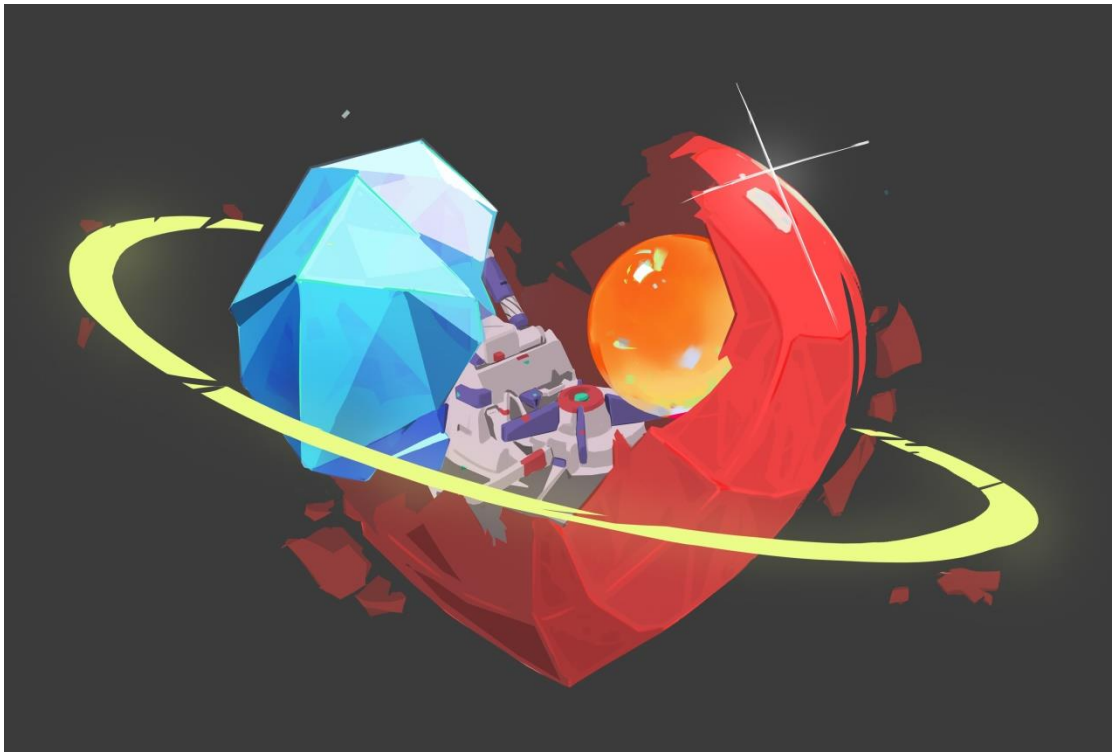
ZirconMIPS 设计报告

哈尔滨工业大学（深圳）1 队



朱旗 尚奕扬 邓起源 丁浩卓

2021/8/16

目录

第一章 项目概览

1.1 总述

本项目是一个基于 MIPS32 指令集设计的 CPU，支持 MIPS32 指令集的一个子集。CPU 采用顺序双发射六级流水线结构，包含一级 ICache 和 DCache，通过 AXI 接口和外设通信。

1.2 性能

我们最终提交的 CPU 主频为 95MHz，IPC 比值为 38.067。

1.3 指令集

我们实现了大赛要求的所有指令。除此之外，还加入了自陷指令、非对齐访存指令、链接加载指令、条件加载指令和特权指令。

1.4 结构

CPU 设计为双发射顺序执行，共有六级流水线。为了尽可能消除跳转指令对 CPU 性能的影响，我们实现了基于历史跳转情况的**动态分支预测**。为了最大限度地减小跳转指令带来的损失，发送给 ICache 的地址信息是 BPU 和 PC 竞争后的结果。

1.5 CP0 与异常处理

我们实现了大赛要求的所有 CP0 寄存器并实现了精确异常，即在流水线上执行的指令发生异常时，并不会立刻处理，而是把它记录下来，送 MEM 阶段统一处理。

1.6 分支预测

我们所使用的分支预测基本策略是在取址阶段送入 ICache 的指令地址值同时传入分支预测模块，当正确预测时流水线将不会有周期损失。

1.7 Cache

我们的 ICache 采用 16KB 四路组相联状态机设计，DCache 采用 8KB 二路组相联状态机设计。

1.8 AXI 总线

我们实现了对外的 AXI 的总线协议。其中 AXI 总线部分分为对外的接口的实现度写猝发的标准 AXI4 协议总线模块和实现了双 Cache 仲裁以及 UnCache 处理的读写双通道的 AXI-Cache 转接桥。实现了一个简单的写缓存逻辑。

