**北京邮电大学课程设计报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **课程设计**  **名称** | **计算机组成原理课程设计** | | **学 院** | **计算机** | **指导教师** | **杨秦** |
| **班 级** | **班内序号** | **学 号** | | **学生姓名** | **成绩** | |
| **2019211310** |  | **2019211447** | | **杨元睿** |  | |
| **2019211310** | **27** | **2019211454** | | **左莎莎** |  | |
| **2019211310** |  | **2019211441** | | **王仲天** |  | |
| **2019211311** |  | **2019213692** | | **杨澜** |  | |
| **课**  **程**  **设**  **计**  **内**  **容** | **教学目的**:融会贯通计算机组成原理各章节内容，加深对CPU各模块工作原理及相互联系的认识，掌握硬布线控制器的设计方法，学习运用可编程逻辑技术进行逻辑设计和调试的  **基本内容和实验方法**：按照给定数据格式、指令系统和数据通路，根据所提供的器件要求，自行设计一个基于硬布线控制器的顺序模型处理机。  基本功能：根据设计方案，在 TEC-8 上进行组装、调试运行。  附加功能:（1）在原指令基础上扩指至少三条。（2）允许用户在程序开始时指定 PC 指针的值。自选题目一：在必选题目基础上，完成流水硬连线控制器的设计，在 TEC-8 上进行组装、调试运行。自选题目二：在必选题目基础上，设计实现带有中断功能的硬布线控制器  分组见附页 | | | | | |
| **课程设计**  **报告**  （附页） | 见附页 | | | | | |
| **课程设计成绩**  **评**  **定** | **评语**:  **成绩**:  指导教师签名：  年 月 日 | | | | | |

目录

[题目分析 5](#_Toc76725493)

[1. 实验目的 5](#_Toc76725494)

[2. 实验任务 5](#_Toc76725495)

[3. 任务分析 5](#_Toc76725496)

[实验环境与设备 6](#_Toc76725497)

[团队分工 7](#_Toc76725498)

[原理详解 7](#_Toc76725499)

[4. 硬连线控制器 VS 微程序控制器 7](#_Toc76725500)

[5. 硬连线控制器的基本原理 7](#_Toc76725501)

[6. 模型计算机时序信号 8](#_Toc76725502)

[7. TEC-8 模型计算机组成模块 & 数据通路 8](#_Toc76725503)

[8. 执行一条机器指令的节拍电位数 9](#_Toc76725504)

[9. 流水 VS 非流水 10](#_Toc76725505)

[10. 由非流水变为流水，指令的变化 11](#_Toc76725506)

[设计详解 12](#_Toc76725507)

[11. 指令系统 12](#_Toc76725508)

[12. 机器指令周期流程图设计 13](#_Toc76725509)

[13. EPM7128 器件的引脚 15](#_Toc76725510)

[14. 本实验控制信号详解 17](#_Toc76725511)

[15. 流水设计 19](#_Toc76725512)

[16. ST0的设计 20](#_Toc76725513)

[17. 框架设计 21](#_Toc76725514)

[18. 代码设计 22](#_Toc76725515)

[19. 扩充指令 28](#_Toc76725516)

[20. 修改PC指针功能 29](#_Toc76725517)

[21. 单级中断方案 30](#_Toc76725518)

[日志 30](#_Toc76725519)

[22. 2021、7、2 30](#_Toc76725520)

[23. 2021、7、3 31](#_Toc76725521)

[24. 2021、7、4 31](#_Toc76725522)

[25. 2021、7、5 33](#_Toc76725523)

[26. 2021、7、6 34](#_Toc76725524)

[27. 2021、7、7 34](#_Toc76725525)

[测试程序&记录 35](#_Toc76725526)

[28. 跳转测试 36](#_Toc76725527)

[29. 循环和任意位置PC测试 37](#_Toc76725528)

[30. 扩展指令测试 38](#_Toc76725529)

[31. 老师指定程序测试 39](#_Toc76725530)

[程序仿真 42](#_Toc76725531)

[遇到的问题及解决 43](#_Toc76725532)

[32. TEC-8控制台的W1,W2,W3灯不亮 43](#_Toc76725533)

[33. W1、W2、W3的含义以及如何设计? 43](#_Toc76725534)

[34. ST0应该在什么时候改变值？ 44](#_Toc76725535)

[35. 进位标志C是如何变化的? 44](#_Toc76725536)

[36. 在流水硬布线中，有关nop指令的设置是怎么样的？ 44](#_Toc76725537)

[37. 参考流程图中，为什么ADD中没有LDC？ 44](#_Toc76725538)

[38. 如何设计按下CLR之后IR里的内容置为00H? 44](#_Toc76725539)

[39. T3的上升沿和下降沿分别完成什么工作？ 45](#_Toc76725540)

[40. PCINC和LAR必须指定在T3的上升沿进行吗？ 45](#_Toc76725541)

[41. SST0如何赋值和如何给ST0赋值？ 45](#_Toc76725542)

[42. 操作错误 45](#_Toc76725543)

[设计调试小结 46](#_Toc76725544)

[各成员心得总结 46](#_Toc76725545)

[硬连线控制器设计]  
计算机组成原理课程设计

王仲天2019211441杨元睿2019211447左莎莎 2019211454 杨澜 2019213692

北京邮电大学计算机学院

2021/7/2

# 题目分析

## 实验目的

* **融会贯通计算机组成原理课程各章节的内容**，通过知识的综合运用加深对计算机系统各模块的工作原理及相互联系的认识，特别是对硬布线控制器的认识。建立清晰的整机概念
* **掌握硬布线控制器的设计方法，掌握流水硬连线控制器的设计方法。**
* **学习运用可编程逻辑技术进行逻辑设计和调试的基本步骤和方法**，熟悉集成开发软件中设计、模拟调试工具的使用，体会可编程逻辑技术相对于传统开发技术的优点
* **培养科学研究的独立工作能力**，取得工程设计与组装调试的实践经验
* 学习运用当代的 EDA 设计工具，掌握用 EDA 设计大规模复杂逻辑电路的方法。

## 实验任务

* **基于Altera CPM7128的硬连线控制器设计**，按照给定数据格式、指令系统和数据通路，根据所提供的器件要求，自行设计一个基于硬布线控制器的顺序模型处理机，根据设计方案，在TEC-8上进行组装、调试运行，在原指令基础上要求扩指至少三条，修改PC指针功能（任意指针）。
* **流水线设计，**在必选题目基础上，完成流水硬连线控制器的设计根据设计方案，在TEC-8上进行组装、调试运行。

## 任务分析

根据课本上的指令系统和 TEC-8 的数据通路，在 EPM7128SLC84-15N 芯片上编写一个硬布线控制器，实现以下的基本功能：

* 能够完成控制台操作：启动程序运行、读存储器、写存储器、读寄存器和写寄存器。可以在任意位置读写存储器。
* 可以分析指令各个节拍的作用，并能操作硬件实现指令功能。完成如下指令：

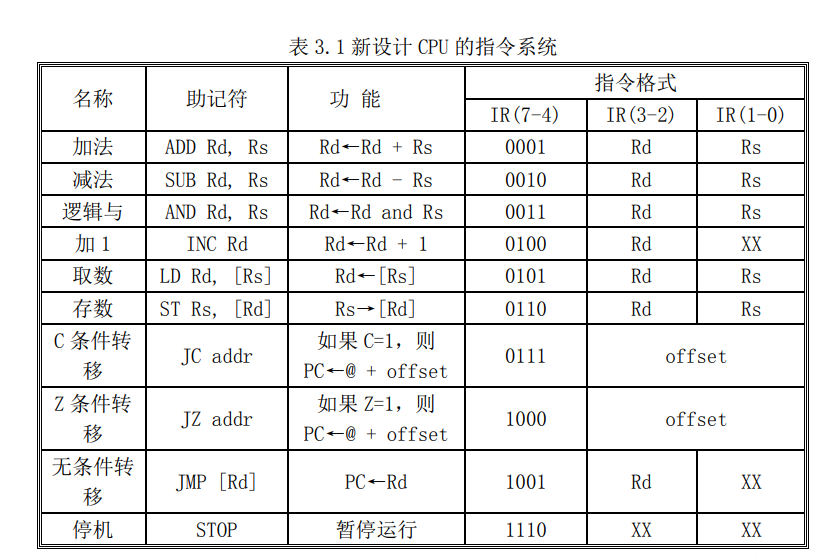


表 13.1 中，XX 代表随意值。Rs 代表源寄存器号，Rd 代表目的寄存器号。在条件转移指令中，@代表当前 PC的值，offset 是一个 4 位的有符号数，第 3 位是符号位，0 代表正数，

* 在完成基本功能的前提下，实现指令多级流水，提高指令执行效率。
* 拓展指令集，让机器能够运行更多指令。
* 实现取指阶段的指针，让机器能够在任意位置开始执行指令。
* 在当前机器上尝试设计单级中断方案。

# 实验环境与设备

* TEC-8 控制台一个，本次设计在 TEC-8 模型机上完成。
* PC一台（Win 7）
* TEC-8 程序下载线
* TEC-8实验系统平台提供芯片：Altera MAX7000系列CPLD芯片-EPM7128
* Quatus2软件平台

