实验一—流水加法器及指令拓展

姓名：张云策 学号：3200105787 学院：计算机科学与技术学院

课程名称：计算机系统II 同组学生姓名：/

实验时间：2021. 实验地点：紫金港机房 指导老师：申文博

一、 实验目的和要求

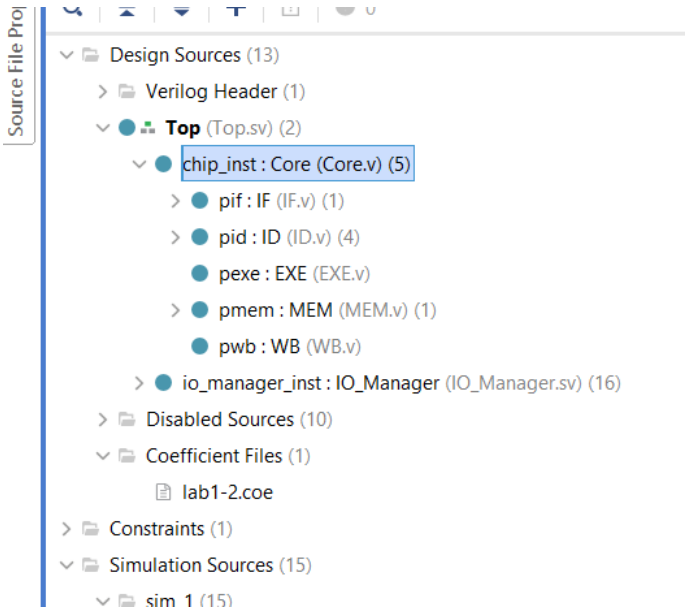
- 理解流水线的基本概念与思想
- 基于在单周期 CPU 中已经实现的模块，实现 5 级流水线框架
- 理解流水线设计在提高 CPU 的吞吐率，提升整体性能上的作用与优越性
- 扩展指令，理解不同类型指令在流水线中的运作差异
- 实现功能基本完善的流水线 CPU

二、 实验内容和原理

2.1 实验内容

实现流水线 CPU 的基本功能

2.2 设计模块



IF:

```
module IF(
    input clk,
    input rst,
    input [31:0] pc_pre,
    input [31:0] pc_j,
    input [31:0] if_inst,
    output reg [31:0] if_pc_out,
    output reg [31:0] id_pc_out,
    output reg [31:0] id_inst
    );
    initial begin
    if_pc_out = 0;
    id_pc_out = 0;
    id_inst= 0 ;
    end
    always @(posedge clk or posedge rst) begin
        if(rst) if_pc_out<=0;
        else if(pc_j!=0) if_pc_out = pc_j;
        else if_pc_out = if_pc_out+4;
    end
    Rom rom_unit (
            .a(if_pc_out[12:2]),
            .spo(if_inst)
    );
    always @ (posedge clk) begin
        id_pc_out <= pc_pre;
        id_inst   <= if_inst;
    end
endmodule
```

ID:

```
module ID(
    input clk,
    input rst,
    input  [31:0] id_inst,
    input  [31:0] id_pc_out,
    input  [31:0] id_result,
    input  [31:0] wb_inst,
    input  [12:0] wb_CF,
    output reg[31:0] exe_data1,
    output reg[31:0] exe_data2,
    output reg[31:0] exe_data3,
    output reg[31:0] exe_inst,
    output reg[31:0] exe_pc_out,
    output reg[12:0] exe_CF
    );
    wire [31:0] data_1,data_2,data_3;
    wire [3:0] alu_op;
    wire [1:0] pc_src, mem_to_reg;
    wire reg_write, alu_src_b, branch, b_type,mem_write;
    wire  [12:0] CF;
    Control lcontrol(
        .op_code(id_inst[6:0]),
        .funct3(id_inst[14:12]),
        .funct7_5(id_inst[30]),
        .pc_src(pc_src),
        .reg_write(reg_write),
        .alu_src_b(alu_src_b),
        .alu_op(alu_op),
        .mem_to_reg(mem_to_reg),
        .mem_write(mem_write),
        .branch(branch),
```

```
        .mem_write(mem_write),
        .branch(branch),
        .b_type(b_type)
    );
    assign CF={alu_op,pc_src,mem_to_reg,mem_write,reg_write,alu_src_b,branch,b_type};
    Regs lregs(
        .clk(clk),
        .rst(rst),
        .we(wb_CF[3]),
        .read_addr_1(id_inst[19:15]),
        .read_addr_2(id_inst[24:20]),
        .write_addr(wb_inst[11:7]),
        .write_data(id_result),
        .read_data_1(data_1),
        .read_data_2(data_2)
    );
    ImmGen lig(
        .inst(id_inst),
        .imm(data_3)
    );
    always @ (posedge clk) begin
        exe_data1<=data_1;
        exe_data2<=data_2;
        exe_data3<=data_3;
        exe_inst<=id_inst;
        exe_pc_out<=id_pc_out;
        exe_CF<= CF;
    end
endmodule
```

