计算机系统结构试验 Lab05

518030910022 杨涵章

目录

1 实验概述：

1.1 实验名称

类 MIPS 单周期处理器的设计与实现

1.2 实验目的

1） 理解 CPU 的组成结构
2） 完成单周期的类 MIPS 处理器

1.3 实验步骤

1） 按照基本流程创建工程，需导入实验三四中实现的各个模块
2） 实现 Instruction memory, mux 等补充模块
3） 对之前实验中实现模块进行更改，使其能够整体运行
4） 创建顶层模块，对各个模块创建实例，用变量连接起来，组装模块为单周期处理器
5） 自主设计汇编语言程序并写出对应的二进制代码，自主设计内存数据
6） 进行模拟仿真，观察波形是否符合预期

2 实验描述：

这次实验需要设计顶层模块，整体思路为在顶层模块中对之前实现的模块实例化，增添新的补充模块，对之前实现的模块进行适当修改，在模块之间用信号线连接，组装成一个单周期处理器，并能够正确执行设计的 MIPS 指令代码。

2.1 补充模块

2.1.1 数据选择器

在顶层模块中可以用（？：）的三元运算符实现该功能，我这里选择使用模块化来实现，主要是因为在原理图中 mux 是作为模块的形式存在，模块化编程设计有助于顶层模块的创建。Mux 将两路数据和选择信号作为输入，根据选择信号输出一路数据。在顶层模块中包括 32 位（主要用于数据、地址等选取）和 5 位（主要用于寄存器的选取）的数据选择器，下面给出两者的代码实现，较为简单。

```
module mux32(
    input sel,
    input [31:0] in1,
    input [31:0] in0,
    output [31:0] out
    );

    assign out =sel?in1:in0;

endmodule
```

```
module mux5(
    input sel,
    input [4:0] in1,
    input [4:0] in0,
    output [4:0] out
    );

    assign out=sel?in1:in0;
endmodule
```

2.1.2 指令存储器

在本实验中，将数据和指令分开存储，因此这里需要实现一个指令存储器。同样这里可以使用之前的 data memory 模块，但由于指令存储器在实验流程中为只读存储器，无读写操作，因此重新实现一个较为简单的指令存储器，有助于顶层模块的建立，在这里我使用 32 位寻址，后续整体也采用 32 位寻址，未采用主流的按字寻址方式。实现代码见下

2.2 修改模块

2.2.1 寄存器

寄存器需要对 reset 信号做出相应的反应，将各个寄存器清零。我采用的是同步上升沿触发的清零方式，通过 for 循环对每个寄存器清零。最终实现一旦获取 reset 信号便进行寄存器清零操作。

下为模块代码，注意到为了避免出错，之前设计的寄存器在时钟下降沿进行赋值操作，

在这里我将 reset 操作放置在上升沿，使得整体协调运行，不会出错。

```
always @ (negedge clk)
    begin
        if (regwrute)
         regFile[writeReg] = writeData;
         end
  always @ (posedge clk)
    begin
        if(reset)
                begin
                    for(i=0;i<32;i=i+1)
                        regFile[i]<=0;
                end
         end
endmodule
```

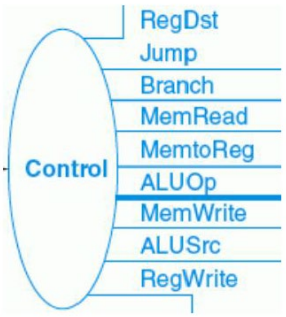
2.3 顶层模块

下面通过若干连线，将之前实现的模块连接起来。我才用的方式为按照原理图，将处理器分割开来，对于各个模块进行对应的实例化和连线处理，连线的时候注意线路命名的一致，我先确实线路的命名规范，然后分模块使用相同的命名方式命名，模块实例化和连线的效率较高。

除开各个部分的连线，顶层模块还需要统一的 clock 信号和 reset 信号，用于各个模块的实例化。下面为各个部分的代码实现。

2.3.1 控制信号

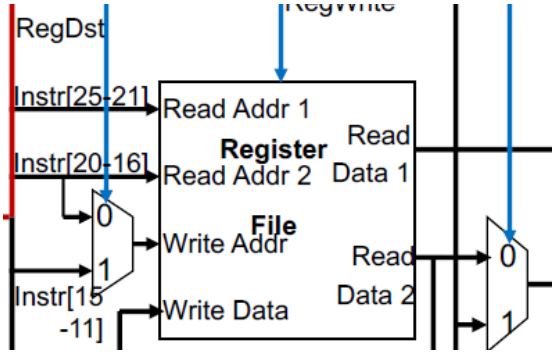
将实验 3 中的 Ctr 实例化即可，读入指令的高六位，输出控制信号。需要的连线和模块化实现中的连线一一对应，注意命名规范即可。下为模块实例化和连线代码。




```
    //CTR
    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),.brance(BRANCH),
                .memRead(MEM_READ),.memToReg(MEM_TO_REG),.aluOp(ALU_OP),
                .memWrite(MEM_WRITE),.aluSrc(ALU_SRC),.regWrite(REG_WRITE));
```

