Basic Idea:

- Use Centos 7.4
- Install Quartus Prime Pro 18.1
- Install gcc 5.4 as per instructions online
- Procure Gtest
- Create CMakeLists.txt file properly
- Export CXX variable as i++ compiler path
- "CMake3 .." or "CMake .." from build directory
- "make"
- run produced executable

Instructions

1.) Set up System

I will assume the reader knows how to install Centos 7.4 (or some other redhat 7.x distro) and Quartus Prime Pro 18.1 without any assistance from me.

2.) Install gcc 5.4.0 so it's used instead of gcc 4.8

This is also fairly straightforward, simply follow the directions here: https://www.intel.com/content/altera-www/global/en_us/index/documentation/ewa1462479481465.html#ulj1521476282903

3.) Procure GTest

On Centos 7.4, I used an old version of GTest that is easily installed with:

```
sudo yum install gtest-devel.x86_64
```

You may choose to download GTest from GitHub. For some reason, I had trouble linking against the newer versions of GTest after building, but you can try it. I found the older version I installed with 'yum' worked out of the box, so I didn't put much effort into this stage since it worked.

Here's a link to various releases: https://github.com/google/googletest/tags

4.) Setting up Your CMakeLists.txt

This step is relatively straightforward if you are familiar with CMake. Otherwise, you may wish to refer to my CMakeLists.txt files. I commented them to make it easier to understand. The "simpleGTestI++" project contains a simpler version of the file that I used when I was teaching myself the setup. I think it's a good idea to make sure this works on your system first because it meant to be less complex (as the name implies). This project should compile and run without changes if you have done the above steps correctly.

One exception:

> If you did not install gtest with yum, but wish to have it as a dependency inside your project, you will need to edit your file so that CMake can find gtest and link against it properly.

5.) Compiling and Running the Simpler Example

In order to compile, you do have to manually specify the compiler you wish to use. I did not want to hard code this into the CMakeLists.txt file because I believe this particular move would fall on the side of bad practice. However, once you specify the path, the CMakeCache.txt file will remember it forever unless you delete it.

When you wish to compile, just run:

```
export CXX="PATH/TO/QUARTUS/INSTALLATION/intelFPGA_pro/18.1/hls/bin/i++"
```

Remember this path will only persist in your terminal as long as it remains open. However, it WILL persist in the CMakeCache.txt that is generated when you run cmake or cmake3 from the build directory. With that out of the way, the next steps should work (in an ideal world anyway).

You should have your highest level CMakeLists.txt file at the highest level of your project directory, which I will assume. Run the following and you should see the test output:

```
mkdir build
cd build
cmake ..  (ALTERNATIVELY, you may need to use "cmake3 ..")
make
./myTest
```

You now should see the first test pass and the second test fail. This is compiled with the x86-64 architecture and has two simple tests using types defined in HLS headers and properly ignores the 'component' keyword before the multer function.

6.) Moving on to the "Full" Example

You now are hopefully going to get past where I am. This is where I get stuck a bit. In the "full" example, I make a small addition to the CMakeLists.txt file that I didn't expect to cause much trouble. I add the compile flags to specify the machine architecture as an FPGA. However, this is all that is needed to cause a peculiar error at link time. Specifically:

```
LMemSplit: Unable to resolve split destination type on:
_ZTVN10__cxxabiv120__si_class_type_infoE
Compiler Error:   Cannot resolve destination address space through pointer indirection.
HLS Main Optimizer FAILED.
make[2]: *** [x86] Error 1
make[1]: *** [CMakeFiles/x86.dir/all] Error 2
make: *** [all] Error 2
```

Okay, so here's all the information I've gathered that I think is useful. After scouring stack overflow, I've found that errors involving the

"_ZTVN10__cxxabiv120__si_class_type_infoE"

tends to involve not linking properly to "libstdc++" I tried to explicitly link to the appropriate shared library in the gcc 5.4.0 installation, but I did not have much luck with this.

I am doing some extrapolation because I do not have a clear understanding of what is happening, so please bear with me if you are kind enough to still be reading.

This is my output from “cmake3 ..” It confuses me that the CXX identification is Clang and not g++. I worry this may actually be the culprit, but I’m not entirely sure how to test that at the moment.

```
cmake3 ..
-- The C compiler identification is GNU 4.8.5
-- The CXX compiler identification is Clang 6.0.1 ←==========================
-- Check for working C compiler: /bin/cc
-- Check for working C compiler: /bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /…/intelFPGA_pro/18.1/hls/bin/i++
-- Check for working CXX compiler: /…/intelFPGA_pro/18.1/hls/bin/i++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features – failed ←============================
-- Found GTest: /usr/lib64/libgtest.so
-- Configuring done
-- Generating done
```

“_ZTVN10__cxxabiv120__si_class_type_infoE” is found on this page: http://refspecs.linuxfoundation.org/LSB_3.1.0/LSB-CXX-generic/LSB-CXX-generic/libstdcxx-ddefs.html

This gave me some useful information about what the above gibberish is actually referring to.