

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		学院	计算机学院 (国家示范性软件学院)	指导教师	杨秦
班级	班内序号	学号		学生姓名	成绩	
2023211308	3	2023211102		苏泽勤		
2023211308	4	2023211103		冯仁宇		
2023211308	12	2023211111		张启涵		
2023211308	20	2023211119		汪顺		
课程设计内容	<p>本次课程设计旨在全面提升我们对计算机组成原理和数字系统设计的实践能力。核心任务是设计并实现一个硬连线顺序模型和流水型处理器，涵盖了数据通路与控制器协同、Verilog HDL 硬件描述、以及完整的 CPLD/FPGA 开发流程。我们不仅实现了基础的无毛刺控制器和指令扩展，更挑战流水线控制器的设计，通过解决数据和控制冒险等高级问题来优化性能，最终在 ALTERA EPM7128 上成功验证了我们基于行为级建模的时序逻辑架构中同时发挥数据流建模中组合逻辑的优势的成功设计。我们还编写了具有实际功能的测试程序，通过数字示波器、逻辑测试笔在硬件平台上进行细致调试，尤其是通过在资源受限下用软件优化弥补硬件不足，培养了解决实际工程问题的能力。小组四个人共同完成了需求分析和报告撰写，冯仁宇、张启涵、苏泽勤负责代码的设计、编写与调试，冯仁宇、苏泽勤负责编写测试程序，汪顺负责所有报告中图、表格的绘制。</p>					
学生课程设计报告(附页)	见附页					
课程设计成绩评定	<p>遵照实践教学大纲并根据以下四方面综合评定成绩：</p> <ol style="list-style-type: none"> 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 2、团队分工是否恰当与合理 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 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 <p>评语：</p> <p>成绩：</p> <p style="text-align: right;">指导教师签名：</p> <p style="text-align: right;">年 月 日</p>					

计算机组成原理课程设计报告

苏泽勤, 冯仁宇, 张启涵, 汪顺

2025 年 7 月 6 日

目录

1 团队分工	3
2 开源说明	4
2.1 为什么选择开源?	4
2.2 如何访问和贡献	4
3 课题硬件环境及总体描述	5
3.1 硬件环境	5
3.2 总体描述	5
4 顺序模型处理器	6
4.1 需求分析	6
4.2 概要设计	6
4.2.1 控制信号的产生	7
4.3 设计详解	8
4.3.1 1.基础功能部分	8
4.3.2 2.拓展功能部分	15
5 流水线模型处理器	21
5.1 需求分析: 流水线硬连线控制器设计	21
5.2 概要设计	21
5.3 设计详解	23
6 测试程序	28
6.1 测试目的	28
6.2 测试程序设计	28
6.3 测试结果与分析	32
6.4 测试程序及对应的C语言代码	33
7 附件1 各成员心得总结	38
7.1 苏泽勤	38
7.2 冯仁宇	38
7.3 张启涵	39
7.4 汪顺	39

8 附件2 工作日志	41
8.1 6月30日	41
8.1.1 今日进展	41
8.1.2 设计思路	41
8.1.3 遇到的问题与解决方法	41
8.2 7月1日	41
8.2.1 今日进展	41
8.2.2 设计思路	41
8.2.3 遇到的问题与解决方法	41
8.3 7月2日	42
8.3.1 今日进展	42
8.3.2 设计思路	43
8.3.3 遇到的问题与解决方法	43
8.4 结语	43
9 附件3 贡献度表	44
10 附件4 RTL仿真图	45

1 团队分工

姓名	学号	工作
苏泽勤	2023211102	需求分析, 代码设计、编写与调试, 编写测试程序, 报告撰写
冯仁宇	2023211103	需求分析, 代码设计、编写与调试, 编写测试程序, 报告撰写
张启涵	2023211111	需求分析, 代码设计、编写与调试, 报告撰写
汪顺	2023211119	需求分析, 报告中图表的绘制, 报告撰写

2 开源说明

我们坚信开源精神在技术发展和知识共享中的重要性。因此，我们已将本项目完全开源，托管于 GitHub 平台。我们希望通过开放我们的代码和设计思想，能够促进学习、激发讨论，并为社区贡献一份力量。

2.1 为什么选择开源？

- **促进知识共享与学习：**作为一个包含了 CPU 硬连线控制器设计、流水线实现以及复杂数据结构测试的综合项目，我们认为它对计算机体系结构、数字逻辑设计以及 Verilog HDL 学习者都具有参考价值。开源能让更多人访问、学习我们的实现细节，从而加深对相关知识的理解。
- **接受社区反馈与协作：**我们深知任何项目都不是完美的。开源允许来自全球的开发者审阅我们的代码，提出宝贵的建议、发现潜在的缺陷，甚至提交改进的贡献。这种开放的协作模式有助于持续提升项目的质量和鲁棒性。
- **实践与传播开源文化：**我们希望通过实际行动支持开源运动。将项目开源是我们践行“取之于社区，回馈于社区”理念的方式，鼓励更多人参与到开源生态中来。
- **提供透明度与可信赖性：**对于任何硬件设计项目，透明性至关重要。开源让所有人都能检查我们的设计逻辑，理解其工作原理，从而建立对其功能和性能的信任。

2.2 如何访问和贡献

本项目已托管在 GitHub 仓库：https://github.com/suzeqin/hardwired_controller。

我们欢迎所有感兴趣的个人和团队查看、使用本项目。如果你对项目有任何疑问、建议，或者希望贡献代码，请通过以下方式参与：

- **提出 Issue：**如果你发现 Bug、有新功能建议或任何疑问，请在 GitHub 仓库的 Issues 页面提交。
- **提交 Pull Request：**如果你希望贡献代码改进项目（例如优化逻辑、修复 Bug、添加新功能等），请 Fork 仓库，进行修改，并提交 Pull Request。我们会认真审阅你的贡献。
- **交流与讨论：**我们也鼓励在 Issues 或其他合适的地方进行技术讨论，共同进步。

我们期待与广大开发者社区一同，让这个项目变得更好。

3 课题硬件环境及总体描述

3.1 硬件环境

- TEC-8计算机硬件综合实验系统，搭载ALTERA EPM7128，开发平台为Quartus。
- EPM7128SLC84-15是一种复杂可编程逻辑器件（CPLD），属于Altera公司的MAX 7000S系列。它的封装为PLCC，芯片制程是0.5微米，采用CMOS工艺。CMOS门电路由P型与N型MOS管构成。它的晶体管数量是2500个，可以实现128个逻辑宏单元和68个输入/输出端。
- 调试器件：数字示波器、逻辑测试笔。

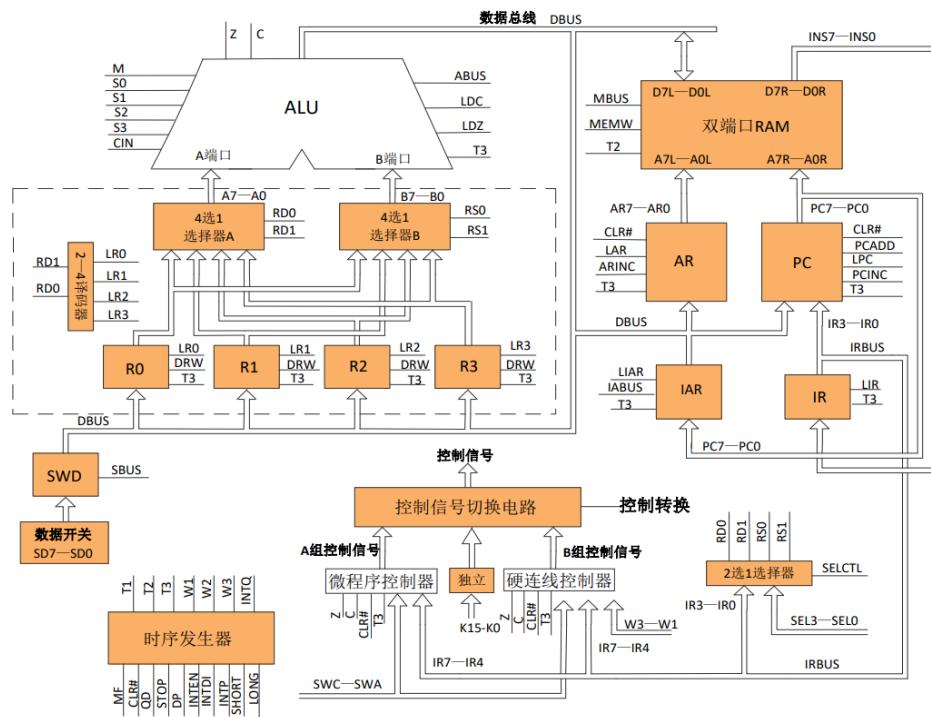


图 1: TEC-PLUS(8)模型机系统框图

3.2 总体描述

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。

基本功能： 在实验平台上进行组装、调试并运行成功

附加功能：

- 在原指令基础上要求扩指功能至少3条
- 修改PC指针功能（指针任意执行功能）课程设计任务

拓展： 基于Altera CPM7128（XILINX Spartan-6）的流水硬连线控制器设计

实现难点： 硬件平台实现流水型控制器，由于存在冒险问题，执行测试程序时也许会发生吞指令现象，观察与分析问题，找到解决方案。

4 顺序模型处理器

4.1 需求分析

目标系统: 设计一个基于硬布线控制器的顺序模型处理器。 **核心功能:**

- 控制器类型: 硬布线控制器 (Hardwired Controller)。
- 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
- 执行模式: 顺序执行指令 (Sequential), 即严格按照“取指-执行”的顺序串行地完成一条指令后再开始下一条。
- 指令系统: 必须实现一个预定义的指令集。
- 数据通路: 必须基于给定的数据通路结构进行设计。

需要达到的标准:

- 设计方案能够在Quartus平台编译通过。
- 将设计下载到EPM7128芯片后, 能够在TEC-8实验系统上正确组装和调试。
- 处理器能够成功运行测试程序, 并且结果符合预期。

“取指-执行”串行执行:

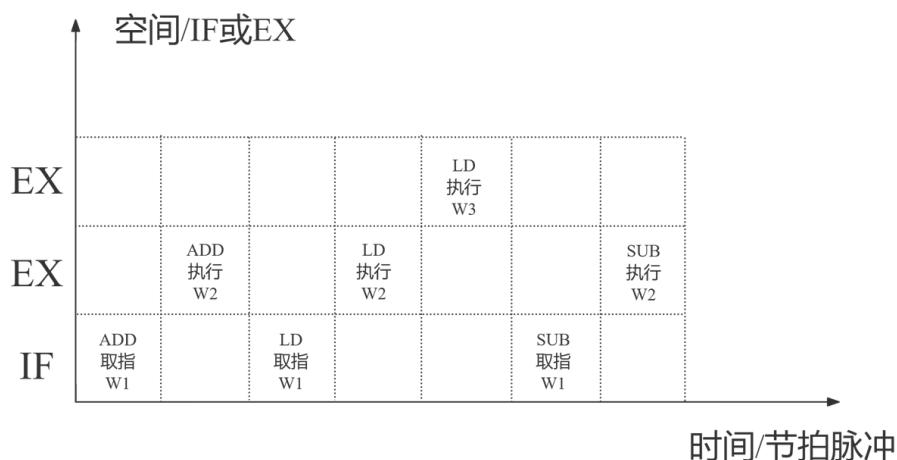


图 2: “取指-执行”串行执行示意图

4.2 概要设计

硬布线控制器, 也叫做组合逻辑控制器, 是一种将处理器的控制逻辑直接通过固定硬件电路实现的设计方法。它不像微程序控制器那样依赖于存储在控制存储器中的微指令, 而是通过门电路和触发器构成一个复杂的网络来生成各种控制信号。这种控制器设计的核心目标是追求极致的速度和最少的元件使用。由于控制信号的产生仅仅是电路门延迟的结果, 因此它能提供比微程序控制器更快的运行速度。

4.2.1 控制信号的产生

硬布线控制器生成特定微操作控制信号 C 的原理，可以用以下逻辑函数来描述：

$$C = f(Im, Mi, Tk, Bj)$$

在这个公式中：

- Im 代表指令操作码译码器输出，它告诉控制器当前指令的类型。
- Mi 是节拍电位，指示当前指令执行周期的特定阶段（例如 W1、W2、W3）。
- Tk 是节拍脉冲，提供精确定时，确保操作在正确的时钟周期内发生。
- Bj 是状态条件信号，比如ALU产生的零标志（Z）或进位标志（C），它们影响条件分支等操作。

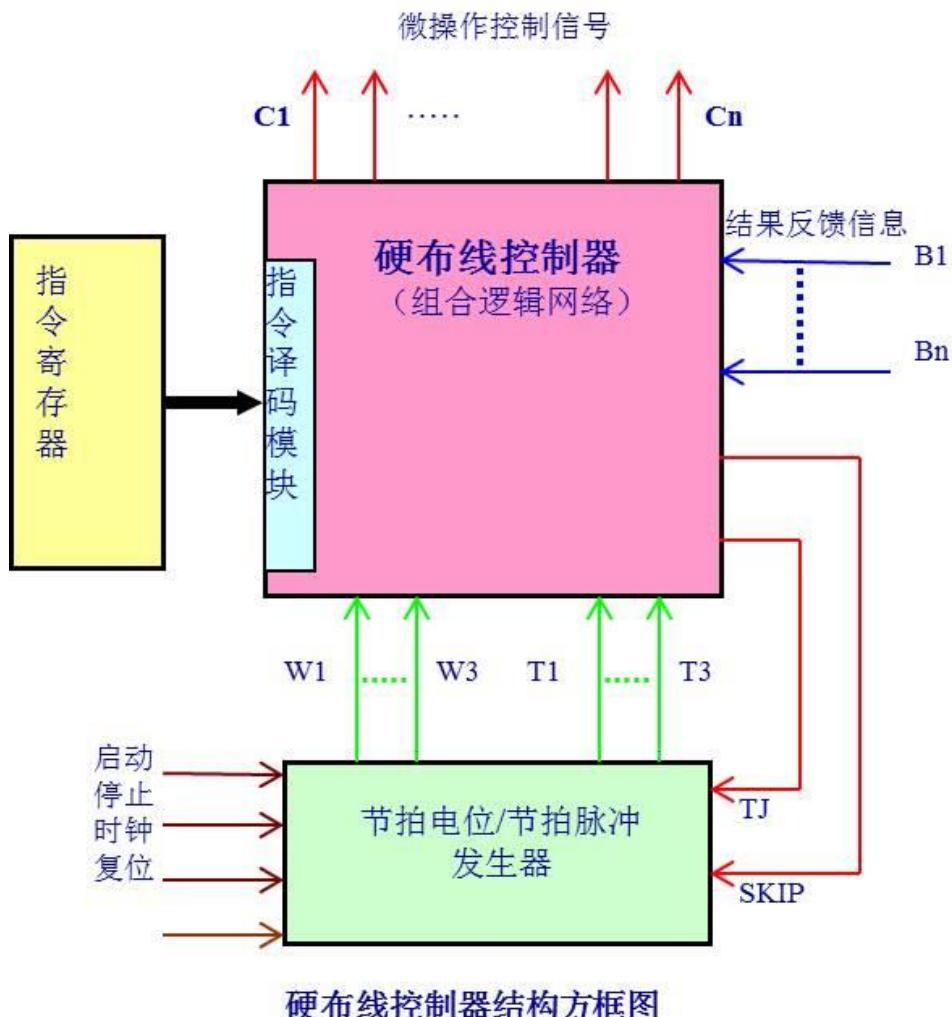


图 3: 硬布线控制器结构方框图

系统顶层架构: 整个处理器系统由两大部分组成：数据通路（Datapath）和控制器（Controller）。

- **数据通路:** 包含执行指令所需的所有硬件单元，如程序计数器(PC)、指令寄存器(IR)、地址寄存器(AR)、数据寄存器(DR)、算术逻辑单元(ALU)以及连接这些单元的内部总线。
- **控制器:** 是整个处理器的大脑。它根据从指令寄存器(IR)中获取的操作码(IRH)以及状态标志(如C标志、Z标志)，生成一系列控制信号，精确地指挥数据通路中各个部件在正确的时钟周期内执行相应的操作(如读/写寄存器、内存读/写、ALU运算等)。

