龙芯杯AmberCPU设计报告

中国科学技术大学Amber队
常文正、刘睿博、于硕、闫泽轩

1 设计简介

我们设计的 CPU 采用顺序双发射九级流水，实现了 龙芯架构 32 位精简版参考手册所要求的67 条指令、26 种 CSR 寄存器、16 种例外[1]。

采用 AXI-4 总线。使用 2 路组相联 8KB icache 和 2 路组相联 8KB dcache，采用写回、按写分配的设计，同时两个cache中都采取了 8 路全相连 1KB victim cache用于优化缓存失效带来的性能损失。并使用分支预测器以减少分支失败带来的性能损失。

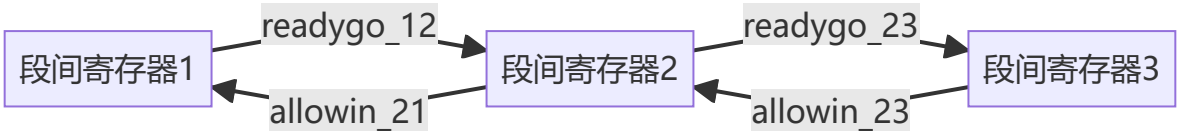
2 二、设计方案

2.1 （一）总体设计思路

> 阐明总体设计思路，即从系统顶层角度出发，概要性地描述整个系统的工作机制，所需要进行哪些设计、完成哪些功能。如果设计比较复杂，那么最好进行模块划分，把每个模块功能和接口的大致情况描述一下。

我们具体设计的cpu基于LoongArch32位精简版参考手册[1]，参考了姚永斌老师的《超标量处理器设计》[2]、汪文祥老师的《计算机体系结构基础》[3]、汪文祥老师和刑金璋老师的《CPU设计实战》[4]等书籍,并参考了开源项目clap[5]的译码器实现。

2.1.1 段间寄存器valid-ready握手协议



readygo_23和allowin_21由流水段传入段间寄存器的组合信号，由段间寄存器产生
readygo_23和allowin_21还与段间寄存器自身情况有关，这里假设自身准备好，如果自身没准备好，对外readygo_23和allowin_21均为0
readygo_23不能受allowin_23控制
伪代码表示:

```
readygo_23 =valid（valid表示自身情况,无阻塞的流水段逻辑恒为1,有阻塞的先0,有效时数据和valid同时为1）
allowin_21 = allowin_23 || ready（ready表示自身情况）
if(readygo_12 && allowin_23&&allowin_21)
    更新段间寄存器2
if(~readygo_12 && allowin_23&&readygo_23)
    清空段间寄存器2（所有控制信号置为0，防止上一周期的指令被重复执行）
if(~allowin_23||~allowin21)
    维持段间寄存器2（所有寄存器值不动，防止上一周期的指令被丢失）
```

注：这种握手对于一般一级段间寄存器已经足够，但是对于FIFO或者CACHE这种不定周期并且不能简单由组合信号阻塞的流水段还需要另外的精心设计来产生握手控制信号。这样的基本目标是满足指令PC不重不漏。

2.2 （二）FIFO模块设计

我们在取指段（IF）和译码段（ID）之间使用了FIFO作为PC和指令等信息的缓冲。

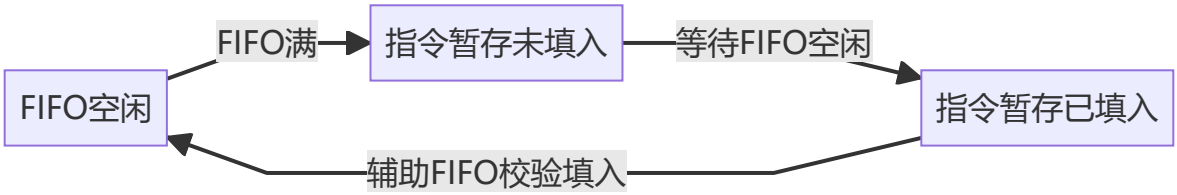
添加FIFO的作用是为了解决流水线前后段速率不匹配的问题，换句话说就是减少前后段Stall互相之间的掣肘，比如在流水线前段可能会出现IcacheMiss，在流水线后段可能会出现DIV/DcacheMiss。在这个时候
FIFO队列在不满时对于前段可以屏蔽后段产生的Stall继续取指，同理对于后段可以一定程度上减少前段IcacheMiss产生的暂停并且继续连续发射指令。

