本科生毕业论文（设计）开题报告

论文题目 多发射乱序 CPU 的 RTL 级实现与验证

学生姓名 周盈坤 学号 2015K8009929023

指导教师 胡伟武 职称 研究员

导师单位 中国科学院计算所

学位类别 工学学士

专　业 计算机科学与技术

学院（系） 计算机科学与技术学院

填表日期 2019-01-07

中国科学院大学制

填 表 说 明

1. 本表内容须真实、完整、准确。

2. “学位类别”名称：填写理学学士、工学学士等。

3. “专业”名称：填写专业全称。

4. “课题类型”：选填论文、设计。

5. “课题性质”：选填基础研究、应用研究、综合研究及其它。

6. “课题来源”从下列项目中选填：

□ 973、863 项目　　□ 国家社科规划、基金项目

□ 教育部人文、社会科学研究项目　□ 国家自然科学基金项目

□ 中央、国家各部门项目　　□ 省（自治区、直辖市）项目

□ 国际合作研究项目　　□ 与港、澳、台合作研究项目

□ 企、事业单位委托项目　　□ 外资项目

□ 学校自选项目　　□ 国防项目

□ 非立项　　□ 其他

7. 该表填写完毕后，须请指导教师审核，并签署意见。

8. 本表格不够可自行扩页。

毕业论文（设计）开题报告

题目	多发射乱序 CPU 的 RTL 级实现与验证		
课题类型	设计	课题性质	基础研究
课题来源	学校自选项目		

选题的背景及意义：

体系结构迎来了开源之光,这是一个大的背景。RISC-V 开源的项目越来越多,有微处理器设计的如 Rocket 和 BOOM;有更高级的刻画电路的语言的如 Chisel;也有针对 RISC-V 的调试工具如 Spike,以及上层已经移植好的基础软件栈如 Linux,GCC 和 LLVM。大大降低了独立设计出一款高性能处理器以及在上面跑系统的难度与门槛。所以计划本科毕业设计顺着这股大的潮流先基于 64 位 RISC-V 自行实现一款高性能的处理器。同时编写调试的过程也是在学习开源资料中优秀的思想的过程,也是不断提高自我修养的过程。事成之后再移植到 MIPS 的 ISA 上,就可以做到同样架构对比两个 ISA 之间的优劣势了,这样更加客观。

但是本科阶段没有发论文的打算,一是体系结构这个领域十年磨一剑,仅凭本科毕设估计很难有成色;二是在本科阶段参与和体系结构、操作系统、编译原理有关的基础性工作,打好以后研究的基础,所以借设计一款高性能的 CPU 并能够运行系统来提高自己软硬件的素养就是这个选题的意义所在。

国内外本学科领域的发展现状与趋势：

1 语言与表示

1.1 Verilog

Verilog 是目前流行通用的硬件描述语言，表现能力强，语言设计参考的是 C 语言而且最开始设计出来是用作仿真的，因而有很多用于仿真的语法。但是真正可以用于综合电路的语言并不多，比较常规的有组合逻辑的 assign 语句，时序逻辑的时钟上升沿触发的同步电路逻辑 always@(posedge clk) begin ... end. 辅以 generate 的写法，避免相同的逻辑代码重复冗余。

1.2 Scala

Scala 是一门非常有野心的语言。它并不是从头开始构建的语言，而是依附在 JAVA 的平台之上，能够复用 JAVA 已有的 library。这本身就是一个很好的考虑。甚至可以说 Scala 就是扩展在 JAVA 之上的脚本语言。它同时兼顾面向对象与函数化编程，并且它还是静态类型的语言，所以不同于 python。而做到脚本语言的同时可以使静态类型编译的关键一点就是类型编译时推导，只需要在关键的地方声明类型即可，比如函数或者方法定义时的传入参数列表。

Verilog 和 Scala，风牛马不相及。然后从本质上来看，所有的语言都服务于一个目的——描述逻辑。而现在这个需求更加的具体化，那就是描述电路的逻辑。所以再往下思考，电路的逻辑需要什么语言要素来刻画。

1. 模块化。功能电路的设计，CPU 的设计是模块化的。
2. 函数化。功能电路关注点在于输入 input，输出 ouput，这一点和函数很像。

分析得出这两个特征，会发现 Scala 的面向对象和函数化编程是多么契合电路的逻辑设计。如此一来 Scala 就有描述电路的可能。除了支持的语言特性吻合电路设计的需求外，Scala 作为一种强类型的语言，同样是电路刻画需要的 (Verilog 也可以看做是强类型的)。而且 Scala 是脚本语言，同样有优势，首先脚本语言简洁，其次电路的描述并没有很大的计算量，这恰恰就是脚本语言所擅长的。

为了代码的简洁与复用，语言的高级化是在所难免的。通用的编程语言从 C 到 C++ 再到 Java 再到如今的 pyhton。然而硬件的描述语言却一直停留在最初的 Verilog，究其原因，电路描述高级语言化的障碍有两点：

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。

1.3 chisel

世上无难事，只怕有心人。Chisel 的出现，将 Verilog 和 Scala 连接起来，将上述的可能变为了现实。那么 Chisel 是如何 (初步) 打消上述的两个障碍和顾虑的呢？那就要看 Chisel 是怎么进行抽象的。

1. 组合电路的对应于 Verilog 中的 wire，赋值用 assign 语句，也可以直接在定义的时候赋值。而 Chisel 首先用的就是两类的数据类型来描述，分别是 UInt 和 Bool。Bool 只是为了强调变量是 1 bit 的代表真假的布尔逻辑变量。而 wire 类型变量的赋值有两种形式

   - 初始定义的 = 运算符

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

     如上并不是真正的赋值，而是类型的申明，而且如果 wire 的 width 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。

   - 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 (本质上 Scala 将其抽象为对象的方法，这是一个非常高明的抽象) 来进行重赋值。

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

     这个时候 Chisel 编译器就可以从 right hand side 表达式中推导出 pc 的宽度。 [5]

2. 时序电路对应于 Verilog 中的 reg，赋值需要用到 always 语句。而在 Chisel 对其进行了一段抽象，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面——input 和 output。而 output 可以理解为 input 信号延迟一拍的副本。所以严格来讲，Reg 是有类型的，就是 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 reset 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。 [5]

3. := 运算符将右手表达式代表的组合逻辑连线到左手变量代表的目的节点上。如果涉及到寄存器，用变量 x 来表示，那么 x 若出现在右手边，用到的就是 output 端；如果出现在左手边，用的就是 input 端。 [5]

4. 对于 wire 类型和 reg 类型的变量，Chisel 统一抽象为了 node。所以整个电路图就是由 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 是可以对于设计中出现的组合环进行报错，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 [5]。

5. 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手法构建更为大型，更为抽象的数据结构，比如 memory。

6. 同时，在 Verilog 对于模块的刻画用了 module 的写法，而在 Chisel 中则是用户自定义 class 来 extends 叫做 Module 的 super class 来刻画电路模块，并且对于 port 接口，定义了一个 IO 的类。如下例 [5]：

```
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，类似于 C 里面的 *Structure*。这里直接 new 出来一个匿名的结构体，然后作为参数传入 IO 的构造方法中，最后将其赋值给 IO。这种写法相比于 verilog 里最大的好处是什么？那就是在 Module 的内部的逻辑中，Chisel 的代码更加清晰，是不是端口的引用或者赋值一目了然，因为凡是带 io. 的就是端口。这样增加了代码的可读性。

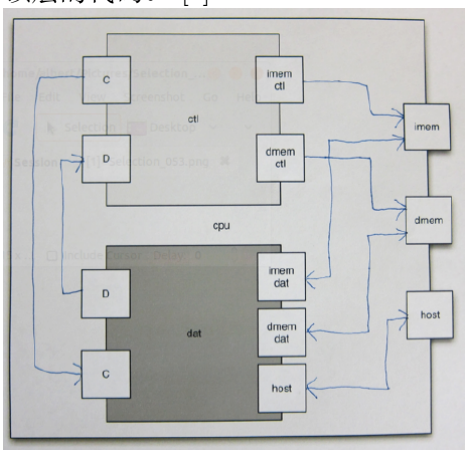
经过上面的分析，可以发现其实电路还是那个电路，Chisel 抽象掉了次要的东西，保留除了真正核心的内容。如果尚有缺点，也就是目前屏蔽了 clock 和 reset，默认为统一时钟同步复位，还不支持异步电路。这个在 Chisel 的文档里也给出了解释是：纵然有交叉时钟的设计需求，但是现代设计方法中也是在每一个同步的 “岛” 中开发与验证 [5]

那么 Chisel 的真正威力体现在哪里？先看两个例子：

