

# 浙江大学

## 本科实验报告

课程名称: 计算机体系结构

姓 名: 郑昱笙

学 院: 计算机科学与技术学院

系: 计算机科学与技术系

专 业: 地理信息科学

学 号: 3180102760

指导教师: 常瑞

2020 年 6 月 18 日

# 浙江大学实验报告

课程名称: 计算机体系结构 实验类型: 综合

实验项目名称: Project 3 流水线 CPU 设计

学生姓名: 郑昱笙 专业: 地理信息科学 学号: 3180102760

同组学生姓名: 胡凯文 指导老师: 常瑞

实验地点: 家里 实验日期: 20202 年 6 月 18 日

## 一、实验目的和要求

1. Design the CPU Controller, Datapath, bring together the basic units into Pipeline CPU;
2. Verify the PCPU with program and observe the execution of program.

## 二、实验内容和原理

- 1、本实验实现了 31 条指令的 MIPS 五级流水线 CPU:

R 型指令:

ADD/ADDU、SUB/SUBU、SLT/SLTU、SLLV、SRLV、SRAV、AND、OR、XOR、NOR

带 shamt 的 R 型指令:

SRL、SLL、SRA

I 型指令:

ADDi、ADDIU、SLTi、SLTiU、ANDi、ORi、XORi

SW、LW、LUI

BEQ、BNE

J 类型指令:

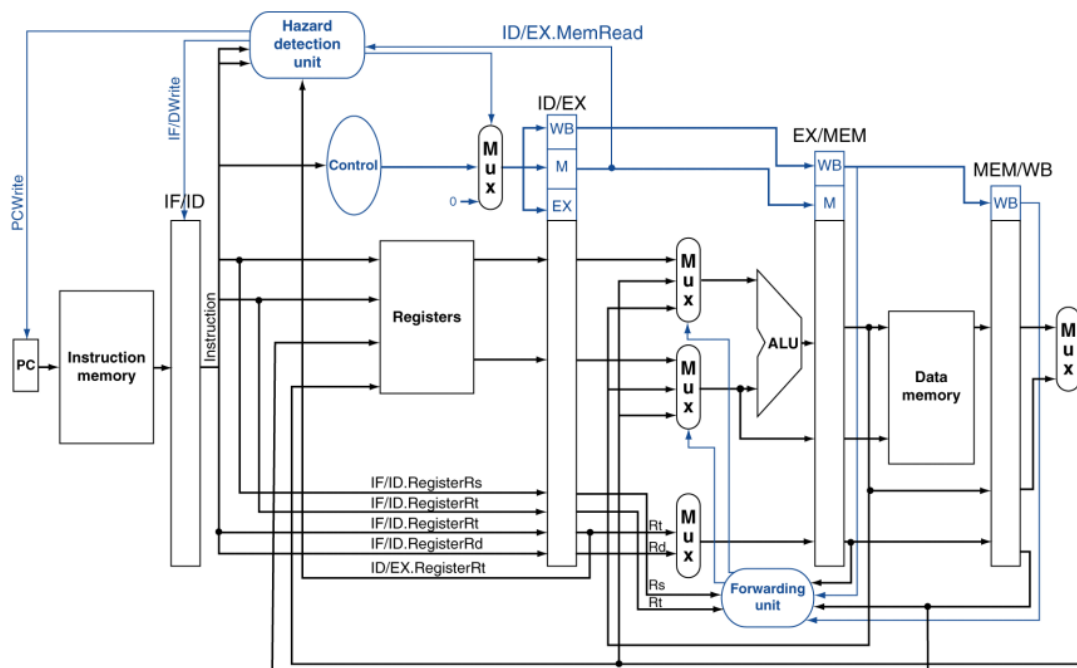
J、JAL、JR

以 32 位为最小操作、编址单位;

实现了流水线数据的前向传输;

不对溢出进行检测, 对控制冒险、部分数据冒险等问题采用插入气泡方式解决;

- 2、本实验所参考设计的五级流水线 CPU 数据通路图:



### 3、 本实验所参考设计的 CPU 控制器状态机图表：

IF	$IF/ID.IR \leftarrow Mem[PC];$ $IF/ID.NPC, PC \leftarrow (if \ ( \langle IF/ID[op] == branch \rangle \ \& \ (Regs[IF/ID.IR[rs]] = 0)) \{ IF/ID.NPC + (IF/ID.IR_{16..31} \ll 2) \} \text{ else } \{ PC+4 \});$		
ID	$ID/EX.A \leftarrow Regs[IF/ID.IR[rs]]; \quad ID/EX.B \leftarrow Regs[IF/ID.IR[rt]]; \quad ID/EX.IR \leftarrow IF/ID.IR;$ $ID/EX.Imm \leftarrow (IF/ID.IR_{16} \ll 16) \# IF/ID.IR_{16..31};$		
	ALU 指令	load/store 指令	分支指令
EX	$EX/MEM.IR \leftarrow ID/EX.IR;$ $EX/MEM.ALU_o \leftarrow ID/EX.A \text{ func } ID/EX.B$ 或 $EX/MEM.ALU_o \leftarrow ID/EX.A \text{ op } ID/EX.Imm;$	$EX/MEM.IR \leftarrow ID/EX.IR;$ $EX/MEM.ALU_o \leftarrow ID/EX.A + ID/EX.Imm;$ $EX/MEM.B \leftarrow ID/EX.B;$	$EX/MEM.IR \leftarrow ID/EX.IR;$ $EX/MEM.ALU_o \leftarrow ID/EX.NPC + ID/EX.Imm \ll 2;$ $EX/MEM.cond \leftarrow (ID/EX.A == 0);$
MEM	$MEM/WB.IR \leftarrow EX/MEM.IR;$ $MEM/WB.ALU_o \leftarrow EX/MEM.ALU_o;$	$MEM/WB.IR \leftarrow EX/MEM.IR;$ $MEM/WB.LMD \leftarrow Mem[EX/MEM.ALU_o];$ 或 $Mem[EX/MEM.ALU_o] \leftarrow EX/MEM.B;$	
WB	$Regs[MEM/WB.IR[rd]] \leftarrow MEM/WB.ALU_o;$ 或 $Regs[MEM/WB.IR[rt]] \leftarrow MEM/WB.ALU_o;$	$Regs[MEM/WB.IR[rt]] \leftarrow MEM/WB.LMD;$	

