

嵌入式处理器与芯片系统设计(H)

课程报告



实践任务： XXX

院 系： 微电子学院

专 业： 微电子科学与工程

单 位： 复旦大学微电子学院

完成日期： 2023 年 5 月 6 日

目录

[一、 体系结构原理介绍（简略） 3](#_Toc169984423)

[1.1 原理一 3](#_Toc169984424)

[二、 玄铁的IP分析与学习（详细） 4](#_Toc169984425)

[2.1 顶层模块gpfb 4](#_Toc169984426)

[2.2 模块tsm 6](#_Toc169984427)

[2.3 模块l1sm/l2sm 7](#_Toc169984428)

[2.4 模块gated\_clk\_cell 9](#_Toc169984429)

[三、 玄铁模块Chisel化 10](#_Toc169984430)

[3.1 端口Chisel化 10](#_Toc169984431)

[3.2 模块l1sm/l2sm 11](#_Toc169984432)

[3.3 模块tsm 12](#_Toc169984433)

[3.4 模块gpfb 12](#_Toc169984434)

[3.5 模块gated\_clk\_cell 15](#_Toc169984435)

[四、 集成仿真 16](#_Toc169984436)

[4.1 gpfb,l1sm,l2sm,tsm模块综合仿真 16](#_Toc169984437)

[4.2 pfb所有模块集成到玄铁上测试 16](#_Toc169984438)

[五、 小组分工说明（越详细越好） 16](#_Toc169984439)

[六、 问题与反馈、感想与建议 17](#_Toc169984440)

[6.1 Chisel化的技巧（可选） 17](#_Toc169984441)

[6.2 其他 17](#_Toc169984442)

**GPFB:Global Prefech Buffer**

# 体系结构原理介绍（简略）

## 原理一

对于步幅连续一致的数据流，具有显著的可预测特点，只要计算出该数据流的步幅，并以当前数据地址为基础，加上步幅即可预测下一步需要的数据地址。

针对该算法，需要解决三个问题：1.如何确定当前的数据流是步幅连续一致的；2.如何确定这样的数据流结束了；3.如果该数据流间断使用，比如对数组处理到中间处理了另外一个数据之后又返回数组处理，如何提高这种情况对步幅检查的效率。

GSDB与GPFB模块就是一个针对连续一致数据流的预取算法，GSDB支持步幅计算，步幅检查，步幅监视，并引入置信度解决了上述问题。Gpfb对gsdb传来的步幅进行计算，得到新的步幅，并进行跨页检查，与BIU和MMU进行交互，以决定是否对预取地址发出预取请求。

# 玄铁的IP分析与学习（详细）

GPFB是全局预取模式相关控制逻辑，GSDB计算得到步幅后，由gpfb来做预取请求，与缓存和主存，MMU等进行交互，对于系统中虚拟地址和物理地址的转换，需要对预取地址跨4K页进行处理。如果没有跨4K页，则直接使用所取地址的ppn即可，如果跨4K页，则需要请求MMU进行翻译，最终对物理地址进行请求。虚拟地址转为物理地址在gpfb内实现，同时gpfb对MMU的请求结果进行处理。gpfb的核心就是完成了对地址的翻译与对MMU和BIU交互的处理。

gpfb的input端口接受来自gsdb的一些信息，比如计算完成的stride，并有一些配置端口，接受配置寄存器控制以及当前的lsu的数据取址，gpfb的output端口分为两部分，一部分与MMU（Memory Management Unit），BIU(Bus Interface Unit)进行交互，一部分记录了预取请求是由l1还是l2发出的，后续这些记录端口信息给予顶层模块pfb处理。

## 顶层模块gpfb

GPFB内部主要是处理从GSDB传来的预取请求详情，包括预取的物理地址和相关的页面属性。它还处理与内存管理单元（MMU）的交互，确保预取请求符合页面访问权限和属性。在gpfb中主要是3个状态机l1sm,l2sm,tsm，含有两个gated\_clk\_cell时钟模块。

gpfb整体工作逻辑不难理解。假想一切为空开始，gsdb计算并且发现是有stride的数据流，则gpfb获得gsdb传来的strideh和预取请求。gpfb先进入tsm状态机，tsm状态机从IDLE状态激活，进入INIT\_INST\_NEW\_VA状态，该状态检查目前的指令访问地址加上stride后是否跨4K页，没有就进入JUDGE状态，并且一直在该状态，直到重置或者新的数据流不同的stride传入。tsm完成使命后，l1sm/l2sm开始激活，进入L1\_REQ\_PF状态，如果预取的地址没有跨4K，则直接进行预取，如果跨4K，会进入一个对MMU进行请求的状态，并根据返回的PPN进行处理。

综上可见，gpfb只是对gsdb传入的stride与目前的取地址进行处理，计算预取地址并与BIU和MMU交互，对跨4K的问题进行处理。

此外还做一些控制信号的计算。如计算有没有超出预取深度，并控制之后l1sm和l2sm，tsm的运行。L1缓存最大支持16个缓存行的预取深度，L2缓存最大支持64个缓存行的预取深度。根据l1sm和l2sm的reinit\_req情况产生gpfb的reinit。如果pfu顶层发出pop信号或者tsm处于运行状态且一些dcache命中或l2dcache命中，那么就产生pop信号。这将会导致状态机tsm与l1sm/l2sm被重置。

