实验成绩：
教　　师：

计算机体系结构
实 验 报 告

姓名：_____杨明达_____

班号：_____22R0311_____

学号：_____2022110829_____

哈尔滨工业大学计算学部

2023 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）

1. 实验设备

处理器：11th Gen Intel(R) Core(TM) i5-11260H @ 2.60GHz 2.61 GHz
机带 RAM 16.0 GB (15.7 GB 可用)
系统类型 64 位操作系统, 基于 x64 的处理器
GPU ：NVIDIA GeForce RTX 3050 Laptop GPU

2. 开发环境

Window 11
Vivado 2023.2

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	In/Out	npc/ ID_EX_PC	下一程序计数器值
IF_ID_IR	32	In/Out	指令寄存器 /conflict、register、 extend、ID_EX_IR	指令寄存器值

ID/EX 段

寄存器名	位数	方向	来源/去向	意义
ID_EX_PC	32	In/Out	IF_ID_PC /EX_MUX1、	下一程序计数器值

			EX_MEM_RS	
ID_EX_IR	32	In/Out	IF_ID_IR /conflict、ALU、zero、EX_MEM_IR、EX_MEM_RS	指令寄存器值
ID_EX_R1	32	In/Out	register/cond_mux1	寄存器 A 的值
ID_EX_R2	32	In/Out	register/cond_mux2	寄存器 B 的值
ID_EX_IM	32	In/Out	extend/EX_MUX2	立即数值

EX/MEM 段

寄存器名	位数	方向	来源/去向	意义
EX_MEM_RS	32	In/Out	EX_MUX/IF_MUX、cond_mux1、cond_mux1、MEM_WB_RS	ALU 运算结果
EX_MEM_RG	32	In/Out	cond_mux2/	操作数 B 的值
EX_MEM_IR	32	In/Out	ID_EX_IR、cond_mux2/conflict、MEM_WB_IR	指令寄存器值
EX_MEM_JP	32	In/Out	zero/IF_MUX	跳转地址

MEM/WB 段

寄存器名	位数	方向	来源/去向	意义
MEM_WB_RS	32	In/Out	EX_MEM_RS /cond_mux1、cond_mux2、WB_MUX	ALU 运算结果
MEM_WB_IR	32	In/Out	EX_MEM_IR /conflict、register、WB_MUX	指令寄存器值
MEM_WB_MM	32	In/Out	数据寄存器/ cond_mux1、cond_mux2、WB_MUX	读取的数据

2. 给出定向控制的设计方案，设计方案要求包括：

① 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）
EX/MEM.RS	000000（算术）111110（比较）	ID/EX.R1	000000（算术）111110（比较）	R1	IR2[25:21] == IR3[15:11]
EX/MEM.RS	000000（算术）111110（比较）	ID/EX.R1	000000（算术）111110（比较）	R2	IR2[20:16] == IR3[15:11]
MEM/WB.RS	000000（算术）111110（比较）	ID/EX.R2	000000（算术）111110（比较）	R1	IR2[25:21] == IR3[15:11]
MEM/WB.RS	000000（算术）111110（比较）	ID/EX.R2	000000（算术）111110（比较）	R2	IR2[20:16] == IR3[15:11]
EX/MEM.MM	100011	ID/EX.R1	000000（算术）111110（比较）	R1	IR2[25:21] == IR4[20:16]
EX/MEM.MM	100011	ID/EX.R2	000000（算术）111110（比较）	R2	IR2[20:16] == IR4[20:16]

② 流水线增设的定向路径

（1）转发路径 1：从 EX/MEM.RS 到 ID/EX.R1（算术或比较）

条件：当前指令的源寄存器 R1 与上一条指令的目的寄存器相匹配

上一条指令是算术指令（操作码 6'b000000，与 ADD、SUB 等 R 型指令相对应）或比较指令（操作码 6'b111110，与 CMP 等指令相对应）。

执行：将来自上一条指令 EX/MEM 时的 ALU 计算结果转发给当前指令 ID/EX 时的 R1 操作数，从而绕过了从寄存器文件读取的需要。

（2）转发路径 2：从 EX/MEM.RS 到 ID/EX.R2（算术或比较）

条件：当前指令的源寄存器 R2 与上一条指令的目的寄存器相匹配

上一条指令是算术指令（操作码 6'b000000，与 ADD、SUB 等 R 型指令相对应）或比较指令（操作码 6'b111110，与 CMP 等指令相对应）。

执行：将来自上一条指令 EX/MEM 时的 ALU 计算结果转发给当前指令 ID/EX 时的 R2 操作数，从而绕过了从寄存器文件读取的需要。

（3）转发路径 3：从 MEM/WB.RS 到 ID/EX.R1（算术或比较）

条件：当前指令的源寄存器 R2 与上两条指令的目的寄存器相匹配

上两条指令是算术指令（操作码 6'b000000，与 ADD、SUB 等 R 型指令相对应）或比较指令（操作码 6'b111110，与 CMP 等指令相对应）。

执行：将来自上两条指令 EX/MEM 时的 ALU 计算结果转发给当前指令 ID/EX 时的 R1 操作数，从而绕过了从寄存器文件读取的需要。

（4）转发路径 4：从 MEM/WB.RS 到 ID/EX.R2（算术或比较）

条件：当前指令的源寄存器 R2 与上两条指令的目的寄存器相匹配

上两条指令是算术指令（操作码 6'b000000，与 ADD、SUB 等 R 型指令相对应）或比较指令（操作码 6'b111110，与 CMP 等指令相对应）。

执行：将来自上两条指令 EX/MEM 时的 ALU 计算结果转发给当前指令 ID/EX 时的 R2 操作数，从而绕过了从寄存器文件读取的需要。

（5）转发路径 5：从 MEM/WB.MM 到 ID/EX.R1（加载指令）

条件：当前指令的源寄存器 R1 与上两条指令的加载指令的基寄存器相匹配

上两条指令是一条加载指令（6'b100011 表示 LW）。

执行：将来自上两条指令的内存加载结果直接转发给当前指令的 R1 操作数。从而绕过从寄存器文件读取的需要。

（6）转发路径 6：从 MEM/WB.MM 到 ID/EX.R2（加载指令）

条件：当前指令的源寄存器 R2 与上两条指令的加载指令的基寄存器相匹配

上两条指令是一条加载指令（6'b100011 表示 LW）。

执行：将来自上两条指令的内存加载结果直接转发给当前指令的 R2 操作数。从而绕过从寄存器文件读取的需要。

③ 定向控制设计框图及功能描述

流水线 CPU 的基本功能的设计框图如下图所示。

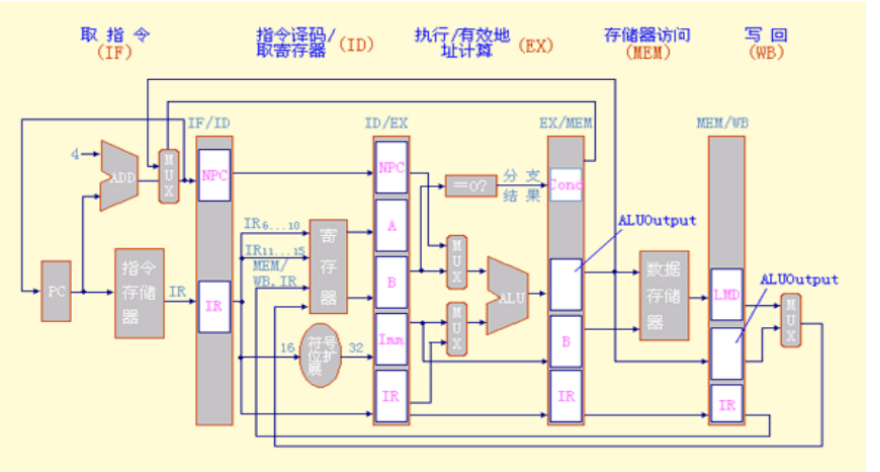


图 1-1 流水线数据通路

而实现定向控制，即在上图的基础上增加 conflict 和 cond_mux 模块进行冲突判断及控制信号生成的操作。

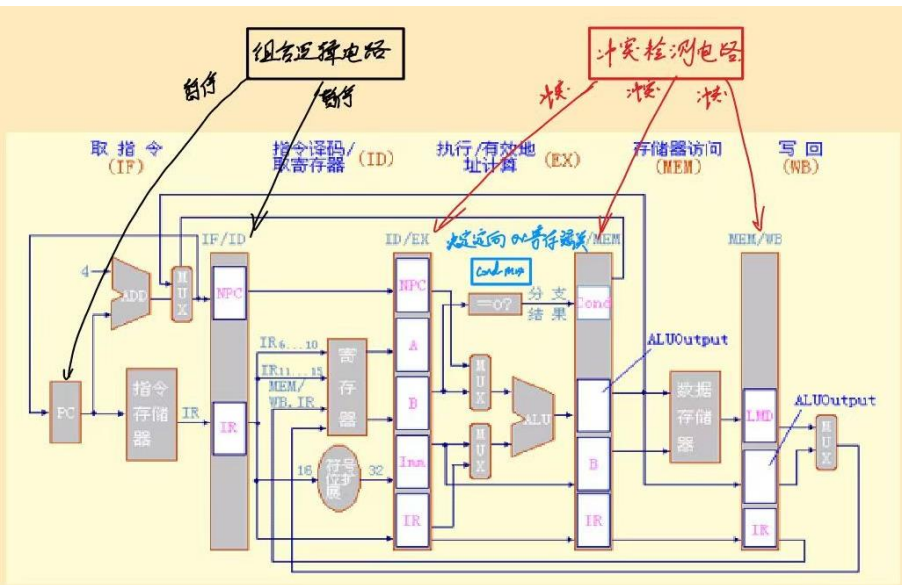


图 1-2 流水线数据通路（加入定向控制）

其中，对 conflict 分析，该模块用于检测和处理流水线中的数据冒险，并通过转发技术解决相应的数据依赖问题，保证处理器流水线高效运行。这个模块主要功能是根据指令之间的依赖关系，判断是否需要插入暂停，以及是否需要通过转发机制将数据从后续流水线阶段传递给前面的阶段，避免不必要的流水线暂停，从而提高流水线处理器的效率。它确保指令能够在不等待写回的情况下，使用前序指令产生的数据，特别是对于加载指令、算术运算指令和比较指令等。通过这些优化，流水线能够更加高效地执行，减少了由于数据依赖带来的延迟。

对 cond_mux 分析，该模块实现了一个**条件多路复用器**，有效地实现了数据转发功能，确保在不同的流水线阶段之间正确传递数据，解决了数据依赖问题，提高了流水线的性能。通过控制信号的选择，模块能够根据流水线中的不同阶段（如 EX/MEM、MEM/WB）的实际情况动态选择转发路径，是正常从寄存器获取数据还是通过定向路径获取数据。

