一.实验任务

全部采用Verilog 硬件描述语言设计实现简单指令集MIPS 微处理器,要求指令存储器在时钟上升沿读出指令，指令指针的修改、寄存器文件写入、数据存储器数据写入都在时钟下降沿完成。完成完整设计代码输入、各模块完整功能仿真，整体仿真，验证所有指令执行情况。

且假定所有通用寄存器复位时取值都为各自寄存器编号乘以4；PC寄存器初始值为0；数据存储器和指令存储器容量大小为32*32，且地址都从0开始，指令存储器初始化时装载测试MIPS汇编程序的机器指令，数据存储器所有存储单元的初始值为其对应地址的取值。需要注意的是数据存储器的地址呈现以下规则：都是4的整数倍。

仿真以下MIPS汇编语言程序段的执行流程：

```
main:   add $4,$2,$3
        lw $4,4($2)
        sw $5,8($2)
        sub $2,$4,$3
        or $2,$4,$3
        and $2,$4,$3
        slt $2,$4,$3
        beq $3,$3,equ
        lw $2,0($3)
equ:    beq $3,$4,exit
        sw $2,0($3)
exit:   j main
```

二.实验目的

1. 掌握简单指令集MIPS 微处理器数据通路图的构成、原理及其设计方法；
2. 掌握简单指令集MIPS 微处理器的实现方法，代码实现方法；
3. 认识和掌握指令与处理器的关系；
4. 掌握测试MIPS 微处理器的方法；
5. 掌握简单指令集MIPS 微处理器的实现方法。

三.实验环境

1. Windows 10操作系统
2. 开发工具：Xilinx Vivado 2018.3
3. 机器码导出工具：Mars 4_5

四.处理器设计方案

1. 简单指令集MIPS 微处理器的运行流程.

为了简化硬件电路的设计，我采用了单周期微处理器的方案。

单周期微处理器的核心思想是：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。其中，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。

在运行程序时，执行每一条指令时的流程图如下：

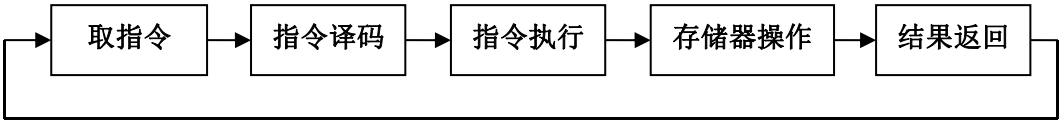


图1：单周期MIPS微处理器指令执行流程

其中，各步骤的具体内容如下：

(1) 取指令：

根据不同的操作又可分为以下两种情况：

①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。

②当遇到“地址跳转”指令时，则控制器把“转移地址”送入PC，以实现指令的长距离跳转。

(2) 指令译码：

对所取得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：

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) 存储器操作：

如果该条指令需要从存储器中获得数据或将数据写入存储器，则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) 结果返回：

指令执行的结果或者从存储器中得到的数据写回相应的目的寄存器中。

1. 微处理器各模块设计.

基于微处理的一般逻辑结构与设计思路，规划各模块的框图结构如图2所示。由图可知，该处理器包含指令存储器、数据存储器、寄存器组、ALU单元、符号数扩张、控制器、ALU控制译码以及多路复用器与时钟信号产生器等。由于是单周期处理器，以上各个部件必须在

