北京大学
PEKING UNIVERSITY

基于AI的数字系统设计

第7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

Lab相关事项

- 目前所有小组的GPT订阅均已经完成

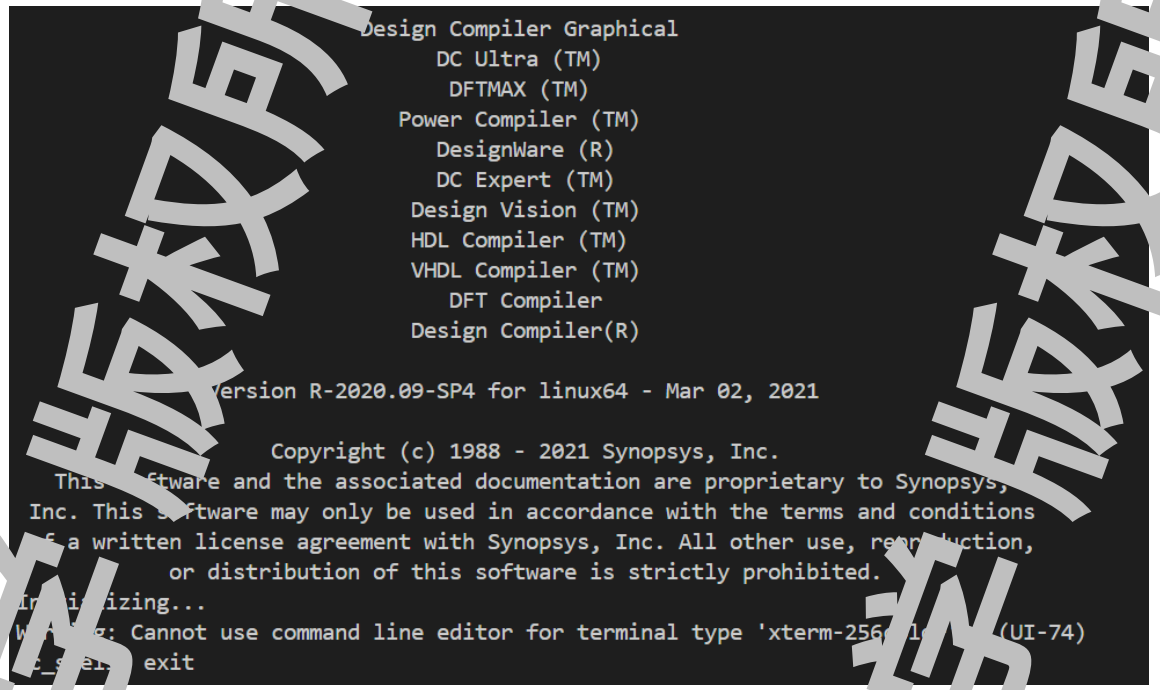
小组序号	组[illegible]1	组员1学号	组员2	组员2学号	组员3	组员3[illegible]号	组员4	组员4学号	组长	chat[illegible]订阅
1	[illegible]睿隽	2400012725	刘轩池	2400012730	丁一帆	24[illegible]012422	李雨农	2300013261	刘轩池	完[illegible]
2	[illegible]冠霖	2300010870	王博达	2300016624	王陈琎	[illegible]00011887	邢梓童	2400012807	吴冠霖	完[illegible]
3	[illegible]	2400012843	孙睿扬	2400012829	迟海	2400012893	郝昶越	2300012703	田先宇	完[illegible]
4	[illegible]绵哲	2400012817	梁墉	2400012890	王梓[illegible]	2400012703	李亦恒	2400012808	王梓淇	完[illegible]
[illegible]	[illegible]杰[illegible]	2400012828	王佳禾	2400012816	张天[illegible]	2400012707	黄颂涵	2400012702	李香逸	[illegible]
[illegible]	郑[illegible]	2400012711	赵智远	2400012709	章[illegible]	2400012892	潘浩天	2400012831	赵智远	完成
[illegible]	王诗[illegible]	2300012855	鄢川林	2400012848	刘重[illegible]	2400012876	罗力航	2400012872	刘重阳	[illegible]成
8	徐培文	2400012812	王志鸿	2400012803	王云翮	2400012713	赵弈涵	2400012733	徐培文	完成
[illegible]	兰航	2400012834	李唯嘉	2400012810[illegible]	[illegible]信阁	2400012883	黄昊	2400012856	冷信阁	完成
[illegible]0	廖泓凯	2400012854	李睿	2400012[illegible]	[illegible]	2400012853	湛梓豪	2400012841	李睿	完成
[illegible]	程沐阳	2400012724	王精午	2400012[illegible]	[illegible]城[illegible]	2300012757	吴博扬	2400010967	梁城[illegible]	[illegible]成
12	杨文昊	2400012879	林子童	2400013227	葛[illegible]昊	2400012891	冯晟轩	2300012848	杨文昊	完成
13	郦查德	2400011035	朱俊安	2400012[illegible]	林轩丞	2400011038	胡凯楠	2200012442	郦查德	完成
14	陈影轩	2400012898	曹恒瑞	2400[illegible]	[illegible]桐	2400012824	曾魁隽	2400012838	[illegible]	完成
15	安正昊	2300012707	何家俊	2400012832	高贺研	2300012882	肖天尧		安[illegible]昊	完成
16	戴嘉震	2400012714	屠跃然	2400011102	[illegible]翔一	2400012857	夏米乐	2300011088	屠跃然	完成
17	杨睿浩	2400012885	王一诺	2400012827	周鸿杰	2400012858	叶尔努尔·叶尔多	2400012[illegible]	[illegible]浩	完成
18	王梓名	2300012761	刘煜淳	2300012872	许家悦	2300012817	罗逸涵	2300012[illegible]	许[illegible]悦	完成
19	马志云	2400012861	吕璨	2400012[illegible]	刘忠坤	2400012819	李涅	2400012[illegible]	吕璨	完成
20										

Lab相关事项

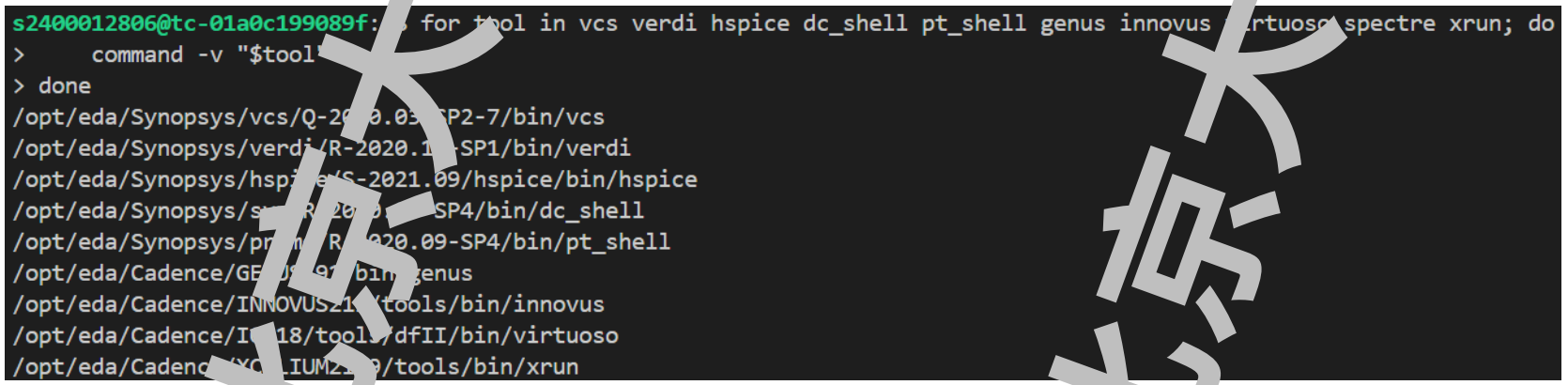
- 目前所有小组的GPT订阅均已经完成

- 由于xlab当前，单个课程内所有同学的子节点都是同一个ip出口
- 如果同一个ip调用多个AI，很容易导致模型性能波动
- 所以请大家组内协调，在自己的本机上分批次登录chatgpt账号编程

  xlab教程：[快速开始 | xLab 文档]



xlab内部的EDA工具



Lab相关事项

- 服务器上已安装的 9 个工具，按设计流程排列：验证 → 综合 → 时序签核 → 物理实现 → 全定制

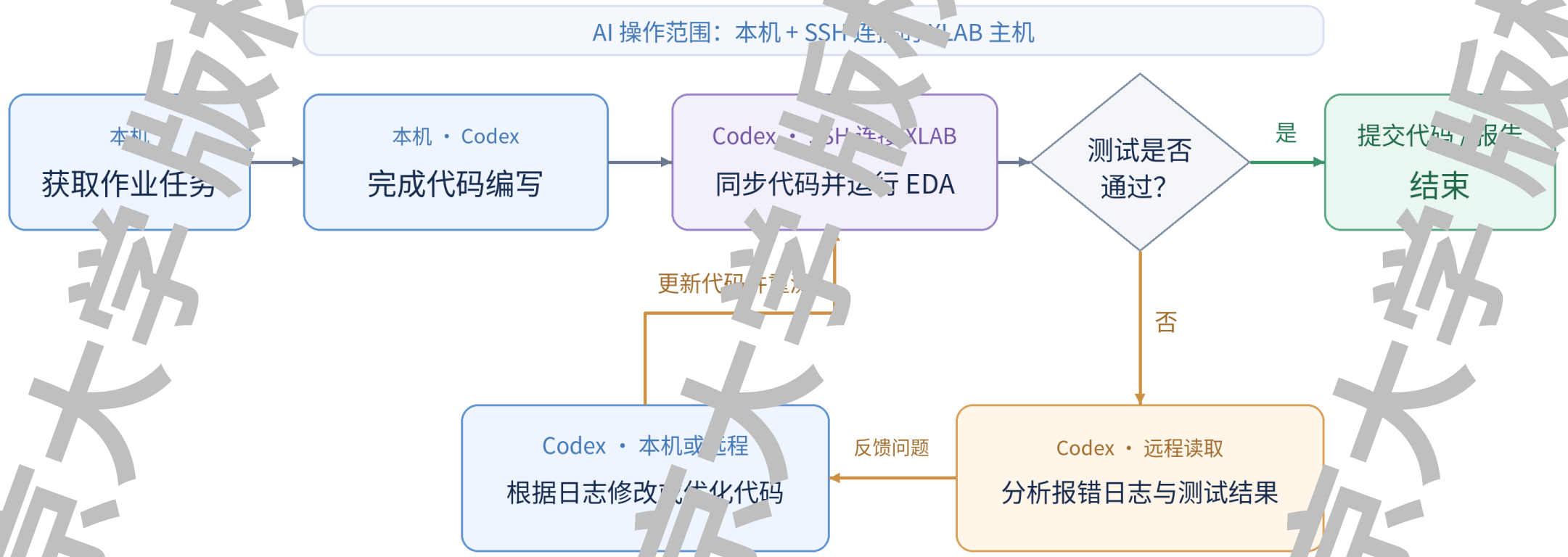
设计阶段	工具（命令）	厂商 · 版本	主要作用
数字仿真与验证	VCS vcs	Synopsys Q-2020.03-SP2	Verilog/SV 编译与逻辑仿真，支持 UVM 和覆盖率
	Xcelium xrun	Cadence 21.09	Cadence 仿真器，xrun 一条命令完成编译到仿真
	Verdi verdi	Synopsys R-2020.12-SP1	波形与源码联动调试，读取 FSDB 波形定位设计
逻辑综合	Design Compiler dc_shell	Synopsys R-2020.09-SP4	RTL → 门级网表，在时序、面积、功耗约束下优化
	Genus genus	Cadence 19.1	Cadence 逻辑综合工具，与 Innovus 后端流程配套
时序签核	PrimeTime pt_shell	Synopsys R-2020.09-SP4	静态时序分析，检查建立 / 保持时间，无需仿真激励
物理实现	Innovus innovus	Cadence 21.1	布局规划、布局、时钟树综合、布线，输出 GDSII 版图
全定制设计	Virtuoso virtuoso	Cadence IC6.1.8	原理图与版图编辑平台，用于模拟 / 混合信号电路设计
电路仿真	HSPICE hspice	Synopsys S-2021.09	晶体管级 SPICE 仿真，精确分析模拟电路和标准单元

思想包袱 兼容并包

Lab相关事项

- 远程XLAB+本地Codex测试与迭代流程

本机 Codex 通过 SSH 操作 XLAB，完成代码编写、远程测试与迭代



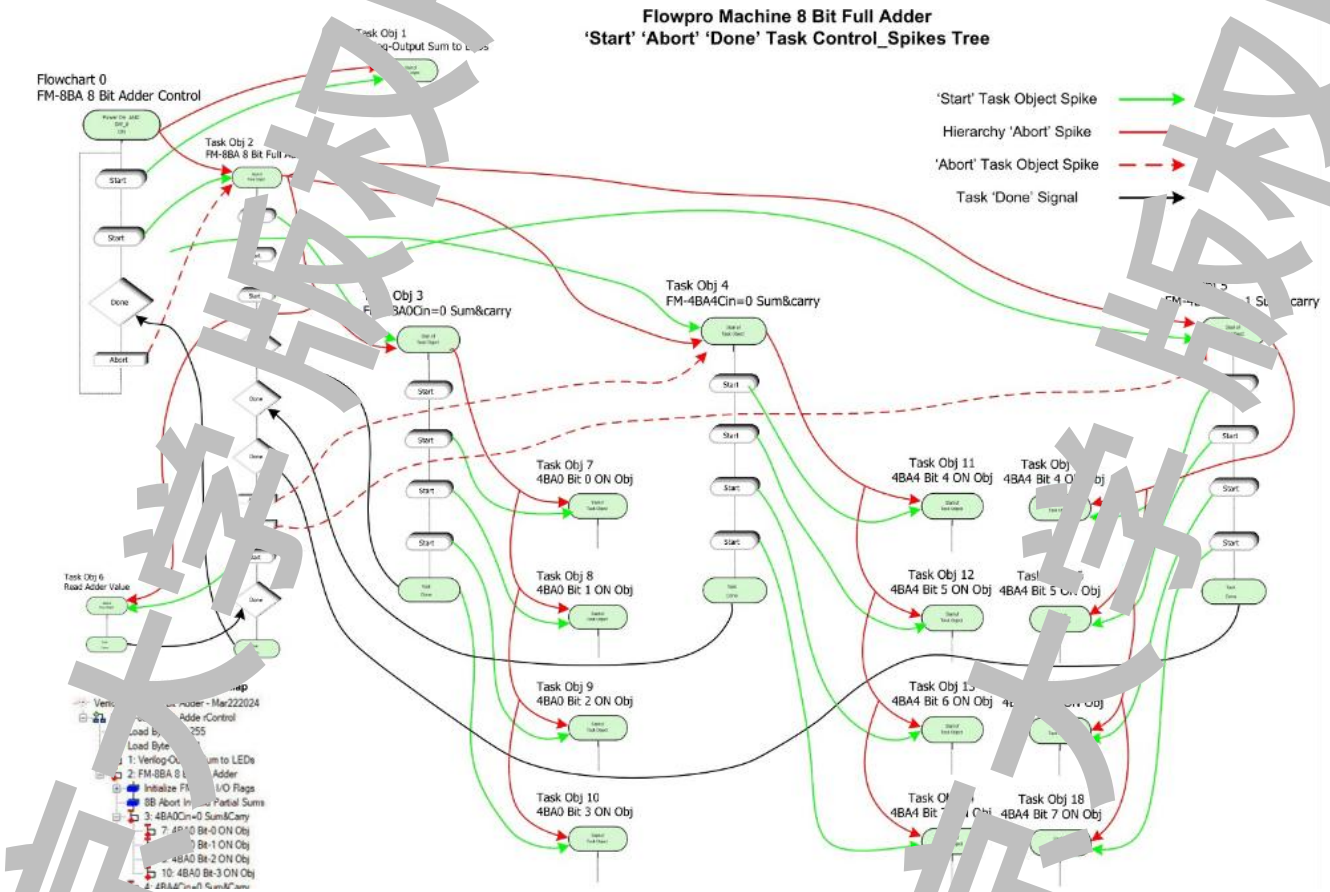
北京大学
PEKING UNIVERSITY

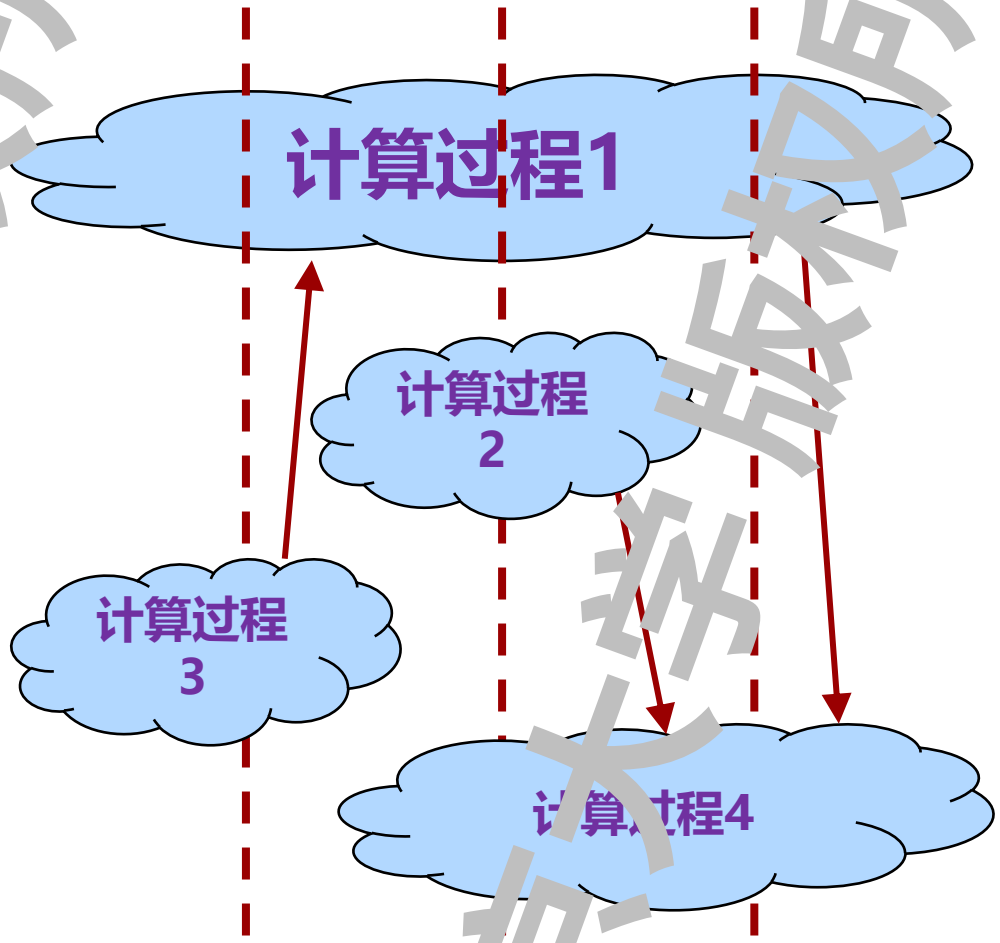
AI辅助时序逻辑设计

北京大学版权所有

电路时序的基本概念

- 电路为什么需要一个时钟?



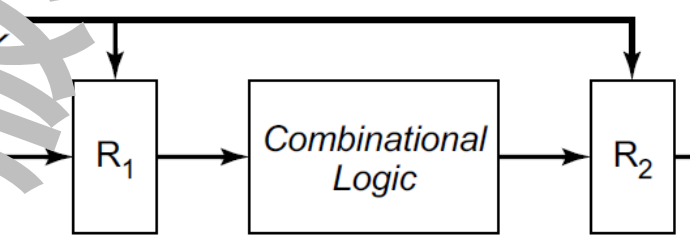


无时钟：非常难以控制每一个信号的有效时间

引入时钟：每隔一段计算将结果同步一次

电路时序的基本概念

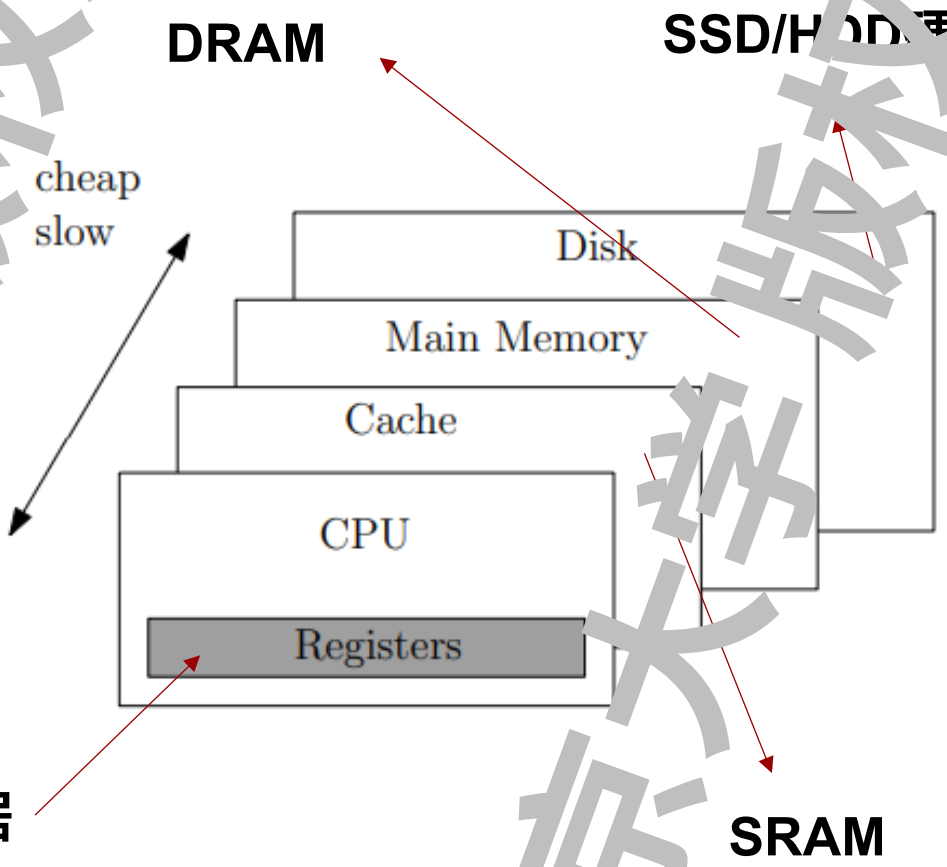
- 同步时序（Synchronous Timing）



寄存器 组合逻辑（各种逻辑门电路）

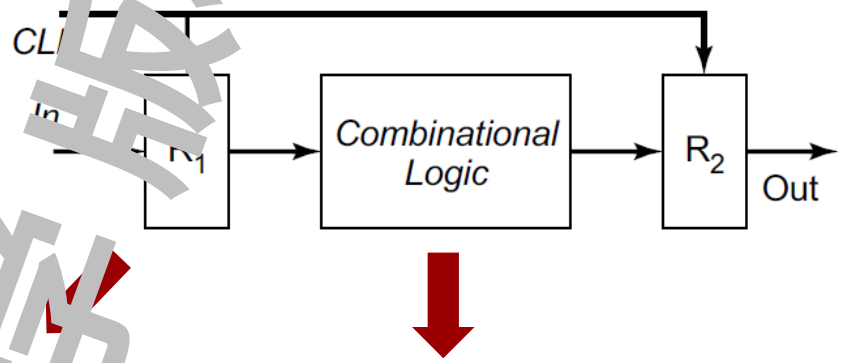
（register）

在芯片内用于暂存随时可能需要改变的数据



电路时序的基本概念

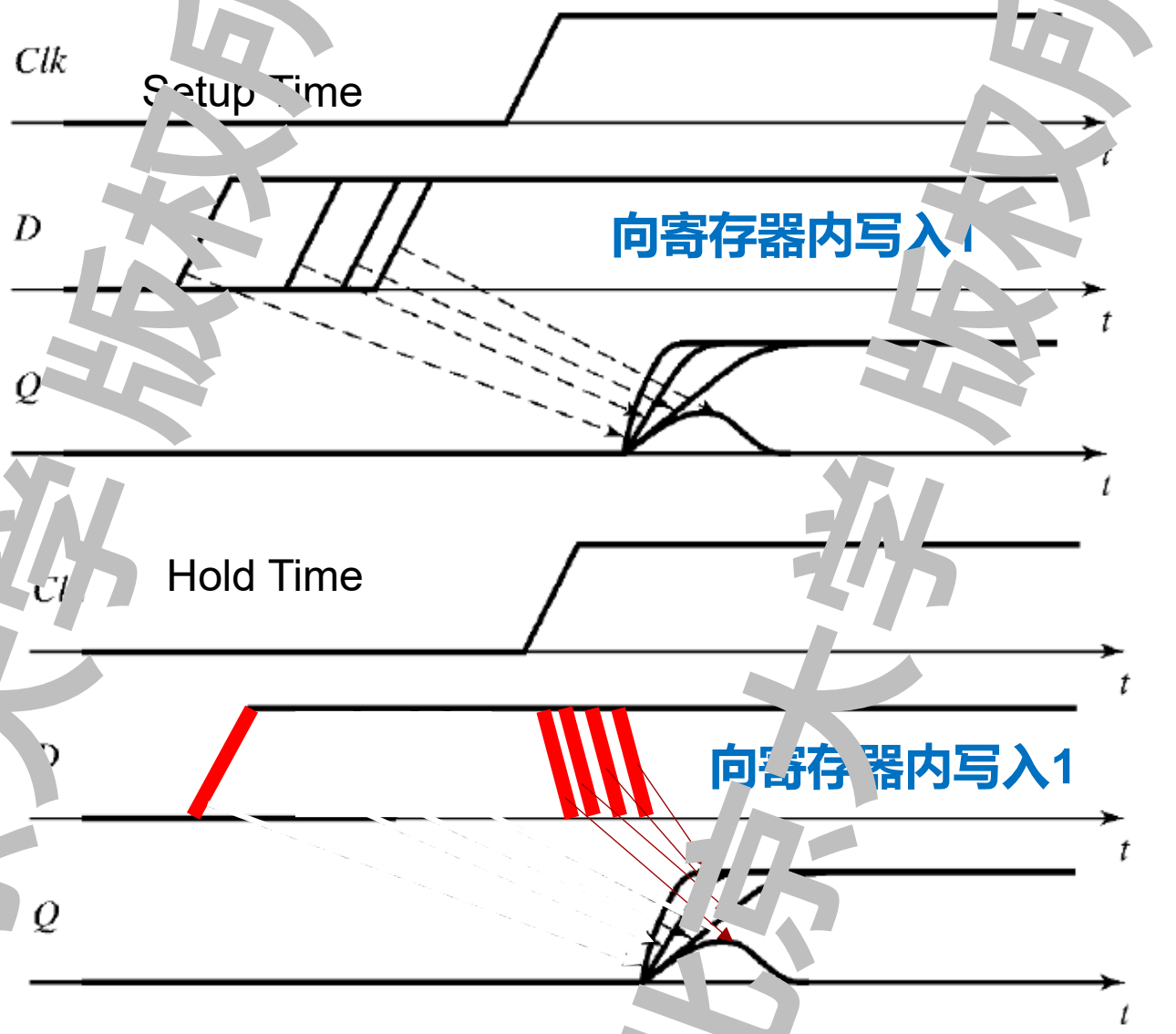
- 同步时序（Synchronous Timing）



寄存器 组合逻辑（各种逻辑门电路）

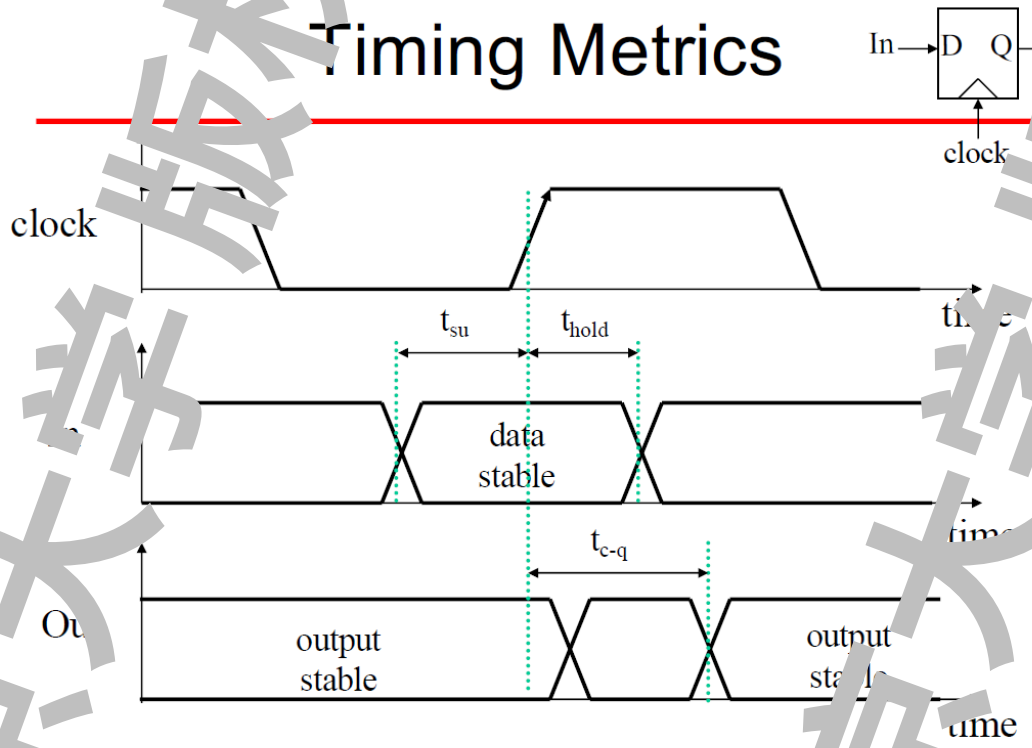
(Register)