控制器设计思想:

- **硬布线逻辑:** 控制器的逻辑将使用组合逻辑电路(由逻辑门构成)实现。其输入为指令操作码、时钟信号和状态标志，输出为数据通路所需的全部控制信号。
- **状态机模型:** 采用时序状态机来定义指令执行的流程。对于顺序处理器，一个指令周期可以划分为固定的几个状态(W1-W3)。
 - 取指周期：W1
 - 执行周期：对于大部分指令(ADD、SUB等)，只需要一个节拍W2来进行执行。特殊地，对于ST和LD指令，W2用来给出内存地址，W3用来访存。
- **Verilog模块设计：**

– 输入:

- * SWC, SWB, SWA: 模式选择开关，用于选择手动读写或自动执行等模式。
- * oriW1, oriW2, oriW3: 外部时序控制信号，用于手动单步操作。
- * CLR: 全局异步复位信号，用于初始化控制器。
- * T3: 系统主时钟。
- * IRH[3:0]: 指令寄存器的高4位，即操作码。
- * C, Z: ALU的状态标志位(进位和零)。

– 输出:

- * 数据通路中的所有控制信号，如DRW(写数据寄存器), LPC(加载PC), MEMW(内存写), S[3:0](ALU操作选择)等。

– 内部逻辑:

- * 状态寄存器 cnt: 用于记录当前所处的T状态。
- * 组合逻辑: 根据输入(SWCBA, IRH, C, Z)和当前状态(cnt)计算出所有输出控制信号的下一状态值(_next)。
- * 时序逻辑: 使用一个always @(negedge T3 or negedge CLR)块，在时钟下降沿将计算出的下一状态值(_next)赋给对应的输出寄存器，并更新状态寄存器cnt。这种全寄存器输出的设计可以有效避免组合逻辑产生的毛刺，使系统更稳定。

4.3 设计详解

4.3.1 1.基础功能部分

1.1 处理器架构与控制方式

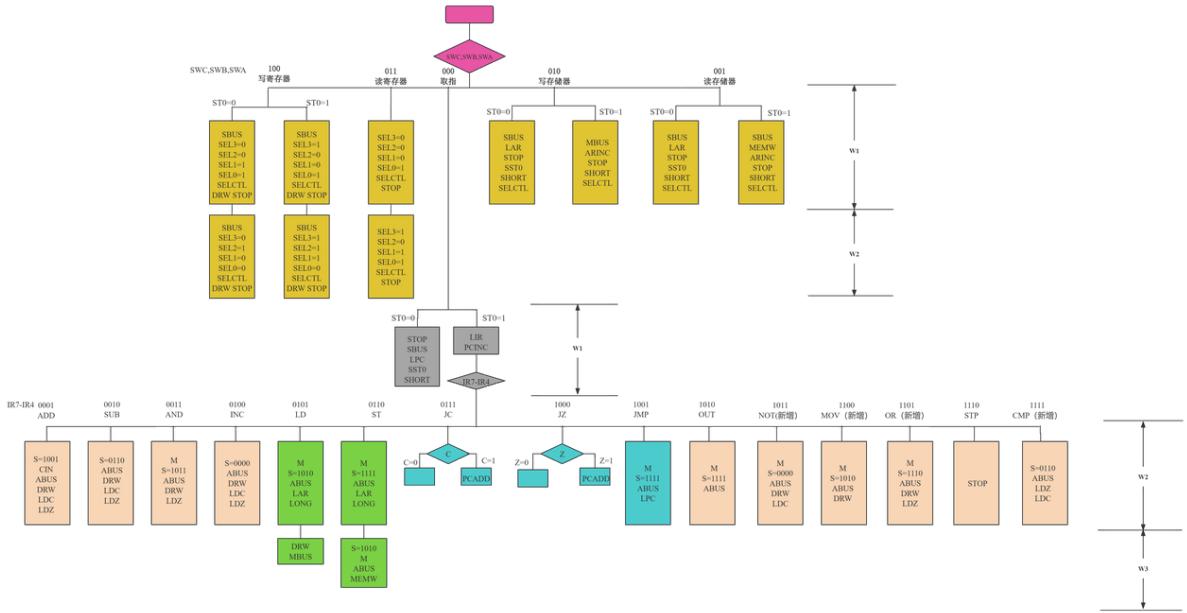


图 4: 顺序模型控制器流程图

- 硬连线控制器:** 处理器核心的控制单元必须采用硬连线逻辑实现, 即控制信号的生成直接由组合逻辑决定。
- 顺序模型处理器:** 指令执行遵循严格的顺序, 每个指令周期 (取指、执行等) 的微操作是串行完成的, 没有指令级的并行处理 (这是与流水线的关键区别)。
- 全时序逻辑输出:** 所有输出控制信号 (DRW, PCINC, LPC, LAR, etc.) 必须是寄存器类型 (reg), 并在时钟 T3 的下降沿同步更新。这旨在消除组合逻辑毛刺, 确保系统稳定性, 是当前代码的核心设计原则, 也是我们的程序的创新之处。

1.2 指令集与微操作支持 (基于代码中的IRH操作码和控制信号逻辑) 控制器需根据输入的4位指令操作码 IRH (Instruction Register High) 和条件标志 c (Carry) / z (Zero) 来生成正确的微操作控制信号。支持的指令包括:

- 算术/逻辑指令:**

- OP_ADD (4'b0001): 加法, 可能影响 LDZ_next, LDC_next, CIN_next, S_next, ABUS_next。
- OP_SUB (4'b0010): 减法, 可能影响 LDZ_next, LDC_next, S_next, ABUS_next。
- OP_AND (4'b0011): 逻辑与, 可能影响 LDZ_next, S_next, M_next, ABUS_next。
- OP_INC (4'b0100): 加一, 可能影响 LDZ_next, LDC_next, S_next, ABUS_next。
- OP_OR (4'b1101): 逻辑或, 可能影响 LDZ_next, S_next, M_next, ABUS_next。
- OP_NOT (4'b1011): 逻辑非, 可能影响 LDC_next, S_next, M_next, ABUS_next。
- OP_CMP (4'b1111): 比较, 可能影响 LDC_next, S_next, LDZ_next, ABUS_next。

- 数据传输指令:**

- OP_LD (4'b0101): 从内存加载, 影响 DRW_next, LAR_next, S_next, M_next, ABUS_next, MBUS_next, LONG_next。
- OP_ST (4'b0110): 存储到内存, 影响 DRW_next, LAR_next, MEMW_next, S_next, M_next, ABUS_next, LONG_next。
- OP_MOV (4'b1100): 数据移动, 影响 DRW_next, S_next, M_next, ABUS_next。

- 控制流指令:

- OP_JC (4'b0111): 进位跳转, 受 C 标志影响, 影响 PCADD_next。
- OP_JZ (4'b1000): 零跳转, 受 Z 标志影响, 影响 PCADD_next。
- OP JMP (4'b1001): 无条件跳转, 影响 LPC_next, S_next, M_next, ABUS_next。

- 其他指令:

- OP_OUTA (4'b1010): 输出寄存器A内容, 影响 S_next, M_next, ABUS_next。
- OP_STP (4'b1110): 停止, 影响 STOP_next。

名称	助记符	功能	指令格式		
			IR7-IR4	IR3-IR2	IR1-IR0
加法	ADD Rd, Rs	Rd \leftarrow Rd + Rs	0001	Rd	Rs
减法	SUB Rd, Rs	Rd \leftarrow Rd - Rs	0010	Rd	Rs
逻辑与	AND Rd, Rs	Rd \leftarrow Rd and Rs	0011	Rd	Rs
加 1	INC Rd	Rd \leftarrow Rd + 1	0100	Rd	XX
取数	LD Rd, [Rs]	Rd \leftarrow [Rs]	0101	Rd	Rs
存数	ST Rs, [Rd]	Rs \rightarrow [Rd]	0110	Rd	Rs
C 条件转移	JC addr	如果 C=1, 则 PC \leftarrow @ + offset	0111	offset	
Z 条件转移	JZ addr	如果 Z=1, 则 PC \leftarrow @ + offset	1000	offset	
无条件转移	JMP [Rd]	PC \leftarrow Rd	1001	Rd	XX
输出	OUT Rd	DBUS \leftarrow Rd	1010	Rd	XX
逻辑非	NOT Rd	Rd \leftarrow ~Rd	1011	Rd	XX
逻辑或	OR Rd, Rs	Rd \leftarrow Rd \vee Rs	1100	Rd	Rs
移动值	MOV Rd, Rs	Rd \leftarrow Rs	1101	Rd	Rs
停机	STP	暂停运行	1110	XX	XX
比较	CMP Rd, Rs	Rd - Rs	1111	Rd	Rs

图 5: ISA示意图

1.3 时序节拍(W1-W3)

1.3.1 设计挑战 作为数字系统设计者, 我们深知时序控制在硬布线控制器中的重要性。在本项目中, 我们发现一个严重的硬件缺陷: 从指令译码器或底层控制逻辑生成的原始微操作使能信号 (我们称之为 oriW1, oriW2, oriW3) 存在固有的一个时钟周期延迟。这意味着, 如果我们直接使用这些 oriW 信号来驱动微操作, 那么实际的动作将比我们逻辑上期望的时刻晚一拍, 这将导致时序错误和性能下降。

为了克服这个挑战，我们构思并实现了一套前瞻性（look-ahead）的 W1, W2, W3 信号。这些信号并非简单的原始使能，而是通过复杂的组合逻辑，结合控制器当前的运行模式(SWCBA)、主状态机 (cnt) 的状态，以及对延迟 oriW 信号的“预判”，来提前一个时钟周期生成我期望的微操作使能。这样，当物理硬件上的 oriW 信号最终生效时，我们的控制器内部的微操作就能准时地启动，从而实现精确、无延迟的时序控制。W1、W2、W3 信号代表了指令周期中的三个主要时序阶段。它们是硬布线控制器实现复杂指令分步执行、消除时序毛刺和确保系统稳定性的重要机制。

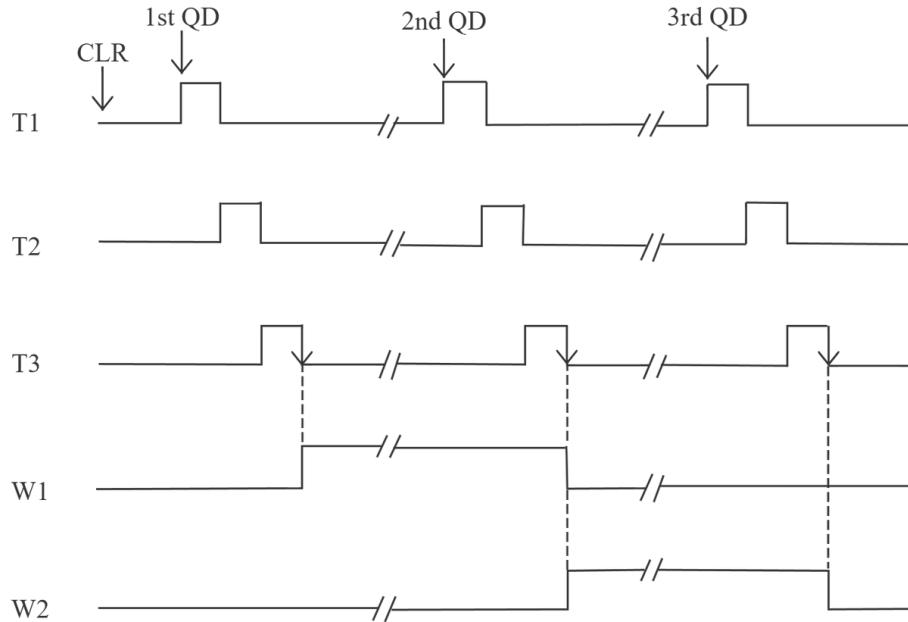


图 6: 基本时序波形

1.3.2 W 信号的理念：弥补硬件的缺陷

- **oriW1, oriW2, oriW3:** 这些是真实的硬件输出，从物理上承载了指令译码后的控制信息，但它们总是滞后一个时钟周期。
- **W1, W2, W3:** 这些是在 Verilog 代码中通过组合逻辑计算出的期望的使能信号。它们被设计成能提前一个时钟周期激活，以便在 oriW 信号的效应实际到达时，相应的微操作能够准时发生。

1.3.3 W 信号的详细设计与功能实现

1.3.3.1 W1 信号：通用启动与取指阶段 W1 为操作流程的第一个时序窗口。它负责指令周期中的通用启动和取指阶段的使能。Verilog 实现：

```
assign W1 = ((write_reg_mode || read_reg_mode) && ((!oriW1 && !oriW2) || oriW2)) || // 寄存器操作模式
          ((read_mem_mode || write_mem_mode) && ((!oriW1 && !oriW2) || oriW1)) || // 内存操作模式
```

```

(fetch_exec_mode && (cnt == 'STATE_T2) && // 取指-执行模式, T2 阶段 (复杂的预判逻辑)
(((IRH == 'OP_LD || IRH == 'OP_ST) && (!oriW1 && !oriW2 && !oriW3) || oriW3)) ||
((IRH != 'OP_LD && IRH != 'OP_ST) && ((!oriW1 && !oriW2 && !oriW3) || oriW2))) ||
(fetch_exec_mode && (cnt == 'STATE_T0 || cnt == 'STATE_T1)); // 取指-执行模式, T0/T1 阶段

```

设计考量与功能:

- 弥补 oriW 延迟: 在 fetch_exec_mode 下, 我们让 W1 在 cnt 的 T0 和 T1 周期就被逻辑激活。这意味着, 即使物理上的 oriW 信号有延迟, 我们的 PCINC (PC 自增)、LPC (加载 PC)、LIR (加载指令寄存器) 等取指微操作也能在期望的节拍开始。
- 多模式覆盖: W1 覆盖了所有主要操作模式的初始阶段。在寄存器和内存操作模式下, 它利用 oriW1/oriW2 的组合来前瞻性地启动如 SBUS (源总线选择)、LAR (加载地址寄存器) 等初始步骤。

