计算机系统结构实验报告

Lab05

类 MIPS 单周期处理器的设计与实现

姓　　名：　徐薪

学　　号：　519021910726

日　　期：　2021 年 6 月 09 日

摘 要

本实验要求实现类 MIPS 单周期处理器。该类 MIPS 单周期处理器的 CPU 支持 9 条 MIPS 指令（包括 R 型指令中的 add、sub、and、or、slt；I 型指令中的 lw、sw、beq；J 型指令中的 j）。本实验的设计基础为实验三和实验四中已完成的相关模块。最后，本实验通过软件仿真的方式进行结果验证。

目录

1. 实验概述

1.1 实验内容

本实验要求实现类 MIPS 单周期处理器。该类 MIPS 单周期处理器的 CPU 支持 16 条 MIPS 指令（包括 R 型指令中的 add、sub、and、or、slt、sll、srl、jr；I 型指令中的 lw、sw、addi、ori、beq；J 型指令中的 j、jal）。本实验的设计基础为实验三和实验四中已完成的相关模块。最后，本实验通过软件仿真的方式进行结果验证。

1.2 实验目的

（1）理解类 MIPS 单周期处理器的工作原理；

（2）实现类 MIPS 单周期处理器的各个功能模块以及顶层模块；

（3）使用功能仿真对实验结果进行验证。

2. 原理分析

2.1 主控制器 Ctr 的设计

2.1.1 Ctr 模块描述

主控制器单元（Ctr）的输入为 MIPS 指令的 opCode 字段。Ctr 的作用是将操作码译码，并向运算单元控制器（ALUCtr）、数据内存（Data Memory）、寄存器（Register）、数据选择器（MUX）等部件输出正确的控制信号。

