同濟大學
TONGJI UNIVERSITY

计算机系统结构课程实验
总结报告

实验题目：简单的流水线 CPU 设计与性能分析

学号：2152118
姓名：史君宝
指导教师：秦国峰

日期：2023 年 11 月 26 日

一、 实验环境部署与硬件配置说明

实验用到的主要环境为下面：

Vivado 软件和 Verilog 语言

FPGA 开发板

Mars 汇编器

二、 实验的总体结构

1、 静态流水线的总体结构

静态流水线的总体结构通常由多个阶段组成，分别负责处理指令执行的不同部分。我们构建的流水线 CPU 的主要功能如下面所示。

首先是指令获取 IF 结构，它能够从指令存储器中获取下一条指令，并将指令传递给下一个阶段。

然后是指令译码 ID 结构，它会对指令进行译码，从而确定指令的类型和操作数。并将译码后的指令和操作数传递给下一个阶段。

之后是指令执行的 EXE 结构，它执行具体的指令。

之后是访存的 MEM 结构，如果指令需要访问内存，在这个阶段可以进行内存读取或写入操作。

最后是写回阶段 WB 结构，它会将执行阶段的结果写回寄存器文件。

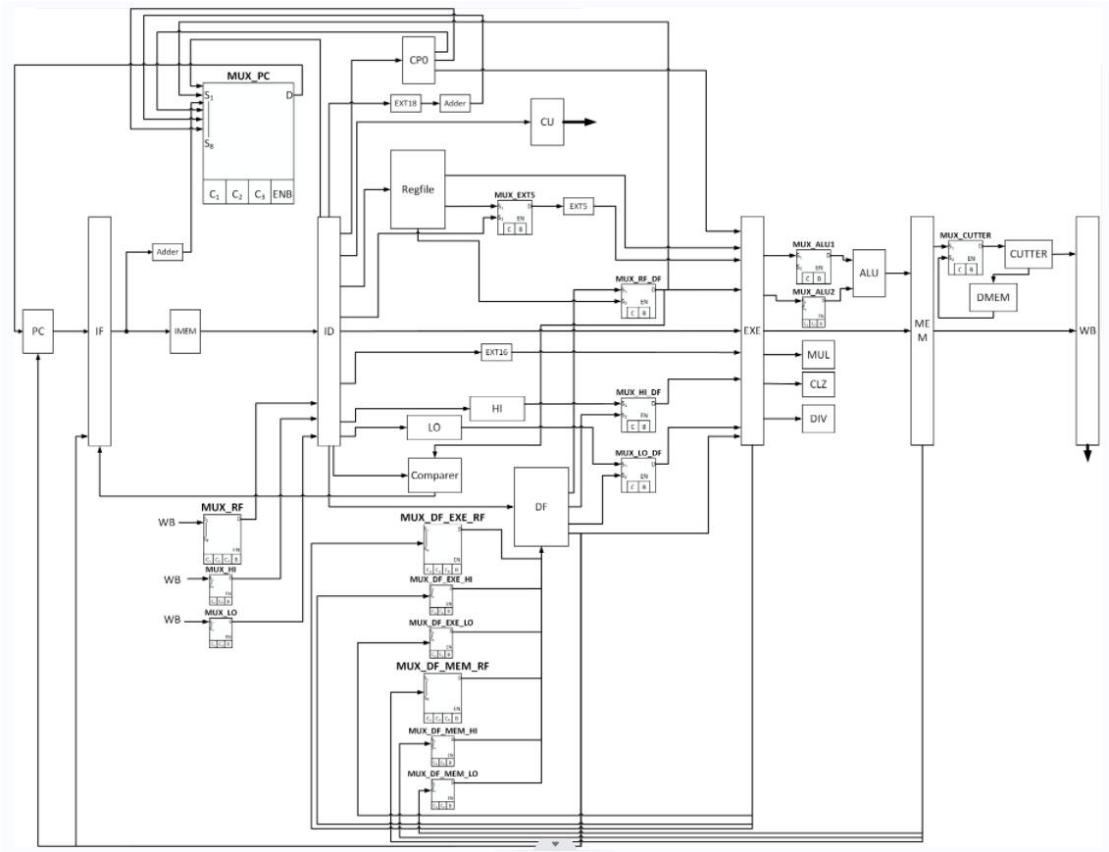
上述的这些阶段按照特定的顺序连接在一起，就形成一个流水线的 CPU，每个阶段会有专门的功能单元执行相关的操作。

这样设计 CPU 可以增强其并行性，因为多个指令可以同时执行，不同指令会占据不同的功能单元，即处在不同的执行阶段。这样提高了 CPU 的并行效率。

2. 静态流水线的设计图：

序号	31条	ucosii V2.52	指令	指令说明	指令格式	OP 31-26	RS 25-21	RT 20-16	RD 15-11	SA 10-6	FUNCT 5-0	指令码 16进制
1	√	√	addi	加立即数	addi rt, rs, immediate	001000				00000	100000	20000000
2	√	√	addiu	加立即数（无符号）	addiu rd, rs, immediate	001001						24000000
3	√	√	andi	立即数与	andi rt, rs, immediate	001100						30000000
4	√	√	ori	或立即数	ori rt, rs, immediate	001101						34000000
5	√	√	sltiu	小于立即数置1（无符号）	sltiu rt, rs, immediate	001011						2C000000
6	√	√	lui	立即数加载高位	lui rt, immediate	001111	00000					3C000000
7	√		xori	异或（立即数）	xori rt, rs, immediate	001110			00000	00000	000000	38000000
8	√		slti	小于置1（立即数）	slti rt, rs, immediate	001010			00000	00000	000000	28000000
9	√	√	addu	加（无符号）	addu rd, rs, rt	000000				00000	100001	00000021
10	√	√	and	与	and rd, rs, rt	000000				00000	100100	00000024
11	√	√	beq	相等时分支	beq rs, rt, offset	000100						10000000
12	√	√	bne	不等时分支	bne rs, rt, offset	000101						14000000
13	√	√	j	跳转	j target	000010						08000000
14	√	√	jal	跳转并链接	jal target	000011						0C000000
15	√	√	jr	跳转至寄存器所指地址	jr rs	000000					001000	00000009
16	√	√	lw	取字	lw rt, offset(base)	100011						8C000000
17	√	√	xor	异或	xor rd, rs, rt	000000				00000	100110	00000026
18	√	√	nor	或非	nor rd, rs, rt	000000				00000	100111	00000027
19	√	√	or	或	or rd, rs, rt	000000				00000	100101	00000025
20	√	√	sll	逻辑左移	sll rd, rt, sa	000000	00000				000000	00000000
21	√	√	sllv	逻辑左移（位数可变）	sllv rd, rt, rs	000000				00000	000100	00000004
22	√	√	sltu	小于置1（无符号）	sltu rd, rs, rt	000000				00000	101011	0000002B
23	√	√	sra	算数右移	sra rd, rt, sa	000000	00000				000011	00000003
24	√	√	srl	逻辑右移	srl rd, rt, sa	000000	00000				000010	00000002
25	√	√	subu	减（无符号）	sub rd, rs, rt	000000				00000	100010	00000022
26	√	√	sw	存字	sw rt, offset(base)	101011						AC000000
27	√		add	加	add rd, rs, rt	000000				00000	100000	00000020
28	√		sub	减	sub rd, rs, rt	000000				00000	100010	00000022
29	√		slt	小于置1	slt rd, rs, rt	000000				00000	101010	0000002A
30	√		srlv	逻辑右移（位数可变）	srlv rd, rt, rs	000000				00000	000110	00000006
31	√		srav	算数右移（位数可变）	srav rd, rt, rs	000000				00000	000111	00000007
32		√	clz	前导零计数	clz rd, rs	011100				00000	100000	70000020
33		√	divu	除（无符号）	divu rs, rt	000000			00000	00000	011011	0000001B
34		√	eret	异常返回	eret	010000	10000	00000	00000	00000	011000	42000018
35		√	jalr	跳转至寄存器所指地址，返回地址保存在	jalr rs	000000		00000			001001	00000008
36		√	lb	取字节	lb rt, offset(base)	100000						80000000
37		√	lbu	取字节（无符号）	lbu rt, offset(base)	100100						90000000

38		√	lhu	取半字（无符号）	lhu rt, offset(base)	100101						94000000
39		√	sb	存字节	sb rt, offset(base)	101000						A0000000
40		√	sh	存半字	sh rt, offset(base)	101001						A4000000
41			lh	取半字	lh rt, offset(base)	100001						84000000
42		√	mfc0	读 CP0 寄存器	mfc0 rt, rd	010000	00000			00000	000000	40000000
43		√	mfhi	读 Hi 寄存器	mfhi rd	000000	00000	00000		00000	010000	00000010
44		√	mflo	读 Lo 寄存器	mflo rd	000000	00000	00000		00000	010010	00000012
45		√	mtc0	写 CP0 寄存器	mtc0 rt, rd	010000	00100			00000	000000	40800000
46		√	mthi	写 Hi 寄存器	mthi rd	000000		00000	00000	00000	010001	00000011
47		√	mtlo	写 Lo 寄存器	mtlo rd	000000		00000	00000	00000	010011	00000013
48		√	mul	乘	mul rd, rs, rt	011100				00000	000010	70000002
49		√	multu	乘（无符号）	multu rs, rt	000000			00000	00000	011001	00000019
50		√	syscall	系统调用	syscall	000000					001100	0000000C
51		√	teq	相等异常	teq rs, rt	000000					110100	00000034
52		√	bgez	大于等于 0 时分支	bgez rs, offset	000001		0001				04010000
53			break	断点	break	000000					001101	0000000D
54			div	除	div rs, rt	000000			00000	00000	011010	0000001A



三、 总体架构部件的解释说明

1、 静态流水线总体结构部件的解释说明

```
//##############################################################################
//# IF 部分
//##############################################################################
    wire jump;          // 跳转信号
    wire stall;         // 延迟信号
    wire [31:0] npc_IF; // 始终是npc=pc+4
    wire [31:0] pc_ID;  // ID段返回的pc值
    wire [31:0] pc_IF;  // IF段流出的pc
    wire [31:0] inst_IF;// IF段流出的inst
    assign npc_IF   = pc + 32'd4;
    assign pc_IF    = (stall) ? pc : (jump) ? pc_ID : npc_IF;
    assign inst_IF  = (stall||jump)? 32'b0 : inst;
    PC PC(clk,rst,pc_IF,pc);

//##############################################################################
//# Pipe_IF_ID 流水线寄存器
//##############################################################################
    wire [31:0] inst_ID;
    wire [31:0] npc_ID;

    Pipe_IF_ID Pipe_IF_ID(
        .clk(clk),
        .rst(rst),
        .npc_IF(npc_IF),
        .inst_IF(inst_IF),
        .npc_ID(npc_ID),
        .inst_ID(inst_ID)
    );
```

