SI100B EE Part Project • Tutorial 1

Sequence Logic & Introduction to RISC-V

Presented by
@Qihang Yan, yanqh2022@shanghaitech.edu.cn
@Quanyu Chen, chenqy2022@shanghaitech.edu.cn

Slides by
@Qihang Yan, yanqh2022@shanghaitech.edu.cn
@Yunxiang He, heyx1@shanghaitech.edu.cn
@Chaofan Li, licf1@shanghaitech.edu.cn

Contents

1. Background ->
2. Sequential logic ->
3. Register File ->
4. Memory Component in Logisim ->

0. About Project

0. About Project

Project 内容一览

- **学习架构**：掌握 RISC-V ISA，理解**软硬件契约**。
- **实现工具链**：编写一个 RISC-V **汇编器 (Assembler)**，体验符号到二进制的翻译。
- **实现硬件**：使用 Logisim 搭建一个单周期 CPU (RegFile, ALU, Control Unit)。
- **全栈联调**：编写汇编测试用例 → 汇编器生成机器码 → CPU 运行验证。
- **撰写报告**：详细展示测试用例设计 (Testbench) 和设计思路【可以使用LaTeX】。

> “ 关于具体的 Project 提交要求，我们会在这周内发布一个完整的文档。

0. About Project

- **关于 Project Report：这部分非常重要！**
  1. 我们会给出一个Reference CPU
  2. 我们**不要求**你的 CPU 行为和 Reference 完全一致（你可以有自己的微架构实现）。
  3. **但是**，你需要用**测试用例 (Test Cases)** 证明你的设计符合 ISA 规范。
  4. **First Principles**：不要只罗列连线，要解释**为什么**这么连。
- **Check 环节**：Face-to-Face Check，现场运行汇编代码。
- **AI Policy**：AI 的使用是被允许的！但是请确保你理解 AI 为你做了什么，“AI 这么说的，我也不太清楚”的回答可能会严重影响你的评分。
- **HDL Track**：如果你想挑战自我使用 Verilog/SpinalHDL，请尽早与我们沟通。

1. Background

1.1 The Toolchain: 从代码到电平

> “ 指令集是计算机中的一个“抽象层级”。什么是抽象层级?

1. 电平：原始电平计算机只认识电信号（HIGH和LOW）
2. 机器码：把低电平和高电平抽象成 0 和 1 的数字——机器码。
3. 汇编语言：把机器码抽象成一些字母，例如LDR（读），STR（写），ADD（加）。汇编器**Assembler**可以把这些字母转换成机器码。
4. 高级语言HLL：把汇编语言抽象成更多的字母，例如C语言，Python等。编译器可以把这些字母转换成汇编语言。
5. Math

Level	Representation
HLL	`if (a < b)`
↓	Compiler
ASM	`blt x1, x2, L1`
↓	Assembler
Bin	`1100011...`
↓	MOS
HW	High/Low Level

> “ 编译器和汇编器的操作就是把我们的意图（计算机无法理解）转换成对计算机有意义的电信号。那，怎样的电信号才是有意义的呢?

指令集就是**有意义的电信号的合集**，是软硬件之间的**契约**，其中每个指令对应着一个**电路操作**——你给电路输入一些电信号，它就会返回一些电信号——意味着硬件电路是根据指令集来设计的。

1.2 IC Design Flow

这个Project，这在真实工业界属于哪个环节?

阶段	核心任务	涉及角色
0. ISA Definition	**定义契约** (RISC-V spec)。手册规定 `ADD` 的行为，但不规定电路怎么搭。	Architect
1. Microarchitecture	**设计蓝图**。决定用 5 级流水线还是乱序执行? **(Project 核心)**	**Logic Designer**
2. RTL Design	**写代码** (Verilog)。用 HDL 描述行为 (Logisim 是图形化 RTL)。	**Logic Designer**
3. Physical Design	**物理实现**。综合 (Synthesis) → 布局布线 → 版图。	PD Engineer
4. Fab	**制造**。台积电/中芯国际光刻。	Foundry

> “ 这一部分相信大家主要在正课上进行了学习。

1.3 CISC vs. RISC:

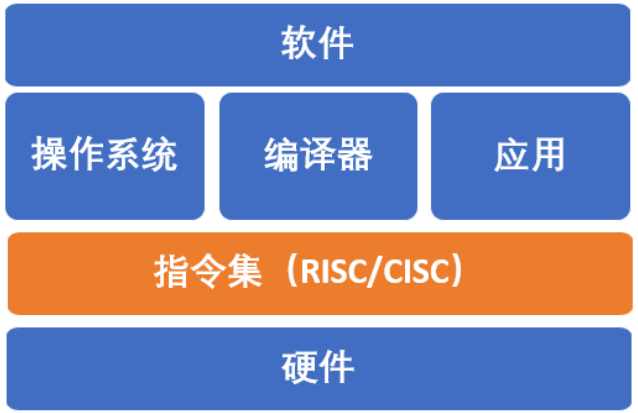
What is RISC-V?

指令集可以被分为两类:

- 复杂指令集CISC（Complex Instruction Set Computer）：一条指令就可以实现一个复杂的功能
- 精简指令集RISC（Reduced Instruction Set Computing）：RISC中的一条指令大多是“基本指令”，它是通过多条指令组合完成一个复杂的功能

RISC-V 就是是一个精简指令集!

> “ 值得注意的是，指令集都是人为规定的!



1.3 CISC vs. RISC:

From Instruction Set to Hardware

指令集定义了硬件支持的基本操作，这包括:

1. Datapath
    1. 算术运算（加减乘除）
    2. 逻辑运算（与或非）
    3. 数据操作（加载、存储）
2. 控制流（分支、跳转）。

- 因此，指令集本身并不先于电路产生，它首先需要寄存器、缓存、执行单元等硬件资源的支持。
- 相比x86，RISC-V指令简单、长度固定，硬件设计更简洁，流水线性能更高。

Example：

RISC-V 的 ADD 指令(`add rd, rs1, rs2`)定义为两个寄存器（`rs1`, `rs2`）相加并将结果存储在目标寄存器(rd)中。硬件通过设计算术逻辑单元（ALU）来实现加法操作，同时更新目标寄存器的值。

为了实现简单的加法运算，我们至少需要设计寄存器、ALU等模块。

接下来，让我们一起开始造轮子吧！

2. Sequential logic

2. Sequential logic



2. Sequential logic



2. Sequential logic



2.1 引入时间：从组合到时序

组合逻辑 (Combinational Logic):

- $Output = f(Input)$
- 无记忆性，像加法器。输入一撤，结果就没了。

时序逻辑 (Sequential Logic):

- 引入 **反馈 (Feedback)** 和 **状态 (State)**。
- $State_{next} = f(Input, State_{current})$
- **存储器**是时序逻辑的核心。

访问时间	存储元件	容量
1ns	**Register**	**<1KB**
2ns	Cache (L1-L4)	32MB
10ns	Main Memory	8GB
10ms	Disk	1TB

