|  |
| --- |
| **实验成绩：** |
| **教 师：** |

**计算机体系结构**

**实 验 报 告**

**姓名：\_\_\_\_\_\_常添\_\_\_\_\_\_\_**

**班号：\_\_\_\_2203102\_\_\_\_\_\_**

**学号：\_\_\_2022111699\_\_\_\_**

**哈尔滨工业大学计算学部**

**2024 年 9月**

**实验要求**

1. 实验必须提前预习，不预习不允许参加实验。
2. 实验内容按时完成，教师在课上根据完成情况给出成绩。
3. 实验不得缺席，否则将不参加最终成绩的评定；任何一个实验题目不能完成也不参加最终成绩的评定。
4. 实验环节考核不通过者，不得参加考试，也不参加最终成绩的评定。
5. 实验如果不能在指定时间完成，需降低评分，但要求必须完成。
6. 可根据设计的创新情况作适当加分处理。
7. 按时完成实验报告，在每个实验结束后一周内完成实验报告。

**实验报告撰写规范**

1. 实验方案部分包括系统设计要求、系统概述、技术方案、关键技术和进度安排等内容。
2. 实验设计部分包括结构设计（含系统框图和各部件功能概述）、系统工作原理描述（工作过程简述）、外接口定义（管脚定义及功能）、系统详细设计（各部件功能详述，设计方法，Verilog程序等）、系统测试（测试方案，测试波形等）等内容。
3. 实验测试部分包括测试准备条件、系统功能概述、系统功能测试（每项功能作为一个部分，要包括功能描述、测试过程和期望的测试结果）。

**实验项目表**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 序号 | 实 验 项 目 | 学时 | 实验要求 | 实验类别 | 每组人数 |
| 1 | 流水线处理器设计 | 8 | 必修 | 设计 | 1 |
| 2 | 分支预测器 | 8 | 必修 | 设计 | 1 |
| 3 | 指令Cache的设计与实现 | 8 | 必修 | 设计 | 1 |

**实验一 流水线处理器**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 预习成绩 | 实验成绩 | 报告成绩 | 总成绩 | 教师签字 |
|  |  |  |  |  |

一、实验目的

1．掌握 Vivado 集成开发环境

2．掌握Verilog语言

3．掌握FPGA编程方法及硬件调试手段

4．深刻理解流水线处理器结构和数据冲突解决技术的工作原理

二、实验环境（实验设备、开发环境）

**Personal Computer：**

Processor：13th Gen Intel(R) Core(TM) 2.20 GHz

Installed RAM：16.0 GB (15.7 GB usable)

System type： 64-bit operating system, x64-based processor

**Xilinx FPGA**：xc7a200tfbg676-2

**Vivado Design Suite: version 2019.2**

**龙芯实验平台**

三、设计思想（实验预习）（如空白不够，可自行加页）

1． 五段流水线的寄存器文件描述

IF/ID段

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 寄存器名 | 位数 | 方向 | 来源/去向 | 意义 |
| if\_id\_pc | 32 位 | IF → ID | 由 program\_counter 写入，供后续 ID 阶段读取 | 保存取指阶段的下一条指令的 PC 值，用于在 ID 阶段进行译码时知道当前指令来自的地址 |
| if\_id\_instruction (\*为 wire) | 32 位 | IF → ID | 由 inst\_sram\_rdata 提供，供 ID 阶段使用 | 保存取指阶段读到的指令内容（机器码），在 ID 阶段执行指令译码和后续操作 |
| debug\_if\_id\_pc | 32 位 | IF → ID（调试用） | 由 program\_counter 写入，供调试监控 | 仅用于调试环境下查看 IF 阶段当前指令的 PC 值，便于问题定位与排查 |

ID/EX段

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 寄存器名 | 位数 | 方向 | 来源/去向 | 意义 |
| id\_ex\_pc | 32 位 | ID → EX | 由 if\_id\_pc 写入，供 EX 阶段读取 | 保存 ID 阶段的 PC 值，便于 EX 阶段进行分支跳转或其他可能需要当前 PC 的操作 |
| id\_ex\_instruction | 32 位 | ID → EX | 由 if\_id\_instruction 提供，供 EX 阶段使用 | 保存指令译码阶段获取的指令机器码，为 EX 阶段的运算或控制判断提供参考 |
| id\_ex\_reg1\_data | 32 位 | ID → EX | 由 regfile\_read\_data1 读取并传递到 EX | 保存从寄存器堆中读出的源操作数 1 数据，供 EX 阶段的 ALU 运算等使用 |
| id\_ex\_reg2\_data | 32 位 | ID → EX | 由 regfile\_read\_data2 读取并传递到 EX | 保存从寄存器堆中读出的源操作数 2 数据，供 EX 阶段的 ALU 运算或 Store 操作使用 |
| id\_ex\_immediate | 32 位 | ID → EX | 由 extended\_immediate 提供，传递到 EX | 保存符号扩展后的立即数，供 EX 阶段进行地址计算、运算偏移量或其他相关操作 |
| debug\_id\_ex\_pc | 32 位 | ID → EX（调试用） | 由 debug\_if\_id\_pc 写入，供调试监控 | 仅用于调试时查看 ID 阶段的 PC 值，便于问题定位与排查 |

EX/MEM段

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 寄存器名 | 位数 | 方向 | 来源/去向 | 意义 |
| ex\_mem\_result | 32 位 | EX → MEM | 在 EX 阶段由 ALU 或跳转控制逻辑计算产生，供 MEM 阶段使用 | 保存 EX 阶段的计算结果：ALU 算术逻辑运算结果 |
| ex\_mem\_reg2\_data | 32 位 | EX → MEM | 在 EX 阶段取自 selected\_operand\_b，供 MEM 阶段使用（如 Store） | 保存 EX 阶段读出的第二个操作数（或经旁路选出的数据），通常在执行 Store 指令时需要写入内存的数据 |
| ex\_mem\_instruction | 32 位 | EX → MEM | 在 EX 阶段由 id\_ex\_instruction 传递并可能有修改 | 保存即将进入 MEM 阶段的指令机器码，用于判断在 MEM 阶段是否执行 Load/Store 等操作 |
| ex\_mem\_jump\_flag | 1 位 | EX → MEM | 由 EX 阶段的跳转判断单元（jump\_control）产生 | 指示是否满足跳转条件，MEM 阶段可能需要根据该信号进行流水线相关控制 |
| debug\_ex\_mem\_pc | 32 位 | EX → MEM（调试用） | 在 EX 阶段由 debug\_id\_ex\_pc 传递 | 仅用于调试，记录 EX 阶段的 PC 值，方便在 MEM 阶段或后续进行调试、监控 |

MEM/WB段

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 寄存器名 | 位数 | 方向 | 来源/去向 | 意义 |
| mem\_wb\_result | 32 位 | MEM → WB | 在 MEM 阶段由 ex\_mem\_result 传递到 WB 阶段 | 保存从 EX/MEM 阶段传来的运算结果，在 WB 阶段可能写回寄存器堆 |
| mem\_wb\_instruction | 32 位 | MEM → WB | 在 MEM 阶段由 ex\_mem\_instruction 传递到 WB | 保存即将进行写回（WB）的指令机器码，用于在 WB 阶段判断是否需要写回、写到哪个寄存器 |
| mem\_wb\_memory\_data | 32 位 | MEM → WB (wire) | 由 data\_sram\_rdata 读取，用于 WB 阶段 | 当指令是 Load 时，从数据存储器中取出的数据，供 WB 阶段选择写回寄存器堆 |
| debug\_mem\_wb\_pc | 32 位 | MEM → WB（调试用） | 在 MEM 阶段由 debug\_ex\_mem\_pc 传递 | 仅用于调试环境下查看 MEM 阶段指令对应的 PC 值，便于进行调试与监控 |

1. 给出定向控制的设计方案，设计方案要求包括：
2. 比较和定向操作

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 包含定向源的流水线寄存器 | 定向源相应指令的操作码 | 包含定向目标的流水线寄存器 | 定向目标相应指令的操作码 | 定向的目标 | 比较操作（如果相等就定向） |
| **ID/EX** <br/>（主要是 selected\_operand\_a、selected\_operand\_b、<br/>id\_ex\_instruction） | 6'b111110 <br/>(比较指令) | **EX/MEM** <br/>（ex\_mem\_result, ex\_mem\_jump\_flag） | - 该行仅用于“比较指令”本身 <br/>（可能与后续“有条件跳转”配合） | 影响后续跳转决策 | 在 **EX** 阶段，通过 ALU 或 jump\_control 进行比较：<br/>• 若“相等”或满足特定比较条件，则会把结果（如等于标志）带入后续流水线；<br/>• 供后续“有条件跳转”(6'b111111) 使用，以决定是否真正跳转。 |
| **ID/EX** <br/>（同上，取到指令和操作数） | 6'b111111 <br/>(有条件跳转) | **EX/MEM** <br/>（ex\_mem\_result, ex\_mem\_jump\_flag） | - 有条件跳转：6'b111111 | program\_counter（PC） | |  | | --- | |  |  |  | | --- | | 在 **EX** 阶段判断：<br/>1. 先看比较结果（若之前是比较指令或与 id\_ex\_instruction 自带的类型判断）<br/>2. 若“相等”/“满足条件”，jump\_condition\_met=1；<br/>3. 流入 **EX/MEM** 阶段后，若 ex\_mem\_jump\_flag=1，则用 ex\_mem\_result（跳转地址）更新 PC，完成“如果相等就定向”的功能。 | |
| **ID/EX** <br/>（同上，取到指令和操作数） | 6'b000010 <br/>(无条件跳转) | **EX/MEM** <br/>（ex\_mem\_result, ex\_mem\_jump\_flag） | - 无条件跳转：6'b000010 | program\_counter（PC） | 在 **EX** 阶段直接计算跳转目标地址（例如 PC[31:28] + 指令中 [25:0] + 2’b00），将其写入 ex\_mem\_result，同时 ex\_mem\_jump\_flag 置 1；<br/>到 **EX/MEM** 阶段若 ex\_mem\_jump\_flag=1，就用 ex\_mem\_result 更新 PC，**不需要比较**，立即定向到目标地址。 |

1. 流水线增设的定向路径

| **信号 / 寄存器 / 组件** | **阶段** | **方向** | **作用 / 功能** | **备注** |
| --- | --- | --- | --- | --- |
| jump\_control 模块 | EX | 读自 id\_ex\_instruction 和操作数 | 负责对操作数（selected\_operand\_a/b）做比较，并根据指令操作码（如 6'b111110/6'b111111 等）决定是否跳转。 | 若满足“相等”或其他跳转条件，即输出 jump\_condition\_met = 1。 |
| jump\_condition\_met | EX | EX → EX/MEM | 表示 EX 阶段的比较结果：1 表示需要跳转，0 表示不跳转。 | 后续会赋给 ex\_mem\_jump\_flag。 |
| ex\_mem\_jump\_flag | EX/MEM 寄存器 | EX → MEM | 在 EX 阶段赋值，指示是否确实要进行跳转。 | 若为 1，则在 **IF** 阶段用跳转地址更新 program\_counter。 |
| ex\_mem\_result | EX/MEM 寄存器 | EX → MEM | 存放 EX 阶段计算出的跳转目标地址（例如无条件跳转 J 指令拼接出的地址，或有条件跳转分支目标）。 | 若 ex\_mem\_jump\_flag = 1，则该值会在 IF 阶段被选为新 PC。 |
| if\_stage\_mux (多路选择器) | IF | EX/MEM → IF | 在 IF 阶段，用 ex\_mem\_jump\_flag 作为选择信号：<br/>• 为 1 时选 ex\_mem\_result；<br/>• 为 0 时选 (program\_counter + 4)。 | 该多路选择器输出即 pc\_mux\_result，随后赋值给 next\_pc。 |
| next\_pc | IF | IF → 寄存器 program\_counter | 若跳转，则 next\_pc = ex\_mem\_result；否则 next\_pc = program\_counter + 4。 | 在下一个时钟沿存入 program\_counter，完成真正的定向/跳转。 |
| program\_counter | IF（全局） | 不断更新（写入 next\_pc） | 流水线中用于取指的 PC 寄存器，一旦更新，下一拍就会从新地址取指执行。 | 若跳转成功，则 PC 改变为跳转目标地址；否则顺序执行。 |

**EX 阶段：**

根据指令操作码以及寄存器操作数，jump\_control 模块判断是否需要跳转（例如比较“是否相等”）。若满足跳转条件，则输出 jump\_condition\_met = 1；同时在 EX 阶段计算出目标地址（存到 ex\_mem\_result）。

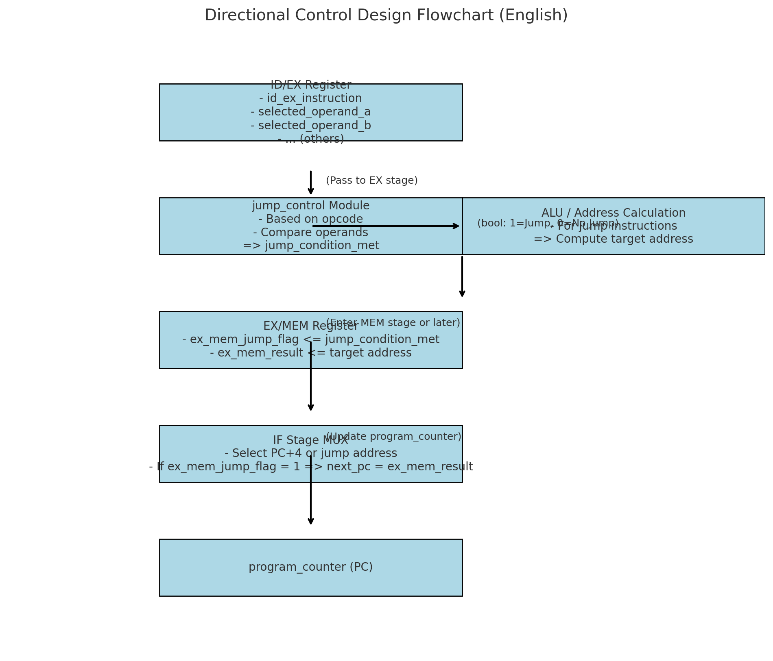
**EX/MEM阶段：**

将 jump\_condition\_met 存入 ex\_mem\_jump\_flag；将跳转目标地址继续保存在 ex\_mem\_result。

**IF 阶段：**

有一个多路选择器（if\_stage\_mux），若 ex\_mem\_jump\_flag = 1，则令 next\_pc = ex\_mem\_result；否则令 next\_pc = program\_counter + 4。下一个时钟沿时，program\_counter 被赋值为 next\_pc。如果是跳转，则实现“定向”；否则按顺序执行下一条指令。

1. 定向控制设计框图及功能描述



**功能描述**

* **ID 阶段**

CPU 从指令存储器取出指令，将其暂存在 IF/ID 级寄存器。读取寄存器堆中的操作数 (reg1, reg2)，生成 selected\_operand\_a、selected\_operand\_b 等。写入 ID/EX 寄存器，带到 EX 阶段。

