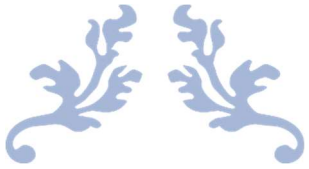

MIPS 微处理器设计

华中科技大学

江榕煜

电信 1805 班

U201890043

简单指令集 MIPS 微处理器设计

一、实验内容

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、通用寄存器写入、数据存储器数据写入都在时钟下降沿完成
- 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

完成完整设计代码输入、各模块完整功能仿真，整体仿真，验证所有给定汇编语言程序指令执行情况。

二、实验目的

1. 了解 MIPS 微处理器的基本结构；
2. 掌握哈佛结构的计算机工作原理；
3. 学会设计简单的微处理器；
4. 学会软件控制硬件工作的基本原理；
5. 学会使用 Vivado 软件。

三、实验环境

1. 操作系统：Windows 10 Pro
2. Verilog 开发 IDE：Xilinx Vivado 2019.1
3. MIPS 汇编 IDE：MARS 4.5

四、MIPS 微处理器设计

整体系统设计框架如下图 1，并做各子模块设计说明如下：

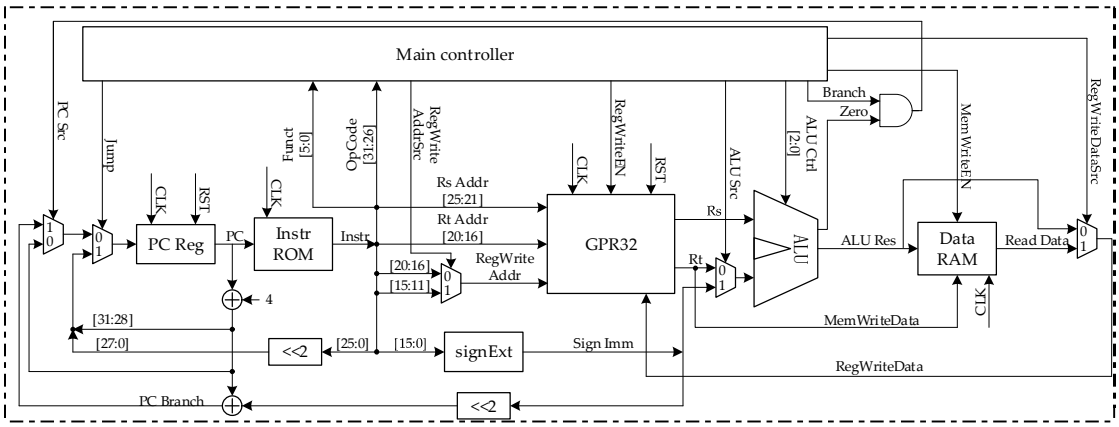

图 1 整体系统框图

本设计中将整体的微控制分模块实现，分别包含：指令存储器模块、数据存储器模块、通用寄存器模块、主控制器模块、ALU 模块、有符号数扩展模块、有符号数移位模块。最后将各模块按照上整体系统框图进行拼接、形成整体微处理器。

1. 指令存储器

分析本次实验内容需求：指令存储器在上升沿读出指令、指令存储器的容量为 32*32、首地址从 0 开始、初始化装载指定测试的汇编指令机器码。

本模块代码如下模块 1 所示。首先使用 reg 语法声明了 32 个位宽为 32 位的存储单

元；并且在 CLK 时钟信号上升沿到来时，将 addr 地址线对应的 data 数据输出；在初始化过程中，首先使用 for 循环将所有存储单元默认初始化为其地址值，再读入指定的指令机器码文件。

```
module INSTR_ROM(
    input [4:0] addr,    //地址线
    output reg[31:0] data,   //32 位数据线
    input CLK    //同步时钟
    );
    reg[31:0] ROM [0:31];    //32*32 存储单元

    always@(posedge CLK)      //在时钟上升沿读出数据 or 指令
    begin
        data = ROM[addr];
    end

    integer i;
    initial
    begin
        for(i=0;i<32;i=i+1) //默认初始化存储单元内容为地址
            ROM[i]=i;

        //初始化存储单元后读入指令文件
        $readmemh("D:/CollegeHomework/MIPS32/executeCode.txt",ROM);
    end
endmodule
```

模块 1 指令存储器

2. 数据存储器

设计需求：写入数据在时钟下降沿完成、大小为 32*32、首地址从 0 开始、初始值为对应的地址值。

数据存储器代码如下模块 2 所示。首先声明 32*32 的存储单元，并且在初始化的过程中，赋值为相应的地址。当 CLK 时钟下降沿到来时，若 WriteEN 为高电平，则写入数据。

```
module DATA_RAM (
    input [4:0] addr,
    output [31:0] ReadData,
    input [31:0] WriteData,
    input WriteEN,
    input CLK
    );
    reg[31:0] RAM[0:31];
    assign ReadData = RAM[addr];
    always @(negedge CLK)
        if(WriteEN & ~CLK)
            RAM[addr] = WriteData;
```

```
    integer i;
    initial
        for(i = 0;i<32;i=i+1)
            RAM[i]=i;
endmodule
```

模块 2 数据存储器

3. 算术逻辑单元 ALU

模块设计需求：能完成两个 32 位数据之间的加、减、与、或、移位功能，并且附加判断结果是否为 0 信号。

算术逻辑单元的代码如下模块 3 所示。ALU 模块共有两个输出信号，RES 计算结果和 ZERO 零结果信号。共有三组输入信号，in1 和 in2 为计算数据信号，CTRL 为控制信号。CTRL 信号通过译码来控制 ALU 的计算。

```
module ALU(
    output reg [31:0] RES,
    output ZERO,
    input [31:0] in1,
    input [31:0] in2,
    input [2:0] CTRL
    );
  assign ZERO = (RES == 0);
  always @(*) begin
    case (CTRL)
      3'b001:   RES = in1 & in2;
      3'b010:   RES = in1 + in2;
      3'b011:   RES = in1 | in2;
      3'b110:   RES = in1 - in2;
      3'b111:   RES = in1 < in2;
      default: RES = 32'hFFFFFFFF;
    endcase
  end
endmodule
```

