# 北京邮电大学课程设计报告

<table>
<tr><td>课程设计名称</td><td colspan="2">计算机组成原理课程设计</td><td>学 院</td><td>计算机学院</td><td>指导教师</td><td>杨秦</td></tr>
<tr><td>班 级</td><td>班内序号</td><td colspan="2">学 号</td><td>学生姓名</td><td colspan="2">成绩</td></tr>
<tr><td>2023211301</td><td></td><td colspan="2">2023210882</td><td>王玉鑫</td><td colspan="2"></td></tr>
<tr><td>2023211301</td><td></td><td colspan="2">2023210892</td><td>穆文钦</td><td colspan="2"></td></tr>
<tr><td>2023211301</td><td></td><td colspan="2">2023210896</td><td>王书翰</td><td colspan="2"></td></tr>
<tr><td>2023211301</td><td></td><td colspan="2">2023211964</td><td>曹忠昊</td><td colspan="2"></td></tr>
<tr><td>课程设计内容</td><td colspan="6">教学目的：1、掌握计算机组成原理课程设计的基础知识及硬布线控制器的工作原理，分析典型的多输入和多输出复杂数字系统及逻辑关系，培养对复杂工程问题进行分解、细化以及设计的能力。<br>2、培养能够运用现代电子技术工具对设计工程进行功能仿真、测试，提高系统分析能力和解决问题的能力。<br>3、培养采用科学方法、正确的设计思路，掌握当今流行的体系结构，设计绿色、安全、应用性更强控制器的能力。<br>4、培养团队协作与沟通能力：通过分组合作完成设计任务，增强责任感和团队意识。<br>5、规范文档与成果展示：撰写实验报告并进行演示，提升工程文档撰写与沟通能力。<br>实验基本内容：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，用 Verilog 语言设计实现基于硬布线控制器的顺序模型处理器。在此基础上，设计并实现了流水式硬布线处理器。<br>实验方法：采用 Altera CPM7128 硬件，实现硬布线控制器的顺序模拟处理器；在 Quartus II 软件中用 verilog 代码来实现逻辑。<br>团队分工如下<br>王玉鑫：负责测试设备，基础功能代码的编写与维护，调试、测试代码，负责文档中流水线硬布线设计详解部分<br>穆文钦：负责测试设备，基础功能代码的编写，调试、测试代码，撰写文档<br>王书翰：负责测试设备，代码编写与维护，调试、测试代码；负责流水硬布线控制器的编写<br>曹忠昊：负责测试设备，代码维护以及运行测试程序，调试、测试代码；记录每天的日志以及文档中调试问题部分的编写</td></tr>
<tr><td>学生课程设计报告<br>(附页)</td><td colspan="6"></td></tr>
</table>

| | |
|---|---|
| 课程设计成绩评定 | 遵照实践教学大纲并根据以下四方面综合评定成绩：<br>1、课程设计目的任务明确，选题符合教学要求，份量及难易程度<br>2、团队分工是否恰当与合理<br>3、综合运用所学知识，提高分析问题、解决问题及实践动手能力的效果<br>4、是否认真、独立完成属于自己的课程设计内容，课程设计报告是否思路清晰、文字通顺、书写规范<br>评语：<br><br>成绩：<br>指导教师签名：<br>年　　月　　日 |

注：评语要体现每个学生的工作情况，可以加页。

北京郵電大學

# 计算机组成原理课程设计

学院：计算机学院（国家示范性软件学院）

专业：计算机科学与技术

班级：2023211301

小组：A 区第 24 组

成员：王书翰 2023210896

王玉鑫 2023210884

穆文钦 2023210892

曹忠昊 2023211964

2025 年 6 月 5 日

# 目录

# 1 环境分析

## 1.1 硬件环境

### 1.1.1 TEC-8 模型计算机时序信号

TEC-8 模型计算机主时钟 MF 频率为 1MHz，周期为 1μs，作为整个系统的基准时钟。执行一条微指令需要 3 个节拍脉冲 T1、T2、T3。TEC-8 模型计算机的时序采用不定长的机器周期，大部分微指令包含 2 个机器指令 W1 和 W2，少数微指令包含 1 个机器指令 W1，或者 3 个机器指令 W1、W2、W3。

单微指令开关 DP 控制节拍脉冲信号 T1、T2、T3 的数目。当 DP 朝上时，处于单微指令运行方式，每按一次 QD 按钮，只产生一组 T1、T2、T3；当 DP 朝下时，处于连续运行方式，每按一次 QD 按钮，开始连续产生 T1、T2、T3；直到按一次 CLR 按钮或者控制器产生 STOP 信号为止。

图 1.1 为 3 个机器周期的时序图

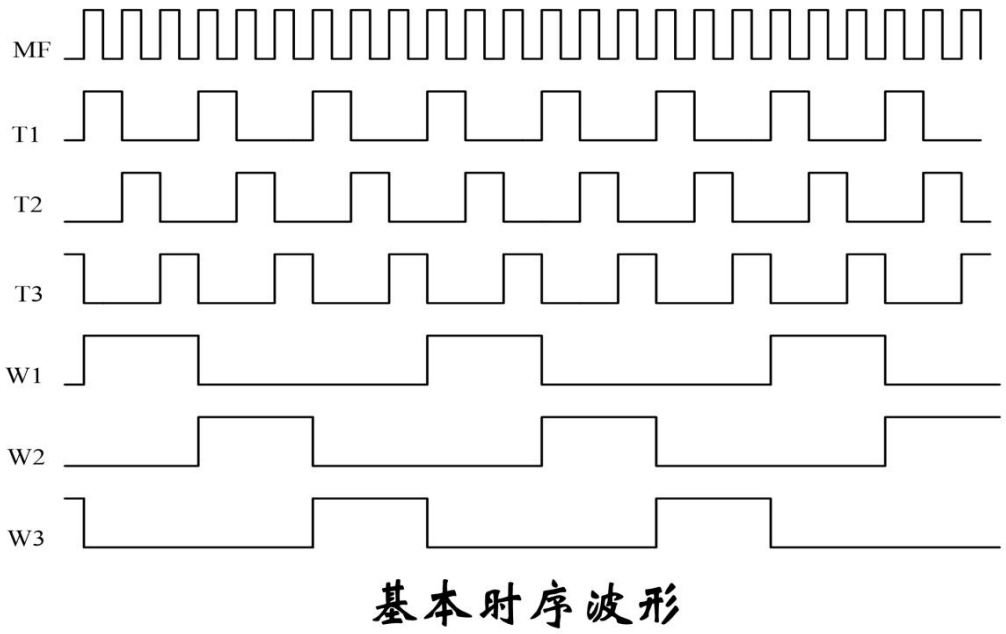


图 1.1

### 1.1.2 TEC-8 模型机系统框图

TEC-8 模型计算机主要由以下几个主要组件组成：

（1）时序发生器：产生节拍脉冲 T1、T2、T3，节拍电位 W1、W2、W3,以及中断请求信号 ITNQ。

（2）算术逻辑单元 ALU：算术逻辑单元由 2 片 74181 加 1 片 7474、一片 74244、一片 74245、一片 7430 组成，进行算术逻辑运算。

（3）双端口寄存器组：双端口寄存器由 1 片可编程器件 EPM7064 组成，向 ALU 提供两个运算操作数 A 和 B，保存运算结果。

（4）数据开关 SD7~SD0：8 位双位开关，朝上位置表示“1”、朝下位置表示“0”。

（5）双端口 RAM：这是一种两个端口可同时进行读、写的存储器，两个端口各有独立的存储器地址、数据总线和读、写控制信号。

（6）程序计数器 PC、地址寄存器 AR 和中断地址寄存器 IAR：程序计数器 PC 具有 PC 复位功能，从数据总线 DBUS 上装入初始 PC 功能，PC+1 功能；地址寄存器向双端口 RAM 的左端口提供存储器地址 AR7~AR0.它具有从数据总线 DBUS 上装入初始 AR 功能和 AR 加 1 功能；中断地址寄存器 IAR：作用是保存中断时的程序地址 PC。

（7）指令寄存器 IR：用于保存从双端口 RAM 中读出的指令。

（8）微程序控制器：微程序控制器产生 TEC-8 模型计算机所需的各种控制信号。

（9）硬连线控制器：硬连线控制器有 1 片可编程器件 EPM7128 组成，产生 TEC-8 模型计算机所需的各种控制信号。

（10）控制信号切换电路：决定模型计算机是使用微程序控制器产生的控制信号还是硬连线控制器产生的控制信号。

（11）2 选 1 选择器：用于在指令中的操作数 IR3~IR0 和控制信号 SEL3~SEL0 之间进行选择，产生目的寄存器编码 RD1、RD0，产生原码寄存器编码 RS1、RS0。

TEC-8 模型机系统框图如下：

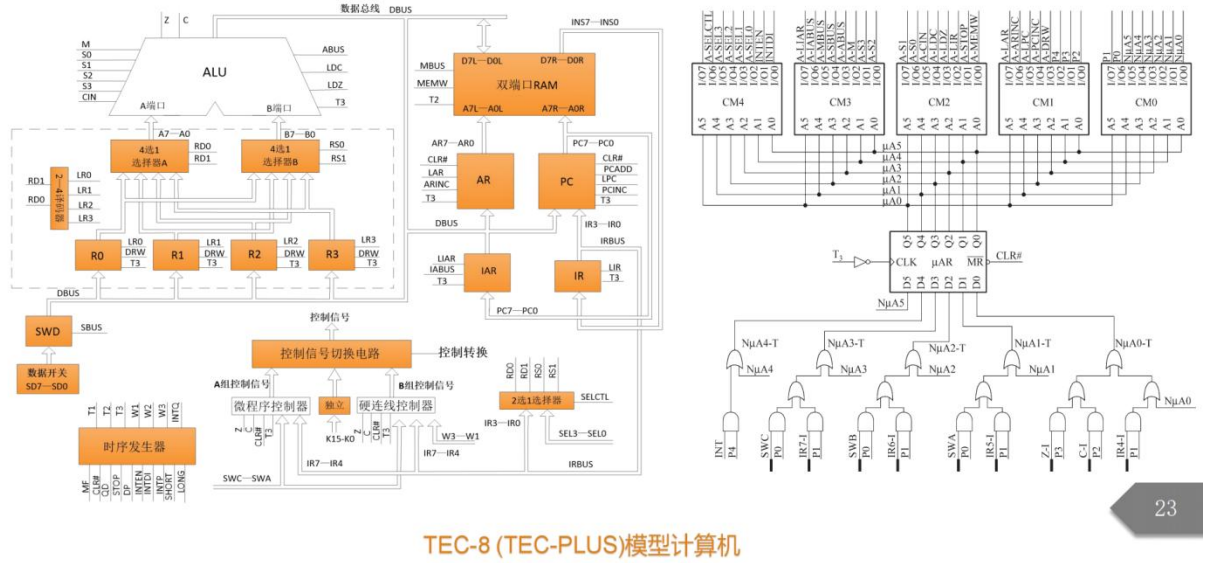


图 1.2

## 1.1.3 TEC-8 硬布线控制器流程

硬布线控制器的状态共有 5 种：写寄存器、读寄存器、读存储器、写存储器、取指。各个状态所对应的参数的数值以及每个指令的二进制表示可见图 1.3

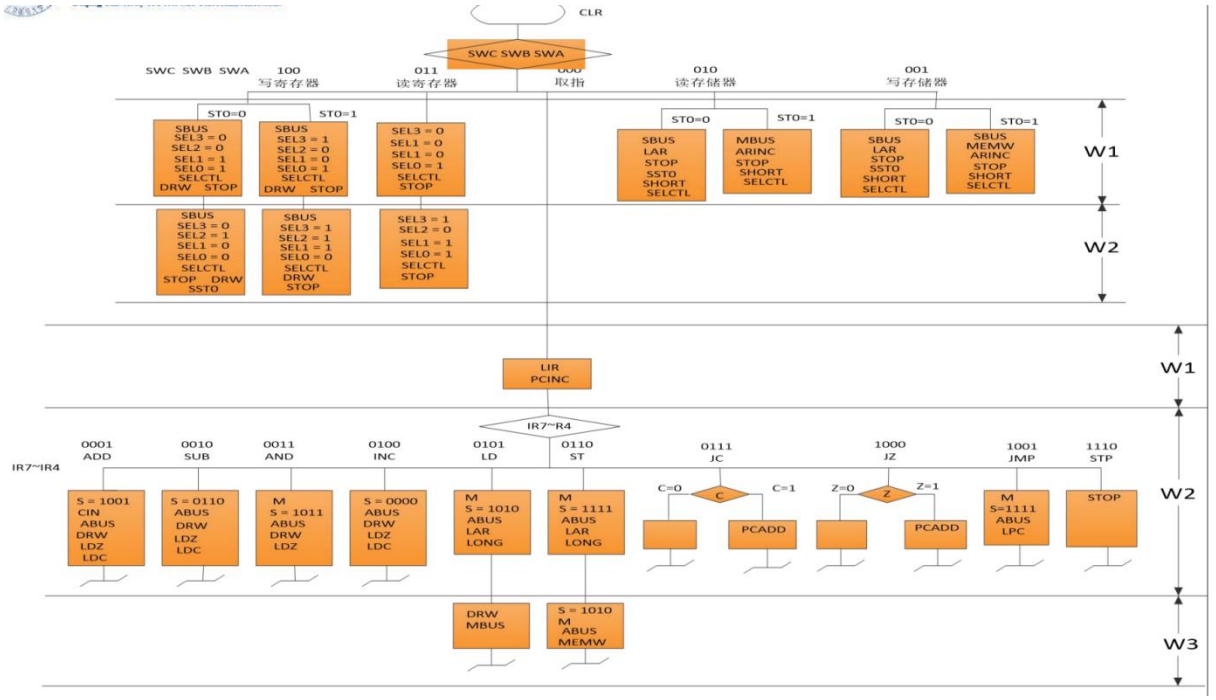


图 1.3

### 1.1.4 EPM7128 芯片

TEC-8 计算机硬件综合实验系统中的硬连线控制器由 1 片 EPM7128 芯片构成。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规定。硬连线控制器 EPM7128 引脚的规定如下图所示：

