**输入输出文件**

1. 输入的是excel文件，格式同jsy现在用的那一份（cob的坐标体系要改一下，和jsy他们说，还有excel当中没有指出xinzhai所在的端口是上下左右哪条边）
2. 文本文件是输出的controlbit

**说明**

1. 读取Excel文件用xlnt库（虽然LibXL也可以，但是这个需要注册付费，否则读取的单元格数量有限制）。
2. xlnt库网址：https://github.com/tfussell/xlnt

**excel文件信息**

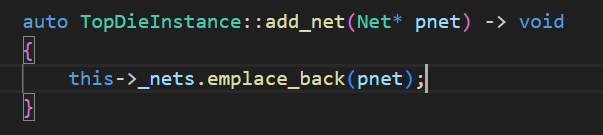
1. RegName格式

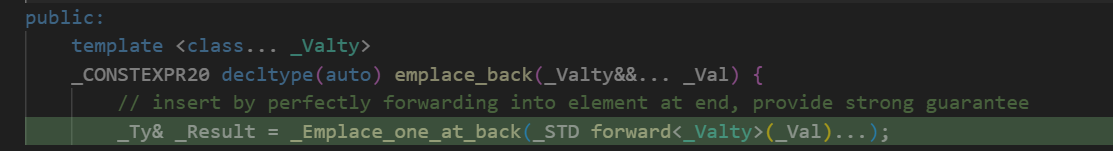
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 前缀 | 中缀 | | 后缀 | 对应硬件 | 获取方式 | 寄存器类 |
| cob\_ x，y坐标 | 上下左右\_sel | | 数字 | 表示cob四条边上的端口信号方向 | 信号方向怎么获取 | CobPortReg |
| sw\_方向 | | 数字 | 从cob\_unit的私有成员变量中获取。需要在顶层cob留一个接口函数，获取6类开关各自的状态，然后在cobunit中填补该函数 | 在cobunit的私有成员变量中 | CobSwReg |
| tob\_x, y坐标 | / | | pulldn |  | 保持初始值 | / |
| tob2bump/bump2tob | bank0/1\_en | 数字 | Tob与bump之间的信号方向 | 信号方向，应该可以由起点/终点位置判断 | TobBumpReg |
| tob2track/track2tob | / | 数字 | Tob和track之间的信号方向 | 也可以直接由起点和终点判断 | TobTrackReg |
| drv/dly | / | 数字 |  | 保持初始值不动 | / |
| bank\_sel | / | 数字 | Bank选择，直接从tob的mux当中读，读出来是十进制数，需要根据对应关系转换为0/1 | Tob | TobMuxReg |
| hctrl/vctrl | / | 数字 | 选择器，直接从tob的mux当中读，读出来是一个十进制数，需要转换成3位二进制数 | Tob | 同上 |
| tob | 编号 | / | det\_out |  | 保持初始值 | / |
| detect | / | / | en |  | 保持初始值 | / |
| xinzhai\_C4\_noi | / | 上下左右 | pad\_ctrl\_数字 |  | 保持初始值 | / |
| Sip |  | 保持初始值 | / |

