Chiselize C910 GPFB

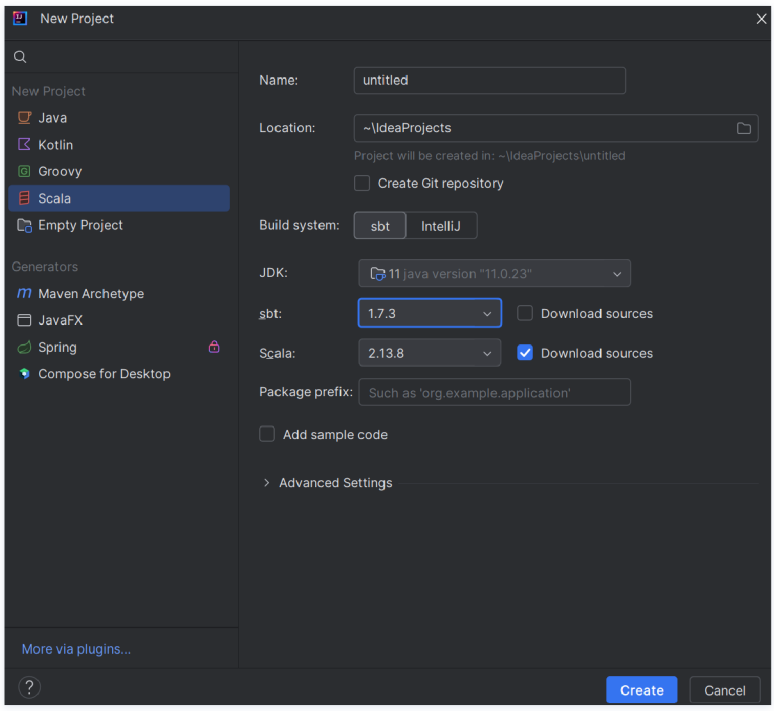
按复旦大学《嵌入式H》的要求，我自学了Chisel，并学习了C910数据预取的GPFB模块，其含有gpfb、`l1sm`、`l2sm`、`tsm`以及`gated_clk`模块。然后我对其Verilog源码实现了Chisel化，并成功发射生成Verilog代码。

目前我组完成了各个模块（除了PFB）的分别测试。从结果看有的coremark分更高了，有的更低了，有的还需要优化。

环境配置

课程给的虚拟环境运行在个人电脑上会出现卡顿等问题，而且文件传输和交流非常麻烦，不利于提高Chisel化效率。我们上网搜索和学习了如何在Windows环境下使用IDEA，并实现写Chisel和发射Verilog。

1. 下载IDEA社区版本，它是免费的。张毅帆成功破解过收费的企业版。
2. 下载JDK 11，因为这个版本比较稳定。我没试其他的版本。JDK 11需要去官网下载。之前用了JDK 22版本，会有一些奇怪问题。
3. 一定要指定好Scala版本，要和需要的Chisel版本、SBT构建文件中指定的版本适配。



4. 从GitHub克隆一个Chisel工程模板，例如，再运行一下SBT构建就OK了。
5. 发射生成Verilog：只需要替换`new`后面模块名字，并且`val PA_WIDTH`可以任意删除。只是示例。

```scala
class SC0076TV extends AnyFreeSpec with ChiselScalatestTester {  FDU-ZLZY

  val PA_WIDTH: Int = 32

  (new chisel3.stage.ChiselStage).execute(
    Array("-X", "verilog"),
    Seq(ChiselGeneratorAnnotation(() => new gpfb(PA_WIDTH)), TargetDirAnnotation("Verilog"))
  )
}
```

Chisel化过程中的一些收获

1. 时钟域：
   可以使用一个 `withClockAndReset` 块，块内声明所有和单个时钟相关的寄存器，接收块返回值。之后该寄存器和外部时序逻辑混用。

2. 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

3. 模块名操作：
   使用了 `override def desiredName: String = s"ct_lsu_pfu_pfb_l${chose+1}sm"`，重载了名称，用到了Scala的字符串插值语法。实现了在例化两个 `l1sm` 的时候根据参数生成不同模块名的实例。

