

计算机组成原理

课程设计报告

专业班级：计算机类——19C243

学生姓名：

指导老师：

**中国地质大学计算机学院**

**2023年 6 月**

目录

[实验一、全相联cache设计 4](#_Toc202535227)

[1.1 实验目的 4](#_Toc202535228)

[1.2 主要任务 4](#_Toc202535229)

[1.3实验方案的总体设计 4](#_Toc202535230)

[1.4 实验的实现过程 5](#_Toc202535231)

[1.5电路图及相关说明 6](#_Toc202535232)

[1.5.1 Cache基本结构 6](#_Toc202535233)

[1.5.2 CAM及其并发查找逻辑 9](#_Toc202535234)

[1.5.3 读命中逻辑 10](#_Toc202535235)

[1.5.4 读缺失逻辑 11](#_Toc202535236)

[1.5.5 块写入逻辑 12](#_Toc202535237)

[1.6实验数据测试 12](#_Toc202535238)

[实验二、4路组相连cache设计 13](#_Toc202535239)

[2.1实验目的 13](#_Toc202535240)

[2.2主要任务 13](#_Toc202535241)

[2.3实验方案的总体设计 14](#_Toc202535242)

[2.4实验的实现过程 15](#_Toc202535243)

[2.5电路图及相关说明 16](#_Toc202535244)

[2.5.1 Cache基本结构 16](#_Toc202535245)

[2.5.2 CAM及其并发查找逻辑 17](#_Toc202535246)

[2.5.3 读命中逻辑 17](#_Toc202535247)

[2.5.4 读缺失逻辑 17](#_Toc202535248)

[2.5.5 块写入逻辑 18](#_Toc202535249)

[2.6实验数据测试 20](#_Toc202535250)

[实验三、2路组相联cache设计 20](#_Toc202535251)

[3.1实验目的 20](#_Toc202535252)

[3.2主要任务 20](#_Toc202535253)

[3.3实验方案的总体设计 21](#_Toc202535254)

[3.4实验的实现过程 22](#_Toc202535255)

[3.5电路图及相关说明 22](#_Toc202535256)

[2.5.1 Cache存储部分基本结构 22](#_Toc202535257)

[3.5.2 CAM及其并发查找逻辑 23](#_Toc202535258)

[3.5.3 读命中逻辑 23](#_Toc202535259)

[2.5.4 读缺失逻辑 24](#_Toc202535260)

[3.5.5块写入逻辑 25](#_Toc202535261)

[3.6实验数据测试 25](#_Toc202535262)

[实验四、单周期MIPS CPU设计 26](#_Toc202535263)

[4.1实验目的 26](#_Toc202535264)

[4.2主要任务 26](#_Toc202535265)

[4.3 实验方案的总体设计 27](#_Toc202535266)

[4.3.1实验规划 27](#_Toc202535267)

[4.3.2总体设计 27](#_Toc202535268)

[4.4 实验的实现过程（结合电路图说明） 28](#_Toc202535269)

[4.4.1指令解析 28](#_Toc202535270)

[4.4.2实现地址转移逻辑 29](#_Toc202535271)

[4.4.3单周期硬布线控制器 29](#_Toc202535272)

[4.4.4单周期CPU总体结构图 31](#_Toc202535273)

[4.5实验数据测试 32](#_Toc202535274)

[4.5.1头歌平台测试结果： 32](#_Toc202535275)

[4.5.2八条指令测试结果： 32](#_Toc202535276)

[实验五、微程序地址转移逻辑设计 35](#_Toc202535277)

[5.1实验目的 35](#_Toc202535278)

[5.2主要任务 35](#_Toc202535279)

[5.3实验方案的总体设计 36](#_Toc202535280)

[5.4实验的实现过程（结合电路图说明） 36](#_Toc202535281)

[5.4.1微程序地址转移逻辑自动生成 36](#_Toc202535282)

[5.4.2使用组合电路分析功能连线 37](#_Toc202535283)

[5.5实验数据测试 37](#_Toc202535284)

[实验六、MIPS微程序CPU设计 38](#_Toc202535285)

[6.1实验目的 38](#_Toc202535286)

[6.2主要任务 38](#_Toc202535287)

[6.2.1完成指令集设计 38](#_Toc202535288)

[6.2.2完成电路逻辑 38](#_Toc202535289)

[6.3实验方案的总体设计 38](#_Toc202535290)

[6.4实验的实现过程（结合电路图说明） 39](#_Toc202535291)

[6.4.1从单周期到多周期 39](#_Toc202535292)

[6.4.2指令集的生成 39](#_Toc202535293)

[6.4.3多周期CPU总体结构图 41](#_Toc202535294)

[6.5电路图及其相关说明 41](#_Toc202535295)

[6.5.1 取指令阶段数据通路 41](#_Toc202535296)

[6.5.2 译码/取数阶段数据通路 42](#_Toc202535297)

[6.5.3其他指令及其指令执行 43](#_Toc202535298)

[6.6实验数据测试 44](#_Toc202535299)

[6.6.1头歌平台测试结果： 44](#_Toc202535300)

[6.6.2八条指令单独测试用例的测试结果 45](#_Toc202535301)

[实验心得 49](#_Toc202535302)

# 实验一、全相联cache设计

## 1.1 实验目的

本实验旨在**理解全相联映射Cache的基本原理和工作机制**。全相联Cache作为Cache设计中的一种极端形式，允许主存中的任意块映射到Cache中的任意位置，从而最大化了Cache空间的利用率，但也带来了更高的硬件复杂性。同时，实验还致力于**掌握全相联Cache的地址划分和查找方法**，明确主存地址如何被解析为标记和块内偏移，以及这些信息在Cache查找过程中的作用。在此基础上，我进一步**实现全相联Cache的并行比较机制**，通过同时比较所有Cache行的标记，来高效判断数据是否命中，尽管这种并行比较需要大量的比较器硬件。最后，本实验也着重于**熟悉替换算法在全相联Cache中的应用**，特别是LRU等高效替换算法如何在Cache满时选择最佳的被替换块，以维持较高的命中率。

## 1.2 主要任务

为了达成上述实验目的，我首先设计了16位主存地址映射到8行全相联Cache的映射方案，明确了主存地址与Cache行之间的对应关系，这是整个Cache系统设计的基础。

接着，我实现了**地址划分**：将16位主存地址分为14位标记（Tag）和2位块内偏移（Offset），确保Cache能够正确解析和识别数据在主存中的位置及其在Cache块中的具体偏移。为了实现高效查找，我构建了8路**并行比较器**，这意味着可以同时比较所有Cache行的标记，从而在一次查找中快速判断数据是否命中，极大地提升了查找效率。

在Cache行满时，为了有效管理Cache空间，我实现了LRU（Least Recently Used）替换算法，确保Cache在需要替换旧数据块时，能够淘汰近期最少使用的块，以优化命中率。这一个任务又可以分为两个子任务：1. 使用LRU算法找到近期最少使用的块（具体实现时也需要考虑到如果存在还未使用的块，那么选择该块）；2. 实现写入逻辑，在对应的块写入准备就绪的数据块。

最终，所有这些设计都被整合，我在Logisim中搭建了完整的全相联Cache电路，这包括了用于存储标记的8×14位寄存器组成的标记存储阵列，以及用于存储实际数据块的8×32位寄存器组成的数据存储阵列。并行比较的关键在于8个14位比较器组成的并行比较电路。此外，我还集成了优先编码器来处理可能的匹配情况（尽管全相联Cache理论上只有唯一匹配），并完善了LRU替换逻辑，使其能够准确跟踪和更新Cache行的使用情况，从而实现正确的替换决策。

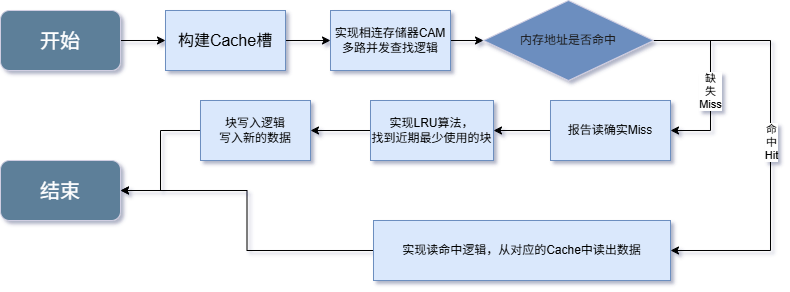
## 1.3实验方案的总体设计

为了实现全相联cache高速缓冲存储器，我设计了下面的五个模块，分别对应了上述提到的具体任务：

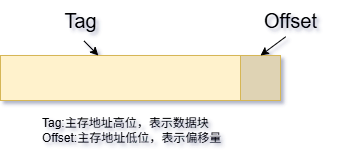
* 构建Cache槽，实质是相连存储器。Cache负责存储数据块和保存辅助信息，如写使能信号、Valid位、命中信号Li等，并提供控制读取和辅助信息的接口。
* 实现相连存储器CAM的多路并发比较逻辑。比较地址中的Tag和每一个Cache槽的有效位信息和。
* 实现读命中逻辑。当内存地址命中Hit时，直接从Cache中读出。
* 实现读缺失逻辑。当内存地址没有命中Miss时，需要报告读缺失Miss；同时还需要实现LRU算法，找到符合条件的槽，并交给块写入逻辑实现写操作。
* 实现块写入逻辑。收到Miss信号时，要在对应的存储块中写入新的数据块。

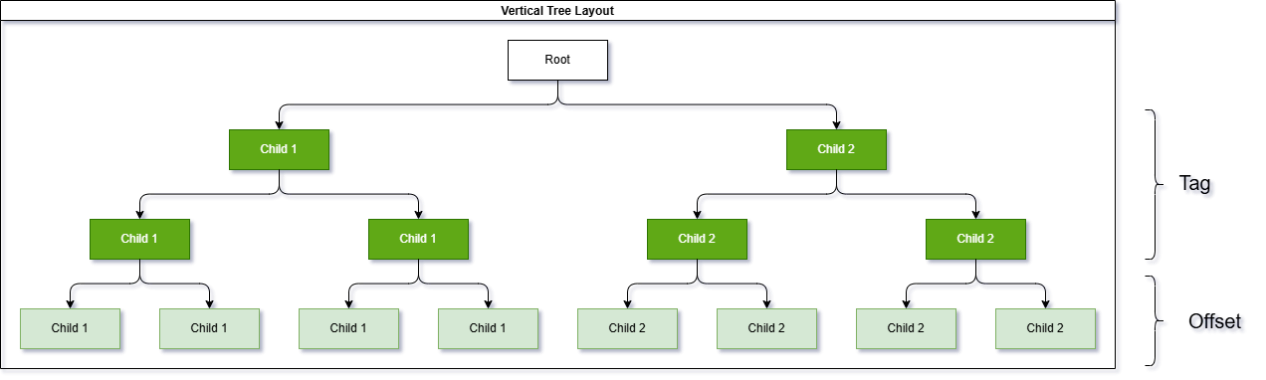
通过这五个模块的紧密协作，全相联Cache能够实现高效的数据存储、查找、命中处理、缺失处理及数据更新，从而实现全相联Cache的主要功能。

## 1.4 实验的实现过程

**本实验严格遵循Cache的实际工作流程进行设计与实现，各功能模块的开发顺序与Cache处理访存请求的时序逻辑完全吻合。**从基础的存储结构构建开始，逐步实现了地址解析、并行查找、命中判断等核心功能，最后完成缺失处理和替换算法，整个实现过程真实模拟了Cache硬件在实际运行时的完整工作序列。这样的实现顺序不仅确保了各功能模块之间的逻辑连贯性，也使我在开发过程中能够更加直观地理解Cache系统从接收到访存请求到最终完成数据访问的完整处理流程。每个阶段的实现都严格对应Cache实际工作中的特定环节，从而保证了最终实现的Cache系统能够准确反映真实硬件的行为特征。下图所示为Cache系统的工作流程示意图，该图完整呈现了Cache处理访存请求的标准工作顺序。该流程图不仅清晰展示了Cache的核心工作流程，也与本实验的实现顺序保持高度一致，为理解Cache的工作原理提供了直观的参考依据。

整个Cache系统的工作流程将这些模块有机整合：接收访存请求后首先解析地址，然后并行查找所有Cache行并判断命中情况。命中时直接执行读写操作，未命中时则启动缺失处理流程，包括选择替换行、从主存读取数据、更新Cache内容等步骤，最终完成访存请求。通过这一系列精心设计的硬件逻辑，我实现了一个高效可靠的全相连Cache系统。

Cache的本质仍然是一个存储系统，最核心的功能是根据给出的地址找到对应的数据。每个字节地址都是一个16位数据，0-1位是offset，表示在4字节的数据块中的偏移量；2-15位是tag，是在内存中的数据块位置。如果考虑一个无穷大且速度无穷快的存储系统，也就是cache的理想情况，那么每给出一个内存地址，都会先根据tag找到对应的内存块，再根据offset找到具体的字节地址。

一种形象的理解是：一个地址表示的是一个满二叉树中叶节点位置，地址高位决定大方向，地址低位则是这棵满二叉树上末端的几支中的地址。虽然cache的硬件设计并非利用了满二叉树的思想，但这是二进制系统在数学上结构特点的通用形象表示。

我将全连接Cache的实现过程分为下面五步，以此实现：

1. 构建存储区域和接口：实现八个cache槽，用来存储，读出和写入数据，并提供相应的接口。
2. 实现并发查找逻辑，原理依据是相联存储器。
3. 实现读命中逻辑，完成电路，直接将读到的数据输出。
4. 实现读缺失逻辑，利用算法找到最合适写入或替换成新数据（从主存中引入）的cache槽，报告给 块写入逻辑。
5. 实现块写入逻辑，将主存中的数据块转移到合适的cache槽中（由读缺失逻辑提供）。

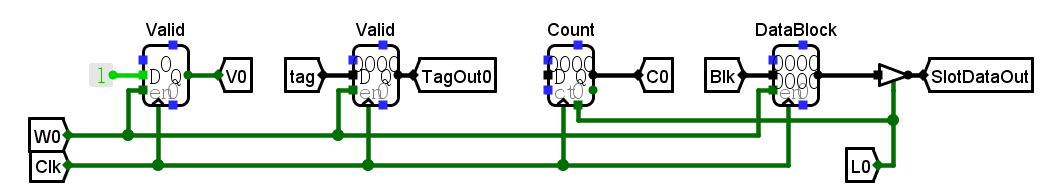
每一步的详细实现会在`1.5 电路图及相关说明`中结合电路图说明。

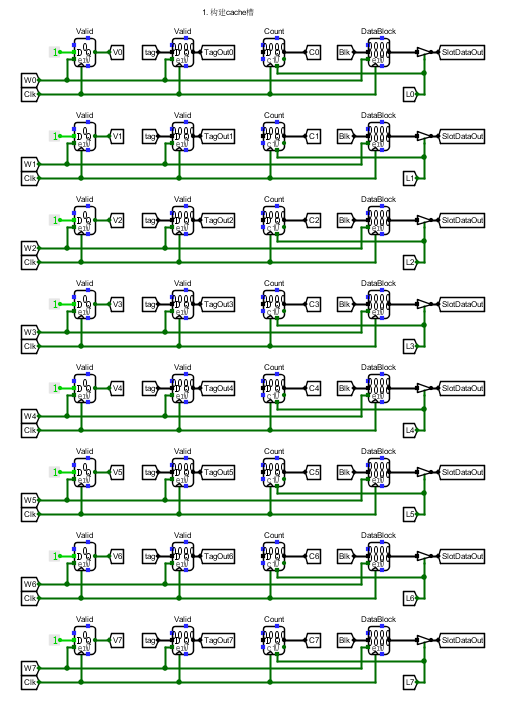
## 1.5电路图及相关说明

### 1.5.1 Cache基本结构

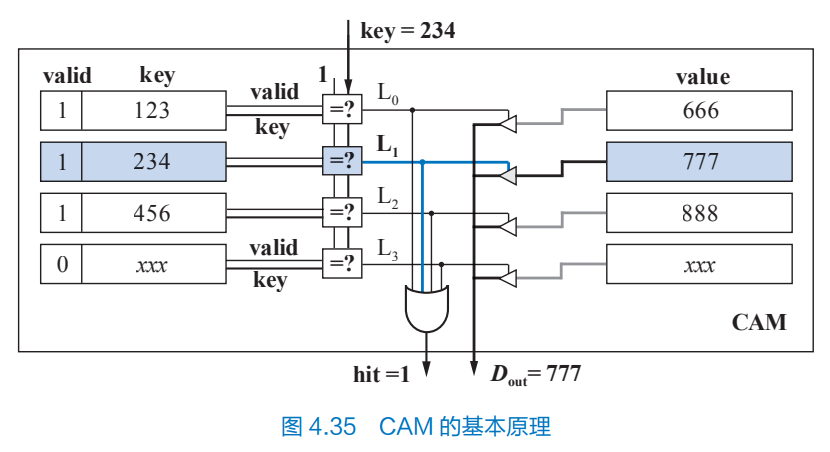
首先进行了Cache基本结构的构建，我设计了由多个Cache行组成的存储阵列，每个Cache行包含有效位、标签位和数据块三个主要部分，以及用于辅助LRU算法实现的最近访问时长计数器。采用全相连映射方式意味着主存中的任何数据块可以被放置到Cache的任何位置，这要求实现能够并行访问所有Cache行的硬件结构。初始化时将所有Cache行的有效位清零，表示初始状态下Cache为空。其要点如下：

1. 有效位：标记当前Cache行是否存储有效数据。
2. Tag：存储主存地址的高位部分，提供输入输出通道，用于写入新的Tag和输出Tag以供项链存储器比较。
3. Count：支持LRU替换策略，记录Cache行的"年龄"。命中时：命中行的计数器清零，其他行计数器递增。缺失时：选择计数值最大的行（最久未使用）替换。这些操作逻辑都提供接口，在其他电路逻辑判断之后可以直接调用接口实现。
4. DataBlock数据块：存储从主存加载的数据块内容。SlotDataOut为数据输出端口，位宽等于数据块大小。读命中时：直接输出对应数据块。
5. 写使能信号Wi：是实现写入逻辑的接口，允许写入时，置Wi位为1，则同步更新有效位，Tag和数据块内的信息。
6. 命中信号Li：标记当前访问是否命中该行，命中时则置Li为1，便可以执行对应的操作，包括Count清零和利用三态门选择对应的数据块输出。



将八个这样的cache组合起来，并完善合适的标号，就得到完整的cache存储结构和其接口。

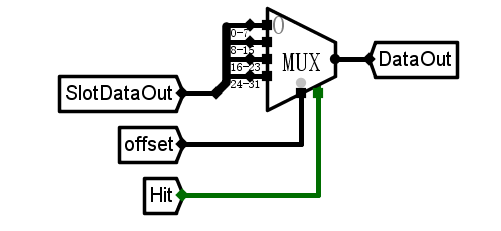
### 1.5.2 CAM及其并发查找逻辑

Cache的并发比较逻辑本质上就是实现一个相联存储器CAM，下图展示了相联存储器的工作原理。转换成电路逻辑就是：分别判断tag值是否等于目标地址，valid是否等于1。通过为每个Cache行配备独立的比较器，系统能够在一个时钟周期内同时比较所有行的标签与访存地址的标签部分。无论结果如何，都将信号传导到三态门，从而决定是否允许将data输出成，同时对每一行的比较结果做或运算，得到Hit信号。Hit信号为1时则判定命中，为了方便后续处理，我还对Hit信号做非运算，得到Miss信号。

### 1.5.3 读命中逻辑

读命中逻辑要解决的问题有三个：

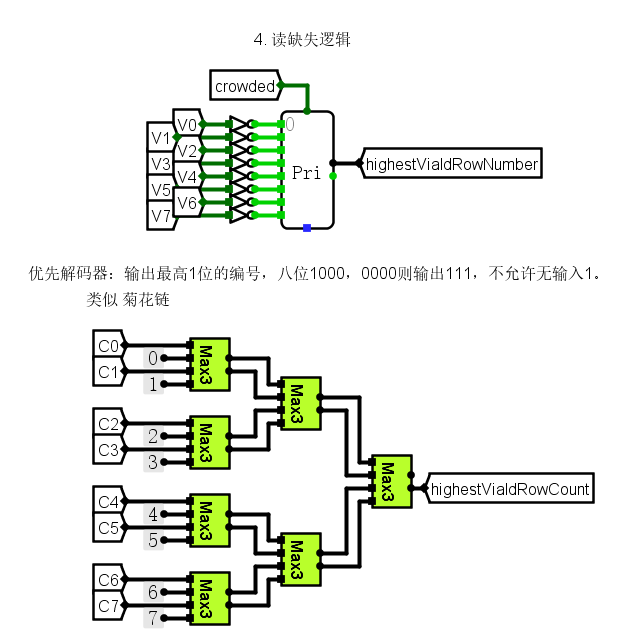
1. 当且仅当Hit命中时才需要调用读命中逻辑；
2. 根据Offset从数据块中选择正确的字节。
3. 辅助LRU算法的实现，将计数器Count置零。

