**Chisel Tutorial**

# Introduction

[CHISEL](https://chisel.eecs.berkeley.edu/) V2 基于**[Scala](http://112.124.120.180/mediawiki/index.php/Scala" \o "Scala)** Embedded Language, 属于internal DSL。直接写scala，调用chisel内置的库构造出硬件结构，然后产生出C和verilog代码，同时也提供testbench用于快速验证。

在[RISC-V Rocket Core](https://github.com/ucb-bar/rocket-chip)项目中实际上使用了chisel3, 他实际上是将chisel分解为两部分，构造出了一个IR层，后端是[firrtl](https://github.com/ucb-bar/firrtl), 是一个简化版的硬件描述语言，属于external DSL, 用[ANTLR](http://www.antlr.org/)编写。他们希望在此之上发展出各种灵活的前端，也许不拘泥于scala语言。

**注意**！chisel3/firrtl任然在alpha版。

# Chisel快速上手

* 确认系统中安装了JDK，[Scala](http://112.124.120.180/mediawiki/index.php/Scala" \o "Scala), [SBT](http://112.124.120.180/mediawiki/index.php/Scala" \l "SBT" \o "Scala)
* 如果需要用最新版本（不是必须的，SBT可以自动下载最新版），下载编译chisel （更新是比较频繁的）

git clone <https://github.com/ucb-bar/chisel.git>

cd chisel

make clean test publish-local

* 写一个build.sbt文件

scalaVersion := "2.11.7"

libraryDependencies += "edu.berkeley.cs" %% "chisel" % "latest.release"

// if you have build chisels and do publish-local, use the following line

//libraryDependencies += "edu.berkeley.cs" %% "chisel" % "2.3-SNAPSHOT"

* 然后就可以开心的写scala文件了，对于严格的sbt规范来说，你需要把scala放到src/main/scala/目录下.
* 每个scala文件通常是一个module加一个unit test。

import Chisel.\_

//A n-bit adder with carry in and carry out

class Adder(val n: Int) extends Module {

val io = new Bundle {

val A = UInt(INPUT, n)

......

}

class AdderTests(c: Adder) extends Tester(c) {

for (t <- 0 until 4) {

val rnd0 = rnd.nextInt(c.n)

......

}

* 这样其实还不行，运行是会说没有main。需要添加一个main

object AdderMain {

def main(args: Array[String]): Unit = {

val tutArgs = args.slice(1, args.length)

val res = chiselMainTest(tutArgs, () => Module(new Adder())) {

c => new AdderTests(c)

}

}

}

* 最后调用sbt就可以了

sbt "run Adder --backend v --targetDir out"

这是一个大文件包含顶层模块和所有的子模块的verilog语言, 不知道如何让它对每一个module产生一个文件。

* 当你写了第二个文件，问题来了，sbt说发现了两个main，这就是为什么chisel-tutorial中引入了一个公用的main, 通过命令行参数调用对应的ChiselMainTest. 如果你用java执行指定的main也是可以的，但是sbt太聪明了，也许我们可以自己写个script来达到这个简单的目的。执行链条是

