实验二：简单指令集MIPS单周期微处理器设计

专业班级：**提高2201班**

姓名：　　**王翎羽**

学号：　　**U202213806**

实验名称

简单指令集MIPS单周期微处理器设计

实验目的

1. 了解微处理器的基本结构
2. 学会设计简单的微处理器
3. 了解软件控制硬件工作的基本原理

实验仪器

Vivado2023.02 、Mars MIPS汇编编译器

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。数据存储器的地址都是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
```

另：扩展实现以下指令：

- 1）addi，ori；
- 2）lb，lbu，lh，lhu；
- 3）bne；bltz；bgez；
- 4）jal，jr；

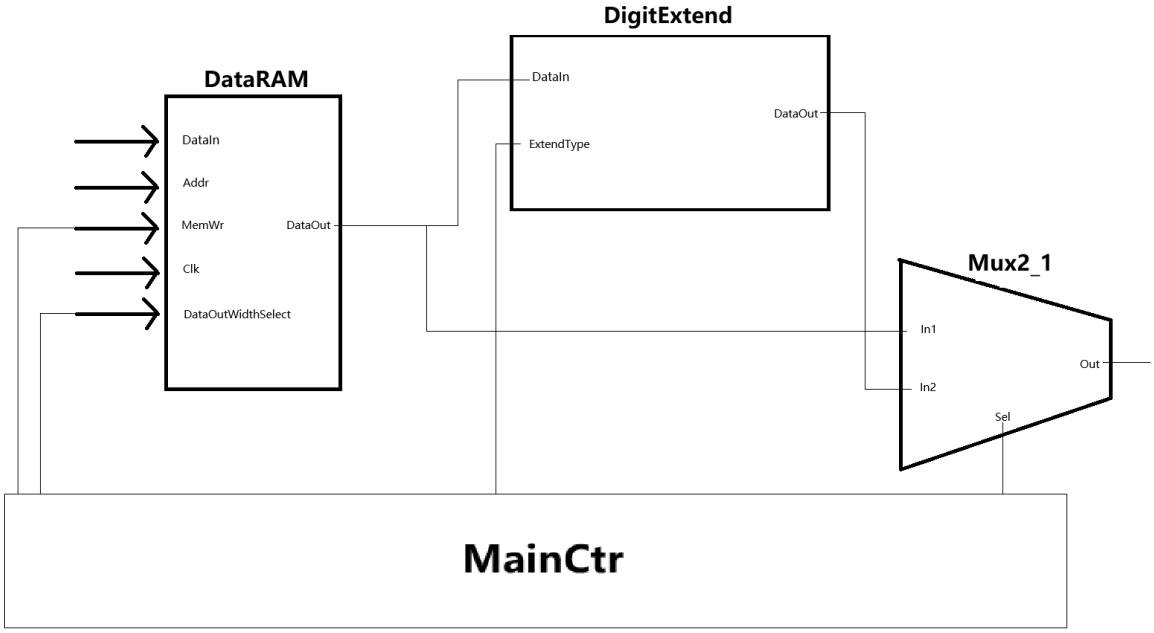
并在汇编程序中添加相应指令仿真验证该指令执行是否正确。

实验过程

扩展原理

对于扩展指令1)，仅需修改`MainCtr.v`文件中控制变量，将R型指令控制信号修改为I型指令的控制信号。

对于扩展指令2)，我修改了`DataRAM`模块，使其可以根据控制信号`DataOutWidthSelect`完成输出位数的自定义，并且添加了一个新的模块`DigitExtend`，该模块调用符号扩展模块`SignalExtend`和实现无符号扩展，根据输入的`ExtendType`来确定进行8位/16位的符号/无符号扩展。如下图所示。



生成机器码

根据上述要求，我选择扩展1)和2)指令，并将其添加进汇编指令代码中，如下：

（请忽略扩展 3)和 4)部分的代码）

```
main:
    add $4,$2,$3  #0
    addi $5,$2,10 # addi
    ori $6,$2,10  # ori
    lb $7,0($2)   # lb
    lbu $8,0($2)  # lbu
    lh $9,0($2)   #lh
    lhu $10,0($2) # lhu
    lw $4,4($2)   #1
    sw $2,8($2)   #2
    sub $2,$4,$3  #3
    or $2,$4,$3   #4
    and $2,$4,$3  #5
    slt $2,$4,$3  #6
    bne $4,$3,exit # bne
    bltz $4,exit  # bltz
    bgez $4,exit  # bgez
    j main        #8
    jal foo       # jal
    jr $31        # jr
exit:
    lw $2,0($3)   #9
    j main        #10
foo: add $2,$4,$5
```

用mars软件转换机器码如下：

```
00432020
2045000a
3446000a
80470000
90480000
8c440004
84490000
944a0000
ac420008
00831022
00831025
00831024
0083102a
14830005
04800004
04810003
08000000
0c000015
03e00008
8c620000
08000000
```

```
00851020
```

源代码

见附件。实现MIPSCPU各组件部分与线上实验相同。在此仅放出扩展部分需要修改部分的代码。

Controller.v

```
`timescale 1ns / 1ps
module Controller (OpCode, Funct, J, B, RegDst, RegWr, ALUSrc, MemWr, Mem2Reg,
ALUCtr, DataOutWidthSelect,ExtendType, OutSelect);
//控制器
input [5:0] OpCode, Funct;
output J, B, RegDst, RegWr, ALUSrc, MemWr, Mem2Reg, OutSelect;
output [1:0] DataOutWidthSelect, ExtendType;
output [3:0] ALUCtr;
wire [3:0] ALUOp;