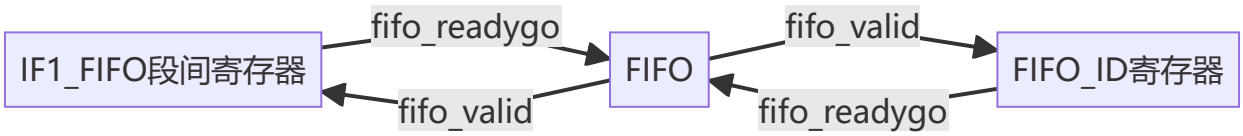
但是，尽管FIFO的设计看起来十分美好，FIFO在满时会产生严重的问题：丢指令。

为了避免丢指令，我们设计了状态机，利用FIFO队列引出的full/nearly/we等信号在即将丢指令的同时巧妙地获取到了即将被丢失的指令的情况下，先阻塞前端流水段，之后在队列有空时插进队列，然后才继续进行流水。

由于我们的Icache不能直接被Stall，并且TLB要经过寄存器传递信息，因此仅仅将FIFO满时的指令插入后可能前面阻塞的流水段仍然在取相同的指令，导致指令可能反而被填充两次FIFO。

经过仔细观察，在IF0_allowin为1时的fetch_pc是每次有效且与时钟周期一一对应（不 会因为allowin阻塞而在寄存器中暂存超过一个周期）的，因此在考察流水线长度后我选择了合适的一个相对小的FIFO作为辅助保证正确性的设计。





