合肥工業大學

2023~2024 学年 第一学期

《系统硬件综合设计》

设计报告

班 级 _计科 21-4 班_

学 号 _2021217189_

姓 名 _刘辰驭_

2024 年 1 月 5 日

目　录

目录

1 设计要求

1.1 课程设计简介

基于先修课程，根据系统设计的思想，使用 Verilog HDL 语言完成一款基于MIPS32或RISC-V或ARM等精简指令集架构的多周期CPU或流水线CPU的设计，并将设计的 CPU 下载至 FPGA 芯片上，在开发板上可以运行测试程序。设计难度为递进式，所有学生必须至少完成一个单周期 CPU 的设计工作。该课程设计贯穿了数字逻辑、计算机组成原理、计算机体系结构课程，实现从逻辑门至完整 CPU 处理器的设计。

1.2 课程设计题目要求

基于 MIPS32 或 RISC-V 或 ARM 等精简指令集架构的多周期流水线 CPU 的设计，所设计的各类指令条数不少于 25 条，其中应当包含乘除法指令，对于指令执行时可能产生的冒险与冲突，能够采取相应的方法合理解决，所设计的结构可以下载至FPGA芯片上，并在开发板上可以运行自己设计的测试程序并验证所有设计的指令。（优）

1.3 课程设计开发环境

软件环境：仿真软件：VIVADO 2017

设计语言：Verilog HDL

硬件环境：EGO1 开发板

2 设计思路

2.1MIPS32 架构

2.1.1MIPS32 架构简介

（1） MIPS32 指令：MIPS32 是一种精简指令集计算机（RISC）架构，它遵循 RISC 的设计原则，采用定长指令格式，每条指令长度为 32 位。

（2） MIPS32 寻址：MIPS32 使用 32 位寻址，每个内存地址可以寻址的数据单元大小为 32 位。

（3） MIPS32 寄存器：MIPS32 架构提供了 32 个通用寄存器（GPR），每个寄存器的大小为 32 位。这些寄存器可以用于存储数据、地址和临时结果等。

2.1.2 MIPS32 指令集

（1） MIPS32 指令集分类

MIPS32 架构中的所有指令都是 32 位，有三种指令格式，分别是 R 类型，I 类型和 J 类型指令，如图 1 所示。其中 op 是指令码、func 是功能码。

R 类型：具体操作由 op、func 结合指定，rs 和 rt 是源寄存器的编号，rd 是目的寄存器的编号，比如：假设目的寄存器是$3，那么对应的 rd 就是 00011（此处是二进制）。MIPS32 架构中有 32 个通用寄存器，使用 5 位编码就可以全部表示，所以 rs、rt、rd 的宽度都是 5 位。sa 只有在移位指令中使用，用来指定移位位数。

I 类型：具体操作由 op 指定，指令的低 16 位是立即数，运算时要将其扩展至 32 位，然后作为其中一个源操作数参与运算。

J 类型：具体操作由 op 指定，一般是跳转指令，低 26 位是字地址，用于产生跳转的目标地址。

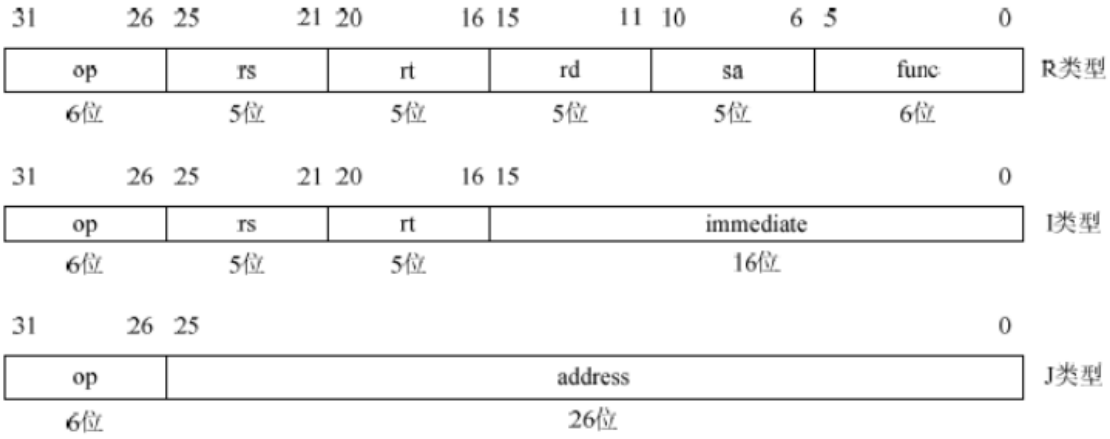


图 1：MIPS32 指令集分类

（2） 指令集

本课设主要实现了以下两类指令：

1、逻辑操作指令

有 8 条指令：and、andi、or、ori、xor、xori、nor、lui，实现逻辑与、或、异或、或非等运算。

2、移位操作指令

有 6 条指令：sll、sllv、sra、srav、srl、srlv。实现逻辑左移、右移、算术右移等运算。

3、算术操作指令

有 21 条指令：add、addi、addiu、addu、sub、subu、clo、clz、slt、slti、sltiu、sltu、mul、mult、multu、madd、maddu、msub、msubu、div、divu，实现了加法、减法、比较、乘法、乘累加、除法等运算。

2.1.3 MIPS32 寻址方式

MIPS32 架构的寻址模式有寄存器寻址、立即数寻址、寄存器相对寻址和 PC 相对寻址四种。本课设根据设计的指令格式主要实现了寄存器寻址和立即数寻址两种寻址方式。

（1）寄存器寻址：寄存器寻址的操作数直接存储在寄存器中。

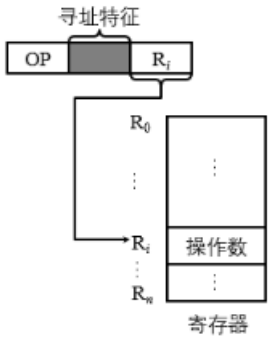


图 2：寄存器寻址

（2）立即数寻址：立即数寻址的操作数作为指令本身的一部分直接提供。

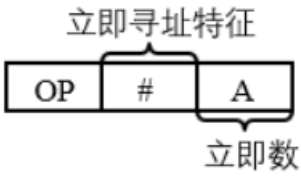


图 3：立即数寻址

2.1.4 MIPS32 寄存器

MIPS32 的寄存器分为两类：

（1） 通用寄存器

MIPS32 架构定义了 32 个通用寄存器，使用$0、$1……$31 表示，都是 32 位。其中$0 一般用做常量 0，如表 1 所示。

表 1 MIPS 通用寄存器

寄存器名	约定命名	用途
$0	zero	总是为 0
$1	at	留作汇编器生成一些合成指令
$2 $3	v0 v1	用于存放子程序的返回值
$4-7	a0-a3	调用子程序时，使用者 4 个寄存器传输前 4 个非浮点参数
$8-15	t0-t7	临时寄存器
$16-23	s0-s7	子程序寄存器变量
$24 $25	t8 t9	临时寄存器
$26 $27	$k0 $k1	由异常处理程序使用
$28	gp	全局指针
$29	sp	堆栈指针
$30	s8/fp	子程序可以用来作为堆栈帧指针
$31	ra	存放子程序返回地址

（2） 特殊寄存器

MIPS32 架构中定义的特殊寄存器有三个：PC（Program Counter 程

序计数器）、HI（乘除结果高位寄存器）、LO（乘除结果低位寄存器）。进行乘法运算时，HI 和 LO 保存乘法运算的结果，其中 HI 存储高 32 位，LO 存储低 32 位；进行除法运算时，HI 和 LO 保存除法运算的结果，其中 HI 存储余数，LO 存储商。

表 2 MIPS 专用寄存器

PC	程序计数器
HI	乘除结果高位寄存器
LO	乘除结果低位寄存器

2.2 五级流水线 CPU 模块设计

CPU 由多个模块构成，包括控制单元（Control Unit）、算术逻辑单元（Arithmetic Logic Unit, ALU）、寄存器（Registers）、流水线（Pipeline）和数据前推模块（Data Forward）、时钟模块（Timing Unit）等。

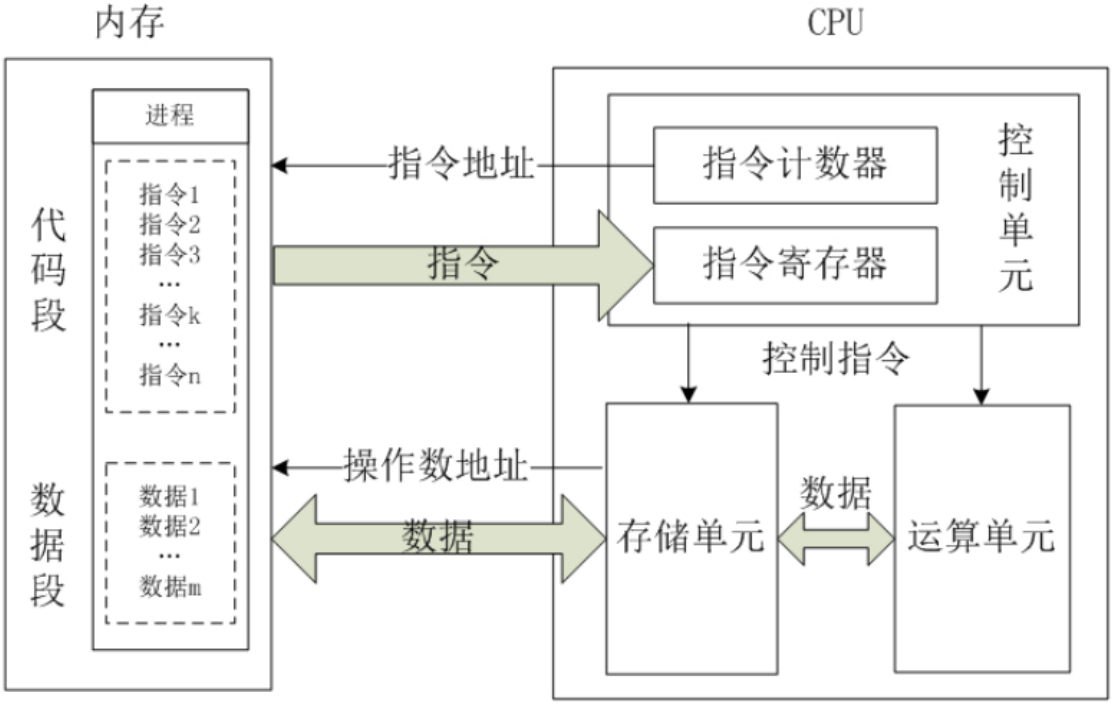


图 4：CPU 结构图

2.2.1 控制单元模块 CU 设计

控制单元负责解析和执行指令。它从存储器中读取指令，解码指令中的操作码，并生成相应的控制信号来控制其他模块的操作。

下面是控制单元解析和执行指令的基本步骤：

（1） 指令获取：控制单元从存储器中获取指令。指令通常存储在主存储器中，控制单元通过地址总线发送指令地址，并使用数据总线接收指令内容。

```
//func: 根据地址取出指令
always @ (*) begin
    if (ce == `ChipDisable) begin
        inst <= `ZeroWord;
    end else begin
        inst <= inst_mem[addr[`InstMemNumLog2+1:2]];
    end
end
```