前两点可以通过分线器+多路选择器实现。先将数据块分线成4×8段共4段，利用多路选择器根据Offset选择合适的数据。使用带有使能端的多路选择器，与Hit相连，从而利用Hit对电路进行控制。

第三点利用Count计数器的清零接口就可以实现，只需将匹配成功或者Hit信号传给计算器的清零接口就可以。这里选择直接传递Li。

电路实现如右图所示。

### 1.5.4 读缺失逻辑



当处理器尝试从缓存中读取数据，但未找到所需数据（即发生读缺失）时，系统会生成一个缺失信号。这个信号会触发缺失处理流程，目的是从主存中加载所需的数据块到缓存中。

在缺失处理流程中，首先需要确定哪个缓存行将被用来存储从主存加载的数据块。这涉及到替换算法的实现，以决定哪个现有的缓存行应该被新加载的数据块替换。

我使用了优先解码器来实现最基本的替换逻辑，即还存在未占用的cache槽。这个优先解码器类似于菊花链结构，用于输出最高1位的编号。具体来说：它接收多个输入（V0到V7），并输出最高有效位（Pri）。这个输出用于确定哪个缓存行应该被替换。例如，如果输入中V7是最高有效位，则输出为111，表示第7行应该被替换。

当所有cache都被占用时，首先优先阶码器会报告拥挤信号Crowded（这是用来启动块写入逻辑），然后根据LRU算法，选择最近最少使用的cache槽来进行替换，即替换掉选择计数器值最大的槽中的数据。

在这一步中，只需要选择出需要替换的行即可，具体的替换操作则由块写入逻辑完成。

### 1.5.5 块写入逻辑



块写入逻辑负责处理缺失时的数据调入操作。

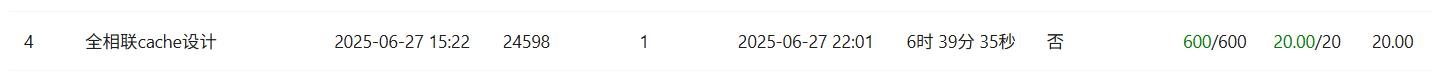
首先，读缺失逻辑电路会产生表示是否拥挤的Crowded信号，我使用多路选择器根据Crowded信号选择使用最优先空行还是LRU算法决策出的原本有数据的行。这样就会产生一个行号。再利用对这个行号做一次解码，便能产生对应行的写使能信号。

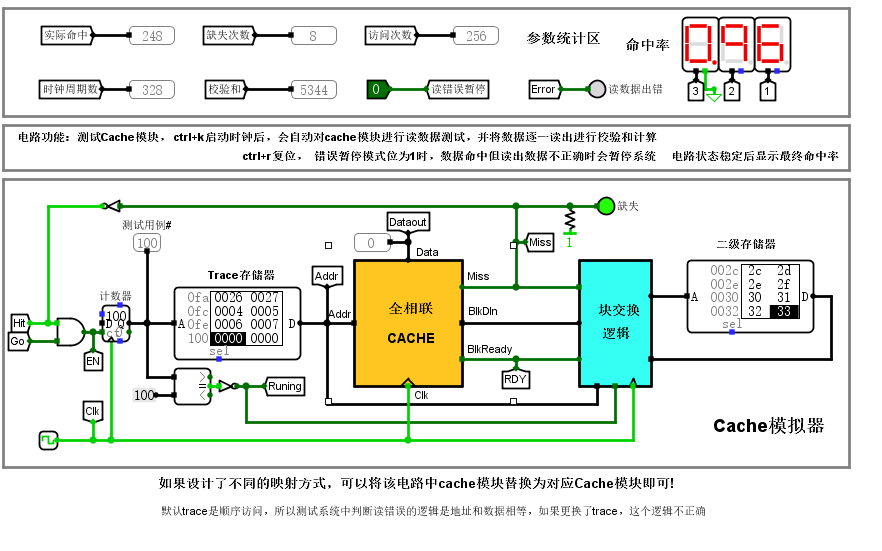
注意，本实验还提供了BlkReady信号（表明数据是否可以写入），因此我对Decd增加一个使能端口，使得只有数据准备就绪时才会进行写入操作。

最后，对应行的写使能信号通过Cache槽的接口实现对特定Cache槽数据的替换。

## 1.6实验数据测试

EduCoder平台通过的截图：



Cache自动测试电路的运行截图如下：

# 实验二、4路组相连cache设计

## 2.1实验目的

本实验旨在掌握4路组相联Cache的基本原理与设计方法。组相联Cache通过折中全相联与直接映射的优点，既降低了硬件复杂度，又保持了较高的命中率。实验需实现以下目标：

1. 地址划分与映射机制：理解主存地址如何划分为标记（Tag）、组索引（Index）和块内偏移（Offset），明确数据块在Cache中的组内灵活映射特性。
2. 并行比较与命中判断：设计4路并行的组内比较逻辑，通过同时比较组内所有行的标记位，快速判断数据是否命中。
3. 替换算法实现：在组内冲突时，应用LRU（Least Recently Used）算法选择替换目标，优化Cache利用率。

## 2.2主要任务

为实现上述目标，除了构建基础Cache槽并提供读出写入等信息接口以外，实验分还需要完成以下核心任务：

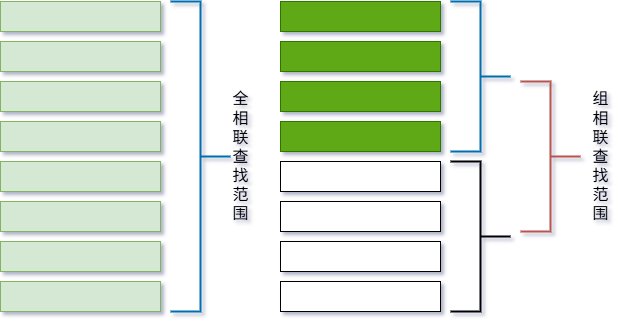
1. **地址划分设计**：将16位主存地址划分为13位标记（Tag）、1位组索引（Index）和2位块内偏移（Offset），确保组索引能定位4个候选Cache行。
2. **组内并行比较**：依照CAM设计并行比较逻辑电路，实现组内标记的并行匹配，生成命中信号（Hit）和行选择信号（Line Select）。
3. **LRU替换策略**：为每组维护一个Count计数器，记录行访问顺序，替换时选择组内最久未使用的行。
4. **数据读写整合**：在Logisim中构建由2组×4行组成的存储阵列，集成标记存储、数据块存储、比较电路及LRU更新逻辑，形成完整Cache系统。

## 2.3实验方案的总体设计

为了实现4路组相联cache高速缓冲存储器，我设计了下面的五个模块，分别对应了上述提到的具体任务：

* 构建Cache槽，实质是相连存储器。Cache负责存储数据块和保存辅助信息，如写使能信号、Valid位、命中信号Li等，并提供控制读取和辅助信息的接口。
* 实现相连存储器CAM的多路并发比较逻辑。比较地址中的Tag和每一个Cache槽的有效位信息和。
* 实现读命中逻辑。当内存地址命中Hit时，直接从Cache中读出。
* 实现读缺失逻辑。当内存地址没有命中Miss时，需要报告读缺失Miss；同时还需要实现LRU算法，找到符合条件的槽，并交给块写入逻辑实现写操作。
* 实现块写入逻辑。收到Miss信号时，要在对应的存储块中写入新的数据块。

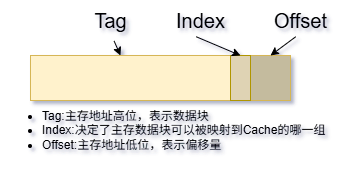
通过这五个模块的紧密协作，4路组相联Cache能够实现高效的数据存储、查找、命中处理、缺失处理及数据更新，从而实现4路组相联Cache的主要功能。

 4路组相联和全相联的区别（仅考虑这里的两组4路组相连）在于，在判断读缺失逻辑时，只需要考虑某一组内的替换位置进而在块写入时也只需要考虑在某一组内的写入位置。这样的设计使得组相联相比全相联在少量比较器下仍保持较高命中率，适合现代CPU设计。

## 2.4实验的实现过程

本实验严格遵循**组相联Cache**的实际工作流程进行设计与实现，各功能模块的开发顺序与Cache处理访存请求的时序逻辑完全匹配。从**存储结构构建**开始，逐步完成**地址解析、组选择、并行比较、命中判断、缺失处理及替换策略**等核心功能，整个实现过程模拟了真实硬件的工作机制。这样的实现顺序不仅保证了模块间的逻辑一致性，也使我能够直观理解Cache从接收访存请求到完成数据访问的完整流程。每个阶段的实现均对应Cache实际工作的关键环节，确保最终系统能准确反映硬件行为。实现流程与全相联Cache基本一致，可参照1.4中的工作流程图，它为理解组相联映射原理提供了清晰的参考。

Cache的核心功能仍是**通过地址快速定位数据**。

在4路组相联设计中，每个16位地址被划分为：

**4路组相联Cache的实现步骤**

1. 构建存储结构与接口：
   * 实现2组×4行的Cache槽，每行包含Valid位、Tag、DataBlock（4字节）及LRU计数器。
   * 提供读写接口（如写使能信号、数据输出端口）。
2. 实现组选择与并行比较逻辑：
   * 通过Index（1位）选择目标组，激活组内4行的Tag比较器。
   * 4路并行比较输入Tag与各行Tag，生成命中信号（Hit）及行选择信号（Line[1:0]）。
3. 实现读命中逻辑：
   * 命中时，根据Offset（2位）从DataBlock中选择对应字节。
   * 更新LRU状态：命中行的计数器清零，其他行递增。
4. 实现读缺失与替换逻辑：
   * 若未命中，通过LRU策略选择组内最久未使用的行（计数器最大值）。
   * 生成替换行号，触发主存数据加载请求。
5. 实现块写入逻辑：
   * 从主存读取数据块后，根据替换行号更新目标行的Tag、DataBlock及Valid位。
   * 重置该行的LRU计数器，标记为最新使用。

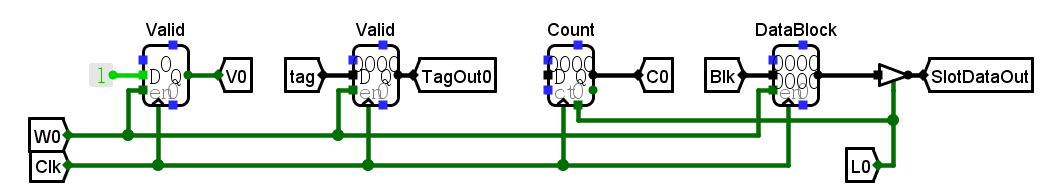
通过这种设计，4路组相联Cache在**硬件成本、命中率和访问速度**之间取得了最优平衡，成为现代CPU Cache的典型方案。

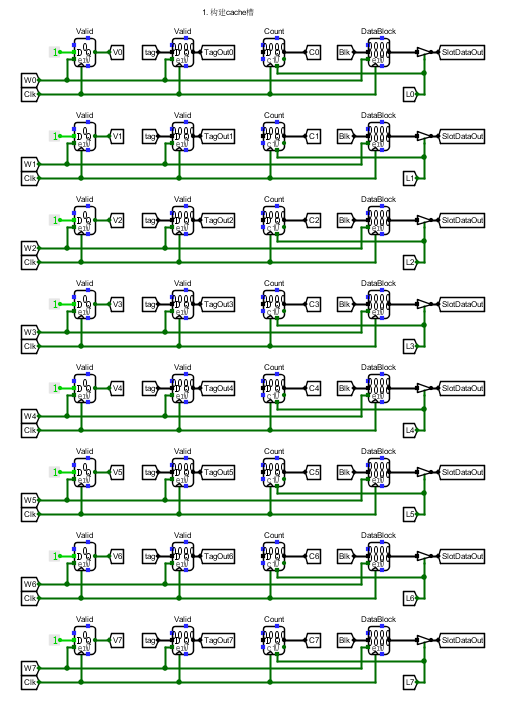
每一步的详细实现会在`2.5 电路图及相关说明`中结合电路图说明。

## 2.5电路图及相关说明

### 2.5.1 Cache基本结构

这部分与全相联Cache槽的设计完全一致。



将八个这样的cache组合起来，并完善合适的标号，就得到完整的cache存储结构和其接口。

### 2.5.2 CAM及其并发查找逻辑

在4路组相联Cache的CAM及其并发查找逻辑设计中，核心思想是通过组索引（Index）选择目标组后，在组内实现并行标记比较。当16位主存地址输入时，系统首先解析出1位Index字段，通过1-2译码器选中对应的Cache组，同时提取13位Tag用于后续比对。该组内的4个Cache行会同时输出各自的有效位（Valid）和标记位（），这些信号与输入Tag通过4个独立的13位比较器进行并行比较。比较器的输出会与对应行的Valid位进行逻辑与运算，只有当Valid=1且Tag匹配时，才会生成该行的命中信号（）。所有命中信号通过或门汇总为最终的全局命中信号（Hit），同时通过优先编码器确定具体的命中行号（Line Select）。这种设计只需4个比较器即可完成组内全并行查找，既保留了全相联的灵活性，又大幅降低了硬件开销。

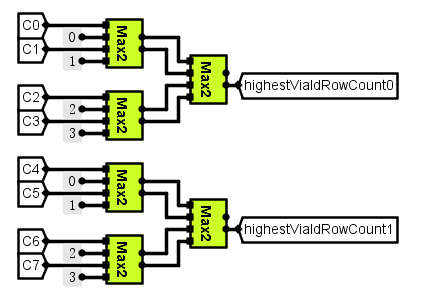
值得注意的是，Index的选择逻辑使得不同组的比较器可以分时复用，进一步优化了电路规模，而有效位的引入确保了无效数据不会产生误命中。整个并发查找过程在一个时钟周期内完成，包括组选择、标记比较和命中判断三个关键阶段，完美体现了组相联Cache在访问速度和硬件成本间的平衡优势。

### 2.5.3 读命中逻辑

这部分的电路实现和全相联完全一致，也就是说，命中时直接从数据块中按照Offset选择对应字节地址的数据即可。

### 2.5.4 读缺失逻辑

 首先我们可以明确一点，在每一个组内，相当于实现了一个全相联。在本实验这个具体的任务中，一共两组每组4行，我只需要对每一组做一个全相连同样的实现就可以。唯一的区别在于，可以用Index连接到优先选择器的使能端（不连接也不影响，只要在后面补充有关的选择逻辑），保证只在对应的组中选择。

LRU算法的实现部分也与全相联一致，只需要在每一组内分别递归实现即可。

### 2.5.5 块写入逻辑

在4路组相联Cache的多组协同工作环境下（本设计共有两组，每组4路），块写入逻辑需要建立高效的组间协调机制。地址结构采用13位Tag + 1位Index + 2位Offset的划分方式。

当发生读缺失（Read Miss）并需要进行块写入时，系统首先通过地址中的1位Index字段确定目标组。这个Index字段将访存请求精准路由到对应的组处理单元。如图所示，有两个独立的组（表示为组0和组1）处理单元。每个组都独立维护自己的LRU状态机（由 highestVialdRowNumber 和 highestVialdRowCount 等信号隐含，用于选择替换的行）和空闲行检测电路（由 Crowded 信号表示拥挤状态），但所有组共享统一的数据总线和控制信号，确保主存数据可以高效分发到各个组。

多组环境下，引入了分布式仲裁和协同写入机制：

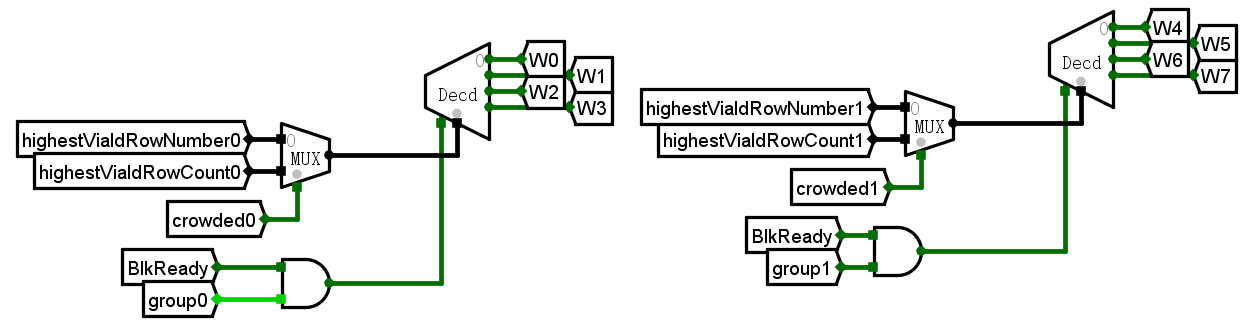
1. **组内行选择与多路复用：**

每个组（例如组0）内部，highestVialdRowNumber0 和 highestVialdRowCount0 信号输入到一个多路选择器（MUX）。这些信号可能用于选择一个最优的替换行（例如基于LRU策略），或者选择一个空闲行。crowded0 信号也作为MUX的一个输入，它指示了当前组是否已经拥满（即没有空闲行）。MUX的输出信号与 BlkReady0 和 group0 信号的AND操作结果共同驱动一个译码器（Decd）。译码器（Decd）根据输入信号，生成具体的写入控制信号（例如 W0, W1, W2, W3 对于组0；W4, W5, W6, W7 对于组1），这些信号用于选择并使能目标行进行数据写入。图中可以看出，每个译码器有4个输出，对应一个组内的4个路。

**2． 分布式仲裁与块就绪信号：**

crowded0 和 crowded1 信号指示了各自组的拥挤状态。这些状态信息会被汇总（图中未直接画出汇总逻辑，但其概念是存在的）到中央控制单元或通过分布式逻辑进行处理。控制单元可能采用优先级编码器（或更简单的逻辑）处理并发写入请求，确保在多个组同时需要更新时，能够按照预设的优先级顺序依次处理。BlkReady 信号是一个关键的广播信号，当主存数据就绪时，它会广播给所有待写入组。每个组的写入使能条件由 BlkReady 与各自的 group 信号进行逻辑AND操作决定（例如，对于组0是 BlkReady AND group0；对于组1是 BlkReady AND group1）。这里的 group0 和 group1 信号是根据地址中的Index位选择的结果。这意味着，尽管 BlkReady 是广播的，但只有被Index选中的目标组（即其 groupX 信号为高）才会真正使能其写入译码器，从而执行写入操作。

