中国科学院大学

University of Chinese Academy of Sciences

学士学位论文

乱序双发射处理器的自主设计实现

作者姓名：周盈坤

指导教师：胡伟武 研究员 中国科学院计算技术研究所

学位类别：工学学士

专　　业：计算机体系结构

学院（系）：计算机科学与技术学院

2019 年 6 月

Independent Design of the Two-wide Superscalar Microprocessor with Out-of-order Execution

A thesis submitted to the
University of Chinese Academy of Sciences
in partial fulfillment of the requirement
for the degree of
Bachelor of Computer Science
in Computer Architecture

By

Zhou Yingkun

Supervisor: Professor Hu Weiwu

Institute of Computing Technology, Chinese Academy of Sciences

June, 2019

中国科学院大学
学位论文原创性声明

本人郑重声明：所呈交的学位论文是本人在导师的指导下独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明或致谢。本人完全意识到本声明的法律结果由本人承担。

作者签名：

日　　期：

中国科学院大学
学位论文授权使用声明

本人完全了解并同意遵守中国科学院大学有关保存和使用学位论文的规定，即中国科学院大学有权保留送交学位论文的副本，允许该论文被查阅，可以按照学术研究公开原则和保护知识产权的原则公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存、汇编本学位论文。

涉密及延迟公开的学位论文在解密或延迟期后适用本声明。

作者签名：　　导师签名：

日　　期：　　日　　期：

摘 要

本文旨在分析探讨采用乱序、多发射、超标量技术的高性能通用处理器结构设计，以及对其性能的考量、权衡与取舍，并自主设计出一款能够运行嵌入式程序的乱序双发射处理器。为了有效的控制设计的复杂度，在较短的时间内设计出一款能够运行程序的乱序双发射处理器就必须有所简化。最后简化的方案如下：首先 ISA 指令集在 32 位架构的前提下选择了目前最为精简的 RISC-V 中已经被冻结的 RV32I 指令集；其次编写的语言采用由加州大学伯克利分校开发的 Chisel 语言，来提高设计的效率以及有效控制设计的复杂度。最后对比同样是自己设计的单发射五级静态流水和双发射五级静态流水架构，双发射乱序流水架构运行程序所需的周期数明显减少，从而性能有了显著的提高。同时，对 benchmark 中的各个性能测试程序自身指令特点和在不同架构下的执行行为作了细致的比较，量化地分析了影响性能的因素以及这些因素是如何抑制或者提高处理器性能的。

关键词：微处理器，乱序，双发射，RISC-V

Abstract

This paper aims to build a totally self-designed microprocessor with out-of-order, superscalar and multi-issue micro-architecture features to enhance performances of executing programs especially in embedded system area. However, the microprocessor is very hard to implement. In order to control the complexity as will as being able to accomplish the design in this half year, some simplifications have been taken and the final design is as follows. Firstly, as one of the simplest 32-bit ISA, RV32I which is frozen by RISC-V has been chosen to be supported by the microprocessor. Secondly, because of the great ability of complexity control, Chisel which is designed and maintained by University of California, Berkeley has been chosen to be the programming language to design the microprocessor efficiently. In the end, the two-wide superscalar microprocessor with out-of-order execution has been developed successfully. And then, compared with single-issue and dual-issue five stage static pipeline processors, the execution performance of the out-of-order processor improved apparently. Finally, by analysis the dynamic trace of every programs after the processors executing them, several elements are found impacting the performance.

Keywords: Microprocessor, Out-of-order, Two-wide Superscalar, RISC-V

目 录

图形列表

表格列表

符号列表

缩写

IPC	Instruction Per Cycle
CAM	Content Addressable Memory
RAM	Ramdom Access Memory
LW	Load and Store instruction/request
ISA	Instruction set architecturey
TLB	Translation lookaside buffer
FSM	Finite state machine
BOOM	The Berkeley Out-of-Order RISC-V Processor
PC	The program counter
LDQ	The load queue
STQ	The store queue
ALU	Arithmetic logic unit
MIPS	Microprocessor without Interlocked Pipelined Stages
RISC	Reduced instruction set computer
ROB	Reorder Buffer
BTB	Branch Target Buffer
RAS	Return Address Stack

第 1 章　引言

1.1　目的与意义

随着集成电路晶体管数量的不断增加，能够有更多的资源来支持更为复杂但效率更高的处理器微结构特性，包括乱序调度，超标量发射和指令缓存。得益于这些复杂的特性，微处理器能够从五级流水的性能瓶颈中解放出来，在存储墙愈加严重的现状下依然能够得到性能的提升。自主设计出一款乱序双发射处理器正是出于对计算性能提高的不懈追求。同时，在设计、调试、评测、分析、优化的过程中能够积累高性能通用处理器设计的宝贵经验。

和静态流水线相对固定的格式不同，乱序多发射的处理器在设计中更加灵活。很多方案、参数糅合在一起复杂地影响着处理器的性能，需要比较、权衡与取舍。在设计的过程中，对处理器设计空间的初步探索，同样具有意义。

1.2　研究背景

近年来，RISC-V 开源体系结构的兴起，使得很多软硬件实力相对于大公司薄弱很多的研究个体受益良多。RISC-V 开源的项目越来越多，其中有微处理器的设计实现如 Rocket 和 BOOM; 有更抽象的电路描述语言如 Chisel; 也有 RISC-V 的指令模拟器如 Spike，可以在上面调试软件，或者得到程序的执行写回信息用来调试处理器；还有已经移植完成的上层应用软件栈如 Linux，GCC 和 LLVM。这些开源的项目大大降低了研究个体独立设计出一款高性能处理器以及在上面运行软件栈的难度与门槛。

从指令集 ISA，编程语言，开源项目三个角度来看，RISC-V 开源体系结构对毕业设计都有很多助益。

1.2.1　MIPS 和 RISC-V

设计一款具体的处理器要对应于具体的 ISA，这里有两个较为简单的指令集可供选择：一个是 MIPS 的 MIPS32 子集，一个是 RISC-V 的 RV32I 子集，两者都是 RISC 的典型 ISA。MIPS 体系结构从最开始 1985 年推出的第一代指令集 MIPS I 和 R2000 处理器起，已经有三十多年的历史；而 RISC-V 作为加州大学伯克利分校的 RISC 系列的第五代指令集，在其师生大力推广下成为了近年来热度最高的开源指令集。

设计中最后选择 RV32I 而不是 MIPS32 有多方面的考虑。首先，出于对新事物的好奇和新鲜感；其次，由于本科的教学的参考 ISA 是 MIPS，通过本科的学习和工程实践对于 MIPS 的有了初步的认识，也能够体察到 MIPS 在 ISA 设计中的一些缺陷：

(a) 延迟槽的引入，这个当初对单发射五级流水从体系结构上做出的优化被日后的工程实践证明是一个历史的包袱。因为分支指令后面紧跟一条延迟槽的设计仅仅适用于像单发射五级静态流水这样的简单的设计中。超标量、乱序投机取指的高性能处理器取指仍然需要做转移猜测，所以延迟槽的引入反而增加了高性能处理器取指和中断例外设计的复杂度。

(b) 为了存储定点乘除法结果而单独加入了 HILO 寄存器，再通过 MFHI, HFLO, MTHI, MTLO 这四条指令与通用寄存器堆进行相互搬运也会增加处理器设计的复杂度。

(c) 继续深入分析，MIPS 的特权态的设计显得很凌乱，对于特权模式的处理机制是逐渐增量式添加的方式，并不是一开始就规划地非常清楚，欠缺体系。

(d) 在 80 年代 MIPS 诞生之初也没有考虑到 64 位虚拟地址空间的需求和嵌入式领域指令压缩的需求，导致指令空间设计考虑不周，64 位尚能够与 32 位兼容，但是为嵌入式压缩指令引入的 microMIPS 就是重新设计的与原有的 MIPS 32/64 并不兼容的另一套 ISA。而且压缩的效果也不及 ARM 的 Thumb-2 和 RISC-V 的 RV32I+RVC。对比不同压缩指令集编译出来的统一程序的二进制代码大小如图1.1所示:

Benchmark	ISA	ARM Thumb-2	microMIPS	x86-32	RV32I+RVC
Insertion Sort	Instructions	18	24	20	19
	Bytes	46	56	45	52
DAXPY	Instructions	10	12	16	11
	Bytes	28	32	50	28

图 1.1 不同压缩指令集编译插入排序和双精度浮点乘加程序的代码大小比较。(Patterson et al., 2017) 图 7.2.

Figure 1.1 Instructions and code size for Insertion Sort and DAXPY for compressed ISAs. Figure 7.2 of (Patterson et al., 2017).

ISA 体系结构的演进发展已经有 50 多年的历史了，在诸如上述设计教训的沉淀下，最为基础核心的设计趋向于收敛。所以比 MIPS 晚诞生 20 多年的 RISC-V 能够站在后来者的角度审视之前诞生的众多指令集的优缺点，从设计开始就有一套清晰的规划，摒弃了很多包括上文提及的 MIPS 指令集设计中的缺陷，做到 32 位，64 位，压缩变长指令集很好的统一，以简洁规整的形式呈现出来，最

大化的简化了硬件指令译码的逻辑。同时，因为有了系统的规划，虽然目前特权态的有些规定尚不明确，高性能的 SIMD 指令亦或是向量指令也不完善，但是对于处理器运行的模式规划，明确而有章法。另外，指令编码空间设计合理使得指令的可拓展性极强，以备日后之需。下面通过对比 MIPS 来具体分析 RISC-V 的特点以及简洁优美的设计考虑。

首先不同于几乎所有先前的 ISA，RISC-V 是模块化的。核心是基础的 ISA — RV32I，简单而不失强大，足以运行整个软件栈。同时，RV32I 是被冻结，也即永远不会改变，是稳定的，这样，对硬件设计人员或者软件设计人员都友好。其他的扩展功能的指令以可选的方式添加，比如也已经被冻结的 RV64I 基础系列和 M(乘除法)、A(原子指令)、F(单精度浮点)、D(双精度浮点) 等拓展系列 (Patterson et al., 2017)。

一、 用户态的对比：

(a) RISC-V 取缔了延迟槽的设计，使得 ISA 和处理器的微结构相对独立，同时编译出来的二进制代码量通常会比 MIPS 少。

(b) RISC-V 取缔了 HILO，但是 32 位的乘法和除法结果都是 64 位的，如何解决乘除法的结果存储问题。RISC-V 选择软件来解决，对于每一个乘除法编译器都会得到两条连续的指令，一条得到低 (商)32 位，一条得到高 (余数)32 位，分别存储在不同的寄存器里。这样做的优势在顺序流水架构下不明显，但是在乱序的架构下就非常明显。本质上来讲，取缔 HILO 寄存器，把结果拆开存入通用寄存器中是一种统一编址的形式，乱序的重命名就只需要管理通用寄存器的映射关系即可，这样简化了重命名的结构。另一方面，就算有 HILO 寄存器，后续的指令要以乘除法的结果为源操作数时，也是要先通过 mfhi，mflo 指令读到通用寄存器中，多传导一次的导致效率不高的同时，还会占用指令空间。另外拆成两条连续指令来执行对于乱序的处理器只会增加少许额外的逻辑，这是因为 ROB 会给每一条指令分配一个标识符 id 号，检测到连续的 id 号的指令就可以同时出结果，不用对乘除法算两次。

(c) RISC-V 取缔了 MIPS 中的条件移动 (conditional-move) 指令，这是出于乱序设计简化的考虑。

(d) RISC-V 没有算术指令溢出例外的规定，所以硬件不做处理。

```
add t0, t1, t2
slti t3, t2, 0
slt t4, t0, t1
bne t3, t4, overflow
```

将其完全移至软件处理，示例代码如上 (Patterson et al., 2017)。

(e) 不像 MIPS 用 LWL，LWR 来支持地址的非对齐访问。由于 RISC-V 指令可以是变长的，所以非对齐的访问是自然支持的。

(f) RISC-V 支持 PC 相对地址访问。在 RISC-V 中设计了 AUIPC (add upper immediate to pc) 指令，该指令从指令中的 20 位立即数的基础上低 12 位填充 0 构成 32 位偏移量与当前的 PC 相加结果存到 rd 寄存器中。这样做的好处在于代码数据整体拷贝时依旧能够运行原有的程序，因为相对位置保持不变。下表1.1是用到 AUIPC 指令比较常用的情形 (Andrew Waterman, 2017b)：

表 1.1 AUIPC 指令常用的情形。

Table 1.1 The popular usage of AUIPC instruction.

Meaning	Base Instruction(s)	Pseudoinstruction
Load address	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	`la rd, symbol`
Load global	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	`l{b|h|w|d} rd, symbol`
Store global	`auipc rd, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rd)`	`s{b|h|w|d} rd, symbol`
Floating-point load global	`auipc rd, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rd)`	`fl{w|d} rd, symbol`
Floating-point store global	`auipc rd, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rd)`	`fs{w|d} rd symbol`
Call far-away subroutine	`auipc x6, offset[31:12]` `jalr x1, x6, offset[11:0]`	`call offset`
Tail call far-away subroutine	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	`tail offset`

二、 特权态的对比：

MIPS 的特权态的管理用到的是 Coprocessor 0(以下简称 CP0)，负责对虚实地址和例外处理进行管理。缺陷在于记录特权状态的寄存器的地址空间仅有区区 5 位 32 个，如果需要用到更多，则要引入 CP1、CP2 等等，导致整体地址空间的管理是独立而不连续的。为了避免这个不足，RISC-V 的特权态寄存器有足足 12 位的地址空间，也即最多 4096 个寄存器可用。这 4096 个寄存器统称为

Control and Status Registers (CSR)，其地址空间的分配有清楚的规定，可以参见 RISC-V 特权态手册 *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*(Andrew Waterman, 2017a)。

虽然 RISC-V 在运行操作系统等大型软件上的经验还不如 MIPS 雄厚，一些手册中的规范没有明确化，比如处理器执行的初始地址需要自定义；虚拟地址的分配没有明确的规定 (包括访问 IO 外设的地址、可以被缓存的地址)。但是其特权态的规范在这几年快速地发展，运行像 Linux 这样的操作系统已经绰绰有余。

用模型的角度来分析，特权态本质上是用户态模型之上更高等级的状态集。这些状态同样是由寄存器来维护的。整个模型通过输入特权态指令来改变以及管理这些状态。那么怎么衡量一个 ISA 对于不同等级模式的刻画干不干净，就要看状态集以及这些状态的变化规不规整。

如图1.2，RISC-V 非常规整地将处理器的状态分为了 3 个等级:

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	*Reserved*	
3	11	Machine	M

图 1.2 RISC-V 的 3 个等级及其编码。(Andrew Waterman, 2017a) 表 1.1.

Figure 1.2 RISC-V privilege levels and their encoding. Table 1.1 of (Andrew Waterman, 2017a).

而这 3 个等级的 2 bits 编码均体现在特权指令的编码 (用户态的指令不用这么编码是因为无论哪个等级都可以执行这些指令，而且省去这两位能够增加 3 倍的指令编码空间) 和特权寄存器空间的编码。这样什么等级的指令执行在什么等级状态下处理器上，要修改什么等级的寄存器，非常的清晰明了。

在三个等级中最为基础的就是 Machine Level(以下简称 M 等级)。而 M 等级中基础的控制寄存器其实基本上和 MIPS CP0 寄存器是一一对应的。列举见表1.2。

这里比较有特色的寄存器是 mscratch。如果编写过或者看过 MIPS 上的操作系统内核，就会知道比如 context switch 和 TLB 的例外处理都需要用到 K0 和 K1 寄存器，这两个寄存器就是专门留给操作系统用于临时存储数据地址用的。虽然这样很高效，但是白白浪费了两个通用寄存器。另外一方面像 context switch 和 TLB 的例外处理是与 M 等级有关的，所以从设计的归类上来讲也应该归特权级别的寄存器管理而不是通用寄存器。这也是 RISC-V 更为干净的一种体现。

在特权态中最为重要的一方面是内存管理，而一个最基本的考虑就是用户

表 1.2 与 MIPS CP0 寄存器功能类似的 CSR 列表。

Table 1.2 The List of CSRs similar to MIPS CP0.

简称	全称	功能备注
`mtvec`	*Machine Trap Vector*	存储例外的跳转地址
`mepc`	*Machine Exception PC*	存储例外发生的指令 PC
`mcause`	*Machine Exception Cause*	指示例外发生的原因与类型
`mie`	*Machine Interrupt Enable*	中断使能向量
`mip`	*Machine Interrupt Pending*	列举当前 pending 住的中断
`mtval`	*Machine Trap Value*	与 MIPS 中的 BADADDR 功能一致，存储例外访存地址
`mscratch`	*Machine Scratch*	一个指令长度的数据临时存储
`mstatus`	*Machine Status*	与 MIPS 中的 STATUS 功能类似，存储处理器状态

态不可信赖的程序要严格限制其只能访问到自己内存范围的内容。从 RISC-V 的手册 *The RISC-V Instruction Set Manual Volume II: Privileged Architecture* 中，可以看得出其设计者在这方面下了很大的功夫。

首先，RISC-V 在 M 等级和 User Level(以下简称 U 等级) 之间设计了一种内存保护机制：Physical Memory Protection (PMP)。这样 M 等级就可以配置 U 等级程序能够访问的内存区域以及访问的权限。上图1.3上半部分是 PMP 地址寄存

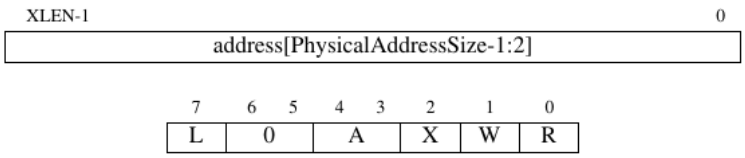


图 1.3 PMP 地址和配置寄存器。地址寄存器右移了 2，如果物理地址比 XLEN-1，那么高位填 0. R, W 和 X 域授权读，写，执行权限。A 域设置 PMP 模式，L 域锁住对应的 PMP 地址寄存器。(Patterson et al., 2017) 图 10.7.

Figure 1.3 A PMP address and configuration register. The address register is right-shifted by 2, and if physical addresses are less than XLEN-2 bits wide, the upper bits are zeros. The R, W, and X fields grant read, write, and execute permissions. The A field sets the PMP mode, and the L field locks the PMP and corresponding address registers. Figure 10.7 of (Patterson et al., 2017).

器，一般处理器会实现 8-16 个这样的寄存器；下半部分是 PMP 配置寄存器，为 8-bit 向量存储在 pmpcfg 的 CSR 寄存器里，负责授权或者禁止读、写、执行权限。当处理器在 U-mode 试图想要取指或者执行访存请求时，其地址将会被和 PMP 地址寄存器进行比较，如果地址大于等于 i 号 PMP 而小于 $i+1$ 号 PMP。那么

$i+1$ 号的 PMP 配置寄存器将会规定访问内存的模式，如果实际的访问方式逾越了配置寄存器里规定的权限，就会引发例外 (Patterson et al., 2017)。

对于嵌入式系统，PMP 机制能够非常有效的对内存实行保护。但是其有两个缺点使之不适合更为复杂的系统和通用计算，分别是：

(a) 只支持最多 16 个内存区域，不具备扩展性。

(b) 这些区域都是连续的，有点像操作系统早期的**段**的概念，所以对内存细粒度的片段化支持不好。

所以 RISC-V 的 ISA 必须支持基于页的虚拟内存机制。而这个特征直接主导了介于 M 等级和 U 等级中间的等级 — supervisor mode(以下简称为 S 等级) 的产生。

但是引入 S 等级又会产生一个问题。RISC-V 中所有的例外都是要将处理器的控制权转移到 M 等级的例外处理程序。但是大多数的 Unix 系统的系统调用包括例外是陷入系统内核，而系统内核又是运行在 S 等级的。如果先由系统将用户进程切换到内核，然后再由处理器硬件将内核 S 等级切换到 M 等级处理，最后 M 等级要重新路由到 S 等级，交给系统内核去退出系统调用 (Patterson et al., 2017)，这样效率就会大大降低。所以 RISC-V 提供了例外授权机制 (exception delegation mechanism)，这样中断和同步例外就能旁路到 M 等级，选择性的授权给 S 等级。而 CSR 寄存器 `mideleg` (Machine Interrupt Delegation) 和 `medeleg` (Machine Exception Delegation) 正是这个机制的载体。对应于 `mip` 和 `mie` 寄存器的 exception code，举例来说，`mideleg[5]` 对应于 S 等级的时钟中断，如果被置上，S 等级的时钟中断将会把控制权转移到 S 等级的 exception handler 上而不是 M 等级的 exception handler；同样，如果 `mideleg[15]` 被置上，则会将 store page faults 授权给 S 等级来处理 (Patterson et al., 2017)。不过需要注意的是，在某等级发生的例外永远不会转移控制权给更低的等级，具体来说就是在 M 等级发生的例外就只能 M 等级来处理；S 等级发生的例外可能由 M 等级或者 S 等级来处理，取决于 delegation 的配置，但永远不会是 U 等级Patterson et al. (2017)。

RISC-V 的分页方案命名方式为 **SvX**，比如 32 位的虚拟地址是 **Sv32**, 其支持 4GiB 的虚拟地址，有两级页表，分别是 2^{10} 个 4MiB 的页表，每个页表又有 2^{10} 个 4KiB 的基页。下图1.4是 **Sv32** 和 **Sv39** 的页表项 (PTE)：**Sv39** 是 $2^9 \times 2^9 \times 2^9 \times 2^{12}$ 的模式，是 3 级页表。如果物理地址比 2^{39} 还要大，RISC-V 还提供了 4 级页表的 **Sv48** 方案。

S 等级用 satp (Supervisor Address Translation and Protection) 控制和状态寄存

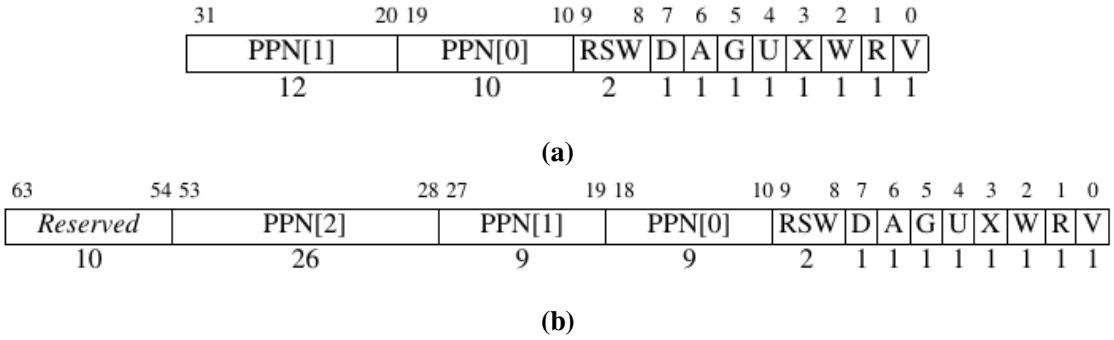


图 1.4 RISC-V 分页。(Patterson et al., 2017) 图 10.10 和 10.11. (a)RV32 Sv32 页表项, (b)RV64 Sv39 页表项。

Figure 1.4 RISC-V SvX. Figure 10.10 and 10.11 of (Patterson et al., 2017). (a)An RV32 Sv32 page-table entry(PTE), (b)An RV64 Sv39 page-table entry(PTE).

器管理分页系统，如下图1.5：在处理器进入 S 等级之前，M 等级首先会写 0 到

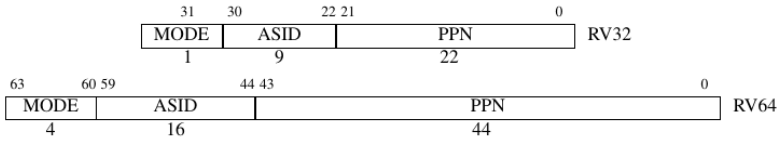


图 1.5 satp 控制和状态寄存器 (Patterson et al., 2017) 表 10.12.

Figure 1.5 The satp CSR. Figure 10.12 of (Patterson et al., 2017).

satp 寄存器，禁止分页；然后等到进入 S 等级在初始化页表之后，处于 S 级内核就可以开始对 satp 寄存器进行操作来对内存进行分配了。

1.2.2 Verilog 和 Chisel

设计的实现是将抽象的逻辑层面想法概念用具体的语言严谨地描述出来，同样，设计处理器这样的硬件需要硬件的描述语言。当今主流的描述语言仍是 90 年代开发的 Verilog。该语言设计参考的是 C 语言，最开始设计出来用作电路功能的仿真，因而有很多语法只适用于软件的仿真，不能作为物理电路的综合布局布线。可被综合的最常用的语句是组合逻辑的 assign 语句块和时序逻辑的同步 always 语句块。事实上这两种语句组合就能够满足绝大多数的电路设计的需求，其中就包括双发射乱序处理器。但是就像能够描述所有软件程序的汇编语言一样，Verilog 的抽象等级太低，细节太多，描述电路的能力先天不足，主要体现在：

(a) 语句的逻辑集成度不高，时常一个简单的逻辑代码量却不少。

(b) 变量名、函数名、参数名的管理机制原始落后，甚至都没有像 C 语言的 struct 结构体这种聚合化的设计。代码量大时，各个符号名称的关联往往复杂而

混乱。

(c) 常用的简单逻辑代码复用度不高，代码复用基本上只能用 Module 这样一种单一的写法，容易冗余。

(d) 对多维的向量数组描述和聚合化操作支持不足。

(e) 没有成熟的类型系统，无论是 Reg 还是 Wire 都是对物理信号的刻画描述，粒度太小。

(f) 代码中容易出现组合环，而且现有的编译器很难跟踪。

(g) 要实例化不同参数的同一模块，只能一个个实例，做不到参数的数组化。

设想原本逻辑已经复杂不堪的双发射乱序处理器采用 Verilog 来描述，又会因为语言层面的弱势，使得实现起来异常的繁琐，复杂度得不到有效地控制。对这样的大工程的搭建、调试以及持续的优化、增量式演进都带来不小的挑战。

硬件工程师迫切需要一个更为高级的语言来解决上述 Verilog 种种缺陷，这就是 Chisel。

Chisel 没有那么神秘，并不是变革性的另一套硬件描述，而是包装在 Verilog 之上具有面向对象和函数化编程特性的 Scala 脚本语言，是一种以 Verilog 中上述两种语句为基本块的更上一层的抽象。多一层的抽象就是为了解决 Verilog 表示能力不足的缺陷。比如 Chisel 支持面向对象里的类和对象、接口以及多态与继承的特性，可以很好地复用代码，有条理地管理变量名、方法名和参数名；另外函数式的编程可以用来很便捷的描述简单的电路逻辑，增强对多维数组操作的支持。对应的，Chisel 的编译器主要工作就是将抽象的 Scala 代码转化为低级的 Verilog，而且是完全可以被综合和物理实现的 Verilog (所以是否是可综合的电路根本不需要顾虑)。类似于 JAVA 编译器将 JAVA 编译成汇编代码。

Chisel 和 JAVA 都是强类型的语言 (Chisel 的类型继承关系如图1.6a (https://chisel.eecs.berkeley.edu/2.2.0/manual.html), 在编译的时候会做类型检查，从而避免很多因为类型不当导致的逻辑错误。而 Verilog 和汇编都是没有类型的概念的，一个 32 位的逻辑寄存器，汇编可以任意地操作而不用在乎这个寄存器代表的是整型还是字符型，同样，Verilog 对一个 32bit 的 Wire 和 Reg 向量也可以任意操作。正是因为这种毫无约束的操作，使得用汇编或者 Verilog 编程都会比 Chisel 或者 JAVA 来得繁琐而易出错。但是略显不同的是，一个初学者可以在不知道汇编语言的情况下就能够掌握 JAVA 语言这样的抽象等级高的语言，但是却不能在没有熟练运用 Verilog 中的两个基本语句的前提下掌握 Chisel。换言之 JAVA 对于汇编的抽象是彻底的，而 Chisel 对于 Verilog 的抽象是不彻底的，

所以 Chisel 适用于有一定硬件设计工程经验的人，而不适用于上手设计电路的初学者。

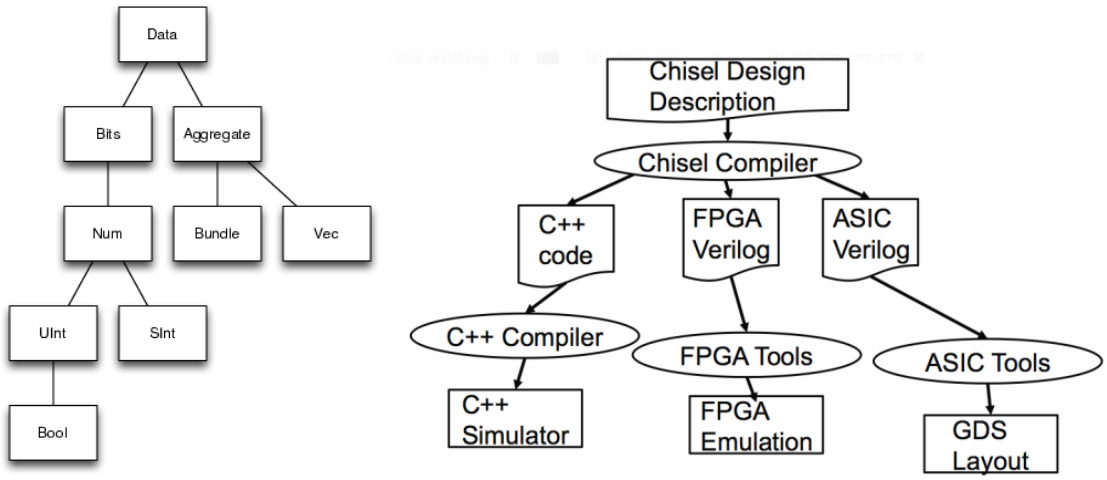


图 1.6 Chisel 的类型层次结构和设计流。(a) Chisel 类型系统，(b) 基于 Chisel 的设计流程框图。(John Wawrzynek, 2016) 图 1.