1.3.3.2 W2 信号: 核心执行阶段 W2 为指令周期的核心执行时序窗口。它紧随 W1 之后, 在 ST0 信号被置位后, 驱动指令的核心运算和数据通路操作。Verilog 实现:

```

assign W2 = (write_reg_mode ||
             read_reg_mode ||
             (fetch_exec_mode && (cnt == 'STATE_T2)) ) && oriW1;

```

设计考量与功能:

- oriW1 的提前利用: W2 的核心逻辑是它依赖于 oriW1 的当前状态。由于 oriW1 是一个延迟的信号, 这意味着当 oriW1 变为高电平 (即它实际的效应在下一拍产生时), W2 已经在本拍逻辑上被激活。这使得像 ALU 运算、标志位加载、数据通路选择等大量核心微操作能够在期望的执行拍准时启动。
- ALU 操作与跳转: 诸如 S (ALU 功能选择)、LDZ (加载零标志)、LDC (加载进位标志)、ABUS (A 总线使能) 等与运算和数据流相关的微操作, 都在 W2 的激活下发生。对于 JMP, JC, JZ 等跳转指令, PCADD 和 LPC 也是在 W2 阶段完成的。
- W2 承载了大部分单周期指令的主要执行逻辑。

1.3.3.3 W3 信号: 内存数据传输 Verilog 实现:

```

assign W3 = fetch_exec_mode && (cnt == 'STATE_T2) &&
           oriW2 && (IRH == 'OP_LD || IRH == 'OP_ST);

```

设计考量与功能:

- 针对 LD/ST 的特殊处理: W3 的激活严格限定在 fetch_exec_mode 的 STATE_T2 状态, 并且仅针对 LD 和 ST 指令。这反映了对内存操作复杂性的认识——它们通常需要额外的时序来确保地址稳定、数据准备就绪。
- oriW2 的提前利用与实际延迟: W3 的激活依赖于 oriW2 的状态。W3 是在确保地址和数据已在 W1/W2 阶段稳定后, 才最终在期望的拍内, 基于提前获得的 oriW2 信息去激活内存传输。这确保了在 MEMW 或 MBUS 实际作用时, 所有先决条件 (地址、数据) 都已就绪且稳定。

- W3 是 LD 和 ST 指令完成其核心内存数据读写功能的最后时序阶段。

W的逻辑表

通过精心设计 W1, W2, W3 信号, 我们成功地在硬布线控制器中实现了对微操作的精确时序控制。这些信号不仅作为指令周期内不同阶段的标志, 更重要的是, 它们通过巧妙地利用了(并补偿了)底层硬件 `oriW` 信号的固有延迟, 确保了我的控制器能够按照逻辑上期望的时序精确地执行每一条指令, 从而保障了系统的稳定性和正确性。这种设计策略是克服物理时序约束, 实现高效、可靠数字系统性能的关键。

	读寄存器	写寄存器	读存储器	写存储器	取指
W1	(!oriW1 && !oriW2) oriW2	(!oriW1 && !oriW2) oriW2	(!oriW1 && !oriW2) oriW1	(!oriW1 && !oriW2) oriW1	cnt == `STATE_T2 (cnt == `STATE_T0) cnt == `STATE_T1)
W2	1	1			cnt == `STATE_T2
W3					(cnt == `STATE_T2) && oriW2 && (IRH == `OP_LD IRH == `OP_ST)

图 7: 顺序模型控制器中W的逻辑表

1.4 复位机制 异步复位: CLR 为低电平有效, 用于将所有寄存器(包括 ST0, DRW, cnt 等)异步复位到初始状态。

1.5 多操作模式控制 模式输入: 通过 SWCBA (SWC, SWB, SWA) 三位输入来选择不同的操作模式。
模式功能: 控制器需正确响应这些模式, 生成相应的控制信号:

- `write_reg_mode` (3'b100)
- `read_reg_mode` (3'b011)
- `read_mem_mode` (3'b010)
- `write_mem_mode` (3'b001)
- `fetch_exec_mode` (3'b000): 标准程序执行模式。

1.6 设计特色 我们的控制器是TEC-8计算机综合实验系统核心处理器的控制单元。我们的设计着重于全时序逻辑输出, 旨在提高系统稳定性, 并采用了清晰的数据流建模与行为级建模相结合的方法。这种方法避免了纯粹的行为级建模中常见的“线与”问题, 提升了我们的程序性能。

我们的控制器模块是一个顺序硬布线控制器, 负责根据输入的指令操作码、时序信号和状态标志, 生成一系列精确的控制信号, 以指挥数据通路中的各个组件执行相应的微操作。其核心设计理念在于:

- **硬布线控制:** 所有控制信号的生成逻辑都直接通过组合逻辑电路实现, 以最大限度地提升操作速度。
- **全时序逻辑输出:** 所有的输出控制信号都被设计为`reg`类型, 并在统一的时钟(T3)的下降沿进行同步更新。这种设计有效地消除了组合逻辑可能产生的毛刺, 确保了系统输出的稳定性和可靠性, 是本设计的创新之处。
- **数据流建模与行为级建模相结合:** 控制逻辑通过`assign`语句进行数据流建模, 计算出每个控制信号在下一个时钟周期的理想值(即`_next`信号)。这些`_next`信号随后在`always @(negedge T3 or negedge CLR)`块中被同步加载到对应的`reg`类型输出端口或内部状态寄存器中。

1.6.1 组合逻辑部分：assign语句实现数据流建模 该部分负责计算所有内部寄存器和输出端口的“下一状态”值。我们利用Verilog的assign语句，通过数据流建模的方式，根据当前的输入信号（如SWCBA模式、IRH操作码、C/Z标志、外部oriW信号以及内部状态ST0和cnt）来决定每个控制信号在下一个时钟周期应为何值。

- **模式解码：** SWCBA输入被解码为write_reg_mode, read_reg_mode, read_mem_mode, write_mem_mode, fetch_exec_mode等模式信号，以区分手动操作和自动指令执行。
- **节拍信号生成：** 内部的W1, W2, W3节拍信号通过复杂的组合逻辑生成。值得注意的是，W1的生成逻辑考虑了各种操作模式，并且包含了节拍重映射的逻辑，将外部oriW信号向后推迟一个机器周期，以解决TEC-8实验板外部时序信号的特定问题。
- **控制信号生成：** 对于每一个输出控制信号（例如DRW_next, PCINC_next, S_next等），我们都编写了详细的assign语句。这些语句定义了在哪个操作模式、哪个指令周期（由W1, W2, W3指示）以及满足何种条件（如C, Z标志）下，该控制信号应被置为高电平。例如：
 - DRW_next（数据寄存器写使能的下一状态）在write_reg_mode下或当处理器在fetch_exec_mode下执行特定指令（如ADD, SUB, LD等）的对应节拍时为高。
 - S_next（ALU操作选择）根据当前指令操作码（IRH）和所处的节拍（W1或其它）来决定ALU应执行何种操作。
- **状态寄存器ST0_next的计算：** ST0是一个内部状态标志，用于控制微操作的阶段划分。其ST0_next逻辑根据当前模式和节拍来决定是否进入或退出该阶段。

通过这种方式，我们清晰地将控制逻辑的决策部分与状态的存储部分分离，使得代码结构清晰，易于理解和调试。

1.6.2 时序逻辑部分：always块实现同步更新 这部分是整个控制器的核心时序驱动器，通过一个always @(negedge T3 or negedge CLR)块实现。

- **异步复位：** 当CLR信号为低电平（有效）时，所有的内部状态寄存器（ST0, cnt）和外部输出寄存器（DRW, PCINC等）都会被立即清零，确保系统能够快速回到一个已知的初始状态。
- **同步更新：** 当CLR为高电平且时钟T3的下降沿到来时，所有在组合逻辑部分（通过assign语句）计算出的_next值，都会被同步加载到对应的reg类型寄存器中。例如：

```
ST0 <= ST0_next;  
DRW <= DRW_next;  
// ... 其他所有输出和状态寄存器
```

这种显式的同步更新机制，是实现“全时序逻辑输出”的关键，它确保了所有控制信号的变化都与系统时钟精确同步，从而有效避免了由组合逻辑传播延迟可能引起的毛刺，提高了系统的稳定性和抗干扰能力。

- **状态机cnt的更新：** cnt寄存器（2位宽，表示STATE_T0, STATE_T1, STATE_T2）也在此处更新，它驱动着指令执行流程中的各个节拍。在fetch_exec_mode下，cnt根据内部逻辑进行递增，并在达到特定状态后归零，实现指令周期的循环。

优势与调试考量

- **高稳定性:** 全时序逻辑输出消除了毛刺，使得数据通路接收到的控制信号干净稳定。
- **逻辑清晰:** `assign`语句的数据流建模直观地描述了控制信号的生成逻辑，而`always`块则清晰地定义了状态的转换和同步。
- **易于拓展:** 清晰的模块化和数据流设计使得后续添加新指令或修改现有指令逻辑变得相对容易，只需修改对应的`assign`语句和状态转换逻辑，而无需担心复杂的组合逻辑反馈路径。

在调试过程中，我们通过在实验板上观察各输出信号指示灯的亮灭情况，并结合Verilog代码中`next`信号的逻辑，我们能够精确地定位问题所在。例如，当某个输出信号“该亮的没亮”或“不该亮的亮了”时，我们可以快速回溯到对应的`assign`语句，分析其输入条件是否满足或计算逻辑是否存在缺陷，从而高效地进行问题排查和修复。

4.3.2 2.拓展功能部分

2.1 指令集扩展 为了提升本微处理器的功能性和灵活性，我们对原有指令集进行了扩展，引入了四条新的指令：`OR` (按位或)、`NOT` (按位非)、`MOV` (数据移动) 和 `CMP` (比较)。这些指令的加入，旨在增强处理器的逻辑运算能力、数据传输效率以及条件判断的灵活性，从而为更复杂的程序提供支持。本次指令集扩展的设计目标是确保新指令的正确实现，同时严格遵循现有硬布线控制器的架构，并保证不对原有指令的执行造成任何负面影响。

名称	助记符	功能	指令格式		
			IR7-IR4	IR3-IR2	IR1-IR0
逻辑非	NOT Rd	Rd $\leftarrow \sim$ Rd	1011	Rd	XX
逻辑或	OR Rd, Rs	Rd \leftarrow Rd \vee Rs	1100	Rd	Rs
移动值	MOV Rd, Rs	Rd \leftarrow Rs	1101	Rd	Rs
比较	CMP Rd, Rs	Rd = Rs	1111	Rd	Rs

图 8: 扩展指令示意图

新指令的功能定义与操作码分配 为了兼容现有指令集并提供清晰的指令编码，我们为这四条新指令分配了如下操作码 (IRH):

- **OP_OR (4'b1101):**
 - 功能: 执行源操作数与目的操作数的按位或 (OR) 运算，并将结果存回目的操作数。
 - 示例: `OR R_dest, R_src (R_dest = R_dest | R_src)`
- **OP_NOT (4'b1011):**
 - 功能: 对目的操作数执行按位非 (NOT) 运算，并将结果存回目的操作数。
 - 示例: `NOT R_dest (R_dest = ~R_dest)`
- **OP_MOV (4'b1100):**
 - 功能: 将源操作数的值移动到目的操作数。
 - 示例: `MOV R_dest, R_src (R_dest = R_src)`

- OP_CMP (4'b1111):

- 功能：比较两个操作数（通常是目的操作数减去源操作数），根据比较结果更新零标志 (Z) 和进位标志 (C)，但不保存结果。
- 示例：CMP R_op1, R_op2 (执行 R_op1 - R_op2，只影响 Z 和 C 标志)

指令译码逻辑修改 为了使控制器能够识别并响应新指令，我们对指令译码逻辑进行了相应的修改。在 IRH 信号（4位操作码）的译码阶段，新增了对 4'b1101, 4'b1011, 4'b1100, 4'b1111 这四个新操作码的识别。这将确保当处理器取到这些指令时，控制通路能正确地识别其意图，并驱动后续的微操作序列。

微操作序列定义与实现 新指令的实现主要通过在现有状态机 (cnt) 的 STATE_T2 阶段定义其对应的微操作序列。我们尽可能复用现有的数据通路和控制信号，仅在必要时引入新的控制逻辑。以下是每条新指令在 fetch_exec_mode 下 ST0 为真，cnt 为 STATE_T2 时的微操作逻辑：

4.1 OP_OR (按位或) 指令 阶段： 主要在 W2 阶段完成。 **微操作定义：**

- S_next = 4'b1110: 设置 ALU 执行 OR 操作。
- M_next = 1'b1: ALU 工作在逻辑模式。
- DRW_next = 1'b1: 使能数据寄存器写入，保存 OR 结果。
- LDZ_next = 1'b1: 根据 ALU 结果更新零标志。
- ABUS_next = 1'b1: 使能 A 总线，提供 ALU 输入。
- LDC_next: 由于OR操作不产生进位，LDC保持原值或置0。

实现考量： 复用 ALU 的现有 OR 功能，以及标准的寄存器写入和标志位更新机制。

4.2 OP_NOT (按位非) 指令 阶段： 主要在 W2 阶段完成。 **微操作定义：**

- S_next = 4'b0000: 可能通过 ALU 的特定模式或旁路实现 NOT（或通过直接连线）。
- DRW_next = 1'b1: 使能数据寄存器写入，保存 NOT 结果。
- LDC_next = 1'b1: NOT 操作可能会影响进位标志（例如，如果输入最高位是1，输出最高位是0）。
- ABUS_next = 1'b1: 使能 A 总线，提供 ALU 输入。
- LDZ_next: NOT操作也可能影响Z标志，需根据实现决定是否置位。

实现考量： 需确保 ALU 或相关逻辑能正确执行按位非操作。考虑到 S_next 为 4'b0000，这可能意味着通过其他控制信号组合实现 NOT，或利用 ALU 的某种旁路模式。

4.3 OP_MOV (数据移动) 指令 阶段： 主要在 W2 阶段完成。 **微操作定义：**

- S_next = 4'b1010: 通常设置为 ALU 的直通 (Pass-through) 模式，将 A 输入直接传到输出。
- M_next = 1'b1: ALU 工作在逻辑模式（或直通模式）。

- DRW_next = 1'b1: 使能数据寄存器写入，保存移动的数据。
- ABUS_next = 1'b1: 使能 A 总线，提供 ALU 输入（源数据）。
- LDZ_next / LDC_next: MOV 操作通常不影响标志位，除非设计为根据移动值是否为零来更新 Z 标志。在当前设计中，未显式更新标志位。

实现考量：复用 ALU 的直通功能，实现数据的直接传输。

4.4 OP_CMP (比较) 指令 阶段：主要在 W2 阶段完成。微操作定义：

- S_next = 4'b0110: 设置为 ALU 执行减法操作 (Sub)。
- LDZ_next = 1'b1: 根据减法结果是否为零更新零标志。
- LDC_next = 1'b1: 根据减法是否产生借位更新进位标志。
- ABUS_next = 1'b1: 使能 A 总线，提供 ALU 输入。
- DRW_next = 0: 关键点，不使能数据寄存器写入，因为比较操作只影响标志位，不保存结果。