MainCtr U0(OpCode,J, B, RegDst, RegWr, ALUSrc, MemWr, Mem2Reg, ALUOp,
DataOutWidthSelect, ExtendType, OutSelect);//主控制器

ALUControl U1(ALUOp, Funct, ALUCtr); //ALU 控制器第二级译码， 产生 ALUCtr[3:0]
endmodule
```

MainCtrl.v

```
`timescale 1ns / 1ps

module MainCtr(
    OpCode,
    J,
    B,
    RegDst,
    RegWr,
    ALUSrc,
    MemWr,
    Mem2Reg,
    ALUOp,
    DataOutWidthSelect,
    ExtendType,
    OutSelect
    );
    input [5:0] OpCode;
    output [3:0] ALUOp;
    output RegDst;
    output RegWr;
    output ALUSrc;
    output MemWr;
    output B;
    output J;
    output Mem2Reg;
    output [1:0] DataOutWidthSelect;
```

```
    output [1:0] ExtendType;
    output OutSelect;


    reg [15:0] outputtemp;
    assign OutSelect = outputtemp[15];
    assign ExtendType = outputtemp[14:13];
    assign DataOutWidthSelect = outputtemp[12:11];
    assign J = outputtemp[10];
    assign B = outputtemp[9];
    assign RegDst = outputtemp[8];
    assign RegWr = outputtemp[7];
    assign ALUSrc = outputtemp[6];
    assign MemWr = outputtemp[5];
    assign Mem2Reg = outputtemp[4];
    assign ALUOp = outputtemp[3:0];
   always @(OpCode)
    case(OpCode)
        6'b000000:outputtemp = 16'b0xx110011_000_0010; // R
        6'b100011:outputtemp = 16'b0xx110001_101_0000; // lw
        6'b101011:outputtemp = 16'b0xx1100x0_11x_0000; //sw
        6'b000100:outputtemp = 16'b0xx1101x0_00x_0001; //beq
        6'b000010:outputtemp = 16'b0xx1110x0_x0x_xxxx; //j
        6'b001000:outputtemp = 16'b0xx110001_100_0010; //addi
        6'b001101:outputtemp = 16'b0xx110001_100_0011; //ori
        6'b100000:outputtemp = 16'b100000001_101_0000; //lb
        6'b100100:outputtemp = 16'b101000001_101_0000; //lbu
        6'b100001:outputtemp = 16'b110010001_101_0000; //lh
        6'b100101:outputtemp = 16'b111010001_101_0000; //lhu
        default: outputtemp = 16'b0000000000000000;
    endcase
endmodule
```

ALUController.v

```
`timescale 1ns / 1ps
module ALUControl (ALUOp, Funct, ALUCtr);// ALU 控制器 ALUControl
input [3:0] ALUOp;
input [5:0] Funct;
output reg [3:0] ALUCtr;
always @(*) //对功能码 Funct[5:0]和 ALUOp[3：0]进行译码
casex({ALUOp, Funct})
10'b0000xxxxxx:ALUCtr=4'b0010;
10'b0000xxxxxx:ALUCtr=4'b0010;
10'b0001xxxxxx:ALUCtr=4'b0110;
10'b0010100000:ALUCtr=4'b0010;
10'b0010100010:ALUCtr=4'b0110;
10'b0010100100:ALUCtr=4'b0000;
10'b0010100101:ALUCtr=4'b0001;
10'b0010101010:ALUCtr=4'b0111;
10'b0011xxxxxx:ALUCtr=4'b0001;
endcase
endmodule
```

DataRAM.v

```
`timescale 1ns / 1ps
module DataRAM(
    Addr, DataIn, MemWR, Clk, DataOut, DataOutWidthSelect
);
    parameter n = 5, m = 32;  // n: 地址位宽, m: 数据位宽
    reg [m-1:0] regs[2**n-1:0]; // 数据存储器

    input [n-1:0] Addr;
    input [m-1:0] DataIn;
    input MemWR, Clk;
    input [1:0] DataOutWidthSelect;  // 用于选择输出数据的位宽的输入
    output reg [m-1:0] DataOut;  // 保持最大位宽

    // 根据DataOutWidthSelect选择输出数据的位宽
    wire [m-1:0] selectedDataOut;
    assign selectedDataOut = (DataOutWidthSelect == 2'b00) ? {24'b0, regs[Addr]
[7:0]} :  // 8位输出
                             (DataOutWidthSelect == 2'b01) ? {16'b0, regs[Addr]
[15:0]} : // 16位输出
                             regs[Addr]; // 默认32位输出

    always @ (posedge Clk)
    begin
        if (MemWR)
            regs[Addr] <= DataIn;  // 写数据操作
    end

    // 同步读取数据
    always @ (posedge Clk)
    begin
        DataOut <= selectedDataOut;
    end

endmodule
```

DigitExtend.v

```
`timescale 1ns / 1ps

module DigitExtend(
    input [15:0] DataIn,  // 输入最多只有16位
    input [1:0] ExtendType,  // ExtendType: 00 for lb, 01 for lbu, 10 for lh, 11
for lhu
    output [31:0] DataOut
);
    wire [7:0] byte_data = DataIn[7:0];  // 最低8位用于字节操作
    wire [15:0] half_data = DataIn[15:0]; // 低16位用于半字操作

    // 符号扩展的输出
    wire [31:0] sign_extend_byte;
    wire [31:0] sign_extend_half;
```

```
    // 实例化 SignedExtend 模型
    SignedExtend #(.n(8), .m(32)) se_byte (.In(byte_data),
.Out(sign_extend_byte));
    SignedExtend #(.n(16), .m(32)) se_half (.In(half_data),
.Out(sign_extend_half));

