本科毕业论文（设计）

面向 FPGA 设计的课程实践软件开发

DEVELOPMENT OF SCALEABLE EDA ASSISTANT FOR FPGA DESIGN

梁鑫嵘

哈尔滨工业大学

2024 年 5 月

密级：公开

本科毕业论文（设计）

面向 FPGA 设计的课程实践软件开发

本　科　生：梁鑫嵘

学　　号：200110619

指 导 教 师：徐勇教授

专　　业：计算机科学与技术

学　　院：深圳校区计算机科学与技术学院

答 辩 日 期：2024 年 05 月 23 日

学　　校：哈尔滨工业大学

摘　要

现场可编程门阵列（FPGA, Field-Programmable Gate Array）由于其灵活性和可编程性，被广泛应用于数字电路设计、信号处理、通信、机器学习计算加速等领域[1]。FPGA市场近年以每年超过 10% 的速度增长，市场规模正在快速扩大中。FPGA 不仅是数字电路设计的重要工具，也是计算机学科的同学们在了解计算机和学习数字电路设计时的重要学习工具。

然而，FPGA 设计的复杂性和困难性使得数字电路设计的初学者在学习 FPGA 设计时往往感到困难。为了帮助包括高校学生在内的数字电路设计学习者更好地学习 FPGA 设计，本文提出并设计了一款面向 FPGA 设计的课程实践软件 Scaleda，意为“Scalable EDA”。

Scaleda 的设计理念旨在降低学习门槛，提高学习效率。通过简化操作界面、提供易于理解的配置选项，以及解决常见编辑 HDL 代码时的问题，Scaleda 使学生能够更轻松地进行 FPGA 的开发和编程。其提供的代码解析、编辑、项目管理等功能，为学生提供了一个全面而易用的学习平台。

除了设计本身，本文还介绍了另一个关键组成部分：Rvcd 波形查看器的设计与开发。Rvcd 不仅支持波形文件的解析和查看，还实现了波形和源代码之间的交互、HDL 代码解析等功能，为学生提供了更加直观、高效的学习工具。

综上所述，本文提出并实现了一套完整的面向 FPGA 设计的课程实践软件，为学生提供了强大的学习工具，促进了数字电路设计教学的发展。通过开源的方式，本项目不仅能够为学生和初学者提供免费、易用的工具，还能够促进 FPGA 领域开源工具的发展，推动数字电路设计教育的开放与创新。这种开源的理念也符合当今数字领域的发展趋势，有利于激发更多人的学习兴趣，培养更多优秀的电子工程师和计算机科学家。项目全部源码已开源在 https://github.com/orgs/Scaleda/repositories，文档与教程也已发布在 https://scaleda.top。

关键词：FPGA；EDA；数字电路设计；开源；软件开发

Abstract

FPGA, characterized by its flexibility and programmability, has found extensive applications in various domains including digital circuit design, signal processing, communication, and machine learning computation acceleration[1]. The FPGA market has been experiencing rapid growth, with an annual increase exceeding 10% in recent years, indicating significant market expansion. FPGAs serve not only as crucial tools for digital circuit design but also as essential learning aids for comprehending computers and studying digital circuit design within the field of computer science.

However, the complexity and challenges associated with FPGA design often present obstacles for beginners in digital circuit design. To facilitate better learning experiences for learners, including university students, this paper proposes and designs Scaleda, a course practice software specifically tailored for FPGA design, denoted as 'Scalable EDA'.

Scaleda's design philosophy revolves around reducing the learning curve and enhancing learning efficiency. Through simplified user interfaces, easily understandable configuration options, and resolutions to common issues encountered during HDL code editing, Scaleda enables students to engage more seamlessly in FPGA development and programming, offering students a comprehensive and user-friendly learning platform.

Additionally, this paper introduces another pivotal component: the development of Rvcd waveform viewer. Rvcd not only facilitates waveform file parsing and visualization but also implements interactions between waveforms and source code, as well as HDL code parsing, thereby providing students with more intuitive and efficient learning tools.

In summary, this paper presents and realizes a comprehensive course practice software tailored for FPGA design, furnishing students with potent learning tools and advancing the development of digital circuit design education. Through open-source endeavors, we not only furnish students and beginners with free and accessible tools but also contribute to the advancement of open-source FPGA tools, propelling the openness and innovation of digital circuit design education. All project source code has been open-sourced at https://github.com/orgs/Scaleda/repositories, and documentation and tutorials have been published at https://scaleda.top.

Keywords: fpga, eda, digital circuit design, open source, software development

目　录

第 1 章 绪论

1.1 课题背景

FPGA是一种可编程逻辑器件，其能够通过编程修改内部的电路工作方式，实现不同的功能。其具有灵活性高、性能强、功耗低等优点，因此在数字电路设计、信号处理、通信等领域有着广泛的应用。

随着 FPGA 在通信、人工智能、IC 设计等领域的广泛应用，其在高校教育教学中也扮演着重要的教学工具角色。FPGA 提供了一个理想的平台，让学生能够直接参与数字电路设计、信号处理等领域的实际项目。然而，这需要学生熟练掌握 FPGA 的开发和编程，而这并不总是容易。

Verilog 作为 HDL 语言，其与传统的编程语言有很大的不同，初学者在学习过程中可能会遇到一些困难。在数字逻辑相关的实际课程教学和实验中，笔者发现许多学生遇到了非常大的困难，例如：不知道如何编写 Verilog 代码、不知道如何使用 EDA 工具、不知道如何进行仿真验证等。

学生需要使用专用的电子设计自动化（EDA）软件，以及硬件描述语言（HDL）如 Verilog 来编写数字电路的设计和控制。这些领域与通常的编程方法存在较大差异，初学者在学习过程中可能会遇到一些困难。此外，市场上的 EDA 工具通常设计用于专业工程师，对于学生来说可能过于复杂，这会对教学和学习的效率产生负面影响。

同学们和其他数字电路设计的初学者们面对 HDL 语言、EDA 软件工具等学习难题，需要花费大量经历学习，导致入门门槛高，学习受挫。

因此，本课题旨在设计和开发一款面向 FPGA 设计的课程实践软件，以帮助学生等数字电路设计初学者更好地学习 FPGA 的开发和编程。同时，该软件具有的友好的用户界面，易于使用的配置功能，能够为开源社区提供一个易于使用的 FPGA 开发工具，进一步推动开源 EDA 开发工具的发展。

1.2 研究的目的和意义

本项目首先是为了解决 FPGA 相关软件在高校教学中的应用问题，提高学生学习 FPGA 的效率。围绕这一目标，本项目的研究内容主要需要完成以下的设计目标:

1. 降低学生的入门难度，提高学习效率: 传统厂商提供的 EDA 软件通常面向工业环境，操作繁琐而功能复杂，并不适合入门学习。作为初学者，同学们需要花费大量时间阅读文档和手册，导致入门门槛高，学习吃力。本项目设计的 FPGA 开发工具通过简化操作界面，提供易于理解的配置选项，并倾力解决编辑 HDL 代码时的常见问题，以降低学生的学习难度，提高学习效率。

2. 增加学生的学习兴趣：通过简化操作界面，提供更加直观的配置选项，以及更加友好的错误提示，让学生能够更加专注于数字电路设计的本质，而不是被繁琐的操作所困扰。这将有助于提高学生的学习兴趣，激发学生的学习热情。

3. 提供创新独立的 FPGA 开发流程：本项目不仅提供一个独立于厂商 EDA 软件的 FPGA 开发工具，还设计提供了一套创新的支持多种 FPGA 工具链联合开发的开发流程。这意味着同学们可以更加灵活地选择不同的工具链来完成学习任务，其他用户也能用以提升 HDL 代码的跨平台应用特性，从而推动 FPGA 开发工具的发展。

4. 提供丰富的未来发展空间：作为一款灵活、通用的 FPGA 开发平台，本项目提供了丰富的未来发展空间。可以预见，通过顺应 FPGA 领域的开源潮流，本项目能够在未来得到更多的用户支持和社区贡献，从而不断完善和发展。

综上所述，本项目的研究目的首先是为了解决 FPGA 相关软件在高校教学中的应用问题，提高学生学习 FPGA 的效率，然后是为 FPGA 以及数字电路设计领域的发展做出开源贡献。这将有助于培养更多优秀的电子工程师和计算机科学家，并对高校计算机数字逻辑部分的教学质量和效果产生积极影响，为提升我国高校在数字电路和 FPGA 领域的教育和研究水平作出贡献。

1.3 国内外研究现状

1.3.1 FPGA 原理与开发流程

FPGA 是一种可编程逻辑器件，其内部由大量的可编程逻辑单元和存储单元（BRAM）、时钟控制（PLL/MMCM）等 ASIC 硬件处理模块组成，通过编程修改内部的电路工作方式，实现不同的功能。

FPGA 内部具有以查找表为基础的可编程逻辑单元，可以实现任意的逻辑功能。以 Xilinx 7-series 为例，其 FPGA 的逻辑单元称为 CLB（Configurable Logic Block），每个 CLB 包括两个 Slice，每个 Slice 包含查找表（LUT, Lookup Table）、寄存器、进位链和多个多数选择器。

通过向 Xilinx 7Serise FPGA 内的查找表写入数据，可以将组合逻辑写入 Slice

内的 SRAM 中，再通过 Slice 内的寄存器和触发器实现时序逻辑。通过将多个 Slice 连接在一起，可以实现更加复杂的逻辑功能。

通过向 Slice 内的 LUT 对应的 SRAM 中写入数据，可以实现任意的逻辑功能。FPGA 芯片的“编程”就是用这样的方式，将用户设计的数字电路映射到 FPGA 芯片的内部逻辑单元中，从而实现用户设计的功能。

FPGA 的开发流程一般包括以下几个步骤：

1. 硬件描述语言（HDL, Hardware Description Languaue）编写：使用 HDL如 Verilog、VHDL 等编写数字电路的设计。

2. 仿真：使用仿真工具对设计的数字电路进行仿真验证。

3. 综合：将 HDL 代码综合成逻辑网表。

4. 实现：将逻辑网表映射到 FPGA 芯片的内部逻辑单元中。

5. 烧写：将实现后的设计下载到 FPGA 芯片中。

HDL编写是数字电路设计的基础，数字电路中的逻辑功能通过 HDL代码来描述。常见的 HDL语言有 Verilog、SystemVerilog、VHDL 等。Verilog 在 20 世纪 80 年代由 Gateway Design Automation 公司开发，用于描述数字电路的结构和行为，并用于逻辑建模和仿真验证。SystemVerilog 是 Verilog 的扩展，提供了更多的特性和功能，如接口、类、多态等。VHDL 是 VHSIC Hardware Description Language 的缩写，是美国国防部为了解决数字电路设计中的问题而开发的一种硬件描述语言。

随着软件编程技术的发展，也出现了许多新的数字电路设计语言，如 Chisel、MyHDL、Bluespec Verilog 等。这些新的数字电路设计语言通常是基于 Python、Scala 等现代编程语言开发的，提供了更加灵活、简洁的设计方式。这些新型的语言大都基于高级语言并编译到 Verilog 或 VHDL，然后再通过传统的数字电路设计流程进行综合、实现。

仿真过程是对设计的数字电路进行验证的过程，通过仿真可以验证设计的数字电路是否符合预期的功能。常见的数字电路仿真工具有 Icarus Verilog、ModelSim、VCS、Verilator 等。常见的 HDL 仿真过程是使用软件功能在虚拟环境中对设计的数字电路进行仿真，并通过仿真结果来验证设计的正确性。具体原理方面，既包括 Icarus Verilog 这样将 HDL 直接解释运行的方式，也包括 VCS、Verilator、ModelSim 这样将 HDL 编译成仿真模型再运行的仿真方法。在仿真阶段方面，还包括行为仿真、综合后仿真、实现后仿真等仿真阶段，也可以分为行为仿真、寄存器传输级仿真、门级仿真等不同的仿真精度。此外，除了直接对 HDL 进行仿真，也可以使用 SystemVerilog、高级编程语言等对电路进行多精度层次的重新建模，以适应不同的仿真规模和精度需求。

综合过程是将 HDL 代码综合成逻辑网表的过程。逻辑网表是一种逻辑电路的抽象表示，其中包含了逻辑门、寄存器、时钟约束、元件连接等信息。综合过程是 HDL 代码到实际硬件的一次初步映射，在其过程中会对 HDL 代码进行分析、检查、优化等操作，然后生成逻辑网表文件，如图1-1。逻辑网表文件一般包括 EDIF、BLIF 等格式，这些格式是一种相对较为标准的逻辑网表描述格式，可以作为一种标准格式的电路描述被后续的实现工具读取。

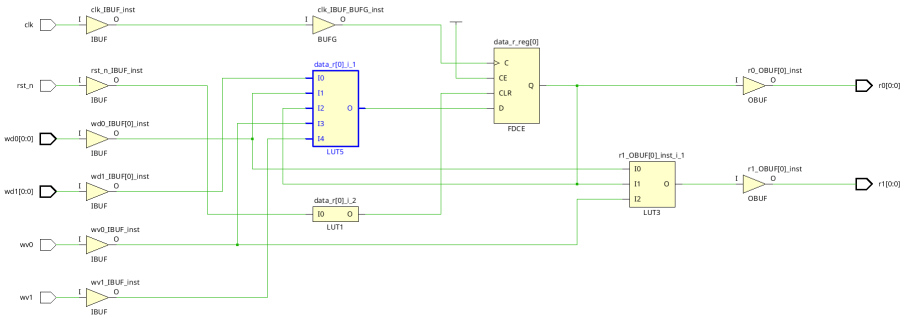


图 1-1　HDL 代码综合成的逻辑网表，以 Vivado 为例

实现过程是将逻辑网表映射到 FPGA 芯片的内部逻辑单元中的过程。FPGA 厂商的 EDA 工具将逻辑网表进行优化、布局布线、时序分析等操作，然后生成能够被 FPGA 芯片直接加载的比特流文件。实现完成后即可得到 FPGA 内部逻辑单元的分配与连接信息，如图1-2。

