计算机系统结构实验报告 实验五

王梓涵 517021911179

概述

实验名称

简单的类 MIPS 单周期处理器实现——整体调试

实验目的

- 完成单周期的类 MIPS 处理器整体设计
- 进行功能仿真与上版验证

顶层模块

模块描述

在顶层模块内，将之前两个实验所实现的是所有逻辑模块实例化，在模块之间用信号线连接，使所有模块成为一个整体，并能正确执行指令。

补充模块

在连接模块之前，需要额外实现模块，使得处理器的部件完备。

指令存储器

在本实验中，数据和指令是分开存储的，需要实现一个指令存储器。指令存储器是只读存储器，不可修改，所以是一个组合逻辑单元。指令存储器是按字节寻址的，一个指令占据四个地址的存储空间。这样是为了和后续 PC + 4、立即数左移两位作为跳转地址等运算相协调。

```
module InstMemory(
    input [31:0] address,
    output [31:0] instruction
);

    reg [7:0] instrFile[0:63];

    assign instruction = {instrFile[address+3], instrFile[address+2],
                          instrFile[address+1], instrFile[address]};

endmodule
```

数据选择器

在顶层模块中用一个三元运算符即可以实现该功能。也可以实现为一个模块，将两路数据和选择信号作为数据，输出一路选择的数据。这样实现顶层代码时，代码可读性会相对好一些。在顶层模块中，既有对 32 位的数据选择器，也有 5 位的数据选择器，都要实现。这里给出 32 位的数据选择器：

```
module Mux32(
    input sel,
    input [31:0] in1,
    input [31:0] in0,
    output [31:0] out
);

    assign out = sel ? in1 : in0;

endmodule
```

修改模块

在实例化之前，要对前两个实验的实现的模块做一些修改，使得它们可以正确地与其它模块协调运作。

寄存器

寄存器需要对 reset 信号作出相应，将各寄存器清零。这里采用的是同步的、上升沿触发的清零方式，通过 for 循环来对每个寄存器清零。

```
reg [5:0] cnt;
always @ (posedge Clk)
    begin
        if (reset)
            begin
                for (cnt = 0; cnt < 32; cnt=cnt+1)
                    regFile[cnt] <= 0;
            end
    end
```

存储器

实验四中实现的存储器是按字寻址的，为了和 MIPS 处理器中按字节寻址的方式一致，这里改为按字节寻址。不过由于目前实现的指令一次需要访问一个字的数据，所以读取时要拼接四个字节的数据输出，写入时要将一个字的数据写到四个连续的地址中。

```
module DataMemory(
    input Clk,
    input [31:0] address,
    input [31:0] writeData,
    input memWrite,
    input memRead,
    output [31:0] readData
);
```

```
    reg [7:0] memFile[0:31];
    reg [31:0] readData;


    always @ (address or memRead)
        begin
            if (memRead)
                readData = {memFile[address+3], memFile[address+2],
                            memFile[address+1], memFile[address]};
            else
                readData = 0;
        end

    always @ (negedge Clk)
        begin
            if (memWrite)
                begin
                    memFile[address] <= writeData[7:0];
                    memFile[address+1] <= writeData[15:8];
                    memFile[address+2] <= writeData[23:16];
                    memFile[address+3] <= writeData[31:24];
                end
        end

endmodule
```

连接模块

下面通过若干连接线将之前实现的模块连接起来。为了有条理地连线，这里将代码分成若干块，每一块内只关心一个模块周围的连线。这样可以避免因为线路混乱而发生错连、漏连的情况。

指令存储器

读入 PC，输出 PC 所对应的 32 位指令。

```
reg [31:0] PC;
wire [31:0] INST;
InstMemory instMem(.address(PC), .instruction(INST));
```

