浙江大学
ZHEJIANG UNIVERSITY

计算机组成 实验报告

姓　　名:	NoughtQ
学　　号:	1145141919810
专　　业:	计算机科学与技术
课程名称:	计算机组成
指导教师:	赵莎
实验地点:	东四 509 教室

2024 年 12 月 10 日

目录

Lab5：流水线 CPU

一、 实验目的和要求

1. 理解流水线 CPU 的基本原理和组织结构
2. 掌握五级流水线的工作过程和设计方法
3. 理解流水线取指、译码、执行、存储器访问、写回、停机的设计原理和解决方法
4. 设计流水线测试程序

二、 实验内容和原理

2.1 流水线处理器集成

2.1.1 实验目标及任务

- 目标：熟悉 RISC-V 五级流水线的工作特点，了解流水线处理器的原理，掌握 IP 核的使用方法，集成并测试 CPU
- 任务：
  1. 集成设计流水线 CPU，在 Exp04 的基础上完成
     - 利用五级流水线各级封装模块集成 CPU
     - 替换 Exp04 的单周期 CPU 为本实验集成的五级流水线 CPU
  2. 设计流水线测试方案并完成测试

2.1.2 实验原理

相比单周期 CPU 一个时钟周期内只能执行一条指令而言，流水线 CPU 具备了在一个时钟周期内同时加工多条指令的能力。流水线虽然没有缩短单步所花的时间（即**时延** (latency)），但是它增加了每个阶段内能够执行的任务（即增大了**吞吐量** (throughput)），从而缩短完成整个任务的总时间。

这里讨论的流水线 CPU 均为五级流水线 CPU，它能在单个时钟周期内至多并行执行五个阶段的任务，包括：

- IF（取指）：从内存中获取指令
- ID（译码）：读取寄存器，对指令进行译码
- EX（执行）：执行（算术 / 逻辑）运算或计算地址
- MEM（访存）：从数据内存中访问操作数
- WB（写回）：将结果写回寄存器中

流水线 CPU 的接口如下所示（实际上与单周期 CPU 的没什么区别）：

```
module Pipeline_CPU(
    input clk,                  // 时钟
    input rst,                  // 复位
    input[31:0] Data_in,        // 存储器数据输入
```

```
    input[31:0] inst_IF,          // 取指阶段指令
    output [31:0] PC_out_IF,      // 取指阶段 PC 输出
    output [31:0] PC_out_ID,      // 译码阶段 PC 输出
    output [31:0] inst_ID,        // 译码阶段指令
    output [31:0] PC_out_EX,      // 执行阶段 PC 输出
    output [31:0] MemRW_EX,       // 执行阶段存储器读写
    output [31:0] MemRW_Mem,      // 访存阶段存储器读写
    output [31:0] Addr_out,       // 地址输出
    output [31:0] Data_out,       // CPU 数据输出
    output [31:0] Data_out_WB     // 写回数据输出
);
```

下面将从**数据通路**和**控制器**这两个部分来简要介绍 流水线 CPU 的工作原理。

- **数据通路**

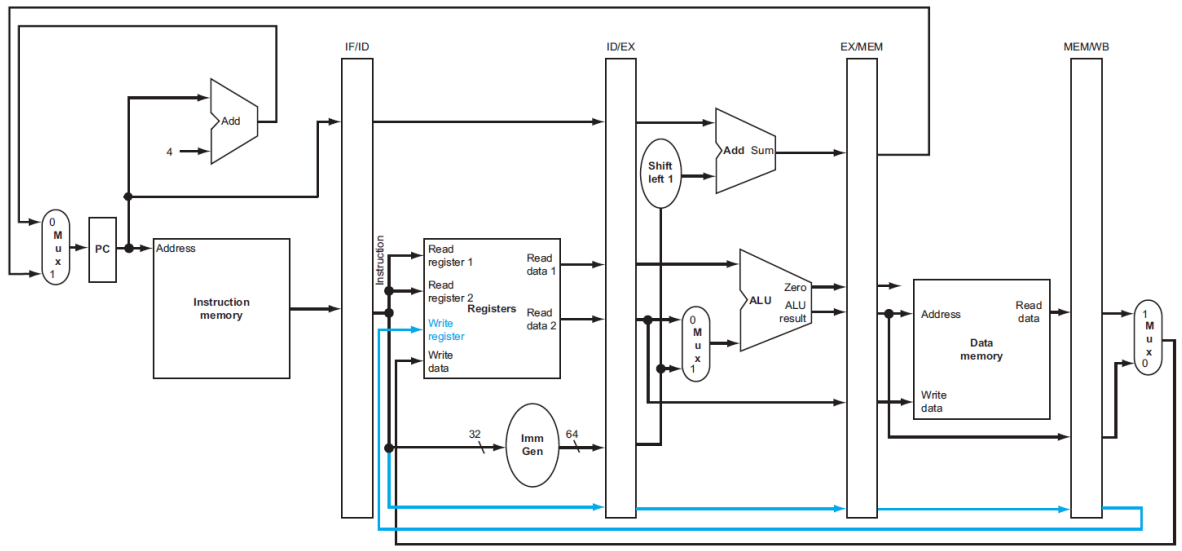


Figure 1: 流水线 CPU 数据通路

相比单周期 CPU 而言，流水线 CPU 最大的变化在于每两个阶段之间加入了一个流水线寄存器，共 4 个：IF/ID、ID/EX、EX/MEM、MEM/WB，用于保存执行指令所需要的数据和控制信号，以达到在同一时间内执行多条指令（的不同阶段）的目的。

· **控制器**

信号	源数目	功能定义	赋值0时动作	赋值1时动作	赋值2时动作
ALUSrc_B	2	ALU端口B输入选择	选择源操作数寄存器2数据	选择32位立即数（符号扩展后）	-
MemToReg	3	寄存器写入数据选择	选择ALU输出	选择存储器数据	选择PC+4
Branch		Beq指令目标地址选择	选择PC+4地址	选择转移目的地址PC+imm（zero=1）	-
BranchN		Bne指令目标地址选择	选择PC+4地址	选择转移目的地址PC+imm（zero=0）	-
Jump		Jal指令目标地址选择	选择PC+4地址	选择跳转目的地址	-
PCSrc	2	PC输入选择（分支跳转的衍生）	=(Branch&zero)|(BranchN&(~zero))|Jump		-
RegWrite	-	寄存器写控制	禁止寄存器写	使能寄存器写	-
MemRW	-	存储器读写控制	存储器**读使能**，存储器写禁止	存储器**写使能**，存储器读禁止	-
ALU_Control	000-111	3位ALU操作控制	参考表ALU_Control（详见实验04）		
ImmSel	00-11	2位立即数组合控制	参考表ImmSel（详见实验04）		

Figure 2: 流水线 CPU 控制信号

流水线 CPU 的控制信号与单周期 CPU 的没有多大的差别，但需要注意的是这些控制信号也要和数据一起在流水线寄存器之间传递。

将控制器及其控制信号（蓝色部分）加入到数据通路上，我们得到了一个完整的流水线 CPU。

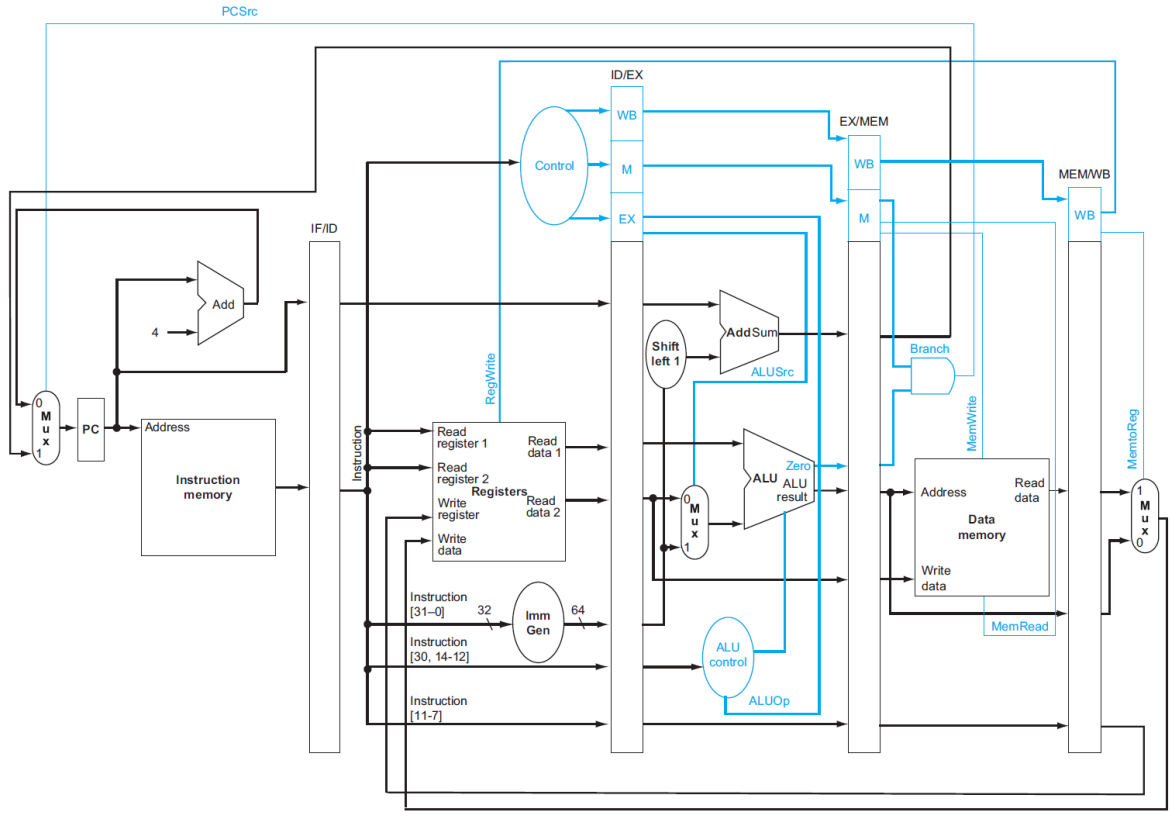


Figure 3: 流水线 CPU 数据通路+控制器

2.2 取指、译码部分设计

2.2.1 实验目标及任务

- 目标：熟悉 RISC-V 五级流水线的工作特点，了解取指、译码的原理，掌握 IP 核的使用方法，集成并测试 CPU
- 任务：
  1. 设计取指（IF）、译码（ID）模块，替换实验九的流水线 CPU 并完成集成
     - 设计取指模块，替换 OExp05-1 的取指模块并完成集成
     - 设计译码模块，替换 OExp05-1 的译码模块并完成集成
  2. 设计流水线测试方案并完成测试

2.2.2 实验原理

取指部分(IF)

取指阶段涉及到的数据通路如下所示:

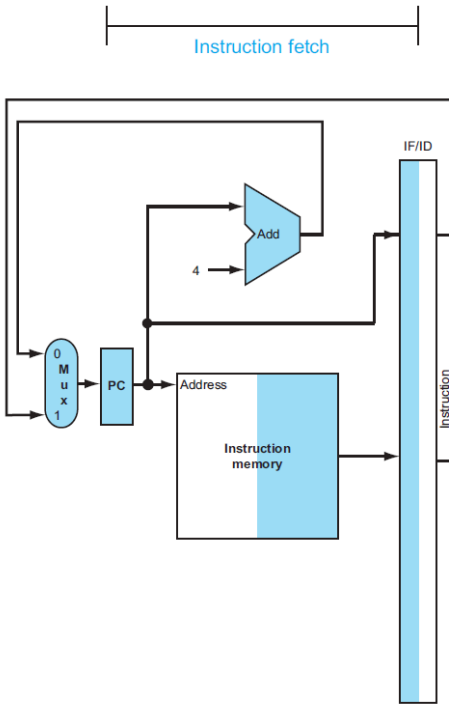


Figure 4: 流水线 CPU 取指部分

取指部分数据通路的接口如下所示:

```
module Pipeline_IF(
    input clk_IF,                   // 时钟
    input rst_IF,                   // 复位
    input en_IF,                    // 使能
    input [31:0] PC_in_IF,          // 取指令 PC 输入
    input PCSrc,                    // PC 输入选择
    output reg [31:0] PC_out_IF
);
```

IF/ID 流水线寄存器的接口如下所示:

```
module IF_reg_ID(
    input clk_IFID,                 // 寄存器时钟
    input rst_IFID,                 // 寄存器复位
    input en_IFID,                  // 寄存器使能
    input [31:0] PC_in_IFID,        // PC 输入
    input [31:0] inst_in_IFID,      // 指令输入
```

```
    output reg [31:0] PC_out_IFID,      // PC 输出
    output reg [31:0] inst_out_IFID     // 指令输出
);
```

译码部分(ID)

译码阶段涉及到的数据通路如下所示:

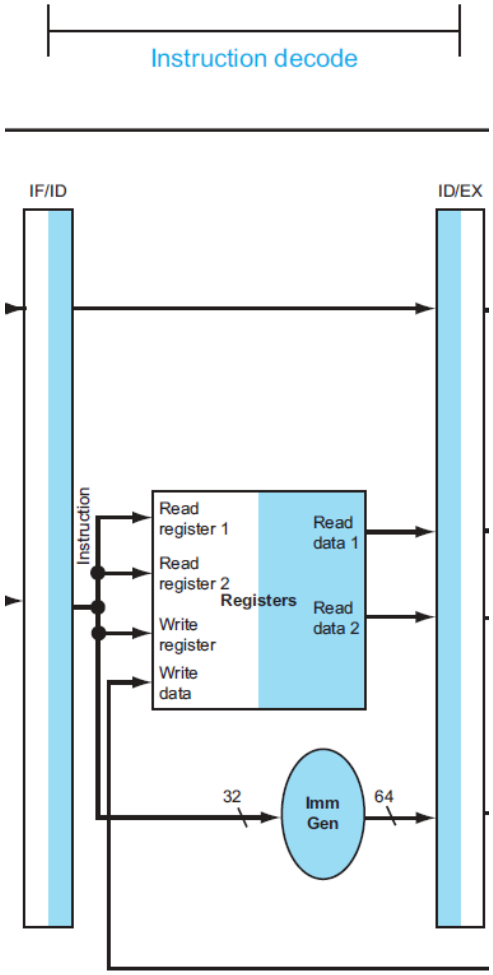


Figure 5: 流水线 CPU 译码部分

译码部分数据通路的接口如下所示:

```
module Pipeline_ID(
    input clk_ID,                       // 时钟
    input rst_ID,                       // 复位
    input RegWrite_in_ID,               // 寄存器堆使能
    input [4:0] Rd_addr_ID,             // 写目的地址输入
    input [31:0] Wt_data_ID,            // 写数据输入
    input [31:0] Inst_in_ID,            // 指令输入
```

```
    output reg [31:0] Rd_addr_out_ID,       // 写目的地址输出
    output reg [31:0] Rs1_out_ID,           // 操作数 1 输出
    output reg [31:0] Rs2_out_ID ,          // 操作数 2 输出
    output reg [31:0] Imm_out_ID ,          // 立即数输出
    output reg ALUSrc_B_ID ,                // ALU B 端输入选择
    output reg [2:0] ALU_control_ID,        // ALU 控制
    output reg Branch_ID,                   // Beq 控制
    output reg BranchN_ID,                  // Bne 控制
    output reg MemRW_ID,                    // 存储器读写
    output reg Jump_ID,                     // Jal 控制
    output reg [1:0] MemtoReg_ID,           // 寄存器写回选择
    output reg RegWrite_out_ID              // 寄存器堆读写
);
```