chiselMainTest.apply -> chiselMain.apply -> Driver.apply

* firrtl跟chisel不同的是它是parse文件的。所以工具可以独立出来

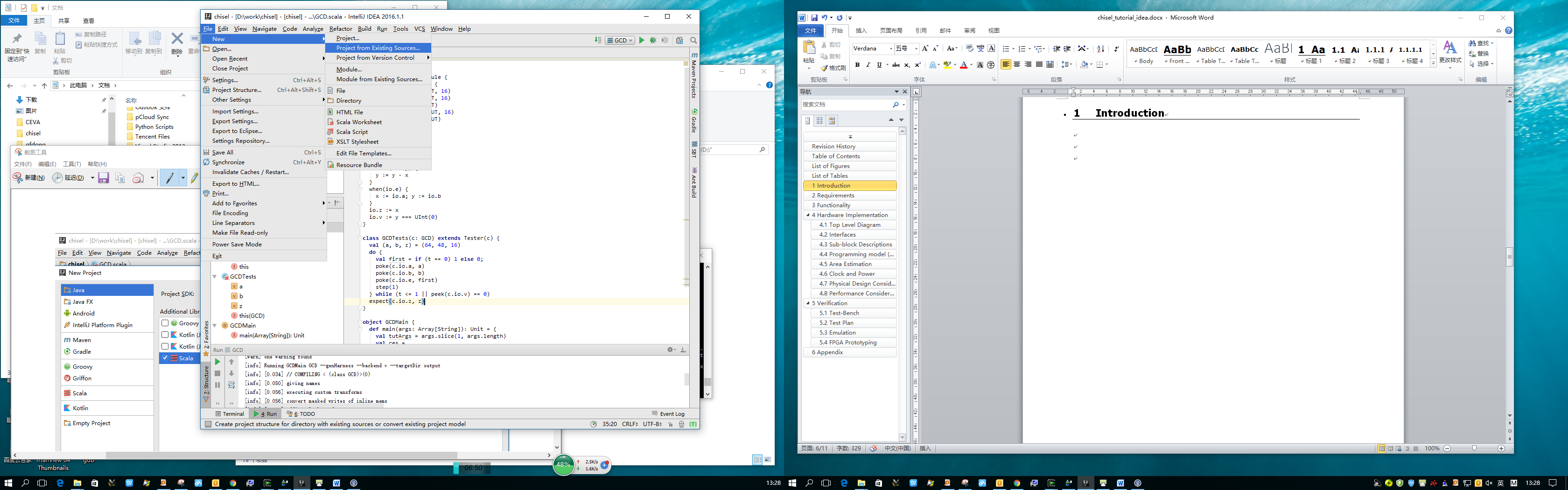
alias firrtl='java -cp <path\_to>/firrtl.jar firrtl.Driver'

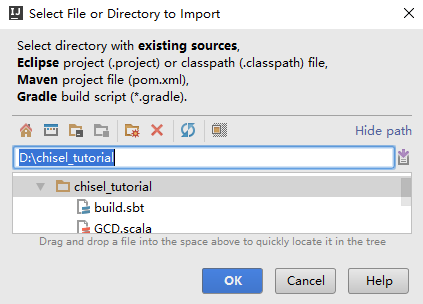
firrtl -i <input>.fir -o <output>.v -X verilog