时钟信号的控制下协调工作。

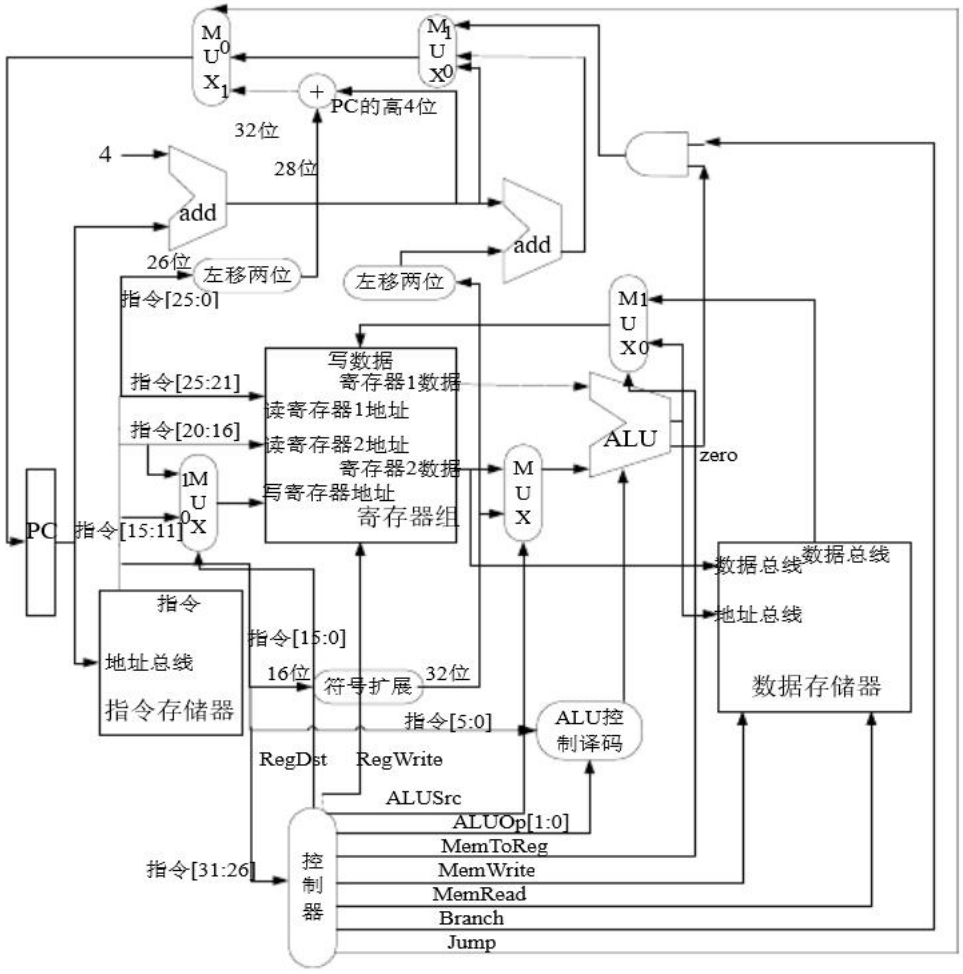


图 2：微处理器各模块设计

其中各部分的功能概况如下：

（1）指令存储器：

指令寄存器为 ROM 类型的存储器，为单一输出指令的存储器。因此其对外的接口为 clk、存储器地址输入信号（指令指针）以及数据输出信号（指令）。

（2）数据存储器：

数据存储器为 RAM 类型的存储器，并且需要独立的读写控制信号。因此其对外的接口输入信号为 clk、we、datain、addr；输出信号为 dataout。 数据存储器基本建立过程同 ROM 的建立。

（3）寄存器：

寄存器组是指令操作的主要对象，MIPS 中一共有 32 个 32 位寄存器。在指令的操作过程中需要区分 Rs、Rt、Rd 的地址和数据，并且 Rd 的数据只有在寄存器写信号有效时才能写入，因此该模块的输入为 clk、RegWriteAddr、RegWriteData、RegWriteEn、RsAddr、RtAddr、reset；输出信号为 RsData、RtData。

（4）ALU：

在这个简单的 MIPS 指令集中，微处理器支持 add、sub、and、or、slt 运算指令，需要利用 ALU 单元实现运算，同时数据存储指令 sw、lw 也需要 ALU 单元计算存储器地址，

条件跳转指令beq需要ALU来比较两个寄存器是否相等。所有这些指令包含的操作为加、减、与、或小于设置5种不同的操作。

（5）ALU控制：

通过2位操作类型码以及6位指令功能码产生ALU单元的4位控制信号。

（6）控制器：

控制器输入为指令的操作码。操作码经过主控制单元的译码，给ALUCtrl、Data Memory、Registers、Muxs等部件输出正确的控制信号。

（7）顶层模块：

顶层模块需要将前面多个模块实例化，通过导线以及多路复用器将各个部件连接起来，并且在时钟的控制下修改PC的值，PC是一个32位的寄存器，每个时钟沿自动增加4。

五.实现过程

1. 微处理器设计思想.

