PipelineMips 设计报告

第四届全国大学生计算机系统能力培养大赛

重庆大学
袁福焱 李果 李雅雯 江焰丰

8/16/2020

目录

1. 设计简介

本次比赛我们计划实现一个流水线版的 MIPS 处理器“PipelineMIPS”。寓意充分发掘流水线的潜力，实现一个较为完善的流水线版 MIPS CPU。

我们的设计目标是实现一个流水线版的 MIPS 处理器，力求：

1. 模块清晰易懂，代码规范
2. 有较为完善的 Cache
3. 主频达到 100MHz
4. 有 TLB 支持，便于移植操作系统

我们的作品为哈佛结构的 32 位五级流水的 CPU。实现了初赛要求的 57 条指令、部分中断例外、重写了 AXI 接口、基于局部历史的两位饱和计数器的分支预测。

设计实现了指令 Cache(I-Cache)与数据 Cache(D-Cache)，其中 I-Cache 容量为 4KB，四路组相联，一个 Cacheline 为 8 字，支持 BURST 传输，处理方式是写直达；D-Cache 容量为 4KB，四路组相联，一个 Cacheline 为 8 字，支持 BURST 传输，处理方式是写回与写分配；

实现了内陷，非对齐等指令、CP0 相关寄存器，设计实现了 TLB 以支持运行系统软件。实现了 32 表项，64 页的 TLB，并成功运行了 PMON。

总体设计框架如下图所示：

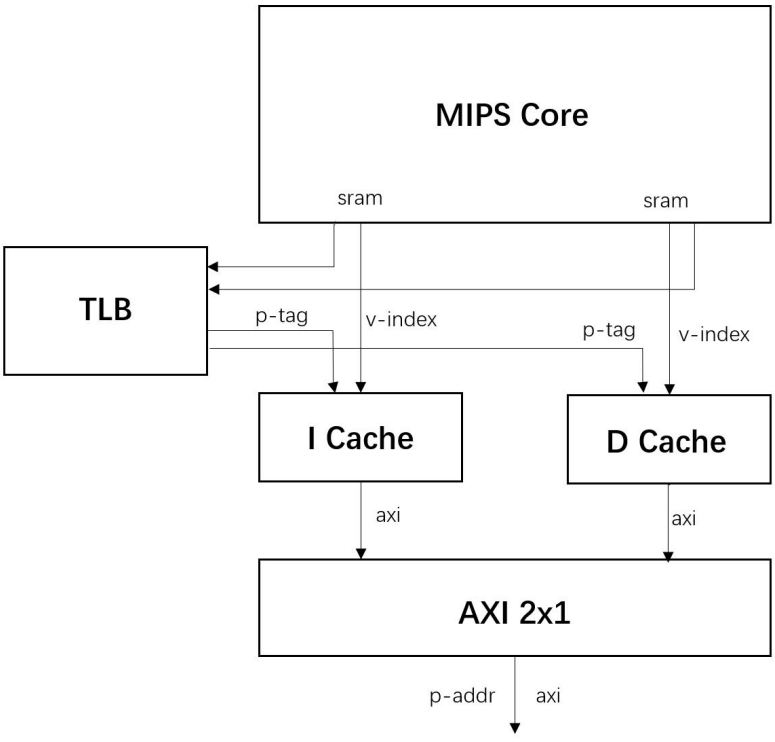

图 1：总体结构示意图

2. 详细设计方案

2.1 CPU 模块化设计

CPU 结构采用的是基础的五级流水线设计，流水线由取指(Fetch)，译码(Decode)，执行(Execute)，访存(Memory)，写回(WriteBack)五个阶段组成。CPU 由数据通路模块(/myCPU/datapath.v)，缓存模块(/myCPU/i_cache.v，/myCPU/d_cache.v)，仲裁模块（/myCPU/arbitrater.v），TLB 模块(/myCPU/tlb.v)组成。

其中 Datapath 模块为五级流水的数据通路，主要完成了流水线取指，译码，执行，访存，写回功能。此外，在该模块中，实现了协处理器 CP0 以支持异常相关指令，控制模块 hazard 以解决数据冲突，hilo 寄存器等；

其中 I_Cache 模块实现了对 CPU 与内存之间指令数据的缓存，并且支持将 sram 接口转换成 AXI 接口。D_Cache 模块实现了对 CPU 与内存之间数据的缓存，同样支持将 sram 接口转换成 AXI 接口，

其中 Arbitrater 模块（仲裁模块）支持对 I_Cace 与 D_Cache 的同时读取请求冲突；我们利用 AXI 接口特性，使得读请求与写请求可以同时发生；

TLB 模块实现了对虚拟地址到物理地址的翻译；将 DataPath 传输的虚拟地址翻译成物理地址后，将物理地址传递给 Cache 模块。

2.1.1 DataPath（数据通路）模块设计

2.1.1.1 DataPath 模块接口说明

Name	Description	Direction	Width
clk	时钟信号	INPUT	1
rst	复位信号	INPUT	1
Ext_int	外部硬件中断信号	INPUT	6
Instr_Cache -- Datapath			
pcF	Fetch 阶段指令地址	OUTPUT	32
Pc_next	取指令地址	OUTPUT	32
inst_enF	Fetch 阶段指令使能	OUTPUT	32
instrF	Fetch 阶段指令	OUTPUT	32
I_cache_stall	指令缓存暂停	INPUT	1
stallF	Fetch 阶段暂停信号	OUTPUT	1
Data_cache --- datapath			
Mem_enM	访存数据使能	OUTPUT	1
Mem_addrM	访存数据地址	OUTPUT	32

Mem_rdataM	访存读取的数据	OUTPUT	32
Mem_wenM	访存写字节使能	OUTPUT	4
mem_wdataM	访存写数据	OUTPUT	32
load_type	访存阶段 load 指令类型	OUTPUT	3
mem_addrE	访存取数据地址	OUTPUT	32
mem_read_enE	执行阶段访存读数据使能	OUTPUT	1
Mem_write_enE	执行阶段访存写数据使能	OUTPUT	1
mem_read_enM	访存阶段读数据使能	OUTPUT	1
mem_write_enM	访存阶段写数据使能	OUTPUT	1
D_cache_stall	访存缓存暂停信号	INPUT	1
cacheM	访存阶段 Cache 指令类型	OUTPUT	7
TLB -- datapath			
flushM	访存刷新信号	OUTPUT	1
TLBR	TLB 指令信号	OUTPUT	1
TLBP	TLB 指令信号	OUTPUT	1
TLBWI	TLB 指令信号	OUTPUT	1
TLBWR	TLB 指令信号	OUTPUT	1
EntryHi_to_cp0	写入 EntryHi 寄存器数据	INPUT	32
PageMask_to_cp0	写入 PageMask 寄存器数据	INPUT	32
EntryLo0_tocp0	写入 EntryLo0 寄存器数据	INPUT	32
EntryLo1_to_cp0	写入 EntryLo1 寄存器数据	INPUT	32
Index_to_cp0	写入 Index 寄存器数据	INPUT	32
EntryHi_from_cp0	从 EntryHi 读取的数据	OUTPUT	32
PageMask_from_cp0	从 PageMask 读取的数据	OUTPUT	32
EntryLo0_from_cp0	从 EntryLo0 读取的数据	OUTPUT	32
EntryLo1_from_cp0	从 EntryLo1 读取的数据	OUTPUT	32
Index_from_cp0	从 Index 读取的数据	OUTPUT	32
Random_from_cp0	从 Random 读取的数据	OUTPUT	32
inst_tlb_refillF	指令 TLB 异常	INPUT	1
inst_tlb_invalidF	指令 TLB 异常	INPUT	1
data_tlb_refillM	数据 TLB 异常	INPUT	1
data_tlb_invalidM	数据 TLB 异常	INPUT	1
data_tlb_modifyM	数据 TLB 异常	INPUT	1

