北京大学
PEKING UNIVERSITY

基于AI的数字系统设计

第五讲：AI辅助组合逻辑设计

主讲：陶耀宇

2026年秋季

北京大学版权所有

北京大学
PEKING UNIVERSITY

AI辅助组合逻辑设计

北京大学版权所有

组合逻辑分类与 AI 辅助要点

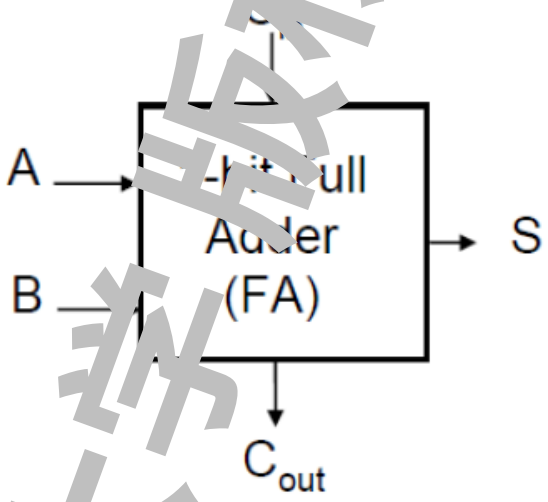
- 结构决定 PPA 上限，AI 在“生成—验证—优化”三个环节提升效率

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)	扇入约束下的展开
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	log₂N 级	N·log₂N 单元	布线感知、稀疏化
Sklansky	分治，高扇出	log₂N 级	(N/2)·log₂N	尺寸与扇出优化
阵列乘法器	AND 阵列 + 加法阵列	O(N)	O(N²)	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(N²)	压缩器树 RL 优化
移位器	对数 / 漏斗	log₂N 级	N·log₂N MUX	统一漏斗模板
编码器	预译码、树形优先编码	O(log N)	O(N) / O(2ⁿ)	无效输入、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

北京大学版权所有

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]



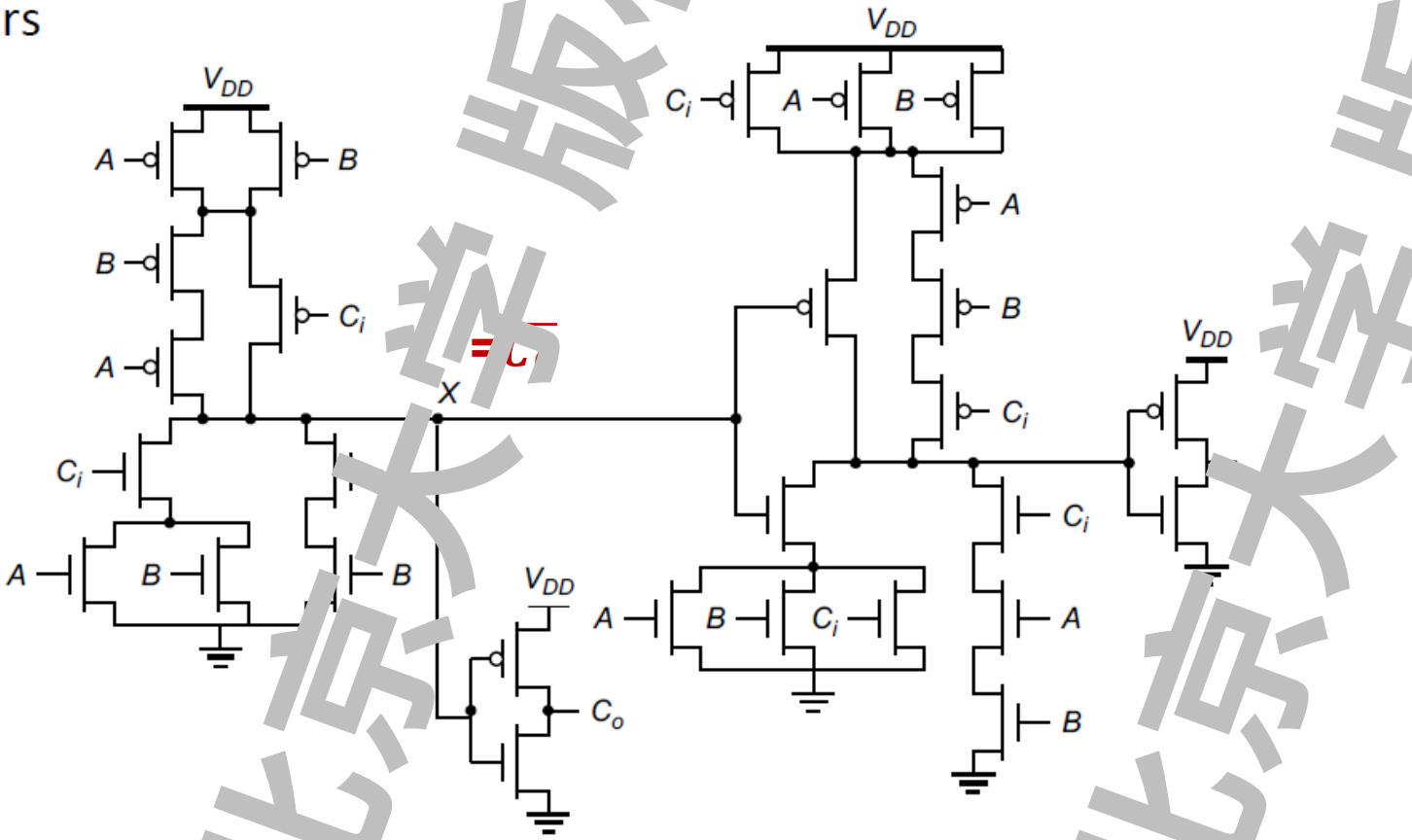
A	B	C_{in}	C_{out}		carry status
0	0	0	0	0	kill
0	0	1	0		kill
0	1	0	0		propagate
0	1	1	1	0	propagate
1	0	0		1	propagate
1	0	1		0	propagate
1	1	0	1	0	generate
1	1	1	1	1	generate

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

单比特加法器逻辑设计

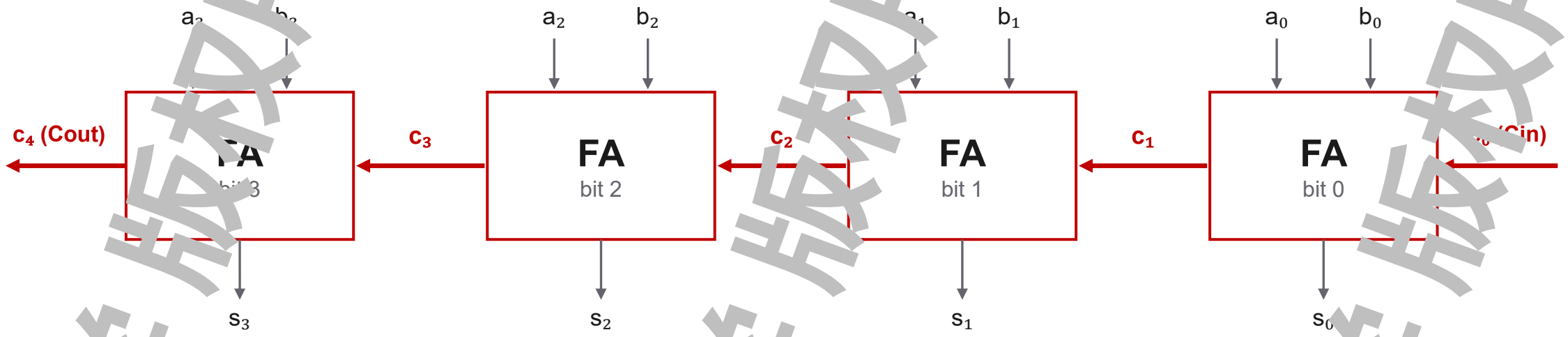
经典静态逻辑电路

- 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 逐位串行穿过每一个全加器（红色路径）

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

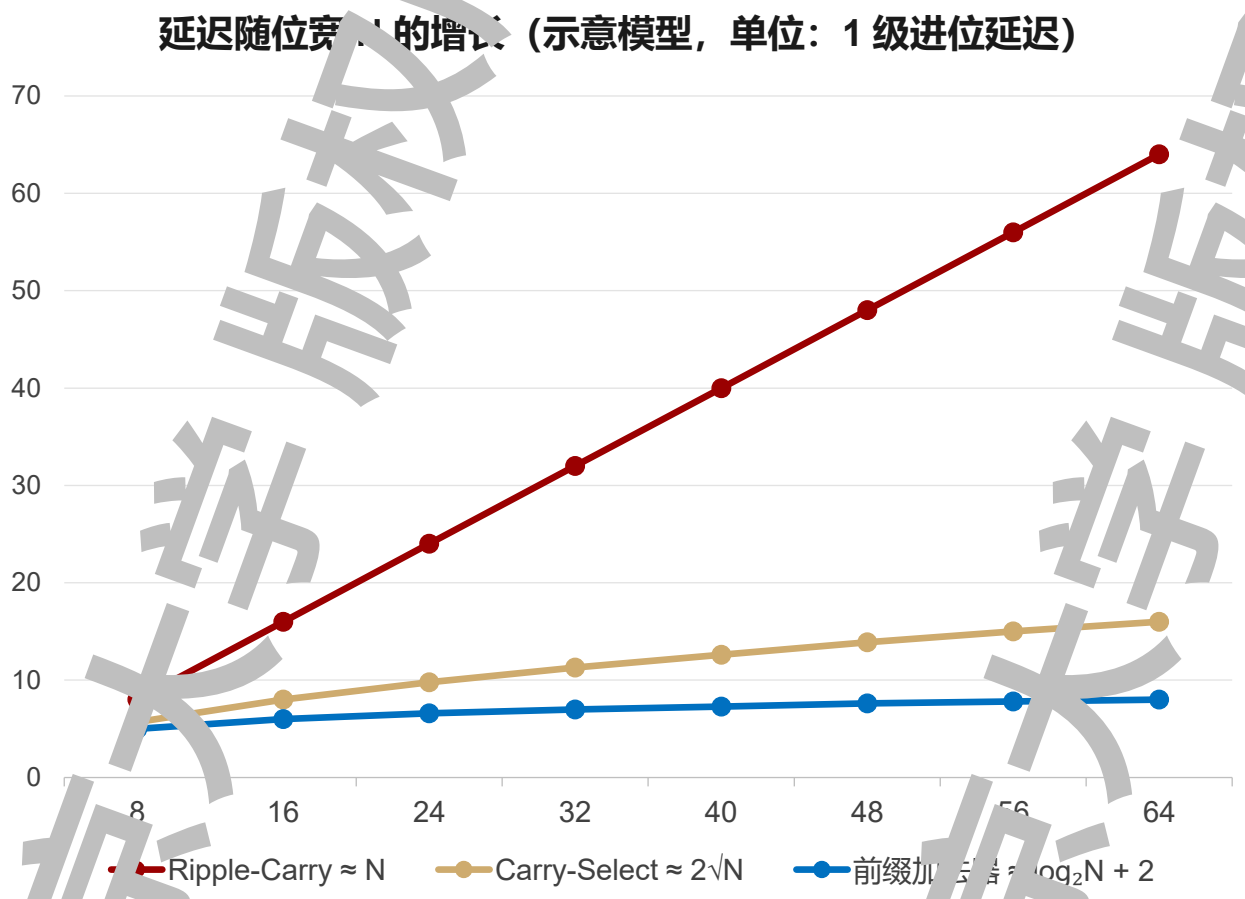
$$c_{i+1} = a_i b_i + c_i(a_i \oplus b_i)$$

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

- 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
- 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