Ctr 的工作模式如下所示：

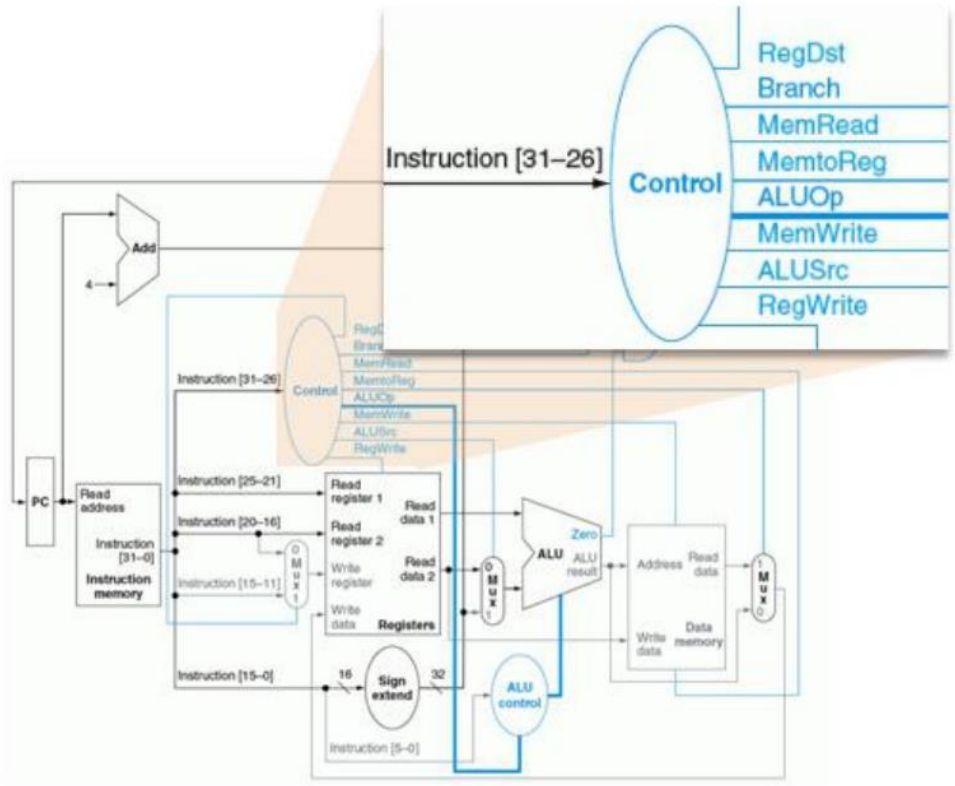


图 1. 主控制模块的 IO 定义

其中，根据输入的指令 instruction[31:26]，可以将 MIPS 指令做如下分类：

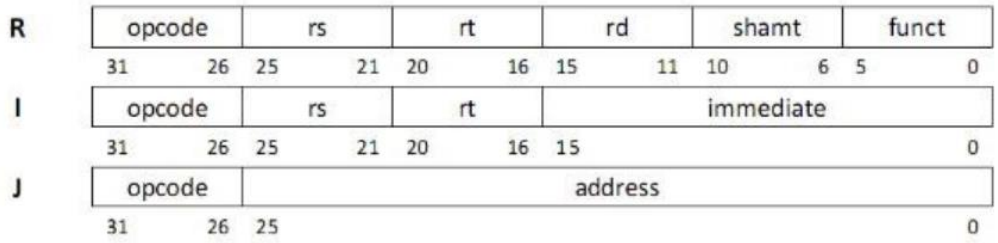


图 2. MIPS 基本指令格式

Ctr 输出的控制信号的含义为：

信号名称	具体含义
RegDst	目标寄存器选择（0：rt，1：rd）
Branch	跳转使能信号（高电平有效）

MemRead	内存读使能（高电平有效）
MemtoReg	内存读取内容写回寄存器信号（高电平有效）
ALUOp	ALU 需执行的操作类型
MemWrite	内存写使能（高电平有效）
ALUSrc	ALU 第二操作数选择信号（0：rt，1：立即数）
RegWrite	寄存器写使能（高电平有效）

表 1. 主控制模块输出的控制信号类型

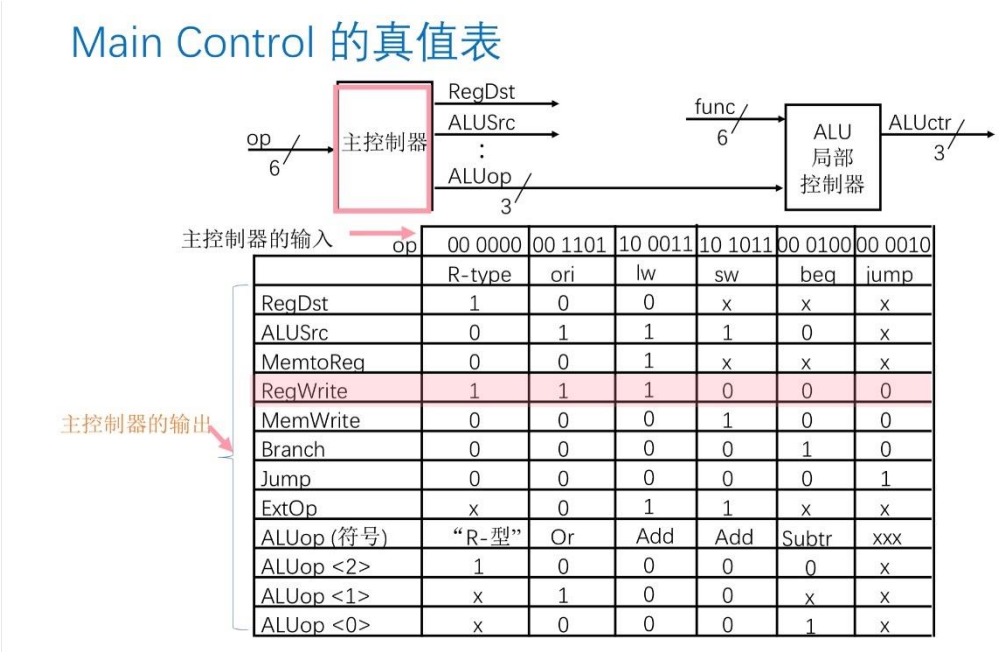
2.1.2 Ctr 模块编码与译码

MIPS 指令主要有 R 型、I 型与 J 型之分，可通过 opCode 进行判别。

指令类型	opCode
R 型：add，sub，and，or，slt	000000
I 型：lw	100011
I 型：sw	101011
I 型：beq	000100
J 型：j	000010

表 2. MIPS 指令的 opCode 编码规则

主控制器 Ctr 再将 opCode 转化为控制信号的译码规则如下图所示：



主控制器的输入 op	00 0000 R-type	00 1101 ori	10 0011 lw	10 1011 sw	00 0100 beq	00 0010 jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (符号)	“R-型”	Or	Add	Add	Subtr	xxx
ALUop <2>	1	0	0	0	0	x
ALUop <1>	x	1	0	0	x	x
ALUop <0>	x	0	0	0	1	x