Figure 1.6 Chisel type hierarchy and Chisel Design Flow. (a) Chisel type hierarchy, (b) Chisel Design Flow. Figure 1 of (John Wawrzynek, 2016).

从 Chisel 设计流程框图1.6b可以发现，首先 Chisel 通过编译器不仅能够编译成 Verilog，还能编译成 C++ 代码再通过 C++ 的编译器得到 C++ 的模拟器。工程实践证明，C++ 模拟器仿速度非常快，跑 500 遍 dhrystone 仅仅需要 1 秒。其次 Chisel 编译器可以针对不同的硬件平台编译采用不同的优化。甚至得到的电路时序比人写的 Verilog 对应的电路还要好。

但是电路描述高级语言化首先需要克服电路方面特有的两大障碍：

(1) 时序逻辑高级语言应该如何描述？

(2) 随着语言的高级化，会不会使得电路的编写模糊化，使得所写的电路很难对应到实际的物理电路上，就像高级语言对垃圾回收做了透明化一样。这恰恰是硬件的工程师所不愿意看到的，因为电路设计要了解电路的所有实现细节。

那么 Chisel 是如何成果性地打消上述的两个障碍和顾虑的呢？那就要看 Chisel 是如何进行抽象的。

(a) 组合电路的对应于 Verilog 中的 Wire，赋值用 assign 语句，也可以直接在 Wire 型变量定义的时候赋值。而 Chisel 首先用的就是两类最基本的数据类型来描述，分别是 UInt 和 Bool。Bool 只是为了强调变量是 1 bit 的代表真假布尔逻辑的变量。而 Wire 型变量的赋值有两种形式：

- 初始定义的 = 运算符

```
val pc = Wire(UInt())
```

如上并不是真正的赋值，而是类型的申明。UInt 的括号中可以传入位宽的参数，如果省略，Chisel 会在编译的时候自动在以后的真正赋值中推导出来，所以还是非常人性化的。

- 因为 val 在 Scala 中是不可变量，也就是变量名指针所指的对象不能更改，所以 chisel 中引入**:=** 运算符来进行再赋值。

```
pc := pcReg + 4.U
```

(b) 时序电路对应于 Verilog 中的 reg，赋值需要用到 always 语句。而在 Chisel 对其进行了抽象，首先它没有具体的类型，是一个 Reg 的元器件。其次这个元器件有两端 — input 和 output。而 output 可以理解为 input 信号延迟一周期的副本。严格来讲，Reg 型变量的类型可以定义为 input 端所连的变量的类型，在 reg 类型变量的定义中还可以指明电路复位的初始值。如下例：

```
val pcReg = Reg(next = pcNext, init = 0.U(32.W))
```

在当前的版本的 Chisel 中，时钟和复位是全局信号，是隐式申明的 (Jonathan Bachrach, 2017)。这样 Reg 型的变量的更新可以做到简单一行的赋值代码，和组合逻辑 Wire 型变量一样使用**:=** 运算符。

(c) 由于赋值运算符的统一，Chisel 对于 wire 类型和 reg 类型的变量统一抽象为了电路上的节点 (node)。整个电路图就是由 node 组成的图。具体来讲，如果是纯组合逻辑，那么这个图就是有向无环图。所以 Chisel 正是通过简单的图算法，对设计中出现的组合环进行非常精确的报错，指出相同的开始节点和结束节点，从而规避了仿真中出现的奇怪的现象。唯一存在有环的情况是时序电路。而且依据这个图，可以用 verilator 工具生成高速的 C++ 的 simulator(Jonathan Bachrach, 2017)。

(d) 变量统一抽象为电路节点反过来将**:=** 运算符的意义统一了 — 将右手表达式代表的节点组合逻辑连线到左手变量代表的目的节点上。但是略有差异而且需要注意的是，如果涉及到 Reg 型变量 x，因其具有两端，若 x 出现在右手表达式，用的是 output 端；若 x 出现在左手表达式，用的是 input 端 (Jonathan Bachrach, 2017)。

(e) 有了基础的抽象，Chisel 可以在其上利用面向对象的方法和继承的语言特性自定义构建更为大型，更为抽象的数据结构。

(f) 类比在 Verilog 对于模块的刻画用了 module 的写法，Chisel 采用用户自定义类去继承称为 Module 的父类的写法，并且对于模块的接口，还定义了一个 IO 的类。如下例 (Jonathan Bachrach, 2017)：

```
class Mux2 extends Module {
  val io = IO(new Bundle{
    val sel = Input(UInt(1.W))
    val in0 = Input(UInt(1.W))
    val in1 = Input(UInt(1.W))
    val out = Output(UInt(1.W))
  })
  io.out := (io.sel & io.in1) | (~io.sel & io.in0)
}
```

这里的 Bundle 是一个 Chisel 里的基类 (见图1.6a)，类似于 C 里面的 *Struct*。但是如上述例子所展示的，可以不需要预先定义而直接新建一个匿名的结构体，然后作为参数传入 IO 的构造方法中，最后将其定义为 io 变量。这种写法相比于 Verilog 里最大的好处在于在 Module 的内部的逻辑中，Chisel 的代码更加清晰。端口因为都带有 *io.* 的前缀，引用或者赋值一目了然。这样增加了代码的可读性。

经过上面的分析，Chisel 语言依旧能够呈现统一时钟复位电路的所有细节。如果尚有缺点，也是因为目前屏蔽了 clock 和 reset，默认为统一时钟同步复位，还不支持异步电路。这个在 Chisel 的文档里也给出了解释：纵然有交叉时钟的设计需求，但是现代设计方法中也是在每一个同步的电路岛屿中开发与验证的 (Jonathan Bachrach, 2017)。

下面用三个例子来展示 Chisel 在电路设计的简便之处：

(1) 下面代码 (Jonathan Bachrach, 2017) 灵活运用了面向对象和可配置参数化函数 (类似于 C++ 的 template) 的语言特性。首先是通过面向对象的继承来充分复用已有的代码逻辑，其次不同功能的 Filter 利用 λ-函数作为参数传入，来充分复用 Filter 共性的逻辑，与此同时并没有掩盖电路的细节。

```
abstract class Filter[T <: Data](dtype: T) extends Module {
  val io = IO(new Bundle {
    val in = Input(Valid(dtype))
    val out = Output(Valid(dtype))
  })
}
```

```
class PredicateFilter[T <: Data](dtype: T, f: T => Bool) extends
Filter(dtype) {
  io.out.valid := io.in.valid && f(io.in.bits)
  io.out.bits  := io.in.bits
}
object SingleFilter {
  def apply[T <: UInt](dtype: T) =
  Module(new PredicateFilter(dtype, (x: T) => x <= 9.U))
}
object EvenFilter {
  def apply[T <: UInt](dtype: T) =
  Module(new PredicateFilter(dtype, (x: T) => x(0).toBool))
}
class SingleEvenFilter[T <: UInt](dtype: T) extends Filter(dtype) {
  val single = SingleFilter(dtype)
  val even   = EvenFilter(dtype)
  single.io.in  := io.in
  even.io.in    := single.io.out
  io.out        := even.io.out
}
```

(2) 拷贝于毕业设计代码中的处理器核顶层文件。可以看到其中颇具特点的运算符 **<>**，在 Chisel 的术语里叫做 *Bulk Connections*，用于带输入输出端口 Bundle 的电路信号之间的整体互连，大大简化了模块实例化后互相连线的逻辑。

```
class Core(implicit conf: CPUConfig) extends Module with BTBParams {
  val io = IO(new Bundle {
    val imem = new AxiIO(conf.xprlen)
    val dmem = new MemPortIo(conf.xprlen)
  })
  val frontEnd = Module(new FrontEnd)
  val backEnd  = Module(new BackEnd)
  frontEnd.io.mem  <> io.imem
  backEnd.io.mem   <> io.dmem
  frontEnd.io.back <> backEnd.io.front}
```

(3) 拷贝于毕业设计代码中的写回结果总线监听逻辑。发射队列有 nEntry 项，每一项存有一条待发射的指令，需要监听两个源操作数，写回总线结果对应于代码中的 *io.bypass*。下面短短几行的代码里就涵盖了将写回总线的寄存器地址和有效使能信号逐一与发射队列里的每一项待发射指令的每一个源操作数进行

比对，判断是否成功监听的逻辑。Chisel 对于数组聚合化操作的强大描述能力可见一斑。

```
for (i <- 0 until nEntry) {
  for (j <- 0 until 2) {
    inst_ctrl.snoop(i)(j) := issue.snoop(i)(j).valid ||
    io.bypass.map(b =>
    b.addr === issue.snoop(i)(j).addr && b.valid).reduce(_||_)
  }
```

1.2.3 RISC-V 开源仓库

RISC-V 近年的流行和其开源仓库是相辅相成的，目前在开源的工程下至较为成熟的处理器设计实现以及整套 SoC 的环境，上至各种软件栈一应俱全。使得大至一个国家如印度，中至一些企业组织如 lowRISC，小至一个个独立的研究个体都愿意加入到 RISV-C 的开发中来。

具体地，对设计影响较大的几个开源工程有开发语言 Chisel，双发射乱序处理器 BOOM 及其设计文档和 RISC-V 整套在 x86 host 机上处理器设计的调试环境。

1.3 研究范围

一个人做出一个功能齐全，能够运行操作系统和整套软件栈的双发射乱序处理器，在半年时间内几乎是不可能完成的。所以研究的范围应该集中在乱序的调度，分支的预测以及取指部件的高效取指上，最后能够运行简单嵌入式程序来初步验证处理器正确性和评测性能即可。

在摘要中已经提及了必要的简化方案，更具体的方案包括：不支持 MMU 以及分页机制；为了实现取指宽度大于 1，自行编写了指令高速缓存代码，并且对外是标准的 AXI 接口，测试的时候带有随机的取指延迟；处理器中的访存部件虽然支持访存延迟，但没有实现数据高速缓存，对外也不是 AXI 接口，访存过程几乎是一个周期的同步时序。

第 2 章　相关工作

在超标量乱序处理器的优秀工作中，前有经典的 Alpha 21264 和 MIPS R10000 成熟产品，后有开源的 RISC-V BOOM 处理器设计。它们在微结构上的设计值得参考与借鉴。

2.1　Alpha 21264

Alpha 21264 是处理器设计历史上高性能的代表之作。下面从文献 *THE ALPHA 21264 MICROPROCESSOR*(Kessler, 1999) 分析 Alpha 21264 的微结构设计。

首先 Alpha 21264 整体的流水级模块级框图2.1一共切分成了 7 个大的阶段。

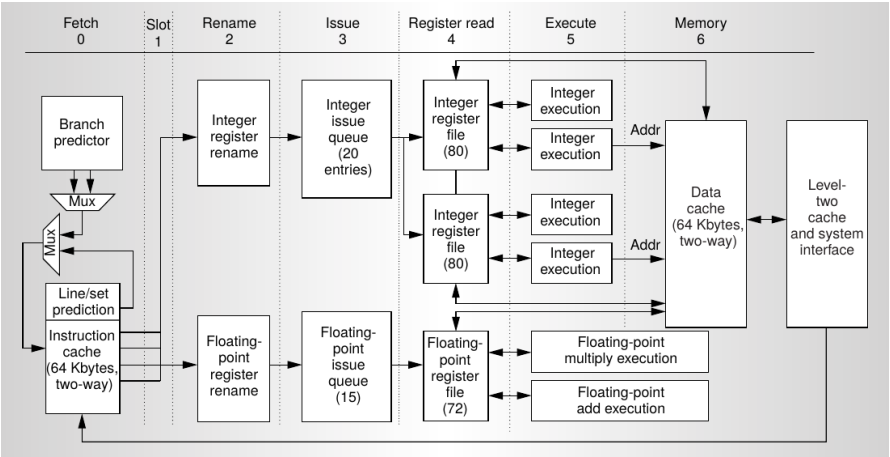


图 2.1 Alpha 21264 流水线阶段。(Kessler, 1999) 图 2.

Figure 2.1 Stages of the Alpha 21264 instruction pipeline. Figure 2 of (Kessler, 1999).

从第 0 级的取指开始往流水线的下游方向逐级分析一些设计的亮点。

(a) **取指**。21264 为了提高取指的效率，着重强调了两个设计，一是 icache 的行和路预测，二是分支跳转预测。

由于 21264 的 icache 采用的是两路组相连，一种方案是把行对应的两个路都读出来，然后在做二选一逻辑，不过这样电路延迟就增加了，而且可拓展性也不好，到四路的延迟更大。所以 21264 选择了另外一种方案 — 路预测，采用和分支预测相似的两位饱和计数的技术，对于大多数程序而言正确率都能达到 85%~100%，猜得准的同时猜错代价很小，绝大多数情况下只有 1 个周期 (Kessler, 1999) 损失。所以采用路预测对电路主频的提高收益大于损失的周期数。

与 icache 路预测情况不同，因为 21264 最多容纳 80 条指令乱序执行，转移指令猜错代价很大，所以分支预测就成为了提高 21264 效率非常重要的一环。为了追求预测的正确率，21264 实现了复杂的锦标赛预测方案，动态的选择两种类型的分支预测器的预测结果 — 一是用跳转指令本身的跳转历史 (local history)，二是用全局的跳转历史 (global history)。预测准确率比分别使用上述两种策略以及更大的表都要好，达到 90%~100%(Kessler, 1999)。具体来看，21264 一共维护了 3 个表，结构如图2.2所示。

- 一张局部历史表，能够存放 1024 条指令 10 bits 的自身跳转历史，面积 1024×10 bits.
- 一张全局预测表，配以一个 12 bits 的全局历史，所以有 4096 项，每一项是 2 位饱和计数器，面积 4096×2 bits.
- 一张选择表，用来选择两种预测机制中更好的一个，和全局预测表一致，共有 4096 项，也采用两位饱和计数器，面积 4096×2 bits.

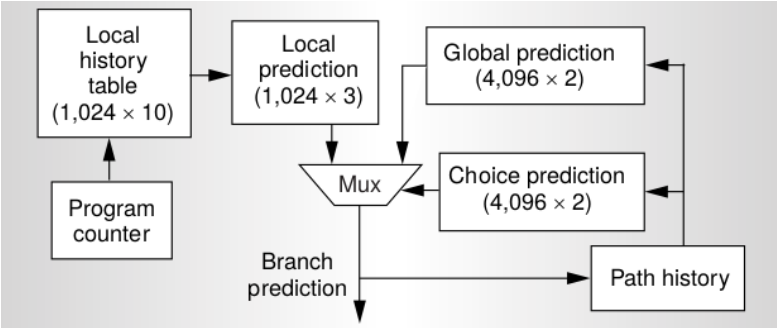


图 2.2 21264 锦标赛分支预测框图。(Kessler, 1999) 图 4.

Figure 2.2 Block diagram of the 21264 tournament branch predictor. Figure 4 of (Kessler, 1999).

(b) **重命名与乱序发射**。在 21264 的设计中，乱序的部分为了效率考虑，采用主频更高的时钟域。

一个周期最多能够取回 4 条指令，先锁存一个周期，然后在 CAM 形式的重命名表中进行重命名和寄存器的分配。需要注意的是，和 MIPS 一样，Alpha 在重命名阶段要特殊处理条件移动指令的映射关系。重命名完毕消除了写后写和读后写的冲突，但是依旧保留了写后读冲突。之后将指令写入发射队列中。发射队列采用分离式，分为整数指令队列和浮点指令队列，最多可以动态发射出 6 条指令，四条整数指令，两条浮点指令。使用记分牌来判断指令的操作数是否准备就绪。发射的细节上，微结构上有一个 20 项的定点队列和一个 15 项的浮点队列，队列只发射的是那些操作数都已经准备好的指令。与此同时，队列由仲裁器

来决定填入新的指令。上述模块的逻辑可以由图2.3直观的描述。上图中有一个

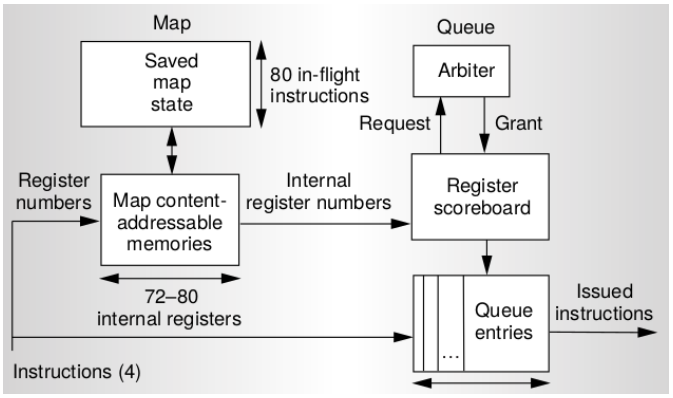


图 2.3 21264 寄存器重命名以及出入队列阶段框图 (Kessler, 1999) 图 5.

Figure 2.3 Block diagram of the 21264’s map (register rename) and queue stages. The map stage renames programmer-visible register numbers to internal register numbers. The queue stage stores instructions until they are ready to issue. These structures are duplicated for integer and floating- point execution. Figure 4 of (Kessler, 1999).

Saved map state 模块，非常重要，它的作用是在转移预测错误时恢复处理器的状态。注意该表有 80 项，也即每一条指令分配一项，这样处理器可以从任何一条指令之后精确地恢复状态，而不会受到跳转指令数量的约束。但是缺点是非常消耗资源。当然不光是转移预测错误的恢复，例外中断的状态恢复同样也是用这个表的，但和分支预测错误的恢复机制略有不同。

(c) **乱序执行引擎**。由于发射队列每一周期能够发射 6 条指令，所以引擎一共有 6 条执行流水线。

21264 具有特色地将整数寄存器堆分裂为两个集群 (cluster)，均有重复的 80 项。对应地，属于整数的 4 条流水线被均等的分配到了两个集群下。虽然这样增加了集群之间互相广播的电路延迟和周期数，但是却使得设计更加简单快速。最根本的原因是减少了寄存器堆的读端口数目。如果不分为两个集群，每条指令需要两个操作数一共就需要 8 个读端口，加上寄存器堆有 80 项，物理布局布线之后的时序非常的差，远远不如只有 4 个读端口的情况。Alpha 这个用空间换时间的做法也是无奈之举。这也给后来的设计者一种警示，为了处理器的主频，必须要控制寄存器堆的读端口数量。

(d) **指令的提交与退出**。指令是按照次序提交退出的。

在 21264 设计文档中给出最为有用的信息是每一条指令都要带着之前旧的目的物理寄存器号，并在顺序提交退出后回收这个该物理寄存器。首先这个机制能够维护着源源不断的物理寄存器可以再生分配，其次说明了这个机制有正确

性的保障。所以毕业设计中的做法和 Alpha 的做法保持一致。

(e) **内部访存部件设计**。

为了降低访存的平均延迟，Alpha 21264 在访存上做了很多不惜成本的设计，将性能做得非常强悍。主要体现在：一个周期能够执行两条乱序的访存请求，也即首先 dcache 即必须是双端口的；访存系统能够同时追踪 32 条 in-flight 的 store 指令，32 条 in-flight 的 load 指令和 8 条 in-flight 的指令或者数据 cache miss；dcache 是 64KB，2 路组相连的结构 (Kessler, 1999)。处理器内部访存控制结构采用经典的 load queue(LDQ) 和 store queue(STQ) 数据结构。每个队列均有 32 项，这和上述可以同时追踪的 load/store 指令数量一致 (Kessler, 1999)。毕业设计中基本的设计参考了 21264 的设计，但是做了一些面积功耗的优化与改进。

(f) **内部 memory 系统**。21264 设计了高带宽和低延迟的 memory 系统。

设计文档Kessler (1999) 最后花了较多的篇幅来介绍，包括 cache 预取，填入和替换策略、总线的介绍，以及用 8 项的 miss address file (MAF) 来同时追踪上文提到的 8 条 in-flght cache miss 访存请求的机制。内部 memory 系统是非常复杂的一个领域，在毕业设计中出于简化的考虑不会继续做深入的分析和设计。

2.2 MIPS R10000

R10000 是一款四发射乱序处理器，执行引擎有五条流水线；为了隐藏访存延迟，R10000 用了两级均为两路组相连写回式 cache，并且是非阻塞式的，也即一个 cache 行的 miss 不会阻塞另外 cache 行的访问；在处理器内部 in-flight 的指令数最多有 32 条 (Yeager, 1996)。处理器的整体设计参见模块框图2.4a，流水线 (图2.4b) 级数的编号是从 1 开始的。

(1) 第一级发出取指请求，并动态对齐下一周期的四条指令。

(2) 第二级对取回的指令做译码然后进行重命名，同时计算跳转分支指令的目的地址反馈到取指单元.

(3) 第三级将重命名完的指令写入发射队列中，直到操作数都已经准备好，才能从队列中发射出来。在后半个周期的时候读寄存器堆获取源操作数。

(4) 第四级开始执行，整数指令需要一个周期，load 指令需要两个周期，浮点指令需要三个周期，

(5) 写回是在得到结果后一个周期的前半个周期。

与 Alpha 21264 相似之处在于对整数指令和浮点指令采用了分离式的处理方案。但是与 Alpha 21264 相比，性能却差了很多。这与微结构的设计是分不开的。

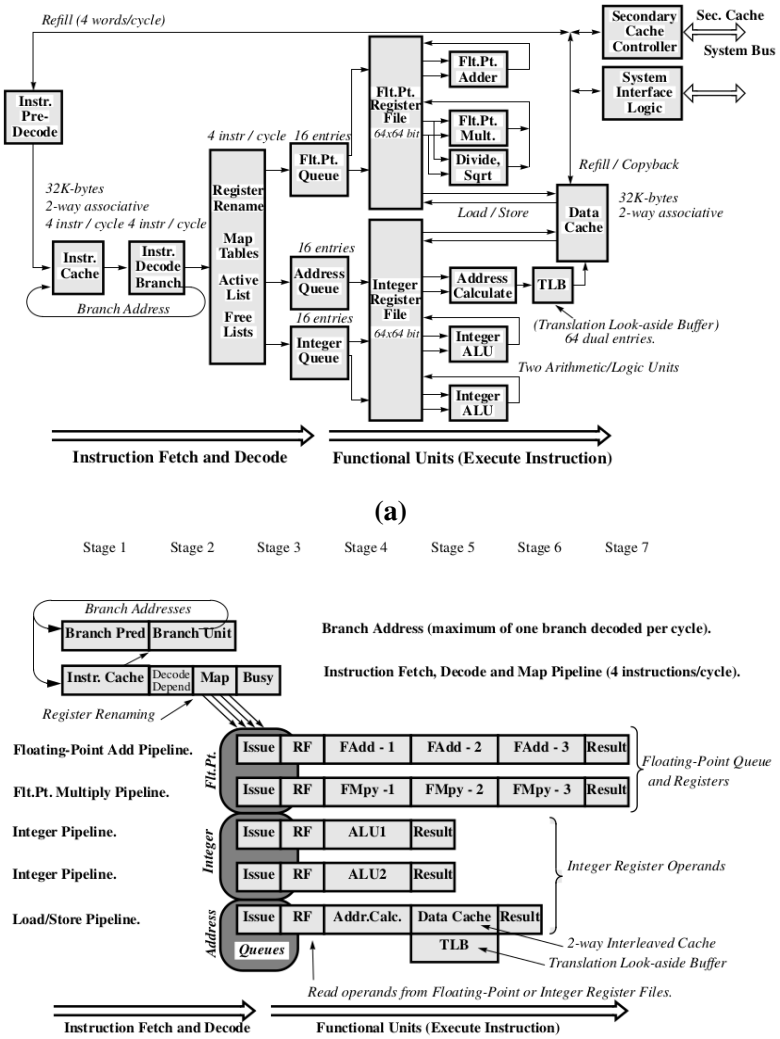


图 2.4 MIPS R10000 整体设计。(Ahi et al., 1995) 图 2 和 3. (a) R10000 模块框图，(b) R1000 流水线。

Figure 2.4 MIPS R10000 overall design. Figure 2 and 3 of (Ahi et al., 1995) (a) R10000 Block Diagram, (b) R10000 Pipelines.

仅仅是通过观察两者的整体设计图以及一些最基本的参数，不考虑转移预测率和访存的性能，都不难找到 R10000 与 21264 之间差距的几点原因：

(a) 因为都是采用分离式结构，所以只需对比整数指令。R10000 最多支持 32 条 in-flight 的指令乱序调度，但是 21264 却支持 80 条，是 R10000 的两倍多，所以对指令的并行性有更大程度的挖掘。还有一点是，由于两者超标量宽度同样是 4，但是由于 R10000 允许的 in-flight 指令比 21264 少太多，很容易导致发射队列塞满而阻塞取指单元供应指令。

(b) 流水级的切分不合理，导致时序太差。如果用统一的从 1 开始编号，那么意味着 21264 是在第 5 个周期读寄存器堆的，而 R10000 在第 3 个周期就已经读取了，足足提前了两个周期。换言之，R10000 将 21264 五个周期的工作压缩到 3 个周期，电路时序之差就可想而知了。有两点特别突出：

i 第二级中，从 icache 同步读回的指令，组合逻辑就要做译码，重命名，算跳转地址，所以第二级的电路延迟非常大。

ii 第三级中，要先从 16 项的定点队列中选出至多两条指令发射，然后再当周期去读寄存器堆的逻辑时序也是非常紧张。

这两点 Alpha 21264 都是分别用两个周期来做的，所以 R10000 少的两周期就是这么来的。

(c) 在 21264 中已经面对的问题 — 对于寄存器堆的读端口要严格控制。为此 21264 还专门做了两份重复的寄存器堆来缩减读端口。如果看配置，21264 是 80 项，有 4 个读端口；而 MIPS 则是 64 项，有 6 个读端口。因为读端口多了两个，所以单就寄存器堆的时序上，R10000 就不如 21264，更何况 21264 是用一整级来做读寄存器堆的逻辑的。

综上所述，在 21264 主频已经达到 500~600MHz 的时候，R10000 主频只能做到 200MHz；所以在 SPEC95 性能测试程序上，21264 定点程序在 30 分以上，浮点程序在 58 分以上；而 R10000 定点程序峰值为 9 分，浮点程序的峰值为 19 分，不足 21264 性能的 1/3(Kessler, 1999; Yeager, 1996)。但这些并不影响 R10000 同样存在一些值得借鉴的设计思路，梳理 R10000 中的设计亮点如下：

(a) 取指部件。

Yeager (1996) 提出的设计思路是处理器取回来的指令带宽上要高于执行部件，将队列填充满非常重要，这样基本上就能在每一个周期都找到可以被发射的指令。和一般取指需要取指宽度对齐不同，R10000 虽然取指宽度是 4 字 (1 指令/字)，但是可以在 16 字指令 cache 行中以任意字对齐，这样可以提高取值的效

率。通过一个对 cache 读出放大器简单的修改，基本上是四选一的逻辑来做到上述动态的对齐。同时，为了简化连续的取指的时序，当发射队列或者活跃表已经满时，指令会被先进入一个八字的指令缓存 (Yeager, 1996)，也即最多可以缓存 8 条未被译码的指令。这一点已经借鉴到了自己的处理器设计当中，参见3.5章节。

(b) 分支跳转单元。

R10000 在这一方面做的也没有 21264 性能高，体现在两个方面。第一，预测策略上，只有一个简单的有 512 项的两位饱和计数器，在 Spec92 整点程序中预测率也只有 87%(Yeager, 1996)；第二，设有分支栈，每当有译码到分支指令时，就会把处理器的状态压入栈中，作用是在分支预测错误时，处理器能够恢复到错误跳转前的状态。对比 21264 为每一个指令都存储对应的状态的做法，R10000 显然会受限于高分支指令占比的程序以及程序片段。为了快速恢复，撤销掉分支预测错误后面的指令，R10000 的每一条指令都带有 4-bit 的掩码，对应于分支栈，指示依赖于哪一项分支跳转。如果依赖该条分支指令，该位掩码置上。当分支预测错误时，可以根据掩码的信息来撤销误取的指令。

(c) 寄存器重命名。

见图2.5，不同于 21264 的 CAM 重命名表，R10000 的重命名表是 RAM 的形式。考虑到 HILO 寄存器的存在，并且排除永远为 0 的 r0 的寄存器，整数指令的重命名表是一张 33×6 bits 的 16 读端口 4 写端口的 RAM(Yeager, 1996)。在重命名的相关控制逻辑上，R10000 使用了三个数据结构 — Free list, Active list 和 Busy-bit tables. 简言之，这三个数据结构的作用分别是：Free list 负责记录更新未被使用的寄存器号用来进行物理寄存器的分配；Active list 作用相当于 Reorder buffer，记录在处理其中 in-flight 的活跃指令；Busy-bit tables 用来指示当前物理寄存器中的值是否有效。

(d) 发射队列。

除了常规的定点队列和浮点队列外，R10000 还增加了一个地址队列，也可以被称为访存指令队列。为循环队列的形式，有 16 项。但这也不是新奇的设计，功能上和 21264 中的 LDQ 和 STQ 设计类似，只是将 load 和 store 指令统一存放在一起而已。

(e) 其他方面如运算单元和内存层级结构的设计并不在毕业设计范围之内，不做探讨。

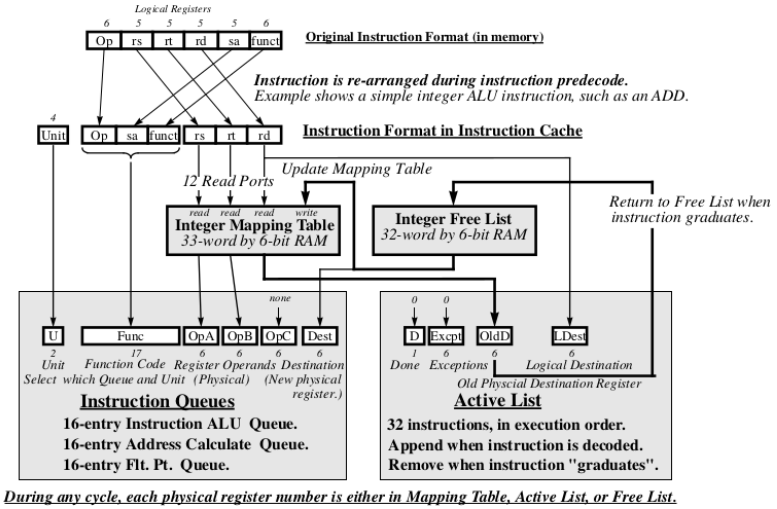


图 2.5 R10000 寄存器重命名机制图解。(Ahi et al., 1995) 图 4.