**3. 组间协同与状态同步：**

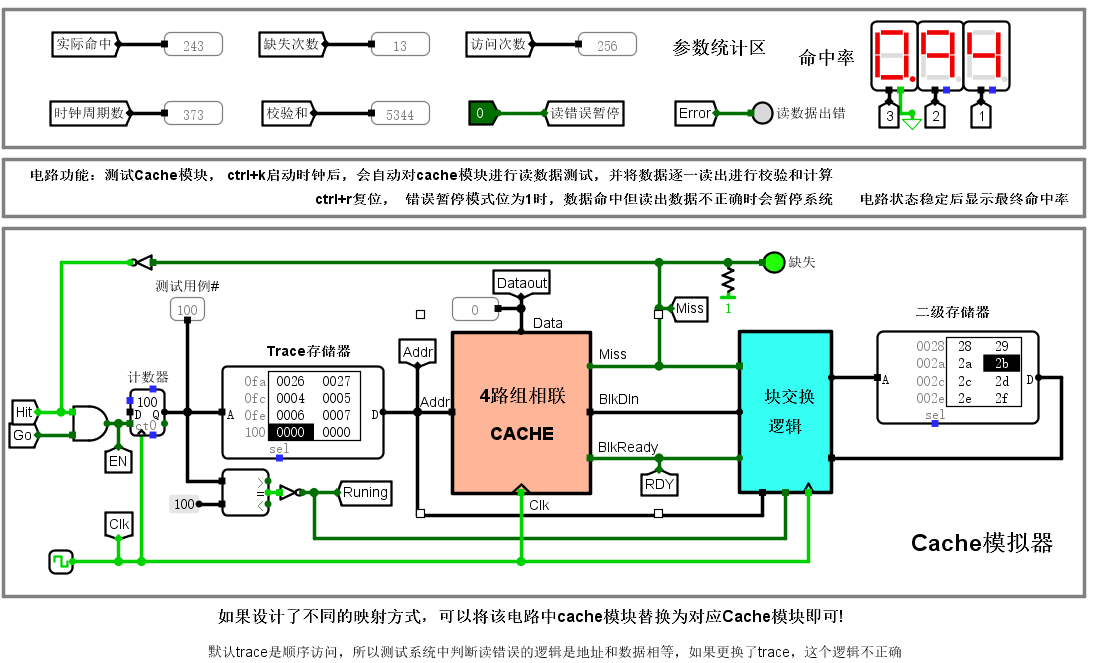
所有组共享同一个LRU策略实现（通过共享 highestVialdRowNumber 和 highestVialdRowCount 的生成逻辑或类似机制），但每个组独立维护自己的访问历史记录（以确保其内部LRU状态机的正确更新）。当某个组执行写入操作时，其状态更新（例如LRU位的更新）会通过组间同步信号（图中未明确画出，但这是隐式的）通知相关组，确保替换策略的一致性。数据总线采用多路复用设计，可以同时支持多个组的并行写入（如果控制器允许，例如在不同时钟周期内对不同组进行写入，或者通过更复杂的数据路径实现），大幅提升了整体吞吐量。这种设计既保留了单组处理的简洁性，又通过巧妙的资源共享机制（如 BlkReady 广播）和分布式控制（如 groupX 选择）实现了多组协同工作，使4路组相联Cache能够高效处理密集的访存请求，优化了写操作的性能和协调性。

## 2.6实验数据测试

EduCoder平台通过的截图：

****

Cache自动测试电路的运行截图：



# 实验三、2路组相联cache设计

## 3.1实验目的

本实验旨在掌握2路组相联Cache的基本原理与设计方法。组相联Cache通过折中全相联与直接映射的优点，既降低了硬件复杂度，又保持了较高的命中率。实验需实现以下目标：

1. 地址划分与映射机制：理解主存地址如何划分为标记（Tag）、组索引（Index）和块内偏移（Offset），明确数据块在Cache中的组内灵活映射特性。
2. 并行比较与命中判断：设计2路并行的组内比较逻辑，通过同时比较组内所有行的标记位，快速判断数据是否命中。
3. 替换算法实现：在组内冲突时，应用LRU（Least Recently Used）算法选择替换目标，优化Cache利用率。

## 3.2主要任务

为实现上述目标，除了构建基础Cache槽并提供读出写入等信息接口以外，实验分还需要完成以下核心任务：

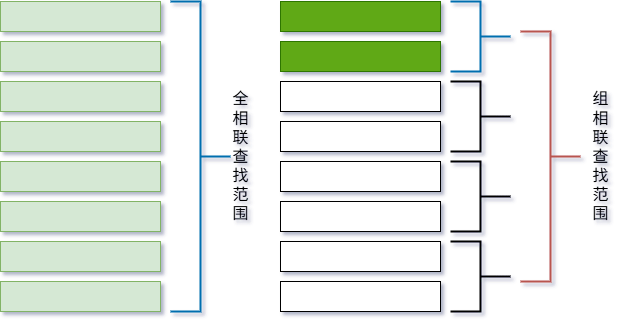
1. **地址划分设计**：将16位主存地址划分为12位标记（Tag）、2位组索引（Index）和2位块内偏移（Offset），确保组索引能定位2个候选Cache行。
2. **组内并行比较**：依照CAM设计并行比较逻辑电路，实现组内标记的并行匹配，生成命中信号（Hit）和行选择信号（Line Select）。
3. **LRU替换策略**：为每组维护一个Count计数器，记录行访问顺序，替换时选择组内最久未使用的行。
4. **数据读写整合**：在Logisim中构建由4组×2行组成的存储阵列，集成标记存储、数据块存储、比较电路及LRU更新逻辑，形成完整Cache系统。

## 3.3实验方案的总体设计

为了实现2路组相联cache高速缓冲存储器，我设计了下面的五个模块，分别对应了上述提到的具体任务：

* 构建Cache槽，实质是相连存储器。Cache负责存储数据块和保存辅助信息，如写使能信号、Valid位、命中信号Li等，并提供控制读取和辅助信息的接口。
* 实现相连存储器CAM的多路并发比较逻辑。比较地址中的Tag和每一个Cache槽的有效位信息和。
* 实现读命中逻辑。当内存地址命中Hit时，直接从Cache中读出。
* 实现读缺失逻辑。当内存地址没有命中Miss时，需要报告读缺失Miss；同时还需要实现LRU算法，找到符合条件的槽，并交给块写入逻辑实现写操作。
* 实现块写入逻辑。收到Miss信号时，要在对应的存储块中写入新的数据块。

通过这五个模块的紧密协作，4路组相联Cache能够实现高效的数据存储、查找、命中处理、缺失处理及数据更新，从而实现4路组相联Cache的主要功能。

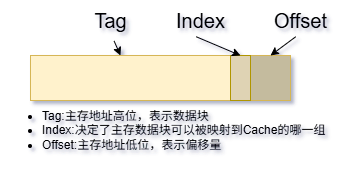


2路组相联和全相联的区别（仅考虑这里的四组2路组相连）在于，在判断读缺失逻辑时，只需要考虑某一组内的替换位置进而在块写入时也只需要考虑在某一组内的写入位置。这样的设计使得组相联相比全相联在少量比较器下仍保持较高命中率，适合现代CPU设计。

## 3.4实验的实现过程

本实验严格遵循**组相联Cache**的实际工作流程进行设计与实现，各功能模块的开发顺序与Cache处理访存请求的时序逻辑完全匹配。从**存储结构构建**开始，逐步完成**地址解析、组选择、并行比较、命中判断、缺失处理及替换策略**等核心功能，整个实现过程模拟了真实硬件的工作机制。这样的实现顺序不仅保证了模块间的逻辑一致性，也使我能够直观理解Cache从接收访存请求到完成数据访问的完整流程。每个阶段的实现均对应Cache实际工作的关键环节，确保最终系统能准确反映硬件行为。实现流程与全相联Cache基本一致，可参照1.4中的工作流程图，它为理解组相联映射原理提供了清晰的参考。

Cache的核心功能仍是**通过地址快速定位数据**。

在4路组相联设计中，每个16位地址被划分为：

**2路组相联Cache的实现步骤可以参照4路组相联的实现。**在这次课设中，全相连的实现时长大约是4路组相联的两倍，4路组相联的实现时长又大约是2路组相联的两倍。

每一步的详细实现会在`3.5 电路图及相关说明`中结合电路图说明。

## 3.5电路图及相关说明

### 2.5.1 Cache存储部分基本结构

与之前一致。

### CAM及其并发查找逻辑

该电路负责检查请求的数据块是否存在于Cache中（命中），如果存在，则检索其对应的Tag并验证其有效性。如果数据块未找到，则生成缺失（Miss）信号。

1. Tag存储与比较：

tag0到tag7信号代表8个Cache行中存储的Tag值。v0到v7信号代表每个Cache行的有效位（Valid Bit）。只有当v位被置位时，Cache行才被认为是有效的。电路中使用了两个大型的多路选择器MUX。每个MUX接收四个Tag输入（例如，顶部MUX接收tag0, tag2, tag4, tag6）。index信号用于从总共8个Cache行中选择与当前访问地址对应的Tag对（以及它们对应的有效位）。由于是4组2路的Cache结构，index会选择相关的组。MUX的输出会与传入的tag通过比较器进行比较。MUX中连接到0的输入表示在index选择到不存在的路或无效条件时，会选择一个默认值0。

1. 有效位检查：

有效位（v0到v7）也以类似Tags的方式被多路复用，确保只有有效的Cache行才参与命中判断。

1. 命中信号生成：

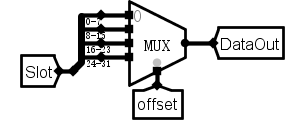
Tag比较器的输出（每个组中每个路的比较结果）与相应的有效位进行逻辑组合。在所选组内，两路的比较结果通过一个或门连接。如果所选组中的任一路的Tag匹配且有效位为真，则会生成Hit信号。Miss信号是Hit信号的非，通过反相器实现。

1. 数据输出解复用器 (DMX)：

存在两个解复用器（DMX）组件，它们用于控制所选组中两路的数据输出。index信号驱动这些DMX，将Tag比较和有效位检查的结果导向到相应的Cache数据行线（L0-L7）。这些信号（L0到L7）代表了每个Cache行的“命中”或“被选中”状态，随后将用于使能从该特定行读取数据。

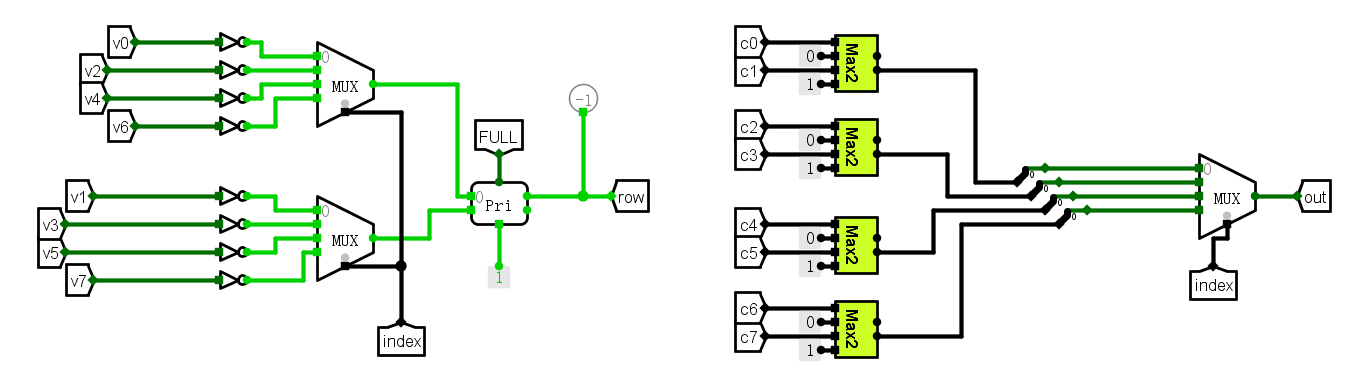
### 读命中逻辑

与先前一样，使用分线器分出数据，再根据Offset值选择，依靠多路选择器实现。如下图所示：



### 2.5.4 读缺失逻辑

下面的电路用于实现读缺失逻辑，也就是未命中时的电路逻辑，需要根据LRU算法找到最合适替换的cache槽，以信号的形式传递给块写入逻辑，再由块写入逻辑负责写入操作。

****

上图实现了Cache中用于实现LRU算法以及查找空闲槽位的核心逻辑。该电路的主要目标是，在需要进行数据写入时，确定在当前Cache组中，应该选择哪一个Cache行进行替换，或者是否存在并选择一个空闲的槽位。

电路的输入是八个有效位（v0到v7），每个v信号代表一个Cache行的有效状态。值得注意的是，这些Valid信号在进入多路选择器MUX之前都经过了反相器，这意味着电路实际上是在寻找**无效**的Cache行，也就是空闲的槽位。

接下来是两个多路选择器，它们都由index信号控制。index信号用于选择当前正在操作的特定Cache组。由于是二路组相联Cache，每个组包含两个路。这两个MUX分别负责处理对应于所选组的两路的有效位。例如，一个MUX负责偶数索引的有效位（如v0, v2, v4, v6），而另一个MUX负责奇数索引的有效位（如v1, v3, v5, v7），然后index信号从中选择出当前组对应的两个有效位状态。

这两个MUX的输出随后作为输入，送给一个优先级编码器（Pri）。优先级编码器是一种组合逻辑电路，它根据输入信号的优先级（通常是最低或最高索引的有效输入）来产生一个二进制输出。在这里，Pri编码器是在寻找第一个为“无效”（即反相后为真）的Cache行，以识别可用的空闲槽位。

优先级编码器有两个重要的输出信号：

1. FULL：这个信号指示所选的Cache组是否已经完全被有效的Cache行占据。如果Pri编码器没有找到任何无效的（即空闲的）行，那么FULL信号就会被置高，表示该组已满，此时需要执行替换策略。
2. row：这个信号是Pri编码器的主要输出，它提供了一个索引值。如果存在空闲槽位，row就指示了第一个空闲槽位的索引。如果组已满，row则指示根据替换策略（例如LRU）应该被替换的Cache行的索引。

总之，该电路通过检查Cache行的有效位，结合index信号来定位特定组，并利用优先级编码器高效地确定当前组的状态（是否已满）以及在需要写入新数据时应选择替换哪一行或使用哪个空闲槽位。

### 3.5.5块写入逻辑

这个子电路实现了Cache的写入使能逻辑，其核心功能是控制哪个具体的Cache行（从write0到write7）将被写入数据，这发生在Cache缺失并需要从主存加载新数据块时。电路的起始是一个与门，它接收两个关键输入：BlkReady和Miss。BlkReady信号表示从主存读取的数据块已经准备就绪，可以被写入Cache；而Miss信号则表明CPU请求的数据在Cache中并未找到。只有当这两个条件都满足时（即数据已就绪且发生了缺失），与门的输出才为真，从而使能后续的写入操作。

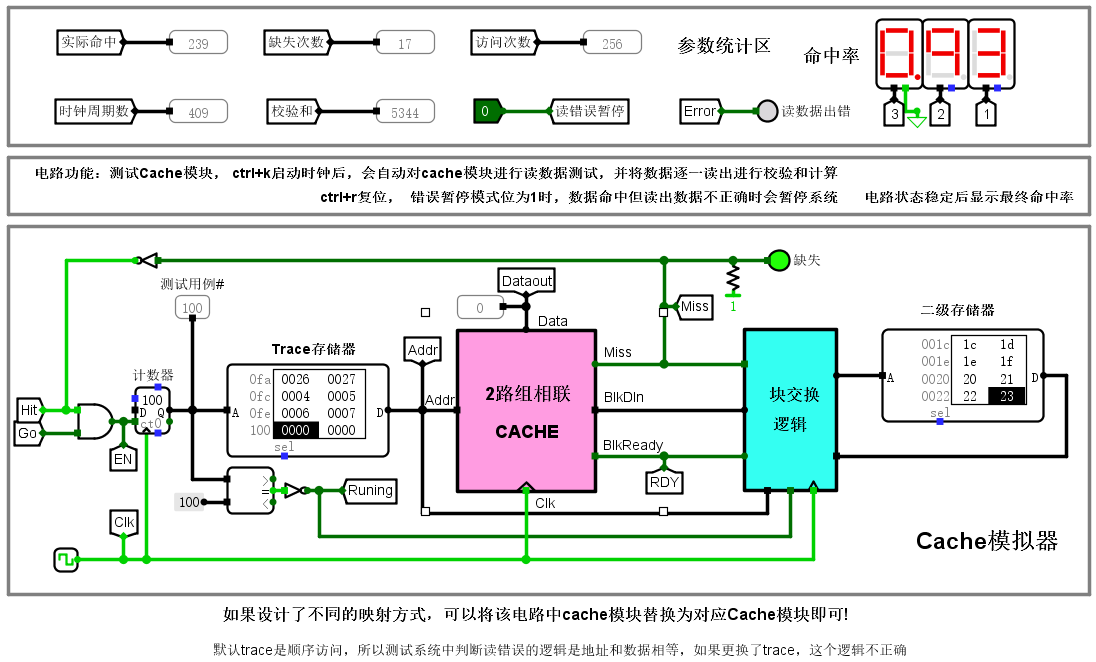
与门之后，是一个多路选择器MUX，它负责根据当前Cache组的状态选择正确的写入目标行。这个MUX有两个数据输入：row和out，并由FULL信号作为选择控制端。FULL信号指示当前选定的Cache组是否已经完全被有效数据块占据（即没有空闲的槽位）。如果FULL为高，表示组已满，此时MUX将选择row信号作为写入目标，这个row信号通常来源于LRU算法或其他替换策略的逻辑，指示应替换掉哪一个现有的Cache行。相反，如果FULL为低，表示组中还有空闲槽位，MUX将选择out信号作为写入目标，这个out信号通常指示了第一个可用的空闲槽位的索引。

多路选择器选择出的最终目标行索引（从0到7，代表总共8个Cache行中的一个）随后作为控制信号输入到一个解复用器DMX。这个DMX的输入是来自与门的使能信号。根据DMX的控制输入（即选定的目标行索引），它会激活其对应的单个输出线（write0到write7中的一条）。当这条输出线被激活时，就意味着对应的Cache行被选中，可以接收从主存加载的新数据块，从而完成写入操作。整个电路的设计确保了在Cache缺失发生时，数据能够被准确无误地写入到Cache中选定的正确位置。

## 3.6实验数据测试

EduCoder平台通过的截图



Cache自动测试电路的运行截图

# 实验四、单周期MIPS CPU设计

## 4.1实验目的

构建一个32位MIPS CPU处理器，这里使用单周期硬布线实现，该处理器应支持核心指令集中列出的所有指令。最终设计完成的CPU应能运行标准测试程序。下表列出了需要支持的汇编指令。

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **指令** | **格式** | **备注** |
| 1 | Add | add $rd, $rs, $rt | 指令功能及指令格式  参考MIPS32指令集 |
| 2 | Add Immediate | addi $rt, $rs, immediate |
| 3 | Load Word | lw $rt, offset($rs) |
| 4 | Store Word | sw $rt, offset($rs) |
| 5 | Branch on Equal | beq $rs, $rt, label |
| 6 | Branch on Not Equal | bne $rs, $rt, label |
| 7 | Set Less Than | slt $rd, $rs, $rt |
| 8 | syscall（display or exit） | syscall | 系统调用，用于停机 |

## 4.2主要任务

单周期硬布线CPU功能部件：

1. 指令计数器PC，用于存放当前指令执行位置的地址；
2. 指令存储器IM，储存CPU要执行的指令；
3. 数据存储器DM，根据设计需求，所有指令都必须在一个时钟周期之内完成，只采用一个存储器不可能在一个时钟周期内同时完成对指令和数据的操作；
4. 立即数扩展器S-EXT，用于将I型指令中的16位立即数扩展为32位；
5. 控制器，产生控制信号，控制指令执行的数据通路；
6. 寄存器堆RegiFile，提供32个MIPS通用寄存器；
7. 算术逻辑单元ALU，产生运算结果。

## 4.3 实验方案的总体设计

本节将详细阐述本次实验的总体设计，包括实验方案的选择与规划，以及系统的整体架构。我选择基于现有的MIPS单周期CPU模型进行仿真与分析。该模型清晰地展示了指令的取指、译码、执行、访存和写回五个阶段，以及各阶段所需的硬件组件（如PC、指令存储器、寄存器堆、ALU、数据存储器等）及其相互连接方式。相较于多周期或流水线CPU设计，单周期CPU在复杂性上更低，更适合作为初学者理解CPU工作原理的起点。

### 4.3.1实验规划

实验规划主要围绕以下几个方面展开：

1. 指令集选择与测试用例设计： 针对MIPS指令集中的核心指令（如add, sub, lw, sw, beq, j等）设计测试用例。这些测试用例将覆盖数据通路的所有关键部分，并验证控制逻辑的正确性。例如，对于给定的addi $s0, $s0, -1指令，我们将重点观察寄存器$s0的值变化、ALU的运算结果以及写回操作。
2. 观察与分析： 在仿真过程中，我们将重点观察以下关键信号和数据：

