中山大學
SUN YAT-SEN UNIVERSITY

《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
专业（班级）：	16 软件工程三（6）班
学 生 姓 名 ：	孙肖冉
学　　号 ：	16340198
时　　间 ：	2018　年　5　月　31　日

成 绩 ：

实 验 二 ：单周期 CPU 设计与实现

（完成时间：第 11、12、13 周）

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）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	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。

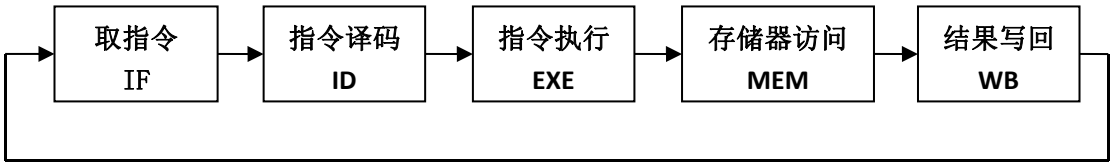
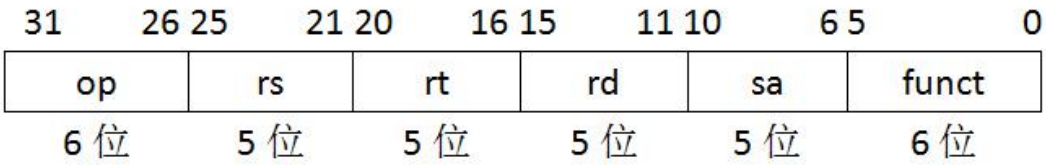


