**重庆大学课程设计报告**

课程设计题目： RISC-V CPU 设计

学 院： 弘深学院

专业班级： 计算机拔尖班01班

年 级： 大二

姓 名： 胡芸明，朱云哲，郑梦仁

学 号： 20220592，20224700，20223714

完成时间： 2024 年 7 月 14 日

成 绩：

指导教师： 刘大江

重庆大学教务处制

一、设计简介

本项目设计了一个基于RISC-V32位指令集的多周期五级流水CPU。项目实现了包括五级流水线、增量开发指令、双模动态分支预测、Cache高速缓存、冒险检测处理以及优化的数据前推在内的模块。本项目设计特色在于实现了分支指令前推，多种数据冒险的处理与解决，双模动态分支预测以及Cache。此外项目采用模块化编写，组件复用性高，方便功能扩展与部件升级。

CPU功能包括：【1】运行时可以以流水执行指令，并在取值阶段预测下一条指令，提升了流水线执行效率。【2】搭载的冒险检测处理模块与优化的数据前推模块可以监测并解决多种数据冒险与控制冒险，保障CPU的功能正确执行。【3】实现了Cache高速缓存，有效减少了指令频繁读取内存的时间开销，保障了流水线正常运行，提升了CPU性能。

本项目全部采用verilog语言实现，调用了vivado IP单端口BRAM作为主存。双模动态分支预测采用状态机实现，cache采用直接映射结构，并采取写返回和写分配策略实现。

本项目CPU运行实验前五个基础测试指令集的结果均正确。两个综合测试的代码仿真执行顺利。其中综合测试2的结果与网站仿真结果一致；综合测试1的寄存器前期行为变化与指令执行顺序均符合预期，但是后期出现死循环且无法解决，结果无法对比。

（一）小组分工说明

朱云哲：负责流水线整体架构搭建，指令增量开发，冒险检测模块的完善优化，测试指令集的仿真与调试。

郑梦仁：负责双模动态分支预测实现，数据前推模块的完善优化，动态分支预测的功能仿真与调试。

胡芸明：负责Cache设计与实现，Cache写回策略的实现，Cache与BRAM的通信，Cache模 块的功能仿真与调试。

**（一）项目实施计划**

**【1】**6.26-6.30完成指令集增量开发，实现全部要求的指令

**【2】**7.1-7.3 完成冒险检测与数据前推模块优化，运行基础测试指令集。

**【3】**7.4-7.8 完成双模动态分支预测并进行仿真测试，同时继续优化冒险检测模块，处理控制冒险。运行综合测试指令集。

**【4】**7.8-7.10 完成Cache的设计与实现，并将Cache和BRAM集成至CPU核实现数据通信，继续运行测试指令集。

**【5】**7.11-7.14完成项目，编写实验报告。

**（二）项目实施节点（里程碑时间）**

**【1】**7.2 运行基础测试指令集，仿真全部通过。

**【2】**7.7 运行基础测试指令集与综合测试，分支预测功能正常，仿真通过。

**【3】**7.9 运行基础测试指令集与综合测试，cache读取写回数据功能正常，仿真通过。

**【4】**7.10 项目基本完成，继续优化。

**【5】**7.14 项目完成，实验报告编写完成。

二、设计方案（30%）

（一）总体设计思路

1.1处理器发展趋势与技术前沿

随着现代社会对计算能力需求的不断增加，处理器的发展之路也越来越“难”。从早期的打表计算，到单周期处理器，流水线处理器，再到超标量处理器（Superscalar Processor），处理器的功能越来越多，性能越来越强，但同时对材料和架构也提出了更高的要求。作为高性能处理器架构的核心，通过多发射（Multiple Issue）和分支预测（Branch Prediction）等技术，超标量处理器显著提高了指令级并行性（Instruction-Level Parallelism, ILP），从而提升了处理器的性能。面对大数据处理、人工智能和高性能计算等领域日益增长的算力需求，

这些技术也在不断演进和优化，以满足更高的性能要求。

超标量处理器是一种能够在单个时钟周期内同时执行多条指令的处理器架构。多发射技术使处理器在同一时钟周期内发射多条指令进行执行，根据 Hennessy 和 Patterson 在《计算机体系结构：量化研究方法》中的描述，多发射可分为静态和动态两种，主要挑战包括资源冲突、数据相关性和控制相关性带来的限制。分支预测用于预测程序中分支指令的执行路径，以减少分支指令造成的流水线停顿。Yeh 和 Patt 的研究指出，常见的分支预测方法包括静态预测和动态预测，动态预测通常使用历史信息和模式检测来提高预测的准确性。

此外，数据作为计算的基础，需要有足够的存储空间来保证计算的连续性。缓存（Cache）是一种高速存储器，位于处理器和主存之间，用于临时存储常用的数据和指令，以缩短访问时间和提高系统性能。缓存的工作原理基于局部性原理（locality principle），包括时间局部性和空间局部性。缓存通常分为三级：一级缓存（L1 Cache）、二级缓存（L2 Cache）和三级缓存（L3 Cache），每级缓存容量逐级增大，访问速度逐级减慢。在多核处理器系统中，缓存一致性（Cache Coherence）协议用于确保多个处理器核心在各自缓存中看到的一致数据，常见的一致性协议包括 MESI（Modified, Exclusive, Shared, Invalid）协议及其变种。

但是单块处理器的性能无论如何强悍，也是有一个无论如何也无法突破的算力瓶颈。为了解决种种诸如此类的问题，多核和众核架构开始称为主流，异构计算也崭露头角。此外，通过研发针对某特定功能的芯片，专门加速特定一类计算任务也是处理器的未来新赛道。

1.2处理器设计因素分析

在当今处理器设计中，考虑到行业规范、安全需求和功耗要求是至关重要的方面。处理器作为现代计算机系统的核心组件，其设计不仅影响着计算性能和能效，还直接关系到系统的安全性和可靠性。

**1.2.1行业规范**

首先，处理器设计必须符合多种行业规范，这些规范旨在确保硬件的互操作性、可靠性和安全性。例如，计算机行业的标准如PCI Express、USB、以太网等，规定了硬件接口和通信协议，保证了不同设备之间的兼容性和通信的稳定性。此外，处理器设计还需遵循电磁兼容性（EMC）和电磁干扰（EMI）标准，以减少设备对外部电磁环境的干扰，确保设备在各种工作环境下的正常运行。

在新兴技术领域，如物联网（IoT）和自动驾驶汽车，处理器设计必须符合特定的行业安全和通信协议标准。这些标准不仅关乎设备的安全性，还涉及到个人隐私保护和数据安全。

**1.2.2安全需求**

随着信息技术的广泛应用，处理器的安全性越来越受到关注。处理器设计必须考虑到硬件级别的安全保护措施，防范物理攻击、侧信道攻击和逻辑攻击等威胁。硬件加密引擎、安全引导和可信执行环境（TEE）等技术被广泛应用于现代处理器中，以确保数据的安全性和系统的完整性。

处理器还需支持安全启动、固件更新和远程认证等功能，以应对软件层面的安全挑战。例如，处理器设计需要与操作系统和应用程序配合，实现硬件级别的安全控制和管理，保护用户数据不被未经授权的访问。

### 1.2.3功耗要求

随着移动设备和数据中心的普及，功耗成为处理器设计的重要考量因素。低功耗设计可以延长电池寿命，减少设备散热需求，并降低数据中心的能源消耗和运营成本。处理器设计中的功耗优化包括优化电源管理、动态电压调整（DVFS）、优化指令集架构（ISA）和集成低功耗组件等措施。

此外，功耗和性能之间存在着经济上的权衡。处理器设计师需要在提高性能的同时，尽可能降低功耗，以满足市场需求和环境保护的要求。新一代制程技术（如7nm、5nm工艺）的引入，对提升处理器性能和降低功耗提供了新的技术路径。

综上所述，处理器设计时需要综合考虑行业规范、安全需求和功耗要求等多方面因素。在面对日益复杂的应用场景和技术挑战时，处理器设计师必须在性能、安全性和能效之间找到平衡点，以满足不断增长的市场需求和用户期望。未来，随着技术的进步和应用领域的扩展，处理器设计将继续面临新的挑战和机遇，推动数字化社会的发展和智能化进程。

1.3本项目设计规划

**1.3.1主要设计内容**

本项目主要解决关键三个技术问题：【1】指令的增量开发【2】动态分支预测的实现【3】Cache的实现。为了解决以上要求，我们重新设计了控制器（controller），扩充了跳转指令和访存指令的控制通路核数据通路，完成了双模动态分支预测器的设计，完成了支持写返回和写分配策略的Cache的设计。此外，我们还优化了冒险检测处理和数据前推模块。

本项目CPU分为五阶段：取值阶段（IF），译码阶段（ID），执行阶段（EX），访存阶段（MEM），写回阶段（WB）。在IF阶段，指令寄存器i-Cache接收pc生成的取值信号输出指令，同时分支预测模块接收pc信号预测下一条指令的地址，随后pc信号，取出的指令instr将会保存至IF/ID流水线寄存器；在ID阶段，取出的指令被送往控制器、通用寄存器堆和其他算术模块。控制器负责译码出各种控制信号，寄存器堆从指令指示的地址重取出数据，算术模块用于移位操作。同时寄存器堆取出的数据将会被compare模块进行比较，结果传回控制器用于决定分支指令的跳转方向，完成分支指令的前推。这时ID/EX流水线寄存器将会保存此阶段的数据信号和控制信号；在EX阶段，首先经过控制信号和前推信号选择出操作数，然后送入ALU中，ALU通过译码出的计算指令进行相应的计算得出结果，并存入EX/MEM流水线寄存器，如果是sw指令，还需要将选择后的存储数值也一并存入；在MEM阶段，对于非访存指令直接将结果存入MEM/WB流水线寄存器，如果是访存指令，对于sw/ld指令，如果Cache发生miss，则信号会传入冒险检测模块进行处理，反之正常执行数据取出与替换。取出的数值将会进入data\_processing模块中，根据functld指示的访存类型进行相应处理，最后存入MEM/WB寄存器中；最后在WB阶段，多路选择器将选择写回的数据方向，将数值写入到指定的寄存器堆地址。综上，指令的全过程执行完毕。以下为本项目结构图，图2-1是结构总览；图2-2是CPU核内细节；图2-3是data\_ram存储模块内部架构：

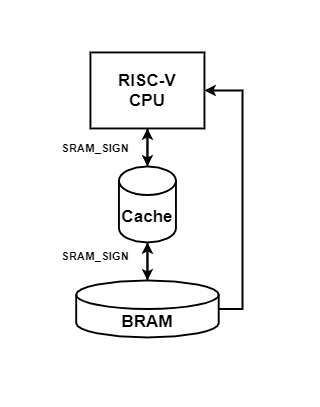


图2-1 系统结构总览

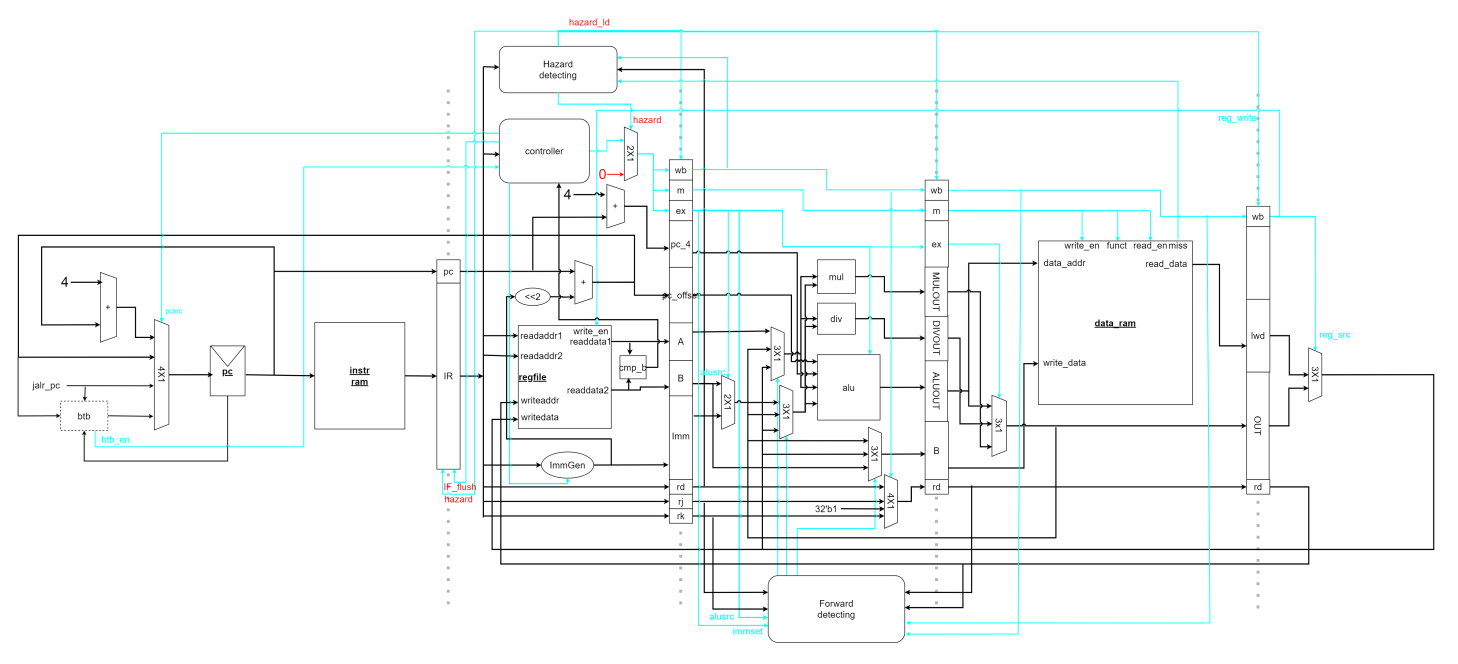


图2-2 RISC-V核内架构

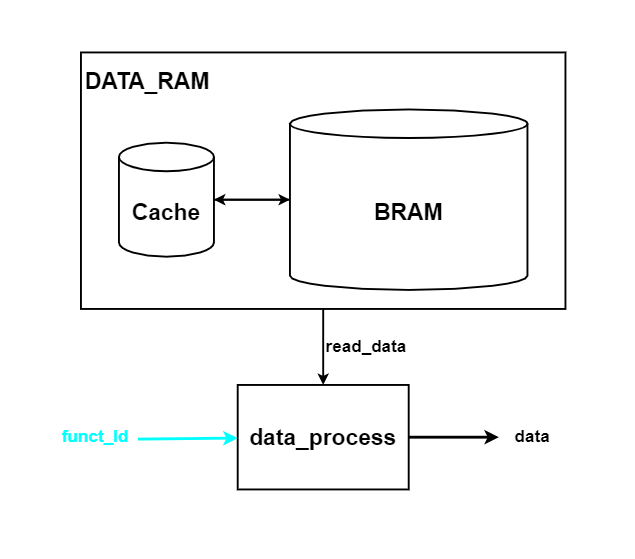


图2-3 data\_ram内部架构

1. Controller模块设计

Controller模块是整个CPU运行的核心部件之一，其核心功能就是对传输进来的指令进行译码，获取各种控制信号，同时还负责分析分支指令的比较结果然后选择pc计数器的方向，此外还负责判断分支错误进行指令冲刷。

Controller模块包含两个附属模块：main\_control和alu\_control。前者主要针对Controller译码出的各种指令进行具体的译码，获得包括分支指令标志、立即数扩展方式、ALU运算指示、访存使能、写回使能和写回方向等控制信号。后者主要对main\_control的ALU运算指示进行更进一步分类，得到具体的运行标识。模块展示如下，图2-4为controller模块；图2-5为main-control模块；图2-6为alu\_control模块：