主控制器的输出

图 3. 主控制模块的译码规则

2.2 运算单元控制器模块 ALUCtr

2.2.1 ALUCtr 模块描述

ALU 的控制器模块(ALUCtr）是根据主控制器输出的控制信号 ALUOp 来判断指令类型的。

特别地，对于 R 型指令，由于一个 opCode 可能对应多种运算指令，单凭 opCode 译码而获得 的 ALUOp 尚不足以确定 ALU 操作类型。因此 R 型指令还需根据指令的后 6 位（funct）进一步区分。ALUCtr 需综合这两处输入，以得到正确的控制信号来控制 ALU 执行正确的操作。

其中 ALUCtr 的工作原理如下图所示：

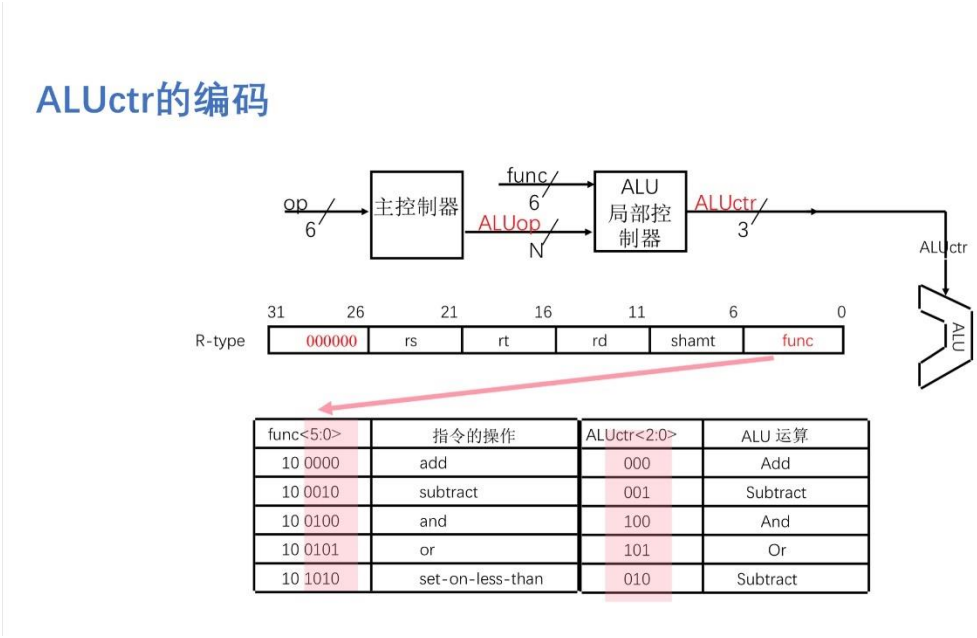


图 4. ALUCtr 的编码

2.2.2 ALUCtr 模块编码与译码

ALUCtr 将 ALUOp 以及 funct 翻译为 ALUctrOut 信号：

图 5. ALUCtr 的 译码规则

2.3 算术逻辑单元 ALU

2.3.1 ALU 模块描述

算术逻辑单元 ALU 根据 ALUCtr 信号将两个输入执行对应的操作，ALURes 为输出结果。若做减法操作，当 ALURes 结果为 0 时，则 Zero 输出置为 1。

