实验 9 单周期处理器实验
第 7 次实验报告
22336313 郑鸿鑫

一．实验准备

本次实验开始整合之前的实验内容，搭建完整的单周期处理器 datapath。在前面几次实验中，分别完成了 ALU 设计和寄存器器堆的设计；掌握了存储器 IP 的使用；实现了单周期 CPU 的取指、译码阶段，完成了 PC、控制器的设计。通过前面 的几次实验，单周期 CPU 的设计的各模块已经具备，再引入数字逻辑课程中所实现的 多路选择器、加法器等门级组件，通过对原理图的理解，分析单条（单类型）指令在数 据通路中的执行路径，依次连接对应端口，将各个模块通过搭积木一样组合起来即可完成单周期 CPU。

二．实验目的

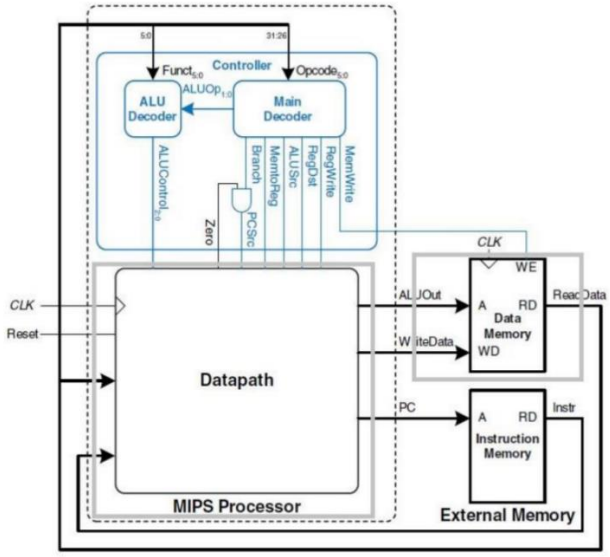
1. 掌握单周期 CPU 数据通路图的构成、原理及其设计方法；
2. 掌握单周期 CPU 的实现方法，代码实现方法；
3. 认识和掌握指令与 CPU 的关系；
4. 掌握测试单周期 CPU 的方法。

三．实验内容

下图是单周期 CPU 的框架图，本实验需要实现 datapath 模块以实现整个单周期处理器的数据通路。其中主要包括 alu 和寄存器堆（实验 7 alu 与寄存器堆实验已实现，存储器（实验 6 存储器实

验已实现），PC 和 controller（实验 8 控制器实验已完成），其他相关模块在数字逻辑课程或前面的课程都已提供。

指令存储器（采用 Single Port Rom）数据存储器（采用 Single Port Ram）使用 Block Memory Generator IP 构造指令，并需要考虑 Pc 地址位数的统一。