控制模块

读入指令高 6 位，输出控制信号。

```
wire REG_DST, JUMP, BRANCH, MEM_READ, MEM_TO_REG, MEM_WRITE, ALU_SRC, REG_WRITE;
wire [1:0] ALU_OP;
Ctr mainCtr(.opCode(INST[31:26]), .regDst(REG_DST), .jump(JUMP), .branch(BRANCH),
            .memRead(MEM_READ), .memToReg(MEM_TO_REG), .aluOp(ALU_OP),
            .memWrite(MEM_WRITE), .aluSrc(ALU_SRC), .regWrite(REG_WRITE));
```

寄存器

根据 RegDst 选择写入的目标寄存器是 Rt 还是 Rd，根据指令的 Rs 和 Rt 位确定要读取的目标寄存器，执行读取和写入操作，同时能响应重置信号。

```
wire [4:0] WRITE_REG;
wire [31:0] READ_DATA_1, READ_DATA_2, REG_WRITE_DATA;
Mux5 writeRegMux(.sel(REG_DST), .in1(INST[15:11]), .in0(INST[20:16]), .out(WRITE_REG));
Registers regs(.Clk(Clk), .readReg1(INST[25:21]), .readReg2(INST[20:16]),
               .writeReg(WRITE_REG), .writeData(REG_WRITE_DATA) , .reset(reset),
               .regWrite(REG_WRITE), .readData1(READ_DATA_1), .readData2(READ_DATA_2));
```

ALU

将指令低 16 位符号扩展，将指令低 6 位和 ALUOp 送给 ALUCtr 译码，根据 ALUSrc 选择 ALU 第二个操作数的来源。将 ALUCtr 的输出信号和两个操作数送给 ALU，输出运算结果和零标志位。

```
wire [31:0] IMM_SEXT, ALU_SRC_B, ALU_RESULT;
wire [3:0] ALU_CTR_OUT;
wire ZERO;
Signext signext(.inst(INST[15:0]), .data(IMM_SEXT));
ALUCtr aluCtr(.funct(INST[5:0]), .aluOp(ALU_OP), .aluCtrOut(ALU_CTR_OUT));
Mux32 aluSrcMux(.sel(ALU_SRC), .in1(IMM_SEXT), .in0(READ_DATA_2),.out(ALU_SRC_B));
Alu alu(.in1(READ_DATA_1), .in2(ALU_SRC_B), .aluCtr(ALU_CTR_OUT), .zero(ZERO),
        .aluRes(ALU_RESULT));
```

数据存储器

根据 MemRead 和 MemWrite 决定要不要读/写，将 ALU 的运算结果作为数据存储器的地址，将寄存器读出的第二个数据作为数据存储器的写入数据。根据 MemToReg 决定写回寄存器的数据来自内存还是 ALU 的运算结果。

```
wire [31:0] MEM_READ_DATA;
DataMemory dataMem(.Clk(Clk), .address(ALU_RESULT), .writeData(READ_DATA_2),
                   .memWrite(MEM_WRITE), .memRead(MEM_READ), .readData(MEM_READ_DATA));
Mux32 regWriteMux(.sel(MEM_TO_REG), .in1(MEM_READ_DATA), .in0(ALU_RESULT),
                  .out(REG_WRITE_DATA));
```

跳转逻辑

决定下一个 PC 的值，PC 的值可能来自 PC + 4、`beq` 指令指定的条件跳转地址或 `j` 指令指定的无条件跳转地址。条件跳转地址为指令低 16 位的立即数符号扩展后加上 PC + 4 的值，无条件跳转地址为指令低 26 位的立即数左移两位再拼接上 PC + 4 的高 4 位的值。先根据 Branch 和 Zero 的与运算结果决定是来自 PC + 4 还是条件跳转地址，再根据 Jump 决定是来自前面计算的地址还是无条件跳转地址，将此结果作为下一个 PC 的值。在时钟上升沿，将 PC 更新为下一个值，如果重置信号为高电平，则将 PC 置为零。

```
wire [31:0] PC_PLUS_4, BRANCH_ADDR, SEL_BRANCH_ADDR, JUMP_ADDR, NEXT_PC, SEXT_SHIFT;
assign PC_PLUS_4 = PC + 4;
assign JUMP_ADDR = {PC_PLUS_4[31:28], INST[25:0] << 2};
assign SEXT_SHIFT = IMM_SEXT << 2;
assign BRANCH_ADDR = PC_PLUS_4 + SEXT_SHIFT;
Mux32 branchMux(.sel(BRANCH & ZERO), .in1(BRANCH_ADDR), .in0(PC_PLUS_4),
                .out(SEL_BRANCH_ADDR));
```

```
Mux32 jumpMux(.sel(JUMP), .in1(JUMP_ADDR), .in0(SEL_BRANCH_ADDR), .out(NEXT_PC));

always @ (posedge Clk)
    begin
        if (reset)
            PC <= 0;
        else
            PC <= NEXT_PC;
    end
```