图 5：部分代码，从指令存储器中取出指令

（2） 指令解码：控制单元解码指令中的操作码和操作数。它识别指令的类型和要执行的操作，并生成相应的控制信号。

```
wire[5:0] op = inst_i[31:26];
wire[4:0] op2 = inst_i[10:6];
wire[5:0] op3 = inst_i[5:0];
wire[4:0] op4 = inst_i[20:16];
```

图 6：部分代码，译码

（3） 控制信号生成：根据指令的操作码和操作数，控制单元生成相应的控制信号。这些信号包括 ALU 的操作选择信号、寄存器的读写控制信号、总线的控制信号等。

```
aluop_o <= `EXE_NOP_OP;     //运算子类型初始化为NOP
alusel_o <= `EXE_RES_NOP;   //运算类型初始化为NOP
wd_o <= inst_i[15:11];      //默认[15:11],即rd为写地址(不同类型指令写地址不同,在指令实现的时候改)
wreg_o <= `WriteDisable;    //默认“写使能”为0，不许写
instvalid <= `InstInvalid;  //默认指令有效
reg1_read_o <= 1'b0;        //默认端口1“读使能”为0，不许读
reg2_read_o <= 1'b0;        //默认端口2“读使能”为0，不许读
reg1_addr_o <= inst_i[25:21];   //默认端口1“读地址”为[25:21](即R类时的rs)
reg2_addr_o <= inst_i[20:16];   //默认端口2“读地址”为[20:16](即R类时的rt)
imm <= `ZeroWord;
```

图 7：部分代码，控制信号生成

2.2.2 算术逻辑单元 ALU 设计

ALU 的设计包含以下两种运算，均使用 Verilog HDL 自带编程语言的运算语句实现。

（1） 算术操作：ALU 可以执行各种算术操作，如加法、减法、乘法和除法。控制信号中的操作类型指定了要执行的具体算术操作。

（2） 逻辑操作：ALU 还可以执行逻辑操作，如逻辑与、逻辑或、逻辑非和位移操作。控制信号中的操作类型指定了要执行的逻辑操作。

2.2.3 寄存器模块 REG 设计

本课设设计了 32 个通用寄存器（参考表 1 和表 2 的寄存器功能），用于临时存储和处理数据。

```
//关键：定义32个32位寄存器
//RegNum=32
reg[`RegBus]  regs[0:`RegNum-1];
```

图 8：32 个通用寄存器 regs[0:31]

2.2.4 五级流水线设计

本课设的 CPU 流水线设计采用了五级流水线设计，如图 9 所示

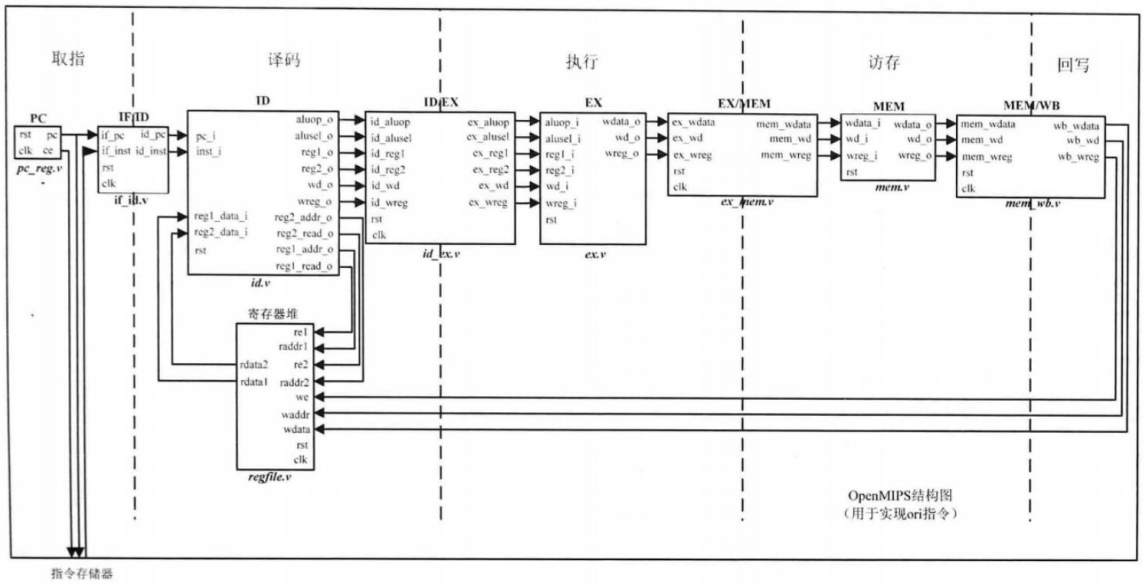


图 9：CPU 五级流水结构图

（1） 取指（IF）：该阶段负责从指令存储器中获取指令，并将其送入流水线。

PC 模块：包含指令指针寄存器 PC，用于存储指令地址。

IF/ID 模块：将取指阶段的结果传递到译码阶段。

（2） 译码（ID）：在这一阶段，指令的操作码和操作数被解码，并生成相应的控制信号。同时，源操作数寄存器中的数据被读取。

ID 模块：根据指令的机器码确定指令类型，并获取操作数和控制信号。

Register File 模块：包含 32 个通用寄存器，提供操作数和写入数据的功能。

ID/EX 模块：将译码阶段的数据传递到执行阶段。

（3） 执行（EX）：在执行阶段，执行算术或逻辑操作。例如，加法、减法、逻辑与、逻辑或等操作被执行。结果存储在目标寄存器中。

EX 模块：根据译码阶段的指令翻译结果使用源操作数进行计算。

EX/MEM 模块：将执行阶段的数据传递到访存阶段。

（4） 访存（MEM）：在此阶段，如果指令涉及访问内存数据（如加载或存储指令），则内存访问操作被执行。这可能涉及读取或写入数据到数据存储器中。

MEM/WB 模块：将访存阶段的数据传递到写回阶段。

（5） 写回（WB）：在最后一个阶段，结果被写回到目标寄存器。这个阶段还包括将结果存储到数据存储器中。

HILO 模块：用于实现乘法和除法运算，其中 HI 寄存器存储乘法指令的低位结果或除法指令的商，LO 寄存器保存乘法指令的高位结果或除法指令的余数。

2.2.5 数据前推模块设计

在本课设采用流水线技术设计 CPU 后，可能会出现数据相关问题。

数据冲突分类：

（1）读后写冲突（RAW）：

如果一条指令在读取寄存器的同时，后面的指令试图写入相同的寄存器，就会发生读后写冲突。这种情况下，后面的指令需要等待前面的指令完成写操作后才能读取正确的数据。

（2）写后读冲突（WAR）：

如果一条指令在写入寄存器的同时，后面的指令试图读取相同的寄存器，就会发生写后读冲突。这种情况下，后面的指令可能读取到前面指令未完成写入的数据，导致错误的结果。

（3）写后写冲突（WAW）：

如果两条指令同时试图写入相同的寄存器，就会发生写后写冲突。这种情况下，后面的指令需要等待前面的指令完成写操作后才能进行写入，以保证数据的一致性。

数据前推模块是用于解决数据相关问题的关键组成部分。数据前推模块通过将数据从执行阶段提前传递到后续阶段，以使得依赖该数据的指令能够在下一个时钟周期中正确执行，而无需等待结果写回到寄存器文件。

数据前推出现的阶段：

（1） 执行阶段（EX）到译码阶段（ID）的前推：

当前指令的执行阶段（EX）产生的结果需要在下一个指令的译码阶

段（ID）使用时，可以通过数据前推将结果提前传递给译码阶段。这种情况通常出现在指令之间存在数据相关的情况下，例如前一条指令的结果是后一条指令的操作数。

```
always @ (*) begin
    if(rst == `RstEnable) begin
        reg2_o <= `ZeroWord;
    //数据前推：如果id段要读，此时ex段要写，并且二者地址相同，则可以直接得到数据，
    //ex在前因为根据流水线，ex段的数据要比mem段的“新”
    end else if((reg2_read_o == 1'b1) && (ex_wreg_i == 1'b1) && (ex_wd_i == reg2_addr_o)) begin
        reg2_o <= ex_wdata_i;
    //数据前推：如果id段要读，此时mem段要写，并且二者地址相同，则可以直接得到数据
    end else if((reg2_read_o == 1'b1) && (mem_wreg_i == 1'b1) && (mem_wd_i == reg2_addr_o)) begin
        reg2_o <= mem_wdata_i;
    //如果可读，从指定寄存器中读取取数据
    end else if(reg2_read_o == 1'b1) begin
        reg2_o <= reg2_data_i;
    //不可读赋值为立即数
    end else if(reg2_read_o == 1'b0) begin
        reg2_o <= imm;
    end else begin //否则为0
        reg2_o <= `ZeroWord;
    end
```

