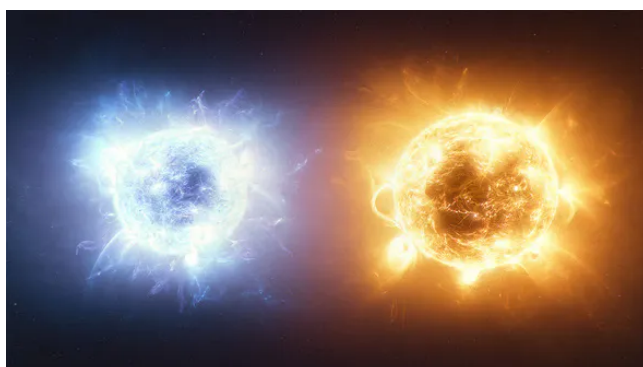


“龙芯杯”第六届全国大学生计算机系统能力培养大赛

重庆大学“所以延迟槽会消失对不队”队

CDIM 项目 决赛设计报告



陈泱宇

cyy@cyysf.name

李燕琴

maxpicca@qq.com

王梓宇

925136384@qq.com

张翀

20194159@cqu.edu.cn

2022 年 8 月

目录

第 1 章 概述	1
1.1 项目背景	1
1.2 名词解释	1
1.3 项目概述	1
第 2 章 CPU 设计方案	2
2.1 双发策略	2
2.2 数据通路设计	3
2.3 冲突处理	4
2.4 CP0 寄存器设计	5
2.5 中断和异常	5
2.6 缓存设计	5
第 3 章 差分测试测试框架	9
3.1 框架介绍	9
3.2 SoC-Simualtor	9
3.3 CEMU	9
第 4 章 操作系统支持	10
4.1 Linux	10

第 1 章 概述

1.1 项目背景

本项目依托于龙芯杯提供的 FPGA 实验平台、Soc 工程环境以及基准测试程序，设计并实现了一个部分兼容 MIPS32 体系结构的小端序 CPU，名为 CDIM（CQU Dual Issue Machine），其能成功通过龙芯杯提供的功能测试、性能测试、系统测试，具有较完善的运算处理、AXI 访问、异常处理、中断响应等功能，并能够运行 u-boot 引导程序、uCore 操作系统和 Linux 操作系统等。

1.2 名词解释

本项目中可能用到的一些名词缩写及其解释如表 1.1 所示。

表 1.1: 名词缩写和解释

名词缩写	全称	解释
MIPS	Microprocessor without Interlocked Pipeline Stages	无内部互锁流水级的微处理器
SOC	System On a Chip	片上系统
MIPS	Microprocessor without Interlocked Pipeline Stages	无内部互锁流水级的微处理器
SOC	System On a Chip	片上系统
CPU	Central Processing Unit	中央处理器
ALU	Arithmetic Logic Unit	算术逻辑单元
GPR	General Purpose Register	通用寄存器
CP0	Co-Processor 0	协处理器 0
BRAM	Block Random Access Memory	块随机访问存储器
FIFO	First In First Out	先进先出
RAW	Read After Write	写后读
WAW	Write After Write	写后写
WAR	Write After Read	读后写

1.3 项目概述

CDIM（CQU Dual Issue Machine），采用对称双发射五级顺序流水线的设计，支持指令 FIFO、分支预测、指令缓存和数据缓存等特殊单元，以提升系统性能。其中双发射，采用对称双发逻辑以充分保证双发率；五级顺序流水线由取指（Instruction Fetch）、译码（Instruction Decode）、执行（Excute）、访存（Memory access），写回（Write Back）五个阶段组成；指令 FIFO 可以隔离取指阶段和后续阶段，以实现高效取指的作用；指令缓存和数据缓存均采用二路组相联和突发传输的设计，单路均为 4KB 以匹配 TLB 页面要求，其中指令缓存一行为 64bit 以适应双发取指要求，数据缓存一行为 32bit。**此部分尚需完善**。此外，CDIM 还支持 U-Boot 引导程序，并基于该引导程序，成功运行 uCoreh 和 Linux 操作系统。