* **EX 阶段**

**jump\_control 模块**读取 id\_ex\_instruction 的操作码，并查看 selected\_operand\_a、selected\_operand\_b：

若是**比较指令** (opcode=6'b111110)，进行操作数比较，产生结果（如“相等”标志）。

若是**有条件跳转指令** (opcode=6'b111111)，先依赖比较结果确定是否跳转 (jump\_condition\_met)。

若是**无条件跳转指令** (opcode=6'b000010)，则直接跳转 (jump\_condition\_met=1)。

同时，ALU 或专门的跳转地址计算单元会计算跳转目标地址 (对 J 指令可能拼接 PC 的高位与指令低 26 位，对 BEQ/条件跳转则可能是 PC + offset 等)。将得到的 “跳转目标地址” 写入 ex\_mem\_result，并将 “是否跳转” 写入 ex\_mem\_jump\_flag。

* **EX/MEM 级 -> IF 阶段**

**IF Stage**有一个多路选择器 (MUX)，根据 ex\_mem\_jump\_flag 判断是否取 ex\_mem\_result 作为新 PC：

若 ex\_mem\_jump\_flag == 1，表示需要跳转，则 next\_pc = ex\_mem\_result；

否则，next\_pc = program\_counter + 4（顺序执行）。

下一个时钟上升沿时，将 next\_pc 写入 program\_counter。若跳转，则**定向**到目标地址执行新的指令；否则继续顺序执行。

1. 代码实现

写出带有详细注释的Verilog实现代码：

**cpu.v**

module cpu(

input clk, // Clock signal

input resetn, // Active-low reset signal

output inst\_sram\_en, // Instruction memory enable

output[31:0] inst\_sram\_addr, // Instruction memory address

input[31:0] inst\_sram\_rdata, // Instruction memory read data

output data\_sram\_en, // Data memory enable

output[3:0] data\_sram\_wen, // Data memory write enable

output[31:0] data\_sram\_addr, // Data memory address

output[31:0] data\_sram\_wdata, // Data memory write data

input[31:0] data\_sram\_rdata, // Data memory read data

// For testing environment CPU correctness check

output[31:0] debug\_wb\_pc, // Current PC of the executing instruction

output debug\_wb\_rf\_wen, // Current register file write enable signal

output[4:0] debug\_wb\_rf\_wnum, // Current register file write-back register address

output[31:0] debug\_wb\_rf\_wdata // Current data to write back to the register file

);

// ===== Global Definitions =====

reg[31:0] program\_counter;

reg[31:0] instruction\_register;

initial begin

program\_counter = 0;

instruction\_register = 0;

end

// ===============================

// ===== IF/ID Stage =====

reg[31:0] if\_id\_pc;

wire[31:0] if\_id\_instruction;

reg[31:0] debug\_if\_id\_pc; // Debugging purpose

// ==========================

// ===== ID/EX Stage =====

reg[31:0] id\_ex\_pc;

reg[31:0] id\_ex\_instruction;

reg[31:0] id\_ex\_reg1\_data;

reg[31:0] id\_ex\_reg2\_data;

reg[31:0] id\_ex\_immediate;

reg[31:0] debug\_id\_ex\_pc; // Debugging purpose

// ==========================

// ===== EX/MEM Stage =====

reg[31:0] ex\_mem\_result;

reg[31:0] ex\_mem\_reg2\_data;

reg[31:0] ex\_mem\_instruction;

reg ex\_mem\_jump\_flag;

reg[31:0] debug\_ex\_mem\_pc; // Debugging purpose

// ==========================

// ===== MEM/WB Stage =====

reg[31:0] mem\_wb\_result;

reg[31:0] mem\_wb\_instruction;

wire[31:0] mem\_wb\_memory\_data;

reg[31:0] debug\_mem\_wb\_pc; // Debugging purpose

// ==========================

// ===== Hazard Detection =====

wire stall\_signal;

wire forward\_ex\_mem\_rs1;

wire forward\_mem\_wb\_rs1;

wire forward\_mem\_wb\_mm1;

wire forward\_ex\_mem\_rs2;

wire forward\_mem\_wb\_rs2;

wire forward\_mem\_wb\_mm2;

// ==========================

// ===== IF Stage =====

assign inst\_sram\_en = !stall\_signal && resetn;

assign inst\_sram\_addr = program\_counter;

wire[31:0] next\_pc;

wire[31:0] pc\_mux\_result;

data\_mux if\_stage\_mux(

.input0 (program\_counter + 4),

.input1 (ex\_mem\_result),

.select\_line (ex\_mem\_jump\_flag),

.output\_data (pc\_mux\_result)

);

assign next\_pc = pc\_mux\_result;

assign if\_id\_instruction = inst\_sram\_rdata;

always @(posedge clk) begin

if (!stall\_signal) begin

program\_counter <= {32{resetn}} & next\_pc;

if\_id\_pc <= {32{resetn}} & next\_pc;

debug\_if\_id\_pc <= {32{resetn}} & program\_counter;

end

end

// ==========================

// ===== ID Stage =====

detector hazard\_detector(

.current\_instruction(if\_id\_instruction),

.previous\_instruction(id\_ex\_instruction),

.two\_before\_instruction(ex\_mem\_instruction),

.three\_before\_instruction(mem\_wb\_instruction),

.stall\_signal(stall\_signal),

.forward\_ex\_mem\_rs1(forward\_ex\_mem\_rs1),

.forward\_mem\_wb\_rs1(forward\_mem\_wb\_rs1),

.forward\_mem\_wb\_mm1(forward\_mem\_wb\_mm1),

.forward\_ex\_mem\_rs2(forward\_ex\_mem\_rs2),

.forward\_mem\_wb\_rs2(forward\_mem\_wb\_rs2),

.forward\_mem\_wb\_mm2(forward\_mem\_wb\_mm2)

);

wire regfile\_write\_enable;

wire[5:0] regfile\_write\_address;

wire[31:0] regfile\_write\_data;

wire[31:0] regfile\_read\_data1;

wire[31:0] regfile\_read\_data2;

register\_file reg\_file(

.clock(clk),

.write\_enable(regfile\_write\_enable),

.read\_address1(if\_id\_instruction[25:21]),

.read\_address2(if\_id\_instruction[20:16]),

.write\_address(regfile\_write\_address),

.write\_data(regfile\_write\_data),

.read\_data1(regfile\_read\_data1),

.read\_data2(regfile\_read\_data2)

);

wire[31:0] extended\_immediate;

sign\_extend imm\_extender(

.input\_data(if\_id\_instruction[15:0]),

.output\_data(extended\_immediate)

);

always @(posedge clk) begin

id\_ex\_pc <= {32{resetn}} & if\_id\_pc;

debug\_id\_ex\_pc <= {32{resetn}} & debug\_if\_id\_pc;

if (stall\_signal) begin

id\_ex\_instruction <= 0;

id\_ex\_reg1\_data <= 0;

id\_ex\_reg2\_data <= 0;

id\_ex\_immediate <= 0;

end else begin

id\_ex\_reg1\_data <= {32{resetn}} & regfile\_read\_data1;

id\_ex\_reg2\_data <= {32{resetn}} & regfile\_read\_data2;

id\_ex\_instruction <= {32{resetn}} & if\_id\_instruction;

id\_ex\_immediate <= {32{resetn}} & extended\_immediate;

end

end

// ==========================

// ===== EX Stage =====

wire[31:0] alu\_operand\_a, selected\_operand\_a;

wire[31:0] alu\_operand\_b, selected\_operand\_b;

assign selected\_operand\_a =

forward\_ex\_mem\_rs1 ? ex\_mem\_result :

forward\_mem\_wb\_rs1 ? mem\_wb\_result :

forward\_mem\_wb\_mm1 ? mem\_wb\_memory\_data :

id\_ex\_reg1\_data;

assign selected\_operand\_b =

forward\_ex\_mem\_rs2 ? ex\_mem\_result :

forward\_mem\_wb\_rs2 ? mem\_wb\_result :

forward\_mem\_wb\_mm2 ? mem\_wb\_memory\_data :

id\_ex\_reg2\_data;

wire alu\_a\_select, alu\_b\_select;

data\_mux ex\_stage\_mux1(

.input0(id\_ex\_pc),

.input1(selected\_operand\_a),

.select\_line(alu\_a\_select),

.output\_data(alu\_operand\_a)

);

assign alu\_a\_select =

(id\_ex\_instruction[31:26] == 6'b000000) | // Arithmetic instructions need R1

(id\_ex\_instruction[31:26] == 6'b101011) | // Store instructions need R1

(id\_ex\_instruction[31:26] == 6'b100011) | // Load instructions need R1

(id\_ex\_instruction[31:26] == 6'b111110) | // Comparison instructions need R1

(id\_ex\_instruction[31:26] == 6'b111111); // Conditional instructions need R1

data\_mux ex\_stage\_mux2(

.input0(id\_ex\_immediate),

.input1(selected\_operand\_b),

.select\_line(alu\_b\_select),

.output\_data(alu\_operand\_b)

);

assign alu\_b\_select =

(id\_ex\_instruction[31:26] == 6'b000000) | // Arithmetic instructions need R2

(id\_ex\_instruction[31:26] == 6'b111110); // Comparison instructions need R2

wire[31:0] alu\_result;

wire[5:0] alu\_operation =

({6{id\_ex\_instruction[31:26] == 6'b000000}} & id\_ex\_instruction[5:0]) | // Arithmetic operation code

({6{id\_ex\_instruction[31:26] == 6'b111110}} & 6'b111110) | // Comparison operation code

({6{id\_ex\_instruction[31:26] == 6'b101011}} & 6'b100000) | // Store instruction as addition

({6{id\_ex\_instruction[31:26] == 6'b100011}} & 6'b100000); // Load instruction as addition

my\_alu ex\_alu(

.InputA(alu\_operand\_a),

.InputB(alu\_operand\_b),

.Result(alu\_result),

.ShiftAmount(id\_ex\_instruction[10:6]),

.Operation(alu\_operation)

);

wire jump\_condition\_met;

jump\_control jump\_check(

.reg1(selected\_operand\_a),

.reg2(selected\_operand\_b),

.instruction(id\_ex\_instruction),

.execute\_jump(jump\_condition\_met)

);

always @(posedge clk) begin

ex\_mem\_reg2\_data <= {32{resetn}} & selected\_operand\_b;

ex\_mem\_instruction <= {32{resetn}} & (

{32{!(id\_ex\_instruction[31:26] == 6'b000000 && id\_ex\_instruction[5:0] == 6'b001010 && selected\_operand\_b != 0)}}

) & id\_ex\_instruction;

ex\_mem\_result <= {32{resetn}} & (

({32{id\_ex\_instruction[31:26] == 6'b000000}} & alu\_result) | // Arithmetic instructions use ALU

({32{id\_ex\_instruction[31:26] == 6'b100011}} & alu\_result) | // Load instructions use ALU

({32{id\_ex\_instruction[31:26] == 6'b101011}} & alu\_result) | // Store instructions use ALU

({32{id\_ex\_instruction[31:26] == 6'b111110}} & alu\_result) | // Comparison instructions use ALU

({32{id\_ex\_instruction[31:26] == 6'b000010}} & {id\_ex\_pc[31:28], id\_ex\_instruction[25:0], 2'b00}) // Unconditional jump

);

ex\_mem\_jump\_flag <= resetn & jump\_condition\_met;

debug\_ex\_mem\_pc <= {32{resetn}} & debug\_id\_ex\_pc;

end

// ==========================

// ===== MEM Stage =====

assign data\_sram\_addr = ex\_mem\_result;

assign data\_sram\_wdata = ex\_mem\_reg2\_data;

assign data\_sram\_wen = ex\_mem\_instruction[31:26] == 6'b101011; // Only store instructions write

assign data\_sram\_en =

(ex\_mem\_instruction[31:26] == 6'b100011) | // Load instructions access memory

(ex\_mem\_instruction[31:26] == 6'b101011); // Store instructions access memory

assign mem\_wb\_memory\_data = {32{resetn}} & data\_sram\_rdata;

always @(posedge clk) begin

mem\_wb\_instruction <= {32{resetn}} & ex\_mem\_instruction;

mem\_wb\_result <= {32{resetn}} & ex\_mem\_result;

debug\_mem\_wb\_pc <= {32{resetn}} & debug\_ex\_mem\_pc;

end

// ==========================

// ===== WB Stage =====

wire wb\_mux\_select;

data\_mux wb\_stage\_mux(

.input0(mem\_wb\_result),

.input1(mem\_wb\_memory\_data),

.select\_line(wb\_mux\_select),

.output\_data(regfile\_write\_data)

);

assign regfile\_write\_address =

({32{mem\_wb\_instruction[31:26] == 6'b100011}} & mem\_wb\_instruction[20:16]) | // Load writes back to reg[20:16]

({32{mem\_wb\_instruction[31:26] == 6'b000000}} & mem\_wb\_instruction[15:11]) | // Arithmetic writes back to reg[15:11]

({32{mem\_wb\_instruction[31:26] == 6'b111110}} & mem\_wb\_instruction[15:11]); // Comparison writes back to reg[15:11]

assign wb\_mux\_select =

(mem\_wb\_instruction[31:26] == 6'b100011); // Only load instructions use memory data

assign regfile\_write\_enable =

((mem\_wb\_instruction[31:26] == 6'b000000) | // Arithmetic writes back

(mem\_wb\_instruction[31:26] == 6'b100011) | // Load writes back

(mem\_wb\_instruction[31:26] == 6'b111110)) & // Comparison writes back

(regfile\_write\_address != 0); // Cannot write to r0

assign debug\_wb\_pc = debug\_mem\_wb\_pc;

assign debug\_wb\_rf\_wen = regfile\_write\_enable;

assign debug\_wb\_rf\_wnum = regfile\_write\_address;

assign debug\_wb\_rf\_wdata = regfile\_write\_data;

endmodule

**data\_mux.v**

`timescale 1ns / 1ps

module data\_mux(

input [31:0] input0,

input [31:0] input1,

input select\_line,

output [31:0] output\_data

);

assign output\_data = select\_line ? input1 : input0;

endmodule

**detector.v**

module detector (

input [31:0] current\_instruction, // Current instruction, IF/ID

input [31:0] previous\_instruction, // Previous instruction, ID/EX

input [31:0] two\_before\_instruction, // Two instructions before, EX/MEM

input [31:0] three\_before\_instruction, // Three instructions before, MEM/WB

output stall\_signal,

output forward\_ex\_mem\_rs1,

output forward\_mem\_wb\_rs1,

output forward\_mem\_wb\_mm1,

output forward\_ex\_mem\_rs2,

output forward\_mem\_wb\_rs2,

output forward\_mem\_wb\_mm2

);

// Stall Signal

assign stall\_signal =

(previous\_instruction[31:26] == 6'b100011 && (current\_instruction[31:26] == 6'b000000 || current\_instruction[31:26] == 6'b111110) && previous\_instruction[20:16] == current\_instruction[25:21]) |

(previous\_instruction[31:26] == 6'b100011 && (current\_instruction[31:26] == 6'b000000 || current\_instruction[31:26] == 6'b111110) && previous\_instruction[20:16] == current\_instruction[20:16]) |