实现考量：利用现有 ALU 的减法功能。特别注意 DRW_next 必须为 0，以避免比较结果覆盖目的寄存器。

	ADD	SUB	AND	INC	LD	ST	JC	JZ	JMP	OUT	NOT	MOV	OR	STP	CMP	写寄存器	读寄存器	写存储器	读存储器	取指	修改PC指针
SST0																!ST0 && W2	!ST0 && W1	!ST0 && W1			!ST0 && W1
DRW	W2	W2	W2	W2	W2					W2	W2	W2				W1 W2					
PCINC																				ST0 && W1	
LPC										W2											!ST0 && W1
LAR					W2	W2												!ST0 && W1	!ST0 && W1		
PCADD							C && W2	Z && W2													
ARINC																	ST0 && W1	ST0 && W1			
SELCTL																1	1	1	1		
MEMW					ST0 && W3												ST0 && W1				
LIR																				ST0 && W1	
LDZ	W2	W2		W2						W2				W2							
LDC	W2	W2		W2						W2				W2							
CIN	W2																				
S0	!W1	!W1				!W1&W2				!W1	!W1										
S1	!W1	!W1			!W1	!W1				!W1	!W1		!W1	!W1							
S2	!W1					!W1&W2				!W1	!W1			!W1							
S3	!W1		!W1		!W1	!W1				!W1	!W1		!W1	!W1							
M	W2				W2	W2 W3				W2	W2	W2	W2	W2							
ABUS	W2	W2	W2	W2	W2	W2	W2 W3			W2	W2	W2	W2	W2	W2		1	ST0 && W1	W1		!ST0 && W1
SBUS																					
MBUS						W3												ST0 && W1			
SHORT																		1	1		!ST0 && W1
LONG					W2	W2															
SEL0																W1	1				
SEL1																(!ST0 && W1) (ST0 && W2)	W2				
SEL2																W2					
SEL3																ST0	W2				
STOP										ST0						1	1	1	1		!ST0

图 9: 译码表

2.2 PC 指针修改功能扩展 为了增强本微处理器的调试能力和单步执行的灵活性，我们扩展了一项关键功能：在 `fetch_exec_mode` (取指-执行模式) 下，允许用户在指令执行之前指定程序计数器 (PC) 的起始地址。这项功能使得用户能够精确控制程序的入口点。

功能目标 此功能是 `fetch_exec_mode` 下的特有行为。当控制器进入 `fetch_exec_mode`，用户可以通过数据输入总线提供目标地址，并通过特定的信号组合触发 PC 的加载。

PC 指针修改的核心在于将用户提供的地址数据传送到 PC 寄存器并通过 LPC 和 SBUS 信号加载。在 SWCBA=000 (`fetch_exec_mode`) 模式下，用户第一次按下 QD 后，应该进入等待阶段，此时用户通过数据总线输入一个地址，再次按下 QD，此时开始取值执行阶段。

- **数据输入：** 用户提供的代码首地址将通过数据总线进入系统。

- **LPC_next 逻辑的扩展：**

- 原有 `LPC_next` 和 `SBUS` 逻辑：

```
assign LPC_next = (fetch_exec_mode && ST0 && IRH == 'OP_JMP && W2); // JMP指  
令时的加载  
assign SBUS_next = (write_reg_mode) ||  
                    (read_mem_mode && !ST0 && W1) ||  
                    (write_mem_mode && W1);
```

- 扩展后的 `LPC_next` 和 `SBUS` 逻辑：

```
assign LPC_next = (fetch_exec_mode && ST0 && IRH == 'OP_JMP && W2) || // JMP指  
令时的加载  
                    (fetch_exec_mode && !ST0 && W1); // 修改PC指针  
assign SBUS_next = (write_reg_mode) ||  
                    (read_mem_mode && !ST0 && W1) ||  
                    (write_mem_mode && W1) ||  
                    (fetch_exec_mode && !ST0 && W1); // 修改PC指针
```

设计调试小结 **设计目标与初步实现：** 项目伊始，我们聚焦于理解处理器核心任务，并复习了 TEC-8 实验板的程序下载方法。我们深入学习了时序发生器的工作原理，特别是节拍电位信号 (W1, W2, W3) 与节拍脉冲 (T1, T2, T3) 的关系，这是我们硬布线控制器时序控制的基础。在代码实现上，我们创新性地采用了完全时序化的逻辑框架。所有输出和内部信号的下一周期值都通过 `assign` 语句建模，并在主时钟 T3 的下降沿进行同步更新。这种方法旨在消除组合逻辑毛刺，确保系统稳定性，并简化后续的调试和拓展。初期代码成功实现了基本的写/读寄存器和写/读存储器功能。**调试过程与问题发现：** 我们的调试方法是直观且实用的：

- 编写代码并运行测试样例。
- 细致观察实验板上所有关键信号的指示灯。
- 根据灯的亮灭情况进行判断：关注“该亮的没亮”或“不该亮的亮了”这类异常现象。

通过这种方法，我们发现了一个关键问题：在 CLEAR 操作并启动处理器后，理论上 CPU 应进入 W1 阶段，但指示灯显示 W1、W2、W3 均为 0。解决方法：针对 W1W2W3 信号异常的问题，我们基于对实验板

特性的理解和对代码的分析，判断可能是外部时序信号与我们内部期望的时序不匹配。我们采用了节拍重映射的策略，将外部的($oriW1$, $oriW2$, $oriW3$)信号逻辑上向后推迟一个机器周期，成功将其与我们内部的($W1$, $W2$, $W3$)对齐，从而解决了时序同步问题，使控制器能够正确进入 $W1$ 阶段。

调试过程中的问题及讨论 在编写顺序模型处理器的代码的过程中，我们的代码采用了完全时序化的思路（所有寄存器的状态均在T3下降沿进行改变）。理论上在对实验板进行CLEAR操作并按下QD按钮之后，CPU会进入 $W1$ 阶段（具体地，在T1上升沿，进入 $W1$ 阶段；在T3下降沿结束 $W1$ 阶段），电路中寄存器的状态应该根据 $W1=1$, $W2=0$, $W3=0$ 进行改变。但经过反复调试，我们发现其状态发生转换的时候， $W1=0$, $W2=0$, $W3=0$ 。为了解决这个问题，我们采用节拍重映射，将($oriW1$, $oriW2$, $oriW3$)映射到($W1$, $W2$, $W3$)，将节拍电位信号向后推迟一个机器周期，实现了正确的控制。除了上述问题，在对执行指令部分代码进行测试的时候，我们发现在最开始取第一条指令的时候存在空拍的问题，执行指令时产生了怪异的现象。经过反复调试，我们发现对 $W1W2W3$ 进行节拍重映射的assign语句未考虑指令执行处的特殊情况，从而产生了错误。为了方便实现执行指令处的状态转移，我们加入了寄存器cnt计数器，从而正确地实现了指令的执行。

5 流水线模型处理器

5.1 需求分析：流水线硬连线控制器设计

核心挑战：流水线 (Pipelining):

- 目标: 将原有的顺序执行模型改造为流水线模型，以提高指令执行的吞吐率。
- 平台: 仍然在EPM7128上实现。
- 实现难点:
 - 冒险 (Hazard):
 - * 这是流水线设计的核心问题。
 - * **结构冒险:** 当两条指令在同一时刻需要访问同一硬件资源时发生（例如，内存）。
 - * **数据冒险:** 当一条指令需要使用到前一条尚未完成计算的指令结果时发生（如 LD R1, (addr) 后紧跟 ADD R2, R1）。
 - * **控制冒险:** 当遇到分支或跳转指令时，不确定下一条要取哪条指令而导致流水线断流。

5.2 概要设计

流水线阶段划分: 对于两个节拍的指令，分为取指阶段和执行阶段；对于三个节拍的指令，分为取指阶段、执行阶段和写回阶段。我们在顺序模型处理器的基础上，把LIR和PCINC这两个控制信号添加到了状态机中合适的位置处，实现了流水线。

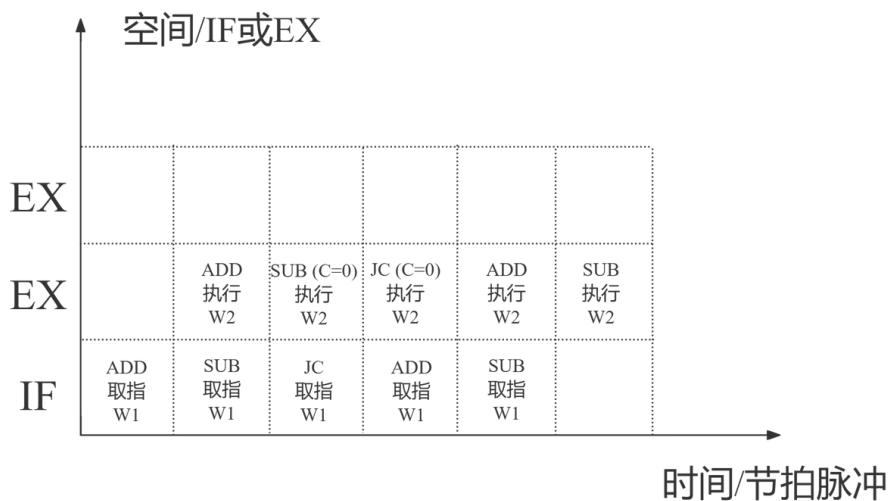


图 10: “取指-执行”并行执行示意图

- 取指 (IF - Instruction Fetch): 从内存获取指令。
- 执行 (EX - Execute): 执行ALU运算等操作，计算地址。

特别地: Load指令有

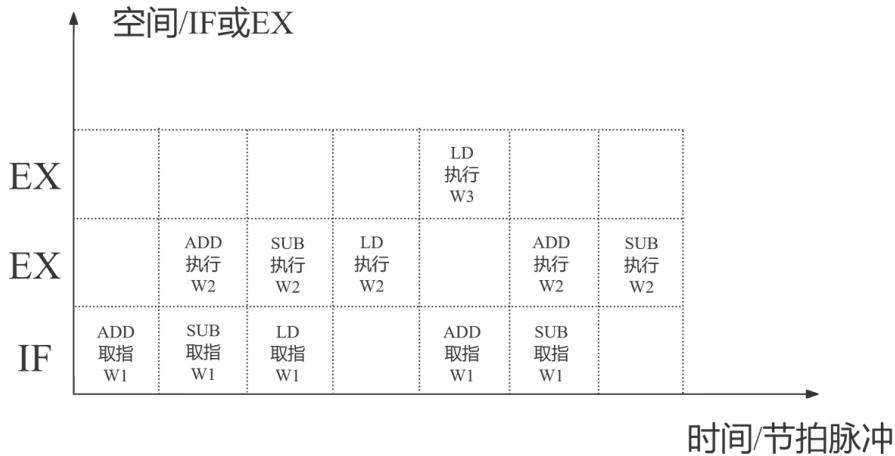


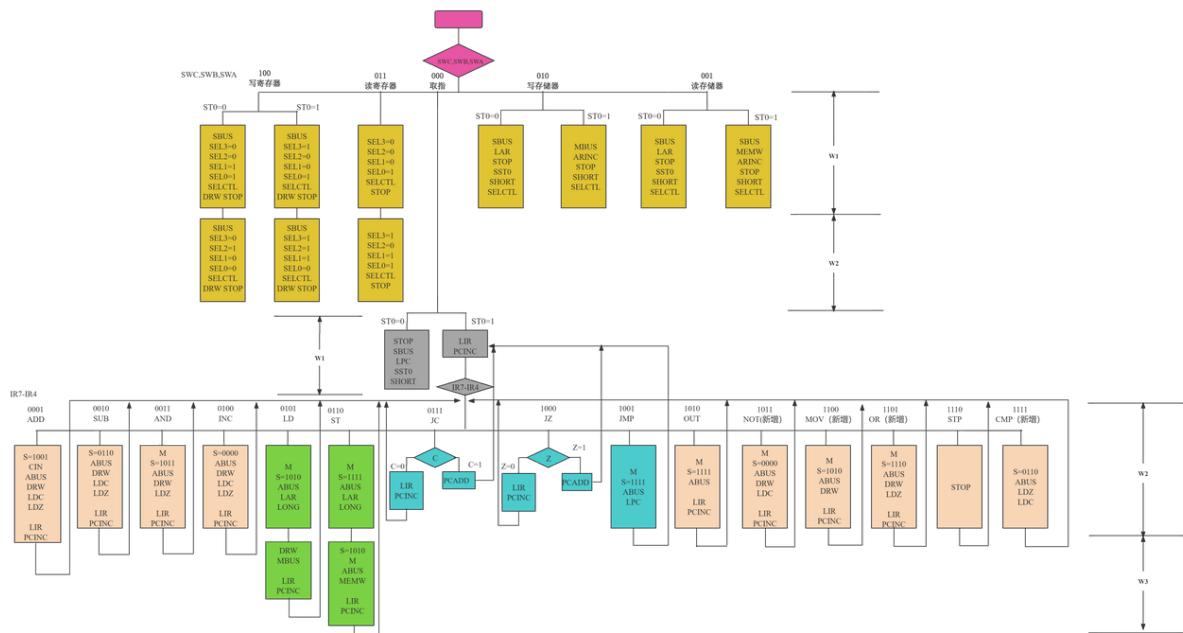
图 11: LD “取指-执行” 并行执行示意图

- 写回 (WB - Write Back): 将结果写回寄存器。

Store指令有

- 访存 (MEM - Memory Access): 内存读写。

流水线控制器状态机设计: 在“取指-执行”模式 ($SWCBA == 3'b000$) 下, 本硬连线控制器摒弃了传统的 $oriW$ 信号驱动的时序机制, 转而采用一个三位计数器 cnt 作为核心状态机。该状态机负责精确控制指令执行的各个流水线阶段, 从而实现处理器指令流的并行处理。通过 cnt 的不同状态, 控制器能够生成对应的控制信号, 驱动数据通路完成取指、译码、执行、访存和写回等操作。这种设计方法通过显式状态序列而非外部信号触发来推进指令周期, 为实现指令流水奠定了基础。



5.3 设计详解

1. 处理器架构与控制方式

- **流水线硬连线控制器:** 这是最核心的需求, 将之前的顺序模型处理器升级为流水线架构。这意味着指令执行将被划分为两个阶段(取指、执行), 不同指令的不同阶段将并行执行, 以提高指令吞吐率。
 - **硬连线控制:** 流水线的控制逻辑仍需通过硬连线方式实现。
 - **全时序逻辑输出:** 所有输出控制信号和流水线寄存器间的中间信号都必须是寄存的, 并在时钟(T_3)的下降沿同步更新。这延续了顺序模型设计中的稳定性要求, 并在流水线中更为关键, 以确保数据在各级之间正确传递, 消除冒险引起的毛刺。