对应的指令控制信号图如下：

opcode	func		BranchFlag	MemtoReg	mem_w	cond	ALU_op	RegWrite	rd_addr_valid	EXE_forward	SignExtend	RegDst	ALUsrc_B
<b>R-Type</b>													
0 0x24		AND	00	01	0	0	0000	1	11	1	X	01	0
0 0x20/0x21		ADD/ADDU	00	01	0	0	0010	1	11	1	X	01	0
0 0x25		OR	00	01	0	0	0001	1	11	1	X	01	0
0 0x22/0x23		SUB/SUBU	00	01	0	0	0110	1	11	1	X	01	0
0 0x26		XOR	00	01	0	0	0011	1	11	1	X	01	0
0 0x27		NOR	00	01	0	0	0100	1	11	1	X	01	0
0 0x2a/0x2b		SLT/SLTU	00	01	0	0	0111/1111	1	11	1	X	01	0
0 0x04/0x06		SLLV/SRLV	00	01	0	0	1010/1001	1	11	1	X	01	0
0 0x07		SRAV	00	01	0	0	1011	1	11	1	X	01	0
<b>R-Type with shamt</b>													
0 0x02/0x00		SRL/SLL	00	01	0	0	0101/1101	1	01	1	X	01	0
0 0x03		SRA	00	01	0	0	1000	1	01	1	X	01	0
<b>I-Type</b>													
0x23/0x2b	0	LW/SW	00	00/01	0/1	0	0010	1/0	10/11	0	1	XX	1
0x04/0x05	0	BEQ/BNE	01	XX	0	0/1	XXXX	0	11	1	1	XX	X
0x0f	0	LUI	00	10	0	0	XXXX	1	00	1	X	00	X
0x08/0x09	0	ADDI/ADDIU	00	01	0	0	0010	1	10	1	1	00	1
0x0d	0	ORI	00	01	0	0	0001	1	10	1	1	00	1
0x0e	0	XORI	00	01	0	0	0011	1	10	1	1	00	1
0x0a/0x0b	0	SLTI/SLTIU	00	01	0	0	0111/1111	1	10	1	1/0	00	1
0x0c	0	ANDI	00	01	0	0	0000	1	10	1	1	00	1
<b>J-Type</b>													
0x02	0	J	10	XX	0	0	XXXX	0	00	1	X	XX	X
0x03	0	JAL	10	11	0	0	XXXX	1	00	1	X	10	X
0 0x08		JR	11	XX	0	0	XXXX	0	10	1	X	XX	X

#### 4、 输入输出：

所用的实验板为 XC6SLX9，参数如下：

### Spartan-6 FPGA Feature Summary

Table 1: Spartan-6 FPGA Feature Summary by Device

Device	Logic Cells <sup>(1)</sup>	Configurable Logic Blocks (CLBs)			DSP48A1 Slices <sup>(3)</sup>	Block RAM Blocks		CMTs <sup>(5)</sup>	Memory Controller Blocks (Max) <sup>(6)</sup>	Endpoint Blocks for PCI Express	Maximum GTP Transceivers	Total I/O Banks	Max User I/O
		Slices <sup>(2)</sup>	Flip-Flops	Max Distributed RAM (Kb)		18 Kb <sup>(4)</sup>	Max (Kb)						
XC6SLX4	3,840	600	4,800	75	8	12	216	2	0	0	0	4	132
XC6SLX9	9,152	1,430	11,440	90	16	32	576	2	2	0	0	4	200

使用的板级外设为开关、视为数码管、流水灯等；具体实验效果解说可参考视频：

通过数码管和开关的切换来显示 PC、指令、内存读写的十六进制数据；

通过流水灯来显示流水线当前状态，包括时钟、复位信号、阻塞、分支、溢出等；

## 三、 实验过程和数据记录

### 1、 组件实现：

#### ALU 实现：

ALU 为纯逻辑实现，支持加减、逻辑、移位等运算，并有溢出检测标志位；

```

module ALU(
    input [31:0] A,
    input [31:0] B,
    input [4:0] shamt,
    input [3:0] ALU_operation,
    output reg [31:0] res,
    output overflow
);

wire [31:0] Bo, And, Or, Srl, Xor, Nor, Sll, Sra, Srlv, Srav, Sllv;
wire [32:0] Sum;

```

```

wire sub = ALU_operation[2];

assign Bo = B ^ {32{sub}}; //if sub is True, Bo = 2^32 - 1 - B else B
assign Sum = A + Bo + {(A[31]^B[31]), 31'b0, ALU_operation[2]};
//overflow occurs when:
//sub A[31] B[31] S
// 1   1   0   0
// 1   0   1   1
// 0   1   1   0
// 0   0   0   1
assign overflow = (Sum[31] & ~(A | (sub ^ B[31]))) | (~Sum[31] & (A & (sub ^ B[31])));

assign And = A & B;
assign Or = A | B;
assign Srlv = B >> A[4:0];
assign Sllv = B << A[4:0];
assign Srav = B >>> A[4:0];
assign Srl = B >> shamt; //extend to SRLV
assign Sll = B << shamt; //SLL, SLLV
assign Sra = B >>> shamt;
assign Xor = A ^ B;
assign Nor = ~A & ~B;

always@(*)begin
    case(ALU_operation)
        4'b0000: res <= And;
        4'b0001: res <= Or;
        4'b0010: res <= Sum[31:0];
        4'b0011: res <= Xor;
        4'b0100: res <= Nor;
        4'b0101: res <= Srl;
        4'b0110: res <= Sum[31:0];
        4'b0111: res <= {31'b0, Sum[32]}; //slt
        4'b1000: res <= Sra;
        4'b1001: res <= Srlv;
        4'b1010: res <= Sllv;
        4'b1011: res <= Srav;
        4'b1101: res <= Sll;
        4'b1111: res <= (A < B);
    endcase
end
endmodule