ID/EX 流水线寄存器的接口如下所示:

```
module ID_reg_Ex(
    input clk_IDEX,                         // 寄存器时钟
    input rst_IDEX,                         // 寄存器复位
    input en_IDEX,                          // 寄存器使能
    input [31:0] PC_in_IDEX,                // PC 输入
    input [4:0] Rd_addr_IDEX,               // 写目的地址输入
    input [31:0] Rs1_in_IDEX,               // 操作数 1 输入
    input [31:0] Rs2_in_IDEX,              // 操作数 2 输如
    input [31:0] Imm_in_IDEX,               // 立即数输入
    input ALUSrc_B_in_IDEX,                 // ALU B 输入选择
    input [2:0] ALU_control_in_IDEX,        // ALU 选择控制
    input Branch_in_IDEX,                   // Beq
    input BranchN_in_IDEX,                  // Bne
    input MemRW_in_IDEX,                    // 存储器读写
    input Jump_in_IDEX,                     // Jal
    input [1:0] MemtoReg_in_IDEX,           // 写回选择
    input RegWrite_in_IDEX,                 // 寄存器堆读写
    output reg [31:0] PC_out_IDEX,          // PC 输出
    output reg [4:0] Rd_addr_out_IDEX,      // 目的地址输出
    output reg [31:0] Rs1_out_IDEX,         // 操作数 1 输出
    output reg [31:0] Rs2_out_IDEX,         // 操作数 2 输出
    output reg [31:0] Imm_out_IDEX,         // 立即数输出
    output reg ALUSrc_B_out_IDEX,           // ALU B 选择
    output reg [2:0] ALU_control_out_IDEX,  // ALU 控制
    output reg Branch_out_IDEX,             // Beq
    output reg BranchN_out_IDEX,            // Bne
```

```
    output reg MemRW_out_IDEX,                  // 存储器读写
    output reg Jump_out_IDEX,                   // Jal
    output reg [1:0] MemtoReg_out_IDEX,         // 写回
    output reg RegWrite_out_IDEX                // 寄存器堆读写
);
```

2.3 执行、访存、写回部分设计

2.3.1 实验目标及任务

- 目标：熟悉 RISC-V 五级流水线的工作特点，了解执行、存储器访问、写回的原理，掌握 IP 核的使用方法，集成并测试 CPU
- 任务：
  1. 设计执行（Ex）、存储器访问（Mem）、写回（WB）模块，替换 lab04 的流水线 CPU 并完成集成
     - 设计执行模块，替换 OExp05-2 的执行模块并完成集成
     - 设计访存模块，替换 OExp05-2 的访存模块并完成集成
     - 设计写回模块，替换 OExp05-2 的写回模块并完成集成
  2. 设计流水线测试方案并完成测试

2.3.2 实验原理

执行部分(EX)

执行阶段涉及到的数据通路如下所示：

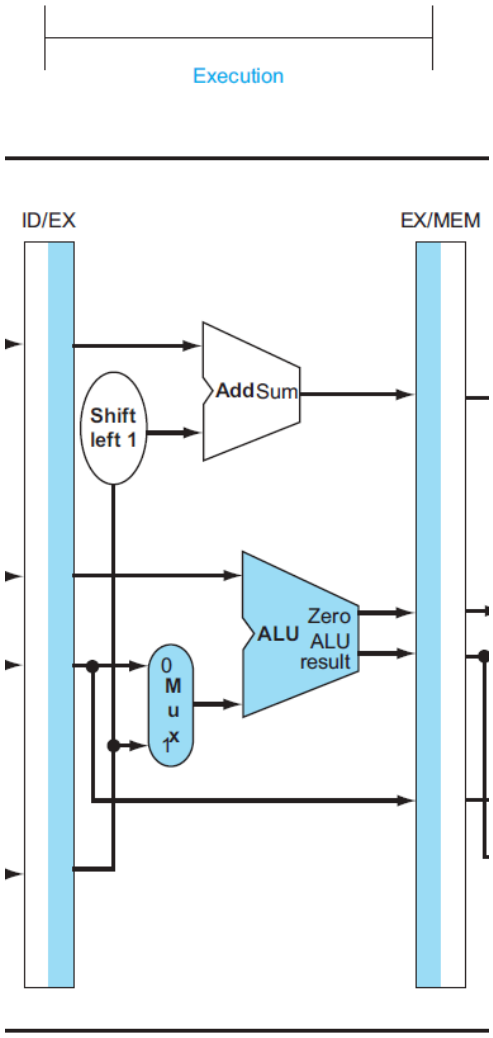


Figure 6: 流水线 CPU 执行部分

执行部分数据通路的接口如下所示:

```
module Pipeline_Ex(
    input [31:0] PC_in_EX,              // PC 输入
    input [31:0] Rs1_in_EX,             // 操作数 1 输入
    input [31:0] Rs2_in_EX,             // 操作数 2 输入
    input [31:0] Imm_in_EX,             // 立即数输入
    input ALUSrc_B_in_EX ,              // ALU B 选择
    input [2:0] ALU_control_in_EX,      // ALU 选择控制
    output reg [31:0] PC_out_EX,        // PC 输出
    output reg [31:0]PC4_out_EX,        // PC+4 输出
    output reg zero_out_EX,             // ALU 判 0 输出
    output reg [31:0] ALU_out_EX,       // ALU 计算输出
    output reg[31:0] Rs2_out_EX         // 操作数 2 输出
);
```

EX/MEM 流水线寄存器的接口如下所示:

```
module Ex_reg_Mem(
    input clk_EXMem,                        // 寄存器时钟
    input rst_EXMem,                        // 寄存器复位
    input en_EXMem,                         // 寄存器使能
    input [31:0] PC_in_EXMem,               // PC 输入
    input [31:0] PC4_in_EXMem,              // PC+4 输入
    input [4:0] Rd_addr_EXMem,              // 写目的寄存器地址输入
    input zero_in_EXMem,                    // zero
    input [31:0] ALU_in_EXMem,              // ALU 输入
    input [31:0] Rs2_in_EXMem,              // 操作数 2 输入
    input Branch_in_EXMem,                  // Beq
    input BranchN_in_EXMem,                 // Bne
    input MemRW_in_EXMem,                   // 存储器读写
    input Jump_in_EXMem,                    // Jal
    input [1:0] MemtoReg_in_EXMem,          // 写回
    input RegWrite_in_EXMem,                // 寄存器堆读写
    output reg [31:0] PC_out_EXMem,         // PC 输出
    output reg [31:0] PC4_out_EXMem,        // PC+4 输出
    output reg [4:0] Rd_addr_out_EXMem,     // 写目的寄存器输出
    output reg zero_out_EXMem,              // zero
    output reg [31:0] ALU_out_EXMem,        // ALU 输出
    output reg [31:0] Rs2_out_EXMem,        // 操作数 2 输出
    output reg Branch_out_EXMem,            // Beq
    output reg BranchN_out_EXMem,           // Bne
    output reg MemRW_out_EXMem,             // 存储器读写
    output reg Jump_out_EXMem,              // Jal
    output reg [1:0] MemtoReg_out_EXMem,    // 写回
    output reg RegWrite_out_EXMem           // 寄存器堆读写
);
```

访存部分(EX)

访存阶段涉及到的数据通路如下所示:

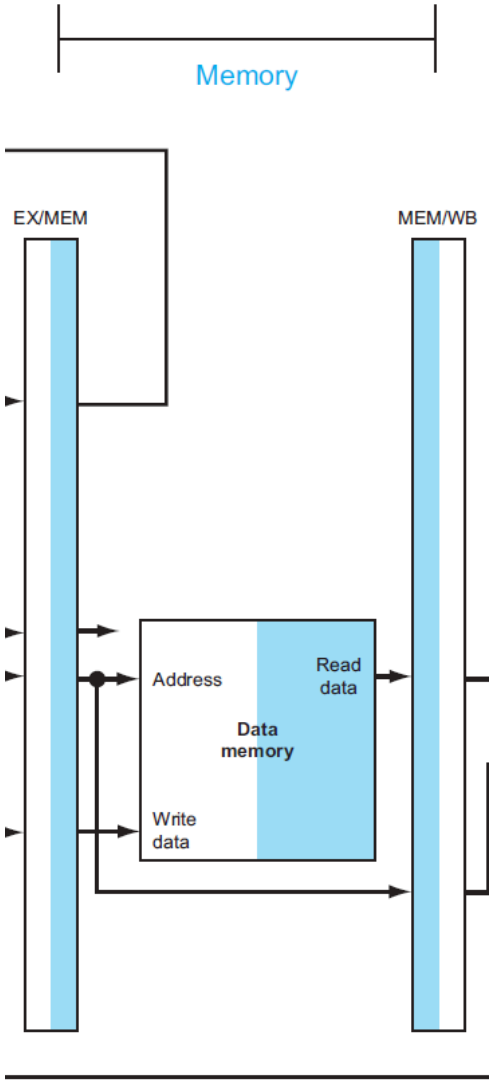


Figure 7: 流水线 CPU 访存部分

访存部分数据通路的接口如下所示:

```
module Pipeline_Mem(
    input zero_in_Mem,        // zero
    input Branch_in_Mem,      // beq
    input BranchN_in_Mem,     // bne
    input Jump_in_Mem,        // jal
    output PCSrc              // PC 选择控制输出
);
```

MEM/WB 流水线寄存器的接口如下所示:

```
module Mem_reg_WB(
    input clk_MemWB,                                    // 寄存器时
```

```
    input rst_MemWB,                        // 寄存器复位
    input en_MemWB,                         // 寄存器使能
    input [31:0] PC4_in_MemWB,              // PC+4 输入
    input [4:0] Rd_addr_MemWB,              // 写目的地址输入
    input [31:0] ALU_in_MemWB,              // ALU 输入
    input [31:0] DMem_data_MemWB,           // 存储器数据输入
    input [1:0] MemtoReg_in_MemWB,          // 写回
    input RegWrite_in_MemWB,                // 寄存器堆读写
    output reg [31:0] PC4_out_MemWB,        // PC+4 输出
    output reg [4:0] Rd_addr_out_MemWB,     // 写目的地址输出
    output reg [31:0] ALU_out_MemWB,        // ALU 输出
    output reg [31:0] DMem_data_out_MemWB,  // 存储器数据输出
    output reg [1:0] MemtoReg_out_MemWB,    // 写回
    output reg RegWrite_out_MemWB           // 寄存器堆读写
);
```

写回部分(WB)

写回阶段涉及到的数据通路如下所示:

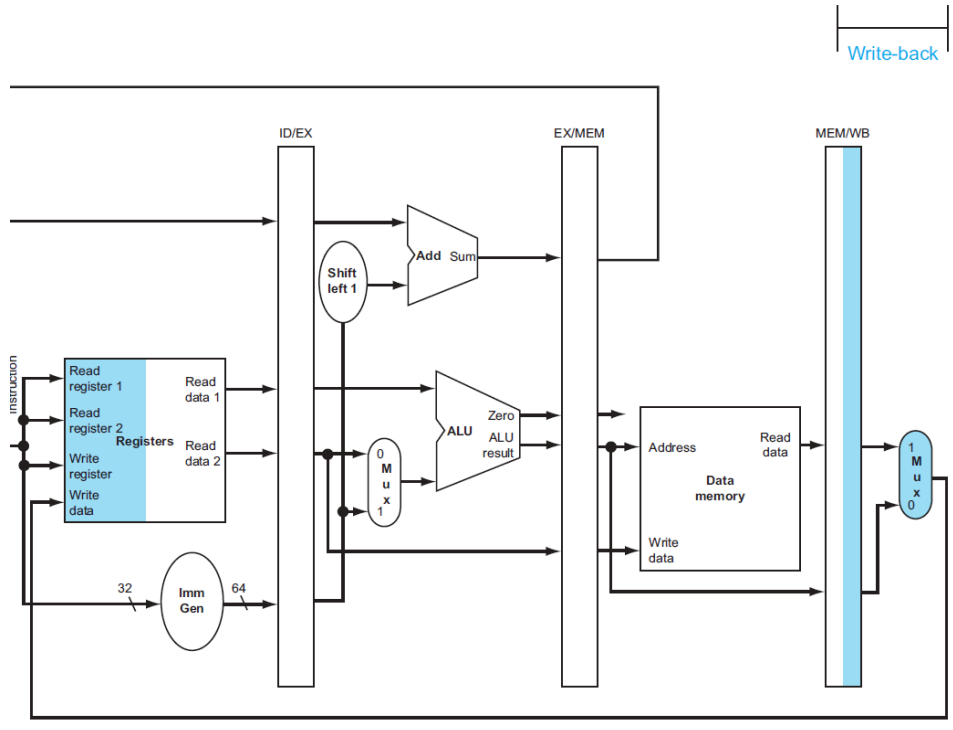


Figure 8: 流水线 CPU 写回部分

写回部分数据通路的接口如下所示:

```
module Pipeline_WB(
    input [31:0] PC4_in_WB,         // PC+4 输入
    input [31:0] ALU_in_WB,         // ALU 结果输出
    input [31:0] DMem_data_WB,      // 存储器数据输入
    input [1:0] MemtoReg_in_WB,     // 写回选择控制
    output [31:0] Data_out_WB       // 写回数据输出
);
```

