计算机系统结构实验报告　实验 5

类 MIPS 单周期处理器的设计与实现

黄骁 520021910388

2022 年 4 月 12 日

摘要

本实验实现了类 MIPS 单周期处理器的几个重要部件，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）。本实验以实验三、四实现的模块为基础，对部分功能进行了修改，同时添加了部分控制信号。本实验通过软件仿真的形式进行实验结果的验证。

目录

1 实验目的

本次实验有如下两个实验目的:

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 (lw, sw, beq, add, sub, and, or, slt, j) CPU 的实现与调试;

   (b) 拓展至 16 条指令 (增加 addi, andi, ori, sll, srl, jal, jr) CPU 的设计与实现;

2 原理分析

2.1 主控制单元 (Ctr) 原理分析

主控制单元 (Ctr) 的输入为指令的操作码 (opCode) 字段，操作码经过 Ctr 的译码，给 ALUCtr，DataMemory，Registers，Muxs 等功能单元输出正确的控制信号。

本实验中，主控制单元 (Ctr) 可以识别 R 型指令 (add, sub, and, or, slt, sll, srl, jr)、I 型指令 (addi, andi, ori, lw, sw, beq)、J 型指令 (j, jar) 并输出对应的控制信号。

本次实验用到的控制信号与实验三类似，其中标注 (*) 为与实验三不同之处，比如增添了 ExtSign 和 JalSign 信号，ALUOp 信号的位数也调整为三位，具体说明如表1所示。

信号	具体说明
*ALUOp	3 位信号，发送给 ALU 控制单元 (ALUCtr) 用来进一步解析运算类型的控制信号
ALUSrc	ALU 第二个操作数来源选择信号；低电平：选择 rt 寄存器值，高电平：选择立即数
Branch	条件跳转信号，高电平说明当前指令是条件跳转指令
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
MemToReg	写寄存器的数据来源选择信号；低电平：选择 ALU 运算结果，高电平：选择内存读取数据
MemRead	内存读使能信号，高电平有效
MemWrite	内存写使能信号，高电平有效
RegDst	目标寄存器的选择信号；低电平：写入 rt 寄存器；高电平：写入 rd 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
*ExtSign	带符号扩展信号，高电平对立即数进行带符号扩展
*JalSign	跳转并链接指令信号，高电平说明当前指令是 jal 指令

表 1: 主控制单元 (Ctr) 产生的控制信号

由于 ALUOp 信号包含三个二进制位以支持 16 条指令，与实验三有所不同，调整后如表 2 所示。

ALUOp 的信号内容	指令	具体说明
000	lw, sw, addi	ALU 执行加法运算
001	beq	ALU 执行减法运算
010	slti	ALU 执行带符号大小比较
011	andi	ALU 执行逻辑与运算
100	ori	ALU 执行逻辑或运算
101	R-format	ALUCtr 结合指令 Funct 段决定最终操作
110	j, jal	ALU 不需要进行操作

表 2: ALUOp 信号的具体含义以及解析方式

因篇幅原因，这里不再列出主控制单元 (Ctr) 产生的各种控制信号与指令 OpCode 段的对应方式。

2.2 ALU 控制单元 (ALUCtr) 原理分析

算数逻辑单元 ALU 的控制单元（ALUCtr）是根据主控制器的 ALUOp 控制信号来判断指令类型，并依据指令的后 6 位区分 R 型指令。综合这两种输入，以控制 ALU 做正确操作。

由于 ALUOp 信号的改动，ALU 控制单元 (ALUCtr) 也需要做出相应的更改，其解析方式如表 3 所示。

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

表 3: ALU 控制单元 (ALUCtr) 的解析方式

为了支持 jr 指令以及带有 shamt 操作数的 sll 与 srl 指令，我们新增了两个信号 jrSign 与 shamt-Sign，分别用来表示该指令是否为 jr、该指令操作数是否在 shamt 中。

2.3 算术逻辑单元 (ALU) 原理分析

算术逻辑单元 ALU 根据 ALUCtr 的控制信号将两个输入执行与之对应的操作。ALURes 为输出结果。若减法操作 ALURes 的结果为 0 时，则 Zero 输出置为 1。该信号用于与 branch 指令结合判断是否满足转移条件。