```

寄存器组实现:

采用寄存器数组的方式实现寄存器：

寄存器中对时钟进行了一次取反，要求在时钟正边沿写入，在时钟负边沿读出数据：

```
module regs(
    input clk,
    input rst,
    input L_S,
    input [4:0] R_addr_A,
    input [4:0] R_addr_B,
    input [4:0] Wt_addr,
    input [31:0] Wt_data,
    output reg [31:0] rdata_A,
    output reg [31:0] rdata_B
);

reg [31:0] register [1:31];    // r1 - r31
integer i;

wire clk1,rst1;
assign clk1 = ~clk;

always@(posedge clk1 or posedge rst1)begin
    if(rst) begin rdata_A <= 32'b0; rdata_B <= 32'b0; end
    else begin
        rdata_A <= (R_addr_A == 0) ? 0 : register[R_addr_A];
        rdata_B <= (R_addr_B == 0) ? 0 : register[R_addr_B];
    end
end

//always write at the posedge, read at negedge
always @(*)begin
    if (rst)
        for (i=1; i<32; i=i+1) register[i] <= 0;        // reset
    else if ((Wt_addr != 0) && (L_S == 1))
        register[Wt_addr] <= Wt_data;                    // write
    end
endmodule
```

**控制器实现 (Controller.v):**

控制器通过指令的 op 字段和 func 字段来决定输出信号；

具体的输出信号赋值如下：

```
assign {BranchFlag_ID, MemtoReg_ID, mem_w_ID, cond_ID, ALU_op_ID, RegWrite_ID, rd_ad
dr_valid, EXE_forward_ID, SignExtend, RegDst_ID, ALUsrc_B_ID} = ctrl_signal;
```

输出信号表：

```

// Control Unit
always @* begin
    case(opcode)
        6'h00: begin
            case(func)
                6'h24: begin ctrl_signal <= 18'b00_01_0_0_0000_1_11_1_0_01_0; end //0x24 and
                6'h20: begin ctrl_signal <= 18'b00_01_0_0_0010_1_11_1_0_01_0; end //0x20 add
                6'h21: begin ctrl_signal <= 18'b00_01_0_0_0010_1_11_1_0_01_0; end //0x21 addu
                6'h25: begin ctrl_signal <= 18'b00_01_0_0_0001_1_11_1_0_01_0; end //0x25 or
                6'h22: begin ctrl_signal <= 18'b00_01_0_0_0110_1_11_1_0_01_0; end //0x22 sub

                6'h23: begin ctrl_signal <= 18'b00_01_0_0_0110_1_11_1_0_01_0; end //0x23 subu
                6'h26: begin ctrl_signal <= 18'b00_01_0_0_0011_1_11_1_0_01_0; end //0x26 xor
                6'h27: begin ctrl_signal <= 18'b00_01_0_0_0100_1_11_1_0_01_0; end //0x27 nor
                6'h2a: begin ctrl_signal <= 18'b00_01_0_0_0111_1_11_1_0_01_0; end //0x2a slt
                6'h2b: begin ctrl_signal <= 18'b00_01_0_0_1111_1_11_1_0_01_0; end //0x2b sltu

                6'h04: begin ctrl_signal <= 18'b00_01_0_0_1010_1_11_1_0_01_0; end //0x04 sllv
                6'h06: begin ctrl_signal <= 18'b00_01_0_0_1001_1_11_1_0_01_0; end //0x06 srlv
                6'h07: begin ctrl_signal <= 18'b00_01_0_0_1011_1_11_1_0_01_0; end //0x07 srav
                6'h00: begin
                    if(inst_in != 32'h0)
                        ctrl_signal <= 18'b00_01_0_0_1101_1_01_1_0_01_0; //0x00 sll
                    else
                        ctrl_signal <= 18'b00_00_0_0_0000_0_00_0_0_00_0; //0x00 nop
                    end
                6'h02: begin ctrl_signal <= 18'b00_01_0_0_0101_1_01_1_0_01_0; end //0x02 srl

                6'h03: begin ctrl_signal <= 18'b00_01_0_0_1000_1_01_1_0_01_0; end //0x03 sra
                6'h08: begin ctrl_signal <= 18'b11_01_0_0_0000_0_10_1_0_01_0; end //0x08 jr
            endcase
        end

        6'h23: begin ctrl_signal <= 18'b00_00_0_0_0010_1_10_0_1_00_1; end //0x23 lw
        6'h2b: begin ctrl_signal <= 18'b00_01_1_0_0010_0_11_0_1_01_1; end //0x2b sw
        6'h04: begin ctrl_signal <= 18'b01_01_0_0_0000_0_11_1_1_01_1; end //0x04 beq
        6'h05: begin ctrl_signal <= 18'b01_01_0_1_0000_0_11_1_1_01_1; end //0x05 bne
        6'h0f: begin ctrl_signal <= 18'b00_10_0_0_0000_1_00_1_1_00_1; end //0x0f lui

        6'h08: begin ctrl_signal <= 18'b00_01_0_0_0010_1_10_1_1_00_1; end //0x08 addi
        6'h09: begin ctrl_signal <= 18'b00_01_0_0_0010_1_10_1_1_00_1; end //0x09 addiu
        6'h0d: begin ctrl_signal <= 18'b00_01_0_0_0001_1_10_1_1_00_1; end //0x0d ori
        6'h0e: begin ctrl_signal <= 18'b00_01_0_0_0011_1_10_1_1_00_1; end //0x0e xori
        6'h0a: begin ctrl_signal <= 18'b00_01_0_0_0111_1_10_1_1_00_1; end //0x0a slti
    end
end