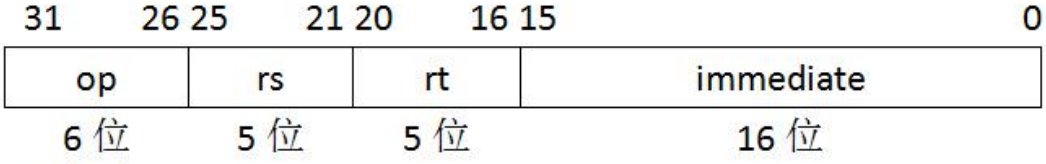
图 1　单周期 CPU 指令处理过程

MIPS 指令的三种格式：

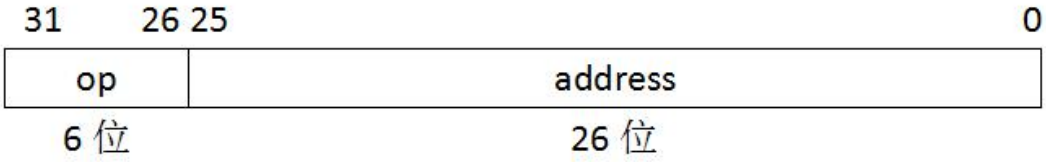
R 类型：



I 类型：



J 类型：



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。

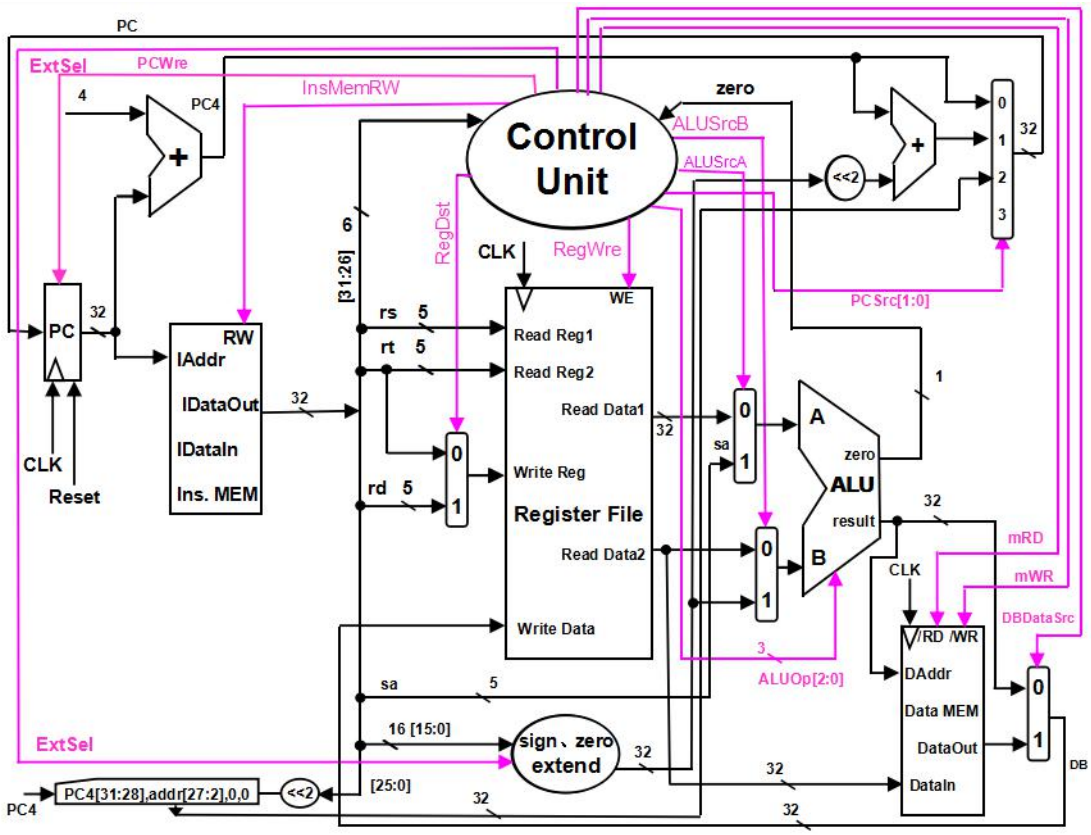


图 2　单周期 CPU 数据通路和控制线路图

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<<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 与 B 不带符号
110	Y=(((rega<regb) && (rega[31] == regb[31])) ||((rega[31] ==1 && regb[31] == 0))) ? 1:0	比较 A 与 B 带符号
111	Y = A ⊕ 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。

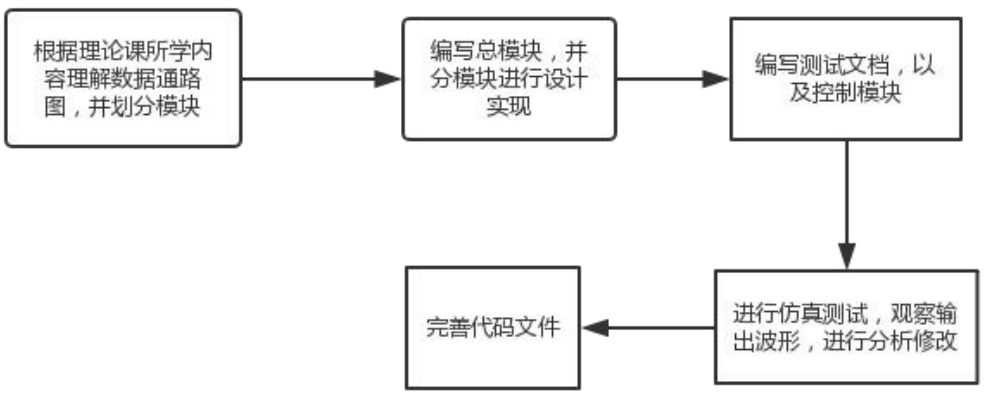
四、实验设备

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

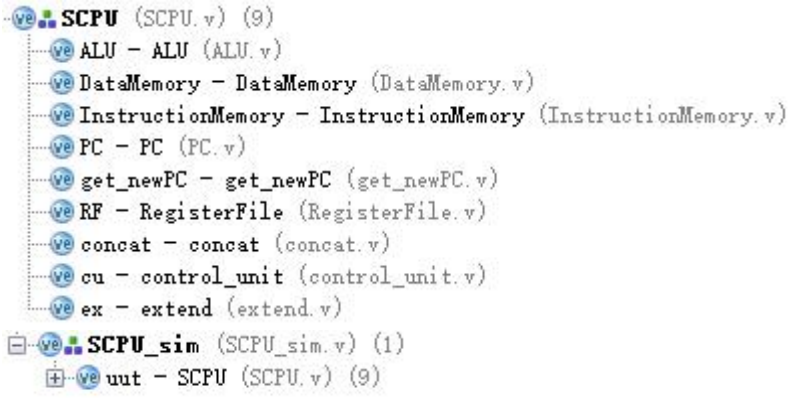
【设计思路】

本次实验是单周期 CPU 的设计，根据如下流程图进行设计：



其中最重要的是模块的设计及实现，首先要理解各个模块的功能以及它们所对应的输入输出信号，并且依据信号表与所给功能表完成具体内容的设计实现。
测试文件的编写：在一开始并未清楚是如何测试其正确性，后来查阅了一些资料，了解到要先写出 MIPS 指令，并且将其译成机器码，并且将其存储在.txt 或者.coe 文件中,如此进行测试。

