《计算机组成原理实验》

实验报告

（实验二）

学 院 名 称 ：	数据科学与计算机学院
专业（班级）：	18 软件工程 2 班
时　　间 ：	2019 年 11 月 7 日
组　　员 ：	南樟 18342077 郑卓民 18342138

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。

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

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

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

（2）sub rd , rs , rt

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

功能：rd←rs - rt。

（3）addiu rt , rs ,**immediate**

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

功能：rt←rs + (sign-extend)**immediate；immediate** 符号扩展再参加“加”运算。

==> 逻辑运算指令

（4）andi rt , rs ,**immediate**

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

功能：rt←rs & (zero-extend)**immediate；immediate** 做“0”扩展再参加“与”运算。

（5）and rd , rs , rt

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

功能：rd←rs & rt；逻辑与运算。

（6）ori rt , rs ,**immediate**

010010	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs | (zero-extend)**immediate；immediate** 做“0”扩展再参加“或”运算。

（7）or rd , rs , rt

010011	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs | rt；逻辑或运算。

==>移位指令

（8）sll rd, rt,sa

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

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

==>比较指令

（9） slti rt, rs,**immediate** 带符号数

011100	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

（10）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。

（11）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

（12）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”。

（13）bne rs,rt,**immediate**

110001	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：与 beq 不同点是，不等时转移，相等时顺序执行。

（14）bltz rs,**immediate**

110010	rs(5 位)	00000	**immediate**(16 位)

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

==>跳转指令

（15）j addr

111000	addr[27:2]

功能：pc <一{(pc+4)[31:28],addr[27:2],2'b00}，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，剩下最高 4 位由 pc+4 最高 4 位拼接上。

==> 停机指令

（16）halt

111111	00000000000000000000000000(26 位)

功能：停机；不改变 PC 的值，PC 保持不变。

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。

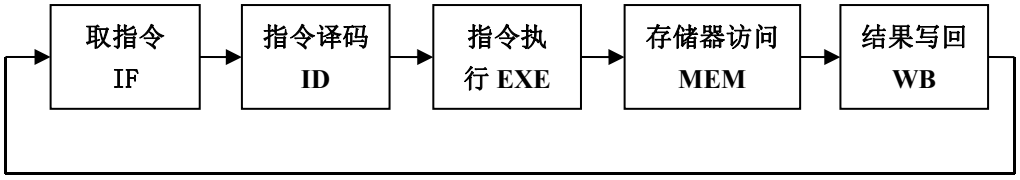
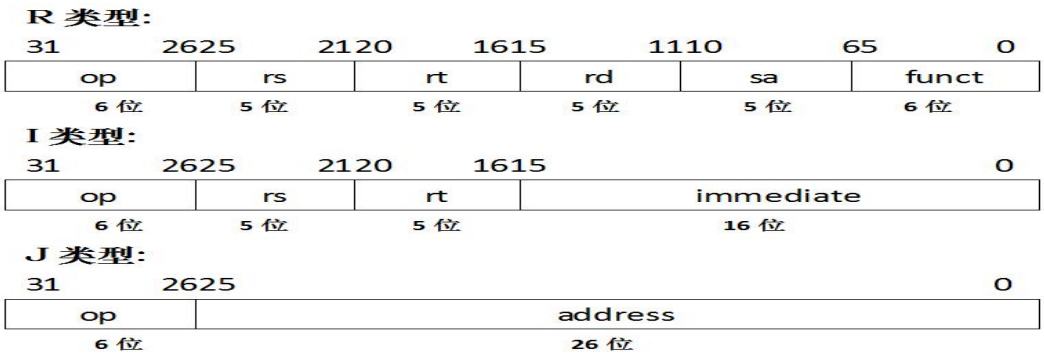


图 1 单周期 CPU 指令处理过程

MIPS 指令的三种格式：



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(Laod)/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。

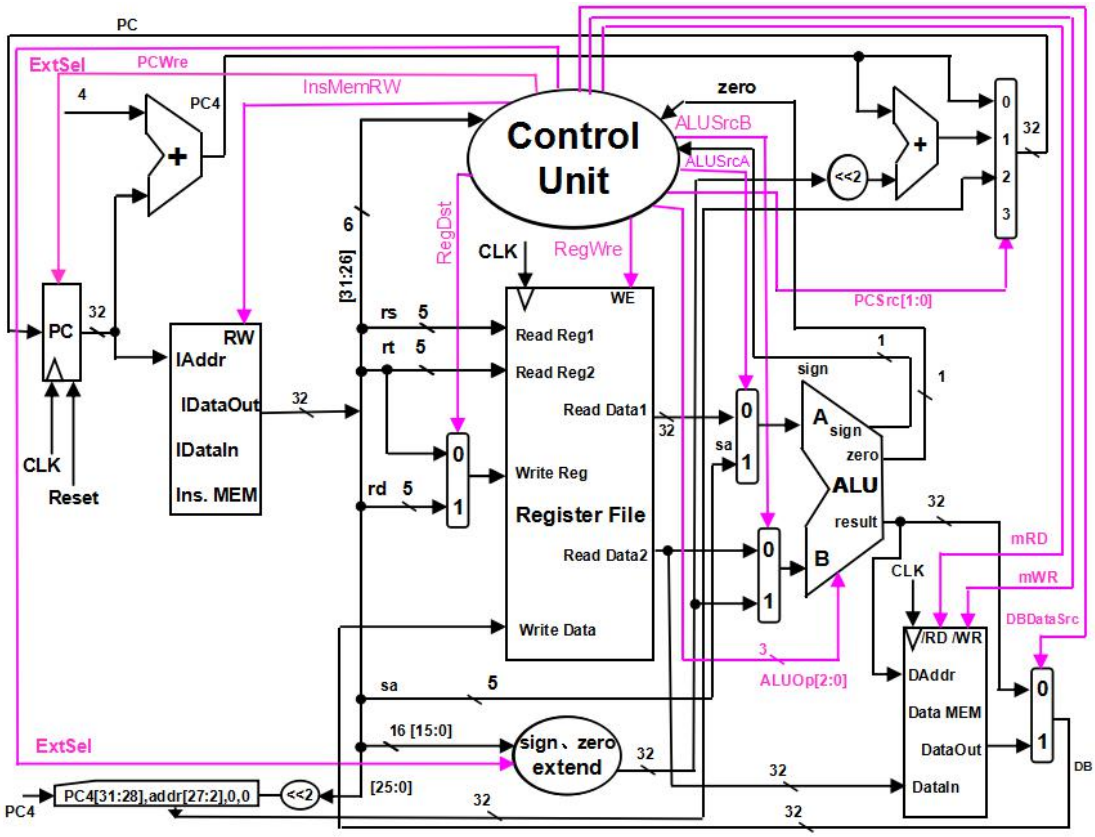


图 2　单周期 CPU 数据通路和控制线路图

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、addiu、or、and、andi、ori、slti、beq、bne、bltz、sw、	来自移位数 sa，同时，进行(zero-extend)sa，即{{27{1'b0}},sa}，相关指令：sll

	lw	
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、or、and、beq、bne、bltz	来自 sign 或 zero 扩展的立即数，相关指令：addi、andi、ori、slti、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addiu、sub、ori、or、and、andi、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、bltz、sw、halt	寄存器组写使能，相关指令：add、addiu、sub、ori、or、and、andi、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：addiu、andi、ori、slti、lw	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：andi、ori	(sign-extend)**immediate**（符号扩展），相关指令：addiu、slti、sw、lw、beq、bne、bltz
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addiu、sub、or、ori、and、andi、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、bltz(sign=0)； 01：pc<－pc+4+(sign-extend)**immediate<<2**，相关指令：beq(zero=1)、bne(zero=0)、bltz(sign=1)； 10：pc<－{(pc+4)[31:28],addr[27:2],2'b00}，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

IDataOut，指令存储器数据输出端口（指令代码输出端口）

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

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 = B<<A	B 左移 A 位
011	Y = A ∨ B	或
100	Y = A ∧ B	与
101	Y= (A<B) ?1: 0	比较 A<B 不带符号
110	Y=(((A<B)&&(A[31] == B[31])) ||((A[31]==1&& B[31] == 0))) ? 1:0	比较 A<B 带符号
111	Y = A ⊕ B	异或

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，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。

