实验二：简单指令集单周期MIPS微处理器设计

班级：**提高2301班**
姓名：**张禹阳**
学号：**U202314270**

实验目的

本次实验目的为：

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

实验任务

本次实验任务为：

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：

- 在时钟上升沿读出指令
- 在时钟下降沿进行指令的修改、寄存器文件的写入、数据存储器的数据写入

完成

- 完整代码设计
- 各模块完整功能仿真
- 整体仿真
- 验证所有指令执行情况

假定：

- 所有通用寄存器复位时取值都为各自寄存器编号乘4
- PC寄存器初始值为0
- 数据存储器和指令存储器的容量大小为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

仿真以下MIPS汇编程序段的执行流程：

```
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
```

另需扩展实现：

- `addi, ori`
- `lb, lbu, lh, lhu`
- `bne`
- `jal, jr`
- `sb, sh`
  并在汇编程序中添加相应的指令仿真验证指令执行是否正确

注意仿真时，需仿真各种场景下指令是否都能正确执行以验证设计的正确性：

- 针对 `addi, ori` 指令的执行要求验证如下类似指令 `addi(ori) $t1, $t2, 5(-5)`

- 针对 `lb, lbu, lhb, lhu` 指令要求对读取的存储器单元初始化时包含正、负符号数
- 针对 `bne` 指令，初始化寄存器 `$t1 = 0xffffffff, $t0 = 0x1` ；依次执行 `bne $t1, $t1, label 1; bne $t1, $t0, label 1`

实验过程

指令扩展

针对第1类指令 `addi, ori`：

- 根据机器码在 `MainCtr` 中添加相应控制信号
- 由于此时对立即数进行的是无符号扩展，添加一个复用器 `Mux2_1` 用于选择最终立即数的类型，是符号扩展还是无符号扩展
- 在 `MainCtr` 中添加信号 `Imm_ExtendType` 作为该复用器的选择信号

针对第2类指令 `lb, lbu, lh, lhu`：
原 `DataRAM` 模块固定输出大小为 `32b` 的数据，故：

- 添加新模块 `DigitExtend`
- 在 `MainCtr` 模块中生成信号 `ExtendType` 控制 `DigitExtend`
- 在 `DigitExtend` 中根据 `Res` 和 `ExtendType` 对来自 `DataRAM` 的 `32b` 数据进行截取以及符号/无符号扩展

通过 Mars 转换得到的机器码可知，`lb, lbu, lh, lhu` 对应的 `OpCode` 分别为：

- 100000
- 100100
- 100001
- 100101
  则对 `MainCtr` 作修改（以 `lb` 为例）：

```
6'b100000:
    begin
        J = 0;
        B = 0;
        RegDst = 0;          // the destination is Rt
```

```
        RegWr = 1;          // enable writing
        ALUSrc = 1;         // the second operator of ALU is Imm
        MemWr = 0;
        Mem2Reg = 1;        // select data from DataRAM as input to
RegFile
        ALUOp = 2'b00;      // ALU adds Imm and RF[$Rs]
        ExtendType = 2'b00; // lb
    end
```

最终 `DigitExtend.v` 代码如下：

```
`timescale 1ns / 1ps
module DigitExtend (ByteSelect, ExtendType, DataIn, DataOut);
    input [1:0] ByteSelect;
    input [1:0] ExtendType;
    input [31:0] DataIn;
    output reg [31:0] DataOut;

    reg [7:0] byte_data;
    reg [15:0] half_data;

    always @(*) begin
        case (ByteSelect)
            2'b00:
                begin
                    byte_data = DataIn[31:24];
                    half_data = DataIn[31:16];
                end
            2'b01:
                begin
                    byte_data = DataIn[23:16];
                    half_data = DataIn[31:16];
                end
            2'b10:
                begin
                    byte_data = DataIn[15:8];
                    half_data = DataIn[15:0];
                end
            2'b11:
                begin
                    byte_data = DataIn[7:0];
                    half_data = DataIn[15:0];
                end
```

```
        endcase
    end

    always @(*) begin
        case (ExtendType)
            2'b00: DataOut = {{(24){byte_data[7]}}, byte_data};   // lb
            2'b01: DataOut = {24'b0, byte_data};                  // lbu
            2'b10: DataOut = {{(16){half_data[15]}}, half_data};  // lh
            2'b11: DataOut = {16'b0, half_data};                  // lhu
            default: DataOut = DataIn;                            // lw
        endcase
    end
endmodule
```

针对第3类指令 `bne`：

- 根据机器码在 `MainCtr` 中添加相应控制信号
- 添加一个复用器 `Mux2_1`，以 `Zero` 和 `~Zero` 作为输入，用于选择最终和 `B` 相与的数据
- 在 `MainCtr` 中添加 `BranchType` 作为该复用器的选择信号