设计微处理器的基本思想是自顶向下，分模块实现。先完成数据通路，把数据通路的每个模块功能所需的输入输出端口统一。数据通路图2所示，接下来就是各个模块的实现。再用顶层模块完成各个模块的连接。

2. 微处理器总体结构.

该微处理器的总体结构如图3所示：

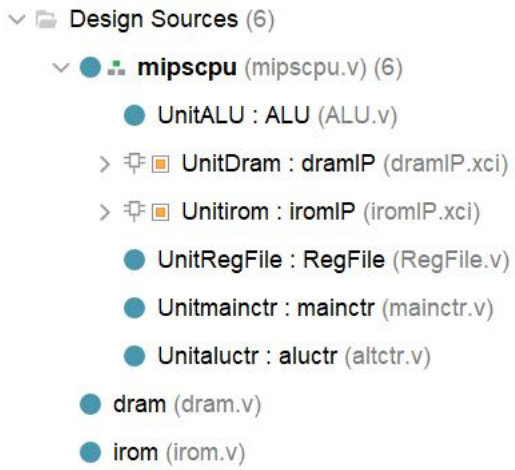


图3：微处理器总体结构

3. 微处理器各模块代码.

（1）RegFile寄存器模块：

RegFile	
Input：	Output：
CLK	RsData
WriteAddr 寄存器写能	RtData
RsAddr 读寄存器号	Memory：

RtAddr 读寄存器号 WriteData 写入的数据	regs[0:31] 31 个寄存器

代码如下：

```
module RegFile(
    input [4:0] RsAddr,
    input [4:0] RtAddr,
    input [4:0] WriteAddr,
    input RegWr,
    input [31:0] WriteData,
    input clk,
    input reset,
    output [31:0] RsData,
    output [31:0] RtData
    );
    reg [31:0]      regs[0:31];
    assign RsData = (RsAddr==5'b0)?32'b0:regs[RsAddr];
    assign RtData = (RtAddr==5'b0)?32'b0:regs[RtAddr];
    integer i;
    always @(posedge clk)
        if(!reset &RegWr)
            regs[WriteAddr]=WriteData;
        else if(reset)
            for(i=0;i<32;i=i+1)
                regs[i]=4*i;
endmodule
```

（2）ROM 指令存储模块：

Irom	
Input： **addr ： PC 当前地址** **Memory：** **regs[0:31] ： 指令存储**	**Output：** **Instr： 指令代码**

代码如下：

```
module irom(
        input [4:0] addr,
        output [31:0] instr
        );
        reg [31:0]    regs[0:31];
        assign instr = regs[addr];
        initial
        $readmemh("C:/Users/lenovo/Desktop/For410/task2.coe",regs,0,10);
endmodule
```

（3）RAM 数据存储模块：

dram	
Input： **MemWR 写使能** **writedata 写入数据** **addr 地址**	**Output：** **readdata 读取数据** **Memory：** **regs[0:31] 数据存储**

代码如下：

```
module dram(
        input [4:0] addr,
        output [31:0] readdata,
        input [31:0] writedata,
        input MemWR
        );
        reg [31:0]        regs[0:31];
        assign readdata =regs[addr];
        always @(addr or writedata or MemWR)
        if(MemWR)
        regs[addr]=writedata;
        integer i;
        initial
        for(i=0;i<32;i=i+1)
        regs[i]=i;
endmodule
```