(*) 为与实验三不一样的控制信号，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 4 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
1100	逻辑或非 nor

表 4: ALUCtrOut 与 ALU 操作的对应关系

2.4 寄存器 (Register)、存储器 (Data Memory) 及有符号扩展单元 (Sign Extension) 原理分析

本部分和实验四的原理类似，但需要注意的是模块一些部分的修改。

在寄存器 (Register) 中可以响应 reset 信号，当 reset 为高电平时，所有寄存器清零。

在有符号扩展单元 (Sign Extension) 中增添了带符号扩展信号 signExt，高电平代表进行带符号扩展，低电平代表无符号扩展。

2.5 指令存储器 (Instruction Memory) 原理分析

本部分和存储器 (Data Memory) 几乎相同，而且不需要支持修改操作。指令存储器接受一个 32 位地址输入, 输出一条 32 位指令。

2.6 程序计数器模块 (PC) 原理分析

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 。

2.7 数据选择器 (Mux) 原理分析

数据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 Mux5 和 Mux32。Mux32 的输入与输出信号均为 32 位, 用于对进行数据进行选择。Mux5 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 顶层模块 (Top) 原理分析

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。顶层模块中的连线包括数据通路和控制通路。数据通路用于传输中间数据, 一般为 32 位或 16 位; 控制通路用于传输控制信号。这里给出整体组装后的单周期 MIPS 处理器的电路设计图，如图 1 所示。