模块 3 算术逻辑单元

4. 通用寄存器

设计需求：寄存器写入在时钟下降沿完成、复位时各寄存器取值为对应地址乘四。

通用寄存器模块代码如下模块 4 所示。首先声明寄存器为 32 个 32 位空间；若地址输入为 0，则直接输出数据为 0；否则直接将给定地址寄存器数据输出。在时钟下降沿到来时，若写使能信号为高有效，则写入数据到寄存器。在时钟沿跳变时，若复位信号为高有效，则复位各寄存器内容为其相应地址乘以四。

```
module GPR32(
    output [31:0] RsData,
    output [31:0] RtData,
    input [31:0] WriteData,
    input [4:0] RsAddr,
```

```
        input [4:0] RtAddr,
        input [4:0] WriteAddr,
        input WriteEN,
        input CLK,
        input RST
        );
    reg[31:0] GPR[31:0];
    integer i;

    assign RsData = (RsAddr != 0) ? GPR[RsAddr] : 0;   //GPR[0] is $zero
    assign RtData = (RtAddr != 0) ? GPR[RtAddr] : 0;
    always @(negedge CLK) //时钟下降沿写入数据
        if(WriteEN) GPR[WriteAddr] <= WriteData;

    always @(CLK) //时钟上升、下降沿都可做同步复位
        if(RST)
            for(i=0;i<32;i=i+1)
                GPR[i] = i*4; //对应地址乘四
endmodule
```

模块 4 通用寄存器模块

5. 符号数扩展

符号数扩展模块代码如下模块 5 所示。该模块采用最高位扩展拼接的方式实现，同时为了提高模块复用率，采用了可变参数定义。

```
module signExt #(parameter resW = 32,srcW = 16)(
    output [resW-1:0] extO,
    input [srcW-1:0] extI
    );
    assign extO = {{(resW-srcW){extI[srcW-1]}}, extI};
endmodule
```

模块 5 有符号数扩展

6. 移位器

本实验中实现的移位器，实现的功能为：有符号左移两位。实现代码如下模块 6 所示。为了后期更加通用设计，本有符号左移位器采用了可变参数设计。

```
module signedShiftLeft #(parameter shiftBits = 2, dataWidth = 32)(
                        output[dataWidth-1:0] shiftRes,
                        input[dataWidth-1:0] shiftSrc);
    assign shiftRes = {shiftSrc[dataWidth - 1], shiftSrc[(dataWidth – shiftBits - 2): 0],
{shiftBits{1'b0} }};
endmodule
```

模块 6 移位器

7. 主控制器

首先分析本次测试代码中必须要执行的指令有：寄存器加减法、数据存取、逻辑与或、逻辑相等跳转、立即数跳转。所以控制器的输出控制信号必须包括有：ALU 控制信号、寄存器写入数据选取信号、存储器写使能信号、分支跳转信号、ALU 的第二操作数

选取信号、寄存器写入地址选取信号、寄存器写入使能信号、跳转信号。由 MIPS 架构的 R、J、I 型指令的结构可得，主控制器的输入信号为：指令码的高 6 位操作码、低 6 位功能码。

该模块代码如下模块 7 所示。主要思想是将所有的控制输出信号，统一连接到内部控制寄存器 Ctr。随后通过译码，给 Ctr 寄存器一次性赋值。

```
module mainController(
      output [2:0] AluCtrl,
      output RegWriteDataSrc,
      output MemWriteEN,
      output Branch,
      output AluSrc,
      output RegWriteAddr,
      output RegWriteEN,
      output Jump,
      input [5:0] OpCode,
      input [5:0] Funct
      );
   reg[9:0] Ctr;
   assign
{RegWriteEN,MemWriteEN,RegWriteAddr,AluSrc,RegWriteDataSrc,Branch,Jump,Al
uCtrl} = Ctr;
   always @(*)
      case (OpCode)
         6'h23:     Ctr <= 10'b1001100010; // lw, +
         6'h08:     Ctr <= 10'b1001000010; // addi, +
         6'h2B:     Ctr <= 10'b0101000010; // sw, +
         6'h04:     Ctr <= 10'b0000010110; // beq, -
         6'h02:     Ctr <= 10'b0000001000; // j
         6'h00:
            case (Funct)                              // R-tpye
               6'h20:     Ctr <= 10'b1010000010; // add
               6'h22:     Ctr <= 10'b1010000110; // sub
               6'h24:     Ctr <= 10'b1010000001; // and
               6'h25:     Ctr <= 10'b1010000011; // or
               6'h2A:     Ctr <= 10'b1010000111; // slt
               default: Ctr <= 10'b1010000000;
            endcase
         default: Ctr <= 10'h0;
      endcase
endmodule
```

模块 7 主控制器

8. Mips 处理器核心

本设计中 MIPS 处理器代码如下模块 8 所示。在该模块中，实例化了主控制器、通用寄存器、有符号数扩展器、有符号数左移 2 位器、PC 寄存器（如下模块 9 所示）、