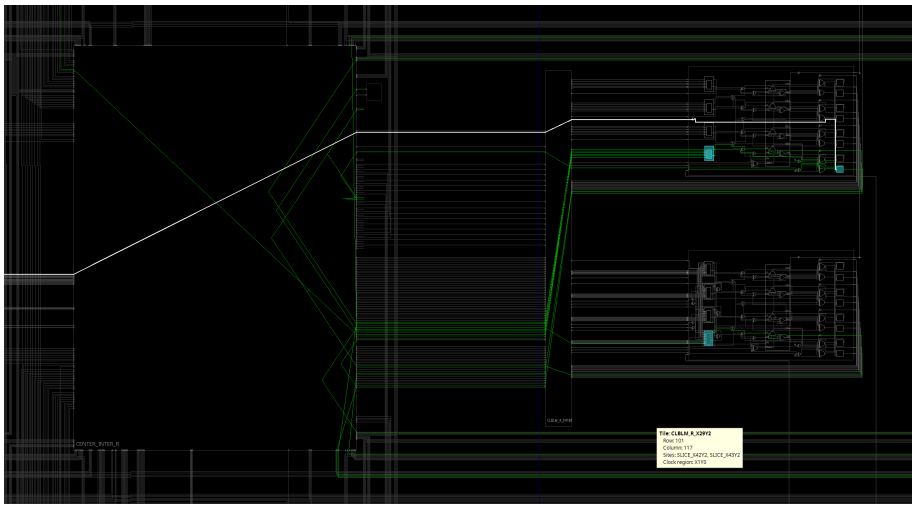


图 1-2　逻辑网表映射到 FPGA 芯片的内部逻辑单元中，以 Vivado 为例

烧写过程是将实现后的设计下载到 FPGA 芯片中的过程。FPGA 芯片常常包括一个 Jtag 接口，可以通过调试接口将比特流下载到 FPGA 芯片对应的存储器中，从而实现用户设计的功能。由于基于 SRAM 实现的 LUTs，每次下电都会丢失所有

数据，所以 FPGA 芯片上电时，会自动重新加载存储器内的数据到 SRAM 中，从而才能开始按照用户设计的功能工作。

以“实现”为界限，可以暂时将 FPGA 开发流程分为“前端”和“后端”。这里指的前端和 ASIC 设计中的前端概念基本一致，即从设计到综合的过程。后端则是指从综合到实现的过程。

FPGA 开发过程和 ASIC 设计过程的主要区别在于它们的后端，FPGA 设计中需要将网表映射到 FPGA 芯片内的已有的逻辑单元中，而 ASIC 设计则需要将逻辑网表映射到具体的晶体管中去。其区别可以理解为 FPGA 设计是在一个已经部分设计好的芯片上进行设计，而 ASIC 设计则是在一个空白的硅片上进行设计，两种设计都可以使用 IP 核等已经设计好的模块，只是在后端的物理实现上有所不同。

FPGA 的网表映射过程是由 FPGA 厂商提供的 EDA 工具完成的，这些工具通常包括综合工具、布局布线工具、时序分析工具等。这些工具通常是闭源的，且功能复杂，对于初学者来说可能会有一定的学习门槛。

1.4 FPGA 厂商 EDA 工具

FPGA 的 EDA 软件是与厂商深度绑定的，通常只能用于该厂商的 FPGA 芯片。常见的 FPGA 厂商包括 Xilinx、Altera（现 Intel）、Lattice 等，这些厂商都提供了自己的 EDA 工具，且提供以厂商 EDA 为核心的 FPGA 开发流程，甚至以 EDA 和 IP 为核心的 FPGA 生态系统。

1.4.1 Xilinx Vivado

Xilinx Vivado 是 Xilinx 公司推出的一款面向 Xilinx 系列 FPGA 的 EDA 工具。Vivado 提供了从设计到验证到实现的一整套解决方案，包括 Vivado Design Suite、Vivado HLS、Vivado SDK 等工具。目前 Vivado 系列已经被 Xilinx 公司融入更大的 Vitis 系列中，提供了更加全面的 FPGA 开发解决方案。

Vivado Design Suite 是 Vivado 的核心工具，提供了从设计到验证到实现的一整套解决方案。Vivado HLS 是一款高层次综合工具，可以将 C、C++、SystemC 等高级语言综合成硬件描述语言。Vivado SDK 是一款软件开发工具，用于 Zynq 等 FPGA 芯片上的软件开发。

本课题项目的主要用户主要面向学生和初学者，因此以下以 Vivado Design Suite 2019.2 为例，简要介绍 Vivado 的使用流程。

Vivado 的使用流程一般包括以下几个步骤：

1. 创建工程：在 Vivado 中创建一个新的工程。

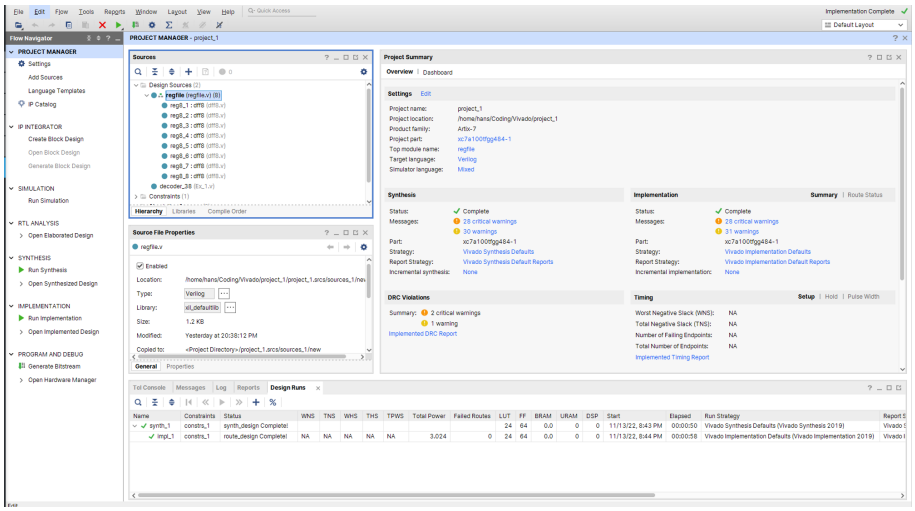


图 1-3　Xilinx Vivado 使用界面示例

2. 添加设计文件：将设计文件添加到工程中。
3. 综合：对设计文件进行综合。
4. 实现：对综合后的设计进行实现。
5. 生成比特流：生成可以下载到 FPGA 芯片的比特流文件。
6. 下载到 FPGA：将比特流文件下载到 FPGA 芯片中。

Vivado 本质是一个 Tcl 脚本驱动的工具，用户可以通过 Tcl 脚本来完成 Vivado 的所有操作。Vivado 的 GUI 是基于 Tcl 的，用户在 GUI 中的每一步操作都会生成对应的 Tcl 脚本，用户可以通过查看 Tcl 脚本来了解 Vivado 的操作流程。但是对于初学者而言，基本很少接触到 Tcl 脚本，大多还是通过 GUI 来完成设计。

窗口左侧是 Flow Navigator，用于导航设计流程，双击执行仿真、综合等操作。窗口右侧是主要的设计区域，用于显示设计的各个部分。窗口底部是消息区域，用于显示 Vivado 调用过程中产生的各种文本信息。

用户需要在 Vivado 自带的编辑器中编写 Verilog 等 HDL 代码，通过项目管理器将 HDL 文件加入项目结构中，然后双击 Flow Navigator 中的仿真、综合、实现等操作，即可完成 FPGA 设计的整个流程。

当仿真、综合等步骤出错时，Vivado 会在消息区域显示错误信息，用户可以根据错误信息来定位问题。Vivado 也提供了一些调试工具，如波形查看器、逻辑分析仪等，用户可以通过这些工具来查看设计的运行状态。

图 1-4　Xilinx Vivado 波形查看器示例

1.4.2 Vivado 在实践中的问题

Vivado 是一款功能强大的 EDA 工具，其专业性让其占有全球一半以上的 FPGA EDA 市场份额。但是 Vivado 也存在一些问题，如操作繁琐、功能复杂、学习曲线陡峭等，这些问题对于初学者来说会有一定的困难。

以 Vivado 2019.2 的安装过程为例，其安装包大小超过 26GiB，且安装后占用磁盘空间超过 40GiB，这对于一般的学生电脑来说是一个很大的负担。此外，Vivado 的安装过程也比较复杂，需要下载安装包、解决可能的安装问题、安装许可证等，不仅需要一定的网络条件，而且费时费力，遇到的安装问题也不少。不少同学光是在本地安装 Vivado 都无法解决，于是只能使用学校实验室内预装了 Vivado 的电脑，这对于学生的学习和实验都是一个很大的限制，甚至有同学因此激起了对学习计算机的抵触情绪，最后转到其他专业学习了。

同学们在使用 Vivado 进行 FPGA 设计时，还会遇到更多困难，如不熟悉 Vivado 项目结构、不清楚仿真流程、不明白各种报错类型的含义和解决办法等。当需要实际上板验证时，同学们遇到的数字电路设计相关的问题更多了，例如时序违例、组合逻辑依赖、生成锁存器、状态机控制等等问题，有很多甚至还是在仿真阶段都无法解决的。

Vivado 自身也存在一些问题，例如编辑代码时缺乏代码提示、缺少自动补全、没有代码格式化等功能，这对于初学者来说是一个很大的困难。

综上所述，Vivado 是一款功能强大的 FPGA EDA 工具，但是对于初学者来说，其学习曲线陡峭，操作繁琐，功能复杂，这对于初学者来说是一个很大的困难。解决这样的问题，需要一款更加简单易用的 FPGA 开发工具，以帮助初学者更好地学习 FPGA 的开发和编程，这也是本课题最初的研究目标之一。

1.4.3 Quartus Prime

Quartus Prime 是 Altera 公司推出的一款面向 Altera 系列 FPGA 的 EDA 工具。Quartus Prime 提供了从设计到验证到实现的一整套解决方案，包括 Quartus Prime、ModelSim、Qsys 等工具。Quartus Prime 是 Altera 公司的核心工具，提供了从设计到验证到实现的一整套解决方案。

Quartus Prime 的使用流程与 Vivado 类似，也是通过创建工程、添加设计文件、综合、实现、生成比特流、下载到 FPGA 的方式来完成 FPGA 设计的整个流程。不过，Quartus Prime 的可选择的任务操作相比 Vivado 要更多，如分区合并、IO 分配分析等。其主界面的窗口结构，与 Vivado 都是基本一致的。Quartus Prime 也是一个 Tcl 脚本驱动的工具，用户可以通过 Tcl 脚本来完成 Quartus Prime 的所有操作。

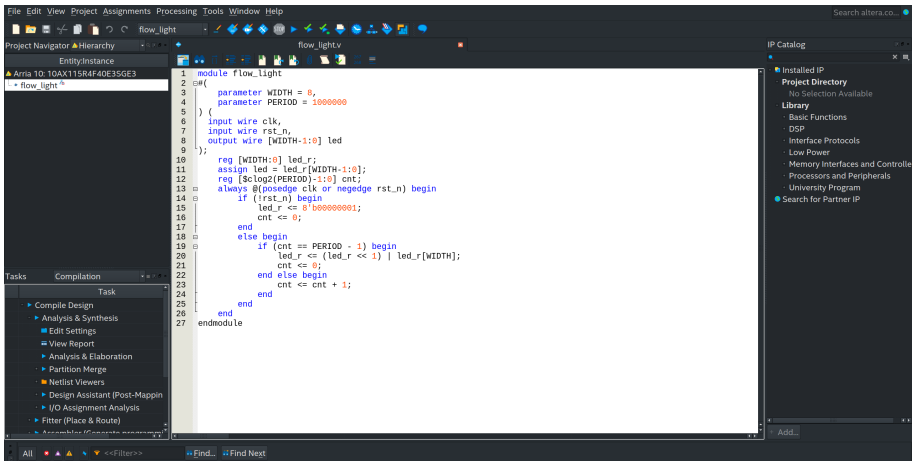


图 1-5　Altera Quartus Prime 使用界面示例

1.4.4 国产 FPGA 以及其 EDA 工具

目前，全球 FPGA 市场由 Xilinx 和 Altera 两家垄断，2021 年这两家厂商的市场份额之和超过 70%，剩下的份额由几家专注于低功耗低成本的厂商瓜分。从国内市场来看，中国 FPGA 市场起步较晚，市场规模较小，紫光同创、安路科技和复旦微电占据总共约 10% 份额。不过，随着国产芯片的发展，国产 FPGA 也在逐渐向行业市场渗透。[2]

随着国产 FPGA 厂商自研芯片产品不断推出，其 EDA 工具也在逐渐完善。目前，各家国产平台的 EDA 平台都已投入市场，如紫光同创的 Pango Design Suite（PDS）、安路科技的 TangDynasty、复旦微电的 Procise 等。

此处以紫光同创的 Pango Design Suite 为例，简要介绍国产 FPGA 产品以及 EDA 工具。

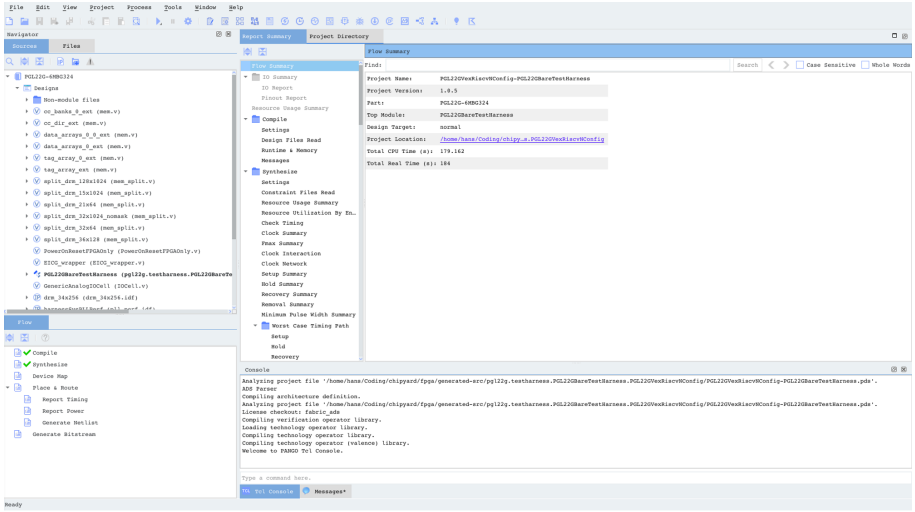


图 1-6　紫光同创 Pango Design Suite 使用界面示例

PDS 2022.2 安装包大小 2.2GiB 左右，安装后占用空间约 2.8GiB，其软件界面与 Vivado 基本一致，也是通过创建工程、添加设计文件、综合、实现、生成比特流、下载到 FPGA 的方式来完成 FPGA 设计的整个流程。但是，PDS 相对于 Vivado 而言，还有相对较大的差距。

