

## 1 环境描述

## 2 题目要求

## 3 设计详解

### 3.1 指令架构

基本指令集 (12)

扩展指令集 (4)

扩展指令集仿真

### 3.2 非流水线硬布线控制器详解

基本原理

指令节拍电位

引脚图和连接

引脚设计图

引脚连接

非流水线控制器指令译码表

非流水线方案流程图

修改pc指针功能

### 3.3 流水线硬布线控制器详解

流水线方案实现

流水线方案流程图

## 4 测试与验证

### 4.1 测试程序

寄存器初值

存储器初值

执行过程

### 4.2 斐波那契数列

寄存器初值

执行过程

## 5 问题和解决

### 5.1 烧写程序BUG

### 5.2 执行MOV指令错误

### 5.3 流水版本指令覆盖问题

## 6 中断设计与测试方案

### 6.1 中断原理与需求分析

### 6.2 指令与信号设计

指令设计

信号设计

### 6.3 硬件设计

### 6.4 中断流程图

### 6.5 VHDL程序设计-中断

## 7 附录

### 7.1 工程进度日志

8月10日 初出茅庐

8月15日 深入学习

8月16日 撸起袖子开始干

8月17日 计日程功，进度可喜

9月2日 牛刀小试，BUG不断

9月3日 高效协作，顺利验收

9月4日 总结复盘，收获满满

### 7.2 小组成员及心得体会

小组成员名单

成员心得体会

周鹏飞

苟志斌

金浩男

- 1 硬连线控制器（VHDL）
- 2 流水硬连线控制器（VHDL）



- **TEC-8**控制台
- **TEC-8**程序下载线
- **Quartus 9.0** 编程环境

## 2 题目要求

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：

- 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。
- 能够完成规定的指令功能。
- 在Quartus II 下对硬连线控制进行编程和编译。
- 将编译后的硬连线控制器下载到TEC-8 实验台上的可编程器件EPM7128S 中去，使EPM7128S 成为一个硬连线控制器。
- 根据指令系统，编写检测硬连线控制器正确性的测试程序，并用测试程序对硬连线控制器在单微指令方式下进行调试，直到成功。
- 在此基础上，扩展指令集，并实现修改pc指针的功能。
- 在此基础上，实现流水线硬连线控制器。

## 3 设计详解

### 3.1 指令架构

#### 基本指令集 (12)

名称	功能	IR7 IR6 IR5 IR4	IR3 IR2	IR1 IR0
ADD Rd, Rs	$Rd \leftarrow Rd + Rs$	0001	Rd	Rs
SUB Rd, Rs	$Rd \leftarrow Rd - Rs$	0010	Rd	Rs
AND Rd, Rs	$Rd \leftarrow Rd \text{ and } Rs$	0011	Rd	Rs
INC Rd	$Rd \leftarrow Rd + 1$	0100	Rd	XX
LD Rd, [Rs]	$Rd \leftarrow [Rs]$	0101	Rd	Rs
ST Rd, [Rs]	$[Rs] \leftarrow Rd$	0110	Rd	Rs
JC offset	若C=1, 则 $PC \leftarrow @ + offset$	0111	offset	
JZ offset	若Z=1, 则 $PC \leftarrow @ + offset$	1000	offset	
JMP Rd	$PC \leftarrow Rd$	1001	Rd	XX
OUT Rs	$DBUS \leftarrow Rs$	1010	XX	Rs
STP	暂停运行程序	1111	XX	XX
NOP	什么也不做	0000	XX	XX

基本指令集满足了课程设计的指令集要求。

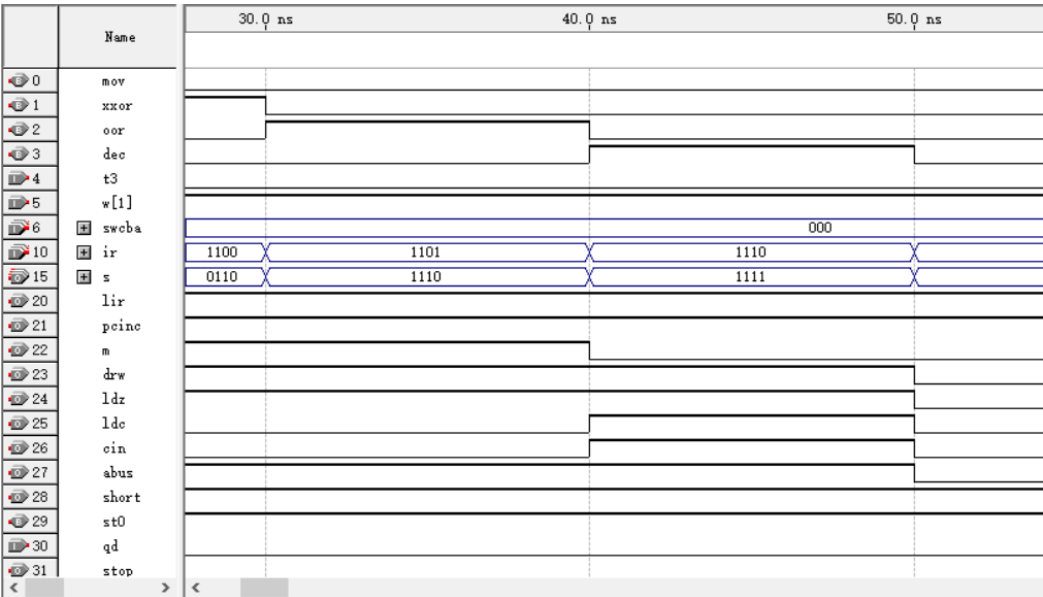
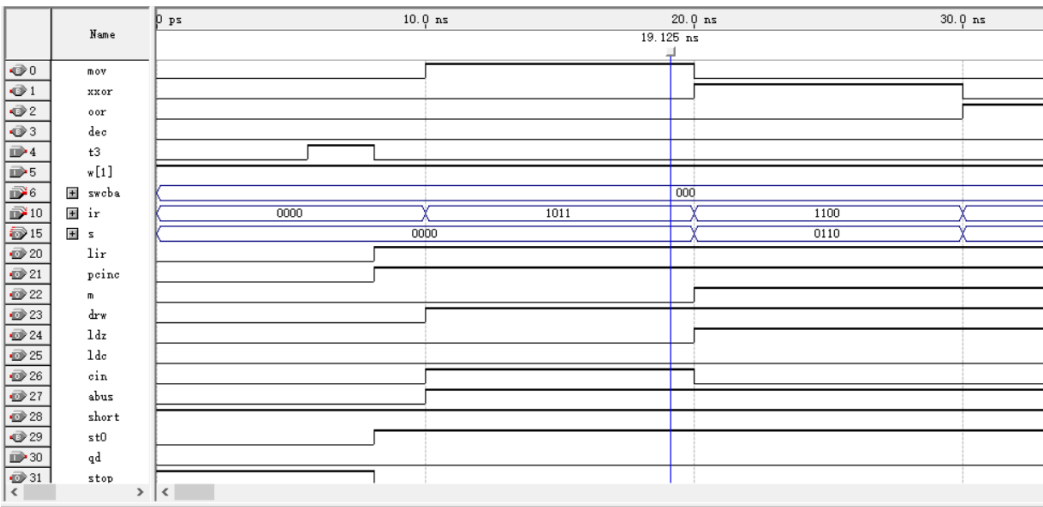
1. 由于STP指令的特殊性，我们修改了**STP**指令的代码，由原来的**1110**修改成**1111**，使其更具有辨识度。
2. 指令集要求中隐含了**NOP**指令，这里我们将NOP指令显式地列出，代码为**0000**。

扩展指令集 (4)

名称	功能	IR7 IR6 IR5 IR4	IR3 IR2	IR1 IR0
MOV Rd Rs	Rd<-Rs	1011	Rd	Rs
XOR Rd, Rs	Rd<-Rd xor Rs	1100	Rd	Rs
OR Rd, Rs	Rd<-Rd or Rs	1101	Rd	Rs
DEC Rd	Rd<-Rd-1	1110	Rd	XX

我们充分利用了4位的指令代码，添加了4条扩展指令：**MOV, XOR, OR, DEC.**

扩展指令集仿真



上面两张图片是流水线模式下对四条扩展指令的仿真模拟。  
输入swcba = "000", 表示一直是“运行程序”模式。  
指令ir每10ns发生变化。前10ns为默认值0000， 从10ns到50ns， 每10ns代表一个指令， 变化顺序为**mov, xor, or, dec.**  
信号t3在前10ns内发生一次跳变，更改st0的值，进入执行程序阶段。  
由于四条指令都是短指令，所以时序信号只设置W<sub>1</sub>, 并且一直处于高有效状态。

