计算机组成原理课程设计：基于 Altera CPM7128 的硬连线控制器设计 实验报告

毛子恒	姜山	赵雨腾	刘洋
2019211397	2019211402	2019211468	2019211453

北京邮电大学 计算机学院

日期：2021 年 7 月 18 日

目录

1 概览

1.1 任务描述

必选题目 按照给定数据格式、指令系统和数据通路，根据所提供的器件要求，自行设计一个基于硬布线控制器的顺序模型处理机。

基本功能 根据设计方案，在 TEC-8 上进行组装、调试运行。

附加功能

- 在原指令基础上扩指至少三条。
- 允许用户在程序开始时指定 PC 指针的值。

自选题目一 在必选题目基础上，完成流水硬连线控制器的设计根据设计方案，在 TEC-8 上进行组装、调试运行。

自选题目二 在必选题目基础上，设计实现带有中断功能的硬布线控制器。

1.2 实验设备和环境

- TEC-8 计算机硬件综合实验系统一台
- 逻辑测试笔一支
- Quartus II 9.0

1.3 成员分工

毛子恒 负责测试设备、代码编写和维护、运行测试程序；负责文档的实验原理、设计详解的流水线和中断的文字部分。

姜山　负责测试设备、运行测试程序；参与程序原理讨论，绘制指令流程图，记录日志。

赵雨腾　负责测试设备、代码编写和维护、编写 7 个二进制机器测试程序；负责文档的伪代码写入。

刘洋　负责测试设备、运行测试程序，参与程序原理讨论，编写指令译码表，负责文档的设计详解的基础功能部分，记录日志。

2　实验原理

2.1　模型计算机时序信号

TEC-8 模型计算机主时钟 MF 的频率为 1MHz，执行一条微指令需要 3 个节拍脉冲 T1、T2、T3。TEC-8 模型计算机时序采用不定长机器周期，绝大多数指令采用 2 个机器周期 W1、W2，少数指令采用一个机器周期 W1 或者 3 个机器周期 W1、W2、W3。模型机的时序图如**图 ??**。

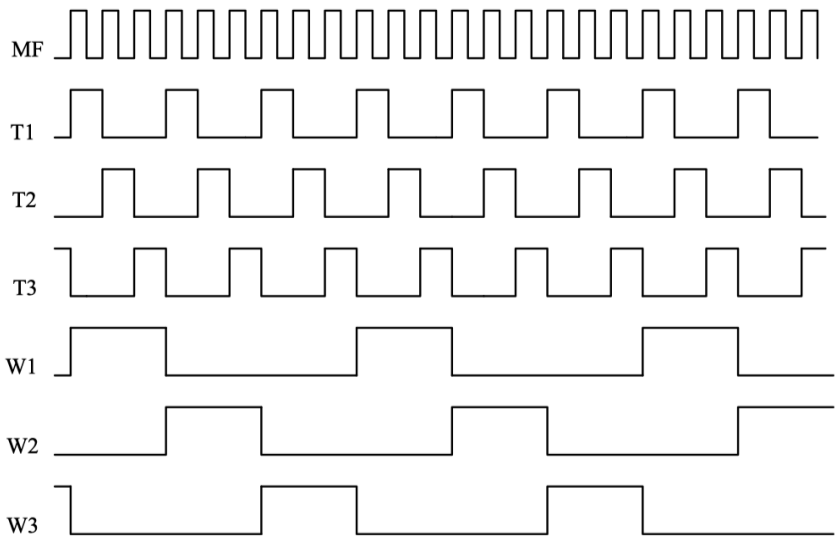


图 1: TEC-8 模型计算机时序图

2.2　模型计算机组成

TEC-8 模型计算机的电路框图如**图 ??**。

2.3　EPM7128 引脚及信号说明

硬连线控制器所需的信号的输出、输入信号的引脚号如**表 ??**。

各信号的说明如**表 ??**。

表 1: 作为硬连线控制器时的 EPM7128 引脚规定

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			

表 2: 各信号的说明

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

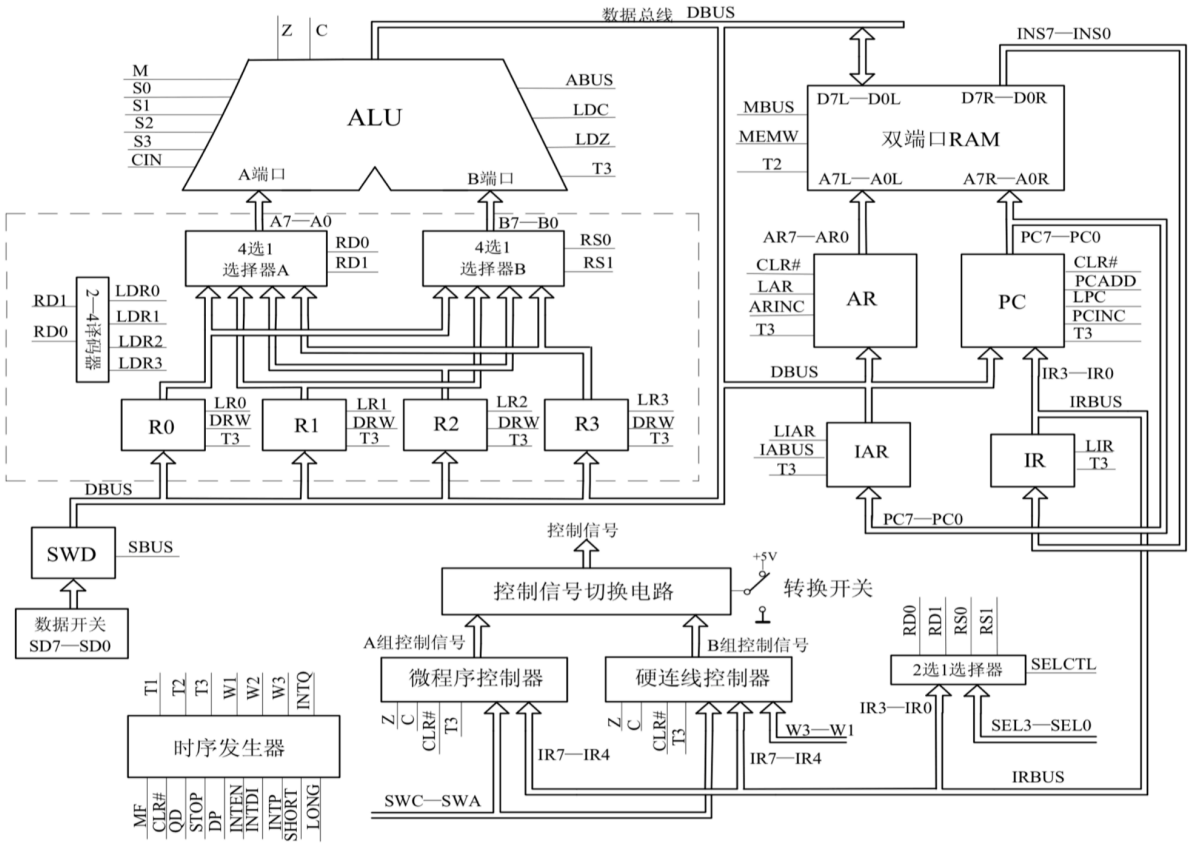


图 2: TEC-8 模型计算机框图

2.4 硬连线控制器设计

2.4.1 硬连线控制器的基本原理

每个微操作控制信号 S 是一系列输入量的逻辑函数，即用组合逻辑来实现：

$$S = f(I_m, M_i, T_k, B_j)$$

其中 I_m 是机器指令操作码译码器的输出信号，M_i 是节拍电位信号，T_k 是节拍脉冲信号，B_j 是状态条件信号。在 TEC-8 实验系统中，节拍脉冲信号 T_k(T1~T3) 已经直接输送给数据通路。因为机器指令系统比较简单，省去操作码译码器，4 位指令操作码 IR4~IR7 直接成为 I_m 的一部分；

由于 TEC-8 实验系统有控制台操作，控制台操作可以看作一些特殊的功能复杂的指令，因此 SWC、SWB、SWA 可以看作是 I_m 的另一部分。M_i 是时序发生器产生的节拍信号 W1~W3；B_j 包括 ALU 产生的进位信号 C、结果为 0 信号 Z 等等。

2.4.2 机器指令周期流程图设计

设计微程序控制器使用流程图。设计硬连线控制器同样使用流程图。微程序控制器的控制信号以微指令周期为时间单位，硬连线控制器以节拍电位（CPU 周期）为时间单位，两者在本质上是一样的，1 个节拍电位时间和 1 条微指令时间都是从节拍脉冲 T1 的上升沿到 T3 的下降沿的一段时间。在微程序控制器流程图中，一个执行框代表一条微指令，在硬连线控制器流程图中，一个执行框代表一个节拍电位时间。

