同济大学
TONGJI UNIVERSITY

计算机系统结构课程实验
总结报告

实验题目：简单的流水线 CPU 设计与性能分析

学号：2252659
姓名：冯甘雨
指导教师：陆有军

日期：2024.10.26

目录

一、实验环境部署与硬件配置说明

- 系统环境：Windows 11
- 开发环境：Vivado 2023.2
- 硬件环境：Xilinix NEXYS A7

二、实验的总体结构

本次实验的包含了对五段静态流水线的实现。该流水线总体结构分为五个阶段，分别是IF（指令获取）、ID（指令译码）、EX（执行）、MEM（访存）、WB（写回）。实现对 8 条 MIPS 指令的五段流水线处理。

1. IF（指令获取）阶段：

- 负责从指令存储器中获取指令。
- 控制逻辑用于管理指令的流入，并将其传递到下一个阶段。

2. ID（指令译码）阶段：

- 负责对获取的指令进行译码，确定指令的类型和操作数。
- 识别操作数的寄存器地址并将其发送到执行阶段。
- 控制逻辑处理分支、跳转等控制指令。

3. EX（执行）阶段：

- 执行ALU（算术逻辑单元）操作，进行算术和逻辑运算。
- 处理分支和跳转条件，计算分支目标地址。

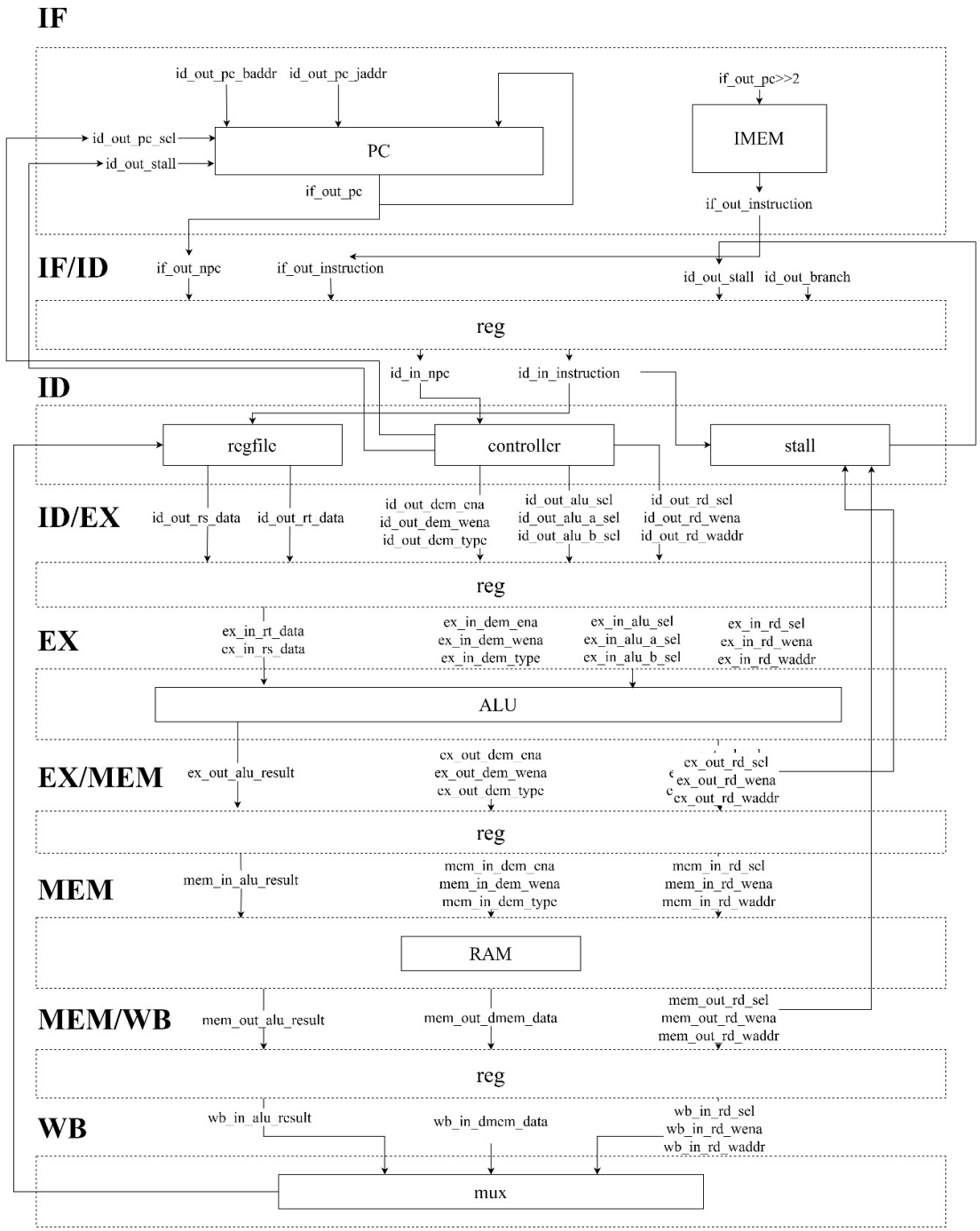
4. MEM（访存）阶段：

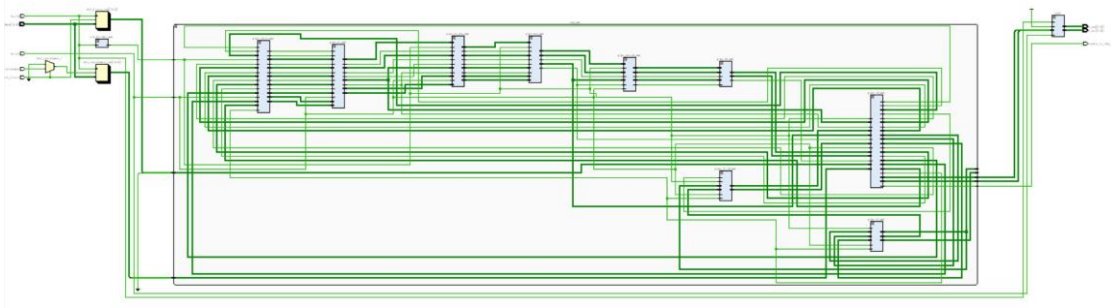
- 处理访存指令，包括加载和存储。
- 访问数据存储器，读取或写入数据。

5. WB（写回）阶段：

- 将计算结果写回寄存器文件，更新寄存器的值。

上述的每个阶段都通过流水线寄存器连接，使得每个阶段的输出可以传递到下一个阶段，形成完整的流水线。这种结构允许在同一时钟周期内处理多个指令的不同阶段，提高了整体性能。通过实现这样的流水线结构，可以更有效地执行 MIPS 指令集中的各种操作。流水线整体架构图（略有省略）如下图所示：





控制信号部分在上学期计算机组成原理课程的单周期 31 条 MIPS 指令集 CPU 基础上略作修改，详见下表：

	IM_R	DM_R	DM_W	ALUC	MUX_PC	MUX_A	MUX_B	MUX_Rd	Reg_W
add	1	0	0	0	0	0	0	0	1
addu	1	0	0	0	0	0	0	0	1
sub	1	0	0	0	0	0	0	0	1
and	1	0	0	0	0	0	0	0	1
or	1	0	0	0	0	0	0	0	1
xor	1	0	0	0	0	0	0	0	1
nor	1	0	0	0	0	0	0	0	1
slt	1	0	0	0	0	0	0	0	1
sltu	1	0	0	0	0	0	0	0	1
sll	1	0	0	0	0	0	0	0	1
srl	1	0	0	0	0	0	0	0	1
sra	1	0	0	0	0	0	0	0	1
sllv	1	0	0	0	0	0	0	0	1
srlv	1	0	0	0	0	0	0	0	1
srav	1	0	0	0	0	0	0	0	1
jr	1	0	0		0				1
addi	1	0	0	0	0	0	0	0	1
addiu	1	0	0	0	0	0	0	0	1
andi	1	0	0	0	0	0	0	0	1
ori	1	0	0	0	0	0	0	0	1
xori	1	0	0	0	0	0	0	0	1
lw	1	0	0	0	0	0	0	0	1
sw	1	0	0	0	0	0	0		1
beq	1	0	0	0	0	0	0		1
bne	1	0	0	0	0	0	0		1
slti	1	0	0	0	0	0	0	0	1
sltiu	1	0	0	0	0	0	0	0	1
lui	1	0	0	0	0	0	0	0	1
j	1	0	0		0				1
jal	1	0	0	0	0	0	0	0	1

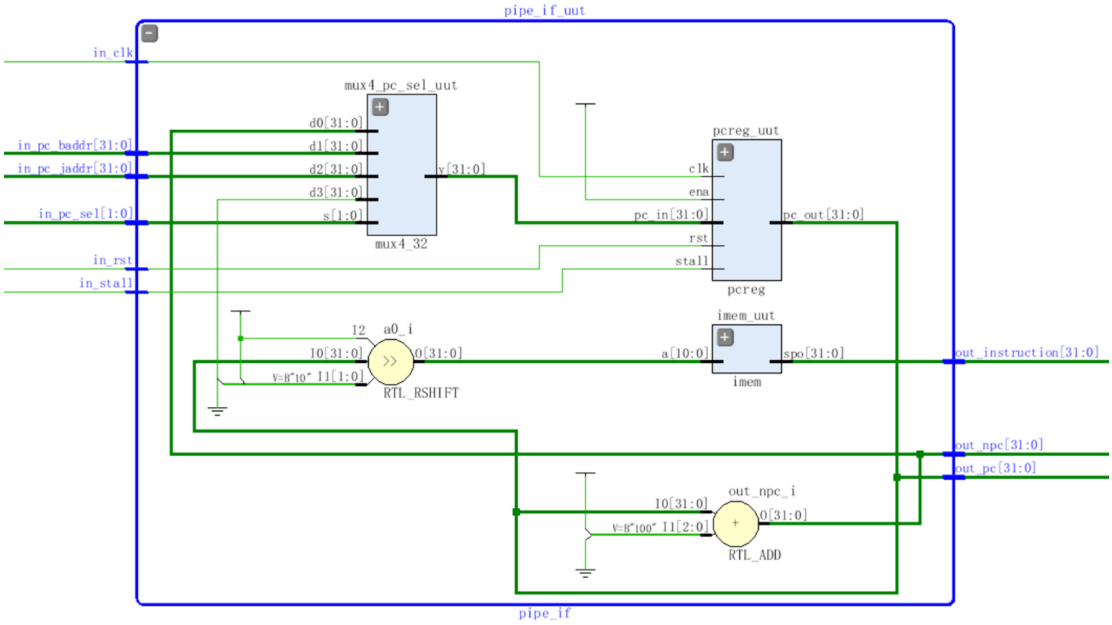
操作码如下（所需 8 位着重标出，包括加载指令（LOAD），存储指令（STORE）,减法运算集（CMP），加法运算集（ADD），跳转指令集（BN，BZ），暂停指令（HALT）与空指令（NOP））

	ENCODEING						Inst.Type
	31…26	25…21	20…16	15…11	10…6	5…0	
LW	100011	sssss	ttttt	iiiii	iiiii	iiiii	M
SW	101011	sssss	ttttt	iiiii	iiiii	iiiii	M
ADD	000000	sssss	ttttt	ddddd	00000	100000	R
ADDU	000000	sssss	ttttt	ddddd	00000	100001	R
SUB	000000	sssss	ttttt	ddddd	00000	100010	R
SUBU	000000	sssss	ttttt	ddddd	00000	100011	R
AND	000000	sssss	ttttt	ddddd	00000	100100	R
OR	000000	sssss	ttttt	ddddd	00000	100101	R

XOR	000000	sssss	ttttt	ddddd	00000	100110	R
NOR	000000	sssss	ttttt	ddddd	00000	100111	R
SLL	000000	sssss	ttttt	ddddd	hhhhh	000000	R
SRL	000000	———	ttttt	ddddd	hhhhh	000010	R
SRA	000000	———	ttttt	ddddd	hhhhh	000011	R
ADDI	001000	sssss	ttttt	iiiii	iiiii	iiiii	I
ADDIU	001001	sssss	ttttt	iiiii	iiiii	iiiii	I
ANDI	001100	sssss	ttttt	iiiii	iiiii	iiiii	I
ORI	001101	sssss	ttttt	iiiii	iiiii	iiiii	I
XOUI	001110	sssss	ttttt	iiiii	iiiii	iiiii	I
BEQ	000100	sssss	ttttt	iiiii	iiiii	iiiii	B
BGEZ	000001	sssss	00001	iiiii	iiiii	iiiii	B
BGTZ	000111	sssss	00000	iiiii	iiiii	iiiii	B
BLEZ	000110	sssss	00000	iiiii	iiiii	iiiii	B
BLTZ	000001	sssss	00000	iiiii	iiiii	iiiii	B
BNE	000101	sssss	ttttt	iiiii	iiiii	iiiii	B
J	000010	iiiii	iiiii	iiiii	iiiii	iiiii	J
NOP	000000	00000	00000	00000	00000	00000	M
HALT	111111	———	———	———	———	———	M