一些重要端口

cp0\_lsu\_icg\_en，cp0\_yy\_clk\_en 等：上网搜索是控制单元的时钟和功耗管理，但是Verilog中看起来并没有起到任何控制作用，他只在gated\_clk\_cell中使用，但该模块的Verilog有很大冗余设计，控制信号并不起作用。

ld\_da\_pfu\_act\_vld, ld\_da\_pfu\_pf\_inst\_vld,ld\_da\_pfu\_va 等：与加载指令相关的数据，用于预取请求的生成和管理。

pfu\_gpfb\_vld, pfu\_gsdb\_gpfb\_create\_vld, pfu\_gsdb\_gpfb\_pop\_req 等：控制GPFB条目的有效性，创建和弹出请求。

pfu\_gsdb\_stride, pfu\_gsdb\_stride\_neg, pfu\_gsdb\_strideh\_6to0：用于处理预取请求的步长信息。

pfu\_l2\_pref\_en, pfu\_dcache\_pref\_en：控制L2和DCache的预取使能。

pfu\_get\_ppn, pfu\_get\_page\_sec, pfu\_get\_page\_share等：提供物理页号和页面属性信息，用于预取请求的物理地址转换。

## 模块tsm



**Tsm的ASMD图**

tsm是一个初步检查是否跨4K页的状态机，tsm上网搜都没说全称是什么，我倾向于test state machine。因为看这个状态机的功能，只对一个数据流第一次预取进行处理，如果第一次预取地址在同一个4K页面内就进入JUDGE状态，并一直处于这个状态，之后的处理都由l1sm/l2sm负责。如果不在同一个4K页面就一直在INIT\_INST\_NEW\_VA状态里面，不停拖拍，让取地址变化，直到取地址和预取地址在同一个页面内。这个状态机似乎只做第一次处理。此外tsm还对l1sm/l2sm状态机进行把控，当pop信号过来，进入IDLE状态，l1sm与l1sm/l2sm均停止。