PC的值变化，以验证指令的顺序执行和分支/跳转逻辑。

指令存储器的输出，确保正确取指。

寄存器堆的读写操作，验证数据读出和写回的正确性。

ALU的输入、操作码和输出，验证算术逻辑单元的功能。

数据存储器的读写操作，验证访存指令的正确性。

控制单元的输出信号，确保其在不同指令下能生成正确的控制信号。

1. 结果记录与报告撰写： 详细记录仿真结果，并基于观察结果撰写实验报告，分析CPU的工作原理和可能存在的问题。

### 4.3.2总体设计

本实验所使用的MIPS单周期CPU的总体设计遵循经典的冯·诺依曼架构，其核心组成部分包括：

1. 程序计数器 (PC): 负责存储下一条待取指令的地址。在每个时钟周期，PC的值会更新为当前指令地址加4（对于顺序执行），或者根据分支/跳转指令的结果更新为新的目标地址。
2. 指令存储器 (Instruction Memory): 根据PC提供的地址，取出对应的机器指令。
3. 寄存器堆 (RegFile): 包含32个通用寄存器，用于存储数据和中间计算结果。它支持两个独立的读端口（R1#，R2#）和一个写端口（W#），以满足指令对多个寄存器的同时访问需求。
4. 算术逻辑单元 (ALU): 执行指令中的算术和逻辑运算（如加、减、与、或等）。其操作类型由控制单元的ALUOp信号决定。
5. 数据存储器 (Data Memory): 用于存储和读取数据。访存指令（lw, sw）通过它来与外部存储器交互。
6. 控制单元 (Control Unit): 是CPU的“大脑”，根据当前指令的OpCode和Func字段，生成控制整个CPU各个组件（如ALUOp、MemToReg、MemWrite、RegWrite、Branch、Jump等）的信号。
7. 多路选择器 (MUX): 用于根据控制信号选择不同数据源，例如选择ALU的第二个输入、写回寄存器的数据来源、以及PC的下一个值。
8. 符号扩展单元 (Sign Extend): 将16位立即数扩展为32位，用于addi、lw、sw等指令。
9. 左移2位单元 (Shift Left 2): 用于PC相对寻址，将分支偏移量左移两位，使其成为字地址偏移。

这些组件通过数据通路和控制信号线相互连接，形成一个完整的指令执行流程：指令从指令存储器取出，经过译码后，相关数据从寄存器堆读出，送入ALU进行运算，结果可能写入数据存储器或写回到寄存器堆。PC的更新确保了指令的连续执行或流程控制的跳转。整个设计确保了每条指令在一个时钟周期内完成所有操作。

## 4.4 实验的实现过程（结合电路图说明）

### 4.4.1指令解析

将32位输入操作码用分线器接出，最高6位为操作码op，21-25位为rs寄存器编号，16-20位为rt寄存器编号、11-15位为rd寄存器编号， 0-5位为功能码func；取低16位，作为I型指令中的立即数操作数。单周期和多周期指令译码方式相同，如下图所示：

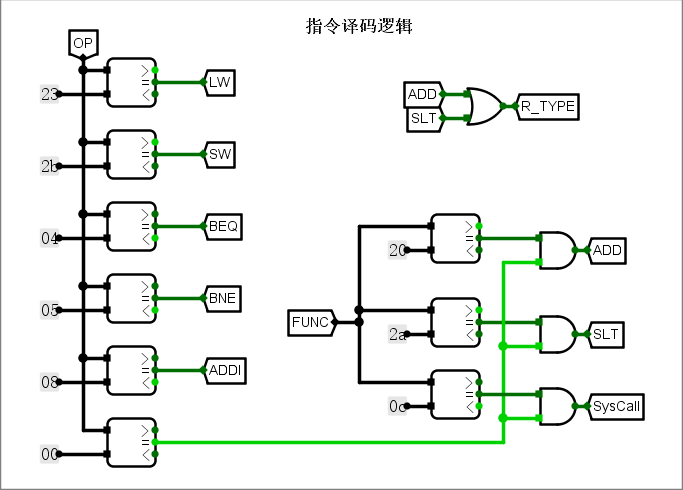


图 4.4.1.1 指令译码

### 4.4.2实现地址转移逻辑

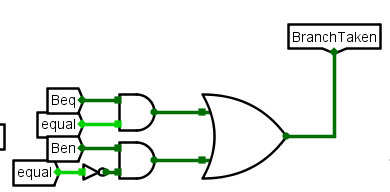
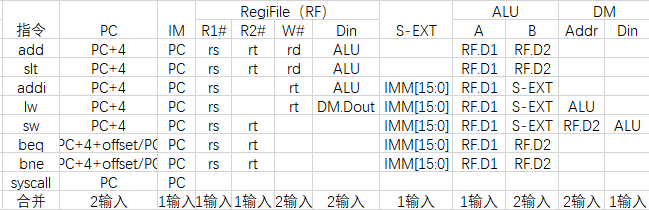
PC+4接到MUX的0输入端，16位立即数扩展结果左移2位，加上PC+4，接到MUX的1输入端，选择信号为branch。实验要求的的分支转移指令为beq和bne，两条指令均为I型指令，由于两条指令的特殊性，可利用ALU的equal信号，最终形成分支信号

图 4.4.1.2 地址转移逻辑

### 4.4.3单周期硬布线控制器

1. 数据通路综合

构建数据通路。绘制主要功能部件输入来源表，该表主要用于描述控制类信号，仅保留数据类信号，具体如表4.4.3.1，最左侧为指令助记符，第一行为数据项。输入源合并。

表 4.4.3.1 单周期功能部件输入来源表

1. 列出所有功能部件、多路选择器控制信号、运算操作选择的产生条件，如表1.3所示，横坐标给出的是不同指令的译码信号，表中有1的位置表示当前指令会产生对应的信号，利用译码电路生成各指令译码信号，然后以行为单位将各个产生信号的条件相加（逻辑或）即可得到控制信号的逻辑表达式。

* RegDst为1表示RegiFile写回地址由R型指令rd字段给出，否则由I型指令rt字段给出；
* RegWrite为1打开RegiFile写使能，表示数据写回RegiFile；
* AluSrcB为1表示ALU的第二个操作数将由立即数扩展器S-EXT给出，否则由RegiFile的第二个输出给出；
* bne为1表示指令为bne；
* beq为1表示指令为beq，beq与bne信号与ALU equal信号结合可控制PC数据来源；
* MemWrite为1打开数据存储器DM写使能，将ALU运算结果写入数据存储器DM；
* MemToReg为1表示从数据存储器DM中选数据送入RegiFile；
* Halt为停机信号，为1时系统停机；
* 加法信号，表示ALU应对两个操作数执行加法操作；
* 比较信号，表示ALU应对两个操作数执行比较操作。

1. 根据以上分析，可最终形成如下表所示指令与信号对应关系表，进而绘制出单周期硬布线控制器电路如下：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | RegDst | RegWrite | MemToReg | MenWrite | AluSrc | Beq | Bne | Halt |
| R\_TYPE | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| ADDI | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| LW | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| SW | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| BEQ | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| BNE | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| SysCall | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

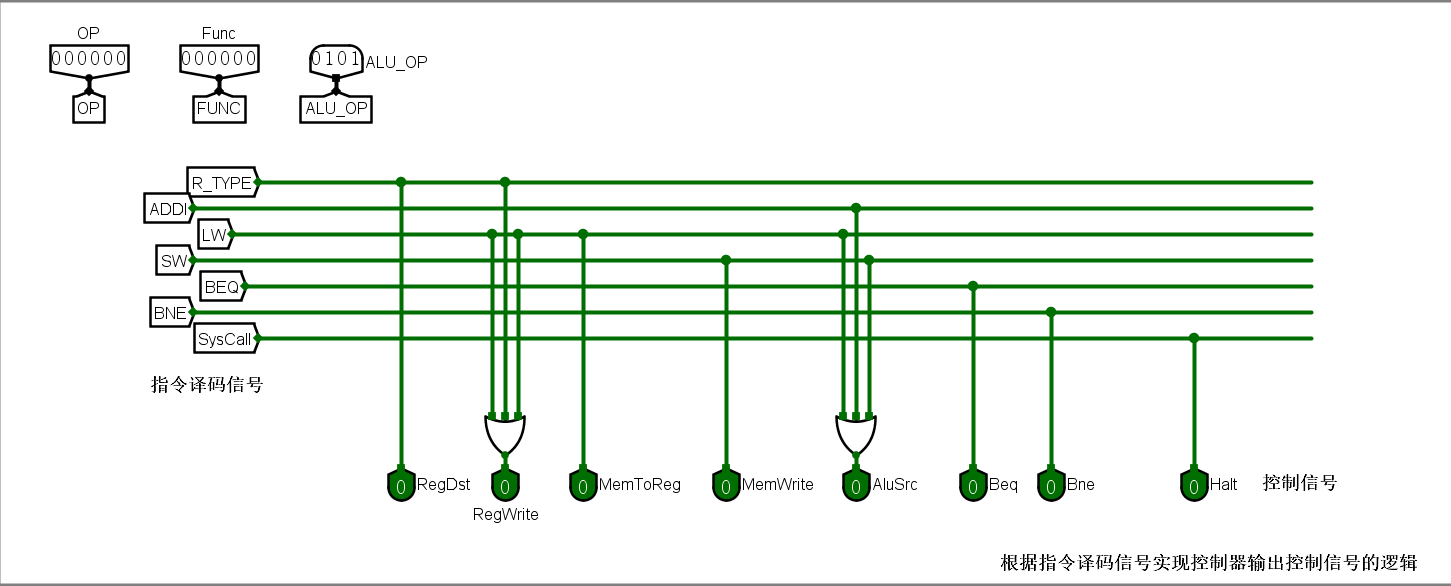




图 4.4.3.1 MIPS单周期CPU控制器电路

### 4.4.4单周期CPU总体结构图

通过上面的分析，我们已经得到单周期CPU的所有功能部件，我们也知道了所有指令的数据通路，以及如何产生控制信号控制指令数据的传递，因此把这些部件和控制信号连接起来就得到了单周期CPU总体结构图。