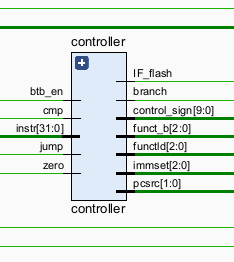
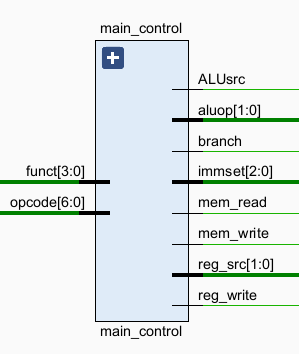
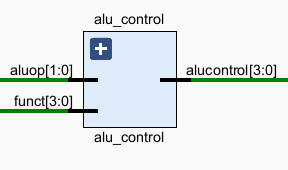
  

图2-4 controller模块 图2-5 main\_control模块 图2-6 alu\_control模块

下面解释各模块解析出的信号含义，表2-1为Controller信号表；表2-2为main-control信号表；表2-3为alu\_control信号表。

表2-1 Controller信号表

|  |  |  |  |
| --- | --- | --- | --- |
| 信号(Controller) | 含义 | | 输出/输入 |
| btb\_en | 分支预测使能信号 | | 输入 |
| instr[31:0] | 指令 | | 输入 |
| jump | 分支预测预测跳转信号 | | 输入 |
| cmp | 分支指令条件判断结果 | | 输入 |
| zero | 空信号，预留接口 | | 输出 |
| branch | 分支指令指示码 | | 输出 |
| immset[2:0] | 立即数扩展方式操作码 | | 输出 |
| pcsrc[1:0] | pc方向选择 | | 输出 |
| funct\_b[3:0] | 分支指令分类码 | | 输出 |
| functld[2:0] | | ld类指令分类码 | 输出 |
| IF\_flash | | 指令冲刷信号 | 输出 |
| control\_sign[9:0] | | 包含reg\_write(寄存器堆写使能)，reg\_src(WB阶段寄存器堆写回数据方向)，mem\_read(Cache/Bram读使能)，mem\_write(Cache/Bram写使能)，alucontrol(ALU操作码)，ALUsrc(ALU操作数方向选择) | 输出 |

表2-2 main\_control信号表

|  |  |  |
| --- | --- | --- |
| 信号(main\_control) | 含义 | 输出/输入 |
| funct[3:0] | funct3与funct7字段结合的指令标识码 | 输入 |
| opcode[6:0] | 指令低七位标识码 | 输入 |

表2-2 main\_control信号表（续）

|  |  |  |
| --- | --- | --- |
| 信号(main\_control) | 含义 | 输出/输入 |
| ALUsr | ALU操作数方向选择 | 输出 |
| aluop[1:0] | ALU操作分类指令 | 输出 |
| branch | 分支指令指示码 | 输出 |
| immset[2:0] | 立即数扩展方式操作码 | 输出 |
| mem\_read | Cache/Bram读使能 | 输出 |
| mem\_write | Cache/Bram写使能 | 输出 |
| reg\_src[1:0] | WB阶段寄存器堆写回数据方向 | 输出 |
| reg\_write | 寄存器堆写使能 | 输出 |

表2-3 alu\_control信号表

|  |  |  |
| --- | --- | --- |
| 信号(alu\_control) | 含义 | 输出/输入 |
| aluop[1:0] | ALU操作分类指令 | 输入 |
| funct[3:0] | funct3与funct7字段结合的指令标识码 | 输入 |
| alucontrol[3:0] | ALU操作码 | 输出 |

以上模块均为组合逻辑实现，其数据通路与控制逻辑如下：

【1】Controller通过解析instr的opcode获取指令类型，每个指令类型解析出四种特征信号（funct、functld、dunct\_b、jflag）。

【2】特征信号传入main\_control与alu\_control，解析出每条指令具体控制信号。

【3】Controller通过分支预测使能信号、分支条件计算结果解析出pc计数器方向与冲刷信号并传出。

【4】各功能部件接收控制信号执行相应动作。

指令开发过程如图2-7：

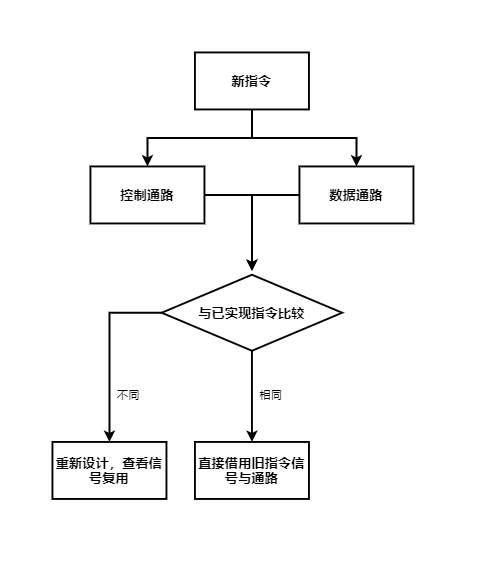


图2-7 指令增量开发流程

其中关于pcsrc和IF\_flash信号的获取如表2-4，表2-5所示，通过真值表获得信号表达式：

表2-4 pcsrc信号真值表

|  |  |  |  |
| --- | --- | --- | --- |
| pcsrc[1:0] | cmp | jflag[1:0]（判断jal与jalr） | branch |
| 00（不跳转） | 0 | 00 | 0 |
| 00（不跳转） | 0 | 00 | 1 |
| 01（jal跳转） | 0 | 10 | 1 |
| 01（B类型指令跳转） | 1 | 00 | 1 |
| 10（jalr跳转） | 0 | 01 | 1 |

表2-5 IF\_flash信号真值表

|  |  |  |  |
| --- | --- | --- | --- |
| IF\_flash | btb\_en | jump | pcsrc |
| 0 | 0 | 0 | 00 |
| 0 | 1 | 0 | 00 |
| 0 | 1 | 1 | 01 |
| 0 | 1 | 1 | 10 |
| 1 | 0 | 0 | 01 |
| 1 | 0 | 0 | 10 |
| 1 | 1 | 0 | 01 |
| 1 | 1 | 0 | 10 |
| 1 | 1 | 1 | 00 |

1. ALU模块设计

ALU模块是整个CPU的执行核心，所有指令的操作都必须经过ALU获取相应结果，在实现了分支前推后，B类型指令可以在ID阶段结束，不需要复用ALU。ALU实现包括加、减、逻辑与、逻辑或、逻辑异或、算术左移、算术右移、逻辑右移、带符号比较和旁路功能，实现了自定义指令max操作。ALU可以用于算术逻辑指令获得计算结果，也可以用于访存指令获取内存地址，其模块如图2-8，信号接口如表2-6。

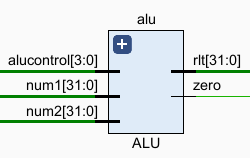


图2-8 ALU模块

表2-6 ALU信号表

|  |  |  |
| --- | --- | --- |
| 信号(ALU) | 含义 | 输出/输入 |
| alucontrol[3:0] | ALU计算行为操作码 | 输入 |
| num1[31:0] | ALU操作数1 | 输入 |
| num2[31:0] | ALU操作数2 | 输入 |
| rlt[31:0] | ALU计算结果 | 输出 |
| zero | 空信号 | 输出 |

ALU数据通路与控制通路如下：

【1】ALU接收ID/EX流水线寄存器储存的ALU操作码，进入到对应的计算模块中。

【2】ALU得到经过选择后的两个32位操作数进行相应计算。

【3】ALU将计算结果储存至EX/MEM流水线寄存器，完成EX阶段。

（四）Hazard-Detecting模块设计

冒险检测模块用于处理数据冒险，负责检测并处理ld-use冒险与Cache-miss冒险，保证流水线的正常运行。其工作原理如下：首先在EX阶段，可以判断first-ld-use冒险，即第一条为ld指令，第二条为冒险指令。Hazard信号表达式为：

hazard=(mem\_readE&&((rd\_out==instr\_out[19:15])||(rd\_out==instr\_out[24:20])))?1'b1:1'b0;

判断冒险后，模块生成hazard信号，流水线暂停一个时钟周期并保持pc与IF/ID寄存器中的信号。如果发生second-ld-use，可以直接通过前推模块解决。

如果访存时发生Cache-miss，检测模块将生成hazard\_ld冒险信号，流水线暂停一个时钟周期，防止发生同时多个数据写回寄存器堆发生结构冒险。并同时检测是否存在first-ld-use冒险。hazard\_ld信号表达式为：

hazard\_ld=(!cache\_hit)?((mem\_readM&&((rd\_outM==instr\_out[19:15])||(rd\_outM==

instr\_out[24:20])))?1'b1:((!cache\_hit&&mem\_readM)?1'b1:1'b0)):1'b0;

最后，冒险检测模块如图2-9，信号表如表2-7所示。

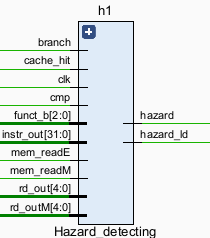


图2-9 Hazard\_detecting模块

表2-7 Hazard\_detecting部分信号表

|  |  |  |
| --- | --- | --- |
| 信号(Hazard\_detecting) | 含义 | 输出/输入 |
| Cache\_hit | Cache命中信号 | 输入 |
| instr\_out[31:0] | IF/ID寄存器保存的指令 | 输入 |
| mem\_readE | ID/EX寄存器保存的Cache/BRAM读使能 | 输入 |
| mem\_readM | EX/MEM寄存器保存的Cache/BRAM读使能 | 输入 |
| rd\_out[4:0] | ID/EX寄存器保存的目标寄存器地址 | 输入 |
| rd\_outM[4:0] | EX/MEM寄存器保存的目标寄存器地址 | 输入 |
| hazard | EX阶段判断数据冒险暂停信号 | 输出 |
| hazard\_ld | MEM阶段判断Cache-miss冒险信号 | 输出 |

（五）Forward-Detecting模块设计

前推模块是处理数据冒险的重要环节，功能强大且逻辑合理的前推模块可以高效解决各种数据冒险，保证流水线的执行效率，减少停顿带来的性能损失。本项目的前推模块为实验四基础上优化后的完全前推模块，实现了包括普通数据冒险(ld-use类)，双重数据冒险，立即数伪数据冒险和ld-sw数据冒险的判断与处理。

在前推信号的组合上，我们采用了拼凑的方式，这样可以解决MEM和WB两个阶段的数据前推，并判断双重数据冒险。模块的控制信号表达式如下：

always@(\*)begin

f0=reg\_writeM&&(rd\_outM!=0)&&(rd\_outM==rs1\_out);

f1=reg\_writeW&&(rd\_outW!=0)&&(rd\_outW==rs1\_out);

f2=reg\_writeM&&(rd\_outM!=0)&&(rd\_outM==rs2\_out)&&(ALUsrcE==1'b0);

f3=reg\_writeW&&(rd\_outW!=0)&&(rd\_outW==rs2\_out)&&(ALUsrcE==1'b0);

f4=reg\_writeM&&mem\_writeE&&(rd\_outM==rs2\_out)&&(rd\_outM!=0);

end

assign forwarda=((f0&f1)!=1'b1)?{f1,f0}:{1'b0,f0};

assign forwardb=((f2&f3)!=1'b1)?{f3,f2}:{1'b0,f2};

assign forwardc=f4;

其中，f0，f1信号负责ALU操作数num1的前推；f2，f3负责ALU操作数num2的前推；f4负责sw-ld数据冒险下的前推。当发生双重数据冒险时，必然可以发现MEM和WB阶段的结果均满足条件，也就是f0&f1或者f2&f3均为1，此时直接选取最近产生的数据即MEM阶段数据，修改高位信号为0，强制选择MEM阶段数据前推，解决双重数据冒险。f4信号只需要判断ld写回地址与sw读取寄存器堆的地址是否相同即可实现前推。此外，为了防止立即数类型指令误判，模块增加ALUsrcE信号。当操作数num2必须是从寄存器读出的时候才进行前推。否则是I类型指令不前推。模块如图2-10所示，信号表如表2-8所示。

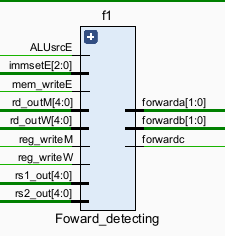


图2-10 Forward\_detecting模块

表2-8 Forward\_detecting模块

|  |  |  |  |
| --- | --- | --- | --- |
| 信号(Forward\_detecting) | | 含义 | 输出/输入 |
| ALUsrcE | | ID/EX寄存器保存的num2操作数方向信号 | 输入 |
| mem\_writeE | | ID/EX寄存器保存的Cache/BRAM写使能 | 输入 |
| rd\_outM[4:0] | | EX/MEM寄存器保存的目标寄存器地址 | 输入 |
| rd\_outW[4:0] | | MEM/WB寄存器保存的目标寄存器地址 | 输入 |
| rs1\_out[4:0] | | ID/EX寄存器保存的源寄存器1地址 | 输入 |
| rs2\_out[4:0] | ID/EX寄存器保存的源寄存器2地址 | | 输入 |
| reg\_writeM | EX/MEM寄存器保存的寄存器堆写使能 | | 输入 |
| reg\_writeW | MEM/WB寄存器保存的寄存器堆写使能 | | 输入 |

表2-8 Forward\_detecting模块（续）

|  |  |  |
| --- | --- | --- |
| 信号(Forward\_detecting) | 含义 | 输出/输入 |
| forwarda[1:0] | ALU操作数num1的前推方向信号 | 输出 |
| forwardb[1:0] | ALU操作数num2的前推方向信号 | 输出 |
| forwardc[1:0] | sw指令目标存储数据的前推方向信号 | 输出 |

（六）动态分支预测模块设计

分支预测模块采用双模预测方法，在PC更新阶段即可以预测下一次的指令地址，初始状态下默认是顺序执行结构，根据实际的跳转情况对缓冲区存储的指令进行更新，并且将双模状态机的状态进行更新。一旦发生预测错误，数据通路datapath中产生相应的错误信号，传入PC更新过程，并且同时充当Flush信号对相应的模块进行更新。

预测器初始化流程如下，状态更新表如图2-11**：**

**指令缓冲存储器（buffer）**存储的是相应PC（指令地址）对应下一条指令的地址（npc）。

**状态机存储器（state）**存储每一个PC对应的跳转状态（双模态共有4个状态），初始化为2’b00,即默认不跳转（顺序执行）。

**指令存储器（Inst\_ram）**存储初始相应的PC地址对应的指令（instr）。

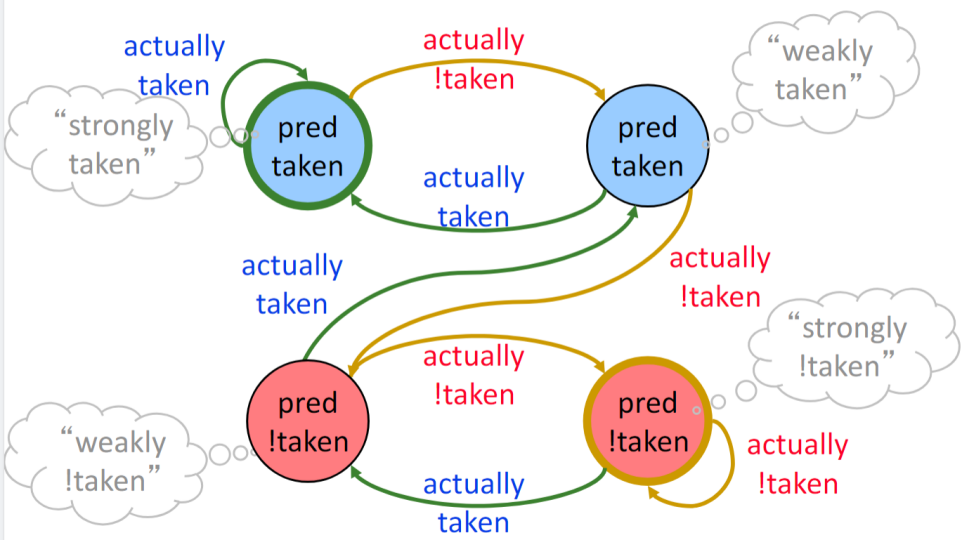


图2-11 双模状态机示意图

根据当前PC值对应的历史状态以及当前该PC对应的实际的跳转情况，对相应的状态值进行更新（其中00表示strongly !taken, 01表示weakly !taken, 10表示weakly taken, 11表示strongly taken）。

对于每个特定的PC预测情况，由于我们需要得到其PC的数据以及相应的跳转情况和预测成功与否的结果，而预测结果是在后续的流水线阶段才能够得到相应的结果，因此输入信号中包括流水线传递的PC值以及相应的预测正确与否的信号，正确的执行地址（dest）。其主要数据行为与控制逻辑如下：

**1.预测成功：更新pc, npc, instr，同时更新状态机**，如图2-12：

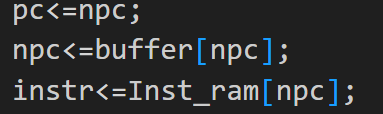


图2-12 更新代码示例

【1】发生跳转时，当状态值为2’b10时，更新到强跳转2’b11

【2】 顺序执行时，当状态值为2’b01时，更新到强不跳转2’b00.

