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

姚宣骋 520021910431

2022 年 3 月 30 日

摘要

本实验基于实验 3、4 的实验结果，将已经实现的 ALU、Registers、Data Memory 等模块进行修改，并新实现 PC、Instruction Memory、Mux 等模块，并将这些模块按照正确的方式连接，实现了类 MIPS 单周期处理器。本实验中的处理器在指导书的基础上做出扩展，支持 16 条 MIPS 指令的处理，包括 R 型指令：add、sub、or、and、slt、sll、srl、jr；I 型指令：lw、sw、addi、ori、beq；J 型指令：j、jal。通过软件仿真运行指令，验证实验结果。

目录

1 实验目的

本次实验有 3 个目的：

- 理解单周期的类 MIPS 处理器的运行逻辑和方式；
- 完成单周期的类 MIPS 处理器的实现；
- 设计支持 16 条 MIPS 指令的处理器；

2 原理分析

2.1 主控制器（Ctr）原理分析

主控制器会对指令的最高 6 位的 OpCode 进行解析，初步判定指令的类型并产生相应的处理器信号。在 Ctr 中，OpCode 可以大致分为 R 型指令；I 型指令中的 load 指令（lw）、store 指令（sw）与 branch 指令（beq）；J 型指令中的 jump 指令（j）。控制信号则如表 1 所示。

其中 ALUOp 信号为两个二进制为的信号，其含义比一般信号复杂，如表 2 所示。此信号还需要经过 ALUCtr 处理后送入 ALU，才能对 ALU 进行控制。

OpCode 指令与控制信号的对应关系如表 3 所示一一对应。若出现其余未设定指令，我们先暂且将所有控制信号置 0，即为空指令（nop）。

表 1: Ctr 控制信号

信号	具体说明
ALUSrc	ALU 的第二个操作数来源（0：使用 rt；1：使用立即数）
ALUOp（*）	发送给 ALUCtr 用来进一步解析运算类型的控制信号
Branch	条件跳转信号，高电平说明当前指令是条件跳转指令 (branch)
Jump	无条件跳转信号，高电平说明当前指令是无条件跳转指令 (jump)
memRead	内存读使能信号，高电平说明当前指令需要进行内存读取 (load)
memToReg	写寄存器的数据来源 (0: 使用 ALU 运算结果；1: 使用内存读取数据)
memWrite	内存写使能信号，高电平说明当前指令需要进行内存写入 (store)
regDst	目标寄存器的选择信号 (0: 写入 rt 代表的寄存器；1: 写入 rd 代表的寄存器)
regWrite	寄存器写使能信号，高电平说明当前指令需要进行寄存器写入

表 2: ALUOp 信号具体含义与解析方式

ALUOp 的信号内容	指令	具体说明
101	R	ALUCtr 结合指令 Funct 段决定最终操作
000	lw,sw,addi,addiu	ALU 执行加法
001	beq	ALU 执行减法
011	andi	ALU 执行逻辑与
100	ori	ALU 执行逻辑或
111	xori	ALU 执行逻辑异或
010	slti	ALU 执行带符号数大小比较
110	sltiu	ALU 执行无符号数大小比较

表 3: OpCode 指令与控制信号的对应关系

OpCode	指令	aluOp	aluSrc	memRead	memToReg	memWrite	regDst	regWrite	extSign	branch	jump	jalSign
000000	R 型指令	101	0	0	0	0	1	1	0	0	0	0
100011	lw	000	1	1	1	0	0	1	1	0	0	0
101011	sw	000	1	0	0	1	0	0	1	0	0	0
000100	beq	001	0	0	0	0	0	0	1	1	0	0
000010	j	101	0	0	0	0	0	0	0	0	1	0
000011	jal	101	0	0	0	0	0	1	0	0	1	0
001000	addi	000	1	0	0	0	0	1	1	0	0	0
001001	addiu	000	1	0	0	0	0	1	0	0	0	0
001100	andi	011	1	0	0	0	0	1	0	0	0	0
001010	xori	111	1	0	0	0	0	1	1	0	0	0
001101	ori	100	1	0	0	0	0	1	1	0	0	0
001010	slti	010	1	0	0	0	0	1	1	0	0	0
001011	sltiu	110	1	0	0	0	0	1	0	0	0	0