（1）文件层次说明：



2）分模块说明（完整代码附在文件 src 中）

SCPU 总模块

//项目的主模块，输入为工作时钟信号(clk)，恢复信号(reset)；输出为各种运算结果
在这个模块中将所有的模块进行了实例化并将其连接构成一个单周期 CPU 的整体，并且定义了所需用到的全部接口。

实例化

```
ALU ALU(ReadData1, ReadData2, sa, extended, ALUSrcA, ALUSrcB, opcode, zero, sign, result);
DataMemory DataMemory(clk, result, ReadData2, RD, WR, DBDataSrc, WriteData);
InstructionMemory InstructionMemory(IAddr, InsMemRW, control_code, rs, rt, rd, sa, extend, jumpAddr);
PC PC(clk, reset, PCWre, newPC, IAddr);
get_newPC get_newPC(IAddr, concatAddr, extended, PCSrc, newPC);
RegisterFile RF(clk, RegDst, RegWre, rs, rt, rd, WriteData, WR, RD, ReadData1, ReadData2);
concat concat(IAddr, jumpAddr, concatAddr);
control_unit cu(control_code, zero, sign, InsMemRW, ExtSel, PCWre, PCSrc, RegDst, RegWre, opcode, ALUSrcA, ALUSrcB, WR, RD, DBDataSrc);
extend ex(extend, ExtSel, extended);
```

ALU 运算模块

//根据输入的 ALUopcode,ALUsrcA,ALUsrcBc 并且参照所给的功能表，选择需要进行运算的数据，进行运算，输出结果等信息。在此模块中要注意 zero 的赋值改变。
//参照功能表实现各个功能

```
33  assign operandA =(ALUSrcA == 0)?  ReadData1 :{ 27'b000000000000000000000000000, sa };
34  assign operandB = (ALUSrcB == 0)? ReadData2 : extended;
35  assign zero = (result==0)? 1: 0;
36  assign sign = (result[31]==0)? 0: 1;
37  always @(opcode or operandA or operandB)
38  begin
39    case(opcode)
40    3'b000: result = operandA + operandB;
41    3'b001: result = operandA - operandB;
42    3'b010: result = operandB << (operandA);
43    3'b011: result = operandA | operandB;
44    3'b100: result = operandA & operandB;
45    3'b101: result = (operandA<operandB)?1:0;
46    3'b110: begin
47          if (operandA < operandB && (operandA[31] == operandB[31] ))
48            result = 1;
49          else if ( operandA[31] && !operandB[31])
50            result = 1;
51          else
52            result = 0;
53        end
54  3'b111: result = operandA & ~operandB | ~operandA & operandB;
```

DataMemory 模块

//进行数据存储器的读写，根据输入的地址和控制指令向存储器写入数据，或者从存储器中读取数据。

```
always @(negedge clk)begin
  if(WR==0)begin
      dataMemory[result + 3] <= ReadData2[7:0];
          dataMemory[result + 2] <= ReadData2[15:8];
          dataMemory[result + 1] <= ReadData2[23:16];
          dataMemory[result] <= ReadData2[31:24];
        end
end

always @(RD or result or DBDataSrc)begin
  if(DBDataSrc == 0)
     WriteData = result;
  else if(RD == 0 )begin
            WriteData[7:0] <= dataMemory[result + 3];
            WriteData[15:8] <= dataMemory[result + 2];
            WriteData[23:16] <= dataMemory[result + 1];
            WriteData[31:24] <= dataMemory[result];
          end
```

InstructionMemory 模块

//指令存储器的读写以及传送各种相关指令

除去 halt 指令，所有指令情况下 InstructionMemory 皆为 1,会根据输入的地址数据输出对应的指令。这个模块首先初始化了 ROM，并且利用$readmemb 语句，读取指定的测试文件。

```
initial begin
   $readmemb("C:/Users/Think/Desktop/test.txt",InsMemory);
end

always @(IAddr or InsMemRW)begin
  if(InsMemRW == 1)begin
        control_code <= InsMemory[IAddr][7:2];
        rs<={InsMemory[IAddr][1:0],InsMemory[IAddr+1][7:5]};
        rt<=InsMemory[IAddr+1][4:0];
        rd<=InsMemory[IAddr+2][7:3];
        sa<={InsMemory[IAddr+2][2:0],InsMemory[IAddr+3][7:6]};
        extend<={InsMemory[IAddr+2][7:0],InsMemory[IAddr+3][7:0]};
        jumpAddr<={ {{InsMemory[IAddr][1:0],InsMemory[IAddr+1][7:0]},InsMemory[IAddr+2][7:0]},InsMemory[IAddr+3][7:0]};
      end
  end
```

PC 模块

//更新 PC 值。具体功能是选择下一位的输出。初始化的输出为 0，输出根据输入的 Reset 和 PCWre.