三、总体架构部件的解释说明

1. IF（指令获取）阶段




```
// 指令内存访问模块
module pipe_if(
input in_clk, // 时钟输入
input in_rst, // 复位输入
input in_stall, // 流水线暂停信号
input [31:0] in_pc_jaddr, // 跳转地址输入
input [31:0] in_pc_baddr, // 分支地址输入
input [1:0] in_pc_sel, // PC选择信号，用于判断跳转或分支
output [31:0] out_pc, // 输出PC
output [31:0] out_npc, // 输出下一个PC
output [31:0] out_instruction //
输出指令
);
```

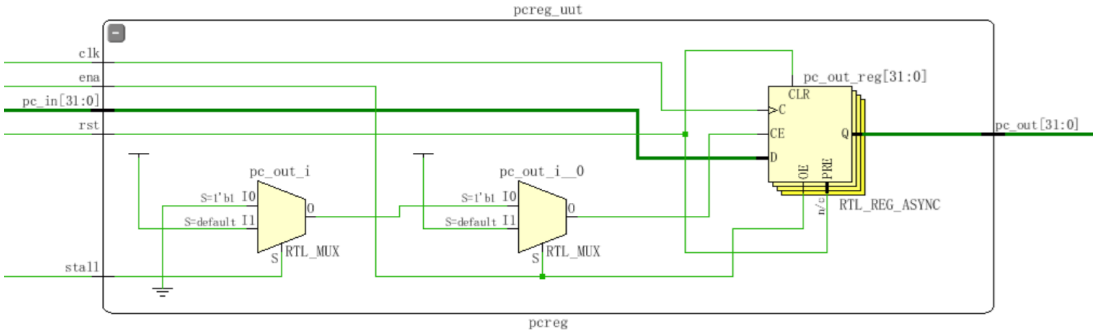
pipe_if 模块是流水线中的指令内存阶段的实现。该模块负责处理当前指令的地址、下一条指令的地址以及当前指令的内容。整体结构分为三个主要部分：

首先，实例化 pcreg 模块，该模块负责维护和更新当前指令的地址。其输入包括时钟信号 in_clk、复位信号 in_rst、暂停信号 in_stall、下一条指令的地址 next_pc，以及输出当前指令的地址 out_pc。

其次，使用了 mux4_32 模块，该模块是一个4选1的32位多路选择器，根据 in_pc_sel 选择不同的地址输入，输出到 next_pc 中，作为下一条指令的地址。其中，out_npc 通过简单的加法操作得到，表示当前指令的地址加上 4。

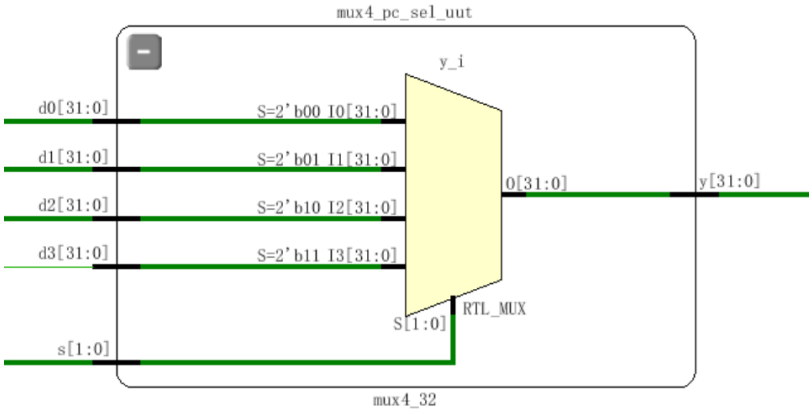
最后，通过 imem 模块读取指令内存，根据当前指令的地址 out_pc 获取相应的指令内容，将其输出为 out_instruction。

① **pc**寄存器模块



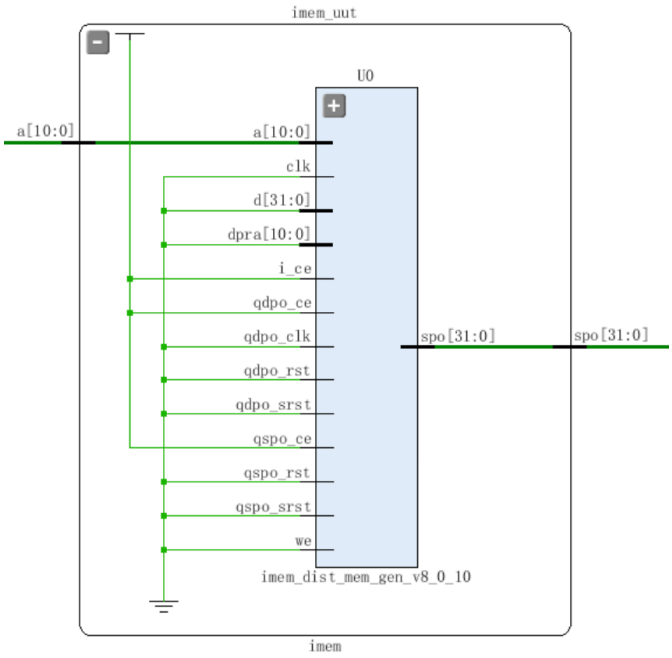

```
// 指令内存流水线寄存器模块
module pcreg(
input clk, // 时钟信号
input ena, // 使能信号
input rst, // 复位信号
input stall, // 暂停信号
input [31:0] pc_in, // 输入地址
output reg [31:0] pc_out // 输出地址
);
```

② **mux**多路选择器模块

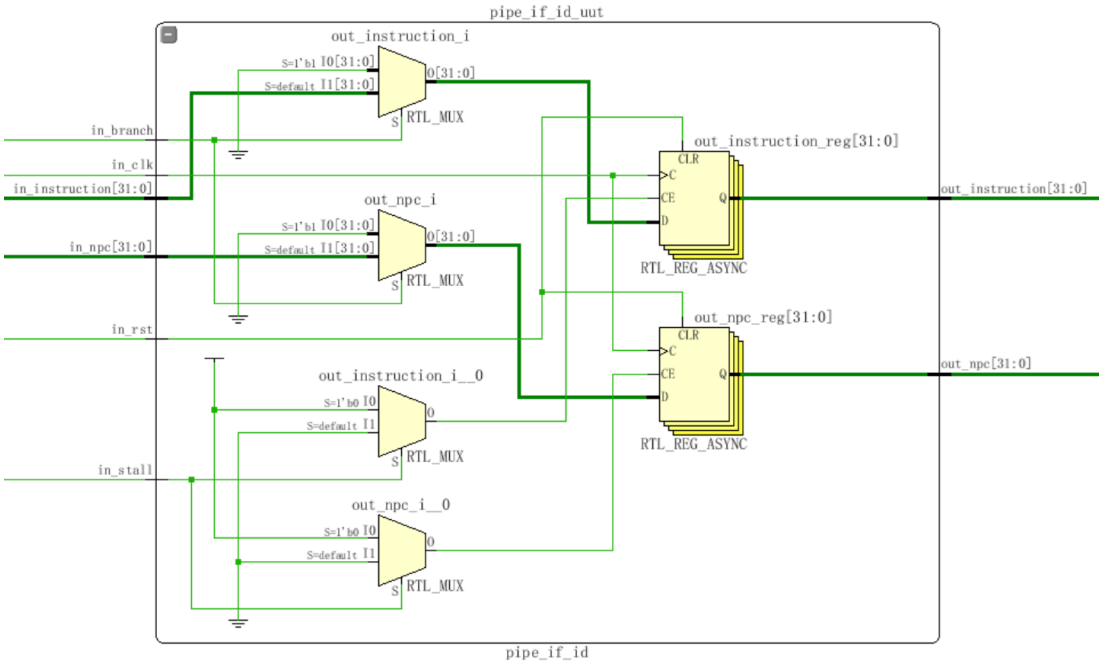



```
// 32位四选一多路器模块
module mux4_32(
input [31:0] d0, // 输入数据0
input [31:0] d1, // 输入数据1
input [31:0] d2, // 输入数据2
input [31:0] d3, // 输入数据3
input [1:0] s, // 选择信号
output reg [31:0] y // 输出结果
);
```

③ **imem**指令存储器模块（调用**IP**核实现**ROM**）



2. IF、ID 段间寄存段




```
// 指令译码流水线寄存器模块
module pipe_if_id(
input in_clk, // 时钟信号
input in_rst, // 复位信号
input in_stall, // 暂停信号
input in_branch, // 分支信号
input [31:0] in_npc, // 下一程序计数器地址
input [31:0] in_instruction, // 输入指令
```

```
output reg [31:0] out_npc, // 输出下一程序计数器地址
output reg [31:0] out_instruction // 输出指令
);
```

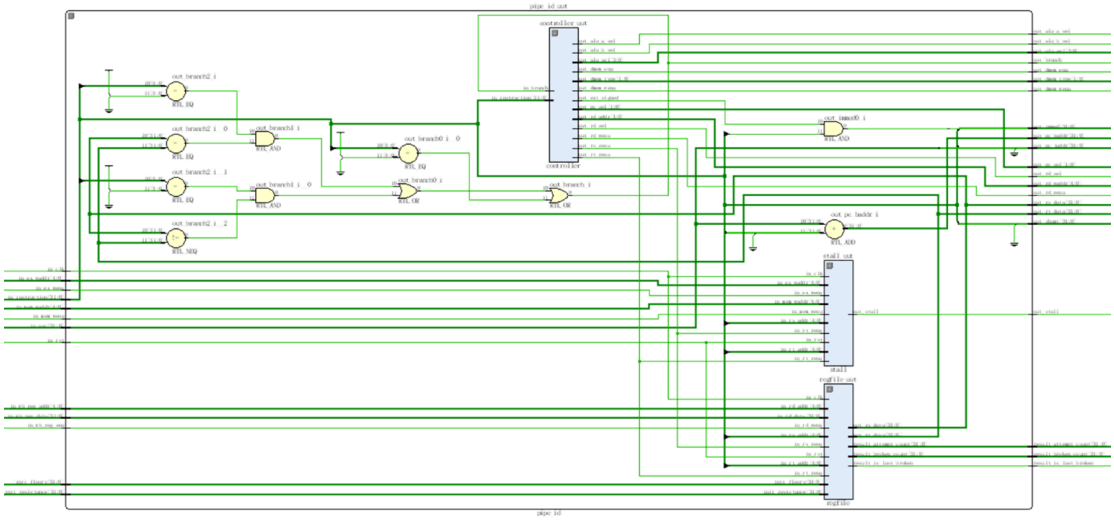
该模块是一个流水线寄存器模块，其主要作用是在流水线中存储并传递指令取指阶段的结果。在时钟信号上升沿或者复位信号上升沿触发时，模块进行操作。如果复位信号（in_rst）处于高电平状态，表示系统正在进行复位，那么输出寄存器（out_npc 和 out_instruction）将被置零。

在非暂停状态（~in_stall）下，根据分支信号（in_branch）的状态来决定是否清零输出寄存器。若分支状态为真，表示处于分支情况，输出寄存器将被清零。否则，将输入的下一程序计数器地址（in_npc）和指令（in_instruction）分别赋给

输出寄存器（

out_npc 和 out_instruction）。在暂停状态下，输出寄存器的值将保持不变。这个模块的设计是为了确保在每个时钟周期内，正确地将译码阶段的输出传递给下一个流水线阶段，以保持流水线的正常运行。