同其它流水段的allowin/readygo相比，FIFO本身本质上是一个新的流水段，因此使用了4个握手信号，分别提供给首尾的段间寄存器进行握手。

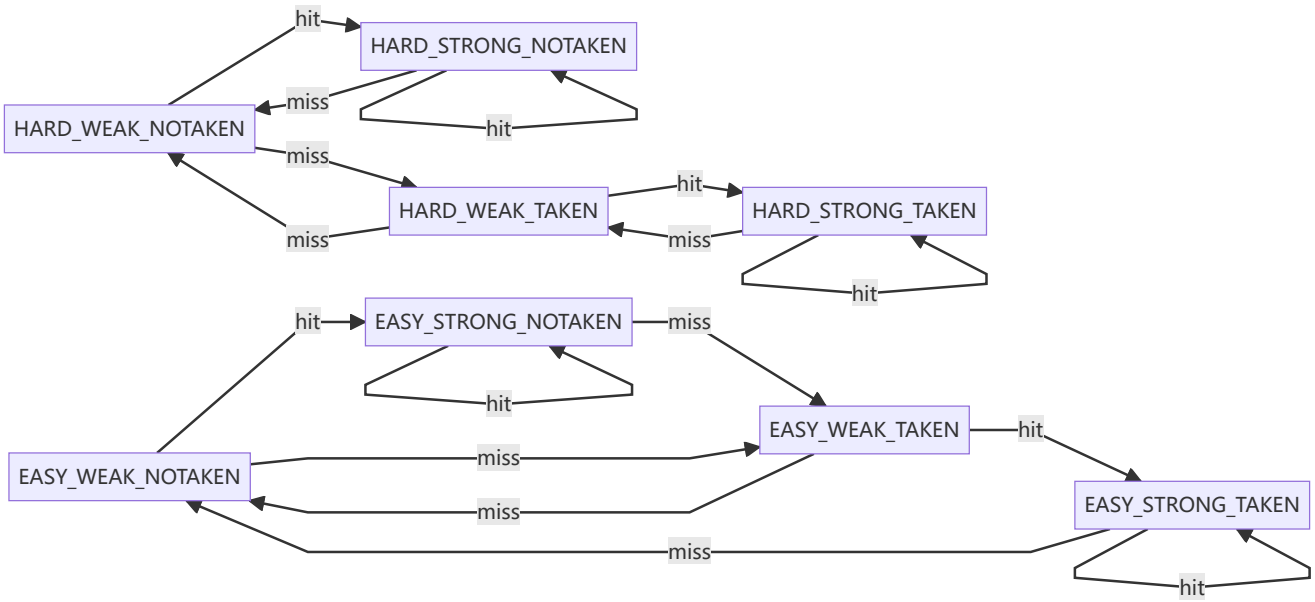
2.3 (三) 分支预测器设计

分支预测器使用了多种设计进行综合判断。

分支预测器作用是预测跳转指令的目标地址，但是困难在于PC更新时简单的分支预测器仅仅能够fetch_pc的信息预测下一条取指地址。同时如果使用查表的方法，为了权衡节约空间和区分不同PC的目的，我们在IF0段需要使用合适的寄存器数组长度来进行跳转与否，跳转地址的预测。

2.3.1 taken判断思路

1. 静态预测器：默认不调转，仅仅作为复位后的初始状态时采用。
2. 全局预测器:使用EASY_STATE和HARD_STATE两种状态进行竞争，两种STATE本质上都是2bit预测器，一种是预测失败就切换成相反的预测，另一种是给予一次额外的机会。两种STATE通过SCORE状态机来判断采纳哪一种。
3. 局部历史预测器：使用一个寄存器数组对于每一个PC对应的INDEX进行预测。
4. 指令类型前递辅助预测：使用取指后的预译码器（pre_decoder）来判断筛选一些无条件跳转，非跳转的指令，将译码的信息使用UMASK，BMASK保存，优先使用两个寄存器数组的掩码信息来判断跳转与否。



2.3.2 PC判断思路

使用DRAM的寄存器堆来存储预测PC。由于指令无论跳转与否，使用PC_relative方法寻址的跳转指令的跳转目标地址是固定的，而对于寄存器寻址的跳转指令又过于灵活，我们认为采用每一种INDEX仅对应一个目标地址的设计已经是比较平衡的设计。

2.3.3 INDEX拼接技巧

对于一个PC的INDEX来说，最自然简单的设计就是取低INDEX_WIDTH位,或者去掉低两位（PC对齐）或者低三位（指令连续双发）。

但是对于程序中跳转指令的空间分布特性可以认为跳转指令不会连续或者高密度出现，并且不同进程占据的空间在较高位会有显著区别，而这时的跳转指令目标地址又几乎不可能在低位部分重合。

因此我们选择的INDEX是{PC[19],PC[14:10],PC[5:4]}.目的是使用更少的PC信息来获得更高的指令命中率。

2.4 (四) EX阶段设计

2.4.1 EX段概述

EX阶段总计三级流水线，ALU指令在EX0段完成，乘法指令在EX0段与EX1段分两个周期完成，除法指令和特权指令在EX0段进入各自的状态机进行，并阻塞流水线，最后向流水线发送ready，然后流水线继续流动。

跳转指令在EX0段判断是否需要跳转并计算跳转目标地址，同时向分支预测器发送信息来协助分支预测，如果分支预测失败，则清空EX0段之前的流水线的指令，IF0段重新取指。