2.2 运算单元控制器（ALUCtr）原理分析

运算单元控制器（ALUCtr）对 ALUOp 信号和指令后 6 位 Funct 进行解析，给出控制 ALU 的控制信号 ALUCtrOut。输入与输出的对应关系如表 4 所示。此外，为了支持 jr 和 jal 指令的实现，ALUCtr 还负责产生控制

表 4: 运算单元控制器（ALUCtr）的解析方式

指令	ALUOp	Funct	ALUCtrOut	具体说明
add	1x	100000	0010	ALU 执行加法运算
sub	1x	100010	0110	ALU 执行减法运算
and	1x	100100	0000	ALU 执行逻辑与运算
or	1x	100101	0001	ALU 执行逻辑或运算
slt	1x	101010	0111	ALU 执行小于时置位运算
lw	00	xxxxxx	0010	ALU 执行加法运算
sw	00	xxxxxx	0010	ALU 执行加法运算
beq	01	xxxxxx	0110	ALU 执行减法运算
j	00	xxxxxx	0010	ALU 执行加法运算

立即数输入的 shamtSign 信号和控制 jr 指令的 jrSign 信号。

2.3 算术逻辑运算单元（ALU）原理分析

ALU 根据 ALUCtr 产生的控制信号 ALUCtrOut 对两个输入数进行对应的算数逻辑运算，并且输出运算结果以及部分控制信号（zero）。该信号用于与 branch 指令结合判断是否满足转移条件。ALU 执行的算术逻辑运算类型与运算单元控制信号 ALUCtrOut 的对应方式如表 5 所示。

表 5: ALU 的算术逻辑运算类型与 ALUCtrOut 的对应方式

ALUCtrOut	ALU 执行算数逻辑运算类型
0000	逻辑与 (and)
0001	逻辑或 (or)
0010	加法 (add)
0110	减法 (sub)
0111	小于时置位 (slt)
1100 (*)	逻辑或非 (nor)

2.4 寄存器（Register）原理分析

寄存器（Register）的功能是暂时存储少量的数据，需要同时支持双通道数据的读取、写入功能。该模块内部共有 32 个寄存器，故两个读取地址 Readregister1、Readregister2 和一个写入地址 Writeregister 都是 5 位 2 进制数。此实验中，寄存器内存储的数据是 32 位的二进制数，故写入数据接口 WriteData 和两个读取数据接口 ReadData1、ReadData2 都是 32 位二进制数。寄存器的主要接口如图 1 所示。由于不确定 WriteReg,WriteData,RegWrite 信号的先后次序，我们采用时钟（样例里命名为 Clk）的下降沿作为写操作的同步信号，防止发生错误。

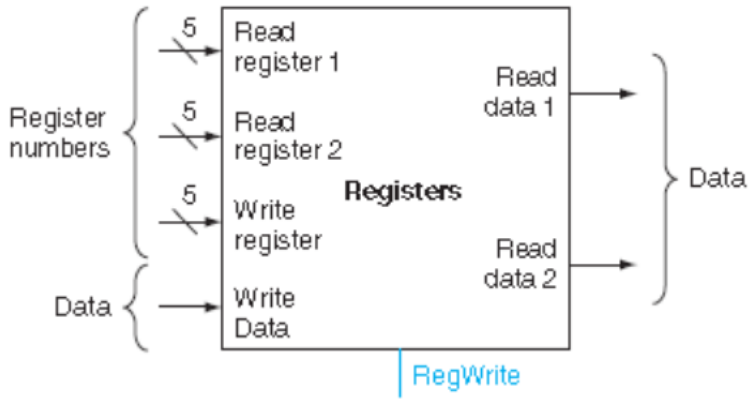

图 1: 寄存器（Register）的主要接口

2.5 数据存储器（Data Memory）原理分析

存储器（Data Memory）的功能是存储大量的数据，支持数据读取、数据写入的功能。该模块比寄存器大，共能存储 64 个 32 位二进制数。因此地址（Address）是 64 位数。而写入数据接口 Writedata 和读取数据接口 Readdata 都是 32 位二进制数。同时为了防止访问错误，需要对地址访问进行判断，防止给定地址超出存储器范围（0 至 63）。存储器的主要借口如图 2 所示。同寄存器，我们也采用时钟（样例里命名为 Clk）的下降沿作为写操作的同步信号，防止发生错误。