Figure 2.5 Register Renaming of MIPS R10000. Figure 4 of (Ahi et al., 1995).

2.3 RISC-V BOOM

BOOM 是伯克利分校为了向学术界和工业界推广 RISC-V 指令集而设计出来的乱序处理器，其全称为 The Berkeley Out-of-Order Machine。具体是由伯克利分校在校的博士生们参与设计，主要参考了 Alpha 21264 和 MIPS R10000 两款经典处理器架构。同时，BOOM 是由 Chisel 语言编写而成，不同于 21264、R10000 采用定制电路，BOOM 在诸如队列的项数，cache 的大小，甚至发射指令数量上都能参数化地设置。但是超标量宽度为 2 的设定在 BOOM 中是固定的，也即 BOOM 的取指单元每一周期最多能取回两条指令，且无法参数化设置。

BOOM 经历了先后两代的演化，从版本 1(BOOMv1) 到版本 2(BOOMv2) 的变化能够十分真切的看出其在微结构上的瓶颈、改进重点和设计权衡。

BOOMv1 在流水级的切分参考了 R10000，分成了六级 — 取指、译码/重命名、发射/读寄存器、执行、访存、写回 (Celio et al., 2017)。这种流水线的切分方案在 MIPS R10000 章节2.2已经分析过是极不合理的，电路主频做不高。而且 BOOMv1 的整数寄存器和浮点寄存器是统一的，这样导致的后果是寄存器的项数和读端口数量 (共有 7 个读端口和 3 个写端口) 都很多；另外，BOOMv1 采用了统一的发射队列，发射指令数量为 3，也即在同一个队列里要一个周期要选出 3 条准备就绪的指令发射 (Celio et al., 2017)。这样的设置同样极不合理，综合得到的电路异常复杂，时序会异常不好。综上，虽然 IPC 的比较上，BOOMv1 比 BOOMv 好了 20%(Celio et al., 2017)，但是 BOOMv1 微结构设计糟糕，参考意义不大。下面着重来分析 BOOMv2 的值得借鉴的微结构设计：

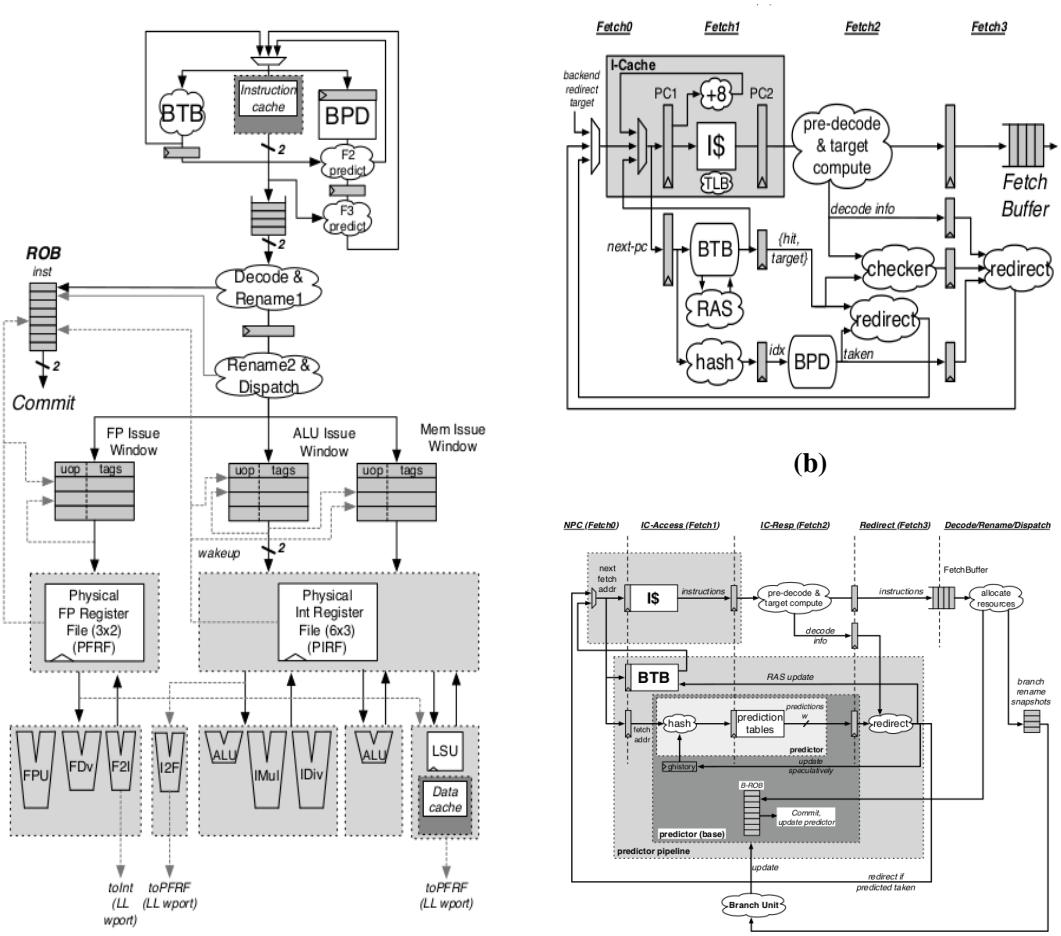


图 2.6 BOOMv2 的全局设计以及前端设计图。(Celio et al., 2017) 图 2(b) 和 3(b), Celio (2018) 图 4.3. (a)BOOMv2 的全局设计。(b) 前端设计图。(c) 前端细节设计图。

Figure 2.6 BOOMv2 overview and Frontend design. Figure 2(b) and 3(b) of (Celio et al., 2017), Figure 4.3 of Celio (2018). (a)BOOMv2 overview, (b)Frontend design, (c)Frontend design with more details.

(a) 前端的设计。

在 BOOM 相关的论文 (Celio et al., 2017; Celio, 2018) 中，强调了前后端的概念。事实上，这是一个非常优秀的理念，很值得借鉴到自主的处理器设计之中。前端负责项后端供应指令，连续不断的指令流的供应就显得尤为重要。为了预测率并且权衡面积、关键路径延迟和预测错误流水线的取消开销等多方面的考虑，在 BOOMv2 的前端中，加入了多种不同的转移预测技术。

跳转目标缓存 (BTB)，结构如图2.7，存储了一定数量的指令地址 (PCs) 到跳转目标的映射集合。处理器用 PC 当做索引，以 CAM 的方式进行查找，如果命中，则重定向指令流从跳转目标开始重新取指。对于分支指令，另外借助饱和计数器来进行跳或不跳的预测。为了面积和时序的优化，可以用比如 20-bit 的 PC 低位的部分来代替整个 32 位或者 64 位的 PC(Celio et al., 2017)，这样对预测率几乎没有任何影响。

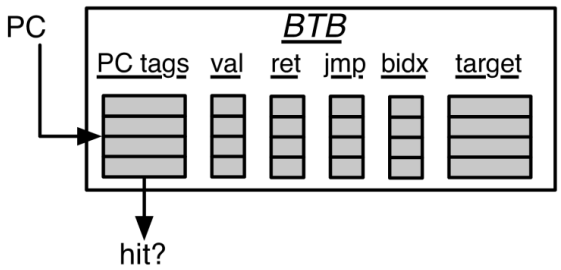


图 2.7 BTB 结构示意图。(Chris Celio, 2018) 图 3.2.

Figure 2.7 BTB Unit Structure. Figure 3.2 of (Chris Celio, 2018).

返回地址栈 (RAS)，用来预测函数调用的返回。寄存器类的跳转指令非常难以预测，因为寄存器的值是不固定的。函数调用后返回的地址虽然也是从寄存器读取的，但是却因为极具规律性 (等于调用该函数的指令的 PC 值加 4)，所以预测准确率很高。函数的调用行为可以用栈来完全的刻画，所以预测器只需在检测到函数调用的时候，把调用函数的指令 $PC + 4$ 就压入栈中，等检测到函数返回的时候再弹出栈，就能很好的预测对绝大多数函数调用的情况 (因为函数嵌套调用深度太深而导致返回地址栈溢出，会导致预测结果出错)。

分支预测器 (BPD)，专门针对分支指令的优化，每一项只存储两位饱和计数器来预测跳或者不跳的跳转方向，所以必须要在知道是跳转指令以及跳转指令结果的情况下才能使用。这样，BPD 可以和 BTB 一起并行查找，当 BTB 得到跳转地址以及是否命中的信息的时候，BPD 刚好能够得到跳转的方向；或者 BPD 等到指令取回开始译码并算得目标地址的时候给出跳转方向。由于每一项的位

数很小只有两位，所以项数可以做的很大，如 1024 项。这样就可以配合着 10-bit 的全局跳转历史通过哈希算法如 gshare 得到对于 BPD 的索引号索引 BPD 表得到跳转方向的预测，见图2.8。

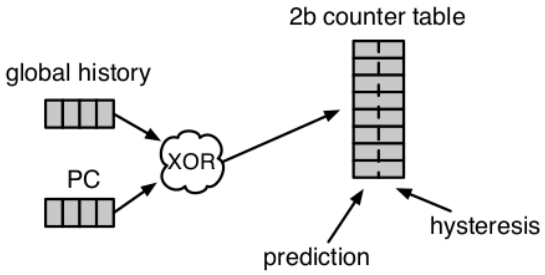


图 2.8 基于 gshare 的预测器，使用全局历史和取指地址哈希起来去索引两位饱和计数器表，高位代表对跳转方向的预测。(Celio, 2018) 图 4.4.

Figure 2.8 A gshare predictor uses the global history hashed with the fetch address (PC) to index into a table of 2-bit counters. The high-order bit makes the prediction. Figure 4.4 of (Celio, 2018).

见图2.6b，展示了前端各个预测单元的交互方式和流水线的组织形式。可以看到指令在 F2 阶段取回，被译码并计算得到目标地址，然后在 F3 阶段给出指令重定向的信息。这样能够给哈希算法整整一个周期的时间计算。BTB 的组织形式可以是多样的，比如参考 cache 的设计采用多路组相连的形式而不一定要采用全相连的形式，这样可以节约电路的时序。

还有一个模块值得注意的是在图2.6c中最灰方框中展现的名为 B-ROB 的单元，这一是个只存放分支跳转指令的小型的重排序缓存，里面保存了非常重要的处理器状态的快照 (Snapshot)，用来在分支预测错误时候对处理器的状态进行恢复。可以看出，这个设计参考的是 MIPS R10000。

(b) 重命名阶段的数据结构和操作都很大程度借鉴了 MIPS R10000 的做法，不做赘述。

(c) BOOMv2 的发射队列采用了和 MIPS R10000 几乎相同的组织形式，一个 16 项的定点指令队列，一个 16 项的浮点指令队列和一个 16 项的访存指令队列，见图2.6a。采用的是移位队列的形式 (collapsing queue) 并采用级联的优先编码器 (cascading priority encoder) 去选择最早就绪的指令发射出去 (Celio et al., 2017)。不过 BOOM 是可配置的，所以 BOOM 同样提供了另外一种 R10000 风格的无严格先后次序的发射，但这样会导致性能不佳。

(d) 访存单元。BOOM 设计了 3 个队列，the Load Address Queue (LAQ), the Store Address Queue (SAQ), and the Store Data Queue (SDQ)(Chris Celio, 2018). 在这

个设计上与 Alpha 21264 类似。如示意图2.9所示，load 和 store 的地址要做一个二选一，选择更早指令的地址去做访存。

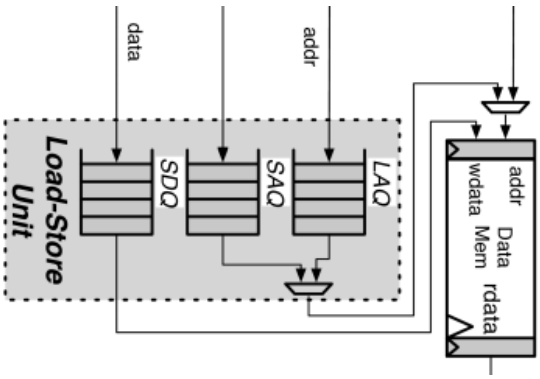


图 2.9 访存单元结构简化示意图。(Chris Celio, 2018) 图 8.1.

Figure 2.9 Load-Store Unit simplified diagram. Figure 8.1 of (Chris Celio, 2018).

第 3 章　处理器微结构设计

双发射乱序处理器的结构复杂，细节众多，不可能一蹴而就。仿照 Alpha 系列处理器从 21164 到 21464 逐代版本演进的成功案例，毕业设计中处理器也采用逐代演进的设计路线，从微结构最简单的单发射五级静态流水线 — CHIWEN 处理器 ⇒ 到次复杂的双发射五级静态流水线 — FUXI 处理器 ⇒ 到最复杂的双发射乱序流水线 — BIAN 处理器。

CHIWEN 和 FUXI 都是版本演进的中间产物，BIAN 才是毕业设计核心处理器，所以本章只集中阐述 BIAN 处理器微结构的设计思路和突出亮点。

3.1　处理器模型

这里的模型指的是一种比具体微结构更为抽象的解耦合形式 — 前端 (Frontend)、后端 (Backend) 模型。在章节2.3中分析 BOOM 时已经做过了初步的介绍。

(a) 前端: 提供稳定的指令流，具体来说就是从初始地址开始源源不断地取回指令提供给后端处理。

(b) 后端: 接受来自前端的指令，进行运算，最后写回寄存器堆或内存。

这个模型的优势在归纳起来：

(a) 前端不需要关心后端指令执行的机制；后端也不需要在乎前端取指状态机的运转流程以及所采用的缓存策略。例如对于可被缓存区域的取指，可以利用 cache 提高取指效率，也可以不用。

(b) 连接两端之间接口的信号非常少而且清晰。屈指可数 — 前端到后端方向有指令以及指令是否有效，指令所在的 PC，以及下一条 PC 的转移猜测信息；后端向前端有对于指令流的反馈如取回的指令是不是由于流水线的繁忙而需要阻塞，分支跳转类指令和例外中断对于指令流的重定向信息，以及对于转移预测数据结构的反馈更新。

(c) 前后端的解耦合模型，清晰的接口对于功能的调试、性能的调优也大有助益。

3.2　处理器高性能因素考量

衡量高性能的通用处理器有两个维度：频率 (时钟一周期经过的时间) 和 IPC(单位周期完成的指令数)。不同于 IPC 维度是硬指标，频率这个维度，不同

的硬件实现 (FPGA/ASIC)，同一逻辑的不同写法，甚至不同的综合工具都会对其产生较大的影响。一个比较客观的分析方法是计算从一个触发器的 Q 端到下一个触发器的 D 端之间最多的门级数 (在设计的时候靠着经验进行大致的估算和衡量，但是由于扇入扇出的影响，会存在误差)。因为现阶段刚刚完成了处理器的设计，还未来得及在综合工具上对电路延迟以及能够达到的主频做分析和磨合，所以目前对于高性能的评价主要依据 IPC 的高低。

要做高性能，基于前后端模型可以非常清晰的解耦合为做到高性能的前端和做到高性能的后端。

3.2.1 高性能的前端

供应指令要快。这个快又可以继续细分用三个维度来衡量：

(a) **能够取回来有效指令的周期占总周期的比重**。由于现代的处理器运算单元频率越来越快，存储器的频率就相对变慢。直接访问存储器要数十上百周期。优化的方法是引入高速缓存 (cache)，面积小频率快。

(b) **指令宽度**。每周期的指令条数，直观上来看，指令宽度越大，指令供应的越快。

(c) **指令的正确率**。例如在没有延迟槽设计的 ISA 下的较为简单的指令宽度为 1 的单发射五级静态流水线结构中，当跳转指令还没有运算出来跳转的方向和地址时，若阻塞前端会白白浪费周期数。所以一般会采用各种预测投机策略来续上指令流。对于结构越复杂，缓存指令数越多的后端，对指令正确率的要求也就越来越高。

对照这三个维度，同时考虑频率时序的优劣，就有很多权衡考虑。

首先来看第一个维度。cache 做多少大，多少路，要有多少个 cache 行，每个 cache 行要长度多少。cache 的大小首要考虑的是 ISA 和操作系统对于分页大小的规定，对于 RISC-V 来说是固定 4KB (参见章节1.2.1)。所以如果 cache 每路的容量大于 4KB，不是实地址低位索引就会有顶着色的问题。但是若采用实地址索引，对于 TLB 的逻辑将是一个极大的挑战。虚地址要先经过 TLB 转化为实地址再去索引 cache，若做成一个周期时序会紧张，若做成两级流水，由于跳转指令的存在，猜错的时候就会多浪费一个周期，效率反而会下降。一种比较简单的方案是 cache 单路的容量做成不超过 4KB，这样，cache 的索引号作为虚实地址的低位是一致的，换言之 TLB 转换和 cache 访问就可以达到并行，也即通过 TLB CAM 表查找得到实地址的高位锁存一拍再和与此时同步 cache 里读出的

tag 域做比较得到是否命中的信号。这样 cache 命中就只需要一个周期。但是代价是 cache 的容量太小，运行像 dhrystone(见表4.1) 这样指令范围大于 4KB 的程序，取指回来有效指令占比较低。出于增大容量的考虑，在毕业设计的处理器中准备采用四路的设计，每一路是 4KB 大小，这样总共就有 16KB 的容量。另外，cache 行的的长度为 64 字节，可以存放 16 条指令。

与 cache 不同，其他两个维度都与后端有着比较紧密的关联。对于指令宽度，如果前端是两条指令的宽度，后端做单发射流水线就不合适，指令供应的过快而后端消化不掉就会导致流水线阻塞，所以执行单元相应的至少需要两套。其次指令的宽度也不是越大越好，原因有如下几点：

(a) 宽度大，会增大前端的逻辑复杂度，同一周期的各个指令非常容易出现前后数据相关、控制相关的问题。

(b) 面向内存的接口宽度仍为一条指令，所以一旦出现 cache miss 就会有大量的空泡产生

(c) 对转移猜测器产生了巨大的压力，每一周期对宽度内的每一个 PC 都要做预测，再根据预测的结果，前面指令无效掉后面指令，整个过程电路的延迟很大。而且不光延迟大，取指的效率也不会呈现出正相关。

最后是指令的正确性。前文提到，后端的结构越复杂，级数越多，缓存的指令越多，如果还支持乱序，那么对于指令的正确性要求更高。一方面是级数越多，跳转指令从取指到执行的周期越长，如果猜错，损失的周期数就越多。换言之提高转移预测的准确性对乱序的提升会比顺序更显著。另外一方面，对转移猜测而言，结构越复杂的后端意味着前端可以有更多的周期进行分级的预测并矫正上游流水级的预测结果，使之更准确。例如乱序处理器一般在第三级做重命名以及分配物理寄存器而来不及执行，那么这个时候跳转结果依旧没有得到，还能够继续进行分支预测矫正。

3.2.2 高性能的后端

指令执行要快。这个 “快” 可以理解为尽可能少的阻塞前端指令的传送，也就是说不能因为少数的 “刺头” 指令拖累了整个处理器的指令通路，这里指的主要是访存类指令。同时通过目前已经设计出来的的单发射五级流水线 CHIWEN，双发射五级流水线 FUXI 处理器运行几个小程序发现，在 icache 都命中、没有访存延迟而且分支预测正确率在 97% 的情况下 (见图4.7)，CHIWEN 的 IPC 可以达到 0.99，FUXI 的 IPC 最多只有 1.33(见图4.6)。也就是说当前端已经全速取指且

指令正确率几乎为百分百的前提下，IPC 与预期的 2(与取指宽度成正比) 相差很大。分析原因有如下几点：

(a) 最大的原因首先是如果在同一周期的两条并行指令中第二条指令源操作数依赖于第一条指令的写回结果，就只能阻塞上游流水，串行执行。这种情况很常见。

(b) 访存指令的影响。其一，load-to-use 的周期数代价更加大了；其二，由于内存访存依然是单端口的，就算添加了 dcache，为了简单，也没有必要做成双端口。这样每一周期就只能允许一条指令去访存的，若遇到两条访存指令就必须拆成串行。

(c) 分支跳转类指令的影响。其一，前端存在两条并行指令只有第一条是有效的，第二条由于第一条被预测为跳转而被无效的情况。其二，两条跳转指令同时在执行级时，也要被拆成串行逐个计算跳转地址、比较预测信息，更新前端的预测数据结构并作出重定向前端指令流的抉择。

如此一来，顺序执行的后端在超标量的前端面前已经达到了一个瓶颈，取指宽度为 2 的取指器，运行程序的 IPC 最高的性能也不会超过 1.4。

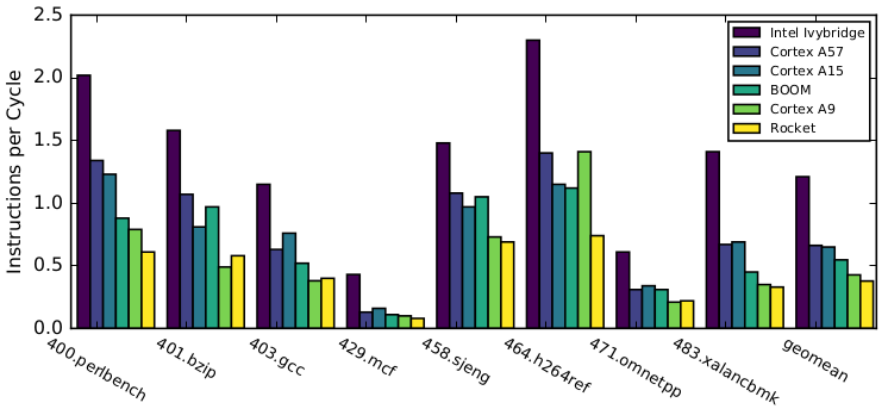
虽然 1.35 已经是个不错的成绩，但是，是基于 icache 全部命中，没有访存延迟，而且转移预测器命中率达到 97% 以上的前提下。在一般的情况下，可想而知连 1 都很难达到。对比图3.1中当前成熟处理器的性能，还有很大的性能提高空间。既然指令宽度为 2 的超标量顺序执行已经达到瓶颈，很自然地脑海中会浮现出三种方案：

(a) 继续提高取指宽度，比如四条指令的超标量，后端继续做顺序。

(b) 不提高取指宽度，因为对于指令宽度为 2，前端供应指令的最大 IPC 可以达到 1.8。计算方法为：每周期指令最多是 2 条，假设跳转指令占总指令的 20%，跳转指令在第一条位置的概率是 50%，跳转目标在第一条指令的概率是 50%，预测单元全部猜对，最后计算得到每周期正确的指令为 1.8 条。这与模拟得出的数据相符，见图4.13。从 1.33 到 1.8 还是有较大的提升空间，所以后端可以采用乱序微结构来做指令的动态调度。

(c) 既增加取指宽度，又把后端做成乱序。

权衡三个方案：第一个反而是最不切合实际的，换言之与高性能的目标背道而驰。宽度为 4 的取指器要在已有的取指器的基础上改成 4 条指令宽度，由前面的分析，电路逻辑复杂，时序不好，对转移猜测极不友好。其次，从 FUXI 的 IPC 瓶颈的分析中，可以看出，真正的瓶颈不在于前端而在于顺序执行的后端。



(a)

benchmark	Intel Ivybridge	Cortex-A57	Cortex-A15	BOOM	Cortex-A9	Rocket
400.perlbench	2.02	1.34	1.23	0.88	0.79	0.61
401.bzip	1.58	1.07	0.81	0.97	0.49	0.58
403.gcc	1.15	0.63	0.76	0.52	0.38	0.40
429.mcf	0.43	0.13	0.16	0.11	0.10	0.08
458.sjeng	1.48	1.08	0.97	1.05	0.73	0.69
464.h264ref	2.30	1.40	1.15	1.12	1.41	0.74
471.omnetpp	0.61	0.31	0.34	0.31	0.21	0.22
483.xalancbmk	1.41	0.67	0.69	0.45	0.35	0.33
geomean	1.21	0.66	0.65	0.55	0.43	0.38

(b)

图 3.1 SPECint2006 IPC 不同 IPC 比较。(a) 图。(Celio, 2018) 图 3.7. (b) 表。(Celio, 2018) 表 3.6.