2. 硬件平台适配

- **目标CPLD:** 设计必须能在 Altera CPM7128 上成功实现。
 - **资源考量:** 流水线设计通常会增加逻辑复杂度和触发器数量。在 EPM7128 (128个逻辑宏单元) 这种相对有限的CPLD资源上实现流水线, 将是极大的挑战。这要求设计必须非常精简高效, 合理分配资源。
 - **时钟频率:** 流水线设计通常旨在提高时钟频率, 或在相同频率下提高吞吐量。设计需要确保能满足目标平台的时序要求。
 - **调试支持:** 设计应能通过测试程序的运行进行信号观测, 尤其是在流水线内部的信号, 以调试冒险问题和数据流。

3. 执行指令模式 ($SWCBA=000$) 下的时序控制的改变 由于TEC-8实验箱产生的W时序信号本身存在错误, 为了顺利地实现流水线模型处理器, 我们需要改变执行指令模式下的时序控制, 即使用有限状态自动机进行状态转移, 从而实现时序控制。

4. 冒险问题 存在冒险问题的指令: 经过理论分析, 我们发现可能在流水线中存在冒险问题的指令有如下几种

- **Store指令:** 如果想实现流水线, 需要在该指令的W3阶段中MEMW信号和LIR信号同时有效, 但是如果即将被写入的内存地址恰好是PC指向的内存地址, 就会出现冒险问题。
- **JUMP/JZ($Z=1$)/JC($C=1$)指令:** 如果想实现流水线, 需要在该指令的W2阶段中修改PC的信号(这个信号在JUMP指令中体现为LPC信号, 在JZ($Z=1$)/JC($C=1$)指令中体现为PCADD信号)和LIR信号同时有效。但这显然是不可能的, 我们不可能做到在同一时刻既写PC寄存器又读PC寄存器。

解决方案:

- **Store指令:** 通过与老师的讨论, 我们得知在操作系统(软件层面)中应该禁止用户修改程序区的内容, 这不应该是硬件设计者考虑的问题。
- **JUMP/JZ($Z=1$)/JC($C=1$)指令:** 为了解决这个冒险问题, 只能选择流水线断流, 以保证程序控制流的正确性。

断流示意图:

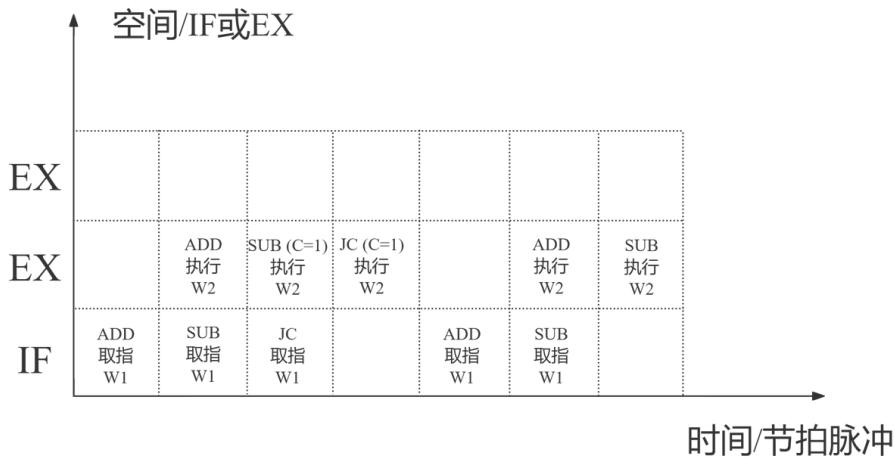


图 13: 流水线断流示意图

5. 流水线控制器状态机设计 在流水线 CPU 设计中, 当 SWCBA = 000 (取指执行模式) 时, 控制器使用时序状态机 cnt 替代 oriW 信号实现流水线控制。状态机包含 7 个状态 (T0-T6), 通过状态转移协调指令执行流程, 关键特性如下:

状态定义 状态机采用 3 位编码, 覆盖指令执行全周期:

```
'define STATE_T0 3'b000 // 初始状态
'define STATE_T1 3'b001 // 用户输入PC值状态
'define STATE_T2 3'b010 // 取指阶段
'define STATE_T3 3'b011 // 执行阶段 (常规指令)
'define STATE_T4 3'b100 // 访存指令准备
'define STATE_T5 3'b101 // 跳转指令执行
'define STATE_T6 3'b110 // 访存指令执行
```

触发条件

- 时钟: STATE_T3 下降沿触发状态转移
- 复位: CLR 低电平异步复位至 T0
- 指令类型: IRH 操作码决定分支路径
- 标志位: C/Z 影响条件跳转状态

核心功能

- 动态适配不同指令类型的执行周期:
 - 单周期指令: STATE_T0 → STATE_T1 → STATE_T2 → STATE_T3
 - 访存指令 (LD/ST): STATE_T2 → STATE_T4 → STATE_T6
 - 跳转指令 (JMP/JC/JZ): STATE_T2 → STATE_T5 → STATE_T2

状态转移逻辑 状态转移在 T3 下降沿更新, 逻辑如下表所示:

当前状态	下一状态条件	下一状态	适用指令类型
STATE_T0	无条件	STATE_T1	所有指令
STATE_T1	无条件	STATE_T2	所有指令
STATE_T2	IRH = LD/ST	STATE_T4	访存指令
	IRH = JMP/JC(C=1)/JZ(Z=1)	STATE_T5	跳转指令
	其他指令	STATE_T3	算术/逻辑指令
STATE_T3	IRH = LD/ST	STATE_T4	访存指令
	IRH = JMP/JC(C=1)/JZ(Z=1)	STATE_T5	跳转指令
	其他指令	STATE_T3	其他普通指令
STATE_T4	无条件	STATE_T6	访存指令第二阶段
STATE_T5	无条件	STATE_T2	跳转后重新开始取指周期
STATE_T6	IRH = LD/ST	STATE_T4	连续访存指令
	IRH = JMP/JC(C=1)/JZ(Z=1)	STATE_T5	访存后跳转
	其他指令	STATE_T3	访存后执行算术操作

状态转移图如下：

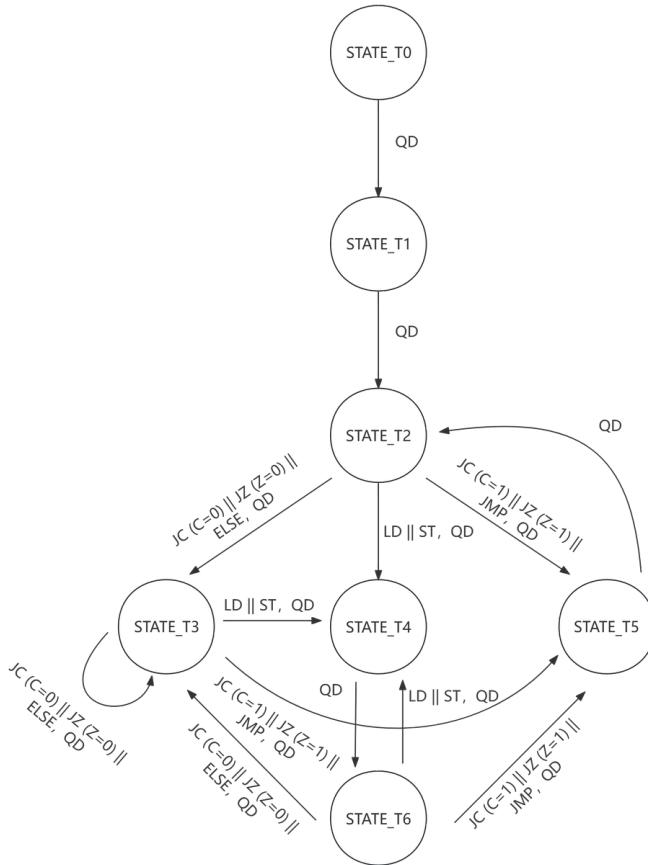


图 14: 状态转移图

关键代码实现：

```
always @ (negedge T3 or negedge CLR) begin
```

```

if (!CLR) cnt <= 'STATE_T0; // 异步复位
else case(cnt)
    'STATE_T0: cnt <= 'STATE_T1;
    'STATE_T1: cnt <= 'STATE_T2;
    'STATE_T2: cnt <= (IRH == 'OP_LD || IRH == 'OP_ST) ? 'STATE_T4 :
        ((IRH == 'OP_JC && C) ||
        (IRH == 'OP_JZ && Z) ||
        IRH == 'OP JMP) ? 'STATE_T5 : 'STATE_T3;
    'STATE_T3: cnt <= ... // 同T2逻辑
    'STATE_T4: cnt <= 'STATE_T6;
    'STATE_T5: cnt <= 'STATE_T2;
    'STATE_T6: cnt <= ... // 同T2逻辑
endcase
end

```

状态机与流水线协同 状态机通过生成 周期标识信号 驱动控制信号:

- W1 = (cnt == STATE_T0 || STATE_T1 || STATE_T5): 取指/跳转阶段
- W2 = (cnt == STATE_T2 || STATE_T3 || STATE_T6): 译码/执行阶段
- W3 = (cnt == STATE_T4): 访存专用阶段

控制信号生成示例 (以 LPC 为例):

```

assign LPC_next =
    (fetch_exec_mode && !ST0 && W1) ||           // 取指阶段加载PC
    (fetch_exec_mode && ST0 && IRH == 'OP JMP && W2); // JMP指令执行

```

动态流水线控制状态机根据指令类型动态调整执行路径:

- 算术指令: STATE_T2 → STATE_T3 → STATE_T3
- 跳转指令: STATE_T2 → STATE_T5 → STATE_T2
- 访存指令: STATE_T2 → STATE_T4 → STATE_T6

关键设计考量

- 跳转指令优化: STATE_T5 → STATE_T2 直接返回译码状态, 避免冗余取指周期
- 状态自循环: STATE_T3 状态支持多周期算术指令 (如移位/乘除)
- 异步复位: 确保上电后强制进入 STATE_T0 初始状态

设计调试小结 在流水线设计的过程中, 我们遇到了一个很大的问题: Load IR 这个操作会在T3上升沿进行, 导致我们在T3下降沿进行非阻塞赋值时会出现逻辑上的错误。通过code review与搜集资料, 我们终于考虑到了Load IR 可能并不在T3下降沿进行的这个问题。于是我们修改自动机的状态定义并解决了这个问题。

调试过程中的问题及讨论 我们发现在有流水线的情况下, 在INC指令之后紧接着执行ST指令会直接进入ST指令的W3阶段 (理应进入ST指令的W2阶段), 导致执行出错。我们怀疑问题在于: 在T3下

降沿到来之前Load IR操作已经执行，导致在T3下降沿的时候，IR寄存器中存储的是下一条指令，而我们编写这段代码的时候的逻辑为IRH是Load IR操作执行之前的IRH，导致生成的W1W2W3信号错误。于是我们修改了自动机的状态定义（增加了两种状态），令W只依赖于自动机状态而不依赖于输入信号（如IRH），从而正确地实现了流水线模型控制器。

6 测试程序

6.1 测试目的

我们为了测试CPU的实际性能、稳定性与鲁棒性，设计了这个基于树状数组（Fenwick Tree，可以实现 $O(\log N)$ 的单点修改与 $O(\log N)$ 的区间查询）的测试程序。本测试程序的设计核心在于从用户真实应用场景出发，通过实现树状数组这一高效数据结构的核心操作，构建贴近实际使用情境的测试环境，这体现了我们“服务用户”的设计思想。

6.2 测试程序设计

1. 测试程序的选定 与传统仅通过堆砌孤立的指令序列来验证处理器各指令功能的测试方法不同，本次控制器验证采取了更高层级的系统级测试策略。我们并没有简单地堆砌指令片段，而是已经实现了一个具备特定功能（树状数组的单点修改与区间查询功能）的完整汇编程序作为核心测试用例。我们选择树状数组（Fenwick Tree / Binary Indexed Tree）的实现作为核心测试程序，是基于以下几个关键考量，它完美契合了验证我们硬连线控制器的需求：

全面的指令覆盖：

- **ALU 运算：** 树状数组的核心操作，如 ADD/SUB 和 SUM 中，涉及大量的算术和逻辑运算。例如，计算最低有效位（lowbit）的经典表达式 $i \& (-i)$ ，在汇编层面就能覆盖 NOT, INC (用于计算 $-i$ ，即 $\sim i + 1$)，以及 AND, ADD (用于 $i += \text{lowbit}$ 或 $i -= \text{lowbit}$) 这四条关键指令。此外，还涉及加法、减法、比较等多种 ALU 指令，这些操作能够充分验证控制器对各种 ALU 指令的正确控制。
- **数据传输 (MOV)：** 在处理数组元素和中间变量时，会频繁进行寄存器之间的数据传输，从而验证 MOV 指令及相关数据通路控制。
- **内存访问 (LD/ST)：** 树状数组的核心在于对内存中数组元素的读写。`update` 操作涉及写入内存，`query` 操作涉及读取内存。这能够充分且真实地验证控制器对 LD 和 ST 这类多周期指令的精确控制，包括其在 STATE_T4 (送地址) 和 STATE_T6 (数据传输) 阶段的正确流转和信号生成。
- **条件分支 (JC/JZ)：** 树状数组的 ADD/SUB 和 SUM 循环中，需要判断是否达到数组边界或循环条件，这必然会使用到比较指令和条件跳转指令 (JC/JZ)，从而验证控制器根据 C/Z 标志进行正确分支的能力。

复杂控制流的真实性： 树状数组的 `update` 和 `query` 函数都包含循环和条件判断，这些复杂的控制流模式能够真实模拟处理器在执行实际程序时频繁的跳转和分支行为。这对于验证控制器 STATE_T5 阶段（分支执行，断流）的正确性至关重要，确保在控制冒险发生时，流水线能够被正确地刷新和重定向。

2. 测试程序的详细设计

2.1 程序概述 该测试程序由地址 125 开始，直至 197 处的 STOP 指令结束。它忠实地实现了 C 语言代码中的树状数组操作，利用微处理器的通用寄存器 R0, R1, R2, R3 完成数据处理、循环控制和地址计算。程序初始化与入口：

- **程序起始：** 控制器通过外部 QD 机制等方式，将程序计数器 (PC) 加载为程序入口地址 125。

- 内部跳转机制：程序首先执行 OR R0, R2 和 JMP [R0]。根据程序开始时 R0=0111 0000 (112) 和 R2=0000 1111 (15) 的初始寄存器值，OR R0, R2 将使 R0 变为 0111 1111 (127)。随后 JMP [R0] 指令将程序计数器跳转到地址 127，这是实际功能逻辑的起点，实现了程序内部的精确跳转。

核心功能流程：

- 第一次更新循环 (ADD(1, 3) 逻辑)：对 tree 数组的指定索引范围（从 1 开始，每次递增 lowbit）进行更新操作（累加 3）。
- 第一次查询循环 (sum(2) 逻辑)：对 tree 数组的索引 2 进行前缀和查询（模拟 query(2)），并将结果输出。
- 第二次更新循环 (SUB(1, 5) 逻辑)：对 tree 数组的指定索引范围进行更新操作（累减 5）。
- 第二次查询循环 (sum(2) 逻辑)：对 tree 数组的索引 2 进行第二次前缀和查询，并将结果输出。
- 程序终止：STOP 指令结束程序执行。