2.6 带符号扩展单元（Sign Extension）原理分析

带符号扩展单元（Sign Extension）的功能是将指令中的 16 位立即数拓展为 32 位立即数。因此只需要根据立即数扩展规则，将这个 16 位带符号立即数的符号位填充在 32 位带符号立即数的高 16 位，再将低 16 位复制到 32 位带符号立即数的低 16 位即可。此外，我们用 signExt 的一位二进制信号决定立即数是否带符号。其主要接口如图 3 所示。

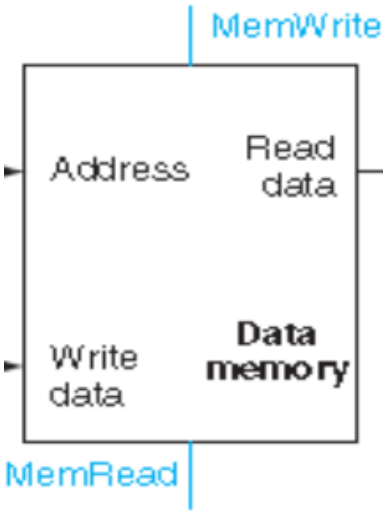

图 2: 数据存储器（Data Memory）的主要接口

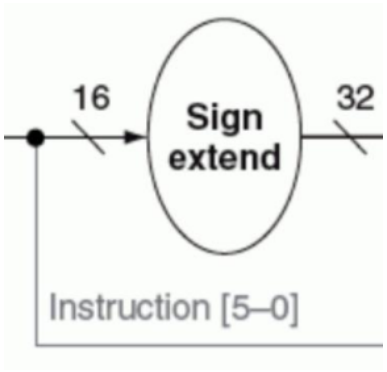

图 3: 带符号扩展单元（Sign Extension）的主要接口

2.7 数据选择器（Mux/RegMux）原理分析

数据选择器的功能即是接受两个输入信号，根据一个选择信号将其中一个输入信号输出。在本实验中，我们有 32 位选择器 Mux 和 5 位选择器 RegMux，分别用于数据和寄存器选取信号的选择。电路模型如图 4 所示。

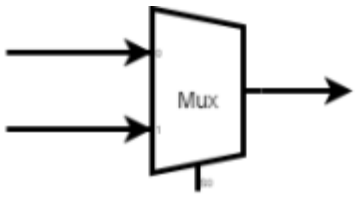

图 4: 数据选择器（Mux/RegMux）的主要接口

2.8 指令存储器 (Instruction Memory) 原理分析

本实验中处理器采取哈佛架构，即指令内存和数据内存相互分离。指令存储器的功能比数据存储器简单，只需要接受一个 32 位地址输入（从 PC）和一条 32 位的指令输出。电路模型如图 5 所示。

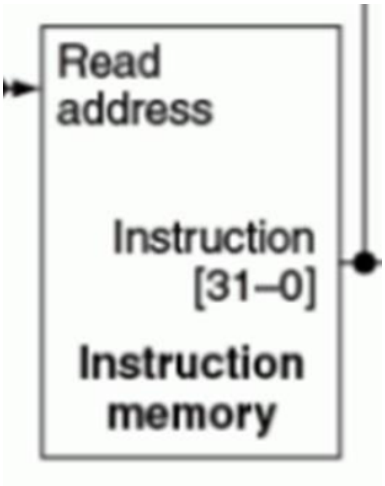

图 5: 指令存储器 (Instruction Memory) 的主要接口

2.9 PC 寄存器原理分析

PC 寄存器模块用于管理 PC 地址。输入 pcIn。在时钟上升沿保存。输出 pcOut，与 PC 寄存器内容即时同步。并用 reset 信号控制实现重置为 0 的操作。

2.10 顶层模块 (TOP) 的原理分析

顶层模块就是将上述所有模块按照设计电路连接起来的，可以完成单周期 CPU 功能的一个处理器模块。其中的连线包括数据通路和控制通路，分别用于在各个子模块之间传输数据或控制信号。部分电路如图 6 所示。

