实验二：简单指令集MIPS单周期微处理器设计

专业班级：**通信2101班**

姓名：　　**罗畅**

学号：　　U202113940

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. 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. 了解软件控制硬件工作的基本原理

实验仪器

Vivado2018.03 、*Mars MIPS*汇编编译器、*Visual Studio Code Insiders*

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：各个小组所属成员需扩展实现各小组要求扩展的指令：

- 1）addi，ori；
- 2）lb，lbu，lh，lhu；
- 3）bne；bltz；bgez；
- 4）jal，jr;
- 5） sb,sh;
- 6）sll,srl;sllv,srlv;

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

生成机器码和iromIP核

根据上述要求，我选择了addi和ori指令，并将其添加进汇编指令代码中，如下：

```
main:
add $4,$2,$3
addi $5,$4,44
lw $4,4($2)
sw $5,8($2)
sub $2,$4,$3
or $2,$4,$3
and $2,$4,$3
ori $2,$2,55
slt $2,$4,$3
beq $3,$3,equ
lw $2,0($3)
equ:
beq $3,$4,exit
sw $2,0($3)
exit:
j main
```

用mars软件转换机器码如下：

```
00432020
2085002c
8c440004
ac450008
00831022
00831025
00831024
34420037
0083102a
10630001
8c620000
10640001
ac620000
08000000
```

生成如下.coe文件：

my_task_two.coe

```
memory_initialization_radix=16;
memory_initialization_vector=
00432020
2085002c
8c440004
ac450008
00831022
00831025
00831024
34420037
0083102a
10630001
8c620000
10640001
ac620000
08000000;
```

使用Vivado的**IPCatalog**功能，选择**Block Memory Generator**，将要生成的IP核命名为iromIP，Interface Type选择本地接口，memory Type选择Single Port ROM，更改读写宽深均为32，更改为总是使能，最后将生成好的my_task_two.coe导入该IP核中，完成生成。后续我们将使用该IP核作为指令储存器

源代码

源代码github地址：GitHub-MIPS_CPU

ALU.v

```verilog
`timescale 1ns / 1ps

module ALU(
    input signed [31:0] inputA,
    input signed [31:0] inputB,
    input [3:0] ALUCtrl,
    output reg [31:0] ALUResult,
    output reg  zero
    );

    always @(inputA or inputB or ALUCtrl)
        begin
            case (ALUCtrl)
                4'b0110:
                    begin
                        ALUResult = inputA - inputB;
                        zero = (ALUResult == 0) ? 1 : 0;
                    end
                4'b0010:
                    begin
                        ALUResult = inputA + inputB;
                        zero = 0;
                    end
                4'b0000:
                    begin
                        ALUResult = inputA & inputB;
                        zero = 0;
                    end
                4'b0001:
                    begin
                        ALUResult = inputA | inputB;
                        zero = 0;
                    end
                4'b0111:
                    begin
                        ALUResult = (inputA < inputB) ? 1 : 0;
                        zero = 0;
                    end
                default:
                    begin
                        ALUResult = 0;
                        zero = 0;
                    end
            endcase
        end
endmodule
```

ALUCtrl.v

```
`timescale 1ns / 1ps

module ALUCtrler(
    input [1:0] ALUop,
    input [5:0] func,
    output reg [3:0] ALUCtrl
    );
    always @(ALUop or func)
        casex ({ALUop,func})
            8'b00xx_xxxx: ALUCtrl = 4'b0010;
            8'b01xx_xxxx: ALUCtrl = 4'b0110;
            8'b10xx_0000: ALUCtrl = 4'b0010;
            8'b10xx_0010: ALUCtrl = 4'b0110;

            8'b10xx_0100: ALUCtrl = 4'b0000;
            8'b10xx_0101: ALUCtrl = 4'b0001;
            8'b10xx_1010: ALUCtrl = 4'b0111;
            default: ALUCtrl = 4'b0000;
        endcase
endmodule
```

dram.v

```
`timescale 1ns / 1ps

module dram(
    input CLK,
    input [4:0] addr,
    output [31:0] readData,
    input [31:0] writeData,
    // write ctrl signal
    input MemWR

);
    reg [31:0] regs [0:31];
    assign readData = regs[addr];
    // always @(addr or writeData or MemWR)
    always @(negedge CLK)
    // data ram write at CLK's negedge
        if (MemWR) regs[addr] = writeData;

    integer i;
    initial
        for (i = 0; i < 32;i = i + 1) begin
            // data ram initializes the value of the address * 4
            regs[i] = i * 4;
        end
endmodule
```

irom.v

```
`timescale 1ns / 1ps

module irom(
    input [4:0] addr,
    input [31:0] instr
    );
    reg [31:0] regs [0:31];

    assign instr = regs [addr];
    initial
    $readmemh("D:/MIPS_asm/task2.coe",regs,0,10);
endmodule
```

