实验一：Chisel / Scala 简介与 Verilog 介绍

1. 实验目的

1.1. 了解学习 Verilog 语言，建立硬件描述语言的基本概念；

1.2. 对 Chisel 语言有初步的认识，了解 Chisel 语言的基本概念和特点；

1.3. 学习掌握基本常用的 Scala 语法。

2. 实验内容

2.1. 学习 Verilog 相关背景知识，理解 HDL 硬件开发与软件开发的区别；

2.2. 学习 Chisel 的相关背景知识；

2.3. 使用 Scala CLI 运行基础的 Scala 程序，并能够运行 Chisel 例程。

3. 实验步骤

3.1. 芯片前端设计开发背景知识

在一个完整的芯片从确定需求到制作完成，其中包含了许多的流程， 可以大致分为设计和制造两大部分，其中设计通常是由开发芯片的公司完成，确定芯片的各种性能参数和规格，设计完成后，将芯片的详细设计图纸送往芯片代工厂，例如中芯国际、台积电等，由他们完成芯片制造这部分的工作。 芯片设计可以理解成一栋大楼的设计师，设计师绘制大楼的建筑图纸，确保这栋大楼结构的稳固和功能的完备。而芯片制造则是负责真正一砖一瓦的将这栋大楼建造完成的工人。

而芯片的设计部分又可以细分为前端设计（也称逻辑设计）和后端设计（也称物理设计）。前端设计主要关注的是芯片如何实现需要的功能， 怎样达到性能要求，而后端设计主要关注的是如何保证前端设计出来的电路在现实中能够制造出来并正常运行。 在本实验课中，主要学习了解的是芯片前端设计的流程，而后端设计流程会在另一套实验课中介绍

芯片前端设计的主要流程如图 1 所示，接下来详细介绍各个流程的含义。

图 1 芯片前端设计流程

架构设计：和软件设计一样，在决定设计制作一款芯片之前，需要进行需求分析，确定芯片的功能、性能、可行性，然后依此设计相关的架构，按功能划分模块，确定各部件的规格，形成一份设计文档，后续开发以此文档作为参考。

框架搭建：设计文档完成后，需要尽快开始搭建整体项目的框架，搭建好开发环境，配置好项目构建工具，为后续模块的开发及验证做好铺垫。

模块实现：按照设计文档，将整体功能划分为多个模块，再将多个模块划分为更细的各个子模块，然后逐一实现，在模块实现的过程中，使用 HDL (hardware description language，硬件描述语言) 来描述电路的实现，通过 HDL，能够精确地描述电路设计中每一个寄存器，每一个逻辑门，每一根线，而通过 HDL 描述的电路代码称为 RTL (Register Transfer Level，寄存器传输级) 代码，这也是前端设计最终交付给后端做物理设计的目标代码。 HDL 有很多种，例如 Verilog、System Verilog、VHDL 等，在本实验中，使用 Chisel 作为主要开发语言，在接下来的一系列实验中，会逐步的介绍 Chisel 的相关基础知识。

单元测试：模块实现完成后，需要对其进行测试，因此需要针对模块编写对应的单元测试，单元测试主要测试模块的功能正确性，应该随模块一起开发。之所以需要单元测试，是因为单元测试的目标模块相比于整个系统更加简单，在模块实现错误时可以快速定位错误，而且 单元测试的速度极快，可以通过快速地迭代，即发现问题-修改代码-再次测试不断循环，直到模块通过测试。确保模块功能的正确性，提升开发效率。单元测试的设计也很关键，决定了对模块的测试是否充分，单元测试越全面，后续这个模块出错的概率越小。

整体仿真验证：在所有模块的单元测试都通过后，需要将所有的模块整合在一起，整个系统进行仿真测试， 之所以叫做仿真 (Simulation) 测试，是因为这个阶段不可能将设计代码用真实的物理器件做出来测试，需要使用专门的仿真软件，来模拟设计的电路的行为，来验证功能的正确性，因此叫做仿真测试。 这一步目的在于验证整体系统的功能，可以发现一些模块间交互的错误，或者之前单元测试没有覆盖到的情况。并且此时可以通过运行一些标准测试程序，来验证整体设计的性能。

FPGA 验证：在仿真验证通过之后，可以通过将设计在 FPGA 上实现来近似的验证，FPGA (Field Programmable Gate Array，可编程逻辑门阵列)是一种半定制电路，可以简单的理解成一

块可以随意改写内部电路的芯片。在 FPGA 上验证可以发现一些仿真时容易忽略的细节，比如通信协议，复位信号的实现。由于 FPGA 上调试相比于单元测试和仿真时更加复杂，因此尽量在整体仿真验证时确保所有功能正常。

时序分析与后端设计：在 FPGA 验证通过后，就可以进行时序分析，以及后端的物理设计。时序分析主要是检查设计的电路是否能够满足设计文档中一些频率等指标的要求，后端设计主要是将 RTL 代码转换为可以用于代工厂制造芯片的文件。

以上就是前端设计的开发流程，本实验课程将通过多个实际的模块例子一步步介绍 Chisel 语言的使用以及介绍部分硬件设计相关概念与调试方法。

3.2. Verilog 介绍

Verilog 是一种专门用于设计和模拟数字电路的硬件描述语言（Hardware Description Language，HDL）。它诞生于 1983 年，由 Phil Moorby 开发，并迅速成为电子工程师设计集成电路（IC）和现场可编程门阵列（FPGA）的重要工具。Verilog 的设计哲学是简洁性和易学性，它借鉴了 C 语言的许多特性，使得有编程背景的人能够快速上手。

Verilog 的强大之处在于它的多抽象层次建模能力，支持从系统级到门级的各种设计。它允许设计者以并行的方式思考问题，同时描述多个硬件组件的协同工作。Verilog 代码可以被综合成实际的硬件电路，也可以用于仿真验证设计的正确性。

