《计算机组成原理与接口技术实验》

实验报告

（实验三）

学 院 名 称：数据科学与计算机学院

学 生 姓 名：朱泽磊

学　　号：16340317

专业（班级）：16 软件工程四（8）班

时　　间：2018 年 6 月 22 日

成 绩 ：

实验二 ： 多周期CPU设计与实现

一. 实验目的

(1) 认识和掌握多周期数据通路原理及其设计方法；
(2) 掌握多周期 CPU 的实现方法，代码实现方法；
(3) 编写一个编译器，将 MIPS 汇编程序编译为二进制机器码；
(4) 掌握多周期 CPU 的测试方法;
(5) 掌握多周期 CPU 的实现方法。

二. 实验内容

设计一个多周期 CPU，该 CPU 至少能实现以下指令功能操作。需设计的指令与格式如下：（**说明：操作码按照以下规定使用，都给每类指令预留扩展空间，后续实验相同。**）

==>算术运算指令

（1）add rd, rs, rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd<－rs + rt

（2）sub rd, rs, rt

000001	rs(5 位)	rt(5 位)	rd(5 位)	reserved

完成功能：rd<－rs － rt

（3）addi rt, rs, **immediate**

000010	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt<－rs + (sign-extend)**immediate**

==>逻辑运算指令

（4）or rd, rs, rt

010000	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd<－rs | rt

（5）and rd, rs, rt

010001	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd<－rs & rt

（6）ori rt, rs, **immediate**

010010	rs(5 位)	rt(5 位)	**immediate**

功能：rt<－rs | (zero-extend)**immediate**

==>移位指令

（7）sll rd, rt,sa

011000	未用	rt(5 位)	rd(5 位)	sa	reserved

功能：rd<－rt<<(zero-extend)sa，左移 sa 位 ，(zero-extend)sa

==>比较指令

（8） slt rd, rs, rt 带符号数

100110	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：if (rs<rt) rd =1 else rd=0，具体请看表 2 ALU 运算功能表，带符号

（9） sltiu rt, rs,immediate 不带符号

100111	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：if (rs <(zero-extend)**immediate**) rt =1 else rt=0， 具体请看表 2 ALU 运算功能表，不带符号

==>存储器读写指令

（10）sw rt, **immediate**(rs)

110000	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：memory[rs+ (sign-extend)**immediate**]<－rt。即将 rt 寄存器的内容保存到 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中。

（11）lw rt, **immediate**(rs)

110001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt <－ memory[rs + (sign-extend)**immediate**]。即读取 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中的数，然后保存到 rt 寄存器中。

==>分支指令

（12）beq rs,rt, **immediate** (说明：**immediate 从 pc+4 开始和转移到的指令之间间隔条数**)

110100	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：if(rs=rt) pc <－pc + 4 + (sign-extend)**immediate** <<2 else pc <－pc + 4

（13）bltz rs,**immediate**

110110	rs(5 位)	00000	**immediate**

功能：if(rs<0) pc←pc + 4 + (sign-extend)**immediate** <<2 else pc ←pc + 4

==>跳转指令

（14）j addr

111000	addr[27:2]

功能：pc <－{(pc+4)[31:28],addr[27:2],2'b00}，跳转。

说明：由于 MIPS32 的指令代码长度占 4 个字节，所以指令地址二进制数最低 2 位均为 0，将指令地址放进指令代码中时，可省掉！这样，除了最高 6 位操作码外，还有 26 位可用于存放地址，事实上，可存放 28 位地址，剩下最高 4 位由 pc+4 最高 4 位拼接上。

（15）jr rs

111001	rs(5 位)	未用	未用	reserved

功能：pc <－ rs，跳转。

==>调用子程序指令

（16）jal addr

111010	addr[27:2]

功能：调用子程序，pc <－ {(pc+4)[31:28],addr[27:2],2'b00}；$31<－pc+4，返回地址设置；子程序返回，需用指令 jr $31。跳转地址的形成同 j addr 指令。