访存指令在EX0段向TLB发送虚地址，EX1段TLB向cache发送实地址，如果dcache命中，则在EX2段完成访存，否则阻塞流水线

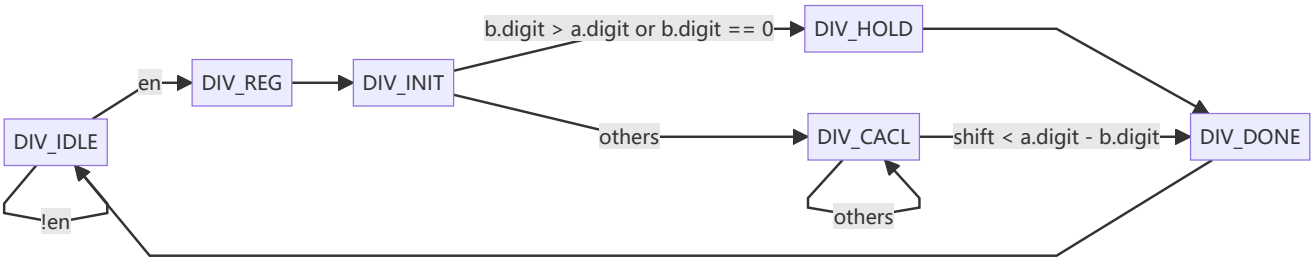
2.4.2 数据前递模块

为处理数据相关问题，EX1,EX2,WB段的数据如果完成则需要进行前递来减少流水线的停顿。

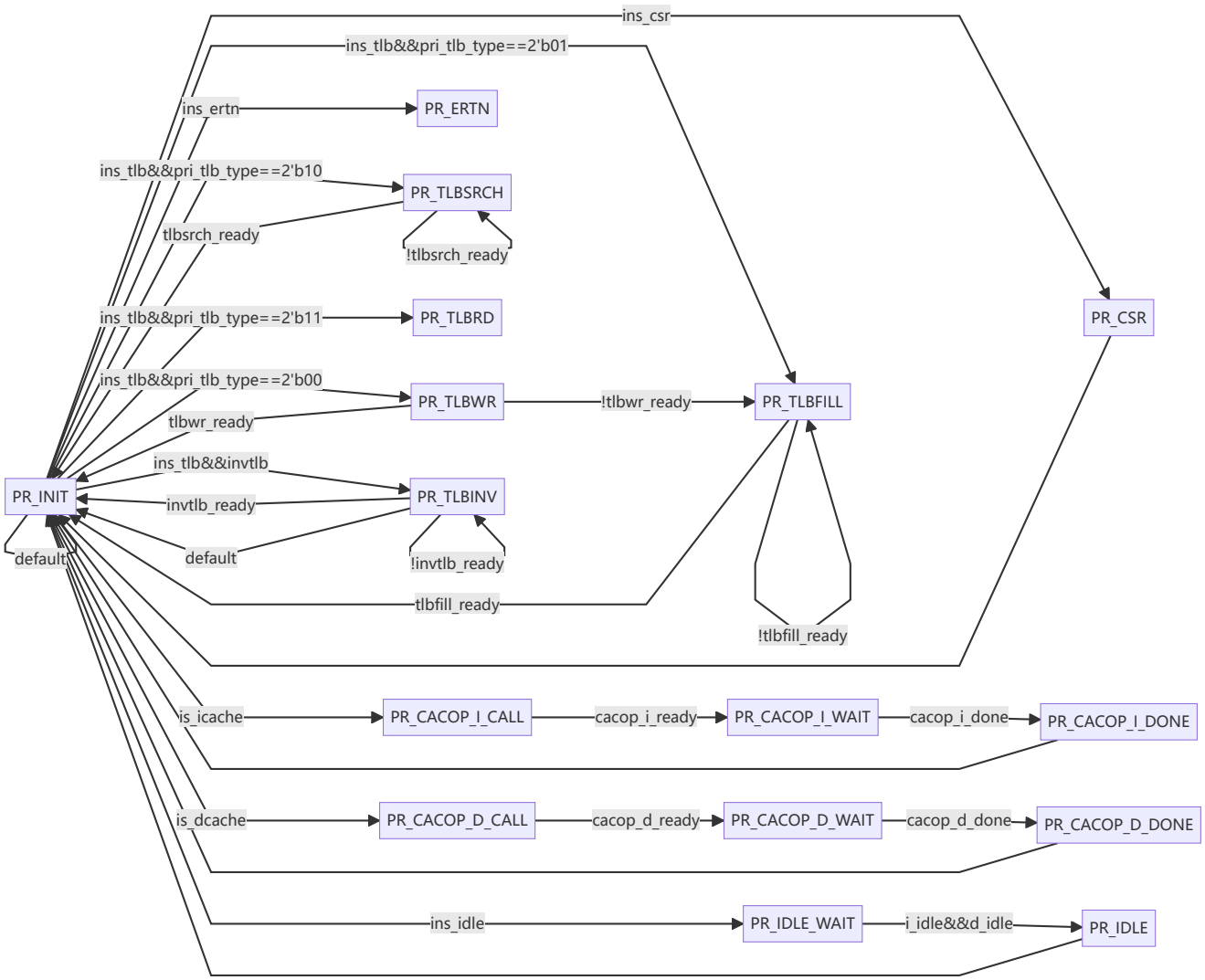
EX1后面的段间寄存器保存段间数据valid信号来确保数据可以正确前递，没有准备好的数据需要前递时，阻塞流水线，待valid置为1时再进行前递。同时为了保证写的数据可以立即被后续指令读取，寄存器堆采取写优先设计，保证数据有效