上面是关于 IF 模块的调用代码，主要作用就是从 COE 的文件中具体的读取相关的指令，送到下一阶段的指令译码。

具体的代码：

```
module Pipe_IF_ID(
    input clk,
    input rst,
    input [31:0] npc_IF,
    input [31:0] inst_IF,
    output reg [31:0] npc_ID = 32'b0,
    output reg [31:0] inst_ID = 32'b0
    );

    always @(posedge clk or posedge rst) begin
        if(rst) begin
            inst_ID <= 32'b0;   npc_ID <= 32'b0;
        end
        else begin
            inst_ID <= inst_IF; npc_ID <= npc_IF;
        end
    end
endmodule
```

然后是 ID 部分：

```
//##############################################################################
//# ID 部分
//##############################################################################
 //-inst_ID指令解码
    wire [5:0] op,func;
    wire [4:0] rs,rt,rd,sa;
    wire [15:0] imm16;
    wire [25:0] index;
    wire [31:0] sa32_ID;       // sa 拓展
    wire [31:0] uimm32_ID;     // imme(offset) 无符号拓展
    wire [31:0] simm32_ID;     // imme(offset) 有符号拓展
    wire [31:0] offset32;   // offset << 2 拓展
    assign func     = inst_ID[5:0];
    assign sa       = inst_ID[10:6];
    assign imm16    = inst_ID[15:0];
    assign index    = inst_ID[25:0];
    assign op       = inst_ID[31:26];
    assign rs       = inst_ID[25:21];
    assign rt       = inst_ID[20:16];
    assign rd       = inst_ID[15:11];
    assign sa32_ID     = {27'b0, sa};
    assign uimm32_ID   = {16'b0, imm16};
    assign simm32_ID   = {{16{imm16[15]}}, imm16};
    assign offset32 = {{14{imm16[15]}}, imm16, 2'b0};
```

```
 //-pc_ID选择
    wire [31:0] rs_data;     // 寄存器rs中的数据
    wire [31:0] rt_data;     // 寄存器rt中的数据
    wire [1:0] mux_pc;       // pc_ID 选择信号
    wire [31:0] pc_add;      // beq bne bgez: pc + offset32
    wire [31:0] pc_jjal;     // j jal: pc[31:28]和index
    wire [31:0] pc_rs;       // jr: reg中rs位置存储的数据
    assign pc_rs  = rs_data;
    assign pc_add  = pc + offset32;
    assign pc_jjal  = {npc_IF[31:28], index, 2'b0};
    assign pc_ID = (mux_pc[1]) ? pc_add : (mux_pc[0]) ? pc_rs : pc_jjal;
```

```
 //-控制信号产生
    wire DM_w_ID;           // ID段DMEM写信号
    wire write_ID;          // ID段写信号
    wire [3:0] aluc_ID;     // ID段aluc
    wire [4:0] waddr_ID;    // ID段写地址
    wire mux_alua_ID;           // ID段alua来源选择信号
    wire [1:0] mux_alub_ID;     // ID段alub来源选择信号
    wire [1:0] mux_waddr_ID;    // ID段写地址选择信号
    wire [1:0] mux_wdata_ID;    // ID段写数据选择信号
    assign waddr_ID = (mux_waddr_ID[1]) ? 5'd31 : (mux_waddr_ID[0]) ? rd : rt;
    control control(
        .op(op),
        .func(func),
        .rs_data(rs_data),
        .rt_data(rt_data),
        .jump(jump),
        .DM_w_ID(DM_w_ID),
        .write_ID(write_ID),
        .aluc_ID(aluc_ID),
        .mux_pc(mux_pc),
        .mux_alua_ID(mux_alua_ID),
        .mux_alub_ID(mux_alub_ID),
        .mux_waddr_ID(mux_waddr_ID),
        .mux_wdata_ID(mux_wdata_ID)
    );
```

```
//-冲突处理
   wire write_EXE;          // EXE段写信号
   wire write_MEM;          // MEM段写信号
   wire [4:0] waddr_EXE;    // EXE段写地址
   wire [4:0] waddr_MEM;    // MEM段写地址
   conflict conflict(
       .jump(jump),
       .inst(inst),
       .write_ID(write_ID),
       .write_EXE(write_EXE),
       .write_MEM(write_MEM),
       .waddr_ID(waddr_ID),
       .waddr_EXE(waddr_EXE),
       .waddr_MEM(waddr_MEM),
       .stall(stall)
   );
```

我们需要按照之前看到的指令码的结构对读取到的指令进行译码，主要就是根据指令的种类取相应部分的位来作为信息。之后我们将所获得的信息送到 control 模块，让其产生控制信号。

具体的代码：

```
module control(
    input [5:0] op,
    input [5:0] func,
    input [31:0] rs_data,
    input [31:0] rt_data,

    output jump,
    output DM_w_ID,
    output write_ID,
    output [3:0] aluc_ID,

    output [1:0] mux_pc,
    output mux_alua_ID,
    output [1:0] mux_alub_ID,
    output [1:0] mux_waddr_ID,
    output [1:0] mux_wdata_ID
    );
```

```
    wire r_type = ~(op[5]|op[4]|op[3]|op[2]|op[1]|op[0]);
    wire ADD = r_type&func[5]&~func[4]&~func[3]&~func[2]&~func[1]&~func[0];
    wire ADDU = r_type&func[5]&~func[4]&~func[3]&~func[2]&~func[1]&func[0];
    wire SUB = r_type&func[5]&~func[4]&~func[3]&~func[2]&func[1]&~func[0];
    wire SUBU = r_type&func[5]&~func[4]&~func[3]&~func[2]&func[1]&func[0];
    wire AND = r_type&func[5]&~func[4]&~func[3]&func[2]&~func[1]&~func[0];
    wire OR = r_type&func[5]&~func[4]&~func[3]&func[2]&~func[1]&func[0];
    wire XOR = r_type&func[5]&~func[4]&~func[3]&func[2]&func[1]&~func[0];
    wire NOR = r_type&func[5]&~func[4]&~func[3]&func[2]&func[1]&func[0];
    wire SLT = r_type&func[5]&~func[4]&func[3]&~func[2]&func[1]&~func[0];
    wire SLTU = r_type&func[5]&~func[4]&func[3]&~func[2]&func[1]&func[0];
    wire SLL = r_type&~func[5]&~func[4]&~func[3]&~func[2]&~func[1]&~func[0];
    wire SRL = r_type&~func[5]&~func[4]&~func[3]&~func[2]&func[1]&~func[0];
    wire SRA = r_type&~func[5]&~func[4]&~func[3]&~func[2]&func[1]&func[0];
    wire SLLV = r_type&~func[5]&~func[4]&~func[3]&func[2]&~func[1]&~func[0];
    wire SRLV = r_type&~func[5]&~func[4]&~func[3]&func[2]&func[1]&~func[0];
    wire SRAV = r_type&~func[5]&~func[4]&~func[3]&func[2]&func[1]&func[0];
    wire JR = r_type&~func[5]&~func[4]&func[3]&~func[2]&~func[1]&~func[0];
    wire ADDI = ~op[5]&~op[4]&op[3]&~op[2]&~op[1]&~op[0];
    wire ADDIU = ~op[5]&~op[4]&op[3]&~op[2]&~op[1]&op[0];
    wire ANDI = ~op[5]&~op[4]&op[3]&op[2]&~op[1]&~op[0];
    wire ORI = ~op[5]&~op[4]&op[3]&op[2]&~op[1]&op[0];
    wire XORI = ~op[5]&~op[4]&op[3]&op[2]&op[1]&~op[0];
    wire LUI = ~op[5]&~op[4]&op[3]&op[2]&op[1]&op[0];
    wire LW = op[5]&~op[4]&~op[3]&~op[2]&op[1]&op[0];
    wire SW = op[5]&~op[4]&op[3]&~op[2]&op[1]&op[0];
    wire BEQ = ~op[5]&~op[4]&~op[3]&op[2]&~op[1]&~op[0];
    wire BNE = ~op[5]&~op[4]&~op[3]&op[2]&~op[1]&op[0];
    wire SLTI = ~op[5]&~op[4]&op[3]&~op[2]&op[1]&~op[0];
    wire SLTIU = ~op[5]&~op[4]&op[3]&~op[2]&op[1]&op[0];
    wire J = ~op[5]&~op[4]&~op[3]&~op[2]&op[1]&~op[0];
    wire JAL = ~op[5]&~op[4]&~op[3]&~op[2]&op[1]&op[0];
```

```
    assign DM_w_ID = SW;
    assign write_ID = ~(JR|SW|BEQ|BNE|J);
    assign jump = JR|J|JAL|(BEQ&(rs_data == rt_data))|(BNE&(rs_data != rt_data));

    assign aluc_ID[3] = LUI|SLL|SLLV|SLT|SLTI|SLTIU|SLTU|SRA|SRAV|SRL|SRLV;
    assign aluc_ID[2] = AND|ANDI|NOR|OR|ORI|SLL|SLLV|SRA|SRAV|SRL|SRLV|XOR|XORI;
    assign aluc_ID[1] = ADD|ADDI|BEQ|BNE|LW|NOR|SLL|SLLV|SLT|SLTI|SLTIU|SLTU|SUB|SW|XOR|XORI;
    assign aluc_ID[0] = BEQ|BNE|NOR|OR|ORI|SLT|SLTI|SRL|SRLV|SUB|SUBU;

    assign mux_pc = (J|JAL)?2'b00:(JR)?2'b01:(BNE|BEQ)?2'b11:2'bxx;
    assign mux_alua_ID = SLL|SRA|SRL;
    assign mux_alub_ID[1] = ~(ADDI|ADDIU|LUI|LW|SLTI|SW|ANDI|ORI|SLTIU|XORI);
    assign mux_alub_ID[0] = ANDI|ORI|SLTIU|XORI;
    assign mux_wdata_ID[1] = JAL;
    assign mux_wdata_ID[0] = LW;
    assign mux_waddr_ID[1] = JAL;
    assign mux_waddr_ID[0] = ~(ADDI|ADDIU|ANDI|LUI|LW|ORI|SLTI|SLTIU|XORI|JAL);
endmodule
```