2.4.3 执行一条机器指令的节拍电位数

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

对于一些控制台操作，需要 4 个节拍电位才能完成规定的功能。为了满足这种情况，可以将控制台操作化成两条机器指令的节拍。为了区分写寄存器操作的 2 个不同阶段，可以用一个特殊的寄存器标志。

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

信号 LONG 和 SHORT 只对紧跟其后的第一个节拍电位的产生起作用。

2.5 实验步骤

1. 设计一个硬连线控制器，和 TEC-8 模型计算机的数据通路结合在一起，构成一个完整的 CPU，该 CPU 要求：
   (a) 能够完成控制台操作：启动程序运行、读存储器、写存储器、读寄存器和写寄存器。
   (b) 能够执行**表 ??**中的指令，完成规定的指令功能。
2. 在 Quartus II 下对硬连线控制器对设计方案进行编程和编译。
3. 将编译后的硬连线控制器下载到 TEC-8 实验台上的 ISP 器件 EPM7128 中去，使 EPM7128 成为一个硬连线控制器。
4. 根据指令系统，编写检测硬连线控制器正确性的测试程序，并用测试程序对硬布线控制器在单拍方式下进行调试，直到成功。
5. 在调试成功的基础上，整理出设计文件。

3 设计详解

3.1 基础功能

该 CPU 需要完成的功能可以分为两类，一类完成控制台操作，即根据 SWC、SWB、SWA 的不同值来读写寄存器、存储器或启动程序运行；另一类执行机器指令，即根据 IR0~IR3 的值来执行不同的指令。分析机器指令流程图可得，第一类操作都可以分成两个机器周期完成，第二类操作的大部分指令都在可以在两个机器周期完成（少数需要三个机器周期）第一个周期为取指周期，第二个周期为执行周期。下面详细分析每个功能如何实现。

表 3: 指令系统

名称	助记符	功能	指令格式		
			IR7~IR4	IR3~IR2	IR1~IR0
空指令	`NOP`	无	0000	XX[1]	XX
加法	`ADD Rd, Rs`	Rd ← Rd[2]+ Rs[3]	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 ← @[4]+ offset[5]	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
输出	`OUT [Rs]`	DBUS ← Rs	1010	XX	Rs
逻辑或	`OR Rd, Rs`	Rd ← Rd ∨ Rs	1011	Rd	Rs
比较	`CMP Rd, Rs`	Rd - Rs	1100	Rd	Rs
移动值	`MOV Rd, Rs`	Rd ← Rs	1101	Rd	Rs
停机	`STOP`	暂停运行	1110	XX	XX

[1] XX 代表随意值。

[2] Rs 代表源寄存器号。

[3] Rd 代表目的寄存器号。

[4] @ 代表当前 PC 的值。

[5] offset 是一个 4 位的补码有符号数。

3.1.1 写寄存器

当 SWC SWB SWA=100 的时候，进入写寄存器状态，一共需要四个机器周期，分别写寄存器 R0、R1、R2、R3，根据标志位 ST0 的值将它分为两个阶段，每个阶段两个节拍，当 ST0=0 时，执行第一个阶段，当 ST0=1 时，执行第二个阶段。

写每个寄存器时，都需要使 SBUS=1，DRW=1，而不同点主要在于 SEL0~SEL3 的值，SEL3~SEL2 指定当前写入的寄存器，SEL1~SEL0 指定上次写入的寄存器即当前读出的寄存器。例如，第一个 W1 时 SEL3~SEL2=00，SEL1~SEL0=11，表示写入寄存器 R0，第一个 W2 时 SEL3~SEL2=01，SEL1~SEL0=00，表示写入寄存器 R1，并读取 R0 的值（指示灯 B7~B0），以此类推。同时为了辅助 ST0 的改变，设置 SST0 标志位，在第一个 W2 节拍时，使 SST0=1，ST0 检测到 W2&SST0=1 时，会由 0 变为 1，标志着进入到第二个 W1、W2。

在写寄存器的第四拍将 ST0 重新设置为 0，如果再进行下一拍则重新从 R0 开始写入。

3.1.2 读寄存器

当 SWC SWB SWA=011 的时候，进入读寄存器状态，一共需要两个机器周期，W1 读寄存器 R0、R1，W2 读寄存器 R2、R3。

根据 SEL3~SEL0 的值选择不同的寄存器，SEL3~SEL2 决定 A 端口指示的寄存器，SEL1~SEL0 决定 B 端口指示的寄存器，W1 时 SEL3~SEL2=00，SEL1~SEL0=01，ALU 的 A 端口（指示灯 A7~A0）指示寄存器 R0 的值，ALU 的 B 端口（指示灯 B7~B0）指示寄存器 R1 的值。之后的 W2 节拍类似，改变 SEL 的值读出 R2、R3 寄存器的值。

3.1.3 写存储器

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，每个阶段只有一个机器周期。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。

3.1.4 读存储器

当 SWC SWB SWA=010 的时候，进入读存储器状态，与写存储器类似，同样根据标志位 ST0 的值将它分为两个阶段，每个阶段只有一个机器周期。第一个阶段指定首地址，第二个阶段不断循环，地址自加后依次读存储单元。

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，MBUS=1，把存储单元的值读到数据总线上，然后 ARINC=1，将 AR 加 1，在下一个节拍中读取下一个存储单元的值，不断循环直到 CLR。

3.1.5 执行指令

当 SWC SWB SWA=000 的时候，进入执行程序状态，第一个机器周期取指令，第二个机器周期进入执行指令阶段，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。

3.1.6 扩展指令

在原指令的基础上，我们新增了五条扩展指令，如**表 ??**。

表 4: 扩展指令

名称	助记符	功能	指令格式		
			IR7~IR4	IR3~IR2	IR1~IR0
空指令	NOP	无	0000	XX	XX
输出	OUT [Rs]	DBUS ← Rs	1010	XX	Rs
逻辑或	OR Rd, Rs	Rd ← Rd ∨ Rs	1011	Rd	Rs
比较	CMP Rd, Rs	Rd - Rs	1100	Rd	Rs
移动值	MOV Rd, Rs	Rd ← Rs	1101	Rd	Rs

这些指令的流程图如**图 ??**所示。

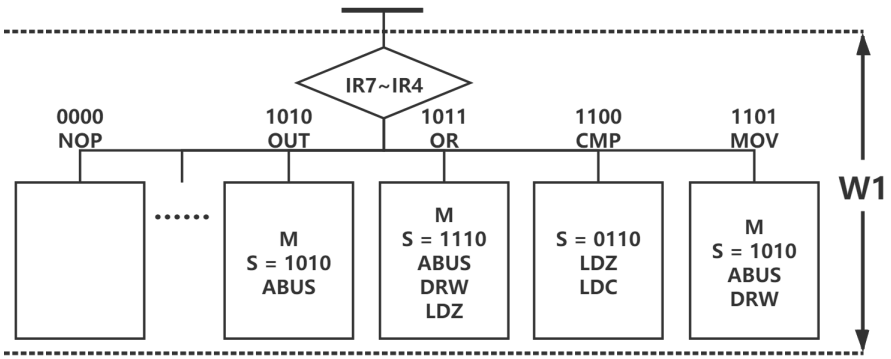


图 3: 扩展指令流程图

3.1.7 用户指定 PC 功能

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 的原理与写寄存器的原理类似，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；第二阶段即是正常的取指令阶段，ST0=1，PCINC=1，LIR=1，开始不断取指令并执行指令。

此功能的流程图如**图 ??**所示。

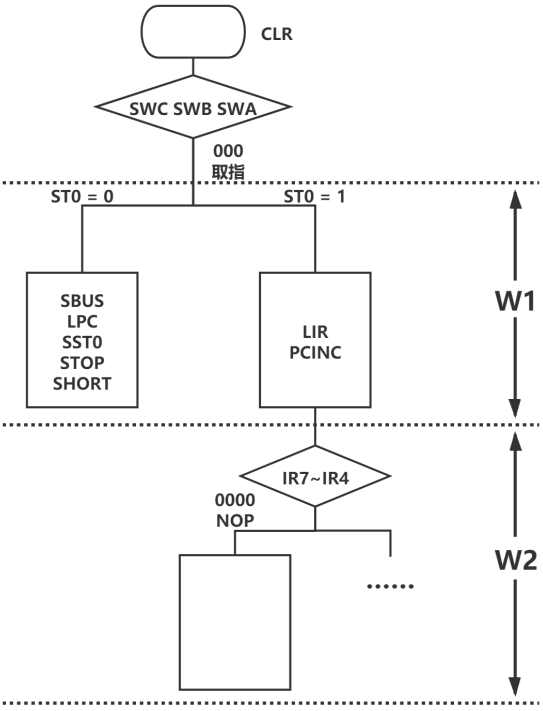


图 4: 用户指定 PC 功能流程图

3.1.8 指令译码表

在机器指令周期流程图的基础上，我们就可以根据流程图来编写译码表，译码表的首行列出了所有的指令信号，首列列出了所有的控制信号，每一个格子表示在什么指令和什么条件下需要发出什么信号。横着看每一行，表示需要发出该控制信号的所有情况；竖着看每一列，表示该条指令在 W1，W2 节拍下需要发出的所有控制信号。针对流水与非流水情况，分别设计了两张译码表。相比非流水情况下，流水情况下的译码表要更加复杂，每一拍需要发出的控制信号更多。