1.9 外设

我们基于 LCD 触摸屏和 VGA，设计了精简的界面和几个用于展示的应用程序，组成了外设系统。

1.10 设计变更说明

在初赛后，我们对 ICache，DCache 和 AXI 总线进行了优化，使 IPC 从初赛提交时的 33.991 提升至最终的 38.067。同时加入了 TLB。

第二章 CPU 设计

2.1 性能

我们最终的 CPU 频率为 95MHz，IPC 比值为 38.067，以下为我们的性能提升记录。

表 2.1 性能分记录

时间	主频/MHz	性能分	备注
2021.7.21	85	29	初版 CPU，加入指令 Cache 与数据 Cache
2021.7.28	85	34.8	优化 CPU 发射结构与 Cache 状态机
2021.7.30	85	41	加入第一版分支预测
2021.7.31	85	48.22	加入第二版分支预测
2021.8.3	85	52.459	重构发射逻辑，简化数据 Cache 状态机
2021.8.6	80	51.956	加入 TLB
2021.8.8	85	56.238	将 Cache 的 bank 数由 4 增加到 8
2021.8.12	88	60.889	优化 CPU 发射结构
2021.8.13	90	65.096	实现 AXI 写缓冲
2021.8.15	95	71.984	再次优化 CPU 关键路径

2.2 指令集

我们实现了比赛要求的全部指令，并额外实现了一些指令。现按照功能分类列出：

表 2.2 实现的指令

算术运算指令	ADD, ADDI, ADDU, ADDIU, SUB, SUBU, SLT, SLTI, SLTU, SLTIU, DIV, DIVU, MULT, MULTU, MUL
逻辑运算指令	AND, ANDI, LUI, NOR, OR, ORI, XOR, XORI
移位指令	SLL, SLLV, SRA, SRAV, SRL, SRLV
分支跳转指令	BEQ, BNE, BGEZ, BGTZ, BLEZ, BLTZ, BLTZAL, BGEZAL, J, JAL,JR, JALR
数据移动指令	MFHI, MFLO, MTHI, MTLO, MOVE, MOVN
自陷指令	TGE, TEGU, TLT, TLTU, TEQ, TNE, TGEI, TGEIU, TLTI, TLTIU,TEQI, TNEI
访存指令	LB, LBU, LH, LHU, LW, LWL, LWR, SB, SH, SW, SWL, SWR
特权指令	BREAK, SYSCALL, ERET, MFC0, MTC

2.3 CPU 架构

2.3.1 总体架构

我们的 CPU 采用了顺序双发射的六级流水线结构，各部分功能简述如下：

取址 1 阶段（IF1）计算出当前 PC，并完成和分支预测模块的竞争。得到的指令地址送入指令 Cache。

取址 2 阶段（IF2）将指令 Cache 取得的指令及其 PC 送入 InstBuffer。

译码发射阶段（ISSUE）从 FIFO 中取得指令，进行译码后决定发射条数，并从寄存器中取出操作数。

执行阶段（EX）执行指令，决定是否跳转，并决定发送给数据 Cache 的请求。

访存阶段（MEM）从数据 Cache 中取出数据，并判断异常。

提交阶段（COMMIT）将指令执行结果提交给寄存器。

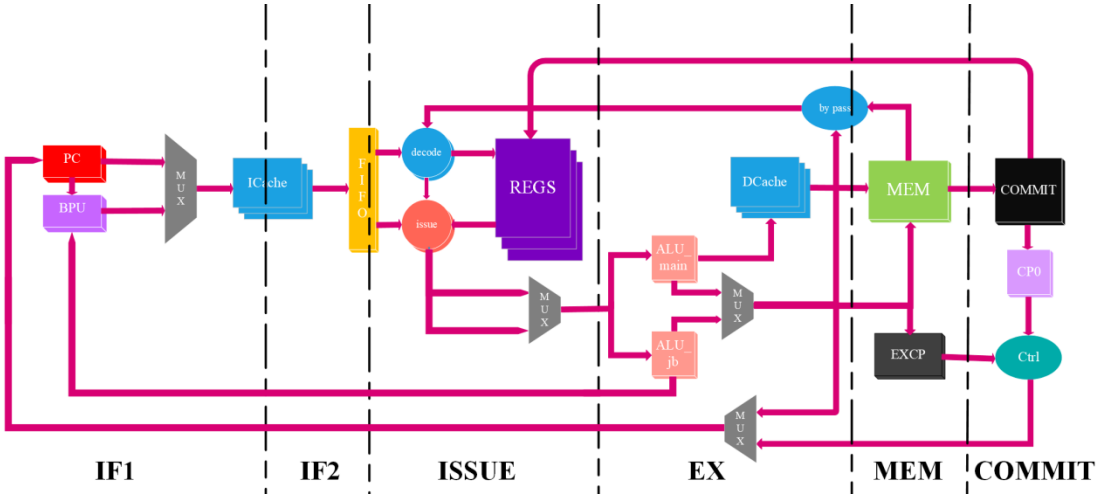


图 2.1 CPU 结构图

2.3.2 各阶段设计说明

取址 1 阶段：PC 向 ICache 发出请求信号，并将地址发给 ICache。默认 PC+8，当发生异常或者分支预测失败时，跳转到相应地址。当指令队列满或 ICache 发出暂停信号时，暂停取指。：

