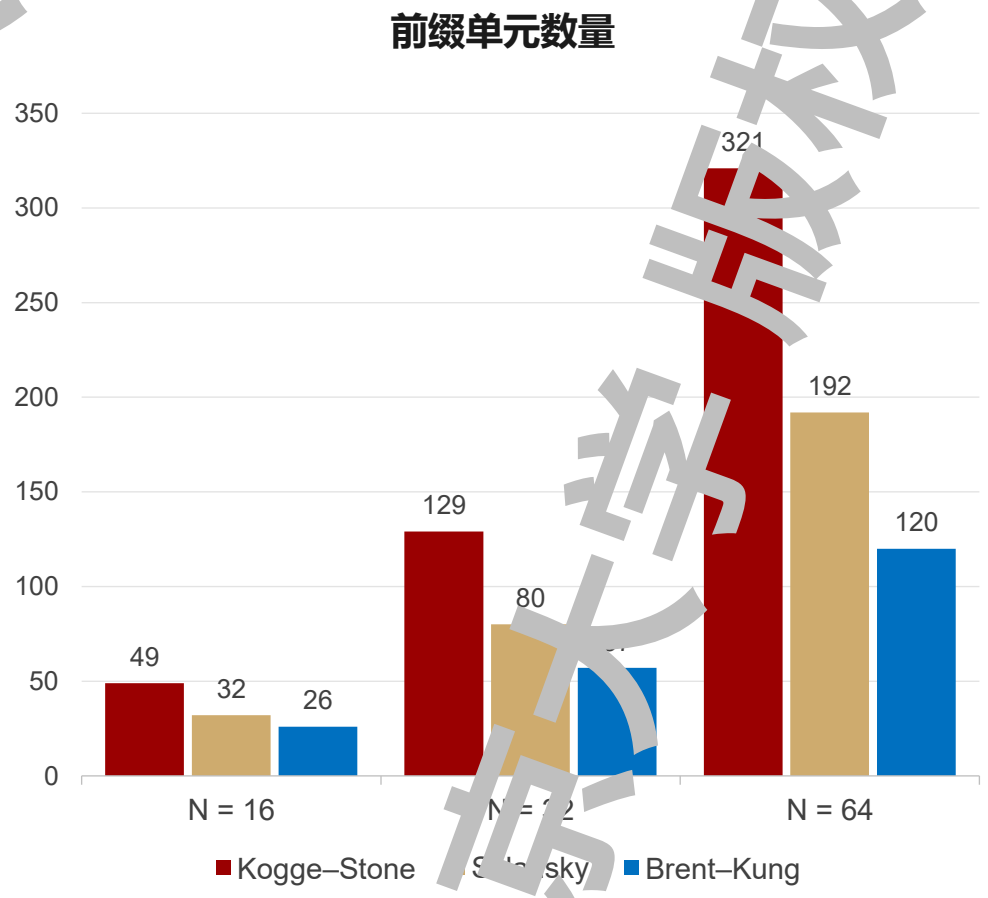


3.14 Kogge–Stone 加法器 – 特性与代价

- 最少级数 + 最小扇出，代价是前缀单元与长连线最多

结构	级数	前缀单元数	最大扇出	布线轨道
Kogge–Stone	$\log_2 N$	$N\cdot\log_2 N - N + 1$	2	N/2
Sklansky	$\log_2 N$	$(N/2)\cdot\log_2 N$	N/2 + 1	
Brent–Kung	$2\log_2 N - 1$	$2N - 2 - \log_2 N$	2	1

- 关键路径短、每个节点只驱动 2 个负载 → 高性能 ALU 的首选
- 代价：前缀单元数最多，跨越 N/2 位的长连线带来布线拥塞与功耗
- 常见折中：Han–Carlson（KS + BK）、稀疏 KS（每 2/4 位一个前缀位）



3.15 AI辅助：Kogge–Stone 生成器与布线感知

- 规则的级联结构非常适合 LLM 生成；布线问题交给 EDA 模型评估

提示词 Prompt

生成参数化 Kogge–Stone 加法器：第 l 级把 i 与 i − 2^{l-1} 合并，越界位直通；c[

LLM 生成的 Veri

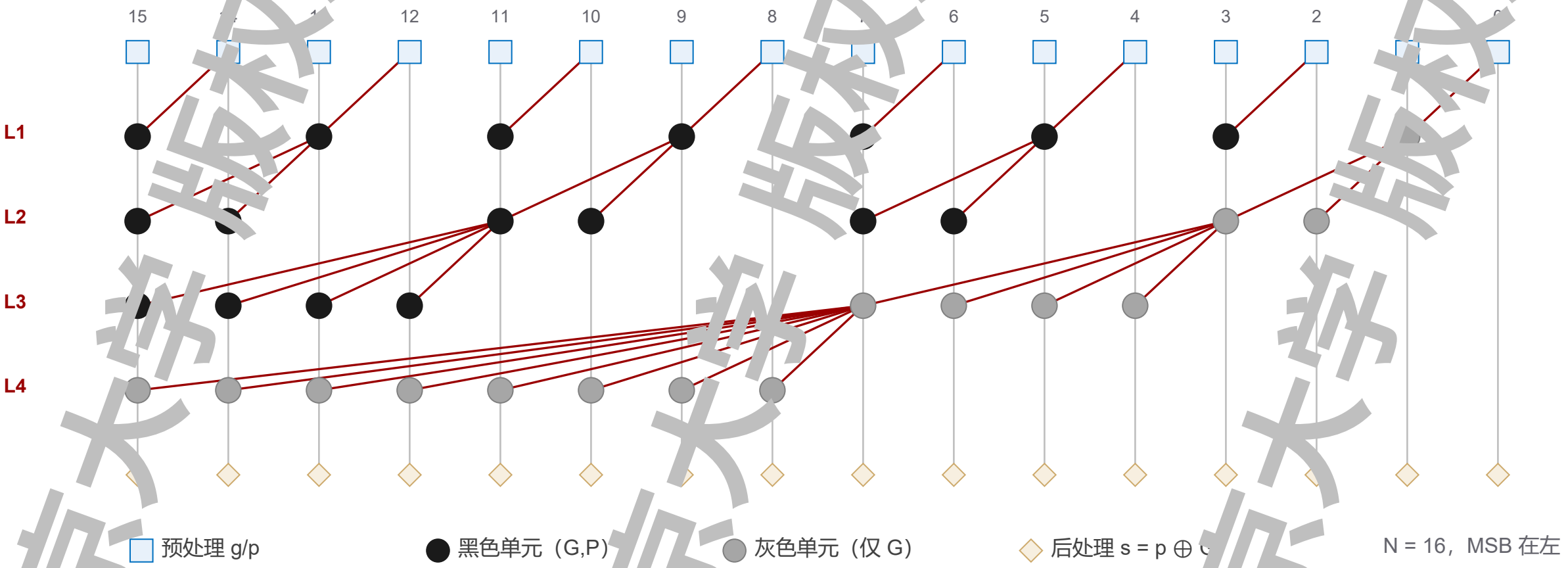
```
module kogge_st ne  #(parameter N = 16, L = $clog2(N)) (
  input  [N-1:0]  a, b,
  output [N-1:0] s,
  output         cout);
  wire [N:0] G [0:L], P [0:L];
  assign G[0] = c & b;  assign P[0] = a ^ b;
  genvar i, l;
  generate for (l = 1; l <= L; l = l + 1) begin : lv     (级数l)
    for (i = 0; i < N; i = i + 1) begin : bt
      if (i >= (1 << (l-1))) begin : blk
        assign G[l][i] = G[l-1][i] | (P[l-1][i] & G[l-1][i-(1<<(l-1))]);
        assign P[l][i] = P[l-1][i] & P[l-1][i-(1<<(l-1))];
      end else begin : pass
        assign G[l][i] = G[l-1][i];
        assign P[l][i] = P[l-1][i];
      end
    end
  end endgenerate
  assign s    = P[0] ^ {G[L][N-2:0], 1'b0};
  assign cout = G[L][N-1];
endmodul
```

