计算机结构实验 lab05

张露伊

2022 年 3 月 29 日

摘要

本实验是在实验 3、实验 4 的基础上对部分已有模块进行修改，并且新实现了指令内存模块、数据选择器模块、PC 寄存器模块。然后通过 top 模块将各个模块连接在一起，实现类 MIPS 单周期处理器。该类 MIPS 单周期处理器支持 16 条 MIPS 指令。本实验将通过软件仿真的形式让处理器运行指令，以此进行实验结果的验证。

目录

1 实验介绍

1.1 实验名称

简单的类 MIPS 单周期处理器的实现——整体调试

1.2 实验目的

1. 理解简单的类 MIPS 单周期处理器的工作原理（即几类基本指令执行时所需的数据通路和与之对应的控制线路及其各功能部件间的互联定义、逻辑选择关系）

2. 完成简单的类 MIPS 单周期处理器
   a) 9 条 MIPS 指令 CPU 的实现与调试（lw,sw,beq,add,sub,and,or,slt,j）
   b) 拓展至 16 条指令 CPU 的设计与实现（增加 addi,andi,ori,sll ,srl,jal,jr）

2 原理分析

2.1 主控制器模块（Ctr）的分析

主控制器 (Ctr) 的输入为指令的操作码 (opCode) 字段，主控制器模块对操作码进行译码，向 ALUCtr、Data Memory、Registers、Mux 等部件输出正确的控制信号。

Ctr 的设计在之前实验三就已经实现了部分指令，要实现 16 条指令，需要再增加识别 jal、jr、srl 等指令并输出对应的信号。

主控制器模块产生的控制信号及说明如下表1所示：

表 1: Ctr 输出信号及含义

信号	内部寄存器	具体说明
regDst	RegDst	目标寄存器的选择信号；低电平：rt 寄存器；高电平：rd 寄存器
aluSrc	ALUSrc	ALU 第二个操作数来源选择信号；低电平：rt 寄存器值，高电平：立即数拓展结果
memToReg	MemToReg	写寄存器的数据来源选择信号；低电平：ALU 运算结果，高电平：内存读取结果
regWrite	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
memRead	MemRead	内存读使能信号，高电平有效
memWrite	MemWrite	内存写使能信号，高电平有效
aluOp	ALUOp	传递给 ALUCtr 来进一步进行运算操作
branch	Branch	条件跳转信号，高电平有效
jump	Jump	无条件跳转信号，高电平有效令
jal	Jal	跳转并链接指令 (JAL) 信号，高电平有效
extOp	ExtOp	带符号扩展信号，高电平有效

其中 aluOp 信号控制 ALU 进行各种运算含义如下表所示：

表 2: aluOp 信号含义

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	加法
001	beq	减法
011	andi	逻辑与
100	ori	逻辑或
111	xori	逻辑异或
010	slti	带符号数大小比较
110	sltiu	无符号数大小比较

主控制器 (Ctr) 解析 OpCode 对应各种控制信号如下表 3 所示:

表 3: Ctr 真值表

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 ALU 控制器模块（ALUCtr）的分析

ALUCtr 是根据主控制器的 ALUOp 控制信号来判断指令类型，并根据指令的后 6 位区分 R 型指令。综合这两种输入，以控制 ALU 做正确操作。

ALUCtr 的设计之前在实验三已经实现，只需要增加几条命令对应的信号即可。

ALUCtr 信号输出与 ALUOp 及 Funct 的对应关系如下表所示:

指令	ALUOp	Funct	ALUCtrOut	具体说明
lw	000	xxxxxx	0010	ALU 执行加法运算
sw	000	xxxxxx	0010	ALU 执行加法运算
beq	001	xxxxxx	0110	ALU 执行减法运算
addi	010	xxxxxx	0010	ALU 执行加法运算
andi	011	xxxxxx	0000	ALU 执行逻辑与运算
ori	100	xxxxxx	0001	ALU 执行逻辑或运算
sll	101	000000	0011	ALU 执行逻辑左移运算
srl	101	000010	0100	ALU 执行逻辑右移运算
jr	101	001000	0101	ALU 不用进行运算
add	101	100000	0010	ALU 执行加法运算
sub	101	100010	0110	ALU 执行减法运算
and	101	100100	0000	ALU 执行逻辑与运算
or	101	100101	0001	ALU 执行逻辑或运算
slt	101	101010	0111	ALU 执行小于时置位运算
j	110	xxxxxx	0101	ALU 不用进行运算
jal	110	xxxxxx	0101	ALU 不用进行运算