3. IF（指令译码）阶段

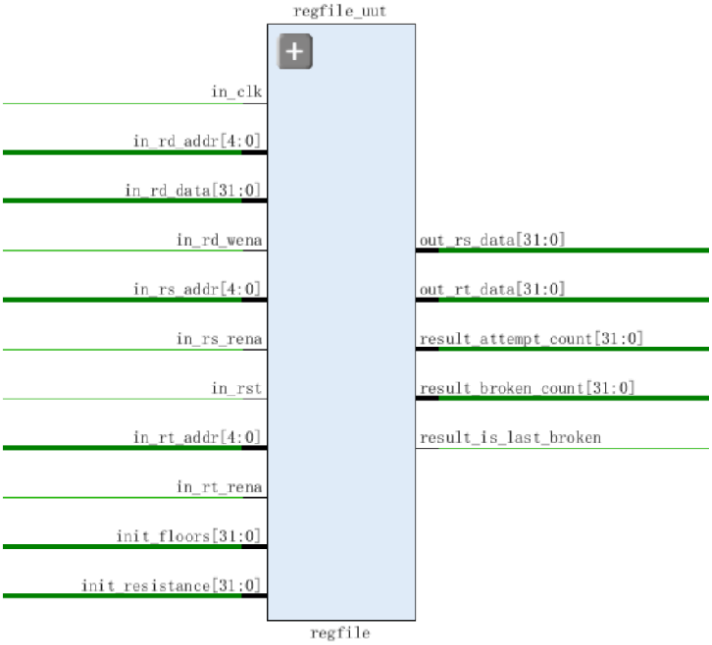


```
// 流水线译码模块，负责解析指令和产生控制信号
module pipe_id(
input in_clk, // 时钟信号
input in_rst, // 复位信号
input [31:0] in_npc, // 当前程序计数器地址
input [31:0] in_instruction, // 输入指令
input [4:0] in_ex_waddr, // 执行阶段写入地址
input [4:0] in_mem_waddr, // 存储阶段写入地址
input in_ex_wena, // 执行阶段写使能
input in_mem_wena, // 存储阶段写使能
input [4:0] in_wb_reg_addr, // 写回阶段写入寄存器地址
input in_wb_reg_ena, // 写回阶段写入寄存器使能
input [31:0] in_wb_reg_data, // 写回阶段写入寄存器数据
input [31:0] init_floors, // 初始楼层数
input [31:0] init_resistance, // 初始耐摔值
output [31:0] out_rs_data, // 读取寄存器 rs 数据
```

```
output [31:0] out_rt_data, //
读取寄存器 rt 数据
output [4:0] out_rd_waddr, // 写回阶段写入地址
output out_rd_sel, // 写回阶段写入选择信号
output out_rd_wena, // 写回阶段写入寄存器使能
output [31:0] out_immed, // 立即数
output [31:0] out_shamt, // 移位位数
output out_dmem_ena, // 数据存储器读使能
output out_dmem_wena, // 数据存储器写使能
output [1:0] out_dmem_type, // 数据存储器读写类型
output [31:0] out_pc_baddr, // 分支时的程序计数器
output [31:0] out_pc_jaddr, // 跳转时的程序计数器
output [1:0] out_pc_sel, // 程序计数器选择信号
output out_alu_a_sel, // ALU A 选择信号
output out_alu_b_sel, // ALU B 选择信号
output [3:0] out_alu_sel, // ALU 操作选择信号
output out_stall, // 暂停信号
output out_branch, // 分支信号
output [31:0] result_attempt_count, // 摔鸡蛋尝试次数
output [31:0] result_broken_count, // 摔鸡蛋摔破次数
output result_is_last_broken // 最后一次是否摔破
);
```

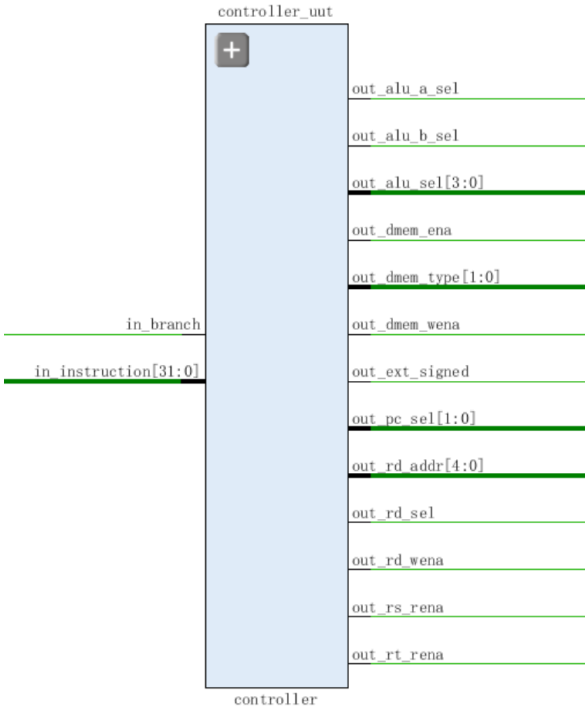
该模块是流水线中的ID（Instruction Decode）译码模块，主要负责对输入的指令进行解析和控制信号的生成。首先，通过解析指令，提取出关键信息如操作码、功能码以及源、目标寄存器的地址。随后，通过调用寄存器堆模块（regfile）实现对寄存器的读取和写入，获取源寄存器和目标寄存器的数据。通过调用控制器模块（controller），解析当前指令并生成相应的控制信号，如读写使能信号、写入寄存器选择信号等。此外，该模块还生成数据存储器相关的控制信号，包括读写使能信号和读写类型信号。针对ALU（算术逻辑单元）的操作，通过控制器生成ALU的输入选择信号和操作选择信号。为了支持分支和跳转，计算并生成程序计数器的基地址和选择信号。最后，通过调用数据冲突判断模块（stall），判断是否存在数据冲突，从而产生相应的暂停信号。整体而言，该模块在流水线中起到了解析指令、生成控制信号以及处理数据冲突的关键作用，为流水线的下一阶
段提供了必要的信息和控制。

① **regfile**寄存器堆




```
// 寄存器堆模块
module regfile(
input in_clk, // 时钟信号
input in_rst, // 复位信号
input in_rs_rena, // 源寄存器读使能
input in_rt_rena, // 目标寄存器读使能
input in_rd_wena, // 目标寄存器写使能
input [4:0] in_rd_addr, // 目标寄存器写地址
input [4:0] in_rs_addr, // 源寄存器地址
input [4:0] in_rt_addr, // 目标寄存器地址
input [31:0] in_rd_data, // 目标寄存器写数据
input [31:0] init_floors, // 初始楼层数
input [31:0] init_resistance, // 初始耐摔值
output reg [31:0] out_rs_data, // 源寄存器数据输出
output reg [31:0] out_rt_data, // 目标寄存器数据输出
output [31:0] result_attempt_count, // 摔鸡蛋总次数
output [31:0] result_broken_count, // 摔碎的鸡蛋总数
output result_is_last_broken // 最后是否摔碎
);
```

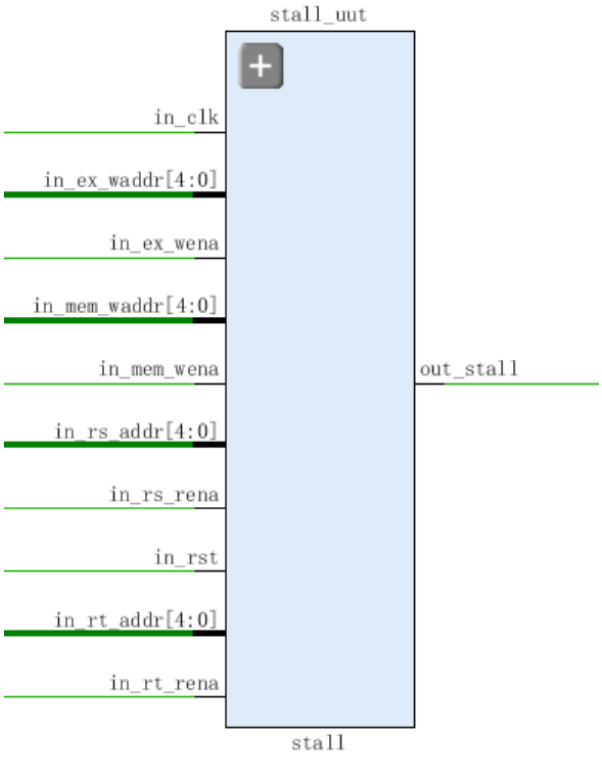
② **controller**控制器




```
// 控制器模块
module controller(
input in_branch, // 分支信号
input [31:0] in_instruction, // 输入指令
output out_rs_rena, // 源寄存器读使能
output out_rt_rena, // 目标寄存器读使能
output out_rd_wena, // 目标寄存器写使能
output [4:0] out_rd_addr, // 目标寄存器写地址
output out_rd_sel, // 目标寄存器写使能
output out_dmem_ena, // 数据存储器读使能
output out_dmem_wena, // 数据存储器写使能
output [1:0] out_dmem_type, // 数据存储器写类型
output out_ext_signed, // 符号扩展信号
output out_alu_a_sel, // ALU A 选择信号
output out_alu_b_sel, // ALU B 选择信号
output [3:0] out_alu_sel, // ALU 控制信号
output [1:0] out_pc_sel // PC 选择信号
);
```

该模块是流水线中的控制器模块，主要功能是解析输入的指令并产生对应的控制信号，以驱动流水线的各个阶段的操作。通过检测输入指令的操作码和功能码，该模块确定指令的类型并生成相应的控制信号。控制信号包括源寄存器读使能、目标寄存器读使能、目标寄存器写使能、目标寄存器写地址、目标寄存器写使能、数据存储器读使能、数据存储器写使能、数据存储器写类型、符号扩展信号、ALU A/B 选择信号、ALU 控制信号、PC 选择信号等。该模块通过逻辑比较和组合逻辑的方式判断当前指令的类型，并生成相应的控制信号，这些信号将用于控制其他模块的行为，协调流水线的运行。

③ **stall**暂停信号模块




```
// 流水线中的暂停模块，用于检测并处理数据冲突，实现对应的暂停机制。
module stall(
input in_clk, // 时钟信号
input in_rst, // 复位信号
input [4:0] in_rs_addr, // 源寄存器地址
input [4:0] in_rt_addr, // 目标寄存器地址
input in_rs_rena, // 源寄存器读使能
input in_rt_rena, // 目标寄存器读使能
input in_ex_wena, // EX阶段写使能
input in_mem_wena, // MEM阶段写使能
input [4:0] in_ex_waddr, // EX阶段写地址
input [4:0] in_mem_waddr, // MEM阶段写地址
output reg out_stall // 输出的暂停信号
);
```

stall 模块的机制是为了解决流水线中可能发生的数据冲突导致的暂停情况。在每个时钟周期的下降沿，模块会检查当前是否需要暂停流水线。

在正常运行状态下，模块会检查数据冲突情况。如果存在 ex 阶段与 id 阶段之间的读写冲突，即 ex 阶段需要写入的目标寄存器与 id 阶段的源寄存器之一相同，模块会将 stall_ltime 设置为 1，表示暂停流水线两个周期。另一方面，如果存在 ex 阶段与 mem 阶段之间的读写冲突，即 ex 阶段需要写入的目标寄存器与 mem 阶段的目标寄存器相同，模块会将 stall_ltime 设置为 0（配合out_stall=1），表示暂停流水线一个周期。在暂停期间，计数器 stall_ltime 会递减，直至减至零。此时，流水线可以继续正常运行。整个机制通过控制 out_stall 信号，实现了在发生数据冲突时暂停流水线，确保数据的正确传递和处理。