图 10：部分代码，ID 段的数据前推

（2） 执行阶段（EX）到访存阶段（MEM）的前推：

某些指令需要在访存阶段（MEM）访问数据存储器，并且需要使用执行阶段（EX）产生的结果作为访存的地址或数据。如果后续指令正好需要使用该数据，可以通过数据前推将执行阶段的结果提前传递给访存阶段，以供后续指令使用。

```
always @ (*) begin
    if(rst == `RstEnable) begin //复位
        {HI,LO} <= {`ZeroWord,`ZeroWord};
    end else if(mem_whilo_i == `WriteEnable) begin //mem段要写hilo，则将mem段要写的数据写入到hilo中
        {HI,LO} <= {mem_hi_i,mem_lo_i};
    end else if(wb_whilo_i == `WriteEnable) begin //wb要写hilo，则将mem段要写的数据写入到hilo中
        {HI,LO} <= {wb_hi_i,wb_lo_i};
    end else begin  //都没有写，那我可以直接更新计算的数据
        {HI,LO} <= {hi_i,lo_i};
    end
end
```

图 11：部分代码，EX 段的数据前推

2.2.6 时钟模块设计

本课设的时钟模块采用计数器循环方式实现，利用一个寄存器记录原始时钟上升沿的次数，并使用一个名为 clk_o 的寄存器作为输出。初始情况下，clk_o 被设置为 0。当原始时钟上升沿的次数达到 10000000 次时，clk_o 的状态将被反转，然后将 clk_o 作为新的时钟信号。通过这种方式，可以在硬件级别上降低时钟频率。

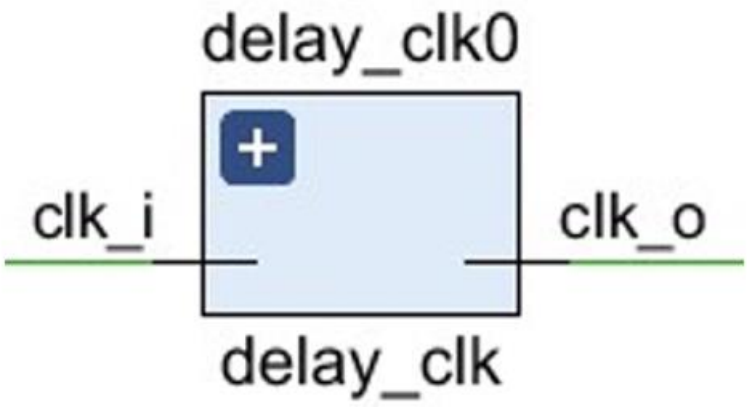


图 12：CLK 模块设计图

```
always @ (posedge clk_i) begin
    //if(count >= 32'd30000) begin
    if(count > 32'd13500000) begin
        count <=  32'd0;
        clk_o <= ~clk_o;
    end else begin
        count <= count + 1'd1;
    end
```

图 13：部分代码，CLK 模块

3　设计框架与实现

3.1 CPU 五级流水线结构设计

3.1.1 IF 段结构设计

IF 段的主要功能是从指令存储器中获取下一条要执行的指令，并将其送入流水线中进行后续的处理。

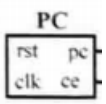


图 14：IF 段 PC 模块结构

IF 段 PC 模块接口设计：

表 3：PC 模块接口

clk	时钟信号
rst	复位信号
pc	指令指针寄存器
ce	指令存储器使能

本课设在 IF 段的功能设计包括以下两个方面：

（1） 指令获取：IF 段从指令存储器 inst_room 中读取下一条要执行的指令。它根据程序计数器 PC 中保存的地址信息，将指令从存储器中提取出来。

```
always @ (*) begin
    if (ce == `ChipDisable) begin
        inst <= `ZeroWord;
    end else begin
        inst <= inst_mem[addr[`InstMemNumLog2+1:2]];
    end
end
```

图 15：部分代码，取指

（2） PC 更新：在 IF 段结束时，需要更新程序计数器的值，以使其指向下一条要执行的指令的地址。这通常是通过将当前 PC 的值加上指令的长度

来实现。

```
always @ (posedge clk) begin
    if (ce == `ChipDisable) begin //禁用指令存储器时
            pc <= `ZeroWord;  //pc=0
        end else begin
            pc <= pc + 4'h4; //否则指向下一条指令pc+=4
        end
end
```

图 16：部分代码，更新 pc 值

3.1.2 ID 段结构设计

ID 段是指令执行流水线中的第二个阶段，也被称为译码阶段。ID 段的主要功能是对从 IF 段获取的指令进行解码并提取出指令中的操作数和相关信息，为后续的执行阶段做准备。

在本设计中 ID 段包括了 ID 模块和 regfile 寄存器模块。

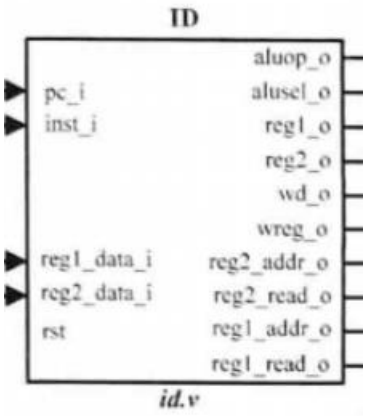


图 17：ID 段结构

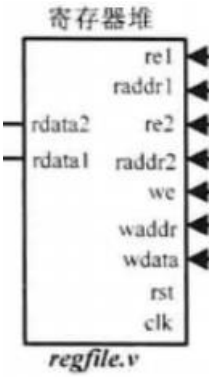


图 18：regfile 模块结构

ID 段接口设计：

表 4：ID 段接口

rst	复位信号
pc_i	取得的指令
inst_i	取得的指令
ex_wreg_i	ex 段是否要“写”
ex_wdata_i	ex 段写“数据”
ex_wd_i,	ex 段写“地址”
mem_wreg_i	mem 段是否要“写”
mem_wdata_i	mem 段写“数据”
mem_wd_i	mem 段写“地址”
reg1_data_i	端口 1 读取的“数据”寄存器
reg2_data_i	端口 2 读取的“数据”寄存器
reg1_read_o	端口 1 读取“使能”
reg2_read_o	端口 2 读取“使能”
reg1_addr_o	端口 1 读取“地址”
reg2_addr_o	端口 2 读取“地址”
aluop_o	运算子类型
alusel_o	运算类型
reg1_o	源操作数 1
reg2_o	源操作数 2
wd_o	写地址
wreg_o	是否有写的目的寄存器

本课设在 ID 段的功能设计包括以下三个方面：

（1） 指令解码：ID 段负责解码从 IF 段获取的指令。它识别指令的类型、操作码和操作数的位置等信息，并将其转化为内部的控制信号和操作数。

（2） 寄存器读取：ID 段根据指令中的寄存器编号，从寄存器文件（regfile）中读取相应的操作数值。这些操作数值将用于后续的指令执行阶段。

（3） 立即数提取：对于包含立即数（Imm）的指令，ID 段将立即数从指令中提取出来，并传递给执行阶段使用。

```
always @ (*) begin
    if(rst == `RstEnable) begin //复位
            rdata2 <= `ZeroWord;
    end else if(raddr2 == `RegNumLog2'h0) begin //读$0寄存器，给0
        rdata2 <= `ZeroWord;
    //数据前推：“读的地址”和“写的地址”刚好相同，并且此时cpu既要“读”也要“写”，则直接得到数据
    end else if((raddr2 == waddr) && (we == `WriteEnable) && (re2 == `ReadEnable)) begin
        rdata2 <= wdata;
    //无法前推则从寄存器中取数据
    end else if(re2 == `ReadEnable) begin
        rdata2 <= regs[raddr2];
    //其他情况赋值为0
    end else begin
        rdata2 <= `ZeroWord;
    end
end
```

图 19：部分代码，寄存器读取（采用数据前推）

3.1.3 EX 段结构设计

EX 段是指令执行流水线中的第三个阶段，也被称为执行阶段。EX 段的主要功能是执行指令中的操作，并进行算术逻辑运算、数据传输或逻辑判断等操作。

本设计的 EX 模块采用多个组合逻辑电路设计，以实现并行处理不同类型的运算。它通过设置中间变量来存储各类运算的结果，并根据指令的操作类型选择最终结果。

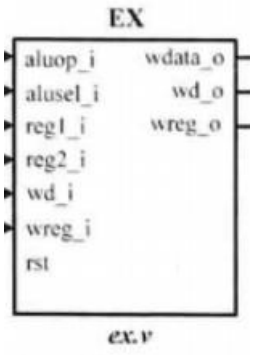


图 20：EX 段结构

EX 段接口设计：

表 5：EX 段接口

rst	复位信号

aluop_i	运算子类
alusel_i	运算主类
reg1_i	ex 段接收到的源操作数 1
reg2_i	ex 段接收到的源操作数 2
wd_i	ex 段接收到的写地址
wreg_i	ex 段接收到的写使能
hi_i	HI 寄存器
lo_i	LO 寄存器
wb_hi_i	wb 段写 hi 的值
wb_lo_i	wb 段写 lo 的值
wb_whilo_i	wb 段是否要写 hilo
mem_hi_i	mem 段写 hi 的值
mem_lo_i	mem 段写 lo 的值
mem_whilo_i	mem 段是否要写 hilo
wd_o	ex 段“写地址”
wreg_o	ex 段“写使能”
wdata_o	ex 段“写数据”
hi_o	ex 段写 hi 的值
lo_o	ex 段写 lo 的值
whilo_o	ex 段是否要写 hilo