“现代计算机中存储部件的对比。Project 中我们主要设计最顶层的 **Register**。

2.1 引入时间：从组合到时序

组合逻辑电路

定义：组合逻辑电路（Combinational Logic Circuits）是没有存储元件的电路，其输出仅依赖当前输入。

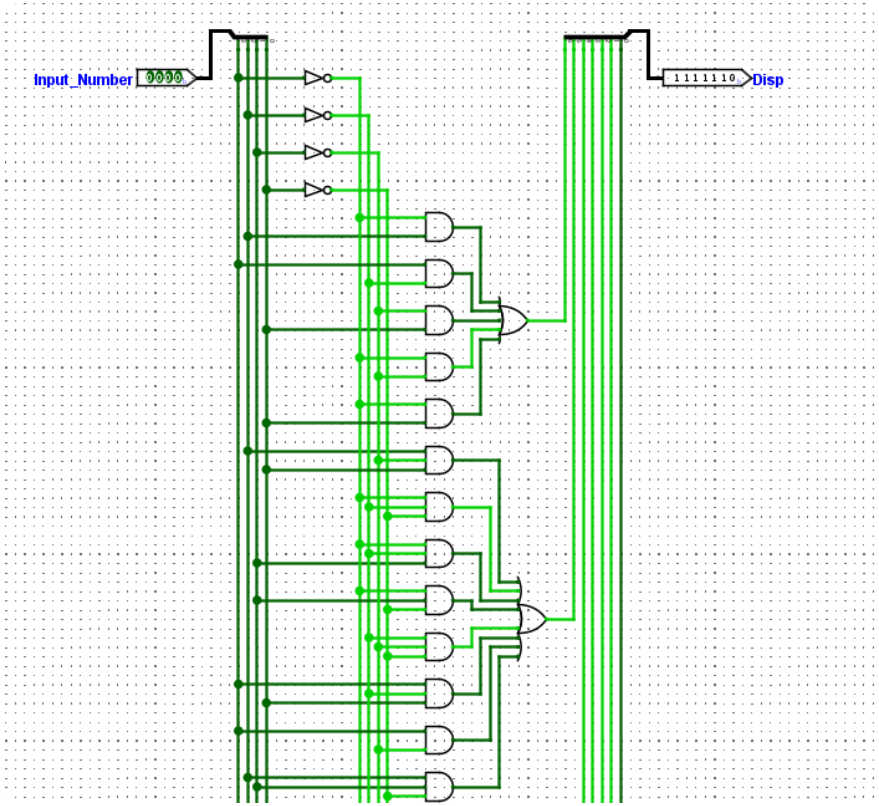
特点：

- 无记忆功能：组合逻辑电路没有存储元素，无法记忆以前的状态。每个时刻的输出仅与当前的输入有关。
- 确定性：在任何给定时刻，输出是输入的确定函数。
- 主要应用：用于算术运算、数据处理和编码、解码等场景。常见的组合逻辑电路包括加法器、减法器、编码器、解码器、多路复用器和去多路复用器。
- 实例：
  - 加法器：例如全加器（Full Adder）。
  - 多路复用器（MUX）：根据选择输入，选择一个特定的输入信号传递到输出端。