# 团队分工

**共同完成：**日志由各组员每天轮流记录，共同参与程序原理讨论，共同完成程序流程设计

**杨元睿**：测试设备，运行测试程序，调试程序。

**左莎莎**：设计流水线，报告的总体撰写，参与调试问题的讨论。

**王仲天**：非流水版本编写与维护，参与调试问题的讨论

**杨澜**：流水版本编写和维护，参与调试问题的讨论

# 原理详解

## 硬连线控制器 VS 微程序控制器

* 微程序控制器的控制信号以微指令周期为时间单位，硬连线控制器以节拍电位(CPU 周期)为时间单位，两者在本质上是一样的，1 个节拍电位时间和 1 条微指令时间都是从节拍脉冲 T1 的上升沿到 T3 的下降沿的一段时间。
* 在微程序控制器流程图中，一个执行框代表一条微指令，在硬连线控制器流程图中，一个执行框代表一个节拍电位时间。
* 硬连线控制器通过每条指令读取时执行相应的硬件管脚信号来实现一条指令对应多个微操作，操作多个硬件来实现一条指令的复杂功能。而微程序控制器则通过将一条指令所需要的微操作制作成微指令存储到微指令存储器当中，当读取一条指令的时候，通过将微指令存储器中的微指令读出到微指令寄存器中，再依次执行，可以实现一条指令的执行。
* 硬连线控制器更加高效，因为我们通过逻辑操作控制不同微操作的执行，而微程序控制器则需要从微指令存储器中读取，效率较低。同样的，硬连线控制器因为需要实现复杂的逻辑操作，实现难度较高。

## 硬连线控制器的基本原理

硬连线控制器的基本原理，每个微操作控制信号 S 是一系列输入量的逻辑函数，即用组合逻辑来实现：

S = f (Im,Mi,Tk,Bj)，

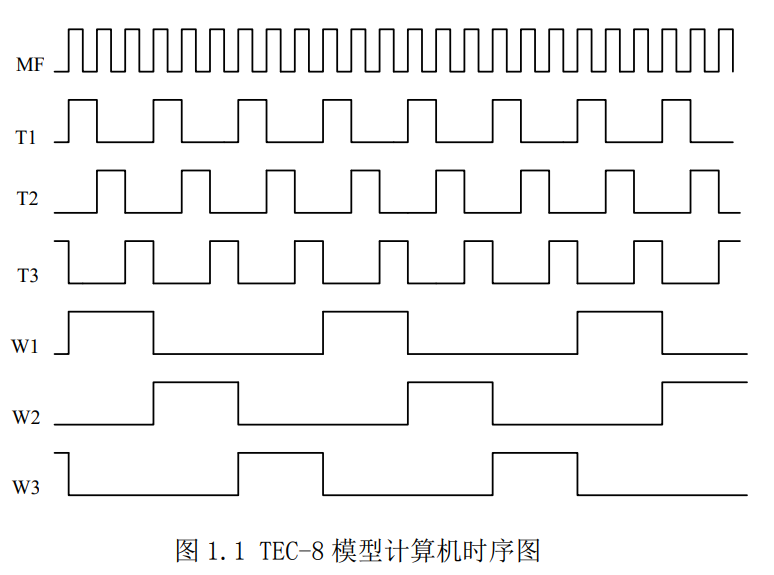
Im 是机器指令操作码译码器的输出信号，因为机器指令系统比较简单，省去操作码译码器，4 位指令操作码 IR4～IR7 直接成为 Im 的一部分；由于 TEC-8 实验系统有控制台操作，控制台操作可以看作一些特殊的功能复杂的指令，因此SWC、SWB、SWA 可以看作是 Im 的另一部分。

Mi 是节拍电位信号，是时序发生器产生的节拍信号 W1～W3；

Tk 是节拍脉冲信号，

Bj 是状态条件信号，包括 ALU 产生的进位信号 C、结果为 0 信号 Z 等等。

## 模型计算机时序信号

TEC-8 模型计算机主时钟 MF 的频率为 1M H z，执行一条微指令需要 3 个节拍脉冲 T 1、T 2、T 3。TEC-8 模型计算机时序采用不定长机器周期，绝大多数指令采用 2 个机器周期 W 1、W 2，少数指令采用一个机器周期 W 1 或者 3 个机器周期 W 1、W 2、W 3。

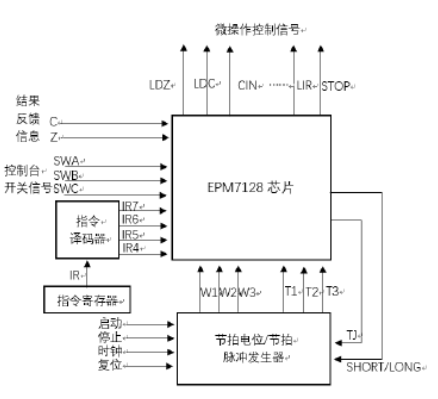
我们实验所涉及的信号，其中：LDZ、LDC、DRW、PCINC、LPC、LAR、ARINC、LIR均在T3的上升沿置入相应的数或者执行相应的操作，MEMW在T2为1期间将DBUS上的数写入存储单元。

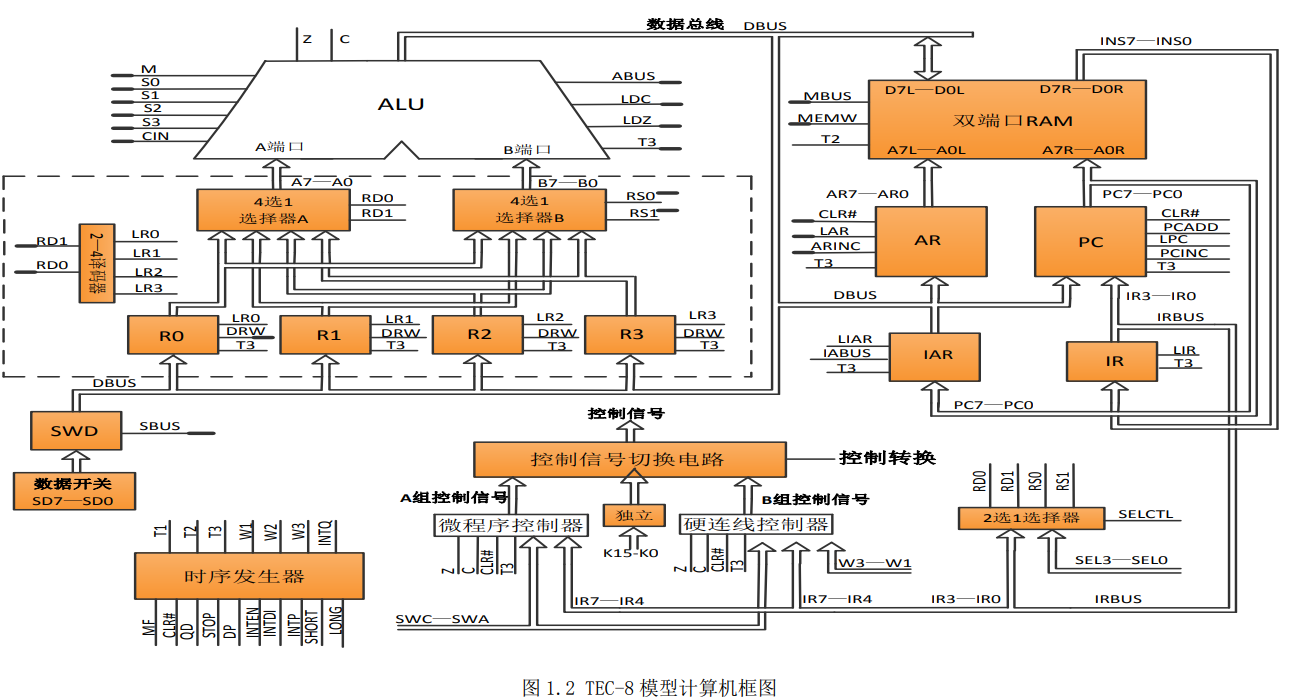
综上，基本上我们实验中的所有操作在T3上升沿之后就已经基本完成了，这就为我们后面的流水设计：从T3为1就可以将LIR、PCINC置为1，同时，其它信号相应进行变换，从而留下充足的时间来避免时间冲突：吞指令。

同时，由于EMP7128给我们的牵的引脚就是T3,没有T1, T2, 这也是我们为什么在程序中只出现T3的原因。

## TEC-8 模型计算机组成模块 & 数据通路

主要组成模块：时序发生器、算逻运算单元 ALU、双端口寄存器组 7064、数据开关 SD、双端口存储器 7132、组合逻辑控制器、微程序控制器、若干寄存器和若干选择器等





## 执行一条机器指令的节拍电位数

在 TEC-8 实验系统中，采用了可变节拍电位数来执行一条机器指令。大部分指令的执行只需 2 个节拍电位 W1、W2，少数指令需要 3 个节拍电位 W1、W2、W3。为了满足这种要求，在执行一条指令时除了产生完成指令功能所需的微操作控制信号外，对需要 3 个电位节拍的指令，还要求它在 W2 时产生一个信号 LONG。信号 LONG 送往时序信号发生器，时序信号发生器接到信号 LONG 后产生节拍电位 W3。

对于一些控制台操作，需要 4 个节拍电位才能完成规定的功能。为了满足这种情况，可以将控制台操作化成两条机器指令的节拍。为了区分写寄存器操作的 2 个不同阶段，可以用某些特殊的寄存器标志标。例如建立一个 ST0标志，当 ST0=0 时，表示该控制台操作的第 1 个 W1、W2；当 ST0=1 时，表示该控制台操作的第 2 个 W1、W2。

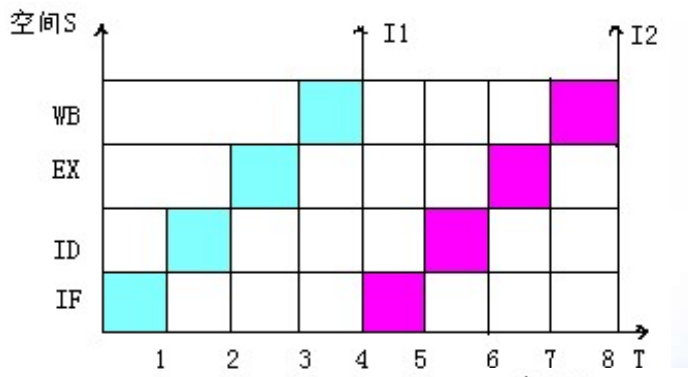
为了适应更为广泛的情况，TEC-8 的时序信号发生器允许只产生一个节拍电位 W1。当1 条指令或者一个控制台在 W1 时，只要产生信号 SHORT，该信号送往时序信号发生器，则时序信号发生器在 W1 后不产生节拍电位 W2，下一个节拍仍是 W1。