MainCtrl.v

```verilog
`timescale 1ns / 1ps

module MainCtrl(
    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) begin
        case (opCode)
            6'b00_0010:outputTemp = 9'bxxx0_001_xx;
            6'b00_0000:outputTemp = 9'b1001_000_10;
            6'b10_0011:outputTemp = 9'b0111_000_00;
            6'b10_1011:outputTemp = 9'bx1x0_100_00;
            6'b00_0100:outputTemp = 9'bx0x0_010_01;
            // immediate num op
            6'b00_1000:outputTemp = 9'b01x1_000_00;
            6'b00_1101:outputTemp = 9'b0101_000_10;
            default: outputTemp = 9'b0000_000_00;
        endcase
    end
endmodule
```

RegFile.v

```verilog
`timescale 1ns / 1ps

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;
    // Register file write at CLK's negedge
    always @(negedge CLK)
        if (!reset && RegWr)
            regs[WriteAddr] = WriteData;
        else if (reset)
            for (i = 0; i < 32; i = i + 1)
                regs[i] = i * 4;
endmodule
```

MIPS_CPU.v

```verilog
`timescale 1ns / 1ps

module MIPS_CPU(
    // clock signal and reset siganl
    input CLK,
    input reset
    );
    // tempPC: value written to PC under clock signal control;
    // SequencePC: value of PC when the program is executed sequentially
    // BranchPC: value of the destination address when a conditional jump holds
    // MuxPC: multiplexer output signal
    // BranchZ: control signal of MuxPC
    // RegWriteData: siganl of writing to register
    // RegWriteAddr: register number
    // Imm32: 16 bit immediate number extend
    // JumpPC: destination Addr
    // PsudeoPC: Pseudo-direct jump 28 bit addr
    wire [31:0] TempPC,MuxPC,JumpPC,BranchPC,SequencePC,Imm32,ImmL2,
         RegWriteData,RSData,RTData,ALUIn2,ALURes,MemoryReadData,Instr;
    wire [4:0] RegWriteAddr;
    wire [27:0] PsudeoPC;
    // signal bit control signal
    // RegWr: regFile write control signal
    // Zero: from ALU, if it is sub calculation and result != 0, zero set to 1, else
set to 0
    // MemWR: memory writeable control signal
    // ALUSrc: ALU's second calcualtion number source(32 bit immediate number or from
regFile)
    wire BranchZ,J,B,Zero,RegDst,RegWr,ALUSrc,MemWR,Mem2Reg;
    // ALU operation control number
    wire [1:0] ALUop;
    wire [3:0] ALUCtrl;
    // program register
    reg [31:0] PC;

    assign PsudeoPC = {Instr[25:0],2'b00};
    assign JumpPC = {SequencePC[31:28],PsudeoPC};
    assign SequencePC = PC + 4;
    assign BranchPC = ImmL2 + SequencePC;
    assign MuxPC = BranchZ ? BranchPC : SequencePC;
    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 RegWriteAddr = RegDst ? Instr[15:11] : Instr[20:16];
    assign RegWriteData = Mem2Reg ? MemoryReadData : ALURes;

    ALU unitALU(RSData,ALUIn2,ALUCtrl,ALURes,Zero);
    dram unitDram(CLK,ALURes[6:2],MemoryReadData,RTData,MemWR);
    // irom unitIrom(PC[6:2],Instr);
    // dramIP unitDram(~CLK,MemWR,ALURes[6:2],RTData,MemoryReadData);
    // read machine code at CLK's posedge
    iromIP unitIrom(CLK,PC[6:2],Instr);
    RegFile unitRegFile(Instr[25:21],Instr[20:16],RegWriteAddr,RegWr,
                        RegWriteData,CLK,reset,RSData, RTData);
    MainCtrl unitMainCtrl(Instr[31:26],ALUop,RegDst,RegWr,
                          ALUSrc,MemWR,B,J,Mem2Reg);
    ALUCtrler unitALUCtrl(ALUop,(ALUSrc && Instr[31:26] == 6'b00_1101 ?
                          6'bxx_0101:Instr[5:0]),ALUCtrl);
    always @(negedge CLK)
        begin
            // PC initial value set to zero
            if (reset) PC <= 0;
            // change program pointer at CLK's negedge
            else PC <= TempPC;
        end
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，

数据储存器使用的是自行编写的dram.v，指令储存器使用的是生成的iromIP核

按照要求：

- 读取指令在时钟上升沿