在 Vivado 电路图仿真环节中，完整的 CPU 设计框图如下所示。

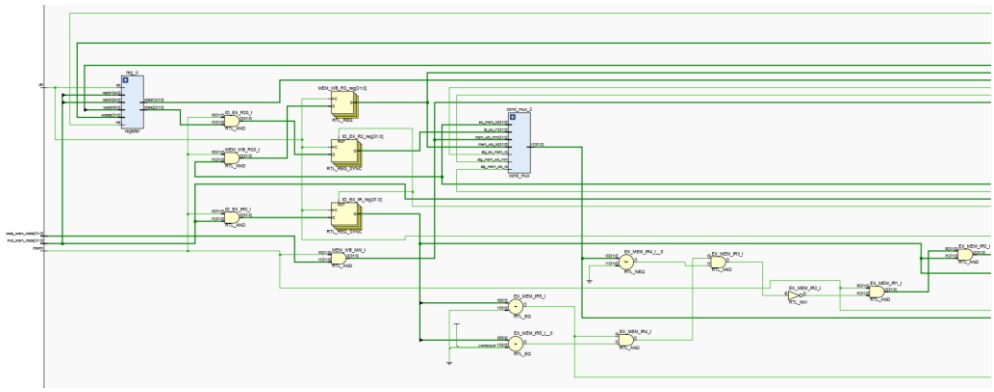


图 1-3 CPU 电路图（1）

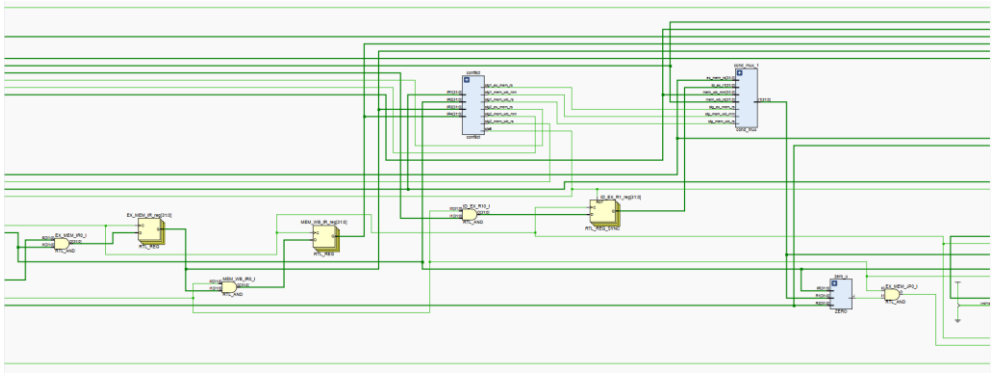


图 1-4 CPU 电路图（2）

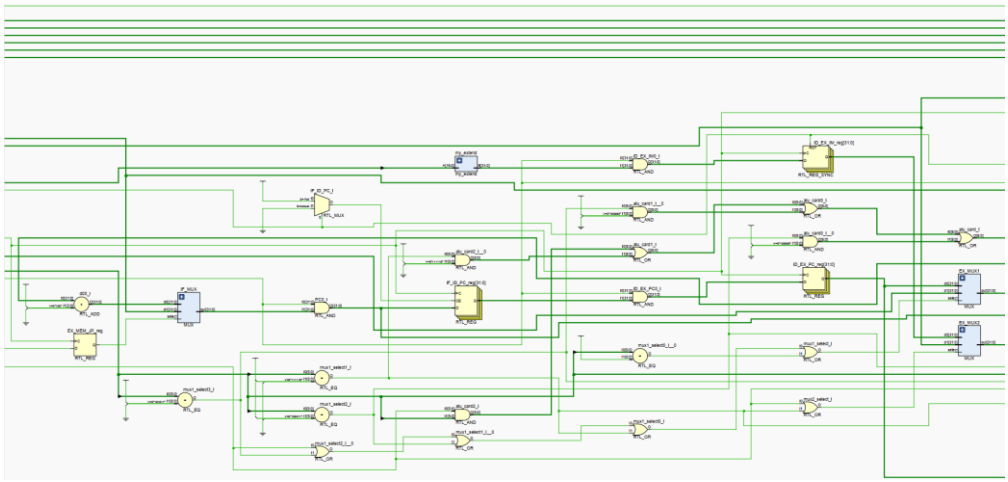


图 1-5 CPU 电路图（3）

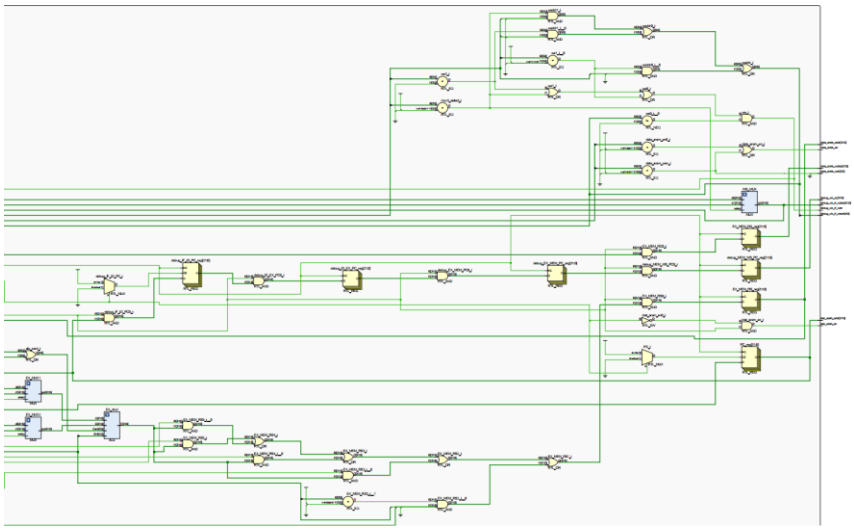


图 1-6 CPU 电路图（4）

四、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）

（1）测试用例 0

图 1-7 测试用例 0 波形图

```
launch_simulation: Time (s): cpu = 00:00:04 ; elapsed = 00:00:51 . Memory (MB): peak = 1686.801 ; gain = 10.746
run all
------------------
   HIT COA Lab1
Pipeline CPU Design
------------------
Run test case 0
CORRECT! You have done well!
------------------
$finish called at time : 69835 ns : File "E:/vivado_project/tixijiegou-lab1/lab1/lab1.srcs/sources_1/new/cpu_checker.v" Line 96
```

图 1-8 测试用例 0 终端结果

（2）测试用例 1

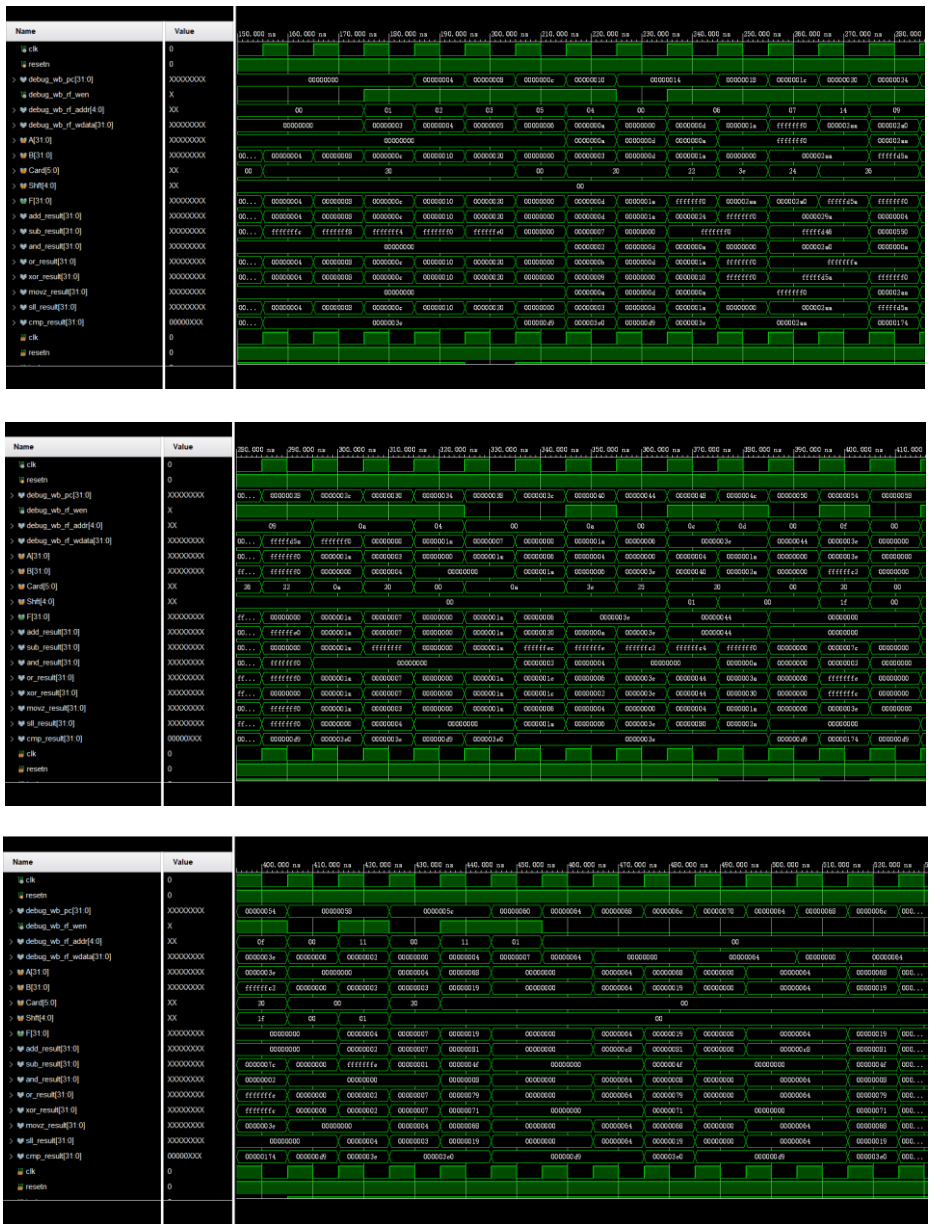


图 1-9 测试用例 1 波形图

```
launch_simulation: Time (s): cpu = 00:00:01 ; elapsed = 00:00:47 . Memory (MB): peak = 2394.207 ; gain = 0.000
run all
--------------------
   HIT COA Lab1
Pipeline CPU Design
--------------------
Run test case 1
CORRECT! You have done well!
--------------------
$finish called at time : 70545 ns : File "E:/vivado_project/tixijiegou-lab1/lab1/lab1.srcs/sources_1/new/cpu_checker.v" Line 96
```

图 1-10 测试用例 1 终端结果

（3）测试用例 2

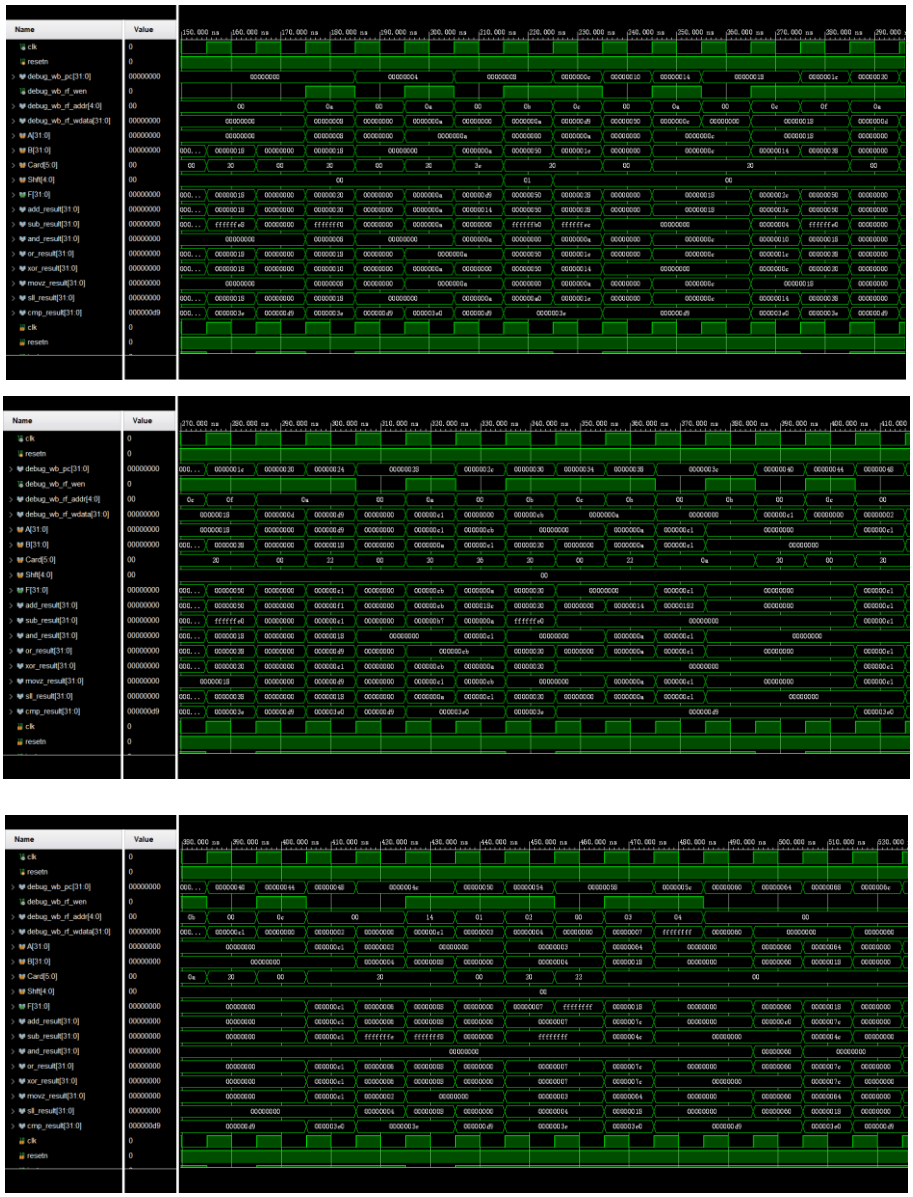


图 1-11 测试用例 2 波形图

```
launch_simulation: Time (s): cpu = 00:00:01 ; elapsed = 00:00:48 . Memory (MB): peak = 2394.207 ; gain = 0.000
run all
------------------
   HIT COA Lab1
Pipeline CPU Design
------------------
Run test case 2
CORRECT! You have done well!
------------------
$finish called at time : 70545 ns : File "E:/vivado_project/tixijiegou-lab1/lab1/lab1.srcs/sources_1/new/cpu_checker.v" Line 96
```

图 1-12 测试用例 2 终端结果

2．暂停和定向控制

① 利用暂停解决数据冲突

通过实验观察及理论分析，可看出在测试用例 0 中，在 sll r12,r4,4 和 sw r10,0x40(r11)中间插入 nop 暂停，确保上两条指令的数据能及时存到 r11 中。在 `movz r5,r12,r11` 和 `cmp r9,r6,r31` 中间插入 nop 暂停，确保上一条指令的数据能及时存到 r6 中。在测试用例 1 中，在 `add r0,r1,r2` 和 `movz r10,r4,r0` 中间插入 nop 暂停，确保上一条指令的数据能及时存到 r4 中。在测试用例 2 中，在 `r10,r10,r12` 和 `r11,r10,r11` 中间插入 nop 暂停，确保上一条指令的数据能及时存到 r10 中。

通过实验，可以明显看出暂停能解决数据冲突，但是利用暂停技术也会增加 CPU 的处理时间，所以引入定向控制。

② 利用定向控制解决数据冲突

通过实验观察及理论分析，可看出在测试用例 1 中，在 lw r4,0x20(r0)和 add r6,r4,r1 中间采用定向技术，将 lw 指令即将写到 r4 的数传入 add 指令的寄存器中，防止暂停的发生。在 add r6,r6,r6 和 sub r7,r4,r6 中间采用定向技术，将 add 指令即将写到 r6 的数传入 sub 指令的寄存器中，防止暂停的发生。在测试用例 2 中，在 lw r10,0x18(r10)和 add r11,r10,r0 中间采用定向技术，将 lw 指令即将写到 r10 的数传入 add 指令的寄存器中，防止暂停的发生。在 add r11,r10,r11 和 xor r12,r11,r10 中间采用定向技术，将 add 指令即将写到 r11 的数传入 xor 指令的寄存器中，防止暂停的发生。

