



《计算机组成原理与接口技术实验》 实验报告

(实验二)

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 5 月 26 日

成绩：

实验二：单周期CPU设计与实现

一. 实验目的

- (1) 掌握单周期 CPU 数据通路图的构成、原理及其设计方法；
- (2) 掌握单周期 CPU 的实现方法，代码实现方法；
- (3) 认识和掌握指令与 CPU 的关系；
- (4) 掌握测试单周期 CPU 的方法；
- (5) 掌握单周期CPU的实现方法。

二. 实验内容

设计一个单周期 CPU，该 CPU 至少能实现以下指令功能操作。指令与格式如下：

==> 算术运算指令

- (1) **add rd, rs, rt** (说明：以助记符表示，是汇编指令；以代码表示，是机器指令)

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

功能： $rd \leftarrow rs + rt$ 。reserved 为预留部分，即未用，一般填“0”。

- (2) **addi rt, rs, immediate**

000001	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

功能： $rt \leftarrow rs + (\text{sign-extend})\text{immediate}$ ；immediate 符号扩展再参加“加”运算。

- (3) **sub rd, rs, rt**

000010	rs(5 位)	rt(5 位)	rd(5 位)	reserved
--------	---------	---------	---------	----------

功能： $rd \leftarrow rs - rt$

==> 逻辑运算指令

- (4) **ori rt, rs, immediate**

010000	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

功能： $rt \leftarrow rs \mid (\text{zero-extend})\text{immediate}$ ；immediate 做“0”扩展再参加“或”运算。

- (5) **and rd, rs, rt**

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

功能： $rd \leftarrow rs \& rt$ ；逻辑与运算。

- (6) **or rd, rs, rt**

010010	rs(5 位)	rt(5 位)	rd(5 位)	reserved
--------	---------	---------	---------	----------

功能： $rd \leftarrow rs \mid rt$ ；逻辑或运算。

==> 移位指令

- (7) **sll rd, rt, sa**

011000	未用	rt(5 位)	rd(5 位)	sa	reserved
--------	----	---------	---------	----	----------

功能： $rd \leftarrow rt \ll (\text{zero-extend})sa$ ，左移 sa 位，(zero-extend)sa

==> 比较指令

- (8) **slti rt, rs, immediate** 带符号

011011	rs(5 位)	rt(5 位)	immediate (16 位)
--------	---------	---------	-------------------------

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

==> 存储器读/写指令

(9) sw rt, **immediate**(rs) 写存储器

100110	rs(5 位)	rt(5 位)	immediate (16 位)
--------	---------	---------	-------------------------

功能: memory[rs + (sign-extend)**immediate**] ← rt; **immediate** 符号扩展再相加。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。

(10) lw rt, **immediate**(rs) 读存储器

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

功能: rt ← memory[rs + (sign-extend)**immediate**]; **immediate** 符号扩展再相加。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

(11) beq rs, rt, **immediate**

110000	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

特别说明: **immediate** 是从 PC+4 地址开始和转移到的指令之间指令条数。**immediate** 符号扩展之后左移 2 位再相加。为什么要左移 2 位? 由于跳转到的指令地址肯定是 4 的倍数 (每条指令占 4 个字节), 最低两位是 “00”, 因此将 **immediate** 放进指令码中的时候, 是右移了 2 位的, 也就是以上说的 “指令之间指令条数”。

(12) bne rs, rt, **immediate**

110001	rs(5 位)	rt(5 位)	immediate
--------	---------	---------	------------------

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

特别说明: 与 beq 不同点是, 不等时转移, 相等时顺序执行。

==> 跳转指令

(13) j addr

111000	addr[27..2]
--------	-------------

功能: pc ← -(pc+4)[31..28], addr[27..2], 2{0}, 无条件跳转。

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 位地址了, 剩下最高 4 位由 pc+4 最高 4 位拼接上。

==> 停机指令

(14) halt

111111	00000000000000000000000000000000(26 位)
--------	--

功能: 停机; 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

单周期 CPU 指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。时钟周期一般也称振荡周期（如果晶振的输出没有经过分频就直接作为 CPU 的工作时钟，则时钟周期就等于振荡周期。若振荡周期经二分频后形成时钟脉冲信号作为 CPU 的工作时钟，这样，时钟周期就是振荡周期的两倍。）