==>停机指令

（17）halt (停机指令)

111111	00000000000000000000000000(26 位)

不改变 pc 的值，pc 保持不变。

在本文档中，提供的相关内容对于设计可能不足或甚至有错误，希望同学们在设计过程中如发现有问题，请你们自行改正，进一步补充、完善。谢谢！

三. 实验原理

多周期 CPU 指的是将整个 CPU 的执行过程分成几个阶段，每个阶段用一个时钟去完成，然后开始下一条指令的执行，而每种指令执行时所用的时钟数不尽相同，这就是所谓的多周期 CPU。CPU 在处理指令时，一般需要经过以下几个阶段：

(1) 取指令(IF)：根据程序计数器 pc 中的指令地址，从存储器中取出一条指令，同时，pc 根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入 pc，当然得到的“地址”需要做些变换才送入 pc。

(2) 指令译码(ID)：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。

(3) 指令执行(EXE)：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。

(4) 存储器访问(MEM)：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

(5) 结果写回(WB)：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

实验中就按照这五个阶段进行设计，这样一条指令的执行最长需要五个(小)时钟周期才能完成，但具体情况怎样？要根据该条指令的情况而定，有些指令不需要五个时钟周期的，这就是多周期的 CPU。

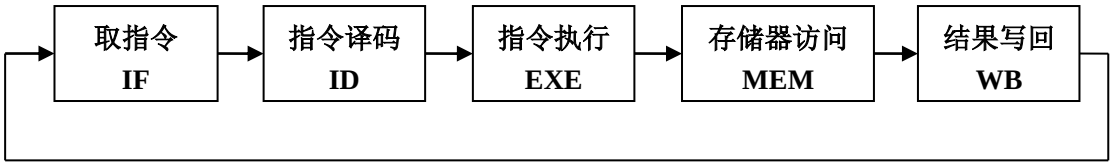
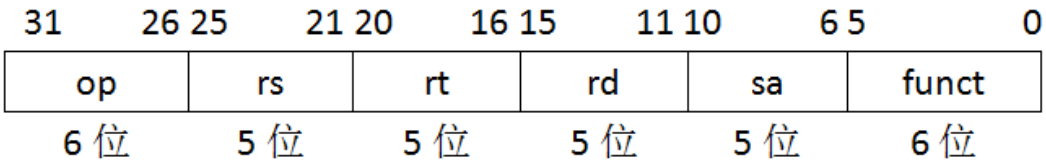


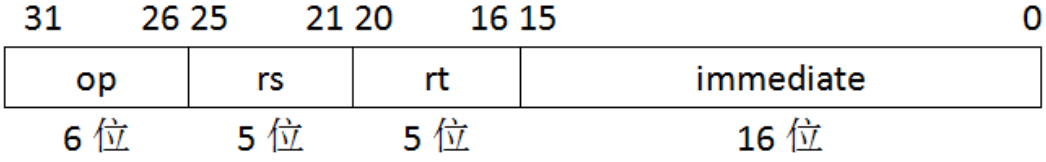
图 1　多周期 CPU 指令处理过程

MIPS 指令的三种格式：

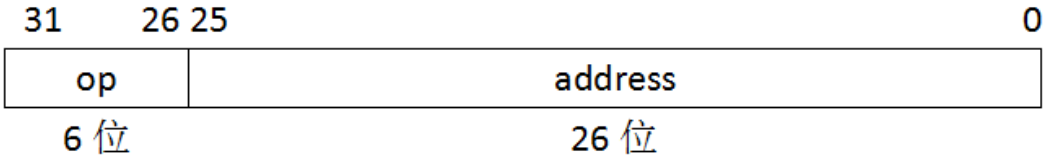
R 类型：



I 类型：



J 类型：



其中，

op：为操作码；

rs：为第 1 个源操作数寄存器，寄存器地址（编号）是 00000~11111，00~1F；

rt：为第 2 个源操作数寄存器，或目的操作数寄存器，寄存器地址（同上）；

rd：为目的操作数寄存器，寄存器地址（同上）；

sa：为位移量（shift amt），移位指令用于指定移多少位；

funct：为功能码，在寄存器类型指令中（R 类型）用来指定指令的功能；

immediate：为 16 位立即数，用作无符号的逻辑操作数、有符号的算术操作数、数据加载（Load）/数据保存（Store）指令的数据地址字节偏移量和分支指令中相对程序计数器（PC）的有符号偏移量；