**2.预测失败：更新pc, npc, instr利用正确目标地址dest，同时更新状态值以以及预测表：**，如图2-13：

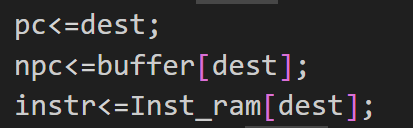


图2-13 更新代码示例2

【1】发生跳转（jump）：

当状态机值为00，更新为01，预测表不更新。

状态值为01时更新为10，并且将预测表进行更新，将下一条指令赋值成dest.

【2】顺序执行(!jump)：

状态值为10时更新为01,预测表不更新。

当状态值为11时，更新为10，将预测表更新成顺序执行的地址。

**3.额外的冒险优先级处理：**

由于存在数据冒险ld-use以及读写缺失是的访存stall的存在，会对动态分支预测模块产生影响。相应的pc, npc, instr的更新产生一定的影响。注意到，我们的分支预测的结果有可能会依赖于访存之后得到的结果，也即是依赖于历史的结果，因此，访存阶段的stall信号的优先级别是最高的，一旦访存的stall信号为高电平，整个流水线都处于停顿的状态而不考虑预测结果（预测的结果在这时甚至是不可靠的）。另外注意到，数据冒险信号hazard在IF/ID流水线寄存器更新后即可以得到相应的结果，然而我们的预测错误信号以及目标地址是在后续阶段获得的，因此错误信号（error）的优先级是大于数据冒险hazard信号的，因为一旦发生预测错误，那么对于预测对象之后发生的指令是需要被冲刷的，因此hazard是最后需要考虑的。所以hazard信号只有在error信号为false时（预测正确）才需要处理。仅需要将pc, npc, instr信号保持不变）（IF/ID寄存器暂停），ID/EX寄存器置为0。

（七）Cache模块设计

Cache 就是缓存，是计算机系统中用来提高数据访问速度的一种高效存储技术。它是一种容量较小但速度极快的存储器，通常位于CPU内部或与CPU非常接近，以便快速访问数据。Cache分为多个级别，通常包括L1（一级缓存）、L2（二级缓存）和L3（三级缓存），每一级缓存的容量逐渐增加，但速度逐渐减慢。Cache的主要作用是提高数据访问速度，减少访问主存（RAM）所需的时间。通过将频繁使用的数据和指令存储在Cache中，可以大大减少CPU等待数据的时间，从而提高整体系统性能。由于Cache位于CPU内部或非常接近CPU，数据传输的延迟较低，这有助于减少程序执行中的延迟时间。此外，Cache还可以通过减少处理数据时的瓶颈，提高处理器的整体效率，同时降低主存的负载和功耗。Cache利用局部性原理来提高数据访问效率，局部性原理包括时间局部性和空间局部性：时间局部性指近期访问过的数据或指令很可能在不久的将来再次被访问；空间局部性指与近期访问的数据或指令相邻的数据或指令很可能也会被访问。当CPU需要访问数据时，会首先检查Cache中是否有该数据。如果数据在Cache中（称为命中），则直接从Cache读取数据；如果数据不在Cache中（称为未命中），则从主存中读取数据，并将该数据加载到Cache中以备后续使用。通过这种方式，Cache大大提高了CPU的数据访问效率，优化了系统性能。

1. Cache的映射策略

Cache从理论层面来讲等价于理论内存的一个真子集。因此我们为了建立这个等价关系需要提出一些可逆的映射来将Cache和理论内存联系起来。如果我们假设传入的地址为Address[31:0]，Cache有N行，每一行的大小为M Byte，一个数据为4Byte。那么常见的映射方式有如下几种：

1.1. 直接映射（Direct Mapped Cache）

直接映射是最简单的一种缓存映射方式。在直接映射缓存中，每个内存块都映射到缓存中的一个固定位置。具体来说，内存地址通过取模操作直接映射到缓存中的某个缓存行。由于每个内存块只能映射到一个固定位置，这种方式的硬件实现相对简单，但冲突率较高，即不同内存块可能会频繁地争用同一个缓存位置。则按照假设，我们的Cache Line中的数据应该有如下计算：

1.1.1. 计算Cache Line位长

Len(Offset)=log2(M/4)

Len(Index)=log2(N)

Len(Tag)=32-Len(Offset)-Len(Index)

Len(Valid)=1

Len(Dirty)=1

1.1.2. 解码地址

Offset=Address[Len(Offset)-1:0]

Index=Address[Len(Offset)+Len(Index)-1:Len(Offset)]

Tag=Address[31:Len(Offset)+Len(Index)]

1.2. 全相联映射（Fully Associative Cache）

在全相联映射中，内存中的任何块都可以加载到缓存中的任何位置。每个缓存行都包含一个标记字段，用于存储内存块的标识。当访问内存时，缓存会搜索所有缓存行的标记字段，以确定数据是否在缓存中。由于每个内存块可以存储在缓存的任何位置，全相联映射具有最低的冲突率，但实现起来复杂且成本较高。

1.2.1. 计算Cache Line位长

Len(Offset)=log2(M/4)

Len(Tag)=32-Len(Offset)

Len(Valid)=1

Len(Dirty)=1

1.2.2. 解码地址

Offset=Address[Len(Offset)-1:0]

Tag=Address[31:Len(Offset)]

1.3. 组相联映射 （Set Associative Cache）

组相联映射是一种折中方案，结合了直接映射和全相联映射的优点。在组相联映射中，缓存被分成多个集合，每个集合包含若干缓存行。内存地址首先通过取模操作映射到一个特定的集合，然后再在该集合中进行全相联映射。常见的组相联映射包括2路、4路和8路组相联缓存。相比于直接映射，组相联映射减少了冲突率；相比于全相联映射，组相联映射的硬件实现相对简单且成本较低。假设路数为W。

1.3.1 计算Cache Line位长

Len(Offset)=log2(M/4)

Len(Set Index)=log2(N/W)

Len(Tag)=32-Len(Offset)-Len(Set Index)

Len(Valid)=1

Len(Dirty)=1

1.3.2. 解码地址

Offset=Address[Len(Offset)-1:0]

Set Index=Address[Len(Offset)+Len(Set Index)-1:Len(Offset)]

Tag=Address[31:Len(Offset)+Len(Set Index)]

三种方法的示意图如图2-14：

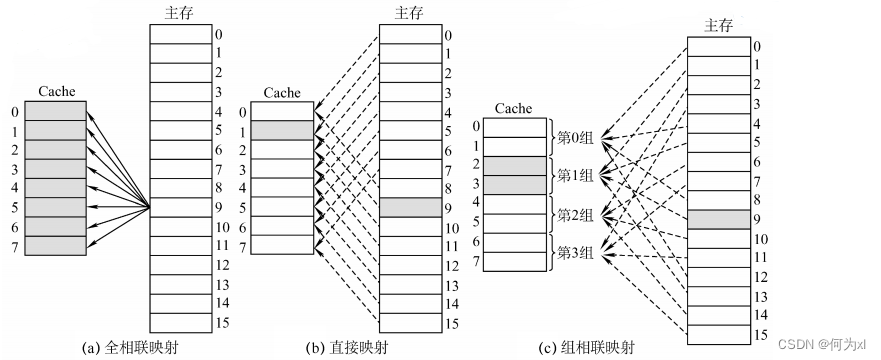


图2-14 Cache的三种映射方式

为了简单起见我们采用直接相联的方式进行映射，同时配置参数如下：

N=28

M=4

因此我们的Cache Line的形式如图2-15：

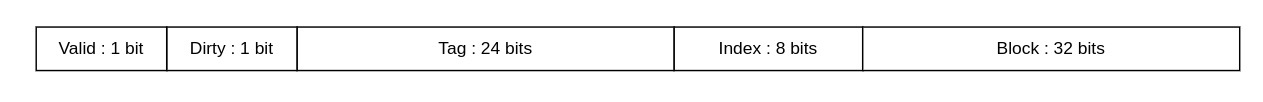


图2-15 本项目Cache的组成

在具体实现中，我们的Cache Line使用寄存器堆实现，在后面的部分会提到。

2. Cache读写策略

在了解到Cache Line的构造之后，我们需要对Cache Line的读写策略进行讨论。Cache Line的读写策略一定程度上也影响了Cache的效率，同时更重要的是它影响Cache读写数据的正确性。在这里，我们将按照写策略、替换策略和分配策略这三个方面来详细说明每种策略的具体操作和优缺点。

2.1. 写策略

写直达策略要求在每次写操作时，数据同时写入Cache和主存。即使只对Cache进行了写操作，也会立即更新到主存中，以保持Cache和主存中数据的实时同步。这种策略确保了系统中Cache和主存数据的一致性，适用于对数据完整性要求较高的应用场景，如数据库系统和事务处理系统。然而，写直达策略可能因为频繁的主存写入而增加延迟，影响系统的响应速度，并且可能导致额外的内存访问和总线带宽开销。而写回策略允许在写操作时首先将数据写入Cache中进行修改，而不立即写入主存。只有在Cache行被替换或者需要写回主存时，才会将修改的数据写回主存。这种策略减少了对主存的写访问次数，降低了内存访问和总线带宽的使用，从而提高了系统的性能和吞吐量。写回策略适合对系统性能要求较高、可以容忍一定数据不一致的应用场景，如图形处理和科学计算等。然而，实现写回策略需要额外的管理机制来处理Cache和主存数据的一致性问题，这可能增加系统设计的复杂性和实现难度。

2.2. 替换策略

2.2.1. 最近最少使用（LRU）

LRU替换策略会替换最近最少使用的Cache Line。具体操作包括使用一种算法来追踪每个Cache Line的使用历史，以确定哪些数据最久未被访问，从而进行替换。优点是通常能很好地预测数据的局部性，减少缺失率；缺点是实现复杂，需要额外的硬件支持和管理。

2.2.2. 先进先出（FIFO）

替换策略会替换最早进入Cache的Cache Line。具体操作包括使用队列来记录每个Cache Line进入Cache的顺序，当需要替换时，选择队列中最早进入的Cache Line进行替换。优点是实现简单，不需要额外的硬件支持；缺点是未能考虑数据的访问模式，可能会导致较高的缺失率。

2.3. 分配策略

2.3.1. 写分配（Write Allocate）

写分配策略要求在写操作时，先将数据加载到Cache中，然后再进行修改。这种策略可以确保Cache中的数据始终与主存中的数据保持一致，适合需要保证数据一致性和可靠性的应用场景。具体操作包括在写操作发生时，检查数据是否在Cache中，如果不在则从主存加载数据到Cache中，然后修改数据并同步到主存。

2.3.2. 写不分配（Write-Non Allocate）

写不分配策略允许在写操作时直接在主存中进行数据修改，而不需要将数据加载到Cache中。这种策略减少了内存访问和总线带宽的使用，提高了系统的响应速度和吞吐量。适用于对系统性能要求较高、可以容忍一定数据不一致的应用场景，如图形处理和科学计算等。

在本实验中，为了更好地提升效率，我们采用了WB+WA+DualPort的设计，这样的设计有以下优势：首先，写回策略（Write-Back）允许数据在写入Cache后暂时不直接写入主存，从而减少了频繁的主存访问，降低了内存访问延迟，显著提升了系统的响应速度和性能表现。其次，写分配策略（Write Allocate）确保每次写操作都加载数据到Cache中，进一步优化了写操作的效率，特别是在数据写入后的多次读取场景中尤为有效。最后，双端口（Dual Port）设计允许多个处理器或线程同时对Cache进行读写操作，这不仅提高了系统的并发处理能力，还有效地降低了竞争带来的性能瓶颈。综上所述，这种设计不仅优化了系统的性能和响应速度，还提升了其在多核处理器和并行计算环境中的适用性，为高性能计算和数据密集型应用提供了强大的支持。

另外为了更好地厘清逻辑，我们先画出其逻辑图，然后将逻辑图转化为逻辑分支表，髯胡分析出每一个执行块的延时，最后进行代码编程。

首先我们画出逻辑图如图2-16所示：

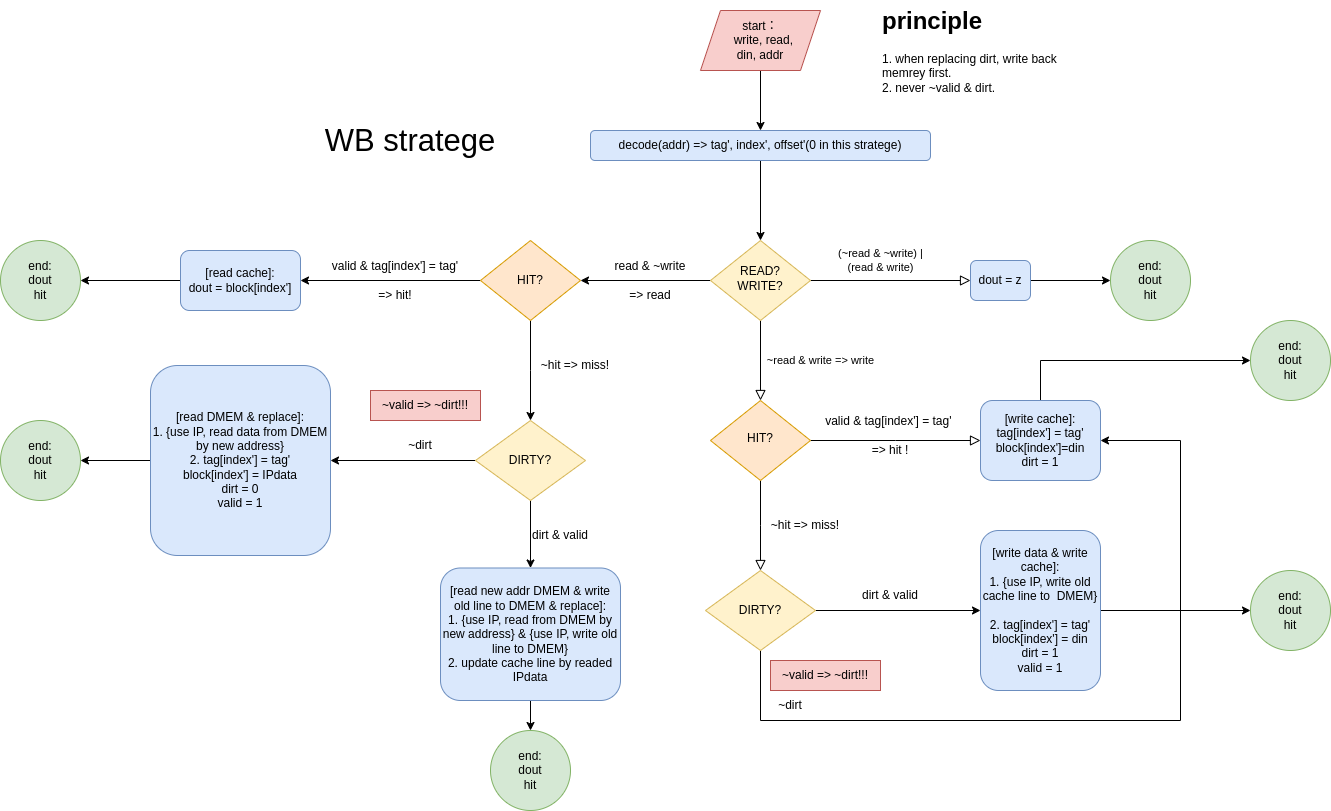


图2-16 Cache执行逻辑图

该图展示了一个写回（Write-Back, WB）和写分配（Write-Allocate, WA）策略的双端口缓存（Dual-Port Cache）的读写流程。图中主要包含以下几个部分：首先，遵循两个原则：当替换脏数据时，先写回内存，并且永不将无效和脏数据同时出现。流程从写、读、数据输入（din）和地址（addr）开始。地址被解码成标签（tag’）、索引（index’）和偏移量（offset’，在此策略中为0）。