EXE:

```
module EXE(
    input clk,
    input [31:0] exe_inst,
    input [31:0] exe_pc_out,
    input [12:0] exe_CF,
    input [31:0] exe_data1,
    input [31:0] exe_data2,
    input [31:0] exe_data3,
    output reg[31:0] mem_result,
    output reg[31:0] mem_data,
    output reg[31:0] mem_addr,
    output reg[31:0] mem_inst,
    output reg[31:0] mem_pc_out,
    output reg[12:0] mem_CF,
    output reg[31:0] pc_j
    );
    wire signed [31:0] sign_data_1,sign_data_2;
    wire [31:0] read_data_1,read_data_2;
    wire zf;
    assign sign_data_1=exe_data1;
    assign sign_data_2=(exe_CF[2]==0)?exe_data2:exe_data3;
    assign read_data_1=exe_data1;
    assign read_data_2=(exe_CF[2]==0)?exe_data2:exe_data3;
    `include "AluOp.vh"
    reg [31:0] ad;
    always @*
        case (exe_CF[12:9])
        ADD:  ad <= read_data_1 + read_data_2 ;
        SUB:  ad <= read_data_1 - read_data_2;
        SLT:  ad <= (sign_data_1<sign_data_2)? 1:0;
        XOR:  ad <= read_data_1 ^read_data_2;
```

```
        ADD:  ad <= read_data_1 + read_data_2 ;
        SUB:  ad <= read_data_1 - read_data_2;
        SLT:  ad <= (sign_data_1<sign_data_2)? 1:0;
        XOR:  ad <= read_data_1 ^read_data_2;
        AND:  ad <= read_data_1&read_data_2;
        OR :  ad <= read_data_1|read_data_2;
        SRL:  ad <= read_data_1>>read_data_2;
        SRA:  ad <= sign_data_1>>sign_data_2;
        SLL:  ad <= read_data_1<<read_data_2;
    endcase
    assign zf=(exe_data1==exe_data2)? 1:0;
    reg [31:0] pc;
    always @* begin
        if(exe_CF[8:7]==2'b00) pc=32'b0;
        else if(exe_CF[8:7]==2'b01) pc=exe_data1;
        else if(exe_CF[1]==1'b0)   pc=exe_pc_out+exe_data3;
        else if(exe_CF[0]^zf==1'b1) pc=exe_pc_out+exe_data3;
        else pc=32'b0;
    end
    always @ (posedge clk) begin
        mem_result<=ad;
        mem_data<=exe_data3;
        mem_addr<=exe_data2;
        mem_inst<=exe_inst;
        mem_pc_out<=exe_pc_out;
        mem_CF<=exe_CF;
        pc_j<=pc;
    end
endmodule
```

MEM:

```
module MEM(
    input clk,
    input [12:0] mem_CF,
    input [31:0] mem_inst,
    input [31:0] mem_pc_out,
    input [31:0] mem_result,
    input [31:0] mem_addr,
    input [31:0] mem_data,
    output reg[31:0] wb_data,
    output reg[31:0] wb_result,
    output reg[31:0] wb_imm,
    output reg[31:0] wb_inst,
    output reg[31:0] wb_pc_out,
    output reg[12:0] wb_CF
    );
    wire [31:0] mem_data_out;
    Ram ram_unit (
        .clka(clk),
        .wea(mem_CF[4]),
        .addra(mem_result),
        .dina(mem_addr),
        .douta(mem_data_out)
    );
    always @ (posedge clk) begin
        wb_data<=mem_data_out;
        wb_result<=mem_result;
        wb_imm<=mem_data;
        wb_inst<=mem_inst;
        wb_pc_out<=mem_pc_out;
        wb_CF<=mem_CF;
    end
```

