尚硅谷嵌入式技术之从零搭建计算机

（作者：尚硅谷研究院）

版本：V1.0.5

第 1 章 数字电路基础

1.1 引言

数字电路是现代科技和工程领域中不可或缺的基础。从计算机系统到通信设备，从家庭电子产品到工业自动化，数字电路无处不在，影响着我们的生活和工作。本章节旨在向读者介绍数字电路的基本概念、原理和应用，为后期学习单片机开发打基础。

1.2 二进制数据表达

1.2.1 二进制简介

二进制是一种数字表示法，它使用两个不同的数字符号，0 和 1，来表示数值。下面是一些二进制的基本概念：

1）位（bit）

不同于十进制中，每个位可以用 0-9 表示，二进制中每位都是 0 或 1，每个二进制位我们称之为一个 bit，可以表示两种状态。位一般用 b 代替，例如 8b 代表 8 个 bit。

2）字节（byte）

8bit 组成一个字节，总共可以表示 256 种状态。字节一般用 B 代替，例如 8B 一般表示 8 个字节。

3）其他计量单位（K，M，…）

K：表示 2 的十次方，例如，1KB 就是 1024B

M：表示 2 的 20 次方。例如：1MB 就是 1048576B

G：表示 2 的 30 次方。

T：表示 2 的 40 次方。

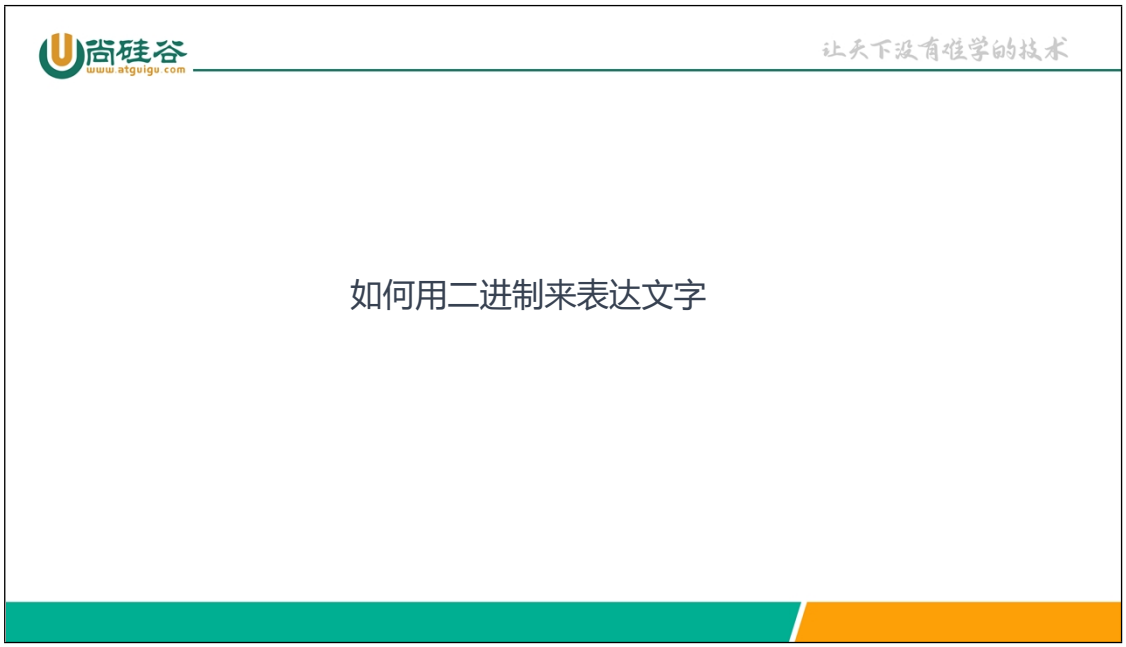
比这些更高的计量单位，尽管有定义，但是现实中我们很少用到，这里就不再赘述。

二进制是计算机科学的基础，现实中的绝大部分信息都可以完全数字化，并进一步用

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

二进制来表示和存储。

1.2.2 用二进制表达文字



1.2.3 用二进制表达图片

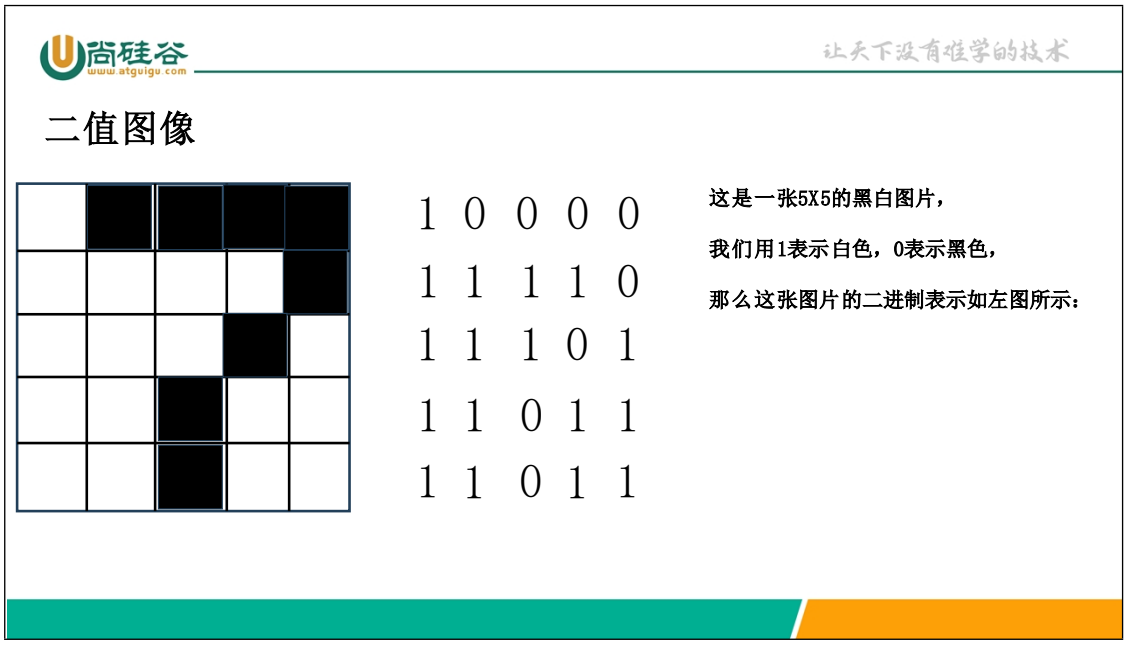
图片数字化的过程，分为几步：

1）图片像素化

现实中的图片是连续的，但如果想用数字表示图像，就需要将一张完整的图像，横纵向分别切成很多份，从而拆成一个一个像素点，每个像素点是一个色块，而横纵切分的份数，我们称之为分辨率。例如，我们将一张 4 比 3 比例的图像，横向拆成 800 份，纵向拆成 600 份，那么我们就将一个图像拆成了 480000 个像素，而这张图像的分辨率为 800x600。由此可见，图像的分辨率越高，图像就越清晰。

2）像素数字化

每个像素就是一个颜色块。具体的颜色，可以用红绿蓝三原色调配成。而每种原色，就可以用一个数字表示其深浅。例如，我们用一个字节（0-255）表示一种颜色的深浅，那么一个像素点就可以用三个字节表示。颜色的深浅是连续的，用 0-255 这种级别如果不够精确，我们可以选择用两个字节表示一个颜色（0-65535）。像素的深浅级别我们称为“位深”，例如 0-255 为 8 位深，0-65535 是 16 位深。位深越大，颜色切分越连续，越不容易出现色阶。现今网络上的图像一般都是 32 位深。



1.2.4 用二进制表达声音

声音的数字化和图像的过程非常类似：

1）采样

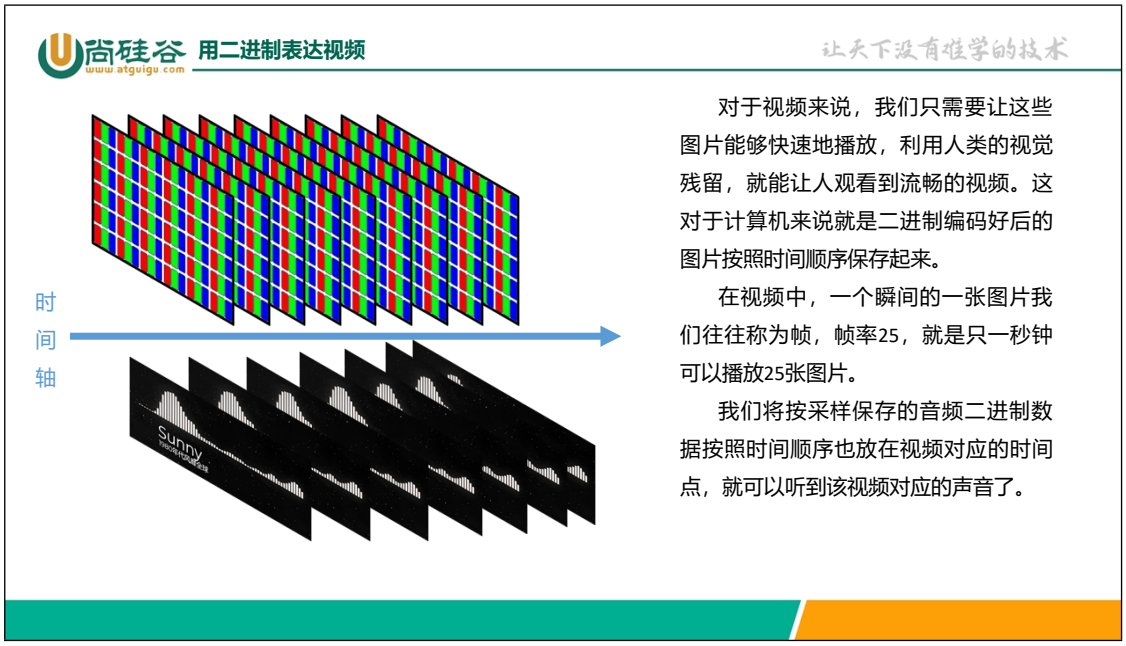
声音来自于物体的振动，是一种连续的波形。我们如果想用数字表达声音，还是要将其离散化。首先我们将一段时间（例如 1 秒钟）的连续声波分成很多份，并将每一份记录一个平均振幅。拆分的份数，我们称之为采样率。拆分过程中必然会产生信息丢失，而采样率越大，信息丢失越少。目前比较常用的采样率是 44100HZ 或者 48000HZ。

2）量化

每个采样我们还要用一个数字表示它的高度（振幅），这又是一个连续量离散化的过程，跟图像的颜色深浅类似，我们也可以用 0-255 或者 0-65535 表示振幅的高度。这个数字的取值范围叫做声音的位深。目前通用的位深一般是 16bit 或者 24bit。

1.2.5 用二进制表达视频

视频就是连续的图片和声音。当然，如果只是耿直地将图片拼接到一起，那么视频的体积会非常大。实际上视频是通过一种有损压缩将很多图片压缩在一起。目前比较流行的视频编码标准为 H.264、H.265、AV1。

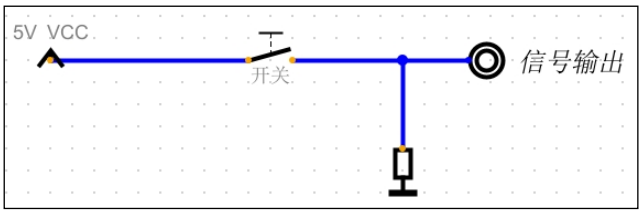


1.3 数字电路

将信息数字化后，我们可以方便的用二进制表示信息。而现代计算机技术的基础——数字电路，研究的就是如何用电表示二进制。其实这件事非常简单，例如下面的一个电路

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

图：



当开关闭合，信号输出就有 5v 电压；当开关断开，由于下拉电阻存在，信号输出电压变为 0，我们可以将信号输出的高电平态视为 1，低电平态视为 0，即可用电表示二进制。所以数字电路就是对电路进行定性分析（高或者低），而不用去考虑精确电压是否会有误差。

1.3.1 数字电路仿真软件——Digital

为了学习数字电路，给大家介绍一款数字电路拟真软件。

官方网站：https://github.com/hneemann/Digital

最新版下载地址：https://github.com/hneemann/Digital/releases/download/v0.30/Digital.zip

该软件运行需要 Java 环境，下载地址为：https://download.oracle.com/java/17/latest/jdk-17_windows-x64_bin.msi

安装完成后打开 digital，看到如下页面即安装成功：

1.3.2 基础逻辑门电路

基础逻辑电路重点掌握表述该逻辑的四种方式：逻辑表达式、真值表、逻辑符号、口诀。

1）非门

（1）从“组件——输入输出”中，选择一个输入和一个输出；

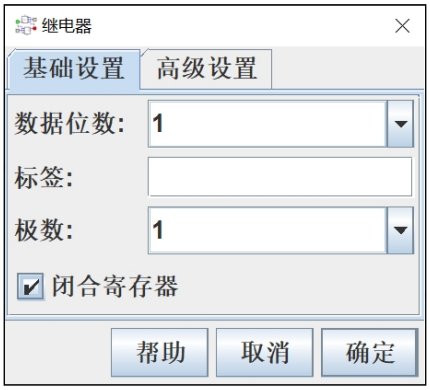
（2）从“组件——导线”中选择一个地、一个电源和一个下拉电阻；

（3）从“组件——开关”中选择一个继电器；

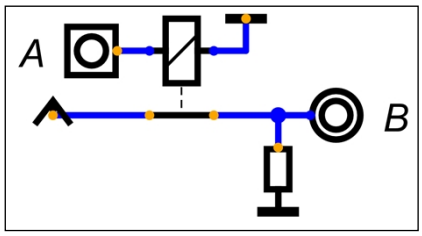
（4）右键点击输入，标签输入 A；

（5）右键点击输出，标签输入 B；

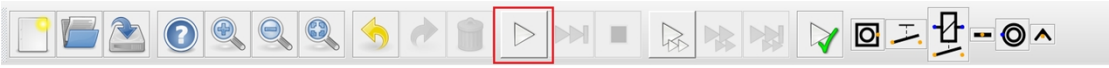
（6）右键点击继电器，勾选“闭合继电器”；



按照如下电路图连接：



连接完成后，选择快捷菜单栏的仿真键开始仿真。



开始仿真后，尝试改变 A 的输入电平，观察 B 输出的电平变化。可以看到 AB 的信号关系如下：

A	B
0	1
1	0

这种输入信号和输出信号完全相反的电路，叫做非门，上面的这张 0-1 关系表，我们称之为真值表。Digital 软件也可以根据电路自动生成真值表：

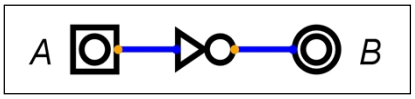
选择“分析——分析”可以得到下图。

A	B
0	1
1	0

$B = \overline{A}$

可以看到 Digital 软件不但给出了 AB 的真值表，还给出了 B 和 A 的逻辑关系。

在今后我们使用非门时，不会再画该电路，而是直接采用“组件——逻辑”里面的非门，如图示：非门逻辑符号。



口诀：输入与输出相反。

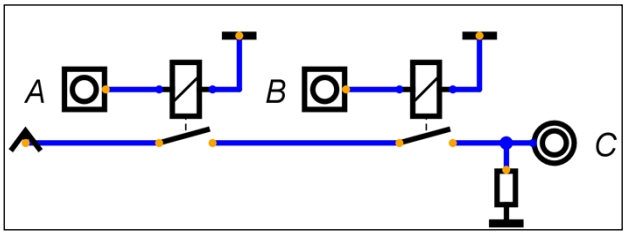
2）与门

（1）从“组件——输入输出”中，选择两个输入和一个输出；

（2）从“组件——导线”中选择两个地、一个电源和一个下拉电阻；

（3）从“组件——开关”中选择两个继电器；

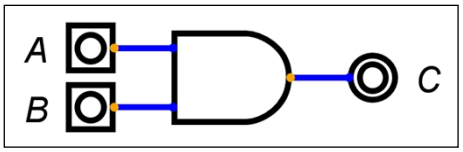
给输入编号 A、B，给输出编号 C，按照如下图连接：



开启仿真，观察 C 与 AB 之间的关系，可得真值表如下图所示：

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1
C = A ∧ B		

在今后我们使用与门时，不会再绘制该电路，而是直接采用“组件——逻辑”里面的与门，如图示：与门逻辑符号。



与门口诀：有 0 出 0，全 1 出 1。

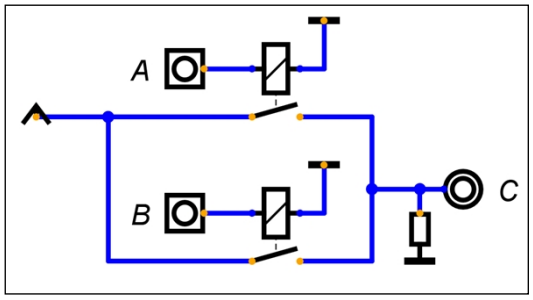
3）或门

（1）从“组件——输入输出”中，选择两个个输入和一个输出；

（2）从“组件——导线”中选择两个地、一个电源和一个下拉电阻；

（3）从“组件——开关”中选择两个继电器；

给输入编号 A、B，给输出编号 C，按照如下图连接：

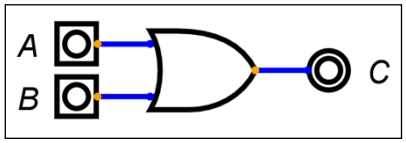


开启仿真，观察 C 与 AB 之间的关系，可得真值表如下图所示：

A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

$C = A \vee B$

在今后我们使用或门时，不会再绘制该电路，而是直接采用“组件——逻辑”里面的或门，如图示：或门逻辑符号



口诀：有 1 出 1，全 0 出 0。

1.3.3 其他门电路

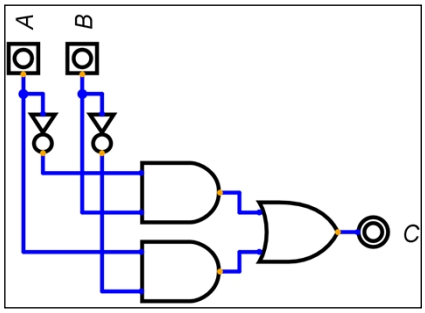
与或非三门我们一般称之为基础门电路，而其他门电路可以用这三种门电路组合得到。下面我们给出一些门电路真值表以及用与或非三门的组合实现。（这些实现并非最简实现，我们的这门数字电路基础，更多只考虑可行性，不考虑优化）

1）异或门

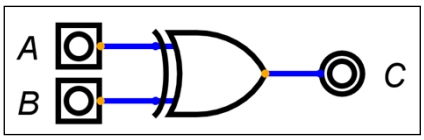
真值表和电路图：

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

$C = (\overline{A} \wedge B) \vee (A \wedge \overline{B})$



简化符号：



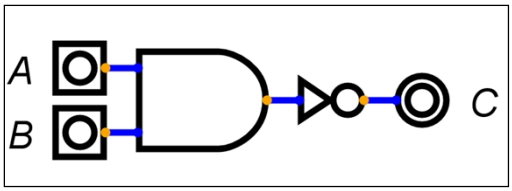
口诀：奇数出 1，偶数出 0。

2）与非门

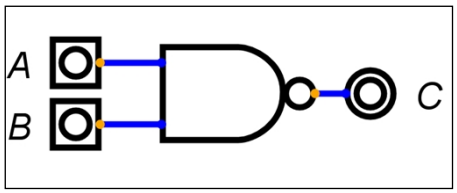
真值表和电路图：

A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

$C = \overline{A} \vee \overline{B}$



简化符号：



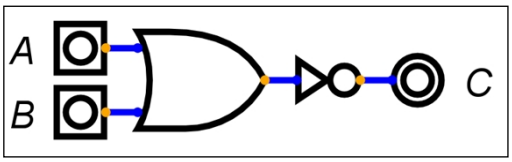
口诀：全 1 出 0，其他出 1。

3）或非门

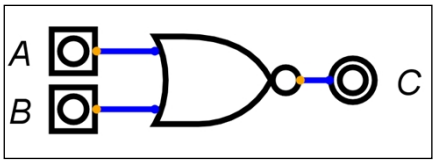
真值表和电路图：

A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

$C = \overline{A} \wedge \overline{B}$



简化符号：



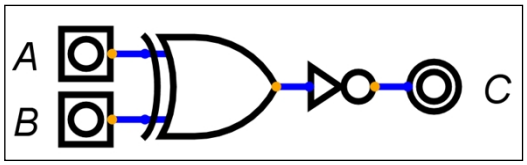
口诀：有 1 出 0，全 0 出 1

4）异或非门

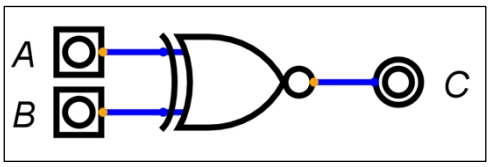
真值表和电路图：

A	B	C
0	0	1
0	1	0
1	0	0
1	1	1

$C = (\overline{A} \wedge \overline{B}) \vee (A \wedge B)$

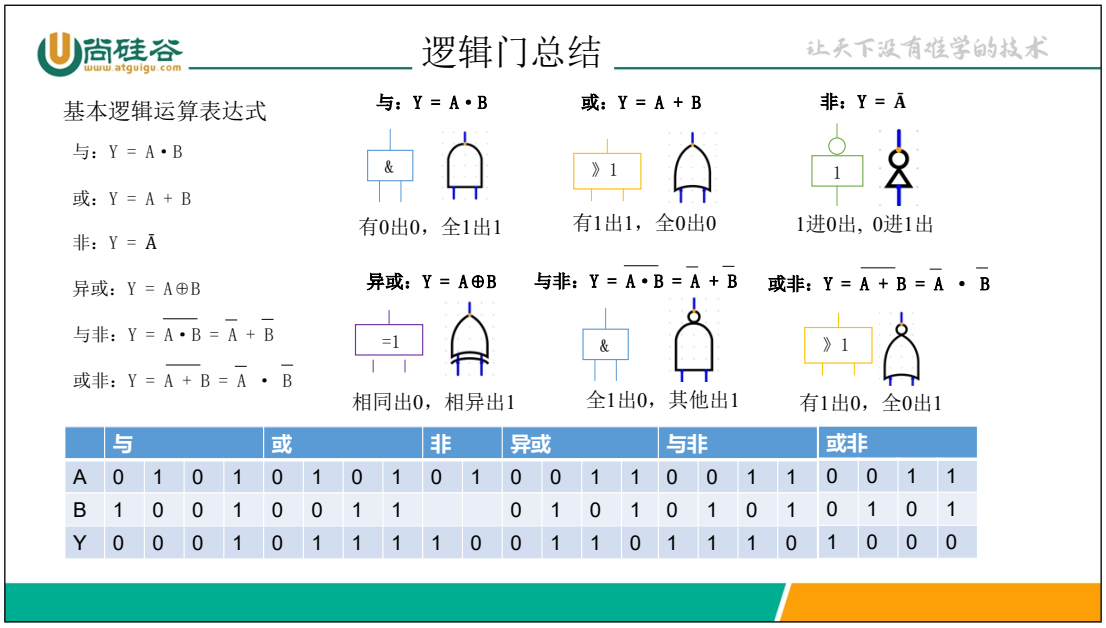


简化符号：



口诀：相同出 1，相异出 0。

5）逻辑门电路总结



	与				或				非		异或				与非				或非			
A	0	1	0	1	0	1	0	1	0	1	0	0	1	1	0	0	1	1	0	0	1	1
B	1	0	0	1	0	0	1	1			0	1	0	1	0	1	0	1	0	1	0	1
Y	0	0	0	1	0	1	1	1	1	0	0	1	1	0	1	1	1	0	1	0	0	0

1.3.4 运算器

1）半加器

现在我们来考虑如何用电路来实现 1 位加法。假如有两个 1 位二进制数 A、B，它们的和为 1 位二进制数 S，那么存在如下枚举：

（1）如果 A=0，B=0，那么 S=0；

（2）如果 A=1，B=0，那么 S=1；

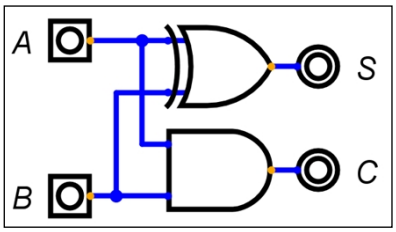
（3）如果 A=0，B=1，那么 S=1；

（4）如果 A=1，B=1，那么 S=0，而且产生一个进位 C=1。

也就是说，对于加法运算，应该两个输入 A 和 B，两个输出 S 和 C，S 表示和，C 表示是否产生进位，也就是说真值表如下图所示：

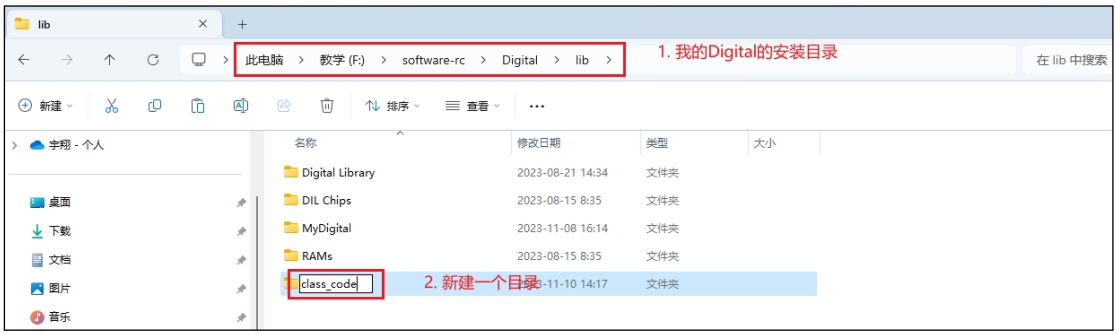
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

其中 A、B 与 S 的关系为异或，与 C 的关系为与，得到如下电路图：

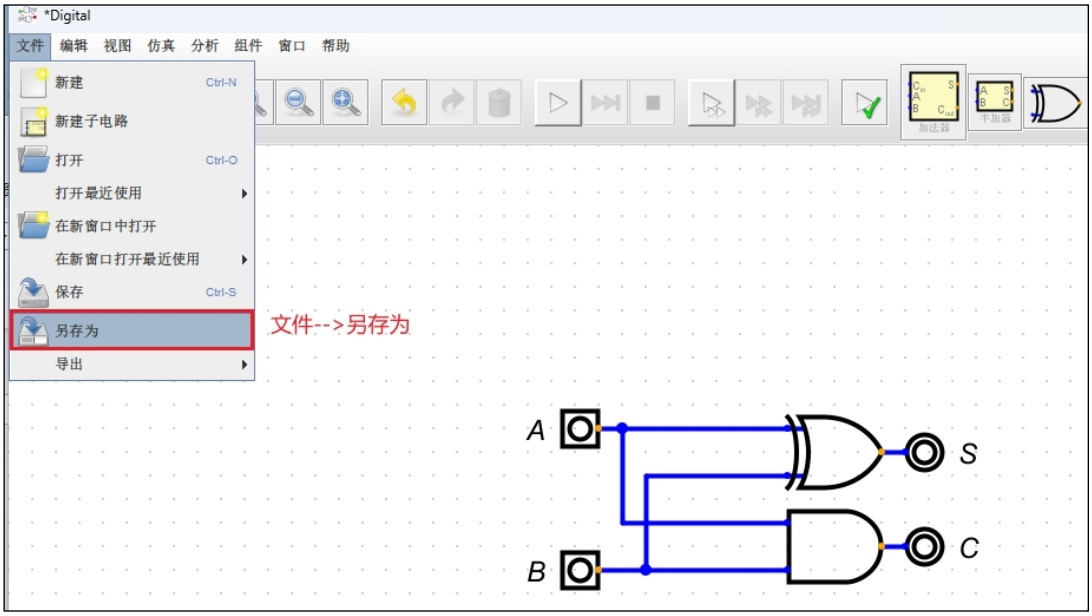


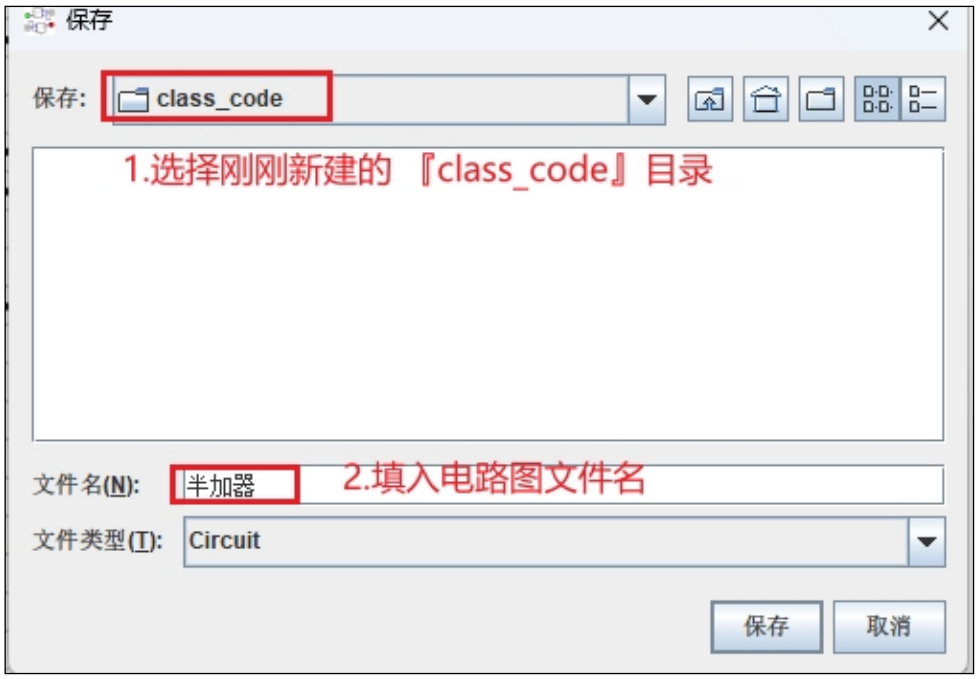
2）将半加器封装成一个组件

（1）将电路另存到 Digital 的安装目录的 lib 中

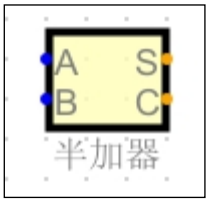


（2）保存半加器电路到 class_code 目录





（3）使用封装好的电路



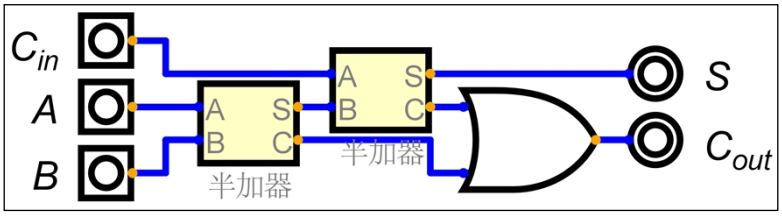
3）加法器

半加器实现一位全加器。

半加器只能处理两个数的运算，但在实际计算中，如果我们做加法的两个位不是最低

的位，那就要考虑低位可能产生的进位。也就是说，一个功能完整的 1 位加法器，应该考虑三个输入，除了 A、B 两个加数，还要考虑是否存在低位进位 Cin。

为了处理三个数相加，我们可以用两个半加器分别处理两个累加过程，同时两个累加器产生的进位 C1 和 C2，不可能同时为 1，（因为如果 C1 位 1，那么 S1=0，S1=0，C2 不可能为 1），且二者只要有一个为 1，则输出进位为 1，所以关系为或，最终呈现的电路如下：



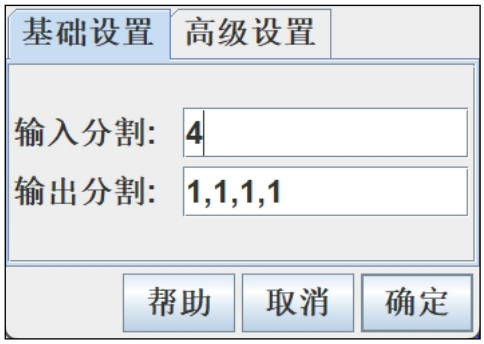
其中的黄色方块为子电路，内容就是我们上面实现的半加器。

4）4 位加法器

多位加法器就是多个加法器的串联，下面我们以 4 位加法器为例来展示多位加法器电路：

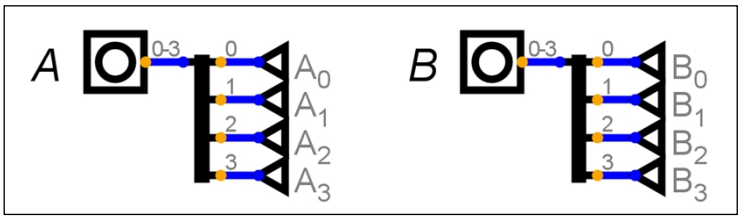
添加输入 Cin，A，B，右键点击 A 和 B，将位数调整为 4；

添加两个“组件——导线”中的分裂器，右键点击将其设置如下图所示。



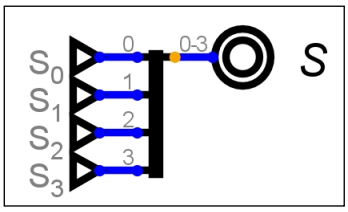
将分裂器输入端与 A、B 相连，并添加 8 个“组件——导线”里面的隧道，分别命名为 A0、A1、A2、A2、B0、B1、B2、B3，并接在分裂器的输出端，如图示：

说明：图中 A0 管道是给标签名起名位 A_0 来实现的。

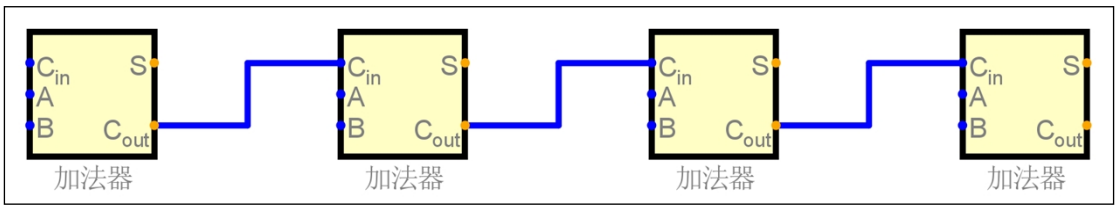


添加输出 S，并将位数调整为 4，并添加分裂器，输入/输出设置为 1，1，1，1/4，并将

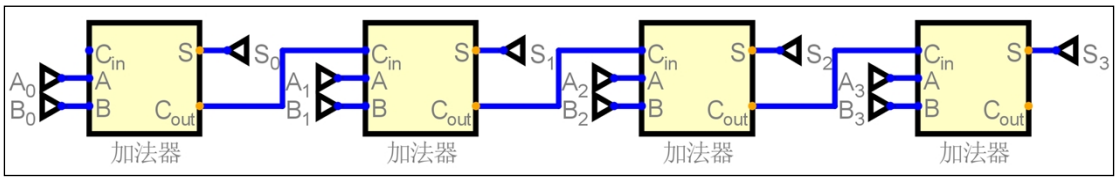
输出端连接到 S，输出端连接四个隧道 S0、S1、S2、S3，如图示：



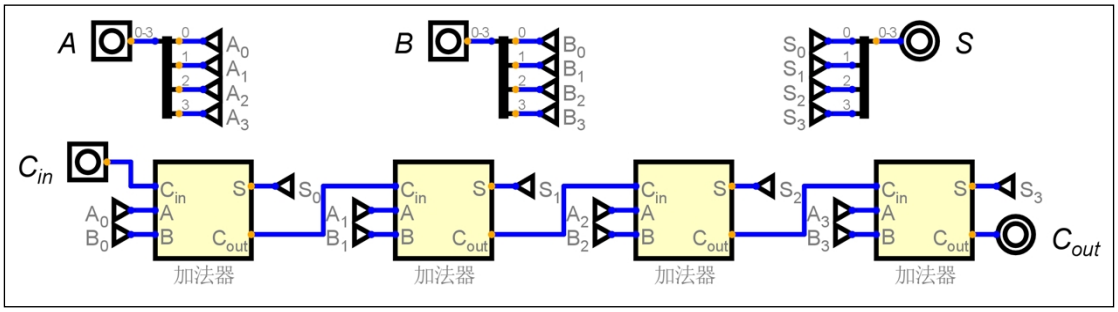
添加四个加法器子电路，并将进位由低到高串联，如图所示：



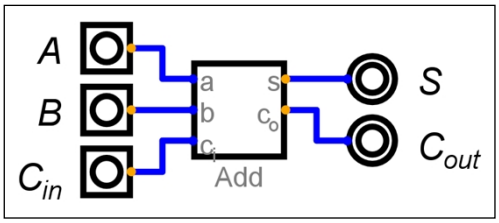
添加 12 条隧道，分别与 A、B、S 的隧道对应，连接到加法器的输入端，如图示：



最后添加输入 Cin 和输出 Cout，最终效果如图：



今后我们在用到多位加法器时，不会再次绘制该电路，而是会直接使用“组件——运算器”中的加法器，如图示：



1.3.5 锁存器和触发器

目前我们画了很多计算电路，这些电路都有一个特点，当输入信号发生变化，输出信号会立刻变化，无法对状态进行存储。这在某些情况下会给我们的计算带来麻烦，例如我

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

们想计算这个算式：25+37+12-9+8。在没有状态存储的情况下，我们只能用 4 个计算电路串联得到最终结果，这显然是不现实的。

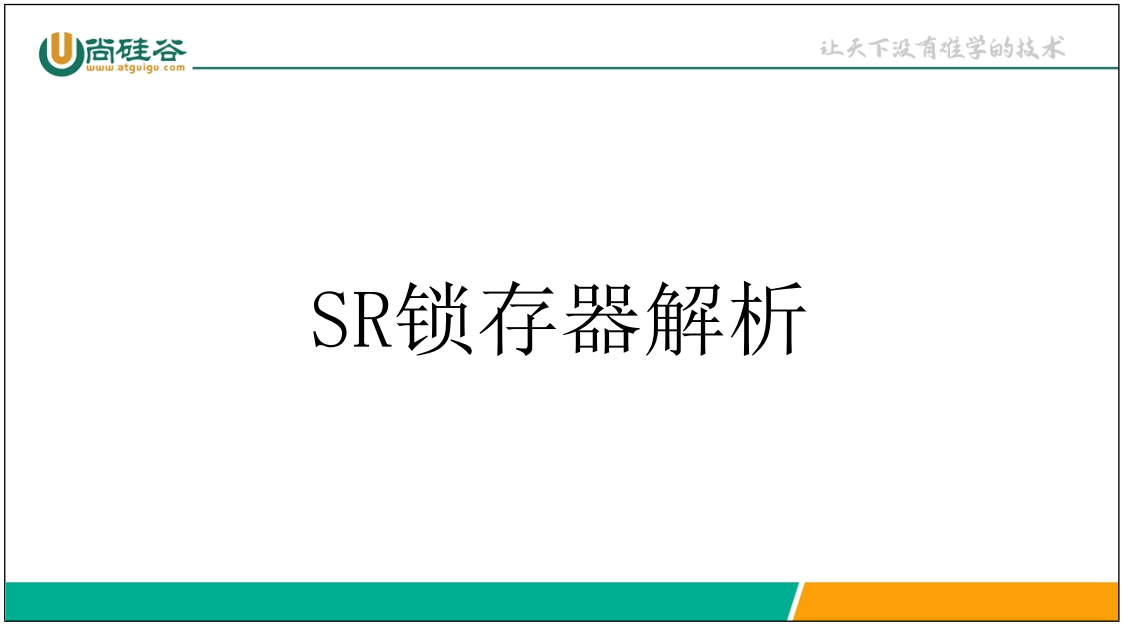
接下来的一系列电路，我们要给大家描述的是，如何储存电信号。

1.3.5.1 SR 锁存器

1）SR 锁存器的电路结构：

SR 锁存器（set-reset-Latch）是静态存储单元中最基本、也是电路结构中最简单的一种电路。SR 锁存器可以有两种构成方式。方式一，由两个或非门构成。方式二，由两个与非门构成。两种方式构成的 SR 锁存器功能相同。此处，我们以与非门构成的 SR 锁存器为例进行介绍。

2）由两个与非门组成的 SR 锁存器的工作原理



总结：

（1）当 S`为 1，R`为 1， Q 和 Q`状态不变；

（2）当 S`为 0，R`为 1， Q=1、Q`=0；(置位)

（3）当 S`为 1，R`为 0， Q=0、Q`=1；(复位)

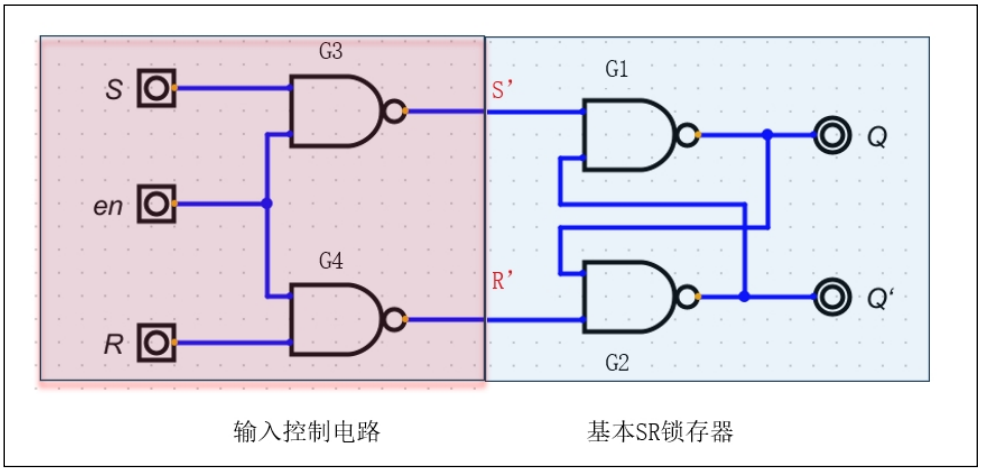
（4）当 S`为 0，R`为 0，电路无意义

S`输入	**R`输入**	**Q 输出**	**Q`输出**
1	1	维持不变	维持不变
0	1	1	0
1	0	0	1
0	0	1(无意义)	1(无意义)

1.3.5.2 带 en 输入的 D 锁存器

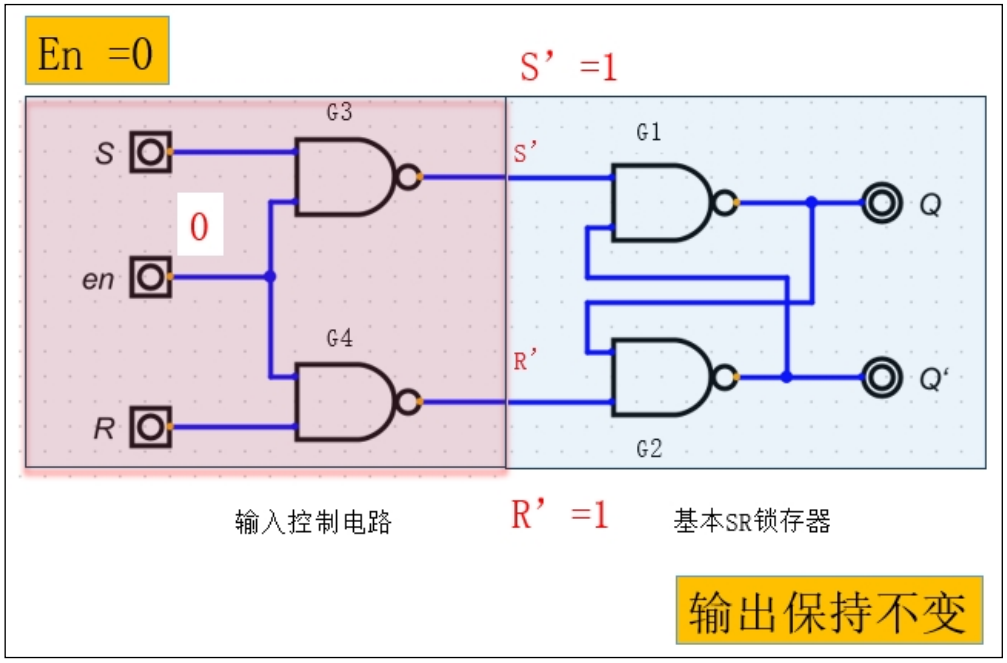
上面的 SR 锁存器尽管可以锁住输出状态，但是我们没法控制设置输出的时机。所以我们给这个电路加上一些其他组件。