address：为地址。

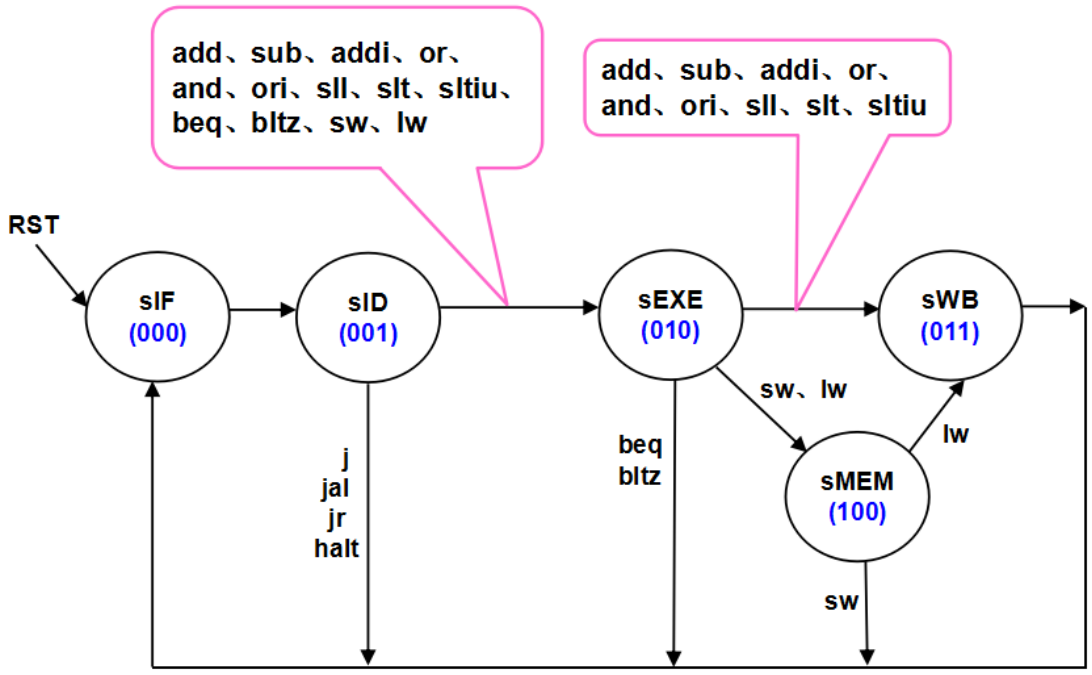


图 2 多周期 CPU 状态转移图

状态的转移有的是无条件的，例如从 sIF 状态转移到 sID 就是无条件的；有些是有条件的，例如 sEXE 状态之后不止一个状态，到底转向哪个状态由该指令功能，即指令操作码决定。每个状态代表一个时钟周期。