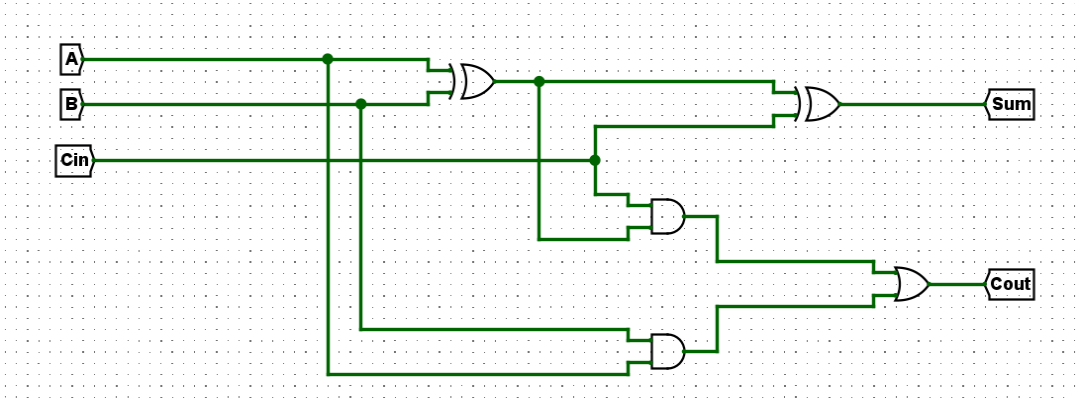
2.1 引入时间：从组合到时序

example

Decoder



Full adder

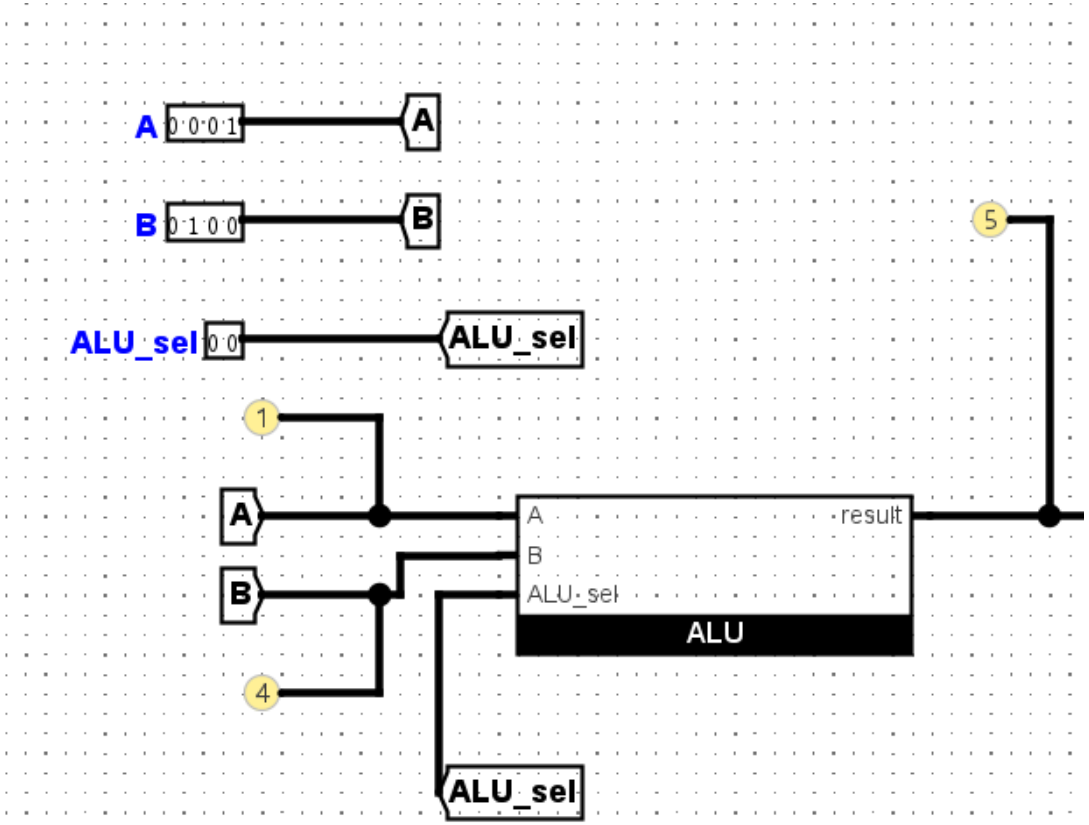


2.1 引入时间：从组合到时序

example

即时性:输入到输出往往只需要一个很短的延时

ALU



2.1 引入时间：从组合到时序

时序逻辑电路

定义：不仅依赖当前输入，还依赖于以前的输入和状态，因为它们包含存储元件，如触发器（Flip-Flops）。

特点：

- 有记忆功能：能够记忆以前的状态，输出不仅取决于当前输入，还取决于电路的历史状态。
- 时钟信号：大多数时序逻辑电路都依赖时钟信号进行同步操作。时钟信号帮助协调电路中各部分的状态变化。

实例：

- D触发器：有数据输入和时钟信号，输出会在时钟信号的上升或下降沿根据输入的数据改变。
- 计数器：例如二进制计数器，它随着时钟信号的每个周期递增计数。

2.2 从Latch 到 Flip-Flop: Latch(锁存器) - 电平触发

- **结构**：简单的反馈环路 + 门控。
- **行为**：$CLK = 1$ 时 **透明 (Transparent)**。输入直接穿透到输出。
- **缺陷**：像一扇一直开着的门，很难精确控制数据只移动“一步”。

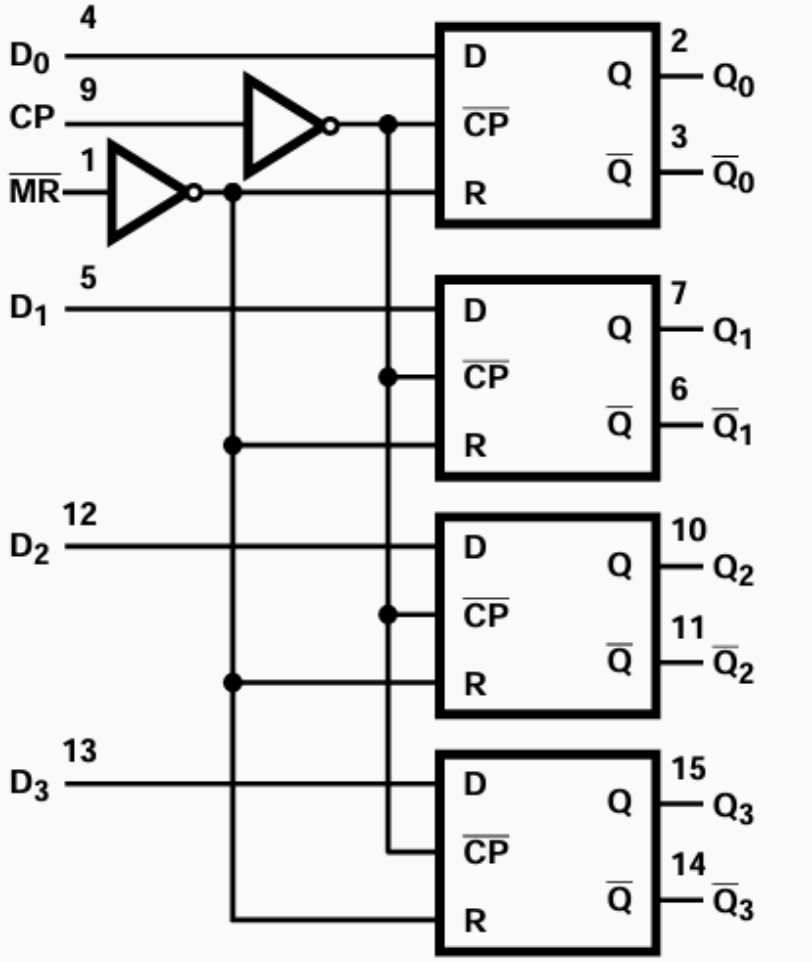
TRUTH TABLE

INPUTS			OUTPUTS	
RESET ($\overline{MR}$)	CLOCK CP	DATA D_n	Q_n	$\overline{Q}_n$
L	X	X	L	H
H	↑	H	H	L
H	↑	L	L	H
H	L	X	Q_0	$\overline{Q}_0$

H = High Voltage Level, L = Low Voltage Level, X = Don't Care, ↑ = Transition from Low to High Level,
Q_0 = Level Before the Indicated Steady-State Input Conditions Were Established.

2.2 从Latch 到 Flip-Flop: `D Flip-Flop`(触发器)- 电路模型

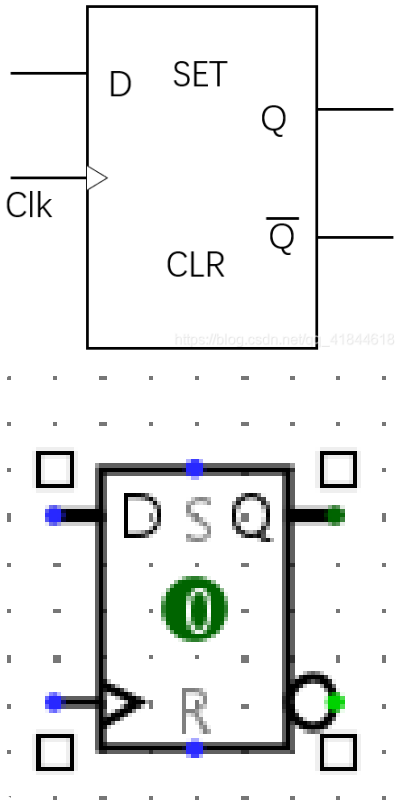
- **结构**：**Master-Slave (主从结构)**。两个 Latch 串联。
  - Master: $\neg CLK$ 导通; Slave: CLK 导通。
- **行为**：像两道气闸门。CLK 上升沿瞬间，Master 关闭，Slave 打开。
- **意义**：让状态只有在时钟上升沿才可以被改变，实现了**离散时间**的同步。



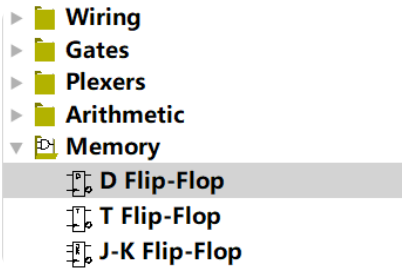
2.2 从Latch 到 Flip-Flop: D Flip-Flop(触发器)- 抽象表现