【主要参考图4.4.4.2】

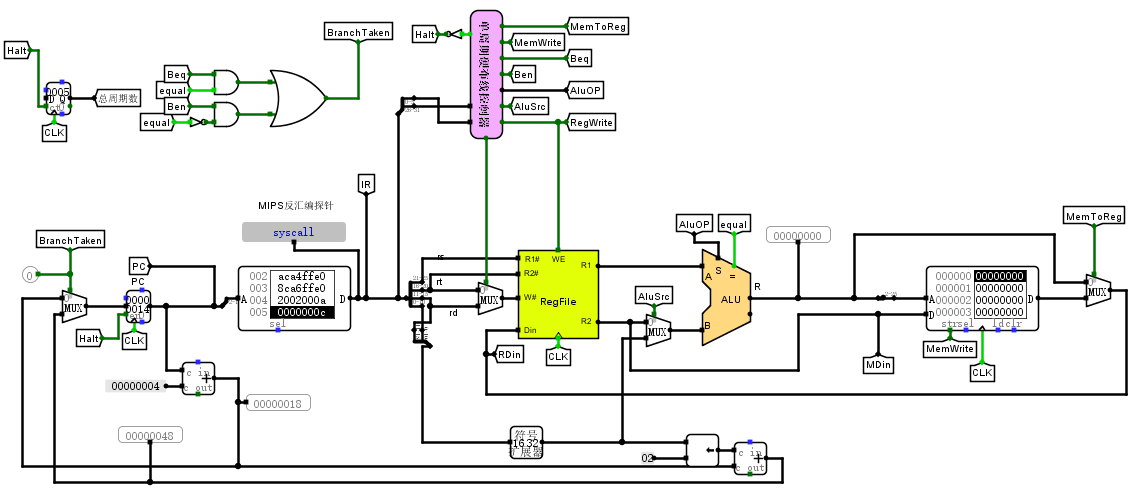


图 4.4.4.1

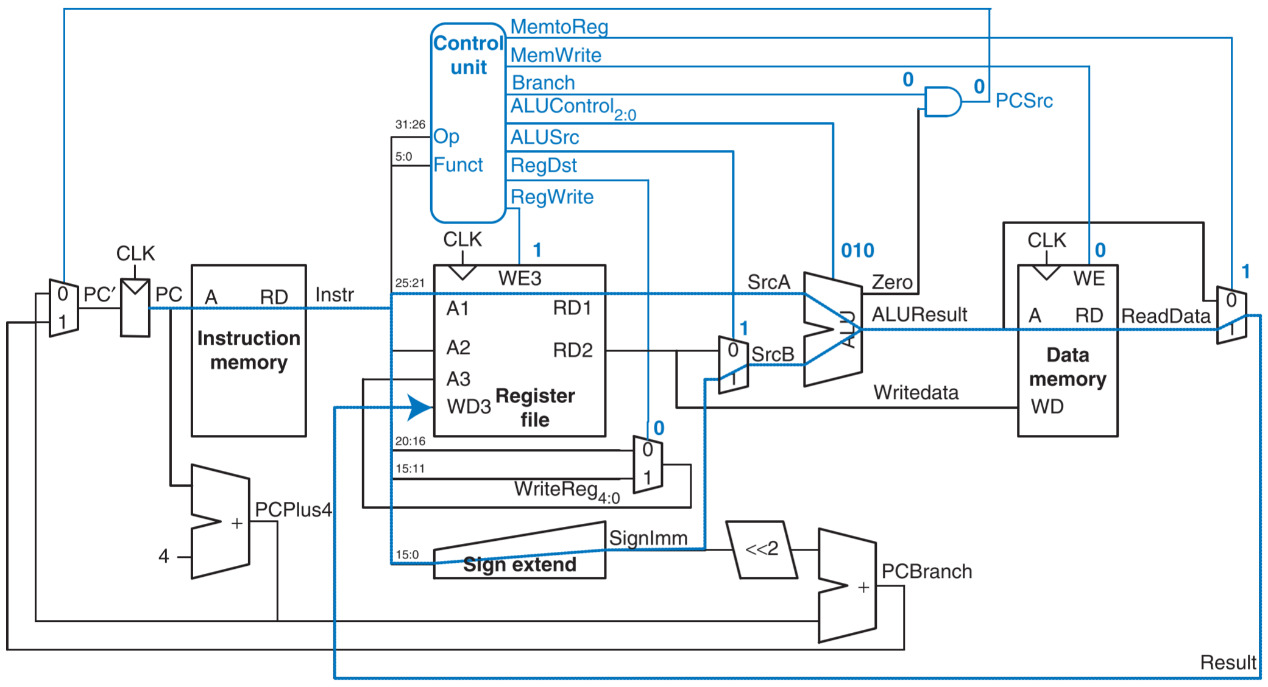


图 4.4.4.2

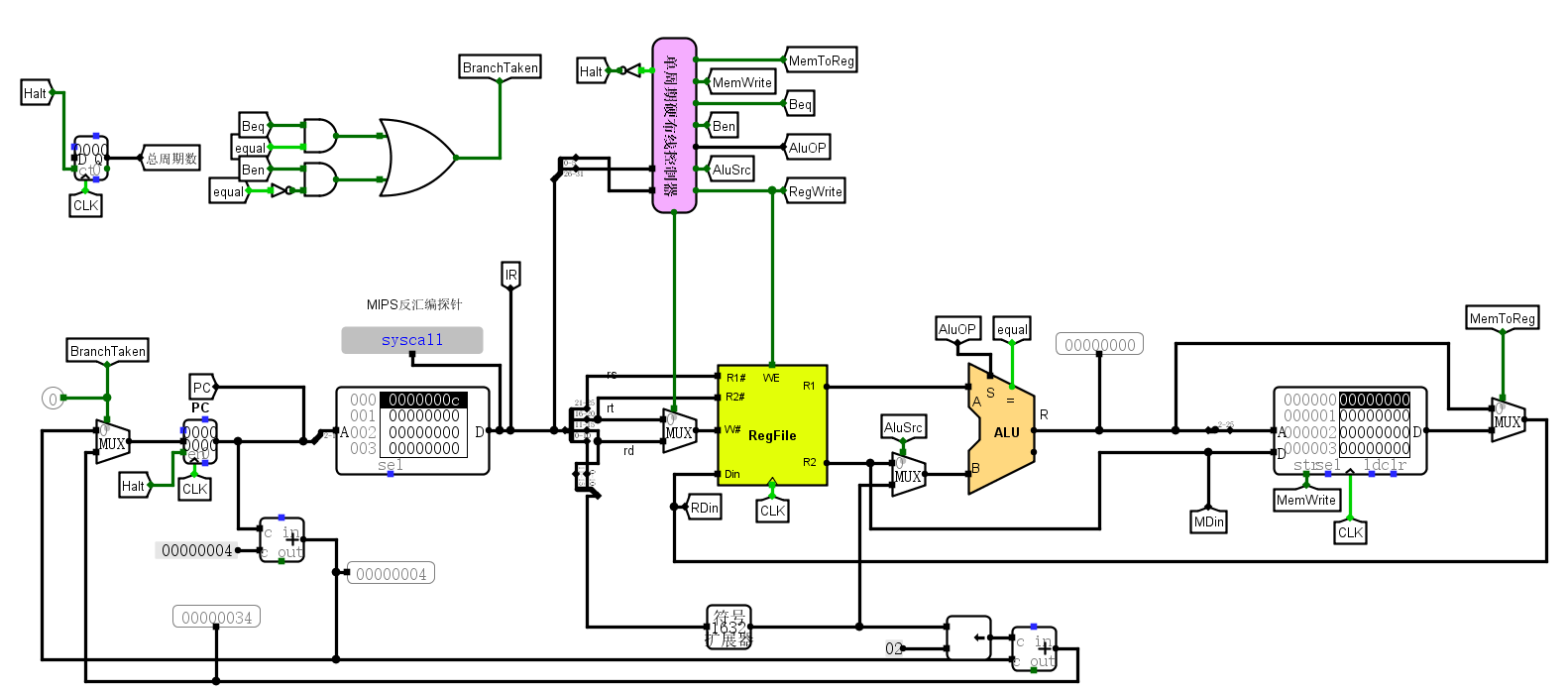
## 4.5实验数据测试

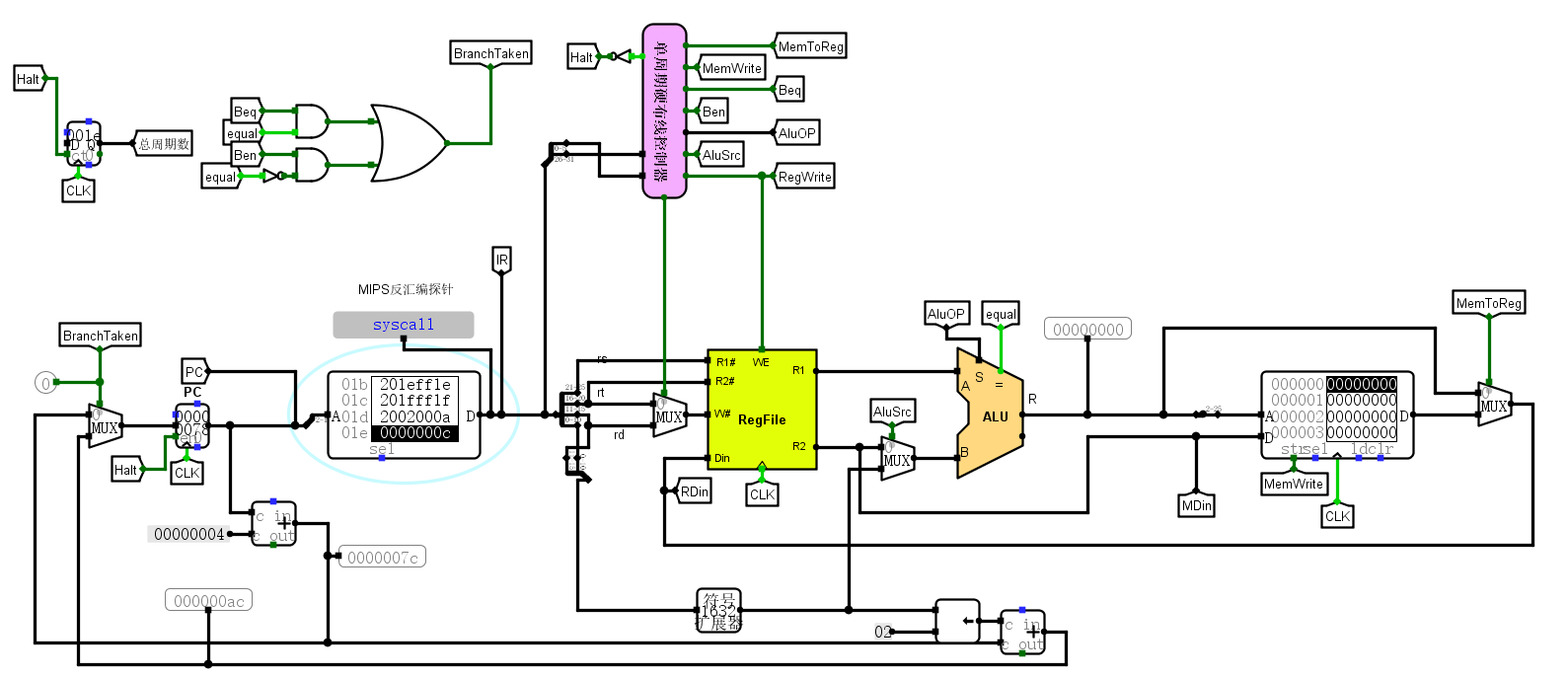
### 4.5.1头歌平台测试结果：

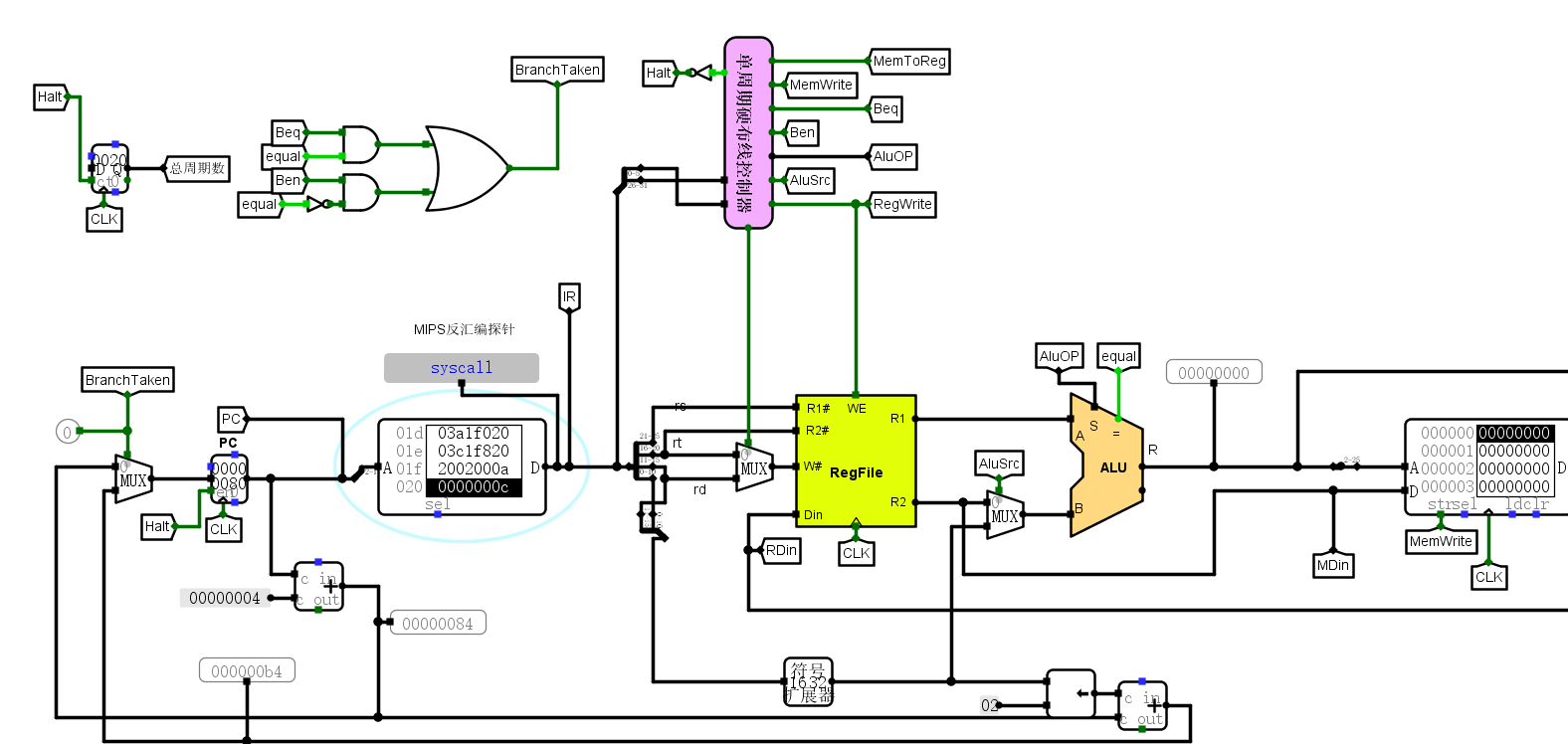
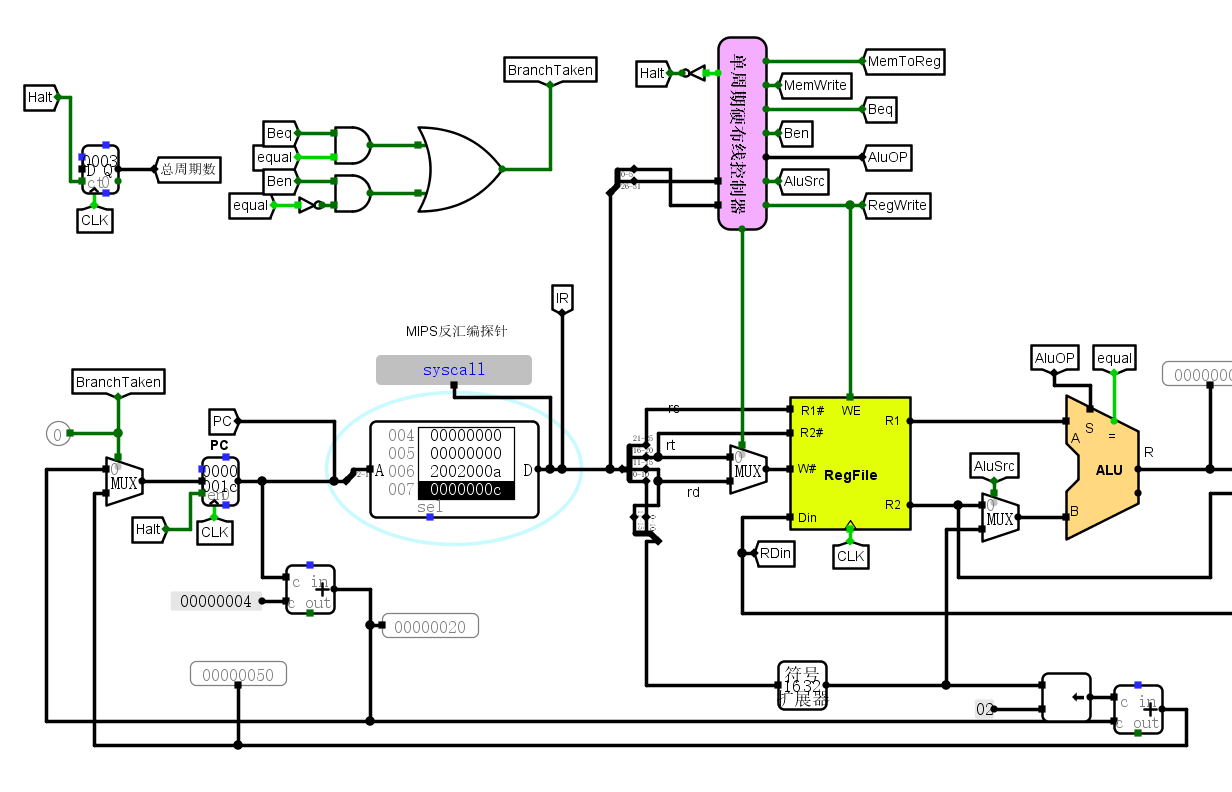
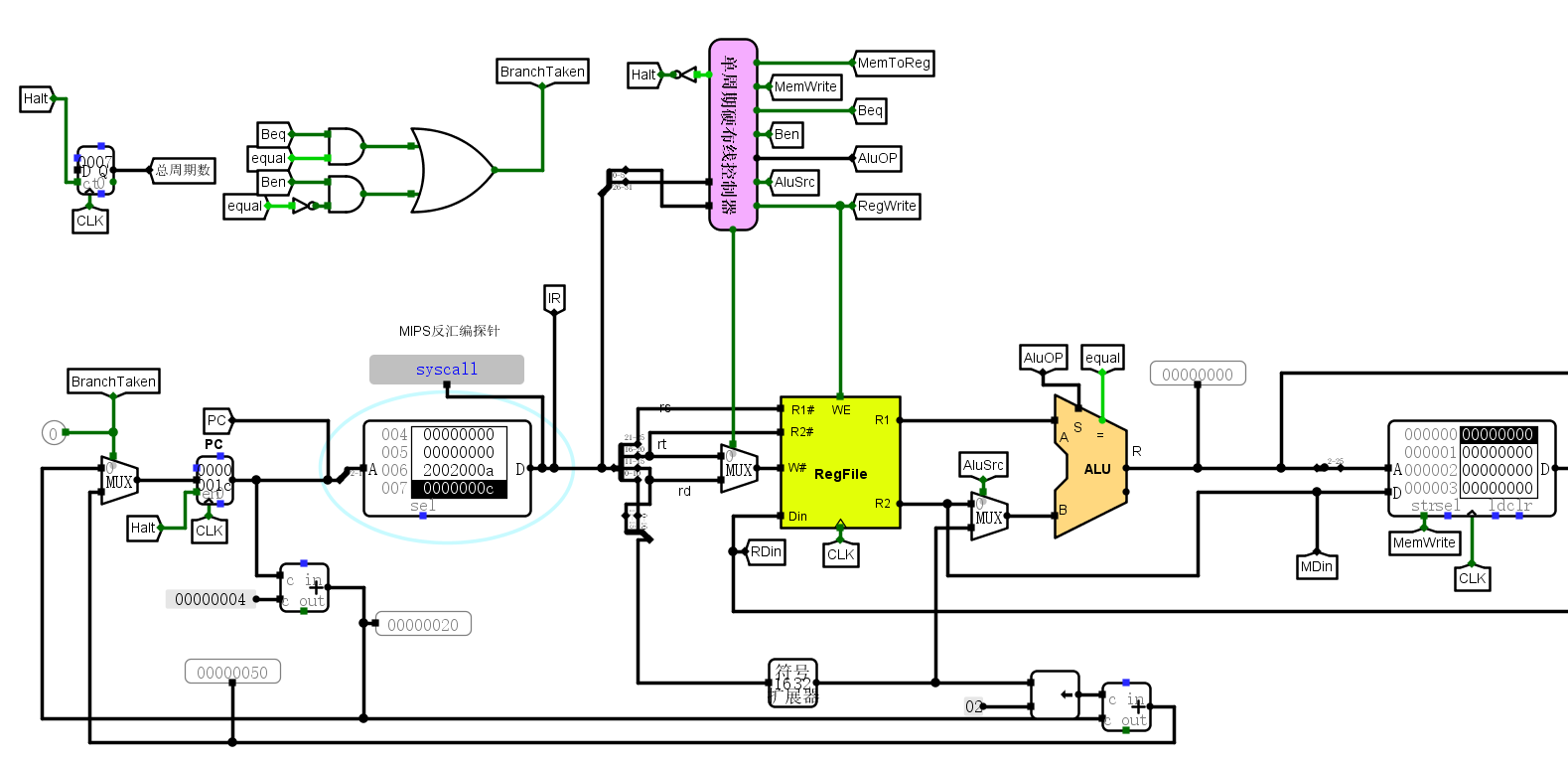
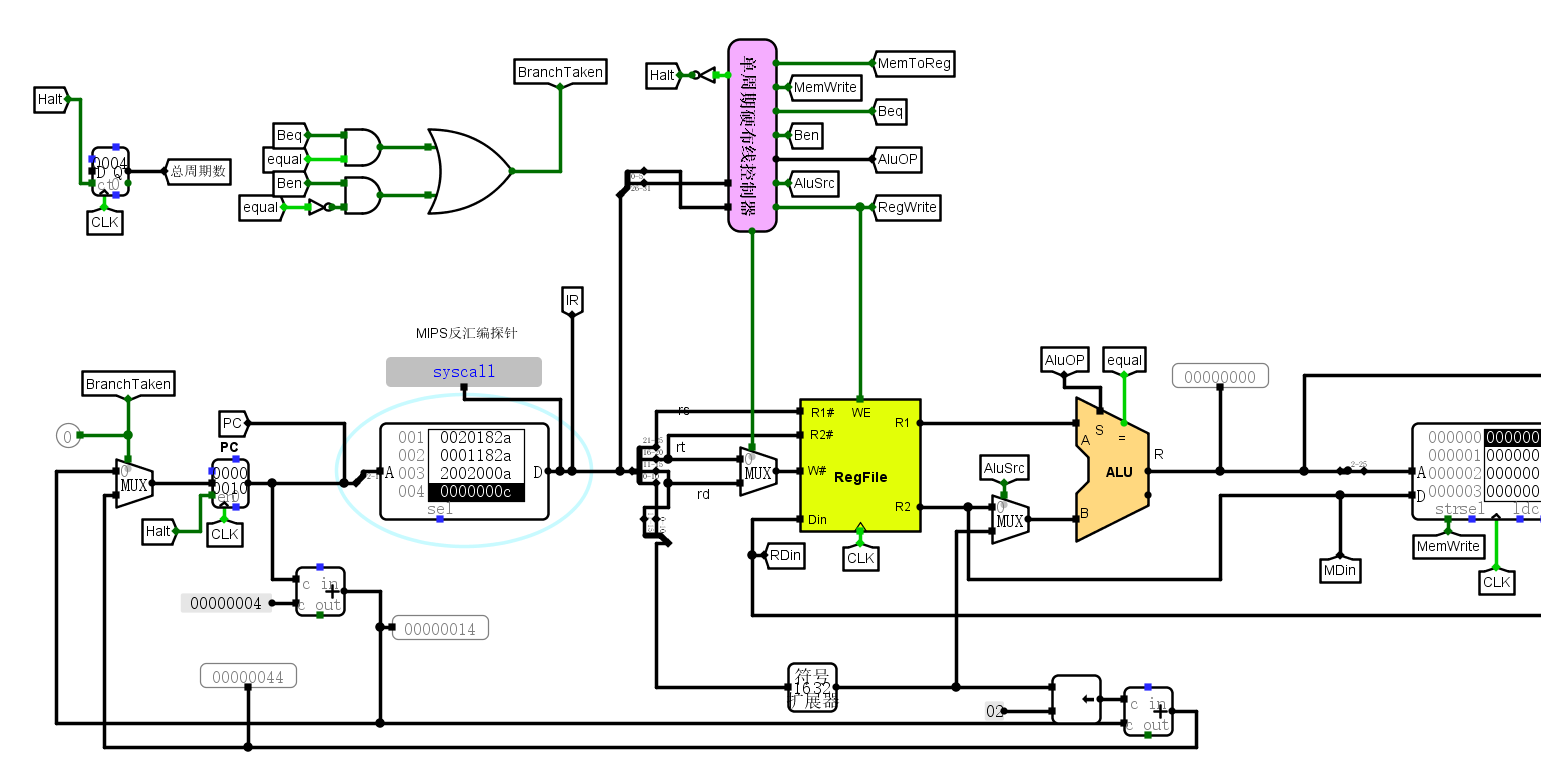
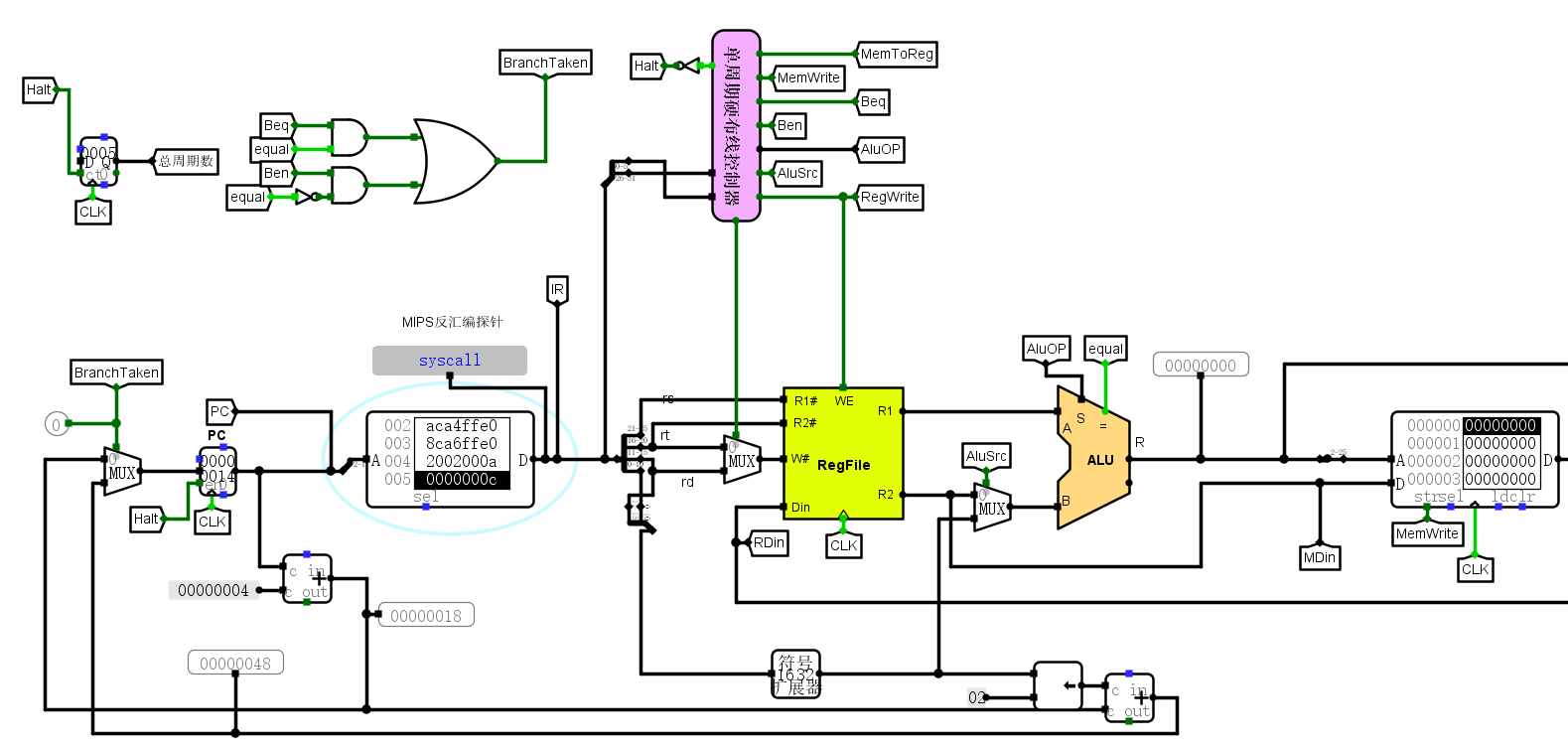
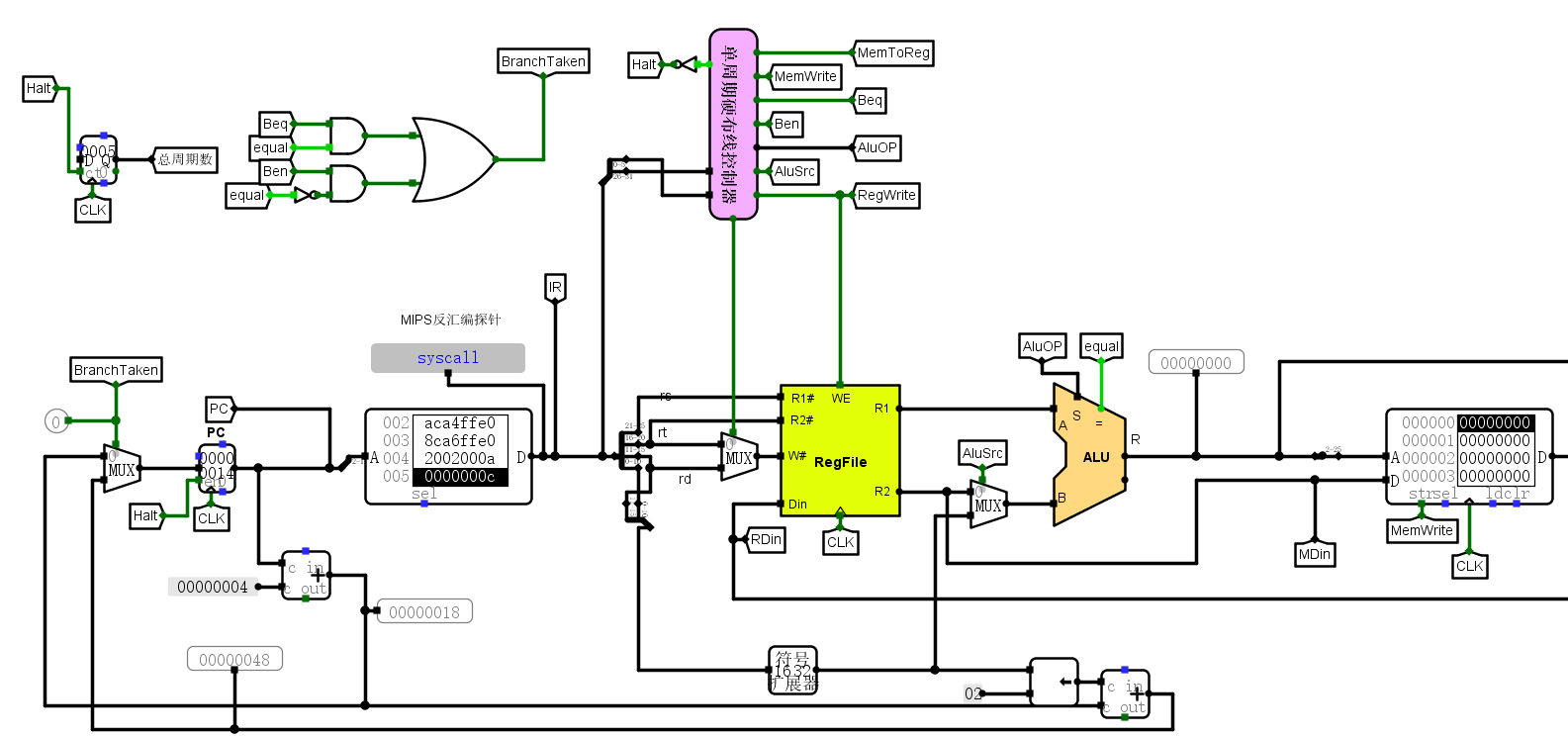