WB:

```
module WB(
    input clk,
    input rst,
    input [12:0] wb_CF,
    input [31:0] wb_inst,
    input [31:0] wb_pc_out,
    input [31:0] wb_result,
    input [31:0] wb_data,
    input [31:0] wb_imm,
    output reg[31:0] id_result
    );
    reg [31:0] mem_to_reg;
    always @* begin
        case(wb_CF[6:5])
            2'b00:mem_to_reg = wb_result;
            2'b01:mem_to_reg = wb_imm;
            2'b10:mem_to_reg = wb_pc_out+4;
            2'b11:mem_to_reg = wb_data;
        endcase
    end
    always @ (posedge clk) begin
        id_result<=mem_to_reg;
    end
endmodule
```

三、 主要仪器设备

Vivado 2020.1
FPGA 开发板

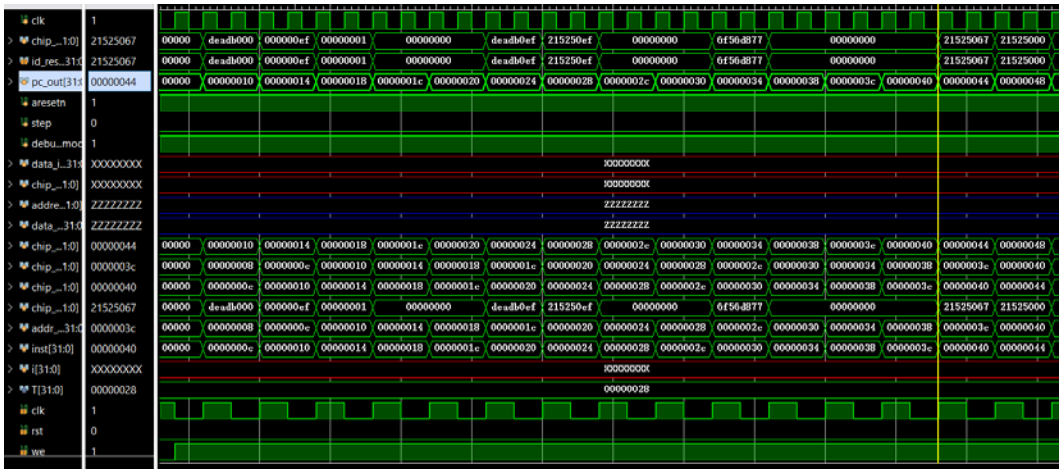
四、 操作方法与实验步骤

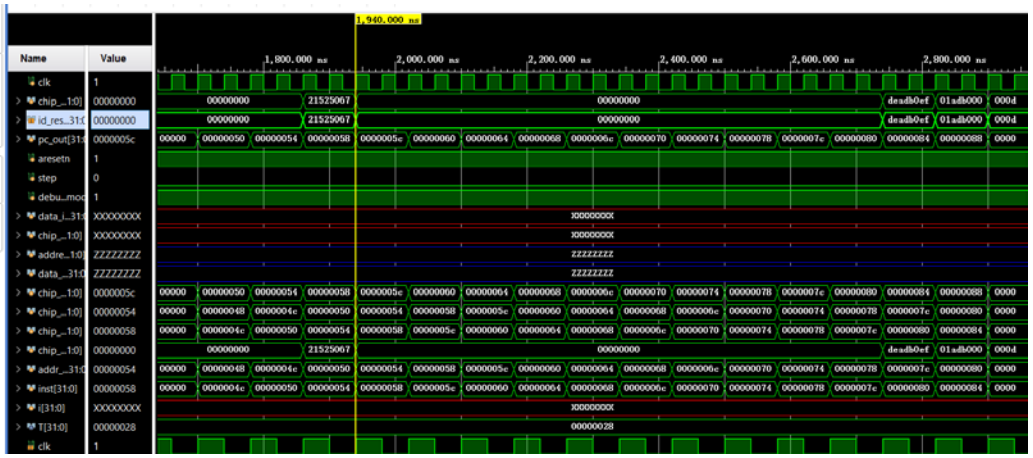
4.1 操作方法

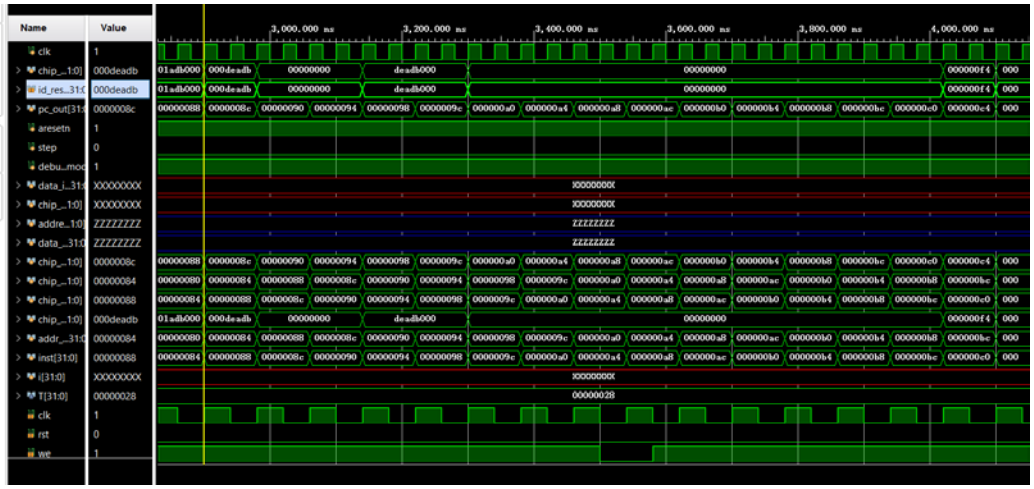
