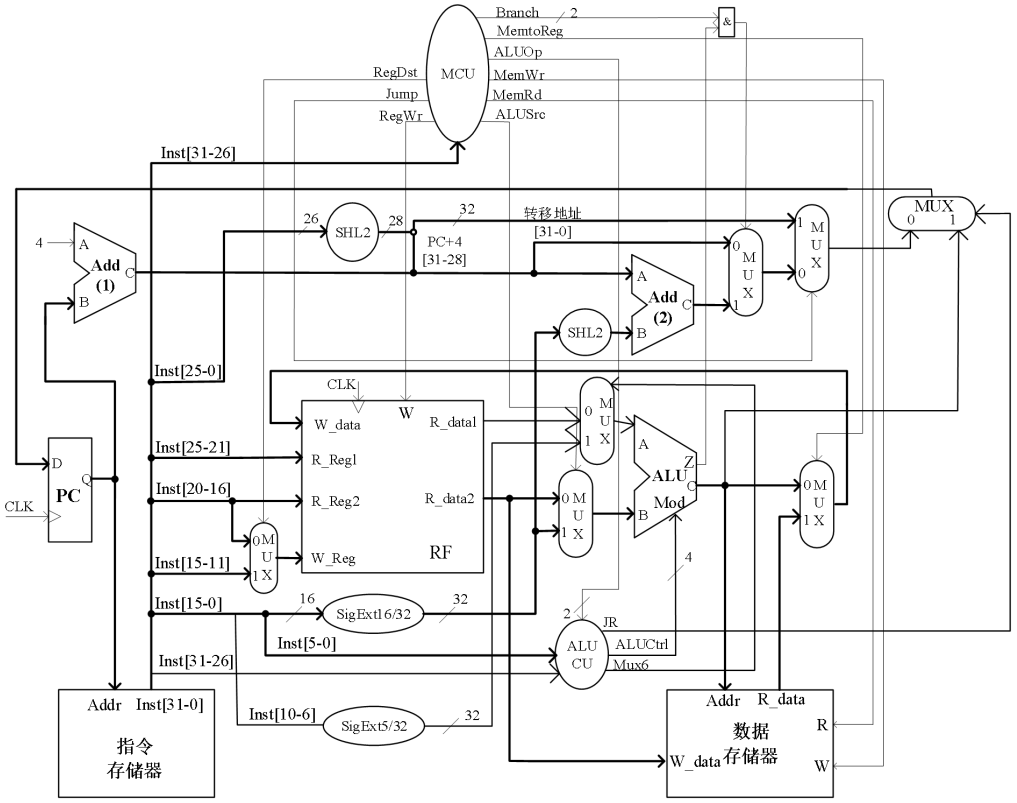
单周期 CPU 实验报告：

1. **实验目的及内容：**使用 verilog 语言编写单周期 CPU，并尽可能多地实现 MIPS 指令集中的指令。

2. 实验过程：

数据通路图：以教材 P391 页的数据通路图为基础，设计能实现更多指令的数据通路。



实现的指令集：

类型	指令	编码						操作
		31-26	25-21	20-16	15-11	10-6	5-0	
R-型	ADD	000000	rs	rt	rd	00000	100000	rd<-rs+rt
	ADDU	000000	rs	rt	rd	00000	100001	rd<-rs+rt
	SUB	000000	rs	rt	rd	00000	100010	rd<-rs-rt
	SUBU	000000	rs	rt	rd	00000	100011	rd<-rs-rt
	AND	000000	rs	rt	rd	00000	100100	rd<-rs&rt

