



天津大学

数字逻辑与数字系统

4

时序逻辑设计

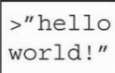


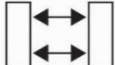
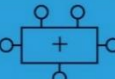

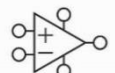


Sequential Logic Design



本章内容

Topic

- 引言
- 锁存器和触发器
- 同步逻辑设计
- 有限状态机
- 时序逻辑中的时序问题
- 时序逻辑模块
- 存储器阵列

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	



本章内容

Topic

□ 引言

□ 锁存器和触发器

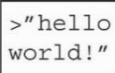


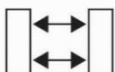
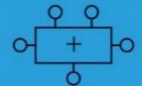
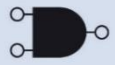
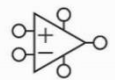


□ 同步逻辑设计

□ 有限状态机

□ 时序逻辑中的时序问题

□ 时序逻辑模块

□ 存储器阵列

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	



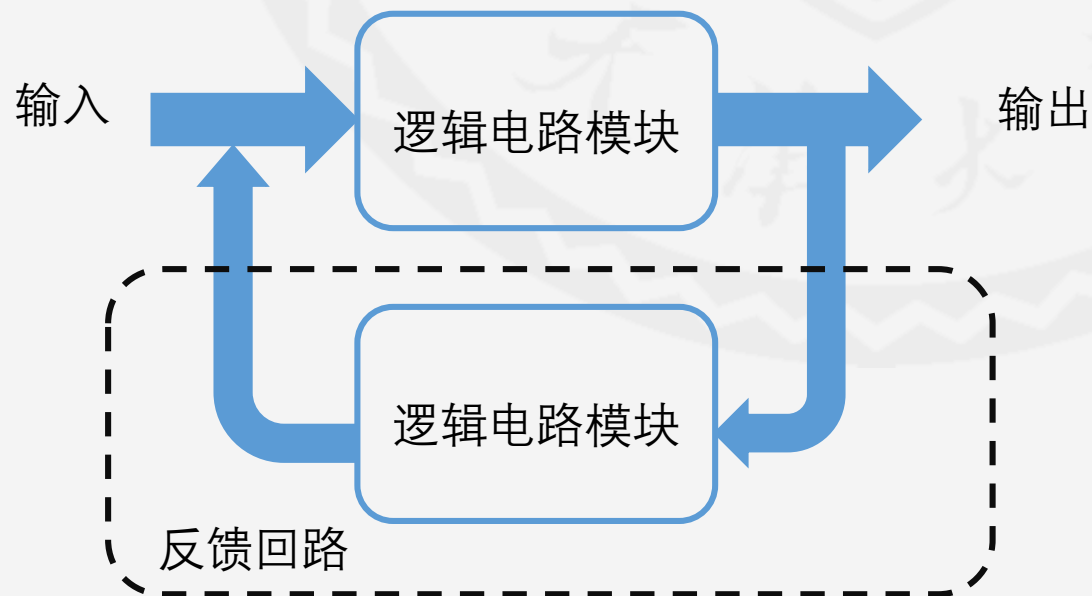
时序逻辑电路

- 时序逻辑电路的输出由当前时刻的输入和之前时刻的输入共同决定
- 时序逻辑电路内部具有记忆
- 一些基本概念
 - 状态：用于解释电路未来行为所需的信息
 - 锁存器与触发器：用于存储1比特状态的模块
 - 同步时序逻辑电路：一类由组合逻辑和一组表示电路状态的触发器所构成的电路



时序逻辑电路的特征

- 按照一定的输入输出时序实现功能
- 电路内部具有短期记忆
- 在输出与输入之间具有反馈回路





本章内容

Topic

□ 引言

□ 锁存器和触发器

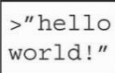


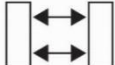
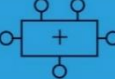
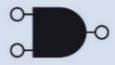
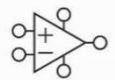


□ 同步逻辑设计

□ 有限状态机

□ 时序逻辑中的时序问题

□ 时序逻辑模块

□ 存储器阵列

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	



锁存器和触发器

Latches & Flip-flops

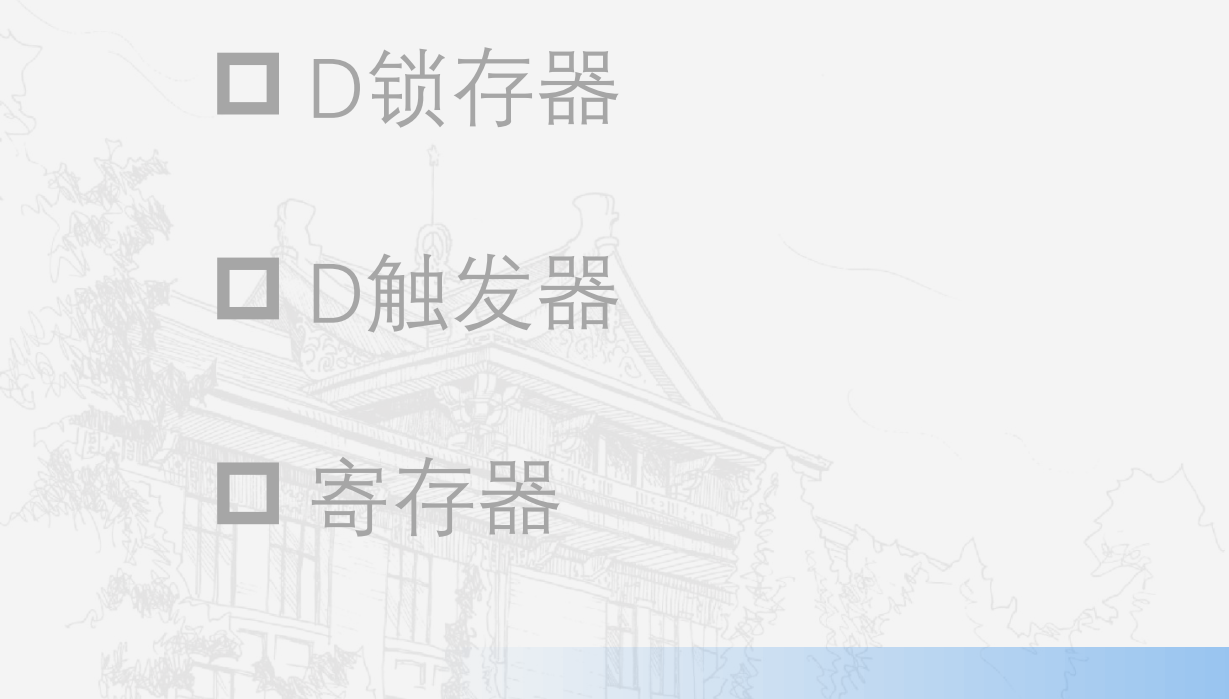
□ 双稳态电路

□ SR锁存器

□ D锁存器

□ D触发器

□ 寄存器





电路中的状态

- 电路中的状态会影响电路未来的行为
- 一些常见的电路状态存储模块
 - 双稳态电路
 - SR锁存器
 - D锁存器
 - D触发器

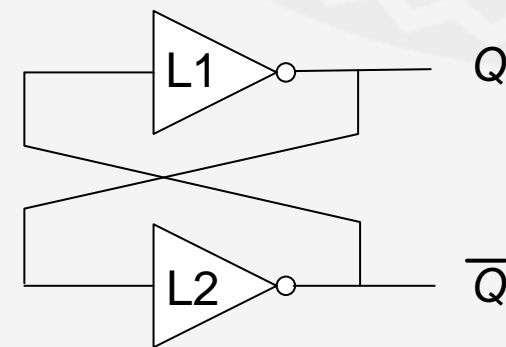
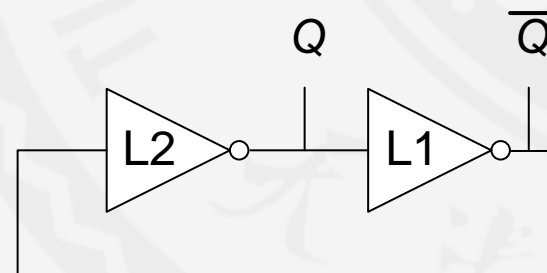


双稳态电路

Bitstable Circuit

双稳态电路

- 双稳态电路是其他存储模块的基础
- 右侧电路具有以下特点
 - 对称性
 - 有两个输出： Q 和 \bar{Q}
 - 没有输入





双稳态电路的分析

■ 考虑以下两种情况

■ $Q = 0$

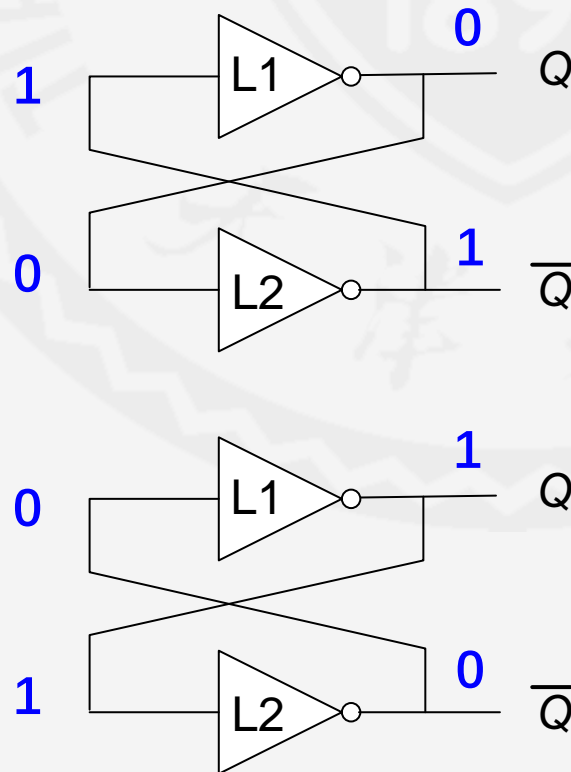
则 $\bar{Q} = 1$, $Q = 0$, 与假设一致

■ $Q = 1$

则 $\bar{Q} = 0$, $Q = 1$, 与假设一致

■ 由以上分析可以看到，电路存在两种状态，用0和1表示，可以用这两种状态存储1比特二进制状态

■ 但该电路中没有输入端，无法控制电路中状态的存储





锁存器和触发器

Latches & Flip-flops

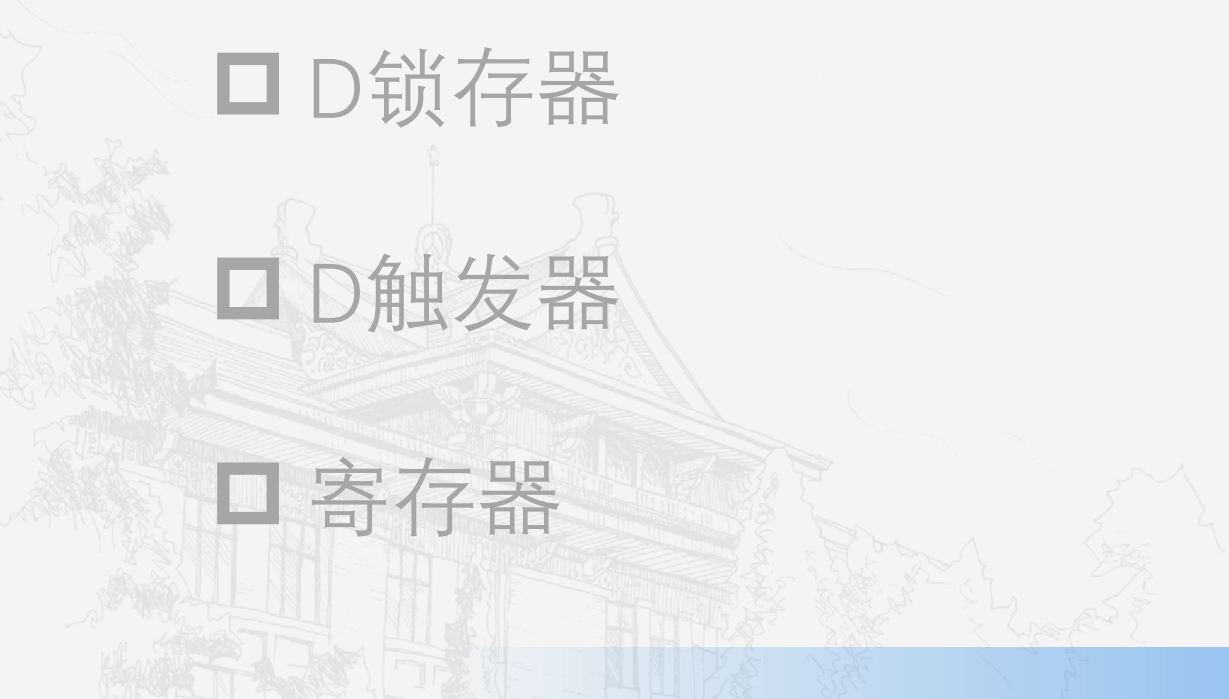
□ 双稳态电路

□ SR锁存器

□ D锁存器

□ D触发器

□ 寄存器





SR锁存器

SR Latch

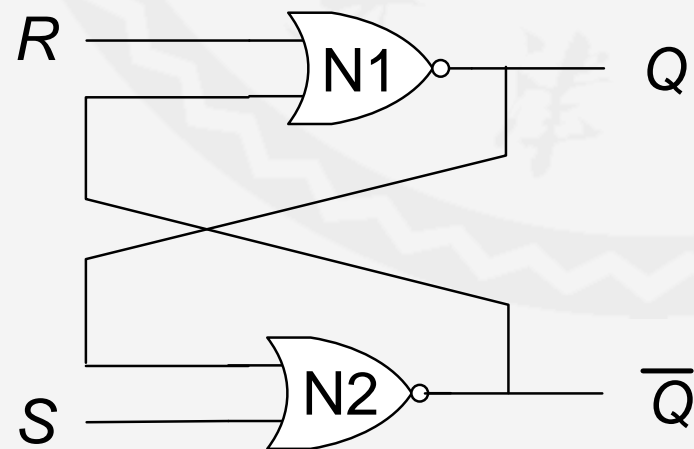
SR 锁存器

SR锁存器

- 两个输入端（激励信号）
- 两个输出端（状态）

考虑以下四种输入情况

- $S = 1, R = 0$
- $S = 0, R = 1$
- $S = 0, R = 0$
- $S = 1, R = 1$





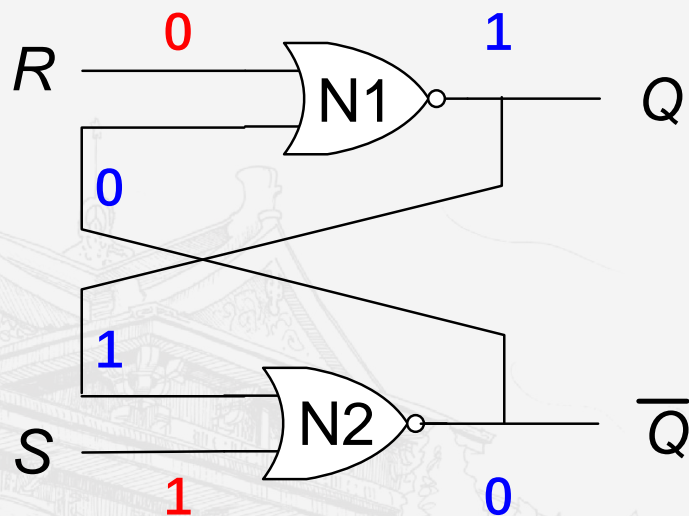
SR锁存器

SR Latch

SR 锁存器分析

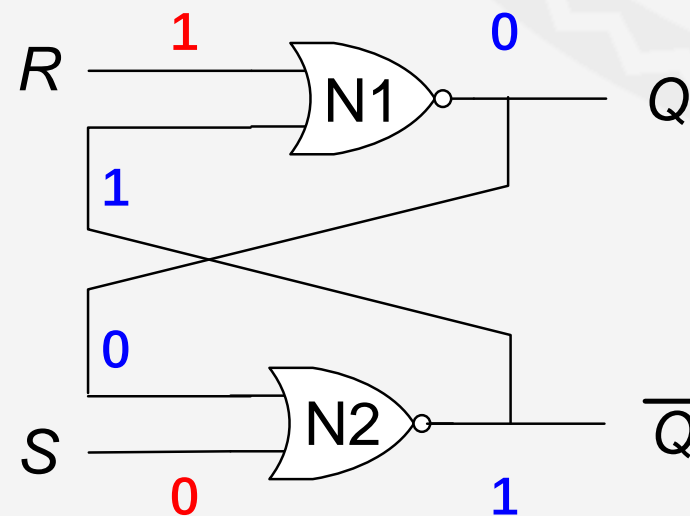
■ $S = 1, R = 0$ 置位

则 $Q = 1$ 且 $\bar{Q} = 0$



■ $S = 0, R = 1$ 复位

则 $Q = 0$ 且 $\bar{Q} = 1$





SR锁存器

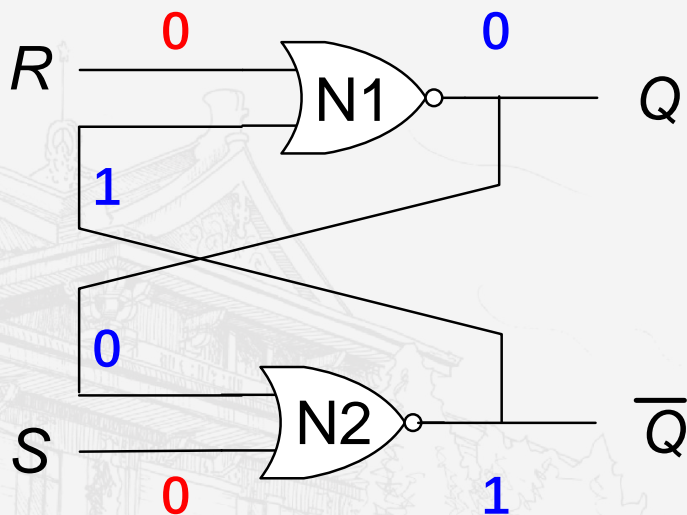
SR Latch

SR 锁存器分析 (cont.)

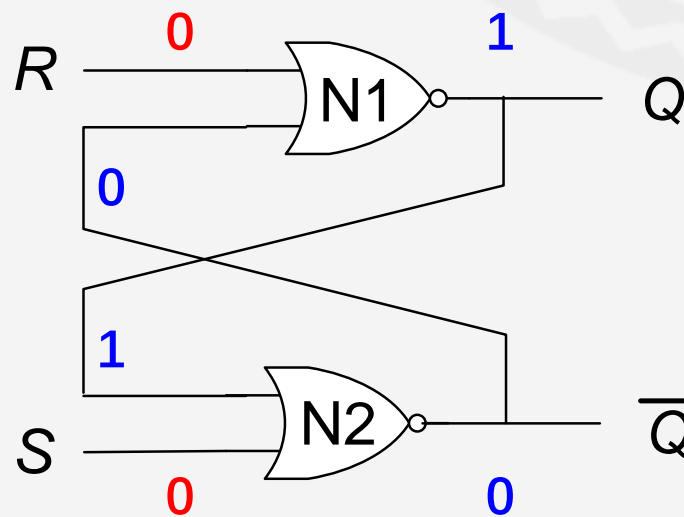
■ 设: 电路的原状态表示为 Q^n , 新状态表示为 Q^{n+1}

■ $S = 0, R = 0$ 则 $Q^{n+1} = Q^n$ 保持

$Q^n = 0$ 时



$Q^n = 1$ 时



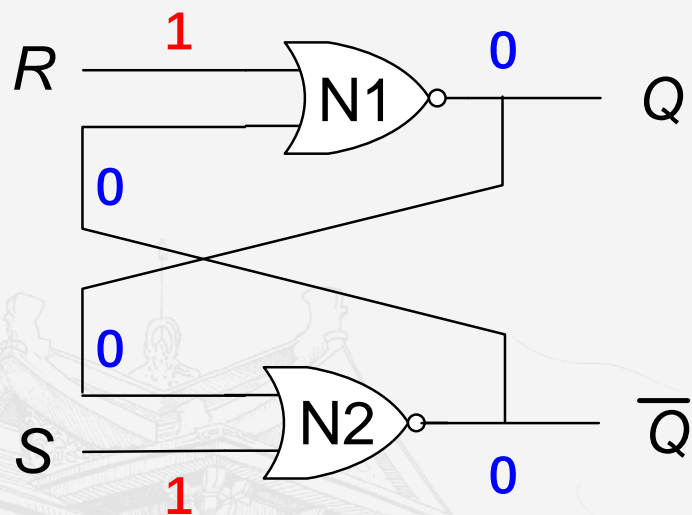


SR锁存器

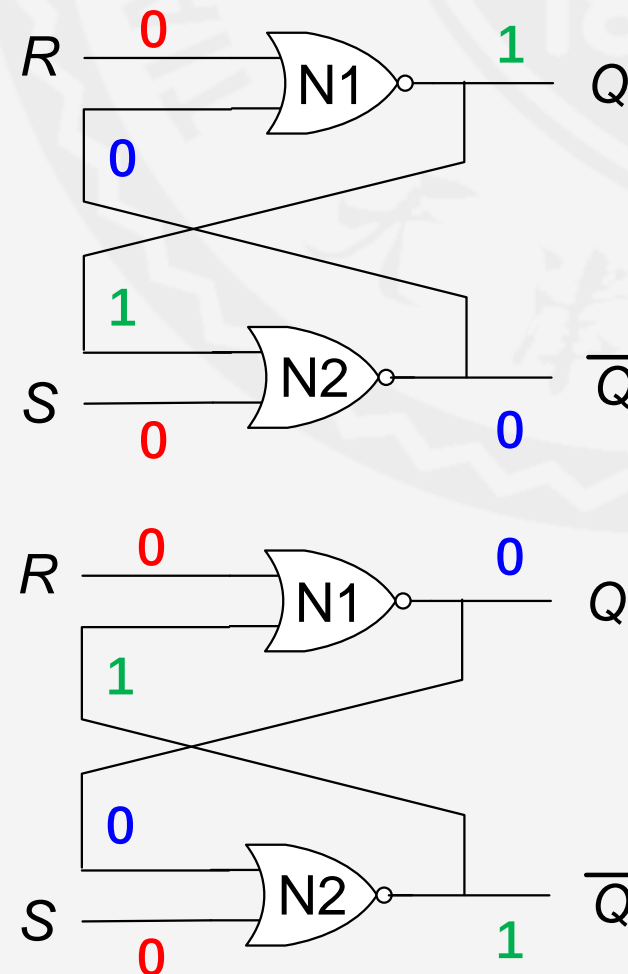
SR Latch

SR 锁存器分析 (cont.)

■ $S = 1, R = 1$ 则 $Q = \bar{Q} = 0$ 不是稳态



激励信号同时消失后



结论：激励信号同时消失后,结果不确定，一般情况下， $S = R = 1$ 应禁止使用。



SR锁存器

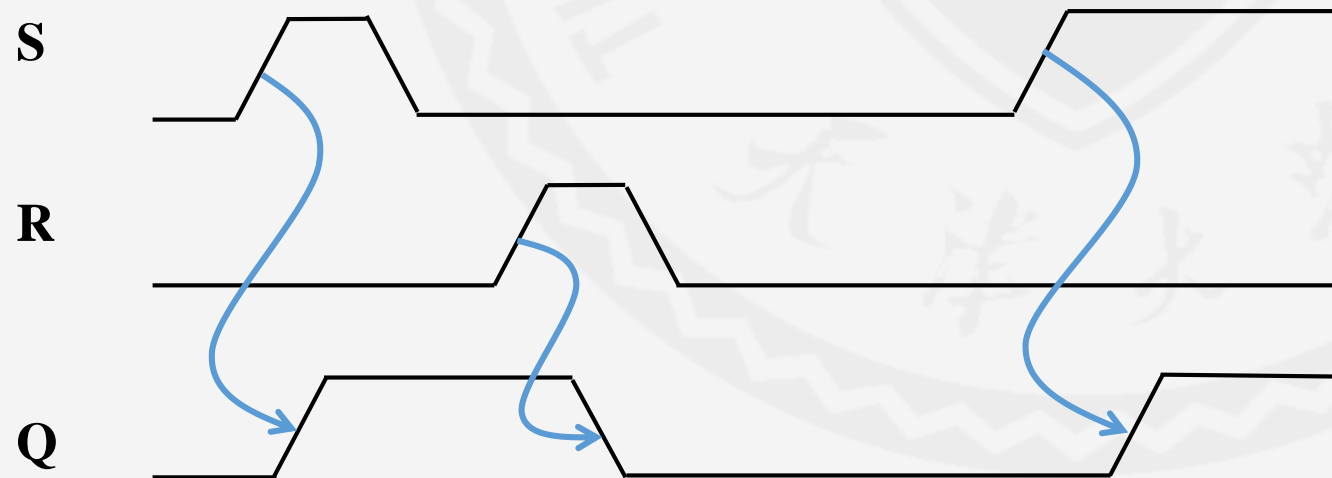
SR Latch

SR 锁存器的真值表

S	R	Q^{n+1}	$\overline{Q^{n+1}}$
0	0	Q^n	$\overline{Q^n}$
0	1	0	1
1	0	1	0
1	1	0	0

波形图

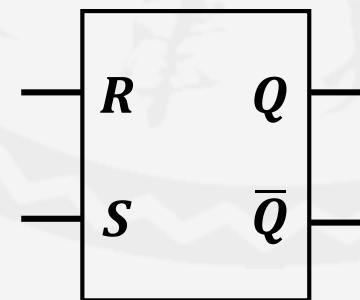
(初始状态假设为0)





SR 锁存器总结

- SR锁存器是一个存储1比特状态的双稳态模块
 - SR分别表示置位 (Set) 和复位 (Reset)
- 通过控制S、R信号的输入，控制锁存器的状态
 - Set 置位，使 $Q = 1$ ($S = 1, R = 0$)
 - Reset 复位，使 $Q = 0$ ($S = 0, R = 1$)
 - 保持，使 $Q^{n+1} = Q^n$ ($S = 0, R = 0$)
 - 禁止输入， $S = R = 1$



SR锁存器的电路符号



锁存器和触发器

Latches & Flip-flops

□ 双稳态电路

□ SR锁存器

□ D锁存器

□ D触发器

□ 寄存器



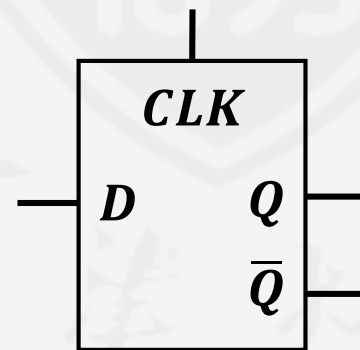


D锁存器

D Latch

D 锁存器的功能

- 包含两个输入端：CLK, D
 - CLK: 控制锁存器状态发生改变的时间
 - D: 数据输入, 控制下一个状态的值
- 功能
 - 当 $CLK = 1$ 时, $Q^n = D$, Q跟随D进行变化, D锁存器是透明的
 - 当 $CLK = 0$ 时, $Q^{n+1} = Q^n$, Q保持原先的状态, D锁存器是不透明的
- D锁存器避免了SR锁存器中S和R信号同时有效而造成的奇怪情况



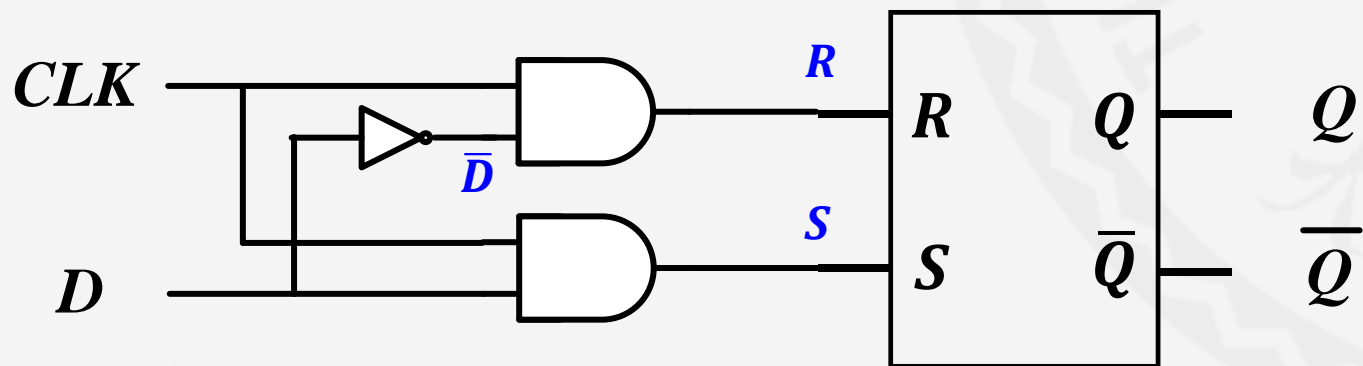
D锁存器的电路符号



D锁存器

D Latch

D 锁存器的实现



CLK	D	\bar{D}	S	R	Q^{n+1}	\bar{Q}^{n+1}
0	X	\bar{X}	0	0	Q^n	\bar{Q}^n
1	0	1	0	1	0	1
1	1	0	1	0	1	0