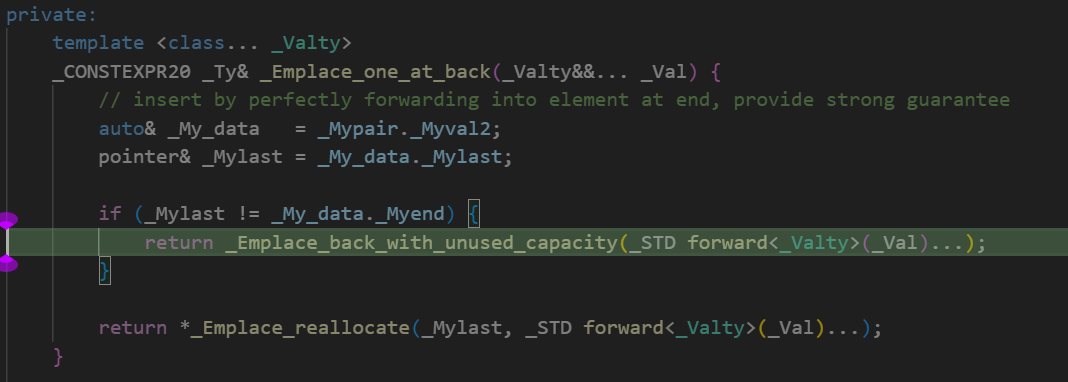
**Debug:**

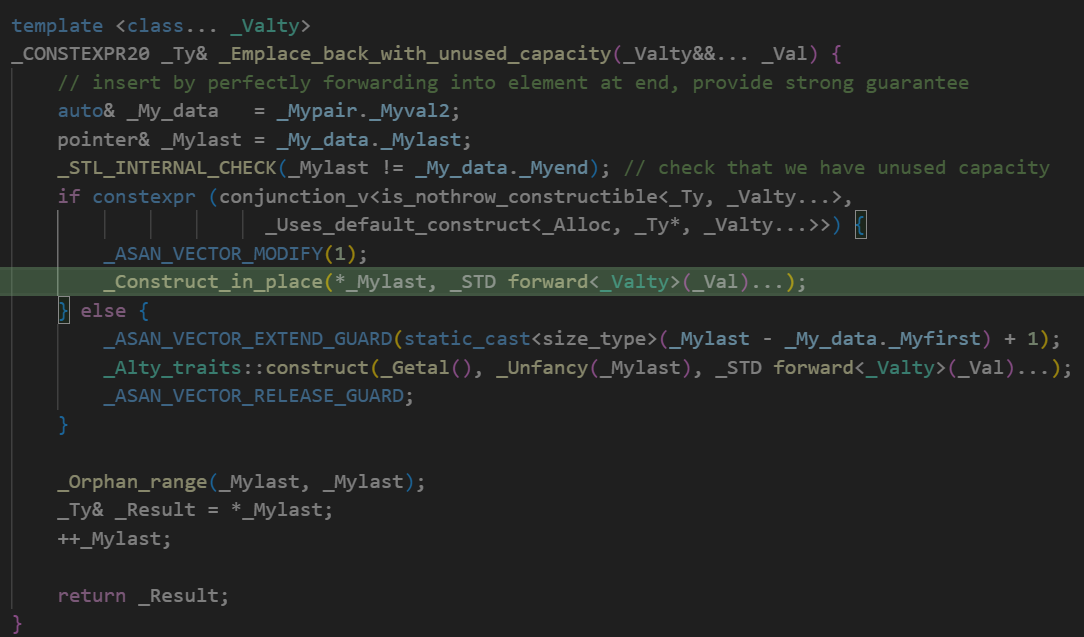
1. Parser部分：

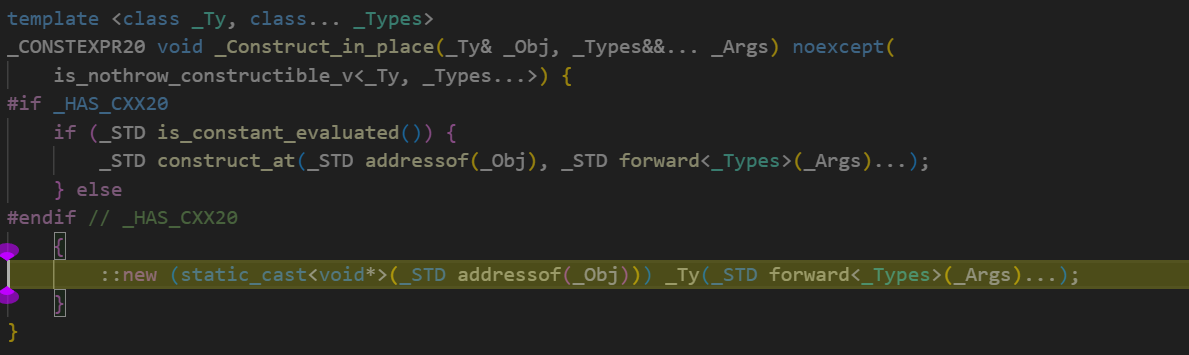
这里的 Net 是一个抽象类



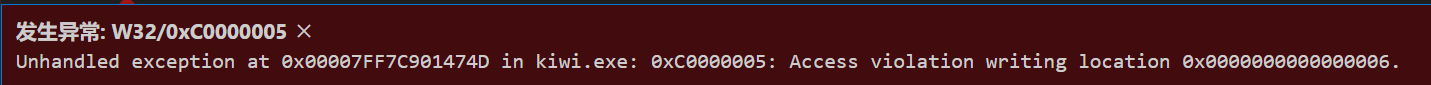








最后报错：



说明有非法内存访问操作，有可能这里的 \_Obj 有问题，其指向的内存有可能不可写或者是无效地址。检查：对抽象类的实例化，不当的解引用，pnet所指向内存配分是否正确，pnet所指向对象没有被销毁。