```

```

6'h0b: begin ctrl_signal <= 18'b00_01_0_0_1111_1_10_1_0_00_1; end //0x0b sltiu
6'h0c: begin ctrl_signal <= 18'b00_01_0_0_0000_1_10_1_1_00_1; end //0x0c andi
6'h02: begin ctrl_signal <= 18'b10_01_0_0_0011_0_00_1_0_00_0; end //0x02 j
6'h03: begin ctrl_signal <= 18'b10_11_0_0_0011_1_00_1_1_10_1; end //0x03 jal
endcase
end

```

### 存储器实现:

存储器采用 Block Memory Generator 生成 ip 核方式实现，需要生成 True Dual Port RAM，并设置两个端口的时钟相同，读写宽度为 32 位，深度为 1024;

```

module mem(
    clka,
    wea,
    addra,
    dina,
    douta,
    clk_b,
    web,
    addrb,
    dinb,
    doutb
);

```

## 2、 流水线 CPU 数据通路实现:

### IF 段:

IF 段完成取址的任务，并检查 ID 段是否是跳转指令，如果是跳转指令，传入 nop 暂停流水线；  
（如果是在 EXE 段决定的跳转过程，此处由于已经事先是分支指令所以不用再进行判断）

```

wire [1:0] BranchFlag_ID;
reg [1:0] BranchFlag_EXE;
reg [31:0] PC_in;
REG32 regPC (
    .clk(clk),
    .rst(reset),
    .CE(~stall),
    .D(PC_in),
    .Q(PC_out)
);
assign stall = data_stall | |(BranchFlag_ID);

```

### 暂停判断:

```

inst_ID <= inst_in & {32{~|(BranchFlag_EXE | BranchFlag_ID)}}; //如果 ID/EXE
PC_ID <= PC_out + 32'h4;

```

### ID 段



ID 段将取出的指令送入控制器，并完成译码和访问寄存器操作：（参见前文）

译码：

```
assign {BranchFlag_ID, MemtoReg_ID, mem_w_ID, cond_ID, ALU_op_ID, RegWrite_ID, rd_addr_valid, EXE_forward_ID, SignExtend, RegDst_ID, ALUsrc_B_ID} = ctrl_signal;
```

立即数扩展：

```
wire [31:0] Imm32_ID;
assign Imm32_ID = SignExtend ? {{16{inst_ID[15]}}, inst_ID[15:0]} : {{16{1'b0}}, inst_ID[15:0]};
```

访问寄存器组：

```
wire [31:0] Reg_A, Reg_B;
regs Regs (
    .clk(clk),
    .rst(reset),
    .L_S(RegWrite_WB),
    .R_addr_A(rs),
    .R_addr_B(rt),
    .Wt_addr(wt_addr_WB),
    .Wt_data(wt_data_WB),
    .rdata_A(Reg_A),
    .rdata_B(Reg_B)
);
```

同时，在 ID 段也需要进行冲突检测和数据向前传输的操作：

当读写地址有效时，若读地址与 EXE/MEM 段指令的写地址一致会发生 data hazard，如果可以回传，即 ID 段读写地址有效，且与 EXE 段写地址一致；或读写地址有效，且与 MEM 段写地址一致，就采用回传部件进行数据的前向传输，如果发生数据冲突但不能进行回传，就暂停流水线：

数据冲突的判断：

```
assign data_hazard = {RegWrite_EXE & ((rd_addr_valid[1] & (wt_addr_EXE == rs)) | (rd_addr_valid[0] & (wt_addr_EXE == rt))),
                    RegWrite_MEM & ((rd_addr_valid[1] & (wt_addr_MEM == rs)) | (rd_addr_valid[0] & (wt_addr_MEM == rt)))};
assign data_stall = data_hazard[1] & ~EXE_forward;
```

下面对 data\_hazard 的四种情况进行判断，分别对应于两个阶段都需要回传，需要回传某一个阶段和不需要进行回传的部分

如果可以进行回传时的条件判断：

```
always@* begin
    case(data_hazard)
        2'b11: begin
            //当 EXE_forward 为 0 时，下面的值未定义
            if(wt_addr_EXE == wt_addr_MEM) begin
                //EXE MEM 的 forwarding 一致，回传 EXE 段数据，另一个数据源一定是 regs
                if (rs == wt_addr_EXE) begin
                    Reg_A_ID <= data_forward_EXE;
                    Reg_B_ID <= Reg_B;
                end
            end
        end
    endcase
end
```

```

        end else begin
            Reg_A_ID <= Reg_A;
            Reg_B_ID <= data_forward_EXE;
        end
    end else begin
        //不一致, ID 段指令的两个数据源都需要 forwarding
        if (rs == wt_addr_EXE) begin
            Reg_A_ID <= data_forward_EXE;
            Reg_B_ID <= data_forward_MEM;
        end else begin
            Reg_A_ID <= data_forward_MEM;
            Reg_B_ID <= data_forward_EXE;
        end
    end

end

end

2'b10: begin
    //当 EXE_forward 为 0 时, 下面的值未定义
    if (rs == wt_addr_EXE) begin
        Reg_A_ID <= data_forward_EXE;
        Reg_B_ID <= Reg_B;
    end else begin
        Reg_A_ID <= Reg_A;
        Reg_B_ID <= data_forward_EXE;
    end
end

2'b01: begin
    if (rs == wt_addr_MEM) begin
        Reg_A_ID <= data_forward_MEM;
        Reg_B_ID <= Reg_B;
    end else begin
        Reg_A_ID <= Reg_A;
        Reg_B_ID <= data_forward_MEM;
    end
end

2'b00: begin
    Reg_A_ID <= Reg_A;
    Reg_B_ID <= Reg_B;
end

endcase
end

```

不能进行回传时, 需要传入空指令, 暂停流水线:

此处只要关注数据冲突即可, 控制冲突加入的空指令已在 IF 段生成, 数据冲突即 EXE 段还未生成回传数据,

但已经产生了冲突的情况。

```
RegWrite_EXE <= RegWrite_ID & ~data_stall;  
mem_w_EXE <= mem_w_ID & ~data_stall;
```