完成了上述的指令译码阶段之后，我们需要具体的执行指令，也就是 EXE 模块：

```
Pipe_ID_EXE Pipe_ID_EXE(
    .clk(clk),
    .rst(rst),

    .DM_w_ID(DM_w_ID),
    .write_ID(write_ID),
    .mux_alua_ID(mux_alua_ID),
    .mux_alub_ID(mux_alub_ID),
    .mux_wdata_ID(mux_wdata_ID),
    .aluc_ID(aluc_ID),
    .npc_ID(npc_ID),
    .waddr_ID(waddr_ID),
    .sa32_ID(sa32_ID),
    .simm32_ID(simm32_ID),
    .uimm32_ID(uimm32_ID),
    .rs_data_ID(rs_data),
    .rt_data_ID(rt_data),
    .DM_wdata_ID(rt_data),

    .DM_w_EXE(DM_w_EXE),
    .write_EXE(write_EXE),
    .mux_wdata_EXE(mux_wdata_EXE),
    .mux_alua_EXE(mux_alua_EXE),
    .mux_alub_EXE(mux_alub_EXE),
    .aluc_EXE(aluc_EXE),
    .npc_EXE(npc_EXE),
    .waddr_EXE(waddr_EXE),
    .sa32_EXE(sa32_EXE),
    .simm32_EXE(simm32_EXE),
    .uimm32_EXE(uimm32_EXE),
    .rs_data_EXE(rs_data_EXE),
    .rt_data_EXE(rt_data_EXE),
    .DM_wdata_EXE(DM_wdata_EXE)
);
```

```
//##############################################################################
//# EXE 部分
//##############################################################################
    wire [31:0] alua;
    wire [31:0] alub;
    wire [31:0] alu_EXE;
    assign alua = mux_alua_EXE ? sa32_EXE : rs_data_EXE;
    assign alub = (mux_alub_EXE[1]) ? rt_data_EXE : (mux_alub_EXE[0]) ? uimm32_EXE : simm32_EXE;
    alu alu(alua,alub,aluc_EXE,alu_EXE);
```

```
//##############################################################################
//# Pipe_EXE_MEM 流水线寄存器
//##############################################################################
    wire DM_w_MEM;
    wire [1:0] mux_wdata_MEM;
    wire [31:0] alu_MEM;
    wire [31:0] npc_MEM;
    wire [31:0] DM_wdata_MEM;

    Pipe_EXE_MEM Pipe_EXE_MEM(
        .clk(clk),
        .rst(rst),

        .DM_w_EXE(DM_w_EXE),
        .write_EXE(write_EXE),
        .waddr_EXE(waddr_EXE),
        .mux_wdata_EXE(mux_wdata_EXE),
        .npc_EXE(npc_EXE),
        .alu_EXE(alu_EXE),
        .DM_wdata_EXE(DM_wdata_EXE),

        .DM_w_MEM(DM_w_MEM),
        .write_MEM(write_MEM),
        .waddr_MEM(waddr_MEM),
        .mux_wdata_MEM(mux_wdata_MEM),
        .npc_MEM(npc_MEM),
        .alu_MEM(alu_MEM),
        .DM_wdata_MEM(DM_wdata_MEM)
    );
```

在这一阶段中我们需要根据之前译码所获得的控制信号，执行具体的操作，主要就是利用其中 ALU 模块，执行一个相加，并根据 ALU 的结果结合具体的指令进行执行就可以了。

之后我们需要的就是 MEM 的访存阶段了：

```
//##############################################################################
//# MEM 部分
//##############################################################################
    wire DM_w;
    wire [31:0] DM_addr;
    wire [31:0] DM_rdata;
    wire [31:0] DM_wdata;
    assign DM_addr = alu_MEM;
    assign DM_wdata = DM_wdata_MEM;
    assign DM_w = DM_w_MEM;
    dmem dmem(~clk,rst,DM_w,DM_addr,DM_wdata,DM_rdata);
```

```
//##############################################################################
//# Pipe_MEM_WB 流水线寄存器
//##############################################################################
    wire write_WB;
    wire [4:0] waddr_WB;
    wire [31:0] alu_WB;
    wire [31:0] DM_rdata_WB;
    wire [1:0] mux_wdata_WB;
    wire [31:0] npc_WB;

    Pipe_MEM_WB Pipe_MEM_WB(
        .clk(clk),
        .rst(rst),

        .write_MEM(write_MEM),
        .waddr_MEM(waddr_MEM),
        .mux_wdata_MEM(mux_wdata_MEM),
        .alu_MEM(alu_MEM),
        .npc_MEM(npc_MEM),
        .DM_rdata_MEM(DM_rdata),

        .write_WB(write_WB),
        .waddr_WB(waddr_WB),
        .mux_wdata_WB(mux_wdata_WB),
        .alu_WB(alu_WB),
        .npc_WB(npc_WB),
        .DM_rdata_WB(DM_rdata_WB)
    );
```

我们根据之前的控制信号，进行数据的读取和存入操作。

最后就是写回阶段：

```
//##############################################################################
//# WB 部分
//##############################################################################
    wire [31:0] wdata_WB;
    assign wdata_WB = (mux_wdata_WB[1]) ? npc_WB : (mux_wdata_WB[0]) ? DM_rdata_WB : alu_WB;
    regfile regfile(clk,rst,write_WB,rs,rt,waddr_WB,wdata_WB,rs_data,rt_data);

endmodule
```

2、 静态流水线结构部件的运行总结

与上学期所做的多周期 CPU 不同的是，流水线 CPU 的主要设计思路就是将原本的指令执行过程进行一个细分，原来在单周期或多周期的设计中并不怎么注重这个，但是在流水线中我们将一个指令的执行过程分解为若干个阶段，每个阶段由不同的组件负责，完成相应的工作，并将结果传给下一组件。

我们让不同的指令同时在不同的阶段执行，充分利用 CPU，提高指令的并行性。而五个阶段的流水线，就是取指（IF）、译码（ID）、执行（EXE）、访存（MEM）和写回（WB）。

取指（IF）任务：在这个阶段，处理器从存储器中读取当前要执行的指令，取到相应的指令之后，会进行程序计数器（PC）的更新，具体用到 NPC 等等，其中 PC 储存的就是下一条要执行指令的地址。

译码（ID）任务：在这个阶段，处理器会对取得的指令进行译码，确定指令的类型和操作数，并准备相应的操作数。根据指令的信息我们会产生不同的控制信号，用来帮助后续的进程。

执行（EXE）任务：在这个阶段，处理器会根据指令产生的控制信号使用算术逻辑单元 ALU 进行运算，并产生运算结果。

访存（MEM）任务：在这个阶段，某些指令有着特殊的访存要求，我们在设计时，要让这个模块能够实现对存储器的访问，实现相应的功能。

写回（WB）任务：在这个阶段，处理器需要进行寄存器写入操作。

四、 实验仿真过程

1、 静态流水线的仿真过程

主要的仿真分为前仿真和后仿真，前者是功能仿真，后者是时序仿真，我们导入具体的 testbench 文件，就可以对其进行仿真：

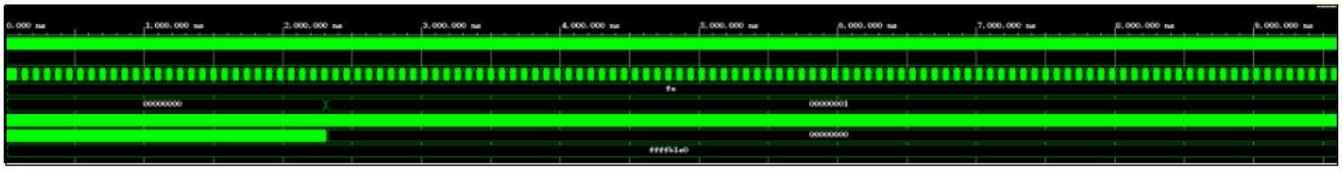
```
module pcpu_top_tb;
    reg clk_in;
    wire clk_real;
    reg reset;
    wire [31:0]inst;
    wire [31:0]pc;
    integer file_output;

    initial  begin
        file_output = $fopen("C:/Users/14065/Desktop/result.txt");
        clk_in = 0;
        reset = 1; #0.5 reset = 0;
    end
    pcpu_top uun(.clk(clk_real),.rst(reset),.inst(inst),.pc(pc));

    always begin
        #0.5 clk_in <= ~clk_in;
    end
    assign clk_real = (inst != 32'b0) ? clk_in : 1'b0;

    always @(posedge clk_real) begin
        $fdisplay(file_output,"pc: %h",pc);
        $fdisplay(file_output,"instr: %h",inst);
        $fdisplay(file_output,"regfiles0: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[0]);
        $fdisplay(file_output,"regfiles1: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[1]);
        $fdisplay(file_output,"regfiles2: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[2]);
        $fdisplay(file_output,"regfiles3: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[3]);
        $fdisplay(file_output,"regfiles4: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[4]);
        $fdisplay(file_output,"regfiles5: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[5]);
        $fdisplay(file_output,"regfiles6: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[6]);
        $fdisplay(file_output,"regfiles7: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[7]);
        $fdisplay(file_output,"regfiles8: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[8]);
        $fdisplay(file_output,"regfiles9: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[9]);
        $fdisplay(file_output,"regfiles10: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[10]);
        $fdisplay(file_output,"regfiles11: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[11]);
        $fdisplay(file_output,"regfiles12: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[12]);
        $fdisplay(file_output,"regfiles13: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[13]);
        $fdisplay(file_output,"regfiles14: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[14]);
        $fdisplay(file_output,"regfiles15: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[15]);
        $fdisplay(file_output,"regfiles16: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[16]);
        $fdisplay(file_output,"regfiles17: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[17]);
        $fdisplay(file_output,"regfiles18: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[18]);
        $fdisplay(file_output,"regfiles19: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[19]);
        $fdisplay(file_output,"regfiles20: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[20]);
        $fdisplay(file_output,"regfiles21: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[21]);
        $fdisplay(file_output,"regfiles22: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[22]);
        $fdisplay(file_output,"regfiles23: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[23]);
        $fdisplay(file_output,"regfiles24: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[24]);
        $fdisplay(file_output,"regfiles25: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[25]);
        $fdisplay(file_output,"regfiles26: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[26]);
        $fdisplay(file_output,"regfiles27: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[27]);
        $fdisplay(file_output,"regfiles28: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[28]);
        $fdisplay(file_output,"regfiles29: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[29]);
        $fdisplay(file_output,"regfiles30: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[30]);
        $fdisplay(file_output,"regfiles31: %h",pcpu_top_tb.uun.pcpu.regfile.array_reg[31]);
    end
endmodule
```

之后我们需要导入具体的 COE 文件，然后分别进行功能仿真和时序仿真就可以了。

五、 实验仿真的波形图及某时刻寄存器值的物理意义

1、 静态流水线的波形图及某时刻寄存器值的物理意义

功能仿真的时序图：



时序仿真的时序图：

相应的 Mars 寄存器：

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x00000000
$sp	29	0x00000000
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

具体的寄存器的值如图所示，我们可以看到学过的 32 个寄存器对应的编号和其中存放的具体数值。我们在具体的指令的执行过程中，在使用算术逻辑单元 ALU 进行计算的时候，会使用到上述的寄存器，在访存和写回的阶段也会使用到上述寄存器。