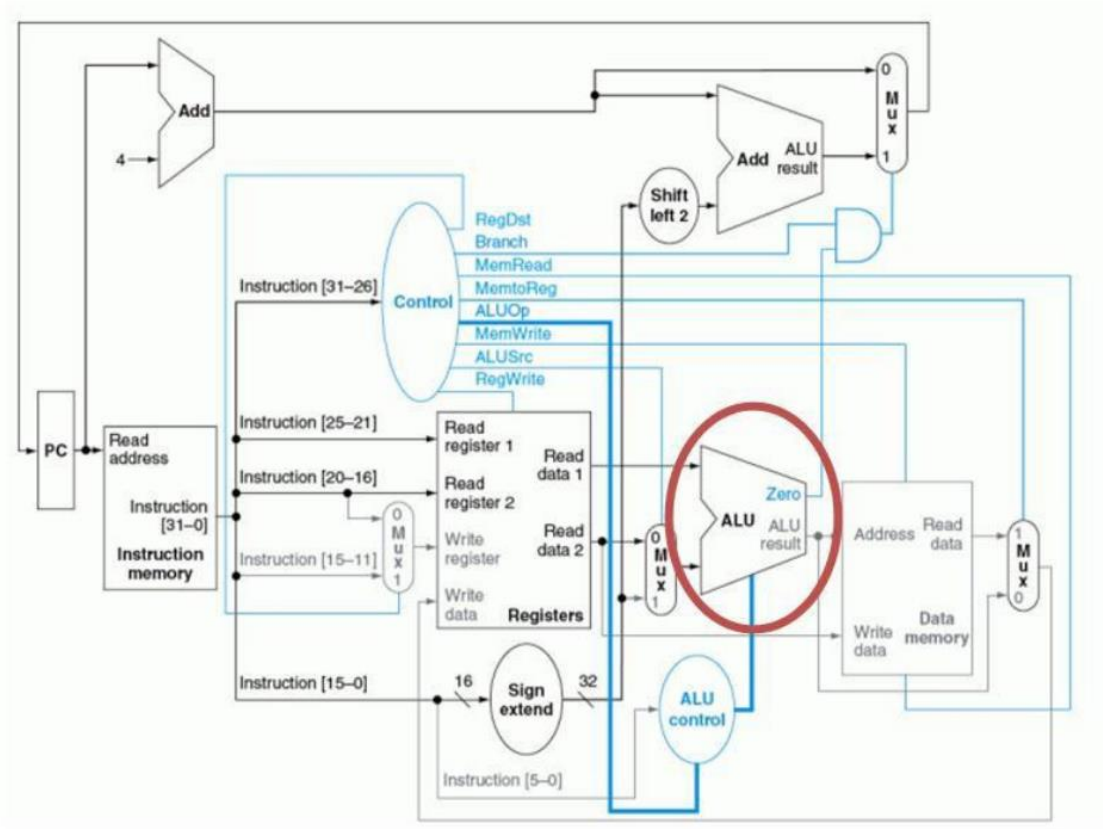


图 6. ALU 模块

2.3.2 ALU 模块编码与译码

ALUctrOut 对应不同的 ALU 功能：

ALUctrOut[3:0]	ALU 功能
0000	与运算

0001	或运算
0010	加法运算
0110	减法运算
0111	小于时置位

表 3.ALUctrOut 编码规则

2.4 寄存器 Register

寄存器（Register）的功能是暂时存储一些数据以及中间运算结果，其主要接口如图 7 所示。寄存器单元内部一共有 32 个寄存器，所以需要五位的二进制数来确定选中的目标寄存器。由于 MIPS 处理器的存储数据都是 32 位的，所以寄存器输入输出的 data bus 都是 32 位的。除此之外，寄存器还有两个控制信号：时钟 CLK 和写操作 RegWrite。其中，CLK 信号用于控制寄存器的读写，我们要求寄存器在 CLK 发生变化时都能读，在时钟下跳沿写，这样设计能很好地解决多周期 MIPS 处理器不同阶段的寄存器读写冲突问题；RegWrite 信号控制是否进行写入寄存器的操作。

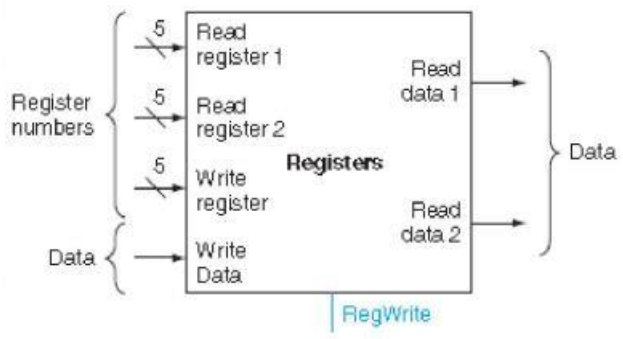


图 7. 寄存器（Register）的图示

2.5 存储器模块 Data Memory

存储器（Data Memory）的功能是存储大量内存数据并支持数据的读写，其示意图如图 8 所示。MIPS 处理器能够支持 32 位的数据地址单元，所以 address 接口是 32 位的 input wire。由于 MIPS 支持存储的内存数据的大小是 32 bit，所以 Write data 输入接口和 Read data 输出接口都是 32 位的。除此之外，存储器还有三个控制信号：CLK、MemWrite 以及 MemRead。其中，时钟 CLK 上升沿支持存储器读，下跳沿支持存储器写，这样通过硬件的方式，能够解决 MIPS 多周期处理器在不同阶段同时用到存储器的冒险冲突问题。MemWrite 信号控制数据是否写入存储器，MemRead 信号控制是否从存储器中读取数据。

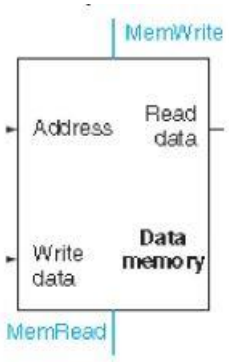


图 8. 存储器（Data Memory）的图示

2.6 有符号扩展单元 Signext

有符号扩展单元（Signext）的功能是将指令中的 16 位有符号立即数扩展成为 32 位有符号立即数。由于是带符号扩展，所以只需要将 16 位有符号立即数的符号位扩展到 32 位有符号立即数的高 16 位即可。这个功能可以用数字逻辑运算来进行实现，在最终的代码实现