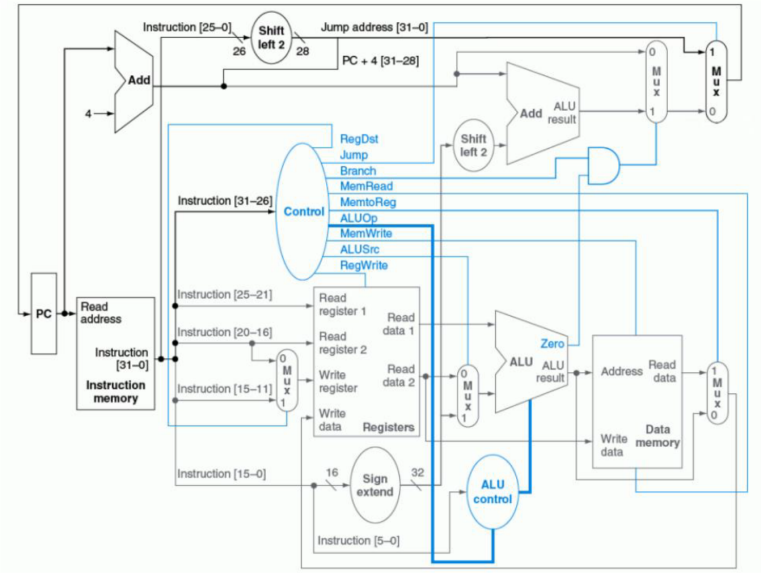

图 1: 单周期 MIPS 处理器电路设计图

3 功能实现

3.1 主控制单元 (Ctr) 功能实现

在第 2.1 节中我们详细介绍了主控制单元 (Ctr) 的设计，我们只需要按照设计的信号进行相应的实现即可。完整代码详见附录 A.1。

```
always @(opCode)
    begin
        case(opCode)
            6'b000000:    // R Type
            begin
                regDst = 1;
                aluSrc = 0;
                memToReg = 0;
                regWrite = 1;
                memRead = 0;
                memWrite = 0;
                branch = 0;
                aluOp = 3'b101;
                jump = 0;
                extSign = 0;
```

```
16              jalSign = 0;
17          end
18          6'b000010:    // jump
19          begin
20              regDst = 0;
21              aluSrc = 0;
22              memToReg = 0;
23              regWrite = 0;
24              memRead = 0;
25              memWrite = 0;
26              branch = 0;
27              aluOp = 3'b110;
28              jump = 1;
29              extSign = 0;
30              jalSign = 0;
31          end
32          6'b000011:   // jal
33          begin
34              regDst = 0;
35              aluSrc = 0;
36              memToReg = 0;
37              regWrite = 1;
38              memRead = 0;
39              memWrite = 0;
40              branch = 0;
41              aluOp = 3'b110;
42              jump = 1;
43              extSign = 0;
44              jalSign = 1;
45          end
46          6'b000100:    // beq
47          begin
48              regDst = 0;
49              aluSrc = 0;
50              memToReg = 0;
51              regWrite = 0;
52              memRead = 0;
53              memWrite = 0;
54              branch = 1;
55              aluOp = 3'b001;
56              jump = 0;
```

```
57              extSign = 1;
58              jalSign = 0;
59          end
60          6'b001000:    // addi
61          begin
62              regDst = 0;
63              aluSrc = 1;
64              memToReg = 0;
65              regWrite = 1;
66              memRead = 0;
67              memWrite = 0;
68              branch = 0;
69              aluOp = 3'b010;
70              jump = 0;
71              extSign = 1;
72              jalSign = 0;
73          end
74          6'b001100:   // andi
75          begin
76              regDst = 0;
77              aluSrc = 1;
78              memToReg = 0;
79              regWrite = 1;
80              memRead = 0;
81              memWrite = 0;
82              branch = 0;
83              aluOp = 3'b011;
84              jump = 0;
85              extSign = 0;
86              jalSign = 0;
87          end
88          6'b001101:   // ori
89          begin
90              regDst = 0;
91              aluSrc = 1;
92              memToReg = 0;
93              regWrite = 1;
94              memRead = 0;
95              memWrite = 0;
96              branch = 0;
97              aluOp = 3'b100;
```

```
98              jump = 0;
99              extSign = 0;
100             jalSign = 0;
101         end
102
103         6'b100011:    // lw
104         begin
105             regDst = 0;
106             aluSrc = 1;
107             memToReg = 1;
108             regWrite = 1;
109             memRead = 1;
110             memWrite = 0;
111             branch = 0;
112             aluOp = 3'b000;
113             jump = 0;
114             extSign = 1;
115             jalSign = 0;
116         end
117         6'b101011:    // sw
118         begin
119             regDst = 0;
120             aluSrc = 1;
121             memToReg = 0;
122             regWrite = 0;
123             memRead = 0;
124             memWrite = 1;
125             branch = 0;
126             aluOp = 3'b000;
127             jump = 0;
128             extSign = 1;
129             jalSign = 0;
130         end
131         default:      // default
132         begin
133             regDst = 0;
134             aluSrc = 0;
135             memToReg = 0;
136             regWrite = 0;
137             memRead = 0;
138             memWrite = 0;
```

```
            branch = 0;
            aluOp = 3'b111;
            jump = 0;
            extSign = 0;
            jalSign = 0;
        end
    endcase
end
```

当出现不支持的指令时，我们将 ALUOp 信号设为 111，表示未定义的操作，同时将其他的所有控制信号置零。

3.2 运算单元控制器 (ALUCtr) 功能实现

在第 2.2 节中我们详细介绍了运算单元控制器 (ALUCtr) 的设计，我们只需要按照第 2.2 节中的表 3 进行相应信号的实现即可。完整代码详见附录 A.2。

```
always @ (aluOp or funct)
    begin
        shamtSign = 0;
        jrSign = 0;
        casex ({aluOp, funct})
            9'b000xxxxxx: aluCtrOut = 4'b0010; // lw, sw
            9'b001xxxxxx: aluCtrOut = 4'b0110; // beq
            9'b010xxxxxx: aluCtrOut = 4'b0010; // addi
            9'b011xxxxxx: aluCtrOut = 4'b0000; // andi
            9'b100xxxxxx: aluCtrOut = 4'b0001; // ori
            9'b101000000: // sll
            begin
                aluCtrOut = 4'b0011;
                shamtSign = 1;
            end

            9'b101000010: // srl
            begin
                aluCtrOut = 4'b0100;
                shamtSign = 0;
            end
            9'b101001000: // jr
            begin
                aluCtrOut = 4'b0101;
                jrSign = 1;
            end
```

```
            9'b101100000: aluCtrOut = 4'b0010; // add
            9'b101100010: aluCtrOut = 4'b0110; // sub
            9'b101100100: aluCtrOut = 4'b0000; // and
            9'b101100101: aluCtrOut = 4'b0001; // or
            9'b101101010: aluCtrOut = 4'b0111; // slt
            9'b110xxxxxx: aluCtrOut = 4'b0101; // j, jal
        endcase
    end
```