具体的物理意义就是模拟 CPU 的运算过程，实现指令的具体执行。

六、 流水线 CPU 实验性能验证模型

实验性能验证模型：比萨塔摔鸡蛋游戏。两个同学在可变换层数的比萨塔上摔鸡蛋，一个同学秘密设定同一批鸡蛋耐摔值；另一个同学在指定层高的比萨塔拿着鸡蛋往下摔，用最少的摔次数和摔破的鸡蛋数

求出鸡蛋的耐摔值。假定在耐摔值的楼层及其下面楼层，鸡蛋摔不破，可以重复使用，否则鸡蛋摔破。要求模型的算法输出包括：摔的总次数、摔的总鸡蛋数、最后摔的鸡蛋是否摔破。请使用 C 语言设计该验证模型的算法，并把 C 语言汇编为 RISC-V 指令汇编程序，同时利用编译器生成 RISC-V 指令集可执行目标程序。

上述问题是一个典型的二分查找的问题，我们在设计具体的 C 语言的程序的时候，可以将其设计为如下的二分查找函数：

```c
#include <stdio.h>

int drop_egg(int n, int k) {
    int lo = 1, hi = n, mid;
    int drop_count = 0, egg_count = 0;

    while (lo <= hi) {
        mid = lo + (hi - lo) / 2;
        drop_count++;
        if (mid < k) {
            lo = mid + 1;
            egg_count++;
        } else if (mid > k) {
            hi = mid - 1;
            egg_count++;
        } else {
            egg_count++;
            printf("Egg didn't break at floor %d\n", mid);
            printf("Total drops: %d\n", drop_count);
            printf("Total eggs used: %d\n", egg_count);
            return 0;
        }
    }
```

```c
    printf("Egg broke at floor %d\n", hi);
    printf("Total drops: %d\n", drop_count);
    printf("Total eggs used: %d\n", egg_count);
    return 1;
}

int main() {
    int n = 100, k = 50;
    drop_egg(n, k);
    return 0;
}
```

然后我们需要将上述的 C 语言代码转换成汇编代码：

```
# Set initial durability threshold for survivable falls
addi $9, $0, 60
# Set initial maximum floor height
addi $10, $0, 100
# Initialize counters for statistics
addi $24, $0, 0 # Total attempts
addi $25, $0, 0 # Broken eggs
# Initialize loop variables
addi $1, $0, 1
add $2, $10, $0
# Loop to determine maximum survivable floor height
initialize_loop:
 # Check if $1 < $2
 slt $5, $1, $2
 bne $5, $0, main_loop
 beq $1, $2, main_loop
 # If not, exit the loop
 j end
```

```
# Main loop
main_loop:
 # Calculate midpoint between $1 and $2
 add $4, $1, $2
 sra $3, $4, 1
 # Check if midpoint is less than durability threshold
 slt $5, $3, $9
 bne $5, $0, lessequ
 beq $3, $9, lessequ
 # Egg breaks (midpoint > threshold)
 addi $24, $24, 1 # Increment total attempts
 addi $25, $25, 1 # Increment broken eggs
 addi $2, $3, -1 # Adjust maximum floor height
 # Check if $1 < $2
 slt $5, $1, $2
 bne $5, $0, main_loop
 beq $1, $2, main_loop
 # Reset $1 to 1 and restart the loop
 addi $6, $0, 1
 j end
```

```
# Successful attempt (midpoint <= threshold)
lessequ:
 addi $24, $24, 1 # Increment total attempts
 # Check if $1 == $3
 bne $1, $3, repeat
 addi $3, $3, 1 # Increment $3 if $1 != $3
# Repeat loop for the next floor
repeat:
 addi $3, $3, 0
 addi $1, $3, 0
 j initialize_loop
# End of program
```

之后需要使用 MARS 软件将上面的 asm 程序转变为对应的 COE 文件：

```
memory_initialization_radix = 16;
memory_initialization_vector =
2009003c
200a0064
20180000
20190000
20010001
01401020
08100007
00222020
00041843
0069282a
14a00007
23180001
23390001
2062ffff
0022282a
14a0fff7
20010001
08100006
23180001
14230001
20630001
20610000
08100006
```

之后我们使用 MARS 和我们的程序各测试一遍，如果最终结果的寄存器没有问题即可：

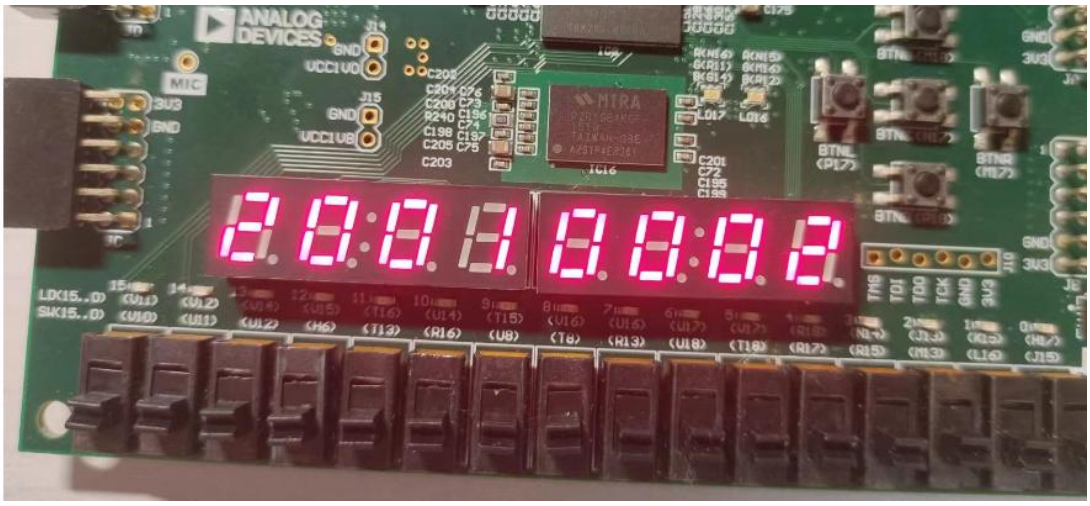
Name	Number	Value
$zero	0	0x00000000
$at	1	0x0000003b
$v0	2	0x0000003b
$v1	3	0x0000003b
$a0	4	0x00000077
$a1	5	0x00000001
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x0000003c
$t2	10	0x00000064
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000aa2
$t9	25	0x00000003
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x00000000
$sp	29	0x00000000
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400018
hi		0x00000000
lo		0x00000000

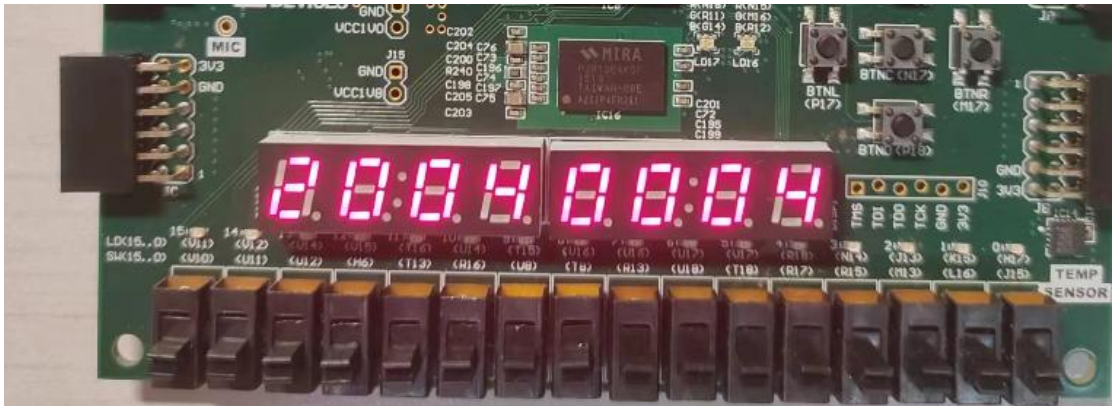
```
pc: 00400060
instr: 01f87823
regfile0: 00000000
regfile1: 00000000
regfile2: 00000000
regfile3: 00000000
regfile4: 00000000
regfile5: 00000000
regfile6: 00000000
regfile7: 00000000
regfile8: 00000238
regfile9: 000003e8
regfile10: 00000237
regfile11: 00000238
regfile12: 00000237
regfile13: 00000001
regfile14: 0000000a
regfile15: 00000006
regfile16: 00000001
regfile17: 00000001
regfile18: 00000000
regfile19: 00000000
regfile20: 00000000
regfile21: 00000000
regfile22: 00000000
regfile23: 00000000
regfile24: 00000001
regfile25: 00000000
regfile26: 00000000
regfile27: 00000000
regfile28: 00000000
regfile29: 00000000
regfile30: 00000000
regfile31: 00000000
```

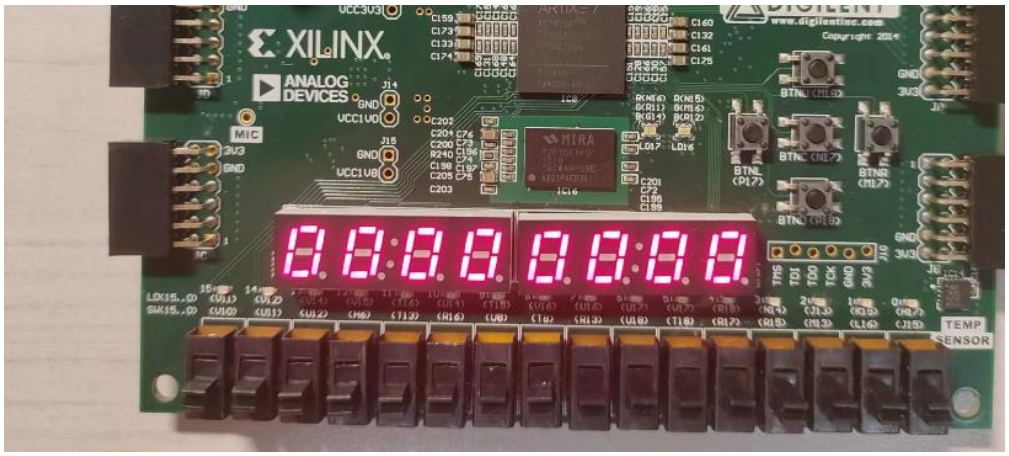
经检查，上述是相同的，验证通过。

七、 实验验算程序下板测试过程与实现

我们利用 MARS 中导出为 COE 文件，进行下板测试，修改相应的 XDC 约束文件，配置相应的管脚。为了使实验结果清晰可见，我们使用七段数码管来观察具体的实验现象。


ANALOG DEVICES
MIC
BTNL (P17)
BTNC (N17)
BTNR (M17)
BTND (P18)