（4） ALU运算单元

ALU

Input： ALUctr 运算控制 in1 运算数据 in2 运算数据	Output： reg 运算结果 zero 零标志

代码如下：

```
module ALU(
    input signed [31:0] in1,
    input signed [31:0] in2,
    input [3:0] ALUctr,
    output reg [31:0] Res,
    output reg zero
    );
    always@(in1 or in2 or ALUctr)
    begin
        case(ALUctr)
            4'b0110://sub
                begin
                Res=in1-in2;
                zero=(Res==0)?1:0;
                end
            4'b0010://sub
                begin
                Res=in1+in2;
                zero=0;
                end
            4'b0000://sub
                begin
                Res=in1&in2;
                zero=0;
                end
            4'b0001://sub
                begin
                Res=in1|in2;
                zero=0;
                end
            4'b0111://sub
                begin
                Res=(in1<in2)?1:0;
                zero=0;
                end
```

```
                default:
                    begin
                    Res=0;
                    zero=0;
                    end
            endcase
        end
endmodule
```

（5） ALU控制单元

aluctr	
Input： **ALUop　操作码** **Func　功能码**	**Output：** **ALUctr　ALU 控制信号**

代码如下：

```
module aluctr(
    input [1:0] ALUop,
    input [5:0] func,
    output reg [3:0] ALUctr
    );
    always @(ALUop or func)
    casex({ALUop,func})
        8'b00xxxxxx: ALUctr=4'b0010;//lw,sw
        8'b01xxxxxx: ALUctr=4'b0110;//beq
        8'b10xx0000: ALUctr=4'b0010;//add
        8'b10xx0010: ALUctr=4'b0110;//sub
        8'b10xx0100: ALUctr=4'b0000;//and
        8'b00xx0101: ALUctr=4'b0001;//or
        8'b00xx1010: ALUctr=4'b0111;//slt
        default: ALUctr=4'b0000;
        endcase
endmodule
```

（6） 主控制器

mainctr	
Input： **opCode,**	**Output：** **ALUop** **RtDst**

	regwr **Imm** **memwr** **M2R**

代码如下：

```
module mainctr(
    input [5:0] opCode,
    output [1:0] ALUop,
    output RtDst,
    output regwr,
    output Imm,
    output memwr,
    output B,
    output J,
    output M2R
    );
    reg [8:0] outputtemp;
    assign RtDst=outputtemp[8];
    assign Imm=outputtemp[7];
    assign M2R=outputtemp[6];
    assign regwr=outputtemp[5];
    assign memwr=outputtemp[4];
    assign B=outputtemp[3];
    assign J=outputtemp[2];
    assign ALUop=outputtemp[1:0];
    always @ (opCode)
        case (opCode)
            6'b000010:outputtemp = 9'bxxx0_001_xx;//jmp
            6'b000000:outputtemp = 9'b1001_000_10;//R
            6'b100011:outputtemp = 9'b0111_000_00;//lw
            6'b101011:outputtemp = 9'bx1x0_100_00;//sw
            6'b000100:outputtemp = 9'bx0x0_010_01;//beq
            default:outputtemp = 9'b0000000000;
        endcase
endmodule
```

（7） 顶层模块

顶层模块需要将前面多个模块实例化，通过导线以及多路复用器将各个部件连接起来，并且在时钟的控制下修改 PC 的值，PC 是一个 32 位的寄存器，每个时钟沿自动增加 4。