(previous\_instruction[31:26] == 6'b100011 && previous\_instruction[20:16] == current\_instruction[25:21]);

// Forward EX/MEM.RS to ID/EX.R1

assign forward\_ex\_mem\_rs1 = (previous\_instruction[25:21] != 5'b00000) &&

(two\_before\_instruction[31:26] == 6'b000000 && previous\_instruction[25:21] == two\_before\_instruction[15:11]) | // Arithmetic instruction

(two\_before\_instruction[31:26] == 6'b111110 && previous\_instruction[25:21] == two\_before\_instruction[15:11]); // Comparison instruction

// Forward EX/MEM.RS to ID/EX.R2

assign forward\_ex\_mem\_rs2 = (previous\_instruction[20:16] != 5'b00000) &&

(two\_before\_instruction[31:26] == 6'b000000 && previous\_instruction[20:16] == two\_before\_instruction[15:11]) | // Arithmetic instruction

(two\_before\_instruction[31:26] == 6'b111110 && previous\_instruction[20:16] == two\_before\_instruction[15:11]); // Comparison instruction

// Forward MEM/WB.RS to ID/EX.R1

assign forward\_mem\_wb\_rs1 = (previous\_instruction[25:21] != 5'b00000) &&

(three\_before\_instruction[31:26] == 6'b000000 && previous\_instruction[25:21] == three\_before\_instruction[15:11]) | // Arithmetic instruction

(three\_before\_instruction[31:26] == 6'b111110 && previous\_instruction[25:21] == three\_before\_instruction[15:11]); // Comparison instruction

// Forward MEM/WB.RS to ID/EX.R2

assign forward\_mem\_wb\_rs2 = (previous\_instruction[20:16] != 5'b00000) &&

(three\_before\_instruction[31:26] == 6'b000000 && previous\_instruction[20:16] == three\_before\_instruction[15:11]) | // Arithmetic instruction

(three\_before\_instruction[31:26] == 6'b111110 && previous\_instruction[20:16] == three\_before\_instruction[15:11]); // Comparison instruction

// Forward MEM/WB.MM to ID/EX.R1

assign forward\_mem\_wb\_mm1 = (previous\_instruction[25:21] != 5'b00000) &&

(three\_before\_instruction[31:26] == 6'b100011 && previous\_instruction[25:21] == three\_before\_instruction[20:16]); // Load instruction

// Forward MEM/WB.MM to ID/EX.R2

assign forward\_mem\_wb\_mm2 = (previous\_instruction[20:16] != 5'b00000) &&

(three\_before\_instruction[31:26] == 6'b100011 && previous\_instruction[20:16] == three\_before\_instruction[20:16]); // Load instruction

endmodule

**jump\_control.v**

module jump\_control (

input [31:0] reg1,

input [31:0] reg2,

input [31:0] instruction,

output execute\_jump

);

assign execute\_jump = (instruction[31:26] == 6'b111111 && reg1[reg2]) | (instruction[31:26] == 6'b000010);

// Conditional or unconditional jump

endmodule

**my\_alu.v**

`timescale 1ns / 1ps

`define ADD 6'b100000 // Addition

`define SUB 6'b100010 // Subtraction

`define AND 6'b100100 // Logical AND

`define OR 6'b100101 // Logical OR

`define XOR 6'b100110 // Logical XOR

`define MOVZ 6'b001010 // Conditional Move

`define SLL 6'b000000 // Logical Shift Left

`define CMP 6'b111110 // Compare

module my\_alu(

input [31:0] InputA, // First operand (Input A)

input [31:0] InputB, // Second operand (Input B)

input [5:0] Operation, // ALU operation code

input [4:0] ShiftAmount, // Shift amount for SLL

output [31:0] Result // ALU computation result

);

// Intermediate results for each operation

wire [31:0] addition\_result = InputA + InputB; // Addition

wire [31:0] subtraction\_result = InputA - InputB; // Subtraction

wire [31:0] and\_result = InputA & InputB; // Logical AND

wire [31:0] or\_result = InputA | InputB; // Logical OR

wire [31:0] xor\_result = InputA ^ InputB; // Logical XOR

wire [31:0] movz\_result = InputA; // Conditional Move

wire [31:0] shift\_left\_result = InputB << ShiftAmount; // Logical Shift Left

// Comparison Result: Encodes multiple comparison outcomes

wire [31:0] comparison\_result = {

22'b0,

!(InputA <= InputB), // InputA > InputB unsigned

!($signed(InputA) <= $signed(InputB)), // InputA > InputB signed

!(InputA < InputB), // InputA >= InputB unsigned

!($signed(InputA) < $signed(InputB)), // InputA >= InputB signed

!(InputA == InputB), // InputA != InputB

InputA <= InputB, // InputA <= InputB unsigned

$signed(InputA) <= $signed(InputB), // InputA <= InputB signed

InputA < InputB, // InputA < InputB unsigned

$signed(InputA) < $signed(InputB), // InputA < InputB signed

InputA == InputB // InputA == InputB

};

// Select the final output based on the Operation code

assign Result =

({32{Operation == `ADD}} & addition\_result) |

({32{Operation == `SUB}} & subtraction\_result) |

({32{Operation == `AND}} & and\_result) |

({32{Operation == `OR}} & or\_result) |

({32{Operation == `XOR}} & xor\_result) |

({32{Operation == `MOVZ}} & movz\_result) |

({32{Operation == `SLL}} & shift\_left\_result) |

({32{Operation == `CMP}} & comparison\_result);

endmodule

**my\_regfile.v**

`timescale 1ns / 1ps

module register\_file (

input clock,

input write\_enable,

input [4:0] read\_address1,

input [4:0] read\_address2,

input [4:0] write\_address,

output [31:0] read\_data1,

output [31:0] read\_data2,

input [31:0] write\_data

);

reg [31:0] register[0:31];

integer idx;

initial begin

for(idx = 0; idx < 32; idx = idx + 1) begin

register[idx] <= 0;

end

end

always @(posedge clock) begin

if(write\_enable) begin

register[write\_address] <= write\_data;

end

end

assign read\_data1 = write\_address == read\_address1 ? write\_data : register[read\_address1];

assign read\_data2 = write\_address == read\_address2 ? write\_data : register[read\_address2];

endmodule

**sign\_extend.v**

`timescale 1ns / 1ps

module sign\_extend (

input [15:0] input\_data,

output [31:0] output\_data

);

assign output\_data = {{16{input\_data[15]}}, input\_data[15:0]};

endmodule

**五、**测试结果及实验分析

1．流水线处理器

1. 处理器仿真测试波形（整体）

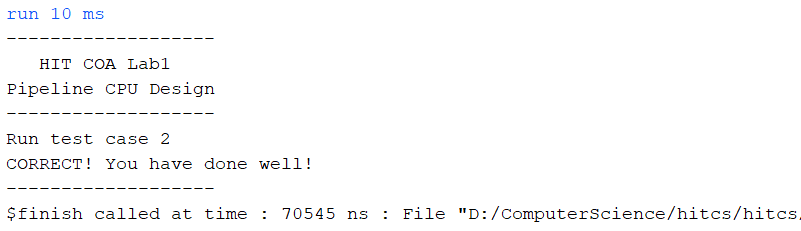
仿真测试结果：

文本, 信件

描述已自动生成

图形用户界面, 文本

描述已自动生成



仿真测试波形图：

图形用户界面

描述已自动生成

图形用户界面

中度可信度描述已自动生成

图形用户界面

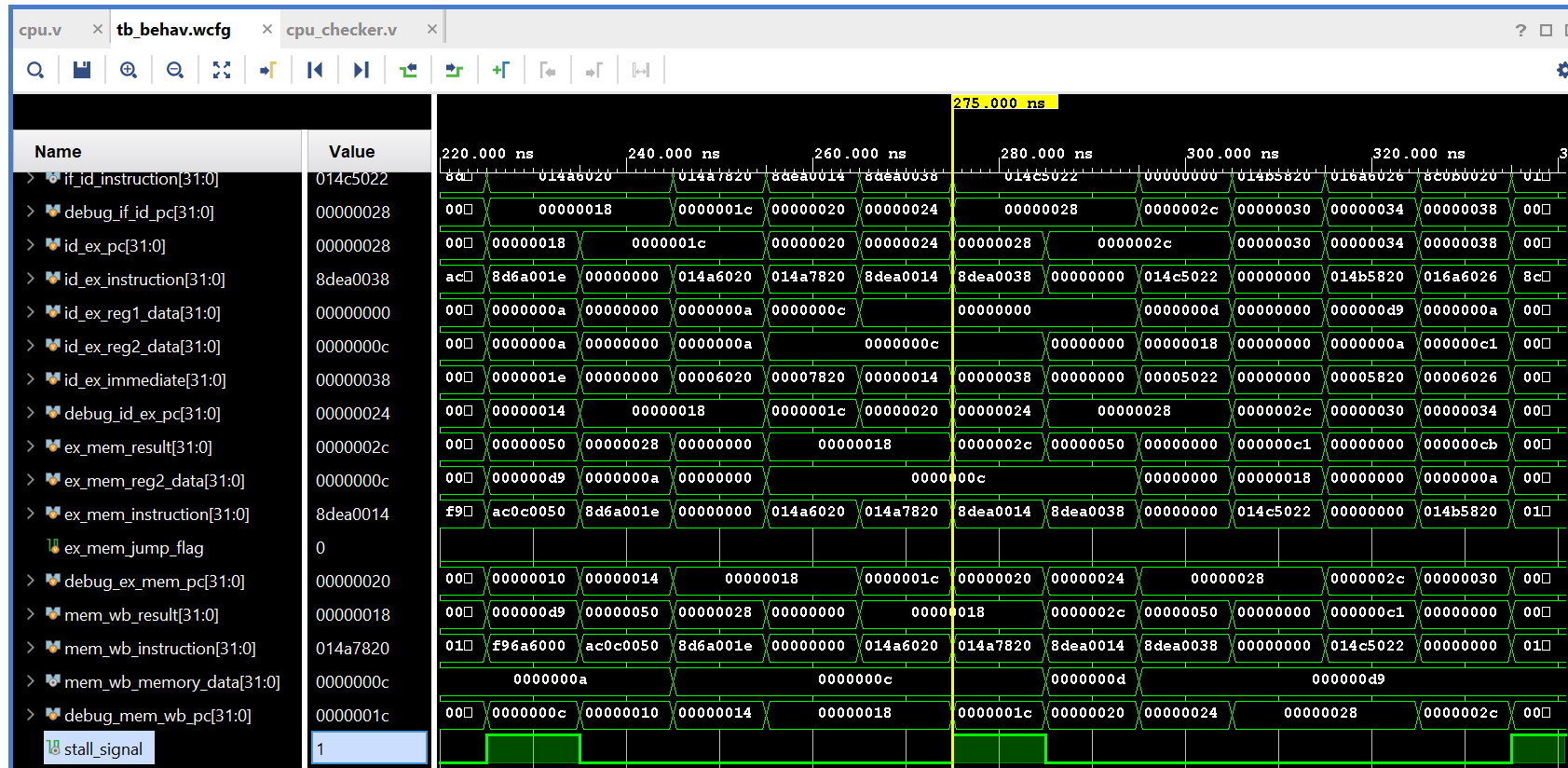
描述已自动生成

2．暂停和定向控制

1. 利用暂停解决数据冲突

图形用户界面

描述已自动生成



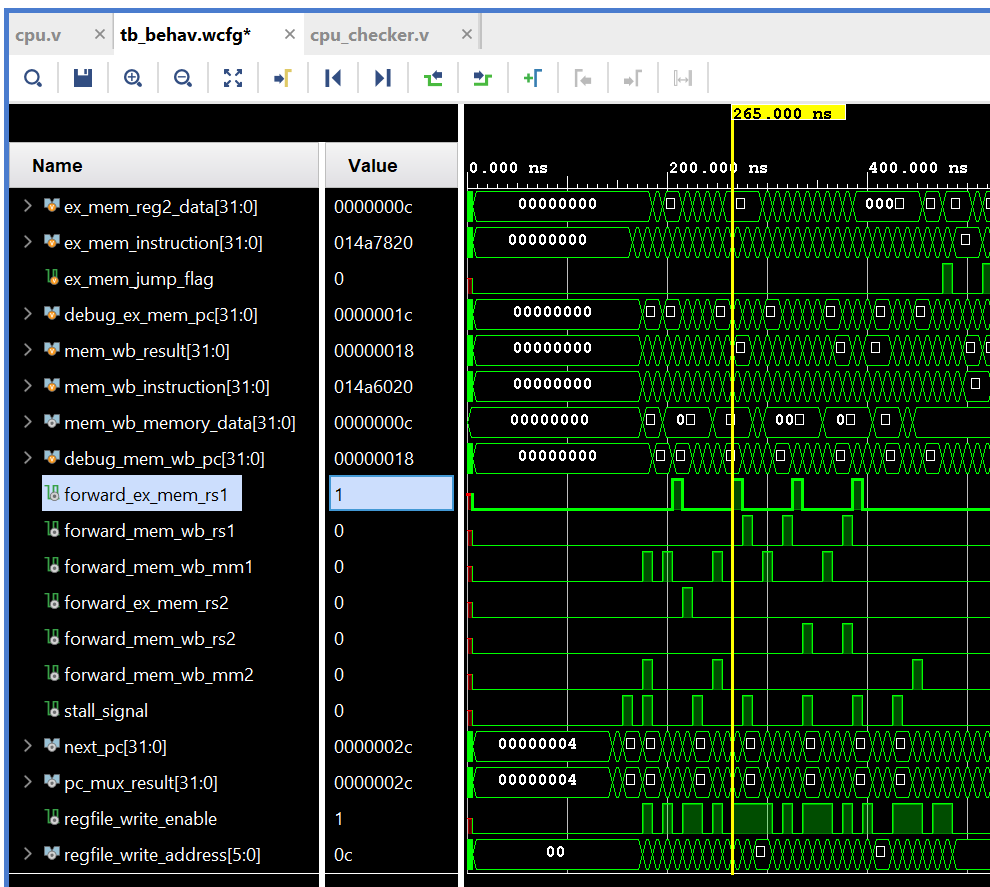
图形用户界面

描述已自动生成

从 275ns 左右开始，stall\_signal 变为高，导致 if\_id\_pc 和其他相关寄存器未更新。

当 stall\_signal 高时，regfile\_write\_enable 为低（0），表示暂停期间寄存器写操作被阻止。

1. 利用定向控制解决数据冲突



图形用户界面

描述已自动生成

在 265ns 附近：

forward\_ex\_mem\_rs1 = 1：说明数据从 MEM 阶段旁路到了 EX 阶段的操作数 rs1。

此时没有 stall\_signal 激活，表明数据冒险被成功解决，无需暂停。

在 265ns 观察到：

jump\_condition\_met 和 ex\_mem\_jump\_flag 的变化表明有条件跳转执行。

next\_pc 和 pc\_mux\_result 跳变，表明 PC 被更新为跳转目标地址，而不是顺序地址。

regfile\_write\_enable = 1 且 regfile\_write\_address 有有效值（如 0c）。

说明当前指令对寄存器文件进行了写回操作，可能是跳转指令更新链接寄存器或普通算术指令。

六、实验总结

通过本次实验，掌握了 Vivado 集成开发环境的基本使用，熟悉了 Verilog 语言的编码规范和 FPGA 编程调试方法。通过实现五段流水线的处理器，深入理解了流水线型处理器的结构及其性能优势，并通过暂停和定向技术解决数据冲突问题的实际操作，对数据相关性、控制相关性及其解决方案有了更加深刻的认识。实验还强化了逻辑设计能力与模块化设计思想，为后续更加复杂的处理器设计和硬件实现奠定了坚实基础。