分为几种情况：1.当 Reset = 0 时，输出为 0；2.当 Reset = 1,PCWre = 1 时，正常产生下一位输出；3.当 Reset = 1 ，PCWre = 0 时，下一个输出则由前一个输出决定

```
module PC(clk,reset,PCWre,newPC,IAddr);
  input clk,reset;
  input PCWre;
  input [31:0]newPC;
  output reg[31:0]IAddr;

always@(posedge clk)begin
  if(reset == 0)
    IAddr=0;
  else begin
   if(PCWre==1)
    IAddr=newPC;
    else
    IAddr=IAddr;
  end
end
```

get_newPC 模块

//根据控制信号进行新 PC 的选择，就是 PC+4+立即数，以及 PC+4d 的选择

```
module get_newPC(
  input [31:0]Addr, concatAddr, extended,
  input [1:0]PCSrc,
  output reg[31:0]newPC
    );
always@(PCSrc or concatAddr)begin
      if(PCSrc == 2'b00)
         newPC=Addr[31:0]+4;
      if(PCSrc == 2'b10)
         newPC=Addr[31:0]+4+4*extended;;
      if(PCSrc == 2'b01)
         newPC=concatAddr;
   end
```

RegisterFile 模块

//进行寄存器的读写。

寄存器与数据的对应：rs--ReadData1 , rt--ReadData2；

在 RegWre = 1 时，则可以想寄存器写入数据；有一点需要注意不能对 0 号寄存器进行修改

```
always@(rs or rt or WR or RD)begin
   ReadData1 = ((rs==0) ? 0: regFile[rs]);
    ReadData2 = ((rt == 0) ? 0: regFile[rt]);
end
always @(negedge clk)begin
      if ( RegWre == 1 && (rd != 0 || rt != 0))
      begin
          if(RegDst == 1)
          begin
              regFile[rd] = WriteData;
          end
          else
              regFile[rt] = WriteData;
      end
```

concat 模块

//实现跳转指令的 pc 拼接

```
module concat(IAddr,jumpAddr,concatAddr);
  input [31:0]IAddr;
  input [25:0]jumpAddr;
  output reg [31:0]concatAddr;

  always @(IAddr or jumpAddr)begin
    concatAddr ={IAddr[31:28],{jumpAddr[25:0],2'b00}};
  end
endmodule
```

Control_unit 模块

//利用指令的前六位的控制信息发出各种的控制信号，要结合控制信号表和 ALU 的功能表进行实现

```
48      always@(opcode or zero )begin
49    //halt->0
50    PCWre =(opcode == 6'b111111)? 0 :1;
51    //sll->1
52    ALUSrcA = (opcode == 6'b011000)? 1 :0;
53    //addi ori slti sw lw->1
54    ALUSrcB = (opcode == 6'b000001 || opcode == 6'b010000 || opcode == 6'b011011 ||
55              opcode == 6'b100110 || opcode == 6'b100111)?1:0;
56    //lw->1
57    DBDataSrc = (opcode == 6'b100111) ? 1 :0;
58    //beq、bne、sw、halt、j -> 0;
59    RegWre= (opcode == 6'b110000 || opcode == 6'b110001 ||
60             opcode == 6'b100110 || opcode == 6'b111111 || opcode == 6'b111000)? 0:1;
61
62    InsMemRW= 1;
63    RD= (opcode == 6'b100111)? 0: 1;
64    WR= (opcode == 6'b100110)? 0: 1;
65    //add、sub、and、or、sll -> 1
66    RegDst = (opcode ==6'b000000 || opcode == 6'b000010 ||opcode == 6'b010001 || opcode ==6'b010010 ||
67              opcode == 6'b011000)? 1:0;
68    //ori->0
69    ExtSel = (opcode == 6'b010000) ?0 :1;
70
71    PCSrc[0] =(opcode == 6'b111000)?1 :0;
72  //beq(zero=1)、bne(zero=0
73    PCSrc[1] = (opcode == 6'b110000 && zero==1 || opcode == 6'b110001 && zero==0 ) ?1:0;
74
75  //sub bne beq ori or -> 1
76    ALUOp[0] = ( opcode == 6'b000010 || opcode == 6'b110001 || opcode == 6'b110000 || opcode == 6'b010000 || opcode == 6'b010010)? 1 :0;
77  //sll ori or slti -> 1
78    ALUOp[1] = (opcode == 6'b011000 || opcode == 6'b010000 || opcode == 6'b010010 || opcode == 6'b011011)? 1 :0;
79
```