ALU 模块。再实例化相应的信号线（见框图 1），将这些模块进行逻辑拼接。

```
module MIPS32(
    output [31:0] ALURes,
    output [31:0] MemWriteData,
    output MemWriteEN,
    output [31:0] PC,
    input [31:0] ReadData,
    input [31:0] Instr,
    input CLK,
    input RST
    );
  wire RegWriteDataSrc,Branch,ALUSrc,Zero,RegWriteEN,RegWriteAddrSrc,PCSrc,Jump;
  wire[2:0] ALUCtrl;
  wire[4:0] RegWriteAddr;
  wire[31:0] Rs,Rt,RegWriteData;
  wire[31:0] SignImm;

  //实例化主控制器
  assign PCSrc = Branch & Zero;
  mainController mainCtrl(ALUCtrl,RegWriteDataSrc,MemWriteEN,Branch,
                          ALUSrc,RegWriteAddrSrc,RegWriteEN,Jump,
                          Instr[31:26],Instr[5:0]);

    //实例化通用寄存器
    assign RegWriteData = RegWriteDataSrc ? ReadData : ALURes;
    assign RegWriteAddr = RegWriteAddrSrc ? Instr[15:11] : Instr[20:16];
    assign MemWriteData = Rt;
    GPR32 generalPurposeRegister(Rs,Rt,
              RegWriteData,RegWriteAddr,
              Instr[25:21],Instr[20:16],
              RegWriteEN,CLK,RST);

    //实例化有符号数扩展器
    signExt signedDataExtender(SignImm,Instr[15:0]);
    //实例化有符号数左移器
    wire[31:0] signedExtShift2Bits;
    signedShiftLeft #(2,32) signedShiftLeft2Bits(signedExtShift2Bits,SignImm);

    //实例化 PC 寄存器
    wire[31:0] nextPC,PCbranch,PCplus4;
    PCReg pcRegister(PC,nextPC,CLK,RST);
    //实例化 PC 自递增
    assign PCplus4 = PC + 4;
    //实例化直接跳转地址计算器
```

```
    wire[27:0] directJumpAddr;
    assign directJumpAddr = {PCplus4[31:28],Instr[25:0],2'b00};
    //实例化 nextPC 源控制逻辑
    assign PCbranch = PCplus4 + signedExtShift2Bits;
    assign nextPC = Jump ? directJumpAddr :(PCSrc ? PCbranch : PCplus4);

    //实例化 ALU
    wire[31:0] ALU_data2;
    assign ALU_data2 = ALUSrc? SignImm : Rt;
    ALU alu(ALURes,Zero,
            Rs,ALU_data2,ALUCtrl);

endmodule
```

模块 8　MIPS 处理器核心

```
module PCReg #(parameter width=32)
            (output reg[width-1:0] Q, input[width-1:0] D, input CLK, Reset);
  always @(negedge CLK, posedge Reset) //指令指针在下降沿修改
    if (Reset) Q <= 0;
    else       Q <= D;
endmodule
```

模块 9　PC 寄存器

9. 整体系统

本设计中，采用了存储器和处理器分离的框架结构。

如下模块 10 所示，在该顶层系统模块中，将 MIPS 处理器核心、指令存储器、数据存储器进行了拼接。

```
module SYSTEM(
    input CLK,
    input RST
    );
    wire[31:0] Instr, PC, ReadData, ALURes, MemWriteData;
    wire MemWriteEN;
  MIPS32 mips(ALURes, MemWriteData, MemWriteEN, PC, ReadData, Instr, CLK, RST);
  INSTR_ROM imem(PC[6:2],Instr,CLK);
  DATA_RAM dmem(ALURes[4:0],ReadData,MemWriteData,MemWriteEN,CLK);
endmodule
```

模块 10　顶层系统模块

五、汇编指令生成和导入 ROM

本次实验测试的 MIPS 汇编代码如下代码块 1 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
```

代码块 1　实验测试汇编程序

将汇编程序导入 MIPS32 编辑仿真软件 MARS 中，并且使用该软件进行汇编，如下图 2：

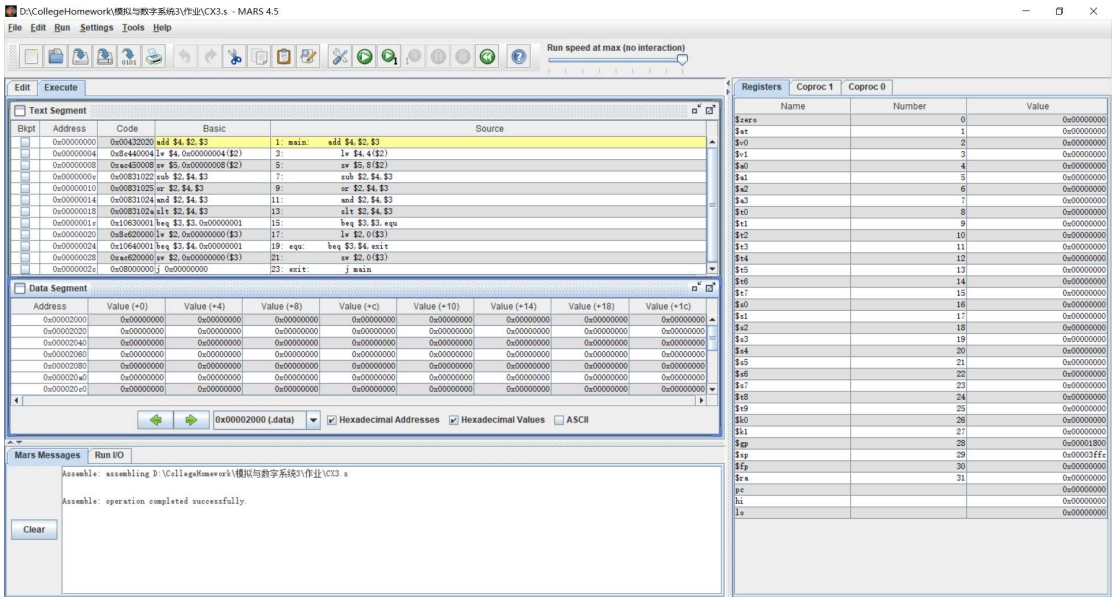

图 2　MARS 汇编

汇编生成的机器码和对应程序如下表 1 所示：

机器码	汇编程序	机器码（续）	汇编程序（续）
00432020	main:　add $4,$2,$3	0083102a	slt $2,$4,$3
8c440004	lw $4,4($2)	10630001	beq $3,$3,equ
ac450008	sw $5,8($2)	8c620000	lw $2,0($3)
00831022	sub $2,$4,$3	10640001	equ:　beq $3,$4,exit
00831025	or $2,$4,$3	ac620000	sw $2,0($3)
00831024	and $2,$4,$3	08000000	exit:　j main

表 1　汇编程序对应的机器码

将机器码以十六进制形式导出到文本文件中，并且在 Vivado 中的指令存储器模块中，引用指定的机器码文件。如下图 3 和图 4 所示：

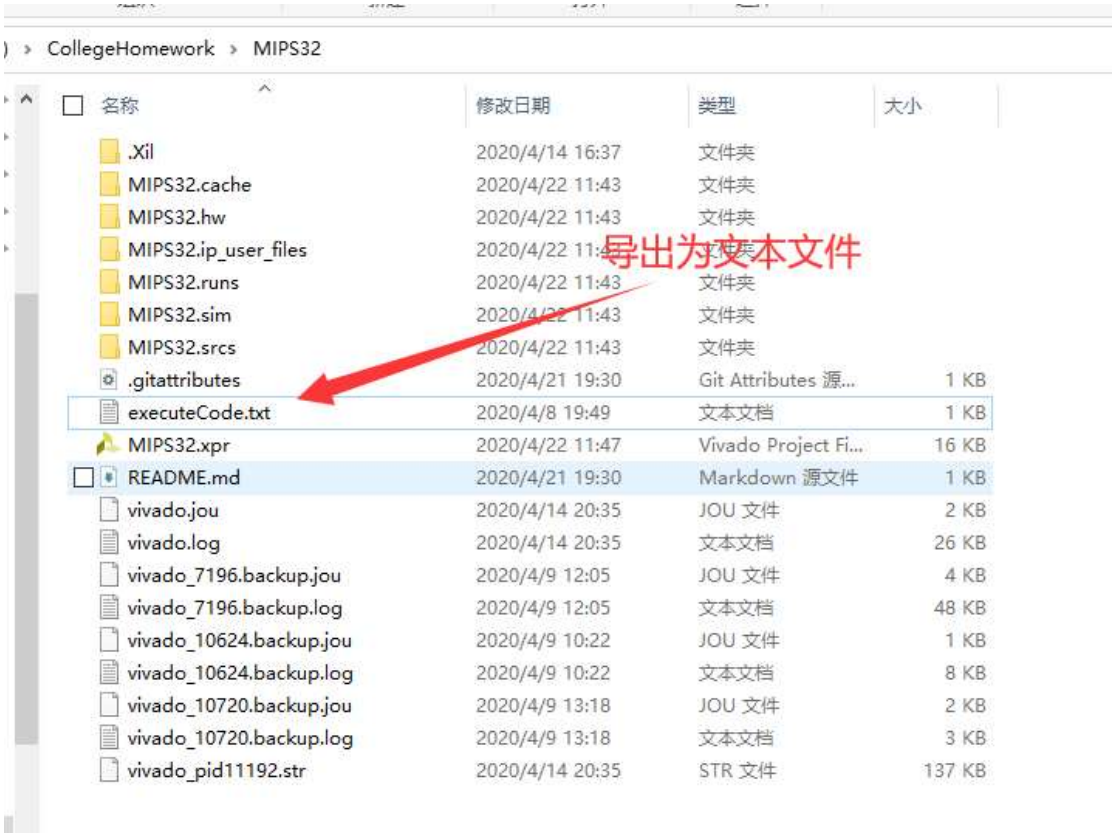

图 3　导出机器码文本文件

图 4　导入机器码到指令存储器中

六、模块仿真

1. 指令存储器仿真

该部分仿真代码如下模块 11 所示。首先对要测试的指令存储器模块做实例化，并且连接地址、数据、时钟总线；在初始化代码中，设置地址线为 0；接着，在时钟生成模块，每隔半个周期，时钟电平跳变一次；最后，使用触发逻辑，在 CLK 时钟下降沿的时候，改变地址总线数据，每次改变地址+1，若到达 32 则返回到 0。