**实验二 分支预测**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 预习成绩 | 实验成绩 | 报告成绩 | 总成绩 | 教师签字 |
|  |  |  |  |  |

一、实验目的

1. 掌握 Vivado 集成开发环境

2. 掌握 Verilog 语言

3. 掌握 FPGA 编程方法及硬件调试手段

4. 深刻理解动态分支预测的原理与方法

二、实验环境（实验设备、开发环境）

**Personal Computer：**

Processor：13th Gen Intel(R) Core(TM) 2.20 GHz

Installed RAM：16.0 GB (15.7 GB usable)

System type： 64-bit operating system, x64-based processor

**Xilinx FPGA**：xc7a200tfbg676-2

**Vivado Design Suite: version 2019.2**

**龙芯实验平台**

三、设计思想（实验预习）（如空白不够，可自行加页）

1. 为什么我们要为CPU设计分支预测器？你认为它起到了什么样的作用？

在流水线处理器中，分支指令（如条件跳转）会带来所谓的“控制冒险（Control Hazard）”或“分支冒险”，也就是当管线尚无法确定分支是否会发生跳转时，就可能出现等待或错误取指的问题，从而降低吞吐量。为了解决这一问题，需要对下一条指令的地址做出预测，也就是“分支预测”，这样可以让 CPU 在尚未完全确定分支结果时，就尽早取指和执行，从而减少因分支带来的气泡和停顿，提高管线的效率和整体性能。

分支预测器的主要作用有以下几点：

1. 减少流水线停顿：如果没有预测器，处理器通常需要等待分支指令在执行阶段确定跳转结果后，才能正确取指，这会引入多周期的空转或泡沫。
2. 提高取指带宽利用率：通过预测器为下一条指令提供可能的取地址，可以使取指单元继续工作，不至于闲置。
3. 提升处理器整体性能：在预测足够准确的前提下，CPU 能在发生错误分支预测之前持续进行有意义的工作，显著提高并行度。

2.解释你的分支预测器是如何工作的，请简要叙述你所使用的分支预测算法。

我的分支预测器使用了**BHT（Branch History Table）+ 2-bit saturating counter** 预测方法。它通过统计每条指令最近的跳转/不跳转行为来预测下一次的行为，并在分支执行完成后进行更新，使下一次预测更准确。

我的分支预测器的工作流程和算法大致可以归纳为以下几个部分：

**预测表（Predictor Table）**

使用了一个大小为 128 的预测表（predictor\_table[127:0]），其中每个表项存储一个“预测的目标地址”。当我们预测分支会被 taken 时，就直接跳转到这个存储的目标地址。对应地还有一个大小相同的 2 位饱和计数器数组（predictor\_counter[127:0]），每个表项用这 2 位计数器来记录分支的历史取向（taken 或 not taken 的倾向）。

**索引（Index）**

预测表和计数器的索引由指令所在 PC 的 [8:2] 位得到（即 id\_index 和 ex\_index），这样可将指令地址映射到对应的预测表项上。id\_index 用来在取指阶段（ID 阶段）获取预测结果，ex\_index 用来在执行阶段（EX 阶段）对该条分支进行更新。

**2 位饱和计数器（2-bit Saturating Counter）**

每个分支对应的 2 位计数器有 4 种状态：

00：强不跳转（Strongly Not Taken） 01：弱不跳转（Weakly Not Taken）

10：弱跳转（Weakly Taken） 11：强跳转（Strongly Taken）

当分支真正被执行并确定跳转时（branch\_execution\_success = 1），计数器会向“跳转”方向（递增）偏移；如果分支未跳转（branch\_execution\_success = 0），计数器会向“不跳转”方向（递减）偏移。这样可以记录最近几次分支的实际历史，用于之后的预测。

**预测判定**

在 ID 阶段，通过读取对应 id\_index 的饱和计数器值，若计数器值 ≥ 2（>= 2'b10），则认为分支是“预测跳转”；否则认为“不跳转”。若预测为跳转，则取 predictor\_table[id\_index] 作为下一条指令地址；若预测为不跳转，则默认 PC + 4。

**更新机制**

在 EX 阶段，当真正知道分支是否发生跳转以及跳转目标时，就更新：

目标地址表（predictor\_table[ex\_index]）：只有在分支实际发生跳转时才更新为真实的 branch\_target\_address。

2 位计数器：根据实际跳转与否进行递增或递减，保证计数器在 [00, 11] 范围内饱和变化。

3.假如CPU频繁执行某分支指令（bne），用“T”表示转移，“N”表示不转移，对于下面的三种转移轨迹序列，模拟你所使用的分支预测算法，分别计算它们的预测失败率（假设初始时刚刚进行了复位）

我所根据的初始 state = 01（弱不跳转, 预测=N）。

**(1)：TTTTTTTTTTTN**

| 第 i 条 | 实际(T/N) | 预测(根据 state) | state 变化前 | 预测是否正确 | state 变化后 |
| --- | --- | --- | --- | --- | --- |
| 1 | T | N (因 state=01) | 01 | **错误** | 01→10 (跳转递增) |
| 2 | T | T (因 state=10) | 10 | 正确 | 10→11 (跳转递增) |
| 3 | T | T (因 state=11) | 11 | 正确 | 11→11 (饱和) |
| 4 | T | T | 11 | 正确 | 11→11 |
| 5 | T | T | 11 | 正确 | 11→11 |
| 6 | T | T | 11 | 正确 | 11→11 |
| 7 | T | T | 11 | 正确 | 11→11 |
| 8 | T | T | 11 | 正确 | 11→11 |
| 9 | T | T | 11 | 正确 | 11→11 |
| 10 | T | T | 11 | 正确 | 11→11 |
| 11 | T | T | 11 | 正确 | 11→11 |
| 12 | N | T (因 state=11) | 11 | **错误** | 11→10 (不跳转递减) |

* **错误预测次数**：2 次（第 1 条和第 12 条）
* **总分支数**：12 条
* **预测失败率** = 2/12 ≈ 16.67%

**(2)：TTNTTNTTNTTN**

| 第 i 条 | 实际 | 预测(根据 state) | state 变化前 | 预测是否正确 | state 变化后 |
| --- | --- | --- | --- | --- | --- |
| 1 | T | N (state=01) | 01 | **错误** | 01→10 |
| 2 | T | T (state=10) | 10 | 正确 | 10→11 |
| 3 | N | T (state=11) | 11 | **错误** | 11→10 |
| 4 | T | T (state=10) | 10 | 正确 | 10→11 |
| 5 | T | T (state=11) | 11 | 正确 | 11→11 |
| 6 | N | T (state=11) | 11 | **错误** | 11→10 |
| 7 | T | T (state=10) | 10 | 正确 | 10→11 |
| 8 | T | T (state=11) | 11 | 正确 | 11→11 |
| 9 | N | T (state=11) | 11 | **错误** | 11→10 |
| 10 | T | T (state=10) | 10 | 正确 | 10→11 |
| 11 | T | T (state=11) | 11 | 正确 | 11→11 |
| 12 | N | T (state=11) | 11 | **错误** | 11→10 |

* **错误预测次数**：5 次（第 1, 3, 6, 9, 12 条）
* **总分支数**：12 条
* **预测失败率** = 5/12 ≈ 41.67%

**(3)：NNNTNNNNNNTT**

| 第 i 条 | 实际 | 预测(根据 state) | state 变化前 | 预测是否正确 | state 变化后 |
| --- | --- | --- | --- | --- | --- |
| 1 | N | N (state=01) | 01 | 正确 | 01→00 (不跳转递减) |
| 2 | N | N (state=00) | 00 | 正确 | 00→00 |
| 3 | N | N (state=00) | 00 | 正确 | 00→00 |
| 4 | T | N (state=00) | 00 | **错误** | 00→01 |
| 5 | N | N (state=01) | 01 | 正确 | 01→00 |
| 6 | N | N (state=00) | 00 | 正确 | 00→00 |
| 7 | N | N (state=00) | 00 | 正确 | 00→00 |
| 8 | N | N (state=00) | 00 | 正确 | 00→00 |
| 9 | N | N (state=00) | 00 | 正确 | 00→00 |
| 10 | N | N (state=00) | 00 | 正确 | 00→00 |
| 11 | T | N (state=00) | 00 | **错误** | 00→01 |
| 12 | T | N (state=01) | 01 | **错误** | 01→10 |

* **错误预测次数**：3 次（第 4, 11, 12 条）
* **总分支数**：12 条
* **预测失败率** = 3/12 = 25%

四、代码实现

写出带有详细注释的Verilog实现代码：

**cpu.v**

`timescale 1ns / 1ps

module cpu(

input clk, // Clock signal

input resetn, // Active low reset signal

output inst\_sram\_en, // Instruction SRAM enable

output[31:0] inst\_sram\_addr, // Instruction SRAM read address

input[31:0] inst\_sram\_rdata, // Instruction SRAM read data

output data\_sram\_en, // Data SRAM enable

output[3:0] data\_sram\_wen, // Data SRAM write enable

output[31:0] data\_sram\_addr, // Data SRAM address

output[31:0] data\_sram\_wdata, // Data SRAM write data

input[31:0] data\_sram\_rdata, // Data SRAM read data

// Debug signals for verification

output[31:0] debug\_wb\_pc, // Program counter of the current instruction

output debug\_wb\_rf\_wen, // Register file write enable

output[4:0] debug\_wb\_rf\_wnum, // Register file write number

output[31:0] debug\_wb\_rf\_wdata // Data to write back to the register file

);

// ========== IF\_ID =============

reg[31:0] ifid\_pc; // Current program counter value

wire[31:0] ifid\_instr; // Instruction fetched

wire[31:0] ifid\_next\_pc; // Address of the next instruction

wire[31:0] ifid\_jump\_addr; // Target address for jump instructions

// ========== ID\_EX =============

reg[31:0] idex\_next\_pc; // Address of the next instruction

reg[31:0] idex\_instr; // Current instruction being processed

reg[31:0] idex\_src1; // Source operand 1

reg[31:0] idex\_src2; // Source operand 2

reg[31:0] idex\_imm; // Immediate value in the instruction

reg[31:0] idex\_jump\_addr; // Target address for jump instructions

reg[31:0] idex\_pc; // Current program counter value

// ========== EX\_MEM ============

reg[31:0] exmem\_result; // Execution result

reg[31:0] exmem\_dest\_reg; // Destination register address

reg[31:0] exmem\_instr; // Current instruction content

reg[31:0] exmem\_pc; // Current program counter value

// ========== MEM\_WB ============

reg[31:0] memwb\_result; // Result from memory or ALU

reg[31:0] memwb\_instr; // Current instruction content

wire[31:0] memwb\_mem\_data; // Data read from memory

reg[31:0] memwb\_pc; // Current program counter value

// ========== Pipeline Control Signals ===========

wire pipeline\_stall; // Stall signal for pipeline

wire forward\_exmem\_src1; // Forward EX\_MEM result to source 1

wire forward\_memwb\_mem\_src1; // Forward MEM\_WB memory data to source 1

wire forward\_memwb\_result\_src1; // Forward MEM\_WB result to source 1

wire forward\_exmem\_src2; // Forward EX\_MEM result to source 2

wire forward\_memwb\_mem\_src2; // Forward MEM\_WB memory data to source 2

wire forward\_memwb\_result\_src2; // Forward MEM\_WB result to source 2

wire jump\_check; // Indicates a jump condition

wire[31:0] jump\_target\_addr; // Target address for the jump

wire is\_jump\_instr; // Whether the current instruction is a jump

wire execute\_jump; // Whether to execute the jump

// =================================================

initial begin

ifid\_pc = -4;

end

// ============ IF ==============

wire[31:0] next\_pc;

assign inst\_sram\_en = !pipeline\_stall && resetn;

assign inst\_sram\_addr = next\_pc;

my\_branch\_predictor my\_branch\_predictor\_inst (

.clock(clk),

.reset(rst),

.branch\_condition\_detect(jump\_check),

.branch\_instruction(is\_jump\_instr),

.branch\_execution\_success(execute\_jump),

.branch\_target\_address(jump\_target\_addr),

.instruction\_id\_stage(ifid\_instr),

.instruction\_ex\_stage(idex\_instr),

.pc\_id\_stage(ifid\_pc),

.pc\_ex\_stage(idex\_pc),

.next\_pc(next\_pc)

);

assign ifid\_instr = {32{resetn}} & inst\_sram\_rdata;

assign ifid\_jump\_addr = {32{resetn}} & next\_pc;

assign ifid\_next\_pc = {32{resetn}} & (ifid\_pc + 4);

always @(posedge clk) begin

if (!pipeline\_stall) begin

ifid\_pc <= resetn ? next\_pc : -4;

end

end

// ============ ID ==============

detector detector\_inst (

.current\_instruction(ifid\_instr),

.previous\_instruction(idex\_instr),

.two\_before\_instruction(exmem\_instr),

.three\_before\_instruction(memwb\_instr),

.stall\_signal(pipeline\_stall),

.forward\_ex\_mem\_rs1(forward\_exmem\_src1),

.forward\_mem\_wb\_rs1(forward\_memwb\_result\_src1),

.forward\_mem\_wb\_mm1(forward\_memwb\_mem\_src1),

.forward\_ex\_mem\_rs2(forward\_exmem\_src2),

.forward\_mem\_wb\_rs2(forward\_memwb\_result\_src2),

.forward\_mem\_wb\_mm2(forward\_memwb\_mem\_src2)

);

wire regfile\_we; // Register file write enable

wire[ 5:0] regfile\_waddr; // Register file write address

wire[31:0] regfile\_wdata; // Register file write data

wire[31:0] regfile\_rdata1;

wire[31:0] regfile\_rdata2;

register\_file regfile\_inst (

.clock(clk),

.write\_enable(regfile\_we),

.read\_address1(ifid\_instr[25:21]),

.read\_address2(ifid\_instr[20:16]),

.write\_address(regfile\_waddr),

.write\_data(regfile\_wdata),

.read\_data1(regfile\_rdata1),

.read\_data2(regfile\_rdata2)

);

wire[31:0] extended\_imm;

sign\_extend sign\_extend\_inst (

.input\_data(ifid\_instr[15:0]), // 16-bit immediate value

.output\_data(extended\_imm)