指令中用到的运算器的功能如**表 ??**所示，指令译码表如**图 ??**所示。

表 5: ALU 逻辑/算术功能表

S (ALU 工作方式选择输入)	M = 1 (逻辑运算)	M = 0 (算术运算)
0000	$\overline{A}$	$A+1$
0110	$\overline{A \oplus B}$	$A-B-1$
1001	$\overline{A \oplus B}$	$A+B$
1010	B	$(A \vee \overline{B}) + A \wedge B$
1011	$A \wedge B$	$A \wedge B - 1$
1110	$A \vee B$	$(A \vee \overline{B}) + A$
1111	A	$A-1$

	NOP	ADD	SUB	AND_I	INC	LD	ST	JC	JZ	JMP	OUT_I	OR_I	CMP	MOV	STP	WRITE_REG	READ_REG	READ_MEM	WRITE_MEM	INS_FETCH
IR7-4	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110					
M	0	0	0	W2	0	W2	3	0	0	W2	W2	W2	0	W2	0	0	0	0	0	0
CIN	0	W2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S3	0	W2	0	W2	0	W2	3	0	0	W2	W2	W2	0	W2	0	0	0	0	0	0
S2	0	0	W2	0	0	0	W2	0	0	W2	0	W2	W2	0	0	0	0	0	0	0
S1	0	0	W2	W2	0	W2	3	0	0	W2	W2	W2	W2	W2	0	0	0	0	0	0
S0	0	W2	0	W2	0	0	W2	0	0	W2	0	0	0	0	0	0	0	0	0	0
ABUS	0	W2	W2	W2	W2	W2	3	0	0	W2	W2	W2	0	W2	0	0	0	0	0	0
MBUS	0	0	0	0	0	W3	0	0	0	0	0	0	0	0	0	0	0	W1*ST0	0	0
DRW	0	W2	W2	W2	W2	W3	0	0	0	0	0	W2	0	W2	0	W1+W2	0	0	0	0
LDZ	0	W2	W2	W2	W2	0	0	0	0	0	0	W2	W2	0	0	0	0	0	0	0
LDC	0	W2	W2	0	W2	0	0	0	0	0	0	0	W2	0	0	0	0	0	0	0
LPC	0	0	0	0	0	0	0	0	0	W2	0	0	0	0	0	0	0	0	0	W1*~ST0
LAR	0	0	0	0	W2	W2	W2	0	0	0	0	0	0	0	0	0	0	W1*~ST0	W1*~ST0	0
LONG	0	0	0	0	W2	W2	W2	0	0	0	0	0	0	0	0	0	0	0	0	0
PCADD	0	0	0	0	0	0	0	W2*C	W2*Z	0	0	0	0	0	0	0	0	0	0	0
SHORT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	W1	W1	W1*~ST0
MEMW	0	0	0	0	0	0	W3	0	0	0	0	0	0	0	0	0	0	0	W1*ST0	0
STOP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	W2	W1+W2	2	W1	W1	W1*~ST0
SEL3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	(W1+W2)*ST0	W2	0	0	0
SEL2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	W2		0	0	0
SEL1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	W1*~ST0+W2*ST0	W2	0	0	0
SEL0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	W1	2	0	0	0
SELCTL	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	W1+W2	2	W1	W1	0
ARINC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	W1*ST0	W1*ST0	W1*ST0
PCINC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	W1*ST0
LIR	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	W1+W2	0	W1*~ST0	W1	W1*~ST0
SBUS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0					

图 5: 指令译码表

3.1.9 程序实现

硬布线控制器的程序实现有两种风格，我们分别称之为数据流描述和行为描述，数据流描述是基于如**图 ??**的译码表的描述，即采用逻辑表达式实现所有信号。行为描述是基于指令流程图，这种实现方式不需要写出译码表，根据流程图的判断条件写出程序即可，这两种描述方式的代码框架如**源代码 ??**和**源代码 ??**。

源代码 1: 数据流描述

```
-- 操作模式
    WRITE_REG <= '1' when SW = "100" else '0';
    READ_REG <= '1' when SW = "011" else '0';
    INS_FETCH <= '1' when SW = "000" else '0';
    READ_MEM <= '1' when SW = "010" else '0';
    WRITE_MEM <= '1' when SW = "001" else '0';

-- 操作码
    ADD <= '1' when IR = "0001" and INS_FETCH = '1' and STO = '1' else '0';
    SUB <= '1' when IR = "0010" and INS_FETCH = '1' and STO = '1' else '0';

    OUT_I <= '1' when IR = "1010" and INS_FETCH = '1' and STO = '1' else '0';
    OR_I <= '1' when IR = "1011" and INS_FETCH = '1' and STO = '1' else '0';
    CMP <= '1' when IR = "1100" and INS_FETCH = '1' and STO = '1' else '0';
    MOV <= '1' when IR = "1101" and INS_FETCH = '1' and STO = '1' else '0;

-- STO 状态
    process(CLR, T3, W)
    begin
        if (CLR = '0') then
            STO <= '0';
        elsif (T3'event and T3 = '0') then
            if (STO = '0' and ((WRITE_REG = '1' and W(2) = '1') or (READ_MEM =
            ↪  '1' and W(1) = '1') or (WRITE_MEM = '1' and W(1) = '1') or
            ↪  (INS_FETCH = '1' and W(1) = '1'))) then
                STO <= '1';
            elsif (STO = '1' and (WRITE_REG = '1' and W(2) = '1')) then
                STO <= '0';
            end if;
        end if;
    end process;

-- 控制信号合成
    SBUS <= ((WRITE_REG or (READ_MEM and not STO) or WRITE_MEM or (INS_FETCH
    ↪  and not STO)) and W(1)) or (WRITE_REG and W(2));
    DRW <= (WRITE_REG and (W(1) or W(2))) or ((ADD or SUB or AND_I or INC or
    ↪  OR_I or MOV) and W(2)) or (LD and W(3));
```

源代码 2: 行为描述

```
    process (SW, IR,  W(1),  W(2),  W(3), T3 ,CLR, C, Z, STO, SSTO)
    begin
-- 初始化控制信号
-- STO 状态
        if (clr = '0') then
            STO <= '0';
        else
            if (T3'event and T3 = '0') and SSTO = '1' then
                STO <= '1';
            end if;

            case SW is
-- ------------------------------------------------------------------
                -- WRITE_MEM
                when "001" =>
-- ------------------------------------------------------------------
                -- READ_MEM
                when "010" =>
-- ------------------------------------------------------------------
                -- READ_REG
                when "011" =>
-- ------------------------------------------------------------------
                -- WRITE_REG
                when "100" =>
-- ------------------------------------------------------------------
                -- INS_FETCH
                when "000" =>
--------------------------------------------------------------------
                    -- 用户置入 PC 的值指定程序初始位置
                    if STO = '0' then
                        LPC <=  W(1); -- 用户置入 PC 的值指定程序初始位置
                        SBUS <=  W(1);
                        SSTO <=  W(1);
                        SHORT <=  W(1);
                        STOP <=  W(1);
                        SELCTL <=  W(1);
--------------------------------------------------------------------
                    -- 执行程序
                    else -- STO='1'
                        LIR <=  W(1);
                        PCINC <=  W(1);
                        case IR is
                            when "0001" => -- ADD
                                S <=  W(2) & '0' & '0' &  W(2);
                                CIN <=  W(2);
                                ABUS <=  W(2);
                                DRW <=  W(2);
                                LDZ <=  W(2);
                                LDC <=  W(2);
                            when "0010" => -- SUB
```

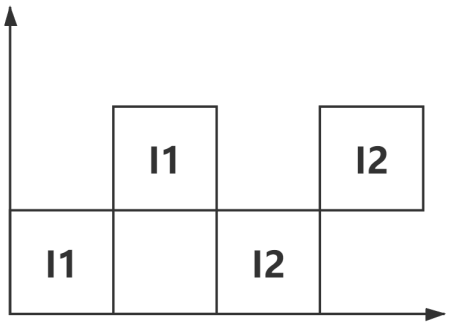
```
51                                  S <= '0' &  W(2) &  W(2) & '0';
52                                  ABUS <=  W(2);
53                                  DRW <=  W(2);
54                                  LDZ <=  W(2);
55                                  LDC <=  W(2);
56                              when others => null; -- IR
57                          end case; -- IR
58                      end if; -- ST0='1'
59                  when others => null;--SW="000"
60              end case; -- SW
61          end if; -- clr='1'
62      end process;
```