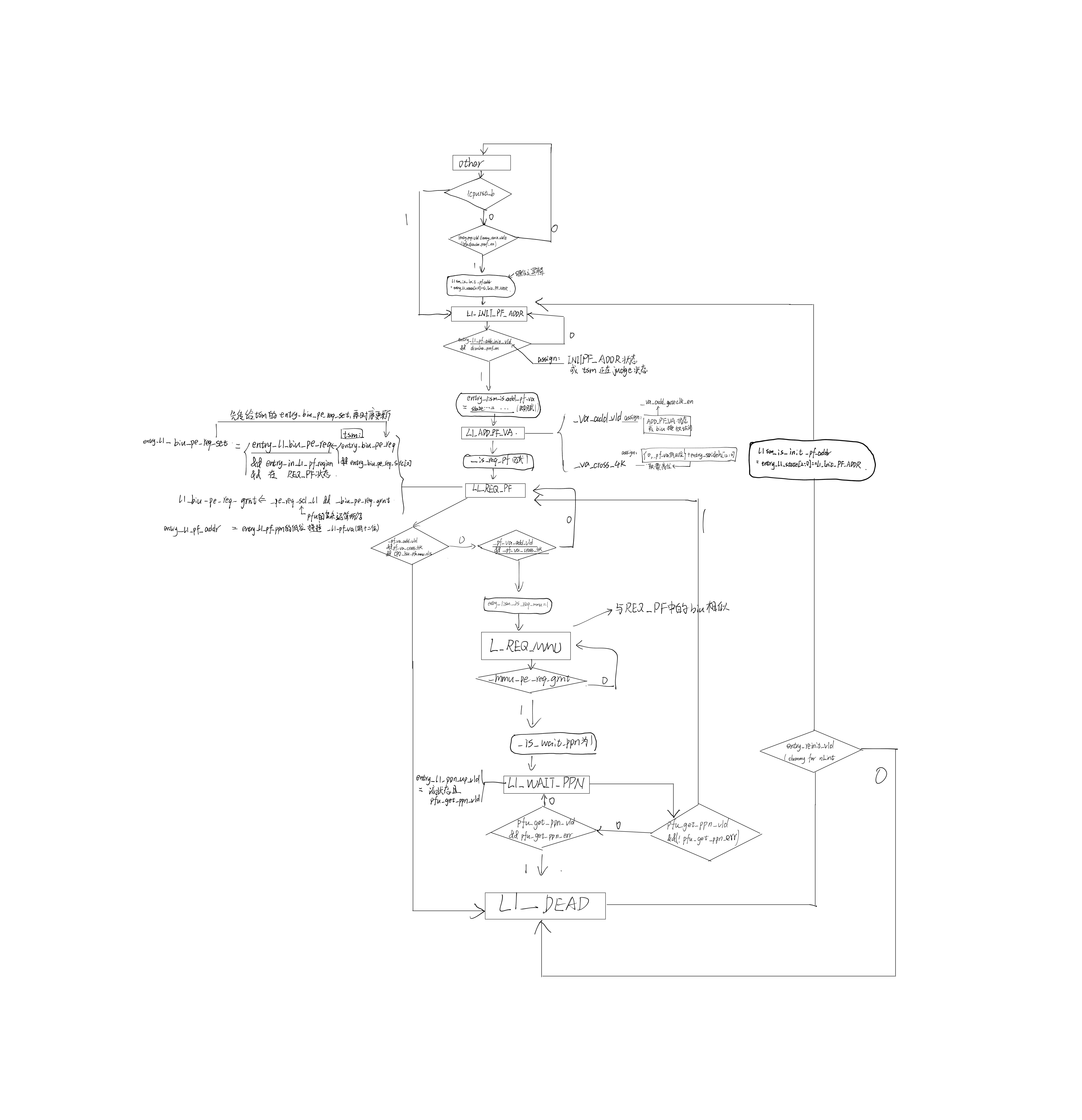
状态机描述：

**IDLE状态**：tsm处于不运行状态，当pop之后或者cpurst之后会处于该状态。当gsdb发送新的stride之后，会进入INIT\_INST\_NEW\_VA状态。该状态下时钟不被使能。

**INIT\_INST\_NEW\_VA状态**：当到达da阶段的指令出现cache miss的时候，将该指令的访问地址加上步幅得到inst\_new\_va，同时需要判断该地址有没有跨4k页，如果没有则跳转到 JUDGE 代表地址计算完成。如果有，就一直呆在INIT\_INST\_NEW\_VA状态。其会一直等待下一个load指令的新va，这时候就不再是跨4K页了，不仅减少了总体功耗，而且简化了设计复杂度。此外，这也体现了C910的数据预取不对跨页预取进行支持，跨页预取的可能性较低，支持跨页预取功能没有太大必要，而且物理地址翻译不断发出会占用MMU，降低整体性能。

**JUDGE状态**:当reinit时候，会转入INIT\_INST\_NEW\_VA状态，否则一直呆在judge状态。pop或者cpurst依然会进入IDLE状态。该状态下代表第一次预取地址计算完成，产生entry\_tsm\_is\_judge信号，进入l1sm或者l2sm的状态机准备向总线发出预取请求。

## 模块l1sm/l2sm



**L1sm的高清ASMD图**

l1sm和l2sm仅有微小的不同，以下以l1sm为例子讲解，最后对微小不同进行分析。

当gsdb完成stride计算后，l1sm会一直运行，不断的加步幅，产生新的va。关注entry\_l1\_pf\_va\_add\_vld这是加步幅信号，他在ADD\_PF\_VA状态或者biu授权后都可以为高。因此，当biu完成授权，就会自己加步幅生成新的va（这个自加va主要依靠biu什么时候授权）。此外该模块还会一直去检测新的va有没有和inst\_va（我认为这个inst指instant）差距达到最大预取深度，达到了就会产生一个信号暂停不断预取。

状态机描述

**other状态**：无论处于什么状态，cpurst,pop,reinit,！pref\_en都会导致回到INIT\_PF\_ADDR状态

**INIT\_PF\_ADDR状态**：当tsm处于judge状态，已经做好准备发出请求，并且使能了pref，进入ADD\_PF\_VA状态，状态机进入工作状态

**ADD\_PF\_VA状态**：这个阶段授权va的add，并计算是否跨4K,不判决，直接进入REQ\_PF状态

**REQ\_PF状态**：最核心的逻辑：处理biu\_pe\_req\_set的更新。在该状态下并且在pf的范围内，如果尚未对biu进行请求，则该信号会拉高，并在tsm中进行处理，使得对biu请求始终为高，直到得到biu授权后biu\_pe\_req才会再次拉低。当biu\_pe\_req拉低后，会再次将biu\_pe\_req\_set拉高，等到下一拍将biu\_pe\_req\_set拉高，继续去请求。这个逻辑横跨l1sm和tsm两个模块，核心就是保证在REQ\_PF状态而且预取在深度内时可以不断去对BIU发出预取请求。

**L1\_REQ\_MMU状态**：请求MMU去翻译跨4K情况，并进入WAIT\_PPN。

**WAIT\_PPN状态**：等待翻译得到的物理地址PPN，给了就再去REQ\_PF，没给就去DEAD。

**DEAD状态**：这个预取地址不可取。当预取要跨4K而且MHINT设置了不能使用MMU去翻译的话，直接挂掉，等待下一个新的想要预取的地址。

l1sm与l2sm的端口与连线区别

去分析gpfb顶层连线，可以发现一些区别：l1sm的entry\_l1\_pf\_va在l2sm中也有，一个是input另一个是output，其在顶层模块里面直接对连了。分析其中逻辑，发现：l1\_pf\_va对l1sm为output对l2sm为input，而且对于l2sm而言，l1\_pf\_va与inst\_new\_va两个端口的逻辑组合是l1sm的inst\_new\_va。而l2sm的entry\_l2\_pf\_va对应于l1sm的entry\_pf\_va，但是l2sm的该信号仅仅作为wire在内部起作用。l2sm继承自l1sm的entry\_l1\_pf\_va看作wire entry\_l2\_pf\_va，将其从端口剔除。

其l2sm的状态逻辑与l1sm一致，在预取深度上有所不同。物理意义上当然也有所不同，但是这些不同在gpfb里面没有更多体现，更多的差异化处理应该在更顶层的地方。

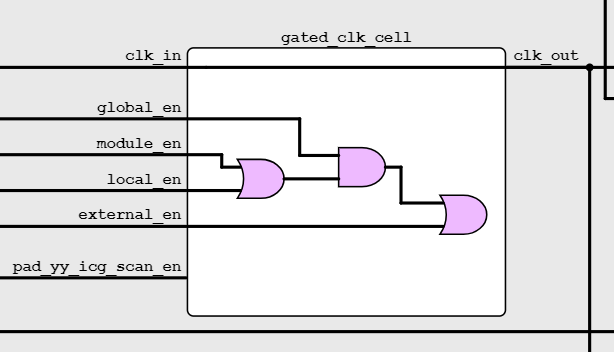
端口辨析

在阅读源码过程中发现entry\_l1\_cmp\_va\_vld与pfu\_gpfb\_l1sm\_va\_can\_cmp英文非常相近，很容易搞错两者，有一些细微差别：pfu\_gpfb\_l1sm\_va\_can\_cmp表示l1sm状态机处于运转（即在那几个重要状态中），而entry\_l1\_cmp\_va\_vld不仅表示l1sm在运转而且得到了biu授权，可以再次计算新的va，即entry\_l1\_pf\_va\_add\_vld开放。