2.4.3 除法器设计

a(被除数) / b(除数) = x(商) · · · y(余数)



2.5 (五) 特权指令实现



2.6 (六) 异常处理

2.6.1 异常处理通用逻辑

例外产生：

例外可以在流水线的多个阶段产生。
取指与访存：取指地址错例外，访存地址错例外，地址非对齐例外，load操作页无效例外，store页无效例外，页修改例外，页特权等级不合规例外，TLB重填例外
译码：指令不存在例外
执行：系统调用例外，断点例外（由SYSCALL和BREAK指令产生，我们在EX段生成异常信号）
写回：中断 （直接将中断信号接入WB段进行处理）

例外处理时机：

例外均在WB进行处理，中断优先级高于例外，例外处理的优先级按照出现在流水线的时间，即取指>译码>执行，取指和访存时，TLB例外优先级低于非TLB例外优先级

例外处理过程

WB段检查exception信号和ecode信号，如果发现exception信号为1，则进入例外处理过程。触发例外时：

1. 将CSR.CRMD的PLV、IE存入CSR.PRMD的PPLV、PIE中，然后将CSR.CRMD的PLV置为0，IE置为0
2. PC存入CSR.ERA
3. 跳转到CSR.EENTRY(TLB重填例外跳转到CSR.TLBRENTRY)
4. 将例外编码存入CSR.ESTAT

ERTN从例外返回

CSR.PRMD的PPLV、PIE恢复到CSR.CRMD的PLV、IE，并跳转到CSR.ERA

2.6.2 例外判断逻辑

中断 INT：8个硬件中断由硬件产生，2个软件中断由特权指令写入中断位产生，核间中断来源于核外中断控制器，定时器中断在核内定时器倒计时至全0时产生

四条TLB内检查表项信息产生的例外，valid为0，dirty为1，或页表项为特权等级但当前状态为用户态

load操作页无效例外 PIL：执行load指令时TLB表项无效，valid位为0
store操作页无效例外 PIS：执行store指令时TLB表项无效
取指操作页无效例外 PIF： 取指时TLB表项无效
页修改例外 PME：取指或访存时表项dirty位为1
页特权等级不合规例外 PPI： 取指或访存时当前特权等级为用户等级，页为特权等级

TLB、icache、dcache检查的地址错例外，cache对地址是否对齐进行检查，TLB对用户态是否访问了特权模式地址进行检查

取指地址错例外 ADEF： 取指时访问了不对齐的地址或用户态访问了特权地址
访存地址错例外 ADEM： 访存时用户态访问了特权模式地址
地址非对齐例外 ALE: 访存是访问了不对齐地址

两条带有例外的指令SYSCALL BREAK

系统调用例外 SYS： 执行的指令为SYSCALL
断点例外 BRK：执行的指令为BREAK

译码检查指令是否合法

指令不存在例外 INE: 译码时发现不是合法指令

EX段检查当前特权等级和指令是否是特权指令

指令特权等级错例外 IPE：在用户模式执行了特权指令

TLB未命中

TLB重填例外 TLBR：取指或访存，TLB未命中，接下来挑转到TLBRENTRY进行重填

2.7 （七）高速缓存模块设计

2.7.1 设计综述

流水线 CPU 装配的指令高速缓存、数据高速缓存通过连接类SRAM~AXI转换模块，采取AXI总线协议与主存进行数据交换，以求获得更小的访存成本。

当 CPU 需要内存访问时，流水线会向 Cache 发起访存请求。如果请求的地址数据在 Cache 中，则 Cache 将在下个时钟周期内送出访问数据；如果请求的地址数据不在 Cache 中，则 Cache 进入缺失状态，将由 Cache 向主存发起访存请求，待请求的数据返回后再将数据返回流水线。