通过实验，可以明显看出定向技术能完美解决数据冲突问题，相比于暂停，该技术更加高效，CPU 处理时间更少。但该方法有一定局限性，即定向技术无法解决所有的数据冲突，所以解决数据冲突要考暂停和定向一起协作完成。

六、实验总结

通过本次实验，通过完成流水线处理器，我充分掌握 Vivado 集成开发环境，掌握 Verilog 语言，掌握 FPGA 编程方法及硬件调试手段，深刻理解流水线型处理器结构和数据冲突解决技术的工作原理。巩固了理论课程中流水线处理的相关知识，知道了流水线处理器的周期划分及数据冲突的处理方法。

暂停控制和定向控制都能有效解决数据冲突的问题。暂停控制在处理加载相关冲突时非常关键，但会引入额外的延迟，影响处理速度。而定向控制则在处理与 ALU 操作相关的冲突时表现更为优越，能够显著提升吞吐量。因此，二者各有其独特优势，合理结合使用可以在不同情况下优化流水线处理器的性能。

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）

1. 实验设备

处理器：11th Gen Intel(R) Core(TM) i5-11260H @ 2.60GHz 2.61 GHz

机带 RAM 16.0 GB (15.7 GB 可用)

系统类型 64 位操作系统, 基于 x64 的处理器

GPU ：NVIDIA GeForce RTX 3050 Laptop GPU

2. 开发环境

Window 11

Vivado 2023.2

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？

因为现代处理器的流水线结构要求指令按顺序执行，而分支指令（如条件跳转指令）可能会改变程序的控制流，导致流水线停滞或失效，所以分支预测器的设计对于现代 CPU 的性能至关重要。

减少流水线停滞。当遇到分支指令时，CPU 需要决定下一条执行的指令地址。如果不能提前知道分支的结果，CPU 需要等待该指令的执行结果，这将导致流水线的空转，浪费宝贵的时钟周期。分支预测器通过预测分支是否会发生，提前加载可能的指令，避免流水线停滞，从而提高处理器的执行效率。

提高指令流水线的效率。在遇到分支指令时，分支预测器提前预测分支是否会被执行，或者预测分支跳转到哪个地址。如果预测正确，处理器可以继续执行流水线中的指令，而无需等待分支结果，从而提高指令的吞吐量。相反，如果预测错误，处理器会丢弃已执行的指令并重新加载正确的指令，但即便如此，及时的预测也能大大减少错误预测带来的影响。

降低分支延迟。分支指令通常会引入较长的延迟，特别是在复杂的指令流水线中。通过有效的分支预测，可以大大减少分支带来的延迟，提升处理器的指令执行速度。尤其在现代超标量和高频率处理器中，分支预测器能够显著提高吞吐量和时钟频率。

优化多指令并行执行。在超标量架构和流水线深度较大的处理器中，多个指令可能在不同的阶段并行执行。分支预测器能够帮助处理器在处理复杂的程序流时，提前预取指令，保持多个指令通道的顺畅，避免无效的等待和阻塞。

提高缓存命中率。分支预测能够帮助程序预取和加载相关指令，从而提高指令缓存的命中率，减少由于缓存未命中的访问延迟，提升整体性能。

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。

在分支预测器的实现过程过，使用的是基于两位饱和计数器的分支预测算法，同时结合了一个分支目标缓冲区来存储跳转指令的目标地址。该算法旨在通过预测分支指令是否会跳转，以及跳转的目标地址来提高流水线效率，减少由于分支指令带来的延迟。实现流程具体为。

首先是初始化，BTB（分支目标缓冲区）存储了每个分支指令的目标地址，并且使用两位饱和计数器（BPB）来指示预测结果。BPB 由两位状态组成，初始值为 11（表示强预测跳转），尽可能避免误判跳转。

其次是分支指令的预测。当处理器遇到分支指令时，分支预测器根据 BTB 中的条目进行预测。BTB 使用 PC 地址的低 6 位（old_PC[7:2]）作为索引，检查该位置的条目。根据 BPB 的状态，判断该分支是否会跳转。其中 00 表示强不跳转。01 表示弱不跳转。10 表示弱跳转。11 表示强跳转。如果预测跳转，new_PC 将被设置为 BTB 中存储的目标地址。如果预测不跳转，则 new_PC 设置为顺序地址（即当前 old_PC + 4）。

接着是更新 BTB。当处理器执行了分支指令并得到了实际的跳转结果（upd_jump），分支预测器将更新 BTB 中的条目以及相应的 BPB。如果预测失败（upd_predfail 为 1），则会调整 BPB，减小其预测的信心。例如，如果预测为跳转（11），但实际没有跳转，则将 BPB 状态调整为 10，表示弱跳转。如果预测成功，则根据实际的跳转结果更新 BPB。如果预测正确且跳转发生，则增加 BPB 的信心（从 01 到 11）。

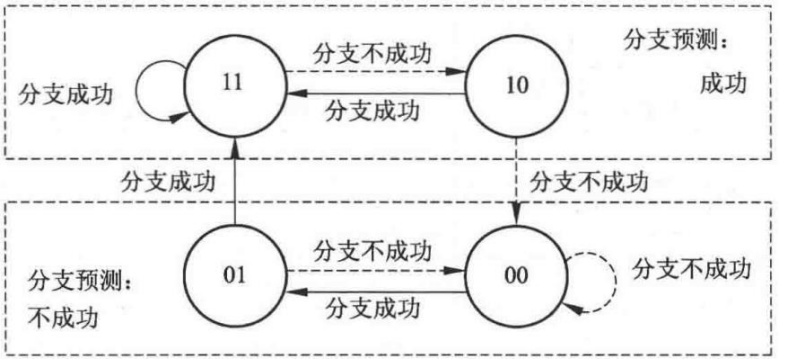


图 2-1 分支预测状态转移图

然后是分支目标更新。每次成功预测跳转后，BTB 会保存跳转的目标地址。当一个分支指令的目标地址被成功计算或获取时，BTB 将更新相应条目的目标地址。对于无条件跳转，使用绝对地址；对于条件跳转，使用相对地址。

接着是分支预测错误时的处理。当发生预测失败时，处理器会重新计算正确的目标地址（rec_target）并执行跳转，更新 new_PC 为新的目标地址。同时，处理器会更新 BTB，调整 BPB 的状态，减少后续错误预测的概率。

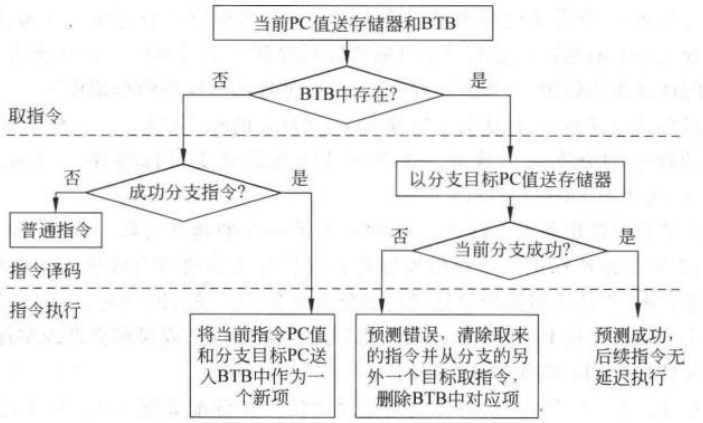


图 2-2 分支目标预测

最后 new_PC 输出预测的下一个指令地址。如果预测失败，new_PC 会被重新计算；如果预测成功且预测跳转，new_PC 将为跳转目标地址。

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）

（1）TTTTTTTTTTN

初始设置 11（强跳转），前 11 次预测跳转，实际也是跳转，故第 12 次分支预测状态 11，即跳转，实际是不跳转，所以**预测失败**，分支预测状态为 10。所以预测 12 次，错误 1 次，预测失败率为 1/12=8.3%。

（2）TTNTTNTTNTTN

初始设置 11（强跳转），前 2 次预测跳转，实际也是跳转，故第 3 次分支预测状态 11，即跳转，实际是不跳转，所以**预测失败**，分支预测状态为 10。第 4 次实际跳转，预测成功，分支预测状态为 11。第 5 次实际跳转，预测成功，分支预测状态为 11。第 6 次实际不跳转，**预测失败**，分支预测状态为 10。第 7 次实际跳转，预测成功，分支预测状态为 11。第 8 次实际跳转，预测成功，分支预测状态为 11。第 9 次实际不跳转，**预测失败**，分支预测状态为 10。第 10 次实际跳转，预测成功，分支预测状态为 11。第 11 次实际跳转，预测成功，分支预测状态为 11。第 12 次实际不跳转，**预测失败**，分支预

测状态为 10。所以预测 12 次，错误 4 次，预测失败率为 4/12=33.3%。

（3）NNNTNNNNNNTT

初始设置 11（强跳转）。第 1 次预测跳转，实际不跳转，预测**失败**，分支预测状态 10。第 2 次实际不跳转，预测**失败**，分支预测状态为 01。第 3 次实际不跳转，预测成功，分支预测状态为 10。第 4 次实际跳转，预测成功，分支预测状态为 11。第 5 次实际不跳转，预测**失败**，分支预测状态为 10。第 6 次实际不跳转，预测**失败**，分支预测状态为 01。第 7 次实际不跳转，预测成功，分支预测状态为 10。第 8 次实际不跳转，预测**失败**，分支预测状态为 01。第 9 次实际不跳转，预测成功，分支预测状态为 10。第 10 次实际不跳转，预测**失败**，分支预测状态为 01。第 11 次实际跳转，预测**失败**，分支预测状态为 00。第 12 次实际跳转，预测**失败**，分支预测状态为 00。所以预测 12 次，错误 8 次，预测失败率为 8/12=66.7%。