2.4 冒险与停顿处理

2.4.1 实验目标及任务

- 目标: 熟悉 RISC-V 五级流水线的工作特点, 了解流水线冒险的产生原因及解决办法, 掌握 IP 核的使用方法，集成并测试 CPU
- 任务:
  1. 集成设计利用 stall 解决冒险的流水线 CPU，在 lab05-3 的基础上完成
     - 设计冒险检测及 stall 消除冒险的流水线 CPU
     - 替换 lab05-3 的 CPU 为本实验集成的带 stall 处理的流水线 CPU
  2. 设计流水线测试方案并完成测试

2.4.2 实验原理

流水线 CPU 的一大问题是流水线冒险，有以下几类不同的冒险类型:

- 结构冒险（不作讨论）: 硬件不支持多条指令在同一时钟周期执行
- **数据冒险**: 当前指令的执行需要等待前一条指令的数据结果
  - 数据冒险发生在 EX 或 Mem 阶段内。当满足以下条件的任意一条时，数据冒险就发生了:

```
// EX hazard
if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
        Data_hazard = 1

if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
        Data_hazard = 1

// MEM hazard
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs1))
        Data_hazard = 1
```

```
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs2))
        Data_hazard = 1
```

- 解决方案：前递(forwarding)、停顿(stalling)，本实验采用后一种方案

- **控制冒险**：指令非顺序执行而导致下一条执行的指令不是真实期望的
  - 当满足以下条件的任意一条时，控制冒险就发生了：

```
if (Branch_ID == 1 or BranchN_ID == 1 or Jump_ID == 1)
  or (Branch_out_IDEX == 1 or BranchN_out_IDEX == 1
        or Jump_out_IDEX == 1)
  or (Branch_out_EXMem == 1 or BranchN_out_EXMem == 1
        or Jump_out_EXMem == 1))
    Control_hazard = 1
```

  - 解决方案：停顿(stalling)、静态/动态分支预测，本实验采用前一种方案

根据上面的判断条件，我们在原来流水线 CPU 的基础上，额外增加一个处理停顿的元件，其接口如下所示：

```
module stall(
    input rst_stall,                    // 复位
    input RegWrite_out_IDEX,            // 执行阶段寄存器写控制
    input [4:0] Rd_addr_out_IDEX,       // 执行阶段寄存器写地址
    input RegWrite_out_EXMem,           // 访存阶段寄存器写控制
    input [4:0] Rd_addr_out_EXMem,      // 访存阶段寄存器写地址
    input [4:0] Rs1_addr_ID,            // 译码阶段寄存器读地址 1
    input [4:0] Rs2_addr_ID,            // 译码阶段寄存器读地址 2
    input Rs1_used,                     // Rs1 被使用
    input Rs2_used,                     // Rs2 被使用
    input Branch_ID,                    // 译码阶段 beq
    input BranchN_ID,                   // 译码阶段 bne
    input Jump_ID,                      // 译码阶段 jal
    input Branch_out_IDEX,              // 执行阶段 beq
    input BranchN_out_IDEX,             // 执行阶段 bne
    input Jump_out_IDEX,                // 执行阶段 jal
    input Branch_out_EXMem,             // 访存阶段 beq
    input BranchN_out_EXMem,            // 访存阶段 bne
    input Jump_out_EXMem,               // 访存阶段 jal
    output en_IF,                       // 流水线寄存器的使能及 NOP 信号
    output en_IFID,
```

```
    output NOP_IFID,
    output NOP_IDEX
);
```

三、 实验设备和环境

- 操作系统：Windows 11
- 开发工具：Xilinx VIVADO 2023.2
- NEXYS A7 开发板

四、 实验步骤

> 注意
>
> 因为 Typst 会将代码块中的 <= 自动转化为 ⩽，但我在代码中多次用到这类非阻塞赋值的符号，因此可能会影响到代码的阅读，请见谅~

4.1 流水线处理器集成

1. 在 Vivado 上新建工程 OExp05-Pipeline_CPU
2. 在工程中新建文件 `Pipeline_CPU.v`，作为顶层模块
3. 将已提供的流水线 CPU 模块（5 个阶段的数据通路 + 4 个流水线寄存器）添加到当前工程的 IP 库目录中，文件结构如下所示：

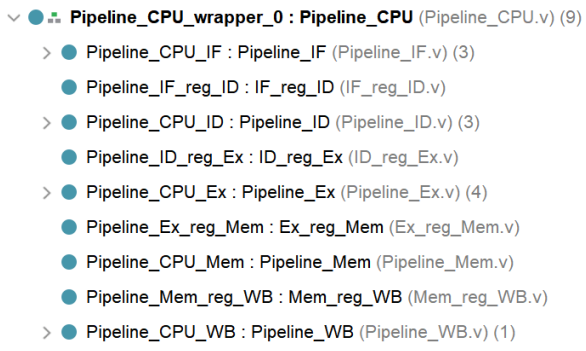


Figure 9: 文件结构(但这里给出的是完成实验第 3 部分之后的文件结构, 大致是一样的)

4. 按照给出的逻辑原理图，利用 Verilog 语言进行模块的调用和连接，代码如下所示：

```
`timescale 1ns / 1ps

module Pipeline_CPU(
    input clk,                  // 时钟
    input rst,                  // 复位
    input[31:0] Data_in,        // 存储器数据输入
    input[31:0] inst_IF,        // 取指阶段指令
```

```
    output [31:0] PC_out_IF,     // 取指阶段 PC 输出
    output [31:0] PC_out_ID,     // 译码阶段 PC 输出
    output [31:0] inst_ID,       // 译码阶段指令
    output [31:0] PC_out_EX,     // 执行阶段 PC 输出
    output [31:0] MemRW_EX,      // 执行阶段存储器读写
    output [31:0] MemRW_Mem,     // 访存阶段存储器读写
    output [31:0] Addr_out,      // 地址输出
    output [31:0] Data_out,      // CPU 数据输出
    output [31:0] Data_out_WB    // 写回数据输出
);

wire Pipeline_Mem_PCSrc;
wire [31:0] Pipeline_IF_PC_out_IF;
wire [31:0] IF_reg_ID_PC_out_IFID;
wire [31:0] IF_reg_ID_inst_out_IFID;
wire [31:0] Pipeline_WB_Data_out_WB;
wire [31:0] Pipeline_ID_Rd_addr_out_ID;
wire [31:0] Pipeline_ID_Rs1_out_ID;
wire [31:0] Pipeline_ID_Rs2_out_ID;
wire [31:0] Pipeline_ID_Imm_out_ID;
wire Pipeline_ID_ALUSrc_B_ID;
wire [2:0] Pipeline_ID_ALU_control_ID;
wire Pipeline_ID_Branch_ID;
wire Pipeline_ID_BranchN_ID;
wire Pipeline_ID_MemRW_ID;
wire Pipeline_ID_Jump_ID;
wire [1:0] Pipeline_ID_MemtoReg_ID;
wire Pipeline_ID_RegWrite_out_ID;
wire [31:0] ID_reg_Ex_PC_out_IDEX;
wire [4:0] ID_reg_Ex_Rd_addr_out_IDEX;
wire [31:0] ID_reg_Ex_Rs1_out_IDEX;
wire [31:0] ID_reg_Ex_Rs2_out_IDEX;
wire [31:0] ID_reg_Ex_Imm_out_IDEX;
wire ID_reg_Ex_ALUSrc_B_out_IDEX;
wire [2:0] ID_reg_Ex_ALU_control_out_IDEX;
wire ID_reg_Ex_Branch_out_IDEX;
wire ID_reg_Ex_BranchN_out_IDEX;
wire ID_reg_Ex_MemRW_out_IDEX;
wire ID_reg_Ex_Jump_out_IDEX;
wire [1:0] ID_reg_Ex_MemtoReg_out_IDEX;
wire ID_reg_Ex_RegWrite_out_IDEX;
wire [31:0] Pipeline_Ex_PC_out_EX;
```

```
wire [31:0] Pipeline_Ex_PC4_out_EX;
wire Pipeline_Ex_zero_out_EX;
wire [31:0] Pipeline_Ex_ALU_out_EX;
wire [31:0] Pipeline_Ex_Rs2_out_EX;
wire [31:0] Ex_reg_Mem_PC_out_EXMem;
wire [31:0] Ex_reg_Mem_PC4_out_EXMem;
wire [4:0] Ex_reg_Mem_Rd_addr_out_EXMem;
wire Ex_reg_Mem_zero_out_EXMem;
wire [31:0] Ex_reg_Mem_ALU_out_EXMem;
wire [31:0] Ex_reg_Mem_Rs2_out_EXMem;
wire Ex_reg_Mem_Branch_out_EXMem;
wire Ex_reg_Mem_BranchN_out_EXMem;
wire Ex_reg_Mem_MemRW_out_EXMem;
wire Ex_reg_Mem_Jump_out_EXMem;
wire [1:0] Ex_reg_Mem_MemtoReg_out_EXMem;
wire Ex_reg_Mem_RegWrite_out_EXMem;
wire [31:0] Mem_reg_WB_PC4_out_MemWB;
wire [4:0] Mem_reg_WB_Rd_addr_out_MemWB;
wire [31:0] Mem_reg_WB_ALU_out_MemWB;
wire [31:0] Mem_reg_WB_DMem_data_out_MemWB;
wire [1:0] Mem_reg_WB_MemtoReg_out_MemWB;
wire Mem_reg_WB_RegWrite_out_MemWB;


Pipeline_IF Pipeline_CPU_IF(
    .clk_IF(clk),
    .rst_IF(rst),
    .en_IF(1'b1),
    .PC_in_IF(Ex_reg_Mem_PC_out_EXMem),
    .PCSrc(Pipeline_Mem_PCSrc),
    .PC_out_IF(Pipeline_IF_PC_out_IF)
);

IF_reg_ID Pipeline_IF_reg_ID(
    .clk_IFID(clk),
    .rst_IFID(rst),
    .en_IFID(1'b1),
    .PC_in_IFID(Pipeline_IF_PC_out_IF),
    .inst_in_IFID(inst_IF),
    .PC_out_IFID(IF_reg_ID_PC_out_IFID),
    .inst_out_IFID(IF_reg_ID_inst_out_IFID)
);
```

```
Pipeline_ID Pipeline_CPU_ID(
    .clk_ID(clk),
    .rst_ID(rst),
    .RegWrite_in_ID(Mem_reg_WB_RegWrite_out_MemWB),
    .Rd_addr_ID(Mem_reg_WB_Rd_addr_out_MemWB),
    .Wt_data_ID(Pipeline_WB_Data_out_WB),
    .Inst_in_ID(IF_reg_ID_inst_out_IFID),
    .Rd_addr_out_ID(Pipeline_ID_Rd_addr_out_ID),
    .Rs1_out_ID(Pipeline_ID_Rs1_out_ID),
    .Rs2_out_ID(Pipeline_ID_Rs2_out_ID),
    .Imm_out_ID(Pipeline_ID_Imm_out_ID),
    .ALUSrc_B_ID(Pipeline_ID_ALUSrc_B_ID),
    .ALU_control_ID(Pipeline_ID_ALU_control_ID),
    .Branch_ID(Pipeline_ID_Branch_ID),
    .BranchN_ID(Pipeline_ID_BranchN_ID),
    .MemRW_ID(Pipeline_ID_MemRW_ID),
    .Jump_ID(Pipeline_ID_Jump_ID),
    .MemtoReg_ID(Pipeline_ID_MemtoReg_ID),
    .RegWrite_out_ID(Pipeline_ID_RegWrite_out_ID)
);

ID_reg_Ex Pipeline_ID_reg_Ex(
    .clk_IDEX(clk),
    .rst_IDEX(rst),
    .en_IDEX(1'b1),
    .PC_in_IDEX(IF_reg_ID_PC_out_IFID),
    .Rd_addr_IDEX(Pipeline_ID_Rd_addr_out_ID),
    .Rs1_in_IDEX(Pipeline_ID_Rs1_out_ID),
    .Rs2_in_IDEX(Pipeline_ID_Rs2_out_ID),
    .Imm_in_IDEX(Pipeline_ID_Imm_out_ID),
    .ALUSrc_B_in_IDEX(Pipeline_ID_ALUSrc_B_ID),
    .ALU_control_in_IDEX(Pipeline_ID_ALU_control_ID),
    .Branch_in_IDEX(Pipeline_ID_Branch_ID),
    .BranchN_in_IDEX(Pipeline_ID_BranchN_ID),
    .MemRW_in_IDEX(Pipeline_ID_MemRW_ID),
    .Jump_in_IDEX(Pipeline_ID_Jump_ID),
    .MemtoReg_in_IDEX(Pipeline_ID_MemtoReg_ID),
    .RegWrite_in_IDEX(Pipeline_ID_RegWrite_out_ID),
    .PC_out_IDEX(ID_reg_Ex_PC_out_IDEX),
    .Rd_addr_out_IDEX(ID_reg_Ex_Rd_addr_out_IDEX),
    .Rs1_out_IDEX(ID_reg_Ex_Rs1_out_IDEX),
```

```
    .Rs2_out_IDEX(ID_reg_Ex_Rs2_out_IDEX),
    .Imm_out_IDEX(ID_reg_Ex_Imm_out_IDEX),
    .ALUSrc_B_out_IDEX(ID_reg_Ex_ALUSrc_B_out_IDEX),
    .ALU_control_out_IDEX(ID_reg_Ex_ALU_control_out_IDEX),
    .Branch_out_IDEX(ID_reg_Ex_Branch_out_IDEX),
    .BranchN_out_IDEX(ID_reg_Ex_BranchN_out_IDEX),
    .MemRW_out_IDEX(ID_reg_Ex_MemRW_out_IDEX),
    .Jump_out_IDEX(ID_reg_Ex_Jump_out_IDEX),
    .MemtoReg_out_IDEX(ID_reg_Ex_MemtoReg_out_IDEX),
    .RegWrite_out_IDEX(ID_reg_Ex_RegWrite_out_IDEX)
);

Pipeline_Ex Pipeline_CPU_Ex(
    .PC_in_EX(ID_reg_Ex_PC_out_IDEX),
    .Rs1_in_EX(ID_reg_Ex_Rs1_out_IDEX),
    .Rs2_in_EX(ID_reg_Ex_Rs2_out_IDEX),
    .Imm_in_EX(ID_reg_Ex_Imm_out_IDEX),
    .ALUSrc_B_in_EX(ID_reg_Ex_ALUSrc_B_out_IDEX),
    .ALU_control_in_EX(ID_reg_Ex_ALU_control_out_IDEX),
    .PC_out_EX(Pipeline_Ex_PC_out_EX),
    .PC4_out_EX(Pipeline_Ex_PC4_out_EX),
    .zero_out_EX(Pipeline_Ex_zero_out_EX),
    .ALU_out_EX(Pipeline_Ex_ALU_out_EX),
    .Rs2_out_EX(Pipeline_Ex_Rs2_out_EX)
);

Ex_reg_Mem Pipeline_Ex_reg_Mem(
    .clk_EXMem(clk),
    .rst_EXMem(rst),
    .en_EXMem(1'b1),
    .PC_in_EXMem(Pipeline_Ex_PC_out_EX),
    .PC4_in_EXMem(Pipeline_Ex_PC4_out_EX),
    .Rd_addr_EXMem(ID_reg_Ex_Rd_addr_out_IDEX),
    .zero_in_EXMem(Pipeline_Ex_zero_out_EX),
    .ALU_in_EXMem(Pipeline_Ex_ALU_out_EX),
    .Rs2_in_EXMem(Pipeline_Ex_Rs2_out_EX),
    .Branch_in_EXMem(ID_reg_Ex_Branch_out_IDEX),
    .BranchN_in_EXMem(ID_reg_Ex_BranchN_out_IDEX),
    .MemRW_in_EXMem(ID_reg_Ex_MemRW_out_IDEX),
    .Jump_in_EXMem(ID_reg_Ex_Jump_out_IDEX),
    .MemtoReg_in_EXMem(ID_reg_Ex_MemtoReg_out_IDEX),
    .RegWrite_in_EXMem(ID_reg_Ex_RegWrite_out_IDEX),
```

```
    .PC_out_EXMem(Ex_reg_Mem_PC_out_EXMem),
    .PC4_out_EXMem(Ex_reg_Mem_PC4_out_EXMem),
    .Rd_addr_out_EXMem(Ex_reg_Mem_Rd_addr_out_EXMem),
    .zero_out_EXMem(Ex_reg_Mem_zero_out_EXMem),
    .ALU_out_EXMem(Ex_reg_Mem_ALU_out_EXMem),
    .Rs2_out_EXMem(Ex_reg_Mem_Rs2_out_EXMem),
    .Branch_out_EXMem(Ex_reg_Mem_Branch_out_EXMem),
    .BranchN_out_EXMem(Ex_reg_Mem_BranchN_out_EXMem),
    .MemRW_out_EXMem(Ex_reg_Mem_MemRW_out_EXMem),
    .Jump_out_EXMem(Ex_reg_Mem_Jump_out_EXMem),
    .MemtoReg_out_EXMem(Ex_reg_Mem_MemtoReg_out_EXMem),
    .RegWrite_out_EXMem(Ex_reg_Mem_RegWrite_out_EXMem)
);

Pipeline_Mem Pipeline_CPU_Mem(
    .zero_in_Mem(Ex_reg_Mem_zero_out_EXMem),
    .Branch_in_Mem(Ex_reg_Mem_Branch_out_EXMem),
    .BranchN_in_Mem(Ex_reg_Mem_BranchN_out_EXMem),
    .Jump_in_Mem(Ex_reg_Mem_Jump_out_EXMem),
    .PCSrc(Pipeline_Mem_PCSrc)
);

Mem_reg_WB Pipeline_Mem_reg_WB(
    .clk_MemWB(clk),
    .rst_MemWB(rst),
    .en_MemWB(1'b1),
    .PC4_in_MemWB(Ex_reg_Mem_PC4_out_EXMem),
    .Rd_addr_MemWB(Ex_reg_Mem_Rd_addr_out_EXMem),
    .ALU_in_MemWB(Ex_reg_Mem_ALU_out_EXMem),
    .DMem_data_MemWB(Data_in),
    .MemtoReg_in_MemWB(Ex_reg_Mem_MemtoReg_out_EXMem),
    .RegWrite_in_MemWB(Ex_reg_Mem_RegWrite_out_EXMem),
    .PC4_out_MemWB(Mem_reg_WB_PC4_out_MemWB),
    .Rd_addr_out_MemWB(Mem_reg_WB_Rd_addr_out_MemWB),
    .ALU_out_MemWB(Mem_reg_WB_ALU_out_MemWB),
    .DMem_data_out_MemWB(Mem_reg_WB_DMem_data_out_MemWB),
    .MemtoReg_out_MemWB(Mem_reg_WB_MemtoReg_out_MemWB),
    .RegWrite_out_MemWB(Mem_reg_WB_RegWrite_out_MemWB)
);

Pipeline_WB Pipeline_CPU_WB(
    .PC4_in_WB(Mem_reg_WB_PC4_out_MemWB),
```

```
    .ALU_in_WB(Mem_reg_WB_ALU_out_MemWB),
    .DMem_data_WB(Mem_reg_WB_DMem_data_out_MemWB),
    .MemtoReg_in_WB(Mem_reg_WB_MemtoReg_out_MemWB),
    .Data_out_WB(Pipeline_WB_Data_out_WB)
);

assign PC_out_EX = Pipeline_Ex_PC_out_EX;
assign PC_out_ID = IF_reg_ID_PC_out_IFID;
assign inst_ID = IF_reg_ID_inst_out_IFID;
assign PC_out_IF = Pipeline_IF_PC_out_IF;
assign Addr_out = Ex_reg_Mem_ALU_out_EXMem;
assign Data_out = Ex_reg_Mem_Rs2_out_EXMem;
assign Data_out_WB = Pipeline_WB_Data_out_WB;
assign MemRW_Mem = Ex_reg_Mem_MemRW_out_EXMem;
assign MemRW_EX = ID_reg_Ex_MemRW_out_IDEX;

endmodule
```