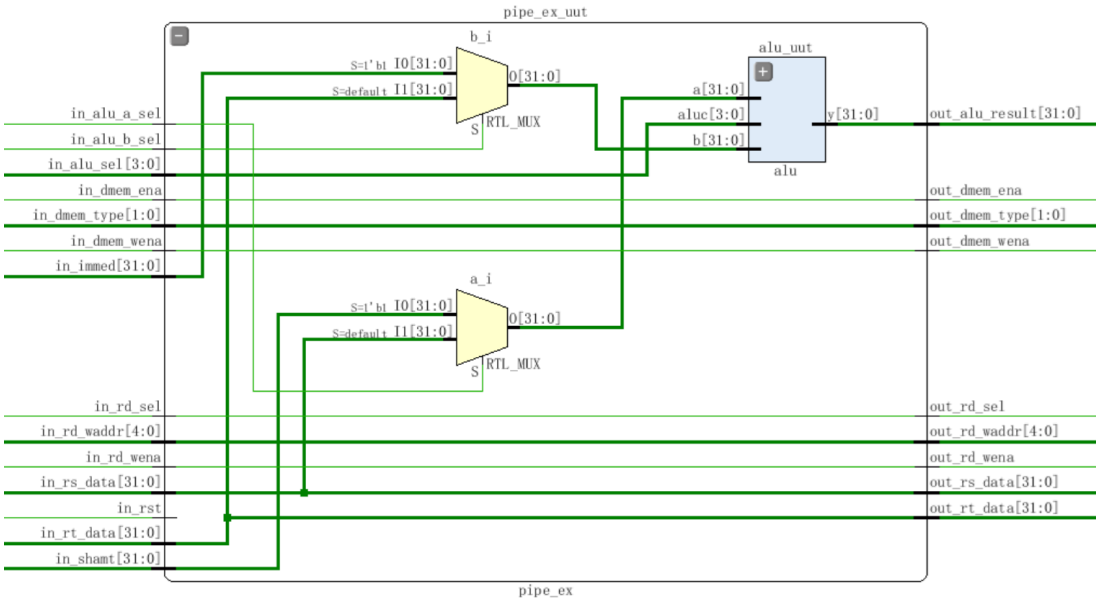
4. ID、EX 段间寄存段


```
// 指令译码到执行流水线寄存器模块
module pipe_id_ex(
input in_clk, // 时钟信号
input in_rst, // 复位信号
input in_dmem_ena, // 数据存储器使能信号
input in_dmem_wena, // 数据存储器写使能信号
input [1:0] in_dmem_type, // 数据存储器访问类型
input [31:0] in_rs_data, // 源寄存器数据
input [31:0] in_rt_data, // 目标寄存器数据
input [4:0] in_rd_waddr, // 写目标寄存器地址
input in_rd_sel, // 写目标寄存器选择信号
input in_rd_wena, // 写目标寄存器使能信号
input [31:0] in_immed, // 立即数
input [31:0] in_shamt, // 移位操作位数
input in_alu_a_sel, // ALU 操作数 A 选择信号
input in_alu_b_sel, // ALU 操作数 B 选择信号
input [3:0] in_alu_sel, // ALU 操作选择信号
input in_stall, // 流水线是否暂停信号
output reg out_dmem_ena, // 输出数据存储器使能信号
output reg out_dmem_wena, // 输出数据存储器写使能信号
output reg [1:0] out_dmem_type, // 输出数据存储器访问类型
output reg [31:0] out_rs_data, // 输出源寄存器数据
output reg [31:0] out_rt_data, // 输出目标寄存器数据
output reg [4:0] out_rd_waddr, // 输出写目标寄存器地址
output reg out_rd_sel, // 输出写目标寄存器选择信号
```

```
output reg out_rd_wena, // 输出写目标寄存器使能信号
output reg [31:0] out_immed, // 输出立即数
output reg [31:0] out_shamt, // 输出移位操作位数
output reg out_alu_a_sel, // 输出 ALU 操作数 A 选择信号
output reg out_alu_b_sel, // 输出 ALU 操作数 B 选择信号
output reg [3:0] out_alu_sel // 输出 ALU 操作选择信号
);
```

该模块是流水线的ID（译码）到EX（执行）阶段的寄存器模块。在时钟上升沿或复位信号上升沿时，根据输入的控制信号和数据，更新流水线的输出信号，为下一个阶段的执行提供必要的输入。模块中包含了对输入信号的判断和处理逻辑，以及根据输入更新输出的逻辑。具体功能包括控制数据存储器的使能信号、写使能信号和访问类型的输出；输出源寄存器数据、目标寄存器数据、写目标寄存器地址、写目标寄存器选择信号和写目标寄存器使能信号；输出立即数和移位操作位数；以及控制 ALU 操作数 A 和 B 选择的信号，以及 ALU 操作选择的信号。此外，模块还考虑了流水线是否处于暂停状态，如果是，则清空输出信号，保持流水线的同步和正确性。该模块能够将控制信号和数据在流水线中传递，为指令的执行提供必要的数据和控制。

5. EX（执行）阶段



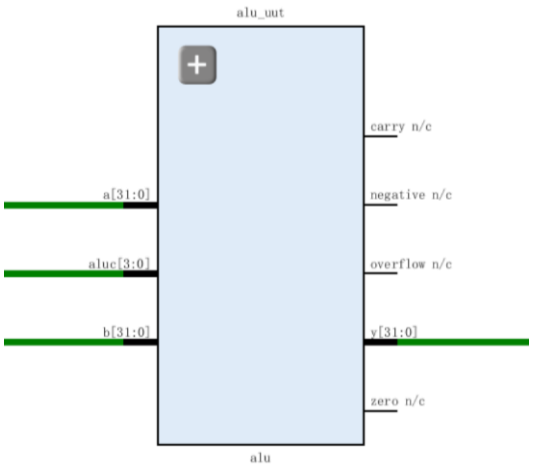

```
// 流水线执行阶段模块
module pipe_ex(
input in_rst,
input in_dmem_ena, // 数据存储器使能信号
input in_dmem_wena, // 数据存储器写使能信号
input [1:0] in_dmem_type, // 数据存储器访问类型
input [31:0] in_rs_data, // 输入源寄存器数据
input [31:0] in_rt_data, // 输入目标寄存器数据
input [4:0] in_rd_waddr, //
输入写目标寄存器地址
input in_rd_sel, // 输入写目标寄存器选择信号
```

```
input in_rd_wena, // 输入写目标寄存器使能信号
input [31:0] in_immed, // 输入立即数
input [31:0] in_shamt, // 输入移位操作位数
input in_alu_a_sel, // 输入 ALU 操作数 A 选择信号
input in_alu_b_sel, // 输入 ALU 操作数 B 选择信号
input [3:0] in_alu_sel, // 输入 ALU 操作选择信号
output out_dmem_ena, // 输出数据存储器使能信号
output out_dmem_wena, // 输出数据存储器写使能信号
output [1:0] out_dmem_type, // 输出数据存储器访问类型
output [31:0] out_rs_data, // 输出源寄存器数据
output [31:0] out_rt_data, // 输出目标寄存器数据
output [4:0] out_rd_waddr, // 输出写目标寄存器地址
output out_rd_sel, // 输出写目标寄存器选择信号
output out_rd_wena, // 输出写目标寄存器使能信号
output [31:0] out_alu_result // 输出 ALU 运算结果
);
```

该模块是流水线执行阶段（pipe_ex）的实现，主要负责控制ALU（算术逻辑单元）的运算。在ALU运算控制方面，模块接收ALU的两个操作数（in_alu_a_sel、in_alu_b_sel、in_alu_sel）进行运算，并输出结果（out_alu_result）。ALU运算数选择方面，输入的ALU运算数根据选择信号（in_alu_a_sel、in_alu_b_sel）进行选择，可以是寄存器数据或者立即数（immediate或shamt）。综上所述，模块实现了流水线执行阶段对指令的计算和对数据存储器的操作，为流水线的正常运行提供了必要的数据和控制信号。

① **ALU**模块

```
// 算术逻辑单元模块
module alu(
input [31:0] a, // 输入操作数a
input [31:0] b, // 输入操作数b
output [31:0] y, // 输出结果
input [3:0] aluc, // ALU
控制信号
output zero, // 零标志位
output carry, // 进位标志位
output negative, // 负数标志位
output overflow // 溢出标志位
);
```



ALU对应的操作信号如下所示：

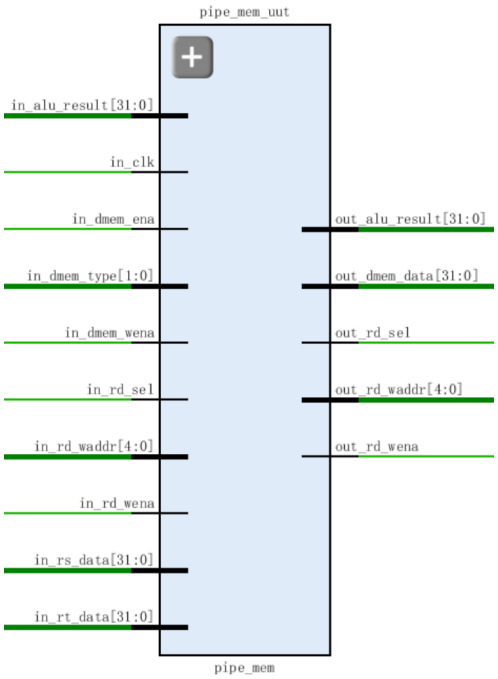
	ALU_SEL[3]	ALU_SEL[2]	ALU_SEL[1]	ALU_SEL[0]
ADD	0	0	0	0
ADDU	0	0	0	1
SUB	0	0	1	0
SUBU	0	0	1	1
AND	0	1	0	0
OR	0	1	0	1
XOR	0	1	1	0
NOR	0	1	1	1
SLT	1	0	0	0
SLTU	1	0	0	1
SLL	1	0	1	0
SRL	1	0	1	1

6. EX、MEM 段间寄存段


```
// 执行、访存模块间寄存器模块
module pipe_ex_mem(
input in_clk,
input in_rst,
input in_dmem_ena, // 数据存储器使能信号
input in_dmem_wena, // 数据存储器写使能信号
input [1:0] in_dmem_type, // 数据存储器类型
input [31:0] in_rs_data, // 寄存器数据
input [31:0] in_rt_data, // 寄存器数据
input [4:0] in_rd_waddr, // 目标寄存器地址
input in_rd_sel, // 目标寄存器选择信号
input in_rd_wena, // 目标寄存器写使能信号
input [31:0] in_alu_result, // ALU运算结果
output reg out_dmem_ena, // 输出数据存储器使能信号
output reg out_dmem_wena, //
输出数据存储器写使能信号
output reg [1:0] out_dmem_type, // 输出数据存储器类型
output reg [31:0] out_rs_data, // 输出寄存器数据
output reg [31:0] out_rt_data, // 输出寄存器数据
output reg [4:0] out_rd_waddr, // 输出目标寄存器地址
output reg out_rd_sel, // 输出目标寄存器选择信号
output reg out_rd_wena, // 输出目标寄存器写使能信号
output reg [31:0] out_alu_result // 输出 ALU 运算结果
);
```

上述模块充当了流水线中执行阶段（EX）和访存阶段（MEM）之间的寄存器，负责传递控制信号和计算结果。其主要功能是在时钟上升沿或复位信号触发时，根据输入信号的状态，将数据存储器（Data Memory）的使能信号、写使能信号和类型信号传递到下一阶段，同时传递寄存器相关的数据、写地址、写选择信号和写使能信号，以及ALU运算的结果。在复位状态下，所有输出信号被置零；在正常运行状态下，输入信号直接传递给对应的输出信号，即完成寄存一个周期的功能。这有助于确保流水线的正确运行，保持数据的传递一致性，同时提供对数据存储器和寄存器的控制和写入。

7. MEM（访存）阶段

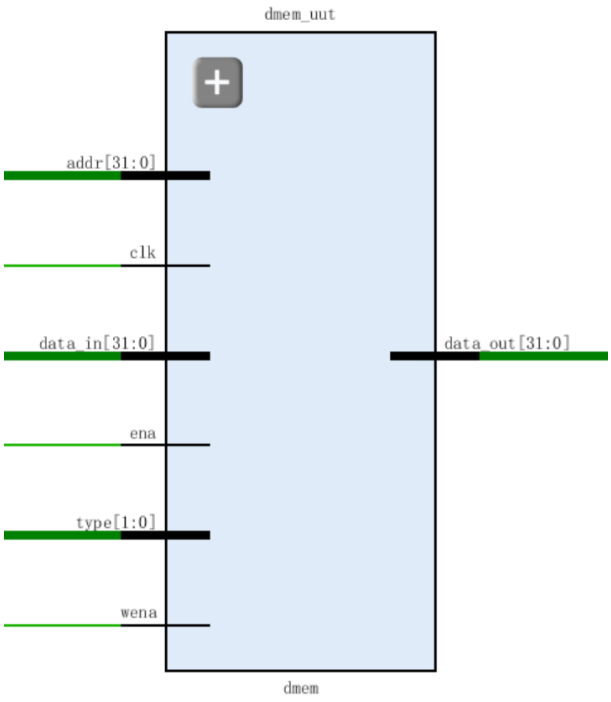



```
// 访存阶段模块
module pipe_mem(
input in_clk, // 时钟信号
input in_dmem_ena, // 数据存储器使能信号
input in_dmem_wena, // 数据存储器写使能信号
input [1:0] in_dmem_type, // 数据存储器类型信号
input [31:0] in_rs_data, // 执行阶段传递的寄存器数据
input [31:0] in_rt_data, // 执行阶段传递的寄存器数据
input [4:0] in_rd_waddr, // 执行阶段传递的写地址
input in_rd_sel, // 执行阶段传递的写选择信号
input in_rd_wena, // 执行阶段传递的写使能信号
input [31:0] in_alu_result, // 执行阶段传递的ALU计算结果
output [4:0] out_rd_waddr, // 输出到写回阶段的写地址
output out_rd_sel, // 输出到写回阶段的写选择信号
output out_rd_wena, // 输出到写回阶段的写使能信号
output [31:0] out_alu_result, // 输出到写回阶段的ALU计算结果
output [31:0] out_dmem_data // 输出到数据存储器的数据
);
```