仿真测试

测试程序

当前的单周期处理器能够执行指导书上所提供的 6 条指令，下面使用这六条指令编写一段简单程序，以测试处理器是否正常工作。

```
START:
lw  $1, 0($0)
lw  $2, 4($0)
lw  $3, 8($0)
OP:
add $1, $1, $2
or  $4, $1, $2
slt $5, $3, $4
sw  $4, 12($0)
beq $2, $3, OP
j   START
```

数据存储器中初始化的值为：

```
0x00-0x03   1
0x04-0x07   2
0x08-0x0B   3
0x0C-0x0F   4
```

在程序开始，执行下列主要步骤：

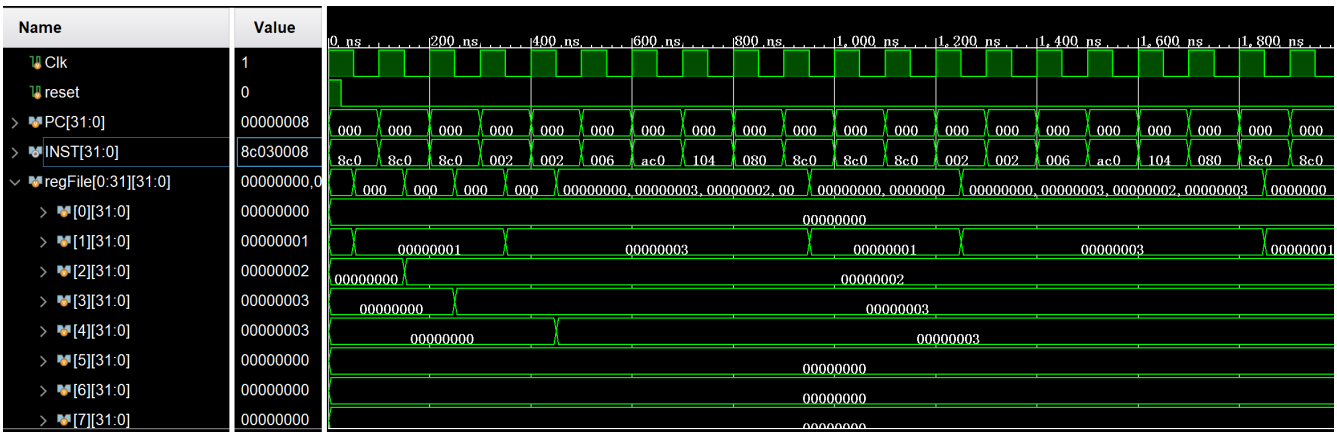
1. 将数据 1，2，3 分别载入寄存器 `$1`、`$2` 、`$3`；
2. 对寄存器中的操作数进行 `add`、`or` 和 `slt` 等运算，`$1` 和 `$4` 中存储的应为 3，`$5` 中存储的应为 0；
3. 将 `$4` 中的数据（即 3）存储到数据存储器的 `0x0C-0x0F` 中；
4. 根据 `$2` 和 `$3` 中数据是否相等决定是否跳转到 `OP`，由于 `$2` 中存储的是 2，`$3` 中存储的是 3，一定不相等，所以该指令不会发生跳转；
5. 无条件跳转到 `START`。