2.3.2 寄存器

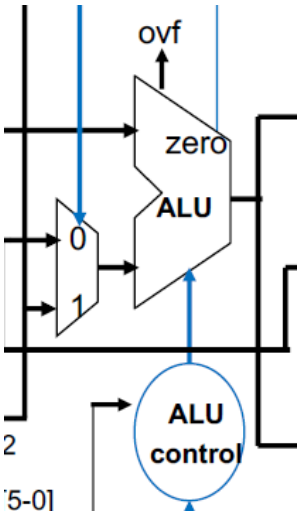
寄存器模块的实例化包括一个寄存器文件模块和一个寄存器选择器。根据 REG_DST 选择写入的目标寄存器，根据指令的 Rs、Rs 位确定读取的目标寄存器，执行读取和写入操作，同时相应 reset 充值信号。连线基本与右图原理图一致。下为模块实例化和连线代码。




```
39      //REG
40      wire [4:0] WRITE_REG;
41      wire [31:0] READ_DATA_1,READ_DATA_2,REG_WRITE_DATA;
42      mux5 writeRegMux(.sel(REG_DST),.in1(INST[15:11]),.in0(INST[20:16]),.out(WRITE_REG));
43      Registers regs(.clk(clk),.readReg1(INST[25:21]),.readReg2(INST[20:16]),
44                     .writeReg(WRITE_REG),.writeData(REG_WRITE_DATA),.reset(reset),
45                     .regwrute(REG_WRITE),.readData1(READ_DATA_1),.readData2(READ_DATA_2));
46
```

2.3.3 ALU

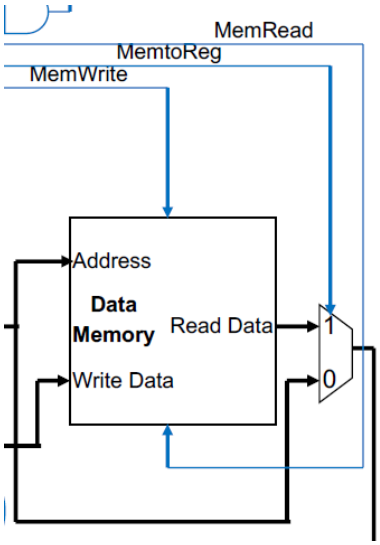
ALU 模块的实例化包括位扩展模块、ALUSrc 选择模块、ALUCtr 模块、ALU 模块。首先需要讲指令中的立即数（低 16 位）进行符号扩展，同时将 funct 和 ALUOp 送入 ALUCtr 进行译码，按照操作数选择器进行选择操作数。最后将输入信号送给 ALU 进行算术逻辑运算，输入运算结果和零标志位。下为模块实例化和连线代码。




```
47      //ALU
48      wire [31:0] IMM_SEXT,ALU_SRC_B,ALU_RESULT;
49      wire [3:0] ALU_CTR_OUT;
50      wire ZERO;
51      signext signext(.inst(INST[15:0]),.data(IMM_SEXT));
52      ALUCtr aluCtr(.funct(INST[5:0]),.aluop(ALU_OP),.aluctrout(ALU_CTR_OUT));
53      mux32 aluSrcMux(.sel(ALU_SRC),.in1(IMM_SEXT),.in0(READ_DATA_2),.out(ALU_SRC_B));
54      ALU alu(.input1(READ_DATA_1),.input2(ALU_SRC_B),.aluctr(ALU_CTR_OUT),.zero(ZERO),.alures(ALU_RESULT));
```