只要保证不会改变 CPU 状态，因此只需要取消写使能即可。

EXE 段：

EXE 段将数据送入 ALU，并完成分支跳转指令地址的计算。

ALU：

```
//ALU  
wire [31:0] ALUout_EXE;  
ALU alu (  
    .A(Reg_A_EXE),  
    .B(ALUsrc_B_EXE?Imm32_EXE:Reg_B_EXE), //ALUsrc_B = 0, use reg_B(R-Type, e.g.); A  
    .LUsrc_B = 1, use Imm  
    .shamt(shamt_EXE),  
    .ALU_operation(ALU_op_EXE),  
    .res(ALUout_EXE)  
);
```

分支判断：

```
assign branch_PC_EXE = PC_EXE + (b_flag?{Imm32_EXE[29:0], 2'b0}:32'h0); //若满足条件  
跳转，否则 PC+4  
always@(*) begin  
    case(BranchFlag_EXE)  
        2'b00: begin PC_in <= PC_out + 32'h4; end //无  
        分支，无延时槽  
        2'b01: begin PC_in <= branch_PC_EXE; end //分  
        支指令  
        2'b10: begin PC_in <= {PC_EXE[31:28], inst_EXE[25:0], 2'b0}; end //j  
        2'b11: begin PC_in <= Reg_A_EXE; end //jr  
    endcase  
end
```

在 EXE 阶段结束后，需要进行数据的前向传递判断：

```
always@(*)begin  
    case(MemtoReg_EXE)  
        2'b00: data_forward_EXE <= 32'h0; //undefined  
        2'b01: data_forward_EXE <= ALUout_EXE; //ALU 型指令，也在 EXE 段  
        forwarding  
        2'b10: data_forward_EXE <= {Imm32_EXE[15:0], 16'b0}; //Lui，在 EXE 段能  
        forwarding  
        2'b11: data_forward_EXE <= PC_EXE; //链接，在 EXE 段  
        forwarding  
    endcase  
end
```

MEM 段：

MEM 段进行访存,

```
assign mem_w = mem_w_MEM;
assign data_out = data_out_MEM;
assign addr_out = ALUout_MEM;
```

并且同样需要进行数据前向传输的判断:

```
always@(negedge clk)begin
    case(MemtoReg_MEM)
        2'b00: data_forward_MEM <= data_in;
        2'b01: data_forward_MEM <= ALUout_MEM;
        2'b10: data_forward_MEM <= {Imm32_MEM[15:0], 16'b0};
        2'b11: data_forward_MEM <= PC_MEM;
    endcase
end
```

WB 段:

WB 段确定写入寄存器组的数据。

```
always@(*)begin
    case(MemtoReg_WB)
        2'b00: wt_data_WB <= data_in_WB;           //LW
        2'b01: wt_data_WB <= ALUout_WB;           //ALU 型指令
        2'b10: wt_data_WB <= {Imm32_WB[15:0], 16'b0}; //Lui
        2'b11: wt_data_WB <= PC_WB;               //链接
    endcase
end
```

### 3、 CPU 测试:

verilog 测试驱动代码 (test\_CPU.v):

```
reg clk, rst;
wire en;
wire [31:0] data_in, inst_in, wt_mem_data;
wire [31:0] data_addr, PC_ID;

Pipeline_CPU pCPU(
    .clk(clk),
    .reset(rst),
    .data_in(data_in),
    .inst_in(inst_in),
    .PC_out(PC_ID),
    .mem_w(en),
    .data_out(wt_mem_data),
```

```

        .addr_out(data_addr)
    );

mem d_RAM(
    .addra(data_addr[12:2]),
    .dina(wt_mem_data),
    .addrb(PC_ID[12:2]),
    .wea(en),
    .clka(clk),
    .clkb(clk),
    .douta(data_in),
    .doutb(inst_in)
);

initial begin clk = 0; rst = 1; end
initial forever #50 clk = ~clk;
initial #500 rst = 0;

```

## 测试:

我们对处理器进行了多种测试，并尽量涵盖了各种指令序列：

以其中一个测试程序为例（包含大多数运算、分支、访存、跳转指令）：

汇编语言测试代码：

```

# test asm:
    # test add/addi
    addi $t1, $zero, 0x5A5A
    addi $t2, $zero, 0x05A5
    add $t0, $t1, $t2
    beq $zero, $t0, label      # beq, not jump
    sw $t1, 756($zero)         # save 0x5a5a
    # test lw/sw between branch
    label: sw $zero, 756($zero) # save 0x0
    lw $t0, 0($zero)           # read 0x20095a5a
    bne $t0, $zero, label2     # bne, jump
label4: j label3
label2: add $t3, $zero, $zero   # t3 = 0
    addi $t4, $zero, 1         # t4 = 1
    sw $t0, 636($zero)         # save 0x20095a5a
label3: addi $t3, $t3, 1       # t3 = 1
    beq $t3, $t4, label4      # beq, jump
    # test other insts
    addi $t3, $t3, -2
    lui $t3, 0x8000