观察输出信号，可以看到信号**mov, xor, oor, dec**依次经历了高有效的状态，持续时间都为10ns。对于控制台输出信号**lir, pcinc, m, drw**等等的模拟结果，我们对照指令译码表之后肯定了其正确性。

## 3.2 非流水线硬布线控制器详解

### 基本原理

硬连线控制器的基本原理，每个微操作控制信号 $S$ 是一系列输入量的逻辑函数，即用组合逻辑来实现， $S = f(I_m, M_i, T_k, B_j)$ ，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 $T_k(T_1 \sim T_3)$ 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 $IR_4 \sim IR_7$ 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 $W_1 \sim W_3$ ； $B_j$ 包括ALU产生的进位信号C、结果为0信号Z等等。

### 指令节拍电位

TEC-8系统中,一条指令可能需要多个节拍电位来执行。

大部分指令需要两个节拍电位。而在正常情况下，实验台发出的电位会在 $W_1$ 和 $W_2$ 之间跳变。

有一部分指令只需要一个节拍电位，比如读存储器和写存储器的指令。这些指令在执行的时候会在第一个节拍电位中使**short**信号有效，然后控制台就不会再发出 $W_2$ 信号。

有一些指令需要三个节拍电位来执行，比如**ST**信号和**LD**信号。这些指令在执行的时候会在第二个节拍电位使**long**信号有效，然后控制台会发出 $W_3$ 信号，延长一个周期。

写寄存器的指令需要四个周期来完成。这时，控制台使用**st0**信号将四个周期划分为两条两周期的指令（**st0**是内部信号，不会出现在控制台的输入或输出端口）。当然**st0=0**的时候，控制台用两周期写入寄存器**R0**和**R1**，之后被设置成**st0=1**，并且不再改变。。当**st0=1**的时候，控制台用两周期写入寄存器**R2**和**R3**。

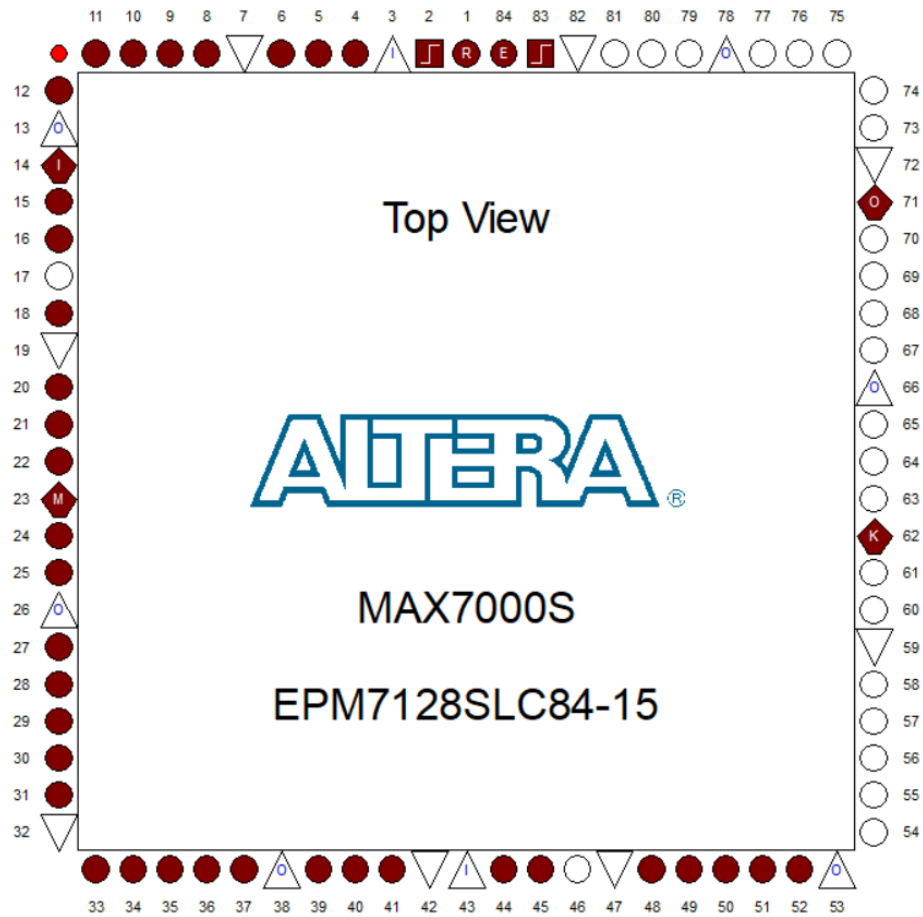
**st0**信号同样也是用来区分初始阶段和进行阶段的标志信号。**st0=0**表示初始阶段，**st0=1**表示进行阶段。

在读存储器和写存储器两个控制台程序中,**st0=0**时，控制台执行的是将首地址写入**AR**寄存器的指令，之后被设置成**st0=1**，并且不再改变。**st0=1**时，控制台开始一个地址一个地址的依次读或写计算器。








































在运行程序时，**st0=0**的时候需要将程序的入口地址写入**pc**寄存器，之后被设置成**st0=1**，并且不再改变。**st0=1**的时候，依次取指令并执行。

引脚图和连接

引脚设计图



引脚连接

	Node Name	Direction	Location
1	 abus	Output	PIN_40
2	 arinc	Output	PIN_24
3	 c	Input	PIN_2
4	 cin	Output	PIN_33
5	 clr	Input	PIN_1
6	 drw	Output	PIN_20
7	 ir[3]	Input	PIN_11
8	 ir[2]	Input	PIN_10
9	 ir[1]	Input	PIN_9
10	 ir[0]	Input	PIN_8
11	 lar	Output	PIN_25
12	 ldc	Output	PIN_31
13	 ldz	Output	PIN_30
14	 lir	Output	PIN_29
15	 lpc	Output	PIN_22
16	 m	Output	PIN_39
17	 mbus	Output	PIN_44
18	 memw	Output	PIN_27
19	 pcadd	Output	PIN_18
20	 pcinc	Output	PIN_21
21	 s[3]	Output	PIN_37
22	 s[2]	Output	PIN_36
23	 s[1]	Output	PIN_35
24	 s[0]	Output	PIN_34
25	 sbus	Output	PIN_41
26	 sel[3]	Output	PIN_51
27	 sel[2]	Output	PIN_50
28	 sel[1]	Output	PIN_49
29	 sel[0]	Output	PIN_48
30	 selctl	Output	PIN_52
31	 short	Output	PIN_45
32	 stop	Output	PIN_28
33	 swcba[3]	Input	PIN_6
34	 swcba[2]	Input	PIN_5
35	 swcba[1]	Input	PIN_4
36	 t3	Input	PIN_83
37	 w[3]	Input	PIN_16
38	 w[2]	Input	PIN_15
39	 w[1]	Input	PIN_12

非流水线控制器指令译码表

	RUN 000	WMEM 001	RMEM 010	RREG 011	WREG 100
SBUS	$!ST_0 * W_1$	$W_1$	$!ST_0 * W_1$		$W_1 + W_2$
SELCTL		$W_1$	$W_1$	$W_1 + W_2$	$W_1 + W_2$
SEL3				$W_2$	$ST_0 * (W_1 + W_2)$
SEL2					$W_2$
SEL1				$W_2$	$ST_0 * W_2 + !ST_0 * W_1$

	RUN 000	WMEM 001	RMEM 010	RREG 011	WREG 100
SELO				$W_1 + W_2$	$W_1$
DRW					$W_1 + W_2$
SHORT	$!ST_0 * W_1$	$W_1$	$W_1$		
STOP	$!ST_0 * W_1$	$W_1$	$W_1$	$W_1 + W_2$	$W_1 + W_2$
LAR		$!ST_0 * W_1$	$!ST_0 * W_1$		
SST0	$!ST_0 * W_1$	$!ST_0 * W_1$	$!ST_0 * W_1$		$!ST_0 * W_2$
LPC	$!ST_0 * W_1$				
MEMW		$ST_0 * W_1$			
ARINC		$ST_0 * W_1$	$ST_0 * W_1$		
MBUS			$ST_0 * W_1$		

