南京大学二队龙芯杯 CPU 设计报告

2018 年 9 月 17 日

队伍名称: 南京大学 2 队

成员: 刘志刚

欧先飞

沈明杰

邱子豪

指导老师: 李俊

蒋炎岩

目录

1 设计简介

1.1 项目结构简述

由于 verilog 语言设计本身的缺陷，其自身抽象能力不足，而且对类型系统的支持完全是 0，这使得我们在使用 verilog 进行 cpu 的编写的时候，麻烦程度不亚于手写汇编。所以在几经考量之下，我们决定用更高级的硬件描述语言 chisel 完成我们的开发流程。

chisel 是一门由伯克利大学开发的基于 scala 的硬件描述语言，由于起源自元编程语言 scala，chisel 拥有相当丰富的语法特性，同时也给我们的开发流程带来诸多便利，因此我们的项目也都是建立在 scala 之上，项目的目录布置也近似于 scala 的官方缺省方案。

项目的布局如下：

- build.sbt，这个是我们整个项目中 scala 源码文件的编译入口，在这个文件中定义了一些搜索规则，和 chisel 的支持包的下载站点，同时做了一些文件的时间戳检查，防止与 Makefile 做了重复的编译工作，文件中语法细节可参考https://www.scala-sbt.org/1.x/docs/index.html
- Makefile，由于 chisel 编写的 core 仅仅是项目的一小部分，除此之外还有很多的 uncore 部分需要处理，Makefile 具体干了哪些事，这里不展开细讲，如果想了解整个项目的编译流程，请详细阅读 Makefile，对于 Makefile 里面的一些特殊用法，可以参阅官方文档https://www.gnu.org/software/make/manual/make.pdf
- src/，这个是我们整个 cpu 的核心源码目录，包括 cpu 的 icache、dcache、l2 cache、流水段设置、cp0、分支预测器、协议转化桥、arbiter、crossbar，具体语言细节请参阅 scala 的官方手册http://people.cs.ksu.edu/~schmidt/705a/Scala/scala_tutorial.pdf，和 chisel 的官方手册https://chisel.eecs.berkeley.edu/api/latest/index.html
- emulator/，这是我们的模拟执行目录，在 src 目录下的所有源码文件最终会与 emulator 下的所有 verilog 文件及 cpp 文件一起编译链接，最终生成模拟器，模拟执行 cpu，并据此进行核内主要功能调试
- scripts/，这里包含一些脚本和工具，用于回归测试和差分测试
- uncore/，由于我们的 cpu 对多种板子均有支持，所以这个目录的存在用于多板子之间的解耦
- zynq_sw/，这个是用于快速上板所存在的目录，包括用于描述板上硬件的 dts 文件，用于加载程序的 loader，和用于引导 os 的 u-boot
- nscscc/，由于龙芯比赛规定，我们暂时无法修改龙芯比赛提供的项目文件和结构，因此我们没有将龙芯板子加到我们的 uncore 里面。而这个目录的存在是专门用于龙芯的上板：

nscscc/cpu_design.bd 用于生成我们的 cpu 在龙芯板子上的 uncore

nscscc/func-prj.tcl 用于生成符合龙芯比赛提交要求的 soc_axi_func 项目目录

nscscc/perf-prj.tcl 用于生成符合龙芯比赛提交要求的 soc_axi_perf 项目目录

nscscc/cpu_top.v 用于将我们使用到的 block design 包装成符合龙芯比赛接口的顶层 cpu

- nemu-mips32/，由我们组编写的 mips32 虚拟机，用于模拟执行 mips32 架构下的可执行文件，同时用于我们 cpu 的调试对比，尚未完工，所以暂时不会开源。

1.2 项目编译说明

我们已经将编译好的 verilog 文件放置在 soc 对应的目录下，无需特地在对项目进行编译。如果的确需要编译我们的源码，需要先安装一些工具：

- scala，scala 语言的编译器
- sbt，用于 scala 的项目管理工具
- verilator，用于编译 verilog 的 verilog 执行工具
- java，ubuntu 缺省的即可

以及一些需要设置的环境变量：

- AM_HOME，指向 AM 的目录，如果不需要运行差分测试和回归测试，可以不设置
- NPC_HOME，用于指向 noop-lo 所在目录的绝对路径
- NEMU_MIPS32_HOME，用于差分的 mips32 模拟器，如果不需要运行测试则不需要指定
- NSCSCC_HOME，用于指定大赛资源包所在的目录，支持第四版