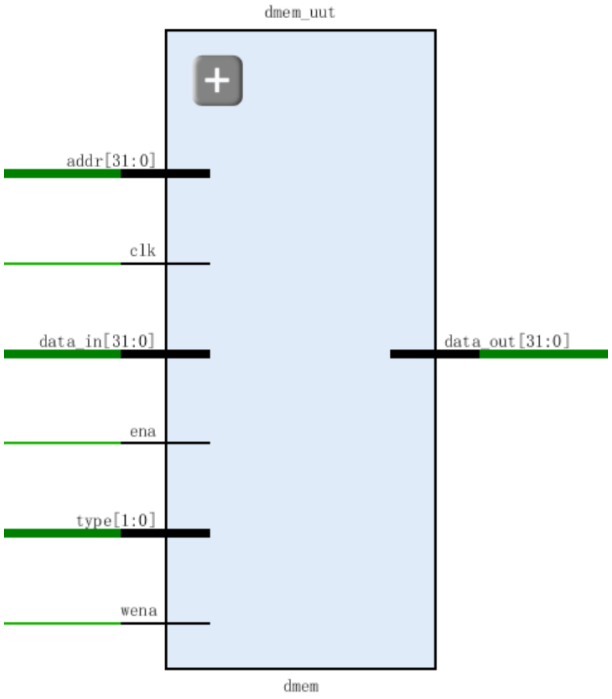
该模块是流水线中的访存阶段（MEM）模块。其功能主要包括：接收来自执行阶段的输入信号，包括寄存器数据、ALU 计算结果等；控制数据存储器的读写操作，通过传递给数据存储器的输入信号实现；将访存阶段计算的结果传递给写回阶段，包括ALU 计算结果和数据存储器的读取数据；将写回阶段的写使能信号、写选择信号和写地址传递给下一阶段。总体而言，该模块负责执行访存阶段的操作，包括数据存储器的读写和与写回阶段的数据传递。

① **dmem**模块

```
// 数据存储器模块
module dmem(
input clk, // 时钟输入
input ena, // 使能信号
input wena, // 写使能信号
input [31:0] addr, // 地址信号
input [1:0] dmem_type, //
数据存储类型
input [31:0] data_in, // 写入数据
output [31:0] data_out // 读取数据输出
);
```



8. MEM、WB 段间寄存段

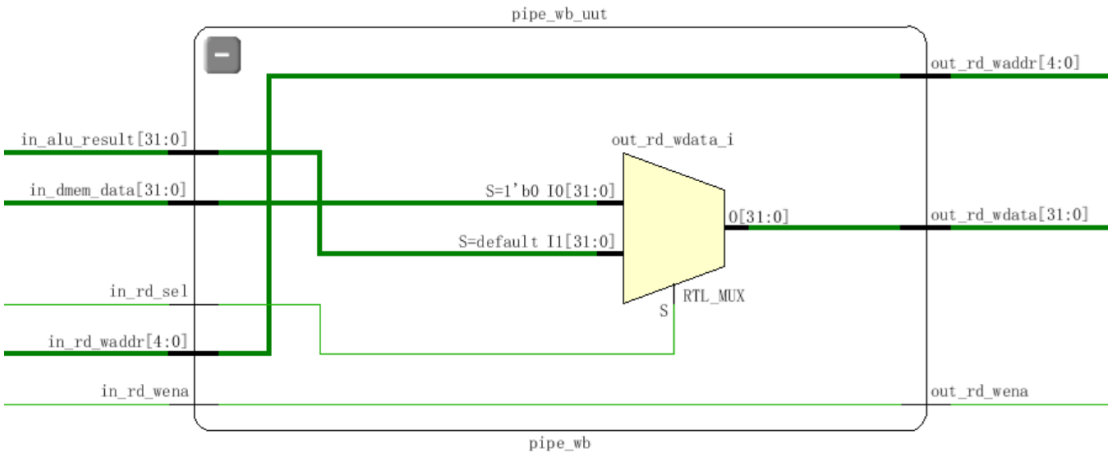



```
// 内存访问和写回间的寄存模块
module pipe_mem_wb(
input in_clk, // 时钟信号
input in_rst, // 复位信号
input [4:0] in_rd_waddr, // 读取寄存器地址
input in_rd_sel, // 读取寄存器选择信号
input in_rd_wena, // 读取寄存器写使能信号
input [31:0] in_alu_result, // ALU 计算结果
input [31:0] in_dmem_data, // 数据存储器读取数据
output reg [4:0] out_rd_waddr, // 读取寄存器地址输出
output reg out_rd_wena, // 读取寄存器写使能输出
output reg out_rd_sel, // 读取寄存器选择输出
output reg [31:0] out_alu_result, // ALU 计算结果输出
output reg [31:0] out_dmem_data // 数据存储器读取数据输出
);
```

该模块是内存访问（MEM）和写回（WB）两个阶段之间的寄存器模块。其主要功能是在时钟上升沿或复位信号触发时，根据输入信号的状态更新输出信号。模块内包含了读取寄存器地址、读取寄存器选择、读取寄存器写使能、ALU 计算结果和数据存储器读取数据等输入信号。在每个时钟周期内，根据复位信号的状态，模块将输入信号的值传递给相应的输出信号，以保持流水线的数据传递。这有助于协调 MEM 和 WB 两个阶段的数据传输，确保数据正确流经流水线的各个阶段，完成指令的执行。

9. WB（写回）阶段

```
// 写回阶段模块
module pipe_wb(
input [4:0] in_rd_waddr, // 写回寄存器地址
input in_rd_wena, // 写回寄存器写使能
input in_rd_sel, // 写回寄存器选择
input [31:0] in_alu_result, // ALU 计算结果
input [31:0] in_dmem_data, // 数据存储器读取数据
output [4:0] out_rd_waddr, //
写回寄存器地址
output out_rd_wena, // 写回寄存器写使能
output [31:0] out_rd_wdata // 写回寄存器写入数据
);
```



该模块是数据通路中Write Back（WB）阶段的寄存器模块。该模块接收来自MEM阶段传递来的ALU计算结果或内存数据，根据控制信号选择将哪一个数据写入寄存器文件。具体而言，当写回使能信号（in_rd_wena）为高时，根据写回选择信号（in_rd_sel）的不同，选择写入ALU计算结果或数据存储器读取的数据。同时，将写入的寄存器地址（in_rd_waddr）输出到regfile写入。

四、实验仿真过程

1. 8 条指令流水线的仿真过程

各条指令的仿真测试随CPU模块的搭建共同进行，较为繁琐，因此此处不进行赘述。最终进行仿真时，选用了第六部分中所阐述的比萨塔摔鸡蛋程序，转为coe文件后初始化imem的IP核。

2.撰写 testbench 程序

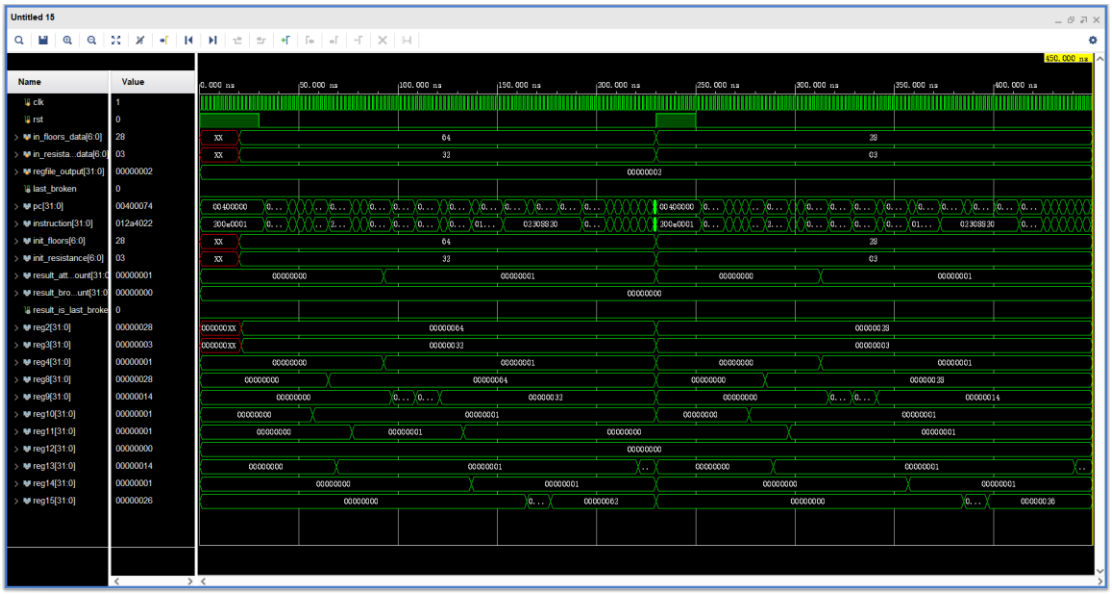
为了在一次下板过程中进行多组数据的调试，初始化的层高与耐摔值从外层模块传入。用于 reset 时直接初始化对应的 regfile 寄存器。testbench 程序如下，可以将每个 clk 时的 cpu 值输出到 log 文件中。

```
module testbench(
);
reg clk;
reg rst;
reg [15:0] init_data;
reg is_init_floors;
reg is_init_resistance;
wire last_broken;
integer regfile_output;
initial
begin
regfile_output = $fopen("regfile_output.txt");
clk = 0;
rst = 1;
#20 init_data = 16'd0030; //  初始化楼高
#20 is_init_floors = 1'b1;
#20 is_init_floors = 1'b0;
#20 init_data = 16'd0019; //  初始化耐摔值
#20 is_init_resistance = 1'b1;
#20 is_init_resistance = 1'b0;
#20 rst = 0;
end
always
begin
#5 clk = ~clk;
end
// ……省略log信息的输出……
board_top board_top_uut(
.in_clk(clk),
.in_rst(rst),
.in_data(init_data),
.is_init_floors(is_init_floors),
```

```
.is_init_resistance(is_init_resistance),
.result_is_last_broken(last_broken)
);
endmodule
```

3. 运行仿真

如上述 testbench 文件所述，测试初始化楼层高度为 30，鸡蛋耐摔值为 19 时的情况。在 Vivado 中进行仿真，可以观察到寄存器当中的值与波形图。与 MARS 运行的 asm 程序比较，可见寄存器中值均相同；与 C 语言的运行结果也一致。