2.3.4 数据存储器

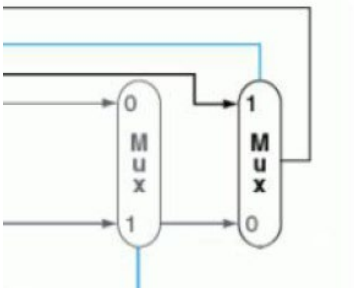
数据存储器的实例化包括数据存储器和数据写入选择器，根据之前的控制信号对应的接线实例化即可。下为模块实例化和连线代码。




```
56      //data memory
57      wire [31:0] MEM_READ_DATA;
58      dataMemory dataMem(.clk(clk),.address(ALU_RESULT),.writedata(READ_DATA_2),
59                         .memwrite(MEM_WRITE),.memread(MEM_READ),.readdata(MEM_READ_DATA));
60      mux32 regwriteMux(.sel(MEM_TO_REG),.in1(MEM_READ_DATA),.in0(ALU_RESULT),
61                        .out(REG_WRITE_DATA));
```

2.3.5 PC 和程序跳转逻辑

PC 是指令执行的关键，在每个周期，PC 都会根据当前的指令更新自己的值，按照这个值作为地址去指令存储器中获取对应的指令，同时将指令送入到整个处理器，完成该指令的处理。因此指令存储器的实例化较为简单。跳转逻辑包括条件转移选择器和无条件转移选择器。主要结构如图所示。



值得注意的是：顶层代码我采用的是 32 位寻址方式（指令和数据均是），PC+4 的操作在代码中等效于 PC+1，跳转逻辑也无需进行两位的移位操作（图中我保留了两位的移位操作是为了后续进行字寻址的调试，对于程序的巡行影响较小）。最后添加对于时钟信号的读取，在每个时钟的上升沿进行 reset 的反应和 PC 的更新。

```
27      //PC
28      reg [31:0] PC;
29      wire [31:0] INST;
30      InstMemory instMem(.address(PC),.instruction(INST));
```

```
63      //jump logic
64      wire [31:0] PC_PLUS_4,BRANCH_ADDR,SEL_BRANCH_ADDR,JUMP_ADDR,NEXT_PC,SEXT_SHIFT;
65      assign PC_PLUS_4=PC+1;
66      assign JUMP_ADDR= {PC_PLUS_4[31:28], INST[25:0]<<2};
67      assign SEXT_SHIFT=IMM_SEXT;
68      assign BRANCH_ADDR=PC_PLUS_4+SEXT_SHIFT;
69      mux32 brachchMux(.sel(BRANCH&ZERO),.in1(BRANCH_ADDR),.in0(PC_PLUS_4),
70                       .out(SEL_BRANCH_ADDR));
71      mux32 jumpMux(.sel(JUMP),.in1(JUMP_ADDR),.in0(SEL_BRANCH_ADDR),.out(NEXT_PC));
72
73      always @ (posedge clk)
74         begin
75             if(reset)
76                 PC<=0;
77             else
78                 PC<=NEXT_PC;
79         end
```

3 模拟仿真：

3.1 MIPS 指令编写

根据实现的单周期我设计了一个 MIPS 指令，并对照真值表写出对应的二进制文件和数据文件。指令数量适中，能较好地展现各种类型指令的运行情况，指令在工程文件根目录下已给出。同时在工程文件相对路径下已添加代码和数据文件，无需操作直接运行即可。

程序操作如下：

1） 分别在寄存器$1、$2、$3、$6 中载入数值 1、2、3、1.

2） 对寄存器中的操作数进行 add、sub、or、slt 操作，$1 中结果为 3，$2 中结果为 1，$4 中结果为 3，5 中结果为 0

3） 根据$2 和$6 的值判断是否跳转到 step 中去，如果跳转则将$4 中的值储存到内存 1 中，否则储存到内存 4 中。这里满足跳转逻辑。

4） 无条件跳转到 START

```
START:
lw  $1, 0($0)
lw  $2, 4($0)
lw  $3, 8($0)
lw  $6, 0($0)
add $1, $1, $2
sub $2, $2, $6
or  $4, $1, $2
slt $5, $3, $4
beq $2, $6, step
sw  $4, 4($0)
step:
sw  $4, 16($0)
j   START
```

Figure 1 汇编程序

```
10001100000000010000000000000000
10001100000000010000000000000001
10001100000000011000000000000010
10001100000000011000000000000000
00000000001000100000100000100000
00000000001000110000100000100010
00000000001000100010000000100101
00000000011001000010100000101010
00010000010001100000000000000001
10101100000000100000000000000001
10101100000000100000000000000100
00001000000000000000000000000000
```