AI 输出检查清单

1. **越界分支**
   i < 2^{l-1} 时必须直通，AI 常漏写该分支
2. **进位对齐**
   s = p ⊕ {G[N−2:0], cin}，移位一位不能错
3. **布线感知**
   预测拥塞热点，改用稀疏 KS + 进位选择
4. **验证**
   与 a + b 形式等价，并对比综合后的 PPA

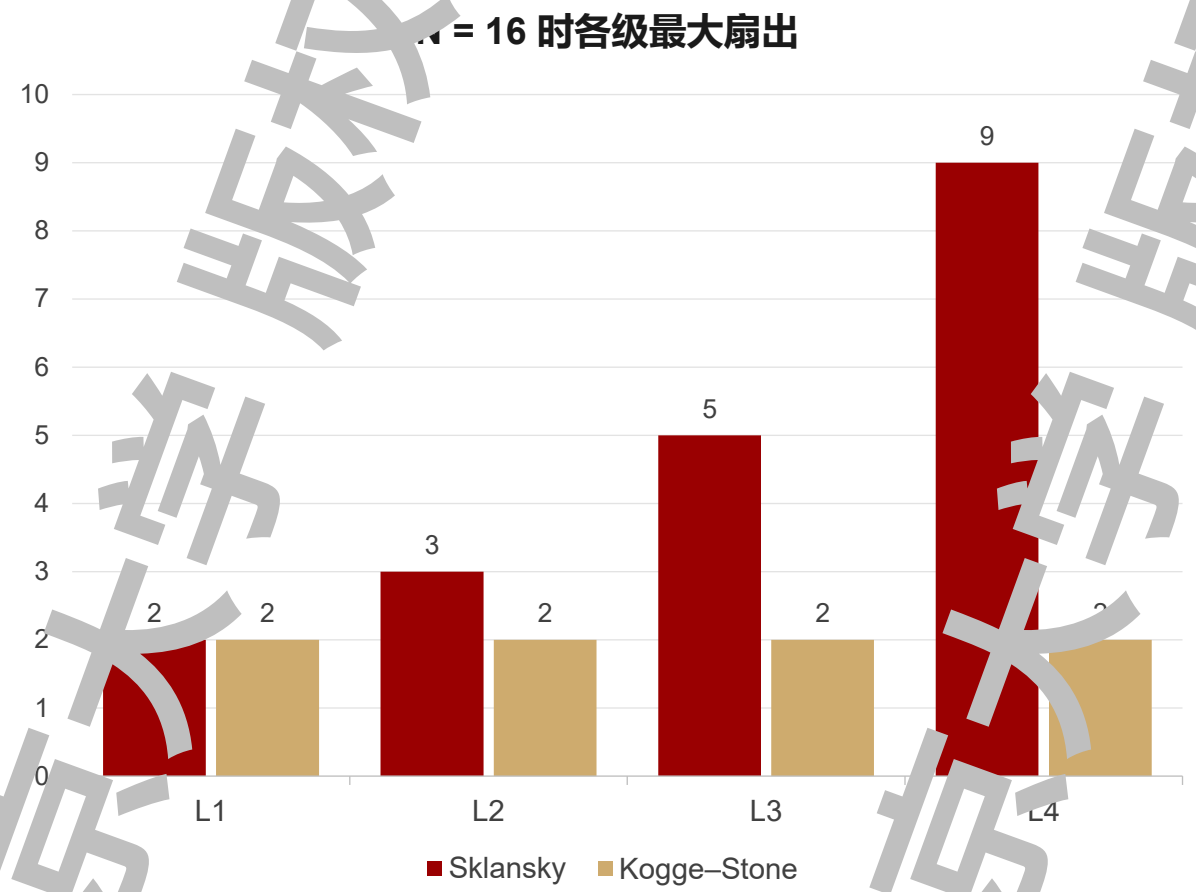
3.16 Sklansky 加法器 – 结构

- 分治思想：每级把相邻两块合并，低半块最高位的结果广播到高半块



3.17 Sklansky 加法器 – 扇出问题与对策

- 级数与 KS 相同，单元更少，但扇出逐级翻倍，最大达 N/2 + 1



高扇出的代价

- 扇出 N/2 + 1 的节点驱动大负载，实际延迟远大于“$\log_2 N$ 级”所暗示的数值
- 优点：单元较少、无长距离交叉连线

对策

1. **逐级放大**
   按逻辑努力增大高扇出节点的驱动尺寸
2. **插入缓冲**
   把扇出分摊给多个缓冲器，代价是增加级数
3. **折中结构**
   Ladner–Fischer 等：多 1 级换取扇出减半

3.18 AI辅助：Sklansky 的扇出驱动尺寸优化

- 生成结构只是第一步，AI 还要给出扇出表供尺寸与缓冲决策

提示词 Prompt

生成参数化 Sklansky 加法器：**第 l 级中 i / D 为奇数的位与 J = (i / D)·D − 1 合并**（整数除法），并输出每一位的扇出。

LLM 生成的 Verilog

```
module sklansky (parameter N = 16, L = $clog2(N)) (
  input  [N-1:0] a, b,
  output [N-1:0] s,
  output         cout
  wire [N-1:0] G [0:L], P [0:L];
  assign G[0] = a & b;  assign P[0] = a ^ b;
  genvar l, i;
  generate for (l = 1; l <= L; l = l + 1) begin : lv
    for (i = 0; i < N; i = i + 1) begin : bt
      localparam integer D = 1 << (l-1);
      localparam integer J = (i / D) * D - 1;
      if ((i / D) % 2 == 1) begin : blk
        assign G[l][i] = G[l-1][i] | (P[l-1][i] & G[l-1][J]);
        assign P[l][i] = P[l-1][i] & P[l-1][J];
      end else begin : pass
        assign G[l][i] = G[l-1][i];
        assign P[l][i] = P[l-1][i];
      end
    end
  end endgenerate
  assign s    = P[0] ^ {G[L][N-2:0], 1'b0};
  assign cout = G[L][N-1];
endmodule
```