**一些想法**

1. 关于同步总线布线，如果写一个指定长度布线的函数完成单根线的布线，并允许资源重复使用，然后用协商布线的思路完成多跟线之间的调整，另外原来的算法是拆掉重布，能不能参考最小代价最大流算法，允许一根线被反向布线。可以先写一下xl的算法看看具体过程。然后 Pathfinder 的内层能不能换成 A\* 算法。
2. Tob和 Tobinst 感觉可以不用（不确定），至少 tob 那个可以去掉，里面存储的信息首先 name 肯定用不到，bump 的信息也存在 net 里面了
3. Nets 当中的前三个含 s 的net可以不用，因为会有bump又连到bump又连到track，并且读文件的时候全都是以二端口的线读入，判断两根线是否共用一个bump或者 track 又困难。可能在布线的时候需要先判断一下bump有没有连过，连过的话直接从路径开始就可以
4. 对于 cob 和 tob 的寄存器，目前只需要一个返回全部寄存器内容的函数，不需要读取某一位
5. 检查各层次的类的析构函数，很多类不是 PLD ，需要手动释放在堆上 new 出来的对象

**总线**

1. Track 在沿着 cob 传播的时候，总是在自己所在的 cobunit 之内传播，从 track 的索引上看，起始点-路径-可到达点上的所有 track 索引对8取模结果相同
2. 按照上面的思路，当 track 确定之后，针对这一条 track 就可以简化整个硬件平面的结构，即在这条 track的视角里，所有的 cob 都只是一个 cobunit
3. 长度最小值是起点、终点所围矩形的半周长，若得不到最小值，则长度超出的量必定是偶数

**笔记**

1. Namespace 可以用来避免全局命名冲突的问题。
2. C++当中的各种标准异常继承了 std::exception ，并且该基类提供了一个虚成员函数 what() 用于返回异常的信息，可用于调试
3. Std::cerr 不经过缓冲区直接输出，比较及时？
4. 各种更细分的数据类型，让数据使用更加准确
5. xlnt 中的行、列索引从1开始.
6. xlnt 获取 cell 可以用 worksheet.cell(xlnt::cell\_reference(col, row)) ，注意是先列再行
7. 任何可能作为基类使用并且可能通过基类指针删除派生类对象的类，都应该将其析构函数声明为 virtual，以确保正确的析构行为。为了避免在析构的时候抛出异常导致未定义行为，可以给析构函数最后加上 noexcept 修饰。
8. 文件中所有用到的类需要引入头文件，如果报错：不允许使用不完整的类，有可能是因为头文件没加导致的，或者类当中有成员函数没有给出定义
9. 如果报错出现“无法解析的外部符号”，有可能是编译的时候没有把对应的cpp文件链接进去（或者cc文件，总之就是实现文件），也有可能是在代码文件中没有把包含定义的文件 include 进来
10. **静态链接库（Static Library）和动态链接库（Dynamic Library）**

**静态：**

**文件类型**：静态库通常以 .lib（Windows）或 .a（Linux、macOS）为后缀。

**链接时间**：在编译时（也叫静态链接），静态库的代码被直接嵌入到可执行文件中。

**工作原理：**在编译和链接过程中，编译器会将静态库中的所有函数和代码与目标文件（如 .obj 或 .o 文件）链接，并把这些代码复制到最终生成的可执行文件中。因为静态库的代码已经被嵌入到可执行文件中，运行时不需要依赖任何外部库文件。

**动态：**

**文件类型**：动态库通常以 .dll（Windows）、.so（Linux）或 .dylib（macOS）为后缀。

**链接时间**：动态链接发生在运行时，而不是编译时。程序运行时动态加载动态库的代码。

**工作原理：**当编译器生成可执行文件时，它不会将动态库的代码直接嵌入到可执行文件中，而是记录一个引用，指向该动态库。在程序运行时，操作系统会动态加载动态库，并在程序中使用该库的函数和代码。动态库的代码是共享的，多个程序可以在内存中同时使用同一份动态库，减少内存和磁盘占用。

在使用 vcpkg 安装 xlnt 库后，你需要在 xmake.lua 中正确配置 vcpkg 管理的库。vcpkg 会将所有安装的库（包括 xlnt）放置在一个统一的目录中，并且会管理库的头文件和链接信息。

**集成 vcpkg 到 XMake**

xmake 具有对 vcpkg 的内置支持，使用 vcpkg 安装的库可以通过 add\_requires() 来自动引入到项目中，而不需要手动设置库的路径或头文件路径。

在你的 xmake.lua 文件中，直接添加对 xlnt 的依赖即可：

lua

复制代码

add\_requires("vcpkg::xlnt") -- 使用vcpkg版本的xlnt库

target("your\_target\_name")

set\_kind("binary") -- 如果是生成可执行文件

add\_files("src/\*.cpp") -- 你的源码路径

-- 链接xlnt库

add\_packages("vcpkg::xlnt")

**如何工作**

* **add\_requires("vcpkg::xlnt")**：这是 xmake 引入 vcpkg 包的方式。它告诉 xmake 你依赖 vcpkg 安装的 xlnt 库。
* **add\_packages("vcpkg::xlnt")**：在你的目标中使用这个包，xmake 会自动处理所有的头文件、库文件路径的设置，并将它们链接到你的项目中。