注：l1sm的状态机图较大，已经是高清图，可以放大观察。我也将其pdf附上，更为清晰。

## 模块gated\_clk\_cell

通过看仿真图和分析Verilog源码，发现其实就是clk\_out与clk\_in直连，其他端口在该模块内做了运算但是没有引出，很奇怪为什么要做这样一个模块。



看这个图像是一个冗余设计。但是我上网查发现是为了电源管理，这里并不明白，看他的源码也难以理解，应该要结合其他工具分析或者后仿真的时候有用？

注：以上gpfb,l1sm,tsm在C910源码阅读学习周已经做过一次整理，但是使用pad手写，画了tsm与l1sm的详细状态机转换图和一些端口分析。已经转化为pdf并附上，比较详细，并附有一些对端口的理解和分析，略有杂乱，可能逻辑有瑕疵。仅以证明努力与成果，仅供参考。

# 玄铁模块Chisel化

遵循我对代码Chisel化的过程进行讲解，先将端口Chisel化，再对最底层模块l1sm/l2sm,tsm进行Chisel化，最后讲解顶层模块Chisel化过程。各个模块中存在有不同的技巧和问题。

## 端口Chisel化

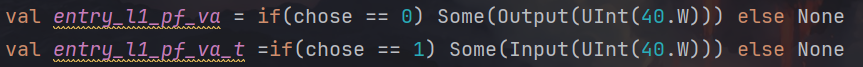
首先将各个模块的端口转为Chisel语言描述并放入package中，名为IOinst，之后在其他模块中只需要import IOinst即可导入需要的端口，如果要对各个模块的端口进行处理，可以集中管理，很方便。而Verilog中则无法方便做到重用端口名，SystemVerilog中虽然有接口，但也比这种更加接近硬件，并不够抽象化。

在Verilog端口转化为Chisel时，我将有明显控制意义的端口声明为Bool类型，有数据意义的声明为UInt等，将reset相关信号声明为AsyncReset类型，将clock信号声明为Clock类型。其实所有的信号都可以声明为UInt类型，只是信号长度有区别而已，在后面的模块内直接.asBool或者.asClock等即可。但是这样声明信号更有助于在后续书写代码时候进行实时检查，对于设计而言可以帮助理清端口性质，有助于设计正确。

端口采用Bundle，将所有io端口都放入一个Bundle中，如gpfbIO就是一个包含gpfb模块所有IO端口的Bundle，再将所有wire放入另外一个Bundle。此外对于reg我也放入一个Bundle中做了处理，但是reg的声明需要在时钟域内才可以，如果要将Verilog转化为Chisel，需要多时钟域，并在不同时钟域内声明例化对应的reg，所以我仅仅在Bundle中说明了reg的位宽，这样如果要调整reg位宽可以在Bundle中直接修改，而不需要再进入对应模块中搜索修改。

此外，因为l1sm与l2sm的极其相似，我在声明l1sm/l2sm的端口的时候，分析了l1sm与l2sm的差别，发现l1sm的entry\_l1\_pf\_va在l2sm中也有，甚至一个是input另一个是output，在顶层模块里面直接对连了。分析其中逻辑，发现l1\_pf\_va对l1sm为output对l2sm为input，而且对于l2sm而言，l1\_pf\_va与inst\_new\_va的功能作用之和为l1sm的 inst\_new\_va。l2sm的entry\_l2\_pf\_va功能等于l1sm的entry\_pf\_va，但是l2sm仅仅作为wire在内部。l2sm继承自l1sm的entry\_l1\_pf\_va看作wire entry\_l2\_pf\_va，将其从端口剔除，同时加上一个新端口entry\_l1\_pf\_va\_t，用来和inst\_new\_va组合为l1sm的inst\_new\_va。

基于以上分析，我在l1sm的端口声明中动了端口声明方式，io端口中：



而wire端口中：



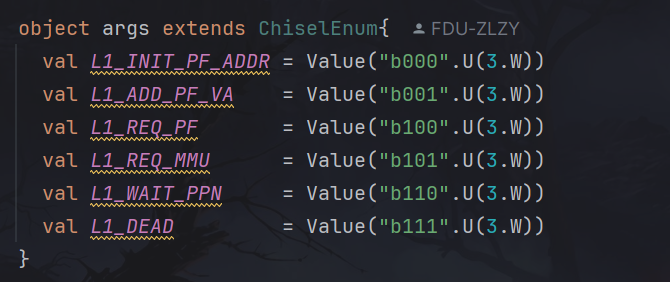
其中端口被l1sm使用时候chose为0，l2sm使用时为1。这样通过使用scala中的some类型，对端口进行参数化，并在l1sm模块中协调使用，最终实现了参数化生成对应的模块。

## 模块l1sm/l2sm

L1sm与l2sm是非常相似的两个模块，我先写了l1sm的Chisel代码，之后在l1sm的基础上参数化，通过指定参数可以生成l1sm和l2sm两种模块。以下主讲l1sm的Chisel化思路。