3.2 流水线

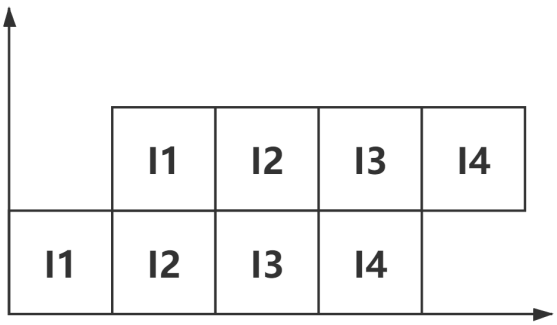
指令周期中包含两个子过程，取指令和执行指令。**图 ??**为非流水计算机的时空图，对于非流水计算机来说，上一条指令的两个子过程全部执行完毕之后才能开始下一条指令。因此，每隔 2 个机器时钟周期才有一个输出结果。

图 ??为流水计算机的时空图，对于流水计算机来说，上一条指令与下一条指令的子过程在时间上可以重叠执行。因此，当流水线满载时，每一个时钟周期就可以输出一个结果。

对于需要两个节拍电位的指令，如 LD 和 ST，则在其第二拍（W2）时取下一条指令，时空图如**图 ??**所示。



(a) 非流水时空图



(b) 流水时空图

(c) 两拍指令的流水时空图

图 6: 时空图

流水线的具体实现方式就是将上一条指令的执行指令和下一条指令的取指令部分合并到一个节拍电位中进行，其流程图如**图 ??**所示。

相应地，扩展指令的流程也要做类似的改动。由于第一条指令之前没有取指令阶段，所以在用户指定 PC 后加一个节拍电位，用于进行第一个指令的取指令，其流程图如**图 ??**所示。

指令译码表如**图 ??**所示。其中，$S_3 \sim S_0$ 参考**表 ??**。

3.3 中断

由于 EPM7128 芯片中没有接入 LIAR 和 IABUS 管脚，因此我们无法控制 IAR 寄存器，也就无法读出和暂存 PC 的值，意味着我们中断之后无法返回断点，这给我们实现中断带来了很大的困难。经过老师的提示，我们采用 R3 寄存器同步 PC 的值，在中断返回时将 R3 的值赋给 PC，以达到回到断点的目的。

3.3.1 对指令集的改动

我们首先对原有的指令集进行如下改动：

指令中只能访问 R0~R2 寄存器 R3 寄存器用于同步 PC 的值，所以其他指令不能访问该寄存器。

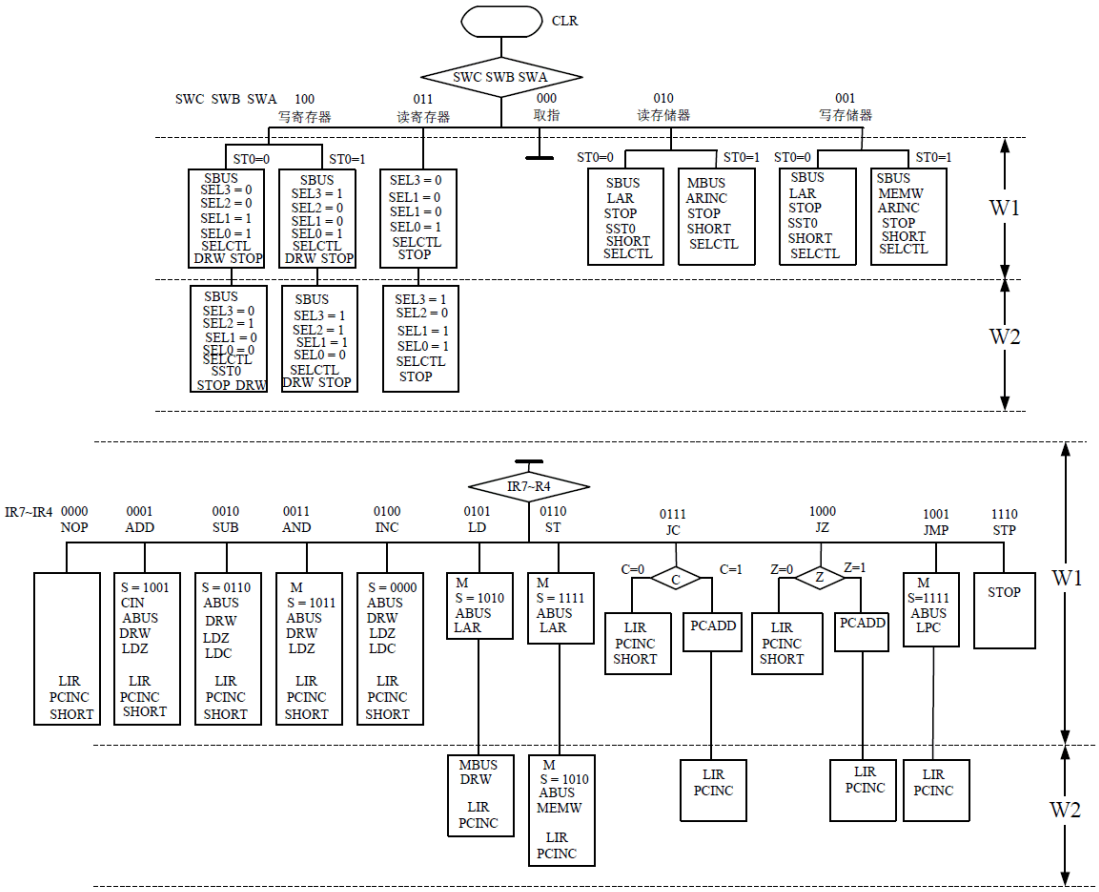
不支持 JZ 和 JC 指令 这两个指令的低四位表示偏移量，我们无法直接读出 IR3~IR0，因此无法实现 R3 与 PC 同步变化。理论上可以通过双端口存储器的左端口读出指令，再通过逻辑与运算分离出低四位，再给 R3 加上这四位得到结果，但是这一系列操作可能需要 6 个以上的节拍电位，过于复杂，由于时间限制而未予实现。

JMP 指令改动 JMP 指令除了要将寄存器中的值赋给 PC 之外，还要赋给 R3，因此需要对指令格式和指令流程进行改动。具体而言，JMP 指令的指令格式如**表 ??**所示。之前程序中的 JMP 指令仍然按照原有格式。

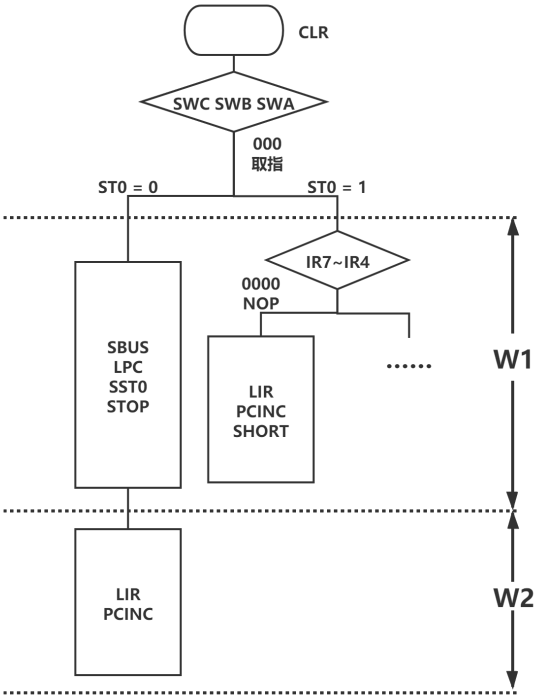
新增 IRET 指令 此指令用于从中断服务程序中返回，它的职能包括开中断、将寄存器 R3 的值赋给 PC，指令格式如**表 ??**。

表 6: 指令系统

名称	助记符	功能	指令格式		
			IR7~IR4	IR3~IR2	IR1~IR0
无条件转移	`JMP [Rd]`	PC ← Rd	1001	11	Rd
中断返回	`IRET`	PC ← R3	1111	XX	XX