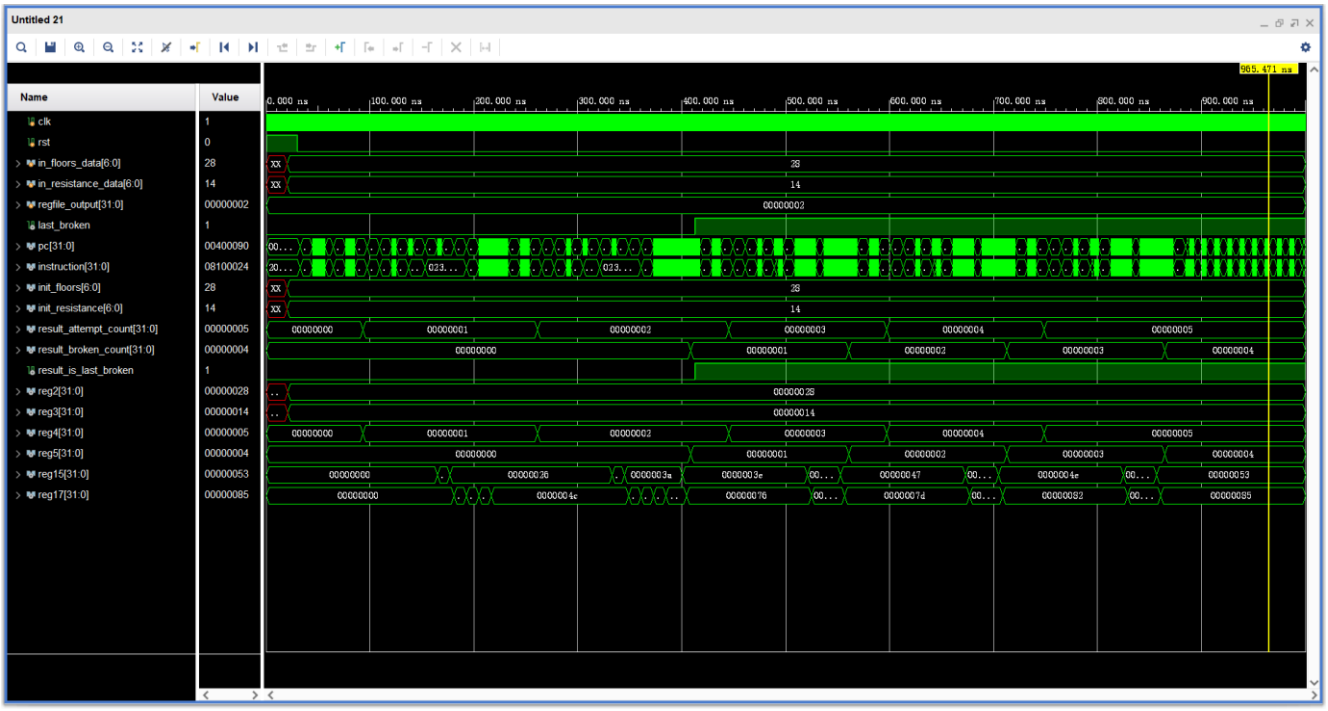
Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000064
$v1	3	0x00000032
$a0	4	0x00000006
$a1	5	0x00000005
$a2	6	0x00000001
$a3	7	0x00000032
$t0	8	0x00000032
$t1	9	0x00000033
$t2	10	0x00000001
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000033
$t6	14	0x00000000
$t7	15	0x000000c0
$s0	16	0x00000002
$s1	17	0x0000014a

```
C:\Users\Administrator\Deskt
请输入楼层数：100
请输入耐摔值：50
进行第1次实验，在50层扔鸡蛋，本次鸡蛋没碎
进行第2次实验，在75层扔鸡蛋，本次鸡蛋摔碎
进行第3次实验，在62层扔鸡蛋，本次鸡蛋摔碎
进行第4次实验，在56层扔鸡蛋，本次鸡蛋摔碎
进行第5次实验，在53层扔鸡蛋，本次鸡蛋摔碎
进行第6次实验，在51层扔鸡蛋，本次鸡蛋摔碎
共尝试6次，摔碎鸡蛋5个，最后一次鸡蛋摔碎
在物质匮乏时期总成本为 192
在人力成本增长时期总成本为 330
--------------------------------
Process exited after 29.64 seconds with return value 0
请按任意键继续. . .
```

五、实验仿真的波形图及某时刻寄存器值的物理意义

1. 流水线波形图

测试初始化楼层高度为40，鸡蛋耐摔值为20时的情况，得到波形图如下：



2. 某时刻寄存器值的物理意义

选择t=6335000μs时刻，分析各个寄存器的值的含义。此时，PC寄存器的值为0x400048，对应END标签。程序已经进入END处的死循环，正确的结果已经被保存至响应的寄存器。

$2 号寄存器的值为 40 对应初始化输入时的楼层总高度；

$3 号寄存器的值为 20，对应鸡蛋耐摔值；

$4 号寄存器的值为 5，代表需要进行 3 次实验；

$5 号寄存器的值为 4，代表摔碎了 4 个鸡蛋；

$6 号寄存器的值为 1，代表最后一次鸡蛋摔碎；

$7 号与$8 号寄存器分别存放二分算法的左边界 l 和右边界 r，此时相等均为 8；

$10 号寄存器通过第一条指令被初始化为常数 1，用于之后的比较；

$13 号寄存器代表自身目前楼层数，在最后一次实验时总是鸡蛋耐摔值+1，用以计算不同时期成本；

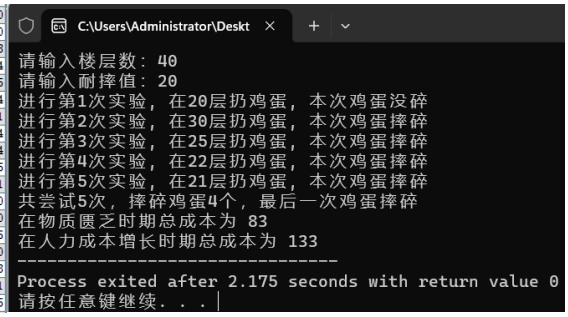
$15 号寄存器为物质匮乏时期总成本为 83，与波形图中的 reg15 最后时刻对应；

$17 号人力成本增长时期总成本为 133，与波形图中的 reg17 最后时刻对应；

$16 为判断上楼还是下楼以计算成本，此刻为 1 可见最后一次实验需要上楼；

上述通过波形图得到的寄存器值与 MARS 中汇编程序运行结果与 C 语言程序（结果时刻）一致：

$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000028
$v1	3	0x00000014
$a0	4	0x00000005
$a1	5	0x00000004
$a2	6	0x00000001
$a3	7	0x00000014
$t0	8	0x00000014
$t1	9	0x00000015
$t2	10	0x00000001
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000015
$t6	14	0x00000000
$t7	15	0x00000053
$s0	16	0x00000001
$s1	17	0x00000085



六、流水线 CPU 实验性能验证模型

实验性能验证模型：比萨塔摔鸡蛋游戏。两个同学在可变换层数的比萨塔上摔鸡蛋，一个同学秘密设定同一批鸡蛋耐摔值；另一个同学在指定层高的比萨塔拿着鸡蛋往下摔，用最少的摔次数和摔破的鸡蛋数求出鸡蛋的耐摔值。假定在耐摔值的楼层及其下面楼层，鸡蛋摔不破，可以重复使用，否则鸡蛋摔破。要求模型的算法输出包括：摔的总次数、摔的总鸡蛋数、最后摔的鸡蛋是否摔破。用你的模型评价该游戏在两个不同历史时期花费的总成本 f=m*p1+n*p2+h*p3，m 为上的楼层总数，n 为下的楼层总数，h 为摔破的鸡蛋总数，p1 为每上 1 层的成本，p2 为每下 1 层的成本，p2 为每个鸡蛋的成本；在物质匮乏时期，p1=2，p2=1，p3=4；在人力成本增长时期，p1=4，p2=1，p3=2。请使用 C 语言设计该验证模型的算法，并把 C 语言汇编为 MIPS 或 RISC-V 指令汇编程序，同时利用编译器生成 MIPS 或 RISC-V 指令集可执行目标程序。

1. C 语言验证程序

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

int main()
{
	int init_floor;
	int init_resistance;
	int location = 1;
	int result_material_cost = 0;
	int result_human_cost = 0;
	printf("请输入楼层数：");
	scanf("%d", &init_floor);

	printf("请输入耐摔值：");
	scanf("%d", &init_resistance);

	int result_attempt_cnt = 0;
	int result_broken_cnt = 0;
	int result_is_last_broken = 0;

	int l = 0, r = init_floor;

	while (l < r) {

		result_attempt_cnt++;

		int mid = (l + r + 1) / 2;
		if (mid > location) {
			result_material_cost += (mid - location) * 2;
			result_human_cost += (mid - location) * 4;
```

```
            }
            else {
                result_material_cost += location - mid;
                result_human_cost += location - mid;
            }
            if (mid > init_resistance) {
                result_material_cost += 4;
                result_human_cost += 2;
                result_is_last_broken = 1;
                r = mid - 1;
                result_broken_cnt++;
            }
            else {
                result_is_last_broken = 0;
                l = mid;
            }
            location = mid;
            printf("进行第%d次实验，在%d层扔鸡蛋，本次鸡蛋", result_attempt_cnt, mid);
            printf(result_is_last_broken ? "摔碎\n" : "没碎\n");
        }

        printf("共尝试%d次，摔碎鸡蛋%d个，最后一次鸡蛋", result_attempt_cnt, result_broken_cnt);
        printf((result_is_last_broken ? "摔碎\n" : "没碎\n"));
        printf("在物质匮乏时期总成本为 %d\n在人力成本增长时期总成本为 %d", result_material_cost, result_human_cost);
        return 0;
}
```

2. MIPS 指令汇编验证程序

```
# 比萨斜塔摔鸡蛋 ASM 汇编程序
# 寄存器用途说明：
# $2 (楼层上限): 建筑物的最大楼层数
# $3 (鸡蛋耐摔值): 鸡蛋的最大可忍受摔落次数
# $4 (总摔鸡蛋次数): 记录总共的摔鸡蛋次数
# $5 (碎裂的鸡蛋数): 记录摔碎的鸡蛋数
# $6 (最后一个鸡蛋是否碎裂): 标志位，1 表示鸡蛋碎裂，0 表示未碎裂
# $7 (左边搜索边界): 二分查找时的左边搜索边界
# $8 (右边搜索边界): 二分查找时的右边搜索边界
# $9 (中间值): 二分查找时的中间值
# $10: 常数 1
.text
    addi $2, $0, 40         # 设置楼层上限 100
    addi $3, $0, 20          # 设置鸡蛋耐摔值 50
    addi $10, $0, 1         # 常数 1（$10=1）
    add $7, $0, $0           # 左边界 l=0
    add $8, $0, $2           # 右边界 r=n
```

```
    addi $13, $0, 1         #自身初始楼层 loc=1

LOOP:
    slt $11, $7, $8         # 检查左边界是否大于等于右边界 ($11=0)
    beq $11, $0, END        # 如果左边界大于等于右边界，跳转到 END ($11=0)
    addi $4, $4, 1          # 增加总摔鸡蛋次数
    add $9, $7, $8
    addi $9, $9, 1
    sra $9, $9, 1           # 计算中间值 mid = (l+r+1)/2
    slt $11, $3, $9         # 检查鸡蛋耐摔值是否大于等于中间值 ($11=0)
    slt $14, $13, $9
    beq $14, $0, COST       # 比较预期摔鸡蛋楼层数与自身楼层数
    sub $16, $9, $13        # 上楼层数
    add $15, $15, $16
    add $15, $15, $16       #计算物质匮乏时期上楼代价
    add $17, $17, $16
    add $17, $17, $16
    add $17, $17, $16
    add $17, $17, $16       #计算人力成本增长时期上楼代价
    add $13, $9, $0         #更新位置 loc

CONTINUE:
    beq $11, $10, BROKE     # 如果鸡蛋耐摔值大于等于中间值，跳转到 BROKE ($11==1)