TEC-8平台 EPM7128 硬布线控制器部分引脚

| 信号 | 方向 | 引脚号 | 信号 | 方向 | 引脚号 |
|---|---|---|---|---|---|
| CLR# | 输入 | 1 | MEMW | 输出 | 27 |
| T3 | 输入 | 83 | STOP | 输出 | 28 |
| SWA | 输入 | 4 | LIR | 输出 | 29 |
| SWB | 输入 | 5 | LDZ | 输出 | 30 |
| SWC | 输入 | 6 | LDC | 输出 | 31 |
| IR4 | 输入 | 8 | CIN | 输出 | 33 |
| IR5 | 输入 | 9 | S0 | 输出 | 34 |
| IR6 | 输入 | 10 | S1 | 输出 | 35 |
| IR7 | 输入 | 11 | S2 | 输出 | 36 |
| W1 | 输入 | 12 | S3 | 输出 | 37 |
| W2 | 输入 | 15 | M | 输出 | 39 |
| W3 | 输入 | 16 | ABUS | 输出 | 40 |
| C | 输入 | 2 | SBUS | 输出 | 41 |
| Z | 输入 | 84 | MBUS | 输出 | 44 |
| DRW | 输出 | 20 | SHORT | 输出 | 45 |
| PCINC | 输出 | 21 | LONG | 输出 | 46 |
| LPC | 输出 | 22 | SEL0 | 输出 | 48 |
| LAR | 输出 | 25 | SEL1 | 输出 | 49 |
| PCADD | 输出 | 18 | SEL2 | 输出 | 50 |
| ARINC | 输出 | 24 | SEL3 | 输出 | 51 |
| SELCTL | 输出 | 52 | | | |

图 1.4

## 1.2 软件环境

本次实验采用 Altera Quartus II 9.1 作为主要开发环境，采用 Verilog HDL 代码，实现了包括硬布线控制器核心模块的设计。

# 2 实验原理

## 2.1 硬布线控制器

硬布线控制器是早期设计计算机的一种方法。硬布线控制器是将控制部件做成产生专门固定时序控制信号的逻辑电路，产生各种控制信号，因而又称为组合逻辑控制器。这种

逻辑电路以使用最少元件和取得最高操作速度为设计目标，因为该逻辑电路由门电路和触发器构成的复杂树形网络，所以称为硬布线控制器。

硬布线控制器主要由组合逻辑网络、指令寄存器和指令译码器、节拍电位/节拍脉冲发生器等部分组成，硬布线控制器的结构方框图如图 2.1 所示。其中组合逻辑网络产生计算机所需的全部操作命令，是控制器的核心。

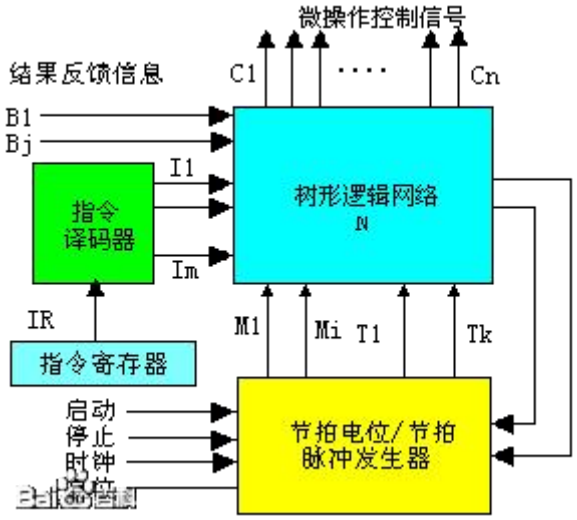


图 2.1

信号输入：

(1)来自指令操作码译码器的输出 I1～Im ,译码器每根输出线表示一条指令，译码器的输出反映出当前正在执行的指令；

(2)来自执行部件的反馈信息 B1～Bj ；

(3)来自时序产生器的时序信号，包括节拍电位信号 M1～Mi 和节拍脉冲信号 T1～Tk。

组合逻辑网络 N 的输出信号就是微操作控制信号 C1～Cn，用来对执行部件进行控制。另有一些信号则根据条件变量来改变时序发生器的计数顺序，以便跳过某些状态，从而可以缩短指令周期。

硬布线控制器的基本原理，归纳起来可叙述为：某一微操作控制信号 C 是指令操作码译码器输出 Im 、时序信号(节拍电位 Mi ，节拍脉冲 Tk )和状态条件信号 Bj 的逻辑函数，其数学描述为：

$$C = f(I_M, M_i, T_k, B_j)$$

本实验系统的控制器结构图如下：

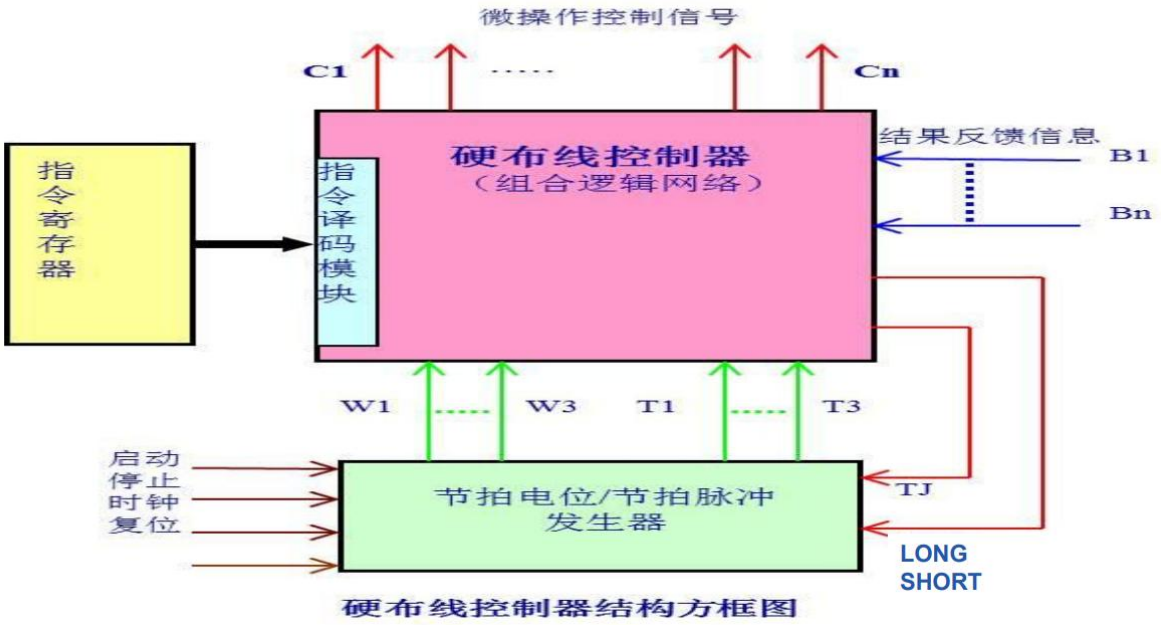


图 2.2

$T_k$拍脉冲信号直接传输给控制器，即模型计算机框图中的 T1、T2、T3。

由于本系统机器指令系统较简单，省去了指令译码阶段，4 位指令操作码 IR4~IR7 直接作为$I_m$的一部分。

$M_i$ 是时序发生器的节拍信号 W1~W3，表示当前所处的微操作节拍阶段

$B_i$ 是从数据通路传入到控制器中的，其来自 ALU、寄存器或存储器的状态位（如进位、零标志等等）

$C_i$ 是控制器向数据通路发出的微操作控制：硬布线逻辑根据“指令译码信号”+“W/T 时序”+“B 状态”产生的输出，用于读、写寄存器/存储器，总线控制等等。

## 2.2 EPM7128 器件引脚

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下图中的规定

| 信号 | 方向 | 引脚号 | 信号 | 方向 | 引脚号 |
|---|---|---|---|---|---|
| CLR# | 输入 | 1 | MEMW | 输出 | 27 |
| T3 | 输入 | 83 | STOP | 输出 | 28 |
| SWA | 输入 | 4 | LIR | 输出 | 29 |
| SWB | 输入 | 5 | LDZ | 输出 | 30 |
| SWC | 输入 | 6 | LDC | 输出 | 31 |
| IR4 | 输入 | 8 | CIN | 输出 | 33 |
| IR5 | 输入 | 9 | S0 | 输出 | 34 |
| IR6 | 输入 | 10 | S1 | 输出 | 35 |
| IR7 | 输入 | 11 | S2 | 输出 | 36 |
| W1 | 输入 | 12 | S3 | 输出 | 37 |
| W2 | 输入 | 15 | M | 输出 | 39 |
| W3 | 输入 | 16 | ABUS | 输出 | 40 |
| C | 输入 | 2 | SBUS | 输出 | 41 |
| Z | 输入 | 84 | MBUS | 输出 | 44 |
| DRW | 输出 | 20 | SHORT | 输出 | 45 |
| PCINC | 输出 | 21 | LONG | 输出 | 46 |
| LPC | 输出 | 22 | SEL0 | 输出 | 48 |
| LAR | 输出 | 25 | SEL1 | 输出 | 49 |
| PCADD | 输出 | 18 | SEL2 | 输出 | 50 |
| ARINC | 输出 | 24 | SEL3 | 输出 | 51 |
| SELCTL | 输出 | 52 | | | |

图 2.3

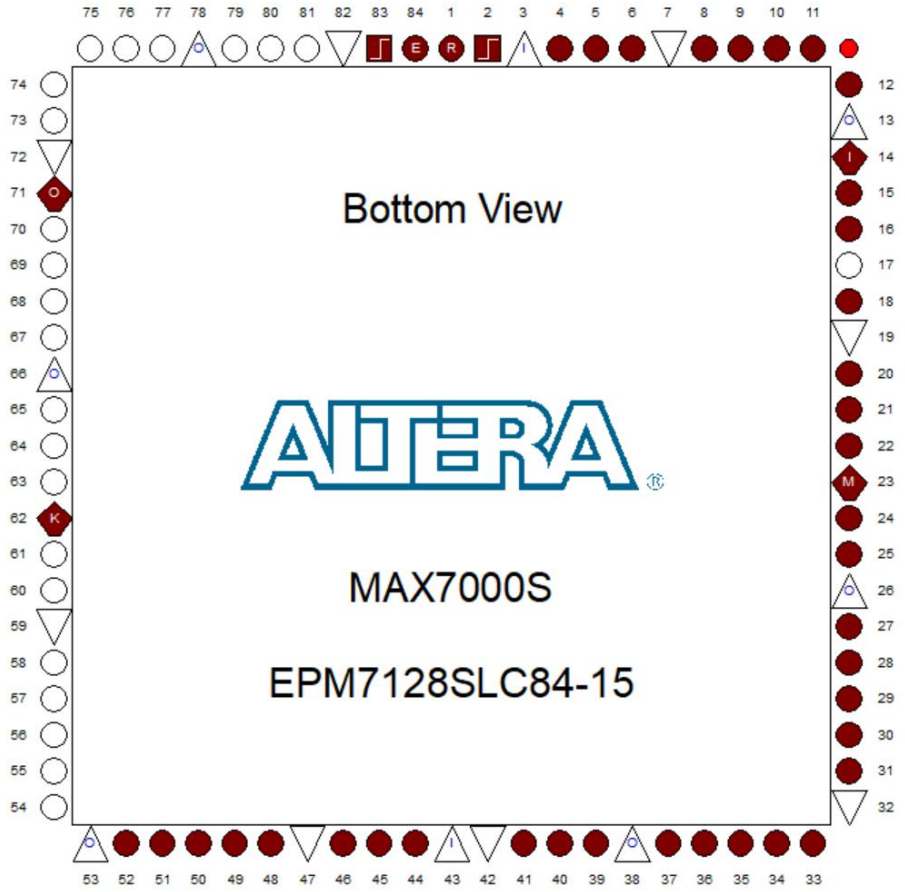


图 2.4

## 2.3 控制信号详解

| 信号 | 说明 |
| --- | --- |
| CLR# | 复位。 |
| T3 | 节拍脉冲信号 |
| SWC、SWB、SWA | 操作模式选择。 |
| IR7~IR4 | 指令寄存器的高四位。 |
| W3~W1 | 节拍电位信号。 |
| C | 进位标志。 |
| Z | 结果为 0 标志。 |
| DRW | =1 时，在 T3 上升沿对 RD1、RD0 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 | =1 时，在 T3 的上升沿 PC 加 1。 |
| LPC | =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 | =1 时，在 T3 的上升沿，将数据总线 DBUS 上的 D7~D0 写入地址寄存器 AR。 |
| PCADD | =1 时，将当前的 PC 值加上相对转移量，生成新的 PC。 |
| ARINC | =1 时，在 T3 的上升沿，AR 加 1。 |
| SETCTL | =1 时，实验系统处于实验台状态。=0 时，实验系统处于运行程序状态。 |
| MEMW | =1 时，在 T2 为 1 期间将数据总线 DBUS 上的 D7~D0 写入双端口 RAM。写入的存储器单元由 AR7~AR0 指定。 |
| STOP | =1 时，在 T3 结束后时序发生器停止输出节拍脉冲 T1、T2、T3。 |
| LIR | =1 时，在 T3 的上升沿将从双端口 RAM 的右端口读出的指令 INS7~INS0 写入指令寄存器 IR。读出的存储器单元由 PC7~PC0 指定。 |
| LDZ | =1 时，如果运算结果为 0，在 T3 的上升沿，将 1 写入到 Z 标志寄存器；如果运算结果不为 0，将 0 保存到 Z 标志寄存器。 |
| LDC | =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 | 低位 74LS181 的进位输入。 |
| M | 运算模式：M=0 为算术运算；M=1 逻辑运算。 |
| S3~S0 | 控制 74LS181 的运算类型。 |
| ABUS | =1 时，将运算结果送数据总线 DBUS。 |
| SBUS | =1 时，数据开关 SD7~SD0 的数送数据总线 DBUS。 |
| MBUS | =1 时，将双端口 RAM 的左端口数据送到数据总线 DBUS。 |
| SHORT | =1 时，指示时序发生器只产生一个节拍电位。 |
| LONG | =1 时，指示时序发生器产生三个节拍电位。 |
| SEL3~SEL2(RD1~RD0) | 选择送 ALU 的 A 端口的寄存器 |
| SEL1~SEL0(RS1~RS0) | 选择送 ALU 的 B 端口的寄存器 |

# 3 功能描述以及需求分析与设计

## 3.1 基本功能

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，以 Verilog 设计实现基于硬布线控制器的顺序模型处理器。根据设计方案，在实验平台上进行组装、调试并运行成功。