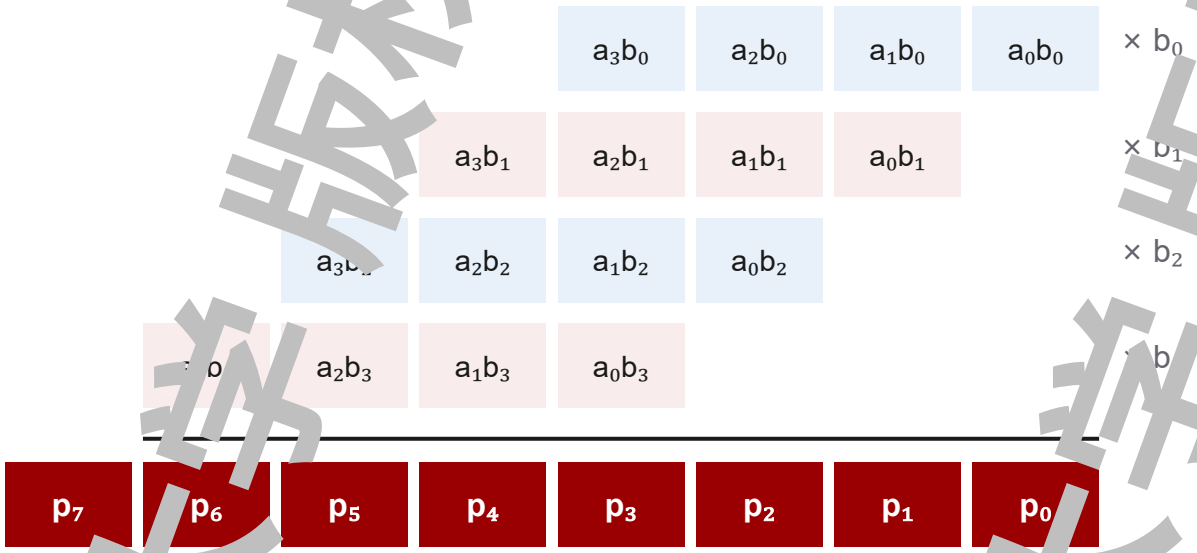
AI 输出检查清单

1. **索引计算**
   J = (i / D)·D − 1（整数除法），仅当 i / D 为奇数时合并
2. **扇出标注**
   让 AI 输出节点扇出表，驱动尺寸与缓冲决策
3. **尺寸优化**
   ML 延迟模型快速评估候选，最终以 STA 为准
4. **混合搜索**
   在 Sklansky ↔ Kogge-Stone ↔ BK 空间中按 PPA 目标折中

4.1 常规阵列乘法器 – 结构

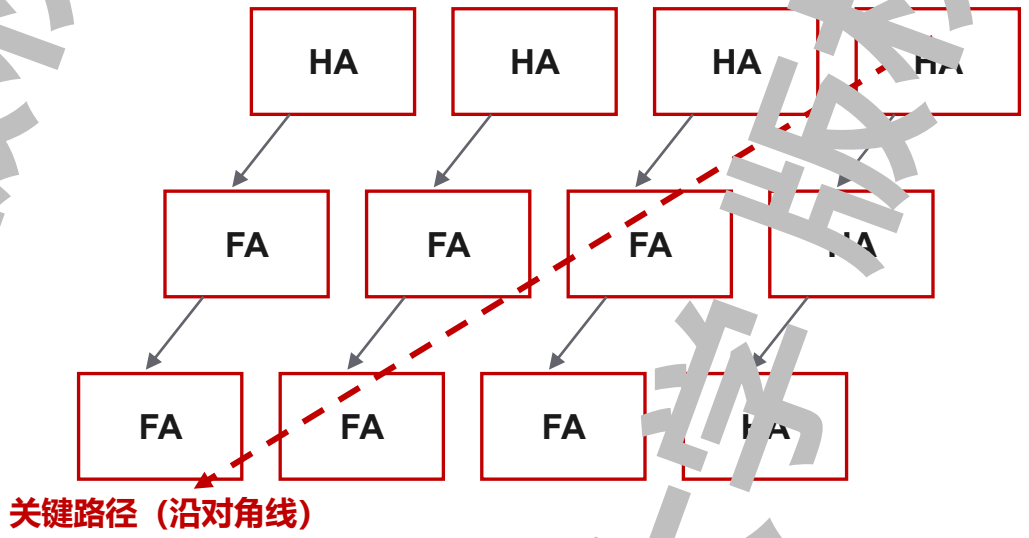
- 与门阵列产生部分积，加法器阵列逐行累加

① 部分积生成（4 × 4）



- N × N 个与门并行产生全部部分积
- 第 j 行部分积左移 j 位（权重 2^j）

② 加法器阵列



- 每行一级加法器，进位同时沿行和列传播
- 结构规则、易于版图拼接

4.2 常规阵列乘法器 – 延迟与面积

- 面积 $O(N^2)$、延迟 $O(N)$：多条几乎等长的关键路径需要同时优化

N^2

个（部分积）

$\approx N(N-2)$

个

$O(N)$

关键路径

关键路径估计

$$t \approx [(N-1) + (N-2)]\cdot t_{carry} + (N-1)\cdot t_{sum} + t_{and}$$

阵列中存在大量长度相近的路径，只加快 t_{carry} 或 t_{sum} 其中之一收益有限。

- 有符号数：Baugh–Wooley 把负权部分转变成正权项 + 常数修正
- 进位保留阵列：进位斜向下传，最后一行用快速加法器
- 适合中小位宽、对规则版图要求高的场景
- 更大位宽 → Booth 减少行数 + Wallace 树压缩

4.3 AI辅助：阵列乘法器的验证策略

- 乘法器最难的不是生成而是验证：按位宽选择穷举、随机或形式方法

提示词 Prompt

为 8×8 无符号阵列乘法器生成穷举测试平台，与行为级 a*b 逐一比较，并统计错误数。

LLM 生成的 Verilog

```
// AI 生成的穷举测试平台 (8×8, 共 65536 组)
module tb;
  reg  [7:0]  a, b;
  wire [15:0] p;
  integer i, j, err = 0;
  array_mul #(.N(8)) dut (.a(a), .b(b), .p(p));
  initial begin
    for (i = 0; i < 256; i = i + 1)
      for (j = 0; j < 256; j = j + 1) begin
        a = i; b = j; #1;
        if (p !== a * b) err = err + 1;
      end
    $display("errors = %0d", err);
    $finish;
  end
endmodule
```

AI 输出检查清单

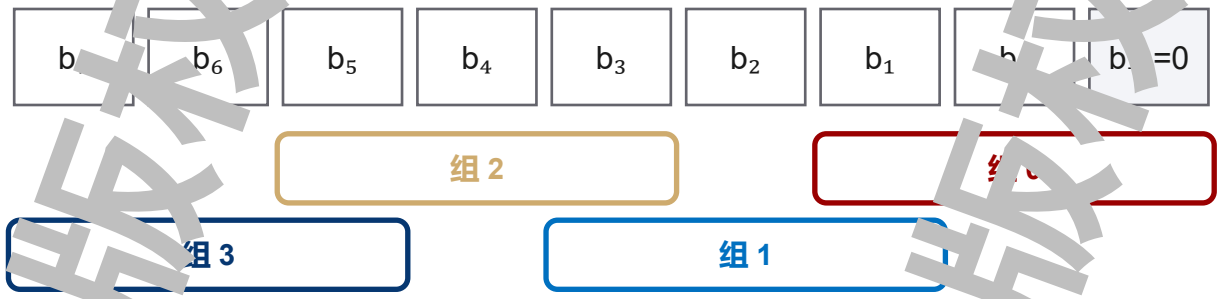
1. **穷举可行性**
   8×8 可穷举；32×32 只能随机 + 形式 / 代数方法
2. **形式验证难点**
   乘法器的 SAT 等价检查很难，宜用结构化参考或代数重写
3. **有符号处理**
   检查 $signed 与 Baugh–Wooley 的修正常数
4. **输出位宽**
   积为 2N 位，防止 AI 在中间表达式中截断高位