4. **参数化生成不同的模块**：
   个人认为，在实现Chisel过程中，这是最能展现Chisel强大之处的特性！根据对 `l1sm` 和 `l2sm` 的分析，两者的逻辑有大量的重合，仅仅在部分端口名字和信号连接上面有不同。针对这种情况，有两种解决方法：

   1. 先写一个基础模板，将所有相同的逻辑写上去，然后 `l1sm` 和 `l2sm` 都继承基础模板，再重写相关的端口名字和部分不同的逻辑。
   2. 写一个综合模板，根据输入的参数不同生成不同的模块。**我就是用这种方法**。首先写了一个 `l1sm` 的综合模块，根据传入的参数 `chose` 来决定生成什么样的模块。

5. 可有可无的（Optional）端口问题：
   此问题一般出现在参数化生成模块的情景中。综合模板为了根据参数生成不同模块，需要对端口和逻辑进行可选化设计。使用Scala的 `Optional` / `Some` 语法，使用到该端口时候要用 `.get` 获取。
   （网上说也可以用0位宽，但是0位宽的意义仅限于生成Verilog时可以将一些0位宽端口综合掉。Chisel反而不适合0位宽，因为0位宽端口的连接逻辑容易出问题。）

6. 快速/批量模块例化与端口连接：

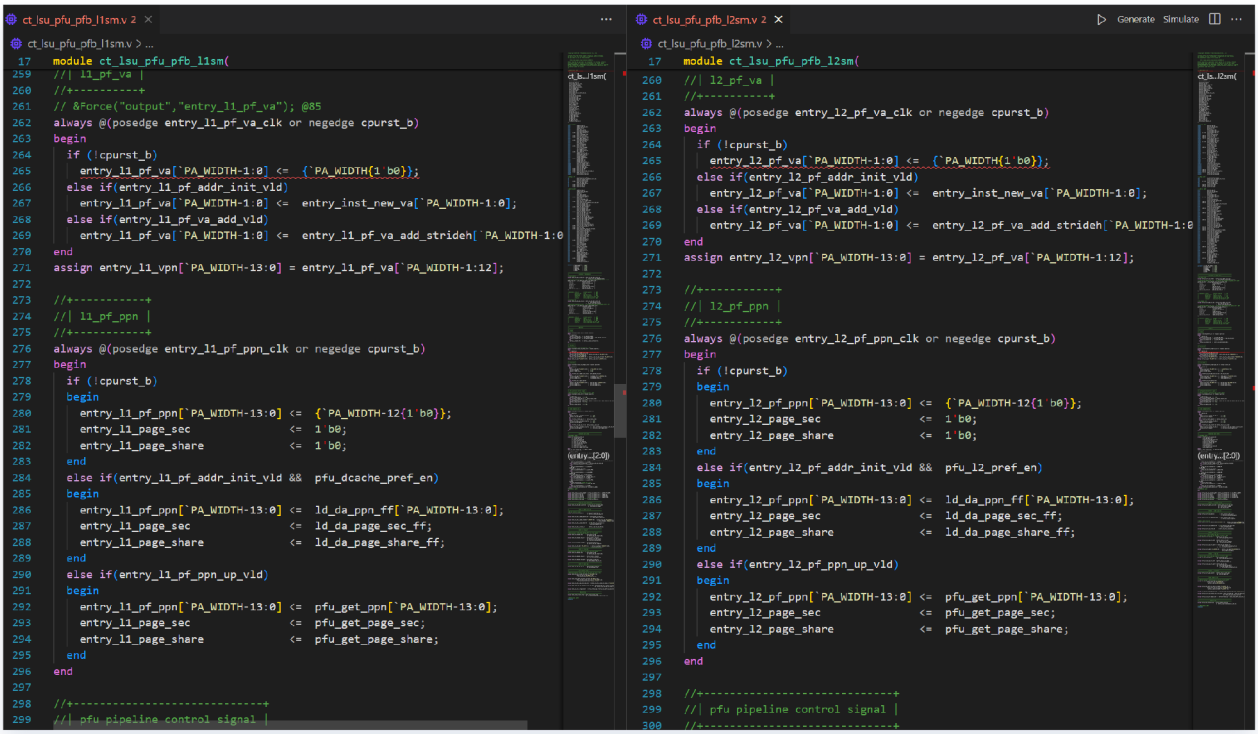
    1. 对于相同的模块例化，可以直接用向量填充 `VecInit(Seq.fill(...))` 来例化多个相同模块，然后模块间也可以直接用 `<>` 快速连接同名信号。
    2. 对于不同的模块例化，如果模块逻辑和结构比较相近，
        1. 对于相同功能的端口用一个 `Bundle` ，然后 `<>` 连接这部分端口。对于结构和逻辑不同端口再手动连接(也可以所有的端口都在一个 `Bundle` ，但是几个相同的端口用 `List` 或者 `Bundle` 再包一层。
        2. 几个模块一起打包为 `List` ，用 `for` 对相似逻辑一起连接， `if` 对不同端口不同操作。
    3. 端口用一个 `Bundle` 打包，然后 `Bundle` 使用 `.asUInt` 转化，转化之后再统一赋值。注意端口方向和位宽，同时每个想要连接的信号在 `Bundle` 里面信号顺序要一致