```

```

    addi $t3, $t3, -1
    ori $t4, $zero, 31
    sw $t3, 756($zero)      # save 0x7fff_ffff
    srl $t3, $t3, 31
    sw $t3, 756($zero)      # save 0x0
    add $a1, $zero, $zero
    bne $t3, $zero, label5  # bne, not jump
    addi $a1, $zero, 3

    # enter function
label5: addi $a1, $a1, 5
        jal sum
        sw $v0, 512($zero)    #save, 512
exit:   j exit

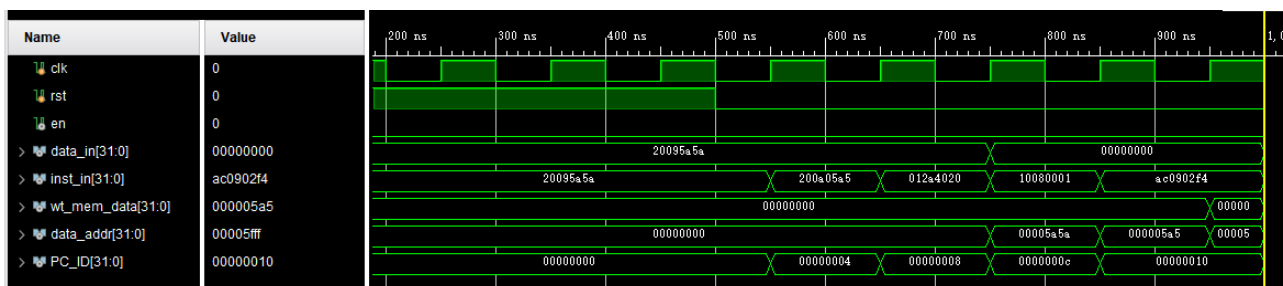
    # test function
sum:    add $a2, $zero, $zero
loop:   add $a2, $a1, $a2
        addi $a1, $a1, -1
        bne $a1, $zero, loop
        add $v0, $zero, $a2
        jr $ra

```

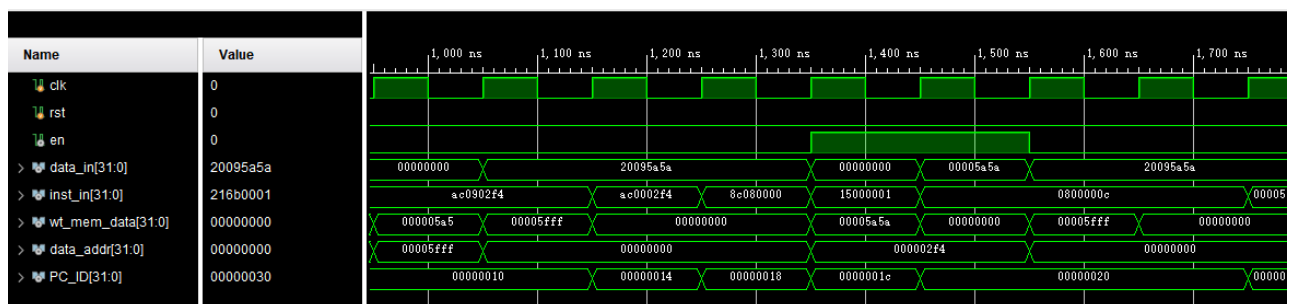
通过 MARS 4.5 将上述指令序列编译为十六进制机器码（output.coe）并作为存储器的 ip 核的初始化文件：

