中山大學
SUN YAT-SEN UNIVERSITY

《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	：	16 计算机类 4 班
学生姓名	：	郑育聪
学　　号	：	16337329
时　　间	：	2017　年　11　月　18　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←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←rs + (sign-extend)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←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←rs | (zero-extend)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←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←rs | 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< - rt<<(zero-extend)sa，左移 sa 位　，(zero-extend)sa

==>比较指令

（8） slt　rd, rs, rt　　带符号数

011100	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

==> 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。

（13）bgtz　rs,immediate

110010	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],0,0}，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

（15）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。

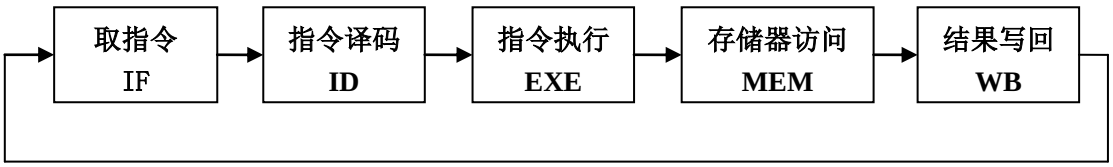
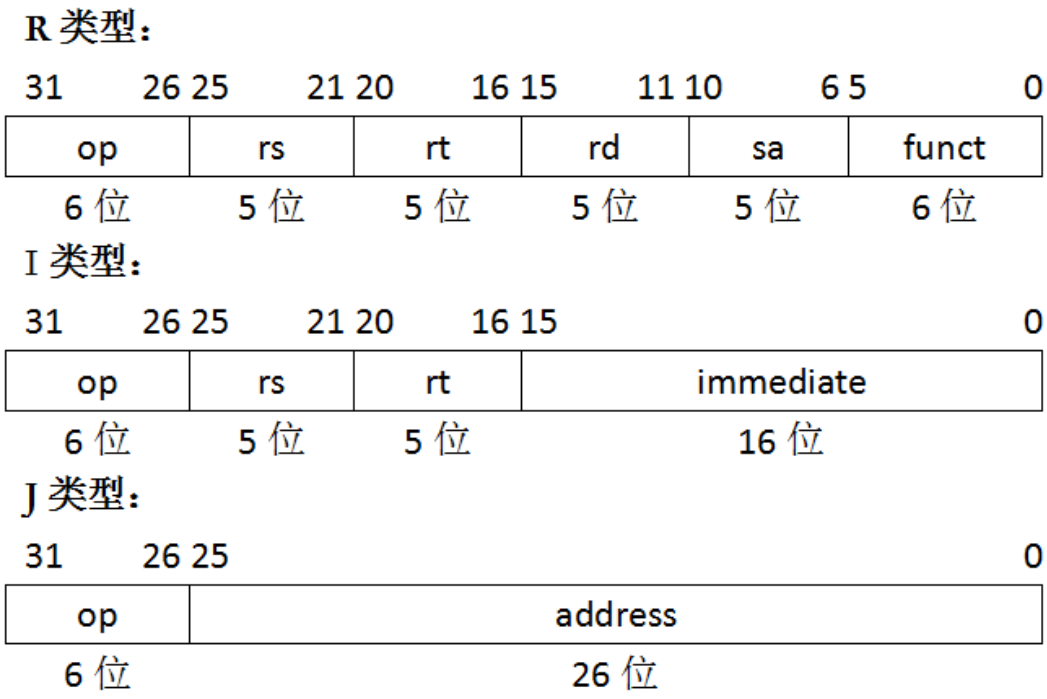


图 1　单周期 CPU 指令处理过程

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（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。