代码如下：

```
module mipscpu(
	input Clk,
	input Reset
	);
	wire[31:0]TempPC,MuxPC,JumpPC,BranchPC,SquencePC,Imm32,ImmL2,R
	egWD,RsData,RtData,ALUIn2,ALURes,MemRD,Instr;
	wire [4:0] RegWA;
	wire [27:0] PsudeoPC;
	wire BranchZ,J,B,Zero,RegDst,RegWr,ALUSrc,MemWR,Mem2Reg;
	wire [1:0] ALUOp;
	wire [3:0] ALUCtr;
	reg [31:0] PC;
	assign PsudeoPC = {Instr[25:0],2'b00};
	assign JumpPC={SquencePC[31:28],PsudeoPC};
	assign SquencePC=PC+4;
	assign BranchPC=ImmL2+SquencePC;
	assign MuxPC = BranchZ?BranchPC:SquencePC;
	assign TempPC = J?JumpPC:MuxPC;
	assign BranchZ =B&Zero;
	assign ImmL2 = {Imm32[29:0],2'b00};
	assign Imm32 = {Instr[15]?16'hffff:16'h0,Instr[15:0]};
	assign ALUIn2=ALUSrc?Imm32:RtData;
	assign RegWA=RegDst?Instr[15:11]:Instr[20:16];
	assign RegWD=Mem2Reg?MemRD:ALURes;
	ALU UnitALU(RsData,ALUIn2,ALUCtr,ALURes,Zero);
	dramIP UnitDram(~Clk,MemWR,ALURes[6:2],RtData,MemRD);
	iromIP Unitirom(~Clk,PC[6:2],Instr);
	RegFile
UnitRegFile(Instr[25:21],Instr[20:16],RegWA,RegWr,RegWD,~Clk,Reset,RsData,
RtData);
	mainctr
Unitmainctr(Instr[31:26],ALUOp,RegDst,RegWr,ALUSrc,MemWR,B,J,Mem2Reg
);
	aluctr Unitaluctr(ALUOp,Instr[5:0],ALUCtr);
	always @(posedge Clk)
	begin
	if(Reset)
	PC<=0;
	else
	PC<=TempPC;
	end
endmodule
```

六.实验结果

1. **写入 MIPS 汇编语言代码.**

采用 Mars 编辑汇编源程序代码，并保存为 Task2.asm 文件。代码如图 4 所示：

```
main:    add $4,$2,$3
         lw $4,4($2)
         sw $5,8($2)
         sub $2,$4,$3
         or $2,$4,$3
         and $2,$4,$3
         slt $2,$4,$3
         beq $3,$3,equ
         lw $2,0($3)
equ:     beq $3,$4,exit
         sw $2,0($3)
exit:     j main
```

图 4 MIPS 汇编语言代码

2. **确定机器码映射格式.**

==> 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。

具体描述如下表所示：

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

==> 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。

归类为以下三种类型：

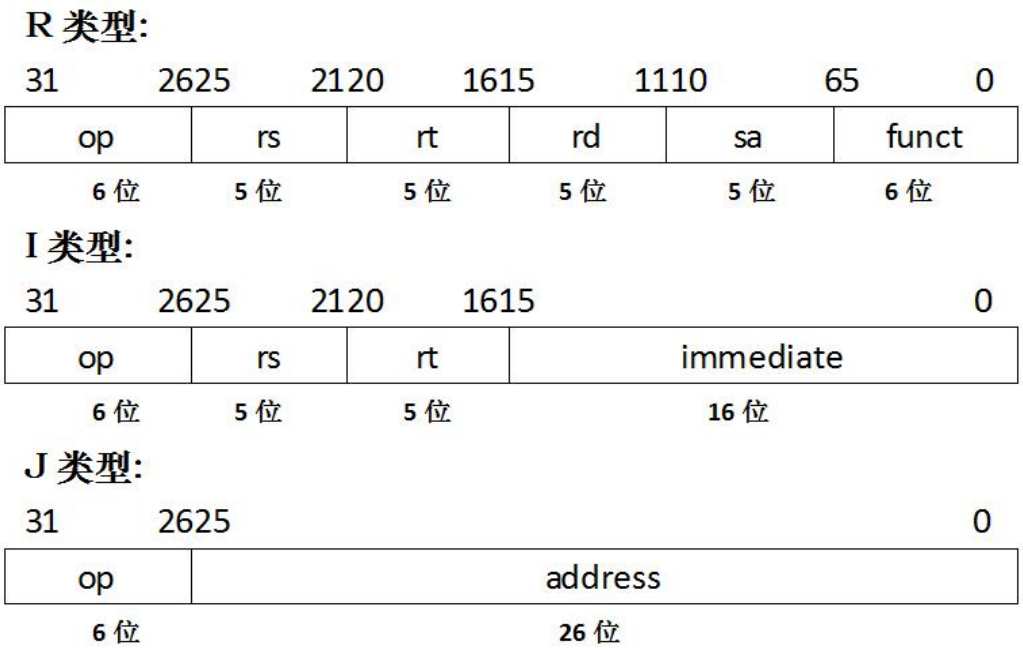


图 5 MIPS 指令的类型

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（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。