信号 LONG 和 SHORT 只对紧跟其后的第一个节拍电位的产生起作用。

## 流水 VS 非流水

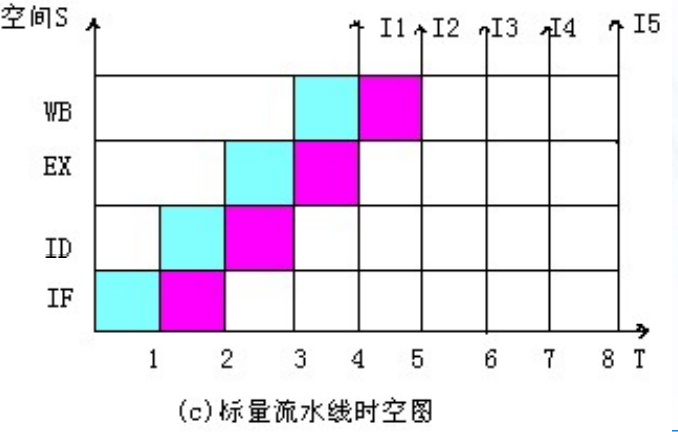
流水可以大幅提高我们的执行效率。

对于非流水：



当没有流水，执行每一条指令时的第一个节拍电位都是用来执行 LIR,PC I NC 读取指令的。如上图所示，一条指令就占用了4个时钟周期。

流水：



可以看到当同样运行了两条指令之后，流水硬连线的执行时间是非流水的 5/8。而且，流水层级越高，执行时间越长，流水的效率就越高

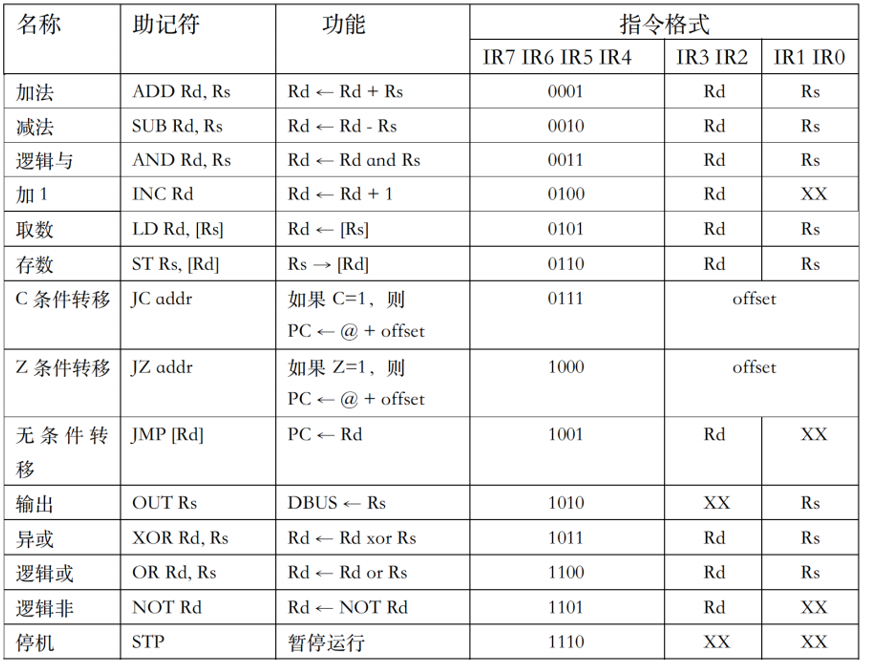
## 由非流水变为流水，指令的变化

我们将原来非流水中的单独为一个节拍的取指令LIR、PCINC转换到每条指令的执行节拍里，这样不会与读写寄存器和读写存储器相冲突，因为不在同一条数据通路上。

那么，在LIR和PCINC都放在了执行节拍里，那最开始我们怎么进入到指令的执行呢？我们将IR为00H的指令设置为nop,这样，在最开始的时候，就可以读取指令，开始流水了。

# 设计详解

## 指令系统



| **名称** | **助记符** | **功能** | **指令格式** |
| --- | --- | --- | --- |
| 空指令 | NOP | PC<——PC+1 | 0000 XXXX |