四、代码实现

写出带有详细注释的 Verilog 实现代码：

```
1. // BRANCH.v
2. module branch_predictor(
3.     input           clk,        //时钟信号，必须与 CPU 保持一致
4.     input           resetn,     //低有效复位信号，必须与 CPU 保持一
   致
5.
6.     //供 CPU 第一级流水段使用的接口：
7.     input[31:0]     old_PC,         //上一个指令地址
8.     input           predict_en,     //这周期是否需要更新 PC（进行
   分支预测）
9.     output[31:0]    new_PC,         //预测出的下一个指令地址
10.    output          predict_jump,       //是否被预测为执行转移的
   转移指令
11.
12.    //分支预测器更新接口：
13.    //更新使能
14.    input           upd_en,
15.    //转移指令地址
16.    input[31:0]     upd_addr,
17.    //是否为转移指令
18.    input           upd_jumpinst,
19.    //若为转移指令，则是否转移
```

```
    input            upd_jump,
    //是否预测失败
    input            upd_predfail,
    //转移指令本身的目标地址（无论是否转移）
    input[31:0]      upd_target,    //分支指令的目标地址
    input[31:0]      failed_pc,     //有关上次失败预测的 pc
    input[31:0]     failed_ir       //有关上次失败预测的 ir

);

    reg [65:0]BTB[63:0];     //创建一个根据 PC[7:2]寻址的 BTB 表项，相当于直接映射后续寻找只需要对比一个项就可以
    integer i;

    initial begin
    for (i = 0; i < 64; i = i + 1) begin
      BTB[i] = 66'h0;             //必须全部初始化，不能存在 XXXXX，否则最后 assign 赋值会赋值失败
    end
    for(i=0;i<64;i=i+1) begin
      BTB[i][1:0]=2'b11;          //将 BPB 两位初始化为 11
    end

    end
    //更加失败的分支指令类型计算目标地址
    //如果指令是无条件跳转（000010 操作码），它会计算绝对地址
    //如果它是一个条件分支（111111 操作码），它会计算一个相对地址
    wire [31:0] rec_target;
    assign rec_target=({32{failed_ir[31:26] == 6'b000010}} & {
     32'd4+failed_pc[31:28], failed_ir[25:0], 2'b00 }) |   // 无条件跳转指令
                ({32{failed_ir[31:26] == 6'b111111}} & (failed_pc + 32'd4+{{14{failed_ir[15]}}, failed_ir[15:0], 2'b00}));
    /*为 BTB 赋值，BTB 初始时没有任何信息，之后随着 upd 反馈来添加转移指令以及转移目*/
    always @(posedge clk) begin
    //upd_en仅当为高且 BTB 条目为空(目标地址字段为零)时才更新条目
```

```
51.     if(upd_en==1'b1&&upd_jumpinst==1'b1&&BTB[upd_addr[7:2]][33
   :2]==32'h0) begin  //只有在指定位置全是 0 才能进行赋值，不是 0 代表已
   经存在指令
52.     BTB[upd_addr[7:2]][33:2]=upd_addr;     //
53.     BTB[upd_addr[7:2]][65:34]=upd_target;  //
54.     end
55.     end
56.
57.     //此块更新给定 BTB 条目的 BPB。它使用 2 位饱和计数器来跟踪预测结
   果
58.     //如果预测失败，则将计数器减少 00
59.     //如果预测成功，它就会向着增加 11
60.     always @(upd_en) begin           //根据转移结果调整两位 BPB
61.     if(upd_jumpinst==1'b1&&upd_en==1'b1&&upd_addr==BTB[upd_add
   r[7:2]]) begin  //更新条件
62.
63.     if(upd_predfail==1'b1) begin    //预测失败而更新
64.
65.     if(BTB[upd_addr[7:2]][1:0]==2'b00) begin
66.     BTB[upd_addr[7:2]][1:0]<=2'b00;
67.     end
68.     else if(BTB[upd_addr[7:2]][1:0]==2'b01) begin
69.     BTB[upd_addr[7:2]][1:0]<=2'b00;
70.     end
71.     else if(BTB[upd_addr[7:2]][1:0]==2'b10) begin
72.     BTB[upd_addr[7:2]][1:0]<=2'b00;
73.     end
74.     else if(BTB[upd_addr[7:2]][1:0]==2'b11) begin
75.     BTB[upd_addr[7:2]][1:0]<=2'b10;
76.     end
77.
78.     end
79.
80.     else if(upd_predfail==1'b0) begin    //预测成功而更新
81.
82.     if(BTB[upd_addr[7:2]][1:0]==2'b00) begin
83.     BTB[upd_addr[7:2]][1:0]<=2'b01;
84.     end
85.     else if(BTB[upd_addr[7:2]][1:0]==2'b01) begin
```

```
86.     BTB[upd_addr[7:2]][1:0]<=2'b11;
87.     end
88.     else if(BTB[upd_addr[7:2]][1:0]==2'b10) begin
89.     BTB[upd_addr[7:2]][1:0]<=2'b11;
90.     end
91.     else if(BTB[upd_addr[7:2]][1:0]==2'b11) begin
92.     BTB[upd_addr[7:2]][1:0]<=2'b11;
93.     end
94.
95.     end
96.
97.     end
98.     end
99.     //预测开始
100.     //upd_predfail 表示预测失败的情况。如果当前周期发生了分支预测失败，分支预测器会纠正之前的错误
101.     //upd_jump == 1：表示当前指令确实发生了跳转，且预测失败。此时 new_PC 被设置为重新计算的目标地址 rec_target
102.     //upd_jump == 0：表示当前指令实际上不跳转，但之前被错误预测为跳转。此时 new_PC 被设置为 failed_pc + 4，也就是该指令的下一条顺序地址
103.     //只有 upd_predfail == 0 的情况下，才会检查这部分代码，即在预测成功的条件下进一步判断 new_PC 的值
104.     //如果以上条件都满足，new_PC 被设置为 BTB 中记录的目标地址 BTB[old_PC[7:2]][65:34]。
105.     //否则，表示预测命中但未预测跳转，new_PC 被设置为顺序地址 old_PC + 4
106.     assign new_PC = (upd_predfail) ?
107.                     (upd_jump ? rec_target : (failed_pc +4))
    :
108.                     (old_PC != 32'b0 && old_PC == BTB[old_PC[
    7:2]][33:2] && predict_en == 1'b1 &&
109.                      (BTB[old_PC[7:2]][1:0] == 2'b10 || BTB[o
    ld_PC[7:2]][1:0] == 2'b11)) ?
110.                      BTB[old_PC[7:2]][65:34] :
111.                      (old_PC + 4);
112.     assign predict_jump=(old_PC!=32'b0&&old_PC==BTB[old_PC[7:
    2]][33:2]&&predict_en==1'b1&&(BTB[old_PC[7:2]][1:0]==2'b10||BT
    B[old_PC[7:2]][1:0]==2'b11))? 1'b1:1'b0;
```

```
113.
114.
115.endmodule
```

```
1. // cpu.v
2. module cpu(
3.     input           clk,                // 时钟信号
4.     input           resetn,             // 低有效复位号
5.
6.     output          inst_sram_en,       // 指令存储器读使能
7.     output[31:0]    inst_sram_addr,     // 指令存储器读地址
8.     input[31:0]     inst_sram_rdata,    // 指令存储器读出的数据
9.
10.    output          data_sram_en,    // 数据存储器端口读/写使能
11.    output[3:0]     data_sram_wen,      // 数据存储器写使能
12.    output[31:0]    data_sram_addr,     // 数据存储器读/写地址
13.    output[31:0]    data_sram_wdata,    // 写入数据存储器的数据
14.    input[31:0]     data_sram_rdata,    // 数据存储器读出的数据
15.
16.    // 供自动测试环境进行 CPU 正确性检测
17.    output[31:0]    debug_wb_pc,        // 当前正在执行指令的 PC
18.    output          debug_wb_rf_wen,    //当前通用寄存器组写使能
19.    output[4:0]     debug_wb_rf_wnum,    // 当前通用寄存器组写回
   的寄存器编号
20.    output[31:0]    debug_wb_rf_wdata  // 当前指令要写回的数据
21.);
22.
23.    // ========== 全局定义 ==========
24.    reg[31:0] PC;   //程序计数器
25.    reg[31:0] IR;   //指令寄存器
26.
27.    initial begin
28.        PC = 0;
29.        IR = 0;
30.    end
31.    // ================================
32.
33.    // ========== IF_ID =============
34.    reg[31:0] IF_ID_PC;
```

```
35.
36.     wire[31:0] IF_ID_IR;
37.
38.     reg[31:0] debug_IF_ID_PC;        // 测试
39.     // ================================
40.
41.     // ========== ID_EX =============
42.     reg[31:0] ID_EX_PC;
43.     reg[31:0] ID_EX_IR;
44.     reg[31:0] ID_EX_R1;
45.     reg[31:0] ID_EX_R2;
46.     reg[31:0] ID_EX_IM;
47.
48.     reg[31:0] debug_ID_EX_PC;        // 测试
49.     // ================================
50.
51.     // ========== EX_MEM ============
52.     reg[31:0] EX_MEM_RS;
53.     reg[31:0] EX_MEM_RG;
54.     reg[31:0] EX_MEM_IR;
55.     reg EX_MEM_JP;
56.
57.     reg[31:0] debug_EX_MEM_PC;       // 测试
58.     // ================================
59.
60.     // ========== MEM_WB ============
61.     reg[31:0] MEM_WB_RS;
62.     reg[31:0] MEM_WB_IR;
63.     wire[31:0] MEM_WB_MM;
64.     reg resetmy;
65.     reg[31:0] debug_MEM_WB_PC;       // 测试
66.     // ================================
67.
68.     // ========== conflict===========
69.     wire stall;
70.     wire sig1_ex_mem_rs;
71.     wire sig1_mem_wb_mm;
72.     wire sig1_mem_wb_rs;
73.     wire sig2_ex_mem_rs;
```

```
74.     wire sig2_mem_wb_mm;
75.     wire sig2_mem_wb_rs;
76.     wire test_result;
77.     // ==============================
78.     assign debug_wb_pc = debug_MEM_WB_PC;    // 写回 PC
79.     assign debug_wb_rf_wen   = we;           // 写回使能
80.     assign debug_wb_rf_wnum  = waddr;        // 写回地址
81.     assign debug_wb_rf_wdata = wdata;        // 写回数据
82.
83.     // ============ IF ==============
84.     assign inst_sram_en   = !stall && resetn;
85.     assign inst_sram_addr = PC;
86.
87.     wire[31:0] nPC;
88.
89.     wire[31:0] mux0_result;
90.
91.     // MUX IF_MUX(
92.     //      .d0        (PC + 4),
93.     //      .d1     (EX_MEM_RS),
94.     //      .select (EX_MEM_JP),
95.     //      .out    (mux0_result)
96.     // );  // EX_MEM.JP ? EX_MEM_RS : PC + 4
97.
98.     // assign nPC = mux0_result;
99.
100.     assign IF_ID_IR = inst_sram_rdata;
101.     ////////////////////PREDICTOR//////////////////////
102.     `timescale 1ns / 1ps
103.wire  predict_jump;
104.reg IF_ID_predict_jump;
105.reg ID_EX_PREDICT_JUMP;
106.reg EX_MEM_pred_jump;
107.wire upd_jumpinst;
108.assign upd_jumpinst = (EX_MEM_IR[31:26] == 6'b000010 || EX_ME
    M_IR[31:26] == 6'b111111) ? 1'b1 : 1'b0;
109.
110.wire [31:0] upd_addr;
111.assign upd_addr = debug_EX_MEM_PC;
```

```
112.
113.reg upd_predfail; //
114.reg upd_predfail_reg; //
115.//分支预测赋值阶段
116.always @(*) begin
117.    if (test_result == 1'b1) begin
118.        if (ID_EX_PREDICT_JUMP == 1'b1 && (ID_EX_IR[31:26] ==
    6'b000010 || ID_EX_IR[31:26] == 6'b111111)) begin
119.            upd_predfail_reg = 1'b0;
120.        end else begin
121.            upd_predfail_reg = 1'b1;
122.        end
123.    end else begin
124.        if (ID_EX_PREDICT_JUMP == 1'b1) begin
125.            upd_predfail_reg = 1'b1;
126.        end else begin
127.            upd_predfail_reg = 1'b0;
128.        end
129.    end
130.end
131.always @(posedge clk) begin
132.    upd_predfail <= upd_predfail_reg;
133.end
134.
135.branch_predictor branch_predictor_u(
136.    .clk(clk),
137.    .resetn(resetn),
138.    .old_PC(PC),
139.    .predict_en(!stall),//
140.    .new_PC(nPC),
141.    .predict_jump(predict_jump),
142.    .upd_en(upd_jumpinst),
143.    .upd_addr(upd_addr),
144.    .upd_jumpinst(upd_jumpinst),
145.    .upd_jump(EX_MEM_JP),
146.    .upd_predfail(upd_predfail),
147.    .upd_target(EX_MEM_RS),
148.    .failed_ir(EX_MEM_IR),
149.    .failed_pc(debug_EX_MEM_PC)
```

```
150.);
151.    always @(posedge clk) begin
152.        if(!stall) begin
153.            PC       <= {32{resetn}} & nPC;
154.            IF_ID_PC <= {32{resetn}} & nPC;
155.
156.            debug_IF_ID_PC <= {32{resetn}} & PC;
157.            IF_ID_predict_jump <= {32{resetn}} & predict_jump
    ;
158.        end
159.    end
160.    // ================================
161.
162.    // ============ ID ==============
163.
164.    conflict conflict(  // 组合逻辑检测是否有冲突信号
165.        .IR1( IF_ID_IR),
166.        .IR2( ID_EX_IR),
167.        .IR3(EX_MEM_IR),
168.        .IR4(MEM_WB_IR),
169.        .stall(stall),
170.        .sig1_ex_mem_rs(sig1_ex_mem_rs),
171.        .sig1_mem_wb_rs(sig1_mem_wb_rs),
172.        .sig1_mem_wb_mm(sig1_mem_wb_mm),
173.        .sig2_ex_mem_rs(sig2_ex_mem_rs),
174.        .sig2_mem_wb_rs(sig2_mem_wb_rs),
175.        .sig2_mem_wb_mm(sig2_mem_wb_mm)
176.    );
177.
178.    wire          we;    // 寄存器堆读使能（写回使能）
179.    wire[ 5:0] waddr;    // 寄存器堆写地址
180.    wire[31:0] wdata;    // 寄存器堆写数据
181.
182.    wire[31:0] regfile_rdata1;
183.    wire[31:0] regfile_rdata2;
184.
185.    register reg_U(
186.        .clk   (clk),
187.        .we    (we),
```

```
        .raddr1(IF_ID_IR[25:21]),
        .raddr2(IF_ID_IR[20:16]),
        .waddr (waddr),
        .wdata (wdata),
        .rdata1(regfile_rdata1),
        .rdata2(regfile_rdata2)
    );// 每个上升沿读入数据到 ID_EX.R1  ID_EX.R2


    wire[31:0] extend_imm;
    my_extend my_extend(
        .A     (IF_ID_IR[15: 0]),    //  16 位做符号扩展
        .B     (extend_imm)
    );
    reg [1:0] test_result_counter;  // 2-bit counter to track 3 cycles (0, 1, 2)

always @(posedge clk) begin
    // Reset and update PC values
    ID_EX_PC <= {32{resetn}} & IF_ID_PC;
    debug_ID_EX_PC <= {32{resetn}} & debug_IF_ID_PC;
    ID_EX_PREDICT_JUMP <= {32{resetn}} & IF_ID_predict_jump;

    if (resetn == 0) begin
        test_result_counter <= 0;  // Reset counter on system reset
    end

    if (stall) begin
        // Stall the pipeline, clear the relevant registers
        ID_EX_IR <= 0;
        ID_EX_R1 <= 0;
        ID_EX_R2 <= 0;
        ID_EX_IM <= 0;
    end
    else if ( upd_predfail_reg|| test_result_counter > 0) begin
        // Handle test_result condition: clear IR registers for 3 cycles
```

```
222.         ID_EX_IR <= 0;
223.         EX_MEM_IR <= 0;
224.         MEM_WB_IR <= 0;
225.         IF_ID_predict_jump <= 0;
226.         ID_EX_PREDICT_JUMP <= 0;
227.         EX_MEM_pred_jump <= 0;
228.         resetmy <= 0;
229.
230.         // Increment the counter each cycle while test_result
    _counter < 3
231.         if (test_result_counter < 2) begin
232.             test_result_counter <= test_result_counter + 1;
233.         end else begin
234.             // After 3 cycles, reset the counter and restore
    normal operation
235.             test_result_counter <= 0;
236.             resetmy <= 1;  // Restore resetmy signal after 3
    cycles
237.         end
238.     end
239.     else begin
240.         // Normal operation: update registers