```
module ROM_sim(
    output [4:0] addr,
    output [31:0] data
    );
```

```
    reg[4:0] simAddr;
    reg CLK;
    assign addr = simAddr;

    //实例化指令存储器
    INSTR_ROM rom(simAddr,data,CLK);

    initial
    begin
        simAddr = 0; //初始化地址为 0
    end

    parameter PERIOD = 10;
    always begin        //时钟生成模块
        CLK = 1'b1;
        #(PERIOD/2) CLK = 1'b0;
        #(PERIOD/2);
    end

    always @(negedge CLK)   //时钟下降沿改变地址
    begin
        simAddr = simAddr+1; //地址递增
        if(simAddr == 32) simAddr = 0;   //地址溢出
    end
endmodule
```

模块 11 指令存储器测试代码

仿真结果如下图 5 所示。观察仿真波形图，可清晰看到，在时钟上升沿时，data 总线上的电平发生变化，即取出指令存储器数据；在时钟下降沿时，addr 总线上的数据发生变化，即改变指令地址；并且对照（五）中生成的机器码可发现，读取出来的机器码是正确的，在没有相应机器码的位置，也将存储器内容初始化成了相应地址。

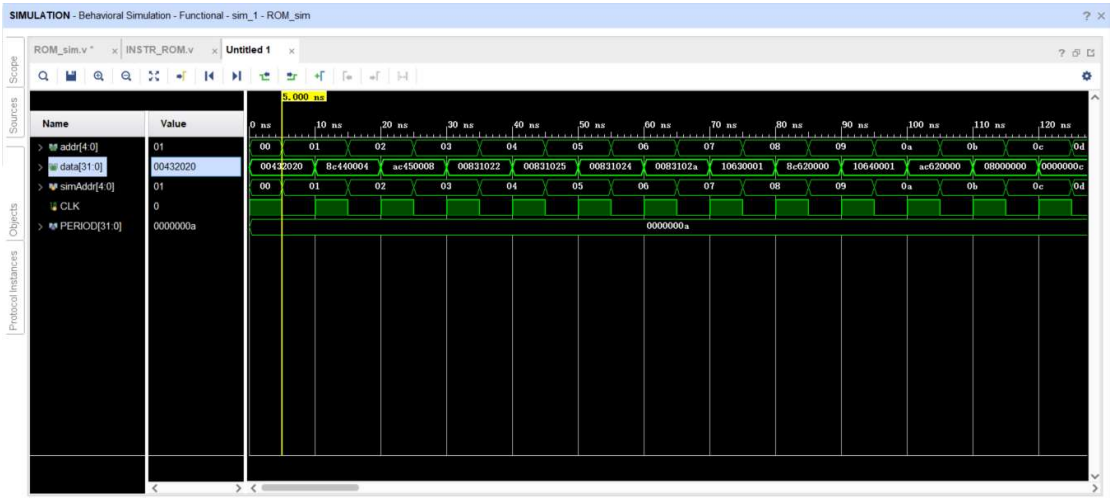

图 5 指令存储器仿真波形

2. 数据存储器仿真

该部分仿真代码如下模块 12 所示。该模块仿真的思想为：将 CLK 时钟做二分频后输入给 WriteEN 写使能信号，并且在写使能信号的上升沿，dataW 写数据总线上的数据+1，为了测试方便写数据到 5 时清零。并且在初始化阶段，addr 初始化地址总线为 2，即固定读写地址为 2。