第 2 章 CPU 设计方案

CDIM 采用对称双发射顺序执行，共有 5 级流水。CDIM 的数据通路示意图如 2.1 所示，其中红线部分为 master path，蓝线部分为 slave path。在对称双发射中，master 和 slave 的主要区别在于前者先于后者执行，在处理异常、提交等事宜时会被优先处理。当然，上述的“对称”双发射，只是相对于只支持 ALU 指令双发的非对称逻辑而言，CDIM 的设计中 slave path 支持的功能和 master path 旗鼓相当。但严格来讲，并不是绝对对称，在处理跳转指令、例外指令、写 TLB 指令等这类可能刷新流水线的指令时，需要优先在 master 处理，这在后续的双发策略中会详细说明。

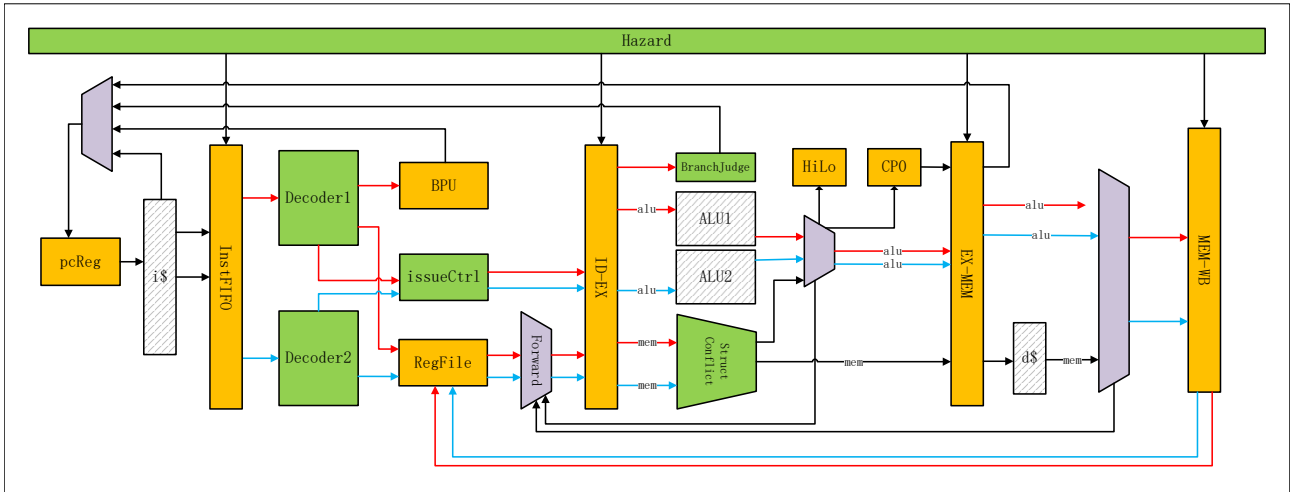


图 2.1: 数据通路设计图

2.1 双发策略

在双发射的 CPU 设计中，双发策略会影响整个数据通路的设计，故我们先来介绍一下 CDIM 的双发策略，即分情况讨论 slave 是否发射。其中在提供的代码中，issueCtrl 模块解释了 CDIM 的整个双发策略。