```
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
}
```

这个例子展现了面向对象和函数化编程以及 Parameterized Functions (类似于 C++ 的 template) 的强大力量，首先是通过面向对象的继承来充分复用已有的代码逻辑，然后由于不同功能的 Filter 利用 λ-函数作为参数传入，来充分复用 Filter 共通的逻辑部分。

第二个例子是给出一个逻辑框图，让我们来编写顶层的代码。 [5]



作为铺垫，比如现在已有如下 Chisel 代码：

```
class RomIo extends Bundle {
  val isVal = Input(Bool())
  val raddr = Input(UInt(32.W))
  val rdata = Output(UInt(32.W))
}
class RamIo extends RomIo {
  val isWr = Input(Bool())
  val wdata = Input(UInt(32.W))
}

class CpathIo extends Bundle {
  val imem = RomIo().flip()
  val dmem = RamIo().flip()
  ...
}

class Cpath extends Module {
  val io = IO(new CpathIo())
  ...
  io.imem.isVal := ...
  io.dmem.isVal := ...
  io.dmem.isWr := ...
  ...
}
class Dpath extends Module {
  val io = IO(new DpathIo())
  ...
  io.imem.raddr := ...
  io.dmem.raddr := ...
  io.dmem.wdata := ...
  ...
}
```

这里用的也是面向对象的技巧来尽可能的复用代码。这样如果要编写一个将这些模块连接起来的顶层模块，Verilog 肯定是一大堆的端口，然后是一大堆的信号，但是用 Chisel 语言，就可以简化为：