2.1.1.2 DataPath 结构

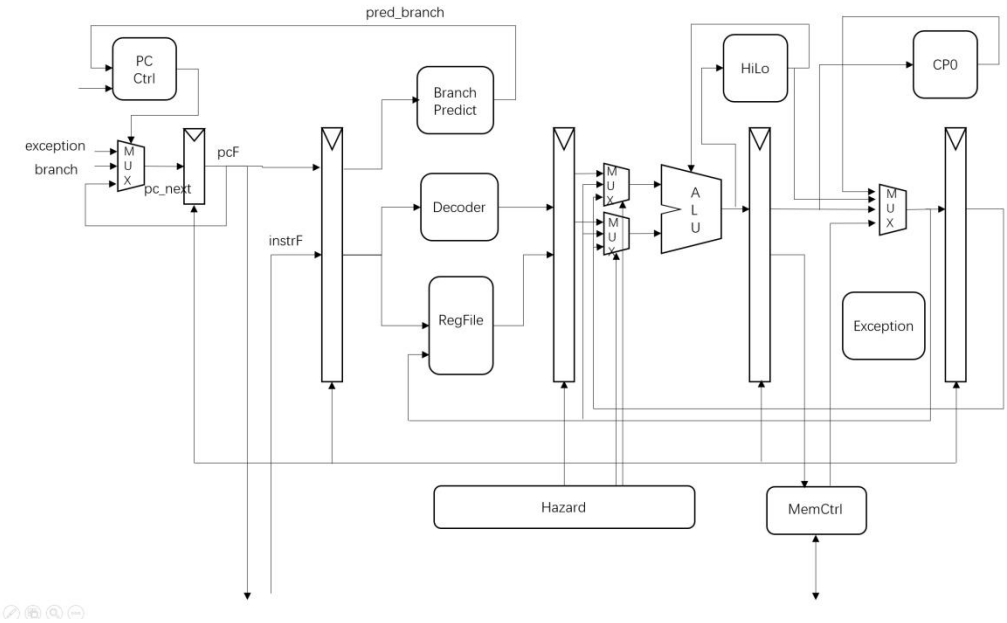

图 2：CPU 数据通路图

2.1.1.3 取指阶段分析

取指阶段功能为从存储器中取出下一条待执行的指令。

该阶段根据指令计数器取出对应指令。包含一个 PC_CTRL 控制器，实现从跳转、异常、分支预测失败等多个地址来源中选择正确的地址。

PC_CTRL 模块接口

Name	Description	Direction	Width
branchD	译码阶段是否是分支指令	INPUT	1
branchM	访存阶段是否是分支指令	INPUT	1
succM	分支指令是否预测成功访存阶段信号	INPUT	1
actual_takeM	分支指令是否应该跳转访存阶段信号	INPUT	1
pred_takeD	分支指令预测跳转译码阶段信号	INPUT	1
flush_exceptionM	异常信号	INPUT	1
jumpD	译码阶段是否是 Jump 指令	INPUT	1
jump_conflictD	译码阶段 Jump 指令是否产生数据冲突	INPUT	1
jump_conflictE	执行阶段 Jump 指令是否产生数据冲突	INPUT	1
PC_SEL	PC 多路选择器控制信号	OUTPUT	3

PC_CTRL 真值表

flush_exceptionM	branchM	succM	actual_takeM	jump_conflictE	jumpD	jump_conflictD	branchD	pred_takeD	PC_SEL
1	x	x	x	x	x	x	x	x	6
0	1	0	0	x	x	x	x	x	5
0	1	0	1	x	x	x	x	x	4
0	[0,0,1]	[1,0,1]	x	1	x	x	x	x	3
0	[0,0,1]	[1,0,1]	x	0	1	0	x	x	2
0	[0,0,1]	[1,0,1]	x	0	[0,0,1]	[0,1,1]	1	1	1
ELSE									0

图 1：其中 branchM：[0,0,1]；succM[1,0,1]分别对应 branchM：0，succM：1 等情况，其余类推

PC_SEL 多路选择器

PC_SEL	PC_NEXT
6	pc_exceptionM
5	pc_plus4E
4	pc_branchM
3	pc_jumpE
2	pc_jumpD
1	pc_branchD
0	pc_plus4F
7	0

图 2：pc_sel 多路选择器选择表

由于指令缓存由 Block Ram 实现，综合考虑时序，该阶段根据 PC_NEXT 取指。其中 TLB 与 Cache 采用的是“virtual index， physical tag”的结构，Cache 采用的是并行访问的结构。所以在该阶段，从 PC_NEXT 中截取到 index，分别传入 TLB 与 Cache 中。然后，将从 TLB 中取出的物理 TAG 与 Cache 中读取的 Tag 进行比对，判断是否命中，若命中，则流水线正常进行；否则发起读缺失请求，并且暂停流水线所有阶段。

我们实现了基于局部历史的两位饱和计数的分支预测。在取指阶段，根据取指阶段的 PC，根据历史记录查询是否跳转，并将预测结果传递给译码阶段。

同时，在该阶段可能发生地址非对齐异常，TLB 异常等，对异常进行标记，然后统一放到访存阶段处理。

2.1.1.4 译码阶段分析

译码阶段完成对指令的解码，访问通用寄存器以及分支跳转。

MIPS32 指令分为 I，J，R 中类型，相关的控制信号跟算数执行单元 ALU 所需控制信号由 main_decoder 跟 alu_decoder 解码产生。相关代码位于：/myCPU/alu_decoder.v 跟 /myCPU/main_decoder.v。

关于分支跳转，由于我们在取指阶段根据 PC 得到预测结果，所以在解码到当前指令为分支指令时，根据解码结果决定是否跳转。

通用寄存器截取指令对应域，返回对应寄存器的值，然后经过流水线传递到

执行阶段。

关于译码阶段可能产生的数据冲突，由于我们实现了分支预测，综合考虑频率，我们在译码阶段不考虑分支跳转指令的数据冲突，而是在执行阶段解决跟分支跳转指令相关的数据冲突。

1. Main_decoder 模块接口说明

Name	Description	Direction	Width
clk	时钟信号	INPUT	1
rst	复位信号	INPUT	1
instrD	译码阶段指令	INPUT	32
sign_extD	立即数是否有符号拓展	OUTPUT	1
is_divD	是否是除法指令	OUTPUT	1
is_multD	是否是乘法指令	OUTPUT	1
l_s_typeD	Load store 类指令	OUTPUT	14
mfhi_loD	控制 hilo 写通用寄存器	OUTPUT	1
reg_dstD	写通用寄存器目的寄存器	OUTPUT	2
alu_imm_selD	src_aE 源操作数是否选择立即数	OUTPUT	1
hilo_wenD	HILO 写使能	OUTPUT	1
mem_write_enD	存储数据使能	OUTPUT	1
reg_write_enD	写寄存器使能	OUTPUT	1
mem_to_regD	Load 写寄存器使能	OUTPUT	1
hilo_to_regD	HILO 写寄存器使能	OUTPUT	1
riD	保留指令	OUTPUT	1
breakD		OUTPUT	1
syscallD	系统调用指令	OUTPUT	1
eretD	异常返回指令	OUTPUT	1
cp0_wenD	CP0 写使能	OUTPUT	1
cp0_to_regD	CP0 写寄存器使能	OUTPUT	1
tlb_typeD	Tlb 指令类型	OUTPUT	4
movzD	Movz 指令	OUTPUT	1
movnD	Movn 指令	OUTPUT	1
cacheD	Cache 指令类型	OUTPUT	5

2.1.1.5 执行阶段分析

执行阶段主要由两个模块组成，算数逻辑运算单元跟分支预测判定单元。

其中算数逻辑运算单元（/myCPU/EX/alu.v）负责逻辑运算与算数运算，大多数指令都可以在一个周期内完成。除法采用自己实现的多周期除法实现（/myCPU/EX/div.v），支持有符号除法跟无符号除法；考虑到乘法的时延较高，所以使用 Xilinx 配置的多周期乘法 ip 实现，同样支持有符号乘法与无符号乘法。

关于分支预测结果判定，相关代码位于（/myCPU/EX/branch_judge.v）中，根据译码阶段识别的指令结果，判断分支指令实际是否应该跳转。并将判断结果传递给访存阶段。