2.2 指令覆盖与控制器行为验证点 该汇编代码的执行流程精确地验证了控制器在处理各种指令类型、多周期操作和复杂控制流时的行为：

程序初始化与启动 (地址 125-126)：

- 初始状态：tree 数组：[1, 1, 0, 1, 0]，R0=0111 0000，R1=0000 0101，R2=0000 1111
- 125：OR R0, R2：验证 OR 逻辑运算指令的正确性。此步计算跳转目标地址。
- 126：JMP [R0]：验证无条件跳转指令的正确性。

第一次更新循环 (地址 127-143)：

- 寄存器初始化 (地址 127-132)：

- 127: SUB R0, R0 (R0=0)
- 128: INC R0 (R0=1)
- 129-132: SUB R2, R2; INC R2; INC R2; INC R2 (R2=3)

这组指令验证了 SUB 和连续的 INC 指令组合来实现寄存器清零和加载小立即数的功能，确保控制器对这些基本算术指令的正确控制。

- 内存访问核心 (地址 133-135)：

- 133: LD R3, [R0]: LD 指令。控制器将经历 STATE_T2 (取指), STATE_T4 (送地址), STATE_T6 (数据读取) 的多周期流转。
- 134: ADD R3, R2: ADD 指令。在 STATE_T3 (执行) 阶段完成算术运算。
- 135: ST R3, [R0]: ST 指令。控制器再次经历 STATE_T2, STATE_T4 (送地址), STATE_T6 (数据写入) 的多周期流转。

这些指令序列是验证控制器多周期内存操作和 ALU 运算协调的关键。

- lowbit 计算与循环控制 (地址 137-143):

- 137: MOV R3, R0: MOV 指令, 验证寄存器之间的数据传输。
- 138: NOT R3: NOT 指令, 验证位反转操作。
- 139: INC R3: INC 指令, 验证自增操作。
- 140: AND R3 R0: AND 指令, 验证位与操作。

这四条指令明确地测试了控制器对 NOT, INC, AND 等逻辑和算术指令的精细控制, 并且它们是 $i \& (-i)$ 逻辑的实现, 展示了复杂逻辑的指令组合。

- 141: ADD R0, R3: 验证 ADD 指令, 更新循环变量。
- 142: CMP R1, R0: CMP 指令, 比较 R1 (N=5) 和 R0。此指令会更新条件标志 C 和 Z。
- 143: JC -8 和 136: JC -4: JC (Jump if Carry) 条件跳转指令, 判断 $R0 \leq R1$ 的循环条件。这里由于硬件限制, JC 与 JZ 偏移量位数不够, 我们不得不采用“连跳”来实现真正的跳转。如果条件满足, PC 将跳转到 133 (即 LD R3, [R0])。这验证了控制器根据 C 标志位进行分支判断的能力, 并触发 STATE_T5 阶段的断流和 PC 更新。

第一次查询循环 (地址 144-159):

- 寄存器初始化 (地址 144-147): 类似更新循环, 初始化 R0 为 2, R2 为 0。
- 内存访问与求和 (地址 148-149):
 - 148: LD R3, [R0]: 再次验证 LD 指令的多周期执行。
 - 149: ADD R2, R3: 验证 ADD 指令, 累加查询结果。
- lowbit 计算 (地址 150-154): 再次执行 MOV, NOT, INC, AND, 验证这些指令的重复执行稳定性。
- 循环控制与特殊分支 (地址 155-159):
 - 155: SUB R0, R3: SUB 指令, 更新循环变量 R0。
 - 156: AND R0, R0: 自身与操作, 用于清除 Z 标志位, 或确保其值不为 0。
 - 157: JZ +2: JZ (Jump if Zero) 条件跳转指令。这部分用于 R0 变为 0 时跳出循环。
 - 158: SUB R3, R3: 清除 R3。
 - 159: JZ -8 和 152: JZ -5: JZ (Jump if Zero) 条件跳转指令。此为循环的主跳, 仍然采用“连跳”的方式, 当 R0 不为 0 时, 会跳转到 148 (LD R3, [R0]) 继续循环。

这些复杂的 JZ 组合和相对跳转, 是验证控制器在处理零标志位和复杂分支逻辑时的关键点, 确保 STATE_T5 的正确性和 PC 更新的准确性。

输出结果 (地址 160):

- 160: OUT-A R2: OUT-A 指令。验证控制器发出外部输出信号的能力, 用于将计算结果 R2 输出到外部。

第二次更新循环 (地址 161-179):

- 指令序列与第一次更新循环类似，但 R2 初始值为 5（通过 SUB R2, R2 和 5 个 INC R2），以及使用 SUB R3, R2 (地址 170) 而不是 ADD R3, R2。这验证了控制器对 SUB 指令的正确执行，以及在不同数据和操作下的循环稳定性。

第二次查询循环 (地址 180-195):

- 指令序列与第一次查询循环完全相同，用于验证控制器在再次执行相同复杂逻辑时的正确性和一致性。

最终输出结果 (地址 196):

- 196: OUT-A R2: 再次验证输出功能。

程序终止 (地址 197):

- 197: STOP: STOP 指令。验证控制器能够正确进入停止状态，结束程序的执行。

2.3 预期输出与验证点 该测试程序在控制器上运行的预期行为如下：

- 初始 tree 数组状态: tree = [1, 1, 0, 1, 0] (地址 1-5)

程序执行开始 (地址 125):

- OR R0, R2 (R0=112, R2=15) -> R0 = 127
- JMP [R0] -> PC 跳转至 127

第一次更新循环 (对应 ADD(1, 3) 逻辑):

- R0 从 1 开始，经过 lowbit 步进，对 tree[1], tree[2], tree[4] 进行 +3 操作。
- 内存 tree 数组最终状态 (第一次更新后): [4, 4, 0, 4, 0]

第一次查询循环 (模拟 query(index=2) 逻辑):

- 程序计算 tree[2] (即 tree[2] + tree[0] 在 BIT 概念中)。
- 预期第一次 OUT-A R2 输出: 4

第二次更新循环 (对应 SUB(1, 5) 逻辑):

- R0 从 1 开始，对 tree[1], tree[2], tree[4] 进行 -5 操作。
- 内存 tree 数组最终状态 (第二次更新后): [-1, -1, 0, -1, 0]

第二次查询循环 (模拟 query(index=2) 逻辑):

- 程序计算 tree[2] (即 tree[2] + tree[0] 在 BIT 概念中)。
- 预期第二次 OUT-A R2 输出: -1

验证点:

- 最终输出值:** 输出的 OUT-A R2 结果是否依次为 4 和 -1。这是程序功能正确性的核心验证。
- 内存/寄存器状态:** 在关键时刻 (如每个循环结束后，或特定指令执行后)，检查模拟内存中 tree 数组的内容和通用寄存器 R0-R3 的值是否与上述预期状态精确一致。

- **控制器状态流转:** 观察仿真波形中 cnt 状态机在执行每条指令时，是否正确地按照预定的多周期路径流转：
 - LD/ST 指令：确认 STATE_T2 → STATE_T4 → STATE_T6 序列的准确执行。
 - 跳转指令 (JMP, JC, JZ)：确认 STATE_T5 阶段的触发、PC 的正确更新以及可能的流水线“断流”行为。
 - 其他 ALU/MOV/NOT/INC/AND 指令：确认在 STATE_T3 等执行阶段的正确控制信号生成。
- **控制信号精确性:** 验证在程序执行过程中，所有控制信号（如 DRW, PCINC, LPC, LAR, MEMW, LIR, LDZ, LDC, CIN, S, M, ABUS, SBUS, MBUS, SHORT, LONG, SEL0-3, STOP）是否在正确的时钟周期内精确地活跃，且没有不期望的毛刺或冲突。

通过运行这段精确翻译的树状数组汇编代码，我们能够对控制器的所有指令类型、多周期操作、条件分支、循环控制以及整体协调能力进行全面、深入且高度真实的验证。

6.3 测试结果与分析

内存中的树状数组初始为 [1, 1, 0, 1, 0]（该树状数组表示的数组为 [1, 0, 0, 0, 0]），在进行了 ADD(1, 3) 操作之后，该树状数组表示的数组变为 [4, 0, 0, 0, 0]，再进行 SUB(1, 5) 操作后，该树状数组表示的数组变为 [-1, 0, 0, 0, 0]（该数组对应的树状数组为 [-1, -1, 0, -1, 0]）。注：测试程序里树状数组在内存中的存储地址为1到5

单拍测试 将寄存器初始值、内存初始值、测试程序写入实验板后，我们进行了单拍测试。我们一次次地按下 QD 按钮，一次次地对比 IR 寄存器中的数据是否符合预期，我们发现程序控制流完全正确。

测试结果 经过实测，我们发现程序执行（无论是单拍执行还是多拍执行）之后在内存单元1至5中的内容依次为 -1, -1, 0, -1, 0，这与我们的预期是一致的。

测试程序优点 我们设计的测试程序显著区别于简单的指令序列堆砌，它是一个具备实际功能、内含实际算法与数据结构的程序，这赋予了它独特的优点：首先，该测试程序的核心在于实现了树状数组（Fenwick Tree）这一高效的数据结构的修改与查询操作。树状数组的应用场景广泛，例如在数据分析、实时统计或算法竞赛中，它能够高效地处理区间查询和单点更新问题。这意味着我们的测试不再是孤立地验证单条指令的功能，而是从一个真实的计算任务出发，全面考察处理器在处理复杂逻辑和数据流时的表现。其次，通过实现树状数组，我们的程序必然涉及到控制流的转移。例如，树状数组的索引计算会产生多种算术逻辑运算；其迭代的特性会使用循环语句（利用到了跳转指令）；而对内存中数组元素的读写则会频繁触发数据传输和访存操作。这种复杂的指令组合并非随机生成，而是由实际算法逻辑驱动，能够更真实地模拟CPU在运行实际软件时所面临的压力和挑战。最后，作为一个有实际功能、既有广度又有深度的程序，它能够帮助我们从用户视角评估处理器性能。我们不仅能验证每一条指令的正确性，更能直观地看到处理器在完成一个有意义的计算任务时的效率和稳定性。这远比仅仅堆砌加法或逻辑门指令来得有意义，因为它直接反映了我们的CPU设计在服务实际应用、解决实际问题方面的能力。这种贴近真实应用场景的测试方法，为处理器的优化和改进提供了宝贵且有针对性的反馈。值得一提的是，我们在编写测试程序的过程中遇到了诸多问题（具体可见调试日志）。面对硬件资源的缺陷，我们使用软件的方法全都解决了，在这个过程中我们对“用软件解决硬件缺陷”这个思想有了更深入的理解与认识，明白了通过软件层面的巧妙设计和弥补，可以规避、掩盖、优化甚至修复硬件层面存在的限制、不足或潜在问题，从而提升系统整体的性能、稳定性、可靠性或功能性。综上，我们不仅对自己设计的硬布线控制器进

行了测试，还有了对计算机系统更深入的理解，而这些都是使用“指令堆砌”式测试程序进行测试收获不到的。

6.4 测试程序及对应的C语言代码

测试程序对应的C语言代码：

```
#include <stdio.h>
#define OUT(x) printf("%d\n", x)

#define N 5
int tree[N + 1] = {0, 1, 1, 0, 1, 0};
int R0, R1 = N, R2, R3;

int main(void)
{
    // 循环体中起始的两行被注释掉的代码表示这个循环体内的操作内容

    // ADD(1, 3)
    R0 = 1;
    R2 = 3;
    while (R0 <= R1)
    {
        // tree[R0] += R2;
        // R0 += R0 & -R0;
        R3 = tree[R0];
        R3 = R3 + R2;
        tree[R0] = R3;
        R3 = R0;
        R3 = ~R3;
        R3++;
        R3 = R3 & R0;
        R0 = R0 + R3;
    }

    // SUM(2)
    R0 = 2;
    R2 = 0;
    while (R0 > 0)
    {
        // R2 += tree[R0];
        // R0 -= R0 & -R0;
        R3 = tree[R0];
```

```

R2 = R2 + R3;
R3 = R0;
R3 = ~R3;
R3++;
R3 = R3 & R0;
R0 = R0 - R3;
}

OUT(R2);

// SUB(1, 5)
R0 = 1;
R2 = 5;
while (R0 <= R1)
{
    // tree[R0] -= R2;
    // R0 += R0 & -R0;
    R3 = tree[R0];
    R3 = R3 - R2;
    tree[R0] = R3;
    R3 = R0;
    R3 = ~R3;
    R3++;
    R3 = R3 & R0;
    R0 = R0 + R3;
}

// SUM(2)
R0 = 2;
R2 = 0;
while (R0 > 0)
{
    // R2 += tree[R0];
    // R0 -= R0 & -R0;
    R3 = tree[R0];
    R2 = R2 + R3;
    R3 = R0;
    R3 = ~R3;
    R3++;
    R3 = R3 & R0;
    R0 = R0 - R3;
}

OUT(R2);

```

```

    return 0;
}

```

测试程序：

指令区	十进制地址	十六进制地址	指令	二进制机器码解释
ADD(1, 3)	125	7D	OR R0, R2	1101 0010
	126	7E	JMP [R0]	1001 0000
	127	7F	SUB R0, R0	0010 0000
	128	80	INC R0	0100 0000
	129	81	SUB R2, R2	0010 1010
	130	82	INC R2	0100 1000
	131	83	INC R2	0100 1000
	132	84	INC R2	0100 1000
	133	85	LD R3, [R0]	0101 1100
	134	86	ADD R3, R2	0001 1110
	135	87	ST R3, [R0]	0110 0011
	136	88	JC -4	0111 1100 与地址143处的JC指令形成接力
	137	89	MOV R3, R0	1100 1100
	138	8A	NOT R3	1011 1100
SUM(2)	139	8B	INC R3	0100 1100
	140	8C	AND R3 R0	0011 1100
	141	8D	ADD R0, R3	0001 0011
	142	8E	CMP R1, R0	1111 0100
	143	8F	JC -8	0111 1000
	144	90	SUB R0, R0	0010 0000
	145	91	INC R0	0100 0000
	146	92	INC R0	0100 0000
	147	93	SUB R2, R2	0010 1010
	148	94	LD R3, [R0]	0101 1100
	149	95	ADD R2, R3	0001 1011
	150	96	MOV R3, R0	1100 1100
	151	97	NOT R3	1011 1100
	152	98	JZ -5	1000 1011 与地址159处的JZ指令形成接力
	153	99	INC R3	0100 1100
	154	9A	AND R3, R0	0011 1100
	155	9B	SUB R0, R3	0010 0011
	156	9C	AND R0, R0	0001 0000
	157	9D	JZ +2	1000 0010
	158	9E	SUB R3, R3	0010 1111
	159	9F	JZ -8	1000 1000