```
module RAM_sim(
    output reg CLK,
    output reg [4:0] addr,
    output [31:0] dataR,
    output reg [31:0] dataW
    );
    reg WriteEN;
    DATA_RAM ram(addr,dataR,dataW,WriteEN,CLK);

    initial
    begin
    WriteEN = 0;
    addr = 2;
    dataW = 0;
    end

    always @(posedge CLK)
    begin    //写使能做时钟二分频
        WriteEN = ~WriteEN;
    end

    always @(posedge WriteEN)
    begin   //写使能上升沿写数据+1
        dataW = dataW +1;
        if(dataW == 5)   //为测试方便，测试数据 5 循环
            dataW = 0;
    end

    parameter PERIOD = 10;
    always begin        //时钟生成模块
      CLK = 1'b1;
      #(PERIOD/2) CLK = 1'b0;
      #(PERIOD/2);
    end
```

模块 12 数据存储器仿真模块

仿真结果如下图 6 所示。为将测试结果更加直观地展示出来，dataW 写数据总线和 dataR 读数据总线上的数据采用“模拟-阶梯波”的形式显示。由测试波形可以看出，在 CLK 时钟下降沿到来和 WriteEN 写使能信号保持高电平时，数据被写入数据存储器。并且，未被修改的数据存储器中，数据保持初始数据，即相应的地址值。

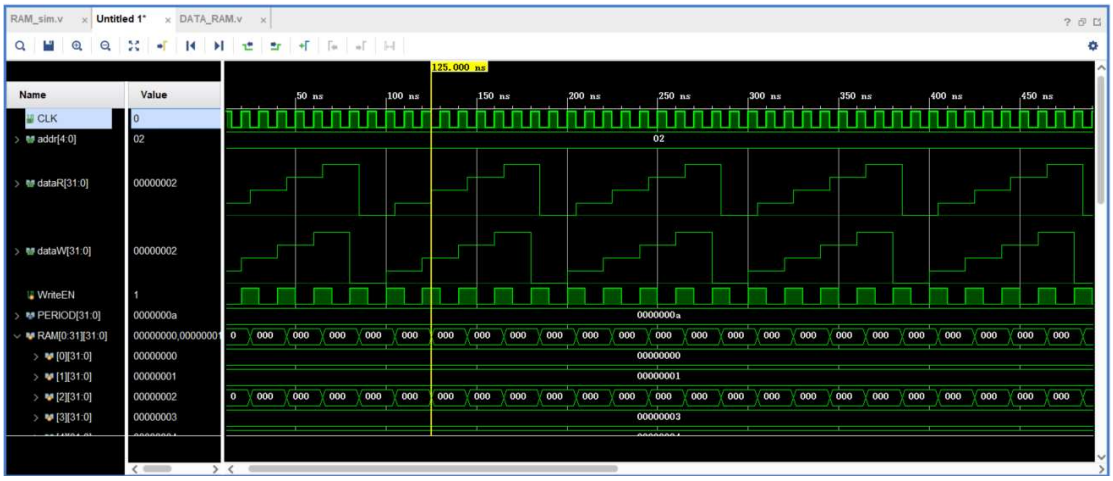

图 6　数据存储器仿真波形

3. 算术逻辑单元 ALU

该部分仿真代码如下模块 13 所示。首先实例化 ALU 模块；并且初始化过程中，给定一组计算数为 0x5f5f2a2a 和 0x1a1ac9c9；接着在 always 语句块中，每隔 10ns 更改一次控制信号，即改变 ALU 工作模式。

```
module ALU_sim(
    output [31:0] ALU_RES,
    output ALU_ZERO,
    output reg [31:0] ALU_I1,
    output reg [31:0] ALU_I2,
    output reg [2:0] ALU_CTRL
    );

    //ALU 模块实例化
    ALU alu(ALU_RES,ALU_ZERO,ALU_I1,ALU_I2,ALU_CTRL);

    initial
    begin
        ALU_I1 = 32'h5f5f2a2a;
        ALU_I2 = 32'h1a1ac9c9;
    end