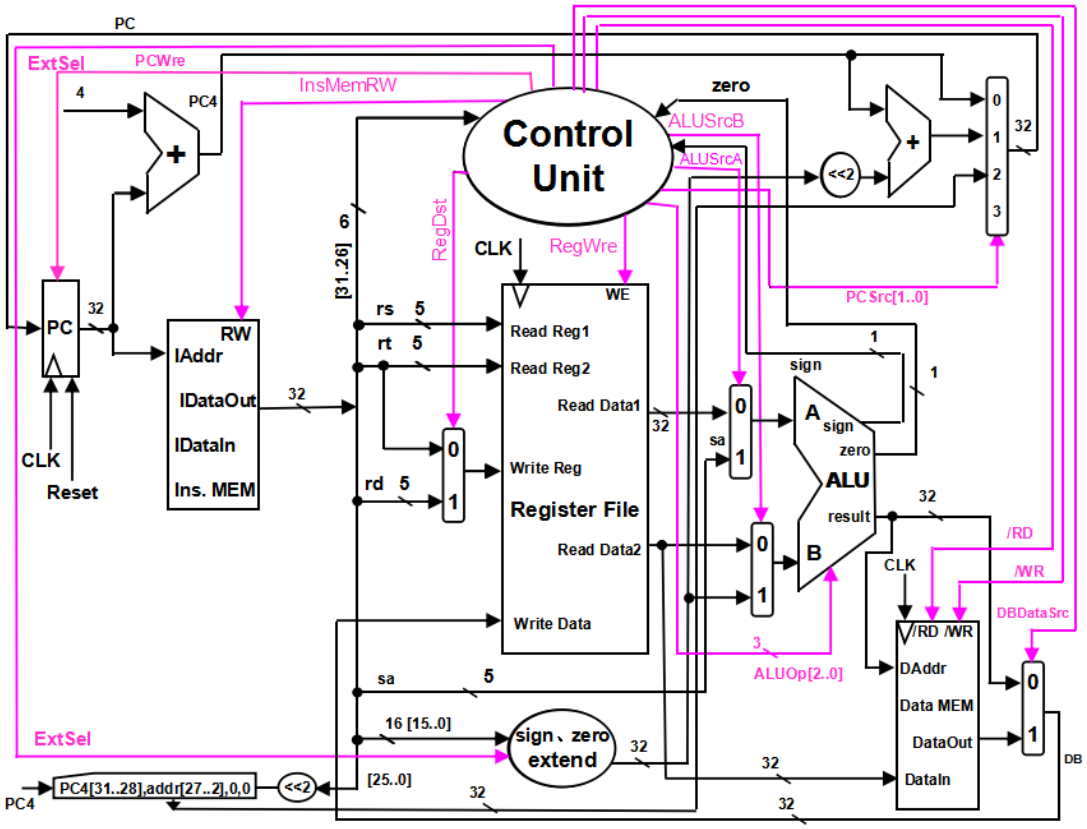


图 2　单周期 CPU 数据通路和控制线路图

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 3 为各指令对应的控制信号。

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、bgtz、slt、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、slt、beq、bne、bgtz	来自 sign 或 zero 扩展的立即数，相关指令：addi、ori、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addi、sub、ori、or、and、slt、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、bgtz、sw、halt、j	寄存器组写使能，相关指令：add、addi、sub、ori、or、and、slt、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
/RD	读数据存储器，相关指令：lw	输出高阻态
/WR	写数据存储器，相关指令：sw	无操作
RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令：addi、ori、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、slt、sll

ExtSel	(zero-extend)immediate（0 扩展），相关指令：ori，sll	(sign-extend)immediate（符号扩展），相关指令：addi、sw、lw、bne、bne、bgtz
PCSrc[1..0]	00：pc<--pc+4，相关指令：add、addi、sub、or、ori、and、slt、sll、sw、lw、beq(zero=0)、bne(zero=1)、bgtz(sign=1，或 zero=1)； 01：pc<--pc+4+(sign-extend)immediate，相关指令：beq(zero=1)、bne(zero=0)、bgtz(sign=0，zero=0)； 10：pc<--{(pc+4)[31..28],addr[27..2],0,0}，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

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	无符号比较 A 与 B
110	**if (A<B &&(A[31] == B[31])) Y = 1;** **else if (A[31] && !B[31]) Y = 1;** **else Y = 0;**	带符号比较 A 与 B
111	Y = A ⊕ B	异或

表 3 控制信号真值表

指令 / signal	R-Type	ori	addi	sll	sw	lw	branch	j	halt
ALUSrcA	0	0	0	1	0	0	0	0	0
ALUSrcB	0	1	1	0	1	1	0	0	0
DBDataSrc	0	0	0	0	0	1	0	0	0
RegWre	1	1	1	1	0	1	0	0	0
InsMemRW	1	1	1	1	1	1	1	1	1
RD	1	1	1	1	1	0	1	1	1
WR	1	1	1	1	0	1	1	1	1
RegDst	1	0	0	1	0	0	0	0	0
ExtSel	1	0	1	0	1	1	1	1	1

其中为使各信号的真值表达式较为简洁，对不影响的控制信号的值做了相应设置，即在不影响有关指令的情况下设置为 0 或 1。

四、实验设备

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

五、仿真 CPU 实验过程

实验分析

根据上述原理分析及数据通路，将单周期 CPU 分为以下几个模块（完整代码在压缩文件中，此处只显示关键代码）：

PCUnit：PC 单元，接收下一条指令的地址，输出当前指令地址

module PCUnit(input clk, Reset, PCWre, input [31:0] nextPC, output reg [31:0] curPC, PC4);

initial begin

PC4 = 32'H00000000; curPC = 32'H00000000; //初始化为 0

end

always@ (posedge clk or negedge Reset) begin

if (Reset == 1'b0) begin

curPC = 32'h00000000; PC4 = 32'h00000000; //reset 为 0，归零

end

else if (PCWre == 1'b1) begin

curPC = nextPC; PC4 = curPC + 4; //PC4 为当前 PC + 4

end

end

endmodule

InstructMem：指令存储器，存放指令，接收当前指令地址，输出当前指令

RegisterFile：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

ControlUnit：控制单元，接收指令 op 段，输出相应的控制信号

module ControlUnit(input sign, zero, Reset, input [5:0] op, output reg PCWre, ExtSel, InsMemRW, RegDst, RegWre, ALUSrcA, ALUSrcB, RD, WR, DBDataSrc, output reg [1:0] PCSrc, output reg [2:0] ALUOp);

always@(sign or zero or op or Reset)begin

```
        if (op == 6'b111000) PCSrc <= 2'b10;
        else if (op == 6'b110000 && zero == 1) PCSrc <= 2'b01;
        else if (op == 6'b110001 && zero == 0) PCSrc <= 2'b01;
        else if (op == 6'b110010 && sign == 0 && zero == 0) PCSrc <= 2'b01;
        else PCSrc <= 2'b00;  //根据 op、sign 和 zero 位判断是否分支或跳转
        PCWre <= (op != 6'b111111 && Reset != 0);   //halt 指令或复位时，置为 0
        ALUSrcA <= (op == 6'b011000); //   sll 指令时操作数来自 sa
        ALUSrcB <= (op == 6'b000001 || op == 6'b010000 || op == 6'b100110 || op ==
6'b100111); //ori、addi、sw、lw 时操作数为立即数
        ......//其余控制信号类似，写出真值表达式即可
        end
    endmodule
```