	ADD 0001	SUB 0010	AND 0011	XOR 1100	OR 1101
LIR	$W_1$	$W_1$	$W_1$	$W_1$	$W_1$
PCINC	$W_1$	$W_1$	$W_1$	$W_1$	$W_1$
M			$W_2$	$W_2$	$W_2$
CIN	$W_2$				
S3	$W_2$		$W_2$		$W_2$
S2		$W_2$		$W_2$	$W_2$
S1		$W_2$	$W_2$	$W_2$	$W_2$
S0	$W_2$		$W_2$		
ABUS	$W_2$	$W_2$	$W_2$	$W_2$	$W_2$
DRW	$W_2$	$W_2$	$W_2$	$W_2$	$W_2$
LDZ	$W_2$	$W_2$	$W_2$	$W_2$	$W_2$
LDC	$W_2$	$W_2$			
LONG					
LAR					
MBUS					
MEMW					
PCADD					
LPC					
STOP					

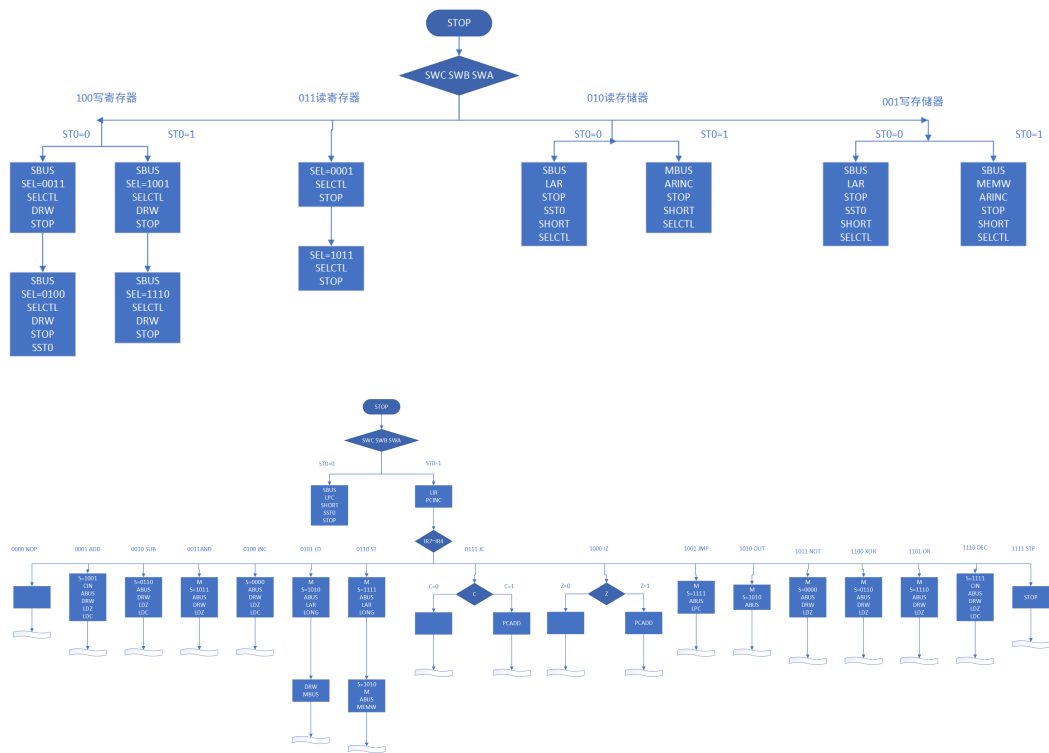
	INC 0100	DEC 1110	LD 0101	ST 0110	MOV 1011
LIR	$W_1$	$W_1$	$W_1$	$W_1$	$W_1$
PCINC	$W_1$	$W_1$	$W_1$	$W_1$	$W_1$
M			$W_2$	$W_2 + W_3$	$W_2$
CIN		$W_2$			
S3		$W_2$	$W_2$	$W_2 + W_3$	$W_2$



	INC 0100	DEC 1110	LD 0101	ST 0110	MOV 1011
S2		$W_2$		$W_2$	
S1		$W_2$	$W_2$	$W_2 + W_3$	$W_2$
S0		$W_2$		$W_2$	
ABUS	$W_2$	$W_2$	$W_2$	$W_2 + W_3$	$W_2$
DRW	$W_2$	$W_2$	$W_3$		$W_2$
LDZ	$W_2$	$W_2$			
LDC	$W_2$	$W_2$			
LONG			$W_2$	$W_2$	
LAR			$W_2$	$W_2$	
MBUS			$W_3$		
MEMW				$W_3$	
PCADD					
LPC					
STOP					

	JC 0111	JZ 1000	JMP 1001	OUT 1010	STP 1111	NOP 0000
LIR	$W_1$	$W_1$	$W_1$	$W_1$	$W_1$	$W_1$
PCINC	$W_1$	$W_1$	$W_1$	$W_1$	$W_1$	$W_1$
M			$W_2$	$W_2$		
CIN						
S3			$W_2$	$W_2$		
S2			$W_2$			
S1			$W_2$	$W_2$		
S0			$W_2$			
ABUS			$W_2$	$W_2$		
DRW						
LDZ						
LDC						
LONG						
LAR						
MBUS						
MEMW						
PCADD	$C * W_2$	$Z * W_2$				
LPC			$W_2$			
STOP					$W_2$	

非流水线方案流程图



### 修改pc指针功能

修改pc指针的功能可以使得程序从任意地址开始执行。我们的具体操作是通过st0这个信号实现的。当st0=0的时候，我们使SBUS和LPC信号有效，按下QD脉冲，将数据开关上的数据锁存到pc寄存器里。同时，在这个节拍电位中，st0被设置成st0=1，并且不再改变。此后，控制台就会逐步执行存储器中的程序。

### 3.3 流水线硬布线控制器详解

我们小组基于已经完成的非流水线硬布线控制器，制定了流水线版的硬布线控制器的设计方案。

### 流水线方案实现

能够实现流水线作业的基本条件是存在能够同时运行的互不干扰的组件。在tec-8系统中，指令的取指部分和执行部分是没有资源冲突的(JZ, JC, JMP指令除外)，因而可以实现流水线作业。

我们小组的设计方案是：不再让取指部分单独占用一个cpu周期，而是将取指写在每条指令的最后一个cpu周期中。每条指令执行最后一个cpu周期同时取出下一条指令寄存到IR。

对于大部分指令,在时序信号发生器产生节拍电位，W<sub>1</sub>时执行当前指令,并且使得LIR信号有效,以便将下一条指令读取到寄存器IR中。同时PCINC信号有效，pc加1。这些指令现在只需要一个节拍电位来执行，所以我们使得SHORT信号有效。取消时序发生器的第二个节拍电位W<sub>2</sub>。

对于LD和ST指令，我们在第二个周期添加了读取下一条指令的操作。

对于**JMP**指令，和满足跳转条件的**JZ, JC**指令，流水线设计与会导致**pc**寄存器发生数据冲突，因为在同一个节拍电位中执行了修改**pc**和取指令的操作。这三种情况下，指令不参与流水线作业。当**JZ**和**JC**指令不满足跳转条件时，可以参与流水线作业。指令在一个节拍电位内完成。

如果画出流水线的时空图，则有以下四种情况：

CPU周期1	CPU周期2	CPU周期3	CPU周期4	CPU周期5
1 取指	执行			
	取指	执行		
2 取指	执行			
	取指	执行1	执行2	
3 取指	执行1	执行2		
		取指	执行	
4 取指	执行1	执行2		
		取指	执行1	执行2

其他情况的流水线都可能出现错误。例如：

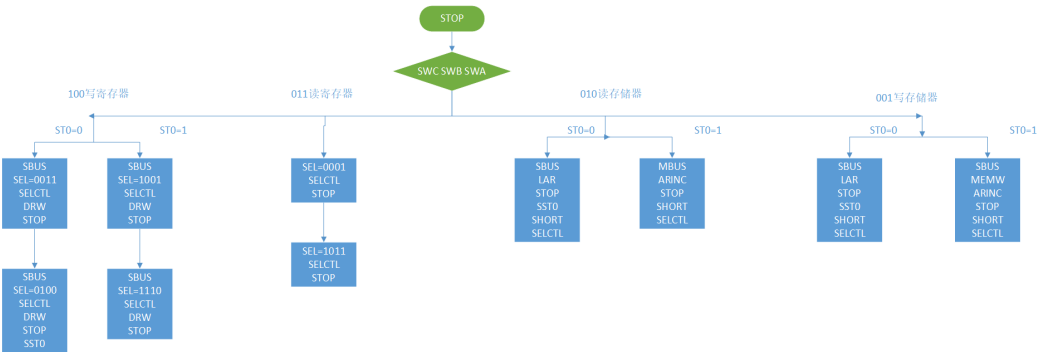
CPU周期1	CPU周期2	CPU周期3	CPU周期4	CPU周期5
1 取指	执行1	执行2		
	取指	执行		