关于数据冲突，可能会有数据冲突的情况发生，算数逻辑运算的源操作数通过旁路网络从访存阶段或者写回阶段前推得到。

关于 TLB，在实现 TLB 的过程中，发现在 MEM 阶段查询对主频有较大的影响。所以将 TLB 的查询提前到了执行阶段。

2.1.1.6 访存阶段分析

访存阶段实现了访问存储内存数据（包括数据 Cache 的访存），HILO 寄存器读写，整合处理异常信息，处理分支预测失败，以及更新分支预测中分支指令历史表和地址历史表。

关于数据访存，数据 Cache 使用 Block Ram 实现。首先，在执行阶段擦汗寻 TLB 获取到访存虚地址对应的物理页号；同时根据访存虚拟地址获取到对应的 Cache Set，并根据物理页号比对 TAG，判断是否命中，如果缺失，则发起对应的读写请求。

关于异常信息的整合处理，在访存阶段，可能会产生地址错例外，结合前面流水传递过来的异常信息，由 Exception 模块根据例外优先级对判断处理异常，并且生成相应的异常处理控制信号。

关于处理分支预测失败，根据执行阶段分支预测判定结果，结合分支预测结果，如果失败，则产生清空流水线信号，将错误取出的指令清除。同时，根据正确的分支指令跳转结果，更新分支预测中 BHT 跟 PHT 表项。

关于数据冲突，在访存阶段根据旁路网络信号，前推冲突数据。、

2.1.1.7 写回阶段分析

写回阶段负责写入通用寄存器，读写 cp0 以及相应的数据前推。

其中，当遇到异常处理与 mtc0 指令或者 TLBR、TLBP 指令时，向 CP0 中写入数据。读取 CP0 时组合逻辑。

1. CP0 模块接口说明

Name	Description	Direction	Width
clk	时钟信号	INPUT	1
rst	复位信号	INPUT	1
ext_int	外部硬件中断	INPUT	1
stallW	写回阶段暂停信号	INPUT	1
addr	读取 cp0 地址	INPUT	1
sel	Cp0, sel 域	INPUT	1
wdata	写 cp0 数据	INPUT	14
rdata	读取 CP0 数据	INPUT	1
flush_exception	异常处理信号	INPUT	1

except_type	异常类型	INPUT	1
pcM	访存阶段地址	INPUT	1
is_in_delayslot	当前指令是否位于延迟槽	INPUT	5
badvaddr	除法异常的错误地址	INPUT	5
tlb_typeM	TLB 指令类型	INPUT	5
entry_lo0_in	从 TLB 写入 CP0 数据	INPUT	5
entry_lo1_in	从 TLB 写入 CP0 数据	INPUT	32
page_mask_in	从 TLB 写入 CP0 数据	INPUT	32
entry_hi_in	从 TLB 写入 CP0 数据	INPUT	32
index_in	从 TLB 写入 CP0 数据	INPUT	32
cp0_statusW	读取 CP0 status 寄存器	INPUT	32
cp0_causeW	读取 CP0 cause 寄存器	INPUT	32
cp0_epcW	读取 CP0 epc 寄存器	OUTPUT	32
cp0_ebaseW	读取 CP0 ebase 寄存器	OUTPUT	32
entry_hi_W	读取 CP0 entry_hi 寄存器	OUTPUT	32
page_mask_W	读取 CP0 page_mask 寄存器	OUTPUT	32
entry_lo0_W	读取 CP0 entry_lo0 寄存器	OUTPUT	32
entry_lo1_W	读取 CP0 entry_lo1 寄存器	OUTPUT	32
index_W	读取 CP0 index 寄存器	OUTPUT	32
random_W	读取 CP0 random 寄存器	OUTPUT	32

2.2 冲突处理

由于数据指令间相互联系，因此可能产生冲突，常见的有数据冲突和控制冲突。对于流水线结构，数据冲突只包含写后读冲突，可以通过数据前推（旁路）和暂停来解决。

2.2.1 数据冲突

对于两条计算类指令读写寄存器堆的冲突，我们可以通过将 M 和 W 阶段的数据前推到 E 阶段来解决。对于执行阶段为 load 类指令，与译码阶段的指令产生数据冲突的情况，我们通过暂停取指和译码阶段一个周期来解决。而对于先后写读 HiLo 寄存器，CP0 寄存器，由于我们 HiLo 寄存器，CP0 寄存器的读写只差一个周期，因而不会产生冲突。对于多周期计算指令如乘法和除法，我们通过暂停来解决

这些前推，暂停逻辑都被封装到 Hazard 模块中解决。

HAZARD 模块接口说明

Name	Description	Direction	Width
clk	时钟信号	INPUT	1
rst	复位信号	INPUT	1
i_cache_stall	指令缓存缺失暂停	INPUT	1
d_cache_stall	数据缓存缺失暂停	INPUT	1
div_stallE	除法暂停	INPUT	1
mult_stallE	乘法暂停	INPUT	1
l_s_typeE	Load store 类指令	INPUT	14
flush_jump_confilctE	由于 jump 冲突产生的清空信号	INPUT	1
flush_pred_failedM	分支预测失败产生的清空信号	INPUT	1
flush_exceptionM	异常清空信号	INPUT	1
flush_branch_likely_M	Branch likelu 清空延迟槽信号	INPUT	1
rsE	执行阶段 Rs	INPUT	5
rsD	译码阶段 Rs	INPUT	5
rtE	执行阶段 Rt	INPUT	5
rtD	译码阶段 Rt	INPUT	5
reg_write_enE	执行阶段写寄存器使能	INPUT	1
reg_write_enM	访存阶段写寄存器使能	INPUT	1
reg_write_enW	写回阶段写寄存器使能	INPUT	1
reg_writeE	写目的寄存器	INPUT	5
reg_writeM	写目的寄存器	INPUT	5
reg_writeW	写目的寄存器	INPUT	5
stallF	取指阶段暂停信号	OUTPUT	1
stallD	译码阶段暂停信号	OUTPUT	1
stallE	执行阶段暂停信号	OUTPUT	1
stallM	访存阶段暂停信号	OUTPUT	1
stallW	写回阶段暂停信号	OUTPUT	1
flushF	取指阶段清空信号	OUTPUT	1
flushD	译码阶段清空信号	OUTPUT	1
flushE	执行阶段清空信号	OUTPUT	1
flushM	访存阶段清空信号	OUTPUT	1
flushW	写回阶段清空信号	OUTPUT	1
forward_aE	前推 src_aE 信号	OUTPUT	2
forward_bE	前推 src_bE 信号	OUTPUT	2

2.2.2 控制冲突

由于我们采用了分支预测，因而需要处理分支预测失败的情况。我们在 M 阶段获得分支指令结果，并以此产生控制信号。

1 如果预测成功，则正常执行

2 如果预测跳转，实际不跳转，则清空 D, E 阶段指令，并跳转到分支指令的 PC+8 处

3 如果预测不跳转，实际跳转，则清空 D, E 阶段指令，并跳转到分支跳转处

由于 jump 类指令包含 jr 等非直接跳转指令，因此我们在译码阶段可能无法取得跳转地址，因此我们在译码阶段判断是否产生 jump 冲突，如果没有，则直接跳转，如果有，则将冲突传递到执行阶段，此时可以获得跳转地址，并产生信号清空译码阶段指令。

2.3 分支预测设计

本项目实现了基于局部历史的两位饱和计数器的分支预测，使用 BHT 保存分支指令历史记录，PHT 来保存分支指令跳转状态，其中配置参数为：BHT：1024*6；PHT：1*64*2。