从实际项目上举例子，PDS 2022.2 提供一个内置的 ADS 综合器，以及可以使用第三方的 Synopsys 综合器，并没有完全自主产权的综合器。这里尝试使用 PDS 打开使用本项目工具生成的 PDS 项目，其包含一个约 8MiB 的 Verilog 文件，以及一些约束文件。项目包括一个基于 Chipyard 生成的 RISC-V 核心，以及基础的 PLL 等外设模块。PDS 在预览和编辑文件时卡顿较多，且在使用 ADS 综合时遇到很多问题，包括生成的 LUT4s 远多于 Vivado 生成的 LUT6s，使用资源仅 70% 就无法完成综合布局布线等问题。

逻辑资源方面，国产 FPGA 单位逻辑单元的成本成本与 Xilinx、Altera 相比不相上下，但是在综合布局布线等方面还有很大的提升空间。而 IP 核方面，国产 FPGA 的 IP 核库还不够完善，缺少很多常用的 IP 核，这也是国产 FPGA 需要进一步发展的方向。

综合而言，这些 FPGA EDA 的共同特点有：

“一条龙”服务，软件臃肿。它们作为工业软件，提供了从设计到验证到实现的一整套解决方案，功能强大，但是也导致了软件体积庞大，操作繁琐，学习曲线陡峭等问题。

只能用于特定厂商的 FPGA，灵活性差。这些 EDA 工具都是与厂商深度绑定的，只能用于该厂商的 FPGA 芯片，不具有通用性。在 Verilog 等 HDL 层面或许是通用的，但是在综合、实现等后端流程上，以及可以使用的 IP 核上由于实际映射的物理结构不投，造成了这种不通用性。

编辑器体验不佳。作为工业软件，这些 EDA 工具的编辑器体验不够好，用户交互部分没有很好的打磨。缺少代码提示、缺少自动补全、没有代码格式化等功能，这对于初学者来说是一个很大的困难。很多厂商都意识到这个问题，且选择将这一部分功能交给第三方完成，如 Xilinx 和 PDS 都选择支持 Visual Studio Code 作为文本编辑器，但是并没有提供语法 LSP 等功能。

各个模块互相独立，且可以友好地从外部调用。这些 EDA 工具的各个模块基本都是经由 Tcl 脚本调用的独立的模块，这使得用户可以通过 Tcl 脚本来完成这些工具的所有操作。这也为进行第三方 EDA 开发提供了可能，为本文的 FPGA 开发工具提供了一定的操作入口。

1.5 开源 FPGA EDA 工具

一款 EDA 软件实质上是若干个相对独立的工具的集合。这些工具之间通过文件格式、命令行参数等方式进行交互，用户可以通过脚本等方式来调用这些工具，完成 FPGA 设计的整个流程。

因此，有一些第三方工具专注于某个环节，如综合、布局布线、时序分析等，这些工具通常是闭源的，但是也有一些开源的工具，如 Yosys[3]、Icarus Verilog[4]、Verilator[5] 等，这些工具可以用于 FPGA 设计的某个环节，也可以用于 FPGA 设计的整个流程[6]。

常见的第三方工具有：

1. 综合器：Synopsys Synplify（商业软件）、Yosys（开源软件）

2. 仿真器：ModelSim（商业软件）、Icarus Verilog（开源软件）、Verilator（开源软件）

3. 布局布线工具：Verilog to Routing[7]（开源软件）、F4PGA[8]（开源软件）

同样得益于 EDA 工具的模块化，打造第三方甚至全开源工具链的 FPGA EDA 是可能的。其基本设计思路可以是，基于 FPGA 开发的共性，设计一个通用的 FPGA EDA 工具链，然后通过插件的方式，将第三方工具集成到这个工具链中，从而实现 FPGA 设计的整个流程。

一些开源的可以用于 FPGA EDA 的工具：

1. Yosys：Yosys 是一款开源的综合工具，可以将 Verilog、SystemVerilog 等 HDL 代码综合成逻辑网表。Yosys 也提供了一些其他功能，如布局布线、时序分析等。

2. Icarus Verilog：Icarus Verilog 是一款开源的 Verilog 仿真工具，可以用于对设计的数字电路进行仿真验证，同时也提供语法解析、语法提示等其他功能。

3. Verilator：Verilator 是一款开源的 Verilog 仿真工具，可以将 Verilog 代码编译成 C++ 代码，然后通过 GCC 等 C/C++ 编译器编译成可执行文件，用于对设计的数字电路进行仿真验证。

4. Digital-IDE[9]：Digital-IDE 是 Visual Studio Code 平台的一款开源的数字电路设计辅助插件，提供了代码提示、代码补全、代码格式化等功能，可以用于辅助 FPGA 设计。从设计功能上，Digital-IDE 与本项目相似，不过设计目的上不完全相同。

1.5.1 开源 HDL 编辑器和 EDA 辅助工具

HDL 编辑器是 FPGA 设计的基础工具，用户通过 HDL 编辑器来编写数字电

路的设计。厂商 FPGA EDA 通常也会提供自己的 HDL 编辑器，如 Vivado、Quartus Prime、PDS 等都提供了自己的 HDL 编辑器，但是这些编辑器通常功能较弱，用户体验不佳。

因此，许多基于已有优秀编辑器的插件被开发出来，例如上文提到的 Digital-IDE。Digital-IDE 是基于 Visual Studio Code 平台的一款数字电路设计辅助插件，提供了代码提示、代码补全、代码格式化等功能，同时也能够与 Vivado、Yosys 等 EDA 工具进行交互。

如1-7所示，Digital-IDE 的 Verilog、SystemVerilog 等的语法支持是通过 Visual Studio Code 的 Json 格式的语法文件实现的。这种 Json 内部实现使用的是正则表达式语法，其编写相对简单，不过解析性能和语法复杂度都相对较低。

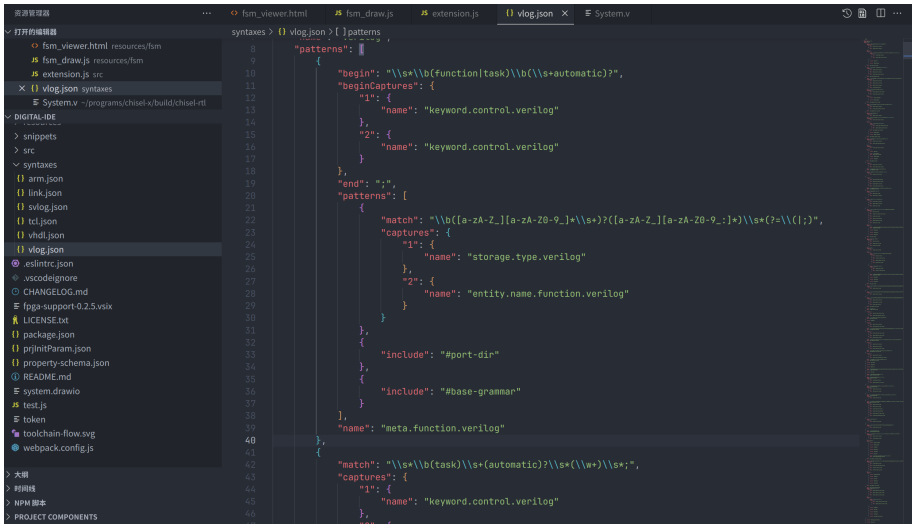


图 1-7　Digital-IDE 部分语言支持源码

在实际应用场景中，例如1-8所示，当在 Digital-IDE 中解析一个 1.6MiB 的 Verilog 文件时，不仅解析速度慢，而且最后解析出错，导致无法使用高亮以外的语法功能。而这种大小的 Verilog HDL 文件在 Chisel、SpinalHDL、Bluespec Verilog 等高级 HDL 的生成文件中是非常常见的，这也是 Digital-IDE 在实际应用中的一个问题。

在 HDL 编辑器功能之外，Digital-IDE 还提供了与 Vivado、Icarus Verilog 等 EDA 工具的交互功能。其主要使用方式是在配置文件中写入 EDA 工具的路径，然后在当前编辑的文件中能够通过快捷操作直接使用 Icarus Verilog 进行项目的仿真，或是在项目操作中设置顶层模块、启动 Vivado GUI，然后输入 Tcl 指令，在 Vivado 的 Project 模式下运行综合、实现等。其仅支持了 Vivado 的 Tcl 指令，对于其他 EDA 工具的支持较弱。

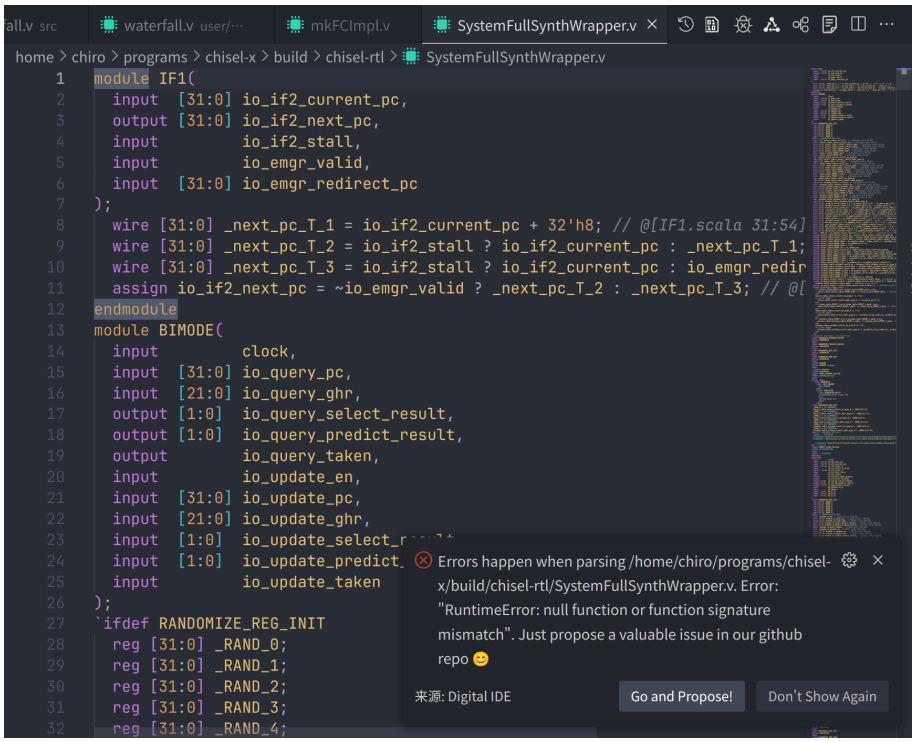


图 1-8　Digital-IDE 解析错误示例

除了 Digital-IDE，当前 Visual Studio Code 平台上下载量最大的是“Verilog-HDL/SystemVerilog/Bluespec SystemVerilog”[10] 这一插件，支持四种语言的高亮等语法功能，并且可以依靠外部工具插件实现其中一些语言的语法解析和跳转等功能。

观察这一插件的代码，其内部通过正则表达式的方式支持多种 HDL 语言的高亮，而代码提示、代码补全等功能则是通过调用外部工具实现的。这种方式的优点是简单易用，能够复用的代码功能更多，但是其可定义性和可扩展性较差，且对于大文件的解析性能较差。此插件的语法检查等使用较高频率的功能都不是开箱即用的，全部需要用户自行配置。且此插件专注于 HDL 语言的查看和编辑，并没有与 EDA 工具进行交互的功能。

1.5.2 开源 FPGA EDA 操作工具

这里指的开源 FPGA EDA 操作工具，指的是使用工具对厂商 EDA 进行各种自动化和个性化操作的工具。例如，Digital-IDE 也可以算是一个较为简单的 FPGA EDA 操作工具，其提供了一些简单的操作，如仿真、综合、实现等，但是其功能较为简单，且只支持 Vivado 的 Tcl 指令。

“edalize”[11] 是一个 Python 包，用于将 EDA 工具的调用封装成 Python API，从而可以通过 Python 脚本来调用 EDA 工具。edalize 支持了多种 EDA 工具，如 Vivado、

Quartus Prime、Yosys 等，用户可以通过 edalize 来调用这些 EDA 工具，完成 FPGA 设计的整个流程。

edalize 的一大特色是，它通过 EDA 工具链之间的统一接口，将各种 EDA 工具的调用封装成了抽象统一的 Python API，把 EDA 工具链的调用抽象为 HDL/网表文件和约束文件的输入，如此就能够在不同的 EDA 工具之间快速切换，而不需要修改代码。

其基本使用方式是，用户通过 Python 脚本来调用 edalize，然后通过 edalize 来调用 EDA 工具，完成 FPGA 设计的整个流程。用户可以通过 Python 脚本来控制 EDA 工具的调用，完成仿真、综合、实现等操作。

edalize 并没有提供 GUI 的操作方式，用户需要通过 Python 脚本来自己构建 FPGA 设计的流程，这对于初学者来说是一个很大的困难。但是 edalize 提供了一个很好的思路，即将不同的工具链的调用抽象为统一的接口，这对本项目的设计也提供了一定的参考。

“openFPGALoader”[12] 是一个开源的 FPGA 烧写工具，可以用于将比特流文件下载到 FPGA 芯片中。openFPGALoader 支持了多种 FPGA 芯片，如 Xilinx、Altera、Lattice 等，用户可以通过 openFPGALoader 来下载比特流文件到 FPGA 芯片中。

openFPGALoader 能够为本项目添加一个可用的独立的 FPGA 烧写工具，用户可以通过 openFPGALoader 来下载比特流文件到 FPGA 芯片中，从而完成 FPGA 设计流程。