## 3.2 附加以及拓展功能

附加功能：（1）在原指令基础上要求扩指功能至少 3 条
（2）修改 PC 指针功能（指针任意执行功能）
拓展功能：流水线硬布线控制器设计

## 3.3 需求分析与设计

所编写的程序需要实现以下功能：顺序模型处理器、读存储器、写存储器、读寄存器、写寄存器、取指以及新增指令、修改 PC 指针、时序信号发生器、流水线设计

### 3.3.1 顺序模型处理器

顺序模型处理器是一种基础的计算机处理器架构，其特点是严格按照指令的存储顺序依次执行，不包含流水线或并行处理技术。处理器需要从指令寄存器中获取指令，根据指令的操作码确定执行的具体操作并生成相应的控制信号来完成指令的执行。还需要根据程序计数器（PC）的顺序递增实现指令的连续获取。处理器还需要负责数据存储器的读写操作，支持加载（LOAD）和存储（STORE）指令，以及将运算结果或从存储器读取的数据写回目标存储器。此外，处理器还要能够管理时序信号，以确保每个指令在适当的时钟周期内完成。

### 3.3.2 写寄存器

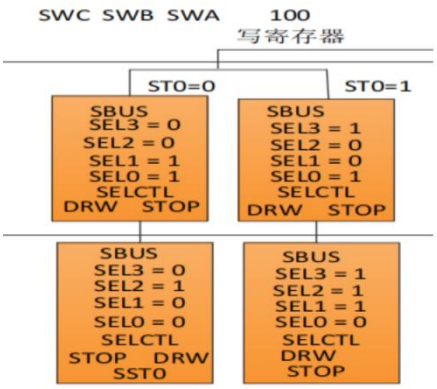


图 3.1

在 SWC_SWB_SWA 设置为 100 时，系统进入寄存器写入模式。该模式需要两个机器周期来完成对 R0 至 R3 四个寄存器的依次写入操作。整个写入过程通过 ST0 状态标志划分为两个明确的阶段，

第一阶段（ST0=0）负责处理前两个寄存器的写入：第一个 W1 节拍完成 R0 寄存器的写入，随后的 W2 周期则处理 R1 寄存器的写入。

第二阶段（ST0=1）完成后两个寄存器的写入：第三个 W1 节拍执行 R2 寄存器写入，最后的 W2 节拍完成 R3 寄存器写入。

寄存器写入操作的核心控制信号保持稳定：每次写入时 SBUS 和 DRW 信号均置为有效状态。区分不同寄存器写入的关键在于 SEL[3:0]信号线的配置。其中，SEL[3:2]用于指定当前待写入的目标寄存器，而 SEL[1:0]则用于选择需要读取显示的上一个操作寄存器。

### 3.3.3 读寄存器

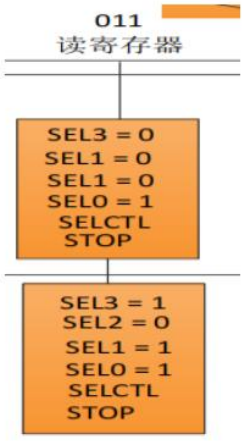


图 3.2

在 SWC_SWB_SWA 设置为 011 时，系统进入寄存器读取工作模式。通过 W1 和 W2 两个节拍分别完成不同寄存器组的读取。

W1 节拍主要负责 R0 和 R1 寄存器的读取工作。此时系统将 SEL3~SEL0 信号线配置为 0001，其中高位 SEL3~SEL2 设置为 00，用于选择 A 端口连接的 R0 寄存器；低位 SEL1~SEL0 设置为 01，用于选择 B 端口连接的 R1 寄存器。通过这种配置，ALU 的 A 端口指示灯（A7~A0）将显示 R0 寄存器的当前值，同时 B 端口指示灯（B7~B0）将显示 R1 寄存器的值。

W2 节拍则处理 R2 和 R3 寄存器的读取操作。此时系统会相应调整 SEL 信号线的配置，使 SEL3~SEL2 选择 R2 寄存器，SEL1~SEL0 选择 R3 寄存器。这种设计确保了在 W2 节拍时，ALU 的 A 端口指示灯显示 R2 寄存器的值，B 端口指示灯显示 R3 寄存器的值，实现了寄存器值的分批次读取和显示功能。

整个读取过程中，系统通过精确控制 SEL 信号线的状态，实现了四组寄存器值的分时读取和并行显示。两个节拍的时序设计既保证了所有寄存器都能被正确读取，又通过指示灯实现了寄存器值的可视化监控

### 3.3.4 写存储器

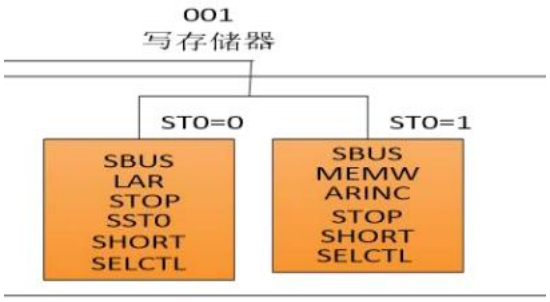


图 3.3

当 SWC SWB SWA=001 的时候，进入写存储器状态，根据标志位 ST0 的值将它分为两个阶段。第一个阶段指定首地址，第二个阶段不断循环，地址自加后依次写入存储单元。

第一个阶段时 ST0=0，SBUS=1，LAR=1，将数据开关的值送入地址寄存器 AR，指定了写存储器的首地址，然后将 SST0 置为 1，表示下一拍将进入 ST0=1 的阶段。第二个阶段时 ST0=1，MEMW=1，把当前数据总线上的值存入指定存储单元，然后 ARINC=1，将 AR 加 1，在下一个节拍中向下一个存储单元写入值，不断循环直到 CLR。

### 3.3.5 读存储器

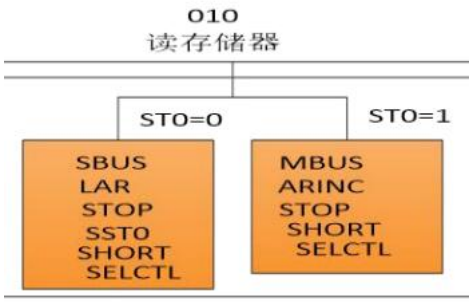


图 3.4

当 SWC SWB SWA=010 的时候，进入读存储器状态，与写存储器状态类似，根据标志位 ST0 的值将它分为两个阶段。第一个阶段指定首地址，第二个阶段不断循环，地址自加后依次从存储单元中读取数据。

第一个阶段时 ST0=0，SBUS=1，LAR=1，将数据开关的值送入地址寄存器 AR，指定了读存储器的首地址，然后将 SST0 置为 1，表示下一拍将进入 ST0=1 的阶段。第二个阶段时 ST0=1，MEMW=1，把当前指定存储单元的值读入到数据总线上，然后 ARINC=1，将 AR 加 1，在下一个节拍中向下一个存储单元写入值，不断循环直到 CLR。

### 3.3.6 取指以及新增指令

当 SWC SWB SWA=000 的时候，进入执行程序状态，第一个阶段取指令，第二个阶段进入执行指令阶段，之后在每条指令结束时再取指令，再执行，不断循环直到 STOP。

第一个阶段时，PCINC=1，LIR=1，取出当前的指令到 IR，PC 自加 1，指向下一条指令的地址，第二个阶段时根据 IR7~IR4 的值进入不同的分支执行指令，指令执行结束的时候再取下一条指令。

基础的指令为：

表 1 指令系统

| 名称 | 助记符 | 功能 | 指令格式 | | |
|---|---|---|---|---|---|
| | | | IR7~IR3 | IR3 IR2 | IR1 IR0 |
| 加法 | ADD Rd Rs | Rd ← Rd+ Rs | 0001 | Rd | Rs |
| 减法 | SUB Rd Rs | Rd ← Rd - Rs | 0010 | Rd | Rs |
| 逻辑与 | AND Rd, Rs | Rd ← Rd ∧ Rs | 0011 | Rd | Rs |
| 加 1 | INC Rd | Rd ← Rd + 1 | 0100 | Rd | XX |
| 取数 | LD Rd, [Rs] | Rd ← [Rs] | 0101 | Rd | Rs |
| 存数 | ST Rs, [Rd] | Rs → [Rd] | 0110 | Rd | Rs |
| C 条件转移 | JC addr | 如果 C=1，则 PC ← @+ offset | 0111 | offset | |
| Z 条件转移 | JZ addr | 如果 Z=1，则 PC ← @ + offset | 1000 | offset | |
| 无条件转移 | JMP [Rd] | PC ← Rd | 1001 | Rd | XX |
| 停机 | STOP | 暂停运行 | 1110 | XX | XX |

表中@表示当前 PC 的值，offset 是一个 4 位的补码有符号数，XX 表示随意值

表 2　扩展指令

| 名称 | 助记符 | 功能 | 指令格式 | | |
|---|---|---|---|---|---|
| | | | IR7~IR3 | IR3 IR2 | IR1 IR0 |
| 输出 | OUT[Rs] | DBUS ← Rs | 1010 | XX | Rs |
| 移动值 | MOV Rd, Rs | Rd ← Rs | 1011 | Rd | Rs |
| 比较 | CMP Rd, Rs | Rd - Rs | 1100 | Rd | Rs |
| 逻辑非 | NOT Rd | Rd ← ¬ Rd | 1101 | Rd | XX |
| 减 1 | DEC Rd | Rd← Rd-1 | 1111 | Rd | XX |

## 3.3.7 修改 PC 指针

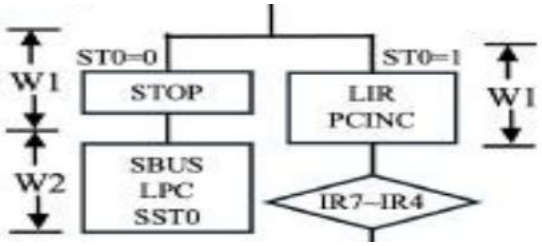


图 3.5

在执行指令的基础上扩展后可实现用户指定 PC 功能。指定 PC 的原理与写存储器的原理类似，即在程序开始执行前要将数据开关的值打到 PC 里，作为程序的首地址。通过标志位 ST0 将它分为两个阶段。第一个阶段指定程序存放的首地址，第二个阶段是取指令、执行指令。

第一个阶段 ST0=0，SBUS=1，LPC=1，将数据开关上的值存到 PC 里，实现了指定程序首地址，并且将 ST0 置为 1；第二阶段即正常的取指令阶段，ST0=1，PCINC=1，LIR=1，开始不断取指令并执行指令

## 3.3.8 时序信号发生器

### 3.3.8.1 节拍电位数

根据 TEC-8 硬件指导书我们可以得知：TEC-8 实验系统采用了可变节拍电位来执行一条机器指令。大部分的指令的执行需要两个节拍电位，例如：AND、SUB 等，少数指令需要三个电位节拍的指令，例如：LD、ST。

为了满足这种指令节拍数量变化的要求，我们采用了 SHORT 和 LONG 信号。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。对于没有产生 LONG 和 SHORT 信号的指令，我们就按照普通情况来处理：提供 2 个电位节拍。

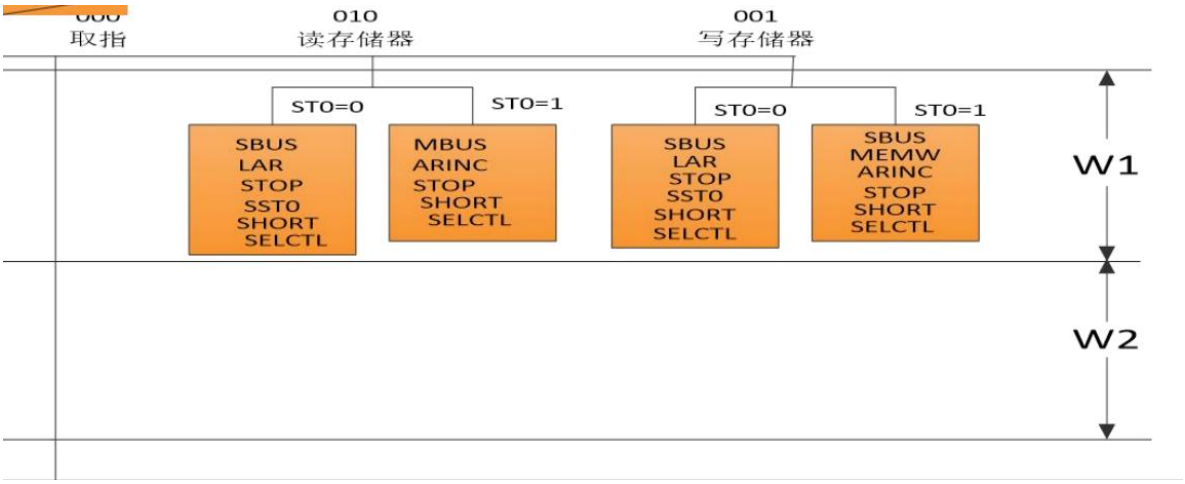


图 3.6

对于一些控制台操作，如读存储器、写存储器、取指操作直接结束的情况，只需要一个电位节拍 W1 就能够完成操作，所以这些操作可以在 W1 时产生一个 SHORT 信号，信号 SHORT 送往时序发生器，则时序信号发生器 W1 后不产生节拍电位 W2，下一个节拍仍然是 W1。读寄存器、写寄存器借用这个 SHORT 信号实现了先输入地址再输入/读取数据的功能。 信号 LONG 与 SHORT 只对紧跟其后的第一个节拍电位的产生起作用。

### 3.3.8.2 自定义节拍电位信号 W1、W2、W3

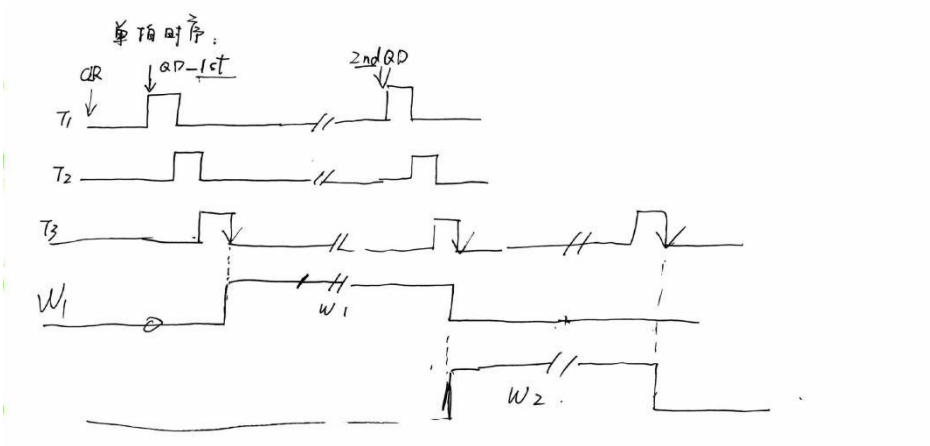


图 3.7

在实验过程中，我们发现在操作过程中会出现时延的问题即当按下 CLR 之后，第一次按 QD 没有任何输出信号，且显示为 W1 状态，当第一个 T3 处于下降沿的时候 W1 才第一次有效，第二次按下 QD 的时候还处于 W1 状态，等到第二个 T3 下降沿到来的时候才进入 W2。