Figure 3.1 Instruction-per-cycle comparison running SPECint2006. (a) The figure of instruction-per-cycle comparison running SPECint2006, Figure 3.7 of (Celio, 2018) (b) The table of instruction-per-cycle comparison running SPECint2006. Table 3.6 of (Celio, 2018)

那么第三个方案呢，也不切合实际，主要是因为目前，是从来没有设计过乱序处理器的新人，水平有限。比较之下，第二个方案是最合适的。

综上分析，最后毕业设计的处理器核心具体的结构就是取指宽度为 2 的前端匹配上乱序执行结构的后端。

3.3 中间层的引入

事实上，处理器前后端的设计思想最初应该是借鉴于编译原理的。编译原理作为计算机领域一门成熟学科，就是通过前后端的设计方法学来降低编译器设计的复杂度。类比于只采用前后端模型的简单编译器，对于简单的顺序执行的五级流水，前后端的划分也是足够的。但是为了做代码优化，编译原理专门提出了中间表示层的概念，依托清晰的结构化设计，再一次的有效地控制了问题的复杂度。而这一思路同样可以借鉴在结构上比顺序更复杂的乱序处理器的设计上。事实上，在乱序的处理器中，要从顺序的前端变化到乱序后端，最后再从乱序的执行变回顺序的提交，恰好需要有这么一个像中间桥梁一样的中间层的存在。

3.4 处理器的状态

章节3.3中提到了引入中间层的必要性。而如果用一句话来概括中间层的作用，那就是管理处理器状态的变化。那么什么是处理器的状态。指令和访存的数据是外部传进来的激励，算不上是处理器的内部状态。所以只有 PC 值，32 项数据寄存器值能算是处理器状态。站在更高的视角来看，所有处理器的物理数据结构，外来的指令数据最后修改的都只有这两类寄存器。而套用前后端的模型，前端管理 PC，后端管理 32 项数据寄存器，中间层则是管理这些状态的变化。

3.5 指令乱序调度和执行级别

论及乱序指令调度，这样的机制虽然做到后面准备好的指令可以先于前面没有准备好的指令执行，但也不意味着要做到真正意义上的毫无差别调度最先准备好操作数的指令，哪怕这条指令距离最早未提交指令的 64 条开外。为了支持在这个例子里所说的远在 64 条开外的指令的乱序调度，所付出的硬件的代价对于效率的收益值不值得，是值得商榷的。但它一定不是一个好的设计思路，因为它没有体现出一种被体系结构所强调的思想 — 层次化。就拿最为经典的存储结构来说，从处理器内部的寄存器堆，到一级高速缓存，二级高速缓存，共享缓存再到内存，硬盘，磁盘，好的设计是把越重要越常用的数据放在越快速的存储

设备中的，而不会是因为寄存器堆很快速就去把寄存器堆做达到百兆的容量。换言之分清主次的层次化才是真正好的设计，处理器的设计亦是如此。所以，在毕业设计中特别强调：对指令调度乱序执行要有等级化、层次化的划分。

回想在顺序流水的设计中，其实同样有指令的执行等级划分，只不过做的很极端而同样不合理 — 处在流水线越下游的指令执行级别越高，而且这种级别精确到每一条指令，只要下游早的指令没有执行完或者没有准备好操作数，上游的所有指令都不能执行。这就是顺序执行自带的而且无法改变的执行等级划分的通病。如果以这种角度来思考，乱序能带来了什么？带来的是可以打破这种僵硬的等级划分通病，带来的是可以自行定义执行等级的自由。当然这种自由是靠更多的硬件资源交换得来的，来之不易就不能随意挥霍。需要对执行等级的自定义深思熟虑，做到符合指令执行的规律，这样才会高效。

大方向的规律非常清晰，就是越早被取回的指令，执行等级越高，被优先执行的倾向越大，反之执行的等级越低。但是又不能像顺序一样走向极端。所以等级一定是按照指令新旧的梯队来划分的，如第一梯队，第二梯队等等，然后每一梯队里面包括了不止一条的若干条指令集合。就以 BIAN 处理器中具体体现出层级化的设计点举例来说，BIAN 微结构上将指令的执行等级分为 3 级：

(a) **第一梯队的指令集合**：容量 $2+2+2+4=10$ 条，参见图3.2。其中两条位于从重命名阶段到执行阶段的锁存器中，其他 8 条分别位于 3 个执行队列中。每条指令都存储住两个操作数，不需要再次读寄存器堆，侦听到写回总写结果立马可以在每个队列中选出最早的一条指令执行，分配有一个通用的 ALU，和一个写回端口。相当于是两个输入三个输出的执行调度站，详细的调度策略参见章节3.9.1。

(b) **第二梯队的指令集合**：容量 $9\times 2=18$ 条，是两个容量都为 $8+1$ 条指令的并行的发射队列，其中 8 条位于 8 项移位队列中，1 条头部寄存器中，这样的设计时序会比直接 9 项的移位队列更好。队列中的每一项通过侦听前递信息判断操作数准备与否。位于队列中的指令会先移到头部；位于头部指令会和旁路过来的流水线二三级之间锁存器中的指令做二选一的逻辑，读寄存器堆获得源操作数，进入执行阶段。

(c) **第三梯队的指令集合**：容量 $16\times 2=32$ 条，存储结构为循环队列，每一项可容纳两条指令。指令顺序出队列进入重命名阶段。容量最大，执行级别最低。

(d) 还有另外一类指令不好纳入这三个梯队中，由于功能特殊，而且是对外

的操作，需要单独占有一个写回端口，这就是访存指令。这类指令对于处理器性能的影响很大，所以需要着重优化，我的设计中采用的是非常经典的 LDQ 和 STQ 分别存放 load 和 store 类指令的信号，参见图3.3，目前的参数分别都是 6 项。同时为了提高访存密集型程序的性能，另外专门设计了一个可以同时存放 load，store 指令的 load/store 队列，里面只存放了很少量的关于访存指令的信息。作为比 LDQ 和 STQ 执行等级更低的存储单元以避免因为 LDQ 和 STQ 容量太小而导致频繁阻塞前端指令供应的情况。具体的设计考虑参见章节3.8.7；具体的运作机制参见章节3.9.3。

如果保守估计在 LDQ 和 STQ 中共分担 4 条访存指令，那么乱序执行的指令总数可达 $10 + 18 + 4 = 32$ 条，在乱序执行的上游还可以最多有 32 条的顺序的指令缓存 (第三梯队)。所以处理器中可以最多容纳 64 条指令，两条并行的发射队列 (第二梯队)，加上 3 条并行的执行队列 (第一梯队)。另外，一共有 3 个 ALU 运算部件；寄存器堆有 4 个读端口，4 个写端口，读写端口较少，特别是读端口数量是双发射架构中最少的。整个处理器架构如图3.3，设计合理。

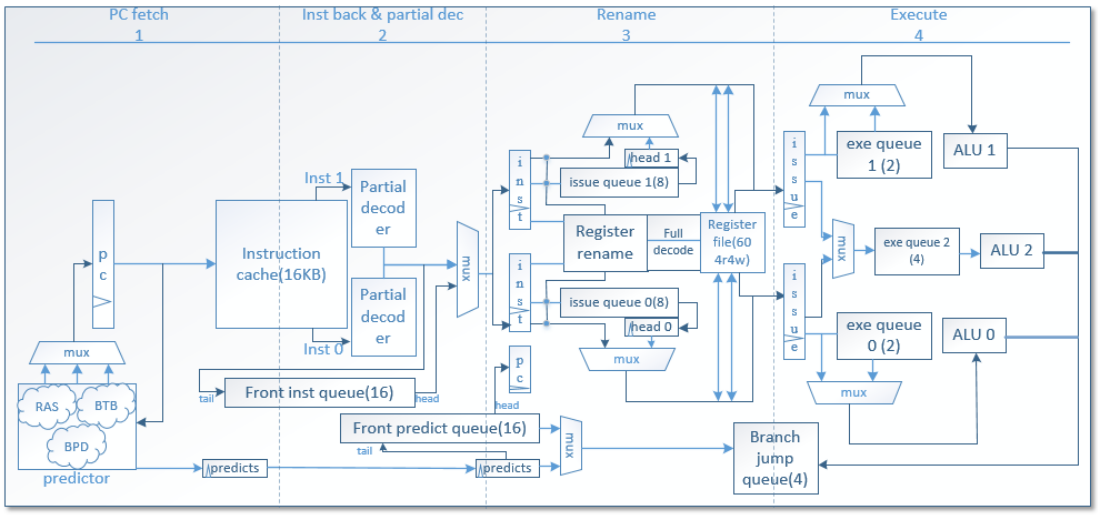


图 3.2 BIAN 处理器整体框图，不包括访存单元

Figure 3.2 Block diagram of BIAN processor (not include load-store unit.)

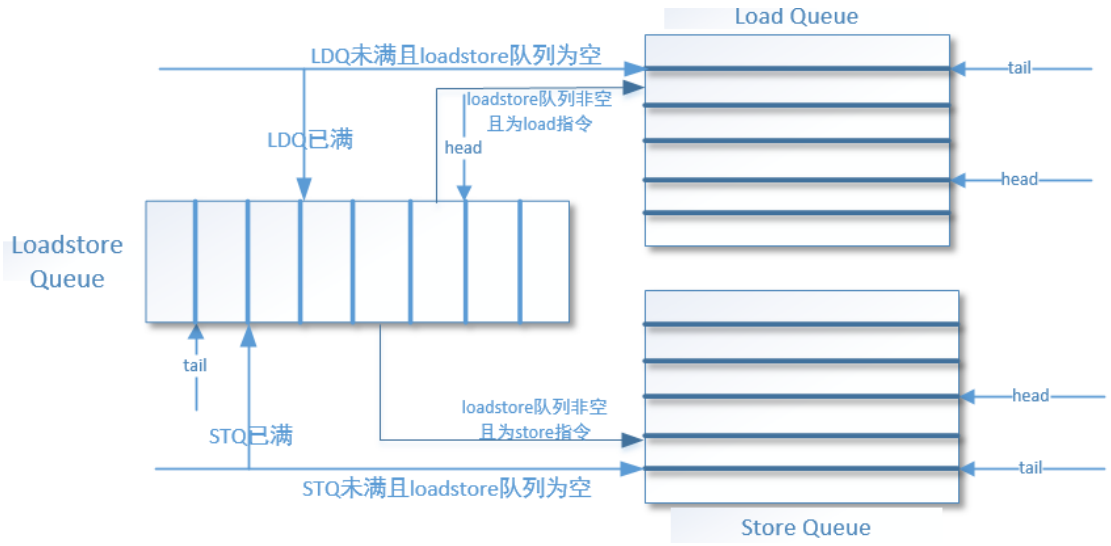


图 3.3 BIAN 处理器访存单元

Figure 3.3 Block diagram of load-store unit in BIAN processor.

3.6 流水线阶段划分

处理器流水级的划分如图3.4：

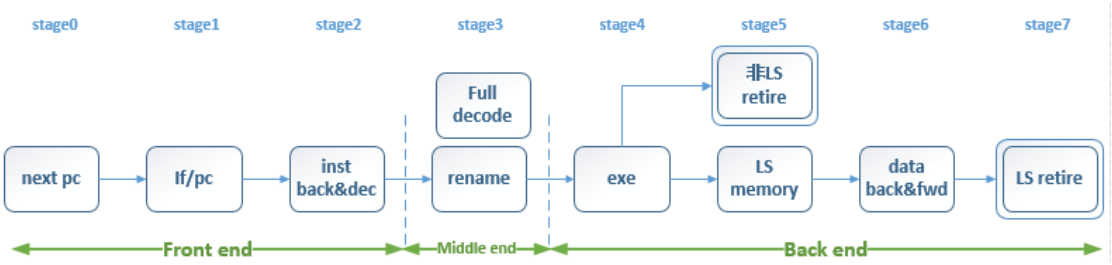


图 3.4 BIAN 处理器的流水线示意图。

Figure 3.4 the pipeline of BIAN processor.

(a) 第 0 级 — **next_pc 级**

界定有没有这一级的标准是对于后端得到的跳转地址以及跳转使能信号，是直接送到 pc 级 (稍后要介绍的第 1 级) 向内存或 icache 发出 PC，还是先送到锁存器锁存一个周期，再送到 pc 级。概括起来前一种做法是组合逻辑，后一种做法是时序逻辑。组合逻辑没有第 0 级，优势在于如果转移猜测错误，浪费的周期数会比时序逻辑要少一拍，最为极端的例子是在带有延迟槽的 ISA 如 MIPS，最简单的单发射五级静态流水采用组合逻辑的方案从而不需要转移猜测。但是组合逻辑的劣势也是很明显的，就是时序不好，从后端计算出来直接跳转目标和有 PC 以及各级转移猜测的目标地址做多选，最后得到发出的 PC 值。pc 级电路延迟很大，导致一方面访问 icache 的时序紧张，另外一方面如果要做转移猜测，那

么 PC 值还要连到转移预测单元中得到下一周期 PC 值。这样就会把整个 pc 级撑得很大，对整个处理器的频率做高不利。所以本质上来讲，next_pc 级是为了缓解 pc 级的压力而增加的一级。

(b) 第 1 级 — **if/pc 级、取指级**

发出 PC 从内存或者 icache 中取指，和经典的五级流水线保持一致。

(c) 第 2 级 — **inst back & dec 级，译码级**

这一级的名称兼容于经典的五级流水，原来的含义是指令在这一级被取回然后译码得到处理器内部操作微码。但是在 BIAN 处理器中，这一级只需做部分译码，得到一些简单的如是否是跳转指令的信息，从而减小这一级的电路延迟。再者，BIAN 不会在接下来的第 3 级立刻执行指令，全译码放到第二级的需求不大。

(d) 第 3 级 — **rename 级、重命名级**

这一级是 BIAN 处理器中至关重要的一级，承接着前端与后端，管理着后端的各种资源。从前端的角度来看，这一级还可以进行更加复杂策略的转移猜测；从后端的角度来看，这一级完成从逻辑寄存器到物理寄存器的转换和指令所需物理资源的分配。这里的物理资源有物理寄存器、内部指令标识符 id 号、也即 ROB 分配的 id 号、访存队列、分支跳转队列、发射队列。读同步寄存器也在重命名级进行。

(e) 第 4 级 — **execute 级、执行级**

在执行队列或者旁路过来的发射指令，选出 3 条准备就绪的指令在 3 个 ALU 中运算执行，如果是单周期 ALU 指令，写回寄存器堆；若是分支跳转指令发送到分支跳转单元；若是访存指令送到访存单元中。

(f) 第 5 级 — **非 LS retire 级/LS memory 级**

乱序处理器流水级的划分从这级开始分化。单周期的 ALU 指令已经是 retire 级了，因为所需操作已经做完，在 ROB 中进行相应的提交操作就可以退出。但是对于访存类的指令，这一级是 memory 级，进行的操作是向内存发出访存地址和 load 请求。

(g) 第 6 级 — **data back & forward 级**

load 所需的数据将在这一级被取回，同时做 forward 操作，将位于该 load 指令之前所有未写回内存，且与该 load 指令地址冲突的 store 指令的数据前递到 load 的数据中。成功加载到内存数据便可写回寄存器堆。

(h) 第 7 级 — **LS retire 级**

对于 load 指令，在 ROB 中提交就可退出；对于 store 指令，当 ROB 队列的头部 id 号等于 STQ 的头部 store 指令 id 号时，将数据写回内存，并且移动 ROB 队列的头指针。退出处理器。这一过程中还会做 backward 操作 — 检测位于该 store 指令之后是否有地址有冲突同时尚未做前递便已经将数据加载写回的 load 指令，若有，前端会以这条 load 指令的 PC 值为起点重新开始取指，做回滚操作。

访存指令从第 0 级到第 7 级完成提交退出，一共要经历 8 个周期，这是 BIAN 处理器指令流的最长路径。从前后端的模型来看，第 0 级到第 2 级的前 3 级流水属于前端 (front end)，第 4 级到第 7 级的后四级流水属于后端 (back end)，第 3 级单独成为一层 — **中间层** (见章节3.3)。

3.7 前端的设计

前端包括取指单元，转移预测单元和一级高速缓存。下面从这三个单元来剖析:

3.7.1 取指单元

从最简单的宽度为 1 的取指单元开始设计，代码如下:

```
/*一共有3个状态:
* sWtAddrOK状态下等待地址握手成功
* sWtInstOK状态下等待指令取回
* sWtForward状态下等待后端接收取回的指令
*/
val sWtAddrOK :: sWtInstOK :: sWtForward :: Nil = Enum(3)
val state = RegInit(sWtAddrOK) //取指单元的初始状态
switch (state) {
  is (sWtAddrOK) { //在sWtAddrOK状态下
    //如果地址握手成功，切换到sWtInstOK状态
    when(addr_ready) { state := sWtInstOK }
  }
  is (sWtInstOK) { //在sWtInstOK状态下
    //若外界发送的指令有效，意味着指令被顺利取回
    when(inst.valid) {
      when(io.forward || inst_kill) {
//被接收或者被取消，判断地址是否握手成功进入相应的状态
        state := Mux(addr_ready, sWtInstOK, sWtAddrOK)
      }.elsewhen(io.dec_kill) {
//如果被流水线下游取消掉取回来的指令，切换到sWtAddrOK重新开始地址握手
        state := sWtAddrOK
      }.otherwise { //如果后端暂时无法接收，切换到sWtForward状态
```

```
          state := sWtForward
        }
      }
    }
    is (sWtForward) { //在sWtForward状态下
      when(io.forward) { //被接收，判断地址是否握手成功进入相应的状态
        state := Mux(addr_ready, sWtInstOK, sWtAddrOK)
      }.elsewhen(io.dec_kill) {
    //如果被流水线下游取消掉取回来的指令，切换到sWtAddrOK重新开始地址握手
        state := sWtAddrOK
      }
  } }
```

上面的代码就是描述整个取指器行为的状态机，一共只有 3 个状态，非常的简洁清晰，可以用图3.5来描述：

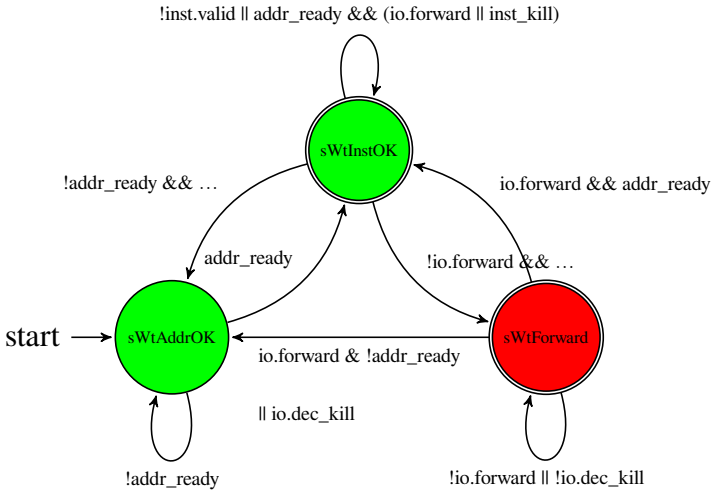


图 3.5 取指单元的有限状态机。

Figure 3.5 the FSM of FetchInst Unit.

除了简洁的状态机模型，取指单元还做了两点设计：

(a) 在电路频率的考虑上，将所有从后端和内存传来的信号，都用锁存器锁存一个周期再去改变取指单元的状态，从而保证了取指单元的时序和外界独立。

(b) 为了配合图3.5中的三个状态的状态机，取指的逻辑上有一个最为基本的限定 — 取指的 in-flight 指令数最多为 1。取指单元和 icache 都要相应地做成阻塞式，也即要等到正在访问内存或者 icache 的 PC 对应的指令被取回并被后端接收，才能够发出下一个 PC 取指请求。这种只能跟踪一条 PC 取指的方式，严格保证了取指的顺序性，同时取指单元的效率又不会有明显的损失。

接下来是将取指单元的宽度从 1 提高为 2，实现超标量。可是这个跨越并不简单。首先必须摆脱的误区是，由于内存的端口宽度是限定为 1 条指令 32 位的，所以如果直接访问内存，那么一个周期内取回的指令宽度仍然是 1 而不是 2。故 icache miss 后 burst 传输是逐条指令传输回来的。宽度为 2 的情况只会出现在 icache 命中的情况下。这样，超标量的取指单元为了保证效率，显然不能等到可以两条指令同时从 cache 取回来的时机。Burst 传输和 uncacheable 下直接访问内存的情况都要做到取回一条指令就立即发出一条指令。这样也可以理解为能够支持单宽度和双宽度之间的来回切换。得益于上述 in-flight 指令数最多为 1 的基本限定，双宽度的取指单元仅仅需要在原有的单宽带的基础上稍加修改。相比单宽度，双宽度有以下几个需要多考虑的逻辑：

(a) 超标量的两条指令是 2 对齐的，第一条指令一定是偶数 PC，第二条一定是奇数 PC。如果 if/pc 级发出偶数的 pc，然而 dec 级由于 cache miss 再通过 burst 传输先传回了第一条偶数的指令，这时候可以不用顾及后端给出的 forward 接收信号，立马发出后一条的奇数 PC 取指。这个考虑使得无论再什么样的情况下，双宽度的取指效率都不会低于单宽度的取指效率。

(b) 指令流会有跳转，而且有一半的概率是在第一条指令的后面需要跳转 (由预测器给出)，这种情况在 BIAN 的设计中被称为 inst split，上述代码中用 `inst_split` 变量来表示，如果第二条指令也取回来了，要取消掉。

(c) 当出现 inst split 的情况，而且预测单元给出的目标地址是奇数 PC 时，取指单元可以同样不用顾及后端的 forward 接收信号，直接发出跳转目标地址去取指。这样能进一步地继续提高取指效率。但是会存在同一个周期里被后端接收的指令不是地址连续的情况，这一点需要注意。

(d) 后端的 forward 接收信号 (也即阻塞信号的取反) 变成了两位的向量，使得并行的两条指令被后端接收的逻辑变得独立。

呈现的状态机代码如下：

```
val sWtAddrOK :: sWtInstOK :: sWtForward :: Nil = Enum(3)
val state = RegInit(sWtAddrOK)
val inst_valid_orR: Bool = inst_valid.reduce(_||_)
/*if_forward变量添加，是双宽度和单宽度之间最主要的不同
* 表示的意义是当状态在sWtInstOK而且指令取回时，
* 能够继续发送PC取指请求的集中不同的条件
* 或表达式的最后一项的Mux多选逻辑参考
* 前面双宽度需要多考虑逻辑的(a)(b)(c)部分
*/
val if_forward: Bool = io.forward(1) || inst_kill || Mux(inst_split,
```

```
    io.pc(conf.pcLSB).toBool, !inst_valid(1))

  switch (state) {
    is (sWtAddrOK) {
      when (addr_ready) { state := sWtInstOK }
    }
    is (sWtInstOK) {
      when(inst_valid_orR) {
        when(if_forward) { // almost the only difference
          state := Mux(addr_ready, sWtInstOK, sWtAddrOK)
        }.elsewhen (io.dec_kill) { state := sWtAddrOK
        }.otherwise { state := sWtForward }
      }
    }
    is (sWtForward) {
      when(io.forward(1)) {
        state := Mux(addr_ready, sWtInstOK, sWtAddrOK)
      }.elsewhen (io.dec_kill) { state := sWtAddrOK }
    }
  }
```

3.7.2 缓存数据组织结构

数据和控制应该相互独立。这是为什么存储的数据组织结构和指令高速缓存 (控制) 分开论述的原因。高速缓存的存储可以有不同的数据组织形式 — 直接映射、全相连、多路组相连。同样，多路的时候可以有不同的替换算法。但这一切都与读写的逻辑无关，对外呈现的读写端口不变。只要求 cache 行的长度对外保持一致即可。将数据和控制逻辑解耦和，可以做到最大程度控制复杂度。另外一方面也可以做到逻辑的复用。比如指令缓存和数据缓存对于读写的控制逻辑不一样，但是它们的存储结构可以做到一致。虽然目前 BIAN 处理器仅是用一个 16KB 的直接映射组织结构来代替四路组向量每一路 4KB 的组织结构，但是由于数据和控制的分离，对数据结构的优化不需要涉及已有控制逻辑的修改。

3.7.3 指令高速缓存

Icache 状态机的代码如下：

```
val sLookUp :: sBurst :: sWriteBack :: Nil = Enum(3)
val state = RegInit(sLookUp)
switch (state) {
  is (sLookUp) {
```

```
      when (pc_miss) { state := sBurst
      }.otherwise { state := sLookUp
    }
    is (sBurst) {
      when (rlast && rvalid) { state := sWriteBack
      }.otherwise { state := sBurst }
    }
    is (sWriteBack) {
      when (pc_double_miss) { state := sBurst
      }.otherwise { state := sLookUp}
    }
  }
```

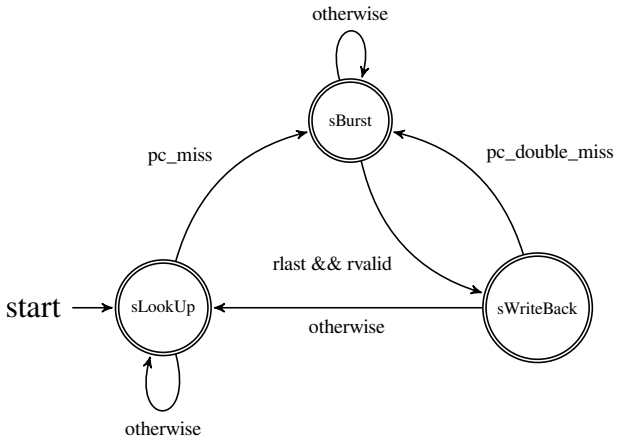


图 3.6 Icache 的有限状态机。

Figure 3.6 the FSM of Icache.

如图3.6，状态机一共有 3 个状态。sLookup 阶段，在 icache 中以 PC 的低位为索引查 PC 对应的指令是否命中，如果出现 miss 了就进入 sBurst 状态进行 burst 传输。burst 传输是 AXI 协议中的一种传输模式 (但也不局限于 AXI 协议，很多高新能的总线协议中都有类似的模式)，发出一个 PC，能够取回这个 PC 所在的一段连续指令。比方说 burst 的长度设置为 16，即使 cache miss，在一段访问内存的延迟后，也有 16 个周期是有指令可以提供的，这样就能够大大降低访存延迟的代价。

BIAN 的 icache 把 burst 传输长度设置为 16 字，64 字节，与 cache 行长度吻合。在 AXI 协议中 burst 有着两种基本的模式，一种是 INCR，一种是 SWAP。通俗来说 INCR 就是在以 PC 为起点，往后取一段指令，所以 INCR 发出的 PC 一般要指令长度对其。SWAP 同样是以 PC 为起点，但是 PC 不要求是指令宽度中

最前段的指令，可以是中间的某条指令，向后取指到达右边界后再循环到左边界直到取回来 PC 的前一条指令。icache 采用的是 INCR 模式，因为其一取指有连续的倾向，而且在很大的情况下导致 burst 传输的 pc 都是 cache 行的第一条，虽然 SWAP 模式一定不会比 INCR 差，但是 SWAP 模式其实和 INCR 效率相差不大；另外，考虑到取值时序紧张，INCR 模式对内对外都比 SWAP 模式友好。故最后采用的 INCR 模式。

回到状态机上，当状态是 sBurst 时，如果 rlast 和 rvalid 信号同时置上了，说明最后一条指令也被取回来了，状态就会跳到 sWriteback 状态。sWriteback 状态下就是把取回来暂时用锁存器缓存的整个 cache 行写回章节3.7.2的存储结构中。为了提高效率，在 burst 传输阶段也是同步可以接收取回来的指令的，这样就会出现可能一个跳转出现第二次的 PC 在 cache 中的 miss，这个时候整个取指单元就会因为 in-flight 取指数至多为 1 的基本限定而卡住动不了了。故在 sWriteback 状态下，如果检查到 PC 的二次 miss 就会直接进入 sBurst 传输阶段，从而提高效率。如果没有出现二次 miss，那么就会又跳回 sLookup 状态。整个 icache 的行为简洁而高效。

3.7.4 转移预测单元

除了高速缓存技术，转移猜测也是高性能处理器的关键。与简单的流水线后端一遇到阻塞就要暂停前端取指不同，高性能的处理器一定是要做到后端遇到阻塞如访存数十拍的延迟时，还能允许前端继续投机地取指。来不及处理的指令将会被缓存在后端。按照转移指令在程序中 15% 左右的占比，意味着指令流会在平均 6~8 条之后可能重定向一次，如果没有转移猜测，一味的增加 PC，等到前面的跳转指令被执行确认指令流需要重定向，那么后面的指令会被取消掉，体现不出指令缓存的作用，浪费存储面积。要真正发挥缓存指令的优势，就必须依赖转移猜测来提高缓存指令的正确性。转移预测可以分布在流水线的如下阶段:

(a) if/pc 级要得到下一周期取指 PC，此时还不知道 PC 对应的指令是不是跳转指令，唯一有用的信息是根据历史存储的从 PC 到目的地址的映射。但是 PC 有 32 位，靠 RAM 来索引不符合实际，剩下的方案只剩 CAM 表。代表性的数据结构是 BTB，参见章节2.3。CAM 表只能存少数的映射关系，比较理想的是 32~64 个。在 BIAN 处理器中实现了用 BHT 和 gshare 算法辅助预测分支跳转方向，64 项 BTB 映射表，能够存储 64 条分支跳转的目标地址。

(b) inst back 级指令已经取回，但是整个周期剩余不了多少时间，最多只能根据局部译码检测到是函数返回指令选择 RAS(参见2.3) 的栈顶地址重定向指令

流。而在目前的 BIAN 处理器中，并没有实现在这一级基于 RAS 的转移预测，而是先锁存了一个周期，在 rename 级做。

(c) rename 级中，对于简单的 jal/j 指令，可以直接由对应的 PC 和指令中的立即数域得到跳转地址并进行指令流重定向；对于 branch 类指令，虽然能够算出跳转地址，但尚不知道跳转方向。可以使用 BHT 和章节2.1中提到的基于 local 历史和 global 历史的锦标赛预测机制来预测方向。目前，在 BIAN 的实现中，这一级仅仅做了 jal/j 指令以及对未在 if/pc 级未做预测的 branch 类指令的跳转。正在准备添加这段逻辑来提高预测率和处理器整体的性能。

3.7.5 预测单元优化

在处理器的设计当中，有很多关于效率，时序，面积的权衡，这在预测单元的设计中得到了充分地体现。预测单元采用更加复杂的预测技术把预测正确率做到更高的同时，要付出更多的面积，更长的电路延迟和更大的功耗。在保证预测正确率的基础上，对于面积时序的优化也显得重要。下面突出在 BTB 结构中 BIAN 处理器所做的优化：

BTB 原理简单，但是却是转移猜测中最为重要的一个部件，直接分担了程序跳转 80% 左右的预测正确率。这个逻辑上简单的从 pc 到 target 的映射关系，值得好好的分析得到更好的物理实现，保证容量和正确率的同时降低面积和延迟的开销。

(1) 观察一：在映射关系中，无论是 PC 还是 target 的高位很少有变化，所以一个想法就是能不能把高位比如说高 16 位和低位比如低 16 位分开，做成两个表的映射。这样考虑到高位很少变动，高位的映射表项数就可以做的很小，比如 4 项 (低位还是 64 项)，如图3.7。计算一下，采用这种的设计方案，总共占用的面积是 $4\times16\times2+64\times16\times2+64\times2=2304=36\times32\times2$ bits。最后加的 64×2 是在低位映射表中存储高位映射表的索引号所需要的面积，这个相当于 36 项 32 位 PC 映射到 32 位 target 的直接映射表。当然，如果是 64 位的 ISA 下，这种优势会更加明显。那么电路延迟如何呢？面积一样的情况下，两张 CAM 表的延迟不会比一张 CAM 表差，反而会因为面积减小，电路的布局布线上有优势。

(2) 观察二：PC 到 target 的映射中高位基本上都是一致的，也就是说程序的行为大多数都是在小范围内相对跳转。针对这一特点，在低位的映射表中再加 1-bit 的标志位，标记映射的高位是不是相等，如果高位相等就不需要占用高位映射表项，大大降低了高位映射表的替换率，从而提高了优化一数据结构的预测稳定性，如图3.8。

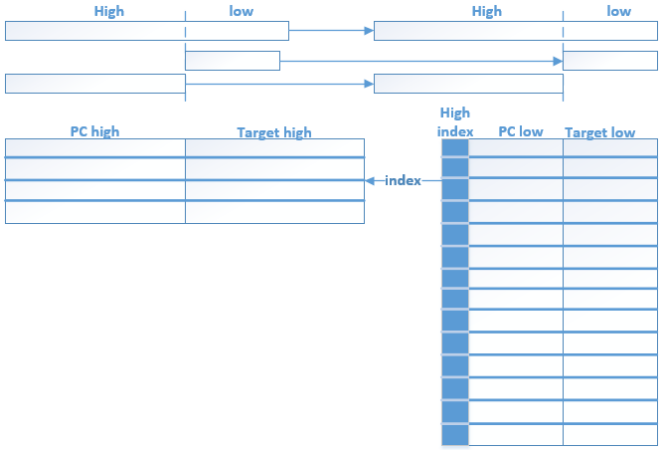


图 3.7 观察一引出的物理数据结构的优化。左侧为高位映射表，右侧为低位映射表

Figure 3.7 the physic data structure optimization by observation one. The left is the high bits map table, the right is the low bits map table.