    // 根据ExtendType选择正确的输出
    reg [31:0] temp_output;
    always @(*) begin
        case (ExtendType)
            2'b00: temp_output = sign_extend_byte;  // lb
            2'b01: temp_output = {24'b0, byte_data}; // lbu
            2'b10: temp_output = sign_extend_half;  // lh
            2'b11: temp_output = {16'b0, half_data}; // lhu
            default: temp_output = 32'b0;
        endcase
    end

    // 输出
    assign DataOut = temp_output;

endmodule
```

MIPS_CPU.v

```
`timescale 1ns /1ps
module MIPSCPU (
input Clk,
input Reset
);
wire [31:0]

CurrentPC,SequencePC,JumpPC,BranchPC,TempPC,Boffset,Instr,MemData,Data2Reg,Res,Rs
Data,RtData,ALUIn2,SignExtended,DataOut,ExtendData;
wire [1:0] PCsel;
wire [3:0] ALUOp;
wire [4:0] WrAddr;
wire J,B,RegDst,ALUSrc,RegWr,MemWr,Mem2Reg,Zero,BZero;
wire [27:0] JumpTarget;
wire [3:0] ALUCtr;
wire [1:0] DataOutWidthSelect;
wire [1:0] ExtendType;
wire OutSelect;

PC U0_PC(TempPC, Clk, Reset, CurrentPC);
InstrROM ROM_U0(CurrentPC[6:2],~Clk,Instr);
defparam ROM_U0.n=5;
Controller U2_Contr(Instr[31:26],Instr[5:0], J, B, RegDst, RegWr, ALUSrc, MemWr,
Mem2Reg ,ALUCtr, DataOutWidthSelect,ExtendType, OutSelect);
Mux2_1 U3(Instr[20:16], Instr[15:11],RegDst,WrAddr);
defparam U3.n=5;
LeftShift U4(Instr[25:0], JumpTarget);
defparam U4.n=26,U4.m=28,U4.x=2;
```

```
Adder U5(CurrentPC,'h4,SequencePC);
defparam U5.n=32;
Concat U6(SequencePC[31:28], JumpTarget, JumpPC);
defparam U6.n=4,U6.m=28;
Adder U7(SequencePC,Boffset,BranchPC);
defparam U7.n=32;
Mux3_1 U8(SequencePC, BranchPC,JumpPC,PCsel, TempPC);
defparam U8.n=32;
LeftShift U9(SignExtended, Boffset);
defparam U9.n=32,U9.m=32,U9.x=2;
RegFile RegFile_U1(Instr[25:21],Instr[20:16], WrAddr,Data2Reg, RegWr,~Clk,
RsData, RtData);
SignedExtend U11(Instr[15:0], SignExtended);
defparam U11.n=16,U11.m=32;
Mux2_1 U12(RtData, SignExtended,ALUSrc,ALUIn2);
defparam U12.n=32;
ALU U13(RsData,ALUIn2,ALUCtr,Res,Zero);

Concat U15(J, BZero, PCsel);
defparam U15.n=1,U15.m=1;
DataRAM RAM_U3(Res[6:2],RtData,MemWr,Clk, MemData, DataOutWidthSelect);
defparam RAM_U3.n=5,RAM_U3.m=32;
DigitExtend U16(MemData, ExtendType, DataOut);
Mux2_1 U17(MemData, DataOut, OutSelect, ExtendData);
defparam U17.n=32;
Mux2_1 U14(Res, ExtendData,Mem2Reg,Data2Reg);
defparam U14.n=32;
and(BZero,B,Zero);
endmodule
```

代码分析

在顶层文件MIPS_CPU.v中，我们定义了各种数据通路，并对各模块都进行了实例化，

按照要求：

- 读取指令在时钟上升沿

```
InstrROM ROM_U0(CurrentPC[6:2],Clk,Instr);
defparam ROM_U0.n=5;
```

- 指令指针修改在时钟下降沿

```
PC U0_PC(TempPC, ~Clk, Reset, CurrentPC);
```

- 寄存器文件写入在时钟下降沿

```
RegFile RegFile_U1(Instr[25:21],Instr[20:16], WrAddr,Data2Reg, RegWr,~Clk,
RsData, RtData);
always @(posedge Clk)
    if(RegWr)
        regs[WrAddr]=DataIn;
```

- 数据存储器数据写入都在时钟下降沿

```
DataRAM RAM_U3(Res[6:2],RtData,MemWr,~Clk, MemData, DataOutWidthSelect);
always @ (posedge Clk)
begin
    if (MemWR)
        regs[Addr] <= DataIn;  // 写数据操作
end