为了避免时延问题的出现，我们选择舍弃了系统的 W1、W2、W3 输入，改为根据 T3 的下降沿以及不同指令的长度自己设定 W1、W2、W3 的输入。这样就能够保证按下 CLR 之后，第一次按 QD 能够正确执行指令的逻辑。根据 short_command 以及 long_command 两个标签来实现短指令以及长指令的节拍数量。在 W1 节拍下，如果 short_command 为 1，则下一节拍仍为 W1；为 0 则下一节拍为 W2。在 W2 节拍下，如果 long_command 为 1，则下一节拍为 W3；为 0 则下一节拍回到 W1。W3 节拍下，下一个节拍回到 W1。

### 3.3.8.3 ST0 以及 SST0 标志位

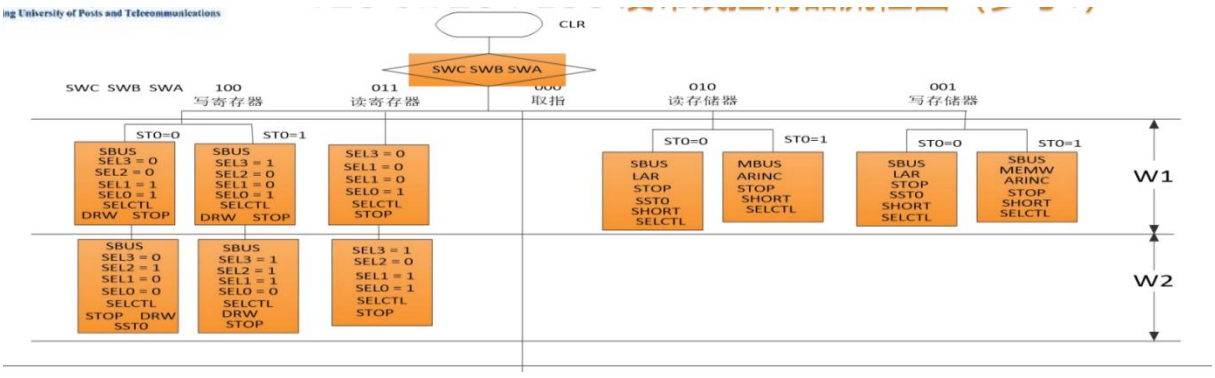


图 3.8

对于一些控制台操作，如写寄存器操作，需要 4 个节拍电位才能完成规定的功能，而一般情况最多能提供 3 个节拍。为了实现这个功能，我们将控制台操作化成两条机器指令的节拍。引入 ST0 标志，用来区分写寄存器的两个不同阶段，当 ST0 等于 0 时，写寄存器 R0 和 R1,当 ST1 等于 1 时，写寄存器 R2 和 R3。

而读、写存储器则是需要先写入地址，再读取/输入数据，所以也需要 STO 的参与，当 ST0 为 0 的时候，写入地址；当 ST0 为 1 的时候，读取/输入数据。

为了使 ST0 能够在 0、1 之间转换，我们采用了 SST0 来实现。SST0 置 1 的条件有以下四个：

当处于写寄存器模式且 ST0=0 且处于 W2 节拍时，将 SST0 置 1；

当处于写存储器模式且 ST0=0 且处于 W1 节拍时，将 SST0 置 1；

当处于读存储器模式且 ST0=0 且处于 W1 节拍时，将 SST0 置 1；

当处于取指模式且 ST0=0 且处于 W2 节拍时，将 SST0 置 1；

而 SST0 为 1 的时候将 ST0 置为 1；ST0 不会自动从 1 翻转回 0，除非被 CLR 复位；

从而实现了 ST0 在 0、1 之间的转换。

### 3.3.9 流水线功能设计

一个指令周期包含两个过程，即为取指令和执行指令过程。在非流水的模式下，上一条指令的执行指令过程结束后，才会开始执行下一条指令的取指令过程。而对于流水模式来说，上一条指令的执行指令过程可以和下一条指令的取指令过程在时间上进行重叠，即这两个过程同时进行如图 3.9 所示。

对此，在实现基础功能的基础上，我们要对输出的信号进行修改，在原来执行指令的过程中同时输出取指令的信号，实现上述执行指令过程和取指令过程同时进行的特性。

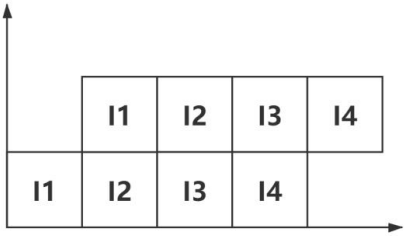


图 3.9

流水线流程图如下：

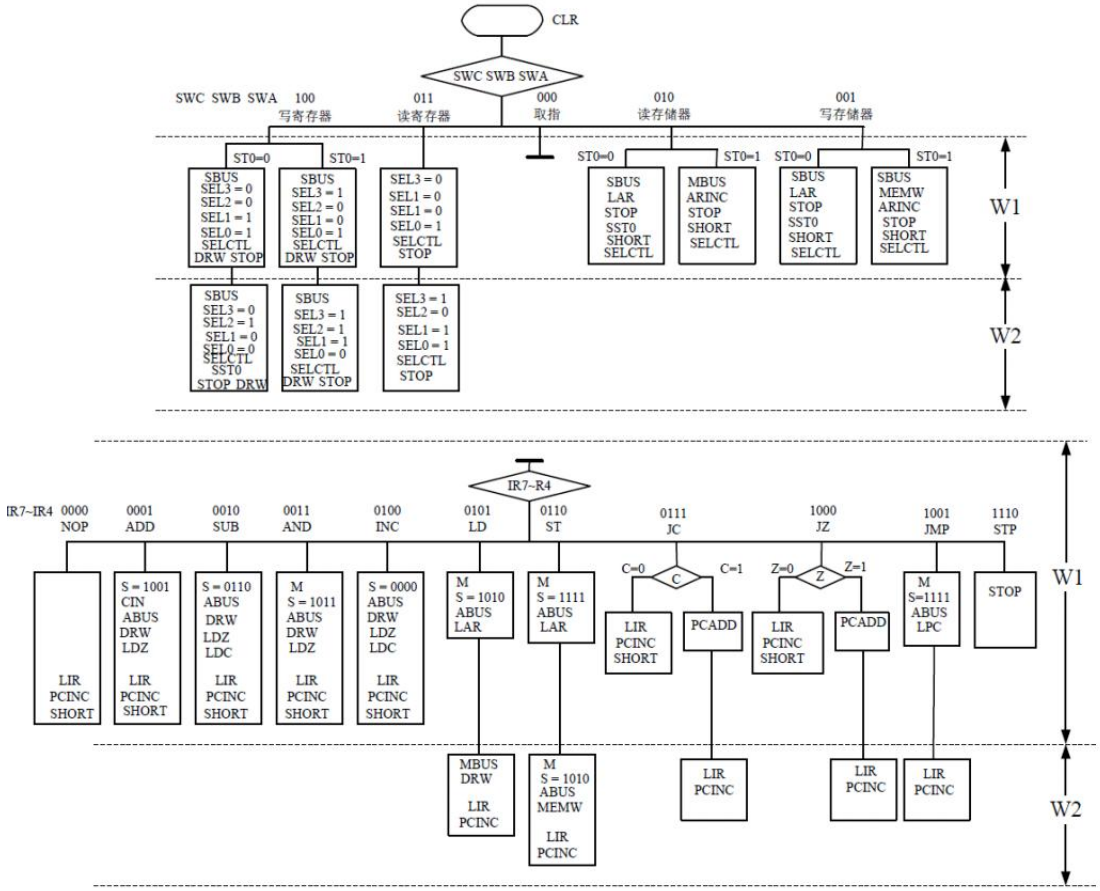


图 3.10

流水线修改 PC 指针过程：

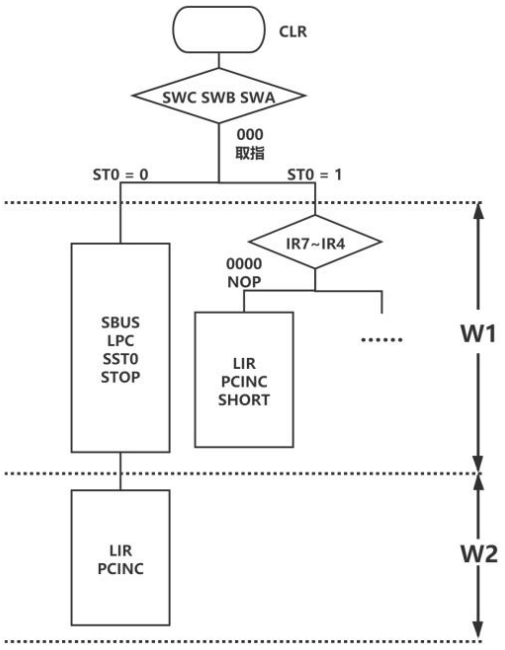


图 3.11

# 4 设计详解

## 4.1 基础功能以及附加功能设计

为了用时序逻辑来实现硬布线控制的设计，我们采用了状态机。定义五个状态：WRITE_REG、READ_REG、INS_FETCH、READ_MEM、WRITE_MEM，根据 SWCBA 的值来决定状态，同时通过 assign 操作来给指令赋值，1 表示要执行这条指令；0 为不执行。根据状态以及要执行的指令从而给最终的输出来赋值。

### 4.1.1 状态选择

这段代码通过一个在时序信号 T3 下降沿触发的组合逻辑，实现了对微程序模式寄存器 Q 的更新与译码，并由此派生出五种基本运行模式。首先，在每个时钟周期的 T3 下降沿，通过对三位模式选择信号 SWC_SWB_SWA 进行 case 判断，将 Q 设置为相应的三位二进制值（100、011、000、010、001），如果输入不在这五种情况之内，则赋予默认状态 111。这种方式保证了当外部通过开关或拨码选择不同操作模式时，内部状态机能够及时捕获并体现当前所需的微操作环境。

接下来，五组 assign 语句将寄存器 Q 与五种工作模式一一对应：当 Q==3'b100 时，工作模式为“写寄存器”；当 Q==3'b011 时，进入“读取寄存器”模式；Q==3'b000 对应“取指”模式；Q==3'b010 和 3'b001 分别对应“读取存储器”和“写入存储器”模式。

```
always @(negedge T3)
    begin
            case   (SWC_SWB_SWA)
                3'b100:   Q=100;
                3'b011:   Q=011;
                3'b000:   Q=000;
                3'b010:   Q=010;
                3'b001:   Q=001;
                default:
                          Q=111;
             endcase
    end
    assign WRITE_REG = (Q == 3'b100) ? 1: 0;//写寄存器模式
    assign READ_REG = (Q == 3'b011) ? 1: 0; //  读取寄存器模式
    assign INS_FETCH = (Q == 3'b000) ? 1: 0; //  指令取指模式
    assign READ_MEM = (Q == 3'b010) ? 1: 0; //  读取存储器模式
    assign WRITE_MEM = (Q == 3'b001) ? 1: 0; //  写入存储器模式
```

### 4.1.2 ST0 以及 SST0 的定义

这段代码定义了一个名为 ST0 的内部状态标志，并通过对复位信号 CLR 和时序信号 T3 的双重敏感触发来控制其置位和复位。当复位信号被拉低（CLR == 0）时，ST0 会立即被清零，从而确保在系统复位期间所有后续微操作都重新从初始状态开始；而在正常运行时，只要满足 SST0 条件且 ST0 目前为 0，就会在 T3 的下降沿将其置为 1，表示进入微指令的执行第一阶段。

SST0 本身则是一个布尔组合逻辑，它在 ST0 == 0 且对应模式的特定时序到来时为真。通过这种方式，巧妙地利用复位、状态判定和时序信号的协同作用，实现了对微指令阶段的精细划分与可靠控制。具体代码如下:

```
always @(negedge T3 or negedge CLR) begin
        if (CLR == 0 ) begin
            ST0 <= 1'b0;
        end
        else if (SST0) begin // st0_set_condition 在 ST0=0 且特定条件满足时为真
            ST0 <= 1'b1;
        end
        else if(ST0 && W2 && WRITE_REG)begin
    ST0 <=1'b0;
    end
    end
assign SST0 = (ST0 == 1'b0) && (
                    (SWC_SWB_SWA == 3'b100 && W2) || // 写寄存器模式，W2 有效
                    (SWC_SWB_SWA == 3'b010 && W1) || // 读存储器模式，W1 有效
                    (SWC_SWB_SWA == 3'b001 && W1) || // 写存储器模式，W1 有效
                    (SWC_SWB_SWA == 3'b000 && W2)
                );
```

### 4.1.3 指令译码

在这段代码中，我们将指令寄存器的高四位与微程序的取指阶段及内部状态标志相结合，采用一系列条件赋值语句对每条指令进行精确识别。具体而言，只有在处于“取指”模式（INS_FETCH == 1）且第一阶段微指令已经启动（ST0 == 1）的时刻，才会对 IR7_IR4 的各个二进制编码进行比对，从而产生对应的指令信号。比如，当高四位恰为 0001 时，ADD 信号被置为真；若是 0010，则为 SUB，以此类推。这种做法不仅涵盖了基本的