表 4: 运算单元控制器 (ALUCtr) 的解析方式

除上表所示之外，本实验中 ALUCtr 还负责产生 shamtSign 信号与 jrSign 信号。关于这两个信号的说明如下：

- shamtSign 信号：当指令为 sll、srl、sra 时需要指令的 shamt 段作为输入，此时该信号为高电平。其余时候为低电平。
- jrSign 信号：当指令为 jr 时为高电平，其余情况下为低电平。

2.3 ALU 模块的分析

ALU 模块接受 ALUCtr 信号，并根据此信号选择执行对于的 ALU 计算功能。ALU 功能与 ALUCtr 信号的对应关系如下表所示：

ALUCtrOut	ALU 执行算术逻辑运算类型
0000	逻辑与 (and)
0001	逻辑或 (or)
0010	加法 (add)
0011	左移 (left-shift)
0100	右移 (right-shift)
0101	无运算 (nop)
0110	减法 (sub)
0111	小于时置位 (slt)

表 5: ALU 执行的算术逻辑运算类型与 ALUCtrOut 的对应方式

ALU 模块产生的输出包括 32 位的运算结果以及 1 位 zero 信号；当运算结果为 0 时，zero 处于高电平，其余时候 zero 处于低电平。

2.4 寄存器 (Register) 模块的原理分析

本部分和实验四的寄存器的原理分析完全相同，不再赘述。

2.5 数据存储器 (Data Memory) 模块的原理分析

本部分和实验四的存储器的原理分析完全相同，不再赘述。

2.6 有符号扩展单元 (Sign Extension) 模块原理分析

本部分和实验四的有符号扩展单元的原理分析完全相同，不再赘述。

2.7 多路选择器模块 (Mux/RegMux) 的分析

多路选择器模块接受两个输入信号和一个选择信号，产生一个输出信号。本实验中，我们使用了两种数据选择器，包括 Mux 和 RegMux。Mux 的输入与输出信号均为 32 位，用于对进行数据进行选择。RegMux 的输出和输出信号为 5 位，用于寄存器选取信号的选择。

2.8 指令内存模块 (InstMem) 的分析

本实验中处理器采用哈佛架构，指令内存与数据内存分离。指令内存模块 (InstMem) 接受一个 32 位地址输入，输出一条 32 位指令. 它的设计和存储器几乎相同，而且不需要支持修改操作。因此在读入初始数据后，后续只要直接输出给定的输入 PC 值所指向的指令即可。

2.9 PC 寄存器模块的分析

PC 寄存器模块用于管理 PC 地址。接受输入 pcIn，在时钟上升沿将 pcIn 保存进 PC 寄存器。输出 pcOut 为当前 PC 地址，pcOut 与 PC 寄存器内容即时同步。当 reset 信号处于高电平时，将 PC 值重置为 0。