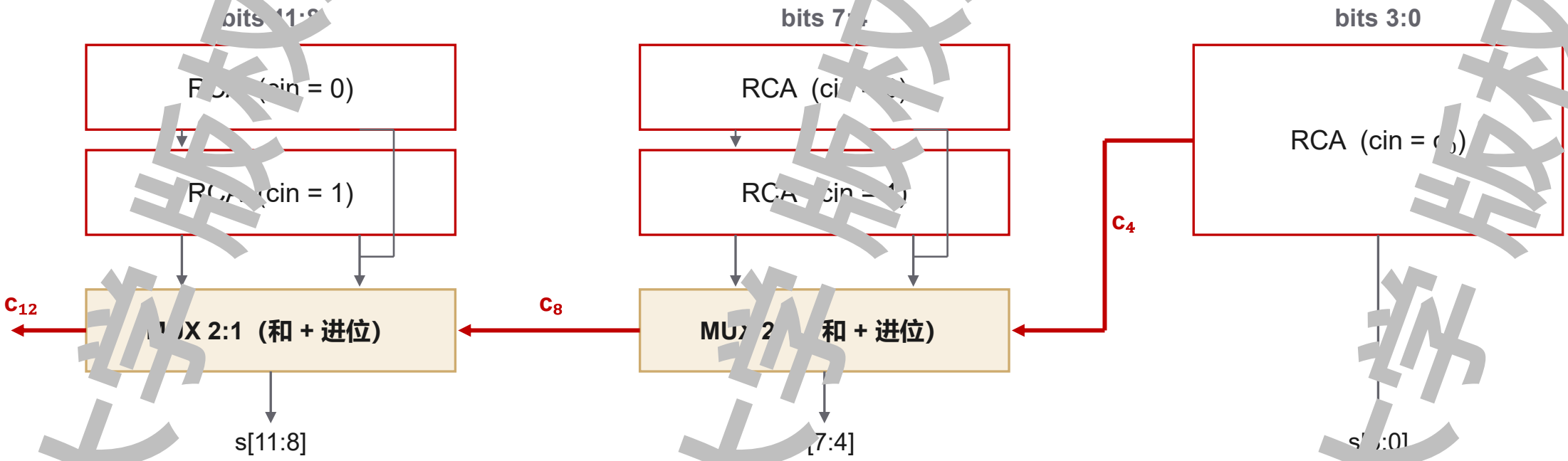
```
module rca #(parameter N = 16) (
  input  [N-1:0] a, b,
  input          cin,
  output [N-1:0] s,
  output         cout
);
  wire [N:0] c;
  assign c[0] = cin;
  genvar i;
  generate for (i = 0; i < N; i = i + 1) begin : g_fa
    assign s[i]   = a[i] ^ b[i] ^ c[i];
    assign c[i+1] = (a[i] & b[i])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

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

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

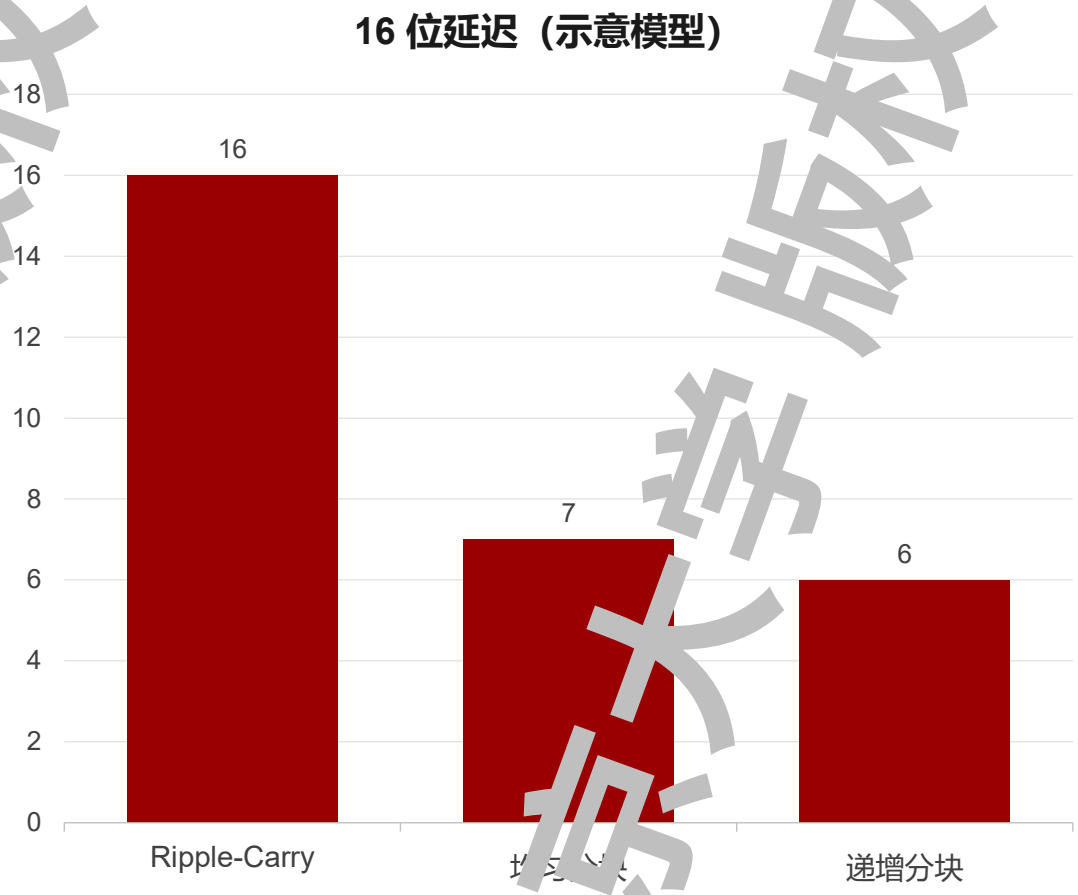
均匀分块 4-4-4-4



递增分块 5-4-3-2-2

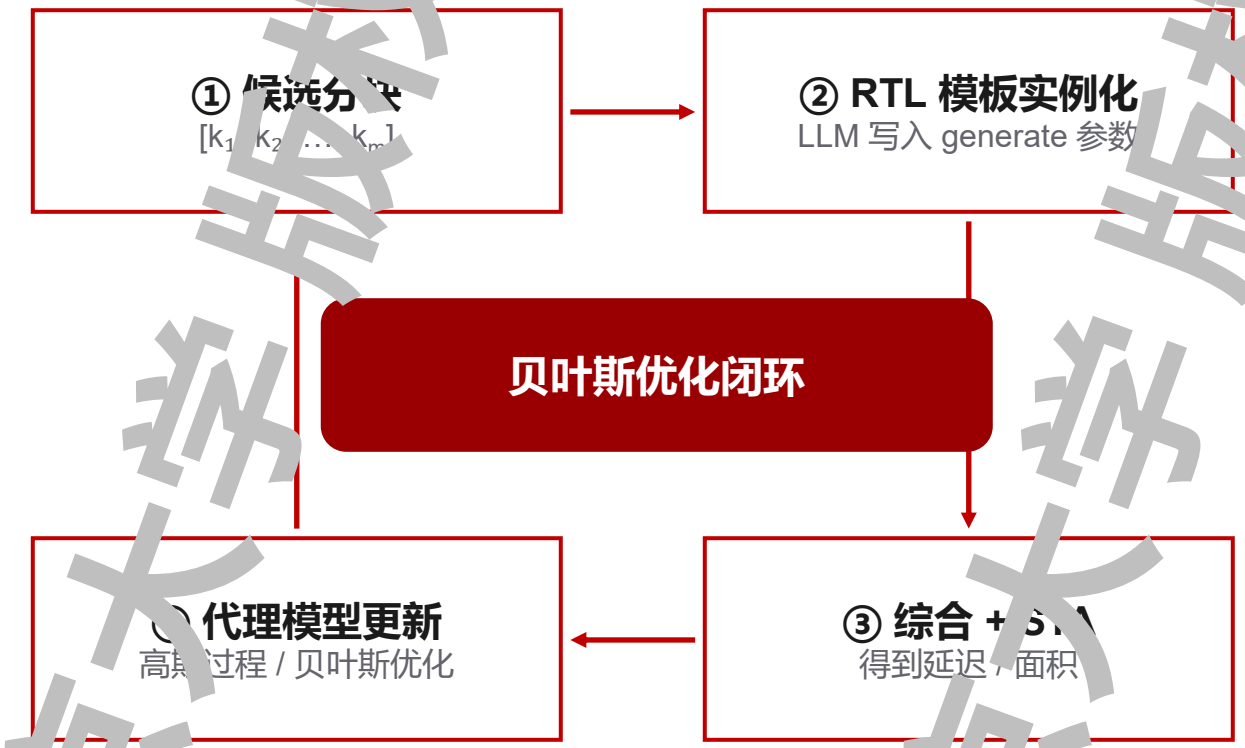


- 递增块：低位块小、高位块大，结果就绪时间与 MUX 选择信号到达时间对齐
- 代价：面积约为 RCA 的 2 倍，外加每位一个 MUX
- 设计可以让：**高位块内部计算延迟 = 前级进位传输延迟**，进位信号就像通过一条平滑的斜坡一样，无缝地一路选择过去，使最后一级输出结果的总延时达到最小。



3.6 AI辅助：Cluster-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 32763 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 a、b 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

$p_i = a_i \oplus b_i$ 传播

$c_{i+1} = g_i + p_i c_i$

$s_i = p_i \oplus c_i$

展开后的进位方程

$c_1 = g_0 + p_0 c_0$

$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$

$c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$

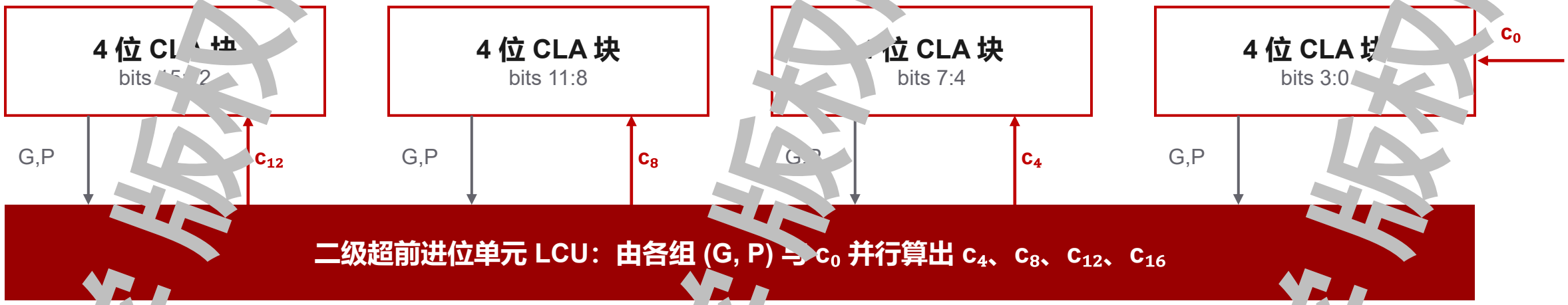
$c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$

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)



$$C_{\cdot} = {}_{3} + {}_{3}g_2 + p_3p_2g_1 + p_3p_2p_1g_0$$

$$P = p_3p_2p_1p_0$$

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

- 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 → 可级联构造
- “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 生成展开式的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：先用可复用的 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：组进位 + 组 (G, P)，每个门扇入 ≤ 4
module cla4
  input  [3:0] g, p,
  input        c0,
  output [3:1] c,
  outp         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);
  ass    G = g[3] | (p[3] & g[2]) | (p[3] & p[2] & g[1])
                  | (p[3] & p[2] & p[1] & g[0]);
  assign    = &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 $p = a \oplus b$ —— AI 常混用
4. **等价验证**
   16 位整体与 a + b 做形式等价检查

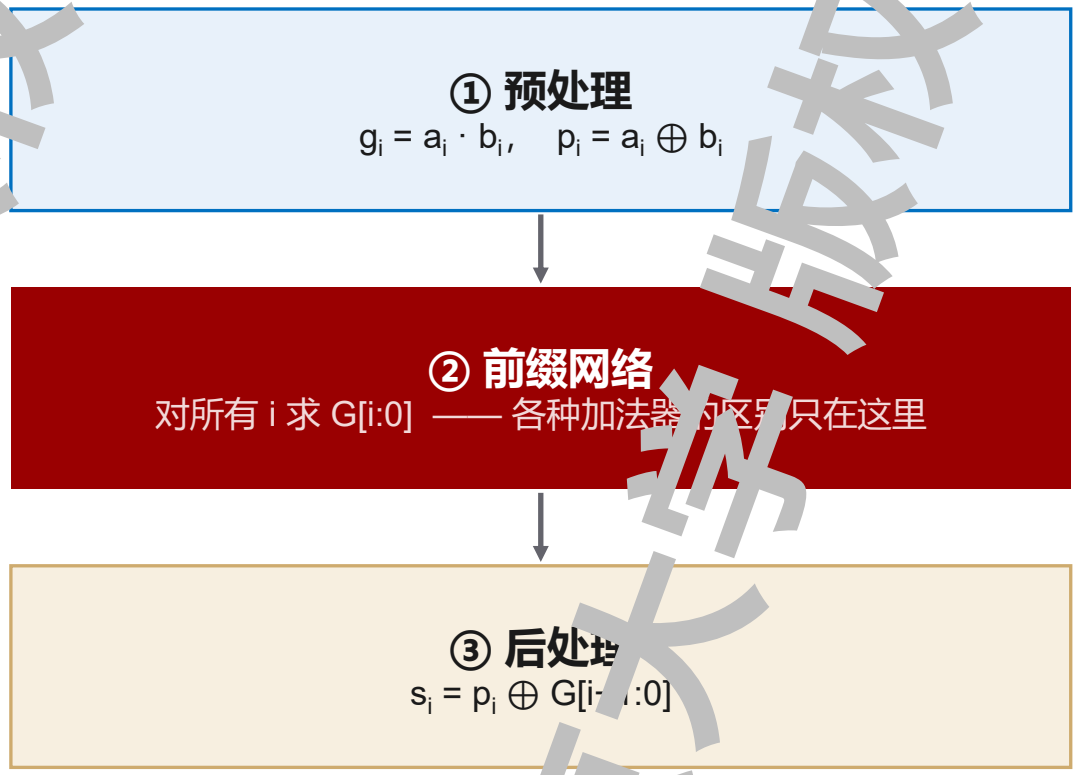
3.10 PGK 加法 - 进位的三种状态

- 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]

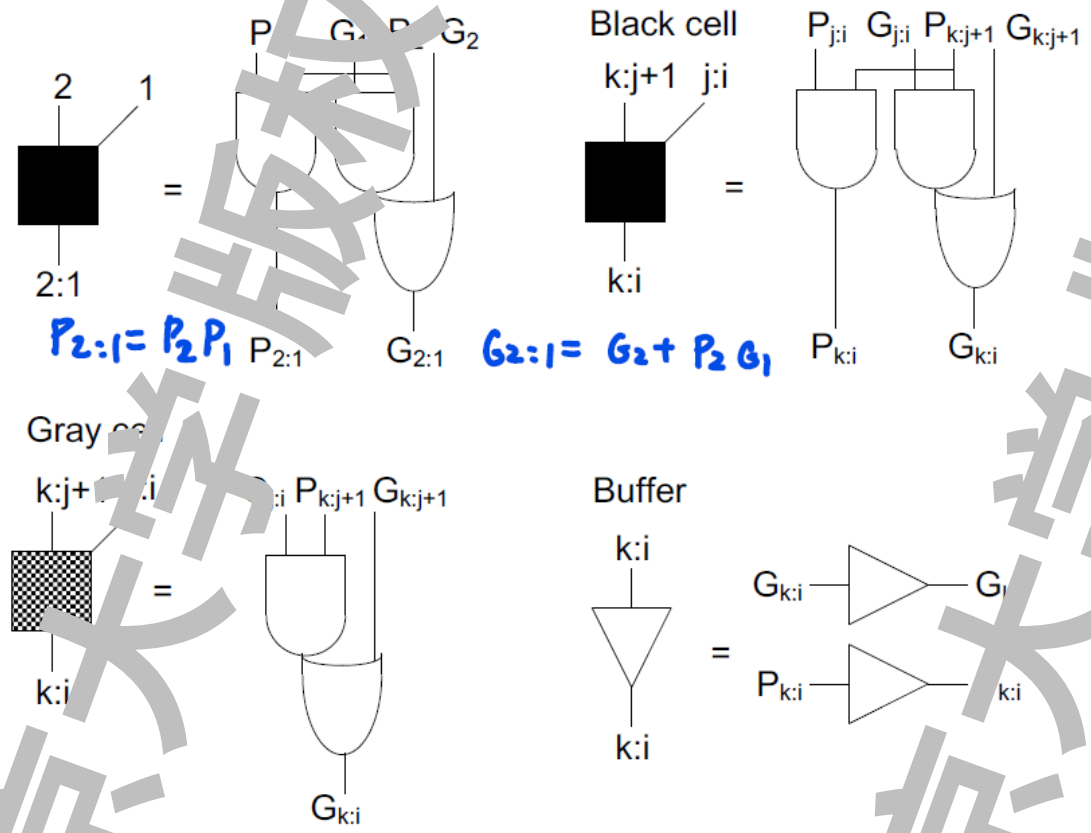
- G[i:j]：该区间自身产生进位；P[i:j] = 1：进位可穿过整个区间
- $c_{i+1} = G[i:0]$（把 c_0 视为第 −1 位的 G）
- $s_i = p_i \oplus G[i-1:0]$