```
class Cpu extends Module {
  val io = IO(new CpuIo())
  val c = Module(new CtlPath())
  val d = Module(new DatPath())
  c.io.ctl <> d.io.ctl
  c.io.dat <> d.io.dat
  c.io.imem <> io.imem
  d.io.imem <> io.imem
  c.io.dmem <> io.dmem
  d.io.dmem <> io.dmem
  d.io.host <> io.host
}
```

这种写法在 Chisel 的术语里叫做 *Bulk Connections*。

那么上面两个例子所体现出的语言特性有没有实际工程上的作用，当然有。一个最为典型的例子就是 AXI 接口，首先 AXI 是有五个通道，而且非常的规整。所以抽取出五个通道的共通之处，写出一个基类。然后对五个通道如果有多余的端口可以扩展这个基类，只需要添加多余的端口就行了。更妙的是，不同组件通过 AXI 总线共连时，用 bulk connection 的写法，每连一对总线，只需要写一行代码。

Chisel 的威力还不止于此，再举一个例子，如果要实例化多个同样的 module，Chisel 可以用 for loop，但是 Verilog 也可以做到，那就是 generate 的写法。但是如果仔细一看，Chisel 的 for loop 翻译为 Verilog 可不是用 generate 的写法，而是有几个实例化几个。一开始觉得这个是笨方法，但是后来发现这是个很明智的选择。首先这些都是 Chisel 编译器干的事情，一点都没有增加人的工作负担，不用复制粘贴。其次这种写法的好处就是更加通用。这个更加通用体现在万一需

求是要实例化多个略有差别的不同配置的 module，比如前面的第一个例子要实例化 100 个不同功能的 Filter，generate 的写法显然不能胜任，但是 Chisel 编译的手法就能在 for loop 里面实现配置。然后编译好的 Verilog 文件真的就有 100 个不同配置的 Filter。

1.3.1 Chisel test

Chisel 沿袭了 Verilog 的传统，同一个语言可以用来设计电路也可以用来仿真电路，这样就不需要在两种语言之间做切换。而且由于 Chisel 承袭了 Scala，而 Scala 承袭了 Java，所以 test 设计的哲学思想也就自然顺承 Java。从发展的角度来看，一开始的 C&C++ 没有严格的代码组织格式，test 的设计也是如此，而且调试主要以真实的应用场景加上单步调试的 GDB 为主。到后来的 Java 对于代码的组织格式有了严格的要求，同时 test 的设计同样跟着代码的组织格式对每个模块进行了用例的 assert 测试。从而提出了一种比较系统的测试方法。引进的原因在于，用 GDB 跟着一个大系统单步调试往往效率是低下的，而把大系统拆分成一个个小的零件 (其实也不用拆分，因为代码的组织格式就是按照一个个小的零件组织的)，然后用一些包含 edge case 的用例来进行 assert 的测试就已经足够。这是一个非常好的想法，以至于后面的 python 更是凭借着解释语言的优势，可以以交互的方式对小零件单独测试，从而减少有这些小零件拼成的大系统的出 bug 数量和概率。所以 Chisel 自然有承袭了这一测试哲学。同时相比于 Verilog 中繁琐的测试代码编写，Chisel 做了相应的简化与核心抽取。最大的一点改变就是不同于 Verilog，在 Chisel 中不需要明确写出 advance 时钟一拍的逻辑，只需要简单写一句 step(1) 就能更新寄存器以及由寄存器驱动的组合逻辑。

1.3.2 sbt 运行环境

和 Scala 配套的 build 编译环境，和 Java 的 maven 类似，但其设计理念对于开发者更加友好。sbt 负责管理编译的依赖关系，比如 Chisel 的库。

1.3.3 Firrtl

Firrtl 是编译系统里面最重要的一层——中间表示层，代表了电路设计与翻译的标准 [7]。事实上，要理解 Chisel 的运行机制，确保生成的电路万无一失，如果直接看由 Chisel 生成的近似于 Netlist 的 Verilog，是 unreadable 的。所以要了解 Chisel 的编译结果，就先要读 Firrtl(Firrtl 还有好几个层次，为的是一步一步将高级的构造语言简化)。另外一点是仿真的效率，因为最后生成的 Verilog 是 slow to simulate，所以 Chisel 快速的仿真是基于 Firrtl 的。

2 ISA

设计一款具体的 CPU，就要考虑到具体基于哪一个 ISA。好在 RISC-V 和 MIPS 非常相似，所以实现了一个，另外一个很多功能部件都能够复用。如果一开始的设计就是尽可能的将微结构和 ISA 接耦合，那么只需要修改少量的逻辑即可。所以计划是先实现 RISC-V，然后在移植到 MIPS。架构是 64 位。

2.1 MIPS

一个最为熟悉但却依旧陌生的 ISA。熟悉的部分是 MIPS 32 中的整数指令部分，以及 CP0 寄存器堆和特权模式处理机制。但是囿于最初并没有考虑到 64 位的需求和嵌入式系统压缩指令长度的需求，导致指令空间设计考虑不周，使得日后出现的 microMIPS 完完全全是不同的指令集和 MIPS32/64 不兼容。MIPS 的 64 位部分、压缩长度指令部分以及浮点部分对我来说是陌生的。

2.2 RISC-V

RISC-V 作为一个 2010 年以后才出现的 ISA，完全有着历史经验的优势，可以充分的借鉴前人设计的优缺点。所以 32 位，64 位，压缩变长指令集都是统一的一个指令集下的不同形式。ISA 的演进已经有 40 多年的历史，就目前而言逐渐趋向于收敛，而且 RISC-V 设计时也考虑到要有极强的可拓展性以便日后之需。同时 RISC-V 的设计模式也和之前所以增量式指令集不同：

不同于几乎所有的先前的 ISAs，RISC-V 是模块化的。核心是基础的 ISA——RV32I，足以运行整个软件栈。RV32I 是被冻结的并且永远不会改变，是稳定的。然后其他的扩展功能的指令以可选的方式可添加。 [4]

那么相比于 MIPS，RISC-V 有什么特点呢?

1. 首先用户态的对比。

   MIPS 有延迟槽这么一条设计。这个被后来证明不是一个好主意，因为其设计的哲学就有问题，延迟槽实际上代表着 ISA 和设计实现的不独立，这是非常糟糕的。比如单发射五级流水有延迟槽就非常 nice，不用停流水或者做转移猜测。但是对于多发射的实现呢? 一条延迟槽不够了。这就是 ISA 和设计实现不独立的后果。在其他细节上，RISC-V 取缔了 HILO，但是 32 位的乘法和除法结果都是 64 位的，怎么解决。RISC-V 选择软件来解决——代码上写两条指令，一条得到低 (商)32 位，一条得到高 (余数)32 位，存在两个不同的寄存器里。那么这个设计有没有设计与 ISA 不独立之嫌呢? 我觉得没有，首先在 32 位的架构下，寄存器都是 32 位的，如果要用到乘除法的结果，也是要通过 mfhi，mflo 指令来操作的。既然如此 ISA 就规定要算出乘除法的完整结果，就是需要两条指令，至于微结构实现要怎么做，那就看微结构的设计者，可以老老实实算两次，也可以一旦发现有连续的两条算高低位的指令，就做一次运算。如果恰好碰到中断之类的，就只好自认倒霉，再算一次了。同时算术指令中也取缔了 overflow 例外，将其移到了软件实现: [1]

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

   在 RISC-V 中较于 MIPS 还有两大亮点：

   - 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。

- RISC-V 支持 pc 相关地址访问。在 RISC-V 中有这样一条指令 AUIPC(add upper immediate to pc)，该指令从指令中的 20 位立即数的基础上第 12 位填充上 0 构成 32 位偏移量与当前的 PC 相加结果哦存到 rd 寄存器中。这样做的好处在于代码数据整体拷贝时，不用改动任何东西就能够运行原来的这段程序，因为相对位置没有改变。

如下是用到 AUIPC 的场合： [1]

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`

2. 其次是特权态的对比

MIPS 的特权态的管理用到的是 Coprocessor 0，负责对虚实地址和例外处理进行管理。记录特权状态的寄存器的地址空间仅有区区 5 位 32 个。但是 RISC-V 却有 12 位的地址空间，最多 4096 个寄存器可用。下图就是 Control and Status Registers(CSR) 的地址空间的分配：[2]

CSR Address [11:10]	CSR Address [9:8]	CSR Address [7:6]	Hex	Use and Accessibility
User CSRs				
