北京大学
PEKING UNIVERSITY

基于AI的数字系统设计

第8讲：AI辅助时序逻辑设计

主讲：陶耀宇

2026年秋季

北京大学版权所有

北京大学
PEKING UNIVERSITY

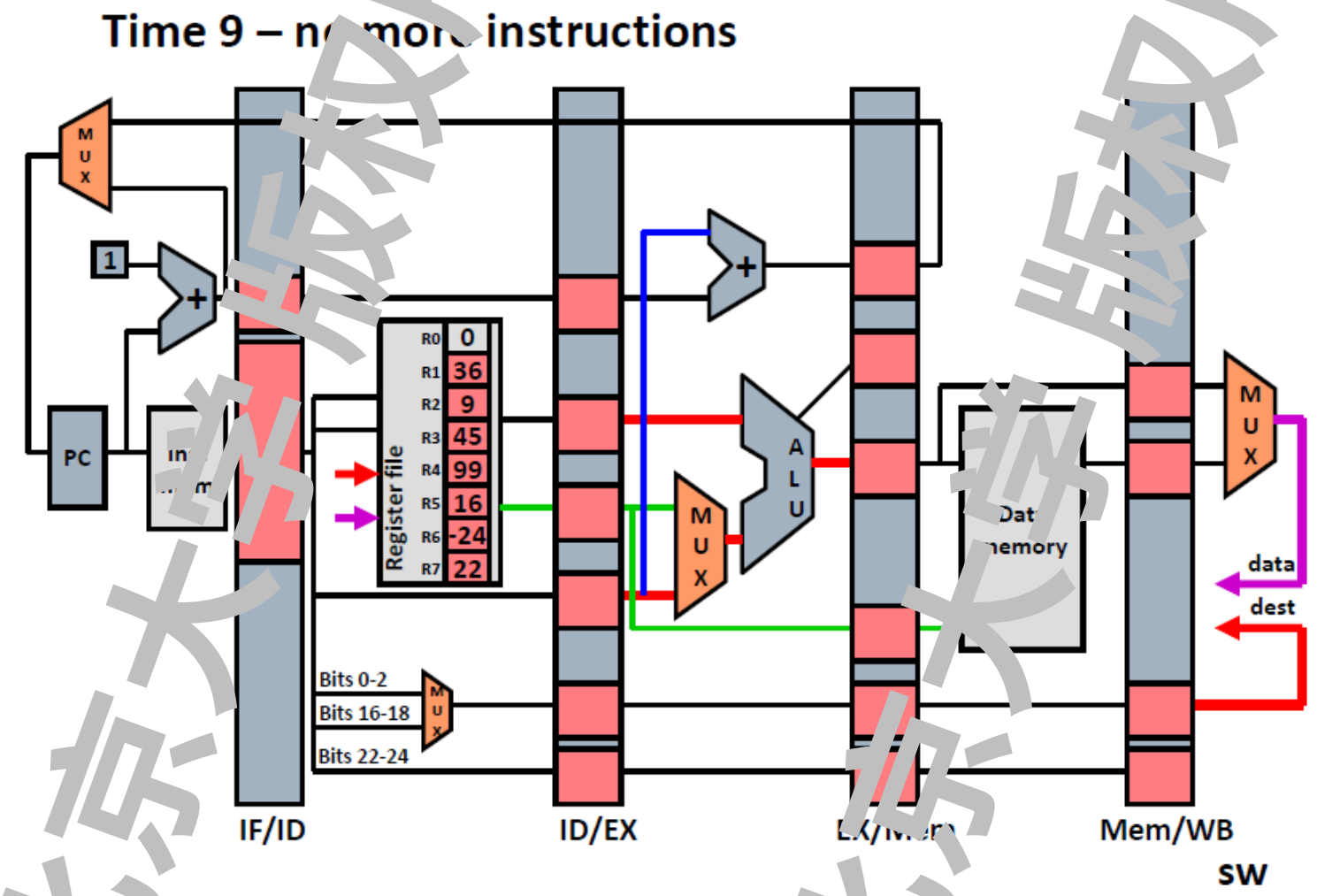
AI辅助时序逻辑设计 – 流水线

北京大学版权所有

5级流水线的实现案例

- 假设运行以下指令在5级流水线上

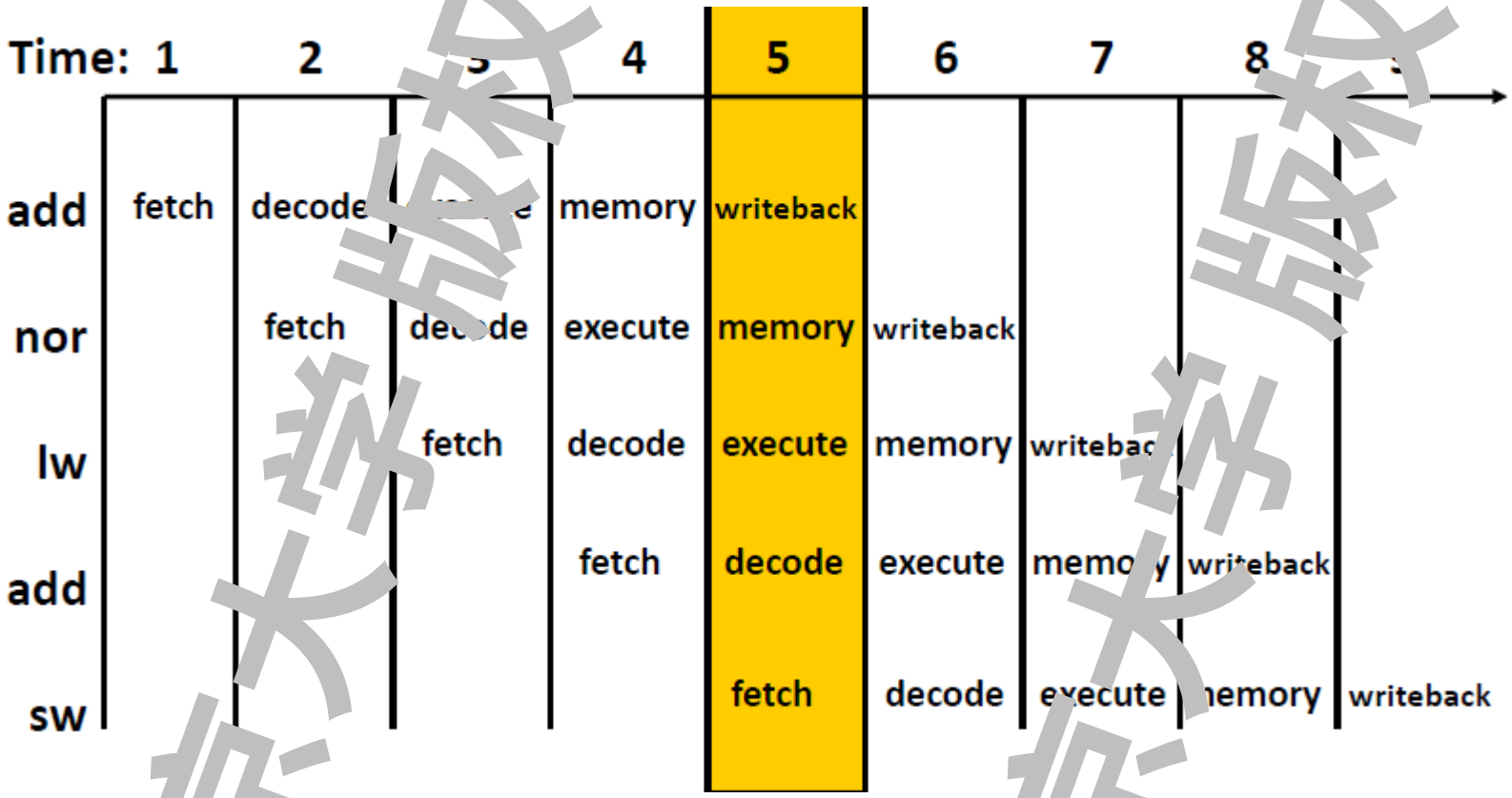
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10] =reg 7



5级流水线的实现案例

- 假设运行以下指令在5级流水线上

- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10] =reg 7



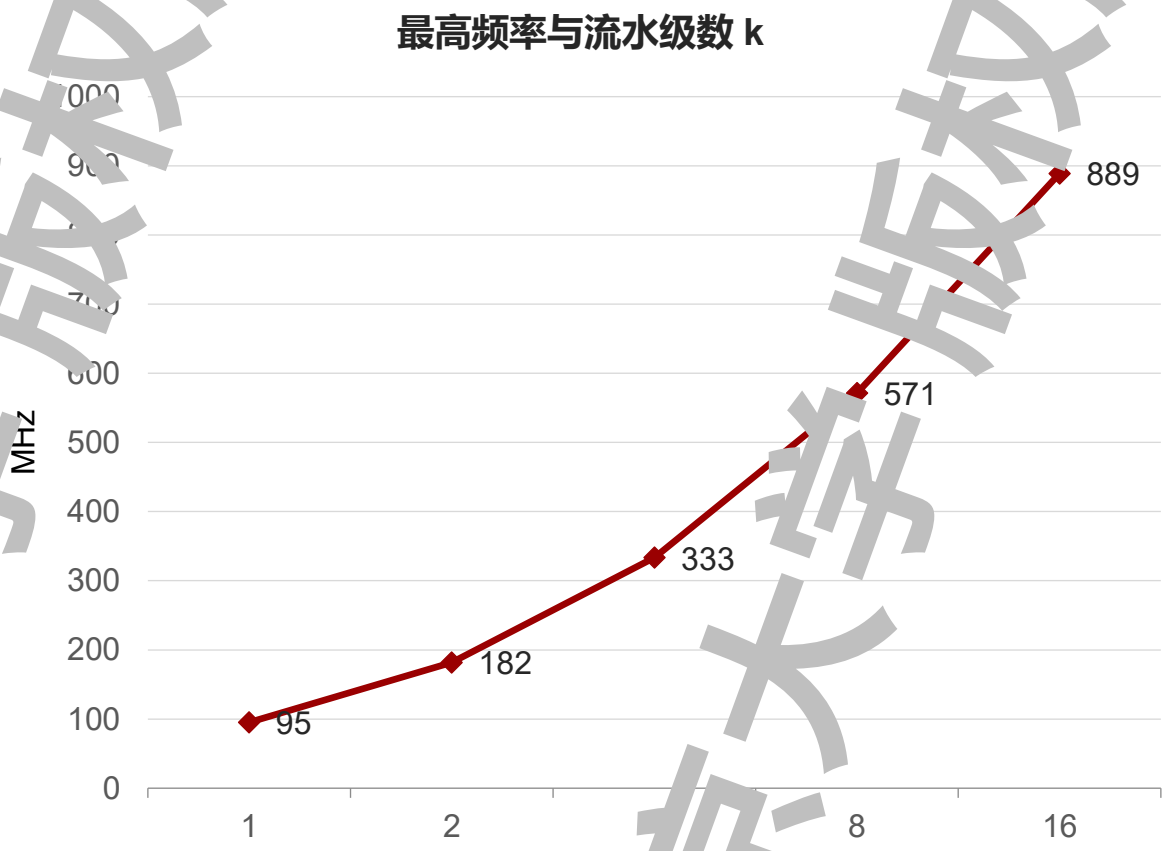
为什么需要做这样的多级流水线

- 流水线用**更多寄存器和更长延迟换取更高时钟频率与吞吐率**

k 级流水线的时钟周期

$$T_{clk} \approx T_{logic} / k + (T_{cq} + T_{su})$$

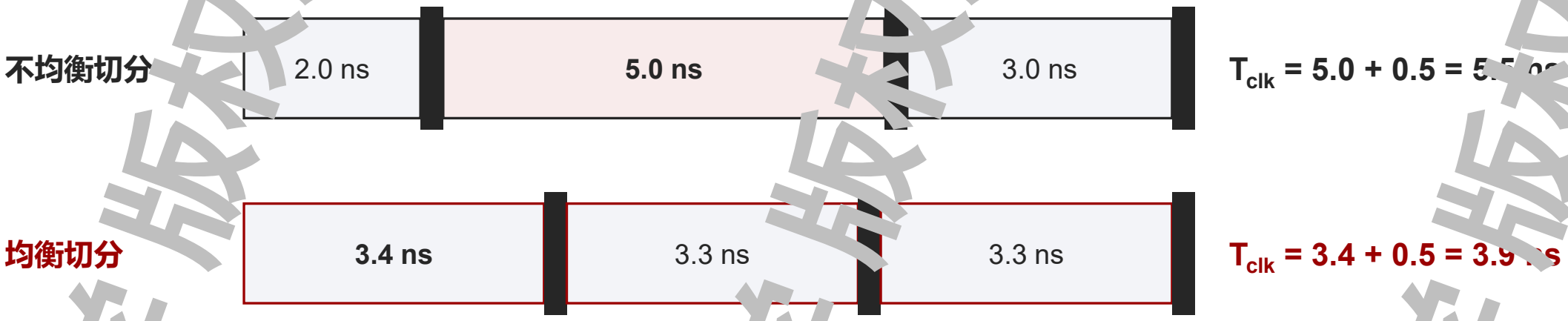
- **吞吐率**：理想情况下**每拍输出一个结果，约提升 k 倍**
- **延迟**：$k \times T_{clk}$，随级数增加而变长
- **收益递减**：寄存器开销与时钟偏斜不随 k 缩小
- **代价**：寄存器面积、功耗、控制复杂度上升



假设：T_{logic} = 10 ns，寄存器开销 0.5 ns；横轴为流水级数 k，延迟从 10.5 ns 增至 18 ns

流水线划分原则

- 时钟周期由最慢的一级决定：切分点要让各级延迟尽量均衡



黑色竖条表示流水线寄存器

1 均衡各级延迟

- 以 STA 报告为依据，而非代码行数
- 关注最差一级的裕量

2 在“最小”切分

- **选择位宽小的位置插寄存器**
- 寄存器数量、面积与功耗低

3 控制信号与数据同步打拍

- valid、tag 与数据一起流水
- 旁路与使能信号同样要对齐

流水线冒险

- 冒险破坏“每拍一条”的理想节奏；**转发、停顿与预测**是三类基本对策



x1 在第 5 拍才写回，但 sub 在第 4 拍就要用它

转发把 EX 结果直接送回下一条指令的 EX 输入

1 结构冒险

- 多条指令争用同一硬件资源
- 对策：资源复制、仲裁、分时
- 例：单端口存储器同时取指与访存

2 数据冒险

- 后续指令依赖未写回的结果
- 对策：转发，load-use 需停顿一拍
- 例：load 后紧跟使用该寄存器

3 控制冒险

- 分支结果晚于取指
- 对策：分支预测、冲刷错误路径
- 例：条件分支、跳转、异常

停顿与反压（Backpressure）

- valid 随数据**向前**传播，ready **向后**传播；下游忙时整条流水线逐级停住



红：valid / data 前向　蓝：ready 反向

```
// 单级流水寄存器（带反压）
assign in_ready = ~out_valid | out_ready;
always_ff @(posedge clk)
  if (in_ready) begin
    out_valid <= in_valid;
    out_data  <= in_data;
  end
```

问题：ready 是长组合路径

- in_ready 依赖 out_ready，逐级串联
- 流水级越多，这条反向路径越长，限制频率

对策：Skid Buffer

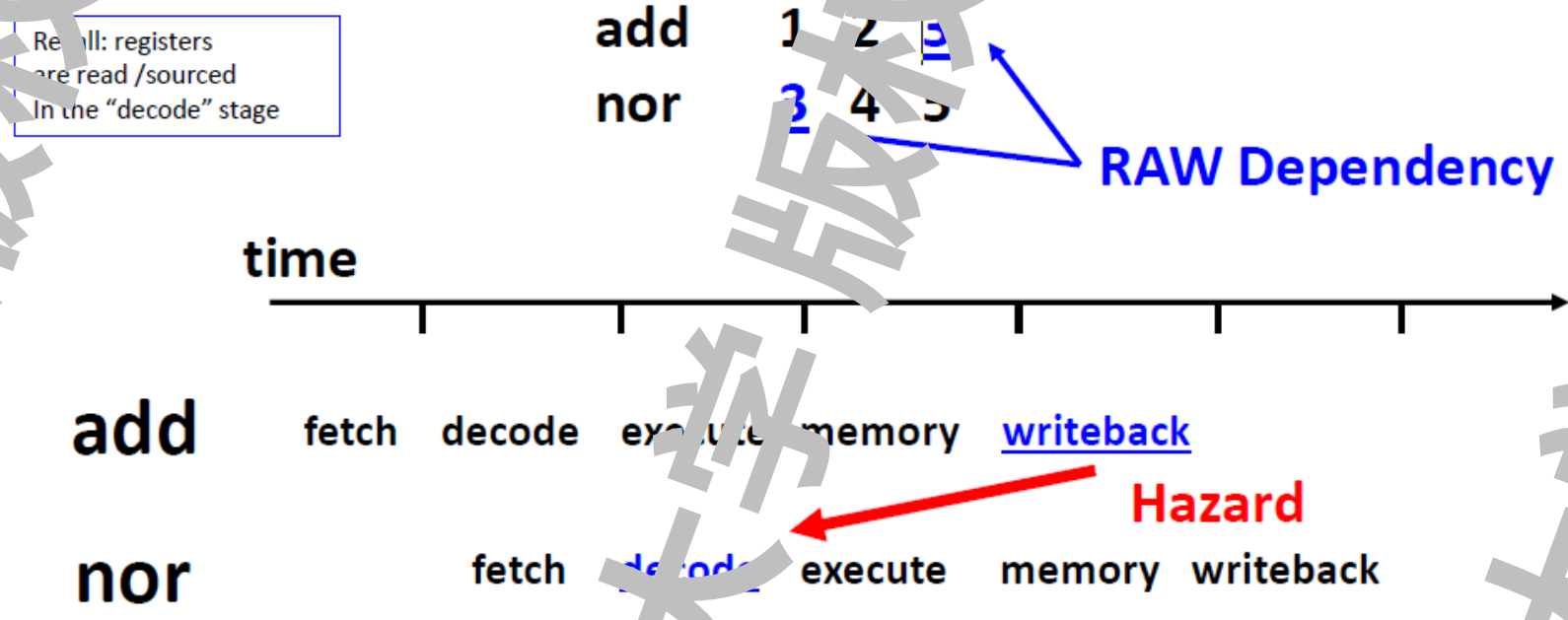
- 每级多放一个暂存槽
- ready 改为寄存输出，切断反向组合链
- 代价：每级多一份数据寄存器

简单5级流水线A存在什么问题?

- **Data hazards:** since register reads occur in stage 2 and register writes occur in stage 5 it is possible to read the wrong value if it is about to be written.
- **Control hazards :** A branch instruction may change the PC, but not until stage 4. What do we fetch before that?
- **Exceptions:** Sometimes we need to pause execution, switch to another task (maybe the OS), and then resume execution... how do we make sure we resume at the right spot

问题1：Data Hazards

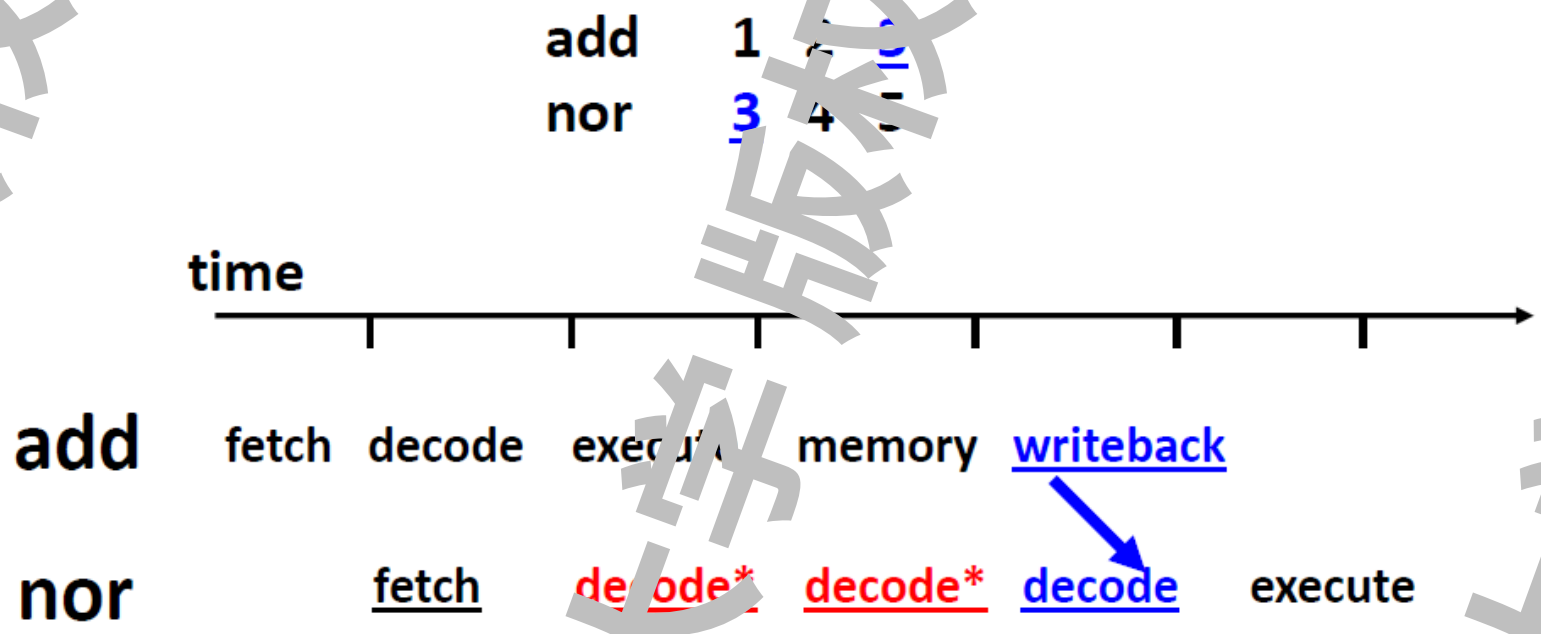
- RAW问题：Read After Write数据冲突



If not careful, nor will read a stale value of register 3

问题1：Data Hazards

- RAW问题：Read After Write数据冲突

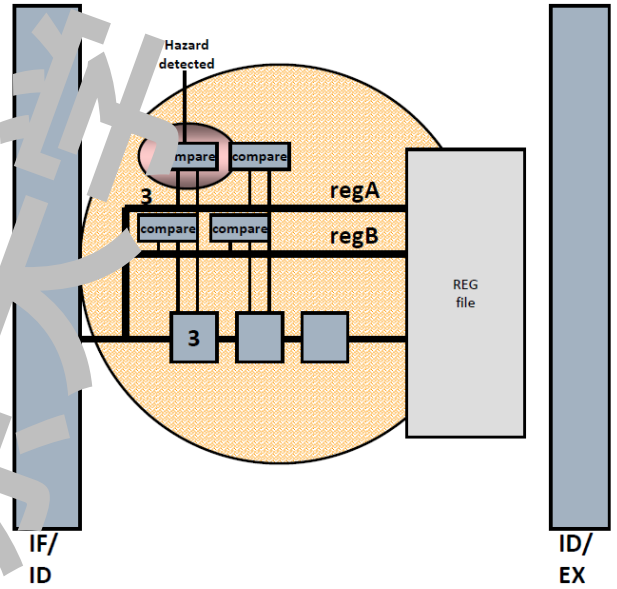


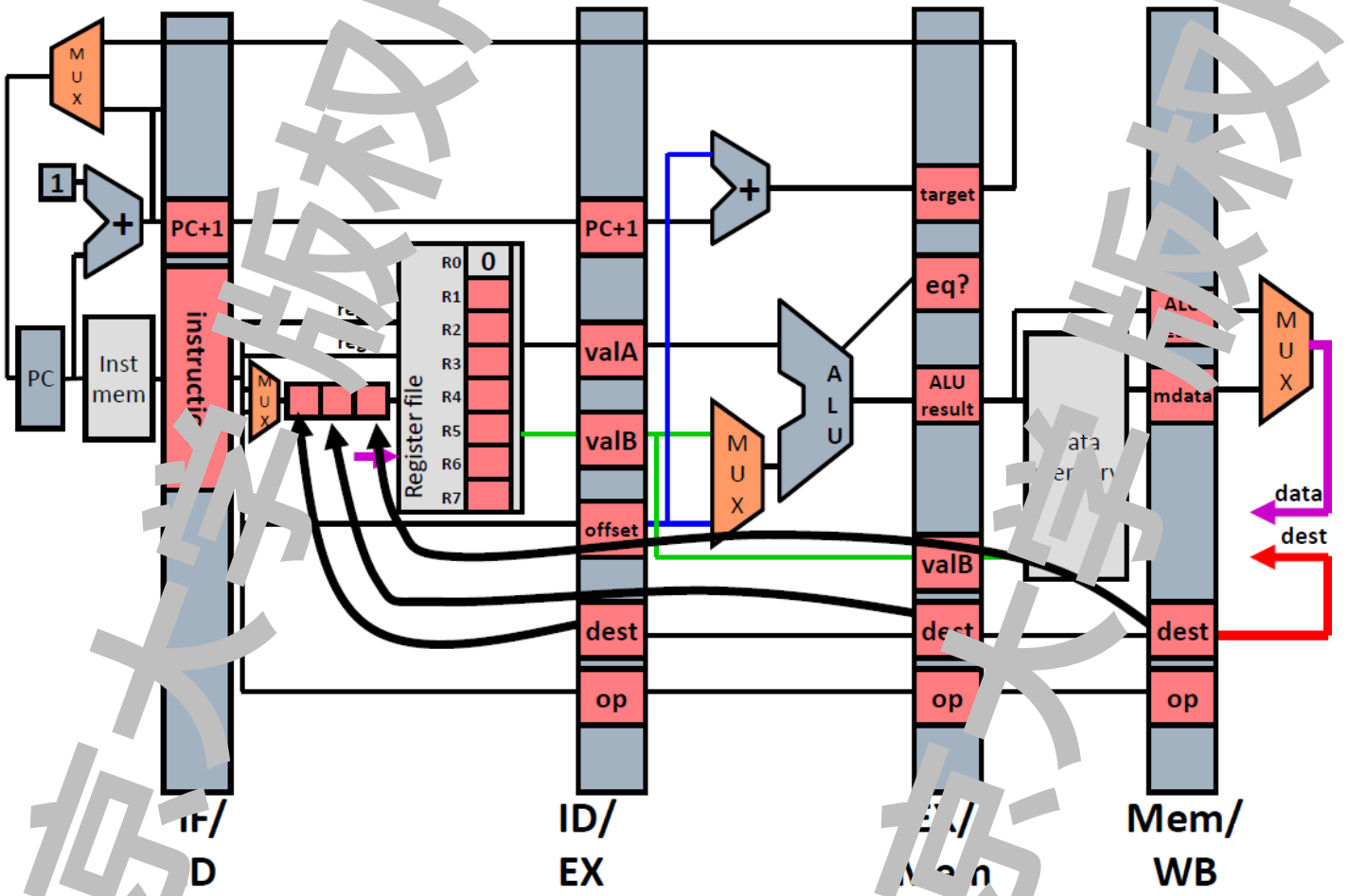
简单解决办法：流水线停顿（Pipeline Stall）

问题1：Data Hazards

- RAW问题：Read After Write数据冲突

1. add 1 2 3
2. nor 3 4 5
3. add 6 3 7
4. [illegible] 3 6 10
5. [illegible] 6 2 12