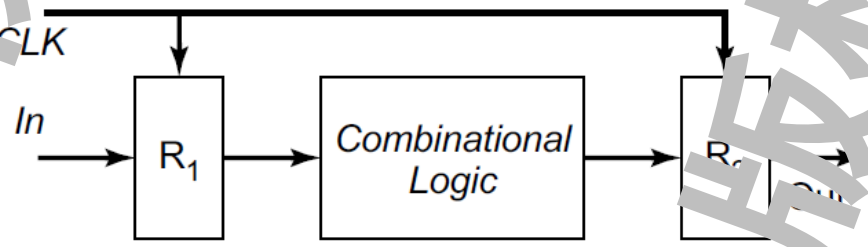
用于暂存随时可能需要改变的数据



电路时序的基本概念

- 同步时序（Synchronous Timing）





$$T_{c\text{-}q} + t_{plogic,min} \geq t_{hold}$$

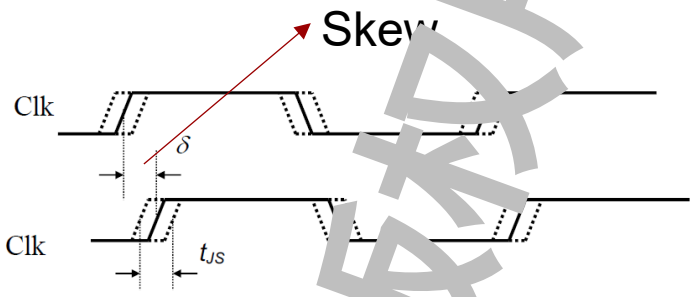
min

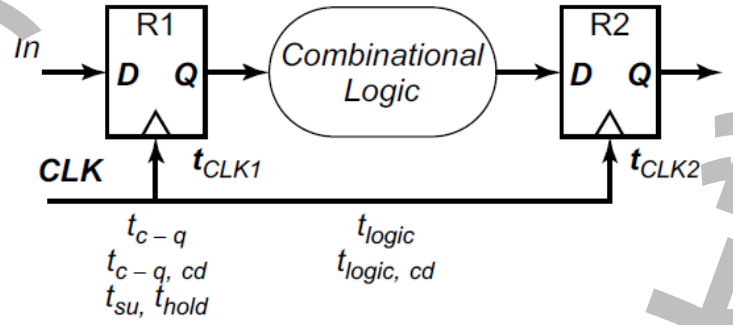
$$T \geq t_{c\text{-}q} + t_{plogic,max} + t_{su}$$

max

电路时序的基本概念

- 时钟的不稳定性

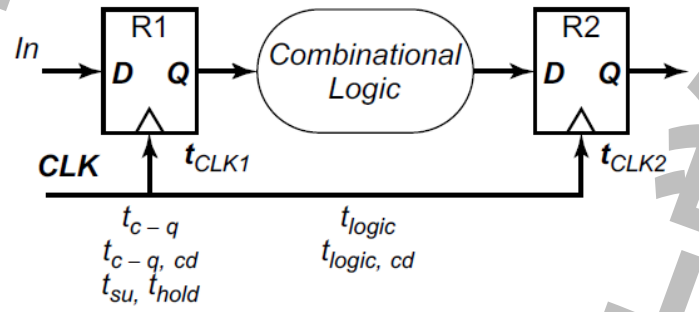




Minimum cycle time:

$T \geq t_{c\text{-}q} + t_{su} + t_{logic} - \delta$

最坏情况为接收边沿过早到达（negative δ）



Hold time constraint:

$t_{(c\text{-}q,\ cd)} + t_{(logic,\ cd)} > t_{hold} + \delta$

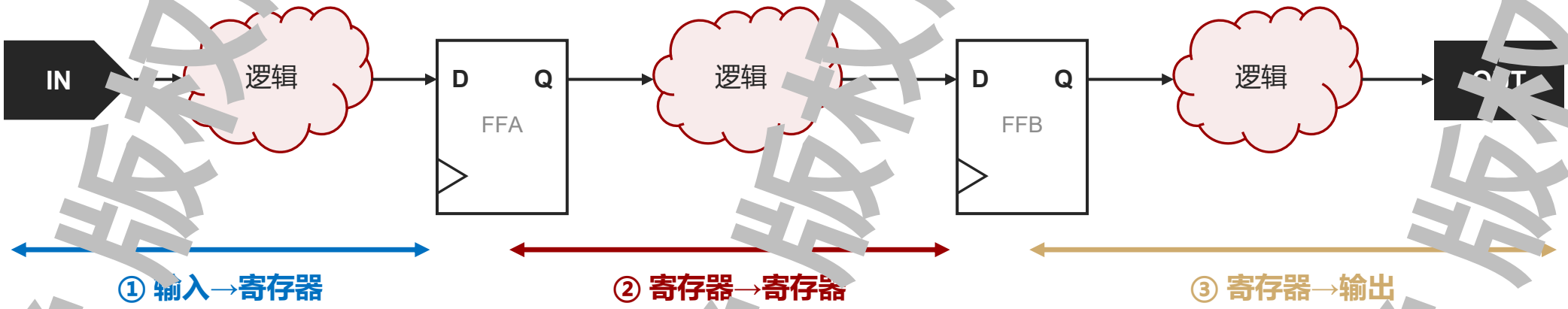
最坏情况为接收边沿过晚到达（正向偏差）

数据和时钟之间的竞争

Cd：contamination delay（最快可能延迟）

静态时序分析（STA）基础

- STA 不需要激励，**穷举所有时序路径**并计算每条路径的**裕量**（Slack）



④ 输入→输出（纯组合穿通路径）同样需要约束，常用 set_max_delay 描述

到达时间 AT

信号从起点出发，经 T_{cq} 与逻辑延迟到达终点的时刻

要求时间 RT

捕获时钟沿减去 T_{su}（建立检查）或加上 T_h（保持检查）

裕量 Slack

建立检查 Slack = RT − AT；Slack < 0 即违例，最差裕量记为 WNS

初步实践：LLM 生成的计数器

- LLM 代码“能编译”不等于“正确”：时序逻辑的每一处错误需要**逐条审查**

counter4.v

```
// LLM 生成：带使能的 4 位计数器
module counter4 (
  input  wire       clk, rst_n, en,
  output reg  [3:0] cnt
);
  reg [3:0] nxt;
  always @(en, begin                    // (3)
    if (en) nxt = cnt + 1'b1;   // (2)
  end
  always @(posedge clk or negedge rst_n)
    if (!rst_n) cnt = 4'd0;     // (1)
    else        cnt <= nxt;
endmodule
```

问题	后果	修正
(1) 时序块用阻塞赋值	仿真竞争，前后仿真不一致	统一用 <=
(2) if 缺少 else	推断出锁存器	补 else nxt = cnt
(3) 敏感列表不完整	仿真与综合结果不一致	改为 always @(*)

课后尝试

让 LLM 再生成一个 8 位移位寄存器，用 Verilator --lint-only 检查，再人工评审

北京大学
PEKING UNIVERSITY

AI辅助时序逻辑设计 - FSM

北京大学版权所有

更复杂的时序逻辑：状态机

- 控制电路的基本

Step 1 – 定义状态并画出状态转换图

Step 2 – 给每一个状态赋值并更新状态转换图

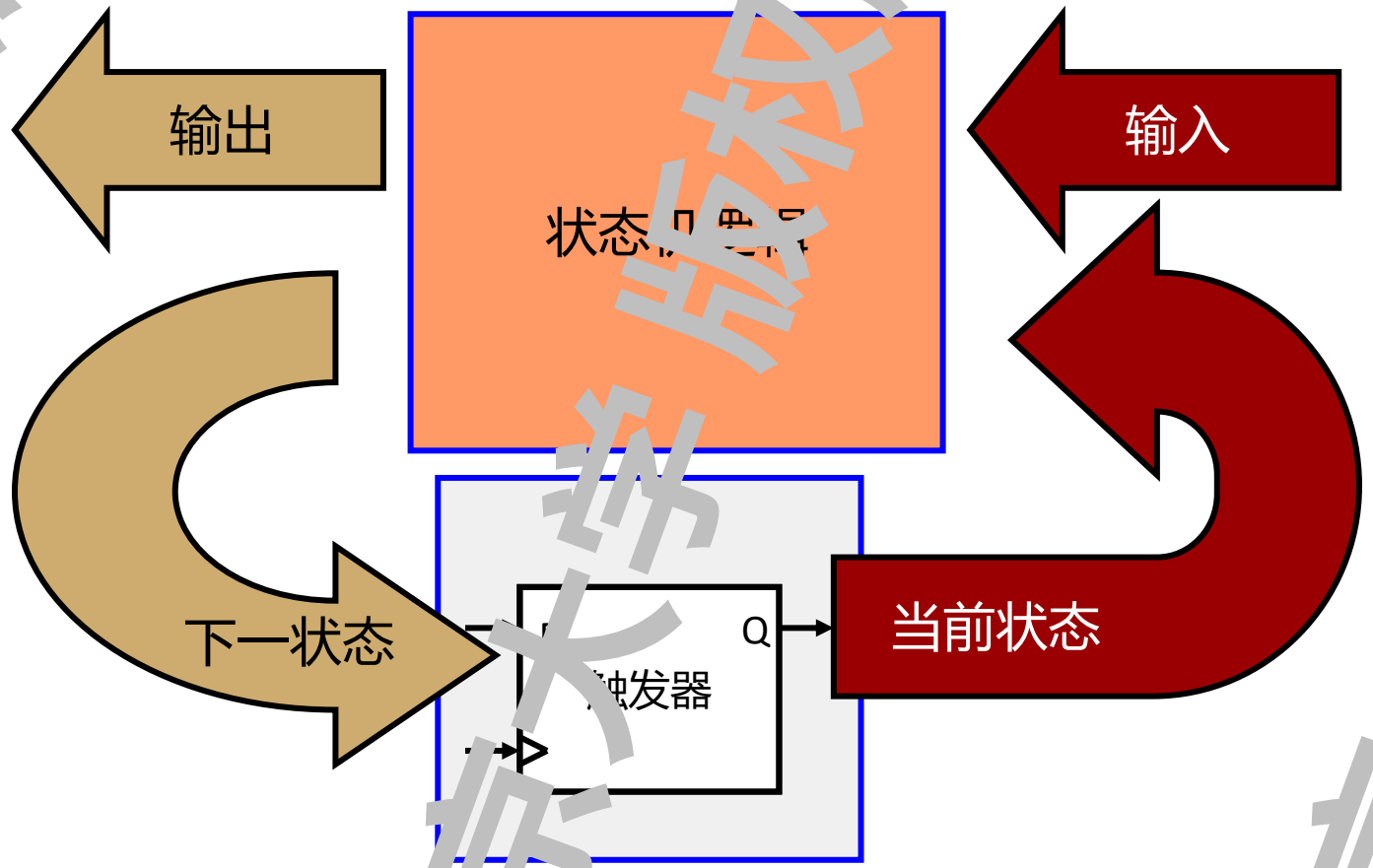
Step 3 – 根据状态转换图写出下一状态和输出的逻辑表达式

Step 4 – 画出实际电路图

状态将在每一个时钟上升沿更新

为什么需要状态机

- 控制电路的基本



有限状态机的形式化定义

- FSM 由六元组精确刻画，这也是让 LLM 生成 FSM 时的“填空清单”

S 状态集合

有限个状态，决定所需触发器数量

I 输入表

所有可能的输入组合

O 输出表

所有可能的输出组合

δ 状态转移函数

$\delta : S \times I \to S$，对应次态逻辑

λ 输出函数

Moore：$\lambda : S \to O$；Mealy：$\lambda : S \times I \to O$

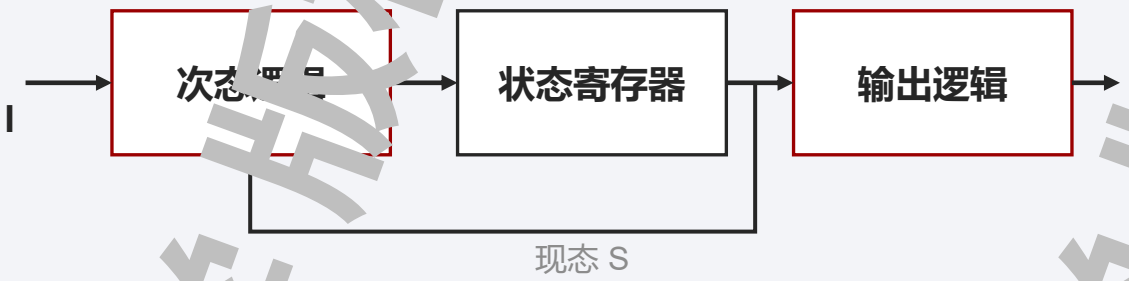
s_0 初始状态

复位后进入的状态，必须明确

Moore 型与 Mealy 型状态机

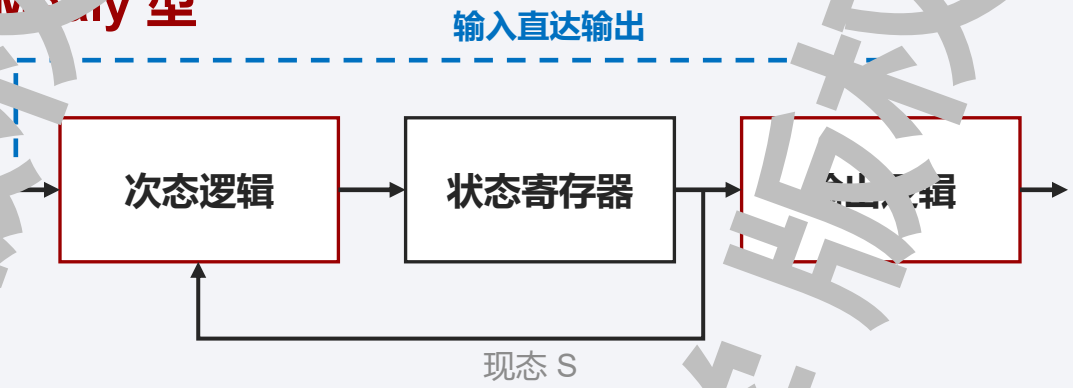
- 差别只在**输出是否直接依赖输入**：Moore 稳、Mealy 快

Moore 型



- 输出 O = λ(S)，只随状态变化
- 输出与时钟对齐，天然无毛刺
- 对输入的响应通常**晚一拍**，状态数偏多

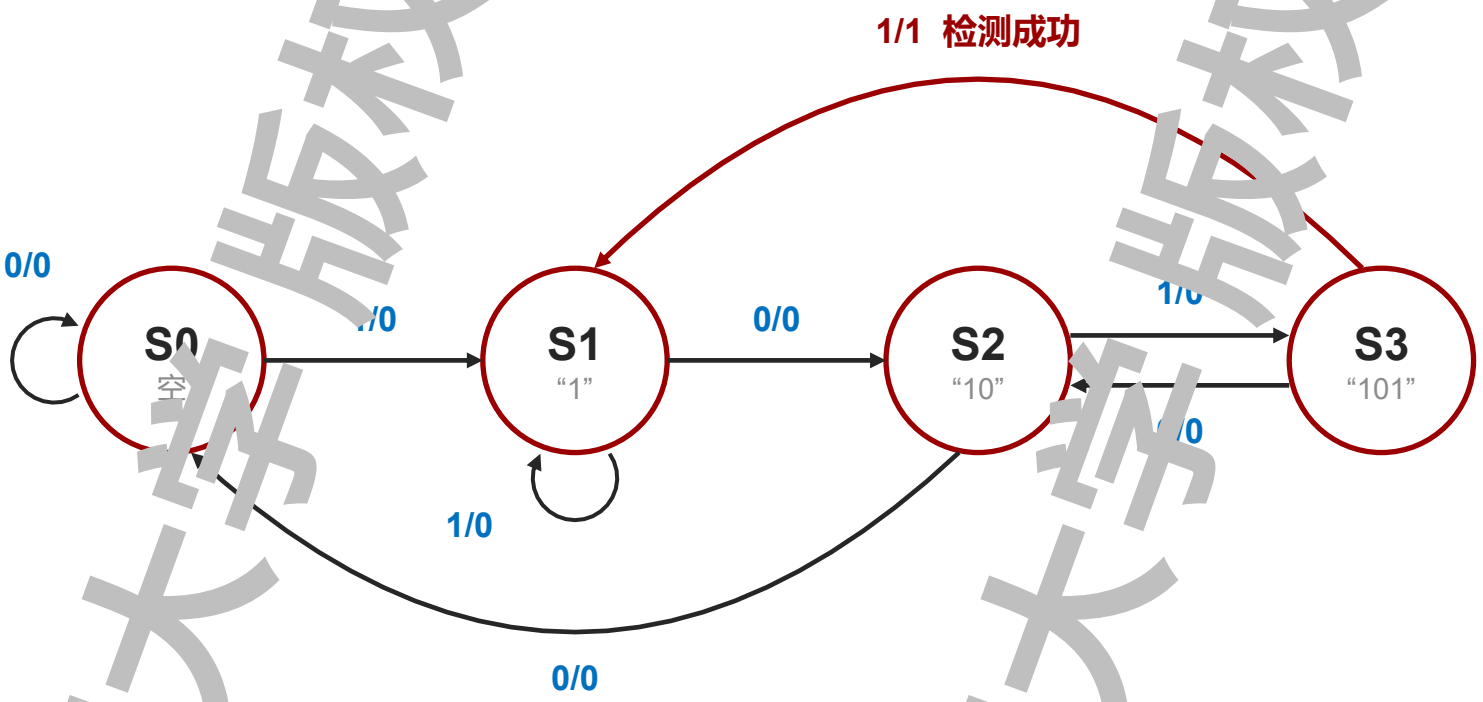
Mealy 型



- 输出 O = λ(S, I)，输入可直接影响输出
- 响应快、状态数少
- 输入毛刺会**传到输出**；常把输出再打一拍

示例：序列检测器“1011”（Mealy，可重叠）

- 每个状态记住已匹配的最长前缀，检测成功后跳回可复用的后缀状态



现态	x = 0	x = 1
S0	S0 / 0	S1 / 0
S1	S2 / 0	S1 / 0
S2	S0 / 0	S3 / 0
S3	S2 / 0	S1 / 1

状态表与状态图一一对应；让 LLM 同时输出两者，便于交叉核对

标注格式：输入 / 输出。S3 收到 1 后跳到 S1，因为“1011”的后缀“1”可作为下一次匹配的开头

三段式写法（SystemVerilog）

- 状态寄存、次态逻辑、输出逻辑**分开描述**：可读、可综合、便于 AI 生成与审查

seq1011.sv

```
typedef enum logic [1:0] {S0, S1, S2, S3} state_t;
state_t cs, ns;
// 第一段：状态寄存器（时序）
always_ff @(posedge clk or negedge rst_n)
  if (!rst_n) cs <= S0;
  else        cs <= ns;
// 第二段：次态逻辑（组合）
always_comb begin
  ns = cs;                          // 默认保持
  case (cs)
    S0: ns = x ? S1 : S0;
    S1: ns = x ? S1 : S2;
    S2: ns = x ? S3 : S0;
    S3: ns = x ? S1 : S2;
    default: ns = S0;               // 非法状态恢复
  endcase
end
// 第三段：输出逻辑（寄存输出）
always_ff @(posedge clk or negedge rst_n)
  if (!rst_n) det <= 1'b0;
  else        det <= (cs == S3) && x;
```

1. **第一段 状态寄存器**

   唯一的时序块；复位值即 s_0

2. **第二段 次态逻辑**

   先写默认赋值，再写 case，杜绝锁存器

3. **第三段 输出逻辑**

   寄存输出无毛刺，但比纯 Mealy 晚一拍

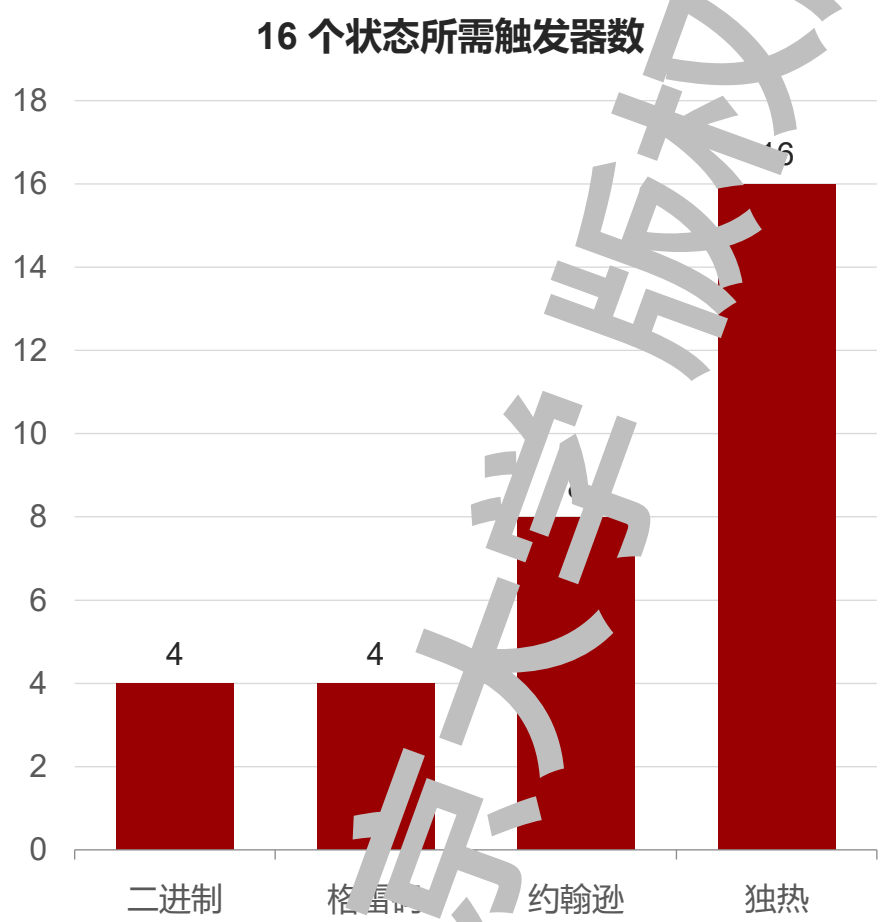
提示词中**指定写法**（三段式、always_ff / always_comb），LLM 生成的风格会稳定得多

状态编码：二进制、格雷码、约翰逊码与独热码

- 编码方式在触发器数量与次态逻辑复杂度之间权衡

编码	触发器数	次态逻辑	典型场景
二进制	$\lceil \log_2 N \rceil$	较复杂	ASIC、面积敏感
格雷码	$\lceil \log_2 N \rceil$	中等	顺序跳转、跨时钟域计数
约翰逊码	N / 2	简单	计数器、相位发生器
独热码	N	最简单、最快	FPGA（触发器资源充足）

- 综合工具可自动重编码（如 Yosys 的 fsm 流程、商用工具的 FSM 提取）
- 安全关键设计需显式处理未用码字，防止单粒子翻转后跳转



状态编码：二进制、格雷码、约翰逊码与独热码

- 三种为“少翻转”或“易译码”而设计的编码，以 8 个状态（0–7）为例对比

状态	二进制	格雷码	约翰逊码	独热码
0	000	000	0000	00000001
1	001	001	1000	00000010
2	010	011	1100	00000100
3	011	010	1110	00001000
4	100	110	1111	00010000
5	101	111	0111	00100000
6	110	101	0011	01000000
7	111	100	0001	10000000

[illegible] = 相对上一状态翻转的位（状态 0 与 7 循环比较）；N [illegible] 态数

格雷码 Gray Code

[illegible]$_2$N] 位

相邻状态只翻转 **1 位**，避免多位同时跳变带来的毛刺和跨时钟采样错误

- **转换：** G = B ^ (B >> 1) （B 为二进制）
- **应用：** 异步 FIFO 读写指针、旋转编码器

约翰逊码 Johnson Code

N/2 位

- 扭环形计数器：移位寄存器末位**取反**后反馈到首位
- 相邻只翻转 1 位，每个状态用 **2 输入** [illegible] 即可译码
- **应用：** 分频器、多相时钟；需处理[illegible]效状态自启动

独热码 One-Hot

N 位

- 任一时刻只有 **1 位为 1**，判断状态只看一位，译码最快
- 每次跳变翻转 2 位，触发器数量较多
- **应用：** FPGA 状态机（触发器资源充裕）

状态最小化：等价状态合并

- 两个状态输出相同且次态等价即可合并；用划分细化迭代到稳定

现态	x = 0	x = 1	输出
A	B	C	0
B	D	E	0
C	D	E	0
D	A	A	1
E	A	A	1

例 Moore 机：5 个状态 → 3 个状态

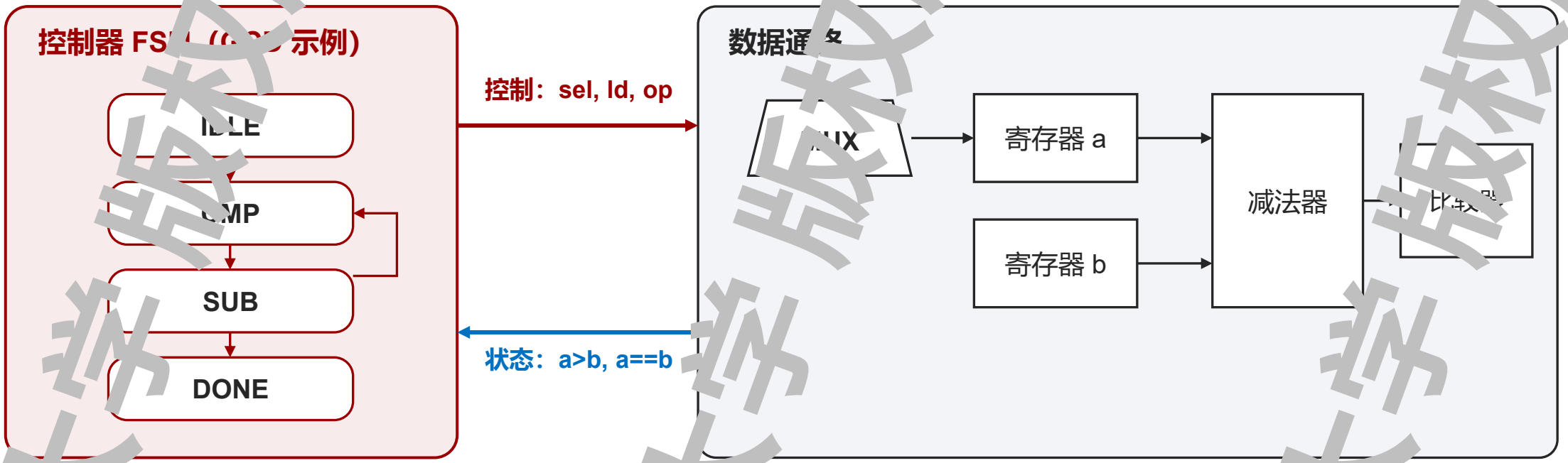
1. P_0：按输出划分
   {A, B, C}　{D, E}
2. P_1：按次态所在块细化
   {A}　{B, C}　{D, E}
3. P_2：再次检查
   与 P_1 相同 → 收敛
4. 合并结果
   A，BC，DE 共 3 个状态

要求 LLM 输出每一步划分而不只给结论，便于逐步核查

思想：包含 兼容并包

层次化状态机与 FSMD

- 复杂控制 = 控制器 FSM + 数据通路：FSM 发控制信号，数据通路回传状态标志



层次化状态机
父状态包含子状态机（Statechart），控制复杂协议

FSMD
多层级算法：乘除法器、GCD、DMA 控制

与 HLS 的关系
高层次综合自动生成的就是 FSMD 结构

AI 实践：从自然语言规格到状态机

- 先让 LLM 产生**结构化中间表示**，再生成代码：每一步都可被机器检查

自然语言规[illegible] → 抽取六元组（JSON） → 图检查与[illegible] → 生成 Verilog → 仿真 +[illegible]