在进行实际的处理器/数字电路设计的过程，通常都指的是 Verilog 代码的编码过程，设计人员会使用 Verilog 来描述电路的结构和行为，从而实现复杂的逻辑功能。与软件设计不同，硬件设计需要考虑时序、并行性和资源限制等多方面的因素，并且在设计的时候是遵循自顶向下的设计方法，这意味着设计人员通常从系统级别的总体框架或者顶层模块开始，逐步细化到具体的模块和子模块，直到实现每个逻辑门和触发器的详细设计，如图 2 所示，相同颜色的表示的是相同的层级（或者叫 hierarchy），一个完整的设计会包括多个模块，这种分层设计方法有助于管理复杂的设计项目，并确保每个模块都能独立验证和优化。

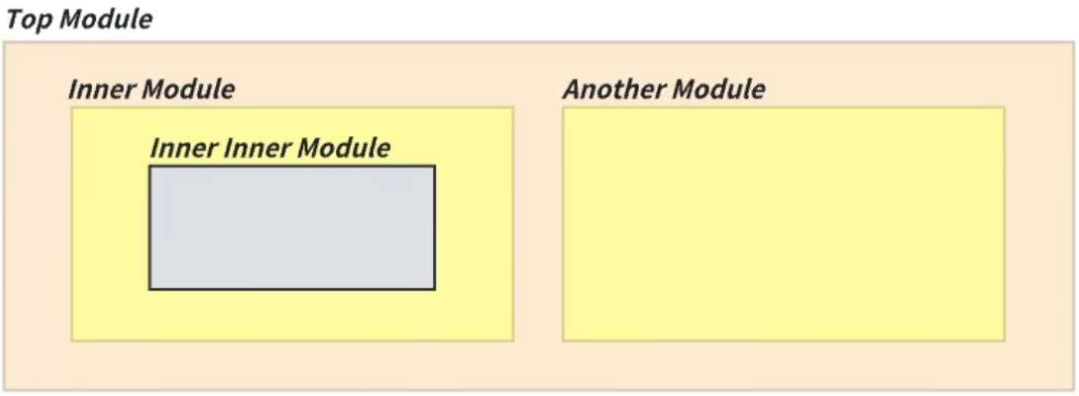


图 2 硬件模块的层级

对于熟悉软件设计的同学，我们可以把硬件设计的过程映射到我们的软件设计中，以 C 语言为例，我们可以把 main 函数看作是顶层模块，其他函数和子程序则对应于硬件设计中的子模块。具体来说，硬件设计中的各个模块可以类似于软件设计中的函数库，每个库负责实现特定的功能，在硬件设计中，顶层模块（类似于 main 函数）通常负责总体的控制和数据流管理，就像在 C 程序中 main 函数负责调用其他函数并控制程序的执行流程一样。顶层模块会实例化多个子模块（类似于调用多个函数），每个子模块实现具体的功能，例如数据处理、信号生成或状态管理。

硬件设计和软件设计有很多不一样的地方，编写 HDL 代码需要考虑很多的硬件相关因素，作为 HDL 代码编写人员同样需要了解一些硬件设计中的重要概念：

1）时序：硬件设计中需要考虑信号的时序，即信号在电路中传播的时间。这与软件设计中的执行时间不同，因为硬件电路是并行工作的，而软件通常是顺序执行的。这就需要设计者在设计时考虑信号的同步和延迟，确保电路在时钟边沿正确工作；

2）并行性：硬件电路天然是并行运行的，多个模块可以同时处理数据。这与软件的串行执行有本质的不同。在硬件设计中，设计者需要充分利用并行性来提高电路的性能；

3）资源限制：硬件设计需要考虑物理资源的限制，例如逻辑门的数量、布线的长度和面积等。这些限制决定了设计的复杂度和实现的可能性。在软件设计中，我们关注的是内存和计算资源的使用效率。

在完成 Verilog 设计之后，我们需要验证我们的设计的行为是否正确，验证功能的这一过程称为功能性仿真（Functional Simulation，或者直接叫仿真 / 验证），事实上除了功能性仿真之外，还有时序仿真，但是这个不是我们实验的重点（感兴趣的同学可以自行搜索查阅相关资料），我们主要关注的是功能性仿真。功能性仿真是在不考虑时序的情况下验证设计的逻辑功能。它主要用于验证 Verilog 代码的行为是否正确，确保设计逻辑在不同的输入条件下都能按预期工作，Verilog 提供了一些可综合的语法以及不可综合的语法，例如，initial 块和 task 函数是不可综合

的语法，这些语法主要用于仿真测试，而不会被综合工具转化为硬件逻辑。为了进行功能性仿真，我们需要编写一个测试平台（testbench），它的主要作用是为设计提供输入激励（stimulus），并监视设计的输出结果。一个典型的测试平台包括以下几个部分：信号声明、实例化待测模块（Device Under Test， DUT）、生成输入激励、监控输出结果及仿真控制。

Verilog 的语法内容相对较多，实验内容篇幅有限，不在这里详细展开介绍，在本小节后面提供一些参考学习资料，提供给感兴趣的同学进行进一步学习。请注意，Verilog 对于从事处理器设计/数字电路设计的人来说是必须了解甚至精通的，且 Verilog 已经成为了一种工业标准，几乎所有的硬件设计相关的 EDA 软件都支持 Verilog，但是因为 Verilog 发明时间较早，且语法较为生硬，实际上已经难以满足现代性能处理器的设计需求，因此也就出现了很多的 Verilog 的替代，例如可以看作是 Verilog 加强版的 SystemVerilog，或者是本课程使用的 Chisel，不过 Chisel 本身是需要将其编译成 Verilog 的，可以简单把 Chisel 和 Verilog 的关系类比为 C 语言和汇编的关系，因此对 Verilog 有一定了解或者说对传统的数字电路设计流程有一定了解对于后面学习 Chisel 语言会有很大的帮助，本次实验只是简单的介绍，后续需要同学自行了解相关的内容。

下面给出一个简单的 Verilog 代码的例子（4-bit 的加法器），主要包括三个部分的内容：

1）模块声明：使用 module 关键字定义模块，模块名为 adder4；

2）端口声明：定义模块的输入和输出端口，包括两个 4 位输入 A 和 B、一个进位输入 Cin、一个 4 位的和输出 Sum 以及一个进位输出 Cout，位于 input / output 后面的则是这个端口的硬件类型，这里定义的是 wire 类型；