取址地址产生条件按优先级排列如下

(1)复位信号使 npc 置为初始地址 0xbfc00000。

(2)由异常造成的 flush 信号使 npc 的值置为 epc 寄存器的地址。

(3)由分支预测失败造成的 flush 信号：

a.实际发生跳转，npc 的值置为 ex 阶段送来的 npc_actual；

b.实际不跳转，npc 的值置为 ex 阶段送来的跳转指令的地址加 8。

(4)由指令队列满或 ICache 发出暂停信号使 npc 暂停。

(5)根据分支预测信息，若分支预测信息标记为跳转，根据跳转指令地址所在的位置：

a.若跳转指令地址在第二条，将 npc 置为 bpu 预测的跳转地址；

b.若跳转指令在第一条，将 npc 置为 bpu 预测跳转地址加 8。

(6)不发生跳转，npc 每次加 8。

取指 2 阶段：正常情况下，ICache 根据 pc 地址取出指令，将两条指令同时发送给指令队列并接收。

若延迟槽指令由 ICache 发送来时位于第一条，那么我们只接收第一条延迟槽指令，不接收第二条指令。

需要注意的是，跳转指令在发射时始终处于第一条的位置，我们要将预测信息始终与其绑定。

译码发射阶段：先对指令队列准备发射的指令进行译码，指令队列根据译码结果，判断能否双发射，然后发射指令。如果不能双发射，指令队列只弹出一条数据，且 id 阶段将清空原本的第二条指令。

为了进一步提高 IPC，简化发射逻辑，我们设计的发射逻辑如下：

对于两条数据相关指令，单发射；

对于乘/除法和访存指令，如果与其一同的指令是乘/除法、访存、CP0 指令，单发射；

对于访存指令，如果与其一同的指令是乘/除法、访存、CP0 指令，单发射；

对于 CP0 相关指令，单发射；

其余所有情况均可双发射。

执行阶段： 此阶段执行指令。在我们实现的所有指令中，只有乘法和除法是多周期指令，其余指令均可在一周期内实现。对于单周期指令可在一周期内得到结果并发送给访存阶段。对于乘除法指令，需要暂停流水线直至指令执行结束。乘法使用 Vivado 提供的 IP 核实现，需要三周期返回结果；除法使用试商法，需要 32 个周期。此外，对于访存指令，会向 DCache 发送数据。若命中，会在下个周期返回数据。

对于跳转指令，会在此阶段与分支预测发来的信息进行比对，若预测正确则不会清空流水线，否则会清空流水线，并将 pc 置为正确的 PC 值。

为了进一步提高流水线执行效率，我们在 EX 阶段实现了**指令的动态调度，即**通过合理调度指令的方法提高 IPC。考虑到双发射的情况，我们将 EX 阶段分成两个 ALU。第一个 ALU 可以执行除跳转指令之外的所有指令，第二个 ALU 可以执行简单指令（逻辑运算、移位以及除了乘法、除法以外的算术运算指令）和跳转指令。通过动态调度指令，我们成功地实现了更多指令的双发射。

以跳转指令为例，在发射阶段时，我们将跳转指令放在第一条，与延迟槽指令一同发射。在进入执行阶段时，进行调度：将跳转指令送入第二个 ALU，延迟槽指令送入第一个 ALU。执行阶段结束后再将两条指令对应的信息还原。需要注意的是：对于延迟槽指令是乘除法指令、mtc0 且分支预测失败的情况，要注意对 id_ex 流水线寄存器输出逻辑的控制；对于延迟槽是加载指令的情况，需要在 mem 阶段对从 DCache 读取到的数据进行相应的分配。

在实现优化过的发射和执行逻辑后，我们的 CPU 双发率得到显著提升：从不到 40%提升到了 62%，同时 IPC 比值也有明显提升。这说明我们的设计是成功的。

访存阶段：此阶段取出 dcache 发出的数据，并进行处理，送至写回阶段。此外还会进行异常比对。若发现异常则将异常信息传至控制模块和 CP0，清除流水线并进行异常处理。