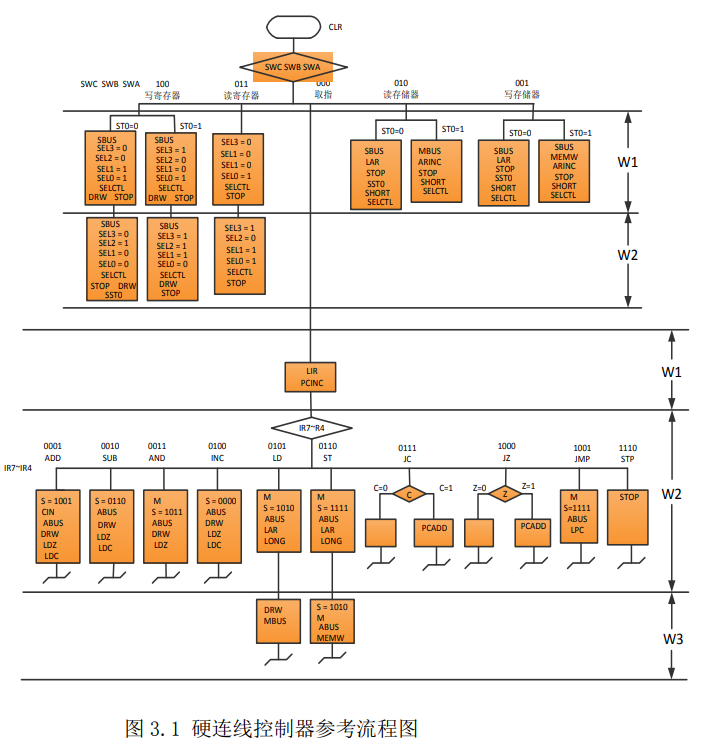
ALU算术/逻辑 运算功能表：



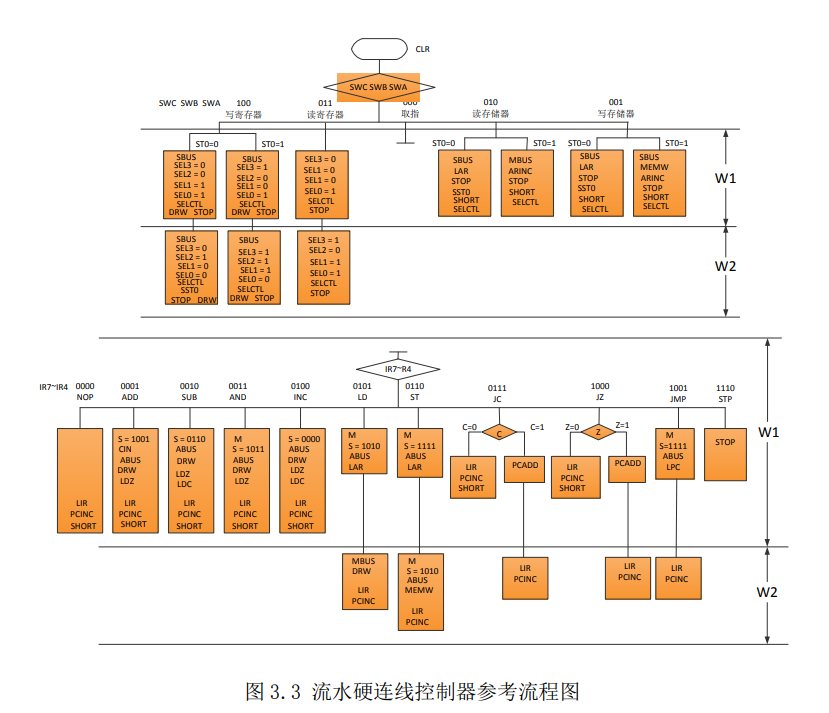
## 机器指令周期流程图设计

在硬连线控制器中，控制台操作的流程图与机器指令流程图类似，下图画出了硬连线控制器的机器周期参考流程图。

**非流水版本**



**流水版本**

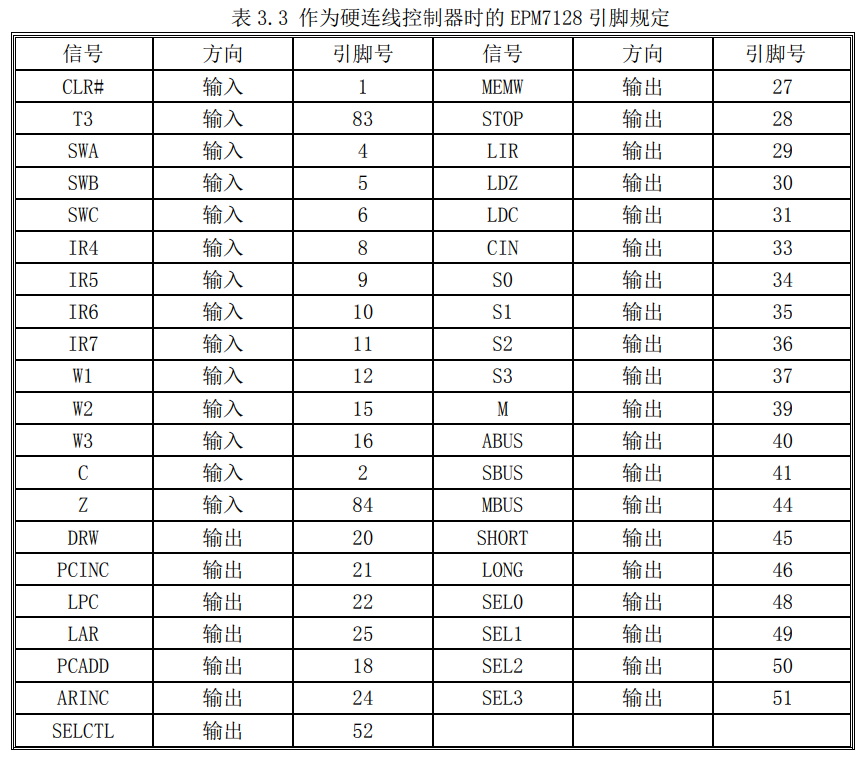


我们使用 VHDL 语言设计时，是直接根据流程图直接写出相应的语言描述。这种方法省略了译码表，且不容易出错。

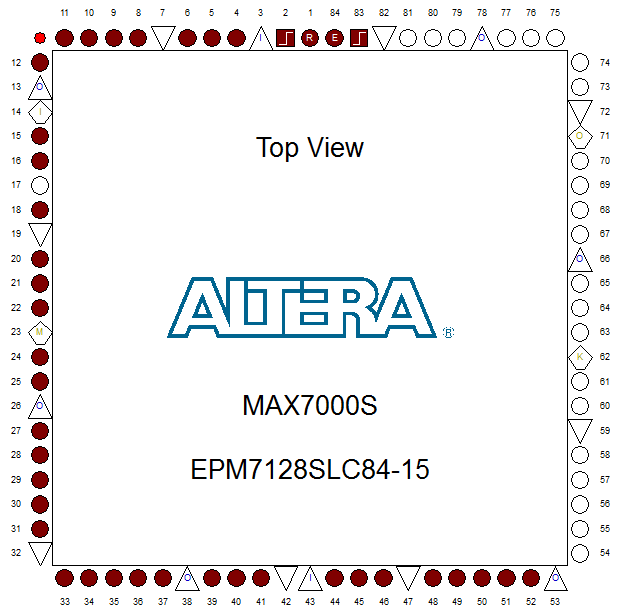
在 TEC-8 实验系统中，由于按下复位按钮 CLR 后，指令寄存器 IR 被复位为 00H，因此，无论存储器中写入什么测试程序，在图 3.3 所示的流程图中，第 1 条被执行的指令的代码总是 00H。我们规定，指令代码为 00H 的指令是条 nop 指令。

## EPM7128 器件的引脚

TEC-8 实验系统中的硬连线控制器是用 1 片 EPM7128 器件构成的。硬连线控制器和数据通路之间不采用接插线方式连接，在印制电路板上已经用印制导线进行了连接。这就要求硬连线控制器所需的信号的输出、输入信号的引脚号必须符合表 3.3 中的规定。



**引脚连接图：**



## 本实验控制信号详解

| **名称** | **程序中的类型** | **功能** |
| --- | --- | --- |
| CIR | In std\_logic | 清零信号 |
| T3, W1, W2, W3 | In std\_logic | 时序信号 |
| C, Z | In std\_logic | 标志信号 |
| LDZ | Out std\_logic | 当它为1时，如果运算结果为0，在T3的上升沿，将1写入寄存器，如果运算结果不为0，将0保存到Z标志寄存器 |
| LDC | Out std\_logic | 当它为1时，在T3的上升沿将运算得到的进位保存到C标志寄存器 |
| CIN | Out std\_logic | 进位输出 |
| M | Out std\_logic | 控制算术逻辑运算类型 |
| ABUS | Out std\_logic | 当它为1时，将开关数据送到数据总线DBUS,当它为0时，禁止开关数据送数据总线DBUS |
| DRW | Out STD\_LOGIC; | 为1时,在T3上升浴对RD1、RDO选中的寄存器进行写操作，将数据总线DBUS上的数D7~D0写入选定的寄存器。 |
| PCINC | Out std\_logic | 当它为1时，在T3的上升沿PC加1。 |
| LPC | Out std\_logic | 当它为1时,在T3的上升沿,将数据总线DBUS上的 D7~D0写入程序计数器PC。 |
| LAR | Out std\_logic | 当它为1时,在T3的上升沿,将数据总线DBUS上的D7~DO写入地址寄存器AR。 |
| PCADD | Out std\_logic | 当它为1时,将当前的PC值加上相对转移量,生成新的PC。 |
| ARINC | Out std\_logic | 当它为1时，在T3的上升沿AR加 1。 |
| SELCTL | Out std\_logic | 当它为1时，TEC-8实验系统处于实验台状态，当它为0时，TEC-8实验系统处于运行程序状态。 |
| MEMW | Out std\_logic | 当它为1时，在T2为1期间将数据总线DBUS上的D7~DO, 写入双端口RAM。写入的存储器单元由AR7~AR0指定。 |
| STOP | Out std\_logic | 当它为1时，在T3结束后时序发生器停止输出节拍脉冲T1,T2,T3 |
| LIR | Out std\_logic | 当它为1时，在T3的上升沿将从双端口RAM的右端口读出的指合 INS7~INSO写入指合寄存器IR。读出的存储器单元由PC7~PCO指定。 |
| SBUS | Out std\_logic | 当它为1时数据开关SD7~SDO的数送数据总线DBUS。 |
| MBUS | Out std\_logic | 当它为1时,将双端口RAM的左端口数据送到数据总线 |
| SHORT | Out std\_logic | 在W1时置为1可以让其不再产生W2 |
| LONG | Out std\_logic | 在W2时置为1可以让其产生节拍W3 |
| SWC, SWB, SWA | In std\_logic | 控制台方式控制信号 |
| IRH | In std\_logic(3 downto 0) | 指令的操作码 |
| S | In std\_logic(3 downto 0) | ALU的方式控制，与M配合以选择逻辑或算数运 |
| SEL | In std\_logic(3 downto 0) | 寄存器选择向量 |

## 流水设计

取指令周期设计

如12机器指令周期流程图设计，下一条指令的取指令周期放在上一条命令的最后一个周期，所以，我们在每条指令的最后一个机器周期加上LIR PCINC指令。而且，因为指令大多只需要一个节拍电位，所以， 这样的设计基本上就可以实现两条流水功能。

**设想：能否设计为三条及以上流水？**

如果三条流水，那么意味着同一时刻内执行着3个周期的任务。一旦三条流水遇到连续的2周期指令，那么必定会出现一个节拍电位出现两个以上的取值周期，会引起资源冲突。同时，我们可以得知指令周期的最小值是流水跟流水线数的最大值相关。

流水情况分析

设计的所有指令都是两周期或者三周期，因此在进行流水并行执行时只会出现以下4种情况：



## ST0的设计

**ST0为0的情况：**

* 在初始化时以及按下clr时应该被置为0
* 修改PC指针时为0

**ST0为1的情况：**

* 写寄存器的W2节拍且ST0=0（将四拍换成两个两拍，用ST0区别）
* 读存储器的W1节拍且ST0=0（用ST0区分获取地址和读操作）
* 写存储器的W1节拍且ST0=0（用ST0区分获取地址和写操作）
* 执行指令（在修改了PC指针之后）

在其它情况下就保持原来的值不变（即使为1），如此一来，便成功使用了ST0,且避免了ST0的无故反转而影响后续操作。

在ST0为1的情况下不能随意更改ST0的值，以便实现循环操作，除非按下CLR。

综上：**只有在按下CLR时ST0才能被置为0，将ST0置为1之后就不会改变ST0的值**

**ST0变化时刻：**

现 ST0 标志位的变化不能在节拍电位的中间时刻进行，而应该在 T3 脉冲的下降沿，即一个节拍电位的结束时刻进行。如果在节拍电位的中间时刻变化了 ST0，则发出的信号可能发生错误。

## 框架设计

修改ST0进程

* st\_PC\_IR是标识T3变化的信号，在取指令process中会用到。
* process (T3, CLR)
* begin
* st\_PC\_IR <= '0';
* if (CLR = '0') then
* ST0 <= '0';
* else
* if (T3'event and T3 = '0') and SST0 = '1' then
* ST0 <= '1';
* end if;
* st\_PC\_IR <= '1';
* end if;
* end process;

取指令进程

* st\_PC\_IR是标识T3变化的信号，在修改ST0的process中赋值。st\_PC\_IR为1即T3变化时，根据当前的SWCBA, ST0, IRH, W1, W2判断是否取指令。
* process (st\_PC\_IR)
* begin
* LIR <= '0';
* PCINC <= '0';
* if (st\_PC\_IR = '1') then
* case SWCBA is
* when "000" =>
* if (ST0 = '1') then
* case IRH is
* when "0000" => --NOP
* LIR <= W1;
* PCINC <= W1;
* 。。。。。。

运行指令进程

具体内容见下方代码设计

## 代码设计

* 我们是直接根据流程图设计的代码，没有使用中间的组合逻辑译码。当然，也会涉及到组合逻辑，比如:写寄存器，执行各种指令等等,为了精减代码也会进行组合逻辑，但思路简单，故不再另外做出译码表。
* 冯·诺伊曼型计算机的 CPU，其工作可以分为 5 个阶段：取指令、指令译码、执行指令、访存取数、结果写回。在这里，由于指令系统比较简单，我们省去了指令译码这一步。访存取数、结果写回也可以看成是执行指令的一部分。因此我们可以将 CPU 的工作简单分为取指令、执行指令两个阶段。