算术与逻辑运算（如加法、减法、与运算、自增等），也包括加载（LD）、存储（ST）、条件跳转（JC、JZ）、无条件跳转（JMP）、以及停止（STP）等控制指令。我们还在相同框架下扩展了新的操作，包括输出（OUT）、寄存器间数据搬移（MOV）、比较（CMP）、按位取反（NOT）以及自减（DEC）。通过将取指阶段、状态标志和指令编码三者紧密耦合，这套译码逻辑能够在时序信号到来时瞬时识别出当前要执行的微操作，并驱动后续的数据通路与控制信号准确地完成相应功能。

```
assign ADD = (IR7_IR4 == 4'b0001 && INS_FETCH == 1 && ST0 == 1) ? 1: 0;
assign SUB = (IR7_IR4 == 4'b0010 && INS_FETCH == 1 && ST0 == 1) ? 1: 0;
assign AND = (IR7_IR4 == 4'b0011 && INS_FETCH == 1 && ST0 == 1) ? 1: 0;
assign INC = (IR7_IR4 == 4'b0100 && INS_FETCH == 1 && ST0 == 1) ? 1: 0;
assign LD = (IR7_IR4 == 4'b0101 && INS_FETCH == 1 && ST0 == 1) ? 1: 0;
assign ST = (IR7_IR4 == 4'b0110 && INS_FETCH == 1 && ST0 == 1) ? 1: 0;
assign JC = (IR7_IR4 == 4'b0111 && INS_FETCH == 1 && ST0 == 1) ? 1: 0;
assign JZ = (IR7_IR4 == 4'b1000 && INS_FETCH == 1 && ST0 == 1) ? 1: 0;
assign JMP = (IR7_IR4 == 4'b1001 && INS_FETCH == 1 && ST0 == 1) ? 1: 0;
assign STP = (IR7_IR4 == 4'b1110 && INS_FETCH == 1 && ST0 == 1) ? 1: 0;

assign OUT = (IR7_IR4 == 4'b1010 && INS_FETCH == 1 && ST0 == 1) ? 1: 0;
assign MOV = (IR7_IR4 == 4'b1011 && INS_FETCH == 1 && ST0 == 1) ? 1: 0;
assign CMP = (IR7_IR4 == 4'b1100 && INS_FETCH == 1 && ST0 == 1) ? 1: 0;
assign NOT = (IR7_IR4 == 4'b1101 && INS_FETCH == 1 && ST0 == 1) ? 1: 0;
assign DEC = (IR7_IR4 == 4'b1111 && INS_FETCH == 1 && ST0 == 1) ? 1: 0;
```

### 4.1.4 控制信号生成

在这段代码中，SBUS、ABUS 和 MBUS 三条总线控制信号根据当前模式和时序信号的组合，灵活地将数据源传送到公共数据总线上。具体来说，当处于写寄存器模式（WRITE_REG）或存储器访问模式（READ_MEM/WRITE_MEM）且在第一个微时序（W1）时，SBUS 会拉高，将手动输入或者存储器输出驱入总线；在写寄存器的第二时序（W2）以及取指第二时序（INS_FETCH && !ST0 && W2）时也会激活，确保在不同阶段都能获得正确的数据输入。对应的，ABUS 在 ALU 运算阶段的第二时序（W2）或者存储操作的第三时序（W3）生效，将寄存器或存储器的数据提供给 ALU。

其余的各种控制信号都根据不同的状况来确定，这里不再阐述。

```
assign SBUS = ((WRITE_REG ||(READ_MEM && !ST0) || WRITE_MEM ) && W1) || (WRITE_REG && W2) ||((INS_FETCH && !ST0) && W2);
    assign SEL[3] = ((WRITE_REG && (W1 || W2)) && ST0) || (READ_REG && W2);
    assign SEL[2] = WRITE_REG && W2;
    assign SEL[1] = (WRITE_REG && ((W1 && !ST0) || (W2 && ST0))) || (READ_REG && W2);
    assign SEL[0] = (WRITE_REG && W1) || (READ_REG && (W1 || W2));
    assign SELCTL = ((WRITE_REG || READ_REG) && (W1 || W2)) || ((READ_MEM || WRITE_MEM) && W1);
    assign DRW = (WRITE_REG && (W1 || W2)) || ((ADD || SUB || AND || INC || NOT || MOV || DEC) && W2) || (LD && W3);
    assign STOP = ((WRITE_REG || READ_REG) && (W1 || W2)) || ((READ_MEM || WRITE_MEM) && W1) || (STP && W2) || (INS_FETCH && !ST0 && W1);
    assign LAR = ((READ_MEM || WRITE_MEM) && W1 && !ST0)||((ST || LD) && W2);
    assign SHORT = ((READ_MEM || WRITE_MEM) && W1);
    assign MBUS = (READ_MEM && W1 && ST0) || (LD && W3);
    assign ARINC = (WRITE_MEM || READ_MEM) && W1 && ST0;
    assign MEMW = (WRITE_MEM && W1 && ST0) || (ST && W3);
    assign PCINC = INS_FETCH && ST0 && W1;
    assign LIR = INS_FETCH && ST0 && W1;
    assign CIN = (ADD || DEC) && W2;
    assign ABUS = ((ADD || SUB || AND || INC || LD || ST || JMP || DEC) && W2) || (ST && W3) || ((MOV || OUT) && W2) || (CMP && W2) || (NOT && W2);
    assign LDZ = (ADD || SUB || AND || INC || CMP || DEC) && W2;
    assign LDC = (ADD || SUB || INC || CMP || NOT || DEC) && W2;
    assign M = ((AND || LD || ST || JMP) && W2) || (ST && W3) || ((NOT || MOV || OUT) && W2);
    assign S[3] = ((ADD || AND || LD || ST || JMP || OUT || MOV || DEC) && W2) || (ST && W3) ;
    assign S[2] = ((SUB || ST || JMP || CMP || DEC) && W2);
    assign S[1] = ((SUB || AND || LD || ST || JMP || OUT || CMP || MOV || DEC) && W2) || (ST && W3);
    assign S[0] = (ADD || AND || ST || JMP || DEC) && W2;
    assign LPC    = (JMP && W2) || (INS_FETCH && !ST0 && W2);
    assign LONG = (ST || LD) && W2;
    assign PCADD = ((C && JC) || (Z && JZ)) && W2;
```

## 4.2 流水式硬布线的设计

### 4.2.1 总体设计

总体可以分为以下八个部分：

1. 模块声明与接口定义
2. 内部信号声明
3. 指令长度判断逻辑
4. W 信号状态机
5. ST0 状态机
6. 操作模式译码
7. 指令译码
8. 控制信号的生成

在复位信号 CLR 的作用下，系统初始化，W 信号设置为初始状态（W[1] = 1），ST0 状态机复位为 0，所有控制信号也复位为 0，系统准备好接收新指令。

随后，系统进入指令取指阶段。通过 SWC_SWB_SWA 信号判断是否处于指令取指模式（INS_FETCH）。如果是，系统根据 IR7_IR4 信号译码当前指令类型（如加法 ADD、减法 SUB 等）。在 T3 信号的下降沿，生成相应的控制信号，例如 PCINC 使程序计数器加 1 以获取下一条指令，LIR 使指令寄存器从数据总线读取指令。

进入指令执行阶段，根据指令类型判断是短周期还是长周期指令。W 信号状态机根据指令长度动态调整状态：短周期指令保持在 W[1]状态，长周期指令依次经过 W[2]和 W[3]状态。ST0 状态机在指令执行时激活，执行完成后清零。控制信号根据指令类型和 W 信号状态动态生成，例如加法指令激活 ABUS 和 DRW 信号，存储器读取指令激活 MBUS 和 ARINC 信号，确保指令正确执行。

指令执行完成后，根据指令类型生成停止信号（STOP），更新状态标志（如零标志 Z 和进位标志 C），并根据需要更新程序计数器（PC）。ST0 和 W 信号复位，系统准备在下一个 T3 下降沿处理下一条指令。

### 4.2.2 模块声明与接口定义

这部分主要定义了模块的名称、输入端口和输出端口。其中，输入部分为实验台的输入信号，输出部分为控制信号。

```
module ver1 (
    // 输入端口
    input wire [2:0] SWC_SWB_SWA,        // 模式选择信号
    input wire [3:0] IR7_IR4,            // 指令寄存器高 4 位
    input wire CLR,                      // 复位信号 (低有效)
    input wire T3,                       // 时序信号 T3 (作为 ST0 的时钟)
    input wire W1, W2, W3,               // 微指令时序信号
    input wire C, Z,                     // 状态标志：进位 C，零 Z
    // 输出端口
    output reg SELCTL,                   // 选择控制信号
    output reg ABUS,                     // 控制 ALU 数据是否进入数据总线
    output reg SBUS,                     // 控制手动输入数据是否进入数据总线
    output reg MBUS,                     // 控制双端口存储器的数据送到数据总线
    output reg M, CIN,                   // M 是控制 ALU 是逻辑运算还是算数运算，CIN 用于区分操作是否有进位
    output reg DRW,                      // 控制寄存器写入信号
    output reg LDZ,                      // 若为 1，当运算结果为 0 时设 Z 为 1，若不为 0 则设 Z 为 0
    output reg LDC,                      // 若为 1，当运算结果产生进位时设 C 为 1，若无进位则设 C 为 0
    output reg MEMW,                     // 控制存储器双端口 RAM 写入信号
    output reg ARINC,                    // 控制地址寄存器 AR 的值加 1 信号
    output reg PCINC,                    // 控制地址寄存器 PC 的值加 1 信号
    output reg PCADD,                    // 控制 PC 加指令地址信号
    output reg LPC,                      // 控制将数据总线上数据写入 PC 寄存器
    output reg LAR,                      // 控制将数据总线上数据写入 AR 寄存器
    output reg LIR,                      // 控制将数据总线上数据写入 IR 寄存器
    output reg STOP,                     // 停止信号
    output reg SHORT,                    // 短延时信号
    output reg LONG,                     // 长延时信号
    output reg [3:0] S, SEL              // S 控制 ALU 产生不同的函数，SEL 用于 2-4 译码器
);
```

### 4.2.3 内部信号声明

这部分主要声明了模块内部使用的寄存器和中间信号。尤其定义了操作模式和指令（包括基础指令和附加指令）。

```
// 内部信号声明
reg ST0;
wire WRITE_REG, READ_REG, INS_FETCH, WRITE_MEM,
READ_MEM;
wire ADD, SUB, AND, INC, LD, ST, JC, JZ, JMP, STP;
wire NOP, OUT, NOT, CMP, MOV, DEC;
reg [3:1] W; // 用于存储 W 信号的状态
wire short_command;
```

### 4.2.4 指令长度判断逻辑

这部分逻辑用于判断指令是短周期指令还是长周期指令。

```
// 在解码阶段确定指令长度
assign short_command = (NOP || ADD || SUB || AND || INC || (JC && !C) ||
                        (JZ && !Z) || OUT || NOT || CMP || MOV || DEC ||
                        READ_MEM || WRITE_MEM);

assign long_command = 1'b0;
```

### 4.2.5 W 信号状态机

这部分是一个状态机，用于控制指令执行的不同阶段。

W 信号状态机根据指令长度动态调整状态：短周期指令保持在 W[1]状态，长周期指令依次经过 W[2]和 W[3]状态。

```
// T3 下降沿计数器逻辑
always @(negedge T3 or negedge CLR) begin
    if (!CLR) begin
        W[1] <= 1'b1;
        W[2] <= 1'b0;
        W[3] <= 1'b0;
    end
    else begin
        if(W[1]) begin
            if(short_command) begin
                W[1] <= 1'b1;
                W[2] <= 1'b0;
                W[3] <= 1'b0;
            end
            else begin
                W[1] <= 1'b0;
                W[2] <= 1'b1;
                W[3] <= 1'b0;
            end
        end
        else if(W[2]) begin
            if(long_command) begin
                W[1] <= 1'b0;
                W[2] <= 1'b0;
                W[3] <= 1'b1;
            end
            else begin
                W[1] <= 1'b1;
                W[2] <= 1'b0;
                W[3] <= 1'b0;
            end
        end
        else if(W[3]) begin
            W[1] <= 1'b1;
            W[2] <= 1'b0;
            W[3] <= 1'b0; //  重置为初始状态
        end
    end
end
```

### 4.2.6 ST0 状态机

这部分是一个状态机，用于控制指令的执行状态。

```
// ST0 状态机 - 指令执行状态
always @(negedge T3 or negedge CLR) begin
    if (!CLR) begin
        ST0 <= 1'b0;
    end
    else begin
        if (!ST0 && ((WRITE_REG && W[2]) || (READ_MEM && W[1]) ||
                    (WRITE_MEM && W[1]) || (INS_FETCH && W[2])))
begin
            ST0 <= 1'b1;
        end
        else if (ST0 && (WRITE_REG && W[2])) begin
            ST0 <= 1'b0;
        end
    end
end
```

### 4.2.7 操作模式译码

这部分逻辑用于根据 SWC_SWB_SWA 信号判断当前的操作模式。

```
// 操作模式译码
assign WRITE_REG = (SWC_SWB_SWA == 3'b100);
assign READ_REG = (SWC_SWB_SWA == 3'b011);
assign INS_FETCH = (SWC_SWB_SWA == 3'b000);
assign READ_MEM = (SWC_SWB_SWA == 3'b010);
assign WRITE_MEM = (SWC_SWB_SWA == 3'b001);
```

### 4.2.8 指令译码

这部分逻辑用于根据 IR7_IR4 信号判断当前的指令类型。

```
// 指令译码
assign ADD = (IR7_IR4 == 4'b0001) && INS_FETCH && ST0;
assign SUB = (IR7_IR4 == 4'b0010) && INS_FETCH && ST0;
assign AND = (IR7_IR4 == 4'b0011) && INS_FETCH && ST0;
assign INC = (IR7_IR4 == 4'b0100) && INS_FETCH && ST0;
assign LD = (IR7_IR4 == 4'b0101) && INS_FETCH && ST0;
assign ST = (IR7_IR4 == 4'b0110) && INS_FETCH && ST0;
assign JC = (IR7_IR4 == 4'b0111) && INS_FETCH && ST0;
assign JZ = (IR7_IR4 == 4'b1000) && INS_FETCH && ST0;
assign JMP = (IR7_IR4 == 4'b1001) && INS_FETCH && ST0;
assign STP = (IR7_IR4 == 4'b1110) && INS_FETCH && ST0;
assign NOP = (IR7_IR4 == 4'b0000) && INS_FETCH && ST0;
assign OUT = (IR7_IR4 == 4'b1010) && INS_FETCH && ST0;
assign NOT = (IR7_IR4 == 4'b1101) && INS_FETCH && ST0;
assign CMP = (IR7_IR4 == 4'b1100) && INS_FETCH && ST0;
assign MOV = (IR7_IR4 == 4'b1011) && INS_FETCH && ST0;
assign DEC = (IR7_IR4 == 4'b1111) && INS_FETCH && ST0;
```

### 4.2.9 控制信号的生成

这部分逻辑用于根据当前的操作模式、指令类型和状态，生成所有控制信号。

```
// 全时序逻辑控制 - 所有控制信号在 T3 下降沿更新
always @(negedge T3 or negedge  CLR) begin
    if (!CLR) begin
        // 复位所有控制信号
        SELCTL <= 1'b0;
        DRW <= 1'b0;
        LPC <= 1'b0;
        PCINC <= 1'b0;
        PCADD <= 1'b0;
        LAR <= 1'b0;
        ARINC <= 1'b0;
        LIR <= 1'b0;
        LDZ <= 1'b0;
        LDC <= 1'b0;
        CIN <= 1'b0;
        M <= 1'b0;
        MEMW <= 1'b0;
        ABUS <= 1'b0;
        SBUS <= 1'b0;
        MBUS <= 1'b0;
        STOP <= 1'b0;
        SHORT <= 1'b0;
        LONG <= 1'b0;
        S <= 4'b0000;
        SEL <= 4'b0000;
    end
    else begin
        // 控制信号合成 - 所有控制信号同步更新
        // （此处省略了具体的控制信号生成逻辑，详见代码）
    end
end
```