1）电路结构：锁存器的基础上，增加一个触发信号输入端。



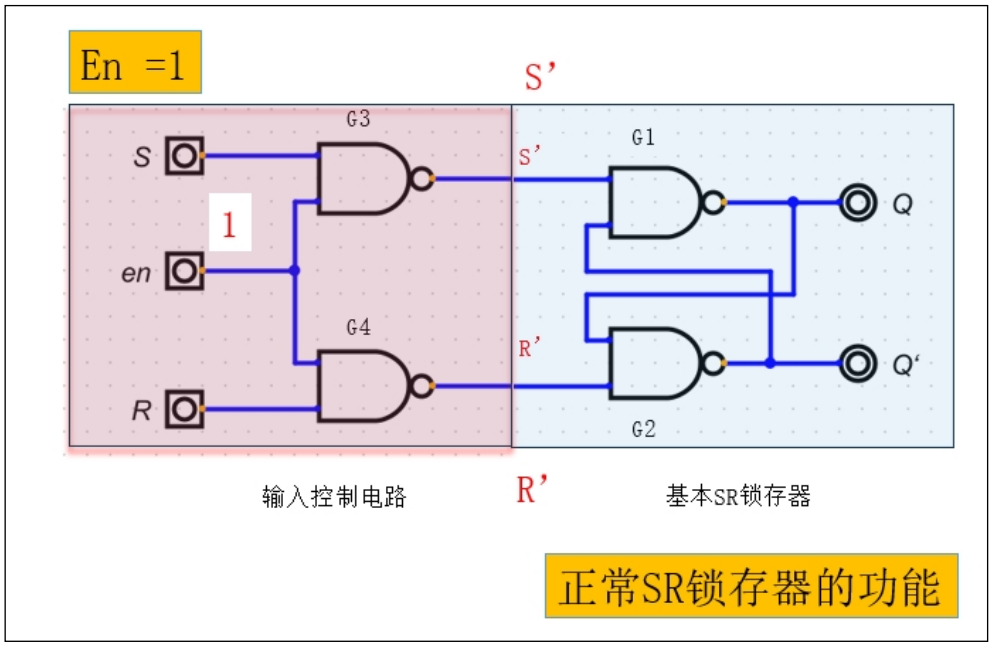
2）工作原理

（1）En = 0 时，输出保持不变



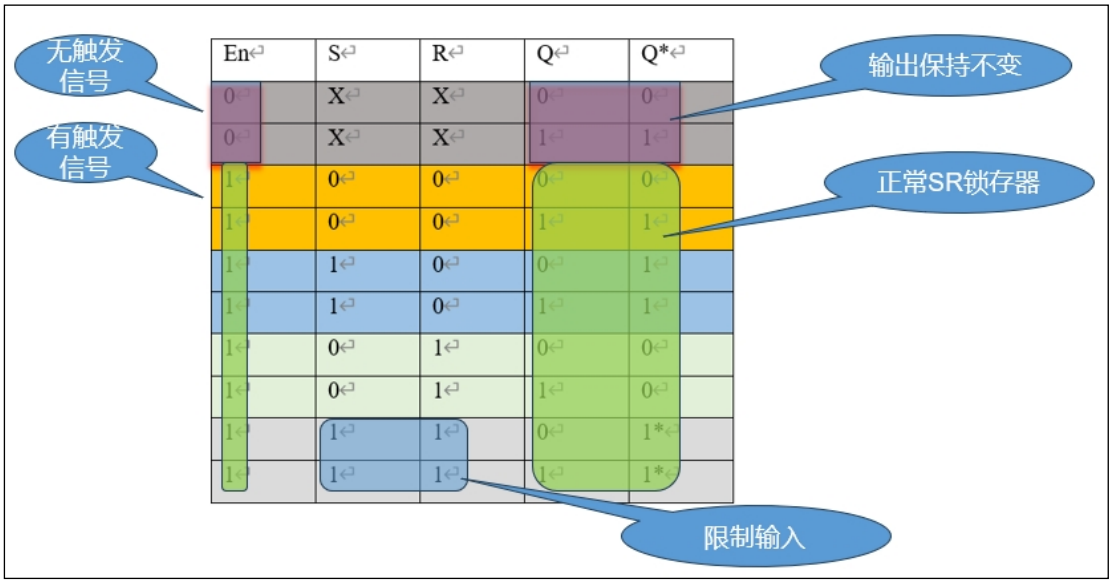
En=0，与非门特性，G3 输出 S1=1，G4 输出 R’=1，因为 S’、R’低电平有效，此处都为 1，处于高电平，既不置 0 也不置 1，输出保持不变。

（2）En=1，正常 SR 锁存器的功能



当 en=1 时，与非门 G3 输出就是 S'，与非门 G4 的输出就是 R'，正好是之前讲过的 SR 锁存器的输入。

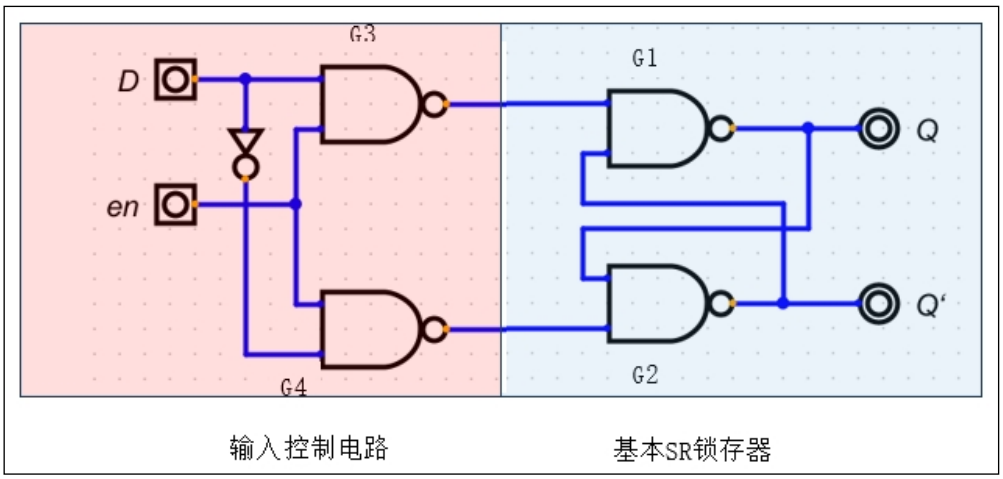
3）真值特性表



En	S	R	Q	Q*
0	X	X	0	0
0	X	X	1	1
1	0	0	0	0
1	0	0	1	1
1	1	0	0	1
1	1	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	1	0	1*
1	1	1	1	1*

4）带 En 输入的 D 锁存器

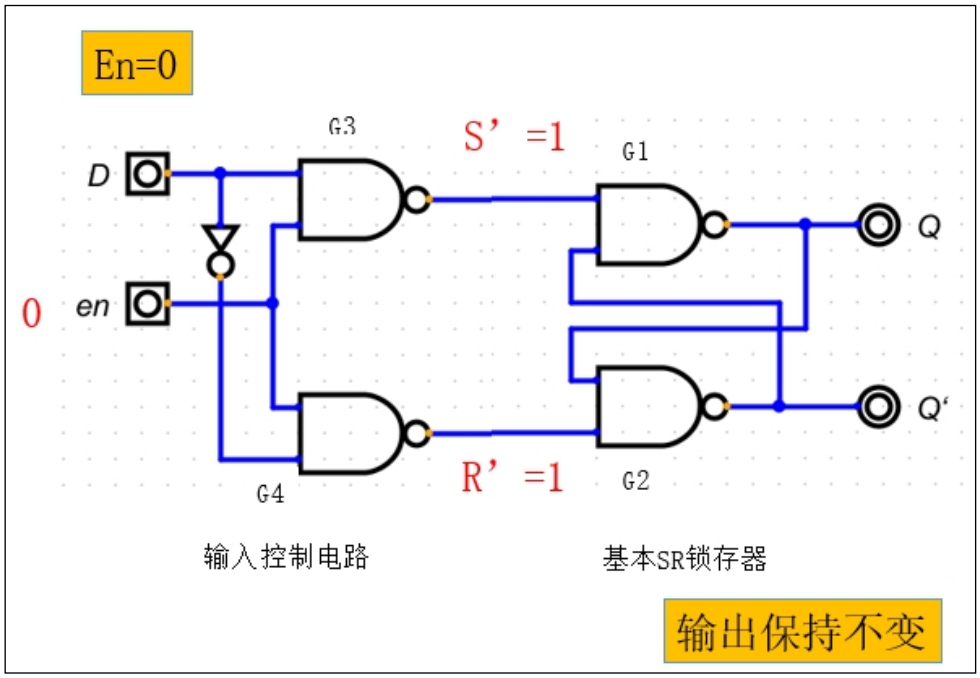
当 en 为 1 时，如果我们想将 Q 设为 1，此时 S 应为 1，R 应为 0；反之，如果我们想将 Q 设置为 0，S 应为 0，R 应为 1。由此可得到，SR 恰好为反相输入时，可以顺利的设置 Q，所以我们对电路稍加改造，将 S 改名为 D，并将其反相输入到 R，如图示：



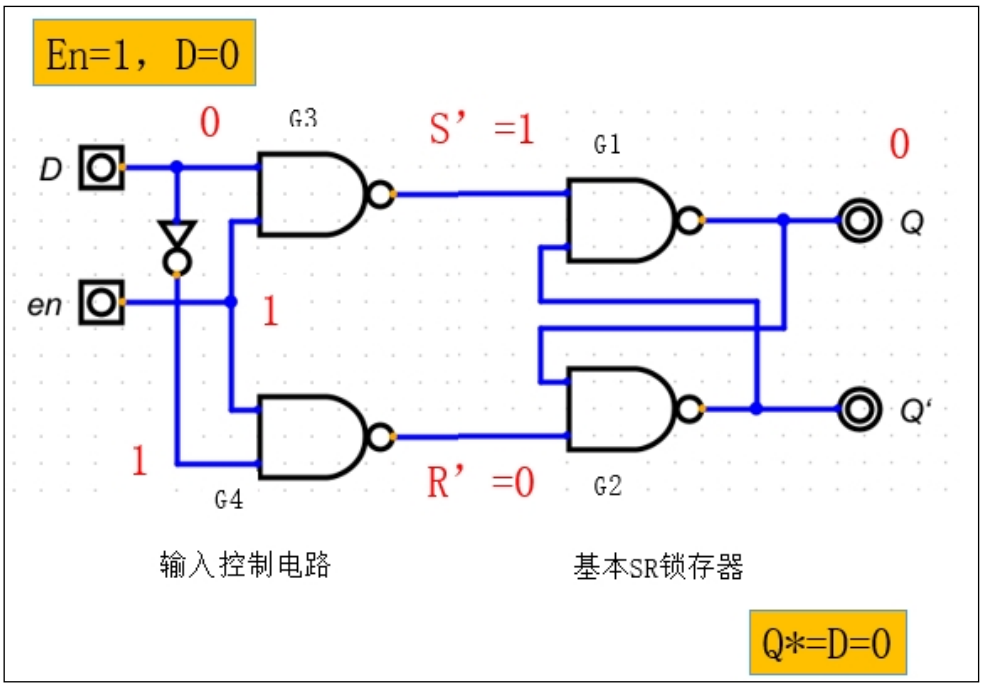
我们就得到了一个带 en 输入的 D 锁存器，其特性为：当 en 为高电平，Q 和 D 的输入保持一致；当 en 为低电平，Q 保持之前状态不变，从而起到存储作用。

（1）工作原理：

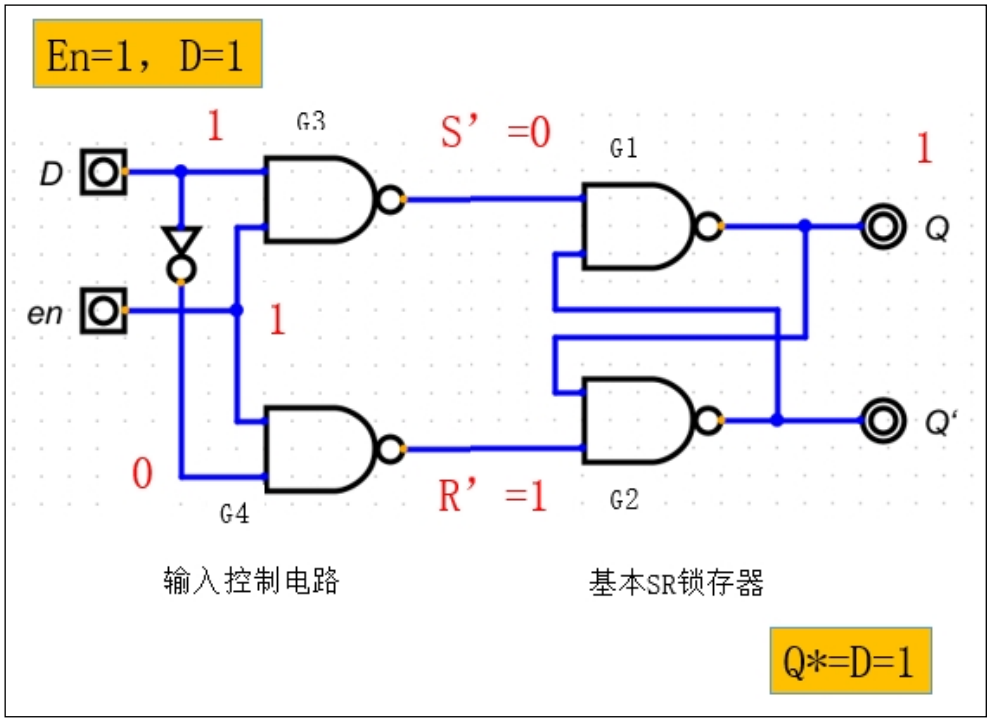
➢ En=0



➢ En=1，D=0



➢ En=1，D=1

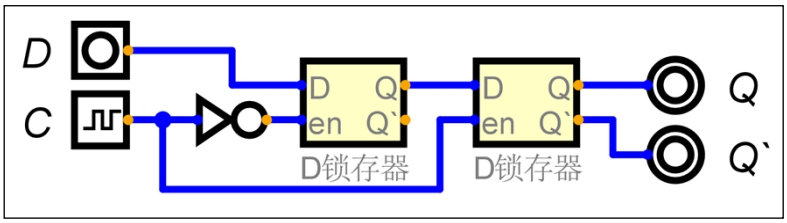


总结：当 en 为 1 时，输出 Q*的值与 D 输入的值相同。

问题：当 en 为 1 时，D 的输入直接影响 Q*的输出，为了提高触发器的可靠性，增强抗干扰能力，希望触发器的次态仅仅取决于 en 的下降沿（或上升沿）到来时的输入信号状态。D 触发器可以解决以上问题。

1.3.5.3 边沿触发的 D 触发器

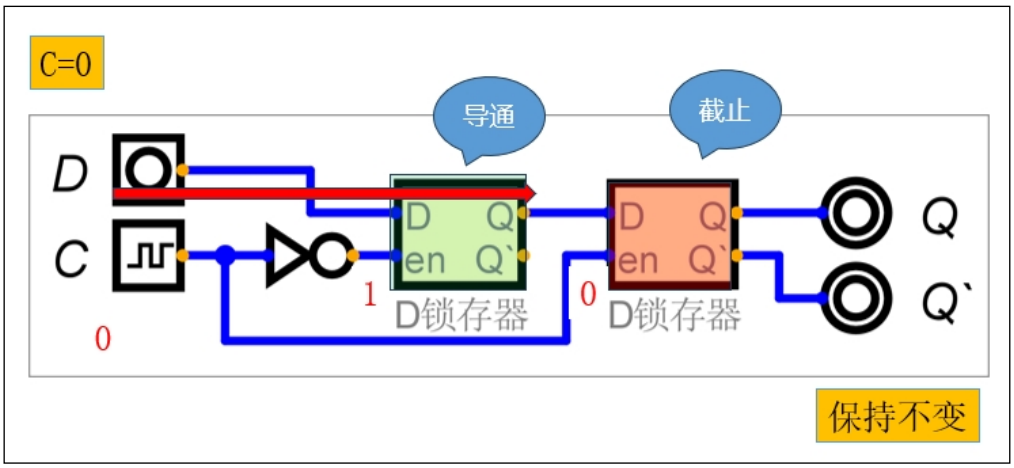
D 锁存器尽管可以起到保存数据的作用，但是当 en 信号为 1 时，D 输入和 Q 输出相当于是联通的，此时如果 D 信号有波动，Q 会跟随波动。我们希望能得到更稳定的输出 Q，不希望 en 高电平时 Q 随 D 波动，而是希望 Q 只在 en 信号由 0 变成 1 的一瞬间随 D 输入变化，其他时间都保持不变。看下面的电路图



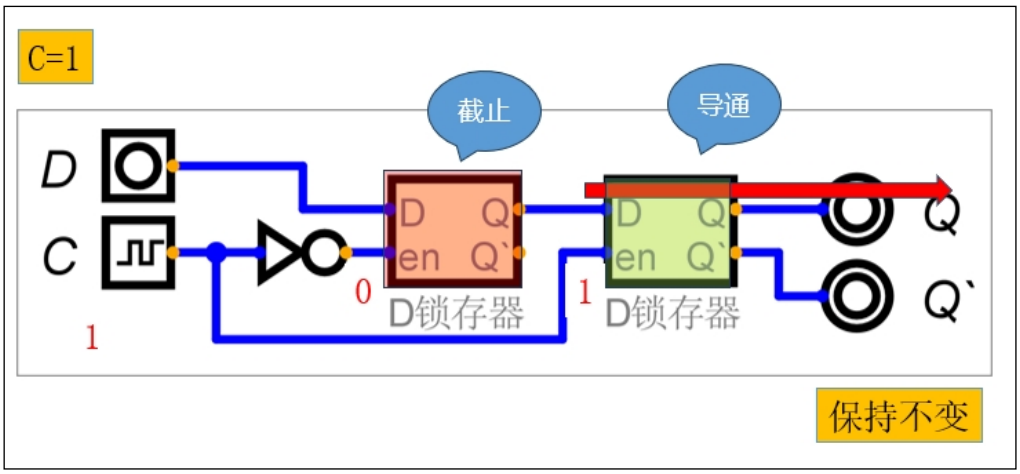
其中输入 C 为时钟输入，添加路径为“组件——输入输出”里面的时钟输入。关于时钟的作用，我们在之后计算机组成原理里面再为大家详细介绍，现在我们可以仅仅把时钟输入当作一个普通输入来用。

1）工作原理

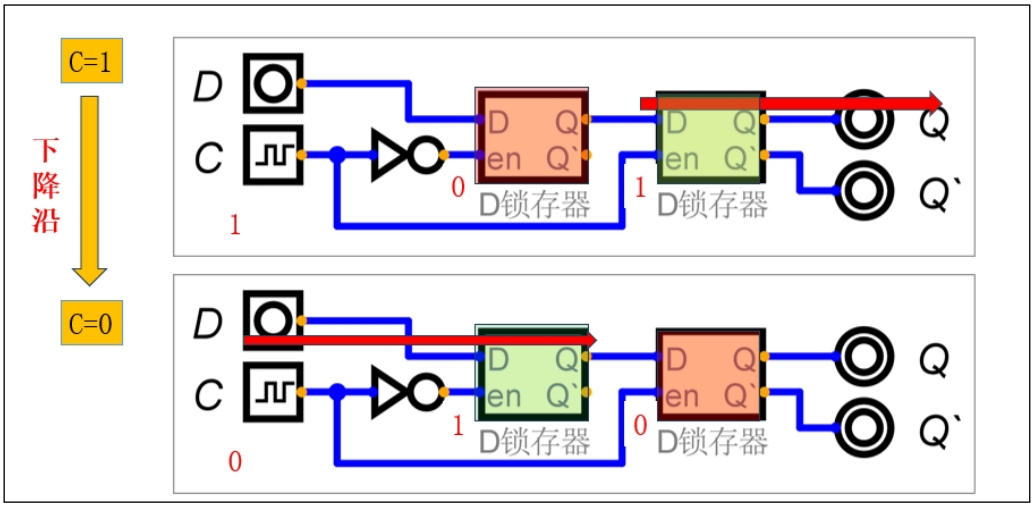
（1）C=0



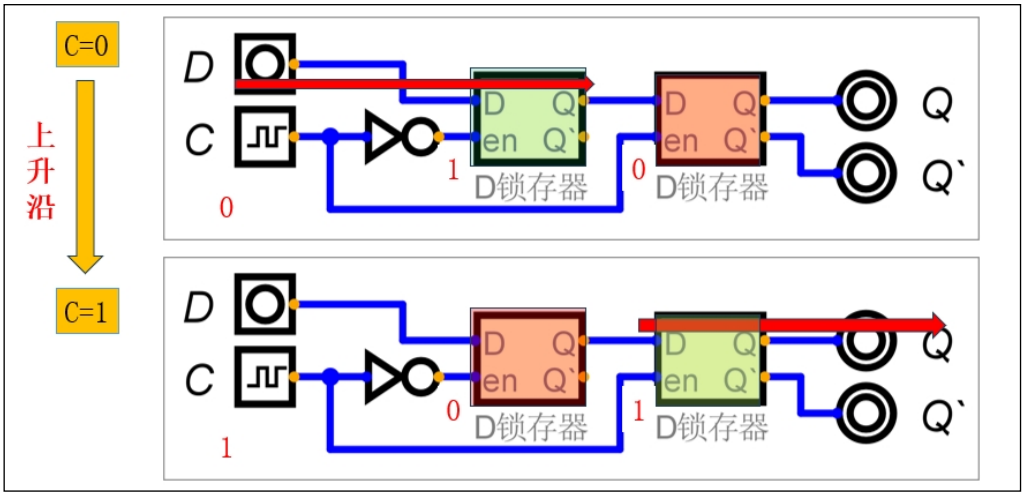
（2）C=1



（3）C 从 1→0



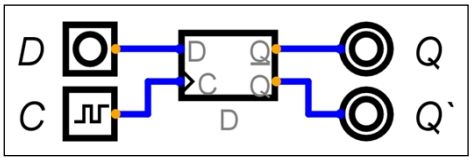
（4）C 从 0→1



开启仿真后，经过实验我们得知，C 信号不论是高电平还是低电平，Q 都不会随 D 变化；只有当 C 信号由 0 变 1 的一瞬间，D 的值能够传递到 Q。这个电路我们称为上升沿触发

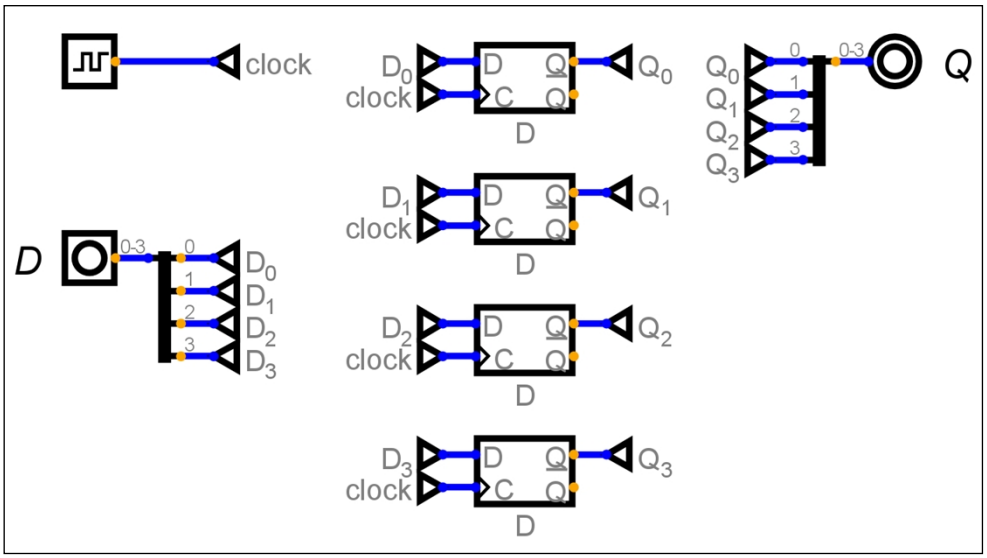
的 D 触发器。其实对电路稍作改变，我们还能得到下降沿触发的 D 触发器，但之后我们如果不做特殊说明，凡是用到 D 触发器，都是指上升沿触发器。

今后我们如果使用 D 触发器，不会再绘制该电路，而是直接使用“组件——触发器”中的 D 触发器。有一点需要注意，Digital 软件中的 D 触发器，Q 输入和标准触 D 发器是反相的，所以我们将 Q`当作 Q 来用，如图示：

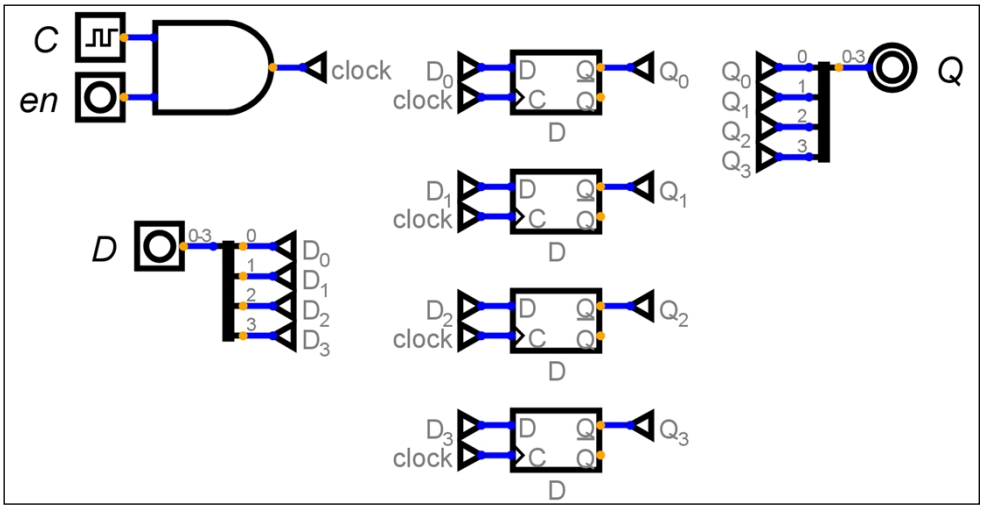


1.3.6 寄存器

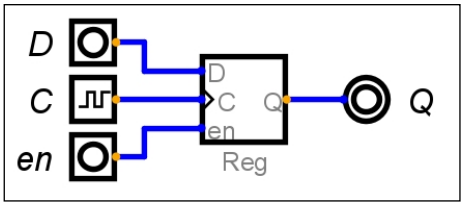
D 触发器可以在时钟上沿存储 1bit 数据，如果我们想存储多个 bit 的数据，就需要用多个 D 触发器并联实现，这种电路我们称之为寄存器。以 4bit 寄存器为例，看如下电路：



4bit 输入 D 会在时钟输入上升沿存储到 Q。再给这个寄存器加上 en 信号，最终效果如图：



今后我们只用寄存器时，不会在绘制该电路，而是直接使用“组件——存储器”中的寄存器，如图示：



第 2 章 计算机组成原理

2.1 冯诺依曼模型的计算机

由科学家冯诺依曼提出的模型理论。基于通用图灵机建造的计算机都是在存储器（内存/寄存器）上存储数据。鉴于程序和数据在逻辑上是相同的，因此程序也能存储在计算机的存储器中。

1）冯诺依曼模型的四个子系统

（1）存储器：用来存储数据和程序的区域

（2）算数逻辑单元（ALU）：用来进行计算（算数运算、逻辑运算、位运算等）的地方。

（3）控制单元：对存储器、算数逻辑单元、输入/输出等子系统进行控制操作。

（4）输入/输出单元：输入子系统负责从计算机外部接收输入数据，输出子系统负责将计算机处理结果输出到计算机外部。

2）冯诺依曼模型-存储程序概念

冯诺依曼模型要求程序也必须存储在存储器(内存)中，现代计算机的存储单元用来存储

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

程序和数据，这意味着程序和数据应该有相同的格式，实际上他们都是以位模式（0 和 1）存储在内存中。

3）冯诺依曼模型-指令的顺序执行

冯诺依曼模型中的一段程序是由一组数量有限的指令组成。

控制单元从内存中提取一条指令，解释指令，接着执行指令，也就是说指令是一条接着一条顺序执行的。

2.2 计算机组成部件

计算机组成部件可以分为三大类。

（1）中央处理单元（CPU）

（2）主存储器（内存）

（3）输入/输出子系统

2.2.1 中央处理单元 CPU

CPU 用于数据的运算【算数逻辑单元（ALU）、控制单元、寄存器组】。

1）CPU 中的算数逻辑单元

算数逻辑单元：对数据进行逻辑、移位和算数运算。

（1）算术运算：整数和浮点数的加减运算。

（2）位移运算：逻辑移位运算和算术移位运算。

（3）逻辑运算：非、与、或、异或，这些运算的输入数据为二进制模式，运算结果也是二进制模式。

2）CPU 中的寄存器

寄存器：用来存放临时数据的高速独立存储单元。

（1）数据存储寄存器：保存运算的中间结果。

（2）指令存储器（IR）：CPU 从内存中逐条取出指令，并存储在指令存储器中，解释并执行指令。

（3）程序计数器（PC）：保存当前正在执行的指令地址，当前指令执行完成后，计数器自动加 1，指向下一条指令的内存地址。

3）CPU 控制单元：

控制单元：控制各个子系统的操作，控制是通过从控制单元到其他子系统的信号来进行的。

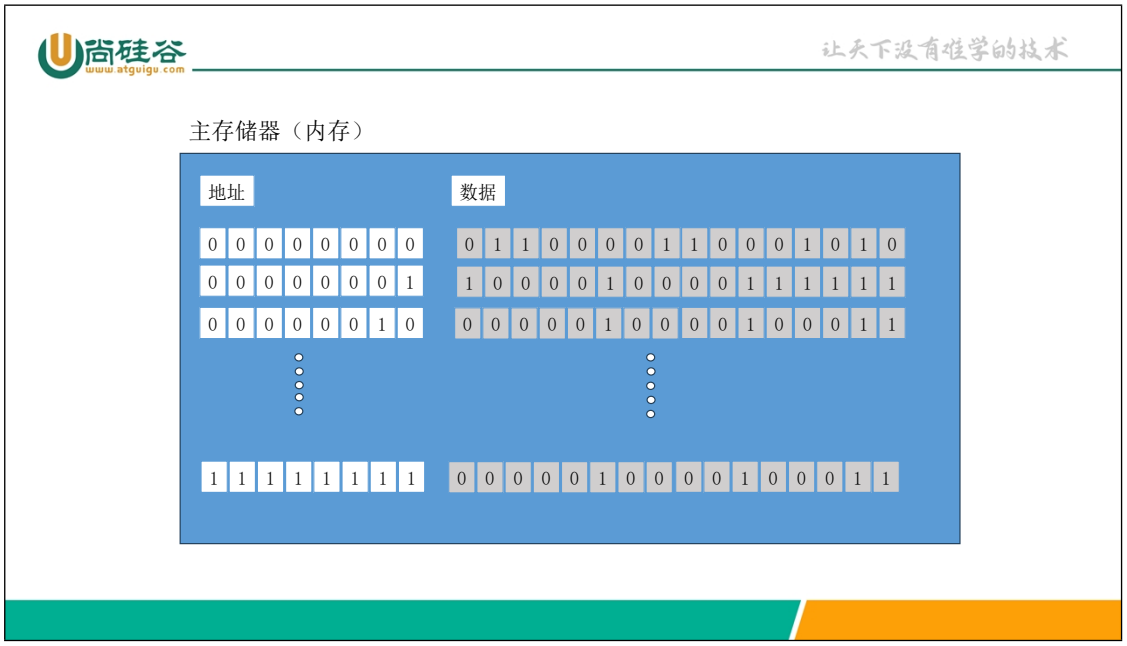
2.2.2 内存

内存是存储单元的集合，每个存储单元都有唯一的标识，称为地址。

数据以“字”的形式在内存中传入传出，字可以是 8 位、16 位、32 位、64 位。如果字

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

是 8 位，一般称为一个字节。



1）内存类型

主要有两种类型：RAM 和 ROM。

（1）随机存取存储器（RAM）

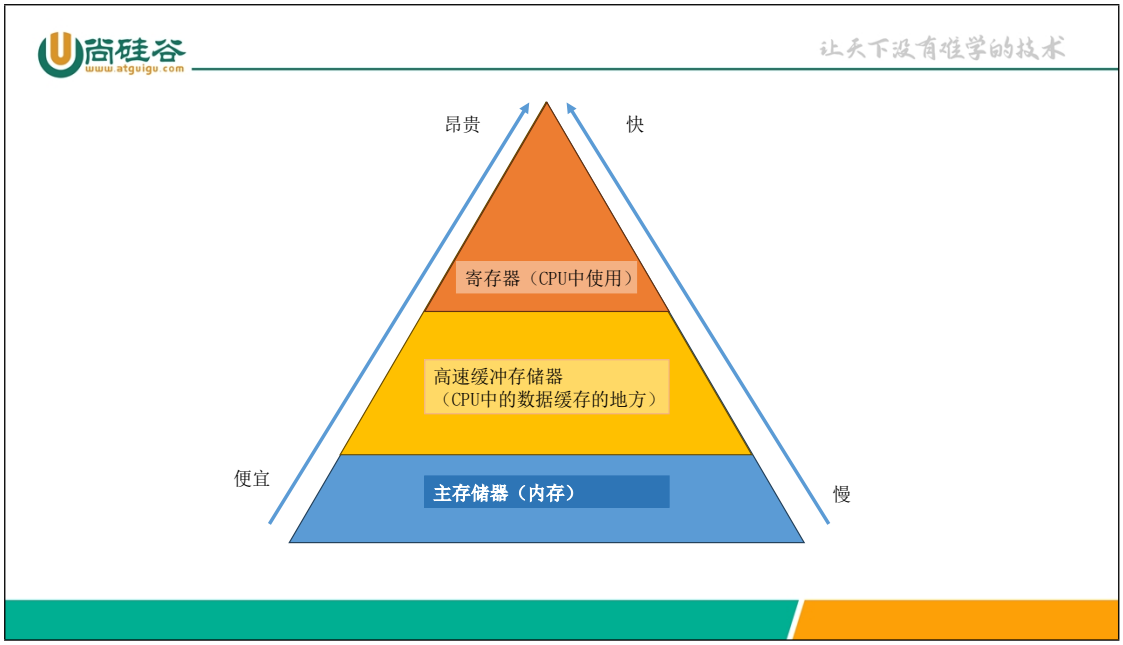
特点：系统断电后，信息（程序或数据）丢失。

（2）只读存储器（ROM）：里面的数据由制造商写进去，用户只能读不能写

特点：系统断电数据不会丢失。常用来存储那些在开机时运行的程序。

（例如：系统开机时的引导程序）。

2）存储器的层次



3）高速缓冲存储器

存储数据的速度比内存快，比寄存器慢。通常容量较小，被置于 CPU 和主存储器（内存之间）。

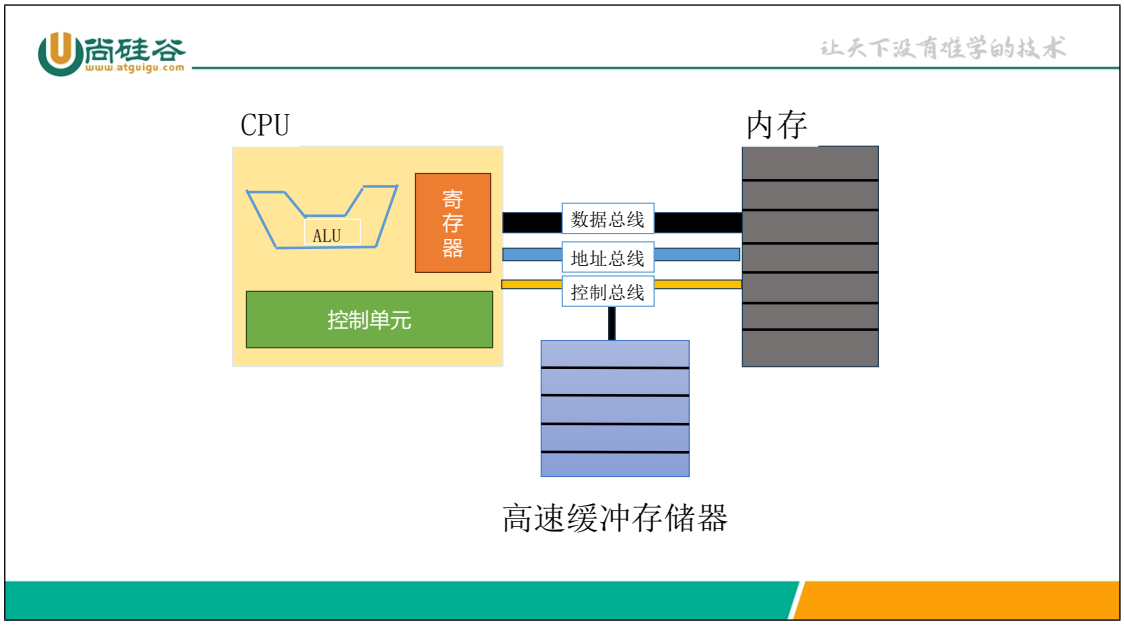
4）CPU 和存储器的连接

CPU 与主存储器之间通常由称为总线的三组线路进行连接。他们分别是：数据总线、地址总线、控制总线。

（1）数据总线：由多根线组成，每根线每次传送 1 个位的数据。线的数量取决于计算机字的大小。例如：计算机的字是 32 位（4 个字节），那么需要 32 根线的数据总线，以便同一时刻同时传送 32 位的数据。

（2）地址总线：允许访问存储器中的某个字的。地址总线的线数取决于存储空间的大小。例如：存储器的容量为 2 的 n 次方个字，那么地址总线一次需要传送 n 位的地址数据，因此需要 n 根线。

（3）控制总线：负责传送指令的。例如：如果计算机有 2 的 m 次方条控制命令，那么控制总线就需要有 m 根。



2.2.3 输入/输出（I/O）系统

可以使计算机与外界进行通信，并在断电情况下存储程序和数据，分为两大类：非存储设备和存储设备。

（1）非存储设备：键盘、鼠标、显示器、打印机等。

（2）存储设备：也称为辅助存储设备，通常有磁介质和光介质两种。

特点：便宜，断电后数据不丢失。

1）I / O 设备的连接

（1）输入/输出设备不能直接与 CPU 和内存的总线相连接，因为输入/输出设备本质与 CPU 和内存的本质不同，输入/输出设备都是磁性或光学设备，而 CPU 和内存是电子设备。与 CPU 和内存相比，输入/输出设备的数据读取速度要慢的多，因此必须要有一个中介来处理这种差异，输入/输出控制器。

（2）输入/输出控制器：连接输入/输出设备到总线上，每一个输入/输出设备都有一个特定的控制器。

（3）I/O 设备的连接控制器：控制器清除了输入/输出设备与 CPU 以及内存在本质上的障碍，控制器可以是串行或并行的设备。

- 串行控制器：只有一根数据线连接在设备上。
- 并行控制器：有多根数据线连接到设备上，一次能同时传送多个位。

（4）常用控制器：SCSI、火线、USB 和 HDMI。

2.2.4 程序的执行

通用计算机使用程序的一系列指令来处理数据，通过执行程序，将输入数据转换为输出数据。程序和数据都放在内存中。

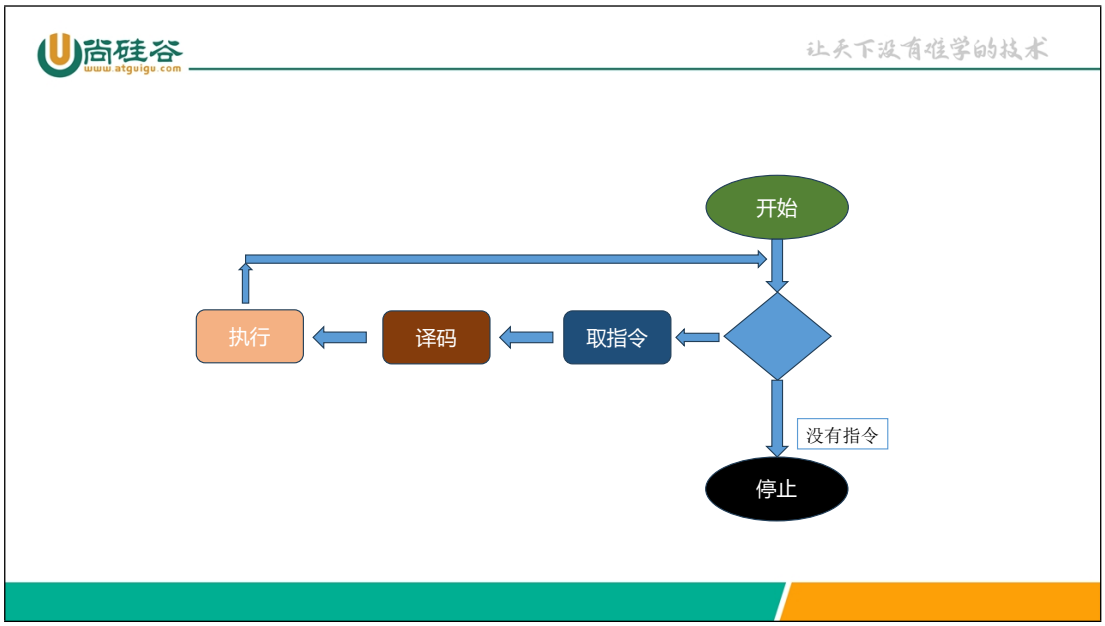
（1）机器周期。

（2）输入/输出操作。

1）程序执行：机器周期

CPU 利用重复的机器周期来执行程序中的指令，一步一条，从开始到结束。

一个周期包括 3 步：取指令→译码→执行。



2）取指令

CPU 的控制单元命令系统将下一条将要执行的指令复制到 CPU 的指令寄存器中，被复制的指令地址保存到程序计数器中，复制完成后，程序计数器自动加 1，指向内存中的下一条指令。

3）译码

当指令置于指令寄存器后，该指令将由控制单元负责译码，指令译码的结果是产生一系列系统可执行的二进制代码。

4）执行

指令译码完毕后，控制单元发送任务命令到 CPU 的某个部件，例如：控制单元告知系统，让它从内存中读取数据。这就是执行阶段。

2.2.5 不同的体系结构

1）CISC（复杂指令集计算机）体系结构

设计策略：是使用大量的指令，包括复杂指令。

优点：程序设计更容易，因为每个简单或复杂的任务都有一条对应的指令。程序员不需要写一大堆的指令去完成复杂的任务。

缺点：指令集的复杂性使得 CPU 和控制单元电路非常复杂。

优化方案：程序在两个层面上运行，CPU 不直接执行机器语言指令，CPU 只执行被称为微操作的简单操作，复杂指令被转化为一系列简单操作后由 CPU 执行，使用微操作的程序设计被称为微程序设计。