输入规格（[illegible]）

主路默认绿灯；支路检测到车辆（car=1）时，主路转黄灯，保持 3 拍后变红，支路绿灯 5 拍，再黄灯 3 拍后恢复主路绿灯。

复位后主路绿灯。

LLM 输[illegible]表示 fsm.json

```
{
  "state[illegible]": ["MAIN_G","MAIN_Y","SIDE_G","SIDE_Y"],
  "reset":  "MAIN_G",
  "transitions": [
    {"from":"MAIN_G","to":"MAIN_Y","cond":"[illegible]"},
    {"from":"MAIN_Y","to":"SIDE_G","cond":"cnt==2"},
    {"from":"SIDE_G","to":"SIDE_Y","cond":"cnt==4"},
    {"from":"SIDE_Y","to":"MAIN_G","cond":"cnt==2"}
  ],
  "counter": "cnt 在每次状态切换时清零"
}
```

为什么需要状态机

- 控制电路的基本

状态机实例1 – 控制一个红绿灯



- 仅考虑红灯和绿灯，灯转换的速度不快于每次30s（0.033 Hz）
- 2个输出
  - NSlight：1=南北向为绿灯；0=南北向红灯
  - EWlight：1=东西向为绿灯；0=东西向红灯
- 2个输入
  - Nscar：1=南北向有车等；0=南北向无车等
  - Ewcar：1=东西向有车等；0=南北向无车等
- 规则
  - 交通灯切换到另一个方向当且仅当另一方向有车等
  - 否则，保持当前交通灯不变

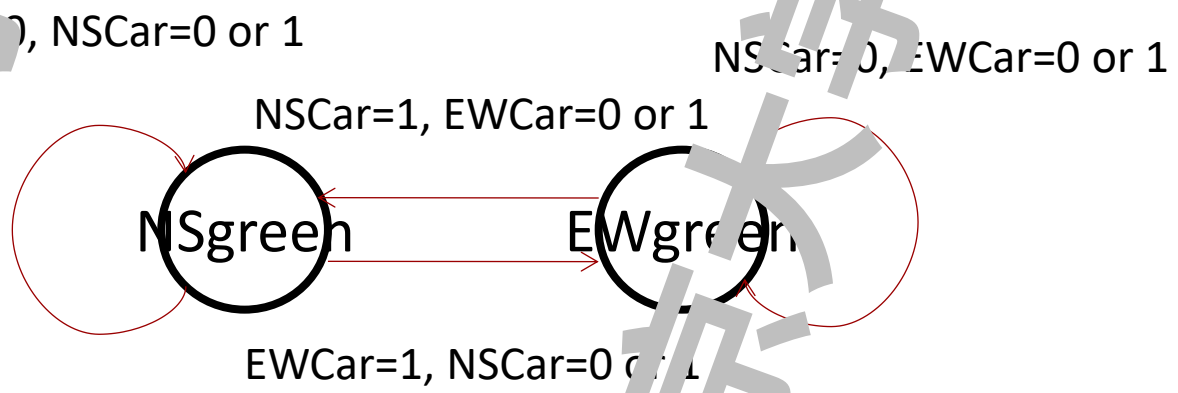
为什么需要状态机

- 控制电路的基本

状态机实例1 – 控制一个红绿灯

- 2个输出
  - NSlight：1=南北向为绿灯；0=南北向红灯
  - EWlight：1=东西向为绿灯；0=东西向为红灯
- 2个输入
  - Nscar：1=南北向有车等；0=南北向无车等
  - Ewcar：1=东西向有车等；0=南北向无车等
- 规则
  - 交通灯切换到另一个方向当且仅当另一方向有车等
  - 否则，保持当前交通灯不变

- 需要2个状态
  - Nsgreen、EWgreen



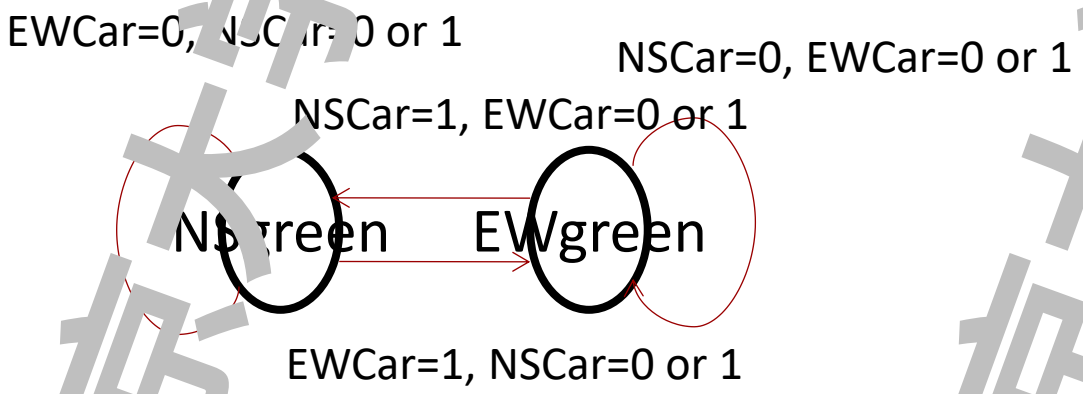
为什么需要状态机

- 控制电路的基本

状态机实例1 – 控制一个红绿灯

- 需要2个状态
  - NSgreen、EWgreen

Current state	Inputs		Next state
NSgreen	0	0	NSgreen
NSgreen	0	1	NSgreen
NSgreen	1	0	NSgreen
NSgreen	1	1	EWgreen
EWgreen	0	0	EWgreen
EWgreen	0	1	EWgreen
EWgreen	1	0	NSgreen
EWgreen	1	1	NSgreen



Current state	Outputs	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1

$$\text{NextState} = (\overline{\text{CurrentState}} \cdot \text{EWcar}) + (\text{CurrentState} \cdot \overline{\text{NScar}})$$

$$\text{NSlite} = \overline{\text{CurrentState}}$$

$$\text{EWlite} = \text{CurrentState}$$

为什么需要状态机

- 控制电路的基本

状态机实例1 – 控制一个红绿灯

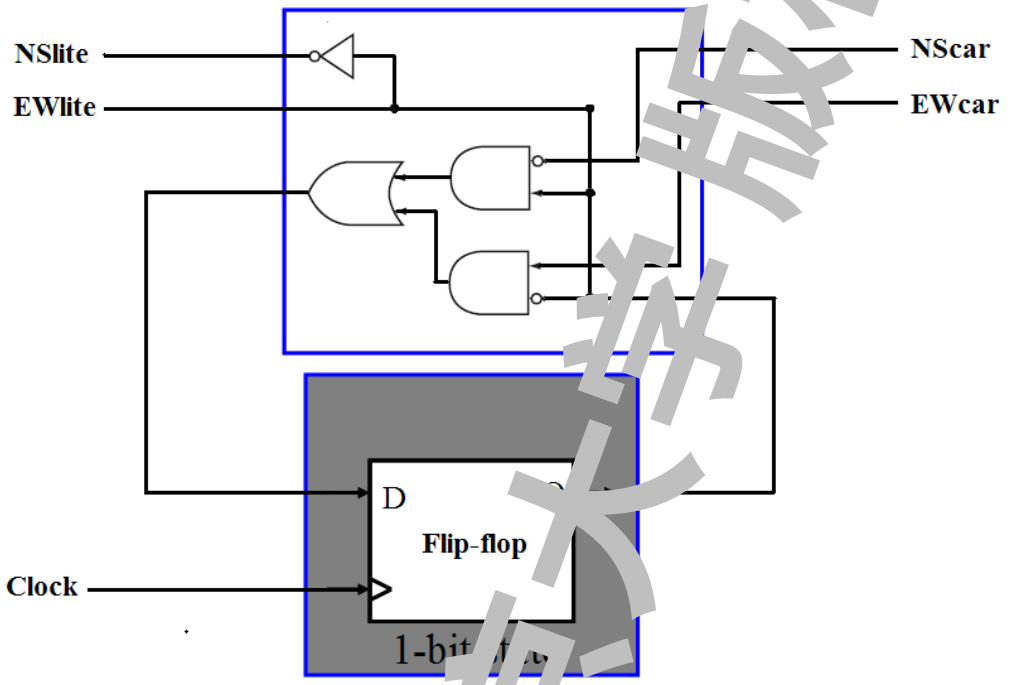
Current state	Inputs		Next state
	NScar	EWcar	
NSgreen	0	0	NSgreen
NSgreen	0	1	EWgreen
NSgreen	1	0	NSgreen
NSgreen	1	1	EWgreen
EWgreen	0	0	EWgreen
EWgreen	0	1	EWgreen
EWgreen	1	0	NSgreen
EWgreen	1	1	NSgreen

Current state	Outputs	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1

$$NextState = (\overline{CurrentState} \cdot EWcar) + (CurrentState \cdot \overline{NScar})$$

$$NSlite = \overline{CurrentState}$$

$$EWlite = CurrentState$$



2.9 用 LLM + 工具自动审查 FSM

- LLM 负责发现可疑点并生成检查，工具负责给出确定结论

1 死锁状态

进入后再也无法离开的非终止状态

检查：图搜索出边 / SVA 活性断言

2 不可达状态

从复位状态出发永远到不了

检查：BFS 可达性；形式化 cover

3 未覆盖的转移

某状态下的输入组合没有定义

检查：枚举条件是否互斥且完备

4 非法状态恢复

编码空间中未使用的码字

检查：case default；故障注入仿真

提示词示例：请阅读下面的 FSM 代码，列出 (1) 不可达状态 (2) 死锁状态 (3) 未定义的转移条件，对每一项给出一条可以在 SymbiYosys 中运行的 SVA cover/assert 语句，不要直接修改代码。

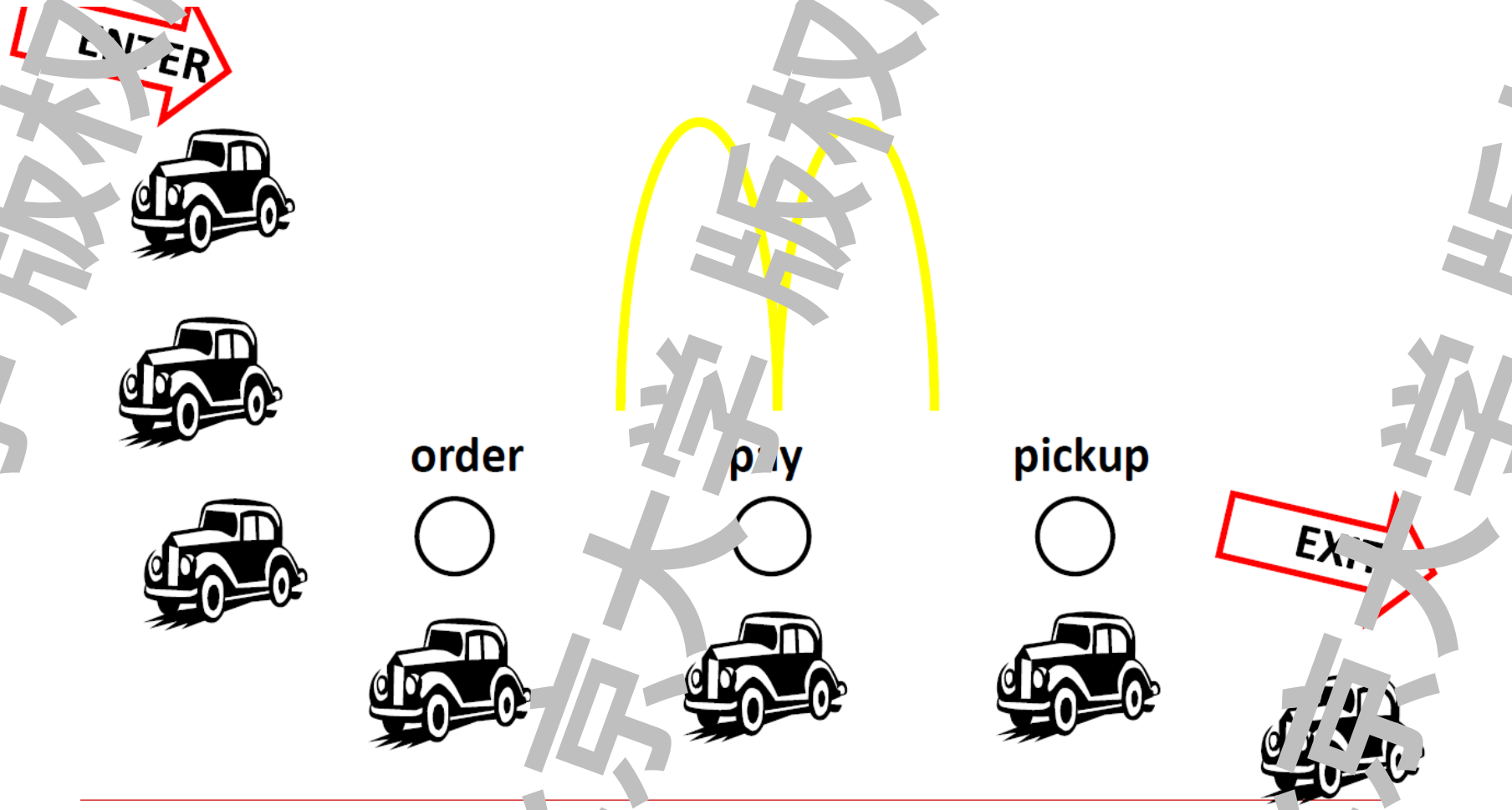
北京大学
PEKING UNIVERSITY

AI辅助时序逻辑设计 – 流水线

北京大学版权所有

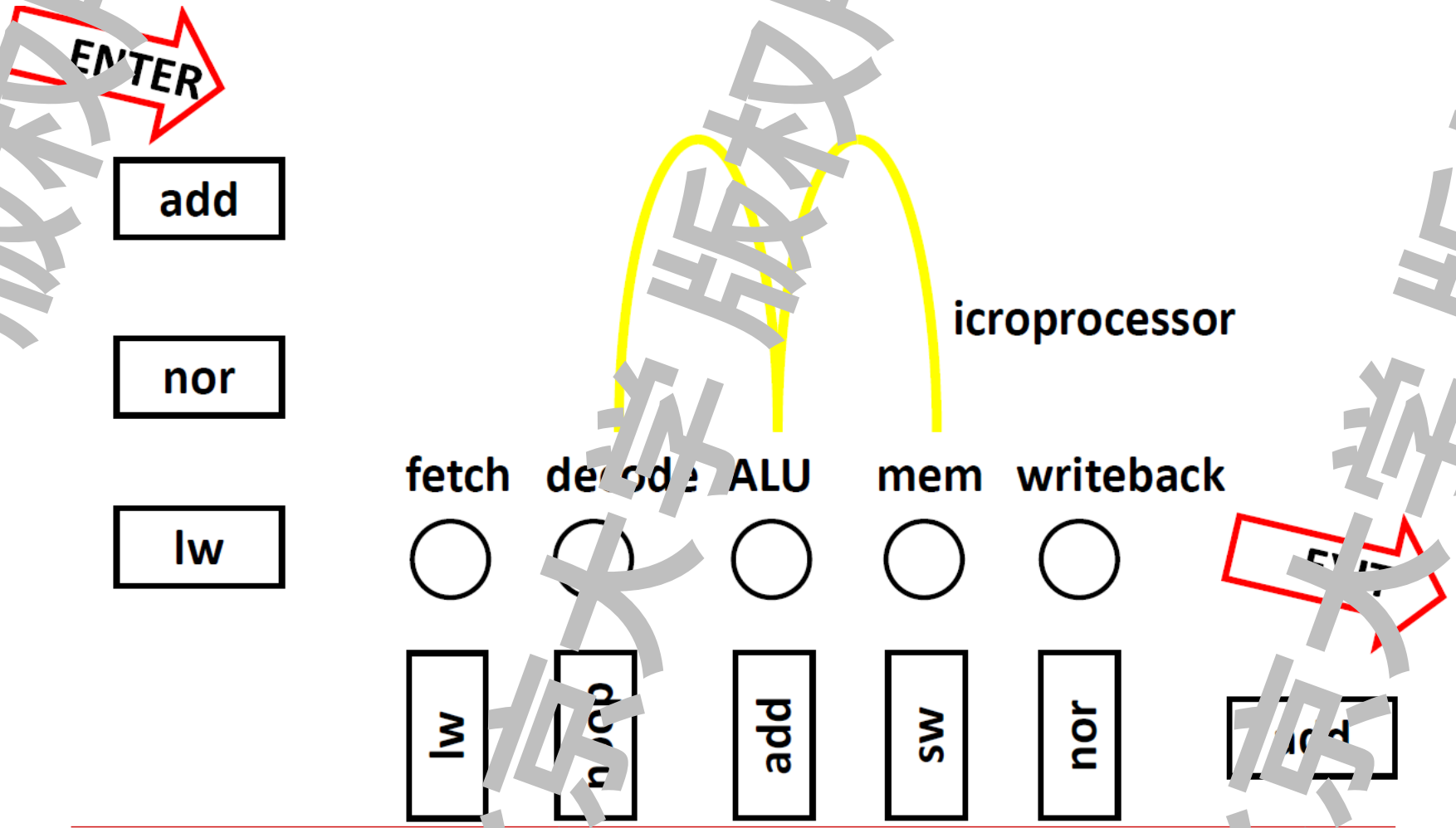
回顾：什么是流水线架构

- 流水线式运行方式 - 提高吞吐率的有效手段



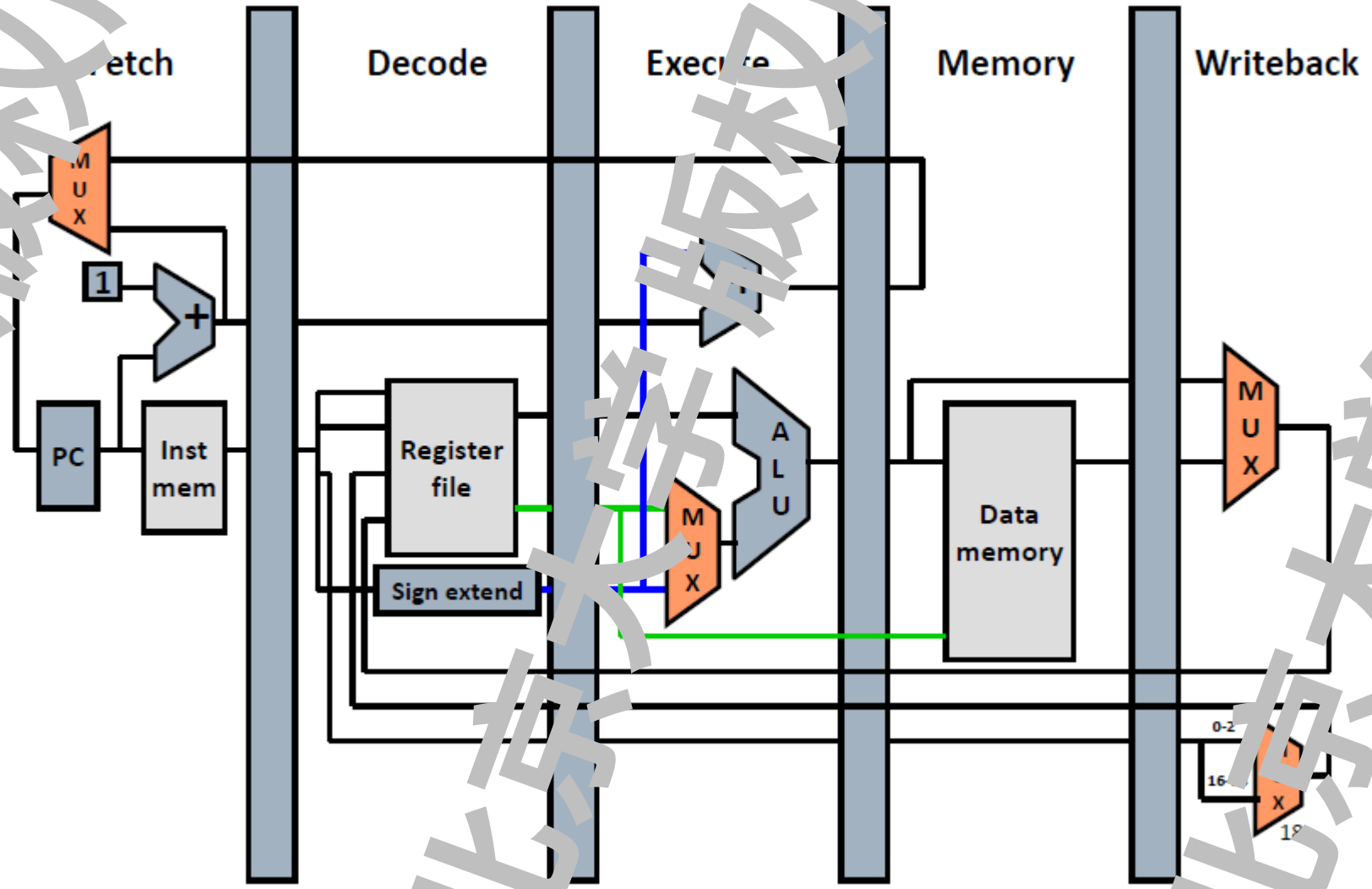
多级流水线架构

- 流水线式运行方式 - 提高吞吐率的有效手段（提高instruction/cycle，CPI）



最基本的单条流水线设计示意图

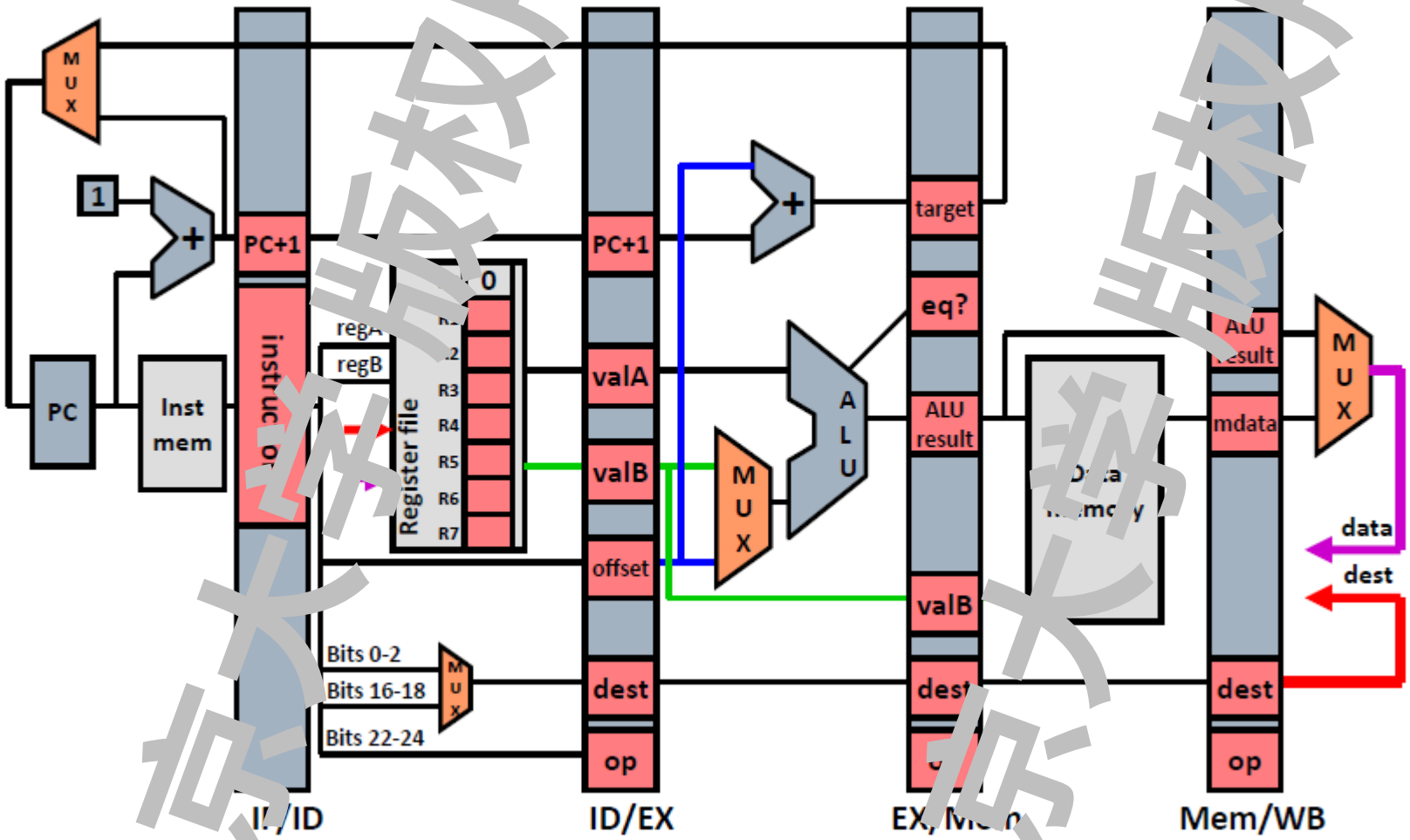
- 5级流水线设计：Fetch、Decode、Execute、Memory、Writeback



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

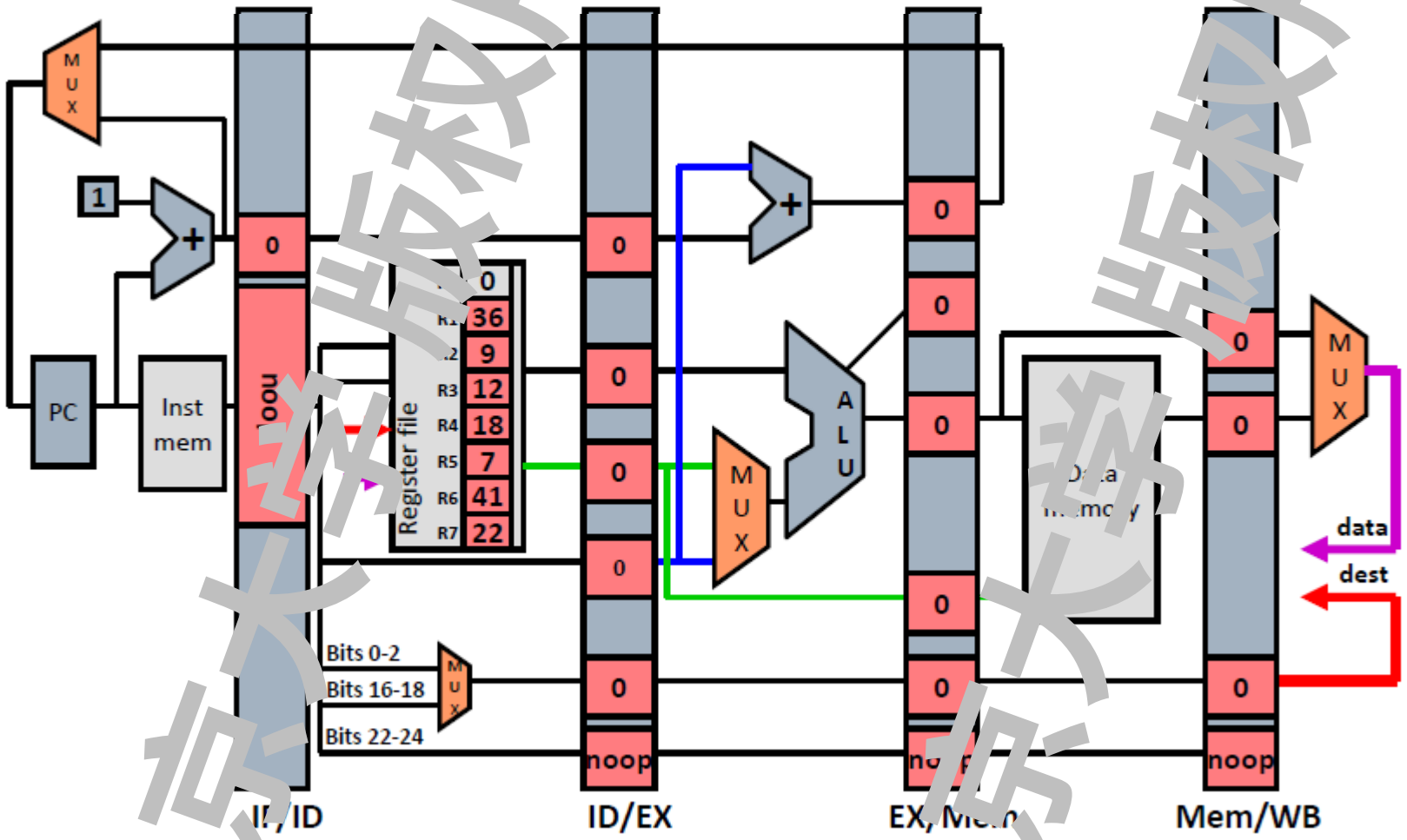
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10] =reg 7