MIC
BTNL (P17)
BTNC (N17)
BTNR (M17)
BTND (P18)
TEMP SENSOR


XILINX
ANALOG DEVICES
MIC
BTNL (P17)
BTNC (N17)
BTNR (M17)
BTND (P18)
TEMP SENSOR


XILINX
ANALOG DEVICES
MIC
BTNL (P17)
BTNC (N17)
BTNR (M17)
BTND (P18)
TEMP SENSOR

八、 流水线的性能指标定性分析（包括：吞吐率、加速比、效率及相关与冲突分析、CPU 的运行时间及存储器空间的使用）

1、 静态流水线的性能指标定性分析

静态流水线的性能指标定性分析，主要可以从以下几个方面来进行分析。分别是吞吐量，延迟，流水线效率和加速比。

吞吐量是指单位时间内完成的指令数量。由于静态流水线能够在同一时间执行多个指令，因此可以提高处理器的吞吐量。我们注意到如果我们减少每个阶段的执行时间，就能够相应的提高吞吐量，但是这也造成了一定的流水线的风险程度，因此选择适宜的阶段周期或者适当的在原有的 5 个阶段中增加阶段都是可以增加流水线吞吐量的方法。

延迟是指从指令进入流水线到完成执行所需的时间。由于指令在流水线中依次经过不同的阶段，每个阶段都需要一定的时间来完成操作。静态流水线的延迟取决于流水线的深度和各个阶段的执行时间。同时不当的延迟可能会增加并行运行的风险，我们在具体的时序仿真可以看到，信号的传递的门逻辑本身有一定的延迟，会加大相关流水线的风险。

流水线效率是指流水线在单位时间内实际执行指令的比例。流水线效率受到流水线冒险（如结构冒险、数据冒险和控制冒险）的影响。

如果流水线冒险发生频率较高，会导致流水线效率下降，因为流水线需要暂停等待冒险解决。较好的流水线设计可以减少冒险的发生，从而提高流水线效率。

加速比是指流水线处理器在相同工作量下的性能提升比例。由于流水线可以并行执行多个指令，因此可以提高处理器的性能。加速比的计算公式为：加速比 = 非流水线处理器的执行时间 / 流水线处理器的执行时间。

九、 总结与体会

本次我们使用之前写的多周期 CPU 进行修改，将其修改成简单的流水线 CPU，感受颇多。

与上学期所做的多周期 CPU 不同的是，流水线 CPU 的主要设计思路就是将原本的指令执行过程进行一个细分，原来在单周期或多周期的设计中并不怎么注重这个，但是在流水线中我们将一个指令的执行过程分解为若干个阶段，每个阶段由不同的组件负责，完成相应的工作，并将结果传给下一组件。

我们让不同的指令同时在不同的阶段执行，充分利用 CPU，提高指令的并行性。而五个阶段的流水线，就是取指（IF）、译码（ID）、执行（EXE）、访存（MEM）和写回（WB）。

取指（IF）任务：在这个阶段，处理器从存储器中读取当前要执行的指令，取到相应的指令之后，会进行程序计数器（PC）的更新，具体用到 NPC 等等，其中 PC 储存的就是下一条要执行指令的地址。

译码（ID）任务：在这个阶段，处理器会对取得的指令进行译码，确定指令的类型和操作数，并准备相应的操作数。根据指令的信息我们会产生不同的控制信号，用来帮助后续的进程。