Cache 还需要处理流水线的 uncache 访存请求，在这里我们复用了一致可缓存地址部分的数据通路，使得强序非缓存的实现并没有无谓浪费更多的资源。具体实现方法为：对 uncache 的请求，强制令其 Cache Miss ，并调整访存的数据长度、大小，来做到精准访问一些外设对内存地址映射的地址空间。

参考超标量处理器设计，当ICache或DCache缺失时，会在向AXI访存的过程中将要被替换的块写入Victim Cache中，Victim Cache采用全相连的设计，在CPU访存中参与判断Cache是否命中，用于变相拓展Cache的空间，以提高Cache命中率。[2]

Cache 均采用**两拍式流水线**进行内存访问，保证了连续 Cache 访问命中时可以无需阻塞流水线，只需两拍顺序将地址送入 Cache 即可。同时， DCache 采用**写回写分配**策略，能够大幅度提升访存密集型程序的运行效率。

在 LoongArch 架构中，需要使用 CACOP 指令来维护 Cache 和主存之间的一致性。对于这类指令，我们依然将其视作访存指令进行处理，并复用了基本的数据通路。为优化逻辑，我们也加入了新的状态来专门处理 CACOP 指令。[4] [3]

2.7.2 ICache设计

基本参数：

- 数据总大小：8KB
- 路数：2 路组相连
- 单路行数：512 行
- 单行大小：64 字节
- 换行算法：LRU 算法
- 为契合流水线的双发射， ICache 一次将给出 2 个字的数据

2.7.3 DCache设计

基本参数：

- 数据总大小：8KB
- 路数：2 路组相连
- 单路行数：64 行
- 单行大小：64 字节
- 换行算法：LRU 算法

2.7.4 Victim Cache设计

基本参数：

- 数据总大小：Icache为16路
- 路数：多路全相连
- 单路行数：64 行
- 单路大小：64 字节
- 替换策略：计时器

有时候Cache中刚刚被替换掉的数据可能马上又要被使用，例如若一个CPU使用了2路组相连的DCache，而一个程序频繁使用的3个数据又恰好位于同一个Cache Set中，那么就会导致一个way中的数据经常被替换掉，然后又经常被写回Cache，这会导致处理器的执行效率大大下降，而为此增加way的个数通常又是不值得的，因为其它Cache Set未必有这样的特征，因此采用Victim Cache保存最近Cache替换掉的数据。