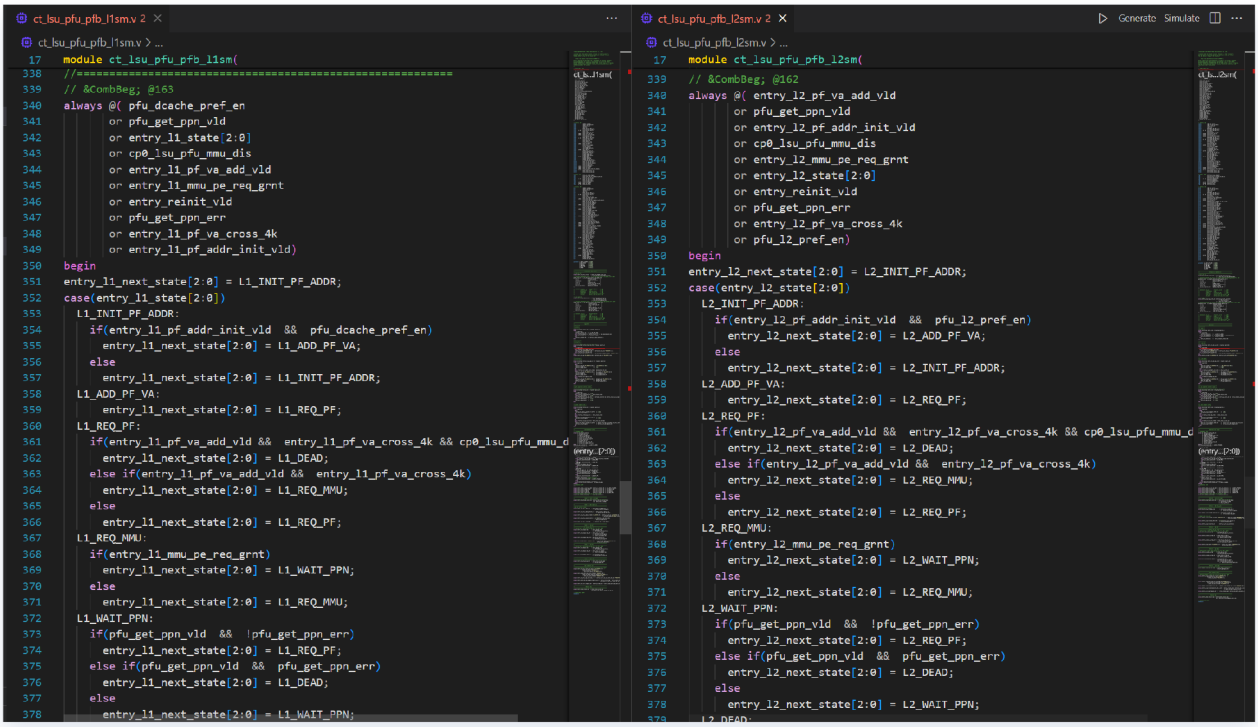
7. Scala和Chisel结合体验：

    控制结构 -> 集合 + 函数编程 -> 面向对象，虚对象虚方法等。此前没有学过面向对象编程，目前面向对象的一些术语，如特质（又叫成员变量、属性），虚方法（能被重定义的成员函数，即能被重定义的方法），重定义等不太懂，代码没能写出面向对象的精髓。

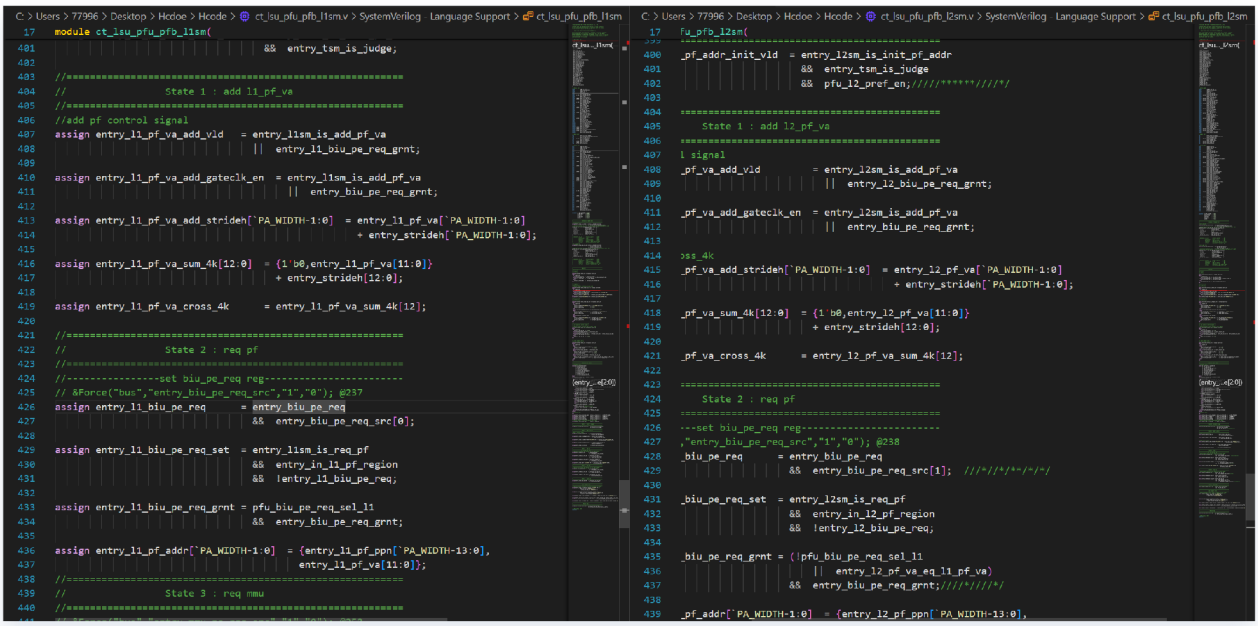
个人觉得最妙的Chisel化改进例子： `l1sm` 与 `l2sm` 的结合：

相似之处：





不同之处：（注意绿色标记的地方）



此外还有一些，不贴图了。

经过仔细分析发现，只有组合逻辑有区别，其次就是信号换了个名字罢了。因此有条件融合这两个模块。接下来只要分辨出不共有的组合逻辑信号，将 `l1sm` 和 `l2sm` 套在一个模块里面，通过指定参数 `chose` 即可。