下图是单周期 CPU 系统连线图，本实验将依照下图将前面实验的信号和数据正确连接，编写 datapath 模块代码。

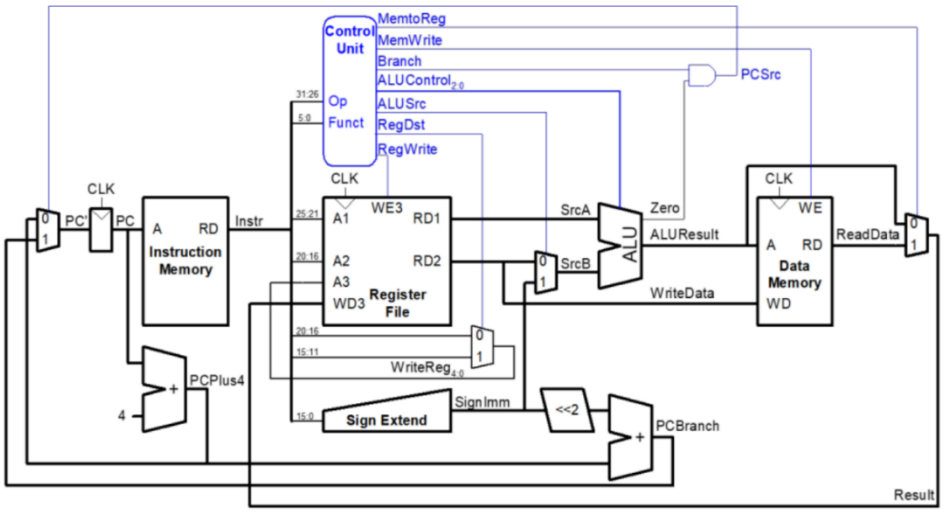
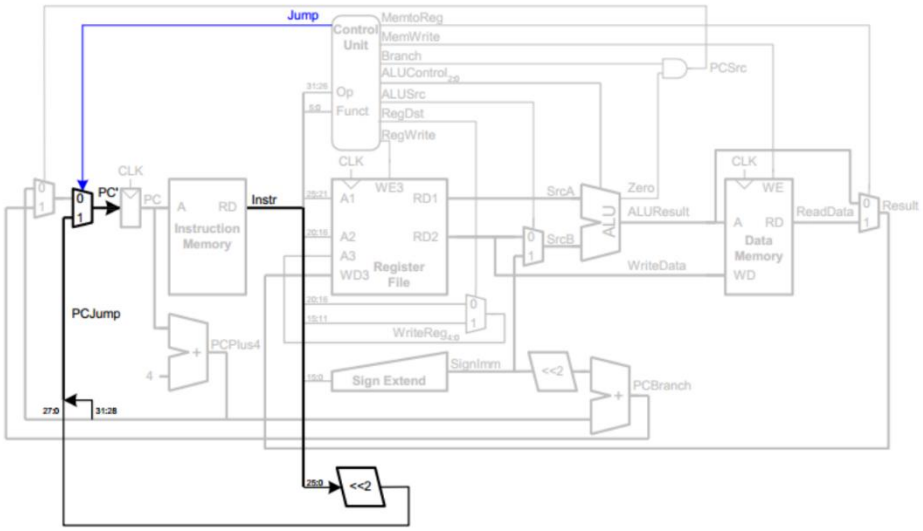


图 2：单周期 CPU 系统连线图

注意在上图的基础上应加上 jump 信号控制的通路如下图



四．实验过程

1. 从前面的实验中，导入 alu 模块和寄存器堆模块；
2. 从控制器实验中导入 PC、Controller 模块；
3. 从提供的基础模块中导入多路选择器、加法器模块；
4. 根据存储器实验使用 Block Memory，其中 inst_mem 导入 coe 文件；
5. 参考实验原理，连接各模块；
6. 导入顶层文件及仿真文件，运行仿真；

模块层次如下：

依照上述信息编写 DATAPATH 模块代码如下：（说明已写在注释中）

```
module datapath(
    input wire clk,rst,
    input wire memtoreg,pcsrc,
    input wire alusrc,regdst,
    input wire regwrite,jump,
    input wire[2:0] alucontrol,
    output wire overflow,zero,
    output wire[31:0] pc,
    input wire[31:0] instr,
    output wire[31:0] aluout,writedata,
    input wire[31:0] readdata
    );
    wire [31:0] a,b,wd3,b1,pcadr1,pc_plus4,pcbr,PC0,PC1,pc1,PC;
    //中间信号声明：
    //a,b 为alu 的两个操作数
    //wd3 为要写入寄存器堆的数
    //b1 为指令的低 16 位经符号扩展后输入，当alusrc 为 1 时将其作为alu 的第二个输入
    //pcadr1 当pcsrc 为 1 的时候时会被选择为下一条指令的地址
    //pc_plus4 由当前指令计数器的值加 4 得到，可直接用于作为下一条指令的pc 也可以用于计算
pcadr1
    //pcbr 由 b1 左移两位得到，用于和pc_plus4 相加得到pcadr1
```

```
    //PC0 是由 pc_plus4 和 pcadr1 由一个控制信号 pcsrc 选择得到，然后最终再接入一个多选器，由 jump 信号确定最终下一条的 PC 的地址
    //pc1 是由指令的低 25 位左移 2 位得到的，将偏移的字数转换位字节数
    //PC1 是由 pc_plus4 的高 4 位与 pc1 的低 28 位合成，如果 jump 信号为 1，则选择 PC1 为下一条 PC，否则选择 PC0
    wire [4:0] wa3;//代表写入的寄存器编号，由于只需 5 位即可表示所有的 32 个寄存器
    mux2 #(5) m1(instr[20:16],instr[15:11],regdst,wa3);//用于选择哪个作为写入的寄存器
    mux2 m2(writedata,b1,alusrc,b);//选择哪个作为 ALU 的第二个输入
    mux2 m3(aluout,readdata,memtoreg,wd3);//选择哪个作为写入寄存器的值
    mux2 m4(pc_plus4,pcadr1,pcsrc,PC0);//选择哪个为下一个 Pc
    mux2 m5(PC0,PC1,jump,PC);//Jump 指令
    regfile Reg(clk,regwrite,instr[25:21],instr[20:16],wa3,wd3,a,writedata);
    alu aa(a,b,alucontrol,aluout,overflow,zero);
    signext si(instr[15:0],b1);//对低 16 位进行符号扩展
    sl2 sl(b1,pcbr);
    sl2 sll({6'b0,instr[25:0]},pc1);
    assign PC1 = {{pc_plus4[31:28]},{pc1[27:0]}};
    adder ad_br(pcbr,pc_plus4,pcadr1);
    adder pc_ad4(pc,32'h4,pc_plus4);
    flopr fl(clk,rst,PC,pc);
endmodule
```