3）逻辑描述：使用 assign 语句将输入 A、B 和 Cin 的和赋值给 {Cout, Sum}。

```
1.  // 4-bit Adder Module
2.  module adder4 (
3.      input  wire [3:0] A,    // 4-bit input A
4.      input  wire [3:0] B,    // 4-bit input B
5.      input  wire       Cin,  // Carry-in
6.      output wire [3:0] Sum,  // 4-bit Sum output
7.      output wire       Cout  // Carry-out
8.  );
9.
10.     assign {Cout, Sum} = A + B + Cin;
11. endmodule
```

以下则是一个简单的功能性仿真测试平台（testbench）的示例，用于验证一个 4 位加法器模块：

1）模块例化：模块实例化是指在测试平台或其他模块中创建并连接一个模块的过程，通过实

例化，可以在不同的上下文中重复使用模块

- adder4：这是被测试的模块的名称。
- uut：这是实例化的模块的实例名，通常表示 "Unit Under Test"（被测试单元）。
- (.A(A), .B(B), .Cin(Cin), .Sum(Sum), .Cout(Cout))：这是端口连接的列表。左边是模块 adder4 的端口名，右边是测试平台中的信号名。

2）initial 块：initial 块是 Verilog 中的一种语句块，用于在仿真开始时执行一次。在测试平台中，initial 块通常用于设置初始条件、应用测试向量以及控制仿真结束。

- initial：声明一个初始块，块中的语句将在仿真开始时执行一次。
- begin ... end：表示初始块的开始和结束。

3）延迟：在 Verilog 中，延迟语句用于控制仿真时间的推进。通过延迟语句，可以模拟信号随时间的变化。

- #10：延迟 10 个时间单位。时间单位可以在仿真的时间尺度中定义（例如，纳秒、微秒等）。

4）$display: $display 是 Verilog 中的系统任务，用于在仿真控制台输出信息。它的功能类似于 C 语言中的 printf。

- $display：系统任务，用于格式化输出信息。
- "A=%b B=%b Cin=%b | Sum=%b Cout=%b"：格式化字符串，其中 %b 表示以二进制格式显示变量。
- A, B, Cin, Sum, Cout：要显示的变量列表。

```
1.  // Testbench for 4-bit Adder
2.  module tb_adder4;
3.
4.      reg [3:0] A;     // 4-bit input A
5.      reg [3:0] B;     // 4-bit input B
6.      reg       Cin;   // Carry-in
7.      wire [3:0] Sum; // 4-bit Sum output
8.      wire      Cout; // Carry-out
9.
10.     // Instantiate the adder4 module
11.     adder4 uut (
12.         .A(A),
13.         .B(B),
14.         .Cin(Cin),
15.         .Sum(Sum),
16.         .Cout(Cout)
```

```
    );

    initial begin
        // Test cases
        A = 4'b0001; B = 4'b0010; Cin = 0;
        #10; // Wait for 10 time units
        $display("A=%b B=%b Cin=%b | Sum=%b Cout=%b", A, B, Cin, Sum, Cout);

        A = 4'b0101; B = 4'b0110; Cin = 1;
        #10;
        $display("A=%b B=%b Cin=%b | Sum=%b Cout=%b", A, B, Cin, Sum, Cout);

        A = 4'b1111; B = 4'b0001; Cin = 0;
        #10;
        $display("A=%b B=%b Cin=%b | Sum=%b Cout=%b", A, B, Cin, Sum, Cout);

        $finish; // End simulation
    end

endmodule
```

Verilog 参考学习资料：

- https://en.wikipedia.org/wiki/Verilog
- https://www.runoob.com/w3cnote/verilog-tutorial.html
- https://www.chipverify.com/tutorials/verilog

3.3. Chisel 简介

Chisel 是一个开源的硬件描述语言（HDL），由 UC Berkeley 于 2012 年发布第一版 Chisel，至今已有十几年，逐渐成为了学术界以及部分工业界所采用的高级硬件描述语言，专门用于数字电路设计，特别是在集成电路和 FPGA 设计领域。它基于 Scala 编程语言，提供了一种现代化、高效且灵活的方式来描述和生成硬件电路。Chisel 的优势在于它结合了高级编程语言的抽象能力和硬件描述语言的精确性，使得设计者能够以更高的层次来思考和构建复杂的硬件系统。

Chisel 的设计理念强调模块化和可重用性。通过使用 Scala 的高级特性，如函数式编程和对象继承，设计人员可以创建参数化的硬件模块，这些模块可以在不同的设计中重复使用。这种方法不仅提高了设计效率，还减少了代码的冗余和错误的可能性。另外 Chisel 也提供了强大的类型系

统和编译时检查功能，帮助设计者在早期阶段捕获设计错误，从而降低调试成本。与 Verilog 相比，Chisel 在设计方法和抽象层次上提供了显著的改进。Verilog 是一种传统的硬件描述语言，其设计风格通常较为低级，关注的是具体电路的细节。这种方法虽然直观，但在处理大型和复杂系统时，容易导致代码冗长、难以维护和调试。相比之下，Chisel 利用 Scala 的高级语言特性，使得设计者能够在更高的抽象层次上工作。通过函数式编程、对象继承和参数化模块，Chisel 允许设计者更灵活地定义和组合硬件组件，从而大大简化了复杂系统的设计过程。

Chisel 编写的代码可以通过通过 Chisel 的编译器生成 Verilog 代码，这些 Verilog 代码可以直接用于传统的硬件综合工具链或者其他 EDA 工具中，所以本质上 Chisel 并没有抛弃传统的硬件设计流程，而是对其进行了现代化的改进和提升。这样设计者既能够享受到高级语言带来的便利和效率，又可以无缝对接现有的硬件设计和验证流程。这种双重优势使得 Chisel 在硬件设计领域中越来越受到欢迎，通过 Chisel，硬件设计变得更加高效、灵活且易于维护，这也是为什么我们这门课程使用 Chisel 的原因。