ALU：运算单元，接收运算数及 ALUOp，输出运算结果

ExtendUnit：扩展单元，接收立即数，输出其零扩展或符号扩展

DataMem：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

PcChoose：PC 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

```
    module PcChoose(input [1:0] PCSrc, input [31:0] PC4, Extend_Imme, input
[31:0]instruction, output reg [31:0] PCnext);
      initial begin
        PCnext = 0;
```

```
    end
    always @(PCSrc or PC4 or Extend_Imme or instruction)begin
      case(PCSrc)
        2'b00: PCnext <= PC4; //若为 00，下一条指令地址即为 PC4 = PC + 4
        2'b01: PCnext <= PC4 + Extend_Imme * 4; //若为分支，为 PC4 + 立即数 <<
2
        2'b10: PCnext <= {PC4[31:27],instruction[25:0], 2'b00}; //若跳转，拼接相应位
        default: PCnext <= 32'h00000000;
      endcase
    end
  endmodule
```

choose32：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

choose5：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

仿真结果

根据以上模块分析，编写相应代码并测试每个模块。

当所有模块测试完毕，编写顶层模块并仿真。

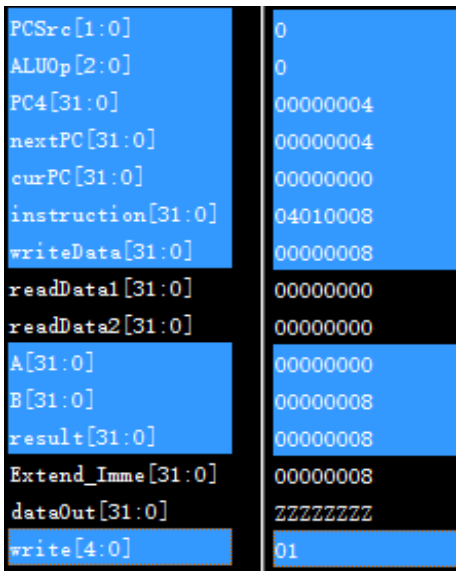
测试程序段及每条指令仿真结果如下：

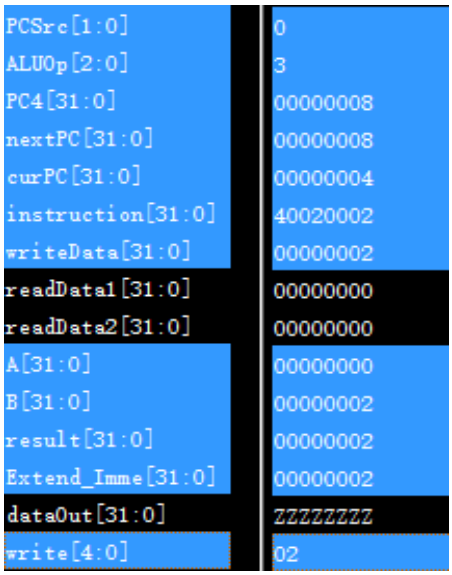
地址	汇编程序	指令代码					
		op（6）	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码	
0x00000000	addi $1,$0,8	000001	00000	00001	0000 0000 0000 1000	=	04010008
0x00000004	ori $2,$0,2	010000	00000	00010	0000 0000 0000 0010	=	40020002
0x00000008	add $3,$2,$1	000000	00010	00001	00011 00000 000000	=	004118000
0x0000000C	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

(curPC 为当前指令地址,PC4 为当前指令地址 + 4, nextPC 是下一指令地址)

addi $1,$0,8：操作数(A = $0 = 0, B = Extend_Imme = 8)，结果寄存器(write = 1)，结果(result = 8)

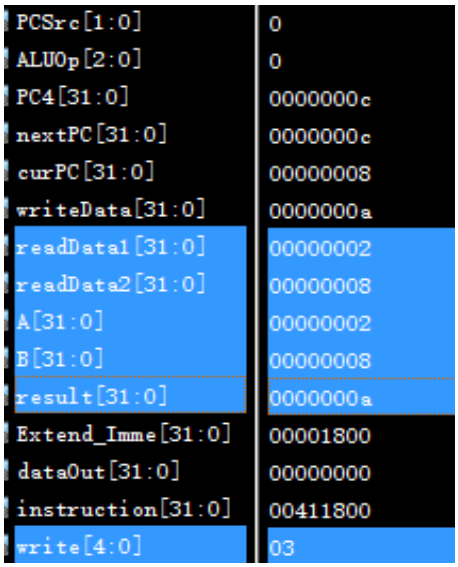
ori $2,$0,2：操作数(A = $0 = 0, B = Extend_Imme = 2)，结果寄存器(write = 2)，结果(result = 2)

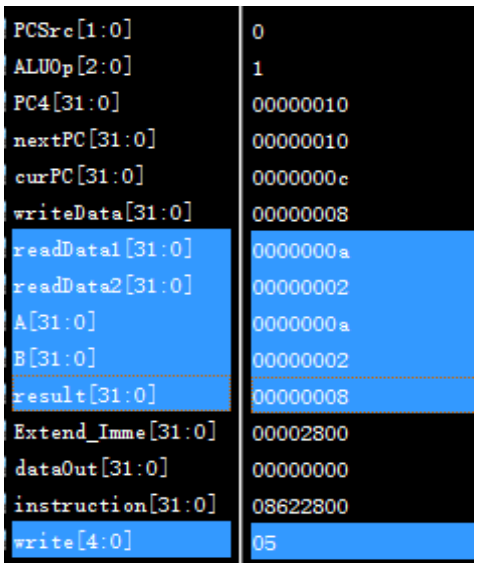




add $3,$2,$1：操作数(A = $2 = 2, B = $1 = 8)，结果寄存器(write = 3)，结果(result = 10)