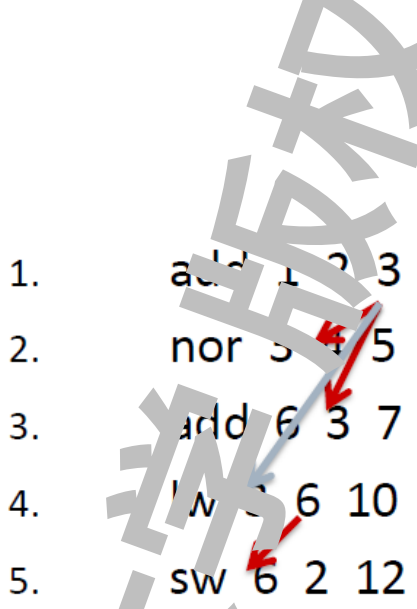


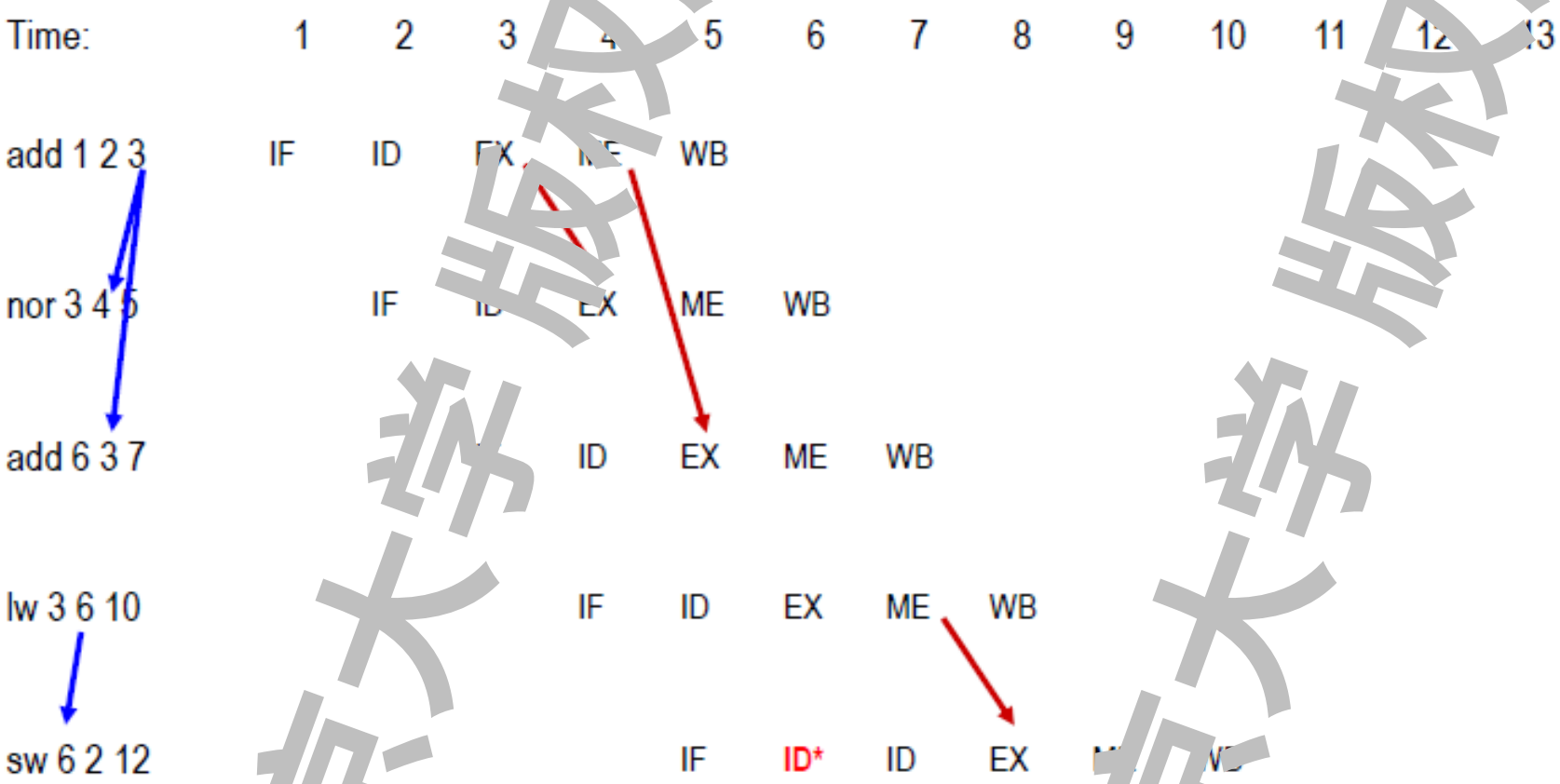


进阶解决办法：Detect and Forward

问题1：Data Hazards

- RAW问题：Read After Write数据冲突





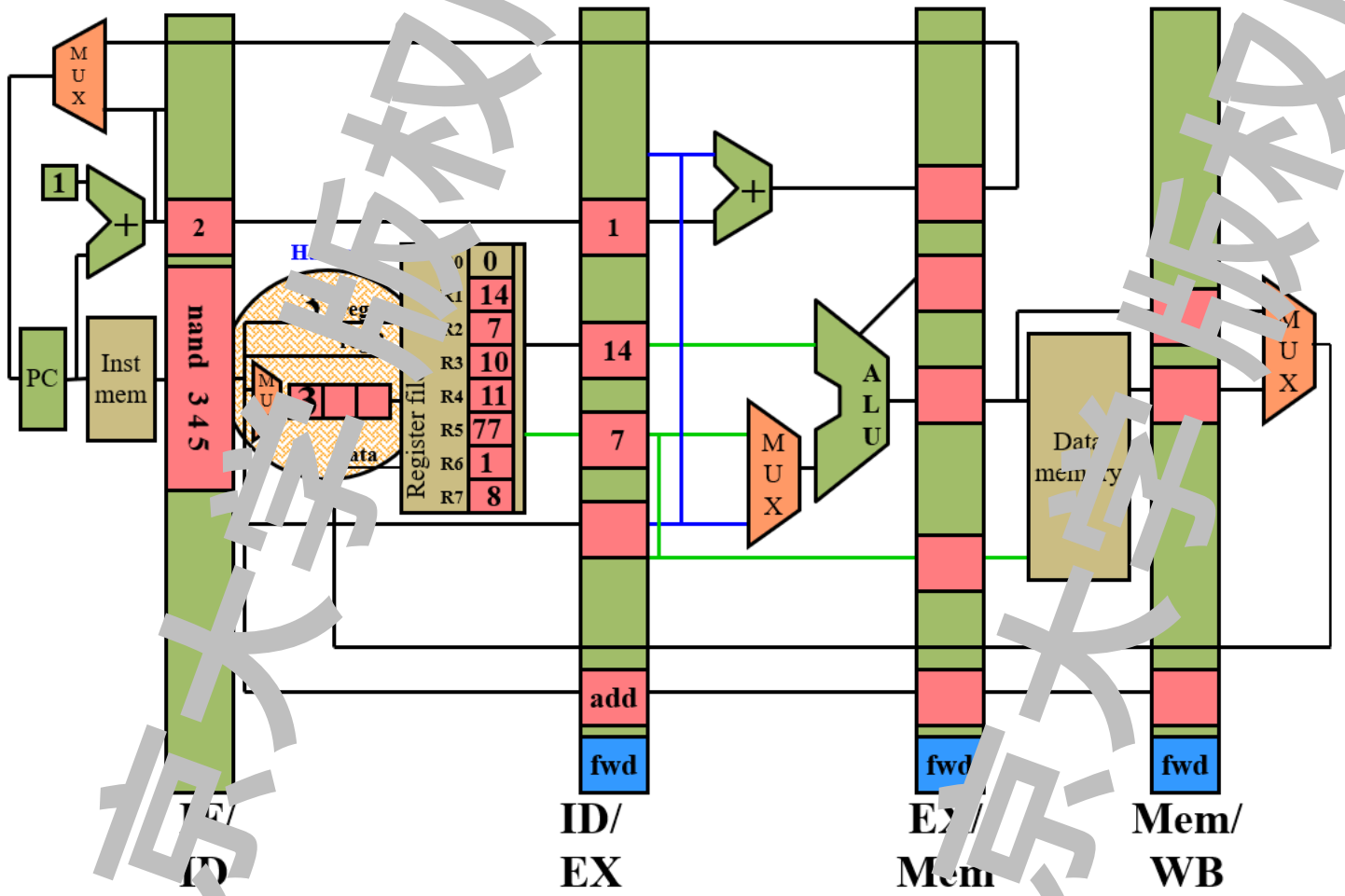
进阶解决办法：Detect and Forward

问题1：Data Hazards

- Detect and forward案例

```
add 1 2 3     // r1 = r1 + r2
nand 3 4 5    // r5 = r3 NAND r4
add 6 3 7     // r7 = r3 + r6
lw 3 6 10     // r6 = MEM[r3+10]
sw 6 2 12     // MEM[r6+12]=r2
```

Cycle 3前半段

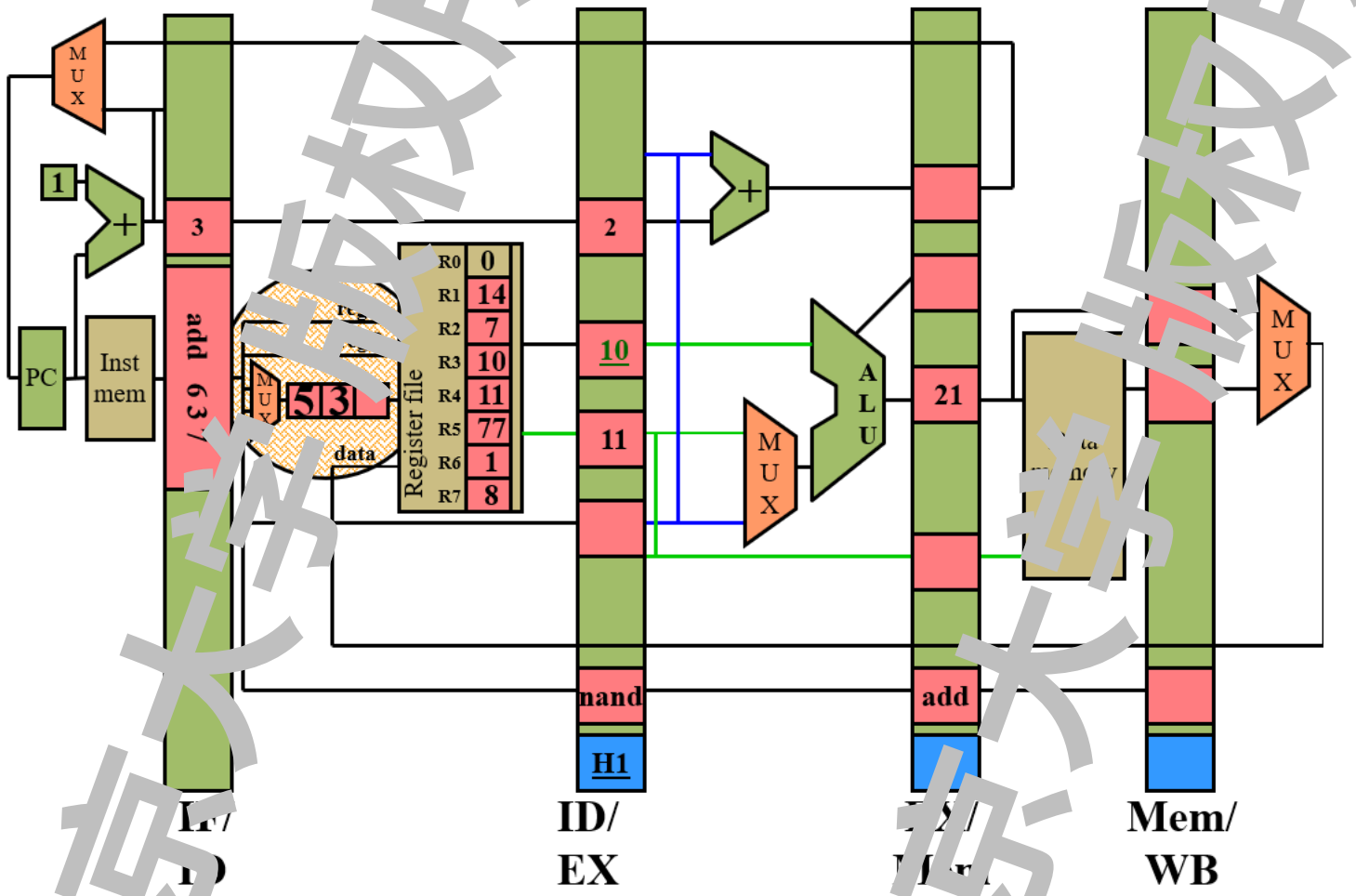


问题1：Data Hazards

- Detect and forward案例

```
add 1 2 3     // r3 = r1 + r2
nand 3 4 5    // r5 = r3 NAND r4
add 6 3 7     // r7 = r3 + r6
lw 3 6 0      // r6 = MEM[r3+10]
sw 6 2 12     // MEM[r6+12]=r2
```

Cycle 3后半段

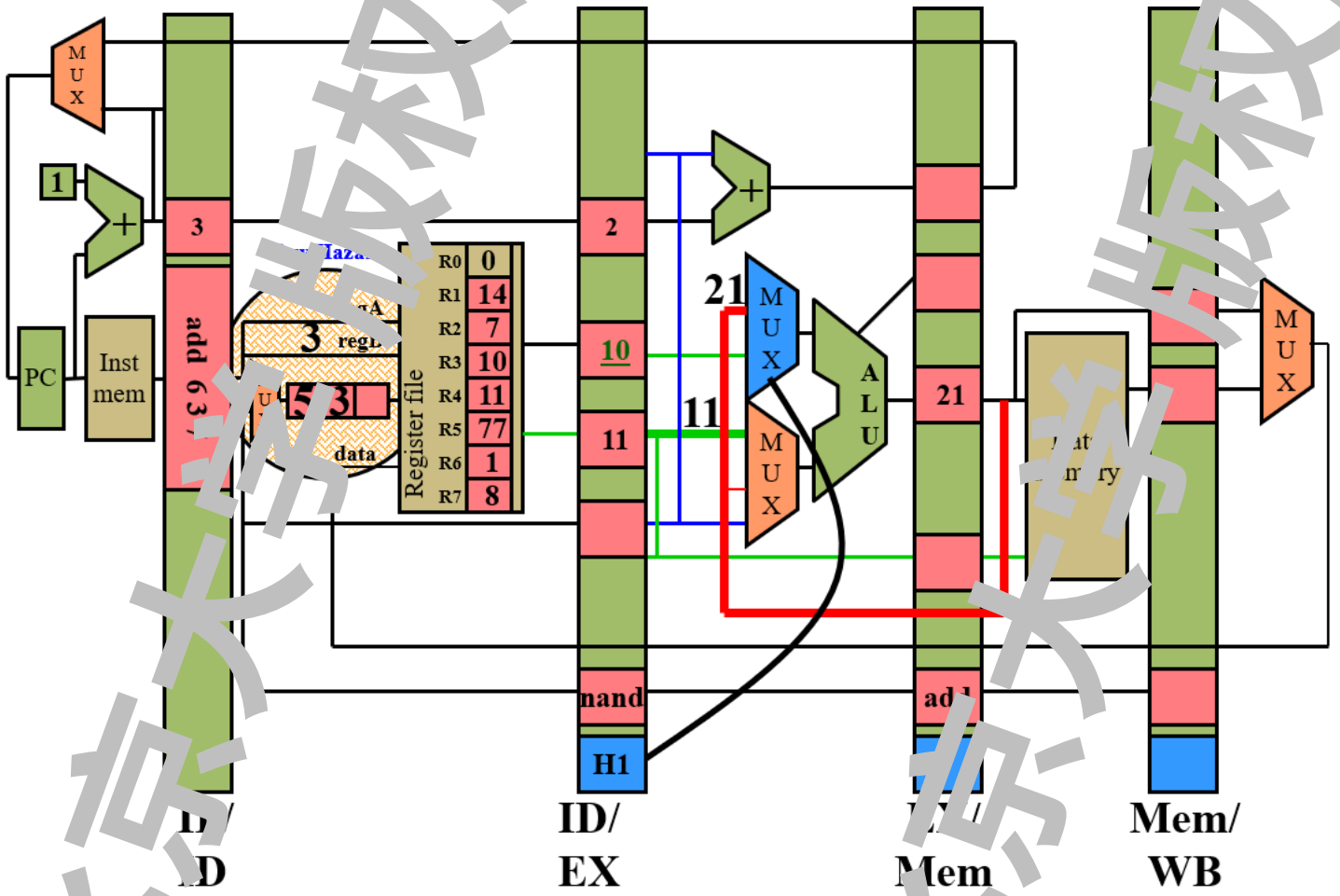


问题1：Data Hazards

- Detect and forward案例

```
add 1 2 3     // r3 = r1 + r2
nand 3 4 5    // r5 = r3 NAND r4
add 6 3 7     // r7 = r3 + r6
lw 3 6 10     // r6 = MEM[r3+10]
sw 6 2 12     // MEM[r6+12]=r2
```

Cycle 4前半段（forwarding）



问题1：Data Hazards

- Detect and forward案例

add 1 2 3 // r1 + r2 = 21

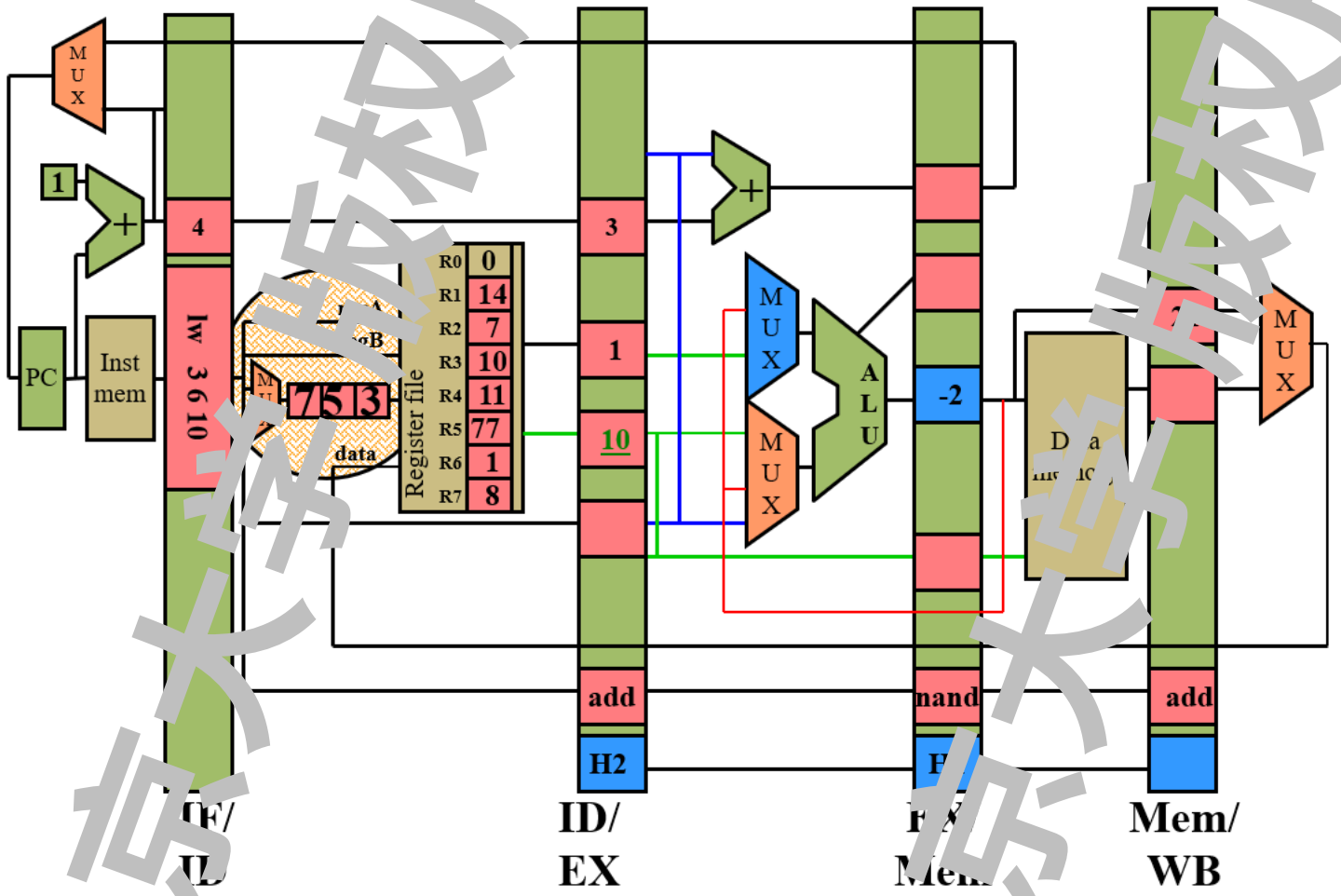
nand 3 4 5 // r5 = r3 NAND r4

add 6 3 7 // r7 = r3 + r6 = 22

lw 3 6 0 // r6 = MEM[r3+10]

sw 6 2 12 // MEM[r6+12]=r2

Cycle 4后半段



问题1：Data Hazards

- Detect and forward案例

add 1 2 3 // r3 = r1 + r2

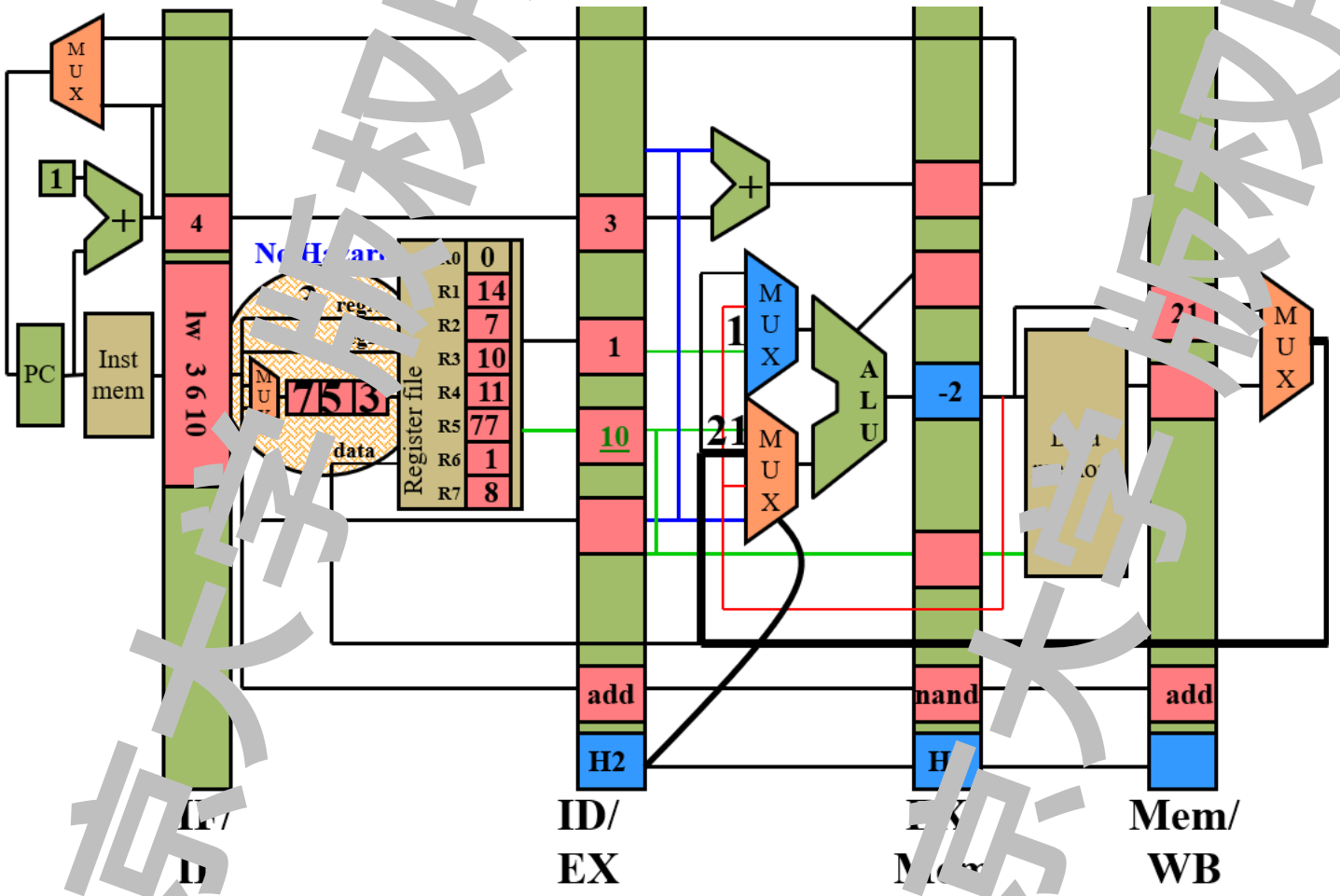
nand 3 4 5 // r5 = r3 NAND r4

add 6 3 7 // r7 = r3 + r6

lw 3 6 0 // r6 = MEM[r3+10]