基于上一小节的 Verilog 代码例子，在这里我们给出一个实现相同逻辑功能的 Chisel 版本的代码例子，当然从目前这个例子来说还不能展示 Chisel 的强大，在这里只是对 Chisel 的语法做一个简单直观的展现。

```
// Import Chisel library
import chisel3._

class Adder4 extends Module {
  // Define input and output ports
  val io = IO(new Bundle {
    val A    = Input(UInt(4.W))  // 4-bit input A
    val B    = Input(UInt(4.W))  // 4-bit input B
    val Cin  = Input(Bool())     // Carry-in
    val Sum  = Output(UInt(4.W)) // 4-bit Sum output
    val Cout = Output(Bool())    // Carry-out
  })

  // Perform the addition and assign to outputs
  val result = io.A +& io.B + io.Cin
  io.Sum := result(3, 0)       // Lower 4 bits
  io.Cout := result(4)         // Carry-out bit
}
```

值得注意的是 Chisel 生成 Verilog 代码（或者 SystemVerilog）的时候，使用了一种特有的中间代码表示：FIRRTL，生成的 FIRRTL 代码再经过 FIRRTL 编译器（firtool）后生成了 Verilog，这与 LLVM 的 IR 概念类似（因此最新的 FIRRTL 也被收录到了 LLVM 项目里的 CIRCT 中），

Chisel 能够实现的大部分的编译优化基本上都是得益于这一编译流程，感兴趣的同学可以搜索资料进行了解。

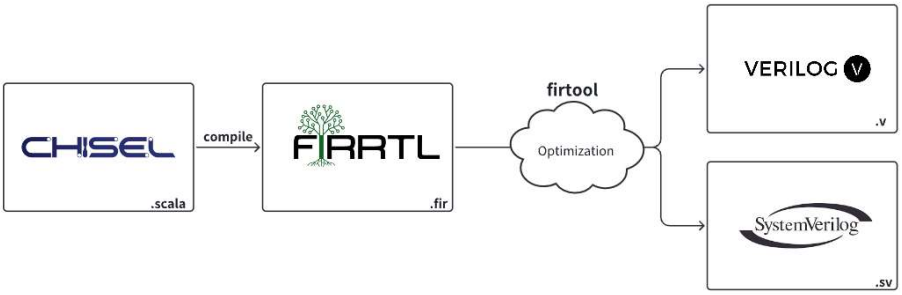


图 3 Chisel、FIRRTL、Verilog/SystemVerilog 的关系

3.4. Scala 入门

Chisel 语言使用 Scala 实现，因此我们首先需要介绍 Scala 的相关语法与特性。Scala 是一种支持多种编程范式的编程语言，Chisel 选择使用它有几个重要原因。首先，Scala 是一种适合托管嵌入式 DSL（领域特定语言）的优秀语言，这使得 Chisel 能够利用 Scala 的语法和特性来创建灵活且强大的硬件描述语言。其次，Scala 有一个强大且优雅的库，用于操作各种数据集合，这对于硬件设计中的数据处理和操作非常有用。此外，Scala 的严格类型系统可以在开发周期的早期（即在编译时）捕获大量错误，从而提高代码的可靠性和稳定性。Scala 还提供了强大的表达和传递函数的方式，使得代码更加简洁和可维护。这些优点在后面介绍 Chisel 时会更加明显，但在本小节中，我们将主要对 Scala 本身进行介绍，以便更好地理解 Chisel 的设计理念和实现方法。

1）变量和常量 - var 和 val

用于创建（可变）变量和常量值（不可变变量）的语句分别以关键字 var 和 val 开头，在大部分情况下，编写我们通常是使用 val。为什么？主要是为了减少在代码中以错误的方式重复使用变量的几率，或者避免使代码变得难以阅读，下面是几个例子：

- 与 Java 和 C 不同，Scala 通常不需要在语句末尾加上分号；
- 当存在换行符时，Scala 会推断出分号。例如，当一条语句分布在多行上，且每行的末尾是一个需要后续代码的运算符时，Scala 通常能够识别出来；
- 唯一需要分号的情况是当你想在一行中写多条语句时；
- 变量的使用方式很直观，两个 var 变量可以被重新赋值，而两个 val 变量在创建后则是不可变的。

```
var numberOfKittens = 6
val kittensPerHouse = 101
val alphabet = "abcdefghijklmnopqrstuvwxyz"
var done = false

numberOfKittens += 1
// kittensPerHouse = kittensPerHouse * 2 // This would not compile ; kittensPerHouse is not updatable
done = true
```

2）条件语句

Scala 的条件语句和绝大多数的编程语言在形式和逻辑上大部分类似，基本的用法如下所示：

```
// A simple conditional; by the way, this is a comment
if (numberOfKittens > kittensPerHouse) {
    println("Too many kittens!!!")
}
// The braces are not required when all branches are one liners. However, the
// Scala Style Guide prefers brace omission only if an "else" clause is included.
// (Preferably not this, even though it compiles...)
if (numberOfKittens > kittensPerHouse)
    println("Too many kittens!!!")

// ifs have else clauses, of course
// This is where you can omit braces!
if (done)
    println("we are done")
else
    numberOfKittens += 1

// And else ifs
// For style, keep braces because not all branches are one liners.

if (done) {
    println("we are done")
}
else if (numberOfKittens < kittensPerHouse) {
    println("more kittens!")
    numberOfKittens += 1
}
else {
    done = true
}
```

但在 Scala 中，"if" 条件语句会返回一个值，这个值是什么呢？它由所选择分支的最后一行决定。在用于函数和类的值初始化时非常强大，其形式如下：

```
val likelyCharactersSet = if (alphabet.length == 26)
    "english"
else
    "not english"

println(likelyCharactersSet)
```