续下一页

续表：测试程序 – 第 35 页

指令区	十进制地址	十六进制地址	指令	二进制机器码解释
SUB(1, 5)	160	A0	OUT-A R2	1010 1000
	161	A1	SUB R0, R0	0010 0000
	162	A2	INC R0	0100 0000
	163	A3	SUB R2, R2	0010 1010
	164	A4	INC R2	0100 1000
	165	A5	INC R2	0100 1000
	166	A6	INC R2	0100 1000
	167	A7	INC R2	0100 1000
	168	A8	INC R2	0100 1000
	169	A9	LD R3, [R0]	0101 1100
	170	AA	SUB R3, R2	0010 1110
	171	AB	ST R3, [R0]	0110 0011
	172	AC	JC -4	0111 1100 与地址179处的JC指令形成接力
	173	AD	MOV R3, R0	1100 1100
	174	AE	NOT R3	1011 1100
	175	AF	INC R3	0100 1100
	176	B0	AND R3 R0	0011 1100
SUM(2)	177	B1	ADD R0, R3	0001 0011
	178	B2	CMP R1, R0	1111 0100
	179	B3	JC -8	0111 1000
	180	B4	SUB R0, R0	0010 0000
	181	B5	INC R0	0100 0000
	182	B6	INC R0	0100 0000
	183	B7	SUB R2, R2	0010 1010
	184	B8	LD R3, [R0]	0101 1100
	185	B9	ADD R2, R3	0001 1011
	186	BA	MOV R3, R0	1100 1100
	187	BB	NOT R3	1011 1100
	188	BC	JZ -5	1000 1011 与地址195处的JZ指令形成接力
	189	BD	INC R3	0100 1100
	190	BE	AND R3, R0	0011 1100
	191	BF	SUB R0, R3	0010 0011
	192	C0	AND R0, R0	0001 0000
	193	C1	JZ +2	1000 0010
	194	C2	SUB R3, R3	0010 1111
	195	C3	JZ -8	1000 1000
	196	C4	OUT-A R2	1010 1000

197	C5	STOP	1110 0000
表格结束			

十进制地址	十六进制地址	二进制机器码
0	0	任意
1	1	0000 0001
2	2	0000 0001
3	3	0000 0000
4	4	0000 0001
5	5	0000 0000

寄存器	二进制机器码
R0	0111 0000
R1	0000 0101
R2	0000 1111
R3	任意

7 附件1 各成员心得总结

7.1 苏泽勤

在设计初期，我们尝试使用行为级建模来描述控制器逻辑。然而，在实际开发过程中，我们很快发现行为级描述在表达复杂硬件控制时，其时序关系不够直观，且难以精准控制输出信号的瞬态行为，容易引入毛刺。面对这一挑战，我们果断调整了设计策略，转而采用了更为适合硬连线控制器实现的数据流建模（Dataflow Modeling）方法。

正是这一转变，成为本次设计中的一个关键亮点。通过采用`assign`语句清晰地描述信号之间的组合逻辑关系，并结合`always`块中的非阻塞赋值（`<=`）实现时序逻辑和寄存器更新，我们成功地将组合逻辑与时序逻辑分离。这种建模方式使得处理器内部的数据流动路径和控制信号的生成逻辑变得异常清晰、易于理解和调试。我们能够直观地追踪数据如何在不同的功能单元（如ALU、寄存器文件、内存）之间传递，以及控制信号是如何在特定时钟周期精准地驱动这些数据流动的。这种设计范式不仅有效消除了输出毛刺，显著提升了设计的稳定性和可预测性，也加深了我们对硬件描述语言本质的理解。我认识到了在数字硬件设计中，选择合适的建模方式对于设计质量、调试效率的重要性。

编写好代码后，我们最初只进行了对单条指令的测试，发现指令均正确。我们又从用户的角度出发编写了一个有实际应用的程序——对一个数组中的元素进行高效的修改与查询，这加强了我从用户角度出发进行程序设计的思维。在测试程序编写的过程中，遇到了诸多硬件资源短缺的问题（具体体现在只能使用四个寄存器、JC/JZ指令的偏移量只有四位）。但我们使用`SUB`指令和`INC`指令联合对寄存器赋值、JC/JZ接力跳等方式解决了这些困难。硬件资源的短缺促使我们编译出对硬件要求更低的汇编代码。这是我第一次在硬件资源短缺的情况下进行编程，我对在资源受限环境中进行高效开发的理念有了更深刻的理解。

7.2 冯仁宇

在实现顺序控制器时，我们最初使用了`always@()`来描述组合逻辑电路，经过测试与讨论，我们发现这种设计会出现毛刺，可能出现不可知的错误。为了改正这个错误，我们又尝试了纯行为级描述，在T3下降沿给控制信号赋值，但是这种设计在赋初值时会出现“线与”问题，而且代码较为臃肿。最后我们采用数据流建模与行为级建模相结合的方法，完美地解决了问题，这提高了我对Verilog语言的并行性以及时序逻辑的理解，提高了我行为级建模与数据流建模的能力。在实现流水型控制器时，我们对可能出现的冒险冲突进行了详细的分析，例如JZ（Z=1）时，跳转指令会给PC赋值，而此时若将LIR置1，会出现写后读的数据冒险，导致错误发生，因此这里必须断流。这加强了我对流水线原理的理解以及对冒险问题的分析能力。在调试过程中，我们还遇到了控制信号给出的时机错误的问题，原因是我们在对时序节拍的理解出了问题，例如在LIR为1时按下QD，则PC指向的指令会在T3上升沿打入IR寄存器，而在T3下降沿才会修改控制信号。在修正逻辑后，我们的控制器成功地执行了所有功能。这提高了我对控制器原理以及计算机时序节拍的理解。在编写最终测试程序时，我们摒弃了堆砌指令的做法，而是从用户角度出发，实现了一个具有实际功能的C程序（树状数组的单点修改与区间查询功能），并将C代码人工转换为可执行的机器代码来测试硬件运行真正的程序的稳定性与正确性。在这个过程中，我们遇到了硬件资源短缺的问题。例如，寄存器资源宝贵、需要对寄存器赋值、JC/JZ指令偏移量只有4位等。这促使我们从软件层面来解决硬件的缺陷，我们使用`SUB+INC`指令来实现寄存器的赋值，使用连续的JZ/JC来实现跳转功能。通过这次测试程序的编写，我提高了对软硬件之间的交互的理解，硬件的缺陷需要高层的软件来解决，我还认识到了寄存器资源的宝贵以及设计者对于性能和成本之间的权衡考虑。

7.3 张启涵

在本次CPU设计项目中，我最重要的收获并非仅仅是实现了一系列指令，而是通过构建控制器这一“CPU的中枢神经系统”，深刻领悟了计算机体系结构的精髓所在。设计之初，我们面临一个核心题：如何用硬件描述语言（Verilog HDL）去描绘一个既有瞬时响应又有记忆功能的复杂系统？这引导我们最终采用了一种将数据流建模与行为级建模精妙结合的架构，这次实践让我对计算机的理解从抽象的框图真正走向了具体的逻辑实现。与纯软件编程不同，硬件设计必须时刻清晰地划分“什么事”和“什么时候发生”。我们最终的设计范式，其亮点在于将这两者完全解耦：

- **组合逻辑（数据流建模）：** 我们使用大量的 `assign` 语句来构建控制器的组合逻辑部分。我将其理解为CPU的“瞬时决策网络”。这个网络根据当前的指令（IRH）、状态标志（C, Z）和时序状态（cnt），立即、无条件地计算出下一步所有可能需要的控制信号（如`DRW_next`, `S_next`等）。这部分代码是并行的、无状态的，它完美地映射了硬件电路中由逻辑门构成的、信号传播只受门延迟影响的物理现实。它回答了“在当前状况下，应该做什么？”的问题。
- **时序逻辑（行为级建模）：** 我们使用一个核心的 `always @(negedge T3 or negedge CLR)` 块来处理所有状态的更新。我将其视为系统的“节拍器”和“状态锁存器”。仅在时钟信号的有效边沿，这个模块才会将组合逻辑计算出的下一状态值（`_next` 信号）同步地、非阻塞地（`<=`）“拍入”到实际的输出寄存器（如`DRW`, `S`）和状态寄存器（`cnt`, `ST0`）中。这确保了所有信号的同步更新，从根本上消除了组合逻辑路径不同而可能产生的毛刺和竞争冒险，保证了系统的稳定可预测。它回答了“在哪个精确时刻，去更新状态？”的问题。

这种“计算与锁存分离”的设计哲学，使我们设计的控制器拥有了清晰的数据流和精准的时序控制。整个CPU的宏观架构——数据通路（Datapath）和控制器（Control Unit）——在我们的代码中得到了完美的体现：`assign` 语句定义了控制信号如何流向数据通路，而 `always` 块则驱动着控制器状态机在每个时钟周期中稳定地跃迁。这次实践，也让我对《计算机组成原理》中的诸多概念有了更为深刻的认识：

- **硬件与软件的接口：** 在资源受限（寄存器少、跳转偏移短）的情况下编写测试程序时，我深刻体会到了“硬件定义边界，软件在边界内创造”的理念。我们通过“SUB+INC”组合指令模拟直接数加载，通过“JC/JZ接力跳”实现大范围跳转，这些软件层面的“巧计”，本质上是在弥补硬件为控制成本和复杂度而做出的妥协。这让我认识到，计算机系统是一个软硬件协同的整体，其性能和功能是二者共同作用的结果，而不仅仅是硬件的堆砌。

总而言之，这次控制器设计不仅是一次编码练习，更是一次对计算机系统从底层逻辑到顶层架构的深度探索。通过亲手搭建组合逻辑与时序逻辑相结合的控制器架构，我真正理解了冯·诺依曼结构中“控制器”的实现机理，也对性能与成本、硬件与软件之间的制衡与协同关系有了更为现实和深刻的感悟。

7.4 汪顺

硬布线控制器的设计核心在于通过硬件逻辑电路产生控制信号，这要求对指令系统和 CPU 的时序有着清晰的理解。时序图在其中扮演了至关重要的角色，它清晰展示了各个信号在时间上的先后顺序和相互关系。设计初期，我们发现实验箱提供的原始信号时序与预想不一致：`CLR` 之后第一次按 `QD`，经历 `T3` 下降沿后 `W1` 才变为 1，导致用时序生成的控制信号延后。通过老师的讲解和团队的帮助，我画出了适配实际工程的控制信号时序图。流程图的绘制也是设计中的重要一环。流程图能将控制器的工作过程以直观的图形方式呈现，把复杂的控制逻辑分解成一个个有序的步骤。

设计初期，我参考老师提供的流程图搭建了基本框架和内容，但扩展的指令（NOT、MOV、OR、CMP）需要结合实际程序设计才能填写正确的控制信号。这一过程让我深刻体会到，流程图是连接抽象指令功能与具体硬件实现的桥梁，能帮助我们更清晰地把握整体设计思路，避免逻辑混乱。除此之外，时空图能直观展示流水型控制器的特点及效率优势，解释控制冲突的解决方法；译码表清晰显示了控制信号与指令操作之间的对应关系；团队设计的有限状态自动机是控制器的核心，能让人快速理解控制器的运行过程。绘制相关图示表格时，团队成员的协作让整个过程事半功倍。我们结合所学知识与工程实际，用图表直观展示设计思想。在大家的共同努力下，绘制的图示表格不仅准确反映了控制逻辑，还能与实验箱的实际运行状态对应，为代码编写和调试提供了可靠依据。这种团队合作的经历让我明白，在未来的工程实践中，与优秀队友协作是完成复杂任务、提升自身能力的重要途径。

8 附件2 工作日志

8.1 6月30日

8.1.1 今日进展

进行需求分析，了解要完成的任务。复习曾经在数字逻辑实验课学习过的下载程序到TEC-8实验板的方法。学习时序发生器产生的节拍电位信号与节拍脉冲之间的关系，为后续工作打下基础。编写了第一版基础代码，实现了写寄存器、读寄存器、写存储器、读存储器四种基本的控制台操作。

8.1.2 设计思路

我们确定了整个程序的时序逻辑框架（即，使用`assign`语句赋值的数据流建模给出所有输出信号、内部信号的下一个周期的值，并且所有信号在T3下降沿到来时进行修改）。这个整体框架是我们的程序的创新点，这样避免了“线与”逻辑带来的CPU性能低下的问题，而且逻辑清晰明了，易于后期拓展功能。

8.1.3 遇到的问题与解决方法

问题 我们的代码采用了完全时序化的思路（所有寄存器的状态均在T3下降沿进行改变）。理论上在对实验板进行CLEAR操作并按下QD按钮之后，CPU会进入W1阶段（具体地，在T1上升沿，进入W1阶段；在T3下降沿结束W1阶段），电路中寄存器的状态应该根据 $W1=1, W2=0, W3=0$ 进行改变，但经过反复调试，发现其状态发生转换的时候， $W1=0, W2=0, W3=0$ （硬件有缺陷）。

解决方法 采用了节拍重映射，将($oriW1, oriW2, oriW3$)映射到($W1, W2, W3$)，将节拍电位信号向后推迟一个机器周期，实现了正确的控制。

8.2 7月1日

8.2.1 今日进展

继续对昨日的程序进行测试，并在其基础上正确地添加了执行指令的功能。同时增加了3条拓展指令。到此为止完成了第一版程序。接着进行流水线的设计，但是在使用自定义指令进行测试的时候出现了问题，尚未解决。

8.2.2 设计思路

在每条指令的最后一个节拍中加入LIR和PCINC信号，即可实现同时地进行本条指令的执行和下一条指令的取指，实现了基本的二级流水线。注：需要注意的是，在JMP指令、JZ指令（Z=1）、JC指令（C=1）执行时，由于一定会发生PC寄存器数值的改动，这里必然会导致流水线的断流（因为不可能在一个节拍内同时对PC寄存器赋值和根据PC寄存器中的值取内存中的指令）。

8.2.3 遇到的问题与解决方法

问题

1. 在对执行指令部分代码进行测试的时候，我们发现在最开始取第一条指令的时候存在空拍的问题，执行指令时产生了怪异的现象。

2. 我们发现在有流水线的情况下，在INC指令之后紧接着执行ST指令会直接进入ST指令的W3阶段（理应进入ST指令的W2阶段），导致执行出错。

解决方法

1. 经过反复调试，我们发现昨日编写的对W1W2W3进行节拍重映射的assign语句未考虑指令执行处的特殊情况，从而产生了错误。为了方便实现执行指令处的状态转移，我们加入了寄存器cnt计数器，从而正确地实现了指令的执行。
2. 该问题在7月1日尚未解决（在7月2日解决了），但我们怀疑问题在于：在T3下降沿到来之前Load IR操作已经执行，导致在T3下降沿的时候，IR寄存器中存储的是下一条指令，而我们编写这段代码的时候的逻辑为IRH是Load IR操作执行之前的IRH，导致生成的W1W2W3信号错误。

```

assign W1 = ((write_reg_mode || read_reg_mode)&& ((!oriW1 && !oriW2) || oriW2)) ||
           ((read_mem_mode || write_mem_mode) && ((!oriW1 && !oriW2) || oriW1)) ||
           (fetch_exec_mode &&
            (cnt == 'b000 ||
             cnt == 'b001 ||
             (cnt == 'b011 &&
              ((IRH == 'b0111 && C == 'b1) ||
               (IRH == 'b1000 && Z == 'b1) ||
               IRH == 'b1001))
            ));