sw 6 2 12 // MEM[r6+12]=r2

Cycle 5前半段

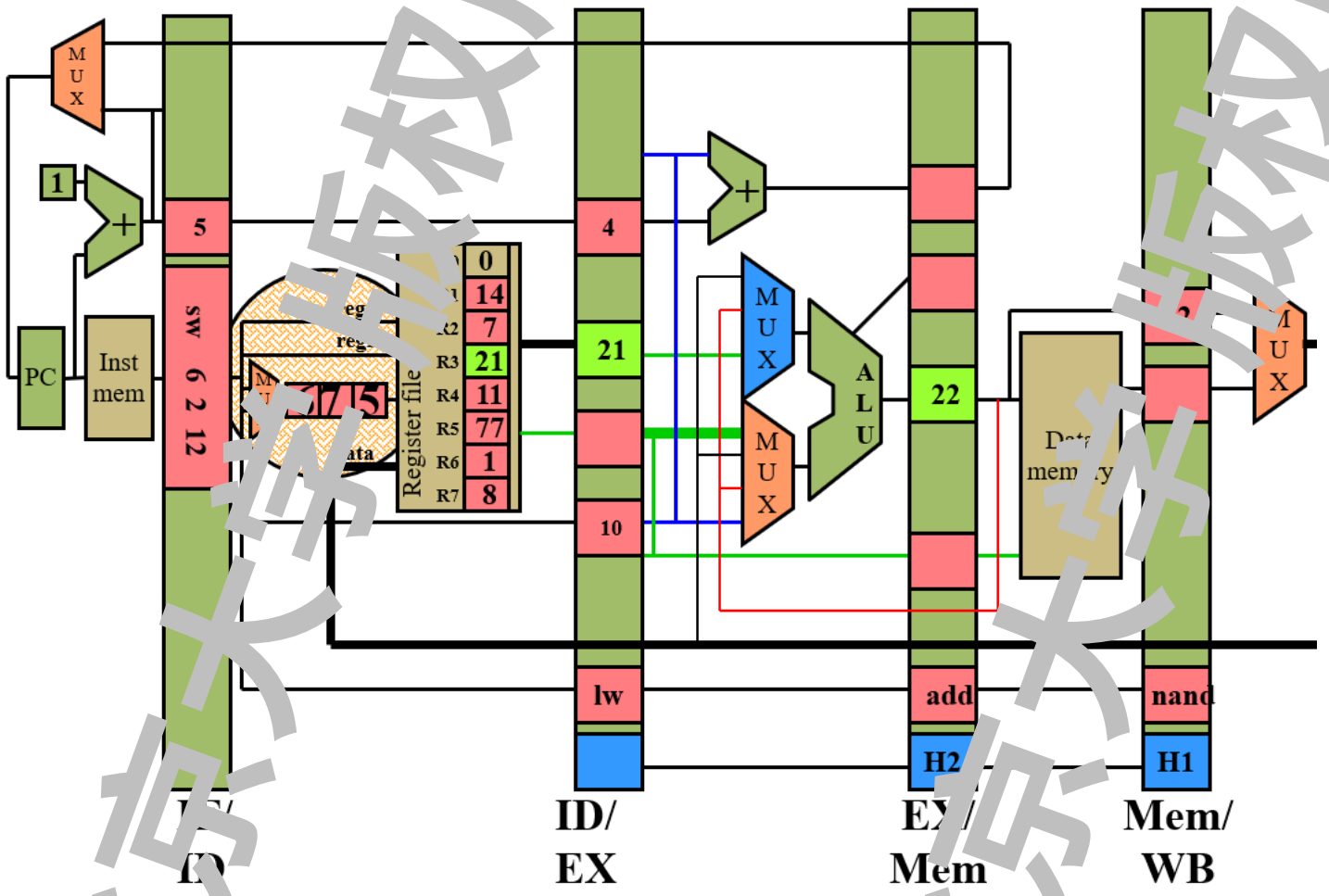


问题1：Data Hazards

- Detect and forward案例

```
add 1 2 3     // r3 = r1 + r2
nand 3 4 5    // r5 = r3 NAND r4
add 6 3 7     // r7 = r3 + r6
lw 3 6 10     // r6 = MEM[r3+10]
sw 6 2 12     // MEM[r6+12]=r2
```

Cycle 5后半段



问题1：Data Hazards

- WAW和WAR问题：Write After Write和Write After Read

- False or Name dependencies
  - WAW – Write after Write

    R1=R2+R3

    R1=R4+R5
  - WAR – Write after Read

    R2=R1+R3

    R1=R4+R5

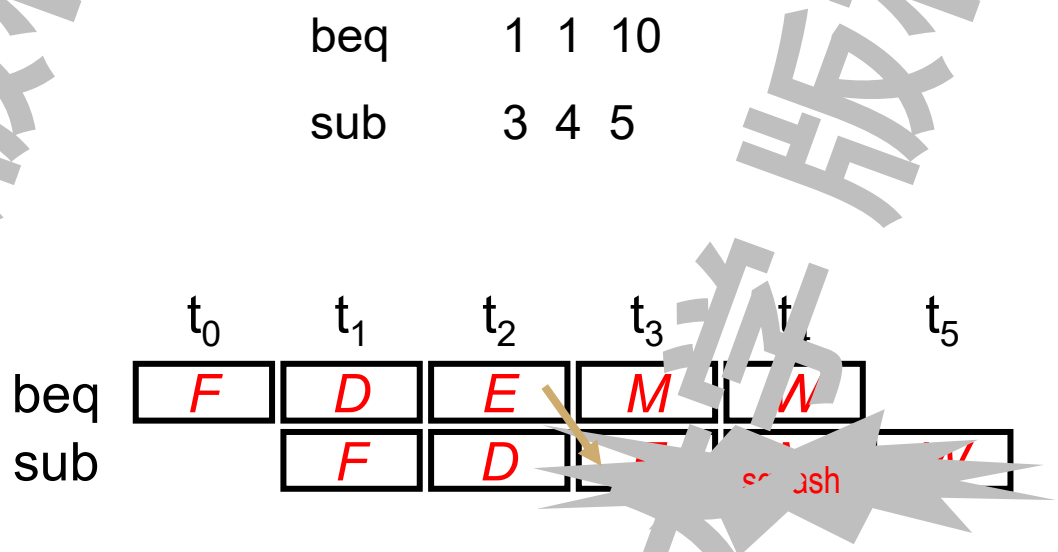
- 在顺序的单条5级流水线上不会出现问题

- **乱序执行则会出现问题，可利用Register重命名解决（后续乱序执行深入讲解）**

问题2：Control Hazards

- Branch类指令

  - Fetch: read instruction from memory
  - Decode, read source operands from reg
  - Execute: calculate target address and test for equality
  - Memory: **Send target to PC** if test is equal
  - Writeback: Nothing left to do



问题2：Control Hazards

- 如何解决Control Hazards

Avoidance (static)

- No branches?
- Convert branches to predication
  - Control dependence becomes data dependence

Detect and Stall (dynamic)

- Stop fetch until branch resolves

Speculate and squash (dynamic)

- Keep going past branch, throw away instructions if wrong

问题2：Control Hazards

- **Detection and Stall：检测-停顿机制**

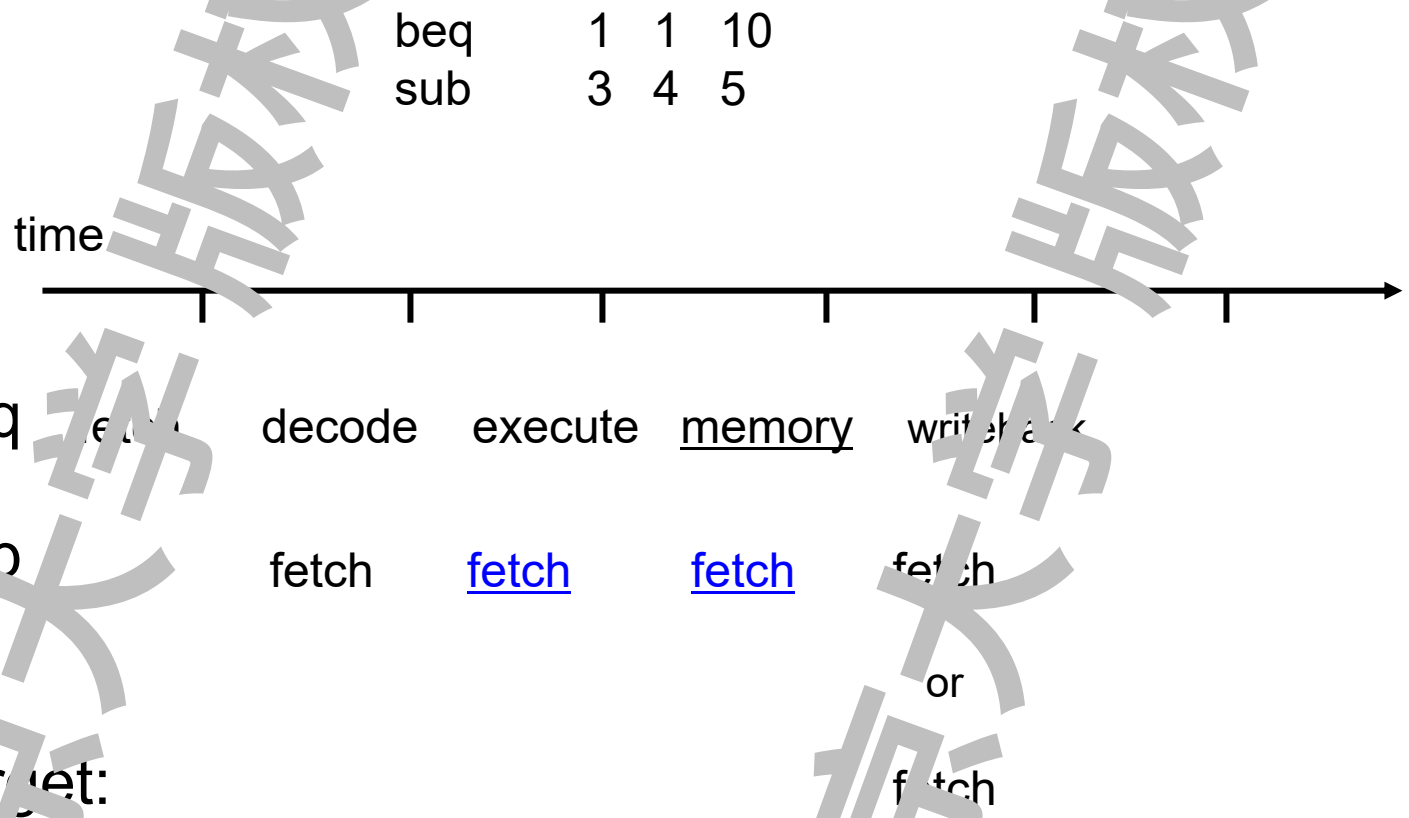
Detection

- In decode, check if opcode is branch or jump

Stall

- Hold instruction in Fetch
- Pass noop to Decode

- CPI increases on every branch
- Are these stalls necessary? Not always!
  - Assume branch is NOT taken
    - Keep fetching, treat branch as noop
    - If wrong, make sure bad instructions don’ t complete



问题2：Control Hazards

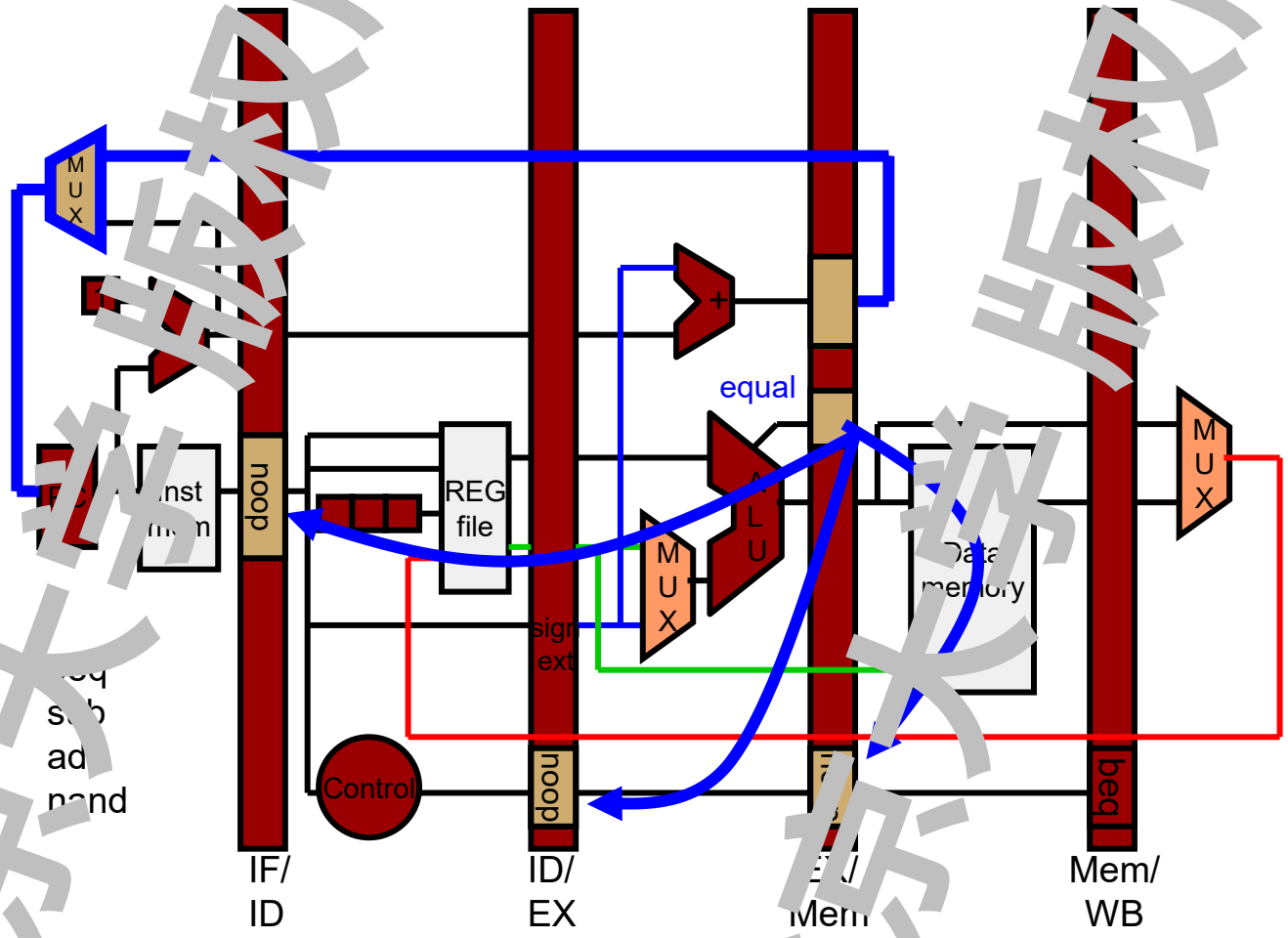
- Speculate and Squash：投机-制止机制

Speculate “Not-Taken”

- Assume branch is not taken

Squash

- Overwrite opcodes in Fetch, Decode, Execute with noop
- Pass target to Fetch



问题2：Control Hazards

- Speculate and Squash：投机-制止机制的问题

Always assumes branch is not taken

Can we do better? Yes.

- **Predict branch direction and target!**
- Why possible? Program behavior repeats.

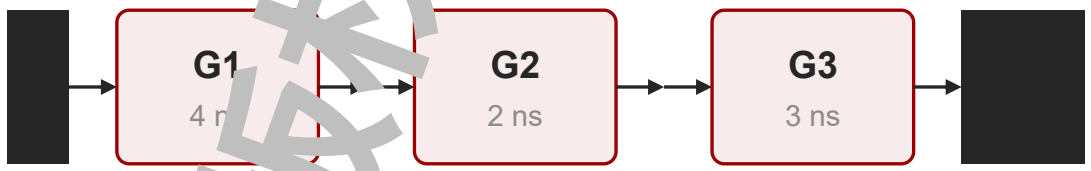
More on branch prediction to come...

北京大学 版权所有

重定时（Retiming）原理 – 在哪里插入寄存器最优？

- 在不改变输入/输出行为的前提下移动寄存器位置，约束关键路径

重定时前



关键路径 9 ns

重定时后：一个寄存器向前移过 G3、G2



关键路径 5 ns

- 寄存器总数与端到端延迟（拍数）保持不变
- 改变了内部状态编码 → 需要**时序等价性检查**

Leiserson–Saxe 形式化

- 电路 → 图 G(V, E)：节点延迟 $d(v)$，边上寄存器数 $w(e)$
- 重定时 $r : V \to \mathbb{Z}$

$$w_r(u \to v) = w(e) + r(v) - r(u)$$

- 约束 $w_r(e) \geq 0$，使所有零寄存器路径延迟 $\leq T$
- 可用最短路 / 整数线性规划求最小周期

寄存器平衡与多周期路径

- 寄存器平衡让工具自动搬寄存器，多周期路径放宽约束，两者都需要正确性保证

寄存器平衡（Register Balancing）



→ 工具把寄存器搬进乘法器内部

- RTL 建模后端集中写 N 级寄存器
- 综合或后端 retiming，自动分散到逻辑中
- 常用于乘法器、DSP、浮点单元

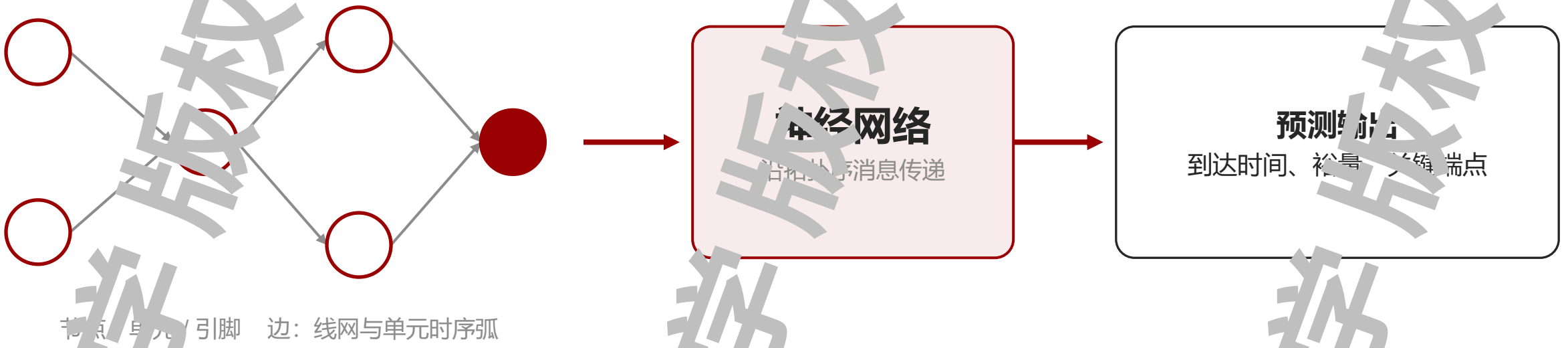
多周期路径（Multicycle Path）

```
set_multicycle_path 2 -setup \
  -from [get_cells a_reg*] -to [get_cells
set_multicycle_path 1 -hold \
  -from [get_cells a_reg*] -to [get_cells
```

- 允许路径用 2 个周期完成，但设计必须确保源数据保持 2 拍、目的端隔拍采样
- hold 约束需同步调整，否则会过度约束
- 错误的例外约束会掩盖真实违例：必须让 LLM 对照 RTL 使能逻辑**交叉审查**

AI 方法一：用神经网络预测时序

- 把网表（Verilog Netlist）当作图，在**布线前**预测到达时间与裕量，替代反复后端迭代



输入特征

- 单元类型、驱动强度
- 扇出、估计负载电容

训练标签

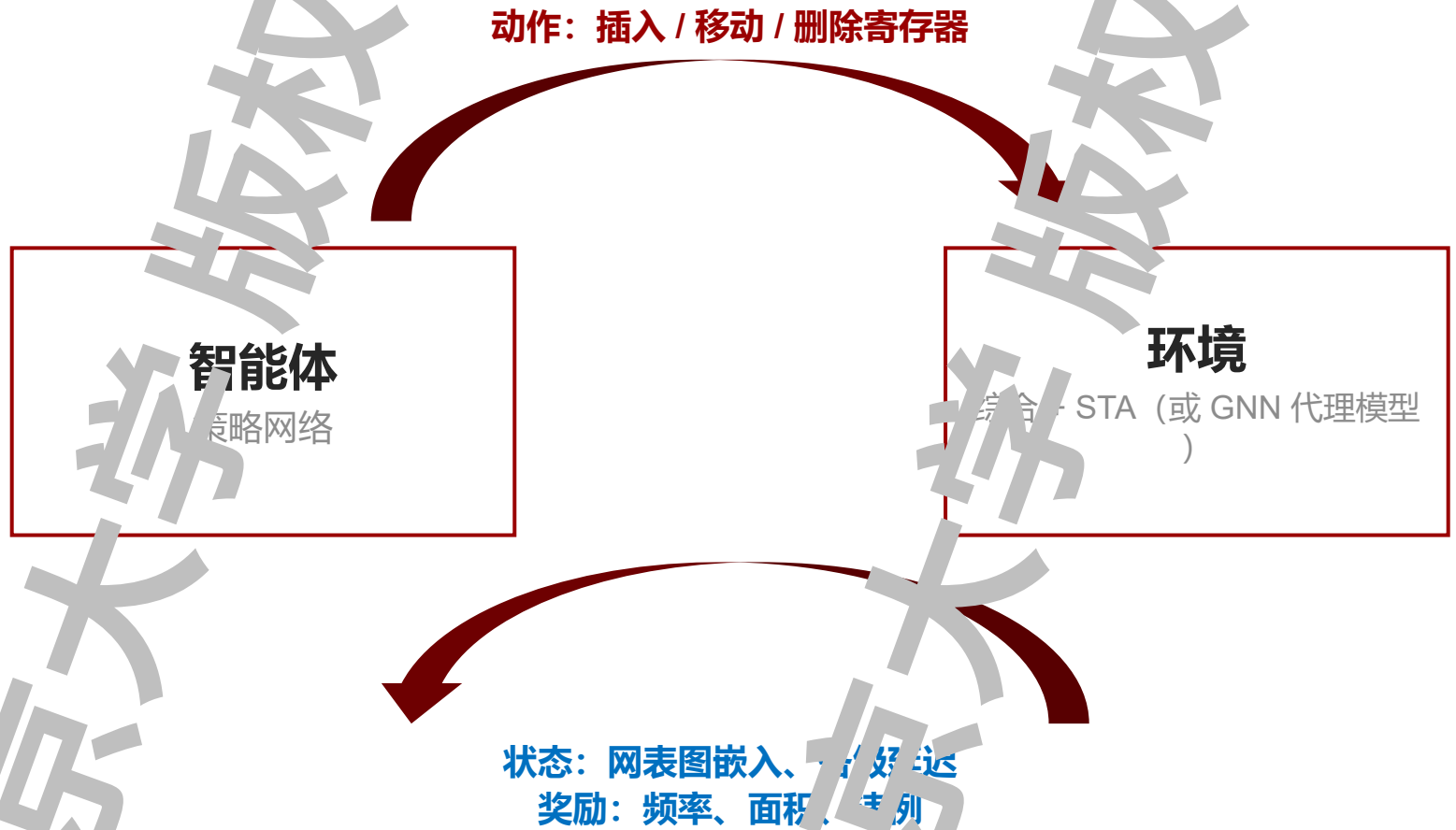
- 布线后 STA 的真实结果
- 跨设计训练，测试泛化

代表工作

- 时序引擎启发的 GNN 预测布线前裕量（DAC 2021）

AI 方法二：强化学习搜索流水线切分

- 把“在哪里插寄存器”建模为**序列决策**，用综合与 STA 的结果作为奖励



奖励函数示例

$R = -T_{clk} - \alpha \cdot N_{reg} - \beta \cdot$ [illegible]

- α、β 调节性能与面积的偏好
- STA 很慢 → 用 GNN 预测器做代理奖励，定期用真实 STA 校准
- 先例：深度强化学习用于芯片布局（Nature 2021）

基于AI的流水线优化

- 流水线优化 = 均衡切分 + 正确握手 + 工具验证，AI 加速其中的搜索与预测

1. 流水线提高频率与吞吐率，代价是延迟与面积
2. 冒险靠转发、停顿、预测解决；反压靠 valid / ready
3. 重定时在保持 I/O 行为的前提下搬移寄存器
4. AI预测时序、RL 搜索切分点，结果需真实 STA 确认

尝试实验：32 位乘加单元流水化

1. 让 LLM 生成无流水的乘加单元并通过仿真
2. 要求插入 2 / 3 / 4 级流水，valid 与数据保持对齐
3. 用 Synopsys DC 测量最高频率与寄存器数
4. 画出“频率—寄存器数”曲线，与理论趋势对比