5. 由于之后需自行设计各模块，因此搭建好这个流水线 CPU 框架后我也没有做额外的测试，直接进入后续实验的设计。

4.2 取指、译码部分设计

1. 在 Vivado 上新建工程 Pipeline_IF 和 Pipeline_ID，分别表示取指部分和译码部分的实现。
2. 先来看取指部分
   - 导入在之前实验设计好的子模块 `MUX2T1_32`、`REG32` 和 `add_32`
   - 在工程中新建文件 `Pipeline_IF.v`（取指部分数据通路），按照给出的逻辑原理图，利用 Verilog 语言进行模块的调用和连接，代码如下所示：

```
`timescale 1ns / 1ps

module Pipeline_IF(
    input clk_IF,                       // 时钟
    input rst_IF,                       // 复位
    input en_IF,                        // 使能
    input [31:0] PC_in_IF,              // 取指令 PC 输入
    input PCSrc,                        // PC 输入选择
    output reg [31:0] PC_out_IF
);

    wire [31:0] add_32_c;
    wire [31:0] MUX2T1_32_o;
```

```
    wire [31:0] REG32_Q;

    MUX2T1_32 MUX2T1_32(
        .I0(add_32_c),
        .I1(PC_in_IF),
        .s(PCSrc),
        .o(MUX2T1_32_o)
    );

    REG32 PC(
        .clk(clk_IF),
        .rst(rst_IF),
        .CE(en_IF),
        .D(MUX2T1_32_o),
        .Q(REG32_Q)
    );

    add_32 add_32(
        .a(32'd4),
        .b(REG32_Q),
        .c(add_32_c)
    );

    always @(*) begin
        PC_out_IF <= REG32_Q;
    end

endmodule
```

· 然后新建文件 IF_reg_ID.v（IF/ID 流水线寄存器），它本质上是一个寄存器，因此可以仿照一般寄存器的设计进行代码编写，如下所示：

```
`timescale 1ns / 1ps

module IF_reg_ID(
    input clk_IFID,                     // 寄存器时钟
    input rst_IFID,                     // 寄存器复位
    input en_IFID,                      // 寄存器使能
    input [31:0] PC_in_IFID,            // PC 输入
    input [31:0] inst_in_IFID,          // 指令输入
    output reg [31:0] PC_out_IFID,      // PC 输出
    output reg [31:0] inst_out_IFID     // 指令输出
);
```

```
    always @(posedge clk_IFID or posedge rst_IFID) begin
        if (rst_IFID == 1) begin
            PC_out_IFID <= 32'b0;
            inst_out_IFID <= 32'b0;
        end
        else if (en_IFID == 1) begin
            PC_out_IFID <= PC_in_IFID;
            inst_out_IFID <= inst_in_IFID;
        end
    end

endmodule
```

3. 再来看译码部分
   - 导入在之前实验设计好的子模块 Reg、SCPU_ctrl 和 ImmGen（沿用 Lab4 未进行指令扩展的版本，但是 SCPU_ctrl 需要额外补上对 bne 指令的处理）
   - 在工程中新建文件 Pipeline_ID.v（译码部分数据通路），按照给出的逻辑原理图，利用 Verilog 语言进行模块的调用和连接，代码如下所示：

```
`timescale 1ns / 1ps

module Pipeline_ID(
    input clk_ID,                           // 时钟
    input rst_ID,                           // 复位
    input RegWrite_in_ID,                   // 寄存器堆使能
    input [4:0] Rd_addr_ID,                 // 写目的地址输入
    input [31:0] Wt_data_ID,                // 写数据输入
    input [31:0] Inst_in_ID,                // 指令输入
    output reg [31:0] Rd_addr_out_ID,       // 写目的地址输出
    output reg [31:0] Rs1_out_ID,           // 操作数 1 输出
    output reg [31:0] Rs2_out_ID ,          // 操作数 2 输出
    output reg [31:0] Imm_out_ID ,          // 立即数输出
    output reg ALUSrc_B_ID ,                // ALU B 端输入选择
    output reg [2:0] ALU_control_ID,        // ALU 控制
    output reg Branch_ID,                   // Beq 控制
    output reg BranchN_ID,                  // Bne 控制
    output reg MemRW_ID,                    // 存储器读写
    output reg Jump_ID,                     // Jal 控制
    output reg [1:0] MemtoReg_ID,           // 寄存器写回选择
    output reg RegWrite_out_ID              // 寄存器堆读写
);
```

```
    wire [31:0] SCPU_ctrl_ImmSel;
    wire [31:0] Regs_Rs1_data;
    wire [31:0] Regs_Rs2_data;
    wire [31:0] ImmGen_out;
    wire SCPU_ctrl_ALUSrc_B;
    wire [1:0] SCPU_ctrl_MemtoReg;
    wire SCPU_ctrl_Jump;
    wire SCPU_ctrl_Branch;
    wire SCPU_ctrl_BranchN;
    wire SCPU_ctrl_RegWrite;
    wire SCPU_ctrl_MemRW;
    wire [2:0] SCPU_ctrl_ALU_Control;

    Reg Regs_0(
        .clk(clk_ID),
        .rst(rst_ID),
        .Rs1_addr(Inst_in_ID[19:15]),
        .Rs2_addr(Inst_in_ID[24:20]),
        .Wt_addr(Rd_addr_ID),
        .Wt_data(Wt_data_ID),
        .RegWrite(RegWrite_in_ID),
        .Rs1_data(Regs_Rs1_data),
        .Rs2_data(Regs_Rs2_data)
    );

    ImmGen ImmGen_0(
        .ImmSel(SCPU_ctrl_ImmSel),
        .inst_field(Inst_in_ID),
        .Imm_out(ImmGen_out)
    );

    SCPU_ctrl SCPU_ctrl_0(
        .OPcode(Inst_in_ID[6:2]),
        .Fun3(Inst_in_ID[14:12]),
        .Fun7(Inst_in_ID[30]),
        // .MIO_ready(1'b0),
        .ImmSel(SCPU_ctrl_ImmSel),
        .ALUSrc_B(SCPU_ctrl_ALUSrc_B),
        .MemtoReg(SCPU_ctrl_MemtoReg),
        .Jump(SCPU_ctrl_Jump),
        .Branch(SCPU_ctrl_Branch),
        .BranchN(SCPU_ctrl_BranchN),
```

```
        .RegWrite(SCPU_ctrl_RegWrite),
        .MemRW(SCPU_ctrl_MemRW),
        .ALU_Control(SCPU_ctrl_ALU_Control)
        // .CPU_MIO(1'b0)
    );

    always @(*) begin
        Rs1_out_ID <= Regs_Rs1_data;
        Rs2_out_ID <= Regs_Rs2_data;
        Imm_out_ID <= ImmGen_out;
        ALUSrc_B_ID <= SCPU_ctrl_ALUSrc_B;
        MemtoReg_ID <= SCPU_ctrl_MemtoReg;
        Jump_ID <= SCPU_ctrl_Jump;
        Branch_ID <= SCPU_ctrl_Branch;
        BranchN_ID <= SCPU_ctrl_BranchN;
        RegWrite_out_ID <= SCPU_ctrl_RegWrite;
        MemRW_ID <= SCPU_ctrl_MemRW;
        ALU_control_ID <= SCPU_ctrl_ALU_Control;
        Rd_addr_out_ID <= Inst_in_ID[11:7];
    end

endmodule
```