241.         ID_EX_R1 <= {32{resetn}} & regfile_rdata1;
242.         ID_EX_R2 <= {32{resetn}} & regfile_rdata2;
243.         ID_EX_IR <= {32{resetn}} & IF_ID_IR;
244.         ID_EX_IM <= {32{resetn}} & extend_imm;
245.         resetmy <= 1;
246.     end
247.
248.     // Jump condition: clear ID_EX_IR
249.     /*if (EX_MEM_JP) begin
250.         ID_EX_IR <= 0;
251.     end */
252. end
253.     // ==============================
254.
255.     // ============ EX ==============
256.     wire[31:0] alu_a, reg_a;
257.     wire[31:0] alu_b, reg_b;
```

```
258.
259.    cond_mux cond_mux_1(
260.        .sig_ex_mem_rs(sig1_ex_mem_rs),
261.        .sig_mem_wb_rs(sig1_mem_wb_rs),
262.        .sig_mem_wb_mm(sig1_mem_wb_mm),
263.        .ex_mem_rs(EX_MEM_RS),
264.        .mem_wb_rs(MEM_WB_RS),
265.        .mem_wb_mm(MEM_WB_MM),
266.        .id_ex_r1(ID_EX_R1),
267.        .r1(reg_a)
268.    );  //  ID_EX.R1 里获取，还是定向获取
269.    cond_mux cond_mux_2(
270.        .sig_ex_mem_rs(sig2_ex_mem_rs),
271.        .sig_mem_wb_rs(sig2_mem_wb_rs),
272.        .sig_mem_wb_mm(sig2_mem_wb_mm),
273.        .ex_mem_rs(EX_MEM_RS),
274.        .mem_wb_rs(MEM_WB_RS),
275.        .mem_wb_mm(MEM_WB_MM),
276.        .id_ex_r1(ID_EX_R2),
277.        .r1(reg_b)
278.    );  //  ID_EX.R2 里获取，还是定向获取
279.
280.    wire mux1_select, mux2_select;
281.
282.    MUX EX_MUX1(
283.        .d0(ID_EX_PC),
284.        .d1(reg_a),
285.        .select(mux1_select),
286.        .out(alu_a)
287.    );
288.    assign mux1_select =
289.        (ID_EX_IR[31:26] == 6'b000000) |    // 运算指令
290.        (ID_EX_IR[31:26] == 6'b101011) |    // 存数指令
291.        (ID_EX_IR[31:26] == 6'b100011) |    // 取数指令
292.        (ID_EX_IR[31:26] == 6'b111110) |    // 比较指令
293.        (ID_EX_IR[31:26] == 6'b111111);     // 条件指令
294.
295.    MUX EX_MUX2(
296.        .d0(ID_EX_IM),
```

```
297.         .d1(reg_b),
298.         .select(mux2_select),
299.         .out(alu_b)
300.     );
301.     assign mux2_select =
302.         (ID_EX_IR[31:26] == 6'b000000) |     // 运算指令
303.         (ID_EX_IR[31:26] == 6'b111110);      // 比较指令
304.
305.     wire[31:0] alu_result;
306.     wire[ 5:0] alu_card =
307.         ({6{ID_EX_IR[31:26] == 6'b000000}} & ID_EX_IR[5:0]) |
    // 运算操作运算码为后五位
308.         ({6{ID_EX_IR[31:26] == 6'b111110}} & 6'b111110)     |
    // 比较指令特殊指定操作
309.         ({6{ID_EX_IR[31:26] == 6'b101011}} & 6'b100000)     |
    // 存数指令做加法
310.         ({6{ID_EX_IR[31:26] == 6'b100011}} & 6'b100000);
    // 取数指令做加法
311.     ALU EX_ALU(       // 选出来的结果做运算
312.         .A(alu_a),
313.         .B(alu_b),
314.         .F(alu_result),
315.         .Shft(ID_EX_IR[10: 6]),
316.         .Card(alu_card)
317.     );
318.
319.
320.     ZERO zero_u(     // 位测试
321.         .R1(reg_a),
322.         .R2(reg_b),
323.         .IR(ID_EX_IR),
324.         .J(test_result)
325.     );
326.
327.     always @(posedge clk) begin
328.         EX_MEM_RG <= {32{resetn}} & reg_b ;
329.         EX_MEM_pred_jump<= {32{resetn}} & ID_EX_PREDICT_JUMP;

330.         EX_MEM_IR <= {32{resetn}} & {32{resetmy}}& (
```

```
331.             {32{!(ID_EX_IR[31:26] == 6'b000000 && ID_EX_IR[5:
   0] == 6'b001010 && reg_b != 0)}}
332.         ) & ID_EX_IR;
333.         // 如果 MOVZ 指令并且 R2  0 就不继续执行
334.         EX_MEM_RS <= {32{resetn}} & (
335.             ({32{ID_EX_IR[31:26] == 6'b000000}} & alu_result)
    |  // 运算指令使用 ALU
336.             ({32{ID_EX_IR[31:26] == 6'b100011}} & alu_result)
    |  // 取数指令使用 ALU
337.             ({32{ID_EX_IR[31:26] == 6'b101011}} & alu_result)
    |  // 存数指令使用 ALU
338.             ({32{ID_EX_IR[31:26] == 6'b111110}} & alu_result)
    |  // 比较指令使用 ALU
339.             ({32{ID_EX_IR[31:26] == 6'b000010}} & { ID_EX_PC[
   31:28], ID_EX_IR[25:0], 2'b00 }) |   // 无条件跳转指令
340.             ({32{ID_EX_IR[31:26] == 6'b111111}} & (ID_EX_PC +
    {{14{ID_EX_IR[15]}}, ID_EX_IR[15:0], 2'b00}))
341.         );
342.         EX_MEM_JP <= resetn & test_result;
343.         debug_EX_MEM_PC <= {32{resetn}} & debug_ID_EX_PC;
344.     end
345.     // ==============================
346.
347.     // ========== MEM ==============
348.     assign data_sram_addr  = EX_MEM_RS;      // 写地址为运算结
   果
349.     assign data_sram_wdata = EX_MEM_RG;      // 写数据为寄存器堆
   的 R2
350.     assign data_sram_wen   = EX_MEM_IR[31:26] == 6'b101011;
      // 只有存数指令写存
351.     assign data_sram_en    =
352.         (EX_MEM_IR[31:26] == 6'b100011) |        // 取数指令访
   存
353.         (EX_MEM_IR[31:26] == 6'b101011);         // 存数指令访
   存
354.
355.     assign MEM_WB_MM = {32{resetn}} & data_sram_rdata;
356.     // 在下个上升沿才能从 SRAM 里面读出数据
357.
```

```
    always @(posedge clk) begin
        MEM_WB_IR <= {32{resetn}} & EX_MEM_IR;
        MEM_WB_RS <= {32{resetn}} & EX_MEM_RS;

        debug_MEM_WB_PC <= {32{resetn}} & {32{resetmy}}& debug_EX_MEM_PC;
    end
    // ==============================

    // ============ WB ==============
    wire mux3_select;

    MUX WB_MUX(
        .d0(MEM_WB_RS),          // 读结果
        .d1(MEM_WB_MM),          // 读内存
        .select(mux3_select),
        .out(wdata)
    );
    assign waddr =
        ({32{MEM_WB_IR[31:26] == 6'b100011}} & MEM_WB_IR[20:16]) |  // 取数指令写回 IR[20:16]
        ({32{MEM_WB_IR[31:26] == 6'b000000}} & MEM_WB_IR[15:11]) |  // 运算指令写回 IR[15:11]
        ({32{MEM_WB_IR[31:26] == 6'b111110}} & MEM_WB_IR[15:11]);   // 比较指令写回 IR[15:11]
    assign mux3_select =
        (MEM_WB_IR[31:26] == 6'b100011);        // 只有取数指令写回访存结果
    assign we =
        ((MEM_WB_IR[31:26] == 6'b000000) |  // 运算指令要写回
         (MEM_WB_IR[31:26] == 6'b100011) |  // 取数指令要写回
         (MEM_WB_IR[31:26] == 6'b111110) ) &// 比较指令要写回
        (waddr != 0);                   // 不能写入 r0 寄存器
    // ==============================

endmodule
```

```
1. // alu.v
2. `define ADD     6'b100000       // 加
```

```
3. `define SUB     6'b100010       // 减
4. `define AND     6'b100100       // 与
5. `define OR      6'b100101       // 或
6. `define XOR     6'b100110       // 异或
7. `define MOVZ    6'b001010       // 条件移动
8. `define SLL     6'b000000       // 移位
9. `define CMP     6'b111110       // 比较
10.
11.module ALU(
12.    input [31:0] A,     // 输入 A
13.    input [31:0] B,     // 输入 B
14.    input [5:0] Card,   // 操作指令
15.    input [4:0] Shft,    // 移位指令
16.    output [31:0] F     // 输出 F
17.    );
18.
19.    wire [31:0]  add_result = A + B;
20.    wire [31:0]  sub_result = A - B;
21.    wire [31:0]  and_result = A & B;
22.    wire [31:0]   or_result = A | B;
23.    wire [31:0]  xor_result = A ^ B;
24.    wire [31:0] movz_result = A;
25.    wire [31:0]  sll_result = B << Shft;
26.
27.    wire [31:0]  cmp_result = {
28.        22'b0,
29.        !(A <= B),
30.        !($signed(A) <= $signed(B)),
31.        !(A <  B),
32.        !($signed(A) <  $signed(B)),
33.        !(A == B),
34.        A <= B,
35.        $signed(A) <= $signed(B),
36.        A <  B,
37.        $signed(A) <  $signed(B),
38.        A == B
39.    };
40.    // wire [31:0] cmp_result = { A[15:0], B[15:0] };
41.
```

```
    assign F =
        ({32{Card ==  `ADD}} &  add_result) |
        ({32{Card ==  `SUB}} &  sub_result) |
        ({32{Card ==  `AND}} &  and_result) |
        ({32{Card ==   `OR}} &   or_result) |
        ({32{Card ==  `XOR}} &  xor_result) |
        ({32{Card == `MOVZ}} & movz_result) |
        ({32{Card ==  `SLL}} &  sll_result) |
        ({32{Card ==  `CMP}} &  cmp_result);
endmodule
```

```
cond_mux.v
module cond_mux(
    input sig_ex_mem_rs,
    input sig_mem_wb_rs,
    input sig_mem_wb_mm,
    input [31:0] ex_mem_rs,
    input [31:0] mem_wb_rs,
    input [31:0] mem_wb_mm,
    input [31:0] id_ex_r1,
    output [31:0] r1
);
//如果 sig_ex_mem_rs 为真，则选择 ex_mem_rs
//如果 sig_ex_mem_rs 为假，sig_mem_wb_rs 为真，则选择 mem_wb_rs
//如果 sig_ex_mem_rs 和 sig_mem_wb_rs 都为假，sig_mem_wb_mm 为真，则选择 mem_wb_mm
//如果 sig_ex_mem_rs、sig_mem_wb_rs 和 sig_mem_wb_mm 都为假，则选择 id_ex_r1
assign r1 = sig_ex_mem_rs ? ex_mem_rs : sig_mem_wb_rs ? mem_wb_rs : sig_mem_wb_mm ? mem_wb_mm : id_ex_r1;
endmodule
```

```
//conflict.v
module conflict (
    input[31:0] IR1,    // 当前条指令，IF/ID
    input[31:0] IR2,    // 上一条指令，ID/EX
    input[31:0] IR3,    // 上二条指令，EX/MEM
    input[31:0] IR4,    // 上三条指令，MEM/WB
    output stall,
```

```
    output sig1_ex_mem_rs,
    output sig1_mem_wb_rs,
    output sig1_mem_wb_mm,
    output sig2_ex_mem_rs,
    output sig2_mem_wb_rs,
    output sig2_mem_wb_mm
);
    // 暂停
    //如果上一条指令是 LW 指令，且当前指令是运算指令或比较指令，且当前指令的第一个操作数 rs 是上一条指令
//如果上一条指令是 LW 指令，且当前指令是运算指令或比较指令，且当前指令的第二个操作数 rt 是上一条指令
//如果上一条指令是 LW 指令，且当前指令的第一个操作数 rs、base 是上一条指令的目的寄存器 rt，则暂停
    assign stall =
        (IR2[31:26] == 6'b100011 && (IR1[31:26] == 6'b000000 || IR1[31:26] == 6'b111110) && IR2[20:16] == IR1[25:21]) |
        (IR2[31:26] == 6'b100011 && (IR1[31:26] == 6'b000000 || IR1[31:26] == 6'b111110) && IR2[20:16] == IR1[20:16]) |
        (IR2[31:26] == 6'b100011 && IR2[20:16] == IR1[25:21])

        ;
        // ||(IR1[31:26]==6'b111111)||(IR2[31:26]==6'b111111);

    // 从 EX/MEM.RS 定向替代 ID/EX.R1
    //如果上一条指令的源寄存器 rs 是有效的，且上二条指令是运算指令或比较指令，且上二条指令的第一个操作数 rs 是上三条指令的目的寄存器 rt，则定向替代
    assign sig1_ex_mem_rs = (IR2[25:21] != 5'b00000) &&
        (IR3[31:26] == 6'b000000 && IR2[25:21] == IR3[15:11]) | // 运算指令
        (IR3[31:26] == 6'b111110 && IR2[25:21] == IR3[15:11]); // 比较指令

    // 从 EX/MEM.RS 定向替代 ID/EX.R2
    //如果上一条指令的源寄存器 rt 是有效的，且上二条指令是运算指令或比较指令，且上二条指令的第二个操作数 rt 是上三条指令的目的寄存器 rt，则定向替代
```

```
34.     assign sig2_ex_mem_rs = (IR2[20:16] != 5'b00000) &&
35.         (IR3[31:26] == 6'b000000 && IR2[20:16] == IR3[15:11])
    | // 运算指令
36.         (IR3[31:26] == 6'b111110 && IR2[20:16] == IR3[15:11]);
      // 比较指令
37.
38.     // 从 EX/MEM.RS 定向替代 ID/EX.R1
39.     //如果上一条指令的源寄存器 rs 是有效的，且上三条指令是运算指令或比
    较指令，且上三条指令的第一个操作数 rs 是上四条指令的目的寄存器 rt，则定
    向替代
40.     assign sig1_mem_wb_rs = (IR2[25:21] != 5'b00000) &&
41.         (IR4[31:26] == 6'b000000 && IR2[25:21] == IR4[15:11])
    | // 运算指令
42.         (IR4[31:26] == 6'b111110 && IR2[25:21] == IR4[15:11]);
      // 比较指令
43.
44.     // 从 EX/MEM.RS 定向替代 ID/EX.R2
45.     //如果上一条指令的源寄存器 rt 是有效的，且上三条指令是运算指令或比
    较指令，且上三条指令的第二个操作数 rt 是上四条指令的目的寄存器 rt，则定
    向替代
46.     assign sig2_mem_wb_rs = (IR2[20:16] != 5'b00000) &&
47.         (IR4[31:26] == 6'b000000 && IR2[20:16] == IR4[15:11])
    | // 运算指令
48.         (IR4[31:26] == 6'b111110 && IR2[20:16] == IR4[15:11]);
      // 比较指令
49.
50.     // 从 EX/MEM.MM 定向替代 ID/EX.R1
51.     //如果上一条指令的源寄存器 rs 是有效的，且上三条指令是 LW 取数指令，
    且上二条指令的第一个操作数 rs 是上三条指令的目的寄存器 rt，则定向替代
52.     assign sig1_mem_wb_mm = (IR2[25:21] != 5'b00000) &&
53.         (IR4[31:26] == 6'b100011 && IR2[25:21] == IR4[20:16]);
      // 取数指令
54.
55.     // 从 EX/MEM.MM 定向替代 ID/EX.R2
56.     //如果上一条指令的源寄存器 rt 是有效的，且上三条指令是 LW 取数指令，
    且上二条指令的第二个操作数 rt 是上三条指令的目的寄存器 rt，则定向替代
57.     assign sig2_mem_wb_mm = (IR2[20:16] != 5'b00000) &&
58.         (IR4[31:26] == 6'b100011 && IR2[20:16] == IR4[20:16]);
      // 取数指令
```

```
59. endmodule
```

```
1.  // extend.v
2.  module my_extend (
3.      input  [15:0] A,
4.      output [31:0] B
5.  );
6.      assign B = {{16{A[15]}}, A[15:0] };
7.  endmodule
```

```
1.  // mux.v
2.  module MUX(
3.      input[31:0] d0,
4.      input[31:0] d1,
5.      input select,
6.      output[31:0] out
7.  );
8.      assign out = select ? d1 : d0;
9.  endmodule
```

```
1.  // regfile.v
2.  module register (
3.      input clk,
4.      input we,
5.      input [4:0] raddr1,
6.      input [4:0] raddr2,
7.      input [4:0] waddr ,
8.      output [31:0] rdata1,
9.      output [31:0] rdata2,
10.     input [31:0] wdata
11. );
12.     reg [31:0] data[0:31];
13.
14.     integer i;
15.
16.     initial begin
17.         for(i = 0;i < 32;i = i + 1) begin
18.             data[i] <= 0;
19.         end
```

```
    end

    always @(posedge clk) begin
        if(we) begin
            data[waddr] <= wdata;
        end
    end
    assign rdata1 = (waddr == raddr1 && waddr!=0)? wdata : dat
a[raddr1];
    assign rdata2 = (waddr == raddr2 && waddr!=0)? wdata : dat
a[raddr2];
endmodule
```

```
// zero.v
module ZERO (
    input[31:0] R1,
    input[31:0] R2,
    input[31:0] IR,
    output J
);
    assign J = (IR[31:26] == 6'b111111 && R1[IR[20:16]]) | (IR
[31:26] == 6'b000010);
        // 位测试或者无条件
endmodule
```