1.5.3 开源 FPGA EDA 工具链和 SoC 构建工具

相比于闭源的厂商 FPGA EDA 工具和生态，开源生态虽然有无法获取闭源、含有专利的 IP 核的问题，但是其开放性、可定制性、可扩展性等优势也是不可忽视的。开源 FPGA EDA 工具链目前正在逐渐发展当中。

F4PGA[8] 来自 CHIPS(Common Hardware for Interfaces, Processors and Systems) Alliance，是一个开源的 FPGA EDA 工具链，自称为 “the GCC of FPGAs”，其目标是打造一个完全开源的 FPGA EDA 工具链，包括综合、布局布线、时序分析等。F4PGA 通过综合使用各种开源工具，完成整个 FPGA 设计流程。

其使用 Yosys ABC 作为综合工具，并使用 nextpnr 和 Verilog to Routing 作为布局布线工具[13]，使用 Project X-Ray 等工具作为 FPGA EDA 内部闭源工具的等效替代。

F4PGA 通过 Conda 安装和管理多种依赖，使用 XLite 等工具完成 SoC 系统构建，再使用它的开源后端工具链完成 FPGA 设计中的综合、布局布线、烧录等操作。目前 F4PGA 主要支持了 Xilinx Artix-7 系列 FPGA。

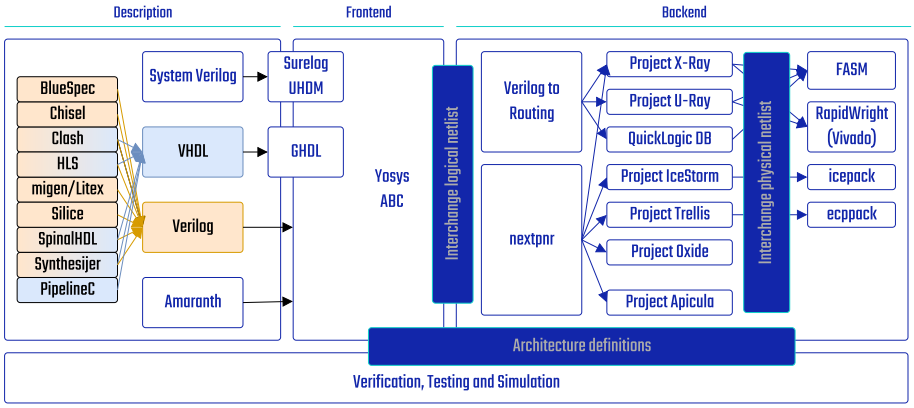


图 1-9　F4PGA FPGA EDA 工具链概览

F4PGA 作为全开源的 FPGA EDA 后端工具，能够为本项目提供一个完全开源的 FPGA EDA 工具链，用户可以通过 F4PGA 来完成 FPGA 设计的整个流程。不过 F4PGA 目前仍在开发过程中，并不面向初学者用户，所以本项目并不直接使用 F4PGA，仅将其作为可用的工具链插件使用。

1.6 本文的主要研究内容

本课题项目的主要研究内容是，通过对 FPGA EDA 各种工具的研究，设计并实现一款 FPGA EDA 工具链的辅助工具，用于辅助 FPGA 设计的各个环节，如代码编辑、仿真、综合、实现等，以辅助实现 FPGA 设计的快速迭代和验证，并主要面向学生和初学者。

具体实现层面，本项目主要包括两个部分：作为 IntelliJ IDEA 插件存在的 Scaleda，和作为相对独立的波形查看器和辅助工具存在的 Rvcd。

Scaleda 主要解决 FPGA EDA 工具链的代码编辑问题，提供了代码提示、代码补全、代码格式化等功能，同时也提供了与 Vivado、Icarus Verilog 等多种 EDA 工具的交互功能，用户可以通过 Scaleda 来完成 FPGA 设计的代码编辑、仿真、综合、实现等操作，实现 FPGA 设计全流程操作。

Rvcd 主要解决 FPGA EDA 工具链的波形查看问题，提供了波形查看、波形分析和辅助代码解析、跳转等功能，用户可以通过 Rvcd 来查看、分析波形数据，辅助 FPGA 设计的调试和验证。

Scaleda 目前代码总行数，排除生成的代码和第三方库代码，约 3.3 万行；Rvcd 目前代码总行数，排除生成的代码和第三方库代码，约 1.0 万行。

第 2 章 Scaleda 软件设计与开发

本章主要介绍 Scaleda 软件的设计思路、设计细节和开发过程。

2.1 概述

本项目 Scaleda 是一款用于 FPGA 开发、数字电路前端开发等功能的集成开发环境。本项目主体为一个 IntelliJ IDEA 平台的插件，致力于提供用户友好的开发平台，支持智能化的 Verilog 等 HDL 代码编写及便捷的仿真调试体验，并可接入 Vivado 等厂商工具链实现逻辑综合、实现等功能。

Scaleda 其名称来源于“Scalable EDA”，意为“可扩展的 EDA 工具”。其设计初衷是提供一个灵活、通用的数字电路设计学习工具，降低学习门槛，提高学习效率。现阶段，Scaleda 能够支持 Verilog、SystemVerilog、Bluespec Verilog 和 Tcl 等语言的代码编辑、代码解析、代码提示、代码检查等功能，支持通过 Vivado、Icarus Verilog、Yosys 等工具链的调用，实现 FPGA 设计中的仿真、综合、实现等功能。

本项目设计时充分考虑了可扩展性，通过添加部分代码和配置内容，可以轻松地对 Scaleda 进行功能扩展，支持更多的语言、更多的工具链、更多的代码提示等功能。

图 2-1　Scaleda 主图标

Scaleda 插件部分的主要设计语言是 Scala。Scala 是一种多范式编程语言，设计初衷是要集成面向对象编程和函数式编程的各种特性。Scala 代码可以编译为 Java 字节码，因此可以运行在 JVM 上。Scala 语言的特性使得 Scaleda 插件的代码更加简洁、易读、易维护。

Scaleda 将 Scala 选择作为主要开发语言的原因有以下几点：

- Scala 语言的多范式、函数式等特性让项目开发更加灵活，切合本项目的灵活性需求。
- Scala 语言的运行环境是 JVM，因此可以利用 Java 生态圈的丰富资源，并且可以顺利地与 Java 代码进行交互，作为 IntelliJ IDEA 插件的开发语言，Scala 语言非常适合。
- Chisel 和 Spinal HDL 都是近期发展迅速的高级 HDLs，其二者本身都是 Scala 的一种领域特定语言，因此 Scala 语言在数字电路设计领域有着很好的适应性。

代码模块化设计方面，Scaleda 采用分离式设计，将用户交互和核心处理结构分离，将 EDA 执行、语法解析等功能划入 scaleda-kernel 模块，将用户交互、UI 设计等功能归入 scaleda 模块。Scaleda Kernel 能够编译为独立的 jar 包并拥有独立的命令行接口，并且能够作为服务器运行，接受来自 Scaleda 的网络 RPC 请求。

功能模块设计方面，Scaleda 主要包括以下几个部分：

- 代码解析和显示：通过 ANTLR4 生成的 HDL 语法解析器，解析用户输入的 HDL 代码，生成并绑定抽象语法树 AST 和 PSI 树。通过生成的 PSI 节点，支持 HDL 语言的高亮、语法错误提示等功能。
- 代码编辑：通过解析 PSI 节点，支持用户对 HDL 代码的自动提示、动态检查、自动补全、动态排错、代码生成等功能。
- 项目管理：支持用户创建、打开、保存项目，管理项目中的文件。
- IP 核管理：支持用户创建、导入、导出 IP 核，管理 IP 核的版本、依赖等。
- EDA 执行：支持用户对项目进行仿真、综合等操作。
- Vivado 集成模式：支持用户在 Vivado 集成模式下使用本地 Vivado 直接进行多种设计操作。
- 项目和全局设置：支持用户设置工具链、调用参数、仿真参数等。
- 命令行接口：支持用户通过命令行接口调用 Scaleda Kernel 的功能。
- 远程服务：用户和鉴权系统，并支持用户将 EDA 执行任务提交到远程服务器进行执行。
- 波形交互：与 Rvcd 联动支持用户查看仿真波形、波形文件的导入导出。

本章节将以以上功能模块为顺序，逐步介绍 Scaleda 的设计和开发过程。

2.2 代码解析和显示

代码解析是 Scaleda 的核心功能之一，代码解析模块主要负责将用户输入的 HDL 代码解析为抽象语法树 AST 和 PSI 树，以便后续的代码显示、代码编辑、EDA 执行等功能的实现。

Scaleda 中的语言功能支持是基于 ANTLR4[14] 的。ANTLR4 是一个强大的语法解析器生成器，它可以根据用户定义的语法规则生成语法解析器。在 Scaleda 中使用 ANTLR4 生成了 HDL 语言的语法解析器，以支持 HDL 代码的解析。

```
parser grammar VerilogParser;

options
  { tokenVocab = VerilogLexer; }
```

```
// START SYMBOL
source_text
   : (directive | description)* EOF
   ;

directive
   : '`' (timescale_directive
   | include_directive
   | default_nettype_directive
   | define_directive
   | ifdef_directive
   | ifndef_directive
   | else_directive
   | elsif_directive
   | endif_directive
   | undef_directive)
   ;

timescale_directive
   : 'timescale' Time_identifier '/' Time_identifier
   ;

defined_flag: Simple_identifier ;

create_defined_flag: defined_flag ;
create_defined_term: (term)? ;
```

Scaleda 中当前已经支持了 4 种语言的解析，分别是 Verilog、SystemVerilog、BlueSpec Verilog 和 Tcl，其语法规则文件位于 `scaleda-kernel/src/main/antlr` 目录下。

Verilog 和 SystemVerilog 的语法规则文件是从多个开源项目中整合而来的，包括 IEEE 官方的 Verilog 和 SystemVerilog 语法规范[15]、Verilog-HDL/SystemVerilog/Bluespec SystemVerilog 插件[10]、Bluespec Verilog 语言官方参考文档[16] 等。Tcl 的语法规则相对简单，主要是对 Tcl 语言的基本调用语法进行了支持。

使用 IntelliJ IDEA 的 ANTLR4 插件可以运行 antlr4 程序，将上述语法文件转换为 Java 代码，生成的 Java 代码位于 `top.scaleda.*.parser` 下，而将其与 PSI 绑定、进行语法高亮、语法检查等操作的 Scala 代码位于 `top.scaleda.*` 下。

为了在 IntelliJ IDEA 内支持新的 HDL 语言，需要在 `plugin.xml` 中注册新的语言、扩展名声明、解析器类、结构视图类等。以 Bluespec Verilog 为例，需要在 `plugin.xml` 中添加如下配置：

```
<!-- Bluespec Support -->
<extensions defaultExtensionNs="com.intellij">
    <fileType
    name="Bluespec"
    implementationClass="top.scaleda.bluespec.BluespecFileType"
    language="Bluespec"
    extensions="bsv"/>
    <lang.parserDefinition language="Bluespec"
            implementationClass="top.scaleda.bluespec.
                BluespecParserDefinition"/>
    <lang.ast.factory language="Bluespec" implementationClass="top.
       scaleda.bluespec.BluespecASTFactory"/>
    <lang.syntaxHighlighterFactory language="Bluespec"
            implementationClass="top.scaleda.bluespec.highlight.
                BluespecSyntaxHighlighterFactory"/>
    <lang.psiStructureViewFactory language="Bluespec"
            implementationClass="top.scaleda.bluespec.structview.
                BluespecStructViewFactory"/>
    <!-- same as Verilog -->
    <lang.commenter language="Bluespec"
            implementationClass="top.scaleda.bluespec.commenter.
                BluespecCommenter"/>
</extensions>
```

在 BluespecParserDefinition 中需要引用新的 Language 对象、导入 ANTLR4 的词汇表、规则表，创建 ANTLR 到 IntelliJ IDEA PSI 的 Lexer 和 Parser 的适配器等。

在 BluespecASTFactory 中需要实现从文本到 LeafPsiElement 的转换，调用 ANTLR4 相关功能等。

BluespecSyntaxHighlighterFactory 中需要根据 Lexer 得到的 Token，以及 AST 中的节点类型，为 PSI 节点设置颜色、样式等。

还有许多其他的功能支持，都是通过这样的方式实现的。

IntelliJ IDEA 的 PSI（Program Struct Interface）是一个抽象语法树，是 IntelliJ IDEA 用于表示代码结构的一种数据结构。在 IntelliJ IDEA 中的语法元素都会映射

到 PsiElement 的子类，这些子类的实例互相联结构成了一个抽象语法树，所有的语言功能都是基于这个抽象语法树实现的。相比于 Visual Studio Code 的 TextMate 格式的标记语法，IntelliJ IDEA 的 PSI 是一种更加统一、更加丰富、更加强大的语法树表示方式，支持代码提示、代码补全、代码检查、代码生成等功能的时候相对也非常方便。PSI 树通过注册的语法解析器生成，并且可以通过注册的语法高亮器、代码生成器、代码检查器等功能进行扩展。而使用 ANTLR4 生成的语法解析器，其生成的是一个统一的 LL(k) 语法解析器，可以很方便地与 IntelliJ IDEA 的 PSI 树进行绑定。

当完成了对一种语言的语法解析的支持，即可在 IntelliJ IDEA 的 PSI 预览窗口中查找到该语言的语法树，以及在代码编辑器中支持该语言的高亮、基于词法的提示等功能。

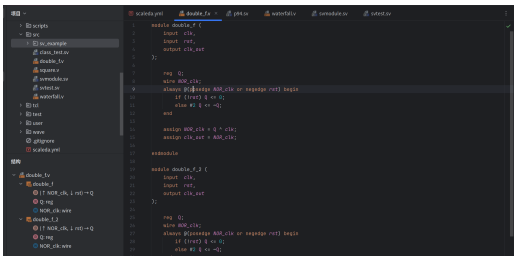
本项目在实现这些语言的支持的过程中，遇到了许多问题。例如，某些 ANTLR4 语法解析过程是无法直接参与 IntelliJ IDEA 的解析过程的，需要对一些结构进行重写，以适应 IntelliJ IDEA 的 PSI 解析过程；又例如，许多开源代码中的语法规则是针对代码解析设计的，而不是针对代码编辑设计的，当用户编辑代码时会出现大面积的语法错误提示，这就需要对语法规则进行适当的调整，以适应用户的编辑操作。最终，本项目解决了这些问题，实现了对 Verilog、SystemVerilog、Bluespec Verilog 和 Tcl 语言的支持，用户可以在 IntelliJ IDEA 中使用 Scaleda 插件对这些语言的代码进行高亮、提示、检查等操作。