CPU 在处理指令时，一般需要经过以下几个步骤：

(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。

单周期 CPU，是在一个时钟周期内完成这五个阶段的处理。

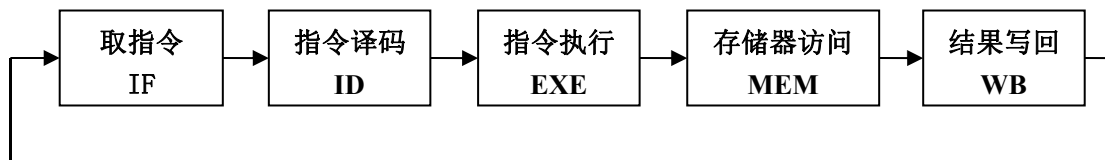


图 1 单周期 CPU 指令处理过程

MIPS 指令的三种格式：

R 类型:

31	26 25	21 20	16 15	11 10	6 5	0
op	rs	rt	rd	sa	funct	
6 位	5 位	5 位	5 位	5 位	6 位	

I 类型:

31	26 25	21 20	16 15	0
op	rs	rt	immediate	
6 位	5 位	5 位	16 位	

J 类型:

31	26 25	0
op	address	
6 位	26 位	

其中,

op: 为操作码;

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

rt: 可读可写。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: 只写。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与操作码配合使用;

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。

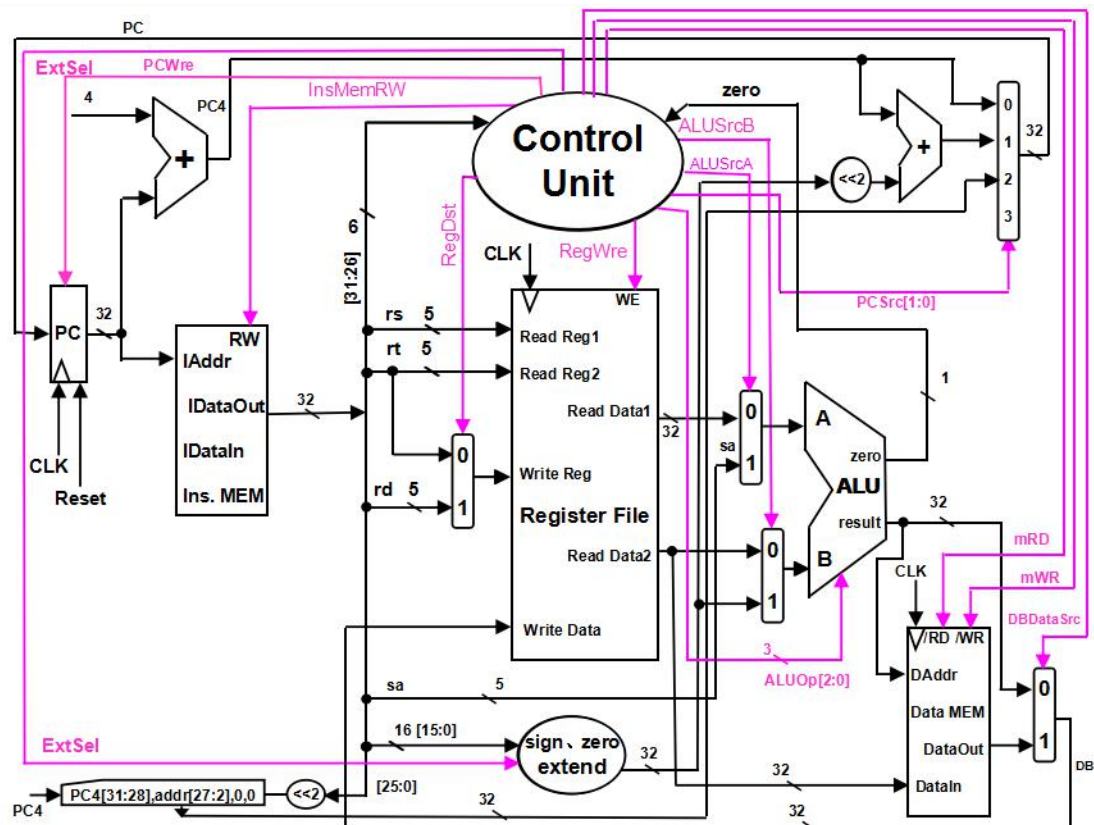


图 2 单周期 CPU 数据通路和控制线路图

图 2 是一个简单的基本上能够在单周期 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，先给出寄存器地址，读操作时，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。图中控制信号作用如表 1 所示，表 2 是 ALU 运算功能表。