RESISIT:
    add $6, $0, $0          # 设置最后一个鸡蛋是否碎裂为 0
    add $7, $0, $9          # 更新左边界为中间值
    j LOOP
BROKE:
    addi $15, $15, 4        #鸡蛋摔碎，物质匮乏时期+4
    addi $17, $17, 2        #鸡蛋摔碎，人力成本增长时期+2
    addi $5, $5, 1          # 增加碎裂的鸡蛋数
    add $6, $0, $10         # 设置最后一个鸡蛋是否碎裂为 1
    sub $8, $9, $10         # 更新右边界为中间值减一
    j LOOP
COST:
    sub $16, $13, $9        # 下楼层数
    add $15, $15, $16       # 计算物质匮乏时期下楼代价
    add $17, $17, $16       # 计算人力成本增长时期下楼代价
    add $13, $9, $0         #更新自身位置位置 loc
    j CONTINUE
END:
    #j END                  # 无限循环（转 coe 文件时添加）
.text   ends
```

七、实验验算程序下板测试过程与实现

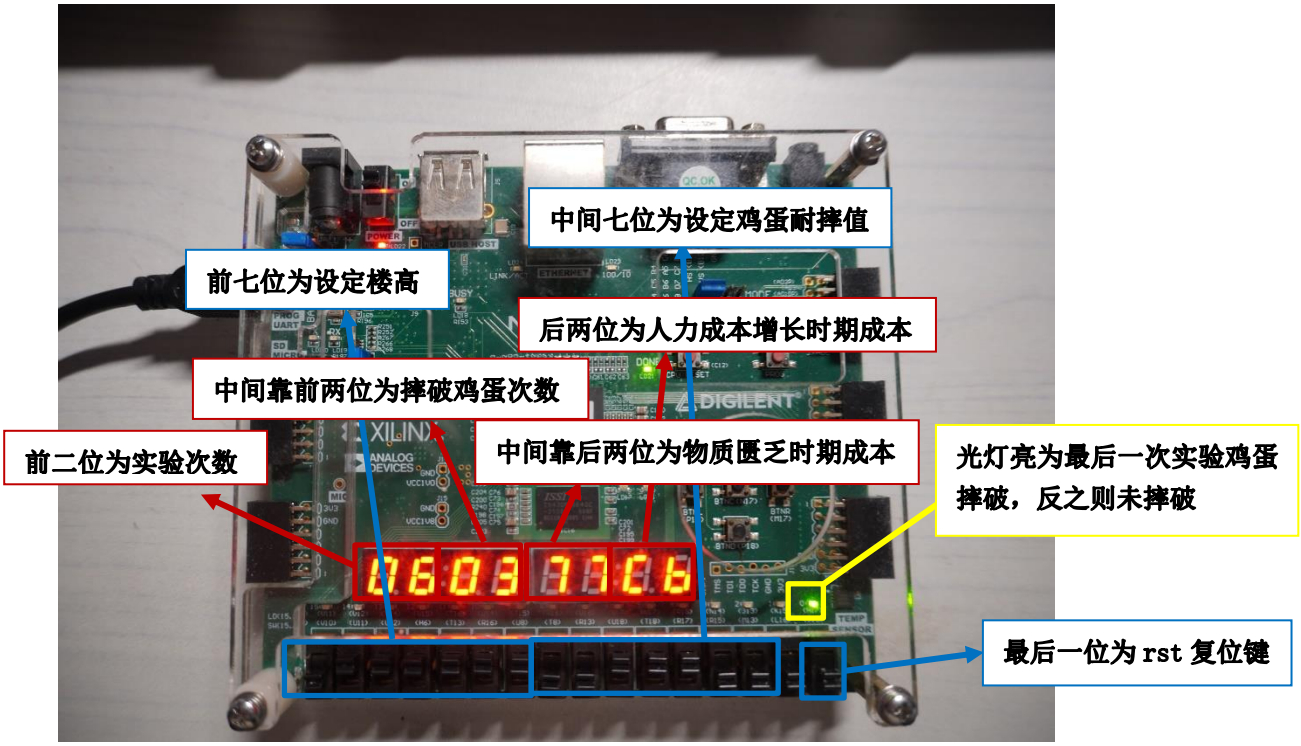
1. 管脚约束文件

```
set_property PACKAGE_PIN T10 [get_ports {o_seg[0]}]
set_property PACKAGE_PIN R10 [get_ports {o_seg[1]}]
set_property PACKAGE_PIN K16 [get_ports {o_seg[2]}]
set_property PACKAGE_PIN K13 [get_ports {o_seg[3]}]
set_property PACKAGE_PIN P15 [get_ports {o_seg[4]}]
set_property PACKAGE_PIN T11 [get_ports {o_seg[5]}]
set_property PACKAGE_PIN L18 [get_ports {o_seg[6]}]
set_property PACKAGE_PIN H15 [get_ports {o_seg[7]}]

set_property PACKAGE_PIN J17 [get_ports {o_sel[0]}]
set_property PACKAGE_PIN J18 [get_ports {o_sel[1]}]
set_property PACKAGE_PIN T9 [get_ports {o_sel[2]}]
set_property PACKAGE_PIN J14 [get_ports {o_sel[3]}]
set_property PACKAGE_PIN P14 [get_ports {o_sel[4]}]
set_property PACKAGE_PIN T14 [get_ports {o_sel[5]}]
set_property PACKAGE_PIN K2 [get_ports {o_sel[6]}]
set_property PACKAGE_PIN U13 [get_ports {o_sel[7]}]

set_property IOSTANDARD LVCMOS33 [get_ports {in_resistance_data[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_resistance_data[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_resistance_data[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_resistance_data[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_resistance_data[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_resistance_data[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {o_seg[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {o_seg[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {o_seg[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {o_seg[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {o_seg[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {o_seg[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {o_seg[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {o_seg[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {o_sel[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {o_sel[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {o_sel[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {o_sel[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {o_sel[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {o_sel[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {o_sel[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {o_sel[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports in_clk]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports in_rst]
set_property IOSTANDARD LVCMOS33 [get_ports result_is_last_broken]

set_property PACKAGE_PIN E3 [get_ports in_clk]
set_property PACKAGE_PIN J15 [get_ports in_rst]
set_property PACKAGE_PIN H17 [get_ports result_is_last_broken]

set_property IOSTANDARD LVCMOS33 [get_ports {in_downfloors_data[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_downfloors_data[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_downfloors_data[0]}]
set_property PACKAGE_PIN V10 [get_ports {in_downfloors_data[2]}]
set_property PACKAGE_PIN U11 [get_ports {in_downfloors_data[1]}]
set_property PACKAGE_PIN U12 [get_ports {in_downfloors_data[0]}]
set_property PACKAGE_PIN H6 [get_ports {in_upfloors_data[5]}]
set_property PACKAGE_PIN T13 [get_ports {in_upfloors_data[4]}]
set_property PACKAGE_PIN R16 [get_ports {in_upfloors_data[3]}]
set_property PACKAGE_PIN U8 [get_ports {in_upfloors_data[2]}]
set_property PACKAGE_PIN T8 [get_ports {in_upfloors_data[1]}]
set_property PACKAGE_PIN R13 [get_ports {in_upfloors_data[0]}]
set_property PACKAGE_PIN R13 [get_ports {in_resistance_data[5]}]
set_property PACKAGE_PIN U18 [get_ports {in_resistance_data[4]}]
set_property PACKAGE_PIN T18 [get_ports {in_resistance_data[3]}]
set_property PACKAGE_PIN R17 [get_ports {in_resistance_data[2]}]
set_property PACKAGE_PIN R15 [get_ports {in_resistance_data[1]}]
set_property PACKAGE_PIN M13 [get_ports {in_resistance_data[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_upfloors_data[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_upfloors_data[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_upfloors_data[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_upfloors_data[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_upfloors_data[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_upfloors_data[0]}]

set_property IOSTANDARD LVCMOS33 [get_ports {in_data[15]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data[14]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data[13]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data[12]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data[4]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {in_data[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_data[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports is_init_floors]

set_property IOSTANDARD LVCMOS33 [get_ports {in_floors_data[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_floors_data[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_floors_data[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_floors_data[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_floors_data[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_floors_data[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_floors_data[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {in_resistance_data[6]}]
set_property PACKAGE_PIN V10 [get_ports {in_floors_data[6]}]
set_property PACKAGE_PIN U11 [get_ports {in_floors_data[5]}]
set_property PACKAGE_PIN U12 [get_ports {in_floors_data[4]}]
set_property PACKAGE_PIN H6 [get_ports {in_floors_data[3]}]
set_property PACKAGE_PIN T13 [get_ports {in_floors_data[2]}]
set_property PACKAGE_PIN R16 [get_ports {in_floors_data[1]}]
set_property PACKAGE_PIN U8 [get_ports {in_floors_data[0]}]
set_property PACKAGE_PIN T8 [get_ports {in_resistance_data[6]}]
```

2. 下板测试过程



下面介绍下板实验的测试流程。上图下方黄色框线内的16个开关（对应V10~J15）表示16位的数据输入，前七位为初始输入的楼层高度，中间七位为鸡蛋耐摔值，最后一位J15为rst复位键，调节楼层高与耐摔值的开关至所需数值，调节复位键至高电平，此时将寄存的楼层高度值与鸡蛋耐摔值打入寄存器堆的对应寄存器（$2与$3），再次调节至低电平，开始计算。计算出的尝试次数与摔破鸡蛋个数，以及物质匮乏时期成本与人力成本增长时期成本分别按顺序显示在七段数码管的其中两位，LED灯H17用于表示最后一个鸡蛋是否摔破。

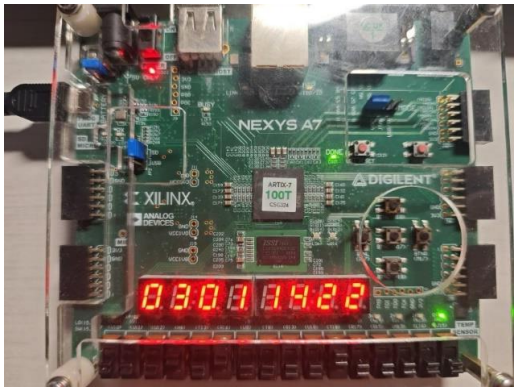
3. 样例测试

① 层高为**30**，耐摔值为**19**：

根据显示结果可得，测试次数为5，摔坏鸡蛋个数为3，最后摔坏了，在物质匮乏时期总成本为 65在人力成本增长时期总成本为 107。与C语言程序相比较可发现结果正确；

```
请输入楼层数：30
请输入耐摔值：19
进行第1次实验，在15层扔鸡蛋，本次鸡蛋没碎
进行第2次实验，在23层扔鸡蛋，本次鸡蛋摔碎
进行第3次实验，在19层扔鸡蛋，本次鸡蛋没碎
进行第4次实验，在21层扔鸡蛋，本次鸡蛋摔碎
进行第5次实验，在20层扔鸡蛋，本次鸡蛋摔碎
共尝试5次，摔碎鸡蛋3个，最后一次鸡蛋摔碎
在物质匮乏时期总成本为 65
在人力成本增长时期总成本为 107
```

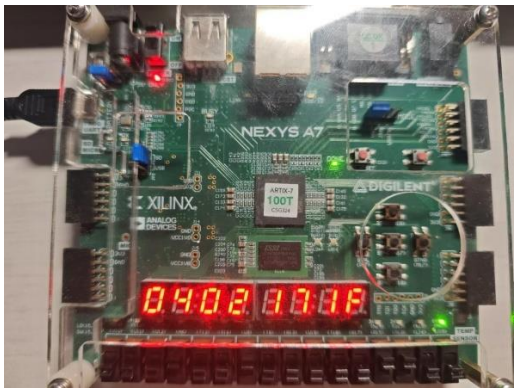
② 层高为**10**，耐摔值为**8**：



根据显示结果可得，测试次数为3，摔坏鸡蛋个数为1，最后摔坏了，在物质匮乏时期总成本为 20在人力成本增长时期总成本为 34。与C语言程序相比较可发现结果正确；

```
请输入楼层数：10
请输入耐摔值：8
进行第1次实验，在5层扔鸡蛋，本次鸡蛋没碎
进行第2次实验，在8层扔鸡蛋，本次鸡蛋没碎
进行第3次实验，在9层扔鸡蛋，本次鸡蛋摔碎
共尝试3次，摔碎鸡蛋1个，最后一次鸡蛋摔碎
在物质匮乏时期总成本为 20
在人力成本增长时期总成本为 34
```

③ 层高为**9**，耐摔值为**3**：



根据显示结果可得，测试次数为4，摔坏鸡蛋个数为2，最后摔坏了。与C语言程序相比较可发现结果正确：