先把所有的输出信号先置0

将输出信号初始化，每次启动进程，即每次有变量变化，都会启动进程，先将这些信号置为0。

我们采用时序逻辑，在进程的最开始先把所有的输出信号置为0，这样，就可以使得在进程结束之后没有被后面的程序所覆盖的信号直接为0，不会成为错误信号干扰程序的进行。

* DRW <= '0';
* PCINC <= '0';
* LPC <= '0';
* LAR <= '0';
* PCADD <= '0';
* ARINC <= '0';
* SELCTL <= '0';
* MEMW <= '0';
* STOP <= '0';
* LIR <= '0';
* LDZ <= '0';
* LDC <= '0';
* CIN <= '0';
* S3\_0 <= "0000";
* M <= '0';
* ABUS <= '0';
* SBUS <= '0';
* MBUS <= '0';
* SHORT <= '0';
* LONG <= '0';
* SEL3 <= '0';
* SEL2 <= '0';
* SEL1 <= '0';
* SEL0 <= '0';
* SBUS <= '0';

同时，我们根据流程图新增加信号 ST0 和SST0, SST0决定ST0为0还是为1，ST0则表示在写寄存器，读存储器和写存储器时进入到哪个一个分支（2个分支）以及是否处于PC指针的修改状态（为0为指针修改状态，为1为程序的执行状态）。所以，我们初始化SST0为：

* SST0 <= '0';

CLR处理

首先进行异步控制，判断如果按下了 CLR，就将 ST0 置为 0，否则进入我们的主程序。

* if (CLR = '0') then
* ST0 <= '0';

SST0

如果 SST0 为 1，则在 T3 下降沿对 ST0 的值做出修改。

* if (T3'event and T3 = '0') and SST0 = '1' then
* ST0 <= '1';
* end if;

根据操作模式开关进入到不同操作模式：

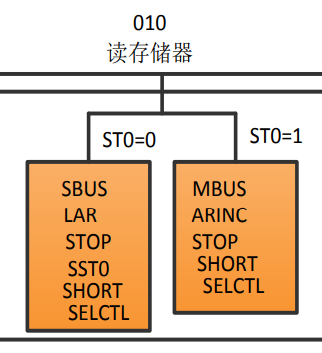
就通过 case 来判断 SWC SWA，执行读存储器、写存储器、读寄存器、写寄存器、执行程序。

* case SWCBA is
* when "000" =>
* 。。。
* when "001" =>
* 。。。
* when "010" =>
* .。。。
* when "011" =>
* .。。。
* when "100" =>
* .。。。
* when others => null;
* end case;

读存储器

读存储器可以分为两个步骤：指定首地址，依次读取存储单元。我们可以通过 ST 0来区分这两个步骤。因为我们只需要在W1节拍里进行操作，所以，我们需要将short置1.

如流程图所示：



开始时 ST 0 = 0，我们打开 SBU S，打开 L AR，将数据开关的值送入地址寄存器 AR，这样就实现了指定首地址。同时我们将 SST 0 置为 1，表示我们即将进入 ST 0 = 1，即读存储器的阶段。

读存储器时，我们将 MBU S 打开，就可以把当前指定存储单元的值读到数据总线上，在 D7 ∼ D0 指示灯观察。然后我们执行 AR I NC 来将 AR 加一，在下一个节拍中读取下一个存储单元的值。

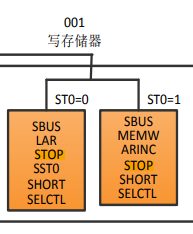
同时，在读取了一次存储器之后，因为SWCBA不变，ST0依旧为1，每按一次QD就可以一次读取下一个存储单元的地址，直到出现了CLR，使得ST0为0，取消读存储器模式。

代码设计如下：

* SELCTL <= W1;
* SHORT <= W1;
* SBUS <= W1 and (not ST0);
* MBUS <= W1 and ST0;
* STOP <= W1;
* SST0 <= W1;
* LAR <= W1 and (not ST0);
* ARINC <= W1 and ST0;

写存储器

写存储器也可以分为两个步骤：指定首地址，依次写入存储单元。我们可以通过ST 0 来区分这两个步骤。因为我们只需要在W1节拍里进行操作，所以，我们需要将short置1.流程图如下：



开始时 ST 0 = 0，我们打开 SBU S，打开 L AR，将数据开关的值送入地址寄存器 AR，这样就实现了指定首地址。同时我们将 SST 0 置为 1，表示我们即将进入 ST 0 = 1，即写存储器的阶段。

写存储器时，我们将 MEMW 打开，就可以把当前数据总线上的值存入指定存储单元。然后我们执行 AR I NC 来将 AR 加一，在下一个节拍中向下一个存储单元写入值。

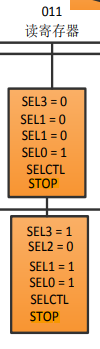
同时，在写入了一次存储器之后，因为SWCBA不变，ST0依旧为1，每按一次QD就可以一次写入下一个存储单元地址，直到出现了CLR，使得ST0为0，取消写存储器模式。

代码设计如下：

* SELCTL <= W1;
* SHORT <= W1;
* SBUS <= W1;
* STOP <= W1;
* SST0 <= W1;
* LAR <= W1 and (not ST0);
* ARINC <= W1 and ST0;
* MEMW <= W1 and ST0;

读寄存器

读寄存器需要 2 个节拍电位，，如下所示：



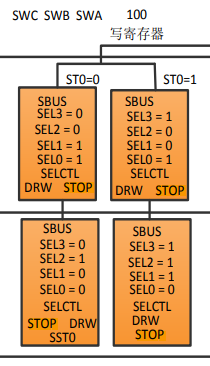
W 1 时我们将 SEL3、SEL2 设为 00，将 SEL1、SEL0设为 01，这时寄存器 R0 的值会送往 ALU 的 A 端口，R1 的值会送往 ALU 的 B 端口。我们可以通过指示灯 A7 ∼ A0、B7 ∼ B0 来观察到这两个值。同理，在 W 2 节拍我们可以读出 R2、R3 寄存器的值。

代码设计如下：

* SELCTL <= '1';
* SEL0 <= W1 or W2;
* STOP <= W1 or W2;
* SEL3 <= W2;
* SEL1 <= W2;

写寄存器

写寄存器需要 4 个节拍电位，我们将其化成两条机器指令的节拍，通过 ST 0 来区分。如下所示：



我们需要先打开 SBU S 和 DRW ，SBU S = 1 允许我们将数据开关的值送到数据总线上，DRW = 1 时在 T 3 上升沿对 RD1、RD0，即 SEL3、SEL2 选中的寄存器进行写操作，将数据总线上的值写入选定的寄存器。

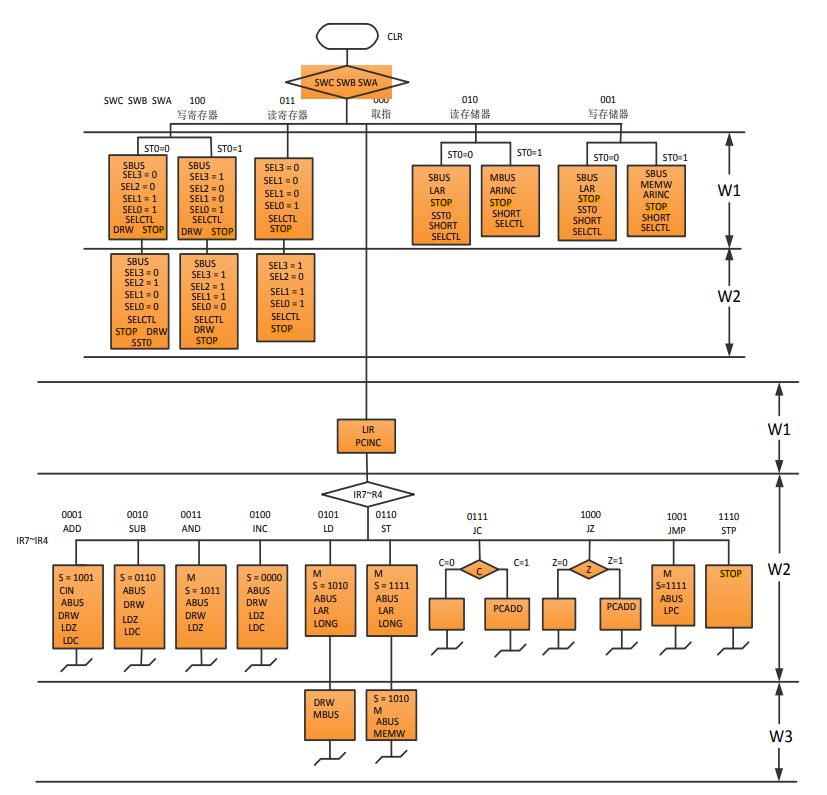
写寄存器需要 4 个节拍电位，我们将其化成两条机器指令的节拍，通过 ST 0 来区分。第一个 W 1 时我们将 SEL3、SEL2 设为 00，将 SEL1、SEL0 设为 11，表示向 R0 寄存器里写入值。接下来 W 2 我们将 SEL3、SEL2 设为 01，将 SEL1、SEL0 设为 00，表示向 R1 写入值，并读取 R0 的值。之后也类似，每次 SEL3、SEL2 指定当前写入的寄存器，SEL1、SEL0 指定上次写入的寄存器，这样每次**可以在 ALU 的 B 端口，即 B7 ∼ B0指示灯检查上次写入的值是否正确**。注意到在第一个 W 2 节拍时，将 SST 0 标志置为 1，然后在 T 3 的下降沿就会将 ST 0 从 0 变为 1，表示我们进入到第二个 W 1、W 2。

代码设计如下：

* SELCTL <= '1';
* SST0 <= W2;
* SBUS <= W1 or W2;
* STOP <= W1 or W2;
* DRW <= W1 or W2;
* SEL3 <= (ST0 and W1) or (ST0 and W2);
* SEL2 <= W2;
* SEL1 <= ((not ST0) and W1) or (ST0 and W2);
* SEL0 <= W1;;