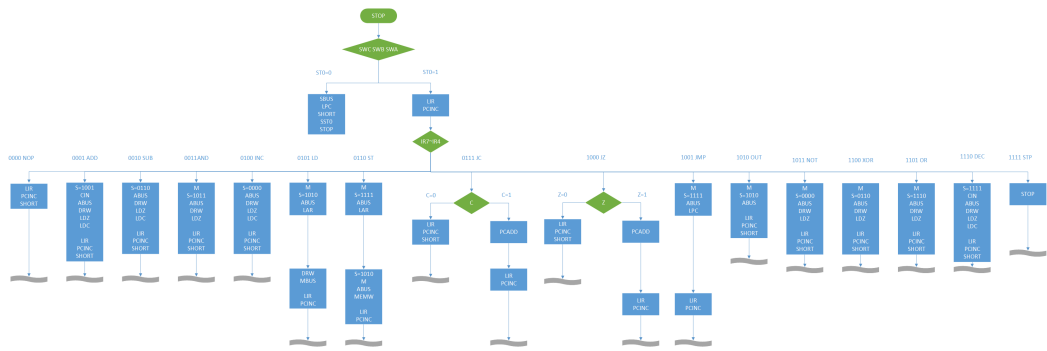
在第一条指令的”执行2“部分还没有完成时，新的指令已经被寄存在**IR**, 导致第一条指令没有被执行完就开始了第二条指令的执行。

因为指令系统中执行时间最短为两个**cpu**周期，所以我们的流水线只能设计为二级流水，令一条指令的取指部分和上一条指令的最后一个执行部分同时进行。想要实现三级流水，则必须要求指令流中出现指令最少需要三个**cpu**周期，并且各部分空间上不能有资源冲突。

在本指令系统中，即使指令流中只有需要三个**cpu**周期才能完成的指令**LD**和**ST**，也不能实现三级流水，因为两条指令的“执行2”部分都需要使用**IR**中寄存的指令内容，造成资源冲突。

流水线方案流程图





## 4 测试与验证

### 4.1 测试程序

#### 寄存器初值

寄存器	初值
R2	60H
R3	FDH

#### 存储器初值

由于执行过程中出现多次跳转，于是记下指令在存储器中的机器码

存储器	初值
00H	0101 0010
01H	0100 1000
02H	0101 0110
03H	0001 0001
04H	0111 0001
05H	0011 0100
06H	0010 0010
07H	0100 0100
08H	0110 0100
09H	0100 1100
0AH	1000 0010
0BH	0101 1011
0CH	1001 1000
0DH	0100 1100
0EH	0100 1100
0FH	0010 0010

存储器	初值
10H	0101 1000
11H	0001 1110
12H	0101 1111
13H	1010 0000
14H	1111 0000
60H	67H
61H	80H
62H	FDH
80H	60H
FEH	03H
FFH	03H

## 执行过程

地址	程序指令	机器码	16进制	R0	R1	R2	R3
00H	LD R0,[R2]	0101 0010	52H	67H		60H	FDH
01H	INC R2	0100 1000	48H	67H		61H	FDH
02H	LD R1,[R2]	0101 0110	56H	67H	80H	61H	FDH
03H	ADD R0,R1	0001 0001	11H	E7H	80H	61H	FDH
04H	JC 06H	0111 0001	71H	C=0 不跳转			
05H	AND R1,R0	0011 0100	34H	E7H	80H	61H	FDH
06H	SUB R0,R2	0010 0010	22H	86H	80H	61H	FDH
07H	INC R1	0100 0100	44H	86H	81H	61H	FDH
08H	ST R0,[R1]	0110 0100	64H	[81H]=86H			
09H	INC R3	0100 1100	4CH	86H	81H	61H	FEH
0AH	JZ 0DH	1000 0010	82H	Z=0 不跳转			
0BH	LD R2,[R3]	0101 1011	5BH	86H	81H	03H	FEH
0CH	JMP R2	1001 1000	98H	PC = 03H			
03H	ADD R0,R1	0001 0001	11H	07H	81H	03H	FEH
04H	JC 06H	0111 0001	71H	C=1 跳转			
06H	SUB R0,R2	0010 0010	22H	04H	81H	03H	FEH
07H	INC R1	0100 0100	44H	04H	82H	03H	FEH
08H	ST R0,[R1]	0110 0100	64H	[82H]=04H			
09H	INC R3	0100 1100	4CH	04H	82H	03H	FFH
0AH	JZ 0DH	1000 0010	82H	Z=0 不跳转			
0BH	LD R2,[R3]	0101 1011	5BH	04H	82H	03H	FFH
0CH	JMP R2	1001 1000	98H	PC = 03H			
03H	ADD R0,R1	0001 0001	11H	86H	82H	03H	FFH
04H	JC 06H	0111 0001	71H	C=0 不跳转			

05H	AND R1,R0	0011 0100	34H	86H	82H	03H	FFH
06H	SUB R0,R2	0010 0010	22H	83H	82H	03H	FFH
07H	INC R1	0100 0100	44H	83H	83H	03H	FFH
08H	ST R0,[R1]	0110 0100	64H	[83H]=83H			
09H	INC R3	0100 1100	4CH	83H	83H	03H	00H
0AH	JZ 0DH	1000 0010	82H	Z=1 跳转			
0DH	INC R3	0100 1100	4CH	83H	83H	03H	01H
0EH	INC R3	0100 1100	4CH	83H	83H	03H	02H
0FH	SUB R0,R2	0010 0010	22H	80H	83H	03H	02H
10H	LD R2,[R0]	0101 1000	58H	80H	83H	60H	02H
11H	ADD R3,R2	0001 1110	1EH	80H	83H	60H	62H
12H	LD R3,[R3]	0101 1111	5FH	80H	83H	60H	FDH
13H	OUT R0	1010 0000	A0H	DBUS=80H			
14H	STP	1111 0000	F0H				

寄存器检测	R0=80H	R1=83H	R2=60H	R3=FDH
存储器检测	[81H]=86H	[82H]=04H	[83H]=83H	

## 4.2 斐波那契数列

### 寄存器初值

寄存器	初值
R0	01H
R1	01H
R3	03H

### 执行过程

地址	程序指令	机器码	16进制	R0	R1	R2	R3
00H	XOR R0,R1	1100 0001	C1H	00H	01H		03H
01H	OR R0,R1	1101 0001	D1H	01H	01H		03H
02H	DEC R0	1110 0000	E0H	00H	01H		03H
03H	ADD R0,R1	0001 0001	11H	01H	01H		03H
04H	STP	1111 0000	F0H				
05H	MOV R2,R1	1011 1001	B9H	01H	01H	01H	03H
06H	MOV R1,R0	1011 0100	B4H	01H	01H	01H	03H

07H	MOV R0,R2	1011 0010	B2H	01H	01H	01H	11H
08H	JMP [R3]	1001 1100	9CH	PC=03H 以下继续计算斐波那契数列			

R0=00H,01H,02H,03H,05H,08H,0DH,15H...