```
请输入楼层数：9
请输入耐摔值：3
进行第1次实验，在5层扔鸡蛋，本次鸡蛋摔碎
进行第2次实验，在2层扔鸡蛋，本次鸡蛋没碎
进行第3次实验，在3层扔鸡蛋，本次鸡蛋没碎
进行第4次实验，在4层扔鸡蛋，本次鸡蛋摔碎
共尝试4次，摔碎鸡蛋2个，最后一次鸡蛋摔碎
在物质匮乏时期总成本为 23
在人力成本增长时期总成本为 31
```

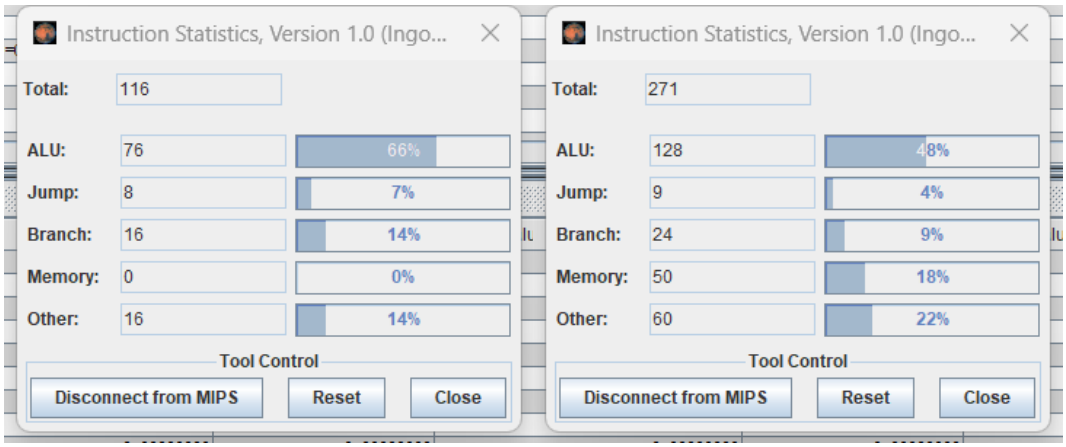
八、流水线的性能指标定性分析

由于比萨斜塔摔鸡蛋的测试程序中不包含访存指令，为了更好地分析流水线性能指标，我选取了两段不同的测试程序。第一个测试程序即以楼层数为 1024、鸡蛋耐摔值为 65 的情况为例；第二个测试程序是如下的冒泡排序：

```
.data
    Array: .space 40
.text
    # 初始化数列
    addi $2, $0, 5
    addi $3, $0, 4
    addi $4, $0, 3
    addi $5, $0, 2
    addi $6, $0, 1
    # 将寄存器中的数值放入内存中以排序
    addi $t0, $0, 0
    sw $2, Array($t0)
    addi $t0, $t0, 4
    sw $3, Array($t0)
    addi $t0, $t0, 4
    sw $4, Array($t0)
    addi $t0, $t0, 4
    sw $5, Array($t0)
    addi $t0, $t0, 4
    sw $6, Array($t0)
    addi $t1, $0, 0 # 控制外层循环
loop1:
    addi $t2, $0, 0 # 控制内层循环
loop2:
    addi $t3, $t2, 4 # 指向下一个数字
    lw $t4, Array($t2)
    lw $t5, Array($t3)
    slt $t0, $t4, $t5 # 比较两数字，若小于则不交换
    bne $t0, $zero, endl2
    sw $t5, Array($t2) # 交换数字
    sw $t4, Array($t3)
endl2:
    addi $t2, $t2, 4
    sll $t3, $t1, 2 # t1 计数 x4
    add $t3, $t2, $t3
    beq $t3, 16, endl1 # 比较循环结束
    j loop2
endl1:
    addi $t1, $t1, 1 # 计数
```

```
    beq $t1, 4, done
    j loop1
done:
    # 将内存中排好序的数组移回寄存器中
    addi $t0, $0, 0 lw $2, Array($t0)
    addi $t0, $t0, 4 lw $3, Array($t0)
    addi $t0, $t0, 4 lw $4, Array($t0)
    addi $t0, $t0, 4 lw $5, Array($t0)
    addi $t0, $t0, 4 lw $6, Array($t0)
end:
```

通过 MARS 分析，两个测试程序的指令条数分别如下（左图与右图）：



1. 吞吐率分析

① 测试程序 1

总共执行的指令数为116条，根据波形图分析得计算所用的总周期数是175：

$$TP_1 = \frac{\text{指令数}n}{\text{时钟周期数}T_k} = \frac{116}{175} = 0.6628$$

② 测试程序 2

总共执行的指令数为271条，根据波形图分析得计算所用的总周期数是502：

$$TP_2 = \frac{\text{指令数}n}{\text{时钟周期数}T_k} = \frac{271}{502} = 0.5398$$

2. 加速比分析

① 测试程序 1

116条指令中，共有 8 条跳转指令、16 条分支指令（仅需要 IF、ID 两阶段），其余 92 条为运算指令（需要 IF、ID、EX、WB 四阶段）：

$$S_1 = \frac{\text{顺序执行时钟周期数}T_S}{\text{流水线执行时钟周期数}T_k} = \frac{92 \times 4 + 24 \times 2\text{个时钟周期}}{175\text{个时钟周期}} = 2.38$$

② 测试程序 2

271条指令中，共有9条跳转指令、24条分支指令（仅需要IF、ID两阶段），188条为运算指令（需要IF、ID、EX、WB四阶段），50条为访存指令（需要IF、ID、EX、MEM、WB五阶段），计算如下：

$$S_1 = \frac{\text{顺序执行时钟周期数}T_S}{\text{流水线执行时钟周期数}T_k} = \frac{50 \times 5 + 188 \times 4 + 33 \times 2\text{个时钟周期}}{502\text{个时钟周期}} = 2.13$$

3. 效率分析

本次流水线共分为 5 段，因此效率计算如下：

$$E_1 = \frac{S}{k} = \frac{104 \times 4 + 19 \times 2}{213 \times 5} = 41.13\%$$

$$E_2 = \frac{S}{k} = \frac{50 \times 5 + 188 \times 4 + 33 \times 2}{502 \times 5} = 42.55\%$$

4. 相关与冲突分析

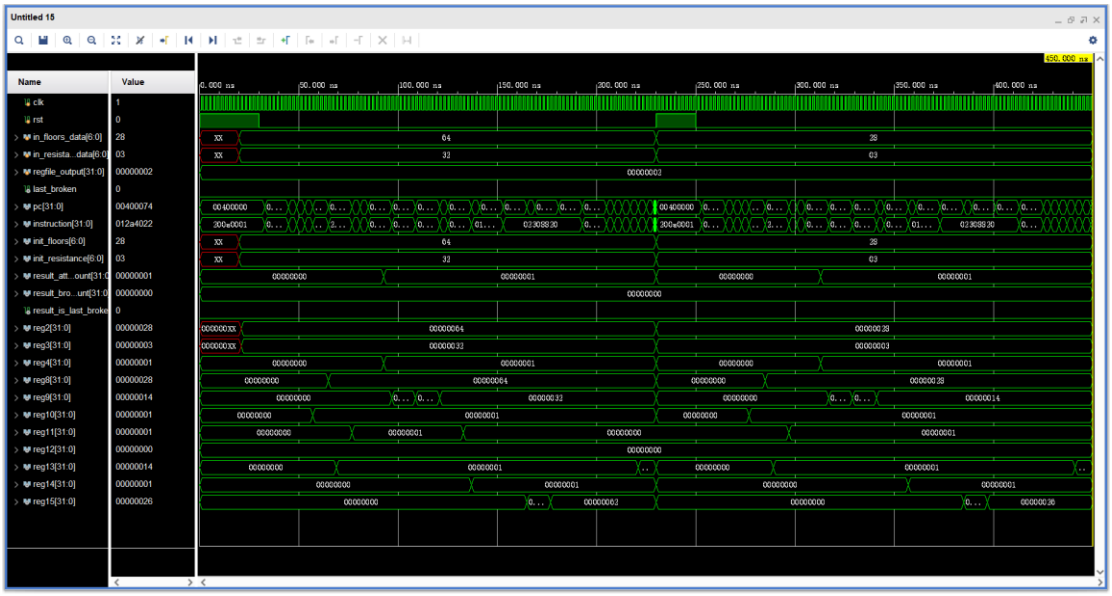
① 数据相关

采用暂停流水线（在流水线中插入流水线气泡）的方法来解决数据相关问题。读寄存器出现在ID段，写寄存器出现在WB段，且由于构造的CPU为静态流水线，所以只会出现先写后读（Read After Write, RAW）的数据相关。由于WB写回在上半个时钟周期，而ID读寄存器值在下半个时钟周期，因此ID-WB不存在RAW冲突。综上所述，需要检测的数据冲突为ID-EX、ID-MEM两种情况。

对于前者，需要插入两个“气泡”，等待当前EX数据执行到WB执行写入寄存器；对于后者，需要插入一个“气泡”，等待下一周期MEM阶段传入到WB阶段写入寄存器后，先写后读冲突解决。

Bkpt	Address	Code	Basic	Source
	0x0040000c	0x20050002	addi $5, $0, 0x00000002	8: addi $5, $0, 2
	0x00400010	0x20060001	addi $6, $0, 0x00000001	9: addi $6, $0, 1
	0x00400014	0x20080000	addi $8, $0, 0x00000000	11: addi $t0, $0, 0
	0x00400018	0x3c011001	lui $1, 0x00001001	12: sw $2, Array($t0)
	0x0040001c	0x00280821	addu $1, $1, $8	
	0x00400020	0xac220000	sw $2, 0x00000000($1)	
	0x00400024	0x21080004	addi $8, $8, 0x00000004	13: addi $t0, $t0, 4
	0x00400028	0x3c011001	lui $1, 0x00001001	14: sw $3, Array($t0)
	0x0040002c	0x00280821	addu $1, $1, $8	
	0x00400030	0xac230000	sw $3, 0x00000000($1)	
	0x00400034	0x21080004	addi $8, $8, 0x00000004	15: addi $t0, $t0, 4
	0x00400038	0x3c011001	lui $1, 0x00001001	16: sw $4, Array($t0)
	0x0040003c	0x00280821	addu $1, $1, $8	
	0x00400040	0xac240000	sw $4, 0x00000000($1)	
	0x00400044	0x21080004	addi $8, $8, 0x00000004	17: addi $t0, $t0, 4
	0x00400048	0x3c011001	lui $1, 0x00001001	18: sw $5, Array($t0)

由于数据相关而插入“气泡”如下图所示。当指令0x400008运行到ID阶段时，指令0x400000运行到IF取指阶段。此时0x400008指令从源寄存器$7，$8取操作数，分别与前两条指令有ID-EX和ID-MEM冲突，因此需要stall两周期。



② 控制相关

对于分支和跳转指令（如J、BEQ、BNE等），采用冻结（freeze）或排空（flush）流水线的方式。当ID阶段检测到分支指令后，就暂停执行其后的指令，将计算得到的新PC值赋予PC，再接着启动流水线。由于提前到ID译码段即进行分支判断，因此使得分支延迟为一个周期。

5. CPU 运行时间

测试程序的时间复杂度与楼高相关。若楼高为n，则时间复杂度为O(logn)。

6. 存储器空间使用

- regfile： 32 个 32 位寄存器，32×32=1Kbit
- IMEM： 2048 个 32 位寄存器，2048×32=64Kbit
- DMEM： 2048 个 32 位寄存器，2048×32=64Kbit

由于本测试程序中未使用数据存储器，因此实际上只用到了寄存器与指令存储器的部分存储空间。

九、总结与体会

在这次实验中，我独立设计并实现了一个基于 MIPS 指令集的八段指令流水线 CPU。整个设计覆盖了指令执行的关键阶段，包括取指（IF）、译码（ID）、执行（EX）、访存（MEM）和写回（WB），并对分支预测、冒险处理等模块进行了优化，以提升 CPU 的并行处理效率。

为了验证 CPU 的正确性和功能性，我构建了一个模拟比萨塔摔鸡蛋的验证程序。程序包括多种指令类型，涵盖了常用的算术、逻辑和分支操作，用于检验流水线设计的完整性和各模块的协作效果。在 Vivado 和 Modelsim 的仿真环境中，通过观察各指令的流水线推进过程，我能够及时发现并解决设计中的潜在问题，确保指令在每个阶段能够被正确执行，并且控制信号在不同指令间能够顺利传递。

完成软件仿真后，我将程序部署到实验板上进行实际测试。实验板验证让我更直观地观察到了流水线的硬件执行效果，包括时钟周期内各阶段的信号波动和数据传递。这种硬件验证帮助我进一步理解了时序设计的重要性，同时也增强了我对数据流在硬件电路中的实际传输过程的理解。

在性能测试环节中，我分析了该流水线 CPU 的吞吐率、加速比和资源利用率。结果表明，在指令序列较长时，流水线 CPU 能显著提升指令执行速度，但在处理频繁分支和依赖关系较强的指令时，性能受到了一定影响。这次分析让我更好地理解了流水线设计的优势与局限性，也促使我探索进一步优化数据前递和分支预测机制的方法，以提高在特殊场景下的处理效率。

通过本次实验，我不仅掌握了 MIPS 指令流水线的设计要点，还在实践中提升了硬件编程的能力。实验的不同环节加深了我对 CPU 架构和性能优化的理解，让我体会到硬件设计的细致与挑战。我希望在后续学习中进一步拓展自己的技能，将理论与实践更好地结合起来，不断改进和优化自己的设计能力。

十、附件（所有程序）