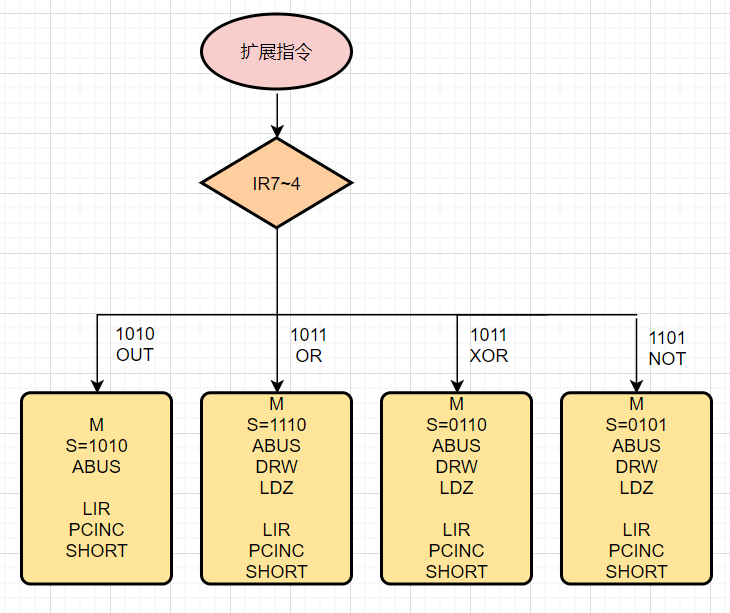
执行程序

把程序存到存储器里，设置寄存器和存储器的初始值，然后开始执行。流程图如下：



## 扩充指令

在原指令的基础上，我们可以再扩展四条：OUT OR XOR NOT，可以根据 ALU的逻辑运算表来实现。设计如下



## 修改PC指针功能

通过 ST 0 来实现这个功能，与我们在读写存储器里的操作类似。初始时ST 0 = 0，我们打开 SBU S 和 LPC，将数据开关上的值存到 PC 里，这就实现了指定程序首地址，同时我们将 SST 0 置为 1。在 ST 0 = 1 阶段我们就开始逐步执行我们的程序。代码设计如下：

* if ST0 = '0' then
* LPC <= W1;
* SBUS <= W1;
* SST0 <= W1;
* SHORT <= W1;
* STOP <= W1;

同时，可以注意到我们这部分的程序没有取指令功能，这是因为我们有指令为00H的nop指令，可以有这条指令进入读取下一条指令，同时，在之后我们的每一条指令后面都可以读取指令。

## 单级中断方案

我们组想了一下关于中断的执行。单级中断方案的中断步骤为关中断->保护断点->保护现场->中断服务->恢复现场->开中断->中断返回

首先是关于中断信号的处理，我们最开始想的是EPM7128的引脚没有关于中断相关信号。于是放弃。过了一段时间组内成员又提出：如何处理中断信号应该是由机器自己来处理的，我们不需要去处理。该问题得以解决。

接下来是关于断点的保存，可以用存储器的指定位置担当IAR，但问题是：我们如何获取PC的值，因为从PC到IAR这段数据通路我们无法进行处理，问题就卡在了这，于是我们组就没有将中断继续下去。

# 日志

## 2021、7、2

记录人：左莎莎

选择了实验桌，并对TEC-8实验台进行了测试，能够成功下载程序到芯片。

同时，我们进行了计算机组成原理的第五次实验，测试成功。

至此，实验台测试成功。

同时，进一步熟悉了quartusII软件。

我们组于下午进行了讨论，对于整个题目不懂的部分进行了商讨。我们讨论了我们所要进行的任务是对于硬布线控制器的输入进行处理，使其能够输出正确的信号，同时，进行一定的创新：修改PC指针，两条流水设计。

并将任务划分成了几个模块：

* 非流水版本
* 流水版本
* 扩展指令、修改PC指针
* 测试程序设计和调试

确定了编程实现方式：时序逻辑。

**选择时序逻辑的原因：**

关于编程的逻辑，有两种方式。一种是类似于时序逻辑的思想,按照流程图一步步下来，通过case 来判别IR高四位的情况来，并且结合W1 W2 W3按顺序写。另外一种就是组合逻辑的方法，搞清楚信号比如LDZ等等在什么情况下要用。然后用LDC = X1 AND X2 AND ......的方法来实现。

虽然第二种组合逻辑的方法代码简短，并且ADN OR很符合数字逻辑的味道。但是这样不利于分辨每条指令的节拍周期，并且很难看出是如何流水的，以及是处在哪个模式下（ST0=0 ST0=1）。第一种方法可能代码较长，但是通过if 语句很容易就辨析出什么时候在ST0=0的置数模式，什么时候在ST0=1的死循环AR或者PC一直加1的模式。根据W1 W2 W3的赋值情况，也可以看出流水到底节省了哪个节拍。

而且时序方式容易纠错。

## 2021、7、3

记录人：杨元睿

今天我们用一些简单的逻辑程序去测试，来探索这个硬布线逻辑到底是什么东西。我们初步分析了一下，首先W1 W2 W3就是指令的节拍。并且它们应该是周期性产生的。比如先W1=1 再W2=1 最后 W3=1，如此交替反复。可是另我们费解的是，好像我们的设备不能产生准确的节拍。我们甚至用最暴力的写法。if(w1='1'),if(w2='1'),if(w3='1')这种语句来对某个管脚强行赋值。发现仍然是不行的。这就让我们心中有大大的问号。最终我们到了其他教室，观察对比了其他箱子。我们得出结论：我们的箱子是坏的。其他的箱子在你按下QD之后，左下角会先W1亮，然后W2亮。而我们的按下QD之后左下角什么灯都不会亮。我们马上就换了一个箱子来做实验。

可以问题又出现了。我们发现按了QD之后W3是一直不会亮的。我们又产生了怀疑，是不是拿到的又是坏的箱子。接着我们仔细看了实验指导书，我们发现，默认情况下都是两拍。也就是说只有W1和W2的。如果要有三拍的话，必须在W2的时候给long信号赋值。这样后面才会产生W3。这个发现让我们可以基本搞清楚这个时序到底是怎么回事了。

我们还分辨了容易搞混的引脚和信号。比如S(3) S(2) S(1) S(0)。这四个是控制ALU的函数发生的。而SEL(3) SEL(2) SEL(1) SEL(0)，这个是用来做选择的。比如在读写寄存器的模式下。SEL(3) SEL(2) 可以 为 00 01 10 11这样就可以选择4个寄存器来来进行操作。所以S和SEL不是一个东西，不能省略其中一个了。IR也是容易混淆的，IR只有高四位是操作码，所以我们只需要根据它的高四位来判别

晚上7点到9点，我们预约了研讨间，探讨了ST0设计，就指令扩展问题和流水线版本的思想和实现问题进行了思考和讨论。

## 2021、7、4

记录人：王仲天

**早上：**

我们组完成了基本题目的代码，并对其设计了一段比较简短的程序进行测试：



我们进行了几组数据测试，分别实现了无跳转、JZ跳转、JC跳转，均测试成功。

经过实验我们发现jmp指令中的4为偏移值的最高位为符号位。

**中午:**

完成了流水代码设计，还未设计扩展指令。

**下午：**

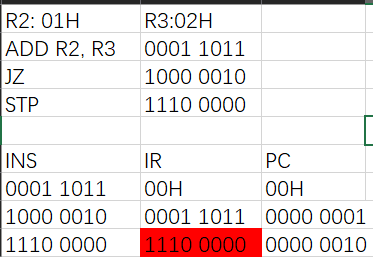
对流水代码进行了测试，我们测试了两个程序（一个很短的程序以及一个很长的程序），都没有问题，但是，当我们测试有关跳转指令的时候，发现了bug。

错误情况如下：

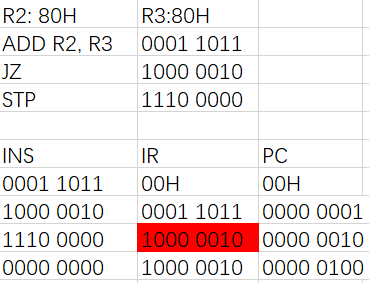
本来要执行jz的指令，此时的INS7-0也明明取出来jz的指令，可是，当我们再按一次QD,IR就显示为jz的下一条指令，此时，INS和PC正常显示。

于是，我们再针对这一问题特别使用了很短的测试程序进行测试：

这是z=0的情况：



这是z=1的情况(能够正常运行)：



我们先是设想：会不会流水就是这样，在不会跳转的时候就直接读取了下一条指令。但是，按照流水的工作原理，只是说取指令的时候会在前一指令执行时，但是，执行指令阶段不会出现被覆盖的情况（我们只有两条流水）。

于是我们又开始痛苦的肝代码过程，一直de到晚上九点，以失败告终，明天再接再厉。

## 2021、7、5

记录人：杨澜

我们就着昨天的问题做了一个尝试：把jmp,jz,jc往后延一个节拍，即在W2完成原来W1的工作，在W3完成原来W2的工作，结果测试发现，可以了！经探讨，我们把这个归结为资源冲突的原因：读后写或者时间太短，不够TEC-8正确完成操作。取完指令之后又立即让LIR信号有效，虽然理论上来说确实没什么影响，因为LIR有效之后是在T3上升沿才置入IR。但就是出现了这样的问题。