4.4 Booth 编码乘法器 – Radix-4 原理

- 乘数每 2 位重编码为 {−2, −1, 0, +1, +2}，部分积数量减半

b_{2i+1}	b_{2i}	b_{2i-1}	编码值	部分积
0	0	0	0	0
0	0	1	+1	+M
0	1	0	+1	+M
0	1	1	+2	+2M
1	0	0	−2	−2M
1	0	1	−1	−M
1	1	0	−1	−M
1	1	1	0	0

乘数 B 的重叠分组（8 位）



- 每次看 3 位（与相邻组重叠 1 位），决定一个部分积
- 部分积数量 N → N/2（无符号乘数需再补 1 行）
- ±M、±2M 只需移位与取反，无需真正的乘法
- Radix-8 部分积更少，但需预先计算 3M（额外加法器）

4.5 Booth 编码乘法器 – 部分积生成与符号扩展

- 编码器产生 one / two / neg，选择器输出部分积，[illegible]常数技巧消除长符号扩展

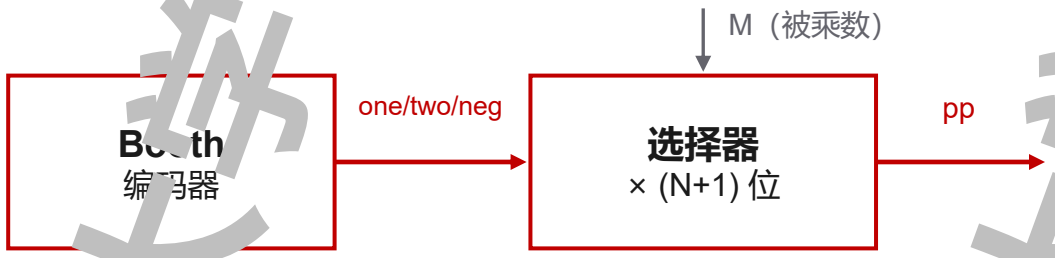
Booth 编码与选择

$$one = b_{2i} \oplus b_{2i-1}$$

$$two = b_{2i+1}\cdot\neg b_{2i}\cdot\neg b_{2i-1} + \neg b_{2i+1}\cdot b_{2i}\cdot b_{2i-1}$$

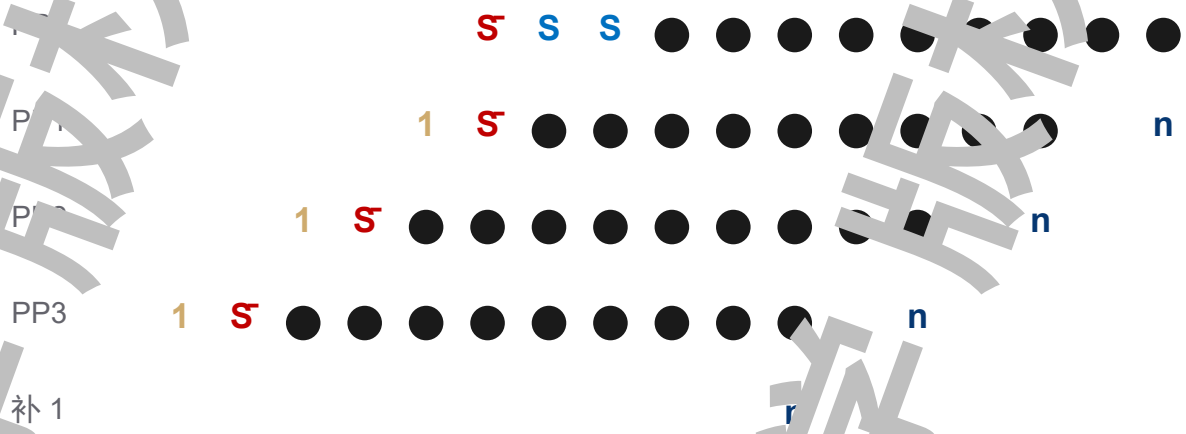
$$neg = b_{2i+1}$$

$$pp_j = (one\cdot m_j + two\cdot m_{j-1}) \oplus neg$$



取反只完成“按位取反”，另需在下一行对应权重补一个 neg 位实现 +1。

8×8 Radix-4 部分积阵列（符号扩展技巧）



- S：部分积符号位；n：上一行取反时补的 +1
- “S̄ S S / 1 S”常数代替每行长串符号位扩展

4.6 AI辅助：Booth 编码器的生成与纠错

- Booth 的错误几乎都集中在符号与 +1 上，检查清单必须覆盖这些点

提示词 Prompt

生成 Radix-4 Booth 部分积模块：输入被乘数 m 与 3 位分组，输出 N+1 位部分积和 neg 信号。

LLM 生成的 Verilog

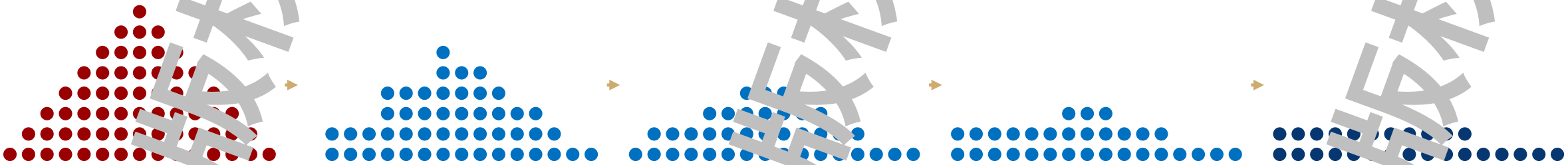
```
// Radix-4 Booth 一组的编码与部分积选择
module booth_pp #(parameter N = 8) (
  input  [N-1:0] m,      // 被乘数（有符号）
  input  [2:0]   grp,    // {b[2i+1], b[2i], b[2i-1]}
  output [N:0]   pp,     // N+1 位部分积（尚未 +1）
  output         neg);
  wire one = grp[1] ^ grp[0];
  wire two = (~grp[2] & ~grp[1] & ~grp[0]) |
             ( grp[2] &  grp[1] &  grp[0]);
  assign neg = grp[2];
  wire [N:0] mx  = {m[N-1], m};              // 符号扩展 1 位
  wire [N:0] sel = ({(N+1){one}} & mx) |
                   ({(N+1){two}} & {mx[N-1:0], 1'b0});
  assign pp = sel ^ {(N+1){neg}};            // +1 由 neg 位补
endmodule
```