应用：英特尔公司开发的奔腾系列 CPU。

2）RISC （精简指令集计算机）体系结构

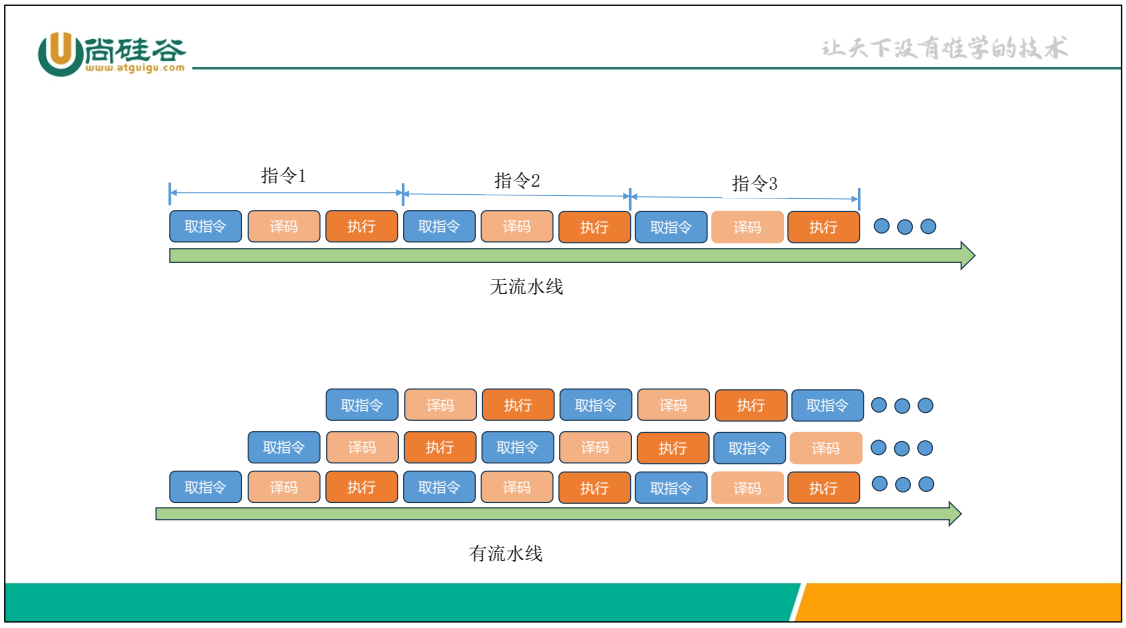
设计策略：是使用少量的指令完成最少的简单操作。

缺点：程序设计更难，复杂指令需要用简单指令模拟。

3）流水线

计算机对每条指令使用取指令、译码、执行三个阶段，早期计算机每条指令的这三个阶段需要串行完成，现代计算机使用流水线技术改善吞吐量（单位时间内完成的指令总数）。

如果控制单元能同时执行两个或三个阶段，那么下一条指令就可以在前一条指令完成前开始。



2.2.6 简单计算机

为了解释计算机的体系结构和指令处理，引入一台简单（非真实）计算机。简单计算机有三部分组成。

（1）CPU

（2）存储器

（3）输入/输出（I/O）系统

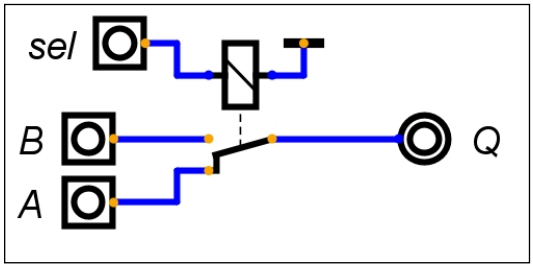
第 3 章 设计计算机硬件

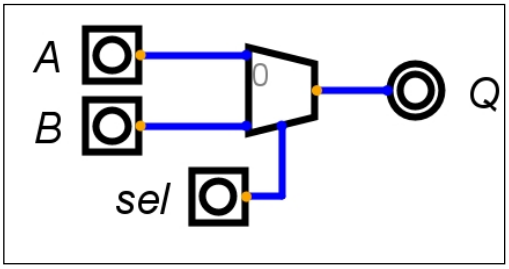
3.1 实现一个 ALU

ALU（算术逻辑单元）是计算机中的一种关键组件，负责执行算术运算和逻辑运算。它是中央处理器（CPU）内部的一个重要功能模块，用于处理数据和执行指令。简单的 ALU，可以看作是计算电路的大杂烩，例如，接下来我们打算实现一个 ALU，它能够实现 A、B 两个 16bit 数字之间的加法、减法、按位与和按位或运算，同时有一个 ZF（Zero Flag）指示 A、B 两数是否相等。

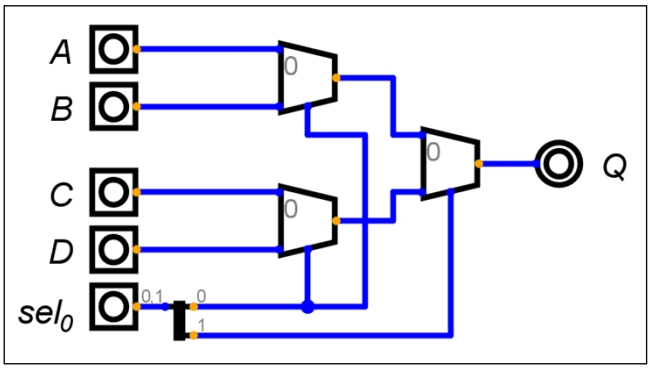
1）选择器

在 ALU 中我们会用到复用器，两相复用器原理和封装如下图：





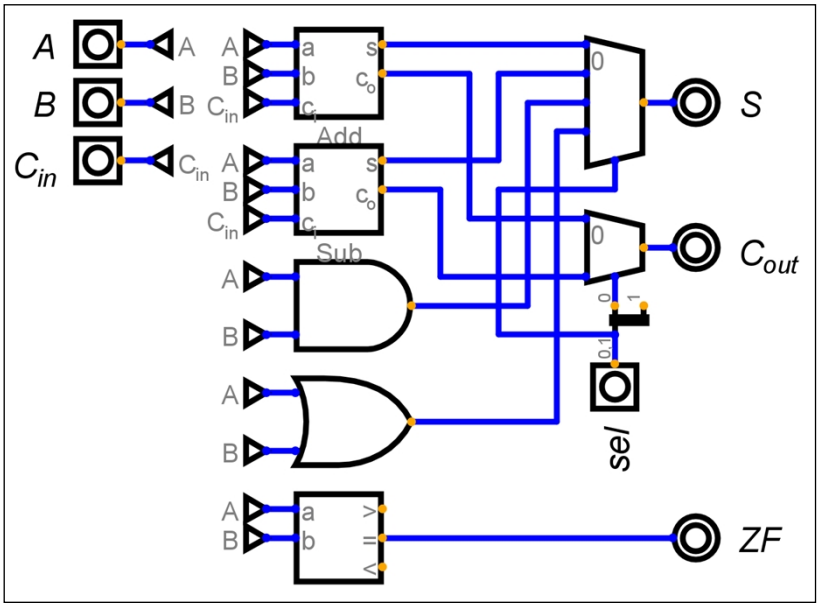
输出 Q 会跟随 sel 来决定是取输入 A 还是输入 B。而四相复用器就是两相复用器的叠加：



今后我们在使用多相复用器时，不会再绘制该电路图。

2）16 位 ALU 实现

查看如下电路：



该电路可以求 A、B 两数的加，减，按位与和按位或，最终的输出取决于 sel。映射关系如下表：

sel	结果
0b00	A+B
0b01	A-B
0b10	A&B
0b11	A|B

Cin 和 Cout 是输入输出的进位标志，ZF 是 A 和 B 是否相等的标志。右键点击 A、B、加法器，减法器，与门，或门，比较器、S，将它们的位数改为 16bit，这个 ALU 的位数就被设置成了 16bit。

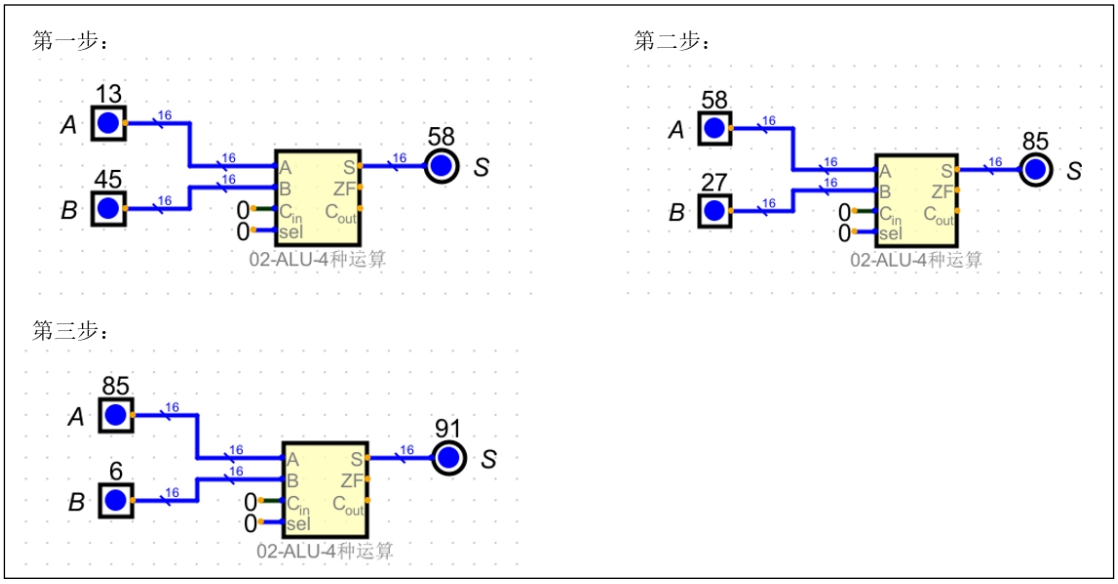
3.2 实现一个简单的计算单元

现在我们希望使用刚刚创建的 ALU 来计算一下 13+45+27+6，实现的步骤如下：

（1）A 输入 13，B 输入 45，然后可以看到 S 是累加后的结果。

（2）记录 S 的结果，并将其输入到 A，在 B 输入 27，那么 S 的结果就是 13+45+27，记录这个结果。

（3）将上一步 S 的结果输入到 A，B 输入 6，最终得到 S 结果 13+45+27+6。

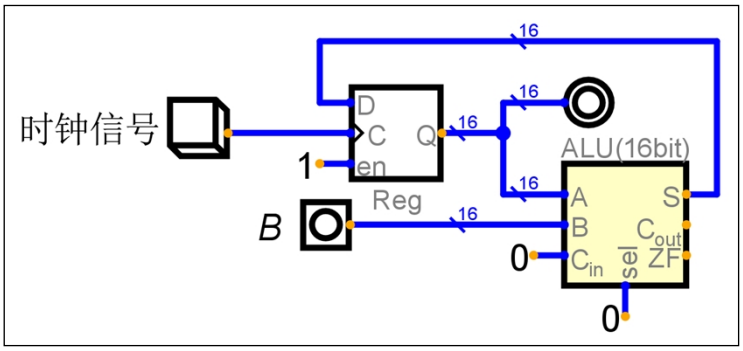


上面的操作需要我们每次手动记录 S 运算后的结果，用这个结果作为输入进行下一步骤的计算。如果我们给 ALU 添加存储功能就可以解决人工记录的问题。

接下来我们将 ALU 和寄存器连接在一起，来实现一个简单的计算单元。

- 添加一个寄存器组件，并设置为 16 位；
- 添加一个刚刚实现的 ALU 组件；
- 从“组件——输入输出”添加一个按钮组件；
- 从“组件——导线”添加三个常量组件；
- 添加一个输入和输出，并设置为 16 位，数据模式设置位 decimal。

按照下图连接电路：



注意：与 sel 连接的常量设置为 2bit，值为 0，表示我们暂时只算加法。接下来我们来尝试计算 13+45+27+6，开启仿真后，执行如下步骤：

- B 输入 13，拍下按钮。
- B 输入 45，拍下按钮。
- B 输入 27，拍下按钮。

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

- B 输入 6，拍下按钮。

最终可以看到寄存器输出显示 91，即是此加法算式的结果。在刚刚的计算过程中，我们拍按钮的动作模拟的是计算机中的时钟信号。时钟信号就是计算机电路中的节拍器，由晶振统一产生，CPU 的计算就是以时钟信号为单位进行的，时钟信号频率越高，CPU 计算速度越快，但相对的单位时间内电路中的电流就越大，发热也越大，这也是现实中限制 CPU 提升频率的一个重要因素。

3.3 添加一块数据存储

刚刚的这个累加过程，我们完全是通过手动操作进行的。而现实情况是，我们的计算一般都是通过预设的程序自动完成。而预设的程序，需要一个外部存储。对于一般的个人计算机来说，这个存储是内存和硬盘，但是此时我们为了简化计算机的外部架构，专注于 CPU 内部设计，我们就统一只使用一种外部存储，一块由多个 16bit 寄存器组成的内存。

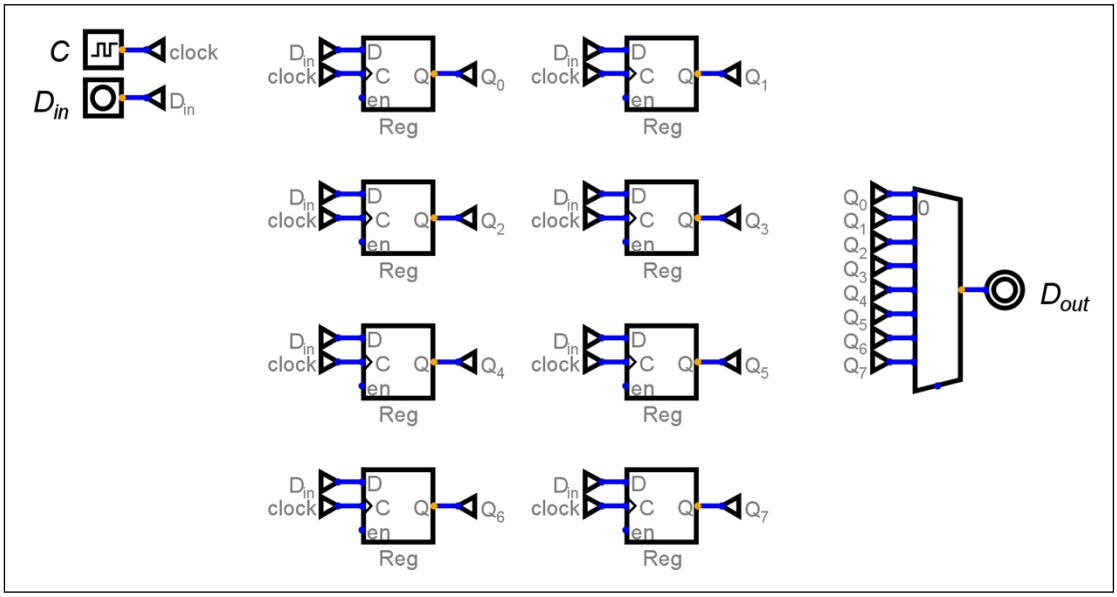
3.3.1 用寄存器实现一块内存

目标：实现一块内存，存储空间是 8*16bit。

实现思路：添加 8 个寄存器（16bit），并给每个寄存器编号（地址），通过地址操作对应的寄存器存取数据。8个寄存器就要有8个寄存器的地址，可以用3位数据来控制。因此，需要先创建一个 3-8 译码器。通过 3 位的数据来控制 8 个内存地址。

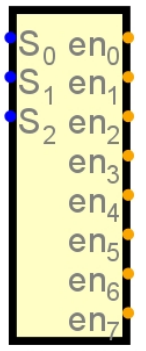
- 添加 8 个寄存器，并设置为 16bit。
- 添加输入 A（地址位），str（写允许），Din（数据），ld（输出允许），将 A 设置为 3bit，Din 设置为 16bit。
- 添加输出 Dout，设置为 16bit。
- 添加下拉电阻。
- 添加复用器，数据位数设置为 16，选择位数设置为 3。
- 从“组件——导线”中添加驱动器，驱动器就相当于一个继电器模块。

首先将时钟输入，数据输入和数据输出按照如下方式连接：



数据输入直接并联了 8 个寄存器，此时我们可以通过控制 en 信号来决定 Din 实际写入哪个寄存器；同时我们可以通过控制复用器的选择信号决定哪个寄存器的数据可以输出到 Dout。例如，我们让第寄存器 1 号的 en 为 1，其余的 en 为 0，即可写入寄存器 1 号；我们让复用器的选择器输入 0b000，即可选择从寄存器 1 号输出。

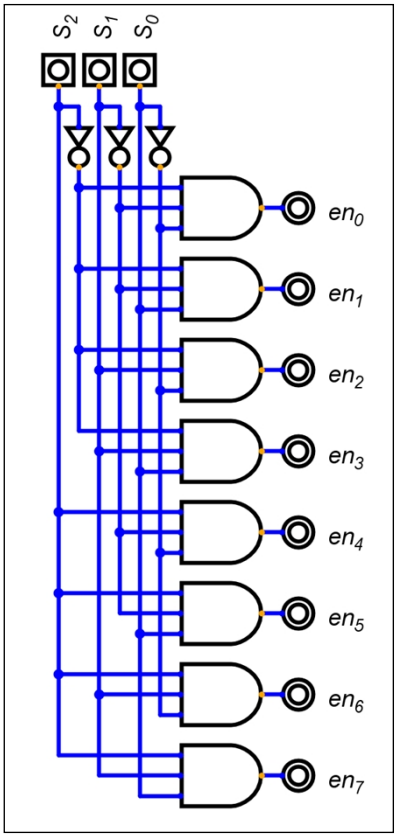
现在的问题是，我们可以通过复用器控制哪个寄存器输出，但是我们需要一个选择器帮我们选择输入的寄存器。这个选择器应该是 3bit 输入，8bit 输出，封装和真值表类似下图：



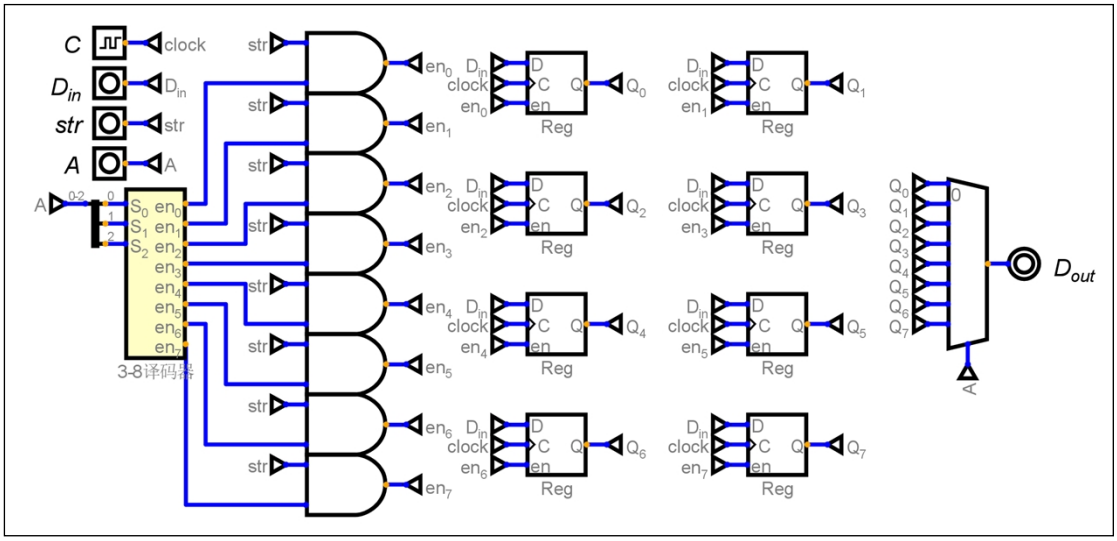
3-8译码器

S2	S1	S0	en0	en1	en2	en3	en4	en5	en6	en7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

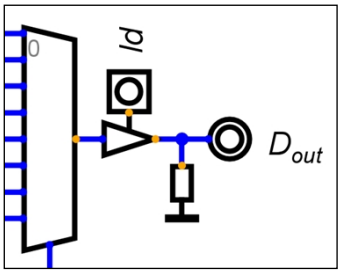
这个经典电路就是 3-8 译码器，电路如下：



我们将封装好的 3-8 译码器放入电路图，效果如下：



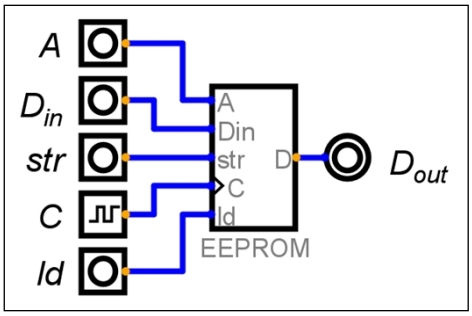
最后我们在输出端加上一个控制电路：



我们就实现了一个读写可控的 16x8bit 内存。

现实中的内存不可能用寄存器这种高成本电路，但为了简化外部存储结构，我们并不会讲解 SRAM 和 DRAM 架构。

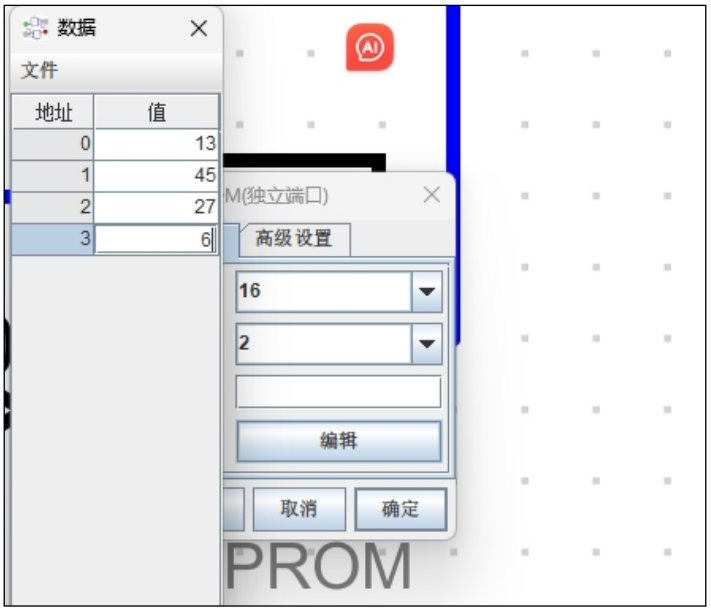
之后我们在使用外部存储的时候，不会再次绘制该电路。为了之后编程方便，我们选择封装好的 EEPROM，如下图：



该封装可以非常方便的预编辑其内部存储，便于我们之后的调试。

3.3.2 给计算单元接入外部存储

对我们之前的计算单元做一点小小的改造，我们添加一块 EEPROM，将数据位数设置为 16，地址位数设置为 2，并将我们刚刚的四个数写到这块存储中。如下图所示。



我们希望计算机可以从该内存中逐条的读取数据，并将数据输入到计算单元中进行计算。那就需要逐条的将内存中的数据按照地址顺序取出。地址位是 2bit，所以，取值是 00、01、10、11。需要设计一个程序计数器 PC，来完成地址位累加的变化。

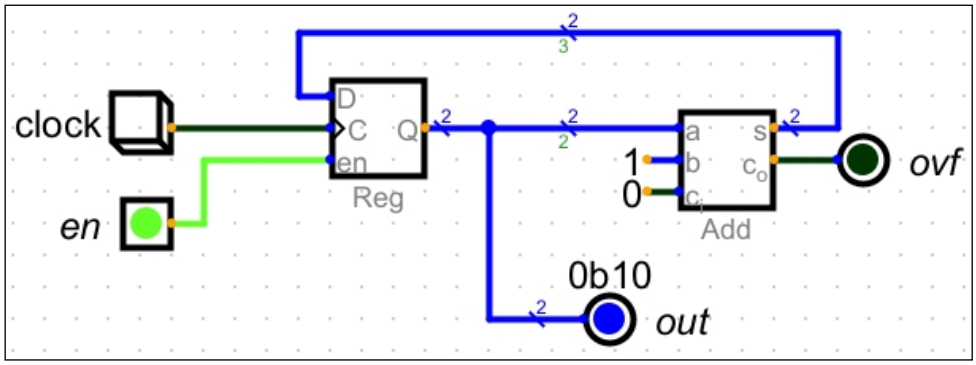
1）程序计数器（Program counter 简称 PC）

由于预设了数据，我们不需要对这块存储进行修改，所以 Din 和 str 全部接 0（Din 注意数据位数），ld 设置为 1，表示我们要一直读取数据。数据读取的地址信号要通过 A 传入，但是 A 正好是每时钟周期递增 1，所以我们可以做一个计数器，每次时钟周期自动递增 1。

计数器的电路比较简单，就是一个寄存器拼一个加法器，这里直接给出电路图：

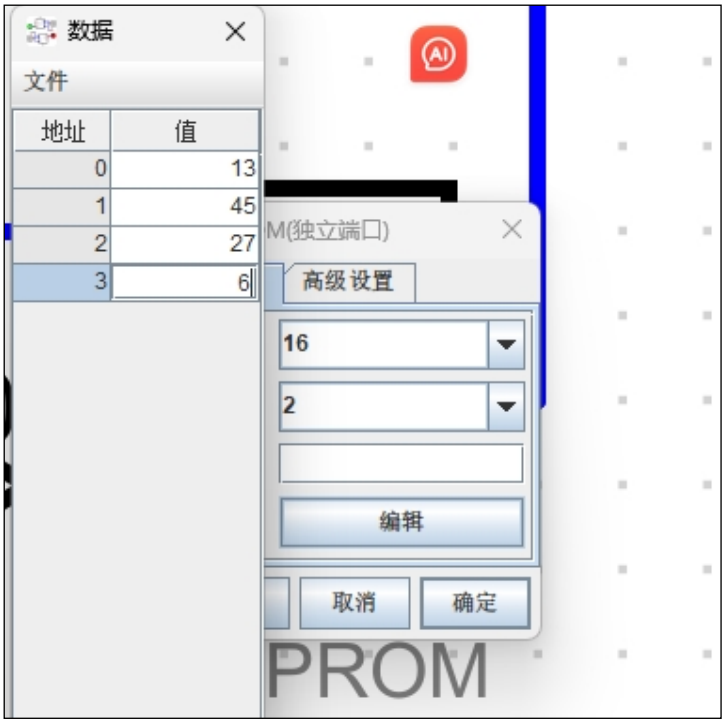
因为要完成 4 个数的累加，我们使用一个 2 位加法器，并且设置 Cin 始终为 0。加法器的一个输入恒为 1，另一个输入是寄存器的输出，也就是说加法器完成的功能就是将寄存器的输出加 1。而加法器的输出又作为寄存器的输入，每当时钟信号上升沿到来的时候都把累加的结果存储在该寄存器中。

这样的话，初始情况下我们只需要将寄存器清零，然后每次时钟信号上升沿到来后，寄存器的输出都会自动加 1～也就是说上图中 out 的值将会是 0、1、2、3 完成累加。



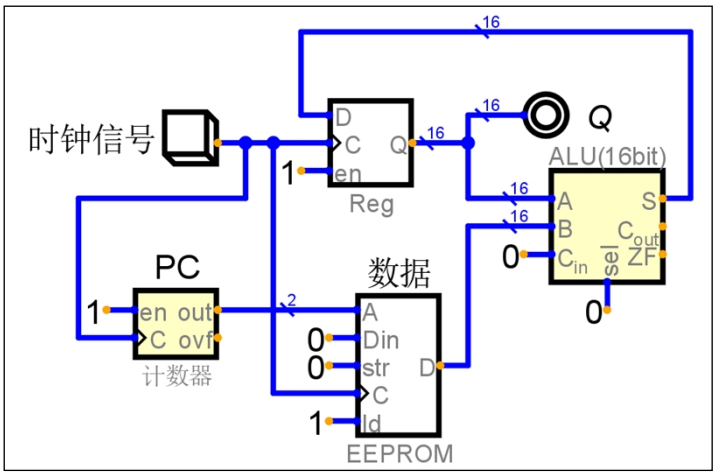
2）通过 PC 计数器逐条读取内存中的数据进行计算

数据 RAM 中的数据存储如下：



PC 计数器累加后的结果恰好就是 EEPROM 的地址，这样就可以通过 PC 计数器取出内存中对应地址存储的数据，进行算数运算了。

当 en 信号打开，每个时钟周期该计数器递增 1，我们将寄存器和加法器的位数都设为 2，以匹配 EEPROM，并将该子电路加入计算单元，最终结果如下：

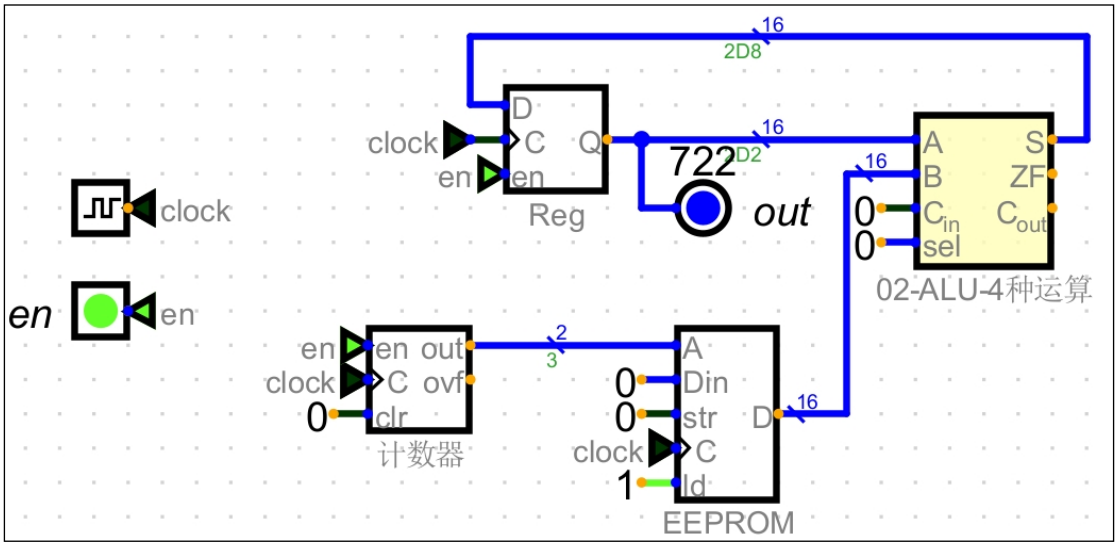


开启仿真，我们连续拍 4 下时钟信号，即可得到加和结果 91。

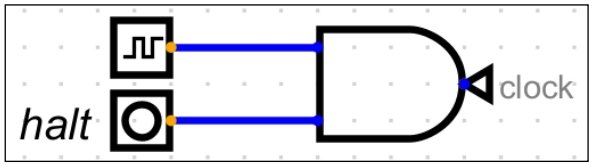
3）添加 halt 信号

以上我们是通过点击的方式模拟时钟信号的，但真正的时钟信号是一个周期性循环的震荡信号。如果我们将其换成震荡时钟信号后，会发现程序计数器会反复从 0,1,2,3,0,1,2,3….执行，进而使得内存中的数据被反复的累加。而不是累加一遍就结束。

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

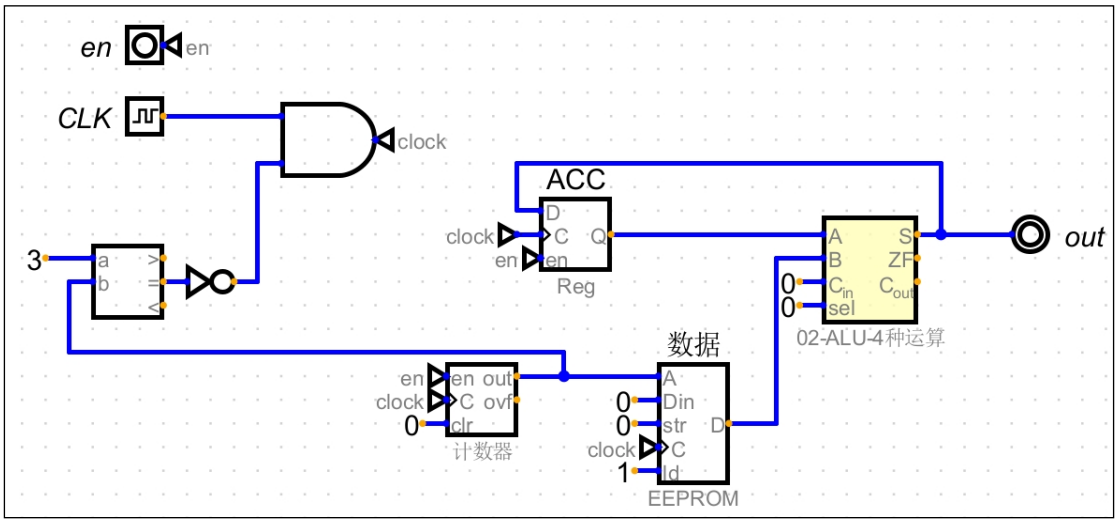


我们希望程序能够在PC计数器输出为3时终止。因此，加入一个halt终止信号，当halt值为 1 时，PC 累加，halt 为 3 时，停止时钟信号的输出。使用一个与门，当 halt 为 1 时，clock 有时钟信号，halt 为 0 时，clock 时钟信号永远输出 0。



接下来的问题是，如何判断 PC 计数器的值为 3 呢？可以通过一个比较器来实现。比较器的作用是，比较两个输入，如果相等则输出 1，否则输出 0。改造后的电路如下：

注意：此处输出取的是 ALU 中的 S 端。



分析一下整个电路不同时钟周期时的输入、输出情况。

时钟周期	PC 输出（数据存储的地址）	RAM 输出	ACC 输出	ALU-sel 控制信号

0	0（00）	13	0	00（加）
1	1（01）	45	13	00（加）
2	2（10）	27	58	00（加）
3	3（11）	6	85	00（加）

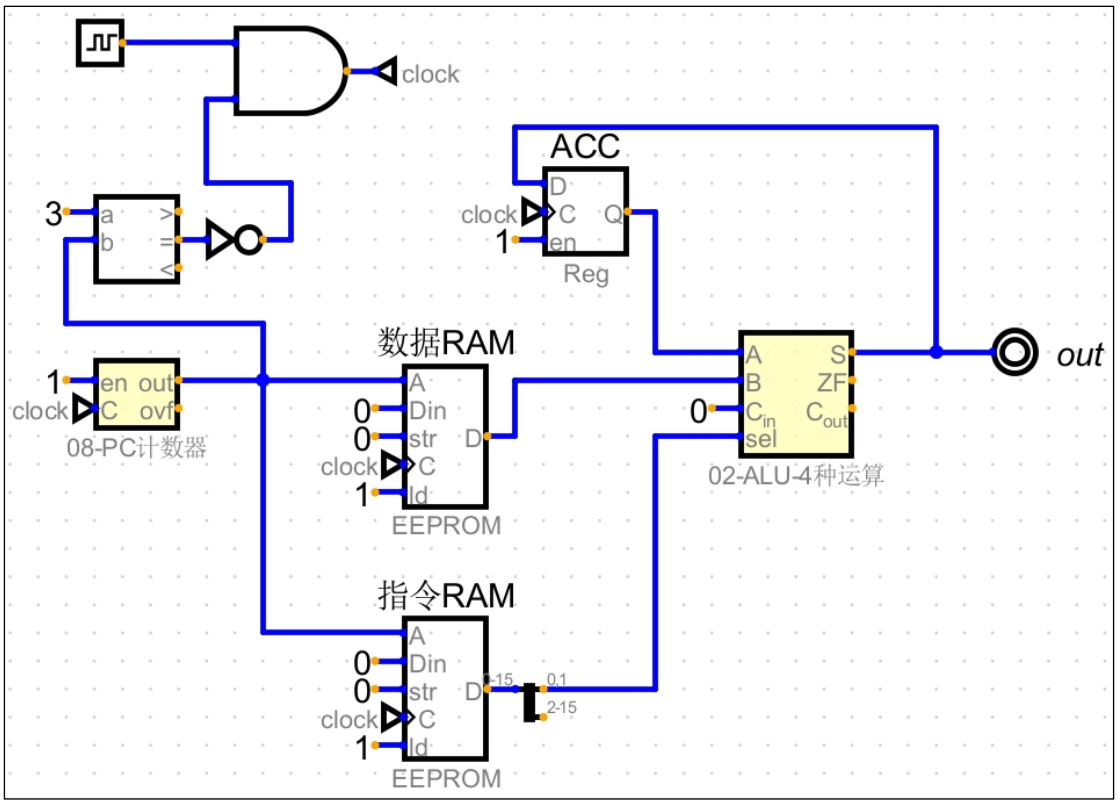
通过分析发现，PC 计数器累加后，可以从内存中读取数据到 ALU 进行运算，而 ALU 的控制引脚一直输入的是 00（加），因此，执行的结果是将数据 RAM 中的数据进行了累加。那如果我们改变 ALU 的控制引脚，使其进行其他操作，那么，我们的程序就可以进行复杂的运算了。因此，我们考虑添加一块指令存储器，用来存储 ALU 的操作模式。

3.4 添加一块指令存储

目前我们的程序只能计算四个数的加法，那如果我们想计算 13+45-27-6 呢？毕竟我们的 ALU 是支持减法运算的。这时候我们就要一段输入信号去控制 ALU 的 sel 输入。

我们加入一块新的 EEPROM，将数据设置为 16bit，地址设置为 2bit，并输入数据 0、0、1、1，对应 ALU 的加法、加法、减法、减法。将 EEPROM 的输出连接到 ALU 的 sel，这中间注意做位数转换。

将这块 EEPROM 的时钟信号和 Clock 相连，Din，str，ld 的处理与数据 RAM 一样。地址位也连上计数器输出，最终效果如图所示：

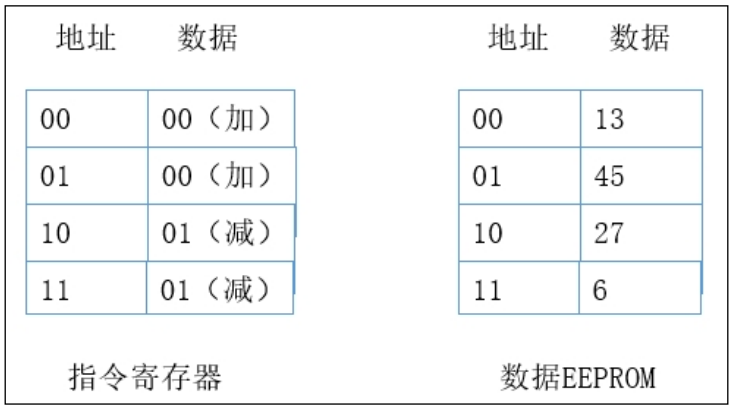


如果我们想要实现 13+45-27-6 的操作，我们只需要改变 ALU-sel 引脚的控制信号即可。

时钟周期	PC 输出（数据存储的地址）	数据 RAM 输出	指令 RAM 输出	ACC 寄存器输出	ALU-sel 控制信号
0	0（00）	13	00	0	00（加）
1	1（01）	45	00	13	00（加）
2	2（10）	27	01	58	01（减）
3	3（11）	6	01	31	01（减）

因为 ALU-sel 引脚的控制信号决定了 ALU 做何种运算，所以，ALU 的 sel 控制信号又叫做指令。而新引入的指令寄存器中目前存储的就是 ALU 的控制信号，专门用来存储指令。原来的 EEPROM 存储的是数据。

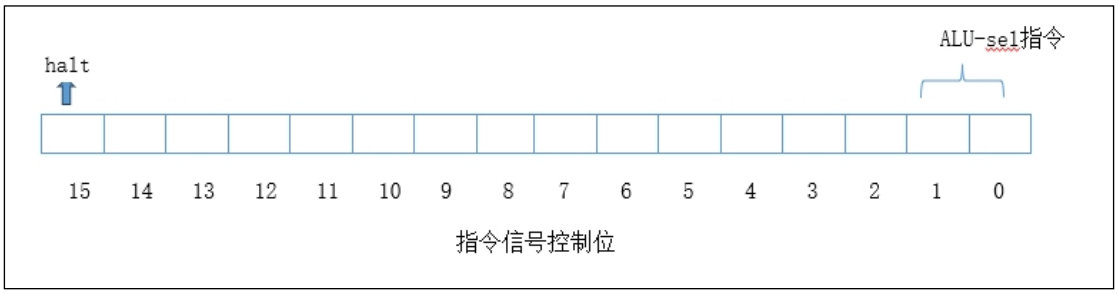
为了实现 13+45-27-6 的操作，那么数据 RAM 中存储数据如下：



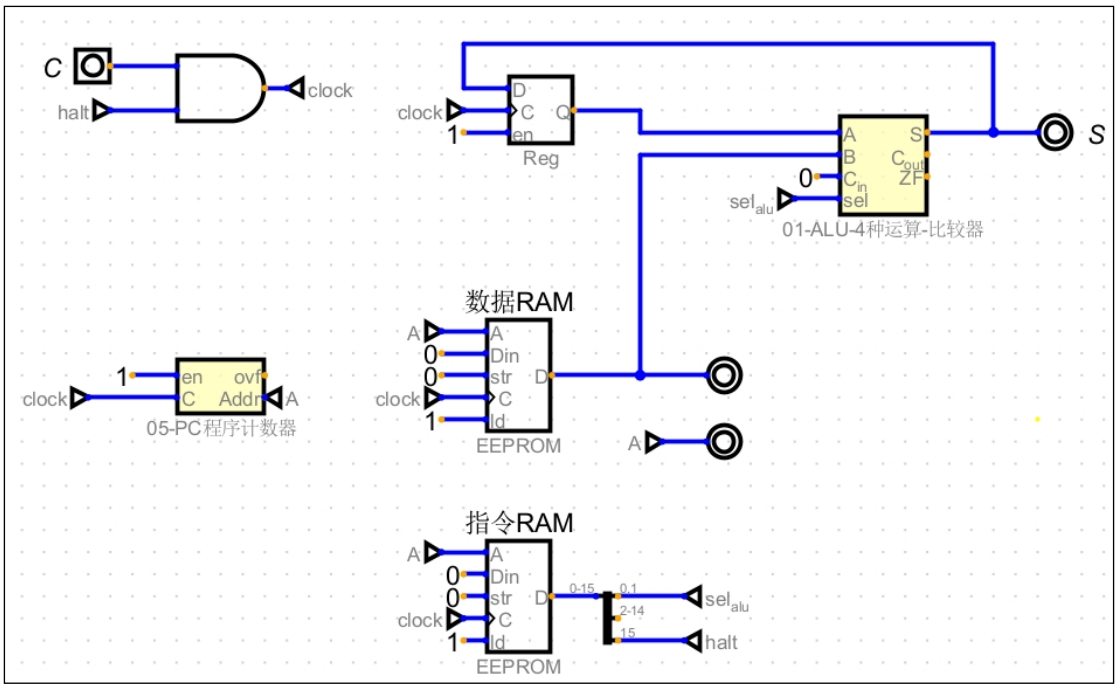
3.5 添加其他控制指令

3.5.1 添加 halt 信号位

上面的电路中没有程序终止的指令，我们给指令寄存器的第 2 位作为程序终止的 halt 信号位，当 halt=1 时，程序执行，当 halt=0 时，程序终止。



加入 halt 终止指令后的电路图如下：



我们想要实现 13+45-27，之后终止程序。那么对应的数据存储和指令寄存器的数据，如下图所示：

halt指令

地址	数据16bit															功能
指令名	halt													sel_alu		
16bit位	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
00	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	加
01	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	加
10	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	减
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	x	x	停止
指令RAM																	