控制器代码如下

```
module controller(
    input wire[5:0] op,funct,
    input wire zero,
    output wire memtoreg,memwrite,
    output wire pcsrc,alusrc,
    output wire regdst,regwrite,
    output wire jump,
    output wire[2:0] alucontrol
    );
    wire[1:0] aluop;
    wire branch;
    maindec md(op,memtoreg,memwrite,branch,alusrc,regdst,regwrite,jump,aluop);
    aludec ad(funct,aluop,alucontrol);
    assign pcsrc = branch & zero;
endmodule
```

其中 maindec 代码如下：

```
module maindec(
    input wire[5:0] op,
    output reg memtoreg,memwrite,
    output reg branch,alusrc,
    output reg regdst,regwrite,
    output reg jump,
    output reg[1:0] aluop
    );
    always@(*)begin
        case(op)
        6'b000000:begin
            regwrite = 1;
            regdst = 1;
            alusrc = 0;
            branch = 0;
            memwrite = 0;
            memtoreg = 0;
            aluop = 2'b10;
            jump = 0;
        end
        6'b000010:begin
            regwrite = 0;
            memwrite = 0;
            jump = 1;
        end
```

```
        6'b100011:begin
            regwrite = 1;
            regdst = 0;
            alusrc = 1;
            branch = 0;
            memwrite = 0;
            memtoreg = 1;
            aluop = 2'b00;
            jump = 0;
        end
        6'b101011:begin
            regwrite = 0;
            alusrc = 1;
            branch = 0;
            memwrite = 1;
            aluop = 2'b00;
            jump = 0;
        end
        6'b000100:begin
            regwrite = 0;
            alusrc = 0;
            branch = 1;
            memwrite = 0;
            aluop = 2'b01;
            jump = 0;
        end
        6'b001000:begin
            regwrite = 1;
            regdst = 0;
            alusrc = 1;
            branch = 0;
            memwrite = 0;
            memtoreg = 0;
            aluop = 2'b00;
            jump = 0;
        end
        default: begin
            regwrite = 0;
            regdst = 0;
            alusrc = 0;
            branch = 0;
            memwrite = 0;
            memtoreg = 0;
            aluop = 2'b00;
            jump = 0;
        end
    endcase
    end
endmodule
```

aludec 的代码如下：

```
module aludec(
    input wire[5:0] funct,
    input wire[1:0] aluop,
    output reg[2:0] alucontrol
    );
    // add your code here
    always @(*) begin
        case(aluop)
        2'b00: begin
            alucontrol = 3'b010;
        end
        2'b01: begin
            alucontrol = 3'b110;
        end
        2'b10: begin
```

```
            case(funct)
            6'b100000: alucontrol = 3'b010;
            6'b100010: alucontrol = 3'b110;
            6'b100100: alucontrol = 3'b000;
            6'b100101: alucontrol = 3'b001;
            6'b101010: alucontrol = 3'b111;
            endcase
        end
        default: alucontrol = 3'b000;
        endcase
    end
endmodule
```