表 3 控制单元关系表

信号名/操作名	Reset	PCWre	ALU SrcA	ALU SrcB	DB Data Src	Reg Wre	Ins Mem RW	mRD	mWR	RegDst	ExtSel	PCSrc	ALUOp
add	1	1	0	0	0	1	1	0	0	1	\	00	000
addi	1	1	0	1	0	1	1	0	0	1	1	00	000
sub	1	1	0	0	0	1	1	0	0	1	\	00	001
ori	1	1	0	1	0	1	1	0	0	1	0	00	011
and	1	1	0	0	0	1	1	0	0	1	\	00	100

or	1	1	0	0	0	1	1	0	0	1	\	00	011
sll	1	1	1	0	0	1	1	0	0	1	0	00	010
slti	1	1	0	1	0	1	1	0	0	1	1	00	110
sw	1	1	0	1	\	0	1	0	1	0	1	00	000
lw	1	1	0	1	1	1	1	1	0	1	1	00	000
beq	1	1	0	0	\	0	1	0	0	0	1	00/01	001
bne	1	1	0	0	\	0	1	0	0	0	1	01/00	001
j	1	1	\	\	\	0	1	0	0	0	\	10	\
halt	\	0	\	\	\	\	\	\	\	\	\	\	\

此外，还要注意的是：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，用模块化的思想方法设计，ALU 设计、存储器设计、寄存器组设计等等。

关于所设计的CPU的检验方法，我们使用一个预先编写的指令文件，读入文件内容后观察测试结果是否如预期所想，测试内容如下：

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	addiu $1,$0,8	000010	00000	00001	0000 0000 0000 1000	=	08010008
0x00000004	ori $2,$0,2	010010	00000	00010	0000 0000 0000 0010	=	48020002
0x00000008	add $3,$2,$1	000000	00010	00001	00011 00000 000000	=	00411800
0x0000000C	sub $5,$3,$2	000001	00011	00010	00101 00000 000000	=	04622800
0x00000010	and $4,$5,$2	010001	00101	00010	00100 00000 000000	=	44A22000
0x00000014	or $8,$4,$2	010011	00100	00010	01000 00000 000000	=	4C824000
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	slti $6,$2,4	011100	00010	00110	0000 0000 0000 0100	=	70460004
0x00000024	slti $7,$6,0	011100	00110	00111	0000 0000 0000 0000	=	70C70000
0x00000028	addiu $7,$7,8	000010	00111	00111	0000 0000 0000 1000	=	08E70008
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	addiu $10,$0,-2	000010	00000	01010	1111 1111 1111 1110	=	080AFFFE

0x0000003C	addiu $10,$10,1	000010	01010	01010	0000 0000 0000 0001	=	094A0001
0x00000040	**bltz $10,-2(<0,转 3C)**	110010	01010	00000	1111 1111 1111 1110	=	C940FFFE
0x00000044	**andi $11,$2,2**	010000	00010	01011	0000 0000 0000 0010	=	404B0002
0x00000048	**j 0x00000050**	111000	00 0000 0000 0000 0000 0001 0100			=	E0000050
0x0000004C	or $8,$4,$2	010011	00100	00010	01000 00000 000000	=	4C824000
0x00000050	**halt**	111111	00000	00000	0000 0000 0000 0000	=	FC000000

将上表转换为txt文件，内容如下：

第一条指令到第十一条	第十二条指令到二十一条
00001000 00000001 00000000 00001000	11000000 11100001 11111111 11111110
01001000 00000010 00000000 00000010	10011000 00100010 00000000 00000100
00000000 01000001 00011000 00000000	10011100 00101001 00000000 00000100
00000100 01100010 00101000 00000000	00001000 00001010 11111111 11111110
01000100 10100010 00100000 00000000	00001001 01001010 00000000 00000001
01001100 10000010 01000000 00000000	11001001 01000000 11111111 11111110
01100000 00001000 01000000 01000000	01000000 01001011 00000000 00000010
11000101 00000001 11111111 11111110	11100000 00000000 00000000 00010100
01110000 01000110 00000000 00000100	01001100 10000010 01000000 00000000
01110000 11000111 00000000 00000000	11111100 00000000 00000000 00000000
00001000 11100111 00000000 00001000	

四. 实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五. 实验过程与结果

CPU的设计：

单周期CPU的设计主要考虑CPU在执行一条指令的过程中各模块所发挥的功能是什么，结合流程图，使用模块化设计的思想，可以将单周期CPU分解为几个大模块以及一些选择器，将各模块设计好之后，可分别进行各单元的测试仿真，确保各模块实现了对应的功能，最后

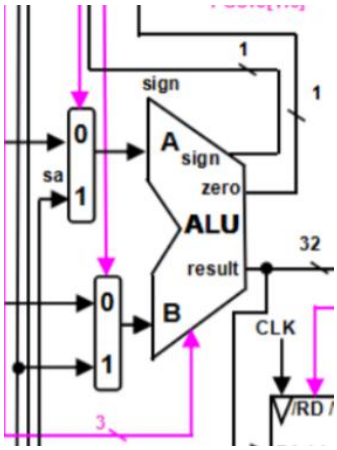
编写一个控制单元（ControlUnit），利用上面所编写的表3控制信号表，利用case语句即可决定信号名对应的值。

根据图2单周期CPU数据通路和控制线路图，将单周期CPU划分为以下模块：**控制单元（ControlUnit），地址管理（PC），指令存储器（InstructionMemory），寄存器组（RegisterFile），算术逻辑单元（ALU），数据存储器（DataMemory），零/符号扩展（ImmediateExtend），选择器（Mux4_32bits，Mux2_32bits，Mux2_5bits）。**

下面将按模块来进行分析如何通过代码实现所需功能。

算术逻辑单元（ALU）：

此部分设计思路是利用表二ALU功能表以及case语句，实现对于不同的ALUopcode实现不同的功能。



在此部分中，使用了如下输入输出接口：

1. input: ALUopcode, rega, regb
2. output: result, zero, sign

其中zero判断结果是否为0，sign判断结果是否为负数。

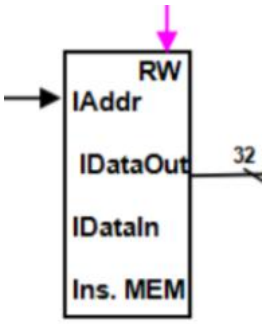
Case语句设计如下：

```
case (ALUopcode)
            3'b000 : result = rega + regb;
            3'b001 : result = rega - regb;
            3'b010 : result = regb << rega;
            3'b011 : result = rega | regb;
            3'b100 : result = rega & regb;
            3'b101 : result = (rega < regb)?1:0;
            3'b110 : begin
                        if((rega[31] == regb[31]) && (rega < regb)) result = 1;
```

```
                        else if(rega[31] == 1 && regb[31] == 0) result = 1;
                        else result = 0;
                    end
            3'b111 : result = rega ^ regb;
            default : begin
                        result = 8'h00000000;
                        $display (" no match");
                    end
    endcase
```

指令存储器（InstructionMemory）：

此部分的重点就在于，在初始化阶段（initial begin）能够读取已有文件中的指令，然后在后续阶段，可以根据地址（IAddr）的输入，输出对应的指令（IDataOut）。



易得，需要的接口为：

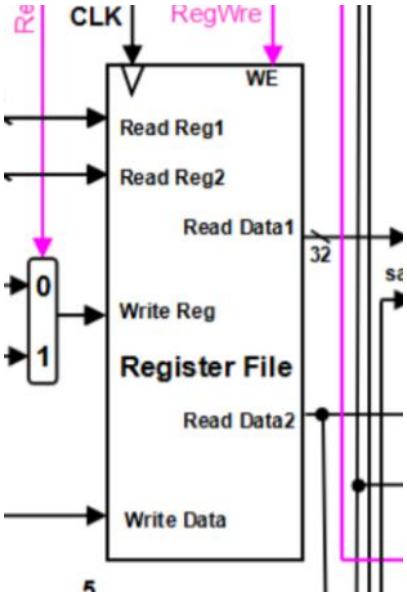
1. input: Iaddr
2. output: IDataOut

关键代码如下，同时使用到了一个数组变量（Container）来存储读入的指令代码：

```
initial begin
    $readmemb("D:/vivado_workplace/SingleCycleCpu/test.txt", Container);
end

assign IDataOut[31:24] = Container[IAddr+0];
assign IDataOut[23:16] = Container[IAddr+1];
assign IDataOut[15:8]  = Container[IAddr+2];
assign IDataOut[7:0]   = Container[IAddr+3];
```

寄存器组（RegisterFile）：

在该部分，主要实现寄存器的读与写功能，主要注意为了提高稳定性，读和写操作分为上升沿和下降沿分别完成，其中还要添加一个标识变量来判断是否需要ReSet将寄存器的值都变为0。