执行（EXE）任务：在这个阶段，处理器会根据指令产生的控制信号使用算术逻辑单元 ALU 进行运算，并产生运算结果。

访存（MEM）任务：在这个阶段，某些指令有着特殊的访存要求，我们在设计时，要让这个模块能够实现对存储器的访问，实现相应的功能。

写回（WB）任务：在这个阶段，处理器需要进行寄存器写入操作。

在上面的过程中我们不仅手写了每一部分的代码，同时，将具体的流水线 CPU 实现出来，进一步加深了对并行指令的了解，收获颇丰，为之后的学习打下了基础。

十、 附件（所有程序）

1、 静态流水线的设计程序

```
module pcpu_top(
    input clk,
    input rst,
    output [31:0] inst,
    output [31:0] pc
    );
```

```
    wire [31:0] a;
    assign a = pc - 32'h00400000;

    imem imem(a[12:2], inst);
    pcpu pcpu(clk,rst,inst,pc);
endmodule
```

```
module pcpu_onboard(
    input clk_in,   // 板子时钟
    input reset,    // 复位信号
    input ready,    // 数据完备信号
    input id,       // 数据输入选择信号
    input yes,      // 数据输入确定信号
    input [7:0] choose, // 输入数据 0-255
    output working,
    output [7:0] o_seg,
    output [7:0] o_sel
    );
    wire [31:0] i_data;
    wire [31:0] eggs,floor;
    wire [31:0] inst,pc,drop_cnt,egg_cnt,last_result;
    wire clk_cpu;
    wire clk_seg;

    //////////////////////////////////////////////////////////////////////
    // CPU 运行前输入 eggs,floor
    reg state = 0;  // 状态机:0-输入 1-运行
    reg [31:0] temp;
    // 状态机变化
    always@(posedge ready or posedge reset) begin
        if(reset==1) begin
            state <= 0; end
        else if(ready==1) begin
            state <= 1; end
        else begin
            state <= state; end
    end
    // 数据保存
    always@(posedge clk_in or posedge reset) begin
        if(reset==1) begin
```

```
            temp <= 0; end
        else begin
            temp <= {24'b0,choose}; end
    end
    // 实际赋值
    assign eggs  = (state) ? eggs:((id==0) ? ((yes)?temp:eggs):eggs);
    assign floor = (state) ? floor:((id==1) ? ((yes)?temp:floor):floor);

    ////////////////////////////////////////////////////////////////////
    // 选择显示 eggs,floor,inst,pc,drop_cnt,egg_cnt,last_result
    assign working = state;
    assign i_data = (state==0) ? ((id==0) ? eggs:floor):(
                    (choose[6]) ? eggs : (
                    (choose[5]) ? floor :(
                    (choose[4]) ? inst :(
                    (choose[3]) ? pc :(
                    (choose[2]) ? drop_cnt :(
                    (choose[1]) ? egg_cnt :(
                    (choose[0]) ? last_result : 32'b0)))))));

    divider #(10000) div_cpu(clk_in,reset,clk_cpu);
    divider #(4) div_seg(clk_in,reset,clk_seg);
    seg7x16(clk_seg,reset,i_data,o_seg,o_sel);

    ////////////////////////////////////////////////////////////////////
    // pcpu_onboard 顶层部分
    wire [31:0] a;
    wire clk_real;
    assign a = pc - 32'h00400000;
    assign clk_real = (state) ? ((inst != 32'b0) ? clk_cpu : 1'b0) : 1'b0;
    imem imem(a[12:2], inst);
    pcpu_board sccpu(clk_real,reset,inst,pc,drop_cnt,egg_cnt,last_result,eggs,floor);
endmodule

////////////////////////////////////////////////////////////
// (1) regfile_board 模块
module regfile_board (
    input clk,
    input rst,
    input write,
    input [4:0] rna,
    input [4:0] rnb,
    input [4:0] waddr,
    input [31:0] idata,
```

```
    output [31:0] odata1,
    output [31:0] odata2,
    output [31:0] reg_3,reg_4,reg_5,

    input [31:0] eggs,floor
    );

    integer i;
    reg [31:0] array_reg[31:0];

    //从寄存器读取数据
    assign reg_3 = array_reg[3];
    assign reg_4 = array_reg[4];
    assign reg_5 = array_reg[5];
    assign odata1 = (rna)?array_reg[rna]:0;
    assign odata2 = (rnb)?array_reg[rnb]:0;
```

```
    //将数据写入寄存器
    always@(posedge clk or posedge rst) begin
        if(rst==1) begin
            for(i=0;i<6;i=i+1) begin
                array_reg[i]<=0; end
            for(i=8;i<32;i=i+1) begin
                array_reg[i]<=0; end
            array_reg[6] <= eggs;
            array_reg[7] <= floor; end
        else if(waddr != 0 && write) begin
            array_reg[waddr]<=idata;
            array_reg[6] <= eggs;
            array_reg[7] <= floor; end
    end
endmodule
```

```
//////////////////////////////////////////////////////////
// (2) pcpu_board 模块
module pcpu_board(
    input clk,
    input rst,
    input [31:0] inst,
    output [31:0] pc,
    output [31:0] reg_3,reg_4,reg_5,

    input [31:0] eggs,floor
    );
```

```
//##############################################################################
//# IF 部分
//##############################################################################
wire jump;          // 跳转信号
wire stall;         // 延迟信号
wire [31:0] npc_IF; // 始终是 npc=pc+4
wire [31:0] pc_ID;  // ID 段返回的 pc 值
wire [31:0] pc_IF;  // IF 段流出的 pc
wire [31:0] inst_IF;// IF 段流出的 inst
assign npc_IF   = pc + 32'd4;
assign pc_IF    = (stall) ? pc : (jump) ? pc_ID : npc_IF;
assign inst_IF  = (stall||jump)? 32'b0 : inst;
PC PC(clk,rst,pc_IF,pc);

//##############################################################################
//# Pipe_IF_ID 流水线寄存器
//##############################################################################
wire [31:0] inst_ID;
wire [31:0] npc_ID;

Pipe_IF_ID Pipe_IF_ID(
    .clk(clk),
    .rst(rst),
    .npc_IF(npc_IF),
    .inst_IF(inst_IF),
    .npc_ID(npc_ID),
    .inst_ID(inst_ID)
);

//##############################################################################
//# ID 部分
//##############################################################################
//-inst_ID 指令解码
wire [5:0] op,func;
wire [4:0] rs,rt,rd,sa;
wire [15:0] imm16;
wire [25:0] index;
wire [31:0] sa32_ID;       // sa 拓展
wire [31:0] uimm32_ID;     // imme(offset) 无符号拓展
wire [31:0] simm32_ID;     // imme(offset) 有符号拓展
wire [31:0] offset32;   // offset << 2 拓展
assign func     = inst_ID[5:0];
assign sa       = inst_ID[10:6];
```

```
assign imm16    = inst_ID[15:0];
assign index    = inst_ID[25:0];
assign op       = inst_ID[31:26];
assign rs       = inst_ID[25:21];
assign rt       = inst_ID[20:16];
assign rd       = inst_ID[15:11];
assign sa32_ID     = {27'b0, sa};
assign uimm32_ID   = {16'b0, imm16};
assign simm32_ID   = {{16{imm16[15]}}, imm16};
assign offset32 = {{14{imm16[15]}}, imm16, 2'b0};

//-pc_ID 选择
wire [31:0] rs_data;    // 寄存器 rs 中的数据
wire [31:0] rt_data;    // 寄存器 rt 中的数据
wire [1:0] mux_pc;      // pc_ID 选择信号
wire [31:0] pc_add;     // beq bne bgez: pc + offset32
wire [31:0] pc_jjal;    // j jal: pc[31:28]和 index
wire [31:0] pc_rs;      // jr: reg 中 rs 位置存储的数据
assign pc_rs  = rs_data;
assign pc_add  = pc + offset32;
assign pc_jjal  = {npc_IF[31:28], index, 2'b0};
assign pc_ID = (mux_pc[1]) ? pc_add : (mux_pc[0]) ? pc_rs : pc_jjal;

//-控制信号产生
wire DM_w_ID;           // ID 段 DMEM 写信号
wire write_ID;          // ID 段写信号
wire [3:0] aluc_ID;     // ID 段 aluc
wire [4:0] waddr_ID;    // ID 段写地址
wire mux_alua_ID;           // ID 段 alua 来源选择信号
wire [1:0] mux_alub_ID;     // ID 段 alub 来源选择信号
wire [1:0] mux_waddr_ID;    // ID 段写地址选择信号
wire [1:0] mux_wdata_ID;    // ID 段写数据选择信号
assign waddr_ID = (mux_waddr_ID[1]) ? 5'd31 : (mux_waddr_ID[0]) ? rd : rt;
control control(
    .op(op),
    .func(func),
    .rs_data(rs_data),
    .rt_data(rt_data),
    .jump(jump),
    .DM_w_ID(DM_w_ID),
    .write_ID(write_ID),
    .aluc_ID(aluc_ID),
    .mux_pc(mux_pc),
    .mux_alua_ID(mux_alua_ID),
```

```
    .mux_alub_ID(mux_alub_ID),
    .mux_waddr_ID(mux_waddr_ID),
    .mux_wdata_ID(mux_wdata_ID)
);

//-冲突处理
wire write_EXE;         // EXE 段写信号
wire write_MEM;         // MEM 段写信号
wire [4:0] waddr_EXE;   // EXE 段写地址
wire [4:0] waddr_MEM;   // MEM 段写地址
conflict conflict(
    .jump(jump),
    .inst(inst),
    .write_ID(write_ID),
    .write_EXE(write_EXE),
    .write_MEM(write_MEM),
    .waddr_ID(waddr_ID),
    .waddr_EXE(waddr_EXE),
    .waddr_MEM(waddr_MEM),
    .stall(stall)
);

//##############################################################################
//# Pipe_ID_EXE 流水线寄存器
//##############################################################################
wire [31:0] sa32_EXE;
wire [31:0] simm32_EXE;
wire [31:0] uimm32_EXE;
wire [31:0] rs_data_EXE;
wire [31:0] rt_data_EXE;
wire [3:0] aluc_EXE;
wire [31:0] npc_EXE;
wire [31:0] DM_wdata_EXE;
wire DM_w_EXE;
wire mux_alua_EXE;
wire [1:0] mux_alub_EXE;
wire [1:0] mux_wdata_EXE;

Pipe_ID_EXE Pipe_ID_EXE(
    .clk(clk),
    .rst(rst),

    .DM_w_ID(DM_w_ID),
    .write_ID(write_ID),
```

```
    .mux_alua_ID(mux_alua_ID),
    .mux_alub_ID(mux_alub_ID),
    .mux_wdata_ID(mux_wdata_ID),
    .aluc_ID(aluc_ID),
    .npc_ID(npc_ID),
    .waddr_ID(waddr_ID),
    .sa32_ID(sa32_ID),
    .simm32_ID(simm32_ID),
    .uimm32_ID(uimm32_ID),
    .rs_data_ID(rs_data),
    .rt_data_ID(rt_data),
    .DM_wdata_ID(rt_data),

    .DM_w_EXE(DM_w_EXE),
    .write_EXE(write_EXE),
    .mux_wdata_EXE(mux_wdata_EXE),
    .mux_alua_EXE(mux_alua_EXE),
    .mux_alub_EXE(mux_alub_EXE),
    .aluc_EXE(aluc_EXE),
    .npc_EXE(npc_EXE),
    .waddr_EXE(waddr_EXE),
    .sa32_EXE(sa32_EXE),
    .simm32_EXE(simm32_EXE),
    .uimm32_EXE(uimm32_EXE),
    .rs_data_EXE(rs_data_EXE),
    .rt_data_EXE(rt_data_EXE),
    .DM_wdata_EXE(DM_wdata_EXE)
);

//##############################################################################
//# EXE 部分
//##############################################################################
wire [31:0] alua;
wire [31:0] alub;
wire [31:0] alu_EXE;
assign alua = mux_alua_EXE ? sa32_EXE : rs_data_EXE;
assign alub = (mux_alub_EXE[1]) ? rt_data_EXE : (mux_alub_EXE[0]) ? uimm32_EXE : simm32_EXE;
alu alu(alua,alub,aluc_EXE,alu_EXE);

//##############################################################################
//# Pipe_EXE_MEM 流水线寄存器
//##############################################################################
wire DM_w_MEM;
wire [1:0] mux_wdata_MEM;
```

```
wire [31:0] alu_MEM;
wire [31:0] npc_MEM;
wire [31:0] DM_wdata_MEM;

Pipe_EXE_MEM Pipe_EXE_MEM(
    .clk(clk),
    .rst(rst),

    .DM_w_EXE(DM_w_EXE),
    .write_EXE(write_EXE),
    .waddr_EXE(waddr_EXE),
    .mux_wdata_EXE(mux_wdata_EXE),
    .npc_EXE(npc_EXE),
    .alu_EXE(alu_EXE),
    .DM_wdata_EXE(DM_wdata_EXE),

    .DM_w_MEM(DM_w_MEM),
    .write_MEM(write_MEM),
    .waddr_MEM(waddr_MEM),
    .mux_wdata_MEM(mux_wdata_MEM),
    .npc_MEM(npc_MEM),
    .alu_MEM(alu_MEM),
    .DM_wdata_MEM(DM_wdata_MEM)
);

//##############################################################################