# 5 团队分工

王玉鑫：负责测试设备，基础功能代码的编写与维护，调试、测试代码，负责文档中流水式硬布线设计详解部分

穆文钦：负责测试设备，基础功能代码的编写，调试、测试代码，撰写文档

王书翰：负责测试设备，代码编写与维护，调试、测试代码；负责流水硬布线控制器的编写

曹忠昊：负责测试设备，代码维护以及运行测试程序，调试、测试代码；记录每天的日志以及文档中调试问题部分的编写

# 6 测试程序

## 6.1 测试集 1

目标: 测试核心的算术指令（ADD, SUB, INC）、内存与寄存器间的数据转移指令（LD, ST），并验证在条件不满足时，条件转移指令（JZ, JC）会继续顺序执行。

初始状态:寄存器 R2 = 12H,寄存器 R3 = 10H,内存 [10H] = 20H,内存 [11H] = 05H

最终状态：程序执行在 0AH 地址暂停。R0: 21H，R1: 05H，R2: 12H，R3: 11H 内存 [12H]: 25H ，DBUS (数据总线) 输出: 21H

| 地址 | 指令 | 机器码 | 功能 | 结果 |
|---|---|---|---|---|
| 00H | LD R0, [R3] | 0101 0011 | R0 ← [R3] | R0 ← 20H |
| 01H | INC R3 | 0100 1100 | R3 ← R3 + 1 | R3 ← 11H |
| 02H | LD R1, [R3] | 0101 0111 | R1 ← [R3] | R1 ← 05H |
| 03H | ADD R0, R1 | 0001 0001 | R0 ← R0 + R1 | R0 ← 25H |
| 04H | ST R0, [R2] | 0110 1000 | R0 → [R2] | 25H → [12H] |
| 05H | SUB R0, R1 | 0010 0001 | R0 ← R0 - R1 | R0 ← 20H |
| 06H | JZ 0AH | 1000 0011 | 如果 Z=1,则 PC ← @ + 03H | Z=0, 不跳转 |
| 07H | JC 0AH | 0111 0010 | 如果 C=1,则 PC ← @ + 02H | C=0, 不跳转 |
| 08H | INC R0 | 0100 0000 | R0 ← R0 + 1 | R0 ← 21H |
| 09H | OUT [R0] | 1010 0000 | DBUS ← R0 | DBUS ← 21H |
| 0AH | STP | 1110 0000 | 暂停运行 | 暂停 |

测试集 1

## 6.2 测试集 2

目标: 重点测试 CMP 指令设置标志位的功能，并验证 JZ 和 JC 在条件满足时能成功转移。同时测试 MOV 和 DEC 指令。

初始状态:寄存器 R3 = 20H，内存 [20H] = 55H，内存 [21H] = 55H，内存 [22H] = 10H，内存 [23H] = 30H。

最终状态：程序在 0FH 地址暂停。R0: E0H，R1: 30H，R3: 23H ，标志位: C=1

| 地址 | 指令 | 机器码 | 功能 | 结果 |
|---|---|---|---|---|
| 00H | LD R0, [R3] | 0101 0011 | R0 ← [R3] | R0 ← 55H |
| 01H | INC R3 | 0100 1100 | R3 ← R3 + 1 | R3 ← 21H |
| 02H | LD R1, [R3] | 0101 0111 | R1 ← [R3] | R1 ← 55H |
| 03H | CMP R0, R1 | 1100 0001 | Rd - Rs | Z=1, C=0 |
| 04H | JZ 07H | 1000 0010 | 如果 Z=1,则 PC ← @ + 02H | Z=1, PC ← 07H |
| 05H | DEC R0 | 1111 0000 | R0 ← R0 - 1 | (被跳过) |
| 06H | ADD R0, R1 | 0001 0001 | R0 ← R0 + R1 | (被跳过) |
| 07H | INC R3 | 0100 1100 | R3 ← R3 + 1 | R3 ← 22H |
| 08H | LD R0, [R3] | 0101 0011 | R0 ← [R3] | R0 ← 10H |
| 09H | INC R3 | 0100 1100 | R3 ← R3 + 1 | R3 ← 23H |
| 0AH | LD R1, [R3] | 0101 0111 | R1 ← [R3] | R1 ← 30H |
| 0BH | SUB R0, R1 | 0010 0001 | R0 ← R0 - R1 | R0 ← E0H, C=1 |
| 0CH | JC 0FH | 0111 0010 | 如果 C=1,则 PC ← @ + 02H | C=1, PC ← 0FH |
| 0DH | MOV R0, R1 | 1011 0001 | Rd ← Rs | (被跳过) |
| 0EH | OUT [R1] | 1010 0001 | DBUS ← R1 | (被跳过) |
| 0FH | STP | 1110 0000 | 暂停运行 | 暂停测试集 2 |

测试集 2

## 6.3 测试集 3

目标: 测试逻辑指令 AND 和 NOT，并测试 JMP 指令实现的无条件程序跳转。

初始状态:寄存器 R3 = 30H，内存 [30H] = AAH，内存 [31H] = 0FH，内存 [32H] = 40H (JMP 的目标地址)

最终状态：程序在 41H 地址暂停。R0: 05H，R1: 0FH，R2: 40H，R3: 32H，DBUS (数据总线) 输出: 05H

| 地址 | 指令 | 机器码 | 功能 | 结果 |
|---|---|---|---|---|
| 00H | LD R0, [R3] | 0101 0011 | R0 ← [R3] | R0 ← AAH |
| 01H | NOT R0 | 1101 0000 | Rd ← ¬Rd | R0 ← 55H |
| 02H | INC R3 | 0100 1100 | R3 ← R3 + 1 | R3 ← 31H |
| 03H | LD R1, [R3] | 0101 0111 | R1 ← [R3] | R1 ← 0FH |
| 04H | AND R0, R1 | 0011 0001 | Rd ← Rd ∧ Rs | R0 ← 05H |
| 05H | INC R3 | 0100 1100 | R3 ← R3 + 1 | R3 ← 32H |
| 06H | LD R2, [R3] | 0101 1011 | R2 ← [R3] | R2 ← 40H |
| 07H | JMP [R2] | 1001 1000 | PC ← Rd | PC ← 40H |
| ... | ... | ... | ... | ... |
| 40H | OUT [R0] | 1010 0000 | DBUS ← R0 | DBUS ← 05H |
| 41H | STP | 1110 0000 | 暂停运行 | 暂停测试集 3 |

测试集 3

## 6.4 测试集 4

目标: 此测试集主要利用循环强化测试分分支跳转，同时进行综合测试

初始状态: R2=60H，R3= FDH，存储器[60H]=67H，[61H] = 80H，[62H] = FDH，[80H] = 60H，[FEH] = 03H，[FFH] = 03H

最终状态：在经过三次循环后，于 14H 地址暂停。R0: 80H， R2: 60H ，R3: FDH DBUS (数据总线) 输出: 80H

| 地址 | 指令 | 机器码 | 功能 | 循环 1 结果 | 循环 2 结果 | 循环 3 结果 |
|---|---|---|---|---|---|---|
| 00H | LD R0, [R2] | 0101 0010 | R0 ← [R2] | R0 ← 67H | | |
| 01H | INC R2 | 0100 1000 | R2 ← R2 + 1 | R2 ← 61H | | |
| 02H | LD R1, [R2] | 0101 0110 | R1 ← [R2] | R1 ← 80H | | |
| 03H | ADD R0, R1 | 0001 0001 | R0 ← R0 + R1 | R0 ← E7H | R0 ← 07H | R0 ← 86H |
| 04H | JC 06H | 0111 0001 | 如果 C=1，则 PC ← @ + 01H | C=0，不跳转 | C=1，PC ← 06H | C=0，不跳转 |
| 05H | AND R1, R0 | 0011 0100 | R1 ← R0 ∧ R1 | R0 ← 80H | | R0 ← 82H |
| 06H | SUB R0, R2 | 0010 0010 | R0 ← R0 - R2 | R0 ← 86H | R0 ← 04H | R0 ← 83H |
| 07H | INC R1 | 0100 0100 | R1 ← R1 + 1 | R1 ← 81H | R1 ← 82H | R1 ← 83H |
| 08H | ST R0, [R1] | 0110 0100 | R0 → [R1] | 86H → [81H] | 04H → [82H] | 83H → [83H] |
| 09H | INC R3 | 0100 1100 | R3 ← R3 + 1 | R3 ← FEH | R3 ← FFH | R3 ← 00H |
| 0AH | JZ 0DH | 1000 0010 | 如果 Z=1，则 PC ← @ + 02H | Z=0，不跳转 | Z=0，不跳转 | Z=1，PC ← 0DH |
| 0BH | LD R2, [R3] | 0101 1011 | R2 ← [R3] | R2 ← 03H | R2 ← 03H | |
| 0CH | JMP [R2] | 1001 1000 | PC ← R2 | PC ← 03H | PC ← 03H | |
| 0DH | INC R3 | 0100 1100 | R3 ← R3 + 1 | | | R3 ← 01H |
| 0EH | INC R3 | 0100 1100 | R3 ← R3 + 1 | | | R3 ← 02H |
| 0FH | SUB R0, R2 | 0010 0010 | R0 ← R0 - R2 | | | R0 ← 80H |

| | | | | | | |
|---|---|---|---|---|---|---|
| 10H | LD R2, [R0] | 0101 1000 | R2 ← [R0] | | | R2 ← 60H |
| 11H | ADD R3, R2 | 0001 1110 | R3 ← R3 + R2 | | | R3 ← 62H |
| 12H | LD R3, [R3] | 0101 1111 | R3 ← [R3] | | | R3 ← FDH |
| 13H | OUT R0 | 1010 0000 | DBUS ← R0 | | | DBUS ← 80H |
| 14H | STP | 1110 0000 | 暂停运行 | | | |

测试集 4

## 6.5 测试集 5

目标: 该测试集专门**测试流水线 bug**。考虑到吞掉指令一般是在两个特定的指令组合下才发生的，我们收集了一些其他组的吞指令组合，例如 JMP + LD ，JMP+ST ，SUB + INC 等，设定专门的测试集进行测试

初始状态：寄存器 R3 = 10H ，内存 [10H] = 50H，内存 [11H] = 20H，内存 [12H] = 0AH (第一个 JMP 的目标地址)，内存 [13H] = 0DH (第二个 JMP 的目标地址)，内存 [20H] = FFH (ST 指令的目标地址)

最终状态：R0: 31H，R1: 20H，R2: 0DH ，R3: 12H，DBUS (数据总线) 输出: 31H

| 地址 | 指令 | 机器码 | 功能 | 结果 |
|---|---|---|---|---|
| 00H | LD R0, [R3] | 0101 0011 | R0 ← [R3] | R0 ← 50H |
| 01H | INC R3 | 0100 1100 | R3 ← R3 + 1 | R3 ← 11H |
| 02H | LD R1, [R3] | 0101 0111 | R1 ← [R3] | R1 ← 20H |
| 03H | SUB R0, R1 | 0010 0001 | R0 ← R0 - R1 | R0 ← 30H |
| 04H | INC R0 | 0100 0000 | R0 ← R0 + 1 | R0 ← 31H |
| 05H | INC R3 | 0100 1100 | R3 ← R3 + 1 | R3 ← 12H |
| 06H | LD R2, [R3] | 0101 1011 | R2 ← [R3] | R2 ← 0AH (加载跳转地址) |
| 07H | JMP [R2] | 1001 1000 | PC ← R2 | PC ← 0AH |
| 08H | LD R1, [R0] | 0101 0100 | R1 ← [R0] | (被 JMP 指令跳过) |
| 09H | INC R3 | 0100 1100 | R3 ← R3 + 1 | (被 JMP 指令跳过) |
| 0AH | LD R2, [13H] | 0101 10.. | R2 ← M[13H] | R2 ← 0DH (加载新跳转地址) |
| 0BH | JMP [R2] | 1001 1000 | PC ← R2 | PC ← 0DH |
| 0CH | ST R0, [R1] | 0110 0001 | R0 → [R1] | (被 JMP 指令跳过) |
| 0DH | OUT [R0] | 1010 0000 | DBUS ← R0 | DBUS ← 31H |
| 0EH | STP | 1110 0000 | 暂停运行 | 暂停运行 |

测试集 5

# 7 调试过程中的问题及讨论

## 7.1 初期组合逻辑问题

在项目初期，团队主要面临由组合逻辑设计不严谨引发的问题。

### 7.1.1 问题：所有控制信号灯上电后恒亮。

分析与过程：首次将代码部署到 TEC-8 平台后，发现几乎所有控制信号灯都被点亮。通过在写寄存器模式（SW=100）下调试，团队观察到只有 sel 信号在正常变化。进一步分析代码发现，其他信号只在满足条件时被赋值为 1，但缺少在不满足条件时被清零的逻辑，导致信号一旦置高便无法拉低。

解决方案： 团队采用了一个直接的临时方案，即在 always 语句块的开始，于每个 T3 时钟下降沿无条件地将所有信号赋零。虽然这能解决眼前的问题，但团队也意识到这种做法可能引入“线与”问题，是一个潜在隐患。

### 7.1.2 问题：写寄存器模式下，程序在写 R2、R3 状态之间无限循环。

分析与过程：测试中发现，在写完 R3 后，程序没有按预期返回写 R1 的状态，而是在 R2 和 R3 之间循环。分析认为这是由于 STO 信号一直为 1 导致的。根本原因是在编写代码时，忽视了 STO 信号在写寄存器、读写存储器、取指令等不同操作模式下的多重含义，导致其置位条件 SSTO 的逻辑不精确。

解决方案：通过修改 SSTO 的组合逻辑，为其加入对不同 SW 模式和 W 周期的精确判断，成功解决了该问题。

### 7.1.3 问题：取指令模式下，处理器卡在 LPC 阶段。

分析与过程： 在测试指令执行时，发现处理器始终停留在 LPC（加载 PC）阶段无法继续。

解决方案： 这本质上与上一个问题同源，是由于 SSTO 的逻辑中忘记添加取指令模式（SWC_SWB_SWA == 3'b000）的判断条件导致的。添加相应逻辑后问题解决。