5级流水线的实现实例

- 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

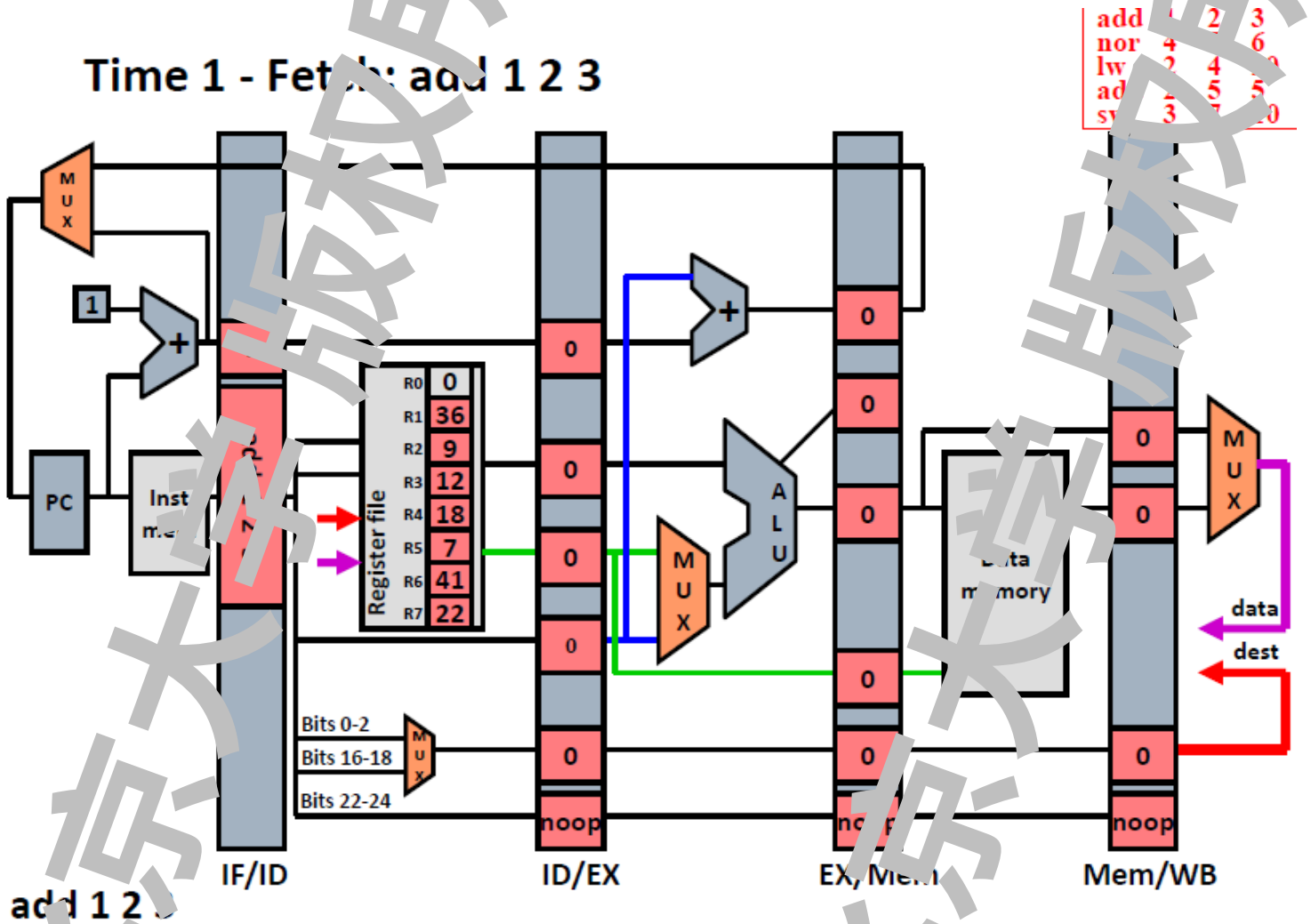


初始状态：t0

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

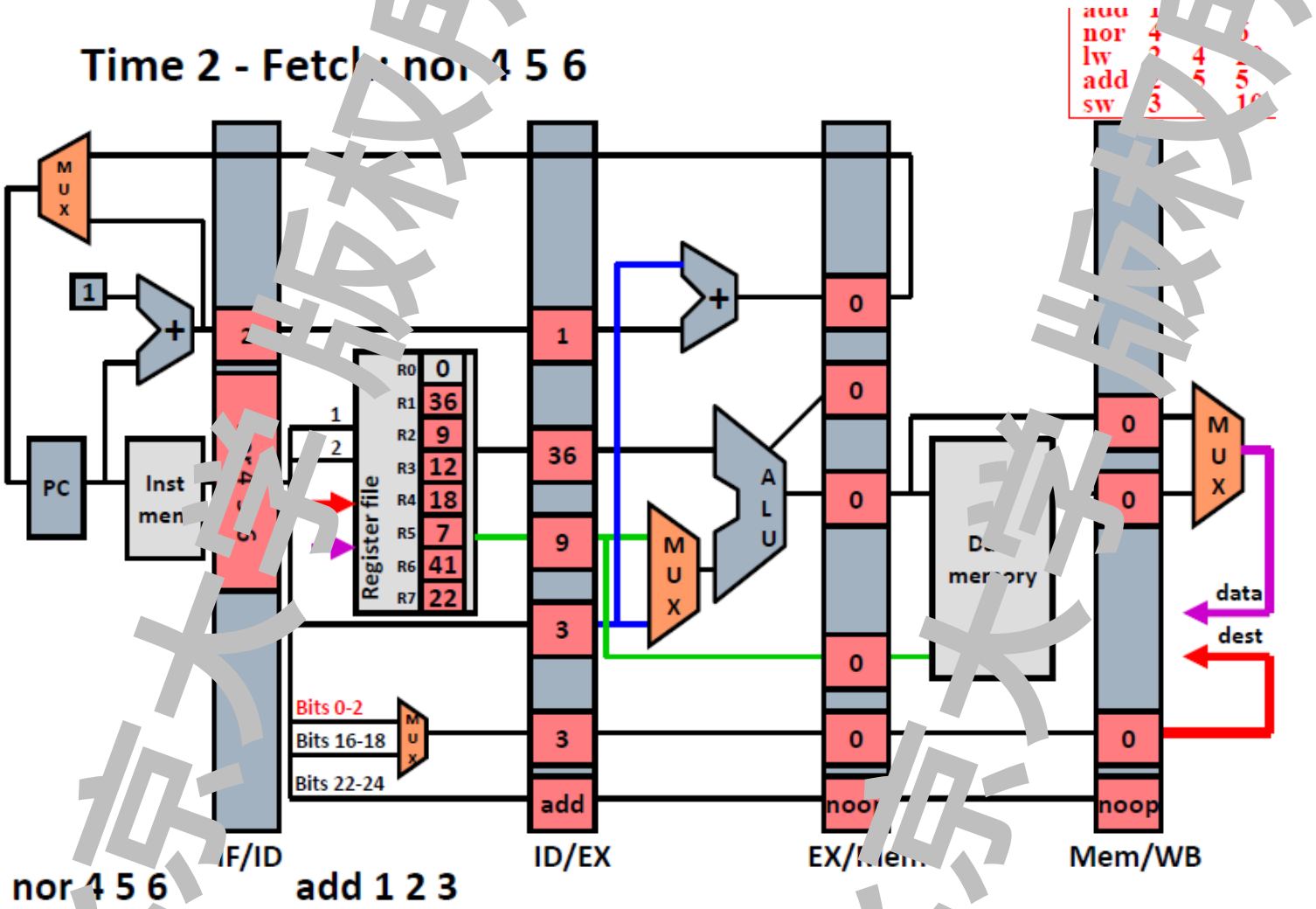
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10] =reg 7



5级流水线的实现实例

- 假设运行以下指令在5级流水线上

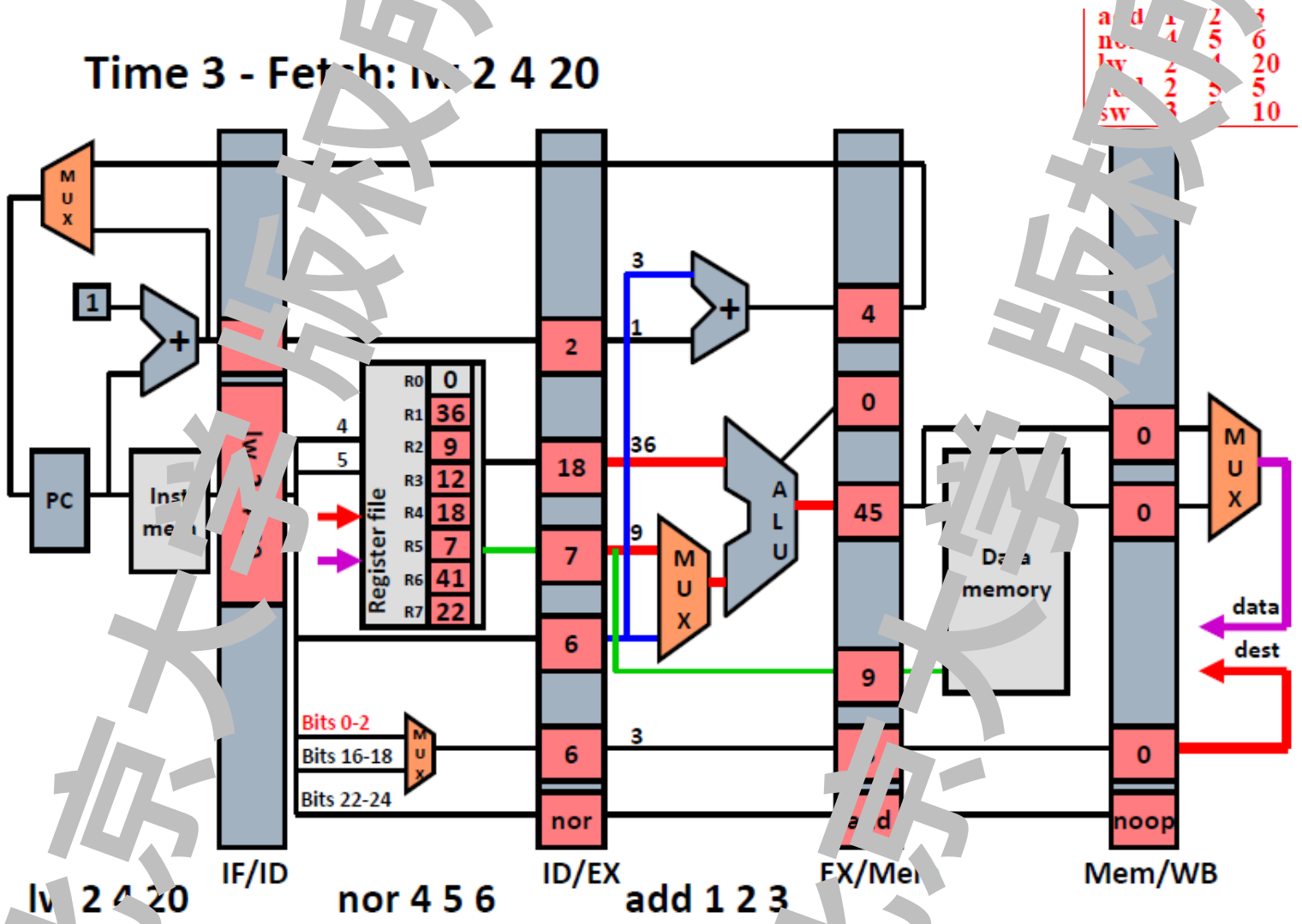
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10] =reg 7



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

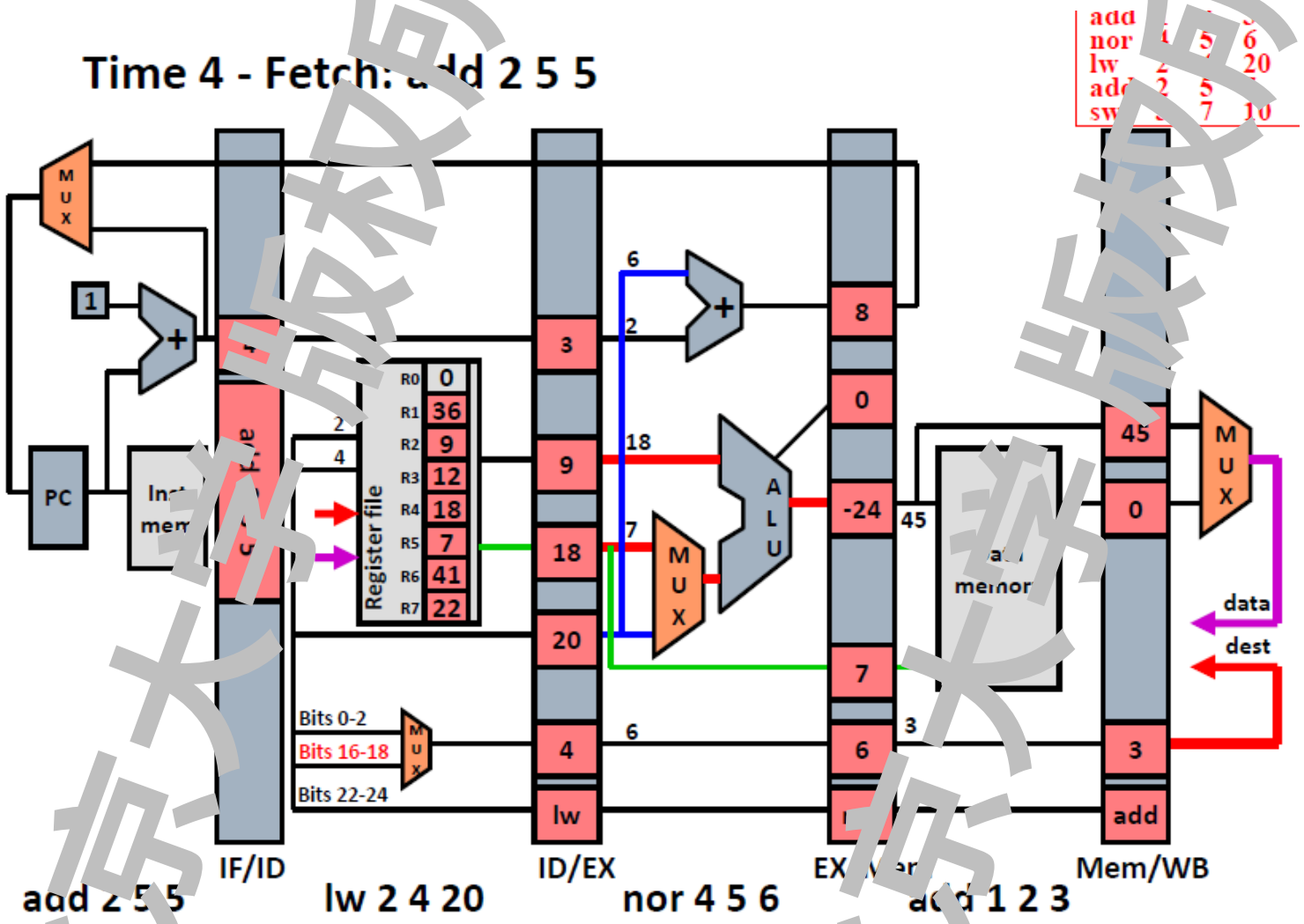
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10] =reg 7



5级流水线的实现举例

- 假设运行以下指令在5级流水线上

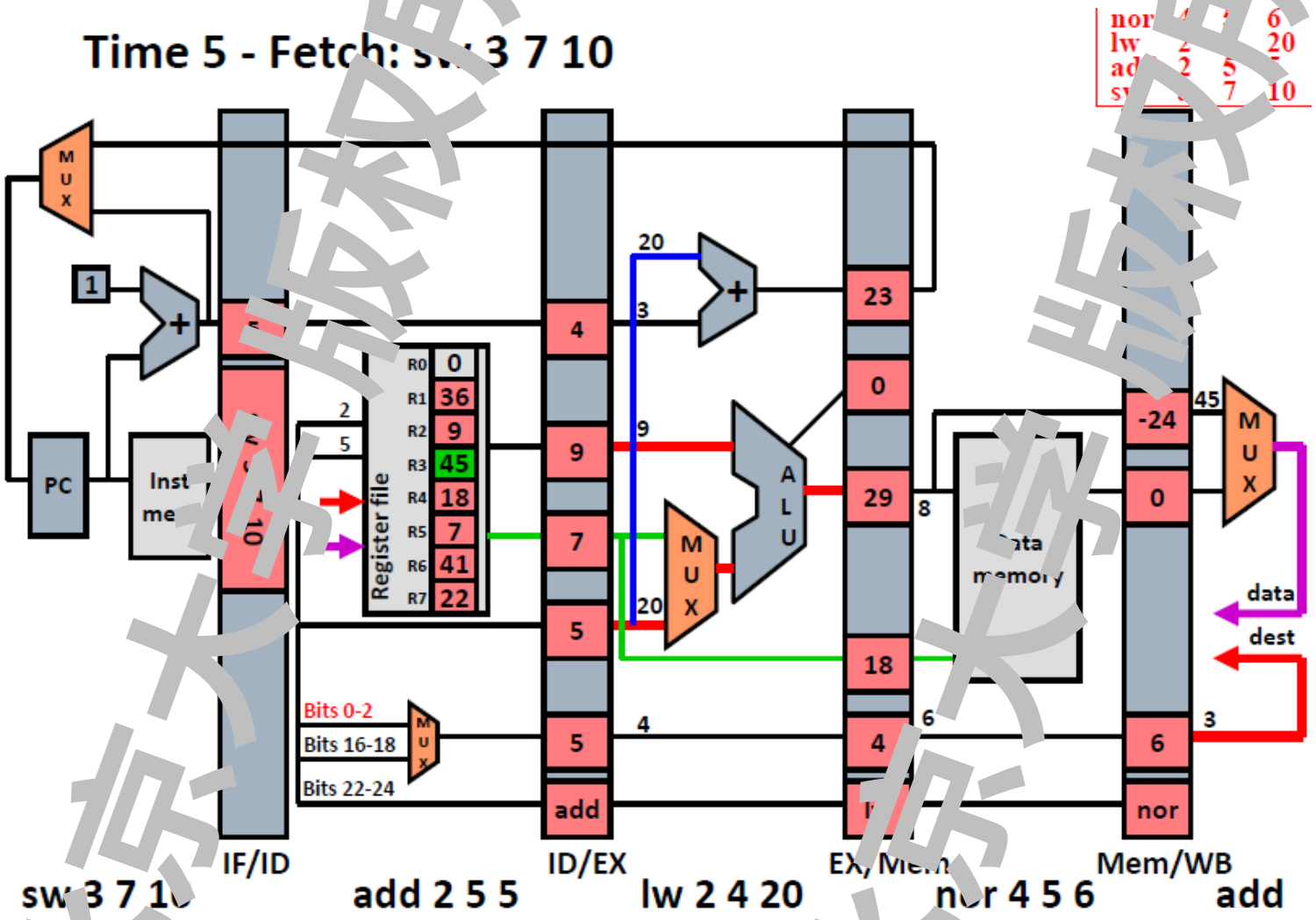
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10] =reg 7



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

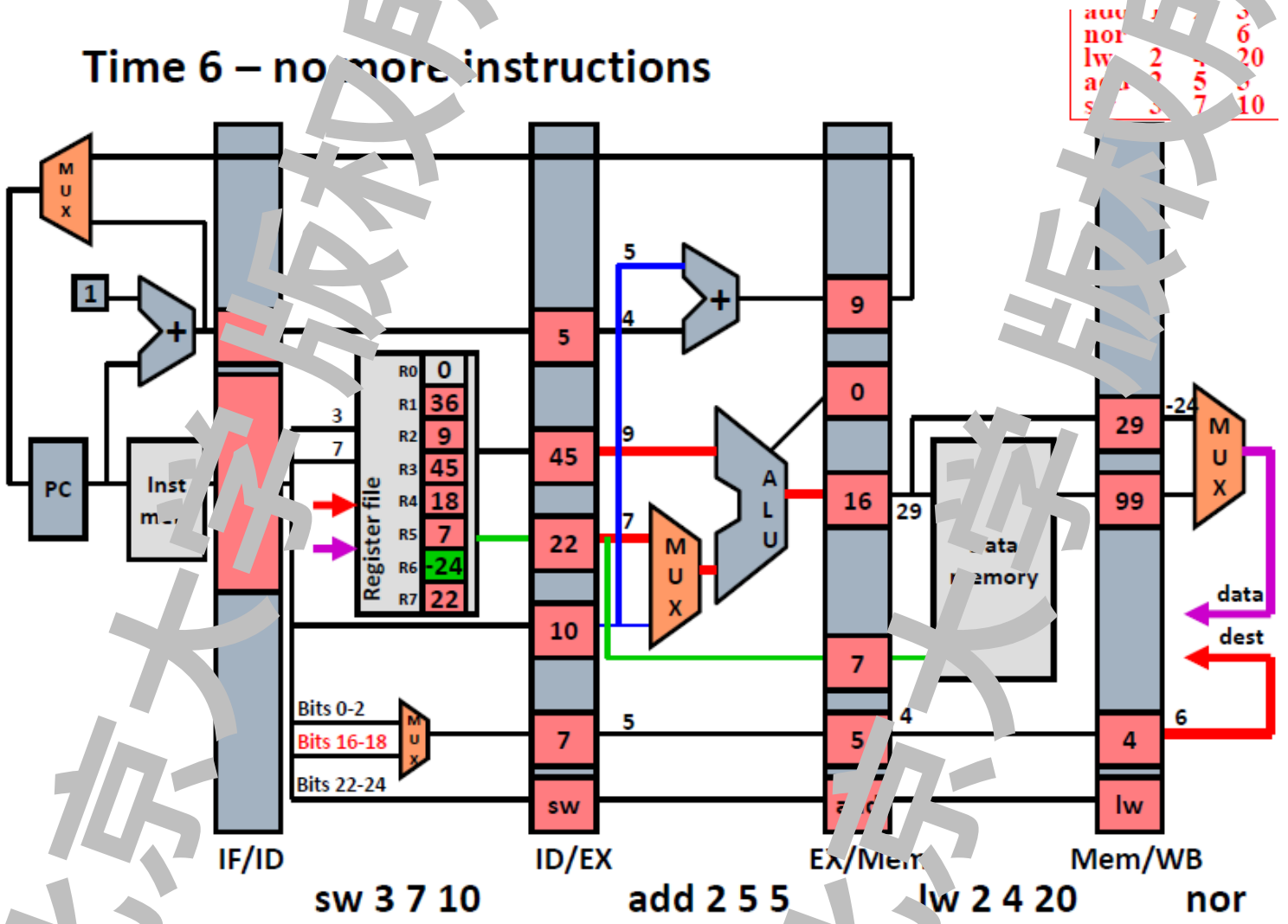
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10] =reg 7



5级流水线的实现实例

- 假设运行以下指令在5级流水线上

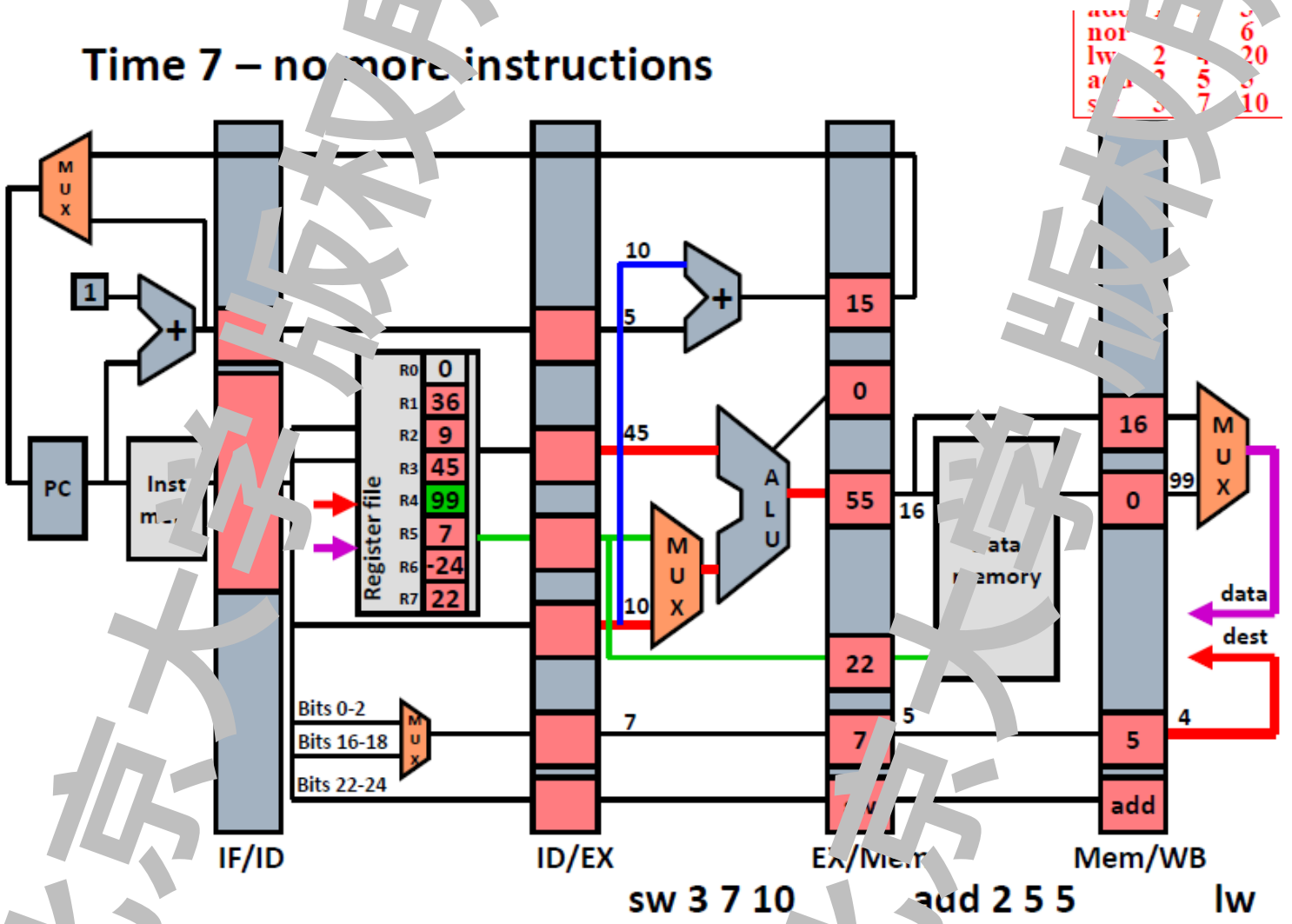
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10] =reg 7



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

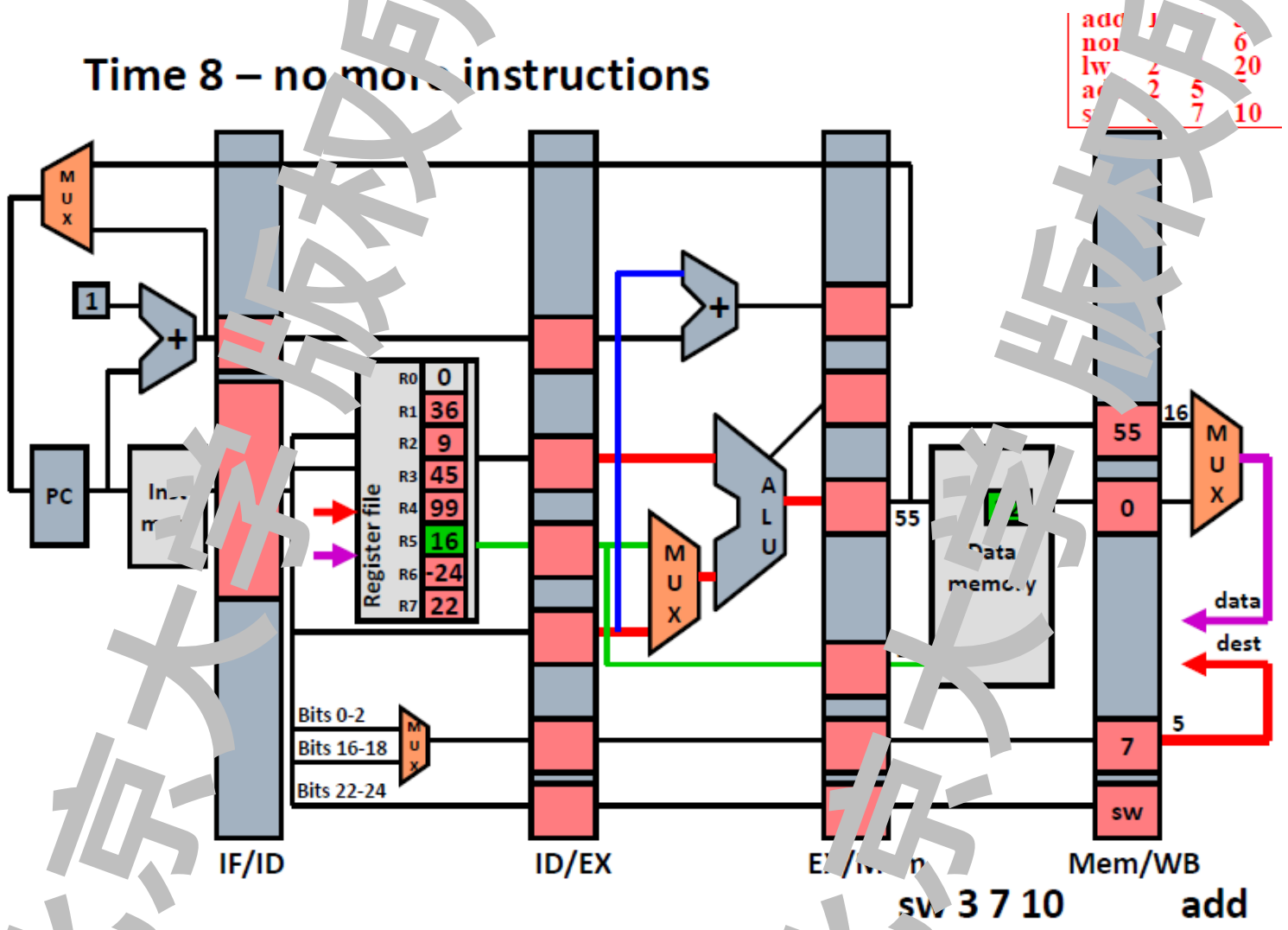
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10] =reg 7