汇编指令的机器码获取

在原汇编程序的基础上，我添加了 12 条指令用于检测 `lb, lbu, lh, lhu` 的执行情况
最终完整汇编程序如下：

```
main:
add $4, $2, $3
addi $10, $2, 4 # addi

lw $4, 4($2)
sw $5, 8($2)

# instruction extension
# lb
lb $6, 0($2)
lb $6, 1($2)
lb $6, 2($2)
lb $6, 3($2)

# lbu
```

```
lbu $7, 0($2)
lbu $7, 1($2)
lbu $7, 2($2)
lbu $7, 3($2)

# lh
lh $8, 0($2)
lh $8, 2($2)

# lhu
lhu $9, 0($2)
lhu $9, 2($2)

sub $2, $4, $3
or $2, $4, $3
ori $2, $4, 4 # ori

and $2, $4, $3
slt $2, $4, $3
beq $3, $3, equ
bne $3, $0, ine
lw $2, 0($3)

equ:
beq $3, $4, exit
sw $2, 0($3)

ine:
add $13, $11, $12

exit:
j main
```

用 Mars 转换为机器码如下：

```
00432020
8c440004
ac450008
80460000
80460001
80460002
80460003
```

```
90470000
90470001
90470002
90470003
84480000
84480002
94490000
94490002
00831022
00831025
00831024
0083102a
10630001
8c620000
10640001
ac620000
08000c00
```

然后将其保存为 `machine_instrs.txt`，至仿真时调用

源代码

所有源代码见附件。由于大部分模块相对线上实验部分未进行修改，在此仅放出针对扩展部分需要修改的代码：

`MainController.v`

```
`timescale 1ns / 1ps

module MainCtr (OpCode, J, B, RegDst, RegWr, ALUSrc, MemWr,
Mem2Reg, ExtendType, LoadType, ALUOp, Imm_ExtendType, BranchType);
    input [5:0] OpCode;
    output J;
    output B;
    output RegDst;
    output RegWr;
    output ALUSrc;
    output MemWr;
    output Mem2Reg;
    output [1:0] ExtendType; // to DigitExtend
    output LoadType;
    output [1:0] ALUOp;
    output Imm_ExtendType; // to Mux2to1 to select signed/unsigned
```

```
extended Imm
    output BranchType; // to Mux2to1 to select beq/bne

    reg [13:0] ControlSignals;
    assign J = ControlSignals[13];
    assign B = ControlSignals[12];
    assign BranchType = ControlSignals[11];
    assign RegDst = ControlSignals[10];
    assign RegWr = ControlSignals[9];
    assign ALUSrc = ControlSignals[8];
    assign MemWr = ControlSignals[7];
    assign Mem2Reg = ControlSignals[6];
    assign ALUOp = ControlSignals[5:4];
    assign ExtendType = ControlSignals[3:2];
    assign LoadType = ControlSignals[1];
    assign Imm_ExtendType = ControlSignals[0];

    always @(*) begin
        casex (OpCode)
            6'b000000: ControlSignals = 14'b0_0_0_1_1_0_0_0_10_xx_1_0;
            6'b100011: ControlSignals = 14'b0_0_0_0_1_1_0_1_00_xx_1_0;
            6'b101011: ControlSignals = 14'b0_0_0_x_0_1_1_x_00_xx_1_0;
            6'b000100: ControlSignals = 14'b0_1_0_x_0_0_0_x_01_xx_1_0;
// beq
            6'b000101: ControlSignals = 14'b0_1_1_x_0_0_0_x_01_xx_1_0;
// bne
            6'b000010: ControlSignals = 14'b1_0_0_x_0_x_0_x_xx_xx_1_0;
            6'b100000: ControlSignals = 14'b0_0_0_0_1_1_0_1_00_00_0_0;
// lb
            6'b100100: ControlSignals = 14'b0_0_0_0_1_1_0_1_00_01_0_0;
// lbu
            6'b100001: ControlSignals = 14'b0_0_0_0_1_1_0_1_00_10_0_0;
// lh
            6'b100101: ControlSignals = 14'b0_0_0_0_1_1_0_1_00_11_0_0;
// lhu
            6'b001000: ControlSignals = 14'b0_0_0_0_1_1_0_x_00_xx_1_1;
// addi
            6'b001101: ControlSignals = 14'b0_0_0_0_1_1_0_x_11_xx_1_1;
// ori
            default: ControlSignals = 14'b0_0_0_0_0_0_0_0_00_00_1_0;
        endcase
    end
endmodule
```