### 4.5.2八条指令测试结果：

每条指令都是从标准Sort.hex程序中选取的相关指令。优先测试Syscall停机指令，并在其他所有测试程序后都添加Syscall指令，以保证测试后停机以供检查。

1. SysCall：
2. Addi

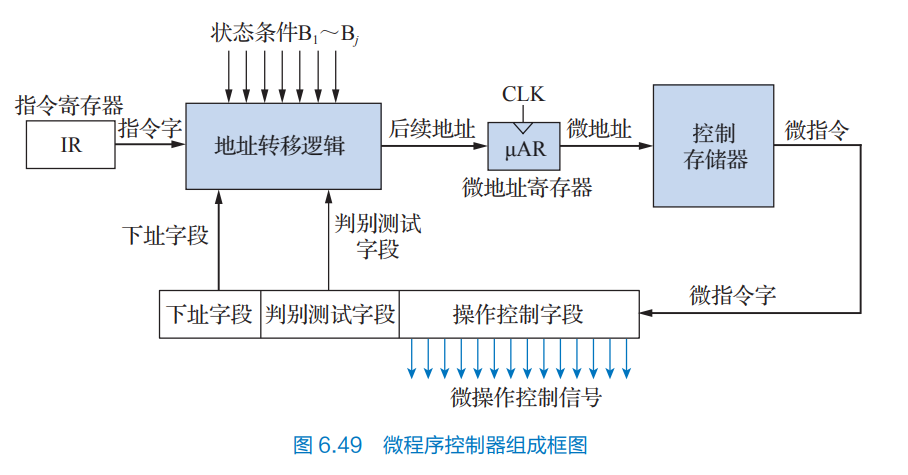


1. Add
2. Beq
3. Bne
4. Slt
5. Sw
6. Lw

# 实验五、微程序地址转移逻辑设计

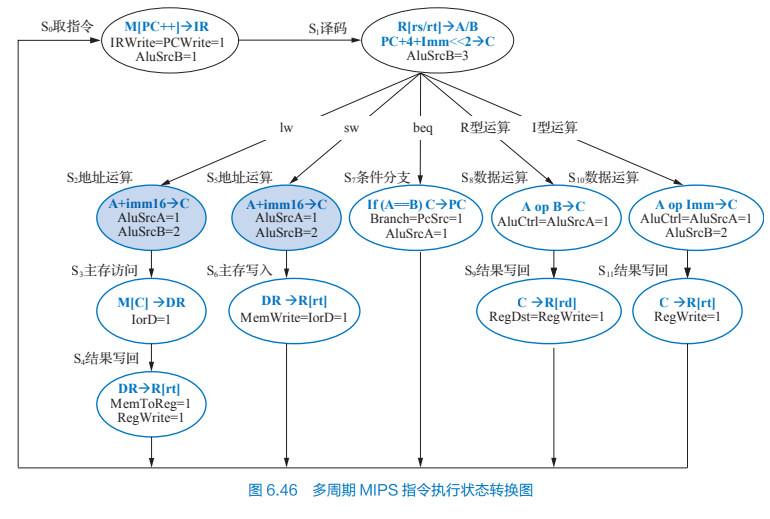
## 5.1实验目的

本实验是多周期MIPS CPU（微程序）的准备部分。多周期的MIPS CPU（微程序）的运行过程中，一条指令会对应执行多条微指令，最后再通过程序计数器PC跳转到下一条指令。对应【图6.49】中红色矩形框标出的部分。



## 5.2主要任务

【部分见`5.1 实验目的`】

1. 明确需要建议的双射关系两端
2. 在下表中找到指令与微指令的转换关系
3. 导入Excel表格中，生成相关函数表达式和对应机器码
4. 将相关函数表达式和对应机器码保存到电路存储控制器中。

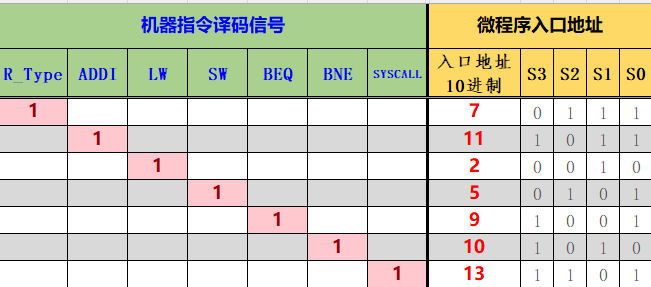
## 5.3实验方案的总体设计

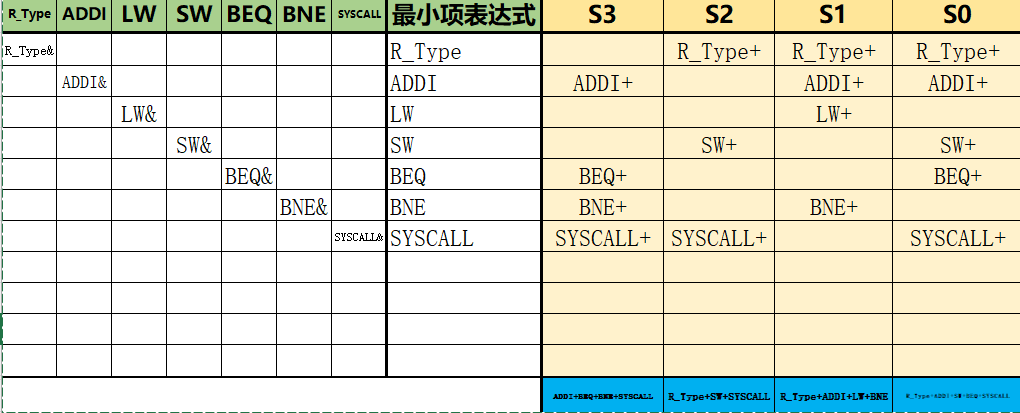
逐步实现`5.2主要任务`中的主要任务。

## 5.4实验的实现过程（结合电路图说明）

### 5.4.1微程序地址转移逻辑自动生成

在【图 6.46】中找到下表左侧机器指令译码信号对应的10进制入口地址，Excel会自动在【工作表2】中为每一条指令生成对应逻辑表达式【见下方蓝色区域】。

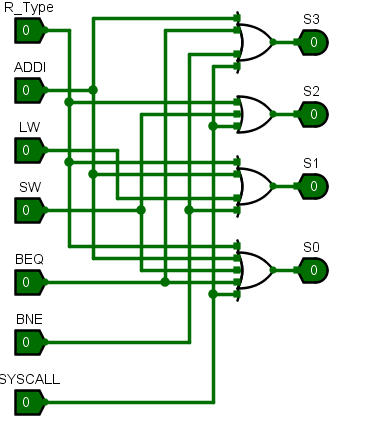
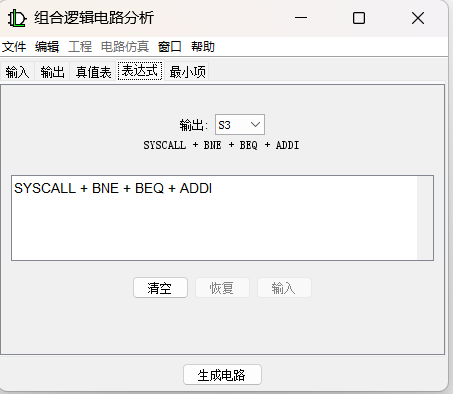




### 5.4.2使用组合电路分析功能连线

在logisim中使用组合电路分析功能，自动在提供的接口上实现。

在左图输入`5.4.1 微程序地址转移逻辑自动生成`中生成的蓝色表达式即可得到右图。



## 5.5实验数据测试

EduCoder平台通过的截图：



# 实验六、MIPS微程序CPU设计

## 6.1实验目的

1. 掌握多周期MIPS CPU设计原理

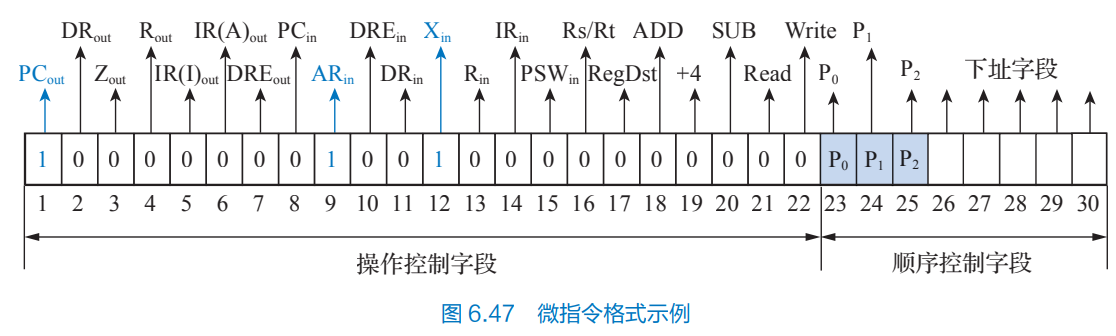
2. 掌握微程序控制器设计的基本原理

3. 利用微程序控制器的设计实现多周期MIPS处理器

## 6.2主要任务

### 6.2.1完成指令集设计

在实现多周期MIPS CPU（微程序控制）时，需要得到每一条指令的机器码和其微指令的对应关系，从而执行对应的微程序。



如图6.47所示，需要建立操作控制字段和微指令的双射关系。本实验需要利用提供的Excel表格（内嵌双射建立逻辑），找到该双射关系，并导入电路中。

### 6.2.2完成电路逻辑

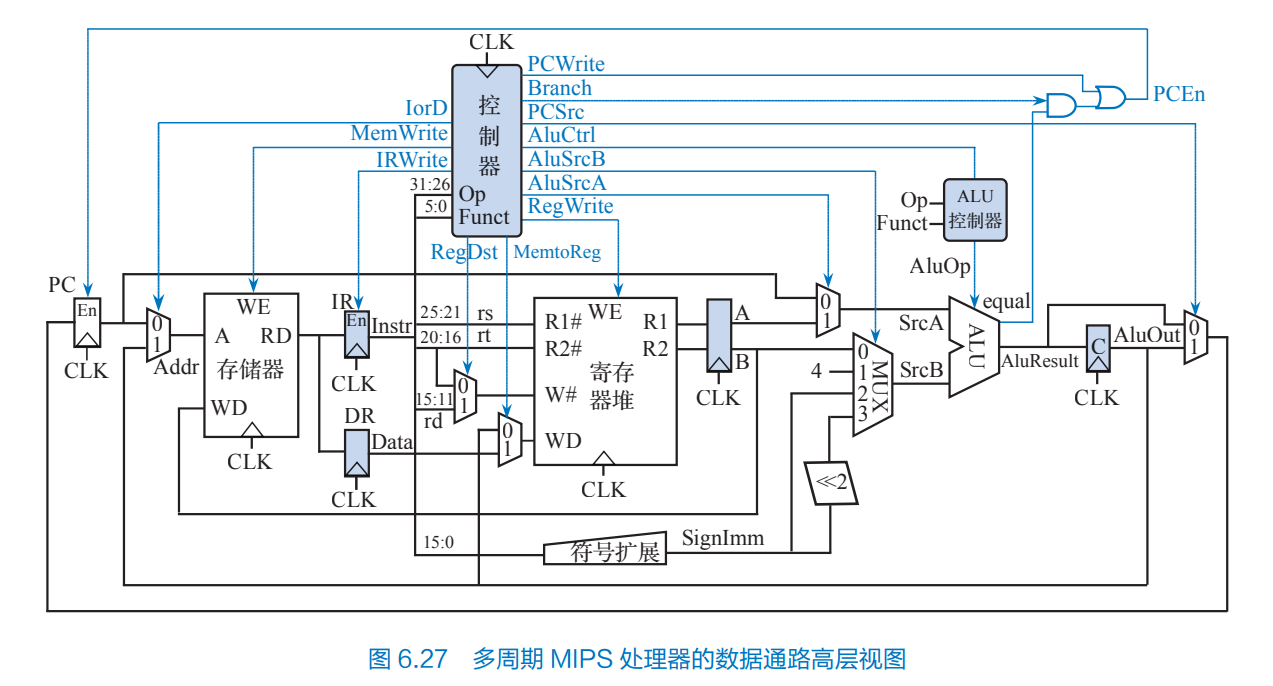
上述所有工作的最终目的是：明确每条指令所需执行的微指令及其执行顺序。基于此，接下来需要设计并实现能够按正确顺序执行这些微指令的逻辑电路。对应电路是：1. 多周期MIPS微程序；2. 多周期微程序控制器。

最后还需要测试联调。

## 6.3实验方案的总体设计

步骤1：设计微程序控制器，包括主要任务6.2.1，见【6.2.1完成指令集设计】和指令译码逻辑等。

步骤2：构建MIPS主机通路，即在电路【多周期MIPS（微程序）】中完成下图所示逻辑通路的连线。



## 6.4实验的实现过程（结合电路图说明）

### 6.4.1从单周期到多周期

多周期处理器中，指令的执行需要占用多个时钟周期，不同的指令所需的时钟周期数不一定相同，相同指令在不同时钟节拍下产生的控制信号也不同。多周期CPU的整体架构也与单周期CPU有些许不同，如下：

1. 不再区分指令存储器与数据存储器，指令和数据保存在同一个存储器中；
2. 部分功能单元，如ALU、RegiFile可以在一条指令执行过程的不同周期中多次使用；
3. 主要功能单元输出端都增加了寄存器，在后续时钟周期中要用到的所有数据必须存储在相应的寄存器中，增加数据寄存器DR，用于存放从存储器读取的数据，增加指令存储器IR，用于存放从存储器读出的指令，增加三个存储器A、B、C保存RegiFile和ALU的输出；
4. 更改后ALU输出结果有三种情况：
   1. 分支目标地址，由beq、bne指令给出，该地址将被写到PC中；
   2. 指令的运算结果，将被写入到寄存器文件中；
   3. 为存储器访问指令lw和sw提供存储器地址；
5. PC作为指令计数器，由于不同指令时钟周期数不同，因此PC不再仅由时钟周期控制，而是增加了专门的写操作控制信号。

### 6.4.2指令集的生成

本节对应【6.2.1 完成指令集设计】。

首先查书可知，多周期MIPS 处理器的控制信号解释如下。

从课本P211开始介绍了如下内容：

1. 取指令阶段数据通路

2. 译码 / 取数阶段数据通路

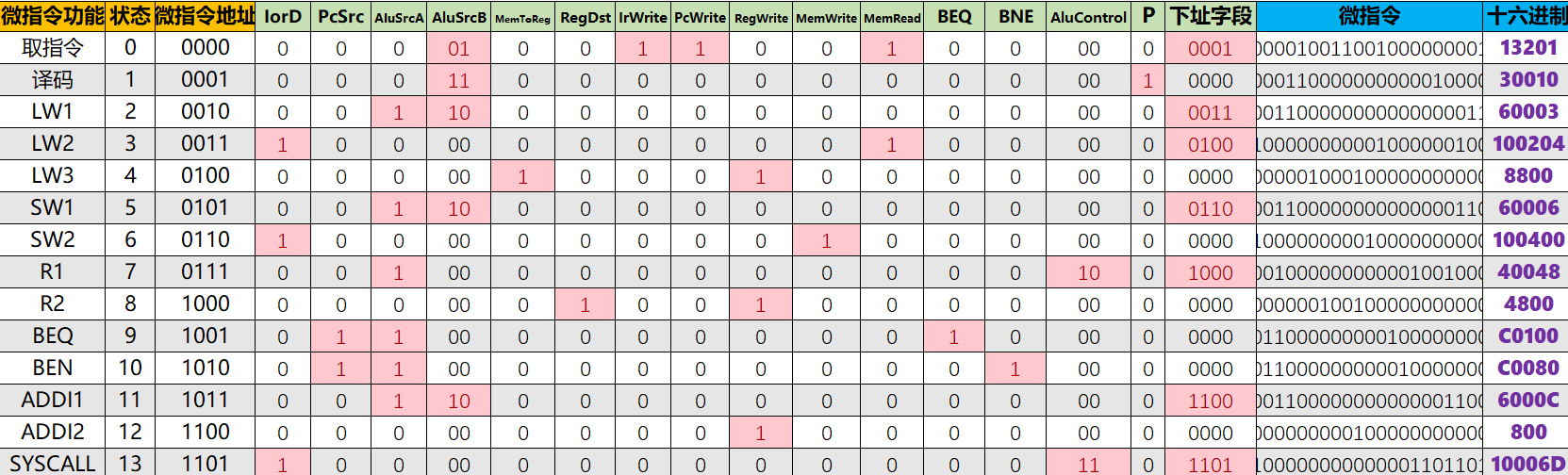
3. R型算术逻辑运算指令执行阶段的数据通路

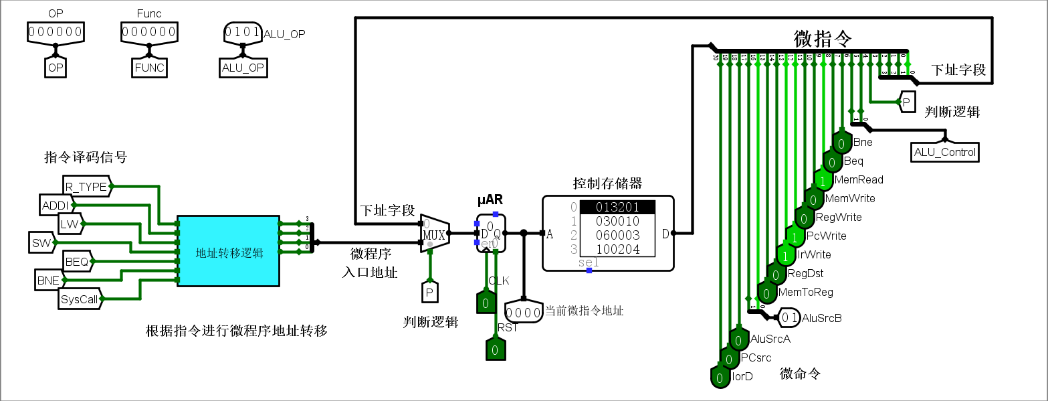
4. I 型算术逻辑运算指令执行阶段的数据通路

5. lw 指令执行阶段的数据通路

6. sw 指令执行阶段的数据通路

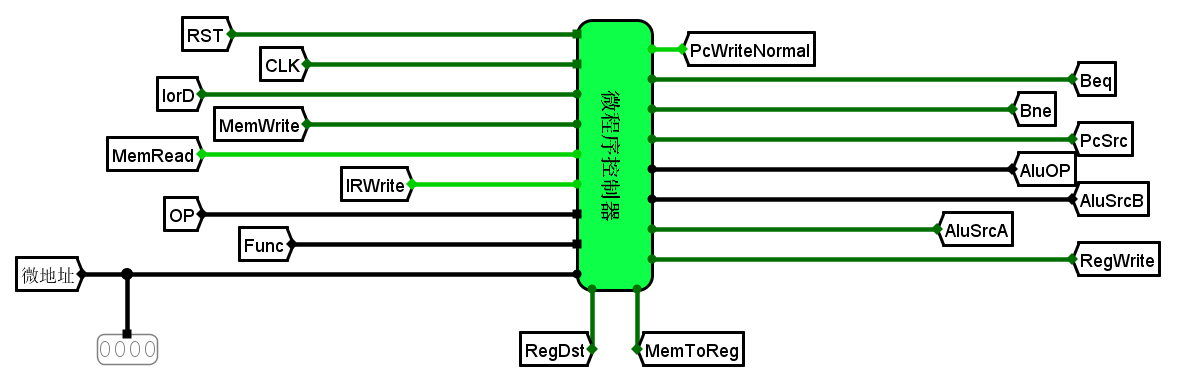
此处尚未开始连线，只需找出每一条指令需要传递的信号就可以填充完整Excel表格【微指令自动生成(2019-4-22).xlsx】，便可以得到如下结果：