    always begin
            ALU_CTRL = 3'b001; //逻辑与
        #10 ALU_CTRL = 3'b010; //算术加
        #10 ALU_CTRL = 3'b011; //逻辑或
        #10 ALU_CTRL = 3'b110; //算术减
        #10 ALU_CTRL = 3'b111; //左移
        #10;
    end
endmodule
```

模块 13　算术逻辑单元仿真模块

仿真结果如下图 7 所示。ALU 的控制信号使之依次工作在与、加、或、减、左移模式下，且 ALU_RES 信号上的计算结果都正确；并且当 ALU_RES 为 0 时，ALU_ZERO 信号拉高。可见，该模块工作正常

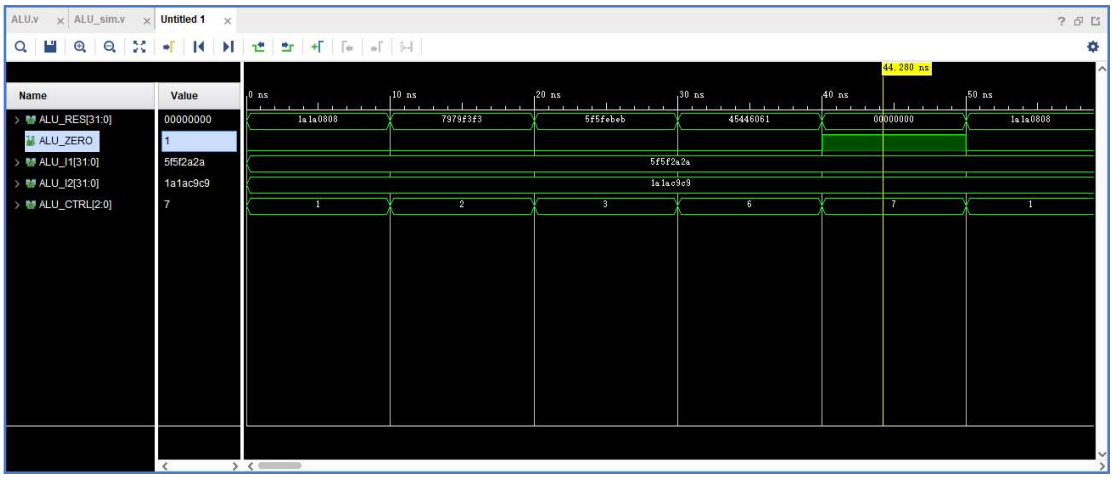

图 7　算术逻辑单元仿真波形图

4. 通用寄存器

该部分仿真代码如下模块 14 所示。模块中首先做时钟的生成，周期为 10ns。在初始化的过程中，设置读取地址 addr1 为 5，读取地址 addr2 和写入地址 addrW 为 6，写入数据为 1，写使能信号 WriteEN 为低，并且给一个指针周期的复位信号。在每一次时钟上升沿到来时，做写使能信号翻转，即时钟二分频，同时写数据递增。

```
module GPR32_sim(
    output reg [4:0] addr1,
    output reg [4:0] addr2,
    output reg [4:0] addrW,
    output [31:0] data1,
    output [31:0] data2,
    output reg [31:0] dataW,
    output reg WriteEN,
    output reg CLK,
    output reg RST
    );
    //时钟生成模块
    parameter PERIOD = 10;
    always begin
       CLK = 1'b0;
       #(PERIOD/2) CLK = 1'b1;
       #(PERIOD/2);
    end
    //实例化通用寄存器模块
    GPR32 generalPurposeRegister(data1,data2,dataW,
          addr1,addr2,addrW,WriteEN,CLK,RST);

    initial
```

```
        begin //初始化
            RST = 1;
            addr1 = 5;
            addr2 = 6;
            addrW = 6;
            dataW = 1;
            WriteEN = 0;
            #PERIOD RST = 0;
        end

        always@(posedge CLK)
        begin //时钟信号二分频作为写使能
            WriteEN <= ~WriteEN;
            //写数据递增
            dataW <= dataW + 1;
            if(dataW == 7) dataW <=0;
        end
endmodule
```

模块 14　通用寄存器仿真模块

仿真结果如下图 8 所示。可见，在复位信号到来后，各寄存器初始化的数据为其地址的 4 倍；当时钟下降沿和写使能为高时，数据被写入通用寄存器。

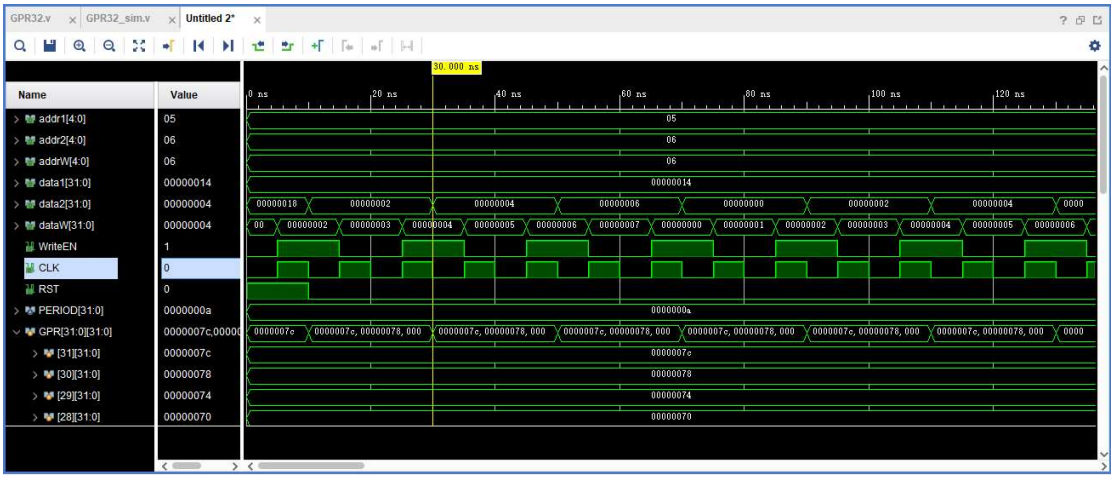

图 8 通用寄存器仿真结果

5. 符号数扩展

该模块仿真代码如下模块 15 所示。将输入的带扩展数初始化为 0xfffb，并且在每个时钟上升沿递增。在数个时钟周期后，即可看到符号数变为正。

```
    module signExt_sim(
        output [31:0] extRES,
        output reg [15:0] extSrc,
        output reg CLK
        );
        parameter PERIOD = 10;
        always begin        //时钟生成模块
```

```
            CLK = 1'b1;
            #(PERIOD/2) CLK = 1'b0;
            #(PERIOD/2);
            end