五、测试结果及实验分析

测试结果（包括你的预测失败率）：

（1）测试用例 0

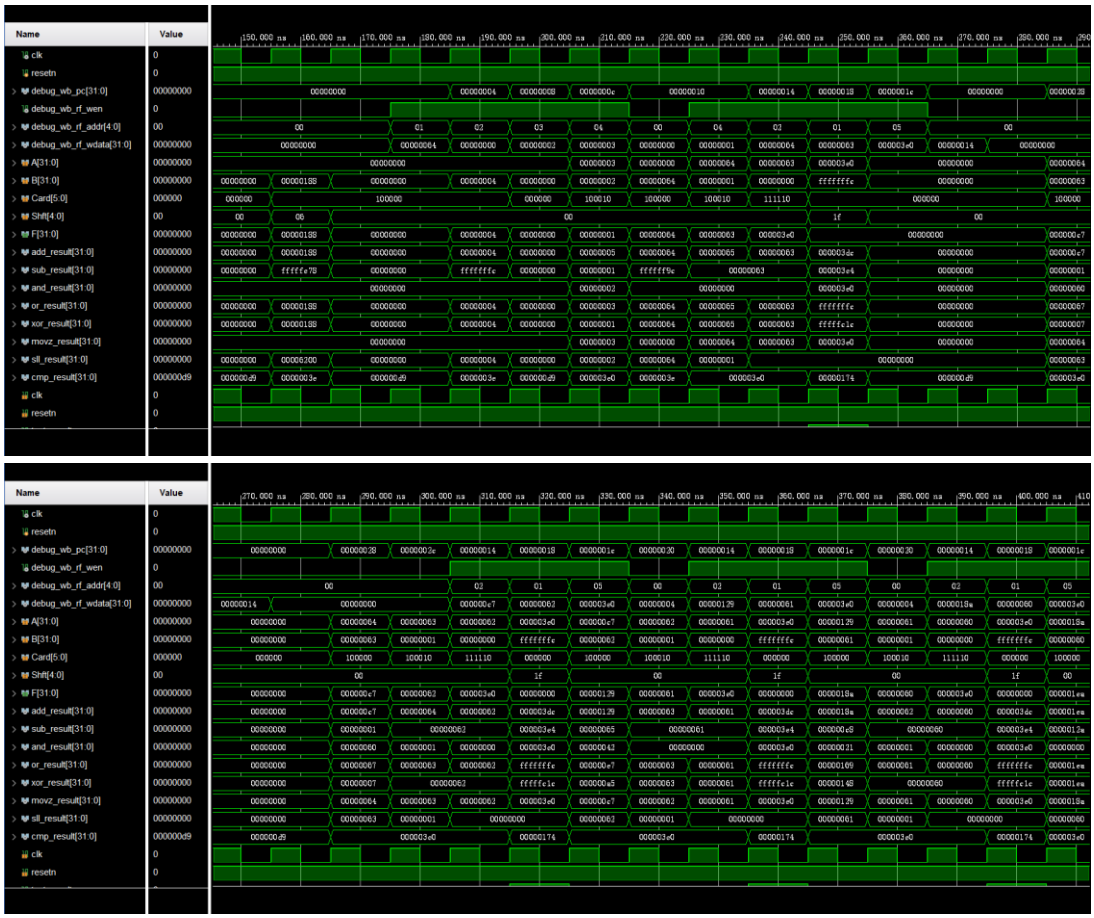


图 2-3 测试用例 0 波形图

```
launch_simulation: Time (s): cpu = 00:00:02 ; elapsed = 00:00:48 . Memory (MB): peak = 1758.395 ; gain = 0.000
run all
WARNING in tb.soc.checker.ir.i0.inst at time                5100 ns:
Reading from out-of-range address. Max address in tb.soc.checker.ir.i0.inst is          31
WARNING in tb.soc.checker.ir.i1.inst at time                5100 ns:
Reading from out-of-range address. Max address in tb.soc.checker.ir.i1.inst is          31
WARNING in tb.soc.checker.ir.i2.inst at time                5100 ns:
Reading from out-of-range address. Max address in tb.soc.checker.ir.i2.inst is          31
--------------------
   HIT C0A Lab2
 Branch Predictor
--------------------
Run test case 0
Your CPU is correct! It spent 417 cycles in total

PASS! You have done well!
--------------------
$finish called at time : 91955 ns : File "E:/vivado_project/tixijiegou-lab2/lab2/lab2.srcs/sources_1/new/cpu_checker.v" Line 133
```

图 2-4 测试用例 0 终端结果

（2）测试用例 1

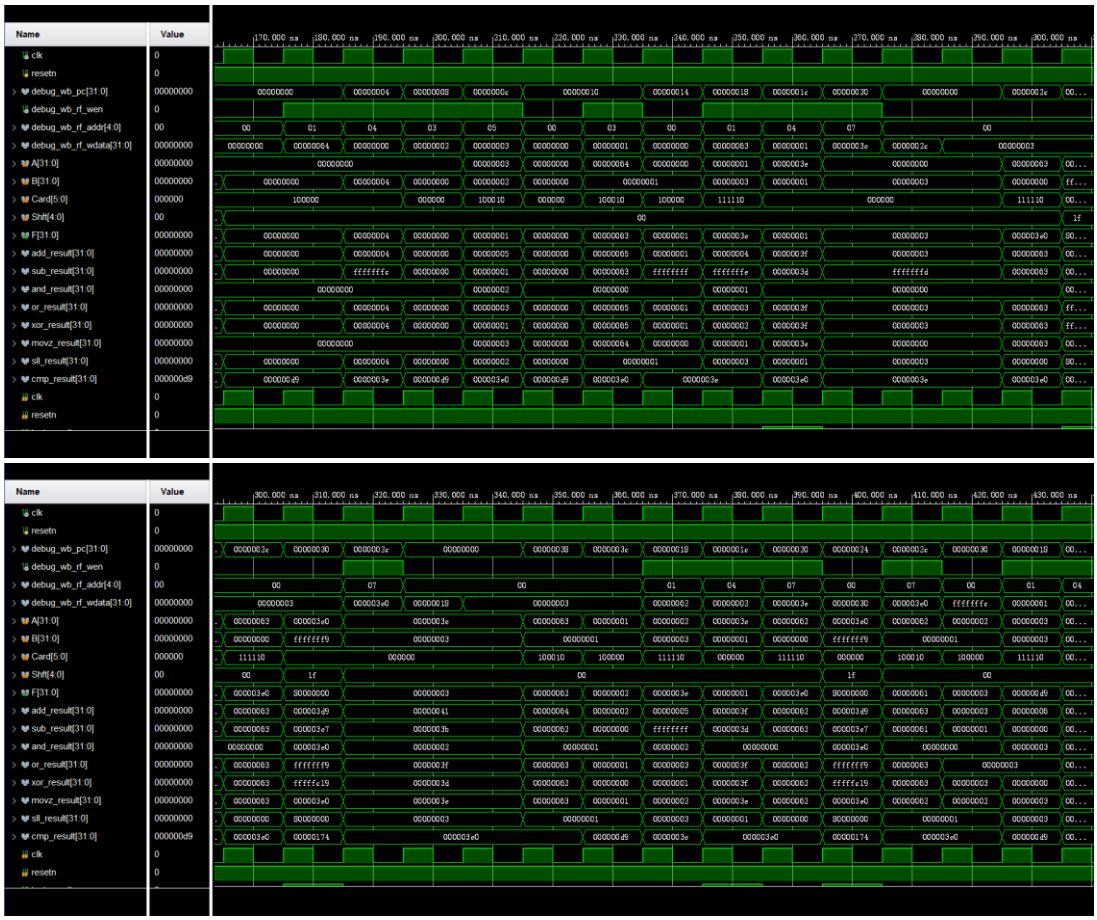


图 2-5 测试用例 1 波形图

```
launch_simulation: Time (s): cpu = 00:00:02 ; elapsed = 00:00:53 . Memory (MB): peak = 2321.426 ; gain = 0.000
run all
WARNING in tb.soc.checker.ir.i0.inst at time                 5100 ns:
Reading from out-of-range address. Max address in tb.soc.checker.ir.i0.inst is          31
WARNING in tb.soc.checker.ir.i1.inst at time                 5100 ns:
Reading from out-of-range address. Max address in tb.soc.checker.ir.i1.inst is          31
WARNING in tb.soc.checker.ir.i2.inst at time                 5100 ns:
Reading from out-of-range address. Max address in tb.soc.checker.ir.i2.inst is          31
--------------------
   HIT C0A Lab2
 Branch Predictor
--------------------
Run test case 1
Your CPU is correct! It spent 753 cycles in total

PASS! You have done well!
--------------------
$finish called at time : 92515 ns : File "E:/vivado_project/tixijiegou-lab2/lab2/lab2.srcs/sources_1/new/cpu_checker.v" Line 133
```