3）方法（函数）

方法使用关键字 def 定义，但我们也可以将它们也称为函数。函数参数（或称为参数）以逗号分隔的列表形式指定，包括参数的名称、类型以及可选的默认值，同时为了清晰起见，应该指定函数的返回类型。在 Scala 中，没有参数的函数不需要使用空括号，在某些情况下，这会让开发者更轻松，比如当类的成员变成函数，因为引用它时需要进行一些计算，按照惯例，没有副作用的无参数函数（即调用它们不会改变任何东西，只是返回一个值）不使用括号，而有副作用的函数（可能它们会改变类的变量或输出一些内容）则应该使用括号。下面是两个例子：

```
// Simple scaling function with an input argument, e.g., times2(3)
 returns 6
// Curly braces can be omitted for short one-line functions.
def times2(x: Int): Int = 2 * x

// More complicated function
def distance(x: Int, y: Int, returnPositive: Boolean): Int = {
    val xy = x * y
    if (returnPositive) xy.abs else -xy.abs
}
```

Scala 中支持函数重载（Overloading），也就是同一个函数名可以以多种方式使用，参数及其类型决定了一个签名，编译器可以通过这个签名来确定应该调用哪个版本的函数。例如下面的代

码片段中 times2 函数能够接收两个不同的输入类型，这就是使用函数重载实现的，函数重载能够让我们复用大部分的代码片段，提到代码的简洁度与可读性。

```
// Overloaded function
def times2(x: Int): Int = 2 * x
def times2(x: String): Int = 2 * x.toInt

times2(5)
times2("7")
```

任何的语言的函数都支持递归，并且 Scala 中还支持嵌套函数，也就是在函数里面再定义函数，Scala 中花括号用于定义代码作用域，在一个函数的作用域内可以包含更多的函数或递归函数调用。在特定作用域内定义的函数只能在该作用域内访问。

```
/** Prints a triangle made of "X"s
  * This is another style of comment
  */
def asciiTriangle(rows: Int) {

    // This is cute: multiplying "X" makes a string with many copies of "X"
    def printRow(columns: Int): Unit = println("X" * columns)

    if(rows > 0) {
        printRow(rows)
        asciiTriangle(rows - 1) // Here is the recursive call
    }
}

// printRow(1) // This would not work, since we're calling printRow outside its scope
asciiTriangle(6)
```

Scala 中的函数还支持命名参数以及参数默认值，例如对于下面的代码片段：

```
def myMethod(count: Int, wrap: Boolean, wrapValue: Int = 24): Unit = { ... }
```

在调用方法的时候可以显式指明具体的参数值，这样可以忽略参数的顺序，当某些参数有默认值（不需要被覆盖）时，调用者只需传递（按名称）不使用默认值的特定参数。注意，参数 wrapValue 有一个默认值 24。

```
myMethod(count = 10, wrap = false, wrapValue = 23)
myMethod(wrapValue = 23, wrap = false, count = 10)
myMethod(wrap = false, count = 10)
```

4）列表

Scala 实现了多种聚合或序列对象，列表与数组非常相似，但支持附加和提取等额外的相关函数操作。

```
val x = 7
val y = 14
val list1 = List(1, 2, 3)
val list2 = x :: y :: y :: Nil       // An alternate notation for assembling a list

val list3 = list1 ++ list2           // Appends the second list to the first list
val m = list2.length
val s = list2.size

val headOfList = list1.head          // Gets the first element of the list
val restOfList = list1.tail          // Get a new list with first element removed

val third = list1(2)                 // Gets the third element of a list (0-indexed)
```

5）for 循环语句

Scala 有一个 for 语句，它的工作方式与传统的 for 语句类似，可以用来遍历一系列的值。

```
for (i <- 0 to 7) { print(i + " ") }
println() // 0 1 2 3 4 5 6 7
```

如果不使用 to 而是使用 until，则会打印 0 ~ 6。

```
for (i <- 0 until 7) { print(i + " ") }
println() // 0 1 2 3 4 5 6
```

添加一个 by 来按固定的增量递增，以下代码会打印出 0 到 10 之间的偶数。

```
for(i <- 0 to 10 by 2) { print(i + " ") }
println()
```

如果你有某种集合并想访问所有元素，可以像在 Java 和 Python 中一样使用 for 作为迭代器，这里我们创建了一个包含 4 个随机整数的列表，然后对它们求和。

```
val randomList = List(scala.util.Random.nextInt(), scala.util.Random.nextInt(), scala.util.Random.nextInt(), scala.util.Random.nextInt())
var listSum = 0
for (value <- randomList) {
  listSum += value
}
println("sum is " + listSum)
```

Scala 语言中的 for 循环有更多的技巧，它能够直观地满足广泛的传统迭代需求，但使用起来可能并不总是最方便的，像对数组元素求和这样的操作，通常更容易通过使用一个特定的函数来完成，这个函数族被称为集合推导（comprehensions），它适用于许多不同类型的元素集合。

6）代码块

代码块用花括号分隔，一个代码块可以包含零行或多行 Scala 代码，最后一行 Scala 代码成为代码块的返回值（可能会被忽略）。一个没有任何行的代码块将返回一个特殊的类似于 null 的对象，称为 Unit。代码块在 Scala 中被广泛使用：它们是类定义的主体，构成函数和方法定义，它们是 if 语句的子句，它们也是 for 和许多其他 Scala 操作符的主体。下面是一个参数化代码

块的，代码块可以接受参数。在类和方法定义的情况下，这些参数看起来像大多数传统编程语言中的参数。在下面的示例中，c 和 s 是代码块的参数（实际上就是函数的函数体）。

```
// A one-line code block doesn't need to be enclosed in {}
def add1(c: Int): Int = c + 1

class RepeatString(s: String) {
  val repeatedString = s + s
}
```

代码块被传递给类 List 的方法 map，map 方法要求其代码块有一个单一参数。代码块会对列表的每个成员调用，并将该成员转换为字符串后返回。Scala 对这种语法的变体几乎过于宽容。你可能会看到许多不同的写法。这种类型的代码块称为匿名函数，有关匿名函数的更多详细信息将在后面的模块中提供。

```
val intList = List(1, 2, 3)
val stringList = intList.map { i =>
  i.toString
}
```