· 然后新建文件 ID_reg_Ex.v（ID/EX 流水线寄存器），它本质上是一个寄存器，因此可以仿照一般寄存器的设计进行代码编写，如下所示：

```
`timescale 1ns / 1ps

module ID_reg_Ex(
    input clk_IDEX,                          // 寄存器时钟
    input rst_IDEX,                          // 寄存器复位
    input en_IDEX,                           // 寄存器使能
    input [31:0] PC_in_IDEX,                 // PC 输入
    input [4:0] Rd_addr_IDEX,                // 写目的地址输入
    input [31:0] Rs1_in_IDEX,                // 操作数 1 输入
    input [31:0] Rs2_in_IDEX,                // 操作数 2 输如
    input [31:0] Imm_in_IDEX,                // 立即数输入
    input ALUSrc_B_in_IDEX,                  // ALU B 输入选择
    input [2:0] ALU_control_in_IDEX,         // ALU 选择控制
    input Branch_in_IDEX,                    // Beq
    input BranchN_in_IDEX,                   // Bne
    input MemRW_in_IDEX,                     // 存储器读写
    input Jump_in_IDEX,                      // Jal
```

```
    input [1:0] MemtoReg_in_IDEX,                // 写回选择
    input RegWrite_in_IDEX,                      // 寄存器堆读写
    output reg [31:0] PC_out_IDEX,               // PC 输出
    output reg [4:0] Rd_addr_out_IDEX,           // 目的地址输出
    output reg [31:0] Rs1_out_IDEX,              // 操作数 1 输出
    output reg [31:0] Rs2_out_IDEX,              // 操作数 2 输出
    output reg [31:0] Imm_out_IDEX,              // 立即数输出
    output reg ALUSrc_B_out_IDEX,                // ALU B 选择
    output reg [2:0] ALU_control_out_IDEX,       // ALU 控制
    output reg Branch_out_IDEX,                  // Beq
    output reg BranchN_out_IDEX,                 // Bne
    output reg MemRW_out_IDEX,                   // 存储器读写
    output reg Jump_out_IDEX,                    // Jal
    output reg [1:0] MemtoReg_out_IDEX,          // 写回
    output reg RegWrite_out_IDEX                 // 寄存器堆读写
);

    always @(posedge clk_IDEX or posedge rst_IDEX) begin
        if (rst_IDEX == 1) begin
            PC_out_IDEX <= 32'b0;
            Rd_addr_out_IDEX <= 5'b0;
            Rs1_out_IDEX <= 32'b0;
            Rs2_out_IDEX <= 32'b0;
            Imm_out_IDEX <= 32'b0;
            ALUSrc_B_out_IDEX <= 1'b0;
            ALU_control_out_IDEX <= 3'b0;
            Branch_out_IDEX <= 1'b0;
            BranchN_out_IDEX <= 1'b0;
            MemRW_out_IDEX <= 1'b0;
            Jump_out_IDEX <= 1'b0;
            MemtoReg_out_IDEX <= 2'b0;
            RegWrite_out_IDEX <= 1'b0;
        end
        else if (en_IDEX == 1) begin
            PC_out_IDEX <= PC_in_IDEX;
            Rd_addr_out_IDEX <= Rd_addr_IDEX;
            Rs1_out_IDEX <= Rs1_in_IDEX;
            Rs2_out_IDEX <= Rs2_in_IDEX;
            Imm_out_IDEX <= Imm_in_IDEX;
            ALUSrc_B_out_IDEX <= ALUSrc_B_in_IDEX;
            ALU_control_out_IDEX <= ALU_control_in_IDEX;
            Branch_out_IDEX <= Branch_in_IDEX;
```

```
            BranchN_out_IDEX <= BranchN_in_IDEX;
            MemRW_out_IDEX <= MemRW_in_IDEX;
            Jump_out_IDEX <= Jump_in_IDEX;
            MemtoReg_out_IDEX <= MemtoReg_in_IDEX;
            RegWrite_out_IDEX <= RegWrite_in_IDEX;
        end
    end


endmodule
```

4. 用这些自行设计的模块替换原来在 Pipeline_CPU 中已经提供的模块

4.3 执行、访存、写回部分设计

1. 在 Vivado 上新建工程 Pipeline_Ex、Pipeline_Mem 和 Pipeline_WB，分别表示执行部分、访存部分和写回的实现。
2. 先来看执行部分
   - 导入在之前实验设计好的子模块 MUX2T1_32、add_32 和 ALU（注意 ALU 模块下也有众多子模块，不要忘记导入）
   - 在工程中新建文件 Pipeline_Ex.v（执行部分数据通路），按照给出的逻辑原理图，利用 Verilog 语言进行模块的调用和连接，代码如下所示：

```
`timescale 1ns / 1ps

module Pipeline_Ex(
    input [31:0] PC_in_EX,              // PC 输入
    input [31:0] Rs1_in_EX,             // 操作数 1 输入
    input [31:0] Rs2_in_EX,             // 操作数 2 输入
    input [31:0] Imm_in_EX,             // 立即数输入
    input ALUSrc_B_in_EX ,              // ALU B 选择
    input [2:0] ALU_control_in_EX,      // ALU 选择控制
    output reg [31:0] PC_out_EX,        // PC 输出
    output reg [31:0]PC4_out_EX,        // PC+4 输出
    output reg zero_out_EX,             // ALU 判 0 输出
    output reg [31:0] ALU_out_EX,       // ALU 计算输出
    output reg[31:0] Rs2_out_EX         // 操作数 2 输出
);

    wire [31:0] MUX2T1_32_o;
    wire [31:0] add_32_0_o;
    wire [31:0] add_32_1_o;
    wire [31:0] ALU_res;
```

```
    wire ALU_zero;


    add_32 add_32_1(
        .a(32'd4),
        .b(PC_in_EX),
        .c(add_32_1_o)
    );

    add_32 add_32_0(
        .a(PC_in_EX),
        .b(Imm_in_EX),
        .c(add_32_0_o)
    );

    MUX2T1_32 MUX2T1_32_0(
        .I0(Rs2_in_EX),
        .I1(Imm_in_EX),
        .s(ALUSrc_B_in_EX),
        .o(MUX2T1_32_o)
    );

    ALU ALU(
        .A(Rs1_in_EX),
        .ALU_operation(ALU_control_in_EX),
        .B(MUX2T1_32_o),
        .res(ALU_res),
        .zero(ALU_zero)
    );

    always @(*) begin
        PC4_out_EX <= add_32_1_o;
        PC_out_EX <= add_32_0_o;
        ALU_out_EX <= ALU_res;
        zero_out_EX <= ALU_zero;
        Rs2_out_EX <= Rs2_in_EX;
    end

endmodule
```

- 然后新建文件 Ex_reg_Mem.v（EX/Mem 流水线寄存器），它本质上是一个寄存器，因此可以仿照一般寄存器的设计进行代码编写，如下所示：

```
`timescale 1ns / 1ps

module Ex_reg_Mem(
    input clk_EXMem,                        // 寄存器时钟
    input rst_EXMem,                        // 寄存器复位
    input en_EXMem,                         // 寄存器使能
    input [31:0] PC_in_EXMem,               // PC 输入
    input [31:0] PC4_in_EXMem,              // PC+4 输入
    input [4:0] Rd_addr_EXMem,              // 写目的寄存器地址输入
    input zero_in_EXMem,                    // zero
    input [31:0] ALU_in_EXMem,              // ALU 输入
    input [31:0] Rs2_in_EXMem,              // 操作数 2 输入
    input Branch_in_EXMem,                  // Beq
    input BranchN_in_EXMem,                 // Bne
    input MemRW_in_EXMem,                   // 存储器读写
    input Jump_in_EXMem,                    // Jal
    input [1:0] MemtoReg_in_EXMem,          // 写回
    input RegWrite_in_EXMem,                // 寄存器堆读写
    output reg [31:0] PC_out_EXMem,         // PC 输出
    output reg [31:0] PC4_out_EXMem,        // PC+4 输出
    output reg [4:0] Rd_addr_out_EXMem,     // 写目的寄存器输出
    output reg zero_out_EXMem,              // zero
    output reg [31:0] ALU_out_EXMem,        // ALU 输出
    output reg [31:0] Rs2_out_EXMem,        // 操作数 2 输出
    output reg Branch_out_EXMem,            // Beq
    output reg BranchN_out_EXMem,           // Bne
    output reg MemRW_out_EXMem,             // 存储器读写
    output reg Jump_out_EXMem,              // Jal
    output reg [1:0] MemtoReg_out_EXMem,    // 写回
    output reg RegWrite_out_EXMem           // 寄存器堆读写
);

    always @(posedge clk_EXMem or posedge rst_EXMem) begin
        if (rst_EXMem == 1) begin
            PC_out_EXMem <= 32'b0;
            PC4_out_EXMem <= 32'b0;
            Rd_addr_out_EXMem <= 5'b0;
            zero_out_EXMem <= 1'b0;
            ALU_out_EXMem <= 32'b0;
            Rs2_out_EXMem <= 32'b0;
            Branch_out_EXMem <= 1'b0;
            BranchN_out_EXMem <= 1'b0;
```

```
            MemRW_out_EXMem <= 1'b0;
            Jump_out_EXMem <= 1'b0;
            MemtoReg_out_EXMem <= 1'b0;
            RegWrite_out_EXMem <= 1'b0;
        end
        else if (en_EXMem == 1) begin
            PC_out_EXMem <= PC_in_EXMem;
            PC4_out_EXMem <= PC4_in_EXMem;
            Rd_addr_out_EXMem <= Rd_addr_EXMem;
            zero_out_EXMem <= zero_in_EXMem;
            ALU_out_EXMem <= ALU_in_EXMem;
            Rs2_out_EXMem <= Rs2_in_EXMem;
            Branch_out_EXMem <= Branch_in_EXMem;
            BranchN_out_EXMem <= BranchN_in_EXMem;
            MemRW_out_EXMem <= MemRW_in_EXMem;
            Jump_out_EXMem <= Jump_in_EXMem;
            MemtoReg_out_EXMem <= MemtoReg_in_EXMem;
            RegWrite_out_EXMem <= RegWrite_in_EXMem;
        end
    end

endmodule
```

3. 再来看访存部分
   - 在工程中新建文件 Pipeline_Mem.v（访存部分数据通路），按照给出的逻辑原理图，利用 Verilog 语言进行模块的调用和连接，代码如下所示：

```
`timescale 1ns / 1ps

module Pipeline_Mem(
    input zero_in_Mem,        // zero
    input Branch_in_Mem,      // beq
    input BranchN_in_Mem,     // bne
    input Jump_in_Mem,        // jal
    output PCSrc              // PC 选择控制输出
);

assign PCSrc = Jump_in_Mem | Branch_in_Mem & zero_in_Mem |
BranchN_in_Mem & ~zero_in_Mem;

endmodule
```

· 然后新建文件 Mem_reg_WB.v（Mem/WB 流水线寄存器），它本质上是一个寄存器，因此可以仿照一般寄存器的设计进行代码编写，如下所示：

```
`timescale 1ns / 1ps

module Mem_reg_WB(
    input clk_MemWB,                        // 寄存器时
    input rst_MemWB,                        // 寄存器复位
    input en_MemWB,                         // 寄存器使能
    input [31:0] PC4_in_MemWB,              // PC+4 输入
    input [4:0] Rd_addr_MemWB,              // 写目的地址输入
    input [31:0] ALU_in_MemWB,              // ALU 输入
    input [31:0] DMem_data_MemWB,           // 存储器数据输入
    input [1:0] MemtoReg_in_MemWB,          // 写回
    input RegWrite_in_MemWB,                // 寄存器堆读写
    output reg [31:0] PC4_out_MemWB,        // PC+4 输出
    output reg [4:0] Rd_addr_out_MemWB,     // 写目的地址输出
    output reg [31:0] ALU_out_MemWB,        // ALU 输出
    output reg [31:0] DMem_data_out_MemWB,  // 存储器数据输出
    output reg [1:0] MemtoReg_out_MemWB,    // 写回
    output reg RegWrite_out_MemWB           // 寄存器堆读写
);

    always @(posedge clk_MemWB or posedge rst_MemWB) begin
        if (rst_MemWB == 1) begin
            PC4_out_MemWB <= 32'b0;
            Rd_addr_out_MemWB <= 5'b0;
            ALU_out_MemWB <= 32'b0;
            DMem_data_out_MemWB <= 32'b0;
            MemtoReg_out_MemWB <= 2'b0;
            RegWrite_out_MemWB <= 1'b0;
        end
        else if (en_MemWB == 1) begin
            PC4_out_MemWB <= PC4_in_MemWB;
            Rd_addr_out_MemWB <= Rd_addr_MemWB;
            ALU_out_MemWB <= ALU_in_MemWB;
            DMem_data_out_MemWB <= DMem_data_MemWB;
            MemtoReg_out_MemWB <= MemtoReg_in_MemWB;
            RegWrite_out_MemWB <= RegWrite_in_MemWB;
        end
    end

endmodule
```

4. 最后来看写回部分
   - 导入在之前实验设计好的子模块 `MUX4T1_32`
   - 在工程中新建文件 `Pipeline_WB.v`（写回部分数据通路），按照给出的逻辑原理图，利用 Verilog 语言进行模块的调用和连接，代码如下所示：

```
`timescale 1ns / 1ps

module Pipeline_WB(
    input [31:0] PC4_in_WB,         // PC+4 输入
    input [31:0] ALU_in_WB,         // ALU 结果输出
    input [31:0] DMem_data_WB,      // 存储器数据输入
    input [1:0] MemtoReg_in_WB,     // 写回选择控制
    output [31:0] Data_out_WB       // 写回数据输出
);

    MUX4T1_32 MUX4T1_32_0(
        .s(MemtoReg_in_WB),
        .I0(ALU_in_WB),
        .I1(DMem_data_WB),
        .I2(PC4_in_WB),
        .I3(PC4_in_WB),
        .o(Data_out_WB)
    );

endmodule
```

5. 用这些自行设计的模块替换原来在 `Pipeline_CPU` 中已经提供的模块
6. 仿照 Lab4 搭建仿真平台，包括了 RAM（对应的 coe 文件用于初始化数据，在 Lab0 中已给出）、ROM（用于存储 RISC-V 指令的机器码，可自行设计）搭建一个仿真平台，检验我们设计的流水线 CPU 的功能正确性。仿真平台的代码如下所示（在 Lab4 的基础上稍作修改）：

```
`timescale 1ns / 1ps

module soc_test_wrapper(
    input clk,
    input rst
);

    wire [31:0] RAM_B_douta;
    wire [31:0] CPU_inst_in;
    wire [31:0] CPU_Addr_out;
    wire CPU_MemRW;
```

```
    wire [31:0] CPU_Data_out;
    wire [31:0] CPU_PC_out;
    wire [9:0] Addr_out_slice;
    wire [9:0] PC_out_slice;
    wire [31:0] CPU_PC_out_ID;
    wire [31:0] CPU_PC_out_Ex;
    wire [31:0] CPU_inst_ID;
    wire CPU_MemRW_Ex;

    RAM_B RAM_B_0(
        .clka(~clk),
        .wea(CPU_MemRW),
        .addra(Addr_out_slice),
        .dina(CPU_Data_out),
        .douta(RAM_B_douta)
    );

    ROM ROM_0(
        .a(PC_out_slice),
        .spo(CPU_inst_in)
    );

    Pipeline_CPU Pipeline_CPU_wrapper_0(
        .clk(clk),
        .rst(rst),
        .Data_in(RAM_B_douta),
        .inst_IF(CPU_inst_in),
        .PC_out_IF(CPU_PC_out),
        .PC_out_ID(CPU_PC_out_ID),
        .inst_ID(CPU_inst_ID),
        .PC_out_EX(CPU_PC_out_Ex),
        .MemRW_EX(CPU_MemRW_Ex),
        .MemRW_Mem(CPU_MemRW),
        .Addr_out(CPU_Addr_out),
        .Data_out(CPU_Data_out)
    );

    assign Addr_out_slice = CPU_Addr_out[11:2];
    assign PC_out_slice = CPU_PC_out[11:2];

endmodule
```