D锁存器

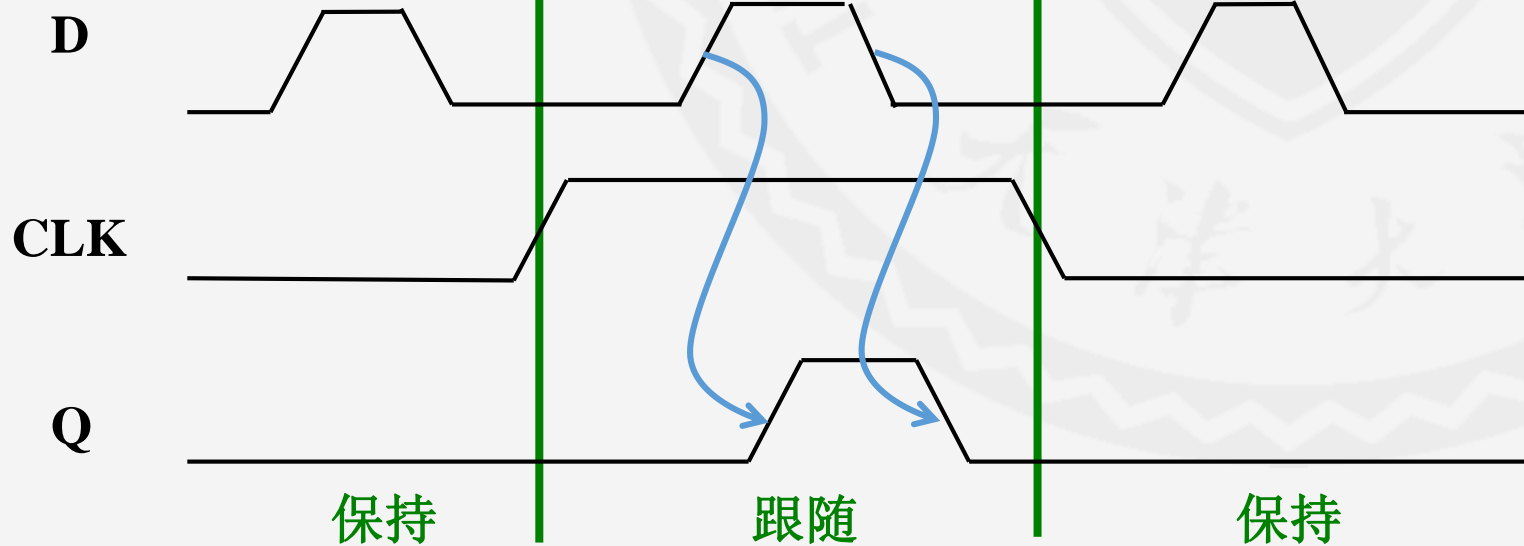
D Latch

D 锁存器的真值表

CLK	D	Q^{n+1}	$\overline{Q^{n+1}}$
0	X	Q^n	$\overline{Q^n}$
1	0	0	1
1	1	1	0

波形图

(初始状态假设为0)





锁存器和触发器

Latches & Flip-flops

□ 双稳态电路

□ SR锁存器

□ D锁存器

□ D触发器

□ 寄存器



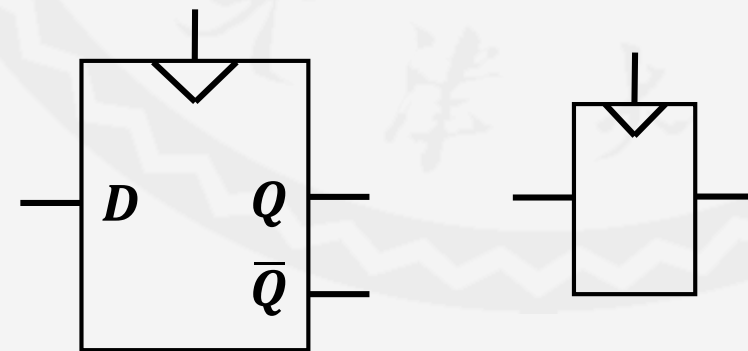


D触发器

D Flip-flops

D 触发器的功能

- 包含两个输入端：CLK, D
- 功能
 - 在CLK的上升沿对D进行采样
 - 当CLK的信号从0到1的瞬间，Q被修改为当前时刻D的值
 - 在其他时刻，Q处于保持状态
 - Q的值只会在CLK上升沿的时刻发生改变
- 这种工作模式被称为边沿触发



D触发器的电路符号

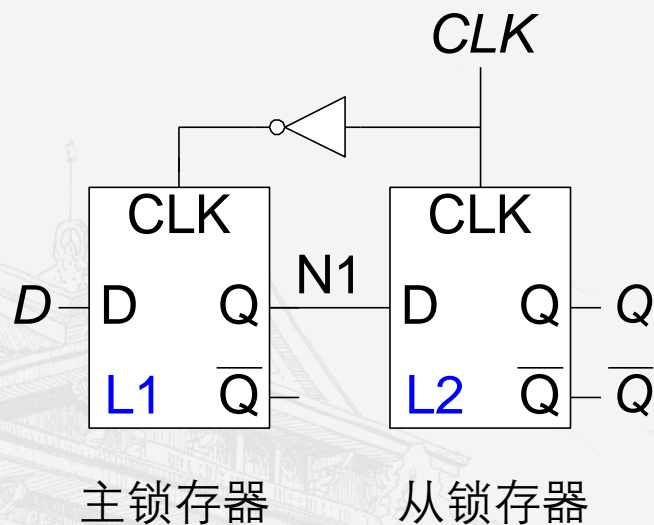


D触发器

D Flip-flops

主从式 D 触发器的实现

- 由两个顺序连接的D锁存器构成 (L1, L2)
- 由一组相反的时钟信号所控制



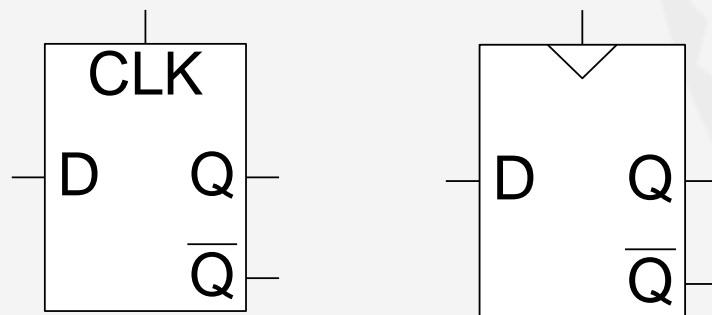
- 当CLK=0时
 - 主锁存器L1是透明的
 - 从锁存器L2是不透明的
 - 主锁存器的状态 (N1) 跟随D变化
- 当CLK=1时
 - 主锁存器L1是不透明的
 - 从锁存器L2是透明的
 - 从锁存器的状态跟随N1变化
- 因此在时钟的上升沿时刻 (CLK 0→1)
 - Q被赋值为D



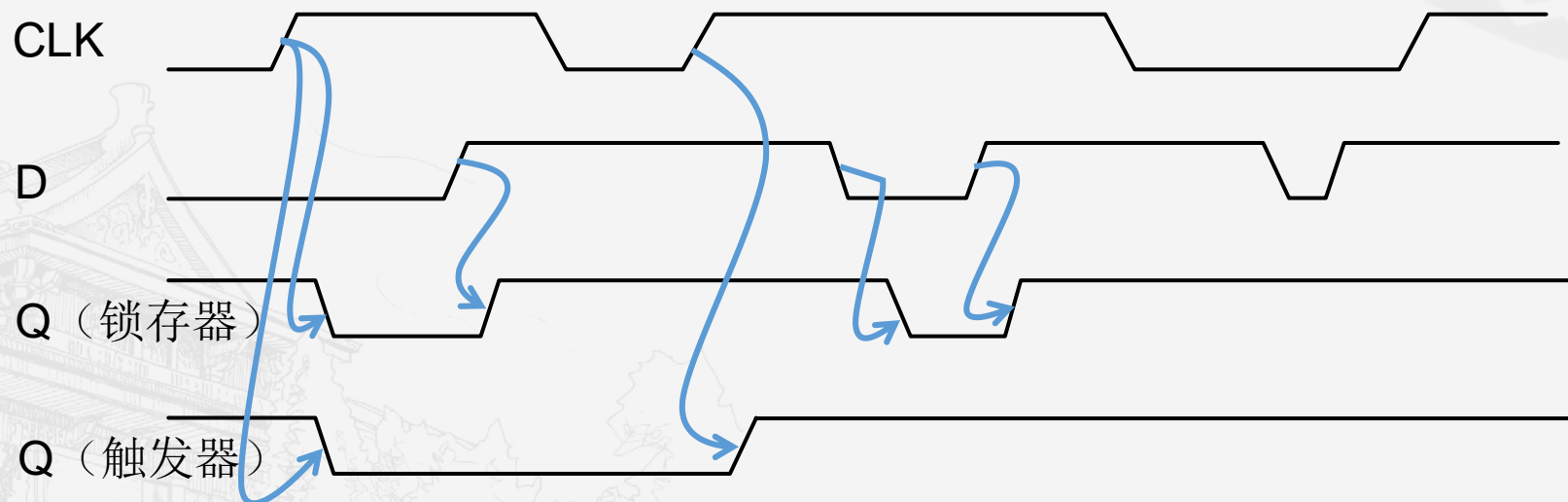
D触发器

D Flip-flops

D 锁存器与 D 触发器



(初始状态假设为1)



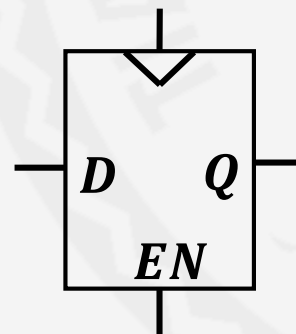


D触发器

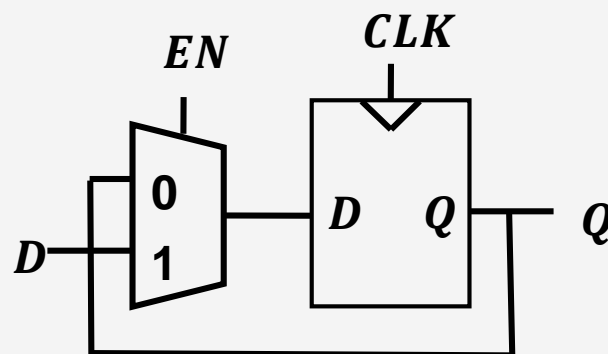
D Flip-flops

带使能端的 D 触发器

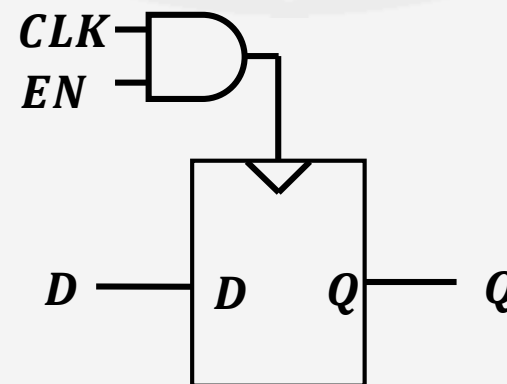
- 三个输入端：CLK, D, EN
 - EN为使能端：用来控制D是否能被触发器存储
- 功能（EN信号高电平有效）
 - EN=1：在时钟上升沿时，Q被更新为当前时刻的D值
 - EN=0：触发器处于保持状态



带使能端D触发器电路符号



电路实现1



电路实现2

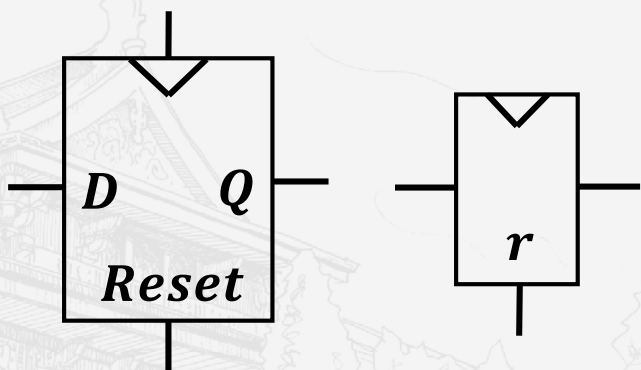


D触发器

D Flip-flops

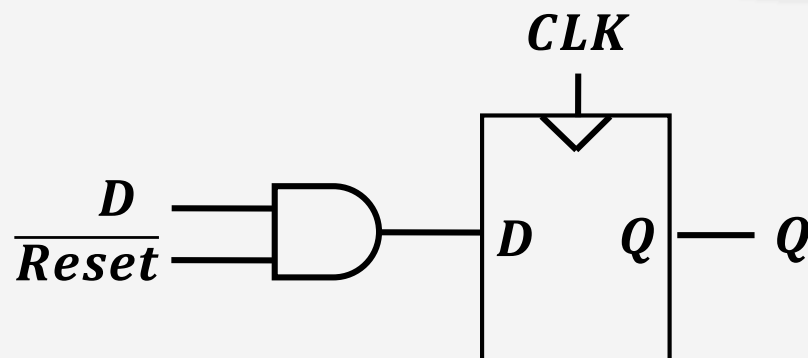
带复位功能的 D 触发器

- 三个输入端：CLK, D, Reset
- 功能（Reset信号高电平有效）
 - Reset = 1: Q被强制设置为0
 - Reset = 0: 触发器正常工作



带复位功能D触发器的电路符号

- 同步复位
 - 只在CLK上升沿进行复位
- 异步复位
 - 只要Reset信号有效即可复位



同步复位D触发器电路实现

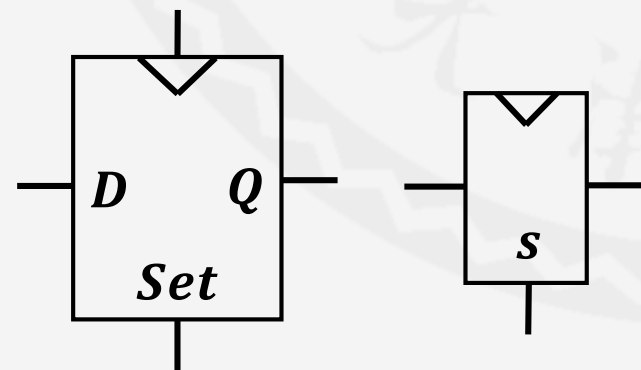


D触发器

D Flip-flops

带置位功能的 D 触发器

- 三个输入端：CLK, D, Set
- 功能（Reset信号高电平有效）
 - Set = 1: Q被强制设置为1
 - Set = 0: 触发器正常工作



带置位功能D触发器的电路符号



锁存器和触发器

Latches & Flip-flops

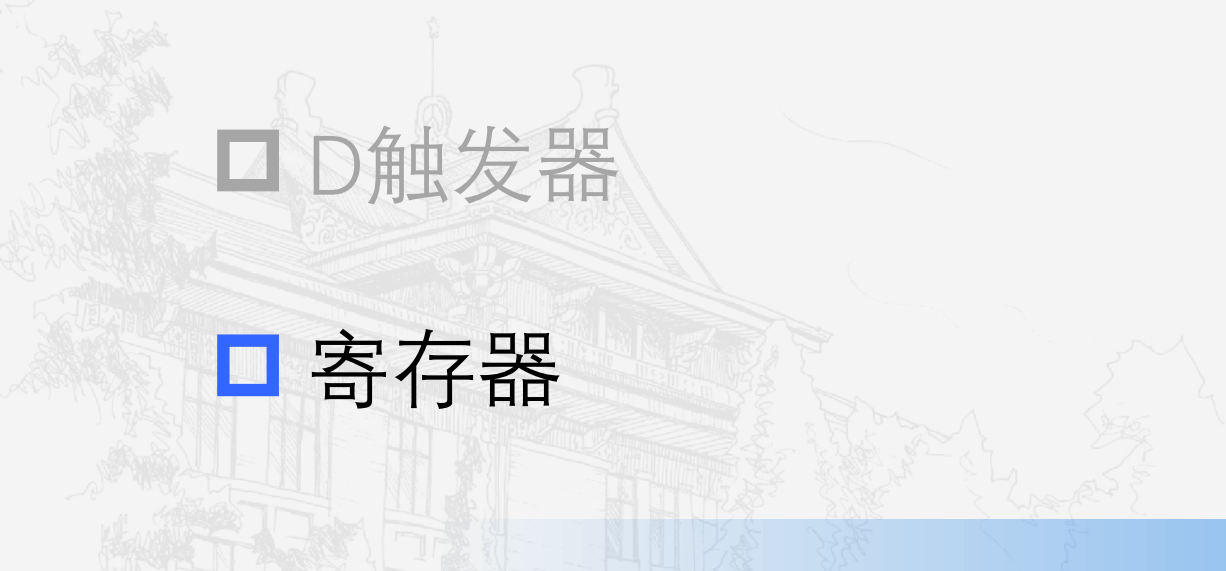
□ 双稳态电路

□ SR锁存器

□ D锁存器

□ D触发器

□ 寄存器



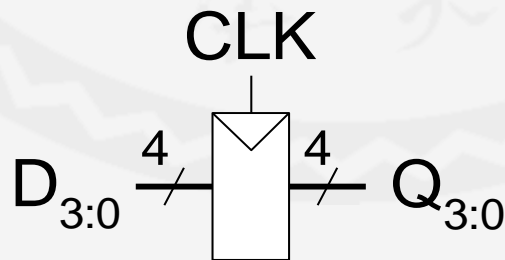
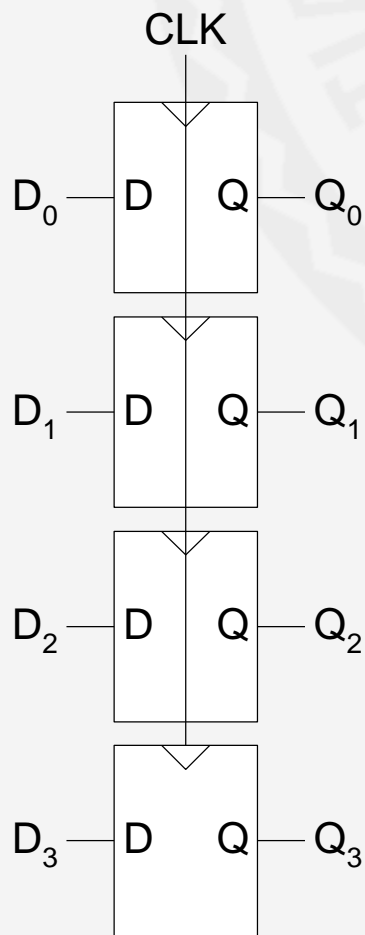


寄存器

Register

寄存器

- 一个N位寄存器由一个共享的CLK输入和N个D触发器组成
- 寄存器的所有位同时被更新
- 是大多数时序电路中的关键组件

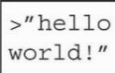


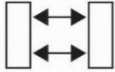
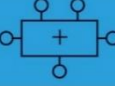

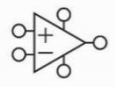
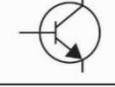





本章内容

Topic

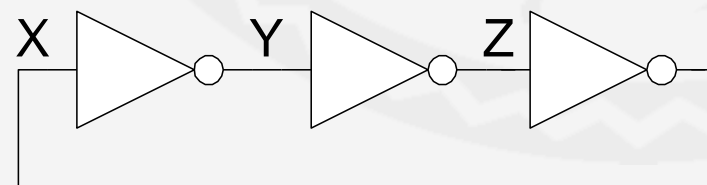
- 引言
- 锁存器和触发器
- 同步逻辑设计**
- 有限状态机
- 时序逻辑中的时序问题
- 时序逻辑模块
- 存储器阵列

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	



时序逻辑电路

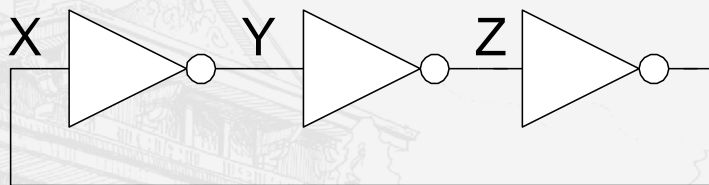
- 时序逻辑电路：所有不是组合逻辑的电路
 - 电路的输出不能简单地通过观察当前输入来确定
- 一个有问题的电路
 - 没有输入
 - 有1-3个输出



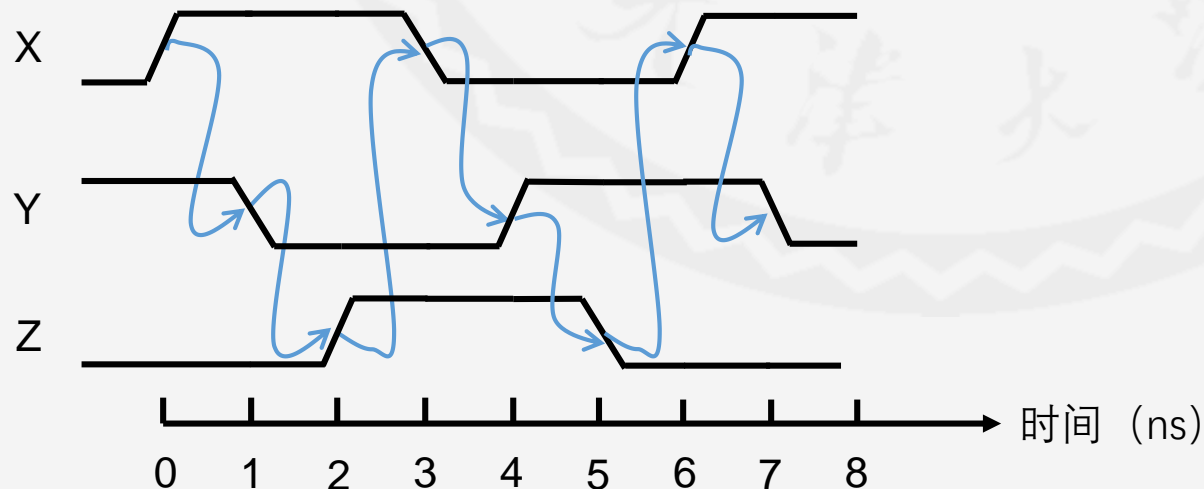


非稳态电路

- 电路的特征：
 - 非稳态电路
 - 输出结果周期性地翻转
 - 电路中具有回路，输出端反馈至输入端
- 环形振荡器



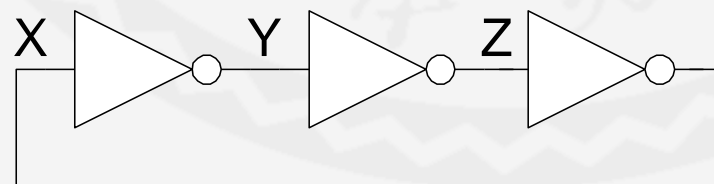
假设：每个反相器有1ns的传播延迟





同步时序电路

- 在信号传播路径中插入寄存器以断开电路中环路
 - 使电路转变成组合逻辑电路和寄存器的集合
- 寄存器包含系统的状态
- 状态仅在时钟边沿（上升沿或下降沿）到达时发生改变
 - 即状态**同步于**时钟信号
- 如果时钟足够慢，使得在下一个时钟沿到达之前，输入到寄存器的信号都可以稳定下来，则所有的冒险将被消除





同步时序电路 (cont.)

- 一个时序电路包含一组有限的离散状态 $\{S_0, S_1, S_2, \dots S_{k-1}\}$
- 同步时序电路有一个时钟输入
- 上升沿表示电路状态转变发生的时间
 - 当前状态（现态）：当前系统的状态
 - 下一个状态（次态）：下一个时钟沿后系统将进入的状态
- 功能规范：
 - 描述当前状态和输入的各种组合所对应的下一个状态和输出
- 时序规范：
 - 建立时间、保持时间



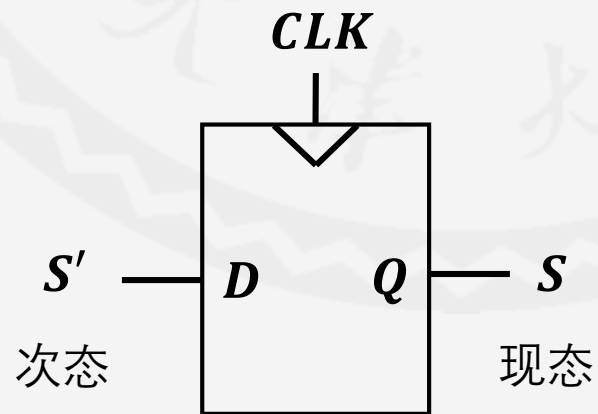
同步时序逻辑电路的组成规则

- 组成规则：
 - 电路中的模块或者是寄存器或者是组合逻辑电路
 - 模块中至少包含一个寄存器
 - 所有的寄存器都共用同一个时钟信号
 - 电路的每个环路中至少包含一个寄存器
- 两类常见的同步时序逻辑电路
 - 有限状态机 (FSM)
 - 流水线



一个简单的同步时序电路

- 一个D触发器是一个最简单的同步时序电路
 - 包含一个输入 D
 - 一个时钟 CLK
 - 一个输出 Q
 - 两个状态 $\{0, 1\}$
- D触发器功能规范
 - 下一个状态是 D
 - 输出 Q 是当前状态

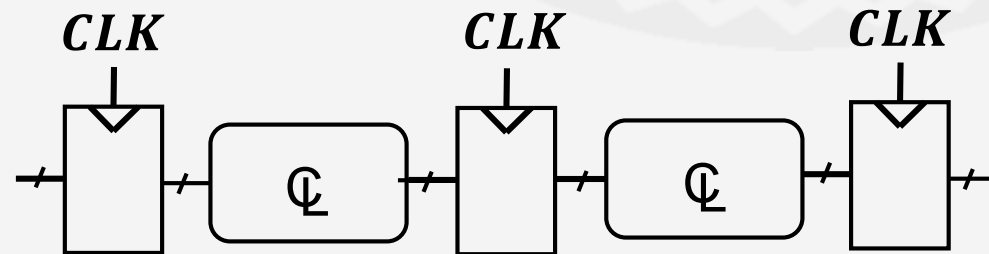
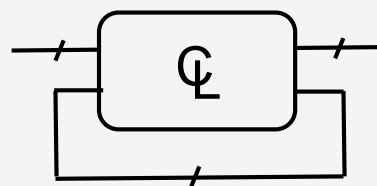
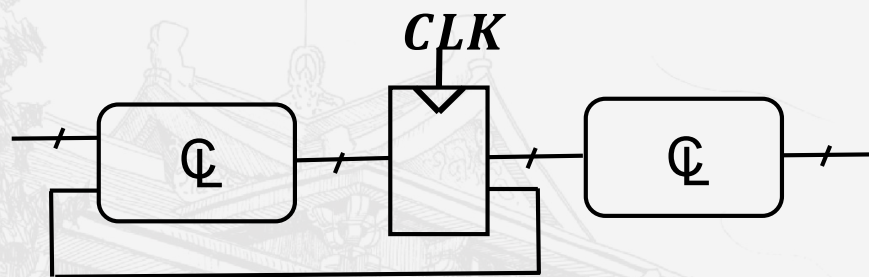
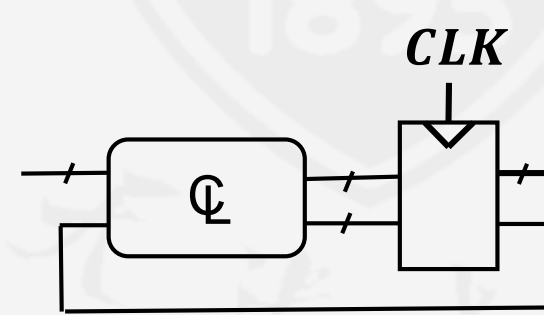
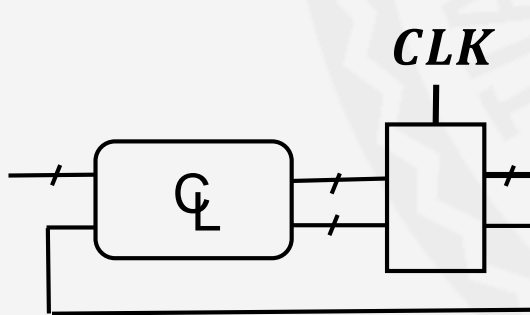
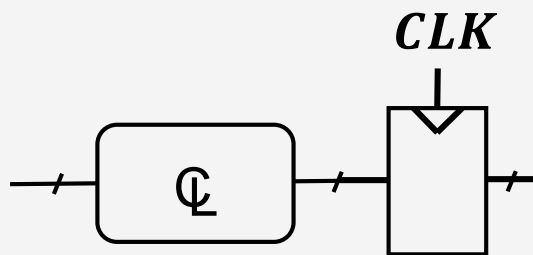
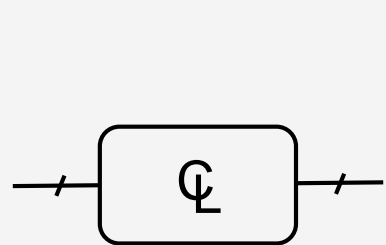




同步逻辑设计

Synchronous Logic Design

思考：下列哪些电路是同步时序电路？





异步时序电路

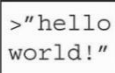


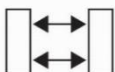
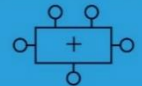

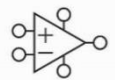


- 非同步的时序电路称为异步时序电路
- 理论上，异步时序电路设计比同步时序电路设计更通用
 - 系统的时序不由时钟控制的寄存器所约束
- 实际上，几乎所有的系统本质上都是同步的
 - 同步时序电路比异步时序电路更容易设计



本章内容

Topic

- 引言
- 锁存器和触发器
- 同步逻辑设计
- 有限状态机**
- 时序逻辑中的时序问题
- 时序逻辑模块
- 存储器阵列

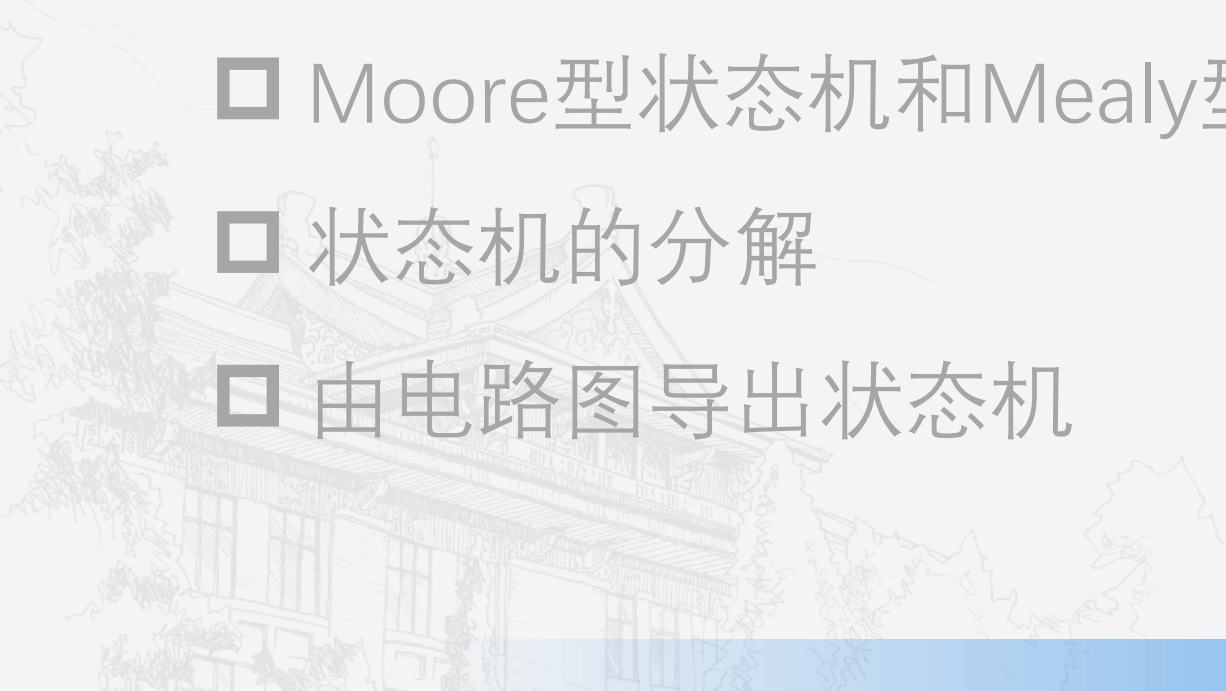
Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	