00	00	XX	0x000–0x0FF	Standard read/write
01	00	XX	0x400–0x4FF	Standard read/write
10	00	XX	0x800–0x8FF	Non-standard read/write
11	00	00–10	0xC00–0xCBF	Standard read-only
11	00	11	0xCC0–0xCFF	Non-standard read-only
Supervisor CSRs				
00	01	XX	0x100–0x1FF	Standard read/write
01	01	00–10	0x500–0x5BF	Standard read/write
01	01	11	0x5C0–0x5FF	Non-standard read/write
10	01	00–10	0x900–0x9BF	Standard read/write
10	01	11	0x9C0–0x9FF	Non-standard read/write
11	01	00–10	0xD00–0xDBF	Standard read-only
11	01	11	0xDC0–0xDFF	Non-standard read-only
Reserved CSRs				
XX	10	XX	Reserved	
Machine CSRs				
00	11	XX	0x300–0x3FF	Standard read/write
01	11	00–10	0x700–0x79F	Standard read/write
01	11	10	0x7A0–0x7AF	Standard read/write debug CSRs
01	11	10	0x7B0–0x7BF	Debug-mode-only CSRs
01	11	11	0x7C0–0x7FF	Non-standard read/write
10	11	00–10	0xB00–0xBBF	Standard read/write
10	11	11	0xBC0–0xBFF	Non-standard read/write
11	11	00–10	0xF00–0xFBF	Standard read-only
11	11	11	0xFC0–0xFFF	Non-standard read-only

虽然 RISC-V 在操作系统等大型软件上的经验还不如 MIPS 雄厚，而且在手册上也明确说是 *draft*，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是绰绰有余了。

用模型的角度来分析，特权态也不过是在用户态模型之上更高等级的状态 (state)。而这个状态是由寄存器 (Control and Status Registers) 来维护的。同时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，就要看这个 state 以及 state 的变化规不规整。从这个角度而言，RISC-V 作为一个参考了当今软件的运行模式的 ISA，是比 MIPS 更干净的。首先 RISC-V 将处理器的状态分为了 3 个等级 (Mode): [2]

Encoding	Name	Abbreviation
00	User	U
01	Supervisor	S
11	Machine	M

而这 3 个等级的 2 bits 编码均体现在 Privileged Instructions 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 CPU 上，要修改什么等级的寄存器，都有非常干净的规定。

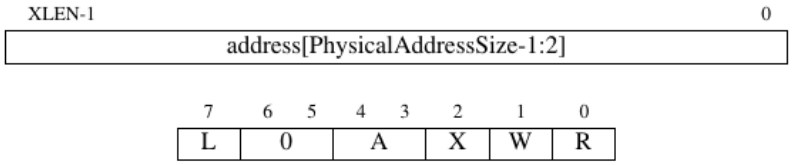
在三个等级中最为基础的就是 Machine Mode。而 M 等级中基础的控制寄存器其实基本上和 MIPS CP0 中的寄存器是一一对应的。如下列举几个： [2]

- `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* 列举了哪些例外处理器会相应，哪些会忽略
- `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
- `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

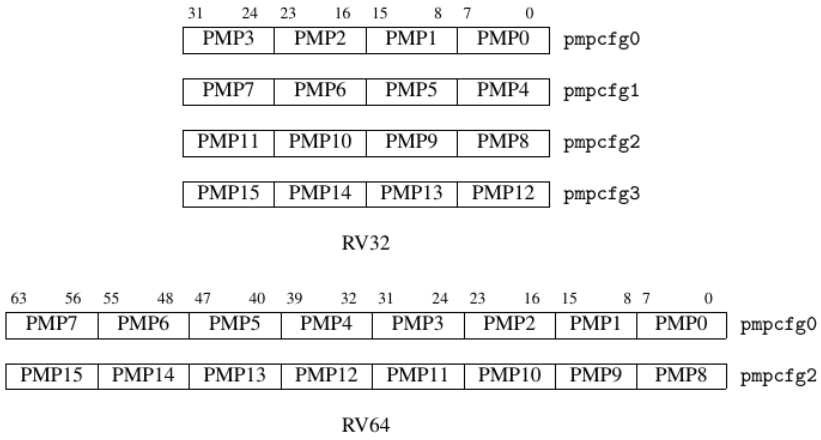
这里比较有特色的寄存器是 mscratch。如果编过 MIPS 的 Linux 内核，就会知道比如 context switch 和 TLB 的例外处理都需要用到 K0 和 K1 寄存器，这两个寄存器就是专门留给操作系统用于临时存储数据用的。虽然这样很高效，但是白白浪费了两个通用寄存器。另外一方面像 context switch 和 TLB 的例外处理是与 M 等级有关的，所以从设计的哲学上来讲也应该归特权级别的寄存器管理而不是通用寄存器。这也是 RISC-V 更为干净的一种体现。

在特权态中最为重要的内存管理，而一个最基本的考虑就是用户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。所以 RISC-V 要想做好，也必须在这方面下功夫。首先 RISC-V 在 M 等级和 U(User) 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 [2] 的上半部分是 PMP 寄存器，一般处理器会实现 8-16 个这样的寄存器。而这个 configuration(上图的下半部分) 也是存储在寄存器里的 (R 代表读，W 代表写，X 代表执行，A 代表 PMP 使能，L 代表 locked)，见下图： [2]



这个 configuration 寄存器授权或者禁止读写执行。当处理器在 U-mode 试图想要取指或者执行 load 和 store 请求时，其地址将会被和 PMP 中的地址进行比较，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 对应的 configuration 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 configuration 里的规定，就会引发例外。 [2]

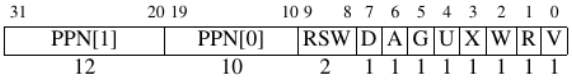
对于嵌入式系统，PMP 机制能够非常有效的对内存实行保护。但是其有两个缺点使之不适合更为复杂的系统和通用计算，分别是：

- 只支持最多 16 个内存区域，不能扩展性
- 这些区域都是连续的，有点像操作系统早期的**段**的概念。所以对内存的片段化支持不好。这也正是段的缺点

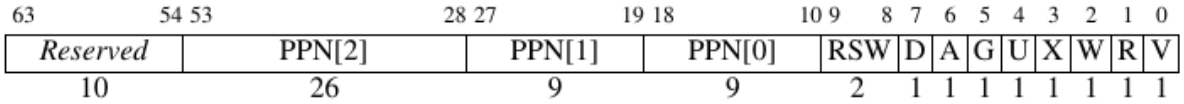
所以 RISC-V 的 ISA 必须支持基于页的虚拟内存机制。而这个特征直接主导了介于 M 等级和 U 等级中间的等级——supervisor mode(S 等级)。

理论上来讲，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 handler。但是大多数的 Unix 系统的例外需要 invoke 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 CPU 将内核切换到 M 等级处理 (或者说 M

等级要重新路由到 S 等级)，[4] 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 bypass 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。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 configuration，但永远不会是 U 等级。RISC-V 的分页方案命名方式为 **SvX**，比如 32 位的虚拟地址就是 Sv32, 其支持 4GiB 的虚拟地址，有两级页表，分别是是 2^{10} 个 4MiB 的页表，然后每个页表又有 2^{10} 个 4KiB 的基页。下图是 Sv32 和 Sv39 的 page-table entry(PTE) [4]

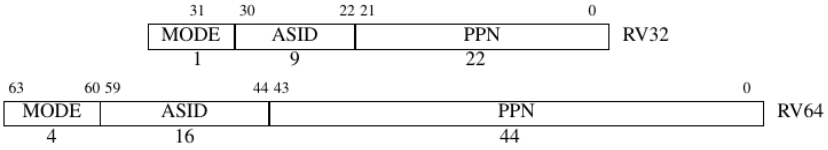


An RV32 Sv32 page-table entry (PTE)



An RV64 Sv39 page-table entry (PTE).

Sv39 是 $2^9 \times 2^9 \times 2^9 \times 2^{12}$ 的模式，所以是 3 级页表。S 等级用 satp(Supervisor Address Translation and Protection) S-mode 的 CSR 来管理分页系统。



RV32		
Value	Name	Description
0	Bare	No translation or protection.
1	Sv32	Page-based 32-bit virtual addressing.

RV64		