接着，判断当前操作是读还是写。如果是读操作，继续下一步；如果是写操作，进行写操作，然后判断是否命中（HIT）。判断命中与否的标准是有效位（valid）且标签（tag[index]）与解码出的标签（tag’）匹配。如果命中，对于读操作，从缓存读取数据；对于写操作，更新缓存中的数据并设置脏位（dirt=1）。如果未命中，则进入未命中处理流程。

在未命中处理流程中，需要判断缓存行是否脏（dirt）且有效（valid）。如果是脏数据，需要将旧数据写回内存，然后替换缓存数据；如果不是脏数据，直接替换缓存数据。对于命中的读操作，从缓存块中读取数据（dout=block[index]）。对于未命中且不是脏数据的情况，从内存读取数据并更新缓存块，重置脏位和有效位。对于未命中且是脏数据的情况，将旧缓存行写回内存，同时读取新地址的数据并更新缓存块。

对于命中的写操作，更新缓存中的标签和数据，设置脏位。对于未命中的写操作，将旧缓存行写回内存，同时将新数据写入缓存并更新标签和脏位。通过该图可以清晰地了解WB+WA+DualPort缓存的读写策略及其流程，包括命中与未命中处理、脏数据处理等关键步骤。然后我们将条件合并，得到图2-17：

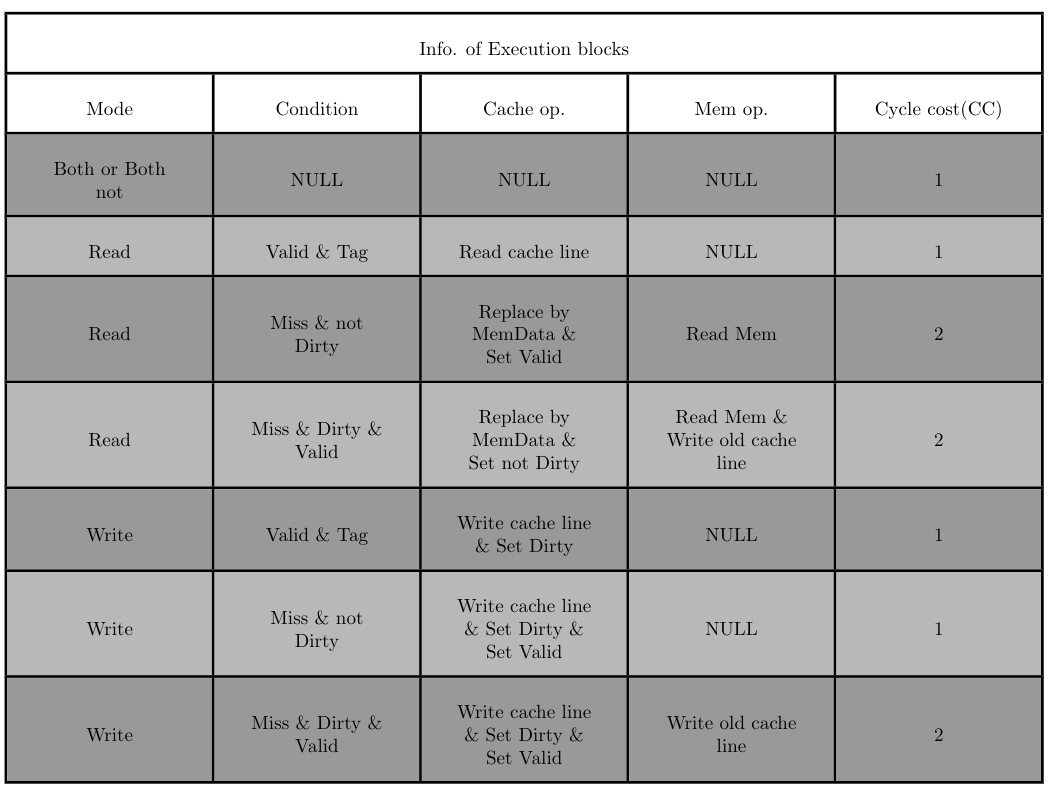


图2-17 Cache执行策略图

在前面的段落中，我们描述了写回（Write-Back, WB）和写分配（Write-Allocate, WA）策略的双端口缓存（Dual-Port Cache）的读写流程。现在，我们通过逻辑图生成的逻辑表进一步分析该流程。该逻辑表列出了不同操作模式下的条件、缓存操作、内存操作和周期成本（Cycle Cost, CC）。具体分析如下：

首先，在“Both or Both not”模式下，当既没有读操作也没有写操作时，所有条件、缓存操作和内存操作均为空，仅消耗一个周期。

其次，在“Read”模式下，有三种情况需要考虑：

i. 当读取操作命中且标签有效时，直接从缓存中读取数据，不涉及内存操作，仅消耗一个周期。

ii. 当读取操作未命中且缓存行不脏时，需要从内存读取数据并替换缓存块，同时设置缓存块为有效，整个过程消耗两个周期。

iii. 当读取操作未命中且缓存行既脏又有效时，需要先将旧缓存行写回内存，再从内存读取新数据并更新缓存块，同时设置缓存块为非脏，整个过程消耗两个周期。

在“Write”模式下，同样有三种情况需要考虑：

i. 当写操作命中且标签有效时，直接更新缓存中的数据并设置为脏，不涉及内存操作，仅消耗一个周期。

ii. 当写操作未命中且缓存行不脏时，直接将新数据写入缓存，并设置为脏和有效，不涉及内存操作，仅消耗一个周期。

iii. 当写操作未命中且缓存行既脏又有效时，需要先将旧缓存行写回内存，再将新数据写入缓存并更新标签，同时设置缓存块为脏和有效，整个过程消耗两个周期。

通过该逻辑表，可以清晰地了解在不同条件下缓存和内存的操作，以及每种操作的周期成本。这些信息有助于更好地理解和评估该缓存策略的执行效率和性能，确保在实际应用中能够有效地利用缓存资源，提高系统整体性能。

3. 具体实现

分析完Cache的理论逻辑部分，我们现在要对Cache部件进行编程实现。首先我们根据表中的条件分配信号，然后根据Cache Operations那一列使用if else语句编写程序。如图2-18：

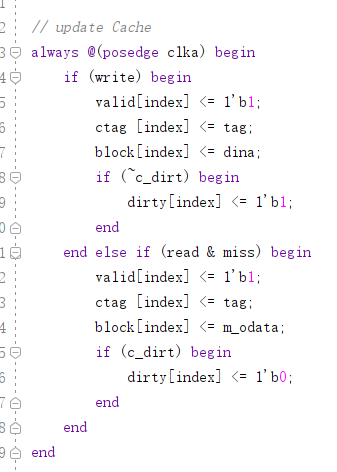


图2-18 Cache更新代码实现

然后我们调用vivado自带的DRAM IP来实现我们的内存部分，其具体配置见第五节参考设计说明。最后我们根据表中的条件分配wire信号即可实现Cache。

三、实验过程以及总结（40%）

（一）设计工作日志

【1】**6.26**：（1）**朱云哲，胡芸明，郑梦仁**：任务：1.算术逻辑指令开发。2.分支指令开发。结果：完成算术逻辑指令开发任务，修改Controller模块，分支指令开发遇到困难。

【2】**6.27**：（1）**朱云哲，胡芸明，郑梦仁**：任务：1.分支指令开发。2.访存指令开发。结果：完成分支指令、访存指令开发任务。Controller模块修改基本完成。

【3】**6.28**：（1）**朱云哲，胡芸明**：任务：1.auipc，lui指令开发。2.运行测试指令集，仿真。结果：完成auipc，lui指令开发。测试指令集仿真结果错误。

（2）**郑梦仁**：任务：1.了解双模动态分支预测，进行代码实现。结果：知识储备完成。

【4】**6.29**：（1）**朱云哲，郑梦仁**：任务：1.运行基础测试指令集，解决仿真错误。结果：修改并完善前推模块，冒险检测模块，通过所有基础测试指令集。

【5】**7.5**： （1）**朱云哲、郑梦仁**：任务：1.接入分支预测模块，测试分支预测模块效果。结果：仿真错误，预测状态机不跳转。

【6】**7.6**： （1）**朱云哲、郑梦仁**：任务：1.重新修改分支预测模块，重写状态机，重新仿真。结果：仿真正确，预测结果正确，状态机在停顿时出现不跳转bug。

【7】**7.8**： （1）**郑梦仁**：任务：1.重新修改状态机，重新编写分支预测模块，仿真综合测试指令集。结果：仿真正确，分支预测及状态机工作正常。

（2）**胡芸明**：任务：1.实现Cache，确立Cache组织结构，确定写策略。结果：完成Cache代码实现。

【8】**7.9**： （1）**朱云哲、胡芸明**：任务：1.接入Cache至CPU，仿真测试指令集。2.修改冒险检测模块，重新规划冒险信号控制通路。结果：Cache-miss时仿真错误。修改冒险模块后Cache-miss处理正常，仿真结果正确。

【9】**7.10**：（1）**胡芸明**：任务：1。解决Cache-miss时写分配策略实现。结果：通过生成单端口BRAM实现，仿真结果正确。

【10】**7.11**：（1）**朱云哲，郑梦仁**：任务：1.仿真所有测试指令集，观察运行结果。结果：仿真结果均正确，项目基本完成。

（二）主要的错误记录

1、错误1

（1）错误现象

仿真基础测试指令集S\_LInstTest\_finsh时，第二条指令sw x3,0x3(x0) 写入位置错误，预期写入位置32’h3，实际写入位置32’hff。

（2）分析定位过程

由于指令出错位置较早，可以直接分析第一条指令ori x3,x0,0x0ff与第二条指令sw x3,0x3(x0)的关系。我们先跟踪了sw指令的ID阶段执行结果，发现取出数据均没有问题，然后分析EX阶段sw的计算结果，发现计算结果为32’hff。因此断定是ALU的操作数发生问题。我们跟踪了ALU的操作数num1与num2，发现num2的前推信号本应是2’b00，但实际是2’b01。然后去前推模块中查看前推信号的执行逻辑，发现前推信号的执行逻辑正确，于是推测可能是传入错误地址，所以继续分析ori指令，发现其目标寄存器地址为x3，与sw的立即数32’h3重合，定位到了错误。

（3）错误原因

由于对于sw指令，其需要通过立即数来偏移获取内存地址，这个时候立即数在指令中所占的位置恰好与r指令的rs2位置相同，同时流水线将sw的立即数作为了ALU操作数传入前推的多路选择器中，导致了前推模块在判断时，将立即数作为了寄存器地址参与了数据冒险判断，导致误判sw需要数据前推，直接前推了ori的x3结果。

（4）修正效果

想要规避此类错误，只需要判断传入的数据是否是立即数即可避免，因此我们选择直接在前推中加入ALUsrcE信号，表示当且仅当源寄存器2的数据被选择进行ALU计算时才进行前推判断，否则为使用立即数计算，不需要进行前推。修正后仿真结果正确，寄存器值变化正确，仿真效果见第四节设计结果。

（5）归纳总结

此类数据冒险处理错误在于指令本身的结构问题，由于RISC-V32位指令架构固定了opcode，rs1，rs2，rd的位置，导致在译码阶段直接截取相应部分获取地址即可。但这样会导致I、S类型指令的立即数混淆地址信号。所以在设计相应的前推模块时，还需要注意指令结构与地址信号来判断。同时在寻找错误时，可以从前推信号入手，倒推错误信号来源一步一步追踪到原因。

2、错误2

（1）错误现象

进行综合测试1时发现在200μs后一直处于死循环状态，疑似出现数据计算问题，如图3-1与3-2所示：

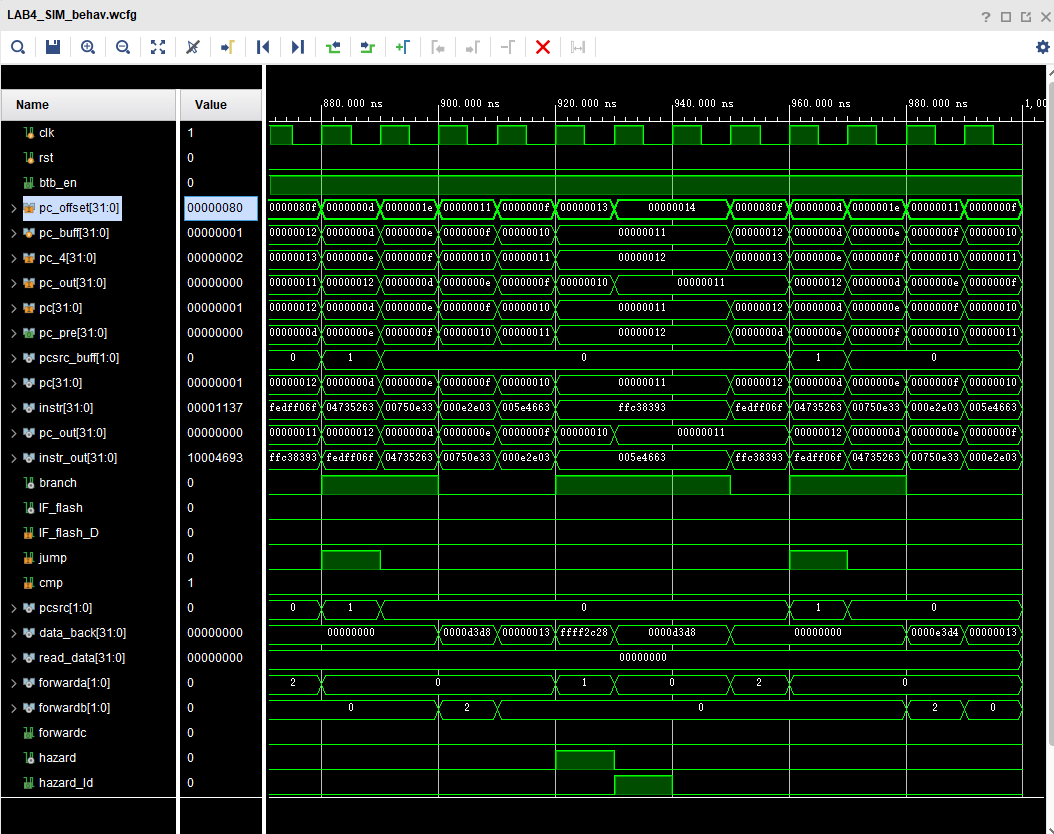


图3-1 执行时间1000ns下综合测试1仿真

执行200μs左右，出现死循环：



图3-2 执行时间200μs下综合测试2仿真

（2）分析定位过程

首先由综合测试1的代码可以发现，程序最后是有死循环出现的，但是此时发现如果是死循环，那么应该不断执行同一条指令，但是发现pc一直在最后不断重复32’h2a~32’h34->32’h8->32’h3d->32’h2a。首先我们需要排除是否可能为立即数写回地址错误，我们通过对比前十个循环，验证了每个寄存器的变化与指令执行顺序，发现在前十个循环的排序中，指令的功能均正确实现，当分支预测错误，Cache-miss时，冒险检测模块也能正确处理。在200μs，也就是死循环出现前，我们认为流水线的执行没有问题。