注：两个控制信号，用于控制操作数是否从 shamt 中选取的 shamtSign 信号以及指令是否为 jr 的 jrSign 信号。

3.3 算术逻辑运算单元 (ALU) 功能实现

在第 2.3 节中我们详细介绍了运算单元控制器 (ALUCtr) 的设计，我们只需要按照第 2.3 节中的表 4 进行相应信号的实现即可。完整代码可参考附录 A.3。

```
always @ (input1 or input2 or aluCtrOut)
    begin
        case (aluCtrOut)
            4'b0000:  // and
                aluRes = input1 & input2;
            4'b0001:  // or
                aluRes = input1 | input2;
            4'b0010:  // add
                aluRes = input1 + input2;
            4'b0011:  // sll
                aluRes = input2 << input1;
            4'b0100:  // srl
                aluRes = input2 >> input1;
            4'b0101:  // no change
                aluRes = input1;
            4'b0110:  // sub
                aluRes = input1 - input2;
            4'b0111:  // slt
                aluRes = ($signed(input1) < $signed(input2));
            4'b1100:  // nor
                aluRes = ~(input1 | input2);
            default:
                aluRes = 0;
        endcase
        if (aluRes == 0)
            zero = 1;
```

```
        else
            zero = 0;
    end
```

3.4 寄存器 (Register)、存储器 (Data Memory) 和有符号扩展单元 (Sign Extension) 功能实现

寄存器 (Register)、存储器 (Data Memory) 和有符号扩展单元 (Sign Extension) 的实现与实验四的基本一致，但仍需注意的是寄存器中 reset 信号的响应、有符号扩展单元 signExt 信号的响应，完整代码可参考附录 A.4。

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

```
assign data = (signExt ? { {16 {inst[15]}}, inst[15 : 0] } : { 16'h0000, inst[15 : 0] });
```

3.5 指令存储器 (Instruction Memory) 功能实现

在第 2.5 节中我们介绍了指令存储器 (Instruction Memory) 的原理，完整代码可参考附录 A.5。

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

    reg [31 : 0] instFile [0 : 1023];
    assign inst = instFile[address >> 2];
endmodule
```

3.6 程序计数器模块 (PC) 功能实现

在第 2.6 节中我们详细介绍了程序计数器模块 (PC)，我们在此利用模块进行实现。完整的代码可参考附录 A.6。

```
module PC(
    input [31 : 0] pcIn,
    input clk,
    input reset,
    output reg [31 : 0] pcOut
    );

    initial pcOut = 0;

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

3.7 数据选择器 (Mux) 功能实现

在第 2.7 节中我们详细介绍了数据选择器 (Mux)，其分为 5 位数据选择器与 32 位数据选择器；两类数据选择器的完整代码可参考附录 A.7。

```
module Mux5(
    input select,
    input [4 : 0] input1,
    input [4 : 0] input2,
    output [4 : 0] out
    );
    assign out = select ? input1 : input2;
endmodule

module Mux32(
    input selectSignal,
    input [31 : 0] input1,
    input [31 : 0] input2,
    output [31 : 0] out
    );
    assign out = selectSignal ? input1 : input2;
```

```
endmodule
```

3.8 顶层模块 (Top) 功能实现

在第 2.8 节中我们详细介绍了顶层模块 (Top)，我们根据图 1 中的连线，定义如下数据线（每一条线都对应着图中的一条线）。

```
module Top(
        input clk,
        input reset
    );
    wire REG_DST, ALU_SRC, MEM_TO_REG, REG_WRITE, MEM_READ;
    wire MEM_WRITE, BRANCH, EXT_SIGN, JAL_SIGN, JUMP;
    wire SHAMT_SIGN, JR_SIGN, ALU_OUT_ZERO;

    wire [2 : 0] ALU_OP;
    wire [3 : 0] ALU_CTR_OUT;

    wire [4 : 0] WRITE_REG1;
    wire [4 : 0] WRITE_REG2;

    wire [31 : 0] INST_ADDR;
    wire [31 : 0] INST;
    wire [31 : 0] REG_OUT1;
    wire [31 : 0] REG_OUT2;
    wire [31 : 0] ALU_INPUT1;
    wire [31 : 0] ALU_INPUT2;
    wire [31 : 0] EXT_RES;
    wire [31 : 0] ALU_RES;
    wire [31 : 0] REG_WRITE_DATA;
    wire [31 : 0] REG_WRITE_DATA_T;
    wire [31 : 0] PC_IN;
    wire [31 : 0] PC_OUT;
    wire [31 : 0] MEM_READ_DATA;
    wire [31 : 0] PC_TEMP1;
    wire [31 : 0] PC_TEMP2;

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