// 同步读取数据
always @ (posedge Clk)
begin
    DataOut <= selectedDataOut;
end
```

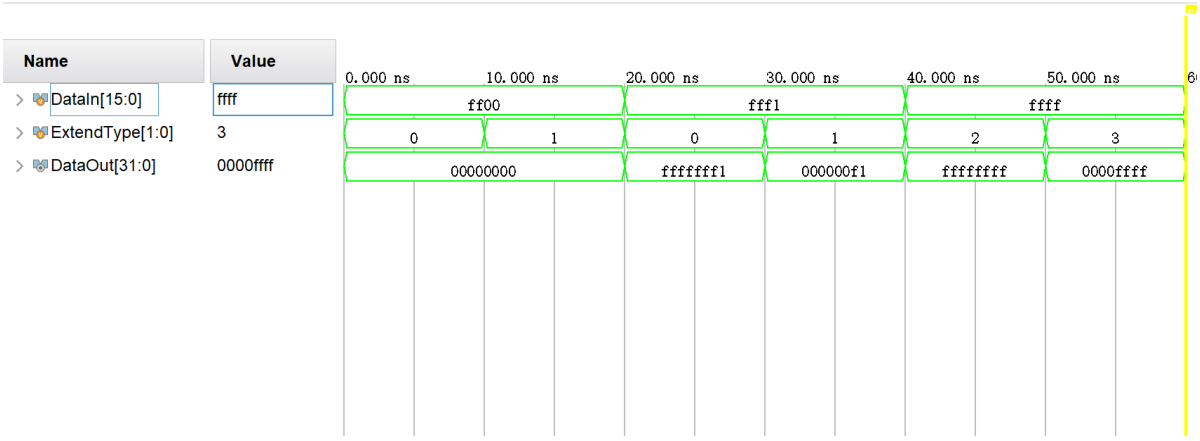
仿真代码

DigitExtendSim.v

```
`timescale 1ns / 1ps

module DigitExtend_tb;

    reg [15:0] DataIn;
    reg [1:0] ExtendType;
    wire [31:0] DataOut;

    DigitExtend dut(
        .DataIn(DataIn),
        .ExtendType(ExtendType),
        .DataOut(DataOut)
    );

    initial begin
        // Test lb operation
        DataIn = 16'hFF00;
        ExtendType = 2'b00;
        #10;

        // Test lbu operation
        DataIn = 16'hFF00;
        ExtendType = 2'b01;
        #10;

        DataIn = 16'hFFF1;
        ExtendType = 2'b00;
        #10;

        DataIn = 16'hFFF1;
        ExtendType = 2'b01;
        #10;

        // Test lh operation
        DataIn = 16'hFFFF;
        ExtendType = 2'b10;
        #10;
```

```
        // Test lhu operation
        DataIn = 16'hFFFF;
        ExtendType = 2'b11;
        #10;

        // End of simulation
        $finish;
    end

endmodule
```

得到仿真波形：

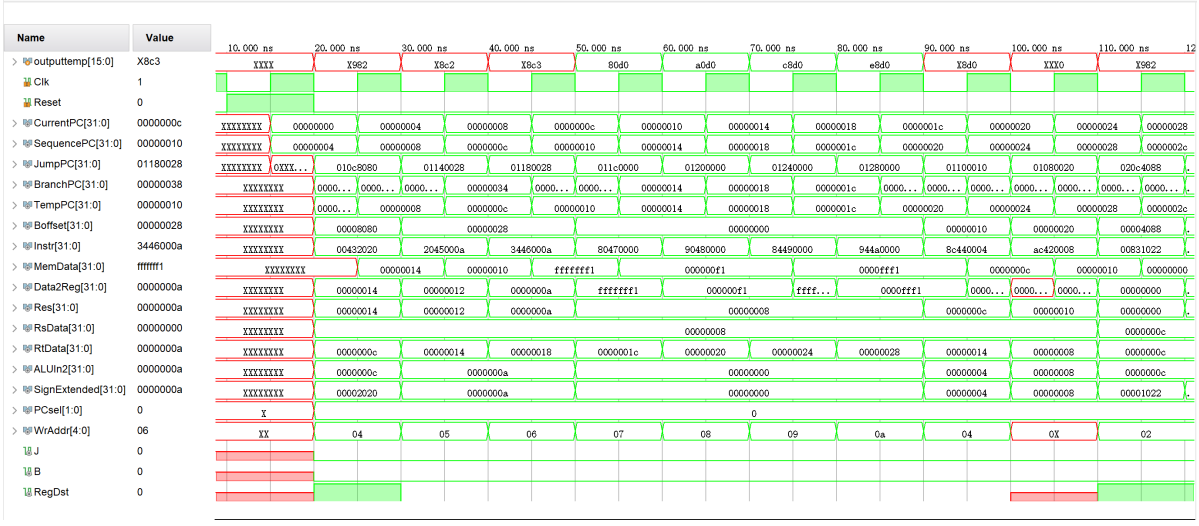


MIPSCPUsim.v

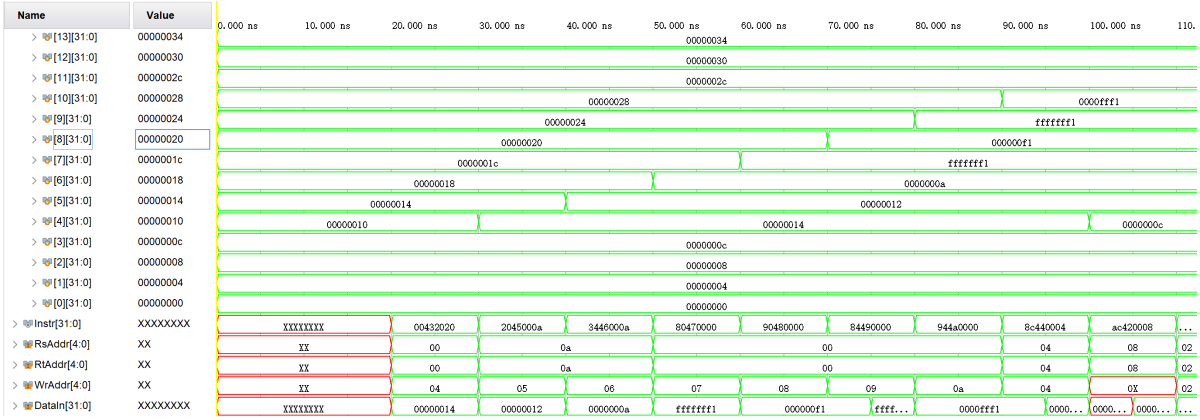
```
`timescale 1ns / 1ps

module mipscpusim();
    reg Clk, Reset;
    MIPSCPU uut(Clk, Reset);