如图 2-2 是 Scaleda 对 Verilog、SystemVerilog、Bluespec Verilog 和 Tcl 语言的语法解析支持的效果展示。

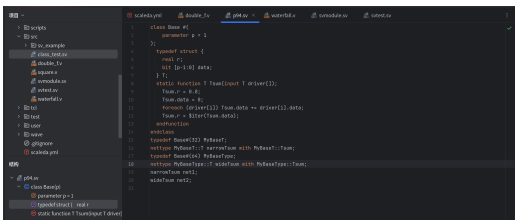
由 PSI 和 ANTLR4 实现的代码解析，相比于使用正则表达式等方式实现的代码解析，具有更好的性能与一致性。在 Scaleda 中，同样打开一个大型的 1.6MiB 大小的 Verilog 文件，能做到几秒钟内对文件进行分析和高亮，并快速地响应用户的操作，如图 2-3 所示；而作为对比，Digital-IDE 的语法解析在打开相同的文件时会出错导致无法正常显示，如图 1-8。

2.3 代码编辑

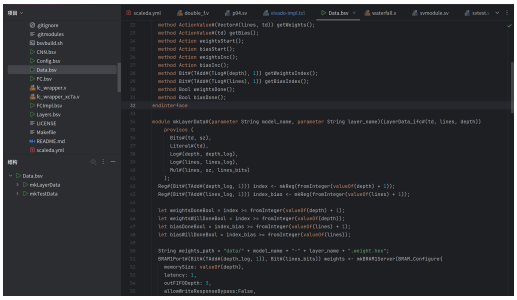
代码编辑是 Scaleda 的另一个核心功能，代码编辑模块主要负责对用户输入的 HDL 代码进行自动提示、动态检查、自动补全、动态排错、代码生成等操作。由于本部分需要的工作量较大，本项目目前只实现了 Verilog 语言的代码编辑功能，其他语言的的编辑辅助功能将在后续版本中继续逐步实现。



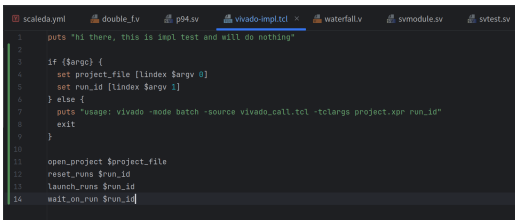
(a) 语法解析：Verilog



(b) 语法解析：SystemVerilog



(c) 语法解析：Bluespec Verilog



(d) 语法解析：Tcl script

图 2-2　支持语言的语法解析效果

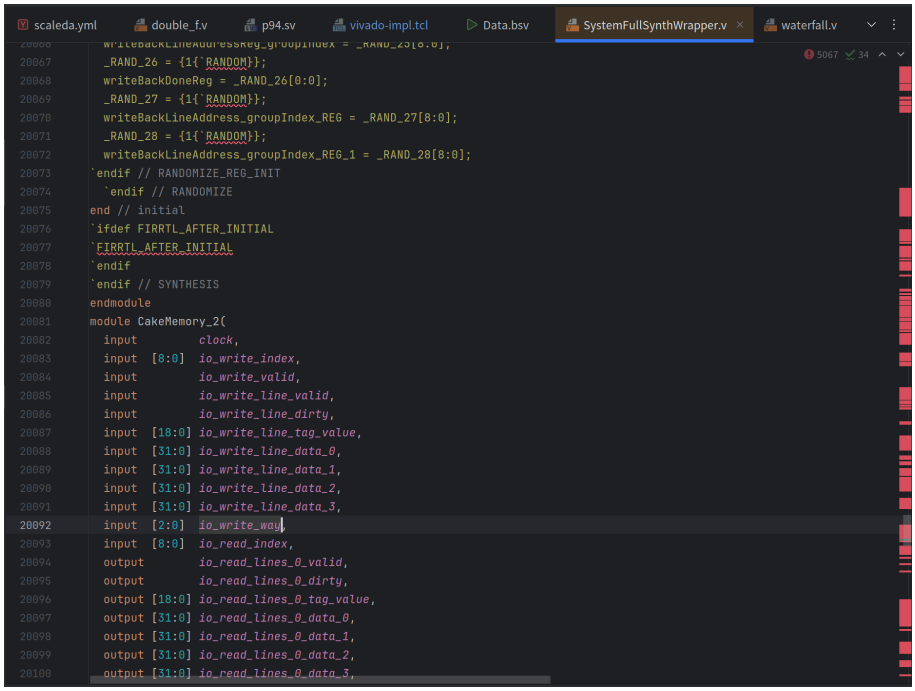


图 2-3　Scaleda 对大型 Verilog 文件的支持效果展示。报错为暂不支持的非标准扩展。

2.3.1 语义级代码补全

当 IntelliJ IDEA 检测到当前 PSI 位置需要进行语法元素补全时，会调用 `plugin.xml` 内注册的 `CompletionContributor` 来对当前语言环境进行补全。在

Scaleda 中，主要实现了以下的补全功能：

```
    <completion.contributor language="Verilog"
    implementationClass="top.scaleda.verilog.completion.signal.
        SignalNameCompletionContributor"/>
<completion.contributor language="Verilog"
    implementationClass="top.scaleda.verilog.completion.keywords.
        VerilogKeywordCompletionContributor"/>
<completion.contributor language="Verilog"
    implementationClass="top.scaleda.verilog.completion.module.
        ModuleItemOuterReferenceCompletionContributor"/>
```

`SignalNameCompletionContributor` 主要负责对信号名进行补全，当用户在模块中、模块例化中、assign 语句中等位置输入信号名时，会自动提示当前模块中的信号名，以及搜索涉及到的其他模块的相关匹配的信号名进行补全。

`VerilogKeywordCompletionContributor` 主要负责对关键字进行补全，当用户输入关键字时，会自动提示当前语言环境中的关键字。此处是根据语言上下文环境不同进行的补全，而不是简单的关键字列表。

`ModuleItemOuterReferenceCompletionContributor` 主要负责对模块例化中的模块名进行补全，当用户输入模块名时，会自动提示当前工程中的模块名，以及搜索涉及到的其他工程的相关匹配的模块进行补全。此补全模块将会按照项目配置，以及当前选定的运行配置（RunConfiguration），在对应路径搜索 HDL 文件并解析出对应需求的模块。

通过本项目的实现，用户可以在使用 Scaleda 时，能够有相比于其他编辑器插件更加智能的代码补全功能，并且开箱即用，提高了用户的编码效率。

2.3.2 语义级代码检查

IntelliJ IDEA 的代码检查功能是通过注册 `Annotator` 来实现的。在 Scaleda 中实现了许多的代码检查功能：

```
<annotator language="Verilog"
    implementationClass="top.scaleda.verilog.annotation.instantiation
        .ModuleInstantiationAnnotator"/>
<annotator language="Verilog"
    implementationClass="top.scaleda.verilog.annotation.instantiation
        .ModulePortConnectionAnnotator"/>
```

```
<annotator language="Verilog"
    implementationClass="top.scaleda.verilog.annotation.assign.
       MultiDrivenAnnotator"/>
<annotator language="Verilog"
    implementationClass="top.scaleda.verilog.annotation.always.
       AmbiguousClockAnnotator"/>
<annotator language="Verilog"
    implementationClass="top.scaleda.verilog.annotation.always.
       AmbiguousAssignInAlways"/>
<annotator language="Verilog"
    implementationClass="top.scaleda.verilog.annotation.always.
       AssignmentsMixedUse"/>
<annotator language="Verilog"
    implementationClass="top.scaleda.verilog.annotation.misc.
       HierarchicalReferenceAnnotator"/>
<annotator language="Verilog"
    implementationClass="top.scaleda.verilog.annotation.misc.
       IncompleteSeqBlockAnnotator"/>
<annotator language="Verilog"
    implementationClass="top.scaleda.verilog.annotation.misc.
       UnmatchedVerilog1995PortAnnotator" />
<annotator language="Verilog"
    implementationClass="top.scaleda.verilog.annotation.misc.
       IncompleteConditionAnnotator"/>
<annotator language="Verilog"
    implementationClass="top.scaleda.verilog.annotation.assign.
       OrOperatorInSensitiveList"/>
<annotator language="Verilog"
    implementationClass="top.scaleda.verilog.annotation.always.
       ConditionalStatementLatchAnnotator"/>
<annotator language="Verilog"
    implementationClass="top.scaleda.verilog.annotation.always.
       CaseStatementLatchAnnotator"/>
```

```
<annotator language="Verilog"
    implementationClass="top.scaleda.verilog.annotation.misc.
       WidthConcatAmbiguousAnnotator"/>
<annotator language="Verilog"
    implementationClass="top.scaleda.verilog.annotation.instantiation
       .ModuleParameterAssignmentsAnnotator" />
<annotator language="Verilog"
    implementationClass="top.scaleda.verilog.annotation.instantiation
       .NamedParameterAssignmentAnnotator" />
<annotator language="Verilog"
    implementationClass="top.scaleda.verilog.annotation.instantiation
       .UnresolvedPortConnectionAnnotator" />
<annotator language="Verilog"
    implementationClass="top.scaleda.verilog.annotation.assign.
       IllegalAssignAnnotator" />
<annotator language="Verilog"
    implementationClass="top.scaleda.verilog.annotation.instantiation
       .EmptyNamedPortConnectionAnnotator" />
```

1. ModuleInstantiationAnnotator 主要负责对模块例化进行检查，当用户在模块例化时，会检查模块名是否存在、参数是否匹配等。

2. ModulePortConnectionAnnotator 主要负责对模块端口连接进行检查，当用户在模块例化时，会检查端口连接是否正确。

3. MultiDrivenAnnotator 主要负责对多驱动信号进行检查，当用户在 assign 语句中，会检查是否有多个驱动信号。

4. AmbiguousClockAnnotator 主要负责对时钟信号进行检查，当用户在 always 语句中，会检查是否有多个时钟信号等问题。

5. AmbiguousAssignInAlways 主要负责对 always 语句中的赋值语句进行检查，当用户在 always 语句中，会检查是否有驱动不明确的赋值语句。

6. AssignmentsMixedUse 主要负责对 always 语句中的赋值语句进行检查，当用户在 always 语句中，会检查是否有多个赋值语句。

7. HierarchicalReferenceAnnotator 主要负责对模块引用进行检查，当用户在模块例化时，会检查模块名是否存在。

8. `IncompleteSeqBlockAnnotator` 主要负责对 always 语句中的 seq block 进行检查，当用户在 always 语句中，会检查 seq block 是否完整。

9. `UnmatchedVerilog1995PortAnnotator` 主要负责对 Verilog-1995 语法的端口进行检查，当用户在模块例化时，会检查端口是否匹配。

10. `IncompleteConditionAnnotator` 主要负责对条件语句进行检查，当用户在编写条件语句时有可能留下不完整的条件。

11. `OrOperatorInSensitiveList` 主要负责对敏感列表进行检查，用户有可能在敏感列表中误用 $a||b$ 这样的表达式。

12. `ConditionalStatementLatchAnnotator` 主要负责对条件语句进行检查，用户有可能在条件语句没有完整驱动从而错误地留下锁存器。

13. `CaseStatementLatchAnnotator` 主要负责对 case 语句进行检查，用户有可能在 case 语句没有完整驱动从而错误地留下锁存器。

14. `WidthConcatAmbiguousAnnotator` 主要负责对宽度拼接进行检查，用户有可能在宽度拼接时留下歧义。

15. `ModuleParameterAssignmentsAnnotator` 主要负责对模块参数进行检查，用户有可能在模块例化时没有正确地赋值参数。

16. `NamedParameterAssignmentAnnotator` 主要负责对命名参数进行检查，用户有可能在模块例化时没有正确地赋值命名参数。

17. `UnresolvedPortConnectionAnnotator` 主要负责对端口连接进行检查，用户有可能在模块例化时没有正确地连接端口。

18. `IllegalAssignAnnotator` 主要负责对 assign 语句进行检查，用户有可能在 assign 语句中使用了不合法的赋值。

19. `EmptyNamedPortConnectionAnnotator` 主要负责对命名端口连接进行检查，用户有可能在模块例化时没有正确地连接命名端口。

上述规则检查的实现逻辑是通过语法文件内的规则捕获，以及通过 PSI 树的外部代码处理实现的。这些语法规则提示都是初学者在编写 HDL 代码时常见的错误，通过这些检查，可以帮助用户更好地编写 HDL 代码。

2.3.3 Verilog 语言支持

前文已经介绍了 Scaleda 对其支持语言的语法解析和代码检查等功能的实现，本小节在表格 2-1 和 2-2 中展示了 Scaleda 对 Verilog 语言的支持细节。