Controller.v

```
`timescale 1ns / 1ps

module Controller (OpCode, Funct, J, B, RegDst, RegWr, ALUSrc,
MemWr, Mem2Reg, ExtendType, LoadType, ALUCtr, Imm_ExtendType,
BranchType);
    input [5:0] OpCode;
    input [5:0] Funct;
    output J;
    output B;
    output RegDst;
    output RegWr;
    output ALUSrc;
    output MemWr;
    output Mem2Reg;
    output [1:0] ExtendType;
    output LoadType;
    output Imm_ExtendType;
    output BranchType;
    output [3:0] ALUCtr;

    wire [1:0] ALUOp;

    MainCtr MC (
        .OpCode(OpCode),
        .J(J),
        .B(B),
        .RegDst(RegDst),
        .RegWr(RegWr),
        .ALUSrc(ALUSrc),
        .MemWr(MemWr),
        .Mem2Reg(Mem2Reg),
        .ExtendType(ExtendType),
        .LoadType(LoadType),
        .ALUOp(ALUOp),
        .Imm_ExtendType(Imm_ExtendType),
        .BranchType(BranchType)
    );

    ALUControl AC (
        .ALUOp(ALUOp),
        .Funct(Funct),
```

```
        .ALUCtr(ALUCtr)
    );
endmodule
```

MIPSCPU.v

```
`timescale 1ns / 1ps

module MIPSCPU (Clk, Reset);
    input Clk;
    input Reset;

    wire [31:0] Instr;
    wire [31:0] CurrentPC;
    wire J;
    wire B;
    wire RegDst;
    wire RegWr;
    wire ALUSrc;
    wire MemWr;
    wire Mem2Reg;
    wire [3:0] ALUCtr;
    wire Zero;
    wire Zero_Final;
    wire [1:0] Sel;
    wire [31:0] RsData;
    wire [31:0] RtData;
    wire [31:0] Res;
    wire [31:0] MemDataOut;
    wire [31:0] ExtendedDataOut;
    wire [31:0] FinalDataOut;
    wire [31:0] Imm32_Signed;
    wire [31:0] Imm32_Final;
    wire [31:0] ALUSrc2;
    wire [31:0] RegDataIn;
    wire [31:0] SeqPC;
    wire [31:0] BranPC;
    wire [31:0] JumpPC;
    wire [31:0] NextPC;
    wire [31:0] Imm32_Signed_offset;
    wire [27:0] Jmp_low_order;
    wire [4:0] WrAddr;
    wire [1:0] ExtendType;
```

```
    wire LoadType;
    wire Imm_ExtendType;
    wire BranchType;

    DataRAM RAM_U3 (
        .Addr(Res[6:2]),
        .DataIn(RtData),
        .MemWr(MemWr),
        .Clk(~Clk),
        .DataOut(MemDataOut)
    );

    InstrROM ROM_U0 (
        .Addr(CurrentPC[6:2]),
        .Clk(Clk),
        .Instr(Instr)
    );

    RegFile RegFile_U1 (
        .RsAddr(Instr[25:21]),
        .RtAddr(Instr[20:16]),
        .WrAddr(WrAddr),
        .DataIn(RegDataIn),
        .RegWr(RegWr),
        .Clk(~Clk),
        .RsData(RsData),
        .RtData(RtData)
    );

    ALU A (
        .In1(RsData),
        .In2(ALUSrc2),
        .ALUCtr(ALUCtr),
        .Res(Res),
        .Zero(Zero)
    );

    Controller CTR (
        .OpCode(Instr[31:26]),
        .Funct(Instr[5:0]),
        .J(J),
        .B(B),
        .RegDst(RegDst),
        .RegWr(RegWr),
```

```
        .ALUSrc(ALUSrc),
        .MemWr(MemWr),
        .Mem2Reg(Mem2Reg),
        .ExtendType(ExtendType),
        .LoadType(LoadType),
        .ALUCtr(ALUCtr),
        .Imm_ExtendType(Imm_ExtendType),
        .BranchType(BranchType)
    );

    DigitExtend DE (
        .ByteSelect(Res[1:0]),
        .ExtendType(ExtendType),
        .DataIn(MemDataOut),
        .DataOut(ExtendedDataOut)
    );

    SignedExtend SE (
        .In(Instr[15:0]),
        .Out(Imm32_Signed)
    );

    Mux2_1 M1 (
        .In1(RtData),
        .In2(Imm32_Final),
        .sel(ALUSrc),
        .Out(ALUSrc2)
    );

    Mux2_1 M2 (
        .In1(Res),
        .In2(FinalDataOut),
        .sel(Mem2Reg),
        .Out(RegDataIn)
    );

    Mux2_1 M3 (
        .In1(ExtendedDataOut),
        .In2(MemDataOut),
        .sel(LoadType),
        .Out(FinalDataOut)
    );

    Mux2_1 M6 (
```

```
        .In1(Imm32_Signed),
        .In2({16'b0, Instr[15:0]}),
        .sel(Imm_ExtendType),
        .Out(Imm32_Final)
    );

    Mux2_1 M7 (
        .In1(Zero),
        .In2(~Zero),
        .sel(BranchType),
        .Out(Zero_Final)
    );

    Mux2_1 # (
        .n(5)
    ) M4 (
        .In1(Instr[20:16]),
        .In2(Instr[15:11]),
        .sel(RegDst),
        .Out(WrAddr)
    );

    Mux3_1 M5 (
        .In1(SeqPC),
        .In2(BranPC),
        .In3(JumpPC),
        .sel(Sel),
        .Out(NextPC)
    );

    Adder A1 (
        .In1(SeqPC),
        .In2(Imm32_Signed_offset),
        .Out(BranPC)
    );

    Adder A2 (
        .In1(32'b0100),
        .In2(CurrentPC),
        .Out(SeqPC)
    );

    PC P (
        .D(NextPC),
```

```
        .Clk(~Clk),
        .Reset(Reset),
        .Q(CurrentPC)
    );

    Concat # (
        .n(1),
        .m(1)
    ) CC1 (
        .In1(J),
        .In2(B & Zero_Final),
        .Out(Sel)
    );

    Concat # (
        .n(4),
        .m(28)
    ) CC2 (
        .In1(SeqPC[31:28]),
        .In2(Jmp_low_order),
        .Out(JumpPC)
    );

    LeftShift LS1 (
        .In(Imm32_Signed),
        .Out(Imm32_Signed_offset)
    );

    LeftShift # (
        .n(26),
        .m(28),
        .x(2)
    ) LS2 (
        .In(Instr[25:0]),
        .Out(Jmp_low_order)
    );
endmodule```

## 源代码分析

针对任务要求：
- 指令存储器在时钟上升沿读出指令