在依赖包安装完毕之后可以通过 make submit 来生成功能测试模块所需的 verilog 文件和项目工程文件，生成完毕可在 outout/nscscc-submit 目录下找到对应的 vivado 项目目录，性能测试同理。如果需要仿真综合，还需要运行 make submit-soft 来构建仿真综合环境。

Makefile 中除了上述用于生成龙芯比赛的伪目标外，还定义了一些其他的伪目标，大概列举如下：

- emu，编译 core 生成模拟器模拟执行
- xsim，调用 vivado 的仿真模块进行仿真

- vivado，生成 zedboard 对应的项目文件并打开
- bit，综合生成 zedboard 的 bitstream 文件
- diff-insttest，用一份可确保正确的 mips32 虚拟机去差分模拟器，测试样例移植自今年龙芯比赛发布的资源。
- diff-cputests，对 cputests 进行差分，cputests 移植自我们 ICS 课程的测试程序，全部由 C 代码构成
- submit，生成符合龙芯杯比赛提交要求的项目目录
- loongson，用于生成符合龙芯比赛要求的 verilog 文件

2 设计方案

2.1 模块设计图纸

我们的 cpu 整体设计为 6 段流水线，如图1所示。由于初赛数据集较小，大部分数据集都可以直接装在 l1 cache，l2 cache 的设置在初赛中反而在某些数据集上起反效果，所以在最终提交的源码中，我们移除了这一组件。以及图中并未显式画出 flush 信号，这一信号用于分支跳转以及中断异常的冲刷。

图中所有的执行单元和写回单元 (wbu) 均有一条虚线连接到 ISU，这条虚线的含义是 bypass 信号，由于实际数据写回至少会晚上 3 个周期，为了能够更快的获取到寄存器数据，防止 ISU 因为等待数据就绪而空耗时间，项目添加了如图中虚线所示的 bypass 信号。

除了 core，为了能够对接到龙芯的开发板，我们加入 block design 用于处理外围引线，其中 block design 的设计如图2所示：

2.2 core 模块简介

CPU 核心的架构如下图所示：

首先是 IFU，取指单元。负责从 icache 中将指令取上来。由于 icache 存储使用的是 block ram，而 block ram 是在下一周期返回读结果的，所以相应的，IFU 是两级流水。第一级流水将 PC 发给 icache，第二个周期从 icache 中将数据取上来。

接下来是 IDU，译码模块。由于我们实现的是初步的乱序结构，在译码阶段，除了要完成指令的译码外，还要完成乱序需要的功能：寄存器重命名：对于源寄存器，要读取 scoreboard，读取 rename table 对于目的寄存器需要分配物理寄存器（处理 WAW）分配 ROB 分配 Branch ID 分配 Load Store queue 项，

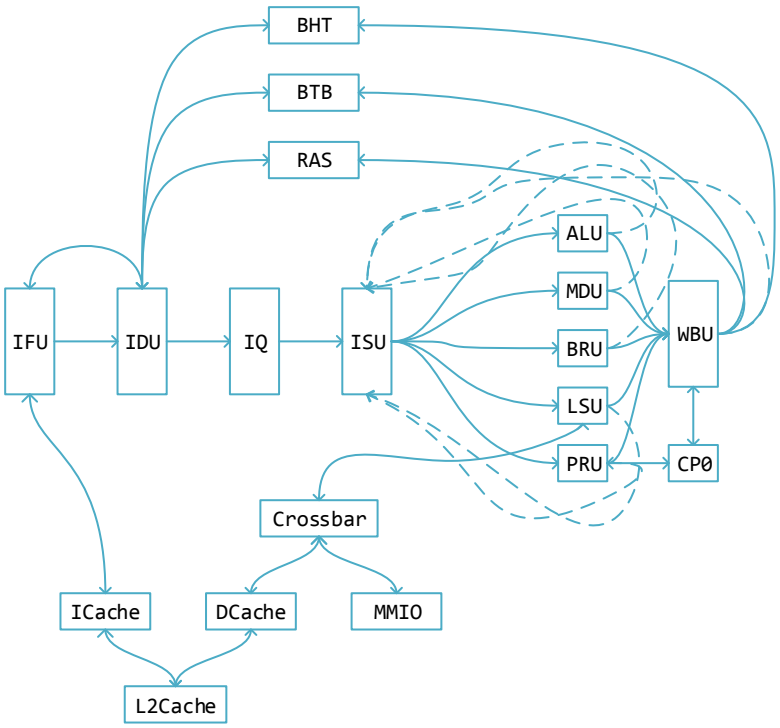
BHT
BTB
RAS
IFU
IDU
IQ
ISU
ALU
MDU
BRU
LSU
PRU
WBU
CP0
Crossbar
ICache
DCache
MMIO
L2Cache