## 5 问题和解决

### 5.1 烧写程序BUG

问题描述：测试流水线程序的时候，烧写程序之后控制台没有反应

一开始，我们试图烧写流水线程序，但是控制台的硬布线控制器一直是原来的非流水线版本程序。经过多次试验，我们发现将数据线重新插拔之后，能够成功烧写程序。

### 5.2 执行MOV指令错误

我们的mov指令编码出现了问题，将IR1IR0选择的寄存器当成了目标寄存器，但是被写入的寄存器只能是IR3IR2选择的寄存器，并且还需要DRW信号高有效。我们修改了指令设计和代码之后，程序运行正常。

### 5.3 流水版本指令覆盖问题

这是一个非常普遍而且致命的问题，主要是因为实验台性能和流水设计不兼容，导致部分指令被覆盖，运行程序的时候当前指令不执行，下一条指令执行两次。

问题分析：在TEC-8实验台中，芯片处理速度存在差异，部分芯片采用GAL运算速度较快，而部分则较慢。在流水CPU的T3上升沿执行PCINC和LIR指令时，VHDL程序设计中两条指令并行执行，理论上在PCINC完成前LIR应该已经取到指令；但因硬件性能问题LIR指令执行慢于PCINC，导致LIR载入的是下一条指令，从而产生了指令覆盖的现象。通过以上分析，我们给出了5种解决方案。

解决方案：

1. 更换性能较高的实验台。经过测试，在某些实验台上，流水CPU程序能够正常运行。
2. 修改测试程序：在INC、DEC指令前后增加NOP指令（较简单）。
3. 修改流水设计：在INC、DEC等被覆盖掉的指令前后增加NOP。
4. 取消部分流水：将INC、DEC改为非流水，在最后一拍进行取指（效果最稳定）。
5. 采用延时机制：使得PCINC操作时延，等LIR加载了上一条指令后才改PC（由于T3上升沿为寄存器属性，也无法修改为T3下降沿再PCINC，经测试该思路因硬件而受限）。

## 6 中断设计与测试方案

因为EPM7128中没有中断相应的引脚，我们无法在现有实验台上实际实现中断，因此我们提供仅中断设计方案和测试方案，方案假设芯片有要可以接收时序产生器发出的PULSE信号，并且能够发出LIAR、INTEN、INTDI信号，该方案参考了基于TEC-8的中断实验。

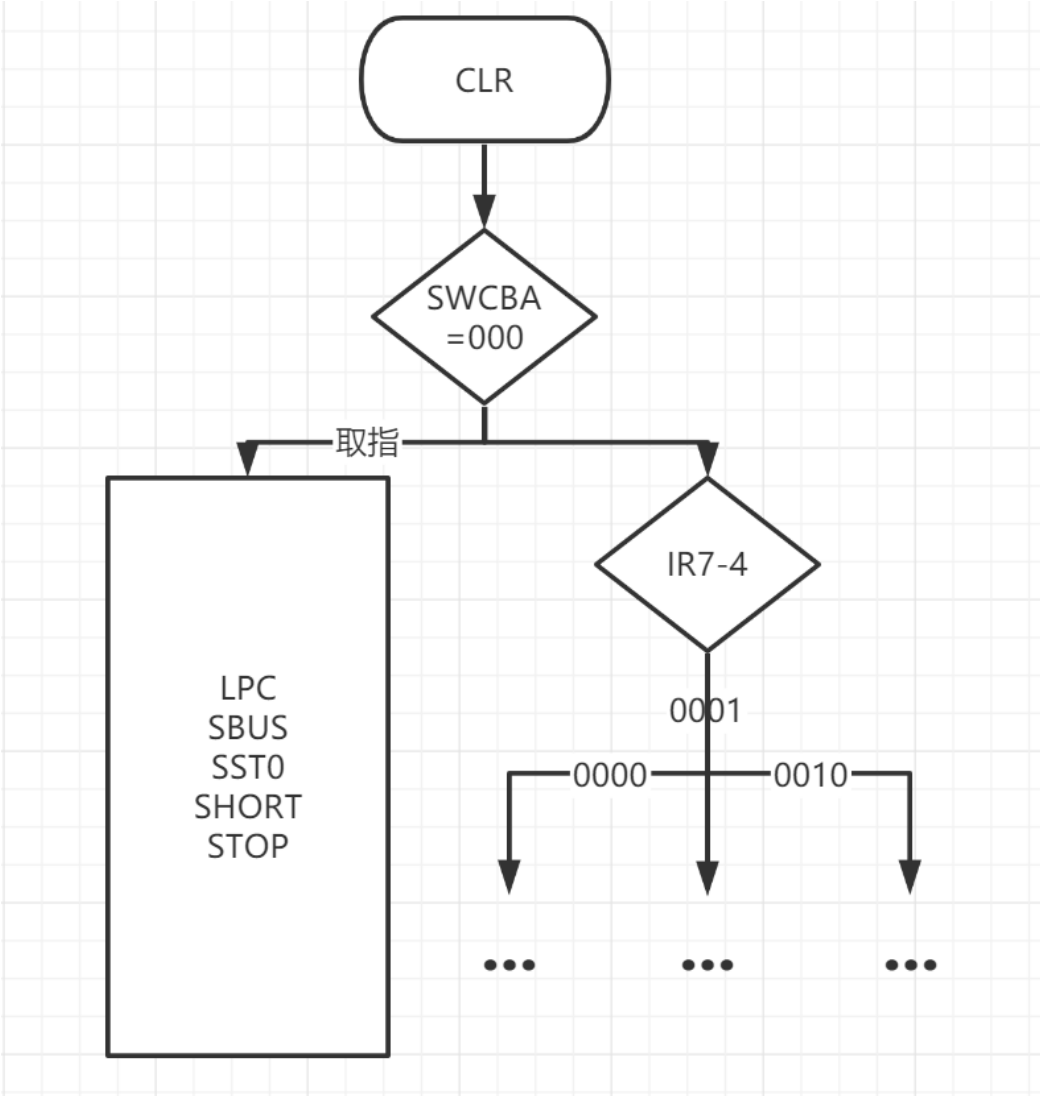
# 6.1 中断原理与需求分析

一般来讲，CPU中断分为两类，内中断（同步 **Software Interrupts**）与外中断（异步 **Hardware Interrupts**）。内中断是CPU内部产生的，不涉及外设的异常，外中断由外设通过芯片针脚发送的信号触发。在我们TEC-8系统中为了简化附加的设计，并且考虑实际需求，我们只考虑外中断的情况，即时序发生器在任意时刻发出**PULSE**信号引发中断。

大体上可以把中断全过程分为5个阶段：中断请求、中断判优、中断响应、中断处理和中断返回。在TEC-8中，我们中断方案实现的主要工作是请求、响应、处理和返回。主要工作首先是保护断点、保存现场、中断屏蔽、进入中断程序，处理之后恢复断点、恢复现场和开中断、中断返回，从而实现一个完整的基本中断过程。

在TEC-8中，具体需要做的是，增加一个**IAR**寄存器用于保存断点**PC**，以及相应的寄存器、**FLAG**寄存器等的保存空间（一般情况下，CPU寄存器在中断保存现场时是**PUSH**到栈中，但因条件限制我们可以采用增加寄存器的方式实现）；中断向量可以用**SBUS**手动输入中断处理程序在存储器中的地址。

指定中断向量流程：



对于多级中断和嵌套中断，由于TEC-8保存现场的能力有限，需要更多的寄存器和**IAR**等的支持才能很好的实现，需要保存多个现场时，其中一种解决方案是增加一定量的寄存器便于存储，利用存储器模拟一个堆栈，采用类似Intel CPU的**PUSH**和**POP**机制来保存断点和现场，这不失为一种可行的方案。



## 6.2 指令与信号设计

### 指令设计

指令	别名	作用
DI指令	关中断指令	此条指令执行后，EN_INT置0，即使发生中断请求，TEC-8也不响应中断请求。
EI指令	开中断指令	此条指令执行后，EN_INT置1，TEC-8可以响应中断。
IRET指令	恢复断点指令	该指令产生 IABUS 信号，恢复断点地址，产生信号 LPC，将断点从数据总线装入 PC，恢复被中断的程序。