(a) 无扩展指令



(b) 用户指定 PC 功能

图 7: 流水线流程图

	NOP	ADD	SUB	AND_I	INC	LD	ST	JC	JZ	JMP	OUT_I	OR_I	CMP	MOV	STP	WRITE_REG	READ_REG	READ_MEM	WRITE_MEM	INS_FETCH
IR7-4	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110					
M	0	0	0	W1	0	W1	W1+W2	0	0	W1	W1	W1	0	W1	0	0	0	0	0	0
CIN	0	W1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S3	0	W1	0	W1	0	W1	W1+W2	0	0	W1	W1	W1	0	W1	0	0	0	0	0	0
S2	0	0	W1	0	0	0	W1	0	0	W1	0	W1	W1	0	0	0	0	0	0	0
S1	0	0	W1	W1	0	W1	W1+W2	0	0	W1	W1	W1	W1	W1	0	0	0	0	0	0
S0	0	W1	0	W1	0	0	W1	0	0	W1	0	0	0	0	0	0	0	0	0	0
ABUS	0	W1	W1	W1	W1	W1	W1+W2	0	0	W1	W1	W1	0	W1	0	0	0	0	0	0
MBUS	0	0	0	0	0	W2	0	0	0	0	0	0	0	0	0	0	0	W1*ST0	0	0
DRW	0	W1	W1	W1	W1	W2	0	0	0	0	0	W1	0	W1	0	W1+W2	0	0	0	0
LDZ	0	W1	W1	W1	W1	0	0	0	0	0	0	W1	W1	0	0	0	0	0	0	0
LDC	0	W1	W1	0	W1	0	0	0	0	0	0	0	W1	0	0	0	0	0	0	0
LPC	0	0	0	0	0	0	0	0	0	W1	0	0	0	0	0	0	0	0	0	W1*~ST0
LAR	0	0	0	0	0	W2	W2	0	0	0	0	0	0	0	0	0	0	W1*~ST0	W1*~ST0	0
LONG	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PCADD	0	0	0	0	0	0	0	W1*C	W1*Z	0	0	0	0	0	0	0	0	0	0	0
SHORT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	W1	W1	W1*~ST0
MEMW	0	0	0	0	0	0	W2	0	0	0	0	0	0	0	0	0	0	0	W1*ST0	0
STOP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	W1	W1+W2	W1+W2	W1	W1	W1*~ST0
SEL3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	(W1+W2) *ST0	W2	0	0	0
SEL2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	W2	0	0	0	0
SEL1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	W2*ST0	W2	0	0	0
SEL0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	W1	W1+W2	0	0	0
SELCTL	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	W1+W2	W1+W2	W1	W1	0
ARINC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	W1*ST0	W1*ST0	0
PCINC	W1	W1	W1	W1	W1	W2	W2	W1*~C+W 2*C	W1*~Z+W 2*~Z	W2	W1	W1	W1	W1	0	0	0	0	0	W2*~ST0
LIR	W1	W1	W1	W1	W1	W2	W2	W1*~C+W 2*C	W1*~Z+W 2*~Z	W2	W1	W1	W1	W1	0	0	0	0	0	W2*~ST0
SBUS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	W1+W2	0	W1*~ST0	W1	W1*~ST0

图 8: 流水线指令译码表

3.3.2 新增的信号和标志

此外，新增了一个输入信号 PULSE，从 61 引脚输入，表示中断脉冲信号。

程序中新增了一些标志，如**表 ??**所示。

表 7: 各标志位的说明

标志	说明
INT	中断标志。
EN_INT	中断允许标志。
ST1	中断周期标志。
INTDI	=1 时，置允许中断标志为 0，禁止 TEC-8 模型计算机响应中断请求。
INTEN	=1 时，置允许中断标志为 1，允许 TEC-8 模型计算机响应中断请求。

3.3.3 中断流程

中断部分的流程图如**图 ??**所示。

R3 与 PC 同步 为了使 R3 与 PC 同步，需要对程序的 PCINC 和 JMP 指令做一些改动。

当取指令时，同时利用 ALU 的运算使 R3 自增 1；当执行 JMP 时，由于预先规定 IR3~IR2 为 11，此时设置 DRW 为 1 即可将另一寄存器的值同时写入 PC 和 R3。

中断周期 默认状态下（即按下 CLR 后），中断允许标志 EN_INT 为 1，中断标志 INT 为 0。即默认开中断，没有手动开中断的指令。

在每个时钟周期的上升沿，根据 INTDI 和 INTEN 来设置中断允许标志；当按下 PULSE 时，根据中断允许标志来设置 INT 的值；在每条指令执行的最后一个节拍电位的 T3 上升沿，如果此时 INT 为 1，则设置 ST1 为 1（在流程图中写作 P0），表示之后进入中断周期。

中断周期中，首先关中断，并且暂停程序等待写入 PC 的值，之后在 W2 的 T3 上升沿将 ST1 置为 0，表示中断周期结束，之后进入中断处理程序。

中断处理程序 中断处理程序与主程序差异不大，只是 R3 和 PC 不需要同步，因此 PCINC 和 JMP 指令不对 R3 做更改。由于我们只实现了单级中断，所以执行处理程序时仍然处于关中断状态，因此可以根据是否开中断来判断是否正在执行中断处理程序。

中断处理程序必须以 IRET 指令结束，表示回到主程序。IRET 指令的 W2 拍开中断，W3 拍将 R3 的值赋给 PC，以达到恢复断点的目的。

由于 W2 拍已经开中断，所以在 W3 拍仍然可以开始中断，将在 IRET 指令执行结束之后再次进入中断周期。

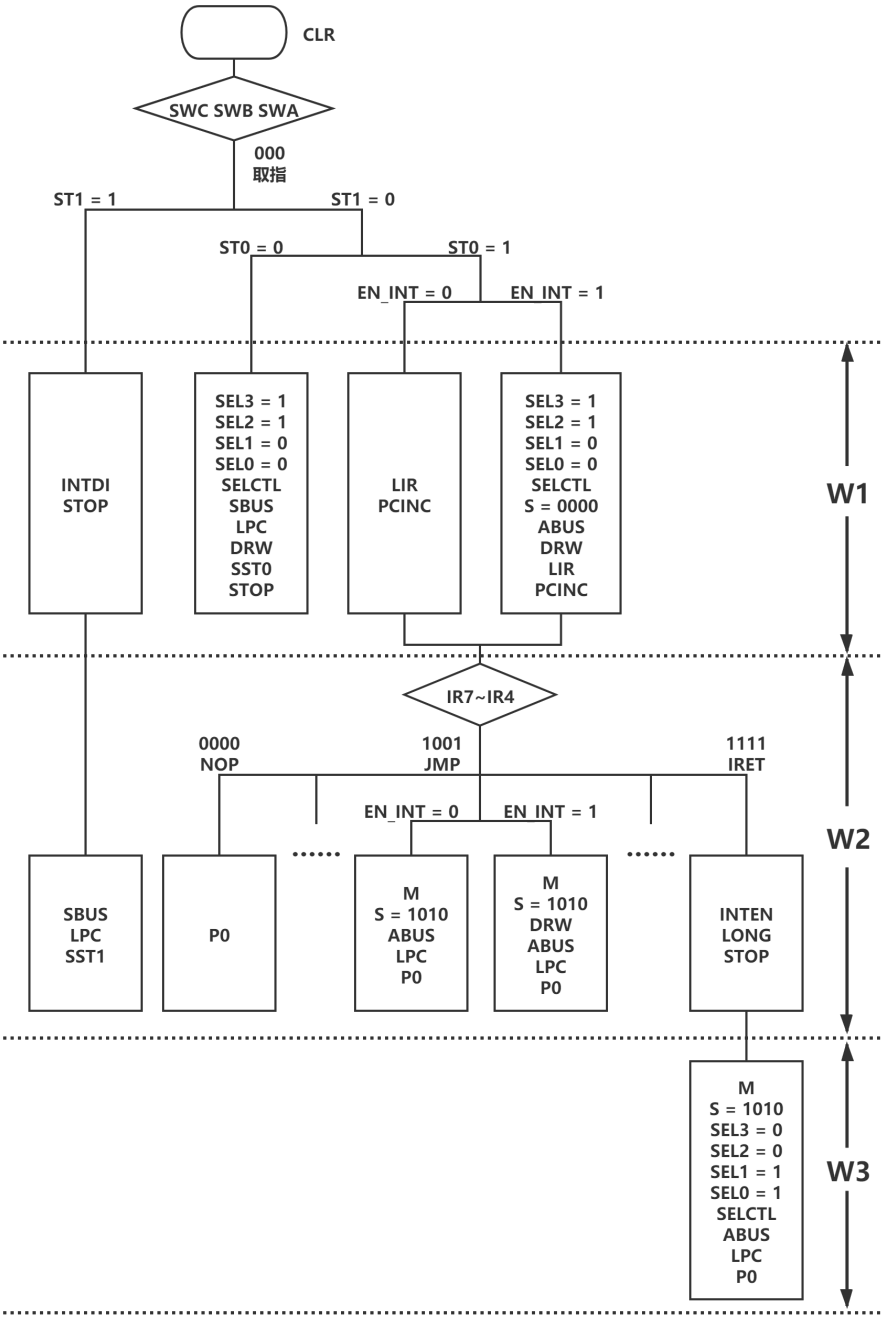

CLR
SWC SWB SWA
000
取指
ST1 = 1
ST1 = 0
ST0 = 0
ST0 = 1
EN_INT = 0
EN_INT = 1
INTDI
STOP
SEL3 = 1
SEL2 = 1
SEL1 = 0
SEL0 = 0
SELCTL
SBUS
LPC
DRW
SST0
STOP
LIR
PCINC
SEL3 = 1
SEL2 = 1
SEL1 = 0
SEL0 = 0
SELCTL
S = 0000
ABUS
DRW
LIR
PCINC
W1
IR7~IR4
0000
NOP
1001
JMP
1111
IRET
EN_INT = 0
EN_INT = 1
SBUS
LPC
SST1
P0
M
S = 1010
ABUS
LPC
P0
M
S = 1010
DRW
ABUS
LPC
P0
INTEN
LONG
STOP
W2
M
S = 1010
SEL3 = 0
SEL2 = 0
SEL1 = 1
SEL0 = 1
SELCTL
ABUS
LPC
P0
W3