Value	Name	Description
0	Bare	No translation or protection.
8	Sv39	Page-based 39-bit virtual addressing.
9	Sv48	Page-based 48-bit virtual addressing.

The encoding of the MODE field in the satp CSR. [2]

在处理器进入 S 等级之前，M 等级首先会写 0 到 satp 寄存器，禁止分页；然后等到进入 S 等级在设置了页表之后，S 级的软件优惠重新写 satp 寄存器最后是虚拟地址转实际地址的过程，注意到如果操作系统修改了 page table，那么就要用 sfence.vma 指令来 flush 相应的 TLB entry。

3 RTL 模型与逻辑

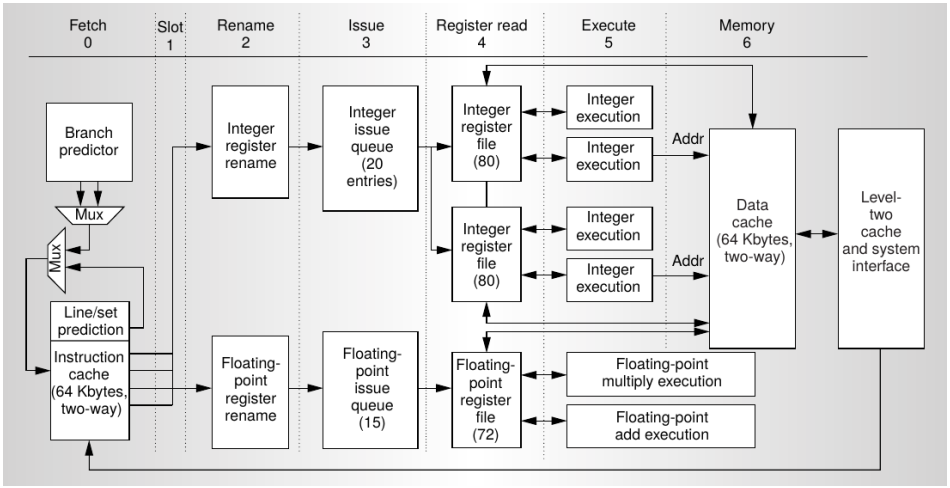
RISC 架构的 CPU，可以看做一个以指令和 load 作为输入，修改内部状态，并通过 store 对外界反馈的器件，这里的内部状态 state 是由程序计数器 PC，通用寄存器，特权态所需的 Control and Status Registers 所构成的。同时所做的运算基本上是二规约的，也即两个操作数变换得到一个结果。在冯诺依曼体系结构下，大致的流程是 CPU 通过取指器件取回来指令并修改下一拍的 PC；同时将取回来的指令进行译码，转化为内部编码。如果是算数指令，等待操作数的准备就绪然后一起发送到运算部件，最后将结果存储入寄存器中，如果是特权指令，修改或者读取相应的 Control and Status Register。下面首先对一些经典的微处理器设计进行分析，然后对关键的组件进行初步的设计考虑。

3.1 可供参考的实例

3.1.1 Alpha 21264

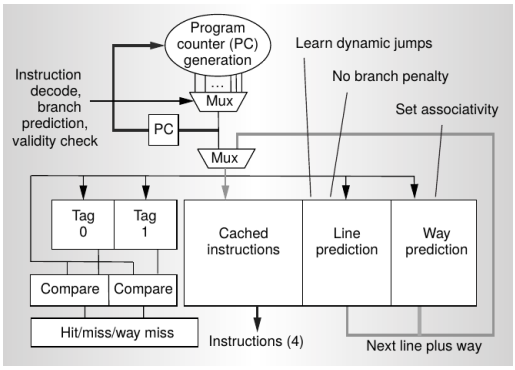
Alpha 21264 是处理器历史上的经典之作。下面就从 *THE ALPHA 21264 MICROPROCESSOR* [6] 这篇技术报告里摘要出来其微结构的设计的 highlight。

下图是 Alpha 21264 的逻辑框图： [6]



1. **取指**。21264 为了提高取指的效率，采用了两种方法，一种是 line and way prediction，另一种是 branch prediction。由于 21264 的 icache 采用的是两路组相连，一种方案是把 line 对应的两个 way 都读出来，然后在 mux 一下，不过这样可能最长路径就

会卡在取回指到译码上，而且可拓展性也不好，如果要实现一个一次取四条指令的CPU，四选一就会更加影响时序。采用预测技术由于猜得准，猜错代价小，所以提高了取指模块的性能。



这幅图 [6] 的大致解读是处理器基于 PC 加上各种预测手段取出下一组指令，与此同时完成上一组取回来指令的有效性检查。训练的 line predictor 对于使用动态链接库的代码有好处。这是因为对于非直接 (subroutine) jump，这个 jump 一定会跳，所以没有 Branch predictor 什么关系，但是在复杂的乱序 pipeline 中要计算出 jump 的地址至少就需要 8 拍，所以 cache 行的预测是非常有必要的，以提供连续的指令流。

而分支猜测同样非常重要，猜错与不猜都是 7 拍的损失。21264 实现了复杂的转移猜测机制，这个机制会动态的选择局部历史和全局历史的结果来预测跳转的方向。[6] 两种方法的结合比任何一种用更大的表的单一方法的准确率都要高。

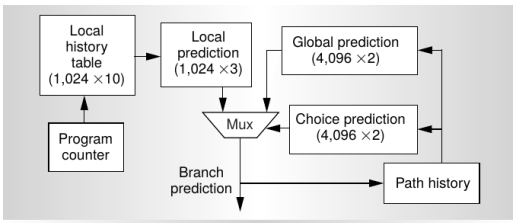


Figure 4. Block diagram of the 21264 tournament branch predictor. The local history prediction path is on the left; the global history prediction path and the chooser (choice prediction) are on the right.

2. **乱序执行**。虽然取回来是 4 条指令，但是在保留站内却是 6 条指令发射，四条整数指令，两条浮点指令。由于中间寄存器 (internal register) 需要指示用户可见的寄存器的对应关系，所以寄存器重命名是一个 content-addressable memory(CAM) 操作。除了各 31 个可见的用户可见的 integer 和 float-point 寄存器，还有额外的 41 个浮点寄存器和 41 个整数寄存器可以用来存放乱序得到的指令的结果。

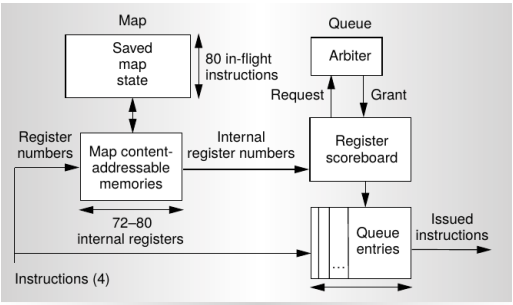


Figure 5. Block diagram of the 21264's map (register rename) and queue stages. The map stage renames programmer-visible register numbers to internal register numbers. The queue stage stores instructions until they are ready to issue. These structures are duplicated for integer and floating-point execution.

发射的细节上，微结构上有一个 20-entry 的 integer queue 和一个 15-entry 的 float-point queue。发射的是那些操作数都已经准备好的指令。如何知道操作数已经准备好了？21264 根据每种指令采用了计分板。同时队列由两个 arbiter 来决定填入新的指令。

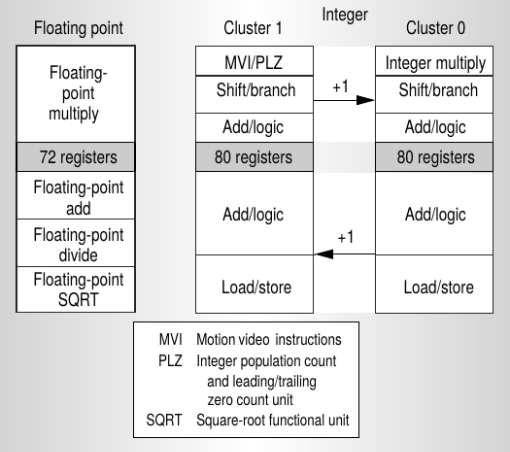


Figure 6. The four integer execution pipes (upper and lower for each of a left and right cluster) and the two floating-point pipes in the 21264, together with the functional units in each.

如上图 [6], alpha21264 的流水线设计使用了 6 条 execution piplines，特别是整数的流水线设计了两个 cluster。设计的基本考虑是要和前面所提到的 issue queue 配套。cluster 使得设计更加简单和快速，即使它会有额外的时钟开销去在各个 cluster 中广播结果。采用动态仲裁哪个 cluster 执行指令。

3. **retire & commit**. 指令结果的 commit 是顺序的，这也意味着两点：

   - 在指令开始执行之后，要等到之前的指令都 retire 完了并且保证自身没有例外发生才可以 retire。
   - 物理寄存器要等到对应的指令提交 retire，才能成为用户可见的。

4. **exception handling**. 21264 实现的是精确例外，也就是说如果老的指令发生了例外，年轻的指令不会改变处理器的 state。但是由于处理器是乱序执行的，所以完全有可能出现年轻的处理器先改变了 internal register 的 state。所以必须有回滚机制，也就是在指令未 retire 之前，要记住原来存储的值。retire 机制维护一个 80 条指令的 inflight window 并且可以一拍内 retire 最多 11 条指令并可以维持每拍 8 条指令的 retire 速率。

5. **memory system & bus interface**. 每拍支持两条从整数流水线出来的内存访问。同时可跟踪 32 条 in-flight loads(32-entry load queue)，32 条 in-flight stores(32-entry store queue) 和 8 条 in-flight(instruction 或者 data) cache miss 请求。dcache 是 64KB 两路组相连的配置。因为每一拍可以支持两次内存访问，所以 dcache 的主频是 CPU 主频的 2 倍。[6]

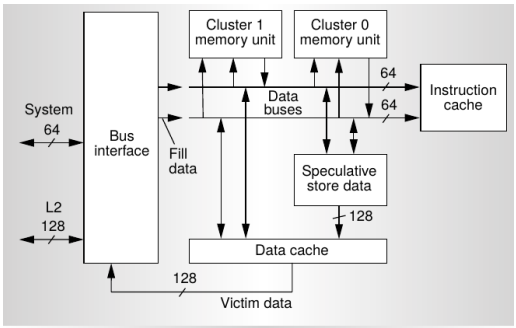


Figure 7. The 21264's internal memory system data paths.

图中两条 64-bit data buses 是 internal memory system 的核心。store 首先会通过数据 bus 将其数据先转移至 store buffer 中。并且 store 可以 forward 数据到后续的 load 操作当 store 的数据依然在数据 buffer 中。实际上这个 store data buffer 起着内存的重命名功能。为了解决 read-after-read, read-after-write, write-after-read 和 write-after-write 的 hazards，21264 使用了双端口的地址 CAMs。 [6]