## 测试结果

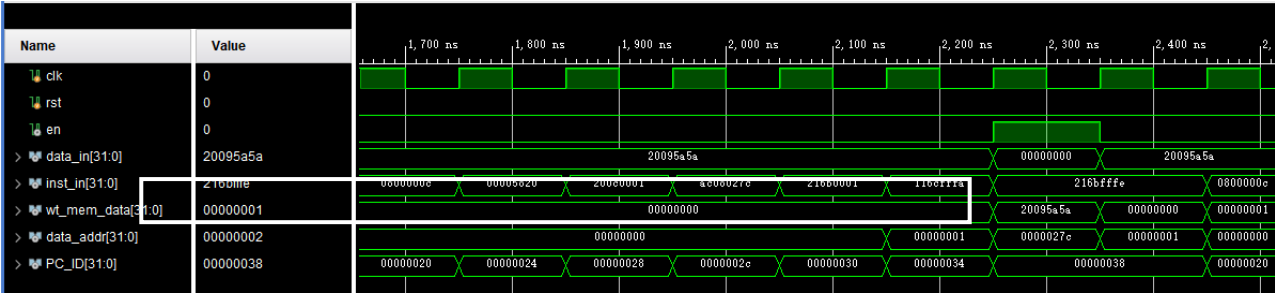
测试代码前 5 条指令，包含立即数/寄存器相加以及尚未完成的分支：



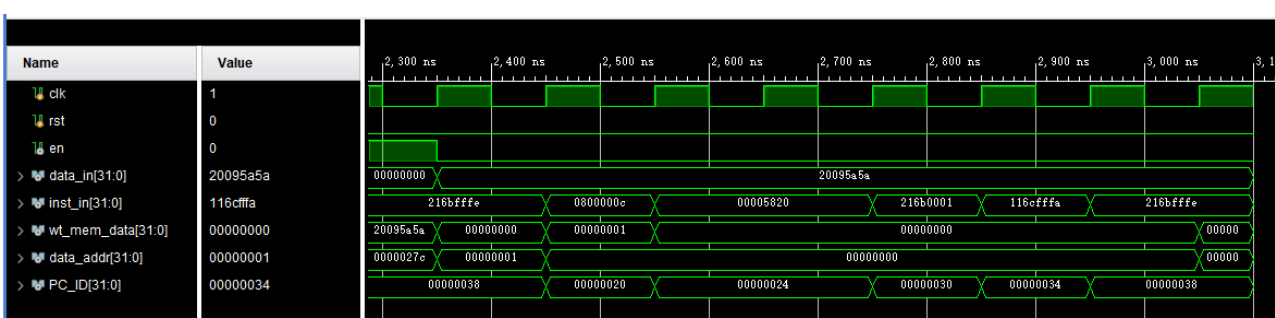
测试代码第 6-8 条指令的执行，包含分支过程的完成和开始：



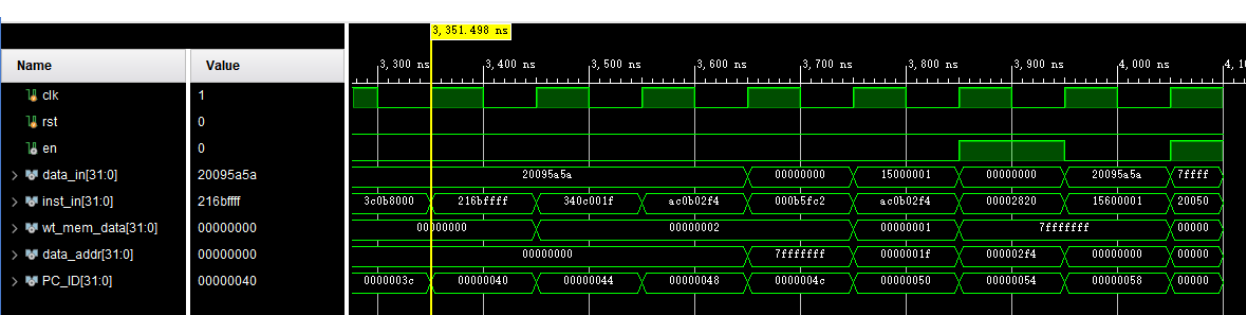
第九条跳转指令，再到 15 条分支指令回到第九条 label4，包含跳转指令和分支指令的序列：



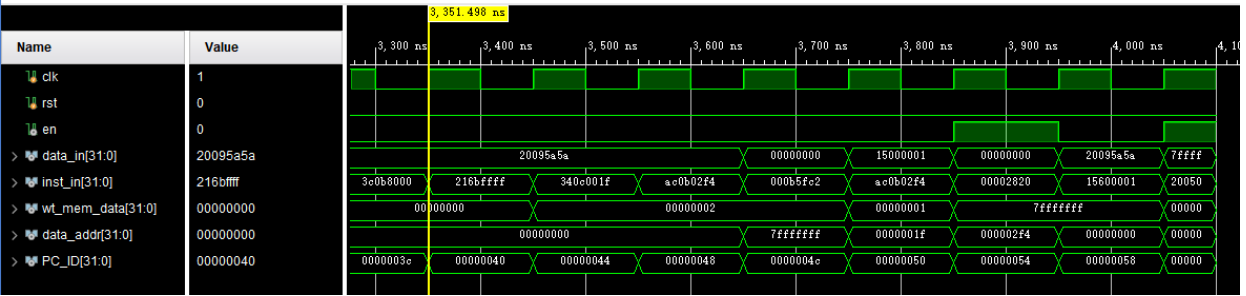
执行第九条指令后跳转至 0x30，这次由于改变了寄存器状态，beq 不分支：



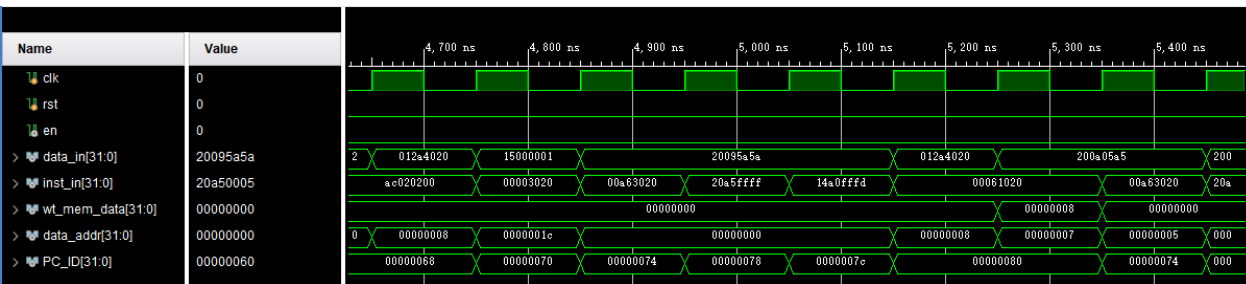
执行到第 25 条指令，保存两个数据：



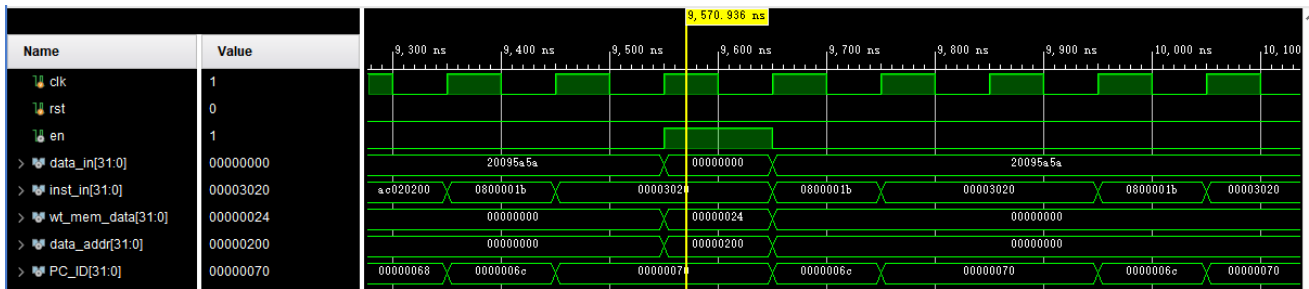
第 27 条指令不分支，随后第 31 条 jal 跳转至一个简单的累加小程序：



在程序过程中进行循环计算：



写入结果 0x24(36=1+2...+8)，之后进入死循环：



对照 MARS 模拟器执行结果，可以发现代码功能正确。

## 四、 开发板烧录与执行结果

在工程文件中实例化 CPU 以及相应内存代码：

开发板外部时钟是 25MHz，可通过 PLL 进行分频；在测试时，为了便于观察，可另将时钟信号转换为 1HZ；

```
Pipeline_CPU pCPU(
    .clk(clk),
    .reset(rst),
    .data_in(data_in),
    .inst_in(inst_in),
    .PC_out(PC_ID),
    .mem_w(en),
    .data_out(wt_mem_data),
    .addr_out(data_addr),
    .stall(stall),
    .data_stall(dstall)
);

mem d_RAM(
    .addra(data_addr[11:2]),
    .dina(wt_mem_data),
    .dinb(32'b0),
    .addrb(PC_ID[11:2]),
    .wea(en),
    .web(1'b0),
    .clka(clk),
    .clkb(clk),
    .douta(data_in),
    .doutb(inst_in)
);
```