北京大学
PEKING UNIVERSITY

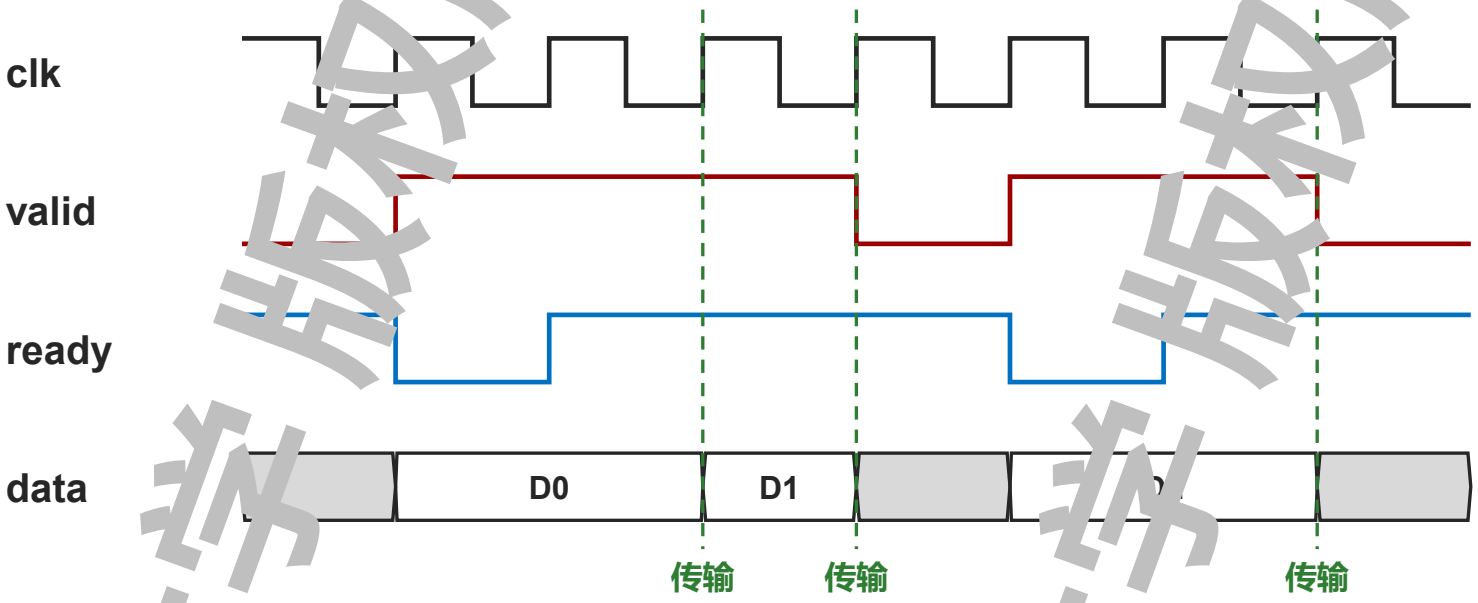
AI辅助时序逻辑设计

握手协议、弹性电路与延迟不敏感设计

北京大学版权所有

valid / ready 握手

- 当且仅当时钟沿上 valid && ready 同时为 1，数据才完成一次传输



绿色虚线：完成传输的时钟沿（D0 等待 1 拍，D2 等待 1 拍）

这套规则也是 AXI、AXI-Stream 等片上协议每个通道的基础

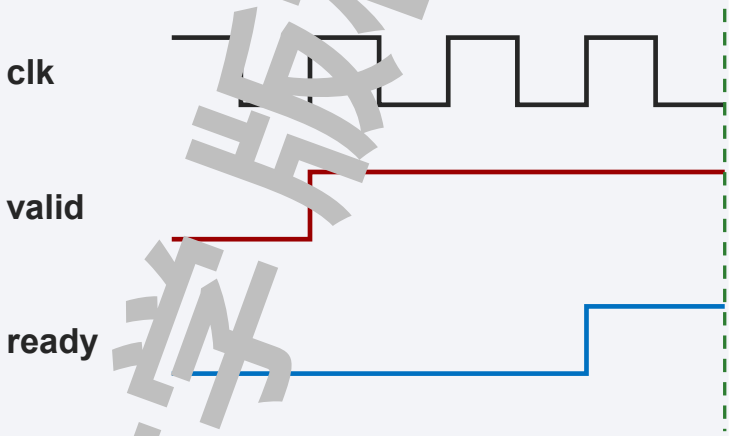
协议规则

- valid 拉高后必须保持，直到
- valid **不得**组合依赖 read
- 等待期间 data 必须保持
- ready **可以**依赖 valid
- 复位期间 valid 为 0

握手的三种时序关系

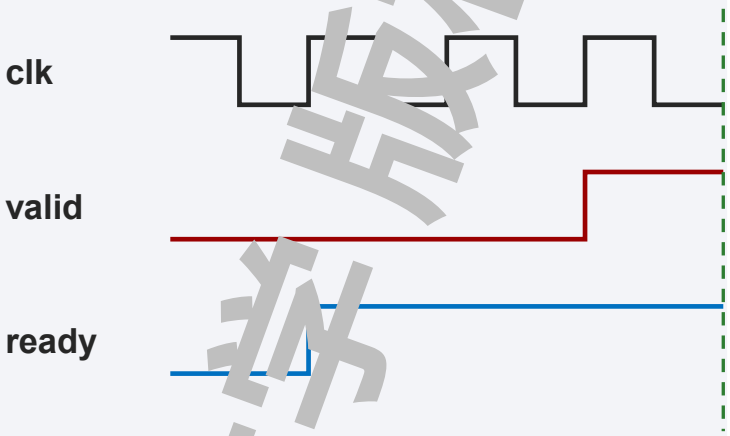
- 谁先拉高都可以，但规则不变：传输只发生在两者同时为高的时钟沿

1 valid 先到



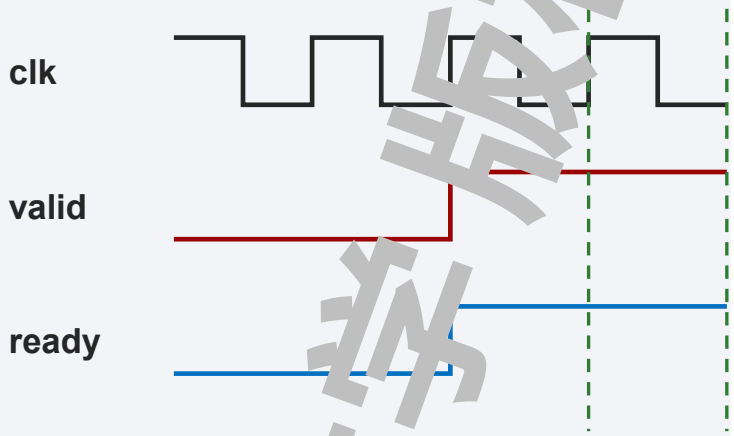
下游已经上游保持 valid 与数据，等待 ready

2 ready 先到



下游空闲提前准备好，valid 一到即传输

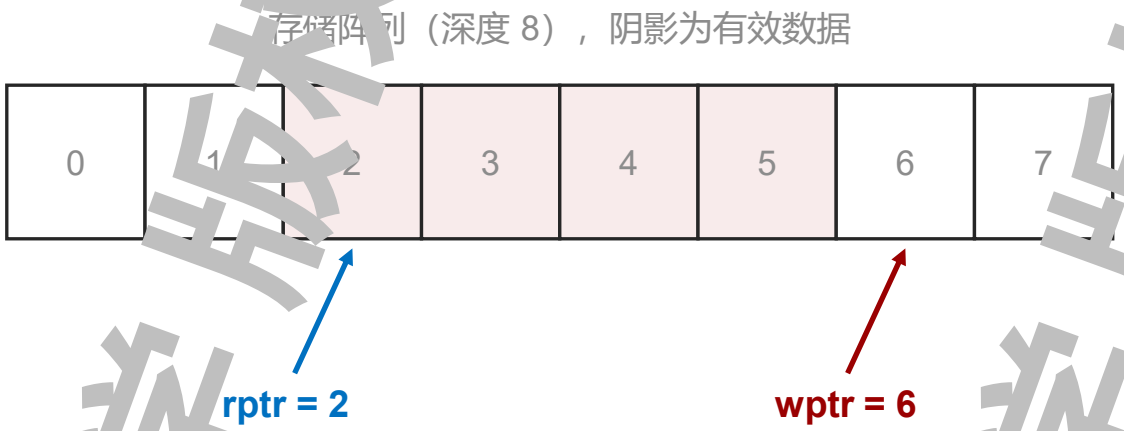
3 同时到达



连续两拍同时为高，每拍传输一个数据

FIFO：最常用的单性缓冲

- 用多一位的读写指针区分“空”和“满”：低位相等时，最高位相同为空，不同为满



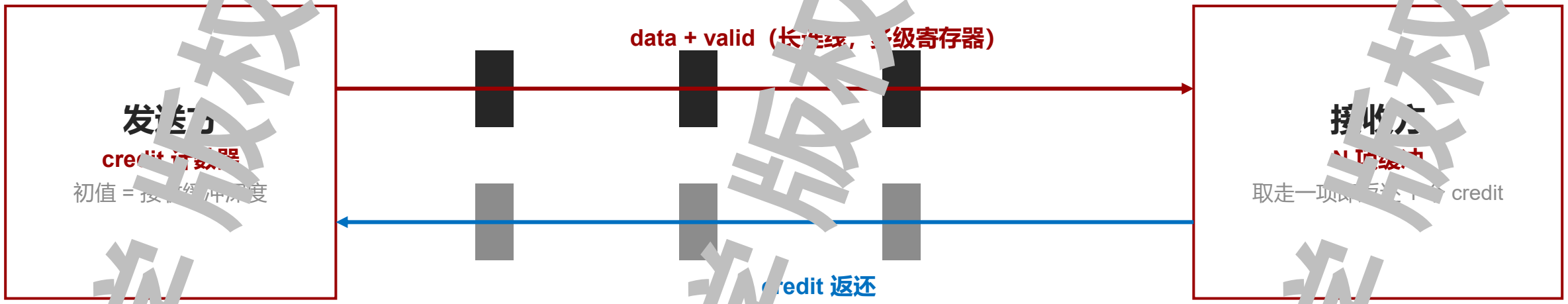
条件	判断（4 位指针，深度 8）
空 empty	wptr == rptr
满 full	wptr[3] != rptr[3] && wptr[2:0] == rptr[2:0]

sync_fifo.sv

```
// 同步 FIFO 核心逻辑（深度 8）
logic [3:0] wptr, rptr;
wire  wr_en = in_valid  & ~full;
wire  rd_en = out_ready & ~empty;
assign in_ready  = ~full;
assign out_valid = ~empty;
assign empty = (wptr == rptr);
assign full  = (wptr[3] != rptr[3]) &&
               (wptr[2:0] == rptr[2:0]);
always_ff @(posedge clk)
  if (wr_en) mem[wptr[2:0]] <= in_data;
always_ff @(posedge clk or negedge rst_n)
  if (!rst_n) {wptr, rptr} <= '0;
  else begin
    wptr <= wptr + wr_en;
    rptr <= rptr + rd_en;
  end
```

信用（Credit）流控

- 发送方预先知道接收方还有多少空位，不必等待 ready 的往返



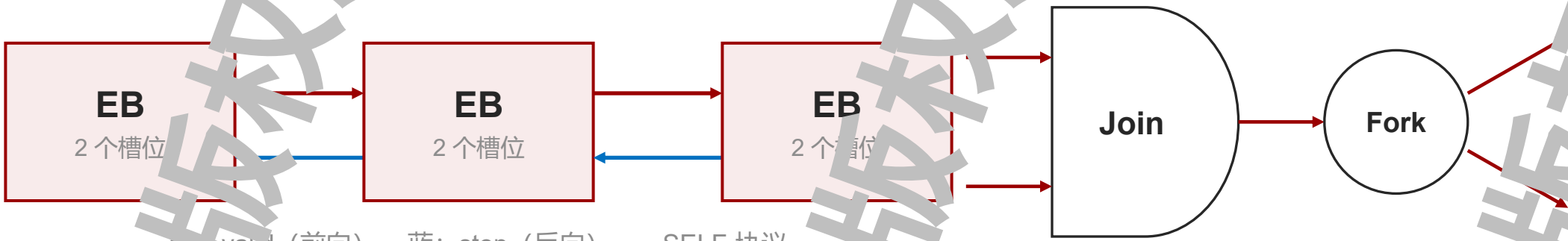
发送	credit > 0 才能发送，每发一个 credit −
返还	接收方释放一项，回传 1 个 credit
缓冲深度	N ≥ 往返延迟（拍） × 每拍数据量，才能满吞吐

与 valid / ready 的对比

- ready 反压要求一拍内往返，长连线上难以满足
- credit 把往返延迟“摊”进缓冲深度，常用于 NoC 与芯粒互连

弹性电路（Elastic Circuits）

- 把普通寄存器换成弹性缓冲（EB），电路可以容忍任意额外延迟而功能不变



红：valid（前向）　蓝：stop（反向）——SELF 协议

Join 等全部输入就绪；Fork 分发到多个后继

1 为什么是 2 个槽位

- st[illegible]寄存后晚到一拍
- 多一槽接住在途数据，保证满吞吐

2 带来的自由度

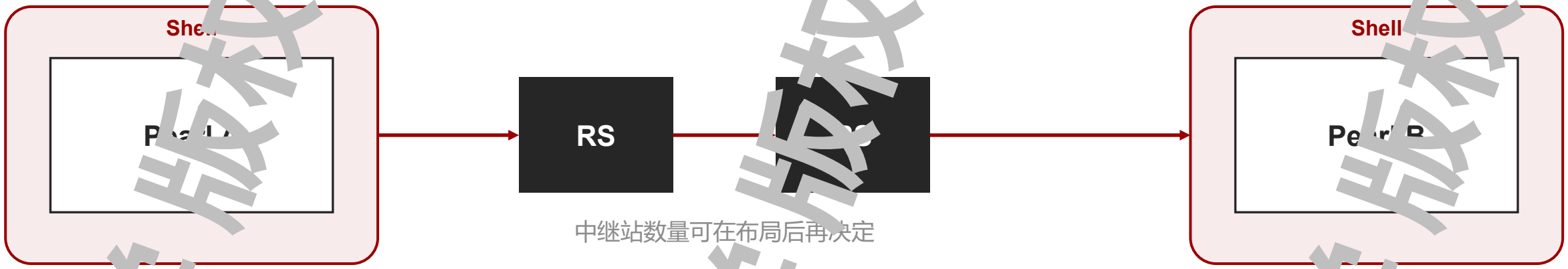
- 后期可[illegible]通道插入 EB
- 不必重[illegible]设计控制逻辑

3 代表工作

- Cortadella、[illegible]nevsky 等提出的同步弹性电路与 SELF 协议

延迟不敏感（La[illegible]cy-Insensitive）设计

- 模块（Pearl）外包一层外壳（Shell），长连线上插入中继站（Relay Station）



中继站数量可在布局后再决定

Pearl
功能模块本身，按同步设计，不感知延迟

Shell
数据未到齐时暂停 Pearl（门控时钟或使能）

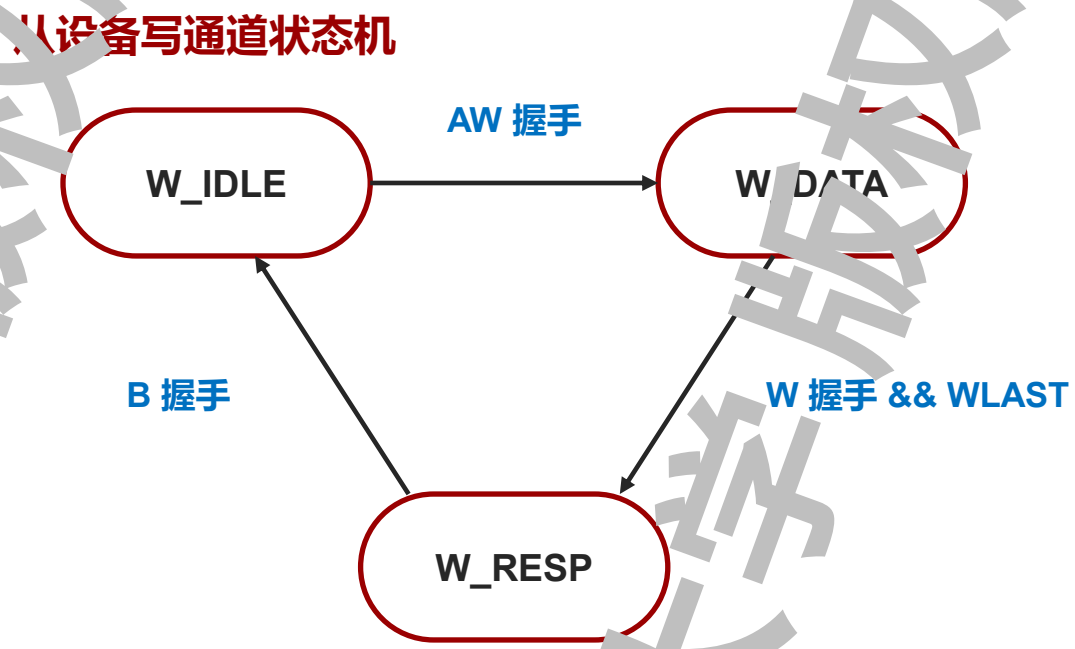
Relay Station
2 槽缓冲，切断长连线，可任意插入

意义：把“线延迟”从时序收敛问题变成**吞吐率问题**，由 Carloni 等提出，是弹性电路的现代源头

片上互连协议中的时序逻辑：AXI 类接口

- AXI = 5 个独立的 valid/ready 通道；接口控制器就是协调这些通道的状态机





- 通道之间相互独立，写数据可以早于写地址到达
- 突发传输：LEN、SIZE、BURST，最后一拍 WLAST / RLAST

每个箭头都是一次 valid && ready；状态机还要负责地址递增与 BRESP 生成

AI 实践：从协议文档生成接口控制器

- 长协议文档容易引发幻觉：先抽取带出处的规则清单，再生成代码与断言



编号	规则（LLM 抽取）	出处
R1	valid 拉高后保持，直到握手完成	握手章节
R2	等待期间数据与控制信号保持稳定	握手章节
R3	写响应必须在最后一个写数据之后	写事务章节
R4	复位期间所有 valid 为低	复位章节

提示要点

- 每条规则必须给出原文出处
- 一条规则对应一条断言
- 不确定的条目标记为“待人工确认”

AI 生成协议检查断言（SVA）

- 断言把规则变成**可执行的检查**；但断言本身也可能写错，需要 cover 与人工复核

axis_checker.sv

```
// R1: valid 拉高后保持，直到握手完成
a_valid_hold: assert property (
  @(posedge clk) disable iff (!rst_n)
  valid && !ready |=> valid);

// R2: 等待期间数据保持稳定
a_data_stable: assert property (
  @(posedge clk) disable iff (!rst_n)
  valid && !ready |=> $stable(data));

// R4: 复位后 valid 为低
a_reset_valid: assert property (
  @(posedge clk) $rose(rst_n) |-> !valid);

// 覆盖：确认握手确实会发生（防空真）
c_handshake: cover property (
  @(posedge clk) valid && ready);
```

LLM 生成断言的常见陷阱

空真（Vacuity）
前件永不成立，断言“永远通过”

|-> 与 |=> 混淆
同拍与下一拍蕴含写反

assert / assume 写反
把对设计的检查写成对环境的假设

每条 assert 配一条 cover，并在**故意改错**的设计上确认能报错

北京大学
PEKING UNIVERSITY

AI辅助时序逻辑设计

多时钟域与异步时序逻辑

北京大学版权所有

跨时钟域（CDC）问题

- 两个时钟没有固定相位关系时，STA **无法保证**跨域路径的建立/保持时间



1 亚稳态传播

- 数据在 clkB 采样窗口内跳变
- FF_B 输出不确定，后级读到不同值

2 多位数据不一致

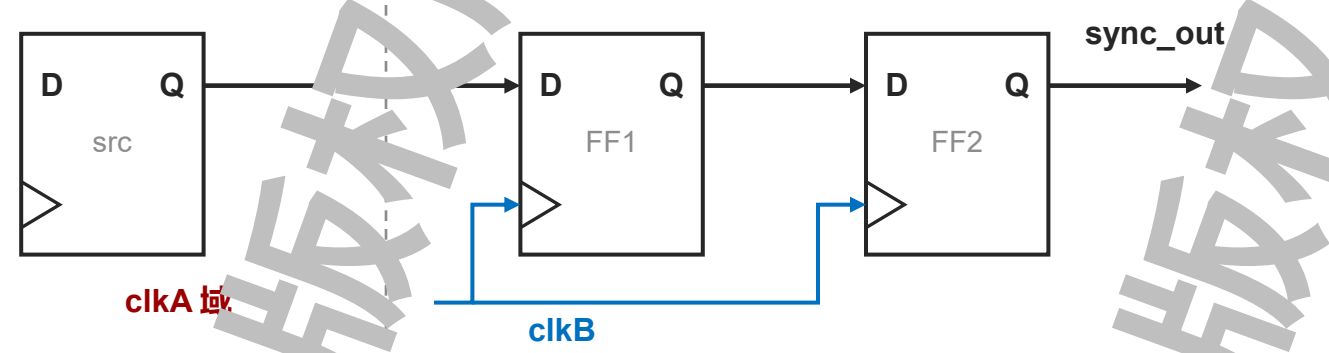
- 各位走线延迟不同
- 同一拍采到“一半新、一半旧”的值

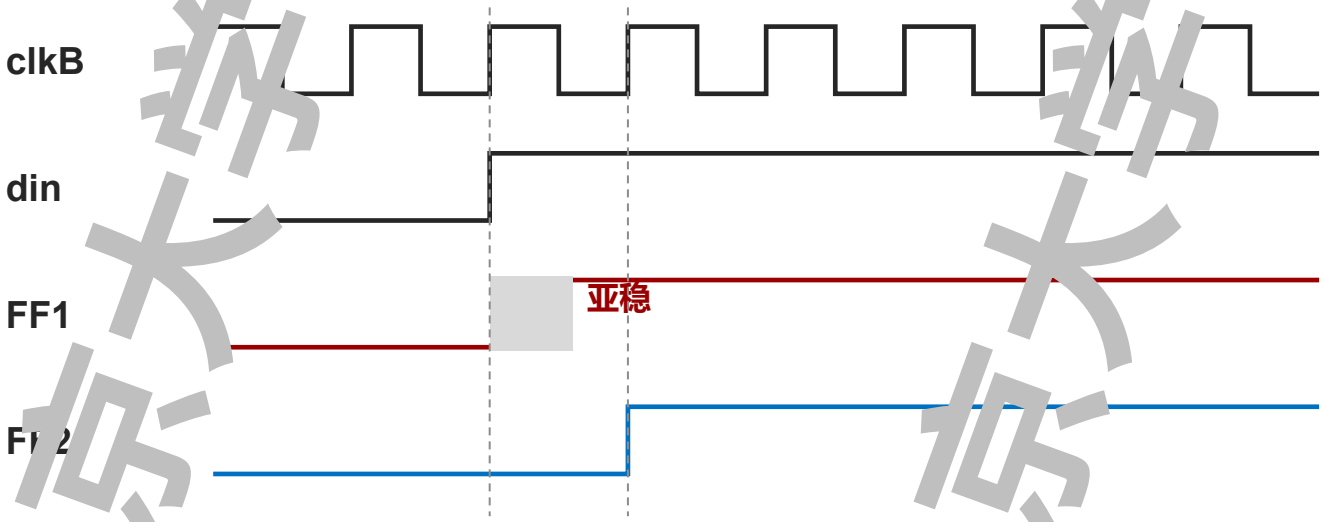
3 数据丢失

- 快时钟域的单个脉冲
- 慢时钟域可能根本采不到

双触发器同步器

- 第一级可能进入亚稳态，**留一个时钟周期**让它稳定，第二级输出才交给后级使用





使用规则

- 只用于**单比特**、变化缓慢的信号
- 源端信号先寄存，同步器前不能有组合逻辑
- 两级触发器紧邻摆放，最大化减小布线延迟
- 高频或高可靠场景使用三级同步
- 同步器单元打上属性，避免综合优化或重定时搬动

多比特与事件的[illegible]方法

- 多比特总线不能逐位同步：要么**只同步一个控制位**，要么改用**格雷码或 FIFO**

1 电平同步

- 2FF 直接同步慢变控制位
- 适用：使[illegible]、状态标志

2 脉冲同步（Toggle）

- 源端翻转电平 → 2FF → 目的端异或检测边沿
- 限制：两次事件间隔要足够大

3 [illegible]应答握手

- req、ack 各自同步，数据在 ack 返回前保持
- 适[illegible]速多比特数据，延迟数拍

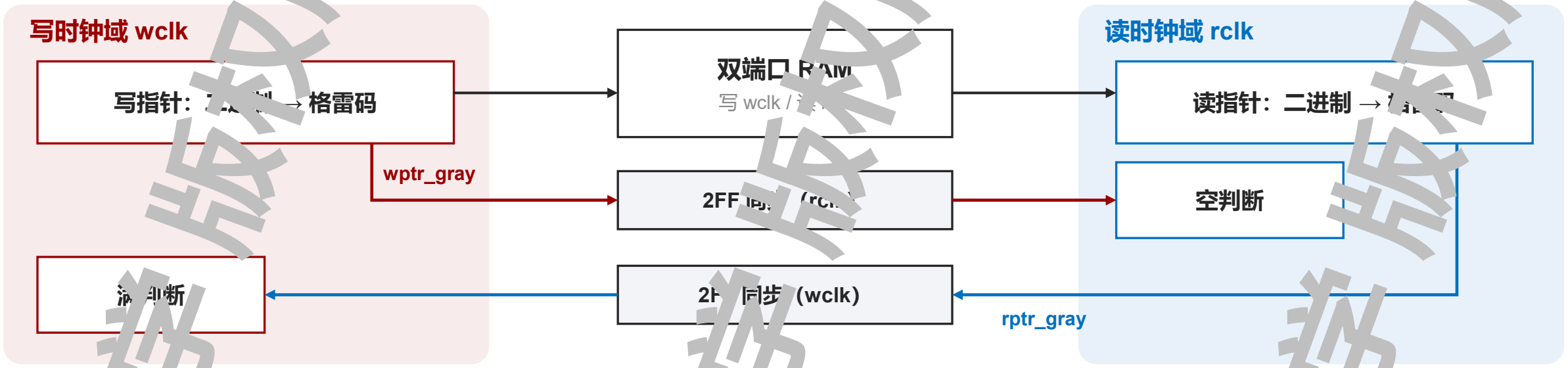
4 数据 + 使能（MCP 法）

- 数据总线保持稳定，只同步使能信号
- 目的端在使能有效后再采样数据

需要高吞吐地[illegible]数据 → 使用异步 FIFO（下一页）

异步 FIFO 与格雷码指针

- 指针以格雷码跨域：每次只变一位，同步后得到的要么是旧值、要么是新值



十进制	0	1	2	3	4	5	6	7
二进制	000	001	010	011	100	101	110	111
格雷码	000	001	011	010	110	111	101	100