有限状态机

Finite State Machines

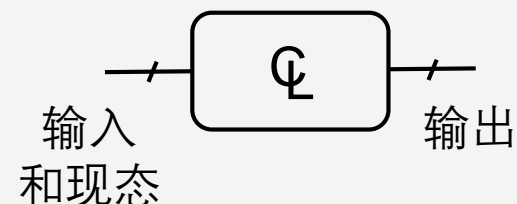
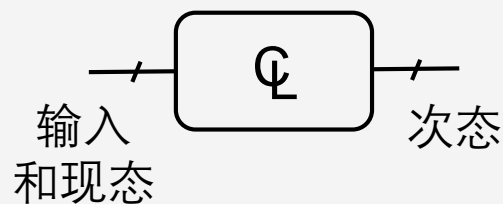
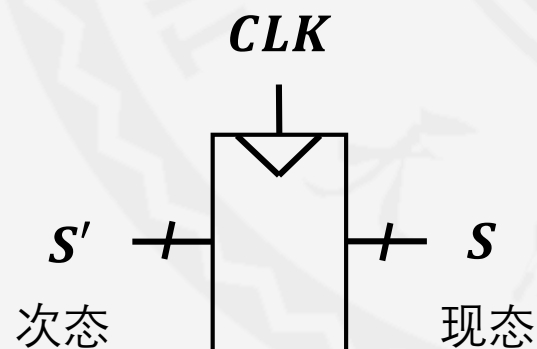
- 基本概念
- 有限状态机设计实例
- 状态编码
- Moore型状态机和Mealy型状态机
- 状态机的分解
- 由电路图导出状态机





有限状态机的结构

- 由以下模块组成：
 - 状态寄存器
 - 存储当前时刻的状态
 - 状态在下一个有效时钟沿发生改变
 - 组合逻辑
 - 计算下一个状态
 - 计算电路的输出

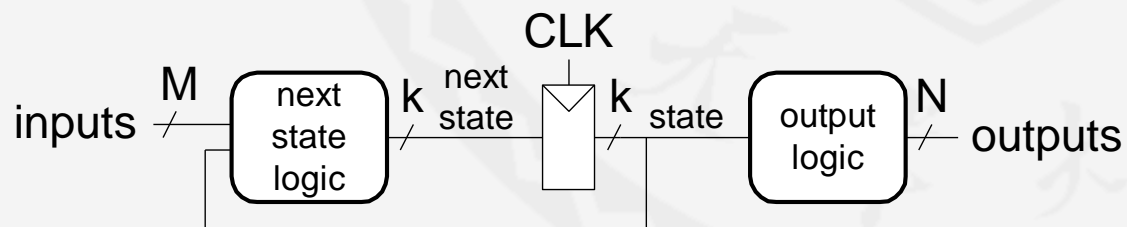




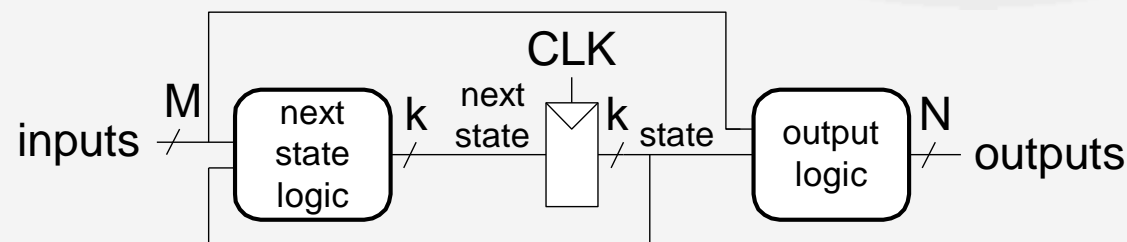
Moore 机和 Mealy 型有限状态机

- 次态由现态和当前输入共同决定
- 根据输出逻辑的不同，可以将有限状态机分为两类
 - Moore型：输出仅由当前时刻状态所决定
 - Mealy型：输出由当前时刻状态和输入共同决定

Moore FSM



Mealy FSM

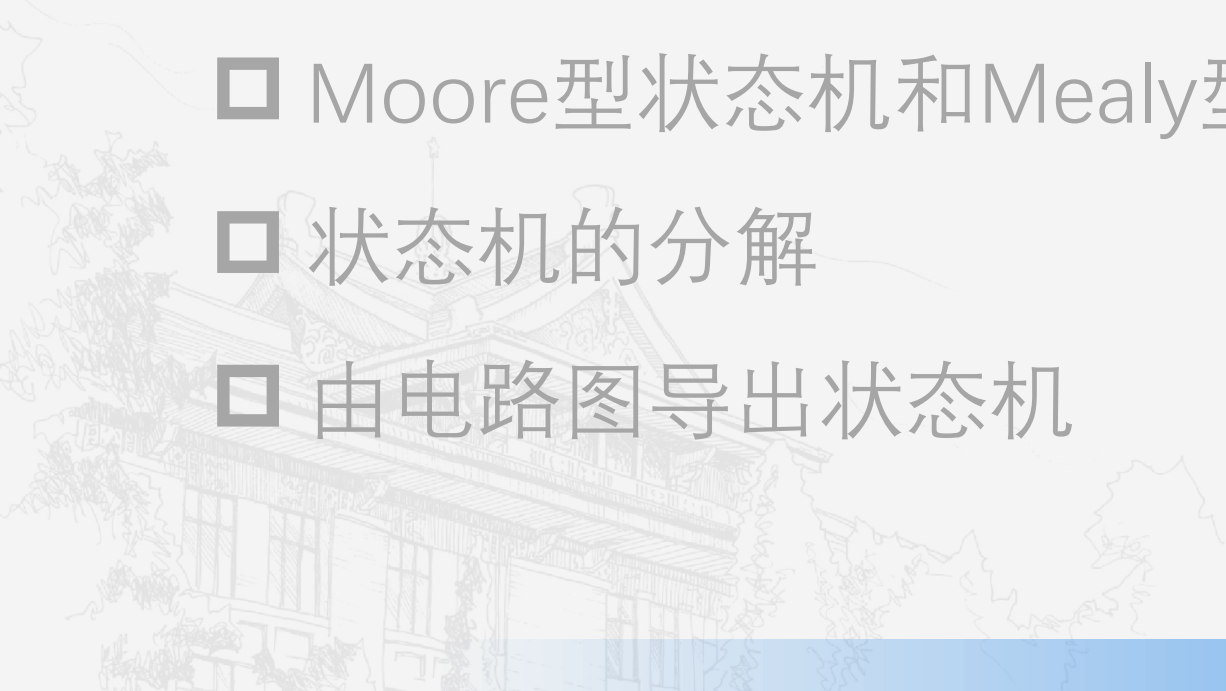




有限状态机

Finite State Machines

- 基本概念
- 有限状态机设计实例
- 状态编码
- Moore型状态机和Mealy型状态机
- 状态机的分解
- 由电路图导出状态机



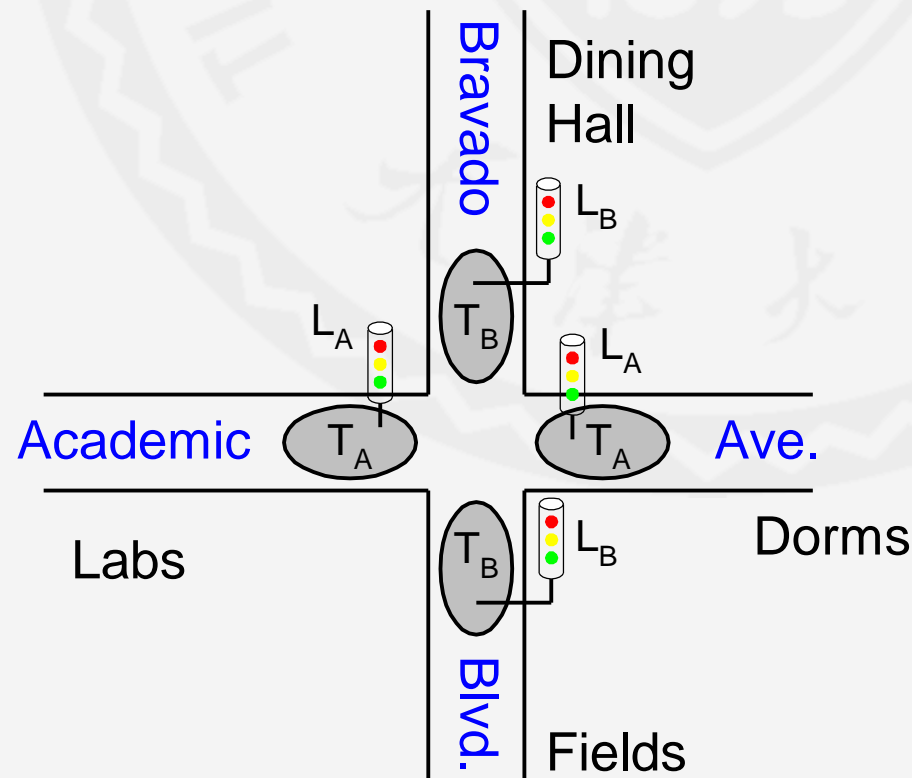


有限状态机设计实例

FSM Design Example

例：交通灯控制器设计

- 输入：两个交通传感器 T_A 和 T_B
 - 当路上有人出现时，传感器返回 TRUE
 - 否则返回 FALSE
- 输出：两个交通灯 L_A 和 L_B
 - 红色、黄色和绿色
- 周期为 5s 的时钟，在每一个时钟沿到达时，灯根据交通传感器来改变
- 一个复位按键，可以使交通灯控制器回到初始状态





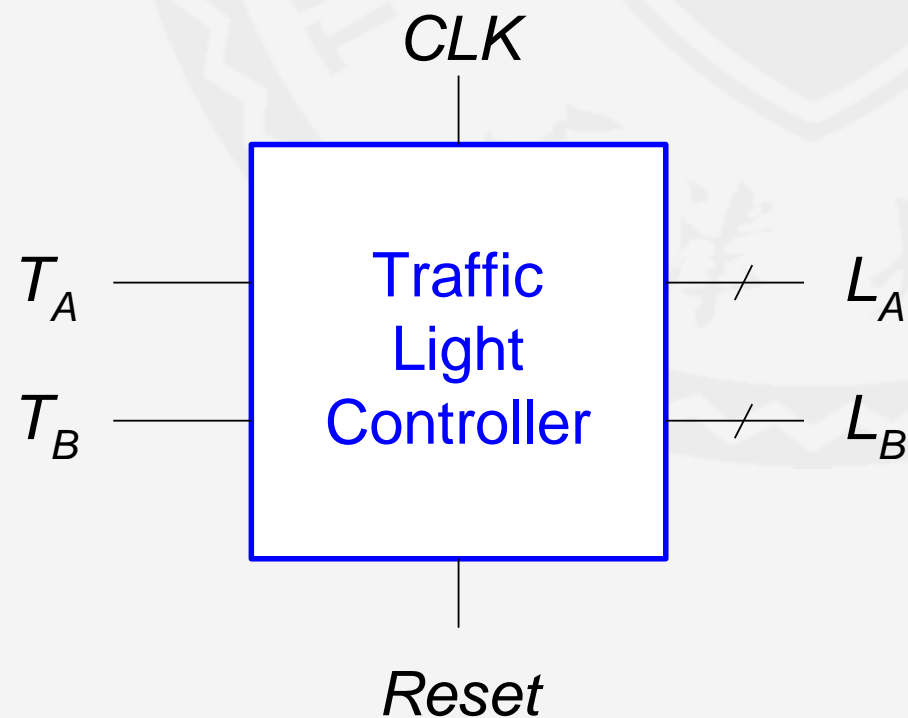
有限状态机设计实例

FSM Design Example

有限状态机黑盒视图

■ 输入: CLK , $Reset$, T_A , T_B

■ 输出: L_A , L_B



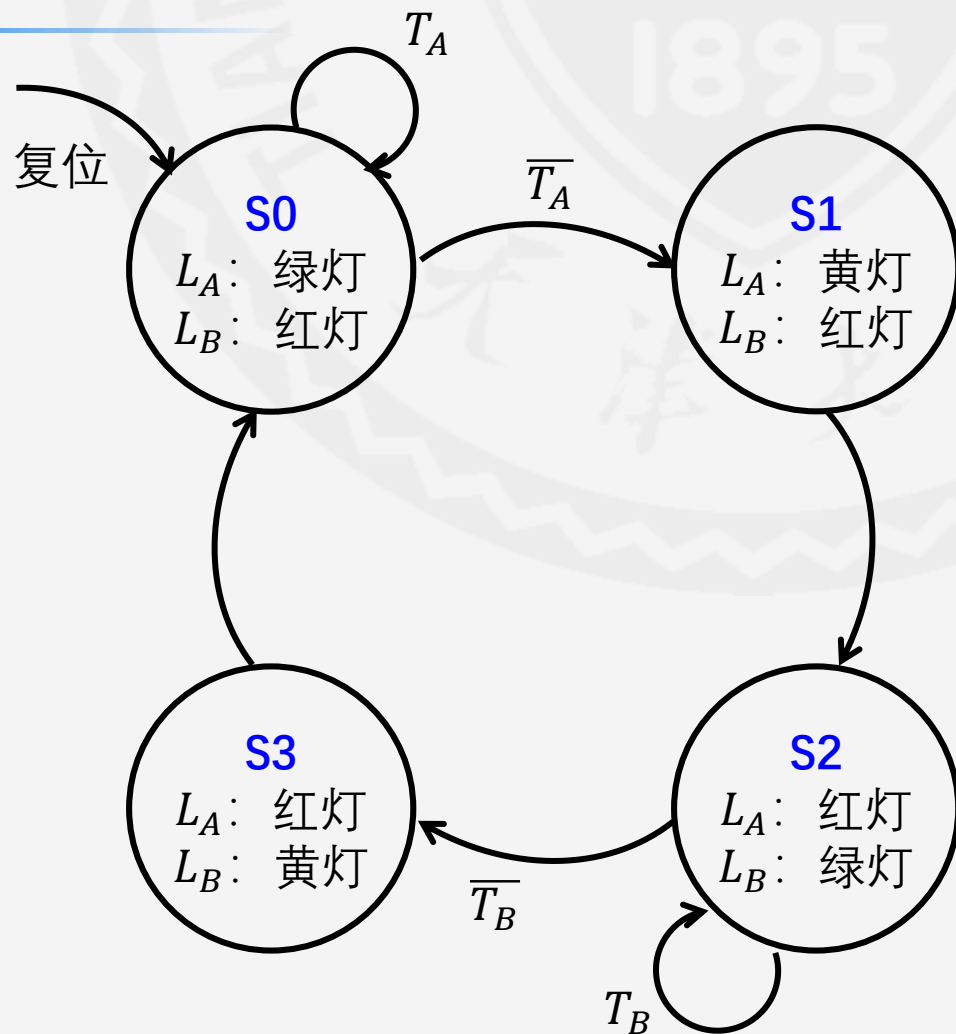


有限状态机设计实例

FSM Design Example

状态转换图

- 图中的每一个**圆圈**代表一个状态
- 图中的每一个**圆弧**代表两个状态之间的转换
- 圆弧上的项表示实现状态转换所需要的输入
- 状态转换发生在时钟有效沿产生的时刻
- Moore型状态机：输出信息标在状态（圆圈）中