地址	数据
00	13
01	45
10	27
11	
数据RAM	

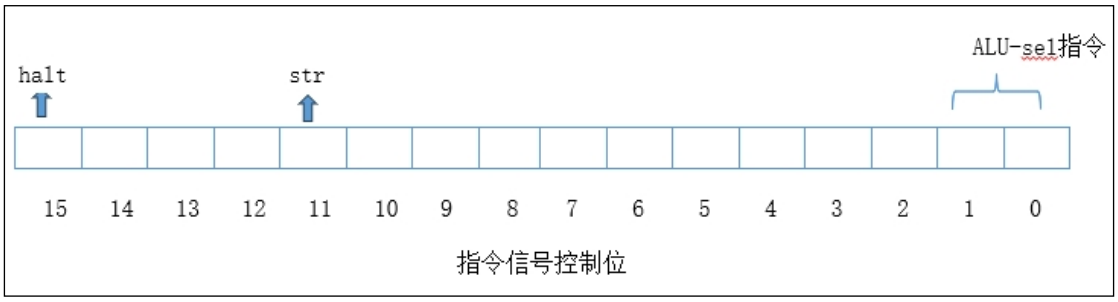
将如上数据输入数据 RAM 和指令寄存器后，仿真执行测试 halt 指令。

电路中各个时钟周期，各设备的输出情况：

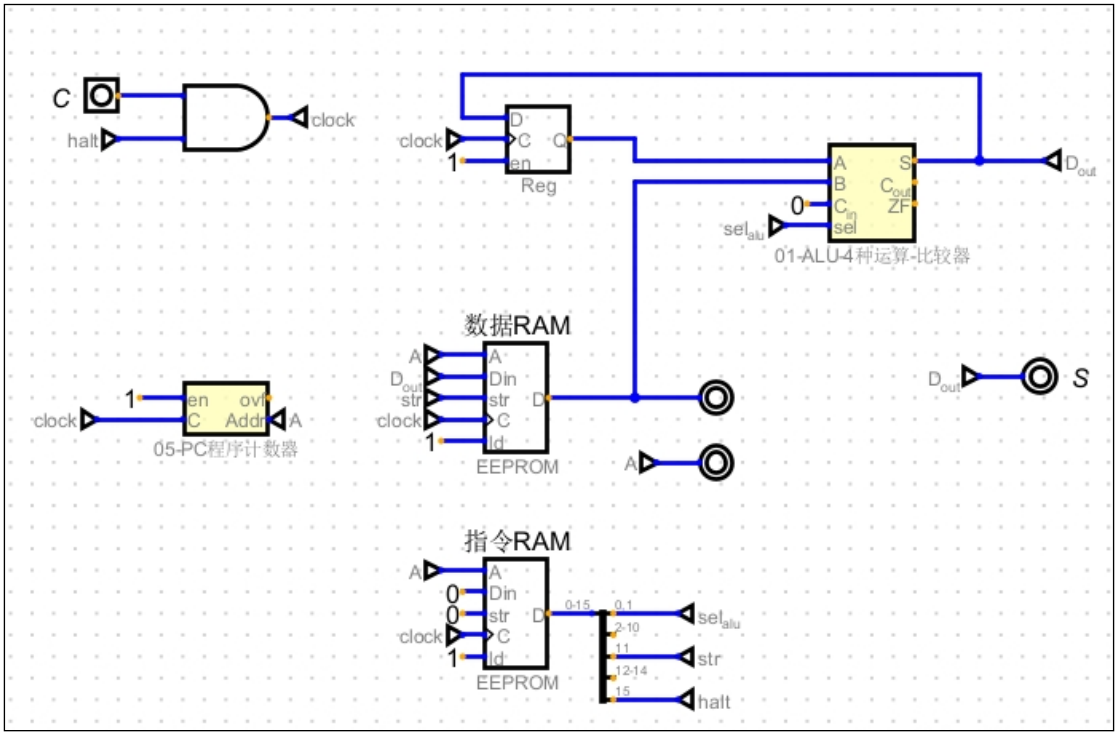
时钟周期	PC 输出（数据存储的地址）	数据 RAM 输出	A 寄存器输出	ALU-sel 控制信号
0	0（00）	13	0	00（加）
1	1（01）	45	13	00（加）
2	2（10）	27	58	01（减）
3	3（11）	xx	xx	xx

3.5.2 添加 str 存储信号位

我们希望将运算之后的结果回存到数据 RAM 中，那就需要对数据 RAM 的 str 引脚信号进行控制，因此指令 RAM 中输出的 16bit 的数据中，需要有 1bit 对其进行控制。此处我们选择 11 号位。加入 str 存储控制后的指令信号控制位如图：



当 str=1 时，将 ALU 运算后的结果，存储进数据 RAM 中。添加 str 控制信号后的电路图如下：



- 指令寄存器 D 输出信号增加分裂器，添加 str 指令信号。
- ALU 的 S 输出接入数据内存的 Din。
- str 引脚接 str 信号。
- 修改数据 RAM 的地址位为 3 位。
- 修改指令 RAM 的地址位为 3 位。

- 修改程序计数器的位数为 3 位。
- 因为最后计算的结果会被存储到数据 RAM 中，所以，取消了 out 输出。

我们要实现一个计算，13+45-27，并将计算的结果存储到数据 RAM 的 011 地址中。

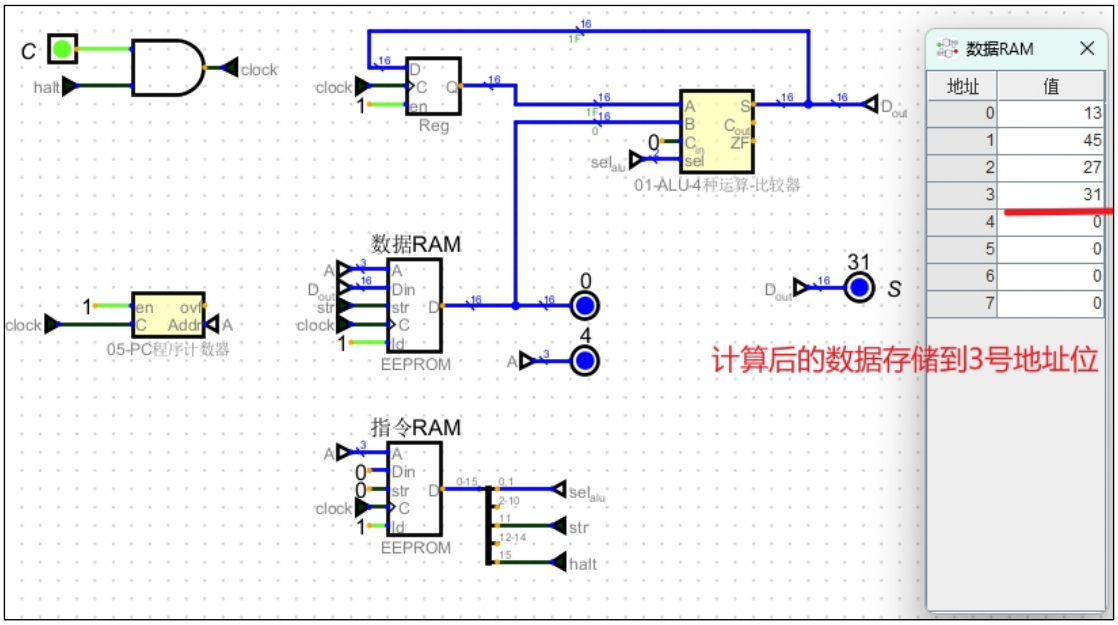
数据 RAM 和指令 RAM 存储程序如下：

str指令

地址	数据16bit																功能
指令名	halt				str										sel_alu		
16bit位	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
000	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	加
001	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	加
010	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	减
011	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	存储
100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	停止
指令RAM																	

地址	数据
000	13
001	45
010	27
011	xx
100	xx
数据RAM	

开启仿真执行后的效果如图：



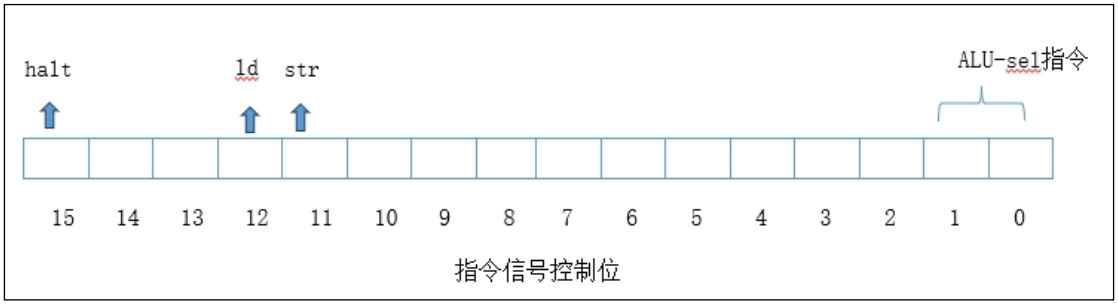
在电路运行各个时钟周期，各个设备的输出情况如图所示：

时钟周期	PC 输出（数据存储的地址）	指令寄 RAM 制信号	数据 RAM 输出	A 寄存器输出
0	0（000）	1000000000000000（加）	13	0
1	1（001）	1000000000000000（加）	45	13
2	2（010）	1000000000000001（减）	27	58
3	3（011）	1000100000000000（str）	xx	31
4	4（100）	0000000000000000（halt）	xx	xx

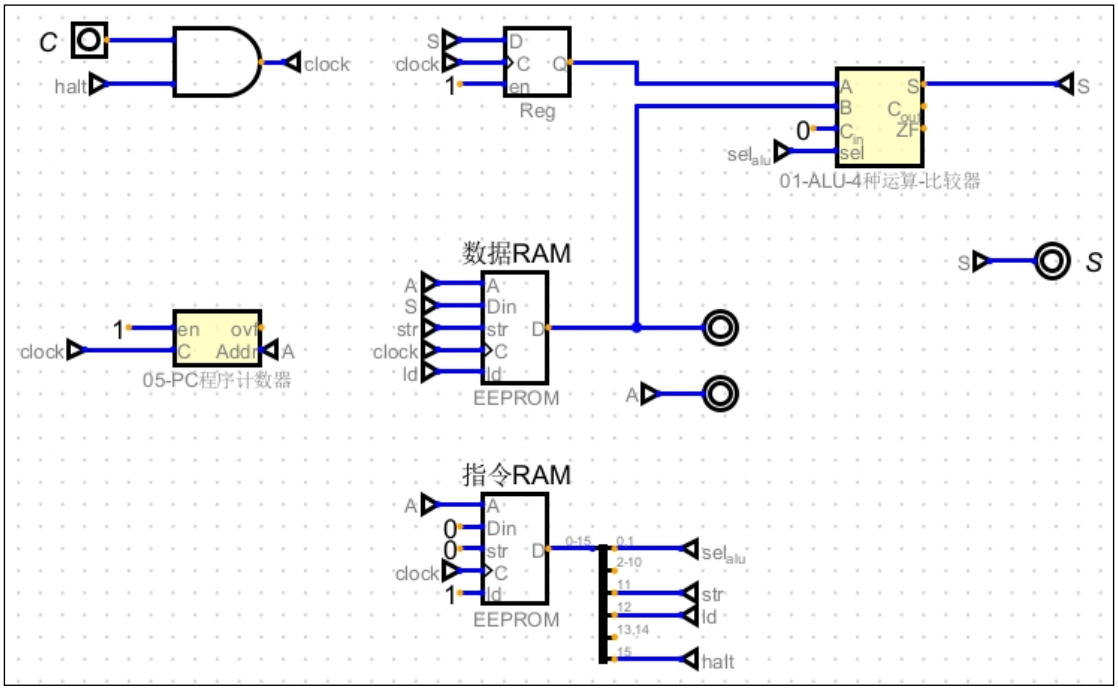
3.5.3 添加 ld 数据 RAM 输入信号、selB 数据选择信号

上面的电路中，当执行 str 存储操作的时候，数据 RAM 的输出其实并不重要，但是这个数据输出仍然会被输入给 ALU 进行计算，我们不希望这样的情况出现，因此，可以通过控制数据 RAM 的 ld 引脚，来控制数据 RAM 的输出。当 ld=1 时，数据 RAM 可以输出数据到 ALU，当 ld=0 时，不输出任何数据。

添加 ld 指令后的信号控制如图：



增加 ld 信号指令后，电路图如下：



注意：此处的电路仿真时会有 bug，为了再现问题，我们先使用这个电路。

- 指令 RAM 添加 ld 信号，及 ld 管道。
- 数据 RAM 的 ld 引脚接入 ld 信号管道。

仍然执行 13+45-27 计算，然后将计算结果存储到 3 号地址，增加 ld 信号指令后，数据 RAM 和指令 RAM 的数据如下：

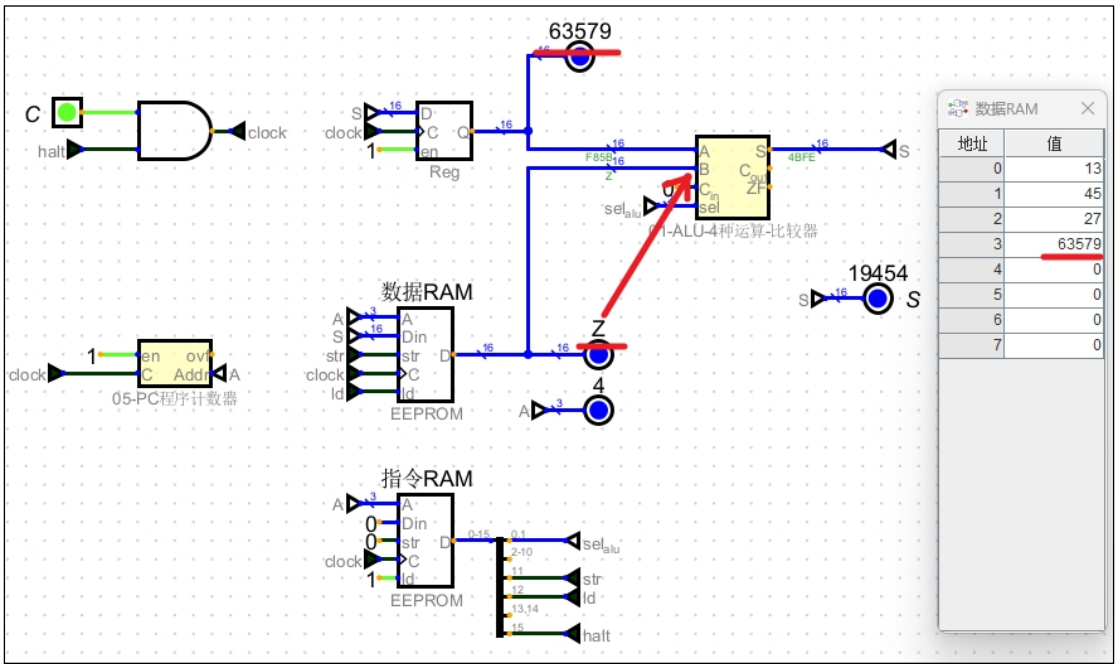
更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

ld指令

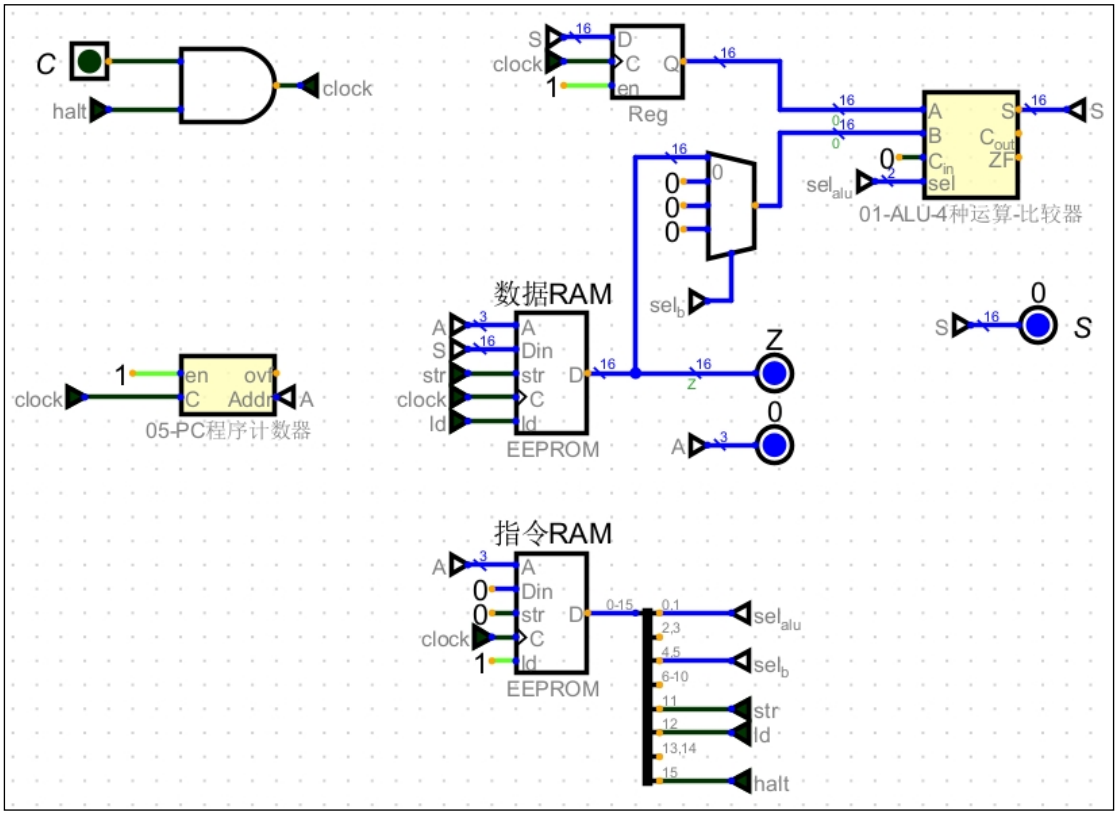
地址	数据16bit																功能
指令名	halt			ld	str						sel_b				sel_alu		
16bit位	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
000	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	加
001	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	加
010	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	减
011	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	存储
100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	停止
指令RAM																	

地址	数据
000	13
001	45
010	27
011	xx
100	xx
数据RAM	

执行仿真，会发现数据 RAM 的 3 号地址存储的数据是随机不确定的，并且数据 RAM 的 D 输出是 Z（Z 表示高阻态），如图所示：



原因在于数据 RAM 的 ld=0 时，D 输出 Z 表示高阻态，导致 ALU 的 B 引脚输入是不确定的。这里我们需要将 ld=0 时的 D 输出设置为 0，ld=1 时能够正常读取数据 RAM 中的数据既可以解决问题。因此需要添加一个复用器，进行数据选择。并且增加一个数据选择的信号控制 sel_B（2bit）。修改电路图如下：



当 ld=0 时，selB=01, ALU 的 B 输入取 0。

当 ld=1 时，selB=00,ALU 的 B 输入为数据 ROM 的 D 输出。

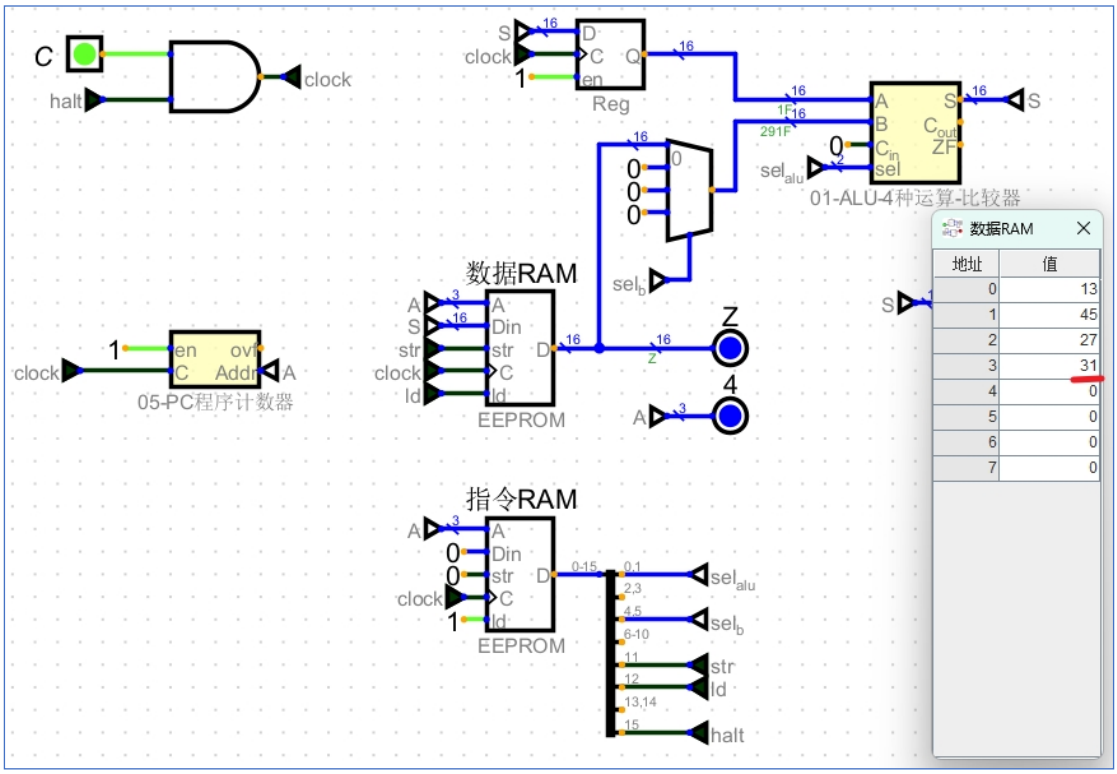
指令 RAM 的数据控制位如图：

sel_b指令

地址	数据16bit															功能
指令名	halt			ld	str						sel_b				sel_alu		
16bit位	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
000	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	加
001	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	加
010	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	减
011	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	存储
100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	停止
指令RAM																	

地址	数据
000	13
001	45
010	27
011	xx
100	xx
数据RAM	

开启仿真，执行后的效果如图：



3.5.4 添加 enA、selA 信号

我们又有了一个新的需求，希望连续计算两个表达式的值。比如：13+45-27=31 的结果存放在地址 3 处和（15|8）&23 = 7 的结果存放在地址 7 处。那么我们可以分别向数据 RAM 和地址 RAM 中输入如下数据：

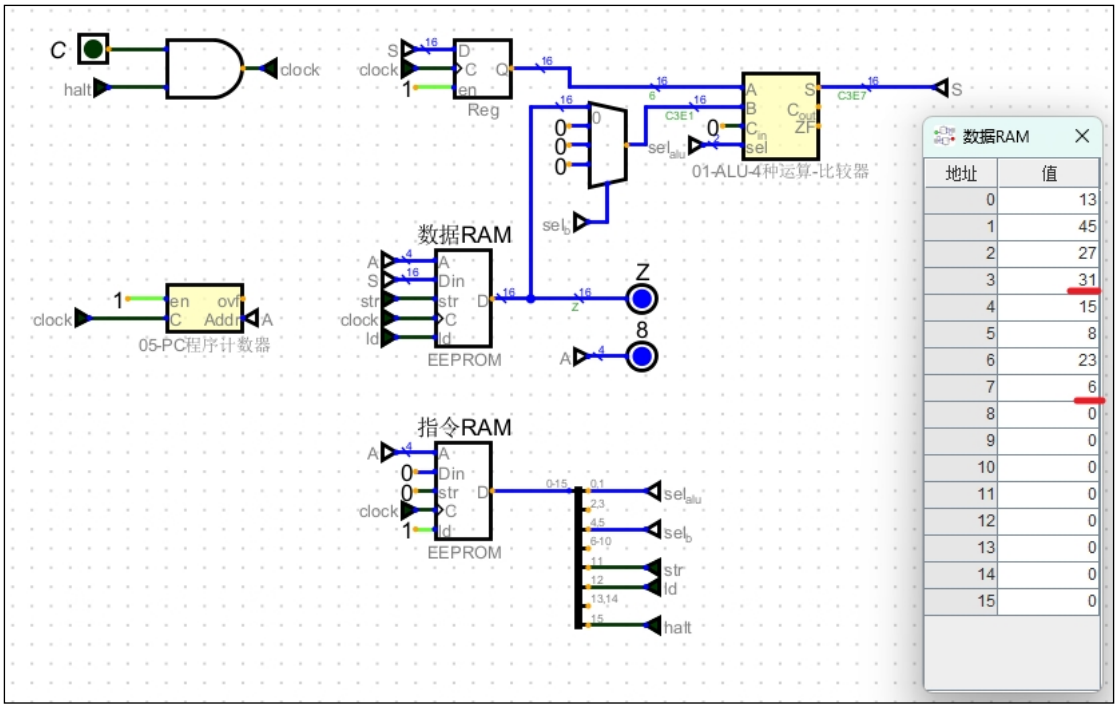
sel_a指令_bug

地址	数据16bit																功能
指令名	halt			ld	str						sel_b				sel_alu		
16bit位	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0000	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	加
0001	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	加
0010	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	减
0011	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	存储
0100	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	加
0101	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	或
0110	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	与
0111	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	存储
1000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	停止
指令RAM																	

地址	数据
0000	13
0001	45
0010	27
0011	xx
0100	15
0101	8
0110	23
0111	xx
1000	xx
数据RAM	

9条指令，已经超过了PC计数器的范围，所以需要将PC计数器改成4位，
相应的数据RAM的地址位、指令RAM的地址位都要改成4位

最终运算的结果如图，跟我们的设想不符合。



我们将电路各个时钟周期，各设备的输出情况罗列，如下图所示：

时钟周期	PC 输出（数据存储的地址）	指令 RAM 控制信号	数据 RAM 输出	A 寄存器输出
0	0（0000）	0b1001000000000000（加）	13	0
1	1（0001）	0b1001000000000000（加）	45	13
2	2（0010）	0b1001000000000001（减）	27	58
3	3（0011）	0b1000100000010000（str）	0	31
4	4（0100）	0b1001000000000000（加）	15	31
5	5（0101）	0b1001000000000011（或）	8	46
6	6（0110）	0b1001000000000010（与）	23	46
7	7（0111）	0b1000100000010000（str）	0	6
8	8（1000）	0b0000000000000000（halt）	xx	6

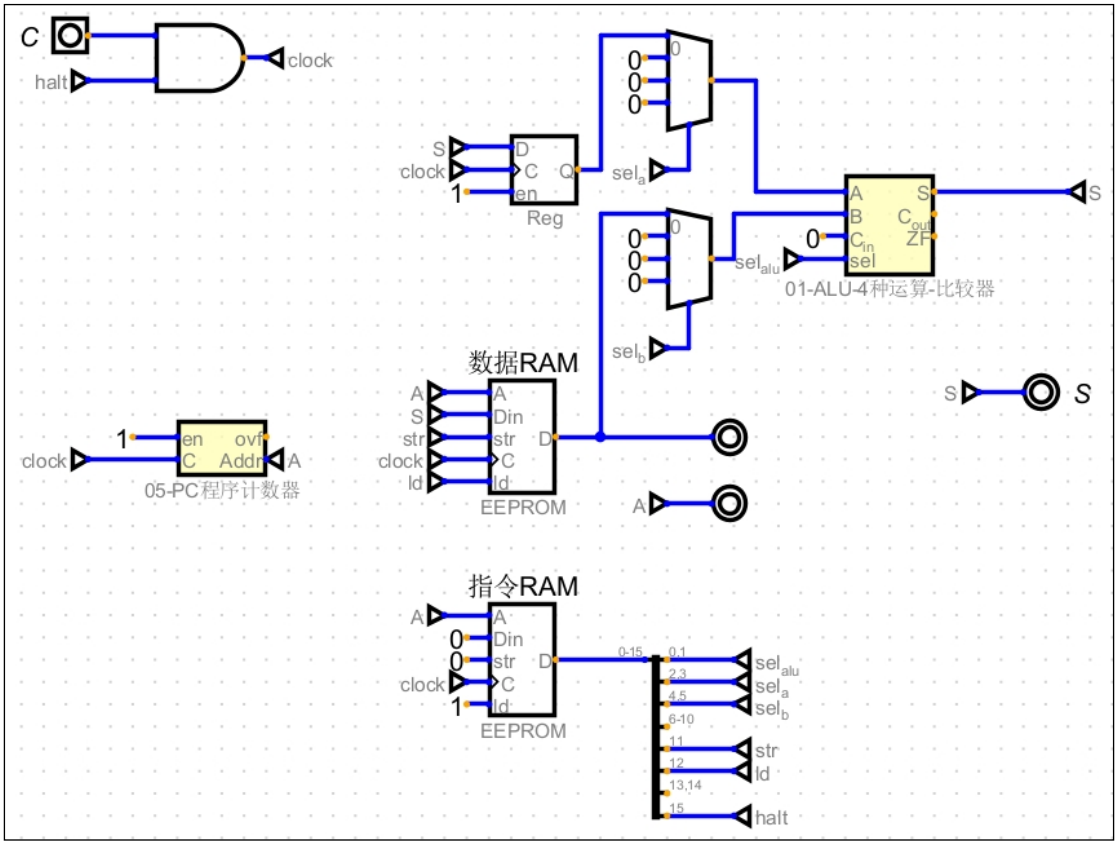
发现问题出现在第 4 个时钟周期，第四个时钟周期将上一次运算的结果，13+45-27=31，存储到地址 3 之后，在进行后面运算的时候，应该将数据 RAM 的值 15 直接存储在 A 寄存器累加器中，但我们这里会发现，A 寄存器累加器进行完上一次运算后，并没有被清 0。仍然保持了上一次运算的结果 31。导致下一次运算的 15 与 A 寄存器累加器的 31 进行了累加得到 46。导致后续的结果都错误了。

因此，我们需要增加控制位来应对，当进行多次运算时，能够直接从数据 RAM 中读取数据到 A 寄存器。如何做到呢？当要执行将数据 RAM 的值直接输入到 A 寄存器时，需要

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

将 ALU 的 A 输入设置为 0 即可，因为 ALU 的 B 输入是从数据 RAM 中直接读取数据，我们选择 ALU 的运算模式为加，那么 ALU 的输出 S=A 输入+B 输入（数据 RAM 的值），我们只需要保证此时 A 输入为 0，那么 ALU 的输出 = 0 + 数据 RAM 中的值。ALU 的输出 S 又是 A 寄存器的输入。因此，数据 RAM 中的数据通过 ALU 的 B 输入配合加运算就会直接输入到 A 寄存器中。那么，通 ALU 的 B 输入一样，我们增加一个复用器，增加一个 selA 的控制信号即可。

改造后的电路图如下：



当完成上一次计算进行新的计算时，我们就让 selA=01，selB=00 使得 ALU 的 A 输入=0，ALU 的 B 输入是数据 RAM 中的数据。数据 RAM 中的值会直接进入 A 寄存器。我们将这个操作称为 ld_a（load a 寄存器缩写）。

添加 selA 指令后的信号控制位及指令如图：

指令信号控制位																
指令位	halt			ld	str						sel_b		sel_a		sel_alu	
16bit位	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
最新指令集																
ld_a	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
add	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
sub	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1
or	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1
and	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0
str	1	0	0	0	1	0	0	0	0	0	0	1	0	0	x	x
halt	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

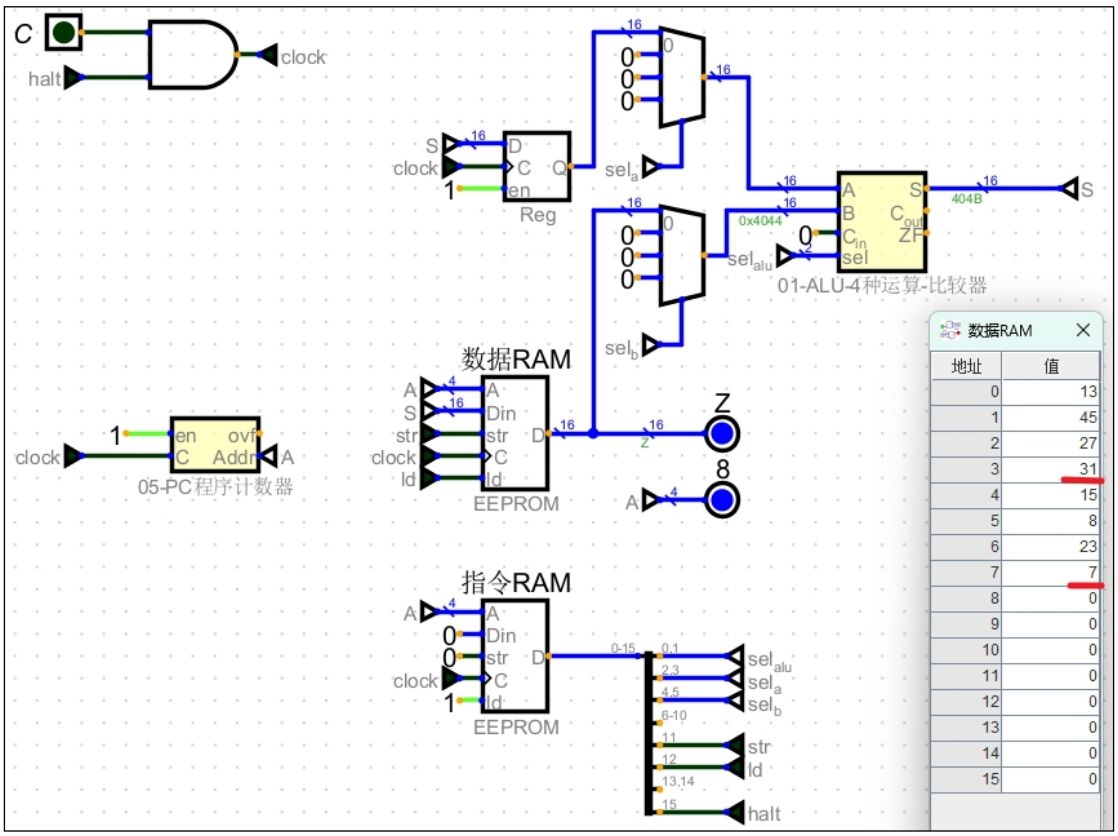
根据最新的电路及指令完成之前的连续两次运算，13+45-27=31 的结果存放在地址 3 处和（15|8）&23 = 7 的结果存放在地址 7 处。那么我们可以分别向数据 RAM 和指令 RAM 中输入如下数据。

sel_a指令_done

地址	数据16bit																功能
指令位	halt			ld	str						sel_b		sel_a		sel_alu		
16bit位	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0000	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	ld_a
0001	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	加
0010	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	减
0011	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	存储
0100	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	ld_a
0101	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	或
0110	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	与
0111	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	存储
1000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	停止
指令RAM																	

地址	数据
0000	13
0001	45
0010	27
0011	xx
0100	15
0101	8
0110	23
0111	xx
1000	xx
数据RAM	

将数据输入数据 RAM 和指令 RAM，执行仿真查看结果，地址 3 的值是 31，地址 7 的值是 7。



3.5.5 添加 jmp 跳转信号

通过上面的学习，我们应该已经有了一种意识，我们通过控制指令 RAM 中的指令的 0、1 状态，去控制电路中的对应开关，来实现对应的功能。我们要实现一种运算，只需要在指令 RAM 填好指令，在数据 RAM 中填好要操作的数据，那么电路就会根据开关指令去操作数据得到结果。

我们现在想要实现一系列的运算 8+17+3-4+2-7+20，并将结果存放在地址 7。那么，数据 RAM 和指令 RAM 的数据如下图：

地址	数据16bit																功能
指令位	halt			ld	str						sel_b		sel_a		sel_alu		
16bit位	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0000	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	ld_a
0001	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	加
0010	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	加
0011	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	减
0100	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	加
0101	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	减
0110	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	加
0111	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	存储
1000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	停止
指令RAM																	

地址	数据
0000	8
0001	17
0010	3
0011	4
0100	2
0101	7
0110	20
0111	xx
1000	xx
数据RAM	

但我们又突然想略过中间的步骤，直接执行后续的操作。比如：8+17-7+20 并存储。如图：

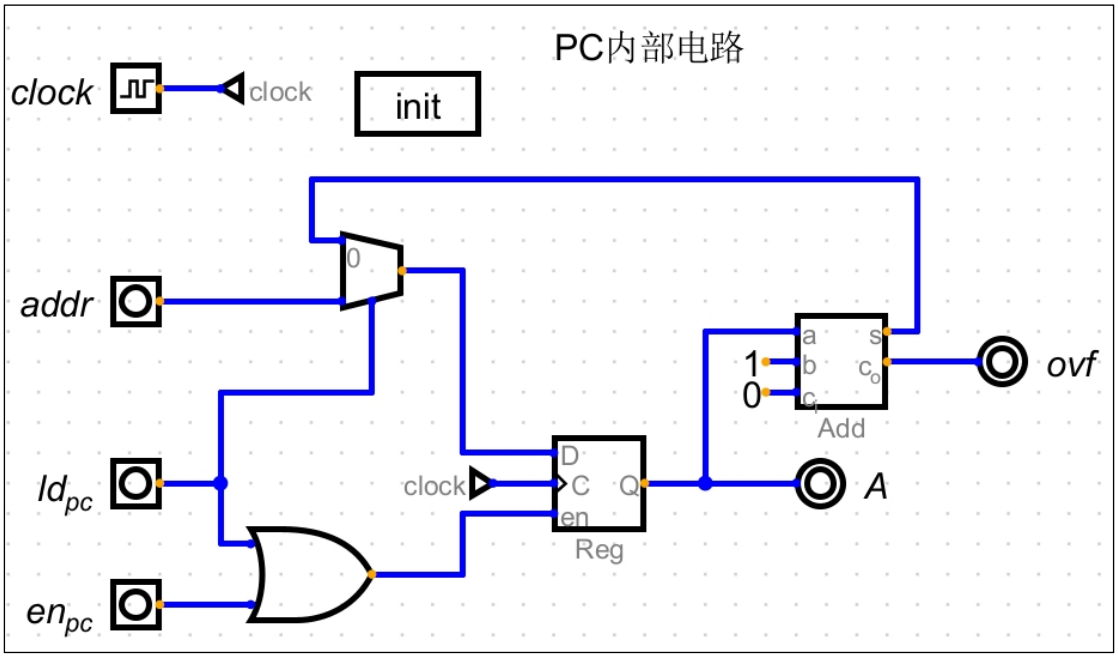
更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

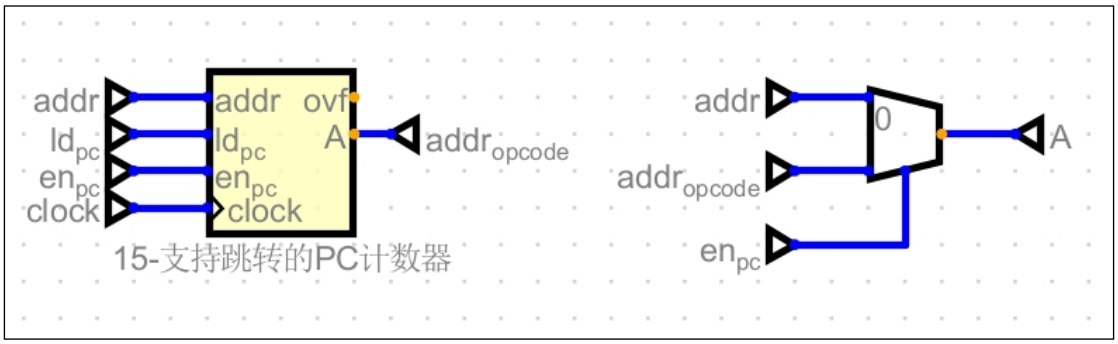
地址	数据16bit																功能
指令位	halt			ld	str						sel_b		sel_a		sel_alu		
16bit位	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0000	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	ld_a
0001	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	加
~~0010~~	~~1~~	~~0~~	~~0~~	~~1~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~加~~
~~0011~~	~~1~~	~~0~~	~~0~~	~~1~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~1~~	~~减~~
~~0100~~	~~1~~	~~0~~	~~0~~	~~1~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~加~~
0101	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	减
0110	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	加
0111	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	存储
1000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	停止
指令RAM																	

地址	数据
0000	8
0001	17
~~0010~~	~~3~~
~~0011~~	~~4~~
~~0100~~	~~2~~
0101	7
0110	20
0111	xx
1000	xx
数据RAM	

我们知道，程序执行的顺序是靠 PC 计数器累加地址实现的。现在我们跳过了中间的步骤，其实是让程序执行 0、1、5、6、7、8 的指令。连续累加的指令执行仍然使用原来 PC 计数器的累加功能即可实现。跳转功能的实现其实就是让 PC 计数器的输出能够指定地址执行。

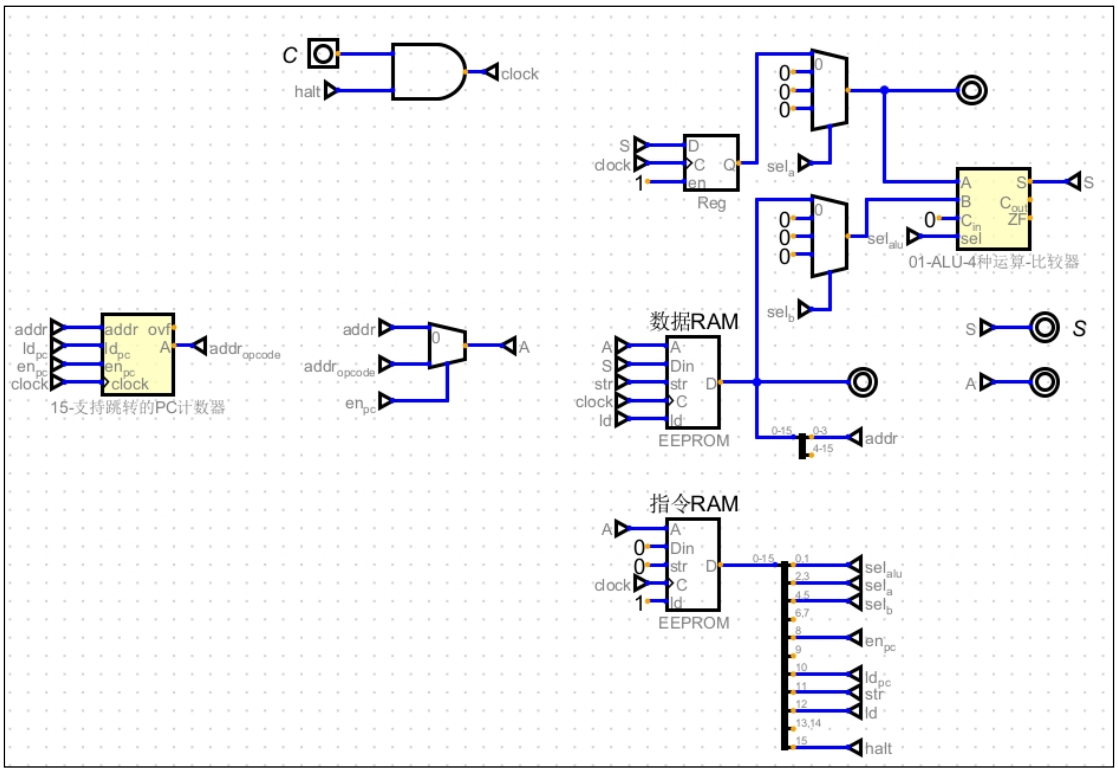
1）PC 计数器增加写入功能以支持跳转





PC 计数器最终有两个控制信号，ld_pc 和 en_pc。当 ld_pc 打开时，addr 的输入会在下个时钟周期被放进寄存器。ld_pc 关闭、en_pc 打开，每个时钟周期寄存器累加 1.en 关闭时，addr 的输出会直接连接到 out，方便地址直通。