    // 定义周期参数
    parameter period = 10;

    // 时钟信号生成
    always begin
        Clk = 1'b0;
        #(period / 2) Clk = 1'b1;
        #(period / 2);
    end

    // 变量定义
    integer i;
    integer j;

    // 初始配置
    initial begin
        // 初始化寄存器和内存
        for (i = 0; i < 32; i = i + 1) begin
            uut.RegFile_U1.regs[i] = i * 4;     // 初始化寄存器文件
            uut.RAM_U3.regs[i] = i * 4;         // 初始化数据RAM
```

```
        end

        // 特殊值设置
        uut.RAM_U3.regs[2] = 32'hfffffff1;

        // 读取并显示存储在ROM中的指令
        $readmemh("ld.txt", uut.ROM_U0.regs);
        for (j = 0; j <= 11; j = j + 1) begin
            $display("%b", uut.ROM_U0.regs[j]);
        end

        // Reset逻辑
        Reset = 0;
        #10;
        Reset = 1;
        #10;
        Reset = 0;
        #200;

        // 结束仿真
        $stop;
    end
endmodule
```

得到仿真波形：

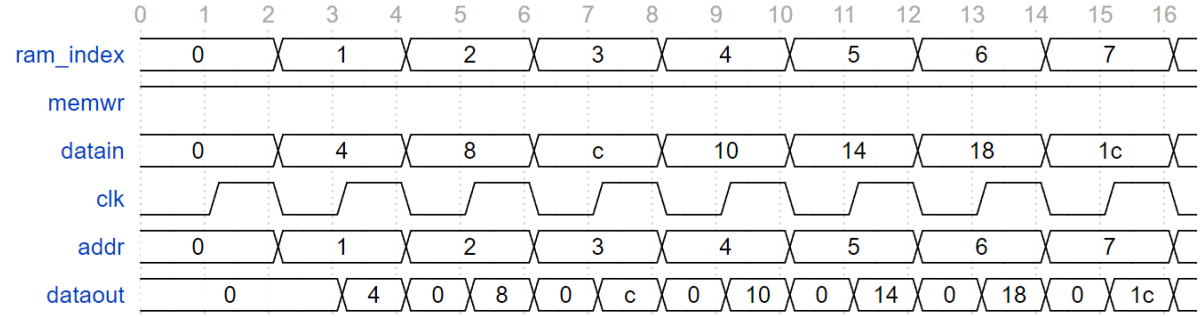


寄存器值波形：

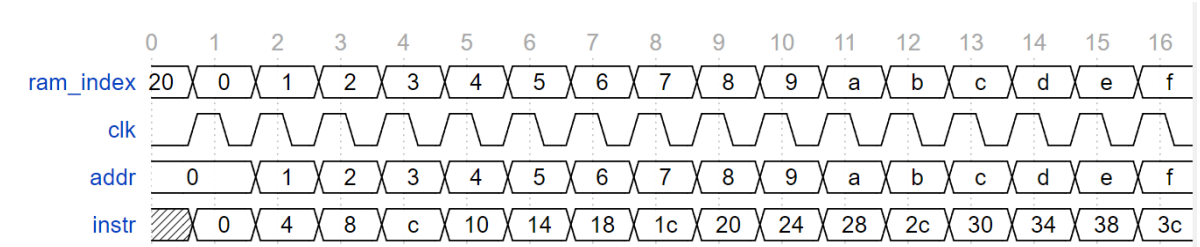


其他各模块仿真

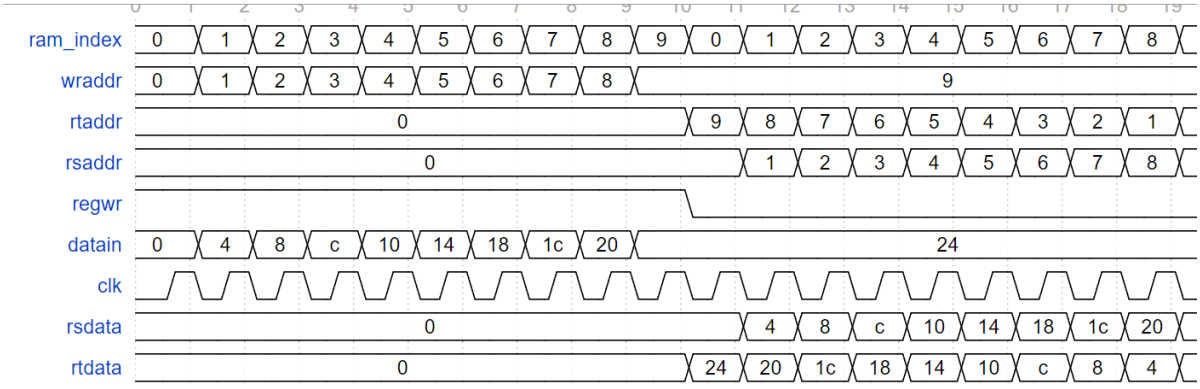
数据存储器模块



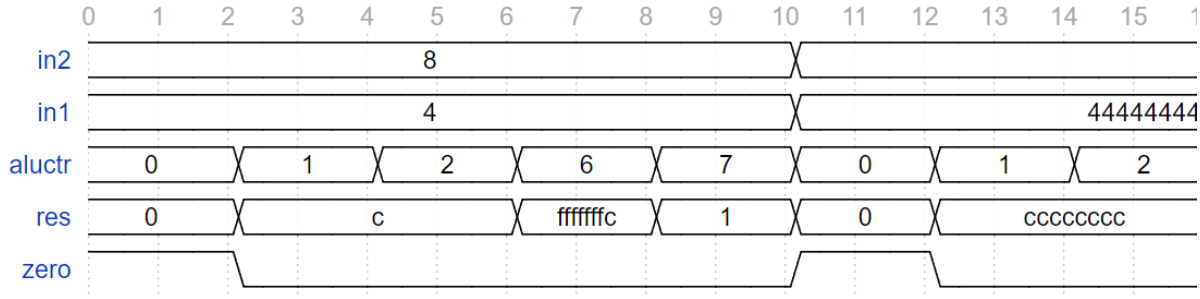
指令存储器模块



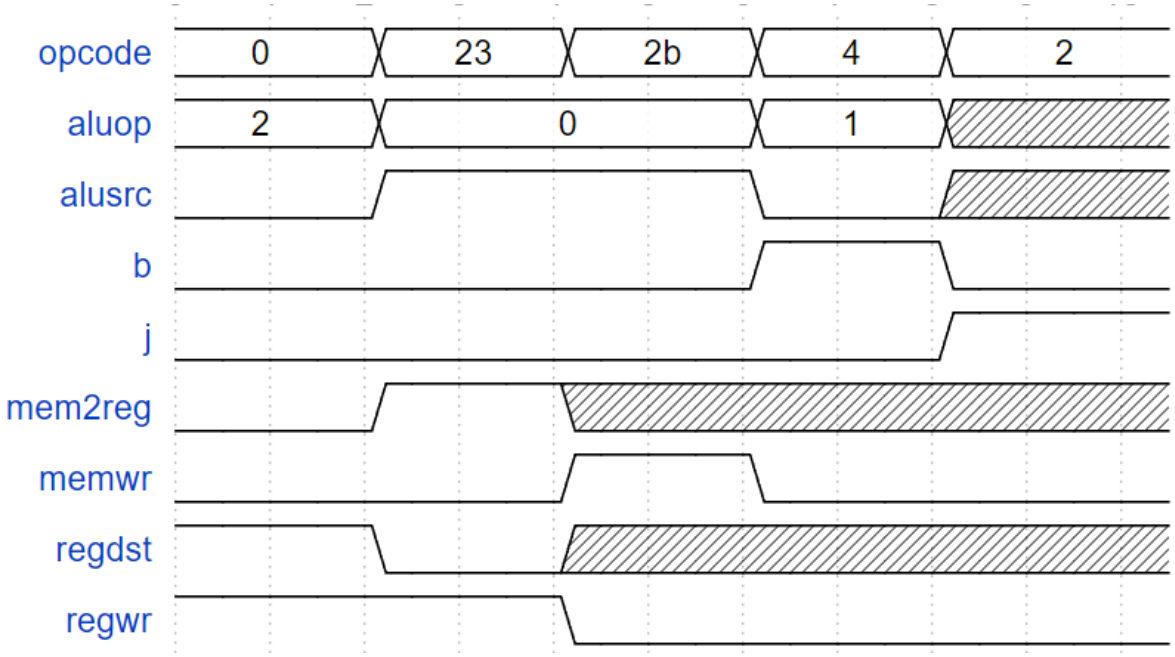
寄存器文件模块



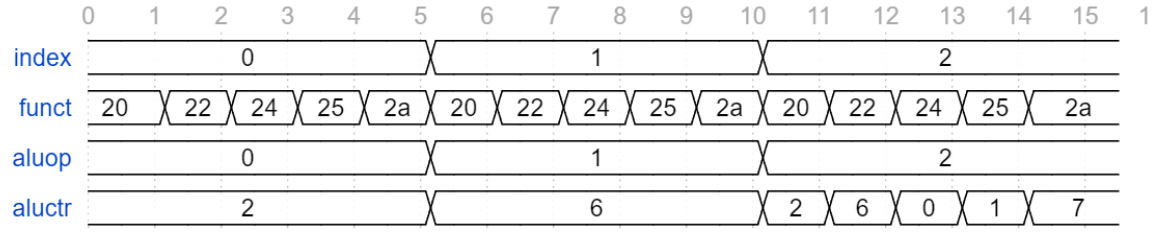
ALU模块



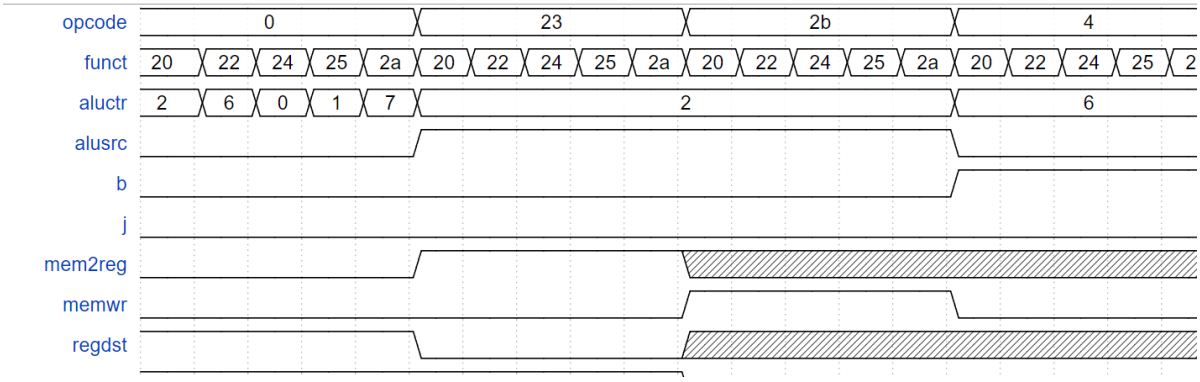
主控制器模块



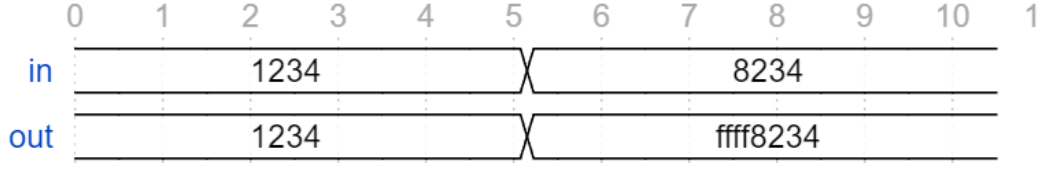
ALU控制器模块



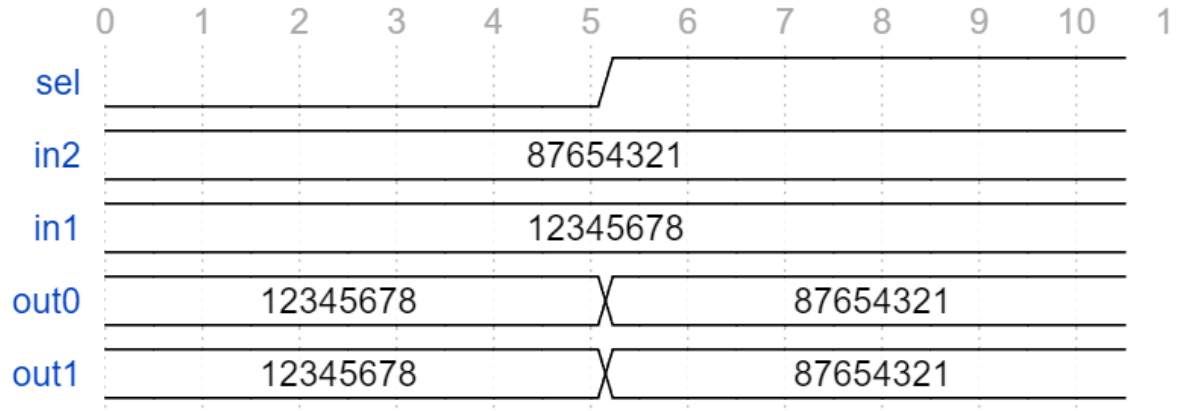
控制器模块



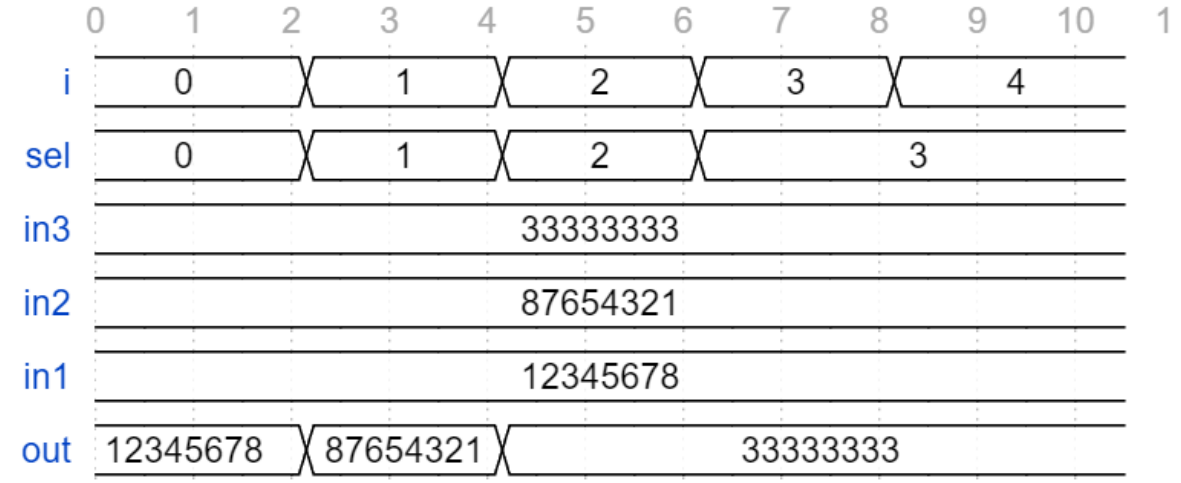
符号扩展模块



2选1多路复用器模块

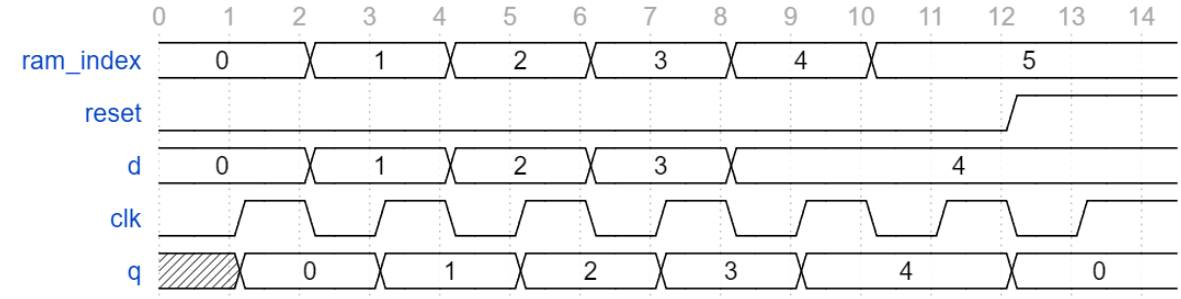


3选1多路复用器模块

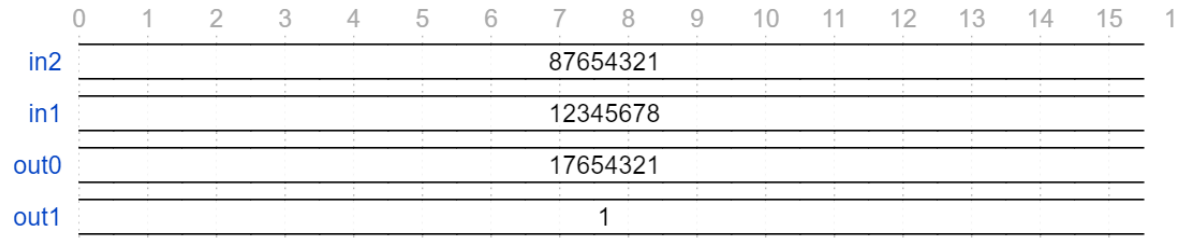


加法器