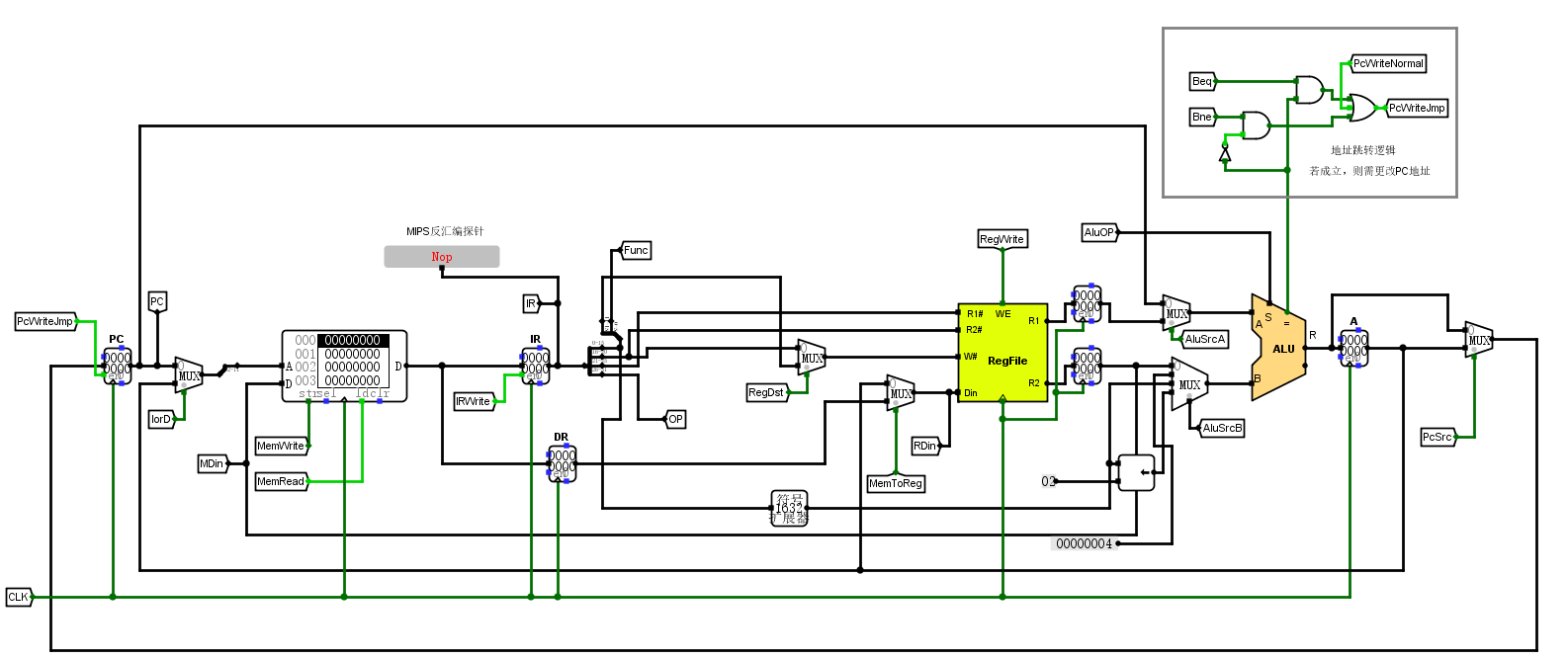
取紫色16进制值保存到电路【多周期微程序控制器】中的控制存储器，该CPU便具有了编译功能，即可以识别2进制机器码（或16进制）从而确定将要执行的指令，电路如下图中红色方框所示。

接下来，我深入理解各个指令的含义和要执行的微指令，构建数据通路。

### 6.4.3多周期CPU总体结构图

经过以上步骤，我可以构建如下微程序控制器接口：

它相当于一个编译器，输入一条指令，经过分线操作，它可以产生需要执行的微指令的信号。接下来，只需要根据【图 6.27多周期 MIPS 处理器的数据通路高层视图，见`6.3实验方案的总体设计`】搭建多周期MIPS处理器即可。使用了上述接口的设计，可以实现更简洁的连线如下：

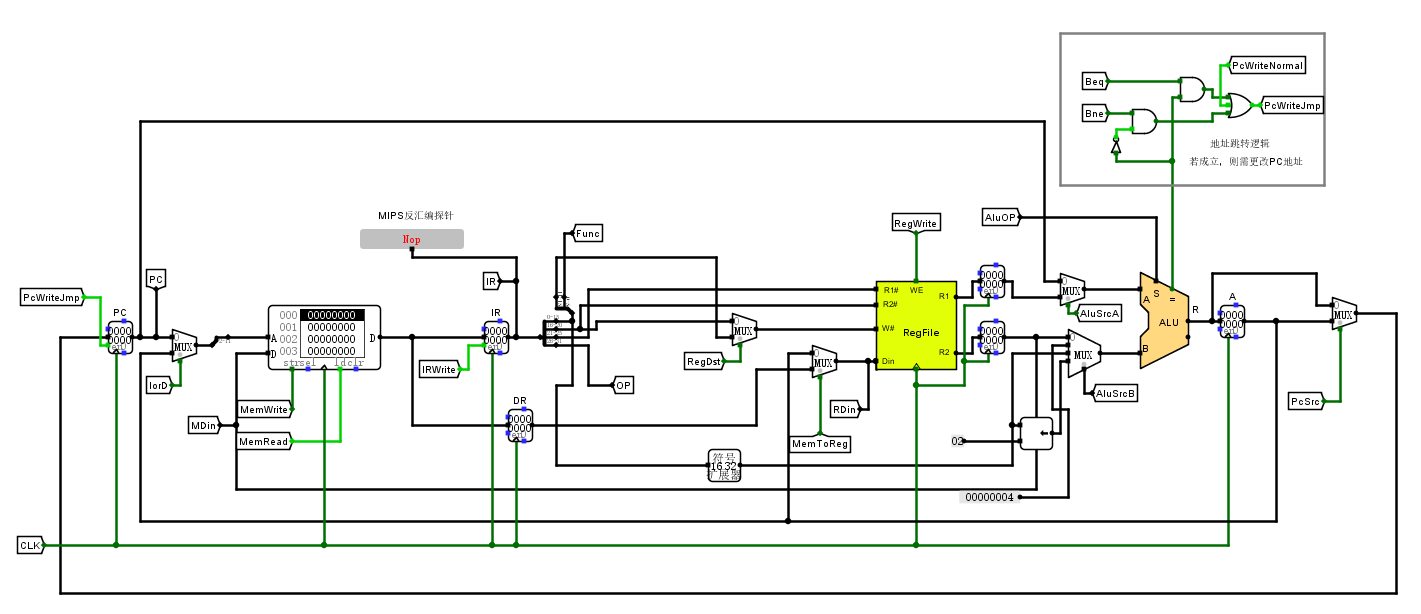


该电路图的具体讲解见`6.5电路图及其相关说明`

## 6.5电路图及其相关说明

### 6.5.1 取指令阶段数据通路

多周期 MIPS 指令周期的第一个时钟周期 T1 是取指令阶段（Instruction Fetch, IF），该阶段包含两条并行工作的数据通路（如图中红色路径所示）。

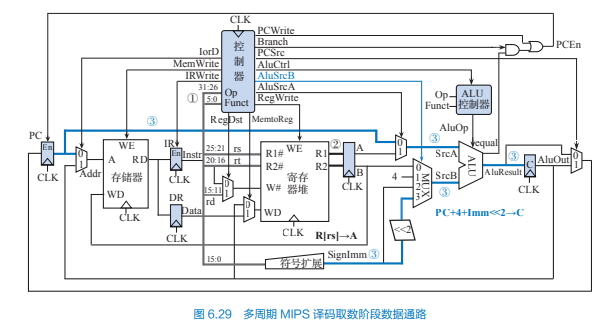


在传统的单周期 MIPS CPU 设计中，程序计数器（PC）的自增通常需要一个专用的加法器来完成，以确保在每个时钟周期结束时 PC 能正确指向下一条指令的地址。然而，在多周期 MIPS 的实现中，为了优化硬件资源的使用，PC 的自增功能并未采用独立的加法器，而是复用 ALU（算术逻辑单元）来完成这一计算。这种设计选择虽然增加了控制逻辑的复杂度，但有效减少了硬件开销，使得整体结构更加紧凑。

这一设计与单周期 MIPS CPU 的实现方式有明显区别。在单周期架构中，由于每条指令的执行仅占用一个时钟周期，硬件资源通常较为独立，以确保各功能单元能并行工作。而在多周期架构中，由于指令的执行被拆分为多个阶段，硬件资源（如 ALU）可以在不同阶段被重复利用，从而提高资源利用率。

如果仔细观察最终的电路实现图，可能会对这一设计产生疑问，因为 PC 的自增并未像传统设计那样直接通过加法器完成，而是依赖于 ALU 的运算结果。这种复用策略虽然在逻辑上是可行的，但在电路布局上可能会显得不够直观，尤其是对于习惯了单周期 MIPS 设计的人来说。因此，理解多周期 MIPS 的数据通路和控制逻辑需要更细致的分析。

### 译码/取数阶段数据通路

第二个时钟周期T2是译码/取数阶段，其数据通路如图 6.29 中粗线所示。

在译码和取数阶段（T2时钟周期），数据通路的主要操作包括指令译码和取操作数。首先，操作控制器根据指令寄存器（IR）中的指令字进行译码，生成相应的译码信号。接着，从寄存器堆中读取操作数，具体过程是将IR中的rs和rt字段分别连接到寄存器堆的R1#和R2#端口，从而选择对应的寄存器值输出到R1和R2端口。这些值随后被送入A和B寄存器中暂存，为下一阶段的指令执行提供操作数。

此外，该阶段还会提前计算分支目标地址。通过符号扩展将立即数扩展为32位后，与程序计数器（PC）的值相加，结果暂存在C寄存器中。如果当前指令是条件分支指令，下一时钟周期可以直接从C寄存器获取分支目标地址，避免了等待时间；如果是其他指令，这个地址则不会被使用。

寄存器堆的读操作是通过组合逻辑实现的，不需要额外的控制信号。A和B寄存器的写入操作仅受时钟控制，因此该通路也不需要设置控制信号。在计算分支目标地址时，数据通路复用了ALU部件，通过设置AluSrcA=0、AluSrcB=3和AluCtrl=0来完成加法操作，从而节省了一个加法器的硬件开销。

完成取指令和译码/取数阶段后，指令字被锁存在IR寄存器中，rs和rt寄存器的值分别锁存在A和B寄存器中，分支目标地址则暂存在C寄存器中。根据指令译码信号，后续的执行阶段会对这些暂存的数据进行进一步处理，以实现不同类型的指令功能，例如R型算术逻辑运算、立即数运算、访存指令和条件分支指令等。

### 6.5.3其他指令及其指令执行

其余重要电路逻辑还有如下四点：

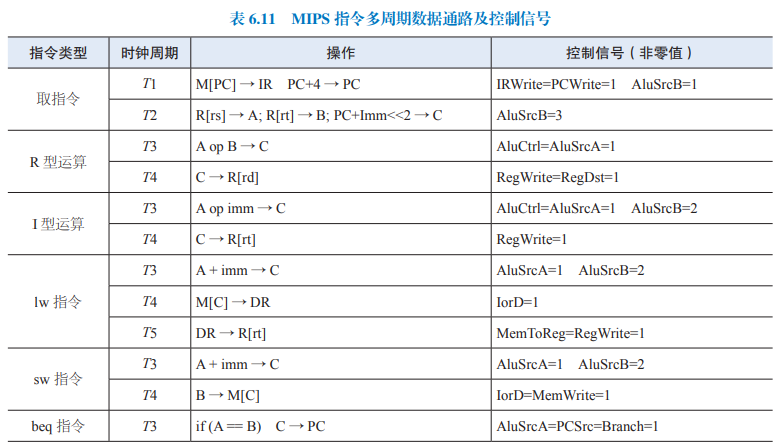
1. R型算术逻辑运算指令执行阶段的数据通路

2. I 型算术逻辑运算指令执行阶段的数据通路

3. lw 指令执行阶段的数据通路

4. sw 指令执行阶段的数据通路

这些都与单周期MIPS CPU类似。整体的转移逻辑如下表所示。



#### R型算术逻辑运算指令执行阶段的数据通路：

在T1时钟周期，PC内容（指令地址）被送入存储器地址寄存器M[PC]，指令被从存储器中读取到指令寄存器IR，同时PC值更新为PC+4。接着在T2时钟周期，根据IR中的RS字段，数据从寄存器堆读取到寄存器A，同时IR中的RT字段对应的寄存器数据或PC+Imm<<2被送入寄存器B。随后在T3时钟周期，ALU对A和B中的数据执行运算（A op B），并将结果送入寄存器C。最后在T4时钟周期，C中的结果被写入寄存器堆中由IR[rd]指定的目标寄存器。

#### I型算术逻辑运算指令执行阶段的数据通路：

T1和T2时钟周期的操作与R型运算指令相同，完成取指令和读取寄存器操作。不同的是，在T3时钟周期，ALU对A和立即数（Imm）执行运算（A op Imm），结果送入寄存器C。接着在T4时钟周期，C中的结果被写入寄存器堆中由IR[rt]指定的目标寄存器。

#### lw指令（load word）执行阶段的数据通路：

其T1和T2时钟周期的操作也与R型运算指令相同，用于取指令和读取寄存器。随后在T3时钟周期，ALU计算出内存地址（A + Imm），并将结果送入寄存器C。在T4时钟周期，C中的地址M[C]所指向的数据被送入数据寄存器DR，完成从内存中读取数据的操作。最终在T5时钟周期，DR中的数据被写入寄存器堆中由IR[rt]指定的目标寄存器。

#### sw指令（store word）执行阶段的数据通路：

T1和T2时钟周期的操作同样与R型运算指令相同，用于取指令和读取寄存器。接着在T3时钟周期，ALU计算出内存地址（A + Imm），并将结果送入寄存器C。最后在T4时钟周期，B中的数据被写入C中的地址M[C]指向的内存位置，完成向内存中写入数据的操作。