## 7.2 流水线设计问题

### 7.2.1 问题：流水线执行 INC 指令时出现不必要的空操作周期。

分析与过程：在测试流水线时，团队发现单周期的 INC 指令实际上占用了 W1 和

W2 两个周期，且 W1 周期没有任何操作。原因是流水线版本由基础版改编而来，基础版中的 INC 是双周期指令。在流水化改造中，虽然逻辑上取指周期（W1）的功能已被前一条指令代劳，但团队忘记将 INC 的执行逻辑判断条件从 W2 改为 W1。

解决方案：将 INC 指令的执行周期判断条件修改为 W1 后，空操作周期问题得到解决。

## 7.3 核心时序与同步问题

这是调试过程中最核心、最复杂的部分，问题从表面现象逐步深入到底层时序关系。

### 7.3.1 问题：处理器“吞指令”，即执行完当前指令后跳过下一条指令，直接执行下下条。

分析与过程： 这个问题在使用复杂测试集时暴露出来。在老师的指导下，团队了解到问题在于 LIR 和 PCINC 的执行时序不一致，理论上应同时发生，但实际表现为 PCINC 先于 LIR 执行。团队最初尝试用门电路延迟信号，但未成功，并意识到他们对微指令执行时间、寄存器值修改时间以及 T/W 周期之间正确的时序关系存在根本性的误解。

最终根因分析： 经过对 TEC-8 原理图和芯片手册的研究和讨论，团队在 7 月 3 日最终定位了“吞指令”的根源：虽然程序的状态（state）被锁存了，但最终输出的控制信号（如 LIR, PCINC）本身没有被锁存。这些 wire 类型的输出信号在 T3 下降沿到来前的任何输入变化都可能直接影响其状态，导致时序错乱。

解决方案： 将所有输出控制信号的类型改为 reg，并确保它们在时序逻辑中被正确地赋值。这样可以保证这些信号被 D 触发器锁存，在下一个 T3 下降沿到来前保持稳定，从而确保 PCINC 和 LIR 能够同步执行，彻底解决“吞指令”问题。

### 7.3.2 问题：按下 CLR 复位后，首次按 QD 无任何输出信号。

分析与过程： 这是在重写时序代码后出现的新问题。团队分析后认识到，TEC-8 平台给出的 W1、W2、W3 信号仅为状态周期的“标记”，其产生时机与团队自己设计的状态机存在偏差。

解决方案： 团队决定舍弃使用系统提供的 W1、W2、W3 输入，改为根据 T3 的下降沿和不同指令的长度，在内部自行设计状态机来生成 W 周期信号。这样就确保了逻辑的自洽和时序的精确可控，解决了复位后的延时问题。

# 8 设计调试小结

对于调试部分，总结经验如下：

本实验的调试方法核心不应该在计算结果上，而在于观察控制信号灯以及 PC，IR，AR，W1,W2,W3,所以这对我们对每个状态的微指令以及微指令的功能要很熟悉，在此基础上便可以迅速定位代码中的具体的错误模块，观察计算结果只是一个快速检验程序正确性的方法之一。

本实验核心是设计一个硬布线控制器，但对于 TEC-8 平台来说，与微程序控制器并无差别，所以在基础功能时候，可以调到微程序模式对比观察控制信号灯的变化，看是否和我们的硬布线控制器一样，特别是 4 种基础 SW 模式，理论上控制信号灯的效果应该和微程序模式一样

测试集应该设计的全面，最开始我们便因为通过了简单测试集就认为代码设计正确，到之后才发现基础版本都需要更改，浪费了大量时间。针对特殊错误，如不同的指令组合可能会导致吞掉指令，应该设计专门的测试集进行测试，这里也可以理解为单元测试吧。虽然最后验收前已经通过了我们设计的所有测试集（还有一些太长的测试集并没有在报告中体现），但倘若作为一个工程师，事实上我们这里给出的测试集还远远不能我们的程序是完全正确的。

Verilog 代码风格和设计一定要符合规范，否则会出现不符合预期的结果。除此以外，还可以通过查看网表，观察是否生成了想要的 D 触发器,如图 8.1 可知输出信号被正确的保持住了。

调试的时候一定要耐心仔细，分析问题要全面

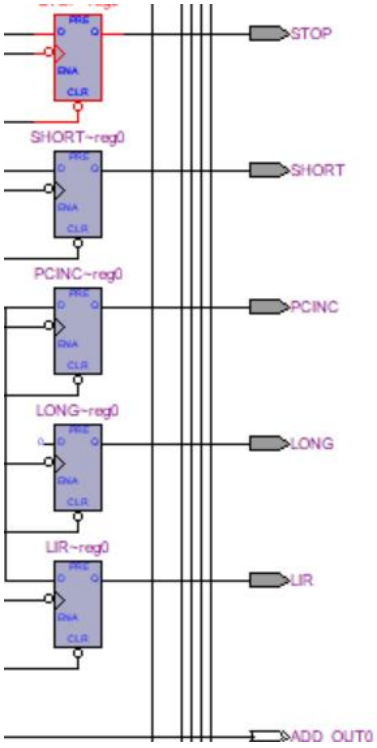
STOP
SHORT~reg0
SHORT
PCINC~reg0
PCINC
LONG~reg0
LONG
LIR~reg0
LIR
ADD_OUT0

图 8.1

# 附录一 小组各成员心得总结

**王玉鑫**

通过本次计算机组成原理课程设计，我深刻体会到了理论与实践相结合的重要性。通过本次课程设计，我加深了对硬连线控制器的理解。在计算机组成原理理论课的期末复习阶段时，我就对设计硬连线控制器的部分印象深刻。在基础功能代码的编写与维护过程中，我将课堂上学到的硬布线控制器理论知识应用到了实际的代码实现中。通过对指令周期流程图的设计和译码表的编写，我更加清晰地理解了指令的执行过程以及各个控制信号之间的时序关系。而本次实践也让我对硬件更加感兴趣，特别是自己设计测试程序时，让我也体会到了硬件控制程序的乐趣。在小组合作中，我们也分工明确，以较高的效率完成了本次课程设计，提高了沟通能力，加强了团队意识。在调试和测试代码的过程中，我遇到了诸如标志位设置错误、程序逻辑错误等问题，但在一次次的排查和修正中，我逐步提高了自己的问题解决能力，也更加明白了细节在编程中的重要性。除此之外，这次课程设计也让我复习了上学期学习的 verilog 语法，特别是阻塞赋值和非阻塞赋值操作，这与平时接触的程序语言很不同，让我印象深刻。同时，我也负责文档中设计详解部分的撰写，让我学会了如何将复杂的技术内容以清晰、规范的方式呈现出来，这不仅锻炼了我的写作能力，也加深了我对整个设计流程的理解。总之，这是一次收获很大的课程设计，让我对计算机组成原理的理解得到了很大地提高，也让我对 TEC-8 实验台的操作更加熟练。这次课程设计让我认识到，只有将理论知识转化为实际操作能力，才能真正掌握一门技术。在未来的学习中，我将继续注重理论与实践的结合，不断提升自己的专业技能。

**穆文钦**

参与本次计算机组成原理课程设计，让我收获颇丰。通过这次深入的实验探索，我对硬件描述语言的理解不仅停留在表面，而是达到了一个全新的层次。特别是对于verilog语言这一特定领域，我有了更为深刻的认识，它与我们日常编程时所熟悉的高级语言之间的差异，如今在我心中刻画得更为清晰。这种差异不仅仅体现在语法结构上，更在于其独特的设计思路和实现机制，这让我意识到硬件编程的复杂性和独特魅力。在基础功能代码的编写过程中，我深入理解了硬布线控制器的工作原理，通过对指令的分解和控制信号的生成，我学会了如何将抽象的指令转化为具体的硬件操作。在实际操作中，我遇到了一些在软件设计中鲜少遇到的挑战，比如程序没有按照想要的要求运行，出现了错误的信号。这些信号可能源自电路设计的细微疏漏或是硬件资源的分配不当。面对这些棘手问题，我不得不从多个角度进行思考，结合电路原理、信号流向以及逻辑分析，逐步排查并定位问题所在。在调试代码时遇到的各种问题，如信号冲突等，让我意识到硬件编程的复杂性和严谨性，也促使我不断学习和探索解决问题的方法。撰写文档的过程则让我学会了如何系统地整理和总结实验过程，这不仅有助于团队成员之间的沟通和协作，也为后续的工作打下了坚实的基础。在这个过程中，我的调试技能得到了显著提升。更重要的是，我培养了一种系统性的思维模式，能够从整体架构出发，逐层分解问题，再由局部到整体，逐步验证解决方案的有效性。这种能力的提升，不仅对当前的实验项目大有裨益，也将成为我未来学习中的宝贵的财富。通过这次实验，我不仅巩固了专业知识，也提升了实践能力和团队协作能力，为未来的学习和工作积累了宝贵的经验。

**王书翰**

在本次课程设计中，我主要承担了测试设备的操作、代码的维护以及运行测试程序的任务，同时还负责记录每天的实验日志和负责文档中调试问题部分的编写。本次课程设计的时间较为紧迫，任务较为繁重。但我们依旧靠团队的力量和个人的不懈努力，成功地完成了设计任务。在计算机组成原理理论学习中，硬连线控制器的设计就是一个复杂而关键的部分。而在此次课程设计中，我通过绘制指令流程图和设计控制操作时序等，逐步掌握了硬连线控制器的设计流程。通过对测试设备的反复操作，我熟悉了 TEC-8 实验台的各种功能和操作流程，能够熟练地进行程序的运行和调试等操作。在代码维护过程中，我学会了如何对代码进行优化和改进，确保代码的稳定性和可读性。在实践过程中，我深刻体会到了设计测试程序的重要性，锻炼了我的逻辑思维和问题解决能力。通过测试程序，我更加明确了 verilog 语言与高级语言之间的差异，这种差异不仅体现在语法结构上，更在于其独特的设计思路和实现机制，这种差异让我深刻认识到硬件编程的复杂性及其独有的吸引力。它不仅拓宽了我对硬件编程的认识，也为我未来的学术探索和技术研究提供了宝贵的视角和深刻的洞见。运行测试程序时，我遇到了各种各样的问题，如程序运行结果与预期不符、硬件信号异常等。在实际操作中，我也遇到了一些在软件设计中不曾遇到的挑战，比如出现了跳过一条指令，同一指令执行了两次的问题。通过逐步逆推找到问题根源，但查阅诸多的资料却依旧无法解决。 这让我意识到硬件设计中问题的隐蔽性和复杂性以及自身知识的局限性。我意识到不仅在面对未知问题时需要更加细致和耐心地排查过程，更要努力学习更多知识，以面对未来更多可能出现的复杂问题。但在团队成员的共同努力下，我们逐一解决了这些问题，这个过程让我深刻体会到了团队协作的力量。记录实验日志和调试问题部分的编写，让我养成了良好的记录习惯，也让我学会了如何清晰地表达实验过程和遇到的问题，这对于后续的实验总结和报告撰写有着重要的意义。这次实验让我认识到，硬件编程不仅需要扎实的理论基础，还需要严谨的态度和良好的记录习惯。同时我也意识到，记录不仅仅是对过程的总结，更是对知识的积累和对问题的反思。通过这次实验，我学会了如何从失败中汲取经验，如何在团队中发挥自己的作用，这些都将是我未来职业生涯中的宝贵财富。综上所述，本次课程设计虽然面临时间紧迫和任务繁重的挑战，但通过团队的协作和个人的不懈努力，我们不仅成功完成了设计任务，而且在这一过程中获得了宝贵的经验。我加深了对硬连线控制器的理解，体验了硬件设计的乐趣与挑战，并在实践中锻炼了逻辑思

维和问题解决能力。

**曹忠昊**

在本次课程设计中，我主要负责了流水硬布线控制器和带有中断的硬布线控制器的设计与实现。本次实验的目的是利用 Quartus 软件为硬连线控制器进行编程，在这个过程中使用的芯片为 Altera EPM7128。在这次的课程设计中，我深刻体会到了事先详尽的资料收集与整理对于后续程序设计及调试工作的重要性。只有拥有了足够的知识储备，才能够更快速地解决在实验过程中遇到的问题，并增强自己灵活应变的能力。我在本次课程设计中进行的调试工作就应用了这一部分的经验，而这也进一步锻炼了我的问题解决能力和创新思维。在流水线设计方面，我深入研究了如何将取指令和执行指令的过程重叠，以提高指令的执行效率。通过合理安排时序，我成功解决了流水线中的冲突问题，实现了高效的指令执行。在中断功能的实现中，我遇到了诸多挑战，如中断无法触发、中断周期执行错误等。但在不断地尝试和调试中，我逐步攻克了这些难题，最终成功实现了中断功能。这个过程不仅锻炼了我的逻辑思维能力和问题解决能力，也让我对中断处理机制有了更深入地理解。在小组合作方面，我们本次的小组合作过程非常愉快，而这得益于清晰明了的事前分工。同时，这也使得我们能够以较高的效率完成本次的课程设计。高效且和谐的合作氛围不仅加速了项目的进度，也让我们在相互学习中不断成长，增强了个人在团队环境中的适应性和贡献度。这次课程设计的经验无疑增强了我的沟通合作能力，让我更加适应了适用于团队的工作流程。同时，通过与团队成员的密切合作，我学会了如何在复杂的项目中分工协作，共同推进项目的进展。这次实验让我对计算机组成原理有了更全面的认识，也让我在硬件设计和编程方面有了更大的进步。总的来说，我从这次的课程设计中收获了许多宝贵的经验。无论是扎实的专业知识、敏锐的问题解决能力，还是高效的沟通合作技巧，都是未来职业生涯中不可或缺的财富。希望在未来的某一天，我能够学以致用，将它们应用于各种类型的工作之中。

# 附录二 调试日志

## A.1 6 月 30 日

我们在 6 月 30 日前完成了第一版组合型基础功能代码，在 30 号进行测试，当天遇到的问题，思考，以及解决办法如下

当第一次把代码下载在 TEC-8 平台上的时候，发现一下载好程序，控制信号灯几乎就全亮，我们尝试忽略信号灯的信息，依旧按照正常的流程先测试 4 种 SW 模式，但是最后测试结果是显然错误的。我们意识到应该通过观察信号灯来查错，于是我们用 SW=100，即写寄存器的模式来仔细调试观察信号灯，同时对比我们的代码。在此过程中，我们观察到只有 sel 信号在随着每一次 QD 在变化.根据这个现象分析代码，发现在写寄存器模式下，sel 信号与其他信号的唯一区别是 sel 信号既可以在一定判断条件被赋值为 0，也可以在另外的判断条件下被赋值为 1，而其他信号只会在 if 或者 case 语句满足的时候赋值为 1，不存在赋值为 0 的情况，所以出现了灯全亮的情况。当时采用了一个简单直接的做法，即在 always 语句一开始就使用**无复位信号判断的赋零操作**，即在每次 T3 下降沿来临时，首先无条件把所有信号赋为 0。然后测试发现这种做法是能够解决我们问题的，但是这种做法会产生线与问题，当时测试时候没看出其带来的影响，选择暂时忽略。