            signExt#(32,16)signExtension(extRES,extSrc);
            initial extSrc = 16'b1111111111111010;
            always@(posedge CLK)
                  extSrc = extSrc+1;
endmodule
```

模块 15 有符号数扩展

仿真结果如下图 9 所示。该模块工作正常。

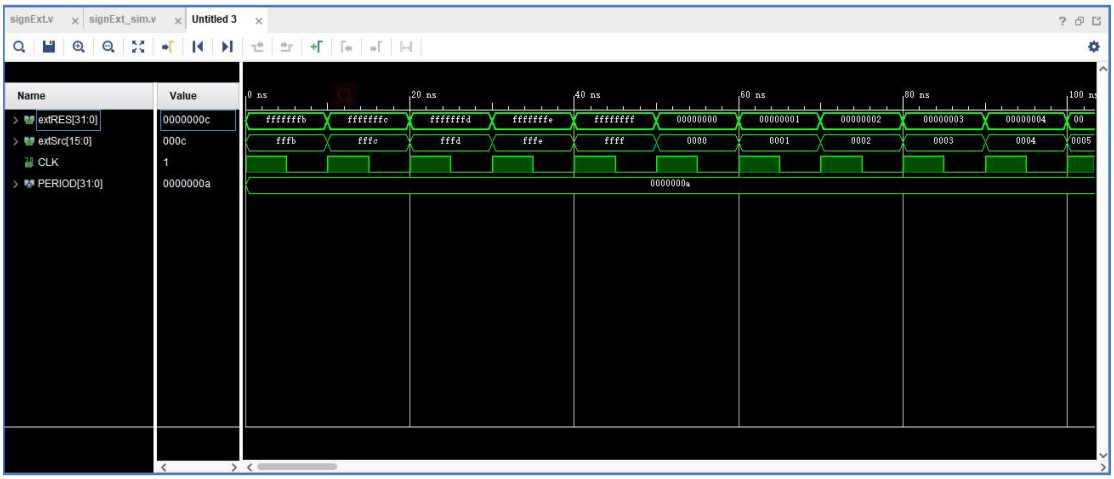

图 9 有符号数扩展

6. 移位器

该模块仿真代码如下模块 16 所示。

```
module shift_sim(
      output reg CLK,
      output reg [31:0] shiftData,
      output [31:0] shift2BitsRes
      );
    parameter PERIOD = 10;
     //时钟发生模块
    always begin
        CLK = 1'b0;
        #(PERIOD/2) CLK = 1'b1;
        #(PERIOD/2);
    end
    //实例化移位有符号左移 2 位模块
    signedShiftLeft #(2,32)sl2(shift2BitsRes,shiftData);

    always @(posedge CLK) //每个时钟上升沿移位数据+1
          shiftData = shiftData+1;
```

```
    initial shiftData = 32'hfffffffa;
endmodule
```

模块 16 有符号数左移 2 位仿真代码

仿真结果如下图 10 所示。

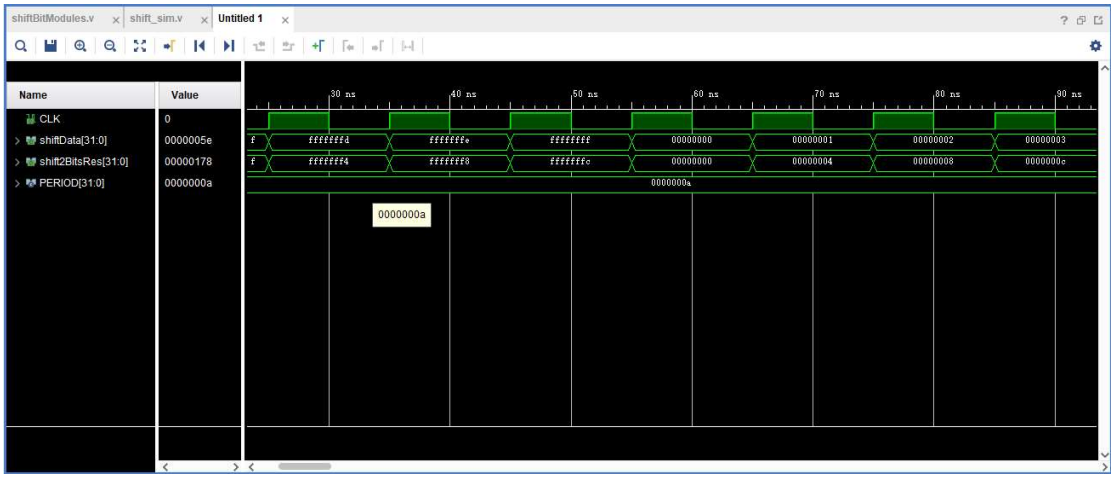

图 10 移位器仿真波形

7. 主控制器

该模块的仿真代码如下模块 17 所示。该仿真模块在测试主控制器的同时，直接使用本次实验需要的测试代码指令。首先实例化在前述测试好的指令存储器；并在每次时钟下降沿累加读取地址；最后实例化主控制器模块，将指令存储器读出的数据传入主控制器。

```
module mainCTRL_sim(
    output reg CLK,
    output [31:0] INSTR,
    output Jump,
    output RegWriteDataSrc,
    output MemWriteEN,
    output Branch,
    output [2:0] AluCtrl,
    output AluSrc,
    output RegWriteSrc,
    output RegWriteEN
    );
    reg[4:0] simAddr;
    //实例化指令存储器
    INSTR_ROM rom(simAddr,INSTR,CLK);
    //实例化主控制器
    mainController mainCtrl(AluCtrl,
            RegWriteDataSrc,MemWriteEN,
            Branch,AluSrc,
            RegWriteSrc,RegWriteEN,
            Jump,
```

```
                INSTR[31:26],INSTR[5:0]);

    initial
    begin
        simAddr = 0; //初始化地址为 0
    end

  parameter PERIOD = 10;
  always begin        //时钟生成模块
      CLK = 1'b1;
      #(PERIOD/2) CLK = 1'b0;
      #(PERIOD/2);
  end