```
36          .memToReg(MEM_TO_REG),
37          .regWrite(REG_WRITE),
38          .memRead(MEM_READ),
39          .memWrite(MEM_WRITE),
40          .branch(BRANCH),
41          .jump(JUMP),
42          .extSign(EXT_SIGN),
43          .jalSign(JAL_SIGN)
44      );
45      ALUCtr aluCtr (
46          .aluOp(ALU_OP),
47          .funct(INST[5 : 0]),
48          .aluCtrOut(ALU_CTR_OUT),
49          .shamtSign(SHAMT_SIGN),
50          .jrSign(JR_SIGN)
51      );
52      ALU alu (
53          .input1(ALU_INPUT1),
54          .input2(ALU_INPUT2),
55          .aluCtrOut(ALU_CTR_OUT),
56          .zero(ALU_OUT_ZERO),
57          .aluRes(ALU_RES)
58      );
59      Registers registers (
60          .readReg1(INST[25 : 21]),
61          .readReg2(INST[20 : 16]),
62          .writeReg(WRITE_REG1),
63          .writeData(REG_WRITE_DATA),
64          .regWrite(REG_WRITE & (~JR_SIGN)),
65          .clk(clk),
66          .reset(reset),
67          .readData1(REG_OUT1),
68          .readData2(REG_OUT2)
69      );
70      dataMemory dataMemory (
71          .clk(clk),
72          .address(ALU_RES),
73          .writeData(REG_OUT2),
74          .memWrite(MEM_WRITE),
75          .memRead(MEM_READ),
76          .readData(MEM_READ_DATA)
```

```
77      );
78      signext signExt (
79          .inst(INST[15 : 0]),
80          .signExt(EXT_SIGN),
81          .data(EXT_RES)
82      );
83      InstMemory instMemory (
84          .address(PC_OUT),
85          .inst(INST)
86      );
87      PC pc_controller (
88          .pcIn(PC_IN),
89          .clk(clk),
90          .reset(reset),
91          .pcOut(PC_OUT)
92      );
93      Mux32 branch_mux (
94          .select(BRANCH & ALU_OUT_ZERO),
95          .input1(PC_OUT + 4 + (EXT_RES << 2)),
96          .input2(PC_OUT + 4),
97          .out(PC_TEMP1)
98      );
99      Mux32 jr_mux (
100         .select(JR_SIGN),
101         .input1(REG_OUT1),
102         .input2(PC_TEMP2),
103         .out(PC_IN)
104     );
105     Mux32 jump_mux (
106         .select(JUMP),
107         .input1(((PC_OUT + 4) & 32'hf0000000) + (INST[25 : 0] << 2)),
108         .input2(PC_TEMP1),
109         .out(PC_TEMP2)
110     );
111     Mux32 jal_mux (
112         .select(JAL_SIGN),
113         .input1(PC_OUT + 4),
114         .input2(REG_WRITE_DATA_T),
115         .out(REG_WRITE_DATA)
116     );
117     Mux32 reg_shamt_mux (
```