1. **only_master 类指令**: 即只能放在 master 发射，且 slave 不发射指令。对于可能触发例外的指令（如 MTC0、SYSCALL、ERET、BREAK、TLBWI 等指令），会导致 PC 跳转到异常处理地址或其返回地址处，进而刷新整个流水线。对于这类指令，如果在 slave 中检测到这类指令，则暂停 slave 发射，将其延迟到下一周期的 master 发射；如果在 master 中检测到这类指令，也会暂停 slave 发射，避免不必要的流水线刷新。
2. **only_in_master 类指令**: 即只能放在 master 发射，且 slave 可发射指令。对于跳转指令，在一定条件下会跳转到其他地址，并刷新整个流水线。和 only_master 不同的是，跳转指令会涉及到延迟槽的处理。故如果在 slave 中检测到这类指令，则暂停 slave 发射，将其延迟到下一周期的 master 发射；如果在 master 中检测到这类指令，若满足 slave 发射条件，则发射该条指令（延迟槽），若不满足，则在下一周期的 master 中发射延迟槽数据。
3. **数据冲突**
 - (a). **RAW 冲突**: 检查 master 和 slave 之间是否存在 HiLo、CP0、GPR 等寄存器的 RAW 情况，若存在，则 slave 不发射。
 - (b). **load to use 冲突**: 检查 slave 读到的寄存器是否是当前在 E 阶段执行的读存指令的结果，若是，则 slave 不发射。
4. **结构冲突**: 由于设计的乘除单元、访存单元只有一个，故二者会产生结构冲突。若 master 和 slave 同时需要占用这类单元，则 slave 不发射。

5. **数据有效性**: slave 发射的前提是 master 必须要发射或当前第二条指令有效。若遇到 master 暂停发射或指令 FIFO 为空的情况下, slave 不发射。

2.2 数据通路设计

2.2.1 取指阶段

为了保证双发射能够正常执行,第一要素便是“一次取指能够取回多条指令”,才能保证传到 D 时,至少有两两条指令可以准备用于双发射。指令 Cache 一行 64bit 的设计,也正适应了双发逻辑中取指的特点,使得 CDIM 传入指令 Cache 一个 PC 地址,可获得该 PC 及 PC+4 对应的两条指令。值得注意的是,为了简化 Cache 命中逻辑(即不存在跨 Cache Line 访问 Cache 的情况),传回数据中,并不是所有 PC 都会传回 2 条指令,故我们添加了 inst_data_ok 信号表示取回的指令是否有效。CDIM 的设计中,一行共 2 字,如果 PC 索引到该行的最后一个字,Cache 只传回 1 条指令,inst_data_ok2 置为 0;除此之外,传回两条数据,inst_data_ok1 和 inst_data_ok2 均置为 1。

出于双发策略中,slave 并不是每次都会发射,故取回的数据需要缓存下来,以用于下次使用,节省 Cache 访问时间。CDIM 中,使用指令 FIFO,将传回的有效指令缓存下来。使用指令 FIFO 还有一个好处是可以分割取指和后续阶段,即取指不受后续阶段产生的阻塞影响(但为了保证结果的正确执行并简化控制逻辑,后续阶段仍然会受到指令 Cache 产生的 stall 影响)。其中,如果指令 FIFO 放满,则暂停前段取指;如果指令 FIFO 为空,则暂停后续发射。

2.2.2 译码阶段

Instruction Decode 阶段中,主要完成以下操作:

- **译码**: 从 FIFO 中读出指令,立即进入译码模块。译码模块中,会分析该指令的功能,并分配控制信号。
- **分支预测阶段**: 为了减少分支跳转带来的流水线刷新数量,我们添加了静态分支预测单元,利用 pc 低位记录跳转数据,利用传统 2bit 策略更新跳转数据的记录。在 D 阶段获取到指令后,立即根据该指令所在 PC 进行预测,其中为了减少译码带来的延迟,分支预测单元内部单独进行分支译码,若是分支指令,即可取出预测结果并更新取指阶段的 PC。
- **发射判断**: 译码结束后,需要进入 issueCtrl 模块进行发射判断,发射判断结果会返回到指令 FIFO 中,控制 FIFO 的读指针的增量。
- **数据准备**: 读取 regfile 中对应的 GPR 数据,准备好数据后进入 Excute 阶段执行(在 CDIM 中,regfile 具有 4 个读端口和 2 个写端口)。其中为了在执行阶段的开头即可直接使用准确的数据,我们将数据前推单元放在了译码阶段的末尾。值得注意的是,为了减少前推带来的延迟,译码阶段中需要 GPR 数据的模块,均使用 GPR 直接读出的数据,而不是前腿数据(如 JR 类指令目标地址的计算),若此时存在 RAW 冲突,则需要推迟到执行阶段进行计算。