L1sm是一个状态机，首先的问题是如何用Chisel写一个状态机。

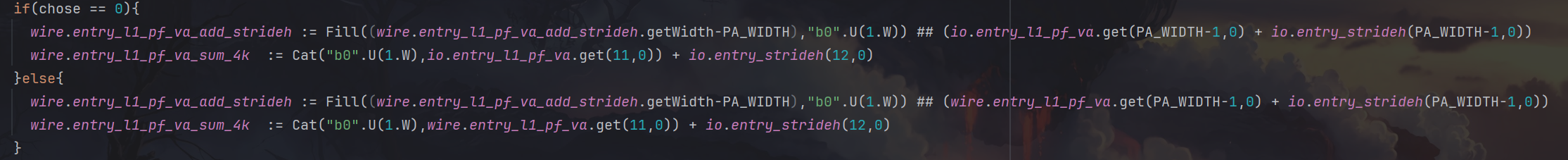
第一个要解决的是enum问题，如果是直接设计，可以用scala生成enum的形式直接生成状态名的数字，而考虑到最终要放回系统中，最好状态名对应的数字与原来Verilog一致。（其实仔细分析，状态名对应数字不一致也没问题，state相关逻辑保持一致就可以）于是使用ChiselEnum类，定义相关的状态名。



此后写状态机逻辑，首先使用withClockAndReset指定一个时钟域的时钟和复位信号，在其中例化声明状态机寄存器state。之后用when，.otherwise，switch，is来实现状态机。Switch检测state目前状态，搭配when实现逻辑判断下一状态，并将state赋值。状态机的写法有八股文之感，当确定设计思路后，只需要结合swich和when这些结构即可。注意switch没有违法的default分支处理，如果state错误变成一个没有指定过的状态代码，则没有办法。也许可以将所有的违法值放入一个list，如果检测到state处于其中之一，则将state复位到初始状态。

此外是时序逻辑和组合逻辑的Chisel化。这部分主要是Chisel的基本使用，没有大的创新点。Verilog中使用了PA\_WIDTH配置了连接逻辑，实现了一个端口或者wire中的部分位被赋值，其他位不做处理，但是Chisel中不支持这样对部分位赋值的处理。个人认为，其实这种问题是伪命题，一个信号声明出来一定需要每个位都用上，高位这里没有处理，那么在整体的代码中也一定处理了，只需要将所有的处理综合起来，一定是可以直接对信号所有位处理。Verilog中如果高位没有用到，最终就是完全悬空态，仿真器会将没有用到的信号优化掉。而Chisel既然更加高级，完全可以在Chisel设计的时候就优化掉，直接在Bundle中进行端口位宽参数化。

最后是对l1sm与l2sm的差别分析，我先设计了l1sm的Chisel代码，之后通过分析相关不同端口和其中不同的组合逻辑区别，我在l1sm的Chisel设计中又使用对some类型的.get方法。当chose为0时，此时要生成l1sm，用if判断并生成对应逻辑，当chose为1时，此时生成l2sm，生成对应逻辑。注意some类型必须要用get方法才可以进行处理。例如：



## 模块tsm

Tsm比较简单，遵循上述l1sm中对状态机的设计思路，首先将tsm的状态机逻辑实现了。之后根据Verilog的代码将组合逻辑转为Chisel，always块中的寄存器更新用时钟块完成。