图 1: CPU 的核内设计

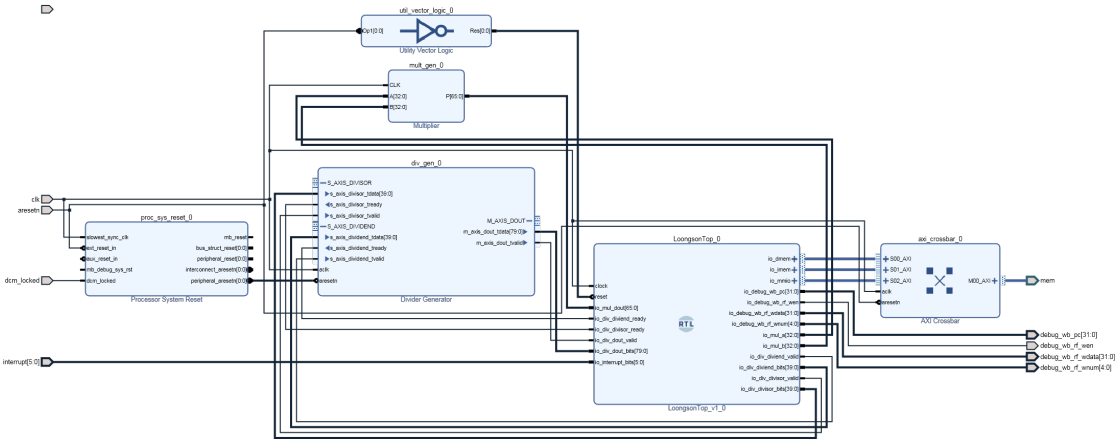

图 2: CPU 的核外布线

分支预测：分支预测模块包含了三样东西：BHT，branch history table，负责进行分支预测，我们实现的是类似 gshare 的分支预测器。BTB，branch target table，负责预测间接跳转指令的跳转地址。RAS，RAS 负责预测函数返回的跳转地址。

计分板 scoreboard 中主要有：32 项的 status，记录每个寄存器现在的状态。寄存器共有三种状态，他们的含义分别是：SB_READY：就绪态，这个寄存器的最新结果就在寄存器中。SB_BUSY：寄存器已经被重命名到 ROB 了，但是结果尚未就绪。SB_ROB_READY：此寄存器已经被重命名到 ROB 中了，结果就绪，但是此指令尚未 retire，所以结果要从 ROB 中读取。

ISU 然后是 ISU，即指令发射队列，译码后的指令进入发射队列中，当源操作就绪，并且后端执行单元空闲时，即可读取发射操作数，并发射到后端执行单元执行。

后端执行单元：后端执行单元执行完指令后，将指令状态以及计算结果写入 ROB 中，并通过 bypass network 传递到 ISU。

ALU：计算算术逻辑指令 MDU：计算乘除法指令 PRU：处理 cp0 指令 BRU: 处理分支跳转指令

LSU: 处理 load store 指令 LSU 以及 Load Store queue 的设计 LSU 主要的工作是：进行译码，地址生成，以及地址转换（查 TLB），将指令派发到 Load Queue 以及 Store queue。如果这条指令是 load，则派发到 Load queue，如果是 Store，则派发到 store queue。load queue、store queue 的释放都是在指令 retire 后进行的。

ROB reorder buffer，存储指令计算的结果，以及指令的例如 pc、异常等基本信息。ROB 是一条有序队列，每个执行单元到 ROB 都有一个写口，另外 ROB 有两个读口，用于 ISU 从中读取数据，另外 ROB 还有一个 commit 口。

WBU 当 ROB 头部的指令就绪时，它就可以 commit 了。然后它就进入了 WBU，写

回单元。写回单元检查指令是否出了异常，并进行相应的处理。另外，对于 non-speculative load/store，WBU 将他们对应的项从 Load store queue 中唤醒，并执行。