);

always @(posedge clk) begin

idex\_next\_pc <= {32{resetn}} & ifid\_next\_pc;

idex\_pc <= {32{resetn}} & ifid\_pc;

if (pipeline\_stall) begin

idex\_instr <= 0;

idex\_src1 <= 0;

idex\_src2 <= 0;

idex\_jump\_addr <= 0;

idex\_imm <= 0;

end else begin

idex\_src1 <= {32{resetn & !jump\_check}} & regfile\_rdata1;

idex\_src2 <= {32{resetn & !jump\_check}} & regfile\_rdata2;

idex\_instr <= {32{resetn & !jump\_check}} & ifid\_instr;

idex\_jump\_addr <= {32{resetn & !jump\_check}} & ifid\_jump\_addr;

idex\_imm <= {32{resetn & !jump\_check}} & extended\_imm;

end

end

// ===============================

// ============ EX ==============

wire[31:0] alu\_input\_a, forward\_src1;

wire[31:0] alu\_input\_b, forward\_src2;

assign forward\_src1 =

forward\_exmem\_src1 ? exmem\_result :

forward\_memwb\_result\_src1 ? memwb\_result :

forward\_memwb\_mem\_src1 ? memwb\_mem\_data :

idex\_src1;

assign forward\_src2 =

forward\_exmem\_src2 ? exmem\_result :

forward\_memwb\_result\_src2 ? memwb\_result :

forward\_memwb\_mem\_src2 ? memwb\_mem\_data :

idex\_src2;

wire mux\_select\_a, mux\_select\_b;

data\_mux ex\_mux\_a (

.input0(idex\_next\_pc),

.input1(forward\_src1),

.select\_line(mux\_select\_a),

.output\_data(alu\_input\_a)

); // Select between PC and source 1

assign mux\_select\_a =

(idex\_instr[31:26] == 6'b000000) | // Arithmetic instruction uses source 1

(idex\_instr[31:26] == 6'b101011) | // Store instruction uses source 1

(idex\_instr[31:26] == 6'b100011) | // Load instruction uses source 1

(idex\_instr[31:26] == 6'b111110);

data\_mux ex\_mux\_b (

.input0(idex\_imm),

.input1(forward\_src2),

.select\_line(mux\_select\_b),

.output\_data(alu\_input\_b)

); // Select between source 2 and immediate

assign mux\_select\_b =

(idex\_instr[31:26] == 6'b000000) | // Arithmetic instruction uses source 2

(idex\_instr[31:26] == 6'b111110); // Compare instruction uses source 2

wire[31:0] alu\_result;

wire[ 5:0] alu\_op =

({6{idex\_instr[31:26] == 6'b000000}} & idex\_instr[5:0]) | // Arithmetic opcode from last 6 bits

({6{idex\_instr[31:26] == 6'b111110}} & 6'b111110) | // Compare instruction

({6{idex\_instr[31:26] == 6'b101011}} & 6'b100000) | // Store instruction

({6{idex\_instr[31:26] == 6'b100011}} & 6'b100000); // Load instruction

my\_alu alu\_inst (

.InputA(alu\_input\_a),

.InputB(alu\_input\_b),

.Operation(alu\_op),

.ShiftAmount(idex\_instr[10:6]),

.Result(alu\_result)

);

jump\_control jump\_control\_inst (

.reg1(forward\_src1),

.reg2(forward\_src2),

.instruction(idex\_instr),

.is\_jump\_condition(is\_jump\_instr),

.execute\_jump(execute\_jump)

);

assign jump\_target\_addr =

execute\_jump ? (

({32{idex\_instr[31:26] == 6'b000010}} & {idex\_next\_pc[31:28], idex\_instr[25:0], 2'b00}) |

({32{idex\_instr[31:26] == 6'b111111}} & ((idex\_imm << 2) + idex\_next\_pc))

) : idex\_next\_pc;

assign jump\_check = is\_jump\_instr && (jump\_target\_addr != idex\_jump\_addr);

always @(posedge clk) begin

exmem\_dest\_reg <= {32{resetn}} & forward\_src2;

exmem\_instr <= {32{resetn}} & (

{32{!(idex\_instr[31:26] == 6'b000000 && idex\_instr[5:0] == 6'b001010 && forward\_src2 != 0)}}

) & idex\_instr;

exmem\_result <= {32{resetn}} & (

({32{idex\_instr[31:26] == 6'b000000}} & alu\_result) | // Arithmetic instruction result

({32{idex\_instr[31:26] == 6'b100011}} & alu\_result) | // Load instruction result

({32{idex\_instr[31:26] == 6'b101011}} & alu\_result) | // Store instruction result

({32{idex\_instr[31:26] == 6'b111110}} & alu\_result) // Compare instruction result

);

exmem\_pc <= {32{resetn}} & idex\_pc;

end

// ============ MEM ==============

assign data\_sram\_addr = exmem\_result; // Address for data memory

assign data\_sram\_wdata = exmem\_dest\_reg; // Data to write to memory

assign data\_sram\_wen = exmem\_instr[31:26] == 6'b101011; // Write enable for store instruction

assign data\_sram\_en =

(exmem\_instr[31:26] == 6'b100011) | // Enable for load instruction

(exmem\_instr[31:26] == 6'b101011); // Enable for store instruction

assign memwb\_mem\_data = {32{resetn}} & data\_sram\_rdata;

always @(posedge clk) begin

memwb\_instr <= {32{resetn}} & exmem\_instr;

memwb\_result <= {32{resetn}} & exmem\_result;

memwb\_pc <= {32{resetn}} & exmem\_pc;

end

// ============ WB ==============

wire mux\_select\_wb;

data\_mux wb\_mux (

.input0(memwb\_result), // Result from ALU

.input1(memwb\_mem\_data), // Data read from memory

.select\_line(mux\_select\_wb),

.output\_data(regfile\_wdata)

);

assign regfile\_waddr =

({32{memwb\_instr[31:26] == 6'b100011}} & memwb\_instr[20:16]) | // Write to rt for load

({32{memwb\_instr[31:26] == 6'b000000}} & memwb\_instr[15:11]) | // Write to rd for R-type

({32{memwb\_instr[31:26] == 6'b111110}} & memwb\_instr[15:11]); // Write to rd for compare

assign mux\_select\_wb =

(memwb\_instr[31:26] == 6'b100011); // Load writes memory data

assign regfile\_we =

((memwb\_instr[31:26] == 6'b000000) | // R-type write enable

(memwb\_instr[31:26] == 6'b100011) | // Load instruction write enable

(memwb\_instr[31:26] == 6'b111110)) & // Compare instruction write enable

(regfile\_waddr != 0); // Do not write to r0

// Debug signals

assign debug\_wb\_pc = memwb\_pc; // PC of the instruction being written back

assign debug\_wb\_rf\_wen = regfile\_we; // Write enable for register file

assign debug\_wb\_rf\_wnum = regfile\_waddr; // Register file write number

assign debug\_wb\_rf\_wdata = regfile\_wdata; // Data to be written back

endmodule

**branch\_predictor.v**

`timescale 1ns / 1ps

module my\_branch\_predictor (

input clock,

input reset,

input branch\_condition\_detect, // Detect branch condition

input branch\_instruction, // Indicates current instruction is a branch instruction

input branch\_execution\_success, // Indicates whether the branch was successfully executed

input[31:0] branch\_target\_address, // Branch target address

input[31:0] instruction\_id\_stage, // Instruction in ID stage

input[31:0] instruction\_ex\_stage, // Instruction in EX stage

input[31:0] pc\_id\_stage, // PC in ID stage

input[31:0] pc\_ex\_stage, // PC in EX stage

output[31:0] next\_pc // Next PC

);

// Predictor table: 128 entries

reg [31:0] predictor\_table[127:0]; // Stores predicted next PC

reg [1:0] predictor\_counter[127:0]; // 2-bit saturating counter

integer index;

// Predictor initialization

initial begin

for (index = 0; index < 128; index = index + 1) begin

predictor\_table[index] = 32'b0; // Initialize predicted PC to 0

predictor\_counter[index] = 2'b01; // Initialize counter to "weak not taken"

end

end

// Index for accessing predictor table

wire [6:0] id\_index = pc\_id\_stage[8:2]; // Use bits [8:2] as index

wire [6:0] ex\_index = pc\_ex\_stage[8:2]; // Use bits [8:2] as index

// Determine if branch is predicted taken or not

wire prediction\_taken = predictor\_counter[id\_index] >= 2'b10; // Strongly or weakly taken

wire [31:0] predicted\_target = predictor\_table[id\_index]; // Predicted target address

// Calculate next PC

assign next\_pc = branch\_condition\_detect ? branch\_target\_address : (prediction\_taken ? predicted\_target : pc\_id\_stage + 4);

always @(posedge clock or negedge reset) begin

if (!reset) begin

// Reset predictor table

for (index = 0; index < 128; index = index + 1) begin

predictor\_table[index] <= 32'b0;

predictor\_counter[index] <= 2'b01;

end

end else if (branch\_instruction) begin

// Update predictor table on branch execution

if (branch\_execution\_success) begin

// Branch taken: update target and increment counter

predictor\_table[ex\_index] <= branch\_target\_address;

case (predictor\_counter[ex\_index])

2'b00: predictor\_counter[ex\_index] <= 2'b01;

2'b01: predictor\_counter[ex\_index] <= 2'b10;

2'b10: predictor\_counter[ex\_index] <= 2'b11;

2'b11: predictor\_counter[ex\_index] <= 2'b11;

endcase

end else begin

// Branch not taken: decrement counter

case (predictor\_counter[ex\_index])

2'b00: predictor\_counter[ex\_index] <= 2'b00;

2'b01: predictor\_counter[ex\_index] <= 2'b00;

2'b10: predictor\_counter[ex\_index] <= 2'b01;

2'b11: predictor\_counter[ex\_index] <= 2'b10;

endcase

end

end

end

endmodule

**data\_mux.v**

`timescale 1ns / 1ps

module data\_mux(

input [31:0] input0,

input [31:0] input1,

input select\_line,

output [31:0] output\_data

);

assign output\_data = select\_line ? input1 : input0;

endmodule

**detector.v**

module detector (

input [31:0] current\_instruction, // Current instruction, IF/ID

input [31:0] previous\_instruction, // Previous instruction, ID/EX

input [31:0] two\_before\_instruction, // Two instructions before, EX/MEM

input [31:0] three\_before\_instruction, // Three instructions before, MEM/WB

output stall\_signal,

output forward\_ex\_mem\_rs1,

output forward\_mem\_wb\_rs1,

output forward\_mem\_wb\_mm1,

output forward\_ex\_mem\_rs2,

output forward\_mem\_wb\_rs2,

output forward\_mem\_wb\_mm2

);

// Stall Signal

assign stall\_signal =

(previous\_instruction[31:26] == 6'b100011 && (current\_instruction[31:26] == 6'b000000 || current\_instruction[31:26] == 6'b111110) && previous\_instruction[20:16] == current\_instruction[25:21]) |

(previous\_instruction[31:26] == 6'b100011 && (current\_instruction[31:26] == 6'b000000 || current\_instruction[31:26] == 6'b111110) && previous\_instruction[20:16] == current\_instruction[20:16]) |

(previous\_instruction[31:26] == 6'b100011 && previous\_instruction[20:16] == current\_instruction[25:21]);

// Forward EX/MEM.RS to ID/EX.R1

assign forward\_ex\_mem\_rs1 = (previous\_instruction[25:21] != 5'b00000) &&

(two\_before\_instruction[31:26] == 6'b000000 && previous\_instruction[25:21] == two\_before\_instruction[15:11]) | // Arithmetic instruction

(two\_before\_instruction[31:26] == 6'b111110 && previous\_instruction[25:21] == two\_before\_instruction[15:11]); // Comparison instruction

// Forward EX/MEM.RS to ID/EX.R2

assign forward\_ex\_mem\_rs2 = (previous\_instruction[20:16] != 5'b00000) &&

(two\_before\_instruction[31:26] == 6'b000000 && previous\_instruction[20:16] == two\_before\_instruction[15:11]) | // Arithmetic instruction

(two\_before\_instruction[31:26] == 6'b111110 && previous\_instruction[20:16] == two\_before\_instruction[15:11]); // Comparison instruction

// Forward MEM/WB.RS to ID/EX.R1

assign forward\_mem\_wb\_rs1 = (previous\_instruction[25:21] != 5'b00000) &&

(three\_before\_instruction[31:26] == 6'b000000 && previous\_instruction[25:21] == three\_before\_instruction[15:11]) | // Arithmetic instruction

(three\_before\_instruction[31:26] == 6'b111110 && previous\_instruction[25:21] == three\_before\_instruction[15:11]); // Comparison instruction

// Forward MEM/WB.RS to ID/EX.R2

assign forward\_mem\_wb\_rs2 = (previous\_instruction[20:16] != 5'b00000) &&

(three\_before\_instruction[31:26] == 6'b000000 && previous\_instruction[20:16] == three\_before\_instruction[15:11]) | // Arithmetic instruction

(three\_before\_instruction[31:26] == 6'b111110 && previous\_instruction[20:16] == three\_before\_instruction[15:11]); // Comparison instruction

// Forward MEM/WB.MM to ID/EX.R1

assign forward\_mem\_wb\_mm1 = (previous\_instruction[25:21] != 5'b00000) &&

(three\_before\_instruction[31:26] == 6'b100011 && previous\_instruction[25:21] == three\_before\_instruction[20:16]); // Load instruction

// Forward MEM/WB.MM to ID/EX.R2

assign forward\_mem\_wb\_mm2 = (previous\_instruction[20:16] != 5'b00000) &&

(three\_before\_instruction[31:26] == 6'b100011 && previous\_instruction[20:16] == three\_before\_instruction[20:16]); // Load instruction

endmodule

**jump\_control.v**

`timescale 1ns / 1ps

module jump\_control (

input [31:0] reg1,

input [31:0] reg2,

input [31:0] instruction,

output is\_jump\_condition,

output execute\_jump

);

assign is\_jump\_condition = (instruction[31:26] == 6'b111111) || (instruction[31:26] == 6'b000010);

assign execute\_jump = (instruction[31:26] == 6'b111111 && reg1[instruction[20:16]]) || (instruction[31:26] == 6'b000010);

endmodule

**my\_alu.v**

`timescale 1ns / 1ps

`define ADD 6'b100000 // Addition

`define SUB 6'b100010 // Subtraction

`define AND 6'b100100 // Logical AND

`define OR 6'b100101 // Logical OR

`define XOR 6'b100110 // Logical XOR

`define MOVZ 6'b001010 // Conditional Move

`define SLL 6'b000000 // Logical Shift Left

`define CMP 6'b111110 // Compare

module my\_alu(

input [31:0] InputA, // First operand (Input A)

input [31:0] InputB, // Second operand (Input B)

input [5:0] Operation, // ALU operation code

input [4:0] ShiftAmount, // Shift amount for SLL

output [31:0] Result // ALU computation result

);