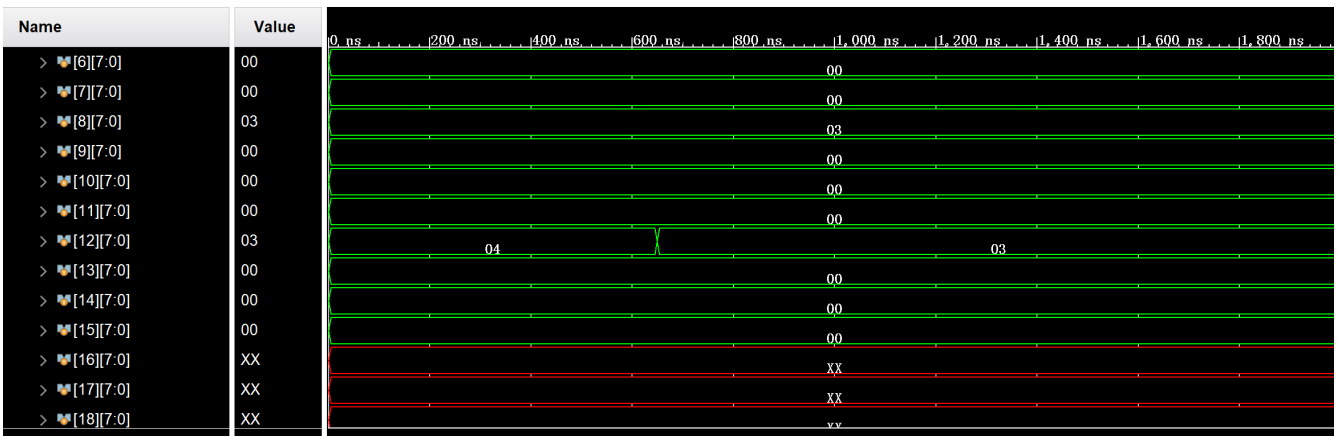
激励信号

编写测试激励模块，使用系统任务 `readmem` 将指令和数据分别加载到指令寄存存储器和数据存储器，时钟周期设为 100 ns，在前 25 ns 重置信号为高电平。

```
module Top_tb();

    reg Clk, reset;
    always #50 Clk = !Clk;

    Top top(.Clk(Clk), .reset(reset));

    initial begin
        $readmemh("data.txt", top.dataMem.memFile);
        $readmemb("code.txt", top.instMem.instrFile);
        Clk = 1;
        reset = 1;
        #25
        reset = 0;
    end

endmodule
```

仿真波形





第一张图为寄存器的内容，第二张图为数据存储器的内容。经观察，仿真波形符合预期，单周期处理器通过仿真测试。

上板验证

目前能够实现将 PC 显示在数码管上，并能通过 LED 显示时钟信号。

模块实现

先通过 `IBUFGDS` 设置时钟信号，将时钟二分频，将时钟和重置信号送给处理器的顶层模块，最后通过实验二给出的显示模块将处理器的 PC 和时钟信号输出。

```
module OnBoard(
    input clk_p,
    input clk_n,
    input [3:0] a,
    input [3:0] b,
    input reset,

    output led_clk,
    output led_do,
    output led_en,

    output wire seg_clk,
    output wire seg_en,
    output wire seg_do
);

    // Configure clock input
    wire CLK_i, CLK_LED;
    IBUFGDS ibufgds(.O(CLK_i), .I(clk_p), .IB(clk_n));

    // Clock frequency division
    reg [15: 0] div;
    always @ (posedge CLK_i)
        begin
            div <= div + 1;
        end
    assign CLK_LED = div[1];

    // CPU
    Top cpu(.Clk(CLK_LED), .reset(!reset));

    // Display
    display dis(.clk(CLK_LED), .rst(1'b0), .en(8'b00001111), .data(cpu.PC),
                .dot(8'b0), .led(~div), .led_clk(led_clk), .led_en(led_en),
                .led_do(led_do), .seg_clk(seg_clk), .seg_en(seg_en), .seg_do(seg_do));

endmodule
```