- R：Reset
- clk(小三角)：时钟信号
- D：写入信号
- Q：当前状态
- $\overline{Q}$：Q取反
- 逻辑：Clk 信号上升沿时 $Q = D$

电路符号



Logisim中的位置



真值表：

D	CLK	Q	$\overline{Q}$
0	↑	0	1
1	↑	1	0
×	0	Q	$\overline{Q}$
×	1	Q	$\overline{Q}$

2.2 从Latch 到 Flip-Flop: D Flip-Flop(触发器)- 抽象表现

D触发器写入信号D

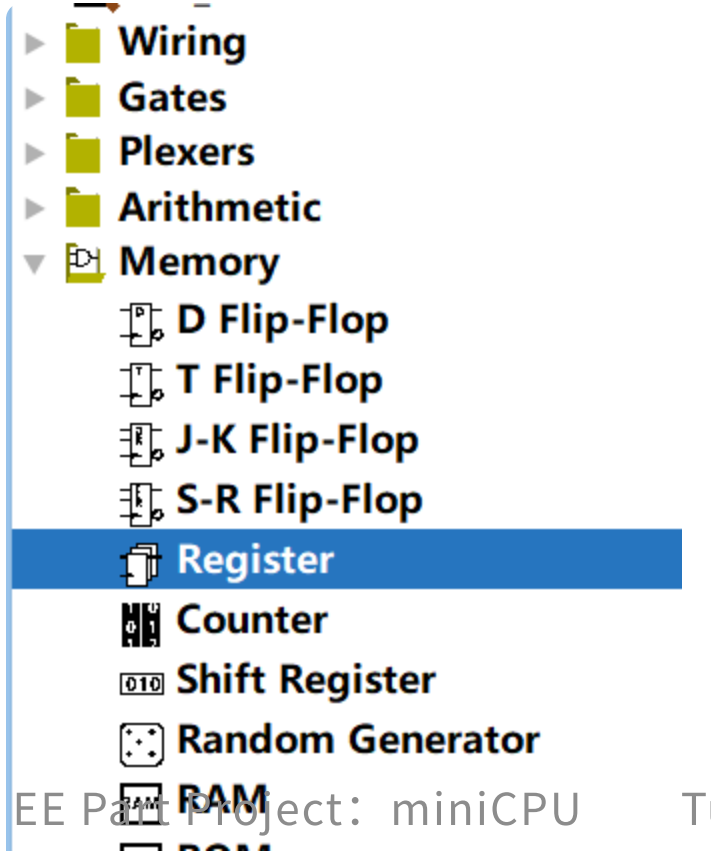
- 在时钟上升沿其他时间内，Q不随除了reset之外的任何信号变化而变化
- Q只在时钟上升沿时候变化
- Q变成上升沿到来前一瞬间的D的值
  - 假设clk在$t = 0$时由低转高
  - 那么**Q在0^+的值**由 **0^-时D的值**决定

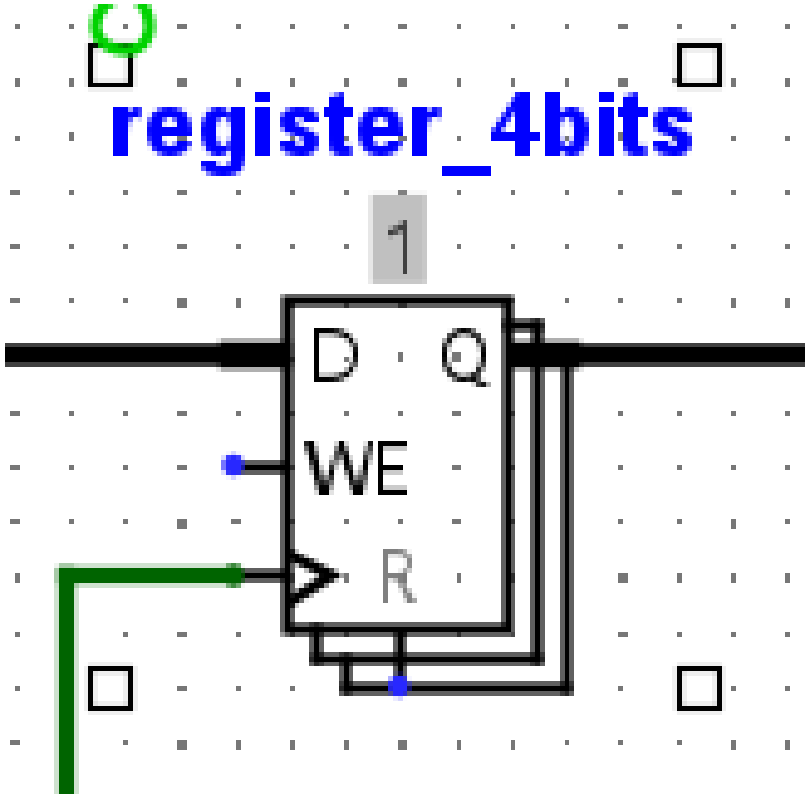
真值表:

D	CLK	Q	$\overline{Q}$
0	↑	0	1
1	↑	1	0
×	0	Q	$\overline{Q}$
×	1	Q	$\overline{Q}$

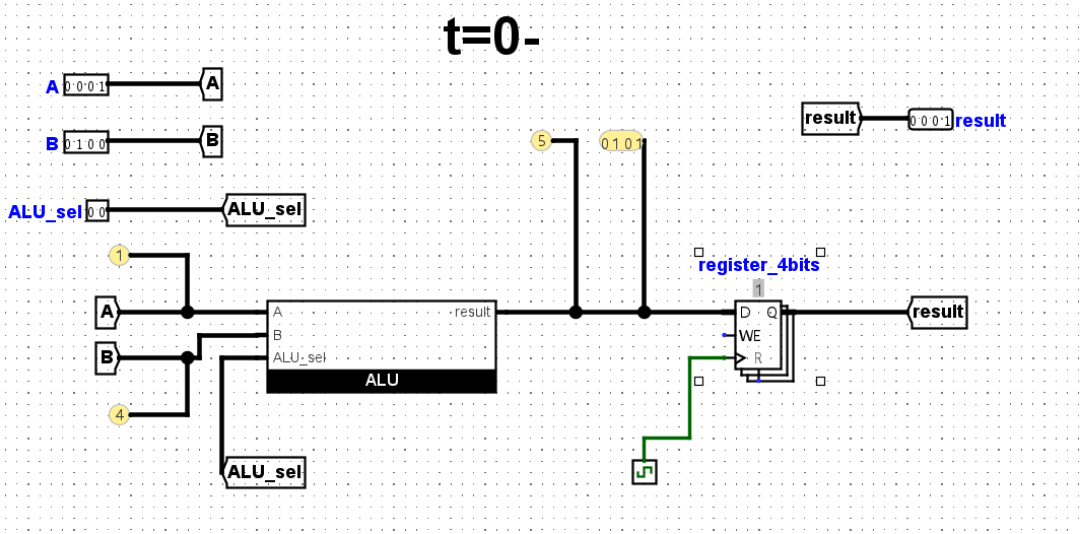
2.2 从Latch 到 Flip-Flop: 寄存器（register）

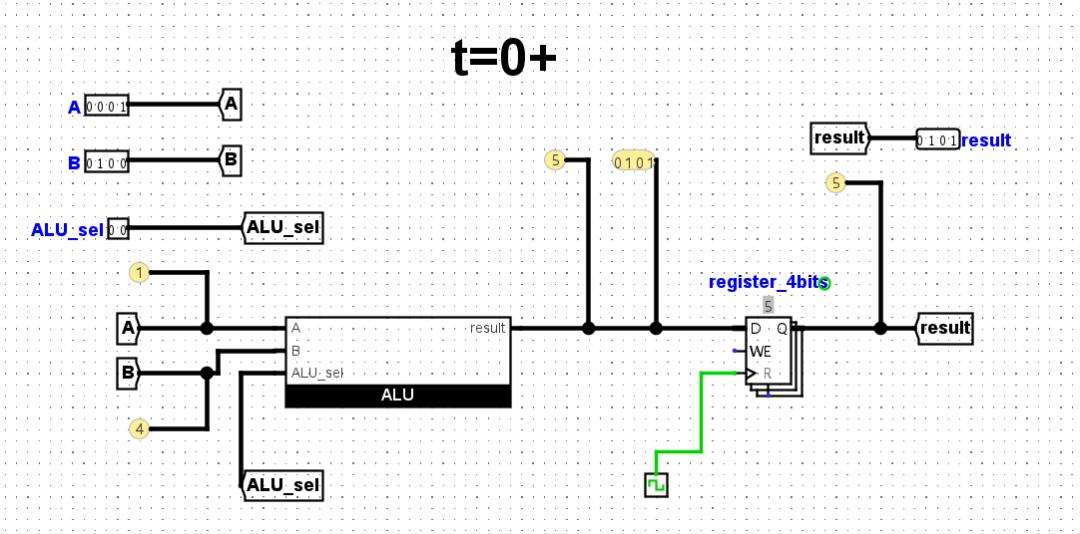
- 由**触发器**组成的存储元件
- 多bit寄存器由多个触发器组成，D和Q变为多位宽信号