所需的接口：

1. input: CLK, RST, RegWre, ReadReg1, ReadReg2, WriteReg, WriteData

2. output: ReadData1, ReadDate2

关键代码如下：

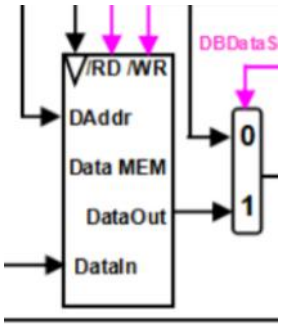
```
1.  reg [31:0] regFile[1:31];
2.  integer i;
3.  assign ReadData1 = (ReadReg1 == 0) ? 0 : regFile[ReadReg1];
4.  assign ReadData2 = (ReadReg2 == 0) ? 0 : regFile[ReadReg2];
5.
6.  always @ (negedge CLK or negedge RST) begin
7.                  if (RST==0) begin  for(i=1;i<32;i=i+1)  regFile[i] <= 0;   end
8.                  else if(RegWre == 1 && WriteReg != 0)  regFile[WriteReg] <= WriteData;
9.              end
```

其中，下降沿来临时，当RegWre信号为1且所写寄存器不是$0时，才能将WriteData写入到regFile [WriteReg] 中。

数据存储器（DataMemory）：

数据存储器就像一个数据中转中心，主要功能在于通过判断nRD和nWR的值，来执行读取对应地址的值，或者写入值进对应地址。

除了需要输入nRD、nWR信号以及输入的地址和数据外，还需要CLK信号来实现下降沿才进行写操作。



因此，接口为：

1. input: clk, address, writeData, nRD, nWR
2. output: Dataout

```
1.  reg [7:0] ram [0:60];
2.
3.  assign Dataout[7:0] = (nRD==0)?ram[address + 3]:8'bz;
4.  assign Dataout[15:8] = (nRD==0)?ram[address + 2]:8'bz;
5.  assign Dataout[23:16] = (nRD==0)?ram[address + 1]:8'bz;
6.  assign Dataout[31:24] = (nRD==0)?ram[address ]:8'bz;
7.
8.  always@( negedge clk ) begin
9.      if( nWR==0 ) begin
10.         ram[address] <= writeData[31:24];
11.         ram[address+1] <= writeData[23:16];
12.         ram[address+2] <= writeData[15:8];
13.         ram[address+3] <= writeData[7:0];
14.     end
15. end
```

其中nRD为0时，正常读；为1,输出高组态z；nWR为0时，正常写；为1，无操作。

零/符号扩展（ImmediateExtend）：

根据具体指令，有的（如ori）需要对立即数进行零扩展，有的（如beq）需要对立即数进行符号扩展，因此在本模块中，主要根据信号ExtSel的值来选择进行零扩展还是符号扩展，然后返回一个扩展为32位之后的值。

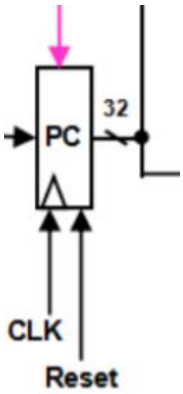
所需接口：

1. input: origin, ExtSel
2. output: extend

```
always @(*) begin
    extend[15:0] <= origin;
    if(ExtSel == 0) begin
        extend[31:16] <= 0;
    end
    else begin
        if(origin[15] == 0) extend[31:16] <= 0;
        else extend[31:16] <= 16'hFFFF;
    end
end
```

地址管理（PC）：

在本模块中，主要考虑：什么时候不需要更改地址？什么时候需要更改地址？显然，我们有信号Reset和PCWre来做标识，PCWre为1时，允许PC更改，为0时，不更改；当Reset为0时，将地址重置为0，当Reset为1时，允许PC接收新地址，值得注意的是，在本模块里，只有当前地址和下一地址两种地址变量，对于地址的偏移量计算以及地址的选择等操作，均不在本模块中实现。因此，只需考虑何使需要将下一地址赋给当前地址。



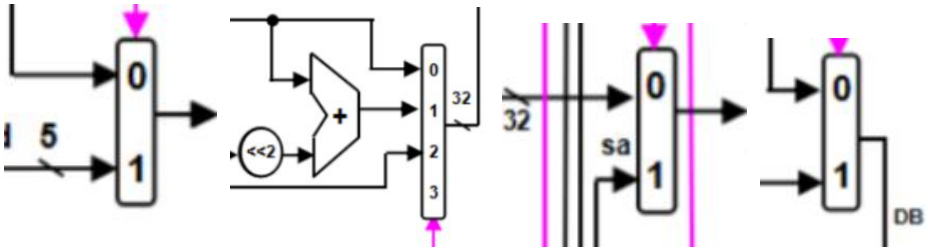
接口如下：

1. input: clk, Reset, PCWre, nextIAddr
2. output: currentIAddr

```
initial currentIAddr <= 0;

always @(posedge clk or negedge Reset) begin
    if(Reset == 0) currentIAddr <= 0;
    else begin
        if(PCWre == 1) currentIAddr <= nextIAddr;
        else currentIAddr <= currentIAddr;
    end
end
```

选择器（Mux4_32bits, Mux2_32bits, Mux2_5bits）：

观察单周期CPU数据通路和控制线路图，会发现有多个数据选择器，且规格不一，对于不同的规格的选择器，我们不妨各写一个对应规格的选择器模块，在需要的地方就调用对应选择器即可，可见，分别有32位的4选选择器、32位的2选选择器、5位的2选选择器。



每个选择器的代码主要通过一个choice输入以及对应的多个输入，然后通过case语句来决定输出哪个输入。

Mux4_32bits：

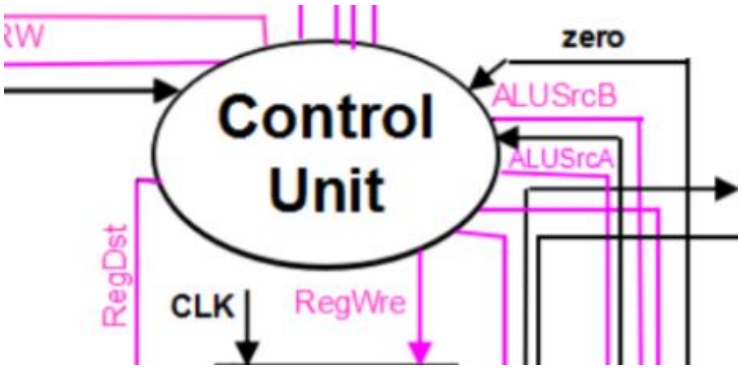
```
always @(choice or in0 or in1 or in2 or in3) begin
    case(choice)
        2'b00: out = in0;
        2'b01: out = in1;
        2'b10: out = in2;
        2'b11: out = in3;
        default: out = 0;
    endcase
end // Mux4_32bits
```

Mux2_5bits、Mux2_32bits：

```
1.  // Mux2_5bits
2.  assign out = (choice==0) ? in0 : in1;
```

控制单元（ControlUnit）：

此部分的作用为根据指令的op码来决定该指令执行的时对应的各信号的取值。主要用到就是case语句，然后根据上面已编写的信号关系表，对各信号进行赋值。



接口详情：

1. input: opcode, zero, sign

2. output PCWre, ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRW, mRD, mWR, RegDst, ExtSel, PCSrc, ALUOp

此处有一点需要注意：always的触发条件处，不能只有opcode。

原因在于：在执行类似beq、bne、bltz的指令时，除了在opcode到来时要对信号进行初始化外，当ALU计算出结果后，还要对PCSrc信号进行一次更新，如此才能正常实现跳转功能，而beq、bne的执行则于result是否为0（zero）有关，当zero为1时候，PCSrc设为01，反之，设置为00；bltz指令则于sign（result是否为负数），当sign为1时候，PCSrc设为1，反之，设为0。因此always的触发条件，必须还要包含在ALU中算得的zero和sign。

关键的case语句：

```
1.  always @(opcode or zero or sign) begin
2.      case(opcode)
3.          //·····省略·····
4.          6'b110000: begin    // beq
5.              PCWre <= 1;
6.              {ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRW, mRD, mWR, RegDst, ExtSel} <= 9'
    b000010001;
```

```
            PCSrc[1:0] <= (zero==1) ? 2'b01 : 2'b00;
            ALUOp[2:0] <= 3'b001;
        end
        6'b110001: begin    // bne
            PCWre <= 1;
            {ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRW, mRD, mWR, RegDst, ExtSel} <= 9'b000010001;
            PCSrc[1:0] <= (zero==0) ? 2'b01 : 2'b00;
            ALUOp[2:0] <= 3'b001;
        end
        6'b110010: begin    // bltz
            PCWre <= 1;
            {ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRW, mRD, mWR, RegDst, ExtSel} <= 9'b000010001;
            PCSrc[1:0] <= (sign==1) ? 2'b01 : 2'b00;
            ALUOp[2:0] <= 3'b000;
        end
    endcase
end
```