	OR	000000	rs	rt	rd	00000	100101	rd<-rs|rt
	XOR	000000	rs	rt	rd	00000	100110	rd<-rs^rt
	NOR	000000	rs	rt	rd	00000	100111	rd<-~(rs|rt)
	SLT	000000	rs	rt	rd	00000	101010	rd<-(rs<rt)?1:0
	SLTU	000000	rs	rt	rd	00000	101011	rd<-(rs<rt)?1:0
	SLL	000000	00000	rt	rd	shamt	000000	rd<-rt<<shamt
	SLLV	000000	rs	rt	rd	00000	000100	rd<-rt<<rs
	SRL	000000	00000	rt	rd	shamt	000010	rd<-rt>>shamt
	SRLV	000000	rs	rt	rd	00000	000110	rd<-rt>>rs
	SRA	000000	00000	rt	rd	shamt	000011	rd<-rt>>shamt
	SRAV	000000	rs	rt	rd	00000	000111	rd<-rt>>rs
	JR	000000	rs	00000	00000	00000	001000	PC<-rs
I-型	LW	100011	rs	rt	immediate			rt<-memory[rs+immediate]
	SW	101011	rs	rt	immediate			Memory[rs+immediate]<-rt
	ADDI	001000	rs	rt	immediate			rt<-rs+immediate
	ADDIU	001001	rs	rt	immediate			rt<-rs+immediate
	ANDI	001100	rs	rt	immediate			rt<-rs&immediate
	ORI	001101	rs	rt	immediate			rt<-rs|immediate
	XORI	001110	rs	rt	immediate			rt<-rs^immediate
	LUI	001111	0000	rt	immediate			rt<-immediate<<16
	SLTI	001010	rs	rt	immediate			rt<-(rs<immediate)?1:0
	SLTIU	001011	rs	rt	immediate			rt<-(rs<immediate)?1:0
	BEQ	000100	rs	rt	immediate			If(rs==rt) PC<-PC+4+immediate<<2
	BNE	000101	rs	rt	immediate			If(rs!=rt) PC<-PC+4+immediate<<2
J-型	J	000010	address					PC<-address<<2

控制单元 CU 的实现：

根据输入的指令码确定控制信号：依据{RegDst,RegWr,ALUSrc,MemRd,MemWr,MemtoReg,Branch,Jump,ALUOp}的顺序输出控制信号。

Inst[31-26]	信号
000000	11000000010
100011	01110100000
101011	x0101x00000
000100	x0000x11001
000101	x0000x10001
000010	xx0x00x001xx

001xxx	01100000011

ALUCU 的实现：考虑到需要实现 I 型代码，即传入立即数作为 ALU 操作数，因此 ALUCU 需要判断这种情况，需要将操作码 Inst[31-26]也传入 ALUCU，同时 ALUCU 产生控制跳转的 JR 控制信号和控制操作数来源的 MUX6 信号。同时，为区分实现 R 型指令传入的 Inst[5-0]实现 I 型指令传入的 Inst[31-26]，修改了由 CU 传入的 ALUOp 分配方案，当为 R 型指令时，ALUOp 为 10，为 I 型时，ALUOp 为 11，00 和 01 分配给 LW 和 SW 之类的指令。

ALUOp	Inst[5-0]	Inst[31-26]	ALUCtrl	JR	MUX6
00	-	-	0000	0	0
01	-	-	0010	0	0
10	100000	-	0000	0	0
	100001	-	0001	0	0
	100010	-	0010	0	0
	100011	-	0011	0	0
	100100	-	0100	0	0
	100101	-	0101	0	0
	100110	-	0110	0	0
	100111	-	0111	0	0
	101010	-	1010	0	0
	101011	-	1011	0	0
	000000	-	1100	0	1
	000010	-	1110	0	1
	000011	-	1111	0	1
	000100	-	1100	0	0
	000110	-	1110	0	0
	000111	-	1111	0	0
	001000	-	1000	1	0
11	-	001000	0000	0	0
	-	001001	0001	0	0
	-	001100	0100	0	0
	-	001101	0101	0	0
	-	001110	0101	0	0
	-	001111	1001	0	0
	-	001010	1010	0	0
	-	001011	1011	0	0

ALU 的实现：由于 ALUCtrl 已经扩充为四位，ALU 内部对不同 ALUCtrl 的采取的不同操作如下：

```
module ALU(
    input [31:0] ReadData1,
    input [31:0] ReadData2,
    input [31:0] inExt,
    input ALUSrc,
    input [3:0] ALUCtrl,
    output reg zero,
    output reg [31:0] result);
wire [31:0] A;
wire [31:0] B;

assign A=ReadData1;
assign B=ALUSrc ? inExt:ReadData2;
always @(*)
    begin
    case(ALUCtrl)
        4'b0000: begin    //add
            result = $signed(A) + $signed(B);
            zero = (result == 0)? 1 : 0;
        end
                4'b0001: begin    //addu
            result = A + B;
            zero = (result == 0)? 1 : 0;
        end
                4'b0010: begin    //sub
            result = $signed(A) - $signed(B);
            zero = (result == 0)? 1 : 0;
        end
                4'b0011: begin    //subu
            result = A - B;
            zero = (result == 0)? 1 : 0;
        end
                4'b0100: begin    //and
            result = A & B;
            zero = (result == 0)? 1 : 0;
        end
                4'b0101: begin    //or
            result = A | B;
            zero = (result == 0)? 1 : 0;
        end
```

```
                4'b0110: begin    //xor
                result = A ^ B;
                zero = (result == 0)? 1 : 0;
            end
                4'b0111: begin    //nor
                result = ~(A | B);
                zero = (result == 0)? 1 : 0;
            end
            4'b1000: begin    //jr
                result = A;
                zero = (result == 0)? 1 : 0;
            end
            4'b1001:begin      //lui
                result = B << 16;
                zero = (result == 0)? 1 : 0;
            end
            4'b1010: begin    //slt
                result = ($signed(A) < $signed(B))?1:0;
                zero = (result == 0)? 1 : 0;
            end
            4'b1011: begin    //sltu unsigned
                result = (A < B)? 1 : 0;
                zero = (result == 0)? 1 : 0;
            end
            4'b1100: begin    //sllv or sll
                result = B << A;
                zero = (result == 0)? 1 : 0;
            end
            4'b1110: begin    //srlv or srl
                result = B >> A;
                zero = (result == 0)? 1 : 0;
            end
            4'b1111: begin    //srav arithmatic
                result = ( $signed(B) ) >>> A;
                zero = (result == 0)? 1 : 0;
            end
        endcase
        end
endmodule
```