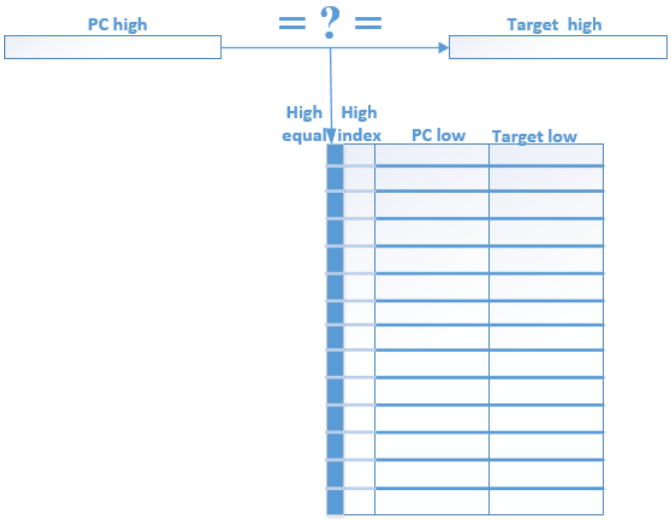


图 3.8 观察二引出的物理数据结构的优化。只包含了低位映射表。

Figure 3.8 the physic data structure optimization by observation two. Only include the low bits map table.

3.7.6 前端部件的整合

上面讲述了前端的各个组件，这些组件整合起来构成了整个处理器 BIAN 的前端，如图3.9。

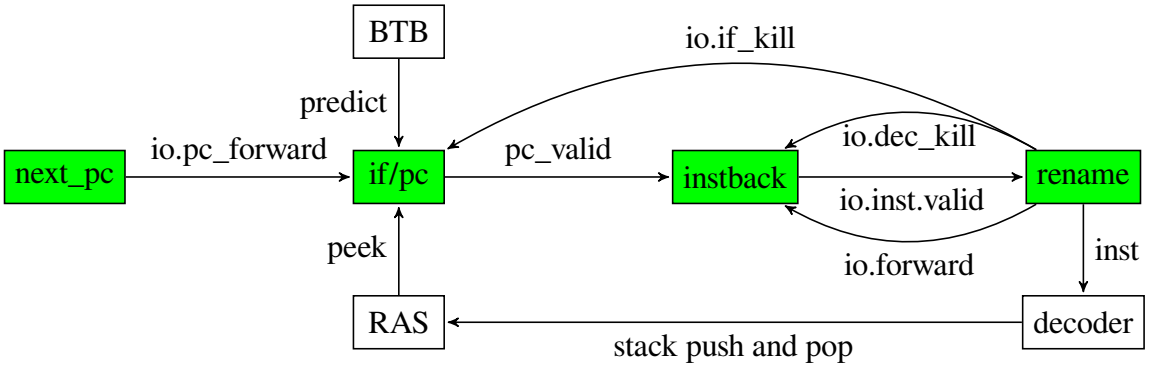


图 3.9 处理器 BIAN 的前端流水线。

Figure 3.9 the pipeline of frontend.

整个前端是一个三级流水，因为指令 in-flight 的请求至多为 1 的基本限定，十分简洁。主要体现在流水线的 3 大控制信号上：`io.pc_forward` 是控制取指 PC 从 next_pc 级流向 if/pc 级的使能信号；`pc_valid` 是控制取指 PC 和 BTB 预测信息从 if/pc 级流向 dec 级的使能信号；`io.inst.valid` 是控制取回指令和地址预测信息从 dec 级最后流向后端 rename 级的使能信号。这 3 级流水的阻塞与流通可以简单的解释为：首先最靠近后端的 dec 级如果在 `io.inst.valid` 和 `io.forward` 同时置上的时候，才能被后端接收，下一周期空出这一级，而 if/pc 级的信号要流向 dec 级须等 dec 级在下一周期能够被空出来，同理 next_pc 级的信号要流向 if/pc 级须等 if/pc 级在下一周期能够被空出来。

相比于单宽度的前端，双宽度的前端还需要额外考虑两条并行的指令之间的干扰现象，具体来说是第一条指令对第二条指令的影响。在前端流水线的 3 级中：

1. **if/pc 级**，虽然对外的取指端口是一个 PC，但是这个时候对内的 PC 已经分化成奇偶两个。在 BTB 的结构中，如果第一条 PC 命中，并且方向为跳转，那么第二条的 PC 就会被无效掉。这一级的干扰现象被取指单元所解决。

2. **inst back 级**，为了使得后端的逻辑简单化，通过局部译码，得到如果两条并行指令都是特权指令或者都是分支跳转指令时，就会阻塞该级流水，使两条指令串行发送给后端。

3. **rename 级**，转移预测单元若在该级的第一条指令上纠正了流水线上游传

输过来的预测信息，同样要将与之并行的第二条指令无效掉。

3.8 中间层的设计

中间层承接着处理器的前端和后端。前端的前端指令队列、分支跳转预测队列的出口；后端的指令发射队列、分支跳转单元队列和访存队列的入口均位于中间层。中间层还专门负责管理处理器的状态变化与恢复，并能够按照每条指令的不同需求，精确地分配各个物理资源。是乱序处理器中最为重要的结构之一。

中间层涉及了很多部件单元。不过在论述每个具体部件之前，不妨先分析清楚承载这些功能单元的基本数据结构 — 队列。

队列在电路中如图3.10有两种实现形式:

(a) **循环队列**，依靠着头尾两组指针来控制逻辑。称之为“组”，是由于考虑到同时入队列和同时出队列可能不止一项。入队列移动尾指针，出队列移动头指针。

(b) **移位队列**，只需要依靠一个计数器来记录当前队列中有多少项即可。入队列从队列的尾部依靠计数器的指示进入；出队列项的后面诸项需要向前移位来填补出队列的空白。

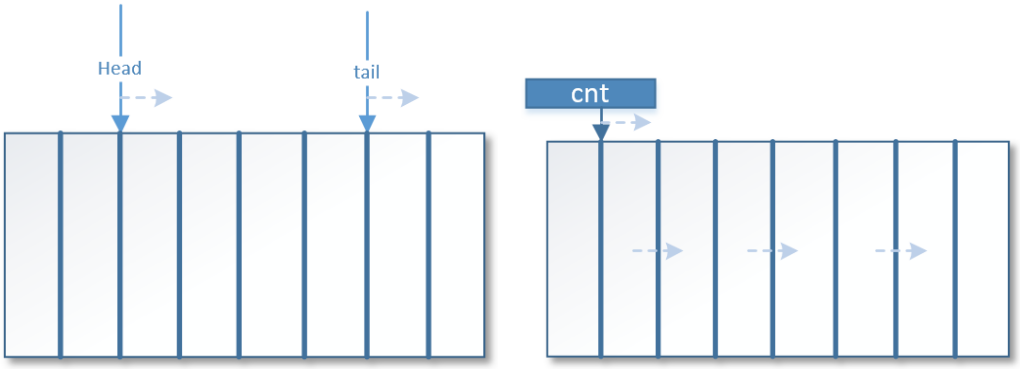


图 3.10 两种类型的队列。(a) 循环队列，(b) 移位队列。

Figure 3.10 Two types of queue. (a) Ring queue, (b) Shift queue.

这两种不同的形式各有优势。循环队列对于出入队列的项数约束较松；移位队列入队列的项数约束较松，但是对于出队列的项数约束较严，一项是最为简洁的。项数一旦大于 1，复杂度就会明显地提高，物理实现变得复杂，同时移位队列的功耗比循环队列大很多，因为移位队列平均 1/2 项数都会移动，而循环队列只需要移动头尾指针。

但是循环队列并不适合承载乱序的逻辑，这是由其逻辑特性决定的，出队列项一定是头指针所指向的；与循环队列相反，移位队列非常适合承载乱序的逻辑，队列中的任何一项都可以出队列，留下的空位由后面项移位来补。同时，乱序中的多项同时可以出队，只选最早一项的逻辑，用移位队列来实现显得非常简单，只需要调用优先编码器，传入可以被发射的队列向量，最后按照优先级得到的独热码对应的就是应该出队列的最早一项。

但是移位队列的功耗毕竟太大，特别是每一项位数很大的时候。出于降功耗的考虑，对移位队列的结构做了优化如图3.11所示，只将能够决定出队列条件的最关键控制信号填入移位队列中，其他的数据信号则填入到数据表中，然后把该表项的索引同时填入移位队列中。出队列时再去索引数据表得到相关的数据。

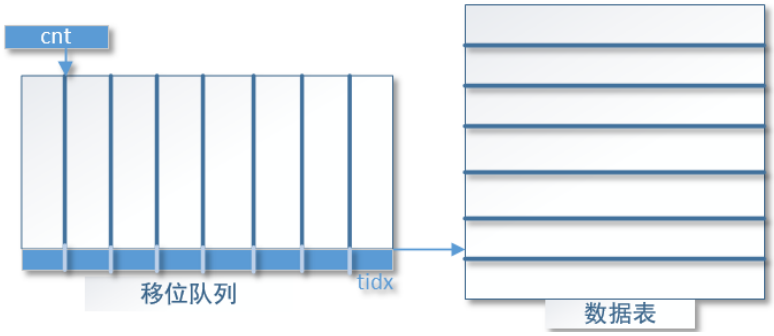


图 3.11 低功耗的优化，移位队列加数据表。

Figure 3.11 shift queue plus data table, a optimization for saving energy.

整个中间层和后端就是用这两个最为基本的数据结构组合而成。一些复杂的操作流程，都可以规约为对队列的操作。以这样的角度去设计乱序多发射处理器，将会最大限度的控制复杂度，做到最大程度的简化。

3.8.1 前端指令队列

除了有高速缓存的前端，高性能处理器在后端 (包括中间层) 中同样有一定数量的指令缓存，在 BIAN 的设计中被划分为了三个执行梯队 (参见章节3.5)。前端指令队列存储第三梯队的指令，入口位于 inst back 级，出口位于 rename 级。前端指令队列的结构如图3.12，为循环队列的实现形式，所以第三梯队的指令是顺序被执行的。

队列的每一项能够容纳两条指令，与取指器的宽度保持一致，让前端指令进入队列的逻辑保持简单。从取指单元取回来至少一条有效的指令，当后端没有剩余空间来接收时，会被压入该队列，从而不影响前端取指单元的继续取指。直

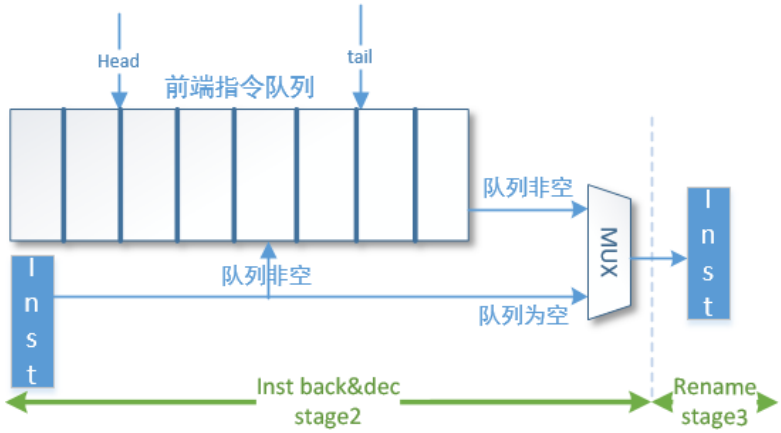


图 3.12 前端指令队列

Figure 3.12 The queue storing for frontend instruction .

到该队列满了，才会阻塞前端。BIAN 中此队列有 16 项，容量一共是 32 条指令。执行 benchmark 程序时，几乎都为空的状态，主要用来隐藏 16 个周期的访存延迟。

3.8.2 分支跳转预测队列

从前端传来的不仅有指令，同时也有转移猜测的信息。直观上来讲，转移猜测的信息可以和指令一起存放在章节3.8.1的前端队列中。但是由于前端指令队列的入口在 inst back 级，对于转移预测信息来说入队列的时机太早，所以不同于前端指令队列，分支的预测信息专门存放在分支预测队列中，在图3.13流水线的中间部位。

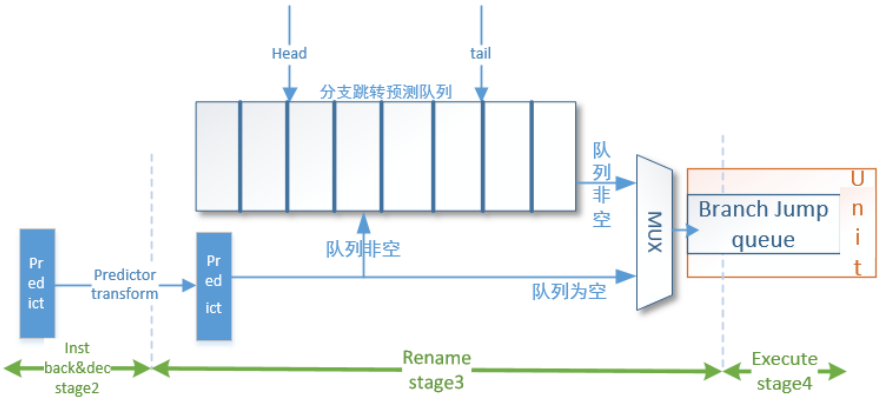


图 3.13 分支跳转流水线

Figure 3.13 The pipeline of branch jump instruction.

该队列为循环队列，入口和出口均在 rename 级，所以预测信息可以等到 rename 级预测单元对预测信息做最后的修改再存入队列中。使用二选一的旁路

来保证和出前端指令队列的指令是一一对应的。最后预测的信息会被写入分支跳转单元中的队列。因为前端在 inst back 级做了当两条并行指令同时为跳转指令将其串行化的逻辑，所以保证每一周期最多只有一条跳转指令。所以分支预测队列每一项的容量是一条指令的预测信息 (通过二选一的逻辑选择出跳转指令的预测信息)，这样，面积的开销减少一半。

3.8.3 为什么不需要 PC 队列

前端传来的不仅有指令和转移猜测信息，还有 PC。直观来看，也应该有一个 PC 队列来存放每个指令所在的 PC 值。但是事实上，每一周期的 PC 也可以由分支预测队列存放的预测目标以及重定向使能信号推导出来。如果预测信息是错误的，也可以通过后端发送的取消信号重新更正 PC。所以没有必要把 pc 存起来，这样，大大降低了处理器的面积开销。

3.8.4 发射队列

发射队列负责存储第二梯队的指令，结构如图3.14，出口和入口都位于 rename 级，指令先进过重命名表，将逻辑寄存器号转换为物理寄存器号，并分配好资源后压入队列的尾部 (为了做一个周期的优化，也做了旁路，当发射队列空时，不需要进入发射队列而可以直接选择发射)。不同于之前提到的各个队列，发射队列采用的是移位队列的实现方式 (图3.11)，这样可以支持乱序出队列，是实现乱序化最重要的一步。在细节上，移位队列每项的信息可以精简到两个物理寄存器号加上是否需要继续侦听的标志位和一个用来索引数据表的 id 号。这样，每项总共 20-bit 左右，其余的信息可以存储在数据表中，故移位逻辑的功耗得到了有效的控制。

由本章节开头对移位队列的分析，出队列项数为 1 是移位队列最合适的设置。但是双发射乱序要求发射项数为 2，为了能够兼顾这矛盾的两点，采用的是两路并行的发射队列，事实上，图3.14中只给出了其中一路。每路每一周期都会从队列里弹出一项读寄存器堆然后发射到执行级。入队列也是严格按照物理的位置静态地每一路至多压入一项。出于时序的考虑不会采用动态分配的策略 (也即如果第一路的队列满了，第二路队列未满，偶数 PC 的指令不会被动态压入第二路队列中)。这样的设定虽然会产生两路队列指令数量不平衡的情况，但是会由下游的执行队列来很大程度的缓解。

图3.14中的 head 是专门考虑电路时序的设计，这样发射的时候就不需要经过发射队列多选的过程。为了规整，移位队列主体为 8 项凑足 2 的幂次。若项数

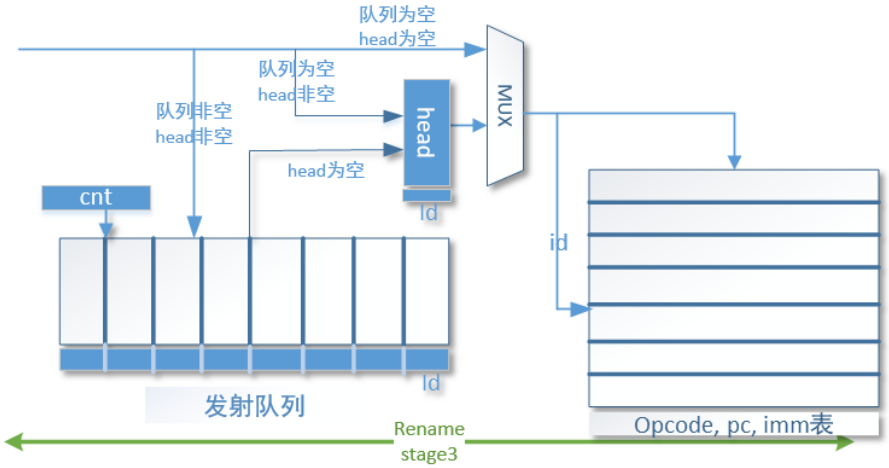


图 3.14 单路发射队列

Figure 3.14 One way of the inst issue queue

比 8 更多，侦听的电路效率就会下降。所以 head 的设计还增加了两条指令的容量。这样，两路发射队列总共的容量是 18 条指令。

在 BIAN 的发射队列设计中，有 Alpha 21264，MIPS R10000 和 BOOM 都没有的独特设计。从第2章的分析中看出，这三款处理器中，发射队列发射的指令必须是操作数都已经准备就绪的，当发射队列没有指令准备就绪时，这些处理器都没有打破僵局的机制和能力。但是 BIAN 处理器不同，发射队列依旧可以发射最早的指令，读取目前已经有效的操作数，然后发射到执行级，缓存入执行队列中 (前提是执行队列未满)，相当于是将指令的执行等级从第二级切换到了第一级。符合指令调度的直观。

3.8.5 处理器状态控制单元

在章节3.4中，已经说明了什么是处理器的状态。处理器状态的正确变化是衡量处理器设计正确性的唯一标准，因此控制处理器状态变化的模块单元是处理器设计的重中之重。

无论是顺序处理器，超标量处理器，乱序处理器，本质上都是根据指令正确的改变处理器的状态。所以以一种更高的角度来看，高性能的处理器是能够高效的改变自身状态的机器 (machine)。

乱序中最大的挑战在于处理器状态的维护和发生指令流发生错误的纠正。主要体现在三点上：

(a) 转移预测错误，要及时清除跳转指令后的错误指令流对处理器状态的改变，恢复至到跳转指令为止的处理器状态。

(b) 访存出现写后读冲突，但是后读的 load 指令先于写提交而且没来得及做数据前递，需要将状态恢复至到该 load 指令前面一条指令为止的处理器状态。前端重新从这条 load 指令起开始取指。这个在处理器设计的专业术语里叫做回滚 (roll back)。

(c) 例外和中断的发生。需要将状态恢复到中断例外指令之前的所有指令提交完的处理器状态。

顺序执行的处理器，其状态是非常好维护的，因为指令都是顺序修改寄存器堆中寄存器值的，指令流出现错误也不会产生错误的处理器状态。但是乱序不一样，错误的指令流可以在未检测到之前对处理器状态做出改变。在目前的 BIAN 处理器设计中，实现的是 RAM 形式的重命名表，故处理器状态的恢复回溯也要以 RAM 重命名表为中心。下面是代码所描述的处理器状态的变量集合，以类的形式封装在一起。

```
class State(val nEntry: Int) extends Bundle {
  val useing = UInt(nEntry.W)
  val usecnt = UInt(log2Ceil(nEntry+1).W)
  val maptb  = Vec(32, UInt(log2Ceil(nEntry).W))
  val rename = Vec(32, Bool())
}
```

基于 RAM 的重命名表在上述代码中是 `maptb` 变量，nEntry 是物理寄存器堆的项数，所以 $log2Ceil(nEntry)$ 就是索引物理寄存器堆地址的位数。BIAN 处理器设置了 60 项物理寄存器。其他变量的含义分别是：

(a) `useing`: 正在使用的物理寄存器分布向量，每一个物理寄存器占一位，1 表示正在被使用，0 表示空闲，可以被分配。

(b) `usecnt`: 正在使用的物理寄存器计数器，产生分配物理寄存器的 ready 信号，比如需要分配一个物理寄存器需要条件 usecnt < nEntry 成立。

(c) `rename`: 逻辑寄存器号是否已经被映射到某一个物理寄存器号，如果是则为 1，不是为 0。含义是清晰的，用于写后写冲突中，后写的指令能够在指令提交时回收旧的物理寄存器。

所以有了明确的状态刻画，需要有多少个 `State` 类只需要 `new` 出来即可。那么一共需要多少个 `State` 类呢？首先要有一个最新的状态，负责当前的逻辑寄存器号的重命名和物理寄存器号的分配，这一个作为主要的 `State`。其次为了精确的状态回溯，考虑要有状态的备份。回到上述提及的三种需要状态回溯的情况，可以分为两种不同的处理机制：第一种，发现指令流错误，马上撤销之后

的指令流和处理器状态变化，发送重定向请求到前端；第二种，等到前面指令都已经顺序提交退出，再做发送重定向的请求，并撤销后续指令流和处理器状态变化。像转移预测错误因为比较频繁发生，所以一般做成第一种机制；而像回滚和中断例外，很罕见，可以做成第二种机制。BIAN 处理器正是采用这样的设计方案。第二种机制需要有一个 State 备份记录至提交指令为止处理器的状态；第一种机制又可以有两种不同的做法，参见章节2.1中对 Alpha 21264 和章节2.3中对 MIPS R10000 的分析，在 BIAN 中采用的是 MIPS R10000 的做法，只针对每一条跳转指令做一个 State 备份。而在 BIAN 处理器中乱序部分最多支持四条分支跳转指令的投机，所以状态有四个备份用来回溯。代码如下：

```
val latest = RegInit({ //当前用于重命名的最新的处理器映射状态
  val w = Wire(new State(nPhyAddr))
  w.maptb  := DontCare
  w.useing := 0.U
  w.usecnt := 0.U
  w.rename := VecInit(Seq.fill(32)(false.B))
  w
})
val commit = RegInit({ //是回滚和中断例外的状态备份
  val w = Wire(new State(nPhyAddr))
  w.maptb  := DontCare
  w.useing := 0.U
  w.usecnt := 0.U
  w.rename := VecInit(Seq.fill(32)(false.B))
  w
})
//nBrchjr = 4, 分别对应每一条投机执行的分支跳转指令的状态备份
val backup = Reg(Vec(nBrchjr, new State(nPhyAddr)))
```

有了状态的描述，回溯恢复就是把相应要回溯的状态变量赋值给 latest 状态即可 (省略复杂的细节)，非常的简单直观。

除了基于重命名表的状态记录和回溯机制，作为处理器状态的主要控制单元，还有两大主要的功能：

(a) 指令 id 号的分配，载体是 ROB，组织形式是循环队列，id 号从尾指针顺序分配，从头指针顺序回收。

(b) 物理寄存号的分配。分配的时候在 latest 状态变量的 using 属性中，从左边找第一个 1，从右边找第一个 1，并行分配两个物理寄存器号。

指令 id 号是标记处理器乱序部分顺序的唯一标识符。BIAN 的分配的 id 号

是 5 位的，对应于乱序部分最多可以 32 条 in-flight 指令。

3.8.6 分支跳转队列

分支队列和访存队列就是面向后端的数据结构，分别在后面章节3.9.2和3.9.3中会详细说明设计中的亮点。

分支跳转队列的入口在中间层，说明是顺序分配资源进入队列，而不是执行时乱序分配的。原因在于分支跳转是控制相关的，带有顺序属性，在乱序中也要保持相对的顺序。采用的数据结构组织形式和发射队列类似，为移位队列，如图3.15。

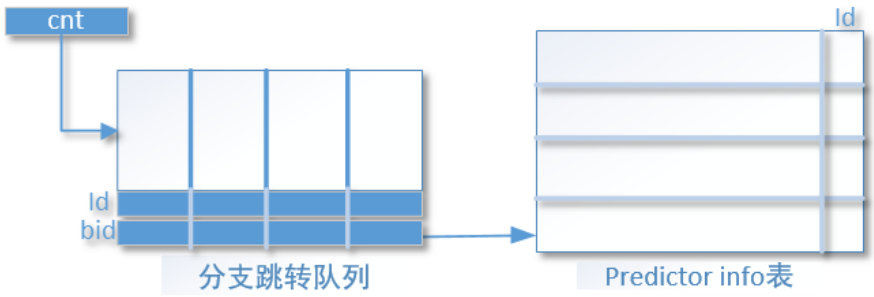


图 3.15 分支跳转队列。

The shift queue storing for branch and jump instruction.