跳转源码分析

再去看 `gpfb` 顶层连线，发现有一点点非常难发现的问题： `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 + inst_new_va == l1sm的inst_new_va

l2sm的entry_l2_pf_va == 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

TODO：优化想法

1. 可以将 `assign` 等赋值语句用 `List.map` 实现，如将相似的与门 `assign` 左侧装到一个 `List`，右侧装到一个 `List`，再对左侧 `List.map`。（节约了代码空间，但是易读性不好说）
2. 在对 `state` 参数进行定义时，使用了 `ChiselEnum` 的语法，但是和 `Reg` 一起使用会有很多奇怪报错，主要在于类型问题。Chisel中这方面经过调试可以跑，但类型匹配还没懂，建议可以用 `List` 等。

Chisel化综合体验：

一开始，Chisel还是挺费解难学的；后来发现，其实能用到的Chisel语法点/知识点也不多，大部分时候都是在搬砖。我觉得这个Verilog转为Chisel的工程，很难完全展现Chisel的强大之处，毕竟Chisel才是更高的语言。如果一开始就用Chisel来一个大的，设计一个大模块，我觉得更有意思。

学习体验大概就是：掌握Chisel基本硬件和数据类型等->掌握Verilog时序与组合逻辑和Chisel实现不同之处->开始搬砖，搬砖过程中费劲心思搞点Chisel和Scala特性（如控制结构，集合，函数，实现参数化等，还有一些 `Mux` 等帮助快速实现逻辑功能的Chisel硬件原语）。

例如：`x := Mux(sel.asBool, y, 0.U)`

例如：`x := MuxCase(default = 0.U(8.W), (0 to 7).map { i ⇒ sel(i).asBool -> (1.U(8.W) << i) })`

发射为Verilog并跑分的问题记录&学习收获

1. 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，个人觉得是这样的意思:

   以下是cookbook的例子：**对于有Bundle的情况**：

If the line is within a bundle declaration or is a module instantiation, it is rewritten to replace the right hand side with a call to `autoNameRecursively`, which names the signal/module.

```
class MyBundle extends Bundle {
  val foo = Input(UInt(3.W))
  // val foo = autoNameRecursively("foo")(Input(UInt(3.W)))
}
class Example1 extends Module {
  val io = IO(new MyBundle())
  // val io = autoNameRecursively("io")(IO(new MyBundle()))
}
```

```
// Generated by CIRCT firtool-1.75.0
module Example1(
  input       clock,
              reset,
  input [2:0] io_foo
);

endmodule
```

可见结果有 `io_` 。

而对于无Bundle的情况：

```
class Example2 extends Module {
  val in = IO(Input(UInt(2.W)))
  // val in = autoNameRecursively("in")(prefix("in")(IO(Input(UInt(

  val out1 = IO(Output(UInt(4.W)))
  // val out1 = autoNameRecursively("out1")(prefix("out1")(IO(Outpu
  val out2 = IO(Output(UInt(4.W)))
  // val out2 = autoNameRecursively("out2")(prefix("out2")(IO(Outpu
  val out3 = IO(Output(UInt(4.W)))
  // val out3 = autoNameRecursively("out3")(prefix("out3")(IO(Outpu
```

```
// Generated by CIRCT firtool-1.75.0
module Example2(
  input        clock,
               reset,
  input  [1:0] in,
  output [3:0] out1,
               out2,
               out3
);
```

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

2. **解决方法：**

    1. 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。
    2. 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。
    3. 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。
    4. 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。
    5. 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。

2. 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。

1. 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"。
2. 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。
3. 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。

C910如何仿真并跑分

详细步骤见荣誉课文档。

我从网上搜索，这个教程可以在自己电脑上运行，不过第一步要破解VCS和Verdi，有教程，保证能破解的兄弟。Eetop上面好像也有。

主要流程：

1. 克隆/下载C910官方开源代码
2. 下载C910相关gcc（xuantie900）并解压
3. 修改 `smart_run` 里面的 `setup/example...` 里面对 `toolextension` 的路径
4. 网上找个教程，运行提供的各种预设文件setup
5. 替换自己的代码到 `gen_rtl` 相应位置，同时更改 `filelist`
6. 运行 `make runcase CASE=hello_world`， `make runcase CASE=coremark`

细节见文档或者网上教程。