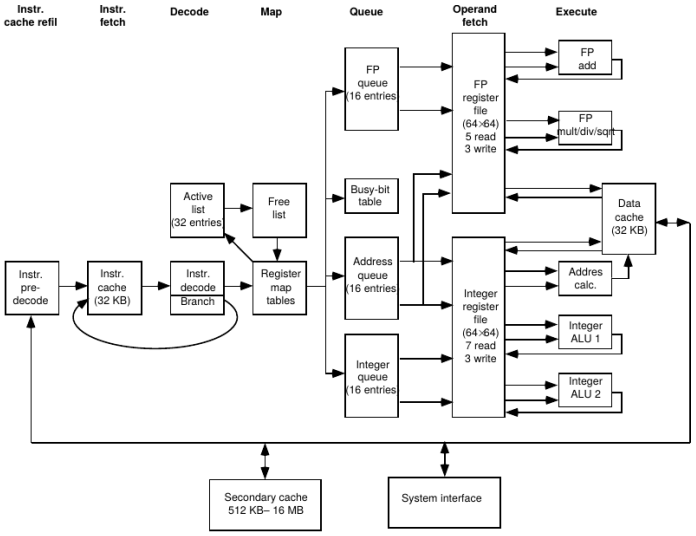
同时，memory system 中还包含了 MAF，简言之就是能够自动合并不同读写相同 block 的请求。

internal memory system 将 MAF 传给 bus interface unit(BIU)，然后由 BIU 来沟通 off-chip 的 L2 cache 以及其系统的其他部

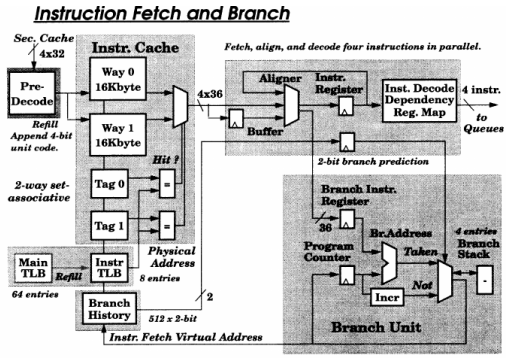
分，值得注意的是 BIU 管理者 dcache 的内容。所以最后它会收到来自系统的 cache probes，然后进行必要的 cache 一致性的操作，最后相应系统。

3.1.2 MIPS r10000

MIPS r10000 也是微处理器设计中广为人知的经典之作。如下是其微结构的逻辑框图： [8]



1. **取指**。r10K 一拍内能取 4 条指令。如果指令因为是 structural hazards 或者指令本身是 special instructions，那么将会被搁置在 8-word 的 instruction buffer 里而先不会被译码。

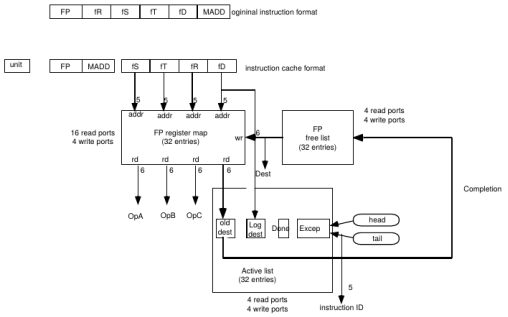


2. **转移猜测**。用 2-bit 512 entry BHT，有 4 个 entry branch stack 存储着 alternate branch address 和 integer and FP register map tables 的副本。意思应该是支持 4 次转移猜测而不阻指令流。如果预测错了，进行 Mispredicted branches 的处理，首先立刻恢复 state，并且依据 alternate branch address 来取指。为了取消掉取错的指令，r10K 每条指令都标记着 4 bit branch mask，对应的是哪一次分支预测后所取的指令。 [8]

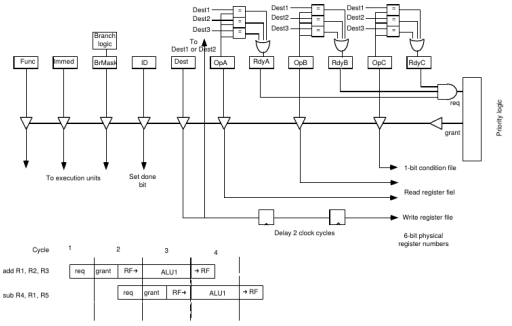
3. **寄存器重命名**。MIPS 逻辑的通用寄存器空间有 5 位，r10K 物理寄存器空间有 6 位，也就是最多 64 个。能够消除 WAR and WAW hazards。同时维护着 33×6 bit RAM 的整数寄存器 map 表和 32×6bit RAM 的浮点寄存器 map 表。r10K 还有 32 entry FIFOs (four parallel, eight-deep) [8] 的 Free List，里面维护的信息是当前未分配的物理寄存器号。与此对应的是 Active list。所有 in-flight 的指令都被记录在 32-

entry 的 FIFO (four parallel, eight-deep) 中，所以为每一条指令提供了了 5-bit 的 ID。操作起来像 reorder buffer，如果运算单元完成了指令的运算，将会发送 tag 到 Active list，然后将对应 entry 的 done bit 置上。每个 entry 中还存储了逻辑的目的寄存器号和旧的物理寄存器号。同时 r10K 支持精确例外，当例外发生，例外指令的后续指令不会被 graduate，处理器也会从 the active list 中恢复旧的映射关系。

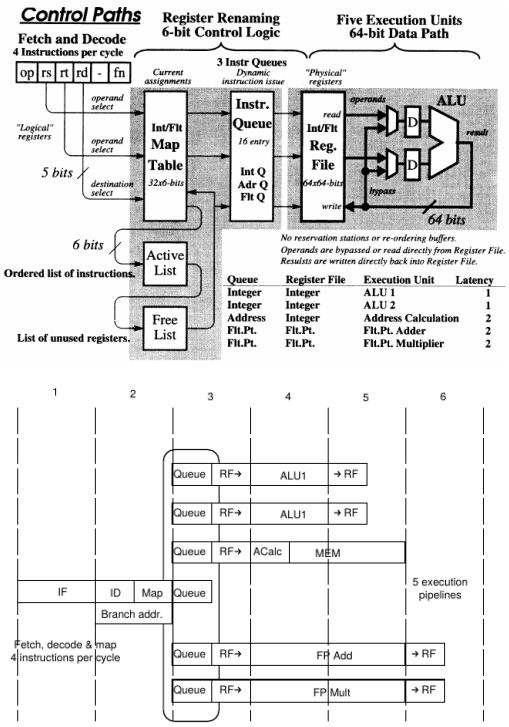
这里是性能的直接体现，前文中的 Alpha 21264 最多支持 80 条指令乱序执行，而 MIPS r10K 是 32 条，差距还是比较明显的。



4. **发射队列**。有三个队列，分别是 Integer，Address 和 Float-point. 队列在整个 CPU 的关键路径上，在译码的时候分配 entries。三个队列的配置相似，都是 16-entries，每个 entry 有 10 个 16 bit comparators 来处理 RAW hazards。 [8] 不同之处在于 Interger 和 FP no order 的，言下之意是如队列是从尾进，出队列 issue 则是没有绝对次序的，只有在大于运算流水能够允许的最大发射数量的时候才会看次序。但是访存的队列就是有 order 的。严格按照次序发射进行地址运算然后访问内存。以 Integer Queue 为例，有下图： [8]



5. **指令执行和流水**。可以用如下框图和时序图来概况 r10K 的行为： [8]



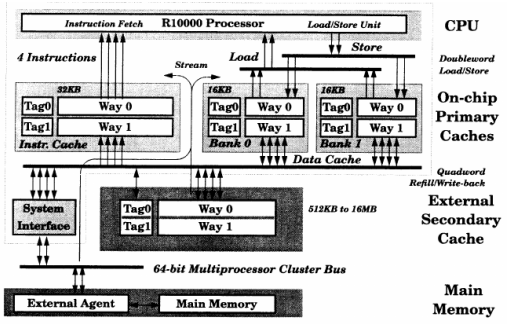
6. **Memory Hierarchy**. 32KB, 2 路，128B line size(16 条 64 位的指令) 的 icache；

   TLB 的结构是有 64 个 entries，每个 entries 2pages，如果每个 page 有 4KB，那么能够快速转化的虚拟地址是 1MB；

   dcache 是 32KB，2-way，64B line size，writeback。还有支持如下特性： [8]

   - 2 路 interleaved 取数存数以及 cache 充填来提高带宽。

- 非阻塞 outstanding 支持 4 条访存请求。



L2 cache 是 128b 的 interface，可配置容量 512KB-16MB，采用 pseudo 2-way SA using 8 Kb MRU table. [8]

3.1.3 BOOM

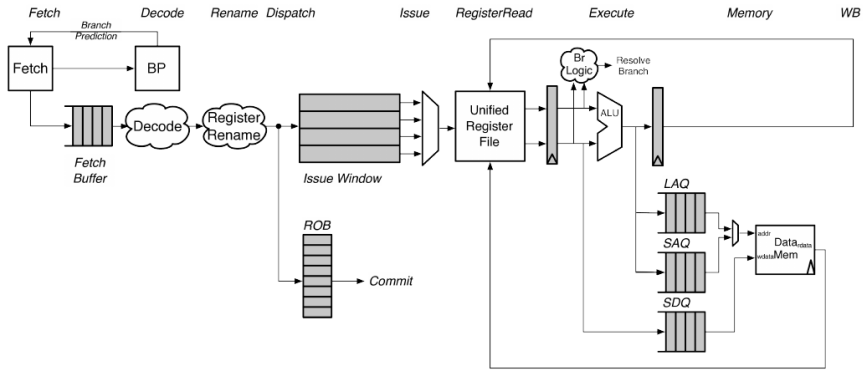
BOOM 是和 RISC-V 一起开源的一款 superscalar 和 out-of-order 处理器核。

BOOM 的设计者是和我一样的学生，所以同样参考的是最为经典 MIPS R10k 和 Alpha 21264。[3] BOOM 的版本经历了两代，下图是两代的对照：

	BOOMv1	BOOMv2
BTB entries	40 (fully-associative)	64 x 4 (set-associative)
Fetch Width	2 insts	2 insts
Issue Width	3 micro-ops	4 micro-ops
Issue Entries	20	16/16/16
Regfile	7r3w	6r3w (int), 3r2w (fp)
Exe Units	iALU+iMul+FMA iALU+fDiv Load/Store	iALU+iMul+iDiv iALU FMA+fDiv Load/Store

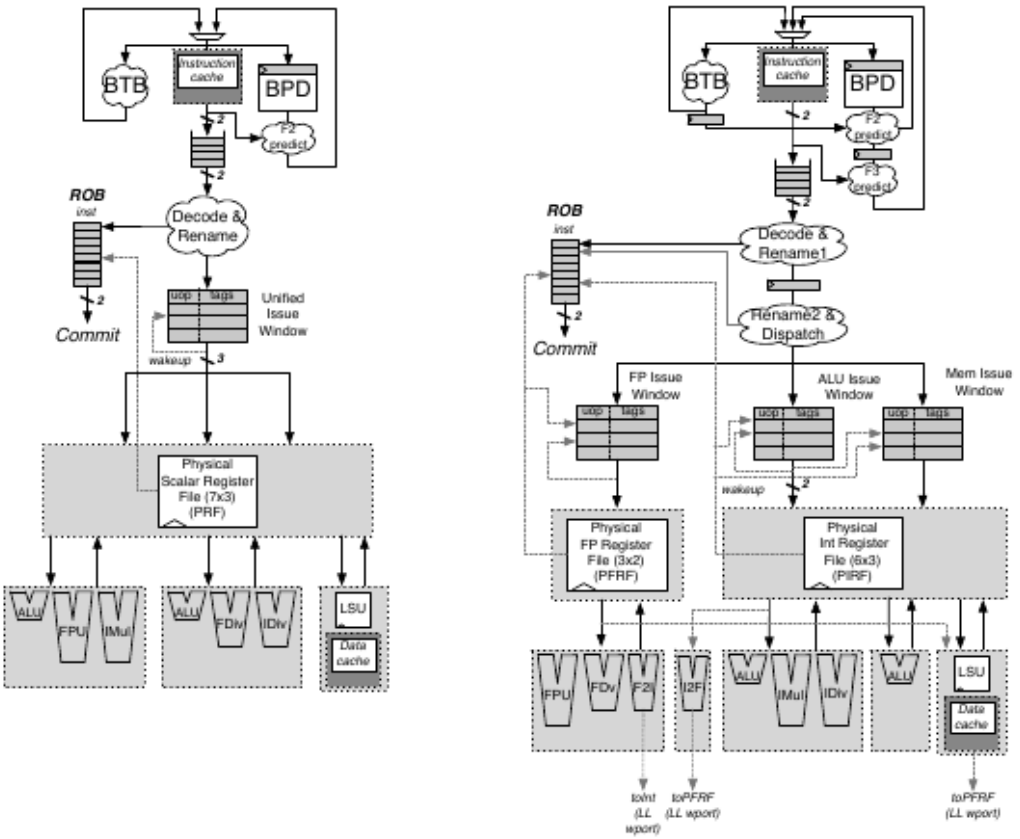
BOOMv1,v2 配置对照表 [3]

但是注意到 BOOM 其实是高度可配置的 generator，换言之前面几个参数都是可以配置的。



BOOMv1 的逻辑框图 [3]

如下是 BOOMv2 和 v1 的结构对照图：



(a) BOOMv1 (b) BOOMv2

BOOMv1 and BOOMv2 的 datapath 对比。v2 的发射窗口和物理寄存器都是分布式的并且在取指和寄存器重命名阶段多加了一拍。 [3]

- **BOOMv1**(下面简称 v1)

  v1 的 IPC 达到 3.91 CoreMarks/MHz (和 A9 相似配置) 和 ARM Cortex-A9 的 IPC 相当, 按我的以前的经验, 单发射五级静态流水最多可以做到 3 出头点的分数 (也就是 IPC 接近于 1)。所以第一代 IPC 不错, 快了差不多 33%。主频方面, 用 IBM 45 nm SOI 的工艺可以达到 1.5GHz, the SRAM access 将会成为 the critical path [3]。感觉言下之意就是不是他设计不行, 奈何工艺不好。如下是 BOOMv1 文档中的各款处理器的性能对比图 [3]:

Table 1: CoreMark results.

Processor	Core Area (core+L1s)	CoreMark/ MHz/Core	Freq (MHz)	CoreMark/ Core	IPC
Intel Xeon E5 2687W (Sandy)†	≈18 mm²@32nm	**7.36**	3,400	25,007	-
Intel Xeon E5 2667 (Ivy)*	≈12 mm²@22nm	**5.60**	3,300	18,474	1.96
ARM Cortex-A15*	2.8 mm²@28nm	**4.72**	2,116	9,977	1.50
RV64 BOOM four-wide*	1.4 mm²@45nm	**4.70**	1,500	7,050	1.50
RV64 BOOM two-wide*	1.1 mm²@45nm	**3.91**	1,500	5,865	1.25
ARM Cortex-A9 (Kayla Tegra 3)*	≈2.5 mm²@40nm	**3.71**	1,400	5,189	1.19