7）包和包的导入

下面定义了一个 package，以及 package 内的一个 class

```
package mytools
class Tool1 { ... }
```

当你需要引用包含上述代码的文件中定义的代码时，应当使用：

```
import mytools.Tool1
```

包名应该与目录层次结构相匹配，但这也不是强制性的，同时按照惯例，包名是小写的，并且不包含像下划线这样的分隔符，因为有时会使得创建可读性强的描述性名称变得困难

正如上面所示，import 语句通知编译器我们正在使用一些额外的库，在编写 Chisel 代码时，可以使用的一些常见 import 包括：

- 第一个导入了 chisel3 包中的所有类和方法，这里的下划线用作通配符；
- 第二个从 chisel3.iotesters 包中导入特定的类。

```
import chisel3._
import chisel3.iotesters.{ChiselFlatSpec, Driver, PeekPokeTester}
```

8）Scala 的面向对象

Scala 是面向对象的，了解这一点对于充分利用 Scala 和 Chisel 非常重要，需要注意的是有多种方式可以描述 Scala 中的面向对象这一特性：

- 变量是对象
- 从 Scala 的 val 声明来看，常量也是对象
- 即使是字面值也是对象
- 即使是函数本身也是对象。稍后我们会详细介绍这一点
- 对象是类的实例，事实上，在 Scala 中，面向对象中的对象在几乎所有重要的方面都被称为实例。
- 在定义类时，程序员需要指定
  - 与类关联的数据（val，var）
  - 类的实例可以执行的操作，称为方法或函数
- 类可以扩展其他类
  - 被扩展的类是超类；扩展的类是子类
  - 在这种情况下，子类继承超类的数据和方法
  - 有许多有用但受控的方法可以让一个类扩展或重写继承的属性

- 类可以从特质（trait）继承。可以将特质视为轻量级的类，它允许以特定、有限的方式从多个超类继承
- （Singleton，单例）对象是 Scala 类的一种特殊类型

下面是一个 Class 的例子，其中：

- class WrapCounter 这是 WrapCounter 的定义
- (counterBits: Int) 表示创建一个 WrapCounter 需要一个整数参数，其名称清晰地表明此参数是计数器的位宽
- 花括号 （{}） 用于分隔代码块。大多数类使用代码块来定义变量、常量和方法（函数）
- max 和 counter 通常被称为类的 成员变量
- 方法 inc 的主体是一个代码块，作用是增加当前 Class 中的 counter 的值并返回
- 最后一行的通过 println 将能计数的最大值打印出来

```
1.  // WrapCounter counts up to a max value based on a bit size
2.  class WrapCounter(counterBits: Int) {
3.
4.    val max: Long = (1 << counterBits) - 1
5.    var counter = 0L
6.
7.    def inc(): Long = {
8.      counter = counter + 1
9.      if (counter > max) {
10.         counter = 0
11.     }
12.     counter
13.   }
14.   println(s"counter created with max value $max")
15. }
```

在定义完 Class 后，就需要进行 Class 的实例化，也就是创建 Class 实例，我们可以使用 new 关键字创建一个上面提到的 Class，你可能会看到实例在创建时不使用 new 关键字，例如，

val y = WrapCounter(6)，这种情况经常发生，值得特别注意，但这需要使用伴生对象（Scala 的 object 的 apply 方法，感兴趣可以提前搜索资料了解，我们在后续的实验中将会使用或提及）。

```
val x = new WrapCounter(2)
x.inc() // Increments the counter

// Member variables of the instance x are visible to the outside, unless they are declared private
if(x.counter == x.max) {
    println("counter is about to wrap")
}

x inc() // Scala allows the dots to be omitted; this can be useful for making embedded DSL's look more natural
```

3.5. Scala CLI 环境安装与测试

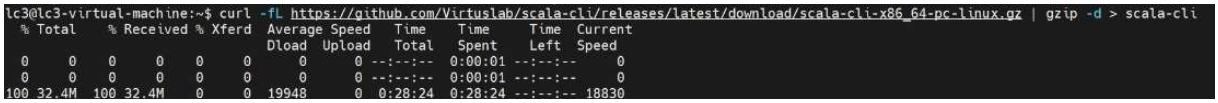
在本节中，我们将安装 Scala CLI，并用来运行 Scala 代码以及一个来自 Chisel 官方的 Chisel 实例代码，Scala CLI 是一个强大的工具，可以简化 Scala 项目的构建和管理。它集成了多种功能，使得开发、测试和部署 Scala 应用变得更加方便。例如 Scala CLI 能够直接运行一个独立的 Scala 脚本、管理依赖、打包应用程序等，这对于我们快速搭建一个可用的 Scala 环境非常有帮助，下面提供 Scala CLI 的安装与测试步骤（如果你使用的是实验提供的虚拟机镜像，则不需要安装 Scala CLI，虚拟机中默认已经安装好）：

在虚拟机中已经默认安装好了 vs-code，在命令行中输入 code 打开。

（1）从 github release 中下载 Scala CLI

需要注意的是，由于 Scala CLI 需要从 github 下载，请确保你的网络环境能够顺畅访问 github，否则可能会出现下载失败或者下载时间很长的情况。

```
curl -fL https://github.com/Virtuslab/scala-cli/releases/latest/download/scala-cli-x86_64-pc-linux.gz | gzip -d > scala-cli
```



如果你使用的是实验提供的虚拟机镜像，可以直接前往 ~/Downloads 目录复制 scala-cli 这个文件，这样就不需要使用 curl 进行下载。

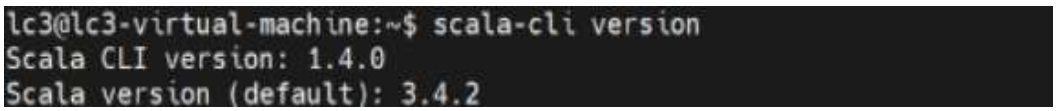
（2）设置下载的可执行文件为可执行的权限并将其移动到用户的 bin 文件夹下

```
chmod +x scala-cli
sudo mv scala-cli /usr/local/bin/scala-cli
```