图 9: 中断流程图

4 调试总结

4.1 测试结果

4.1.1 测试集 1

此测试集主要用于测试LD、INC、SUB、ST、ADD、JC、AND、OUT、STP指令，指令如**表 ??**所示，初始状态下寄存器 R2 = 12H，R3 = 0FH，存储器 [0FH] = 85H，[10H] = 23H，[11H] = EFH。

表 8: 测试集 1

地址	指令	机器码	功能	结果
00H	LD R0, [R3]	0101 0011	R0 ← [R3]	R0 ← 85H
01H	INC R3	0100 1100	R3 ← R3 + 1	R3 ← 10H
02H	LD R1, [R3]	0101 0111	R1 ← [R3]	R1 ← 23H
03H	SUB R0, R1	0010 0001	R0 ← R0 - R1	R0 ← 62H
04H	JZ 0BH	1000 0110	如果 Z=1， 则 PC ← @ + 06H	Z=0， 不跳转
05H	ST R0, [R2]	0110 1000	R0 → [R2]	62H → [12H]
06H	INC R3	0100 1100	R3 ← R3 + 1	R3 ← 11H
07H	LD R0, [R3]	0101 0011	R0 ← [R3]	R0 ← EFH
08H	ADD R0, R1	0001 0001	R0 ← R0 + R1	R0 ← 12H
09H	JC 0CH	0111 0010	如果 C=1， 则 PC ← @ + 02H	C=1， PC ← 0CH
0AH	INC R2	0100 1000		
0BH	ST R2, [R2]	0110 1010		
0CH	AND R0, R1	0011 0001	R0 ← R0 ∧ R1	R0 ← 02H
0DH	OUT R2	1010 0010	DBUS ← R2	DBUS ← 12H
0EH	STP	1110 0000	暂停运行	

4.1.2 测试集 2

此测试集主要用于测试JC指令，指令如**表 ??**所示，初始状态下寄存器 R2 = 12H，R3 = 0FH，存储器 [0FH] = 23H，[10H] = 23H，[11H] = EFH。

4.1.3 测试集 3

此测试集主要用于测试不同的分支跳转，指令如**表 ??**所示，初始状态下寄存器 R2 = 12H，R3 = 0FH，存储器 [0FH] = 85H，[10H] = 23H，[11H] = 01H。

4.1.4 测试集 4

此测试集主要用于测试CMP、MOV、OR指令，指令如**表 ??**所示，初始状态下寄存器 R2 = 12H，R3 = 0FH，存储器 [0FH] = 85H，[10H] = 23H，[11H] = E9H。

表 9: 测试集 2

地址	指令	机器码	功能	结果
00H	`LD R0, [R3]`	0101 0011	R0 ← [R3]	R0 ← 23H
01H	`INC R3`	0100 1100	R3 ← R3 + 1	R3 ← 10H
02H	`LD R1, [R3]`	0101 0111	R1 ← [R3]	R1 ← 23H
03H	`SUB R0, R1`	0010 0001	R0 ← R0 - R1	R0 ← 00H
04H	`JZ 0BH`	1000 0110	如果 Z=1， 则 PC ← @ + 06H	Z=1， PC ← 0BH
05H	`ST R0, [R2]`	0110 1000		
06H	`INC R3`	0100 1100		
07H	`LD R0, [R3]`	0101 0011		
08H	`ADD R0, R1`	0001 0001		
09H	`JC 0CH`	0111 0010		
0AH	`INC R2`	0100 1000		
0BH	`ST R2, [R2]`	0110 1010	R2 → [R2]	12H → [12H]
0CH	`AND R0, R1`	0011 0001	R0 ← R0 ∧ R1	R0 ← 00H
0DH	`OUT R2`	1010 0010	DBUS ← R2	DBUS ← 12H
0EH	`STP`	1110 0000	暂停运行	

表 10: 测试集 3

地址	指令	机器码	功能	结果
00H	`LD R0, [R3]`	0101 0011	R0 ← [R3]	R0 ← 85H
01H	`INC R3`	0100 1100	R3 ← R3 + 1	R3 ← 10H
02H	`LD R1, [R3]`	0101 0111	R1 ← [R3]	R1 ← 23H
03H	`SUB R0, R1`	0010 0001	R0 ← R0 -R1	R0 ← 62H
04H	`JZ 0BH`	1000 0110	如果 Z=1， 则 PC ← @ + 06H	Z=0， 不跳转
05H	`ST R0, [R2]`	0110 1000	R0 → [R2]	62H → [12H]
06H	`INC R3`	0100 1100	R3 ← R3 + 1	R3 ← 11H
07H	`LD R0, [R3]`	0101 0011	R0 ← [R3]	R0 ← 01H
08H	`ADD R0, R1`	0001 0001	R0 ← R0 + R1	R0 ← 24H
09H	`JC 0CH`	0111 0010	如果 C=1， 则 PC ← @ + 02H	C=0， 不跳转
0AH	`INC R2`	0100 1000	R2 ← R2 + 1	R2 ← 13H
0BH	`ST R2, [R2]`	0110 1010	R2 → [R2]	13H → [13H]
0CH	`AND R0, R1`	0011 0001	R0 ← R0 ∧ R1	R0 ← 20H
0DH	`OUT R2`	1010 0010	DBUS ← R2	DBUS ← 13H
0EH	`STP`	1110 0000	暂停运行	

表 11: 测试集 4

地址	指令	机器码	功能	结果
20H	LD R0, [R3]	0101 0011	R0 ← [R3]	R0 ← 85H
21H	INC R3	0100 1100	R3 ← R3 + 1	R3 ← 10H
22H	LD R1, [R3]	0101 0111	R1 ← [R3]	R1 ← 43H
23H	CMP R0, R1	1100 0001	R0 - R1	
24H	JZ 0BH	1000 0110	如果 Z=1， 则 PC ← @ + 06H	Z=0， 不跳转
25H	MOV R0, R2	1101 1000	R0 → R2	85H → R2
26H	INC R3	0100 1100	R3 ← R3 + 1	R3 ← 11H
27H	LD R0, [R3]	0101 0011	R0 ← [R3]	R0 ← E9H
28H	ADD R0, R1	0001 0001	R0 ← R0 + R1	R0 ← 2CH
29H	JMP [R0]	1001 0010	PC ← R0	PC ← 2CH
2AH	INC R2	0100 1000		
2BH	ST R2, [R2]	0110 1010		
2CH	OR R0, R1	1011 0001	R0 ← R0 ∨ R1	R0 ← 6FH
2DH	OUT R2	1010 0010	DBUS ← R2	DBUS ← 12H
2EH	STP	1110 0000	暂停运行	

4.1.5 测试集 5

此测试集主要用于测试分支跳转，指令如**表 ??**所示，初始状态下寄存器 R2 = 60H，R3 = FDH，存储器 [60H] = 67H，[61H] = 80H，[62H] = FDH，[80H] = 60H，[FEH] = 03H，[FFH] = 03H。

4.1.6 测试集 6

此测试集主要用于测试中断。进入程序，设置 PC 为 01H，DP 设为 0，按下 QD，在任意时刻按下 PULSE，进入中断，在开关设置中断服务程序入口地址为 45H，之后按下 QD 进入中断服务子程序。中断服务子程序执行完毕后，程序在 IRET 指令的 W3 拍暂停，此时可以确认断点和之前相同，此后按下 QD，返回断点处继续执行主程序。指令如**表 ??**所示。初始状态下寄存器 R0 = 00H，R1 = 01H。

4.2 调试过程中遇到的问题及解决方案

4.2.1 基础功能——标志位设置

在读写寄存器和存储器的过程中，程序完全无法正常运行，经讨论，发现 ST0 标志位的变化不能在节拍电位的中间时刻进行，而应该在 T3 脉冲的下降沿，即一个节拍电位的结束时刻进行。如果在节拍电位的中间时刻变化了 ST0，则发出的信号可能发生错误。修复了这个问题之后我们可以正常地读写寄存器和存储器。