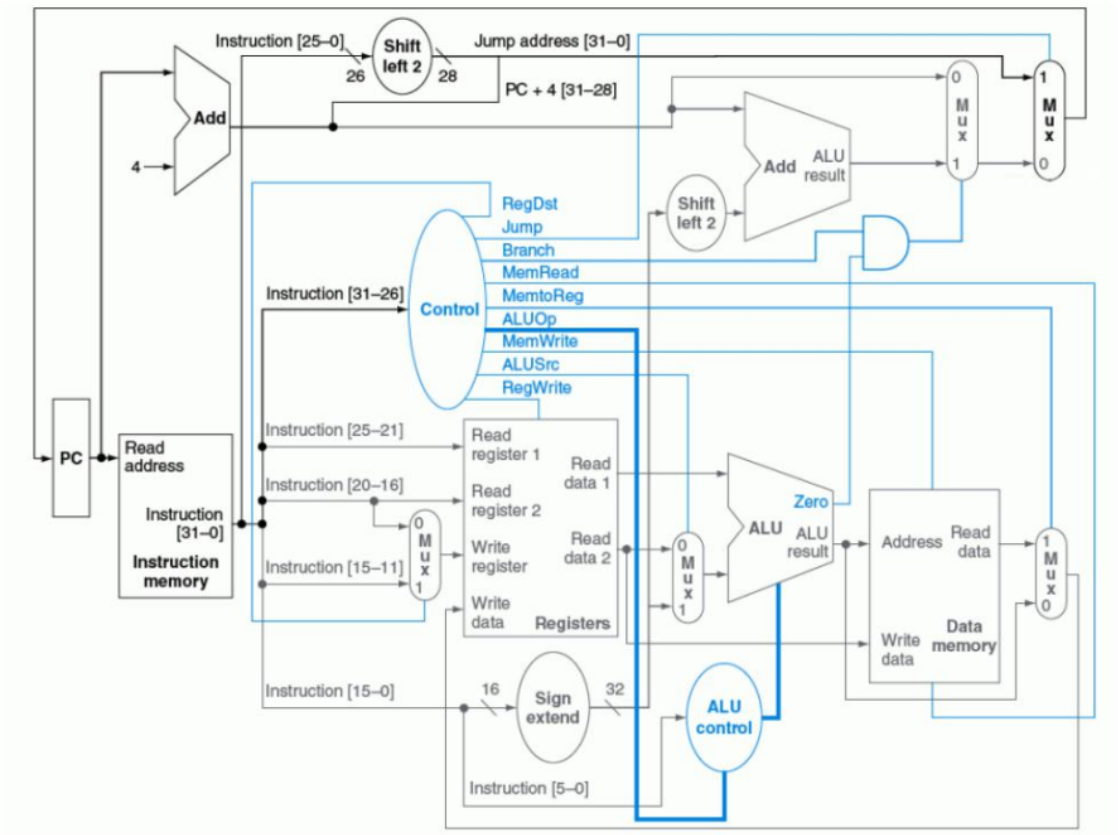

图 6: 顶层模块 (TOP) 原理图

需要注意的是，图 6 显示电路仅仅只支持 9 条 MIPS 指令运行的处理器示意图，为支持 16 条指令，在基础通路外，还需添加部分元件。具体改动如下。

- 添加一个 RegMux 和一个 Mux 用于选择写入寄存器编号和寄存器写

入的数据以此完成 jal 指令的实现。处理 jal 指令时，jalSign 信号与 regWrite 信号处于高电平，31 号寄存器被选中，同时寄存器写数据端口输入 PC + 4；

- 在 branch 和 jump 选择器（即图 6 右上角两个选择器）之间插入 jr 选择器，以此实现 jr 指令的实现。处理 jr 指令时，jrSign 信号处于高电平，jr 选择器选择寄存器读取结果，同时由于 jump 信号处于低电平，寄存器读取结果最终被返回至 PC 寄存器模块；