```verilog
```
```

```
InstrROM ROM_U0 (
    .Addr(CurrentPC[6:2]),
    .Clk(Clk),
    .Instr(Instr)
);

always @(posedge Clk) begin
    Instr <= regs[Addr];
end
```

- 指令指针修改在时钟下降沿

```
PC P (
    .D(NextPC),
    .Clk(~Clk),
    .Reset(Reset),
    .Q(CurrentPC)
);

always @(posedge Clk, posedge Reset) begin
    if (Reset)
        Q = 32'b0;
    else
        Q <= D;
end
```

- 寄存器文件写入在时钟下降沿

```
RegFile RegFile_U1 (
    .RsAddr(Instr[25:21]),
    .RtAddr(Instr[20:16]),
    .WrAddr(WrAddr),
    .DataIn(RegDataIn),
    .RegWr(RegWr),
    .Clk(~Clk),
    .RsData(RsData),
    .RtData(RtData)
);

always @(posedge Clk) begin
    if (RegWr)
```

```
        regs[WrAddr] <= DataIn;
end
```

- 数据存储器数据写入在时钟下降沿

```
DataRAM RAM_U3 (
    .Addr(Res[6:2]),
    .DataIn(RtData),
    .MemWr(MemWr),
    .Clk(~Clk),
    .DataOut(MemDataOut)
);

always @(posedge Clk) begin
    if (MemWr)
        regs[Addr] <= DataIn;
end
```

仿真代码

由于大部分模块相对线上实验部分未进行修改，在此仅放出针对扩展部分需要重新设计的仿真文件：

DigitExtend_sim.v

```
`timescale 1ns/1ps
module DigitExtend_tb;
    reg [1:0] ByteSelect;
    reg [1:0] ExtendType;
    reg [31:0] DataIn;
    wire [31:0] DataOut;
    integer i;

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

    initial begin
        DataIn = 32'h123489ab;
```

```
        // Test lb
        ExtendType = 2'b00;
        for (i=0; i<4; i=i+1) begin
            ByteSelect = i[1:0];
            #10;
        end

        // Test lbu
        ExtendType = 2'b01;
        for (i=0; i<4; i=i+1) begin
            ByteSelect = i[1:0];
            #10;
        end

        // Test lh
        ExtendType = 2'b10;
        for (i=0; i<2; i=i+1) begin
            ByteSelect = {i[0], 1'b0};
            #10;
        end

        // Test lhu (unsigned halfword extend)
        ExtendType = 2'b11;
        for (i=0; i<2; i=i+1) begin
            ByteSelect = {i[0], 1'b0};
            #10;
        end

        $finish;
    end

endmodule
```

Controller_sim.v

```
`timescale 1ns/1ps
module Controller_tb;
    reg [5:0] OpCode;
    reg [5:0] Funct;
    wire J;
    wire B;
    wire RegDst;
    wire RegWr;
```

```
    wire ALUSrc;
    wire MemWr;
    wire Mem2Reg;
    wire [1:0] ExtendType;
    wire LoadType;
    wire [3:0] ALUCtr;

    reg [5:0] test_opcodes [8:0];
    reg [5:0] test_functs [4:0];
    integer i, j;

    Controller ctrut (
        .OpCode(OpCode),
        .Funct(Funct),
        .J(J),
        .B(B),
        .RegDst(RegDst),
        .RegWr(RegWr),
        .ALUSrc(ALUSrc),
        .MemWr(MemWr),
        .Mem2Reg(Mem2Reg),
        .ExtendType(ExtendType),
        .LoadType(LoadType),
        .ALUCtr(ALUCtr)
    );

    initial begin
        test_opcodes[0] = 6'b000000;
        test_opcodes[1] = 6'b100011;
        test_opcodes[2] = 6'b101011;
        test_opcodes[3] = 6'b000100;
        test_opcodes[4] = 6'b000010;
        test_opcodes[5] = 6'b100000;
        test_opcodes[6] = 6'b100100;
        test_opcodes[7] = 6'b100001;
        test_opcodes[8] = 6'b100101;

        test_functs[0] = 6'b100000;
        test_functs[1] = 6'b100010;
        test_functs[2] = 6'b100100;
        test_functs[3] = 6'b100101;
        test_functs[4] = 6'b101010;

        for (i = 0; i < 9; i = i + 1) begin
```

```
            OpCode = test_opcodes[i];
            for (j = 0; j < 5; j = j + 1) begin
                Funct = test_functs[j];
                #1;
            end
        end

        $finish;
    end

endmodule
```