图 2-6 测试用例 1 终端结果

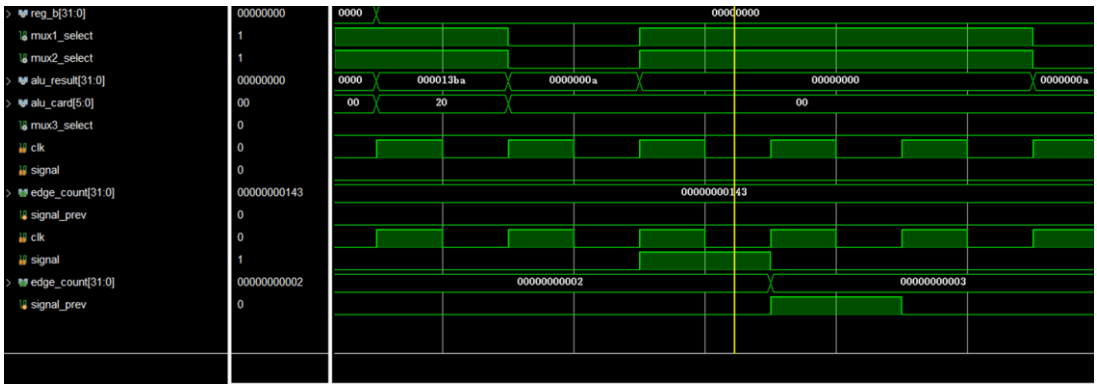
（3）测试用例 2

图 2-7 测试用例 1 波形图

```
launch_simulation: Time (s): cpu = 00:00:02 ; elapsed = 00:00:52 . Memory (MB): peak = 2347.266 ; gain = 0.000
run all
WARNING in tb.soc.checker.ir.i0.inst at time                5100 ns:
Reading from out-of-range address. Max address in tb.soc.checker.ir.i0.inst is          31
WARNING in tb.soc.checker.ir.i1.inst at time                5100 ns:
Reading from out-of-range address. Max address in tb.soc.checker.ir.i1.inst is          31
WARNING in tb.soc.checker.ir.i2.inst at time                5100 ns:
Reading from out-of-range address. Max address in tb.soc.checker.ir.i2.inst is          31
--------------------
   HIT C0A Lab2
 Branch Predictor
--------------------
Run test case 2
Your CPU is correct! It spent 1154 cycles in total

PASS! You have done well!
--------------------
$finish called at time : 92955 ns : File "E:/vivado_project/tixijiegou-lab2/lab2/lab2.srcs/sources_1/new/cpu_checker.v" Line 133
```

图 2-8 测试用例 1 终端结果



对于测试用例 0。通过统计指令运行结束时，最后以此预测使能上升沿为 143，失败次数为 3。故预测失效率为 3/143=2.10%

对于测试用例 1 和 2，利用相同方法，测试失效率为 16.43%和 17.29%。

实验结果分析：

通过实验结果看出，测试用例 0 的预测失效率最低，测试用例 1 和 2 相近，且都高于测试用例 0 的数值。test case1 中无分支预测执行约需 700 个时钟周期，采用两位 BTB 预测仅需 400 左右时钟周期。所以两位 BTB 预测器可实现较好的分支预测效果，加速了处理器处理速度。同时，对于分支预测失败的情况，处理器也能正确地清空流水线。

六、实验总结

通过本次实验，通过完成流水线处理器，我充分掌握 Vivado 集成开发环境，掌握 Verilog 语言，掌握 FPGA 编程方法及硬件调试手段，深刻理解动态分支预测的原理与方法。巩固了理论课程中流水线控制相关的动态解决技术的相关知识。

分支预测是提高流水线处理器性能的关键技术之一。在现代 CPU 架构中，指令执行通常涉及分支指令（如条件跳转、无条件跳转等），这些指令会改变程序的执行流。由于分支指令的不可预测性，若不能提前知道分支的实际结果，处理器必须等待条件判断完成后才能继续执行，这会造成流水线停滞，浪费大量周期。因此，分支预测的主要目的是通过预测分支指令的执行结果来减少流水线停滞，提高 CPU 的指令吞吐量。

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）

1. 实验设备

处理器：11th Gen Intel(R) Core(TM) i5-11260H @ 2.60GHz 2.61 GHz
机带 RAM 16.0 GB (15.7 GB 可用)
系统类型 64 位操作系统, 基于 x64 的处理器
GPU ：NVIDIA GeForce RTX 3050 Laptop GPU

2. 开发环境

Window 11
Vivado 2023.2

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。

（1）状态说明

设计中有几个关键状态，涉及命中和缺失的处理，以及数据加载的过程。我们将缓存的行为划分为以下几种状态：

IDLE (空闲状态)：该状态表示 Cache 未被访问，等待来自 CPU 的请求。当没有 CPU 请求时，系统处于空闲状态。

MISS (缺失状态)：如果 CPU 请求的地址在 Cache 中没有命中，则 Cache 进入缺失状态。在此状态下，Cache 会向主存发起数据请求，开始加载数据。

LOAD (加载状态)：当缓存未命中并且数据正在从主存加载时，系统进入加载状态。在此状态下，系统会等待数据的返回。

HIT (命中状态)：如果 CPU 请求的数据在 Cache 中找到了匹配项，系统进入命中状态，Cache 直接返回数据。

WAIT (等待状态)：如果当前加载数据或者数据还没有准备好，系统会进入等待状态，直到数据加载完成。

UPDATE (更新状态)：当缓存加载完数据后，系统会进入更新状态，将新的数据写入缓存，并准备好供 CPU 使用。

（2）命中与缺失的工作流程

命中（HIT）：如果 Cache 中已经有数据并且标签匹配，系统就会认为是命中。此时：Cache 将数据直接返回给 CPU。cache_data_ok 和 cache_addr_ok 会被设置为 1，表示数据已经准备好。进入 HIT 状态后，系统不会与主存交互，而是直接响应 CPU 请求。

缺失（MISS）：如果 Cache 中没有命中（即标签不匹配），系统进入 MISS 状态，流程如下：Cache 会向主存发起读取请求（通过 AXI 协议）。此时，arvalid 信号被拉高，表示 Cache 正在请求数据。当主存响应并返回数据后，Cache 将数据加载到对应的数据存储器（RAM）中。在加载过程中，cache_data_ok 和 cache_addr_ok 会保持为 0，直到数据加载完成。加载完成后，系统进入 UPDATE 状态，更新缓存，并将数据返回给 CPU。

（3）状态转移与信号说明

IDLE → HIT：当 CPU 发起请求，且数据在 Cache 中命中时，直接返回数据。

信号：cache_rdata = out_data1 或 out_data2。

Cache 响应：cache_data_ok = 1, cache_addr_ok = 1。

IDLE → MISS：当 CPU 发起请求，且数据未命中时，向主存发起数据请求。

信号：arvalid = 1, araddr = cpu_addr。

Cache 响应：cache_data_ok = 0, cache_addr_ok = 0。

MISS → LOAD：当主存返回数据时，进入加载状态，等待数据加载完成。

信号：rvalid = 1, rdata = loaded_data。

Cache 响应：data_ok = 1，等待数据准备好。

LOAD → UPDATE：数据加载完成，更新缓存，并准备数据返回给 CPU。

信号：更新缓存内容，返回数据给 CPU。

UPDATE → HIT：数据已经加载并写回缓存，缓存更新完毕后，返回数据。

信号：cache_rdata 返回给 CPU。

（4）缓存工作过程

CPU 请求:CPU 通过 cpu_req 发送请求到缓存。Cache 会根据地址对比标签（tag），并决定是否命中。

缓存命中（Hit）:如果缓存命中，直接返回数据。此时，cache_rdata 会从缓存中读取数据并发送给 CPU，同时 cache_addr_ok 和 cache_data_ok 变为 1。

缓存缺失（Miss）:如果缓存未命中，缓存会发出 AXI 协议的读请求（arvalid 设置为 1），请求数据从主存读取。在加载数据过程中，cache_addr_ok 和 cache_data_ok 保持为 0，直到数据准备好为止。

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。

对于二路组相联，可以通过实例化两个 icache_tagv_table 模块，分别对应每个组，来实现二路组相联缓存。cpu_index 将决定当前请求映射到哪个组，然后通过分别查询两个组的 tags 和 valids 来检查是否命中。

对于四路组相联，可以实例化四个 icache_tagv_table 模块，对应四个组（每组包含多个缓存行）。cpu_index 决定请求映射到哪个组，多个 tags 和 valids 数组分别用于存储不同组的数据。

在算法描述方面。针对地址映射，根据 CPU 请求的地址（cpu_addr），提取索引部分 cpu_index（如 cpu_addr[11:5]）以及标签部分 r_cpu_tag（如 cpu_addr[31:12]）。针对选择组，cpu_index 用于选择缓存组（组相联的数量）。针对缓存行查找，在组内，检查每个缓存行的标签与请求标签是否匹配，并检查该行的有效位是否为 1。针对命中/未命中，如果有任何缓存行命中，则输出 hit 为高电平；否则，输出为低电平。