3 功能实现

3.1 主控制器（Ctr）功能实现

根据 2.1 节中的控制信号对应表，使用 Verilog 中的 case 语句分别对各种情况进行实现，部分代码如下，完整代码详见附录 A.1。

```
1  always @(opCode)
2      begin
3          case(opCode)
4          6'b000000://R type
5  ␣␣␣␣␣␣␣␣begin
6  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣RegDst = 1;
7  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣ALUSrc = 0;
8  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣MemToReg = 0;
9  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣RegWrite = 1;
10 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣MemRead = 0;
11 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣MemWrite = 0;
12 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣Branch = 0;
13 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣ExtSign = 0;
14 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣JalSign = 0;
```

```
15 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣ALUOp␣=␣3'b101;
16                  Jump = 0;
17              end
18              6'b100011:␣//lw
19 ␣␣␣␣␣␣␣␣begin
20 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣RegDst␣=␣0;
21 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣ALUSrc␣=␣1;
22 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣MemToReg␣=␣1;
23 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣RegWrite␣=␣1;
24 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣MemRead␣=␣1;
25 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣MemWrite␣=␣0;
26 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣Branch␣=␣0;
27 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣......
28 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣MemWrite␣=␣0;
29 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣Branch␣=␣0;
30 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣ALUOp␣=␣2'b000;
31                  Jump = 0;
32              end
33              endcase
34      end
```

3.2 运算单元控制器（ALUCtr）功能实现

根据 2.2 节中的控制信号对应表，使用 Verilog 中的 casex 语句分别对各种情况进行实现，部分代码如下，完整代码详见附录 A.2。

```
1 always @(aluOp or funct)
2     begin
3         ShamtSign = 0;
```

```
4           JrSign = 0;
5           casex({aluOp,funct})
6               9'b000xxxxxx:  //lw,sw,add,addiu
7   ALUCtrOut = 4'b0010;      //add
8               9'b001xxxxxx:  //beq
9   ALUCtrOut = 4'b0110;      //sub
10              9'b010xxxxxx:  //stli
11  ALUCtrOut = 4'b0111;
12              9'b110xxxxxx:  //stliu
13  ALUCtrOut = 4'b1000;
14              ......
15                  ALUCtrOut = 4'b1011;
16  9'b101100111:   // nor
17                  ALUCtrOut = 4'b1100;
18  9'b101101010:   // slt
19                  ALUCtrOut = 4'b0111;
20  9'b101101011:   // sltu
21                  ALUCtrOut = 4'b1000;
22  endcase
23  end
```

3.3 算术逻辑运算单元（ALU）功能实现

根据 2.3 节中的控制信号对应表，使用 Verilog 中的 case 语句分别对各种情况进行实现，并用 if 语句对 zero 信号单独设置。部分代码如下，完整代码详见附录 A.3。

```
1 always @(input1 or input2 or aluCtr)
2 begin
```

```
3         case(aluCtr)
4         4'b0000://AND
5     ALURes=input1 & input2;
6     4'b0001://OR
7         ALURes=input1 | input2;
8         4'b0010://ADD
9     ALURes=input1 + input2;
10    4'b0011://Left-shift
11        ALURes=input2 << input1;
12        4'b0100://Right-shift
13    ALURes=input2 >> input1;
14    4'b0101://xor
15        ALURes=input1;
16        4'b0110://SUB
17    ......
18    endcase
19    if(ALURes==0)
20        Zero=1;
21    else
22        Zero=0;
23    end
```

3.4 寄存器（Register）功能实现

寄存器(Register)可以一直进行读取操作,但是需要用 RegWrite 信号控制其是否允许写入。同原理 2.4 节中所言,由于不确定 WriteReg,WriteData,RegWrite 信号的先后次序，我们采用时钟（样例里命名为 Clk）的下降沿作为写操作的同步信号，防止发生错误。此外，在模块中还需对 32 个寄存器进行初始化置零处理以及接受 reset 指令后进行清零。部分实现代码如下，详见附录

A.4。

```
initial begin
        for(i=0;i<32;i=i+1)
                RegFile[i] = 0;
    end

    assign readData1 = RegFile[readReg1];
    assign readData2 = RegFile[readReg2];

    always @(negedge clk or reset)
    begin
        if(reset)
        begin
            for(i=0;i<32;i=i+1)
                RegFile[i] = 0;
        end
        else begin
            if(regWrite)
                RegFile[writeReg] = writeData;
        end
    end
```