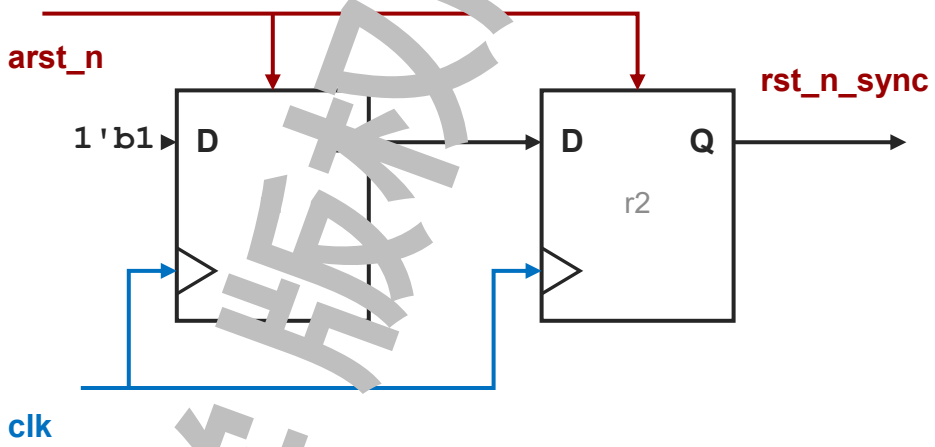
空：rptr_gray == wptr_gray_sync

满：wptr_gray == {~rptr_sync[N:N-1], rptr_sync[N−2:0]}

同步带来的延迟只会让判断偏保守，不会溢出

复位同步与复位域交叉（RDC）

- 复位异步断言、同步释放；不同复位域之间的路径同样可能引发亚稳态



reset_sync.sv

```
always_ff @(posedge clk or negedge arst_n)
  if (!arst_n) {r2, r1} <= 2'b00;  // 异步断言
  else         {r2, r1} <= {r1, 1'b1}; // 同步释放
assign rst_n_sync = r2;
```

为什么要同步释放

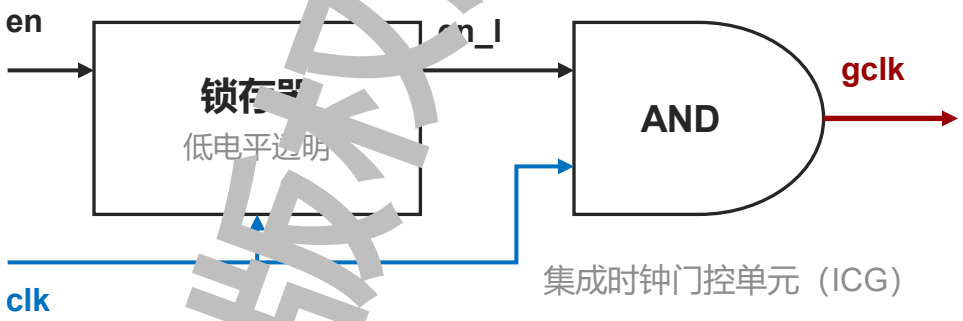
- 复位撤除若靠近时钟沿，触发器出现恢复 / 移除时间违例
- 各触发器退出复位的拍数不一致，状态机起步错拍

复位域交叉（RDC）

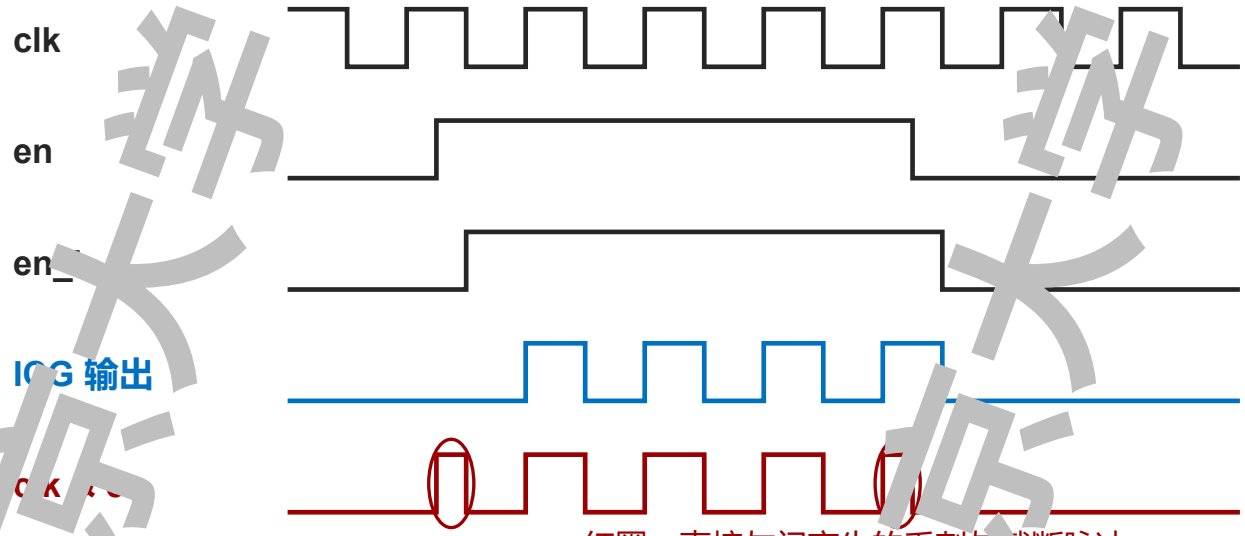
- 源寄存器被复位，而目的寄存器仍在工作
- 复位瞬间源端异步跳变 → 目的端亚稳态
- 对策：控制复位顺序、加隔离、目的端同步

时钟门控与低功耗时序设计

- 动态功耗 $P = \alpha \cdot C \cdot V^2 \cdot f$：关掉空闲寄存器的时钟，直接降低翻转率 α



集成时钟门控单元（ICG）



红圈：直接与门产生的毛刺与残断脉冲

RTL 自动门控

综合工具把带使能的寄存器组替换为 ICG

多级门控

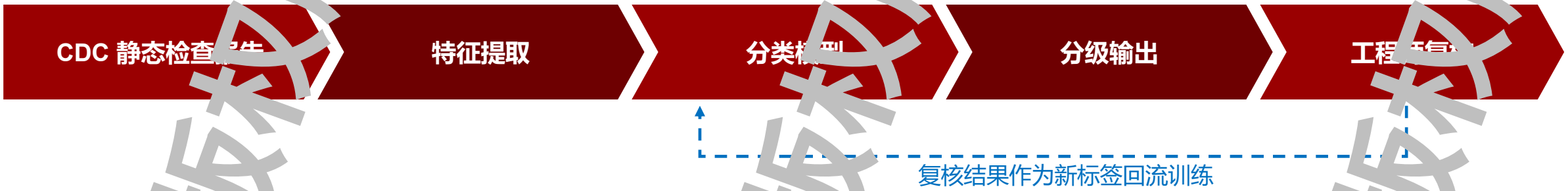
模块级 + 寄存器级，门控越靠近时钟，节省越大

时序检查

使能到 ICG 的路径同样有建立/保持约束（clock gating check）

5.9 AI 方法：CDC 违例分类与误报过滤

- 大型 SoC 的 CDC 报告常有**成千上万条**，机器学习先分级，人把精力用在真问题上



真违例
必须修复

零人工确认
附 LLM 解释

已知安全模式
建议豁免，经核后生效

可用特征

- 同步器结构类型、源 / 目的时钟关系
- 位宽、（时钟分组）、信号类别、历史豁免

LLM 的角色与底线

- 解释违例路径、建议同步方案、起草豁免理由
- 漏掉一条真违例代价极高：阈值偏向**高召回**

北京大学
PEKING UNIVERSITY

AI驱动的功能验证

测试TB、激励与UVM框架

北京大学版权所有

功能验证：芯片项目的关键路径

- 设计复杂度激增，首次流片成功率跌至二十年来最低

14%

首次流片成功率

IC/ASIC项目，2024年，二十年来最低

75%

项目进度落后于计划

历史水平约为三分之二

~30%

历史上的首次流片成功率

作为对比基线

复杂度从何而来

- **异构SoC / AI加速器**：多核、NoC、多级存储，[illegible]成倍增加
- **配置与协议空间爆炸**：参数组合呈指数增长，人工难以穷举
- **规格频繁变更**：大量重流片源于设计错误与规格变化
- **人工搭环境、写用例**：周期长，corner case容易遗漏

数据来源：Siemens EDA & Wilson Research Group, 2024 IC/ASIC Functional Verification Trend Report；SemiEngineering《First-Time Silicon Success Plummets》

思想自由 兼容并包

北京大学版权所有

验证闭环与AI切入点

- 验证是“计划—环境—激励—检查—覆盖—回归”的闭环，AI可在每一环提速



核心思想：AI不替代验证方法学，而是嵌入每个环节，缩短“发现问题 → 定位 → 修复 → 收敛”的周期

为什么需要验证方法学

- 从手写Verilog Testbench到统一标准UVM：可复用、可扩展、可互操作



Verilog
IEEE 1364：用task手写定向测试，难以复用

SystemVerilog
IEEE 1800：OOP、约束随机、断言、覆盖率

VMM
Synopsys与ARM推出的验证方法学

OVM
Cadence与Mentor联合开源方法学

UVM 1.0
Accellera发布，统一三大EDA厂商

UVM 1.2
广泛使用的Accellera版本

IEEE 1800.2
UVM成为IEEE标准

可复用
VIP / agent跨项目、跨层级复用

可扩展
factory+配置机制，不改原代码即可扩展

可互操作
统一类库，跨仿真器、跨厂商运行

思想包袱：兼容并包

UVM依赖的SystemVerilog验证特性

- UVM是建立在SystemVerilog语言特性之上的标准类库与方法学

类与面向对象

继承、多态、虚方法，让组件可扩展、可替换

```
class my_dr
  extends uvm_driver #(txn);
```

约束随机

rand / randc + constraint，由求解器产生合法激励

```
assert(tr.randomize()
       with {len < 8;});
```

interface与virtual interface

把静态的RTL信号交给动态的类对象访问

```
virtual fifo_if vif;
```

SVA断言

描述时序属性，仿真中实时检查协议

```
ass t property (@(posedge clk)
r  |-> ##[1:3] ack);
```

covergroup功能覆盖

定义功能点与交叉组合，量化验证完备度

```
cove group cg; coverpoint op;
endg oup
```

UVM = 类库 + 方法学

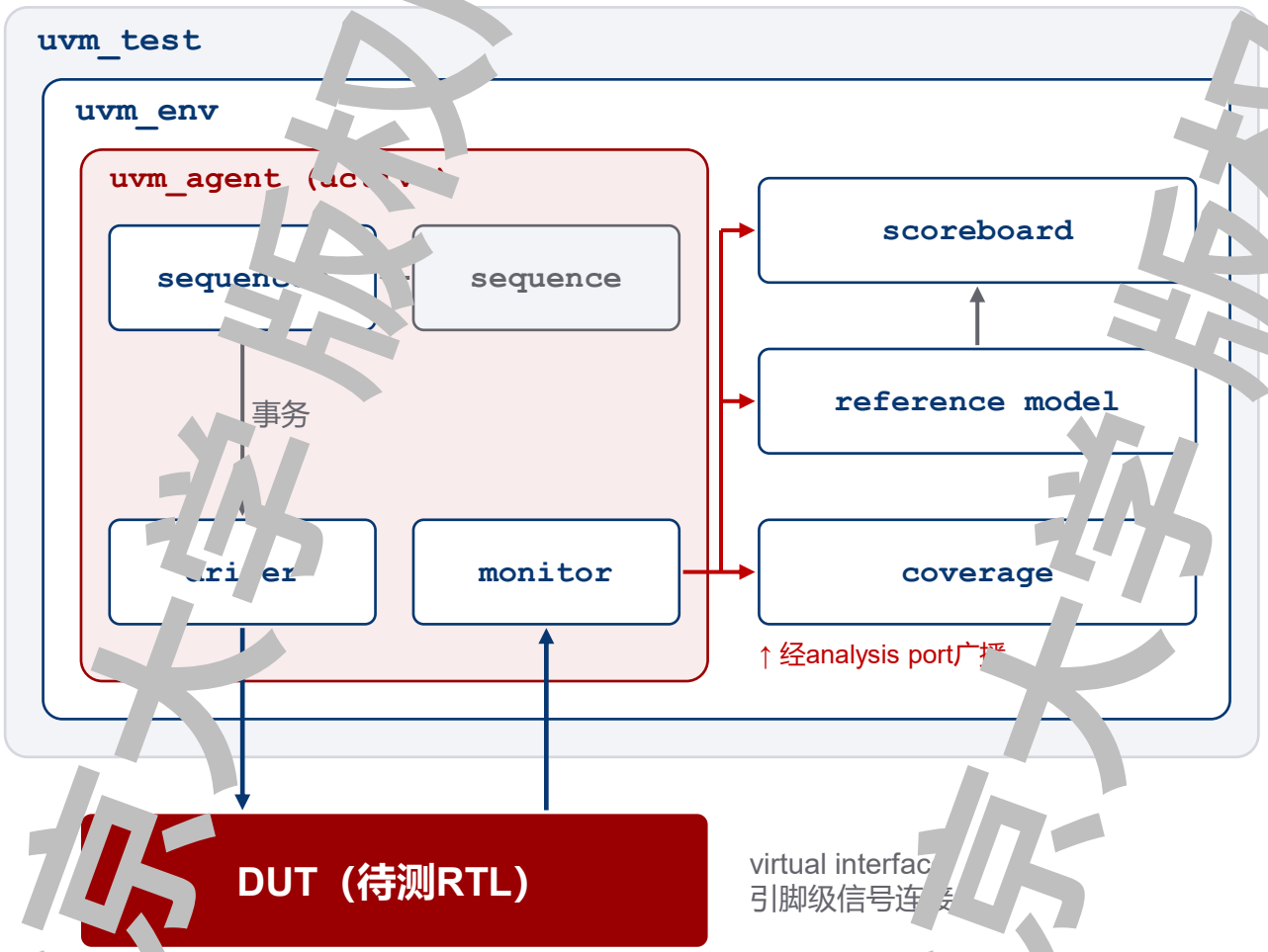
把上述语言特性封装为标准基类、宏与机制：

phase · objection · factory · config_db · TLM · RAL

思想：包容并包

UVM Testbench分层架构

- UVM以分层、可复用的组件组织验证环境，是AI生成代码的“目标骨架”



test层

选择场景：配置环境、启动sequence、设置override

env层

组装agent、scoreboard、coverage，完成连接

agent层

封装一个接口协议：sequencer + driver + monitor

信号层

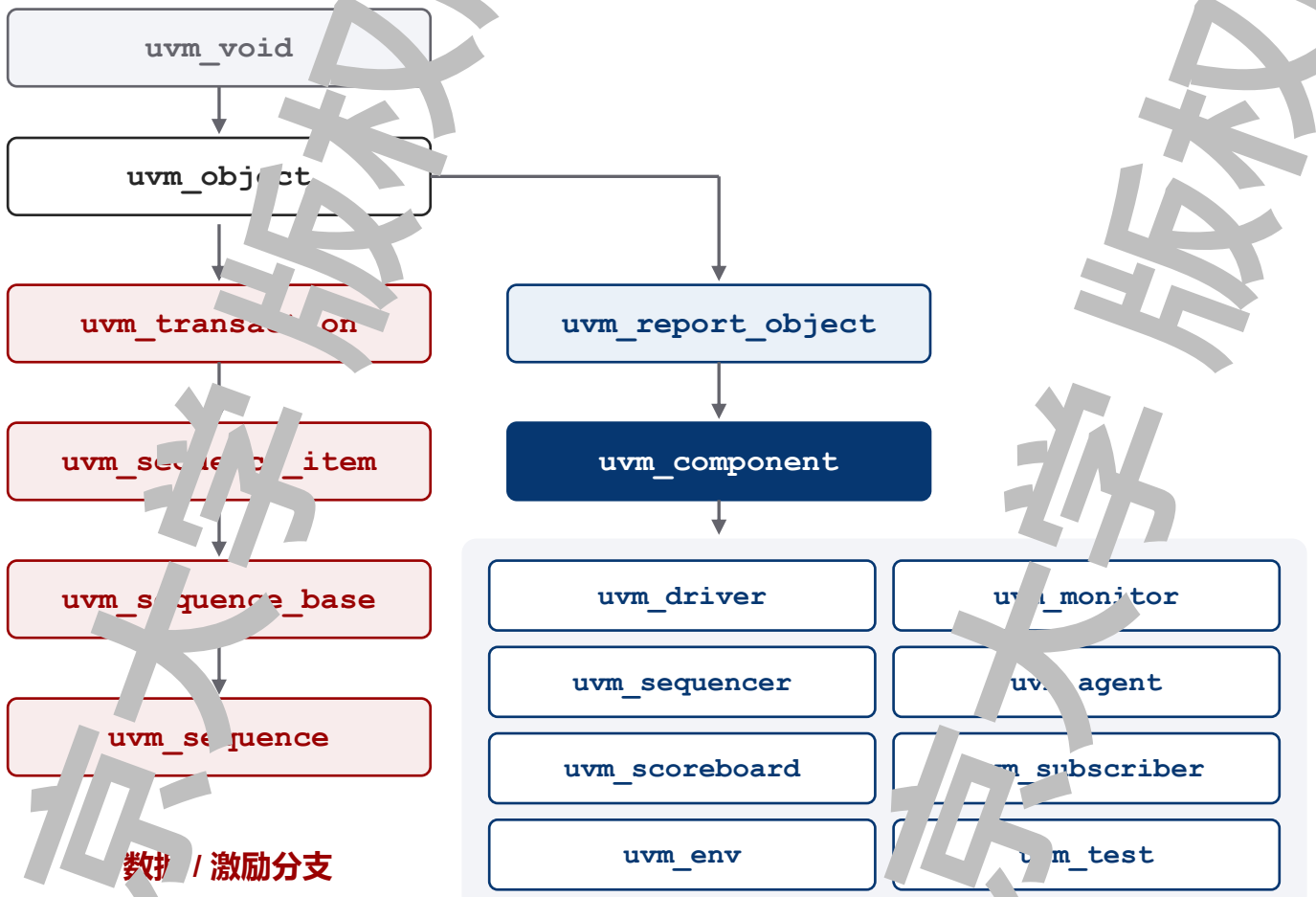
driver / monitor经virtual interface与DUT交互

思想包袱 兼容并包

北京大学版权所有

UVM类库层次：object与component

- 数据与激励派生自uvm_object，验证平台结构派生自uvm_component



维度	uvm_object	uvm_component
构造函数	new(name)	new(name, parent)
注册宏	`uvm_object_utils	`uvm_component_utils
层次结构	无父节点，可自由传递	有parent，构成组件树
生命周期	随时创建、随时丢弃	build时创建，贯穿仿真
phase	不参与	参与全部phase
典型	事务、sequence、配置对象	driver、env、test

思想包袱 兼容并包

UVM核心组件职责一览

- 每类组件职责单一、接口固定：这是可复用，也是可自动生成的基础

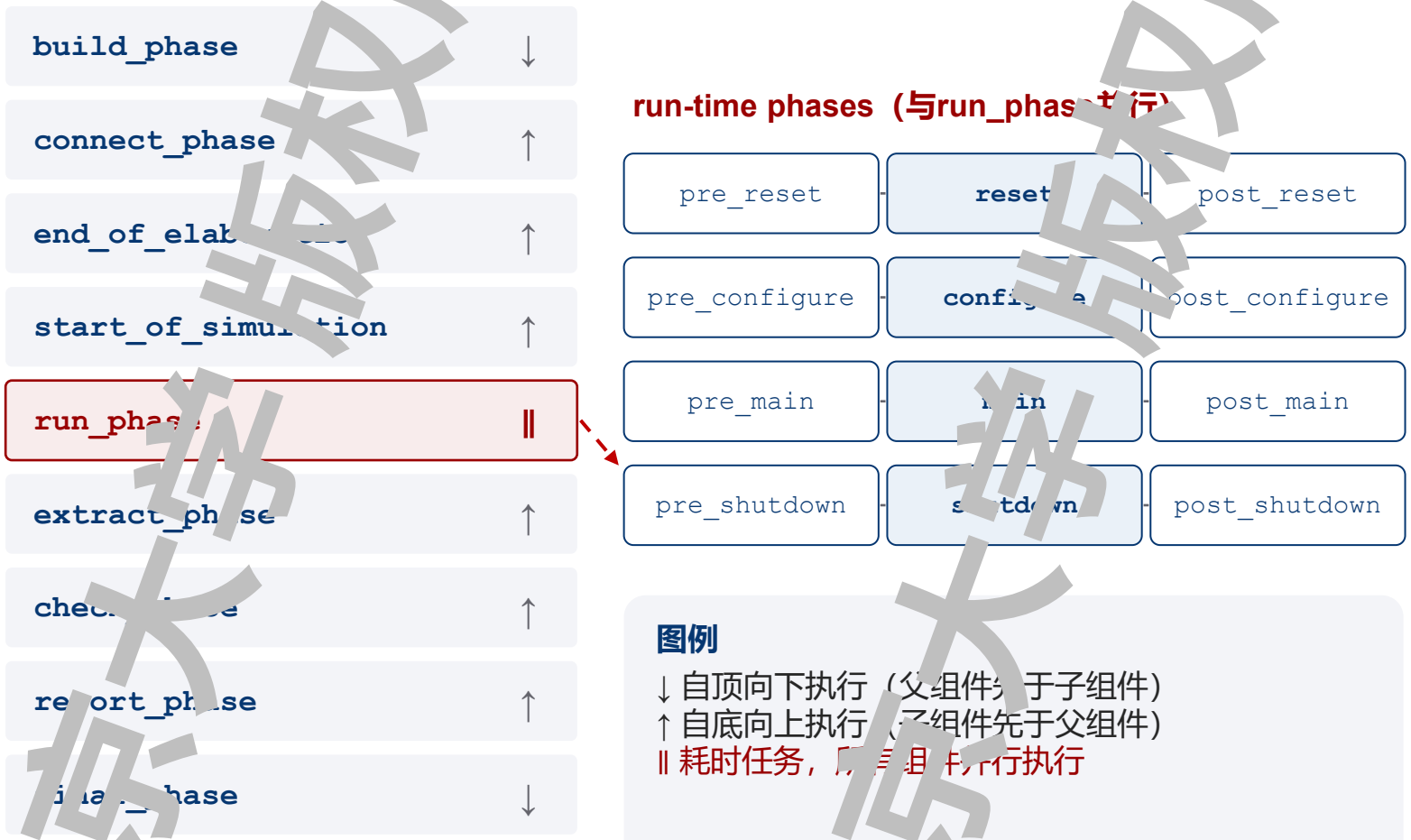
组件	职责	关键方法 / 端口
uvm_sequence_it	事务对象：一次传输的数据与控制字段	do_copy / do_compare / convertz ng
uvm_sequence	产生事务序列，描述一个测试场景	body()、start_item / finish_i e
uvm_sequencer	仲裁多个sequence，把事务转交driver	seq_item_export
uvm_driver	事务 → 引脚级时序，驱动DUT	seq_item_port.get_next_item / item_done
uvm_mon or	采样引脚 → 还原为事务并广播	uvm_analysis_port::writ
uvm_agen	封装一个接口：active含driver，passiv 只监测	is_active (UVM_ACTIVE / UVM_PASSIVE)
uvm_ co eb d	与参考模型比对，判定正确性	uvm_analysis_imp + w it ()
uvm_s osc iber	订阅事务，采样功能覆盖率	write()中调用covergrou .sa le()
uv _e v	组装agent / scoreboard / c v rage	build_phase / con c_ h e
uvm_ est	选择配置与sequence，启动 景	run_test()、+UVM_TE NAM