Scaleda 对 Verilog 额外提供了自定义配色方案功能，用户可以在设置中选择自己喜欢的配色方案，以适应不同的视觉需求，如图 2-4 所示。

表 2-1　对 Verilog 语言的语法检查和电路驱动检查

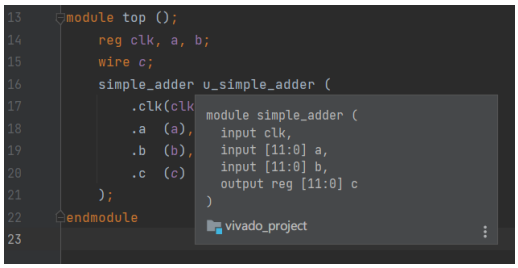
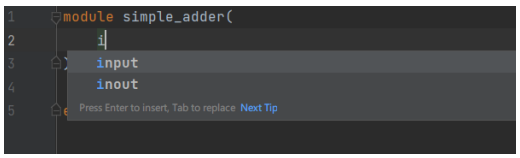
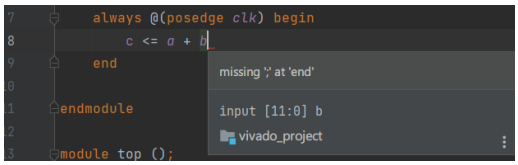
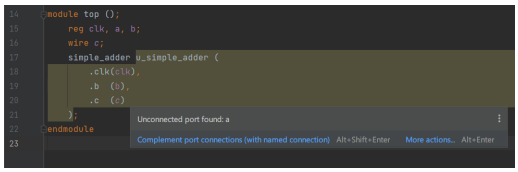
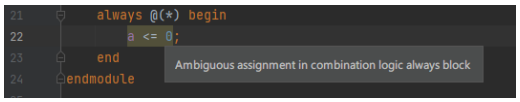
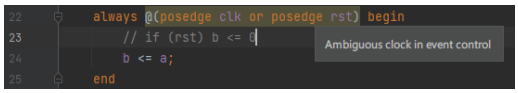
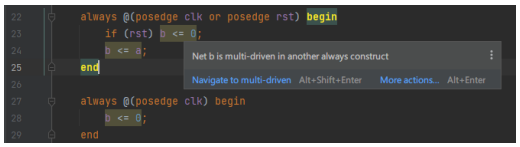
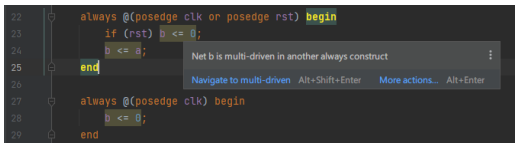
<table>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>线网信号信息：代码中所有的信号会根据它们的类型使用不同的字体表示。例如上图中所有紫色斜体表示“线网”类型，而紫色正体表示“寄存器 (reg)”类型。将鼠标移动到某个信号上，浮窗会展示此信号的定义，同时也可以按住 Ctrl 点击以跳转到那个信号的定义处。</td>
<td>模块实例信息：与信号信息类似，电路结构中被实例化的模块也有信息显示、引用跳转等功能。 使用鼠标悬停在模块实例上，会显示模块实例的定义以及端口声明情况等。使用跳转功能可以快速定位到模块定义处。</td>
</tr>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>语义级语法元素补全：自动在当前语境下提示可能用到的语法元素。例如，当用户在模块实例化时输入模块名时，会自动提示当前工程中的模块名，以及搜索涉及到的其他工程的相关匹配的模块进行补全；当用户在写模块定义时，自动显示可用的端口声明关键字等。</td>
<td>语法元素缺失提示：当用户输入的代码中有语法元素缺失时，会自动提示用户缺失的语法元素以供参考，如缺失模块名称、缺失分号等。</td>
</tr>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>驱动信号检查：对常见的错误驱动进行检查，如驱动方向错误，尝试驱动 <code>input</code> 类型的端口，尝试在 <code>always</code> 语句中驱动 <code>wire</code> 类型的线网，尝试用 <code>assign</code> 语句驱动 <code>reg</code> 类型的寄存器等。</td>
<td>模块实例检查：对模块实例化进行检查，如模块名不存在，模块参数不匹配，模块端口连接不匹配等。当模块提供的端口未完全连接，将进行警告。通过此功能可以快速进行大型模块的实例化。</td>
</tr>
</table>

表 2-2　对 Verilog 语言的 always 语句检查和多驱动检查

	
always 语句赋值混用检查：一般来说，对于边沿敏感的 always 块，其建议使用非阻塞赋值即 <=；而对于电平敏感的 always 块，则建议使用阻塞赋值即 =。如果在代码中混用，则会出现提示。	always 语句敏感信号模糊检查：对于电平敏感的 always 块，其敏感信号若未在块内被显式使用，则会被作为时钟处理，这可能在综合时产生多时钟问题。因此，若 always 敏感信号有未被使用的，则会出现提示。
	
always 语句多驱动检查：如果一个信号在多个 always 块中被驱动，则有很大概率会在综合时产生问题（多驱动问题）。Scaleda 会标记出这样的信号，并可在多处驱动之间来回跳转。	潜在的锁存器检测：对于组合逻辑的 if 或 case 语句，若没有形成完整的逻辑通路（即 if 总是与 else 配对、case 总是有 default），则会在综合中产生锁存器，这是不好的设计。Scaleda 会对这一问题进行标注。

此外，Scaleda 还通过内置的 Verible 工具对 Verilog 代码提供了格式化操作，并且能够通过图形化界面对格式化的参数进行调整，如图 2-5 所示。

2.4 项目管理

类似于 Digital-IDE、F4PGA 等 FPGA 开发工具，Scaleda 也实现了一套自己的项目格式系统，用于管理用户的项目文件路径、操作执行参数等。

2.4.1 项目文件格式

Scaleda 的项目文件是当前项目目录下的 scaleda.yml 文件，该文件是一个 YAML 格式的文件，用于存储项目的配置信息。在该文件中，用户可以配置项目的名称、工具链、仿真参数、综合参数等。此文件经过 Jackson 库，映射到 Scala Case Class 中，以便于 Scala 代码的读取和修改。相比于 Digital-IDE、F4PGA 等 FPGA 开发工具的使用 Json 描述的项目文件，Scaleda 的项目文件更加简洁、易读、易写。

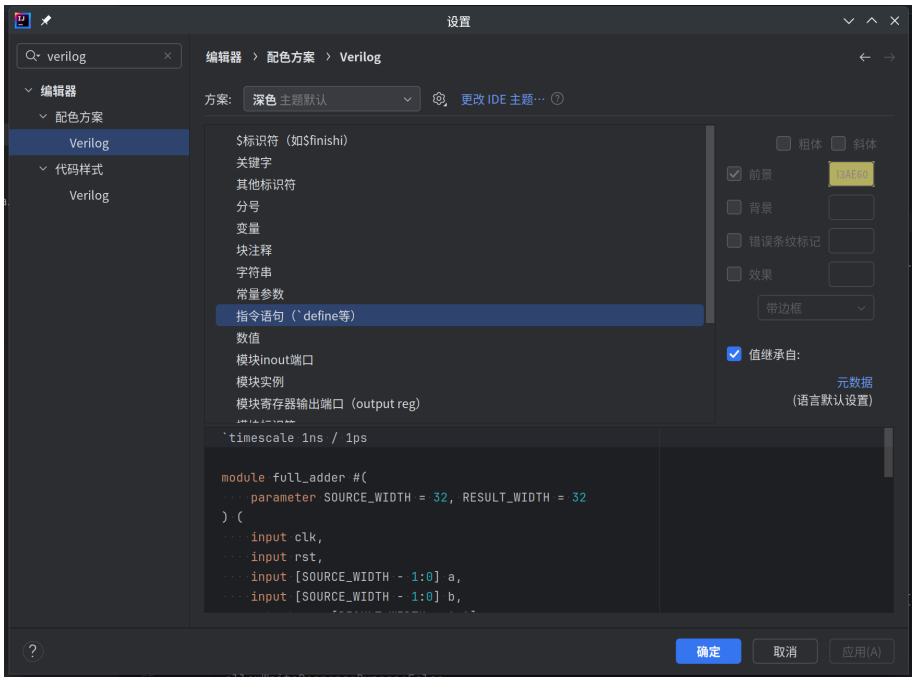


图 2-4　Verilog 语言配色方案支持

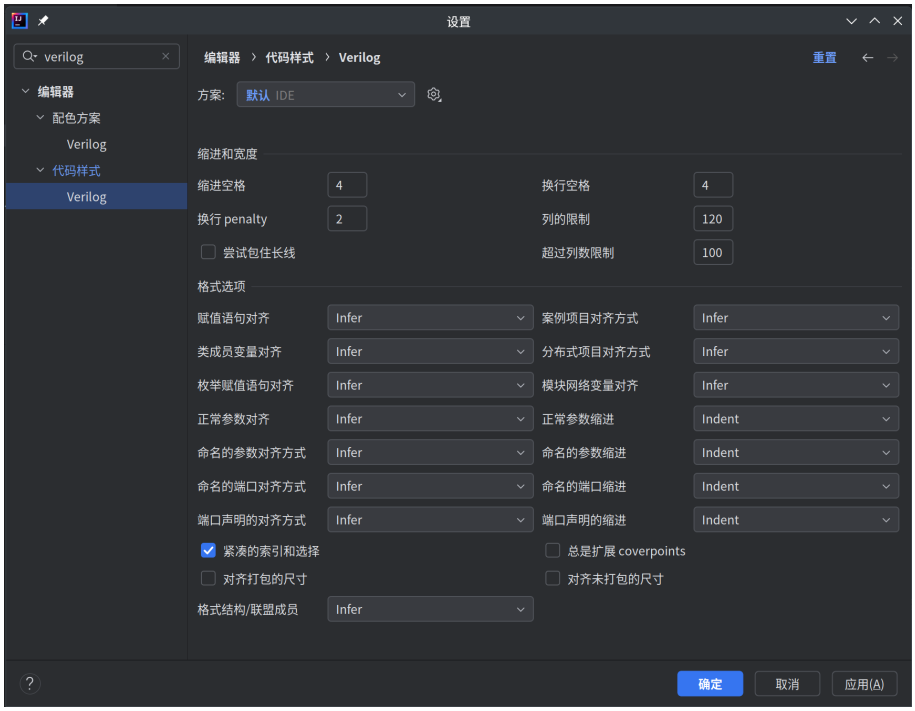


图 2-5　Verilog 代码格式化支持

一个简单的配置文件示例如下：

```
---
name: scaleda-sample-project
version: 0.1
```

```
type: rtl
source: src/
sources:
- extra_src/
test: test/
tests:
- extra_tests/
topModule: waterfall
targets:
- name: Icarus
  toolchain: iverilog
  tasks:
  - name: Run iverilog simulation
    type: simulation
  - name: pll-test
    type: simulation
    topModule: pll_tb
  - name: clk-doubler-test
    type: simulation
    topModule: clk_doubler_tb
  - name: double-f-test
    type: simulation
    topModule: double_f_tb
- name: Vivado
  toolchain: vivado
  options:
    part: xc7z010clg400-1
  tasks:
  - name: Vivado Synth
    type: synthesis
  - name: Vivado Implementation
    type: implementation
  - name: Vivado Simulation
    type: simulation
    topModule: tb_waterfall
  - name: Vivado Programming
    type: programming
  - name: Vivado Synth double-f
    type: synthesis
    topModule: double_f
- name: Yosys
  toolchain: yosys
  tasks:
  - name: Test Yosys Synth
    type: synthesis
    topModule: waterfall
  - name: Test Yosys
```

```
    type: synthesis
    topModule: top
- name: MLFSP
  toolchain: mlfsp
  tasks:
  - name: Test Synth
    type: synthesis
    topModule: waterfall
```

此配置文件中，包含项目指定的源文件路径、顶层模块等，以及项目中配置的目标工具链参数、运行任务参数等，包括 Icarus Verilog、Vivado、Yosys、MLFSP[17] 四种工具链的配置。配置文件第一层包含了项目的名称、版本、类型、源文件路径、测试文件路径、顶层模块名、等信息。用户可以通过修改该文件，来配置项目的各种参数。配置文件第二层是 `targets`，表示项目的目标，即项目的工具链。在该字段中，用户可以配置项目的工具链名称、工具链类型、工具链参数等。在第三层的 `tasks` 字段中，用户可以配置项目的任务，即项目的具体操作。在 `tasks` 字段中，用户可以配置任务的名称、任务的类型、任务的顶层模块名、任务的源文件路径等。

这三层配置的一些信息是从下往上逐层继承、覆盖的，即在 `tasks` 字段中配置的信息会覆盖在 `targets` 字段中配置的信息，而在 `targets` 字段中配置的信息会覆盖在第一层配置的信息，这些信息包括 `topModule`、`source`、`sources`、`test`、`tests` 等。这样的设计使得用户可以在配置文件中更加灵活地配置项目的参数。

这三层配置从外到内依次是：`ProjectConfig` → `TargetConfig` → `TaskConfig`，这三个 `case class` 均继承自 `abstract class ConfigNode`，`ConfigNode` 是一个抽象类，其中定义了从下到上搜索继承覆盖配置项的方法。其伪代码类图如图 2-6 所示。

一些关键参数的说明：

- `source`：源文件路径。可以指定文件或文件夹，不过只会选择一个项目。
- `sources`：额外的源文件路径。同上，可以指定多个。
- `test`：测试文件路径。在仿真时被指定，格式同上。
- `tests`：额外的测试文件路径。
- `topModule`：顶层模块名。当前任务执行时使用的顶层模块名。
- `constraints`：约束文件路径。
- `exports`：项目自身作为 IP 被导出时导出的 HDL 模块。
- `ipFiles`：文件形式的 IP 路径列表。
- `ipPaths`：IP 搜索路径列表。

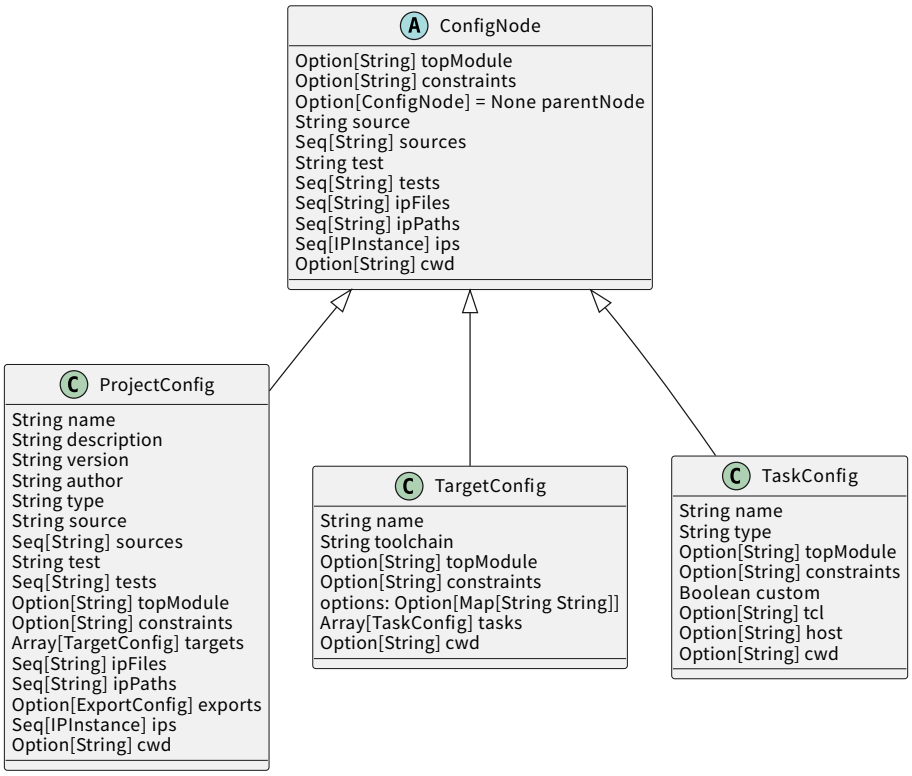


图 2-6　Scaleda 项目配置类图

- `ips`：IP 实例列表。
- `cwd`：当前工作目录。

2.4.2 嵌套化项目配置

由于 Scaleda Project 描述足够详细，能够描述一个抽象的 FPGA 项目，因此 Scaleda 项目本身可以作为一个 IP 被其他 Scaleda 项目所引用。