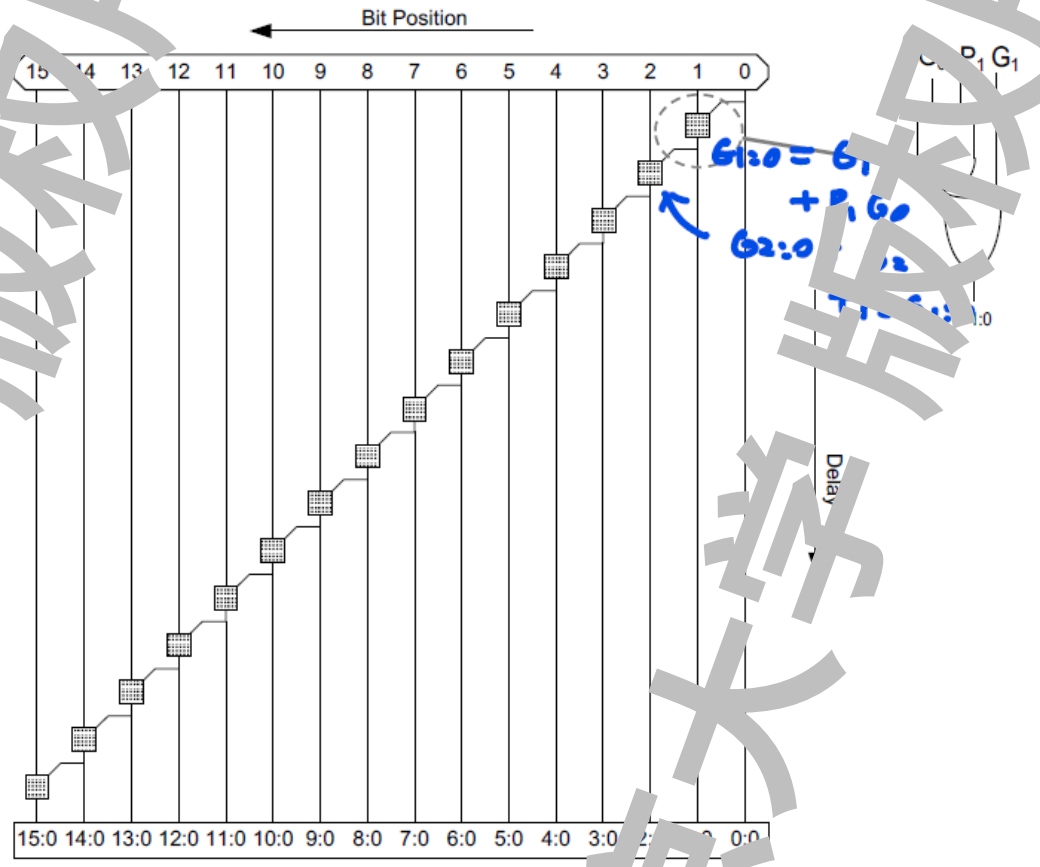
RCA、CLA、KS、Sklansky 都是 ② 的不同实现

加法器设计

- 基于PGK的加法器设计方法



PG生成逻辑



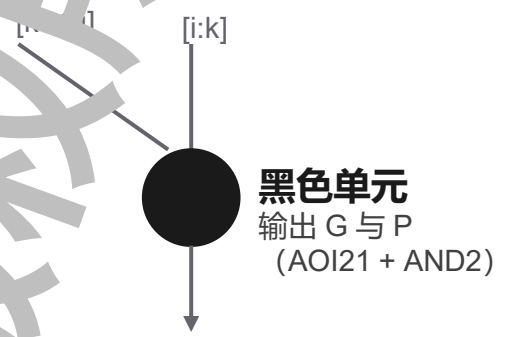
Carry Ripple的PG图

3.11 PGK 加法 - 前缀运算子

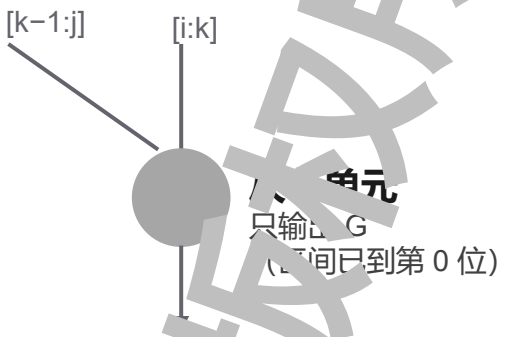
- 前缀运算满足结合律：括号怎么加都正确，不同加括号方式 = 不同前缀树

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

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

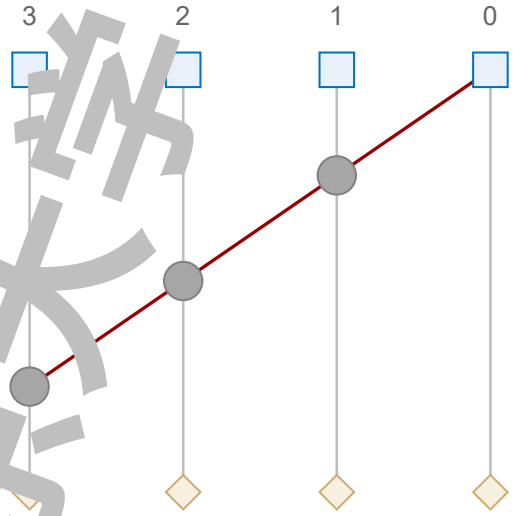


黑色单元
输出 G 与 P
（AOI21 + AND2）



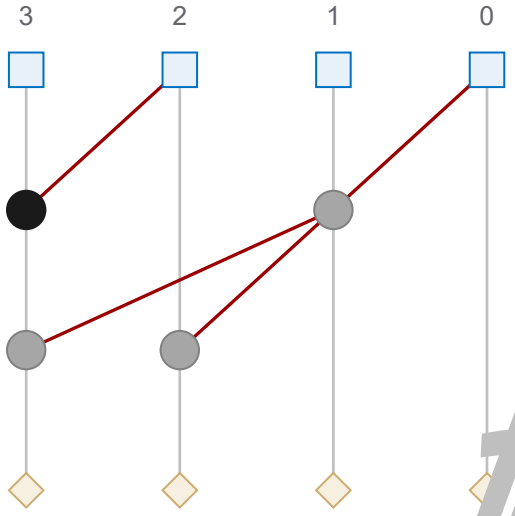
白色单元
只输出 G
（P 已到第 0 位）

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



3 级 —— 本质就是 RCA

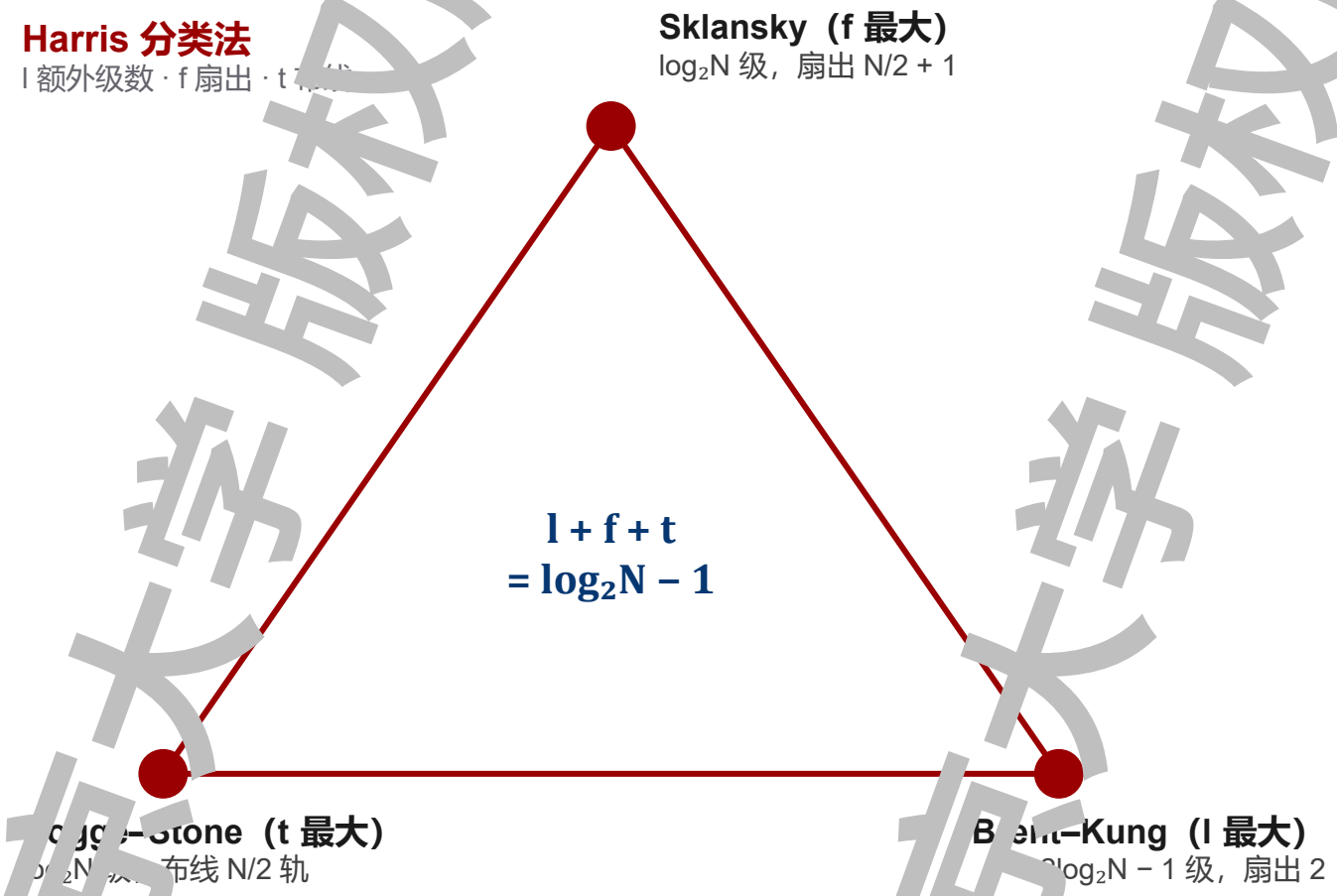
树形：$(g_3 \circ g_2) \circ (g_1 \circ g_0)$



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

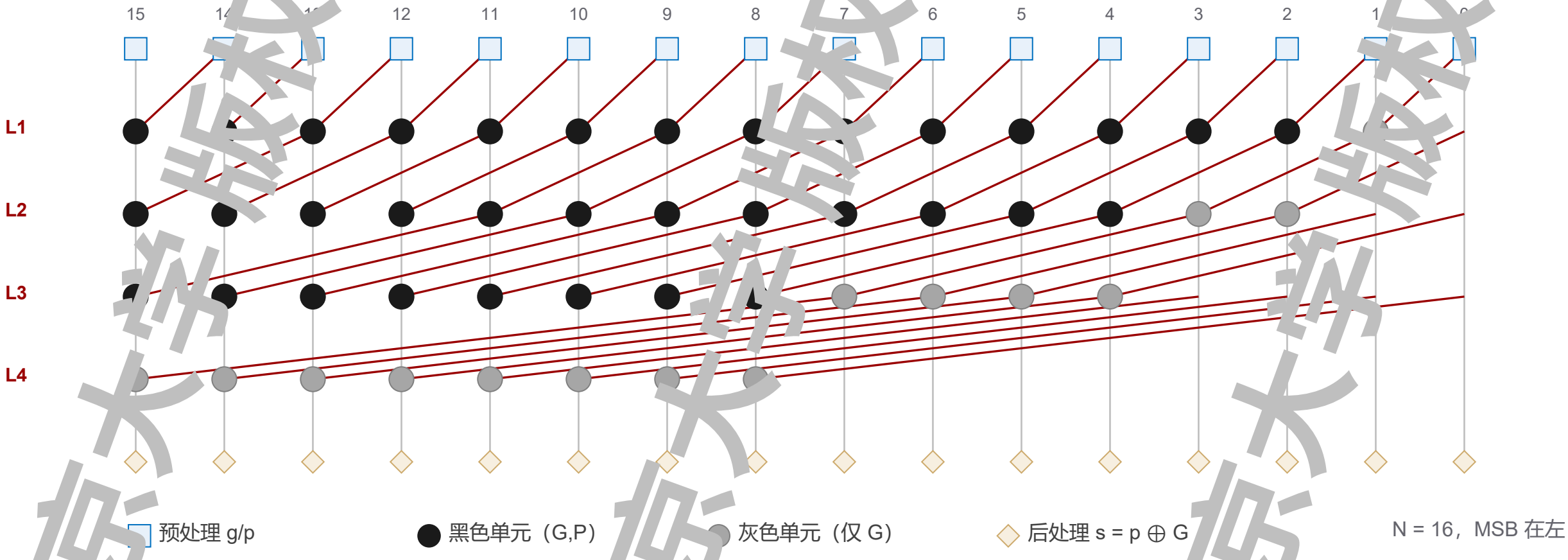
- 状态：N×N 网格，节点 (i, j) 表示计算 G[i,j]
- 动作：增加 / 删除一个节点，自动补全依赖使图合法
- 奖励：综合后面积与延迟的加权改进
- 已报道的结果在面积-延迟上优于传统结构的加法器

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

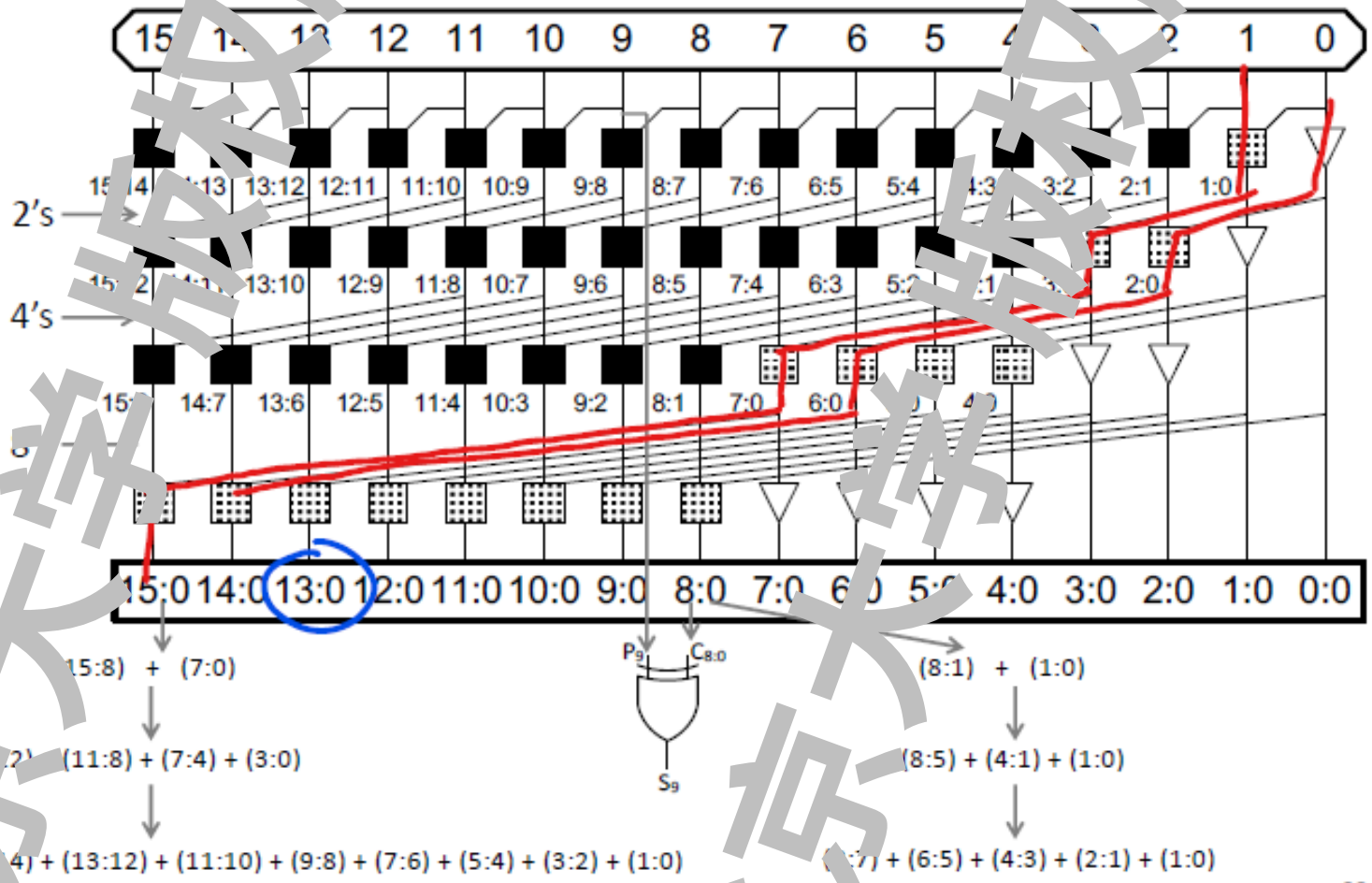
3.13 Kogge–Stone 加法器 – 结构

- 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



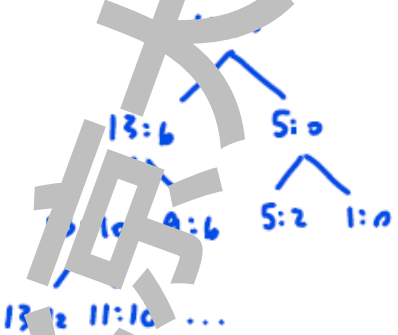
$\log_2(N)$

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

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

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

$G_{5:0} = G_{5:2} + P_{5:2}\,G_{1:0}$

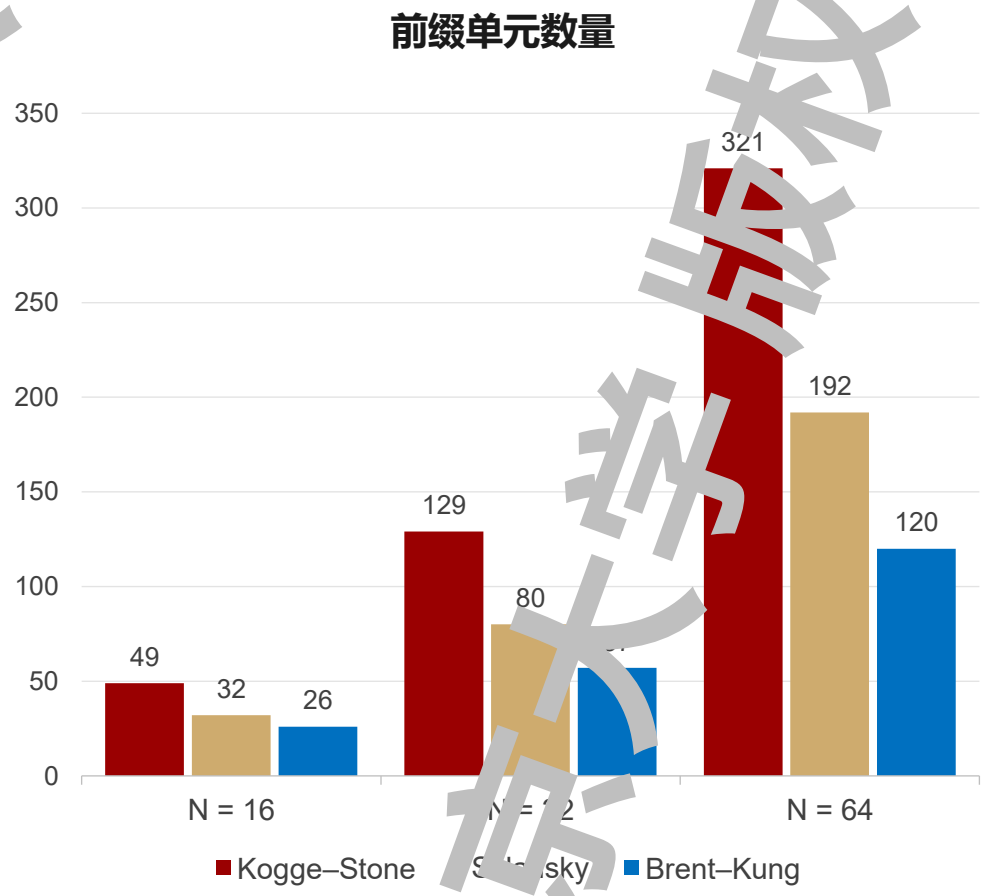


3.14 Kogge–Stone 加法器 – 特性与代价

- 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；

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；cin = 0。

LLM 生成的 Verilog

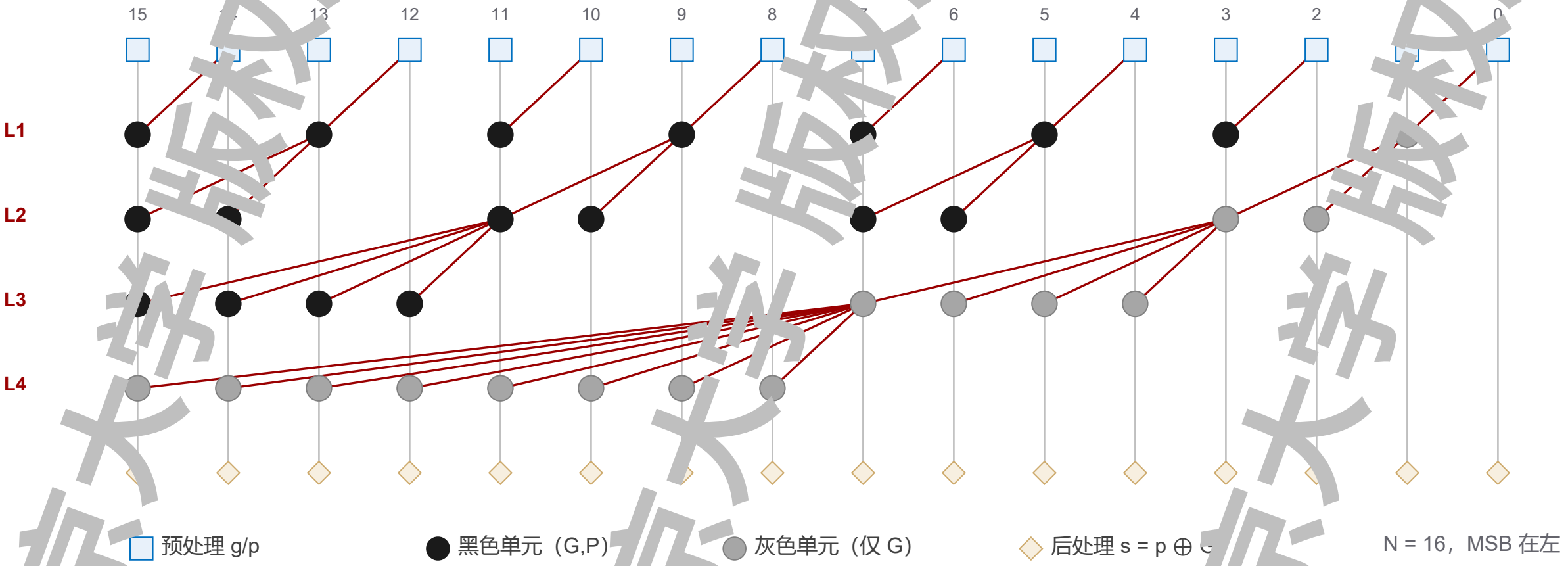
```
module kogge_stone #(parameter N = 16, L = $clog2(N)) (
  input  [N-1:0] a, b,
  output [N-1:0] s,
  output         cout);
  wire [N-1:0] G [0:L], P [0:L];
  assign G[0] = a & b;  assign P[0] = a ^ b;
  genvar i, l;
  generate for (l = 1; l <= L; l = l + 1) begin : lv     (级数l)
    for (i = 0; i < 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];
endmodule
```

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

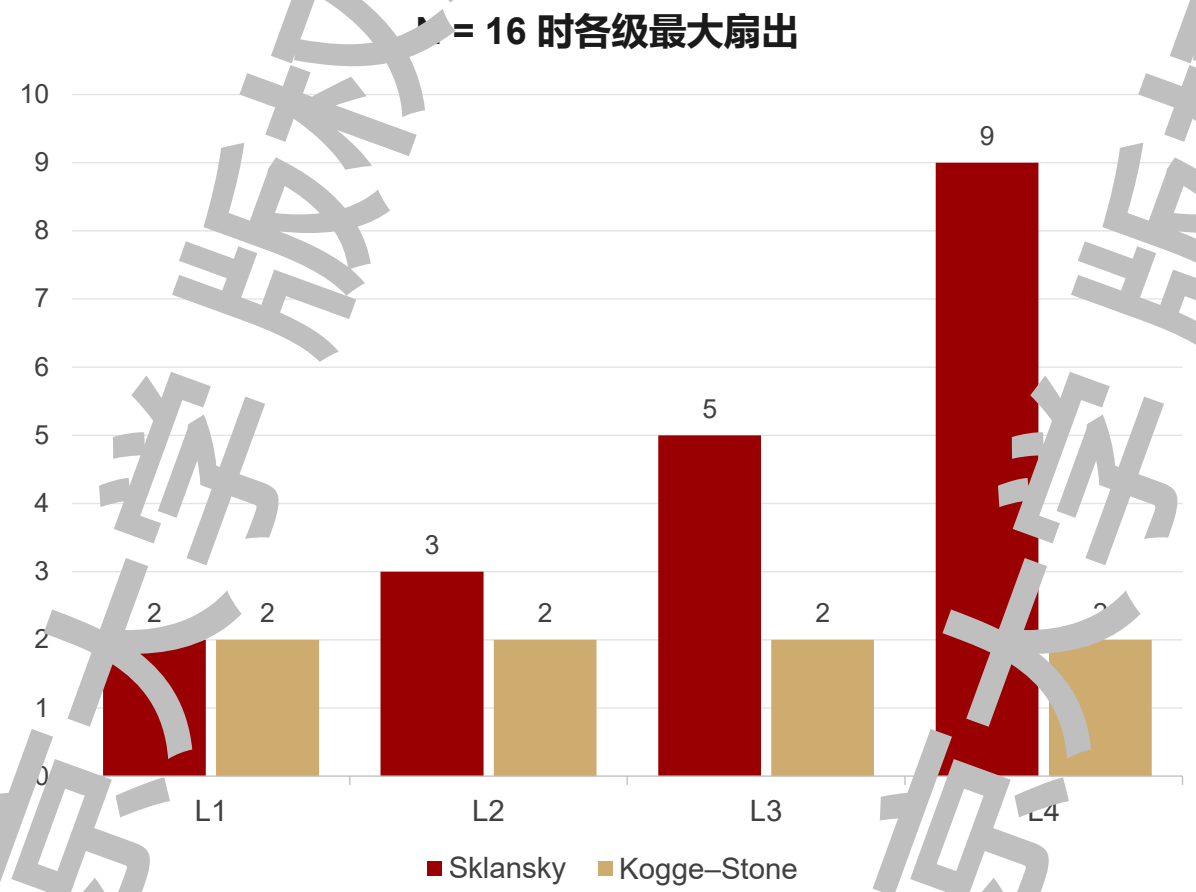
3.16 Sklansky 加法器 – 结构

- 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
  “$\log_2 N$ 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
2. **插入缓冲**
   把扇出分摊给多个缓冲器，代价是
3. **折中结构**
   Ladner–Fischer 等：多 1 级

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），并输出每个节点的扇出。

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
```