### 信号设计

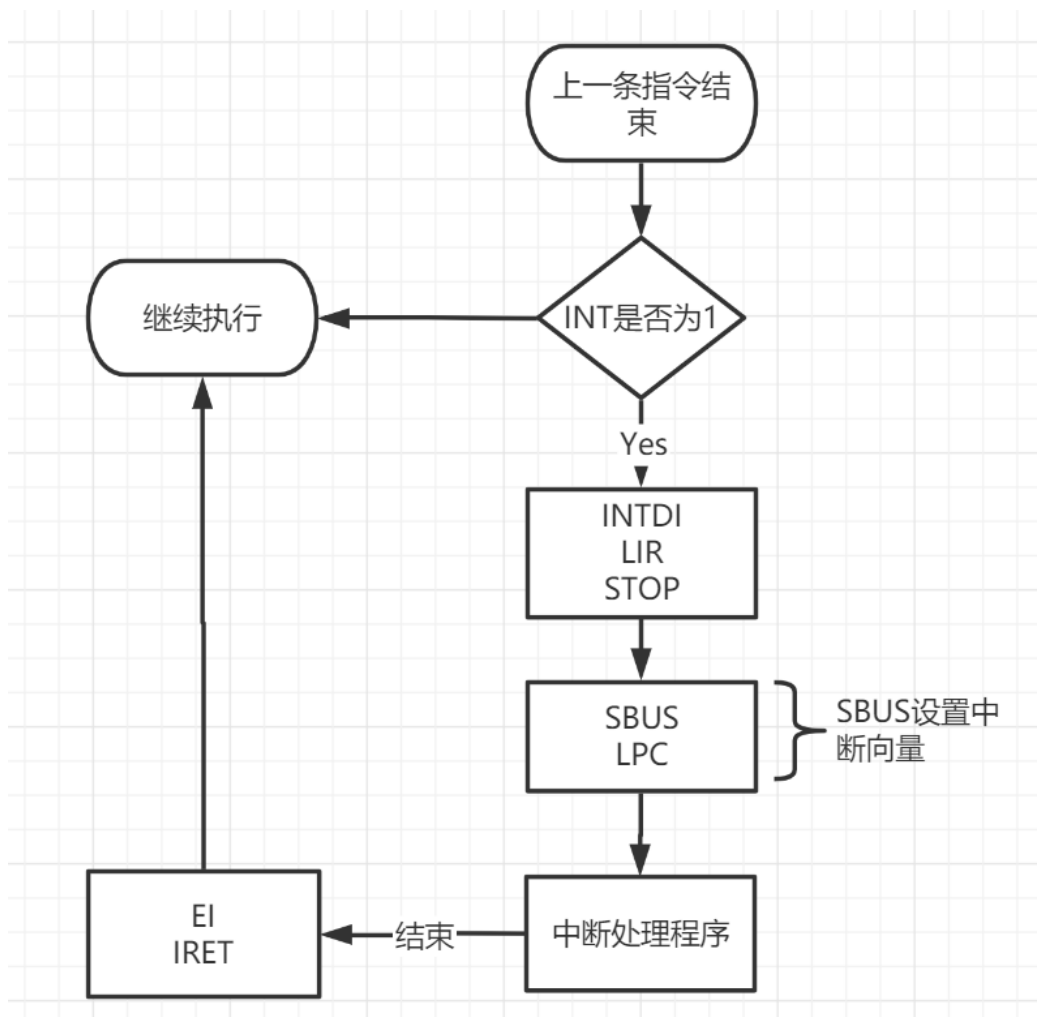
信号	作用
INT	1:当前正在响应中断; 0:当前没有在响应中断。应该在开始中断的时候设置为1，中断结束之后设置为0。
EN_INT	0:禁止中断;1:允许中断。由信号EI和DI控制，进行中断时需要设置为0，结束之后重新修改成1。
IABUS	信号为1时，从IAR到PC的数据通路打开，IAR中的数据可以在t3下降沿时从IAR写入PC。中断结束恢复现场的时候用到。
PULSE	控制台上的脉冲信号，外部打入，用来制造中断请求。
LIAR	IAR寄存器的允许打入信号，为1时允许pc寄存器中的值在t3下降沿打入IAR。

如果当前处于能够接收中断请求的状态，那么中断将会在接收到PUSLE信号的指令结束后开始，在遇到IRET指令之前结束。

## 6.3 硬件设计

- 需要1个IAR寄存器用于保存断点PC，在恢复断点时通过LPC信号将断点存入PC中。
- 需要6个寄存器用于保存寄存器和C、Z标志寄存器，或者预留将来不再使用的存储器空间实现类似栈的功能，用于保存现场时一次PUSH保存寄存器，恢复现场时反序依次POP恢复寄存器。保存寄存器和恢复寄存器的操作均在中断服务程序中。

## 6.4 中断流程图



## 6.5 VHDL程序设计-中断

根据设计的信号可以写出中断相关逻辑表达式：

-- 三条指令的逻辑表达式

```

di <= (st0='1' and run='1' and ir=code for di) ;
ei <= (st0='1' and run='1' and ir=code for ei) ;
iret <= (st0='1' and run='1' and ir=code for iret) ;

```

由于tec-8系统已经不支持我们再增加3条指令，所以我们这里保留三条指令的代码，留待以后能够支持更多指令的时候再行设计。

```

process(clr, mf, ei, di, pulse, en_int)
begin
    if clr='0' then
        en_int<='0' ;
    elsif rising_edge(mf) then
        en_int <= ei or (en_int and not di) ;
    end if ;
    int <= en_int and pulse ;
end process ;

```

```

process(int)
begin

```

```
    if rising_edge(int) then
        st0='0' ;
    end process ;

-- 利用信号int 再创建一个分支，如果st0=0 且 int=1，就进入断点保存，写入pc
-- 如果st0=0 且 int=0 就直接进入写入pc
-- 如果st0=1 直接执行程序
liar <= (not st0 and int and w(1)) ;
lpc <= ... (not st0 and not int and w(1)) or (not st0 and int and
w(2)) or (iret and w(2));
iabus <= (iret and w(2)) ;
```

## 7 附录

---

### 7.1 工程进度日志

#### 8月10日 初出茅庐

暑假休息了一个多月，关于计算机组成原理的一些内容已经忘了不少。三个人一起探讨了，用QQ视频一起复习课上讲的关于CPU的内容。还用虚拟仿真实验的TEC-8一起做了数据通路实验。

#### 8月15日 深入学习

虽然复习了一系列的内容，但是对于写硬布线控制器，我们还是毫无头绪。翻找资料找到了《计算机硬件基础实验教程》和《计算机组成与设计实验教程》，学习硬布线控制器的相关知识。

#### 8月16日 撸起袖子开始干

和前一天一样，继续学习硬布线控制器的知识。通过研究非流水硬布线的流程图和流水硬布线控制器的相关知识，搞清楚了流水和非流水的硬布线控制器在译码表方面的区别。仿照非流水硬布线的流程图，大家一起搞出了流水硬布线的流程图，还写完了基本指令集，拓展指令集和两张译码表。

#### 8月17日 计日程功，进度可喜

写完了译码表就开始写代码，但是大一学习的VHDL语言已经忘得差不多了。花费了一些的时间学习了VHDL。基本完成非流水的代码的编程，并着手研究流水的实现。

## 9月2日 牛刀小试，BUG不断

回到学校开始用上了TEC-8的试验台。好久没用试验台了，手都生疏了。在试验台上测试数据通路的实验。

一开始做实验的时候，没发现拨到了编程开关，一直做无用功。花了不少功夫找到了一台没有坏的试验台，完整地做完了数据通路的实验。很快，第一版的流水代码就改完了。代码比较繁琐，周鹏飞特别优化了一下代码，又出了一版。在电脑上的仿真实验，验证了正确性。

周鹏飞继续写流水版本的代码，金浩男根据老师给的测试程序，写完了机器码和流程中的寄存器变化，然后和苟志斌一起上机烧录非流水版本的代码，一人报管脚编号，一人输入。苟志斌操作机器，一遍通过了测试程序。接着周鹏飞写完了代码，上机测试，发现周期的变化不符合流水硬布线的变化。周鹏飞和苟志斌看了很久代码，并没有发现问题。最后发现是没有烧录进去，拔掉USB-PLUS再插一次，就烧录进去了。但是在ST指令出了问题。

## 9月3日 高效协作，顺利验收

昨天晚上解决了ST指令的问题，周鹏飞发现写在指令集中的Rd和Rs换了位置，但是代码中写反了。在大家的讨论下，决定换掉原来的一个拓展指令，换成MOV指令，就可以顺利实现斐波那契数列的计算。苟志斌和金浩男一起写了斐波那契数列的代码和机器码。周鹏飞用改完的程序上机测试，发现DEC指令和INC指令出现了后一条指令覆盖前一条指令的情况。

在电脑上测试了流水版本的代码，确定其没有错误。然后换了一台机器测试斐波那契数列的代码，发现没有错误。但在测试老师给的测试程序，在INC指令处发生了之前一样的情况。在询问了老师和其他同学后，我们确定了，机器上的芯片发热，PCINC快了，先执行了，导致写入了下一条指令。在尝试了各种方式后（具体见问题与解决），用了最稳妥的方法（把INC和DEC改成非流水版本），顺利通过了测试程序，然后顺利通过了验收。

## 9月4日 总结复盘，收获满满

今天大家聚在一起讨论了中断的设计方案，我们参考了计算机组成原理课程中的TEC-8中断实验，并且对中断原理进行了系统的复习。在一上午的共同努力下，我们给出了中断指令、信号和硬件设计，并给出了VHDL示例代码。

最后，我们完善了实验报告，大家研讨总结了这门课程设计，各自谈了谈心得体会，大家都觉得有不断学习和代码的兢兢业业，也有顺利跑起来流水的喜出望外，收获满满！

## 7.2 小组成员及心得体会

### 小组成员名单

姓名	学号	班级
金浩男	2018211121	2018211318
苟志斌	2018213292	2018211318
周鹏飞	2018211123	2018211318