2.10 顶层模块 (top) 的分析

顶层模块将以上所有模块连接在一起，完成单周期 CPU 的功能。顶层模块主要的数据通路与控制通路如图1所示。

3 功能实现

3.1 主控制器模块（Ctr）的实现

该模块的实现相比实验三多出几条添加的指令，opCode 从 2 位变为 3 位，其他部分和实验三的实现相同。

部分核心代码如下：

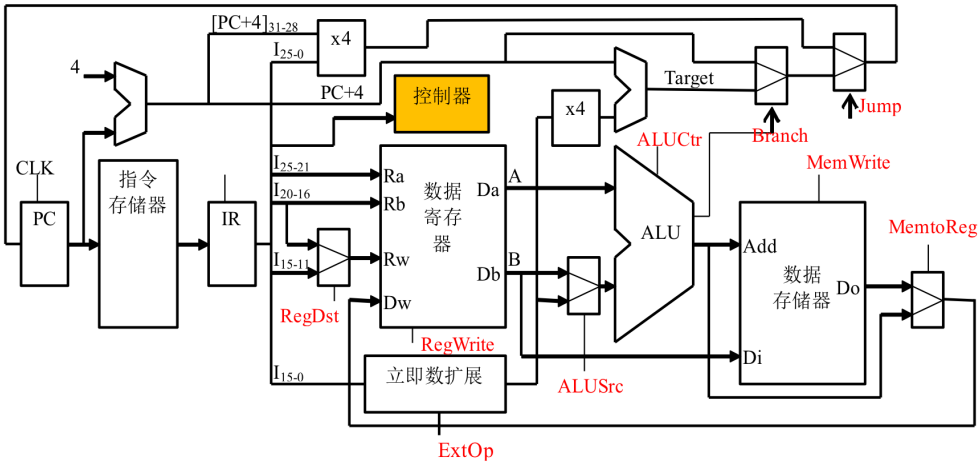

图 1: 顶层模块基础数据通路与控制通路

```
always @(opCode)
    begin
        case(opCode)
        6'b000000: //R type
        begin
            RegDst = 1;
            ALUSrc = 0;
            MemToReg = 0;
            RegWrite = 1;
            MemRead = 0;
            MemWrite = 0;
            Branch = 0;
            ExtSign = 0;
            JalSign = 0;
            ALUOp = 3'b101;
            Jump = 0;
        end
        \\其余类似
        endcase
   end
```

3.2 ALU 控制器模块（ALUCtr）的实现

改模块的实现与实验三种 ALUCtr 的实现类似，额外添加了几条指令对应的输出。当指令为 sll、srl、sra 时需要指令的 shamt 段作为输入，shamtSign 置为 1，其他情况为 0. 当指令为 jr 时，jrSign 置为 1，其他情况为 0。部分核心代码如下：

```
always @ (aluOp or funct)
    begin
        casex({aluOp,funct})
            9'b000xxxxxx:  // lw,sw,add,addiu
                ALUCtrOut = 4'b0010;     //add
            9'b001xxxxxx:  // beq
                ALUCtrOut = 4'b0110;     //sub
            //其余类似
        endcase
        if({aluOp,funct}==9'b101000000 || {aluOp,funct}==9'b101000010
        || {aluOp,funct}== 9'b101000011)
             ShamtSign = 1;
        else  ShamtSign = 0;

        if({aluOp,funct} == 9'b101001000)
            JrSign = 1;
        else JrSign = 0;

    end
```

3.3 ALU 模块的实现

3.4 寄存器 (Register) 模块的原理实现

本部分和实验四的寄存器的原理实现相似，本实验中寄存器模块可以响应 reset 信号，当 reset 为高电平时，所有寄存器清零。核心代码如下：

```
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 数据存储器 (Data Memory) 模块的原理实现

本部分和实验四的存储器的原理实现完全相同，不再赘述。

3.6 有符号扩展单元 (Sign Extension) 模块原理实现

本部分和实验四的有符号扩展单元的原理实现相似，为了区分有符号数扩展和无符号数扩展，可以通过一个三目运算符根据 signExt 信号在两种扩展结果中进行选择。其余部分和实验四相同。核心代码如下：

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

3.7 多路选择器模块 (Mux/RegMux) 的实现

核心代码如下：

```
module Mux(
    input select,
    input [31:0] input0,
    input [31:0] input1,
    output [31:0] out
    );
    assign out = select ? input1:input0;
endmodule
```

```
module Mux(
    input select,
    input [5:0] input0,
    input [5:0] input1,
    output [5:0] out
    );
    assign out = select ? input1:input0;
endmodule
```

3.8 指令内存模块（InstMem）的实现

该模块根据输入的 PC 地址输出相应的指令。

核心代码实现如下：

```
assign inst = instFile[address>>2];
```

3.9 PC 寄存器模块的实现

核心代码如下：

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

3.10 顶层模块（Top）的实现

3.10.1 连接线路声明