在上板验证过程中，无法将通过系统任务初始化指令存储器和数据存储器。需要在指令存储器和数据存储器的 `intial` 块里分别将这些内容逐个输入。其过程较为繁琐，在此省略。

约束文件

基本沿用了实验二给出的约束文件，仅加入了重置按钮接口。

```
set_property PACKAGE_PIN AC18 [get_ports clk_p]
set_property IOSTANDARD LVDS [get_ports clk_p]
set_property IOSTANDARD LVDS [get_ports clk_n]

set_property PACKAGE_PIN W13 [get_ports reset]
set_property IOSTANDARD LVCMOS18 [get_ports reset]

set_property PACKAGE_PIN AA12 [get_ports {a[3]}]
set_property PACKAGE_PIN AA13 [get_ports {a[2]}]
set_property PACKAGE_PIN AB10 [get_ports {a[1]}]
set_property PACKAGE_PIN AA10 [get_ports {a[0]}]
set_property IOSTANDARD LVCMOS15 [get_ports {a[3]}]
set_property IOSTANDARD LVCMOS15 [get_ports {a[2]}]
set_property IOSTANDARD LVCMOS15 [get_ports {a[1]}]
set_property IOSTANDARD LVCMOS15 [get_ports {a[0]}]

set_property PACKAGE_PIN AD10 [get_ports {b[3]}]
set_property PACKAGE_PIN AD11 [get_ports {b[2]}]
set_property PACKAGE_PIN Y12 [get_ports {b[1]}]
set_property PACKAGE_PIN Y13 [get_ports {b[0]}]
set_property IOSTANDARD LVCMOS15 [get_ports {b[3]}]
set_property IOSTANDARD LVCMOS15 [get_ports {b[2]}]
set_property IOSTANDARD LVCMOS15 [get_ports {b[1]}]
set_property IOSTANDARD LVCMOS15 [get_ports {b[0]}]

set_property PACKAGE_PIN N26 [get_ports led_clk]
set_property PACKAGE_PIN M26 [get_ports led_do]
set_property PACKAGE_PIN P18 [get_ports led_en]
set_property IOSTANDARD LVCMOS33 [get_ports led_clk]
set_property IOSTANDARD LVCMOS33 [get_ports led_do]
set_property IOSTANDARD LVCMOS33 [get_ports led_en]

set_property PACKAGE_PIN M24 [get_ports seg_clk]
set_property PACKAGE_PIN L24 [get_ports seg_do]
set_property PACKAGE_PIN R18 [get_ports seg_en]
set_property IOSTANDARD LVCMOS33 [get_ports seg_clk]
set_property IOSTANDARD LVCMOS33 [get_ports seg_do]
set_property IOSTANDARD LVCMOS33 [get_ports seg_en]
```

验证结果

LED 上能显示时钟信号，数码管上能显示从 00 到 1C、间隔为 4 的十六进制数，验证结果符合仿真波形。

总结与反思

重点与难点

- 本实验中要将之前所实现的部件组合在一起，需要数十根连线，有条理地将这些线路连接到合适的模块是正确实现单周期流水线的关键。在我的实验中，将顶层模块代码依照各模块来划分成若干部分，这样增强了代码的条理性，使得连线更为规范。

- 上板验证也是本实验较为耗时的部分。这是由于上板工程实现无法通过仿真来验证其正确性，在仿真过程中能正确输出的结果在上板时却不能输出；同时生成二进制代码也需要不少等待时间。要尽量减少调试时间，只能尽可能地多地将某些信号显示到板上，然后立足于符合预期的那些输出再去找问题。

待改进之处

上板的部分最好能输出处理器的运算结果。但是在我的实验过程中，使用各种方法始终无法将寄存器的数据显示在数码管上，只能输出 PC。目前我无从得知何处出现了问题，这是需要后续探索和研究的问题。