顶层模块（CPU）：

本模块用于整合上面所写的模块，合并为一个单周期CPU，分配各种端口信号。

下面以PC和ALU模块为例（其他模块的设置类似）：

```
PC PC(
    .clk(clk), .Reset(Reset), .PCWre(PCWre), .nextIAddr(nextIAddr),
    .currentIAddr(currentIAddr)
);
ALU ALU(
    .ALUopcode(ALUopcode), .rega(rega), .regb(regb),
    .result(result), .zero(zero), .sign(sign)
);
```

CPU的仿真：

设计好单周期CPU后，就需要编写一个仿真文件来验证所编写的代码是否能实现所需功能，仿真得到波形后，通过验证每一条指令对应的波形段中的各值是否与期望结果相符来判断CPU的功能是否有误。

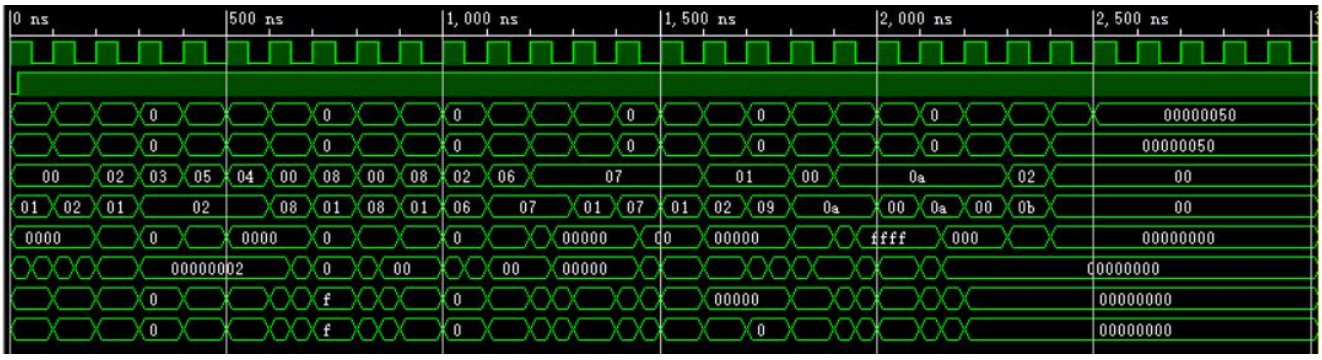
在仿真中，我们希望波形中包含的信息有clk，reset，currentIAddr，nextIAddr，rs，rt，ReadData1，ReadData2，ALU_Result，DataBus：

```
CPU uut(
    .clk(clk),
    .Reset(Reset),
    .currentIAddr(currentIAddr), .nextIAddr(nextIAddr),
    .rs(rs), .rt(rt),
    .ReadData1(ReadData1), .ReadData2(ReadData2),
    .ALU_result(ALU_result), .DataBus(DataBus)
);
```

测试的关键代码就是将clk设置成每隔50ns反转一次，而sim的initial begin初始化阶段则要设置仿真时间（3000ns），同时还要注意仿真开始前要设置一小段时间（20ns）来让reset初始化完成，否则会出错（不稳定）。

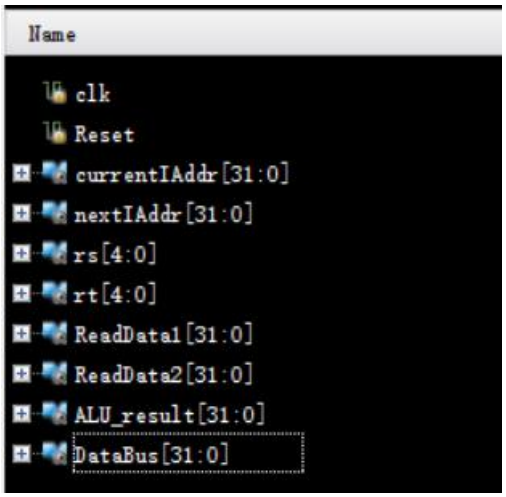
```
initial begin
    clk = 1;
    Reset = 0;
    #20; //等待 20ns 来让 reset 完成
    Reset = 1;
    #3000;
    $stop;
end

always #50 clk = ~clk; //时钟周期设置为 100ns
```

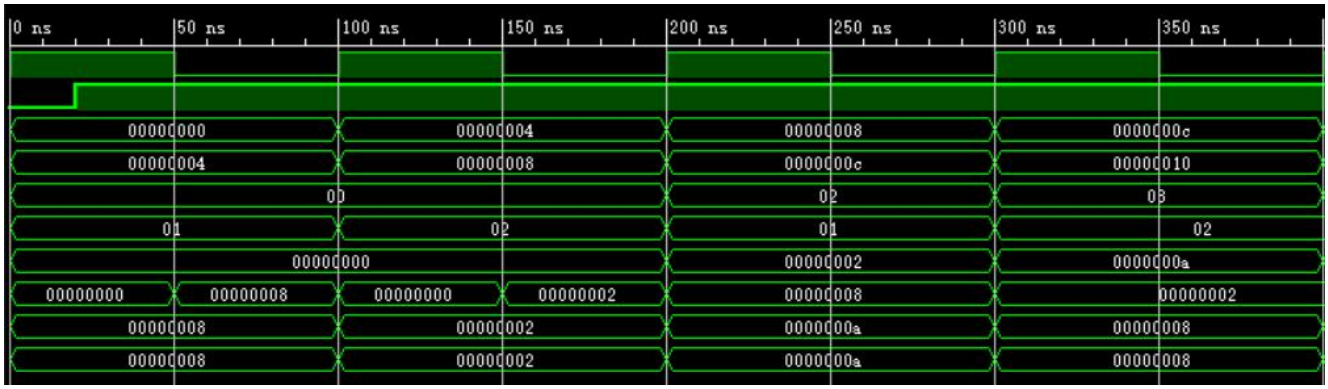
整体波形如下：



结合我们的测试指令集，对波形进行细致分析：

其中仿真读取的信号名为：





从左到右分别为以下四条指令：

1. **addiu $1,$0,8 ：** 在0ns-100ns区间内，可见当前地址0，下一地址为4，rs为0，rt为1，Data1为0($0)，Data2为8(immediate)，Result为0+8=8，DataBus为8。

寄存器情况：$1 = 8

2. **ori $2,$0,2 ：** 在100ns-200ns区间内，可见当前地址4，下一地址为8，rs为0，rt为2，Data1为0($0)，Data2为2(immediate)，Result为0+2=2，DataBus为2。

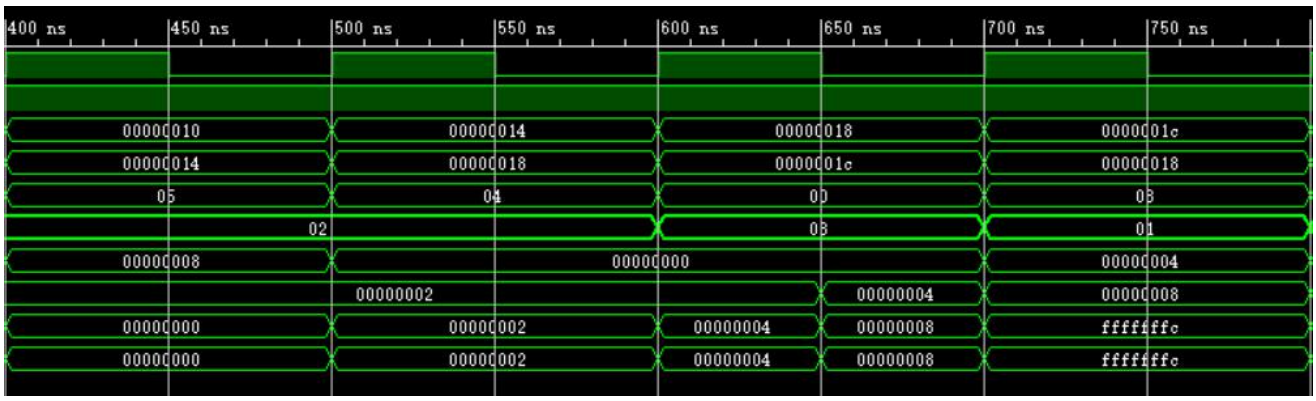
寄存器情况：$1 = 8; $2 = 2;

3. **add $3,$2,$1 ：** 在200ns-300ns区间内，可见当前地址8，下一地址为c，rs为2，rt为1，Data1为2($2)，Data2为8($1)，Result为2+8=a，DataBus为a。

寄存器情况：$1 = 8; $2 = 2; $3 = a;

4. **sub $5,$3,$2 ：** 在300ns-400ns区间内，可见当前地址c，下一地址为10，rs为3，rt为2，Data1为a($3)，Data2为2($2)，Result为a-2=8，DataBus为8。

寄存器情况：$1 = 8; $2 = 2; $3 = a; $5 = 8;