```
//控制信号连线
wire RegDst;
wire RegWrite;
wire ExtOp;
wire ALUSrc;
wire[2:0] ALUOp;
wire[3:0] ALUCtr;
wire Branch;
wire Jump;
wire JalSign;
wire MemWrite;
wire MemRead;
wire MemToReg;
wire ALUZero;
wire ShamtSign;
wire JrSign;
wire[4:0] WriteRegID;
wire[4:0] WRITE_REG_ID_AFTER_JAL_MUX;
//数据通路连线
wire[31:0] INST;
wire[31:0] REG_WRITE_DATA_AFTER_JAL_MUX;
wire[31:0] REG_WRITE_DATA;
wire[31:0] REG_READ_DATA1;
```

```
wire[31:0] REG_READ_DATA2;
wire[31:0] EXT_IMM;
wire[31:0] ALU_INPUT1;
wire[31:0] ALU_INPUT2;
wire[31:0] ALU_OUTPUT;
wire[31:0] MEM_OUTPUT_DATA;
wire[31:0] MEM_INPUT_DATA;
wire[31:0] PC_IN;
wire[31:0] PC_OUT;
wire[31:0] PC_AFTER_BRANCH_MUX;
wire[31:0] PC_AFTER_JR_MUX;
wire[31:0] JUMP_ADDR;
```

3.10.2 Ctr 端口连接

```
Ctr main_ctr(
        .opCode(INST[31:26]),
        .regDst(RegDst),
        .aluSrc(ALUSrc),
        .memToReg(MemToReg),
        .regWrite(RegWrite),
        .memRead(MemRead),
        .memWrite(MemWrite),
        .branch(Branch),
        .aluOp(ALUOp),
        .jump(Jump),
        .extSign(ExtOp),
        .jalSign(JalSign)
    );
```

其余模块的端口连接与 Ctr 类似。值得注意的是，reset 需要与 PC 寄存器模块、寄存器模块相连，用于完成处理器重置工作；clk 需要与 PC 寄存器模块、寄存器模块、数据内存模块相连，用于同步数据写入的时间。

顶层模块的完整实现可以参照 top.v 文件。

4 结果验证

仿真数据可以参见 mem_data, 详细指令数据可以参见 mem_inst。由于指令数目较多故不再赘述，下面仅给出汇编代码。

测试指令如下：