3.8.7 访存队列

这里解释两个和访存队列有关的问题：其一，直观来讲，访存指令并不是控制相关的，那么为什么访存单元队列的入口设置在中间层，要求顺序的时候分配资源呢？其二，从图3.3给出的访存队列数据组织结构中，为什么不和发射队列和分支队列一样采用的移位队列，而是用循环队列呢？

核心的原因在于：首先，对比普通寄存器数操作指令，写后写，写后读的冲突可以由重命名机制完美地解决，但是重命名机制却做不好访存指令的写后写，写后读冲突，因为访存的地址不是在 rename 级得到的，而是在乱序的执行级得到的。乱序执行是无法检测访存指令之间写后写，写后读的冲突。所以为了解决这个问题，这里就必须用队列的顺序分配来对保持访存指令之间的相对顺序。其次，因为 store 的写操作是对外的，所以已经超出了章节3.8.5中论述的处理器状态控制的范畴，要实现精确的状态恢复，store 指令必须在 LS commit 级将数据顺序写回内存中，其顺序性需要有队列这样的 first in first out 的结构来保证。

队列类型不用移位队列而采用移位队列的原因在于：为了判读写后写，写后读的冲突，整个 32 位或者 64 位地址都将是关键信号，也要在移位队列中进

行移位，如此一来功耗将会非常大。所以从功耗的开销和所获得的收益的角度来考虑，BIAN 处理器做成 6 项循环 LDQ 和 6 项循环 STQ，已经大概率的满足 in-flight 的 32 条指令中访存指令的存放需求。

3.8.8 精确资源分配

在中间层的设计过程当中，发现资源的精确分配逻辑非常复杂。资源的精确分配，指的是根据指令精确到对各个物理部件资源的不同需求和对应的各个物理部件单元剩余容量，做出资源分配。

这个逻辑的复杂性在于：

(a) 不同指令的需求是不一样的。有的指令有写寄存器的需求，所以需要分配物理寄存器；有的是跳转分支指令，在分支队列要分配项数；有的是访存指令，需要在访存队列中分配项数。

(b) 不同资源的剩余容量参差不齐，只要有一种资源指令有需求，但是无法分配，这条指令相关的其他指令都不能分配。

(c) 同一指令的需求是多样的，可能既需要物理寄存器，也需要访存队列

(d) 需要对两条并行的指令进行同时分配，第一条指令的分配情况会组合逻辑的作用于第二条指令发分配情况。所以并行的指令如果要实现精确分配物理资源就必须是串行的。

表 3.1 中间层需分配资源列表。

Table 3.1 The list of resources in the Middle end.

资源名称	备注
指令标识符 id	每条指令都必须分配，最多同时分配 2 项
物理寄存器号	有写寄存器的需求的指令分配，最多同时分配 2 项
发射队列 entry	每条指令都必须分配，每路队列最多分配 1 项
访存队列 entry	访存指令需要分配，最多分配 2 项
分支队列 entry	分支跳转指令除了 jal 类，最多分配 2 项

所以多发射宽度更高如四发射的很多处理器都采用了保守发射的逻辑来避免精确带来的复杂性。只要指令有效，就认为指令有所有的需求。换言之，在分配资源时只需要顾及指令是否有效即可。另外，在复杂处理器的设计中，由于各个部件单元的面积都很大，布局布线后的位置也会相隔较远，所以若采用精确发射走线延迟会很大。但是 BIAN 处理器后端面积不大，资源队列的项数较小，故走线延迟在可接受范围以内。另一方面，项数少就要采用精确分配的逻辑来节约

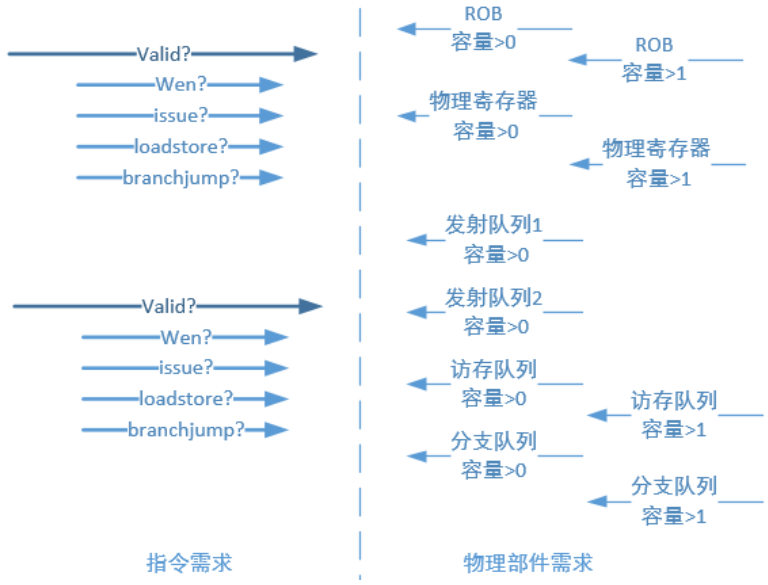


图 3.16 资源精确分配分析。

Figure 3.16 analysis of precisely resources allocation.

使用。所以权衡之下，BIAN 处理器中间层资源分配采用精确的方式，来提高资源的利用率和执行的效率。

BIAN 处理器中间层需要分配的资源有五个，如表3.1所示。一方面是两条并行指令的各个需求以及是否有效，另外一方面是各个物理部件的容量信息，如图3.16所示。

3.9 后端的设计

执行队列是完全属于后端的单元；分支跳转单元和访存单元除了队列入口在中间层外，部件的主体在后端。在 RV32I 中，执行队列里出队列的指令若为 ALU 指令，因为是单周期的，所以直接写回寄存器堆并做旁路前递即可；若为跳转指令，会进入分支跳转单元；若为访存指令，会进入访存单元。

3.9.1 执行队列

这个队列就是上文提及的执行等级的第一梯队，和发射队列一致，采用图3.11移位队列。但是有 3 个特质使之区别于发射队列：

(a) 在数据表中每条指令都存储了两个源操作数，发射指令出去执行的时候不需要读寄存器堆，不会增加读端口的数量。

(b) 没有像发射队列 (图3.14) 中 head 的结构设计。发射靠组合逻辑多选。

(c) 必须等到操作数都已经准备就绪，才能出队列执行。

为什么会有执行队列的设计。一方面它符合指令调度的直观。另外一方面，从处理器的设计经验来看，电路延迟最大流水级是第二级的 inst back 级和第三级的 rename 级，反倒是执行级会很快。而且从处理器发展的历史来看，运算单元如 ALU 会越来越快。所以执行级的时序是相对充裕的。如何使这一级“充足”起来，不浪费多余的时序，这就是执行队列设计的初衷，也即通过在执行级加入少量的不会使之成为关键路径的指令缓存，来提高指令调度的效率。

在 BIAN 的设计中，一共有 3 路并行的执行队列，是 $2+2+4$ 的模式，其中两个 2 项的队列只能允许从 rename 级传过来的对应物理位置的 issue 入队，也即是每一路 issue 所私有的。4 项的执行队列是两路 issue 所共享的，但是每一周期只允许有一个 issue 入队，若有两路 issue 均有效，不能执行，且被私有的执行队列拒收，而同时产生了进入共享的队列的请求，这个时候会通过仲裁比较 issue 的先后选择早的 issue 入队列。

每一路队列都要求从头部开始指令的 id 号递增，由于在 rename 级的发射队列中存在：投机发射以 load 指令结果为源操作数的指令的机制，访存指令因为 LDQ 和 STQ 空间不够而阻塞发射机制的一些细节，无法保证在执行级入执行队列的时候 id 号是严格递增的，所以这个需要额外的逻辑来维护。当发现 id 号不是递增的，将不会被写入执行级队列，继续锁存在执行级一开始的 issue 锁存器里，同时阻塞 rename 级发射新的 issue 到 issue 锁存器中。

执行队列虽然每一项的开销大，但是项数很少，而且分了 3 路，并行程度高，并不会给执行级带来很大的负担。同时因为有共享队列的存在，能够在一定程度上平衡了 rename 级两路发射队列的指令数量。

3.9.2 分支跳转单元

由分支跳转队列 (见图3.15) 构成，对接着前端的转移预测器，构成了整个反馈回路的分支跳转系统。这一系统在高性能处理器的重要性仅次于访存系统。高性能的分支跳转系统在设计的时候有 3 大指标需要考虑：

(a) 高准确率，是设计的首要指标。

(b) 低周期数，从跳转指令的下一周期开始，到跳转指令被执行能够向前端发出反馈信号为止，需要的反馈周期数要尽可能的少。

(c) 其他：低延迟，小面积，低功耗。

在分支队列的设计中，着重需要考虑的指标是低周期数。在 BIAN 的设计中队列有四项，能够支持 4 项分支的处理器状态恢复。乱序的 32 条指令中这四条

分支能够把整个指令流分成 5 段，每段的平均长度在 6 ~ 7 之间，符合跳转分支指令在一般程序中的频率。图3.15和标准的移位队列加数据表 (图3.11) 有些出入。主要体现在移位队列和数据表都多了指令 id 域。这样，执行级乱序执行的指令可以利用 CAM 比较在数据表和移位队列并行查找。并行的考虑正是为了降低反馈周期数而专门设计的。反馈做到可以走组合逻辑路线的而不需要走时序逻辑路线。在 RISC-V 的分类中，jalr 类指令需要以寄存器为源操作数进行运算；branch 指令在中间层能够通过 PC 和指令中的立即数能够算出跳转地址，但还不知道跳转方向；jal 类指令在中间层直接可以算出跳转地址，并直接对前端取指进行重定向，就不需要进入分支队列。故只有 branch 和 jalr 两类指令会进入队列。在执行级，branch 真正做的操作只有两个操作数的比较，工作量相对较少，所以就非常适合组合逻辑反馈到前端；而 jalr 类指令因为要算地址，所以会先进入队列和表项中，在下一周期再对前端做出反馈。当同时有多个分支从 ALU 执行后进入分支单元是，选出最早的一条分支指令去反馈，其他的会被存到队列中。

3.9.3 访存单元

访存部件由 3 个循环队列构成，分别是 LDQ，STQ 和 Loadstore 队列 (参见图3.3)。在重命名阶段分配，每个队列的每项都有全局指令 id 域，在执行级乱序执行的指令可以利用 CAM 查找到队列中对应的项并做出更新。LDQ 和 STQ 意义和功能都非常明确，资源的释放也都是在 load 和 store 指令提交的时候。

LDQ 和 STQ 意义都是非常明确的，而被称为 Loadstore 队列的这第三个队列有什么设计上的考虑和用意。

引入这第三个队列的考虑在于：凡是在中间层分配的队列资源，都有一个缺点，就是只要指令有入队列的请求，但是队列已满，那么后端就会一律暂停接收来自前端或者中间层前端指令队列发送过来的指令。所以跳转指令和访存指令都存在这个问题。对于跳转指令而言，由于是控制相关的，维护起来代价较大，不易做多，不得不牺牲这方面的考量。而访存指令的情况不同于跳转指令，首先，一长串的连续的访存指令不少见，尤其在访存密集的程序中，6 项的 LDQ 或者 6 项的 STQ 如果遇到这种情况不够用，就必须阻塞后面的指令。而且此时有可能发射队列还很空，如 6 条 load 指令分散于两路，也就是每路 3 条，乱序资源得不到有效的利用，所以有必要对 LDQ 和 STQ 扩容；但是另外一方面，在一长串的访存类指令中，一般都是处于同一或者相邻 cache 行的，所以真正需要长时间等待访存结果返回的指令较少，对 LDQ 和 STQ 扩容会增加无谓的面积去

存储暂时不被执行而得不到的地址数据。

综上所述，现在面临的两难局面是：一边为了提高乱序资源利用率而主张扩容，一边为了不增加访存队列面积而反对扩容。解决的方案就是增加一个 Load-store 队列。这个队列里每一项的信息非常少，只有一个全局指令标识符 id 号，一个访存类型，一个 load-store 标记位。这个队列做到 8 项，占用的总资源也不会超过 80-bit，非常的小巧。访存指令会优先选择进入 STQ 或者 LDQ，只有到 LDQ 或者 STQ 满了，才会填入 Loadstore 队列。而 STQ 或 LDQ 释放的资源后也会优先让 Loadstore 队列中的项填入，只有 Loadstore 队列为空是，才会将中间层的访存指令信息填入。同时为了逻辑的正确性，必须规定凡是在这个 Loadstore 队列里的指令不能被执行，直到该条访存指令被转移到了 LDQ 或者 STQ，才能重新获得被执行的权利。这样，Loadstore 队里就会在保证的处理器正确性时，提高运行访存密集程序的性能。

访存部件涉及到后端所有的流水级，而且又是与 CPU 的外部信号交互，是后端中最复杂的一个部件。下面粗略的说明每一级的操作 (省略了很多细节)：

(a) **rename 级 —** 访存指令进入相应的队列

(b) **execute 级 —** 计算出访存的地址写入相应的队列中

(c) **LS memory 级 —** 向外发出加载请求，并且遍历队列得到写后读的相关 forward 信号

(d) **data back&fwd 级 —** 数据加载回来，进行必要的 **forward 操作**，写回。

(e) **LS retire 级 —** 如果是 load 指令直接退出；如果是 store 指向内存写完数据之后退出，同时 store 要做相关的 **backward 操作**，判读是否与已经取指回来的 load 指令有写后读的冲突，做必要的回滚。

由于写后读的冲突的存在，使得 STQ 和 LDQ 虽然是两个队列，但有着密切的关联。BIAN 处理器中采用 forward 和 backward 两种操作操作来维护 STQ 和 LDQ 之间的关联。forward 操作的主体是正在向内存发出访存请求的 load 指令，动作是在 STQ 中前向扫描在该 load 指令之前的 store 指令，如果有地址冲突，store 的数据就要前递到 load 的写回数据中；backward 的操作主体是处在 STQ 队列头部的 store 指令，动作是在 LDQ 中后向扫描在该 store 指令之后的 load 指令，如果存在读写地址冲突的情况，并且对应的 load 指令在没有做相应的 forward 操作的情况下，就已经写回了，这时要向位于中间层的状态控制单元发出回滚的请求。

forward 和 backward 操作示意图如图3.17。为了确定 forward 和 backward 的

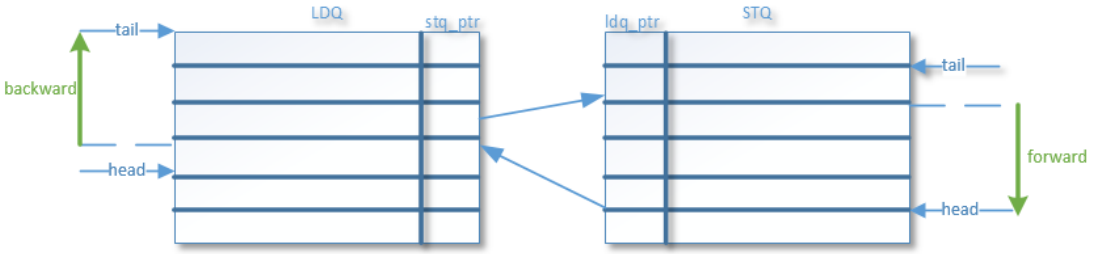


图 3.17 LDQ 和 STQ 的前向传导、后向传导操作。

Figure 3.17 forward and backward operation of LDQ and STQ.

起点，在原来的队列中各要加入指针域，在 LDQ 中是索引 STQ 的指针 `stq_ptr`，在 STQ 中是索引 LDQ 的指针 `ldq_ptr`。

3.9.4 特权态指令执行

前面提到的后端各个数据结构存储的都是针对用户态指令的。对于特权态指令，因为其稀少 (不包括少数内核程序) 而且特殊，所以在目前 BIAN 的设计中并没有对 CSR 采用重命名的机制，所以特权态指令不支持乱序。而且考虑到潜在的副作用，特权态的指令不能投机执行。BIAN 的这种处理方式和 BOOM 类似。

3.9.5 后端状态恢复机制

状态的回溯在前面中间层的处理器状态控制单元 (章节3.8.5) 已经论述过，除了中间层状态控制单元负责处理器以重命名表为主的状态回溯以外，后端的数据结构的状态同样需要精确地恢复。由于后端数据结构都可以规约到队列的形式，或者循环队列，或者移位队列，这大大简化了状态恢复的逻辑。

对于移位队列，如果是例外中断或者访存回滚，计数器置为 0 即可；如果是跳转指令预测错误撤销，将计数器置为队列中标识符 id 号小于该跳转指令标识符 id 号的指令个数。

对于循环队列，如果是例外中断或者访存回滚，将尾指针置为当前的头指针，也即头尾指针相等；如果是跳转指令预测错误撤销，将尾指针移到最小的比该跳转指令标识符 id 号大的那一项处。

同时，BIAN 处理器的中间层和后端的所有数据结构的恢复都会在 kill 信号 (中断例外，访存回滚，转移预测错误的统称) 被置上的延后一个周期，这是为了关键路径的时序考虑。同时，由于前端还尚未发送指令到后端，所以这一做法对处理器的效率没有任何影响。

3.10 小结

从 BIAN 的整体框图3.2可以看出，在设计的过程中为了进一步提高处理器的性能，用到了很多的旁路技术，来压榨剩余的电路延迟。时序上的好坏可以通过后期的综合工具来磨合。比如涉及到关键路径的旁路，如果对性能的影响不大，就可以去掉的，来提高处理器主频。

BIAN 的 60 项寄存器堆读端口数量维持在四个，对物理的布局布线，电路主频都十分友好。写回总线的宽度为 4，也即寄存器堆的写端口为四个，在取指宽度为 2 的乱序处理器中设置合理。

执行队列的引入、三个执行等级的划分、以及乱序指令缓存 (第一, 第二梯队) 和顺序指令缓存 (第三梯队) 的结合都是 BIAN 中的设计亮点。这使得 BIAN 处理器在中间层和后端一共能够支持 32 条左右 in-flight 的乱序指令和 32 条 in-flight 的顺序指令。

另外，在执行队列中，对访存指令做了特殊的优化，使之能够优先的将算出地址的访存指令发送到访存单元中，这样能够提前检测到写回式一级数据高速缓存 (目前暂没有实现) 可能出现的 miss 情况，并将 cache 行提前取回，从而更好的隐藏访存延迟。

第 4 章　实验与分析

4.1　仿真实验平台

本科学习期间采用的指令集是 MIPS32，实验课上设计的单发射五级静态流水线处理器是基于龙芯提供的实验平台。设计中经历的步骤程序有：**step1**: 仿真时与龙芯提供的处理器 loogson132 生成 (打印) 的写回级 trace 对自动化的逐拍比对 ⇒**step2**: 若比对出现不同则停止仿真，通过看波形找到产生不同写回信息的原因，并修正之 ⇒ 如此循环往复直到仿真调通 ⇒**step3**: 之后再用 Vivado 的集成化工具综合布局布线 ⇒**step4**: 最后生成 bitstream — 可配置 FPGA 的格式文件，用 JTAG 线下载至 Xilinx 公司的 FPGA 上，产生物理上预期的效果就算设计完成。需要注意的是为了简化，课上实验中的内存是使用板卡上的 SRAM 模拟的，并不是真正的内存。

但是这样一套比较熟悉的平台，却很难将 RISC-V 移植进来。没有现成的可以用来比对的标准 trace；Chisel 代码内部逻辑变量没有很好的波形调试。所以原来的实验平台并不奏效。

很自然会想到在 RISC-V 开源处理器仓库中有没有可以略加修改便可使用的设计验证平台，于是尝试着阅读了 Rocket 和 BOOM 的说明文档及其代码。BOOM 外围的很多代码是复用 Rocket 的，意味着存在着一种可能性 — 区分出 BOOM 处理器自身和复用的 Rocket 代码，找到两者之间的接口，然后把 BOOM 替换为自主设计的处理器核心。但是不得不说，阅读代码的过程非常痛苦。首先 BOOM 和 Rocket 都是达到工业级别的颇为成熟的大工程了，里面的文件众多，每个文件中又有不同的类，这些类之间又有着非常复杂的交互关系。其次这些处理器的设计者和 Chisel 编译器的设计者属于一个团队，里面灵活使用了很多 Chisel 和 Scala 中高级的语言特性来更为简洁，扩展性更强的描述电路，但是这些特性却是初学者一时很难理解的障碍。加之本设计的任务并不是在 BOOM 的基础上添加模块，也不是修改 BOOM 的微结构来达到性能的提高 (再者说，这样难免有抄袭之嫌)，任务在于自主设计出一款和 BOOM 一样完整的处理器内核。所以如果需要和 BOOM 一样复用 Rocket 的外围代码，然后去跑仿真验证，就必须清楚的了解其中的没一个细节，这一个熟悉的过程最少也需要几个月的时间，时间上不允许。

最后采用的是加州大学伯克利分校体系结构课教学配套的适用于初学者的

实验平台。如今，这个实验平台以及几个简单的能够在平台上运行的处理器核已经作为伯克利分校 RISC-V 开源仓库列表ucb-bar中的riscv-sodor 仓库开源了。另外，还有一个论坛社区(https://gitter.im/librecores/riscv-sodor) 来答疑和解释该平台运行的机制。其机制可以简化地用图4.1来概括。

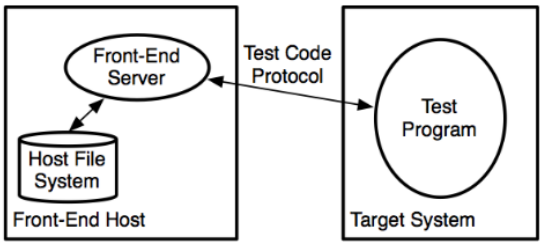


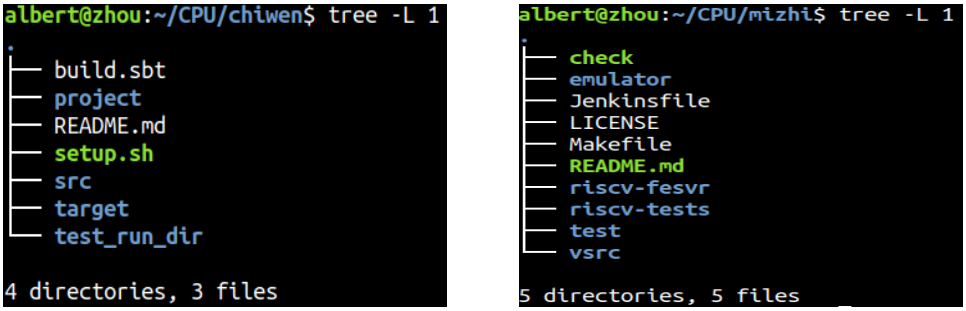
图 4.1 测试环境。前端服务器将 RISC-V 二进制文件加载到主机的文件系统中，启动目标系统的模拟器，发送 RISC-V 二进制代码到目标模拟器填充入模拟的处理器指令内存中。一旦前端服务器完成发送测试代码后，服务器重启目标处理器然后目标处理器就可以始从固定的地址开始执行程序。(John Wawrzynek, 2016) 图 2.

Figure 4.1 The Testing Environment. The front-end server (fesvr) loads the RISC-V binary from the Host file system, starts the Target system simulator, and sends the RISC-V binary code to the Target simulator to populate the simulated processor's instruction memory with the program. Once the fesvr finishes sending the test code, the fesvr resets the Target processor and the Target processor begins execution at a fixed address in the program. Figure 2 of (John Wawrzynek, 2016).

对这个实验平台，稍作修改，将原来糅杂在一起的处理器 Chisel 源代码，测试程序代码和测试工具链 (如 fesvr 和生成 C++ 模拟器的库文件) 分离成两个相对独立的私有仓库，用 git 来管理，使得代码的编写工作变得更有条理。一级的文件目录结构如图4.2所示。另外，对原来处理器源码的组织形式做了略微的调整，使得更符合 JAVA 的代码组织结构风格。对于 Makefile 编译文件，也做了必要的修改，使之符合修改后的文件目录组织。

运行的流程大致为：

(a) 在 `chiwen` 目录下运行 sbt，Scala 的构建工具或者说是集成化的编译器 (Chisel 的编译器实际上是在 sbt 中加入前端的 Chisel 语法树分析，和后端的 Chisel 到 Verilog 的转化规则)。然后选择处理器的工程名，这样 sbt 就会产生一个处理器的 C++ 精确到周期的描述。为了简化这一步骤，如图4.2a编写了一个脚本，执行命令`terminal -> ./setup.sh <processor_name>` 即可。这个脚本最后会把生成的 Verilog 代码和 Chisel 的中间层表示 Firrtl 格式文件放到 `mizhi` 的 `emulator` 目录中以处理器命名的目录下。



(a) (b)

图 4.2 毕业设计工程仓库。(a) 处理器 Chisel 源代码仓库。(b) 测试软件工具链和测试程序仓库。

Figure 4.2 the repositories of the design. (a) The git repository which contains processors Chisel source, (b) The git repository which contains verification tool chain and test programs.

(b) 编译文件 Makefile 会用 verilator 软件将第一步得到的生成代码再生成和处理器对应的 C++ 模拟器。

(c) 编译文件 Makefile 会调用 RISC-V 的前端服务器 (fesvr) 在文件系统中为第二步得到的 C++ 二进制模拟器打开一个 socket，然后将 RISC-V 的二进制文件文件传输到目标处理器去执行，见图4.1.

(d) 在 `mizhi` 仓库里，已经提供了现成的测试程序包括指令功能测试和嵌入式 benchmark 程序性能测试，其中的 benchmark 程序集如表4.1所示。所有测试程序位于 `riscv- tests` 目录中；自定义的测试程序，可以放至 test 目录中，见图4.2b。程序的编译用 RV32I 的 GCC 工具链标准编译。具体的命令封装在 Makefile 中，只需要输入命令`cd riscv-tests/isa -> make` 生成指令测试二进制文件或者`cd riscv-tests/benchmarks -> make` 生成性能测试的二进制文件。相应的，让目标处理器去仿真模拟执行只需要输入命令`make run-asm-tests` 或者`make run-bmarks-test` 就能运行指令功能测试或者 benchmark 性能测试。

(e) 最后如果通过，会在终端给出 PASSED 字样，否则就是 FAILED 来给出处理器的验证结果。大致的机理是 Makefile 会先用 Spike，RISC-V 的 ISA 级模拟器跑出一遍 trace，然后比对处理器的输出结果。前端服务器 (fesvr) 也会通过 debugIO 通过访存来检测处理器的状态是否达到预期。不过存在着自动检测在 FAILED 时却给出 PASSED 的少见情况。针对这种情况，用 python 编写了一个脚本来比对待测试处理器产生的 trace 和 Spike 跑出的标准 trace 每一个周期的有效写回寄存器堆结果，或者也可以和 sodor 已经提供的几个简单处理器打印的 trace 做对比。这样就能严格保证处理器执行已有程序的正确性。

4.2 设计方案

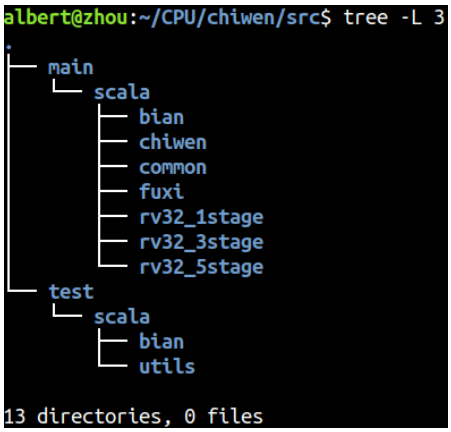


图 4.3 目前为止已有的六款 RISC-V RV32I 处理器核。其中 3 款处理器 CHIWEN(螭吻), FUXI，BIAN(狴犴) 架构分别是自主设计的单发射五级静态流水线，双发射五级静态流水线, 和双发射乱序流水线。另外 3 款 rv32_1stage, rv32_3stage, rv32_5stage 是riscv-sodor提供的单周期、单发射三级静态流水线和单发射五级静态流水线处理器。common 子目录包含了处理器共有的模块。

Figure 4.3 The six Microprocessors based on RISC-V RV32I isa so far. There are three processors are self designed, including CHIWEN(single-issue, five-stage pipeline processor), FUXI(dual-isse, five-stage pipeline processor) and BIAN(two-wide superscalar processor with out-of-order). Other three simple processors are provided by riscv-sodor, with single cycle, three-stage and five stage single-issue architecture separately. `common` subdirectory contains common modules shared with all processors.