```

```

assign W2 = ((write_reg_mode || read_reg_mode) && oriW1) ||
           (fetch_exec_mode &&
            (cnt == 'b010 ||
             (cnt == 'b011 &&
              !(IRH == 'b0101 || IRH == 'b0110 ||
               (IRH == 'b0111 && C == 'b1) ||
               (IRH == 'b1000 && Z == 'b1) ||
               IRH == 'b1001)) || cnt == 'b100
            ));

```

```

assign W3 = fetch_exec_mode && cnt == 'b011 && (IRH == 'b0101 || IRH == 'b0110);

```

8.3 7月2日

8.3.1 今日进展

根据昨日的猜测，我们修改了自动机的状态定义（增加了两种状态），令W只依赖于自动机状态而不依赖于输入信号（如IRH），从而正确地实现了流水线模型控制器。除此之外，我们以树状数组这种数据结构为基础，编写了一份精妙的测试程序，测试到了所有类型的指令，同时说明了我们的控制器可以正确实现高效的算法与数据结构。

8.3.2 设计思路

在实现了顺序模型处理器和流水线模型处理器的代码之后，我们对所有指令的所有情况进行了测试，均无误。但是我们并不满足于此，我们希望对其有更深刻的理解、模拟用户使用的真实情景、测试CPU的实际性能与稳定性，于是在TEC-8实验平台上使用实际的算法与数据结构对其进行测试，选择了一个操作实现并不太复杂的数据结构——树状数组。我们编写了一段程序，实现了对树状数组的两次修改与两次查询，并输出查询结果。

8.3.3 遇到的问题与解决方法

问题

1. 经过代码测试，发现在 $Z=1$ 时，执行JZ指令后LIR未被置为1。
2. 在测试程序的编写过程中，我们为了实现循环判断（如 $i \leq n$ ），起初的想法是结合SUB和JC/JZ进行判断，SUB会修改掉存储n的寄存器的值从而导致我们需要多用一个寄存器来存储n的值，但是寄存器非常宝贵，已经使用了四个。
3. 在测试程序的编写过程中，我们有时需要修改寄存器的值为某一定值，但是由于寄存器非常宝贵，已经使用了四个，我们无法使用一个寄存器存储这个定值。
4. 在编写测试程序的机器代码时，我们发现，一些for循环里的JC和JZ指令的偏移量超过了四位（如-9），导致无法实现跳转的功能。
5. 在测试程序的编写过程中，我们有时需要使用JMP指令，但是由于该指令需要一个另外的寄存器用来存储要跳转到的地址，与问题2和问题3类似，我们的寄存器又不够用了。

解决方法

1. 我们之前把LIR_next的意义搞错了（即LIR_next为下一步的LIR，我们把它当成了当前的LIR），修改LIR_next的逻辑后得到了正确的代码。
2. 我们加入了第四条拓指，CMP指令。
3. 我们想到了一个方案来做到在不借助其他寄存器的情况下对某寄存器进行赋定值x：使用SUB指令让该寄存器清零，再使用x次INC指令。
4. 为了保证程序功能的完整性，我们不希望删除循环里的指令。我们不得已采用了多次跳转（例如连续两次LC），而我们的输入数据保证了在顺序执行时，第二次的LC不跳转，只有循环时才会触发第二个LC。
5. 幸运的是，在这个JMP指令执行处，R3寄存器可以随意更改，于是我们使用SUB指令和JZ指令制造了一个 $Z=1$ 情况下的JZ跳转，实现了PC相对寻址，解决了需要一个寄存器来存储要跳转到的地址的问题。

8.4 结语

至此，我们实现了顺序模型处理器和流水线模型处理器，准备于7月3日进行验收。

9 附件3 贡献度表

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				
基本功能	基本功能	实现的功能	附加功能	实现的功能
		<input checked="" type="checkbox"/> 顺序模型处理器 <input checked="" type="checkbox"/> 存储器功能 <input checked="" type="checkbox"/> 寄存器功能 <input checked="" type="checkbox"/> 加减乘除功能		<input checked="" type="checkbox"/> 扩展 4 条指令 <input checked="" type="checkbox"/> 修改 PC 指针功能 <input checked="" type="checkbox"/> 其他: 消除毛刺、避免线与、数据流建模+行为级建模、时序逻辑、RTL 仿真
拓展题目	基本功能	<input checked="" type="checkbox"/> 解决冒险问题的流水型控制器	附加功能	<input checked="" type="checkbox"/> 扩展 4 条指令 <input checked="" type="checkbox"/> 修改 PC 指针功能 <input checked="" type="checkbox"/> 其他: 消除毛刺、避免线与、数据流建模+行为级建模、时序逻辑、RTL 仿真
学号	姓名	承担的工作		贡献度 (总共 100%)
2023211102	苏泽勤	需求分析, 代码设计、编写与调试, 编写测试程序, 报告撰写		28%
2023211103	冯仁宇	需求分析, 代码设计、编写与调试, 编写测试程序, 报告撰写		28%
2023211111	张启涵	需求分析, 代码设计、编写与调试, 报告撰写		28%
2023211119	汪顺	需求分析, 报告中图表的绘制, 报告撰写		16%

图 15: 贡献度表

10 附件4 RTL仿真图

以下电路是由Quartus II根据流水型控制器代码生成的RTL仿真图：这部分电路展示了代码中数据流建模的组合逻辑（用于生成next信号）。

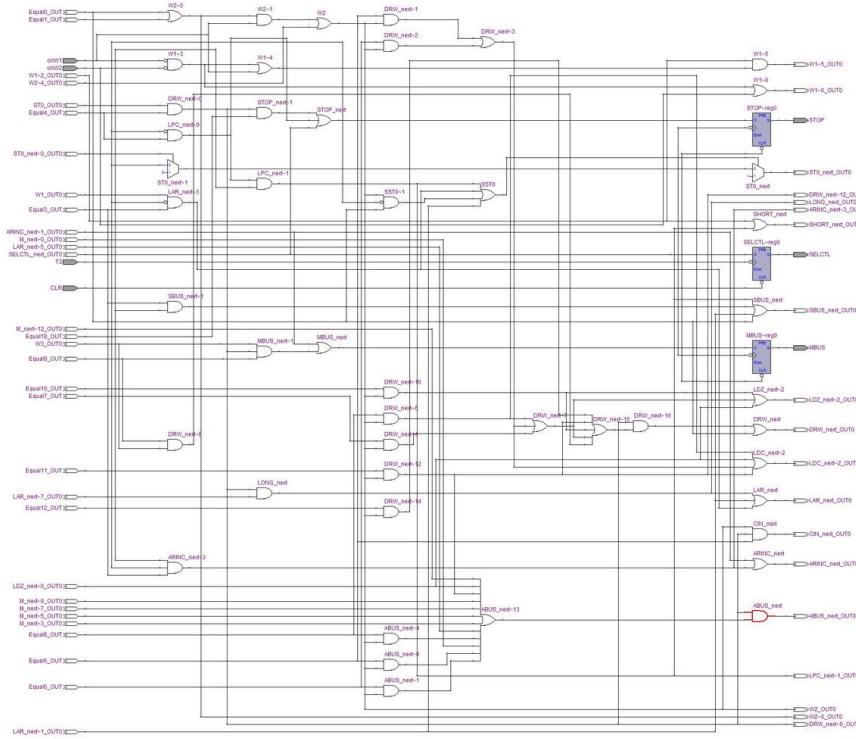
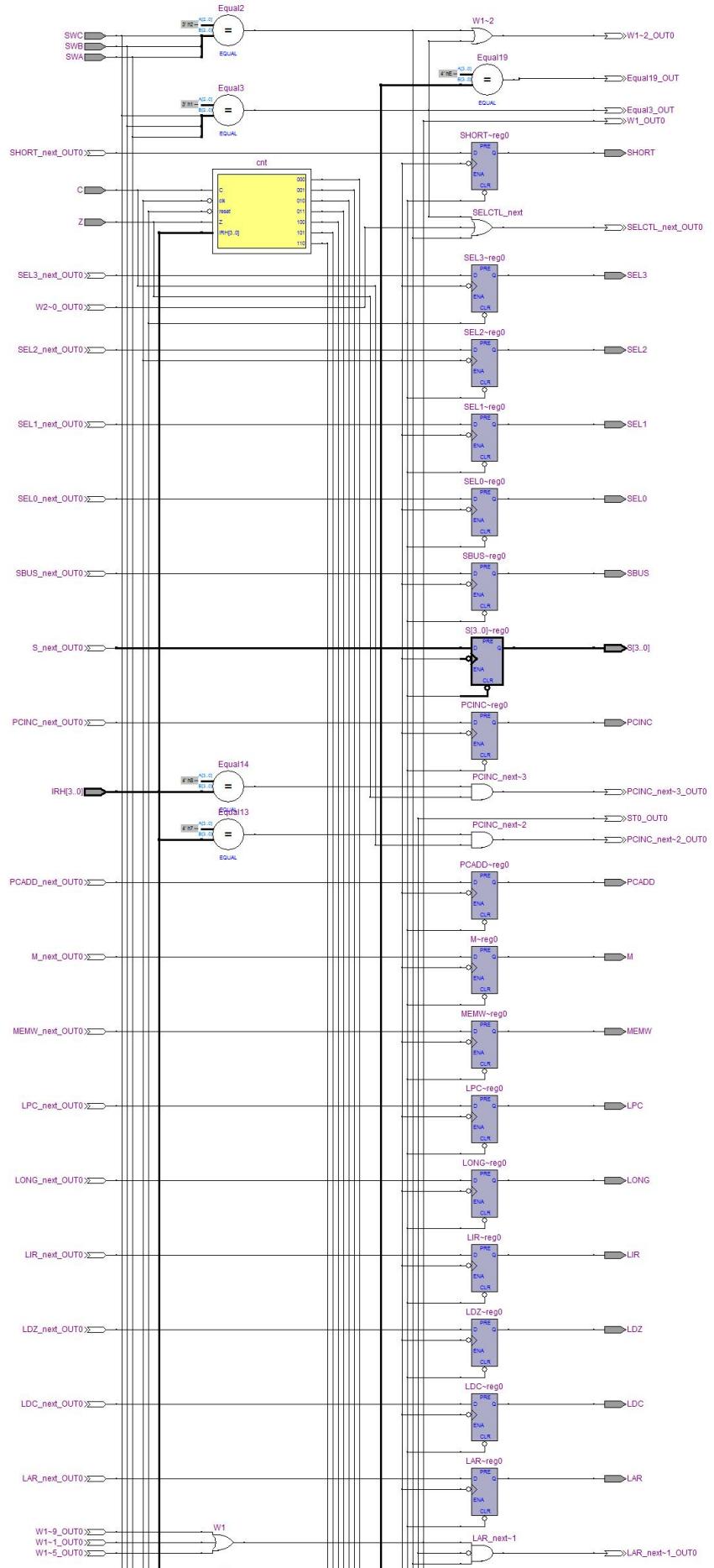


图 16: 组合逻辑部分硬件仿真图

这部分电路展示了我们代码中行为级建模的时序逻辑（用于将next信号赋值给对应的寄存器）：



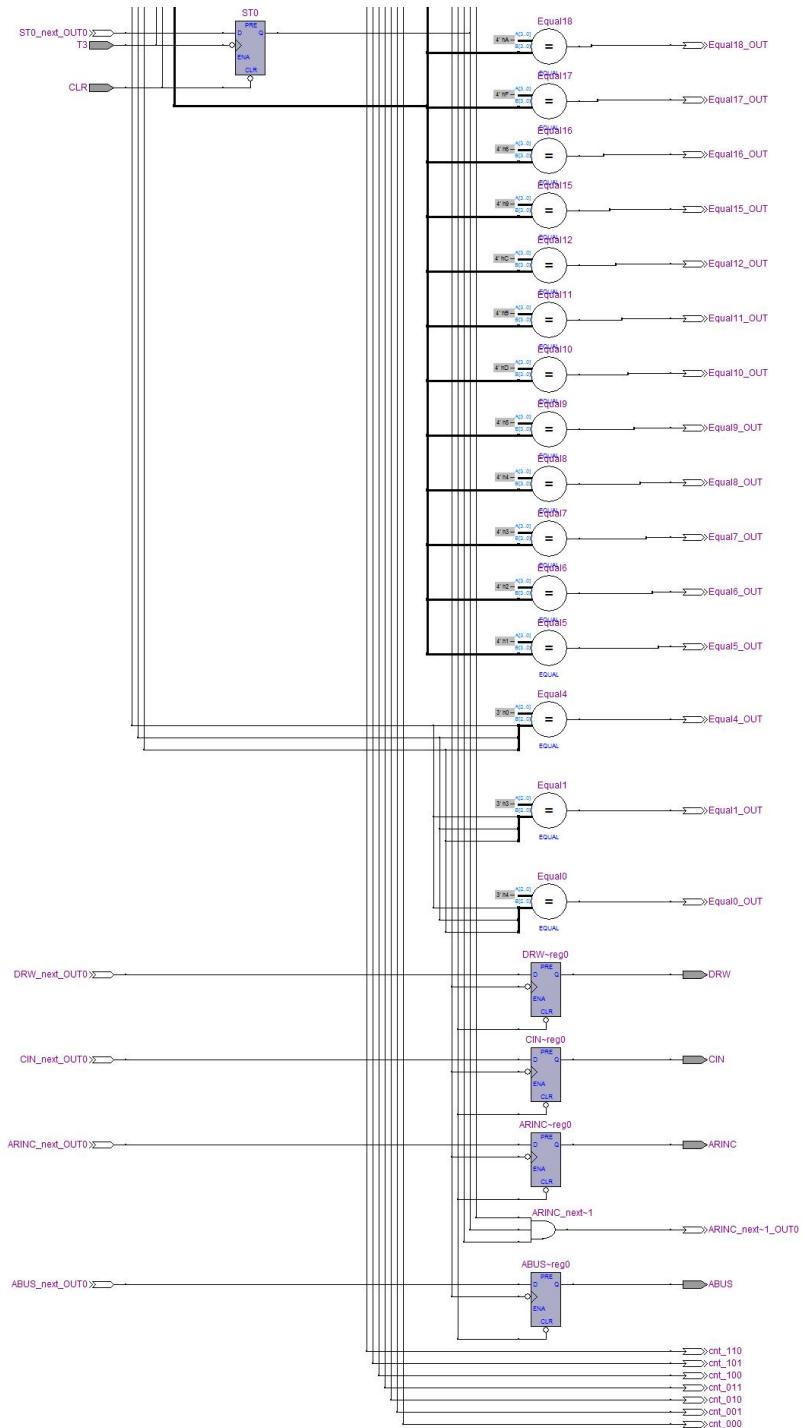


图 18: 时序逻辑部分硬件仿真图-续