3.5 存储器（Data Memory）功能实现

存储器（Data Memory）由 MemRead 和 MemWrite 分别用来控制读取和写入操作。同原理 2.5 节中所言，我们采用时钟（样例里命名为 Clk）的下降沿作为写操作的同步信号，防止发生错误。同样，对模块中的 64 个存储进行初始化置零操作，并且对地址范围加以检查限制，防止访问错误。

部分实现代码如下，详见附录 A.5。

```
initial begin
      for(i=0;i<1024;i=i+1)
            memFile[i]=0;
    end
    always @(memRead or address or memWrite)
    begin
        if(memRead)
        begin
            if(address<1023)
                ReadData = memFile[address];
            else
                ReadData = 0;
        end
    end

    always @(negedge clk)
    begin
        if(memWrite)
            if(address<1023)
                memFile[address] = writeData;
        if(memRead)
            if(address<1023)
                ReadData = writeData;
    end
```

3.6 带符号扩展单元（Sign Extension）功能实现

有符号扩展单元（Sign Extension）将 16 位有符号立即数扩展位 32 位有符号立即数。如原理 2.6 节中扩展规则所述，我们利用 Verilog 提供的拼接操作完成扩展。部分代码如下，详见附录 A.6。

```
assign data = signExt?{{16{inst[15]}},
      inst[15:0]}:{{16{0}},inst[15:0]};
```

3.7 数据选择器 (Mux/RegMux) 功能实现

使用 Verilog 自带的三目运算符即可实现数据选择器的功能。核心代码如下，详见附录 A.7。

```
assign out = select?in1:in0;
```

3.8 指令存储器（Instruction Memory）功能实现

指令存储器只需要根据输入的 PC 地址输出对应命令即可。需注意，要对模块中的寄存器变量初始化置零。部分代码如下，详见附录 A.8。

```
initial begin
      for(i=0;i<1024;i=i+1)
            instFile[i]=0;
    end
    assign inst = instFile[address/4];
```

3.9 PC 寄存器模块功能实现

如原理 2.9 中所述，PC 寄存器在时钟上升沿将 pcIn 保存，同时保证了 pcOut 输出与 PC 寄存器内容一致。部分代码如下，详见附录 A.9。

```
initial p = 0;
    always @ (posedge clk or reset)
    begin
        if(reset)
            p = 0;
        else
            p = pcIn;
    end
    assign pcOut = p;
```

3.10 顶层模块（Top）功能实现

在顶层模块中，需声明需要的所有连接线路，将其对应连接到相应的模块上。声明代码如下。

```
wire REG_DST;
wire REG_WRITE;
wire EXT_OP;
wire ALU_SRC;
wire[2:0] ALU_OP;
wire[3:0] ALU_CTR;
wire BRANCH;
wire JUMP;
wire JAL_SIGN;
wire MEM_WRITE;
wire MEM_READ;
wire MEM_TO_REG;
wire ALU_ZERO;
wire SHAMT_SIGN;
```

```
15  wire JR_SIGN;
16  wire[4:0] WRITE_REG_ID;
17  wire[4:0] WRITE_REG_ID_AFTER_JAL_MUX;
18
19  wire[31:0] INST;
20  wire[31:0] REG_WRITE_DATA_AFTER_JAL_MUX;
21  wire[31:0] REG_WRITE_DATA;
22  wire[31:0] REG_READ_DATA1;
23  wire[31:0] REG_READ_DATA2;
24  wire[31:0] EXT_IMM;
25  wire[31:0] ALU_INPUT1;
26  wire[31:0] ALU_INPUT2;
27  wire[31:0] ALU_OUTPUT;
28  wire[31:0] MEM_OUTPUT_DATA;
29  wire[31:0] MEM_INPUT_DATA;
30  wire[31:0] PC_IN;
31  wire[31:0] PC_OUT;
32  wire[31:0] PC_AFTER_BRANCH_MUX;
33  wire[31:0] PC_AFTER_JR_MUX;
34  wire[31:0] JUMP_ADDR;
35  wire[31:0] JUMP_ADDR_debug;
```