从左到右分别为以下四条指令：

1. **and $4,$5,$2** ： 在400ns-500ns区间内，可见当前地址10，下一地址为14，rs为5，rt为2，Data1为8($5)，Data2为2($2)，Result为8 & 2=0，DataBus为0。

寄存器情况：$1 = 8; $2 = 2; $3 = a; $4 = 0; $5 = 8;

2. **or $8,$4,$2** ： 在500ns-600ns区间内，可见当前地址14，下一地址为18，rs为4，rt为2，Data1为0($4)，Data2为2($2)，Result为0 | 2=2，DataBus为2。

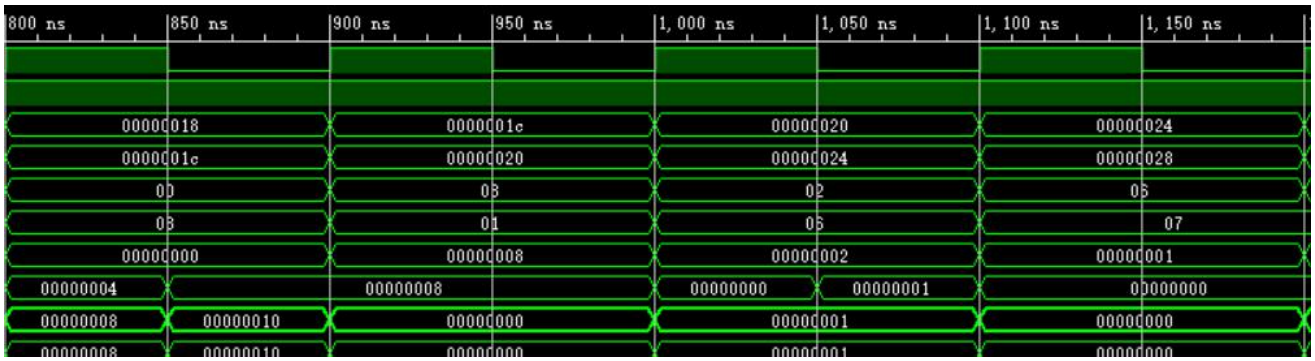
寄存器情况：$1 = 8; $2 = 2; $3 = a; $4 = 0; $5 = 8; $8 = 2;

3. **sll $8,$8,1** ： 在600ns-700ns区间内，可见当前地址18，下一地址为1c，rs为0(没用到)，rt为8，Data1为0($0)(没用到)，Data2为2($8)，Result为2 << 1=4，DataBus为4。

寄存器情况：$1 = 8; $2 = 2; $3 = a; $4 = 0; $5 = 8; $8 = 4;

4. **bne $8,$1,-2 （≠,转18）**： 在700ns-800ns区间内，可见当前地址1c，下一地址为18，rs为8，rt为1，Data1为4($8)，Data2为8($1)，Result为4-8=-4(true)，DataBus为-4。两个寄存器的值不相等，所以地址返回到18。

寄存器情况：$1 = 8; $2 = 2; $3 = a; $4 = 0; $5 = 8; $8 = 4;



从左到右分别为以下四条指令：

1. **sll $8,$8,1** ： 在800ns-900ns区间内，可见当前地址18，下一地址为1c，rs为0(没用到)，rt为8，Data1为0($0)(没用到)，Data2为4($8)，Result为4 << 1=8，DataBus为8。

寄存器情况：$1 = 8; $2 = 2; $3 = a; $4 = 0; $5 = 8; $8 = 8;

2. **bne $8,$1,-2 (=,转20)** ： 在900ns-1000ns区间内，可见当前地址1c，下一地址为20，rs为8，rt为1，Data1为8($8)，Data2为8($1)，Result为8-8=0(false)，DataBus为0。两个寄存器的值相等，所以地址正常+4。

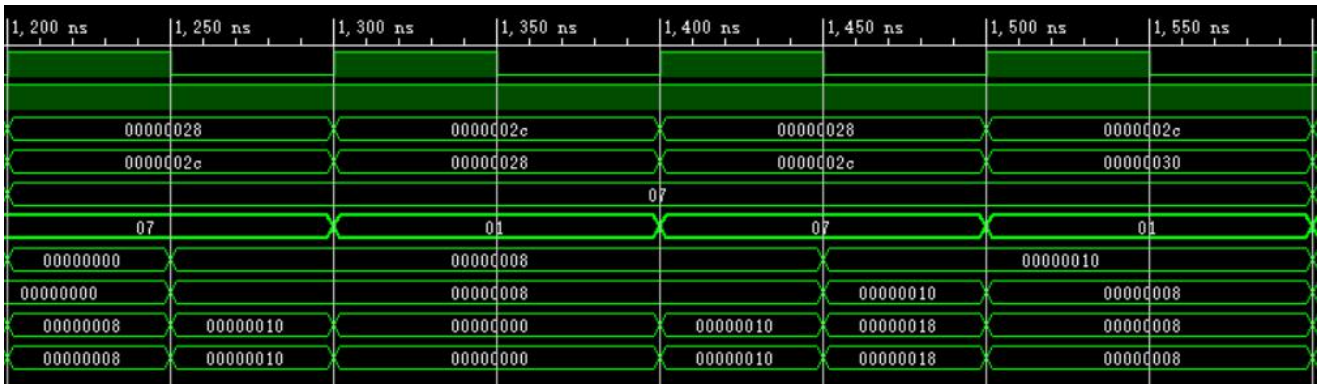
寄存器情况：$1 = 8; $2 = 2; $3 = a; $4 = 0; $5 = 8; $8 = 8;

3. **slti $6,$2,4** ： 在1000ns-1100ns区间内，可见当前地址20，下一地址为24，rs为2，rt为6，Data1为2($2)，Data2为0($6)，Result为1(2<4(immediate))，DataBus为1。

寄存器情况：$1 = 8; $2 = 2; $3 = a; $4 = 0; $5 = 8; $6 = 1; $8 = 8;

4. **slti $7,$6,0** ： 在1100ns-1200ns区间内，可见当前地址24，下一地址为28，rs为6，rt为7，Data1为1($6)，Data2为0($7)，Result为0(1>0)，DataBus为0。

寄存器情况：$1 = 8; $2 = 2; $3 = a; $4 = 0; $5 = 8; $6 = 1; $7 = 0; $8 = 8;



从左到右分别为以下四条指令：

1. **addiu $7,$7,8** ： 在1200ns-1300ns区间内，可见当前地址28，下一地址为2c，rs为7，rt为7，Data1为0($7)，Data2为0($7)，Result为0+8=8，DataBus为8。

寄存器情况：$1 = 8; $2 = 2; $3 = a; $4 = 0; $5 = 8; $6 = 1; $7 = 8; $8 = 8;

2. **beq $7,$1,-2 (=,转28)** ： 在1300ns-1400ns区间内，可见当前地址2c，下一地址为28，rs为7，rt为1，Data1为8($7)，Data2为8($1)，Result为8-8=0(true)，DataBus为0。两个寄存器的值相等，所以地址返回到28。

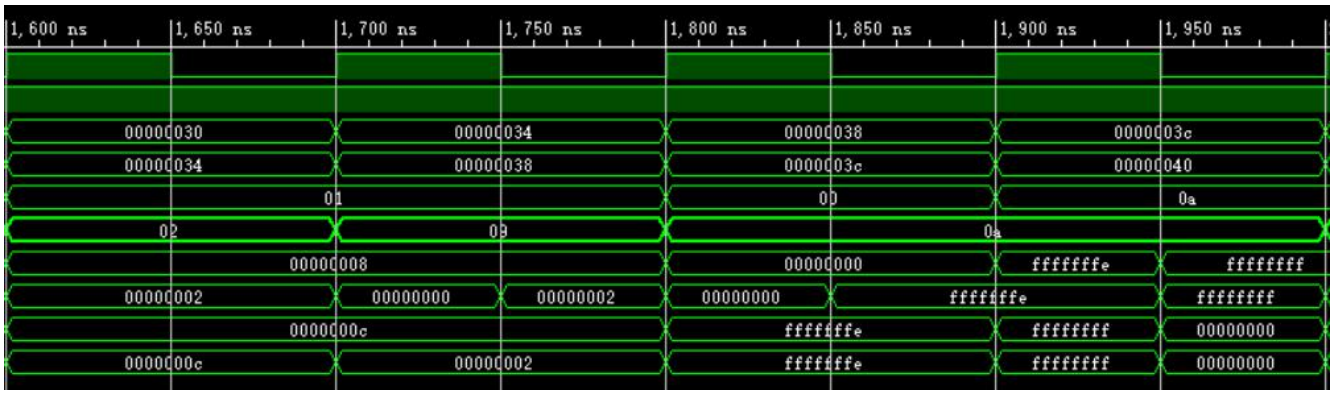
寄存器情况：$1 = 8; $2 = 2; $3 = a; $4 = 0; $5 = 8; $6 = 1; $7 = 8; $8 = 8;