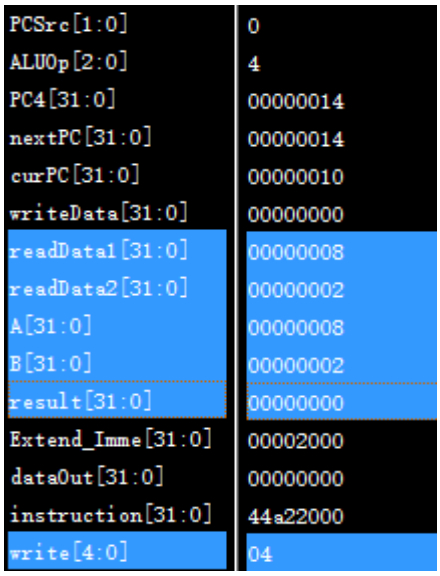
sub $5,$3,$2：操作数(A = $3 = 10 , B = $2 = 2)，结果寄存器(write = 5)，结果(result = 8)

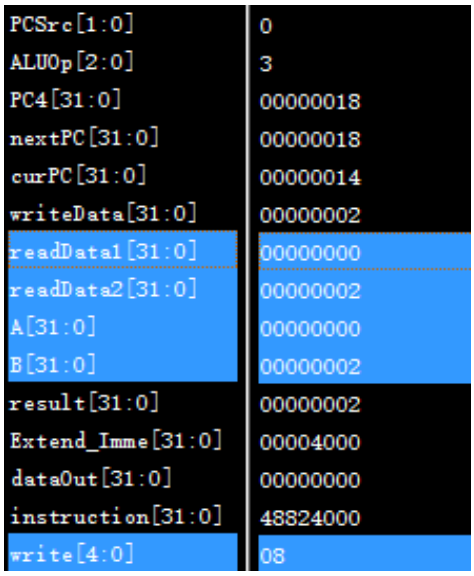




and $4,$5,$2：操作数(A = $5 = 8, B = $2 = 2)，结果寄存器(write = 4)，结果(result = 0)

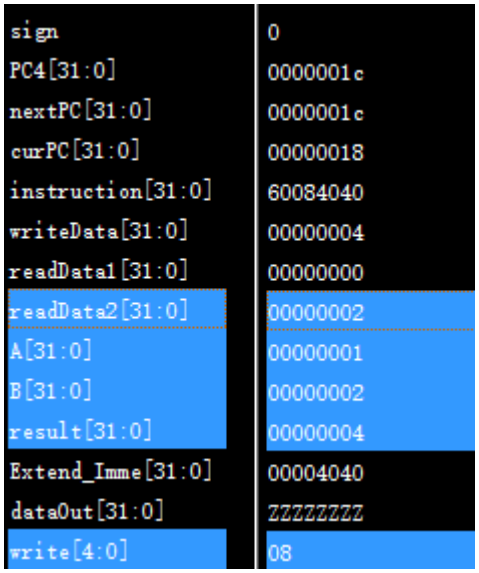
or　$8,$4,$2：操作数(A = $4 = 0 , B = $2 = 2)，结果寄存器(write = 8)，结果(result = 2)

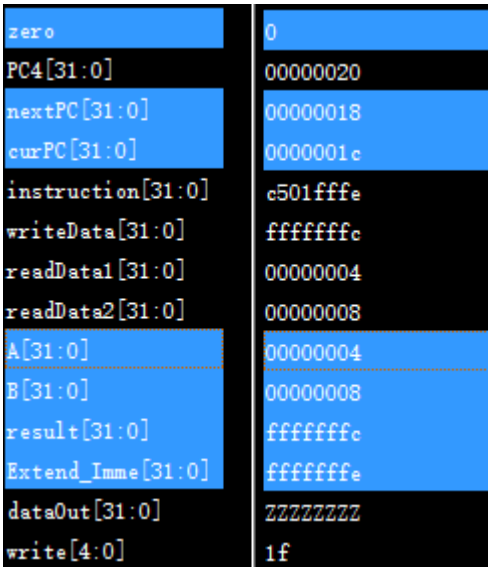




sll　$8,$8,1：操作数(A = $8 = 2, B = sa = 1)，结果寄存器(write = 8)，结果(result = 4)

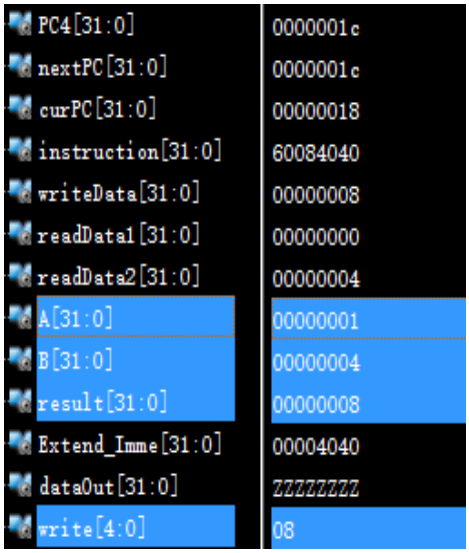
bne $8,$1,-2：操作数(A = $8 = 4 , B = $1 = 8),结果(result = -4), zero = 0，跳转至 PC + 4 + Imme * 4 = 18h，即 nextPC = 18h