写回阶段：此阶段写寄存器。包括寄存器堆、HI/LO 寄存器和 cp0 寄存器。

2.4 CP0

CP0 寄存器是 MIPS 指令系统中定义的一组独立的寄存器，MIPS 指令系统将与特权态相关的功能都定义在协处理器 0(Coprocessor0 ,CP0)中。

目前，CPU 实现的 CP0 寄存器如下：

表 2.3 实现了的 CP0 寄存器

Reg	Sel.	寄存器名称	功能定义
8	0	BadVAddr	记录最新地址相关例外的出错地址
9	0	Count	处理器时钟计数器
11	0	Compare	计时器中断控制
12	0	Status	处理器状态与控制寄存器
13	0	Cause	存放上一次例外原因
14	0	EPC	存放上一次发生例外指令的 PC
15	0	PRId	处理器标识符和版本
15	1	EBase	中断向量基地址寄存器
16	0	Config	处理器配置寄存器

2.5 TLB

为支持在 CPU 上跑起系统，我们加入了 16 路全相联的 TLB。

对于 TLB 相关的例外，我们在数据缓存将指令送入时将异常信息与对应指令绑定。对 TLB 指令的后一条指令打上重取标志，在访存阶段对该指令进行重取。对于地址映射，TLB 将 CPU 送出的地址进行转换，同时将页表项的属性位交给指令缓存，指令缓存将其与对应的指令绑定后送入指令队列。对于数据缓存送出的 TLB 异常信息，在数据缓存中用寄存器打一拍之后在访存中立即处理。指令、数据缓存在在得到 TLB 异常信息，通过页表项属性判断出存在异常时不进行任何工作。

对于 TLB 指令，我们通过让读写 CP0 相关的指令独占流水线以解决潜在数据相关的问题。由于我们的设计是固定的双取指，为解决 TLB 地址跨页的问题，交给指令缓存在工作中自行再次查询。

为提升频率，我们将 TLB 送出的页号延迟一拍再送给指令缓存。

2.6 异常与中断

我们实现了精确异常，具体方法为：在流水线上执行的指令发生异常时，CPU 并不会立刻处理，而是把它记录下来，送 MEM 阶段统一处理。

CPU 支持的中断包括软件中断和硬件中断，具体实现的中断如下表

表 2.4 支持的异常类型

ExcCode	助记符	描述
0x00	Int	中断（软件和硬件）
0x04	AdEL	地址错例外（读数据或取指令）
0x05	AdES	地址错例外（写数据）
0x08	Sys	系统调用例外
0x09	Bp	断点例外
0x0a	RI	保留指令例外
0x0c	Ov	算出溢出例外
0x0d	Tr	自陷例外

2.7 分支预测

我们所使用的分支预测基本策略是在取值阶段送入 ICache 的指令地址值同时传入分支预测模块，分支预测模块通过查找 BTB 表，读出对应位置存储的跳转方向和目的地址的信息，在跳转指令进入到指令缓冲队列阶段(FIFO)时，通过一个仲裁模块选择取值阶段的 PC 值和分支预测阶段得到的目的地址中的一个有效信息进入 ICache。引入动态取值后，正确预测时流水线将不会有周期损失，平均命中率约 89%。

目的方向预测：我们采用的是二位饱和计数器的经典方法。通过将二位饱和计数器的值存入 BTB 表中的高二位，在通过指令地址寻址的时候会同时将二位饱和计数器的值取出，我们采用的编码格式为：

strong_not_jump: 2'b11
weak_not_jump : 2'b00
weak_jump : 2'b01
strong_jump : 2'b10

这样一来跳转方向可以通过二位饱和计数器的值异或得到，而非使用判等符。

对于二位饱和计数器的更新策略，我们采用的是在执行阶段获得正确的跳转信息之后对分支预测进行更新。由于我们在对 BTB 寻址时采用了整个 PC 值的高位作为判断命中的 tag 而非 hash,实际上不会出现不同的两条指令对同一个位置查找的情况，所以并没有采用立即更新的策略，而是通过执行阶段返回的真实跳转信息进行更新。这样做的优势是能确保只有跳转指令才会进行预测，只是每一条新出现的跳转指令都很可能至少预测错误一次。而由于预测方向并没有采用局部历史跳转表，所以更新不及时的代价也比较小。其中，执行阶段返回的更新信息包含了跳转的类型，这样一来即使是预测失败但还是将 J 型指令和 JR 等必然跳转的指令便直接将二位饱和计数器置为强跳转。

目的地址预测：对于目的地址的预测，我们采用的也是经典的历史跳转地址缓存的方式，BTB 共有 512 项历史跳转地址，预测时将指令地址的拆分成索引部分和 tag 部分，实现方法类似 Cache。