（3）验证 Scala CLI 是否安装成功

```
scala-cli version
```

如果安装成功命令行将会出现下面的信息（版本信息可能会略有不同）



（4）简单测试 Scala CLI 是否能正常工作

注意：执行 Scala CLI 的时候，如果缺少相关依赖库，Scala CLI 则会自动下载并配置依赖库，这可能会耗费一定的时间。

在命令行直接输入 scala-cli，载入后输入 println("hello world")，能正常输出 "hello world" 则说明 Scala CLI 能够正常工作。

```
lc3@lc3-virtual-machine:~$ scala-cli

Downloading JVM temurin:17
Welcome to Scala 3.4.2 (17, Java OpenJDK 64-Bit Server VM
Type in expressions for evaluation. Or try :help.

scala> println("hello world")
```

（5）通过 Scala CLI 执行单独的 scala 文件

Scala CLI 除了可以在命令行直接使用，一行一行执行 Scala 代码外，还能直接执行一个 scala 文件，例如我们创建下面一个简单的 Hello World 程序（可以使用命令行的 vim 进行创建或者使用 gedit 图形化界面进行创建），称为 Hello.scala，内容如下：

```
object Hello {
  def main(args: Array[String]): Unit = {
    println("Hello, Scala!")
  }
}
```

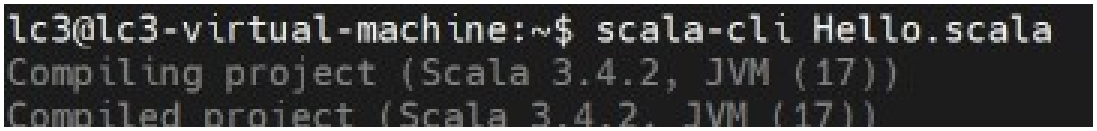
使用 scala-cli 命令执行上述的文件：

```
scala-cli Hello.scala
```

可以在命令行中得到如下的输出：

```
lc3@lc3-virtual-machine:~$ scala-cli Hello.scala
Downloading compilation server 1.5.17-sc-2
Starting compilation server
Compiling project (Scala 3.4.2, JVM (17))
```

可以看到 scala-cli 成功执行了其中的 main 函数，后续的实验作业可以采用 scala-cli 执行文件的方式来执行编写的 scala 代码，这里解释一下为什么会自动运行这个 main 函数，这是因为在 Scala 中，约定将程序的入口点定义在名为 main 的方法中。当您运行一个 Scala 程序时，系统会自动查找并执行具有特定签名的 main 方法，即 `def main(args: Array[String]): Unit`，如果我们将 main 函数名改为 NotMain，再次执行上述命令，则会得到下面的报错信息:



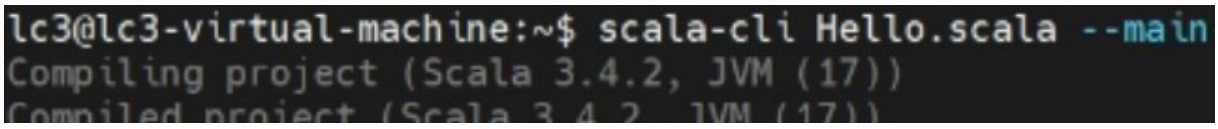
还有我们可以发现上述代码中定义了一个 object，名为 Hello，在 Scala 中大多数的程序实体都是用 object 包裹起来管理的，作为一个单例对象，如果我们的文件中有多个 object，都有 main 函数，那么在使用 scala-cli 执行的时候需要使用 scala-cli 的--main-class 参数来指定具体执行的是哪一个 object，例如考虑如下的代码：

```
object Hello {
  def main(args: Array[String]): Unit = {
    println("Hello, Scala!")
  }
}

object Another {
  def main(args: Array[String]): Unit = {
    println("Hello from Another object!")
  }
}
```

此时如果我们想要执行 Another 这个 object 下的 main，可以执行如下的命令：

```
scala-cli Hello.scala --main-class Another
```



3.6. 使用 Scala CLI 运行 Chisel 官方提供的例程

使用下面的命令下载 Chisel 官方提供给 Scala CLI 的例子：

```
curl -O -L
https://github.com/chipsalliance/chisel/releases/latest/download/
chisel-example.scala
```

如果你使用的是实验提供的虚拟机镜像，可以直接前往~/Downloads 目录复制 chisel-example.scala 这个文件，这样就不需要使用 curl 进行下载。

代码内容如下：

需要注意的是代码开头的`//>`等开头的字段是 Scala CLI 用于识别依赖的特殊语法，实际的 Chisel 代码不会包括这部分内容。

```
//> using scala "2.13.12"
//> using dep "org.chipsalliance::chisel:6.4.0"
//> using plugin "org.chipsalliance:::chisel-plugin:6.4.0"
//> using options "-unchecked", "-deprecation", "-language:reflectiveCalls", "-feature", "-Xcheckinit", "-Xfatal-warnings", "-Ywarn-dead-code", "-Ywarn-unused", "-Ymacro-annotations"

import chisel3._
// _root_ disambiguates from package chisel3.util.circt if user imports chisel3.util._
import _root_.circt.stage.ChiselStage

class Foo extends Module {
  val a, b, c = IO(Input(Bool()))
  val d, e, f = IO(Input(Bool()))
  val foo, bar = IO(Input(UInt(8.W)))
  val out = IO(Output(UInt(8.W)))

  val myReg = RegInit(0.U(8.W))
  out := myReg

  when(a && b && c) {
    myReg := foo
  }
  when(d && e && f) {
    myReg := bar
  }
}

object Main extends App {
  println(
    ChiselStage.emitSystemVerilog(
      gen = new Foo,
      firtoolOpts = Array("-disable-all-randomization", "-strip-debug-info")
    )
  )
```