2）整体电路如下：



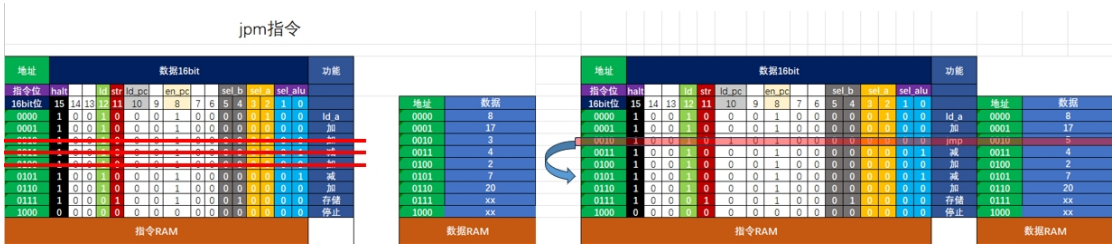
添加 ld_pc 和 en_pc 后的信号控制位及指令如图：

指令信号控制位																
指令位	halt			ld	str	ld_pc		en_pc			sel_b		sel_a		sel_alu	
16bit位	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
最新指令集																
ld_a	1	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0
add	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0
sub	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1
or	1	0	0	1	0	0	0	1	0	0	0	0	0	0	1	1
and	1	0	0	1	0	0	0	1	0	0	0	0	0	0	1	0
str	1	0	0	0	1	0	0	1	0	0	0	1	0	0	x	x
halt	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
jmp	1	x	x	1	0	1	x	1	x	x	x	x	x	x	x	x

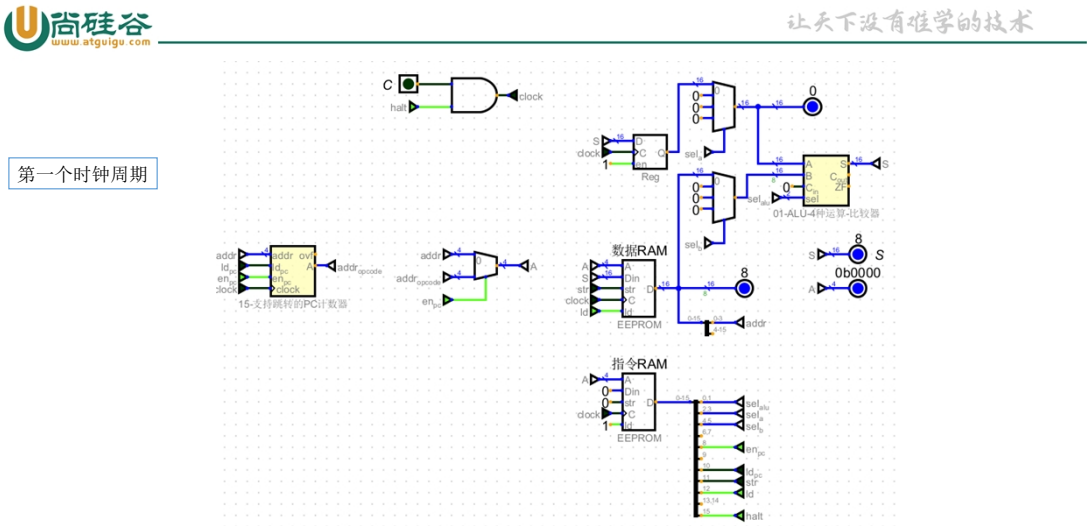
其他操作都需要让程序计数器+1，所以 en_pc=1，ld_pc=0，jmp 是要跳转，因此，en_pc 和 ld_pc 都要是 1。

ld_pc =1，reg 寄存器处于写入状态，并且写入的数据源是外部输入的 addr，en_pc=1 使得当前时钟周期，输入的 addr 不能作为 PC 计数器的输出。而是在下一个时钟周期，en_pc=1，ld_pc=0 时，reg 寄存区输出上一个时钟周期输入的跳转 addr 地址，完成跳转。

接下来我们实现之前的跳转需求，修改数据 RAM 和指令 RAM 的数据如下图：



当前电路及数据下，当执行到第四个时钟周期 jmp 5 时，A 寄存器将跳转地址 5 也进行了累加。如图所示：



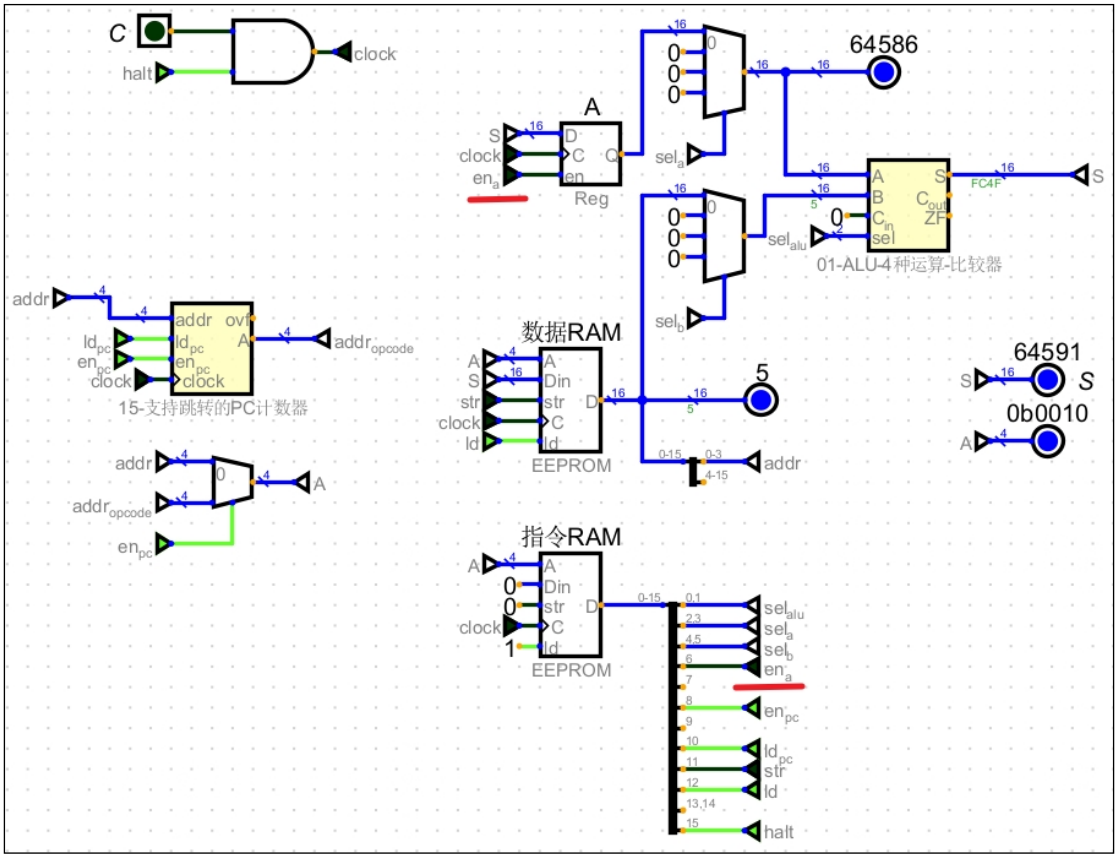
因此，我们需要在跳转时不让数据 RAM 输出的跳转地址进入寄存器 A，因此需要增加一个控制信号 enA，来控制 A 寄存器的输入使能。

添加 enA 指令后的信号控制位及指令如图：

指令信号控制位																
指令位	halt			ld	str	ld_pc		en_pc		en_a	sel_b		sel_a		sel_alu	
16bit位	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
最新指令集																
ld_a	1	0	0	1	0	0	0	1	0	1	0	0	0	1	0	0
add	1	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0
sub	1	0	0	1	0	0	0	1	0	1	0	0	0	0	0	1
or	1	0	0	1	0	0	0	1	0	1	0	0	0	0	1	1
and	1	0	0	1	0	0	0	1	0	1	0	0	0	0	1	0
str	1	0	0	0	1	0	0	1	0	1	0	1	0	0	x	x
halt	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
jmp	1	x	x	1	0	1	x	1	x	0	x	x	x	x	x	x

当执行 jmp 跳转的时候，enA=0,其他情况 enA=1。

修改电路后如下图：

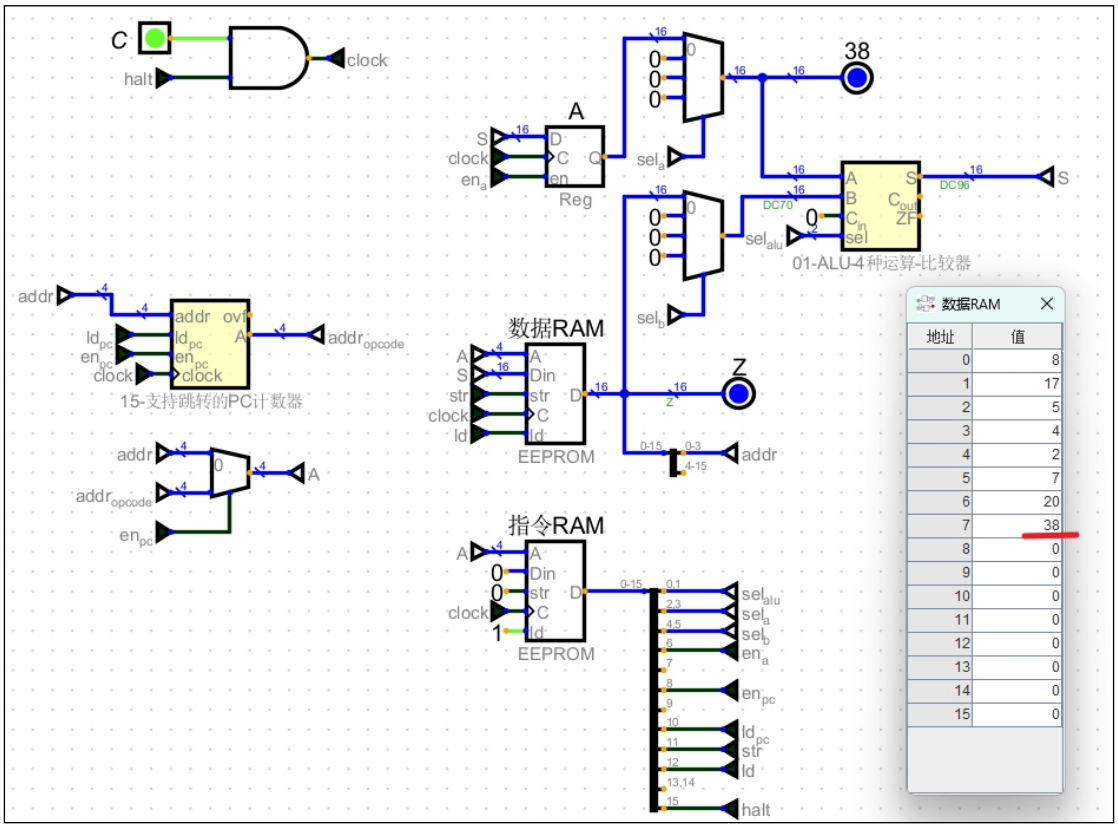


根据最新电路图修改指令 RAM 中的数据为：

jmp_enA指令

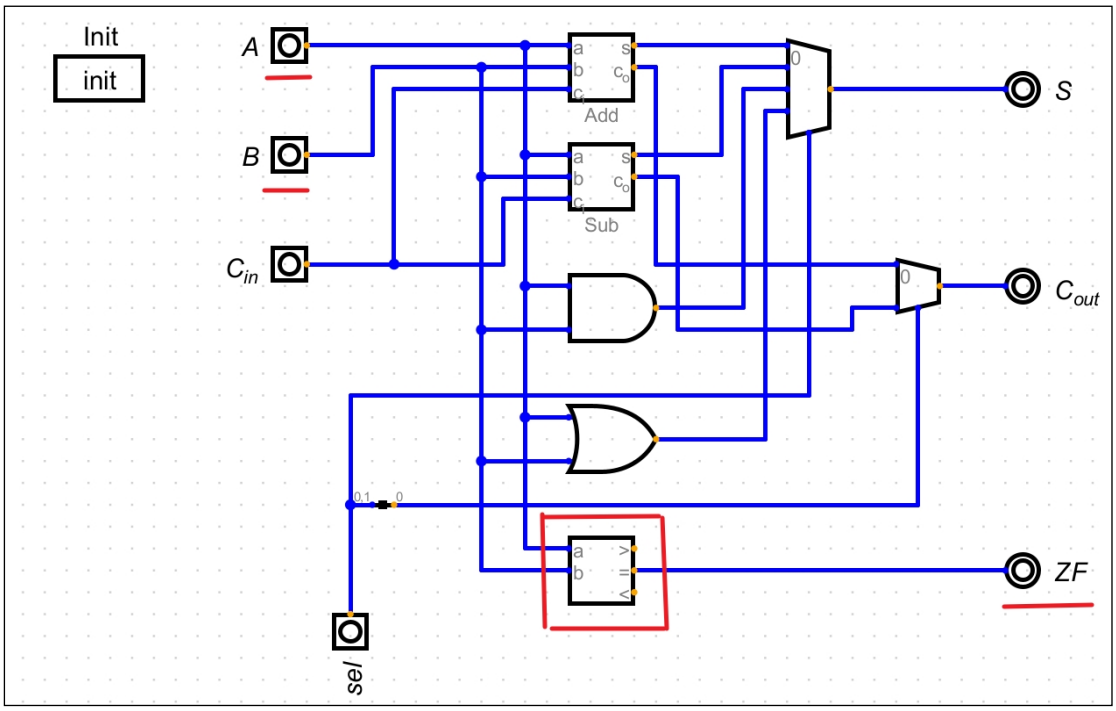
地址	数据16bit															功能		
指令位	halt			ld	str	ld_pc		en_pc		en_a	sel_b		sel_a		sel_alu			
16bit位	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		地址	数据
0000	1	0	0	1	0	0	0	1	0	1	0	0	0	1	0	0	ld_a	0000	8
0001	1	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0	加	0001	17
0010	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	jmp	0010	5
0011	1	0	0	1	0	0	0	1	0	1	0	0	0	0	0	1	减	0011	4
0100	1	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0	加	0100	2
0101	1	0	0	1	0	0	0	1	0	1	0	0	0	0	0	1	减	0101	7
0110	1	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0	加	0110	20
0111	1	0	0	0	1	0	0	1	0	1	0	1	0	0	0	0	存储	0111	xx
1000	0	0	0	0	0	0	0	0	0	x	0	0	0	0	0	0	停止	1000	xx
指令RAM																		数据RAM	

开启仿真执行，8+17+3-4+2-7+20，最终执行的算式为：8+17-7+20 并将结果 38 存储到地址 7 的位置。如图所示：

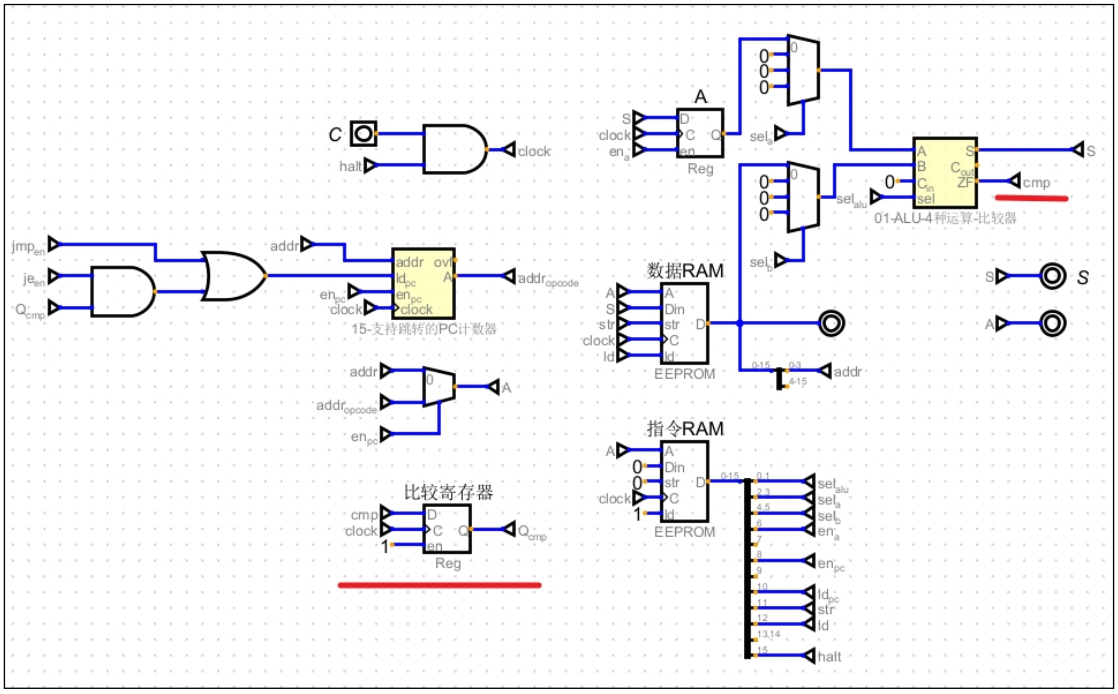


3.5.6 添加 je 条件跳转信号

jmp 跳转是无条件的跳转，程序流程控制也需要有条件的跳转，什么是有条件的跳转呢？比如：如果年龄=18，可以参加活动！所以，最常见的条件就是比较两个数的大小。如果相等就执行相应的操作。我们的 ALU 在设计阶段就已经实现了比较的功能。如图：



ALU 可以比较 A、B 的输入，如果相等则 ZF=1，如果不相等则 ZF=0。接下来我们就可以通过 ZF 是否=1 来作为跳转的依据了。我们可以引入 je（jump equal 的缩写）指令。执行这条指令时，如果上一个时钟周期 ZF=1，那么跳转到指定地址继续执行。如果 ZF=0，那么不跳转。因此，我们需要一个寄存器来存储上一个时钟周期 ZF 比较的结果。该寄存器我们命名为比较寄存器。加入比较寄存器后的电路图如下：



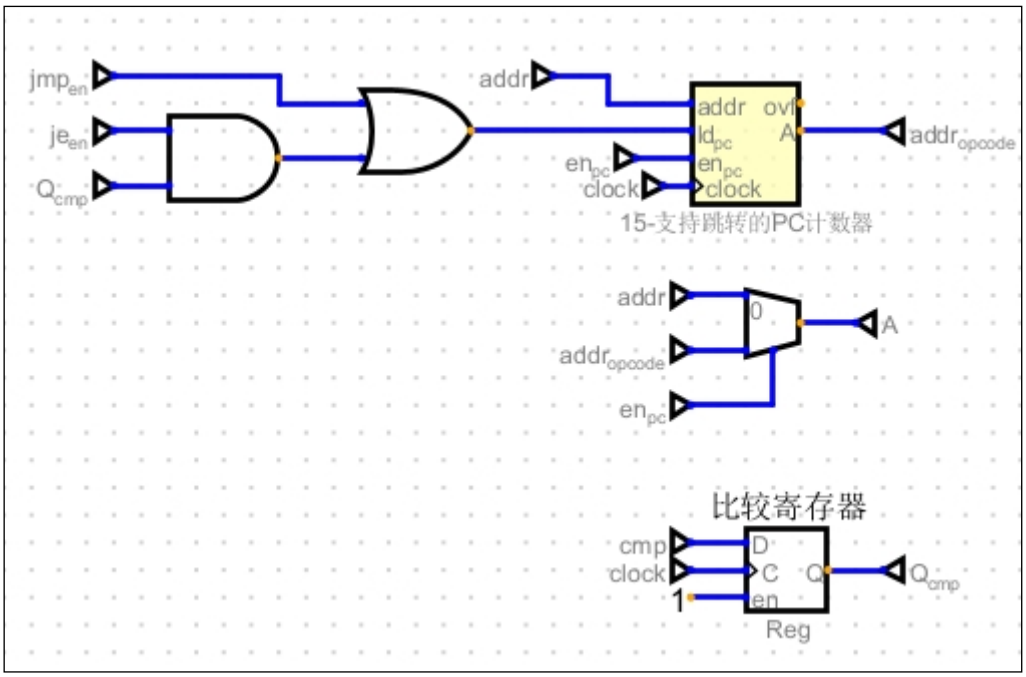
- ALU 增加 ZF 输出管道 cmp。
- 增加比较寄存器。

之前我们 jmp 指令的时候，使用了两个控制信号对跳转进行控制，ld_pc 和 en_pc，跳转时 ld_pc=1，en_pc=1，PC 累加时 ld_pc=0，en_pc=1，可以发现控制跳转的主要信号是 ld_pc，因为 en_pc 始终=1。我们又加入了新跳转 je，所以将原来 jmp 的跳转信号 ld_pc 重新命名为 jmp_en，je 跳转信号命名为 je_en。

jmp_en=1 时，进行无条件的跳转。

je_en=1，并且 Qcmp = 1 时，进行有条件的跳转。

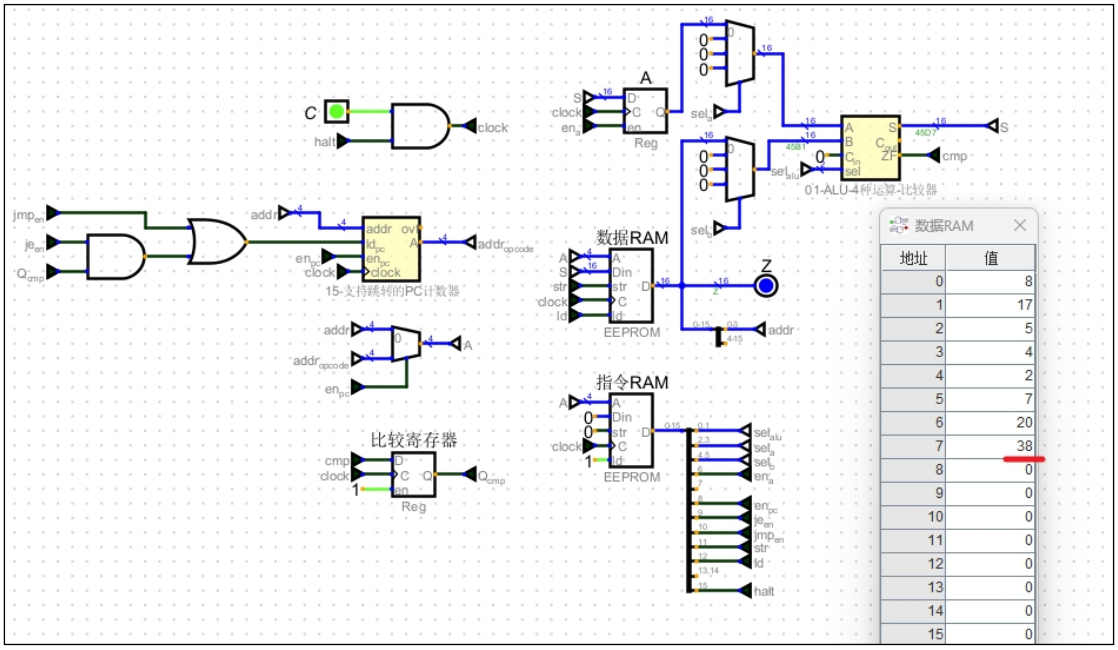
根据该逻辑 PC 计数器的电路应该修改为如图所示：



增加 je_en、jmp_en 后的指令控制位及最新指令如图：

指令信号控制位																
指令位	halt			ld	str	jmp_en	je_en	en_pc		en_a	sel_b		sel_a		sel_alu	
16bit位	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
最新指令集																
ld_a	1	0	0	1	0	0	0	1	0	1	0	0	0	1	0	0
add	1	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0
sub	1	0	0	1	0	0	0	1	0	1	0	0	0	0	0	1
or	1	0	0	1	0	0	0	1	0	1	0	0	0	0	1	1
and	1	0	0	1	0	0	0	1	0	1	0	0	0	0	1	0
str	1	0	0	0	1	0	0	1	0	1	0	1	0	0	x	x
halt	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
jmp	1	x	x	1	0	1	0	1	x	0	x	x	x	x	x	x
je	1	x	x	1	0	0	1	1	x	0	x	x	x	x	x	x

修改后，首先测试一下原来的 jmp 无条件跳转是否仍然有效。数据 RAM 和指令 RAM 的数据不变。

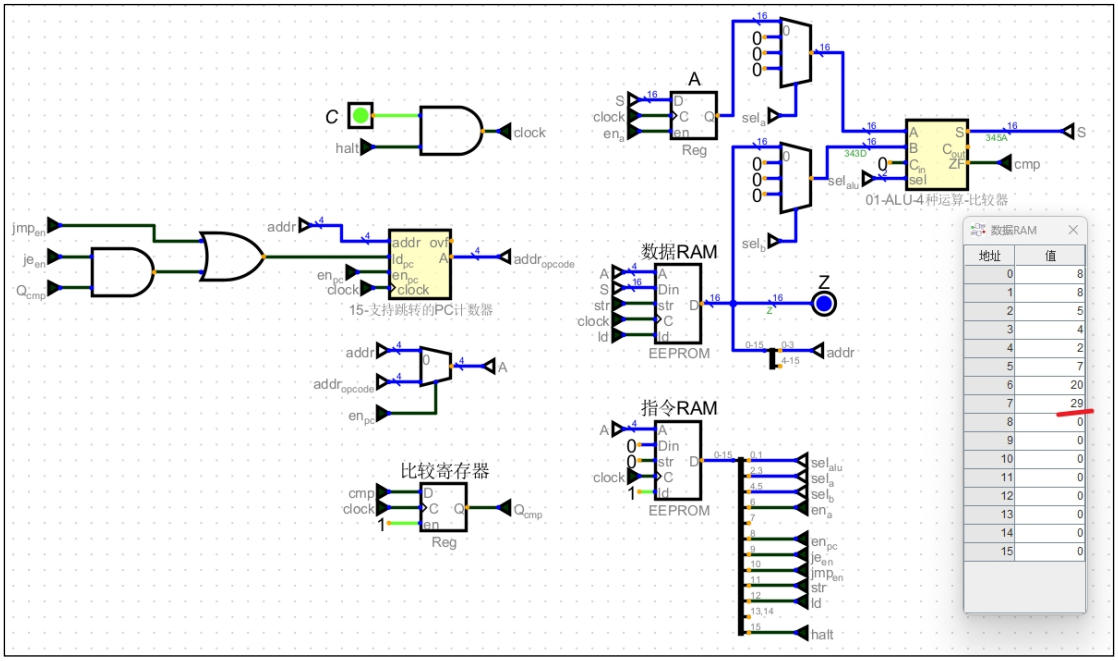


接下来测试 je 跳转，将数据 17 修改为 8，这样前两条数据都是 8 满足 Qcmp=1 的条件。将第三条指令修改为 je 5。具体数据 RAM 和指令 RAM 数据如图：

地址	数据16bit															功能
指令位	halt			ld	str	jmp_en	je_en	en_pc		en_a	sel_b		sel_a		sel_alu		
16bit位	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0000	1	0	0	1	0	0	0	1	0	1	0	0	0	1	0	0	ld_a
0001	1	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0	加
0010	1	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	je
0011	1	0	0	1	0	0	0	1	0	1	0	0	0	0	0	1	减
0100	1	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0	加
0101	1	0	0	1	0	0	0	1	0	1	0	0	0	0	0	1	减
0110	1	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0	加
0111	1	0	0	0	1	0	0	1	0	1	0	1	0	0	0	0	存储
1000	0	0	0	0	0	0	0	0	0	x	0	0	0	0	0	0	停止
指令RAM																	

地址	数据
0000	8
0001	8
0010	5
0011	4
0100	2
0101	7
0110	20
0111	xx
1000	xx
数据RAM	

8+8-7+20，结果=29 并存储到 7 号地址位就正确了。运行结果如下：



3.6 添加控制器

通过前面的学习，我们应该感受到了指令的本质就是控制信号的开关。

当前我们的控制信号如图所示：

指令信号控制位															
指令位	halt			ld	str	jmp_en	je_en	en_pc		en_a	sel_b		sel_a		sel_alu	
16bit位	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

0、1：控制 ALU 的计算选择（加、减、与、或）。

2、3：ALU 的 A 输入选择信号。

4、5：ALU 的 B 输入选择信号。

6：寄存器 A 的写使能信号。

8： PC 计数器的写使能信号。

9： je 条件跳转控制信号。

10：jmp 无条件跳转控制信号。

11：数据 RAM 写使能信号。

12：数据 RAM 读使能信号。

15：终止信号。

现在已经有 11 位的信号控制了，如果电路越来越复杂，控制信号会越来越多。每一个控制信号都需要一个 2 进制位来表示它，现代计算机的指令多达几百上千个，那我们指令

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

占用的位就太多了，我们必须用一种方式简化指令占用的控制位。

我们知道一个二进制位表示 2 个数字，目前我们的系统一共有如下 9 条指令，那么只需要 2^4=16，也即 4 个二进制位就可以表示。

指令信号控制位																
指令位	halt			ld	str	jmp_en	je_en	en_pc		en_a	sel_b		sel_a		sel_alu	
16bit位	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
最新指令集																
ld_a	1	0	0	1	0	0	0	1	0	1	0	0	0	1	0	0
add	1	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0
sub	1	0	0	1	0	0	0	1	0	1	0	0	0	0	0	1
or	1	0	0	1	0	0	0	1	0	1	0	0	0	0	1	1
and	1	0	0	1	0	0	0	1	0	1	0	0	0	0	1	0
str	1	0	0	0	1	0	0	1	0	1	0	1	0	0	x	x
halt	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
jmp	1	x	x	1	0	1	0	1	x	0	x	x	x	x	x	x
je	1	x	x	1	0	0	1	1	x	0	x	x	x	x	x	x

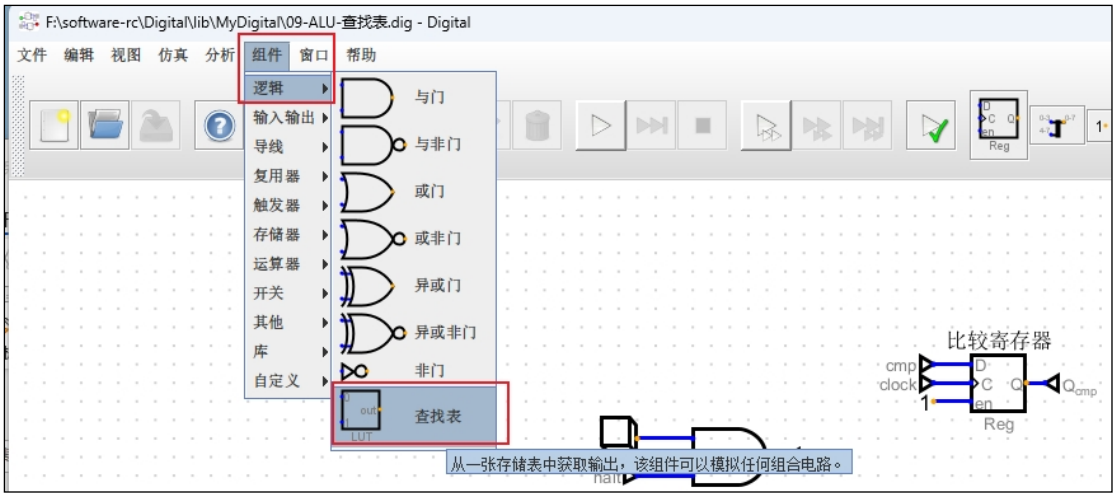
为了后期进一步扩展完善系统，我们选择 2^5，32 条指令的空间留有扩展的余地。5 个二进制位在这里我们称为操作码，英文缩写 opcode。

假如我们把每一条指令和一个 5 位 opcode 按照如下方式映射起来：

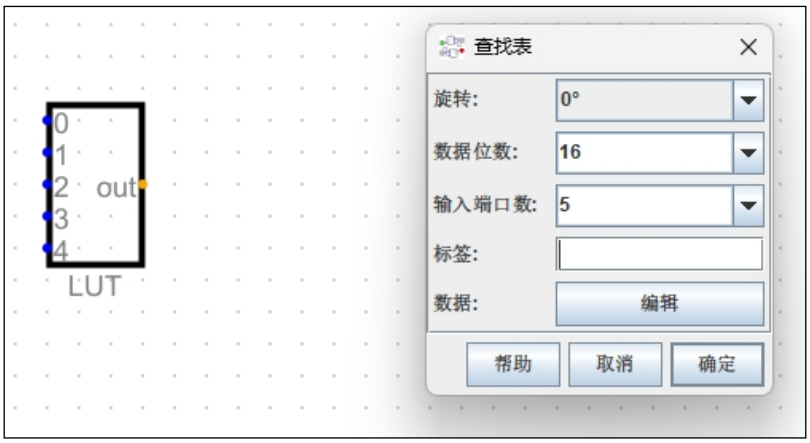
		5位的opcode信号控制进行映射															
opcode	opname	halt	x	x	ld	str	jmp_en	je_en	en_pc	x	en_a	sel_b	sel_a	sel_alu	二进制开关	指令寄存器二进制	opname
00000	halt	0	0	0	0	0	0	0	0	0	0	00	00	00	0000000000000000	0b0000000000000000	halt
00001	ld_a	1	0	0	1	0	0	0	1	0	1	00	01	00	1001000101000100	0b1001000101000100	ld_a
00010	add	1	0	0	1	0	0	0	1	0	1	00	00	00	1001000101000000	0b1001000101000000	add
00011	sub	1	0	0	1	0	0	0	1	0	1	00	00	01	1001000101000001	0b1001000101000001	sub
00100	or	1	0	0	1	0	0	0	1	0	1	00	00	11	1001000101000011	0b1001000101000011	or
00101	and	1	0	0	1	0	0	0	1	0	1	00	00	10	1001000101000010	0b1001000101000010	and
00110	str	1	0	0	0	1	0	0	1	0	1	01	00	00	1000100101010000	0b1000100101010000	str
00111	jmp	1	0	0	1	0	1	0	1	0	0	00	00	00	1001010100000000	0b1001010100000000	jmp
01000	je	1	0	0	1	0	0	1	1	0	0	00	00	00	1001001100000000	0b1001001100000000	je

也就是当 opcode=00000 时，表示指令 halt，当 opcode=00001 时，表示指令 ld_a，以此类推。这种映射有了之后，如何让我们的系统支持该映射呢？可以通过查找表来实现这种映射。

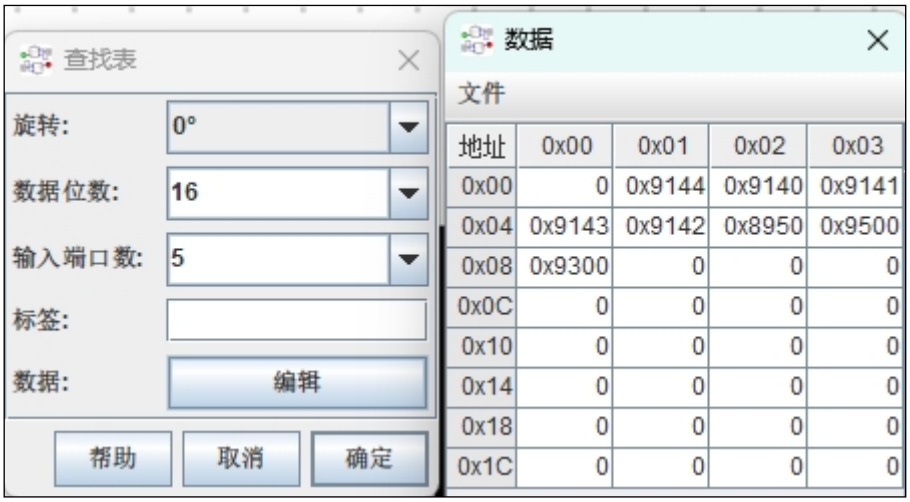
新建一个电路图，取名控制器，然后在电路图中添加查找表，如下图所示：



查找表设置输入位为 5 位（与 opcode 一致），数据位 16 位。

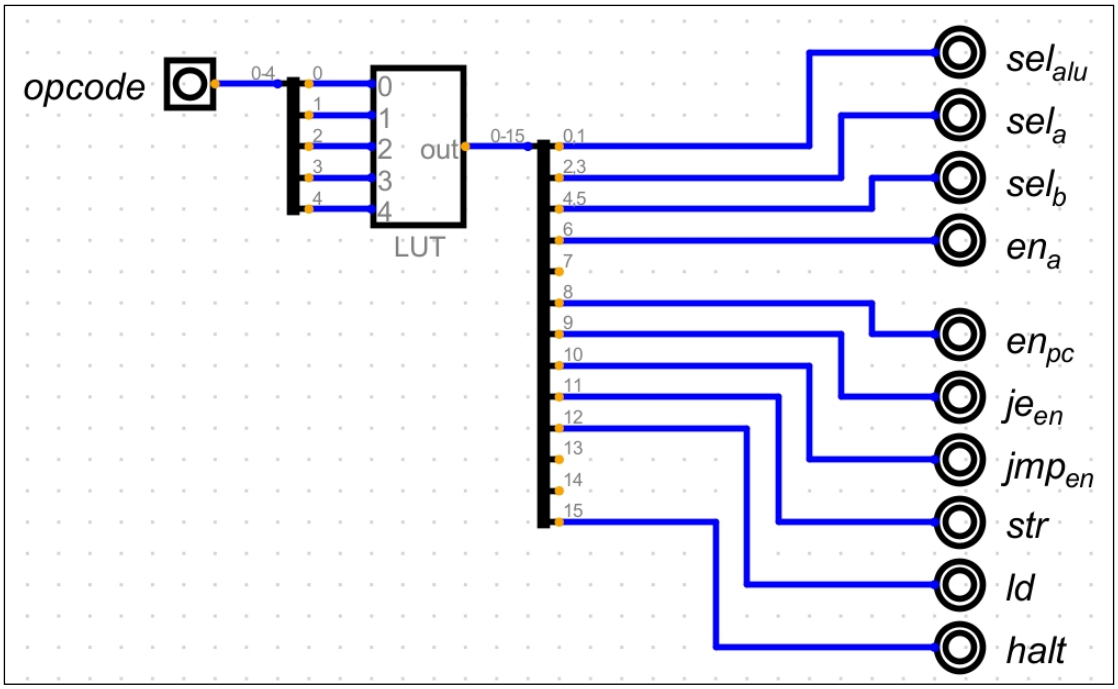


点击编辑，输入映射关系数据（查找表默认显示的 16 进制数据）。



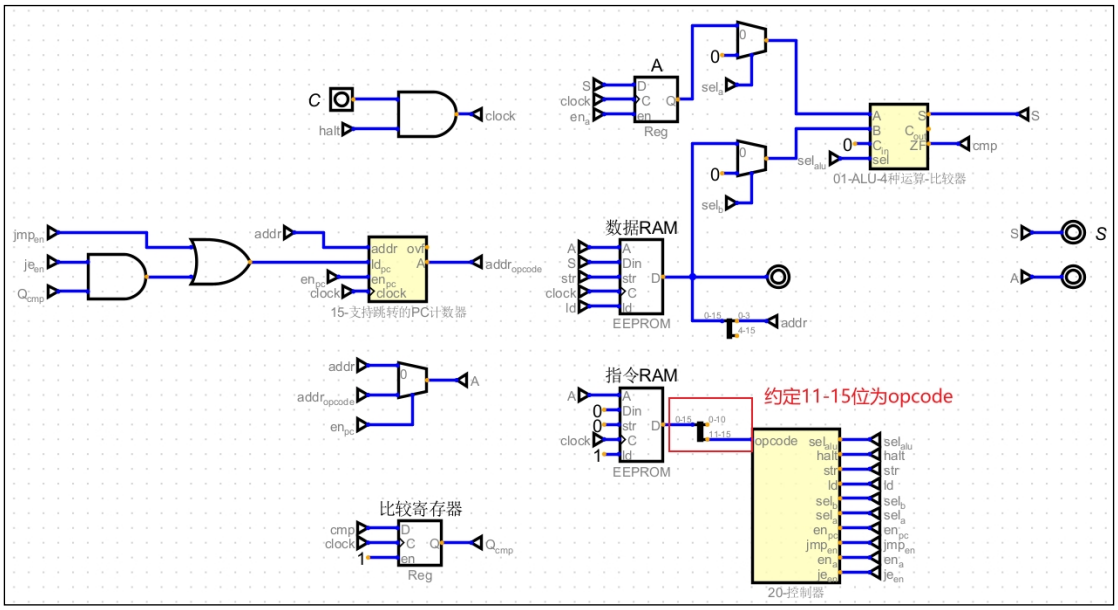
地址	0x00	0x01	0x02	0x03
0x00	0	0x9144	0x9140	0x9141
0x04	0x9143	0x9142	0x8950	0x9500
0x08	0x9300	0	0	0
0x0C	0	0	0	0
0x10	0	0	0	0
0x14	0	0	0	0
0x18	0	0	0	0
0x1C	0	0	0	0

完善控制器电路如图：



以上控制器电路，就实现了通过 5 位的 opcode，映射出对应指令的控制信号。

有了控制器之后，我们的指令寄存器中存放的数据就修改为 opcode 即可。修改电路如下图：



指令寄存器中存放 opcode（11-15），通过 ALU 控制器中的查找表映射出控制信号。

我们来测试一下加入控制器后的电路，计算 29+38-5-3，并将运算的结果存放到数据 RAM 的地址 5 的位置。

数据 RAM 和指令 RAM 的数据如下：

红色表示opcode

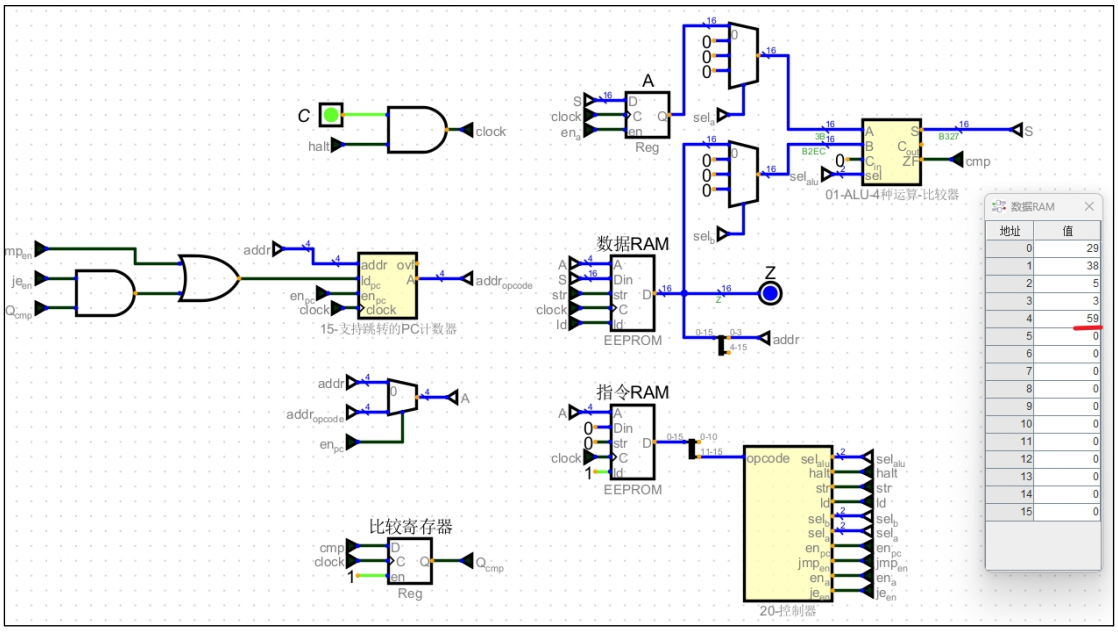
地址	数据
000	0000100000000000（ld a）
001	0001000000000000（加）
010	0001100000000000（减）
011	0001100000000000（减）
100	0011000000000000（str）
101	0000000000000000（halt）