以寄存器（Registers）模块的连接代码为例，如下。

```
1  Registers registers(
2          .readReg1(INST[25:21]),
3          .readReg2(INST[20:16]),
4          .writeReg(WRITE_REG_ID_AFTER_JAL_MUX),
5          .writeData(REG_WRITE_DATA_AFTER_JAL_MUX),
6          .regWrite(REG_WRITE & !JR_SIGN),
```

```
7               .clk(clk),
8               .reset(reset),
9               .readData1(REG_READ_DATA1),
10              .readData2(REG_READ_DATA2)
11          );
```

值得注意的是，在 Top 模块设计的时候，使用到了不止一个 Mux 模块，即数据选择器模块将被复用。在连接时，为不同的 Mux 取不同的对应名称，方便区分其功能与连线。代码详见附录 A.10。

4 结果验证

在激励文件 test.v 中，我们使用

```
1 $readmemh("mem_data.dat",top.data_mem.memFile);
2 $readmemb("mem_inst.dat",top.inst_mem.instFile);
```

两条命令分别采用相对路径的形式,从 mem_data.dat 文件和 mem_inst.dat 文件中直接读取数据和指令存入对应的数据存储器（Data Memory）和指令存储器（Instruction Memory）中。这两个文件应当位于.../lab05.sim/sim_1/behav/xsim 文件夹中。

指令代码根据 MIPS 指令对应的汇编代码编写，具体对应关系详见 https://blog.csdn.net/goodlinux/article/details/6731484

需要注意的是，本实验中的类 MIPS 单周期处理器仅支持 16 条 MIPS 指令的处理，包括 R 型指令：add、sub、or、and、slt、sll、srl、jr；I 型指令：lw、sw、addi、ori、beq；J 型指令：j、jal。在 mem_inst.dat 文件中随即顺序写入这些指令，进行仿真模拟。

将寄存器（Registers）、数据存储器（Data Memory）、指令存储器（Instruction Memory）以及控制器的各个控制信号加入到仿真波形图的监视中，实验结果如图 7、图 8 所示。 可以看出我们设计的类 MIPS 单周期