TLB 我们实现了 64 项 TLB，支持虚拟内存。TLB 本质上是一个大的 CAM 查找表。

L1 ICache、DCache L1 ICache 负责处理来自 IFU 的请求，DCache 处理来自 load store queue 的请求，L1 ICache 与 DCache 均用 block ram 实现，均为两级流水，在 cache hit 时，能一直流水。在 cache miss 时，则阻塞。另外 L1 DCache 支持背靠背的 load，store，连续的 load、store，store、load 之间的数据转发均可以处理。

L2 cache 我们实现了 L2 cache，AXI4 接口，blocking，一次只能只能处理一个请求。由于龙芯测试集 working set 很小，所以在性能测试中，我们没有使用这个 L2 cache。

3 实现中的一些细节

3.1 未重命名的指令以及寄存器

此外由于我们现在暂时没有对 cp0 寄存器以及 Hi，Lo 寄存器进行重命名，为了保持依赖，同时为了实现简便。现在在 IDU 阶段，cp0 指令以及乘除法指令或者对上述寄存器的所有操作，都必须要等 rob 空了才会接着执行。IDU 负责识别这些指令，并对前端进行必要的阻塞。

3.2 发射队列的实现

然后是 ISU，即指令发射队列，译码后的指令进入发射队列中，发射队列中的每一项在进来的时候，都会记住自己的源寄存器的状态（在 IDU 阶段从 scoreboard 中读取），以及被重名到了 ROB 的哪一项。如果指令的源操作数的状态是 SB_READY 或者 SB_ROB_READY，则可以被直接发射，并不需要等待依赖满足。但是如果某一个操作数的状态是 SB_BUSY，则这条指令需要待在 Issue queue 中，等待唤醒。

issue queue 中的指令主要要满足的依赖有：源操作数就绪，执行单元空闲。

3.3 唤醒操作

我们从后端的每一个执行单元，以及 WBU 写回单元，都拉出一组线到 issue queue，包含要结果就绪的寄存器下标、rob 项下标以及算出来的值。我们将寄存器以及 rob 地址，与 issue queue 中的每一项进行比对，判断该条指令是否有源操作数就绪。这构成了 wake up 操作。另外，传递过来的算出来的新值，构成了 bypass network。

3.4 发射

源数据就绪的指令，即可被发射，由于受发射宽度的限制（现在为 1），我们用 priority encoder 选择一条指令出来发射。指令在发射时读取源操作数，源操作数来自于：寄存器，ROB 或者 bypass network。Speculative load、store。由于对 mmio 的 load/store 以及对 memory 的 store 是有副作用的，为了实现精确内存，这些 load store 是必须在指令 retire 时才能执行的。而对 memory 的 load，是可以推测执行的。

3.5 精确内存，speculative load store

由于 mmio load/store，memory store 是有副作用的，仅 memory load 无副作用。所以 load queue 中对 memory 的 load 是可以推测执行的，而对 load queue 中对 mmio 的访问，以及 store queue 中的任务是在指令 retire 后执行的。

3.6 dynamic memory disambiguation 以及维护 memory RAW dependency

由于指令是乱序发射并且 load 是允许推测执行的，所以可能出现新的 load 在老的 store 之前被 issue 到 lsq，被执行完成，造成依赖关系错误。由于我们将 load queue 以及 store queue 实现成了两条无序 queue，无法单从 queue 的位置关系，判断 load，store 指令间的先后关系。同时，我们也没有实现 store queue 到 load queue 之间的 bypass。所以我们是在 store 指令 retire 时，将其地址与 load queue 中的每一项进行匹配，如果有匹配上的，并且已经执行的 load，则说明出现了 dependency failure，然后通过冲刷流水线来解决。这样子做是可行的，因为 store 指令是在 retire 时执行，此时，我们可以保证 load queue 和 store queue 中的指令都比它年轻。

3.7 ICache、DCache 一致性

我们实现了 Cache 指令，支持 ICache、DCache 一致性。

3.8 从 I4 到 IO2I

我们在初赛时提交的版本是一个全顺序的流水线，In order fetch，in order issue, in order execute, in order retire，简称 I4。在初赛后，我们对其进行了下面这些提升，将性能测试跑分提升到了 52 分：

a. 更低的乘除法器延迟

b. 原有的实现中的 cache line 的 valid，dirty 均是用触发器实现，因为无法综合成 BRAM，占据了大量的资源，影响了频率。在解决这个问题后，频率从 70M 提升到 90M。

c. 提升分支预测准确率：

(a) perf test 中的绝大部分测试需要很短的 history，小部分，如 bubble sort、quick sort 需要很长的 history，因此我们使用了两个 history 长度不同的 BHT，并取预测效果最好的作为分支预测结果。