过程中，我将它设计为了一个 module。

2.7 指令存储器 InstMemory

指令存储器 InstMemory 的原理与存储器 Data Memory 相似，只不过 InstMemory 不支持写操作，只能读取输入地址所对应的指令。

2.8 数据选择器 Mux

数据选择器 Mux 的原理是根据控制信号，从两个输入里面选择对应的一个进行输出，Mux 的示意图如图 9 所示。数据选择器的实现很简单，一个三目运算符即可：

```
Assign OUT = SEL ? INPUT1 : INPUT2;
```

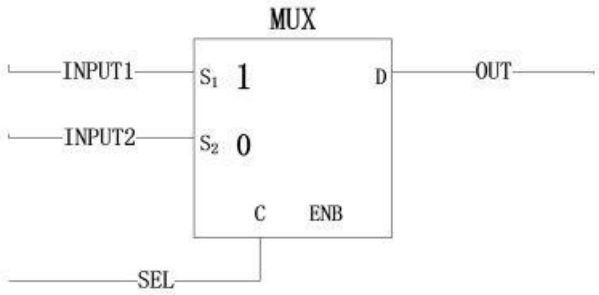


图 9. 数据选择器（Mux）的原理图

2.9 程序计数器 PC

程序计数器的功能是输出下一个指令的地址。它有两个控制信号：CLK 和 Reset。其中，CLK 信号控制 PC 何时读取下一条指令的地址，在我的实现代码里，PC 在 CLK 上升沿读取地址；Reset 信号控制何时将 PC 计数器置零，当 Reset 信号为 1 时，PC 的输出置为零。因为有多个控制信号，我将它设计为一个 module。

2.10 顶层模块 Top

顶层模块 Top 的功能是将之前单独设计的 MIPS 功能模块有机组合在一起，使之能合作完成类 MIPS 单周期处理器的功能。MIPS 单周期处理器的电路设计图如图 10 所示。

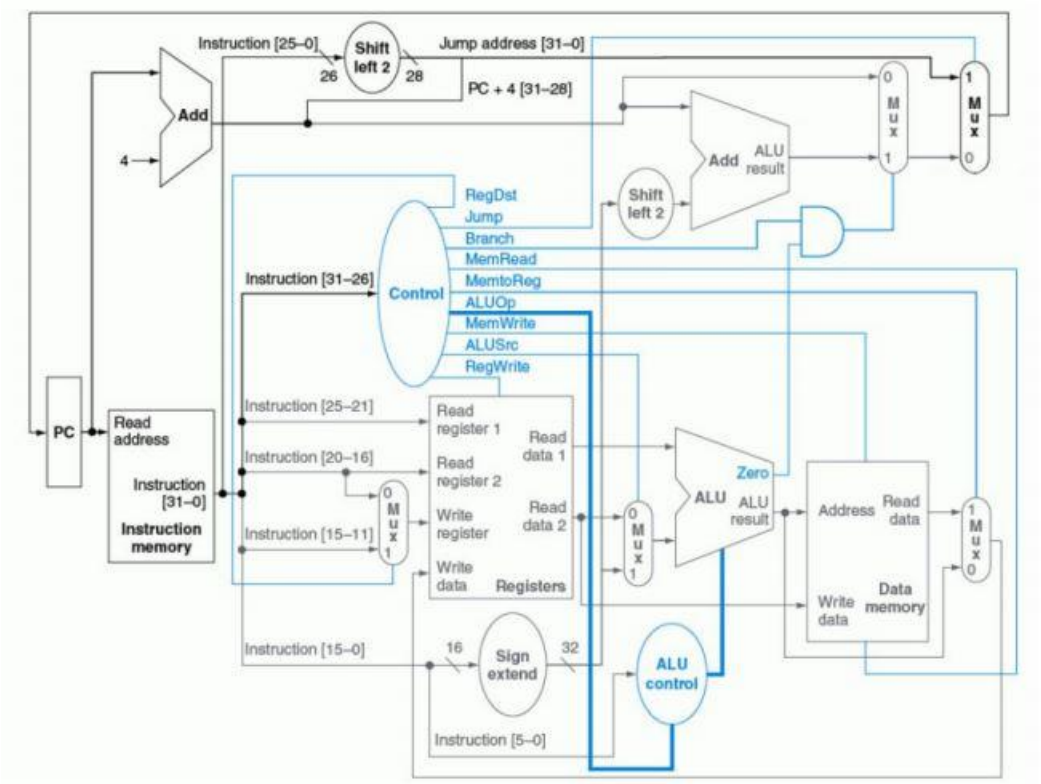


图 10. 类 MIPS 单周期处理器的原理图

3. 功能实现

这里只展示了新写的 module 的实现代码：InstMemory、Mux（输入位宽为 32 bit）、Mux5（输入位宽为 5 bit）、PC、Top，其余模块（如 Ctr、ALU、ALUCtr 等模块）的代码在实验三和实验四中已展示，这里不再赘述。