由于该开发板只有四位数码管可供显示，因此可以采用通过开关切换的方式切换显示数值：

分别可以显示当前指令地址、数据地址、指令值、数据值；

```
assign display_num =
```



```

    (switch[0] ?
      (switch[2]?
        ( switch[1]? data_in[31:16] : data_in[15:0] ):
        ( switch[1]? inst_in[31:16] : inst_in[15:0] )
      ):
      (switch[2]?
        ( switch[1]? data_addr[31:16] : data_addr[15:0] ):
        ( switch[1]? PC_ID[31:16] : PC_ID[15:0] )
      )
    );

```

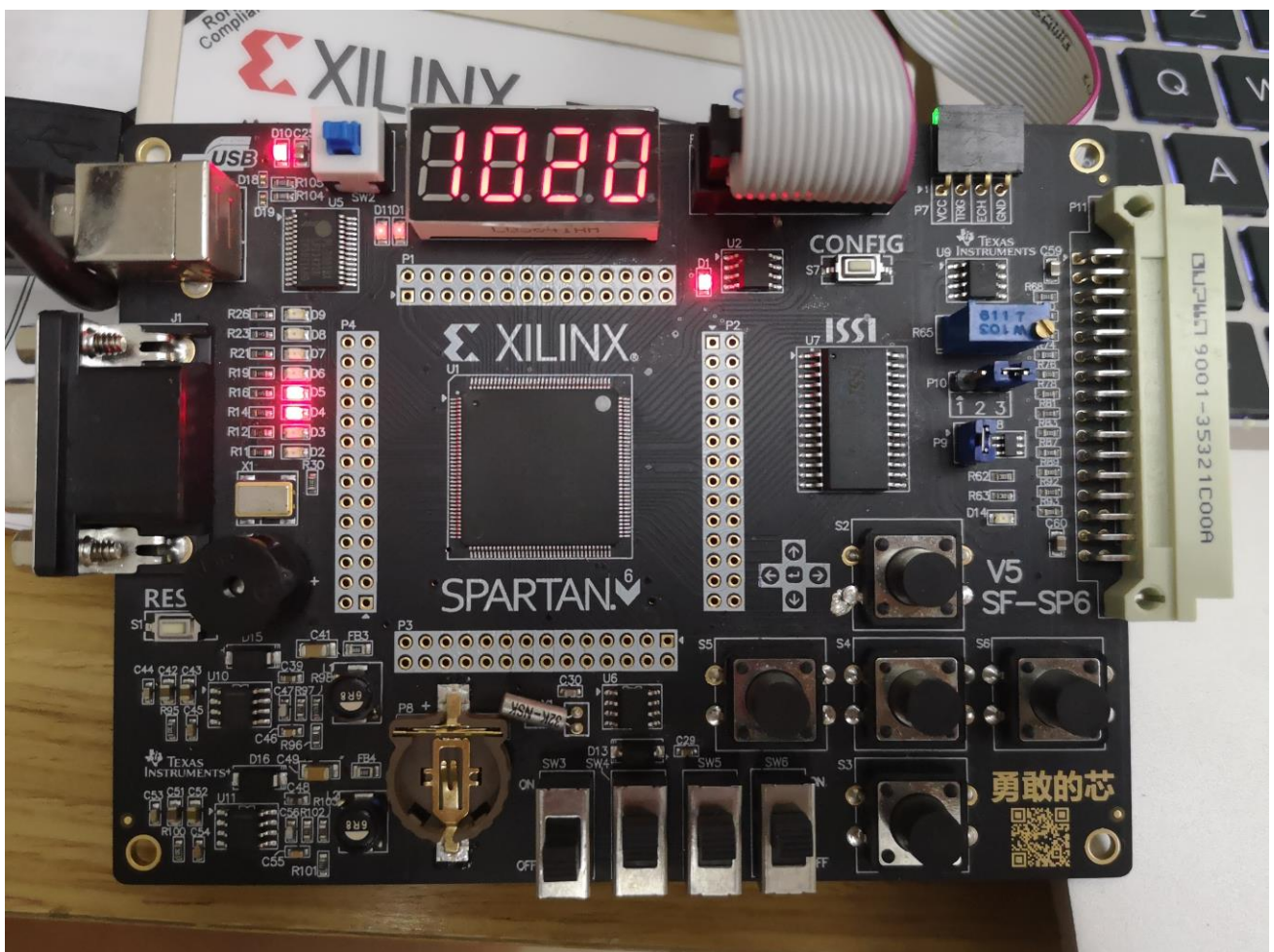
通过流水灯信号可表示时钟信号、复位信号、内存写信号、指令 stall、数据 stall 等:

```

assign led[0] = ~clk_1s;
assign led[1] = ~rst;
assign led[2] = ~en;
assign led[3] = ~stall;
assign led[4] = ~dstall;

```

将前述的汇编代码经综合、布线后下载至开发板，运行结果与仿真相同



## 五、 讨论与心得

实验分工：

胡凯文：流水线 CPU 的设计实现与在 vivado 上的仿真测试；

郑昱笙：流水线 CPU 迁移至 ISE14.6 的仿真测试、部分错误修复与烧录至开发板调试，实验报告的撰写；

我们实现的流水线 CPU 还是相对较为简单的，大部分参照课程讲授的实现，没有分支判断、指令调度等功能，指令条数也相对较少，后续可以继续改进。总体来说，还是对现代的流水线 CPU 有了一个更直观和深入的认识，还是收获不少的。

另外，由于使用的 sp6 fpga 开发板信号相对老旧，vivado 不支持，但流水线 CPU 的工程文件使用其进行开发，导致了在项目从 vivado 迁移至 ise14.6 的过程中也出现了不少意料之外的情况，比如仿真和实现结果不一致等，需要仔细调试和对 verilog 代码进行一些额外处理（这部分也花了不少时间）。