2.2.3 执行阶段

Excute 阶段中,主要完成以下操作:

- **跳转处理**: 在译码阶段中,Branch 指令需要在执行阶段中判断跳转是否成功,并将判断结果送回分支预测单元进行数据更新,更新此时取值阶段的 PC 并刷新流水线。同理,如果是译码阶段遇到数据冲突的 JR 类指令,其在本阶段获取到准确的目标地址后,更新此时取值阶段的 PC 并刷新流水线。
- **ALU 计算**: 可以处理单周期运算指令和多周期运算指令。其中,多周期运算指令包括乘除法指令。CDIM 中乘法指令需要 2 周期、除法指令需要 35 周期,且在运算完成前会产生 stall 信号,阻塞 D、E、M、W 四个阶段。其中,为减少资源占用,流水线中只提供一个乘法器和除法器,故 master 和 slave 在使用时需要

进行仲裁。同时获取到乘除结果后，即在执行阶段写 HiLo 寄存器，以期在访存阶段即可获取新的 HiLo 数据值，避免了 HiLo 数据的前推。

- **访存仲裁及其地址计算：**由于 data cache 是 BRAM 结构，需要一个周期取出数据，我们需要在 E 阶段将访存信号传递给 Data Cache。因为 master 和 slave 都支持访存（但不会同时访存），故在传递访存前，需要用 StructConflict 模块进行仲裁；并保证访存阶段的刷新信号置 0 且使能信号置为 1，以使访存信号能由 E 阶段正常传到 M 阶段，保证访存阶段数据的正常传回。同时，为了减少访存地址计算经过的路径，我们分离了执行阶段中的 ALU 路径和 MEM 路径（如图 2.1 所示），单独利用 $\text{base(rs value)} + \text{offset(immediate value)}$ 计算访存地址。同时根据访存地址及其比特要求对传给 Data Cache 的地址进行地址错例外判断（AdEl 和 AdEs 判断）。
- **TLB 地址转换：**此部分尚需完善。TLB 地址处理上，在上一级获取到地址后，在 E 阶段访问 TLB 获取。
- **异常汇总：**由上述可以看出，在执行阶段，可以获取从任何指令涉及到的所有异常信号，故我们在执行阶段汇总异常信号，并更新 CP0 寄存器，以期在访存阶段即可获取新的 CP0 数据值，避免了 CP0 数据的前推。

2.2.4 访存阶段

Memory Access 阶段，主要完成以下操作：

- **访存控制：**虽然在 E 阶段传递给了 Data Cache 基本的访存信号，告知 Data Cache 需要准备对应地址的数据；但是并没有告知需要取出的数据的比特数。E 阶段汇总时若发现异常，则传到 M 阶段后，会将 Data Cache 的握手信号 data_sram_enM 将会置为 0，阻止 Data Cache 将错误的地址传到 AXI 总线上。反之，则正常发送 data_sram_enM 信号，并得到传回的数据，再通过 StructConflict 模块，将数据交付到对应 path（master 或 slave）上。
- **写回数据选择：**master 和 slave 两条路径上的计算结果（alu）和访存结果（mem）的选择。
- **一级 TLB 未命中的处理：**此部分尚需完善。

2.2.5 写回阶段

Write Back 阶段，主要执行写回 GPR 请求。若此时读取 GPR 的地址，恰好是写 GPR 的地址，需要进行写前递操作。

2.3 冲突处理

2.3.1 数据冲突

- **RAW 冲突：**当 master 或 slave 需要读取的数据是已经发射且进入后续流水阶段的指令，会产生 RAW 冲突。本设计将需要的数据均前推到 D 阶段，前推后立即进入触发器，避免前推数据所在关键路径过长。其中，若需要读取的数据是当前 E 阶段的访存数据，则需要进行阻塞以等待访存数据在 M 阶段返回。若需要读取的数据是当前 E 阶段的计算结果、或 M 阶段的计算结果、或 M 阶段的访存结果，则通过 forward_top 模块前推到 D 阶段。
- **WAW 冲突：**当同周期的 master 和 slave 写入寄存器的地址一致时，会产生 WAW 冲突。因为 slave 是 master 后一条指令，故 regfile 更新时，会优先写入 slave 的结果。
- **WAR 冲突：**由于本设计为顺序流水线，故不存在 WAR 冲突。