# 成员心得体会

## 周鹏飞

这么课程设计对于我们的逻辑关系掌握，代码设计能力和实际动手操作的能力要求都很高。我们一开始就以实现流水线版本的tec-8硬布线控制器为目标，但是我们决定循序渐进，先完成非流水版本的代码编写和调试，在此基础上设计流水版本，并进行调试和优化。

实验过程并不如想象中顺利。我们遇到了很多问题，从设计构思的一开始我们就发现自己对于硬件设计已经生疏了许多。复习硬件设计的思路(尤其是硬件的并发性花了好长一段时间去适应)，复习vhdl语言的书写格式。这些问题都是我们一开始没有想到的，但好在他们都很容易克服。

最有挑战性的还是上机调试的时候。虽然我们提前编写好了代码，做过仿真，但是上机调试之后还是发现了很多意想不到的问题。我们对于tec-8模型机的操作也不是很熟悉，操作速度花了接近半天的时间才渐渐提升起来，能够流畅地将数据写入寄存器和存储器，从指示灯上读出当前的指令和数据。

测试流水线代码的时候，我们因为机器吞指令的问题头疼了一上午，提出了5种解决方案，最后终于有一种能够起作用，让我们成功完成了验收。在此过程中，我们不断根据自己指令执行过程中发生的错误返回去修改代码，甚至修改我们的指令设计架构，其中的工作量不可谓不大。事实证明，事前看起来再精密的构思设想，遇上现实中的大量不确定，不可控的因素，都会碰一鼻子灰。

实验让我受益匪浅，尤其是我们思考机器发生吞指令状况的原因的时候，需要对信号发生的时间顺序进行精密的分析。我的硬件编程能力也得到很大的提高。

## 苟志斌

从指令系统设计，到译码表制作，再到实际编写VHDL代码，以及更加耗费时间的实验台调试，我们坚持理论联系实际的思想，以实践证实理论，从实践中加深对理论知识的理解和掌握。实验是我们快速认识和掌握理论知识的一条重要途径。尤其是对于流水CPU版本的调试，先是遇到了下载器无效下载而程序却提示成功的隐蔽BUG，然后又因为实验台性能等的原因导致PCINC与LIR指令执行出错，以至于后一条指令覆盖前一条，弄得我们焦头烂额，调试了将近一上午，可谓是“屋漏偏逢连夜雨，船迟又遇打头风”。

在分析具体深层原因和排查各类错误之后，我们经过探讨，提出了5种有效的解决方案（见“问题和解决”），最终采取了较为稳定的修改部分指令（INC、DEC）为非流水执行的方案，这种“弃车保帅”的方式在实验台较为“恶劣”的环境下能够像非流水版本一样顺利运行，并求因为采用流水的方式，效率大大提高。

其次，在计算机组成原理课程设计这门课程中，我在收获知识的同时，还收获了阅历，收获了成熟，在此过程中，我们通过查找大量资料，请教老师，以及不懈的努力，不仅培养了独立思考、动手操作的能力，在各种其它能力上也都有了提高。更重要的是，在基于TEC-8的实践中，我们学会了很多学习的方法。而这是日后最实用的，可谓是受益匪浅。要面对社会的挑战，只有不断的学习、实践，再学习、再实践。

最后，在这门实践课程中，我们培养了团队协作的精神，大家积极沟通交流，一直保持同步行动，对于设计的各方面保持了认知的同步，到达了非常好的协作效果，效率比较高，在提前写好程序的情况下，用1天多的时间就将VHDL程序的两个版本（流水和非流水）在实验台成功运行。

总而言之，这门课程设计对于上承计算机硬件学以致用，下启操作系统的体系化学习，让我受益匪浅，满载而归！

## 金浩男

上学期我计算机组成原理学的不够好，在这次的流水硬布线控制器的实验中，为了能够完成实验，我自己系统地学习了计算机组成原理的课程内容。为了写代码，需要捡起数字逻辑时的VHDL代码。

在学习玩理论后，实际操作同样非常困难。这次的试验台操控虽然比数字逻辑的实验少了连线，但是同样非常困难，特别是计算机组成原理的课内实验都是在仿真实验平台完成的，实际机器的各种问题比仿真平台要多得多，需要考虑更多。手动在内存输入机器码也是非常麻烦。如果没有队友的相互配合，一个人不够细致，没法完成这个复杂的实验。

代码是正确的，但是没法在硬件上通过测试，这让我明白了，硬件上的小问题所引起的困难，比软件上的困难要大得多。用了很多方法，通过软件上的小取巧通过硬件上的问题，这个成就感比单纯地通过实验要大得多。

最后看到机器上的寄存器和存储器的值完全符合正确答案，AB端口的红绿灯跑出完美的斐波那契数列，我已经从原来对机器的恐惧，转变为了自信，征服了硬件，真是一件不容易的事。如果没有队友间的相互配合，我们也不可能在短短几天内完成这个实验。

## 7.3 代码附件

### 1 硬连线控制器（VHDL）

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity pipeline is
port (
    clr, t3, c, z : in std_logic ;
    swcba, w : in std_logic_vector (3 downto 1) ;
    ir : in std_logic_vector (3 downto 0);
    drw, pcinc, lpc, lar, pcadd, arinc, memw, stop, lir, ldz, ldc,
    short, long, cin, m, abus, sbus, mbus, selctl : out std_logic ;
    s, sel : out std_logic_vector(3 downto 0)
);
end pipeline ;

architecture rule of pipeline is
    signal run, wmem, rmem, rreg, wreg, st0, nop, add, sub, aand,
    inc, ld, st, jc, jz, jmp, oout, mov, xxor, oor, dec, stp :
    std_logic ;
begin

    -- first check swcba
    run<='1' when swcba="000" else '0' ;
```

```

wmem<='1' when swcba="001" else '0' ;
rmem<='1' when swcba="010" else '0' ;
rreg<='1' when swcba="011" else '0' ;
wreg<='1' when swcba="100" else '0' ;

-- if run and st0=1, then check ir
nop <= '1' when ( ir="0000" and run='1' and st0='1') else '0' ;
add <= '1' when ( ir="0001" and run='1' and st0='1') else '0' ;
sub <= '1' when ( ir="0010" and run='1' and st0='1') else '0' ;
aand <= '1' when ( ir="0011" and run='1' and st0='1') else '0' ;
;
inc <= '1' when ( ir="0100" and run='1' and st0='1') else '0' ;
ld <= '1' when ( ir="0101" and run='1' and st0='1') else '0' ;
st <= '1' when ( ir="0110" and run='1' and st0='1') else '0' ;
jc <= '1' when ( ir="0111" and run='1' and st0='1') else '0' ;
jz <= '1' when ( ir="1000" and run='1' and st0='1') else '0' ;
jmp <= '1' when ( ir="1001" and run='1' and st0='1') else '0' ;
oout <= '1' when ( ir="1010" and run='1' and st0='1') else '0' ;
;
mov <= '1' when ( ir="1011" and run='1' and st0='1') else '0' ;
xxor <= '1' when ( ir="1100" and run='1' and st0='1') else '0' ;
;
oor <= '1' when ( ir="1101" and run='1' and st0='1') else '0' ;
dec <= '1' when ( ir="1110" and run='1' and st0='1') else '0' ;
stp <= '1' when ( ir="1111" and run='1' and st0='1') else '0' ;

-- deal with st0
-- st0 needs to be changed when clr and some parts of wmem,
rmem, wreg
process(clr, w, t3)
begin
    -- if and only if you hit clear, st0 will be 0
    if clr='0' then
        st0 <= '0' ;
        -- st0 will be 1 at the falling edge of t3 of w1 or w2,
        depending on the choice
    elsif falling_edge(t3) then
        if st0='0' and ( (w(2)='1' and wreg='1') or (w(1)='1'
and (wmem='1' or rmem='1' or run='1') ) ) then
            st0 <= not st0 ;
        end if ;
    end if ;
end process ;

-- write the expressions of out signals based on the
translation table
drw <= (wreg and (w(1) or w(2))) or (w(2) and (add or sub or
aand or xxor or oor or inc or dec or mov)) or (w(3) and ld) ;
pcinc <= run and st0 and w(1) ;
lpc <= (run and (not st0) and w(1)) or (jmp and w(2)) ;
lar <= ((not st0) and w(1) and (wmem or rmem)) or (w(2) and (ld
or st)) ;

```