今天的OR扩展指令在测试的时候存在问题，精简了敏感信号表之后结果正确。

今天我们在测试ST指令的时候，发现结果不正确。于是，我们组又进入了漫长的debug过程，最后，我们发现ST指令写错了，Rd应该放在Rs前面，如下：



再次修改之后，结果正确。

## 2021、7、6

记录人：左莎莎

今天是实验的最后一天，明天就开始验收了。我们组回到实验室，将新编写的程序单脉冲和连续脉冲都跑了一遍，没有问题。

最后，我们进行了实验报告的整理，将实验过程中各个组员的理解、感悟和记录整理出来。

## 2021、7、7

今天，我们又去实验室测了一下数据，发现，还是会有吞指令的现象！

如下是我们组痛苦的debug过程：

* 1.JZ指令,当Z=0时,出现吞掉下一条指令的现象
* 2.原因:使用其他机器测试后,发现没有问题. 经检查是我们的机器问题
* 3.方案:将JZ的拍数延长,两个只有W1的指令相连,有近一半的概率使后面的被吞掉. 当JZ,当Z=0时,强行多出一拍后,可以防止机器内部的信号问题.
* 1.ST指令,发现PC有极小概率乱飞了
* 2.原因:使用其他机器测试后,发现没有问题,经检查是我们的机器问题
* 3.方案:对ST指令而言,第二拍MEMW为1时,在T2的上升沿写入存储器,猜测虽然此时存储器右端口是只读的,但是机器比较奇怪,此后,可能会影响LIR和PCINC在T3的执行.扩展一个节拍,使得ST指令的第二个节拍做MEMW置为1,而第三个节拍再取下一条指令
* 1.LD指令,发现PC有极小概率飞了
* 2.原因:使用其他机器测试后,发现没有问题,经检查是我们的机器问题
* 3.方案:对LD指令而言,第二拍MBUS为1时,把存储器里的内容送数据总线,DRW也在同时刻(T3上升沿)写数据到指定的寄存器
* 1.接连两条单拍指令会使得下一条指令有很大概率被吞掉
* 2.原因:使用其他机器测试后,发现没有问题,经检查是我们的机器问题
* 3.方案:对这种类型的指令,我们加入了NOP去隔断接连的两个单拍指令
* 1.经过上面的改动后,我们发现我们的流水线好像又和非流水线相差无几了,所以要想办法规避这些机器的问题
* 2.原因:代码设计问题,应当把LIR和PCINC作为公共部分提出来,因为在T3的上升沿,如果做的事情过多,可能就引发了不可预测的问题或未定义行为
* 3.方案:尝试使用一个变量st\_PC\_IR,这个变量可以控制所有指令的PCINC,LIR在合适的时机触发,利用一个新的进程,在T3上升沿将st\_PC\_IR修改为1,默认为0. 这样,在T3上升沿的时候,PCINC,LIR在合适的时候触发
* 也就是把取下一条指令,和PC自增的操作放在了另一个进程里.这样由于Qutartus设计出的电路图和之前不一样,也解决了之前的问题.

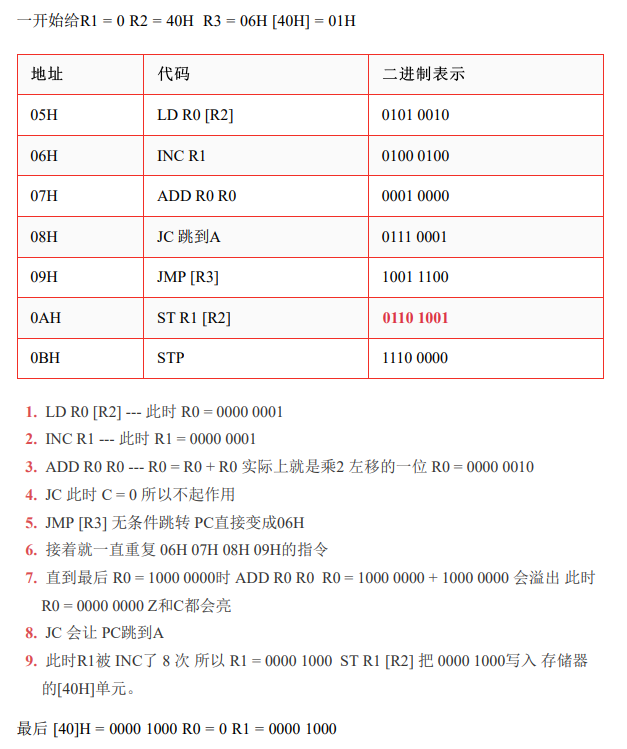
# 测试程序&记录

下载我们编译好的程序，以单拍方式（DP=1）执行编排好的验证程序，通过检查数据、指令和信号的指示灯来验证程序的正确性。

## 跳转测试



## 循环和任意位置PC测试



## 扩展指令测试

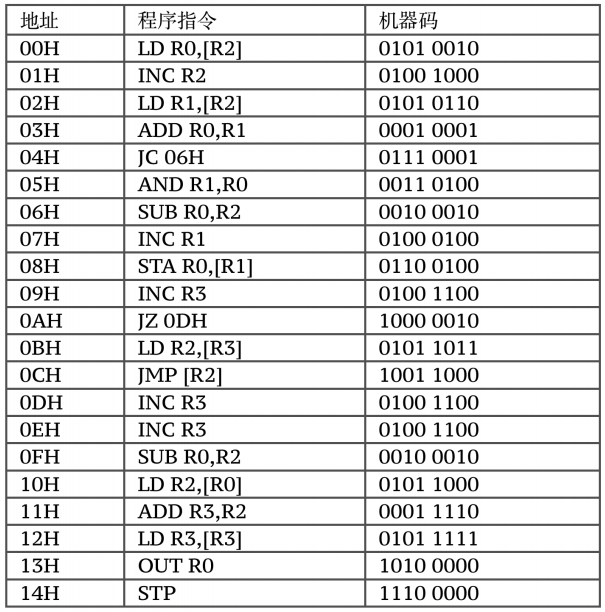
OR XOR



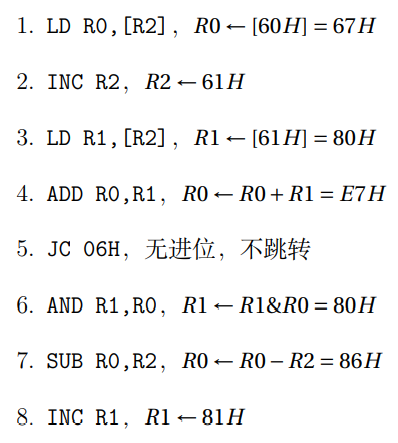


第三个扩展指令OUT见下面测试程序

## 老师指定程序测试

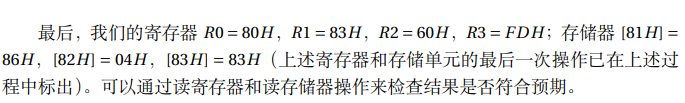






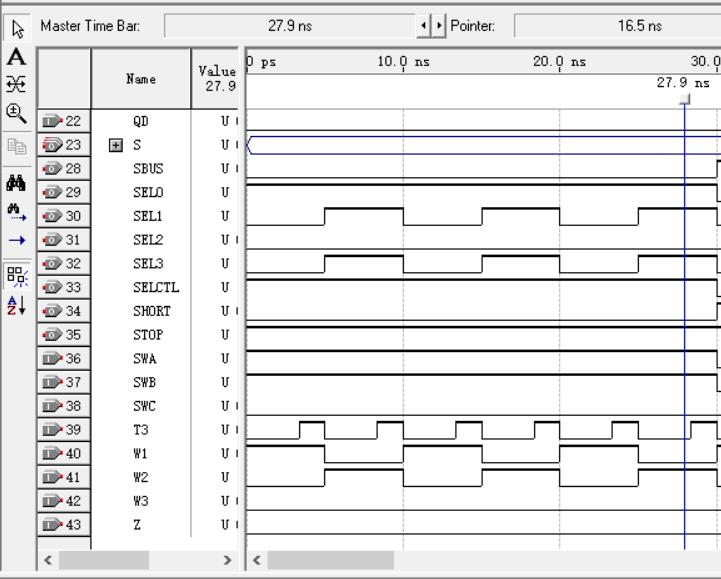


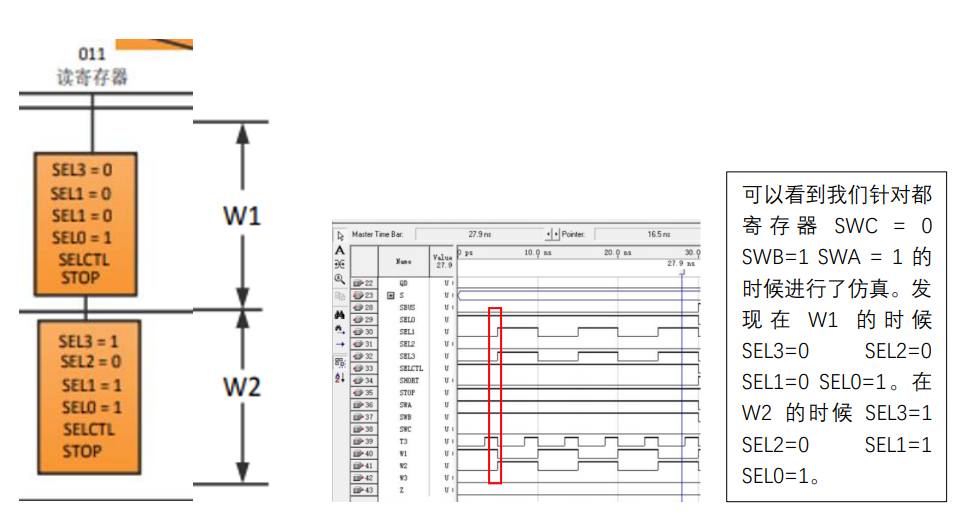




# 程序仿真

我们对程序进行了简单的仿真。如下：





# 遇到的问题及解决

我们记录了从理解题目，到设计程序、编码、测试和调试整个过程遇到的问题，经过探讨与实践，我们的问题得到了解决。

## TEC-8控制台的W1,W2,W3灯不亮

原因:机器问题,经查,发现其他机器都没有问题

方案:向教授说明情况后,更换机器

## W1、W2、W3的含义以及如何设计?

W1、W2、W3为硬连线控制器产生的节拍电位信号，T1、T2、T3为一组节拍脉冲。硬连线控制器以节拍电位(CPU 周期)为时间单位，所以，W1、W2、W3为机器周期, 是从节拍脉冲 T1 的上升沿到 T3 的下降沿的一段时间，一个执行框代表一个节拍电位时间。