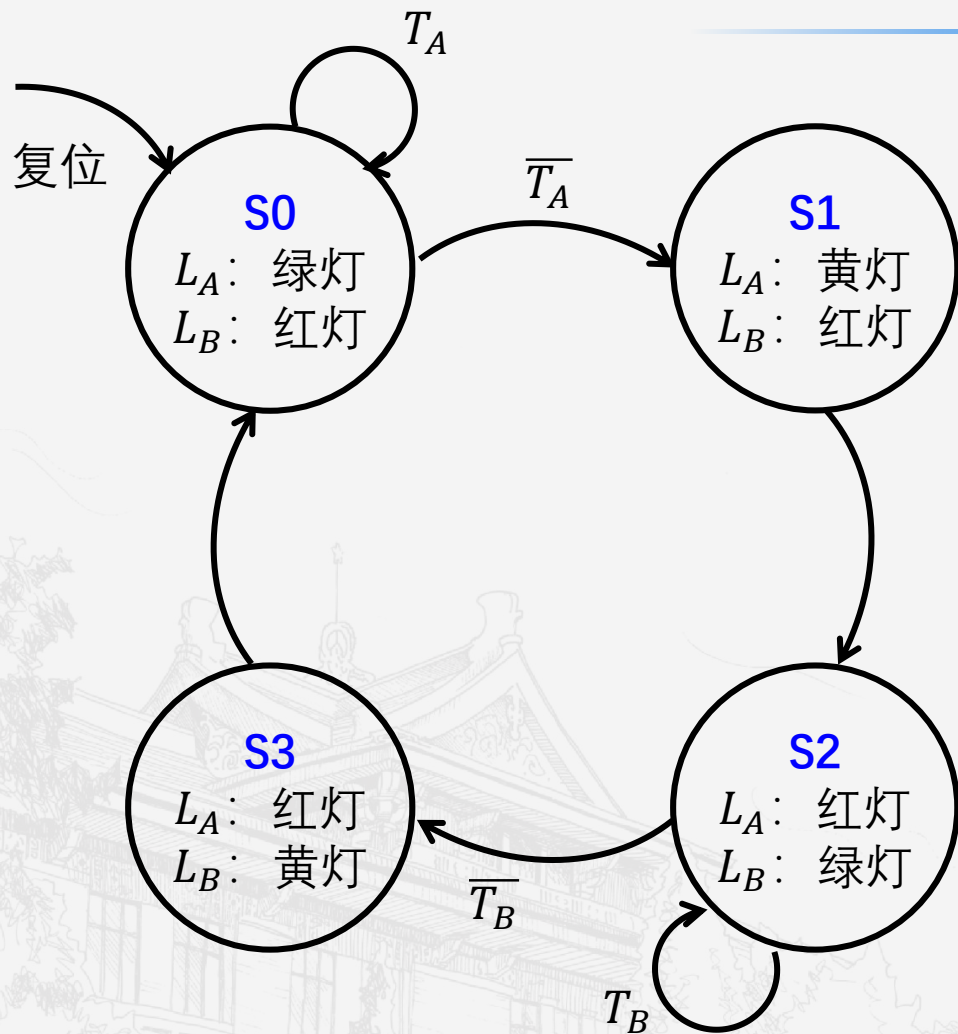




有限状态机设计实例

FSM Design Example

状态转换表



现态	输入		次态
S	T_A	T_B	S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0



有限状态机设计实例

FSM Design Example

对状态进行编码

状态	编码
S0	00
S1	01
S2	10
S3	11

现态		输入		次态	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

$$S'_1 = S_1 \oplus S_0$$

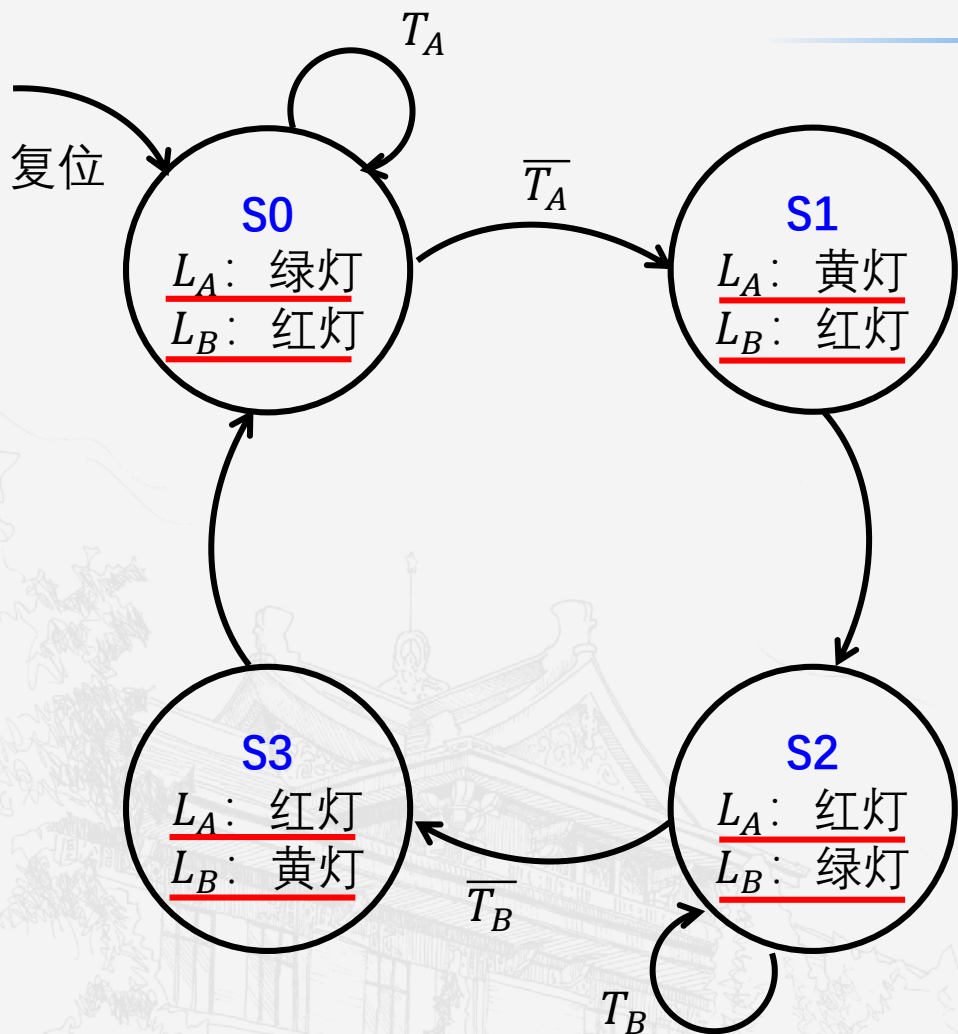
$$S'_0 = S_1 S_0 T_A + S_1 S_0 T_B$$



有限状态机设计实例

FSM Design Example

输出表



输出	编码
绿	00
黄	01
红	10

现态		输出			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

$$L_{A1} = S_1$$

$$L_{A0} = \bar{S}_1 S_0$$

$$L_{B1} = \bar{S}_1$$

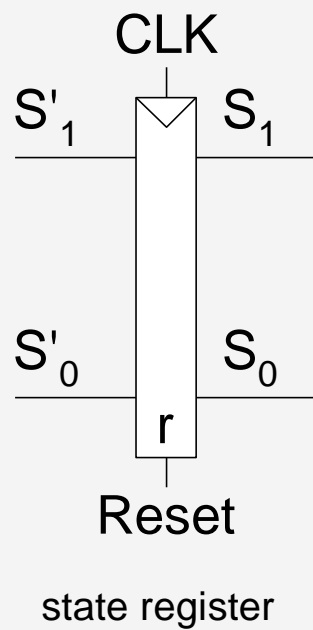
$$L_{B0} = S_1 S_0$$



有限状态机设计实例

FSM Design Example

原理图：寄存器

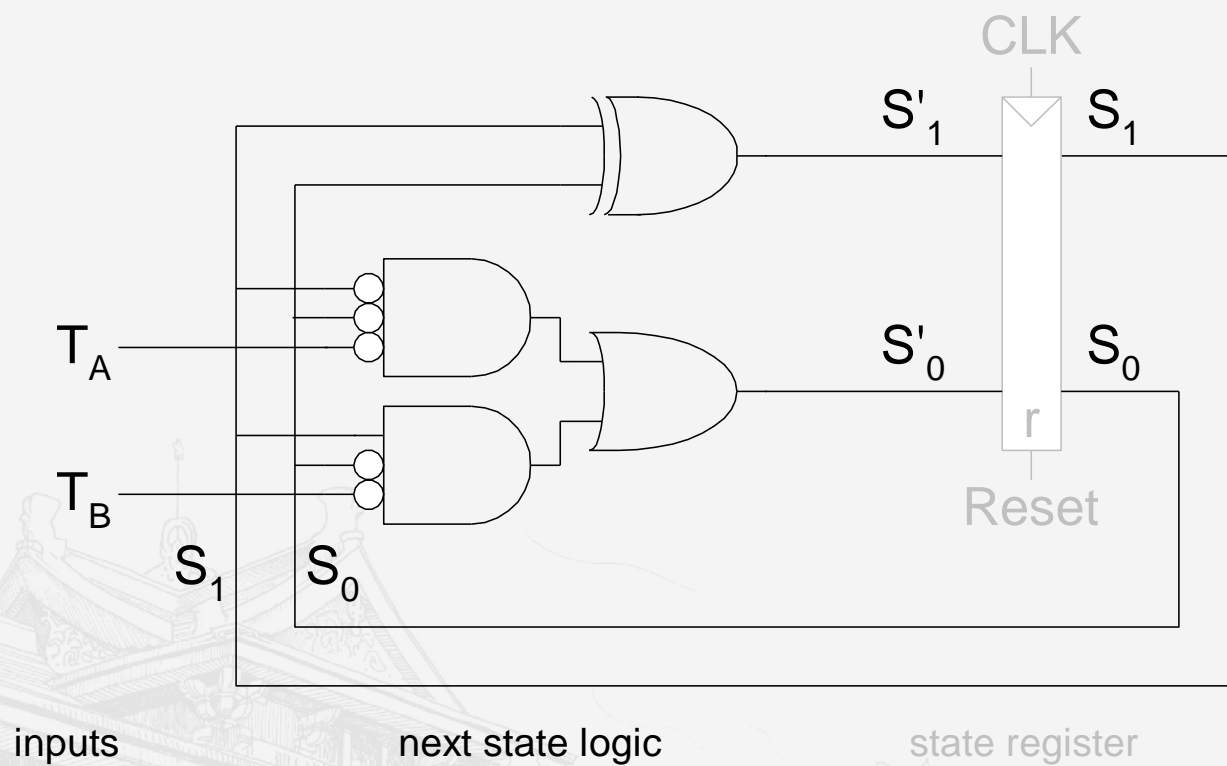




有限状态机设计实例

FSM Design Example

原理图：次态逻辑



$$S'_1 = S_1 \oplus S_0$$

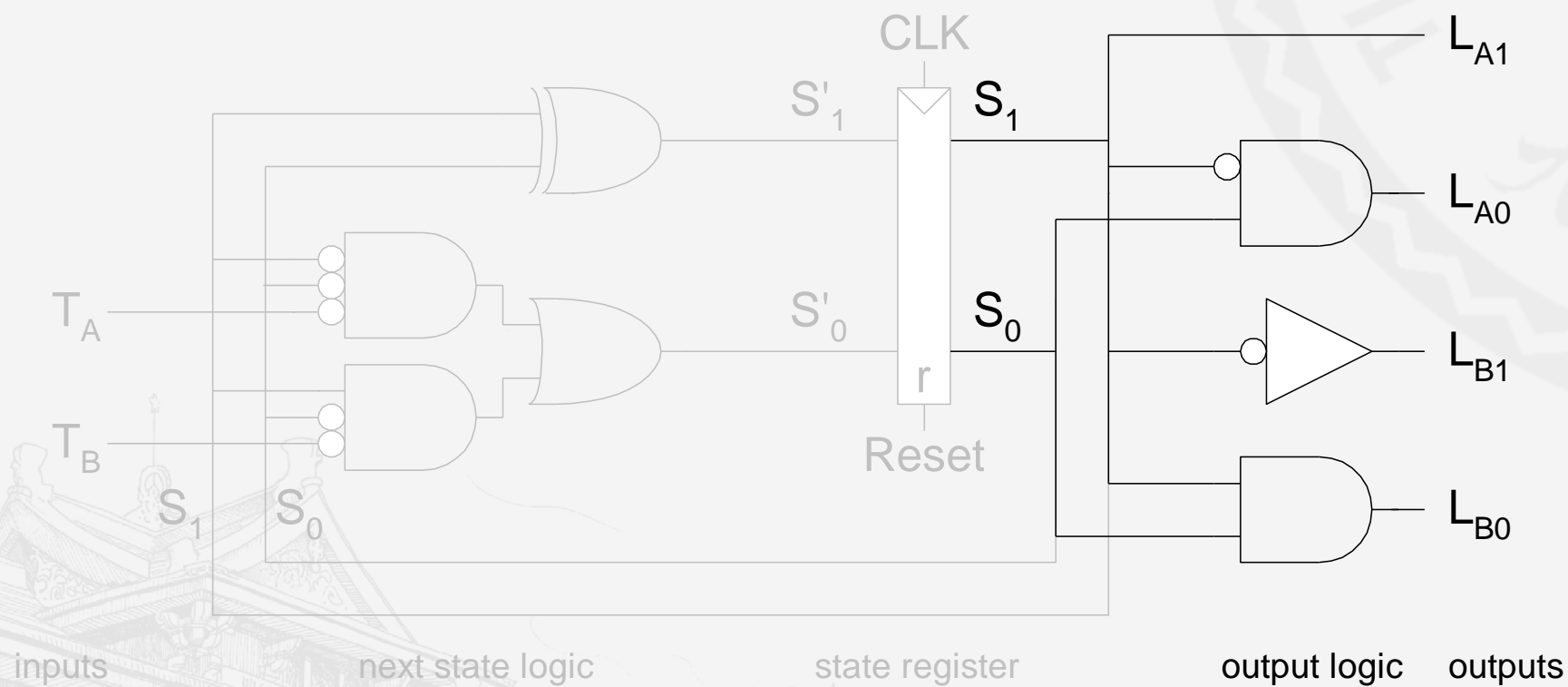
$$S'_0 = S_1 S_0 T_A + S_1 S_0 T_B$$



有限状态机设计实例

FSM Design Example

原理图：输出逻辑



$$L_{A1} = S_1$$

$$L_{A0} = S_1 S_0$$

$$L_{B1} = S_1$$

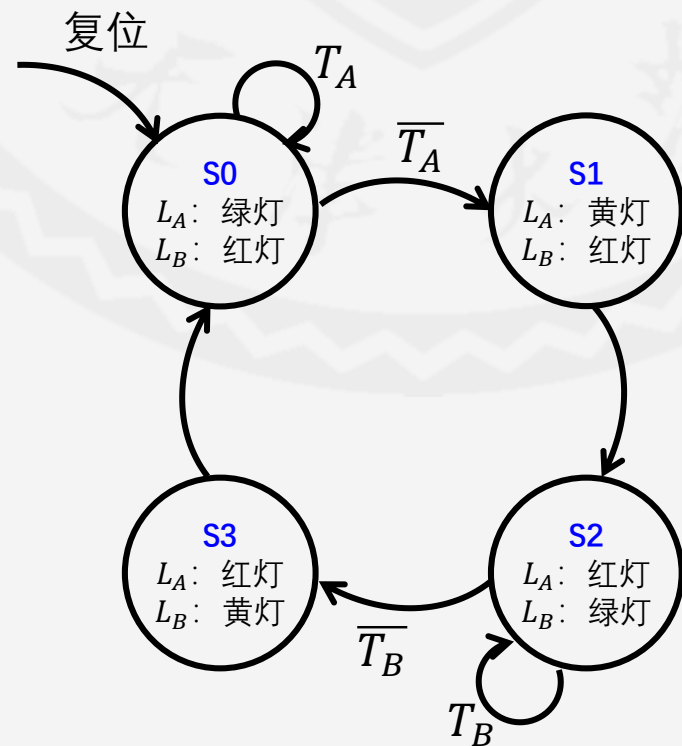
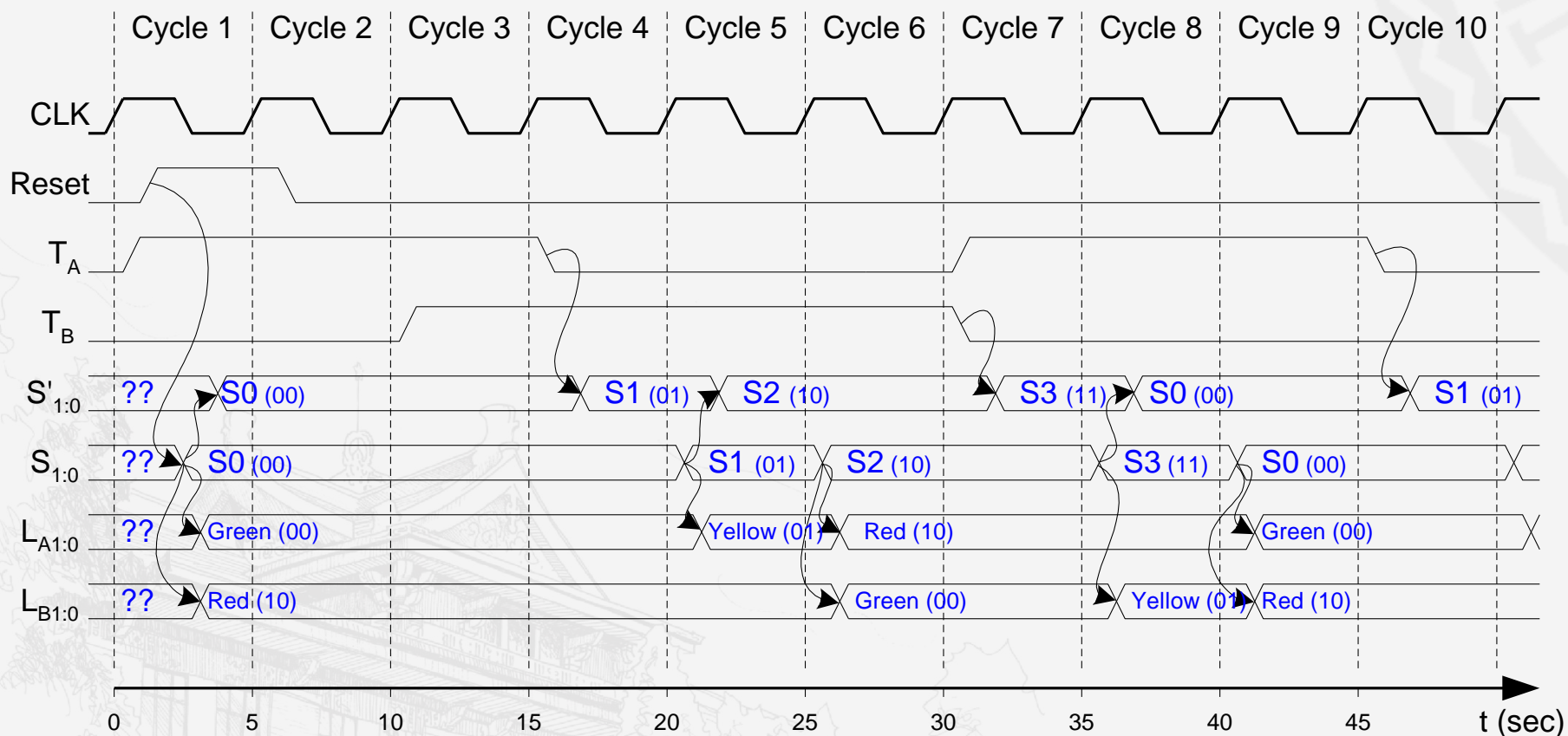
$$L_{B0} = S_1 S_0$$



有限状态机设计实例

FSM Design Example

时序图





Moore型有限状态机设计方法

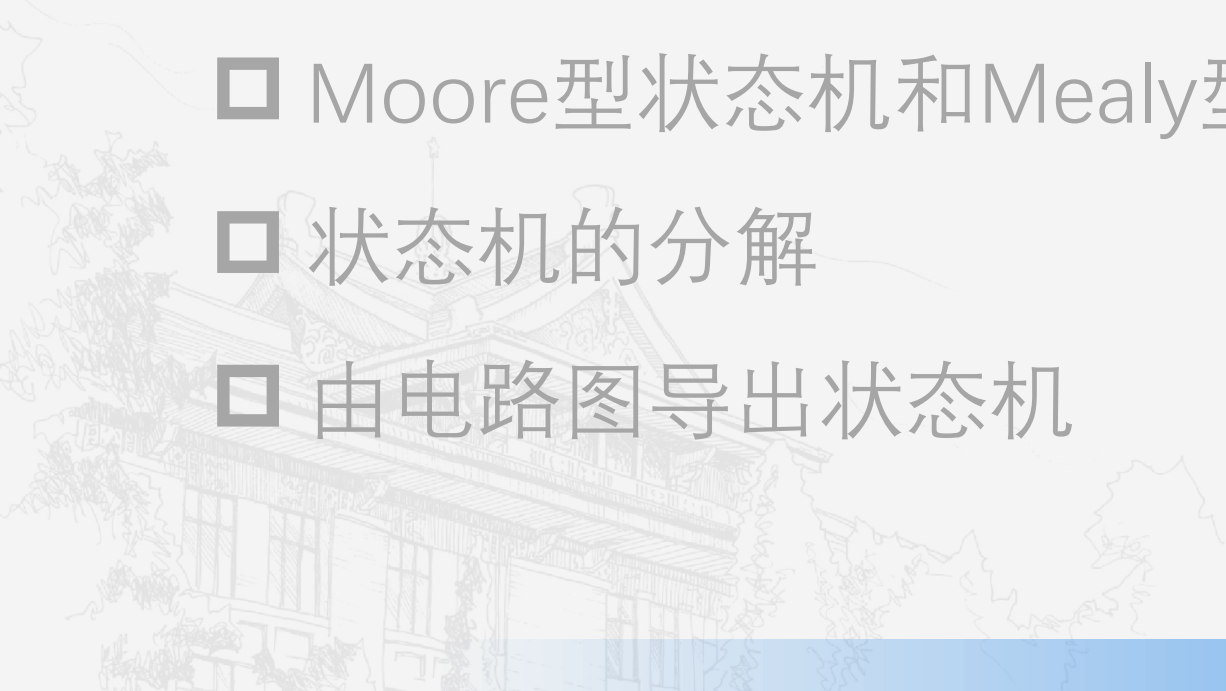
- ① 根据问题进行抽象，确定输入输出以及对应的逻辑含义
- ② 画出状态转换图
- ③ 列出状态转换表
- ④ 对状态进行编码，并列出次态方程
- ⑤ 列出输出表
- ⑥ 对输出进行编码，并列出输出方程
- ⑦ 绘制原理图



有限状态机

Finite State Machines

- 基本概念
- 有限状态机设计实例
- 状态编码**
- Moore型状态机和Mealy型状态机
- 状态机的分解
- 由电路图导出状态机





状态编码

- 不同的状态编码和输出编码会产生不同的电路
- 进行合理的编码，使之能产生一个逻辑门数最少且传播延迟最短的电路
- 目前没有一种简单的办法可以找到
- 可以通过计算机辅助设计（CAD）工具来解决
- 常见的状态编码有两种
 - 二进制编码（Binary encoding）
 - 独热编码（One-hot encoding）



状态编码 (cont.)

■ 二进制编码

- 例如：4种状态，对应的编码为：00, 01, 10, 11

■ 独热编码

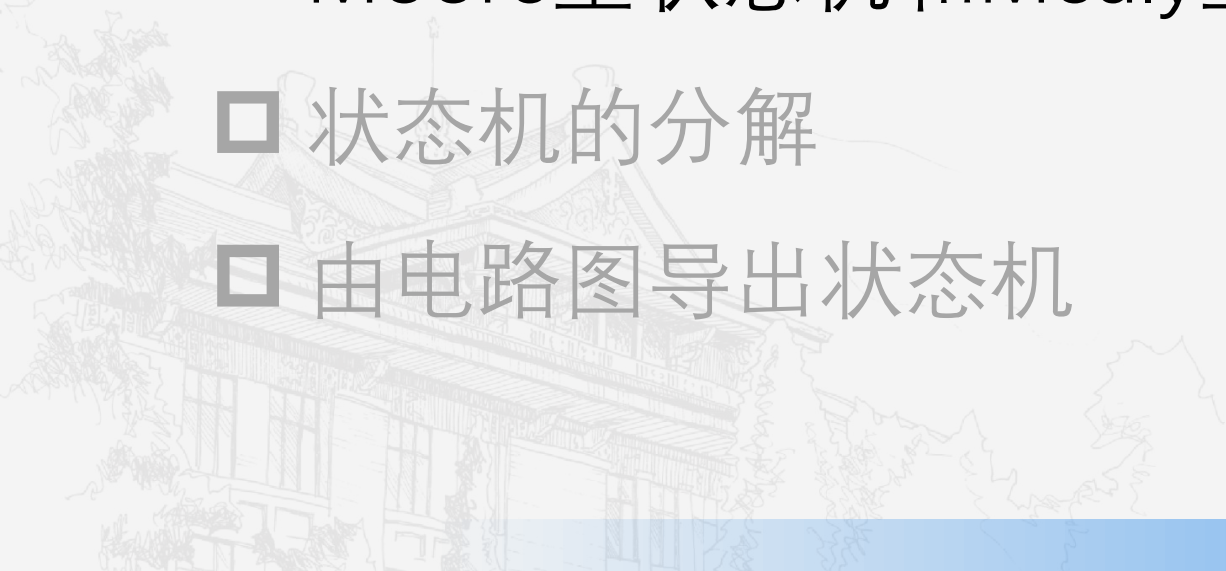
- 状态的每一位 (1 bit) 表示一种状态
- 任何时候只能有一位是“热的” (TRUE 或 1)
- 例如：4种状态，对应的编码为：0001, 0010, 0100, 1000
- 需要使用更多的触发器
- 相比于二进制编码，次态逻辑和输出逻辑实现起来更加的简单



有限状态机

Finite State Machines

- 基本概念
- 有限状态机设计实例
- 状态编码
- Moore型状态机和Mealy型状态机
- 状态机的分解
- 由电路图导出状态机





Moore型状态机和Mealy型状态机

Moore and Mealy Machines

Moore型状态机 vs. Mealy 型状态机

- 例：一个蜗牛在一条写满0和1的纸带上爬行。
当它爬过的最后两位是01时，蜗牛会
对着你微笑。请设计一个Moore型状态机和
Mealy型状态机来模拟蜗牛的行为。





Moore型状态机和Mealy型状态机

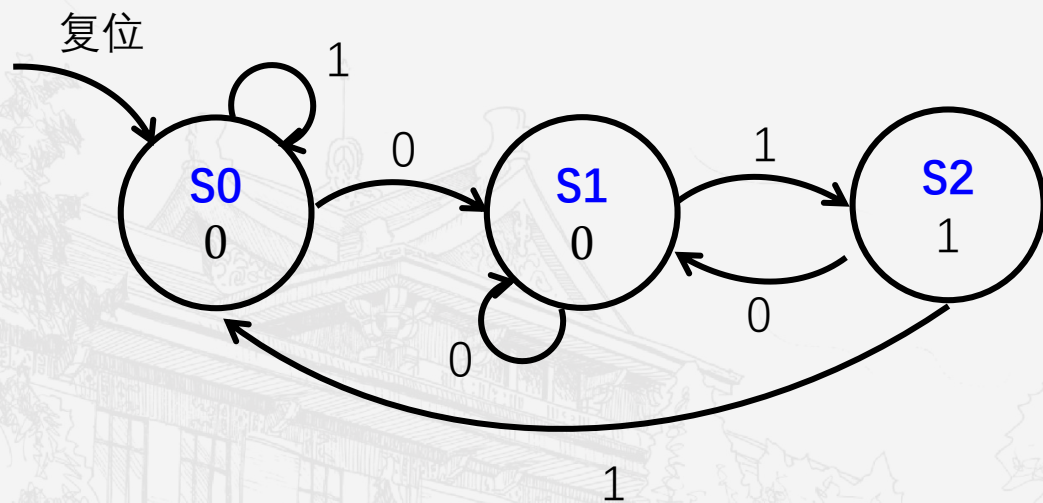
Moore and Mealy Machines

状态转换图

Moore 型状态机

输出只与当前状态有关

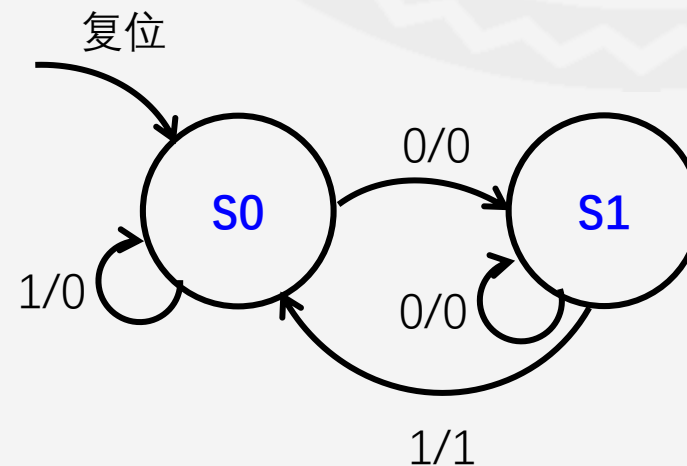
输出画在圆圈中



Mealy 型状态机

输出与当前状态和当前输入有关

输出标记在圆弧上：输入/输出

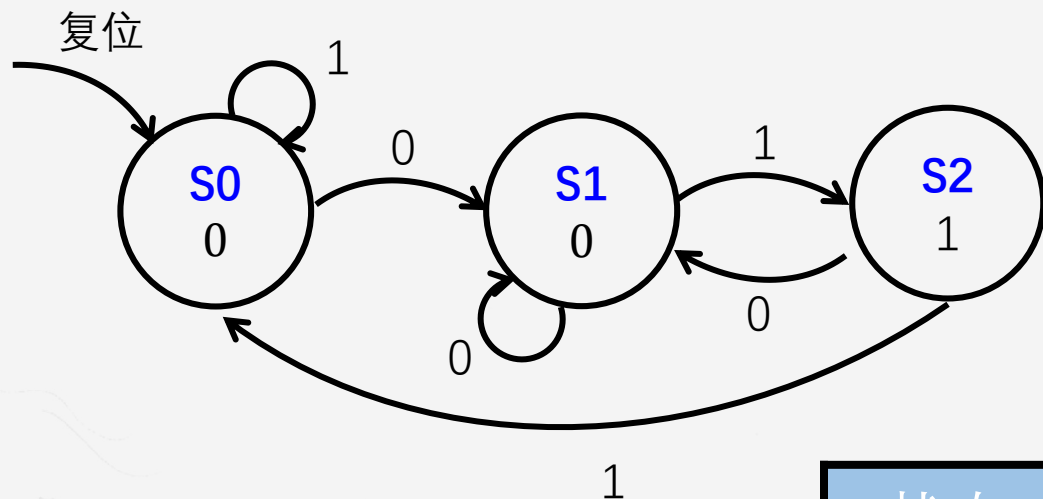




Moore型状态机和Mealy型状态机

Moore and Mealy Machines

Moore型状态机的状态转换表



$$S'_1 = S_0 A$$

$$S'_0 = \bar{A}$$

状态	编码
S0	00
S1	01
S2	10

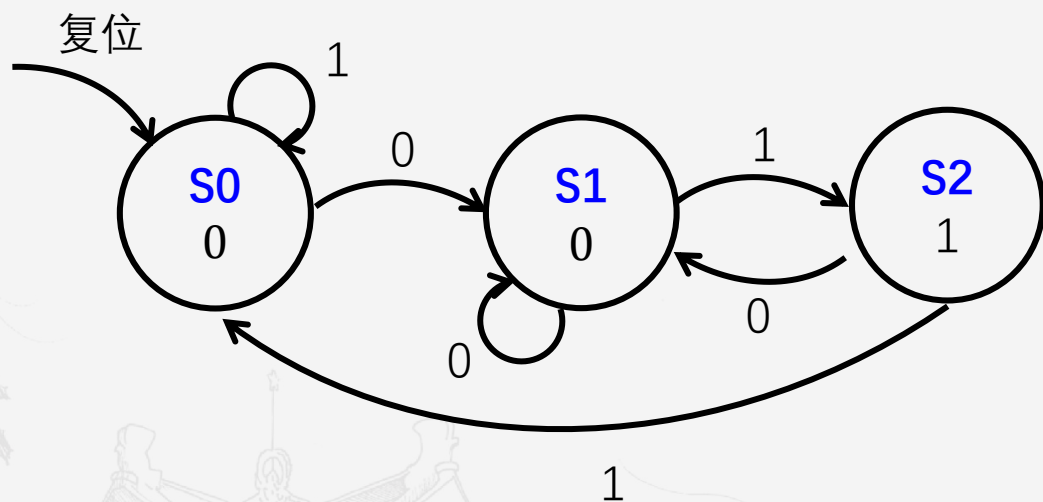
现态		输入 A	次态	
S_1	S_0		S'_1	S'_0
0	0	0	0	1
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0



Moore型状态机和Mealy型状态机

Moore and Mealy Machines

Moore型状态机的输出表



现态		输出
S_1	S_0	Y
0	0	0
0	1	0
1	0	1

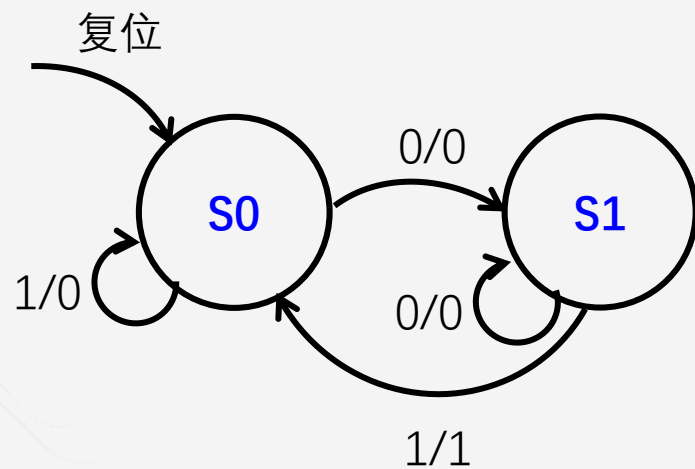
$$Y = S_1$$



Moore型状态机和Mealy型状态机

Moore and Mealy Machines

Mealy型状态机状态转换表和输出表



现态	编码
S0	0
S1	1

现态	输入	次态	输出
S_0	A	S'_0	Y
0	0	1	0
0	1	0	0
1	0	1	0
1	1	0	1

$$S'_0 = \bar{A}$$

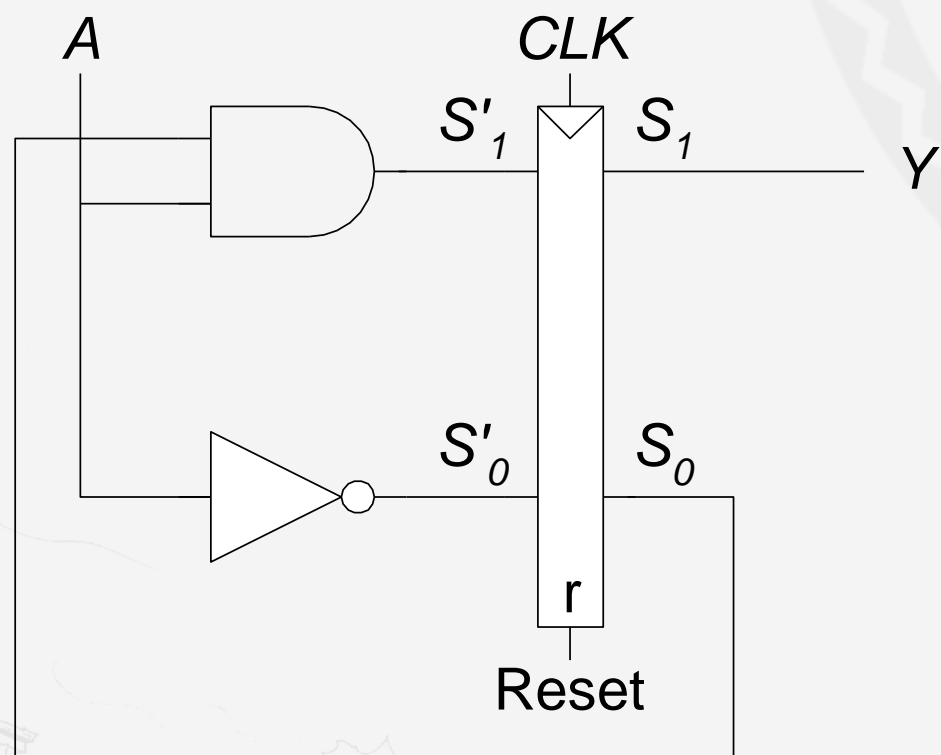
$$Y = S_0 A$$



Moore型状态机和Mealy型状态机

Moore and Mealy Machines

Moore 型状态机原理图

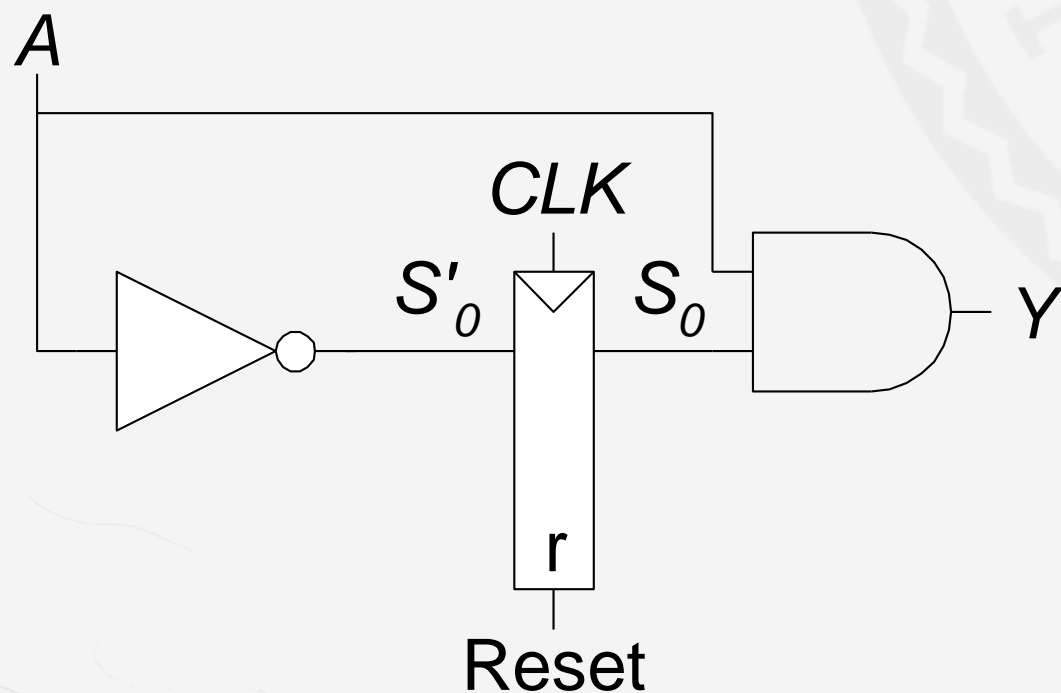




Moore型状态机和Mealy型状态机

Moore and Mealy Machines

Mealy 型状态机原理图

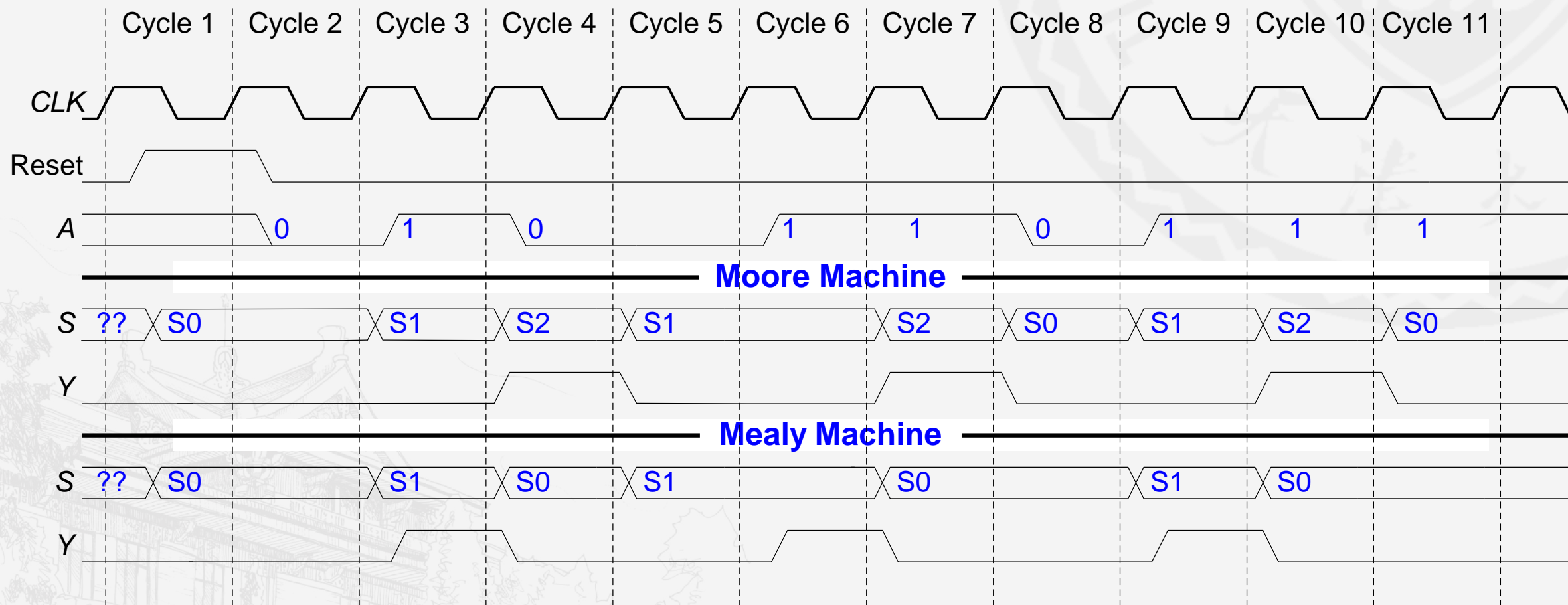




Moore型状态机和Mealy型状态机

Moore and Mealy Machines

Moore型和Mealy型状态机时序图





Mealy型有限状态机设计方法

- ① 根据问题进行抽象，确定输入输出以及对应的逻辑含义
- ② 画出状态转换图
- ③ 列出状态转换表和输出表（可同时列出）
- ④ 对状态和输出进行编码，并列出次态方程和输出方程
- ⑤ 绘制原理图



Moore型状态机与Mealy型状态机的总结

- Moore型状态机：输出只和现态有关
 - 在状态转换图中，输出标记在圆圈内
 - 输入后，需要等待状态的变化才能得到输出
 - 相同情况下需要使用更多的状态