4.2 实验步骤

- 使用开发工具建立工程，推荐在 Vivado 2019.2 以上版本完成。
- 根据参考设计图或自己设计的设计图搭建完整的流水线加法机,继续使用 Nexys7 对应的外设进行实验，实现 lab1 的流水加法机替换 lab0 中的 CPU 模块。请注意 Memory 必须使用 Vivado 提供的 Block RAM IP 核完成，以免 LUT 资源不够支持 CPU 设计。
- 在本实验中不要求对数据通路进行专门的封装，推荐直接将 Control Unit (Decoder) 视为译码模块放在 ID 段中。
- 进行仿真测试，以检验 CPU 基本功能。
- 调整时钟频率，确保时钟周期长度不小于最长流水段的信号延迟。
- 进行上板测试，以检验 CPU 设计规范，上板测试调试工具相关内容请参考 lab0。

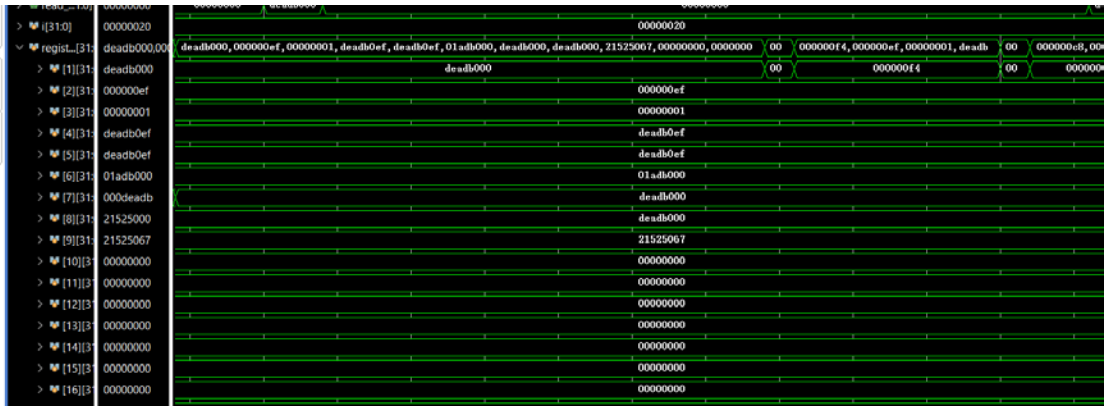
五、 实验结果与分析







如波形图所示，ID_RESULT 和 DEBUG_DATA 显示的为写入寄存器的值，而寄存器的值如下图所示：



六、 讨论、心得

Lab1-1:

单周期 CPU：CPI = 1；

流水线 CPU：CPI = 16/12 ≈ 1.33

流水线 CPI > 单周期 CPI，说明流水线 CPU 处理一条指令需要多个时钟周期进行运算执行，在最后一条指令进行 IF 阶段后还需要一段时间才可以结束。NOP 指令的意义是提供给

流水线 CPU 时间，专门处理当前指令，从而避免发生冲突。

Lab1-2:

NOP 指令数足够，不存在出现冲突且没有给足 NOP 的情况，而根据 lab1-2.s 的代码，所计算出的 NOP 恰好为现今 NOP 的分布及其数量。