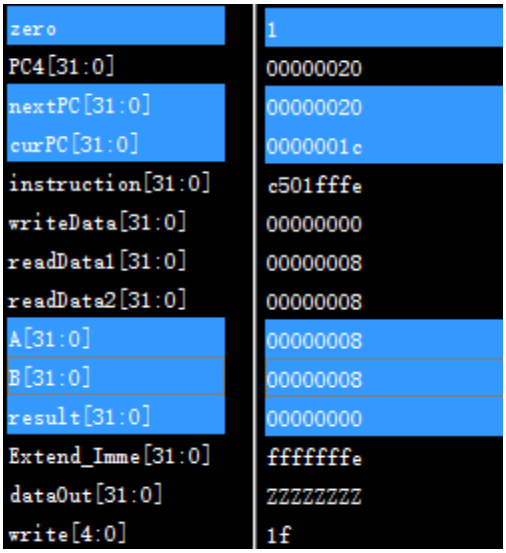




sll $8,$8,1：操作数(A = $8 = 4, B = sa = 1)，结果寄存器(write = 8)，结果(result = 8)

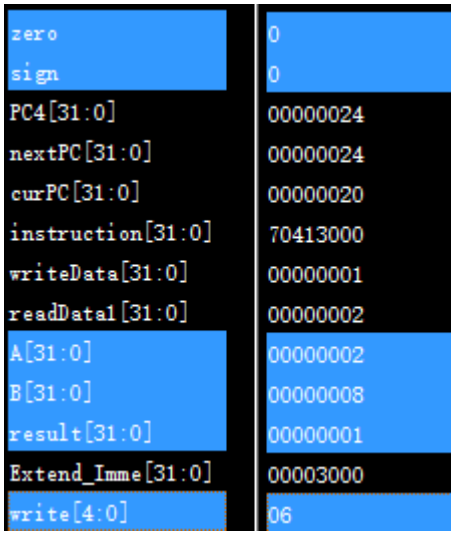
bne $8,$1,-2：操作数(A = $8 = 8 , B = $1 = 8),结果(result = 0), zero = 1，不跳转,即 nextPC = 20h

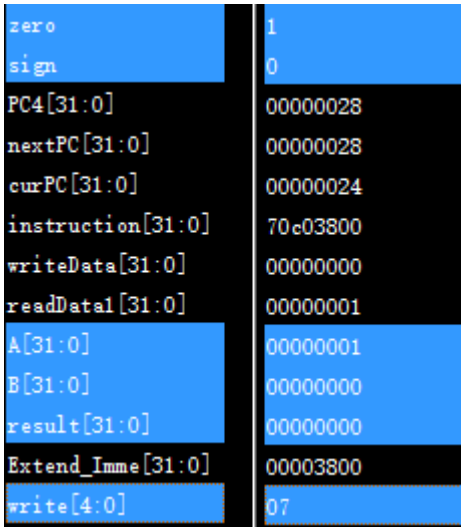




slt $6,$2,$1：操作数(A = $2 = 2, B = $1 = 8)，结果寄存器(write = 6)，结果(2<8,即 result = 1)

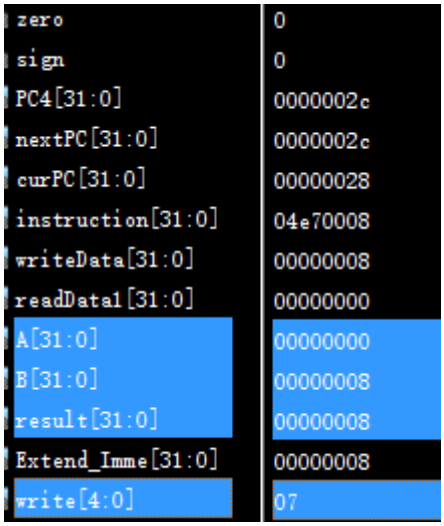
slt $7,$6,$0：操作数(A = $6 = 1, B = $0 = 0)，结果寄存器(write = 7)，结果(1>0,即 result = 0)

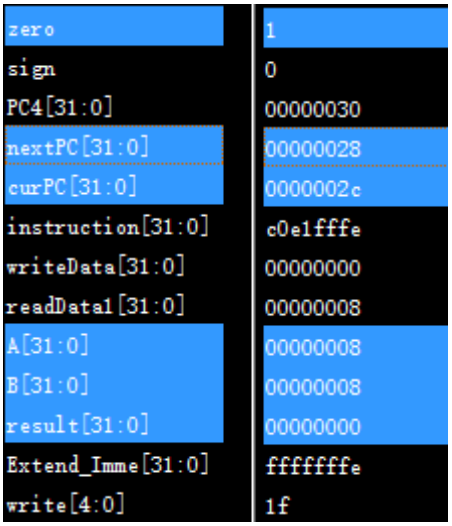




addi $7,$7,8：操作数(A = $7 = 0, B = Imme = 8)，结果寄存器(write = 7)，结果(result = 8)

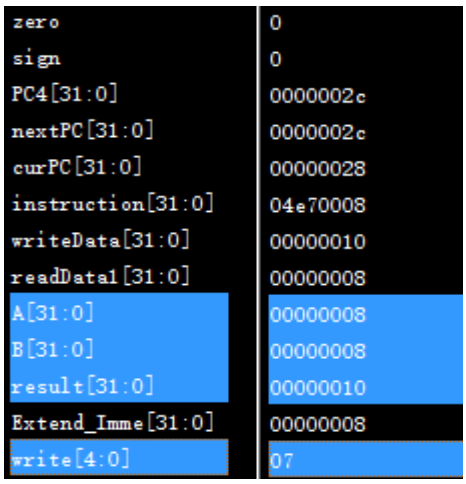
beq $7,$1,-2：操作数(A = $7 = 8, B = $1 = 8),结果(result = 0),zero = 1，跳转至 PC + 4 + Imme * 4 =28h，即 nextPC = 28h