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

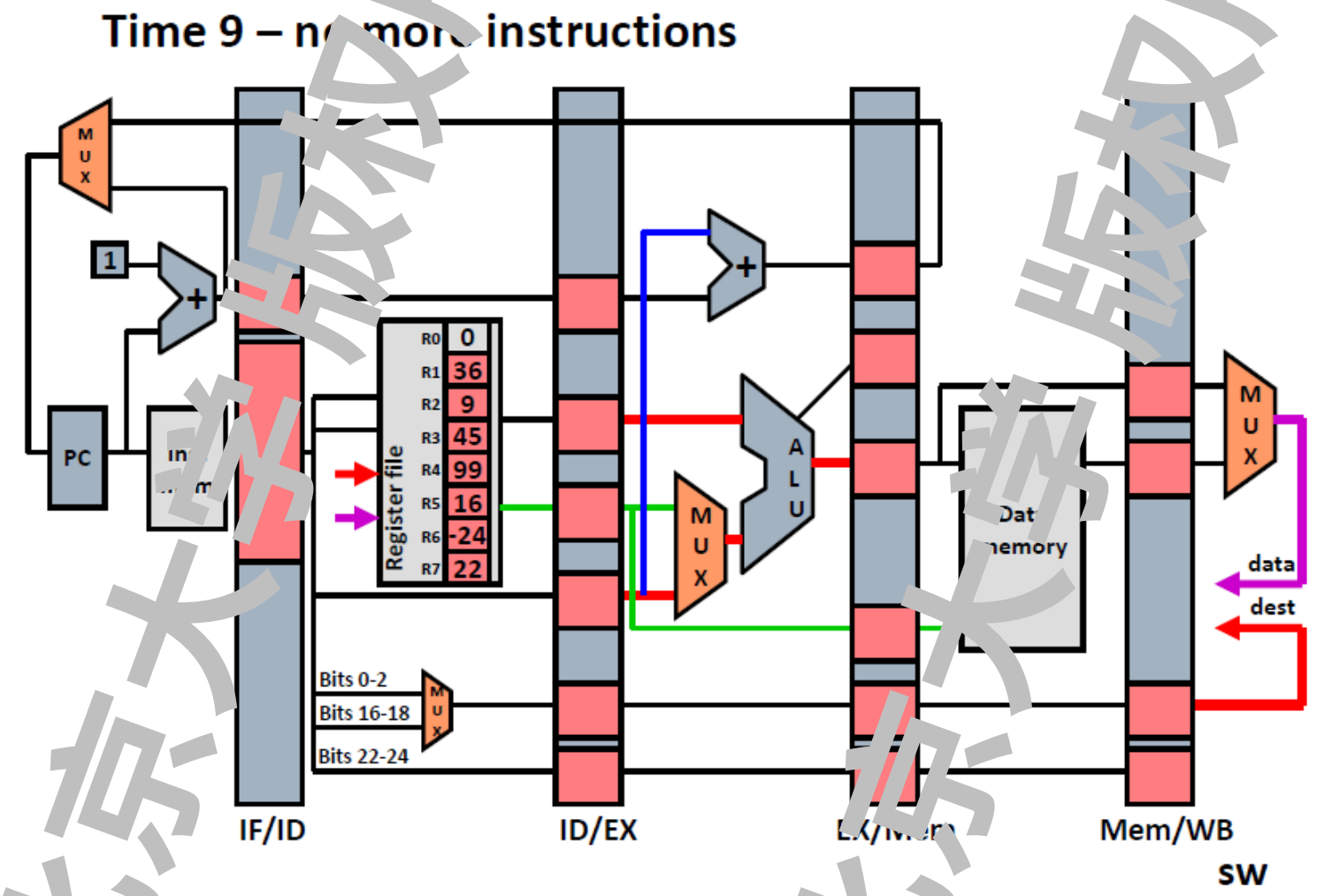
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10] =reg 7



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

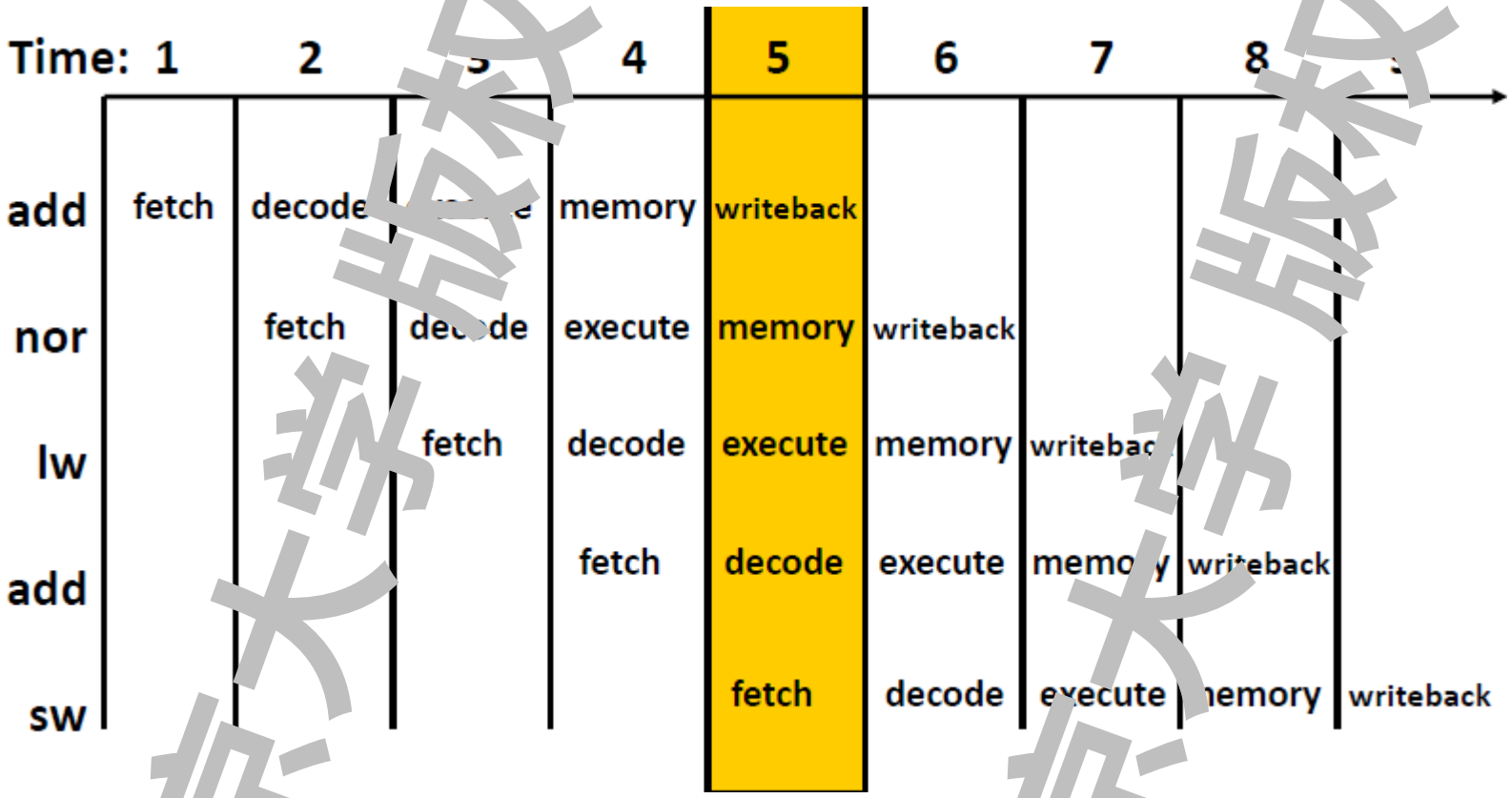
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10] =reg 7



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
```



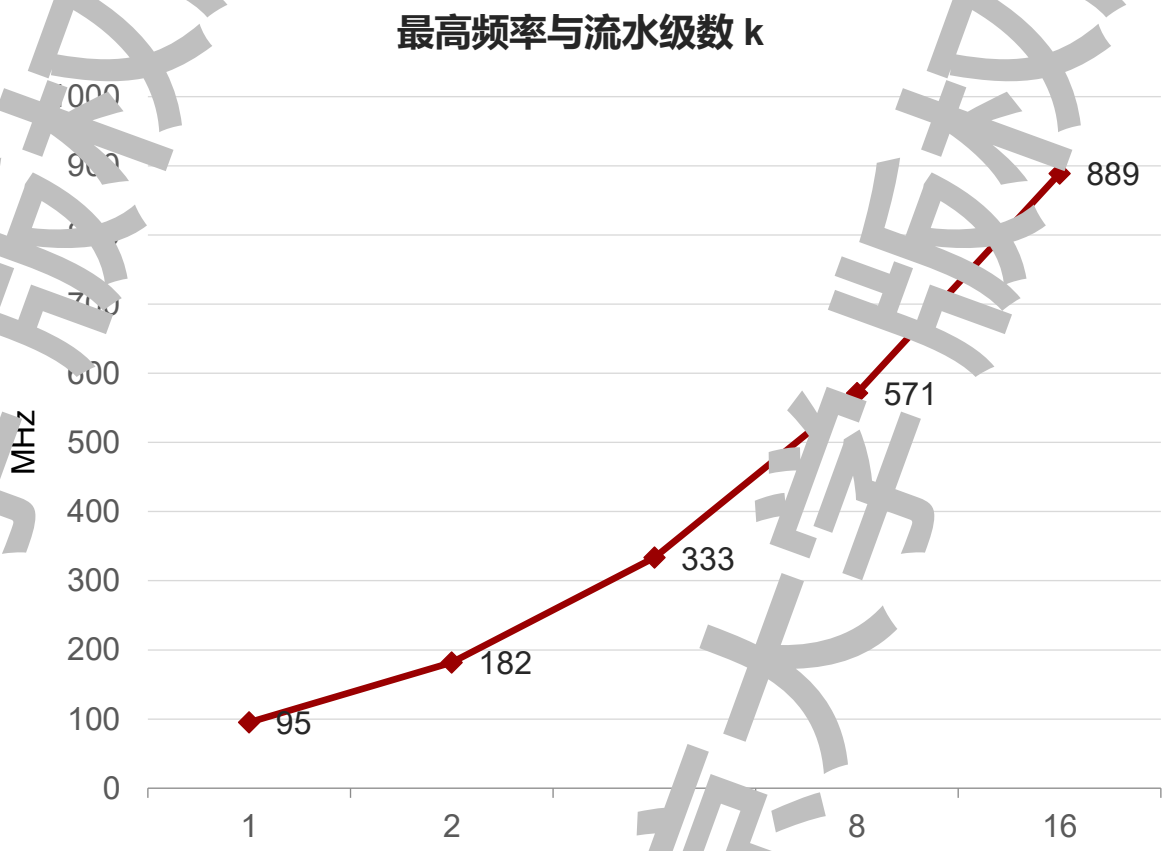
为什么需要做这样的多级流水线

- 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

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

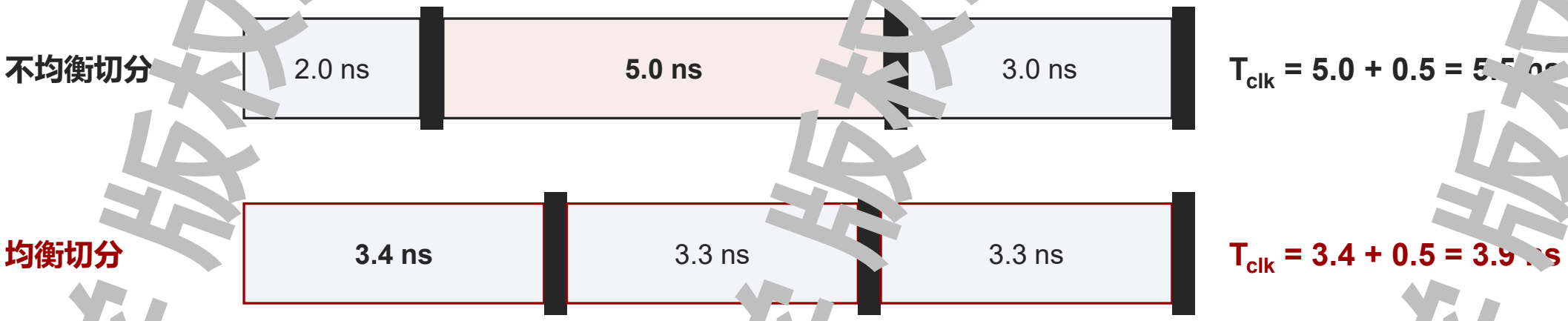
- **吞吐率**：理想情况下**每拍输出一个结果，约提升 k 倍**
- **延迟**：$k \times T_{clk}$，随级数增加而变长
- **收益递减**：寄存器开销与时钟偏斜不随 k 缩小
- **代价**：寄存器面积、功耗、控制复杂度上升



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 随数据向后传播，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

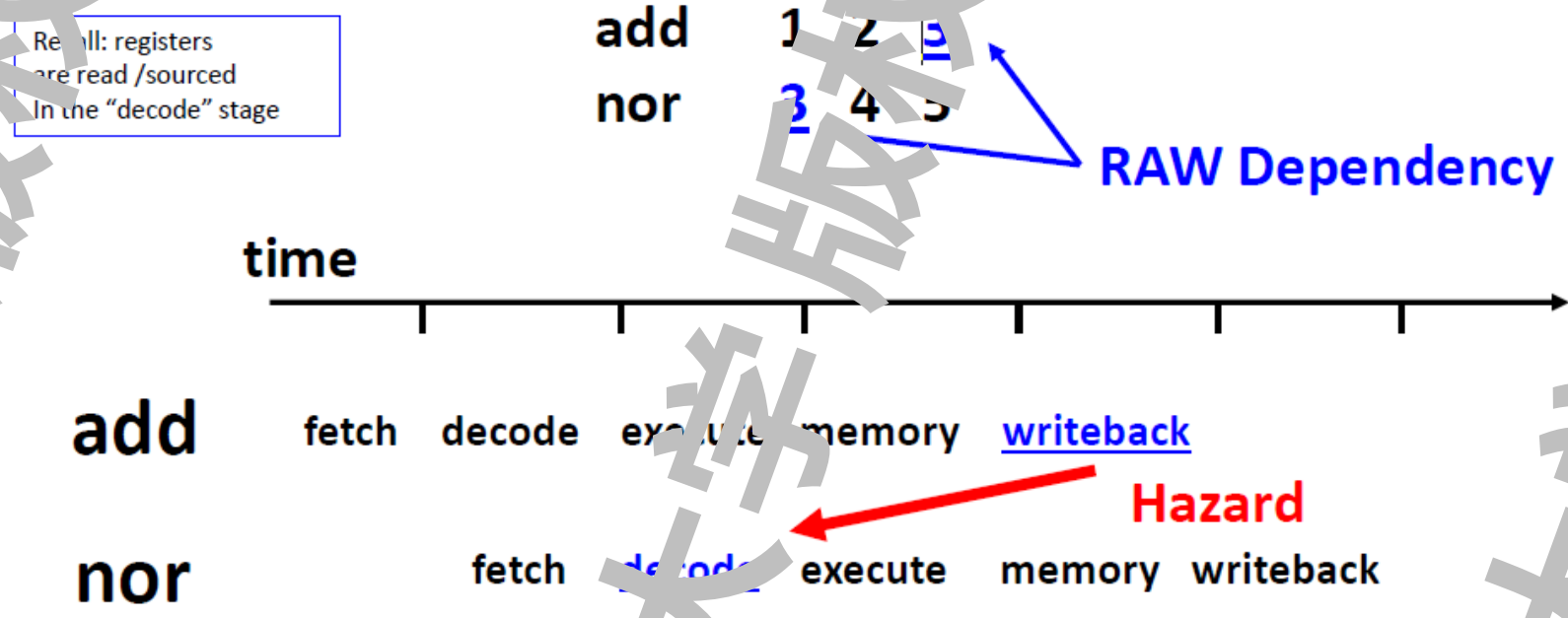
- 每级多放一个暂存槽
- ready 改为寄存输出，切断反向组合链
- 代价：每级多一份数据寄存器

简单5级流水线A仍存在什么问题?

- **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

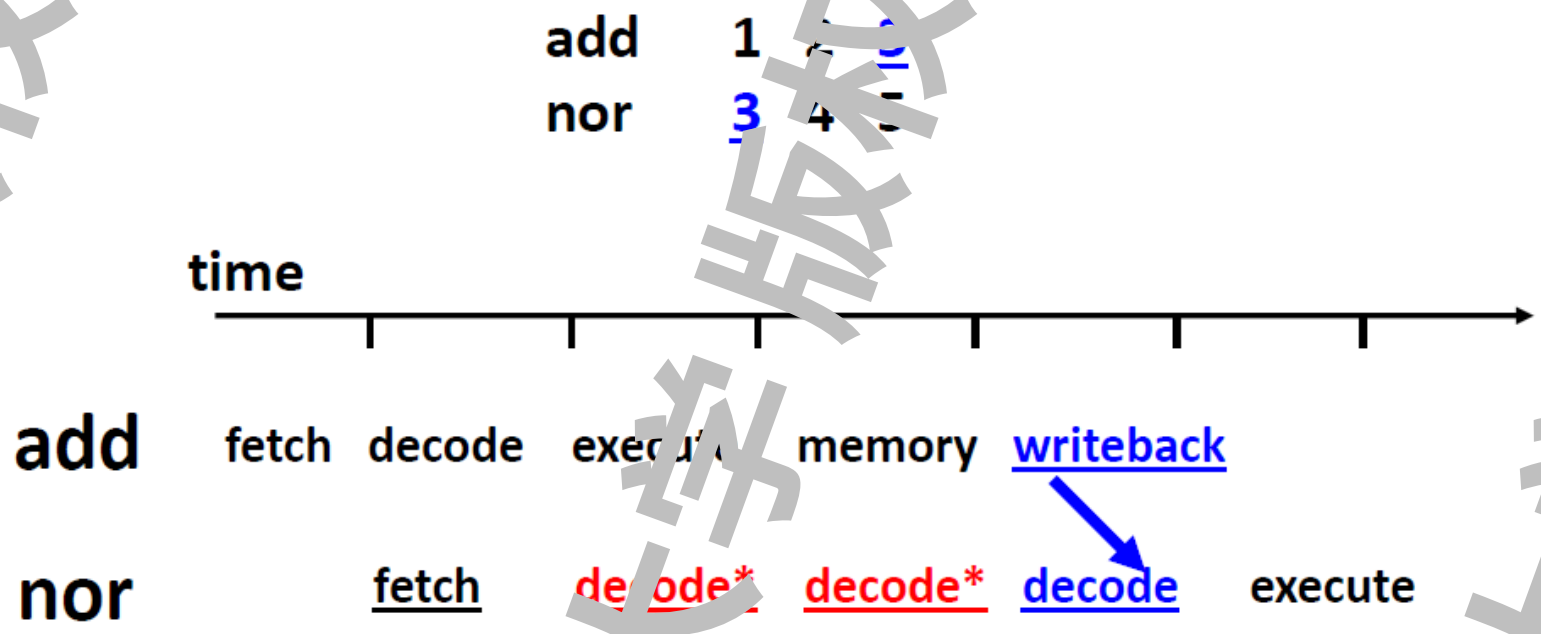
- RAW问题：Read After Write数据冲突



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

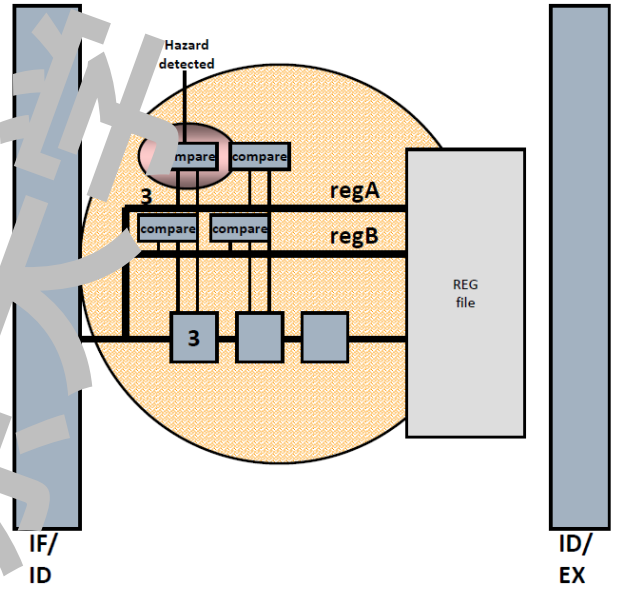


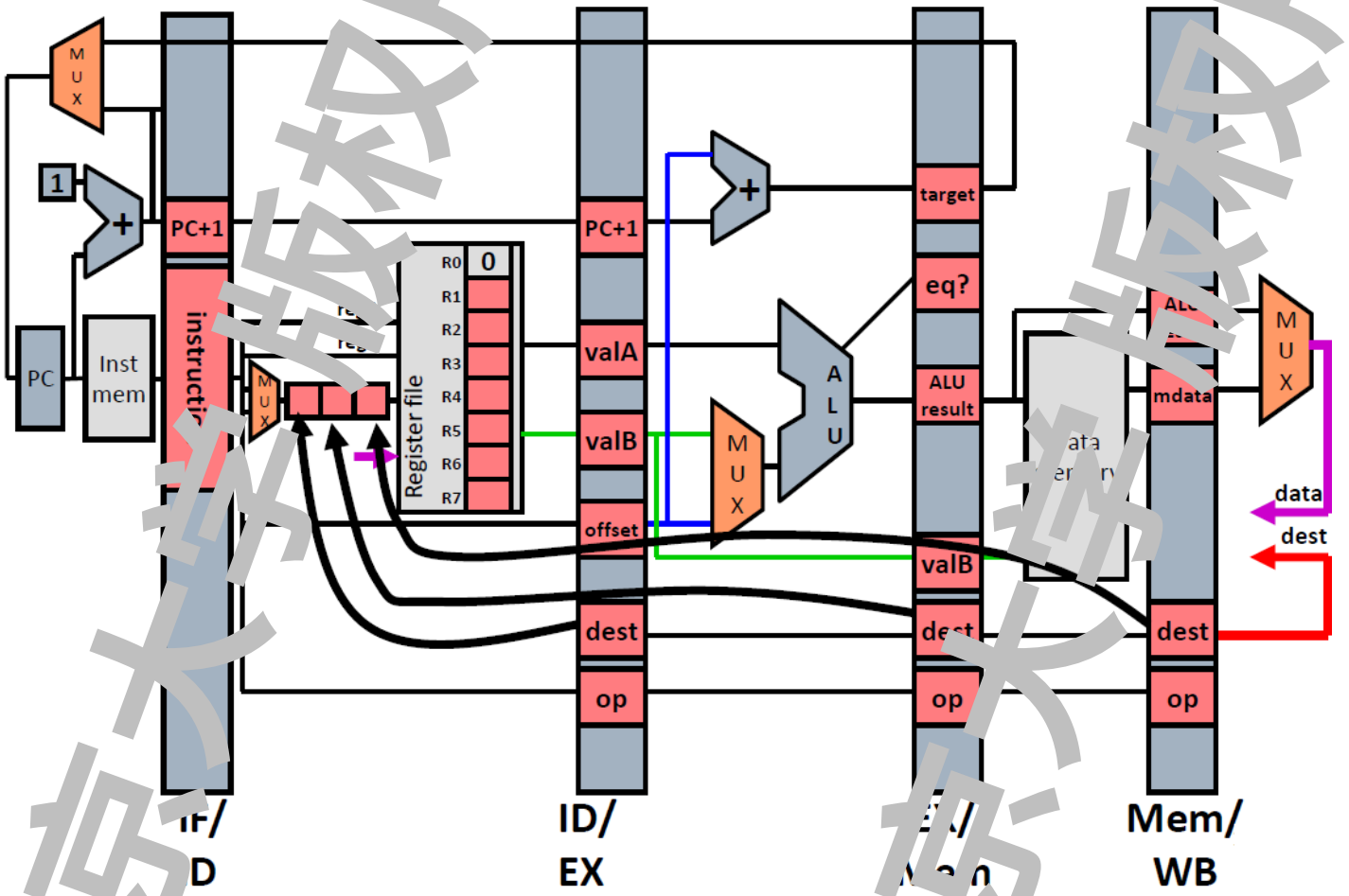
简单解决办法：流水线停顿（Pipeline Stall）

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

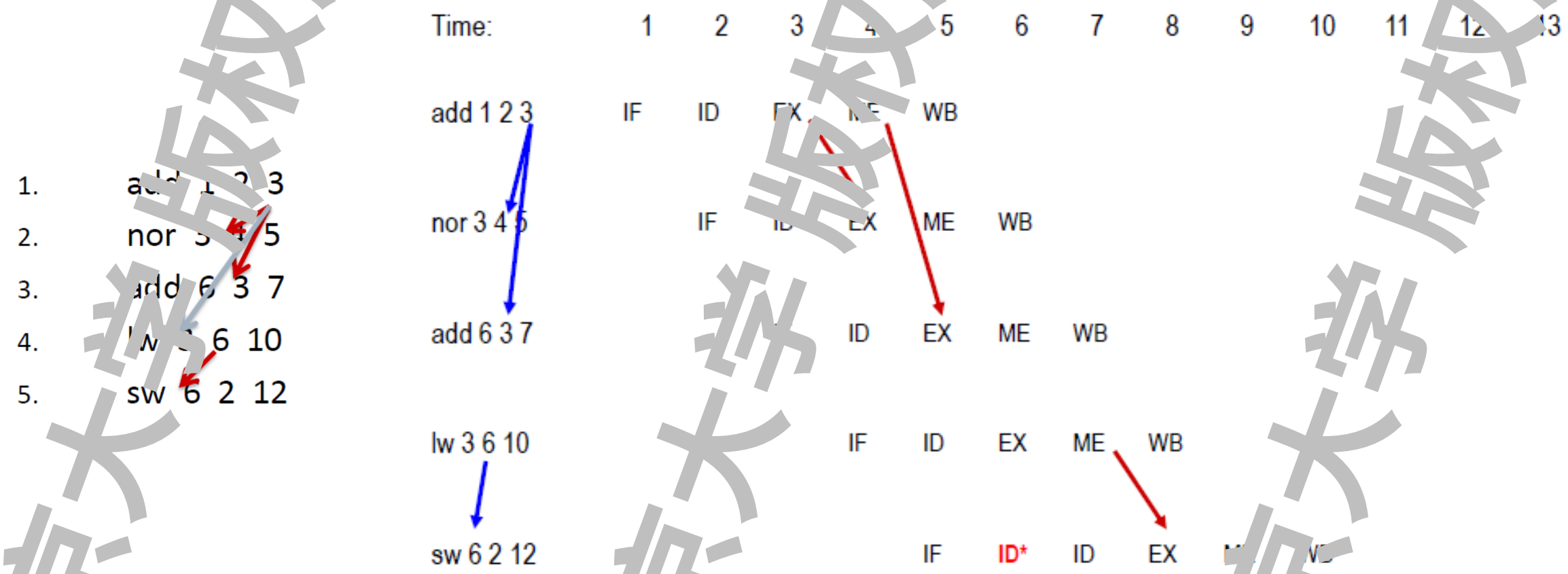




进阶解决办法：Detect and Forward

问题1：Data Hazards

- RAW问题：Read After Write数据冲突



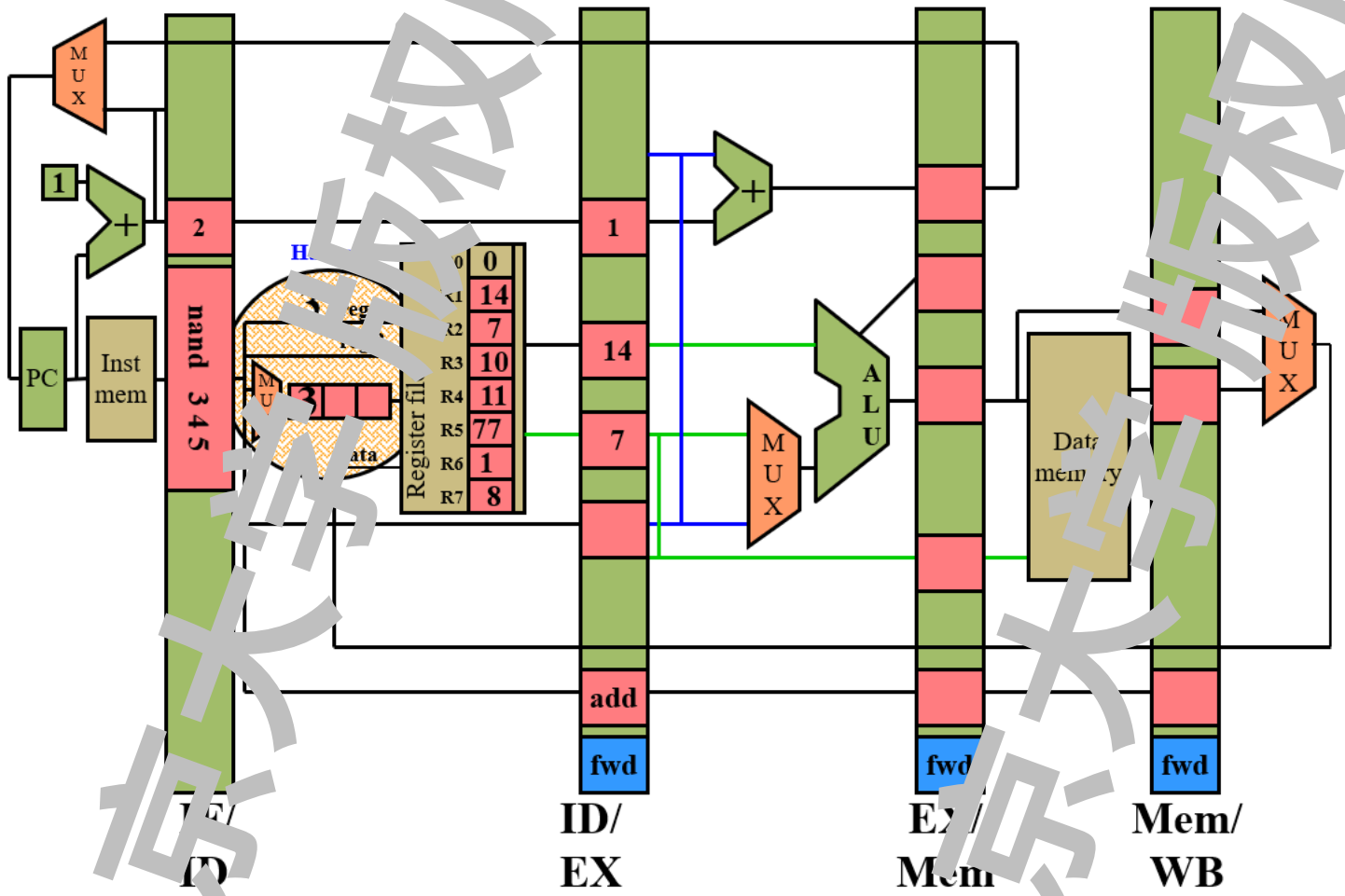
进阶解决办法：Detect and Forward

问题1：Data Hazards

- Detect and forward案例