extend 模块

//实现立即数的零以及符号的拓展，

```
module extend(extend,ExtSel,extended);
  input [15:0]extend;
  input ExtSel;
  output reg [31:0]extended;
  always @(extend or ExtSel)begin
      if(ExtSel==0 || extend[15] == 0)   //zero
        extended = {16'b0000000000000000,extend};
      else
        extended = {16'b1111111111111111,extend};
  end
endmodule
```

3）测试数据及结果分析：

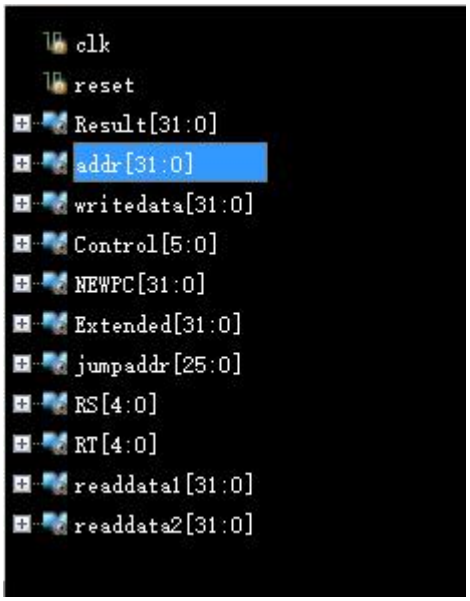
A) 测试程序段：

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	0001 1000 0000 0000	=	00411800
0x0000000C	sub $5,$3,$2	000010	00011	00010	0010 1000 0000 0000	=	08622800
0x00000010	and $4,$5,$2	010001	00101	00010	0010 0000 0000 0000	=	44A22000
0x00000014	or $8,$4,$2	010010	00100	00010	0100 0000 0000 0000	=	48824000
0x00000018	sll $8,$8,1	011000	00000	01000	0100 0000 0100 0000	=	60084040
0x0000001C	bne $8,$1,-2 (≠,转 18)	110001	00001	01000	1111 1111 1111 1110	=	C428FFFE
0x00000020	slti $6,$2,8	011011	00010	00110	0000 0000 0000 1000	=	6C460008
0x00000024	slti $7,$6,0	011011	00110	00111	0000 0000 0000 0000	=	6CC70000
0x00000028	addi $7,$7,8	000001	00111	00111	0000 0000 0000 1000	=	04E70008
0x0000002C	beq $7,$1,-2 (=,转 28)	110000	00001	00111	1111 1111 1111 1110	=	C027FFFE
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	**j 0x00000040**	111000	00000	00000	0000 0000 0001 0000	=	E0000010
0x0000003C	addi $10,$0,10	000001	00000	01010	0000 0000 0000 1010	=	040A000A
0x00000040	**halt**	111111	00000	00000	0000000000000000	=	FC000000
0x00000044							
0x00000048							
0x0000004C							

B）测试文档：test.txt

文档内容：

C）输出及其对应内容



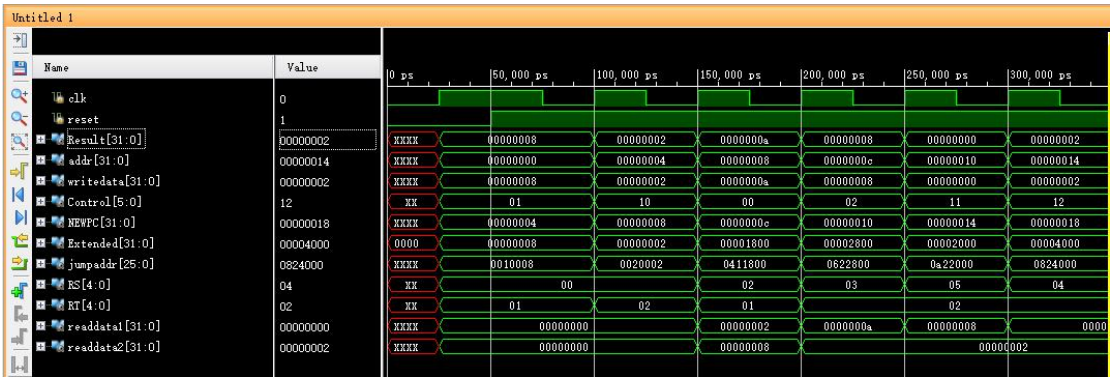
由上至下：
工作时钟，恢复信号，ALU 计算结果，当前地址，数据存储器后的选择结果，控制信号
下一条指令的地址，立即数 0 和符号的拓展结果，跳转指令有关立即数的部分，rs 对应的值