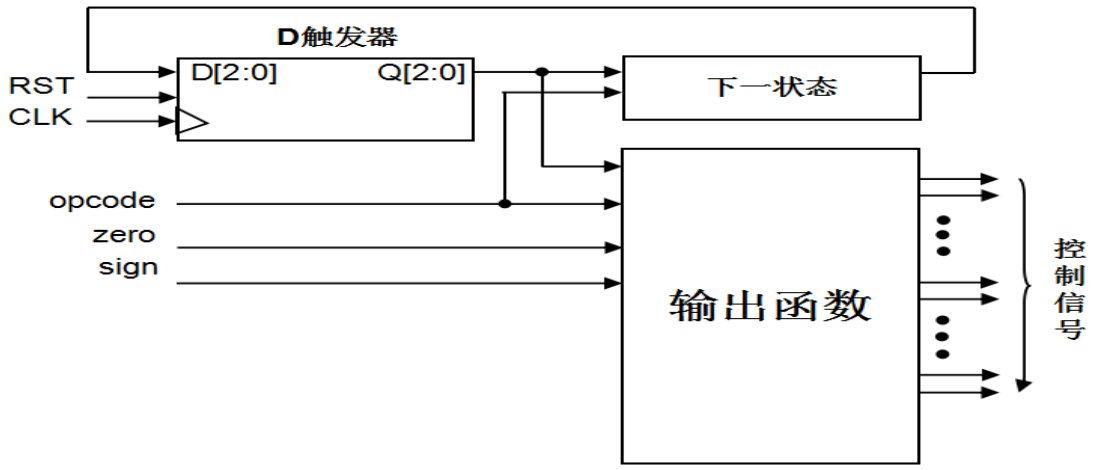


图 3 多周期 CPU 控制部件的原理结构图

图 3 是多周期 CPU 控制部件的电路结构，三个 D 触发器用于保存当前状态，是时序逻辑电路，RST 用于初始化状态“000“，另外两个部分都是组合逻辑电路，一个用于产生下一个阶段的状态，另一个用于产生每个阶段的控制信号。从图上可看出，下个状态取决于指令操作码和当前状态；而每个阶段的控制信号取决于指令操作码、当前状态和反映运算结果的状态 zero 标志和符号 sign 标志。