3. **addiu $7,$7,8 ：** 在1400ns-1500ns区间内，可见当前地址28，下一地址为2c，rs为7，rt为7，Data1为8($7)，Data2为8($7)，Result为8+8=10，DataBus为10。

寄存器情况：$1 = 8; $2 = 2; $3 = a; $4 = 0; $5 = 8; $6 = 1; $7 = 10(hex); $8 = 8;

4. **beq $7,$1,-2 ：** 在1500ns-1600ns区间内，可见当前地址2c，下一地址为30，rs为7，rt为1，Data1为10(hex)($7)，Data2为8($1)，Result为10-8=8(false)，DataBus为8。两个寄存器的值不相等，所以地址正常+4。

寄存器情况：$1 = 8; $2 = 2; $3 = a; $4 = 0; $5 = 8; $6 = 1; $7 = 10(hex); $8 = 8;



从左到右分别为以下四条指令：

1. sw $2,4($1) ： 在1600ns-1700ns区间内，可见当前地址30，下一地址为34，rs为1，rt为2，Data1为8($1)，Data2为2($2)，Result为8+4=c(数据存储器地址)，DataBus为c(数据存储器地址)。此指令效果即将2($2中的值)写入到地址c的数据存储器中。

寄存器情况：$1 = 8; $2 = 2; $3 = a; $4 = 0; $5 = 8; $6 = 1; $7 = 10(hex); $8 = 8;

2. lw $9,4($1) ： 在1700ns-1800ns区间内，可见当前地址34，下一地址为38，rs为1，rt为9，Data1为8($1)，Data2为0($9)，Result为8+4=c，DataBus为2。此指令效果即将地址为12(c)的数据存储器中的值(2)存入到寄存器$9中。**显然这个2就是上一条指令中存进去的。**

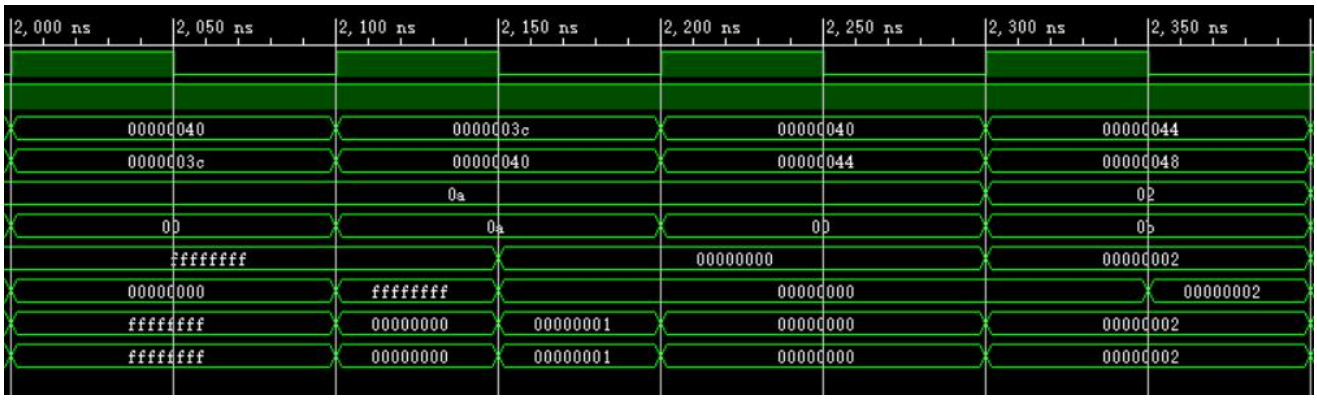
寄存器情况：$1 = 8; $2 = 2; $3 = a; $4 = 0; $5 = 8; $6 = 1; $7 = 10(hex); $8 = 8; $9 = 2;

3. addiu $10,$0,-2： 在1800ns-1900ns区间内，可见当前地址38，下一地址为3c，rs为0，rt为a，Data1为0($0)，Data2为0($10)，Result为0+(-2)=-2，DataBus为-2。

寄存器情况：$1 = 8; $2 = 2; $3 = a; $4 = 0; $5 = 8; $6 = 1; $7 = 10(hex); $8 = 8; $9 = 2; $10 = -2;

4. addiu $10,$10,1： 在1900ns-2000ns区间内，可见当前地址3c，下一地址为40，rs为a，rt为a，Data1为-2($10)，Data2为-2($10)，Result为-2+1=-1，DataBus为-1。

寄存器情况：$1 = 8; $2 = 2; $3 = a; $4 = 0; $5 = 8; $6 = 1; $7 = 10(hex); $8 = 8; $9 = 2; $10 = -1;



从左到右分别为以下四条指令：

1. **bltz $10,-2(<0,转3C)** ： 在2000ns-2100ns区间内，可见当前地址40，下一地址为3c，rs为a，rt为0，Data1为-1($10)，Data2为0($0)，Result为-1-0=-1，DataBus为-1。由于$10中的值小于0，所以条件成立，跳转回地址为3c的指令。

寄存器情况：$1 = 8; $2 = 2; $3 = a; $4 = 0; $5 = 8; $6 = 1; $7 = 10(hex); $8 = 8; $9 = 2; $10 = -1;

2. **addiu $10,$10,1**： 在2100ns-2200ns区间内，可见当前地址3c，下一地址为40，rs为a，rt为a，Data1为-1($10)，Data2为-1($10)，Result为-1+1=0，DataBus为0。

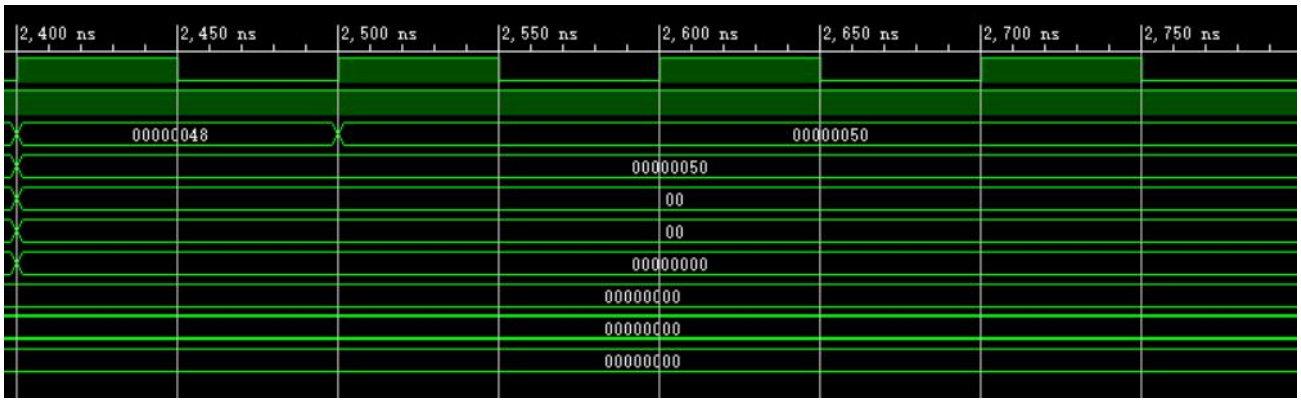
寄存器情况：$1 = 8; $2 = 2; $3 = a; $4 = 0; $5 = 8; $6 = 1; $7 = 10(hex); $8 = 8; $9 = 2; $10 = 0;

3. **bltz $10,-2**： 在2200ns-2300ns区间内，可见当前地址40，下一地址为44，rs为a，rt为0，Data1为0($10)，Data2为0($0)，Result为0-0=0，DataBus为0。0不小于0，条件不成立，地址正常+4执行指令。

寄存器情况：$1 = 8; $2 = 2; $3 = a; $4 = 0; $5 = 8; $6 = 1; $7 = 10(hex); $8 = 8; $9 = 2; $10 =0;

4. **andi $11,$2,2**： 在2300ns-2400ns区间内，可见当前地址44，下一地址为48，rs为2，rt为b，Data1为2($2)，Data2为2，Result为2&2=2，DataBus为2。

寄存器情况：$1 = 8; $2 = 2; $3 = a; $4 = 0; $5 = 8; $6 = 1; $7 = 10(hex); $8 = 8; $9 = 2; $10 = -1; $11 = 2;



从左到右分别为以下三条指令：

1. **j 0x00000050**：在2400ns-2500ns区间内，跳转语句，当前地址为48，下一地址为50。
2. **or $8,$4,$2**（被跳过，无执行）
3. **halt**：在2500ns之后的区间，halt指令使得PC保持不变，信号、数值都保持不变了。