表 12: 测试集 5

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			

表 13: 测试集 6

地址	指令	机器码	功能	结果
51H	INC R0	0100 0000	R0 ← R0 + 1	
52H	INC R0	0100 0000	R0 ← R0 + 1	
53H	INC R0	0100 0000	R0 ← R0 + 1	
54H	INC R0	0100 0000	R0 ← R0 + 1	
55H	INC R0	0100 0000	R0 ← R0 + 1	
56H	INC R0	0100 0000	R0 ← R0 + 1	
57H	INC R0	0100 0000	R0 ← R0 + 1	
58H	INC R0	0100 0000	R0 ← R0 + 1	
59H	JMP [R1]	1001 1101	PC ← R1	PC ← 51H
45H	ADD R0, R0	0001 0000	R0 ← R0 + R0	
46H	IRET	1111 0000	返回断点	

4.2.2 基础功能——程序 bug

在进行第一次测试时发现测试未通过，寄存器的值与预期不一致，经检查是 DRW 信号的逻辑表达式中缺少了`LD and R3`项，此外，在 M 信号的表达式中，AND_I 项被写成了 ADD。更正了两个 bug 之后通过了测试集 1。

4.2.3 基础功能——缺少引脚

测试集 2 未通过，经检查是 Z 信号的引脚没有设置，设置引脚之后通过测试。

4.2.4 基础功能——完善节拍信号

在测试程序时，我们发现初始阶段下（即按下 CLR 之后），此时 W1、W2、W3 均为 0，但设置 SWC SWB SWA 之后仍然有输出信号，经检查是程序设计时的漏洞：程序设计时没有考虑到三个节拍电位信号都为 0 的情况，于是部分单节拍的信号没有与`W1`，部分在 W1、W2 都出现的信号也没有与`W1 and W2`。因此，给所有这样的信号都与相应的节拍电位后，前述错误被修复。

在实现用户指定 PC 功能时犯了同样的错误，给 PC 赋初值的两个节拍电位的信号没有与 W1 或者与 W2，更正之后问题被修复。

4.2.5 基础功能——写入指令错误

在测试时，误将 JC 指令的操作码写错，导致测试样例不能通过，经检查存储器后发现该问题，之后重新写入指令后修复。

4.2.6 流水线——取指令错误

在连续执行状态下执行测试集 5，执行结果不正确，表现为 3 个寄存器的值全都一样。改为单拍执行，跟踪每一步的执行结果，后发现在执行在 0FH 的 SUB 指令时取指令错误。巨日表现为：在 PC 为 0FH，INS7~INS0 显示 0010 0010，即 SUB 指令的机器码，此时 LIR 和 PCINC 指示灯亮，按下 QD，PC 的值变为 10H，理论上完成取指令后 IR7~IR0 的值应为 SUB 的机器码 0010 0010，然而此时 IR7~IR0、INS7~INS0 的值均为 0101 1000，即 10H 处的指令 LD 的机器码，取出了错误的指令。

我们检查存储器的对应地址发现该条指令的机器码没有写错，分析是取指令信号出现问题。之后我们依次进行如下尝试：

1. 使用非流水的程序再次执行程序，通过测试；
2. 在另一地址测试该程序，bug 出现；
3. 通过指定 PC，直接从 OEH 开始执行程序，bug 出现；
4. 将 0EH 处的指令 INC 更换 NOP，bug 未出现；
5. 将 10H 处的指令 LD 更换 NOP，bug 未出现；
6. 将 SUB 指令改为 ADD 和 OR 指令，bug 未出现；
7. 将 SUB 指令的操作数改为 R1 和 R3 寄存器，bug 未出现。

由此，我们推测是由于前后的指令访问相同的寄存器，出现寄存器访问冲突导致取指令失败，我们将 PCINC 和 LIR 改为在 T3 的上升沿执行，写入后 bug 仍然出现，此时我们发现之前多次烧录程序之前没有按 Auto Detect 按钮，导致每次烧录的不是最新的程序。

按 Auto Detect 按钮之后重新写入，问题解决。

4.2.7 中断——中断无法触发

在编写完成中断程序后，我们发现无论什么时机按下 PULSE，都无法触发中断，于是我们尝试观察 PULSE 信号。通过在实验台上连接彩虹线，我们可以通过观察交通灯（或者数码管）的值来确认某个信号是否触发。在之后的调试中，我们通过设置一个调试信号，将其接入 17 引脚，之后查看数码管 LG3 的值来观察某个信号是否触发。

4.2.8 中断——VHDL 语法问题

VHDL 语法中不容许同时监测两个时序，因此判断条件中不能同时写两个时钟沿。

4.2.9 中断——中断周期执行错误

初次实现终端时，我们参考微程序的流程，采用 INT 标志来标记中断周期，然而中断周期的第一个节拍电位就会关中断，导致 INT 标志变为 0 而立即退出中断周期。因此我们类比 ST0 标志，又设置了一个 ST1 标志来表示中断周期，在每条指令的结束时通过检查 INT 标志来设置 ST1 标志，在中断周期的最后将 ST1 标志设为 0 来表示结束中断周期。

4.2.10 中断——返回断点错误

在中断服务程序结束运行结束之后，检查 PC 的值发现并没有返回之前的断点，跟踪 R3 的值时发现了漏洞：执行中断服务程序时 R3 的值仍然和 PC 同步，导致断点未被 R3 记录。因此我们将 PCINC 和 JMP 指令的条件进行修改，当 EN_INT 为 0 时（即正在执行中断服务程序）时 R3 的值保持不变。

4.2.11 中断——指令优化

在 IRET 指令的 W2 拍加入 STOP 信号，以便展示断点的恢复过程。

4.2.12 版本 2——行为描述

编写行为描述方式时，一开始认为根据操作码进入分支后就可以指定 M 和 S。但在编写 ST 指令时发现，W2 时 S=1111，W3 时 S=1010。因此，把所有的 S 改成由 W2 和 W3 控制的信号。

在编写 CMP 和 SUB 指令时，因为 CMP 指令和 SUB 指令非常接近，错给 CMP 增加 DRW 信号，在测试时发现该错误并且改正。

4.3 实验总结与心得

毛子恒 在理论课中，由于缺少上机时间，我对硬布线控制方式不甚理解。计组课程设计给我亲手设计一个硬布线控制器的契机，加深了我对整个设计流程——设计指令、画流程图、写译码表、编写程序——的体会，之后在实现基础功能的两个特殊需求的时候，我对这一套流程的应用更加熟练。

实验的第二步是设计流水线，由于指令集比较简单，我们也只实现了一个 2 级流水。实现时发现由于很难出现资源冲突的情况，所以流水线不需要处理太复杂的情况，最后编写程序时也较为轻松。只是在测试时发现了寄存器访问冲突的情况，经检查还是我们程序编写的漏洞造成的。

最后是中断的设计，由于引脚的缺失，我们并不能像微程序控制方式那样简单地实现中断，因此我们在与老师交流之后决定采用 R3 寄存器同步 PC 寄存器的方式实现中断。在中断功能设计的过程中，我们就遇到许多小问题，比如如何利用 SELCTL 和 SEL 信号的功能等等，这些细节的实现需要对 TEC-8 模型机非常熟悉，对数据通路和运算器的理解透彻才可以进行。幸运的是，我们最终克服了种种困难，通过对原有指令集的取舍和修改，成功完成中断功能。

课程设计使我大大加深了对整个模型机的理解，对理论课的知识也有了更深的体会，在之前的实验课中遗留下来的有关数据通路和中断的问题也都在课程设计中被重新提出、消化和理解。

最后十分感谢我的非常靠谱的队友们，我们组内分工十分明确，他们非常高效率地完成了测试、绘图和文档中大部分文字内容的撰写工作，高效协作也使得我可以专注于我的工作内容，最终在有限的时间内我们完成了所有的功能需求，并且确保每个组员都理解了程序的内容。

姜山 通过本次计组课程设计，我们加深了对硬布线控制器的理解，对 TEC-8 实验台的掌握也更加完整。本次实验的关键点在于 TEC-8 模型计算机的整体理解，以及硬布线控制器涉及到的各个控制信号，同时也对我们编写代码、调试代码的能力有更高的要求。与计算机组成原理实验相比，在 Quartus 上编程并烧录芯片的方式比直接采用 TEC-8 实验台微程序控制模式更加深入，如果对寄存器、存储器、运算器、数据通路、时序发生器甚至选择器等知识点一知半解，则很难理解程序原理和调试；与数字逻辑试验相比，硬连线控制器的课题主要围绕 CPU 设计原理，与计算机组成原理理论课程紧密联系，并且时序逻辑控制方式对编码能力要求更高。

在本次实验过程中，我们灵活运用多种程序调试方式：将出错指令的前后指令设为空、通过设置初始 PC 跳转至出错位置运行、将监视的信号赋值至调试变量分配管脚……当然，其中最基础也是重要的还是查看几个关键信号的指示灯，检查指令寄存器、地址寄存器和数据通路中是否出现异常。调试程序、发现错误、找出根源的过程锻炼了我们遇到问题坚持不懈的品质，增强了我们面对程序时的耐心和毅力。