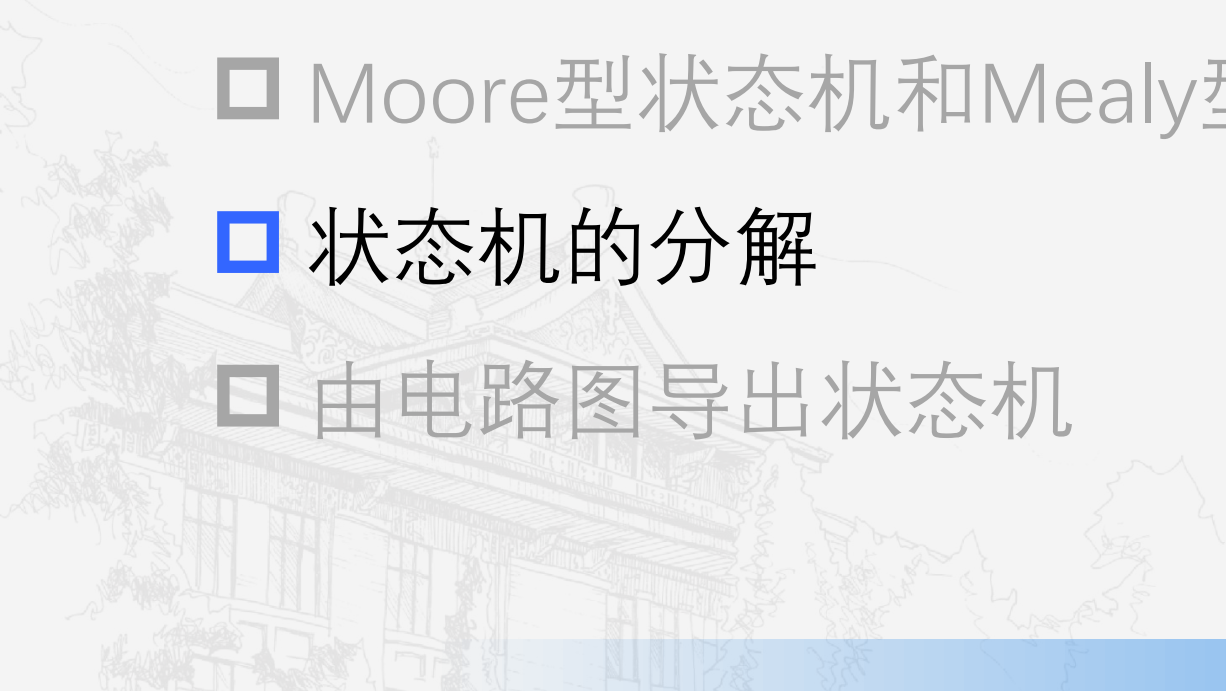
- Mealy型状态机：输出与现态和输入都有关
 - 在状态转换图中，输出标记在圆弧上
 - 输出可以直接响应输入
 - 相同情况下需要使用更少的状态



有限状态机

Finite State Machines

- 基本概念
- 有限状态机设计实例
- 状态编码
- Moore型状态机和Mealy型状态机
- 状态机的分解**
- 由电路图导出状态机



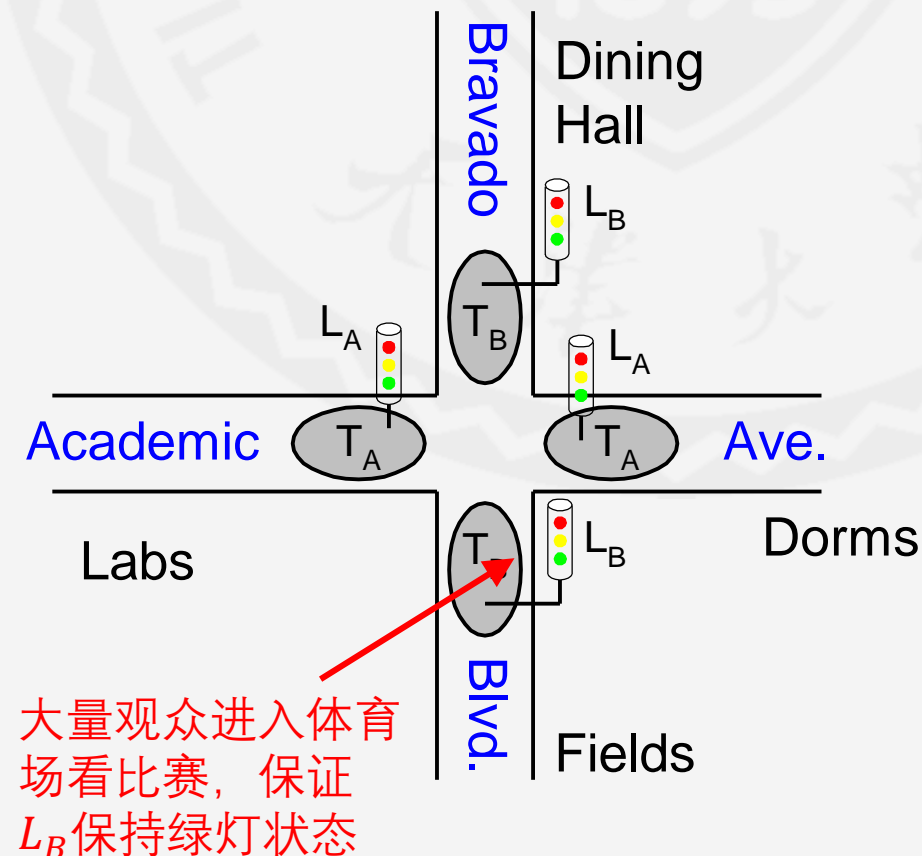


状态机的分解

Factoring State Machines

状态机的分解

- 目标：将复杂的有限状态机分解成多个的简单状态机
 - 其中一些状态机的输出是另一些状态机的输入
 - 层次化、模块化
- 例子：为交通灯控制器增加“游行”模式
 - 增加两个输入端：P, R
 - 当 $P = 1$ 时，进入“游行”模式，保证Bravado Blvd.大道的交通灯一直为绿色
 - 当 $R = 1$ 时，退出“游行”模式



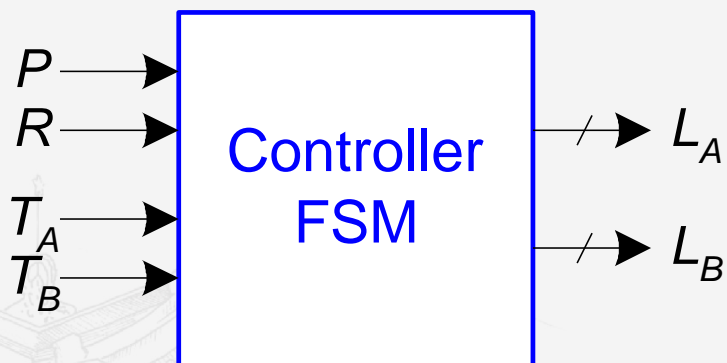


状态机的分解

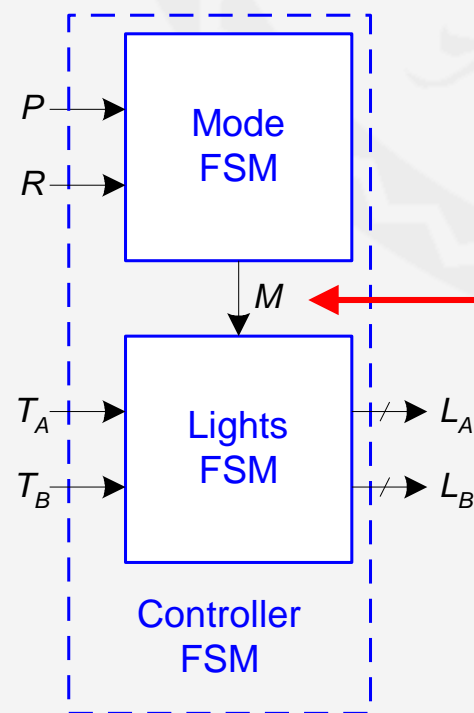
Factoring State Machines

“游行”模式状态机设计

未分解的状态机



状态机分解后



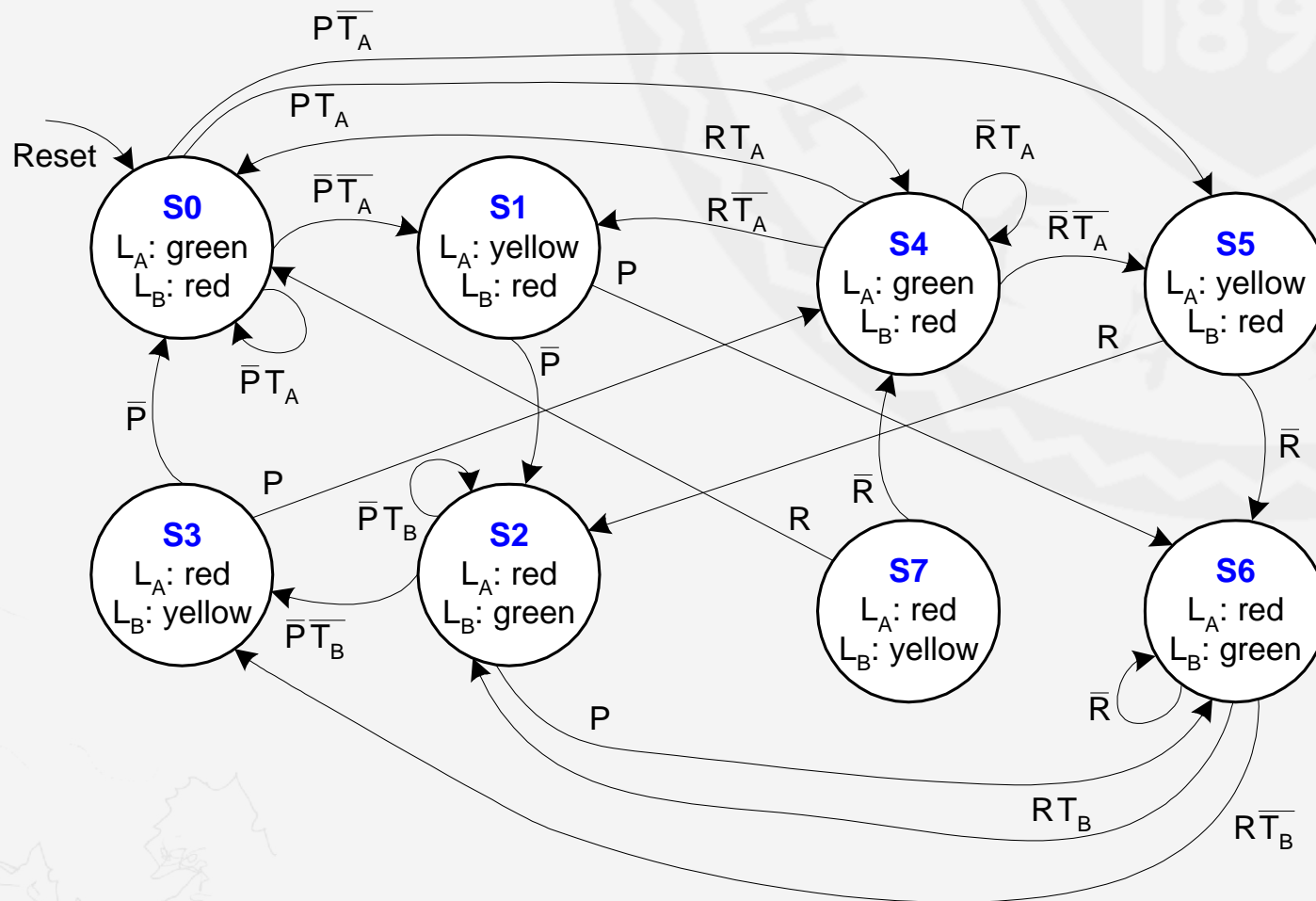
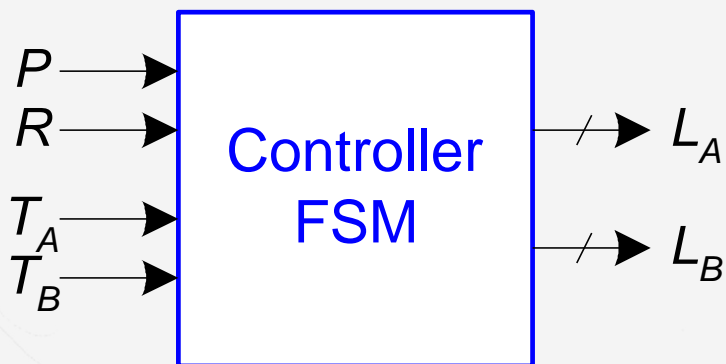
处于游行模式时M为1，
否则为0



状态机的分解

Factoring State Machines

“未分解”的状态转换图



共有8种状态，16种输入的组合

需要考虑 $2^{3+4} = 128$ 种状态转移的情况

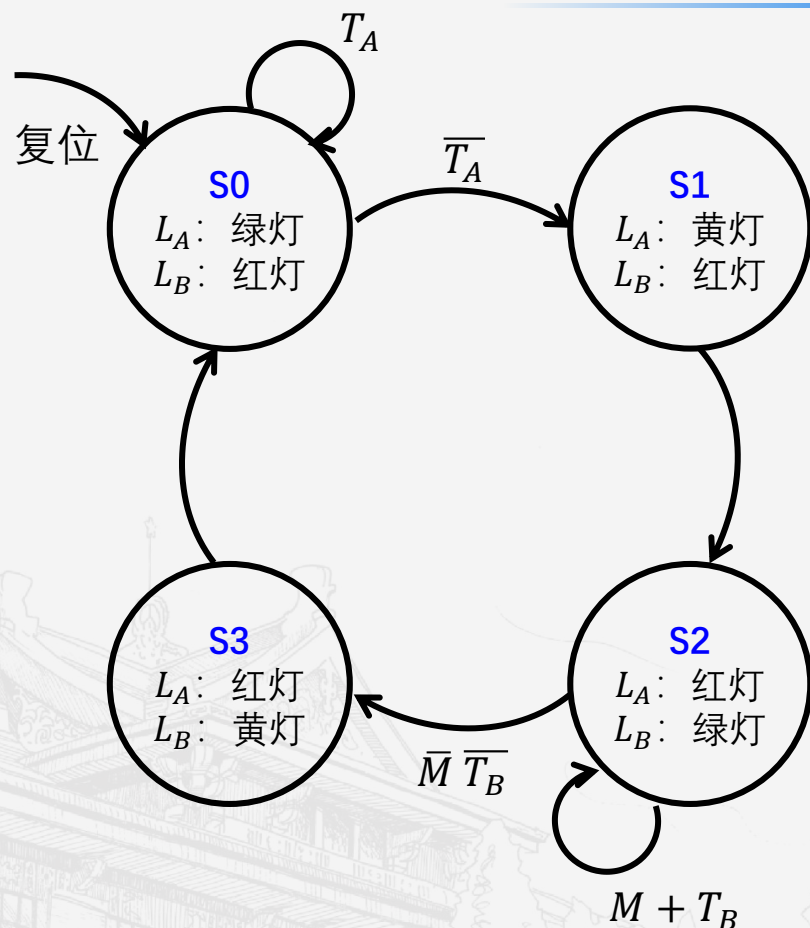
设计十分复杂



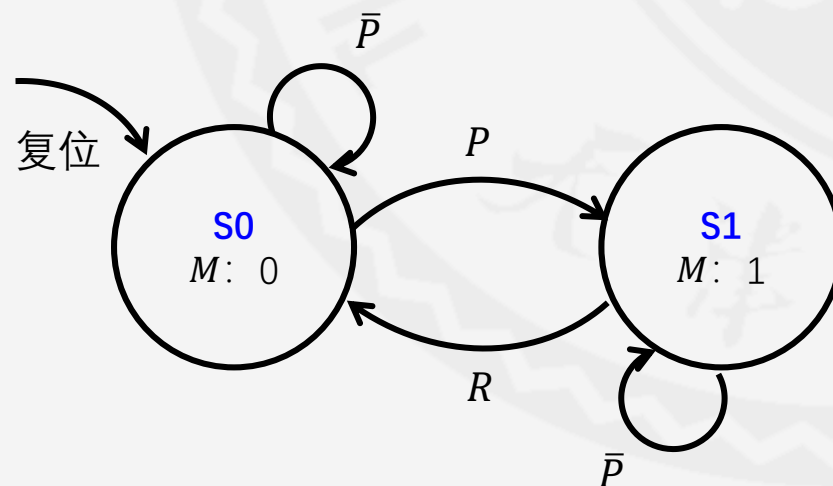
状态机的分解

Factoring State Machines

“分解后”的状态转换图



灯状态机



模式状态机

减少了每个状态机的状态空间，降低了设计的复杂度

逻辑关系更加清晰



有限状态机设计小结

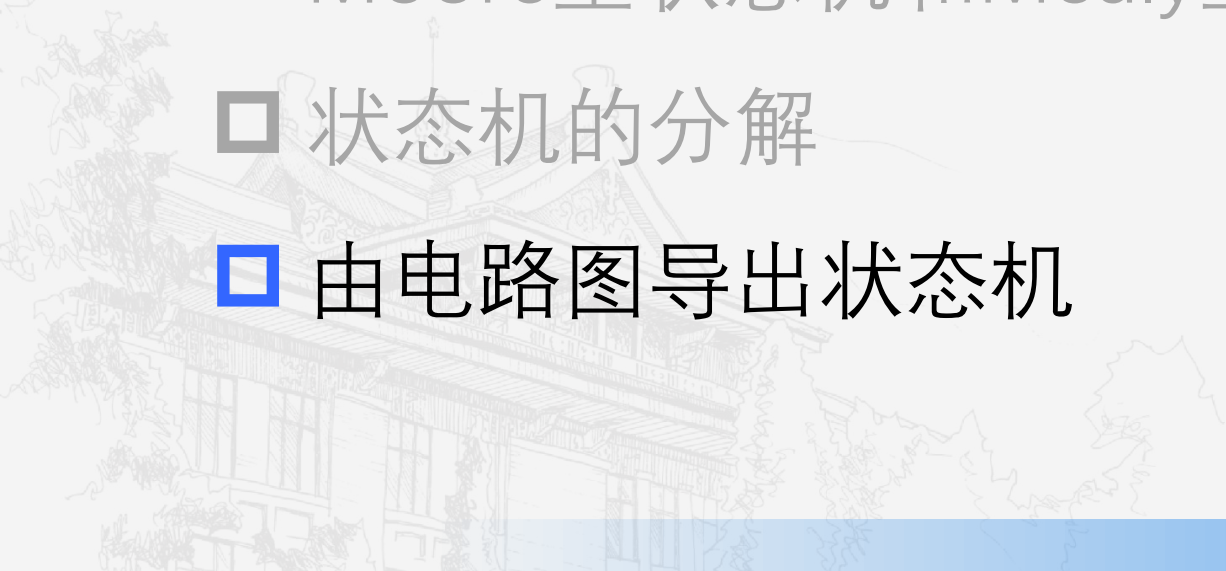
- ① 确定输入和输出
- ② 画状态转换图
- ③ 对于Moore型状态机
 - 写出状态转换表
 - 写出输出表
- ④ 对于Mealy型状态机：
 - 写出组合的状态转换表和输出表
- ⑤ 选择状态编码
- ⑥ 写出次态方程和输出方程
- ⑦ 绘制电路原理图



有限状态机

Finite State Machines

- 基本概念
- 有限状态机设计实例
- 状态编码
- Moore型状态机和Mealy型状态机
- 状态机的分解
- 由电路图导出状态机





由电路图导出状态机

Deriving an FSM from a Schematic

由电路图导出状态机

■ 电路分析：在给定电路原理图的情况下推断电路的逻辑功能，是电路设计的逆过程

■ 当承担一个没有完整文档的项目

■ 开展基于他人系统的逆向工程

■ 主要步骤：

- ① 检查电路，标明输入输出和状态位
- ② 写出次态方程和输出方程
- ③ 列出状态表和输出表
- ④ 删除不可达状态以简化状态表
- ⑤ 给每个有效状态编码指定状态名称
- ⑥ 用状态名称重写状态表和输出表
- ⑦ 画出状态转换图
- ⑧ 使用文字描述有限状态机的功能

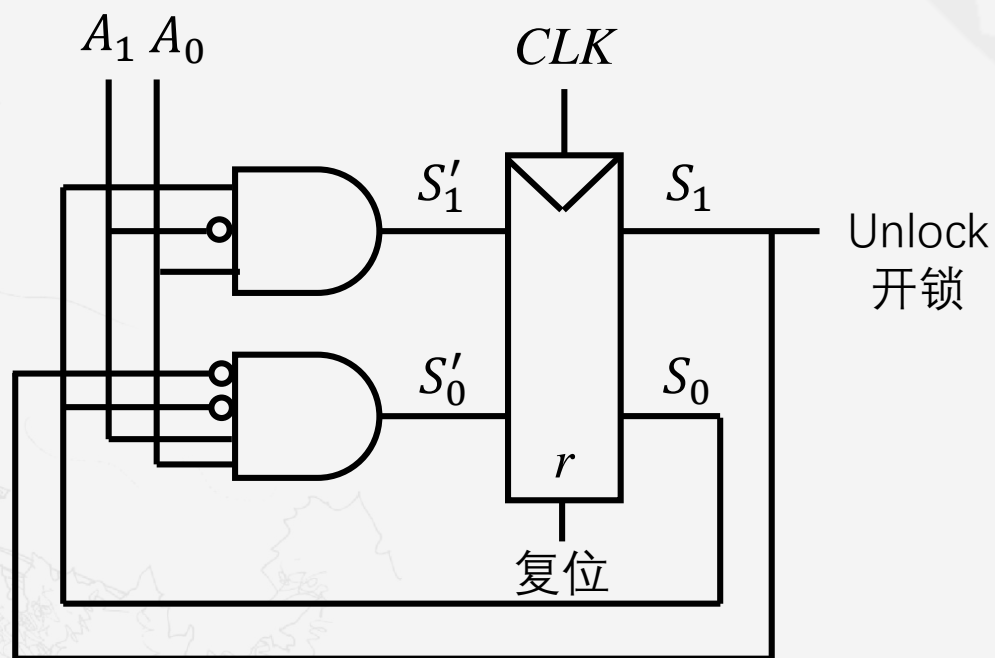


由电路图导出状态机

Deriving an FSM from a Schematic

例：键盘锁电路分析

- 下图是一个键盘锁电路，包含两个输入和一个输出，当输出为1时表示开锁成功。试分析，如何进行输入才能使电路产生开锁信号。





由电路图导出状态机

Deriving an FSM from a Schematic

① 检查电路，标明输入输出和状态位

- 输入 A_0, A_1

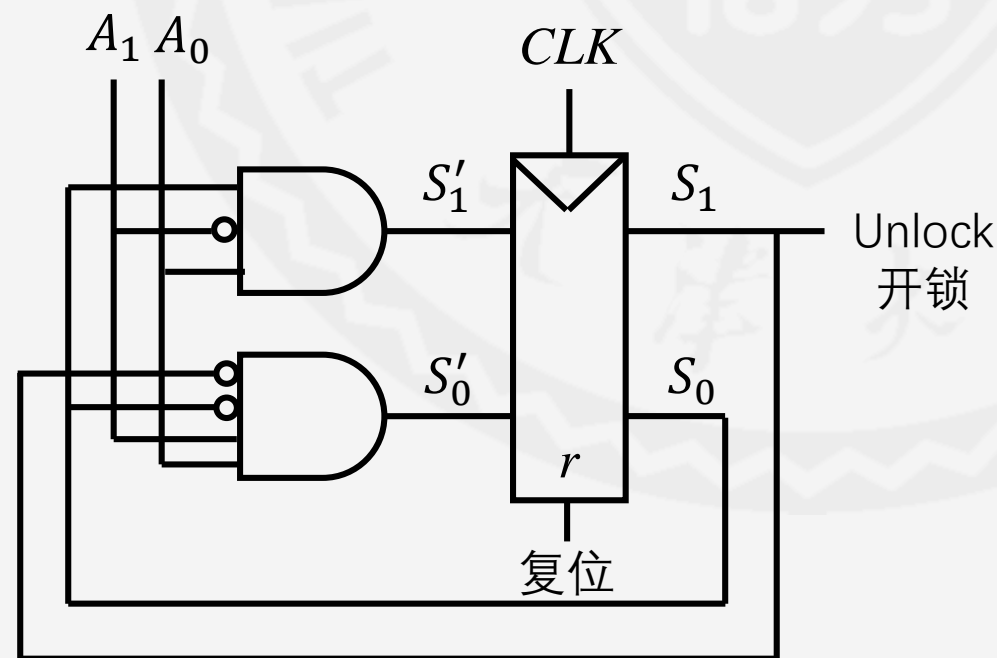
- 输出 $Unlock$

- 状态位 S_0, S_1

- 电路的输出只取决于状态位

$$Unlock = S_1$$

- 电路是一个 Moore 型状态机





由电路图导出状态机

Deriving an FSM from a Schematic

② 写出次态方程和输出方程

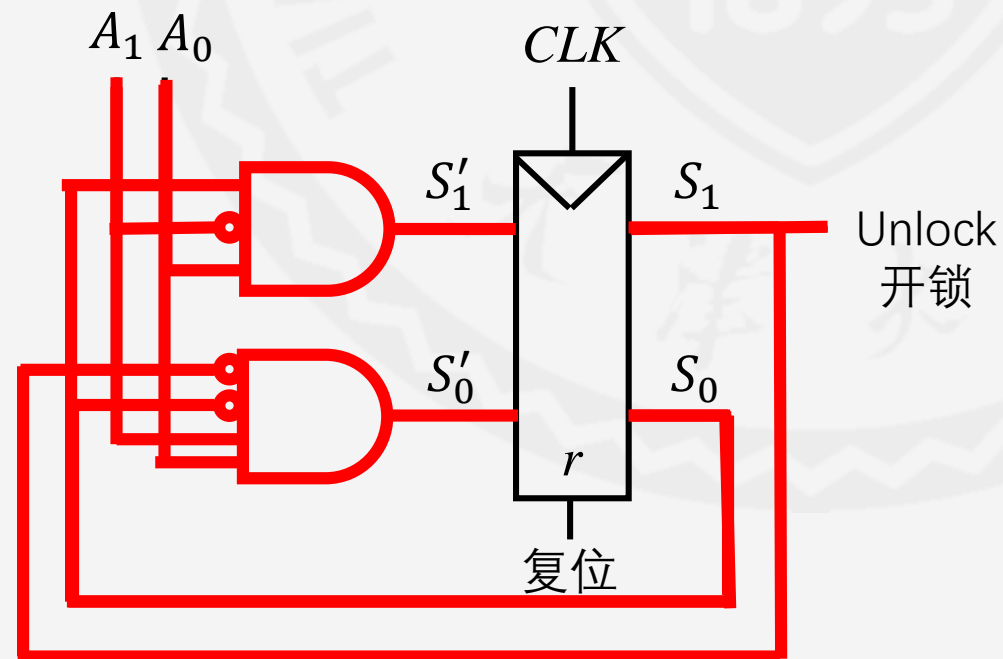
■ 次态方程:

$$S'_1 = S_0 \overline{A_1} A_0$$

$$S'_0 = \overline{S_1} \overline{S_0} A_1 A_0$$

■ 输出方程

$$Unlock = S_1$$





由电路图导出状态机

Deriving an FSM from a Schematic

③ 列出状态表和输出表

■ 次态方程: $S'_1 = S_0 \overline{A_1} A_0$ $S'_0 = \overline{S_1} \overline{S_0} A_1 A_0$

■ 输出方程: $Unlock = S_1$

输出表

现态		输出
S_1	S_0	$Unlock$
0	0	0
0	1	0
1	0	1
1	1	1

状态表

现态		输入		次态	
S_1	S_0	A_1	A_0	S'_1	S'_0
0	0	0	0		
0	0	0	1		
0	0	1	0		
0	0	1	1		
0	1	0	0		
0	1	0	1		
0	1	1	0		
0	1	1	1		
1	0	0	0		
1	0	0	1		
1	0	1	0		
1	0	1	1		
1	1	0	0		
1	1	0	1		
1	1	1	0		
1	1	1	1		



由电路图导出状态机

Deriving an FSM from a Schematic

- ④ 删除不可达的状态，以简化状态表
- 通过观察状态表可以发现：状态 $S_{1:0} = 11$ 从未在表中作为次态出现过
 - 删除不可达状态 $S_{1:0} = 11$ ，状态表和输出表中所有以 11 作为现态的行都可以删去
 - 对于现态 $S_{1:0} = 10$ ，次态总是 $S_{1:0} = 00$ ，与输入无关，可以使用无关项 X 来进一步简化状态表

状态表

现态		输入		次态	
S_1	S_0	A_1	A_0	S'_1	S'_0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	1	0	0



由电路图导出状态机

Deriving an FSM from a Schematic

简化后的状态表

现态		输入		次态	
S_1	S_0	A_1	A_0	S'_1	S'_0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	X	X	0	0

简化后的输出表

现态		输出
S_1	S_0	$Unlock$
0	0	0
0	1	0
1	0	1



由电路图导出状态机

Deriving an FSM from a Schematic

⑤ 给每个有效状态编码指定状态名称

$S_{1:0}$ 编码	状态
00	S_0
01	S_1
10	S_2

⑥ 用状态名称重写状态表和输出表

状态表

现态	输入	次态
S	A	S'
S_0	0	S_0
S_0	1	S_0
S_0	2	S_0
S_0	3	S_1
S_1	0	S_0
S_1	1	S_2
S_1	2	S_0
S_1	3	S_0
S_2	X	S_0

输出表

现态	输出
S	$Unlock$
S_0	0
S_1	0
S_2	1



由电路图导出状态机

Deriving an FSM from a Schematic

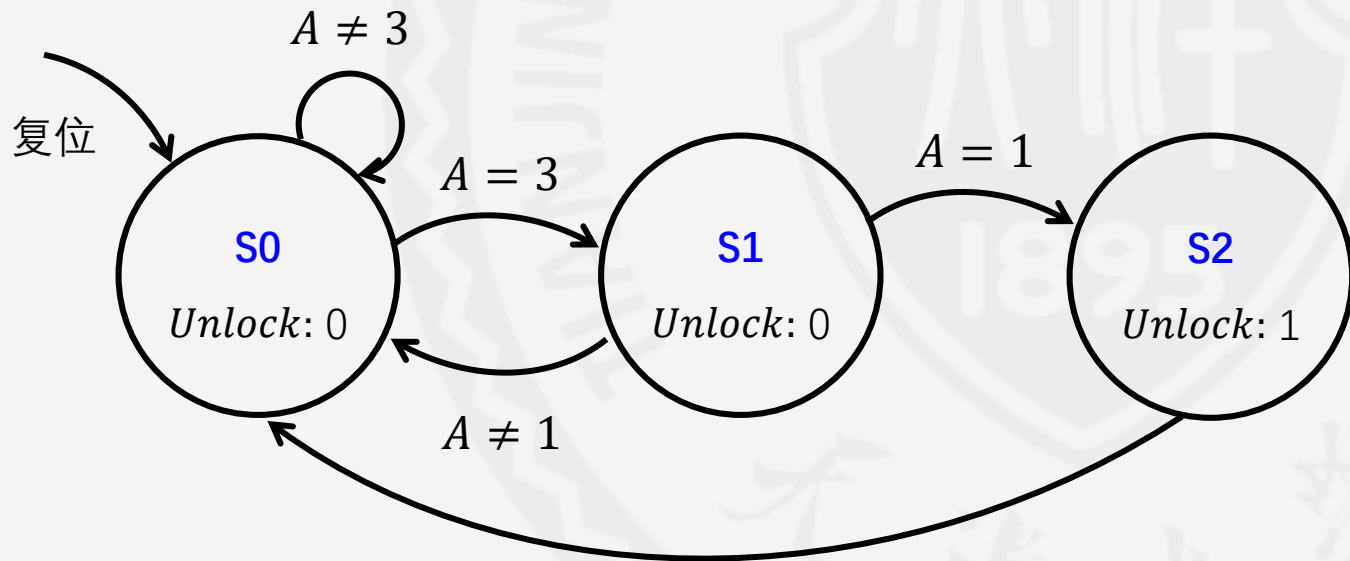
⑦ 画出状态转换图

状态表

现态	输入	次态
S	A	S'
S_0	0	S_0
S_0	1	S_0
S_0	2	S_0
S_0	3	S_1
S_1	0	S_0
S_1	1	S_2
S_1	2	S_0
S_1	3	S_0
S_2	X	S_0