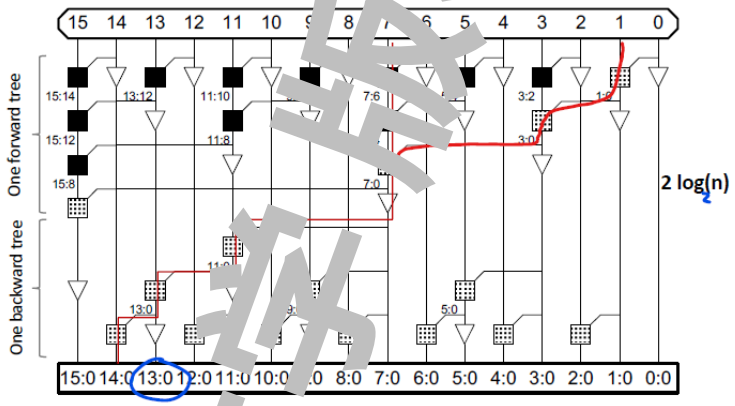
AI 输出检查清单

1. **索引计算**
   J = (i / D)·D − 1（整数除法），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 ↔ [illegible] ↔ BK 空间中按 PPA 目标折中

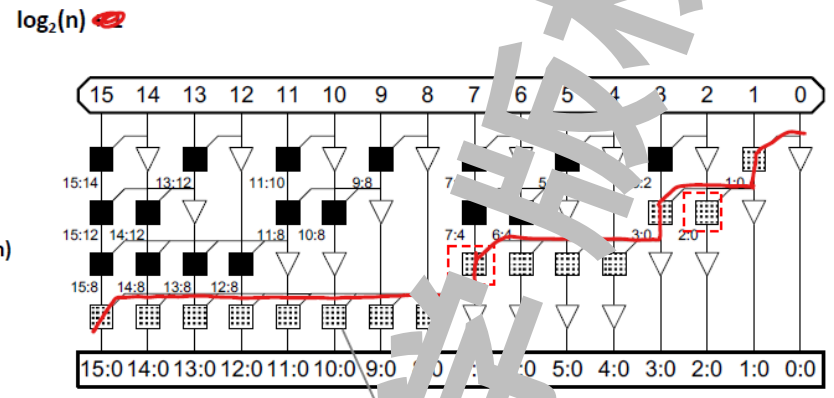
加法器设计

- 基于PGK的加法器设计方法 – 复杂PG树加法器

Brent-Kung

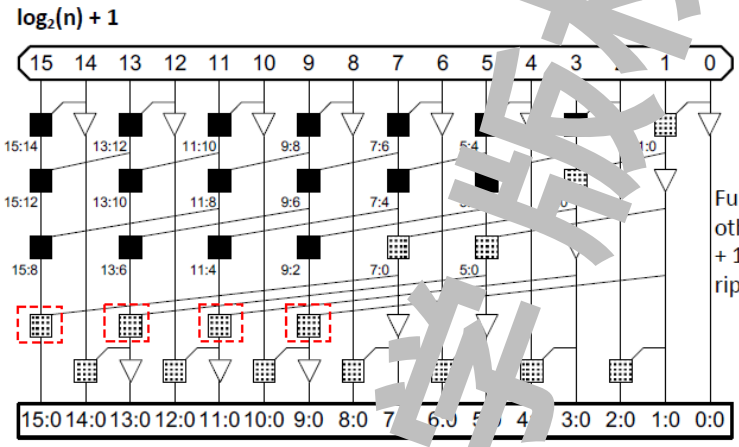


Sklansky



- Uneven sizing (10:8) + (7:0)
- Large fanout

Han-Carlson



Low fanout, tradeoff between logic levels and wiring
Reduces wire length by half ! → half power compared to Kogge Stone

- Kogge-Stone: low logic levels, low fanout, high wiring
- Brent-Kung: low fanout, low wiring, high logic levels
- Sklansky: low logic levels, low wiring, high fanout

加法器 – 参数化搜索空间

- 在级数 l、扇出 f、布线 t 之间折中：l + f + t = log₂N − 1（Harris 分类）



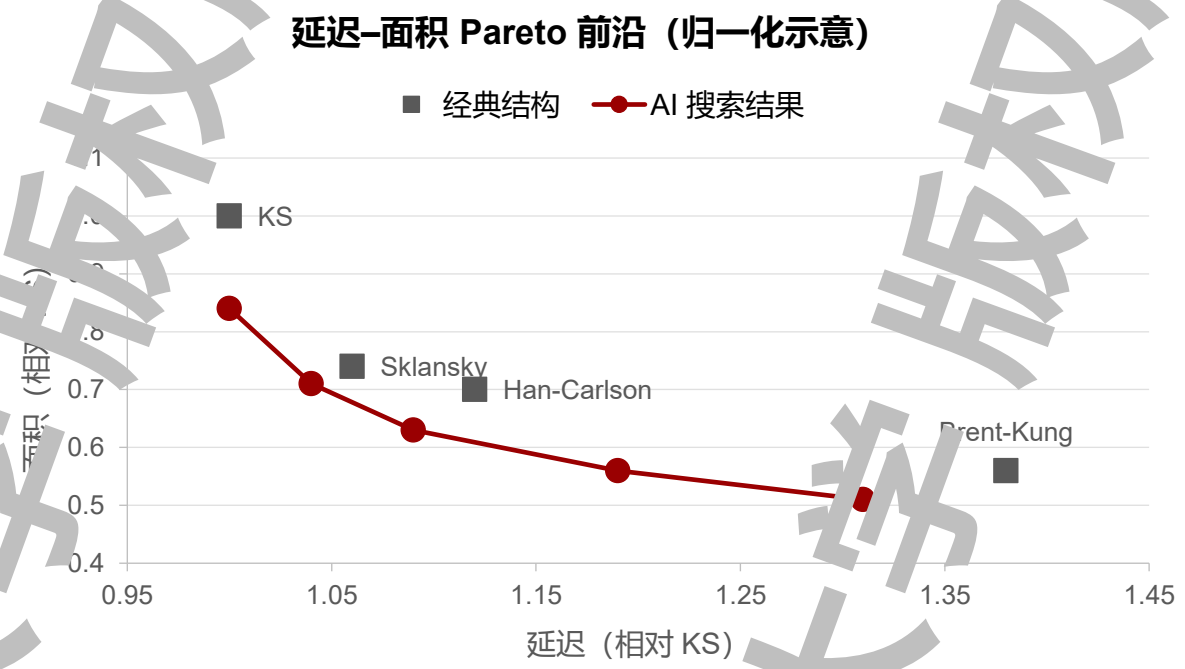
参数	取值	主要影响
位宽 N	8~数K	规模基线
稀疏度（进位抽头间隔）	1 / 2 / 4 / 8....	面积↓，末端需进位连接
基数 radix	2 / 3 / 4 / 5 / …	级数↓，单元更复杂
(l, f, t) 折中	l+f+t = $\log_2 N-1$	延迟与面积此消彼长
门尺寸 / 阈值电压	连续值	驱动能力与功耗
节点增删（图编辑）	任意合法前缀图	RL 的动作空间