EX 段主要功能如下：

用于指令的执行，主要功能为执行包括加法、减法、乘法、除法等算术运算，以及位操作、逻辑运算和比较操作等逻辑运算。根据指令的类型和操作码，EX 段根据操作数执行相应的运算。

本课设为该 CPU 设计了共 37 种类型的指令，其中包括：

（1） R 类：OR、AND、XOR、NOR、SLLV、SRLV、SRAV、MFHI、MFLO、MTHI、MTLO、MOVN、MOVZ、SLT、SLTU、ADD、ADDU、SUB、SUBU、MULT、MULTU、DIV,DIVU 共 23 种

（2） I 类：ORI、ANDI、XORI、LUI、SLTI、SLTU、ADDI、ADDIU 共 8 种

（3） SPECIAL2：CLZ、CLO、MUL 共 3 种

（4） 移位运算：SLL、SRL、SRA 共 3 种

```
//func:算数运算
always @ (*) begin
    if(rst == `RstEnable) begin //复位
        arithmeticres <= `ZeroWord;
    end else begin
        case (aluop_i)
            `EXE_SLT_OP, `EXE_SLTU_OP:      begin //r1 < r2
                arithmeticres <= reg1_lt_reg2 ;
                end
```

图 21：部分代码，EX 段执行算术运算

3.1.4 MEM 段结构设计

MEM 段是指令执行流水线中的第四个阶段，也被称为存储器访问阶段。MEM 段的主要功能是处理与内存相关的操作，包括数据的读取和写入。

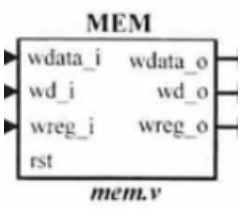


图 22：MEM 段结构

MEM 段接口设计：

表 6：MEM 段接口

rst	复位信号
wd_i	写地址
wreg_i	写使能
wdata_i	写数据
hi_i	hi 数据
lo_i	lo 数据
whilo_i	hilo 写使能
wd_o	写地址
wreg_o	写使能
wdata_o	写数据
hi_o	hi 数据
lo_o	lo 数据
whilo_o	hilo 写使能

MEM 段主要设计有以下三种功能：

（1） 数据存储器访问：如果指令需要读取或写入内存中的数据，MEM 段负责与主存储器进行数据交互。它通过计算内存地址并发送相应的读取或写入信号来实现数据的存取。

（2） 数据读取：对于读取指令，MEM 段从主存储器中读取相应的数据。它根据地址计算结果和读取信号，将数据传递给下一个阶段以供使用。

（3） 数据写入：对于写入指令，MEM 段负责将数据写入主存储器的指定地址。它根据地址计算结果和写入信号，将数据写入内存。

3.1.5 WB 段结构设计

WB 段是指令执行流水线中的最后一个阶段，也被称为写回阶段。WB 段的主要功能是将执行阶段的结果写回寄存器文件（regfile）或其他存储位置。

在本课设中 WB 段主要包含在 regfile 模块。

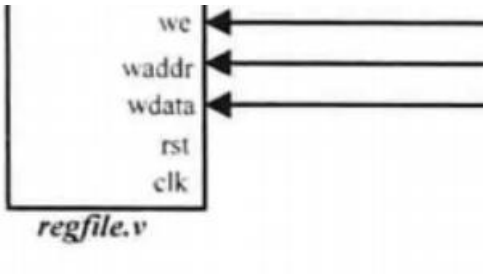


图 23：WB 段写回寄存器 regfile

本课设设计的 WB 段的主要功能如下：

寄存器写回：如果指令的结果需要写回到寄存器中，WB 段负责将执行阶段计算得到的结果写入到寄存器文件的相应位置。这样，指令执行的结果将被存储在寄存器中，供后续的指令使用。

```
//func:给指定寄存器写入数据
always @(posedge clk) begin
    if(rst == `RstDisable) begin //不复位
    //写数据 且 写的不是0寄存器（0寄存器值为零，不允许更改）
        if((we ==   `WriteEnable) && (waddr != `RegNumLog2'h0)) begin
            regs[waddr] <= wdata;
        end

    end
end
```

图 24：部分代码，WB 段寄存器写回

3.2 时序同步结构设计

在 CPU 实现了流水线各个功能之后，需要使用时序逻辑电路将各个模块按照相同有规律的时序，在每一个时钟的上升沿进行数据与控制信号的传递，保证流水线的有序性。

3.2.1IF_ID 段结构设计

采用时序逻辑电路，在每一个时钟上升沿将 IF 段的数据与控制信号传递给 ID 段，实现 IF 段到 ID 段的时钟同步。

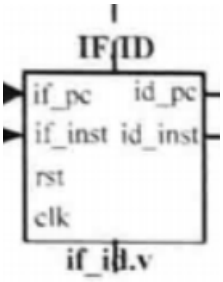


图 25：IF_ID 段结构

本课设设计如下 IF_ID 段接口：

表 7：IF_ID 段接口

clk	时钟信号
rst	复位信号
if_pc	if 段来的 pc

if_inst	if 段来的指令
id_pc	传给 id 段的 pc
id_inst	传给 id 段的指令

```
//时序电路
always @(posedge clk)begin
    //先考虑是否复位
    if(rst == `RstEnable)begin //复位
        id_pc <= `ZeroWord; //0
        id_inst <= `ZeroWord; //0
    end else begin
        // 将if段的数据传递给id段
        id_pc <= if_pc;
        id_inst <= if_inst;
    end
end
```

图 26：部分代码，IF_ID 时钟同步

3.2.2ID_EX 段结构设计

采用时序逻辑电路，在每一个时钟上升沿将 ID 段的数据与控制信号传递给 EX 段，实现 ID 段到 EX 段的时钟同步。

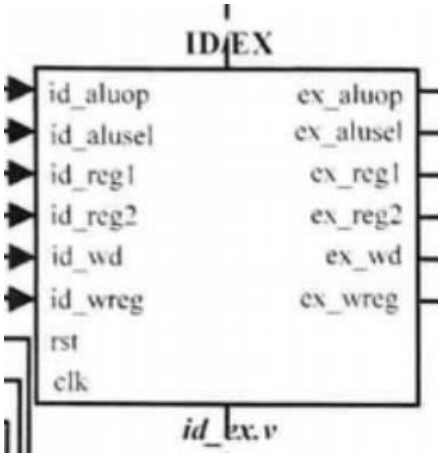


图 27：ID_EX 段结构

本课设设计如下 ID_EX 段接口：

表 8：ID_EX 段接口

clk	时钟信号
rst	复位信号
id_aluop	id 段传来的运算子类
id_alusel	id 段传来的运算主类
id_reg1	id 段传来的源操作数 1
id_reg2	id 段传来的源操作数 2
id_wd	id 段传来的写地址
id_wreg	id 段传来的是否有写的目的寄存器
ex_aluop	传向 ex 段的运算子类
ex_alusel	传向 ex 段的运算主类
ex_reg1	传向 ex 段的源操作数 1
ex_reg2	传向 ex 段的源操作数 2
ex_wd	传向 ex 段的写地址
ex_wreg	传向 ex 段的是否有写的目的寄存器

```
//时序逻辑电路
//func:将id段的信号传递到ex段
always @ (posedge clk) begin
    if (rst == `RstEnable) begin //复位
        ex_aluop <= `EXE_NOP_OP;
        ex_alusel <= `EXE_RES_NOP;
        ex_reg1 <= `ZeroWord;
        ex_reg2 <= `ZeroWord;
        ex_wd <= `NOPRegAddr;
        ex_wreg <= `WriteDisable;
    end else begin
        //将id段所有信号依次赋值给ex段信号即可
        ex_aluop <= id_aluop;
        ex_alusel <= id_alusel;
        ex_reg1 <= id_reg1;
        ex_reg2 <= id_reg2;
        ex_wd <= id_wd;
        ex_wreg <= id_wreg;
    end
end
```

图 28：部分代码，ID_EX 时钟同步

3.2.3EX_MEM 段结构设计

采用时序逻辑电路，在每一个时钟上升沿将 EX 段的数据与控制信号传递给 MEM 段，实现 EX 段到 MEM 段的时钟同步。

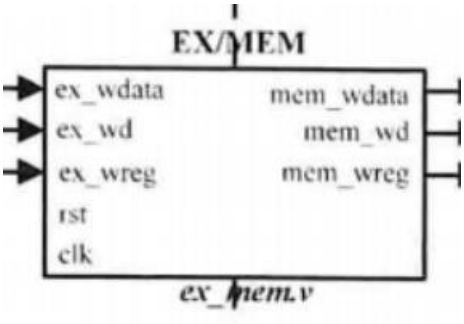


图 29：EX_MEM 段结构

本课设设计如下 EX_MEM 段接口：

表 9：EX_MEM 段接口

clk	时钟
rst	复位
ex_wd	ex 段写使能
ex_wreg	ex 段写的地址
ex_wdata	ex 段写的数据
ex_hi	ex 段 hi 的值
ex_lo	ex 段 lo 的值
ex_whilo	ex 段传来的 hilo 的写使能
mem_wd	传向 mem 段写的地址
mem_wreg	传向 mem 段写的地址
mem_wdata	传向 mem 段写的数据
mem_hi	传向 mem 段的 hi 的值
mem_lo	传向 mem 段的 lo 的值
mem_whilo	传向 mem 段写 hilo 的使能

```
//时序电路
//func: 传送ex的信号到mem段
always @(posedge clk) begin //时钟上升沿
    //如果复位则将所有数据恢复为默认值
    if(rst==`RstEnable) begin
        mem_wd <= `NOPRegAddr; //写地址位为空
        mem_wreg <= `WriteDisable;//写使能为0，即——不许写
        mem_wdata <= `ZeroWord; //写的数据是0
        mem_hi <= `ZeroWord; //写的数据是0
        mem_lo <= `ZeroWord; //写的数据是0
        mem_whilo <= `ZeroWord; //写的数据是0
    end else begin
        //将ex段的数据全部传递到mem段
        mem_wd <= ex_wd;
        mem_wreg <= ex_wreg;
        mem_wdata <= ex_wdata;
        mem_hi <= ex_hi;
        mem_lo <= ex_lo;
        mem_whilo <= ex_whilo;
    end
end
```