此时文件结构如下所示:

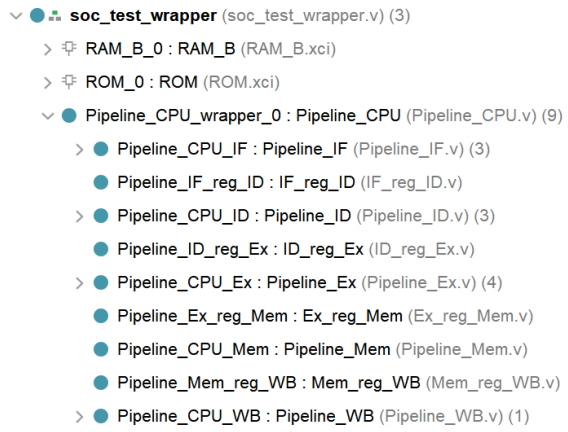


Figure 10: 流水线 CPU 仿真平台对应的文件结构

7. 通过与 Lab4 相同的仿真代码, 对仿真平台进行仿真, 观察波形图是否符合预期变化, 从而判断 CPU 功能的正确与否（波形图及其分析将在“实验结果与分析”一节给出）

8. 用我们刚刚设计好的整个 Pipeline_CPU 模块替换 Lab2 的 SOC 系统中预先提供给我们的 SCPU 模块；并且替换 VGA 相关的代码文件，以便在上板实验时正确显示流水线 CPU 相关的调试信息。此时文件结构如下所示：

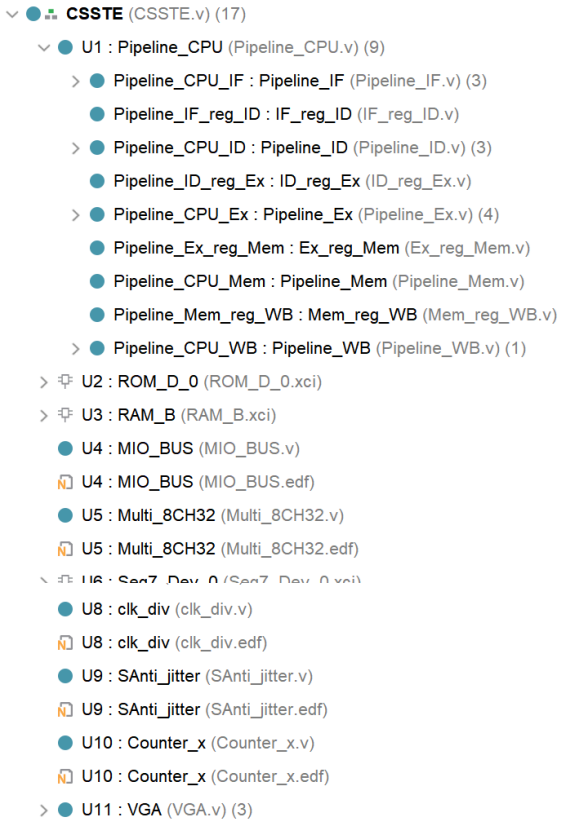


Figure 11: 加入流水线 CPU 后的 SOC 对应的文件结构

9. 对 SOC 系统工程进行综合、实现，烧录为 bit 流文件，然后上板验证，通过 VGA 观察结果是否符合预期，并需要设计测试记录表格，记录程序的 VGA 输出。

4.4 冒险与停顿处理

1. 在 Vivado 上新建工程 Pipeline_stall，用于实现冒险停顿相关的功能。
   · 在工程内新建文件 stall.v，用于处理停顿，代码如下所示：

```
`timescale 1ns / 1ps

module stall(
    input rst_stall,                    // 复位
    input RegWrite_out_IDEX,            // 执行阶段寄存器写控制
    input [4:0] Rd_addr_out_IDEX,       // 执行阶段寄存器写地址
    input RegWrite_out_EXMem,           // 访存阶段寄存器写控制
    input [4:0] Rd_addr_out_EXMem,      // 访存阶段寄存器写地址
    input [4:0] Rs1_addr_ID,            // 译码阶段寄存器读地址 1
    input [4:0] Rs2_addr_ID,            // 译码阶段寄存器读地址 2
    input Rs1_used,                     // Rs1 被使用
    input Rs2_used,                     // Rs2 被使用
    input Branch_ID,                    // 译码阶段 beq
    input BranchN_ID,                   // 译码阶段 bne
    input Jump_ID,                      // 译码阶段 jal
    input Branch_out_IDEX,              // 执行阶段 beq
    input BranchN_out_IDEX,             // 执行阶段 bne
    input Jump_out_IDEX,                // 执行阶段 jal
    input Branch_out_EXMem,             // 访存阶段 beq
    input BranchN_out_EXMem,            // 访存阶段 bne
    input Jump_out_EXMem,               // 访存阶段 jal
    output en_IF,                       // 流水线寄存器的使能及 NOP 信号
    output en_IFID,
    output NOP_IFID,
    output NOP_IDEX
);

reg Data_stall;
reg Control_stall;
reg enIF;
reg enIFID;
reg NOPIFID;
reg NOPIDEX;

always @(*) begin
    // Data hazard
    // MEM hazard
     if (RegWrite_out_EXMem && Rs1_used && Rs1_addr_ID != 0 &&
```

```
Rd_addr_out_EXMem == Rs1_addr_ID)
        Data_stall = 1;
    else if (RegWrite_out_EXMem && Rs2_used && Rs2_addr_ID != 0
&& Rd_addr_out_EXMem == Rs2_addr_ID)
        Data_stall = 1;
    // EX hazard
    else if (RegWrite_out_IDEX && Rs1_used && Rs1_addr_ID != 0 &&
Rd_addr_out_IDEX == Rs1_addr_ID)
        Data_stall = 1;
    else if (RegWrite_out_IDEX && Rs2_used && Rs2_addr_ID != 0 &&
Rd_addr_out_IDEX == Rs2_addr_ID)
        Data_stall = 1;
    else
        Data_stall = 0;

    if (Data_stall) begin
        enIF = 0;
        enIFID = 0;
        NOPIDEX = 1;
    end else begin
        enIF = 1;
        enIFID = 1;
        NOPIDEX = 0;
    end

    // Control hazard
    if ((Branch_ID == 1 || BranchN_ID == 1 || Jump_ID == 1) ||
(Branch_out_IDEX == 1 || BranchN_out_IDEX == 1 || Jump_out_IDEX ==
1) || (Branch_out_EXMem == 1 || BranchN_out_EXMem || Jump_out_EXMem
== 1))
        Control_stall = 1;
    else
        Control_stall = 0;

    if (Control_stall) begin
        NOPIFID = 1;
    end else begin
        NOPIFID = 0;
    end
end

assign en_IF = enIF;
```

```
assign en_IFID = enIFID;
assign NOP_IFID = NOPIFID;
assign NOP_IDEX = NOPIDEX;

endmodule
```

2. 由于新增了与停顿处理相关的控制信号，因此需要对流水线 CPU 的一些原有模块（ID 数据通路，以及四个流水线寄存器）做一些调整。但由于调整内容不大（仅额外增加对停顿控制信号的处理），故这里不列出修改过后的代码。