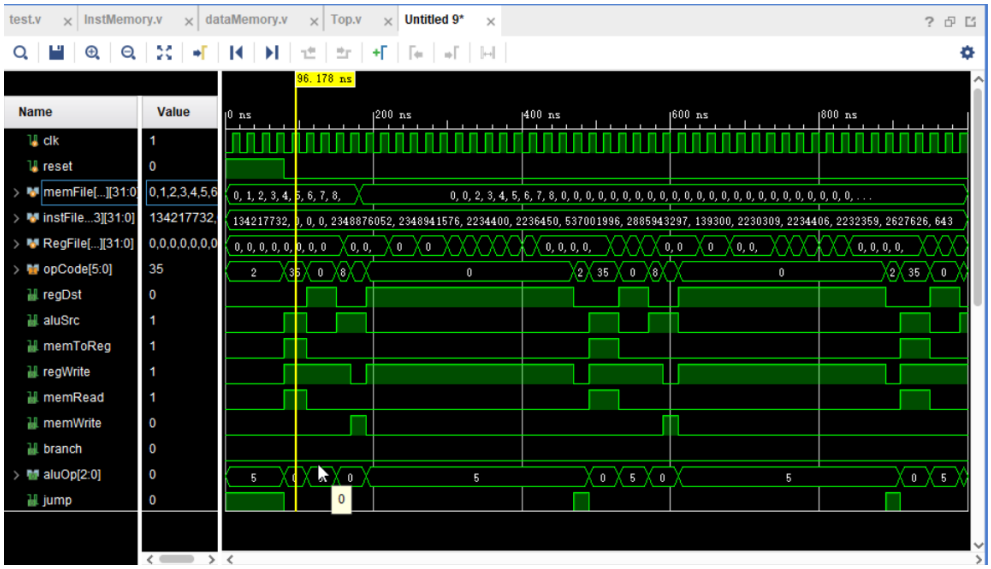

图 7: 处理器仿真波形图 1

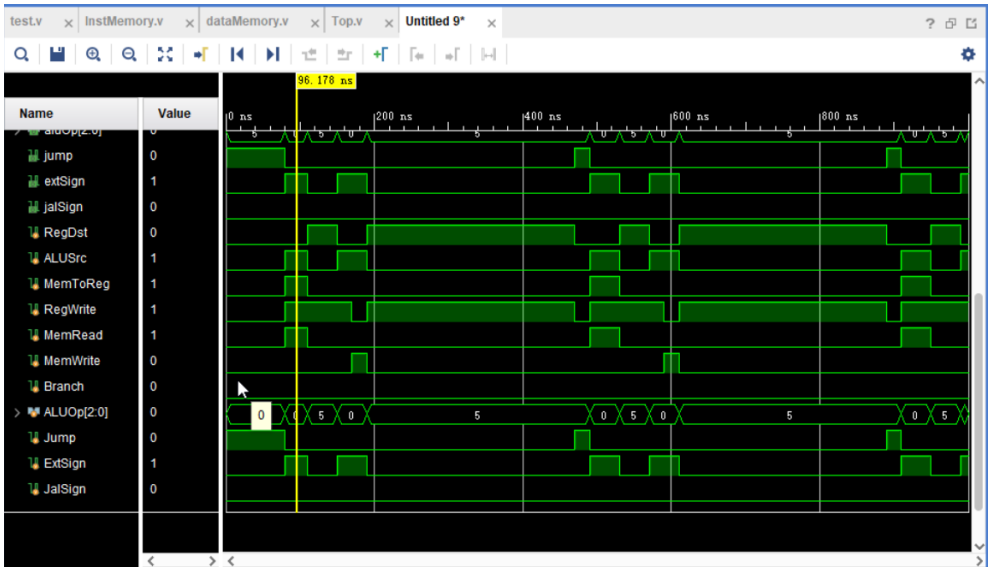

图 8: 处理器仿真波形图 2

处理器完成了所有功能。激励文件代码详见附录 B。

5 总结反思

本实验结合了实验 3、4 所设计的功能模块，并且新设计了数据选择器等更多的模块，将其按照正确的方式连接后，我们就得到了一个支持 16 条 MIPS 指令执行的类 MIPS 单周期处理器。

在计算机系统结构的课程当中，我们已经从原理上学习了 MIPS 单周期处理器的运行逻辑方式，在这次实验中，我们更是从实践的角度，更加深入地理解了处理器的运行机制和各个模块的连接方式和以及为什么要这么连接的原因。

本次实验让我们第一次，由简至繁地设计一个我们生活中常见的，可以与计算机真正联系在一起的部件模块，让我更加理解了计算机的发展规律。

此外，本次实验的难点在于正确区分各个连线与模块的连接（毕竟子功能模块是已经设计好了的），这需要正确的电路原理图的指导。只有在明确各个连线的功能与传输内容后，才能更快更准确地进行连接。

总之，在此次实验中，我收益颇丰。

6 致谢

感谢本次实验中指导老师在课程微信群里为同学们答疑解惑；

感谢上海交通大学网络信息中心提供的远程桌面资源；

感谢计算机科学与工程系相关老师对于课程指导书的编写以及对于课程的设计，让我们可以更快更好地学习相关知识，掌握相关技能；

感谢电子信息与电气工程学院提供的优秀的课程资源。

A 设计文件完整代码实现

A.1 主控制器（Ctr）的代码实现

参见代码文件 Ctr.v

A.2 运算单元控制器（ALUCtr）的代码实现

参见代码文件 ALUCtr.v

A.3 算术逻辑运算单元（ALU）的代码实现

参见代码文件 ALU.v

A.4 寄存器（Register）的代码实现

参见代码文件 Register.v

A.5 数据存储器（Data Memory）的代码实现

参见代码文件 dataMemory.v

A.6 有符号扩展单元（Sign Extension）的代码实现

参见代码文件 signext.v

A.7 数据选择器（Mux/RegMux）的代码实现

参见代码文件 Mux.v 和 RegMux.v

A.8 指令存储器（Instruction Memory）的代码实现

参见代码文件 InstMemory.v

A.9 PC 寄存器模块的代码实现

参见代码文件 PC.v

A.10 顶层模块（Top）的代码实现

参见代码文件 Top.v

B 激励文件完整代码实现

参见代码文件 test.v。