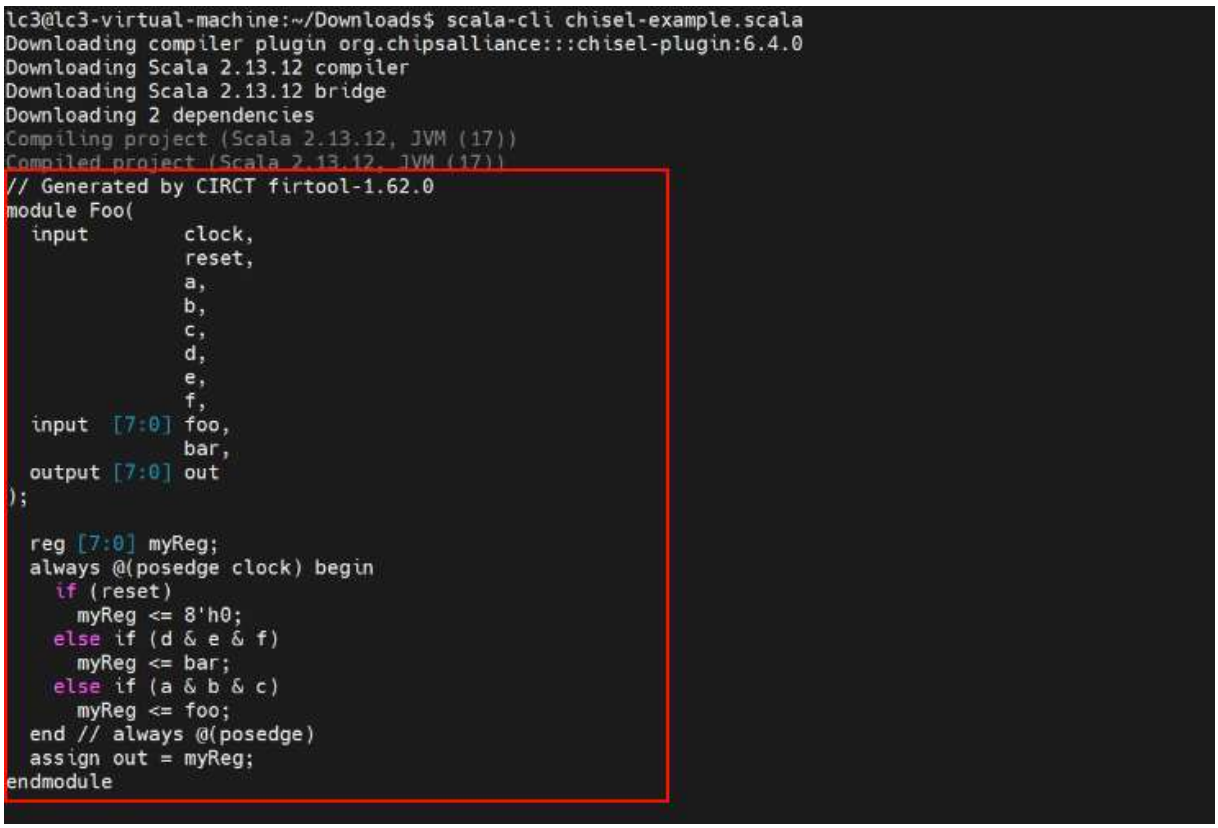
```
34. }
```

这里使用了 `object Main extends App` 的写法，实际上实现的功能和我们上一小结手动实现 main 函数一样的功能，可用看作是一种简化的写法。

接下来使用 scala-cli 命令进行运行（同样可能会需要一段时间来下载各种依赖）：

```
scala-cli chisel-example.scala
```

成功运行将会得到如下的结果，其中红色方框的内容即为生成的 Verilog：



任务一：根据实验指导手册，完成开发环境的搭建，或导入虚拟机镜像，并且能够编译运行 Chisel 模板工程的编译与测试，需要能够在命令行中看到运行的输出信息。

任务二：根据实验指导，安装 Scala CLI 并测试可用性，之后运行 Chisel 官方提供的代码例子，在命令行中得到 Chisel 编译输出的 Verilog 代码。

任务三：使用 Scala 创建和操作一个简单的计数器类。

要求：

你需要创建一个名为 SimpleCounter 的类，该类具有以下功能：

- 接受一个整数参数 maxValue 来定义计数器的最大值。
- 使用一个变量 currentValue 来存储计数器的当前值。
- 该类应包含以下方法：increment(): 将 currentValue 增加 1。如果 currentValue 达到 maxValue，则将 currentValue 重置为 0。
- decrement(): 将 currentValue 减少 1。如果 currentValue 小于 0，则将 currentValue 设置为 maxValue。
- reset(): 将 currentValue 重置为 0。
- isMax(): 返回一个布尔值，指示 currentValue 是否等于 maxValue。
- 在类的构造函数中打印初始的 currentValue 和 maxValue。
- 创建一个对象实例并测试上述方法。

步骤：

- 定义类 SimpleCounter：定义一个类 SimpleCounter，该类接受一个整数参数 maxValue。
    - 在类的构造函数中，定义一个变量 currentValue 并初始化为 0（需要使用 var，而不是 val）。
    - 在类的构造函数中打印初始的 currentValue 和 maxValue。
- 实现方法：
    - increment(): 实现将 currentValue 增加 1 的方法。
    - decrement(): 实现将 currentValue 减少 1 的方法。
    - reset(): 实现将 currentValue 重置为 0 的方法。

- isMax(): 实现返回 currentValue 是否等于 maxValue 的方法。
- 创建对象并测试方法：创建一个 SimpleCounter 类的实例，并测试各个方法，验证它们的功能。

在此给出需要实现的代码框架：

SimpleCounter.scala

```
// 定义 SimpleCounter 类
class SimpleCounter(maxValue: Int) {
    // TODO: 实现你的代码
}

object TestSimpleCounter {
  def main(args: Array[String]): Unit = {
    // TODO: 实例化对象并测试你的功能是否正常，具体的测试形式不限，需要能够运行这份代码并在命令行看到相应的输出
    // ！本实验任务主要是测试基本的 Scala 语法的掌握情况，如果你已经对 Scala 相对熟悉，可以自行发挥这一实验任务的内容，但是至少需要包括 class 的创建以及测试
  }
}
```