3. 此外，整个流水线 CPU 的顶层模块也要进行调整，代码如下所示：

```
`timescale 1ns / 1ps

module Pipeline_CPU(
    input clk,                  // 时钟
    input rst,                  // 复位
    input [31:0] Data_in,       // 存储器数据输入
    input [31:0] inst_IF,       // 取指阶段指令
    output [31:0] PC_out_IF,    // 取指阶段 PC 输出
    output [31:0] PC_out_ID,    // 译码阶段 PC 输出
    output [31:0] inst_ID,      // 译码阶段指令
    output [31:0] PC_out_EX,    // 执行阶段 PC 输出
    output [31:0] MemRW_EX,     // 执行阶段存储器读写
    output [31:0] MemRW_Mem,    // 访存阶段存储器读写
    output [31:0] Addr_out,     // 地址输出
    output [31:0] Data_out,     // CPU 数据输出
    output [31:0] Data_out_WB   // 写回数据输出
);

wire Pipeline_Mem_PCSrc;
wire [31:0] Pipeline_IF_PC_out_IF;
wire [31:0] IF_reg_ID_PC_out_IFID;
wire [31:0] IF_reg_ID_inst_out_IFID;
wire [31:0] Pipeline_WB_Data_out_WB;
wire [31:0] Pipeline_ID_Rd_addr_out_ID;
wire [31:0] Pipeline_ID_Rs1_out_ID;
wire [31:0] Pipeline_ID_Rs2_out_ID;
wire [31:0] Pipeline_ID_Imm_out_ID;
wire Pipeline_ID_ALUSrc_B_ID;
wire [2:0] Pipeline_ID_ALU_control_ID;
wire Pipeline_ID_Branch_ID;
wire Pipeline_ID_BranchN_ID;
```

```
wire Pipeline_ID_MemRW_ID;
wire Pipeline_ID_Jump_ID;
wire [1:0] Pipeline_ID_MemtoReg_ID;
wire Pipeline_ID_RegWrite_out_ID;
wire [31:0] ID_reg_Ex_PC_out_IDEX;
wire [4:0] ID_reg_Ex_Rd_addr_out_IDEX;
wire [31:0] ID_reg_Ex_Rs1_out_IDEX;
wire [31:0] ID_reg_Ex_Rs2_out_IDEX;
wire [31:0] ID_reg_Ex_Imm_out_IDEX;
wire ID_reg_Ex_ALUSrc_B_out_IDEX;
wire [2:0] ID_reg_Ex_ALU_control_out_IDEX;
wire ID_reg_Ex_Branch_out_IDEX;
wire ID_reg_Ex_BranchN_out_IDEX;
wire ID_reg_Ex_MemRW_out_IDEX;
wire ID_reg_Ex_Jump_out_IDEX;
wire [1:0] ID_reg_Ex_MemtoReg_out_IDEX;
wire ID_reg_Ex_RegWrite_out_IDEX;
wire [31:0] Pipeline_Ex_PC_out_EX;
wire [31:0] Pipeline_Ex_PC4_out_EX;
wire Pipeline_Ex_zero_out_EX;
wire [31:0] Pipeline_Ex_ALU_out_EX;
wire [31:0] Pipeline_Ex_Rs2_out_EX;
wire [31:0] Ex_reg_Mem_PC_imm_out_EXMem;
wire [31:0] Ex_reg_Mem_PC4_out_EXMem;
wire [4:0] Ex_reg_Mem_Rd_addr_out_EXMem;
wire Ex_reg_Mem_zero_out_EXMem;
wire [31:0] Ex_reg_Mem_ALU_out_EXMem;
wire [31:0] Ex_reg_Mem_Rs2_out_EXMem;
wire Ex_reg_Mem_Branch_out_EXMem;
wire Ex_reg_Mem_BranchN_out_EXMem;
wire Ex_reg_Mem_MemRW_out_EXMem;
wire Ex_reg_Mem_Jump_out_EXMem;
wire [1:0] Ex_reg_Mem_MemtoReg_out_EXMem;
wire Ex_reg_Mem_RegWrite_out_EXMem;
wire [31:0] Mem_reg_WB_PC4_out_MemWB;
wire [4:0] Mem_reg_WB_Rd_addr_out_MemWB;
wire [31:0] Mem_reg_WB_ALU_out_MemWB;
wire [31:0] Mem_reg_WB_DMem_data_out_MemWB;
wire [1:0] Mem_reg_WB_MemtoReg_out_MemWB;
wire Mem_reg_WB_RegWrite_out_MemWB;
wire stall_en_IF;
wire stall_en_IFID;
```

```
wire stall_NOP_IDEX;
wire stall_NOP_IFID;
wire IF_reg_ID_valid_IFID;
wire [4:0] Pipeline_ID_Rs1_addr_ID;
wire [4:0] Pipeline_ID_Rs2_addr_ID;
wire Pipeline_ID_Rs1_used;
wire Pipeline_ID_Rs2_used;
wire [31:0] ID_reg_Ex_Inst_out_IDEX;
wire ID_reg_Ex_valid_out_IDEX;
wire [31:0] Ex_reg_Mem_PC_out_EXMem;
wire [31:0] Ex_reg_Mem_Inst_out_EXMem;
wire Ex_reg_Mem_valid_out_EXMem;


Pipeline_IF Pipeline_CPU_IF(
    .clk_IF(clk),
    .rst_IF(rst),
    .en_IF(stall_en_IF),
    .PC_in_IF(Ex_reg_Mem_PC_imm_out_EXMem),
    .PCSrc(Pipeline_Mem_PCSrc),
    .PC_out_IF(Pipeline_IF_PC_out_IF)
);

IF_reg_ID Pipeline_IF_reg_ID(
    .clk_IFID(clk),
    .rst_IFID(rst),
    .en_IFID(stall_en_IFID),
    .PC_in_IFID(Pipeline_IF_PC_out_IF),
    .inst_in_IFID(inst_IF),
    .NOP_IFID(stall_NOP_IFID),
    .PC_out_IFID(IF_reg_ID_PC_out_IFID),
    .inst_out_IFID(IF_reg_ID_inst_out_IFID),
    .valid_IFID(IF_reg_ID_valid_IFID)
);

Pipeline_ID Pipeline_CPU_ID(
    .clk_ID(clk),
    .rst_ID(rst),
    .RegWrite_in_ID(Mem_reg_WB_RegWrite_out_MemWB),
    .Rd_addr_ID(Mem_reg_WB_Rd_addr_out_MemWB),
    .Wt_data_ID(Pipeline_WB_Data_out_WB),
    .Inst_in_ID(IF_reg_ID_inst_out_IFID),
```

```
    .Rd_addr_out_ID(Pipeline_ID_Rd_addr_out_ID),
    .Rs1_out_ID(Pipeline_ID_Rs1_out_ID),
    .Rs2_out_ID(Pipeline_ID_Rs2_out_ID),
    .Rs1_addr_ID(Pipeline_ID_Rs1_addr_ID),
    .Rs2_addr_ID(Pipeline_ID_Rs2_addr_ID),
    .Rs1_used(Pipeline_ID_Rs1_used),
    .Rs2_used(Pipeline_ID_Rs2_used),
    .Imm_out_ID(Pipeline_ID_Imm_out_ID),
    .ALUSrc_B_ID(Pipeline_ID_ALUSrc_B_ID),
    .ALU_control_ID(Pipeline_ID_ALU_control_ID),
    .Branch_ID(Pipeline_ID_Branch_ID),
    .BranchN_ID(Pipeline_ID_BranchN_ID),
    .MemRW_ID(Pipeline_ID_MemRW_ID),
    .Jump_ID(Pipeline_ID_Jump_ID),
    .MemtoReg_ID(Pipeline_ID_MemtoReg_ID),
    .RegWrite_out_ID(Pipeline_ID_RegWrite_out_ID)
);

ID_reg_Ex Pipeline_ID_reg_Ex(
    .clk_IDEX(clk),
    .rst_IDEX(rst),
    .en_IDEX(1'b1),
    .NOP_IDEX(stall_NOP_IDEX),
    .valid_in_IDEX(IF_reg_ID_valid_IFID),
    .PC_in_IDEX(IF_reg_ID_PC_out_IFID),
    .Inst_in_IDEX(IF_reg_ID_inst_out_IFID),
    .Rd_addr_IDEX(Pipeline_ID_Rd_addr_out_ID),
    .Rs1_in_IDEX(Pipeline_ID_Rs1_out_ID),
    .Rs2_in_IDEX(Pipeline_ID_Rs2_out_ID),
    .Imm_in_IDEX(Pipeline_ID_Imm_out_ID),
    .ALUSrc_B_in_IDEX(Pipeline_ID_ALUSrc_B_ID),
    .ALU_control_in_IDEX(Pipeline_ID_ALU_control_ID),
    .Branch_in_IDEX(Pipeline_ID_Branch_ID),
    .BranchN_in_IDEX(Pipeline_ID_BranchN_ID),
    .MemRW_in_IDEX(Pipeline_ID_MemRW_ID),
    .Jump_in_IDEX(Pipeline_ID_Jump_ID),
    .MemtoReg_in_IDEX(Pipeline_ID_MemtoReg_ID),
    .RegWrite_in_IDEX(Pipeline_ID_RegWrite_out_ID),
    .PC_out_IDEX(ID_reg_Ex_PC_out_IDEX),
    .Inst_out_IDEX(ID_reg_Ex_Inst_out_IDEX),
    .Rd_addr_out_IDEX(ID_reg_Ex_Rd_addr_out_IDEX),
    .Rs1_out_IDEX(ID_reg_Ex_Rs1_out_IDEX),
```

```
    .Rs2_out_IDEX(ID_reg_Ex_Rs2_out_IDEX),
    .Imm_out_IDEX(ID_reg_Ex_Imm_out_IDEX),
    .ALUSrc_B_out_IDEX(ID_reg_Ex_ALUSrc_B_out_IDEX),
    .ALU_control_out_IDEX(ID_reg_Ex_ALU_control_out_IDEX),
    .Branch_out_IDEX(ID_reg_Ex_Branch_out_IDEX),
    .BranchN_out_IDEX(ID_reg_Ex_BranchN_out_IDEX),
    .MemRW_out_IDEX(ID_reg_Ex_MemRW_out_IDEX),
    .Jump_out_IDEX(ID_reg_Ex_Jump_out_IDEX),
    .MemtoReg_out_IDEX(ID_reg_Ex_MemtoReg_out_IDEX),
    .RegWrite_out_IDEX(ID_reg_Ex_RegWrite_out_IDEX),
    .valid_out_IDEX(ID_reg_Ex_valid_out_IDEX)
);

Pipeline_Ex Pipeline_CPU_Ex(
    .PC_in_EX(ID_reg_Ex_PC_out_IDEX),
    .Rs1_in_EX(ID_reg_Ex_Rs1_out_IDEX),
    .Rs2_in_EX(ID_reg_Ex_Rs2_out_IDEX),
    .Imm_in_EX(ID_reg_Ex_Imm_out_IDEX),
    .ALUSrc_B_in_EX(ID_reg_Ex_ALUSrc_B_out_IDEX),
    .ALU_control_in_EX(ID_reg_Ex_ALU_control_out_IDEX),
    .PC_out_EX(Pipeline_Ex_PC_out_EX),
    .PC4_out_EX(Pipeline_Ex_PC4_out_EX),
    .zero_out_EX(Pipeline_Ex_zero_out_EX),
    .ALU_out_EX(Pipeline_Ex_ALU_out_EX),
    .Rs2_out_EX(Pipeline_Ex_Rs2_out_EX)
);

Ex_reg_Mem Pipeline_Ex_reg_Mem(
    .clk_EXMem(clk),
    .rst_EXMem(rst),
    .en_EXMem(1'b1),
    .PC_imm_EXMem(Pipeline_Ex_PC_out_EX),
    .PC4_in_EXMem(Pipeline_Ex_PC4_out_EX),
    .PC_in_EXMem(ID_reg_Ex_PC_out_IDEX),
    .valid_in_EXMem(ID_reg_Ex_valid_out_IDEX),
    .Inst_in_EXMem(ID_reg_Ex_Inst_out_IDEX),
    .Rd_addr_EXMem(ID_reg_Ex_Rd_addr_out_IDEX),
    .zero_in_EXMem(Pipeline_Ex_zero_out_EX),
    .ALU_in_EXMem(Pipeline_Ex_ALU_out_EX),
    .Rs2_in_EXMem(Pipeline_Ex_Rs2_out_EX),
    .Branch_in_EXMem(ID_reg_Ex_Branch_out_IDEX),
    .BranchN_in_EXMem(ID_reg_Ex_BranchN_out_IDEX),
```

```
    .MemRW_in_EXMem(ID_reg_Ex_MemRW_out_IDEX),
    .Jump_in_EXMem(ID_reg_Ex_Jump_out_IDEX),
    .MemtoReg_in_EXMem(ID_reg_Ex_MemtoReg_out_IDEX),
    .RegWrite_in_EXMem(ID_reg_Ex_RegWrite_out_IDEX),
    .PC_imm_out_EXMem(Ex_reg_Mem_PC_imm_out_EXMem),
    .PC4_out_EXMem(Ex_reg_Mem_PC4_out_EXMem),
    .PC_out_EXMem(Ex_reg_Mem_PC_out_EXMem),
    .valid_out_EXMem(Ex_reg_Mem_valid_out_EXMem),
    .Inst_out_EXMem(Ex_reg_Mem_Inst_out_EXMem),
    .Rd_addr_out_EXMem(Ex_reg_Mem_Rd_addr_out_EXMem),
    .zero_out_EXMem(Ex_reg_Mem_zero_out_EXMem),
    .ALU_out_EXMem(Ex_reg_Mem_ALU_out_EXMem),
    .Rs2_out_EXMem(Ex_reg_Mem_Rs2_out_EXMem),
    .Branch_out_EXMem(Ex_reg_Mem_Branch_out_EXMem),
    .BranchN_out_EXMem(Ex_reg_Mem_BranchN_out_EXMem),
    .MemRW_out_EXMem(Ex_reg_Mem_MemRW_out_EXMem),
    .Jump_out_EXMem(Ex_reg_Mem_Jump_out_EXMem),
    .MemtoReg_out_EXMem(Ex_reg_Mem_MemtoReg_out_EXMem),
    .RegWrite_out_EXMem(Ex_reg_Mem_RegWrite_out_EXMem)
);

Pipeline_Mem Pipeline_CPU_Mem(
    .zero_in_Mem(Ex_reg_Mem_zero_out_EXMem),
    .Branch_in_Mem(Ex_reg_Mem_Branch_out_EXMem),
    .BranchN_in_Mem(Ex_reg_Mem_BranchN_out_EXMem),
    .Jump_in_Mem(Ex_reg_Mem_Jump_out_EXMem),
    .PCSrc(Pipeline_Mem_PCSrc)
);

Mem_reg_WB Pipeline_Mem_reg_WB(
    .clk_MemWB(clk),
    .rst_MemWB(rst),
    .en_MemWB(1'b1),
    .PC4_in_MemWB(Ex_reg_Mem_PC4_out_EXMem),
    .PC_in_MemWB(Ex_reg_Mem_PC_out_EXMem),
    .Inst_in_MemWB(Ex_reg_Mem_Inst_out_EXMem),
    .valid_in_MemWB(Ex_reg_Mem_valid_out_EXMem),
    .Rd_addr_MemWB(Ex_reg_Mem_Rd_addr_out_EXMem),
    .ALU_in_MemWB(Ex_reg_Mem_ALU_out_EXMem),
    .DMem_data_MemWB(Data_in),
    .MemtoReg_in_MemWB(Ex_reg_Mem_MemtoReg_out_EXMem),
    .RegWrite_in_MemWB(Ex_reg_Mem_RegWrite_out_EXMem),
```

```
    .PC4_out_MemWB(Mem_reg_WB_PC4_out_MemWB),
    // .PC_out_MemWB(Mem_reg_WB_PC_out_MemWB),
    // .Inst_out_MemWB(Mem_reg_WB_Inst_out_MemWB),
    // .valid_out_MemWB(Mem_reg_WB_valid_out_WB),
    .Rd_addr_out_MemWB(Mem_reg_WB_Rd_addr_out_MemWB),
    .ALU_out_MemWB(Mem_reg_WB_ALU_out_MemWB),
    .DMem_data_out_MemWB(Mem_reg_WB_DMem_data_out_MemWB),
    .MemtoReg_out_MemWB(Mem_reg_WB_MemtoReg_out_MemWB),
    .RegWrite_out_MemWB(Mem_reg_WB_RegWrite_out_MemWB)
);

Pipeline_WB Pipeline_CPU_WB(
    .PC4_in_WB(Mem_reg_WB_PC4_out_MemWB),
    .ALU_in_WB(Mem_reg_WB_ALU_out_MemWB),
    .DMem_data_WB(Mem_reg_WB_DMem_data_out_MemWB),
    .MemtoReg_in_WB(Mem_reg_WB_MemtoReg_out_MemWB),
    .Data_out_WB(Pipeline_WB_Data_out_WB)
);

stall Pipeline_stall(
    .rst_stall(rst),
    .Rs1_addr_ID(Pipeline_ID_Rs1_addr_ID),
    .Rs2_addr_ID(Pipeline_ID_Rs2_addr_ID),
    .RegWrite_out_IDEX(ID_reg_Ex_RegWrite_out_IDEX),
    .Rd_addr_out_IDEX(ID_reg_Ex_Rd_addr_out_IDEX),
    .RegWrite_out_EXMem(Ex_reg_Mem_RegWrite_out_EXMem),
    .Rd_addr_out_EXMem(Ex_reg_Mem_Rd_addr_out_EXMem),
    .Rs1_used(Pipeline_ID_Rs1_used),
    .Rs2_used(Pipeline_ID_Rs2_used),
    .Branch_ID(Pipeline_ID_Branch_ID),
    .BranchN_ID(Pipeline_ID_BranchN_ID),
    .Jump_ID(Pipeline_ID_Jump_ID),
    .Branch_out_IDEX(ID_reg_Ex_Branch_out_IDEX),
    .BranchN_out_IDEX(ID_reg_Ex_BranchN_out_IDEX),
    .Jump_out_IDEX(ID_reg_Ex_Jump_out_IDEX),
    .Branch_out_EXMem(Ex_reg_Mem_Branch_out_EXMem),
    .BranchN_out_EXMem(Ex_reg_Mem_BranchN_out_EXMem),
    .Jump_out_EXMem(Ex_reg_Mem_Jump_out_EXMem),
    .en_IF(stall_en_IF),
    .en_IFID(stall_en_IFID),
    .NOP_IDEX(stall_NOP_IDEX),
    .NOP_IFID(stall_NOP_IFID)
```

```
);

assign PC_out_EX = Pipeline_Ex_PC_out_EX;
assign PC_out_ID = IF_reg_ID_PC_out_IFID;
assign inst_ID = IF_reg_ID_inst_out_IFID;
assign PC_out_IF = Pipeline_IF_PC_out_IF;
assign Addr_out = Ex_reg_Mem_ALU_out_EXMem;
assign Data_out = Ex_reg_Mem_Rs2_out_EXMem;
assign Data_out_WB = Pipeline_WB_Data_out_WB;
assign MemRW_Mem = Ex_reg_Mem_MemRW_out_EXMem;
assign MemRW_EX = ID_reg_Ex_MemRW_out_IDEX;

endmodule
```

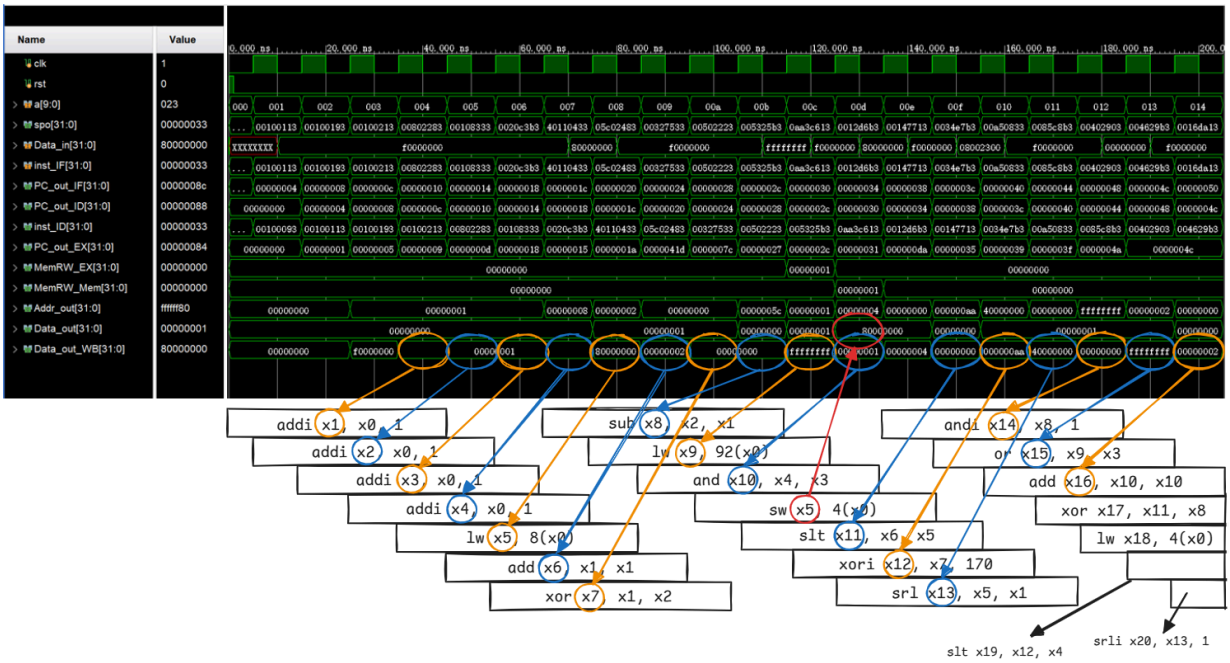
4. 仿照上面介绍过的步骤，先搭建仿真平台进行仿真测试（波形图及分析放在“实验结果与分析”一节），然后再上板检验，看结果是否符合预期。