在取指阶段根据取指地址 PC 查询分支预测的 BHT 跟 PHT，得到预测结果，然后将预测结果传递到译码阶段。

在译码阶段，根据预测结果决定是否进行跳转。

在执行阶段，判断分支指令实际跳转方向。

在访存阶段，根据分支指令实际跳转方向，判断分支预测是否预测成功。若预测失败，则清空取指跟译码两个阶段的流水线，并将指令计数器置为正确的跳转地址。

2.3.1 分支预测模块接口说明

Name	Description	Direction	Width
clk	时钟信号	INPUT	1
rst	复位信号	INPUT	1
instrD	译码阶段指令	INPUT	32
immD	译码阶段立即数	INPUT	32
pcD	译码阶段地址	INPUT	32
pcM	访存阶段地址	INPUT	32
branchM	访存阶段是否是分支指令	INPUT	1
actual_takeM	访存阶段分支指令实际跳转方向	INPUT	1
branchD	译码阶段是否是分支指令	OUTPUT	1
branchL_D	译码阶段是否是 branch_likely 指令	OUTPUT	1
pred_takeD	预测跳转方向	OUTPUT	1

2.3.2 分支预测模块预测、更新逻辑

本项目采用的是基于局部历史的两位饱和计数器的分支预测，其本质上是利用一条分支指令的历史跳转记录训练一个两位饱和计数器，由饱和计数器的值来

决定是否跳转。

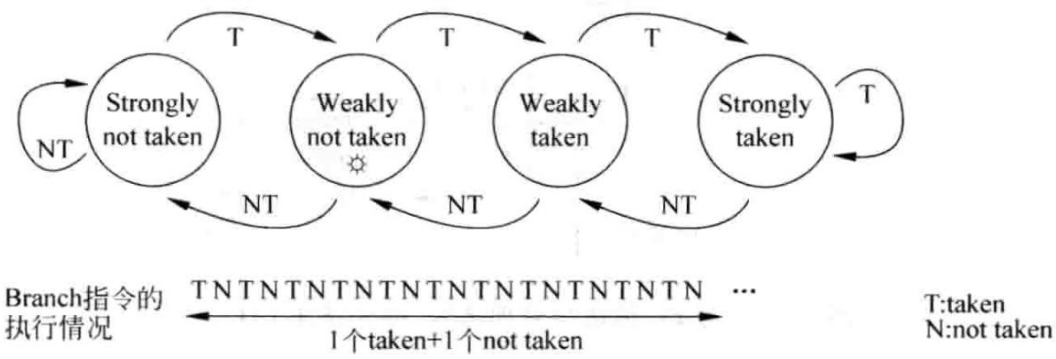

图 3：基于两位饱和计数器的分支预测，以及失效情况。引用自《超标量处理器设计》

对于普通的两位饱和计数分支预测，在某种规律的跳转下易失效。利用分支指令跳转的局部历史来训练计数器，可以识别分支指令跳转规律，从而更准确的预测跳转方向。

为了避免由于 PHT 个数过多而对主频产生影响，本项目实现的分支预测结构是一个 1024*6 的 BHT 表和一个 64*2 的 PHT 表。

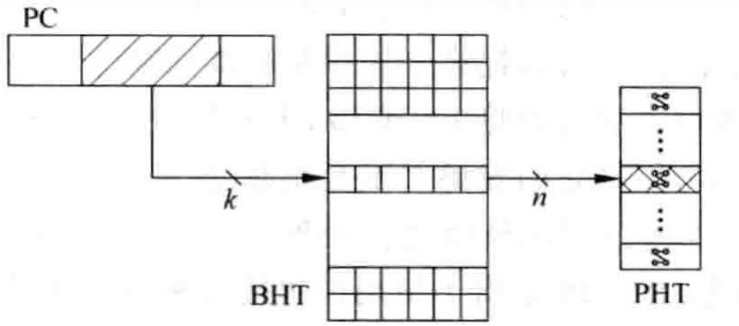

图 4： 为所有 PC 使用同一个 PHT，引用自《超标量处理器设计》

分支预测预测逻辑是根据 PC 索引 BHT 表项，然后根据 BHT 表项内容索引到 PHT 表项，然后根据表项的内容决定是否要进行跳转。

在访存阶段，根据分支指令实际跳转方向，通过 PC 索引 BHT、PHT，更新对应的表项。

结合性能测试程序中分支跳转指令特点

1. 由于 CPU 采用字节寻址，所以地址低两位不用于寻址 BHT；
2. 同时考虑到性能测试程序大小 10MB 左右
3. 4 条指令中连续出现两条分支预测指令的概率较小

综合考虑后，基于以上原因，为了充分利用 BHT 与 PHT 的空间，我们使用 PC[13:3]来寻址 BHT。

相关代码位于“/myCPU/branch_judge.v”。

2.4 AXI 接口设计

如下图所示，AXI 接口是 CPU 对外进行访存的最后一层。主要的功能为汇集

I cache 和 D cache 的请求，对外提供统一的 axi 接口。且当 I cache 和 D cache 同时发请求时需要做出仲裁。

有两种实现方式，第一种为 I cache 和 D cache 实现为类 sram 接口，并在 axi 模块中同时实现类 sram 转 axi 的功能。官方提供了类 sram 转 axi 的转换桥，但是似乎没有实现仲裁的逻辑。

第二种方式为 cache 直接实现为 axi 接口，axi 模块只需要实现仲裁。由于我们的 cache 需要使用到 axi burst 传输的特性，因此我们采用了第二种方式。

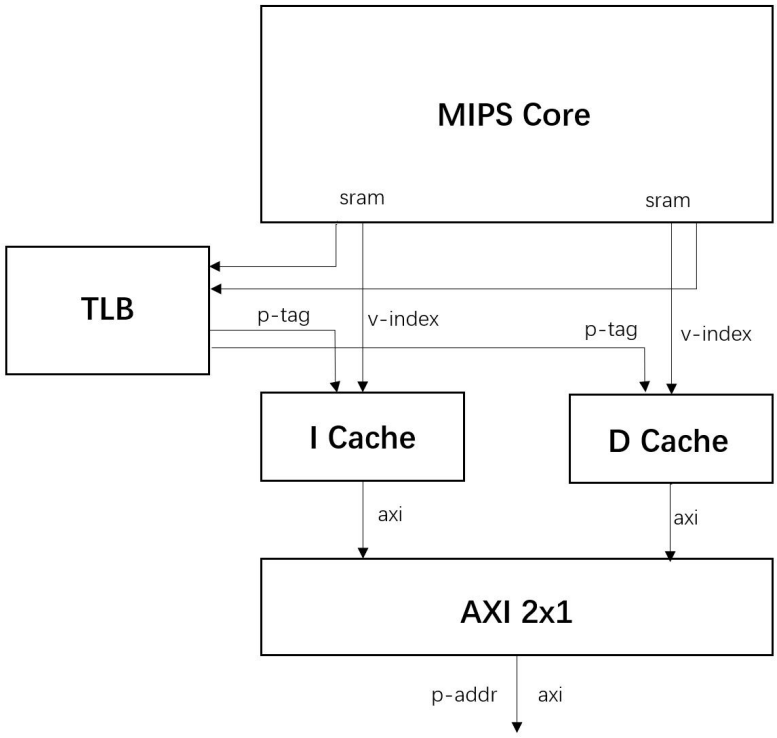

图 5： CPU 顶层结构

关于仲裁逻辑，由于 axi 接口的读写信道是分开的，因此可以同时发送读请求和写请求。又因为只有 D cache 会发送写请求，因此需要仲裁的情况只有 I cache 和 D cache 同时发送读请求的情况。我们的仲裁逻辑为：优先发送指令的读请求，在指令的地址握手成功后(arvalid & arready)，再发送数据的读请求，并且用 arid=0 代表指令，arid=1 代表数据。当 axi 接口接收到数据时(rvalid)，便根据 rid 来将数据分发给指令 cache 和数据 cache。

Axi 模块(arbitrater)接口如下:

Name	Description	Direction	Width
clk	时钟信号	INPUT	1
rst	复位信号	INPUT	1
I cache 读			
i_araddr	I cache 发出读请求的地址	INPUT	32

i_arlen	Burst 事务长度	INPUT	4
i_arvalid	(i-cache->arbitrater)地址握手	INPUT	1
i_arready	(arbitrater ->i-cache)地址握手	INPUT	1
i_rdata	返回给 I cache 的数据	INPUT	32
i_rlast	标识最后一个数据	INPUT	1
i_rvalid	(arbitrater ->i-cache)数据握手	INPUT	1
i_rready	(i-cache->arbitrater)数据握手	INPUT	1
D cache 读			
d_araddr	D cache 读地址	OUTPUT	1
d_arlen	Burst 事务长度	INPUT	4
d_arsize	不同于指令Cache固定为2。对于lb, lh等指令且不经过 cache 时，需要精确控制 arsize 信号。（不能采用读一个字然后取其中若干字节的方法）	INPUT	3
d_arvalid	(d-cache->arbitrater)地址握手	INPUT	1
d_arready	(arbitrater ->d-cache)地址握手	OUTPUT	1
d_rdata	返回给 D cache 的数据	OUTPUT	32
d_rlast	标识最后一个数据	OUTPUT	1
d_rvalid	(arbitrater ->d-cache)数据握手	OUTPUT	1
d_rready	(d-cache->arbitrater)数据握手	INPUT	1
D cache 写			
d_awaddr	D cache 写地址	INPUT	32
d_awlen	写事务 burst 长度	INPUT	4
d_awsize	写的大小	INPUT	3
d_awvalid	(d-cache->arbitrater)写地址握手	INPUT	1
d_awready	(arbitrater ->d-cache)写地址握手	OUTPUT	1
d_wdata	写数据	INPUT	32
d_wstrb	控制写哪个字节通道	INPUT	4
d_wlast	表示最后一个写数据	INPUT	1
d_wvalid	(arbitrater ->d-cache)写数据握手	INPUT	1
d_wready	(d-cache->arbitrater)写数据握手	OUTPUT	1
d_bvalid	(arbitrater ->d-cache)写响应握手	OUTPUT	1
d_bready	(d-cache->arbitrater) 写响应握手	INPUT	1
Outer（对外接口）			
标准的 axi 接口,故省略	*握手信号具体含义见 AXI4 手册*		

2.5 Cache 设计

2.5.1 设计目标

在我们的设计宣言中便已经明确了我们要设计一个较高性能的 Cache。为了达到这个目标我们考虑了以下几点：

1. 采用组相联而不采用直接映射
   直接映射的 cache 实现起来比较简单，但是缺点是会导致 cache 的冲突缺失率比较高。因为直接映射一个数据块只能存放再一个 cache line 中，当 cpu 不断访问两个被映射到同一个 cache line 的地址时，便会导致不停的缺失。而采用组相联时，一个数据块可以放在多个 cache line 中，通过合理的替换算法，可以大大降低 cache 的冲突缺失率。
2. Cache 块大小采用多个字
   当 cache 块大小使用 1 个字时，cache 实现起来比较简单。当写缺失时，CPU 可以将要写的数据直接写入 cache（如果 dirty 则还需将原来的数据写入内存）。而采用块大小为多个字时，由于写入的数据只是一个字，而块大小为多个字，因此需要从内存中读取 cache 块大小的数据，然后将要写的数据拼接到其中才能写入 cache。
   但是由于 cache 为多个字时，可以利用取指和访存的空间局部性，从而提高 cache 的命中率。且由于 AXI 的 burst 传输，因此一下取 8 条指令的时间远比 8 次取 1 条指令的时间短，因此块大小为多字可以提高整体的 IPC。
3. Cache 采用写回策略
   这一点毋庸置疑，因为相比于 CPU 执行一条指令，访问内存的时间要长得多。如果每次写 cache 时都需要写回内存，CPU 大多数时间都会是暂停的。而采用写回策略时，只有 cache 被修改过（dirty）且被替换回内存时才需要访问内存。因此效率大大提升了。不过缺点是会造成 cache 和内存的不一致，且可能造成指令 cache 和数据 cache 的不一致。
4. Cache 实现为物理 cache 还是虚拟 cache
   实现物理 cache 最为简单，但是需要在一个周期内串行地完成 TLB 地址转换和 cache 访问，会对时序造成较大影响。而实现虚拟 cache 又会出现重名问题，浪费 cache 的大小（多个虚拟地址对应同一个物理地址），甚至导致数据的不一致（两个虚拟地址对应同一个物理地址，只对其中一个更改会导致与另一个的不一致）。因此我们采用物理 tag，虚拟 index 的方式来实现特殊的物理 cache，这种方式要求 cache 一路的大小与页大小相等。如页大小为 4KB，当 cache 的 index+offset 为 12 位时，物理地址的物理页号和 cache 中的 tag 完全对应，又由于物理地址中的页内偏移和虚拟地址的页内偏移一致，因此可以采用虚拟地址的 index 取访问 cache，同时进行 TLB 转换，将得到的物理页号用于 cache 的命中判断。从而既实现了物理 cache 又不会影响时序。但缺点便是 cache 一路的大小被限制在 4KB，不过好在我们可以通过扩大 cache 的路数来增大 cache 的容量，因此对于一级 cache 容量无需太大的情形，采用这种方式是比

较好的。

本项目实现了一级的指令 cache 和数据 cache。指令 cache 和数据 cache 均为 16KB，4 路组相联，块大小为 8 个字，且数据 cache 采用写回-写分配策略。Cache 替换算法采用伪 LRU 算法。另外，cache 代码层面采用了参数化设计，可以动态调整 cache 的相联度和块大小。

1.5.1 Cache 结构

采用 xilinx 的 BlockMemoryGenerator ip 核作为 ram，用于存储 tag 和 data 部分。而对于容量比较小的 valid 位和 dirty 位，采用了 reg 实现。

一个 Cache line 的参数如下表：（单位为 1 比特）

	tag	valid	dirty	data
I Cache	20	1		32*8
D Cache	20	1	1	32*8

因此为了存放 tag 部分，我们实例化了一个宽度为 20 的 BRAM（BlockMemoryGenerator 生成的 ram）。而为了存储 data 部分，我们定制了一个宽度为 32 的 BRAM，然后实例化了 8 次，即 data_bank0, data_bank1,…data_bank7。