2.3.2 结构冲突

当 master 和 slave 同时需要访存或占用乘法器或占用除法器时，会产生结构冲突。本设计中，结构冲突的指令只发射 master 一条，保证 CPU 正确执行。

2.3.3 控制冲突

当遇到跳转指令时，会产生控制冲突。MIPS 指令系统中，跳转指令包含延迟槽。本设计中，若产生跳转（译码阶段或执行阶段），则根据发射情况判断当前 fifo 中的第一条读数据是否是延迟槽数据。如果是，则需要判断当时是否有阻塞信号，若没有则直接执行该第一条读数据，若有则需要保留该数据到指定 `delayslot_data` 寄存器，以备阻塞取消后使用；此外跳转会引起流水线的刷新，但需要保证延迟槽数据能够不被刷新并正常流入下一阶段执行。如果不是，则正常刷新流水线即可。

2.4 CP0 寄存器设计

此部分尚需完善。

2.5 中断和异常

为使处理器的功能完整，CDIM 支持异常处理。根据 MIPS 规范要求，CDIM 支持精确异常，即出现异常后，准确记录发生异常的指令地址，并存放在 EPC 寄存器中以待异常返回时使用；且保证发生异常前的所有指令正常提交；发生异常的指令及其之后的指令不提交，并跳转至异常处理程序入口进行异常处理。CDIM 中支持的指令优先级顺序如下：

此部分尚需完善。TLB 异常陈述

1. 中断例外：包括硬件中断、软件中断和计时器中断。
2. 地址错例外（取指）：PC 地址未对齐四字节。
3. 保留指令例外：当执行一条未实现的指令时，触发保留指令例外。
4. 自陷例外、系统调用例外：执行到 Syscall、Break 等自陷指令时。
5. 整型溢出例外：执行 ADD, SUB, ADDI, SUBI 等指令发生溢出时。
6. 地址错例外（数据访问）：访问数据的地址未对齐，包括 AdEl、AdEs 两类错误。
7. 地址转换异常：访问 TLB 时，TLB 表中没有有效的转换对应项可用时，会发生地址转换异常，包括 TLB 重填异常、TLB 无效异常（有 load 页无效、store 页无效、modify 页无效等）等。

2.6 缓存设计

CDIM 中，设计实现了指令 Cache 和数据 Cache，且均为虚拟索引物理标签（VIPT）记录，二者的突发传输数量、大小、组相联数量均为可配置。其中数据 Cache 支持写缓冲，以减少写回的阻塞数量。

2.6.1 内存管理

MIPS 指令系统对操作模式和存储访问类型进行了严格的限定，将整个 4 GB 大小的虚地址空间划分为 5 段，如图 2.2 所示。

虚拟地址到物理地址的翻译上，共有两种方式：一是直接地址翻译，对应图中 Unmapped 段，翻译较快，延迟较少；二是映射地址翻译，对应图中 Mapped 段，需要访问 TLB 表项，延迟较大。在 CDIM 的缓存设计中，通过多级 TLB 的方式，减少映射地址翻译模式的时延和缺失率，以在实际地址翻译中，可以综合使用直接地址翻译和映射地址翻译。

缓存使用上，指令 Cache 和数据 Cache 均支持 Uncached 段的单独处理，这样发送 AXI 请求时，只需要做指令和数据请求的两路仲裁即可，极大简化仲裁逻辑。