// Intermediate results for each operation

wire [31:0] addition\_result = InputA + InputB; // Addition

wire [31:0] subtraction\_result = InputA - InputB; // Subtraction

wire [31:0] and\_result = InputA & InputB; // Logical AND

wire [31:0] or\_result = InputA | InputB; // Logical OR

wire [31:0] xor\_result = InputA ^ InputB; // Logical XOR

wire [31:0] movz\_result = InputA; // Conditional Move

wire [31:0] shift\_left\_result = InputB << ShiftAmount; // Logical Shift Left

// Comparison Result: Encodes multiple comparison outcomes

wire [31:0] comparison\_result = {

22'b0,

!(InputA <= InputB), // InputA > InputB unsigned

!($signed(InputA) <= $signed(InputB)), // InputA > InputB signed

!(InputA < InputB), // InputA >= InputB unsigned

!($signed(InputA) < $signed(InputB)), // InputA >= InputB signed

!(InputA == InputB), // InputA != InputB

InputA <= InputB, // InputA <= InputB unsigned

$signed(InputA) <= $signed(InputB), // InputA <= InputB signed

InputA < InputB, // InputA < InputB unsigned

$signed(InputA) < $signed(InputB), // InputA < InputB signed

InputA == InputB // InputA == InputB

};

// Select the final output based on the Operation code

assign Result =

({32{Operation == `ADD}} & addition\_result) |

({32{Operation == `SUB}} & subtraction\_result) |

({32{Operation == `AND}} & and\_result) |

({32{Operation == `OR}} & or\_result) |

({32{Operation == `XOR}} & xor\_result) |

({32{Operation == `MOVZ}} & movz\_result) |

({32{Operation == `SLL}} & shift\_left\_result) |

({32{Operation == `CMP}} & comparison\_result);

endmodule

**my\_regfile.v**

`timescale 1ns / 1ps

module register\_file (

input clock,

input write\_enable,

input [4:0] read\_address1,

input [4:0] read\_address2,

input [4:0] write\_address,

output [31:0] read\_data1,

output [31:0] read\_data2,

input [31:0] write\_data

);

reg [31:0] register[0:31];

integer idx;

initial begin

for(idx = 0; idx < 32; idx = idx + 1) begin

register[idx] <= 0;

end

end

always @(posedge clock) begin

if(write\_enable) begin

register[write\_address] <= write\_data;

end

end

assign read\_data1 = write\_address == read\_address1 ? write\_data : register[read\_address1];

assign read\_data2 = write\_address == read\_address2 ? write\_data : register[read\_address2];

endmodule

**sign\_extend.v**

`timescale 1ns / 1ps

module sign\_extend (

input [15:0] input\_data,

output [31:0] output\_data

);

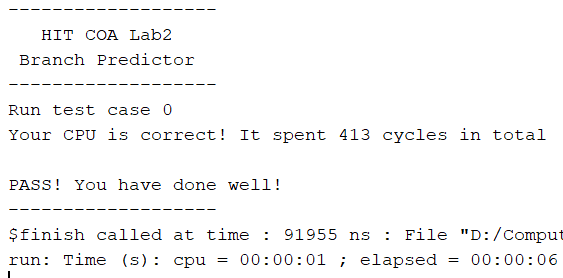
assign output\_data = {{16{input\_data[15]}}, input\_data[15:0]};

endmodule

**五、**测试结果及实验分析

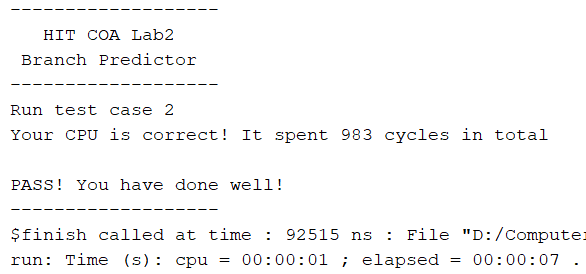
测试结果（包括你的预测失败率）：

仿真结果输出：

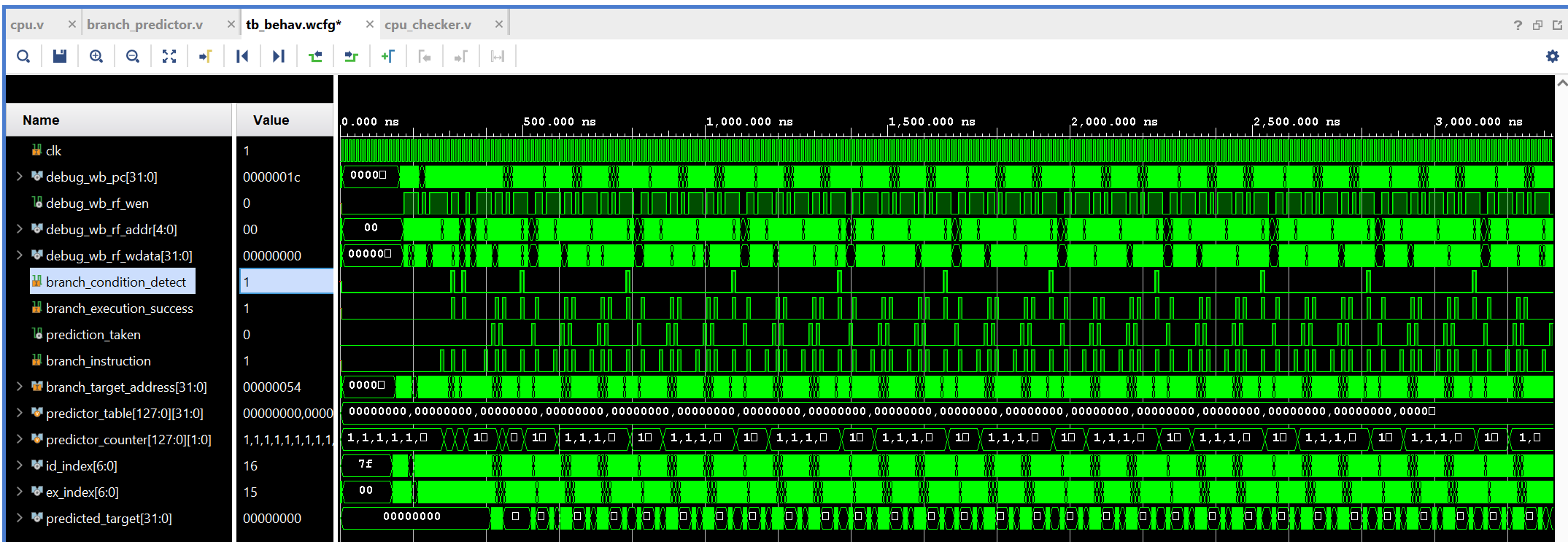


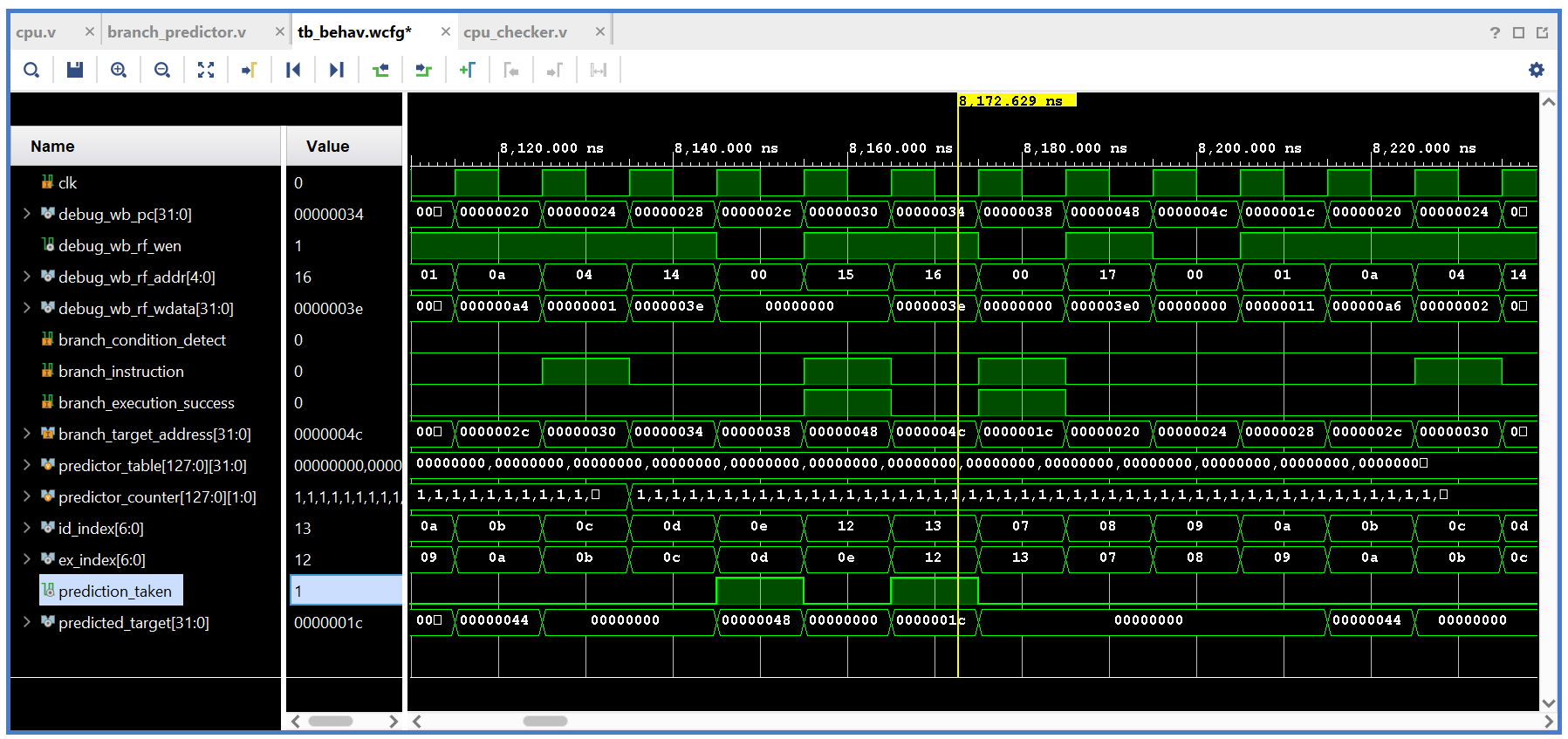
文本, 信件

描述已自动生成



仿真波形信号抓取：





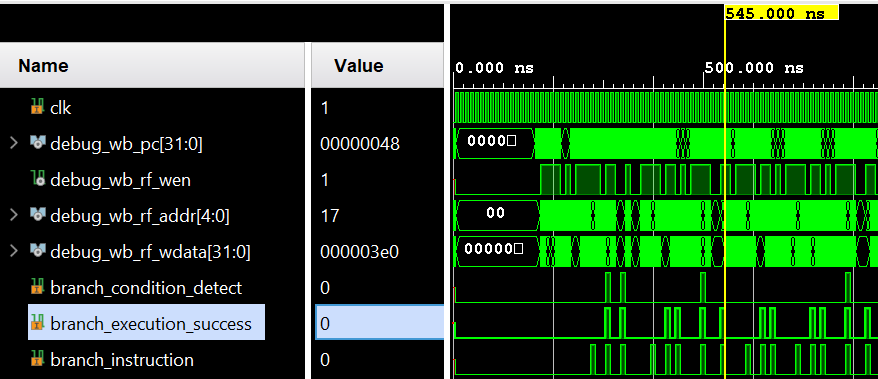
预测失败率的计算：

检查有多少个**branch\_instruction**,获得总的分支个数。然后再计算**branch\_execution\_success**: 表示分支预测是否成功（值为1时，预测成功）

根据波形图一个周期循环（从275ns到545ns）的结果，

图形用户界面

描述已自动生成



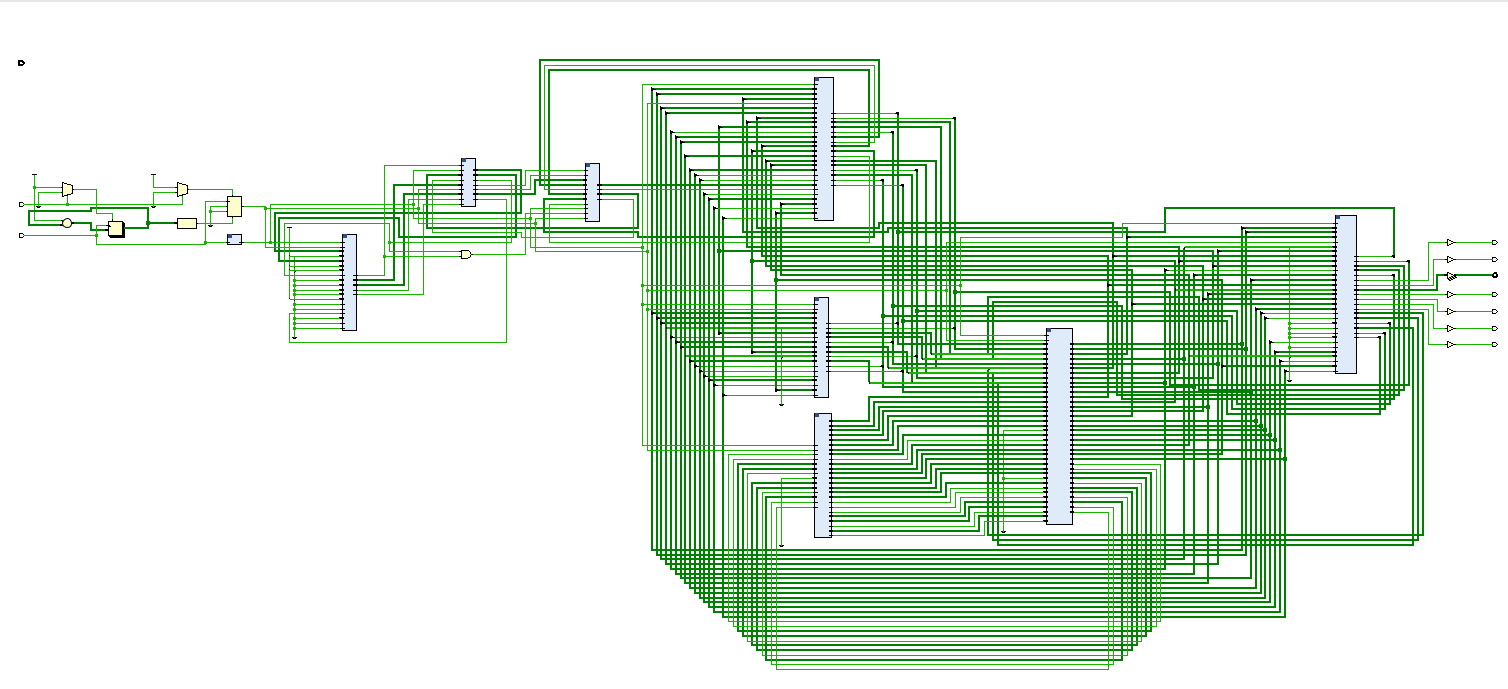
**branch\_instruction = 8**

**branch\_execution\_success = 6**

**分支预测失败率 = （8 - 6）/ 8 = 25%**

**实验结果分析：**

精细设计结果：



同步设计结果：

图示

中度可信度描述已自动生成

实现设计图：

电脑萤幕画面

中度可信度描述已自动生成

六、实验总结

通过本次实验，进一步掌握了 Vivado 集成开发环境和 Verilog 语言在设计复杂处理器架构中的应用。实验中实现了动态分支预测器，对流水线 CPU 的性能提升具有重要意义。通过设计与验证分支预测机制，深入理解了控制相关对流水线性能的影响以及分支预测的原理与实现方法。实验还增强了逻辑分析能力和硬件调试技能，为后续更高效的 CPU 设计和体系结构优化奠定了扎实基础。此外，实验使我们认识到不同预测技术对性能优化的效果差异，也为未来的分支预测研究提供了新的思路和方向。