在地址更新方面，并没有通过观察反汇编的代码，我们并没有加入 RAS 来专门区分 JR 指令，所以如果是不确定跳转地址的 R 型指令，我们会用最新一次跳转地址进行更新。

动态取指：由于分支预测的加入，引入新的延时槽相关问题。为了最大发挥分支预测的

性能，我们选择将送入 ICache 的指令地址由单独的一个指令地址多路选择器来仲裁究竟是使用分支预测得到的目的地址还是取值阶段的指令地址。由于双发射处理器设计时选择的是 ICache 一次固定读出 2 条指令，于是出现了三种情况：

（1）预测不跳转

（2）预测地址对应的第一条指令跳转

（3）预测地址对应的第二条指令跳转

上述的第一种种情况无需特殊处理，而第二种情况指令地址选择模块会将有效的 PC 值选择出来送入 ICache 和分支预测用于下一次更新，同时将取址阶段的 npc（next_pc）寄存器的值更新为目的跳转地址加 8，即目的跳转地址的下一次取址地址，而在译码阶段处理器会将取出来的两条指令中的延迟槽指令标记出来；但在第三种情况下，分支预测要需发出控制信号，让指令地址选择器先选择取值阶段送过来的指令地址，同时将 npc 更新为目的跳转地址，与此同时给 ICache 发送控制信号，ICache 会将这个控制信号保存至下两条条指令（此时延迟槽指令为取出的第一条指令）取出，并与指令一同发给指令缓存阶段，当指令缓存接收到这个信号，会只将第一条指令也就是延迟槽指令送入指令缓存。通过这种方式来解决第二条指令是跳转指令时延迟槽指令的问题。

2.8 Cache

Cache 在 ZirconMIPS 的架构中发挥着至关重要的作用，Cache 在 ZirconMIPS 中可以看作一个大的寄存器堆，能否以小的代价取得 CPU 需要的指令和数据是 Cache 设计的核心。虽然 ZirconMIPS 中 Cache 是由状态机实现的，但无论是 ICache 还是 DCache 都只会在发生缺失的情况下向 CPU 发起暂停信号，在取得了 CPU 需要的指令（或数据）后，Cache 会第一时间取消暂停信号并将指令（或数据）送往 Cache。而在连续命中的情况下 Cache 能够连续的接收地址并提供 CPU 需要的指令和数据，实现了 CPU 与 Cache 的深度耦合。

ZirconMIPS 中 Cache 作为 CPU 与主存进行数据交互的唯一通道，除了要承担 Cache 类的数据请求，还承担了 UnCache 类的数据请求。具体的实现是在 ICache 和 DCache 内都为状态机增了专门用于处理 UnCache 数据请求的状态。

ICache 为 16KB 的四路组相联，DCache 为 8KB 的二路组相联，块大小由初赛的 16 个字节增加到了 32 个字节，该优化对于频繁进行读取和存储指令的程序有明显的优化。例如 stream copy，在将块的大小翻倍后，ICache 的命中率由 97.740%提升到 98.60%。DCache 的命中率由 88.21%提升到 93.98%，在相同频率下 stream copy 的性能分增加了 10 分。

在替换策略方面，**ICache 采用 FIFO 的替换策略，DCache 采用伪 LRU 替换策略**。ICache 需要替换时会选择最早进入 ICache 的块，DCache 需要替换时选择近期没有使用的块。

由于 ICache 为四路组相联，容量较大，经过我们的实际测量，采用伪 LRU 算法很难装满一行中的四路数据。此外四路组相联的伪 LRU 算法因为较为复杂，所以相较 FIFO 会占用较多的寄存器资源，会在 Vivado 实现时浪费资源，从而限制了 CPU 的频率。此外在 88MHz 的频率下经过我们的实际测量，两种方法的性能分数差小于 0.1。所以我们将 ICache 的替换策略由初赛时的伪 LRU 改为了最终的 FIFO。而 DCache 为二路组相联，容量较小，容易装满且二路组相联的伪 LRU 算法较简单，故我们 DCache 的替换算法沿用初赛时的伪 LRU 算法。

DCache 采用的主存写策略为写回，DCache 在遇到写不命中的情况，会先从主存读数据保存在 DCache 后再进行写入。

2.8.1 ICache

ZirconMIPS 采用了状态机 ICache，在命中的时候能够连续的从 CPU 的取值阶段接受指令地址，并在译码阶段不停的将指令写入指令缓存队列。状态机包括六个状态，分别是：