rt 对应的值，寄存器数，寄存器数

d)仿真结果及其分析：

Step1：算数运算指令和逻辑运算指令：

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	0001 1000 0000 0000	=	00411800
0x0000000C	sub $5,$3,$2	000010	00011	00010	0010 1000 0000 0000	=	08622800
0x00000010	and $4,$5,$2	010001	00101	00010	0010 0000 0000 0000	=	44A22000
0x00000014	or $8,$4,$2	010010	00100	00010	0100 0000 0000 0000	=	48824000

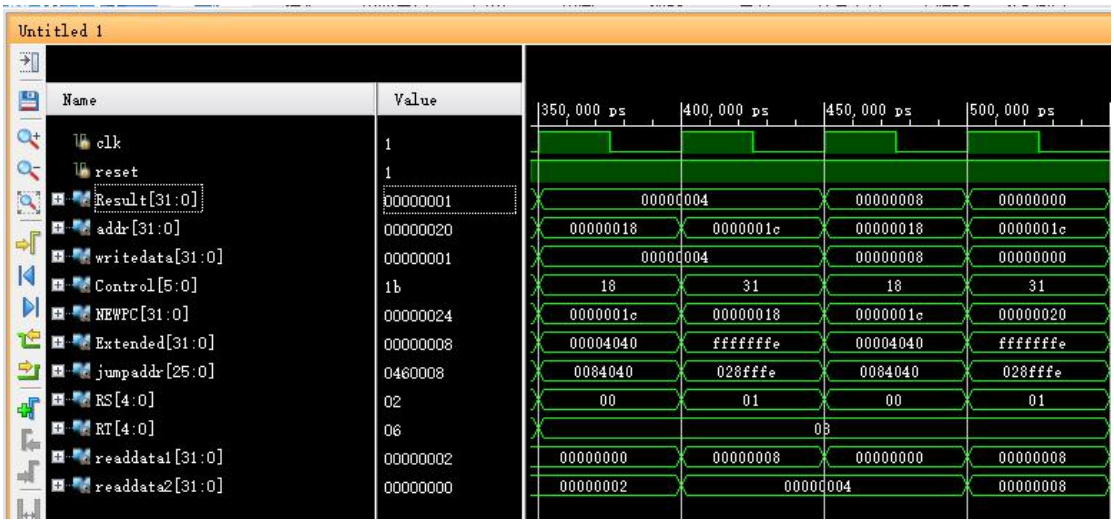


Result[31:0] 00000008 -> 00000002 -> 0000000a -> 00000008 -> 00000000 -> 00000002（十六进制）

实际过程： 0+8 => 8 0|2 => 2 8+2 => aH aH-2 => 8 8&2 => 0 0|2=>2

Step2:左移指令和跳转指令检测：

0x00000018	sll $8,$8,1	011000	00000	01000	0100 0000 0100 0000	=	60084040
0x0000001C	bne $8,$1,-2 (≠,转 18)	110001	00001	01000	1111 1111 1111 1110	=	C428FFFE