MIPSCPU_sim.v

```
`timescale 1ns/1ps

module MIPSCPU_tb;
    reg Clk, Reset;
    MIPSCPU uut (
        .Clk(Clk),
        .Reset(Reset)
    );

    parameter period = 10;

    always begin
        Clk = 1'b0;
        #(period / 2) Clk = 1'b1;
        #(period / 2);
    end

    integer i, j;
    initial begin
        for (i = 0; i < 32; i = i + 1) begin
            uut.RegFile_U1.regs[i] = i * 4;
            uut.RAM_U3.regs[i] = i * 4;
        end

        uut.RAM_U3.regs[2] = 32'h123489ab;      //
00010010_00110100_10001001_10101011

        $readmemh("machine_instrs.txt", uut.ROM_U0.regs);
        for (j = 0; j < 28; j = j + 1) begin
```

```
            $display("%b", uut.ROM_U0.regs[j]);
        end

        Reset = 1'b0;
        #10;
        Reset = 1'b1;
        #10;
        Reset = 1'b0;
        #220;

        $stop;
    end
endmodule
```

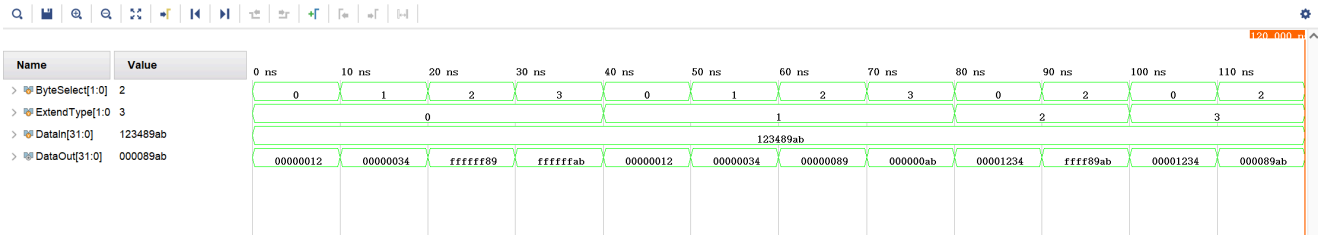
顶层文件仿真结果分析

- `add $4, $2, $3` 20ns 时，`RegFile` 中4号寄存器的值为 0x00000014 = 0×00000008 + 0×0000000c，执行正确
- `lw $4, 4($2)` 30ns时，`RegFile` 中4号寄存器的值为 0x0000000c，为 `DataRAM` 中3号寄存器的值，执行正确
- `sw $5, 8($2)` 40ns时，`DataRAM` 中4号寄存器的值为 0x00000014，为 `RegFile` 中5号寄存器的值，执行正确
- `lb $6, 0($2)` 50ns时，`RegFile` 中6号寄存器的值为 0x00000012，为 `DataRAM` 中2号寄存器的第一个字节的值 12 作符号扩展的结果，执行正确
- `lb $6, 1($2)` 60ns时，`RegFile` 中6号寄存器的值为 0x00000034，为 `DataRAM` 中2号寄存器的第二个字节的值 34 作符号扩展的结果，执行正确
- `lb $6, 0($2)` 70ns时，`RegFile` 中6号寄存器的值为 0xffffff89，为 `DataRAM` 中2号寄存器的第三个字节的值 89 作符号扩展的结果，执行正确
- `lb $6, 0($2)` 80ns时，`RegFile` 中6号寄存器的值为 0xffffffab，为 `DataRAM` 中2号寄存器的第四个字节的值 ab 作符号扩展的结果，执行正确
- `lbu $7, 0($2)` 90ns时，`RegFile` 中7号寄存器的值为 0x00000012，为 `DataRAM` 中2号寄存器的第一个字节的值 12 作无符号扩展的结果，执行正确
- `lbu $7, 1($2)` 100ns时，`RegFile` 中7号寄存器的值为 0x00000034，为 `DataRAM` 中2号寄存器的第二个字节的值 34 作无符号扩展的结果，执行正确
- `lbu $7, 3($2)` 110ns时，`RegFile` 中7号寄存器的值为 0x00000089，为 `DataRAM` 中2号寄存器的第三个字节的值 89 作无符号扩展的结果，执行正确
- `lbu $7, 0($2)` 120ns时，`RegFile` 中7号寄存器的值为 0x000000ab，为 `DataRAM` 中2号寄存器的第四个字节的值 ab 作无符号扩展的结果，执行正确