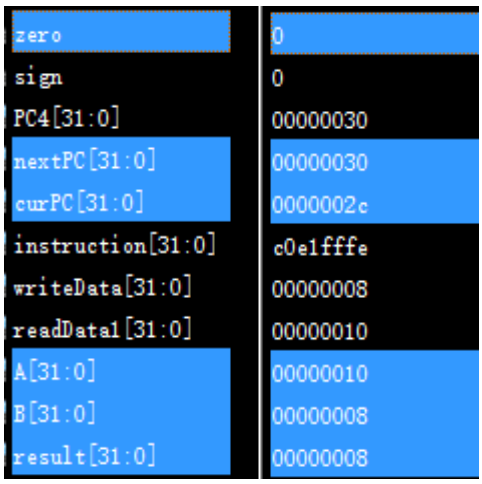




addi $7,$7,8：操作数(A = $7 = 8, B = Imme = 8)，结果寄存器(write = 7)，结果(result = 16)

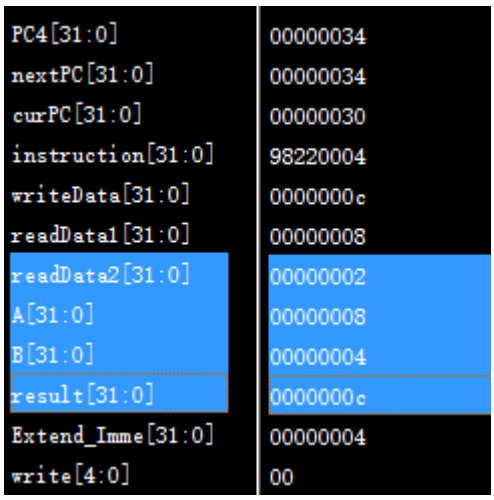
beq $7,$1,-2：操作数(A = $7 = 16, B = $1 = 8),结果(result = 8),zero = 0，不跳转，即 nextPC = 30h

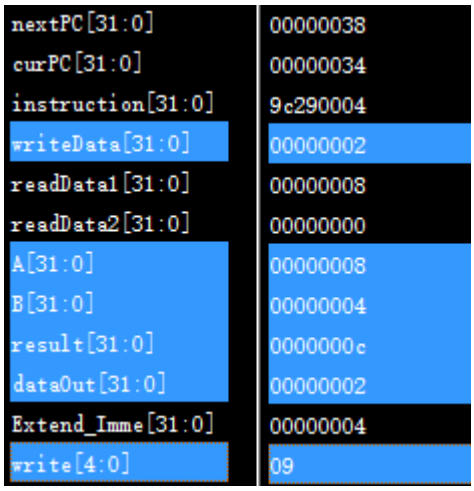




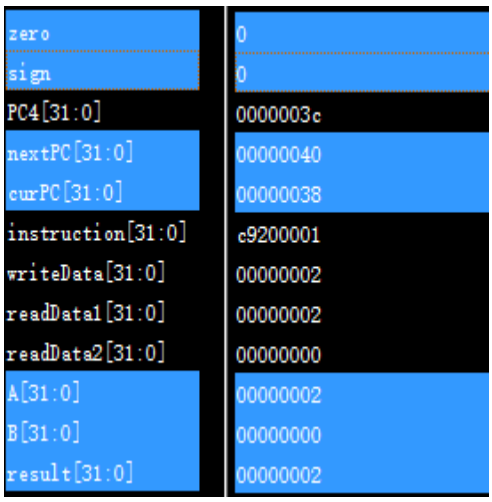
sw $2,4($1)：操作数(A = $1 = 8, B = Imme = 4), $2 = 2 写入数据存储器，地址为 12

lw $9,4($1)：操作数(A = $1 = 8, B = Imme = 4), 数据存储器地址为 12 的值(即 2)写入$9



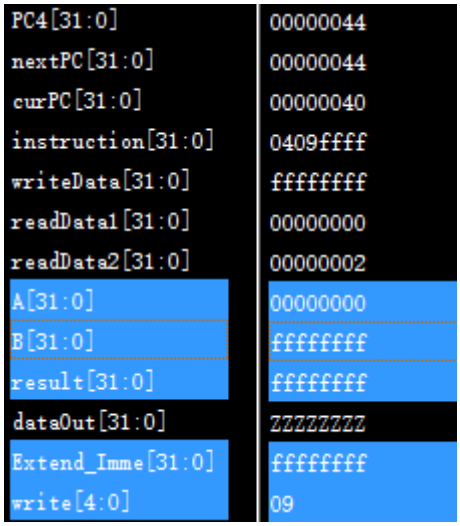


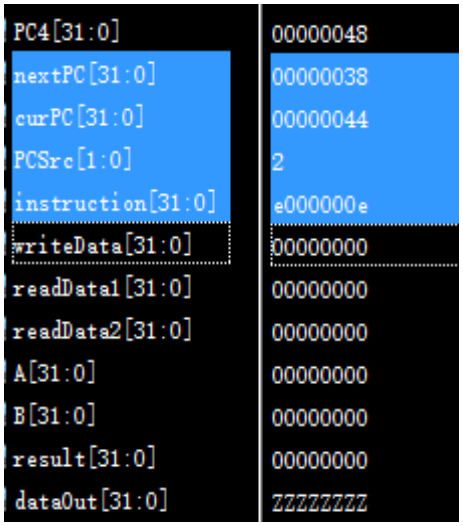
bgtz　$9,1：操作数(A = $9 = 2, B = 0), sign = 0，zero = 0，跳转至 PC + 4 + Imme = 40h，即 nextPC = 40h



addi　$9,$0,-1：操作数(A = $0 = 0, B = Imme = -1)，结果寄存器(write = 9)，结果(result = -1)