AI 输出检查清单

1. **+1 补码**
   neg 位必须放到下一行的对应权重，AI 常遗漏
2. **最后一组**
   无符号乘数需补 0 扩展出额外一组
3. **符号扩展常数**
   检查“S S S / 1 S”的位置是否正确
4. **验证**
   与 $signed(a) * $signed(b) 等价；8×8 可穷举

4.7 Wallace 树乘法器 – 结构

- 用 3:2 压缩器（全加器）并行压缩部分积，行数按约 2/3 的比例递减



初始部分积
最高 8 行

第 1 级后
最高 6 行

第 2 级后
最高 4 行

第 3 级后
最高 3 行

第 4 级后
最高 2 行

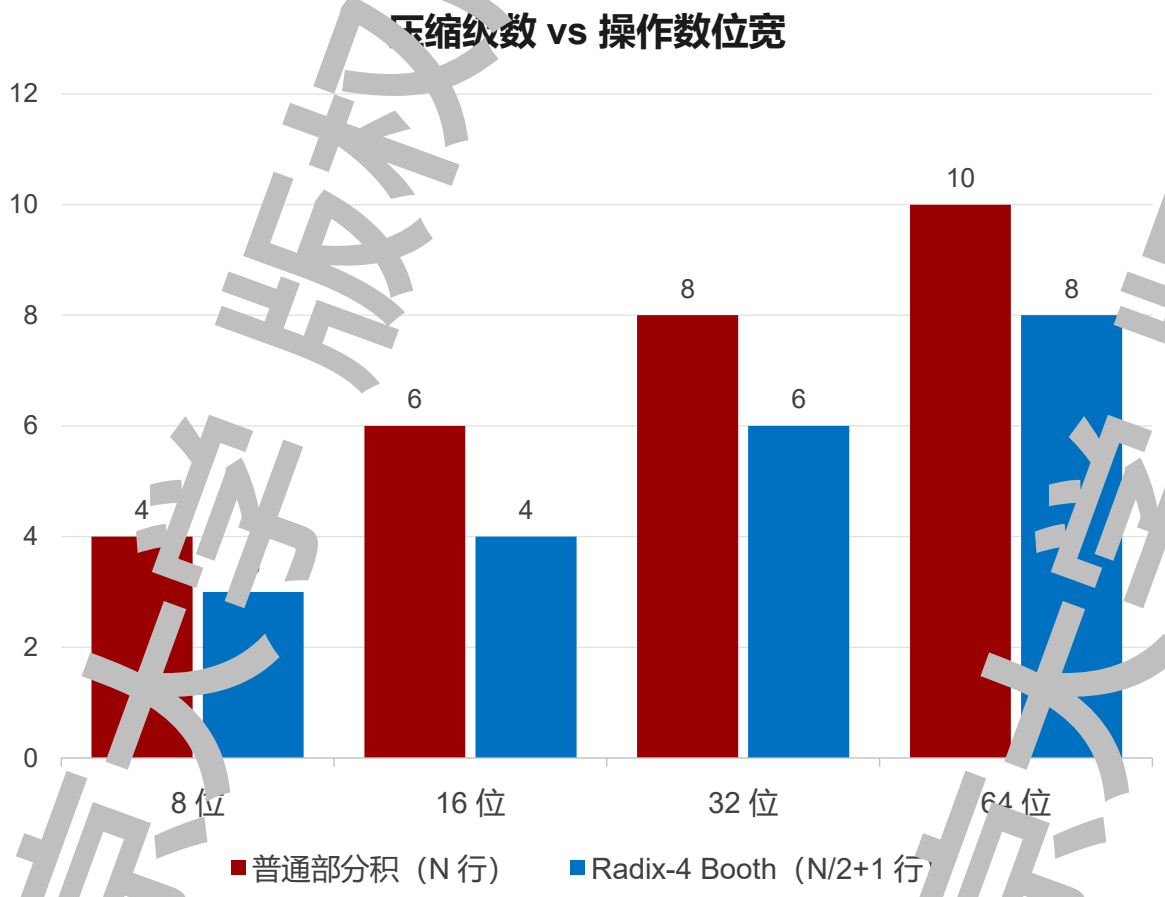
3:2 压缩器 = 全加器

同一列 3 位 → 本列 1 位和
+ 高一列 1 位进位
进位不横向传播 → 各列并行

- 8×8：8 行部分积经 4 级压缩（8 → 6 → 4 → 3 → 2）剩下两行
- 最后两行交给快速进位传播加法器（如 Kogge–Stone）得到乘积
- 压缩级数与操作数按 $O(\log_{3/2} N)$ 增长，远快于阵列乘法器的 O(N)

4.8 Wallace 树乘法器 – 级数分析与 Dadda / 4.2 压缩

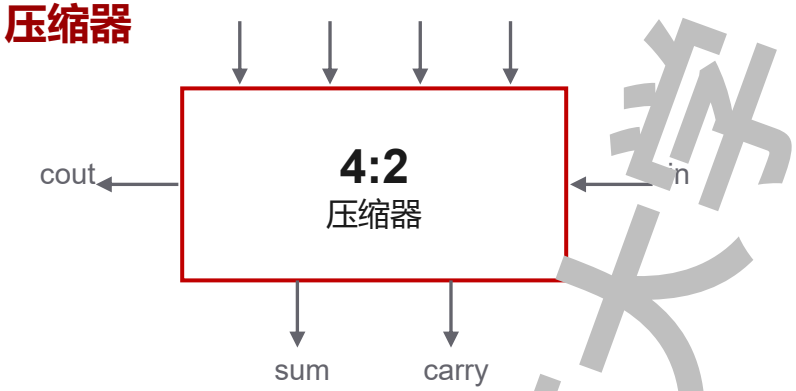
- Booth 减行 + 树形压缩是现代高性能乘法器的标准结构



Wallace 与 Dadda

- Wallace：每级尽可能多地压缩，速度快但结构不规则
- Dadda：只在必要时压缩（行数上限 2,3,4,6,9,13…），乘法器更少、级数相同

4:2 压缩器



5 入 3 出，cout 不依赖 cin；延迟约 1 个全加器，树更规则

4.9 AI辅助：压缩器树的强化学习优化

- 压缩器的数量与安放构成巨大离散空间，RL 以综合结果为奖励进行搜索

代表性工作：RL-MUL（DAC 2023）等，将乘法器结构编码为矩阵并以深度强化学习优化

1 业务表示

每列的 3:2 / 2:2 压缩器数量构成矩阵；列高约束保证每一步都是合法结构

2 联合优化

压缩器树与最终进位传播加法器的前缀结构一起搜索，避免局部最优

3 时序细节

3:2 的 sum / carry 到达时间不同，按到达时间分配端口进一步降低延迟

5.1 复杂移位器 – 功能与结构分类

- 按功能分为逻辑 / 算术 / 循环，按实现分为阵列、对数、漏斗三类

功能分类（a = 1011 0010，k = 2）

操作	含义	结果
SLL	逻辑左移，低位补 0	1100 1000
SRL	逻辑右移，高位补 0	0010 1100
SRA	算术右移，高位补符号位	1110 1100
ROR	循环右移，移出位回到高位	1010 1100