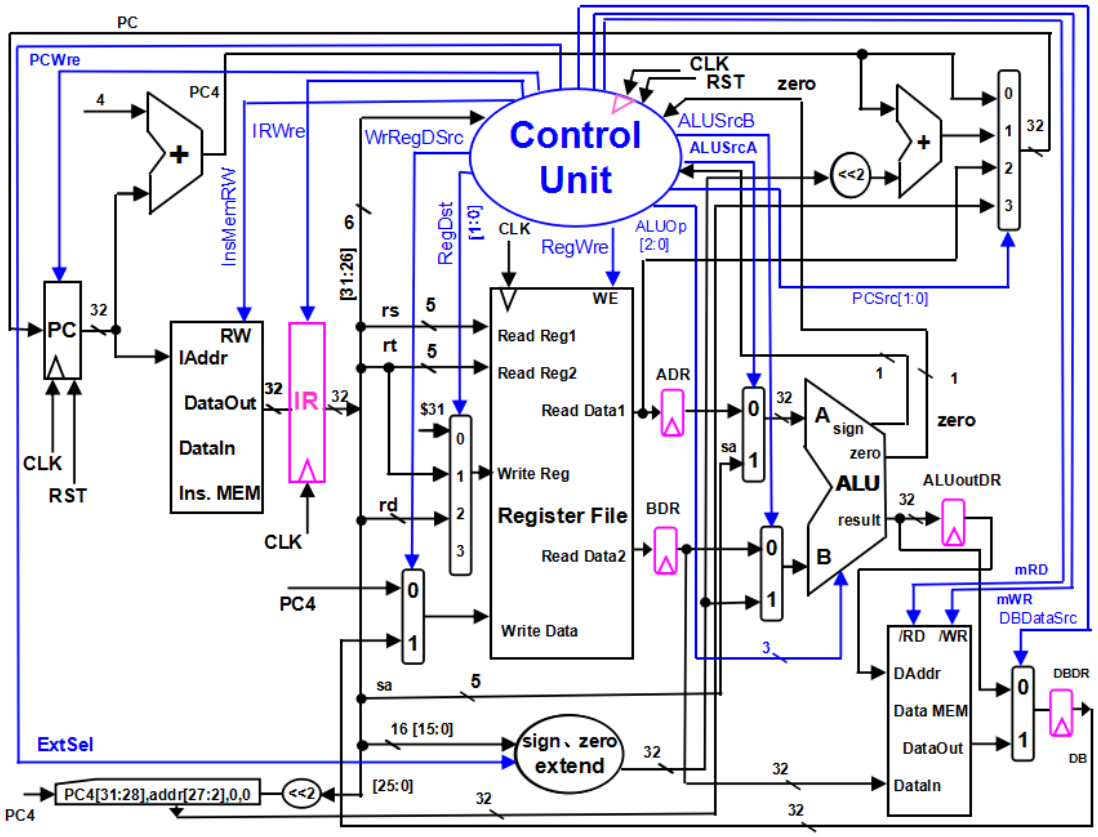


图 4　多周期 CPU 数据通路和控制线路图

图 4 是一个简单的基本上能够在多周期 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储器。访问存储器时，先给出内存地址，然后由读或写信号控制操作。对于寄存器组，给出寄存器地址（编号），读操作时不需要时钟信号，输出端就直接输出相应数据；而在写操作时，在 WE 使能信号为 1 时，在时钟边沿触发将数据写入寄存器。图中控制信号功能如表 1 所示，表 2 是 ALU 运算功能表。

特别提示，图上增加 IR 指令寄存器，目的是使指令代码保持稳定，pc 写使能控制信号 PCWre，是确保 pc 适时修改，原因都是和多周期工作的 CPU 有关。ADR、BDR、ALUoutDR、DBDR 四个寄存器**不需要写使能信号，**其作用是切分数据通路，将大组合逻辑切分为若干个小组合逻辑，大延迟变为多个分段小延迟。

表 1　控制信号作用

控制信号名	状态“0”	状态“1”
RST	对于 PC，初始化 PC 为程序首地址	对于 PC，PC 接收下一条指令地址
PCWre	PC 不更改，相关指令：halt，另外，除‘000’状态之外，其余状态慎改 PC 的值。	PC 更改，相关指令：除指令 halt 外，另外，在‘000’状态时，修改 PC 的值合适。
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、addi、or、and、ori、beq、bltz、slt、sltiu、sw、lw	来自移位数 sa，同时，进行(zero-extend)sa，即 {{27{1'b0}},sa}，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指	来自 sign 或 zero 扩展的立即数，相关

	令：add、sub、or、and、beq、bltz、slt、sll	指令：addi、ori、sltiu、lw、sw
DBDataSrc	来自 ALU 运算结果的输出,相关指令：add、sub、addi、or、and、ori、slt、sltiu、sll	来自数据存储器（Data MEM）的输出，相关指令：lw
RegWre	无写寄存器组寄存器，相关指令：beq、bltz、j、sw、jr、halt	寄存器组寄存器写使能，相关指令：add、sub、addi、or、and、ori、slt、sltiu、sll、lw、jal
WrRegDSrc	写入寄存器组寄存器的数据来自 pc+4(pc4），相关指令：jal，写$31	写入寄存器组寄存器的数据来自 ALU 运算结果或存储器读出的数据，相关指令：add、addi、sub、or、and、ori、slt、sltiu、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
mRD	存储器输出高阻态	读数据存储器，相关指令：lw
mWR	无操作	写数据存储器，相关指令：sw
IRWre	IR(指令寄存器)不更改	IR 寄存器写使能。向指令存储器发出读指令代码后，这个信号也接着发出，在时钟上升沿，IR 接收从指令存储器送来的指令代码。与每条指令都相关。
ExtSel	(zero-extend)**immediate**，相关指令：ori、sltiu；	(sign-extend)**immediate**，相关指令：addi、lw、sw、beq、bltz；
PCSrc[1:0]	00：pc<－pc+4，相关指令：add、addi、sub、or、ori、and、slt、sltiu、sll、sw、lw、beq(zero=0)、bltz(sign=0，或 zero=1); 01：pc<－pc+4+(sign-extend)**immediate**，相关指令：beq(zero=1)、bltz(sign=1，zero=0); 10：pc<－rs，相关指令：jr; 11：pc<－{(pc+4)[31:28],addr[27:2],2'b00}，相关指令：j、jal;	
RegDst[1:0]	写寄存器组寄存器的地址，来自： 00：0x1F($31)，相关指令：jal，用于保存返回地址（$31<-pc+4）； 01：rt 字段，相关指令：addi、ori、sltiu、lw; 10：rd 字段，相关指令：add、sub、or、and、slt、sll; 11：未用;	
ALUOp[2:0]	ALU 8 种运算功能选择(000-111)，看功能表	