```
lw $1,0($0)
lw $2,1($0)
lw $3,7($0)
lw $4,11($3)
add $5,$1,$2
sub $7,$4,$1
lw $16,5($7)
and $5,$1,$4
or $8,$16,$2
sw $8,7($5)
addi $10,$1,256
ori $12,$16,256
lw $9,12($0)
beq $9,$8,1
sll $13,$11,4
sll $14,$11,4
srl $15,$10,4
slt $17,$15,$14
slt $18,$15,$16
j 22
slt $18,$15,$14
jal 24
beq $16,$11,3
addi $11,$11,2
jr $31
addi $11,$11,2
addi $11,$11,2
```

仿真结果截图如下：

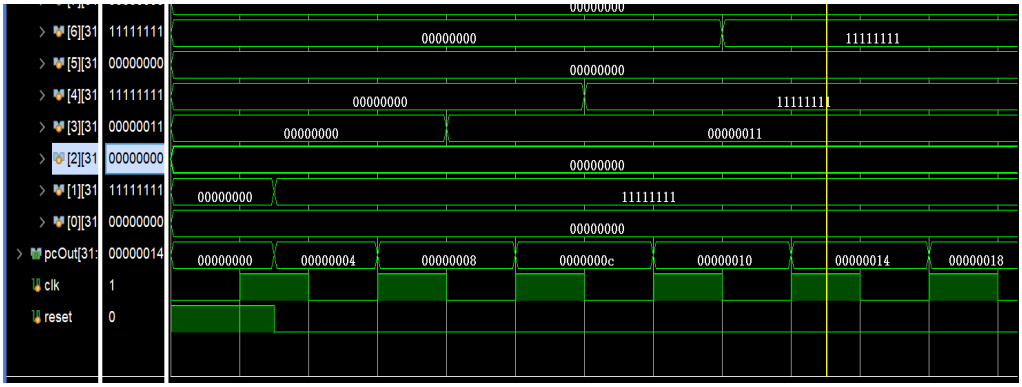

图 2: 单周期处理器仿真截图 1

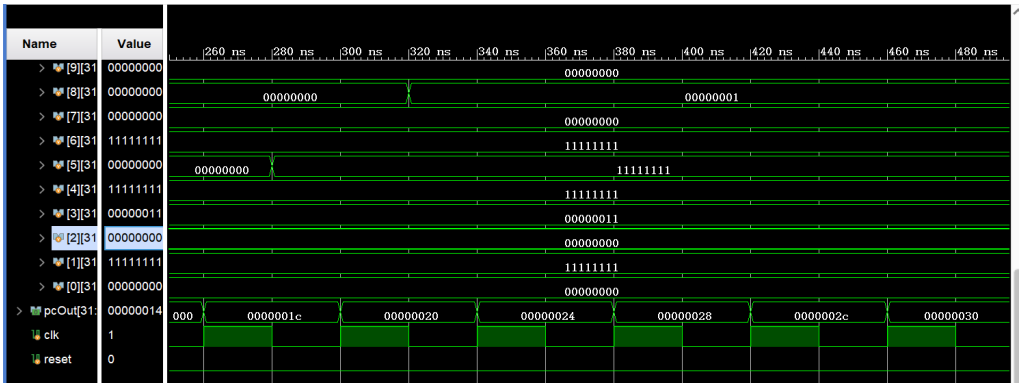

图 3: 单周期处理器仿真截图 2

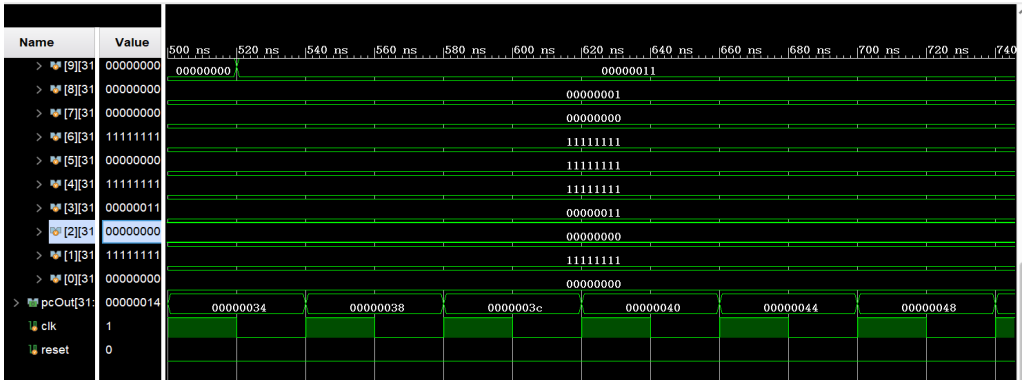

图 4: 单周期处理器仿真截图 3

由于数据过长，则只给出部分仿真结果截图，详细仿真结果可以参考提交的文件。通过仿真结果可以知道单周期处理器的设计合理，仿真结果符合预期，设计正确。

5 心得体会

实验 5 在实验 3 和实验 4 的基础上进行实验，添加了一些必要的模块使得整个 MIPS 单周期处理器的功能得以实现。而我之前的疑问——模块之间的整合也得到了解决。top 模块可以将分散的模块进行整合连接，使得一个模块的输出可以作为另一个模块的输入。

本实验在模块设计方面其实比较简单，只要弄明白每个单独组件的输入输出即可，这部分知识可以参考计算机结构课程上所讲的 MIPS 单周期处理器的设计。真正的难点在于各部分组件之间的整合与连线。一旦命名出现错误就可能导致仿真的错误。而如果连线和输入错误则会导致仿真结果根本没有输出。而自己本身对于 vivado 的调试较为陌生，因此在调试方面花费了很大的功夫。

本次实验让我对 MIPS 单周期处理器的组成设计更加熟悉，使我受益匪浅。