图 30：部分代码，EX_MEM 时钟同步

3.2.4 MEM_WB 段结构设计

采用时序逻辑电路，在每一个时钟上升沿将 MEM 段的数据与控制信号传递给 WB 段，实现 MEM 段到 WB 段的时钟同步。

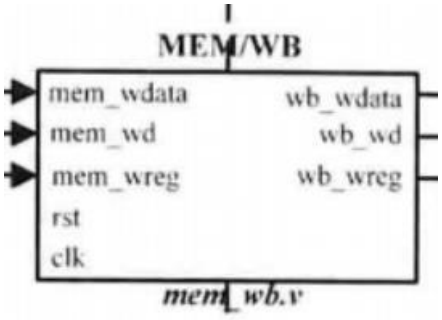


图 31：MEM_WB 段结构

本课设设计如下 MEM_WB 段接口：

表 10：MEM_WB 段接口

clk	时钟信号
rst	复位信号
mem_wd	mem 写地址
mem_wreg	mem 写使能
mem_wdata	mem 写数据
mem_hi	mem 段 hi
mem_lo	mem 段 lo
mem_whilo	mem 段写 hilo 使能
wb_wd	wb 写地址
wb_wreg	wb 写使能
wb_wdata	wb 写数据
wb_hi	wb 段 hi
wb_lo	wb 段 lo
wb_whilo	wb 段写 hilo 使能

```
//时序逻辑电路
//func
always @ (posedge clk) begin
    if(rst == `RstEnable) begin //复位
        wb_wd <= `NOPRegAddr;
        wb_wreg <= `WriteDisable;
        wb_wdata <= `ZeroWord;
        wb_hi <= `ZeroWord;
        wb_lo <= `ZeroWord;
        wb_whilo <= `WriteDisable;
    end else begin
        //将mem段的数据传送到wb段
        wb_wd <= mem_wd;
        wb_wreg <= mem_wreg;
        wb_wdata <= mem_wdata;
        wb_hi <= mem_hi;
        wb_lo <= mem_lo;
        wb_whilo <= mem_whilo;
    end
end
```

图 32：部分代码，MEM_WB 时钟同步

3.3 指令集设计

本课设为该 CPU 设计了共 37 种类型的指令，其中包括：

（1） R 类：OR、AND、XOR、NOR、SLLV、SRLV、SRAV、MFHI、MFLO、MTHI、MTLO、MOVN、MOVZ、SLT、SLTU、ADD、ADDU、SUB、SUBU、MULT、MULTU、DIV,DIVU 共 23 种

助记符	指令编码						格式	操作及其解释
Bit #	31..26	25..21	20..16	15..11	10..6	5..0		
Reg-type	Op	rs	rt	rd	shamt	func		
and	0	rs	rt	rd	0	100100	AND rd, rs, rt	rd←rs and rt
or	0	rs	rt	rd	0	100101	OR rd, rs, rt	rd←rs or rt
xor	0	rs	rt	rd	0	100110	XOR rd, rs, rt	rd←rs xor rt
nor	0	rs	rt	rd	0	100111	NOR rd, rs, rt	rd←rs nor rt
sllv	0	rs	rt	rd	0	100	SLLV rd, rs, rt	rd ← rt << rs
srlv	0	rs	rt	rd	0	110	SRLV rd, rs, rt	rd ← rt >> sa
srav	0	rs	rt	rd	0	111	SRAV rd, rs, rt	rd← rt>>sa(arithmetic)
mflo	0	0	0	rd	0	10010	MFLO rd	rd ← LO
mfhi	0	0	0	rd	0	10000	MFHI rd	rd ← HI
mtlo	0	rs	0	0	0	10011	MTLO rs	LO ← rs
mthi	0	rs	0	0	0	10001	MTHI rs	HI ← rs
movn	0	rs	rt	rd	0	1011	MOVN rd, rs, rt	if rt ≠ 0 then rd ← rs
movz	0	rs	rt	rd	0	1010	MOVZ rd, rs, rt	if rt = 0 then rd ← rs
slt	0	rs	rt	rd	0	101010	SLT rd, rs, rt	rd ← (rs < rt)
sltu	0	rs	rt	rd	0	101011	SLTU rd, rs, rt	rd ← (rs < rt)
add	0	rs	rt	rd	0	100000	ADD rd, rs, rt	rd←rs + rt
addu	0	rs	rt	rd	0	100001	ADDU rd, rs, rt	rd←rs + rt(unsigned)
sub	0	rs	rt	rd	0	100010	SUB rd, rs, rt	rd←rs - rt
subu	0	rs	rt	rd	0	100011	SUB rd, rs, rt	rd←rs - rt(unsigned)
mult	0	rs	rt	0	0	11000	MULT rs, rt	(HI, LO) ← rs × rt
multu	0	rs	rt	0	0	11001	MULTU rs, rt	(HI, LO) ← rs × rt
div	0	rs	rt	0	0	11010	DIV rs, rt	(HI, LO) ← rs / rt
divu	0	rs	rt	0	0	11011	DIV rs, rt	(HI, LO) ← rs / rt

图 33：设计的 R 型指令

（2） I 类：ORI、ANDI、XORI、LUI、SLTI、SLTU、ADDI、ADDIU 共 8 种

助记符	指令编码						格式	操作及其解释
Bit #	31..26	25..21	20..16	15..11	10..6	5..0		
Reg-type	Op	rs	rt	rd	shamt	func		
ori	1101	rs	rt	immediate			ORI rt,rs,immediate	rt ← rs or immediate
xori	1110	rs	rt	immediate			XORI rt,rs,immediate	rt ← rs xor immediate
lui	1111	0	rt	immediate			LUI rt, immediate	rt ← immediate || 0000H
slti	1010	rs	rt	immediate			SLTI rt,rs,immediate	rt ← (rs < immediate)
sltiu	1011	rs	rt	immediate			SLTI rt,rs,immediate	rt ← (rs < immediate)
addi	1000	rs	rt	immediate			ADDI rt,rs,immediate	rt ← rs + immediate
addiu	1001	rs	rt	immediate			ADDIU rt,rs,immediate	rt ← rs + immediate

图 34：设计的 I 型指令

（3） SPECIAL2：CLZ、CLO、MUL 共 3 种

助记符	指令编码						格式	操作及其解释
Bit #	31..26	25..21	20..16	15..11	10..6	5..0		
Reg-type	Op	rs	rt	rd	shamt	func		
clo	11100	rs	rt	rd	0	100001	CLO rd, rs, rt	rd← count_leading_ones rs
clz	11100	rs	rt	rd	0	100000	CLZ rd, rs, rt	rd←count_leading_zeros rs
mul	11100	rs	rt	rd	0	10	MUL rd, rs, rt	rd ← rs × rt

图 35：设计的特殊指令

（4） 移位运算：SLL、SRL、SRA 共 3 种

助记符	指令编码						格式	操作及其解释
Bit #	31..26	25..21	20..16	15..11	10..6	5..0		
Reg-type	Op	rs	rt	rd	shamt	func		
sll	0	0	rt	rd	shamt	0	SLL rd, rs, rt	rd ← rt << sa
srl	0	0	rt	rd	shamt	10	SRL rd, rs, rt	rd ← rt >> sa
sra	0	0	rt	rd	shamt	11	SRA rd, rs, rt	rd← rt>>sa(arithmetic)