**xmake 全局配置 vcpkg**

在某些情况下，xmake 可能需要知道 vcpkg 的安装路径。如果你没有配置过全局的 vcpkg 路径，你可以用以下命令全局设置 vcpkg 的路径：

bash

复制代码

xmake g --vcpkg =/path/to/vcpkg

例如，假设你在当前项目目录下安装了 vcpkg，那么你需要指定：

bash

复制代码

xmake g --vcpkg=./vcpkg

这将告诉 xmake 去哪里寻找 vcpkg，并且可以在多个项目中复用 vcpkg。

**调试与 Release 区分**

xmake 会自动处理 Debug 和 Release 版本的库。如果你使用 xmake f -m debug 或 xmake f -m release，xmake 会根据你的配置自动选择合适的 vcpkg 库。

另外 xmake 有内置的包管理工具，在本机路径 C:\Users\17914\AppData\Local\.xmake\repositories 下有一个 xmake-repo 文件夹，里面有相关的包管理文件，文件可在 <https://github.com/xmake-io/xmake-repo?tab=readme-ov-file> 网址下载。准备好文件之后，在 xmake.lua 当中加入

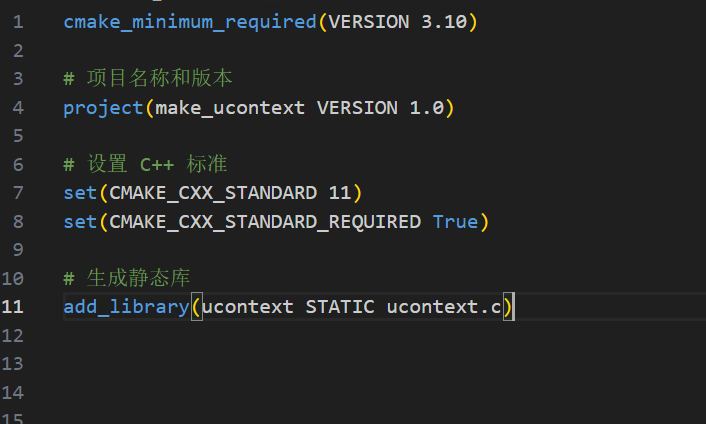
Add\_requires(“package\_name”)

Add\_packages(“package\_name”)

然后直接在 xmake.lua 所在目录下运行 xmake build 即可。

1. 生成静态链接库

先写 CMakeLists.txt ，注意需要加上 add\_library 选项



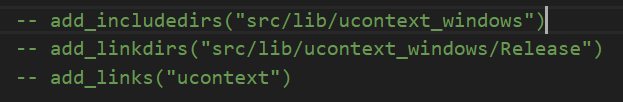
然后运行 cmake 命令：

Linux 上用：cmake /path/to/CMakeList.txt -DBUILD\_SHARED\_LIBS=OFF

Windows 上需要加上：-G "Visual Studio 17 2022"

再用 cmake --build . --config Release 生成对应的库文件，Linux 会直接生成 .a 文件，Windows 会生成 .lib 文件

如用 xmake 管理项目，需要加上这些配置：



在调用库文件内容的时候需要 include 对应头文件，并加入头文件的父目录在 includedirs 中，然后在 add\_linkdirs 中加入 .lib/.a 文件所在父目录。

1. C++处理异常

异常：程序中的各种错误。

**传统错误返回需要一层层返回错误，而异常处理可以直接到达能够处理异常的地方。意思是传统错误需要按照函数的调用关系一级一级的向上返回，但是异常处理的时候可以直接返回到main函数。**

try {

throw RouteError{nullptr};

} catch (const RouteError& e) {

std::cerr << "Caught RouteError: " << e.what() << std::endl;

} catch (const std::exception& e) {

std::cerr << "Caught std::exception: " << e.what() << std::endl;

} catch (...) {

std::cerr << "Caught unknown exception" << std::endl;

}

可以用上述方式处理异常。当try捕获到异常的时候，会进入 catch 模块处理对应的异常，可以有多个 catch ，并且会根据捕获的异常自动匹配 catch 后面括号里面的异常对象进入相应的模块。如果不清楚异常类型，可以用 … 代替，这样确保覆盖所有异常。如果没有覆盖到，也会有缺省的功能，会自动调用 abort 函数终止运行。

1. Github 开发流程

单人：先创建仓库，得到远程仓库的地址

然后使用 git 指令

Git init

Git add .

Git commit -m “message”

Git remote add origin /address/to/repository

git push --set-upstream origin a\_branch\_name

git push