四、代码实现

给出实现 Cache 的 Verilog 代码：

```
1. // cache.v
2. module cache (
3.     input             clk, // clock, 100MHz
4.     input             resetn, // active low
5.     //  Sram-Like 接口信号，用于 CPU 访问 Cache
6.     input          cpu_req      ,    //由 CPU 发送至 Cache，CPU 请求信号，表示 CPU 发起读操作
7.     input  [31:0] cpu_addr     ,    //由 CPU 发送至 Cache，CPU 请求地址
8.     output reg [31:0] cache_rdata  ,    //由 Cache 返回给 CPU，缓存返回给 CPU 的数据
9.     output   reg    cache_addr_ok,    //由 Cache 返回给 CPU，缓存地址确认信号，地址已就绪
10.    output   reg  cache_data_ok,    //由 Cache 返回给 CPU，缓存数据确认信号，数据已就绪
11.    //  AXI 接口信号，用于 Cache 访问主存
12.    output reg [3 :0] arid   ,              //Cache 向主存发起读请求时使用的 AXI 信道的 id 号
13.    output reg [31:0] araddr ,              //Cache 向主存发起读请求时所使用的地址
```

```
    output    reg      arvalid,                //Cache 向主存发起读请求的请求信号
    input          aready,          //读请求能否被接收的握手信号
    input  [3 :0] rid,//主存向Cache返回数据使用的AXI信道的id号
    input  [31:0] rdata  ,          //主存向 Cache 返回的数据
    input          rlast  ,//是否是主存向Cache返回的最后一个数据
    input          rvalid ,//主存向Cache返回数据时的数据有效信号
    output    reg     rready                  //标识当前的 Cache 已经准备好可以接收主存返回的数据
);
    /*TODO：完成指令 Cache 的设计代码*/
    reg w_tag_1;     // 双向关联缓存的标签表的写使能信号
    reg w_tag_2;     // 双向关联缓存的标签表的写使能信号
    wire hit_1;      // 指示标签表中的数据是否与请求的数据匹配
    wire hit_2;      // 指示标签表中的数据是否与请求的数据匹配
    reg hit_temp; // 保存 hit
    wire [31:0]out_data1;
    wire [31:0]out_data2;
    reg hit; // 确定最终是否命中
    integer i;  // 计数器
    reg data_ok;
    reg miss; // 标记是否缺失
    reg load; // 标记是否正在载入
    reg reset_i; // 重置 i 信号
    reg [2:0]temp_tr;    // 保存未命中时地址的 offset
//    reg rlast_temp; // 保存 last 数据
    // 当前取数地址的寄存器
    reg [31:0] reg_addr;     //当前请求的地址
    reg [19:0] reg_tag    ; //当前请求地址的 tag
    reg [6 :0] reg_index  ; //当前请求地址的 index
    reg [4 :0] reg_offset ; //当前请求地址的 offset
    integer switch;
//    reg [9 :0] write_addr;
//    reg [31:0] write_data_temp;
    // 实例化两个 tag 表
   // 两个标签表分别对应缓存的两个组，addr 和 tag 用于地址和标签匹配
    // we 用于写使能，hit 用于判断是否命中
    icache_tagv_table tag_table1(
        .clk(clk),
```

```
51.         .resetn(resetn),
52.         .wen(w_tag_1),
53.         .valid_wdata(1'b1),
54.         //         .tag_wdata(reg_tag),
55.         .tag_wdata(reg_addr[31:12]),
56.//         .windex(reg_index),
57.         .windex(reg_addr[11:5]),
58.         .rden(1),
59.         .cpu_addr(cpu_addr),
60.         .hit(hit_1)
61.     );
62.
63.     icache_tagv_table tag_table2(
64.         .clk(clk),
65.         .resetn(resetn),
66.         .wen(w_tag_2),
67.         .valid_wdata(1'b1),
68.//         .tag_wdata(reg_tag),
69.         .tag_wdata(reg_addr[31:12]),
70.//         .windex(reg_index),
71.         .windex(reg_addr[11:5]),
72.         .rden(1),
73.         .cpu_addr(cpu_addr),
74.         .hit(hit_2)
75.     );
76.
77.     // 实例化两个存储数据的 ram
78.     // 表示两组的数据存储器，用于缓存数据块，wea 信号控制写入数据
79.     //addra 表示地址，dina 表示写入的数据，addrb 表示读取的地址，doutb
   表示读取的数据
80.     blk_mem_gen_0 data_1(
81.         .clka(clk),
82.         .wea(w_tag_1),
83.//         .addra(write_addr),
84.         .addra({reg_addr[11:5], 3'd0} + i),
85.//         .dina(write_data_temp),
86.         .dina(rdata),
87.         .clkb(clk),
88.         .enb(1'b1),
```

```
89.         .addrb(i == 0 ?  cpu_addr[11:2] : reg_addr[11:2]),
90.         .doutb(out_data1)
91.     );
92.
93.     blk_mem_gen_0 data_2(
94.         .clka(clk),
95.         .wea(w_tag_2),
96.//         .addra(write_addr),
97.         .addra({reg_addr[11:5], 3'd0} + i),
98.//         .dina(write_data_temp),
99.         .dina(rdata),
100.         .clkb(clk),
101.         .enb(1'b1),
102.         .addrb(i == 0 ?  cpu_addr[11:2] : reg_addr[11:2]),
103.         .doutb(out_data2)
104.     );
105.
106.     // 这个块在 i 改变时触发，判断 temp_tr 和 i 的值
107.     // 当 temp_tr 为 3'b111 且 i 为 7 时，将数据返回给 CPU
108.     // 否则，根据 w_tag_1 和 w_tag_2 的值，返回对应的数据
109.     always@(i) begin
110.         if (temp_tr == 3'b111 && i == 7)  begin
111.                 assign cache_rdata = rdata;
112.         end
113.         else begin
114.             if (w_tag_1) begin
115.                 assign cache_rdata = out_data1;
116.             end
117.             else begin
118.                 assign cache_rdata = out_data2;
119.             end
120.         end
121.     end
122.
123.     always@(hit, i, load, miss, posedge clk) begin
124.         // 当命中时，且 i 为 7 或 0 时，并且没有加载时，设置
    cache_data_ok 和 cache_addr_ok 为 1
125.         if(hit && (i == 7 || i == 0) && load == 1'b0) begin
```

```
126.             // 如果 data_ok 为 1，说明数据已经准备好，设置
   cache_data_ok 为 1
127.             if (data_ok) begin
128.                 cache_data_ok = 1'b1;
129.                 cache_addr_ok = 1'b1;
130.         end
131.            // 否则，设置 data_ok 为 1
132.            data_ok = 1'b1;
133.         end
134.         // 若 i=7 且未命中（hit=0），也设
   置 cache_data_ok 和 cache_addr_ok 为 1。
135.         if (i == 7 && hit == 1'b0) begin
136.             cache_data_ok = 1'b1;
137.             cache_addr_ok = 1'b1;
138.//             i = 0;
139.         end
140.         // 当加载状态 load=1 时，
   cache_data_ok 和 cache_addr_ok 置为 0
141.         else if (load == 1'b1) begin
142.             cache_data_ok <= 1'b0;
143.             cache_addr_ok <= 1'b0;
144.         end
145.         // 若未命中且复位信号失效，也
   将 cache_data_ok 和 cache_addr_ok 设置为 0
146.         else if (hit == 1'b0 && resetn == 1'b1) begin
147.             cache_data_ok = 1'b0;
148.             cache_addr_ok = 1'b0;
149.         end
150.     end
151.
152.
153.     always@(posedge clk, i) begin
154.     // 如果复位信号有效，将所有信号置为 0
155.     if (!resetn) begin
156.             cache_data_ok = 1'b0;
157.             cache_addr_ok = 1'b1;
158.             arvalid = 1'b0;
159.             rready = 1'b0;
160.             i = 0;
```

```
161.             w_tag_1 = 1'b0;
162.             w_tag_2 = 1'b0;
163.             miss = 1'b0;
164.             data_ok = 1'b0;
165.             load = 1'b0;
166.             switch = 0;
167.//             rlast_temp = 1'b0;
168.         end
169.         // 否则
170.     else begin
171.         // 如果 i=7，将写使能信号置为 0，重置 i
172.         if (i == 7) begin
173.              // 关闭写使能
174.              w_tag_1 <= 1'b0;
175.              w_tag_2 <= 1'b0;
176.              reset_i <= 1'b1;
177.              i = 0;
178.         end
179.         // 如果 CPU 请求信号有效，将 CPU 地址写入 reg_addr
180.         if (cache_addr_ok) begin
181.             reg_tag<=cpu_addr[31:12];
182.             reg_index<=cpu_addr[11:5];
183.             reg_offset<=cpu_addr[4:0];
184.             reg_addr<=cpu_addr;
185.         end
186.         // hit 命中信号为两个 tag 表的 hit 信号的或
187.         assign hit = hit_1 | hit_2;
188.         hit_temp <= hit_1; // 保存上一周期的 hit 数据
189.         // 如果命中，且 i 为 7 或 0，且未加载，则设置 miss 为 0
190.         if (hit == 1'b1 && (i == 7 || i == 0) && load == 1'b0
    ) begin  // 保证装填完成后再执行
191.         miss <= 1'b0;
192.             // 准备将要返回的数据
193.             if (hit_temp) begin
194.                 assign cache_rdata = out_data1;
195.             end
196.             else begin
197.                 assign cache_rdata = out_data2;
198.             end
```

```
199.          end
200.          //否则，要阻塞 cache
201.          else begin
202.              // 阻塞 cache
203.
204.              cache_data_ok <= 1'b0;
205.              cache_addr_ok <= 1'b0;
206.              miss <= 1'b1;
207.              load <= 1'b1;
208.              data_ok = 1'b0;
209.
210.              if (switch % 2) begin
211.              // 打开写使能
212.              w_tag_1 <= 1'b1;
213.              w_tag_2 <= 1'b1;
214.              end
215.
216.              // 设置握手信号
217.
218.              //地址握手阶段，如下图所示，Cache 向主存送出欲读数据的
    地址（araddr），
219.              //以及一些其它信息（arid，在本实验中置 0 即可），通过拉
    高 arvalid 表示读请求。
220.              //若某个时钟上升沿时，arvalid 和 arready 同时为高，则握
    手成功，主存接受了地址，开始准备数据。
221.              //此时 Cache 必须拉低 arvalid 信号，否则可能会重复握手导
    致错误
222.
223.              //数据握手阶段，如下图所示，当 Cache 准备好接收数据字时，
    将 rready 拉高，
224.              //当主存向 Cache 通过 rdata 传送一个数据字时，将 rvalid
    拉高。
225.              //若某个时钟上升沿时，rvalid 和 rready 同时拉高，则握手
    成功，Cache 成功接收了一个数据字。
226.              //在本实验中，一次 AXI 传输固定为 8 个数据字，也就是说需
    要握手 8 次（一般是连续的 8 个周期，但也不一定），
227.              //在传送最后一个数据字时，主存会将 rlast 拉高，表示这是
    最后一个。
228.              arid <= 3'b000;
```

```
            assign araddr = {reg_addr[31:5], 5'd0};
            arvalid <= 1'b1;
            if (arready == 1'b0) begin
                rready <= 1'b1;
                arvalid <= 1'b0;
                if (rvalid) begin
//                write_data_temp <= rdata;
//                rlast_temp <= rlast;
                  if (!rlast) begin
                        // 更新将要写入的 addr
                        write_addr <= {cpu_addr[11:5], 3'd0
} + i;
                      i = i+1;
                  end
                  else begin
                      rready = 1'b0;
                  end

                end
            end
        end
    end
    end
    // rlast 是一个信号，通常在总线协议（如 AXI 协议）中表示“读数据
传输的最后一拍”。
    //也就是说，当 rlast 为高电平时，表示数据传输的最后一批数据已经
到达
    //当 rlast 触发时，i 被设置为 7，这与之前的代码保持一致，
    //即在 i == 7 的时候数据已准备好，可以从 rdata 中读取
    //load 被置为 1'b0，表明缓存操作已完成，不再需要加载更多数据
    always@(rlast) begin
        if (rlast) begin
            i = 7;
            load = 1'b0;
        end
    end
    //cache_data_ok 表示缓存数据是否准备好
    //在 hit == 1'b0 和 i != 7 的条件下，temp_tr 会被设置
为 cpu_addr[4:2] 的值
```

```
    //当缓存未命中且 i 还未达到结束状态时，temp_tr 会被设置为 cpu_addr[4:2] 的值，用于后续的加载操作
    always@(cache_data_ok) begin
        if(hit == 1'b0 && i != 7) begin
            temp_tr = cpu_addr[4:2];
        end
    end
    //每个时钟周期 switch 增加 1，用于周期性地改变一些状态，可能控制缓存数据的切换或刷新
    always@(posedge clk) begin
        switch = switch + 1;
    end


endmodule
```

```
// icache_tagv_table.v
/* 该表仅缓存了一路的 tag，对于两路的设计需要例化两个该表 */
module icache_tagv_table(
    input           clk,        //时钟信号
    input           resetn,     //低有效复位信号

    //写端口（Cache miss 加载新 Cache 行后）
    input           wen,        //写使能
    input           valid_wdata,//写入有效位的值（一般为 1）
    input[19:0]     tag_wdata,  //写入 tag 的值
    input[6:0]      windex,     //写入 Cache 行对应的 index

    //读端口（Cache 第一级流水段提出请求）
    input           rden,       //读使能
    input[31:0]     cpu_addr,   //来自 CPU 的地址
    output hit                  //命中结果，在上升沿后输出
    );

    reg[127:0]      valids;     //有效位，实现为 128 位寄存器
    reg[19:0]       tags[0:127];//每个 Cache 行的 tag

    genvar i;
    // 初始化模块
```

```
    // 使用 generate 和 initial 块来初始化 tags 数组，将所有标签的初始值设为 0。
    //由于复位时有效位已经全部清 0，不需要再清除标签
    generate
        for(i=0; i < 128; i = i + 1) begin
            initial begin
                tags[i]=0;      //将 tag ram 在初始时清 0
                //不需在复位时对 tag ram 清 0,因为复位时已将有效位清 0 了
            end
        end
    endgenerate

    //写入处理
    // 在时钟上升沿进行写入操作，
    //如果 resetn 为低电平，将 valids 全部清零（所有缓存行设置为无效）
    //如果 wen 为高电平（写使能有效），则根据 windex 将 valids 中对应的有效位设置为 valid_wdata，
    //并更新 tags 数组中的对应缓存行标签值为 tag_wdata
    always@(posedge clk) begin
        if(~resetn)
            valids <= 0;        //复位时将所有 Cache 行置为无效
        else begin
            if(wen) begin
                valids[windex] <= valid_wdata;
                tags[windex] <= tag_wdata;
            end
        end
    end

    //一二级流水段间的中间寄存器
    reg[19:0]       reg_tag;    //一级流水段读出的 tag
    reg             reg_valid;  //一级流水段读出的有效位
    reg[31:0]       reg_cpu_addr;   //缓存的 CPU 地址

    wire[6:0]       cpu_index=cpu_addr[11:5];   //CPU 地址的 index，作为读地址（第一级流水）
```

```
57.     wire[19:0]      r_cpu_tag=reg_cpu_addr[31:12];    //CPU地址的tag，用于选路（第二级流水）
58.
59.     //读处理
60.     //在时钟上升沿，如果 resetn 为低电平，
61.     //将寄存器 reg_tag、reg_valid 和 reg_cpu_addr 清零
62.     //如果 rden 有效，则将一级流水的输出结果（tags[cpu_index]、valids[cpu_index]、cpu_addr）
63.     //分别存入 reg_tag、reg_valid 和 reg_cpu_addr
64.     always@(posedge clk) begin
65.         if(~resetn) begin
66.             reg_tag <= 0;
67.             reg_valid <= 0;
68.             reg_cpu_addr <= 0;
69.         end
70.         else begin
71.             if(rden) begin
72.                 reg_tag <= tags[cpu_index];
73.                 reg_valid <= valids[cpu_index];
74.                 reg_cpu_addr <= cpu_addr;
75.             end
76.         end
77.     end
78.
79.     //在第二级流水段根据读出的tag判断命中
80.     //在二级流水中，通过比较 r_cpu_tag（CPU地址标签）和 reg_tag（缓存行标签）来确定是否命中。
81.     //当 r_cpu_tag == reg_tag 且 reg_valid 为1时，hit 输出为高电平，表示命中，
82.     //否则未命中
83.     assign hit=(r_cpu_tag == reg_tag) & reg_valid;
84.
85. endmodule
```

五、测试结果及实验分析

测试结果：

图 3-1 测试波形图

```
Your cache is correct! In total 100011 requests:
Total 561 misses (miss rate: 0.56%)
Total 105558 cycles (1.05 cycles per request)

PASS! You have done well!
--------------------
$finish called at time : 1158215 ns : File "E:/vivado_project/tixijiegou-lab3/lab3/lab3.srcs/sources_1/new/cache_checker.v" Line 79
run: Time (s): cpu = 00:00:01 ; elapsed = 00:00:39 . Memory (MB): peak = 1700.488 ; gain = 2.871
```

图 3-2 测试终端结果

自动化测试是否通过	不通过的 Miss rate
是	0.56%

六、实验总结

通过本次实验，通过完成二/四路组相连指令 Cache，我充分掌握 Vivado 集成开发环境，掌握 Verilog 语言，掌握 FPGA 编程方法及硬件调试手段，深刻理解动态分支预测的原理与方法。我不仅深化了对指令 Cache 技术的理解，还获得了许多硬件设计与调试的实际经验。特别是在实现两级流水的 Cache 设计时，我学会了如何高效地管理数据存储、如何设计状态机实现复杂的协议（如 AXI 接口），并有效地解决了 Cache miss 和替换策略问题。

通过实验表明，针对二/四路组相联的设计，指令 Cache 在提高 CPU 执行效率中发挥的重要作用。通过 LRU 算法优化了数据存取性能，有效提高了 Cache 的命中率。