(b) RAS speculative update。由于流水线较长（主要是由于中间 instruction queue 的存在），而我们原来 RAS 压栈出栈都是在 WBU 阶段进行的，导致对于非常小的函数调用，call 尚未 commit，并将返回地址压入 RAS，return 就已经进了流水线了。我们支持了让 RAS 支持 speculative update，在 IDU 阶段遇到 call，return，就更新 RAS。由于是 speculative update，所以当出现分支预测错误时，RAS 会进行部分恢复（恢复 stack top 指针）。添加了这个后，对于小函数调用多的 stringsearch，提升明显，几乎所有的 test 的 return 预测错误，都降到了接近 0

d. DCache store、load bypass。对于 stringsearch、stream copy 等测试，存在明显的 store，load 背靠背的情况，我们处理了背靠背地址冲突的情况，加上了 bypass。明显提升了这些项的分数。

3.9 IO2I：初步的乱序处理器

in order fetch，OoO issue，OoO execute，In order retire。此款处理器能够运行在 70M，跑分 33 分左右。

考虑到乱序的复杂性，为了保证正确性，尽早拿出一个能够运行的版本，我们在实现中做了如下取舍，这保证了实现的低复杂度与正确定，但是性能不太好，甚至不如 I4 版本处理器。

1、显式重命名（重命名到物理寄存器）VS 隐式重命名（重命名到 ROB）重命名主要要处理好：

scoreboard：寄存器 ready，busy 状态物理寄存器资源分配

对于 scoreboard，这两种方案复杂性差不多。但是对于物理寄存器资源的分配，差别则很大。显式重命名需要实现 active list，free list 来维护物理寄存器的分配，在遇到分支时，需要做 active list 以及 rename table 进行 check point，分支预测错误时需要进行回滚，复杂度更高。同时在异常时，需要 walk back rob，复杂度太高。上述这些可能看起来还不是太复杂，但是要考虑到：在 checkpoint 以及恢复 free list，scoreboard 以及 rename table 时，这些资源的分配，回收在同时进行，corner case 非常多，不容易写对。一不小心就容易出现：1、资源泄露，程序卡死（free list 没有回收好）2、依赖处理错误，程序错误（scoreboard 没有处理好）3、资源被重分配，程序错误（rename talbe 没有处理好）

而隐式重命名只需要维护 ROB，flush 直接 flush rob 即可，不需要复杂的 checkpoint 以及恢复操作。

2、分支预测错误冲刷流水线的时机问题：由于 ROB 的存在，如果在 retire 时冲刷，会导致分支预测错代价太大了，分支预测错要尽早 flush。按理说，对于重命名到 ROB 的实现，只要将 branch 后的指令全部从 rob 中刷掉就可以了，由于我们需要 scoreboard 来维护寄存器是否 ready，所以 scoreboard 必须要做 checkpoint，这就又是一个复杂的点。另外，由于我们要支持延迟槽指令，即使 branch 是放到了 retire 时冲刷，我们也不能直接对 scoreboard 进行 reset，因为此时延迟槽指令还在流水线中。所以对于这种方案，我们也要对 scoreboard 进行 checkpoint，并在 flush 时恢复。为了让实现简单正确，我们采取的是分支预测错的 flush，要等到延迟槽指令 retire 时，才发出，这样子可以保证流水线中没有任何处理器，scoreboard 也可以直接 flush，而不必进行 checkpoint。

3、load store queue 的实现对于 load store queue，由于我们采取的是 load queue，store queue 都是无序 queue 的实现，load 指令无法判断哪条 store 比它老，因此也就无法进行数据 bypass。我们采取了更加简单的方式来保证 RAW dependency，store 时，如果发现了有比它新的 load 被执行了，那就 flush 流水线。

4、简单实现的资源占用问题对于单发射的乱序处理器，理论上，wake port 可以只有一个，ROB 的读口应该可以控制到三个（两个用来读数据，一个用来 commit），写口应该限制到一个。我们为了简单起见，所有的 FU 都是直接 wake up issue queue，都直接将结果写入 ROB，这导致了 ROB 的写口数量暴增，issue 的 CAM 逻辑资源使用量也很大，进而限制了主频，以及 issue queue 与 ROB 的项数，进而限制了性能。