**实验三 指令Cache的设计与实现**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 预习成绩 | 实验成绩 | 报告成绩 | 总成绩 | 教师签字 |
|  |  |  |  |  |

一、实验目的

1．掌握 Vivado 集成开发环境

2．掌握Verilog语言

3．掌握FPGA编程方法及硬件调试手段

4．深刻理解指令 Cache 的结构和整体工作原理

二、实验环境（实验设备、开发环境）

**Personal Computer：**

Processor：13th Gen Intel(R) Core(TM) 2.20 GHz

Installed RAM：16.0 GB (15.7 GB usable)

System type： 64-bit operating system, x64-based processor

**Xilinx FPGA**：xc7a200tfbg676-2

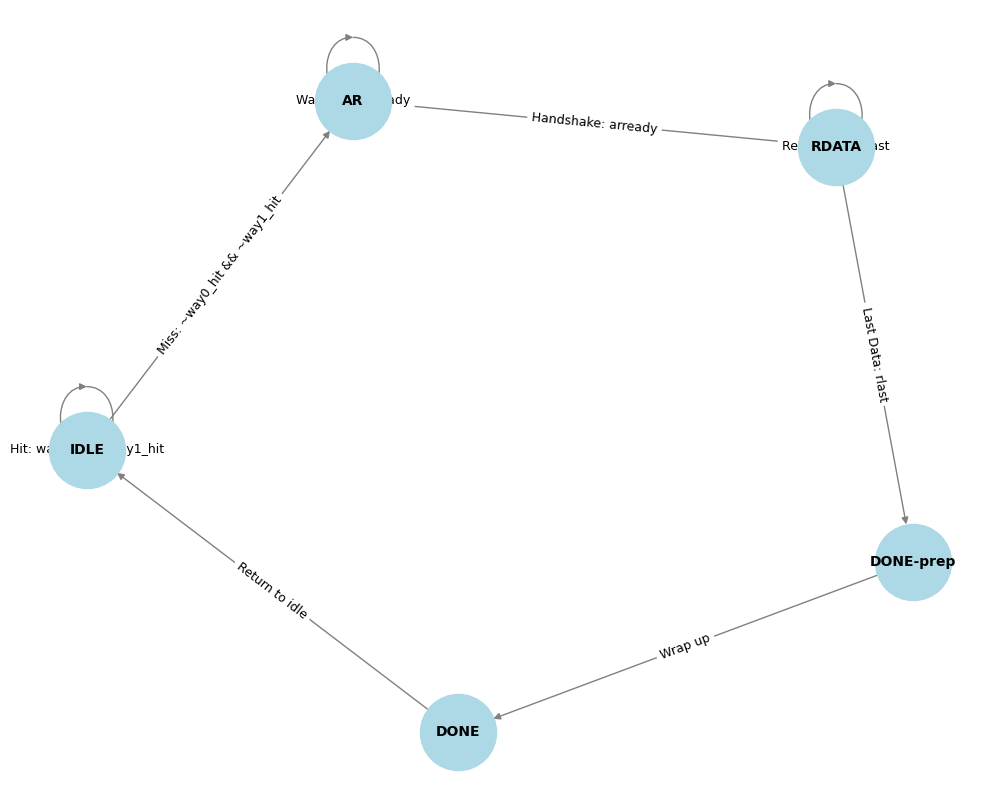
**Vivado Design Suite: version 2019.2**

**龙芯实验平台**

三、设计思想（实验预习）（如空白不够，可自行加页）

1．给出设计的指令Cache的状态转移自动机，解释各个状态，并简要说明Cache在hit和miss时都是如何进行工作的。

下面给出所使用的有限状态机（FSM）的解释。



1. 指令Cache的状态转移自动机

该设计的FSM使用一个3位的state\_reg来表示5个状态，如代码所示：

* 0：IDLE
* 1：AR
* 2：RDATA
* 3：DONE-prep
* 4：DONE

以下简要说明每个状态的意义和在Cache命中/未命中时的处理流程：

(1) IDLE (state\_reg = 0)

* 状态含义：空闲状态，等待CPU发起新的取值请求。
* 进入条件：复位后或完成一次完整的读事务后，系统回到IDLE。
* 状态行为：  
  a) 如果flag\_reg=1，说明该周期已成功捕获CPU地址并完成标志更新，需要检查是否命中。  
  b) Cache命中（way0\_hit || way1\_hit为真）：
  + 将命中way的数据直接返回给CPU (cache\_data\_ok <= 1, cache\_rdata <= wayX\_data\_out)。
  + 更新替换位：如果way1被命中，就把way0的replacement bit设为1，反之亦然。
  + 不需要发起AXI访问，状态机基本停留在IDLE或很快回到IDLE等待下一个请求。  
    c) Cache未命中（~way0\_hit && ~way1\_hit）：
  + 发起AXI读请求(arvalid <= 1)，araddr设置为所请求cache line的起始地址({request\_tag, request\_index,5'b00000})。
  + 选择要替换的way(way\_override)，更新相应way的有效位(valid=1)、tag，以及replacement位。
  + 状态跳转到AR (state\_reg <= 1)。

(2) AR (state\_reg = 1)

* 状态含义：等待AXI的arready握手。
* 进入条件：IDLE状态判断到cache未命中，需要向AXI发起读请求。
* 状态行为：  
  a) 当arready=1时，表示地址握手成功，arvalid <= 0，rready <= 1，用于接收数据。  
  b) 状态跳转到RDATA (state\_reg <= 2)。

(3) RDATA (state\_reg = 2)

* 状态含义：在此状态下从AXI分多拍(burst)接收一整行cache line的数据。
* 进入条件：AR状态中arready握手完成。
* 状态行为：  
  a) 如果rvalid=1，表示有数据到达。
  + 根据要替换的way(way\_override)，将rdata写入对应的BRAM (bram0\_write\_enable 或 bram1\_write\_enable = 1)。
  + 每次写入后地址加1，以存储整条cache line。
  + 实现“关键字优先返回”(critical word first)：如果当前写地址恰好是CPU请求的字，就立即将该字(cache\_rdata)返回给CPU并置cache\_data\_ok=1，加速CPU拿到所需数据。  
    b) 如果检测到rlast=1，表示接收完burst的最后一个数据：
  + rready <= 0结束接收；
  + 状态跳转到DONE-prep (state\_reg <= 3)。

(4) DONE-prep (state\_reg = 3)

* 状态含义：收尾操作，例如关掉写使能等。
* 进入条件：RDATA状态中接收到rlast=1。
* 状态行为：
  + 关闭对应的写使能 (bram0\_write\_enable或bram1\_write\_enable <= 0)。
  + 置transfer\_done\_reg <= 1，表示一次cache line的填充完成。
  + 跳转到DONE (state\_reg <= 4)。

(5) DONE (state\_reg = 4)

* 状态含义：若前面还没返回关键字，则在此状态补发cache\_data\_ok。
* 进入条件：DONE-prep状态后直接进入。
* 状态行为：
  + 如果critical\_word\_fetched=0，表示在接收数据时没有早期返回，此时需要在DONE状态将数据返回给CPU(cache\_data\_ok <= 1)。
  + transfer\_done\_reg <= 0后，状态回到IDLE (state\_reg <= 0)。

命中/未命中流程总结：

* hit：直接从对应way读数据并返回，状态机保持或回到IDLE；
* miss：转移到AR发AXI读请求，再到RDATA接收整条cache line，最后回到DONE和IDLE。

2．解释你设计的指令Cache是如何实现二/四路组相联的，请简要说明实现的算法。  
代码中使用了两个BRAM (ram0, ram1)，分别对应way0和way1；并且有tags\_way0和tags\_way1两个tag数组，tag字段中包含tag bits、valid位、replacement位。

· tag bits: 用于地址高位匹配。

· valid位: 表示该entry有效无效。

· replacement位: 用于标记LRU或类似替换策略（在2路场景可用1bit表示谁是替换候选）。  
通过wire way0\_hit和way1\_hit并行比较两个way，如果任意匹配则命中。若都不匹配，则执行替换(way\_override根据replacement位或valid位决定替换谁)，从AXI中取数据写入替换的way。

本设计通过：  
(1) 5态的FSM完成命中/未命中、AXI突发读等流程；  
(2) 两份BRAM (way0/way1) + 两份tag数组 + 1bit replacement位，实现了简单的2路组相联；

四、代码实现

给出实现 Cache 的Verilog代码：

module cache (

input clk , // Clock signal

input resetn , // Active-low reset signal

// Sram-like interface between CPU and Cache

input cpu\_req , // CPU request signal

input [31:0] cpu\_addr , // CPU address signal

output reg [31:0] cache\_rdata , // Data read from Cache

output cache\_addr\_ok, // Cache address handshake signal

output reg cache\_data\_ok, // Cache data handshake signal

// AXI interface between Cache and memory

output reg [3 :0] arid , // AXI request ID

output reg [31:0] araddr , // AXI address signal

output reg arvalid, // AXI address valid signal

input arready, // AXI address ready signal

input [3 :0] rid , // AXI response ID

input [31:0] rdata , // Data from AXI

input rlast , // Last data signal for AXI burst

input rvalid , // Valid data signal from AXI

output reg rready // Ready signal for AXI data

);

//===========================================================

// Internal wires and registers

//===========================================================

wire [31:0] way0\_data\_out, way1\_data\_out; // Data read from two cache ways

wire [9:0] bram0\_read\_addr; // BRAM0 read address

wire bram0\_read\_enable; // BRAM0 read enable

reg [19:0] way0\_tag\_reg, way1\_tag\_reg; // Tags for both ways

reg way0\_valid\_reg, way1\_valid\_reg;// Valid bits for both ways

reg way0\_repl\_reg, way1\_repl\_reg; // Replacement bits

reg [31:0] request\_addr\_reg; // Registered request address

reg flag\_reg, fetch\_flag\_reg;

reg transfer\_done\_reg; // Indicates completion of line fetch

reg [9:0] bram0\_write\_addr; // BRAM0 write address

reg [31:0] bram0\_write\_data; // BRAM0 write data

reg bram0\_write\_enable; // BRAM0 write enable

wire [9:0] bram1\_read\_addr; // BRAM1 read address

wire bram1\_read\_enable; // BRAM1 read enable

reg [9:0] bram1\_write\_addr; // BRAM1 write address

reg [31:0] bram1\_write\_data; // BRAM1 write data

reg bram1\_write\_enable; // BRAM1 write enable

// Each entry in tags\_wayX is 22 bits:

// [21] = replacement bit

// [20] = valid bit

// [19:0] = tag bits

reg [21:0] tags\_way0[127:0]; // Tag array for way0

reg [21:0] tags\_way1[127:0]; // Tag array for way1

reg [2:0] state\_reg; // Simple FSM: 0=IDLE, 1=AR, 2=RDATA, 3=DONE-prep, 4=DONE

// Replacement logic: decide which way to replace

wire way\_override = (!way1\_valid\_reg || way1\_repl\_reg);

// Address parsing

wire [19:0] request\_tag = request\_addr\_reg[31:12];

wire [ 6:0] request\_index = request\_addr\_reg[11:5];

wire [ 4:0] request\_offset = request\_addr\_reg[4:0];

wire [19:0] cpu\_tag = cpu\_addr[31:12];

wire [ 6:0] cpu\_index = cpu\_addr[11:5];

wire [ 4:0] cpu\_offset = cpu\_addr[4:0];

// Hit detection

wire way0\_hit = (way0\_valid\_reg && way0\_tag\_reg == request\_tag);

wire way1\_hit = (way1\_valid\_reg && way1\_tag\_reg == request\_tag);

// CPU stalling logic

wire cache\_stall;

// Flag used to indicate that we have returned the critical word to CPU

// (so we won't send cache\_data\_ok again for the same miss)

reg critical\_word\_fetched;

integer i;

initial begin

for(i = 0; i < 128; i = i + 1) begin

tags\_way0[i] = 22'b0;

tags\_way1[i] = 22'b0;

end

end

//===========================================================

// Main sequential logic

//===========================================================

always @(posedge clk) begin

if(~resetn) begin

// Reset internal registers

state\_reg <= 3'd0;

cache\_data\_ok <= 1'b0;

arvalid <= 1'b0;

rready <= 1'b0;

flag\_reg <= 1'b0;

fetch\_flag\_reg <= 1'b0;

transfer\_done\_reg <= 1'b0;

way0\_tag\_reg <= 20'd0;

way1\_tag\_reg <= 20'd0;

way0\_valid\_reg <= 1'b0;

way1\_valid\_reg <= 1'b0;

way0\_repl\_reg <= 1'b0;

way1\_repl\_reg <= 1'b0;

request\_addr\_reg <= 32'd0;

bram0\_write\_enable <= 1'b0;

bram1\_write\_enable <= 1'b0;

bram0\_write\_addr <= 10'd0;

bram1\_write\_addr <= 10'd0;

bram0\_write\_data <= 32'd0;

bram1\_write\_data <= 32'd0;

critical\_word\_fetched <= 1'b0;

end

else begin

// Default signals

cache\_data\_ok <= 1'b0;

bram0\_write\_enable <= 1'b0;

bram1\_write\_enable <= 1'b0;

// Latch current tag/valid bits if not stalled

if(~cache\_stall) begin

way0\_tag\_reg <= tags\_way0[cpu\_index][19:0];

way1\_tag\_reg <= tags\_way1[cpu\_index][19:0];

way0\_valid\_reg <= tags\_way0[cpu\_index][20];

way1\_valid\_reg <= tags\_way1[cpu\_index][20];

way0\_repl\_reg <= tags\_way0[cpu\_index][21];

way1\_repl\_reg <= tags\_way1[cpu\_index][21];

request\_addr\_reg <= cpu\_addr;

flag\_reg <= 1'b1;

end

// FSM

case (state\_reg)

3'd0: begin

// IDLE state

if(flag\_reg) begin

// If hit in either way

if(way0\_hit || way1\_hit) begin

// Provide data to CPU immediately

cache\_data\_ok <= 1'b1;

cache\_rdata <= way0\_hit ? way0\_data\_out : way1\_data\_out;

// Update replacement bits

tags\_way0[request\_index][21] <= way1\_hit; // If way1 was hit => set way0's bit

tags\_way1[request\_index][21] <= way0\_hit; // If way0 was hit => set way1's bit

end

else begin

// Miss => start AXI read request

arvalid <= 1'b1;

araddr <= { request\_tag, request\_index, 5'b00000 };

arid <= 4'd0;

// Reset critical\_word\_fetched for this new miss

critical\_word\_fetched <= 1'b0;