现在排除了单个指令执行错误的可能，那就要验证是否是跳转逻辑执行出现问题。这个验证任务十分耗时，我们在184750ns处发现了死循环的开头，如图3-3，当程序跳转至0x00008067指令时，程序后面的执行逻辑为：

QuickSortReturn-> PartationEnd->QuickSort->QuickSortReturn

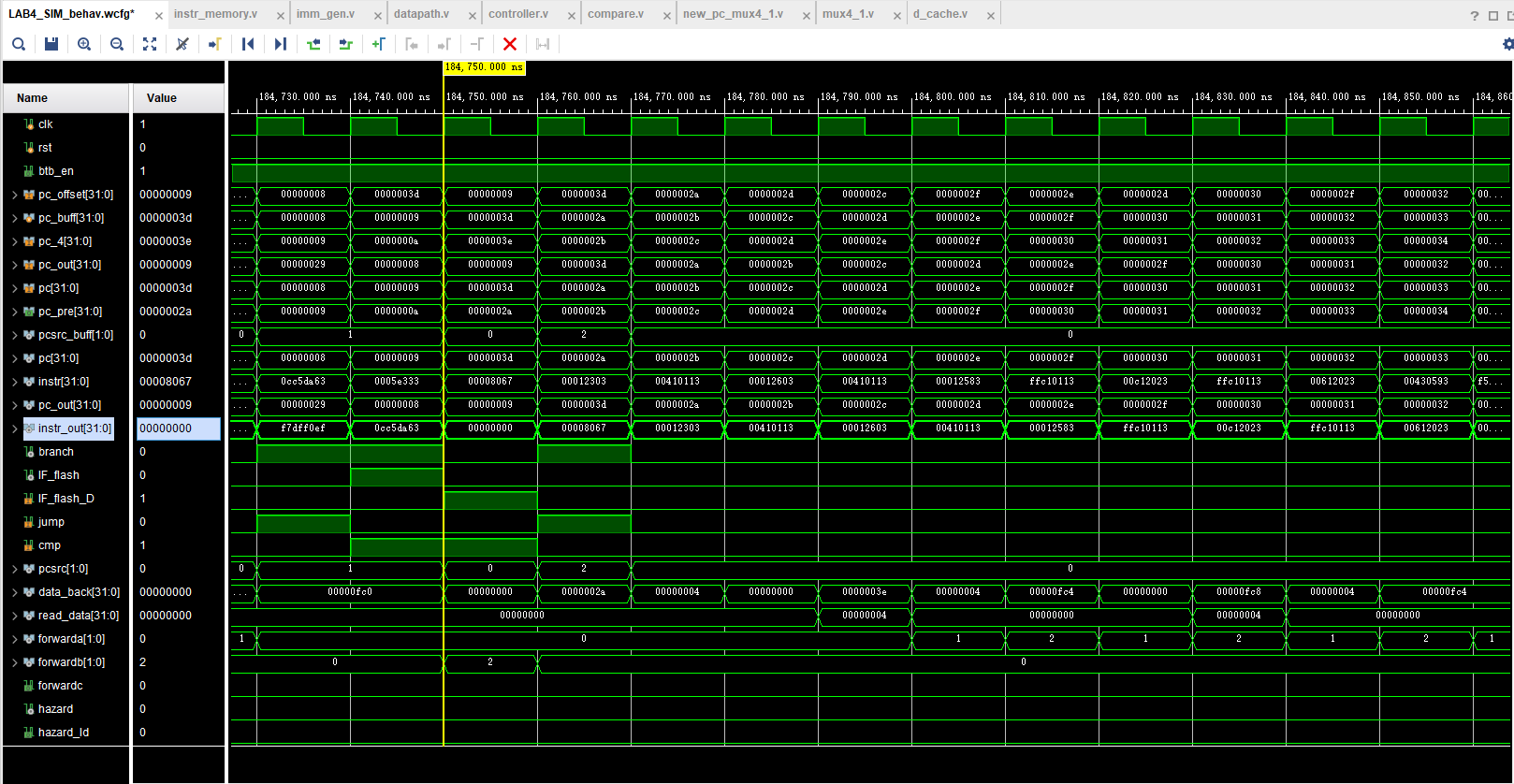


图3-3 死循环出现时仿真图

由此程序陷入死循环。

（3）错误原因

找到了死循环的位置后，我们分析了前后一个循环的指令，发现造成循环的关键指令是 bge t1, t2, PartationEnd # if i>=j, branch to next step。在整个汇编文件中，t2的行为只有t2=t2-4这一种操作，这意味着t2一定会变为0；而t1的来源只有t1=t1+4与内存中栈的值，而栈的值初始为正，则t1一定≥0，因此该语句执行足够长一定会跳转，跳转时t2=0，t1≥0，这样的话第一层循环PartationEnd无法逃脱。然后我们关注到循环PartationEnd内部，我们发现死循环的核心逻辑是不断弹出栈并写入栈，但是该循环中sp值一直处于循环状态，这就导致核心寄存器t1，a1，a2的值一直无法变化，这就导致所有跳转指令均会跳转，构成了循环且无法逃脱。

（4）修正效果

由于前面的排序循环中，我们关注了寄存器的变化与逻辑均正常，指令的执行逻辑与跳转均正常，Cache的写入与命中处理均正常，所以我们暂时找不到数据通路与控制通路中的异常。尽管我们发现了死循环，但是这并没有影响流水线执行指令的逻辑与正确性，因此我们希望能在未来对该测试指令集进行修改再次运行，或者采用小规模数据进行模拟来获得异常原因与解决办法。

1. 错误3

（1）错误现象

进行分支预测后，执行错误的分支情况，也即使在分支预测错误的情况下未进行冲刷以及相应处理。

1. 分析定位过程

发现预测错误仍然继续执行而程序没有做出反应，立马意识到信号的优先级考虑出现了问题。又由于自己含糊地将hazard数据冒险信号优先处理，因此可以很快地发现问题并进行修正。

1. 错误原因

思考不到位或者是浅显地考虑问题。只是考虑到分支预测的结果，考虑到分支预测中进行两个寄存器数的计算比较结果可能是需要等待相应的访存结果，因此想当然地认为数据冒险的优先级要高于预测错误（也即是优先对hazard信号进行处理）。但是实际上，我们知道，只要仔分析流水线寄存器框架图，以及相应的信号的传递过程，很清晰地发现，分支预测以及冒险检测的结果对应的两条指令，毫无疑问是分支预测传递的结果对应的指令是更久远的，这也就意味着吗，一旦历史发生错误，后面进行的指令都有可能被冲刷，因此，可以肯定分支预测相应的信号处理一定要优先于数据冒险的。

1. 修正效果

进行优先级的调整，也即是优先判断是否发生预测错误，并对相应的寄存器进行更新。仿真后发现预测错误的结果可以得到反馈，并且得到冲刷。

1. 归纳总结

逻辑上成立的东西很容易给人以错觉，在未来学习的道路上，切忌“想当然”，经过深切思考与研究的方法才有可能经得起考验。

1. 错误4
2. 错误现象

在缓存管理系统中，当尝试读取一个缺失的脏缓存行时，系统需要先将旧的缓存行（脏缓存行）写回内存，然后从内存加载新数据到缓存中。这个过程在单端口缓存设计中需要至少4个时钟周期（ClockCycle），导致显著的处理延迟。

1. 分析定位过程

【1】初步分析：在性能瓶颈分析中，发现缓存处理缺失的脏位时性能显著下降。

【2】模拟测试：通过模拟不同的缓存操作，确认在处理缺失脏位时，单端口缓存的处理速度慢于双端口。

【3】策略评估：评估修改策略（即读取时不更新缓存行）和硬件改进（双端口设计）两种方法，通过对比硬件资源消耗和性能提升来确定最优解。

1. 错误原因

当出现读缺失脏位的时候，我们首先需要将old Cache Line写入内存，然后从内存中读取新数据对Cache Line进行替换。但是此时如果是单端口的话则至少需要4个CC来运行这个执行块。

1. 修正效果

为了使我们的缺失惩罚对齐，我们想了两个办法：一个是修改策略，即在这种情况下我们直接从Cache读取数据返回，而不对Cache Line进行更新。另一种就是现在所说的双端口设计。第一种方法的好处是可以节省硬件资源。第二种方法的好处是可以在一定情况下更快。综合考虑下，我们选择使用第一种方法来进行改进，最终测试通过。

（三）项目计划调整情况

项目按照计划进行。但是在进行综合测试1的时候出现死循环，需要重新检查指令执行顺序，因此原定计划7.12完成全部项目变为14号完成所有项目并编写实验报告。7.12-7.14继续进行综合测试1的仿真与调试，在进一步探究死循环成因时，由于数据规模较大，人力比对执行结果较困难，因此我们只能确保指令的执行顺序正确与寄存器的变化正确。

为了解决该问题带来的项目停顿，组员朱云哲负责对该问题进行解决，组员胡芸明与郑梦仁继续报告编写，保证报告与项目的顺利提交。

四、设计结果

（一）设计交付物说明

本项目代码见压缩包内附件，此处不粘贴，仅说明提交的代码文件结构及相关操作要求。

Src——项目源代码

|——ip——项目使用IP核

| |——blk\_mem\_gen\_0.xci

|

|——rtl——项目设计verilog文件

| |——top.v——项目顶层文件

| |——riscv.v——CPU 核顶层文件

| |——controller.v——控制器

| |——main\_control.v——主控制器

| |——alu\_control.v——ALU控制器

| |——datapath.v——数据通路

| |——new\_pc.v——pc计数器

| |——regfile.v——通用寄存器堆

| |——imm\_gen.v——立即数扩展指令

| |——ALU.v——ALU计算单元

| |——sl2.v——pc左移对齐地址模块

| |——adder.v——32位加法器

| |——pc\_offset\_adder.v——32位加法器

| |——new\_pc\_mux4\_1.v——pc方向选择器

| |——btb\_pre.v——动态分支预测模块

| |——mux2\_1.v——二选一选择器

| |——mux4\_1.v——四选一选择器

| |——mux2\_1\_control.v——二选一选择器

| |——mux3\_1.v——三选一选择器

| |——compare.v——分支指令前推比较模块

| |——data\_process.v——数据处理模块

| |——IF\_ID.v——IF/ID流水线寄存器

| |——ID\_EX.v——ID/EX流水线寄存器

| |——EX\_MEM.v——EX/MEM流水线寄存器

| |——MEM\_WB.v——MEM/WB流水线寄存器

| |——Hazard\_detecting.v——冒险检测模块

| |——Forward\_detecting.v——前推检测模块

| |——instr\_memeory.v——指令寄存器堆

| |——d\_cache.v——D\_Cache

|

|——testbench——项目仿真文件

|——LAB3\_SIM.v——项目整体仿真文件

操作要求：

【1】本项目使用vivado版本为2023.2，其中BRAM模块生成配置如参考设计说明所示。

【2】LAB3\_SIM文件为仿真文件，如果要运行相应测试指令集，直接在instr\_memeory.v文件种找到目标测试集解注释，保存后单击仿真即可开始测试。

【3】对于综合测试1、2，由于所需时间较长，这里建议增加仿真时间。对于综合测试1，仿真2s，然后大约观察到出现死循环即可停止；对于综合测试2，仿真800μs即可观察到计算结果。

【4】如果有信号无法观察，可以在Scope处选中目标模块，然后在Objects处选择信号，邮件add to window即可观察信号；如果信号为空，重新仿真即可观察。

【5】对于综合，将文件导入后直接进行RTL综合即可观察电路。

（二）仿真结果及其分析

测试指令集的仿真结果如下面所示，本小节分析CPU功耗、时延与关键路径。

前五个基础测试指令集较为简单，我们选取L\_Sinst这个较综合的测试来进行分析；综合测试1在前面的错误分析中已经探讨过，此处不再分析。

【1】L\_Sinst仿真结果分析

**行为仿真：**S\_Linst指令集主要测试sw与ld访存指令，可以帮助分析有无Cache的性能。其coe文件如图4-1所示：

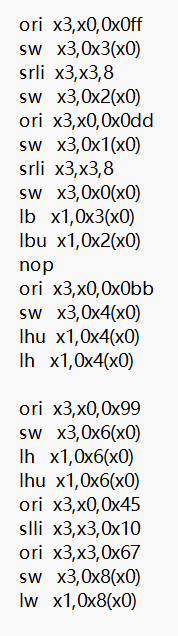


图4-1 L\_Sinst指令文件

我们发现第一条sw指令将x3存入了内存32’h3位置，但是由于初始Cache为空，Cache发生miss，但由于我们块大小为1字，可以直接将数据存入Cache并置dirty。此时当lb x1,0x3(x0)进入时，Cache-hit，可以在一个周期就快速取出先前的数据，如下图4-2所示：