```
// read machine code at CLK's posedge
iromIP unitIrom(CLK,PC[6:2],Instr);
```

- 指令指针修改在时钟下降沿

```
always @(negedge CLK)
        begin
            // PC initial value set to zero
            if (reset) PC <= 0;
            // change program pointer at CLK's negedge
            else PC <= TempPC;
        end
```

- 寄存器文件写入在时钟下降沿

```
RegFile unitRegFile(Instr[25:21],Instr[20:16],RegWriteAddr,RegWr,
                        RegWriteData,CLK,reset,RSData, RTData);

// Register file write at CLK's negedge
    always @(negedge CLK)
        if (!reset && RegWr)
            regs[WriteAddr] = WriteData;
        else if (reset)
            for (i = 0; i < 32; i = i + 1)
                regs[i] = i * 4;
```

- 数据存储器数据写入都在时钟下降沿

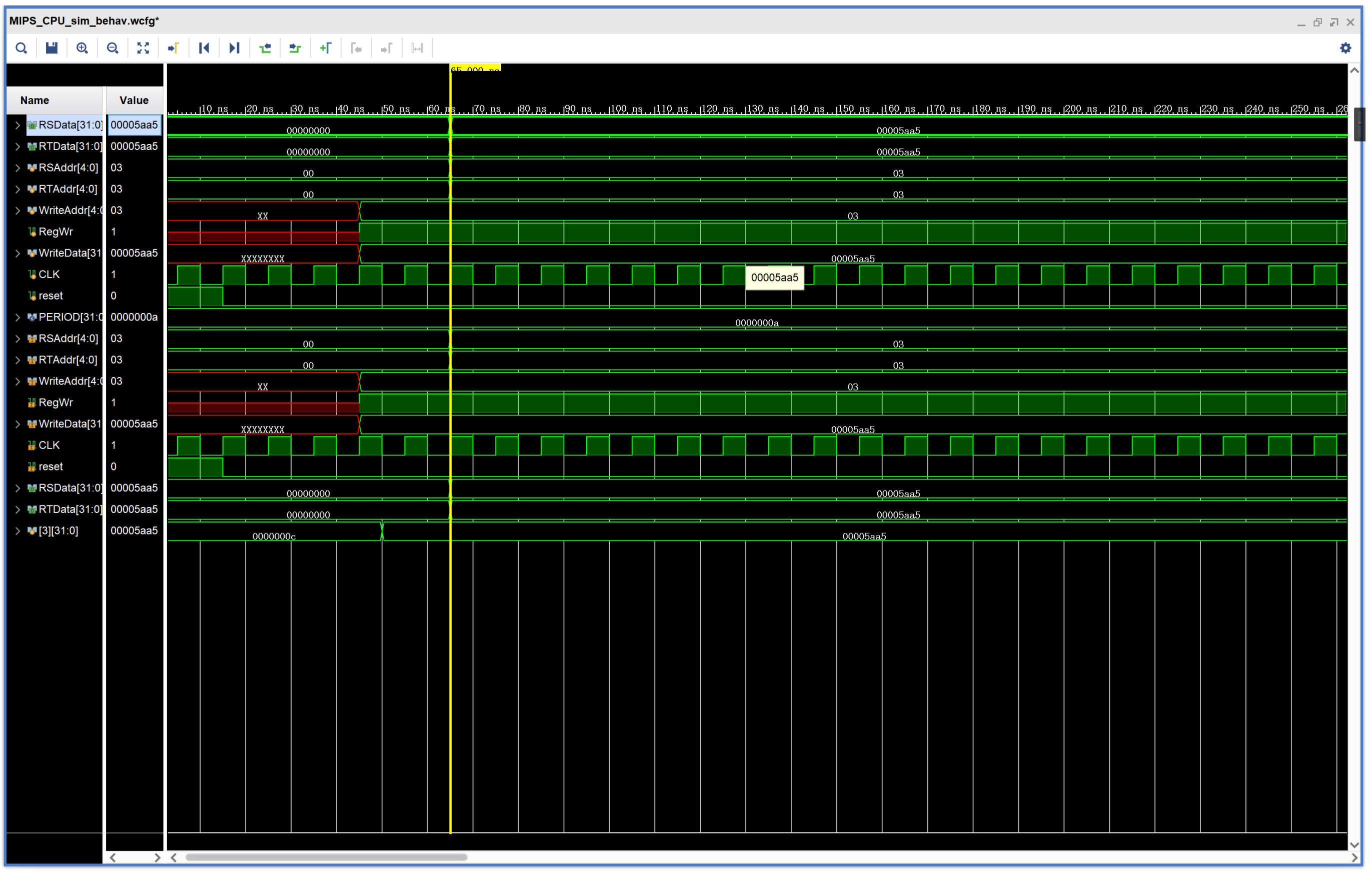
```
dram unitDram(CLK,ALURes[6:2],MemoryReadData,RTData,MemWR);