图 36：设计的移位指令

4　实验结果与分析

4.1 指令运算实验结果仿真与验证（以有符号除法为例）

（1） 验证代码（汇编）：

ori $1,$0,-10　#$1 ← -10

ori $2,$0,3　　#$2 ← 3

divu $1,$2　　#HI ← $1/$2 , LO ← $1%$2

（2） 验证代码（十六进制）：

3401fff6

34020003

0022001a

（3） 仿真结果：

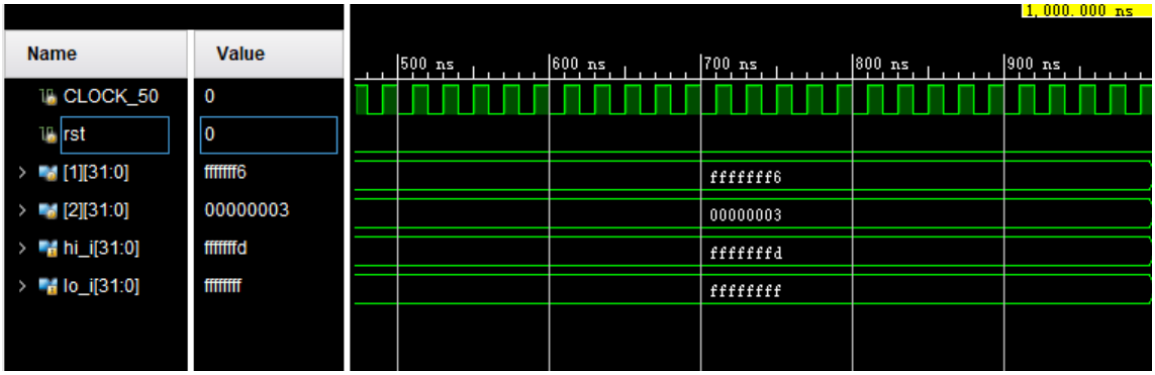


图 37：符号除法仿真结果

（4） 结果验证：

-10 的 16 进制数为 fffffff6，存在 1 号寄存器中。

3 的 16 进制数为 00000003，存在 2 号寄存器中。

-10/3 的商为-3，16 进制数为 fffffffd，存放在 hi 寄存器中。

-10/3 的余数为-1，16 进制数为 ffffffff，存放在 lo 寄存器中。

所得结果与预期相吻合，实验结果正确。

4.2 五级流水线仿真与验证

仍然采用上例有符号除代码。

（1） 验证代码（汇编）：

ori $1,$0,-10 #$1 ← -10

ori $2,$0,3 #$2 ← 3

divu $1,$2 #HI ← $1/$2，LO ← $1%$2

（2） 验证代码（十六进制）：

3401fff6

34020003

0022001a

（3） 仿真结果：

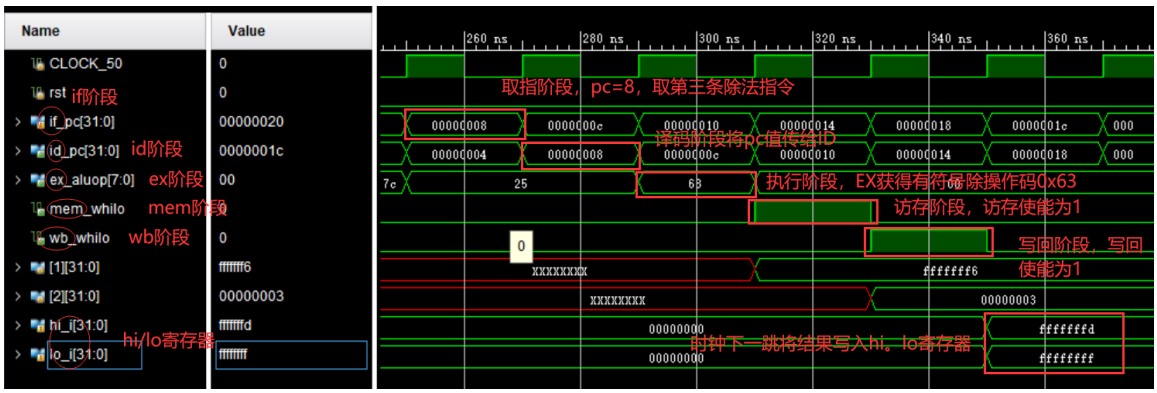


图 38：五级流水线仿真结果

（4） 结果验证：

CPU 有五级流水，并通过时钟同步机制在每次时钟的上升沿同步，与预期结果相符，五级流水设计正确。

4.3 数据相关与数据前推仿真与验证

仍然采用上例有符号除代码。

（1） 验证代码（汇编）：

```
ori $1,$0,-10    #$1 ← -10
ori $2,$0,3      #$2 ← 3
divu $1,$2     #HI ← $1/$2 , LO ← $1%$2
```

（2） 验证代码（十六进制）：

```
3401fff6
34020003
0022001a
```

（3） 仿真结果：

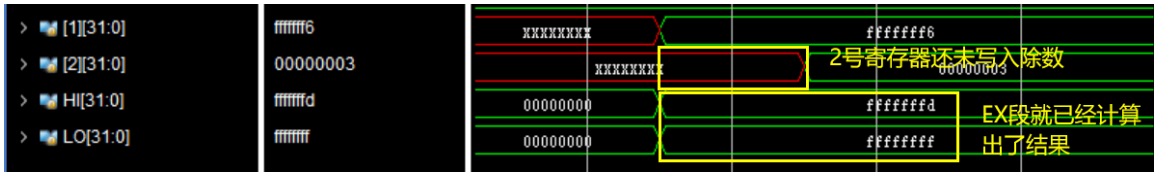


图 39：数据前推仿真结果

（4） 结果验证：

当第二条指令（ori $2,0,3）除数还未写回 2 号寄存器时第三条指令（divu $1,$2）就进入了 EX 阶段执行除法运算，出现了读后写相关。

此时发生数据前推，EX 段直接获得 WB 段写回的数据。

与预期结果相符，数据前推设计正确。

4.4 显示斐波那契数列实验仿真验证

实验目的：显示斐波那契数列 1，2，3，5，8，13，……

（1） 验证代码（汇编）：

```
#循环展开
ori $1,$0,0        #$1<--0
ori x2,$0,1        #$2<--1
addu $3,$1,$2      #$3<--$1+$2
```

```
or $1,$0,$2        #$1<--$2
or $2,$0,$3        #$2<--$3
addu $3,$1,$2      #$3<--$1+$2
or $1,$0,$2        #$1<--$2
or $2,$0,$3        #$2<--$3
addu $3,$1,$2      #$3<--$1+$2
or $1,$0,$2        #$1<--$2
or $2,$0,$3        #$2<--$3
addu $3,$1,$2      #$3<--$1+$2
or $1,$0,$2        #$1<--$2
or $2,$0,$3        #$2<--$3
addu $3,$1,$2      #$3<--$1+$2
or $1,$0,$2        #$1<--$2
or $2,$0,$3        #$2<--$3
```

（2） 验证代码（十六进制）：

```
34010000
34020001
00221821
00020825
00031025
00221821
00020825
00031025
00221821
00020825
00031025
00221821
00020825
00031025
```

00221821

00020825

00031025

（3） 仿真结果：

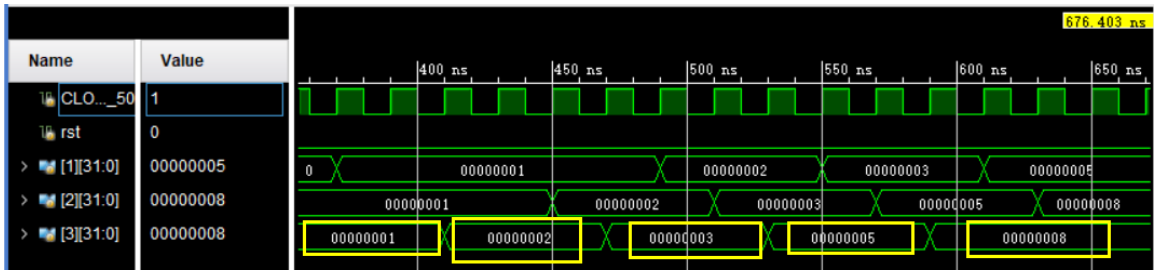


图 40：斐波那契数列仿真结果

（4） 结果验证：

1 号寄存器和 3 号寄存器作为加数，斐波那契数列存储在 3 号寄存器中。验证 3 号寄存器中的结果为 1，2，3，5，8，……与预期相符，说明实验结果正确。

4.5 显示斐波那契数列 EGO1 开发板验证

（1）更改时钟定时器的值使其频率降为 1Hz 便于在 EGO1 开发板上显示。

```
if(count > 32'd13500000) begin //1秒 1Hz
```

图 41：部分代码，更改时钟频率

（2）将 3 号寄存器的值与 8 个 LED 输出线 led_i 连接，使 8 个 led 灯显示斐波那契数列的值。

```
assign led0_o = regs[32'h00000003][32'h00000000]; //led0
assign led1_o = regs[32'h00000003][32'h00000001]; //led1
assign led2_o = regs[32'h00000003][32'h00000002]; //led2
assign led3_o = regs[32'h00000003][32'h00000003]; //led3
assign led4_o = regs[32'h00000003][32'h00000004]; //led4
assign led5_o = regs[32'h00000003][32'h00000005]; //led5
assign led6_o = regs[32'h00000003][32'h00000006]; //led6
assign led7_o = regs[32'h00000003][32'h00000007]; //led7
```

图 42：部分代码，连接 LED 输出线

（3）将 pc 值绑定数码管输出线，使 EGO1 的数码管显示指令的条数，对应斐波那契数列显示。

```
        pc=sw[1:0];
        an=8'b00000001;//显示第一个数码管，高电平有效
    end
    3'b001 :
    begin
        pc=sw[3:2];
        an=8'b00000010;//显示第二个数码管，低电平有效
    end
```

图 43：部分代码，连接数码管输出线

（4）配置引脚，添加对 LED 灯的约束和对数码管的约束的约束文件。

```
set_property PACKAGE_PIN K2 [get_ports led0_o]
set_property PACKAGE_PIN J2 [get_ports led1_o]
set_property PACKAGE_PIN J3 [get_ports led2_o]
set_property PACKAGE_PIN H4 [get_ports led3_o]
set_property PACKAGE_PIN J4 [get_ports led4_o]
set_property PACKAGE_PIN G3 [get_ports led5_o]
set_property PACKAGE_PIN G4 [get_ports led6_o]
set_property PACKAGE_PIN F6 [get_ports led7_o]
set_property PACKAGE_PIN P5 [get_ports rst]
```

```
set_property PACKAGE_PIN P5 [get_ports {sw[1]}]
set_property PACKAGE_PIN P4 [get_ports {sw[0]}]
set_property PACKAGE_PIN P3 [get_ports {sw[3]}]
set_property PACKAGE_PIN P2 [get_ports {sw[2]}]
set_property PACKAGE_PIN R2 [get_ports {sw[5]}]
set_property PACKAGE_PIN M4 [get_ports {sw[4]}]
set_property PACKAGE_PIN N4 [get_ports {sw[7]}]
set_property PACKAGE_PIN R1 [get_ports {sw[6]}]
```