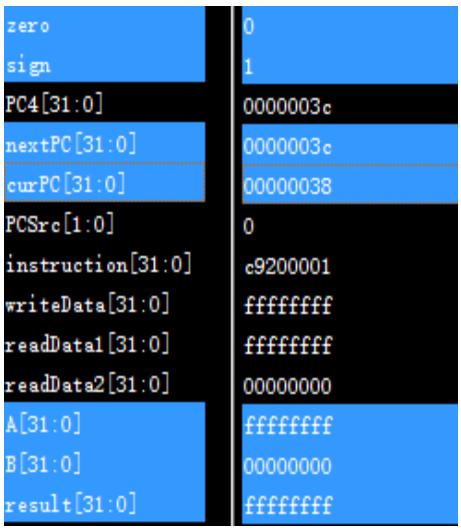
j　0x00000038：nextPC = {PC+4[31:28], instruction[25:0],2'b00} = 38h

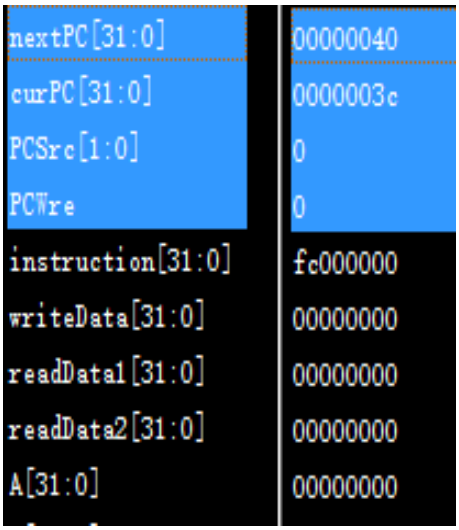




bgtz　$9,1：操作数(A = $9 = -1, B = 0), sign = 1，zero = 0，不跳转，即 nextPC = 3ch

halt：停机，PCWre = 0，即 curPC 不再变化。





综上，所有指令正确执行，仿真结果正确。

六、basys3 烧录 CPU 实验过程

实验分析

（a.改动部分）CPU 实现方法同仿真时相同，但由于 basys3 板子内部的接口有限，不能容纳上述实现。对上述实现在不影响指令功能的情况下，对部分接口及代码实现做以下调整：

1. 指令地址，即 PC 的宽度由 32 位变为 8 位
2. 寄存器宽度由 32 位变为 8 位，相应地，运算数、运算结果等做同样处理
3. 立即数仍为 16 位，但经过扩展单元后变为 8 位，即不进行 0 扩展或符号扩展，而直接取低 8 位
4. 跳转指令的目标地址变为立即数左移 2 位所得结果的低 8 位，即不再与 PC 拼接

（b.补充部分）为在板子上显示指令执行结果，新增模块如下（完整代码在压缩文件中，此处只显示关键代码）：

divide：分频单元，对板子上自带时钟进行分频

choose_show：接收选择信号，输出在数码管上要显示的两个数据

count4：4 进制计数器，根据上述分频时钟实现四进制计数

chooseNUM：根据上述计数器计数结果及显示的数据，选择当前时刻显示的数字（数码管一个时刻只能显示一个数字）

NUMPOS：根据上述计数器计数结果，选择当前时刻显示的数字在数码管上的位置

seg_7：七段码译码器

buttom：按键消抖模块

```
module buttom( input clk, input button, output reg rise);
```

```
    reg[7:0] samp = 0;//移位寄存器采集 button 键值
    always@(posedge clk) begin
        samp<={samp[6:0],button};
        //若有抖动，则产生信号序列为 00000001->0000001(1/0)->.....
    end
    always@(posedge clk) begin
        if(samp==8'b00000001) rise<=1'b1;
        else rise<=1'b0;   //从而对于抖动产生的不稳定电平，只产生一个上升沿
    end
endmodule
```

新增 SW_in 信号：控制数码管显示的数据

为 0 时显示当前指令地址及下一条指令地址；为 1 时显示 RS 寄存器号和相应的数据；为 2 时显示 RT 寄存器号和相应的数据；为 3 时显示 ALU 运算结果及 DBDataSrc 的输出