表 2.5 ICache 状态机的状态

1	MIDLE	当前 CPU 没有任何请求发往 ICache 或者 ICache 完成对指令缺失、UnCache 类请求和清空流水线请求的处理
2	LOOKUP	ICache 当前收到请求并得到了它的查询结果
3	REFILL	ICache 当前处理的请求缺失，且正在等待 AXI 总线的读取结束标志信号
4	SEARCH	ICache 当前处理的请求对应的两条指令位于两个数据块并且得到了第二条指令的查询结果
5	MISS	ICache 当前处理的请求对应的第二条指令缺失，且正在等待 AXI 总线的读取结束标志信号
6	UNCACHE	ICache 收到 UnCache 类的请求，且正在等待 AXI 总线的读取结束标志信号

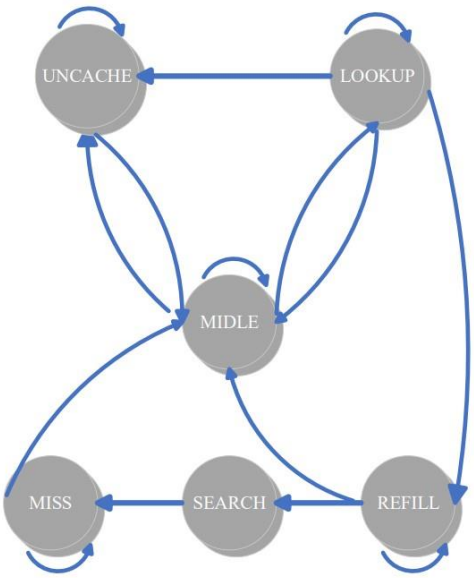


图 2.2 ICache 状态机

2.8.3 DCache

ZirconMIPS 采用了状态机 DCache，在 CPU 执行阶段接受访存信号，在访存阶段送回数据。在命中的情况下能够进行数据的连续读写。DCache 的采用伪 LRU 的替换策略，由于数据 Cache 采用写回策略，故只有替换的数据块为脏块时才会发起对主存的写请求。而当写请求发生缺失时，DCache 将会先从主存中取到对应的数据，进行数据合并后再写入 DCache。DCache 的状态机包含七个状态，分别是：

表 2.6　DCache 状态机的状态

1	MIDLE	当前 CPU 没有任何请求发往 DCache 或者 DCache 完成对数据缺失和 UnCache 类请求的处理
2	LOOKUP	DCache 当前收到请求并得到了它的查询结果
3	MISS	DCache 当前处理的请求缺失并得到了是否需要发起写回脏块的结果
4	REPLACE	需要将脏块写回主存，且正在等待 AXI 总线的写回结束标志
5	REFILL	DCache 当前处理的请求缺失，且正在等待 AXI 总线的读取结束标志信号
6	UNREAD	DCache 收到 UnCache 类的读请求，且正在等待 AXI 总线的读取结束标志信号
7	UNWRITE	DCache 收到 UnCache 类的写请求，且正在等待 AXI 总线的写回结束标志信号