图 44：部分代码， LED 灯和数码管的约束文件

（5）实验结果

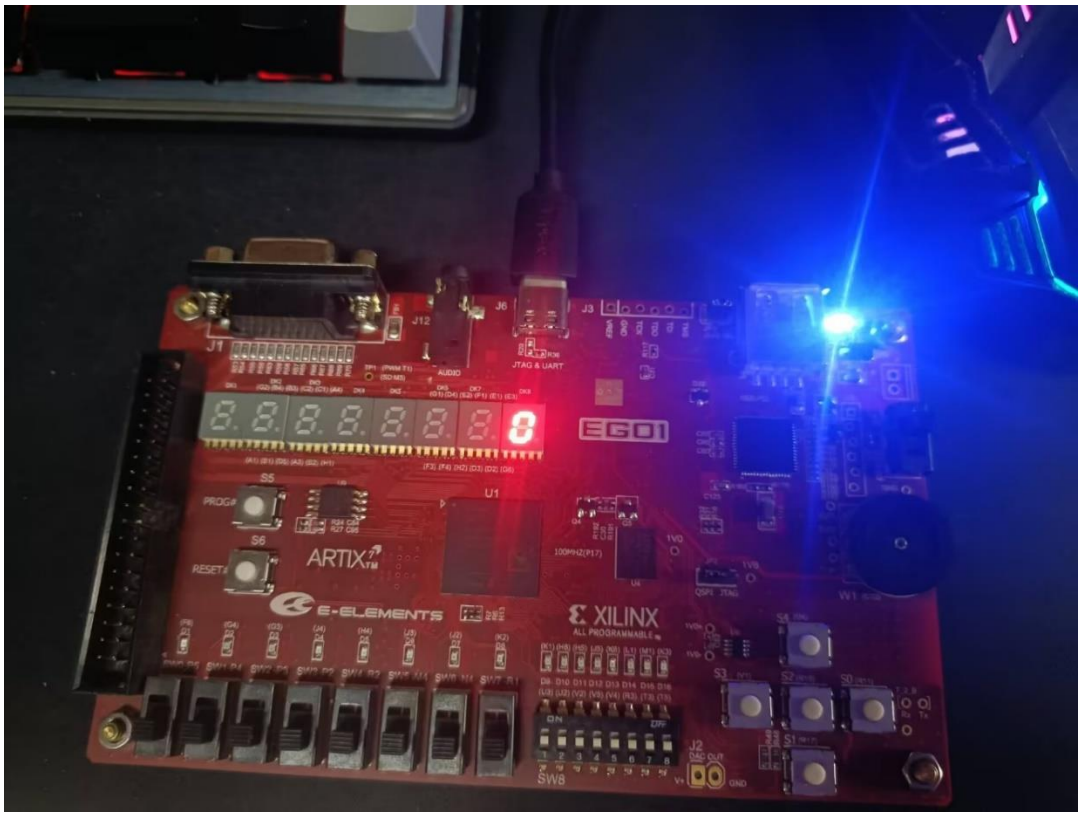


图 44：初始时指令为第 0 条（数码管），3 号寄存器初始值为 0（led）

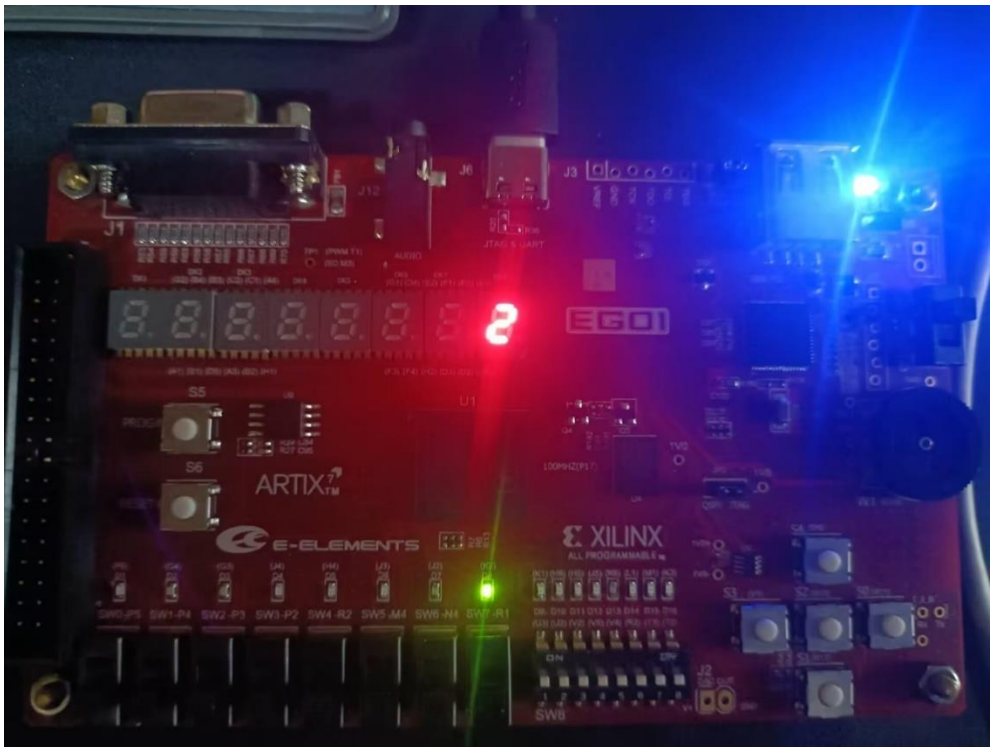


图 45：指令为第 2 条时（数码管），执行第 1 次 addu，3 号寄存器的值变为 1（led）

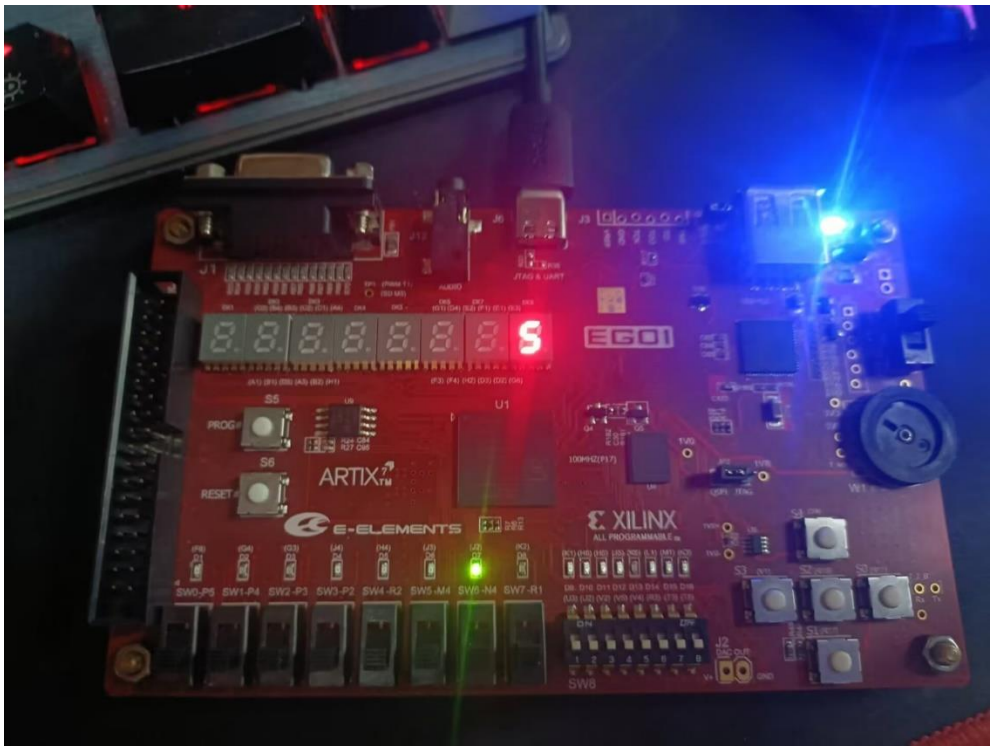


图 46：指令为第 5 条时（数码管），执行第 2 次 addu，3 号寄存器的值变为 2（led）

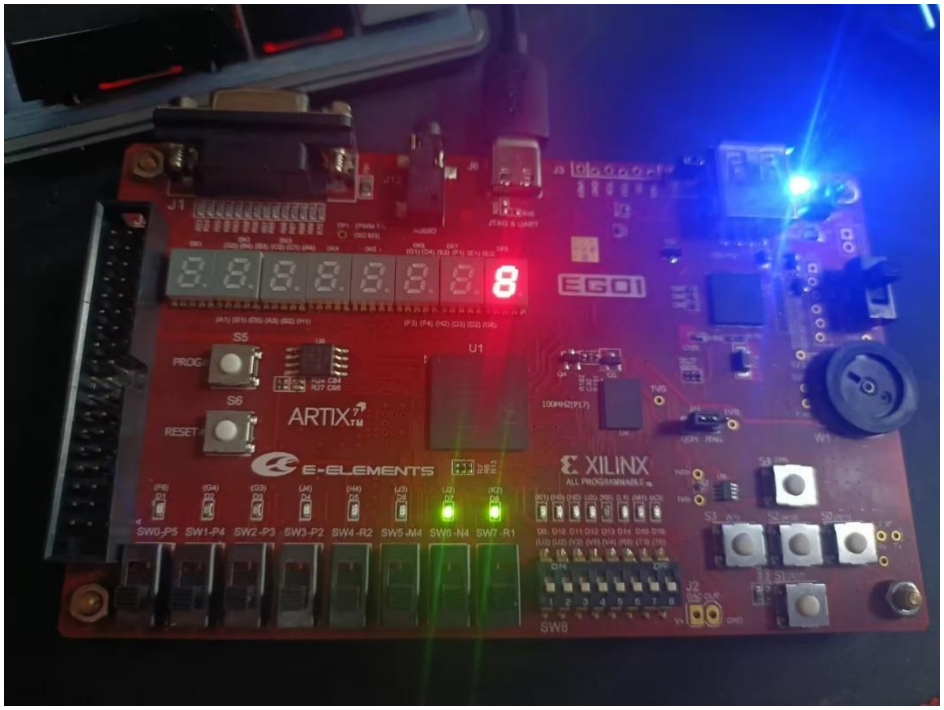


图 47：指令为第 8 条时（数码管），执行第 3 次 addu，3 号寄存器的值变为 3（led）

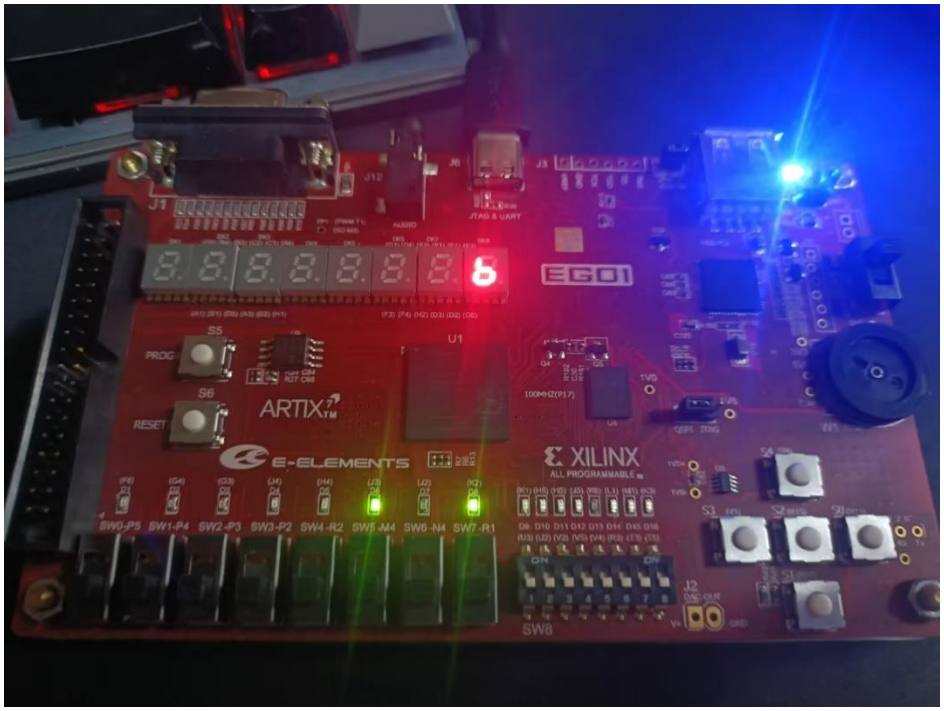


图 48：指令为第 11 条时（数码管），执行第 4 次 addu，3 号寄存器的值变为 5（led）

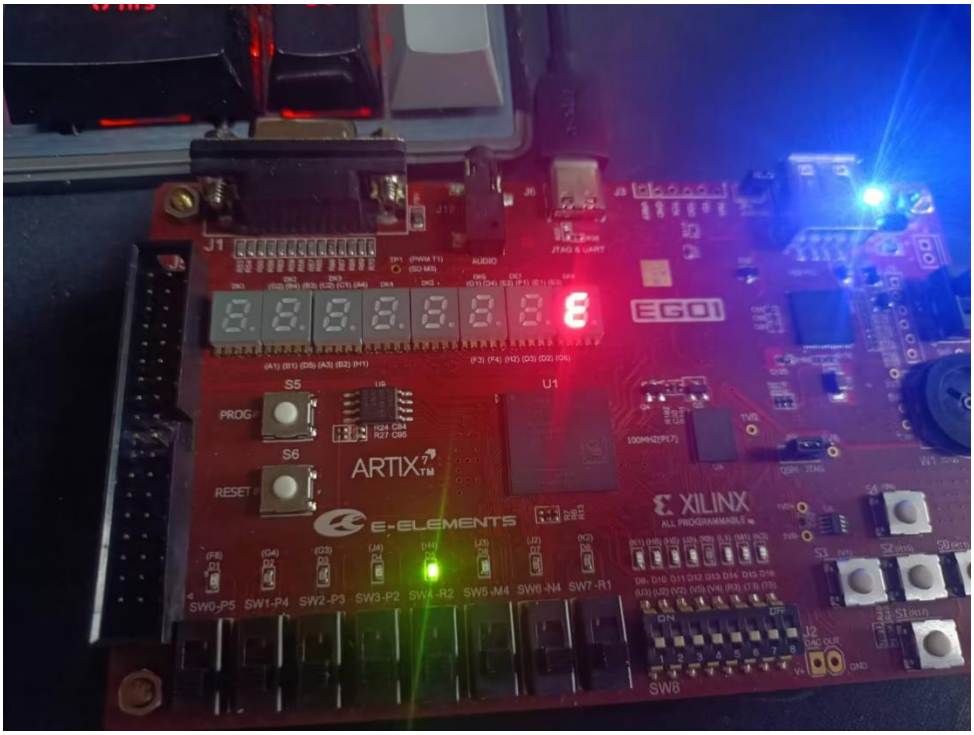


图 49：指令为第 14 条时（数码管），执行第 5 次 addu，3 号寄存器的值变为 8（led）

（6）结果验证

斐波那契数列显示正确，同时指令条数与斐波那契数列显示值一一对应，与预期结果相符，说明实验正确。

5　实验过程遇到的困难和解决办法

5.1 CPU 组成结构不清晰

完整的 CPU 可以非常复杂，涉及到多个模块和组件之间的相互作用。在一开始的设计时我们并未参考书籍，而是盲目的实现功能，导致设计变得混乱和难以管理。

解决方法：解决方法：参考组成原理等书籍，将 CPU 划分为多个模块，每个模块负责特定的功能，如控制单元、ALU、寄存器等。使用层次化设计方法，每个模块都有清晰的接口和功能，并确保模块之间的通信和数据传输正确。

5.2 未使用时序逻辑电路进行时钟同步

在一开始我们并没有使用时序逻辑电路，导致无法正确处理不同信号之间的

时间关系，使得流水线间的信号变得冲突和不稳定，造成结果的混乱。

解决方法：增加了中间的同步模块如 if_id 等，使用时序逻辑电路在时钟上升沿传递信号，使得各级流水线之间的信号传输同步。

5.3 LED 灯与数码管引脚配置错误

在实验过程中出现了引脚配置错误的问题，导致结果与预期不符。

解决方法：仔细查看 EGO1 开发板手册

5.4 LED 灯常亮

在实验一开始我们忘记配置延时模块就进行上板操作，由于时钟频率过高导致灯的闪烁速度过快，使人眼感觉到 LED 灯常亮。

解决方法：使用软件延时，增加一个延时计数器，每到输入时钟上升沿就将计数器减 1，当计数器的值为 0 时使输出时钟信号反转。我们使用了一个模块 delay_clk 来实现延迟操作，将时钟频率减慢至 1Hz，使得灯的闪烁速度肉眼可见。