PrefixRL 编码：状态 = N×N 网格上的前缀图，动作 = 增/删一个节点

案例一：Kogge-Stone加法器 – 优化流程与结果

- 以经典结构为种子，AI 在延迟–面积平面上把 Pareto 前沿整体向左下推移

1. **初始化** — KS / BK / Sklansky 作为初始种群
2. **生成 + 评估** — 输出 Verilog，按多个目标延迟综合
3. **策略更新** — 代理模型或 RL 策略给出下一批结构
4. **合法化** — 自动修补前缀图，保证加法结果正确
5. **输出前沿** — 保留非支配解，形成 Pareto 前沿

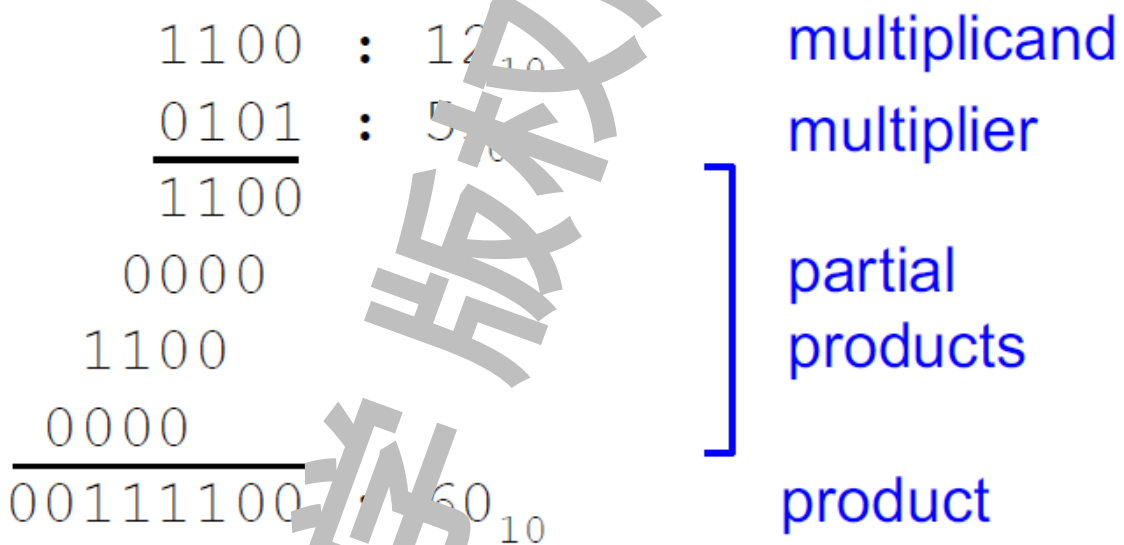


PrefixRL（NVIDIA, DAC 2021）： 64 位加法器在同等延迟下，面积比商用综合工具结果约小 25%

乘法器设计

- 乘法器设计的核心是部分和累加

Example:



M x N比特乘法

- 产生N个M比特部分乘积
- 求和得到M+N比特的结果

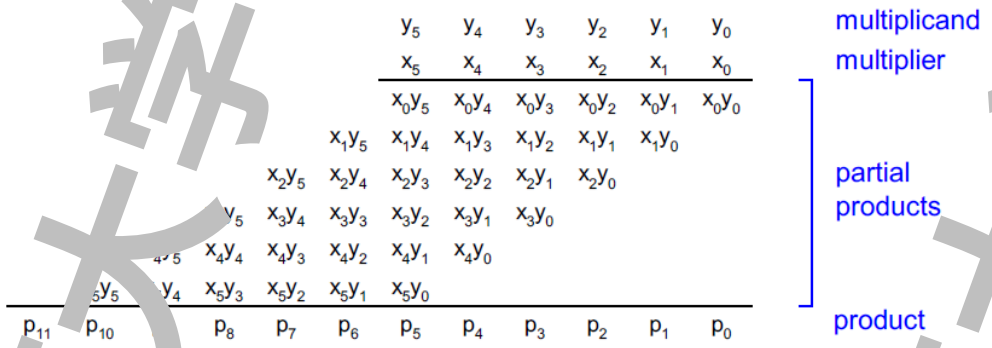
乘法器设计

- 乘法器设计的核心是部分和累加

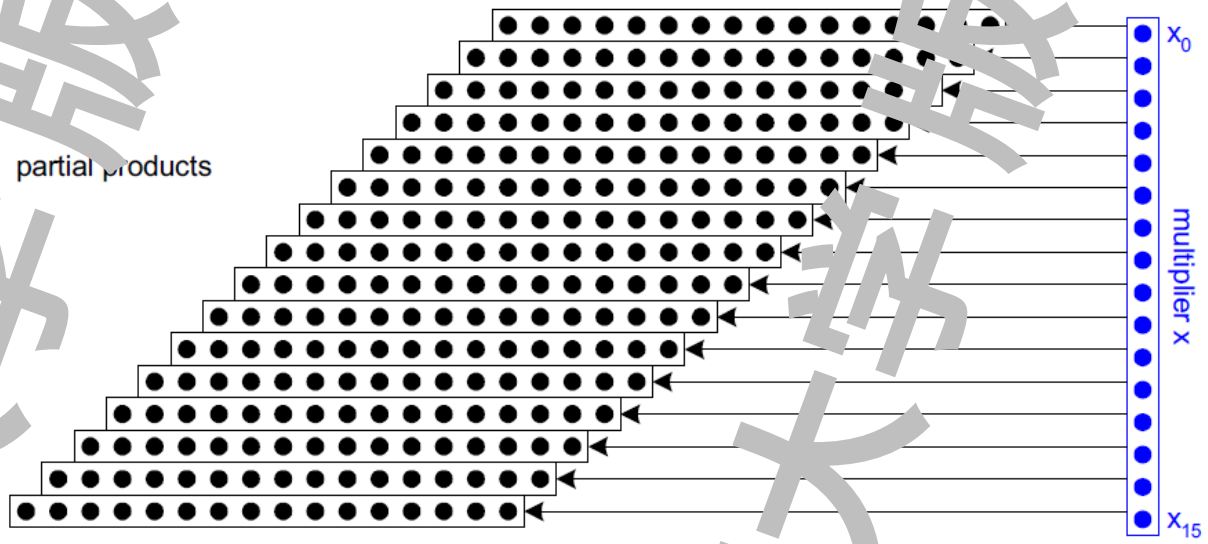
Multiplicand: $Y = (y_{M-1}, y_{M-2}, \ldots, y_1, y_0)$

Multiplier: $X = (x_{N-1}, x_{N-2}, \ldots, x_1, x_0)$

Product: $\left(\sum_{j=0}^{M-1} y_j 2^j\right)\left(\sum_{i=0}^{N-1} x_i 2^i\right) = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} x_i y_j 2^{i+j}$



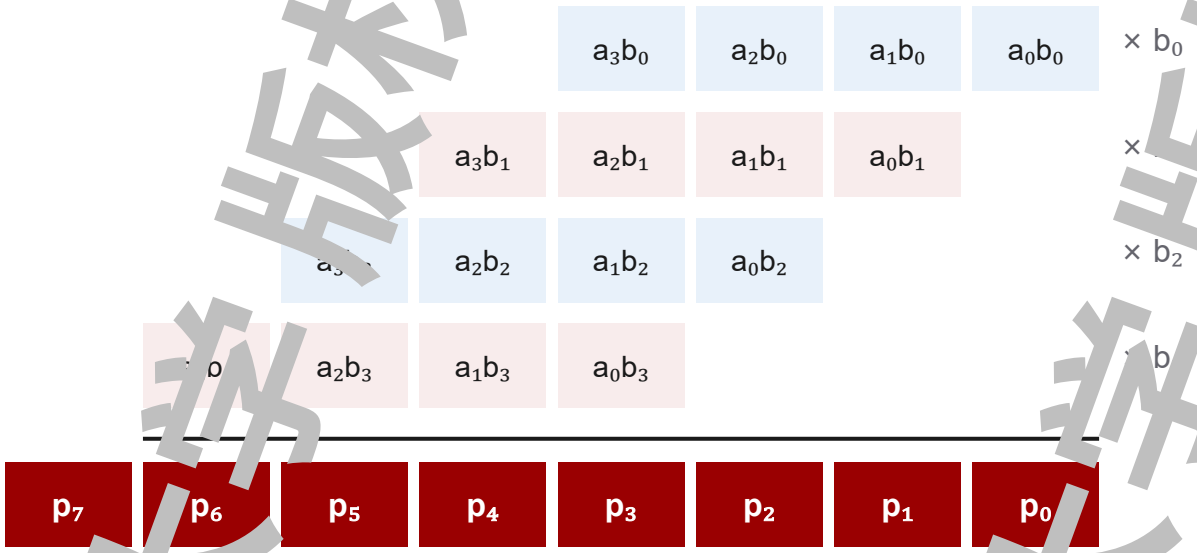
Each dot represents a bit



4.1 常规阵列乘法器 – 结构

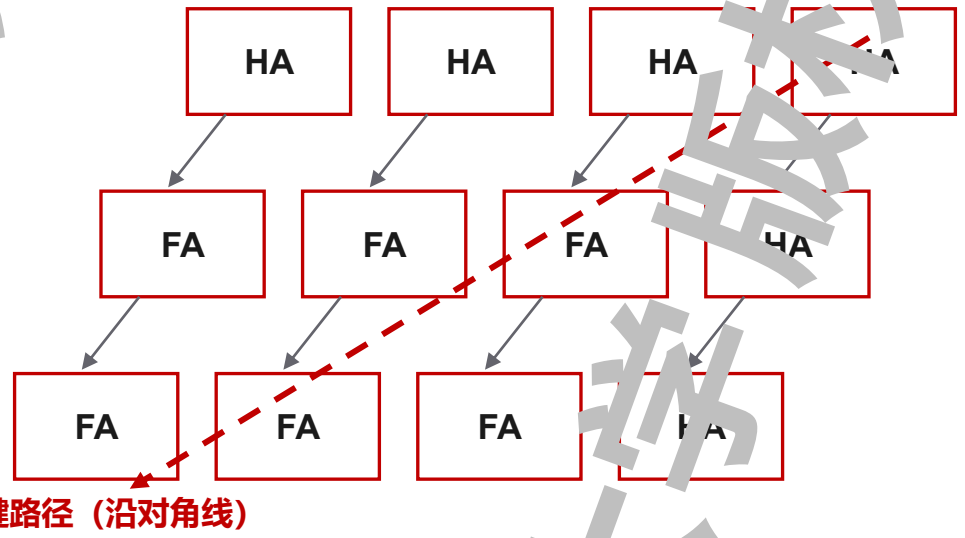
- 与门阵列产生部分积，加法器阵列逐行累加

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）

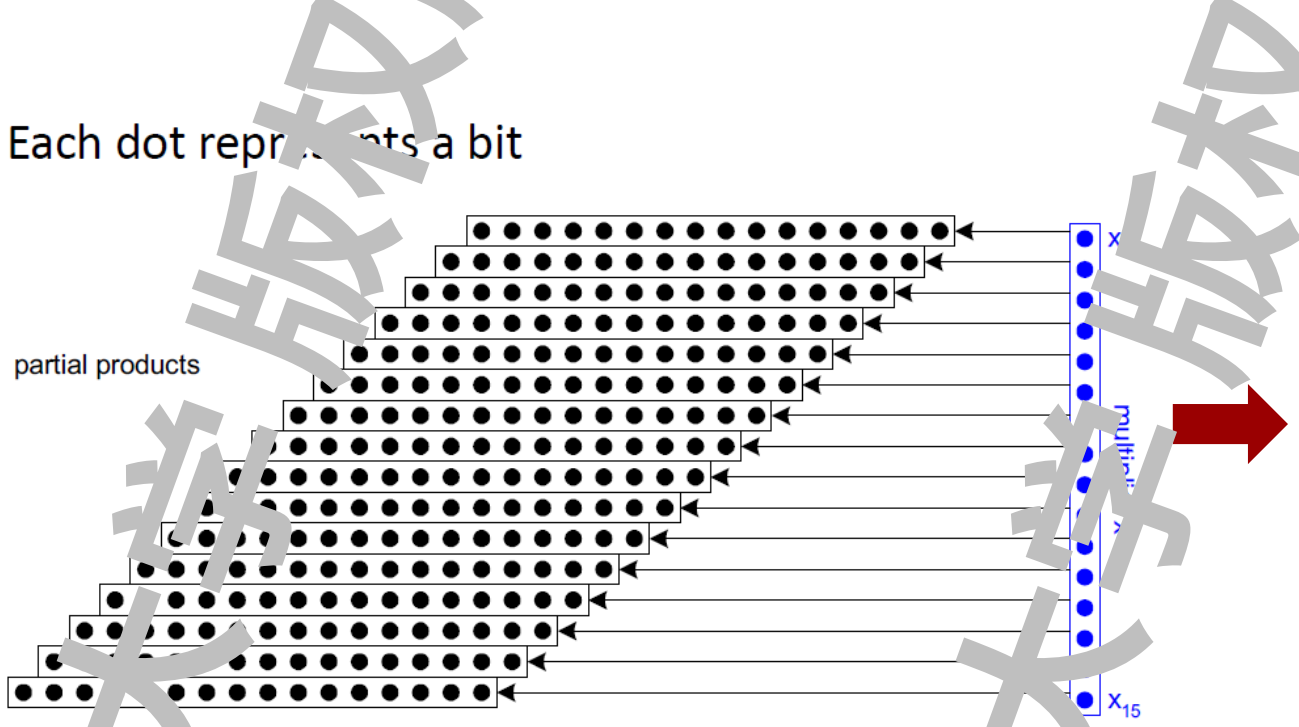
② 加法器阵列

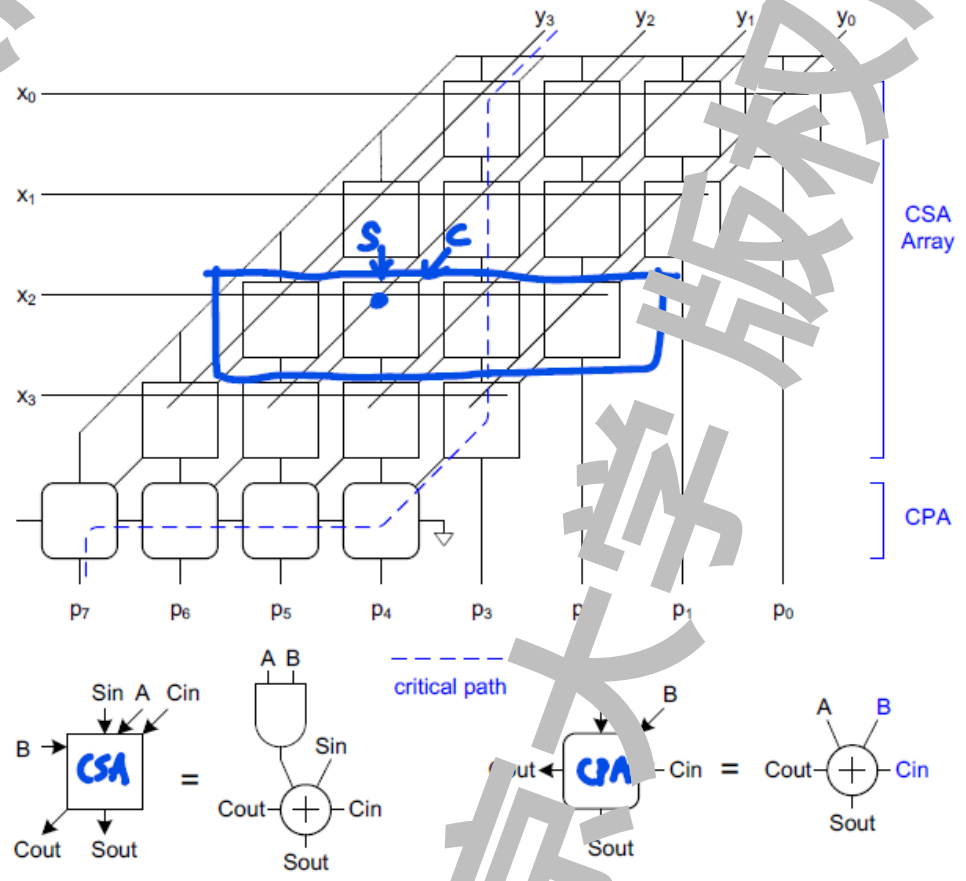


- 每行一级加法器，进位同时沿行和列传播
- 结构规则、易于版图拼接

乘法器设计

- 乘法器设计的核心是部分和累加





4.2 常规阵列乘法器 – 延迟与面积

- 面积 O(N²)、延迟 O(N)：多条几乎等长的关键路径需要同时优化

N^2

个 1 位与（部分积）

$\approx N(N-2)$

个全加器

O(N)