// Update replacement bits for the victim way

if(way\_override) begin

bram1\_write\_addr <= { request\_index, 3'b000 } - 1;

tags\_way1[request\_index] <= { 1'b0, 1'b1, request\_tag };

end

else begin

bram0\_write\_addr <= { request\_index, 3'b000 } - 1;

tags\_way0[request\_index] <= { 1'b0, 1'b1, request\_tag };

end

state\_reg <= 3'd1;

end

end

end

3'd1: begin

// AR state: wait for arready handshake

if(arready) begin

arvalid <= 1'b0;

rready <= 1'b1;

state\_reg <= 3'd2;

end

end

3'd2: begin

// RDATA state: receive AXI data burst

if(rvalid) begin

if(way\_override) begin

// Write to way1

bram1\_write\_enable <= 1'b1;

bram1\_write\_data <= rdata;

bram1\_write\_addr <= bram1\_write\_addr + 1'b1;

// Early return of the critical word if not yet fetched

if(!critical\_word\_fetched &&

(bram1\_write\_addr + 1'b1 == { request\_index, request\_offset[4:2]})) begin

cache\_rdata <= rdata;

cache\_data\_ok <= 1'b1;

critical\_word\_fetched <= 1'b1;

end

end

else begin

// Write to way0

bram0\_write\_enable <= 1'b1;

bram0\_write\_data <= rdata;

bram0\_write\_addr <= bram0\_write\_addr + 1'b1;

// Early return of the critical word if not yet fetched

if(!critical\_word\_fetched &&

(bram0\_write\_addr + 1'b1 == { request\_index, request\_offset[4:2]})) begin

cache\_rdata <= rdata;

cache\_data\_ok <= 1'b1;

critical\_word\_fetched <= 1'b1;

end

end

// If we have reached the last word of the burst

if(rlast) begin

rready <= 1'b0;

state\_reg <= 3'd3;

end

end

end

3'd3: begin

// Wrap up

bram0\_write\_enable <= 1'b0;

bram1\_write\_enable <= 1'b0;

transfer\_done\_reg <= 1'b1;

state\_reg <= 3'd4;

end

3'd4: begin

// Final step: if we never returned the critical word, do it now

if(!critical\_word\_fetched) begin

cache\_data\_ok <= 1'b1;

end

transfer\_done\_reg <= 1'b0;

state\_reg <= 3'd0;

end

endcase

end

end

//===========================================================

// BRAM (Block Memory) instantiations for both ways

//===========================================================

blk\_mem\_gen\_0 ram0(

.clka(clk),

.clkb(clk),

.addra(bram0\_write\_addr),

.addrb(bram0\_read\_addr),

.wea (bram0\_write\_enable),

.enb (bram0\_read\_enable),

.dina (bram0\_write\_data),

.doutb(way0\_data\_out)

);

blk\_mem\_gen\_0 ram1(

.clka(clk),

.clkb(clk),

.addra(bram1\_write\_addr),

.addrb(bram1\_read\_addr),

.wea (bram1\_write\_enable),

.enb (bram1\_read\_enable),

.dina (bram1\_write\_data),

.doutb(way1\_data\_out)

);

//===========================================================

// BRAM read enable and addresses

//===========================================================

assign bram0\_read\_enable = cpu\_req && resetn && ~cache\_stall;

assign bram1\_read\_enable = cpu\_req && resetn && ~cache\_stall;

assign bram0\_read\_addr = { cpu\_index, cpu\_offset[4:2] };

assign bram1\_read\_addr = { cpu\_index, cpu\_offset[4:2] };

//===========================================================

// CPU stall logic

//===========================================================

// If we have captured the CPU address (flag\_reg=1) but it's a miss

// and the line fetch is not yet finished, we stall the CPU

assign cache\_stall = flag\_reg && (~way0\_hit && ~way1\_hit && ~transfer\_done\_reg);

// Address handshake is valid if not stalled

assign cache\_addr\_ok = ~cache\_stall && resetn;

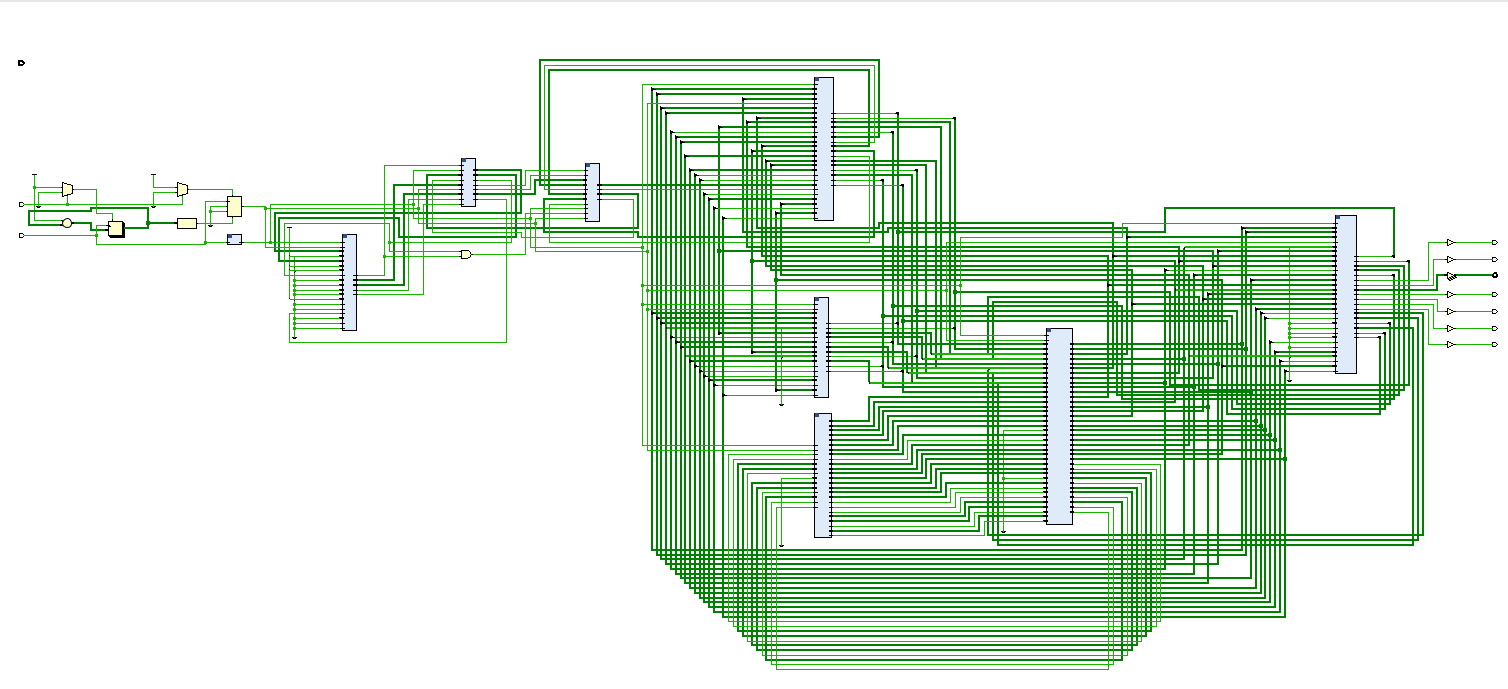
endmodule

**五、**测试结果及实验分析

测试结果：

|  |  |
| --- | --- |
| 自动化测试是否通过 | 不通过的Miss rate |
| 通过 | 0.43% |

精细设计结果：

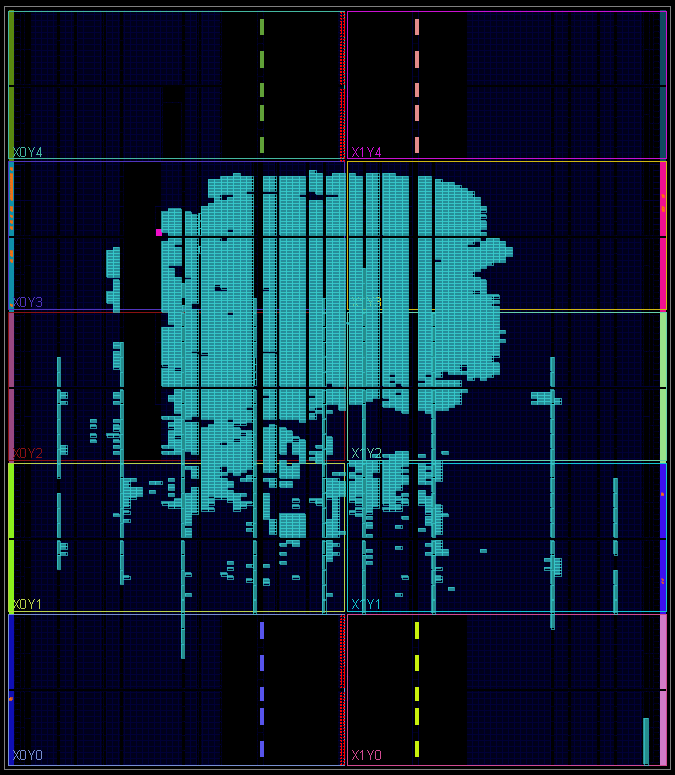


同步设计结果：

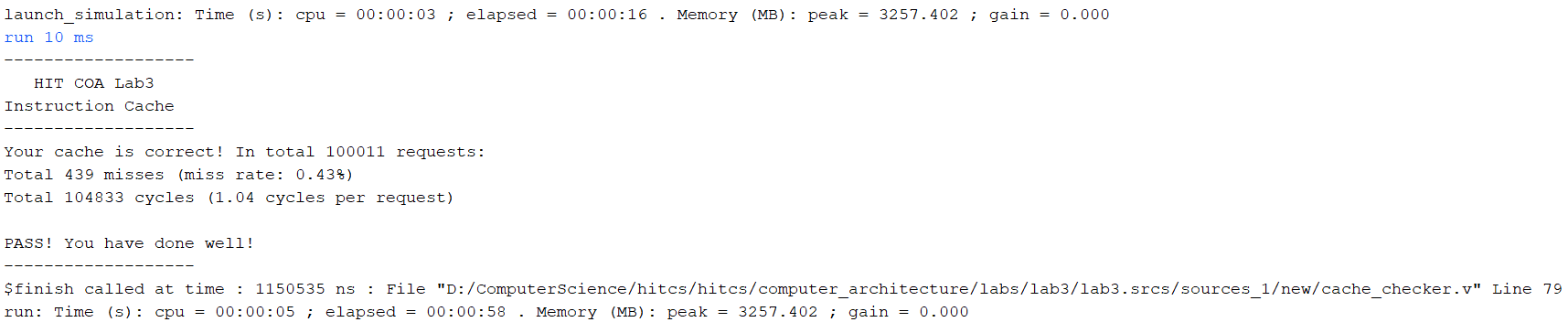
图示

描述已自动生成

实现设计图：



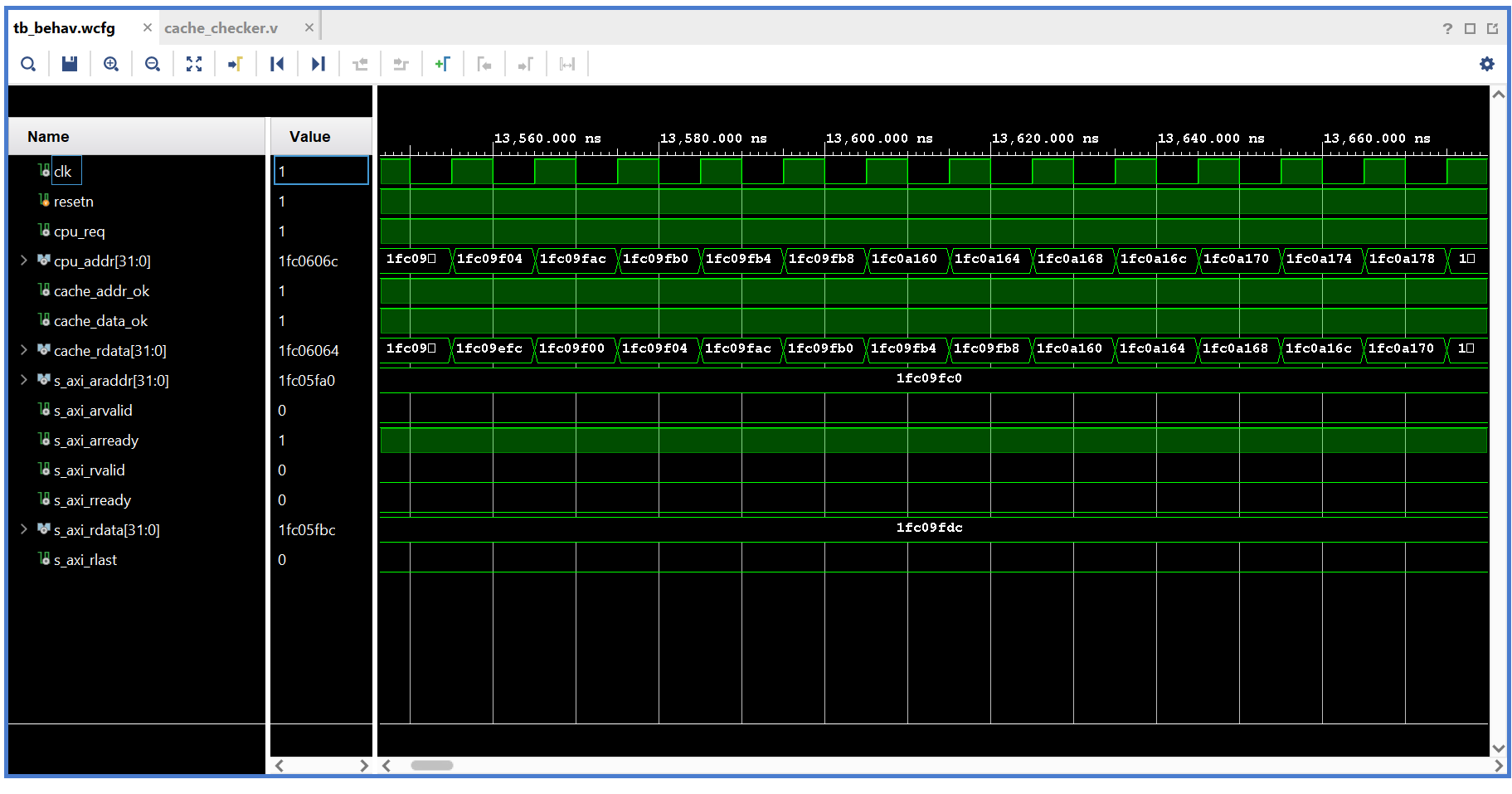
仿真结果输出：



仿真波形信号抓取：

图形用户界面

描述已自动生成



六、实验总结

本实验通过设计和实现一个基于有限状态机（FSM）的二路组相联Cache，展示了Cache在命中和未命中情况下的处理流程。通过详细的状态划分（包括IDLE、AR、RDATA、DONE等），实现了对CPU请求的高效响应，同时利用关键字优先返回（critical word first）优化了延迟。替换策略采用简单的LRU近似算法，确保了Cache的动态性和可用性。实验结果验证了该设计在命中判断、多路选择以及AXI总线交互中的正确性和效率。