相关部件及引脚说明：

Instruction Memory：指令存储器

Iaddr，指令地址输入端口

DataIn，存储器数据输入端口

DataOut，存储器数据输出端口

RW，指令存储器读写控制信号，为 0 写，为 1 读

Data Memory：数据存储器

Daddr，数据地址输入端口

DataIn，存储器数据输入端口

DataOut，存储器数据输出端口
/RD，数据存储器读控制信号，为 0 读
/WR，数据存储器写控制信号，为 0 写

Register File：寄存器组

Read Reg1，rs 寄存器地址输入端口
Read Reg2，rt 寄存器地址输入端口
Write Reg，将数据写入的寄存器，其地址输入端口（rt、rd）
Write Data，写入寄存器的数据输入端口
Read Data1，rs 寄存器数据输出端口
Read Data2，rt 寄存器数据输出端口
WE，写使能信号，为 1 时，在时钟边沿触发写入

IR：　指令寄存器，用于存放正在执行的指令代码

ALU：　算术**逻辑单元**

result，ALU 运算结果
zero，运算结果标志，结果为 0，则 zero=1；否则 zero=0
sign，运算结果标志，结果最高位为 0，则 sign=0，正数；否则，sign=1，负数

表 2 ALU 运算功能表

ALUOp[2..0]	功能	描述
000	Y = A + B	加
001	Y = A － B	减
010	Y= (A<B) ?1: 0	比较 A 与 B 不带符号
011	Y=(((rega<regb) && (rega[31] == regb[31])) ||((rega[31] ==1 && regb[31] == 0))) ? 1:0	比较 A 与 B 带符号
100	Y = B<<A	B 左移 A 位
101	Y = A ∨ B	或
110	Y = A ∧ B	与
111	Y = A ⊕ B	异或

值得注意的问题，设计时，用模块化、层次化的思想方法设计，关于如何划分模块、如何整合成一个系统等等，是必须认真考虑的问题。

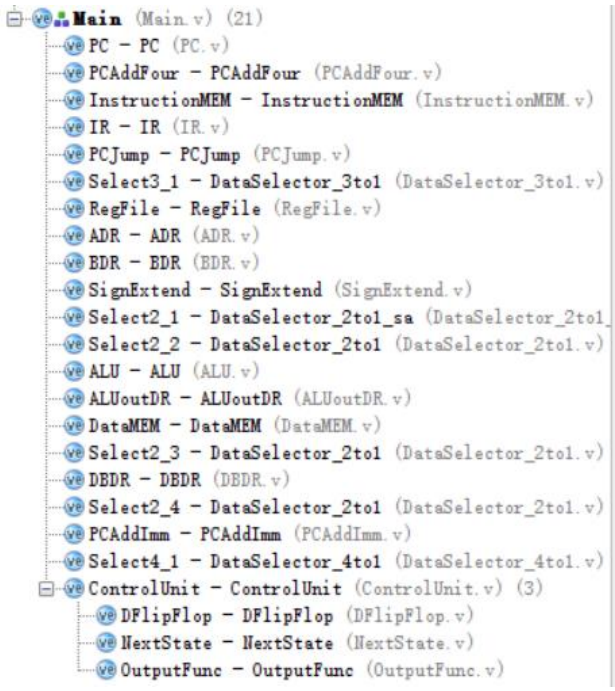
四. 实验器材

PC 机一台，BASYS 3 实验板一块，Xilinx Vivado 开发软件一套。

五. 实验分析与设计

1.设计部分：

根据多周期 CPU 通路图，将整个 CPU 分为以下部分分别实现：PC，指令寄存器，寄存器组，控制单元，ALU，符号扩展单元，存储单元，临时寄存器，N 选一选择器。具体结构如下图所示：