2.2 从Latch 到 Flip-Flop: 寄存器（register）的数据传递





2.2 从Latch 到 Flip-Flop: 寄存器（register）的数据传递

三国杀：判定

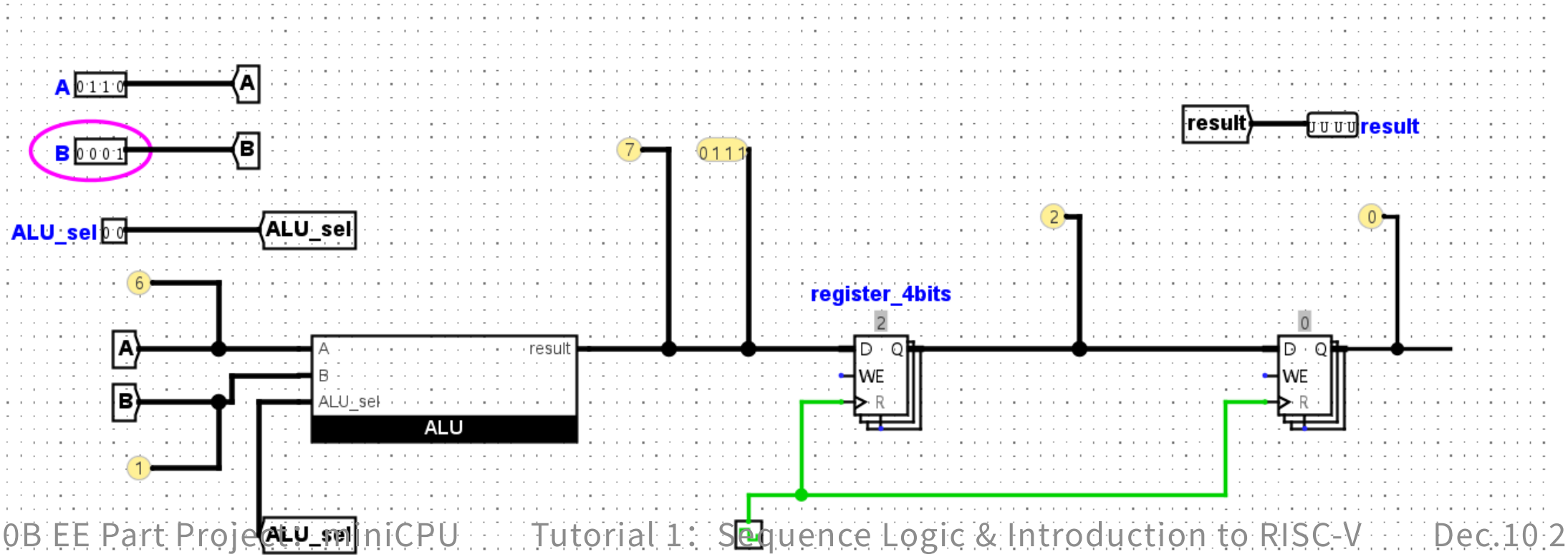


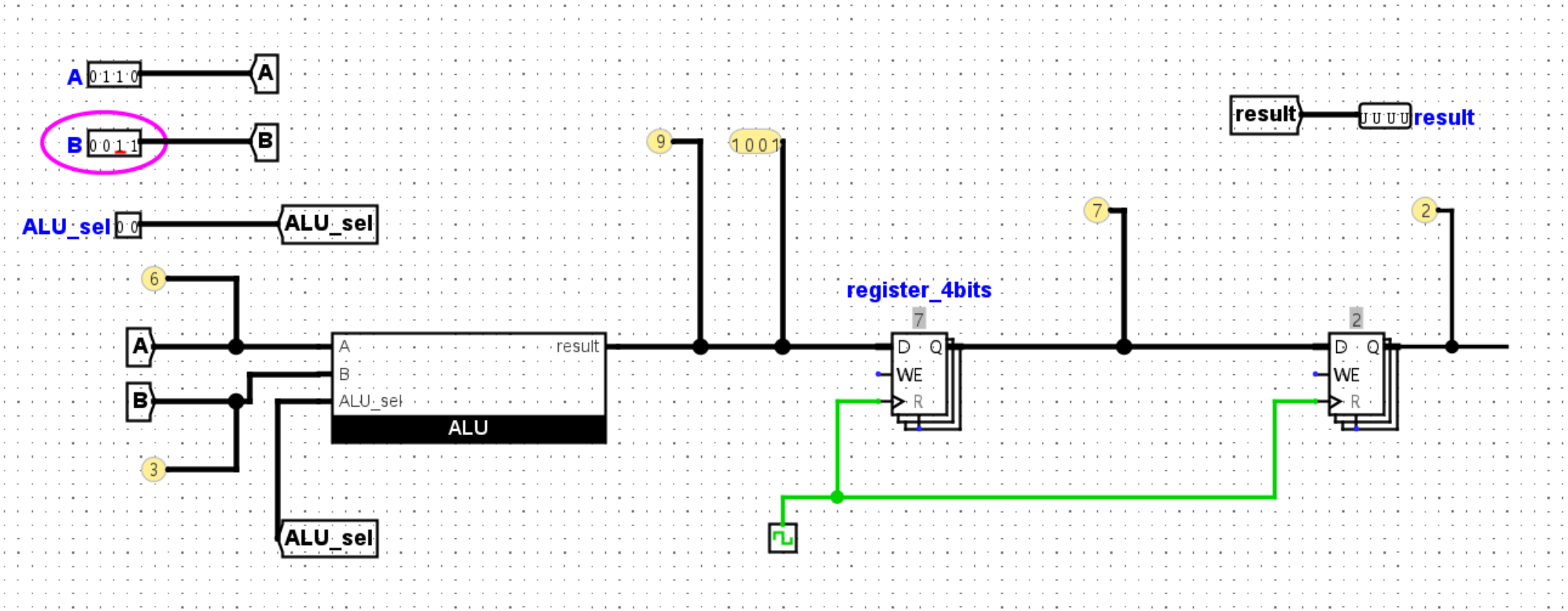
- 简化模型，每位玩家只有判定阶段和出牌阶段，忽视角色技能和”无懈可击“
- 你的判定结果永远取决于**判定阶段开始前一瞬间**时的牌库的第一张牌，和你的判定区状态的**组合逻辑**

2.2 从Latch 到 Flip-Flop: 寄存器（register）的数据传递

级联 (同步时序逻辑电路)

> “ 依赖于Master和Slave触发器的**导通互斥性**




A
0 1 1 0
A
B
0 0 1 1
B
ALU_sel
0 0
ALU_sel
6
A
B
3
A
B
ALU_sel
result
ALU
ALU_sel
9
1 0 0 1
register_4bits
7
D
Q
WE
R
7
2
D
Q
WE
R
2
result
U U U U
result

注意

- clk的接口只能连clk，不允许把clk&A 作为时钟信号
  - 现实中的数字电路会对clk进行滤波和消抖处理，保证clk是较平滑的方波，而组合后的信号不好保证
- WE信号是写入使能（Write Enable），悬空状态下默认为1（总是允许写入），决定输入信号能不能写入的时机也是上升沿的前一瞬间

3. Register Files

3. Register Files: The CPU's Workbench

Definition

A register file is an array of processor registers in a central processing unit (CPU). The instruction set architecture of a CPU will almost always define a set of registers which are used to stage data between memory and the functional units on the chip. The register file is part of the architecture and visible to the programmer, as opposed to the concept of transparent caches. In simpler CPUs, these architectural registers correspond one-for-one to the entries in a physical register file (PRF) within the CPU. More complicated CPUs use register renaming, so that the mapping of which physical entry stores a particular architectural register changes dynamically during execution.

> “ What's this?

3. Register Files: The CPU's Workbench

核心区别：Register vs Register File

Register vs. Register File

- Register：CPU在进行运行时，往往需要一部分存储空间暂时（临时）存储在运算过程中产生的中间数据，而这个存储空间往往就是寄存器。
- Registers File：寄存器堆是**由一组寄存器组成的二维阵列**。为方便访问其中的寄存器，对寄存器堆中的寄存器进行统一编码，称为寄存器编号或者寄存器地址，每个寄存器均通过制定的寄存器编号进行访问。

可以注意到，“寄存器堆”其实就是一组被有序排列并编号的寄存器。

在RISC-V中，寄存器堆的大小一般为32个寄存器，分别编号为x0-x31。

在RISC-V等Load-Store Architecture，ALU无法直接处理RAM中的数据，必须将Register File作为中转站.

虽然ALU的主要数据来源为Register，但也可以接受来自于Program Counter的地址和来自Instruction Decoder的立即数（Immediate Number）等位置的数据作为输入.

3.1 Register Files's I/O

端口定义 (Ports)

- **2 Read Ports** (`rs1`, `rs2`):
  - 支持 `ADD x3, x1, x2`。
  - 需要同时读出两个数。
- **1 Write Port** (`rd`):
  - 写回运算结果。
  - 需要 `RegWrite` (WE) 信号控制。
- **Clock**: 同步写操作。

输入信号	信号位宽	描述
`clk`	1	时钟信号
`we`	1	Write Enable 信号，决定寄存器是否被写入
`rs1`, `rs2`	5	两个输入寄存器的地址信号
`rd`	5	被写入的寄存器的信号
`D`	32	写入 `rd` 寄存器的数据

输出信号	信号位宽	描述
`X_rs1`, `X_rs2`	32	`rs1` 和 `rs2` 对应的寄存器中的值

3.2 Read Logic: Multiplexing

如何从 32 个寄存器中挑出 1 个?

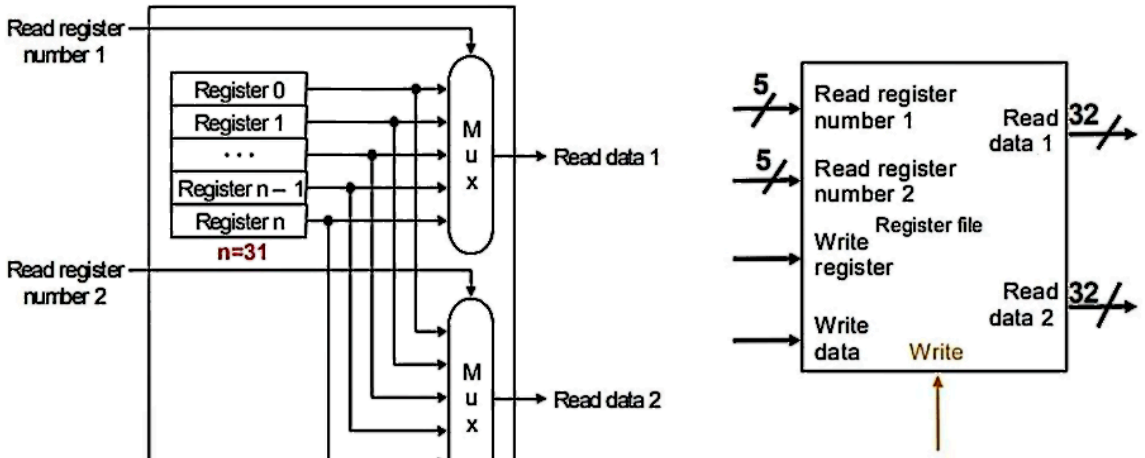
- **原理**：多路复用 (Multiplexing)。
- **组件**：**32-to-1 MUX**。

逻辑公式:

$$Data1 = \text{MUX}(Reg[0]...Reg[31]; rs1)$$

$$Data2 = \text{MUX}(Reg[0]...Reg[31]; rs2)$$

> “ **注意**：因为有两个读端口，你需要**两个独立**的 MUX!