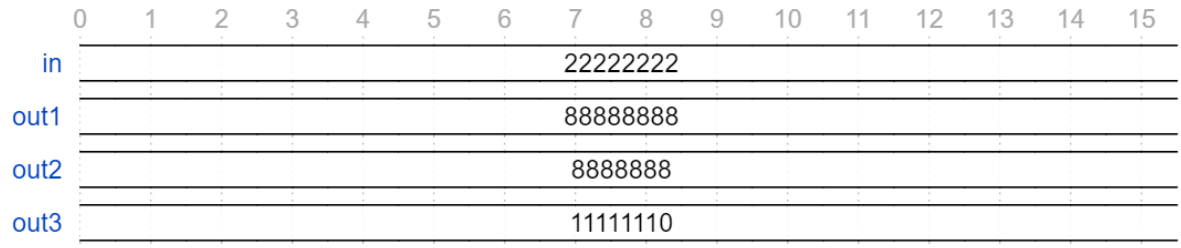
PC寄存器



位拼接器模块



左移运算器模块



顶层文件仿真结果分析

对MIPS_CPU的仿真得到如上波形，下面将分析基础功能和扩展功能：

基础功能

对于`add $4,$2,$3`，可以看到寄存器值波形中在30ns时，4号寄存器的值变为了2号寄存器的值与3号寄存器的值的和。

对于`lw $4,4($2)`,可以看到寄存器值波形中在100ns时，4号寄存器的值变为了2号寄存器的值+4对应的数据存储器（也就是3号存储器）的值。

同理，`sw $2,8($2)`,可以看到数据存储器值波形中在110ns时，4号存储器的值变为了也就是4号寄存器的值。

之后的 sub,or,and,slt 都为 R 型指令， 由于4 与3 对应的值相等，都为 0xc，他们相减后为 0，相与、相或之后的结果相同，左移之后右边补 0，所以最终得到的是 0.

对于原有的 beq 指令，在原波形中可看到4,3 寄存器对应的值相等，Zero 输出的值为1， 跳转到exit 指令处， 再往下执行一条 j main ， 之后回到第 0 条指令。

扩展功能

`addi $5,$2,10`和`ori $6,$2,10`

对应机器码为：

`2045000a`和`3446000a`

在波形窗口中，我们可以看到，35ns时，获取指令`addi $5,$2,10`，将2号寄存器中的值`0x0000_0008`和立即数`10`（十进制）相加得到值`0x0000_0012`；40ns时，值`0x0000_0012`写入寄存器$5中，**执行正确**；