若直接进行双发射乱序的设计，会因为没有很好的设计着力点而变得步履维艰。所以制定出一套详细的从易到难循序渐进的方案显得非常重要。同时，Chisel 对于自己来说是一门全新的语言，也需要用一些简单的任务来充分地熟悉。

在 riscv-sodor 中已经提供的有三款简单的处理器，微结构分别是单周期 (rv32_1stage)、单发射三级静态流水线 (rv32_1stage) 和单发射五级静态流水线 (rv32_5stage)。将它们拷贝至自己的仓库的文代码目录中，参见图4.3。这些简单的处理器除去共有的 CSR 特权寄存器逻辑和仿真内存逻辑代码之外，核心部分均仅仅只有 500 多行代码，代码简洁易读，尤其是其译码部分。处理器采用的结构是经典的数据通路和指令通路。但是事实上这样的设计模式不好，两个通路之间是强耦合的。在阅读 rv32_5stage 的过程中，也发现数据通路和控制通路的两个文件中相当有一部分代码重复冗余。为了可拓展性和控制复杂性的考虑，第一步是调整处理器的设计模式，采用章节3.1中论及的前后端模型。修改的同时，

逐渐掌握了 Chisel 描述电路的常用写法。

表 4.1 benchmark 性能测试程序说明。除了 coremark 是自己添加的，其余的都是riscv-tests已经提供的

Table 4.1 The explanation of the set of benchmarks provided by the riscv-tests repository except for coremark

名称	功能解释	备注
coremark	A synthetic embedded integer benchmark	合成的嵌入式整数程序
dhrystone	A synthetic embedded integer benchmark.	合成的嵌入式整数程序
median	Performs a 1D three element median filter.	一维的中位数查找
multiply	A software implementation of multiply.	乘法的软件实现
qsort	Sorts an array of integers using quick sort.	快速排序
rsort	Sorts an array of integers using radix sort.	基数排序
towers	Solves the Towers of Hanoi puzzle recursively.	汉诺塔
vvadd	Sums two arrays and writes into a third array.	数组向量加法

但是 CHIWEN 不是对 rv32_5stage 简单重写。事实上，rv32_5stage 运行 benchmark 的统计结果如图4.4所示，IPC 普遍偏低。

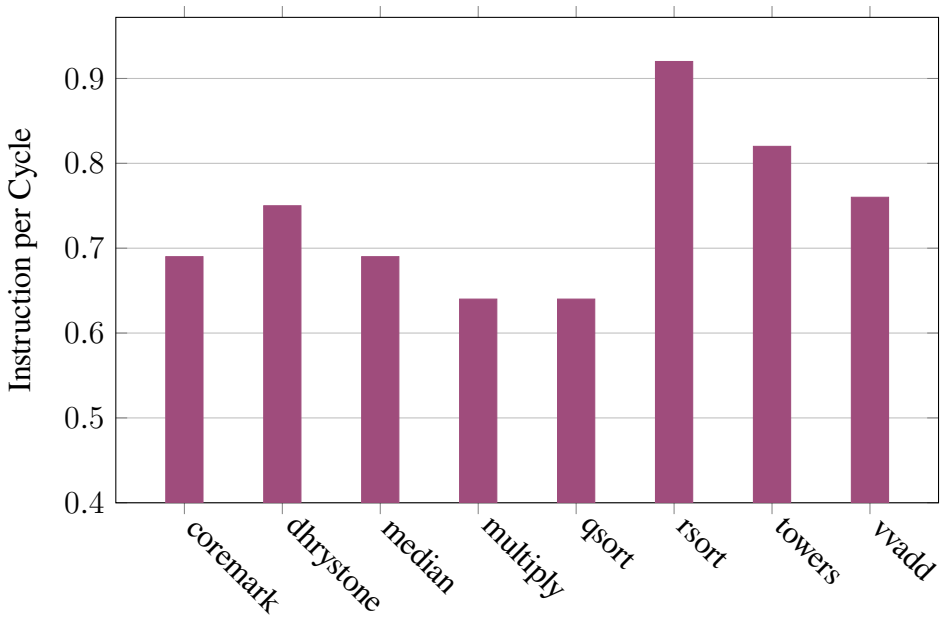


图 4.4 rv32_5stage 运行 benchmark 程序集的 IPC。

Figure 4.4 the IPC of rv32_5stage executing benchmark programs.

导致性能偏低的原因在于 rv32_5stage 没有转移猜测的机制，如果跳转指令的结果还没计算出来，只会顺序往后取指。RISC-V 没有延迟槽，而且在 rv32_5stage

的设计中，第三级计算出跳转目标和方向还会锁存一个周期反馈到取指部件进行重定向取指。所以猜错一个分支跳转指令就会损失 2 个周期，这样 IPC 就不会高。所以第二个任务就是在 CHIWEN 的前端加入转移预测器，参考借鉴的是文献Celio et al. (2017) 中的 BTB，BPD 和 RAS 设计，并做了一定简化。

在 riscv-sodor 平台上取指和访存都是没有延迟的，默认是同步的逻辑。为了模拟真实的外围内存系统，写了一个带有随机性的延迟的中间桥，置于处理器核和模拟的同步内存之间，并且采用的是 AXI 的标准接口。接着构建了容量为 16KB 的 icache 来缓存指令，提高在有延迟的取指环境中的处理器性能。但是出于简化设计的考虑，并没有构建 dcache，访存没有延迟，为默认的同步时序。综上，基于前后端模型的五级流水线架构，辅以转移预测器和 icache 构成了 CHIWEN 处理器。

得益于前后端模式的强大优势，从单发射的 CHIWEN 过渡到超标量宽度为 2 的双发射五级静态流水线 FUXI 的过程非常平稳，代码基本上能够复用，不用做大的修改，开发的时间和调试的时间都大大缩短了。大致的修改如下：

(a) 后端的修改较为简单，把 CHIWEN 的后端用 Chisel 里 Vec 的写法重新写一遍，构建出两条并行的后端执行流水线。然后解决两条并行指令之间的如写后写、写后读、同时为跳转指令、同时为访存指令、同时为特权态指令等冲突。具体的解决做法是：将后一条指令停住一个周期做成串行执行。不光写起来简单，调试的时候也不会出现很多的 bug。

(b) 前端的修改是重点，也是难点，修改取指单元，参见章节3.7.1。同时，icache 和分支预测单元都要做出修改，将输入 PC 信号 (if/pc 级) 和输出指令信号 (icache) 或预测信号 (分支预测单元) 从单端口调整为双端口。

有了 CHIWEN 和 FUXI 这两代处理器核微结构的演进，才过渡到双发射乱序处理器 BIAN 的设计调试中。由于 FUXI 和 BIAN 都是超标量宽度为 2 的处理器，所以 BIAN 的前端可以完全复用 FUXI 的，而不需要单独重新设计。解决了前端的问题，最后只需集中力量进行后端乱序发射的逻辑设计即可。前后端强大的优势再次得到了很好的体现。而且可以看出，高性能双发射乱序处理器的成功设计离不开循序渐进的正确设计路线。

4.3 调试手段

首先是模块级的调试验证。Verilog 常用的做法是给待测模块写激励测试。这个激励测试相当于更大一级的测试激励模块，里面实例化了待测模块并构建有

时钟时序和测试输入，然后通过仿真器手动查看输出信号波形是否符合预期。缺点在于，激励模块的编写过程十分麻烦，不仅是因为语句的粒度太小描述能力差，而且入如果波形出现和预期不同的结果，也不能马上得出待测模块有漏洞的结论，因为也有可能是激励模块有漏洞。因此，Chisel 语言自封闭的设计了一套基于 Chisel 和 Scala 编写验证模块的方式。它封装了 `chisel3.iotesters.PeekPokeTester` 的基类，编写待测模块的激励模块首先要继承这个基类，待测模块输入端的赋值调用 poke 语句，输出端的验证可以用 expect 语句和预期值进行自动比对，也可以打印出来。同时，因为 Scala 是脚本语言，所以复杂的且带随机性的激励测试构造是便捷的。另外，最为关键的一点是，`chisel3.iotesters.PeekPokeTester` 基类隐藏了时序的构造，因为电路的何时复位，时钟的频率多少，何时在上升沿一般情况下不会对处理器电路产生任何影响。电路经过一个时钟周期的逻辑在 Chisel 的测试代码中可以用简单的 `step(1)` 实现 (括号中的 1 表示一个周期)。语义上，在 `step(1)` 之后的代码逻辑会发生在 `step(1)` 之前的代码逻辑一个周期之后。Chisel 的测试代码的组织形式和 JAVA 保持一致，见图4.3中的 test 目录，这样方便管理和测试，特别是在 IDE 的编辑环境中。

但是上述调试方法不再适用于将各个模块整合起来的整个系统。整个系统的激励输入还是要依赖于 riscv-sodor 的测试环境 (见图4.1)。那么输出端又如何去验证呢? 输出端核心的信号有哪些? 最核心的是每一周期的寄存器堆写回信号和 store 指令的内存写回信息。如果这两类信号都符合预期，就认为处理器的输出是正确的。这个预期可以由 RISC-V 的模拟器 Spike 或者 QEMU 得到，也可以由经过验证的开源处理器核如 rv32_5stage 得到。调试的流程为：**step1**: 写回信息打印出来以文本的形式存储 ⇒**step2**: 用自己编写的一个自动对比的 python 脚本进行文本操作自动比对 ⇒**step3**: 定位到开始出现不同的第一个周期 ⇒**step4**: 在源代码中加入在这个周期之前一个窗口内打印与产生差错有关联的更多变量的代码 ⇒**step5**: 重新编译执行 ⇒**step6**: 通过分析文本中的调试信息来定位到源代码中的 bug。

这样调试方式虽然原始，如果需要重新加入打印信号，就需要重新编译执行。但是就其速度而言，因为不用做整个系统全信号的仿真模拟，仅仅是打印出是每一周期的写回信息以及一段窗口内的详细的相关变量信号而已，就算加上重新编译和执行的时间，也比波形图仿真更快。另外，由于设计的时候先采用了较为彻底的模块级验证，等到整合为完整系统时，bug 数量大大降低了，而且基本集中在各个模块交互的逻辑上。同时，文本的优势在于可以灵活地编写一些文

本操作的脚本做自动化的分析，这恰恰是波形所不具备的。比如需要得到某一周期执行级具体的指令，波形图上只能显示一个 32 位的数值，转化需要人工手动。但是文本就可以用一个脚本来自动转化。还有例如分析分支跳转预测正确率的过程中需要统计精确到不同跳转指令的误预测率时或者某一周期窗口内的跳转行为以及预测行为时，文本 + 脚本分析的手段都要明显优于波形图调试。

正是采用了这种调试手段，从最开始的调试方式摸索阶段，用两周时间调通了带转移预测器的 CHIWEN 处理器，并且预测器的行为符合预期；再到用一周时间调通 FUXI 的前端，加上三天调通 FUXI 的后端；最后到用两周的时间 (一周进行模块级的调试验证，一周进行整合系统的调试验证) 调通了整合到 FUXI 前端的 BIAN 的后端。调试的时间的缩短使得在微结构设计上的时间变得相对充裕。

4.4 结果分析

表4.2是各个程序的大致动态指令数，以及运行的时候执行的指令数。之所以是大致，是因为由于一些与串口打印交互的轮循和依赖执行周期的汇编代码的存在，不同的处理器执行各个程序的行为会出现偏差，指令数也就不一样。可以看出 coremark, dhrystone, qsort 和 rsort 的规模比较大，达到了十万量级；而 median, multiply, towers, vvadd 的规模较小，为万量级。

表 4.2 benchmark 各个程序大致动态指令数。

Table 4.2 The approximate number of dynamic instructions of each program in the benchmark.

benchmark	指令数
coremark	~780K
dhrystone	~350K
median	~15K
multiply	~49K
qsort	~230K
rsort	~370K
towers	~18K
vvadd	~11K

图4.5分析了八个程序不同类型指令的占比情况。可以看出，coremark, median, qsort, multiply 四个程序分支跳转指令 (Branch+Jal+Jalr) 占比很大，均在 30% 以上，

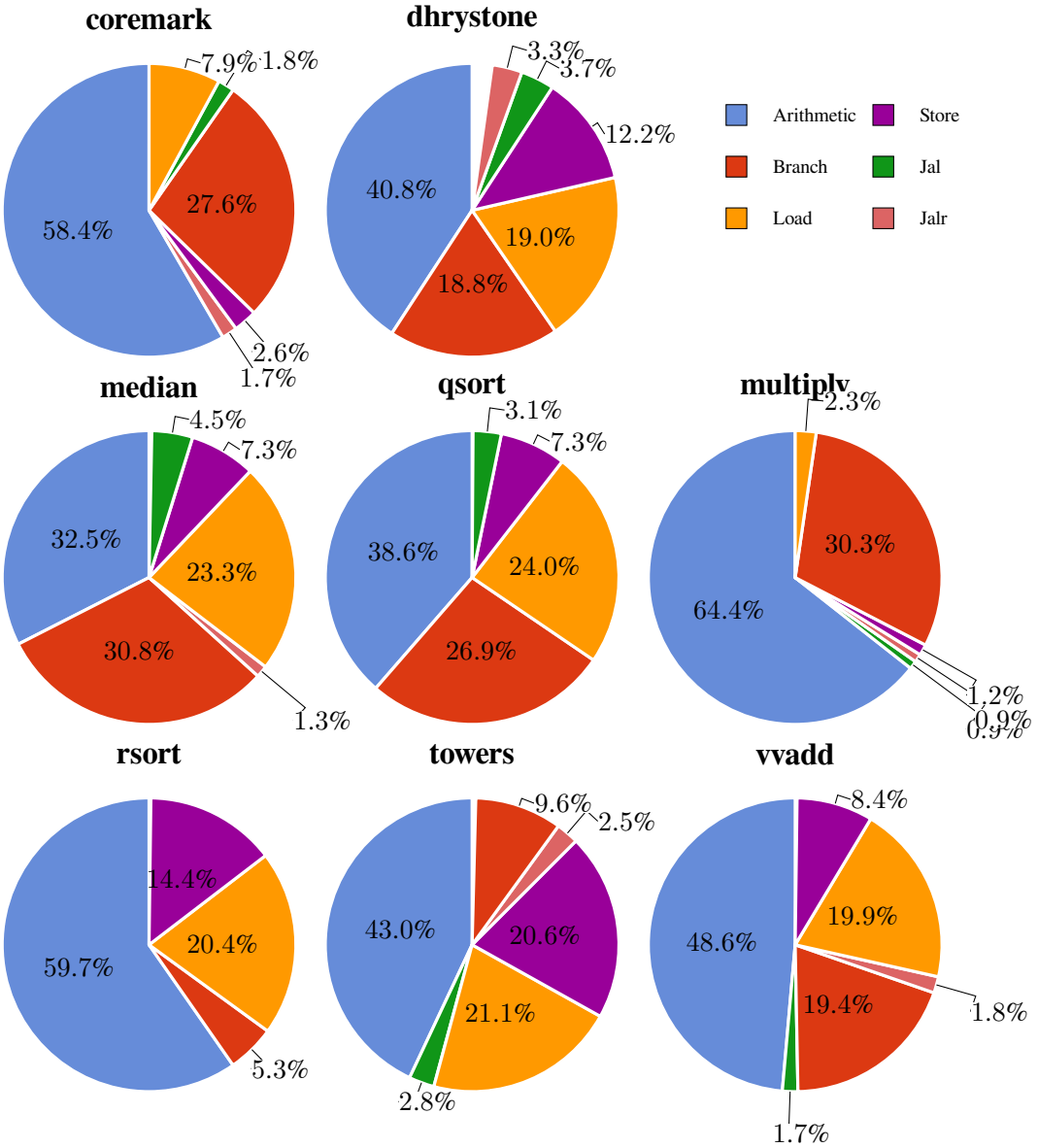


图 4.5 benchmark 中各个程序不同指令的占比。Arithmetic: 算数指令；Branch: 分支指令；Load: 加载指令；Store: 存储指令；Jal: 立即数为偏移量的跳转链接指令；Jalr: 寄存器为偏移量的跳转链接指令。

Figure 4.5 The ratios of different instruction types in every programs among the benchmark.

相当于 3 条指令中就有一条分支跳针指令，非常密集。而 dhrystone, rsort, towers, vvadd 这四个程序转移跳转指令占比较为合理，属于一般意义上的正常程序。

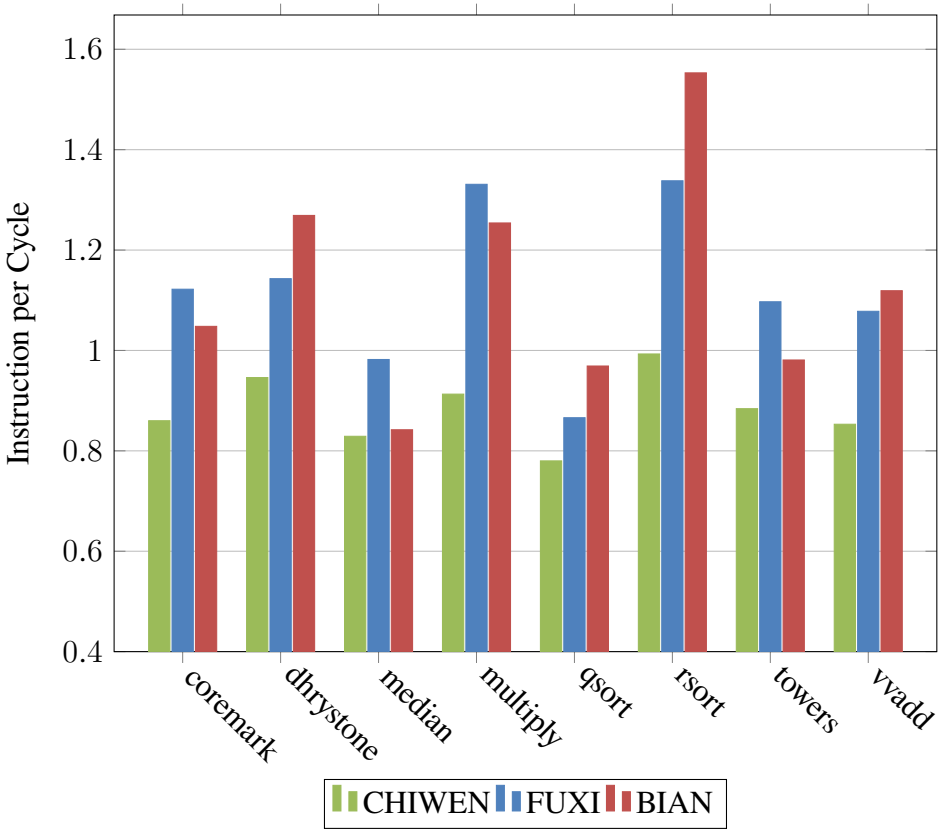


图 4.6 三款处理器执行各个程序的每周期指令数。

Figure 4.6 the Instruction per Cycle of each program over three processors.

如图4.6，对比了 3 款处理器执行程序的性能 (用 IPC 来衡量)。可以看到，最终结果并没有像预期一样，乱序双发射的 BIAN 执行各个程序都完美地优于顺序流水线架构下的 CHIWEN 和 FUXI。BIAN 对比单发射的 CHIWEN，性能都是有所提高的。但是对比 FUXI，却是互有胜负。dhystrone, qsort, rsort, vvadd 处理器 BIAN 效率更高，而 coremark, median, multiply, towers 却是双发射顺序流水线的 FUXI 效率更高。

分析原因，首先非常直观的一个观察是在 BIAN 劣于 FUXI 的 4 个程序中，跳转分支指令占比大于 30% 的占了 3 席：coremark, median, multiply. 为了更为客观的观察分支跳转指令对处理器性能的影响，图4.7专门统计了 3 款处理器运行各个程序的转移预测正确率，BIAN 的预测正确率在 82% 以下的一共有 4 个程序：coremark, median, qsort, towers. 其中，和 BIAN 性能劣于 FUXI 的 4 个程序重合了 3 个。coremard 和 median 更是跌到 80% 以下。所以 coremark 和 median 在没有访存延迟的环境下，乱序结构不如顺序结构性能高的现象基本上可以归因

于转移跳转指令占比高的同时转移预测正确率又很低。

表 4.3 三款处理器执行各个程序的每周期指令数。

Table 4.3 the Instruction per Cycle of each program over three processors.

benchmark	CHIWEN	FUXI	BIAN
coremark	0.860	1.122	1.048
dhrystone	0.946	1.143	1.269
median	0.829	0.982	0.842
multiply	0.913	1.331	1.254
qsort	0.780	0.866	0.969
rsort	0.993	1.338	1.553
towers	0.884	1.097	0.981
vvadd	0.853	1.078	1.119

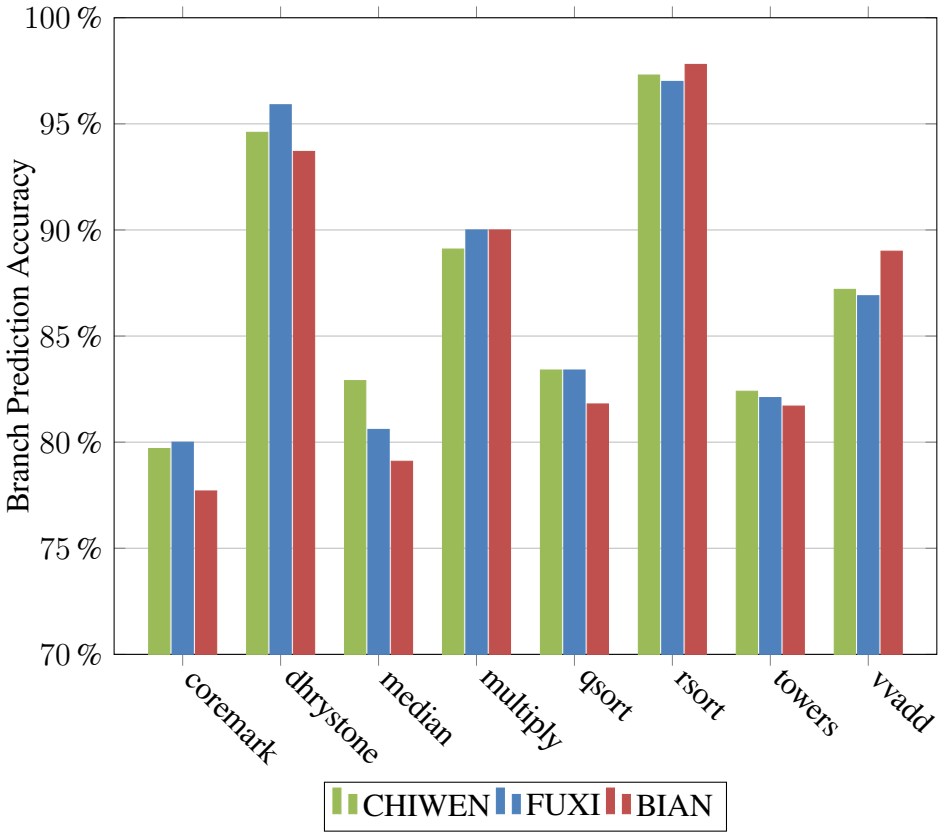


图 4.7 三款处理器执行各个程序的转移预测正确率。

Figure 4.7 The Branch Prediction Accuracy of each program over three processors.

从图4.7中还有一个有趣的现象是不同处理器执行同一程序的转移预测率是不同的。但是这三款处理器均是采用相同的数据结构和预测策略。产生这种现

表 4.4 三款处理器执行各个程序的转移预测正确率。

Table 4.4 The Branch Prediction Accuracy of each program over three processors.

benchmark	CHIWEN	FUXI	BIAN
coremark	79.7%	80.0%	77.7%
dhrystone	94.6%	95.9%	93.7%
median	82.9%	80.6%	79.1%
multiply	89.1%	90.0%	90.0%
qsort	83.4%	83.4%	81.8%
rsort	97.3%	97.0%	97.8%
towers	82.4%	82.1%	81.7%
vvadd	87.2%	86.9%	89.0%

象的原因在于由于在流水线上，转移预测的数据结构的更新有反馈周期 (解释见3.9.2) 的延迟，不同微结构的反馈周期不同，有些预测的结果就是预测单元在结构未做更新时产生的，所以会对预测结果产生小范围的波动效应。

表 4.5 三款处理器只采用两位饱和计数器的 BTB 表预测技术，执行各个程序的每周期指令数。

Table 4.5 the instruction per cycle of each program for three processors which only use 2-bit BTB structure.

benchmark	CHIWEN	FUXI	BIAN
coremark	0.856	1.110	1.051
dhrystone	0.909	1.083	1.174
median	0.781	0.927	0.773
multiply	0.908	1.321	1.242
qsort	0.779	0.864	0.975
rsort	0.992	1.337	1.548
towers	0.864	1.065	0.938
vvadd	0.838	1.053	1.088

高分支跳转指令占比，低分支预测率只不过是乱序处理器性能不高的表面原因。考虑到 FUXI 和 BIAN 的转移正确预测率基本上保持一致，仅有小范围的波动，乱序处理器在执行这类程序性能不高的本质原因在于：乱序结构对转移预测正确率的敏感度比顺序要高，或者说转移预测正确率对乱序的微结构影响更大。为了量化地说明这一点，在 3 款处理器中同时关闭了 gshare 算法预测技术

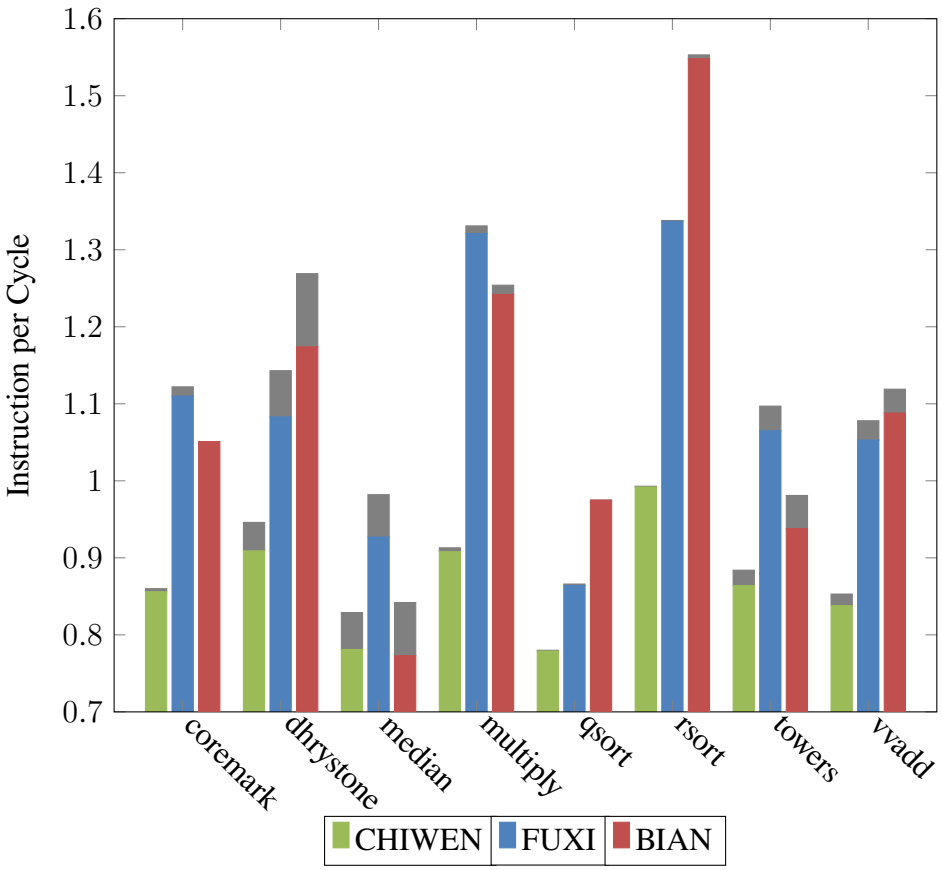


图 4.8 三款处理器只采用两位饱和计数器的 BTB 表预测技术，执行各个程序的每周期指令数。灰色部分是相比用原先预测策略的减少量。

Figure 4.8 the instruction per cycle of each program for three processors which only use 2-bit BTB structure. The gray is the decrement compared with the original prediction strategy.

表 4.6 三款处理器只采用两位饱和计数器的 BTB 表预测技术，执行各个程序的转移预测正确率。

Table 4.6 The Branch Prediction Accuracy of each program over three processors which only use 2-bit BTB structure.

benchmark	CHIWEN	FUXI	BIAN
coremark	78.7%	79.0%	77.8%
dhrystone	86.1%	87.4%	88.2%
median	72.6%	72.9%	70.3%
multiply	87.9%	88.2%	88.1%
qsort	83.2%	83.3%	81.7%
rsort	96.1%	96.3%	96.8%
towers	73.8%	75.0%	72.5%
vvadd	81.4%	82.7%	85.3%

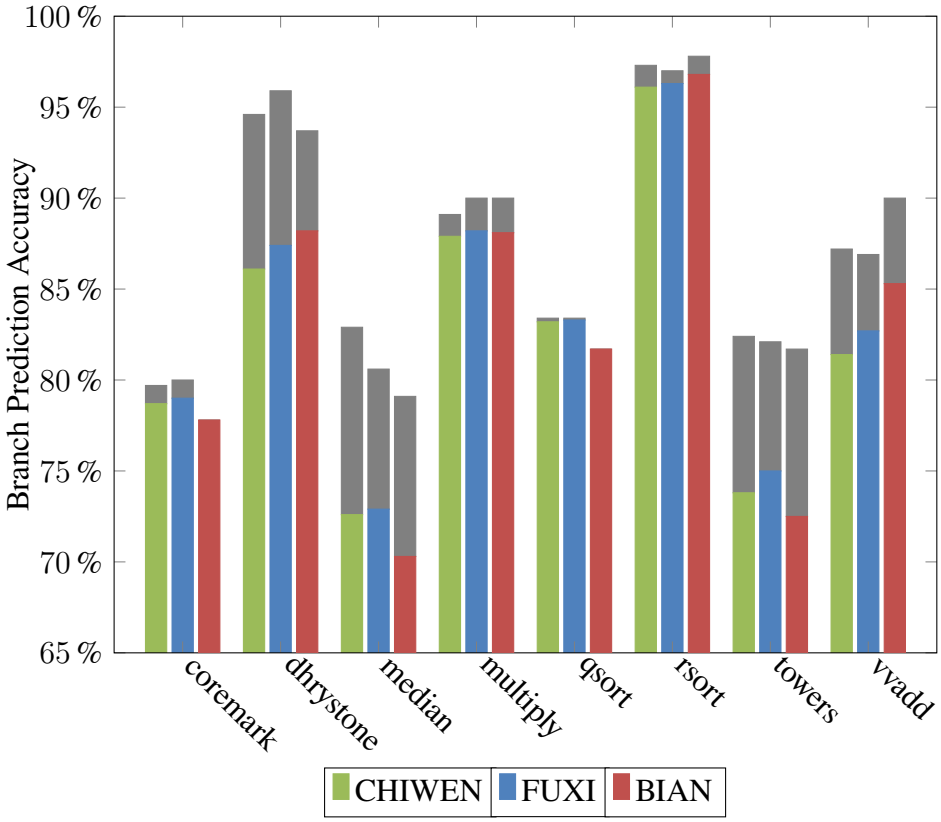


图 4.9 三款处理器只采用两位饱和计数器的 BTB 表预测技术，执行各个程序的转移预测正确率。灰色部分是相比用原先预测策略的减少量。

Figure 4.9 The Branch Prediction Accuracy of each program over three processors which only use 2-bit BTB structure. The gray is the decrement compared with the original prediction strategy.