实验结果

根据上述改动及补充部分的分析，编写相应代码并测试每个模块。

测试完毕仿真及烧录至 basys3 上。

测试代码段同上，仿真结果如下：

(show1、show2 为数码管显示的数字)

addi $1,$0,8：操作数(A = $0 = 0, B = Extend_Imme = 8)，结果寄存器(write = 1)，结果(result = 8)，SW_in 为 0，show1、show2 分别为 curPC = 00h、nextPC = 04h

ori　$2,$0,2：操作数(A = $0 = 0, B = Extend_Imme = 2)，结果寄存器(write = 2)，结果(result = 2)，SW_in 为 0，show1、show2 分别为 RS = 00h、[RS] = 00h

nextPC[7:0]	04
curPC[7:0]	00
writeData[7:0]	08
readData1[7:0]	00
readData2[7:0]	00
A[7:0]	00
B[7:0]	08
result[7:0]	08
Extend_Imme[7:0]	08
dataOut[7:0]	00
instruction[31:0]	04010008
write[4:0]	01
SW_in[1:0]	0
show1[7:0]	00
show2[7:0]	04

nextPC[7:0]	08
curPC[7:0]	04
writeData[7:0]	02
readData1[7:0]	00
readData2[7:0]	00
A[7:0]	00
B[7:0]	02
result[7:0]	02
Extend_Imme[7:0]	02
dataOut[7:0]	00
instruction[31:0]	40020002
write[4:0]	02
SW_in[1:0]	1
show1[7:0]	00
show2[7:0]	00

其余指令执行情况同 32 位 CPU 仿真情况相同，此处不在赘述。

烧录情况如下（以前三条指令为例）：

addi $1,$0,8（左下为 SW_in 开关）

SW_in = 00 时，PC = 00, nextPC = 04

SW_in = 01 时，RS = 00, [RS] = 00

SW_in = 10 时，RT = 01, [RT] = 08(因为运算结果此时已写入 1 号寄存器，故显示 08)

SW_in = 11 时，result = 08，　DBDataSrc = 08


ori　$2,$0,2（左下为 SW_in 开关）

SW_in = 00 时，PC = 04, nextPC = 08

SW_in = 01 时，RS = 00, [RS] = 00

SW_in = 10 时，RT = 02, [RT] = 02(因为运算结果此时已写入 2 号寄存器，故显示 02)

SW_in = 11 时，result = 02，　DBDataSrc = 02



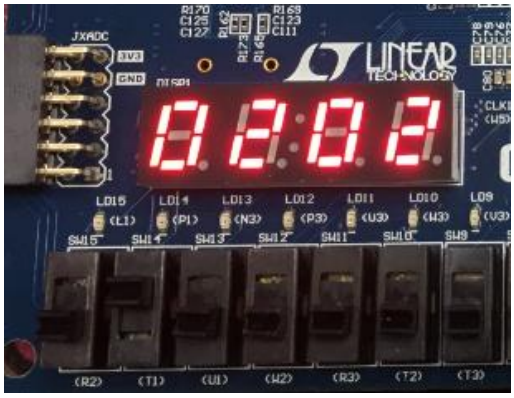

add $3,$2,$1（左下为 SW_in 开关）

SW_in = 00 时，PC = 08, nextPC = 0C

SW_in = 01 时，RS = 02, [RS] = 02

SW_in = 10 时 ，RT = 01, [RT] = 08

SW_in = 11 时 ，result = 0A ， DBDataSrc = 0A

经测试，其余指令均正确执行，此处不再展示。

综上，basys3 烧录 CPU 正确实现。

七、实验心得

从星期二开始，星期六结束。花了五天的时间，完成了这次设计单周期 CPU 的实验。

其实在大一下学期学数电的时候，当时期末的大作业老师建议我们设计一个单周期 CPU，当时就觉得这个应该对之后的学习有蛮大帮助的，所以当时就选择了去实现单周期 CPU。当时因为不是强制性的，老师也没有给任何资料，连单周期 CPU 是什么意思我都不知道。但是还是网上一点一点地查，一点一点地理解，即使当时不知道 MIPS 是什么意思，看不懂 MIPS 指令，看那个数据通路图一脸茫然，到最后也是基本理

解了，然后模仿着别人的代码实现了。对这个学期计组的学习的帮助是巨大的——理论课上讲 CPU 部分时，在之前的基础上，我的理解越来越清晰；而对这次实验同样，上一次设计 CPU 让我对 CPU 的模块结构和对 vivado 的使用有了一定基础，加上老师所给的文件中提到的注意点，调试耗时比上次少了许多。至于在数码管上显示不同的信息，大一下数电实验课期末作业是实现一个可调时间的时钟，对数码管的显示同样需要用到分频等这些操作。

所以，这次实验可能遇到的问题会相对少一些，但过程中还是出了一些状况。操作失误和粗心之类的，比如，为了烧录进入 basys3，新建了一个工程并直接把 32 位的文件添加，然后在这个基础上进行修改，后来才发现修改的是原来的代码，而之前又没有保存代码，内心是崩溃的！只能重新改回来。比如，看错指令，把 add 看成 and，只能改回来重新截图波形。等等，小状况还是蛮多的。好在整个过程思路很清晰。

遇到最大的问题应该是烧录到板子上的执行结果与仿真时不相同。在仿真正确的情况下，烧录到板子上面 PC 不会改变。于是改了一句代码，板子上可以了，仿真时 PC 一直是 XX。其次是执行到 halt 指令时，调整 Reset，nextPC 没有归零。想了很久，应该还是 PC 单元的问题。在 Reset 为 0 时，将 PC 与 nextPC 均置为 0，做了这样的调整之后，仿真时 PC 与 nextPC 均初始化为 0 而不是 XX，这样在解决上述问题的同时，也不会出现在 Reset = 0 时先执行了第一条指令的情况。

总的来说，这次实验目标很清晰，对于理论的学习和实验的学习都有很大的锻炼，一方面巩固了对知识点的理解，另一方面真的做起来还是有很多大大小小的问题，而在查资料解决这些问题的时候，学习到的东西还有这个过程，我觉得蛮重要的。最后，看着正确波形图和板子上按照预期的指令正确执行，成就感太大啦！！当然，肯定还是会有很多不足的，希望下次可以做得更好。