3.1 指令存储器模块 InstMemory

3.1.1 InstMemory 模块实现代码

```
module InstMemory(
    input [31:0] instAddress,
    output [31:0] inst
    );

    reg [31:0] instFile[0:63];
    reg [31:0] Inst;

    always@(instAddress)
        Inst = instFile[instAddress / 4];

    assign inst = Inst;

endmodule
```

3.2 数据选择器模块 Mux

3.2.1 Mux 模块实现代码

```
module Mux(
    input ctrSignal,
    input [31:0] input1,
    input [31:0] input2,
    output [31:0] result
    );

    assign result = ctrSignal ? input1 : input2;

endmodule
```

3.2.2 Mux5 模块实现代码

```
module Mux5(
    input ctrSignal,
    input [4:0] input1,
    input [4:0] input2,
    output [4:0] result
    );

    assign result = ctrSignal ? input1 : input2;

endmodule
```

3.3 程序计数器 PC

3.3.1 PC 模块实现代码

```
module PC(
    input RESET,
    input Clk,
    input [31:0] inAddress,
    output [31:0] outAddress
    );

    reg [31:0] OutAddress;
    reg [1:0] count;

    initial begin
    OutAddress = 0;
    count = 0;
    end

always@(posedge Clk or RESET)
    begin
        if(RESET)
            OutAddress = 0;
        else
        begin
            count = count + 1;
            if (count == 2)
            begin
            OutAddress = inAddress;
            count = 0;
            end
        end
    end

    assign outAddress = OutAddress;

endmodule
```

3.4 顶层设计模块 Top

3.4.1 Top 模块实现代码

```
module Top(
    input RESET,
    input CLK
    );

    wire REG_DST,
```

```
    JUMP,
    BRANCH,
    MEM_READ,
    MEM_TO_REG,
    MEM_WRITE;
    wire [1:0] ALU_OP;
    wire ALU_SRC,
    REG_WRITE;
    wire [31:0] INST;
    wire [31:0] INST_ADDRESS;
    wire [31:0] IN_ADDRESS;
    wire [4:0] WRITE_REG;
    wire [31:0] WRITE_DATA;
    wire [31:0] REG_READ_DATA1;
    wire [31:0] REG_READ_DATA2;
    wire [31:0] SIGNEXT_OUT;
    wire [31:0] ALU_INPUT;
    wire ZERO;
    wire [31:0] DATA_MEM_ADDRESS;
    wire [3:0] ALUCTR_OUT;
    wire [31:0] dataMem_READ_DATA;
    wire [31:0] PC4;
    wire [31:0] addALU_OUT;
    wire [31:0] muxToMux_OUT;


    Ctr mainCtr(
        .opCode(INST[31:26]),
        .regDst(REG_DST),
```

```
        .aluSrc(ALU_SRC),
        .memToReg(MEM_TO_REG),
        .regWrite(REG_WRITE),
        .memRead(MEM_READ),
        .memWrite(MEM_WRITE),
        .branch(BRANCH),
        .aluOp(ALU_OP),
        .jump(JUMP)
    );

    InstMemory instMemory(
        .instAddress(INST_ADDRESS[31:0]),
        .inst(INST[31:0])
    );

    PC Pc(
        .RESET(RESET),
        .Clk(CLK),
        .inAddress(IN_ADDRESS[31:0]),
        .outAddress(INST_ADDRESS[31:0])
    );

    Mux5 muxWriteReg(
        .ctrSignal(REG_DST),
        .input1(INST[15:11]),
        .input2(INST[20:16]),
        .result(WRITE_REG[4:0])
    );
```

```
Registers register (
        .Reset(RESET),
        .Clk(CLK),
        .readReg1(INST[25:21]),
        .readReg2(INST[20:16]),
        .writeReg(WRITE_REG),
        .writeData(WRITE_DATA),
        .regWrite(REG_WRITE),
        .readData1(REG_READ_DATA1),
        .readData2(REG_READ_DATA2)
);

signext SignExt(
        .inst(INST[15:0]),
        .data(SIGNEXT_OUT)
);

Mux muxAluInput (
        .ctrSignal(ALU_SRC),
        .input1(SIGNEXT_OUT),
        .input2(REG_READ_DATA2),
        .result(ALU_INPUT)
);

ALU Alu(
        .input1(REG_READ_DATA1),
        .input2(ALU_INPUT),
        .aluCtr(ALUCTR_OUT[3:0]),
        .zero(ZERO),
```

```
        .aluRes(DATA_MEM_ADDRESS[31:0])
    );

    ALUCtr ALUCTR(
        .aluOp(ALU_OP),
        .funct(INST[5:0]),
        .aluCtrOut(ALUCTR_OUT[3:0])
    );

    dataMemory dataMem(
        .Clk(CLK),
        .address(DATA_MEM_ADDRESS[31:0]),
        .writeData(REG_READ_DATA2),
        .memWrite(MEM_WRITE),
        .memRead(MEM_READ),
        .readData(dataMem_READ_DATA)
    );

    Mux muxWriteData(
        .ctrSignal(MEM_TO_REG),
        .input1(dataMem_READ_DATA),
        .input2(DATA_MEM_ADDRESS[31:0]),
        .result(WRITE_DATA)
    );

    ALU PCadder(
        .input1(INST_ADDRESS),
        .input2(4),
        .aluCtr(4'b0010),
```

```
        //.zero(0),
        .aluRes(PC4)
);

/*
always @ (posedge CLK)
        PC4 <= INST_ADDRESS + 4;
*/

ALU addALU(
        .input1(PC4),
        .input2(SIGNEXT_OUT << 2),
        .aluCtr(4'b0010),
        //.zero(0),
        .aluRes(addALU_OUT)
);

Mux muxToMux(
        .ctrSignal(BRANCH & ZERO),
        .input1(addALU_OUT),
        .input2(PC4),
        .result(muxToMux_OUT)
);

Mux muxToPC(
        .ctrSignal(JUMP),
        .input1((INST[25:0] << 2) + (PC4 & 32'hF0000000)),
        .input2(muxToMux_OUT),
        .result(IN_ADDRESS[31:0])
```

```
    );


endmodule
```