//# MEM 部分
//##############################################################################
wire DM_w;
wire [31:0] DM_addr;
wire [31:0] DM_rdata;
wire [31:0] DM_wdata;
assign DM_addr = alu_MEM;
assign DM_wdata = DM_wdata_MEM;
assign DM_w = DM_w_MEM;
dmem dmem(~clk,rst,DM_w,DM_addr,DM_wdata,DM_rdata);

//##############################################################################
//# Pipe_MEM_WB 流水线寄存器
//##############################################################################
wire write_WB;
wire [4:0] waddr_WB;
wire [31:0] alu_WB;
wire [31:0] DM_rdata_WB;
```

```
    wire [1:0] mux_wdata_WB;
    wire [31:0] npc_WB;

    Pipe_MEM_WB Pipe_MEM_WB(
        .clk(clk),
        .rst(rst),

        .write_MEM(write_MEM),
        .waddr_MEM(waddr_MEM),
        .mux_wdata_MEM(mux_wdata_MEM),
        .alu_MEM(alu_MEM),
        .npc_MEM(npc_MEM),
        .DM_rdata_MEM(DM_rdata),

        .write_WB(write_WB),
        .waddr_WB(waddr_WB),
        .mux_wdata_WB(mux_wdata_WB),
        .alu_WB(alu_WB),
        .npc_WB(npc_WB),
        .DM_rdata_WB(DM_rdata_WB)
    );

    //##############################################################################
    //# WB 部分
    //##############################################################################
    wire [31:0] wdata_WB;
    assign wdata_WB = (mux_wdata_WB[1]) ? npc_WB : (mux_wdata_WB[0]) ? DM_rdata_WB : alu_WB;
    regfile_board
regfile_board(clk,rst,write_WB,rs,rt,waddr_WB,wdata_WB,rs_data,rt_data,reg_3,reg_4,reg_5,eggs,
floor);
```

```
endmodule
```

```

module Pipe_EXE_MEM(
    input clk,
    input rst,
```

```
    input DM_w_EXE,
    input write_EXE,
    input [4:0] waddr_EXE,
    input [1:0] mux_wdata_EXE,
    input [31:0] alu_EXE,
```

```
    input [31:0] npc_EXE,
    input [31:0] DM_wdata_EXE,
```

```
    output reg DM_w_MEM = 1'b0,
    output reg write_MEM = 1'b0,
    output reg [4:0] waddr_MEM = 5'b0,
    output reg [1:0] mux_wdata_MEM = 2'b0,
    output reg [31:0] alu_MEM = 32'b0,
    output reg [31:0] npc_MEM = 32'b0,
    output reg [31:0] DM_wdata_MEM = 32'b0
    );
```

```
    always @(posedge rst or posedge clk) begin
        if(rst) begin
            write_MEM <= 1'b0;
            alu_MEM <= 32'b0;
            waddr_MEM <= 5'b0;
            mux_wdata_MEM <= 1'b0;
            DM_wdata_MEM <= 32'b0;
            DM_w_MEM <= 1'b0;
            npc_MEM <= 32'b0;
        end
        else begin
            write_MEM <= write_EXE;
            alu_MEM <= alu_EXE;
            waddr_MEM <= waddr_EXE;
            mux_wdata_MEM <= mux_wdata_EXE;
            DM_wdata_MEM <= DM_wdata_EXE;
            DM_w_MEM <= DM_w_EXE;
            npc_MEM <= npc_EXE;
        end
    end
endmodule
```

```

module Pipe_ID_EXE(
    input clk,
    input rst,
```

```
    input DM_w_ID,
    input write_ID,
```

```
input mux_alua_ID,
input [1:0] mux_alub_ID,
input [1:0] mux_wdata_ID,
input [3:0] aluc_ID,
input [31:0] npc_ID,
input [4:0] waddr_ID,
input [31:0] sa32_ID,
input [31:0] simm32_ID,
input [31:0] uimm32_ID,
input [31:0] rs_data_ID,
input [31:0] rt_data_ID,
input [31:0] DM_wdata_ID,
```

```
output reg DM_w_EXE = 1'b0,
output reg write_EXE = 1'b0,
output reg mux_alua_EXE = 1'b0,
output reg [1:0] mux_alub_EXE = 2'b0,
output reg [1:0] mux_wdata_EXE = 2'b0,
output reg [3:0] aluc_EXE = 4'b0,
output reg [31:0] npc_EXE = 32'b0,
output reg [4:0] waddr_EXE = 5'b0,
output reg [31:0] sa32_EXE = 32'b0,
output reg [31:0] simm32_EXE = 32'b0,
output reg [31:0] uimm32_EXE = 32'b0,
output reg [31:0] rs_data_EXE = 32'b0,
output reg [31:0] rt_data_EXE = 32'b0,
output reg [31:0] DM_wdata_EXE = 32'b0
);
```

```
always @(posedge rst or posedge clk) begin
    if(rst) begin
        rs_data_EXE <= 32'b0;
        sa32_EXE <= 32'b0;
        simm32_EXE <= 32'b0;
        uimm32_EXE <= 32'b0;
        rt_data_EXE <= 32'b0;
        mux_alua_EXE <= 1'b0;
        mux_alub_EXE <= 2'b0;
        aluc_EXE <= 4'b0;
        write_EXE <= 'b0;
        waddr_EXE <= 5'b0;
        mux_wdata_EXE <= 1'b0;
        DM_wdata_EXE <= 32'b0;
        DM_w_EXE <= 1'b0;
```

```
            npc_EXE <= 32'b0;
        end
        else begin
            rs_data_EXE <= rs_data_ID;
            sa32_EXE <= sa32_ID;
            simm32_EXE <= simm32_ID;
            uimm32_EXE <= uimm32_ID;
            rt_data_EXE <= rt_data_ID;
            mux_alua_EXE <= mux_alua_ID;
            mux_alub_EXE <= mux_alub_ID;
            aluc_EXE <= aluc_ID;
            write_EXE <= write_ID;
            waddr_EXE <= waddr_ID;
            mux_wdata_EXE <= mux_wdata_ID;
            DM_wdata_EXE <= DM_wdata_ID;
            DM_w_EXE <= DM_w_ID;
            npc_EXE <= npc_ID;
        end
    end
endmodule
```

```
module Pipe_IF_ID(
    input clk,
    input rst,
    input [31:0] npc_IF,
    input [31:0] inst_IF,
    output reg [31:0] npc_ID = 32'b0,
    output reg [31:0] inst_ID = 32'b0
    );

    always @(posedge clk or posedge rst) begin
        if(rst) begin
            inst_ID <= 32'b0;   npc_ID <= 32'b0;
        end
        else begin
            inst_ID <= inst_IF; npc_ID <= npc_IF;
        end
    end
endmodule
```

```
module Pipe_MEM_WB(
    input clk,
    input rst,

    input write_MEM,
    input [4:0] waddr_MEM,
    input [1:0] mux_wdata_MEM,
    input [31:0] alu_MEM,
    input [31:0] npc_MEM,
    input [31:0] DM_rdata_MEM,
```

```
    output reg write_WB = 1'b0,
    output reg [4:0] waddr_WB = 5'b0,
    output reg [1:0] mux_wdata_WB = 2'b0,
    output reg [31:0] alu_WB = 32'b0,
    output reg [31:0] npc_WB = 32'b0,
    output reg [31:0] DM_rdata_WB = 32'b0
    );
```

```
    always @(posedge rst or posedge clk) begin
        if(rst) begin
            write_WB <= 1'b0;
            alu_WB <= 32'b0;
            DM_rdata_WB <= 32'b0;
            waddr_WB <= 5'b0;
            mux_wdata_WB <= 1'b0;
            npc_WB <= 32'b0;
        end
        else begin
            write_WB <= write_MEM;
            alu_WB <= alu_MEM;
            DM_rdata_WB <= DM_rdata_MEM;
            waddr_WB <= waddr_MEM;
            mux_wdata_WB <= mux_wdata_MEM;
            npc_WB <= npc_MEM;
        end
    end
endmodule
```

```
module regfile(
```

```
    input clk,
    input rst,
    input write,
    input [4:0] rna,
    input [4:0] rnb,
    input [4:0] waddr,
    input [31:0] idata,
    output [31:0] odata1,
    output [31:0] odata2
    );

    integer i;
    reg [31:0] array_reg[31:0];

    //从寄存器读取数据
    assign odata1 = (rna)?array_reg[rna]:0;
    assign odata2 = (rnb)?array_reg[rnb]:0;

    //将数据写入寄存器
    always@(posedge clk or posedge rst) begin
        if(rst==1) begin
            for(i=0;i<32;i=i+1) begin
                array_reg[i]<=0; end
        end
        else if(waddr != 0 && write) begin
            array_reg[waddr]<=idata;
        end
    end
//    // 仅在需要输入时测试用
//    always@(posedge clk or posedge rst) begin
//        if(rst==1) begin
//            for(i=0;i<6;i=i+1) begin
//                array_reg[i]<=0; end
//            for(i=8;i<32;i=i+1) begin
//                array_reg[i]<=0; end
//            array_reg[6] <= 2;
//            array_reg[7] <= 6;
//        end
//        else if(waddr != 0 && write) begin
//            array_reg[waddr]<=idata;
//        end
//    end
endmodule
```

```
module seg7x16(
    input clk,
    input reset,
    input [31:0] i_data,
    output [7:0] o_seg,
    output [7:0] o_sel
    );
```

```
    reg [14:0] cnt;
    always @ (posedge clk, posedge reset)
        if (reset)  cnt <= 0;
        else        cnt <= cnt + 1'b1;

    wire seg7_clk = cnt[14];
    reg [2:0] seg7_addr;
    always @ (posedge seg7_clk, posedge reset)
       if(reset)   seg7_addr <= 0;
       else        seg7_addr <= seg7_addr + 1'b1;

    reg [7:0] o_sel_r;
    always @ (*)
       case(seg7_addr)
           7 : o_sel_r = 8'b01111111;
           6 : o_sel_r = 8'b10111111;
           5 : o_sel_r = 8'b11011111;
           4 : o_sel_r = 8'b11101111;
           3 : o_sel_r = 8'b11110111;
           2 : o_sel_r = 8'b11111011;
           1 : o_sel_r = 8'b11111101;
           0 : o_sel_r = 8'b11111110;
       endcase

    reg [31:0] i_data_store;
    always @ (posedge clk, posedge reset)
       if(reset) i_data_store <= 0;
       else      i_data_store <= i_data;

    reg [7:0] seg_data_r;
    always @ (*)
       case(seg7_addr)
           0 : seg_data_r = i_data_store[3:0];
```

```
            1 : seg_data_r = i_data_store[7:4];
            2 : seg_data_r = i_data_store[11:8];
            3 : seg_data_r = i_data_store[15:12];
            4 : seg_data_r = i_data_store[19:16];
            5 : seg_data_r = i_data_store[23:20];
            6 : seg_data_r = i_data_store[27:24];
            7 : seg_data_r = i_data_store[31:28];
        endcase

    reg [7:0] o_seg_r;
    always @ (posedge clk, posedge reset)
        if(reset)
            o_seg_r <= 8'hff;
        else case(seg_data_r)
            4'h0 : o_seg_r <= 8'hC0;
            4'h1 : o_seg_r <= 8'hF9;
            4'h2 : o_seg_r <= 8'hA4;
            4'h3 : o_seg_r <= 8'hB0;
            4'h4 : o_seg_r <= 8'h99;
            4'h5 : o_seg_r <= 8'h92;
            4'h6 : o_seg_r <= 8'h82;
            4'h7 : o_seg_r <= 8'hF8;
            4'h8 : o_seg_r <= 8'h80;
            4'h9 : o_seg_r <= 8'h90;
            4'hA : o_seg_r <= 8'h88;
            4'hB : o_seg_r <= 8'h83;
            4'hC : o_seg_r <= 8'hC6;
            4'hD : o_seg_r <= 8'hA1;
            4'hE : o_seg_r <= 8'h86;
            4'hF : o_seg_r <= 8'h8E;
        endcase

    assign o_sel = o_sel_r;
    assign o_seg = o_seg_r;
endmodule
```

```
module cla4(
    input [3:0] g, p,
    input in,
    output og,  //4 位超前进位加法器的进位产生函数
```

```
    output op,  //4 位超前进位加法器的进位传递函数
    output [4:1] out
    );
    assign out[1] = g[0]|(p[0]&in);
    assign out[2] = g[1]|(p[1]&g[0])|(p[1]&p[0]&in);
    assign out[3] = g[2]|(p[2]&g[1])|(p[2]&p[1]&g[0])|(p[2]&p[1]&p[0]&in);
    assign out[4] = g[3]|(p[3]&g[2])|(p[3]&p[2]&g[1])|(p[3]&p[2]&p[1]&g[0])
                   |(p[3]&p[2]&p[1]&p[0]&in);
    assign og = g[3]|(p[3]&g[2])|(p[3]&p[2]&g[1])|(p[3]&p[2]&p[1]&g[0]);