指令RAM

地址	数据
000	29
001	38
010	5
011	3
100	xx
101	xx

数据RAM

仿真执行后结果如图：



3.7 将两块内存合二为一

3.7.1 现有系统运行方式梳理

在合并改造之前，我们先回顾一下我们现在的系统。

（1）开始我们有一个 PC 程序计数器，程序计数器的输出是一个 4 位的地址。

（2）该 4 位的地址既是我们指令 RAM 的地址，用来获取指令 RAM 中存放的 opcode，也是数据 RAM 的读取地址，用来读取数据 RAM 中存放的数据。

（3）接下来我们的 opcode 通过控制器中的查找表，会映射出整个电路的信号开关控

制情况，进而实现相应的功能。

（4）不同的指令会对数据 RAM 中读取的数据进行不同的操作。

- ld_a：将数据 RAM 中的数据加载到寄存器 A 中。
- add：将数据 RAM 中的数据累加到寄存器 A 中。
- sub：将寄存器 A 中的数据减掉数据 RAM 中的数据，并存储到寄存器 A 中。
- str：将 ALU 的计算结果存储到数据 RAM 的指定地址。
- 等等。。。。。

（5）数据 RAM 的输出有两个作用。

- 作为数据输出。
- 当执行 jmp、je 跳转的时候，作为要跳转的地址。

ALU 进行运算，A 输入数据来源于 A 寄存器或 0，通过 selA 信号控制，B 输入数据来源于数据 ROM 或 0，通过 selB 控制。

（6）比较寄存器存储 A、B 输入比较的结果，如果相同存储 1，不同存储 0，该比较结果用来做为 je 跳转的依据。以上就是我们当前系统的相关功能。

3.7.2 现有系统的指令

opname	opcode	操作
halt	00000	停止时钟
ld_a	00001	将数据 RAM 的数据载入寄存器 A
add	00010	数据 RAM 的数据+寄存器 A 再存入寄存器 A
sub	00011	寄存器 A-数据 RAM 再存入寄存器 A
or	00100	寄存器 A||数据 RAM 再存入寄存器 A
and	00101	寄存器 A&&数据 RAM 再存入寄存器 A
str	00110	将寄存器 A 中数据存入数据 RAM
jmp	00111	无条件跳转到指定地址
je	01000	比较相等时跳转到指定地址

3.7.3 操作数和操作码

将两个 RAM 合并为一个 RAM 之前，我们先介绍一种数据存储的方式，操作码+操作数。

现在我们的系统计算一个算式：13+45-27 时，数据 RAM 和指令 RAM 中的数据存储情

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

况如下：

地址	数据（操作码）	操作数
000	ld_a	000
001	add	001
010	sub	010
011	str	011

指令RAM

地址	数据
000	13
001	45
010	27
011	xx

数据RAM

各个指令执行的操作：

PC 计数器为 000 时，指令 RAM 中地址为 000 的指令 ld_a 会将数据 RAM 中地址为 000 的数据 13 加载到寄存器 A 中。

PC 计数器为 001 时，指令寄存器中地址为 001 的指令 add 会将数据 RAM 中地址为 001 的数据 45 累加到寄存器 A 中。

PC 计数器为 010 时，指令寄存器中地址为 010 的指令 sub 会将数据 RAM 中地址为 010 的数据 27，与寄存器 A 中的数据进行相减操作存入寄存器 A。

PC 计数器为 011 时，指令寄存器中地址为 011 的指令 str，进行存储操作，会将上面计算的结果 31，存入数据 RAM 中地址为 011 的位置。

从上面的例子可以看出，指令的地址是什么，那么该条指令操作（ld_a、add、sub、str…..）的数据的地址也要是什么。

这样的操作过于僵化，而且数据 RAM 的空间也不能得到重复的利用。我们希望指令操作的数据 RAM 的地址可以被手动指定。因此，引入操作数的概念，如何理解操作数呢？当前我们可以认为，操作数就是操作码要操作的数据 RAM 的地址。

引入操作数概念后，我们上面的计算 13+45-27，在指令 RAM 和数据 RAM 中的存储如下：

地址	数据（操作码）	操作数
000	ld_a	000
001	add	001
010	sub	010
011	str	011

指令RAM

地址	数据
000	13
001	45
010	27
011	xx

数据RAM

ld_a 000 表示，将数据 RAM 中的地址为 000 的数据 13 加载到寄存器 A 中。

add 001 表示，将数据 RAM 中的地址为 001 的数据 45 累加到寄存器 A 中。

sub 010 表示，将寄存器 A 中的数据减掉数据 RAM 中的地址为 010 的数据 27，再回存到寄存器 A 中。

str 011 表示，将寄存器 A 中的数据存储到数据 RAM 的 011 位置。

会发现加入了操作数的指令跟没加操作数没有区别啊！反而更麻烦了。但是如果我们将操作数改变一下，就会发现它的优势。

地址	数据（操作码）	操作数
000	ld_a	000
001	add	001
010	sub	010
011	str	011

指令RAM

地址	数据
000	13
001	45
010	27
011	xx

数据RAM

ld_a 001 表示， 将数据 RAM 中地址为 001 的 45 加载到寄存器 A 中。

add 010 表示，将数据 RAM 中的地址为 010 的数据 27 累加到寄存器 A 中。

sub 000 表示，将寄存器 A 中的数据减掉数据 RAM 中的地址为 000 的数据 13，再回存到寄存器 A 中。

str 000 表示，将寄存器 A 中的数据存储到数据 RAM 的 000 位置。

我们会发现，通过操作码和操作的数的结合，使得我们的指令操作更加的灵活，并且使得数据 RAM 还可以被重复的利用。

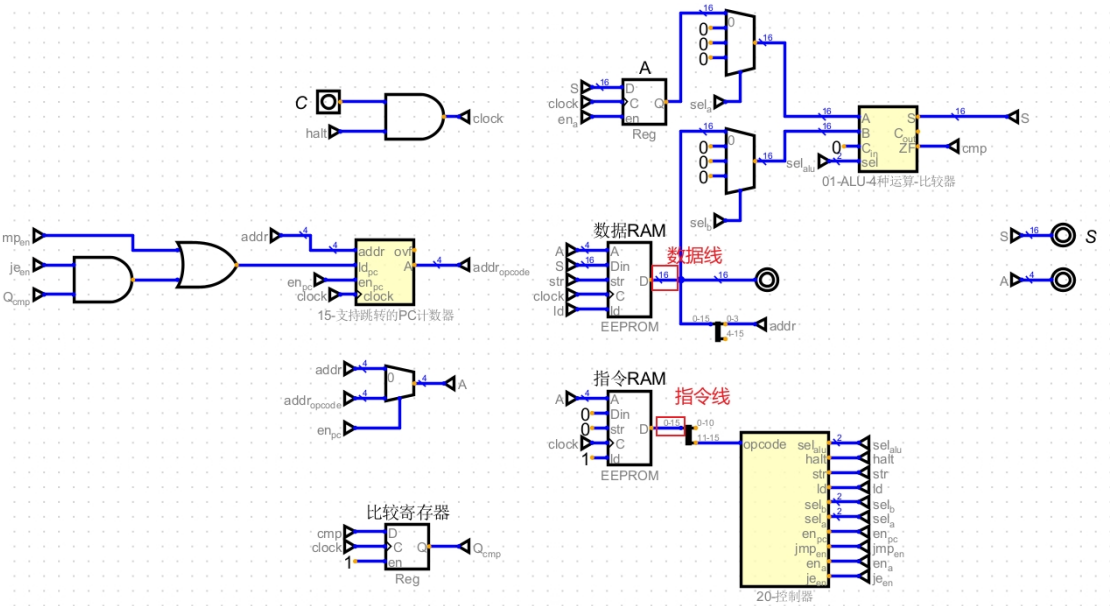
我们合并成一个 RAM 之后，RAM 中存储指令的方式就是操作码+操作数的方式。这里介绍了这种方式，方便后面的理解。那么接下来就让我们来合并 RAM 吧。

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

3.7.4 合并为一个 RAM

目前我们的数据存储和指令存储是分开的，这和我们使用计算机的直觉不符，毕竟我们都是用一块内存完成编程。这是因为现在主流计算机都是冯·诺依曼架构，只有一块内存，所以接下来我们就将两块内存合并成一块。

在原来的系统中，我们有两块 RAM，一块数据 RAM，一块指令 RAM，在一个时钟周期内，我们有一根数据线从数据 RAM 中读取数据，有一根指令线，从指令 RAM 中读取指令。如图：



如果我们将两块 RAM 合并为一块，那么数据线和指令线就变成了一条线，那么在某个时钟周期内，该怎么知道这条线中传递的是数据地址还是指令地址呢？因为在一个时钟周期内不可能既表示数据又表示指令。那该怎么办？

3.7.5 取指令和执行指令

既然一个时钟周期做不到，那么我们就将程序的执行分成两个时钟周期来完成一个任务。第一个时钟周期用于读取指令。第二个时钟周期用于读取数据并执行指令。

取址阶段：在这个阶段，RAM 的地址信号是指令的地址。RAM 输出的是指令。

执行阶段：在这个阶段，RAM 的地址信号是数据的地址。RAM 输出的是数据。也是在这个阶段去执行上一个时钟周期读取到的指令。因为执行指令阶段需要获取到上一个时钟周期的指令，因此，需要有一个指令寄存器去存储上一个时钟周期获取的指令。

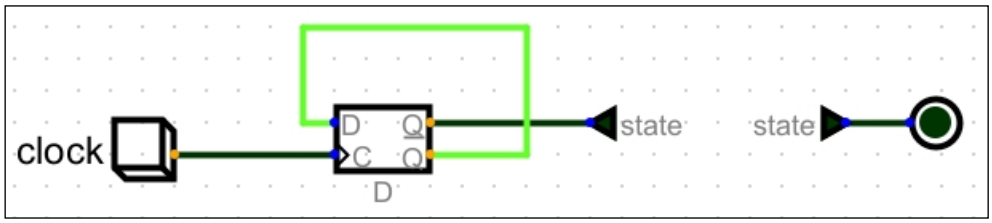
时钟周期	阶段	状态-state

0	取指令	0
1	执行指令	1
2	取指令	0
3	执行指令	1

程序的执行过程如上图所示：

我们程序的执行状态只有两种，因此，我们可以约定 state=0 时为取指阶段，state=1 时为执行指令阶段。

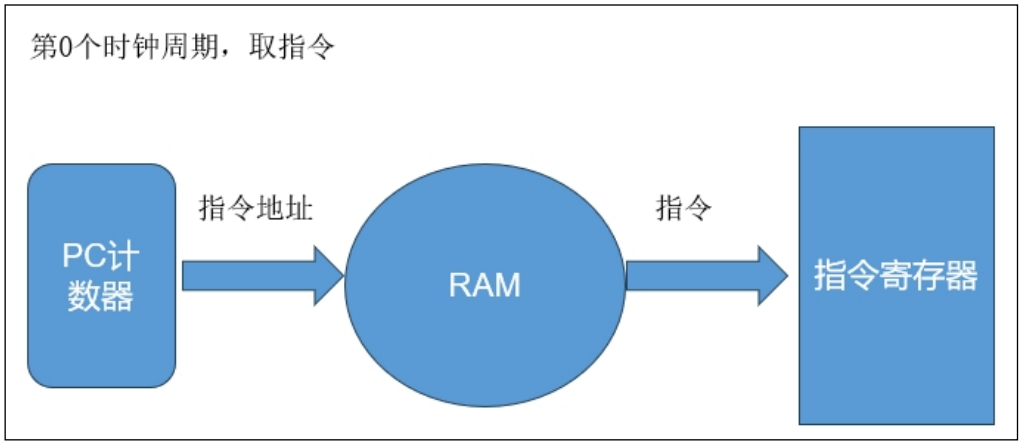
State 只能在 0，1 之间切换。可以用 D 触发器来实现 state 的电路切换。如下图：

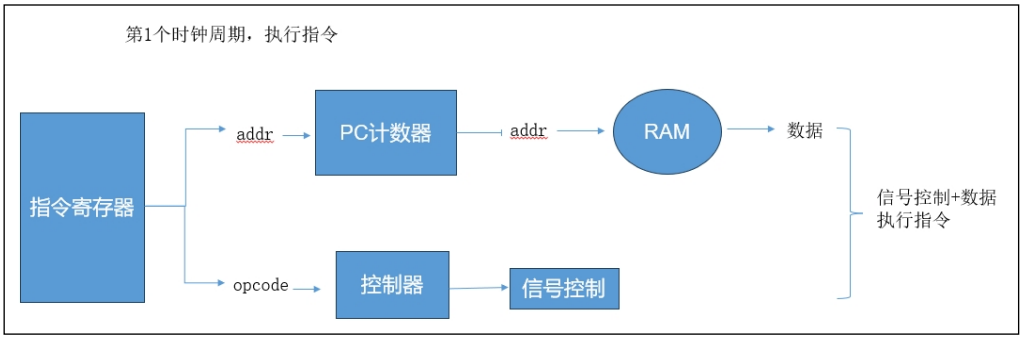


3.7.6 电路实现

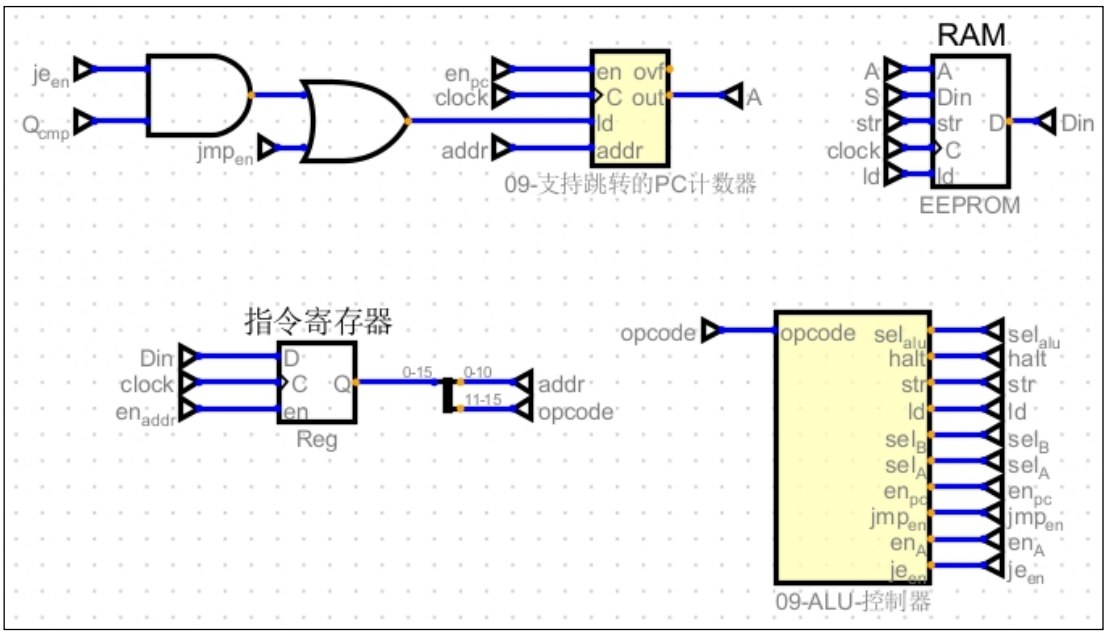
我们之前是一个时钟周期执行一条指令，现在是两个时钟周期执行一条指令，分为取指令（取指令就是根据 PC 计数器输出的地址从 RAM 中读取指令并存储到指令寄存器中），执行指令（读取指令寄存器中的指令，以该指令作为输入，通过控制单元产生各种控制信号，使得各个器件执行不同的功能）。

我们需要两个时钟周期执行一条指令，那指令在各个器件间是如何传输执行的呢？





3.7.6.1 创建 RAM、指令寄存器



enAddr 是指令寄存器的写允许控制信号。取址阶段 enAddr=1，执行阶段 enAddr=1，所以，state 和 enAddr 的映射关系也应该在控制器查找表中做映射。关于查找表的映射配置，我们在所有电路完成之后一起配置。

3.7.6.2 RAM 的数据结构

将数据 RAM 和地址 RAM 合并成一块 RAM，那么就要求新的 RAM 中既有指令又有数据。此处以 4 条指令和 4 条数据为例。我们将前 4 条指令储存在内存的前 4 个地址，将 4 条数据存储在内存的后 4 个地址。要计算 19+69-27 的结果，并要将结果存放在地址 7 的位置。

地址	指令	opcode（操作码）	addr（操作数）	新代码 / 数据
000	ld_a	0b00001	0b100	0b0000100000000100
001	add	0b00010	0b101	0b0001000000000101
010	sub	0b00011	0b110	0b0001100000000110
011	str	0b00110	0b111	0b0011000000000111

100				0b0000000000010011
101				0b0000000001000101
110				0b0000000000011011
111				

蓝色是操作数，彩色的部分是操作码，此处操作码是数据存放的地址。

3.7.6.3 修改 PC 计数器的地址位为 11 位

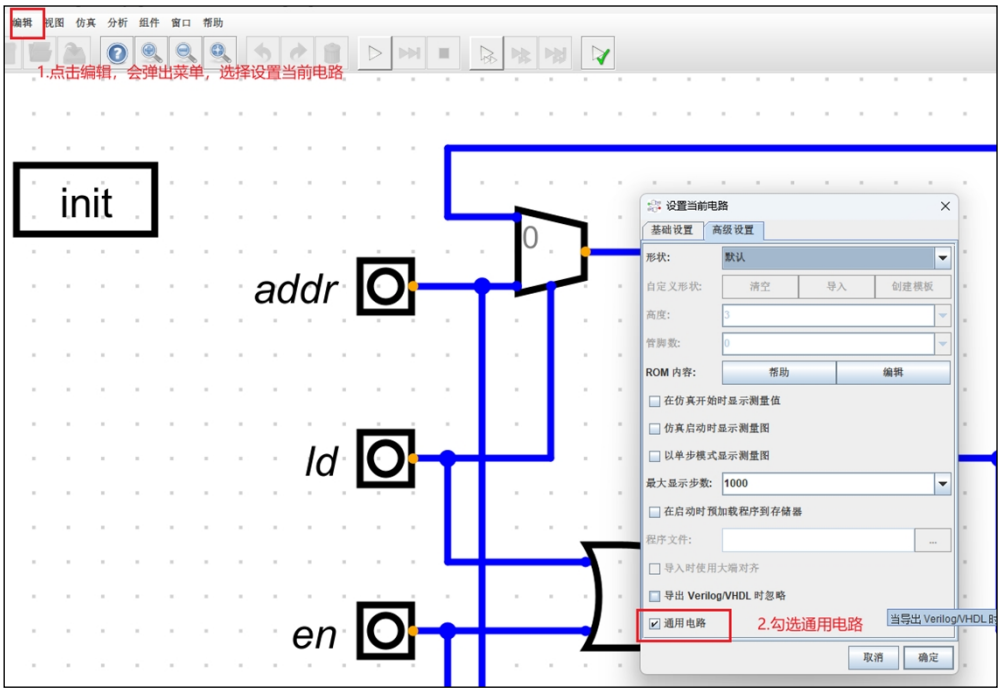
指令寄存器输出的内容为 opcode 和 addr，addr 是 11 位的。addr 又会作为 PC 计数器的输入，我们的 PC 计数器原来的地址位是 4 位的，因此要将 PC 计数器整体修改为 11 位。

同时，PC 计数器输出的 A 地址也会变为 11 位，A 地址又作为 RAM 的地址输入，因此 RAM 的地址位也要改为 11 位。

如果改为 11 位之后，前面用到 PC 计数器的电路都是 4 位的将不能使用。因此，我们需要将 PC 计数器做成通用组件。

（1）选择设置通用组件

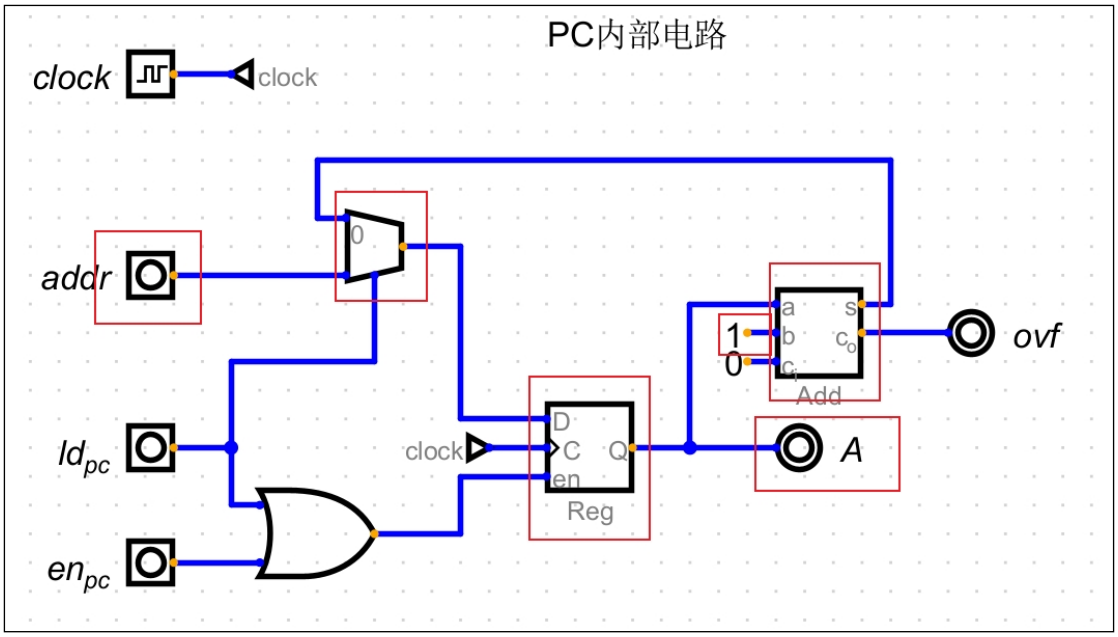
打开 PC 电路→『顶部菜单--编辑』→『设置当前电路』→『高级设置』→『勾选通用电路』。



（2）给器件添加通用参数

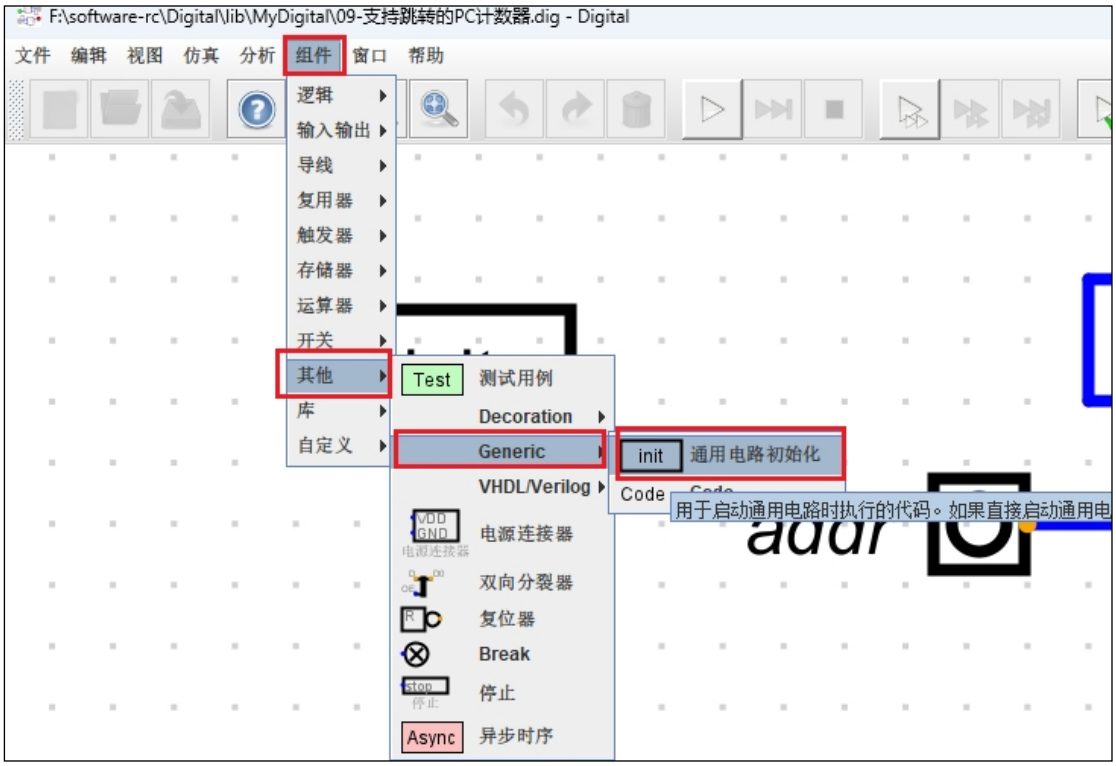
通用参数：this.Bits=args.Bits。

PC 计数器需要进行如上修改的器件如下图：



（3）添加通用电路初始化器件

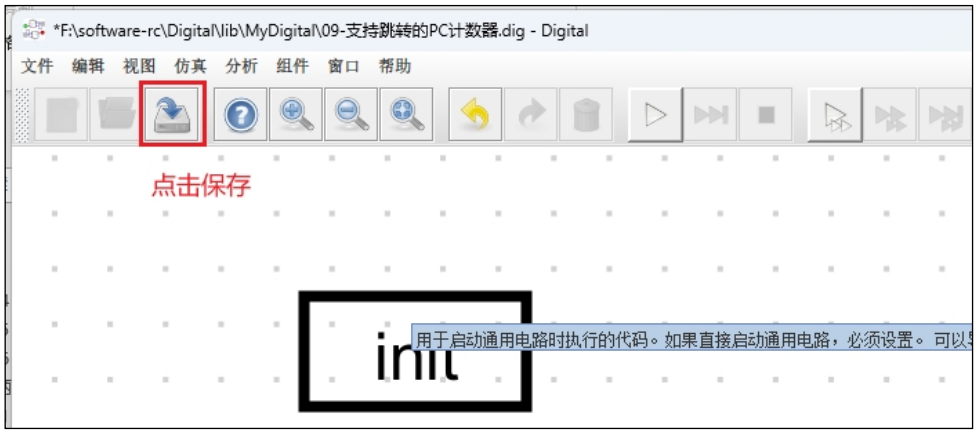
『组件』→『其他』→『Generic』→『init 通用电路初始化』。



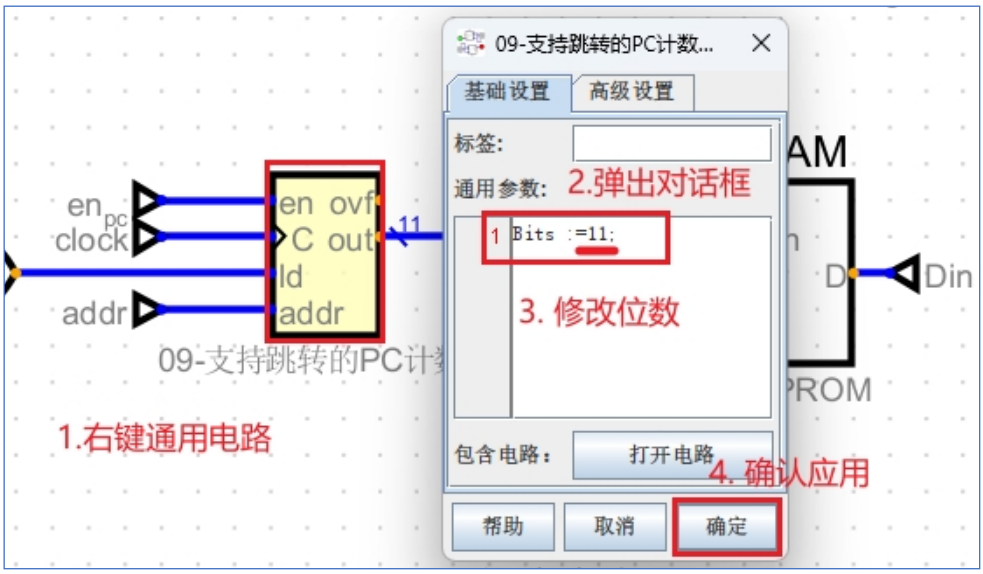
（4）设置通用电路初始化参数

通用参数：Bits := 11。

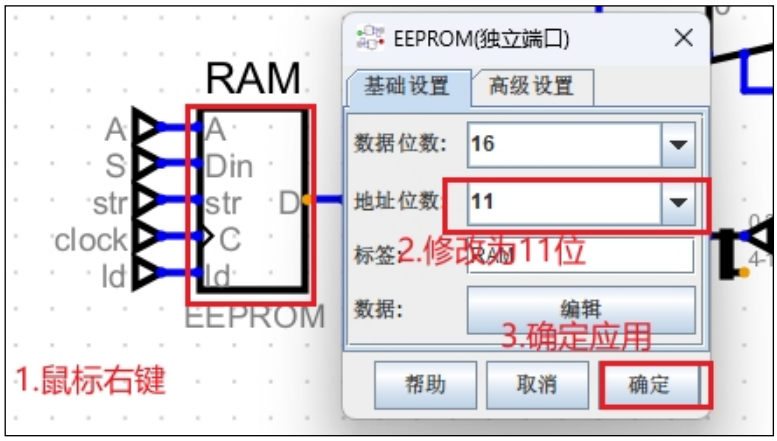
（5）保存通用设置



（6）使用通用电路设置位数



3.7.6.4 修改 RAM 的地址位为 11 位

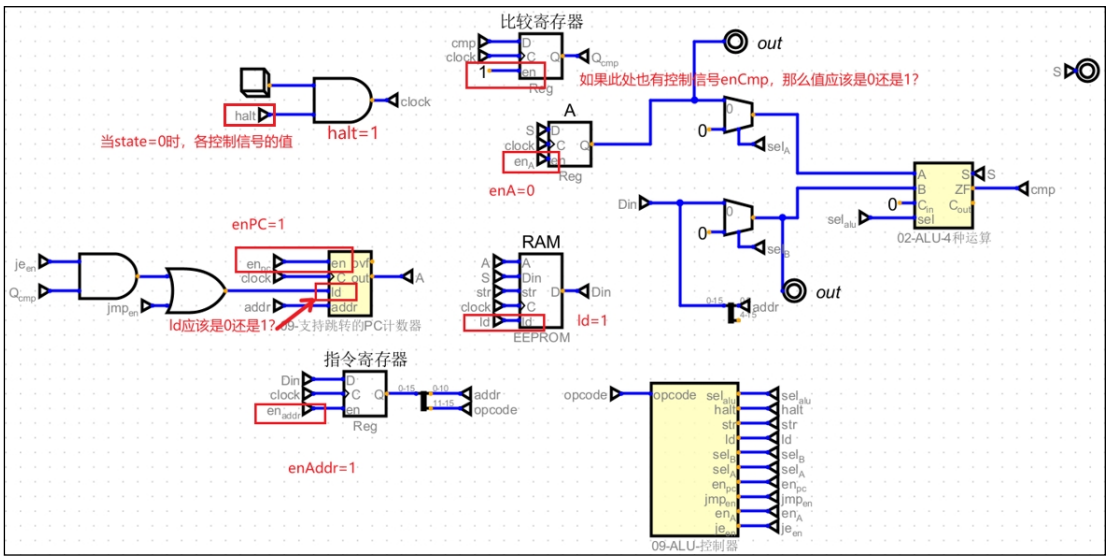


3.7.6.5 控制器改造

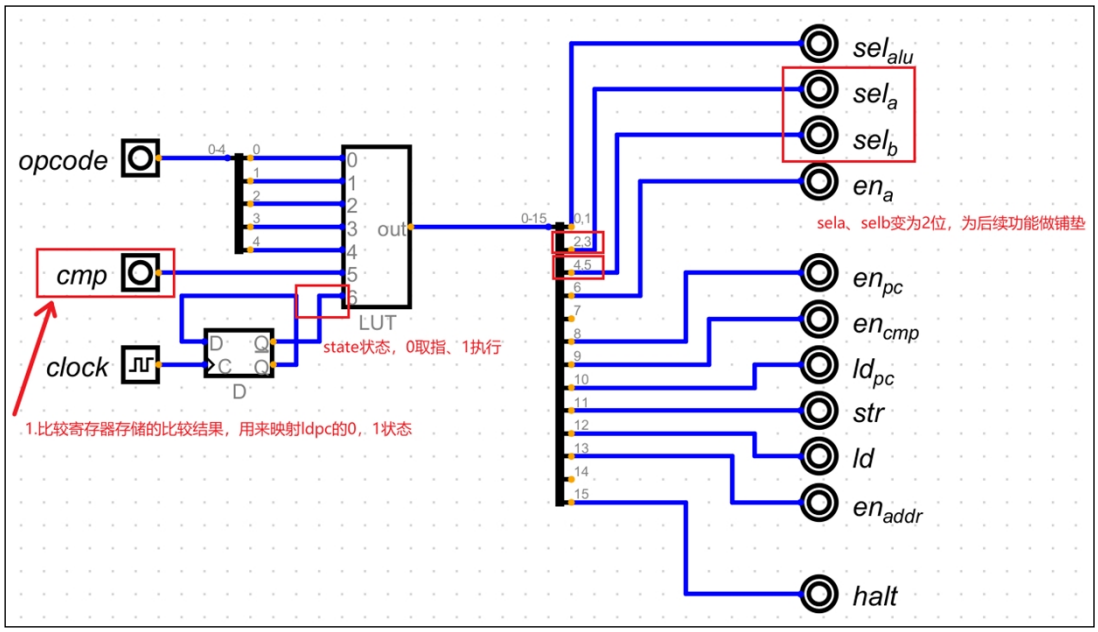
1）查找表加入 state 和 Qcmp 输入

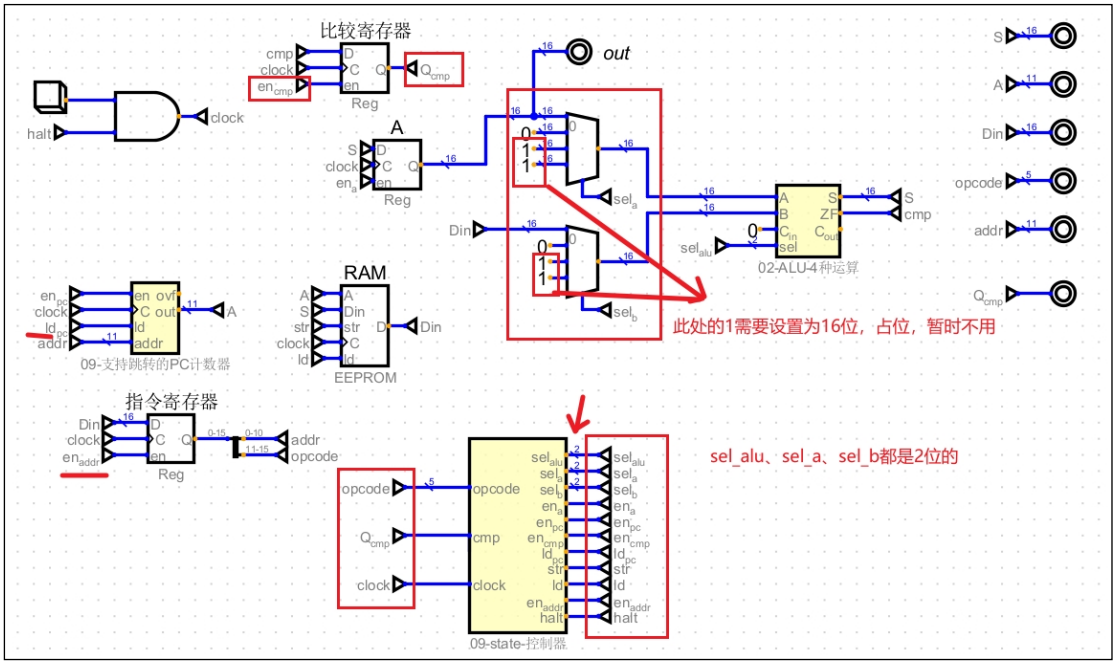
指令的执行改为两个阶段之后（取指令、执行指令），其实就相当于是要根据 state=0、

state=1 时，去设置不同器件的控制信号开关情况（0、1），比如：state=0 时，是取指令阶段，取指令阶段要做的事情是，读取 PC 计数器的地址，并将地址累加 1。用该地址获取到 RAM 中的指令数据。并将指令数据存储到指令寄存器中。要实现以上功能，各器件的控制信号如图所示：



因此，我们需要重构查找表，给控制器加入 state。既然重构，我们将比较寄存器的控制信号也加入其中。Qcmp 存储的是比较两个值的结果，该结果用来进行 je 的跳转。其实是用来控制 PC 计数器的 ld 信号输入。其实也是根据 Qcmp 控制相关信号的 0、1 状态，进而实现跳转的功能，所以也可以将 Qcmp 信号加入到控制器的输入。根据以上分析，我们的控制器输入就有了 opcode、state、Qcmp。重构后如图所示：





2）重新编辑查找表

（1）查找表的映射逻辑

State	Cmp	Opcode	操作
0	0	XX	取指并存到指令寄存器
0	1	XX	取指并存到指令寄存器
1	0	opcode	执行对应指令
1	1	Je	执行 je 跳转

（2）查找表数据

	input	state	cmp	opcode	opname	sel_alu	sel_a	sel_b	en_a	预留	en_pc	en_cmp	ld_pc	str	ld	en_addr	预留	halt	二进制开关	指令寄存器二进制	opname
	0000000	0	0	00000	halt	00	00	00	0	0	1	0	0	0	1	1	0	1	1011000100000000	0b1011000100000000	
	0000001	0	0	00001	ld_a	00	00	00	0	0	1	0	0	0	1	1	0	1	1011000100000000	0b1011000100000000	
	0000010	0	0	00010	add	00	00	00	0	0	1	0	0	0	1	1	0	1	1011000100000000	0b1011000100000000	
	0000011	0	0	00011	sub	00	00	00	0	0	1	0	0	0	1	1	0	1	1011000100000000	0b1011000100000000	
	0000100	0	0	00100	or	00	00	00	0	0	1	0	0	0	1	1	0	1	1011000100000000	0b1011000100000000	
	0000101	0	0	00101	and	00	00	00	0	0	1	0	0	0	1	1	0	1	1011000100000000	0b1011000100000000	
	0000110	0	0	00110	str	00	00	00	0	0	1	0	0	0	1	1	0	1	1011000100000000	0b1011000100000000	
	0000111	0	0	00111	jmp	00	00	00	0	0	1	0	0	0	1	1	0	1	1011000100000000	0b1011000100000000	
0	0001000	0	0	01000	je	00	00	00	0	0	1	0	0	0	1	1	0	1	1011000100000000	0b1011000100000000	
1																					
2	0100000	0	1	00000	halt	00	00	00	0	0	1	0	0	0	1	1	0	1	1011000100000000	0b1011000100000000	
3	0100001	0	1	00001	ld_a	00	00	00	0	0	1	0	0	0	1	1	0	1	1011000100000000	0b1011000100000000	
4	0100010	0	1	00010	add	00	00	00	0	0	1	0	0	0	1	1	0	1	1011000100000000	0b1011000100000000	
5	0100011	0	1	00011	sub	00	00	00	0	0	1	0	0	0	1	1	0	1	1011000100000000	0b1011000100000000	
6	0100100	0	1	00100	or	00	00	00	0	0	1	0	0	0	1	1	0	1	1011000100000000	0b1011000100000000	
7	0100101	0	1	00101	and	00	00	00	0	0	1	0	0	0	1	1	0	1	1011000100000000	0b1011000100000000	
8	0100110	0	1	00110	str	00	00	00	0	0	1	0	0	0	1	1	0	1	1011000100000000	0b1011000100000000	
9	0100111	0	1	00111	jmp	00	00	00	0	0	1	0	0	0	1	1	0	1	1011000100000000	0b1011000100000000	
0	0101000	0	1	01000	je	00	00	00	0	0	1	0	0	0	1	1	0	1	1011000100000000	0b1011000100000000	
1																					
2	1000000	1	0	00000	halt	00	00	00	0	0	0	0	0	0	0	0	0	0	0000000000000000	0b0000000000000000	
3	1000001	1	0	00001	ld_a	00	01	00	1	0	0	0	0	0	1	0	0	1	1001000001000100	0b1001000001000100	
4	1000010	1	0	00010	add	00	00	00	1	0	0	0	0	0	1	0	0	1	1001000001000000	0b1001000001000000	
5	1000011	1	0	00011	sub	01	00	00	1	0	0	0	0	0	1	0	0	1	1001000001000001	0b1001000001000001	
6	1000100	1	0	00100	and	10	00	00	1	0	0	0	0	0	1	0	0	1	1001000001000010	0b1001000001000010	
7	1000101	1	0	00101	or	11	00	00	1	0	0	0	0	0	1	0	0	1	1001000001000011	0b1001000001000011	
8	1000110	1	0	00110	str	00	00	01	0	0	0	0	0	1	1	0	0	1	1001100000010000	0b1001100000010000	
9	1000111	1	0	00111	jmp	00	00	00	0	0	0	0	1	0	0	0	0	1	1000010000000000	0b1000010000000000	
0	1001000	1	0	01000	je	00	00	00	0	0	0	0	0	0	0	0	0	1	1000010000000000	0b1000010000000000	
1																					
2	1100000	1	1	00000	halt	00	00	00	0	0	0	0	0	0	0	0	0	0	0000000000000000	0b0000000000000000	
3	1100001	1	1	00001	ld_a	00	01	00	1	0	0	0	0	0	1	0	0	1	1001000001000100	0b1001000001000100	
4	1100010	1	1	00010	add	00	00	00	1	0	0	0	0	0	1	0	0	1	1001000001000000	0b1001000001000000	
5	1100011	1	1	00011	sub	01	00	00	1	0	0	0	0	0	1	0	0	1	1001000001000001	0b1001000001000001	
6	1100100	1	1	00100	and	10	00	00	1	0	0	0	0	0	1	0	0	1	1001000001000010	0b1001000001000010	
7	1100101	1	1	00101	or	11	00	00	1	0	0	0	0	0	1	0	0	1	1001000001000011	0b1001000001000011	
8	1100110	1	1	00110	str	00	00	01	0	0	0	0	0	1	1	0	0	1	1001100000010000	0b1001100000010000	
9	1100111	1	1	00111	jmp	00	00	00	0	0	0	0	1	0	0	0	0	1	1000010000000000	0b1000010000000000	
0	1101000	1	1	01000	je	00	00	00	0	0	0	0	1	0	0	0	0	1	1000010000000000	0b1000010000000000	