Basys3数码管显示结果：

设计完CPU后，我们考虑使用Basys3板来运行所设计的CPU，通过4个数码显示器来查看CPU执行指令的情况。

目标：

实现数码管通过button可选择显示的内容：（左边为左两位数码管显示的内容，右边为右边两位数码管显示的内容）

1、当前指令地址PC:下条指令地址PC

2、RS寄存器地址:RS寄存器数据

3、RT寄存器地址:RT寄存器数据

4、ALU结果输出:DB总线数据

原理：

1. 说明：指令存储器中的指令地址范围：0~255；数据存储器中的数据地址范围：0~255。也就是只使用低8位。

2. 开关说明： （以下数据都来自CPU）（SW15、SW14、SW0为Basys3板上开关名，BTNR为按键名）

开关SW_in (SW15、SW14)状态情况如下。

显示格式： 左边两位数码管BB ： 右边两位数码管BB。

以下是数码管的显示内容。

SW_in = 00：显示 当前 PC值:下条指令PC值

SW_in = 01：显示 RS寄存器地址:RS寄存器数据

SW_in = 10：显示 RT寄存器地址:RT寄存器数据

SW_in = 11：显示 ALU结果输出 :DB总线数据。

复位信号（reset）接开关SW0，按键（单脉冲）接按键BTNR。

另外：

1、7段数码管的位控信号AN3-AN0，每组编码中只有一位为0（亮），其余都是1（灭）。

2、七段数码显示器编码与引脚对应关系为(左到右,高到低)：七段共阳极数码管->1gfedcba;七段共阴极数码管->0gfedcba。

3、必须有足够的刷新频率，频率太高或太低都不成，系统时钟必须适当分频，否则效果达不到。指令执行采用单步（按键控制）执行方式，由开关（SW15、SW14）控制选择查看数码管上的相关信息，地址和数据。

设计思路：

1、实现CPU在板上运行需要两个时钟信号，CPU工作时钟和Basys3板系统时钟。CPU工作时钟即为按键，是CPU正常工作时钟信号，按键必须进行消抖处理；Basys3板系统时钟即为板提供的正常工作时钟信号，即为100MHZ。Basys3板系统时钟信号引脚对应管脚W5。

2、每个按键周期,4个数码管都必须刷新一次。数码管位控信号 AN3-AN0是1110、1101、1011、0111，为0时点亮该数码管，当然，还应该为数码管各位“1gfedcba”引脚输出信号，最高位为“1”。比如，“当前PC值”低8位中的高4位和低4位，必须经下页转换后送给数码管各引脚。

显示模块设计大概分为4个部分：

（1）对Basys3板系统时钟信号进行分频，分频的目的用于计数器；

（2）生成计数器，计数器用于产生4个数。这4数用于控制4个数码管；

（3）根据计数器产生的数生成数码管相应的位控信号（输出）和接收CPU来的相应数据；

（4）将从CPU 接收到的相应数据转换为数码管显示信号，再送往数码管显示（输出）。还必须清楚，数码管显示的内容是受开关控制的，不同情况显示内容是不同的。

下面将根据上面根据显示模块需求所分的四部分编写代码：

分频：

```
always @(posedge clk) begin
    if(div_counter >= 50000) begin  //Basys3 板上 w5 管脚提供了一个 100MHz 的时钟
        clk_sys <= ~clk_sys;
        div_counter <= 0;
    end
    else div_counter <= div_counter + 1;
end
```

计数器：

```
always @(posedge clk) begin
    if(count == 2'b11) count <= 0;  // 复位
    else count <= count + 1;  // 计数
end
```

生成位控信号：

```
always @(*) begin
    case(choice)
        0: out <= in0;
        1: out <= in1;
        2: out <= in2;
        3: out <= in3;
    endcase
end
```

将数据转换为数码管显示信号：

```
always @(hex) begin
    //  '0'- 亮灯，'1'- 熄灯
    // [7:0] 从左到右：DP,g,f,e,d,c,b,a
    case(hex)
        4'h0: dispcode <= 8'b1100_0000; //0
        4'h1: dispcode <= 8'b1111_1001; //1
        4'h2: dispcode <= 8'b1010_0100; //2
        4'h3: dispcode <= 8'b1011_0000; //3
        4'h4: dispcode <= 8'b1001_1001; //4
        4'h5: dispcode <= 8'b1001_0010; //5
        4'h6: dispcode <= 8'b1000_0010; //6
        4'h7: dispcode <= 8'b1101_1000; //7
```

```
        4'h8: dispcode <= 8'b1000_0000; //8
        4'h9: dispcode <= 8'b1001_0000; //9
        4'hA: dispcode <= 8'b1000_1000; //A
        4'hB: dispcode <= 8'b1000_0011; //b
        4'hC: dispcode <= 8'b1100_0110; //C
        4'hD: dispcode <= 8'b1010_0001; //d
        4'hE: dispcode <= 8'b1000_0110; //E
        4'hF: dispcode <= 8'b1000_1110; //F
        default: dispcode <= 8'b0000_0000;
    endcase
end
```

除此之外，我们是使用basys3板上的按钮来使CPU指令的运行，使用到板上的按钮，就要考虑按键抖动的时长问题，否则则会出现按一次按钮，运行了多条指令的情况。因此，我们需要对按钮进行消抖处理，消抖的原理是按下按钮后延迟一段时间(例如20ms或25ms等等)才接受下一次按下。

代码如下：

```
module Button_Debounce
(
    clk,
    btn_in,
    btn_out
);
    input clk;
    input btn_in;
    output btn_out;

    reg key_reg1,key_reg2,key_out;
    reg [24:0]count;

    always @( posedge clk)
        begin
            count<=count+1;
            if(count==250000)
                begin
                    key_reg1<=btn_in;
                    count<=0;
                end
            key_reg2<=key_reg1;
            key_out<=key_reg2&(!key_reg1);
         end
```

```
    assign btn_out = key_out;
endmodule
```

上述模块都写完之后，我们就可以着手与Basys3板的CPU整合模块，我们编写一个顶层文件Basys3_CPU.v来控制CPU在basys3板上的实现情况。

易知Basys3_CPU.v的作用是将从basys3板上获得的信号以及与板子有关的输出信号，进行整合并分配部分给CPU模块，使得CPU的运算结果能够在数码管上进行显示。

```
module Basys3_CPU(
    input basys3_clk,
    input reset,
    input [1:0] SW_in,  // 展示内容选择
    input next_button,
    output [3:0] enable,  // 位选
    output [7:0] displaycode
    );
    wire [31:0] currentIAddr, nextIAddr;
    wire [4:0] rs, rt;
    wire [31:0] ReadData1, ReadData2;
    wire [31:0] ALU_result, DataBus;
    wire next_signal;  // 消抖后的信号
    wire [15:0] Display;  // 数码管上要展示的信息数组（16 位），分配时候按 4 位为单位获取一个位的数字（hex）
    CPU CPU(
        // 省略
    );
    LED LED(
        // 省略
    );
    Mux4_16bits Mux4_16bits(
        // 省略
    );
    Button_Debounce Button_Debounce(
       // 省略
    );
endmodule
```

烧录的过程中，还会遇到一个错误报告：

Vivado [Place 30-574] Poor placement for routing between an IO pin and BUFG

原因在于：我们自己实现了分频器，而没有用到板子上的晶振，也就是E3管脚，但是又有时钟信号，如此导致的后果就是有一个信号作为了一个同步电路的时钟输入，但这个信号不是同步的信号。

解决办法就是在XDC文件中添加：**set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets clk_in]**

正确的xdc文件应类似如下（特别注意最后一行对按钮输入接口的处理）：

```
set_property IOSTANDARD LVCMOS33 [get_ports {dispcode[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dispcode[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dispcode[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dispcode[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dispcode[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dispcode[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dispcode[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dispcode[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {enable[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {enable[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {enable[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {enable[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW_in[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW_in[0]}]
set_property PACKAGE_PIN R2 [get_ports {SW_in[1]}]
set_property PACKAGE_PIN T1 [get_ports {SW_in[0]}]
set_property PACKAGE_PIN V7 [get_ports {dispcode[7]}]
set_property PACKAGE_PIN U7 [get_ports {dispcode[6]}]
set_property PACKAGE_PIN V5 [get_ports {dispcode[5]}]
set_property PACKAGE_PIN U5 [get_ports {dispcode[4]}]
set_property PACKAGE_PIN V8 [get_ports {dispcode[3]}]
set_property PACKAGE_PIN U8 [get_ports {dispcode[2]}]
set_property PACKAGE_PIN W6 [get_ports {dispcode[1]}]
set_property PACKAGE_PIN W7 [get_ports {dispcode[0]}]
set_property PACKAGE_PIN W4 [get_ports {enable[3]}]
set_property PACKAGE_PIN V4 [get_ports {enable[2]}]
set_property PACKAGE_PIN U4 [get_ports {enable[1]}]
set_property PACKAGE_PIN U2 [get_ports {enable[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports next_button]
set_property IOSTANDARD LVCMOS33 [get_ports reset_sw]
set_property PACKAGE_PIN V17 [get_ports reset_sw]
set_property PACKAGE_PIN W19 [get_ports next_button]
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets next_button]
set_property IOSTANDARD LVCMOS33 [get_ports basys3_clock]
set_property PACKAGE_PIN W5 [get_ports basys3_clock]
```