其中大多数单元与单周期类似，有较大不同的主要是控制单元，控制单元可以细分为 3 个部分，D 触发器、NextState 部分和 OutputFunction 部分，其中 D 触发器的作用主要是在每次时钟上升沿改变当前的运行状态，NextState 的功能是根据当前的情况判断下一个状态，OutputFunction 则是根据当前的状态给控制单元的各个控制信号匹配合适的值。

相比于单周期，多周期多了几个临时寄存器，他们会在时钟和控制信号满足条件时将数据传输到下一个原件内，作用主要是分割数据通路，确保数据不会在一个时钟周期内就传遍整个 CPU，而是根据当前状态慢慢前移。

2.设计测试指令：

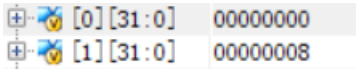
这里直接填补了老师给出的例子，完成了下图指令文件：

地址	汇编程序	指令代码			
		op（6）	rs(5)	rt(5)	rd(5)/immediate (16)
0x00000000	addi $1,$0,8	000010	00000	00001	0000 0000 0000 1000
0x00000004	ori $2,$0,2	010010	00000	00010	0000 0000 0000 0010
0x00000008	or $3,$2,$1	010000	00010	00001	0001 1000 0000 0000
0x0000000C	sub $4,$3,$1	000001	00011	00001	0010 0000 0000 0000
0x00000010	and $5,$4,$2	010001	00100	00010	0010 1000 0000 0000
0x00000014	sll $5,$5,2	011000	00000	00101	0010 1000 1000 0000
0x00000018	beq $5,$1,-2(=,转 14)	110100	00101	00001	1111 1111 1111 1110
0x0000001C	jal 0x0000040	111010	00000	00000	0000 0000 0001 0000
0x00000020	slt $8,$12,$1	100110	01100	00001	0100 0000 0000 0000
0x00000024	addi $13,$0,-2	000010	00000	01101	1111 1111 1111 1110
0x00000028	slt $9,$8,$13	100110	01000	01101	0100 1000 0000 0000
0x0000002C	sltiu $10,$9,2	100111	01001	01010	0000 0000 0000 0010
0x00000030	sltiu $11,$10,0	100111	01010	01011	0000 0000 0000 0000
0x00000034	addi $13,$13,1	000010	01101	01101	0000 0000 0000 0001
0x00000038	bltz $13,-2 (<0,转 34)	110110	01101	00000	1111 1111 1111 1110
0x0000003C	j 0x000004C	111000	00000	00000	0000 0000 0001 0011
0x00000040	sw $2,4($1)	110000	00001	00010	0000 0000 0000 0100
0x00000044	lw $12,4($1)	110001	00001	01100	0000 0000 0000 0100
0x00000048	jr $31	111001	11111	00000	0000 0000 0000 0000
0x0000004C	halt	111111	00000	00000	0000000000000000

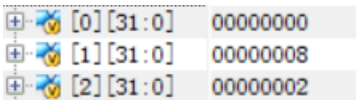
3.指令测试：

仿真文件的时钟周期设为100ns，仿真结果如下：

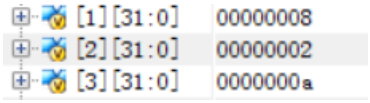
- addi $1,$0,8



- ori $2,$0,2



- or $3,$2,$1



- sub $4,$3,$1

[1][31:0]	00000008
[2][31:0]	00000002
[3][31:0]	0000000a
[4][31:0]	00000002

- and $5,$4,$2

[2][31:0]	00000002
[3][31:0]	0000000a
[4][31:0]	00000002
[5][31:0]	00000002

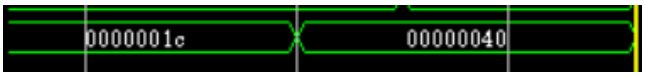
- sll $5,$5,2

[5][31:0]	00000008

- beq $5,$1,-2(=,转 14)