表 1 控制信号的作用

控制信号名	状态 “0”	状态 “1”
Reset	初始化 PC 为 0	PC 接收新地址
PCWre	PC 不更改，相关指令：halt	PC 更改，相关指令：除指令 halt 外
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、addi、or、and、ori、beq、bne、slti、sw、lw	来自移位数 sa，同时，进行 (zero-extend)sa，即 $\{27\{0\}, sa\}$ ，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、or、and、sll、beq、bne	来自 sign 或 zero 扩展的立即数，相关指令：addi、ori、slti、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addi、sub、ori、or、and、slti、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、bne、sw、halt、j	寄存器组写使能，相关指令：add、addi、sub、ori、or、and、slti、sll、

		lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
mRD	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
RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令：addi、ori、lw、slti	写寄存器组寄存器的地址，来自 rd 字段，相关指令：add、sub、and、or、sll
ExtSel	(zero-extend)immediate(0 扩展)，相关指令：ori	(sign-extend)immediate (符号扩展)，相关指令：addi、slti、sw、lw、beq、bne
PCSrc[1..0]	00: pc←-pc+4，相关指令：add、addi、sub、or、ori、and、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)； 01: pc←-pc+4+(sign-extend)immediate，相关指令：beq(zero=1)、bne(zero=0)； 10: pc←-{(pc+4)[31:28],addr[27:2],2{0}}，相关指令：j； 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，指令存储器地址输入端口
- IDataIn，指令存储器数据输入端口（指令代码输入端口）
- IDataOut，指令存储器数据输出端口（指令代码输出端口）
- 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，数据存储器地址输入端口
- DataIn，数据存储器数据输入端口
- DataOut，数据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端口，其地址来源 rt 或 rd 字段
- 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

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

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	$Y = B \ll A$	B 左移 A 位
011	$Y = A \vee B$	或
100	$Y = A \wedge B$	与
101	$Y = (A < B) ? 1 : 0$	比较 A 与 B 不带符号
110	$Y = (((\text{rega} < \text{regb}) \&\& (\text{rega}[31] == \text{regb}[31])) \vee ((\text{rega}[31] == 1 \&\& \text{regb}[31] == 0))) ? 1 : 0$	比较 A 与 B 带符号
111	$Y = A \oplus B$	异或

需要说明的是以上数据通路图是根据要实现的指令功能的要求画出来的，同时，还必须确定 ALU 的运算功能(当然，以上指令没有完全用到提供的 ALU 所有功能，但至少必须能实现以上指令功能操作)。从数据通路图上可以看出控制单元部分需要产生各种控制信号，当然，也有些信号必须要传送给控制单元。从指令功能要求和数据通路图的关系得出以上表 1，这样，从表 1 可以看出各控制信号与相应指令之间的相互关系，根据这种关系就可以得出控制信号与指令之间的关系表（留给学生完成），再根据关系表可以写出各控制信号的逻辑表达式，这样控制单元部分就可实现了。

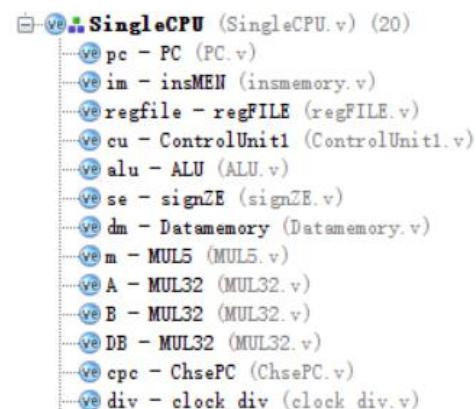
指令执行的结果总是在时钟下降沿保存到寄存器和存储器中，PC 的改变是在时钟上升沿进行的，这样稳定性较好。另外，值得注意的问题，设计时，用模块化、层次化的思想方法设计，关于如何划分模块、如何整合成一个系统等等，是必须认真考虑的问题。