MIPS 74K‡	2.5 mm²@65nm	**2.50**	1,600	4,000	-
RV64 Rocket*	0.5 mm²@45nm	**2.32**	1,500	3,480	0.76
ARM Cortex-A5‡	0.5 mm²@40nm	**2.13**	1,000	2,125	-

Results collected from *the authors (using gcc51 -O3 and perf), †[3], or ‡ [1]. The Intel core areas include the L1 and L2 caches.

这里有一个我没有搞清楚的地方是 two-wide 和 four-wide, 文档里没有交代, 我估计是发射的宽度, 也即从保留站里每拍至多发射多少条指令去执行流水线上执行。

v1 效仿 MIPS R10K 的设计, 采用了 6 级流水 – fetch, decode/rename, issue/register-read, execute, memory, and

writeback。对应分支跳转的预测在分支指令被译码的后一拍执行。同时物理寄存器是 unified 的，integer 和 float-point 寄存器统一在一起，包括运算单元也是整数部件和浮点部件混合在一起. [3]

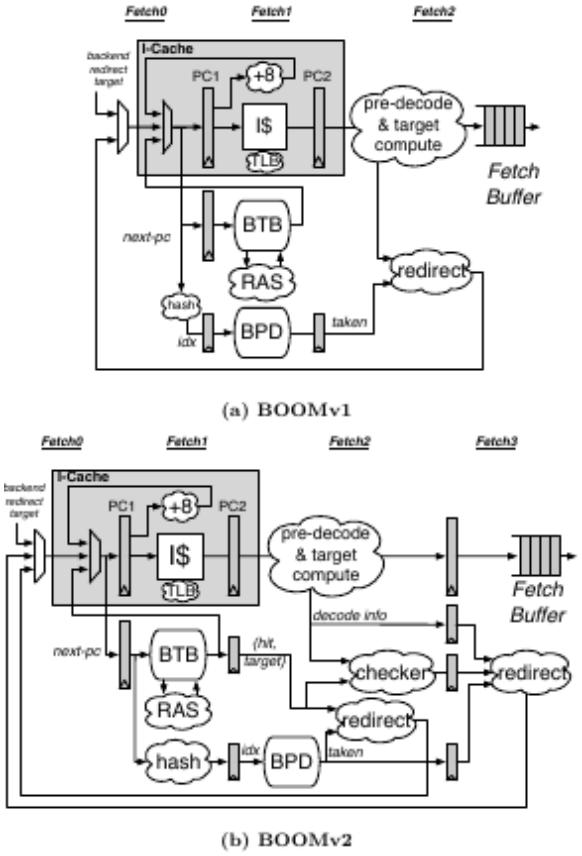
- **BOOMv2**(下面简称 v2) 现有的文档更多的是介绍第二代版本。所以 v1 和 v2 的比较以及整个 BOOM 的微结构设计都放在这里说明。首先需要交代的一点就是相比于 v1, v2 缩短了关键路劲的延迟，但是 IPC 却比 v1 降低了 20%. [3]

  1. **取指**。包括转移猜测在内，整个提供指令流的部分被成为 frontend. BOOM 采用了 3 中手段来提高转移猜测的准确率。

     (a) Branch Target Buffer (BTB) 维护的是一个 PCs 到 branch target 的映射表，用 PC 地址索引查表然后对比 tag，命中则做出预测。同时有一些 hysteresis bits 来帮助做出 taken/not-taken 的决定。BTB 是非常耗费资源的结构，大概要 20 bits 的 tag 和整个 64 bits 的 target(整个架构是 64 位)

     (b) Return Address Stack (RAS) 预测函数的返回地址。函数的调用和返回可以用栈的结构来刻画。

     (c) Conditional Branch Predictor (BPD) 针对分支决定跳与不跳的预测，不会存储耗费资源的 branch targets，而是通过取回来的分支指令计算得到跳转的地址。一种常见的 BPD 是 global history predictor。通过跟踪钱 N 次跳转的历史，并将这个历史 hash 到预测表中。



BOOMv1 and BOOMv2 的前端流水线对转图. [3]

     可以看出 v2 的转移猜测比 v1 复杂。而且 v2 增加了一级并且把 BPD 的前序工作 hash 移到 F1 阶段无非就是消除之前 v1 的关键路径。同样把 BTB 改成组相连也是处于消除关键路径的考虑。但是对此新的设计也会因此浪费了一拍即使预测器猜对了，这也是导致 IPC 下降的原因之一。

  2. **发射队列 (保留站)** v1 是 20 unified entries，而 v2 则是分为 3 个队列 (integer, memory, floating point)，各 16 entries. [3]

  3. **寄存器堆的设计**。我之前并没有注意到会存在这个问题，可是从 BOOM 的工程实践来看，在 v1 上，由于采

用的是 unified 的 register file，就是无论 integer 还是 FP，都是同一个物理寄存器堆，不仅大，而且读写端口多，造成的后果就是不仅在关键路径上，而且还很难布局布线。所以在 v2 上，就采用了 3 种方法去提高 register file 的性能。

- 将发射和读寄存器分成两级，issue-select 先选出来哪些指令要发射，然后时钟打一拍，下一拍再从寄存器里读取操作数。
- 将 Integer 和 FP register 分开来。从而降低分别的寄存器的数量以及读写端口的数量。
- 最后一个方法是物理的设计，是布局布线的优化，囿于有限的知识水平，就只截了一张图 [3]：

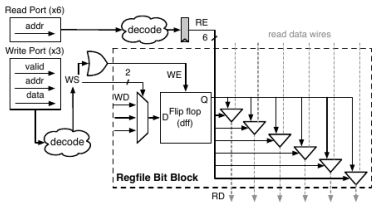


Figure 4: A Register File Bit manually crafted out of foundry-provided standard cells. Each read port provides a read-enable bit to signal a tri-state buffer to drive its port's read data line. The register file bits are laid out in an array for placement with guidance to the place tools. The tools are then allowed to automatically route the 18 wires into and out of each bit block.

课题主要研究内容、预期目标：

1 CPU 设计

这一部分是毕业设计中最为核心的部分，由于当前还没有具体的设计，所以只能初步谈谈当前的构思，等到中期报告的时候这一部分会具体展开。但是框架的基础是基于 MIPS R10000 的。虽然 MIPS R10000 比 Alpha 21264 性能低，但是相对而言简单些，同时不难看出 BOOM 借鉴的更多的也是 MIPS R10000。

1.1 instruction fetch

处理器的 IPC 高不高，首先就要看指令供不供得上，所以这一部分也是优先设计并且测试的模块。本科毕业的设计采用每一排取两条指令的设计。两条指令也即意味着需要两个译码器，译码完成后 push 到发射队列里面等待发射。

1.1.1 branch prediction

转移猜测是取指器中重中之重的结构。猜测猜什么？猜跳转地址 (非直接跳转)，猜跳与不跳 (分支指令)。几个在 BOOM 里面提到过的结构 BTB，RAS，BPD 要实现，同时在 Alpha 21264 中提到的全局和局部猜测结果混合的技术也可以考虑。现实情况是这一部分很难设计，而且容易成为关键路径。

1.2 multi-issue

指令取回来就要进入发射队列或者叫保留站等待多发射，然后同时进行重命名操作。如果是静态调度的结构，那么指令队列也就是发射队列，但是对于动态调度，保留站在指令队列的下游。一开始的设计从简，所以就先考虑静态的调度。

1.2.1 rename

重命名什么，从逻辑地址到物理地址。所以最多一个目的寄存器需要重命名，最多两个源寄存器地址需要从重命名的表中找到逻辑地址对应的物理地址。

1.3 out-of-order pipeline

要多少个 ALU，多少个浮点运算器，访存部件队列有多大都是值得商榷的问题，要注意和 ROB 大小，保留站大小，重命名表大小之间的平衡。

1.4 instruction retire & result commit

一次能 retire 最多几条指令也是要平衡的问题。

1.4.1 precise exception handling

例外的硬件处理一定是要做成精确例外的，所以例外的 restore state 机制要做，必要的信息要加。

1.5 memory system

为了提高性能，除了取指，访存的结构设计也是非常重要的，而且也会复杂。这是因为访存并不是由 CPU 完全控制的，而且延迟随机。对外接口是握手信号，同时例外处理也要 restore 访存队列状态，保持到下一次有效访存请求出现。

1.5.1 load & store mechanism

访存队列做多大，outstanding 程序要做到什么程度，一拍接受多少个访存请求也是掌握平衡。

1.6 the design which decouples microarchitecture and ISA

由于最后还是要做一个同样微结构的 MIPS 处理器，所以设计要尽可能的将微结构和 ISA 解耦合。

2 CPU 可同时配置 MPIS 的 generator

不同的 ISA 怎么比较，我认为最为客观的方法是用一模一样的微结构。而且由于 RISC-V 和 MIPS 设计相近，如果完成了一个的设计，切换到另外一个也不是难事。

3 设计空间分析与优化

正如前面所说，在具体的设计中有很多有很多的参数大小需要平衡。优化的时候可能还会尝试用到更为复杂的微结构，比如说更为复杂的转移猜测，保留站的模式等等。

4 预期目标