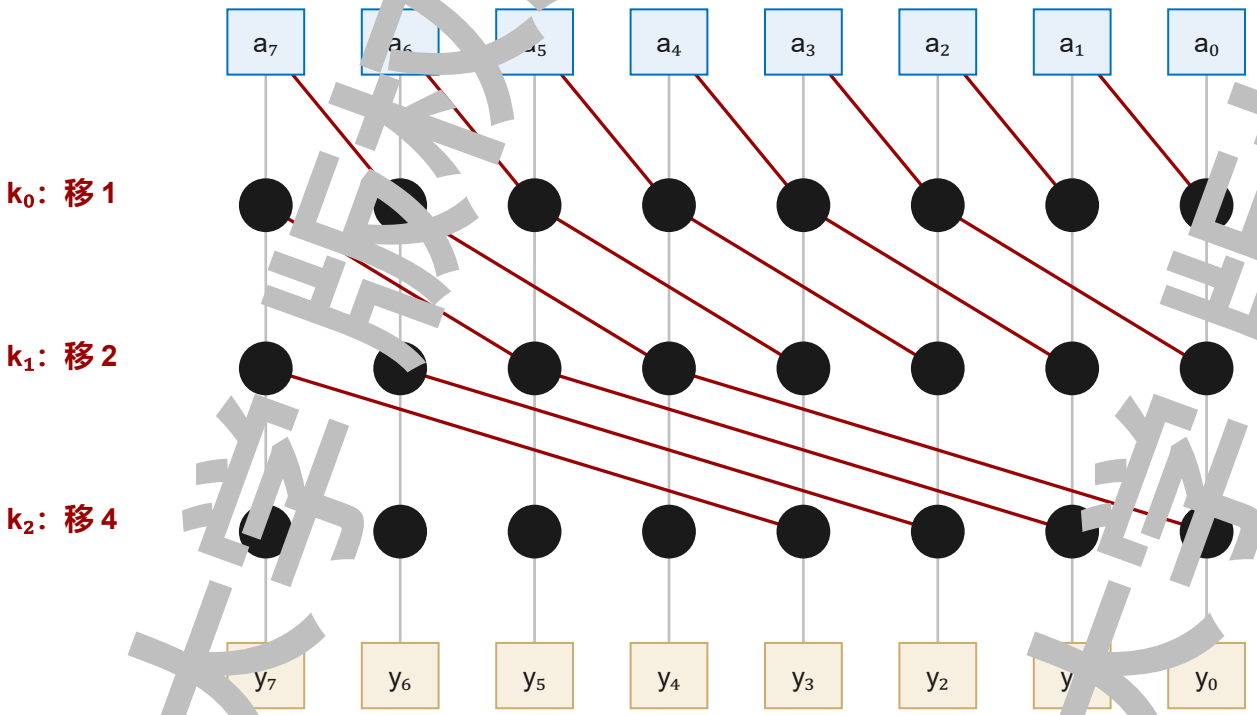
结构分类

结构	级数	规模	特点
阵列（桶形）	1	N^2 个开关	延迟小，电容与面积大
对数移位器	$\log_2 N$	$N \cdot \log_2 N$ 个 MUX	面积小，最常用
漏斗移位器	$\log_2 N$	2N−1 位输入窗口	一个结构支持全部四种操作

- 移位量 k 是控制信号，数据通路本身没有运算
- 延迟主要由 MUX 级数与导线负载决定
- 处理器 ALU、浮点对阶 / 规格化中大量使用
- 左移可通过位反转或窗口偏移复用右移硬件

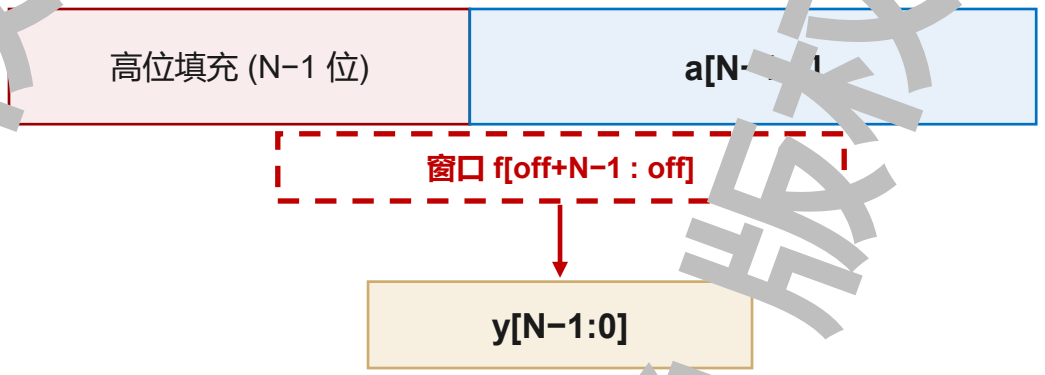
5.2 对数移位器与漏斗移位器

- 对数移位器按 k_i 对各位分级移动 1、2、4… 位；漏斗移位器从 2N−1 位窗口中截取



8 位右移：3 级 2:1 MUX（黑点），每级由 k 的一位控制，越界输入补 0 或符号位

漏斗移位器



操作	漏斗输入 f	偏移 off
SRL	{0…0, a}	k
SRA	{s…s, a}	k
ROR	{a[N-2:0], a}	k
SLL	{a, 0…0}	¬k (= N-1-k)