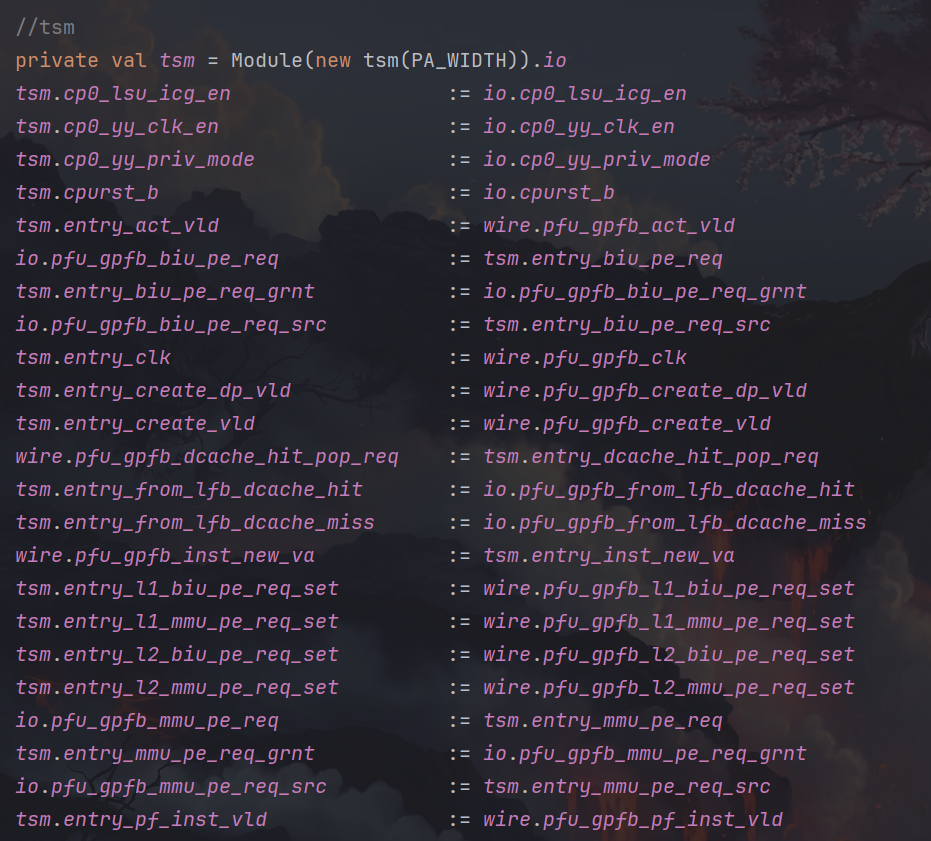
## 模块gpfb

Gpfb的组合逻辑与时序逻辑转为Chisel也比较简单。这里讲在顶层gpfb中的例化问题。

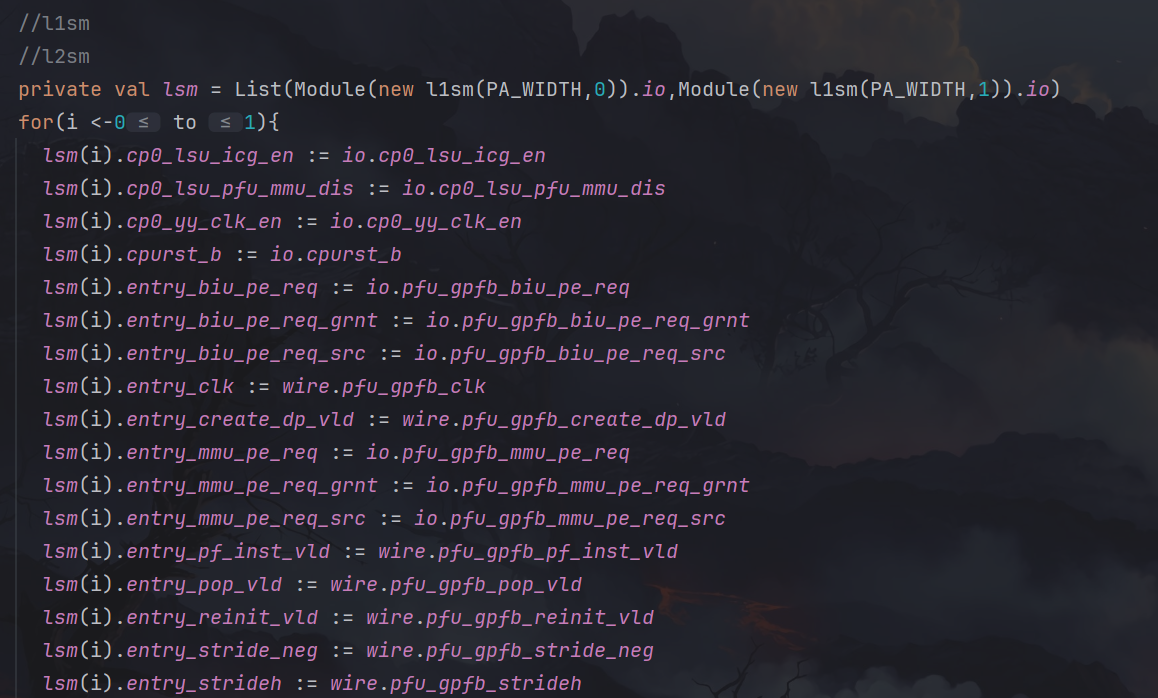
对于相同的模块批量例化，可以是用Seq.fill将需要的模块例化多次并转为序列，再将这个序列变为VecInit类型，之后可以通过索引对其操作，比如进行接线连接。例如gpfb中例化了两个gated\_clk\_cell，通过for对相同的端口进行连接，再通过if进行差异连接。



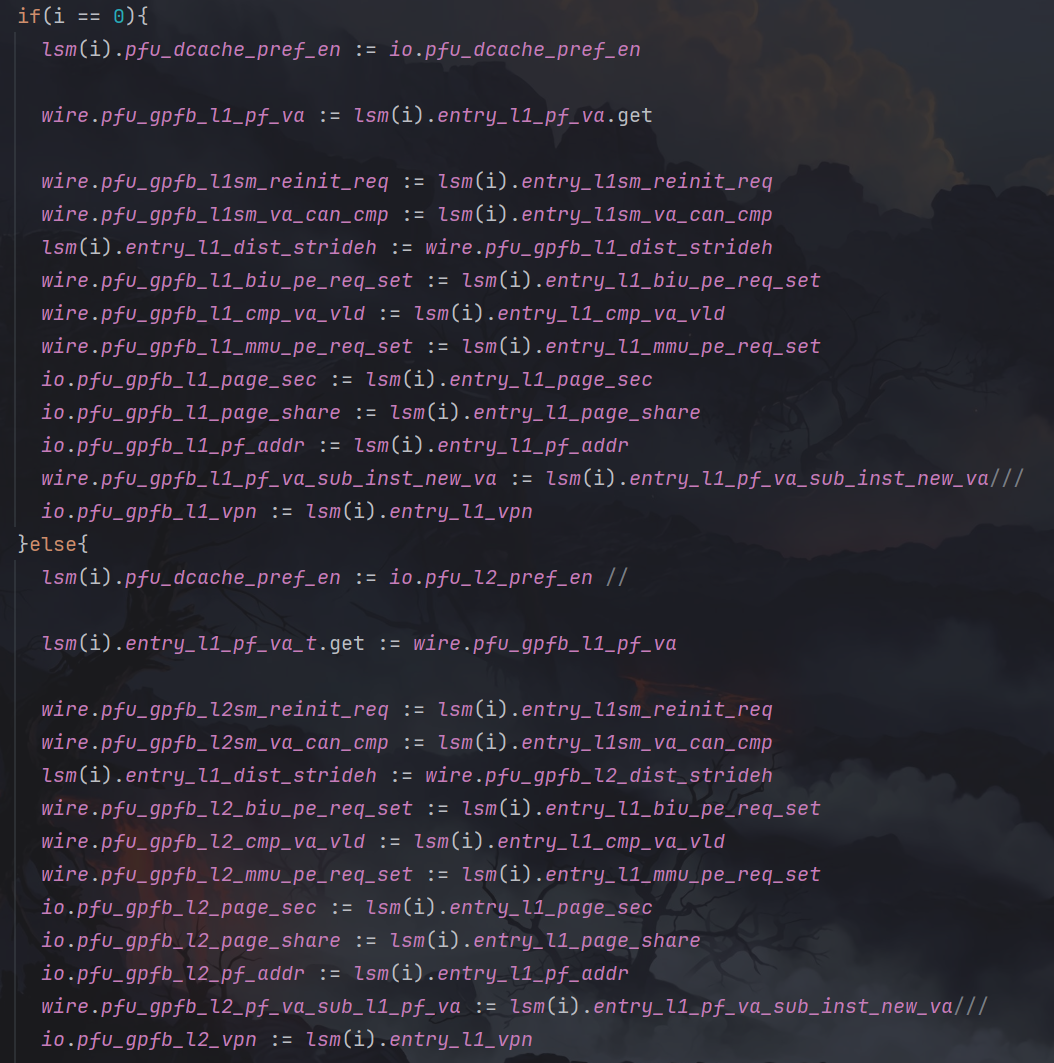
对tsm的例化比较简单，是用Module和new例化模块之后将对应端口连接即可。例如下图，仅截取部分。