always @(negedge CLK)
    // data ram write at CLK's negedge
    if (MemWR) regs[addr] = writeData;
```

仿真代码

RegFileSim.v

```verilog
`timescale 1ns / 1ps

module RegFileSim(
    output [31:0] RSData,
    output [31:0] RTData
    );
    reg [4:0] RSAddr;
    reg [4:0] RTAddr;
    reg [4:0] WriteAddr;
    reg RegWr;
    reg [31:0] WriteData;
    reg CLK;
    reg reset;
    parameter PERIOD = 10;

    RegFile U0(RSAddr,RTAddr,WriteAddr,RegWr,WriteData,
               CLK,reset,RSData,RTData);

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

    initial
        begin
        reset = 1;
        RSAddr = 5'h0;
        RTAddr = 5'h0;
        #15
        reset = 0;
        #30
        RegWr = 1;
        WriteAddr = 5'h03;
        WriteData = 32'h5aa5;
        #20
        RSAddr = 5'h03;
        RTAddr = 5'h03;
        end

endmodule
```

得到仿真波形:

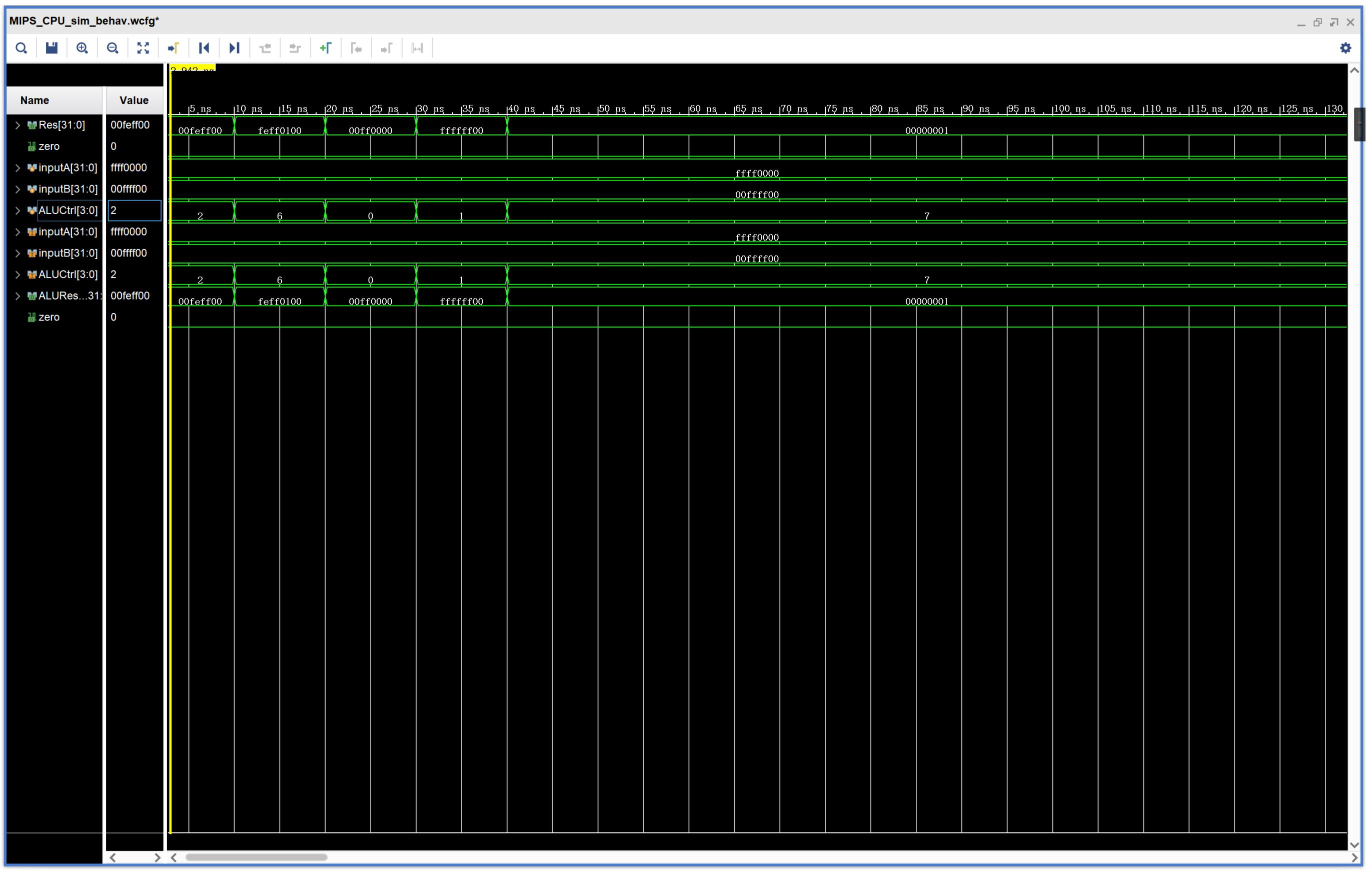


Name	Value
RSData[31:0]	00005aa5
RTData[31:0]	00005aa5
RSAddr[4:0]	03
RTAddr[4:0]	03
WriteAddr[4:0	03
RegWr	1
WriteData[31	00005aa5
CLK	1
reset	0
PERIOD[31:0	0000000a
RSAddr[4:0]	03
RTAddr[4:0]	03
WriteAddr[4:0	03
RegWr	1
WriteData[31	00005aa5
CLK	1
reset	0
RSData[31:0]	00005aa5
RTData[31:0]	00005aa5
[3][31:0]	00005aa5

ALUsim.v