五．实验结果分析

coe 代码如下：

```
memory_initialization_radix = 16;
memory_initialization_vector =
20020005,
2003000c,
2067fff7,
00e22025,
00642824,
00a42820,
10a7000a,
0064202a,
10800001,
20050000,
00e2202a,
00853820,
00e23822,
ac670044,
8c020050,
08000011,
20020001,
ac020054;
```

TEST_BENCH 代码如下：

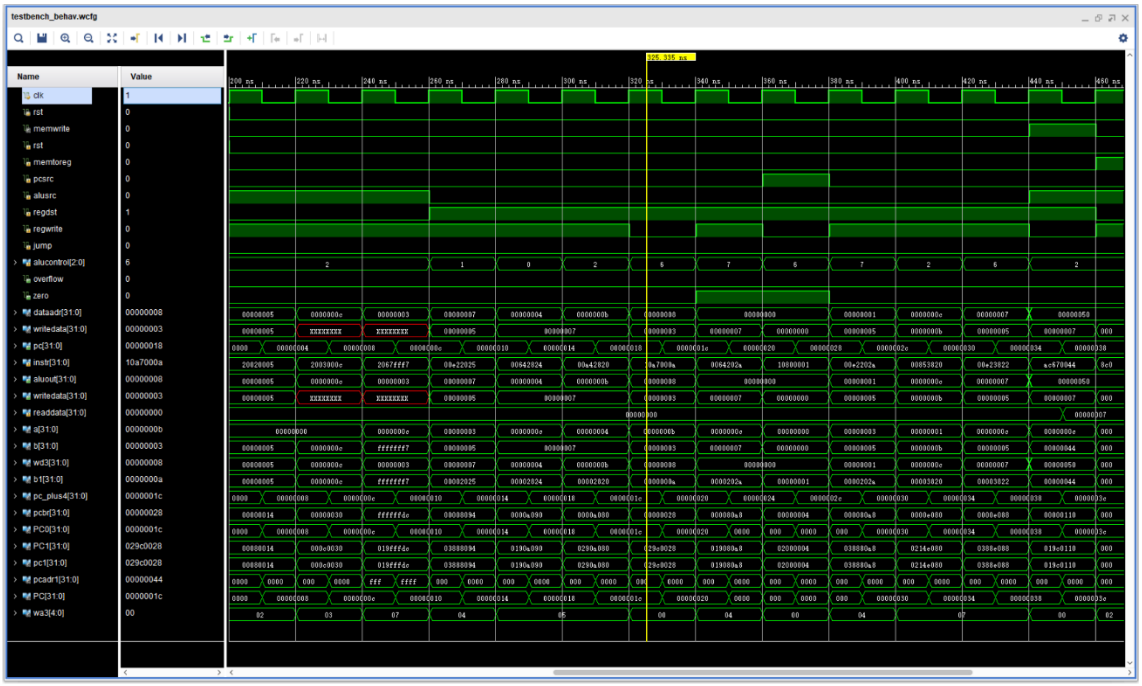
```
module testbench();
    reg clk;

    reg rst;
    wire[31:0] writedata,dataadr;
    wire memwrite;
    top dut(clk,rst,writedata,dataadr,memwrite);
    initial begin
        rst <= 1;
        #200;
        rst <= 0;
    end
    always begin
        clk <= 1;
        #10;
        clk <= 0;
```