# 使用IDEA开发

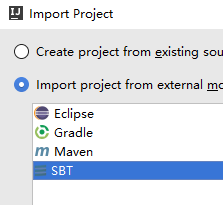
在工作目录下放置scala source文件和build.sbt

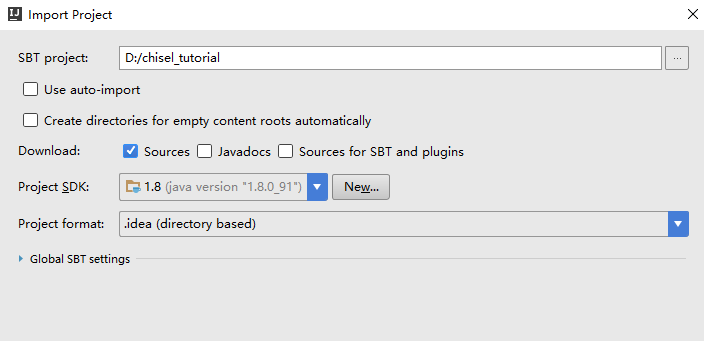
在此之上创建新项目

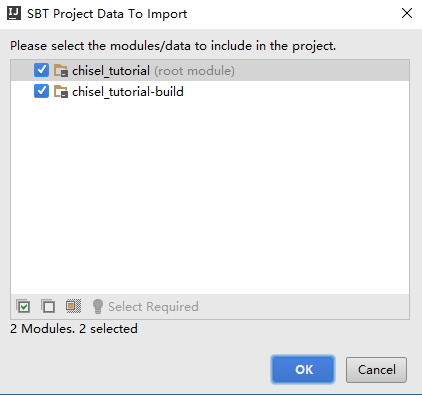




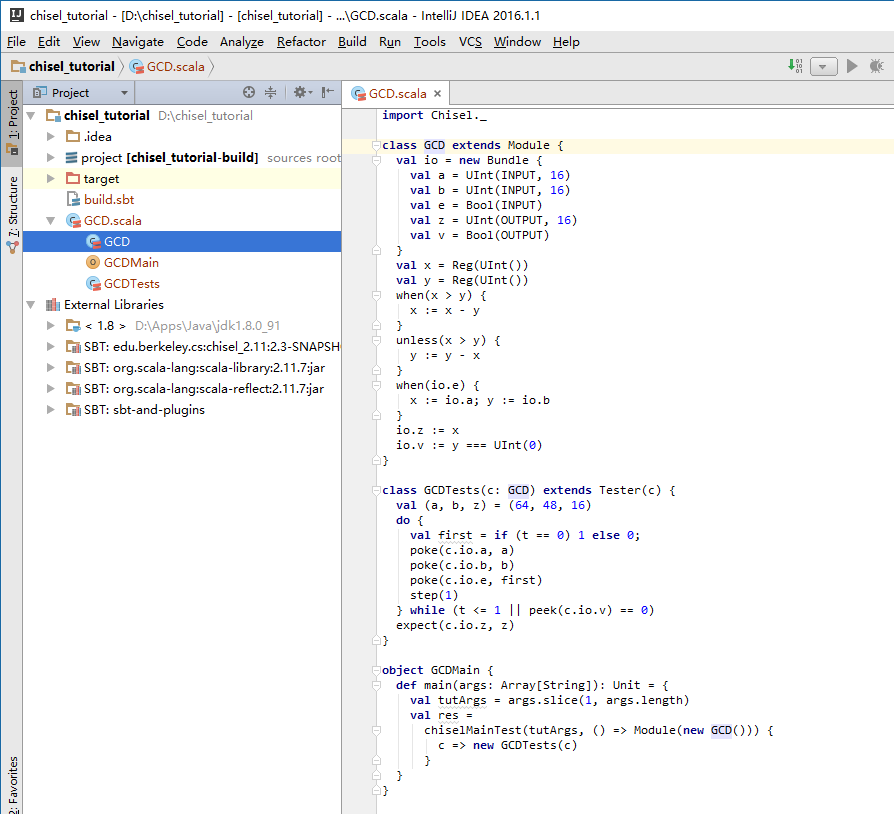
选择导入SBT项目



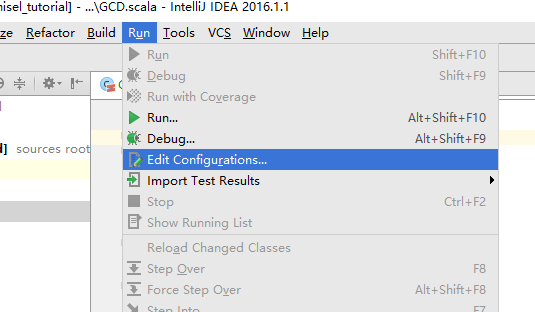




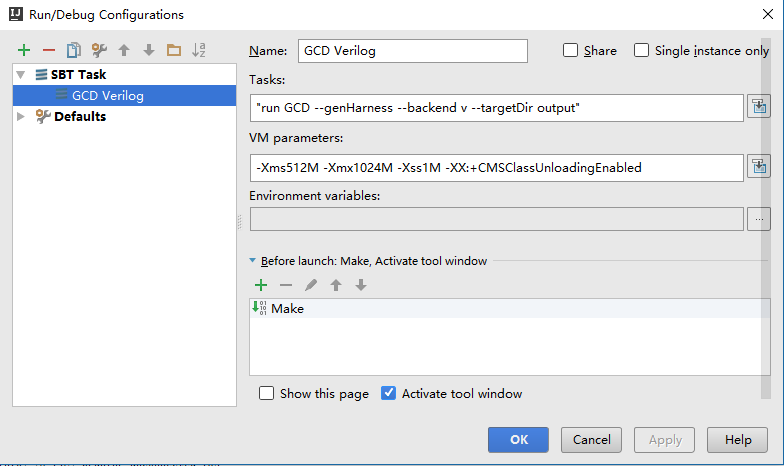
可以开始编辑source文件



build产生Verilog文件，需要构建新的sbt task



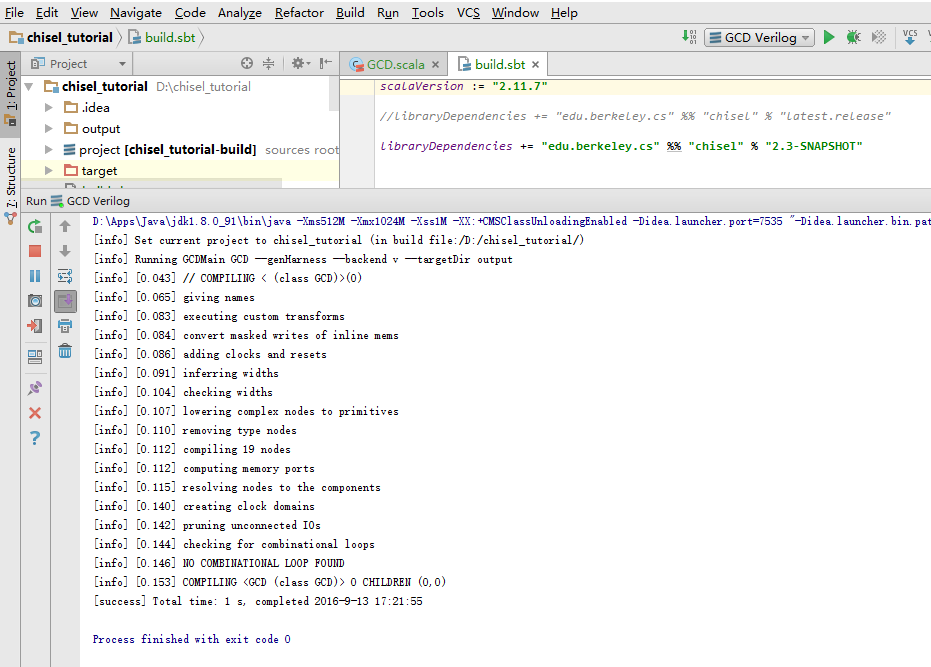
+新建一个SBT task，如下设置