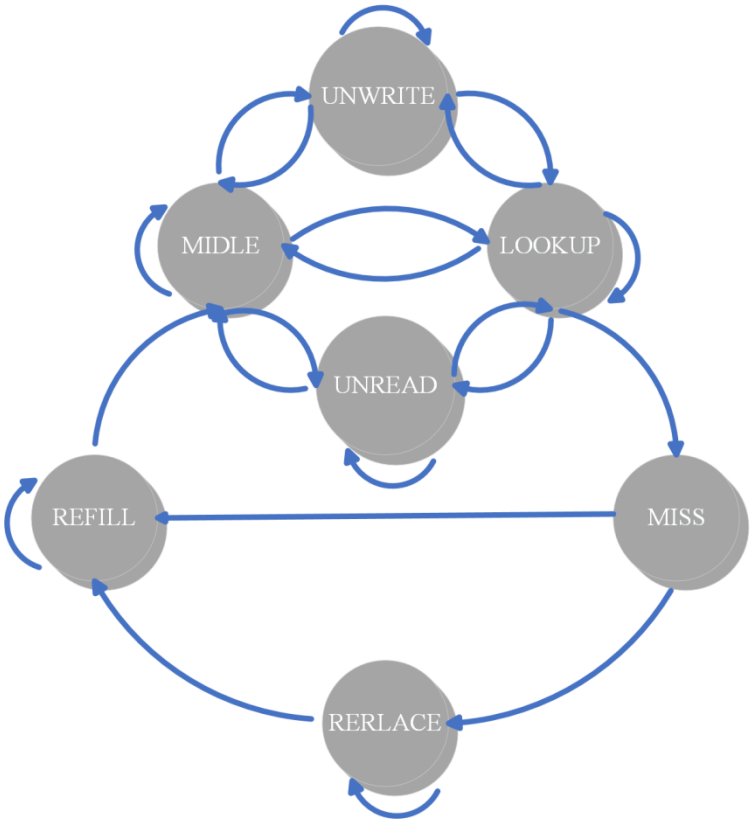


图 2.3 DCache 状态机

数据 Cache 处理读写命中与不命中的策略如下：

主存读命令命中 Cache：将 Tag 和第零路和第一路读出的 Tag 进行对比确定是否命中，命中则返回相应数据。

主存读命令未命中 Cache：如果发现 Cache 未命中，暂停 CPU 并根据伪 LRU 决定要替换的数据块。如果要被替换的数据块为脏块，则先向 AXI 发起写回请求并等待 AXI 的写回结束标志。如果被替换的数据块不是脏块或 AXI 的写回结束信号拉高，则将缺失数据的物理地址发送给 AXI 并发起读请求，收到 AXI 的读出结束标志后将数据返回给 CPU。

主存写命令命中 Cache： 将命中数据与写入的数据进行组合重写进 Cache。

主存写命令未命中 Cache： 如果发现 Cache 未命中，暂停 CPU 并根据伪 LRU 决定要替

换的数据块。如果要被替换的数据块为脏块，则先向 AXI 发起写回请求并等待 AXI 的写回结束标志。如果被替换的数据块不是脏块或 AXI 的写回结束信号拉高，则将缺失数据的物理地址发送给 AXI 并发起读请求，收到 AXI 的读出结束标志后将读出的数据与写入的数据进行组合重写进 Cache。

2.3.3 命中率

通过在 Cache 内部对请求和命中进行计数，我们通过仿真测算出了十个性能测试中指令 Cache 和数据 Cache 的命中率。

序号	测试名称	命中次数	请求次数	命中率
1	bitcount	24187	24365	99.27%
2	bubble sort	134860	135316	99.66%
3	coremark	271316	273253	99.29%
4	crc32	171636	172179	99.68%
5	dhrystone	32846	33138	99.12%
6	quick sort	123688	124429	99.40%
7	select sort	105428	105808	99.64%
8	sha	102093	102386	99.71%
9	stream copy	8024	8138	98.60%
10	string search	80634	80832	99.76%

表 2.4 指令 Cache 命中率

序号	测试名称	命中次数	请求次数	命中率
1	bitcount	3976	4002	99.35%
2	bubble sort	61738	61828	99.85%
3	coremark	83050	83173	99.85%
4	crc32	53526	53586	99.89%
5	dhrystone	9261	9340	99.15%
6	quick sort	38341	38747	98.95%
7	select sort	21735	21827	99.58%
8	sha	39983	40115	99.67%
9	stream copy	4250	4522	93.98%
10	string search	33159	33322	99.51%

表 2.5 数据 Cache 命中率

2.3.4 UnCached

指令和数据地址在 0xA000000 至 0xBFFFFFFF 段将不经过 Cache，直接和内存进行交互。Uncached 需要 CPU 能够感知不到其存在。为了解决这个问题，我们在 Cache 状态机内部增设了专门用于处理 UnCache 类请求的状态。 具体来说，首先我们先进行地址映射，进行相应的 Uncached 和 Cached 判断， 然后 CPU 发送请求使 Cache 的状态机进入由于处理 UnCache 类请求的状态。由于 UnCache 类请求都是针对字或字节的操作，这使其与 AXI 交互需要的时间少于 Cache 缺失时需要的时间。

2.9 AXI 总线

我们通过一个转接桥来转换 AXI 协议和 Cache 的类 sram 协议，并采用等待排队器来完成仲裁。写缓冲逻辑采用了一个写占用地址寄存器和写占用标志位。写占用地址寄存器保存了当前写事务的目标地址，写占用标志位表示现在 AXI 总线正处在写事务中。我们将原先等待 AXI 完成写事务后返回握手信号 wvalid 告诉 DCache 完成写事务提前到了使用 AXI 完成数据传输后的信号 wlast & wready 来告诉 DCache 完成写事务，这样一来 Cache 就不会发出暂停流水线的请求，可以极大幅度的缩减写数据带来的流水线暂停时间。

为了保证写事务的正确性，在写事务未真正完成之前，写占用标志位会拉低同时转接桥会截断所有对写占用地址寄存器保存的写地址的读事务请求。而由于 AXI 的写状态机并没有真正回到可以接受写事务的状态，这个时候所有的写事务会被挂起。但由于在写事务较多的程序中，写事务之间存在一定间隔，这段时间可以用来填充写事务响应时间，从而大幅度降低了写事务的代价。