输出表

现态	输出
S	$Unlock$
S_0	0
S_1	0
S_2	1



⑧ 使用文字描述有限状态机的功能

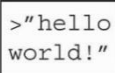


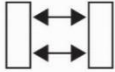
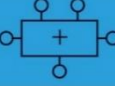

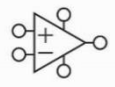
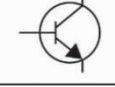

在复位状态下，输入3，然后输入1，门解锁；
然后门再次锁上



本章内容

Topic

- 引言
- 锁存器和触发器
- 同步逻辑设计
- 有限状态机
- 时序逻辑中的时序问题**
- 时序逻辑模块
- 存储器阵列

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	



时序逻辑中的时序问题

Timing of Sequential Logic

□ 动态约束

□ 系统时序





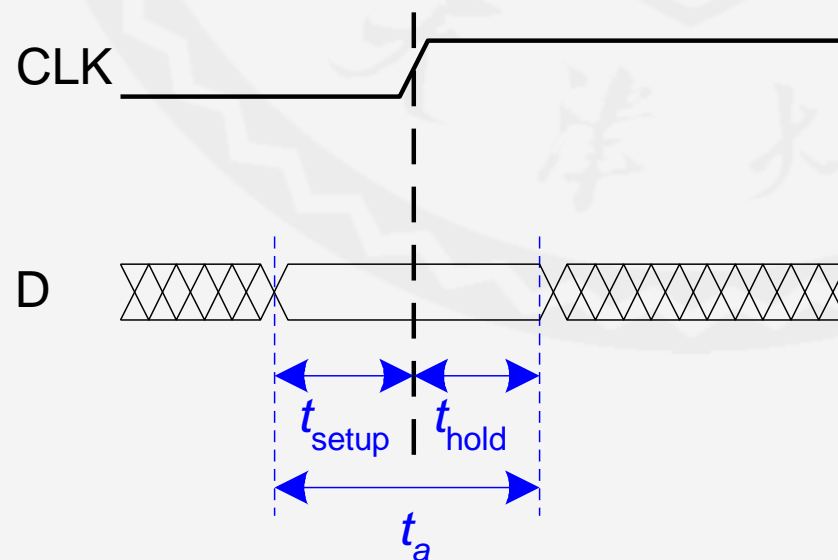
时序

- D 触发器在时钟的有效边沿（上升沿/下降沿）对输入 D 采样，并赋值给 Q
- 在采样的时刻，D 必须处在一个稳定的状态，为 0 或为 1
- 这个过程如同照相一样
 - 只有当被拍摄的物体静止不动时才能获得清晰的图像
- 如果在采样时刻，D 未处于稳定的状态，则会产生亚稳态



输入时序约束

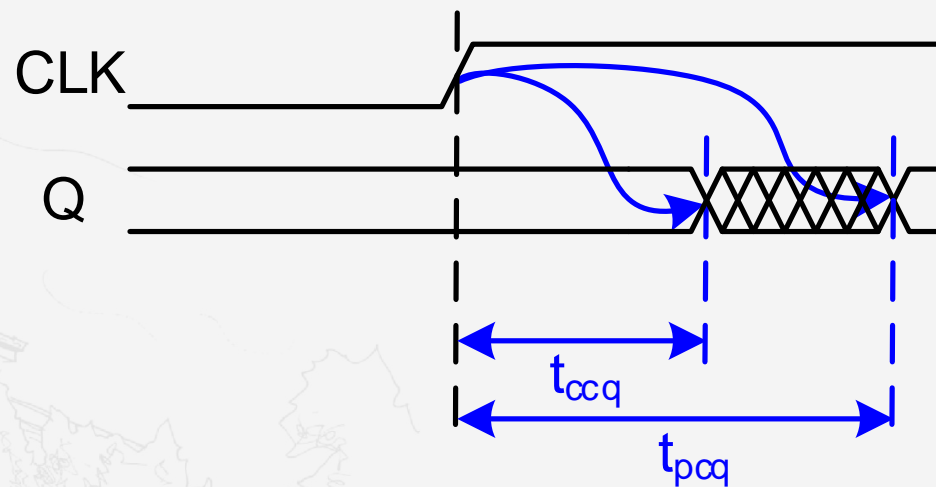
- 建立时间 (Setup time) : t_{setup} = 在时钟有效边沿到来前输入信号所需要的稳定时间
- 保持时间 (Hold time) : t_{hold} = 在时钟有效边沿到来后输入信号所保持稳定时间
- 孔径时间 (Aperture time) : t_a = 在时钟边沿附近输入信号需要保持稳定的总时间
- $t_a = t_{setup} + t_{hold}$





输出时序约束

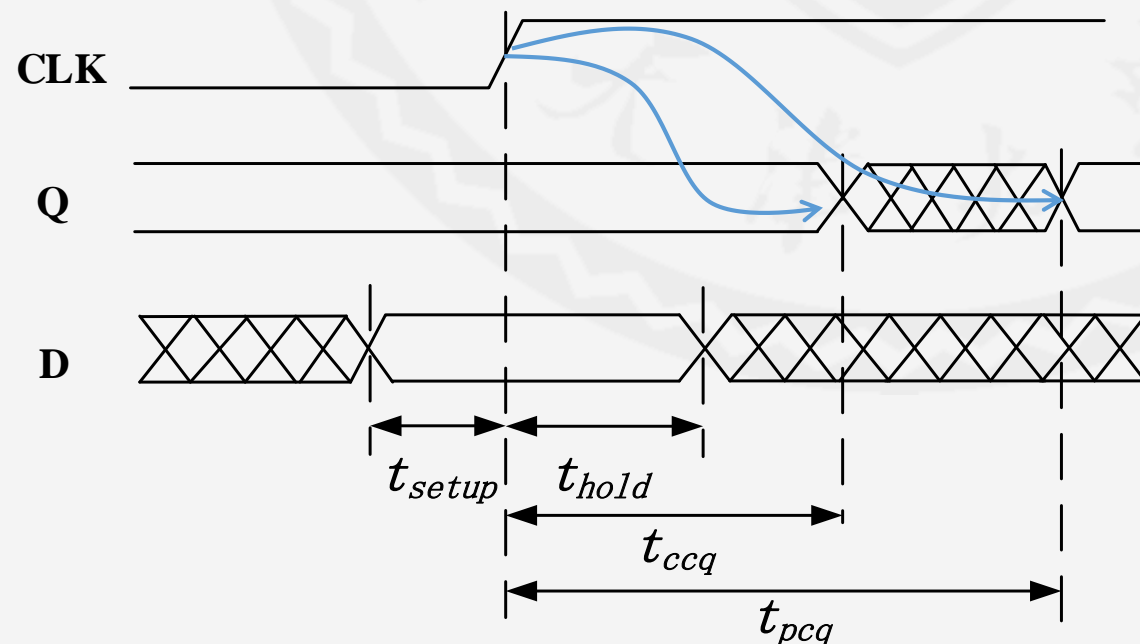
- 传播延迟 (Propagation delay) : t_{pcq} = 时钟有效边沿到达后到 Q 最终稳定所需的最长时间
- 最小延迟 (Contamination delay) : t_{ccq} = 时钟有效边沿到达后到 Q 开始改变所需的最短时间





动态约束

- 同步时序电路中，输入必须在时钟有效边沿附近的孔径时间内保持稳定
- 输入信号必须稳定
 - 在时钟有效边沿到达前，至少稳定 t_{setup}
 - 在时钟有效边沿到达后，至少稳定 t_{hold}





时序逻辑中的时序问题

Timing of Sequential Logic

□ 动态约束

□ 系统时序

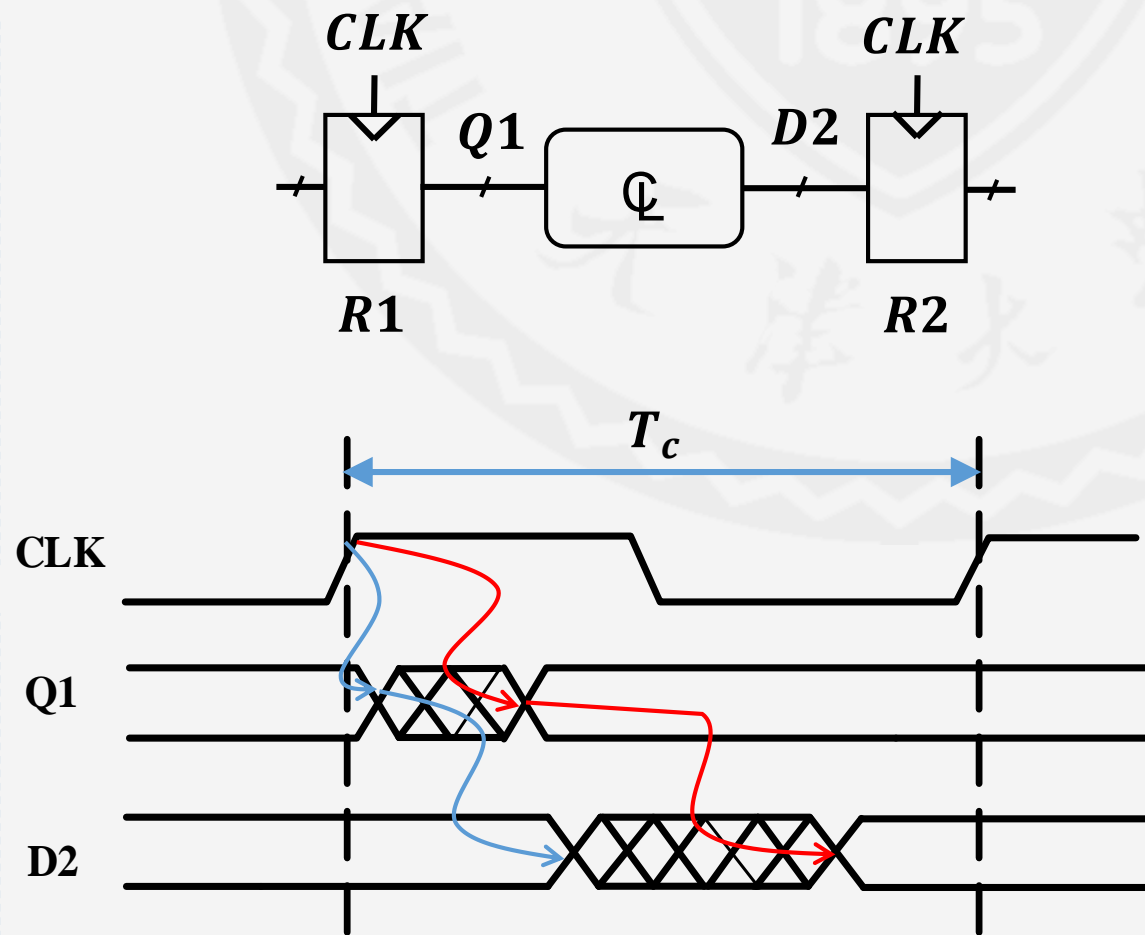




系统时序

- 时钟周期 T_c 是两个时钟上升沿（下降沿）之间的间隔
- $f_c = 1/T_c$ ，表示时钟频率
- 提高时钟频率可以增加数字系统在单位时间内完成的工作量，但频率不能无限制的增加

- 如右图所示，两个寄存器间的延迟具有最小和最大延迟
- 这些延迟由其中的电路模块的延迟所决定



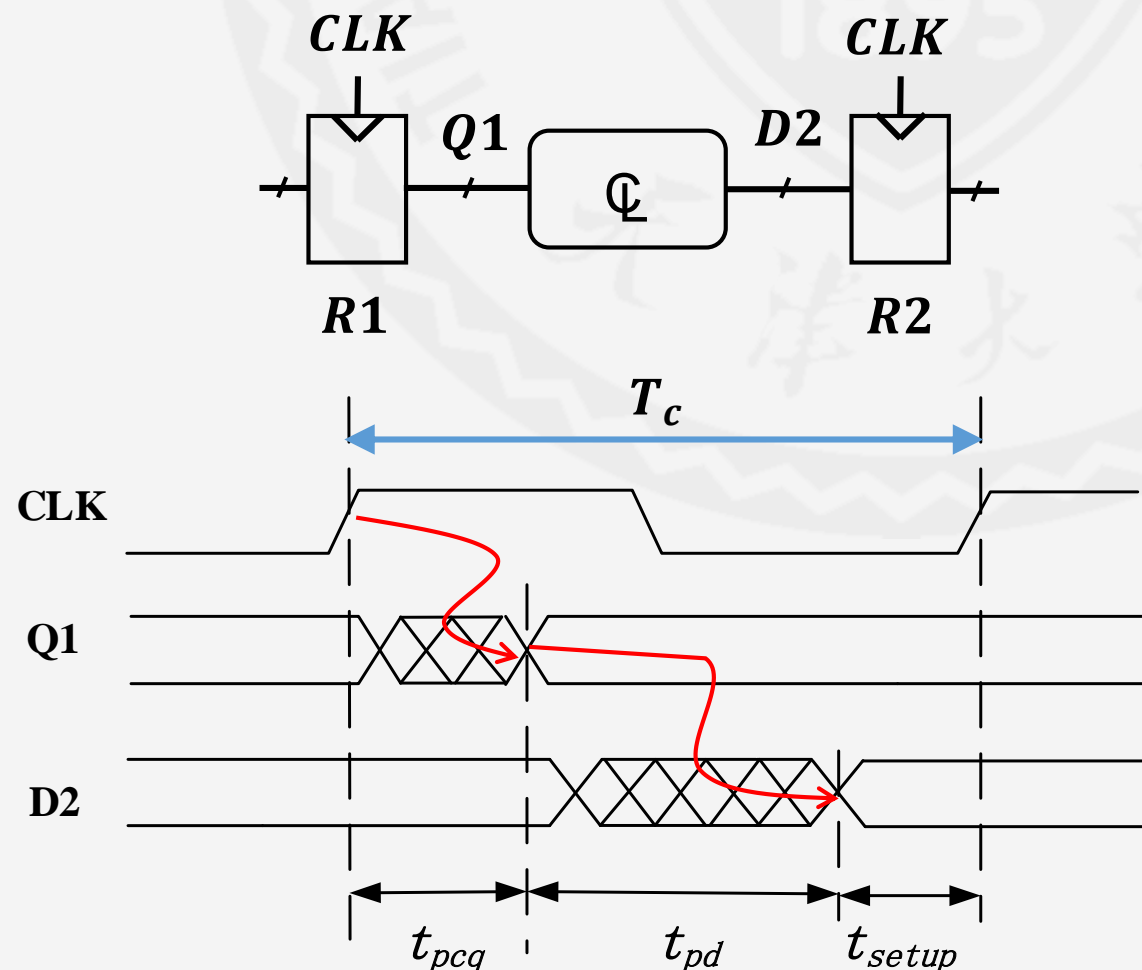


建立时间约束

- 建立时间约束由路径**R1**至**R2**间的最大延迟所决定
 - 寄存器的传播延迟 t_{pcq}
 - 组合逻辑电路的传播延迟 t_{pd}
- 寄存器**R2**的输入信号必须在下一个时钟上升沿的 t_{setup} 时间前稳定

$$T_c \geq t_{pcq} + t_{pd} + t_{setup}$$

$$t_{pd} \leq T_c - (t_{pcq} + t_{setup})$$





建立时间约束 (cont.)

- $t_{pd} \leq T_c - (t_{pcq} + t_{setup})$
- 上式被称为**建立时间约束**或**最大延迟约束**
- 限制了通过组合逻辑的最大延迟
- 在商业设计中
 - T_c 由研发总监和市场部提出，以确保产品的竞争性
 - 制造商确定触发器的传播延迟 t_{pcq} 和建立时间 t_{setup}
 - $t_{pcq} + t_{setup}$ 被称为时序开销，由芯片的生产工艺所决定
 - 通常，只有 t_{pd} 是设计人员能够控制的变量



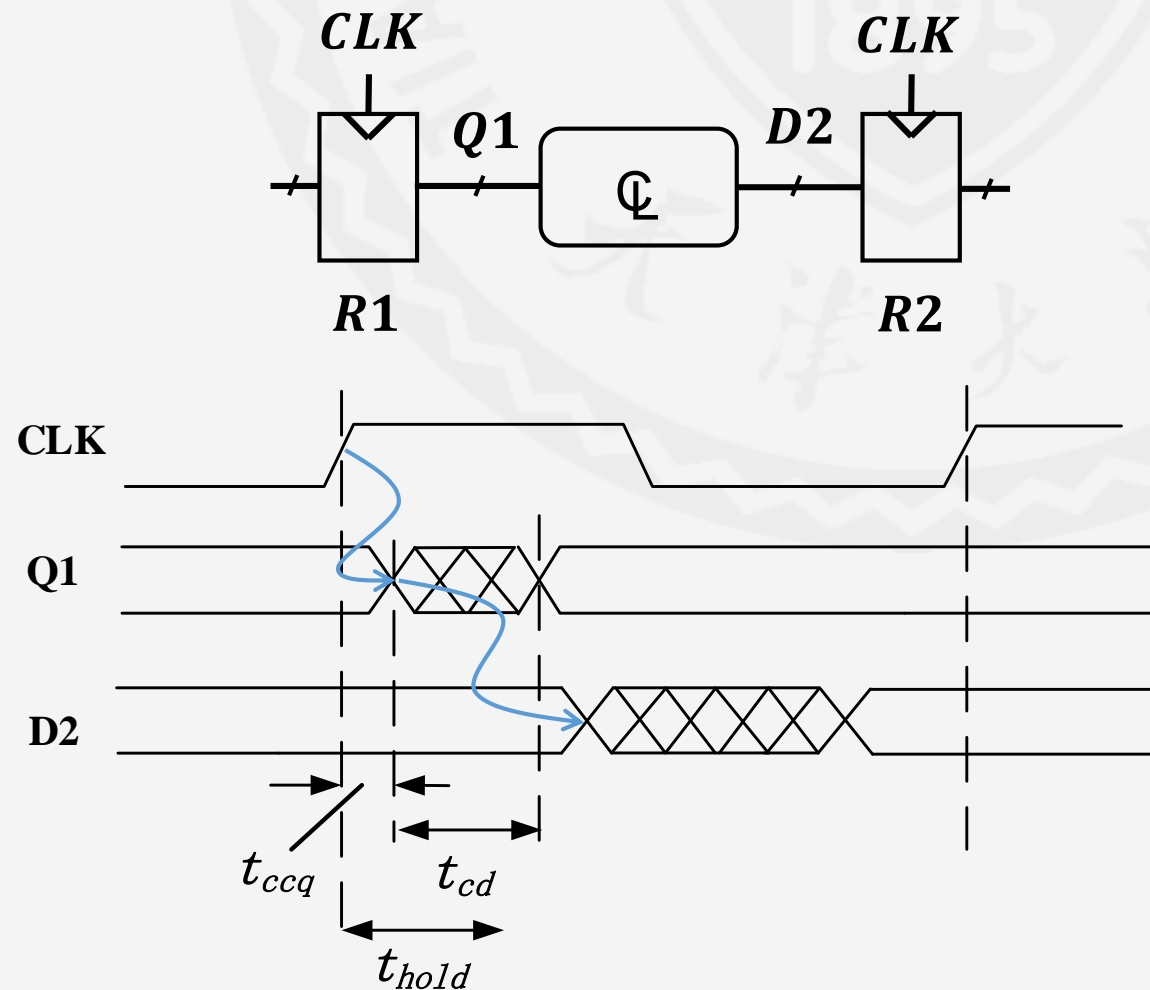


保持时间约束

- 保持时间约束由路径**R1**至**R2**间的最短延迟所决定
 - 寄存器的最小延迟 t_{ccq}
 - 组合逻辑电路的最小延迟 t_{cd}
- 寄存器**R2**的输入信号必须在时钟上升沿后至少稳定 t_{hold} 时间

$$t_{hold} \leq t_{ccq} + t_{cd}$$

$$t_{cd} \geq t_{hold} - t_{ccq}$$

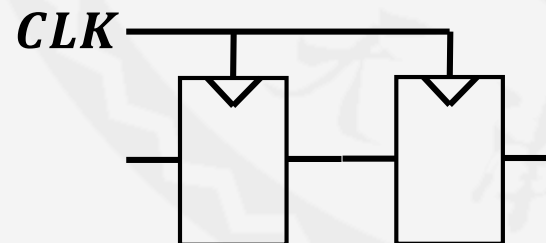




保持时间约束 (cont.)

- $t_{cd} \geq t_{hold} - t_{ccq}$
- 上式被称为**保持时间约束**或**最小延迟约束**
- 限制了通过组合逻辑的最小延迟

- 一个特殊的情况：触发器背靠背相连



- 触发器之间没有组合逻辑，所以 $t_{cd} = 0$
- 如果保证电路不违反保持时间约束，则
$$t_{hold} \leq t_{ccq}$$
- 一个可靠的触发器，保持时间比最小延迟短



保持时间约束 (cont.)

- 在实际应用中，经常将触发器设计成 $t_{hold} = 0$ ，以保证保持时间约束在各种情况下都可以满足
- 教材中如非特别注明，后面的讨论中会忽略保持时间约束
- 保持时间约束又非常重要，如果一旦违反则必须重新设计电路
 - 与建立时间约束不同，不能通过调整时钟周期来改正
- 因此一旦违反保持时间约束会产生非常严重的后果

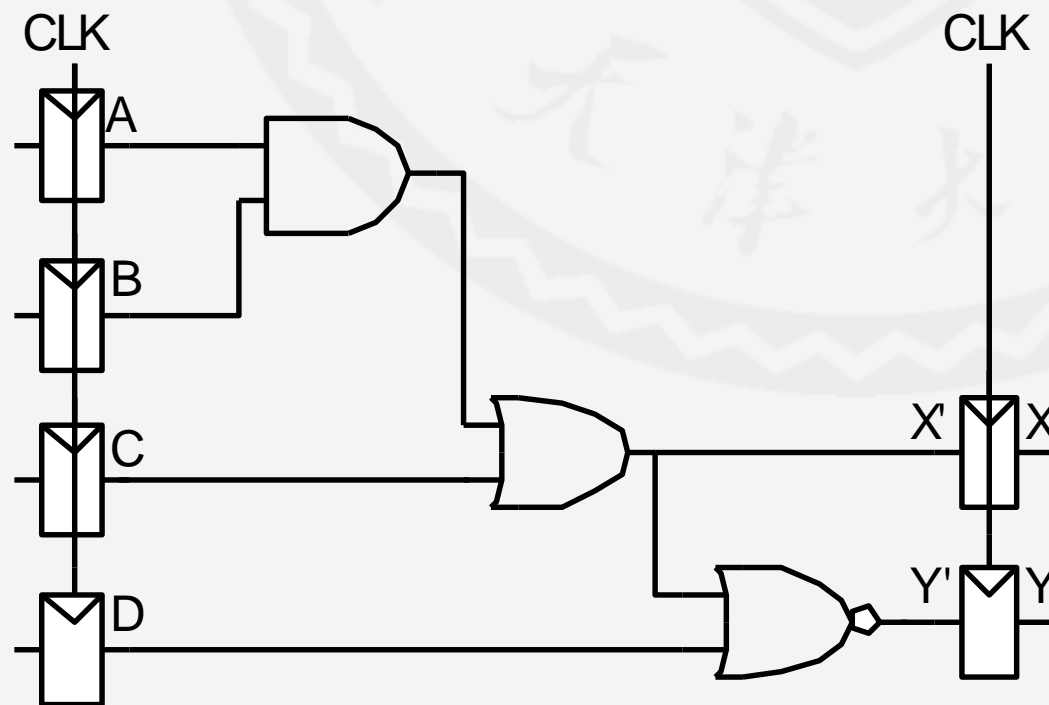


时序分析

在右图中

- 触发器的最小延迟 $t_{ccq} = 30ps$
- 触发器的传播延迟 $t_{pcq} = 80ps$
- 触发器的建立时间 $t_{setup} = 50ps$
- 触发器的保持时间 $t_{hold} = 60ps$
- 逻辑门的最小延迟 $t_{cd_gate} = 25ps$
- 逻辑门的传播延迟 $t_{pd_gate} = 35ps$

求最短时钟周期，并确定是否满足保持时间约束





时序分析 (cont.)

已知:

$t_{ccq} = 30ps, t_{pcq} = 50ps$

$t_{setup} = 60ps, t_{hold} = 70ps$

$t_{cd_gate} = 25ps, t_{pd_gate} = 35ps$

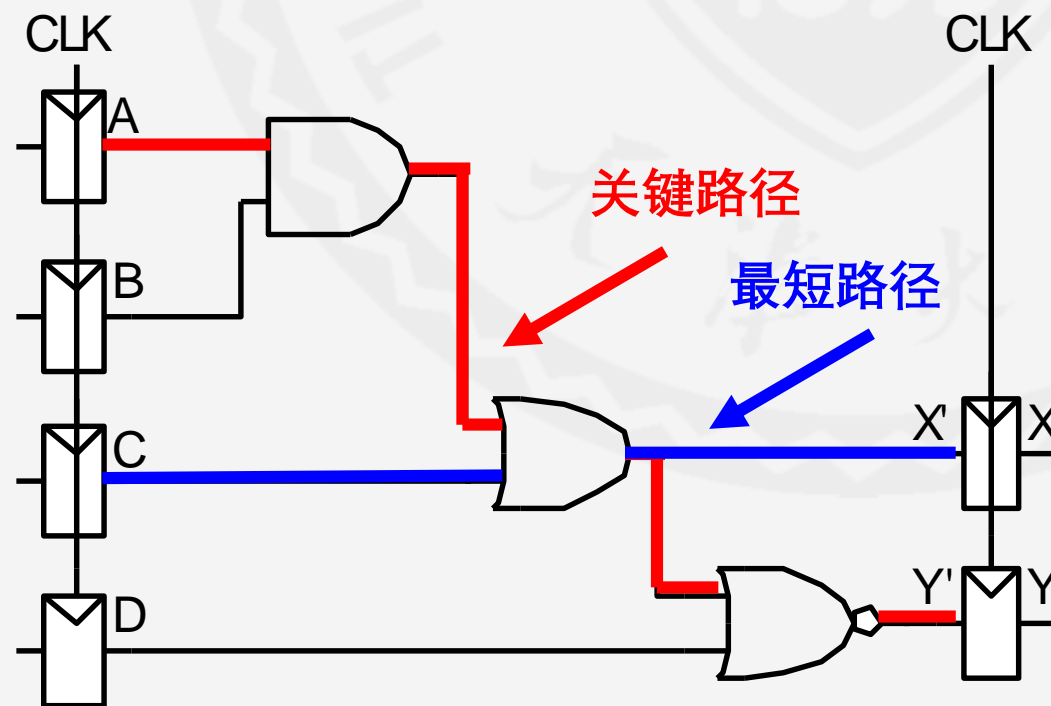
解:

$t_{pd} = 35ps \times 3 = 105ps$

$t_{cd} = 25ps$

$T_c \geq t_{pcq} + t_{pd} + t_{setup} = 50 + 105 + 60 = 215ps$

$t_{cd} = 25ps, t_{hold} - t_{ccq} = 70 - 30 = 40ps$



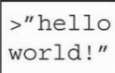


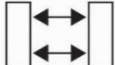
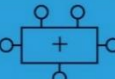