继续测试，发现在写寄存器的时候出现了写 R0,R1,R2,R3,R2,R3,R2,R3 状态，即处于写 R2，R3 的循环。理论上写完 R3 后应该返回到写 R1.经过阅读代码以及图 1 可知，分析到可能的原因是在写寄存器时候 ST0 信号一直为 1，所以一直在写 R2 和写 R3 中循环。最后，我们发现是写代码的时候忽视了 **ST0 在不同操作下（写寄存器，读写储存器、取指令）的多种含义**，导致在写寄存器操作下的 ST0=1 状态也被当作循环了。因此，我们通过对 SST0（ST0 置位的条件，SSTO 为 1 的时候才将 ST0 置位为 1）组合逻辑进行修改，即

```
assign SST0 = (ST0 == 1'b0) && (
                                    (SWC_SWB_SWA == 3'b001 && W1) || //  写存储器模式，W1 有效
                                (SWC_SWB_SWA == 3'b100 && W2) || //  写寄存器模式，W2 有效
                                (SWC_SWB_SWA == 3'b010 && W1) || //  读存储器模式，W1 有效
);
```

经过重新测试写寄存器操作，可以观察到问题成功解决

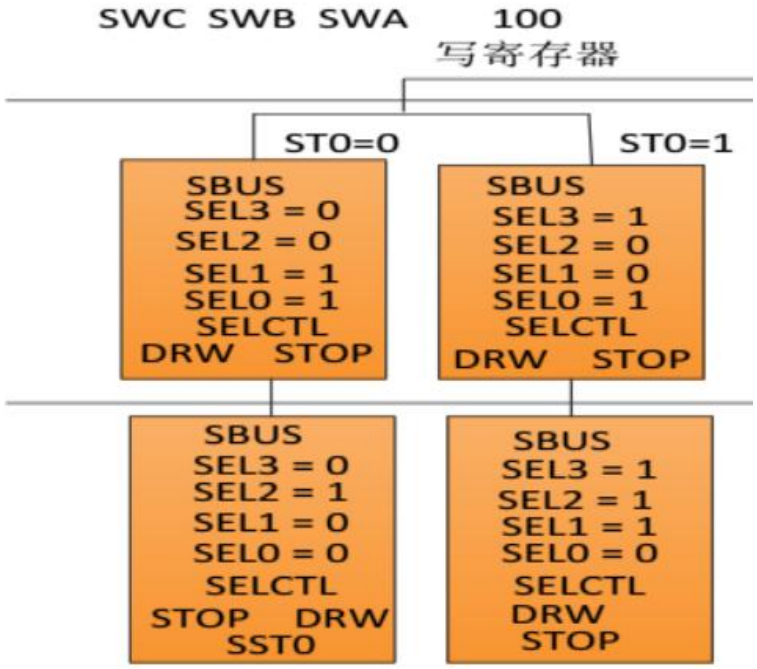


图 1

在尝试完基础的 4 种 SW 模式后，尝试写测试指令进行运行，在取指令执行模式下观察到处理器始终处于 LPC 阶段，如图 2 不会进入到下一个状态，通过调试以及分析信号灯和代码的关系，发现实际原因是在写取指令代码的时候忘记了更新 SST0（ST0 置位的条件）导致的，这里加上其取指令相关组合逻辑即可，本质上还是应该归结于 ST0 的赋值更新逻辑，即添加 SST0=(SWC_SWB_SWA == 3'b000 && W1)。

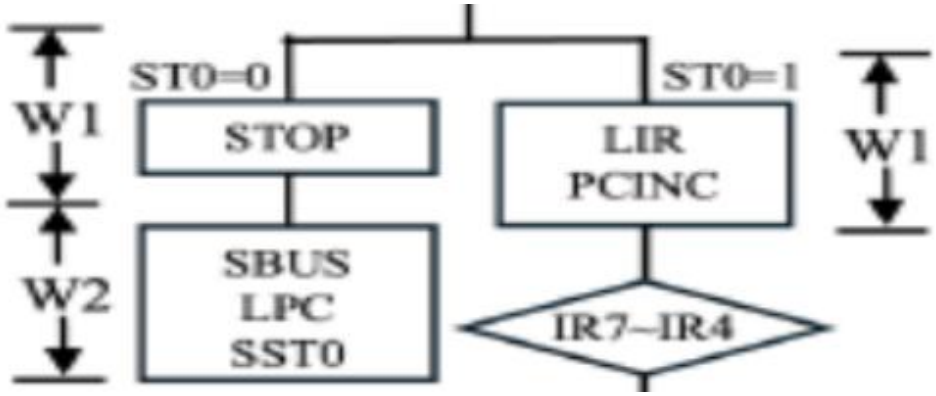


图 2

## A.2 7 月 1 日

今日在基础版本上测试了组合逻辑的流水线版本。首先仍然是测试流水线的 4 种基本操作模式在确认无误后，再测试流水线指令执行情况。在测试流水线时候，我们发现存在空操作周期的问题。例如测试数据的第一条指令为 LD ，第二条指令为 INC ，理论上，在我们设计的流水线指令流程图 图 3 中，LD 为双周期指令，INC 为单周期指令，在实际测试中，LD 的两个周期内控制信号灯正确，但是 INC 指令却有 W1，W2 两个周期，并且 W1 周期内没有任何操作，即**空操作周期**，而所有操作都在 W2 周期内。经过阅读代码，发现原因如下：流水线版本是在基础功能的代码上改编而来，而基础功能的 INC 指令为 2 周期指令，逻辑上我们把 W1 周期的取值操作（即取指令）提前到了上一条指令的最后一个周期执行，INC 就变成了单周期指令，然后需要把 W2 周期即执行周期移动到 W1 周期，但是实际代码实现时候忘记了把 INC 指令的执行周期判断条件由 W2 改为 W1，所以导致了以上问题。最后把判断条件修改为 W1，便解决了问题。我们在简单测试集上测试了该版本无误（这是不全面的，后面会再次提及原因）后，便开始思考将组合逻辑修改为时序逻辑，经过讨论，决定设计自动机和状态转移方程，最后的输出引脚信号由状态寄存器的各种组合逻辑确定

## A.3 7 月 2 日

昨日我们已经有了一种正确的方向，但在代码实现的时候我们进行了一些错误的简化，将组合逻辑改为了自己所认为是正确的时序逻辑。我们对其进行了编译和调试，能够运行基本的测试集，最后在下午得出了基本功能的最初版，之后我们又探究了如何将其修改为流水线，在较为简单的测试集上测试通过了流水线版本，此时我们认为革命胜利了转而去完善我们的组合逻辑，但当我们尝试新的复杂的测试集时，却发现存在**吞指令**的问题——当前指令执行完后会直接执行下下条指令，但不会执行下一条指令。在老师的指导下，我们得知了是 LIR 和 PCINC 的执行顺序发生了错误，理论上 LIR 和 PCINC 应该同时给出，但根据实验现象可知，是先 PCINC 再 LIR，我们尝试用增加与或非门来对信号进行延迟，但最后无功而返。在与老师以及其他同学的讨论中，我们意识到，我们之前错误理解了微指令的执行时间、寄存器值修改的时间、以及 T1、T2、T3、W1、W2、W3 之间正确的先后顺序和关系。于是我们各自上网查询 TEC-8 以及 EPM74ls28 芯片的说明书，想要理明白这些关系。在回到本部后，又经过了一晚上的讨论分析，对此前提到的时序系统的理解便有了一个初步的正确认识，在此基础上，我们重写了基础

版本代码。

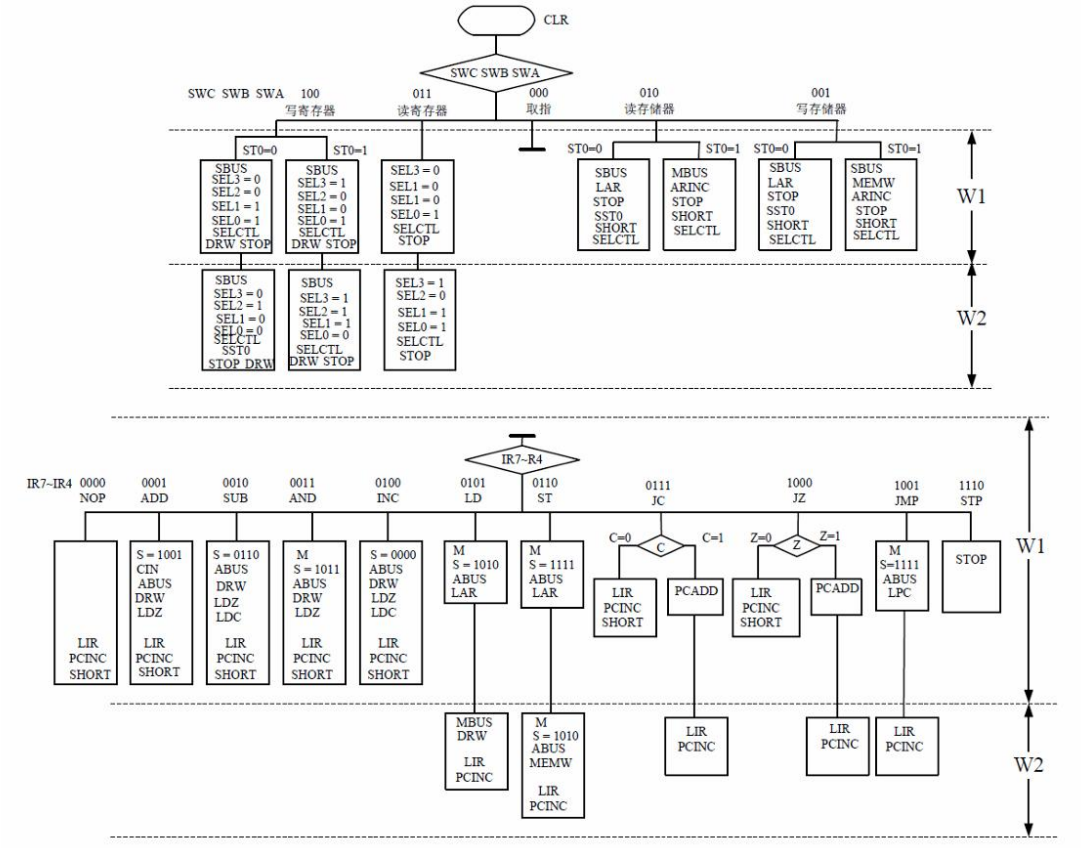


图 3

## A.4 7 月 3 日

在测试了昨天晚上赶出的代码的时候，我们发现运行过程中又出现了时延的新问题即当按下 CLR 之后，第一次按 qd 没有任何输出信号，且显示为 W1 状态；对这个的解决方法是分析老师发出的图片以及查阅相关的资料，我们认识到这个错误的原因来自于老师发在群里的时序关系问题。本质上 W1，W2，W3 是 TEC-8 给出的时序标志，仅仅起到**标记状态周期**的作用，但是平台的标记和我们的理解是有偏差的，所以我们当然可以舍弃系统的 W1、W2、W3 输入，根据 T3 的下降沿自行设计 W1，W2，W3 的状态转移，以及不同指令的长度自己设定 W1、W2、W3 的输入.，这样就能够保证 按下 clr 之后 第一次按 QD 能够判断 W1，从而正确执行指令，自此我们解决了时延问题。但是，我们仍然没有解决吞指令的问题，我们结合群里时延问题的示意图，以及 TEC-8 平台的原理图，进一步讨论出可能在指令执行的过程中只是锁住了状态而忽略了**输出信号的锁存**，由于最后的输出的控制信号还是 Wire 类型，所以在 T3 下降沿来临前输入的改变会

通过某种方式影响到输出。我们给出的解决方案是：使用 REG 类型来存储输出信号以及状态标识，这样能够确保这些输出信号和当前状态在下一个 T3 下降沿之前会被寄存器锁住，不会改变，只要保证了输出的控制信号能够在下一个 T3 下降沿之前保持稳定，则就能够保证 PCINC 和 LIR 同时执行，便可解决吞指令问题。吸取之前测试集过于简单导致的 bug 遗漏，我们收集了其他组的吞指令组合，最后测试了若干测试程序以及多种高发吞指令组合，发现没有再出现吞指令现象。在等待老师验收的过程中，我们无意间想到对比所有代码版本的网表图，我们惊奇地发现。虽然我们 6 月 30 日第一版中的所有输出控制信号都是 reg 类型，但是最终网表图中控制信号并不是有 D 触发器给出，这一点引起了我们的好奇，于是我们尝试对其进行修改，最后也能得到输出控制信号由 D 触发器给出的网表图。

# 附录三 小组各成员贡献度表

<table>
<tr><td colspan="5">已实现的功能（在实现的功能前打勾、扩展实验自己写功能），提交作业时一并提交</td></tr>
<tr><td>基本功能</td><td>基本功能</td><td>◊ 顺序模型处理器<br>◊ 存储器功能<br>◊ 寄存器功能<br>◊ 加减乘除功能</td><td>附加功能</td><td>◊ 修改 6 条指令<br>◊ 修改 PC 指针功能<br>◊ 其他：_____</td></tr>
<tr><td>拓展题目</td><td>基本功能</td><td>实现流水线功能</td><td>附加功能</td><td></td></tr>
</table>

| 学 号 | 姓名 | 承担的工作 | 贡献度（总共 100%） |
|---|---|---|---|
| 2023210896 | 王书翰 | 负责测试设备，代码编写与维护，调试、测试代码；负责流水硬布线控制器的编写 | 25 |
| 2023211964 | 曹忠昊 | 负责测试设备，代码维护以及运行测试程序，调试、测试代码；记录每天的日志以及文档中调试问题部分的编写 | 25 |
| 2023210892 | 穆文钦 | 负责测试设备，基础功能代码的编写，调试、测试代码，撰写文档 | 25 |
| 2023210882 | 王玉鑫 | 负责测试设备，基础功能代码的编写与维护，调试、测试代码，负责文档中流水线硬布线设计详解部分 | 25 |