这样的设计使得代码的复用率更高，用户可以将常用的配置抽象为一个 Scaleda 项目，然后在其他项目中引用该项目，从而减少了配置文件的重复编写。同时，这样的设计也为 Scaleda 扩展 IP 管理、IP 依赖等功能提供了基础。

2.4.3 创建新项目

用户可以在 IntelliJ IDEA 的菜单栏中选择 `File -> New -> Project -> Scaleda Project` 来创建新项目，如图 2-7 所示。

在创建新项目时，用户可以在弹出的配置窗口中选择项目的名称、版本、类型、源文件路径、测试文件路径、顶层模块名等信息，确定后将对应目录下生成新的 `scaleda.yml` 文件。

目前通过已有代码自动创建 Scaleda 模块的功能还有各种问题，仍在实验中。

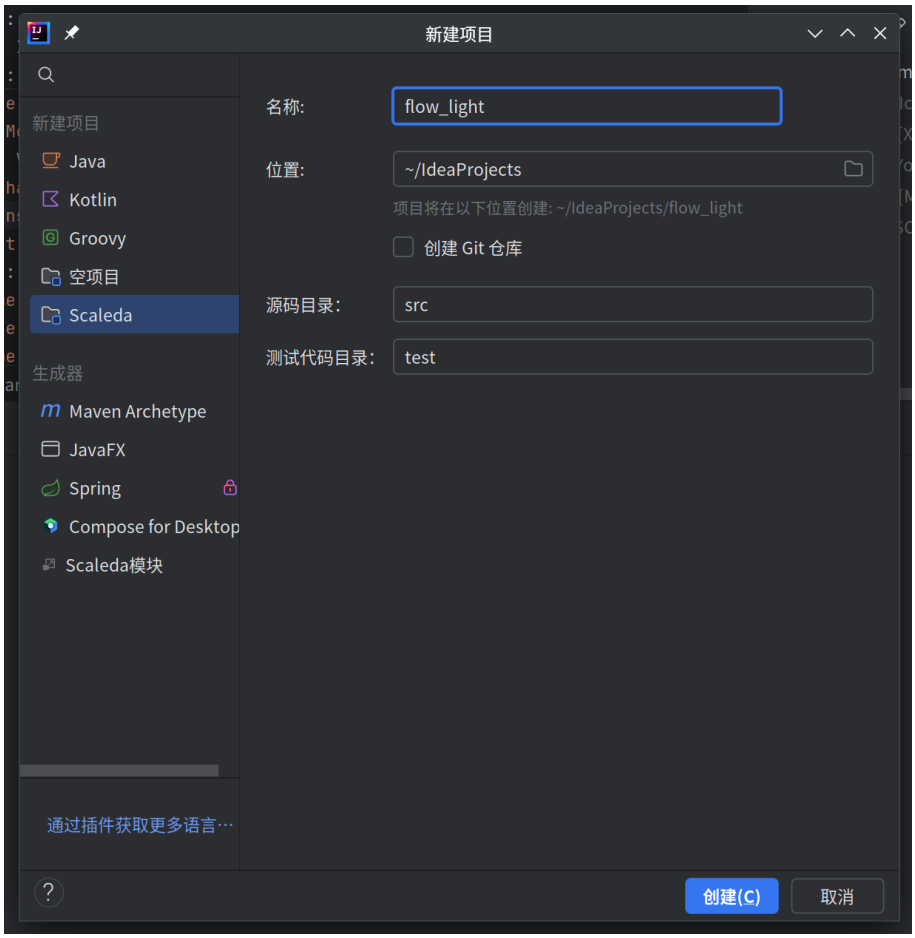


图 2-7　创建新 Scaleda 项目

2.4.4 项目配置编辑

为了让用户更加方便地编辑项目配置文件，Scaleda 实现了一个项目配置编辑器，用户可以通过该编辑器来编辑项目配置文件，而不需要手动编辑 YAML 文件。

在安装 Scaleda 后，用户可以在 IntelliJ IDEA 的右上角工具栏中找到 Scaleda 的任务面板（2-8），用户可以在该面板中查看项目的配置信息、编辑当前项目配置，以及执行项目的任务等。

任务面板上方的工具栏中有多个实用按钮：

- 运行任务：选中并回车将会执行当前选中的任务。
- 展开所有：将会展开配置树中所有的任务。
- 收起所有：将会收起配置树中所有的任务。
- 编辑：打开项目编辑器，用户可以在该编辑器中编辑项目配置。
- IP 核管理：打开 IP 核管理器，用户可以在该管理器中管理项目中的 IP 核。
- 刷新：重新加载当前项目配置。

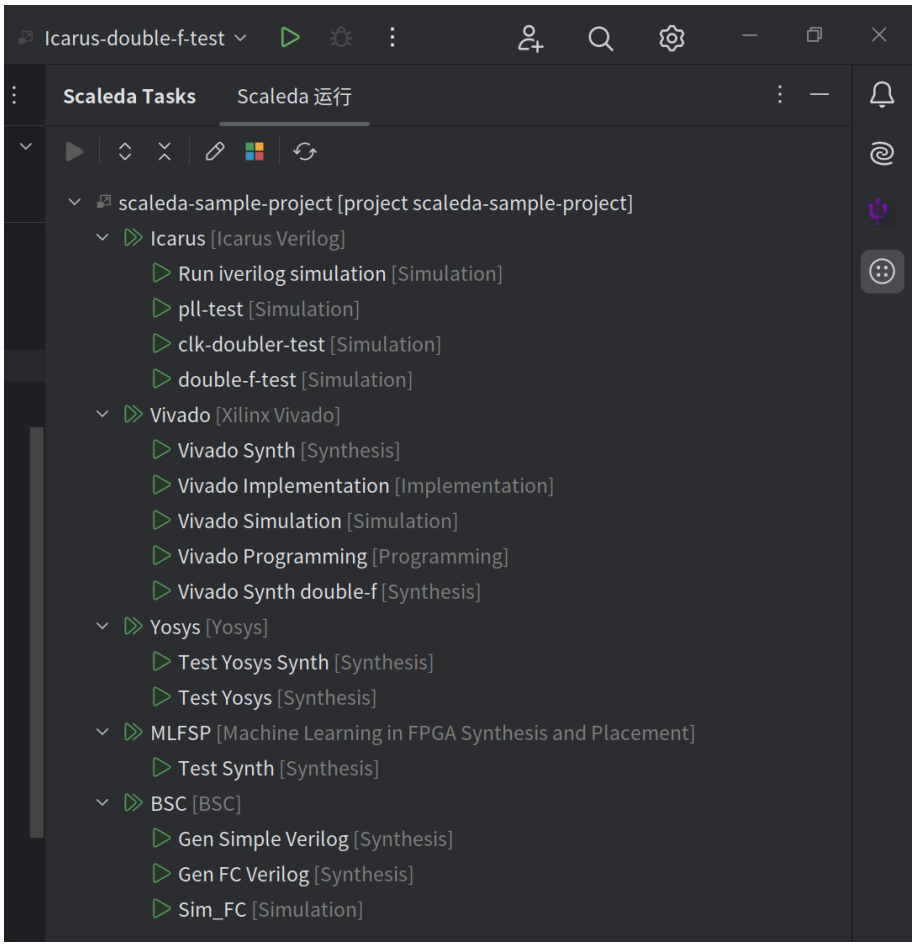


图 2-8　Scaleda 项目任务面板

Scaleda 中的任务面板就类似 Vivado 的 Flow Navigator，用户可以在该面板中选择需要执行的任务。但是 Scaleda 中任务面板与 Vivado 的 Flow Navigator 不同的是，用户可以在任务面板中编辑项目配置，让可执行的任务从单个 FPGA EDA 工具扩展到多个 FPGA EDA 工具，提升工具链层面的选择灵活性；同时能够选择不同的仿真、综合、实现、编程等任务，在同类但是不同的配置当前快速切换，提升了配置选择的灵活性。

当用户打开项目编辑器时，会弹出一个配置窗口如图 2-9，此窗口左侧是项目配置树，右侧是当前选定的项目/目标/任务的配置信息。用户可以在左侧的配置树中选择项目、目标、任务，然后在右侧的配置信息中编辑项目的配置。右侧的配置信息是根据左侧的配置树动态生成的，用户在此编辑器内的操作都是临时的，只有当用户点击保存按钮时，配置文件才会被保存。

其中用户在选择文件时，会动态解析目录下的 HDL 代码，智能地显示可选的 HDL 模块。项目保存时，会将源文件重新格式化，并筛除其中的默认参数，以确保配置文件的简洁性。

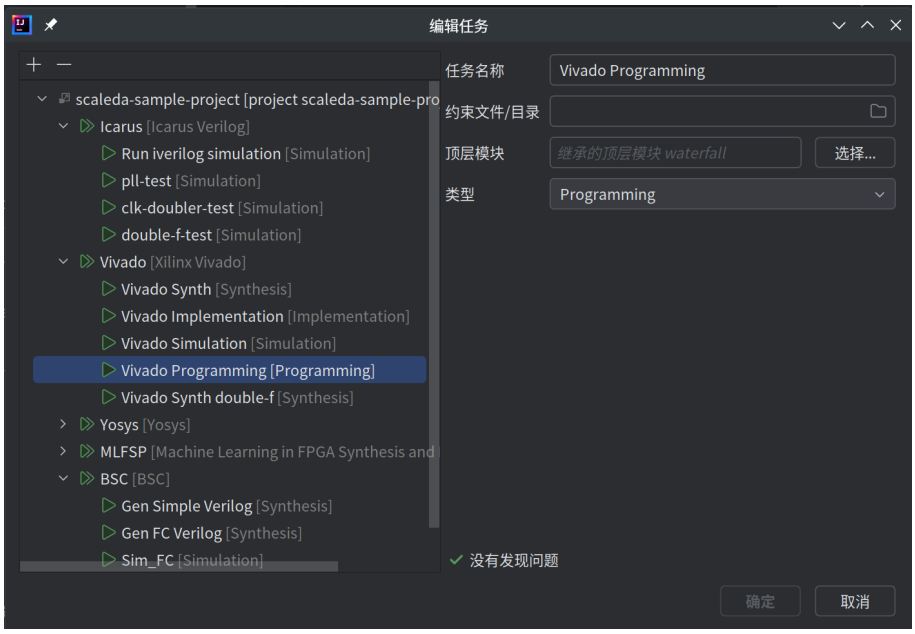


图 2-9　Scaleda 项目配置编辑器

2.4.5 IDEA 运行配置

在 IntelliJ IDEA 中，用户还需要使用 IDEA 的 Run Configuration 来运行用户指定的操作，于是 Scaleda 中同样加入了对 IDEA 的 Run Configuration 的支持。

在用户双击任务面板中的任务时，会自动生成一个 `RunConfiguration`，此配置会链接到配置文件内的一个 `TaskConfig`，并且同样能够覆盖一些经常会改变的配置项。例如远程服务器的地址，其并不适合写入配置文件，因为保存后会泄露用户的隐私，且服务器地址可能随时更改，适合保存在临时的 Run Configuration 中。或者代码执行过程中需要设置一些临时的秘钥等环境变量参数也可以保存在 Run Configuration 中。

用户保存的 Run Configuration 存在于临时目录中，但是用户也可以通过 IDEA 来保存 Run Configuration 到项目目录中，这样可以方便地与其他用户共享配置。

2.5 IP 核管理

IP 核是 FPGA 设计中的重要组成部分，是一种可重用的数字电路模块，用户可以将其作为一个整体插入到 FPGA 设计中。

Scaleda 的 IP 遵循了这一嵌套重用的设计思路，用户可以在 Scaleda 中创建、导入、导出 IP 核，然后将其作为一个整体插入到另一 Scaleda 设计中。

但是，由于 IP 核本身的最终物理实现是与 FPGA 厂商的物理实现相绑定的，因此 Scaleda 中的 IP 核实现并不类似传统 FPGA EDA 内的 IP 设计，而是设计为一

个 IP 核生成器。IP 核通过当前执行任务的参数生成语法桩文件、仿真文件、综合文件、约束文件等，且其可用性是根据用户的配置和当前任务的执行环境而变化的。

在这种情况下，IP 核的管理就变得更加复杂，因为不同的配置参数环境、不同的目标任务都会对 IP 提出不同的要求。因此，目前在 Scaleda 中的 IP 库实现了两个：一个是用于生成存储块的 BRAM IP，一个是用于生成 PLL 时钟的 PLL IP。

2.5.1 IP 核的设计限制和实现

由于许多厂商 IP 核的实现是封闭的，用户无法直接访问其内部实现，因此 Scaleda 中的 IP 核是一种功能层面的抽象，需要根据不同的目标任务生成不同的实现。

例如，当用户需要仿真时，PLL IP 需要生成对应仿真平台的文件。PLL 是 FPGA 内部的 ASIC 时钟模块，其实现是与 FPGA 厂商的物理实现相绑定的，只能在特定条件下被仿真。于是在当前的实现中，PLL IP 在使用 Icarus Verilog 仿真时，如果使用了 PLL IP，Scaleda 会生成一个警告，提示用户无法仿真 PLL IP。而当使用 Vivado 仿真时，PLL IP 会生成一个 Vivado IP 并被添加到 Vivado 仿真中。