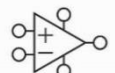


$t_{cd} < t_{hold} - t_{ccq}$ 违反保持时间约束



本章内容

Topic

- 引言
- 锁存器和触发器
- 同步逻辑设计
- 有限状态机
- 时序逻辑中的时序问题
- 时序逻辑模块**
- 存储器阵列

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	



时序逻辑模块

Sequential Building Blocks

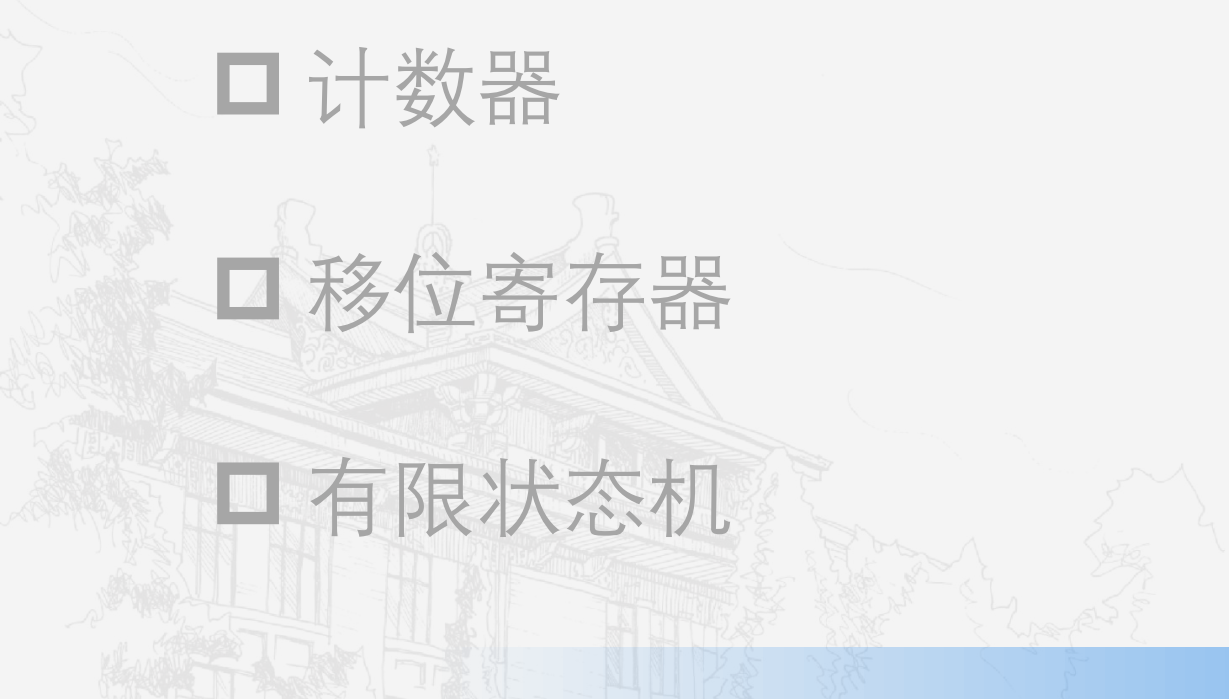
□ 寄存器与锁存器

□ 非阻塞赋值

□ 计数器

□ 移位寄存器

□ 有限状态机





基于SystemVerilog HDL的时序逻辑设计

- SystemVerilog 使用一些特殊的编码风格 (idioms) 描述锁存器、触发器和状态机
- 其他的编码风格虽然可以正确的进行仿真，但是综合后会产生错误的电路



always 过程块的时序逻辑建模

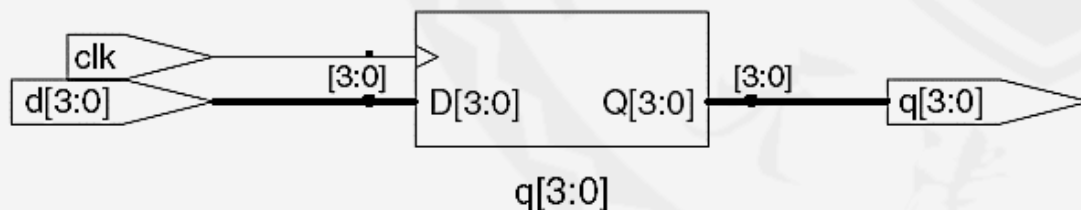
- always过程块分为三种类型：
 - always_comb（描述组合逻辑）， always_latch， always_ff（后两者用于描述时序逻辑）
- always 过程块结构如下：

```
always @(sensitivity list)
    statement;
```
- sensitivity list 是敏感事件列表，当列表中的事件产生时，过程块中的语句开始工作



寄存器建模

```
module flop ( input  logic      clk,  
              input  logic [3:0] d,  
              output logic [3:0] q);  
  
    always_ff @(posedge clk)  
        q <= d;  
  
endmodule
```



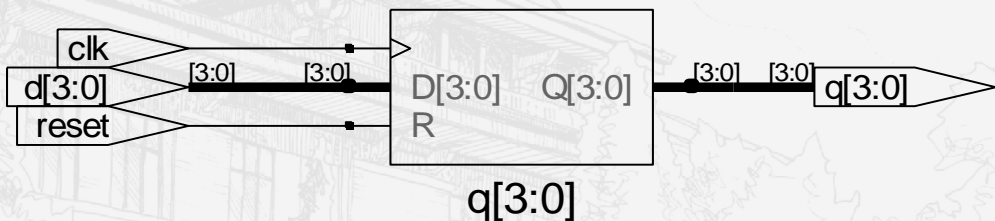
- 使用正边沿D触发器实现
- always_ff 用来表示触发器
- posedge clk 表示 clk 信号上升沿
- <= 是非阻塞赋值，暂时可以把它看做是普通的赋值语句



带复位端的寄存器建模

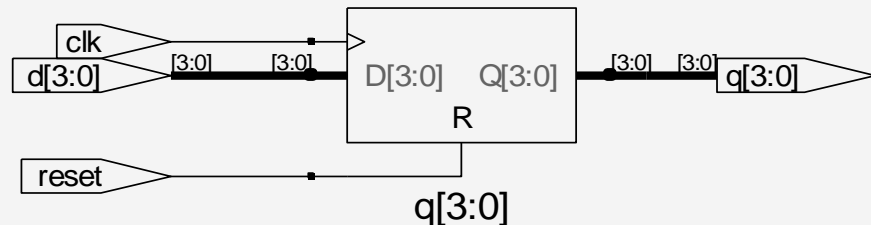
同步复位功能

```
module flopr ( input  logic    clk,  
               input  logic    reset,  
               input  logic [3:0] d,  
               output logic [3:0] q);  
  
    always_ff @(posedge clk)  
        if (reset) q <= 4'b0;  
        else      q <= d;  
  
endmodule
```



异步复位功能

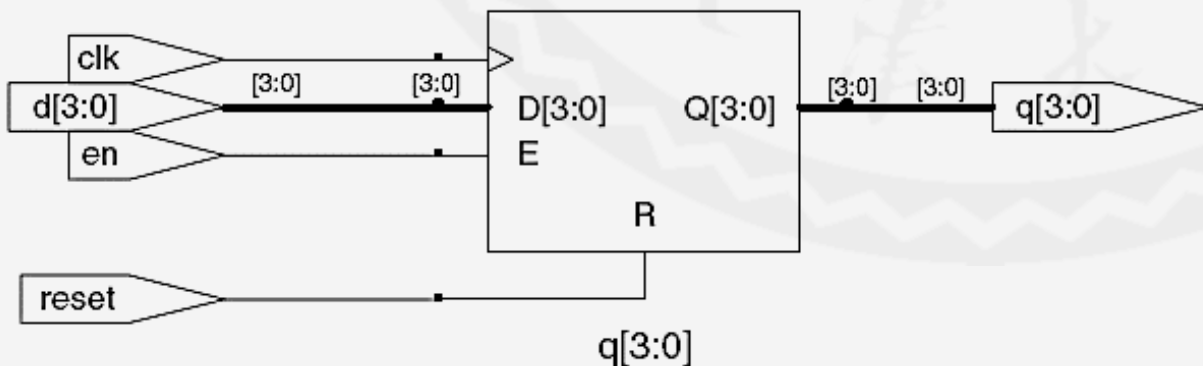
```
module flopr ( input  logic    clk,  
               input  logic    reset,  
               input  logic [3:0] d,  
               output logic [3:0] q);  
  
    always_ff @(posedge clk, posedge reset)  
        if (reset) q <= 4'b0;  
        else      q <= d;  
  
endmodule
```





带使能端的寄存器建模

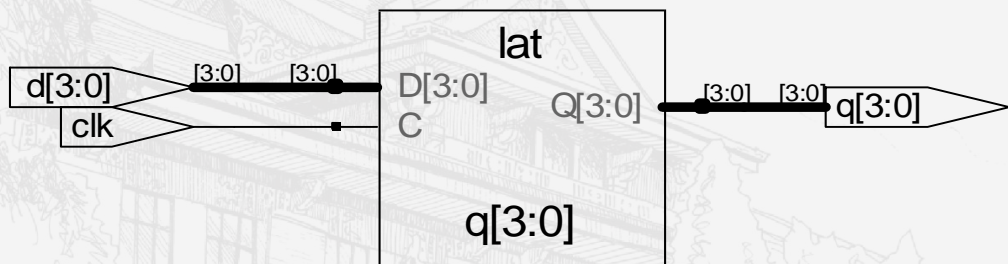
```
module flopren ( input  logic      clk,  
                 input  logic      reset,  
                 input  logic      en,  
                 input  logic [3:0] d,  
                 output logic [3:0] q);  
  
    always_ff @(posedge clk, posedge reset)  
        if (reset)    q <= 4'b0;  
        else if (en)  q <= d;  
  
endmodule
```





锁存器

```
module latch ( input  logic      clk,  
               input  logic [3:0] d,  
               output logic [3:0] q);  
  
    always_latch  
        if (clk)    q <= d;  
  
endmodule
```



- 不是所有的综合工具都能很好地支持锁存器
- 除非你明确地知道工具支持锁存器，或者你有理由使用锁存器
- 最好不要使用锁存器而是使用边沿触发器
- 此外还要防止HDL代码意外生成锁存器



时序逻辑模块

Sequential Building Blocks

□ 寄存器与锁存器

□ 非阻塞赋值

□ 计数器

□ 移位寄存器

□ 有限状态机





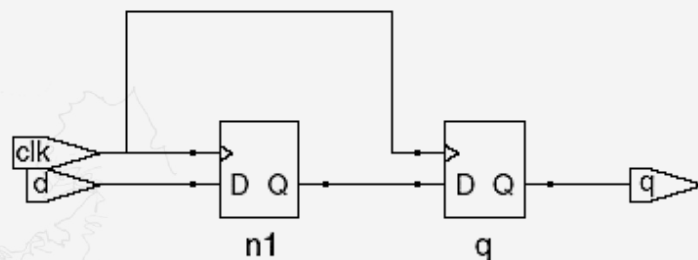
非阻塞赋值

Nonblocking Assignments

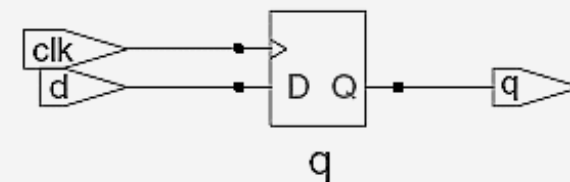
阻塞赋值 vs. 非阻塞赋值

- `<=` 是非阻塞赋值
 - 与其它的语句同时工作
- `=` 是阻塞赋值
 - 按照语句的在代码中的顺序依次工作

```
module syncgood( input logic clk,  
                 input logic d,  
                 output logic q);  
  
    logic n1;  
    always_ff @(posedge clk) begin  
        n1 <= d; // 非阻塞赋值  
        q <= n1; // 非阻塞赋值  
    end  
endmodule
```



```
module syncbad(input logic clk,  
               input logic d,  
               output logic q);  
  
    logic n1;  
    always_ff @(posedge clk) begin  
        n1 = d; // 阻塞赋值  
        q = n1; // 阻塞赋值  
    end  
endmodule
```





赋值语句使用规则

- 同步时序逻辑电路中使用 **always_ff** **@(posedge clk)** 和非阻塞赋值 **<=**

```
always_ff @(posedge clk)
    q <= d;
```

- 简单的组合逻辑电路中使用持续赋值语句 **assign**

```
assign y = a & b;
```

- 使用 **always_comb** 和阻塞赋值语句 **=** 描述复杂组合逻辑

```
always_comb begin
    p = a ^ b;
    g = a & b;
    s = p ^ cin
    cout = g | (p & cin)
end
```

- 不要多于1个always语句块或者持续赋值语句中对同一个信号赋值



时序逻辑模块

Sequential Building Blocks

□ 寄存器与锁存器

□ 非阻塞赋值

□ 计数器

□ 移位寄存器

□ 有限状态机





计数器的结构

■ N位二进制计数器

■ 输入：时钟、复位信号

■ 输出：N位计数结果

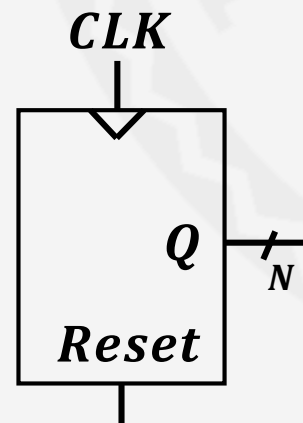
■ 功能：时钟上升沿到达时将结果加1并输出

■ 能够实现循环计数

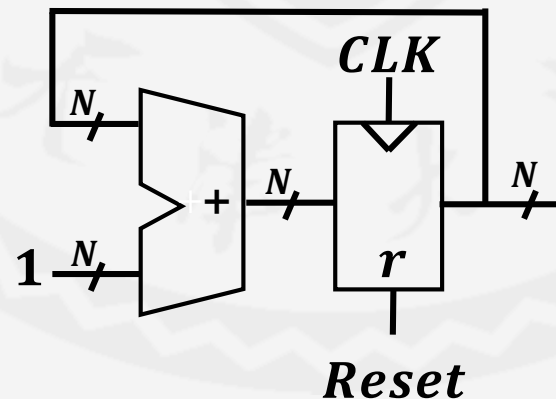
■ 例如：000, 001, 010, 011, 100, 101, 110, 111, 000, 001...

■ 典型应用

■ 数字时钟、程序计数器 (PC)



计数器的电路符号

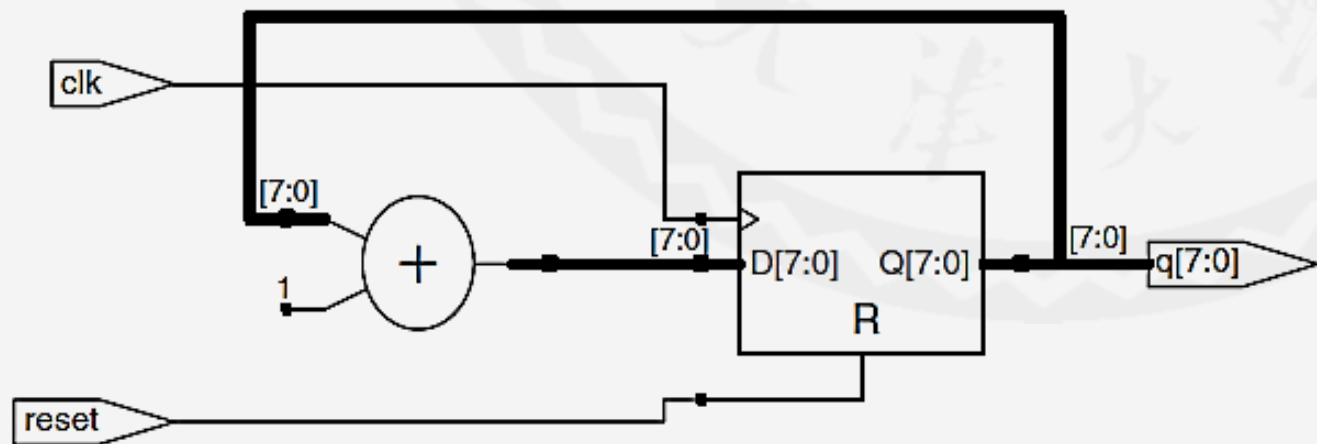


计数器原理图



计数器建模

```
module counter #(parameter N=8)
    ( input  logic clk,
      input  logic reset,
      output logic [N-1: 0]q );
    always_ff @(posedge clk, posedge reset)
    begin
        if (reset) q <= 0;
        else      q  <= q + 1;
    end
endmodule
```





时序逻辑模块

Sequential Building Blocks

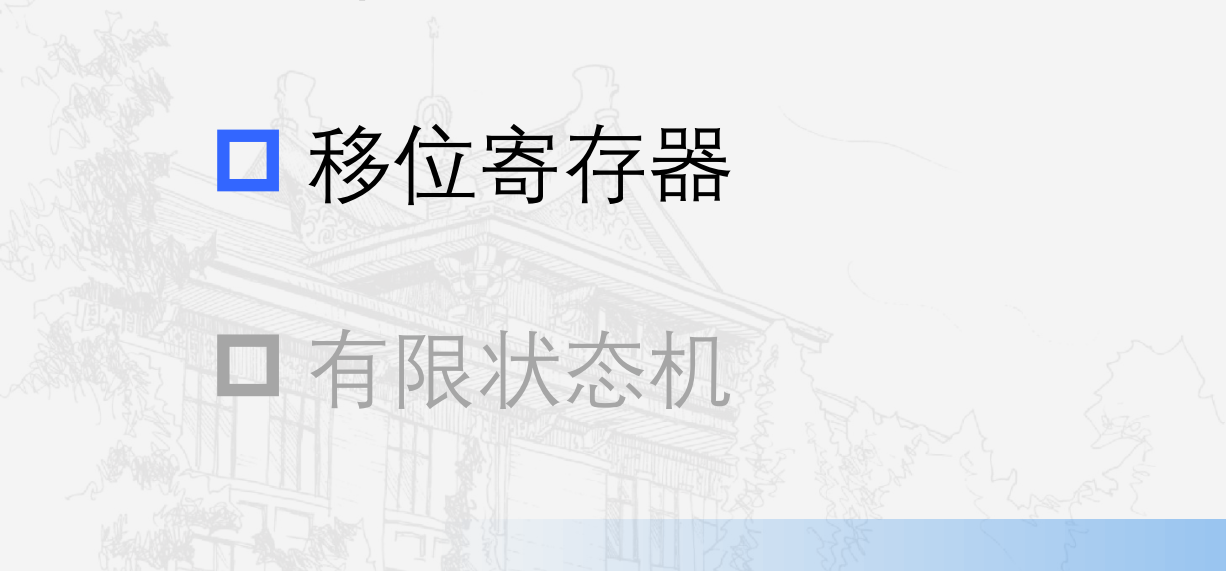
□ 寄存器与锁存器

□ 非阻塞赋值

□ 计数器

■ 移位寄存器

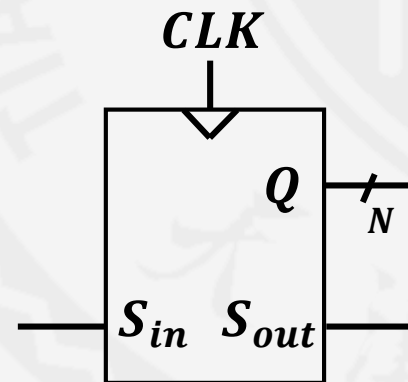
□ 有限状态机



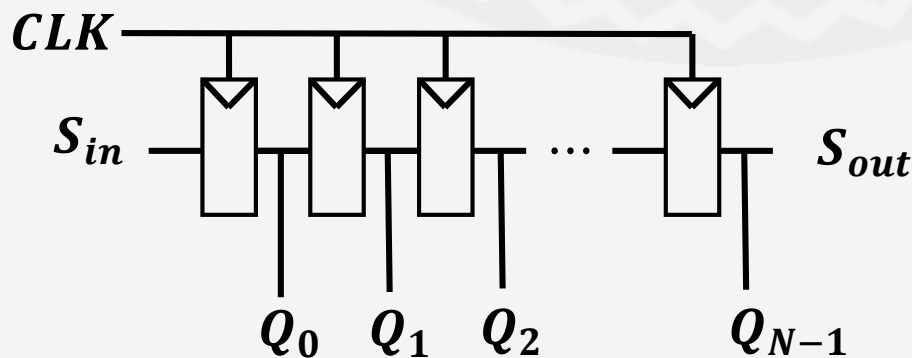


移位寄存器

- 移位寄存器
 - 输入：时钟、串行输入 S_{in}
 - 输出：串行输出 S_{out} 、N位并行输出 $Q_{N-1:0}$
- 功能：在时钟的每一个上升沿，从 S_{in} 移入一个新的位，寄存器中所有内容都向前移动一位，最前面的位移入 S_{out}
- 可以看做是一个串行到并行的转换器，每个周期从 S_{in} 输入一位，N个周期后可以通过 $Q_{N-1:0}$ 直接访问N位输入
- 实现：N个D触发器串联



电路符号



原理图



带并行加载的移位寄存器

带并行加载的移位寄存器

输入：在移位寄存器基础上增加了加载 ($Load$) 信号和并行输入信号 $D_{N-1:0}$

功能：

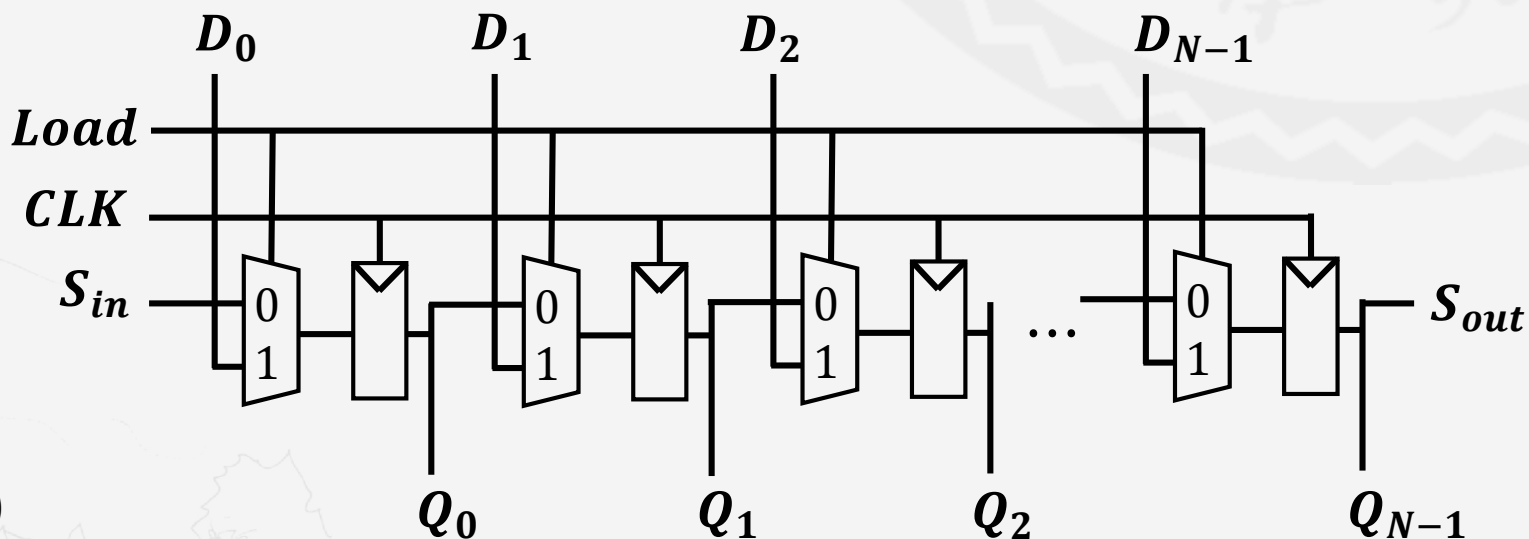
$Load = 1$ ，并行加载N位

$Load = 0$ ，移位寄存器

可以实现

串行转并行 (S_{in} 到 $Q_{N-1:0}$)

并行转串行 ($D_{N-1:0}$ 到 S_{out})

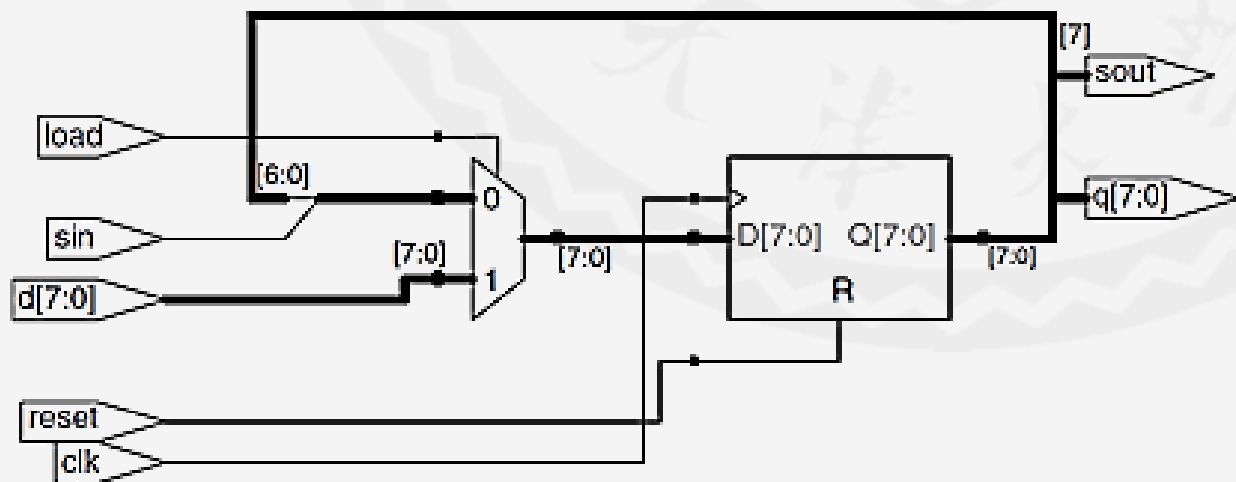




带并行加载的移位寄存器建模

```
module shiftreg #(parameter N=8)
    ( input logic      clk,
      input logic      reset, load,
      input logic      sin,
      input logic [N-1: 0] d
      output logic [N-1: 0] q
      output logic      sout );

    always_ff @(posedge clk, posedge reset)
        if (reset)      q <= 0;
        else if (load)   q <= d;
        else             q <= { q[N-2 : 0], sin };
    assign sout = q[N-1];
endmodule
```





时序逻辑模块

Sequential Building Blocks

□ 寄存器与锁存器

□ 非阻塞赋值

□ 计数器

□ 移位寄存器

□ 有限状态机





有限状态机结构

有限状态机主要包含三个模块：

状态寄存器

存储当前时刻的状态

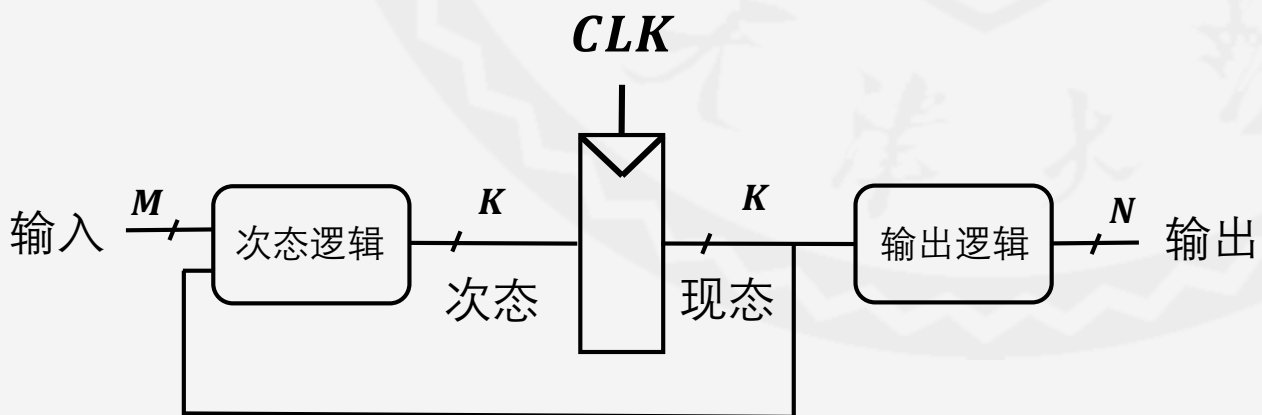
状态在下一个有效时钟沿发生改变

组合逻辑

计算下一个状态（次态逻辑）

计算电路的输出（输出逻辑）

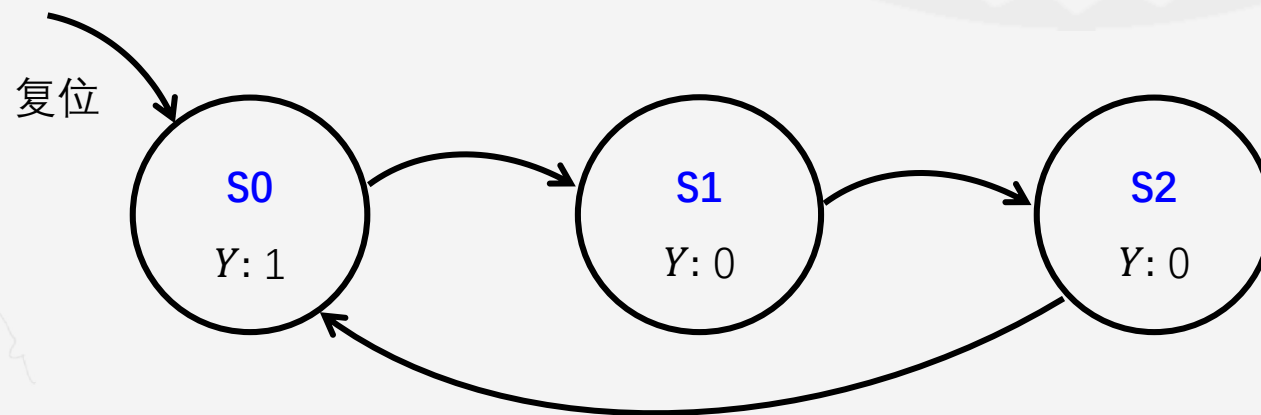
在状态机建模时也包含着三个对应的语句块





有限状态机实例

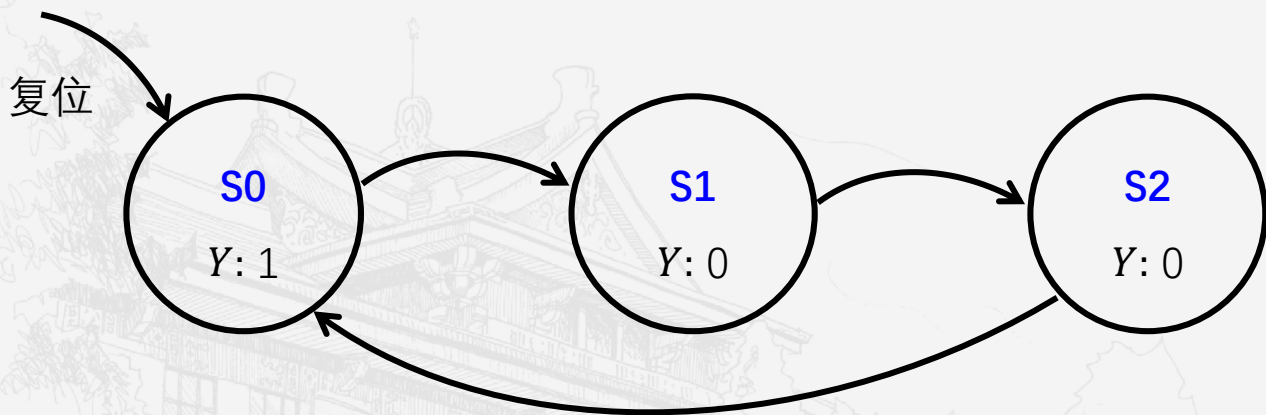
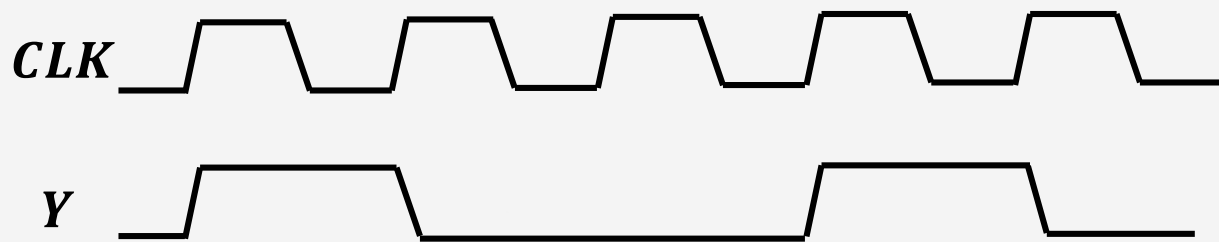
- 例：3分频计数器
 - 一个时钟输入
 - 一个输出
 - 每3个时钟周期后输出产生一个周期的高电平
 - 输出是时钟的3分频





有限状态机

FSM HDL



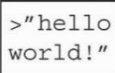


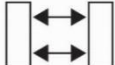
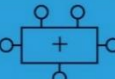

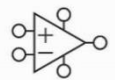


```
module divideby3FSM ( input logic clk, reset,  
                      output logic q);  
  
    typedef enum logic [1:0] {S0, S1, S2} statetype;  
    statetype [1:0] state, nextstate;  
  
    // 寄存器  
    always_ff @ (posedge clk, posedge reset) begin  
        if (reset) state <= S0;  
        else state <= nextstate;  
    end  
  
    always_comb begin // 次态逻辑  
        case (state)  
            S0: nextstate = S1;  
            S1: nextstate = S2;  
            S2: nextstate = S0;  
            default: nextstate = S0;  
        endcase  
    end  
  
    assign q = (state == S0); // 输出逻辑  
endmodule
```




本章内容

Topic

- 引言
- 锁存器和触发器
- 同步逻辑设计
- 有限状态机
- 时序逻辑中的时序问题
- 时序逻辑模块
- 存储器阵列**

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	



存储器阵列

Memory Arrays

□ 概述

□ 存储器的组织

□ DRAM与SRAM

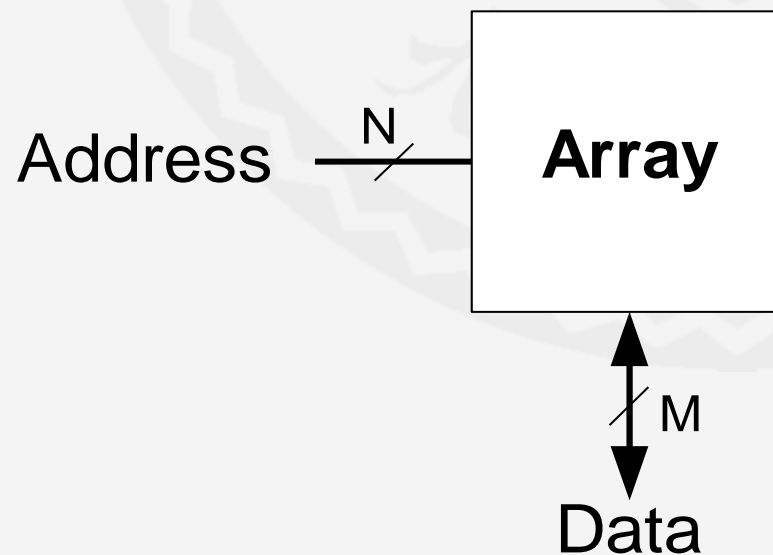
□ 寄存器文件





存储器阵列

- 一种有效的存储大量数据的模块
- 每个N位地址都可以读出或写入M位的数据
 - 数据：存储的内容
 - 地址：数据的索引

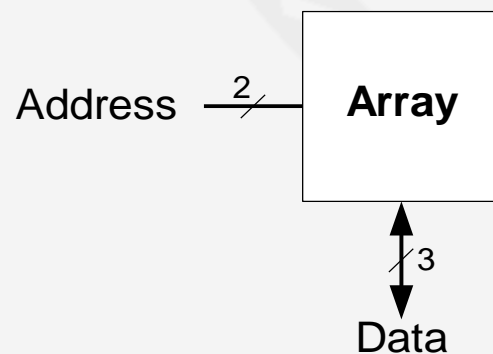


通用存储器阵列的电路符号



存储器阵列 (cont.)