[1][31:0]	00000008
[2][31:0]	00000002
[3][31:0]	0000000a
[4][31:0]	00000002
[5][31:0]	00000008

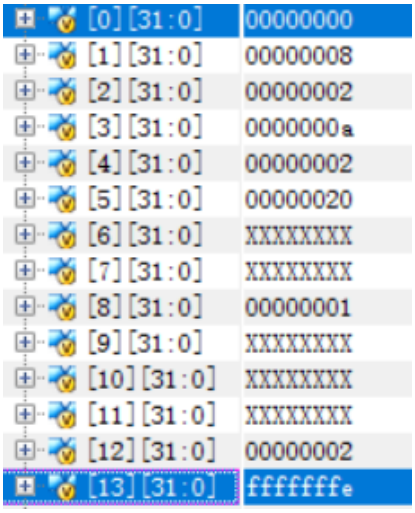
- jal 0x0000040



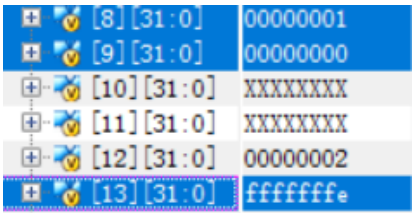
- slt $8,$12,$1

[1][31:0]	00000008
[2][31:0]	00000002
[3][31:0]	0000000a
[4][31:0]	00000002
[5][31:0]	00000020
[6][31:0]	XXXXXXXX
[7][31:0]	XXXXXXXX
[8][31:0]	00000001
[9][31:0]	XXXXXXXX
[10][31:0]	XXXXXXXX
[11][31:0]	XXXXXXXX
[12][31:0]	00000002

- addi $13,$0,-2

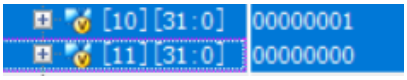


- slt $9,$8,$13



- sltiu $10,$9,2

- sltiu $11,$10,0



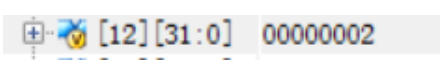
- addi $13,$13,1

- bltz $13,-2 (<0,转 34)

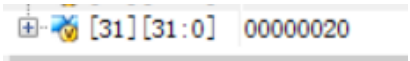
- j 0x000004C

- sw $2,4($1)

- lw $12,4($1)



- jr $31



- **halt**

运行过程可见指令并无错误。

六. 实验心得

本次多周期 CPU 的设计虽然从难度上来说应该是比单周期 CPU 设计难的，但是因为有了单周期的基础和知识，这次实验在知识学习方面并没有耗费像单周期那么多的时间，只需要对单周期的东西稍微加工，多周期的框架就出来了，但是这次的难点在于如何协调好各个单元与各个状态之间的数据传输、功能配合等，切割数据通路倒是最容易实现的环节，最耗费时间的是控制单元的 NextState 和 OutputFunction 部分，对于每个不同的指令需要调整不同的控制信号和状态数字，经常出现一些逻辑结构是对的但是却就是找不出原因的 bug，可能是我对于 vivado 这个软件的使用还不是很熟练，对于它内部一些默认值的定义还是不清楚，导致很多东西得出来的结果与预期不同，在无法 debug 的情况下只能重新寻求其他解决方案，这耗费了我大量时间，希望在本次实验结束之后可以对 vivado 的一些常见错误进行一个总结。

总体来说，这次实验还是很有意义的，帮助我了解到了一个真正可以生产使用的 CPU 是如何工作的，也进一步巩固了电路和计算机组成的知识，收获匪浅。