Cache 的整体结构图如下：

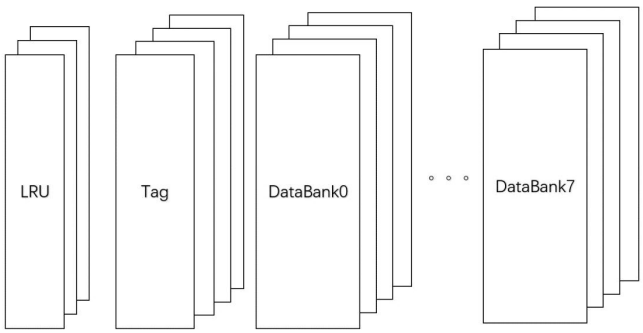

图 6: cache ram 组织

1.5.2 状态机

由于 I Cache 相比较于 D Cache 减少了写的逻辑，因而更简单。故我们只对 D Cache 进行说明。

写回写分配的 cache 的处理流程图如下：

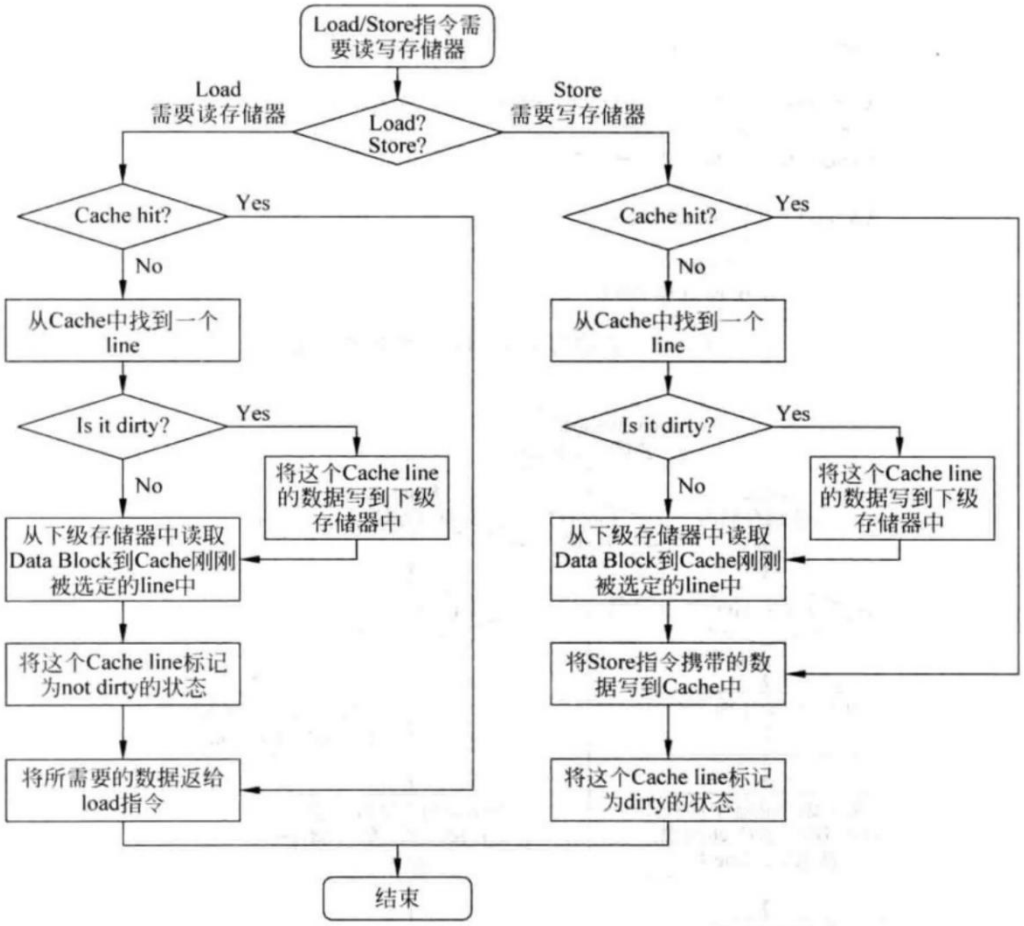

图 7 写回写分配 cache，姚永斌，《超标量处理器设计》p28

我们实现的 D cache 的状态机如下：

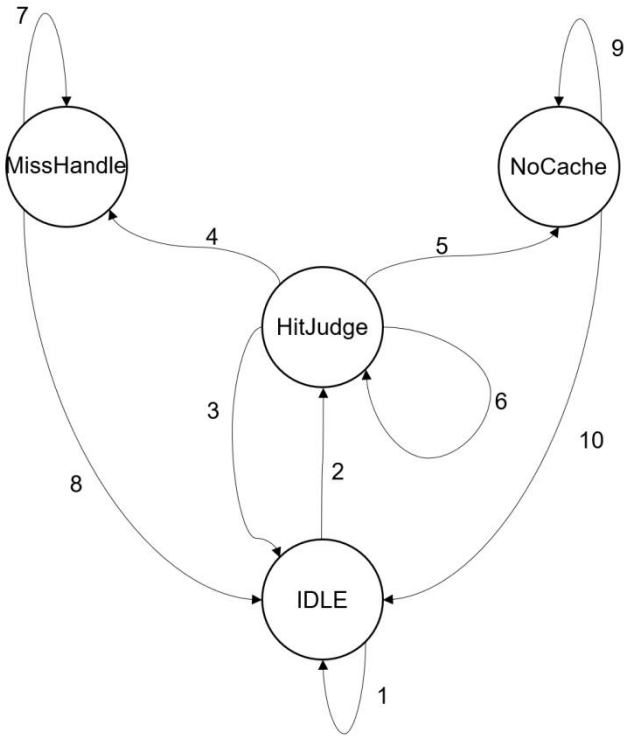

图 8　D Cache 状态图

状态

IDLE：表示 cache 处于空闲状态

HitJudge：表示 cache 在该阶段进行 hit 判断（此时访存指令到达 MEM 阶段）

MissHandle：表示 cache 正在处理缺失(读/写)，等待从内存读取数据或写数据到内存

NoCache：表示地址是不经过 cache 的，因此需要直接访问内存

路径

Path1: 当没有访存指令时，Cache 保持空闲

Path2：当 EXE 阶段有一条访存指令时，进入命中判断阶段

Path3：MEM 阶段的访存指令命中，且此时 EXE 阶段不是另一条访存指令，回到空闲

Path4：MEM 阶段的访存指令缺失，进入缺失处理

Path5：MEM 阶段的访存指令地址不经过 Cache，进入直接访存状态（此时 burst len 置为 1）

Path6：MEM 阶段的访存指令命中，且此时 EXE 阶段是另一条访存指令，保持 HitJudge

Path7：缺失处理未完成，没有从内存读完数据，或没有将数据写完到内存

Path8：缺失处理完成，回到空闲

由于采用了 BRAM ip 核，因此读取数据至少有一个周期的延迟，为了掩盖该一个周期的延迟。我们在一条访存指令到达 EXE 阶段时便将其地址传给了 D

Cache。当该指令到达 MEM 阶段的时钟上升沿，可以从 BRAM 中取出 Tag 和 Data，而我们的 D Cache 状态机也到达了 HitJudge 阶段，从而可以进行 Hit 判断。通过判断结果控制控制是否暂停流水线。

关于缺失时访问内存：当 cache 读缺失时，我们在进入 MissHandle 的同时会产生一个 read_req，表示对内存进行读操作。同时如果被替换的块是脏的，我们也会同时产生一个 write_req，表示对内存进行写操作。由于无法判断读和写操作哪个先完成，Path8 的逻辑是判断读写操作都完成。而对于写缺失时，由于块大小为多字，我们需要也要向内存发送一个读请求，并且如果替换的块是脏的，也同样要发一个写请求，因此可以看到读请求和写请求的控制信号是基本相同的。

关于 burst 传输：由于我们的一次读请求会返回多个字，因此需要使用一个计数器，来累计当前传输了多少个字。由于我们 cache line 的 data 部分采用了多个 BRAM 的实例(data_bank0, data_bank1, … data_bank7)，因此可以分别控制写使能，因此我们采用的写 cache 的逻辑是，axi 每返回一个数据，我们就将数据写入对应的 data bank。为了做到这一点，需要将计数器解码，生成一个 8 位的独热码，来控制每一个 data bank 的写使能。当然，对于读 cache 缺失和写 cache 缺失的处理不太一样。前者直接将从内存读到的数据写入 cache。而后者则还需要控制某一个 bank 写入原本的写数据。

2.5.3 替换算法

为了发挥组相联 cache 的优势，设计一个合理的替换算法是非常重要的（直接映射 cache 没有替换算法）。我们采用的是一种基于年龄位的伪 LRU 算法。原理如下

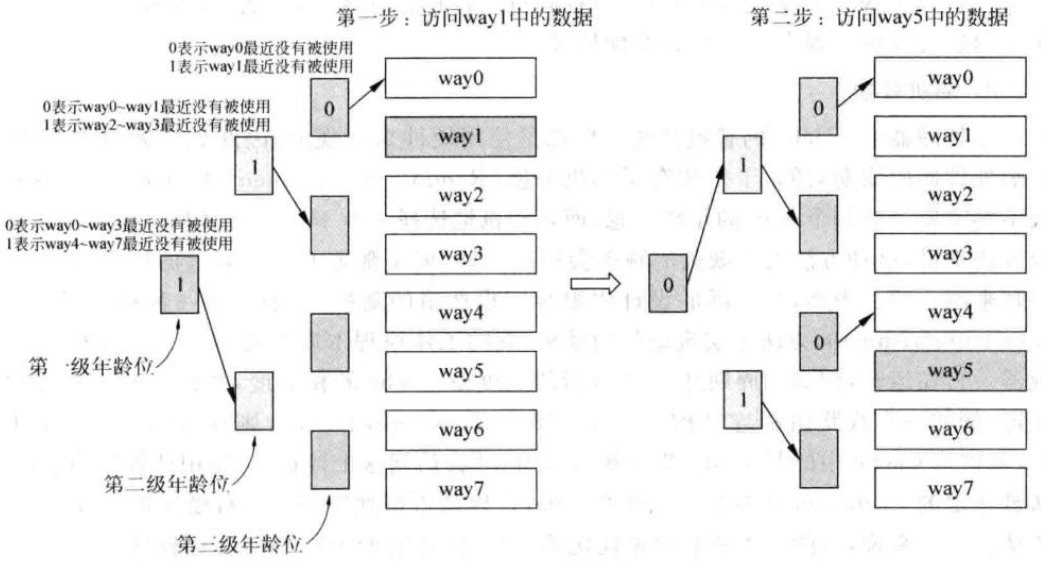

图 9 基于年龄位的伪 LRU 算法，姚永斌，《超标量处理器设计》，p29

对于一个有 2^n 路的 cache，每一个 cache 行需要 2^n-1 位来存储使用信息。我们的 cache 为 4 路，因此需要 3 位来存储最近的使用信息。当访问 cache 的某