```
add 1 2 3     // 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 3前半段



问题1：Data Hazards

- Detect and forward案例

add 1 2 3 // r3 = r1 + r2

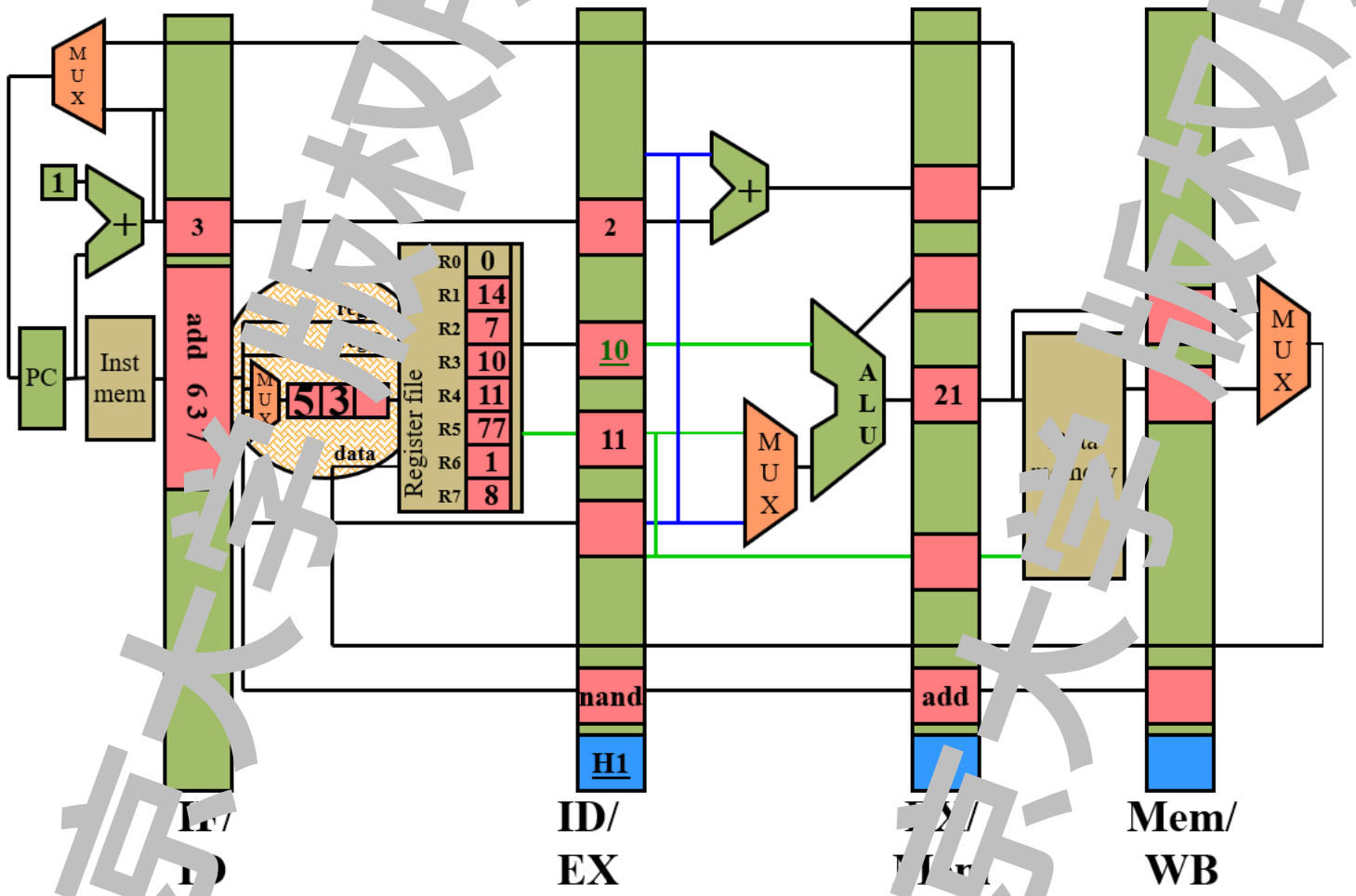
nand 3 4 5 // r5 = r3 NAND r4

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

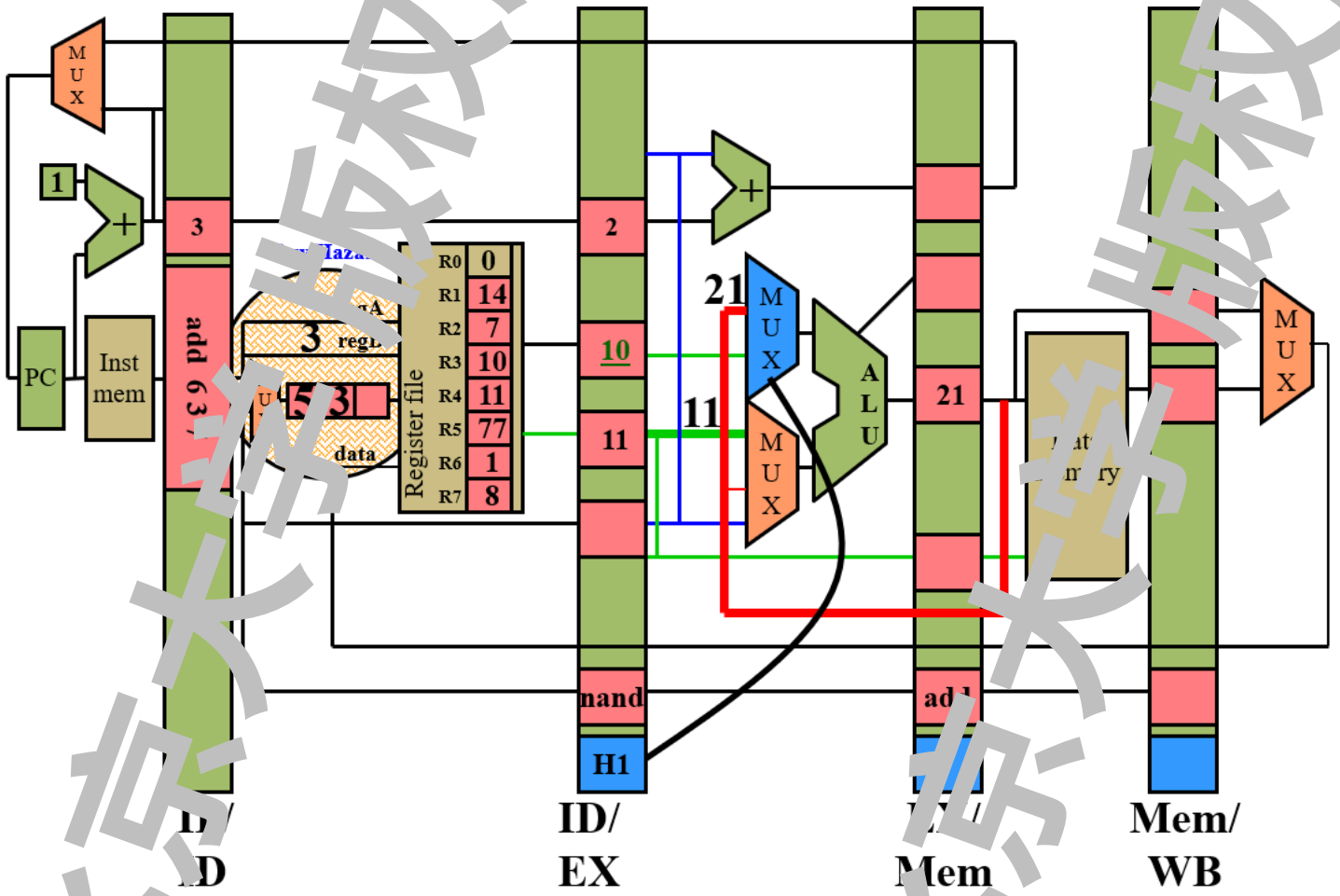


问题1：Data Hazards

- 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

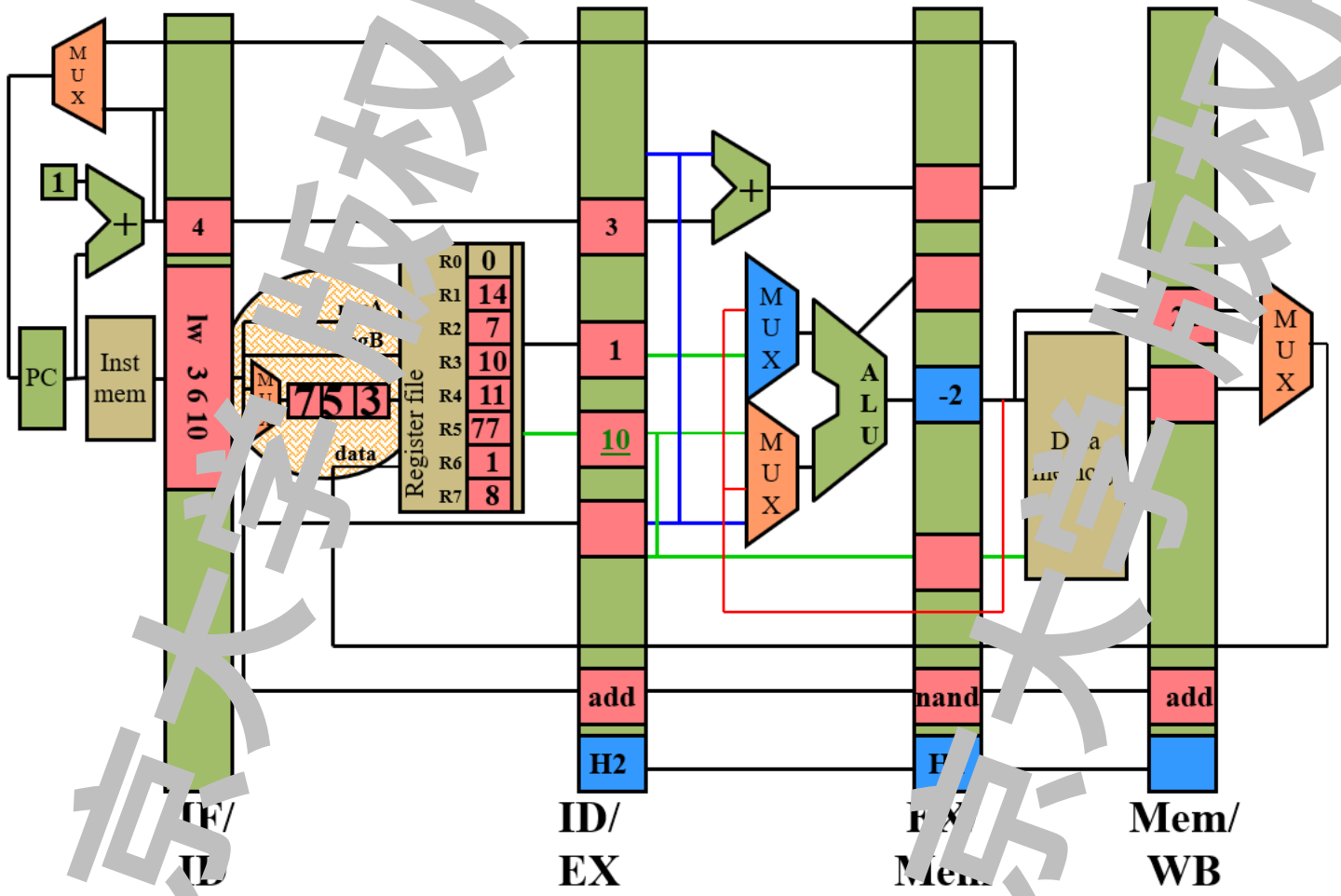
nand 3 4 5 // r5 = r3 NAND r4

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

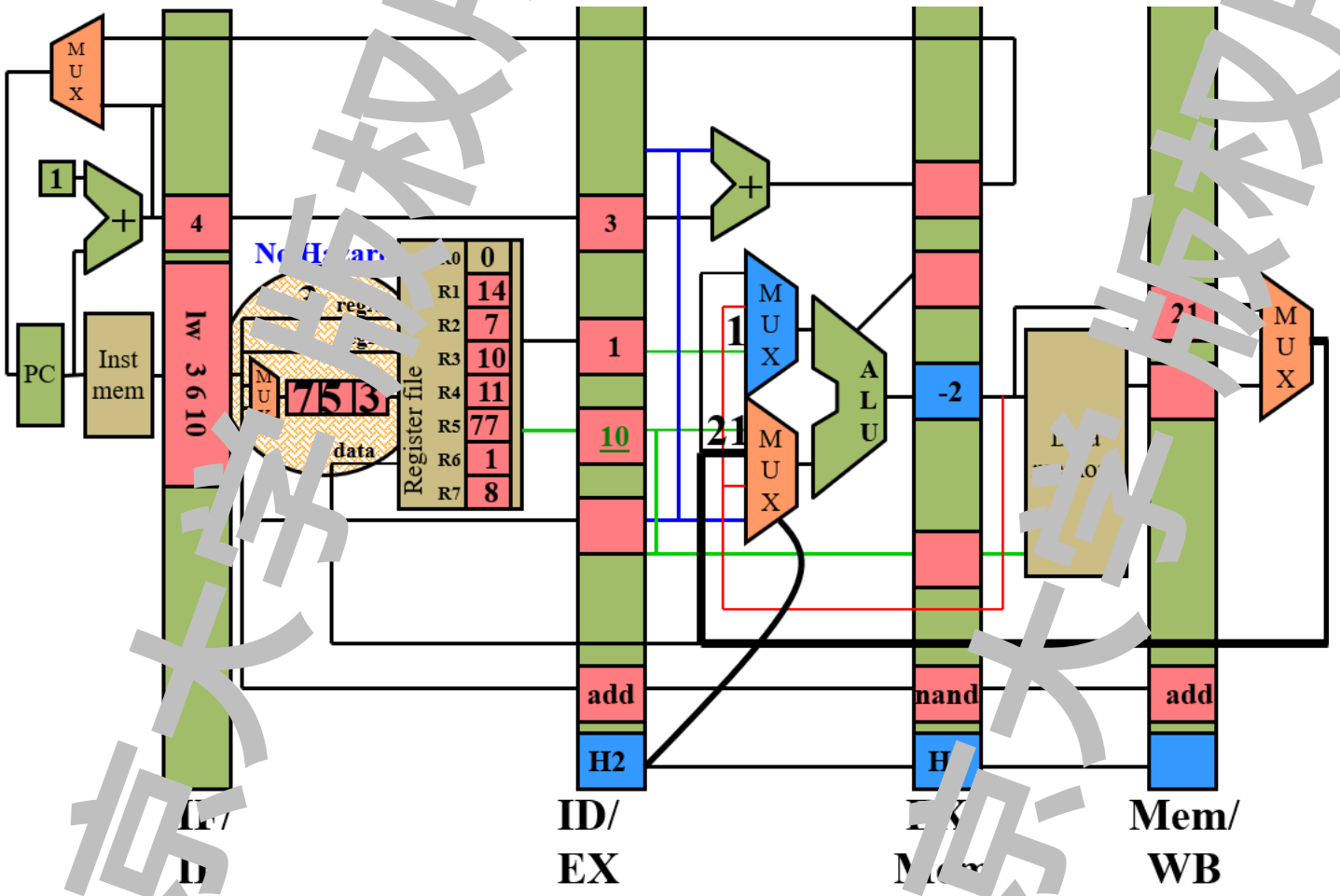
nand 3 4 5 // r5 = r3 NAND r4

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

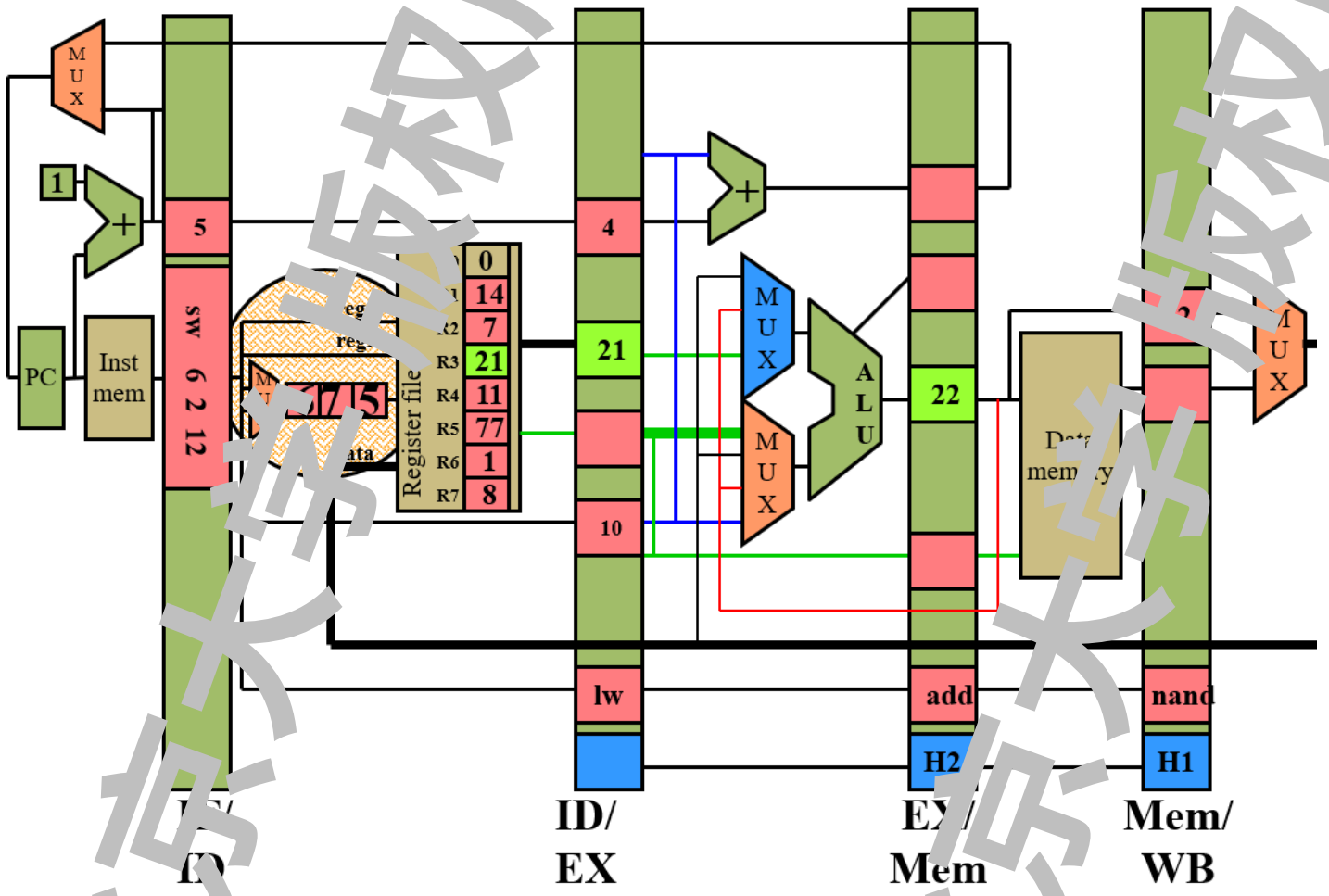
nand 3 4 5 // r5 = r3 NAND r4

add 6 3 7 // r7 = r3 + r6

lw 3 6 0 // 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

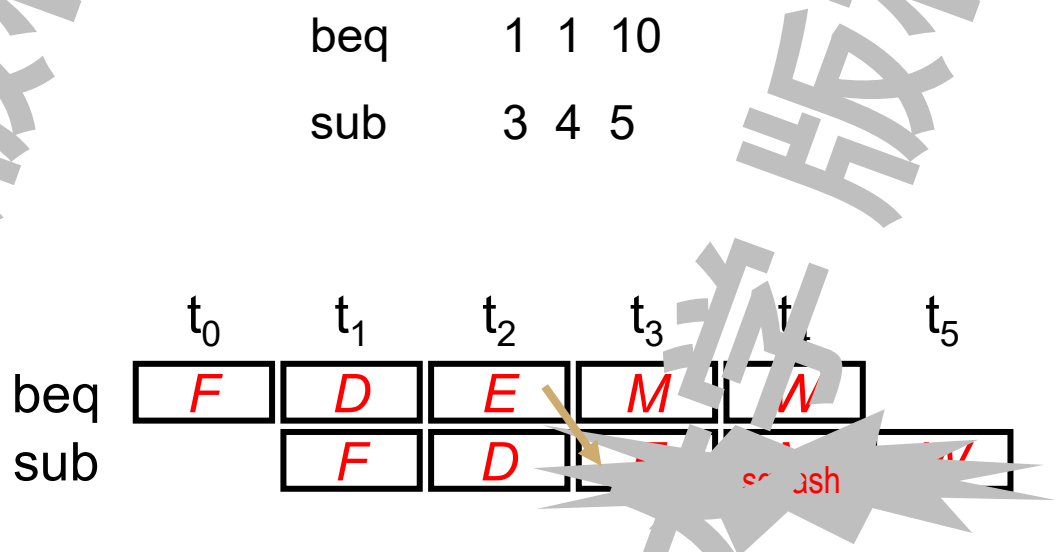
- 在顺序的单条5级流水线上不会出现问题

- **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**

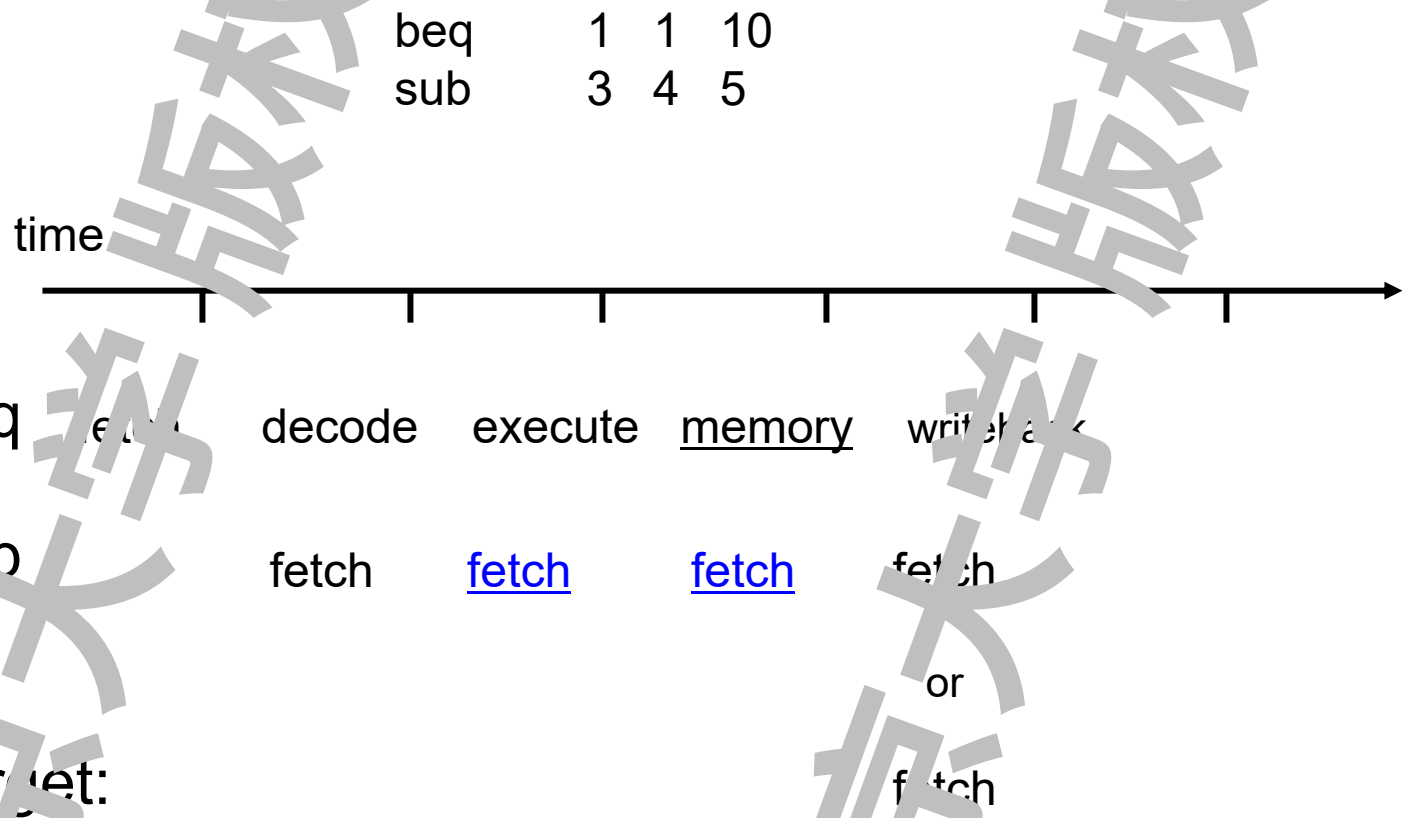
Detection

- 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

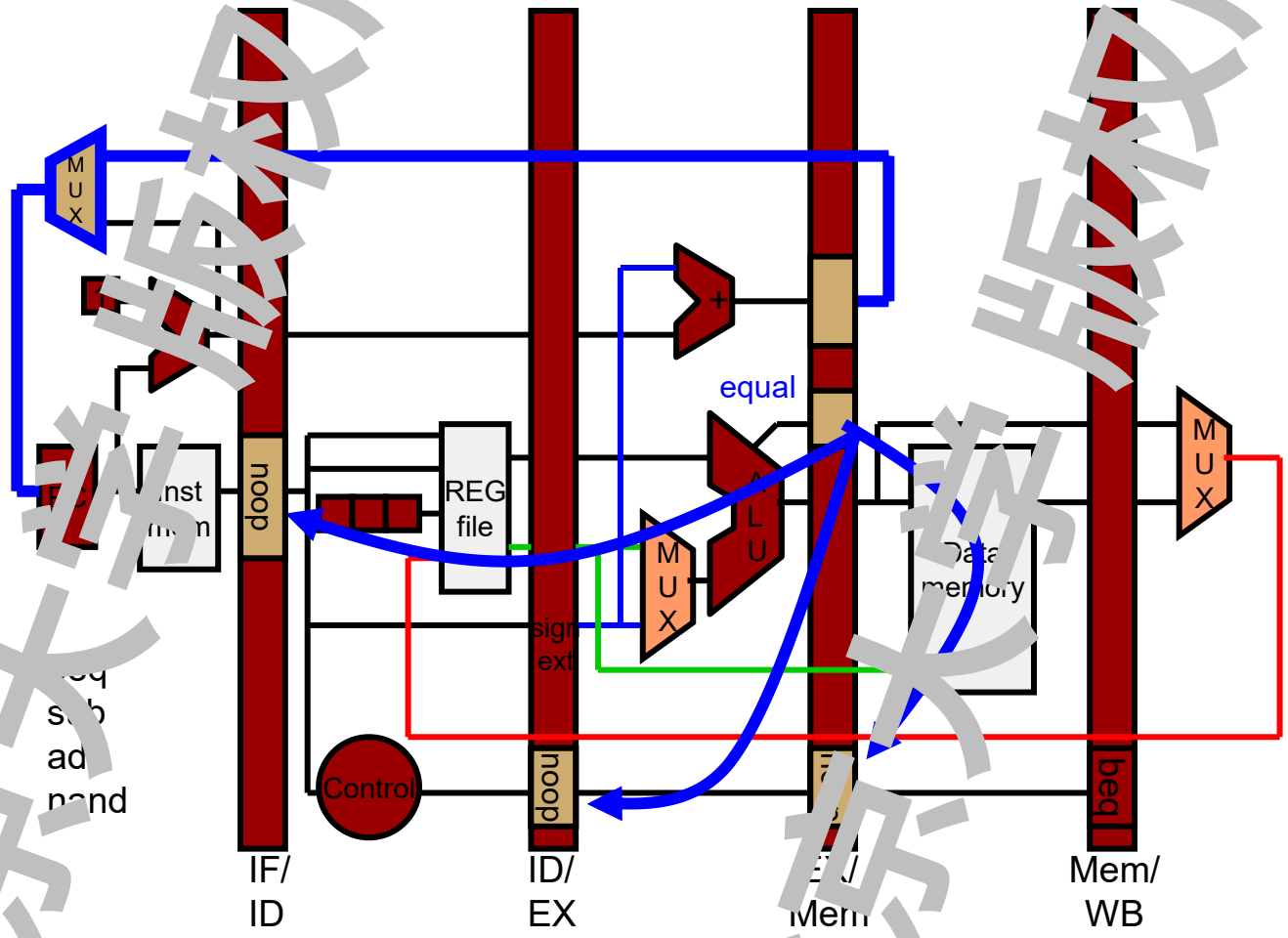
- Speculate and Squash：投机-制止机制

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.