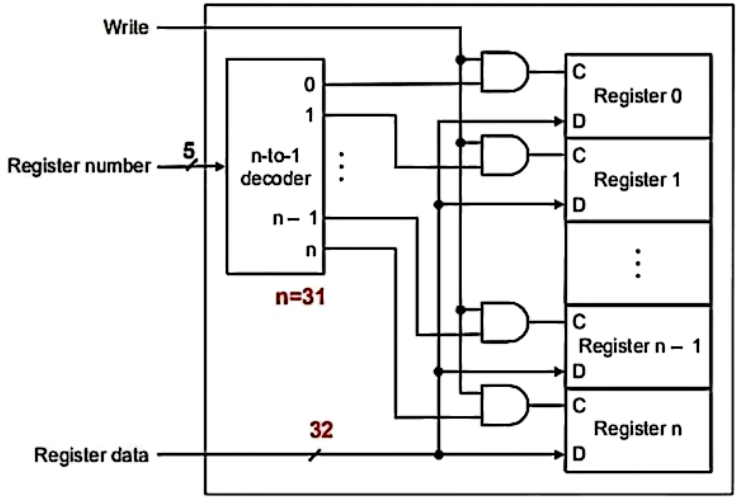
3.3 Write Logic: Decoding

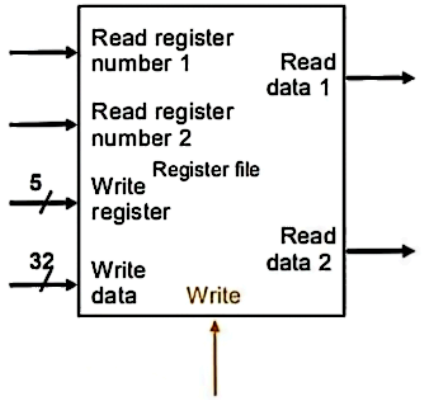
如何只写入目标寄存器?

- **原理**：译码 (Decoding)。
- **组件**：**Decoder + AND Gates**。

逻辑推导：

1. **译码**：$Signal_i = (rd == i)$ (One-hot)
2. **使能**：$WE_i = Signal_i \wedge RegWrite$
   - 这里的`RegWrite`是全局写使能信号
3. **连接**：WE_i 接到 Reg[i] 的 Enable 端





> **x0 处理**：确保 x0 的 WE 恒为 0，或

拓展：深入Flip-Flop的设计- Setup & Hold Time

理想模型认为状态瞬间改变，但物理世界存在延迟。这是**Master-Slave**结构带来的物理代价。

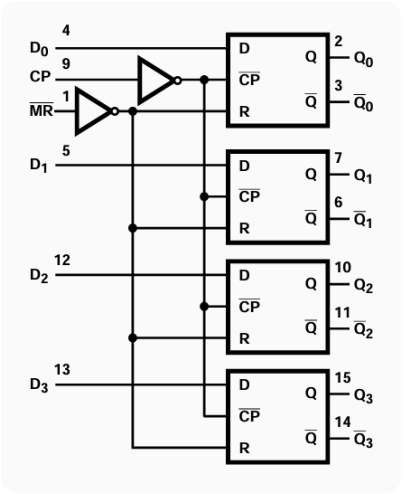
1. **Setup Time (t_{su}): "Master 关门前，客人必须坐稳。"**
   - 在上升沿到来**前**，输入 D 必须保持稳定。
   - 后果：否则 Master 关门瞬间会截断电平，导致 **亚稳态 (Metastability)**。
2. **Hold Time (t_h): "Master 关门后，必须确认门锁死了。"**
   - 在上升沿**后**，输入 D 必须继续保持稳定。
   - 后果：防止数据在门完全关死前溜走。

> “ **Takeaway**: 时钟频率不能无限高，因为数据必须要在 Setup Time 之前跑完组合逻辑电路。

拓展：Register 内部结构 (74HC175 Example)

D 触发器真值表 (Edge Triggered):

CLK	D	Q (Next State)
↑	0	0 (Sampled)
↑	1	1 (Sampled)
0/1	X	Q (Hold)



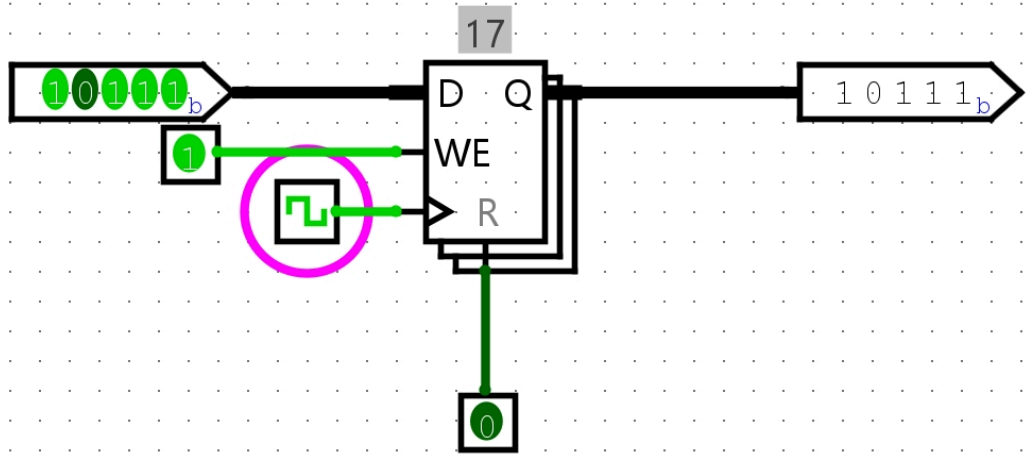
图示芯片 (74HC175) 内部包含 4 个 D-FF，共用 CLK 和 Clear 信号。这其实就是一个 **4-bit Register**。

> “ Logisim 中的 Register 本质是 Flip-Flop 的阵列。

Register (100,300)	
FPGA supported:	Supported
Data Bits	8
Trigger	Rising Edge
Label	
Label Location	↑ North
Label Font	SansSerif Bold 16
Show in Registers Tab	No
Appearance	Logisim-Evolution

4. Memory Component in Logisim

4.1 Register in Logisim



关键设置:

- **Trigger**: Rising Edge (上升沿)。不要选 High Level (那是 Latch!)。
- **Data Bits**: 32 (RISC-V 标准)。
- **Label**: 清楚标记 (如 PC, IR)。