既然是做多发射乱序的 CPU，那么最基本的预期就是性能要比单发射五级静态流水 CPU 要提高 50%∼100%. 同时非常期待 RISC-V 和 MIPS 的比较结果。最后更为进阶的目标是能够跑通 Linux，甚至能够支持多并且跑多进程能够看到加速比，并奢望能够赶上 Cortex A53 的水平。也许本科毕设阶段做不完，但这是一个长期的项目。

拟采用的研究方法、技术路线、实验方案及其可行性分析：

乱序多发射的 CPU 是通往高性能处理器的必经之路，Intel、AMD、ARM 都应经有了成熟的技术。一个典型的代表就是 ARM Cortex A53。作为一个双发射 8 级静态调度的处理器核，面积很小，功耗很低，SPEC2000 INT base 能够达到 423 分/GHz (report) . 但是另外一方面，乱序多发射的设计并不容易，而且比设计更加困难的是调试，如何保证复杂执行的 CPU 能够有条不紊，毫无错误的运行比如像 Linux 操作系统这样的大型软件更是难上加难。仅靠一个人的力量是很难在短短数月里面完成的。所幸前有 Alpha 的技术报告，后有 BOOM 的开源实现可供参考，而 BOOM 背后更是有整个 RISC-V 的开源生态，有整套的调试方法和 software stack toolchain。RISC-V 作为近些年在总结了前人 ISA 设计中的利弊的基础上推出的一套设计清晰，干净整洁而优雅的开源 ISA，大受学术界工业界的欢迎。鉴于其日臻完善的开源的开发调试工具，有如下考虑方案：

1. UC, Berkeley 开发的高级硬件描述语言 Chisel 对单一时钟，同步复位的电路逻辑的刻画效率比 verilog 更高，而且对于可综合电路的描述能够做到和 verilog 同样的精确，故而完全可以采用 Chisel 进行毕设 CPU 的设计开发。
2. 可以参考借鉴开源的 BOOM 代码实现。
3. 可以学习借用 RISC-V 开源调试工具与手段。
4. 可以低成本地在 FPGA 平台上的运行目前 RISC-V 开源的整套以 Linux 为首的软件栈。

综上，计划的第一步是实现一个基于 RISC-V ISA 的乱序双发射的 CPU，调试完毕能够跑通简单的测试程序，但深知这绝非易事。接着在时间允许的情况下做操作系统的移植，继而能够支持多核。注意到一开始的设计必须考虑到微结构和 ISA 之间的解耦合，这样将 RISC-V 改成 MIPS 工作量会降低很多。得到 MIPS 版本的 CPU 目的在于采用相同的微结构可以客观的对比两个 ISA 的优劣。同时作为工作的非常重要的一部分，性能的优化和设计空间的搜索分析必不可少，会尝试一些方法比如更换各个结构的配置参数，甚至调整逻辑部件的整个结构。

已有研究基础与所需的研究条件：

前文的**国内外本学科领域的发展现状与趋势**已经论述了很多关于已有的研究基础的内容，有代码语言层面的 Chisel，有 ISA 层面的 MIPS 和主要是 RISC-V 的设计考虑，有 RTL 的逻辑设计的前人经验：Alpha 21264, MIPS R10000 和 BOOM。下来再来论述基于 RISC-V 的调试方法和 software stack 以及 toolchain.

伯克利分校一开始的 RISC-V 的开源项目模式是 Tethered，而非 Standalone Systems 模式。这两种模式在本科阶段的实验课都涉及到。大二下的计算机组成原理实验课就是 Tethered 模式，不能独立运行，要靠其他成熟的 Host(如 ARM 的核) 才能运行。大三上的计算机体系结构是 Standalone Systems 模式，设计的 CPU 可以独立运行。网上有如何用 Rocket 的核运行在 Xilinx Zynq 板卡上并启动运行 Linux 内核的教程 `http://eliaskousk.teamdac.com/entry/welcome`。板卡上用的是 ARM A9 的核，然后 FPGA 的配置和组成原理实验课的板卡竟然惊人的相似，所以我就像张科老师要来了一块以前实验课的板卡，准备在以后的几天看看能不能复现。当然复现只是第一步，接下来就是熟悉 RISC-V 开源的平台，最重要的是调试环境。

首先 RISC-V 推荐的是他们自行开发的 Spike. Spike 作为一个软件实现的 ISA 模拟器，是个 ISA interpreter。以 tethered RISC-V 系统为 model，并作为功能正确的基准。设计的目标是快速而且易于修改。对于移植操作系统，RISC-V 也有一个 Proxy Kernel 能够运行在自行设计的 CPU 上，包括一下特性：

- 支持单用户进程
- I/O 相关的系统调用 forward 到 host 机上
- 实现了 Linux ABI 的子集

而一个完整的 Tethered RISC-V Systems 还包括 Frontend server 和 Host-Target Interface(HTIF)。Frontend server 负责代表 proxy kernel 执行 I/O 系统调用。另外一个功能是 boot loader(把 kernel 的 ELF 文件 load 到 target 的 memory 中)。HTIF 是一个 host 与 target 的 state 交互的简单协议。
此外 RISC-V 也有 QEMU 的 Full-System Simulator.
而上层的 toolchain 有 Linux，FreeBSD，GCC 和 LLVM。
最后后面要移植到龙芯的 MIPS ISA，也有相应的调试环境。
具体的调试手段会在中期报告中展开论述。

研究工作计划与进度安排：

1. 2019/01/08∼02/12 把乱序双发射 CPU 设计完毕。
2. 2019/02/13∼03/15 一个月后把 CPU 调试完成，跑通单核的 Linux。
3. 2019/03/16∼03/31 半个月完成移植 MIPS 的 ISA 的同样架构的 CPU，并调通。
4. 2019/04/01∼04/15 半个月完成对 MPIS 和 RISC-V 的两款 CPU 进行跑嵌入式 benchmark 如 coremark 等程序的量化性能比较与性能分析。
5. 2019/04/16∼ 五月中旬剩下的时间用于性能的调优，量化结果的分析总结，毕业论文的编写。若有余力，考虑多核的架构并能够运行多核的 Linux 能够有加速比。

参考文献

[1] Krste Asanović Andrew Waterman. *The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2*, May 2017.

[2] Krste Asanović Andrew Waterman. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Privileged Architecture Version 1.10 Document Version 1.10*, May 2017.

[3] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David A. Patterson, and Krste Asanović. Boom v2: an open-source out-of-order risc-v core. Technical Report UCB/EECS-2017-157, EECS Department, University of California, Berkeley, Sep 2017.

[4] Andrew Waterman David Patterson. *The RISC-V Reader: An Open Architecture Atlas Beta Edition, 0.0.1.* Oct 2017.

[5] John Wawrzynek Jonathan Bachrach, Krste Asanović. Chisel 3.0 tutorial (beta). Technical report, EECS Department, UC Berkeley, Jan 2017.

[6] R. E. Kessler. The alpha 21264 microprocessor. Technical report, Compaq Computer Corporation, 1999.

[7] Jonathan Bachrach Patrick S. Li, Adam M. Izraelevitz. Specification for the firrtl language. Technical report, EECS Department, University of California, Berkeley, Sep 2018.

[8] Kenneth C. Yeager. The mips rio000 superscalar microproce. Technical report, Silicon Graphics, Inc., 1996.

指导教师意见（课题难度是否适中、工作量是否饱满、进度安排是否合理、工作条件是否具备、是否同意开题等）：

指导教师签名：

年　月　日

学院（系）意见：

审 查 结 果：□ 同 意　□ 不 同 意

学院（系）负责人签名：

年　月　日