3. **获取机器代码，并保存为 coe 文件.**

（1）利用 Mars 装载 Task2.asm，并测试功能是否正常。

装载之后的用户代码段在 Mars 中的结构如图 6 所示：

Text Segment

Bkpt	Address	Code	Basic	Source
	12288	0x00432020	add $4,$2,$3	1: main: add $4,$2,$3
	12292	0x8c440004	lw $4,4($2)	2: lw $4,4($2)
	12296	0xac450008	sw $5,8($2)	3: sw $5,8($2)
	12300	0x00831022	sub $2,$4,$3	4: sub $2,$4,$3
	12304	0x00831025	or $2,$4,$3	5: or $2,$4,$3
	12308	0x00831024	and $2,$4,$3	6: and $2,$4,$3
	12312	0x0083102a	slt $2,$4,$3	7: slt $2,$4,$3
	12316	0x10630001	beq $3,$3,1	8: beq $3,$3,equ
	12320	0x8c620000	lw $2,0($3)	9: lw $2,0($3)
	12324	0x10640001	beq $3,$4,1	10: equ: beq $3,$4,exit
	12328	0xac620000	sw $2,0($3)	11: sw $2,0($3)
	12332	0x08000c00	j 12288	12: exit: j main

图 6 Mars 中的代码段

（2）利用 Mars 导出机器码：

将机器码存为“task2.coe”的 coe 文件，内容如图 7.

```
MEMORY_INITIALIZATION_RADIX=16;
MEMORY_INITIALIZATION_VECTOR=
00432020
8c440004
ac420008
00831022
00831025
00831024
0083102a
10830002
10650001
8c620000
08000000
```

图 7 “task2.coe”文件内容

4. 仿真验证 CPU 的正确性.

使用Vivado进行仿真，得到仿真波形图，通过分析波形图中当前PC值和返回PC值、指令、控制单元、寄存器值和存储器值等来判断程序的正确性。测试文件中时钟和置位端口初始化

及行进模块如下。

代码如下：

```
module mipscpu(
    input Clk,
    input Reset
    );
    wire                                                    [31:0]
TempPC,MuxPC,JumpPC,BranchPC,SquencePC,Imm32,ImmL2,RegWD,RsData,
RtData,ALUIn2,ALURes,MemRD,Instr;
    wire [4:0] RegWA;
    wire [27:0] PsudeoPC;
    wire BranchZ,J,B,Zero,RegDst,RegWr,ALUSrc,MemWR,Mem2Reg;
    wire [1:0] ALUOp;
    wire [3:0] ALUCtr;
    reg [31:0] PC;
    assign PsudeoPC = {Instr[25:0],2'b00};
    assign JumpPC={SquencePC[31:28],PsudeoPC};
    assign SquencePC=PC+4;
    assign BranchPC=ImmL2+SquencePC;
    assign MuxPC = BranchZ?BranchPC:SquencePC;
    assign TempPC = J?JumpPC:MuxPC;
    assign BranchZ =B&Zero;
    assign ImmL2 = {Imm32[29:0],2'b00};
    assign Imm32 = {Instr[15]?16'hffff:16'h0,Instr[15:0]};
    assign ALUIn2=ALUSrc?Imm32:RtData;
    assign RegWA=RegDst?Instr[15:11]:Instr[20:16];
    assign RegWD=Mem2Reg?MemRD:ALURes;
    ALU UnitALU(RsData,ALUIn2,ALUCtr,ALURes,Zero);
    dramIP UnitDram(~Clk,MemWR,ALURes[6:2],RtData,MemRD);
    iromIP Unitirom(~Clk,PC[6:2],Instr);
   // dram UnitDram(ALURes[6:2],MemRD,RtData,MemWR);
   // irom Unitirom(PC[6:2],Instr);
    RegFile
    UnitRegFile(Instr[25:21],Instr[20:16],RegWA,RegWr,RegWD,~Clk,Reset,Rs
    Data,RtData)
    mainctrUnitmainctr(Instr[31:26],ALUOp,RegDst,RegWr,ALUSrc,MemWR,B,
    J,Mem2Reg);
```

```
    aluctr Unitaluctr(ALUOp,Instr[5:0],ALUCtr);
    always @(posedge Clk)
    begin
    if(Reset)
    PC<=0;
    else
    PC<=TempPC;
    end
endmodule
```