关键路径延迟

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

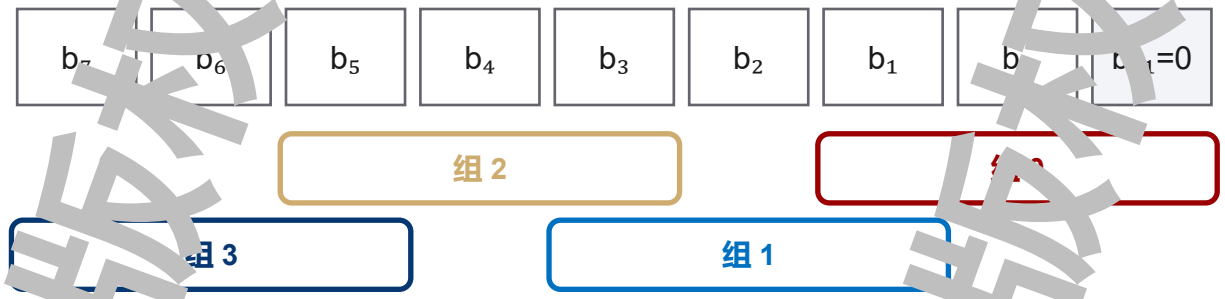
1. **穷举可行性**
   8×8 可穷举；32×32 只能随机 + 形式 / 代数方法
2. **形式验证难点**
   乘法器的 SAT 等价检查很难，宜用结构参考或代数重写
3. **有符号处理**
   检查 $signed 与 Baugh–Wooley 的修正项数
4. **输出位宽**
   积为 2N 位，防止 AI 在中间表达式中截断高位

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

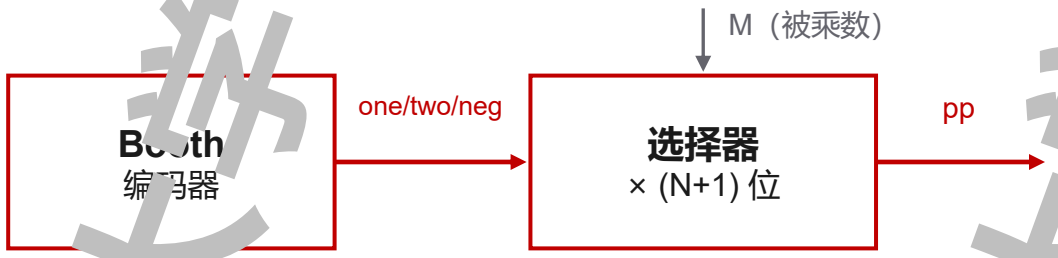
Booth 编码与选择

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

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

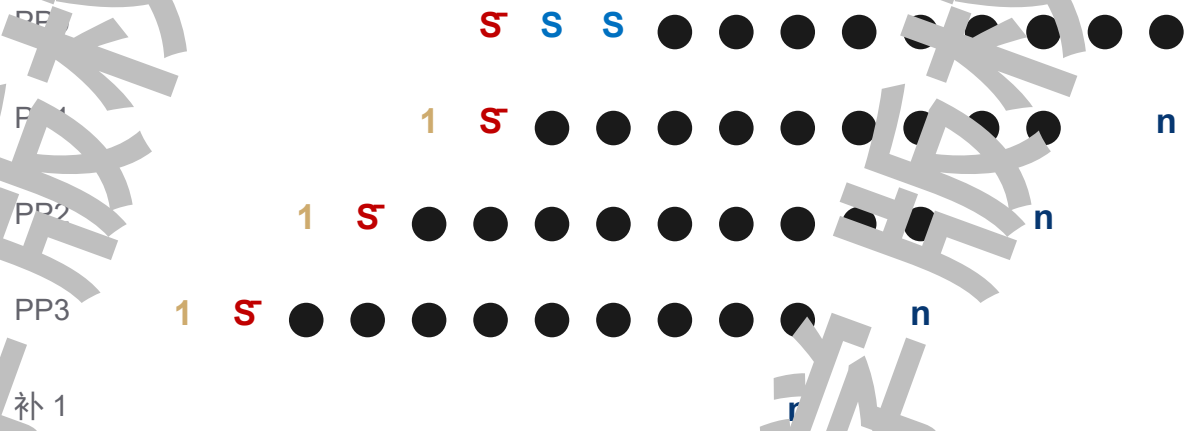
$$neg = b_{2i+1}$$

$$pp_j = (one \cdot m_j + two \cdot [illegible]_{j-1}) \oplus neg$$



取反只实现“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

4.5 Booth 编码乘法器 – 部分积生成与符号扩展

- 如何减少部分和累加的次数 – 布斯编码（Radix-2^r）

布斯编码的几点要求：

- 乘数、被乘数、结果均为补码
- 乘法计算前应在乘数末尾补零
- 被乘数双符号位
- 符号位参与计算

Inputs			Partial Product	Booth Selects		
x_{2i+1}	x_{2i}	x_{2i-1}	PP_i	$SINGLE_i$	$DOUBLE_i$	NEG_i
0	0	0	0	0	0	0
0	0	1	Y	1	0	0
0	1	0	Y	1	0	0
0	1	1	$2Y$	0	1	0
1	0	0	$-2Y$	0	1	1
1	0	1	$-Y$	1	0	1
1	1	0	$-Y$	1	0	1
1	1	1	$-0\ (=0)$	0	0	1

假设计算 Y x Q = -6 x -7，Q 是乘数，Y 是被乘数（4bit）

1、Y = -6 =1010　Q = -7 = 1001　-Y = 6 = 0110

2、乘数Q后补零，Q = 10010

3、被乘数双符号位，Y = 11010，-Y = 00110

3、乘法步骤（A为部分和、Q为乘数）

Step 1：Q = 10010

PP = 11111010　Q = 1001　Q-1 = 0　补码符号扩展

Step 2：Q = 10010

PP = 00110000　Q = 1001　Q-1 = 0　左移补零

结果：11111010（-6）+ 00110000（48）= 42

4.5 Booth 编码乘法器 – 部分积生成与符号扩展

- 如何减少部分和累加的次数 – 布斯编码（Radix-2^r）

布斯编码的几点要求：

- 乘数、被乘数、结果均为补码
- 乘法计算前应在乘数末尾补零
- 被乘数双符号位
- 符号位参与计算

Inputs			Partial Product	Booth Selects		
x_{2i+1}	x_{2i}	x_{2i-1}	P_i	$SINGLE_i$	$DOUBLE_i$	NEG_i
0	0	0	0	0	0	0
0	0	1	Y	1	0	0
0	1	0	Y	1	0	0
0	1	1	$2Y$	0	1	0
1	0	0	$-2Y$	0	1	1
1	0	1	$-Y$	1	0	1
1	1	0	$-Y$	1	0	1
1	1	1	$-0\ (=0)$	0	0	1

假设计算 Y x Q = -6 x 7，Q 是乘数，Y 是被乘数（6bit）

1、Y = -6 = 111010　Q = 7 = 000111　-Y = 6 = 000110

2、乘数 Q 后补零　Q = 0001110

3、被乘数双符号位，Y = 1111010，-Y = 0000110

3、乘法步骤（A为部分和、Q为乘数）

Step 1：Q = 0001110

PP = 000000000110　Q = 000111　Q-1 = 0　补码符号扩展

Step 2：Q = 0001110

PP = 111111010000　Q =000111　Q-1 = 0　左移/符号位扩展

Step 3：Q = 0001110

结果 = 000000000110（6）+ 111111010000（-48）= -42

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，检查清单要覆盖这些点

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 Verilo

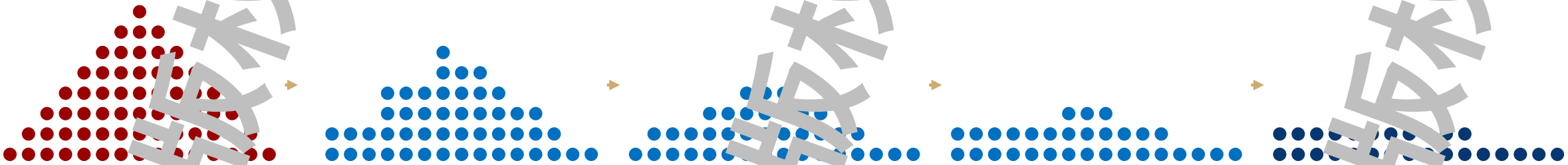
```
// Radix-4 Boo      组的编码与部分积选择
module booth_     #(p     eter N = 8) (
  input  [N-1     ,          // 被乘数（有符号）
  input  [2:0]    g      ,  // {b[2i+1], b[2i], b[2i-1]}
  output [N:0]    pp,       // N+1 位部分积（尚未 +1）
  output          neg);
  wire one = grp[1] ^ grp[0];
  wire two =  grp[2] & ~grp[1] & ~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
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
4. **验证**
   与 $signed(a) * $signed(b) 等价；×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

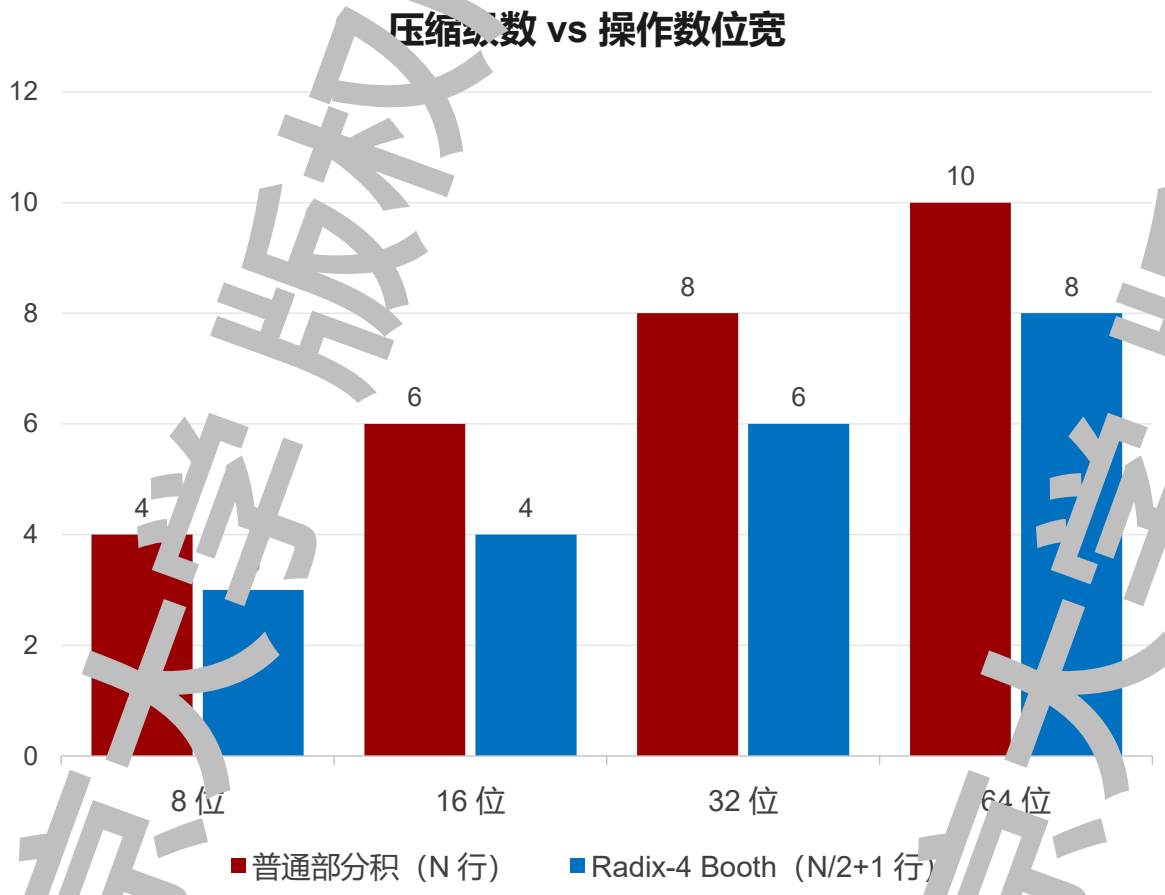
3:2 压缩器 = 全加器

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
- 压缩级数随位数按 $O(\log_{3/2} N)$ 增长，远快于阵列乘法器的 O(N)
- **直接使用加法器三级就可以实现：也就是8->4->2->1，岂不是运算更快?**
- **这里需要注意了，这样计算每一级逻辑是多位加法器延时，而本节的Wallace树结构每一级逻辑都是1位全加器延时。**

4.8 Wallace 树乘法器 – 级数分析与 Dadda / 4:2 压缩

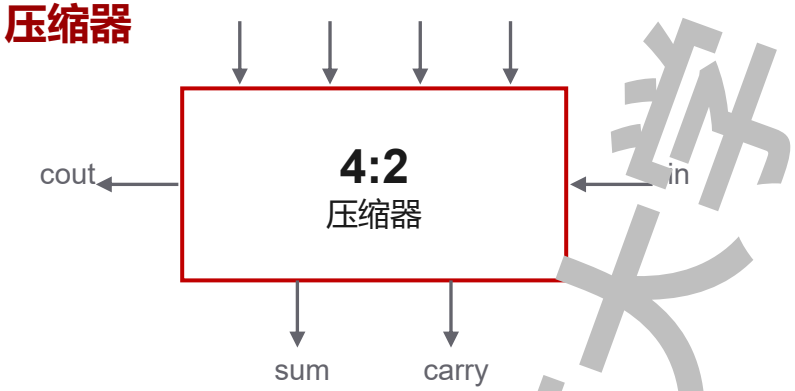
- Booth 减行 + 树形压缩是现代高性能乘法器的标准组合



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…），加法器更少、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

- 压缩器的数量与放置构成巨大离散空间，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 状态表示

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、对数与漏斗三类

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

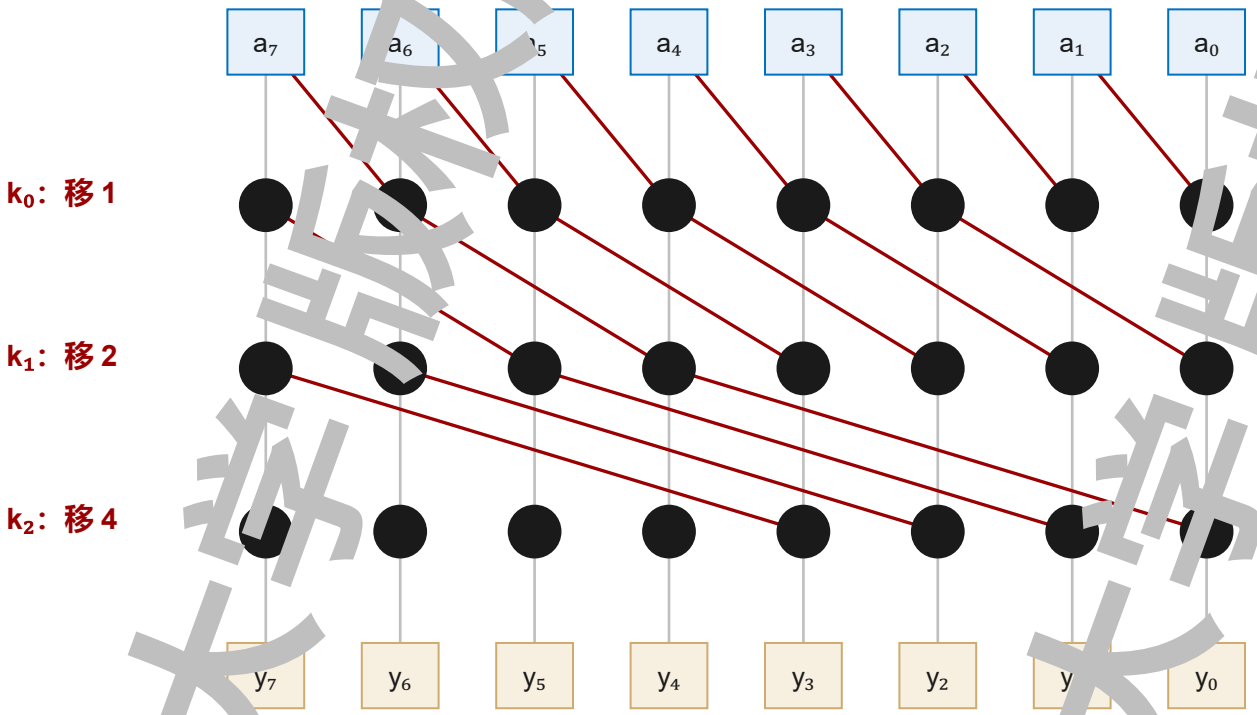
结构分类

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

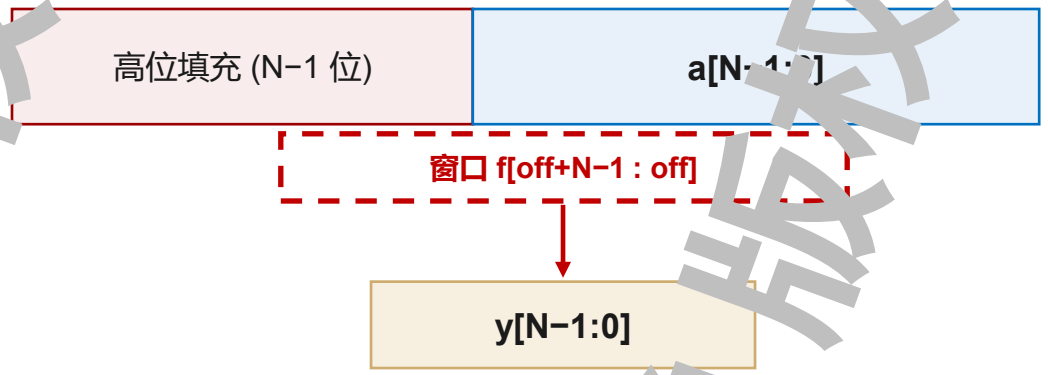
5.2 对数移位器与漏斗移位器

- 对数移位器按 k 的各位分级移动 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，人负责检查语义边界

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 Verilo