    assign op = p[3]&p[2]&p[1]&p[0];
endmodule
```

```
module add1(
    input a, b,
    input carry,
    output g,   //1 位超前进位加法器产生函数
    output p,   //1 位超前进位加法器的传递函数
    output out
    );
    assign g = a & b;
    assign p = a | b;
    assign out = a ^ b ^ carry;
endmodule
```

```
module add4(
    input [3:0] a,b,
    input in,
    output og,  //4 位超前进位加法器的进位产生函数
    output op,  //4 位超前进位加法器的进位传递函数
    output [3:0] s
    );
    wire [3:0] g, p;
    wire [4:1] carry;
    add1 add_1(a[0], b[0], in, g[0], p[0], s[0]);
    add1 add_2(a[1], b[1], carry[1], g[1], p[1], s[1]);
    add1 add_3(a[2], b[2], carry[2], g[2], p[2], s[2]);
    add1 add_4(a[3], b[3], carry[3], g[3], p[3], s[3]);
    cla4 cla4(g, p, in, og, op, carry);
endmodule
```

```
module add16(
    input [15:0] a,b,
    input in,
    output og,  //16 位超前进位加法器的进位产生函数
```

```
    output op,  //16 位超前进位加法器的进位传递函数
    output [15:0] s
    );
    wire [3:0] g,p;
    wire [4:1] carry;
    add4 add_1(a[3:0], b[3:0], in, g[0], p[0], s[3:0]);
    add4 add_2(a[7:4], b[7:4], carry[1], g[1], p[1], s[7:4]);
    add4 add_3(a[11:8], b[11:8], carry[2], g[2], p[2], s[11:8]);
    add4 add_4(a[15:12], b[15:12], carry[3], g[3], p[3], s[15:12]);
    cla4 cla4(g, p, in, og, op, carry);
endmodule
```

```
module add32(
    input [31:0] a,b,
    input in,
    output [31:0] r,
    output out
    );
    wire temp;
    wire [1:0] g, p;
    assign temp = g[0]|(p[0]&in);
    assign out = g[1]|(p[1]&g[0])|(p[1]&p[0]&in);
    add16 add_1(a[15:0], b[15:0], in, g[0], p[0], r[15:0]);
    add16 add_2(a[31:16], b[31:16], temp, g[1], p[1], r[31:16]);
endmodule
```

```
module neg32(
    input [31:0] in,
    input s,
    output [31:0] out
    );
    assign out[0] = s ^ in[0];
    assign out[1] = s ^ in[1];
    assign out[2] = s ^ in[2];
    assign out[3] = s ^ in[3];
    assign out[4] = s ^ in[4];
    assign out[5] = s ^ in[5];
    assign out[6] = s ^ in[6];
    assign out[7] = s ^ in[7];
    assign out[8] = s ^ in[8];
    assign out[9] = s ^ in[9];
    assign out[10] = s ^ in[10];
    assign out[11] = s ^ in[11];
    assign out[12] = s ^ in[12];
```

```
    assign out[13] = s ^ in[13];
    assign out[14] = s ^ in[14];
    assign out[15] = s ^ in[15];
    assign out[16] = s ^ in[16];
    assign out[17] = s ^ in[17];
    assign out[18] = s ^ in[18];
    assign out[19] = s ^ in[19];
    assign out[20] = s ^ in[20];
    assign out[21] = s ^ in[21];
    assign out[22] = s ^ in[22];
    assign out[23] = s ^ in[23];
    assign out[24] = s ^ in[24];
    assign out[25] = s ^ in[25];
    assign out[26] = s ^ in[26];
    assign out[27] = s ^ in[27];
    assign out[28] = s ^ in[28];
    assign out[29] = s ^ in[29];
    assign out[30] = s ^ in[30];
    assign out[31] = s ^ in[31];
endmodule
```

```
module adder(
    input [31:0] a,
    input [31:0] b,
    input aluc,
    output [31:0] r
    );
    wire [31:0] b_real;
    wire carry_in;
    wire carry_out;
    assign carry_in = aluc ? 1: 0;
    neg32 neg32(b, aluc, b_real);
    add32 add32(a, b_real, carry_in, r, carry_out);
endmodule
```

```

module alu(
    input [31:0] a,
    input [31:0] b,
```

```
input [3:0] aluc,
output reg [31:0] result
);
```

```
// 加减法操作结果
wire [31:0] result_0;
adder adder(a, b, aluc[0], result_0);
```

```
// lui 及 slt 操作结果
wire [31:0] result_1;
luislt luislt(a, b, aluc[1:0], result_1);
```

```
// result 最终输出
always @(*) begin
    case(aluc[3:2])
    2'b00: begin //加减操作
        result = result_0; end
    2'b10: begin //lui 和 slt 操作
        result = result_1; end
    2'b01: begin //与或异或非或操作
        case(aluc[1:0])
        2'b00: begin    //与
            result = a & b; end
        2'b01: begin    //或
            result = a | b; end
        2'b10: begin    //异或
            result = a ^ b; end
        default: begin  //非或
            result = ~(a | b); end
        endcase end
    2'b11: begin //移位操作
        case(aluc[1:0])
        2'b00: begin    //算术右移
            result = a[0]? {b[31], b[31:1]}: b;
            result = a[1]? {{2{result[31]}}, result[31:2]}: result;
            result = a[2]? {{4{result[31]}}, result[31:4]}: result;
            result = a[3]? {{8{result[31]}}, result[31:8]}: result;
            result = a[4]? {{16{result[31]}}, result[31:16]}: result; end
        2'b01: begin    //逻辑右移
            result = a[0]? {1'b0, b[31:1]}: b;
            result = a[1]? {2'b0, result[31:2]}: result;
            result = a[2]? {4'b0, result[31:4]}: result;
            result = a[3]? {8'b0, result[31:8]}: result;
            result = a[4]? {16'b0, result[31:16]}: result; end
```

```
            default: begin  //左移
                result = a[0]? {b[30:0], 1'b0}: b;
                result = a[1]? {result[29:0], 2'b0}: result;
                result = a[2]? {result[27:0], 4'b0}: result;
                result = a[3]? {result[23:0], 8'b0}: result;
                result = a[4]? {result[15:0], 16'b0}: result; end
            endcase end
        default: begin
            result = 32'b0; end
        endcase
    end
endmodule
```

```
module conflict(
    input jump,
    input [31:0] inst,
    input write_ID,
    input write_EXE,
    input write_MEM,
    input [4:0] waddr_ID,
    input [4:0] waddr_EXE,
    input [4:0] waddr_MEM,
    output stall
    );
```

```
    wire [4:0] rs;
    wire [4:0] rt;

    assign rs = inst[25:21];
    assign rt = inst[20:16];
```

```
    assign stall =  (~jump)&&(((write_ID && waddr_ID !=
5'b0)&&((rs==waddr_ID)||(rt==waddr_ID)))||
                    ((write_EXE && waddr_EXE != 5'b0)&&((rs==waddr_EXE)||(rt==waddr_EXE)))||
                    ((write_MEM && waddr_MEM != 5'b0)&&((rs==waddr_MEM)||(rt==waddr_MEM))));
endmodule
```

```
module control(
    input [5:0] op,
    input [5:0] func,
    input [31:0] rs_data,
    input [31:0] rt_data,
```

```
    output jump,
    output DM_w_ID,
    output write_ID,
    output [3:0] aluc_ID,
```

```
    output [1:0] mux_pc,
    output mux_alua_ID,
    output [1:0] mux_alub_ID,
    output [1:0] mux_waddr_ID,
    output [1:0] mux_wdata_ID
    );

    wire r_type = ~(op[5]|op[4]|op[3]|op[2]|op[1]|op[0]);
    wire ADD = r_type&func[5]&~func[4]&~func[3]&~func[2]&~func[1]&~func[0];
    wire ADDU = r_type&func[5]&~func[4]&~func[3]&~func[2]&~func[1]&func[0];
    wire SUB = r_type&func[5]&~func[4]&~func[3]&~func[2]&func[1]&~func[0];
    wire SUBU = r_type&func[5]&~func[4]&~func[3]&~func[2]&func[1]&func[0];
    wire AND = r_type&func[5]&~func[4]&~func[3]&func[2]&~func[1]&~func[0];
    wire OR = r_type&func[5]&~func[4]&~func[3]&func[2]&~func[1]&func[0];
    wire XOR = r_type&func[5]&~func[4]&~func[3]&func[2]&func[1]&~func[0];
    wire NOR = r_type&func[5]&~func[4]&~func[3]&func[2]&func[1]&func[0];
    wire SLT = r_type&func[5]&~func[4]&func[3]&~func[2]&func[1]&~func[0];
    wire SLTU = r_type&func[5]&~func[4]&func[3]&~func[2]&func[1]&func[0];
    wire SLL = r_type&~func[5]&~func[4]&~func[3]&~func[2]&~func[1]&~func[0];
    wire SRL = r_type&~func[5]&~func[4]&~func[3]&~func[2]&func[1]&~func[0];
    wire SRA = r_type&~func[5]&~func[4]&~func[3]&~func[2]&func[1]&func[0];
    wire SLLV = r_type&~func[5]&~func[4]&~func[3]&func[2]&~func[1]&~func[0];
    wire SRLV = r_type&~func[5]&~func[4]&~func[3]&func[2]&func[1]&~func[0];
    wire SRAV = r_type&~func[5]&~func[4]&~func[3]&func[2]&func[1]&func[0];
    wire JR = r_type&~func[5]&~func[4]&func[3]&~func[2]&~func[1]&~func[0];
    wire ADDI = ~op[5]&~op[4]&op[3]&~op[2]&~op[1]&~op[0];
    wire ADDIU = ~op[5]&~op[4]&op[3]&~op[2]&~op[1]&op[0];
    wire ANDI = ~op[5]&~op[4]&op[3]&op[2]&~op[1]&~op[0];
    wire ORI = ~op[5]&~op[4]&op[3]&op[2]&~op[1]&op[0];
    wire XORI = ~op[5]&~op[4]&op[3]&op[2]&op[1]&~op[0];
    wire LUI = ~op[5]&~op[4]&op[3]&op[2]&op[1]&op[0];
    wire LW = op[5]&~op[4]&~op[3]&~op[2]&op[1]&op[0];
```

```
    wire SW = op[5]&~op[4]&op[3]&~op[2]&op[1]&op[0];
    wire BEQ = ~op[5]&~op[4]&~op[3]&op[2]&~op[1]&~op[0];
    wire BNE = ~op[5]&~op[4]&~op[3]&op[2]&~op[1]&op[0];
    wire SLTI = ~op[5]&~op[4]&op[3]&~op[2]&op[1]&~op[0];
    wire SLTIU = ~op[5]&~op[4]&op[3]&~op[2]&op[1]&op[0];
    wire J = ~op[5]&~op[4]&~op[3]&~op[2]&op[1]&~op[0];
    wire JAL = ~op[5]&~op[4]&~op[3]&~op[2]&op[1]&op[0];
```

```
    assign DM_w_ID = SW;
    assign write_ID = ~(JR|SW|BEQ|BNE|J);
    assign jump = JR|J|JAL|(BEQ&(rs_data == rt_data))|(BNE&(rs_data != rt_data));

    assign aluc_ID[3] = LUI|SLL|SLLV|SLT|SLTI|SLTIU|SLTU|SRA|SRAV|SRL|SRLV;
    assign aluc_ID[2] = AND|ANDI|NOR|OR|ORI|SLL|SLLV|SRA|SRAV|SRL|SRLV|XOR|XORI;
    assign aluc_ID[1] = ADD|ADDI|BEQ|BNE|LW|NOR|SLL|SLLV|SLT|SLTI|SLTIU|SLTU|SUB|SW|XOR|XORI;
    assign aluc_ID[0] = BEQ|BNE|NOR|OR|ORI|SLT|SLTI|SRL|SRLV|SUB|SUBU;
```

```
    assign mux_pc = (J|JAL)?2'b00:(JR)?2'b01:(BNE|BEQ)?2'b11:2'bxx;
    assign mux_alua_ID = SLL|SRA|SRL;
    assign mux_alub_ID[1] = ~(ADDI|ADDIU|LUI|LW|SLTI|SW|ANDI|ORI|SLTIU|XORI);
    assign mux_alub_ID[0] = ANDI|ORI|SLTIU|XORI;
    assign mux_wdata_ID[1] = JAL;
    assign mux_wdata_ID[0] = LW;
    assign mux_waddr_ID[1] = JAL;
    assign mux_waddr_ID[0] = ~(ADDI|ADDIU|ANDI|LUI|LW|ORI|SLTI|SLTIU|XORI|JAL);
endmodule
```

```
module divider#(parameter num = 2)(
    input I_CLK,
    input rst,
    output reg O_CLK
    );
    integer i = 0;
    always @ (posedge I_CLK or posedge rst) begin
        if(rst == 1) begin
            O_CLK <= 0; i <= 0; end
        else begin
            if(i == num - 1) begin
                O_CLK <= ~O_CLK; i <= 0; end
            else begin
```

```
                i <= i + 1; end
        end
    end
endmodule
```

```
module dmem(
    input clk,
    input rst,
    input write,
    input [31:0] addr,
    input [31:0] idata,
    output [31:0] odata
    );

    integer i;
    reg [31:0] memory[0:2047];
    wire [31:0] addr_real;

    assign addr_real = addr - 32'h10010000;
    assign odata = memory[addr_real[10:0]];
```

```
    always @(negedge clk or posedge rst) begin
        if(rst==1) begin
            for(i=0;i<2048;i=i+1) memory[i] <= 0; end
        else if(write==1) begin
            memory[addr_real[10:0]] <= idata; end
    end
endmodule
```