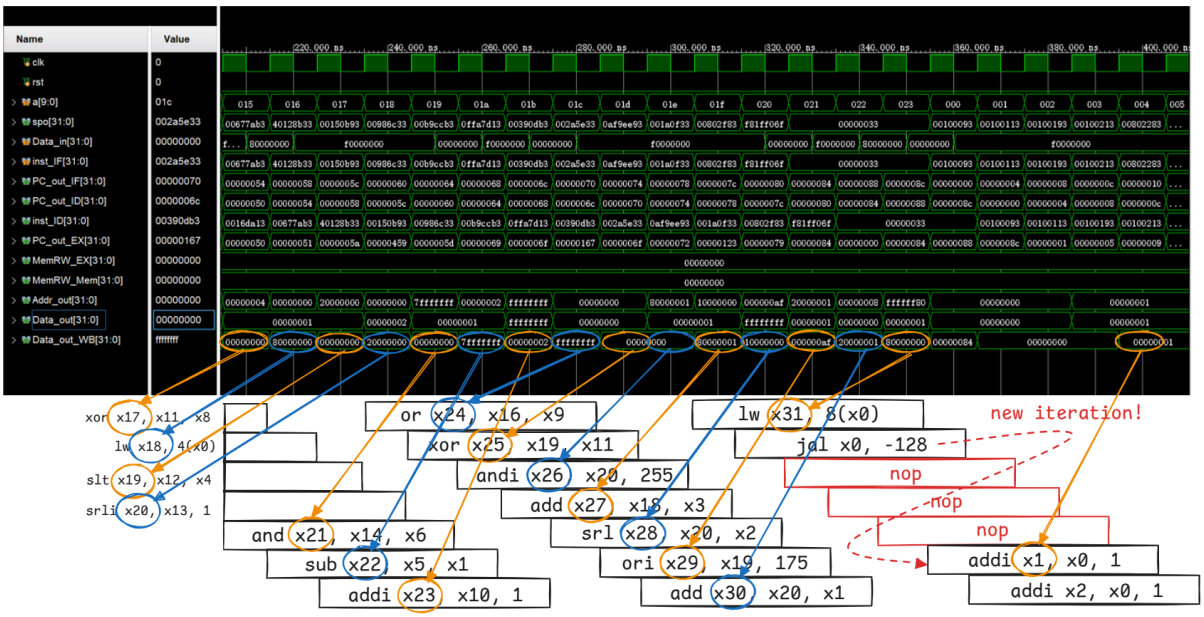
五、 实验结果与分析

5.1 基础流水线 CPU

> 注意
>
> 这里的结果包含了实验的前三个部分，即所有的模块都是自己搭建的。

5.1.1 仿真结果分析





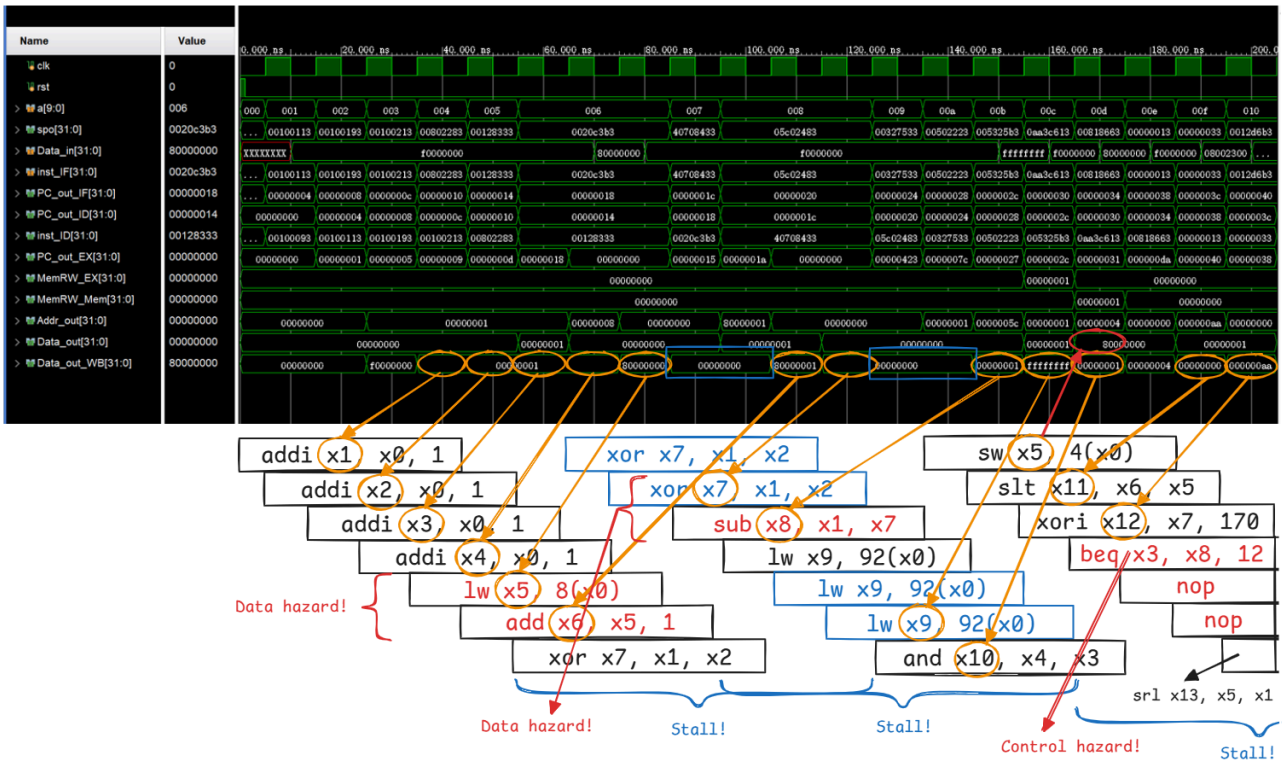
上图展示了流水线 CPU 在执行 P.coe 文件内的指令时的对应的仿真波形图，同时附上简要的分析（主要观察 `Data_out_WB` 输入端口的结果），发现结果符合预期（即符合 p_mem.pdf 文件所示结果）。

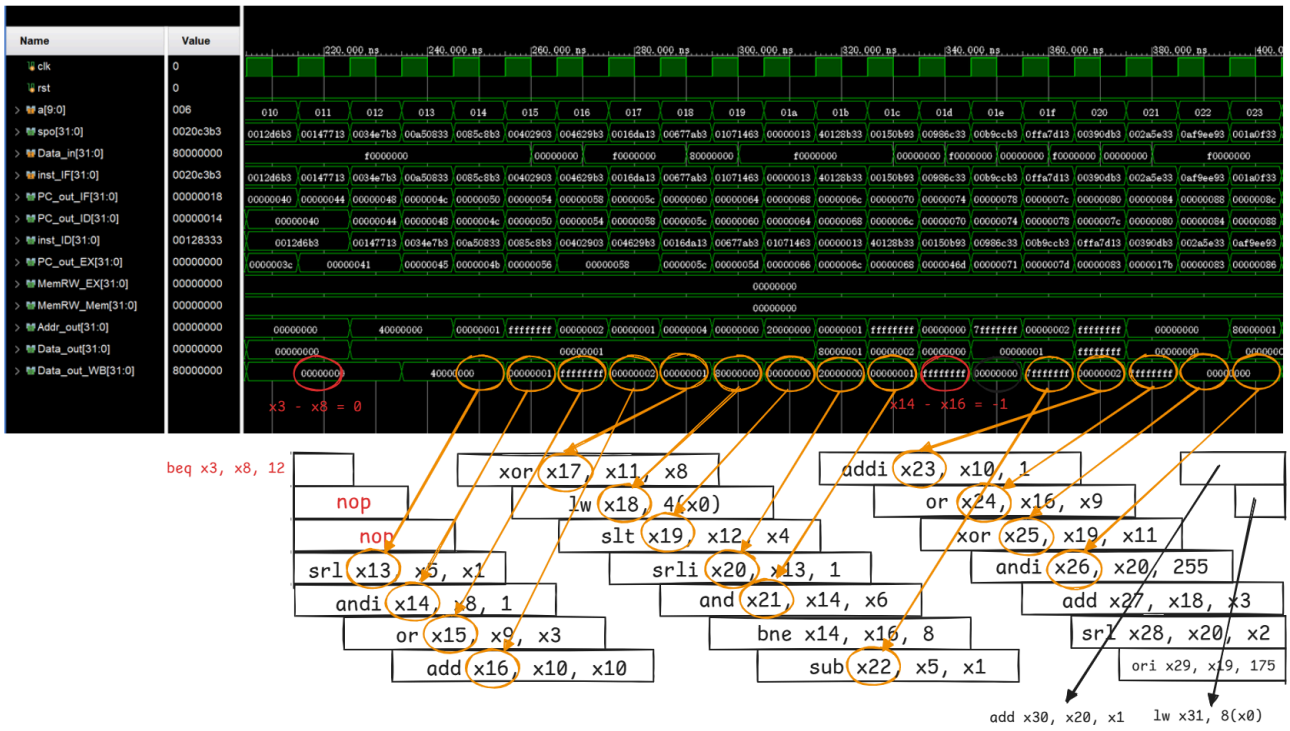
5.1.2 上板验证

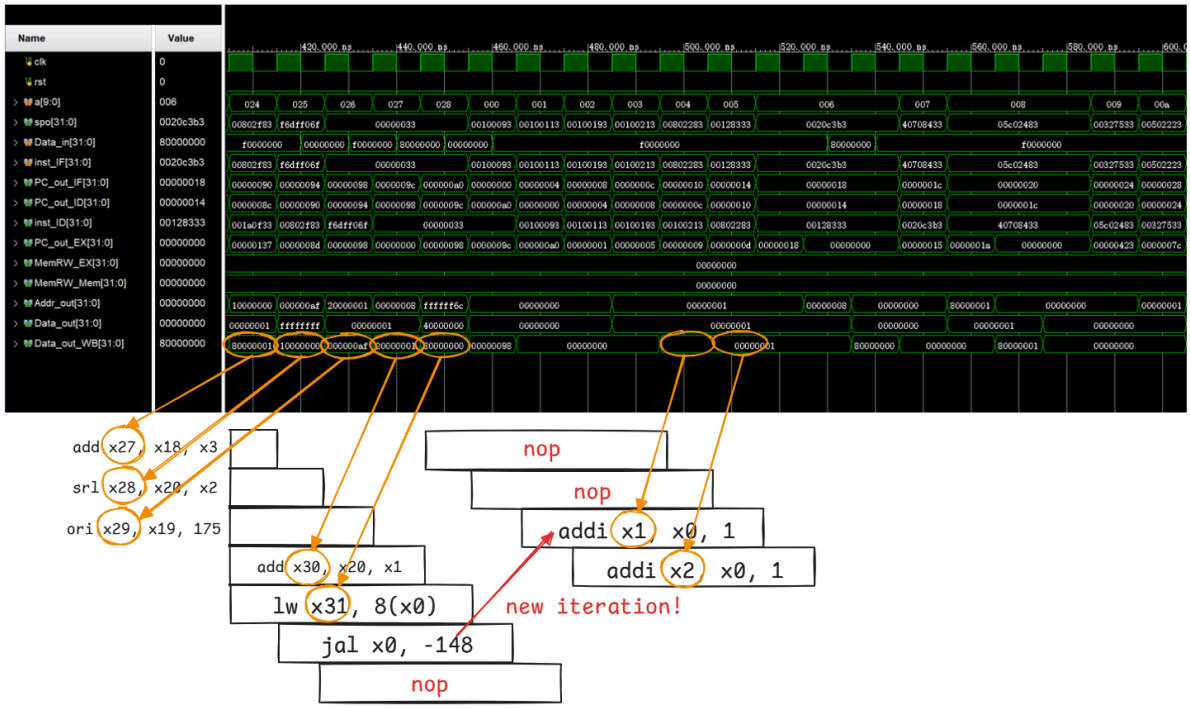
将烧录好的 bit 流文件传入 NEXYS 板上，通过 VGA 显示器观察结果，与波形图的信号进行比对，发现均符合预期，且经助教验收通过。

5.2 带冒险处理的流水线 CPU

5.2.1 仿真结果分析







上图展示了流水线 CPU 在执行 h.coe 文件内的指令时的对应的仿真波形图，同时附上简要的分析（主要观察 `Data_out_WB` 输入端口的结果），发现结果符合预期（即符合 h_mem.pdf 文件所示结果，能够在数据冒险或控制冒险发生时及时停顿）。

5.2.2 上板验证

将烧录好的 bit 流文件传入 NEXYS 板上，通过 VGA 显示器观察结果，与波形图的信号进行比对，发现均符合预期，且经助教验收通过。

六、 实验心得

可能是吸取前面实验的教训，加上对处理器更深一步的了解的缘故吧，个人感觉这次实验的总体难度比上次单周期 CPU 的实验稍微低一些。但这并不意味着这次实验做得相当轻松，中间还是遇到了一些磕磕绊绊，比如观察仿真波形时由于没有理解好各个信号的含义，所以常常会出现刚开始以为 CPU 实现功能有误，结果其实是正确的，只是看错信号了而已的情况，弄得我哭笑不得。

总的来说，这次实验让我对流水线 CPU 的理解更进一步。终于完成最后一个必做试验了，完结撒花 🎉