思想包袱 兼容并包

北京大学版权所有

Phase机制详解

- 统一的执行时序：构建 → 连接 → 运行 → 收尾，所有组件步调一致



图例

↓ 自顶向下执行（父组件先于子组件）
↑ 自底向上执行（子组件先于父组件）
‖ 耗时任务，[illegible]并行执行

执行顺序

build_phase自顶向下：父组件build时创建子组件；connect等其余函数ph[illegible]自底向上

run_phase并行

所有组件的run_phase[illegible]的启动；每个run-time phase须全部组[illegible]进入下一个

使用建议

多数项目只用run_phas[illegible] + objection；子phase适合复[illegible]、重配置等多阶段场景

Objection机制与仿真结束

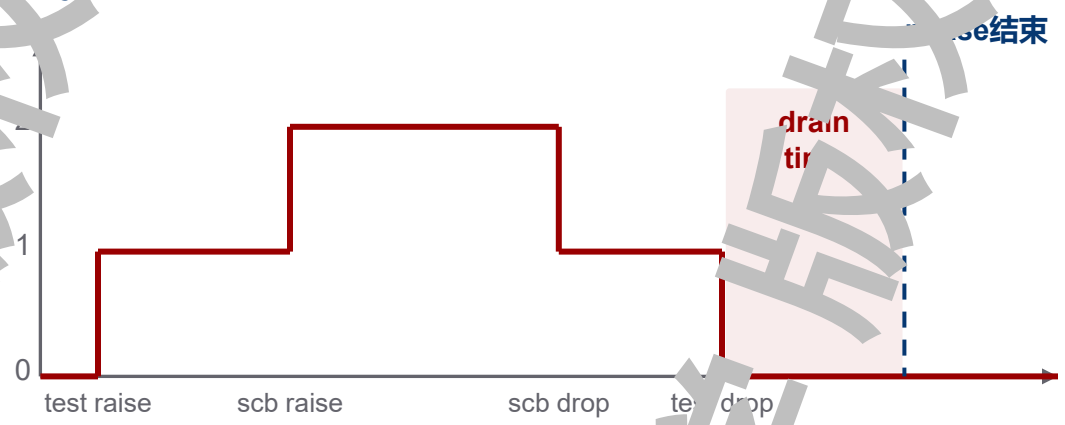
- run_phase何时结束？由各组件“举手反对”的计数决定

raise / drop / drain time

```
class base_test extends uvm_test;
  `uvm_component_utils(base_test)
  fifo_env env;
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  task run_phase(uvm_phase phase);
    fifo_seq seq = fifo_seq::type_id::create("seq");
    phase.raise_objection(this, "main seq start");
    seq.start(env.agt.sqr);
    // 计数归零后再等 200ns，让流水线排空
    phase.get_objection().set_drain_time(this, 200ns);
    phase.drop_objection(this, "main seq done");
  endtask
endclass
```

objection计数随时间变化（示意）



drain time	计数归零后延时结束，等待最后的响应被比对
忘记raise	run_phase立即结束，仿真在0时刻退出
忘记drop	仿真挂起直到全局超时（set_timeout）

思想自由 兼容并包

事务与约束随机

- rand / randc、约束、分布与优先级：定义激励的合法空间与概率

SystemVerilog / UVM

```
class bus_txn extends uvm_sequence_item;
  rand  bit [31:0]  addr;
  rand  bit [31:0]  data;
  rand  bit         wr;
  rand  int unsigned len;
  randc bit [3:0]   id;       // 循环随机：16 个值遍历完才重复
  constraint c_addr { addr[1:0] == 2'b00;
                      addr inside {[32'h0:32'hFFFF]}; }
  constraint c_len  { len dist {1 := 50, [2:8] :/ 40,
                                [9:16] :/ 10}; }
  constraint c_ord  { solve wr before len;
                      wr -> len <= 8; }
  constraint c_dflt { soft data == 32'h0; }
  `uvm_object_utils(bus_txn)
  function new(string name = "bus_txn");
    super.new(name);
  endfunction
endclass
```

randc循环随机

取值不重复，直到遍历完全部取值后ID、通道号

:= 与 :/ 的区别

:= 给区间内每个值同样权重；:/ 把权重平分给整个区间

solve … before

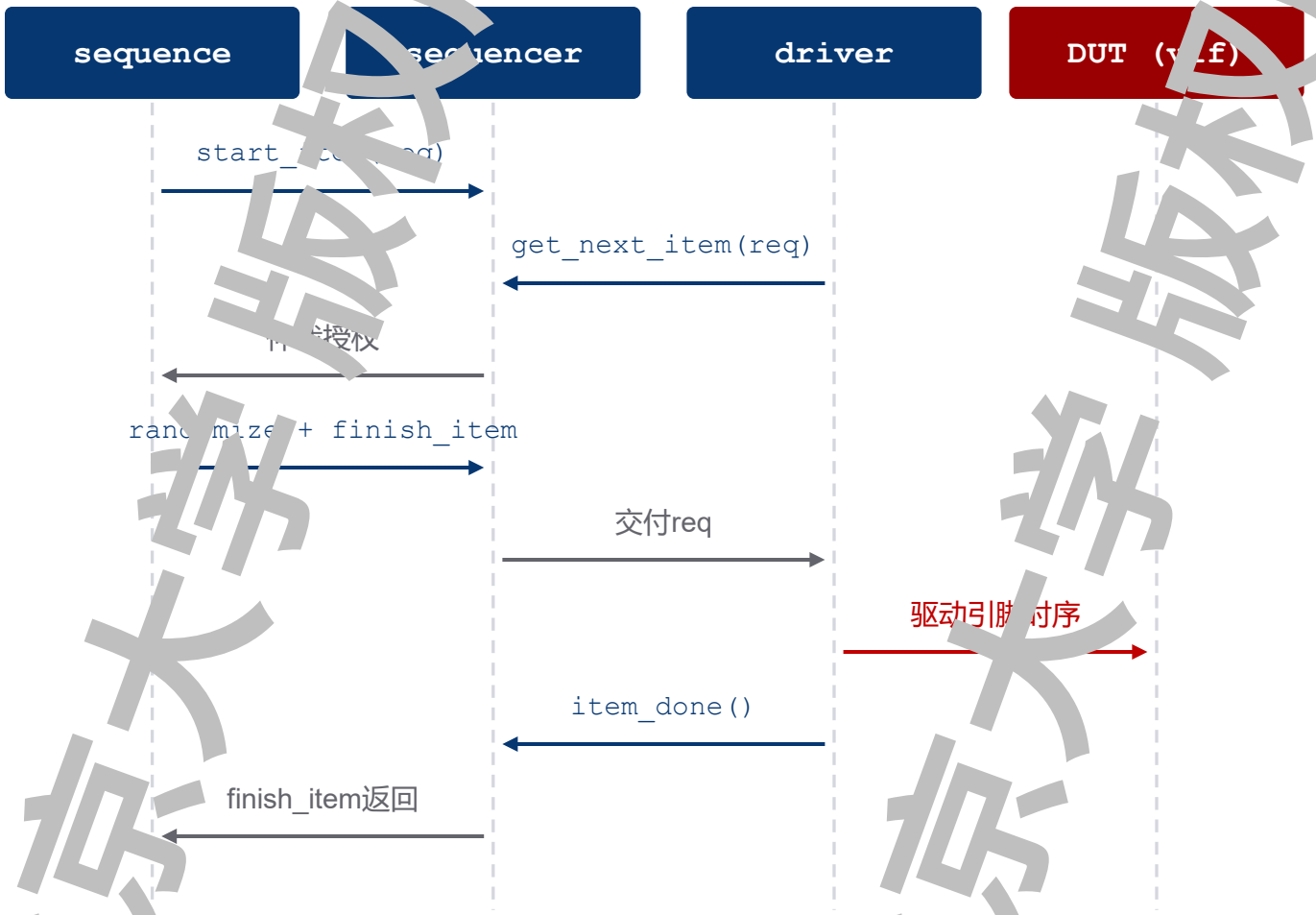
指定求解顺序，改变分布但不改变合法解空间

soft软约束

提供默认值，可被randomize() with {...} 覆盖

Sequence–Sequencer–Driver握手

- start_item / finish_item与get_next_item / item_done成对出现



成对调用

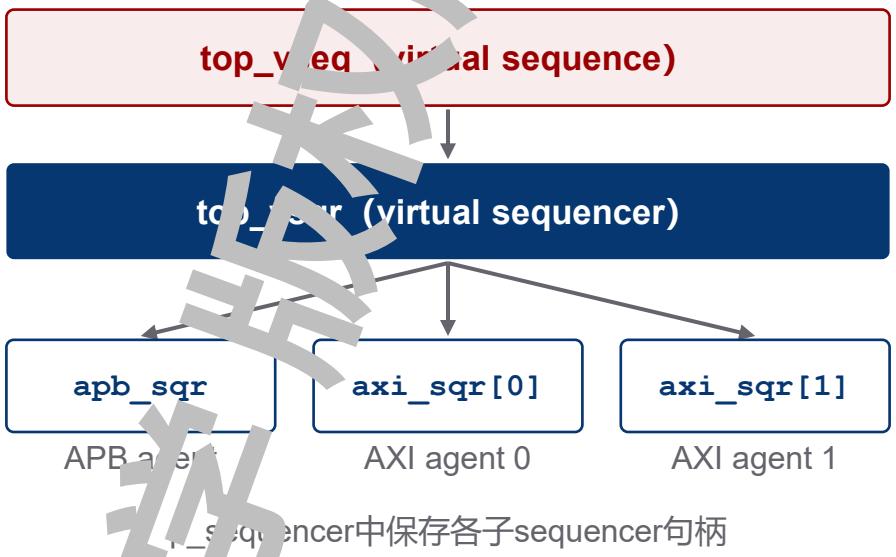
```
// driver: 拉取模式
task run_phase(uvm_phase phase);
  forever begin
    seq_item_port.get_next_item(req);
    drive_item(req);
    seq_item_port.item_done();
  end
endtask

// sequence: 产生事务
task body();
  repeat (10) begin
    req = bus_txn::type_id::create("req");
    start_item(req);
    assert(req.randomize());
    finish_item(req);
  end
endtask
```

兼容并包

Sequence进阶：嵌套、库与Virtual Sequence

- virtual sequence不产生事务，只负责调度多个接口的sequencer



p_sequencer中保存各子sequencer句柄

virtual sequence示例

```
class top_vseq extends uvm_sequence;
  `uvm_object_utils(top_vseq)
  `uvm_declare_p_sequencer(top_vsqr)
  function new(string name = "top_vseq"); super.new(name); endfunction
  task body();
    cfg_seq_c  = cfg_seq::type_id::create("c");
    traffic_seq t0 = traffic_seq::type_id::create("t0");
    traffic_seq t1 = traffic_seq::type_id::create("t1");
    c.start(p_sequencer.apb_sqr);          // 1. 先配寄存器
    fork                                   // 2. 两路并发流量
      t0.start(p_sequencer.axi_sqr[0]);
      t1.start(p_sequencer.axi_sqr[1]);
    join
  endtask
endclass
```

嵌套sequence

在body()中start子sequence，像搭积木一样组合复杂场景

sequence library

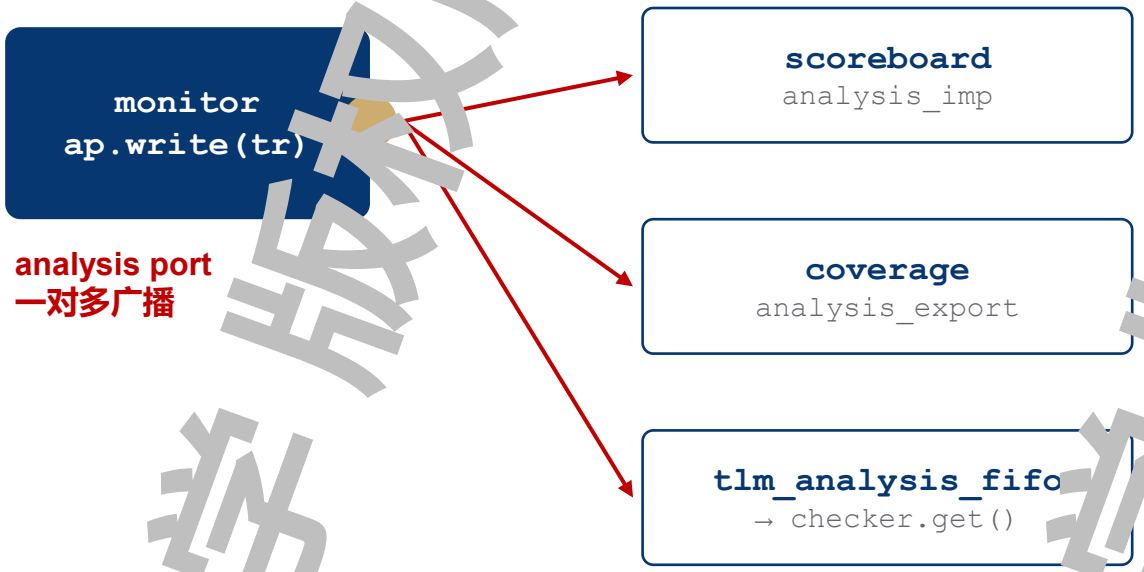
uvm_sequence_library按随机 / 循环模式挑选已注册的sequence

virtual sequence

跨接口协调时序：先配置、再并发、最后检查

TLM通信：port、export、imp与analysis port

- 组件之间只通过事务级端口通信，彼此解耦、可独立替换



analysis port
一对多广播

port	发起方：调用put / get / write
export	中转：把请求转发到内部组件
imp	实现方：真正实现write()等方法

analysis连接示例

```
// monitor
  analysis_port #(bus_txn) ap;
ap = new("ap", this);              // build_phase
ap.write(tr);                      // 采到一笔事务即广播

// scoreboard
  analysis_imp #(bus_txn, fifo_scb) imp;
function void write(bus_txn t);
  // 比对逻辑
endfunction

// env.connect_phase: port → imp/export
agt.mon.ap.connect(scb.imp);
agt.mon.ap.connect(cov.analysis_export);
agt.mon.ap.connect(fifo.analysis_export);
```

思想包袱 兼容并包

Factory机制：注册 创建与覆盖

- 用type_id::create代替new，测试层即可替换任何组件或事务类型

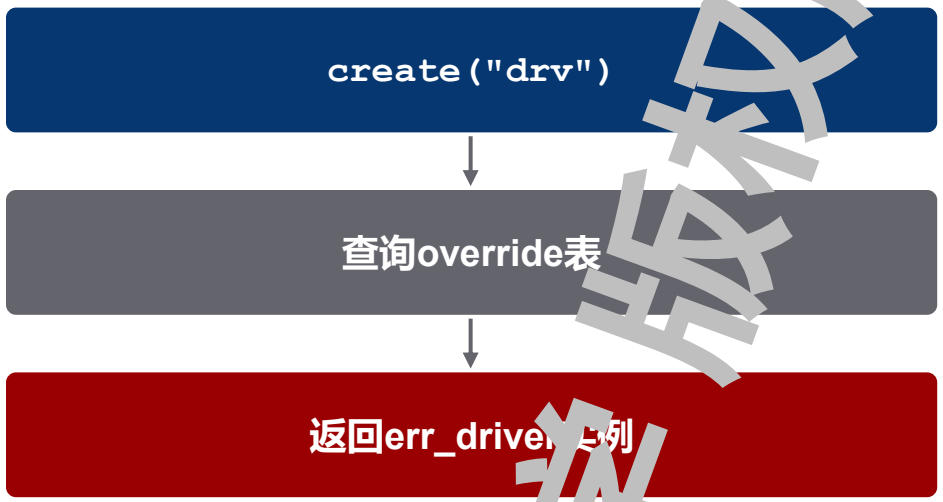
SystemVerilog / UVM

```
class my_driver extends uvm_driver #(bus_txn);
  `uvm_component_utils(my_driver)          // 注册到 factory
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction
endclass

class err_driver extends my_driver;        // 注错版本
  `uvm_component_utils(err_driver)
  // new() 同上; 重写 drive_item() 注入错误
endclass

// agent.build_phase: 通过 factory 创建
drv = my_driver::type_id::create("drv", this);

// env.build_phase: 不改 env 代码即可替换
set_type_override_by_type(my_driver::get_type(),
                          err_driver::get_type());
set_inst_override_by_type("env.agt1.drv",
  my_driver::get_type(), err_driver::get_type());
```



type override

全局替换该类型的所有实例

inst override

替换指定层次路径的实例，优先级更高

思想：包容 兼容并包

uvm_config_db：层次化配置机制

- 按“路径 + 名字 + 类型”存取配置，最典型的用途是传递virtual interface

set / get

```
// tb_top (module)：把接口句柄放入配置库
initial begin
  uvm_config_db#(virtual fifo_if)::set(
      null, "uvm_test_top.env.agt*", "vif", fifo_if0);
  run_test();
end

// fifo_driver.build_phase：取出接口
if (!uvm_config_db#(virtual fifo_if)::get(
      this, "", "vif", vif))
  `uvm_fatal("NOVIF", "virtual interface not set")

// base_test.build_phase：把 agt1 配成 passive
uvm_config_db#(uvm_active_passive_enum)::set(
    this, "env.agt1", "is_active", UVM_PASSIVE);
```

作用域

inst路径相对context组件，支持 * 通配；get时按自身路径匹配

优先级

build期间层次更高的set优先；同一层次后set覆盖先set

类型须一致

set与get的参数化类型必须完全相同，否则get返回0

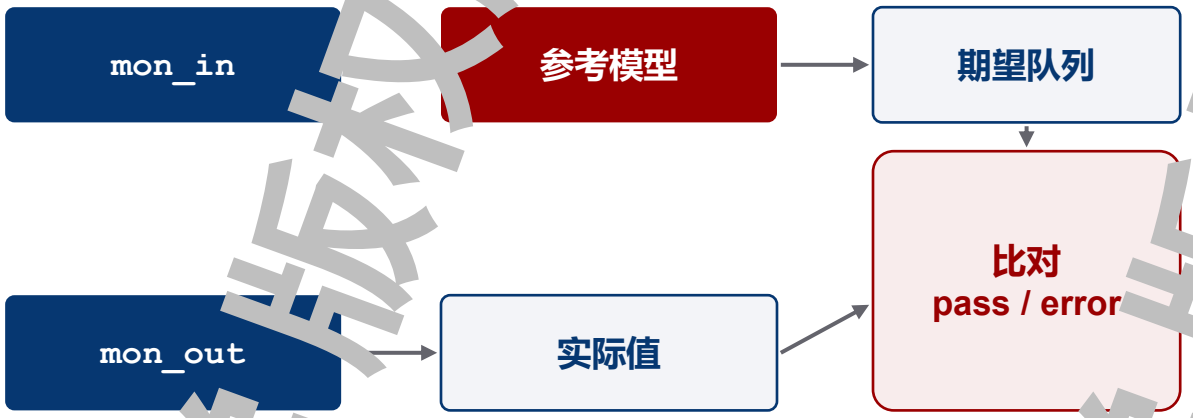
经验：get失败一律 `uvm_fatal，避免空句柄在仿真中途才报错

思想包袱 兼容并包

北京大学版权所有

Scoreboard与参考模型

- 预测值与实际值逐笔比对：参考模型可用SV、C/C++（DPI-C）或Python实现



in-order比对

期望值进入FIFO队列，实际值到达时逐笔弹出比对；适合保序接口

out-of-order比对

以[illegible]地址为键存入关联数组，乱序返回时按键匹配；适合A[illegible]乱序协[illegible]

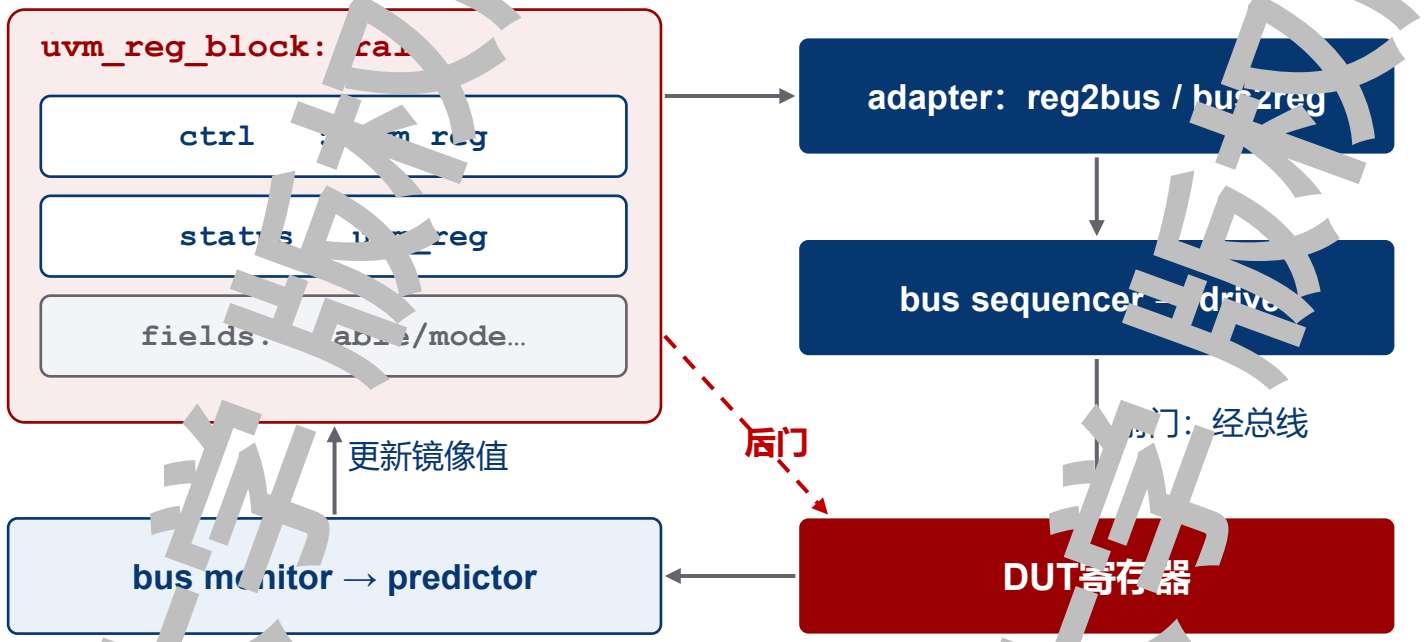
in-order scoreboard + DPI-C参考模型

```
[illegible]analysis_imp_decl(_in)
[illegible]analysis_imp_decl(_out)
[illegible]t "DPI-C" function int unsigned
 c_ref_alu(int unsigned a, int unsigned b, i[illegible]t [illegible]);

class alu_scb extends uvm_scoreboard;
  uvm_analysis_imp_in  #(alu_txn, alu_scb) i[illegible]mp;
  uvm_analysis_imp_out #(alu_txn, alu_scb) ou[illegible]mp;
  int unsigned exp_q[$];
  // ... `uvm_component_utils、new、build 略
  function void write_in(alu_txn t);
    exp_q.push_back(c_ref_alu(t.a, t.b, t.o[illegible]));
  endfunction
  function void write_out(alu_txn t);
    int unsigned expv = exp_q.pop_front();
    if (t.res !== expv)
      `uvm_error("SCB", $sformatf("e[illegible]=0h [illegible]=%0h",
                                  exp[illegible]t.res))
  endfunction
endclass
```

寄存器模型RAL

- 用抽象的寄存器读写代替总线细节：同一份测试可跨总线协议复用



寄存器sequence片段

```
uvm_status_e   status;
uvm_reg_data_t val;

// 前门：经总线写 / 读
ral.ctrl.enable.set(1);
ral.ctrl.update(status);
ral.status.read(status, val);

// 后门：直接修改 RTL 寄存器
ral.ctrl.poke(status, 32'h1);

// 读回并与镜像值比对
ral.status.mirror(status, UVM_CHECK);
```

前门访问frontdoor

write / read经总线完成，验证真实访问路径

后门访问backdoor

peek / poke经HDL路径直接读写，零时间、快速初始化

Testbench目录结构与运行

- 一次编译、多次测试：用命令行参数选择测试、打印级别与随机种子

典型目录

```
tb/
├── tb_top.sv          // DUT + interface + run_test()
├── fifo_if.sv
├── fifo_pkg.sv        // `include 全部 class
├── agent/
│   ├── fifo_xtn.sv        fifo_seq_lib.sv
│   ├── fifo_driver.sv     fifo_monitor.sv
│   └── fifo_agent.sv
├── env/
│   ├── fifo_scb.sv        fifo_cov.sv
│   └── fifo_env.sv
└── test/
    ├── base_test.sv       smoke_test.sv
    └── stress_test.sv
```

编译与运行

```
# Synopsys VCS
vcs -sverilog -ntb_opts uvm-1.2 \
    -timescale=1ns/1ps -f filelist.f -l comp.log
./simv +UVM_TESTNAME=smoke_test \
       +UVM_VERBOSITY=UVM_MEDIUM \
       +ntb_random_seed=1234 -l sim.log

# Cadence Xcelium
xrun -uvm -f filelist.f \
     +UVM_TESTNAME=stress_test -svseed random
```

+UVM_TESTNAME 同一份编译结果运行不同测试

+UVM_VERBOSITY 控制打印级别：LOW / MEDIUM / HIGH / DEBUG

随机种子seed 记录种子可精确复现失败用例

思想自由 兼容并包

小结：UVM的规范性让AI生成成为可能

- 固定的结构、接口与报告格式，正是LLM擅长“照模板填空”的场景

UVM机制	对AI生成的意义
固定的类结构与宏	可模板化生成，生成后易于静态检查
标准phase与TLM端口	组件可独立生成，按约定自动连接
factory与config_db	生成代码以override方式注入，不改动已有环境
统一的日志与报告格式	编译 / 仿真错误可被解析，反馈给LLM修复

下一部分

AI辅助Testbench生成

模板负责结构，LLM负责行为，EDA工具负责把关

北京大学
PEKING UNIVERSITY

AI驱动的功能验证

AI辅助Testbench生成

北京大学版权所有

为什么用LLM生成Testbench

- Testbench代码量大、模式化强——正是LLM擅长的“结构化生成”