这些数据通路均与单周期MIPS CPU的工作方式类似，并且表格中也详细列出了在每个时钟周期中需要激活的控制信号。

## 6.6实验数据测试

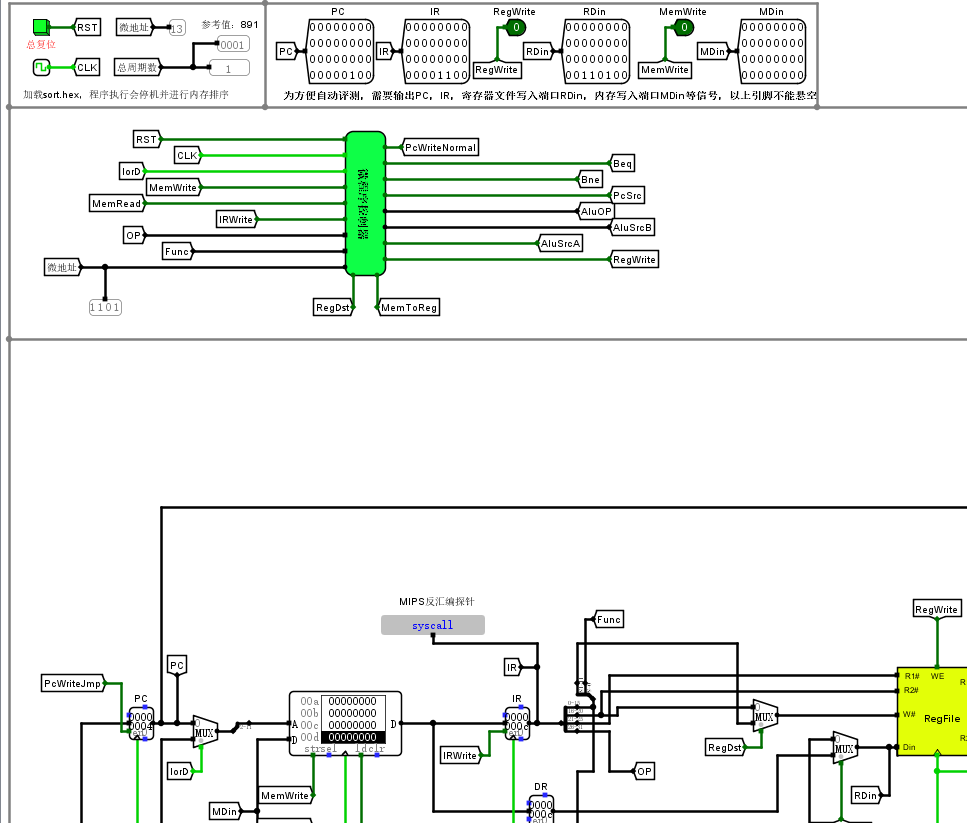
### 6.6.1头歌平台测试结果：

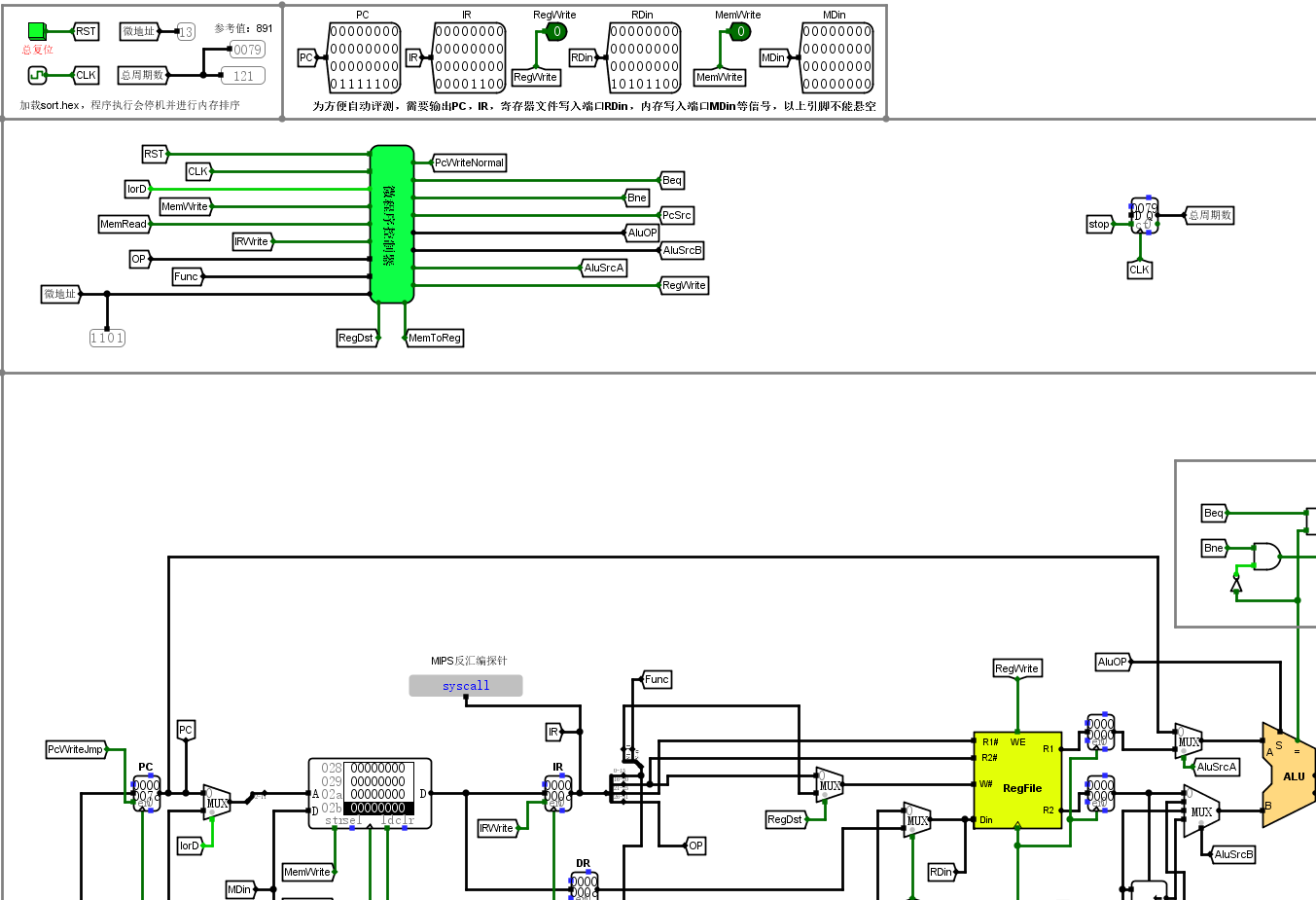
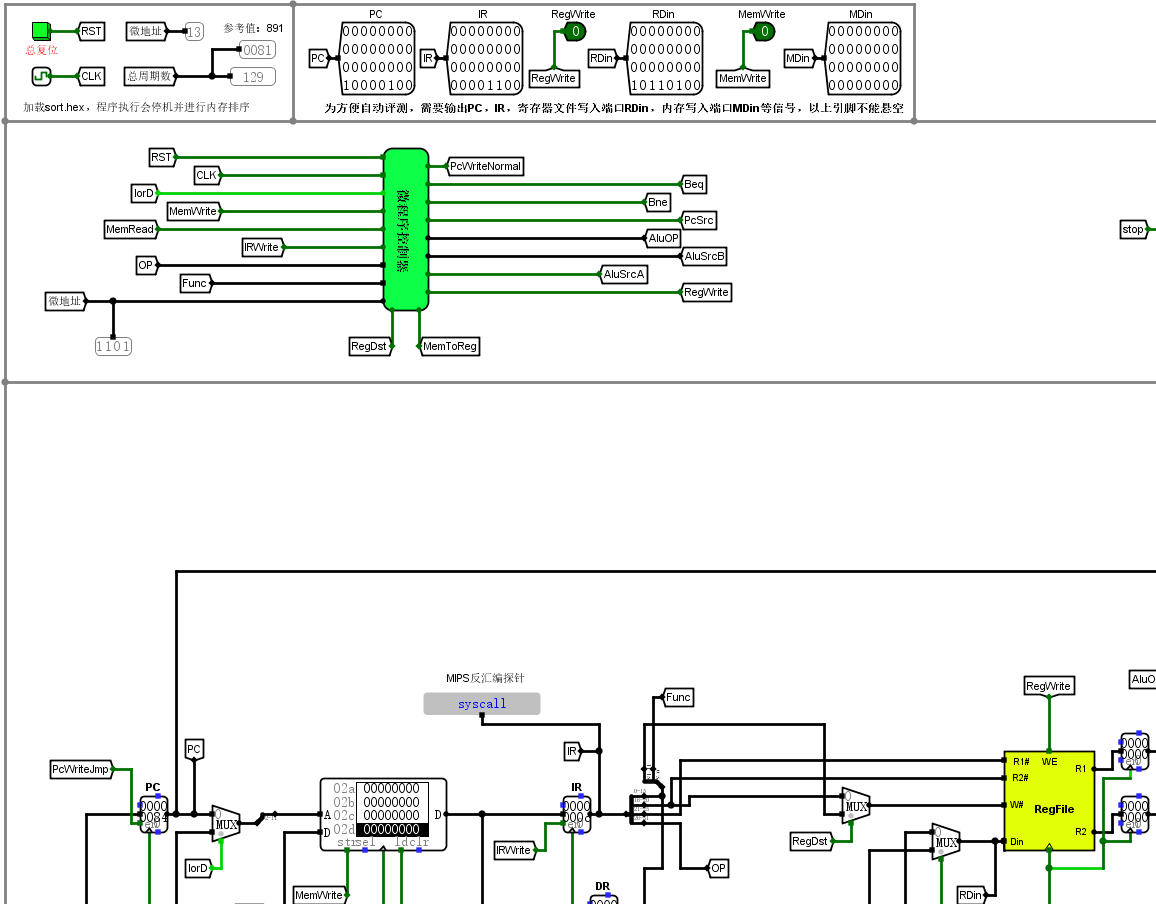
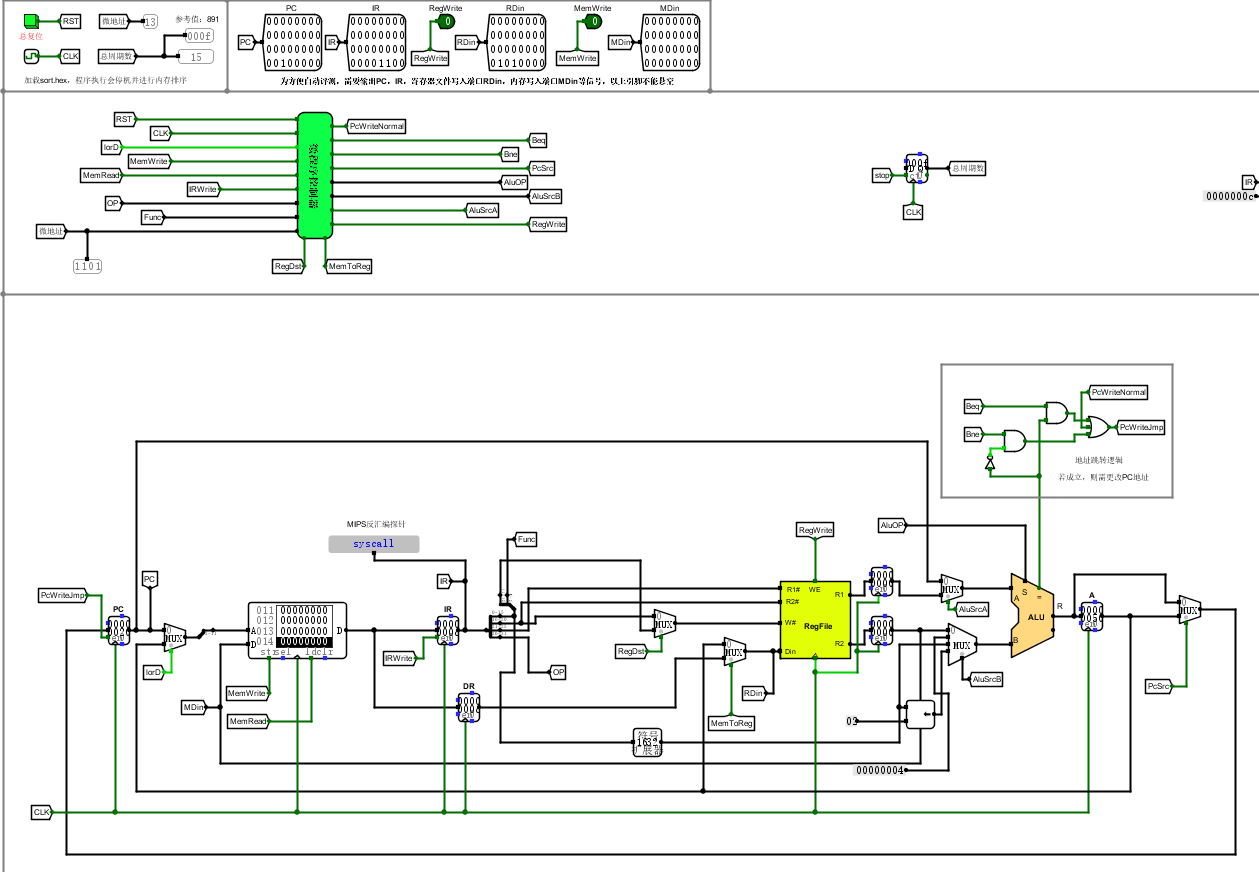
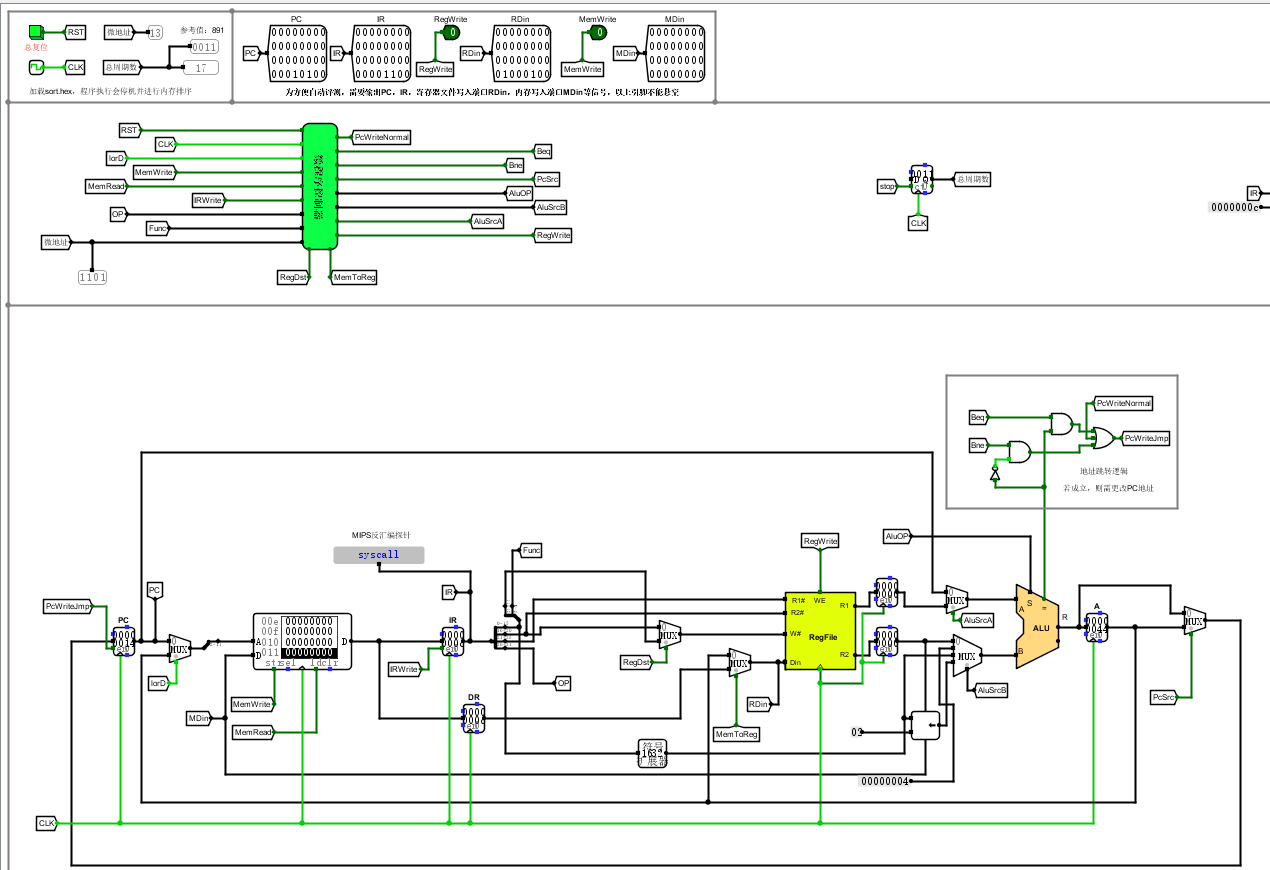
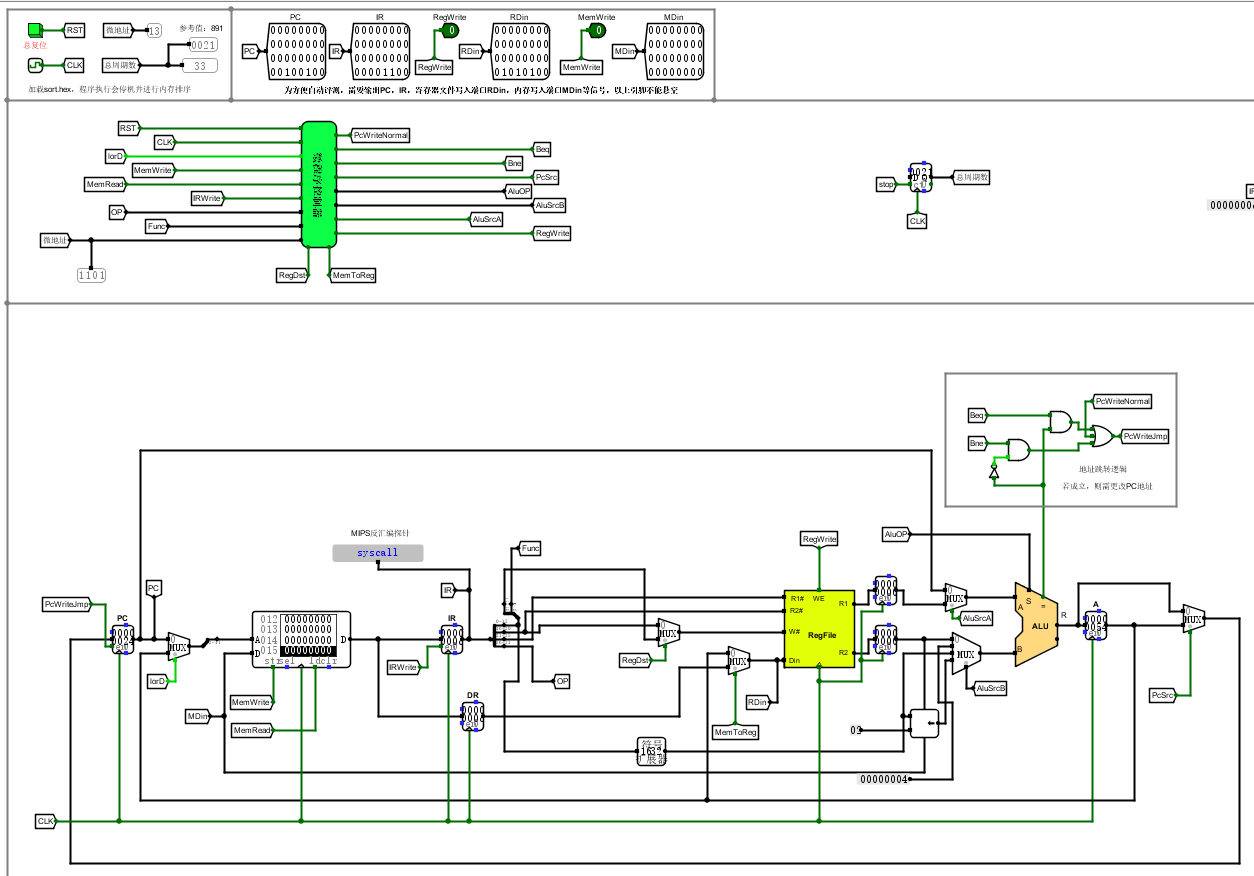
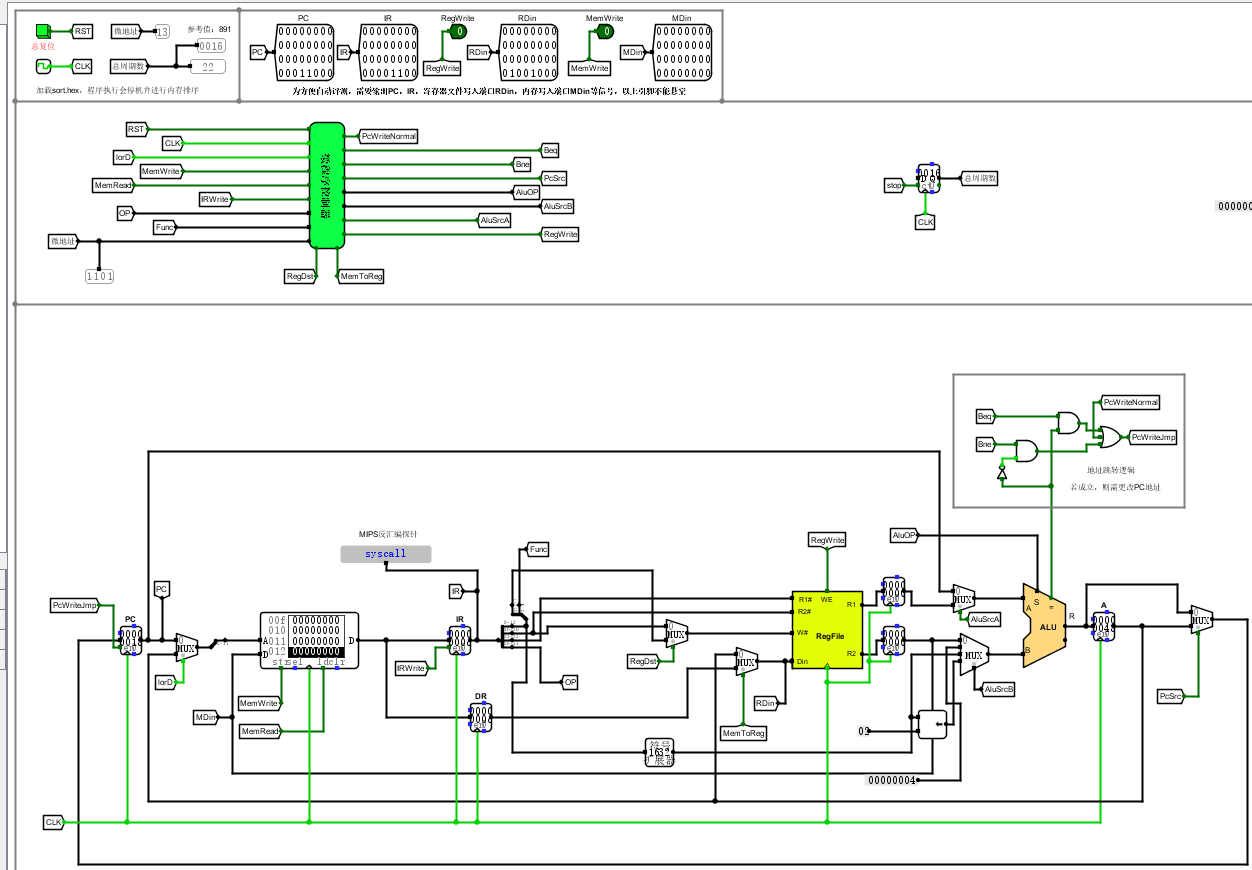


### 6.6.2八条指令单独测试用例的测试结果

每条指令都是从标准Sort.hex程序中选取的相关指令。优先测试Syscall停机指令，并在其他所有测试程序后都添加Syscall指令，以保证测试后停机以供检查。

1. SysCall：



1. Addi 
2. Add 
3. Beq 
4. Bne 
5. Slt 
6. Sw 
7. Lw 

# 实验心得

计算机组成原理课程的所有资料（除课本）都会上传到Github（后续应该不会维护更新）：

<https://github.com/Spercent521/GraduateWorkArchive_CSer>

1. 学会了使用工程化方法生成电路，了解了logisim常用debug方法；
2. 深入理解Cache设计机制，了解接口和模块设计机制；
3. 宏观上掌握了CPU的运行机理和模块划分，而且还学会了从微观上准确地生成每一个控制信号；
4. 熟悉了MIPS核心指令，对MIPS的三类指令有了较为大概的实现思路；
5. 实验最大的感受是对于课程内容理解的加深。在上完课以后往往没有真的学懂知识，在做实验的过程中，自然就会主动地复习上课讲过的内容，而且必须真的弄懂才能完成实验；
6. 实验里有一点不好的体验就是文档更新不及时，常常不知道实验需求，建议老师在就实验过程中同学反馈的关键点在课程学习群中发布公告，或者以文档形式实时更新，供其他同学参考；
7. 课设报告模板不合理，太多重复之处，不能体现工作，负担大于收获。下面左图是华科的报告模板目录，右图是本文的原始模板（只截取CPU实验）。