```
118         .select(SHAMT_SIGN),
119         .input1({27'b0, INST[10 : 6]}),
120         .input2(REG_OUT1),
121         .out(ALU_INPUT1)
122     );
123     Mux32 alu_src_mux (
124         .select(ALU_SRC),
125         .input1(EXT_RES),
126         .input2(REG_OUT2),
127         .out(ALU_INPUT2)
128     );
129     Mux32 mem_to_reg_mux (
130         .select(MEM_TO_REG),
131         .input1(MEM_READ_DATA),
132         .input2(ALU_RES),
133         .out(REG_WRITE_DATA_T)
134     );
135     Mux5 reg_dst_mux (
136         .select(REG_DST),
137         .input1(INST[15 : 11]),
138         .input2(INST[20 : 16]),
139         .out(WRITE_REG2)
140     );
141     Mux5 jar_reg_mux (
142         .select(JAL_SIGN),
143         .input1(5'b11111),
144         .input2(WRITE_REG2),
145         .out(WRITE_REG1)
146     );
147 endmodule
```

4 结果验证

编写如下的汇编代码进行测试，

地址	数据	地址	数据	地址	数据	地址	数据
0x00	0x00000000	0x01	0x00000001	0x02	0x00000002	0x03	0x00000003
0x04	0x00000004	0x05	0x00000005	0x06	0x00000006	0x07	0x00000007

表 5: 内存中的初始值

指令地址	指令	指令解释	执行结果
0x00	100011 00000 00001 0000000000000000	lw $1, 0($0)	$1 = Mem[0] = 0
0x04	000010 00000000000000000000000010	j 2	go to 0x0c
0x08	000000 00000 00000 0000000000000000	nop	(not executed)
0x0c	100011 00000 00010 0000000000000001	lw $2, 1($0)	$2 = Mem[1] = 1
0x10	100011 00000 00011 0000000000000100	lw $3, 4($0)	$3 = Mem[4] = 4
0x14	100011 00000 00100 0000000000000111	lw $4, 7($0)	$4 = Mem[7] = 7
0x18	001000 00001 00101 0000000100000000	addi $5, $1, 256	$5 = 0 + 256 = 256
0x1c	001000 00010 00110 0000000011111111	addi $6, $2, 255	$6 = 1 + 255 = 256
0x20	001100 00011 00111 0000000011111111	andi $7, $3, 255	$7 = 4 & 255 = 4
0x24	001101 00100 01000 0000000011111111	ori $8, $4, 255	$8 = 7 | 255 = 255
0x28	101011 00101 00101 0000000000001011	sw $5, 11($5)	Mem[16] = 256
0x2c	000100 00101 00110 0000000000000001	beq $5, $6, 1	go to 0x3c
0x30	000000 00000 00000 0000000000000000	nop	(not executed)
0x34	000000 00000 00111 01001 00010 000000	sll $9, $7, 2	$9 = 4 « 2 = 16
0x38	000000 00001 01000 01010 00000 100000	add $10, $1, $8	$10 = 0 + 255 = 255
0x3c	000000 00101 00010 01011 00000 100010	sub $11, $5, $2	$11 = 256 - 1 = 255
0x40	000000 00110 00100 01100 00000 100100	and $12, $6, $4	$12 = 256 & 7 = 0
0x44	000000 01000 00111 01101 00000 100101	or $13, $8, $7	$13 = 255 & 4 = 255
0x48	000000 00000 00110 01110 00010 000010	srl $14, $6, 2	$14 = 256 » 2 = 64
0x4c	000000 01100 01011 01111 00000 101010	slt $15, $12, $11	$15 = 1
0x50	000000 01110 01101 10000 00000 101010	slt $16, $14, $13	$16 = 1
0x54	000011 00000000000000000000010111	jal 23	go to 0x5c
0x58	000100 01001 01110 0000000000000010	beq $9, $14, 2	go to 0x64
0x5c	000000 00000 01001 01001 00010 000000	sll $9, $9, 2	$9 = 16 « 2 = 64
0x60	000000 11111 000000000000000 001000	jr $31	go to 0x58
0x64	000000 01010 01011 01010 00000 100000	add $10, $10, $11	$10 = 255 + 255 = 510

表 6: 汇编代码及其解释、执行结果

其中，执行结果中表明“(not executed)”表示该指令由于跳转等原因没有被执行。可以看出，我们的测试汇编代码完整测试了本实验要求的所有的 16 条汇编指令，因此该测试结果能够反映出处理器的整体实现准确与否。

我们在激励文件中使用 readmemh 命令将其读入存储器模块的对应位置进行测试。