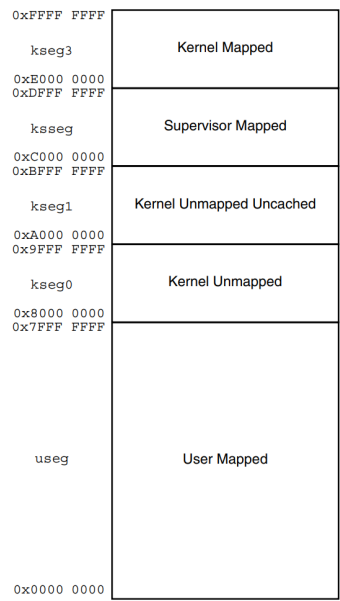


图 2.2: 虚实地址映射关系图

2.6.2 缓存结构

CDIM 的缓存结构和资源使用如图2.3所示，指令 Cache 默认为二路组相联且一路 4KB，一行 data 共 2 字 64bit，一行 tag 对应 8 行 data；数据 Cache 默认为二路组相联且一路 4KB、一行 data 共 1 字 32bit，一行 tag 对应 16 行 data。因为 Bram 读取数据时需要一周期后返回数据，故我们提前发送了下一个访存地址（指令 Cache 对应 pc_next，数据 Cache 对应 E 阶段的访存地址）进行提前查询。当到达对应的访存阶段时（指令 Cache 对应 F 阶段，数据 Cache 对应 M 阶段），在根据上一周期的命中情况判断能否同周期获取对应数据，以及是否发送 AXI 请求。在具体访问 AXI 时，Cached 请求支持突发传输，默认为 16 字突发（即一个 tag 对应的总字数）；Uncached 请求默认不支持突发传输。

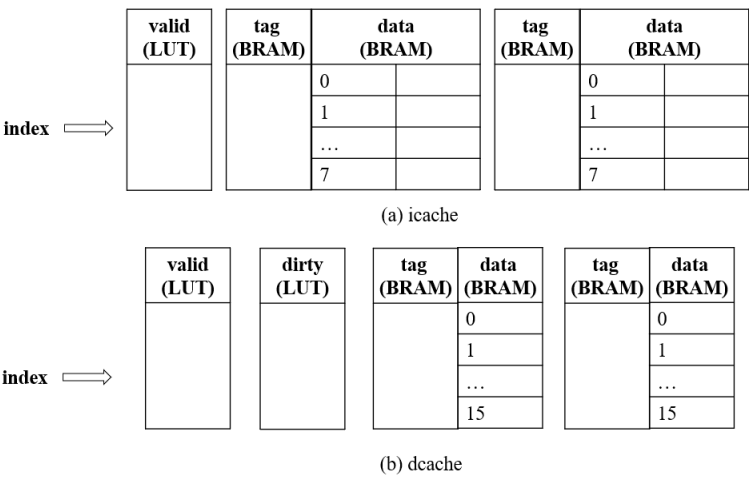


图 2.3: CDIM 缓存结构示意图

2.6.3 CACHE 指令

在 MIPS 指令系统中, CACHE 指令要求清除特定地址或索引位置的数据缓存。由于清除“索引位置”已经包含了清除“特定地址”, 故 CDIM 的设计中, Data Cache 操作全部实现为 Index Writeback Invalid, Instruction Cache 操作全部实现为 Index Invalid, 这使我们能够在满足运行操作系统所必备执行修改后的指令以及刷新 DMA 缓冲区的情况下, 简化 Cache 的逻辑, 使 CPU 能够实现更高的频率。

2.6.4 状态机

Cache 的控制实现上, 我们采用状态机控制的方式, 状态转换的情况如图2.4所示, 其中, Data Cache 包含写回操作, 故比 Inst Cache 多一个“CACHE_WRITEBACK”状态, 即图中绿色部分。