```
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-2:0] 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）

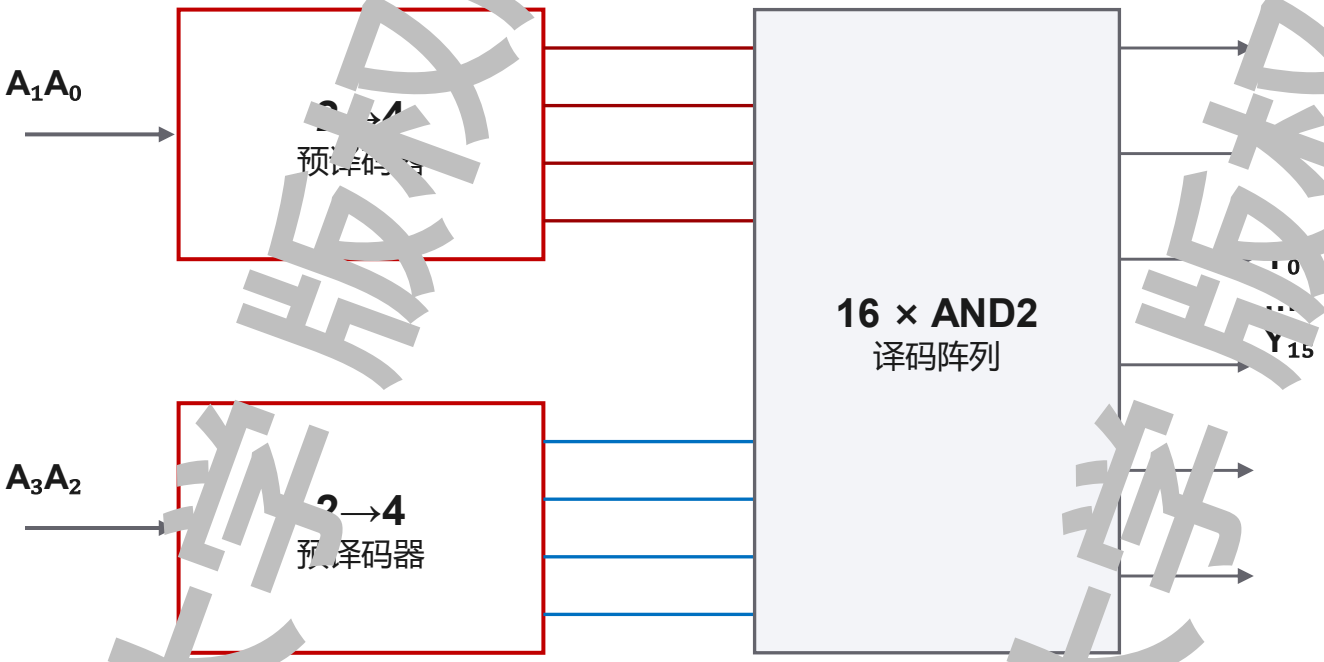
2. **SRA 符号位**
   必须复制 a[N−1]；若改用 >>> 需声明 signed

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 → 2ⁿ 路 one-hot 输出；预译码把高扇入门拆成两级小门



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 × AND2	2

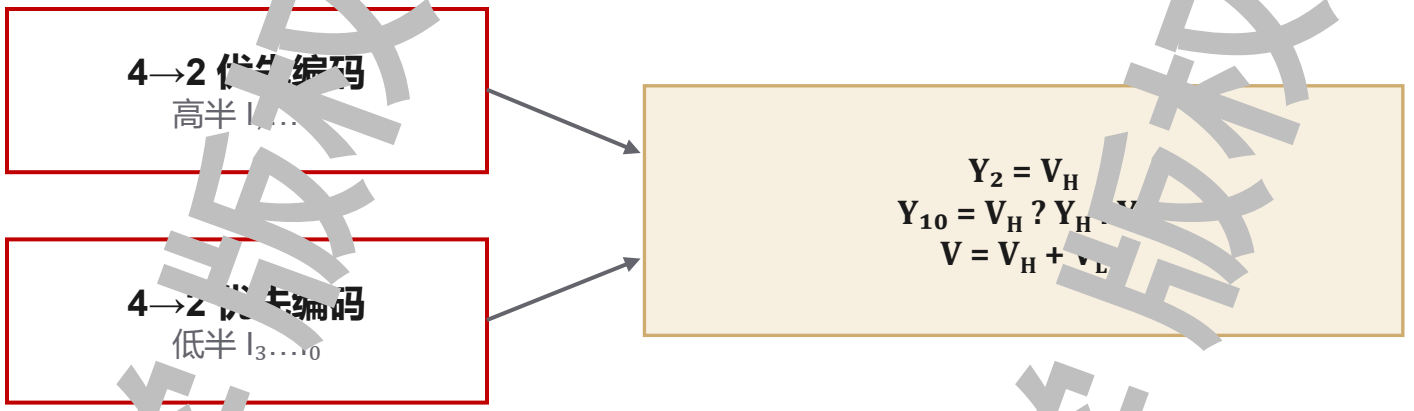
- one-hot 输出：任意时刻只有一路有效
- SRAM 行译码常用多级预译码 + 字线驱动器
- 预译码减小扇入、分摊输入负载，更便于按逻辑努力分配尺寸

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结构

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 priority_encoder #(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

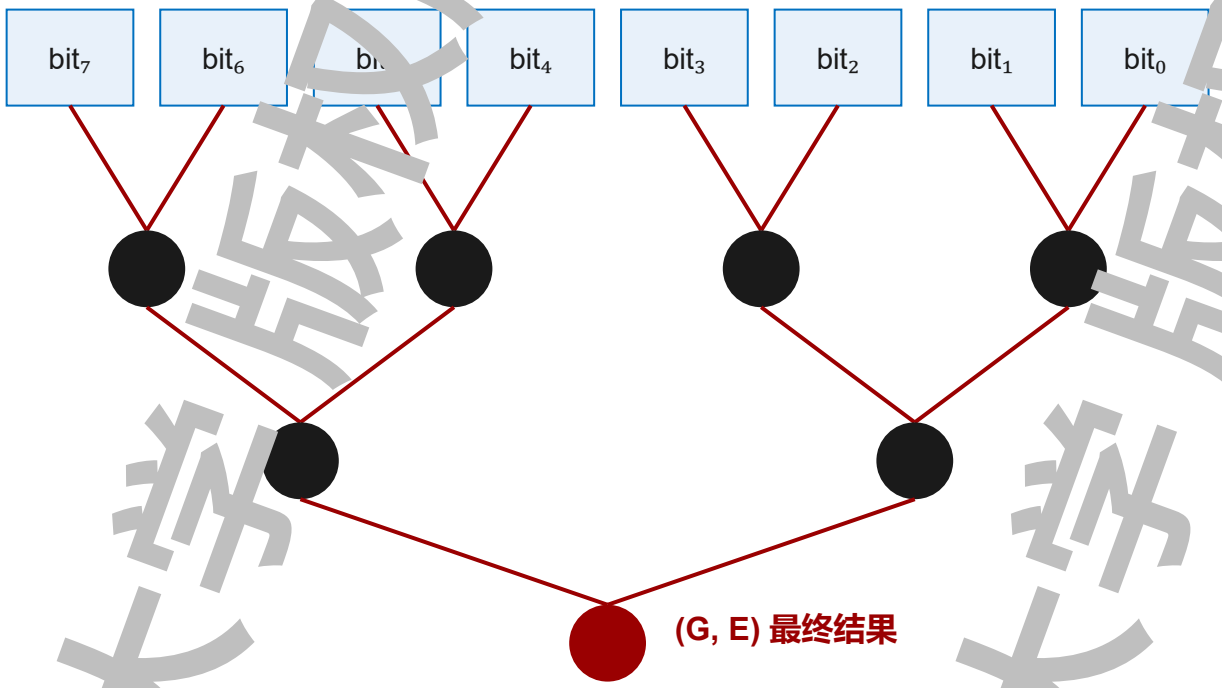
6.4 比较器 – 三种典型实现

- 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	减法型比较器	树形比较器
$a_i \odot b_i$（XNOR） × N	A + ¬B + 1	每位 (g_i, e_i)
AND 归约树	加法器 sum / cout	(G, E) 逐级合并
EQ	cout = 1 ⇔ A ≥ B	GT / EQ / LT
• 深度 $\log_2 N$ 级 • 只判断相等，面积最小	• 复用 ALU 加法器，几乎零额外面积 • 延迟 = 加法器进位延迟	• $g_i = a_i \cdot \neg b_i$，$e_i = a_i \odot b_i$ • 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-Sele	双路预计算 + MUX	O(√N)	≈ 2× RCA	分块方案搜
Carry-Looka d	分组 (G, P) + LCU	O(log N)	O(N)	扇入约束 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- tone	跨度倍增，扇出 2	log N 级	$N \cdot \log_2 N$ 单元	布线感知、稀疏化
Skla sky	分治，高扇出	$\log_2 N$ 级	$(N/2) \cdot \log_2 N$	尺寸与 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
Boot 乘法器	Radix-4 重编码	部分积减半	编码器开销	符号与 +1 纠错
W e 树	3:2 / 4:2 压缩树	O(log N)	$O(N^2)$	压缩树 RL 优化
移位器	对数 / 漏斗	$\log_2$ 级	$N \cdot \log_2 N$ MUX	统一漏斗模板
编码器	预译码、树形优先编码	O(log N)	O(N) / $O(2^n)$	优先 器、树形化
比较器	相等 / 减法 / 树形	O(log N)	O(N)	符号与 界、资源共享

北京大学版权所有

AI辅助复杂模块[illegible]设计1 - 卷积加速模块

- 卷积 – AI计算[illegible]常用的算子

卷积最初是一种信号处理滤波操作 –

“AI神经网络也可看作是一种滤波”

WinoGrad算法起源于1980年，是Shmuel Winograd提出用来减少FIR滤波器计算量的一个算法

输入d = [d0,d1,d2,d3]，卷积核g = [g0, g1, g2]

1维卷积可表达为

输入尺寸 = 2 + 3 -1 = 4

$$F(2,3) = \begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix} \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} = \begin{bmatrix} r0 \\ r1 \end{bmatrix}$$

输出尺寸　卷积核尺寸

6次乘法

AI辅助卷积加速软件设计

- 卷积 – AI计算中最常用的算子

输入d = [d0, d1, d2,d3]，卷积核g = [g0, g1, g2]

1维卷积可表达为

Winograd

$$F(2,3) = \begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix} \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} = \begin{bmatrix} r0 \\ r1 \end{bmatrix}$$

存在重复元素

$$F(2,3) = \begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix} \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} = \begin{bmatrix} m1 + m2 + m3 \\ m2 - m3 - m4 \end{bmatrix}$$

$$m1 = (d0 - d2)g0 \quad m2 = (d1 + d2)\frac{g0 + g1 + g2}{2}$$

$$m4 = (d1 - d3)g2 \quad m3 = (d2 - d1)\frac{g0 - g1 + g2}{2}$$

预算好一次g的加减后可重复复用 -> 4次乘法

AI辅助卷积加速算法设计

- 卷积 – AI计算中最常用的算子

element-wise multiplication（Hadamard product）

输入d = [d0, d1, d2,d3]，卷积核g = [g0, g1, g2]

1维卷积可表达为

$$F(2,3) = \begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix} \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} = \begin{bmatrix} m1+m2+m3 \\ m2-m3-m4 \end{bmatrix}$$

$$m1 = (d0 - d2)g0 \quad m2 = (d1+d2)\frac{g0+g1+g2}{2}$$

$$m4 = (d1 - d3)g2 \quad m3 = (d2-d1)\frac{g0-g1+g2}{2}$$

1D Winograd

$$Y = A^T\left[(Gg) \odot (B^T d)\right]$$

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} m1+m2+m3 \\ m2-m3-m4 \end{bmatrix}$$

$$g = [g_0 \quad g_1 \quad g_2]^T$$

$$d = [d0 \quad d1 \quad d2 \quad d3]^T$$

AI辅助卷积加速算法设计

- 卷积 – AI计算中最常用的算子

将一维卷积运算定义为$F(m,r)$，m为Output Size，r为Filter Size，则输入信号的长度为$m+r-1$，卷积运算是对应位置相乘然后求和，**输入信号每个位置至少要参与一次乘法**，所以乘法数量最少与输入信号长度相同，记为

$$\mu(F(m,r)) = m+r-1$$

在行列上分别进行一维卷积运算，可得到二维卷积，记为$F(m\times n, r\times s)$，输出为$m\times n$，卷积核为$r\times s$，则输入信号为$(m+r-1)(n+s-1)$，乘法数量至少为

$$\begin{aligned}\mu(F(m\times n, r\times s)) &= \mu(F(m,r))\mu(F(n,s)) \\ &= (m+r-1)(n+s-1)\end{aligned}$$