- 存储器由一个二维存储单元阵列构成
- 每个位单元存储1位数据
- 一个N位地址M位数据的阵列：
 - 有 2^N 行和 M 列
 - 深度 (Depth) : 阵列的行数
 - 宽度 (Wdith) : 阵列的列数
 - 阵列的总大小 (Array Size) : 深度 \times 宽度 = $2^N \times M$
 - 字 (Word) : 每行数据都称为一个字

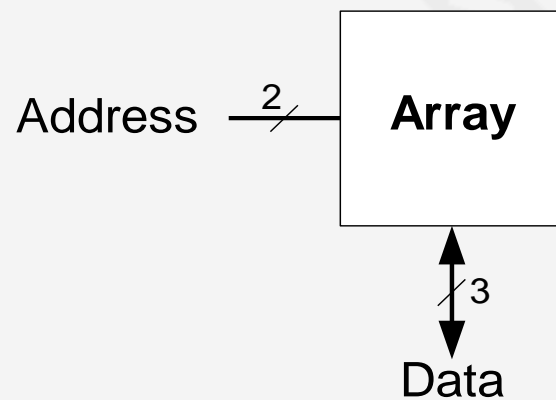


Address	Data			
11	0	1	0	depth ↑ ↓
10	1	0	0	
01	1	1	0	
00	0	1	1	
	width ← →			



存储器阵列分析

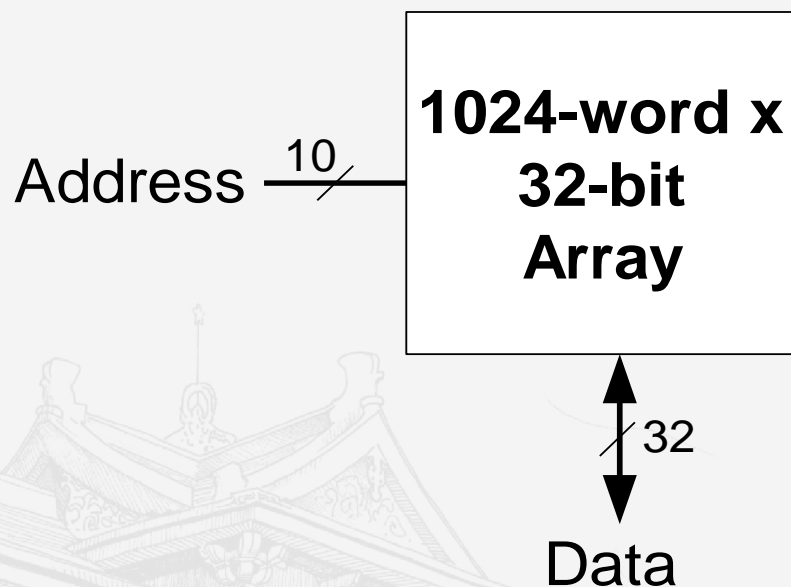
- 2位地址和3位数据的存储器阵列
 - 阵列深度：4（4行）
 - 数据字个数：4个
 - 字长：3 位
- 如右图所示，10 地址存放的数据为 100



Address	Data			
11	0	1	0	↑ depth ↓
10	1	0	0	
01	1	1	0	
00	0	1	1	
	↔ width ↔			



思考题



- 数据字个数? **1024**
- 字长? **32位**
- 容量? **32Kib=4KiB**



存储器阵列

Memory Arrays

□ 概述

□ 存储器的组织

□ DRAM与SRAM

□ 寄存器文件



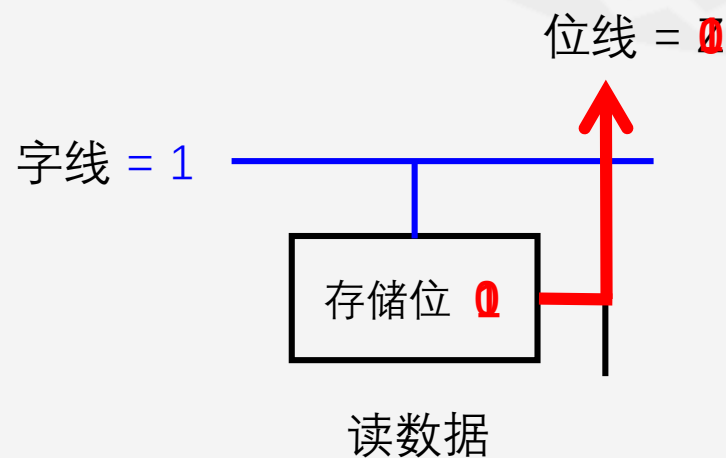
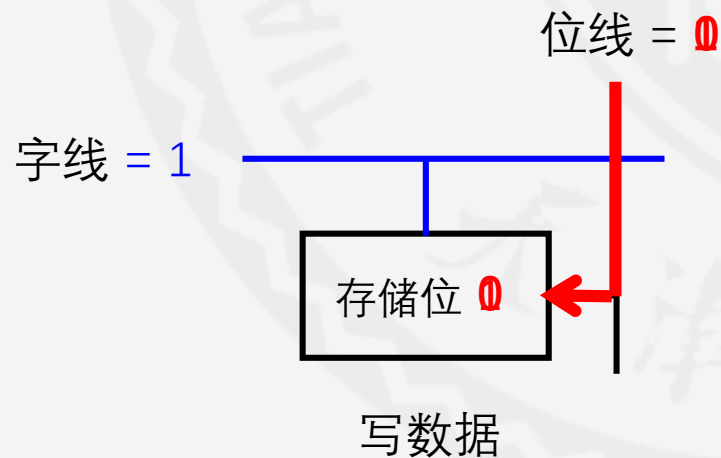
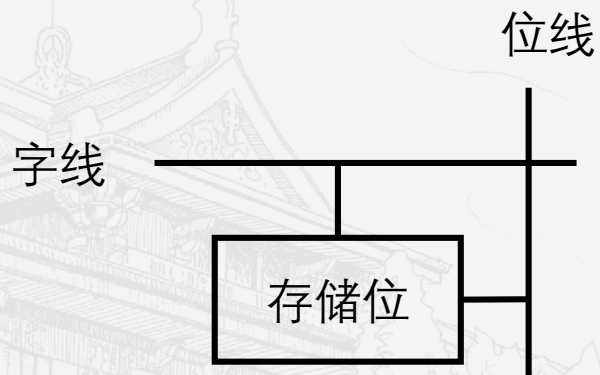


存储器的组织

Memory Organization

位单元

- 存储器阵列由位单元 (bit cell) 阵列组成
- 每个位单元存储1位数据
- 每一个位单元与一个字线 (wordline) 和一个位线 (bitline) 相连





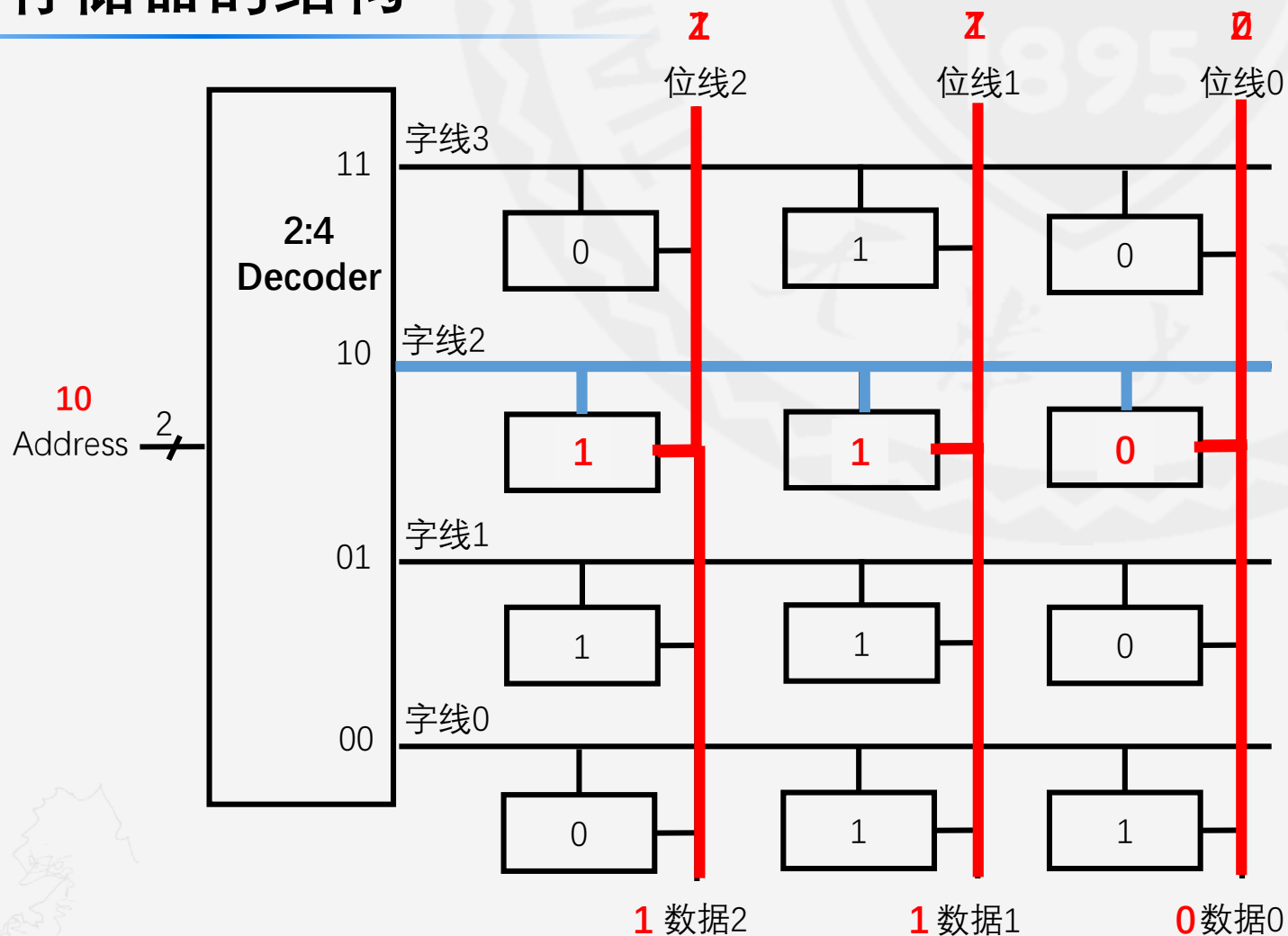
存储器的组织

Memory Organization

存储器的结构

字线:

- 与使能端相似
- 用于控制阵列中一行数据的读/写
- 对应着唯一的地址
- 同一时刻至多有一个字线为高电平



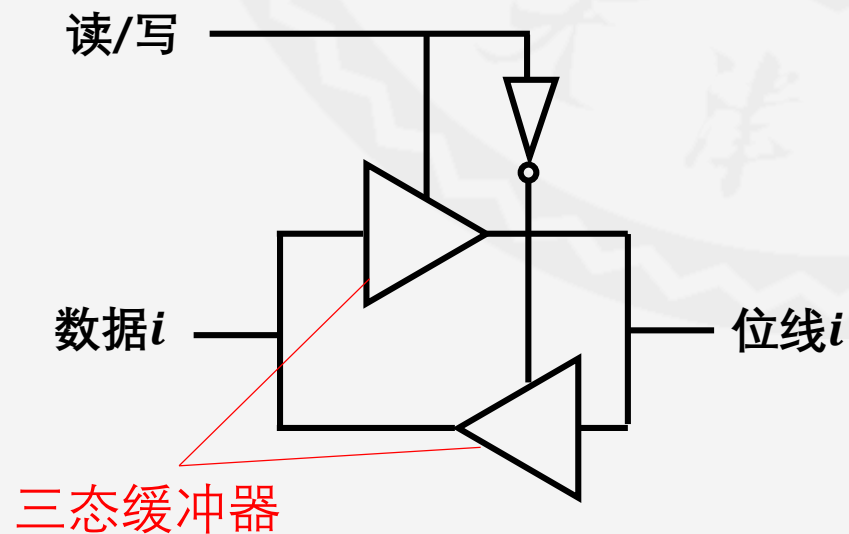
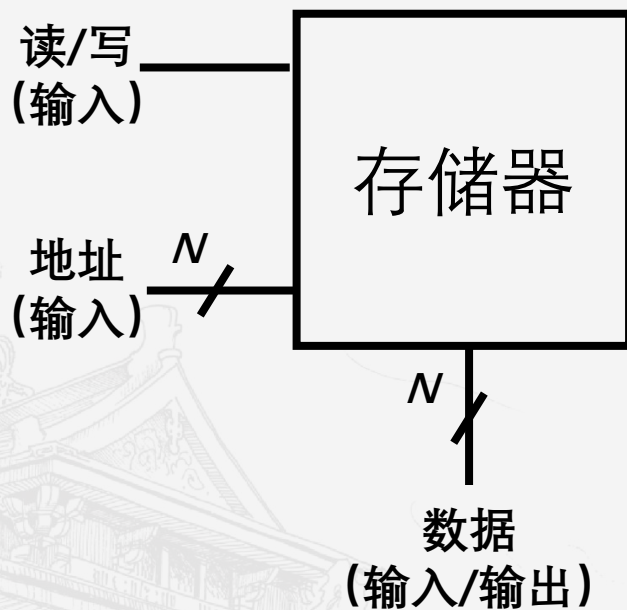


存储器的组织

Memory Organization

存储器端口

单端口可读/写存储器



双向数据接口的实现原理



存储器阵列

Memory Arrays

- 概述
- 存储器的组织
- 存储器的类型
- 寄存器文件





存储器的类型

- 两种主要的类型：
 - 随机访问存储器（Random Access Memory）：易失的（volatile）
 - 动态随机访问存储器（DRAM）：计算机的主存
 - 静态随机访问存储器（SRAM）：CPU中的高速缓存
 - 只读存储器（Read Only Memory）：非易失的（non-volatile）
- RAM和ROM的命名是由于历史的原因
 - ROM也是可以随机访问的，大多数现代ROM也是可读写的



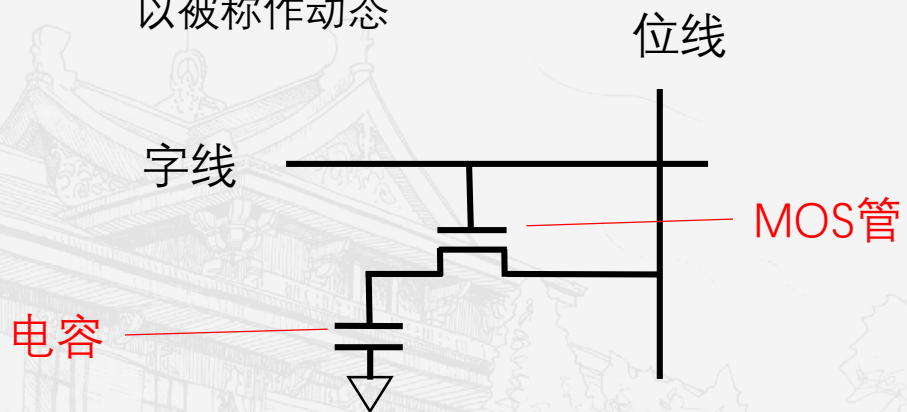
存储器的类型

Memory Types

存储原理

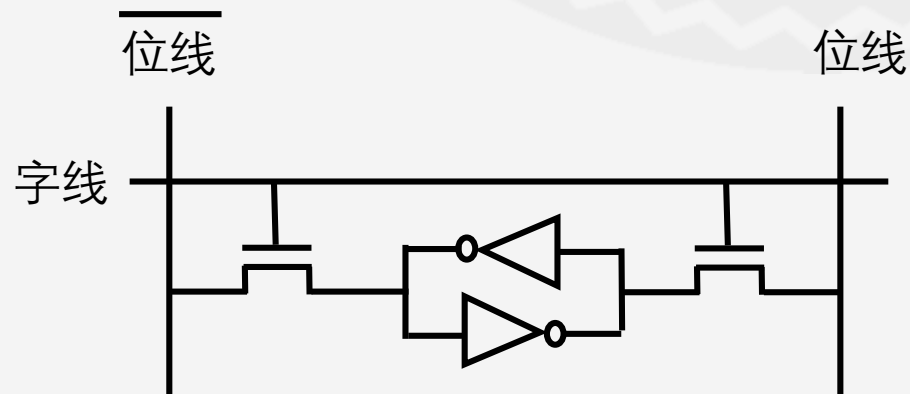
■ DRAM

- 数据存储在一个电容上
- 读操作后，存储的数据会被破坏
- 电容上储存的电荷也会慢慢泄漏
- 内容需要频繁刷新（读，然后重写），所以被称作动态



■ SRAM

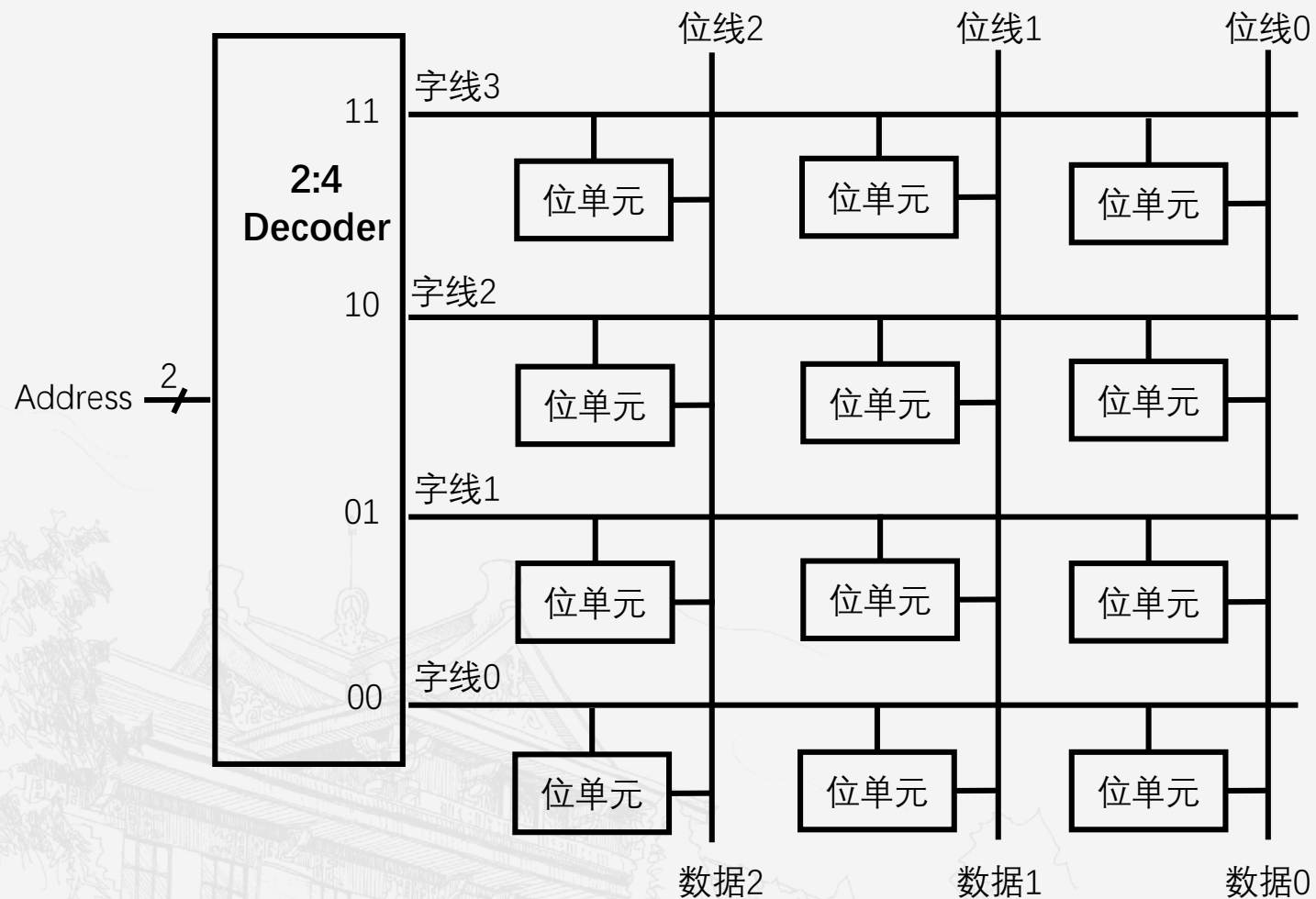
- 数据位存储在一个交叉耦合的反相器中
- 交叉耦合反相器具有很强的抗干扰能力
- 不需要刷新



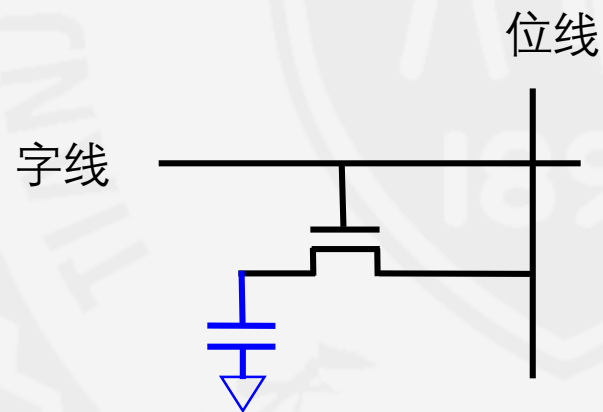


存储器的类型

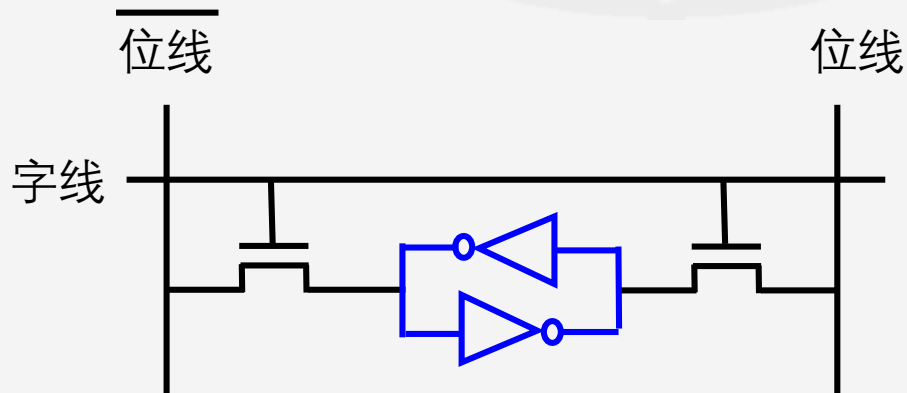
Memory Types



DRAM存储位单元



SRAM存储位单元





存储器的类型

Memory Types

几种存储器的比较

存储器类型	每个位单元的晶体管数	成本	延迟
触发器	≈ 20	高	低
SRAM	6	中等	中等
DRAM	1	低	高

ROM：读速度快，写速度较慢

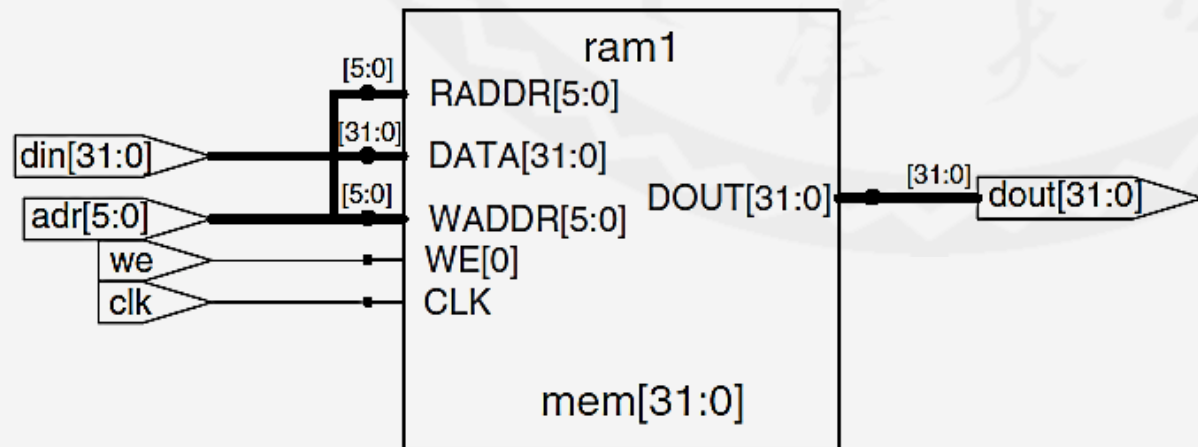


存储器的类型

Memory Types

RAM 建模

```
module ram # ( parameter N=6, M=32 )  
    ( input logic      clk, we,  
      input logic [N-1: 0]  adr,  
      input logic [M-1: 0]  din,  
      output logic [M-1: 0] dout);  
  
    logic [M-1:0] mem[2**N-1:0];  
    always_ff @(posedge clk) ** 幂运算,  $2^{N-1}$   
        if (we)  mem[adr] <= din;  
        assign dout = mem[adr];  
endmodule
```





ROM 建模

```
module rom ( input  logic [1: 0]  adr,  
             output logic [2: 0]  dout);  
  
    always_comb  
        case(adr)  
            2'b00: dout <= 3'b011;  
            2'b01: dout <= 3'b110;  
            2'b10: dout <= 3'b100;  
            2'b11: dout <= 3'b010;  
        endcase  
    endmodule
```



存储器阵列

Memory Arrays

- 概述
- 存储器的组织
- DRAM与SRAM
- 寄存器文件





寄存器文件

- 寄存器文件（Register files）通常是一个小型多端口SRAM阵列
- 右图是一个32寄存器×32位的3端口寄存器文件
- 有两个读端口 $A1/RD1$ 和 $A2/RD2$
- 一个写端口 $A3/WD3$
- 地址线均为5位，可寻址 $2^5=32$ 个寄存器
- 可以同时读两个寄存器和写一个寄存器