涉及到l1sm和l2sm的Chisel实现参数化问题，在例化l1sm和l2sm时需要更多小心，主要在端口连接上。在例化两个模块的时候遇到了问题，采用上述VecInit(Seq.fill(2)(Module(new gated\_clk\_cell).io))这样的方法的时候，在发射为Verilog的时候会报错，转为List(Module(new l1sm(PA\_WIDTH,0)).io,Module(new l1sm(PA\_WIDTH,1)).io)这样的形式。最后也可以方便量化多个模块并且连接。例如



之后比较复杂和需要注意的主要在分析Verilog原来的连接逻辑上面。分析可以看到l1sm和l2sm的许多input端口是一致的，直接用for连接，如上图。之后按照之前对l1sm和l2sm端口不同之处的分析，对其差异化连接。注意对于some类型的端口的需要用.get方法连接。



最后，是模块名的问题。为了让生成模块的名字和原来Verilog的名字相对应，需要加上一句override def desiredName: String = s"ct\_lsu\_pfu\_gpfb\_t"，在l1sm/l2sm的Chisel代码中加上的是override def desiredName: String = s"ct\_lsu\_pfu\_pfb\_l${chose+1}sm\_tmp"。这个可以修改最终发成Verilog的模块名字，而且使用了scala的插值字符串，对于像l1sm/l2sm这样的参数化模块非常有用。

## 模块gated\_clk\_cell

这个模块完全按照Verilog转化为Chisel是比较容易的。但是在发射为Verilog之后，Chisel会进行优化，导致与冗余逻辑相关的端口全部消失，最后在嵌入到系统上会出现错误。不得已为了能够集成到系统上仿真，使用了黑盒。黑盒只需要声明端口即可，Chisel不会对这部分进行优化。

# 集成仿真

## gpfb,l1sm,l2sm,tsm模块综合仿真

仿真：

## pfb所有模块集成到玄铁上测试

正文。。。。略,需要小组协作写这个

注意：

1. 需包含测试结果截图
2. 要求未经改动的Chisel化的代码测试结果与玄铁C910源代码的测试结果一致。
3. 如果对架构有所改动，需要与玄铁C910源代码的dhrystone和coremark跑分进行对比，并对结果进行分析。

# 小组分工说明（越详细越好）

|  |  |  |
| --- | --- | --- |
| 小组成员 | 学号 | 分工 |
| 赵龙征洋 | 22307130163 | 1. 开学第4周加法器设计汇报 2. gpfb,l1sm,l2sm,tsm的源码阅读 3. gpfb,l1sm,l2sm,tsm的Chisel化 4. 数据预取小组Chisel化结果pre 5. gpfb的生成Verilog集成到玄铁C910进行测试 6. 5月31日最终报告Chisel化进度与跑分情况 |

# 问题与反馈、感想与建议

## Chisel化的技巧（可选）

在chisel转为Verilog中遇到了一些细节问题，同时对于chisel化的一些技巧有收获。

1. Chisel不支持部分位赋值的问题：（即左值不允许取下标的问题） Chisel支持位选读的写法，但不支持位选写的写法。我觉得设计这种语言机制其实没有问题。Verilog使用位选写的语法，如果是wire类型变量被位选写，那么没有提及的位其实被认为是高阻态了。我们可以在Chisel中直接实现在端口定义的时候就参数化，位选的宽度直接在声明端口的时候就算清楚。（但其实我在Chisel化过程中没有这么做，而是对于没有提及的位用了补0的手段，实现方法是用的##拼接。）

1. 使用Vec实现部分赋值
2. 使用##进行赋值
3. 使用掩码进行部分赋值

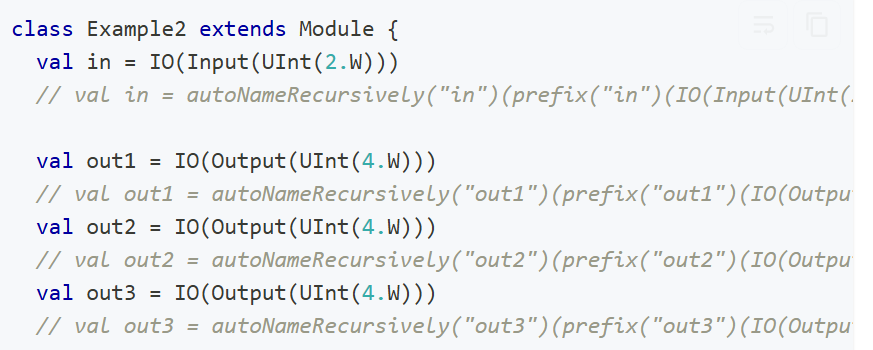
2. 生成Verilog时候的端口名问题：当发射生成Verilog的时候，模块端口名字会被自动带上io\_前缀等情况。