四. 实验器材

电脑一台、Xilinx ISE 软件一套。

五. 实验分析与设计

1. 实验总体分析：



CPU整体架构如图所示，其中

PC: PC 单元, 接收下一条指令的地址, 输出当前指令地址

insMen: 指令存储器, 存放指令, 接收当前指令地址, 输出当前指令

regFile: 寄存器堆, 接收读取寄存器号, 输出对应寄存器的数据; 接受写寄存器号及写入的数据, 将数据写入该寄存器中

ControlUnit1: 控制单元, 接收指令 op 段, 输出相应的控制信号

ALU: 运算单元, 接收运算数及 ALUOp, 输出运算结果

signZE: 扩展单元, 接收立即数, 输出其零扩展或符号扩展

Datamemory: 数据存储器, 接收读数据地址, 输出该地址的数据; 接收写数据地址及写入的数据, 将数据写入该地址

ChsePC: PC 下一次执行地址选择单元, 根据控制信号判断是否分支或跳转至目标地址, 输出下一条指令的地址

MUL32: 接收控制信号及待选的数据, 输出其中一个数据, 数据宽度为 32 位, 用于 ALU 操作数的选择及写入寄存器数据的选择


MUL5: 接收控制信号及待选的数据, 输出其中一个数据, 数据宽度为 5 位, 用于写入寄存器号的选择

仿真测试:

根据下图所示的测试指令编写测试文件并测试 (采用文件读取的方式输入测试指令, 下载代码包后仿真需检测 insMen 文件代码中 test.txt 文件的位置是否正确)

地址	汇编程序	指令代码					
		op(6)	rs(5)	rt(5)	rd(5)/immediate(16)	16 进制数代码	
0x00000000	addi \$1,\$0,8	000001	00000	00001	0000 0000 0000 1000	= 04010008	
0x00000004	ori \$2,\$0,2	010000	00000	00010	0000 0000 0000 0010	= 40020002	
0x00000008	add \$3,\$2,\$1	000000	00010	00001	00011 00000 000000	= 004118000	
0x0000000C	sub \$5,\$3,\$2	000010	00011	00010	00101 00000 000000	= 08622800	
0x00000010	and \$4,\$5,\$2	010001	00101	00010	00100 00000 000000	= 44A22000	
0x00000014	or \$8,\$4,\$2	010010	00100	00010	01000 00000 000000	= 48824000	
0x00000018	sll \$8,\$8,1	011000	00000	01000	01000 00001 000000	= 60084040	
0x0000001C	bne \$8,\$1,-2 (≠,转 18)	110001	01000	00001	1111 1111 1111 1110	= C501FFFE	
0x00000020	slt \$6,\$2,\$1	011100	00010	00001	00110 00000 000000	= 70413000	
0x00000024	slt \$7,\$6,\$0	011100	00110	00000	00111 00000 000000	= 70C03800	
0x00000028	addi \$7,\$7,8	000001	00111	00111	0000 0000 0000 1000	= 04E70008	
0x0000002C	beq \$7,\$1,-2 (≠,转 28)	110000	00111	00001	1111 1111 1111 1110	= C0E1FFFE	
0x00000030	sw \$2,4(\$1)	100110	00001	00010	0000 0000 0000 0100	= 98220004	
0x00000034	lw \$9,4(\$1)	100111	00001	01001	0000 0000 0000 0100	= 9C290004	
0x00000038	bgtz \$9,\$1 (>0,转 40)	110010	01001	00000	0000 0000 0000 0001	= C9200001	
0x0000003C	halt	111111	00000	00000	0000000000000000	= FC000000	
0x00000040	addi \$9,\$0,-1	000001	00000	01001	1111 1111 1111 1111	= 0409FFFF	
0x00000044	j 0x00000038	111000	0000 0000 0000 0000 0000 1110 00			= E000000E	

1.addi \$1,\$0,8

	[1][31:0]	00000008	Array
---	-----------	----------	-------

2.ori \$2,\$0,2

	[2][31:0]	00000002	Array
---	-----------	----------	-------

3.add \$3,\$2,\$1

	[3][31:0]	0000000a	Array
---	-----------	----------	-------

4.sub \$5,\$3,\$2

	[5][31:0]	00000008	Array
---	-----------	----------	-------


5.and \$4,\$5,\$2

	[4][31:0]	00000000	Array
---	-----------	----------	-------









6.or \$8,\$4,\$2

	[8][31:0]	00000002	Array
---	-----------	----------	-------

7.sll \$8,\$8,1

	[8][31:0]	00000004	Array
---	-----------	----------	-------









8.bne \$8,\$1,-2 (≠,转 18)