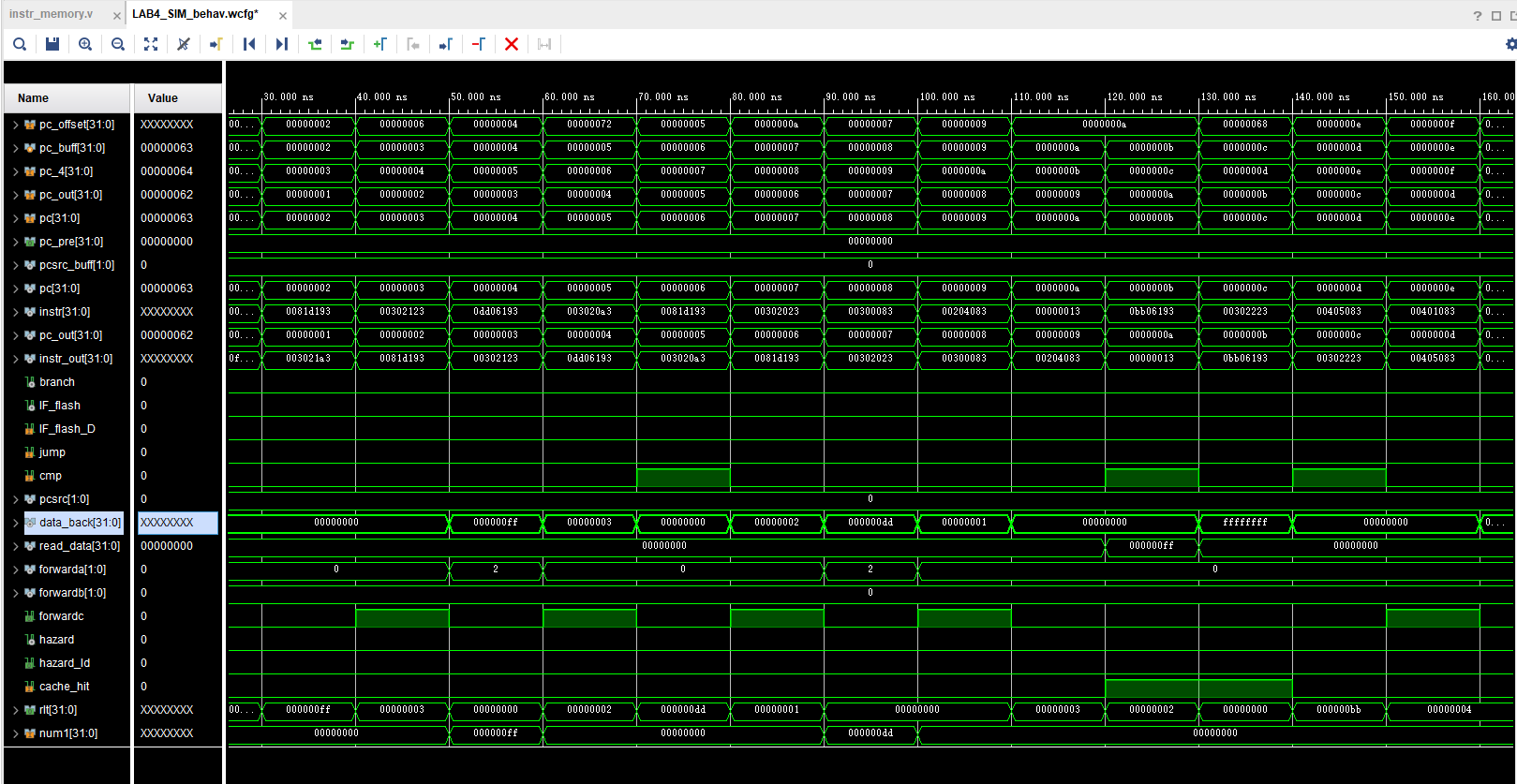


图4-2 L\_Sinst仿真结果分析

lb x1,0x3(x0)为第九条指令，对应0x30083，我们向后数两个周期，也就是在120ns处，该指令进行到MEM阶段，此时观察cache\_hit信号发现为1，表示Cache命中，同时lb指令为取有符号字节，因此取出0xffffffff，且在第130ns在WB阶段写回。可以发现Cache的加入缩短了访存指令周期，将2个周期时延变为1个周期。

**时延分析：**本测试流水线没有任何阻塞，数据冒险被数据前推解决，sw指令的访存不存在数据冒险，不需要阻塞。在测试了所有测试指令集后，我们发现影响本项目CPU频率的主要因素在于访存指令。由于本项目的Cache容量较小，导致在进行综合测试这一类测试时，Cache中存储的数据经常会替换而且需要向内存读取数据，因此后续换用更大容量的Cache是改进方法。

**功耗分析：**本项目未上板验证，故无法分析实际功耗。但主要功耗应该处于ALU计算单元和访存d\_cache单元，前者需要进行大量运算，后者需要频繁更改寄存器的值。

**关键路径：**由于仿真时不考虑延时的，我们只能理论上分析处理的关键路径，对于本测试指令集，我们选取ld指令分析。其关键路径为：

pc->regfile->mux\_2\_1->mux3\_1->ALU->d\_Cache->data\_process->mux4\_1->regfile

（三）设计演示结果

本项目设计仿真结果如图4-3~4-10所示，由于部分指令集的结果较长，图片无法完全放置，故只截取了部分仿真图片，可以按照data\_back信号和instr\_outi判断每条指令的执行结果和执行顺序是否正确。

注：instr\_out对应流水线ID阶段。假设要观察指令A，那么在instr\_out种寻找到指令A，然后从A向后数三个周期即为A指令的WB阶段，此时纵向寻找data\_back信号即为A指令最终写回寄存器堆的结果。