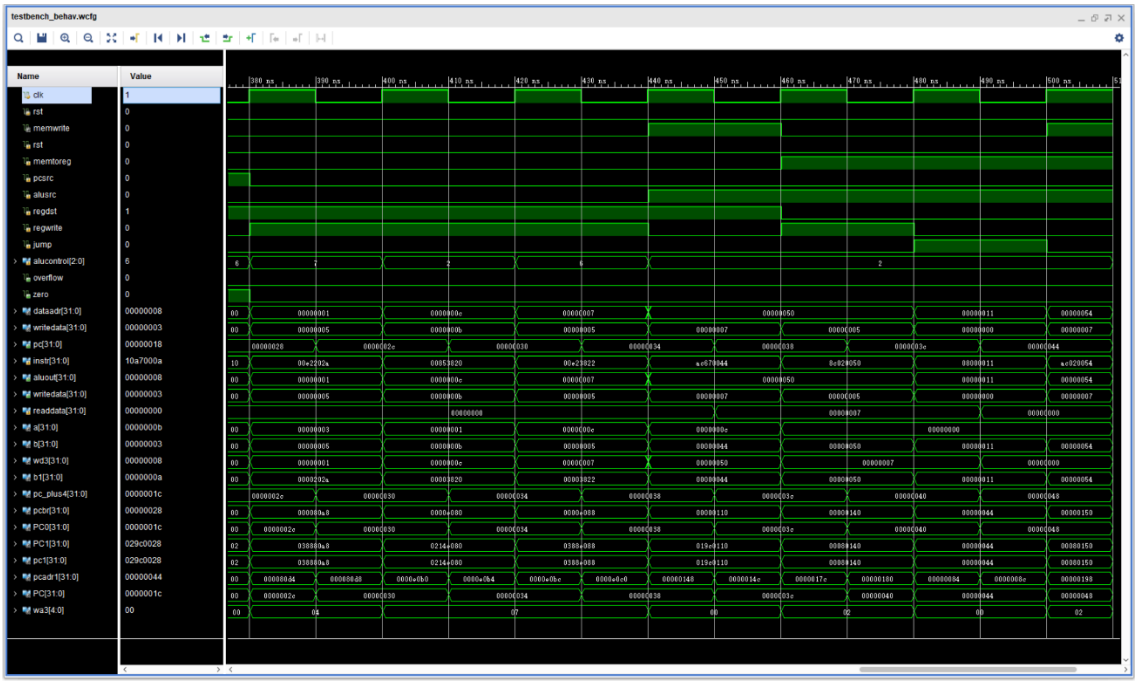
```
        #10;
    end
    always @(negedge clk) begin
        if(memwrite) begin
            if(dataadr === 84 && writedata === 7) begin
                /* code */
                $display("Simulation succeeded");
                $stop;
            end
        end
    end
endmodule
```

TOP 模块的代码如下：

```
module top(
    input wire clk,rst,
    output wire[31:0] writedata,dataadr,
    output wire memwrite
    );
    wire[31:0] pc,instr,readdata;
    mips mips(clk,rst,pc,instr,memwrite,dataadr,writedata,readdata);
    Inst_Rom imem(clk,pc[9:2],instr);
    Data_Ram dmem(~clk,memwrite,dataadr,writedata,readdata);
endmodule
```

通过综合仿真得到如下波形图：





将 coe 文件与汇编语句相对应得到下图：

#	Assembly	Description	Address	Machine	
main:	addi $2, $0, 5	# initialize $2 = 5	0	20020005	20020005
	addi $3, $0, 12	# initialize $3 = 12	4	2003000c	2003000c
	addi $7, $3, -9	# initialize $7 = 3	8	2067fff7	2067fff7
	or $4, $7, $2	# $4 <= 3 or 5 = 7	c	00e22025	00e22025
	and $5, $3, $4	# $5 <= 12 and 7 = 4	10	00642824	00642824
	add $5, $5, $4	# $5 = 4 + 7 = 11	14	00a42820	00a42820
	beq $5, $7, end	# shouldn't be taken	18	10a7000a	10a7000a
	slt $4, $3, $4	# $4 = 12 < 7 = 0	1c	0064202a	0064202a
	beq $4, $0, around	# should be taken	20	10800001	10800001
	addi $5, $0, 0	# shouldn't happen	24	20050000	20050000
around:	slt $4, $7, $2	# $4 = 3 < 5 = 1	28	00e2202a	00e2202a
	add $7, $4, $5	# $7 = 1 + 11 = 12	2c	00853820	00853820
	sub $7, $7, $2	# $7 = 12 - 5 = 7	30	00e23822	00e23822
	sw $7, 68($3)	# [80] = 7	34	ac670044	ac670044
	lw $2, 80($0)	# $2 = [80] = 7	38	8c020050	8c020050
	j end	# should be taken	3c	08000011	08000011
	addi $2, $0, 1	# shouldn't happen	40	20020001	20020001
end:	sw $2, 84($0)	# write adr 84 = 7	44	ac020054	ac020054