    always @(negedge CLK)   //时钟下降沿改变地址
    begin
        simAddr = simAddr+1; //地址递增
        if(simAddr == 12) simAddr = 0;   //地址溢出
    end
endmodule
```

模块 17 主控制器仿真

该模块仿真结果如下图 x 所示。对照本次实验给出的各指令，和设计框图，可见主控制器工作正常。例：在指令为 0x08000000 时，即（j 0）指令，除 Jump 信号拉高外，其他信号都为低，可见工作正常。

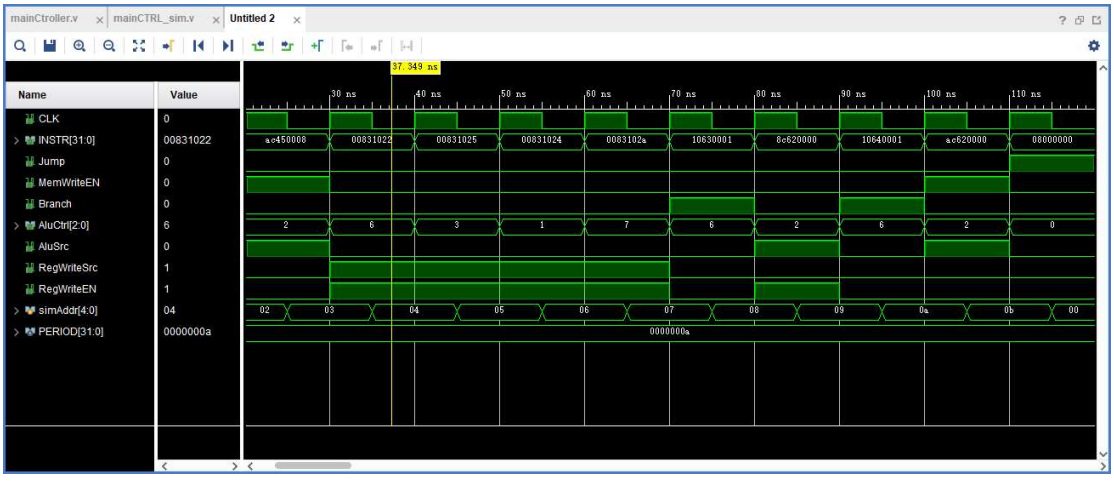

图 11 主控制器仿真

七、整体系统仿真

整体系统的仿真，仿真模块如下代码块 x 所示。该仿真模块原理相对简单，首先实例化本次实验中设计的整个微控制器，送入 CLK 时钟信号和 RST 复位信号。接着在 always 语句块中产生时钟激励，和 initial 语句中在初始情况下产生的复位信号。

```
module systemSimulation(
    output reg CLK,
    output reg RST
```

```
    );

    //实例化本实验设计的整个微处理器
    SYSTEM wholeSysModule(CLK,RST);

    //时钟激励生成
    parameter PERIOD = 10;
    always begin
      CLK = 1'b0;
      #(PERIOD/2) CLK = 1'b1;
      #(PERIOD/2);
    end
    //复位信号生成
    initial begin
    RST = 1;
    #10 RST = 0;
    end

endmodule
```

模块 18 全系统仿真模块

模块仿真结果如下图 x 所示。通过在 MARS 中仿真可以知道，本次实验中，对数据存储器中的地址为 16 的存储单元进行了操作，还有通用寄存器中的 2 号与 4 号寄存器进行了操作，所以下图波形中在展示了整体存储器和整体通用寄存器的情况下，还分别独立取出了数据存储器的 16 地址存储单元（红色框）和通用寄存器的 4 号、2 号存储单元（橘色框）。并且，为了方便查看运行的准确性，图中蓝框中的信号为执行指令和 PC 信号；紫色框中的 ALU 运算结果。

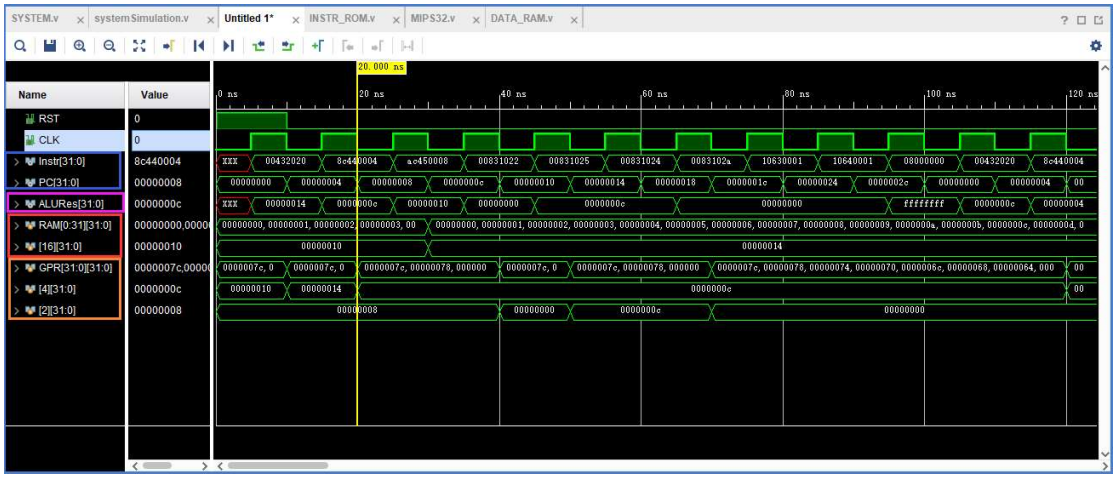

图 12 整体系统仿真结果波形图

观察上波形图，可见，测试代码中依次进行的寄存器加法、存储器读、存储器写、减法、寄存器或、寄存器与、逻辑小于、分支跳转、直接跳转功能都成功运行，并得到了准确的计算结果。在运行到测试代码的最后一句“j main”，由仿真波形信号（100ns 处）可见，PC 信号成功地跳回了 0，程序循环运行。综上所述，本次实验成功，结果准确。

八、实验总结

本次实验是一次对课内知识和动手技能都有很高要求的实验。在这次实验中，我扎实地了解了微处理器的工作原理、设计逻辑和设计方法等。这次实验，收获最大的就是动手实践的过程，在参考理解 MIPS 逻辑款图的基础上，我采用了分模块的思想，把一个大问题，分解为一个个小的问题去实现，在完成子模块的设计、验证后，将这些小模块正确地组装起来，检查是否运行正确，让我的 Verilog 编写能力和数字波形理解能力都有了很大的提升。

同时这次的实验，也是一次很好锻炼耐心、磨砺科学钻研精神的机会，在将小模块组装起 MIPS 处理器核心的过程中，出现了一两次由于模块接线顺序错误造成的错误，在发现程序运行结果和预期不相符的情况下，回头检查代码，并且不断在行为仿真波形中添加怀疑出错的端口，来逐级检查错误。

综合来说，本次实验是很好的锻炼能力、巩固知识、磨炼意志的过程，并且取得了准确的预期的实验结果。