以及锦标赛仲裁的逻辑，也即分支预测仅仅依靠两位饱和计数器。统计得到的 IPC 和跳转预测率分别如图4.8和图4.9。其中，灰色部分代表的是与原来没有关闭 gshare 的锦标赛逻辑相比数值上减少的部分。可以看出在 BIAN 和 FUXI 分支预测正确率的下降量相当时，BIAN 的 IPC 比 FUXI 的下降得更多。证实了乱序结构对转移预测正确率更敏感的结论。

图4.9还反映出了关闭 gshare 功能和锦标赛逻辑后，对 coremark 和 qsort 这两个程序的预测正确率影响不大。在 BIAN 中，coremark 和 qsort 甚至出现了轻微的负增长波动。这同样是一个非常有意思的现象。目前采用的 10-bit 全局历史和 gshsare 算法对于这两个程序影响不大说明了这两个程序对于短全局历史的规律性不强。这在代码逻辑上能够得到解释。在实验平台上，coremark 一共循环执行了 20 次，其中分支逻辑占比最多的是字符串比较；qsort 是快速排序，而快速排序又是著名的随机化算法。无论是字符串比较，还是快速排序，其跳转都是非常随机的，基本上没有什么规律 (工业界对于 coremark 预测率的提高是依靠循环上千次的大循环加上长全局历史得到的，区别于实验中的小循环和短全局历史)，所以进一步的提高预测正确率从而提高乱序结构执行 coremark 和 qsort 的性能是困难的。

量化地分析 coremark 和 qsort 的跳转行为。如图4.10所示，两条跳转指令的数量就占到所有跳转指令数量的 2/3，而 PC 0x80005920 对应的分支指令误预测数量更是占到误预测总数的超过六成。另外，如表4.7所示，对频率最高的几条跳转指令，两位饱和计数器加全局历史 gshare 的锦标赛预测策略误预测率都很高。qsort 程序没有 coremark 那么极端，不过从图4.11和表4.8出现频率最高的几条分支指令 PC，预测做的同样不好。

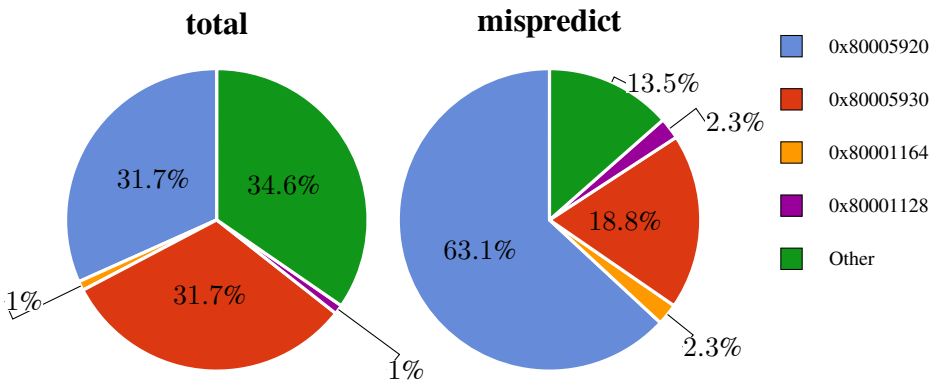


图 4.10 coremark 动态执行分支指令分布 (左)。coremark 动态执行误预测分支指令分布 (右)。

Figure 4.10 The distribution of total branch instructions over executing coremark(left). The distribution of mispredict branch instructions over executing coremark(right).

表 4.7 coremark 在不同 PC(对应分支指令) 的误预测率。

Table 4.7 The misprediction rate on different PC of coremark.

PC	误预测率
0x80005920	44.2%
0x80005930	13.2%
0x80001164	50.4%
0x80001128	49.3%
the others	8.6%

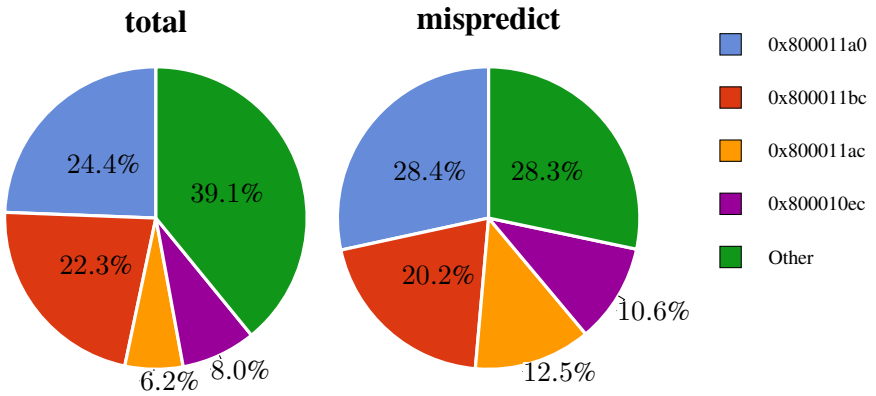


图 4.11 qsort 动态执行分支指令分布 (左)。qsort 动态执行误预测分支指令分布 (右)。

Figure 4.11 The distribution of total branch instructions over executing qsort(left). The distribution of mispredict branch instructions over executing qsort(right).

表 4.8 qsort 在不同 PC(对应分支指令) 的误预测率。

Table 4.8 The misprediction rate on different PC of qsort.

PC	误预测率
0x800011a0	21.3%
0x800011bc	16.6%
0x800011ac	37.0%
0x800010ec	24.2%
the others	13.2%

通过各个处理器执行效率 IPC 和转移预测正确率的比较，还反映出了三个异常的现象。

(1) qsort 和 towers 两个程序，转移预测正确率在同一水平上 (见图4.7)，因此从 BIAN 的 IPC 结果 (见图4.6) 中可以看出，通过乱序调度，两个程序如预期一样达到基本相同的 IPC。但是在顺序流水线 FUXI 的 IPC 结果中，两者却相差很大。

(2) multiply 程序的转移预测正确率在 90% 左右 (见图4.7)，不及 dhrystone 的预测正确率，因此在 BIAN 的 IPC 结果 (见图4.6) 中，multiply 的 IPC 不及 dhrystone。但是 FUXI 的 multiply 效率却反常的高。

(3) 程序在 BIAN 中的 IPC 基本上与转移预测正确率成正比。但是 coremark 和 qsort、towers 之间却出现了反常 — coremark 的转移预测正确率比 qsort 和 towers 都低，但是 IPC 却比后两者高。

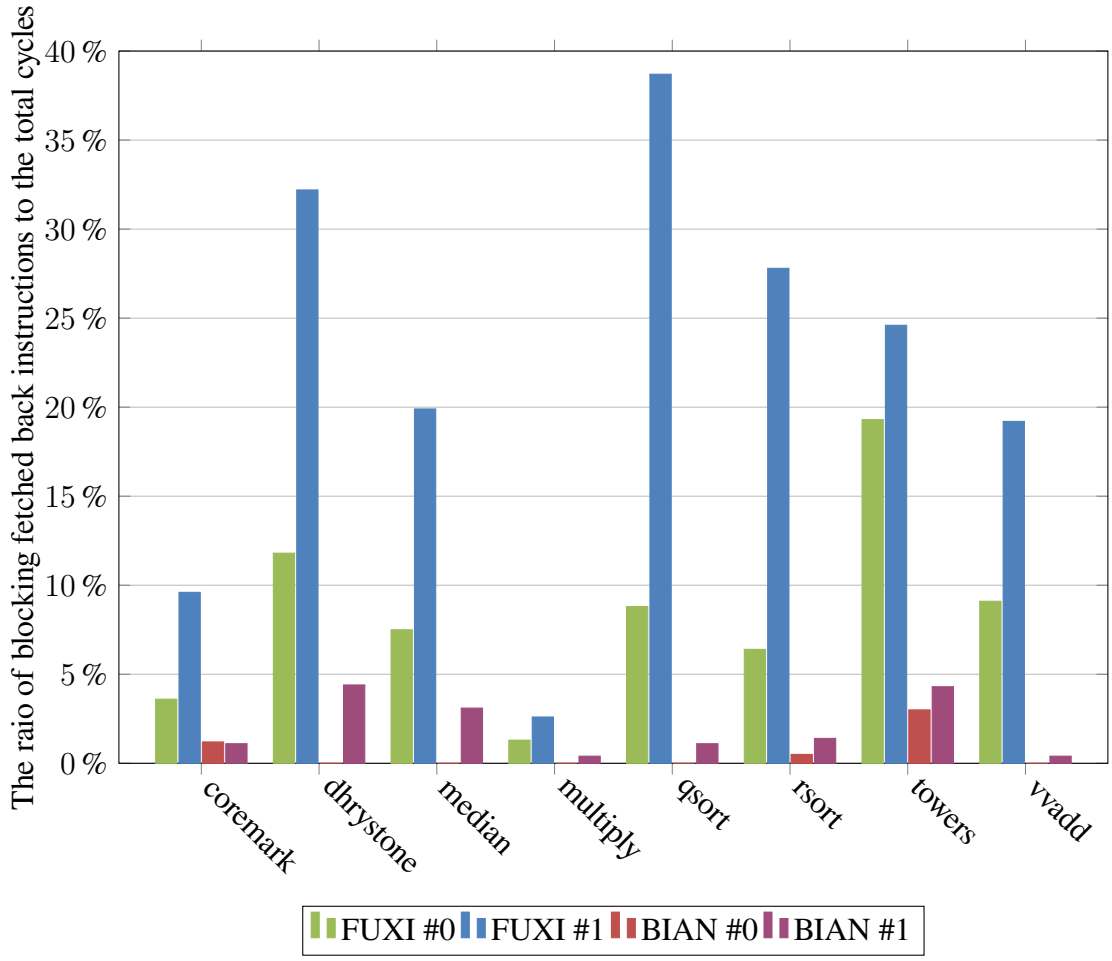


图 4.12 FUXI 和 BIAN 前端两条指令各被阻塞的周期占执行总周期数比例。

Figure 4.12 The raio of blocking fetched back instructions to the total cycles over CHIWEN and FUXI.

表 4.9 FUXI 和 BIAN 前端两条指令各被阻塞的周期占执行总周期数比例。

Table 4.9 The raio of blocking fetched back instructions to the total cycles over CHIWEN and FUXI.

benchmark	FUXI#0	FUXI#1	BIAN#0	BIAN#1
coremark	3.6%	9.6%	1.2%	1.1%
dhrystone	11.8%	32.2%	0.0%	4.4%
median	7.5%	19.9%	0.0%	3.1%
multiply	1.3%	2.6%	0.0%	0.4%
qsort	8.8%	38.7%	0.0%	1.1%
rsort	6.4%	27.8%	0.5%	1.4%
towers	19.3%	24.6%	3.0%	4.3%
vvadd	9.1%	19.2%	0.0%	0.4%

对这些异常的现象进行量化分析，根本的原因还是在于前后端的交互效率上。如图4.12，统计了 FUXI(顺序) 和 BIAN(乱序) 后端阻塞从前端取回两条指令的占比。因为如果第一条指令阻塞了，第二条指令也会跟着阻塞，所以两路并行指令的被阻塞占比差可以很好地量化程序的串行度，差越大代表着程序的串行度越高。柱形图4.12中，FUXI 两路指令在 qsort 一栏的高度差是最大的，所以 qsort 的串行度最高。而第二条指令 (FUXI#1) 一旦被阻塞，FUXI 的前端也会被阻塞，所以第二条指令被阻塞占比上的差距使得 qsort 和 towers 在 FUXI 的 IPC 结果中相差很大。

第二个现象，multiply 程序的 IPC 在 FUXI 中反常的高也是因为前端的供指因素。在图4.12中，multiply 两条并行执行的被阻塞率都极低，而且差距很小，程序的可并行度很高。所以虽然转移预测率只有 90%，但是在 FUXI 中的 IPC 却可以媲美 rsort(rsort 虽然转移预测正确率很高，但是其串行度也较高)。

第三个现象，由于在 BIAN 中三个程序指令的被阻塞率都很低，所以问题不出在阻塞上。那么问题是不是出在前端的取指带宽上 (每周期平均能够取回的指令数)。如图4.13所示，如果按照非对齐的取指，三个程序的取指效率差不多。如果按照对齐取指，那么反而 coremark 的取指效率是最低的。目前 BIAN 的设计中采用的是对齐取指。最后剩下的一种合理的解释是，相比 coremark 较高的并行度，qsort 的并行度太低；而 towers 由于访存指令的占比太大，导致后端对前端的阻塞虽然也在 5% 以下，但是也比 coremark 要高出 2 ~ 3 倍，同时也是由于访存指令占比大，导致了 load-to-use 的分支指令的反馈周期变大，预测正确率的

对 towers 的影响就比 coremark 更大。

最后分析双宽度的前端是否需要做成非对齐取指的方式。虽然非对齐的方式在带宽上较高，但是却增加了逻辑和电路的复杂性。对齐与非对齐差距较大的如 coremark 和 median。由图4.5，两者均是高分支占比的程序；而且这些高分支占比的程序由图4.7，又是预测正确率很低的。大量的指令被取回又被取消掉，导致最后的 IPC 很低。如图4.6，coremark 接近 1.05，median 只有 0.84，采用对齐取指方式，都会白白浪费很大的供指带宽。做成更激进的非对齐取指，没有意义。

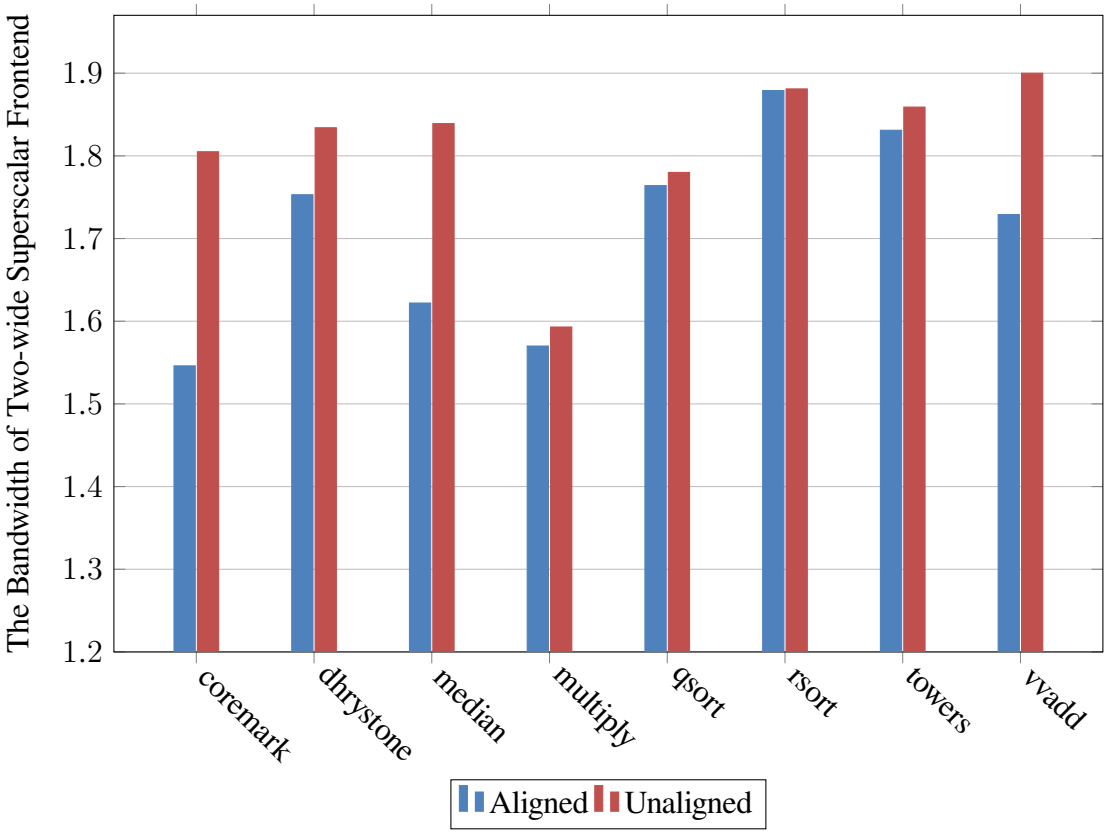


图 4.13 对齐取指和非对其取指的前端供指带宽 (没有取指延迟)。

Figure 4.13 The Bandwidth of aligned and unaligned two-wide superscalar frontend(without instruction fetching delay).

4.5 小结

整个实验平台采用有 20 个周期左右的随机取指延迟配以 16KB 的 icache，以及无访存延迟的环境。通过量化的方法，分析和对比了影响乱序和顺序性能的几个要素，有转移预测正确率，动态执行的各类指令占比，前后端之间的指令阻塞比例，程序的串行度和前端的取指带宽。得出的结论是：转移预测正确率对乱序

处理器性能的影响极大；程序串行度也有不小的影响；取指的带宽相对于转移预测争取率和后端的执行效率是足够的，并不会成为性能瓶颈。

第 5 章　结论与展望

通过第4章的分析，相比于单发射五级静态流水线再到双发射五级静态流水线，双发射乱序处理器后端对前端的阻塞大大降低了，但是目前分支预测正确率不高的几个程序运行效率都没有达到预期的提升。

目前已经有很多具体的工作正在实施或者筹划中，但是很遗憾地来不及展示在本文中，如：

(1) icache 真正做到四路组相连的结构，每一路 4KB。

(2) 参考文献Kessler (1999); Yeager (1996)，构建高效的内部内存系统 (Inner Memory System).

(3) 处理器对外支持整套 AXI 接口，同时仿真平台也能够支持整套以 AXI 总线交互的外设，如带延迟的内存。乱序的引入很大程度上就是为了通过硬件的调度来隐藏访存延迟。一旦访存带有延迟，支持四路组相连的 dcache，BIAN 相比于顺序流水的 CHIWEN 和 FUXI 效率会高不少。会从研究的角度来看，实验结果更漂亮；同时从实际应用的角度来看，也更加具有实用性。

(4) 参考章节1.2.1中介绍 RISC-V 的特权部分，使 BIAN 处理器支持 MMU 和虚拟内存管理。

(5) 尝试去移植在 Rocket 和 BOOM 中的仿真外设，真正做到系统的模拟仿真。

(6) 在设计过程中，被诟病最多的一点是 rename 级的基于 RAM 形式的重命名表，加上旁路技术读寄存器堆的设计。这里，已经构思好了两部分的解决方案。第一，将重命名表从 RAM 形式改为 CAM 形式，CAM 形式在后来的讨论中被证实是时序更好，面积更小的方案。第二，取消在 rename 阶段的旁路，见框图5.1。

(7) 正在尝试采用更为复杂的预测策略和数据结构如文献Celio (2018) 提到的 2bc-table、TAGE 预测器 (见图5.2)；文献Kessler (1999) 提到的 Local History Table.

(8) 在目前 BIAN 的设计中，后端各队列项数的设置都是凭着设计直觉的。正考虑修改各个队列的参数，通过仿真结果来找到各个队列参数最优解。而且出于之前针对 rename 级提出的两个解决方案带来的时序改善，会考虑多做几项分支跳转的状态回溯备份，并增加乱序执行的指令条数和物理寄存器堆项数。在不

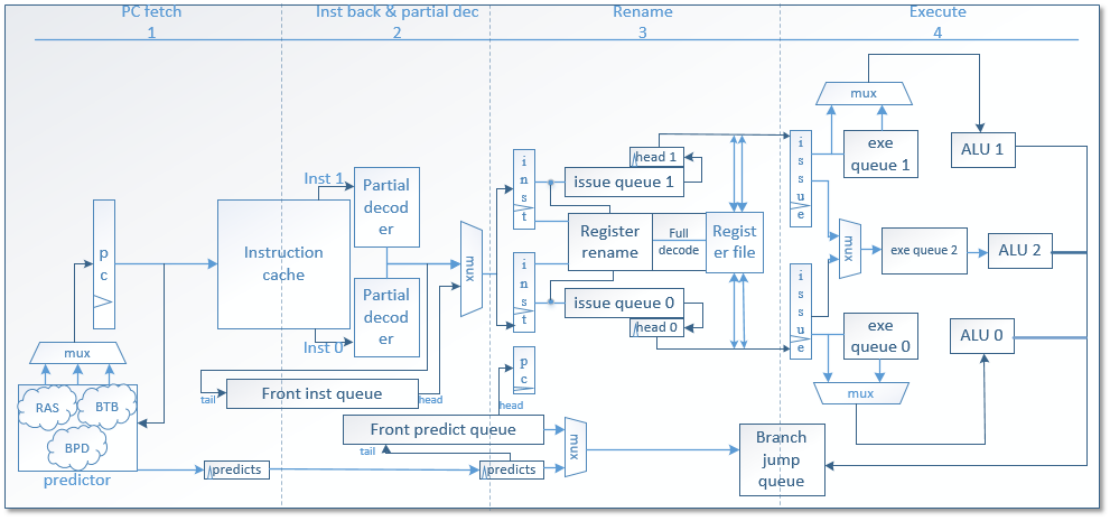


图 5.1 BIAN 处理器整体框图，不包括访存单元并取消重命名阶段的旁路

Figure 5.1 Block diagram of BIAN processor (not include load-store unit and remove the bypass in Rename Stage.)

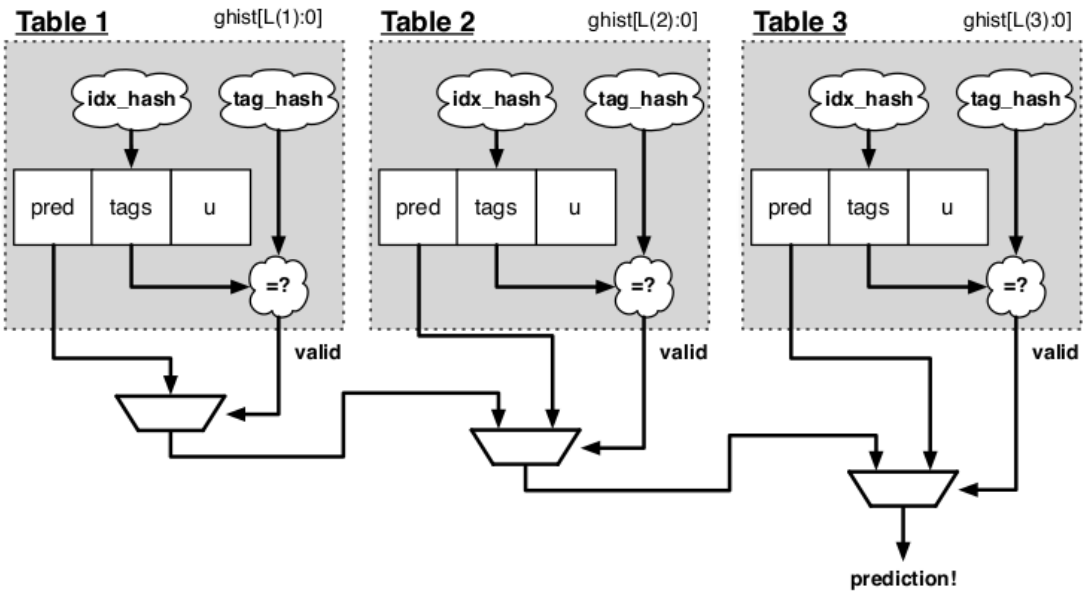


图 5.2 TAGE 预测器。请求地址和全局历史会被输入各个表的所以哈希和标签位哈希函数。然后每个表都会提供它们的预测结果，最后选择拥有最长历史的表。表存储的历史按几何级数的逐表递增。(Celio, 2018) 图 4.1.

Figure 5.2 The TAGE predictor. The requesting address and the global history are fed into each table’s index hash and tag hash functions. Each table provides its own prediction (or no prediction) and the table with the longest history wins. Each table has geometrically more history bits than the previous table. Figure 4.1 of (Celio, 2018).

断提升处理器性能的同时，也对乱序处理器的设计空间有更深入的探索。

除了一些具体的工作安排，还有一些更为遥远的展望，如：

(1) 通过完善的系统仿真验证，并移植成功 Rocket 外设和浮点运算部件。最后，能够在 FPGA 上运行一整套软件栈 (如 Linux 操作系统和 GCC 编译器)，并能够进行 SPEC2000/2006 的性能测试，在性能上不输 RISC-V 的 BOOM 和 ARM 的 Cortex A53，A57 系列。

(2) 实现双核的并行。

(3) 目前国内自主研发主流的 ISA 还是 MIPS，肯定会将目前的基于 RISC-V 的双发射乱序处理器原型机修改为基于 MIPS 的版本。并且希望能够通过不懈的努力，有朝一日能够将自己的设计真正的流片出来，在实际中得到应用。

参考文献

Ahi A, Chen Y C, Conrad R, et al., 1995. R10000 superscalar microprocessor[J].

Andrew Waterman K A, 2017a. The risc-v instruction set manual volume ii: Privileged architecture privileged architecture version 1.10 document version 1.10[M]. SiFive Inc., CS Division, EECS Department, University of California, Berkeley.

Andrew Waterman K A, 2017b. The risc-v instruction set manual volume i: User-level isa document version 2.2[M]. SiFive Inc., CS Division, EECS Department, University of California, Berkeley.

Celio C, 2018. A highly productive implementation of an out-of-order processor generator: number UCB/EECS-2018-151[D/OL]. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-151.html.

Celio C, Chiu P F, Nikolic B, et al., 2017. Boom v2: an open-source out-of-order risc-v core: number UCB/EECS-2017-157[R/OL]. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-157.html.

Chris Celio A G B K, Jerry Zhao, 2018. Riscv-boom documentation[M]. Department of Electrical Engineering and Computer Science University of California, Berkeley.

John Wawrzynek M M, 2016. Cs 152 laboratory exercise 1[M]. Department of Electrical Engineering and Computer Science University of California, Berkeley.

Jonathan Bachrach J W, Krste Asanović, 2017. Chisel 3.0 tutorial (beta)[R]. EECS Department, UC Berkeley.

Kessler R E, 1999. The alpha 21264 microprocessor[J/OL]. IEEE Micro, 19(2):24-36. DOI: 10.1109/40.755465.

Patterson D, Waterman A, 2017. The risc-v reader: An open architecture atlas[M]. 1st ed. Strawberry Canyon.

Yeager K C, 1996. The mips r10000 superscalar microprocessor[J/OL]. IEEE Micro, 16(2):28-41. DOI: 10.1109/40.491460.

致　谢

感谢吴瑞阳学长根据自己丰富的工程经验，和我具体而激烈地探讨了开始阶段处理器设计中，无论是在电路延迟上，还是在执行效率上存在的问题以及改进的方案。同时，他也给了我很多微结构设计上的有用的意见和电路编写中非常实用的小技巧。

感谢郑雅文学姐提供的 coremark 源代码，并用 RISC-V GCC 编译成功；目前，郑雅文学姐正在帮助构建带有访存延迟和支持 AXI 多通道，interleaving，outstanding 等特性的仿真环境。

同时还要感谢杨灿学姐关于电路时序和龙芯处理器微结构设计的细心答疑，以及李策学长关于转移预测策略的答疑。

最后感觉胡伟武老师对于这个选题的肯定和支持。