说明：由于空间有限，没有将所有控制信号和中间信号都显示出来，而是选择相对关键的控制信号作为展示。

下面是对仿真波形图的检查说明：

1.第一条指令 20020005 对应的汇编指令是一条 I 型指令，代表将 0 号寄存器的值 0 和立即数 5 相加赋给 2 号寄存器，可以看到 a 为 0，b 为 5 是 ALU 的两个源操作数，aluout 和 writedata 为 5，和预期相符合；

2．接下来两条指令也是 I 型指令，同理可知符合预期；

3.接下来，两个读寄存器分别为 7 号和 2 号，写回寄存器选择的是 4 号，aluout 为 7（3 和 5 按位或的结果），和预期符合；

4.接下来 add 指令和 and 指令也同理，检查知符合实验预期；

5.然后到第一条 beq 指令，由仿真图中可以看出 10a7000a 的下一条指令是 0064202a，并未发生跳转，而是采用 PC+4 为下一条指令的地址，这是因为 5 号寄存器中存的是 11，7 号寄存器中存的是 3，所以不相等不产生跳转，符合预期；

6.可以看到第二条 beq 指令是在 PC 等于 20 的时候，此时检查 4 号寄存器中的值是否为 0，发现恰好为 0，所以应该跳转，由仿真图可以看出，该指令 10800001 的下一条是 00e2202a，跳过了它原本的下一条指令 20050000，跳到 around 地址对应的指令，也符合预期；

7.接下来可以看到 sw 指令，此时 writedata 变为 7，memwrite 变为 1，符合预期；

8.紧接着 lw 指令 readdata 变为 7 为相应访存的位置的值，其余时间 readdata 都为 0，也符合预期；

9.最后一个检查的点是 j 指令，当 PC 为 3c，指令为 08000011 时，需要跳过它的下一条指令 20020001，跳转到 end 的位置，对应 ac020054，观察仿真波形图可知，成功跳过，符合预期。

六．实验总结

实验结果与预期符合良好，仿真波形图经检查发现无误。

实验中遇到的问题：

1.注意多选器的输入输出位数问题：

本实验中有的多选器要选择的是 32 位也有 5 位，为了不用重复添加模块，采用参数将他们统一起来

```
module mux2 #(parameter WIDTH = 32)(
    input wire[WIDTH-1:0] d0,d1,
    input wire s,
    output wire[WIDTH-1:0] y
    );
    assign y = s ? d1 : d0;//信号为1选择d1，0选择d0
endmodule
```

在对应的位数和默认参数不同时，在实例化时对参数赋值：

```
mux2 #(5) m1(instr[20:16],instr[15:11],regdst,wa3);//用于选择哪个作为写入的寄存器
mux2 m2(writedata,b1,alusrc,b);//选择哪个作为ALU的第二个输入
mux2 m3(aluout,readdata,memtoreg,wd3);//选择哪个作为写入寄存器的值
mux2 m4(pc_plus4,pcadr1,pcsrc,PC0);//选择哪个为下一个Pc
mux2 m5(PC0,PC1,jump,PC);
```

如上图在 M1 实例化的时候将参数改为 5 位

2．实验过程中出现我认为代码无问题但波形图甚至连 PC 都是 XXXX 的情况，最终检查发现，有一个使用到的中间信号并未提前声明，所以将所有使用到的中间信号全部提前声明，改正后得到正确结果。