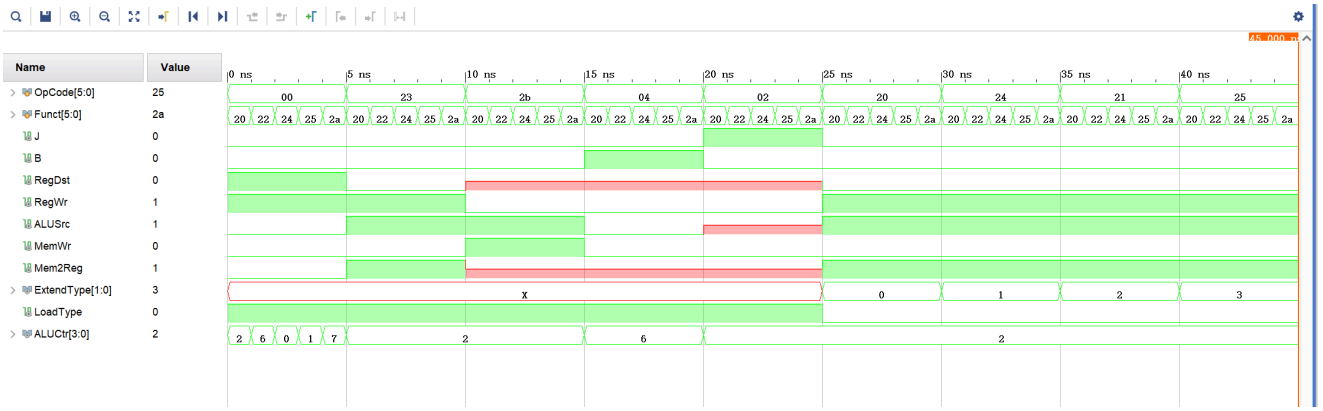
- `lh $8, 0($2)` 130ns时，`RegFile` 中8号寄存器的值为 0x00001234，为 `DataRAM` 中2号寄存器的第一个半字的值 1234 作符号扩展的结果，执行正确
- `lh $8, 2($2)` 140ns时，`RegFile` 中8号寄存器的值为 0xffff89ab，为 `DataRAM` 中2号寄存器的第二个半字的值 89ab 作符号扩展的结果，执行正确
- `lhu $8, 0($2)` 150ns时，`RegFile` 中8号寄存器的值为 0x00001234，为 `DataRAM` 中2号寄存器的第一个半字的值 1234 作无符号扩展的结果，执行正确
- `lh $8, 2($2)` 160ns时，`RegFile` 中8号寄存器的值为 0x000089ab，为 `DataRAM` 中2号寄存器的第二个半字的值 89ab 作无符号扩展的结果，执行正确
- `sub $2, $4, $3` 170ns时，`RegFile` 中2号寄存器的值为 0，此前4号寄存器和3号寄存器的值均为 0x0000000c，执行正确
- `or $2, $4, $3` 180ns时，`RegFile` 中2号寄存器的值为 0x0000000c，执行正确
- `and $2, $4, $3` 190ns时，`RegFile` 中2号寄存器的值为 0x0000000c，执行正确
- `slt $2, $4, $3` 200ns时，`RegFile` 中2号寄存器的值为 0，执行正确
- `beq $3, $3, equ` 跳转执行 `beq $3, $4, exit`，又跳转执行 `j main`