第三章 外设与系统

3.1 概述

ZirconMIPS 的外设设计是基于 soc_up 上，对 AXI 转接桥进行修改。并通过一个自制的 AXI 转接模块操控 VGA，LCD 等外部设备。我们将这些外设分派到指定的地址以方便 CPU 统一访问。

3.2 VGA 控制器设计

我们实现了一个 VGA 控制器，该控制器能够通过 CPU 对指定地址进行写操作来完成图片的显示。我们采用 bram 进行图片数据的存储，该 bram 可通过 CPU 交互写入数据。但由于 bram 存储空间有限，输出的图片并不能实现 1920*1080 的分辨率。

3.3 LCD 控制器设计

我们同时制作了一个 LCD 屏幕空值模块，用户可以通过其进行交互。

附录 A 参考设计及资料

参考设计说明：

CPU 整体架构为顺序双发射六级流水线。我们在设计之初参考了雷思磊《自己动手写 CPU》的五级流水线以及 2020 年获奖作品 UltraMIPS 的架构，沿用其部分命名，并进行了重构。

参考资料：

在本项目设计、开发过程中，我们参考的主要资料如下：

- 《自己动手写 CPU》. 雷思磊
- 《CPU 设计实战》. 汪文祥、邢金璋
- 《超标量处理器设计》. 姚永斌
- UltraMIPS 设计文档. 李程浩、刘定邦、宫浩辰、任翔宇
- 《计算机体系结构基础》. 第二版 胡伟武

附录 B 量化分析 CPU 性能

我们在对发布包中的测试平台进行修改，搭建了我们 CPU 的性能分析平台，使用统计学方法通过量化分析性能测试中 CPU 的各项数据，统计出 CPU 在各项测试程序中的数据表现中，找出短板进行针对性优化。

对于发射逻辑优化的思考，初次统计发现双发射百分比低，可通过指令在执行阶段的动态调度进行优化。

对于提升分支预测正确率、缓存的命中率的优化，我们观察不同程序中分支预测和缓存的表现，找出正确率和命中率较低的程序，对其暴露出来的问题进行针对优化。

对于 AXI 写事务的优化，发现性能测试中 dhtystone、stringsearch，分析发现数据缓存暂停时间总量十分巨大，进一步量化分析发现大量的对 Uncache 段地址的写指令，发现在这种情况下 AXI 的写事务耗时较长，导致程序得分远低于平均表现，于是考虑增加 AXI 写缓冲逻辑，使得这两个程序的表现提升至平均水平。

	bitcount	bubble sort	coremark	crc32	dhrystone
mul1/次数	7	0	9491	0	10
mul2/次数	0	0	2685	0	10
div1/次数	0	0	0	0	0
div2/次数	14	12	83	633	105
jb2/次数	3136	4511	46326	49361	9014
ls1/次数	3741	65353	75938	53725	9360
ls2/次数	3547	55000	57449	51611	8087
cp01/次数	6	6	7	5	6
cp02/次数	3	3	3	2	3
reg3/次数	2343	4012	32598	45962	3994
reg4/次数	1292	593	28073	43250	3224
hilo/次数	0	0	0	0	0
iss1/次数	10934	90609	179729	148114	21534
iss2/次数	13434	45480	122749	57127	15613
iss_total/次数	24368	136089	302478	205241	37147
stall ex/周期	413	350	40352	18410	3365
stall_id/周期	10259	43587	128388	66205	17415
stall_dcache/周期	6150	9062	27366	26452	45816
buffer_full/周期	4371	17277	102090	28026	46257
ex_times/次数	18	10	9561	526	105
id_times/次数	2182	29927	61093	47777	5729
dcache_times/次数	221	255	1052	1144	2006
buffer_times/次数	331	3492	27769	1831	4093

quicksort	select sort	sha	stream_copy	stringsearch	
0	0	0	0	0	9508
0	0	5	0	0	2700
0	0	0	0	0	0
12	12	195	10	12	1088
13382	11912	19162	1863	18284	176951
37044	22501	35836	4558	33576	341632
19962	2316	32589	3480	27344	261385
6	6	7	6	6	61
3	3	3	3	3	29
17134	5856	24380	400	9127	145806
13860	446	15222	1119	7625	114704
0	0	0	0	0	0
72741	36144	81781	8518	58249	708353
54043	73871	48303	3234	33146	467000
126784	110015	130084	11752	91395	1175353
350	350	6075	280	350	70295
52433	35583	31733	7050	20900	413553
26335	9070	15248	18946	80255	264700
30736	7432	36924	21216	77162	371491
10	10	178	8	10	10436
32699	27909	19361	2499	13055	242231
565	255	472	433	3531	9934
3984	518	9827	1734	8880	62459