4. 仿真测试

我们首先在内存中载入如下数据：

```
00000000
00000001
00000002
00000003
00000004
00000005
00000006
00000007
00000008
00000009
0000000A
0000000B
0000000C
0000000D
0000000E
0000000F
00000010
00000011
00000012
00000013
00000014
00000015
```

```
00000016
00000017
00000018
00000019
0000001A
0000001B
0000001C
0000001D
0000001E
0000001F
```

并用以下命令读取内存文件：

$readmemh("F:/archlabs/lab05/mem_data.dat",MIPS.dataMem.memFile);

同时，设计如下汇编指令进行测试：

```
10001100000000010000000000001000 // lw $1,8($0) $1 = 8
10001100000000010000000000000000 // lw $1,0($0) $1 = 0
10001100000000010000000000000001 // lw $2,1($0) $2 = 1
10001100000000011000000000000010 // lw $3,2($0) $3 = 2
00000000001000100010000000100000 // add $4 = $1 + $2 = 1
00000000011000010010100000100010 // sub $5 = $3 - $1 = 2
00000000010000110011000000100100 // and $6 = $2 and $3 = 0
00000000010000110011100000100101 // or $7 = $2 or $3 = 3
00000000110001110100000000101010 // slt $6,$7,$8    $8 = 1
10101101000001110000000000011110 // sw $7,30($8) mem_data[31] = $7 = 3
00010000010010000000000000000001 // beq $2,$8,1. if $2==$8 go to jump inst
00000000000000000000000000000000 // nop
00001000000000000000000000000001 // jump to inst1
00000000000000000000000000000000 // nop
00000000000000000000000000000000 // nop
```