一路时（命中或者缺失后替换），将访问路径上经过的 bit 按照含义(该 bit 为 0，表示上一半最近没有访问过，为 1 表示下一半最近没有访问过)进行改变，而其他未访问的位保持不变。当需要替换一个块时，我们从顶层开始，根据该 bit 存储的信息，选择一个分支进入下一层，最终便能找到需要被替换的一路。

1.5.3 cache 指令

处理器在某些情况下需要直接对 cache 进行控制。如当处理器执行一些“自修改”指令时，由于新的指令会先写到 D-Cache 中，因此 I-Cache 中未必能够读到新的指令。此时就要将 D-Cache 中的这些内容全都写回到内存中，并将 I-Cache 中这些指令置为无效。

官方文档推荐实现的 Cache 指令的 6 个操作如下：

op 编码	操作名称	地址使用方式	功能描述
I Cache			
0b00000	Index Invalid	Index	使用地址找到 Cache line 后，将其无效。
0b01000	Index Store Tag	Index	使用地址找到 Cache line 后，将 CP0 寄存器 TagLo 指定的 Tag, V, D 域更新进该 Cache line。
0b10000	Hit Invalid	Hit	使用地址查找 I Cache，如果命中，则将该 Cache line 无效
D Cache			
0b00001	Index Writeback Invalid	Index	使用地址找到 Cache line 后，将该行无效掉。如果该行是有效且脏的，则需要先写回到系统内存里。
0b01001	Index Store Tag	Index	使用地址找到 Cache line 后，将 CP0 寄存器 TagLo 指定的 Tag, V, D 域更新进该 Cache line。
0b10001	Hit Invalid	Hit	使用地址查找 D Cache，如果命中，则将该行无效掉。即使该行是有效且脏的，也不需要写回到系统内存里。
0b10101	Hit Writeback Invalid	Hit	使用地址查找 D Cache，如果命中，则将该行无效掉。如果该行是有效且脏的，则需要先写回到系统内存里。

对于地址使用 Index 方式的操作都比较好实现，基本可以复用原有的逻辑。地址来源需要复用原本的访存指令的地址。但是对于 I Cache 的 Hit 类操作，由于 I Cache 既需要进行下一条取指的 Hit 判断，也需要进行输入地址的 Hit 判断，因此可能需要暂停流水线。由于 Cache 指令在我们跑 PMON 中只在初始化时使用，为了不增加复杂度，我们将其实现为不进行 Hit 判断，而是直接将索引到的该 cache set 全部置为无效。

2.6 TLB 设计

为了支持系统软件的运行，加快虚实地址转换的速度。本项目实现了 TLB。并实现了 TLBP，TLBR，TLBWI，TLBWR 四种 TLB 指令，并且支持 TLB Refill，TLB Invalid，TLB Modified 三种 TLB 异常例外。

TLB 为 32 项全相连结构，每一项分奇偶页共对应 64 页。每项均包含有效位、脏位、物理页号跟虚拟页号。翻译时，获取到虚拟地址的虚拟页号，在 TLB 表中进行逐项查找，匹配到项目后，返回其物理页号。

关于 TLB 与 Cache 的结构，我们采用的是“virual index, phycical tag”的结构，由于 CPU 设计页大小为固定 4K 大小，Cache 一路容量也为 4KB，所以我们成功避免了由于这种结构所带来的“重名”问题。

为了提高频率，我们将 I_TLB 拆分为两个周期查询（五级流水，查询过程不暂停流水线）；D_TLB 提前到执行阶段访问。

2.6.1 TLB 模块接口说明

Name	Description	Direction	Width
clk	时钟信号	INPUT	1
rst	复位信号	INPUT	1
stallM	暂停信号	INPUT	1
flushM	刷新信号	INPUT	1
inst_en	读指令使能信号	INPUT	1
mem_read_enM	读数据使能信号	INPUT	1
mem_write_enM	写数据使能信号	INPUT	1
inst_pfn	指令物理页号	OUTPUT	20
data_pfn	数据物理页号	OUTPUT	20
no_cache_i	是否经过 i_cache	OUTPUT	1
no_cache_d	是否经过 d_cache	OUTPUT	1
inst_tlb_refill	指令 TLB Refill 异常	OUTPUT	1
inst_tlb_invalid	指令 TLB Invalid 异常	OUTPUT	1
data_tlb_refill	数据 TLB Refill 异常	OUTPUT	1
data_tlb_invalid	数据 TLB Invalid 异常	OUTPUT	1
data_tlb_modify	数据 TLB Modified 异常	OUTPUT	1
TLBP	访存阶段 TLBP 指令	INPUT	1
TLBR	访存阶段 TLBR 指令	INPUT	1
TLBWI	访存阶段 TLBWI 指令	INPUT	1
TLBWR	访存阶段 TLBWR 指令	INPUT	1
EntryHi_in	写 TLBEntryHi 数据	INPUT	32
PageMask_in	写 TLBEntryHi 数据	INPUT	32
EntryLo0_in	写 TLBEntryHi 数据	INPUT	32

EntryLo1_in	写 TLBEntryHi 数据	INPUT	32
Index_in	查找 TLB 索引	INPUT	32
Random_in	查找 TLB 索引	INPUT	32
EntryHi_out	写入 cp0EntryHi 寄存器数据	OUTPUT	32
PageMask_out	写入 cp0PageMask 寄存器数据	OUTPUT	32
EntryLo0_out	写入 cp0EntryLo0 寄存器数据	OUTPUT	32
EntryLo1_out	写入 cp0EntryLo1 寄存器数据	OUTPUT	32
Index_out	写入 cp0Index 寄存器数据	OUTPUT	32

2.6.2 TLB 结构

查询 TLB 的主要有三种逻辑：第一种是指令虚拟地址翻译；第二种是数据虚拟地址翻译；第三种是 TLB 指令读取数据；为了提高时序，我们针对三种不同的逻辑使用不同时序的方式访问。如下图所示。

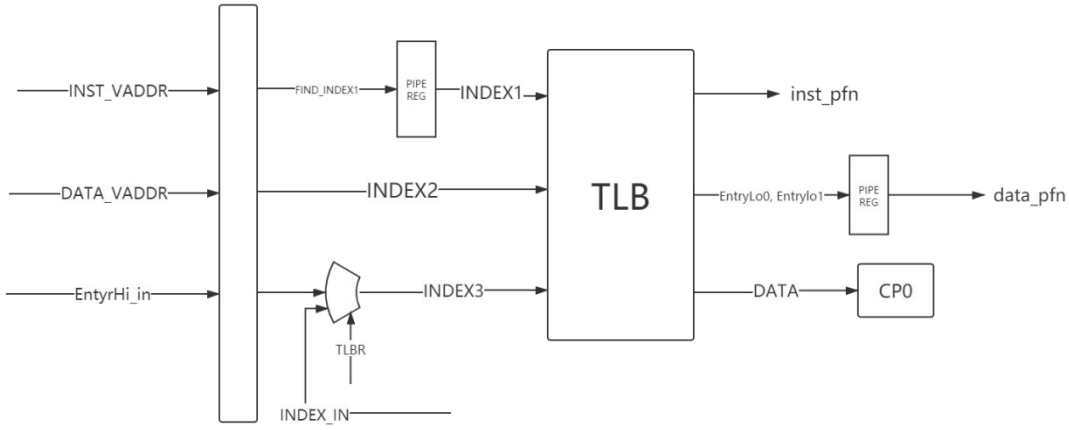

图 10： TLB 结构以及访问逻辑

3. 系统软件

此次系统软件我们成功运行起来了 PMON，可以设置网卡并 ping 通电脑。Linux 由于时间原因没有尝试，load 后运行会输出一些信息后报 PANIC 错误并卡死。