人工搭建的痛点

样板代码多

每个agent都要interface、driver、monitor等多个文件

协议时序多

握手、背压、时序对齐，稍有疏忽即测错或漏测

搭建周期长

新IP往往要先花大量时间搭环境，才能开始真正的验证

依赖个人经验

风格不统一、新人上手慢，知识难以沉淀复用

LLM擅长

- 按模板生成样板代码
- 把规格描述翻译成代码
- 解释编译错误与仿真日志
- 起草断言与covergroup
- 补全注释与文档

LLM不擅长

- 精确的周期级时序推理
- 跨多个文件保持一致
- 规格中未写明的隐含约定
- 保证功能正确——会“自信地出错”
- 超长上下文中的细节

合理分工

LLM写初稿 → EDA工具验证 → 工程师审查签核

北京大学版权所有

LLM在硬件验证[illegible]的能力谱

- 从“补全一行代码”到“自主完成验证闭环”，自主性与集成度逐级提升



L1
代码补全
行级、片段级补全，
IDE内即时提示

已普遍可用

L2
组件生成
按规格生成UVM组件

可用，需审查

L3
断言与覆盖
生成SVA与covergroup
草稿

可用，需审查

L4
调试辅助
解析日志、定位错误、
给出修复

研究与试点

L5
验证智能体
自主规划、生成、仿真
与迭代

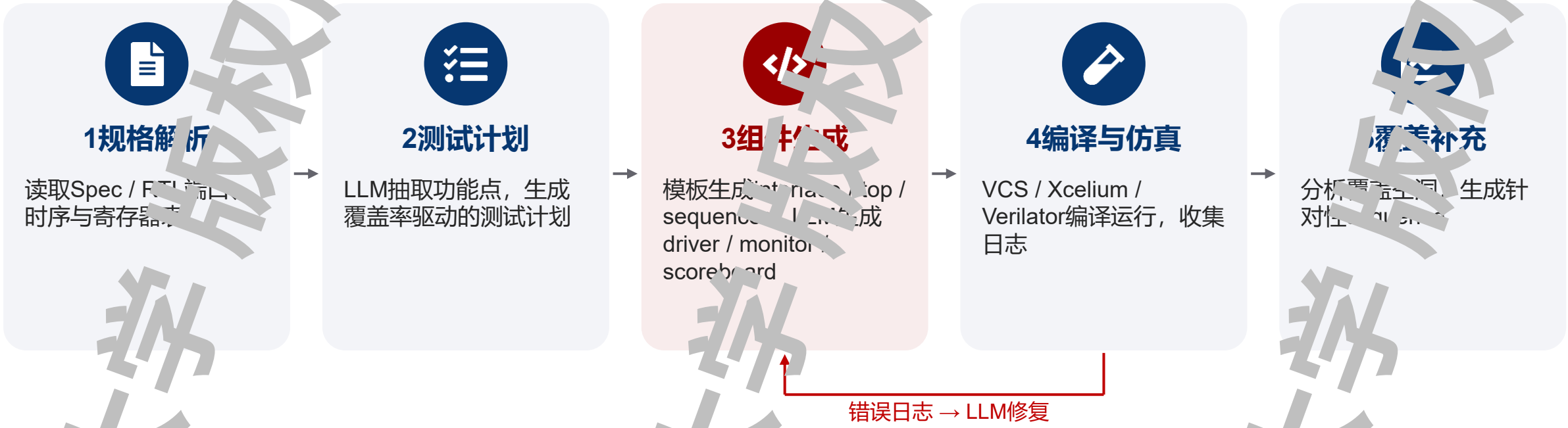
前沿探索

思想包容 兼容并包

北京大学 版权所有

LLM生成Testbench的整体流程

- 从规格到可运行的UVM环境：模板保证结构，LLM补全行为



代表性工作：UVM²（2025）

1个RTL设计（加密核、ALU、SPI、UART）上的实验结果

87.44%
平均代码覆盖率

89.58%
平均功能覆盖率

≈15.8×
环境搭建平均提速

来源：J. Ye et al., “From Concept to Practice: an Automated LLM-aided UVM Machine for RTL Verification”, arXiv:2504.19959；提速为相对资深工程师（约103分钟 vs 约27小时）

思想自由 兼容并包

输入准备：把规格结构化

- 自然语言规格先转成机器可读的JSON / YAML，再交给模板引擎与LLM

原始规格（自然语言）

sync_fifo为深度16、位宽32的同步FIFO。wr_en有效且未满时在时钟上升沿写入din；rd_en有效且非空时，下一拍dout输出数据。rst_n低有效异步复位，复位后empty=1。

为什么要结构化

- 消除自然语言歧义，缺项一目了然
- 模板引擎与LLM共用同一份“事实”
- 可与RTL端口自动比对，提前发现不一致

LLM抽取 + 人工确认的结构化规格（JSON）

```
{
  "module": "sync_fifo",
  "params": { "DEPTH": 16, "WIDTH": 32 },
  "clock": { "name": "clk", "edge": "posedge" },
  "reset": { "name": "rst_n", "active": "low", "async": true },
  "ports": [
    { "name": "wr_en", "dir": "in",  "width": 1  },
    { "name": "din",   "dir": "in",  "width": 32 },
    { "name": "rd_en", "dir": "in",  "width": 1  },
    { "name": "dout",  "dir": "out", "width": 32 },
    { "name": "full",  "dir": "out", "width": 1  },
    { "name": "empty", "dir": "out", "width": 1  }
  ],
  "timing": [ "rd_en & !empty -> dout valid at +1 cycle" ],
  "reset_values": { "empty": 1, "full": 0 }
}
```

测试计划自动生成

- LLM从规格抽取功能点，生成“功能点 → 场景 → 覆盖点 → 检查”的映射

结构化规格 → LLM抽取功能点 → 工程师评审确认 → 验证计划（Plan）

功能点	测试场景	覆盖点	检查方式
F1复位行为	复位期间及释放后的读写	cp_reset × cp_op	SVA：a_reset
F2基本读写	随机长度的读写序列	cp_op	scoreboard比对
F3满状态	写满后继续写入	cp_level.full	SVA + scoreboard
F4空状态	空时发起读操作	cp_level.empty × rd	SVA：empty保持
F5同时读写	各水位下同时读写	cross cp_op × cp_level	scoreboard比对
[illegible]	连续多拍读写不间断	transition bins	scoreboard比对

LLM只产出初稿：遗漏的功能点、错误的优先级都要由验证工程师在评审中纠正，确认后的Plan才进入下游生成。

思想包袱 兼容并包

北京大学出版社

Prompt与上下文构建

- 生成质量取决于“喂给模型什么”：结构化规格 + 检索增强 + 编码规范

结构化设计规格

端口列表、时序关系、协议、寄存器表——尽量表格化

UVM模板库 / VIP示例

RAG检索相似agent作为参考，减少“自由发挥”

团队编码规范

命名约定、宏用法、目录结构，保证可维护

历史错误与修复记录

作为few-shot示例，规避常见编译错误

Prompt示例

```
[角色] 你是资深 UVM 验证工程师
[背景] 模块: sync_fifo  DEPTH=16  WIDTH=32
       端口: clk, rst_n, wr_en, rd_en,
             din[31:0], dout[31:0], full, empty
       时序: rd_en 有效后下一拍 dout 有效
[任务] 生成 fifo_driver 与 fifo_monitor
[约束] 1. 遵循附带模板的类结构与宏
       2. 所有信号经 virtual interface 访问
       3. 只输出 SystemVerilog 代码
[参考] <RAG 检索到的 APB agent 示例>
```

经验：规格越结构化、约束越明确，生成代码的一次编译通过率越高

RAG知识库：让模型“照着样例写”

- 离线把模板、VIP与历史Testbench建成索引，在线按任务检索最相关的参考

离线：构建知识库



在线：检索增强生成



按语法单元切块

不要按固定字数切，避免把一个class切成两半

元数据过滤

协议类型、UVM版本不匹配的样例没有样例更糟

保留来源引用

生成代码标注参考出处，便于审查与追溯

思想包容 兼容并包

北京大学版权所有

模板 + LLM混合生成

- 结构固定的交给模板，行为相关的交给LLM，并按依赖顺序逐层生成

组件	生成方式	原因
interface / top / package	模板	结构固定，只依赖端口列表
sequencer / [illegible] / env	模板	连接关系规整，可参数化
sequence_item	模板 + LLM	字段来自规格，约束需理解语义
driver / monitor	LLM	协议时序行为，需读懂规格
scoreboard / 参考模型	LLM + 人工	功能语义，正确性最关键
sequence / test	LLM	测试场景由意图激励

被依赖者先生成并编译通过，再把其代码作为上下文生成依赖者，减少接口不一致。生成顺序参考UVM²（arXiv:2504.19959）。

依赖驱动的生成顺序

① **公共基础**
package · interface · transaction

② **叶子组件**
driver · monitor · sequencer

③ **组合组件**
agent（组装叶子组件）

④ **环境级**
scoreboard · env · test

思想自由 兼容并包

北京大学版权所有

示例：LLM生成的Driver

- 把“协议怎么驱动”写成时钟沿上的非阻塞赋值，并正确处理空闲周期

```
class fifo_drive  xtends uvm_driver #(fifo_txn);
  `uvm_component_uti s(fifo_driver)
  virtual fifo_if vif
  // new() 略
  function void bu ld_phase(uvm_phase phase);
    if (!uvm_co fig_db#(virtual fifo_if)::get(this, "", "vi ", vif))
      `uvm_f  ("NOVIF", "virtual interface not set")
  endfuncti
  task run_p ase(u m_phase phase);
    wait (vif.rs  n === 1'b1);
    forever begin
      @(p sedge vif.clk);
      se _ m_port.try_next_item(req);
        re  = null) begin                  // 无事务：驱 空闲
        if wr_en <= 1'b0;  vif.rd_en <= 1'b0;
      e d else begin
        vif r_en <= req.op inside {WRITE, WRITE_READ};
        vif.rd_en <= req.op inside {READ,  WRITE_REA
        if.din   <= req.data;
        s q_item_port.item_done();
      end
    e d
  nd sk
en as
```

1. **config_db获取vif**
   接口由top注入，driver不直接引用层次路径
2. **try_next_item处理空闲**
   无事务时拉低使能，有事务时与DUT背靠背驱动
3. **时钟沿 + 非阻塞赋值**
   在posedge用 <= 驱动，避免与DUT采样竞争

LLM常见错误（审查清单）

- 阻塞赋值 = 或在错误时钟沿驱动
- 漏写item_done导致sequence挂死

示例：Monitor与Scoreboard

- monitor只采样不驱动；scoreboard用队列实现FIFO的参考行为

fifo_monitor.sv（new/build_phase略）

```
class fifo_monitor extends uvm_monitor;
  `uvm_component_utils(fifo_monitor)
  virtual fifo_if vif;
  uvm_analysis_port #(fifo_txn) ap;
  bit rd_q;                          // 上一拍是否发生有效读
  task run_phase(uvm_phase phase);
    forever begin
      @(posedge vif.clk);
      if (rd_q) publish(READ, vif.dout);  // 读数据晚一拍
      if (vif.rst_n && vif.wr_en && !vif.full)
        publish(WRITE, vif.din);
      rd_q = vif.rst_n && vif.rd_en && !vif.empty;
    end
  endtask
  function void publish(fifo_op_e op, bit [31:0] d);
    fifo_txn t = fifo_txn::type_id::create("t");
    t.op = op;  t.data = d;
    ap.write(t);
  endfunction
endclass
```

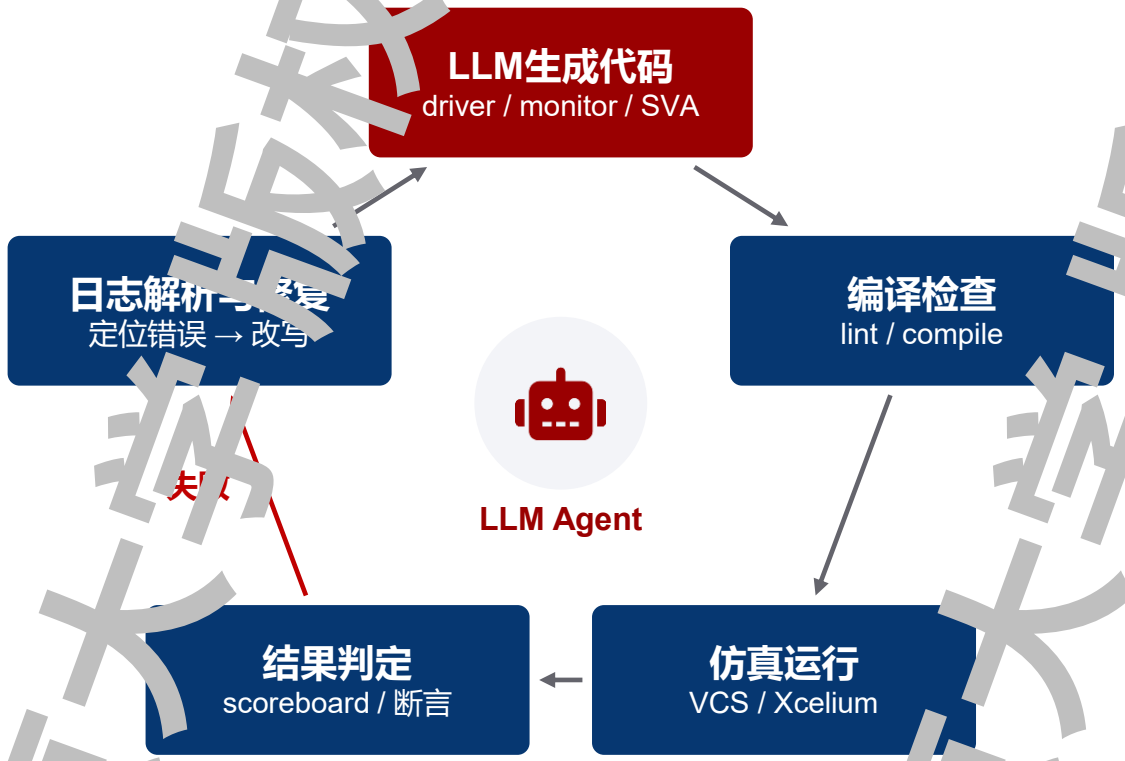
fifo_scoreboard.sv（new略）

```
class fifo_scoreboard extends uvm_scoreboard;
  `uvm_component_utils(fifo_scoreboard)
  uvm_analysis_imp #(fifo_txn, fifo_scoreboard) imp;
  bit [31:0] model_q[$];          // 参考模型：队列
  function void write(fifo_txn t);
    if (t.op == WRITE)
      model_q.push_back(t.data);
    else if (model_q.size() == 0)
      `uvm_error("SB", "read from empty model")
    else begin
      bit [31:0] exp = model_q.pop_front();
      if (t.data !== exp) `uvm_error("SB",
        $sformatf("exp %h, got %h", exp, t.data))
    end
  endfunction
endclass
```

本页要点：读数据晚一拍——LLM常在同一拍采样dout，漏掉“晚一拍”写法漏掉背靠背读

生成—编译—仿真—修复闭环

- 让EDA工具当“裁判”：编译器与仿真器的反馈驱动LLM迭代自修复



闭环的工程要点

- 限定修复轮数，每轮保留版本快照
- 修复失败则回退到最近一次可编译版本
- 日志先过滤：只给首个错误及其上下文
- 每轮只改一个文件，便于diff审查

UVM²：最多两轮修复后的生成成功率

69.13% → 86.67%

首轮　　修复后

来源：UVM², arXiv:2504.19959（Testbench级生成成功率）

兼容并包

示例：编译错误自动修复

- 错误日志 → LLM定位根因 → 最小修改 → 重新编译验证

① 编译日志（示例，格式化）

```
Error-[IND] Ide      not declared
fifo_driver.sv,
  Identifier 'W      has not been
  declared yet.
```

② LLM定位与分析

根因：driver使用的枚举名WR_RD与fifo_txn中定义的WRITE_READ不一

策略：以公共类型定义为准，只改driver，不改transaction

③ LLM输出的最小修改（unified diff）

```
--- a/fi    driver.sv
+++ b/fi    driver.sv
@@ -17,    +17,2 @@
-       vif.wr_en <= req.op inside {WRITE, WR_RD};
-       vif.rd_en <= req.op inside {READ,  WR_RD}
+       vif.wr_en <= req.op inside {WRITE, WRITE_READ};
+       vif.rd_en <= req.op inside {READ,  WRITE_READ};
```

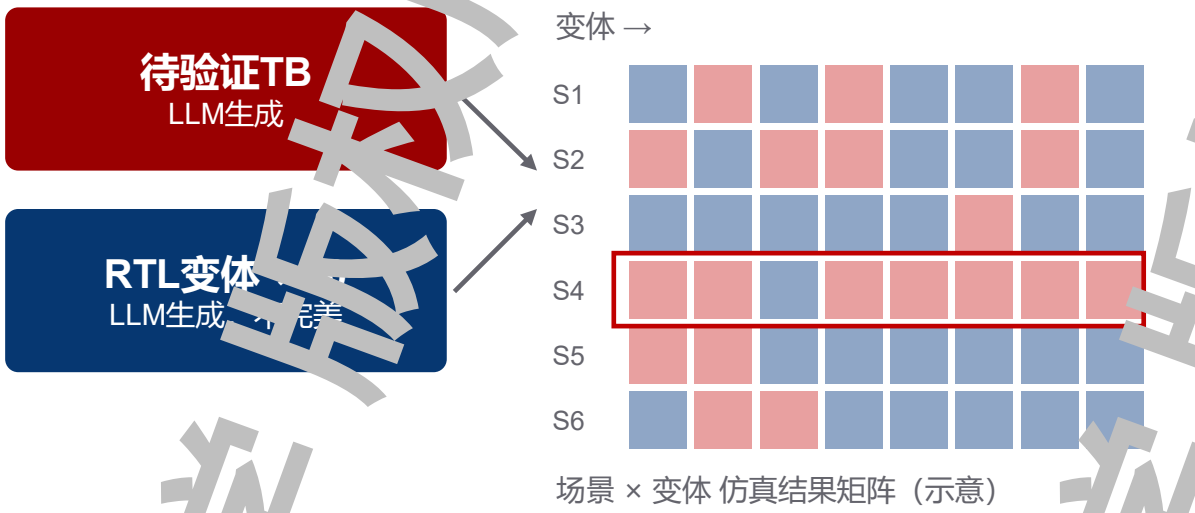
只喂首个错误 + 上下文

限定修改范围

改完立即重新编译

功能正确性：Testbench自己也可能是错的

- 编译通过 ≠ 测对了——CorrectBench用多份RTL变体交叉验证TB



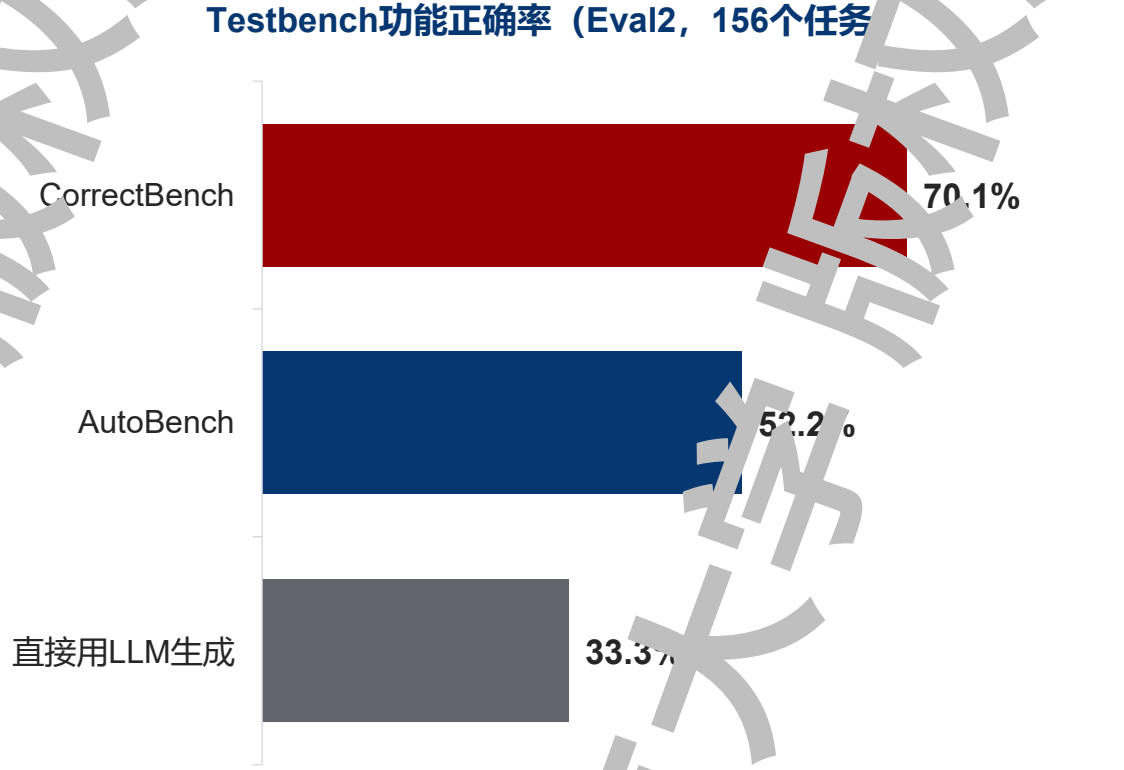
判定规则

- 某场景≥70%的变体与TB不一致 → 判为TB错误场景
- 全部变体完全通过 → 直接接受该TB

修正 ≤3次

重启生成 ≤10次

确认



来源：R. Qiu et al., CorrectBench, arXiv:2411.085[illegible] AutoBench, arXiv:2407.03891。验证器判定准确率88.85%。

思想自由 兼容并包

参考模型生成：LLM写C模型，经DPI-C接入

- 用与RTL不同的实现路径产生“期望值”，降低同源错误

C参考模型

```
// fifo_model.c  —— 生成的 C 参考模型
#include "svdpi.h"
#define DEPTH 16
static unsigned int q[DEPTH];
static int head = 0, tail = 0, cnt = 0;

void model_reset(void) { head = tail = cnt = 0; }

int model_push(unsigned int d) {     // 满则返回 0
  if (cnt == DEPTH) return 0;
  q[tail] = d; tail = (tail + 1) % DEPTH; cnt++;
  return 1;
}

int model_pop(unsigned int *d) {     // 空则返回 0
  if (cnt == 0) return 0;
  *d = q[head]; head = (head + 1) % DEPTH; cnt--;
  return 1;
}
```

SystemVerilog侧：DPI-C导入与调用

```
import "DPI-C" function void model_reset();
import "DPI-C" function int model_push(int unsigned d);
import "DPI-C" function int model_pop(output int unsigned d);

// scoreboard 中调用参考模型
function void write(fifo_txn t);
  int unsigned exp;
  if (t.op == WRITE) void'(model_push(t.data));
  else if (!model_pop(exp))
    `uvm_error("SB", "read from empty model");
  else if (t.data !== exp)
    `uvm_error("SB", "data mismatch");
endfunction
```

- **独立实现路径**：与RTL由不同prompt、不同语言产生
- **更接近算法规格**：C / Python描述行为，LLM生成正确率更高
- **可跨层复用**：同一模型可用于系统级、软硬件协同验证

示例：LLM生成FIFO断言（SVA）

- 自然语言性质 → SVA断言，可写入interface或通过bind挂到DUT

自然语言性质	生成的断言
复位释放时FIFO为空	`p_reset_empty`
满时只写不读，计数保持不变	`p_no_write_when_full`
full与empty不能同时为1	`p_full_empty_mutex`
full状态至少被观测一次	`cover property`

人工审查要点

- 时钟 / 复位语义与disable iff条件
- |-> 与 |=> 的一拍之差
- 配套cover，避免断言“空洞通过”

LLM输出（经人工审查）

```
// 1. 复位释放时 FIFO 为空
property p_reset_empty;
  @(posedge clk) $rose(rst_n) |-> empty;
endproperty

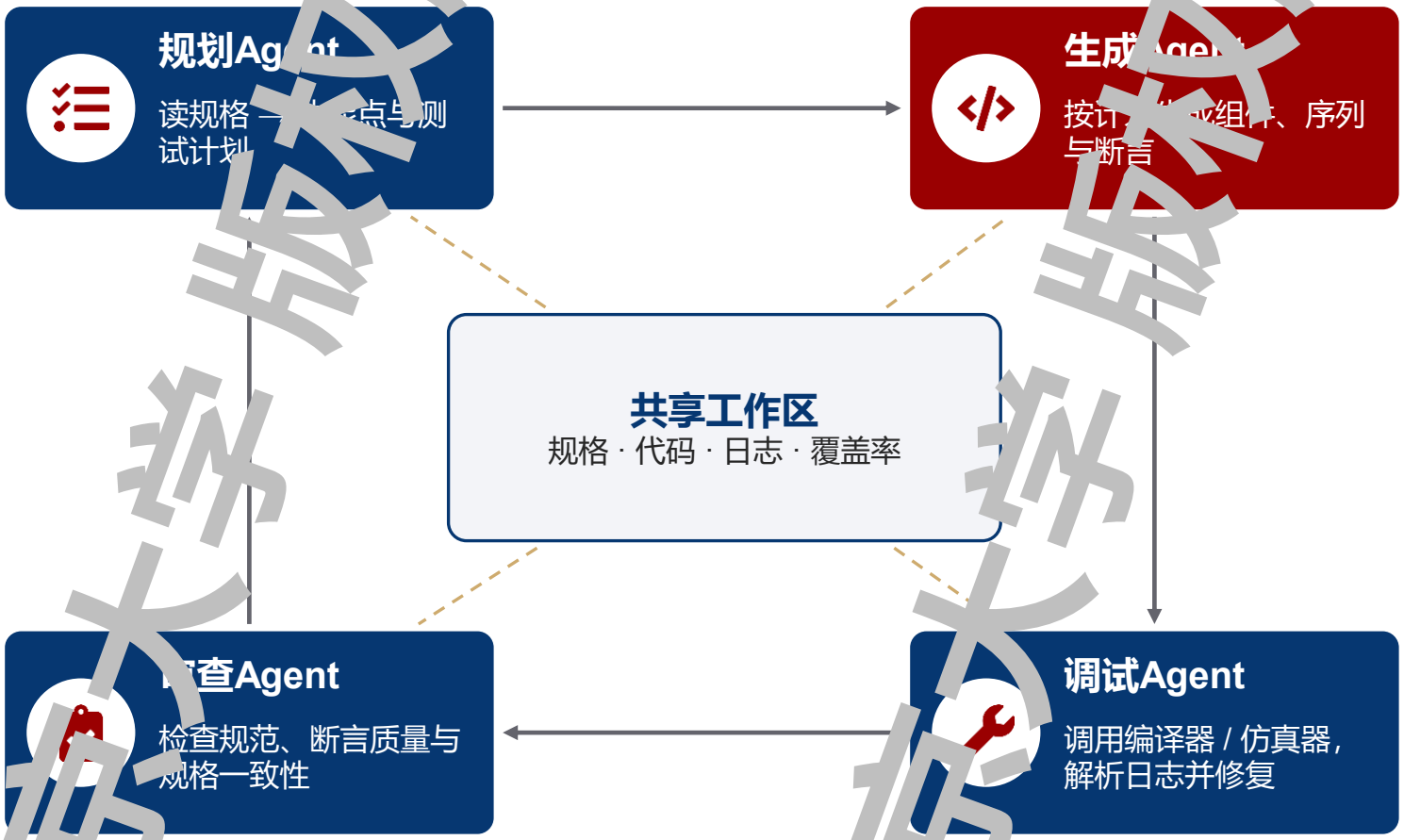
// 2. 满时只写不读：计数保持不变
property p_no_write_when_full;
  @(posedge clk) disable iff (!rst_n)
  (full && wr_en && !rd_en) |=> $stable(count);
endproperty

// 3. full 与 empty 互斥
property p_full_empty_mutex;
  @(posedge clk) disable iff (!rst_n) !(full && empty);
endproperty

a_reset : assert property (p_reset_empty);
a_full  : assert property (p_no_write_when_full);
a_mutex : assert property (p_full_empty_mutex);
c_full  : cover  property (@(posedge clk) full);
```