各个状态的含义如下。

- **IDLE**: Cache 空闲状态, 进行数据提前读取、地址 L1TLB 查询或直接翻译。
- **TLB REFILL**: L1TLB 缺失时, 查找 L2TLB、处理可能遇到的 TLB 重填异常或 TLB 无效异常、L2TLB 命中后更新 L1TLB。
- **UNCACHE**: 向 AXI 发送读请求或写请求(无突发传输), 并等待 AXI 访存结束。
- **CACHE REPLACE**: cache 缺失后, 若没有脏位, 需要重新向 AXI 发送读请求(突发传输), 访存并替换对应路的缓存行。其中替换原则采用 1bit 替换方式。
- **CACHE WRITEBACK**: 当需要清除或替换缓存行时, 若存在脏位, 则需要将脏数据写回, 即向 AXI 发送写请求(突发传输)。
- **SAVE RESULT**: 当 Cache 因为 TLB 缺失或 Uncached 导致 Cache 暂时无法正常访问数据时的保留状态。

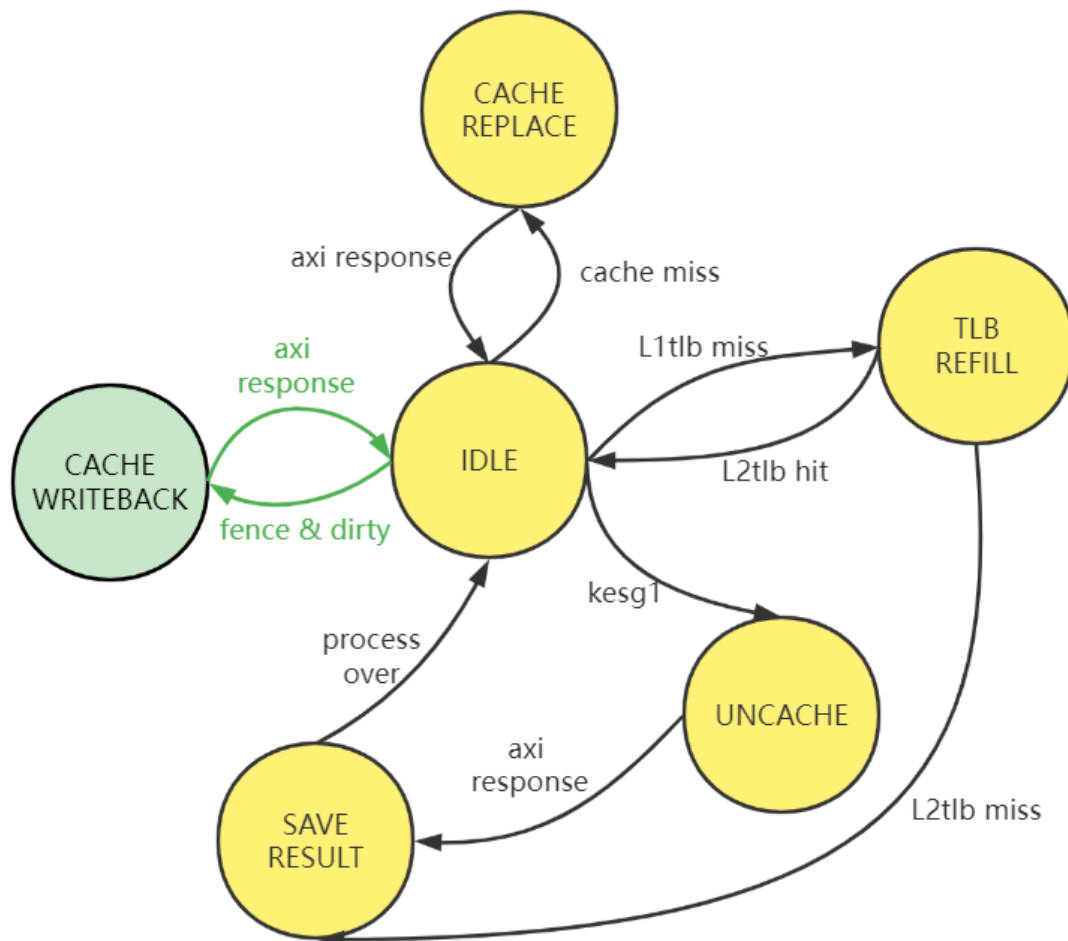


图 2.4: CDIM 缓存状态机示意图

2.6.5 写缓存

在 CDIM 的设计中，对于 Uncached 的 AXI 写请求，我们设置了 Store Buffer，即一个写请求队列，将 AXI 写请求数据缓存下来，再逐个写回，该期间如果没有 Cached 的访问 AXI 请求，则流水线不会因为写回 Store Buffer 数据而阻塞，提高了流水线的执行效率。