3 三、设计结果

3.1 （一）设计交付物说明

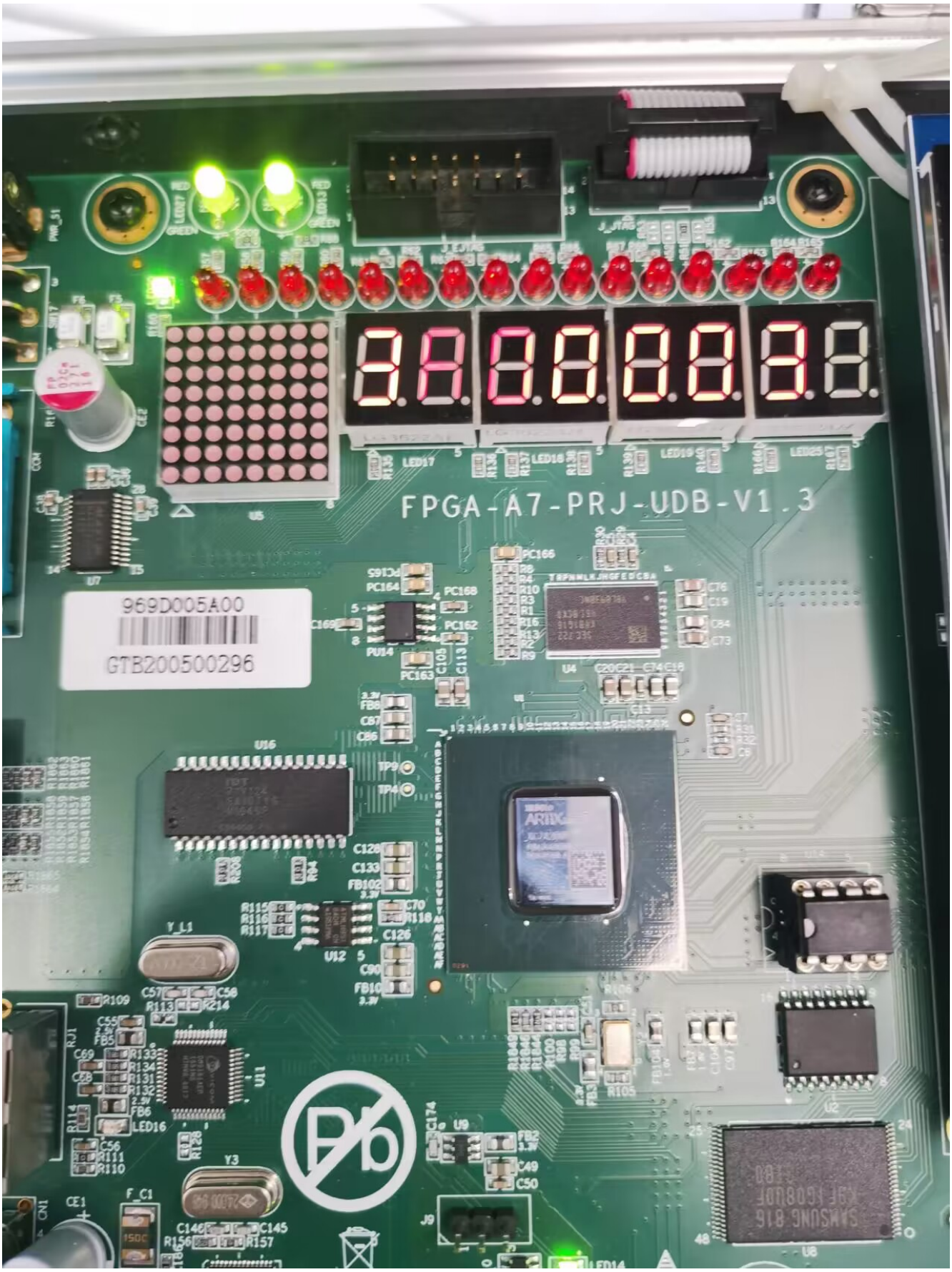
我们小组在时间十分有限的情况下夜以继日完成了CPU设计的主要工作，并完成了功能测试和性能测试的全部仿真测试，并完成了功能测试的上板测试，但是很遗憾我们在性能测试中实现还不够完善，部分性能测试无法通过，且不同个性能测试不能通过按reset键来进行重新测试，每个测试都要提前拨好开关后，保持要查看的测试的开关，重新烧板