```verilog
`timescale 1ns / 1ps

module ALUsim(
    output [31:0] Res,
    output zero
    );
    reg [31:0] inputA;
    reg [31:0] inputB;
    reg [3:0] ALUCtrl;

    ALU U0(inputA,inputB,ALUCtrl,Res,zero);
    initial begin
        inputA = 32'hffff_0000;
        inputB = 32'h00ff_ff00;
        ALUCtrl = 4'h2;
        #10
        ALUCtrl = 4'h6;
        #10
        ALUCtrl = 4'h0;
        #10
        ALUCtrl = 4'h1;
        #10
        ALUCtrl = 4'h7;
    end
endmodule
```

得到仿真波形：

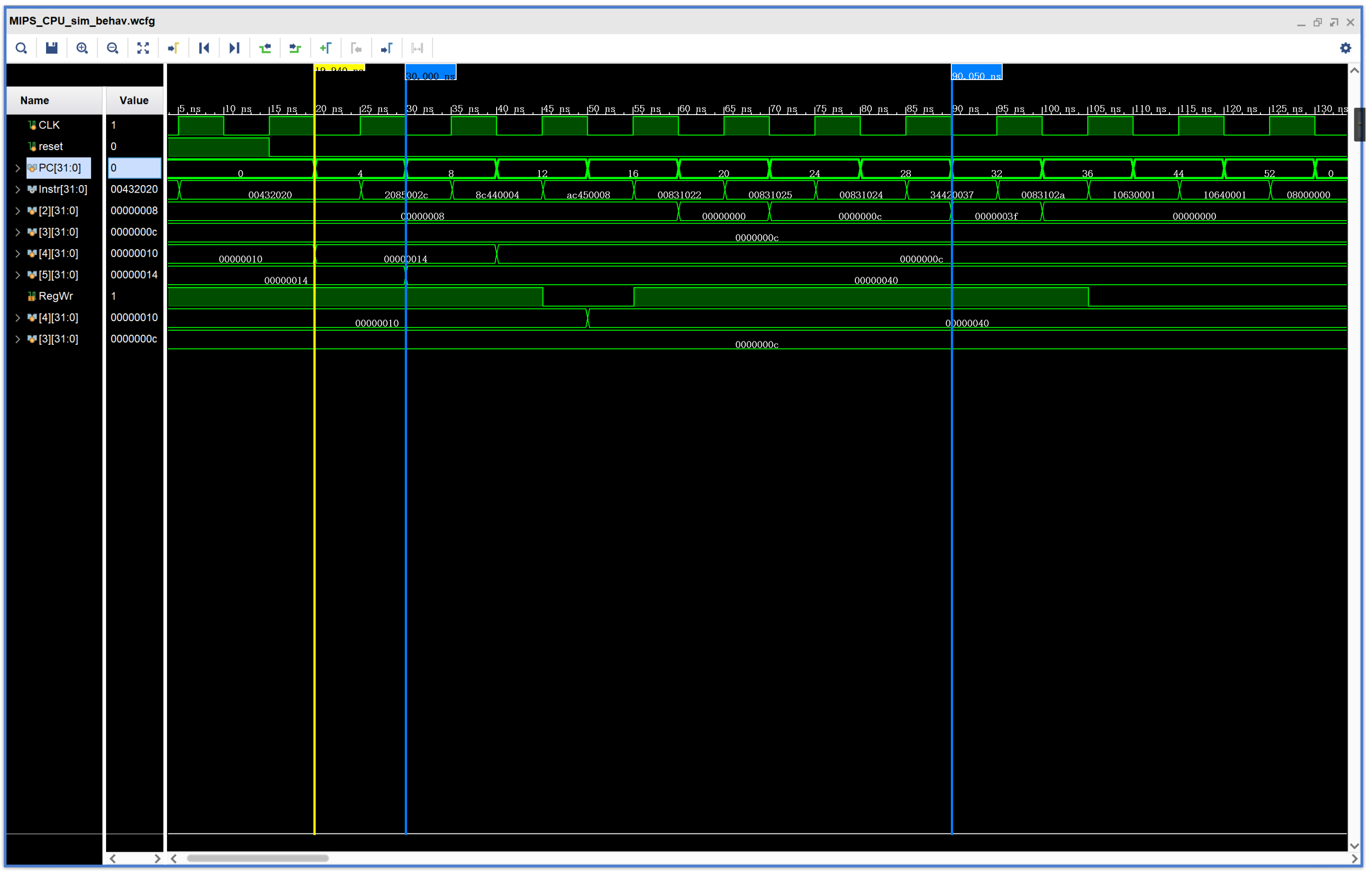


Name	Value
Res[31:0]	00feff00
zero	0
inputA[31:0]	ffff0000
inputB[31:0]	00ffff00
ALUCtrl[3:0]	2
inputA[31:0]	ffff0000
inputB[31:0]	00ffff00
ALUCtrl[3:0]	2
ALURes...31:	00feff00
zero	0

MIPS_CPU_sim.v

```
`timescale 1ns / 1ps

module MIPS_CPU_sim(

    );
    reg CLK,reset;
    MIPS_CPU u0(CLK,reset);
    parameter PERIOD = 10;