```
1 $readmemh("D:/Archlabs/lab05/mem_data.txt", cpu.dataMemory.memFile);
2 $readmemb("D:/Archlabs/lab05/mem_inst.txt", cpu.instMemory.instFile);
```

我们用上述汇编命令以及内存的初始情况进行测试，测试结果如图 2 所示。

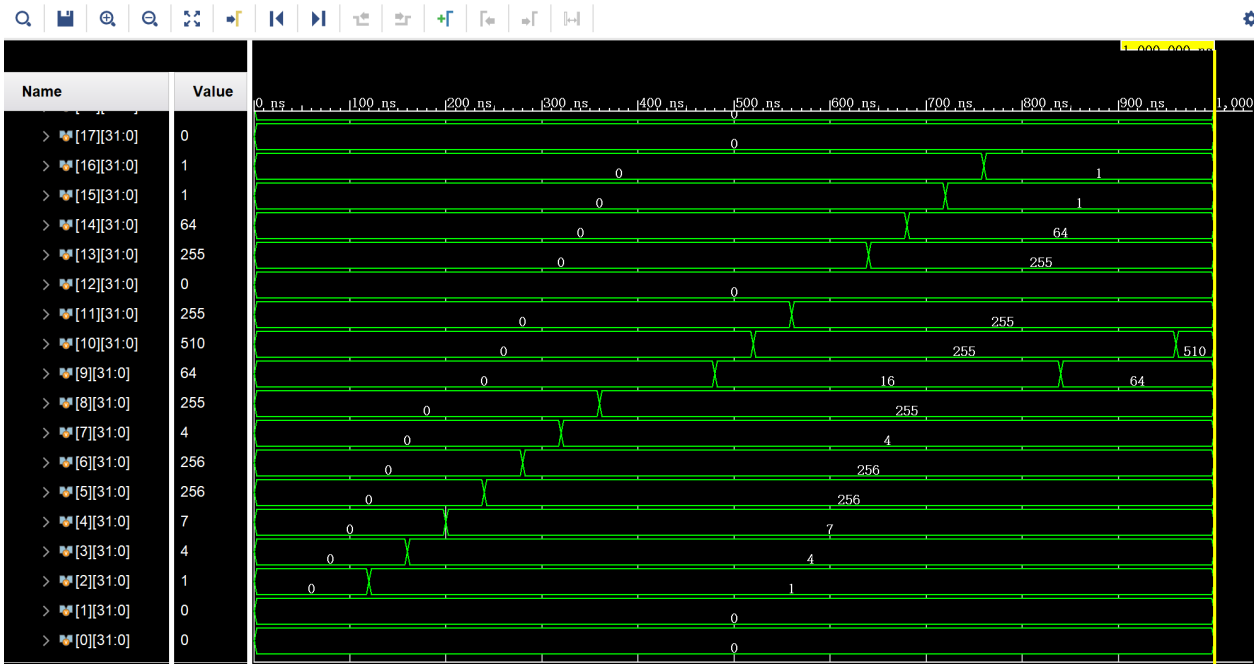

图 2: 单周期 MIPS 处理器测试结果

对照图 2 与表 6 中的执行结果可知，运行结果完全正确，仿真成功，说明整个单周期 MIPS 处理器实现正确。

5 总结与反思

本实验实现了一个完整的支持 16 指令的单周期 MIPS 处理器，以实验三、四的模块为基础，添加部分其他模块后，将模块连线组装成一个完整的单周期 MIPS 处理器。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。

通过这次实验，我对于 MIPS 处理器的数据通路、信号通路等都有了一个更加清晰地了解，也对单周期 MIPS 有了更加深刻的理解。同时，在调试的过程中，我对 MIPS 处理器的指令也有了更加深刻的理解；通过自行编写对应的汇编代码、手动模拟汇编代码的运行结果、与自己写的处理器的运行结果进行对照等过程，对汇编代码也有了更加深刻的理解。

附录 A　设计文件代码实现

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)、存储器 (Data Memory) 和有符号扩展单元 (Sign Extension) 的代码实现

参见代码文件 Registers.v，dataMemory.v 及 signext.v。

A.5　指令存储器 (Instruction Memory) 的代码实现

参见代码文件 instMemory.v。

A.6　程序计数器模块 (PC) 的代码实现

参见代码文件 PC.v。

A.7　数据选择器 (Mux) 的代码实现

参见代码文件 Mux5.v 与 Mux32.v。

A.8　顶层模块 (Top) 的代码实现

参见代码文件 Top.v。

附录 B　激励文件代码实现

参见代码文件 Top_tb.v。