同时，本次实验还培养了我们的发散性思维。在基础功能实现时，我们分别编写了总体时序内部串行执行的程序和总体并行执行包含时序的程序。虽然我们采用了后者，但这一过程提高了我们的编码能力，锻炼了我们的思维。在中断功能实现时，我们曾经因为没有中断地址寄存器 IAR 的控制信号而一筹莫展，而后将寄存器 R3 取代的想法出现，最终将难以攻克的单级终端功能实现并调试成功。在此过程中我们碰撞出了思维的火花，相互帮助学习，每一位成员

都受益良多。

赵雨腾 在计组课设中我对组成原理这门课的知识又有了更近一步的认识。从 VHDL 的复习，我对硬件编程语言 VHDL 有了进一步的学习，对数据流描述、行为描述理解更加深刻。在程序编写过程中，我查阅了 QuartusII 的 ieee.std_logic_1164. 的 vhd 文件，研究了他们写程序的风格，学习了他们，并应用到我们的程序中。在先前的运算器、CPU 组成、中断这些组成原理实验中，我只了解脉冲和节拍，但在本次实验中，我们需要对所有的器件控制信号进行时序分析，将每一个控制信号安排到相应的位置上，以保证有序工作。在流水程序的设计中，我学到了将取址和执行指令放在同一拍。有的时候需要注意访问冲突，控制好时序。在中断过程中，我们设想在程序里跟踪 PC，到后来用 R3 跟踪 PC。但是要注意，R3 在进入中断的时候不需要再跟踪 PC，中断结束之后，继续跟踪 PC。学习中断隐指令需要保存断点、关中断、进入中断服务子程序。

实验过程因为自己的粗心，犯了很多错误：基础测试过程中，机组做实验过中，管脚分配有时候会漏，AND 和 ADD 指令二进制机器指令操作码被写反，MOv 写成 SUB 指令的二进制机器指令操作码。测试中断时，我们基本上每 1min 就会出一个 bug，然后和子恒一 debug。测试其他时，在 7 号我们没有能一次通过 7 个测试的时候。在实验中，遇到的硬件挫折比软件挫折更折磨人。

最后，我们收获了充实的 7 天，每个队友都完成了关键的环节。除了知识点收获，我们还收获了同学之间的友谊。感谢老师指导、感谢队友力挺！

刘洋 在做计组实验的过程中，我心情的起伏大致可以用一个 U 型来表示，刚开始听到任务的时候，觉得这就是把计组实验课上做的“CPU 组成与机器指令的执行“实验复现一遍，但是随着不断深入理解实验原理，以及每天在实验室调试总是遇到其奇奇怪怪的 BUG 的时候，心情逐渐失落了起来。不过办法总比困难多，在这里要非常感谢我的队友们，家人们能力真的很强，而且责任心也很强，即使我们组遇到很多困难，但最后都摸着石头过河，不断试错，硬着头皮解决了。到后来，调试是一种折磨也是一种快乐，心情又好了起来。

虽然实验的时间很短，只有一周，但是在这一周的时间里学到了很多东西。在知识方面，有很多疑问，例如 SELCTL 信号是干什么的？SEL3~SEL0 怎么选择送 ALU 的 A、B 端口的寄存器？当时上机组理论课包括计组实验课时候没有完全懂的地方，全都在这次实验中完美的暴露了出来，所谓出来混总是要还的，没学明白的知识最后总是要花更多的时间搞明白。这次试验和我们曾经做过的“CPU 组成与机器指令的执行”很像，但难度却不是一个级别的，我们之前是利用 TEC-8 的已经内置好的环境来模拟 CPU 取指令执行指令的功能，而现在是需要自己写搭建环境，然后在此基础上执行自己的测试程序，属于是全程都需要自己动手了。但是整个流程走下来，会更加清晰的了解 CPU 取指令和执行每条不同指令的过程，对于 TEC-8 的环境也是更加熟悉。TEC-8 试验台是我们的老朋友了，伴随我们一直从数字逻辑实验再到计算机组成原理的实验，再到现在的计算机组成原理课程设计，一路走来我们逐渐揭开了它的神秘面纱。此次实验给我们留下了难以磨灭的深刻印象，加深了我们对 CPU 设计与实现原理的理解，大家受益匪浅。

A 日志

A.1 7 月 2 日

布置任务的第一天，我们首先根据理论课的教学内容，认真学习了 PPT。同时，我们也复习了 VHDL 语言的语法和 TEC-8 实验台原理，其中重点复习了本学期计算机组成原理实验课上所讲的运算器组成实验、双端口存储器实验、数据通路实验和 CPU 组成与机器指令的执行实验，并参考了计算机组成原理实验指导书。

完成复习后，我们通过编写与门程序测试了 TEC-8 试验台芯片能否正常烧录程序。我们还对 TEC-8 实验台的寄存器、存储器、数据通路还有其他信号进行了测试，包括中断信号 PULSE。我们大致安排了五天时间的计划，先完成基本功能（包括指定 PC 和扩指功能）和流水线功能，经过测试两者均无误后，在有余力的情况下考虑单级中断功能的完成。

A.2 7 月 3 日

在充分复习了 VHDL 语言以及计组知识的基础上，我们已经对这个实验的原理以及设计流程有了很好的把握，可以说是跃跃欲试了，按照实验指导书上的参考流程，我们先设计了针对基础指令的机器指令周期流程图，然后根据流程图画出了组合译码逻辑表，最后根据译码表编写程序，整个流程走下来我们对于自己编写的程序实在是很满意，然后一起去实验室进行测试。然而硬件编程与软件调试的差别就像梦想与现实的差别一样大，我们发现编程编好了，仅仅是万里长征第一步，遇到的奇奇怪怪的 BUG 实在是让人摸不着头脑。详情见遇到的问题之基础功能。

A.3 7 月 4 日

所谓万事开头难，昨天将基础指令完成算完成了入门任务，然后就在此基础上不断扩展，不断新增 BUG，不断 DEBUG，今天我们编写了流水线功能，流水线与非流水线相比，主要的难点在设计上，程序的编写不过是和昨天基础指令一样的模板，但关键在于组合译码逻辑表的编写，这需要深入的理解 CPU 流水实现的原理。流水在每条指令的执行的时候取下一条指令，有效的缩短了指令需要的机器周期。

在程序编写完成后进行了测试，不出所料，测试的时候也遇到很多的问题。最大的问题是测试时 INC 指令执行后，转为执行 SUB 指令时，SUB 指令后面的 LD 指令会冲掉前面的 SUB 指令，令人百思不得其解。经过思考和交流，我们参考 TEC-8 模型计算机框图，考虑到取指时可能产生的冲突问题，解决方法则是将取指指令 PCINC 和 LIR 放在 T3 上升沿时序中实现。详情见遇到的问题之流水。

A.4 7 月 5 日

不得不承认写文档是一种能力，之前在设计的时候写的都是草稿和大白话，这些东西是上不了台面的，我们需要把它转换为规范的表述，才能放到报告里。需要用到专业画图的工具，绘

制表格工具等。对于表格部分，我们使用 Excel 绘制；对于流程图部分，我们使用 ProcessOn 绘制。代码有所修改后图表也进行相应地修改。

在仔细阅读 TEC-8 组成原理指导书后，我们编写了文档的原理和设计部分，既参考了老师指定的要求，也结合了自己的思考成果。

按照我们的计划，今日的主要难题是中断代码的编写。虽然已经明了了放弃 IAR 并用 R3 代替的技术路线原理，但在没有逻辑图表的情况下，设计中断相关指令难上加难，全凭临时的思维火花转化成的草稿。但是我们并没有放弃或气馁，而是深知实践出真知的道理，在不断编码和测试中，终端程序迎来了一个又一个迭代版本，过程艰难但由于思维投入，许多设计上的问题和小错误得以及时修改。编写过程中我们同时进行了测试，详情见遇到的问题之中断前半部分。

A.5　7 月 6 日

验收前的最后一天，所有工作都进入了收尾阶段。首先对编写好的中断代码进行进一步测试，通过仔细调试对出现的错误情况进行复现，成功使终端服务程序正常运行并返回中断地址。同时，我们也对相关图表进行了及时修改。之后，我们对行为描述方式的代码也进行了测试，使用非流水的测试集。测试过程中解决的问题详情见遇到的问题之中断后半部分和版本 2——行为描述。

我们集中开会编写文档的剩余部分，统一调整文档格式，发现需要修改之处及时修改，并且相互沟通讨论。讨论中发现程序需要改动立即做出修改，并前往实验室烧录测试。在此期间，我们形成了本次实验的自我总结，并相互分享本次课程设计过程中的心得体会。最终组长对本次组成原理课程设计作总结发言，每一位组员均感到受益良多，在本次实验中感触颇深。