1. 分析:我翻阅了Chisel的cookbook，以下是cookbook的例子。对于有Bundle的情况：



可见结果有io\_。

而对于无Bundle的情况：



可见结果无io\_。

个人看他的意思是：Bundle内部声明并赋值的时候加了一个前缀，这个前缀就是我们要的端口名字，而当val io=IO(...)的时候，又会加一个前缀，这个就是可恶的io\_。

但是如果我们的赋值和声明不用Bundle，就不会有可恶的io\_了。中间变量应该也是一样，因为在嵌入到系统进行仿真的时候这个不是有影响的问题，我没深入分析，应该一样的

1. 解决方法：
2. 加一个掩层（proxy代理层，decorator装饰器），掩层的端口名字不带io\_，掩层和生成的顶层Verilog文件进行端口连接就可。其中连接关系就用vim或者正则表达式生成一下，非常快。
3. 顶层不要用Bundle，全部用val a = IO(Input(Bool()))类似的形式来例化，我觉得太笨了，不过也可以用一些trick，比如for、list.map等看起来正常一些，但还是觉得苯。
4. 用awk将生成的所有输入输出端口的io\_去掉。但是有风险，可能会导致一些信号名字被错误处理，导致逻辑问题。负责pfb的张乐然同学就曾受此困扰。
5. 用noprefix[函数]，这是我看到的一个去除前缀的巧妙方法，但是，所有能找到的例子都只针对中间变量，官方也没提供相关的示例。我试了，行不通，不知道谁有对这个东西的更深理解。
6. 不知道有没有这种方法：通过调整编译参数，指定要求发射生成Verilog的时候，端口不要加前缀。我觉得这个是最好的解决方法。但是翻看了很多网站也没有提供这方面解决方法。

很奇怪的是，官方竟然对于这个Bundle+IO例化端口名字如何去除io\_没有任何例子。可能是我们这个需求太奇怪了，居然用写好的Verilog转Chisel，还“奢望”发射生成的Verilog与原文一致！可能官方也没有想到会有这种问题。

3.端口被优化问题： 发射生成Verilog时，一些冗余逻辑被综合掉了。我本来还觉得这是很正常也很好的功能，但是当我们需要嵌入到系统中做仿真跑分的时候，就非常不友好了。

没有找到相关的解决方法，希望有一种编译的时候指定选项”是否优化模块输入输出端口“。从此可见，C910的设计在某种程度上很奇怪，存在有冗余设计，但是这种冗余的地方极其不合理——在gated\_clk内部居然做了组合逻辑但是不连出。我最后用黑盒例化了gated\_clk，Chisel不会动这个黑盒，于是和这个关键冗余设计来源的端口就没有被优化了。

## 其他

对于gpfb，我将自己的Verilog分析总结，Chisel平台，小组Chisel化pre报告，最终报告都已放入我的github中。

SSH方式：[git@github.com:Echoro/Chiselize\_C910\_gpfb\_with\_template.git](mailto:git@github.com:Echoro/chiselize_C910_gpfb_with_template.git)；

HTTPS链接：[https://github.com/Echoro/Chiselize\_C910\_gpfb\_with\_template.git](https://github.com/Echoro/chiselize_C910_gpfb_with_template.git)；

感想：

通过本荣誉课程，我通过自学掌握了chisel，并加深了对verilog的理解，从中受益良多。为我提供了今后对芯片设计方向的重要技能。在verilog转为Chisel过程中，对chisel是一个高级的语言理解更加深刻，其参数化和抽象化的能力非常强大。在学习chisel过程中也了解到了scala语言，对函数式编程和面向对象有了更多体悟。

反馈与建议：

在学习chisel过程中，其实所给的教程我并没有太多的去看，而是买了一本书直接去读。书名《敏捷硬件开发语言Chisel与数字系统设计》，也许也可以给下届同学推荐本书。书中内容和助教推荐的网站有很多相似之处，更加系统，更重要的是可以方便做笔记。如果可以发给同学一些源代码和原文的markdown文件或许更好。有的教程太过繁琐，重点不突出，也无法做笔记，实在是一种对学习效率的降低。

将verilog进行Chisel化我觉得是极好的，但是个人体验，在chisel化一开始是对同学理解chisel有帮助的，但是在更多模块的chisel化过程中并不友好。比如我在chisel化gpfb与l1sm,tsm过程中，只有刚开始的模块l1sm是我收获最多的，其他的更有搬砖的感受。我建议可以减少chisel化的工作量，而是让同学们在verilog转chisel任务完成后进行独立的chisel开发，可以用chisel写一个简单的五级cpu这种项目，也可以让同学对阅读的论文进行复现。

助教给的虚拟机用起来不如Linux更好用。开学去拷贝虚拟机也比较费时间。我建议直接给大家Windows配置chisel环境的方法，只需要做好一个有示范的github仓库，让大家在自己电脑下载好JDK与IDEA，然后直接拉取仓库就可以。我在本学期Verilog转chisel过程中就使用Windows的环境完成的，一些配置我写在了chisel化pre的文稿里面，希望可以有用。