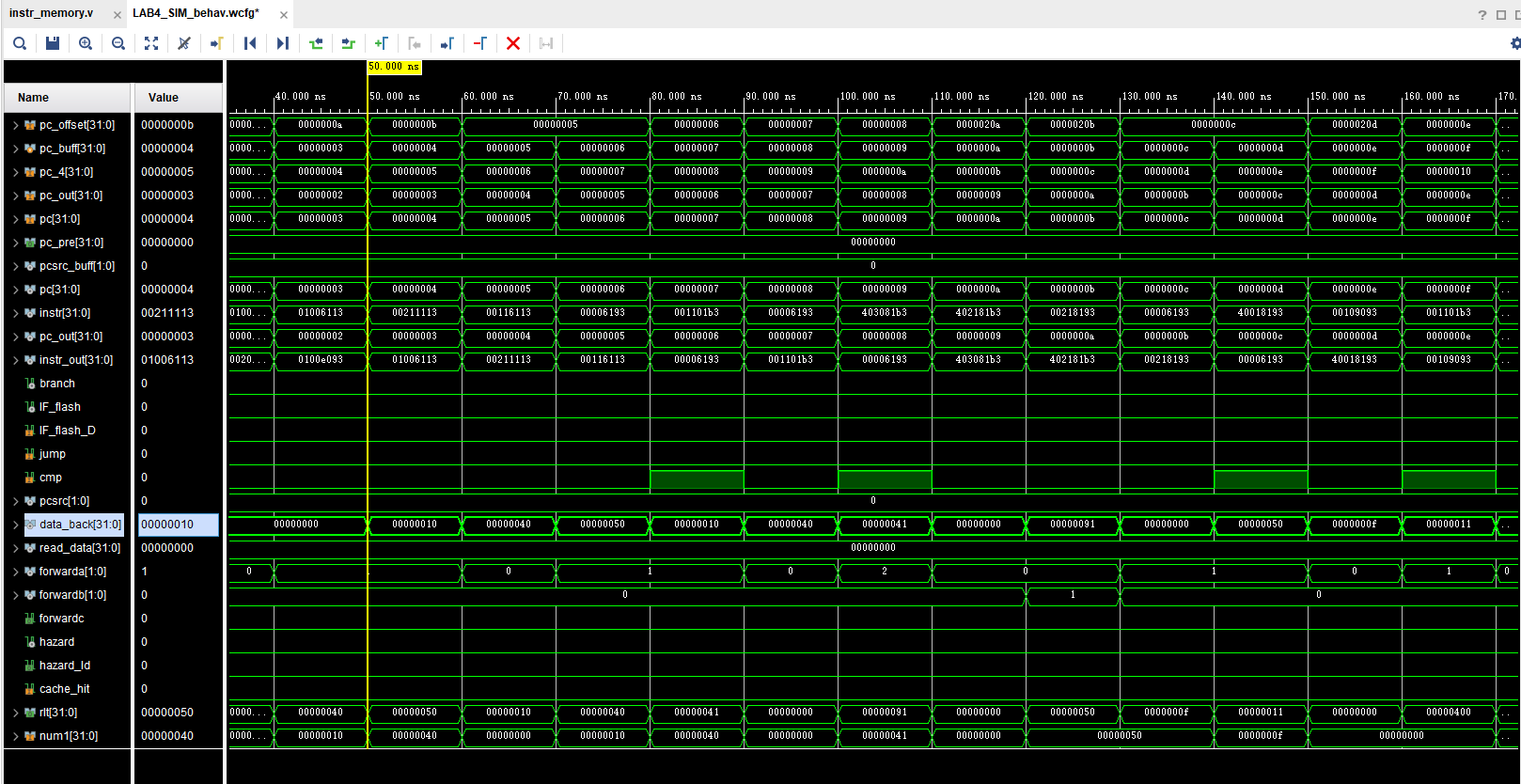


图4-3 算术指令测试仿真结果

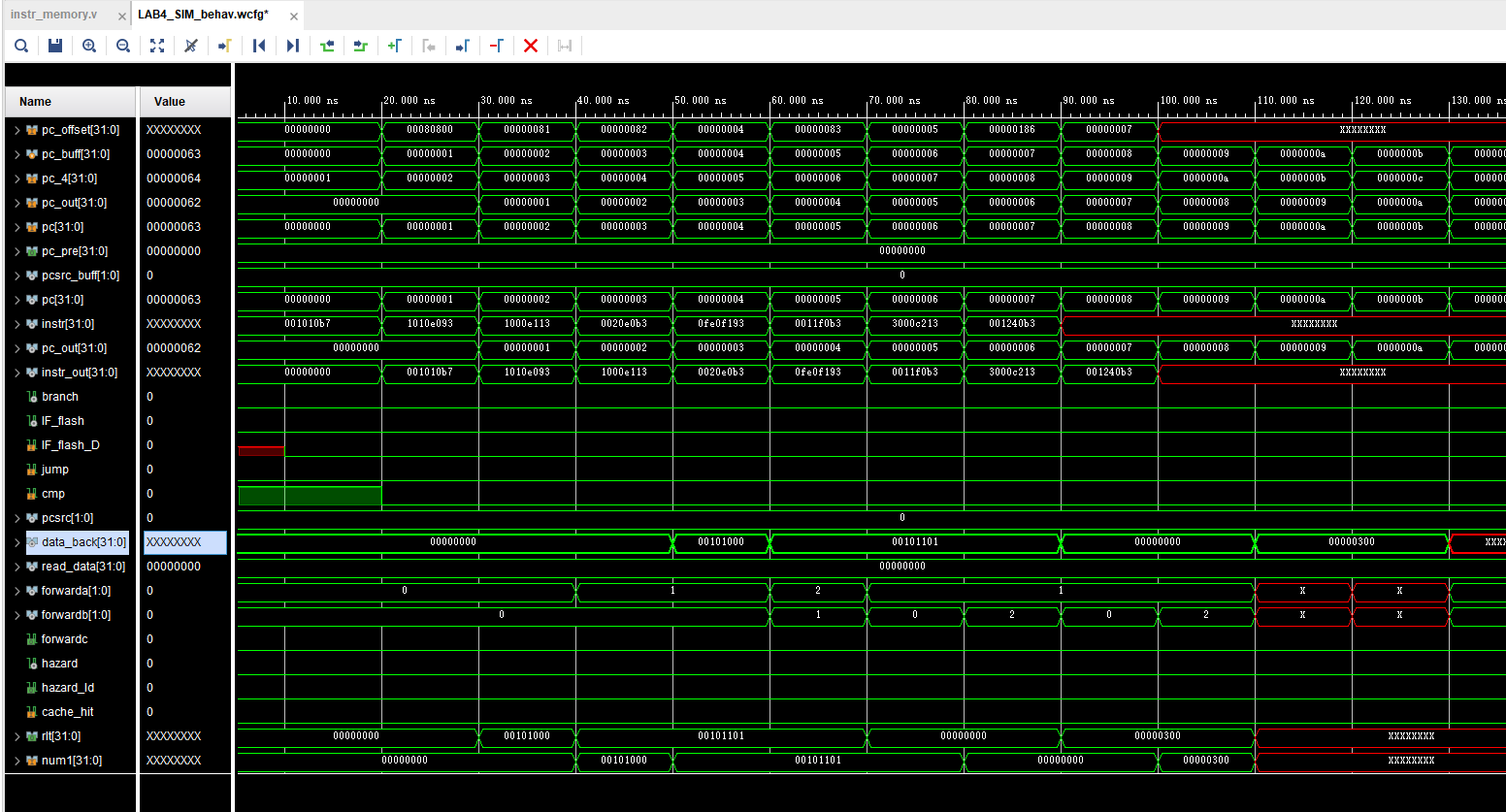


图4-4 逻辑指令测试仿真结果

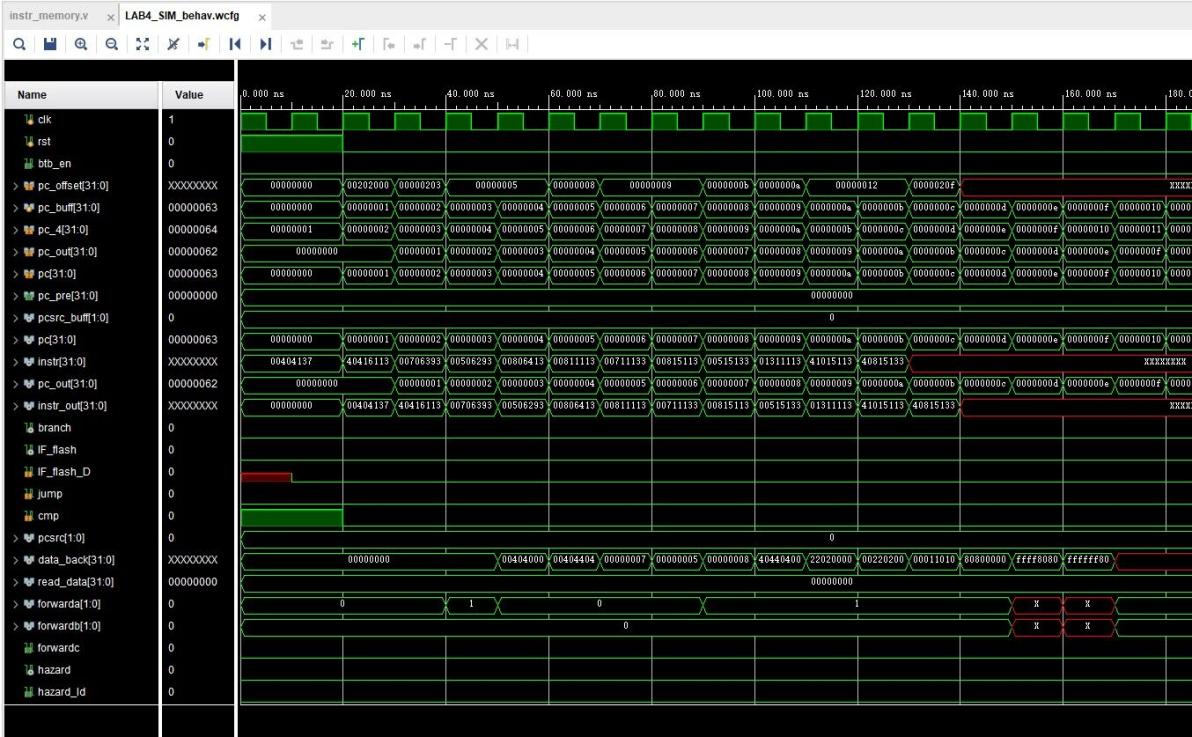


图4-5 移位测试仿真结果

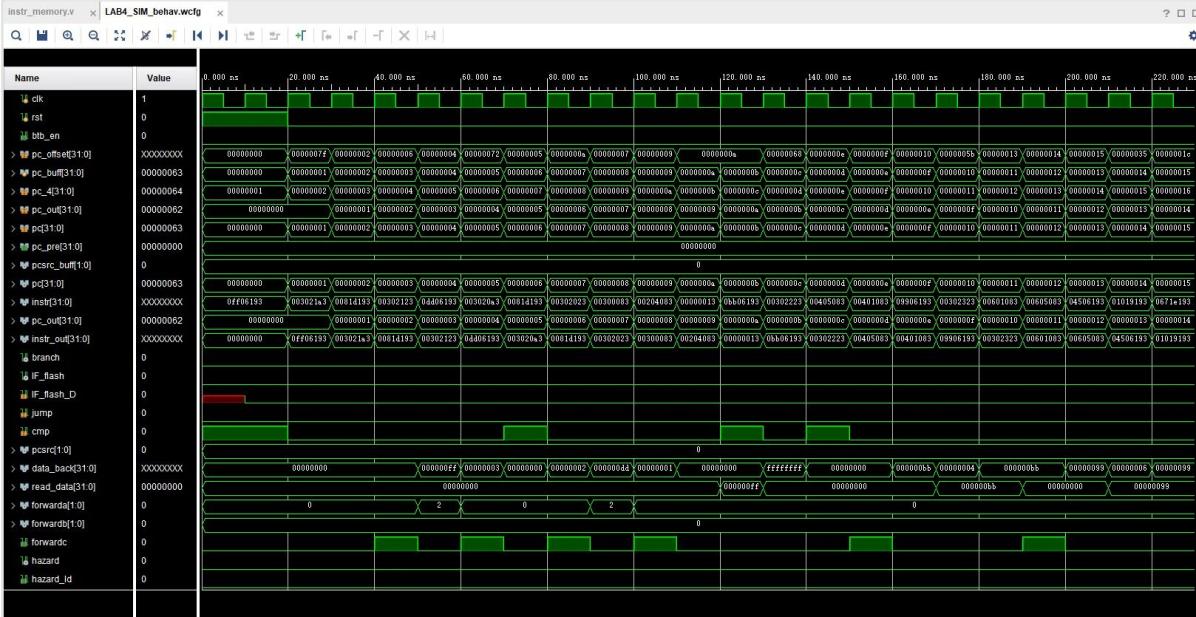


图4-6 L\_Sinst测试仿真结果

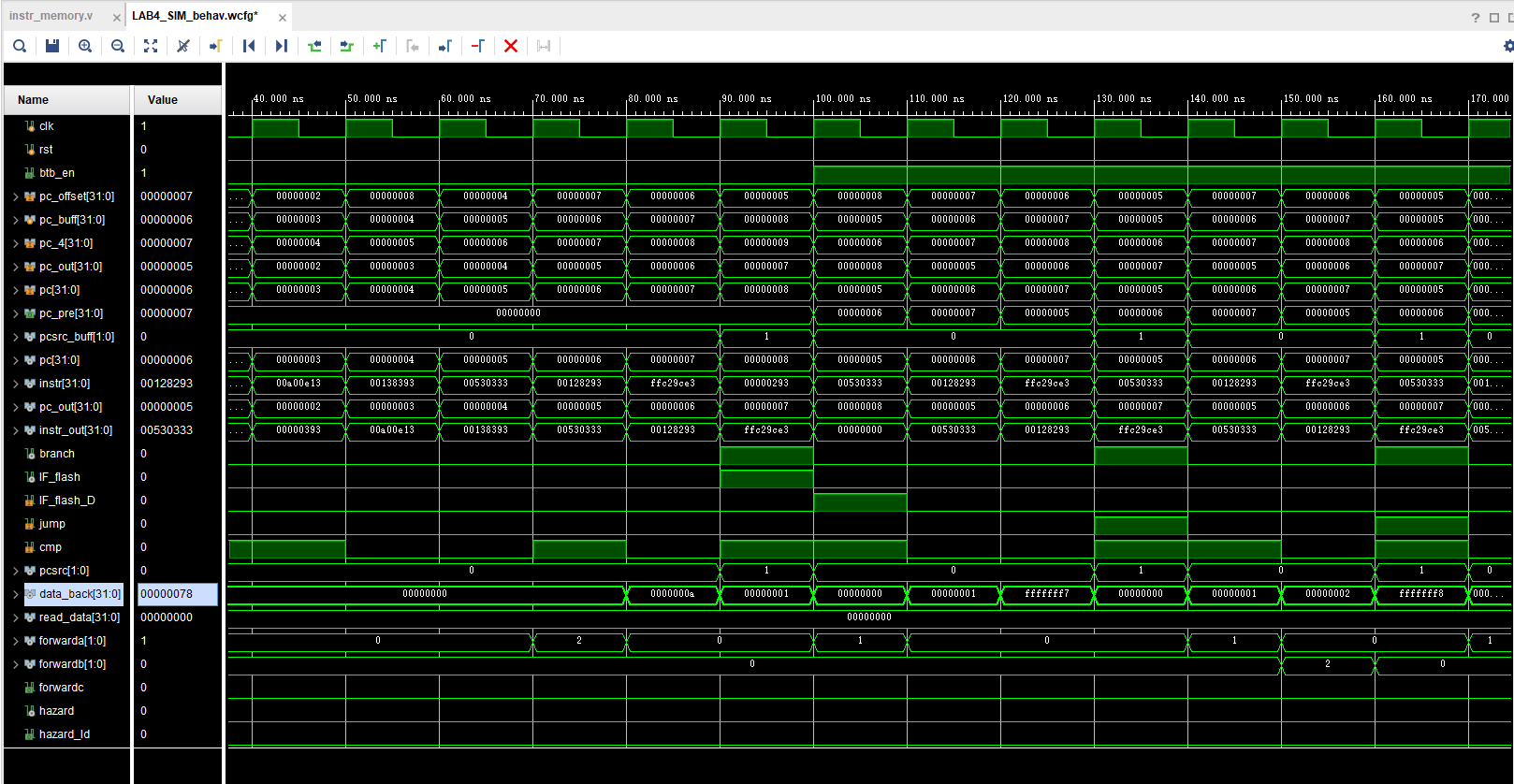


图4-7 分支指令bht测试仿真结果

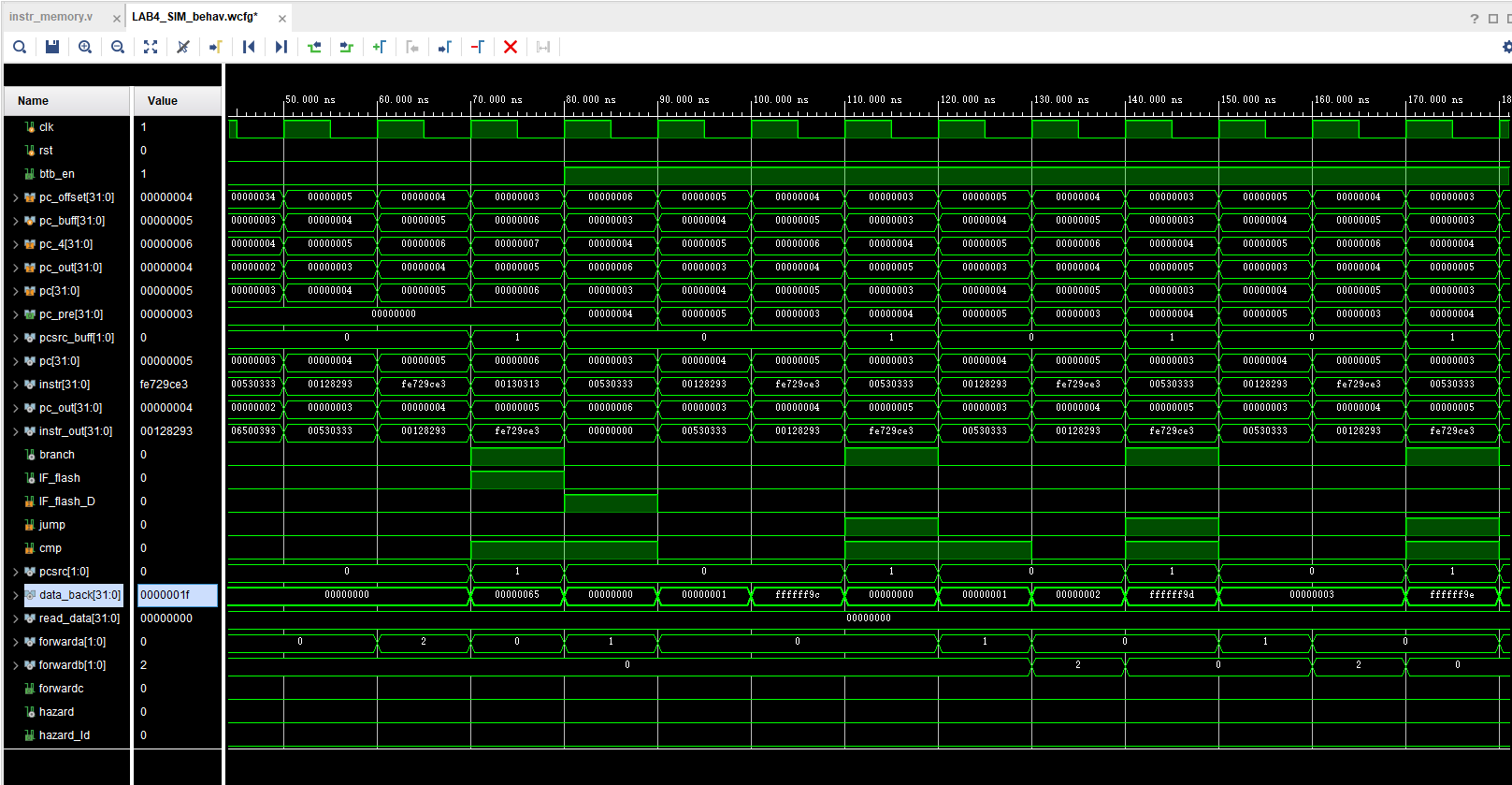


图4-8 分支指令btb测试仿真结果

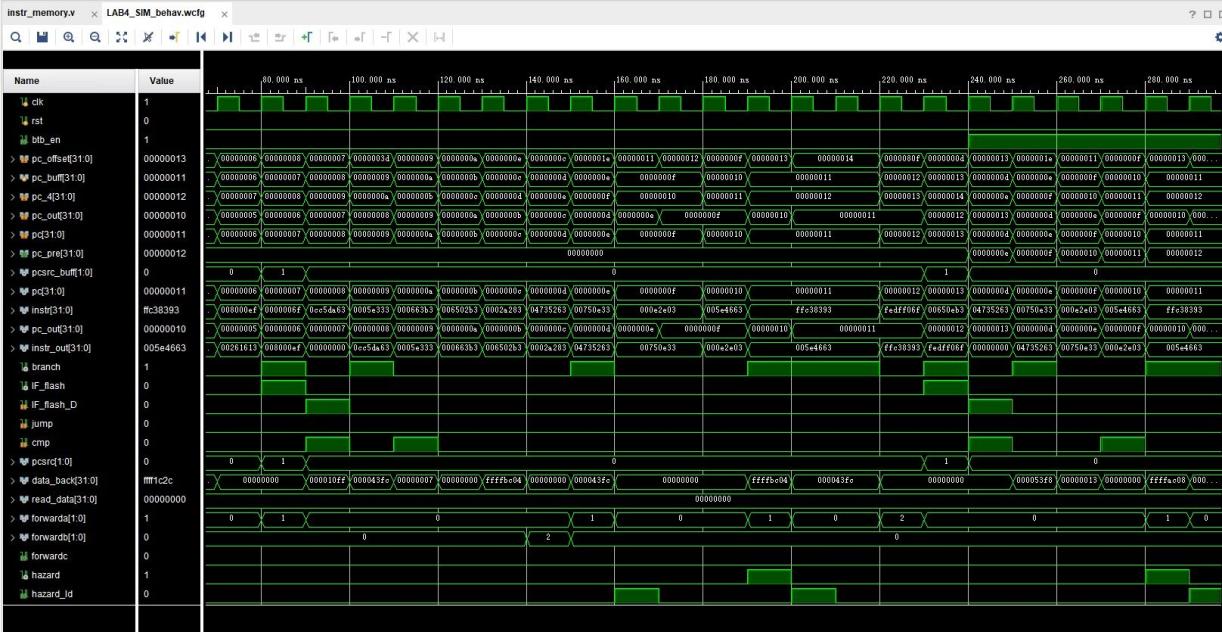


图4-9 综合测试1仿真结果

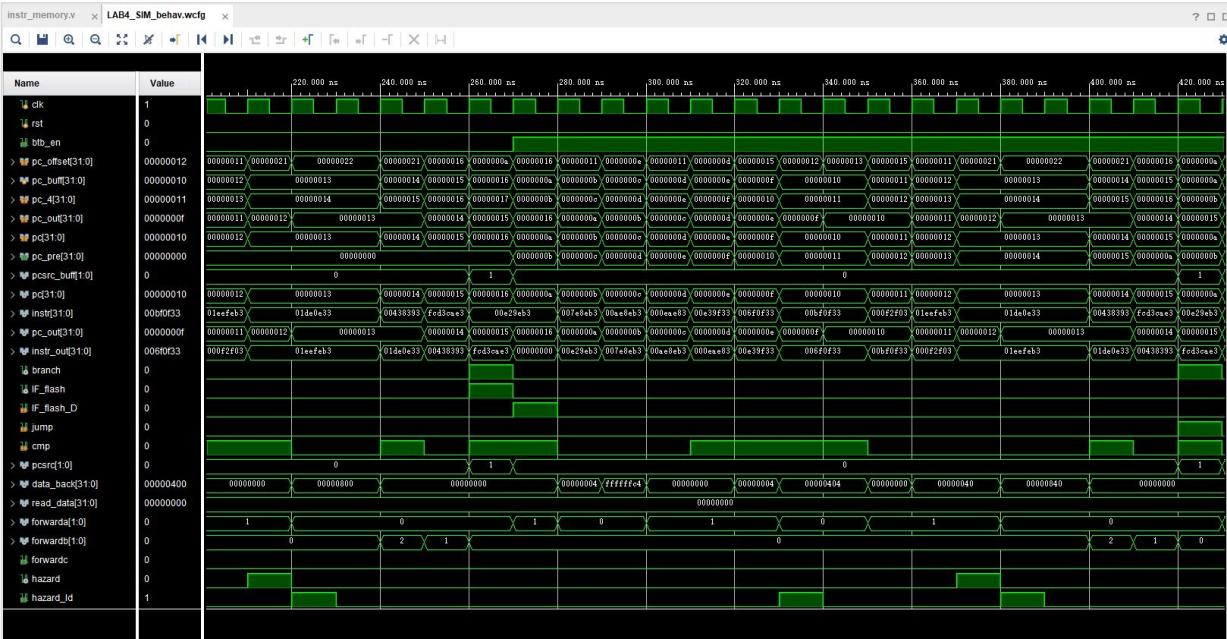


图4-10 综合测试2仿真结果

五、参考设计说明

【1】本项目使用了vivado2023.2版本的单端口BRAM IP。其设置如图5-1，5-2，5-3所示：

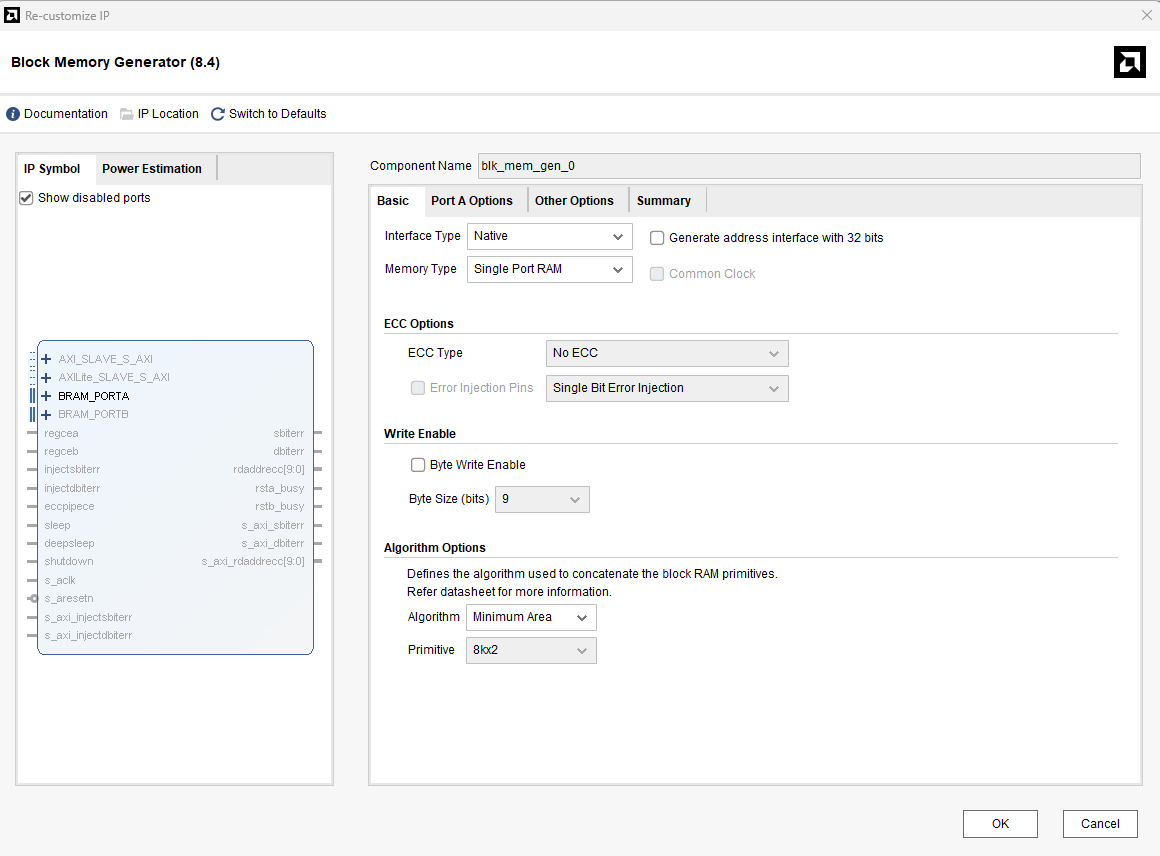


图5-1 BRAM IP配置1

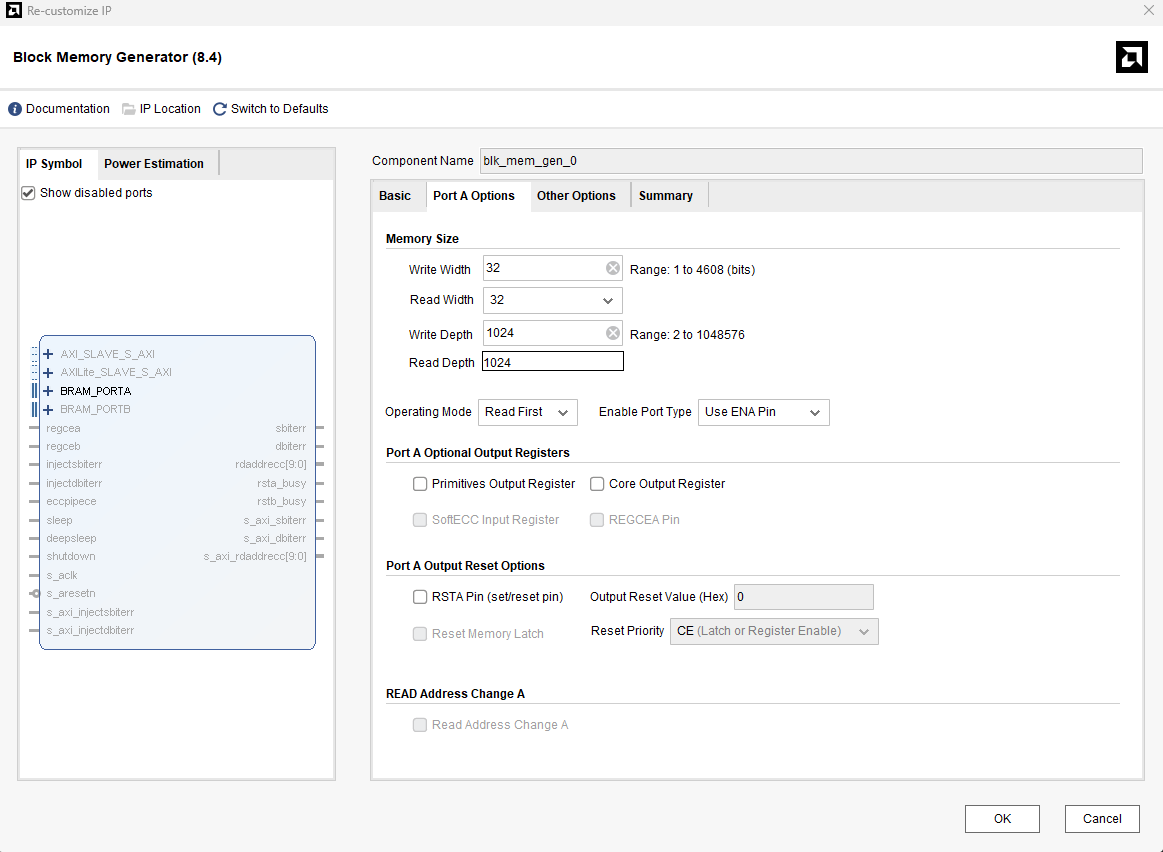


图5-2 BRAM IP配置2

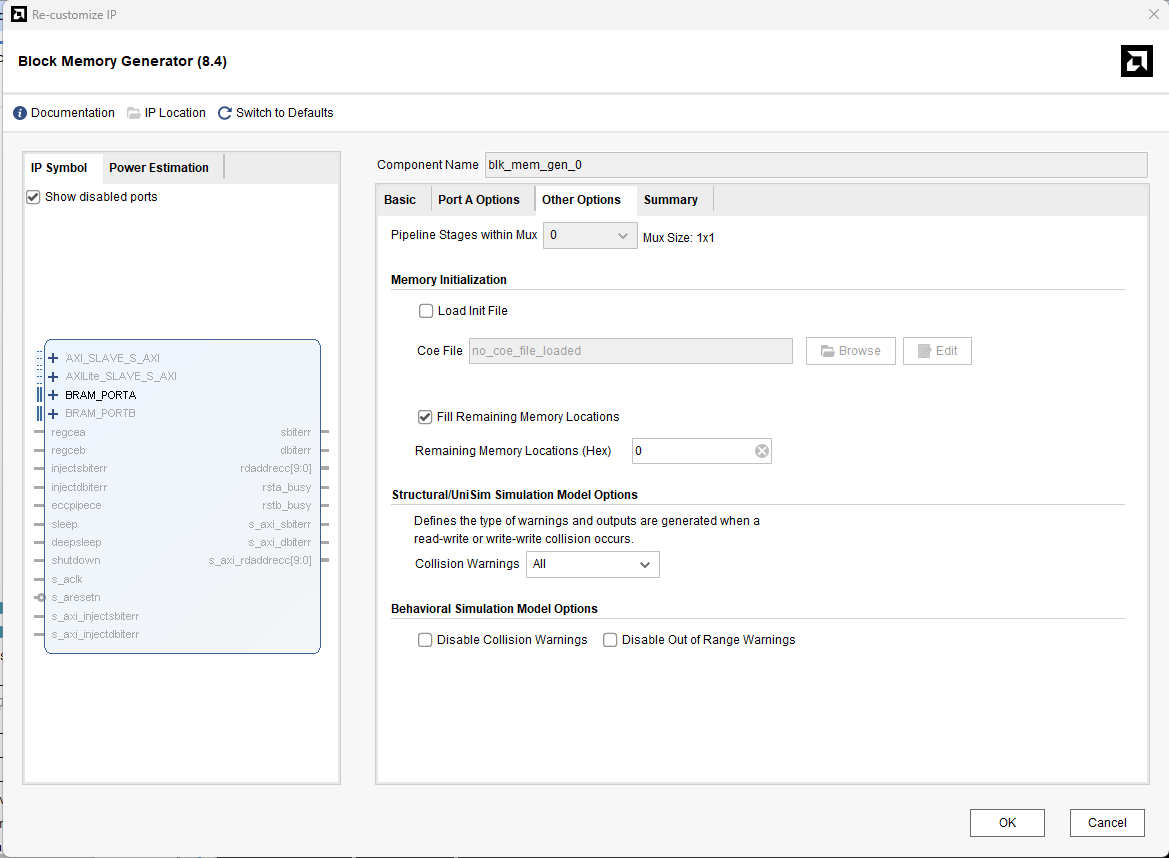


图5-3 BRAM IP配置3

【2】本项目借鉴了双模动态分支预测的状态机设计，出处为计算机组成原理课程PPTCST123-Lecture-18 P2。项目中使用地方为btb\_pre模块的状态定义。

六、总结

（一）总结及其展望

本项目CPU完成了19条算术逻辑指令，6条访存指令，8条分支指令。同时实现了双模动态分支预测与一级Cache，Cache实现了写返回与改进的写分配策略，并完成了模块的搭载使用。基本完成了所有规定任务，并优化了冒险处理与数据前推模块。

本项目的CPU并未上板实验，故无法获知其具体实际性能，但是在仿真运行时，得益于较好的分支预测，冒险处理与数据前推，流水线基本能在一个cycle内完成一条指令。

本项目通过了全部5个基础测试指令集，通过了综合测试2矩阵乘法，在综合测试1的快速排序中出现死循环且无法解决，但是能够正确分析死循环成因，并保证指令的逻辑与数据执行正确，因此我们希望在未来能够改进综合测试1的汇编程序或者采用小规模数据来更方便地获取死循环成因并解决。

本项目设计中地不足在于时序逻辑的使用，对于流水线，组合逻辑可以用于各阶段快速获取信号与数据，而时序逻辑可以保存目标信号与数值用于流水线各类判断与计算。但是本项目在编写时，过多使用时序逻辑进行分情况判断信号，导致在分支预测中有时会出现信号的突变，后续可以将其更改为组合逻辑，提升模块稳定性。其次，我们的Cache的容量并不是很大，块大小为1字在某种程度上简化了写回策略的实现，尽管如此，也能很明显发现Cache的效率高于直接使用BRAM读取数据。在未来我们可以尝试更多形式的Cache实现方式如多路组相联，更改块大小，调用双端口BRAM等。

（二）组员1朱云哲的总结：

我在本项目中主要负责搭建五级流水线，实现冒险检测处理与数据前推优化，参与了分支预测的实现。经过三周小学期的实验，我对计算机组成原理这门课的了解程度更加地深入，对CPU和计算机地底层更加感兴趣。通过实验，我更好地掌握了书本上的知识，从搭建一个基础的五级流水线，到实现具备各种完善功能的流水线，我的代码能力与知识运用都得了很好的锻炼。希望未来能够继续优化实验项目并参加相关比赛，锻炼计算机体系结构方面的能力。

1. 组员2郑梦仁的总结：