40ns时，获取指令`ori $6,$2,10`，将2号寄存器中的值`0x0000_0008`和立即数`0x0000_000a`进行或运算，得到值`0x0000_000a`；50ns时，值`0x0000_000a`写入寄存器$6中，**执行正确。**

接下来分析扩展功能：

`lb $7,0($2) #lb`、`lbu $8,0($2) #lbu`、`lh $9,0($2) #lh`和`lhu $10,0($2) #lhu`

对应的机器码为:

`80470000` `90480000` `8c440004` `84490000`

在波形窗口中，我们可以看到，50ns时，获取指令`lb $7,0($2)`，将2号存储器的值`0xffff_fff1`的低八位经过符号扩展得到`0xffff_fff1`，在60ns时写入寄存器$7中，**执行正确**；

60ns时，获取指令`lbu $8,0($2)`，将2号存储器的值`0xffff_fff1`的低八位经过无符号扩展得到`0x0000_00f1`，在70ns时写入寄存器$8中，**执行正确**；

70ns时，获取指令`lh $9,0($2)`，将2号存储器的值`0xffff_fff1`的低十六位经过符号扩展得到`0xffff_fff1`，在80ns时写入寄存器$9中，**执行正确**；

80ns时，获取指令`lhu $10,0($2)`，将2号存储器的值`0xffff_fff1`的低十六位经过无符号扩展得到`0x0000_fff1`，在90ns时写入寄存器$10中，**执行正确**；

实验小结

这次实验是基于线上实验微处理器分模块设计任务完成的，认真完成线上实验可以为本次实验打下基础，以及在理论课上的MIPS指令扩展的任务为本次扩展实验提供了理论基础。为了完成本次实验，我认真复习了第二和第三章的有关知识，亲自动手完成相关代码，收获巨大。