为了运行系统软件，我们的 CPU 进行了如下修改：

1. 实现了 MIPS32 Release1 中除 CP1，JTAG 外的 100 条指令(perf, sync, wait 实现为 nop)，见附录 1
2. 实现 11 种异常，见附录 2
3. 实现 MIPS32 Release1 要求的 20 个 CP0 寄存器(TagHi, TagLo 简易实现)，见附录 3

3.1 PMON

经过我们的尝试，我们最终发现运行 PMON 只需在初赛基础上添加：

1. Status BEV 位和 Ebase 寄存器
2. TLB 指令(TLBP, TLBR, TLBWI, TLBWR)
3. TLB CP0 寄存器(EntryHi, EntryLo0, EntryLo1, Index, Wired, Random)
4. Cache 指令 (IndexInvalid, IndexWriteBackInvalid, HitInvalid, HitWriteBackInvalid)
5. Branch likely 指令（初始化 Nand 时使用到）
6. 非对齐指令(lwl, lwr, swl, swr)（配置网卡 ip 时使用到）
7. 正确设置 CP0 Prid 寄存器（使用龙芯的编号 0x00004220）

曾遇到的一些错误：
1. 初始化信息不打印。这是由于我们实现的 lb, lbu, lh, lhu 指令在不经过 cache 时发出的 axi 信号中的 arsize 为 2’b10，在官方文档中已对该错误进行了说明。
2. Bad eraseblock 从 0 打印到 1023。Cache 指令实现的不正确。
3. 设置网卡 ip 后电脑无法 ping 通。lwl 等非对齐指令实现不正确（如大小端弄反）。

3.2 LINUX

尝试 load 了一下 linxu 内核，g 执行后输出一些信息便卡死了。不过从输出的信息可以判断已经进入了 linux 内核的入口。
由于时间原因，没有进一步的调试。

4. 总结

相较于往年，此次比赛中，我们队伍在性能方面跟系统软件方面都有突破。

在性能方面，我们重构了代码，合理设计 CPU 结构，平衡各个流水线时延；实现了组相联、块多字的 cache，支持 burst 传输，并且直接实现为 AXI 接口，提高了传输效率；添加了分支预测模块。最高频率达到 100MHZ，性能分 55 分。

在系统软件方面，我们实现了 TLB 并成功运行 PMON，进入 linux 内核入口。设计 TLB 跟 cache 的结构为：“virtua index， physical tag”。并且根据 CPU 结构，合理规划 TLB 的查询逻辑，在添加了 8 项 16 页 TLB 的情况下，频率达到 90MHZ。

总之，相较于往年，我们在两个方面均有突破，不过在系统软件上的成绩仍显乏味，需要进一步加强。

5. 参考文献

[1] Patterson, Hennessy. 计算机组成原理与设计：硬件软件接口. 机械工业出版社, 2015

[2] 雷思磊. 自己动手写 CPU. 电子工业出版社, 2014.

[3] 姚永斌 . 超标量处理器设计. 清华大学出版社, 2014.

[4] MIPS32 Architecture For Programmers Volume I: Introduction to the MIPS32™ Architecture

[5] MIPS32 Architecture For Programmers Volume II: The MIPS32™ Instruction Set

[6] MIPS32 Architecture For Programmers Volume III: The MIPS32™ Privileged Resource Architecture

[7] 俞子舒 杨丰远 徐易难 周盈坤,《第二届-myCPU 启动 Linux 指南-国科大》

[8]《基于龙芯 1A 平台的 PMON 源码编译和启动分析》

附录 1：添加的 100 条指令

1 初赛要求的 57 条

2 TLB 指令
TLBP Probe TLB for Matching Entry
TLBR Read Indexed TLB Entry
TLBWI Write Indexed TLB Entry
TLBWR Write Random TLB Entry
WAIT Enter Standby Mode

3 乘累加等指令
CLO Count Leading Ones in Word
CLZ Count Leading Zeros in Word
MADD Multiply and Add Word to Hi, Lo
MADDU Multiply and Add Unsigned Word to Hi, Lo
MSUB Multiply and Subtract Word to Hi, Lo
MSUBU Multiply and Subtract Unsigned Word to Hi, Lo
MUL Multiply Word to GPR

4 非对齐指令
LWL Load Word Left
LWR Load Word Right
SWL Store Word Left
SWR Store Word Right

5 原子指令
LL Link load
SC Store Conditional Word

6 内陷指令
TEQ Trap if Equal
TEQI Trap if Equal Immediate
TGE Trap if Greater or Equal
TGEI Trap if Greater of Equal Immediate
TGEIU Trap if Greater or Equal Immediate Unsigned
TGEU Trap if Greater or Equal Unsigned
TLT Trap if Less Than
TLTI Trap if Less Than Immediate
TLTIU Trap if Less Than Immediate Unsigned
TLTU Trap if Less Than Unsigned
TNE Trap if Not Equal
TNEI Trap if Not Equal Immediate

7 条件赋值和 branch likely 指令
MOVN Move Conditional on Not Zero
MOVZ Move Conditional on Zero
BEQL Branch on Equal Likely
BGEZALL Branch on Greater Than or Equal to Zero and Link Likely
BGEZL Branch on Greater Than or Equal to Zero Likely
BGTZL Branch on Greater Than Zero Likely
BLEZL Branch on Less Than or Equal to Zero Likely
BLTZALL Branch on Less Than Zero and Link Likely
BLTZL Branch on Less Than Zero Likely
BNEL Branch on Not Equal Likely

8 cache 指令
CACHE Perform Cache Operation

9 实现为 nop
PREF Prefetch
SYNC Synchronize Shared Memory
WAIT Enter Standby Mode

附录 2：实现的异常支持

异常名	描述
Int	Interrupt
Mod	TLB 异常（store 且 TLB EntryLo dirty 位为 0，不准写）
TLBL	TLB 异常（取指 refill, invalid 或 load refill, invalid）
TLBS	TLB 异常（store refill, invalid）
AdEL	地址错例外（取指或 load 指令）
AdES	地址错例外（store 指令）
Sys	系统调用
Bp	断点例外
RI	保留指令
Ov	计算溢出
Tr	内陷指令

附录 3：实现的 CP0 寄存器

Index Register (CP0 Register 0, Select 0)
Random Register (CP0 Register 1, Select 0)
EntryLo0, EntryLo1 (CP0 Registers 2 and 3, Select 0)
Context Register (CP0 Register 4, Select 0)
PageMask Register (CP0 Register 5, Select 0)

Wired Register (CP0 Register 6, Select 0)
BadVAddr Register (CP0 Register 8, Select 0)
Count Register (CP0 Register 9, Select 0)
EntryHi Register (CP0 Register 10, Select 0)
Compare Register (CP0 Register 11, Select 0)
Status Register (CP Register 12, Select 0)
Cause Register (CP0 Register 13, Select 0)
Exception Program Counter (CP0 Register 14, Select 0)
Processor Identification (CP0 Register 15, Select 0)
EBase Register (CP0 Register 15, Select 1)
Configuration Register (CP0 Register 16, Select 0)
Configuration Register 1 (CP0 Register 16, Select 1)
TagLo Register (CP0 Register 28, Select 0, 2)
TagHi Register (CP0 Register 29, Select 0, 2)