指令代码表如下：

行号	汇编程序	指令代码					
		op（6）	rs(5)	rt(5)	rd(5)/immediate(16)		16 进制数代码
1	add $4, $2, $3	000000	00010	00011	0010 0000 0000 0000	=	00432020
2	lw $4, 4($2)	100011	00010	00010	0000 0000 0000 0100	=	8c440004
3	sw $5, 8($2)	101011	00010	00010	0000 0000 0000 1000	=	ac420008
4	sub $2, $4, $3	000000	00100	00011	0001 0000 0010 0010	=	00831022
5	or $2, $4, $3	000000	00100	00011	0001 0000 0010 0101	=	00831025
6	and $2, $4, $3	000000	00100	00011	0001 0000 0010 0100	=	00831024
7	slt $2, $4, $3	000000	00100	00011	0001 0000 0010 1010	=	0083102a
8	beq $3, $3, equ	000000	00100	00011	0000 0000 0000 0010	=	10830002
9	lw $2, 0($3)	00100	00011	00101	0000 0000 0000 0001	=	10650001
10	beq $3, $4, exit	100011	00011	00010	0000 0000 0000 0000	=	8c620000
11	j main	000010	00000	00000	0000 0000 0000 0000		08000000

根据上表，使用 Vivado 仿真结果来验证微处理器的正确性：

使用 Vivado 仿真如图 8。

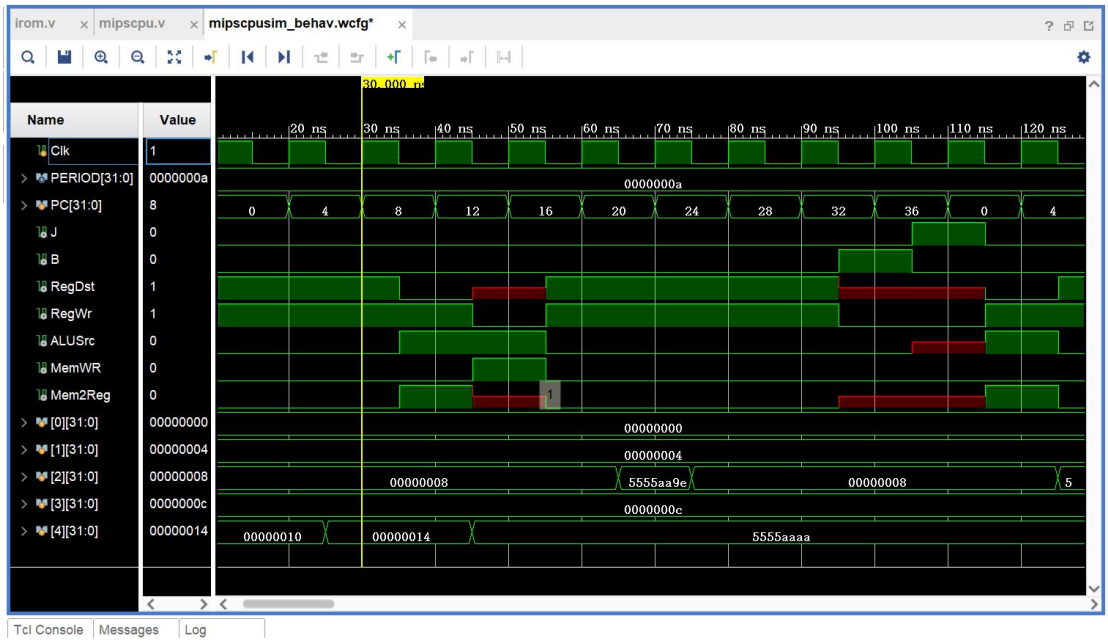
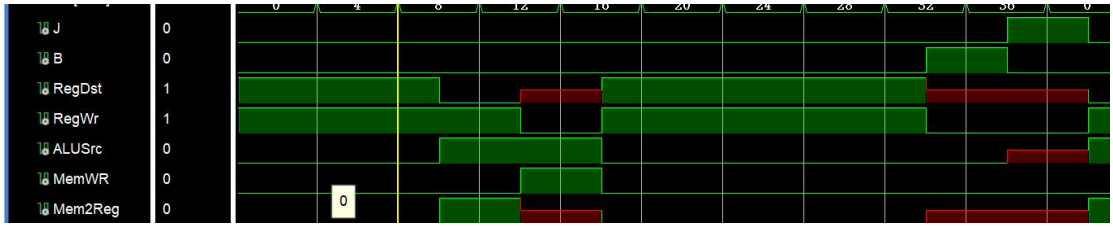