值得注意的是，为了保证读写数据的准确，当 Cached 数据需要访问 AXI 时，若这个时候 Store Buffer 仍然在占用 AXI 通道，则需要等待 Store Buffer 写完后，高速缓存才能去处理这个 Cached 数据请求，这个阶段流水线会因为该 Cached 数据请求得不到响应而阻塞。

第 3 章 差分测试测试框架

3.1 框架介绍

为了提高处理器开发效率，我们自己开发了差分测试框架，分为 SoC-Simulator 与 CEMU（CQU Emulator）两个部分。

3.2 SoC-Simualtor

SoC-Simulator 是我们开发的一个基于 Verilator 的 SoC 仿真框架。它使用 C++ 语言编写，通过软件实现了 AXI Slave 设备功能。其内置了 MMIO Crossbar、UARTLite、UART8250、NSCSCC confreg 等设备，能够满足运行龙芯杯功能测试、性能测试、系统测试，启动 uCore 和 Linux 的需求。

此外，SoC-Simulator 还能够支持 AXI 延迟的自定义，在我们开发处理器期间经过了多次测试，在 AXI 延迟设置为 23 周期的情况下，SoC-Simulator 仿真的性能得分与上板真实成绩误差在 1 分以内。

同时得益于 Verilator 的仿真效率，使用 CDIM 在当前主流的台式机（AMD Ryzen 7 5800X）上实现了高速测试，2 秒完成功能测试（含 trace 比对），7 秒钟完成性能测试并输出 CP0 count 寄存器的结果。同时，在 SoC-Simulator 上运行 uCore 可以 10 秒内进入 shell，运行 Linux v5.19 配合我们基于 defconfig 修改的配置大约 4 分钟开始执行 init 进程，这大大方便了我们修改 RTL 后及时得到 IPC 反馈与问题定位与调试工作。

3.3 CEMU

CEMU(CQU Emulator) 是我们开发的一个 CPU 模拟器，目前支持了 RV64IMA 与 MIPS Release 1 指令集，LoongArch32 的支持正在实现中，我们使用它与 CDIM 进行差分测试，

由于 CEMU 与 SoC-Simulator 都是我们自行开发的，因而采用了完全相同的外设接口（C++ 的抽象类），从而 CEMU 与 SoC-Simulator 能够运行在完全相同的 SoC 环境中，并通过处理器 RTL 输出的 CP0 Count、CP0 Cause、CP0 Random、是否中断等信号对 CEMU 的各寄存器状态进行同步，这帮助我们在调通 uCore 后仅 2 天时间就完成了 Linux 的调试。

第 4 章 操作系统支持

4.1 Linux

我们的处理器符合 MIPS Release 1 规范，实现了除 Branch-Likely 指令外的所有指令，并实现了 CP0 中所有 MIPS 规范在实现了 TLB 和 Cache 的情况下所要求的寄存器，这让我们运行 Linux 成为可能。

最终经过调试，在 SoC-Simulator 与 MegaSoC 上都成功运行起了最新 Linux 内核主线（v5.19），且不需要除关闭 Branch-Likely 指令外的特殊移植。

参考资料

- [1] MIPS® Architecture For Programmers I, II, III. Imagination Technologies LTD.
- [2] 计算机组成与设计: 硬件/软件接口. David A.Patterson
- [3] Sirius 设计文档. 于海鑫, 尹思维
- [4] 自己动手写 CPU. 雷思磊