TEC-8 模型计算机时序采用不定长机器周期，绝大多数指令采用 2 个机器周期 W1、W2，少数指令采用一个机器周期 W1 或者 3 个机器周期 W1、W2、W3。

**必要性：**

* 一般的指令只需要一个节拍电位。但是，对于ST和LD指令，需要去读写存储器，在一个节拍脉冲的时间里来不及完成将地址放入AR,同时从存储器中取出数据。所以，需要两个节拍电位来完成执行指令的操作。加上取指令一共是三个节拍电位。
* 又比如在流水硬连线控制器中的JC、JZ、Jmp,因为我们需要我们需要把 PC 更新之后再跳转到PC 指向的指令，这时也无法用一个节拍电位提供的三个节拍脉冲实现。

3 个电位节拍的指令，要求它在 W2 时产生一个信号 LONG，一个电位节拍的指令，要求它在W1时产生一个信号SHORT。四个电位节拍，则使用ST0实现，当 ST0=0 时，表示该控制台操作的第 1 个 W1、W2；当 ST0=1 时，表示该控制台操作的第 2 个 W1、W2。

## ST0应该在什么时候改变值？

应该在T3的下降沿，必须保证ST 0 的值在该节拍结束后及时改变，不能过早也不能过晚。否则，会进入到错误的机器周期里或者错误的分支里，导致程序发生错误。

## 进位标志C是如何变化的?

**现象**：负数与负数的相加显示溢出（即z亮）。

**解释：**74181 对减法运算采用的是补码运算方式，即先求得 [-减数] 的补码，然后和被减数的补码相加的方式完成。因此一个较大的数减去一个较小的数，或者 2 个相等的数相减时产生进位。

## 在流水硬布线中，有关nop指令的设置是怎么样的？

指令系统中，指令操作码 0000B 没有对应的指令，实际上指令操作码 0000B 对应着一条nop 指令，即什么也不做的指令。当复位信号为 0 时，对指令寄存器 IR 复位，使 IR 的值为00000000B，对应一条 nop 指令。这样设计的目的是适应指令流水的初始状态要求。

当执行nop指令时，就会执行pc+1,以便后续程序的执行。

## 参考流程图中，为什么ADD中没有LDC？

经过分析和讨论，我们还是按照自己的思考进行下去，将LDC加入流程图。

## 如何设计按下CLR之后IR里的内容置为00H?

这部分的内容不需要我们去设计代码，是直接TEC-8已经处理好了的。

## T3的上升沿和下降沿分别完成什么工作？

在调试过程中，我们最开始设计的代码是全部在T3下降沿进行工作，但是，有问题。于是，我们猜想会不会是在T3还分上升沿和下降沿进行不同的工作。

整理如下：

* 对于运算器操作来说，产生控制运算类型的信号 M、S3、S2、S1、S0 和 CIN；产生控制写入 Z 标志寄存器的信号 LDZ 和控制写入 C 标志寄存器的信号 LDC，产生将运算的数据结果送往数据总线 DBUS的控制信号 ABUS。这些控制信号保持到 T3 结束
* 在 T3 的上升沿，保存运算的数据结果到一个 8 位寄存器中，同时保存进位标志 C和结果为 0 标志 Z。
* 当 DRW 信号为 1 时，如果 LR0 为 1，则在 T3 的上升沿，将数据总线 DBUS 上的数写入 R0 寄存器，其余类推。
* 程序计数器 PC 由 2 片 GAL22V10(U53 和 U54)组成。向双端口 RAM 的右端口提供存储器地址。当复位信号 CLR#为 0 时，程序计数器复位，PC7~PC0 为 00H。当信号 LPC 为 1 时，**在T3 的上升沿，将数据总线 DBUS 上的数 D7~D0 写入 PC**。当信号 **PCINC 为 1 时，在 T3 的上升沿，完成 PC 加 1**。**当 PCADD 信号为 1 时，PC 和 IR 中的转移偏量(IR3~IR0)相加，在 T3 的上升沿，将相加得到的和写入 PC 程序计数器**。
* 指令寄存器 IR 是 1 片 74LS273(U47)，用于保存指令。**当信号 LIR 为 1 时，在 T3 的上升沿，将从双端口 RAM 右端口读出的指令 INS7~INS0 写入指令寄存器 IR。**
* STOP =1 时，在 T3 结束后时序发生器停止输出节拍脉冲 T1、T2、T3。

## PCINC和LAR必须指定在T3的上升沿进行吗？

经过研究，是TEC-8自己完成在T3上升沿的相关工作，我们只需要保持在此期间PCINC和LAR为1即可。

## SST0如何赋值和如何给ST0赋值？

在刚进入进程的时候，有SST0 <= 0,由于进程只会在结束时给各个信号赋值，且多次赋值只取最后一次赋值。所以，若在后面SST0又需要被赋值为1，则同样可以使用SST0 <= 0实现SST0的赋值。

这样，在上一节拍时SST0 = 1, 在再次进入进程时，就可以给ST0赋值。

* if (T3'event and T3 = '0') and SST0 = '1' then
* ST0 <= '1';

## 操作错误

桌面上有多个文件夹，当我们将已编译成功的程序下载到实验台上时，不是我们需要测试的程序，而是桌面上我们原来还有Bug的程序。这个错误也同样花费了我们组很多时间，导致之后我们尽量保持桌面的文件简洁。

# 设计调试小结

在设计部分，因为有了参考流程图，所以，设计并不是很难。因为我们采用的是时序编码方式，相对于组合逻辑的编码方式，需要更强的编程能力和逻辑性。

但是，测试与调试部分花了大量的时间。调试的时间比设计和编码的时间长，再加上硬件操作是我们组整体的一个弱项，调试这一部分还是比较艰难的。

TEC-8总是有各种各样莫名奇妙的错误，我们组一致觉得我们组能成功运行程序真的太不容易了。

# 各成员心得总结

左莎莎

在此次计算机组成原理课设中，经过我们全组成员的通力合作，我们顺利完成了硬布线控制器的设计，包括流水和非流水版本，涵盖了基本的读写寄存器和存储器操作，支持执行基本的指令集。从流程设计，代码编写，测试和调试，整个过程真的使我受益良多。

一开始的时候真的很懵，无从下手。但是后面经过和组员一起商讨，以及自己所学到的知识的理解，对实验任务和内容有了大体的认识。

我们de了很多bug, 其中，最痛苦的莫过于吞指令了，在反反复复地以为吞指令成功了！还没有！真的太痛苦了，到了最后的一天（验收前一天晚上），我们还没有解决。然后，我们全组熬夜，每人写了一版代码，第二天早早地测试程序，好不容易有一版没有吞指令了，却发现，无法STOP，后面再进入debug模式，总算成功了，喜极而泣！

经过五六天的辛苦付出，我们组的课设总算完成了，依靠着我们各个组员所学的知识，团队协作，真的太开心了！

杨澜

本次的实验是设计硬布线控制器的实验，利用已经给我们连好线的EPM7128芯片和对应的各种实验台资源，深入理解计算机最底层的工作原理，更深入地理解了机器代码是如何控制硬件工作的，通过本次实验，建立了整个代码逻辑和硬件细节的概念。

在本次实验中我进一步了解了cpu的运行原理，熟悉了vhdl语言的语法和编程以及tec-8实验系统的具体操作。编写代码的困难在于对于进程和时序逻辑的设计，课本和各类电子文档在这方面提供了很大帮助。测试中的难题主要在于给芯片设置管脚和输入指令的过程比较繁琐，以及在代码的测试中遇到了一些吞指令的问题，需要多次输入不同指令调试，而且有时候很难区分bug是出自代码还是试验台本身冲突的问题。这些问题给调试带来了困难，但是解决困难的过程增强了大家的协作能力和调试能力。我在这次实验中最终获得了很多的收获，感谢队友付出的努力。

杨元睿

实验开始前：我异常兴奋！我很想知道怎么给CPU硬布线，毕竟CPU这个词一直在我耳边出现过，但是我却没有真正地深入去了解它的工作原理。所以我和队友们讨论并且整理了大量数据逻辑和计算机组成原理的知识。尤其是上学期数字逻辑与数字系统的关于VHDL部分的PPT，都整理搜集完毕。

实验中：我异常崩溃！因为总是出现一些奇奇怪怪的错误。并且很难分清楚到底是机器的原因，还是自己程序的原因，还是测试指令的原因。最终我们发现三个原因都是存在的。由于我对箱子的操作还是比较了解的，所以程序队友交给我调试了。每一次置数，记录指令被吞掉的情况十分繁琐。但是最终大家还是一起解决了问题。

实验完成后：我收获满满！更加了解了硬布线和流水线的知识，以及如何用一些方法解决数据冲突！还有就是自己的思维层面获得了扩展，可以有一丝丝从高级语言->汇编->门电路的想法了

王仲天

我在做实验的时候,发现代码已经在实验指导手册上以流程图的形式给出了. 在这个条件下,我仔细的去检查了流程图上需要打开的每一个信号,又反复的去看了TEC-8的数据通路. 我发现之前的做的计组基础实验,还是很有用的,但因为我对前几个实验没有太上心.于是一开始不知道问题在什么地方,我反复的去看ALU,RAM和数据通路这三个基础实验. 最终可以对信号,数据通路,代码间建立了一些联系.

一开始不知道问题是怎么产生的,后来能对问题做一些猜测,虽然因为对硬件不了解,但也能通过微调做一些规避. 这个箱子对我来说就是一个黑盒,内部的信号问题,在外部编程的时候只能去规避,而解决不了.