直接按滑动窗口方式计算卷积，一维时需要$m\times r$次乘法，二维时需要$m\times n\times r\times s$次乘法，**远大于上面计算的最少乘法次数**。

使用Winograd算法计算卷积快在哪里？一言以蔽之：**快在减少了乘法的数量**，将乘法数量减少至$m+r-1$或$(m+r-1)(n+s-1)$。

AI辅助卷积加速硬件设计

- 卷积 – AI计算中最常用的算子

$$Y = A^T\left[(G\,g) \odot (B^T d)\right]$$

g：表示卷积核

d：表示输入信号

G：卷积核变换矩阵，尺寸为$(m+r-1)\times r$

B^T：输入变换矩阵，尺寸为$(m+r-1)\times(m+r-1)$

A^T：输出变换矩阵，尺寸为$m\times(m+r-1)$

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

$$g = [g_0 \quad g_1 \quad g_2]^T$$

$$d = [d0 \;\; d1 \;\; d2 \;\; d3]^T$$

计算过程可分为4步：

（1）输入变换

（2）卷积核变换

（3）[illegible]

（4）输出变换

1D Winograd

AI辅助卷积加速硬件设计

- 卷积 – AI计算中最常用的算子

$$Y = A^T \left[(G g) \odot (B^T d) \right]$$

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

$$g = [g_0 \quad g_1 \quad g_2]^T$$

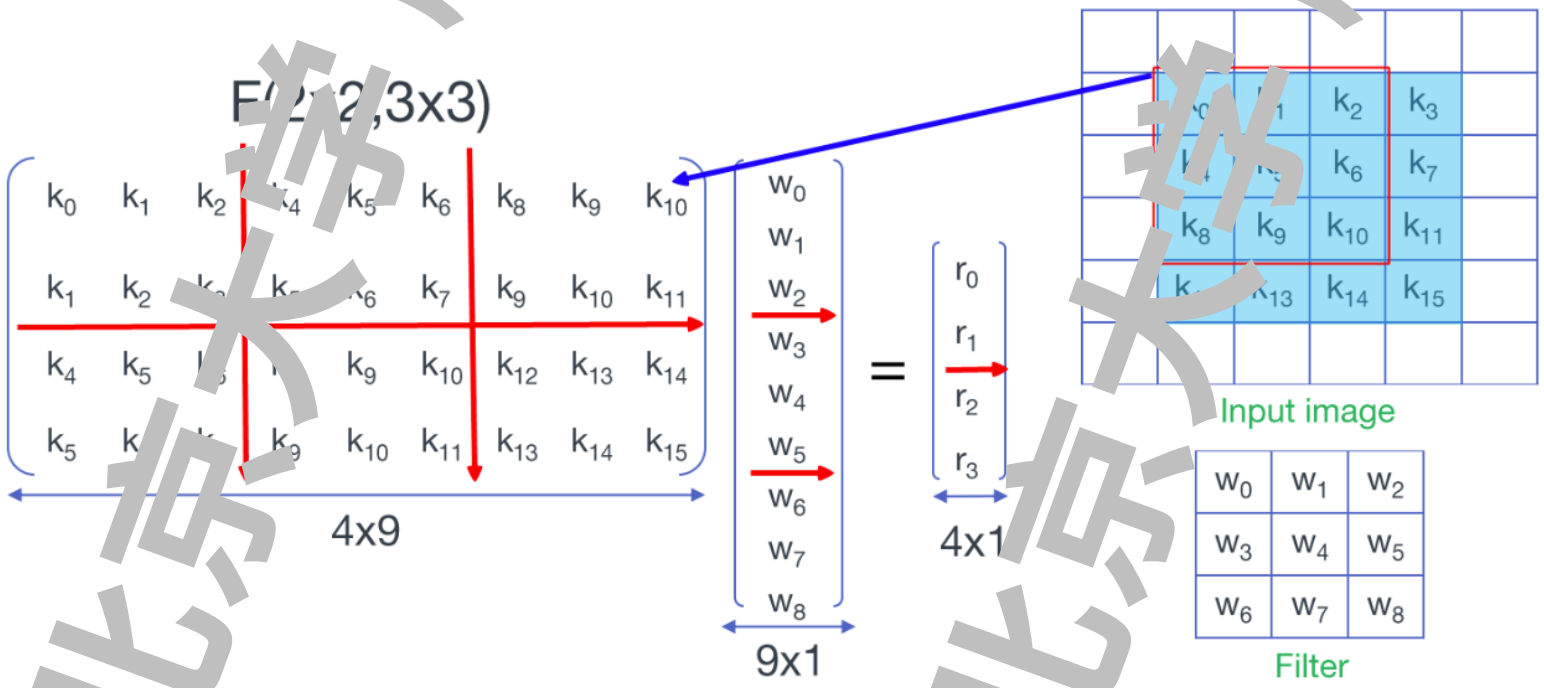
$$d = [d0 \quad d1 \quad d2 \quad d3]^T$$

2D Winograd

A minimal 1D algorithm F(m, r) is **nested with itself** to obtain a minimal 2D algorithm (m×m, r×r).

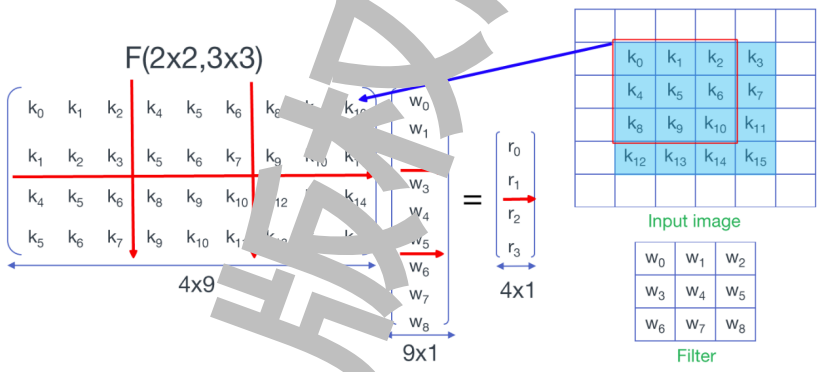
$$Y = A^T \left[\left[G g G^T \right] \odot \left[B^T d B \right] \right] A$$

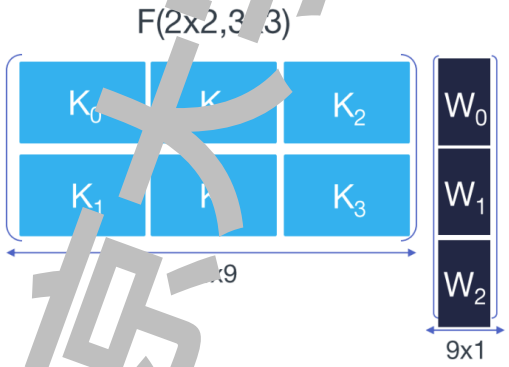
g为$r \times r$ filter，d为$(m+r-1) \times (m+r-1)$



AI辅助卷积加速核设计

- 卷积 – AI计算中最常用的算子





令$D_0 = [k_0, k_1, k_2, k_3]^T$，即窗口一的第0行元素，$D_1$ D_2 D_3表示第1、2、3行；$W_0 = [w_0, \quad w_2]^T$，

$$\begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} R_0 \\ R_1 \end{bmatrix} = \begin{bmatrix} K_0W_0 + K_1W_1 + K_2W_2 \\ K_1W_0 + K_2W_1 + K_3W_2 \end{bmatrix}$$

$$= \begin{bmatrix} A^T\left[(GW_0) \odot (B^TD_0)\right] + A^T\left[(GW_1) \odot (B^TD_1)\right] + A^T\left[(GW_2) \odot (B^TD_2)\right] \\ A^T\left[(GW_0) \odot (B^TD_1)\right] + A^T\left[(GW_1) \odot (B^TD_2)\right] + A^T\left[(GW_2) \odot (B^TD_3)\right] \end{bmatrix}$$

$$= A^T\left[\left[G[W_0\ W_1\ W_2]G^T\right] \odot \left[B^T[d_0\ d_1\ d_2\ d_3]B\right]\right]A$$

$$= A^T\left[\left[GgG^T\right] \odot \left[B^TdB\right]\right]A$$

$$\begin{bmatrix} K_0 & K_1 & K_2 \\ K_1 & K_2 & K_3 \end{bmatrix} \begin{bmatrix} W_0 \\ W_1 \\ W_2 \end{bmatrix} = \begin{bmatrix} R_0 \\ R_1 \end{bmatrix} = \begin{bmatrix} M_0 + M_1 + M_2 \\ M_1 - M_2 - M_3 \end{bmatrix}$$

Matrix multiply F(2,3)! **4** multiplications

$$M_0 = (K_0 - K_2) \cdot W_0 \qquad M_1 = (K_1 + K_2) \cdot \frac{W_0 + W_1 + W_2}{2}$$

$$M_3 = (K_1 - K_3) \cdot W_2 \qquad M_2 = (K_2 - K_3) \cdot \frac{W_0 - W_1 + W_2}{2}$$

AI辅助卷积加速器设计

- NVDLA中的Winograd卷积核设计

在MAC并行计算单元中计算

$$A^T\left[\left[GgG^T\right]\odot\left[B^TdB\right]\right]A$$

预先算好　　在输入datapath计算

For m = 3, r = 2

$$B^T=\begin{bmatrix}1&0&-1&0\\0&1&1&0\\0&-1&1&0\\0&-1&0&1\end{bmatrix},G=\begin{bmatrix}1&0\\\frac{1}{2}&\frac{1}{2}\\\frac{1}{2}&-\frac{1}{2}\\0&1\end{bmatrix}$$

$$A^T=\begin{bmatrix}1&1&1&0\\0&1&-1&0\\0&1&1&1\end{bmatrix}$$

For m = 4, r = 3

$$B^T=\begin{bmatrix}4&0&-5&0&1&0\\0&-4&-4&1&1&0\\0&4&-4&-1&1&0\\0&-2&-1&2&1&0\\0&-2&-1&-2&1&0\\0&4&0&-5&0&1\end{bmatrix}$$

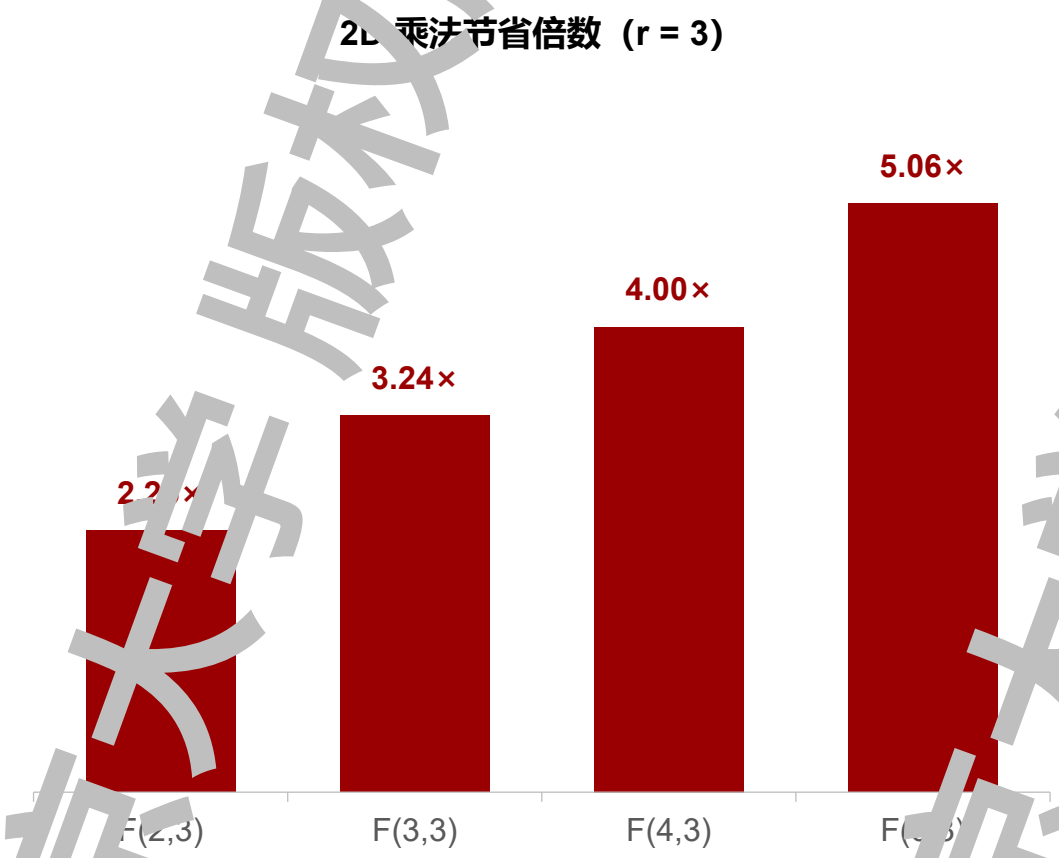
$$G=\begin{bmatrix}\frac{1}{4}&0&0\\-\frac{1}{6}&-\frac{1}{6}&-\frac{1}{6}\\-\frac{1}{6}&\frac{1}{6}&-\frac{1}{6}\\\frac{1}{24}&\frac{1}{12}&\frac{1}{6}\\\frac{1}{24}&-\frac{1}{12}&\frac{1}{6}\\0&0&1\end{bmatrix}$$

$$A^T=\begin{bmatrix}1&1&1&1&1&0\\0&1&-1&2&-2&0\\0&1&1&4&4&0\\0&1&-1&8&-8&1\end{bmatrix}$$

北京大学版权所有

案例二：Winograd 卷积 – 参数空间与折中

- tile 越大乘法越少，但变换中的加法数量、系数动态范围与数值误差同步上升



1. **tile 尺寸 m**
   决定乘法数 n^2 与变换矩阵规模
2. **插值点选择**
   0、±1、±½、±2、∞ 等；决定系数大小与数值稳定性
3. **定点位宽**
   变换后动态范围扩大，需要额外保护位
4. **变换电路结构**
   B^T/A^T 为常系数矩阵，可做公共子表达式消除
5. **并行度**
   PE 阵列规模需与存储带宽匹配

案例二：Wino[illegible]卷积 – AI联合搜索

- 硬件参数与数[illegible]参数联合优化，把精度作为约束放进搜索，而不是事后检查

搜索变量

x = (m, 插值点[illegible]位宽 b, 变换加法树, PE 数)

优化目标

最小化 每次[illegible]能耗 与 芯片面积

约束[illegible]

量化后[illegible]各[illegible]损失 ≤ ε（如 0.5%）

搜[illegible]

多[illegible]真贝叶斯优化：低保真快速筛选，高保真精确确认



常见结论：低位宽时，中等 tile + 优化插值[illegible]注[illegible]盲目增大 tile

总结与启示

- AI 让组合逻辑设计从“挑选经典结构”走向“搜索最优结构”

	Kogge-Stone 加法器	Winograd 卷积
关键参数	稀疏度、基数、扇出/布线、节点结构	tile 尺寸、插值点、位宽、变换加法树
空间形态	图结构、离散、规模极大	离散/连续混合、规模中等
适合方法	RL 图编辑、进化算法	多保真贝叶斯优化
优化目标	延迟–面积 Pareto 前沿	能耗/面积 + 精度约束

1 参数化生成器是前提

结构必须能由参数自动生成 RTL，搜索才能闭环

2 评估保真度要分层

便宜模型负责广度，精确综合负责深度

3 正确性由构造保证

合法化步骤确保每个候选在功能上都正确