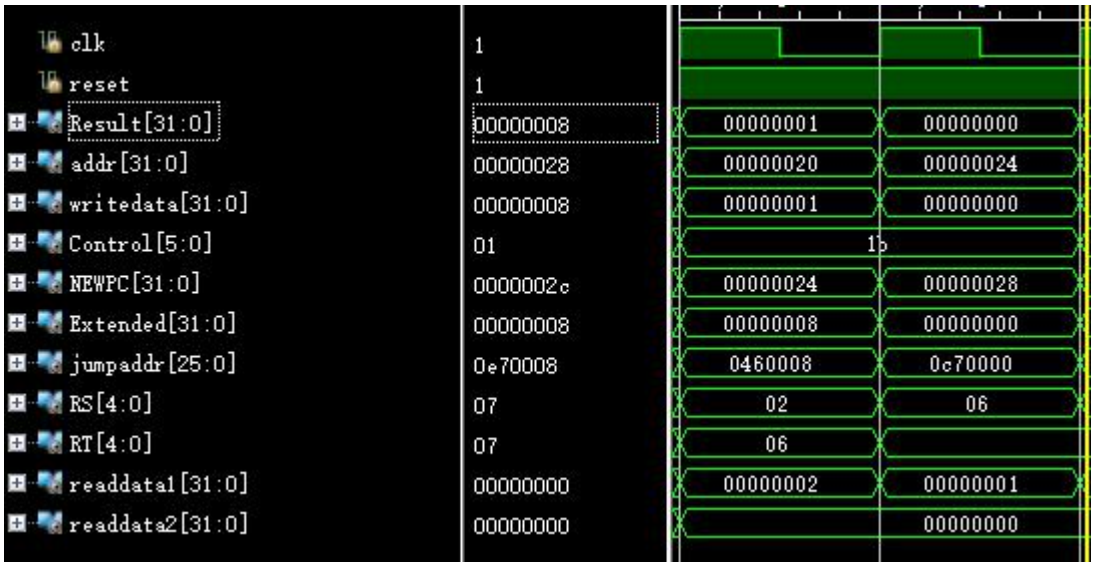


Result[31:0]　00000004 -> 00000004 -> 00000008 -> 00000000

实际过程：　2<<1 = 4　4!=8(回 18)　4<<1 = 8　8 = 8(跳转到 20H)

Step3：slti 指令检测

0x00000020	slti　$6,$2,8	011011	00010	00110	0000 0000 0000 1000	=	6C460008
0x00000024	slti　$7,$6,0	011011	00110	00111	0000 0000 0000 0000	=	6CC70000

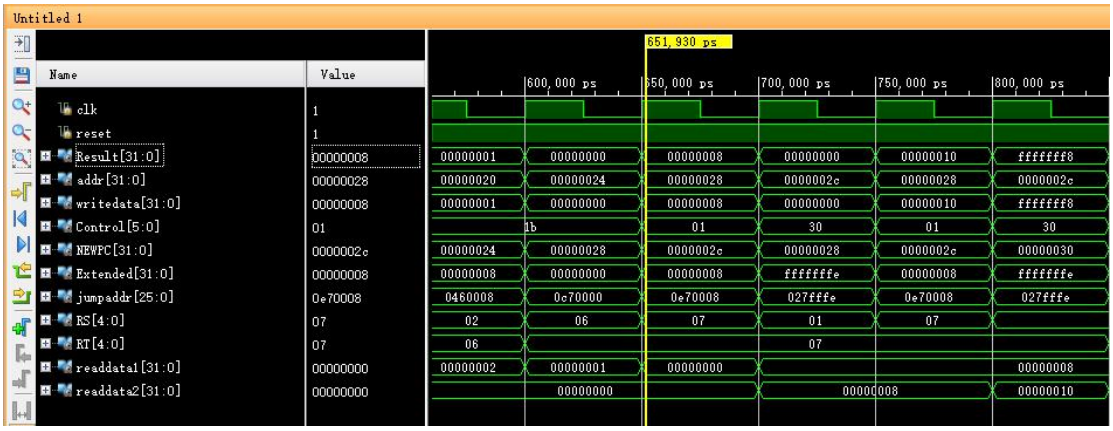


Result[31:0]　00000001 -> 00000000

实际过程　2<8 set1　1<0 set0

Step4：　aadi 指令，beq 指令检测

0x00000028	addi $7,$7,8	000001	00111	00111	0000 0000 0000 1000	=	04E70008
0x0000002C	beq $7,$1,-2 (=,转 28)	110000	00001	00111	1111 1111 1111 1110	=	C027FFFE