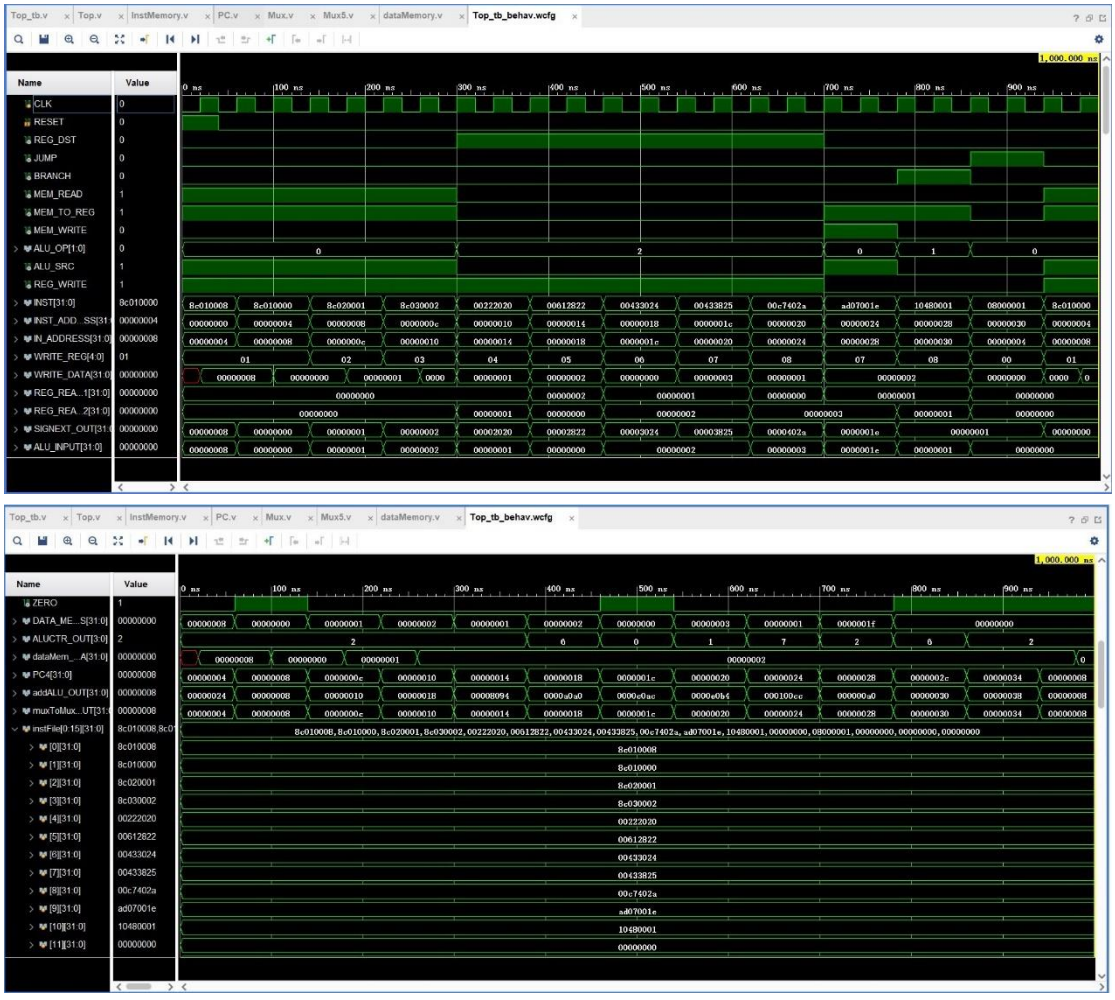
```
00000000000000000000000000000000 // nop
```

用以下指令读取汇编指令文件：

$readmemb("F:/archlabs/lab05/mem_inst.dat",MIPS.instMemory.instFile

);

基于以上文件，我们进行仿真测试，测试结果如图 10 所示。



从仿真结果可以看出，我们的类 MIPS 单周期处理器实现成功。

5. 实验总结

5.1 实验评价

本实验实现了一个类 MIPS 单周期处理器。本实验的基础是实验三和实验四的相关模块，同时还增加了一些新的模块，如 PC、Mux（Mux5）、InstMemory 等。最后我们用顶层设计模块 Top 将这些功能部件连接起来，成功实现了类 MIPS 单周期处理器的功能，并得到了正确的仿真结果。

5.2 实验心得

这一次的实验比之前的实验都要难，有各种各样的问题发生。我在这个实验所花费的时间是六个实验里面最长的。

这次的实验我主要有两个收获。一是我对 Verilog 里面的模块思想有了更加深刻的理解。模块之上还可以再定义模块，类 MIPS 单周期处理器就可以看成是一个巨大的模块。在设计实现 Top 模块时，我对 MIPS 的整个处理流程有了更加清晰的认识，这将帮助我理解 MIPS 多周期处理器的工作流程，同时也让我更好地学习计算机系统这门课程。二是我深刻明白了不同指令在不同阶段的处理时间不一样这个问题。在最开始的仿真测试中，我发现数据还没来得及写入存储器里面，下一条指令就开始读取数据了。我尝试了延长一个周期的时间，发现并不能很好地解决问题。后来在朋友的帮助下，我意识到错误在于存储器写的时候下一个时钟上升沿就来到了，从而阻碍了存储器的读写。最终，我把程序改为每两个时钟上升沿读取下一条指令，问题就得到了解决。