5.3 AI辅助：统一漏斗移位器生成

- 一个模板覆盖四种移位：AI 负责写模板，人负责检查边界

提示词 Prompt

用漏斗结构实现 32 位移位器，支持 SLL / SRL / SRA / ROR，只使用一个右移端口。

LLM 生成的 Verilog

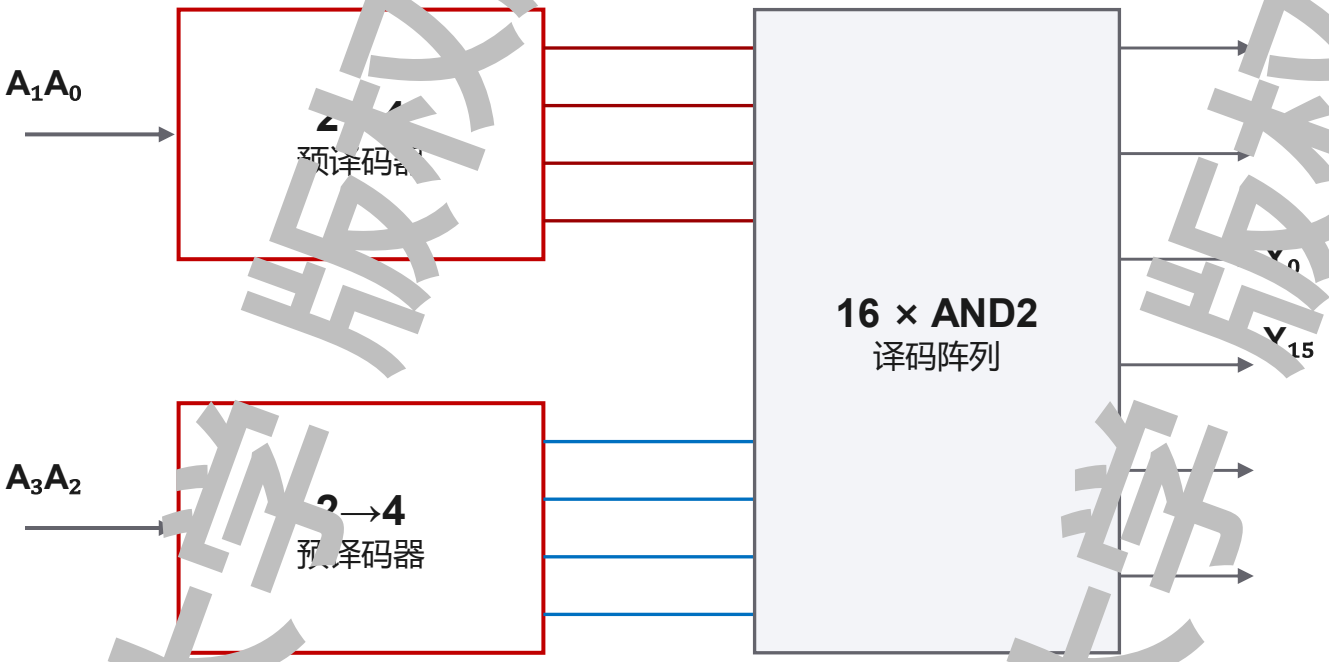
```
module funnel_shifter #(parameter N = 32, S = $clog2(N)) (
  input  [N-1:0] a,
  input  [S-1:0] k,
  input  [1:0]   op,     // 00 SLL, 01 SRL, 10 SRA, 11 ROR
  output [N-1:0] y);
  reg [2*N-1:0] f;
  reg [S-1:0]   off;
  always @(*) begin
    case (op)
      2'b00:   begin f = {a, {(N-1){1'b0}}};    off = ~k; end
      2'b01:   begin f = {{(N-1){1'b0}}, a};    off =  k; end
      2'b10:   begin f = {{(N-1){a[N-1]}}, a};  off =  k; end
      default: begin f = {a[N-2:0], a};         off =  k; end
    endcase
  end
  wire [2*N-1:0] t = f >> off;   // 综合为 log2(N) 级 MUX
  assign y = t[N-1:0];
endmodule
```

AI 输出检查清单

1. **SLL 复用**
   off = ¬k 等价于 N−1−k（要求 N 为 2 的幂）
2. **SRA 符号位**
   必须复制 a[N−1]；若改用 >>> 需声明 signed
3. **边界用例**
   k = 0、k = N−1、a 为最小负数、全 0、全 1
4. **结构确认**
   检查综合报告：f >> off 应映射为 $\log_2 N$ 级 MUX

6.1 译码器 – 二进制译码与预译码

- n 位输入 → 2^n 路 one-hot 输出；预译码把高扇入[illegible]成两级小门



每条预译码线只驱动 4 个 AND2；直接实现则每个地址位（原/反）要驱动 8 个 AND4

方案	门数	最大扇入
直接实现	16 × AND4	4
两级预译码	8 × AND2 + 16 × AND	2

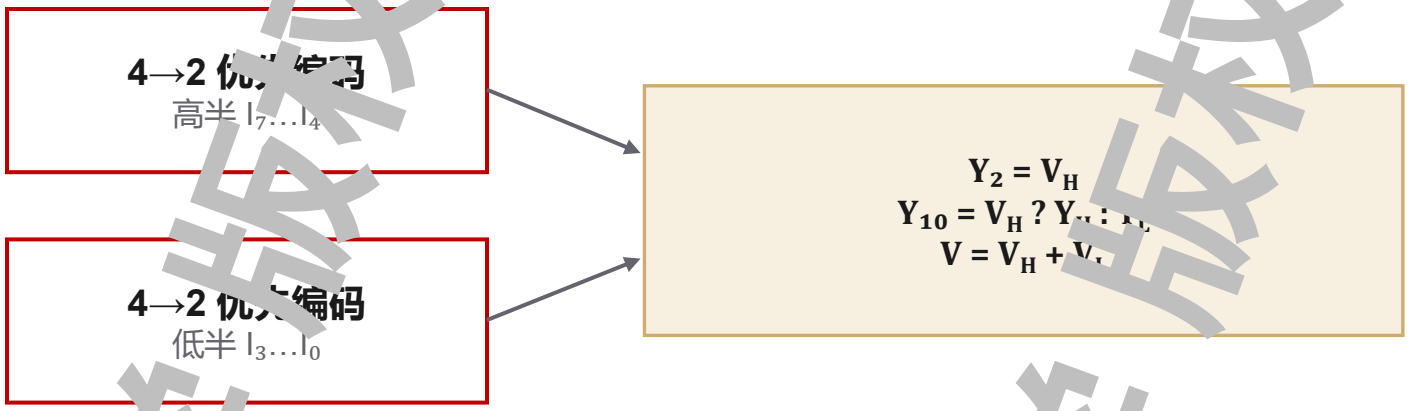
- one-hot 输出：任意时刻只有一路有效
- SRAM 行译码常用多级预译码 + 字线驱动器
- 预译码减小扇入、分摊输入负载，[illegible]更于按逻辑努力分配尺寸

6.2 编码器与优先编码器

- 优先编码器输出最高位有效请求的编号，可递归二分构成 O(log N) 树

$I_7 \dots I_0$	$Y_2Y_1Y_0$	V
1xxxxxxx	111	1
01xxxxxx	110	1
001xxxxx	101	1
0001xxxx	100	1
00001xxx	011	1
000001xx	010	1
0000001x	001	1
00000001	000	1
00000000	xxx	0

树形实现（8 → 3）



- 普通编码器（one-hot → 二进制）：Y_k 是所有第 k 位为 1 的输入之“或”
- 优先编码器：高位优先；每层一个 MUX，深度 O(log N)
- 应用：总线仲裁、中断控制、浮点规格化中的前导零计数（LZC）

6.3 AI辅助：编码器 / 译码器的生成与验证

- 描述简单但陷阱多：锁存器、X 传播与综合出的长链组合

提示词 Prompt

生成参数化优先编码器（高位优先），输出编号 idx 与有效标志 valid，要求纯组合、无锁存器。

LLM 生成的 Verilog

```
module prio_enc #(parameter N = 8, W = $clog2(N)) (
  input      [N-1:0] req,
  output reg [W-1:0] idx,
  output             valid);
  integer i;
  always @* begin
    idx = {W{1'b0}};                  // 默认赋值：避免锁存器
    for (i = 0; i < N; i = i + 1)
      if (req[i]) idx = i;            // 后赋值覆盖 → 高位优先
  end
  assign valid = |req;
endmodule
```

AI 输出检查清单

1. **默认赋值**
   always 开头先给 idx 赋默认值，否则推断出锁存器
2. **casez 而非 casex**
   casex 会把输入中的 X 当无关项，掩盖仿真问题
3. **结构检查**
   循环描述可能综合为线性链；大 N 时要求 AI 生成树形
4. **穷举验证**
   N = 8 只有 256 种输入，可全部与参考模型比对