主要参与动态分支预测板块和数据前推优化的设计。动态分支预测在听老师授课过程中其实是能够理解的，并且知道其原理，但是用在设计上，在一开始的阶段确实有点手足无措，感到很乏力。但是静下心来，对整个预测过程进行多次模拟，并将双模的状态转换过程在图纸上进行预演，这个过程在脑海中愈发清晰起来。之后在编程设计时，便是信手拈来，可以很清晰的构建相应的模块，并且进行模块之间的连线。即使没有一次成功，在不断的debug中不断迭代更新，但是无论是编程能力或者是思维能力，我自己可以感觉到有不少的提升。此外，尽管主要针对动态分支预测模块，但是我还是得将整个数据通路以及相应的数据依赖与前推，流水级之间的信号流动，访存时发生命中或者缺失后需要整个流水线怎么运转等等内容完全通透。只有这样，我才能保证动态分支预测模块不出差错，可以稳定地适用于各种情况。尤其是针对上述提到的数据冒险以及分支预测结果的信号分析，这一点是不可忽视的。此外，流水级中一旦发生miss，那么需要进行内存的访问，那么这里的分支预测是可能会收到历史数据依赖的影响，因此访存的stall延迟信号优先级的最高的。再一次说明了流水线CPU中历史的结果可能对后续的指令产生一定的影响，此外进行stall也是为了保证流水级的有序性。针对前推模块的设计，在增加指令数目的情况下，需要在原来实验四的基础上进行一定的改进。例如，针对Jalr指令,需要计算目标地址需要rs1寄存器存储的数值，因此在得到ID/EX流水线寄存器之后，需要对可能传入adder模块的数据（rs1对应数值或者PC）进行数据的二选一筛选，那么这里有可能存在前推结果影响。因此，传入选择模块的rs1寄存器对应的数据一定要保证经过了前推信号的选择。另外，针对store指令由于待存储的数据可能依赖于之前执行的结果，因此需要对rs2存储的数据进行额外的数据筛选，再传入下一级流水线中。此外，为了确保EX阶段计算的结果正确性，在针对rs2寄存器对应数值和立即数imm的选择顺序中，我们使用前推信号对rs2数据筛选之后，再使用alusrcE信号对imm和寄存器数据的筛选，以确保是数据的正确性。

测试阶段，我进行了所有包含了有关分支指令的测试，仿真结果符合人为的验算模拟推进，并且能够吻合双模预测的正确结果。唯一的不足之处在于仿真时间的限制，好在我手动在Cache中初始化一系列的数值。针对综合测试1（快速排序），是一个非常不错的可以用来测试分支模块正确性的程序，在手动设定Cache中的数组的数值后，可以较好地模拟出快速排序的部分过程（1000时间限制问题），并且所有的分支过程都是正确的。针对前推的正确性验证，我通过了所有的测试集。

另外，值得改进的地方在于可以设计出类似于数据存储中的Cache也即是Icache以及相应的内存Imemory，不好评价这针对于我们的测试集这种小型的程序会产生什么样的效果，但是其设计理念仍然是考虑了访存时间优化，旨在提升CPU的性能。

综合来看，在这个设计过程中可以很清晰地感受到五级流水的层次化，并且知晓流水级之间的相互依赖关系。整个设计过程会时常感到迷茫，但是经过多次的debug以及仿真图像分析，可以知道整个流水级CPU就是通过4个流水级寄存器建构起来的(个人觉得可以5个，也即是在4个的基础上将regfile寄存器也视为一个流水线寄存器)，相邻指令与指令之间相差一个流水级，同一指令在流水级之间持续流动，直至所有阶段的过程全部完成。归根结底，我们的工作其实是在流水级传递过程中，处理好各种数据与信号，处理好各种数据依赖与矛盾，处理好指令的正确执行顺序，处理好数据的访问与储存带来的影响，并且尽可能地以较优的方式方法促进程序快速高效地运转。与小组合作完成此次硬件综合设计，我体会到了真正的团队合作，我对计算机体系架构有了更深层次的了解，对中央处理单元的运行模式有了更加深刻的认知。

1. 组员3胡芸明的总结：

我主要参与了本实验Cache的设计。本实验最初实用的Cache并非我们现在所看到版本，而是一个单端口WB+WA的Cache，通过对策略的推理我们发现内存架构的问题并综合考虑硬件资源消耗以及时间的关系，提出了折衷的解决办法，最后收获了不错的效果。通过本次实验，我对与Cache设计的理解又上升了一个层次，很多学习过程种产生的疑惑都得到了解决。同时，我还体会到了小组分工合作的作用，通过我们小组三人的合作，我们完成了很多一个人无法完成的任务，收获了很好的合作效应。总的来讲，本次实验我收益颇多，不仅有对学习内容的进一步认识，也有对于团队合作能力的提升。

七、参考文献

[1] Hennessy, J. L., & Patterson, D. A. Computer Architecture: A Quantitative Approach[M]. Morgan Kaufmann, 2017: 1-51.

[2] Yeh, T.-Y., & Patt, Y. N. Two-Level Adaptive Training Branch Prediction[J]. Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO-24), 1991: 51-61.