多智能体验证框架

- 按角色拆分：规划、生成、调试、审查各司其职，围绕共享工作区协作



设计要点

- **角色分离**：生成者不自评，审查交给独立agent
- **工具即裁判**：编译、仿真、覆盖率结果作为客观反馈
- **共享状态 + 版本管理**：每步产物可追溯、可回退
- **人在回路**：测试计划、覆盖率核等关键节点由工程师审批

实例：UVM² 采用分析（AgentA）、生成（AgentG）、优化（AgentC）分工（arXiv:2504.19959）

思想包袱 兼容并包

研究与产业进展一览

- 学术界聚焦TB生成与自校正，工业界把AI嵌入调试与回归平台

工作	年份	核心思路	代表结果
VerilogEval	[illegible]	Verilog设计题 + 参考实现的评测基准	常被用作TB生成的评测载体
AutoBench	2024	Verilog驱动 + Python检查器的混合TB [illegible]	Eval2通过率52.18%
CorrectBench	2024	RTL变体交叉验证TB，功能自校正	Eval2通过率70.13%
MEIC	2024	LLM迭代修复RTL的语法与功能错误	RTL调试自动化
UVLLM	2024	UVM环境 + LLM定位并修复RTL错误	语法修复率86.99%，功能修复率71.92%
UVM²	2025	多智能体生成完整UVM环境 + 覆盖率补充	代码 / 功能覆盖87.44% / 89.[illegible]%

工业界：EDA厂商推出生成式AI助手（如Synopsys.ai Copilot），并把AI嵌入调试与回归平台（如Cadence Verisium）[illegible]能力以厂商文档为准

来源：VerilogEval arXiv:2309.07544；AutoBench arXiv:2407.03891；CorrectBench arXiv:2411.08510；MEIC arXiv:2405.06840；UVLLM arXiv:2411.16238；UVM² arXiv:2504.19959

思想自由 兼容并包

局限、风险与人机协同

- LLM不是“验证审核者”：需要独立参考与人工把关

风险	对策
语法正确、语义错误 协议时序“幻觉”，编译通过却测错	**独立参考模型** C / Python golden model与DUT比对
同源偏差 TB与RTL基于同一份错误理解，互相“印证”	**变异测试（Mutation）** 向RTL注入缺陷，检验TB能否发现
覆盖过弱或过强 漏洞通过，或大量误报淹没真问题	**cover + vacuity检查** 确认每条断言的前件确实被触发
数据与IP安全 规格和RTL外泄到公有云模型	**私有化部署** 本地模型 + 本地RAG知识

北京大学版权所有

北京大学
PEKING UNIVERSITY

AI驱动的功能验证

时序逻辑的形式化验证与时态逻辑

北京大学版权所有

为什么时序逻辑需要形式化验证

- 仿真只能覆盖跑过的场景，形式化验证对所有可达状态给出证明或反例

$2^{64} \approx 1.8 \times 10^{19}$

64 位触发器可能的状态数

约 585 年

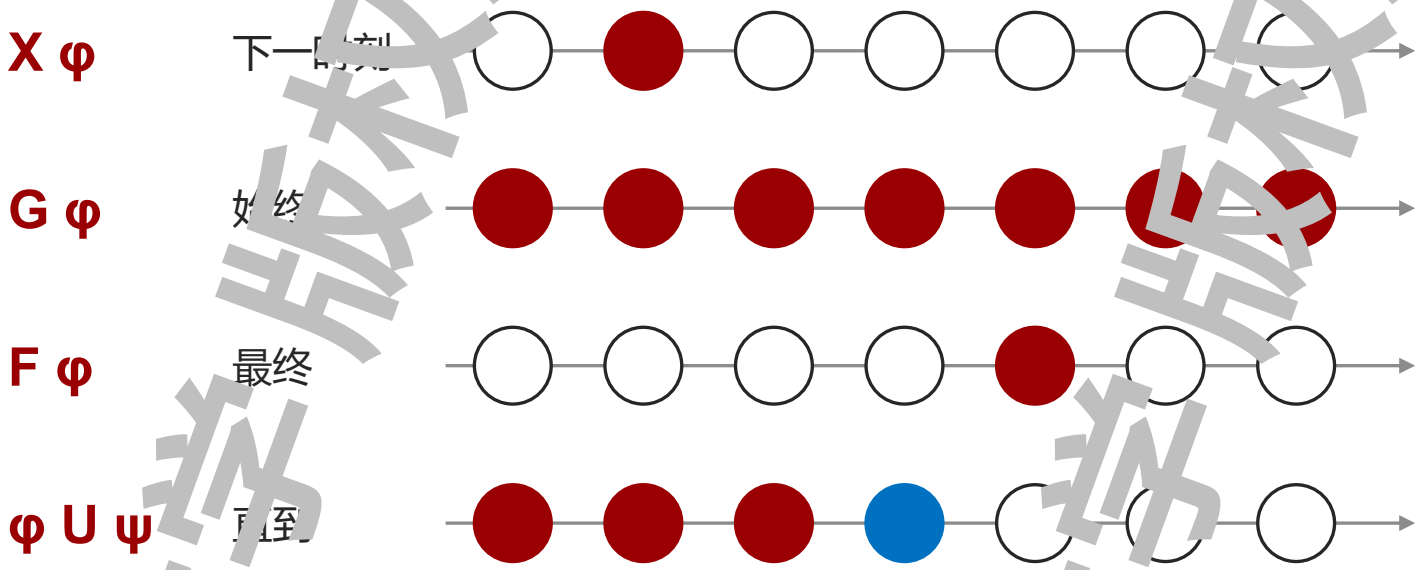
即使每秒遍历 10^9 个状态，穷举仍需的时间

	仿真	形式化验证
激励	人工或随机测试向量	无需激励，约束输入即可
覆盖范围	实际跑过的场景	所有可达状态（边界内）
结论	“没有发现错误”	证明成立，或给出反例
适用规模	可到整芯片	模块级、控制密集型逻辑
开源工具	Verilator、Icarus	SymbiYosys、ABC

FSM、仲裁器、握手接口、FIFO 控制逻辑……是形式化验证的“甜区”

线性时序逻辑（LTL）

- LTL 描述一条执行路径上性质随时间的变化，是 SVA 的理论基础



红点：φ 成立　蓝点：ψ 成立　从左到右为时间

G (req → F gnt)
每个请求最终都会被授权

G ¬(gnt0 ∧ gnt1)
两个授权不同时有效

安全性与活性

安全性：坏事永远不会发生

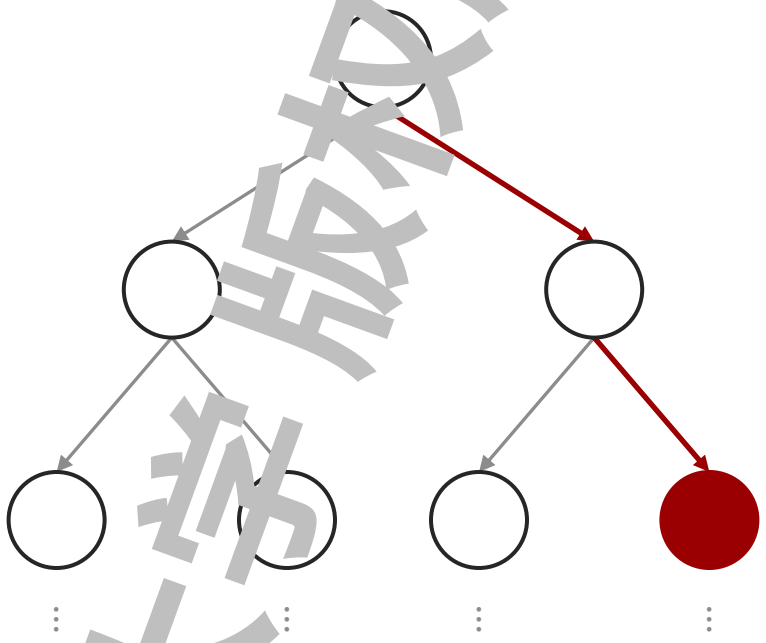
- 反例是一条有限长的路径
- 如互斥、FIFO 不溢出

活性：好事最终会发生

- 反例是一个无限循环
- 如无饥饿、无死锁

计算树逻辑（CTL）与路径量词

- CTL 在**分支的计算树**上推理：A 表示所有路径，E 表示存在某条路径



示例 EF p：存在一条路径（红）最终到达 p 成立的状态

AG p

所有路径上始终 p

不变式：FIFO 永不溢出

EF p

存在路径最终到达 p

可达性：错误状态可能出现

AF p

所有路径最终到 p

必然响应：请求必被处理

EG p

存在路径始终保持 p

可能永远停在某状态

LTL 与 CTL 的表达能力**互不包含**：工业界断言语言（SVA、PSL）主要基于线性时序

CTL 的 EF 常用于“这个状态能否到达”的覆盖检查

SystemVerilog 断言（SVA）

- SVA 用周期级语法书写时态逻辑：assert、assume、cover 各司其职

arbiter_prop_.sv

```
// 请求上升后 1~4 拍内必须授权
property p_req_gnt;
  @(posedge clk) disable iff (!rst_n)
  $rose(req) |-> ##[1:4] gnt;
endproperty
a_req_gnt: assert property (p_req_gnt);

// 授权互斥（最多一个有效）
a_mutex: assert property (
  @(posedge clk) $onehot0({gnt0, gnt1, gnt2}));

// 环境假设：请求在被授权前保持
m_req_hold: assume property (
  @(posedge clk) req && !gnt |=> req);

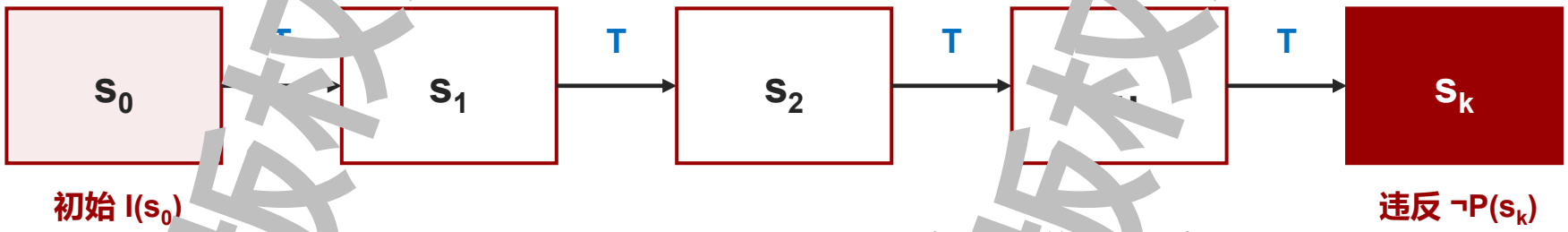
// 覆盖：同一请求者连续两次获得授权
c_b2b: cover property (@(posedge clk) gnt0 ##[1:$] gnt0);
```

算子	含义
##n	延迟 n 拍
##[m:n]	延迟 m 到 n 拍
|->	同拍蕴含
|=>	下一拍蕴含
$rose / $fell	上升沿 / 下降沿
$stable / $past	保持不变 / 前一拍值
[*n]	连续重复 n 次

assert ≈ AG，cover ≈ EF

模型检测与有界模型检测（BMC）

- BMC 把电路展开 k 步交给 SAT：快速找到最短反例，但不是完整证明



展开深度 k
逐步加深

SAT Solver具体实现方式（学习可自行研究

$$I(s_0) \wedge T(s_0, s_1) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge (\neg P(s_0) \vee \ldots \vee \neg P(s_k)) \rightarrow \text{SAT 求解器}$$

1 模型检测思想

- 在状态空间上穷尽检查性质
- 符号化方法（BDD、SAT）缓解状态爆炸

2 可满足 → 反例

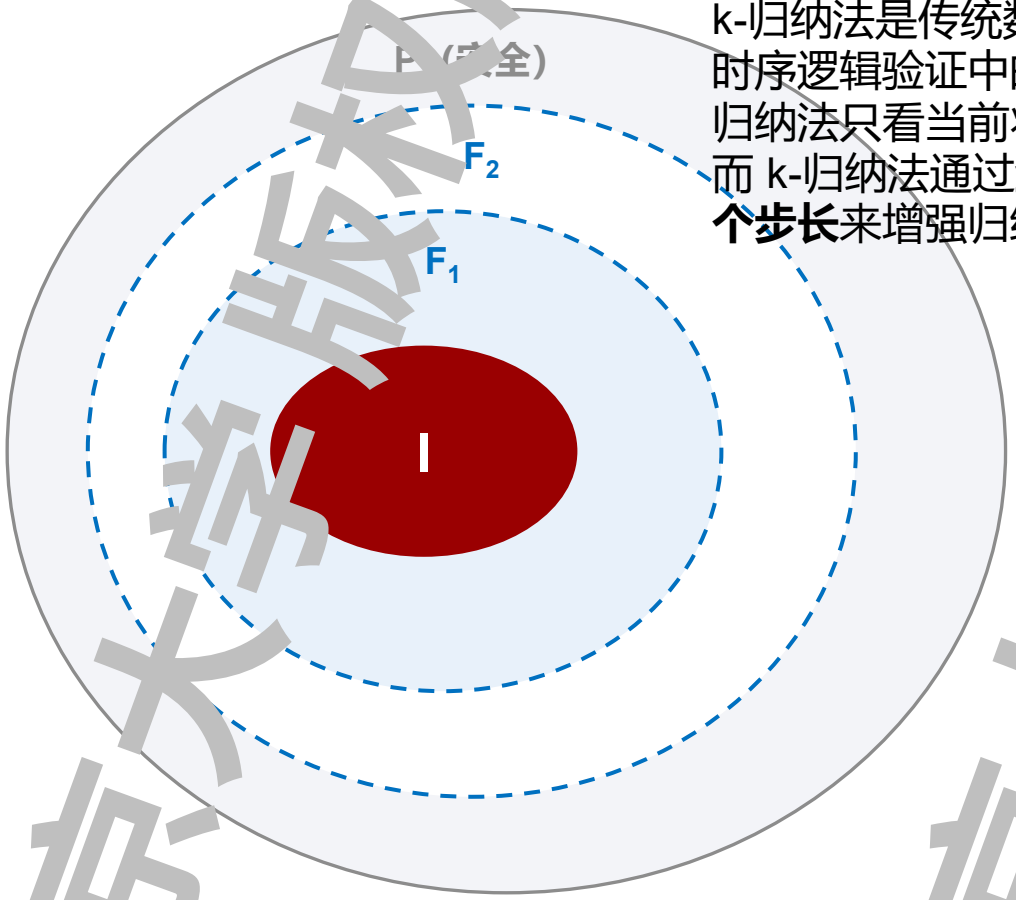
- 得到一条长度 ≤ k 的违例轨迹
- 导出 VCD 波形，便于调试

3 不可满足 → 界内安全

- k 步内不存在反例
- **不等于**完整证明，需要归纳

k 归纳与 IC3 / PDR

- **完整证明需要归纳不变式**：包含初始状态、对转移封闭、排除坏状态



F_i：i 步内可达状态的过近似；$I \subseteq F_1 \subseteq F_2 \subseteq \ldots \subseteq P$

k-归纳法是传统数学归纳法在时序逻辑验证中的扩展。传统归纳法只看当前状态（k=0），而 k-归纳法通过**查看连续的 k 个步长**来增强归纳假设。



k 归纳

- 基础：BMC 证明前 k 步都满足 P
- 归纳：任意连续 k 步满足 P，则下一步也满足
- 难点：不可达状态破坏归纳，需要补充引理

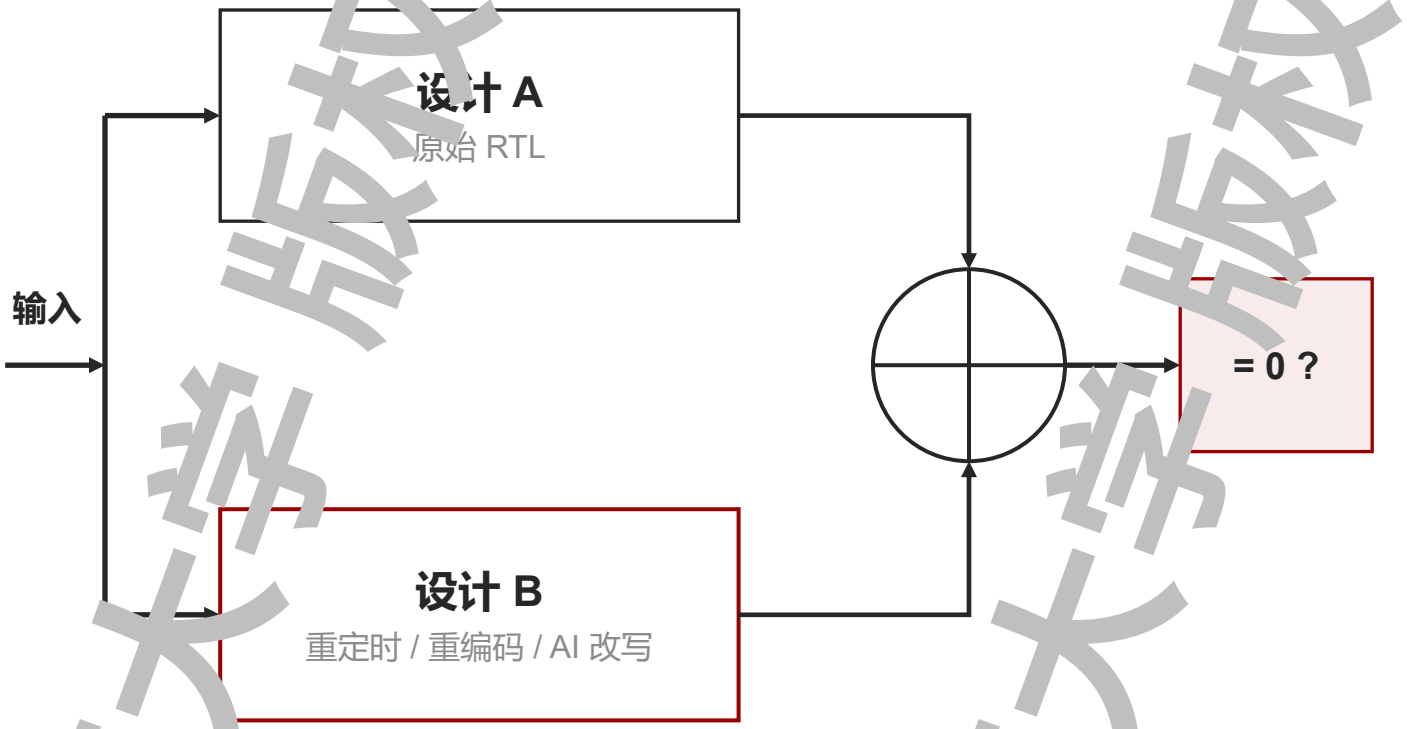
IC3 / PDR

逐帧学习子句，阻断反例前驱；相邻两帧相同即得不变式

AI 机会：LLM 提出**候选不变式**（如“计数器不超过深度”），交给工具验证

时序等价性检查（SEC）

- 重定时、重编码或 AI 改写之后，用 SEC 证明新旧设计**输出始终一致**



Miter 电路：共享输入、比较输出，证明异或结果在所有可达状态下恒为 0

	组合等价	时序等价
前提	寄存器一一对应	不要求对应
方法	逐个比较组合锥	乘积机可达性 / 归纳
典型用途	综合前后检查	重定时、FSM 重编码、门控插入

与 AI 的结合

LLM 做“性能优化”改写后，SEC 是判断是否改坏功能的最后一道关

AI 方法：LLM 从规格生成 SVA 断言

- LLM 擅长把自然语言翻译成断言，但每条断言都要经过多重筛选才能入库

规格语句	LLM 生成的 SVA	需要人确认的点
请求后 4 拍内必须授权	$rose(req) |-> ##[1:4] gnt	“4 拍内”是否包含第 4 拍
FIFO 满时不允许写入	full |-> !wr_en	wr_en 指外部请求还是内部写使能
复位释放后 valid 为低	$rose(rst_n) |-> !valid	检查同拍还是下一拍

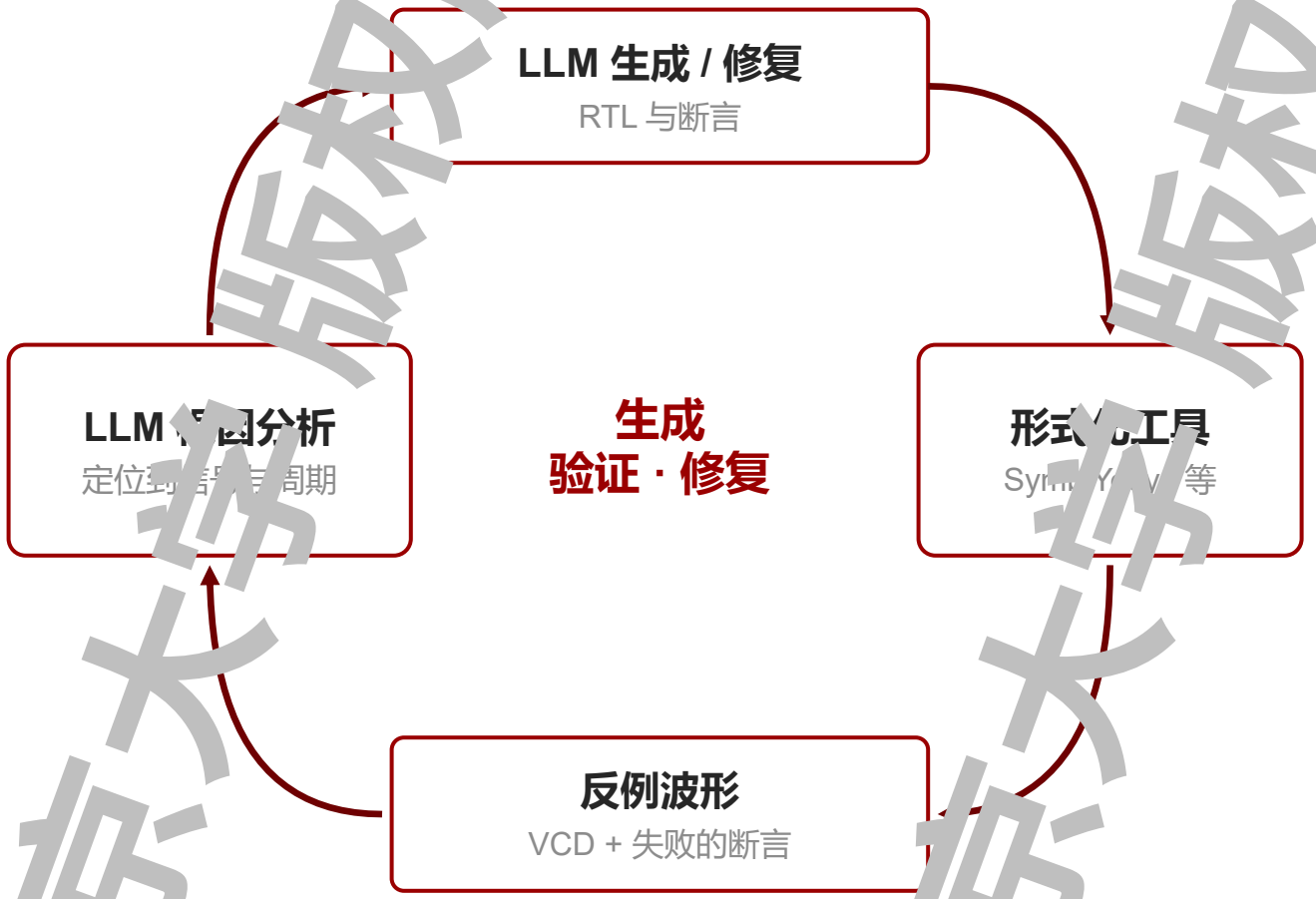
断言质量筛选流程

语法与编译检查 → 在参考设计上证明 → Cover 检查非空真 → 变异测试应能报错 → 人工审核入库

代表工作：AssertLLM 等用多个 LLM 从设计文档生成并评估断言（2024）；开源 RTL 与断言基准正在快速增加

“生成—验证—修复”闭环

- LLM 负责提出方案，形式化工具负责裁决，**反例**是两者之间的通信语言



停止条件

- 全部断言证明通过，且 cover 均可达
- 达到迭代上限（如 5 轮）后转人工

风险与对策

- LLM 改断言“让它通过” → 断言文件只读
- assume 过强使证明空洞 → assume 需人工审批
- 有界证明被当成完整证明 → 报告中标注证明方式与深度