额外部件说明：

And 部件：

数据通路图中 Branch 变为两位以决定通过何种方式进行获得下一条

指令的地址(相等跳转、不等跳转、顺序执行)，因此需要一个原件对 Branch 和 zero 的值的组合进行判断并输出控制信号，该元件接受 Branch 和 zero 输入并给出控制信号，通路图中仍以“&”原件表示，但在源代码中以“And”原件形式存在。这里将 Branch 的值 00 分配给顺序跳转，10 分配给 BNE，11 分配给 BEQ。

And 原件的输入输出对应表如下:

Branch	zero	out
0x	-	0
10	0	1
10	1	0
11	0	0
11	1	1

写成逻辑表达式即为：out=Branch[1] & (Branch[0] ^~ zero)

SigExt5/32 部件：

由于 R 型指令中部分指令存在传入 5 位立即数 shamt 的情况，因此需要对立即数进行位扩充以满足 ALU 运算要求。该部件用于将 5 位立即数符号扩展为 32 位。

验证方式：

将已经转换为二进制代码的指令存储于 IM 中，并初始化 RF 和 DataMem，在顶层测试模块中给予 CPU 激励（clk）和 PC 的 reset 信号，并将存储结果的单元单独连出，观察仿真波形图并验证结果是否正确。

CPU 连接代码如下：

```
module Single_CPU(input clk,reset);
wire[31:0]
PC_in,PC_out,add4_out,Inst,SignExt_out,R_data1,R_data2,ALU_out,DMR_data,RFW_data;
wire RegDst,Jump,RegWr,ALUSrc,MemRd,MemWr,MemtoReg,JR;
```

```
wire[1:0] ALUOp,Branch;
wire[3:0] ALUCtrl;
wire[4:0] mux1_out;
wire[31:0] mux3_out;
wire[31:0] add2_out;
wire[27:0] shl2_out;
wire[31:0] mux4_out;
wire[31:0] mux6_out;
wire Mux6;
wire[31:0] mux6_i1;
wire zero;
wire And_out;

PC pc(clk,reset,PC_in,PC_out);
Add4 add4(PC_out,add4_out);
IM im(PC_out,Inst);
SignExt16_32 signext16_32(Inst[15:0],SignExt_out);
CU cu(Inst[31:26],RegDst,Jump,RegWr,MemtoReg,MemWr,MemRd,ALUSrc,ALUOp,Branch);
mux2_to_1 mux1(mux1_out,Inst[20:16],Inst[15:11],RegDst);defparam mux1.width=5;
RegFiles rf(clk,RegWr,RFW_data,Inst[25:21],Inst[20:16],mux1_out,R_data1,R_data2);
ALUCU alucu(Inst[5:0],Inst[31:26],ALUOp,ALUCtrl,JR,Mux6);
ALU alu(mux6_out,R_data2,SignExt_out,ALUSrc,ALUCtrl,zero,ALU_out);
SignExt5_32 sig(Inst[10:6],mux6_i1);
mux2_to_1 mux6(mux6_out,R_data1,mux6_i1,Mux6);
And _And(Branch,zero,And_out);
Add add2(add4_out,SignExt_out<<2,add2_out);
mux2_to_1 mux3(mux3_out,add4_out,add2_out,And_out);
SHL2_26 shl2_26(Inst[25:0],shl2_out);
mux2_to_1 mux4(mux4_out,mux3_out,{add4_out[31:28],shl2_out},Jump);
mux2_to_1 mux5(PC_in,mux4_out,ALU_out,JR);
DataMem dm(MemRd,MemWr,ALU_out,R_data2,DMR_data);
mux2_to_1 mux2(RFW_data,ALU_out,DMR_data,MemtoReg);

Endmodule
```