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

    initial
        begin
            reset = 1;
            #15
            reset = 0;
        end
endmodule
```

得到仿真波形：



顶层文件仿真结果分析

对MIPS_CPU的仿真得到如上波形，下面将分析扩展功能：

`addi $5,$4,44` 和 `ori $2,$2,55`

对应机器码为：

`2085002c` 和 `34420037`

在波形窗口中，我们可以看到，25ns时，获取指令 `addi $5,$4,44` ，将4号寄存器中的值 `0x0000_0014` 和立即数 `44` （十进制）相加得到值 `0x0000_0040`；30ns时，在CLK下降沿作用下将值 `0x0000_0040` 写入寄存器$5中，**执行正确**；

85ns时，获取指令 `ori $2,$2,55` ，将2号寄存器中的值 `0x0000_000c` 和立即数 `0x0000_0037` 进行或运算，得到值 `0x0000_003f`；90ns时，在CLK下降沿作用下将值 `0x0000_003f` 写入寄存器$2中，**执行正确。**

在所有指令执行完成后，执行指令 `j main` ，再次回到第0条指令，循环执行irom中储存的指令。

实验小结

在这次实验中，我用verilog语言设计了一个MIPS单周期微处理器，更加深入地理解了MIPS单周期微处理器的运行原理。理解了软件控制硬件的工作原理，也加深了对MIPS微处理器的基本结构与其每个模块工作过程的理解。

由于vivado代码编辑器不支持自动补全，语法高亮也不怎么实用，这对于我一个用习惯了JetBrains编译器的程序员非常不友好，所以我额外配置了VSCode作为写代码的辅助工具，让我的写代码效率快了很多

这次实验收获很大！