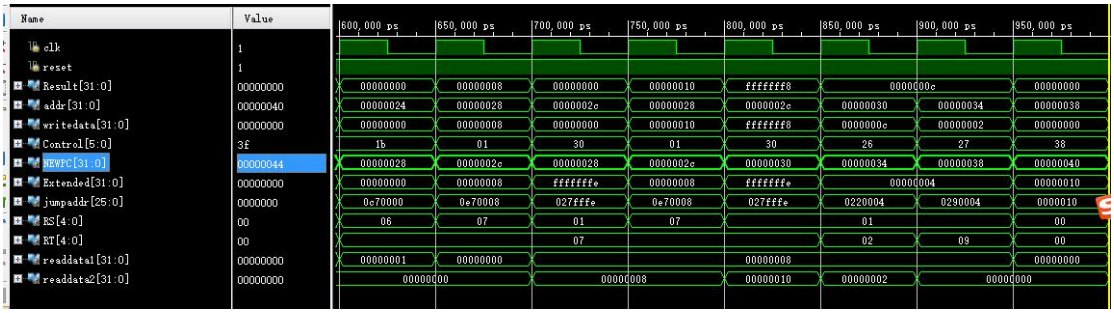


Result[31:0] 00000008 -> 00000000 -> 00000010 -> ffffffff8

实际过程 8+0=8 8=8（转 28H） 8+8=16 16！=8（转下一条）

Step5：sw ， lw ， j ,halt 指令检测

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	**j 0x00000040**	111000	00000	00000	0000 0000 0001 0000	=	E0000010
0x0000003C	addi $10,$0,10	000001	00000	01010	0000 0000 0000 1010	=	040A000A
0x00000040	**halt**	111111	00000	00000	0000000000000000	=	FC000000



Addr[31:0] 00000030 -> 00000034 -> 00000038 -> 00000040

实际过程 将寄存器$2 中的值存入 将$1 偏移 4 个长度的值到$9 中 无条件跳转 停机

各个寄存器的数值变化：

地址	汇 编 程序	寄存器数值									
		$0	$1	$2	$3	$4	$5	$6	$7	$8	$9

0x00000000	addi $1,$0,8	0	8								
0x00000004	ori $2,$0,2	0	8	2							
0x00000008	add $3,$2,$1	0	8	2	10						
0x0000000C	sub $5,$3,$2	0	8	2	10		8				
0x00000010	and $4,$5,$2	0	8	2	10	0	8				
0x00000014	or $8,$4,$2	0	8	2	10	0	8			2	
0x00000018	sll $8,$8,1	0	8	2	10	0	8			4	
0x0000001C	bne $8,$1,-2 (≠, 转 18)	0	8	2	10	0	8			4	
0x00000020	slti $6,$2,8	0	8	2	10	0	8	1		4	
0x00000024	slti $7,$6,0	0	8	2	10	0	8	1	0	4	
0x00000028	addi $7,$7,8	0	8	2	10	0	8	1	8	4	
0x0000002C	beq $7,$1,-2 (=, 转 28)	0	8	2	10	0	8	1	8	4	
0x00000030	sw $2,4($1)	0	8	2	10	0	8	1	8	4	4
0x00000034	lw $9,4($1)	0	8	2	10	0	8	1	8	4	4

0x00000038	**j 0x00000040**	0	8	2	10	0	8	1	8	4	4
0x0000003C	addi $10,$0,10	0	8	2	10	0	8	1	8	4	4
0x00000040	**halt**	0	8	2	10	0	8	1	8	4	4

六．实验心得

这次实验的前期准备花费了很多时间，因为理论课知识掌握的不牢固，所以一开始在理解如何划分模块，以及模块之间的联系时遇到了很多困难，感谢其他同学的帮助。

大致叙述一下遇到的一些问题和解决的方案：

（1）verilog 语言的学习，因为在大一时接触过该语言，所以以为掌握的已经可以了，没想到在做本次实验时，才发现自己的理解存在很多偏差：例如 wire 型与 reg 型的区别;wire 箱单与物理的连接，而 reg 则是存储单元，以及 always 块内语言的使用特点，对 reg 型赋值时可以不使用 assign 标示。

（2）测试文件的书写，一开始我直接将二进制的指令无格式划分的写入文件中，但是无法进行仿真，查阅资料发现需要按照 8 位长度存放。

（3）关于测试改错，一开始并没有注意到错误信息的提示，所以有很多语法的错误，例如少写了 end 的情况，在最后进行测试时发现了许多问题，然后才进行改错，花费了超多时间，应该分模块进行检测，注意错误信息的提示

（4）Mips 指令的理解，一开始对于各个指令的理解并没有参照老师所给的而是按照自己的想法，产生了很多偏差。例如将 beq 和 bne 指令弄混淆，错误的进行了跳转判断。

（5）各种命名的错误，因为对整体思路不明晰，所以在运用各个寄存器，线路，以及控制信号时会写错大小写或者是命名直接写错。最后我的解决方式是将所有模块所要对应的输入输出的名称在纸上列出，对照其进行编写，这样降低了我的拼写错误率，也使我更加清楚的明白各个模块之间的数据传输。

从整体来看本次实验涉及到了很多知识点：mips 的指令的运用，单周期 CPU 的模块划分以及 verilog 语言的使用。按照老师所给的控制信号表和功能表实现单周期 CPU 的功能，使得我对其工作原理有了进一步的理解，十分感谢老师的讲解以及给予的资料。