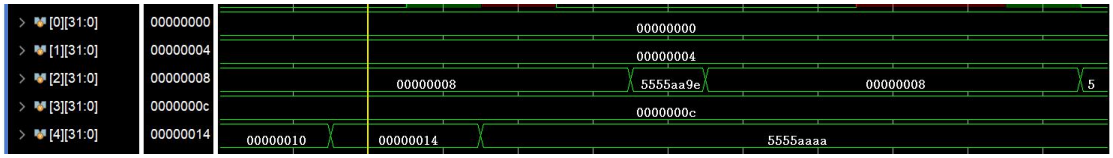
图 8 仿真结果

我们通过观察 PC 的值：

控制信号的值：



以及 0~4 号寄存器的值的变化：



可知，结果与期望相符，所设计的微处理器能达到基本功能。

七. 实验总结

在我看来，这是一次有趣又充满挑战性的实验。

首先，在硬件设计中，因为我对上学期数电的HDL语言的知识有所遗忘，所以这次实验基本就是重新学了一遍。关于结构，主要就是对那个数据通路图的熟悉，各条指令怎么一步

一步被执行，数据分配等等。左老师在网课的视频里带着我们把所有指令的走向都分析了一遍，听完基本就有很清晰的概念了。

下面是我在实验中遇到的一些问题：

（1）变量名的问题：同一条线连在不同模块接口名字不一样，所以导致有点混乱，这个在仿真时会有信息提示可以去找。

（2）关于 Jump：26 位地址移位变 28 位，余下的四位由 PC+4 的高四位补上，我写的时候不小心把 31:28,写成了 31:29，然后这个小错误就很难发现。

（3）信号的位宽：那几个选择器看起来差不多，我没注意到一个信号是 4 位的，另一个是 32 位的，而直接复制了过来。这一个小细节，虽然看起来很简单，但是往往显示错误不会在这里报错，最后我只好一个模块接着一个模块检查才排查到这里。

总之，通过这次实验，我了解了微处理器的基本结构，掌握了哈佛结构的计算机工作原理，学会了设计简单的微处理器，同时，了解了软件控制硬件工作的基本原理。 对于硬件开发也有了初步的心得体会：首先，做事一定要冷静，过程中有时会出现各种各样的 BUG，辛辛苦苦改了几遍，都出现莫名其妙的问题，同学们都说没遇到过我的问题，最后，我利用网上的电子论坛，通过高手写的博客与帖子，找到了解决方法。其次，分析波形一定要冷静下来，不然很容易看到波形头晕眼花。

我在这次实验中学到了许多，更重要的是对于硬件电路与设计与计算机的思维模式有了更加直接与深刻的理解。