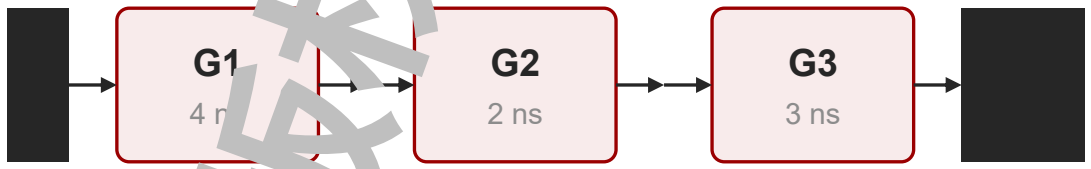
More on branch prediction to come...

北京大学 版权所有

重定时（Retiming）原理 – 在哪里插入寄存器最优？

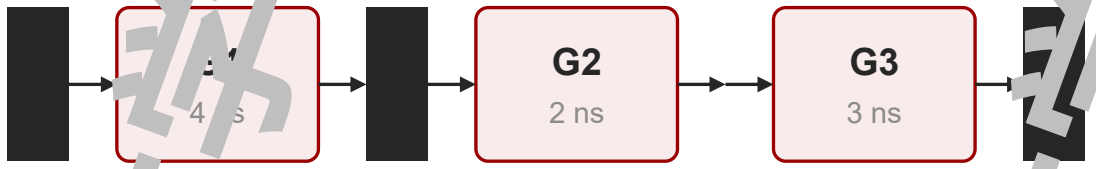
- 在不改变输入/输出行为的前提下移动寄存器位置，约束关键路径

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**（第 6 节）

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
- 综合或布局 retiming，自动分散到逻辑中
- 常用于乘法器、DSP、浮点单元

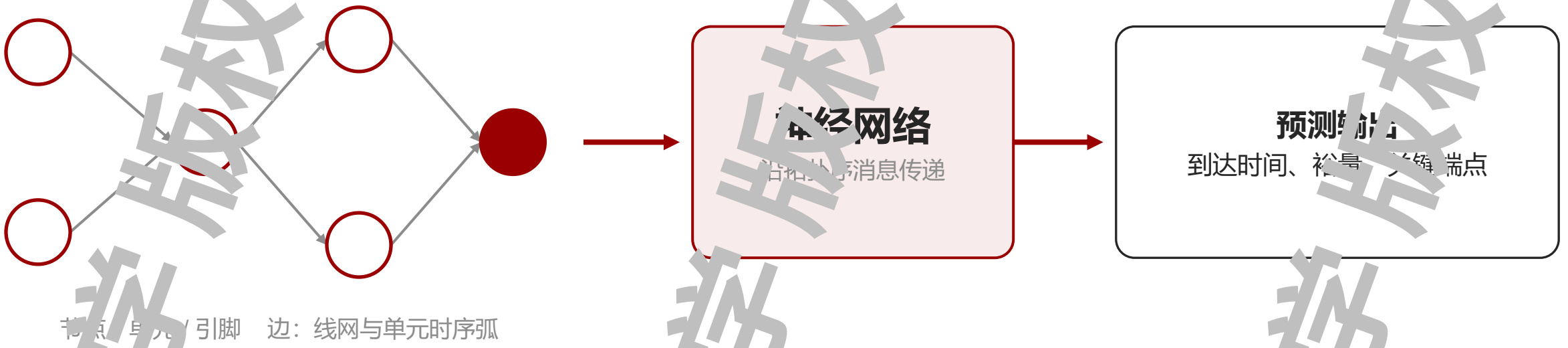
多周期路径（Multicycle Path）

```
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

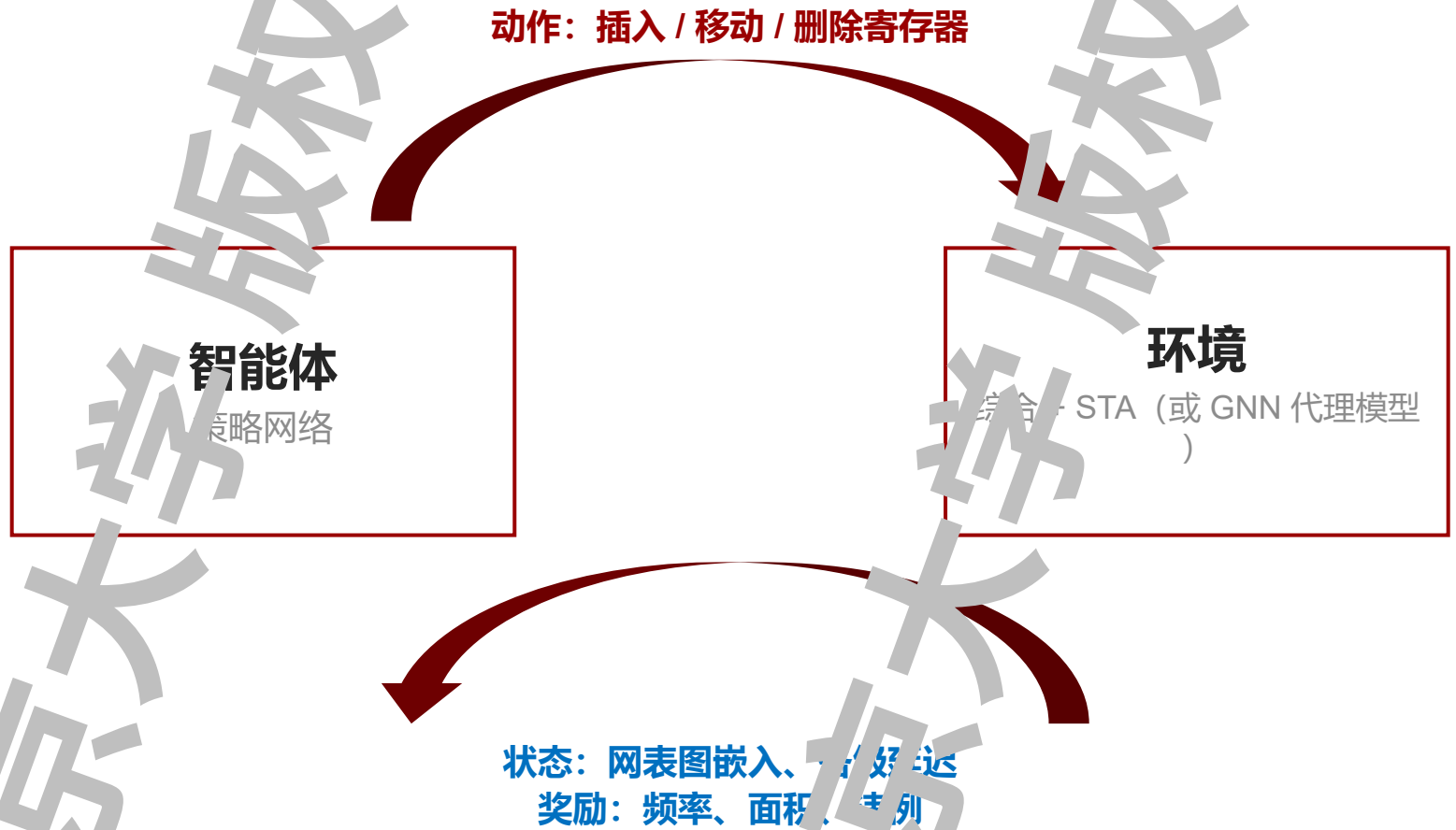
- 布线后 STA 的真实结果
- 跨设计训练，测试泛化

代表工作

- 时序引擎启发的 GNN 预测布线前裕量（DAC 2022）

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

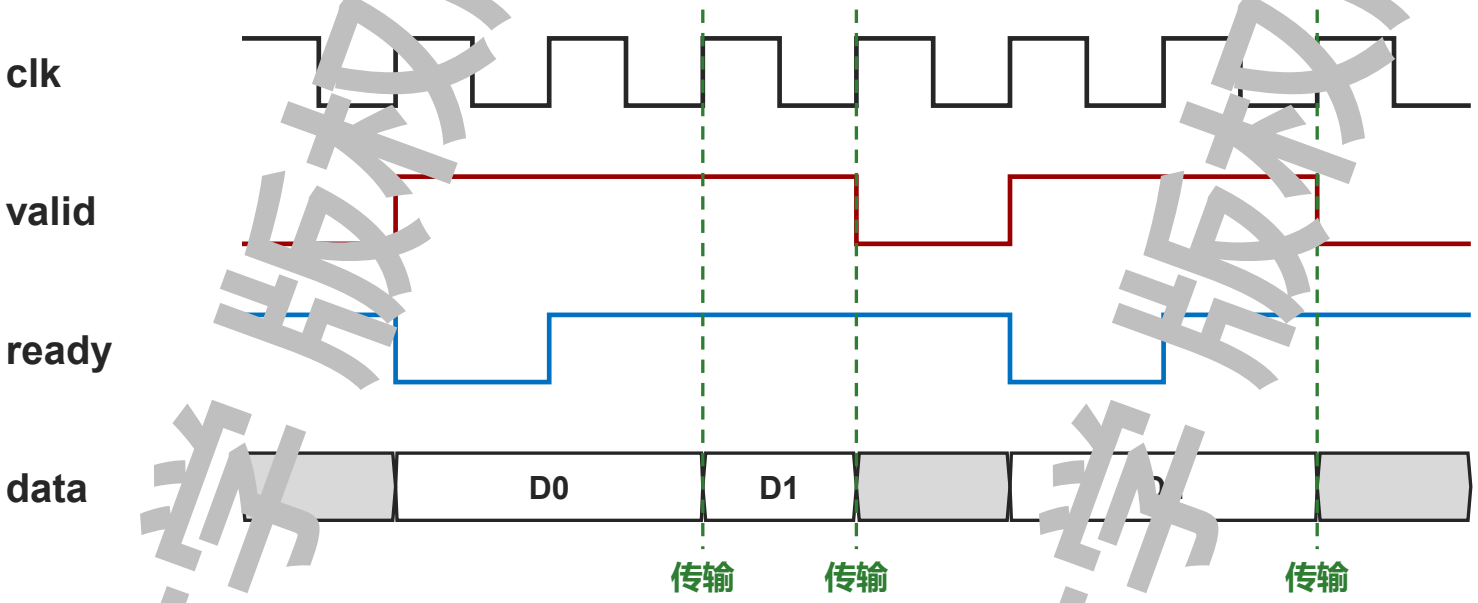
AI辅助时序逻辑设计

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

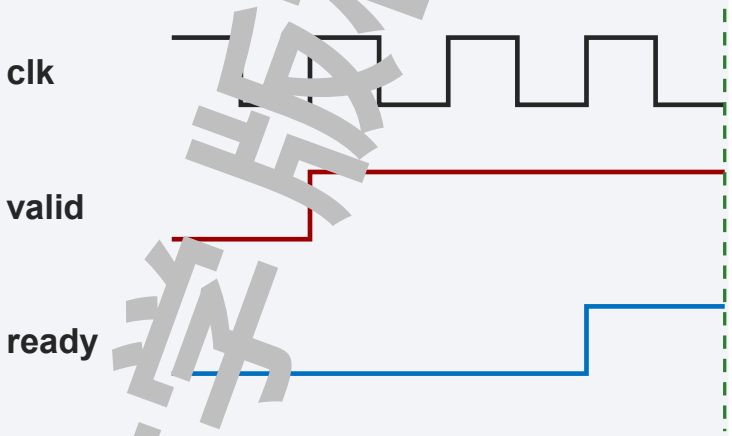
协议规则

- 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

4.2 握手的三种相对关系

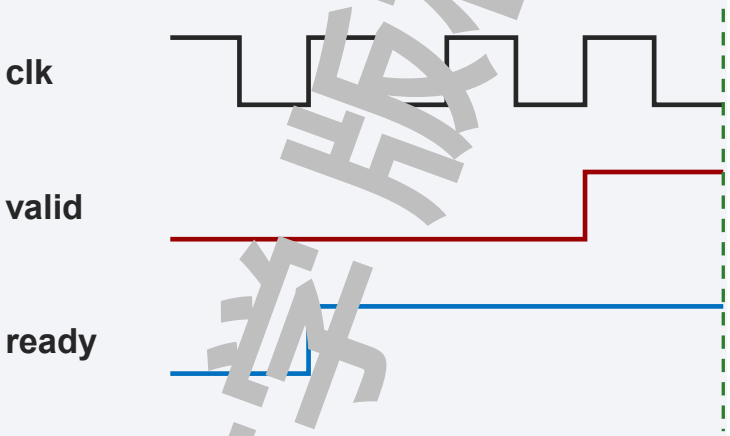
- 谁先拉高都可以，但规则不变：传输只发生在两者同时为高的时钟沿

1 valid 先到



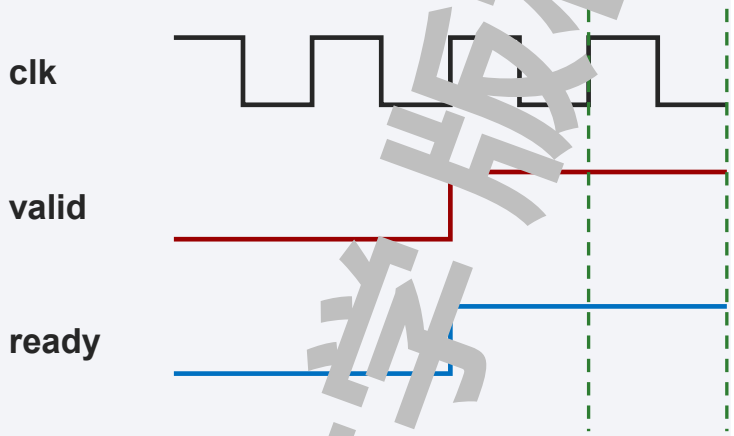
下游已经上游保持 valid 与数据，等待 ready

2 ready 先到



下游空闲提前准备好，valid 一到即传输

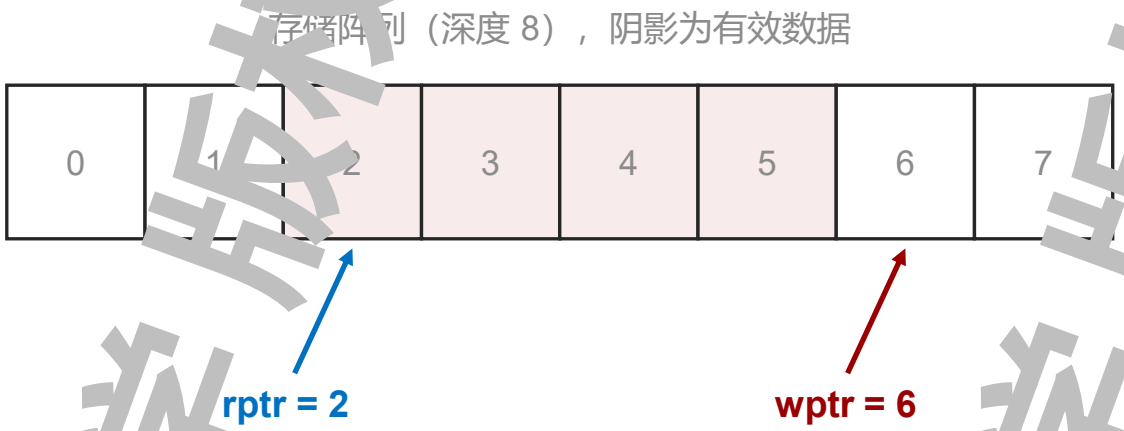
3 同时到达



连续两拍同时为高，每拍传输一个数据

4.3 FIFO：最高效的弹性缓冲

- 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



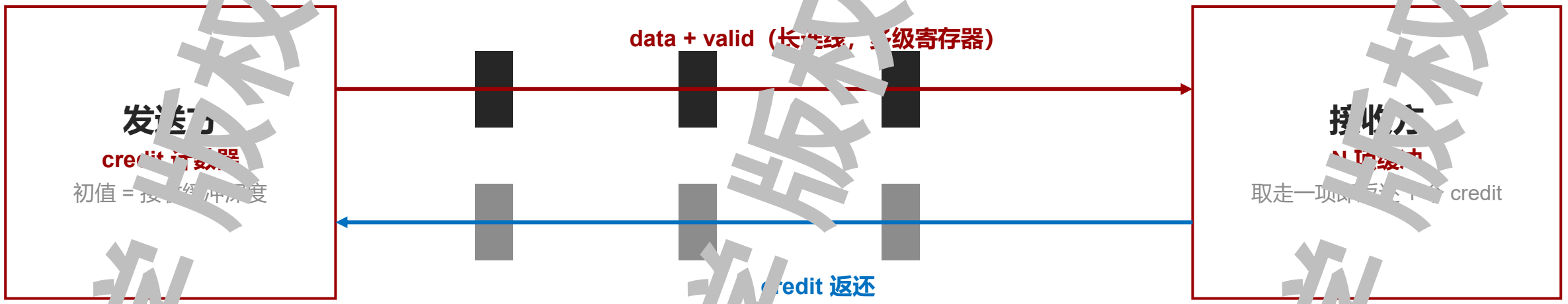
条件	判断（4 位指针，深度 8）
空 empty	wptr == rptr
满 full	wptr[3] != rptr[3] && wptr[2:0] == rptr[2:0]

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
```

4.4 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



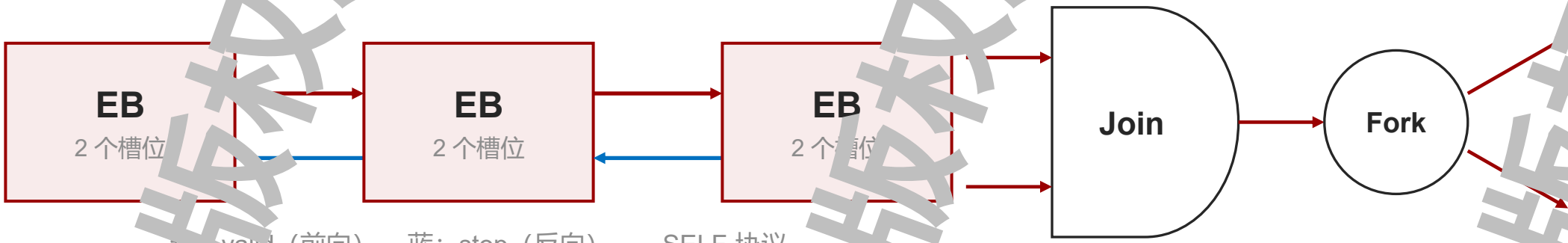
发送	credit > 0 才能发送，每发一个 credit −
返还	接收方释放一项，回传 1 个 credit
缓冲深度	N ≥ 往返延迟（拍） × 每拍数据量，才能满吞吐

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

4.5 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

- stop 信号寄存后晚到一拍
- 多一槽接住在途数据，保证满吞吐

2 带来的自由度

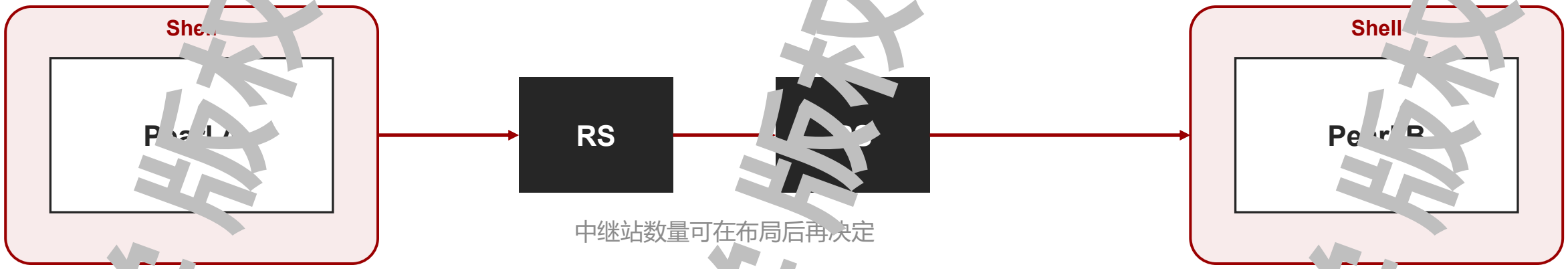
- 后期可在任意通道插入 EB
- 不必重新设计控制逻辑

3 代表工作

- Cortadella、Theobald、Sokolov 等提出的同步弹性电路与 SELF 协议

4.6 延迟不敏感（Latency-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）

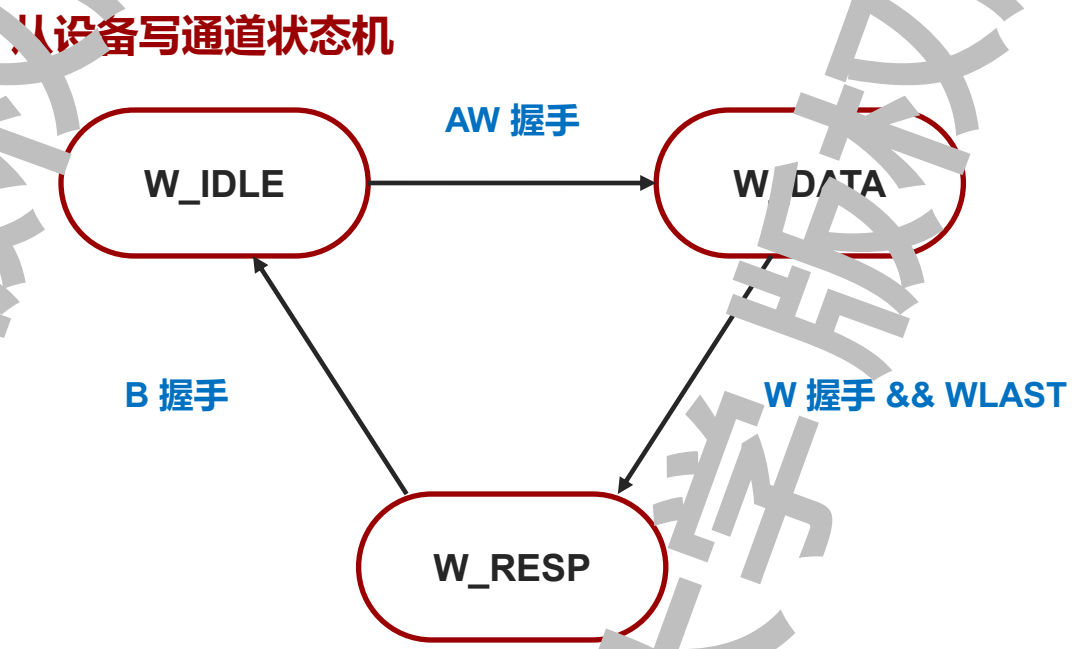
Relay Station
2 槽缓冲，切断长连线，可任意插入

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

4.7 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

4.8 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”

4.9 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

第五讲 LECTURE 05

多时钟域与异步时序逻辑

北京大学版权所有

5.1 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

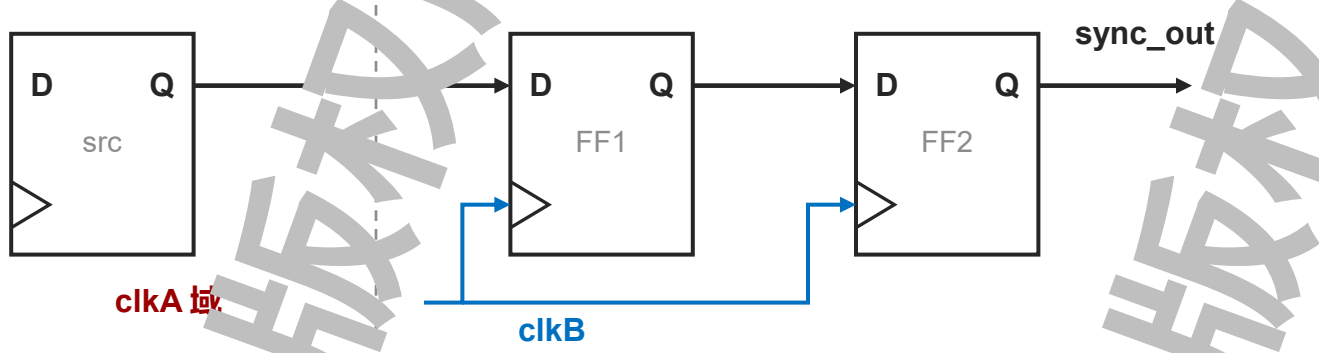
- 各位走线延迟不同
- 同一拍采到“一半新、一半旧”的值

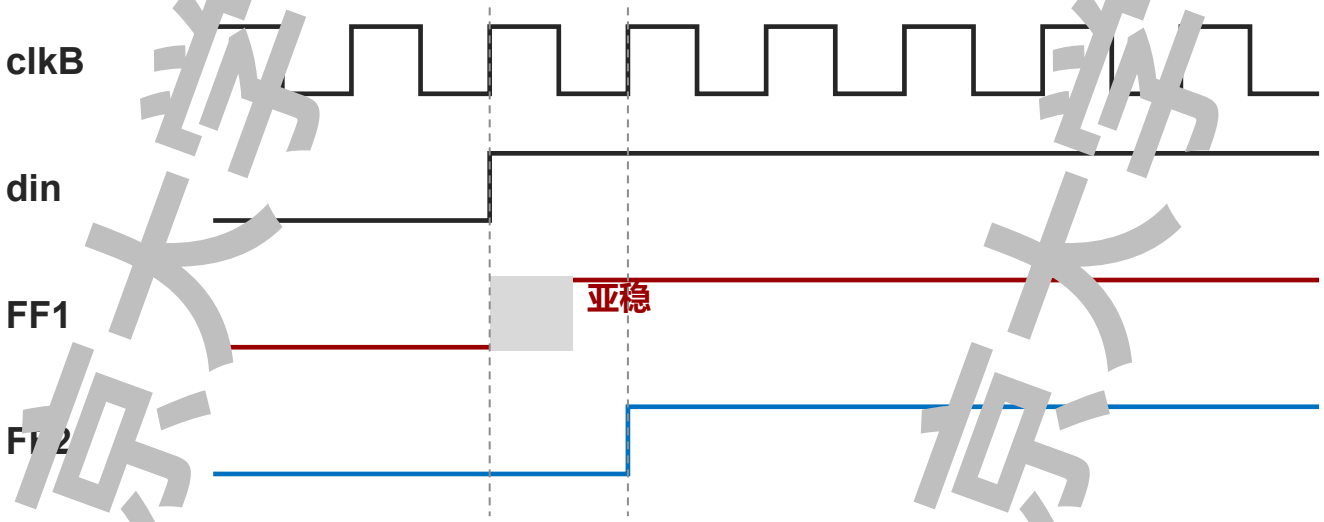
3 数据丢失

- 快时钟域的单个脉冲
- 慢时钟域可能根本采不到

5.2 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，最大化减小间距
- 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