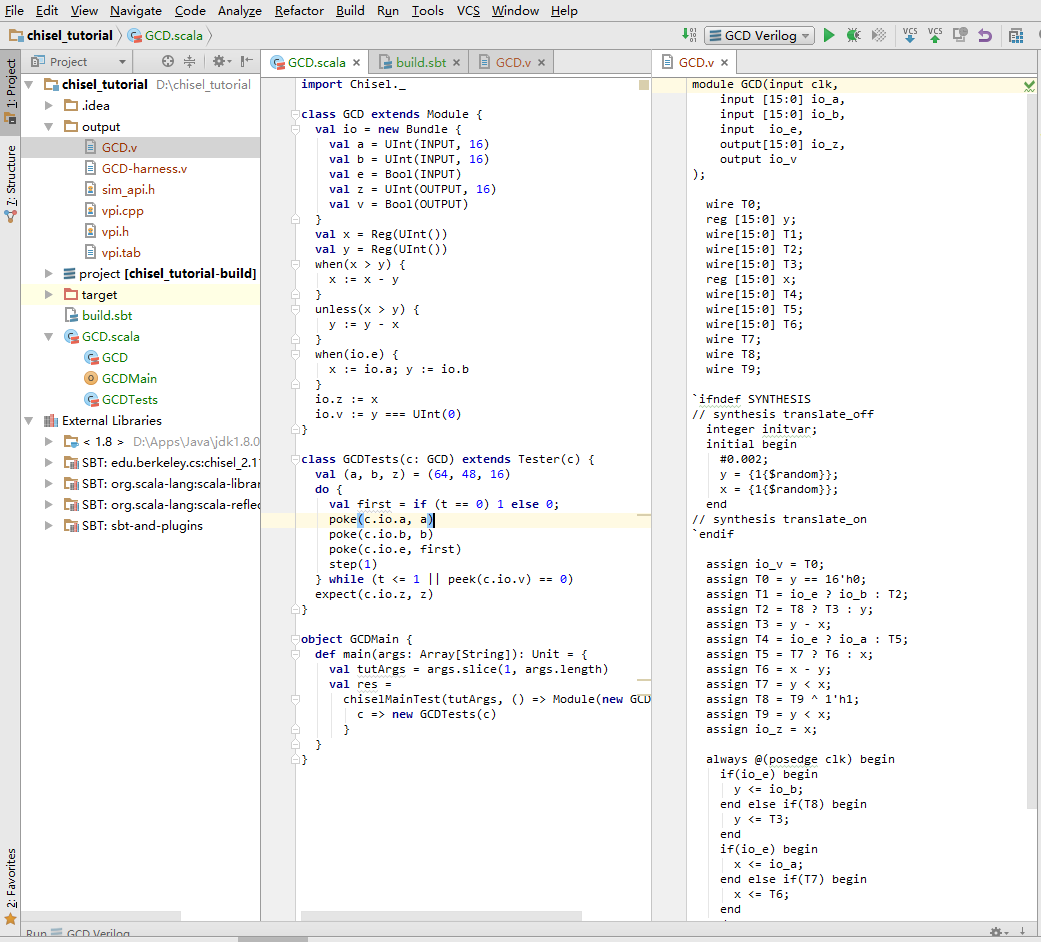
可以看到右上方出现刚才的设置。



点运行按钮

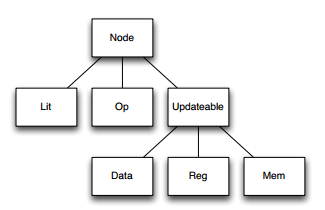


在output有verilog文件出现了



# Chisel quick ref

# Chisel 类



RF

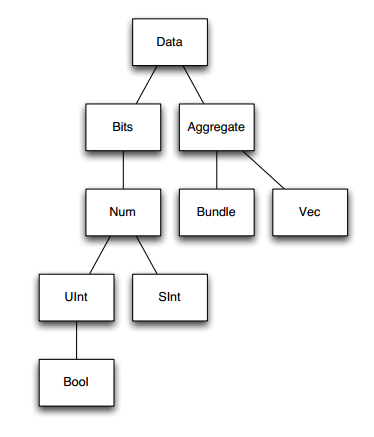
val rf = Mem(32, UInt(width = 64))  
when (wen) { rf(waddr) := wdata }  
val dout1 = rf(waddr1)  
val dout2 = rf(waddr2)

2P

val ram1r1w =  
Mem(1024, UInt(width = 32), seqRead = true)  
val dout = Reg(UInt())  
when (wen) { ram1r1w(waddr) := wdata }  
when (ren) { dout := ram1r1w(raddr) }

1P

val ram1p =  
Mem(1024, UInt(width = 32), seqRead = true)  
val dout = Reg(UInt())  
when (wen) { ram1p(waddr) := wdata }  
.elsewhen (ren) { dout := ram1p(raddr) }



Reg

RegNext

RegInit

**when** (<cond>) {

**updateWhen**(<cond>, <value>)

} **elsewhen** (<cond>) {

} **otherwise** {

}

**unless** (<cond>) {

} # same as when( ! <cond>)

**switch**(<var>) {

**is** (value) {

}

}