```

pcadd <= (w(2) and jc and c) or (w(2) and jz and z) ;
arinc <= (st0 and w(1) and (wmem or rmem)) ;
memw <= (st0 and w(1) and wmem) or (w(3) and st) ;
stop <= ((not st0) and w(1) and run) or (w(1) and (wmem and
rmem)) or ((w(1) or w(2)) and (rreg or wreg)) or (w(2) and stp) ;
lir <= run and st0 and w(1) ;
ldz <= w(2) and (add or sub or aand or xor or oor or inc or
dec) ;
ldc <= w(2) and (add or sub or inc or dec) ;
short <= ((not st0) and w(1) and run) or (w(1) and (wmem or
rmem)) ;
long <= (w(2) and (ld or st)) ;
cin <= (w(2) and (add or dec)) ;
m <= (w(2) and (aand or xor or oor or ld or jmp or oout or
mov)) or ((w(2) or w(3)) and st) ;
abus <= (w(2) and (add or sub or aand or xor or oor or inc or
dec or mov or ld or jmp or oout)) or ((w(2) or w(3)) and st) ;
sbus <= ((not st0) and w(1) and (run or rmem)) or (w(1) and
wmem) or ((w(1) or w(2)) and wreg) ;
mbus <= (st0 and w(1) and rmem) or (w(3) and ld) ;
selctl <= (w(1) and (wmem or rmem)) or ((w(1) or w(2)) and
(rreg or wreg)) ;
sel(3) <= (w(2) and rreg) or (st0 and (w(1) or w(2)) and wreg)
;
sel(2) <= w(2) and wreg ;
sel(1) <= (w(2) and rreg) or (((st0 and w(2)) or ((not st0) and
w(1))) and wreg) ;
sel(0) <= ((w(1) or w(2)) and rreg) or (w(1) and wreg) ;
s(3) <= (w(2) and (add or aand or oor or dec or ld or jmp or
oout or mov)) or ((w(2) or w(3)) and st) ;
s(2) <= (w(2) and (sub or xor or oor or dec or st or jmp)) ;
s(1) <= (w(2) and (sub or aand or xor or oor or dec or ld or
jmp or oout or mov)) or ((w(2) or w(3)) and st) ;
s(0) <= (w(2) and (add or aand or dec or st or jmp)) ;

end rule ;

```

## 2 流水硬连线控制器（VHDL）

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity pipeline is
port (
    clr, t3, c, z : in std_logic ;
    swcba, w : in std_logic_vector (3 downto 1) ;
    ir : in std_logic_vector (3 downto 0);
    drw, pcinc, lpc, lar, pcadd, arinc, memw, stop, lir, ldz, ldc,
    short, cin, m, abus, sbus, mbus, selctl : out std_logic ;

```



```

s, sel : out std_logic_vector(3 downto 0)
) ;
end pipeline ;

architecture rule of pipeline is
    signal run, wmem, rmem, rreg, wreg, st0, nop, add, sub, aand,
inc, ld, st, jc, jz, jmp, oout, mov, xxor, oor, dec, stp :
std_logic ;
begin

    -- first check swcba
    run<='1' when swcba="000" else '0' ;
    wmem<='1' when swcba="001" else '0' ;
    rmem<='1' when swcba="010" else '0' ;
    rreg<='1' when swcba="011" else '0' ;
    wreg<='1' when swcba="100" else '0' ;

    -- if run and st0=1, then check ir
    nop <= '1' when ( ir="0000" and run='1' and st0='1') else '0' ;
    add <= '1' when ( ir="0001" and run='1' and st0='1') else '0' ;
    sub <= '1' when ( ir="0010" and run='1' and st0='1') else '0' ;
    aand <= '1' when ( ir="0011" and run='1' and st0='1') else '0'
;

    inc <= '1' when ( ir="0100" and run='1' and st0='1') else '0' ;
    ld <= '1' when ( ir="0101" and run='1' and st0='1') else '0' ;
    st <= '1' when ( ir="0110" and run='1' and st0='1') else '0' ;
    jc <= '1' when ( ir="0111" and run='1' and st0='1') else '0' ;
    jz <= '1' when ( ir="1000" and run='1' and st0='1') else '0' ;
    jmp <= '1' when ( ir="1001" and run='1' and st0='1') else '0' ;
    oout <= '1' when ( ir="1010" and run='1' and st0='1') else '0'
;

    mov <= '1' when ( ir="1011" and run='1' and st0='1') else '0' ;
    xxor <= '1' when ( ir="1100" and run='1' and st0='1') else '0'
;

    oor <= '1' when ( ir="1101" and run='1' and st0='1') else '0' ;
    dec <= '1' when ( ir="1110" and run='1' and st0='1') else '0' ;
    stp <= '1' when ( ir="1111" and run='1' and st0='1') else '0' ;

    -- deal with st0
    -- st0 needs to be changed when clr and some parts of wmem,
rmem, wreg
    process(clr, w, t3)
    begin
        -- if and only if you hit clear, st0 will be 0
        if clr='0' then
            st0 <= '0' ;

            -- st0 will be 1 at the falling edge of t3 of w1 or w2,
depending on the choice
            elsif falling_edge(t3) then
                if st0='0' and ( (w(2)='1' and wreg='1') or (w(1)='1'
and (wmem='1' or rmem='1' or run='1') ) ) ) then
                    st0 <= not st0 ;

```

```

        end if ;
    end if ;

end process ;

-- jc, jz, jmp, ld, st 需要第二个阶段执行取指令,非流水
-- write the expressions of out signals based on the
translation table
    drw <= (wreg and (w(1) or w(2))) or (w(1) and (add or sub or
aand or xor or oor or inc or dec or mov)) or (w(2) and ld) ;
    pcinc <= (w(1) and (nop or add or sub or aand or inc or oout or
mov or xor or oor or dec or (jc and not c) or (jz and not z))) or
(w(2) and ((jc and c) or (jz and z) or jmp or ld or st)) ;
    lpc <= (run and (not st0) and w(1)) or (jmp and w(1)) ;
    lar <= ((not st0) and w(1) and (wmem or rmem)) or (w(1) and (ld
or st)) ;
    pcadd <= (w(1) and ((jc and c) or (jz and z))) ;
    arinc <= (st0 and w(1) and (wmem or rmem)) ;
    memw <= (st0 and w(1) and wmem) or (w(2) and st) ;
    stop <= ((not st0) and w(1) and run) or (w(1) and (wmem and
rmem)) or ((w(1) or w(2)) and (rreg or wreg)) or (w(1) and stp) ;
    lir <= (w(1) and (nop or add or sub or aand or inc or oout or
mov or xor or oor or dec or (jc and not c) or (jz and not z))) or
(w(2) and ((jc and c) or (jz and z) or jmp or ld or st)) ;
    ldz <= w(1) and (add or sub or aand or xor or oor or inc or
dec) ;
    ldc <= w(1) and (add or sub or inc or dec) ;
    short <= w(1) and (wmem or rmem or (not st0 and run) or nop or
add or sub or aand or inc or oout or mov or xor or oor or dec) ;
    cin <= (w(1) and (add or dec or mov)) ;
    m <= (w(1) and (aand or xor or oor or ld or jmp or oout)) or
((w(1) or w(2)) and st) ;
    abus <= (w(1) and (add or sub or aand or xor or oor or inc or
dec or mov or ld or jmp or oout)) or ((w(1) or w(2)) and st) ;
    sbus <= ((not st0) and w(1) and (run or rmem)) or (w(1) and
wmem) or ((w(1) or w(2)) and wreg) ;
    mbus <= (st0 and w(1) and rmem) or (w(2) and ld) ;
    selctl <= (w(1) and (wmem or rmem)) or ((w(1) or w(2)) and
(rreg or wreg)) ;
    sel(3) <= (w(2) and rreg) or (st0 and (w(1) or w(2)) and wreg)
;
    sel(2) <= w(2) and wreg ;
    sel(1) <= (w(2) and rreg) or (((st0 and w(2)) or ((not st0) and
w(1))) and wreg) ;
    sel(0) <= ((w(1) or w(2)) and rreg) or (w(1) and wreg) ;
    s(3) <= (w(1) and (add or aand or oor or dec or ld or jmp or
oout)) or ((w(1) or w(2)) and st) ;
    s(2) <= (w(1) and (sub or xor or oor or dec or st or jmp)) ;
    s(1) <= (w(1) and (sub or aand or xor or oor or dec or ld or
jmp or oout)) or ((w(1) or w(2)) and st) ;
    s(0) <= (w(1) and (add or aand or dec or st or jmp)) ;

```

```
end rule ;
```