各仿真波形图如下：

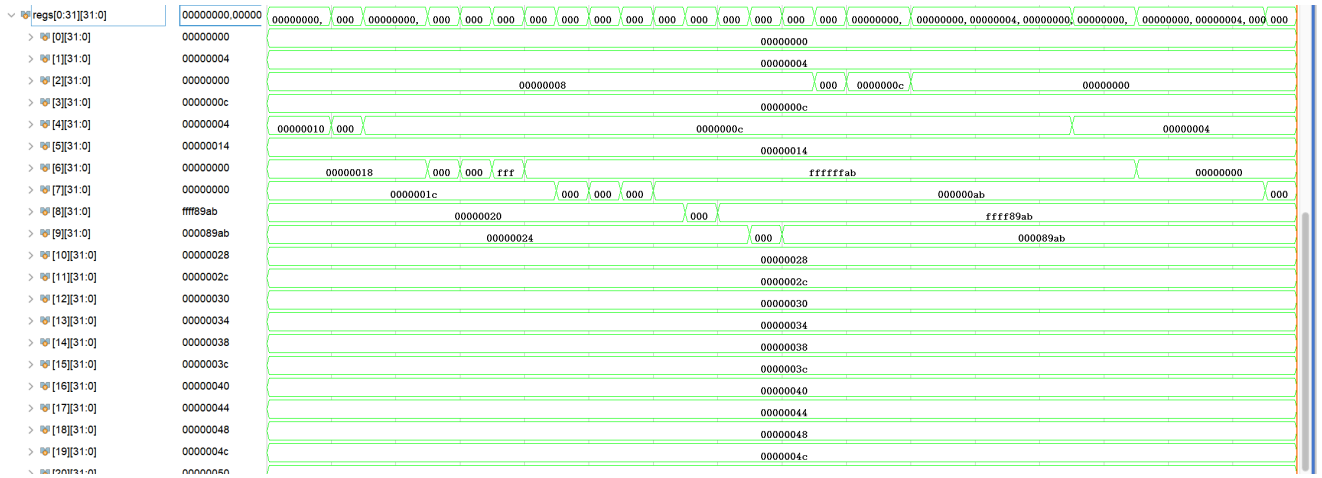
`DigitExtend`

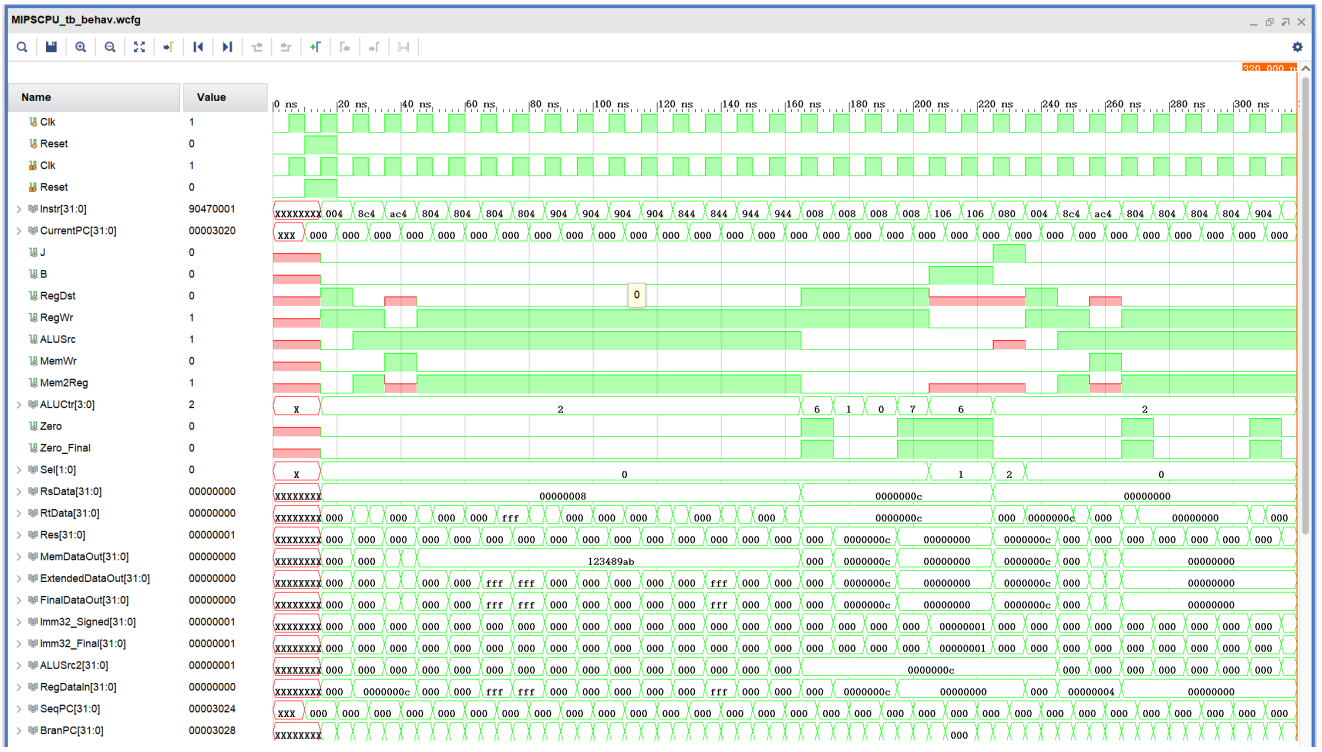


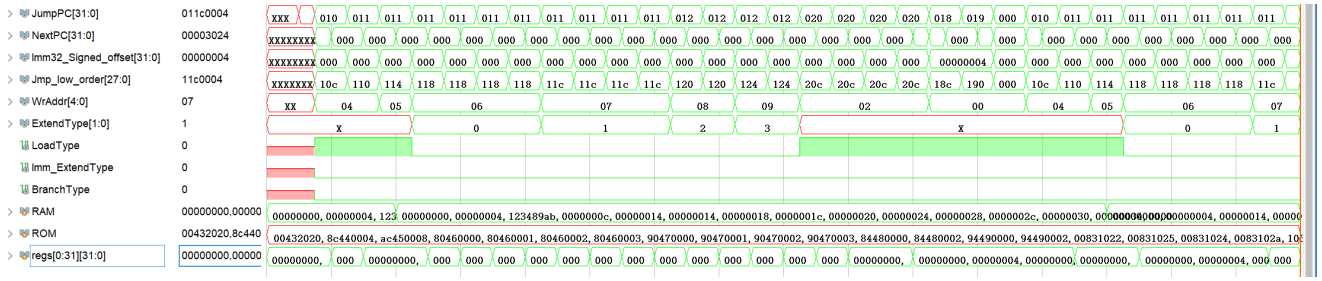
`Controller`

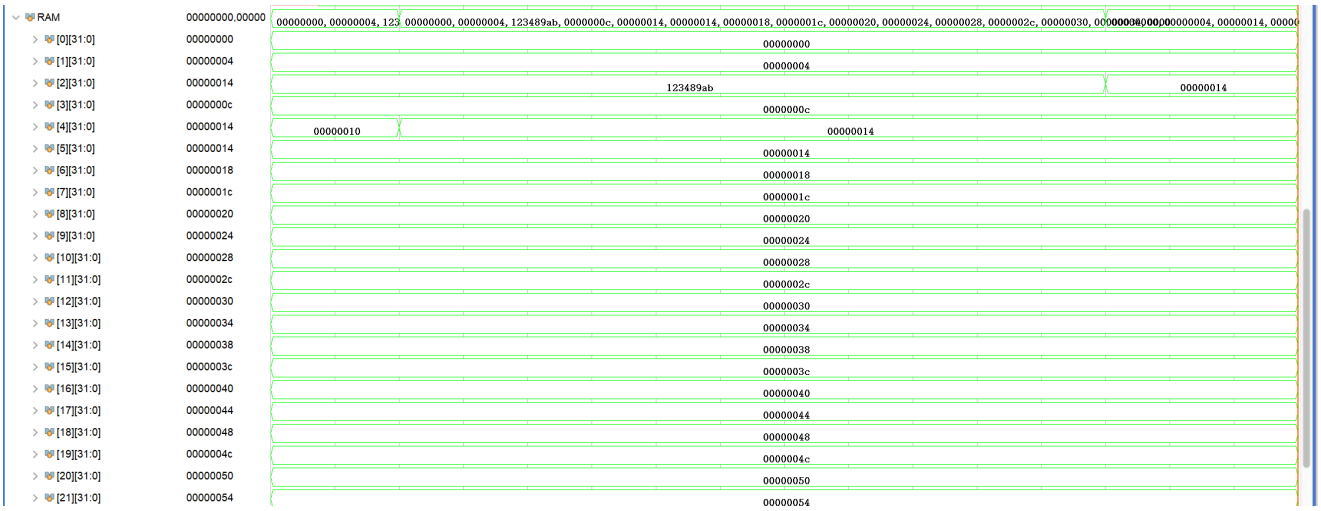


MIPSCPU


regs[0:31][31:0]
00000000,00000
[0][31:0] 00000000
[1][31:0] 00000004
[2][31:0] 00000000
[3][31:0] 0000000c
[4][31:0] 00000004
[5][31:0] 00000014
[6][31:0] 00000000
[7][31:0] 00000000
[8][31:0] ffff89ab
[9][31:0] 000089ab
[10][31:0] 00000028
[11][31:0] 0000002c
[12][31:0] 00000030
[13][31:0] 00000034
[14][31:0] 00000038
[15][31:0] 0000003c
[16][31:0] 00000040
[17][31:0] 00000044
[18][31:0] 00000048
[19][31:0] 0000004c


MIPSCPU_tb_behav.wcfg
Name
Value
Clk 1
Reset 0
Clk 1
Reset 0
Instr[31:0] 90470001
CurrentPC[31:0] 00003020