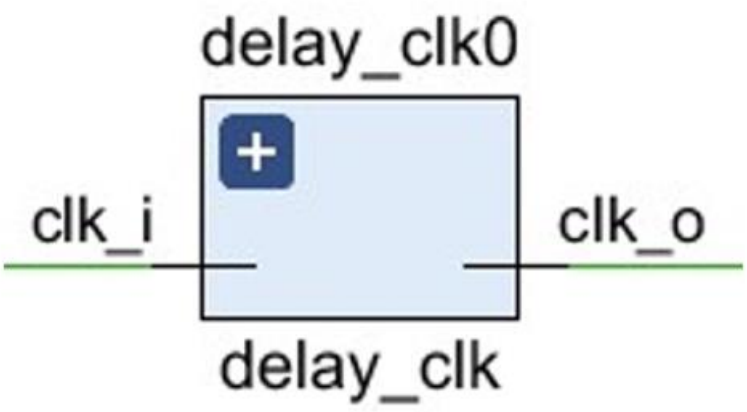


图 50：延时模块

6 总结

本实验旨在通过 Verilog HDL 语言设计一个基于 MIPS32 指令集架构的流水线 CPU，并将其部署到 FPGA 芯片上进行验证。通过与队友的合作，我们顺利的完成了这一实验的设计任务，在实践中增强了自己的动手能力。

在本次实验中，我们完成了以下任务：

（1） 根据 MIPS32 指令集架构的规范，我们设计了一个基于流水线的 CPU。流水线分为指令取指、指令译码、执行、访存和写回等多个阶段，每个阶段都完成特定的任务。我们合理划分和组织了 CPU 内部的数据通路和控制信号，并设计了适当的寄存器和数据路径。

（2） 根据实验要求，我们实现了 37 种 MIPS32 指令集，包括常见的算术逻辑指令和移位指令等。我们使用 Verilog HDL 编写了每个指令的处理逻辑，并在 CPU 的控制单元中实现了指令译码和控制信号生成。

（3） 在流水线 CPU 设计中，我们遇到了数据冒险，我们通过数据前推处理，确保了指令的正确执行顺序和数据的正确性。

（4） 在设计完成后，我们设计了一系列的实验，通过观察输出结果和波形图，确保 CPU 在不同的指令序列和数据输入下能够正常工作。我们还将其下载到 FPGA 芯片上，在硬件上验证了实验的正确性。

在本次实验中，我们的收获：

（1） 本次实验涵盖了数字逻辑、计算机组成原理和计算机体系结构等多个课程的内容，使我能够将所学的理论知识应用于实际的 CPU 设计中。

（2） 通过本次实验的学习，我深入理解了 CPU 的基本原理和指令集的组成，熟悉了 Verilog HDL 语言的语法和编程技巧。我还学习和理解了流水线设计的原理和优化方法，提高了实践能力和问题解决能力，学会了通过实验获得验证和调试的经验。

7 参考文献

[1] 雷思磊. 自己动手写 CPU[M]. 北京：电子工业出版社，2014.

[2] 阎石 / 王红 / 清华大学电子学教研组. 数字电子技术基础[M]. 北京: 高等教育出版社，2016.

[3] 唐朔飞. 计算机组成原理 [M]. 北京：高等教育出版社，2008.

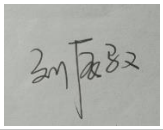
[4] 斯坦福大学. 计算机体系结构 量化研究方法[M]. 北京：机械工业出版社，2007.

[5] ycc_job. 自己动手写 CPU_4_第一条指令 ori 的实现. https://www.cnblogs.com/ycc1997/p/12098846.html, 2020-01-09

一、原创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

特此声明！

作者签字: 

	评价内容	权重	得分
验收		0.4	
设计报告		0.6	
合计			
指导教师（签章）: 2024 年 1 月 日			