5.3 多比特与事件的跨域方法

- 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：使能、模式、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 握手（req/ack 应答握手）

- req、ack 各自同步，数据在 ack 返回前保持
- 适用：低速多比特数据，延迟数拍

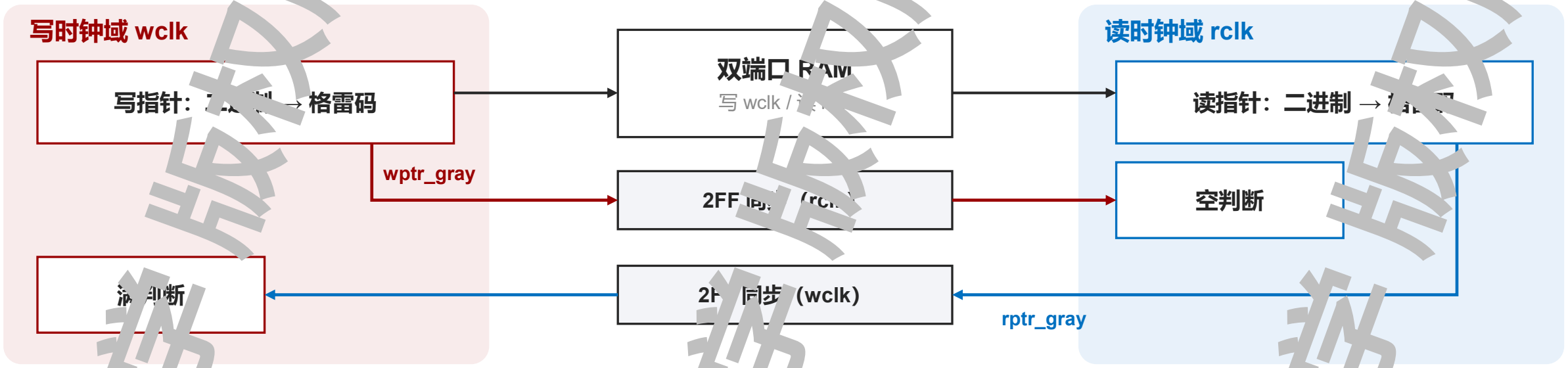
4 数据 + 使能（MCP 法）

- 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

需要高吞吐地传递连续数据 → 使用异步 FIFO（下一页）

5.4 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

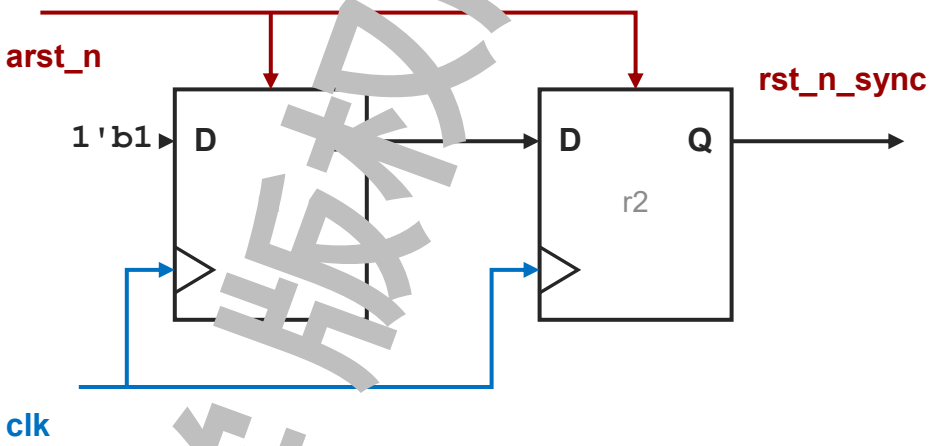
空：rptr_gray == wptr_gray_sync

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

5.5 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

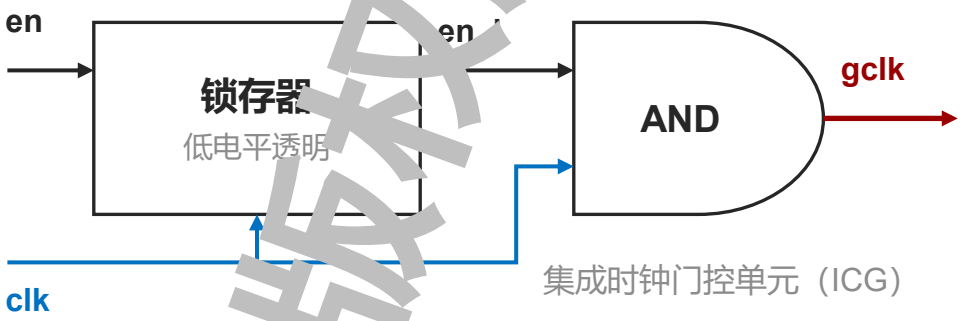
- 复位撤除若靠近时钟沿，触发器出现恢复 / 移除时间违例
- 各触发器退出复位的拍数不一致，状态机起步错拍

复位域交叉（RDC）

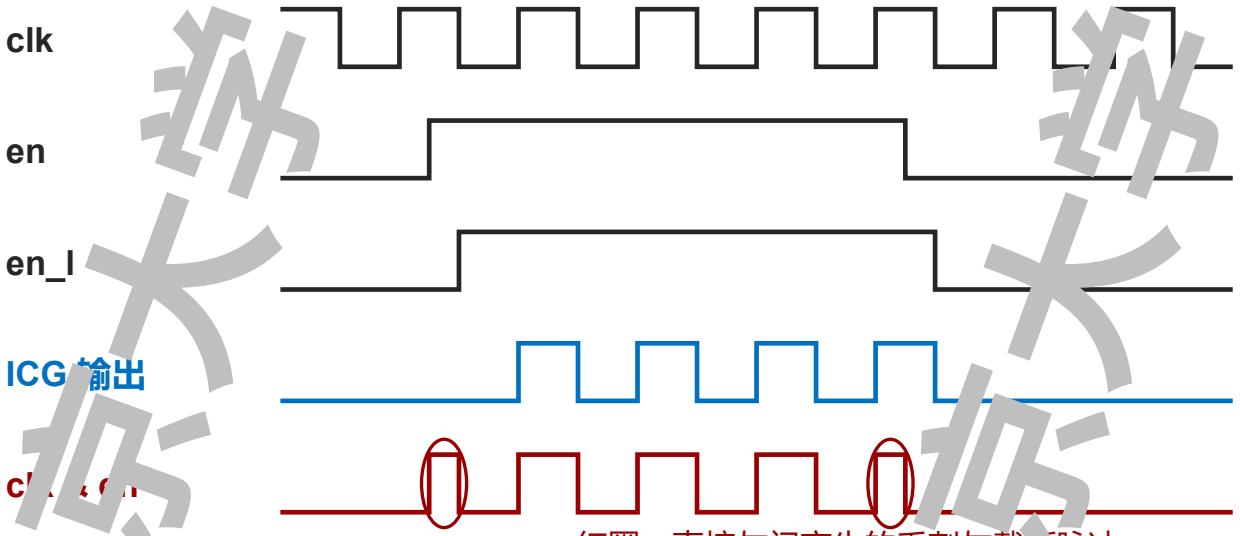
- 源寄存器被复位，而目的寄存器仍在工作
- 复位瞬间源端异步跳变 → 目的端亚稳态
- 对策：控制复位顺序、加隔离、目的端同步

5.6 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截断脉冲

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

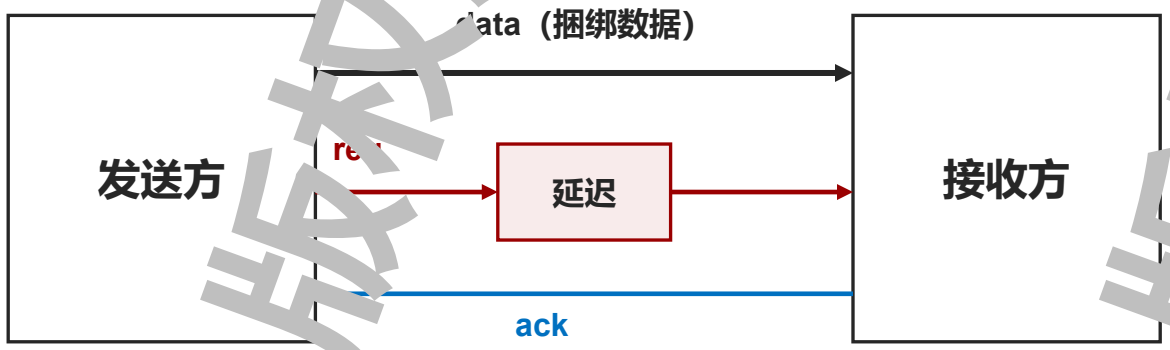
模块级 + 寄存器级，门控越靠近时钟，省电越大

时序检查

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

5.7 异步（无时钟）电路基础

- 没有全局时钟，模块之间靠**握手**同步：数据何时有效由协议或编码本身表达





四相握手：req↑ → ack↑ → req↓ → ack↓

优点：无时钟树功耗，平均情况性能，电磁辐射低

双轨编码（Dual-Rail）

d.t	d.f	含义
0	0	空值 NULL（初始）
1	0	逻辑 1
0	1	逻辑 0
1	1	非法

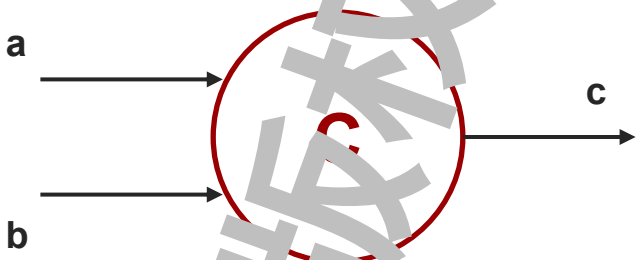
数据自带“完成”信息，无需匹配延迟

挑战：面积开销大，EDA 工具与验证方法不成熟

思想自由 兼容并包

5.8 Muller C 单元与零空逻辑（NCL）

- C 单元在输入一致时才改变输出，是异步电路中的“状态保持”元件



a	b	c^+
0	0	0
0	1	c（保持）
1	0	c（保持）
1	1	1

NCL 阈值门 THmn（带滞回）

门	置 1 条件	置 0 条件
TH12	任一输入为 1	全部输入为 0
TH22	两个输入都为 1	全部输入为 0
TH23	3 个输入中至少 2 个为 1	全部输入为 0

TH22 即 C 单元；其余情况保持原值

NCL 的数据波与空值波交替推进

DATA → NULL → DATA → NULL → DATA

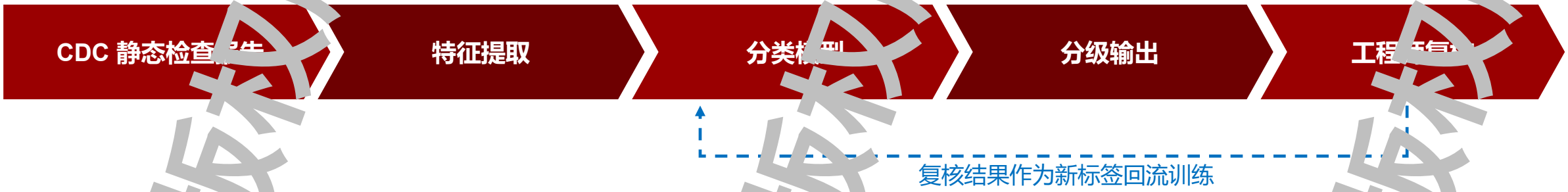
完成检测电路确认所有输出都变为 DATA（或 NULL）后，才允许下一波进入

AI 机会

异步电路缺少成熟工具，LLM 可辅助把同步 RTL 改写为 NCL 结构，再用形式化方法检查

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

第六讲 LECTURE 06

时序逻辑的形式化验证与时态逻辑

北京大学版权所有

6.1 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 个触发器可能的状态数

约 585 年

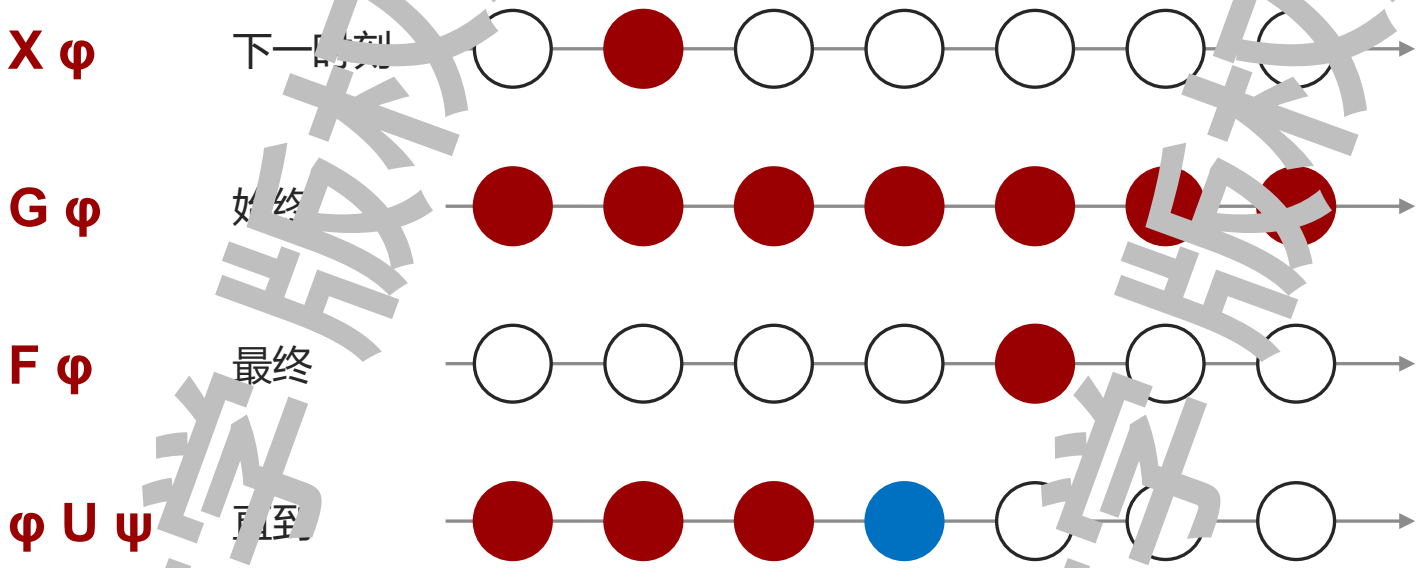
即使每秒遍历 10^9 个状态，穷举仍需的时间

	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”

6.2 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

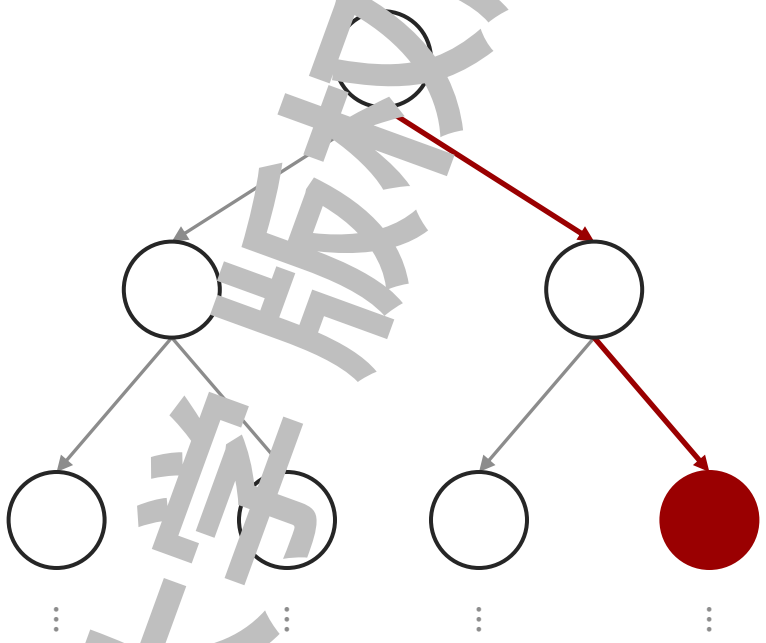
- 反例是一条有限长的路径
- 如互斥、FIFO 不溢出

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

6.3 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

6.4 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_props.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);
```

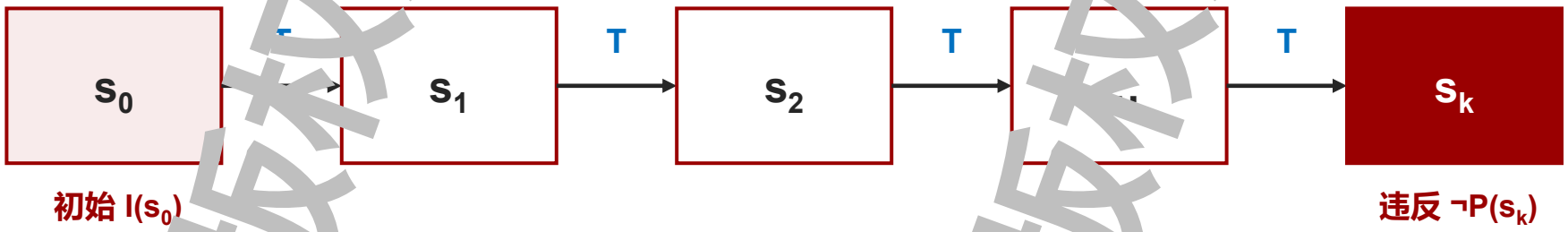
算子	含义
##n	延迟 n 拍
##[m:n]	延迟 m 到 n 拍
|->	同拍蕴含
|=>	下一拍蕴含
$rose / $fell	上升沿 / 下降沿
$stable / $past	保持不变 / 前一拍值
[*n]	连续重复 n 次

assert ≈ AG，cover ≈ EF

北京大学版权所有

6.5 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



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

1 有界模型检测思想

- 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

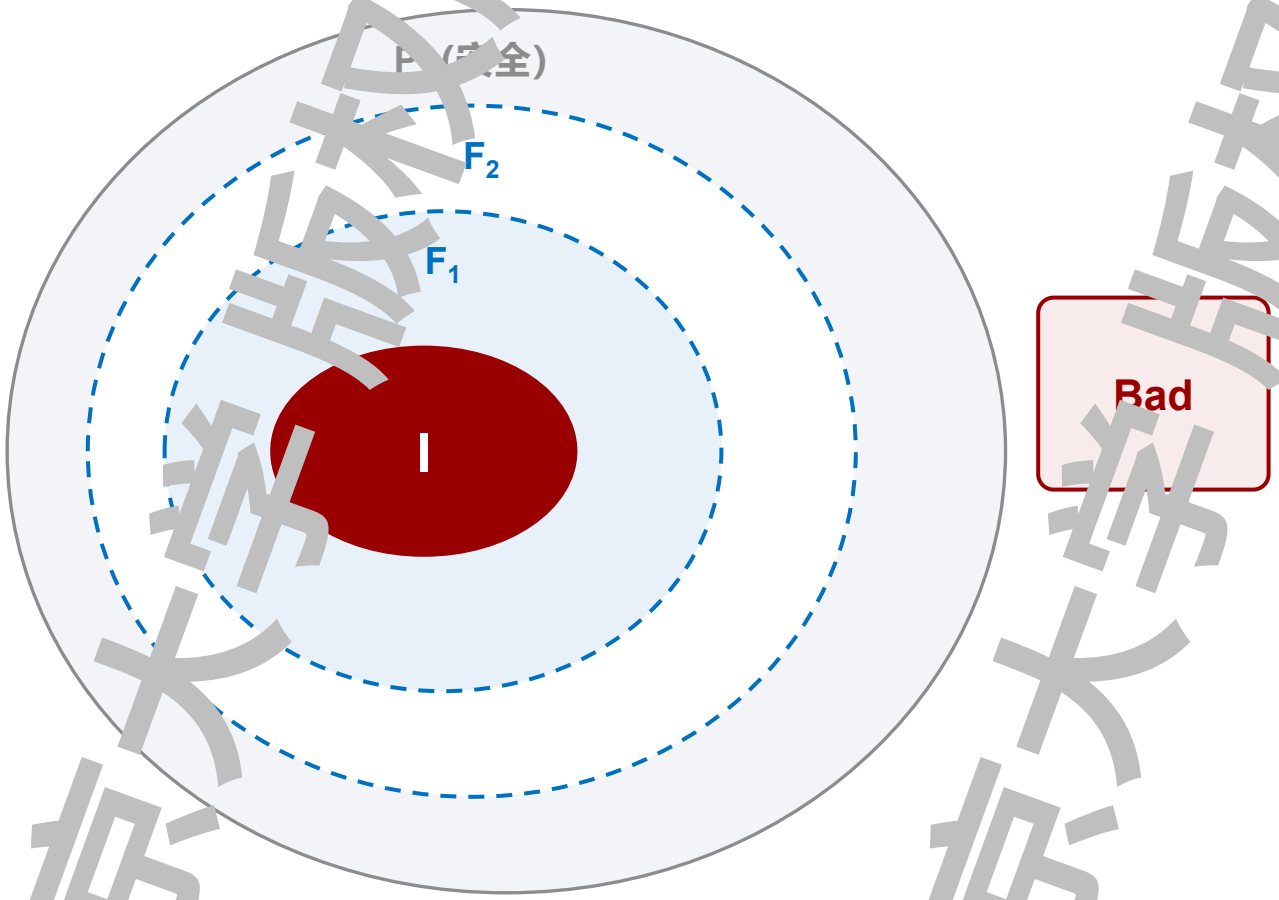
- 得到一条长度 ≤ k 的违例轨迹
- 导出 VCD 波形，便于调试

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

6.6 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

- 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

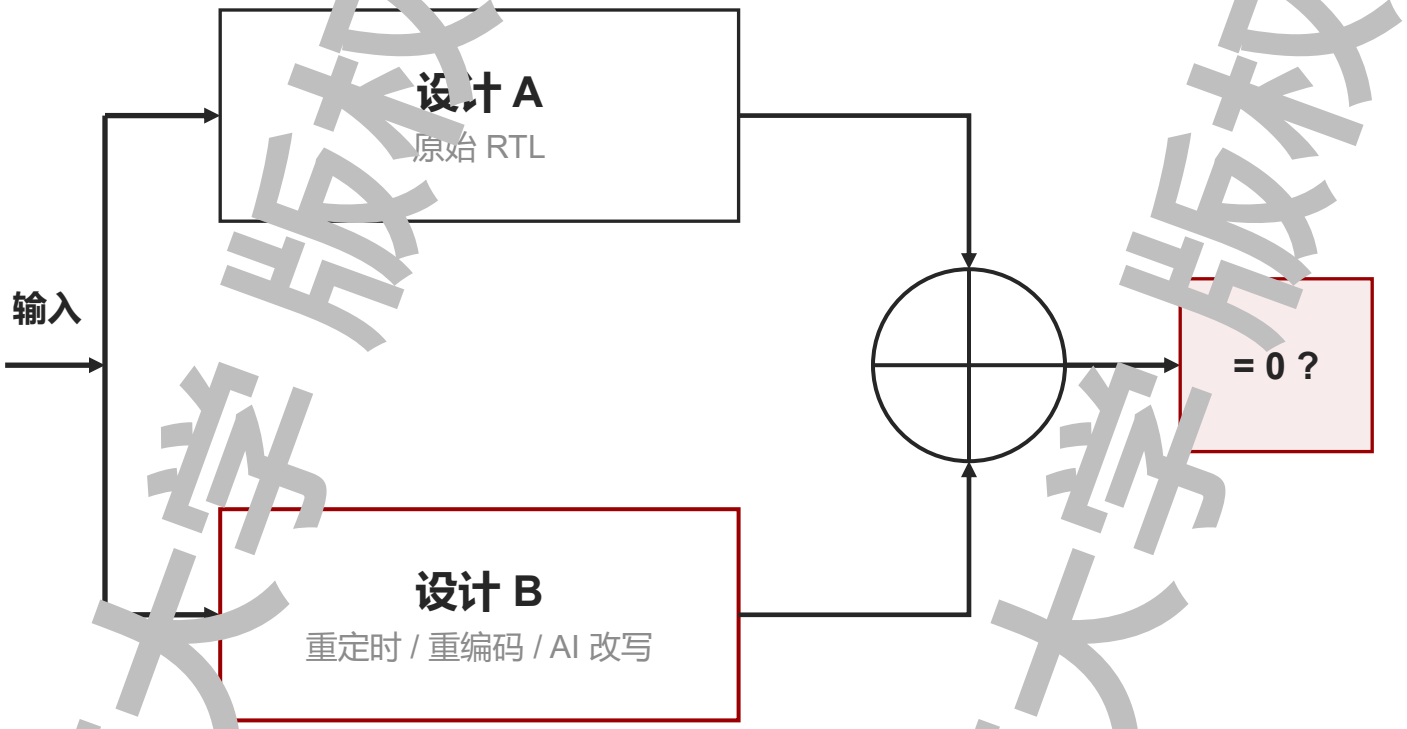
逐帧学习子句，阻断反例前驱；相邻两帧相同即得不变式

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

北京大学版权所有

6.7 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

6.8 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

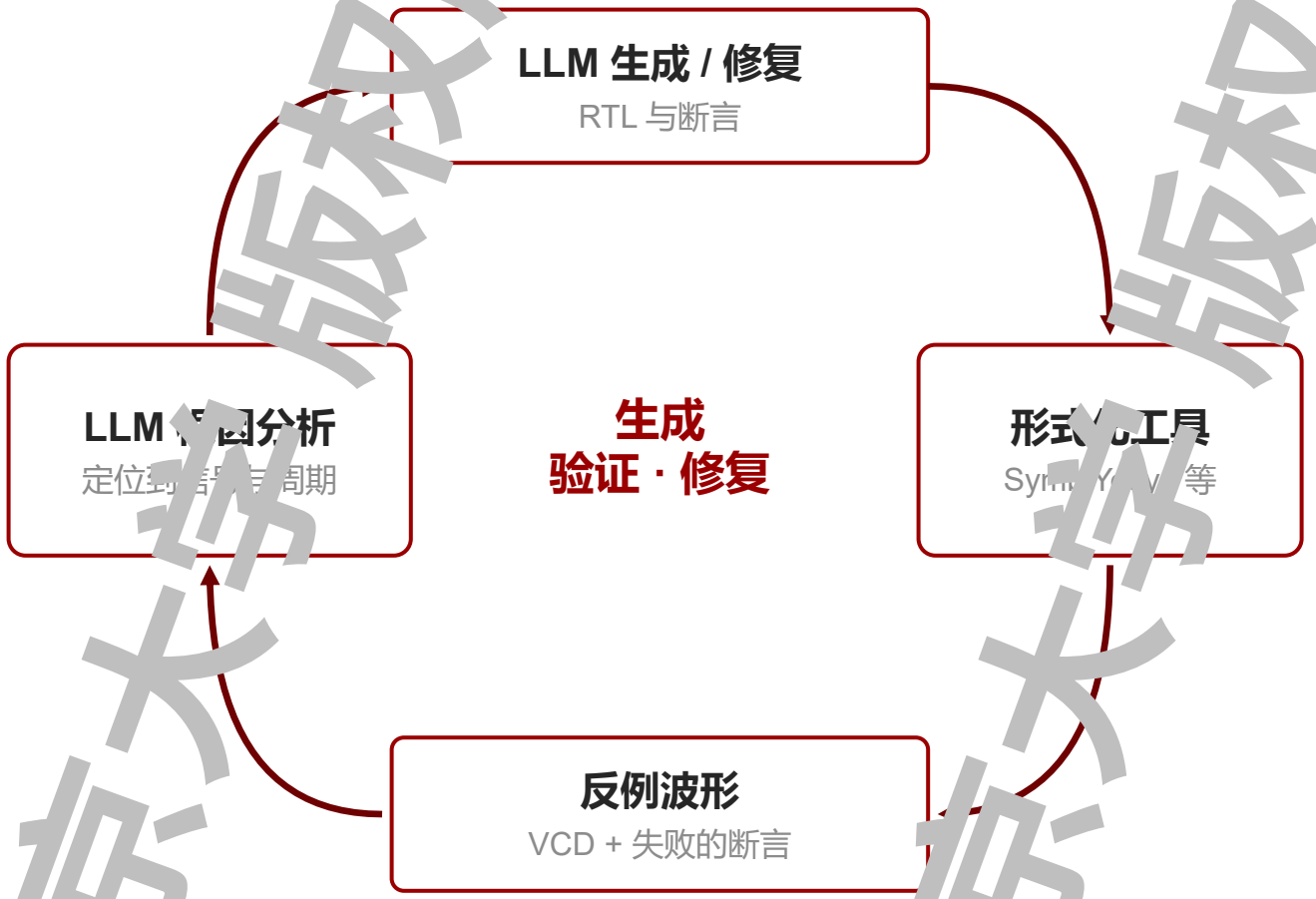
断言质量筛选流程

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

6.9 “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