下面结合Basys3运行的实际情况，对单周期CPU的设计正确性做最后的检验：

指令	当前地址:下一地址	RS地址:RS值	RT地址:RT值	ALU结果:DB数据
addiu $1,$0,8				
ori $2,$0,2				
add $3,$2,$1				
sub $5,$3,$2				
and $4,$5,$2				
or $8,$4,$2				
sll $8,$8,1				
bne $8,$1,-2				
sll $8,$8,1				
bne $8,$1,-2				
slti $6,$2,4				
slti $7,$6,0				
addiu $7,$7,8				
beq $7,$1,-2				
addiu $7,$7,8				
beq $7,$1,-2				
sw $2,4($1)				
lw $9,4($1)				
addiu$10,$0,-2				

addiu$10,$10,1				
bltz $10,-2				
addiu$10,$10,1				
bltz $10,-2				
andi $11,$2,2				
j 0x00000050				
or $8,$4,$2	X	X	X	X
halt				

六. 实验心得

1. 本次实验的最大收获，就是对Verilog代码的编写有了更深的理解和更强的综合实践应用，对一些小概念（小细节）问题也有了答案，例如reg和wire的区别（基于时序逻辑的时候就用reg，组合逻辑一般用wire。always中只能用reg，才能对其赋值。）

2. 通过本次实验，将理论课中对单周期CPU的理论学习付诸实践，不仅将补全了理论课中漏学的细节知识，更让理论课中学到的东西，真正能够熟练的运用起来。

3. 在本次实验中，设计的过程秉承着模块化设计的思想，将Basys3板的显示设计与CPU的逻辑设计分割，将CPU的逻辑设计又分割为多个子模块，这样模块化的设计思想，让大的任务分割为小的任务，让看似不能完成的大苦难变成了可以一步一步完成的阶梯式过程，同时，这种模块化设计的思路也影响了其他编程学科的学习，加深了印象。

4. 在仿真的过程中，一开始遇到了一个很奇怪的现象，在仿真代码中我设置了仿真3000ns，然而波形图中仅仅仿真了1000ns，我尝试修改代码仿真2000ns，结果波形图仍然为1000ns，因此我判断出现这种情况其实与仿真代码无关（仿真代码无错误），我开始尝试查看仿真的设置（setting），终于，在仿真时间的默认设置里，我看到了1000ns的默认值，因此，我把默认值修改为5000ns，此时所编写的仿真代码中的3000ns就能起作用了。

5. 由于我们使用了按钮来模仿时钟的上升沿和下降沿来作为cpu的clk输入，而按钮会出现信号抖动的现象，这种不稳定的上下抖动，就会造成按一次按钮就运行了好多条指令的情况，

为了解决这个问题，特意编写了一个按键消抖模块，原理就是简单地在接收到按钮信号后执行25ms的延迟。

6. 初次烧录basys3板后，运行情况出现了偏差，最重要的还是代码仿真的结果是没有任何问题的，而basys3的实际执行情况却在地址18-1c之间反复交替，需要按几十次才会跳转到20地址以后的指令。返回到测试指令集中，可以得知，出问题的指令是一个分支判断语句，判断$8和$1是否不相等，在我们的预期中，只要对$8寄存器进行两次左移操作，两者的值就会都是8而使得判断不成立，不跳转。那么造成这种两个地址之间来回跳转的情况，显然是两个寄存器的值总是不相等，在basys3板上调整SW_in，查看rs和rt的值，发现$1时钟为0（而它理应为8），回溯到$1的赋值语句，发现是第一条指令，查看寄存器情况，果然没有存储成功，$1仍然为0。

经过思考，出现这种情况的原因如下：

（a）仿真的过程中，我设定了暂停20ns，来让需要reset的东西正常reset，**且来让第一条指令是通过上升沿加下降沿完成。**

（b）我们知道我们设定的寄存器的写入是在下降沿完成的（原因在于上面我们所说的稳定性问题）

（c）因此问题显然是由于在basys3板上，一开始并没有下降沿，无法写入，而按下按钮后，直接读取了下一条地址，因此，第一条指令并没有执行。

考虑解决方法：

我尝试在reset置位时，允许对寄存器进行一次写入。

于是我将registerfile中的代码添加了如下：

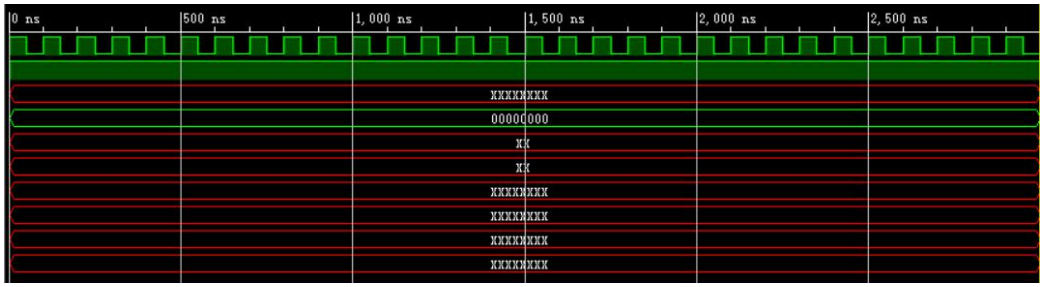
```
always @ (posedge RST) begin
                    if(RegWre == 1 && WriteReg != 0)
                        regFile[WriteReg] <= WriteData;
end
```

为了验证是否成功，我将仿真代码中的一部分代码注释掉：

```
initial begin
    clk = 1;
    Reset = 0;
```

```
4.      // #20; //等待 20ns 来让 reset 完成
5.       Reset = 1;
6.       #3000;
7.       $stop;
8.   end
```

可惜的是，失败了，如下图所示。考虑到可能因为在并行方面的问题，导致这样的做法其实没用甚至会导致代码的执行出错。



寻求另一种解决办法，既然要真正地执行完整第一条指令，那么就需要满足下述条件：reset为1且获得一个下降沿。那么我们可以让在basys3上执行的开始修改为，先按住button，然后将reset从0拨到1，然后松开button，此时产生的下降沿，就将该存入寄存器中的writedata，都成功存入，就不会对后续指令执行产生影响。

7. 除此之外，在烧录的过程中，还遇到了一个前所未见的错误，所幸通过百度错误信息内容，找到了问题发生的根源在于使用的是按钮来替代clk的功能而没有使用basys3上提供的时钟，且扫描时钟信号的实现使用到了分频器。解决：手动在约束文件（xdc文件）添加了一条指令后，达到了类似屏蔽此类警告的效果，运行就成功了。

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	addiu $1,$0,8	000010	00000	00001	0000 0000 0000 1000	=	08010008
0x00000004	ori $2,$0,2	010010	00000	00010	0000 0000 0000 0010	=	48020002
0x00000008	add $3,$2,$1	000000	00010	00001	00011 00000 000000	=	00411800
0x0000000C	sub $5,$3,$2	000001	00011	00010	00101 00000 000000	=	04622800
0x00000010	and $4,$5,$2	010001	00101	00010	00100 00000 000000	=	44A22000
0x00000014	or $8,$4,$2	010011	00100	00010	01000 00000 000000	=	4C824000
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	slti $6,$2,4	011100	00010	00110	0000 0000 0000 0100	=	70460004

0x00000024	slti $7,$6,0	011100	00110	00111	0000 0000 0000 0000	=	70C70000
0x00000028	addiu $7,$7,8	000010	00111	00111	0000 0000 0000 1000	=	08E70008
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	addiu $10,$0,-2	000010	00000	01010	1111 1111 1111 1110	=	080AFFFE
0x0000003C	addiu $10,$10,1	000010	01010	01010	0000 0000 0000 0001	=	094A0001
0x00000040	**bltz $10,-2(<0,转 3C)**	110010	01010	00000	1111 1111 1111 1110	=	C940FFFE
0x00000044	**andi $11,$2,2**	010000	00010	01011	0000 0000 0000 0010	=	404B0002
0x00000048	**j 0x00000050**	111000	00 0000 0000 0000 0000 0001 0100			=	E0000050
0x0000004C	or $8,$4,$2	010011	00100	00010	01000 00000 000000	=	4C824000
0x00000050	**halt**	111111	00000	00000	0000 0000 0000 0000	=	FC000000