	[1][31:0]	00000008	Array
	[2][31:0]	00000002	Array
	[3][31:0]	0000000a	Array
	[4][31:0]	00000000	Array
	[5][31:0]	00000008	Array
	[6][31:0]	00000000	Array
	[7][31:0]	00000000	Array
	[8][31:0]	00000004	Array

9.sll \$8,\$8,1

	[8][31:0]	00000008	Array
---	-----------	----------	-------

10.bne \$8,\$1,-2 (≠,转 18)

	[1][31:0]	00000008	Array
	[2][31:0]	00000002	Array
	[3][31:0]	0000000a	Array
	[4][31:0]	00000000	Array
	[5][31:0]	00000008	Array
	[6][31:0]	00000000	Array
	[7][31:0]	00000000	Array
	[8][31:0]	00000008	Array

11.slt \$6,\$2,\$1

	[6][31:0]	00000001	Array
--	-----------	----------	-------

12.slt \$7,\$6,\$0

	[7][31:0]	00000000	Array
--	-----------	----------	-------

13.addi \$7,\$7,8

	[7][31:0]	00000008	Array
--	-----------	----------	-------

14.beq \$7,\$1,-2 (≠,转 28)

	[1][31:0]	00000008	Array
	[2][31:0]	00000002	Array
	[3][31:0]	0000000a	Array
	[4][31:0]	00000000	Array
	[5][31:0]	00000008	Array
	[6][31:0]	00000001	Array
	[7][31:0]	00000008	Array

15.addi \$7,\$7,8

	[7][31:0]	00000010	Array
--	-----------	----------	-------

(ps: 此处为 16 进制 0000 0010)

16.beq \$7,\$1,-2 (≠,转 28)

	[1][31:0]	00000008	Array
	[2][31:0]	00000002	Array
	[3][31:0]	0000000a	Array
	[4][31:0]	00000000	Array
	[5][31:0]	00000008	Array
	[6][31:0]	00000001	Array
	[7][31:0]	00000010	Array

17.sw \$2,4(\$1)

	[2][31:0]	00000002	Array
--	-----------	----------	-------

18.lw \$9,4(\$1)

	[9][31:0]	00000002	Array
--	-----------	----------	-------

19.bgtz \$9,1(>0,转 40)

	[9][31:0]	00000002	Array
--	-----------	----------	-------

20.addi \$9,\$0,-1

	[9][31:0]	ffffffff	Array
--	-----------	----------	-------

21.j 0x00000038

	Nextadd[31:0]	00000038	Array
--	---------------	----------	-------

22.bgtz \$9,1 (>0,转 40)

 [9][31:0] ffffffff Array

23.halt

 PCWre 0 Logic

六. 实验心得

本次实验可以说是对我学习毅力和学习能力的一次巨大考验，因为在本次实验开始之初，理论课还并没有学习如何设计CPU，甚至我开始进行本次实验时都不知道单周期CPU的功能，对CPU的理解仅仅是它是一台计算机必不可少的部分。纵使这样，我还是没有选择直接放弃，先是通过百度搜索关于CPU的种种知识，比如最基本的它是什么，能干什么，结构如何等等问题，然后，看着网上一些关于制作单周期CPU的博客，先是通过模仿的办法，实现一些固有的功能，再进一步理解，编写出了自己的第一个单周期CPU。接下来又是头疼的时刻了，在仿真时，出现了各种各样的错误，vivado的查错工具我不太会用，也觉得不是很好用，索性使用最原始的办法，从错误的数据出发，一步一步往上进行追踪，然后得以找到错误的第一次产生之处，再对此处的代码进行进一步的思索。

对于制作本次CPU，我觉得对于我这样的硬件小白来说及时向同学请求帮助也是非常必要的，因为CPU的运作非常的复杂，环环相扣，有时候自己一个人想会陷入思维的死胡同，这时候就如果有一名局外高人帮你点拨迷津，可以快速走出困境，但是这样的坏处就是可能会失去很多思考的机会，所以我这次作业与一些可以形成互补的同学一起学习，相互解答疑惑，既达到了思考的目的，又防止一个人钻入死循环感到心力憔悴。

本次实验的遗憾是没能完成basy3上的CPU程序，不算一次完美的作业，但我觉得对于我这样一个在本次实验前对于CPU毫无一丝一毫知识储备的人来说，学到的知识已经很多了，在这方面来说，这次作业我已经很满足了，当然，还是希望在下次作业中做得更好。