Figure 2 二进制指令程序

```
00000001
00000002
00000003
00000004
```

Figure 3 数据文件

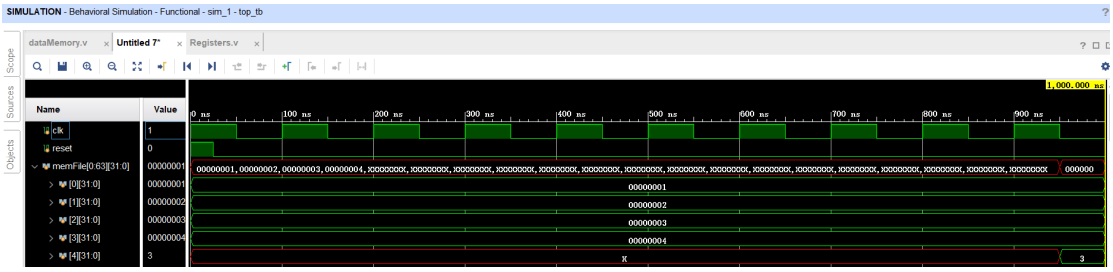
3.2 仿真程序

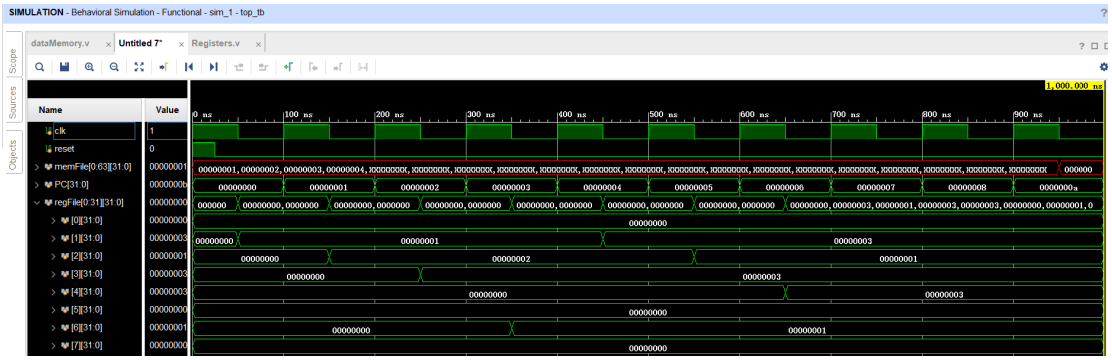
仿真程序中设置时钟信号，读取相对路径下的文件（这里我已经将文件放置在相对路径下）。设置 reset 信号，模拟运行即可。

```
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.instFile);
    clk = 1;
    reset = 1;
    #25
    reset = 0;
    #2000;
end

endmodule
```

3.3 仿真结果

下面展示了数据存储器和寄存器内值的变化，可以看到程序的运行结果和预期的指令结果一致，通过 PC 和内存中值的变化也可以看到跳转指令能够正确执行。经过调试运行，程序能够正确运行，并能够模拟得到自主设计的 MIPS 指令的波形，实验结果良好，满足预期，实验较为成功。





4 总结与反思：

这次实验难度较大，主要考察了对于单周期处理器结构的认识。与之前单个模块的设计不同，这次实验要求将之前的模块组合起来，完成单周期处理器。通过这次实验，进一步加深了对于单周期处理器的了解，同时对于模块较大的程序的调试方法也有了一定的了解，对于实验六也有较大的帮助。

本实验主要难点包括各部分的连接，适应整体操作对于各个模块的微调。其中较难实现的是取指模块和程序跳转模块，我刚开始一直按照书本中的 PC+4 和跳转指令的移位操作进行处理，但在实际操作过程中忽略了书本中的处理器按照字节进行寻址，我之前实现的模块和指令的书写均为 32 位寻址。后来发现之后进行更改。

其次便是程序的调试较为困难，需要将各个部分的指令信号放入波形窗口中，跟踪信号的变化，进而对程序做出微调。

最后便是控制信号连线的命名，起初我采用的是对照各个模块分开命名。后来发现在操作过程中容易出错，导致连线断裂。后续我改变命名策略，先找出所有的连线，命名之后再进行各个模块的实例化，这样效率较高，准确率较高。

这次设计的单周期处理器仍有较多值得改进的地方。首先便是寻址位数，按照手册的操作统一是按照 32 位进行寻址，但在实际的处理器中，大多按照 8 位进行寻址，因此实现 8 位的处理器能够更加贴合学到的知识，实现起来更易于理解。但 8 位寻址则需要对指令存储器和数据存储器以及跳转逻辑进行改动，实现较为简单，便不再展示。