预赛提交的作品的目录格式如下（submission/LoongArch_USTC_1_liuruibo）：

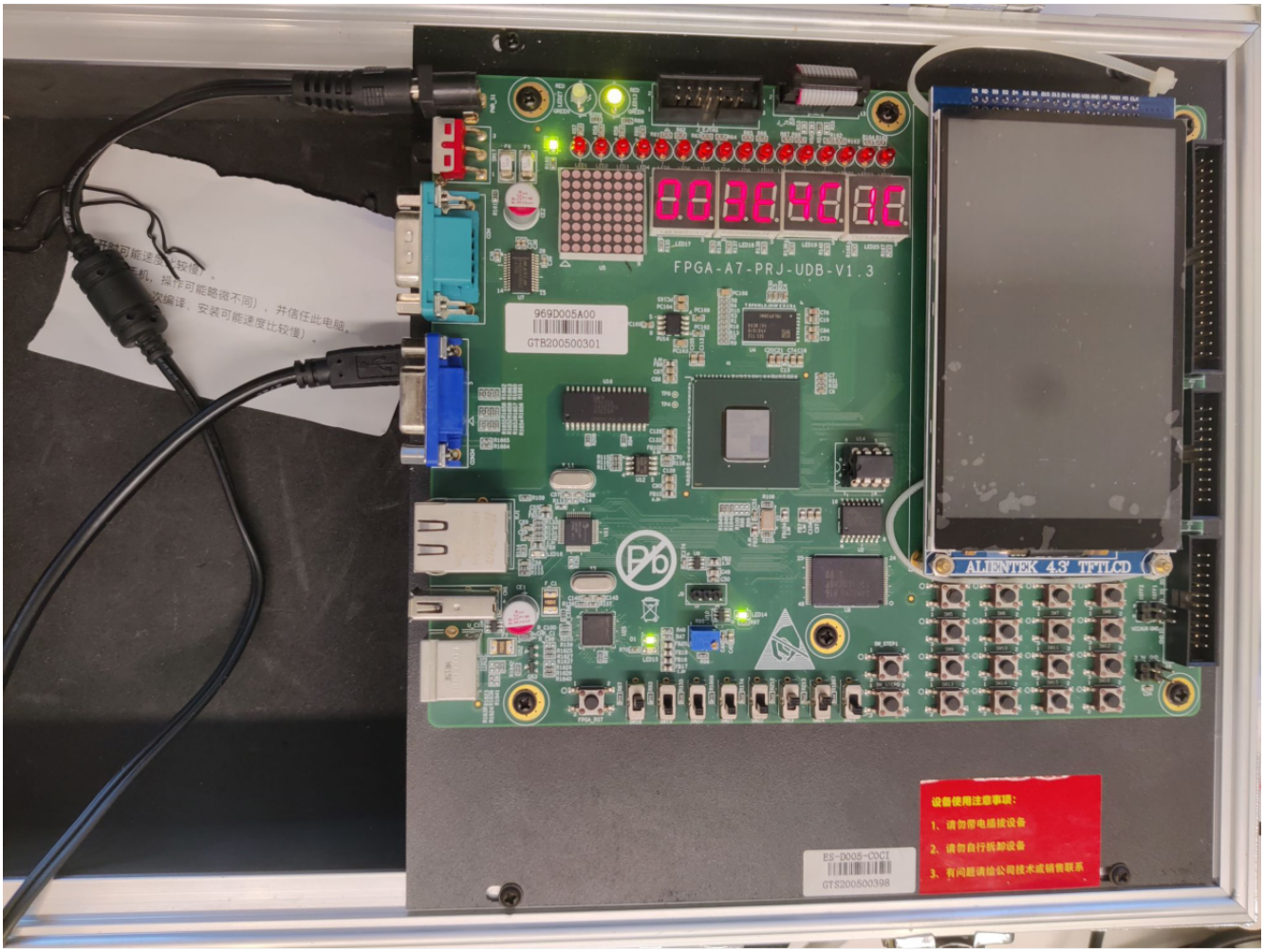
|-score.xlsx	**Excel表格，包含功能测试、性能测试得分的计算**
|-design.pdf	PDF文件，为myCPU设计报告
|--sram_src/	目录，SRAM目录
| |--mycpu	目录，存放SRAM工程新增的源码文件，我们直接实现了AXI接口，该目录为空
|--src/	目录，AXI目录
| |--mycpu	目录，存放AXI相关工程新增的源码文件，我们的代码均存放在此处
| |--perf_clk_pll.xci	性能测试的perf_clk_pll.xci。

3.2 （二）设计演示结果

功能测试上板结果，七段显示译码管显示3A（时间紧张，拍照效果略差），两个绿灯亮起



性能测试结果示例



4 四、参考设计说明

我们的AmberCPU中绝大多数设计都是完全自主完成的，但是也有一些设计参考了其他开源项目，这里列出了我们参考的开源项目：

项目名称	项目地址	说明
[clap][^2]	https://github.com/npz7yyk/clap	参考译码器的实现

5 五、参考文献

1. 龙芯中科技术股份有限公司 芯片研发部. 龙芯架构 *32* 位精简版参考手册[S/OL]. 北京: 龙芯中科技术股份有限公司. 2023, v1.03. https://www.loongson.cn/FileShow ↩ ↩

2. 姚永斌. 超标量处理器设计[M]. 北京: 清华大学出版社. 2014. ↩ ↩

3. 胡伟武. 计算机体系结构基础[M]. 北京: 机械工业出版社. 2021, ed. 3. ↩ ↩

4. 汪文祥 and 刑金璋. *CPU* 设计实战[M]. 北京: 机械工业出版社. 2021. ↩ ↩

5. npz7yyk. *clap* [CP/OL]. GitHub. 2022 (20220814). https://github.com/npz7yyk/clap ↩ ↩