数据

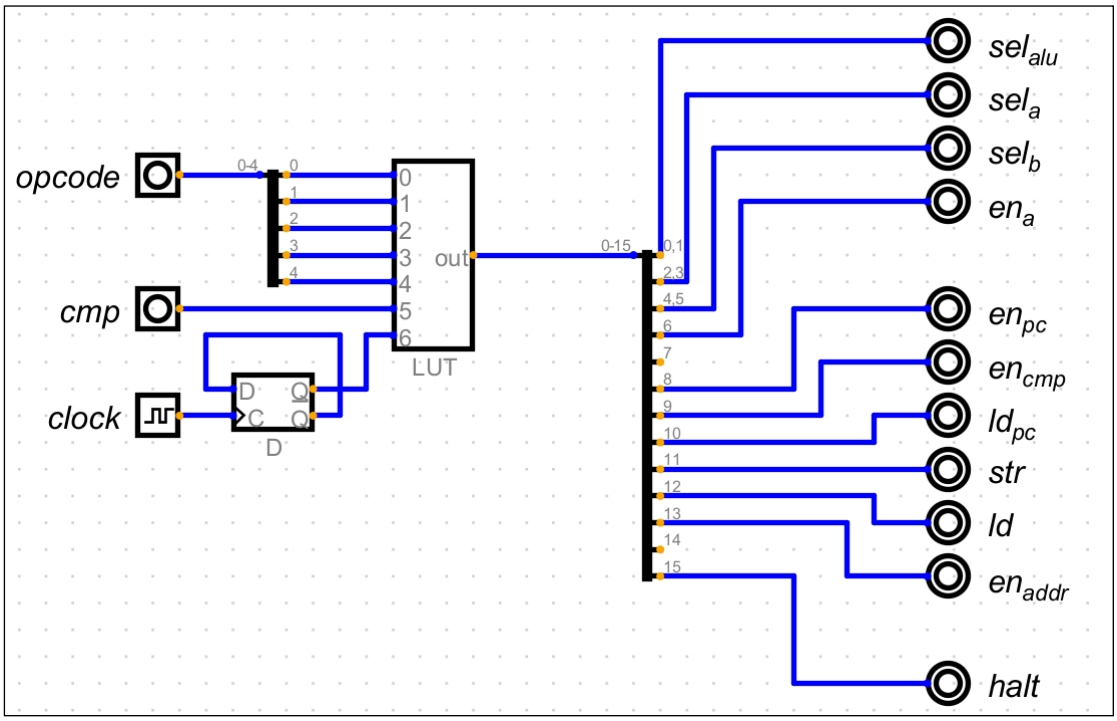
文件

地址	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
0x00	B100	B100	B100	B100	B100	B100	B100	B100
0x08	B100	0	0	0	0	0	0	0
0x10	0	0	0	0	0	0	0	0
0x18	0	0	0	0	0	0	0	0
0x20	B100	B100	B100	B100	B100	B100	B100	B100
0x28	B100	0	0	0	0	0	0	0
0x30	0	0	0	0	0	0	0	0
0x38	0	0	0	0	0	0	0	0
0x40	0	0x9044	0x9040	0x9041	0x9042	0x9043	0x9810	0x8400
0x48	0x8000	0	0	0	0	0	0	0
0x50	0	0	0	0	0	0	0	0
0x58	0	0	0	0	0	0	0	0
0x60	0	0x9044	0x9040	0x9041	0x9042	0x9043	0x9810	0x8400
0x68	0x8400	0	0	0	0	0	0	0
0x70	0	0	0	0	0	0	0	0
0x78	0	0	0	0	0	0	0	0

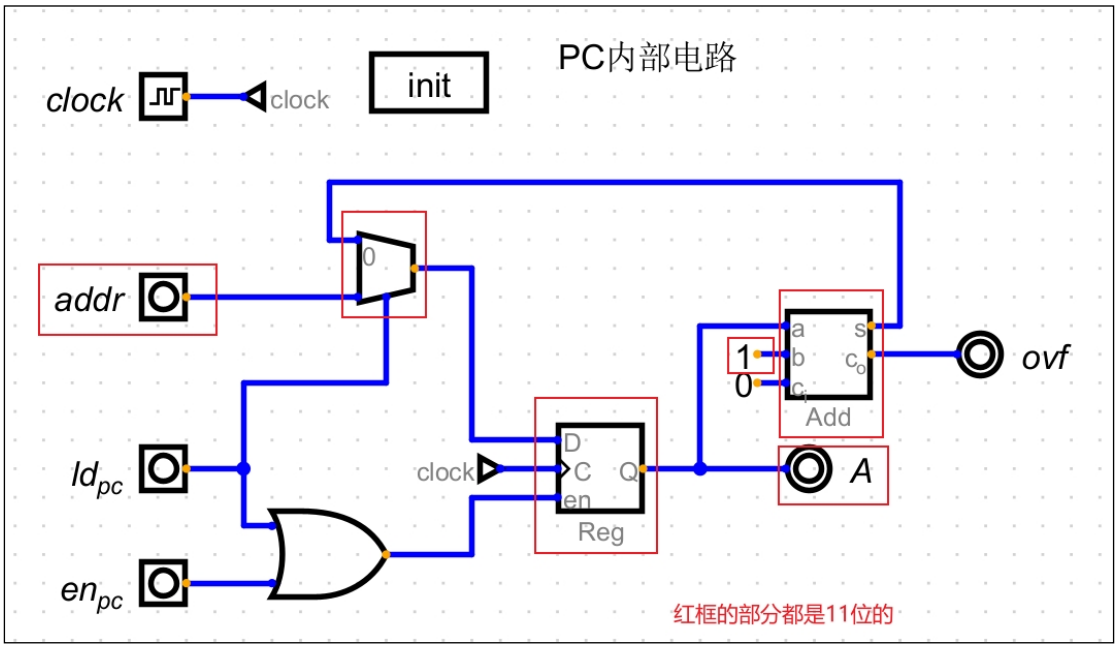
确定

3.7.6.6 最终电路图

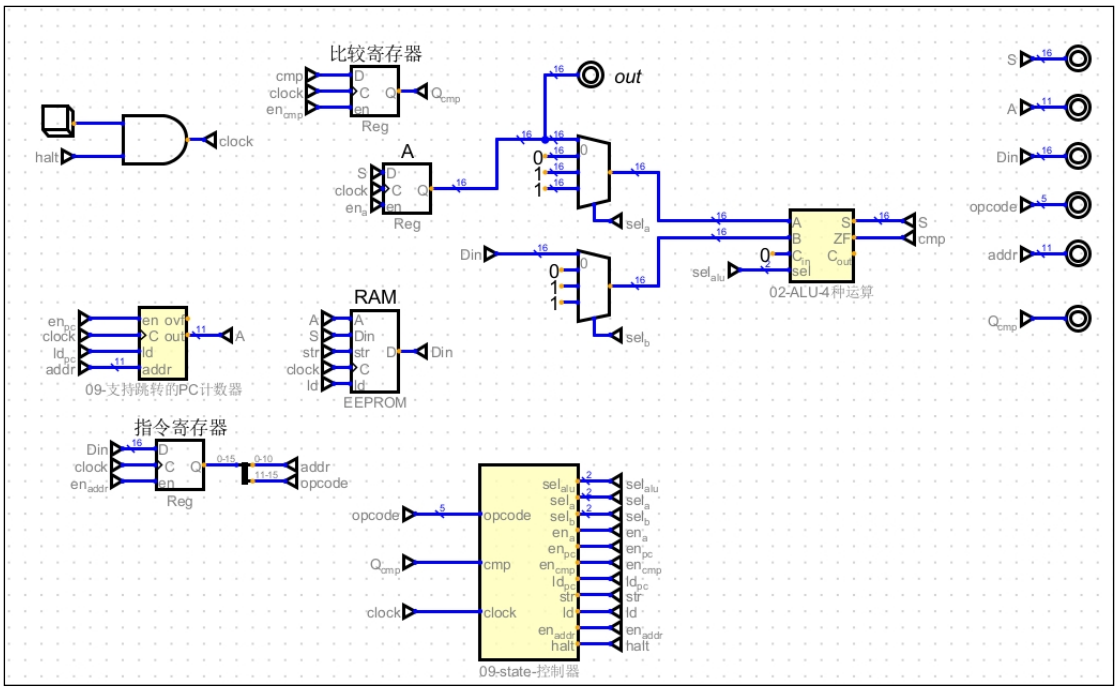
1）控制器电路图



2）PC 计数器电路图

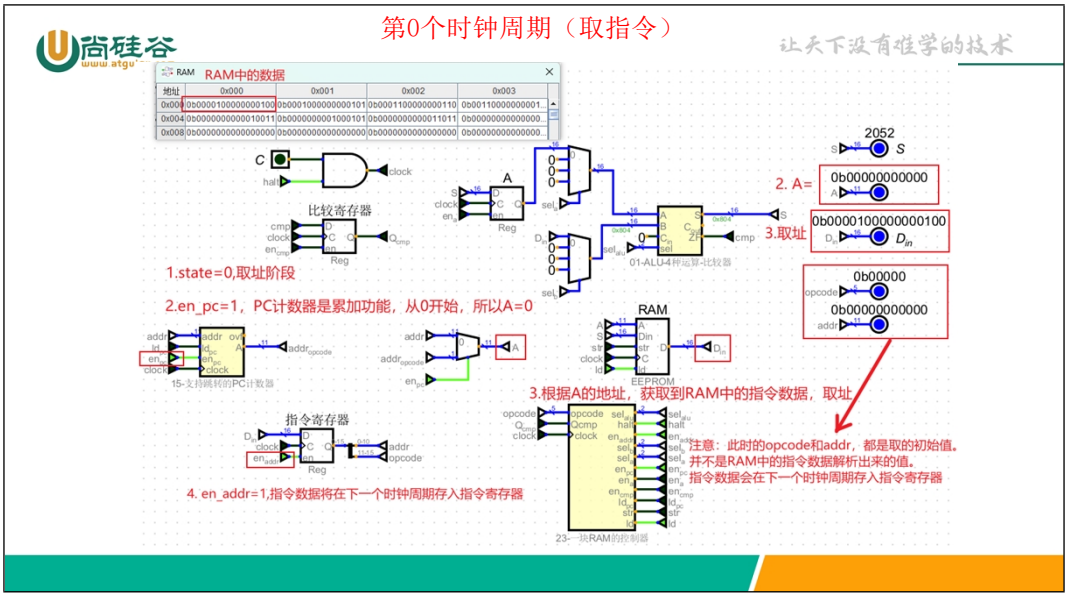


3）整体电路图



3.7.6.7 执行过程演示

现在我们要用两个时钟周期完成，第一个时钟周期获取指令，第二个时钟周期执行指令。所以我们的计数器要两个周期再递增。



3.8 添加立即数

现有系统的数据 RAM 存储方式（操作码+操作数）。

地址	指令	opcode（操作码）	addr（操作数）	新代码 / 数据
000	ld_a	0b00001	0b100	0b0000100000000100

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

001	add	0b00010	0b101	0b0001000000000101
010	sub	0b00011	0b110	0b0001100000000110
011	str	0b00110	0b111	0b0011000000000111
100				0b0000000000010011
101				0b0000000001000101
110				0b0000000000011011
111				

现有我们的 RAM 中，操作数指的是数据 RAM 的地址。也就是在执行指令的过程中，需要操作数指定的地址到 RAM 中读取数据运算所需要的数据。这个数据获取的过程是间接的。

我们希望增加一个直接获取运算的数据的方式，也就是说操作数不是数据 ROM 的地址，而是表示具体的要参与运算的数据。这样的操作数称为立即数。我们举例说明：比如我们要计算 19+69-27。

地址	**指令**	**opcode（操作码）**	**addr（操作数）**	**新代码 / 数据**
000	ld_a	0b00001	0b00000010011（19）	0b0000100000010011
001	add	0b00010	0b00001000101（69）	0b0001000001000101
010	sub	0b00011	0b00000011011（27）	0b0001100000011011
011	str	0b00110	0b00000000011（3）	0b0011000000000011

以上就是操作码+立即数的方式。蓝色是操作码，红色是立即数。

这样可以直接操作数据，但是也会有一个新的问题，就是什么时候操作码表示数据 ROM 的地址，什么时候操作码表示立即数呢？

比如，有一条指令 ld_a 7, 那么 7 表示的是数据 RAM 的地址？还是立即数 7 呢？现有系统我们无法区分，所以，既然通过操作数无法区分，我们可以通过指令来区分，也就是设计两套指令，一套指令操作数表示数据 RAM 的地址，一套指令操作数表示立即数。我们现有指令就是操作数是数据 RAM 地址的指令，因此，我们只需要增加设计一套操作数的指令即可。

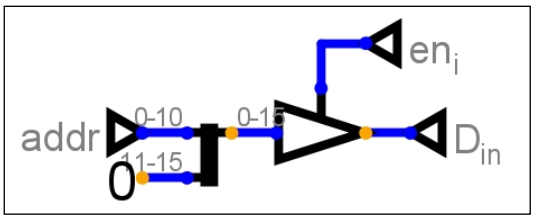
3.8.1 增加立即数后的指令

opname	**opcode**	**操作**
halt	00000	停止时钟
ld_a	00001	将数据 RAM 的数据载入寄存器 A

add	00010	数据 RAM 的数据+寄存器 A 再存入寄存器 A
sub	00011	寄存器 A-数据 RAM 再存入寄存器 A
or	00100	寄存器 A||数据 RAM 再存入寄存器 A
and	00101	寄存器 A&&数据 RAM 再存入寄存器 A
str	00110	将寄存器 A 中数据存入数据 RAM
jmp	00111	无条件跳转到指定地址
je	01000	比较相等时跳转到指定地址
ld_ia	01001	将立即数存入寄存器 A
add_ia	01010	将立即数+寄存器 A 中的数再存到寄存器 A
sub_ia	01011	寄存器 A-立即数再存到寄存器 A
or_ia	01100	寄存器 A||立即数再存到寄存器 A
and_ia	01101	寄存器 A&&立即数再存到寄存器 A

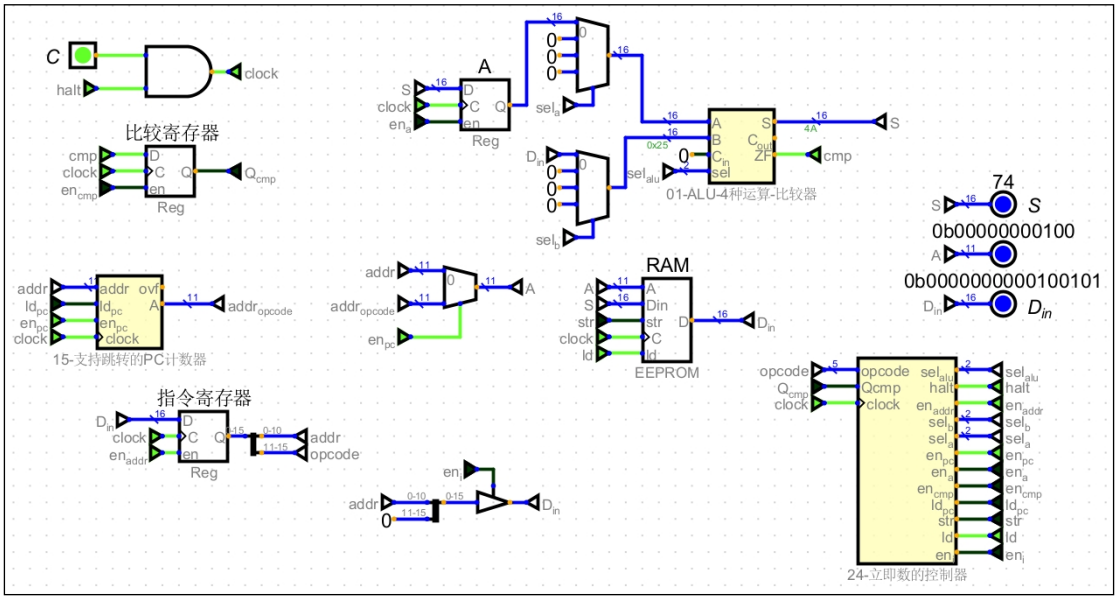
3.8.2 电路改造

立即数就是将操作码当做数直接进入 ALU 计算，我们的操作码 addr 原来表示的是 RAM 的地址，位数是 11。因此，头部补 5 位的 0，就凑成了 11 位的 Din，作为 ALU 的 B 信号的输入了。



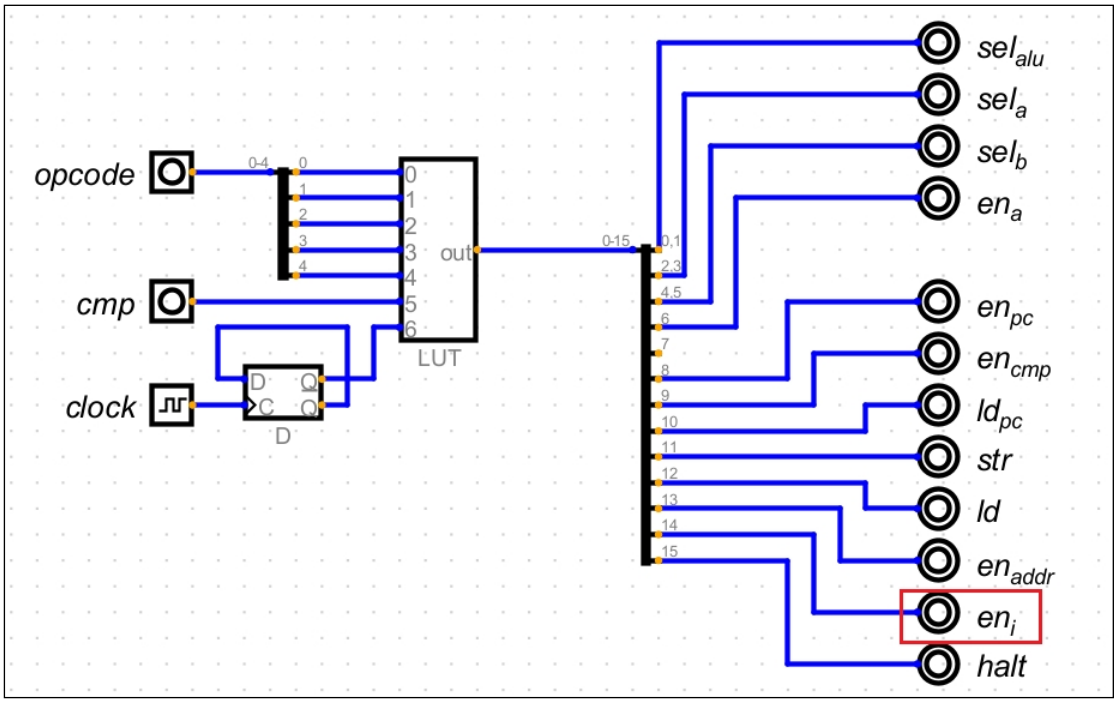
将指令寄存器输出的 11bit 地址补全到 16 位接入地址总线。通过 en_i 信号控制。

整体电路如下图：



3.8.3 控制器改造

1）增加 en_i 控制信号



2）修改控制器查找表

数据

文件

地址	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
0x00	B100	B100	B100	B100	B100	B100	B100	B100
0x08	B100	B100	B100	B100	B100	B100	0	0
0x10	0	0	0	0	0	0	0	0
0x18	0	0	0	0	0	0	0	0
0x20	B100	B100	B100	B100	B100	B100	B100	B100
0x28	B100	B100	B100	B100	B100	B100	0	0
0x30	0	0	0	0	0	0	0	0
0x38	0	0	0	0	0	0	0	0
0x40	0	0x9044	0x9040	0x9041	0x9042	0x9043	0x9810	0x8400
0x48	0x8000	C044	C040	C041	C043	C042	0	0
0x50	0	0	0	0	0	0	0	0
0x58	0	0	0	0	0	0	0	0
0x60	0	0x9044	0x9040	0x9041	0x9042	0x9043	0x9810	0x8400
0x68	0x8400	C044	C040	C041	C043	C042	0	0
0x70	0	0	0	0	0	0	0	0
0x78	0	0	0	0	0	0	0	0

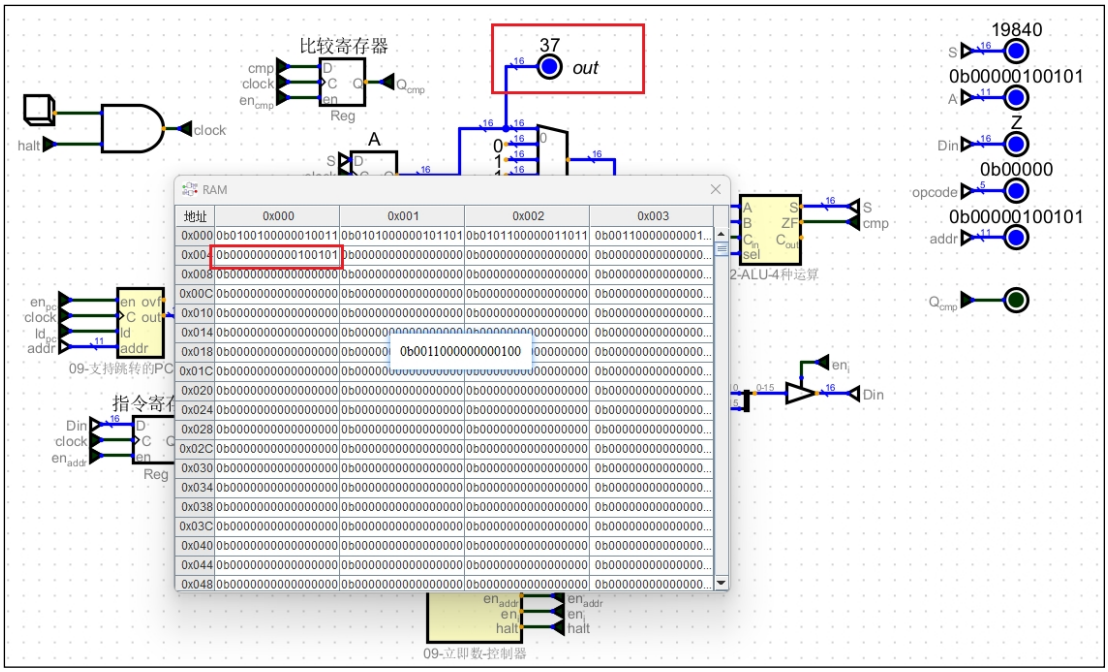
确定

3.8.4 测试立即数功能

我们要计算 19+45-27，并存储到 4 号地址位，那么我们 RAM 存储的数据如下图：

内存地址	opname	opcode	操作数	二进制表示
00	ld_ia	01001	00000010011(19)	0b0100100000010011
01	add_ia	01010	00000101101(45)	0b0101000000101101
10	sub_ia	01011	00000011011(27)	0b0101100000011011
11	str	00110	00000000100(4)	0b0011000000000100

执行结果如图：

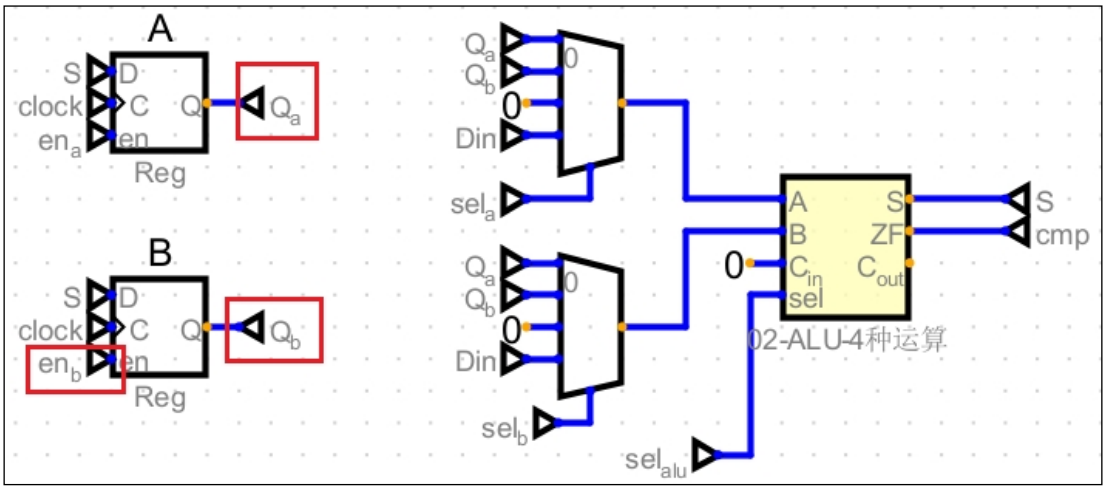


3.9 添加 B 寄存器

目前我们只有一个寄存器 A，寄存器相当于是编程中的一个变量，也就是我们现在的系统只支持一个变量的运算。如果我们需要进行多个变量的运算，一个寄存器是不够的。因此，我们需要为系统再添加一个寄存器 B。

3.9.1 电路改造

1）添加 B 寄存器，修改 ALU 输入端信号



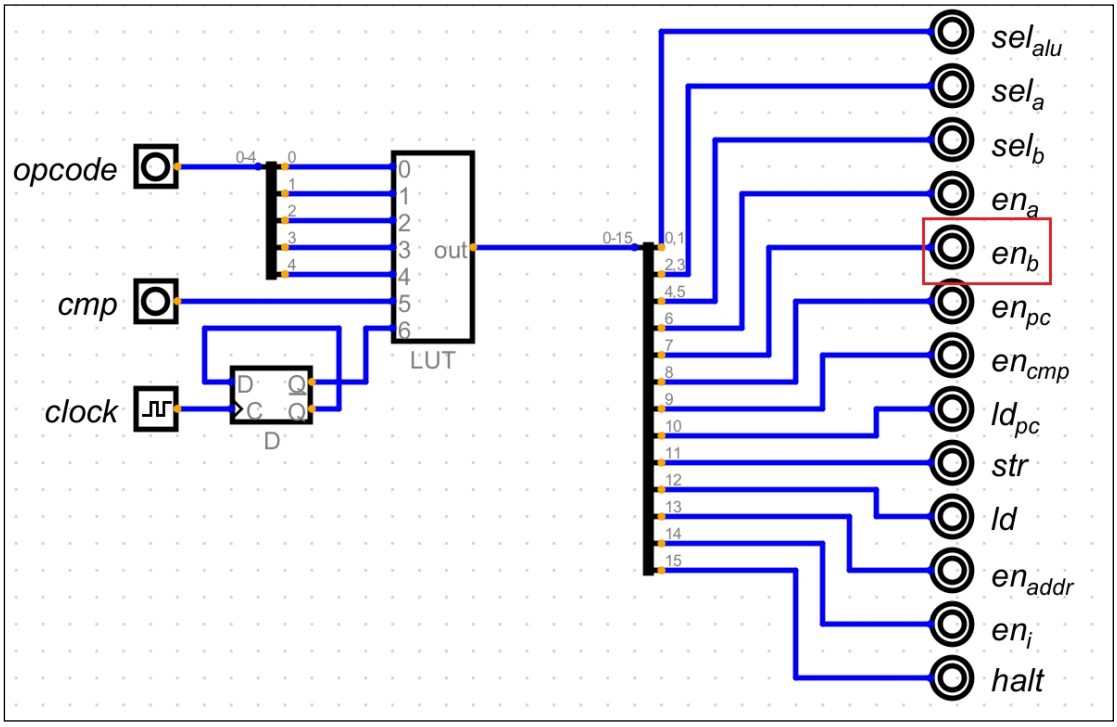
B 寄存器写入使能端 en_b，在控制器里需要增加 en_b 控制信号，同时需要修改控制器查找表控制 en_b。

增加 B 寄存器后，ALU 的 A、B 输入可以有 4 中选择，从 A、B、RAM（立即数）、0

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

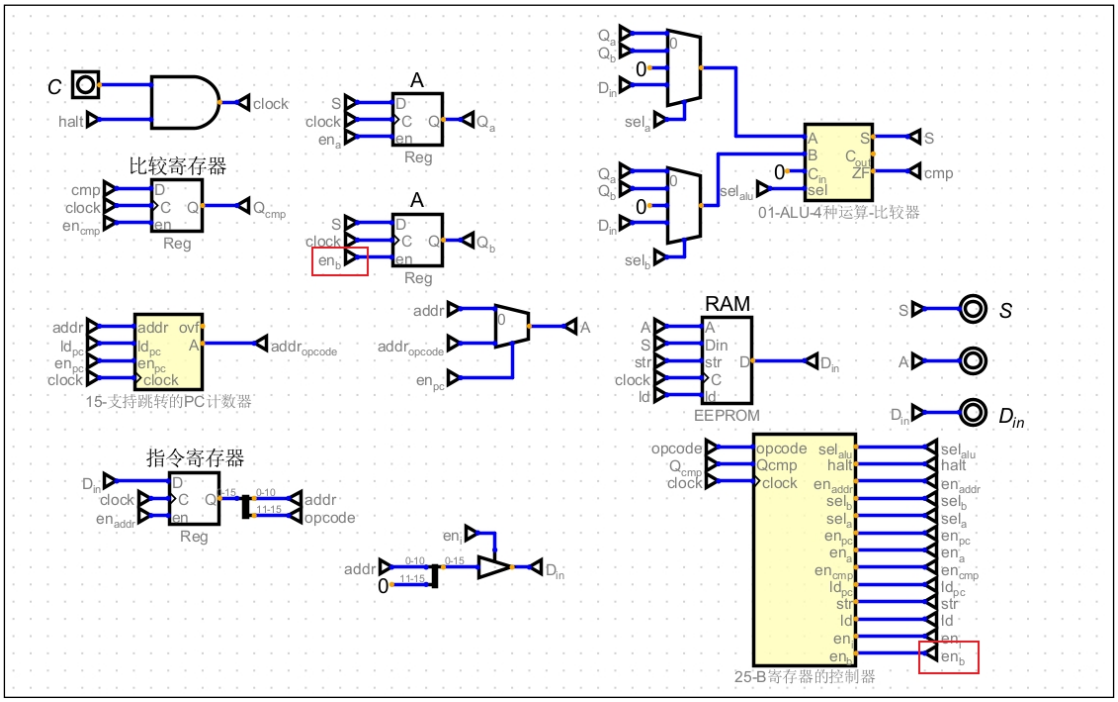
获取。通过 sel_a、sel_b 来控制。

2）控制器改造



预留的 6 号引脚控制 en_b。

3）整体电路



3.9.2 指令集

指令集，就是你希望CPU能够完成的操作。现在流行的指令集有x86,arm和risc-V。前两个指令集都是有知识产权的，不能随便用，而risc-V是开源指令集。指令集不光是随便设计一套指令就可的，最重要的是上下游的生态环境。比如同学们之前学过的C语言，在编译过程中，gcc工具链可以将C语言完整编译成x86指令，gcc-eabi工具链可以编译arm指令，risc-V也有对应的工具链，但是其上下游其他配套设施无法与前两者相提并论。

无论是x86，arm，还是risc-V指令集，对于我们设计的这个基础CPU都过于复杂，所以我们接下来会自己设计一套指令集，相对的，我们就没有配套的编译器为我们完成编译工作，接下来的编译工作我们只能自己手动进行。我们的指令集支持无符号16位数的加、减、按位与、按位或，支持基于等值的循环和分支控制，支持立即数对寄存器的运算。

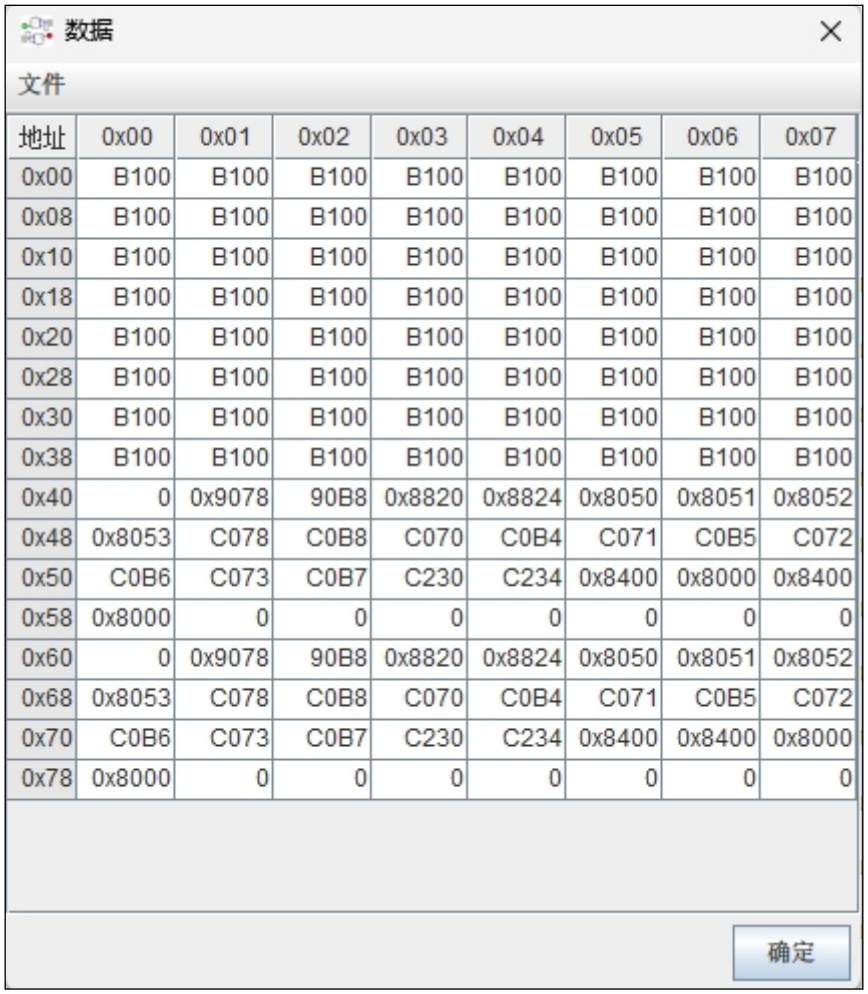
我们的指令长度为16bit，前5bit为操作码，后11bit为地址或者立即数。操作码与指令的对照表如下：

指令	操作码	注释
halt	0b00000	终止程序
ld_a	0b00001	将内存RAM中的值载入a寄存器
ld_b	0b00010	将内存RAM中的值载入b寄存器
save_a	0b00011	将寄存器a的值保存到内存RAM
save_b	0b00100	将寄存器b的值保存到内存RAM
add	0b00101	将寄存器ab相加，保存到a
sub	0b00110	将寄存器ab相减，保存到a
and	0b00111	将寄存器ab按位与，保存到a
or	0b01000	将寄存器ab按位或，保存到a
ld_ia	0b01001	将立即数载入寄存器a
ld_ib	0b01010	将立即数载入寄存器b
add_ia	0b01011	将立即数加到寄存器a
add_ib	0b01100	将立即数加到寄存器b
sub_ia	0b01101	从寄存器a中减去立即数
sub_ib	0b01110	从寄存器b中减去立即数
and_ia	0b01111	将寄存器a与立即数取与
and_ib	0b10000	将寄存器b与立即数取与
or_ia	0b10001	将寄存器a与立即数取或

更多Java –大数据 –前端 –python人工智能资料下载，可百度访问：尚硅谷官网

or_ib	0b10010	将寄存器 b 与立即数取或
cmp_a	0b10011	比较寄存器 a 和立即数是否相等
cmp_b	0b10100	比较寄存器 b 和立即数是否相等
jmp	0b10101	无条件跳转到指定地址执行
je	0b10110	如果比较结果相等才跳转
jne	0b10111	如果比较结果不等才跳转
nop	0b11000	什么都不干，跳过一步指令

3.9.3 更新控制器查找表



地址	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
0x00	B100	B100	B100	B100	B100	B100	B100	B100
0x08	B100	B100	B100	B100	B100	B100	B100	B100
0x10	B100	B100	B100	B100	B100	B100	B100	B100
0x18	B100	B100	B100	B100	B100	B100	B100	B100
0x20	B100	B100	B100	B100	B100	B100	B100	B100
0x28	B100	B100	B100	B100	B100	B100	B100	B100
0x30	B100	B100	B100	B100	B100	B100	B100	B100
0x38	B100	B100	B100	B100	B100	B100	B100	B100
0x40	0	0x9078	90B8	0x8820	0x8824	0x8050	0x8051	0x8052
0x48	0x8053	C078	C0B8	C070	C0B4	C071	C0B5	C072
0x50	C0B6	C073	C0B7	C230	C234	0x8400	0x8000	0x8400
0x58	0x8000	0	0	0	0	0	0	0
0x60	0	0x9078	90B8	0x8820	0x8824	0x8050	0x8051	0x8052
0x68	0x8053	C078	C0B8	C070	C0B4	C071	C0B5	C072
0x70	C0B6	C073	C0B7	C230	C234	0x8400	0x8400	0x8000
0x78	0x8000	0	0	0	0	0	0	0

3.9.4 测试

x=19，y=45，计算 x+y 的值 64 存储到 RAM 的 7 号地址中。

分析：x 变量看成寄存器 A，y 变量看成寄存器 B，那么上面的功能翻译成操作数和操作码，数据如下：

内存地址	opname	opcode	操作数	二进制表示

000	ld_ia	01001	00000010011(19)	0b0100100000010011
001	ld_ib	01010	00000101101(45)	0b0101000000101101
010	add	00101	xxx	0b0010100000000000
011	save_a	00011	00000000111(7)	0b0001100000000111
100	halt	00000	00000000000	0b0000000000000000

RAM

地址	0x000	0x001	0x002	0x003
0x000	0b0100100000010011	0b0101000000101101	0b0010100000000000	0b0001100000000111
0x004	0b0000000000000000	0b0000000000000000	0b0000000000000000	0b0000000001000000
0x008	0b0000000000000000	0b0000000000000000	0b0000000000000000	0b0000000000000000

第 4 章 设计计算机汇编语言

4.1 用自己的汇编语言编写程序

上面的例子过于简单，体现不出我们系统的强大，接下来我们来实现一段程序。0 到 100 求和。

请看下面这段代码片段：

```
int main()
{
    int s=0;
    for (int i=1;i!=101;i++)
    {
        s+=i;
    }
}
```

我们可以在 https://gcc.godbolt.org/ 网站将这段代码编译成 x86 的汇编语言，结果如下：

```
main:
        //将 rbp 压栈
        push    rbp
        //将 rsp 的值赋给 rbp
        mov     rbp, rsp
        //将 rbp 低 8 位指针指向的值赋 0（[rbp-4]就是 s 变量的指针地址）
        mov     DWORD PTR [rbp-4], 0
        //将 rbp 低 8 位指针指向的值赋 1（[rbp-8]就是 i 变量的指针地址）
        mov     DWORD PTR [rbp-8], 1
        //跳到.L2 标签
        jmp     .L2
.L3:
        //将 i 的值赋给 eax 寄存器
        mov     eax, DWORD PTR [rbp-8]
        //将 eax 寄存器的值加到 s
        add     DWORD PTR [rbp-4], eax
        //将 i 的值加 1
        add     DWORD PTR [rbp-8], 1
```

```
.L2:
        //比较 i 和 100，结果保存到比较寄存器
        cmp     DWORD PTR [rbp-8], 101
        //比较寄存器不为 1（不相等），跳到.L3 标签
        jne     .L3
        //将 0 赋给 eax 寄存器
        mov     eax, 0
        //弹栈，赋给 rbp（相当于 rbp 恢复道进入函数之前）
        pop     rbp
        //函数结束
        ret
```

尽管我们自己设计的汇编语言没有编译器，但是我们可以仿照上面的 x86 汇编将上面的代码复刻一段并在我们自己的 CPU 上执行。

```
//给寄存器 a 赋值 0
0. ld_ia 0
//给寄存器 b 赋值 1
1. ld_ib 1
//跳到第 5 条指令
2. jmp 5
//a=a+b
3. add 0 将寄存器 ab 相加保存到 a
//b=b+1
4. add_ib 1
//比较 b 和 101
5. cmp_b 101
//不相等跳到第 3 条指令
6. jne 3
//将 a 保存到地址 16
7. save_a 16
//停止程序
8. halt
```

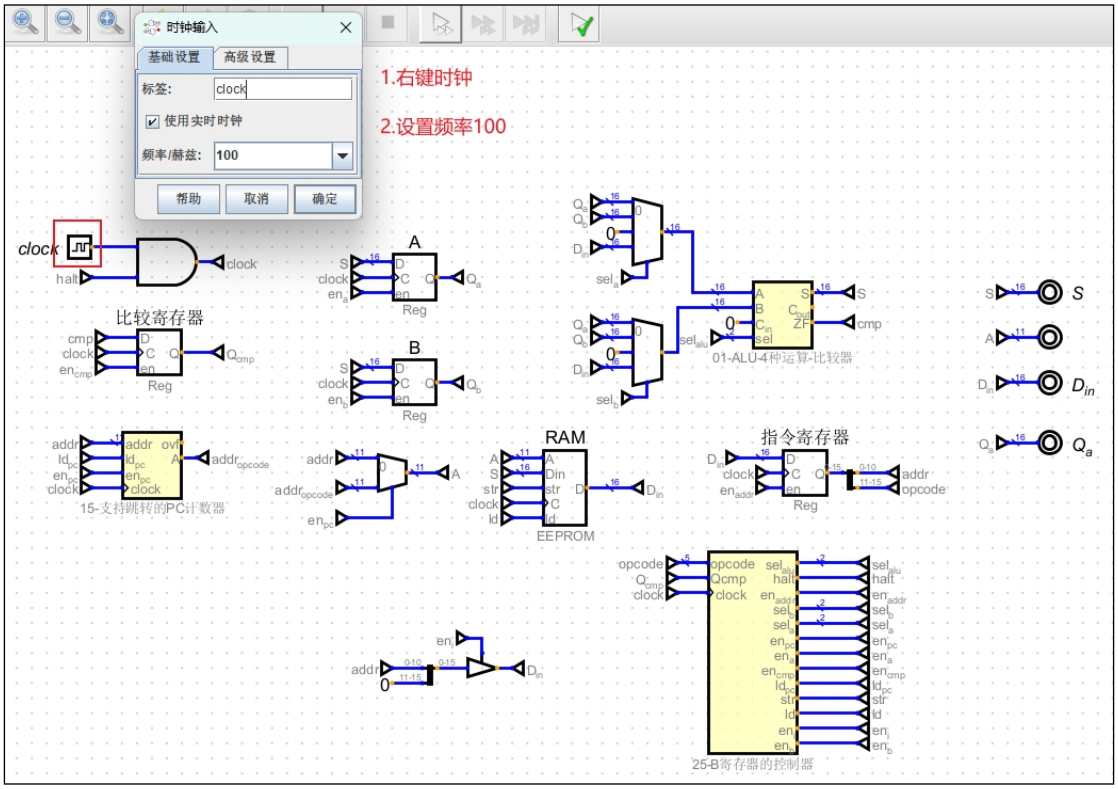
翻译成二进制。

```
0b0100100000000000        ld_ia   0        0
0b0101000000000001        ld_ib   1        1
0b1010100000000101        jmp     5        2
0b0010100000000000        add     0        3
0b0110000000000001        add_ib  1        4
0b1010000001100101        cmp_b   101      5
0b1011100000000011        jne 3            6
0b0001100000010000        save_a  16       7
0b0000000000000000        halt             8
```

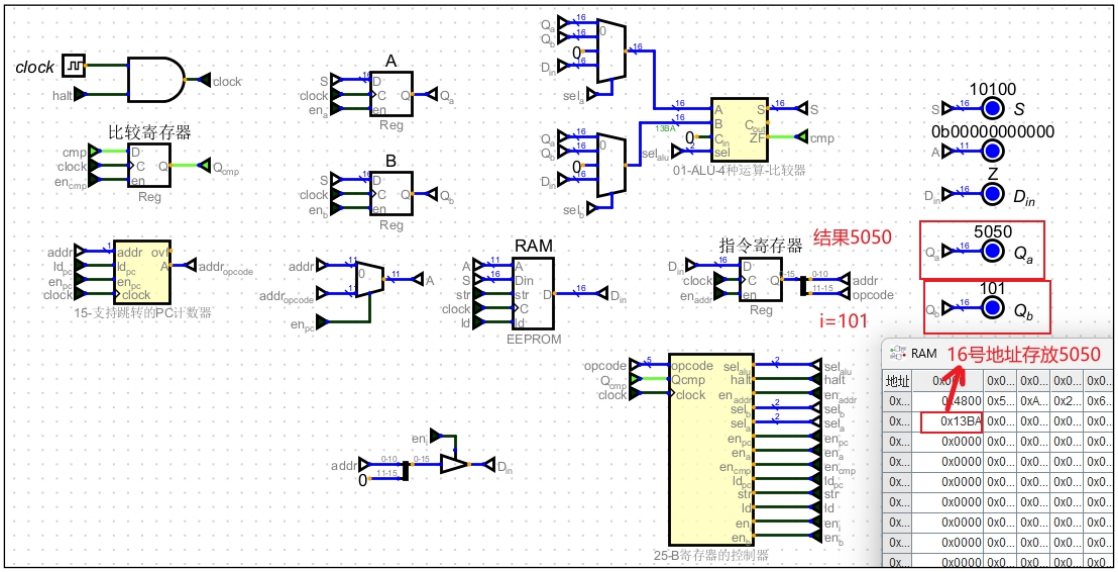
烧录进 eeprom，将时钟频率调整位 500，开始仿真，程序结束后注意观察内存中 16 地址的值。