5、对 Hi，Lo 没有重命名，乘除法指令等待 ROB 空了再发射，这严重影响了 coremark 等乘除法密集型应用的性能。

如图1所示，图中列出了主要的 core 设计所涉及的模块，其中各个模块的含义大致描述如下：

a. IFU 为取指单元，其通过握手信号向 ICache 发送取指信号，为了防止照成关键路径，ICache 内部数据存储使用的是 block ram，所以 ICache 在收到数据的下一周期才能返回数据，同时 ICache 的内部实现了流水化，所以迟一个周期返回数据几乎没有影响。

b. IDU 为译码单元，用于对 IFU 取得的指令进行译码，以取得这条指令的操作数寻找模式，操作数扩展方式，访问的寄存器编号，写回的寄存器编号，所需发往的执行单元编号。

c. IQ 为指令队列，用于存储所有已经译码成功的指令，为了防止后端执行周期过长阻塞前端。

d. ISU 为分发单元，通过译码所得的结果读取寄存器，同时在寄存器读写约束不满足的时候进行必要的等待，索引转发结果。

e. ALU 为算术运算单元，用于完成加法、减法、移位等可在一个周期内出结果的运算指令

f. MDU 为乘除法单元，用于完成乘法和除法的运算，这一模块主体执行功能放在了核外的 IP 核上，而 MDU 内部逻辑主要是等待乘除法器的执行结果。

g. BRU 为分支跳转单元，通过对 ISU 传入的操作数进行计算，以获取跳转目标，同时对于条件跳转指令还需要计算一下是否跳转

h. LSU 为访存单元，用于 load store 指令的执行，若 dcache miss，还需要额外的周期等待访存结果

i. PRU 为特权指令单元，非特权指令不允许发送到这个单元，需要发送到这个单元的指令有 MFC0、MTC0、SYSCALL、ERET，同时对于取指异常的指令和译码异常的指令，也需要发送到这个单元用以下拉异常信号

j. CP0 为 0 号协处理器，用于完成中断异常主要计算功能

k. WBU 为写回单元，用于将执行单元获取到的操作数写回到目标寄存器

l. BHT、BTB、RAS 为分支预测单元，由于延迟槽的存在，整体的分支预测请求由 IDU 发出，并用于重定位 IFU，wbu 在写回寄存器时，也需要将 BRU 的计算结果更新到分支预测单元

4 设计结果

4.1 设计交付物说明

交付物目录结构基本符合大赛规定,其中 soc_axi_func 为功能测试的源码目录,soc_axi_perf 为性能测试的源码目录，score.xls 为性能测试的跑分结果。除此之外还有一个 noop-lo 目录，其中存放着我们整个 cpu 设计的实际源码。

在 soc 目录下的 rtl/myCPU 下总计有三个文件，其中 cpu_design.bd 是用于设计 cpu 外围结构的 block design 文件，LoonsgonTop.v 是由 noop-lo 目录下的源码编译生成的 verilog 文件，cpu_top.v 是一层对 block design 的 wrapper，用以将接口包装到符合大赛规定的接口。

4.2 设计演示结果

具体细节可以参见 score.xls，我们的 cpu 最高主频能够达到 73Mhz，跑分 42.09，但由于 vivado 的电路优化算法带有一定的随机性（本质上是随机算法），所以在每次综合实现的时候，并不总能到达 73Mhz，大部分情况下会因为时序关系不满足而导致负的 wns 值，因此我们将频率降到稳定的 71Mhz，经过多次综合实现确认，在这个频率下 vivado 的综合结果总能满足时序要求，但由于频率低了 2Mhz，跑分也相对应的低了 1 分，在 71Mhz 下，

cpu 的跑分为 41.13 分。但为了稳定考量，我们选择将频率固定在 71Mhz，同时由于我们组所有人的电脑均为 ubuntu 系统，在 windows 下使用 vivado 综合实现是否能满足 71Mhz 的时序要求我们并未测试。

5 参考设计说明

在整个 CPU 的设计实现过程中，我们参考了 riscv 的 rocket-chip 的一些设计理念，如布线方式，对 scala 高级特性的使用。由于二者指令集并不一致，因而除了一些设计理念之外，只有一些架构无关的部分可以稍作参考。

在 uncore 之中，cpu 使用了总计三个 IP 核，由 vivado 提供的乘除法器 IP 核，用于将 cpu 出口的访存请求综合起来的 vivado 提供的 crossbar 模块。由于除法器需要特殊的方式进行 reset，所以我们还额外使用了 vivado 提供的 reset 模块用来初始化除法器。