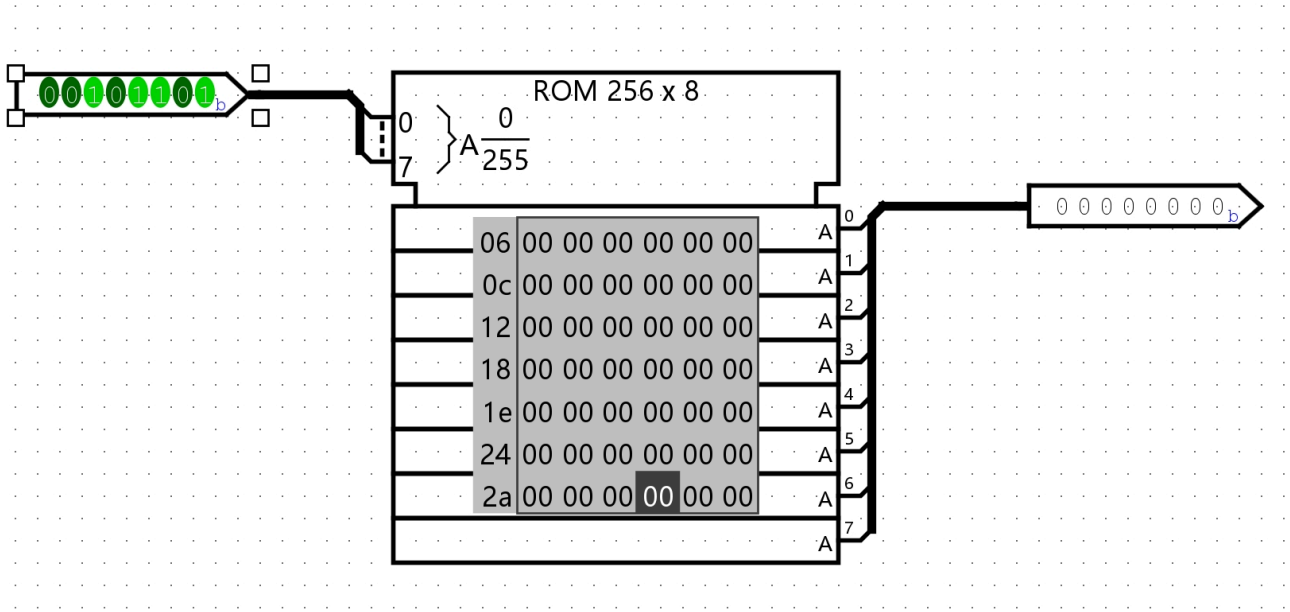
调试技巧:

- 红色线：多驱动冲突 (Conflict)。
- 蓝色线：未连接 (Floating)。
- `Reset` 信号通常是**异步**的。

Try Register:

- Try to pass data

4.2 ROM (Instruction Memory)



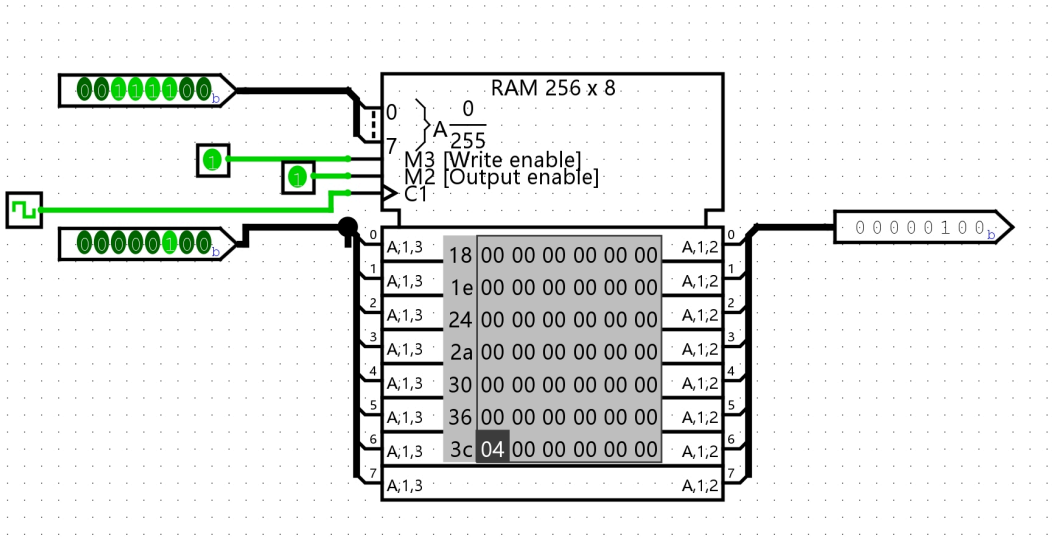
ROM (Read-Only Memory)

- **性质**：组合逻辑。给定地址，立即出数据。
- **用途**：存储指令 (Text Segment)。
- **加载**：右键 -> Load Image -> 加载机器码文件。

Try ROM:

- Create
- Modify
- Read
- Try different addresses

4.3 RAM (Data Memory)



RAM (Random Access Memory)

- **性质**：时序逻辑 (写操作依赖 CLK)。
- **配置建议**：
  - Select **"Separate Load and Store ports"**.
  - 这样可以同时支持 Read 和 Write 连线，符合哈佛架构数据通路。
- **注意**：RAM 的 Address width 也是重要参数。

Try RAM:

- Create
- Modify
- Read & Write
- Try different addresses

4.4 Build Your Program Counter

The Heartbeat of CPU

PC 本质上只是一个 **32-bit Register**，但它决定了 CPU 的控制流。

状态转移方程 (State Transition):

- **Sequential Execution**:

$$PC_{next} = PC_{current} + 4$$

> “ Why +4? RISC-V 指令长 32-bit (4 Bytes)，内存按字节编址。

- **Branch/Jump** (Future): $PC_{next} = Target$

构建一个mini PC:

1. **Storage**: 放置一个 4-bit Register。
2. **Compute**: 连接 Adder 和常量
3. **Feedback**: Adder 输出 → Register 输入 `D`。
4. **Init**: 务必连接 `Reset` (通常初始化为 0x0)。

> “ **Critical Thinking**:
> 如果没有 **Clock** 控制 Register (例如误用了 Latch)，这个电路会发生什么?

内容预告

内容预告

下次 Tutorial，我们将开展：

- **Release Project Document**: 我们会 Release 我们的 Project 要求文档
- **RISC-V ISA Deep Dive**: 开始进入 RISC-V 指令集的学习
- **Logisim Advanced**: Logisim 的进阶使用技巧，并开始设计我们的 mini CPU（新建文件夹！）
- **Assembler**: 随着指令集的学习，你可以开始实现你的 RISC-V 汇编器了。我们会在最后提供一个把汇编器结果转化到 Logisim Memory File 的脚本

> **Homework**:
>
> 1. 复习今天的时序逻辑
> 2. Build A Program Counter
> 3. 预习 RISC-V 基础指令格式。

Welcome to the world of Computer Architecture!

@ Presented by Qihang Yan
@ ShanghaiTech University