J 0
B 0
RegDst 0
RegWr 1
ALUSrc 1
MemWr 0
Mem2Reg 1
ALUCtr[3:0] 2
Zero 0
Zero_Final 0
Sel[1:0] 0
RsData[31:0] 00000000
RtData[31:0] 00000000
Res[31:0] 00000001
MemDataOut[31:0] 00000000
ExtendedDataOut[31:0] 00000000
FinalDataOut[31:0] 00000000
Imm32_Signed[31:0] 00000001
Imm32_Final[31:0] 00000001
ALUSrc2[31:0] 00000001
RegDataIn[31:0] 00000000
SeqPC[31:0] 00003024
BranPC[31:0] 00003028


JumpPC[31:0] 011c0004
NextPC[31:0] 00003024
Imm32_Signed_offset[31:0] 00000004
Jmp_low_order[27:0] 11c0004
WrAddr[4:0] 07
ExtendType[1:0] 1
LoadType 0
Imm_ExtendType 0
BranchType 0
RAM 00000000,00000
ROM 00432020,8c440
regs[0:31][31:0] 00000000,00000



实验总结

此次实验内容较多，我在完成了线上网站实验以及指令集扩展电路设计作业的基础上用硬件描述语言实现并仿真，既复习了指令架构，又对指令的具体执行流程更加清楚，锻炼了从简单到复杂的构建能力