6.4 比较器 – 三种类型实现

- 相等比较用 XNOR + AND 树；大小比较可复用减法器，或用专门的树形结构

相等比较器

$a_i \odot b_i$ (XNOR) × N

↓

AND 归约树

↓

EQ

- 深度 $\log_2 N$ 级
- 只判断相等，面积最小

减法型比较器

A + ¬B + 1

↓

加法器进位 cout

↓

cout = 1 ⇔ A ≥ B

- 复用 ALU 加法器，几乎零额外面积
- 延迟 = 加法器进位延迟

树形比较器

每位 (g_i, e_i)

↓

(G, E) 逐级合并

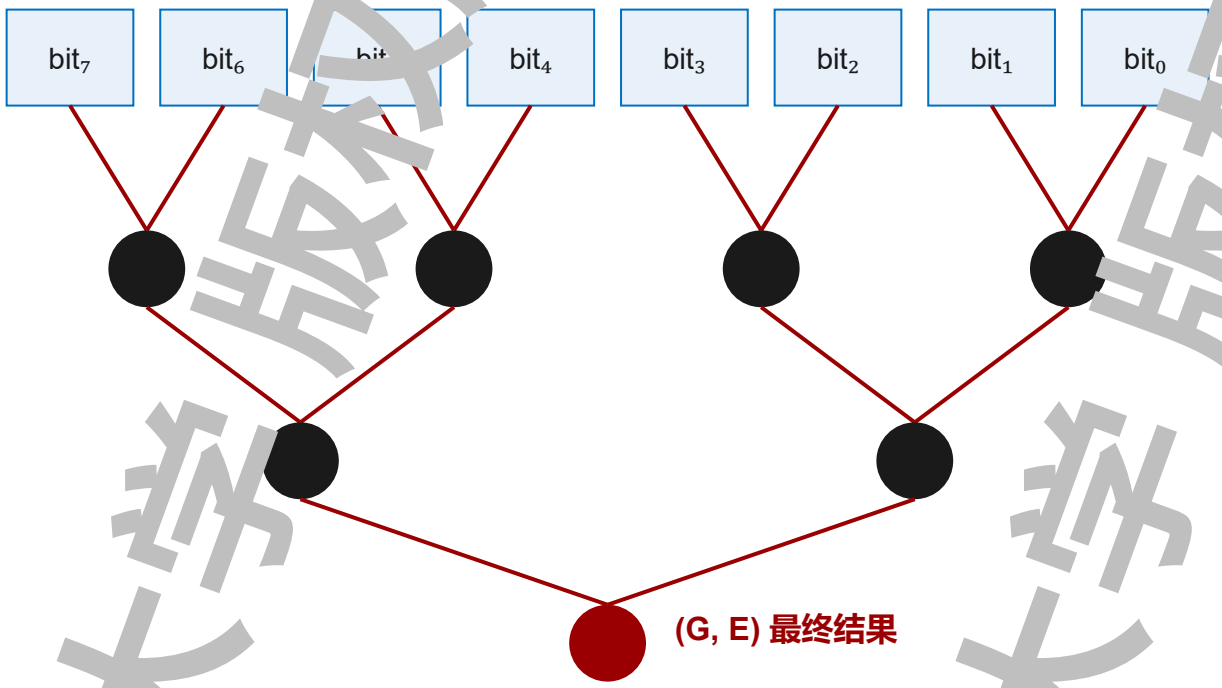
↓

GT / EQ / LT

- $g_i = a_i \cdot \neg b_i$，$e_i = a_i \odot b_i$
- 深度 O(log N)，面积 O(N)

6.5 树形比较器 – 结构与有符号处理

- (G, E) 合并运算满足结合律，与前缀加法器的 (G, P) 完全同构



叶节点：$g_i = a_i \cdot \neg b_i$，$e_i = a_i \odot b_i$；每个黑点做一次 (G, E) 合并，高位在左

$$(G, E) = (G_H + E_H \cdot G_L,\ E_H \cdot E_L)$$

$A > B \Leftrightarrow G = 1$；$A = B \Leftrightarrow E = 1$；$A < B \Leftrightarrow \neg G \cdot \neg E$

- 与前缀加法器 (G, P) 同构 → 可复用前缀网络与 AI 前缀图搜索成果
- 有符号比较：两数最高位取反后按无符号比较（补码 → 偏移码）
- 深度 $\log_2 N$，面积 O(N)，适合宽位宽比较

6.6 AI辅助：比较器生成与数据通路共享

- 行为级描述交给综合器；AI 负责补齐有符号语义并发现可共享的硬件

提示词 Prompt

生成同时支持有符号/无符号的 32 位比较器，输出 lt / eq / gt，要求用最高位取反技巧处理有符号数

LLM 生成的 Verilog

```
module cmp #(parameter N = 32) (
  input  [N-1:0] a, b,
  input          sgn,          // 1: 有符号比较
  output         lt, eq, gt);
  // 有符号：最高位取反，转换为无符号比较
  wire [N-1:0] ax = {a[N-1] ^ sgn, a[N-2:0]};
  wire [N-1:0] bx = {b[N-1] ^ sgn, b[N-2:0]};
  assign eq = (a == b);
  assign gt = (ax > bx);
  assign lt = ~eq & ~gt;
endmodule
```

AI 输出检查清单

1. **有符号边界**
   -2^{N-1} 与 $2^{N-1}-1$ 互比，以及相等的负数
2. **结构控制**
   行为级 > 由综合器选结构；需要 log 深度时要求 AI 写树形
3. **资源共享**
   AI 分析数据通路，让比较与减法共用同一个加法器
4. **验证**
   随机 + 边界用例，并与 $signed 参考模型形式等价

总结：组合逻辑分类与 AI 辅助要点

- 结构决定 PPA 上限，AI 在“生成—验证—优化”三个环节提升效率

类别	典型结构	延迟	面积	AI 辅助重点
静态 CMOS 门	NAND / NOR / AOI / OAI	逻辑努力模型	晶体管数	等价检查、尺寸建议
Ripple-Carry	全加器级联	O(N)	O(N)	参数化生成、最坏向量
Carry-Select	双路预计算 + MUX	O(√N)	≈ 2× RCA	分块方案搜索
Carry-Lookahead	分组 (G, P) + LCU	O(log N)	O(N)	扇入约束下展开
PGK / 前缀	预处理-前缀-后处理	O(log N)	由前缀图决定	前缀图 RL 搜索
Kogge-Stone	跨度倍增，扇出 2	log N 级	$N \cdot \log_2 N$ 单元	布线感知、稀疏化
Sklansky	分治，高扇出	$\log_2 N$	$(N/2) \cdot \log_2 N$	尺寸与扇出优化
阵列乘法器	AND 阵列 + 加法阵列	O(N)	$O(N^2)$	穷举 / 形式验证
Booth 乘法器	Radix-4 重编码	部分积减半	编码器开销	符号与 +1 纠错
Wallace 树	3:2 / 4:2 压缩树	O(log N)	$O(N^2)$	压缩器树 RL 优化
移位器	对数 / 漏斗	$\log_2 N$ 级	$N \cdot \log_2 N$ MUX	统一漏斗模板
编码 / 译码器	预译码、树形优先编码	O(log N)	O(N) / $O(2^n)$	无效码、树形化
比较器	相等 / 减法 / 树形	O(log N)	O(N)	符号与边界、资源共享