4.2 测试运行

1）替换时钟信号，并设置频率为 100HZ

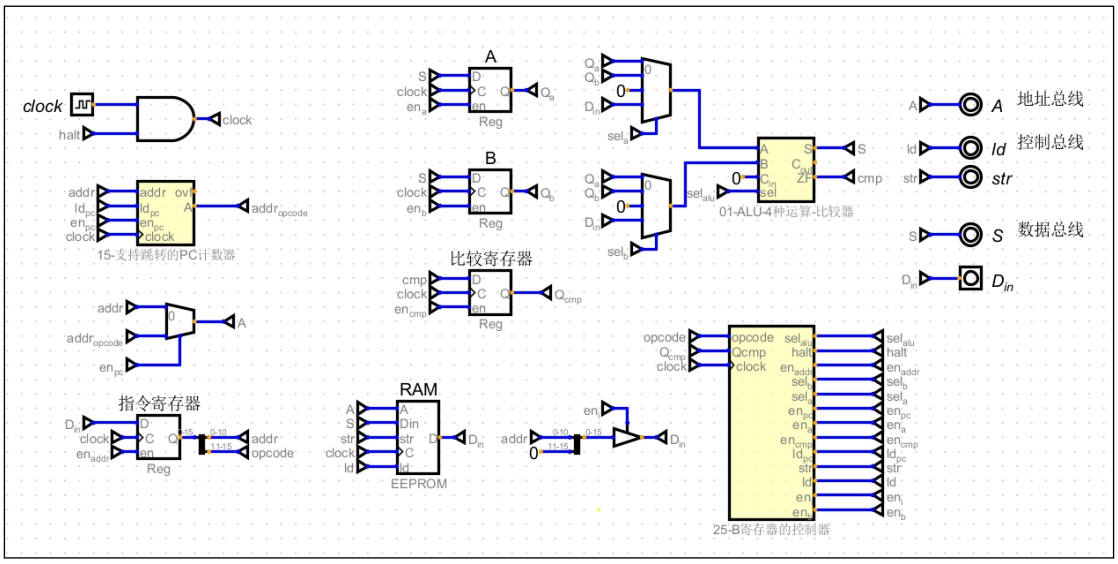


2）仿真执行查看结果



设计计算器总线与其他输入输出设备

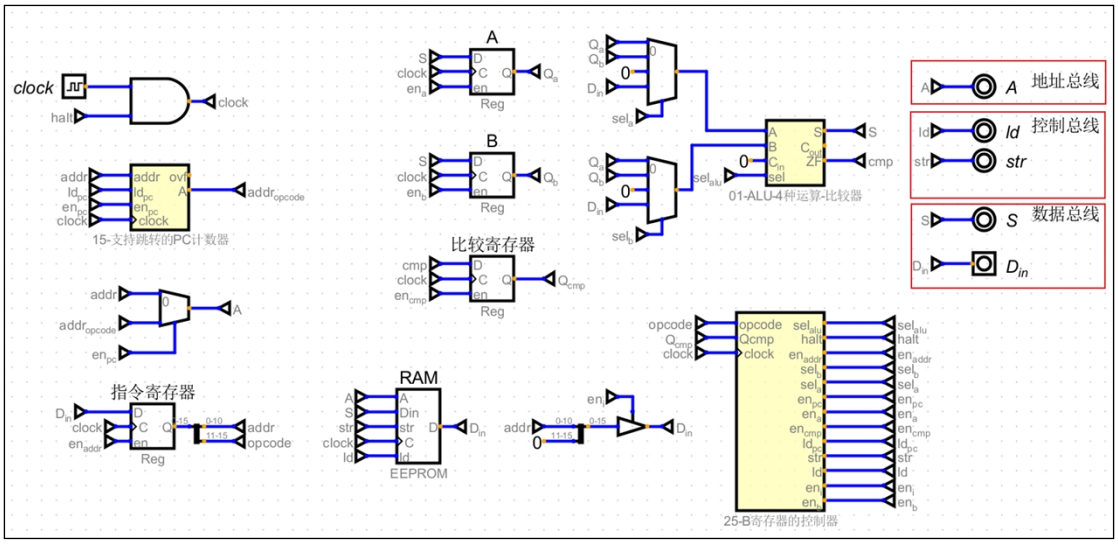
4.3 CPU 电路



我们之前实现的电路，其实是一个完整的 CPU 电路，这个 CPU 包含完整的 ALU，控制器和寄存器组，还有一个内置的RAM。那如果我们想通过CPU控制外部设备，又该如何实现呢？接下来我们以内存，硬盘和显示器为例，来讲解 CPU 如何通过总线控制外部设备。

4.4 什么是总线

总线，在前面的课程中我们有过介绍，但是没有相应的实物，看起来比较抽象，其实总线就是 CPU 和外部通讯的导线，如果我们将 CPU 结构稍作改造，例如下图：



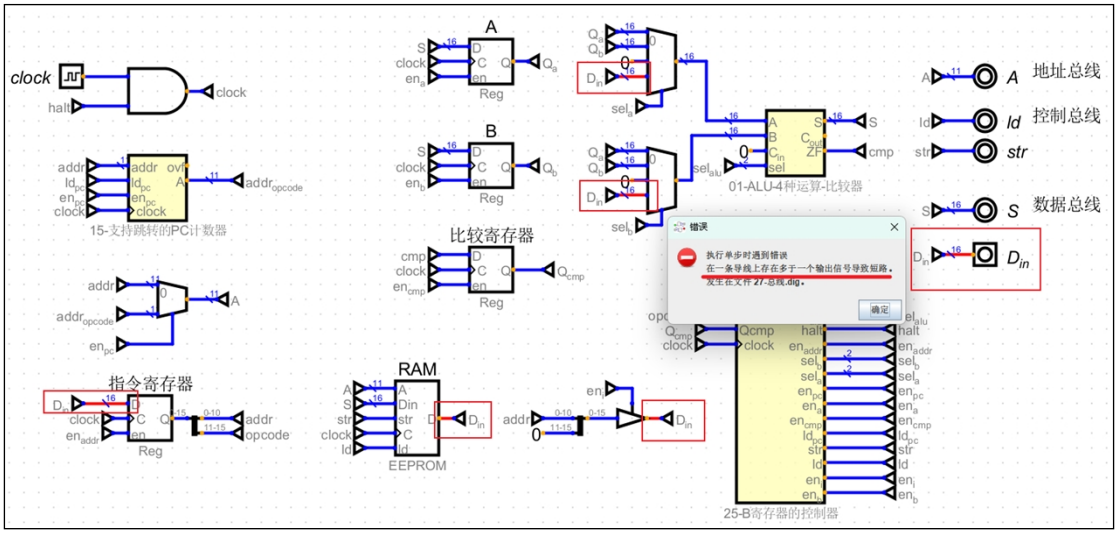
我们将 A，str，ld，Dout 和 Din 五根导线分别接上输入和输出，就可以实现 CPU 和外部数据的交换，而这五根线就是总线，其中：

- A 线为数据总线，用来传输 CPU 寻址地址。

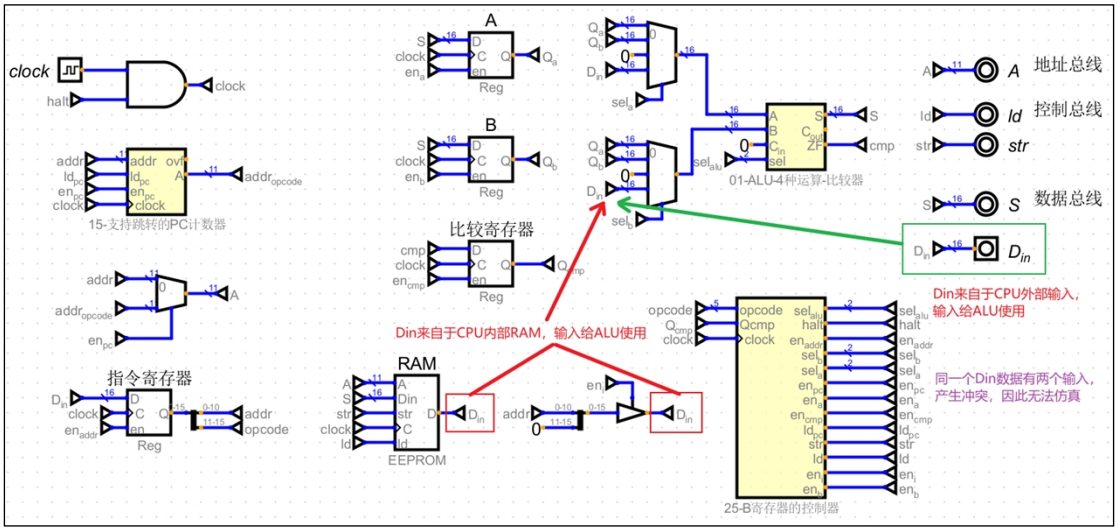
- ld 和 str 线为控制总线，用来传输 CPU 控制信号，表明 CPU 是打算对地址进行读，还是写。
- Din 和 Dout 为数据总线，用来传输数据。

这时就会产生一个问题：我们的 Din 数据源，既可以从内部 RAM 读取数据，也可以从外部设备读取数据，那我们以哪一个为准呢？或者说，如果存在多个外部设备，CPU 如何判断自己控制的是哪一个外部设备呢？

在讲总线地址映射之前，我们会发现现在的 CPU 电路已经无法单独仿真了，会提示 Din 短路。这是为什么呢？

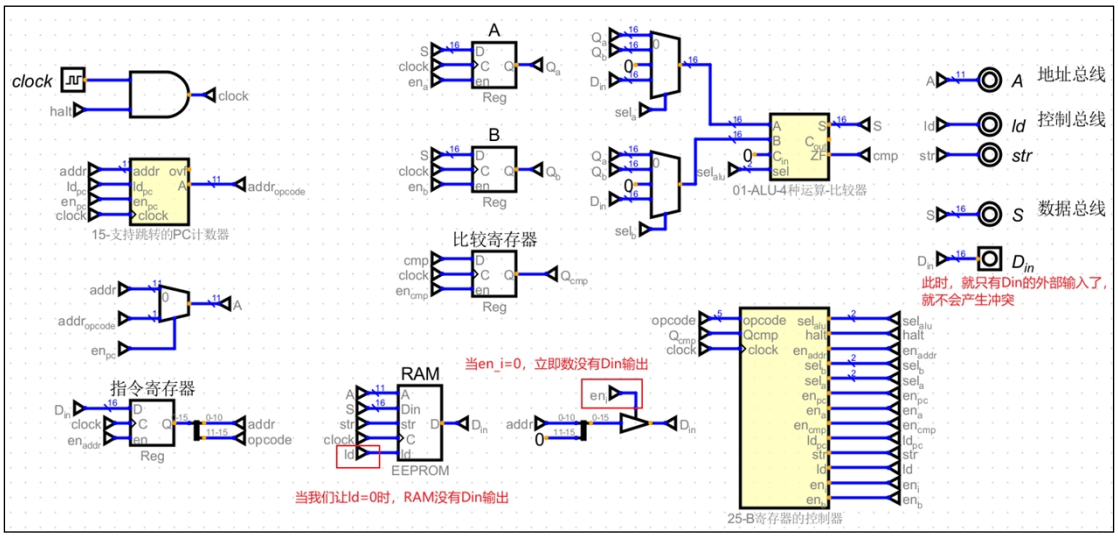


修改后的CPU电路，我们的Din不仅可以从RAM中读取数据，输入给ALU进行处理。还可以通过外部输入 Din 数据给 ALU 处理。那么在当前 CPU 电路中，Din 的输入来源会有两处，会出现短路状态。因此，无法仿真。如图：



那如何解决 Din 冲突的问题呢？通过控制 ld 和 en_i 的信号。原理是什么呢？如下图：

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网



当 RAM 的 ld=0 时，内部 RAM 的 Din 就没有输入了，当 en_i=0 时，立即数 Din 也没有输入源，此时，Din 信号就只有外部输入的 Din。所以，在实际电路应用中，我们可以通过控制信号来让电路生效。这就是我们当前 CPU 电路不能单独进行仿真的原因了。

4.5 总线地址映射

现代计算机中，一种解决方案是，采用 CPU 地址映射的方式来区别外部设备。例如，我们的 A 信号（寻址信号）长度为 11 位，我们可以用其中的 3 位表示总线上的设备编号，后 8 位表示总线上的外部设备的内部地址，就可以对设备进行区分。来看如下指令：

```
ld_a     0b010 00000000
save_a   0b001 00000000
```

按照我们之前学习的知识，上面这段指令的意思是从 0b010 00000000 读取数据，写入到 0b001 00000000 地址。但是如果我们按照总线地址映射的方式来看，这两句的指令的意思就是：

```
// 从第三个设备的 00000000 地址读取数据
ld_a     0b010 00000000
// 保存到第二个设备的 00000000 地址
save_a   0b001 00000000
```

这样我们就可以通过简单的寻址动作，完成 CPU 和外部设备通信。接下来，我们来完成 CPU 外部地址映射的设计：

CPU 寻址地址	外部设备映射
0b000 00000000~0b000 11111111	内部 Rom，简称 iRom （0-255）
0b001 00000000~0b001 11111111	外部扩展 Ram，就是我们的计算机内存 （256-511）
0b010 00000000~0b010 11111111	外部扩展 Rom，可以看作是计算机硬盘

	（512-767）
0b011 00000000	终端地址，向这个地址写数据可以在终端显示（768）

这里其实存在几个小问题，例如：

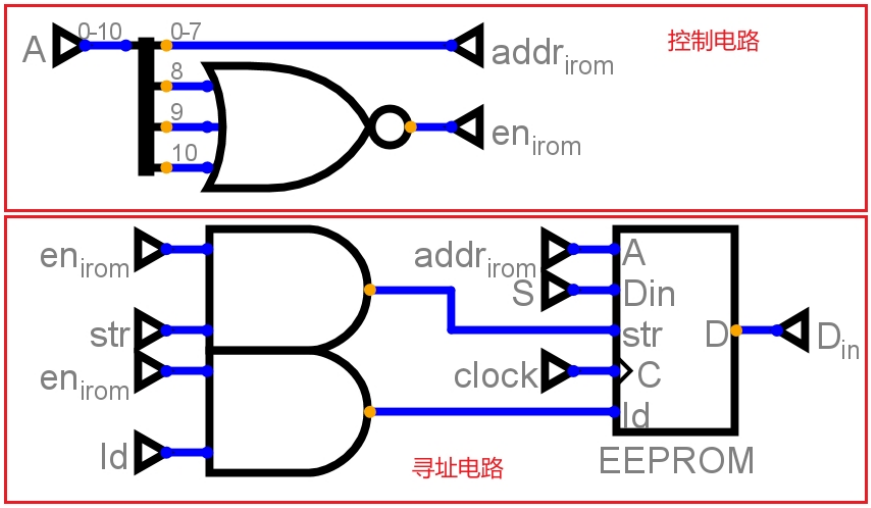
- 我们的寻址和标准的寻址不同，一次返回两个字节；
- iRom 寻址空间一般非常小；
- 外部 RAM 和 ROM 都偏小；
- CPU 不支持中断，无法接入外部输入设备；
- 我们没有用完所有的寻址空间。

以上这些问题确实存在，但是我们的课程以帮助大家理解 CPU 内部原理为主，并没有按照现实 1：1 复刻。

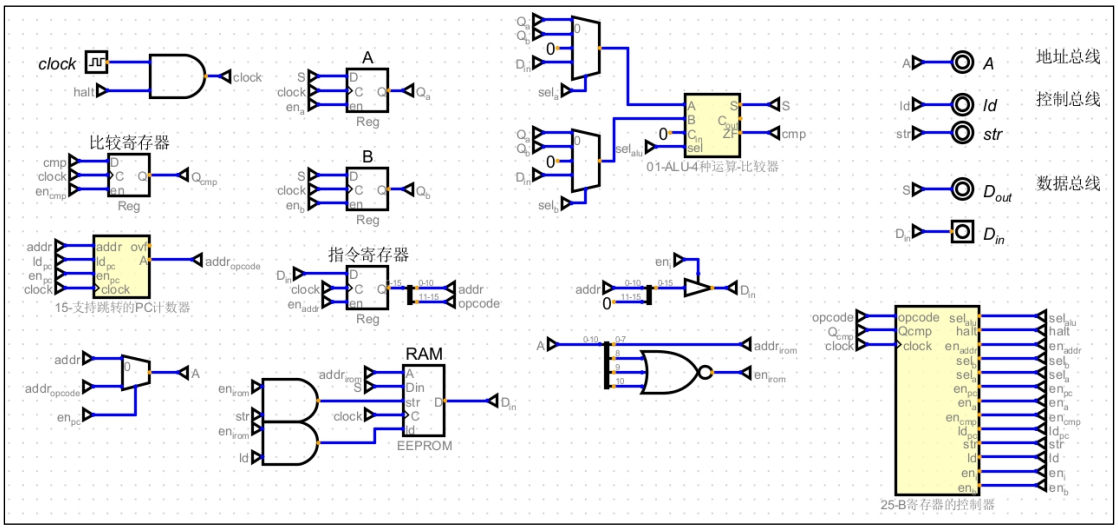
基于以上地址映射设计，做如下电路更改：

1）CPU 内部 RAM 电路修改

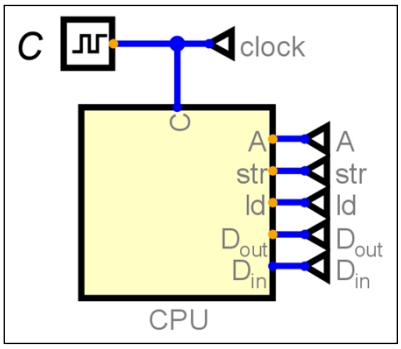
现在我们需要限制内部 RAM 只响应 0b000 00000000~0b000 11111111 的寻址，需要对其电路做一定更改，如下图：



在控制电路中，我们将高 3bit 组合成了 en 信号，从而实现只响应 0b000 信号；而低 8bit 直接接入 iRom 寻址地址实现寻址。改造过后，CPU 电路如下图：

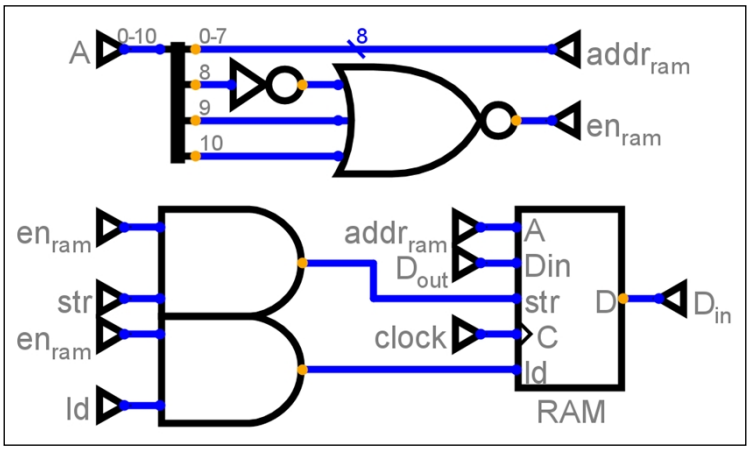


将 CPU 封装成子电路，方便我们以后调用。如下图：



2）内存模块

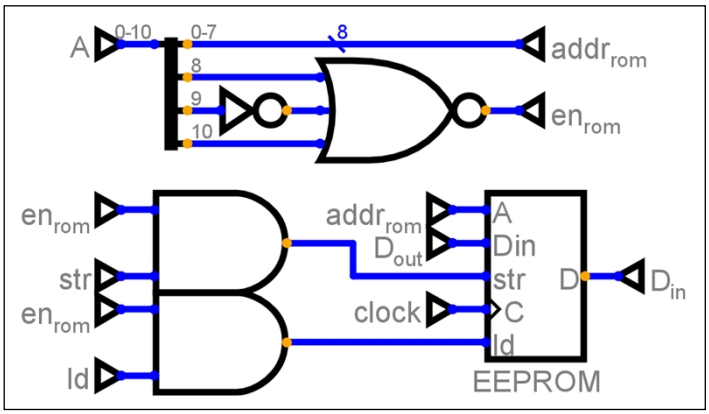
内存模块 11bit 地址中高 3bit 是 0b001，低 8bit 是内部地址空间，那么我们就可以仿照 CPU 内部 ROM 电路，做出一点点修改即可，最终电路如下：



注意内存中的存储介质我们选用的是 RAM（独立端口）模块，这个模块和 EEPROM 的区别在于它不能预先编程，只能在上电以后由程序进行修改。

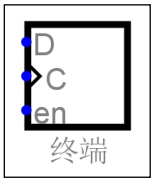
3）磁盘模块

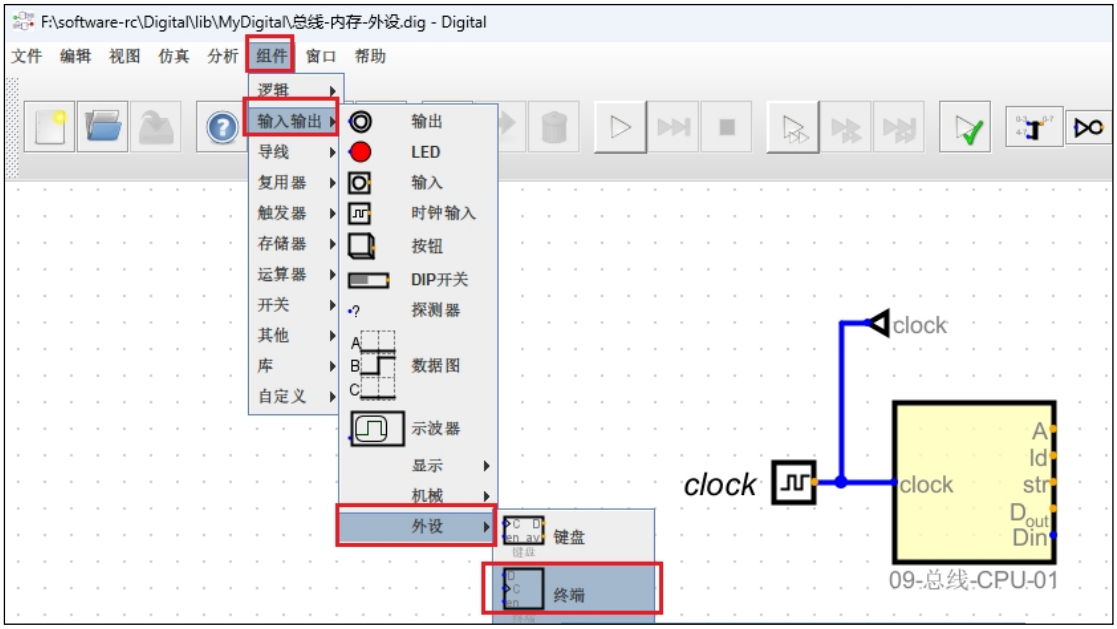
磁盘模块代表的是持久化存储模块，其实和内存模块电路类似，只是高 3bit 相应 0b010，低 8bit 是内部地址，并且存储介质要找一个持久化介质，这里我们还是采用了独立端口的 EEPROM，电路如下：



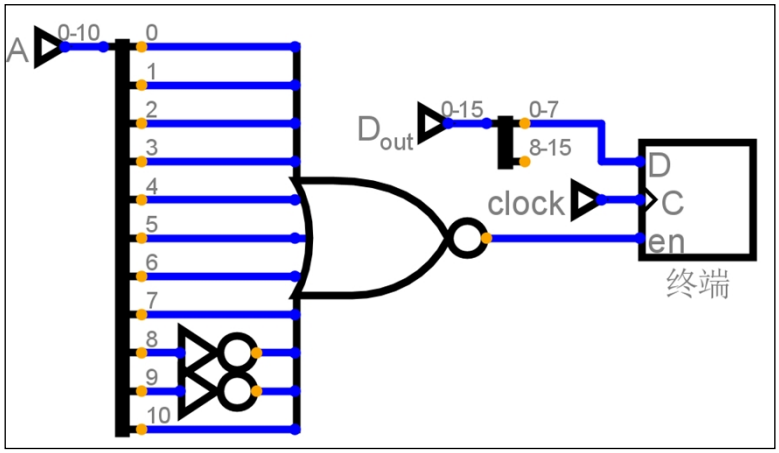
4）终端模块

最终我们需要在终端模块显示一串字符串。Digital 里的终端如下图：



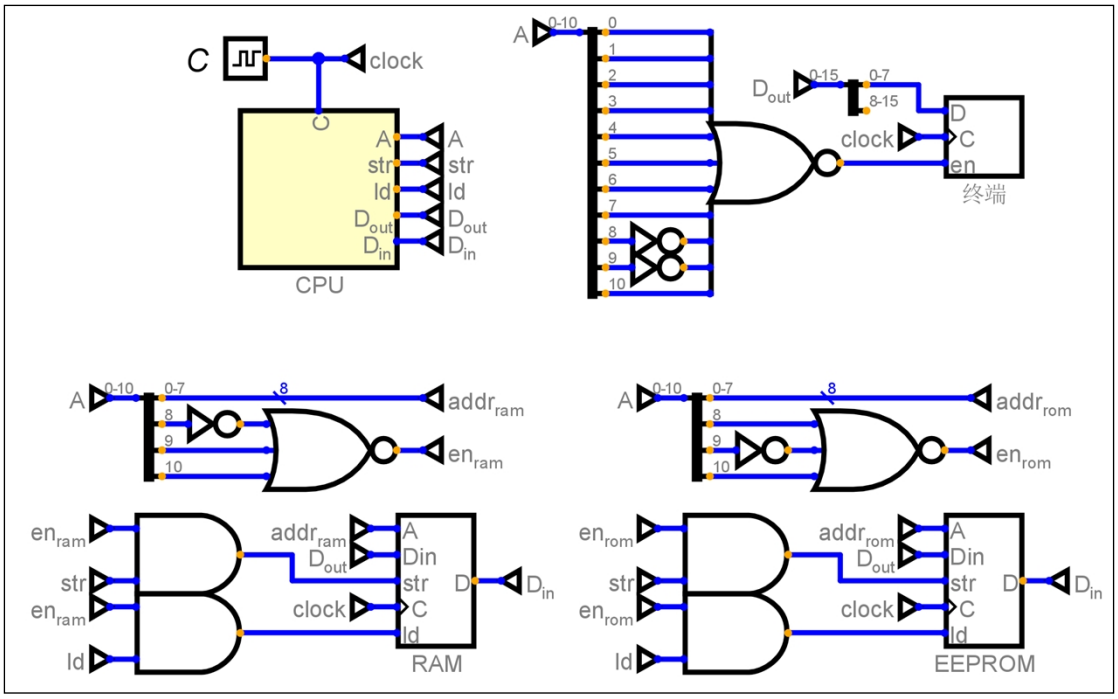


当 en 信号为 1，经过一个时钟周期上升沿，D 的数据就会显示在终端上。唯一需要注意的是 D 的信号为 8bit。我们只需要将寻址信号组成 en 信号，将输出总线的低 8bit 接入 D，即可实现终端，最终电路如下：



这样，我们只需要向 0b011 00000000 地址低 8bit 写入我们想要的 ASCII 码，一个时钟周期之后就会显示出来。

5）最终效果图



注意这里我们没有将其他模块封装，是为了编程方便以及观察最终的效果。这样，CPU 就通过总线和 3 个外部设备相连：内存、硬盘、终端。

第 5 章 设计引导程序

5.1 概述

现代计算机的开机通常要经历一个复杂且精心设计的过程，这个过程叫做引导。引导是计算机启动和运行操作系统的初始步骤，它确保了系统能够正确地加载和执行所需的程序。引导过程随 CPU 架构不同而不同，我们平常接触最多的是 x86 架构的引导，而后面的课程我们主要涉及单片机以及 ARM 架构的引导。今天的课程，我们主要精力放在给我们自己设计的计算机设计一个引导过程。

5.2 引导流程设计

目前我们的计算机总共有三块存储：

- iRom：CPU 内部很小的一块存储，只读；
- 内存：无法持久化保存数据，只有上电之后才能由程序修改；
- 硬盘：存放我们的自定义代码和数据。

CPU 在上电之后首先会去执行 iRom 的代码，这一段代码应该是写死在 CPU 内部不能修改的，所以这一段代码的主要作用应该是引导 CPU 去执行我们的自定义代码。而我们的

自定义代码存放在硬盘上，直接执行效率很低，需要将硬盘里面的数据加载到内存执行。所以 CPU 的 iRom 代码应主要功能应该是：

- 将硬盘中的程序拷贝到内存。
- 跳转到内存地址接着执行。

但这里存在一个问题：我们的程序长度是不固定的，但 iRom 只能加载固定长度的数据到内存。为了容纳更长的程序，这个固定长度只能预先设计的非常大；但这样又会造成引导效率太低。所以为了解决这个问题，我们做这样一个约定：iRom 只加载一段固定长度的短程序并执行，这段短程序再负责加载更长的我们的自定义程序。由于这段短程序我们可以自定义，就不存在之前的问题了。所以最终的设计如下：

- iRom 负责将硬盘前 32 个寻址长度的程序加载到内存并跳转；
- 硬盘前 32 个寻址长度负责加载自定义程序并跳转；
- 最终执行自定义程序。

而硬盘前 32 个寻址地址存放的程序，我们就称之为引导程序，而这 32 个寻址地址，我们称之为引导区。

5.3 程序实现

CPU 寻址地址	外部设备映射
0b000 00000000~0b000 11111111	内部 Rom，简称 iRom （0-255）
0b001 00000000~0b001 11111111	外部扩展 Ram，就是我们的计算机内存（256-511）
0b010 00000000~0b010 11111111	外部扩展 Rom，可以看作是计算机硬盘（512-767）
0b011 00000000	终端地址，向这个地址写数据可以在终端显示（768）

1）iRom

```
// 向寄存器 b 写 0
0   ld_ib      0
// 下两句话是将 16 地址的指令（jmp 10）拷贝到 511 地址，根据我们前面做的映射，511 地址实际上指向内存空间
1   ld_a       16
2   save_a     511
// 下面三句话是将 14 地址的指令（ld_a 512）加上寄存器 b 后拷贝到 509
3   ld_a       14
4   add        0
5   save_a     509
```

```
// 下面三句话是将 15 地址的指令（save_a 256）加上寄存器 b 后拷贝到 510
6   ld_a       15
7   add        0
8   save_a     510
// 跳转到 509。截止现在 509、510、511 有三条指令（ld_a 512、save_a 256、
jmp 10）合起来表示将 512 地址的内容拷贝到 256，并跳转到 10 地址继续执行
9   jmp        509
// 从 511 跳转回来后给 b 累加 1，拷贝下一个地址（从 513 拷贝到 257）。如果 b 等于
32，则结束循环
10  add_ib     1
11  cmp_b      32  （所以只能循环 0-31，32 次，比较后的结果会存储在比较寄存器）
12  jne        3
// 执行到这里说明相等了，证明 32 条内存的指令已经都拷贝到了内存中。
跳转到内存中的 256 地址继续执行
13  jmp        256
// 指令模板，以下三行不会执行
14  ld_a       512
15  save_a     256
16  jmp        10
```

上面的程序可以实现将地址 512-543（硬盘）的内容拷贝到地址 256-287（内存），并跳转到 256（内存）继续执行，而根据主线地址映射，512-543 是硬盘的头 32 地址，256-287 是内存的头 32 地址，所以这一段程序可以将硬盘的程序拷贝到内存并执行。

将这段代码翻译为二进制：

```
v2.0 raw
5000
810
19ff
80e
2800
19fd
80f
2800
19fe
a9fd
6001
a020
b803
a900
a00
1900
a80a
```

保存为“CPU 内置 iROM.hex”，并加载到 CPU 内置的 iRom 上。

2）硬盘代码

硬盘 511-543 之间的代码，已经被上面的引导程序拷贝到了内存中，即便我们硬盘当前只有 17 条指令，也就是只有 511-528 是实际有效的拷贝。但引导程序仍然会将 511-543 之间的程序全部拷贝过去。因为 CPU 中的引导程序是不可修改的。

```
0   ld_ib      0
```

```
1   ld_a      272
2   save_a    511
3   ld_a      270
4   add       0
5   save_a    509
6   ld_a      271
7   add       0
8   save_a    510
9   jmp       509
10  add_ib    1
11  cmp_b     25
12  jne       259
13  jmp       288
14  ld_a      544
15  save_a    288
16  jmp       266
```

这段代码的内容和上面的 iRom 内容类似，主要作用是将硬盘 544-568 地址之间的程序拷贝到内存288-312地址并执行，由于这一段代码我们可以自定义，拷贝的长度是和程序相关的。

```
32  ld_ia     72                H
33  save_a    768
34  ld_ia     101               e
35  save_a    768
36  ld_ia     108               l
37  save_a    768
38  ld_ia     108               l
39  save_a    768
40  ld_ia     111               o
41  save_a    768
42  ld_ia     32                空格
43  save_a    768
44  ld_ia     119               w
45  save_a    768
46  ld_ia     111               o
47  save_a    768
48  ld_ia     114               r
49  save_a    768
50  ld_ia     108               l
51  save_a    768
52  ld_ia     100               d
53  save_a    768
54  ld_ia     33                !
55  save_a    768
```

这段代码主要就是将一个个数字存到 768 地址。根据总线地址映射，这个地址实际上是终端的数据输入结构，换句话说我们每次保存的数据，在下个时钟周期就会在终端显示出来。

将这段代码翻译成二进制文件：

```
v2.0 raw
5000
0910
```

```
19FF
090E
2800
19FD
090F
2800
19FE
A9FD
6001
A019
B903
A920
0A20
1920
A90A
15*0
4848
1b00
4865
1b00
486c
1b00
486c
1b00
486f
1b00
4820
1b00
4877
1b00
486f
1b00
4872
1b00
486c
1b00
4864
1b00
4821
1b00
```

将上面的内容保存为“外置引导 + 程序.hex”，并加载到外置 EEPROM 里面。

3）执行

将时钟频率设置为 50，开启仿真，在仿真的过程中，右键点击 RAM 并监视 RAM 内容的变化。可以看到 RAM 的地址内部不断有数据被拷贝进来并执行。最后会在终端上显示“Hello world！”。当然同学们也可以自己写一段程序来执行，效果是差不多的。

5.4 单片机引导流程

单片机引导流程要比一般 x86 计算机简单。我们可以把单片机视为 ROM 和 RAM 都内置在 CPU 中，只暴露一些输入输出总线接口和外部通信的设备。那么引导过程其实就相当

于在 ROM 上直接执行程序，只有需要用到 RAM 时才会向 RAM 中写入数据，并不需要向一般 PC 引导过程一样需要多步跳转。以我们马上就要学习的 51 单片机为例，我们来看一下单片机的引导程序。

来看如下代码：

```
#include <REG52.h>

sbit led = P1^0;

void main() {
    led = 0;
    while(1);
}
```

这是一个简单的 51 单片机点灯程序。至于这个程序为什么能够点灯，我们这里不去关心，我们着重要看的是，单片机从上电到执行到我们的 main 函数，都经历了哪些过程。当我们开启单片机的调试程序。将程序编译，并开启调试模式，可以看到我们程序编译完成以后的汇编文件如下：

```
                    C_STARTUP:
C:0x0000    020003    LJMP     STARTUP1(C:0003)
                    STARTUP1:
C:0x0003    787F      MOV      R0,#0x7F
C:0x0005    E4        CLR      A
                    IDATALOOP:
C:0x0006    F6        MOV      @R0,A
C:0x0007    D8FD      DJNZ     R0,IDATALOOP(C:0006)
C:0x0009    758107    MOV      SP(0x81),#0x07
C:0x000C    02000F    LJMP     main(C:000F)
                    main:
C:0x000F    C290      CLR      led(0x90.0)
C:0x0011    80FE      SJMP     C:0011
C:0x0013    00        NOP
```

可以看到我们的 mian 函数的指令地址是 0x000F，而第一条指令 0x0000 的作用是跳转到 0x0003，从 0x0003 到 0x000C 这一段代码其实是 51 单片机的固定初始化流程，当这段代码执行完毕后，最后一句 0x000C 的代码就是跳转到 0x000F，而 0x000F 就是我们的 main 函数入口了。这就是 51 单片机的引导过程了，可以看到由于单片机的结构简单，它的引导过程甚至没有我们设计的 CPU 复杂。

第 6 章 常用数字芯片

数字芯片在企业开发中大量使用。大家一定要学会如何购买芯片。熟悉芯片的原理图、封装和查看手册。

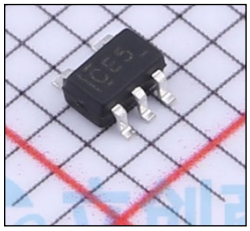
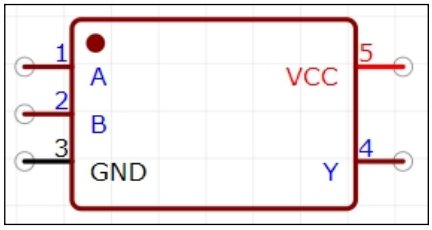
6.1 与或非门

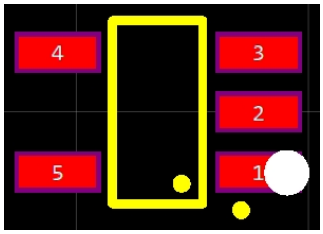
1）与门

（1）购买链接

https://item.szlcsc.com/3422605.html

（2）实物图、原理图、封装



（3）数据手册

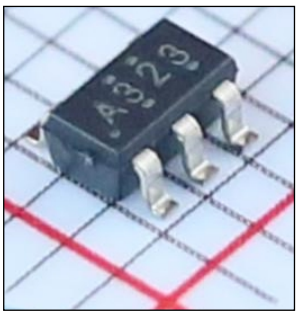
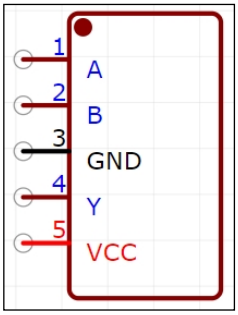
C2983745_逻辑门_SN74LVC1G08DC

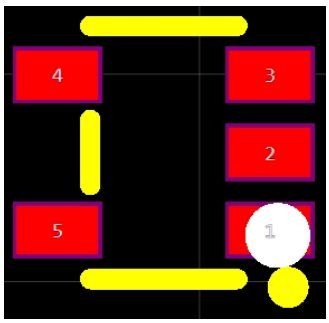
2）或门

（1）购买链接

https://item.szlcsc.com/7947.html

（2）实物图、原理图、封装



（3）数据手册

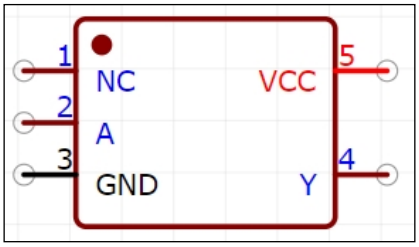
sn74ahc1g32.pdf

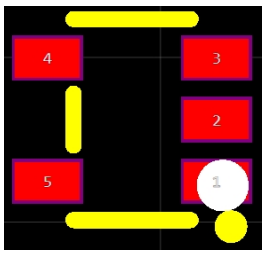
3）非门

（1）购买链接

https://item.szlcsc.com/431038.html

（2）实物图、原理图、封装





（3）数据手册

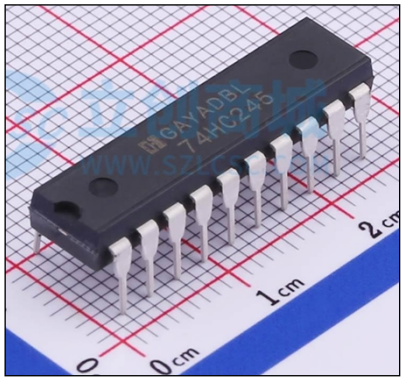
C434069_反相器_SN74LVC1G14DB

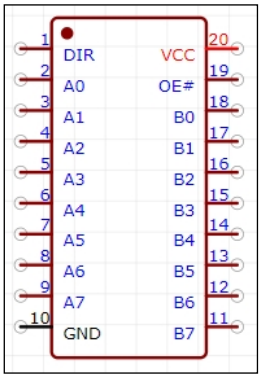
6.2 74HC245N（缓冲器、驱动器）

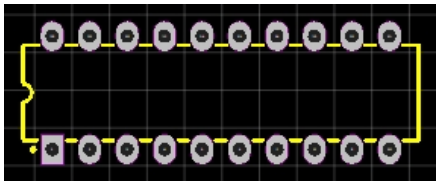
（1）购买链接

https://item.szlcsc.com/520620.html

（2）实物图、原理图、封装







（3）数据手册

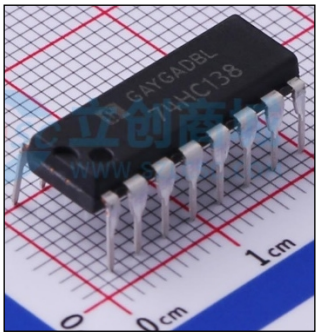
C507182_1A8D0050CB0D66F37C5F

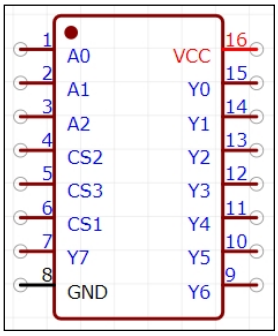
6.3 74HC138（解码器）

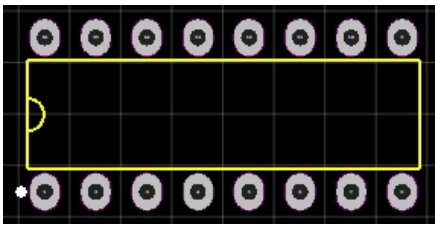
（1）购买链接

https://item.szlcsc.com/520626.html

（2）实物图、原理图、封装







（3）数据手册

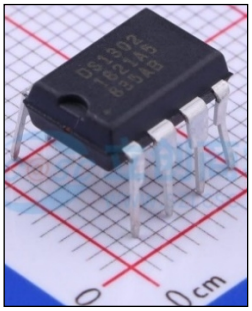
C507188_信号开关-编解码器-多路复

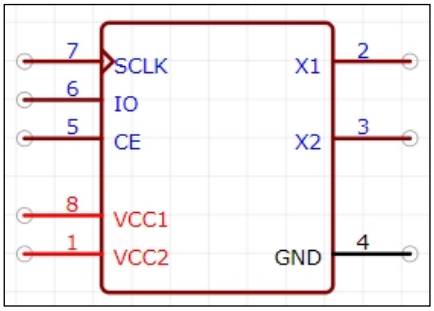
6.4 DS1302（实时时钟）

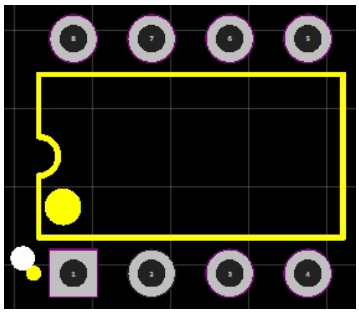
（1）购买链接

https://item.szlcsc.com/15455.html

（2）实物图、原理图、封装







（3）数据手册

C14785_实时时钟RTC_DS1302+_规格