顶层测试模块代码如下：

```
module S_CPUt;
reg clk;
reg reset;
wire [31:0]aluout;
wire [31:0]i;
wire [31:0]result;
wire [31:0]A,R_data2;
wire [3:0]ctrl;
```

```
wire zero;
//wire [4:0]out5;
initial begin
clk=0;
reset=1;
end

always #20 begin
clk=~clk;
reset=0;
end

Single_CPU cpu(clk,reset);
assign aluout=cpu.ALU_out;
assign result=cpu.dm.memory[5];//cpu.rf.regs[1];//for visualizing the target result
assign A=cpu.mux6_out;
assign R_data2=cpu.R_data2;
assign ctrl=cpu.ALUCtrl;

assign zero=cpu.zero;

assign i=cpu.PC_out;

endmodule
```

测试用指令代码如下：

```
//initialize for test

//{memory[0],memory[1],memory[2],memory[3]}=32'b000000_00000_00001_00111_00000_100000;//add:rf[7]=rf[0]+rf[1]...addu
//{memory[0],memory[1],memory[2],memory[3]}=32'b000000_00000_00001_00111_00000_100010;//sub:rf[7]=rf[0]-rf[1]...subu

//{memory[0],memory[1],memory[2],memory[3]}=32'b000000_00010_00011_00111_00000_100100;//and:rf[7]=rf[2]&rf[3]
//{memory[0],memory[1],memory[2],memory[3]}=32'b000000_00000_00001_00111_00000_100101;//or:rf[7]=rf[0] | rf[1]
//{memory[0],memory[1],memory[2],memory[3]}=32'b000000_00001_00010_00111_00000_100110;//xor:rf[7]=rf[1]^rf[2]
//{memory[0],memory[1],memory[2],memory[3]}=32'b000000_00001_00010_00111_00000_100111;//nor:rf[7]=~(rf[1]|rf[2])

//{memory[0],memory[1],memory[2],memory[3]}=32'b000000_00110_00100_00111_00000_101010;//slt:rf[7]=rf[6]<rf[4]?1:0...sltu
//{memory[0],memory[1],memory[2],memory[3]}=32'b000000_00000_00010_00111_00001_000000;//sll:rf[7]=rf[2]<<1...sllv
//{memory[0],memory[1],memory[2],memory[3]}=32'b000000_00000_00010_00111_00001_000010;//srl:rf[7]=rf[2]>>1...srlv
//{memory[0],memory[1],memory[2],memory[3]}=32'b000000_00011_00000_00111_00000_000110;//srlv:rf[7]=rf[0]>>rf[3]
//{memory[0],memory[1],memory[2],memory[3]}=32'b000000_00011_00101_00111_00000_000011;//sra:rf[7]=rf[5]>>rf[3]...srav

//{memory[0],memory[1],memory[2],memory[3]}=32'b001000_00000_00111_0000000000000011;//addi:rf[7]=rf[0]+3...addiu
//{memory[0],memory[1],memory[2],memory[3]}=32'b001100_00000_00111_0000000000000100;//andi:rf[7]=rf[0]&4
```

//{memory[0],memory[1],memory[2],memory[3]}=32'b001101_00000_00111_0000000000000011;//ori:rf[7]=rf[0]|3

//{memory[0],memory[1],memory[2],memory[3]}=32'b001110_00000_00111_0000000000000100;//xori:rf[7]=rf[0]^4

//{memory[0],memory[1],memory[2],memory[3]}=32'b001111_00000_00111_0000000000000010;//lui:rf[7]=2<<16

//{memory[0],memory[1],memory[2],memory[3]}=32'b001010_00011_00111_0000000000000011;//slti:rf[7]=rf[3]<3?1:0...sltiu

//{memory[0],memory[1],memory[2],memory[3]}=32'b100011_00101_00111_0000000000000001;//lw:rf[7]=DM[rf[5]+1]

//{memory[0],memory[1],memory[2],memory[3]}=32'b101011_00011_00010_0000000000000100;//sw:DM[rf[3]+4]=rf[2]

//{memory[0],memory[1],memory[2],memory[3]}=32'b000010_00000000000000000000000011;//J:PC=1100;

//{memory[0],memory[1],memory[2],memory[3]}=32'b000000_00000_00000_00000_00000_001000;//Jr:PC=rf[0]

//{memory[0],memory[1],memory[2],memory[3]}=32'b000100_00010_00100_0000000000000100;//beq:PC=PC+4+4<<2;

//{memory[0],memory[1],memory[2],memory[3]}=32'b000101_00010_00011_0000000000000100;//bne:...

经测试，结果均与预期相符，所编写单周期 CPU 功能正常。

流水线 CPU 实验报告：

1. 实验目的及内容：。。。。

2. 实验过程：

数据通路图：

实现的指令集：

类型	指令	编码						操作
		31-26	25-21	20-16	15-11	10-6	5-0	
R-型	ADD	000000	rs	rt	rd	00000	100000	rd<-rs+rt
	ADDU	000000	rs	rt	rd	00000	100001	rd<-rs+rt
	SUB	000000	rs	rt	rd	00000	100010	rd<-rs-rt
	AND	000000	rs	rt	rd	00000	100100	rd<-rs&rt
	OR	000000	rs	rt	rd	00000	100101	rd<-rs|rt
	SLT	000000	rs	rt	rd	00000	101010	rd<-(rs<rt)?1:0
I-型	LW	100011	rs	rt	immediate			rt-<memory[rs+immediate]
	SW	101011	rs	rt	immediate			Memory[rs+immediate]<-rt
	BEQ	000100	rs	rt	immediate			If(rs==rt) PC<-PC+4+immediate<<2
J-型	J	000010	address					PC<-address<<2

控制单元 CU 的实现：

```
module CU(input [5:0]Inst,output reg RegDst,RegWr,MemtoReg,MemRd,MemWr,Branch,Jump,output reg [1:0]ALUOp,output reg ALUSrc);
always@(*)begin
case(Inst)
    6'b000000:begin//R

    {ALUOp,ALUSrc,Jump,Branch,MemRd,MemWr,RegDst,RegWr,MemtoReg}=10'b1000000110;
    end
    6'b100011:begin//LW

    {ALUOp,ALUSrc,Jump,Branch,MemRd,MemWr,RegDst,RegWr,MemtoReg}=10'b0010010011;
    end
    6'b101011:begin//SW

    {ALUOp,ALUSrc,Jump,Branch,MemRd,MemWr,RegDst,RegWr,MemtoReg}=10'b0010001x0x;
    end
    6'b000010:begin//J

    {ALUOp,ALUSrc,Jump,Branch,MemRd,MemWr,RegDst,RegWr,MemtoReg}=10'bxxx1000x0x;
    end
    6'b000100:begin//BEQ

    {ALUOp,ALUSrc,Jump,Branch,MemRd,MemWr,RegDst,RegWr,MemtoReg}=10'b0100100x0x;
    end
endcase
end
endmodule
```

ALUCU 的实现：

```
module ALUCU(input[5:0]func,input[1:0]ALUOp,output reg[2:0]ALUCtrl);
always@(*)begin
if(ALUOp==00)begin
ALUCtrl=3'b100;
end
if(ALUOp==01)begin
ALUCtrl=3'b110;
end
if(ALUOp[1]==1'b1)begin
case(func)
    6'b100000:begin
        ALUCtrl=3'b100;//ADD
```

```
        end
        6'b100001:begin
            ALUCtrl=3'b101;//ADDU
        end
        6'b100010:begin
            ALUCtrl=3'b110;//SUB
        end
        6'b100100:begin
            ALUCtrl=3'b000;//AND
        end
        6'b100101:begin
            ALUCtrl=3'b001;//OR
        end
        6'b101010:begin
            ALUCtrl=3'b011;//SLT;
        end
endcase
end
end
Endmodule
```

额外部件：

myOR：由于在流水线 CPU 中 PCSrc 的确定值要在第一条指令执行到第三阶段才能得出，而在这之前，第二第三条指令应当已经进入流水，因此若不对原数据通路图中的或门进行修改，PCSrc 将呈不定值，第二第三条指令的地址无法得到，因此，使用 myOR 元件，将 PCSrc 给予初始值并在结果不确定的情况下输出 0 以确保 PC 值能正确地递增，具体实现如下：

```
module myOR(input A,B,output reg O);
reg result;
initial begin
result=0;
O=result;
end
always@(*)begin
```

```
if(A|B==1||A|B==0)begin
result=A|B;
O=result;
end
end
endmodule
```