尽管如此，这样的 IP 核设计仍然局限性很大。还是以 PLL 为例，IP 核需要根据当前频率参数生成 Vivado IP，但是 Scaleda 并没有办法访问 Vivado 的 IP 生成器参数，因此目前只能让用户手动打开 Vivado，生成 PLL IP 文件后导入 Scaleda 中使用。

在限制相对没有那么大的情况下，BRAM IP 的设计就相对简单了。BRAM IP 是一个存储块，其实现是相对简单的。Scaleda 会根据用户请求的容量参数生成一个可以直接综合和仿真的 Verilog 文件，其中与具体硬件 Block Memory 的绑定能够交给 Vivado 在综合实现时自动完成。

2.5.2 Scaleda IP 的特性

上文基本描述了 Scaleda IP 的设计思路和实现方法，尽管目前设计相对还不成熟，但是这样实现的 Scaleda 已经能够显著增强 IP 核的灵活性。

首先，`scaleda.yml` 作为一个被附加在项目当中的描述性配置文件，通过内部的配置可以灵活配置路径、参数等信息，能够在不改变代码的情况下生成 Scaleda IP。于是类似 Archlinux AUR 的 PKGBUILD 的设计，用户可以方便地在开源社区上分享自己的 IP 核。当前由于各 EDA 内部完全使用不同的路径参数等，使得网络上的开源设计都不能被直接导入使用，而 Scaleda 的设计可以使得用户快速导入已经适配好的开源 IP 核。

由此，配合 Scaleda 的项目管理功能，用户能够在项目中直接指定 Github 等仓库的 IP 核，然后通过 Scaleda 的 IP 核管理器直接导入到项目中，而不需要手动下载、解压、复制等操作。

借由此 IP 核和项目结构，能极大地提高 IP 核的复用率，使得用户能够更加方便地使用开源社区的 IP 核，促进数字电路设计的开源化。

2.5.3 IP 核管理器

Scaleda 实现了以上描述的特性，并为用户提供了一个图形化的 IP 核管理器。在“Scaleda Tasks”侧边栏中点击“IP 管理器”，将打开本项目的 IP 核管理器窗口，如图 2-10 所示。

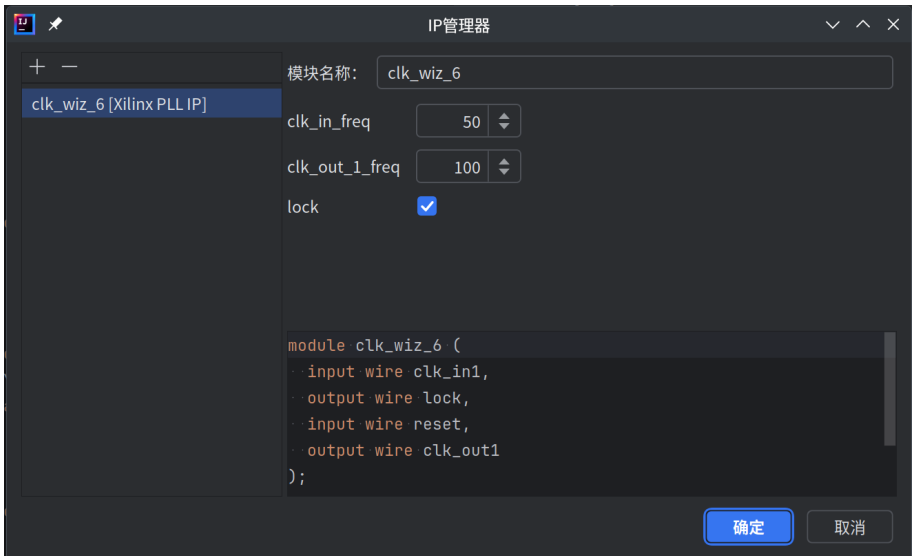


图 2-10　Scaleda IP 核管理器

IP 核管理器左侧为已经添加的 IP 核列表，右侧为当前选中的 IP 核的详细信息，包含当前 IP 参数以及将要生成的 Verilog 模块接口等信息。

点击加号，将会打开一个对话框，用户可以在该对话框中选择 Scaleda IP 核，其中将显示各个支持的 IP 核对每个工具链的支持情况，如图 2-11 所示。

2.6 执行 EDA 功能

目前 Scaleda 内部并没有实现 EDA 工具链的仿真、综合、实现、编程等功能，这些功能都是在 Scaleda 外部实现的，本章节将介绍 Scaleda 如何处理 EDA 工具链的执行过程。

图 2-11　使用 Scaleda IP 核管理器添加 IP 核

2.6.1 主要设计思路

类似于背景中提到的 edalize，Scaleda 也是通过调用外部的 EDA 工具链来实现仿真、综合、实现、编程等功能的，且也在内部为多种 EDA 工具链提供了抽象统一的 API 接口。

一个 EDA 工具链的执行过程在 Scaleda 中被抽象为：

1. 输入 HDL 设计文件
2. 输入约束文件
3. 输入 IP 核文件
4. 生成工具链参数
5. 设定程序工作环境
6. 执行工具链，收集输出
7. 判定执行结果，解析输出
8. 执行后续任务

在上述步骤中，Scaleda 可以通过配置文件内的各种参数修改上述步骤的执行过程，以及执行过程中产生的文件内容。例如，通过配置的 `sources` 目录参数生成需要执行的 HDL 设计文件列表，又或者根据配置的 `targets.options` 设置目标 FPGA 平台的硬件参数，让工具链选择正确的硬件库。

同时，由于工具链的执行过程也被抽象为任务的执行，所以将工具链任务拉远到服务器上执行也是可行的了。

2.6.2 EDA 工具链调用 API

Scaleda 项目实现初期，为了方便地调用外部的 EDA 工具链，设想过直接使用 edalize 等工具，但是由于 edalize 使用 Python 库设计，并不能开箱即用，因此 Scaleda 还是项目内部实现了一个 EDA 工具链调用相关的 API。

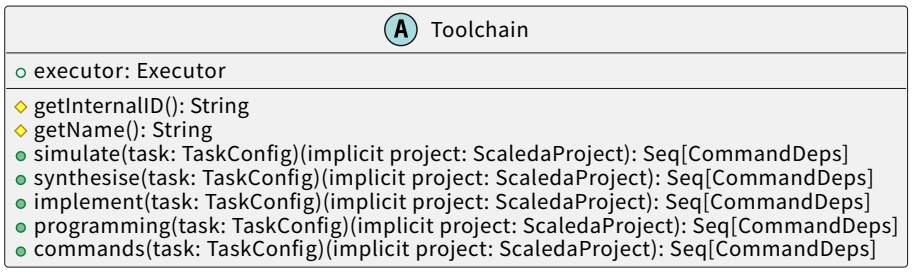


图 2-12　EDA 工具链调用 API

`simulate`、`synthesise`、`implement`、`programing` 方法分别对应了仿真、综合、实现、编程等任务的执行，这些方法会根据当前的任务配置，生成需要执行的命令以及对应环境参数的列表，这些列表会被传递给后续方法，以在本地或远程执行对应的任务。

目前已经实现了以下 EDA 工具链的调用，如图 2-13 所示。

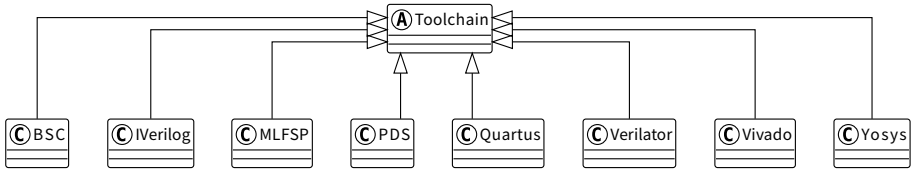


图 2-13　已支持的 EDA 工具链调用

关于 EDA 工具链的调用相关的配置，是被存储在用户目录下的 `.scaleda` 目录中的，这个目录是 Scaleda 的全局配置目录，用于存储用户的配置文件、秘钥、解压的可执行程序等。

2.6.3 Vivado 工具链调用的支持

Vivado 是 Xilinx 公司的 FPGA EDA 工具链，是目前业界最流行的 FPGA EDA 工具之一，因此 Scaleda 项目中对 Vivado 的支持是必不可少的。本小节将以 Vivado 工具链调用为例，说明 Toolchain API 的调用过程。

目前 Scaleda 中实现的 Vivado 调用，是在 Vivado 的 Project 模式下执行的，即 Scaleda 需要在 Vivado 中创建一个 Project，然后修改这个临时的 Project 的配置文件，最后执行 Vivado 的命令行工具来执行这个 Project。

创建、配置临时 Vivado Project 的过程是通过 Tcl 脚本实现的，而此 Tcl 脚本是 Scaleda 根据当前配置，从内部的 Jinja2 模板生成的。

以内部的 `create_project.tcl.j2` 模板为例：

```
source args.tcl
file mkdir $project_dir
```

```
puts "Generating project ${project_name}"

create_project -part $part -force $project_name $project_dir

{{ insertTclSection }}

{% for s in sourceList %}
puts "Importing source: {{ s }}"
add_files -fileset sources_1 { {{ s }} }
{% endfor %}

{% if sim %}
puts "Importing testbench: {{ testbenchSource }}"
add_files -fileset sim_1 { {{ testbenchSource }} }
set_property top {{ top }} [get_filesets sim_1]
{% else %}
set_property top {{ top }} [get_filesets sources_1]
{% endif %}

update_compile_order -fileset sources_1
update_compile_order -fileset sim_1

{% for ip in ipList %}
puts "Importing IP core: {{ ip }}"
import_files -norecurse { {{ ip }} }
{% endfor %}

{% for xdc in xdcList %}
puts "Importing xdc file: {{ xdc }}"
add_files -fileset constrs_1 { {{ xdc }} }
{% endfor %}

close_project
```

在这个模板中，{{ insertTclSection }} 是一个占位符，用于插入用户自定义的 Tcl 脚本段，用户可以在配置文件中配置这个段落，以实现更加复杂的配置。用户的配置文件中，可以配置 Vivado Project 的名称、目标 FPGA 平台、源文件列表、IP 核列表、约束文件列表等信息，这些信息会被填充到模板中，生成一个可以直接执行的 Tcl 脚本。

替换完成后，还需一个特殊的正则替换，以修正 Jinja2 造成的一些格式问题：

```
/** { {{module}} } =template=> { module_value } ==> {module_value}
  */
object TemplateContextReplace extends RegexReplace("\\{␣([^\\{\\}]+)
   ␣\\}", Seq("{$1}"))
```

RegexReplace 是 ImplicitPathReplace 的子类，这些替换用的 Implicit 会在程序执行的上下文中自动插入替换。

Vivado 执行前，总共替换这些模板文件：

```
class TemplateRenderer(rt: ScaledaRuntime)(implicit replace:
   ImplicitPathReplace = NoPathReplace)
   extends ResourceTemplateRender(
     "tcl/vivado",
     rt.executor.workingDir.getAbsolutePath,
     Map(
       "args.tcl.j2"         -> "args.tcl",
       "create_project.tcl.j2" -> "create_project.tcl",
       "run_sim.tcl.j2"      -> "run_sim.tcl",
       "run_synth.tcl.j2"    -> "run_synth.tcl",
       "run_impl.tcl.j2"     -> "run_impl.tcl",
       "run_program.tcl.j2" -> "run_program.tcl"
     )
   )(replace)
```

其他的工具链的支持逻辑也是类似的，有些工具链需要不同的脚本模板，有些工具链只需要配置不同的调用参数，也有些工具链支持通过容器方式调用，这些能通过全局配置中的工具链配置实现。

2.6.4 自定义 EDA 工具链

当用户需要使用 Scaleda 中没有支持的 EDA 工具链时，用户可以在全局的

`.scaleda/toolchains` 目录下创建一个新的配置文件，然后在配置文件中配置该工具链的调用参数。目前支持的相关配置参数是通过环境变量等传递给用户的自定义脚本的。

万其琛同学在他的毕业设计 [17] 中，将强化学习应用于 Berkeley ABC 综合器[18]，取得了不错的效果，这样的工作也可以通过 Scaleda 的自定义工具链支持。

在自定义的运行脚本 `run.sh`、`run.bat` 中，通过添加自定义 Yosys 脚本对 Yosys 内部的 ABC 综合器进行了替换，再通过 `abc-wrapper.py` 脚本对强化学习算法应用的 ABC 综合器进行了封装。当使用者在 Scaleda 中选择 “MLFSP” 工具链时，会调用这个自定义的脚本，从而实现了将自定义的综合器放入 Scaleda 的工具链中。

2.6.5 EDA 工具链的检测与验证

在 Scaleda 中，用户可以通过配置文件指定 EDA 工具链的路径，但是这样的配置是不够安全的，因为用户可能会配置错误的路径，或者用户的电脑上没有安装对应的 EDA 工具链。

为了解决这个问题，Scaleda 实现了一个 EDA 工具链检测与验证的功能，用户在输入工具链路径后，Scaleda 会自动检测该路径下是否存在对应的工具链可执行文件，如果存在，Scaleda 会尝试执行该可执行文件，以验证该文件是否可用。只有当该文件可用时，Scaleda 才会将该路径保存到配置文件中。

此功能的实现也类似上面 `Toolchain.simulate` 等方法，通过一个返回程序调用环境的信息，再判断执行程序后的返回值以及输出是否正常，从而判断工具链是否可用。

当程序第一次启动时，会自动检测在一些默认路径下检查是否存在已经支持的工具链，如果存在则自动配置，否则提醒用户手动配置。

2.7 Vivado 集成模式

通过 Toolchain API 来支持 Vivado 工具链，固然提升了灵活性，降低了初学者的入门门槛，但是对于已经习惯了 Vivado 的用户来说，这样的设计可能并不友好。

因此，Scaleda 还实现了 Vivado 集成模式，用户可以在 Scaleda 中直接打开 Vivado 格式的项目，并且直接在 Scaleda 中执行 Vivado 的仿真、综合、实现、编程等任务。

相比于 Toolchain API，Vivado 集成模式每次只打开一个 Vivado 项目，其操作逻辑等同于直接打开 Vivado，已经熟悉了 Vivado 的用户使用 Scaleda 就没有太多的学习成本。

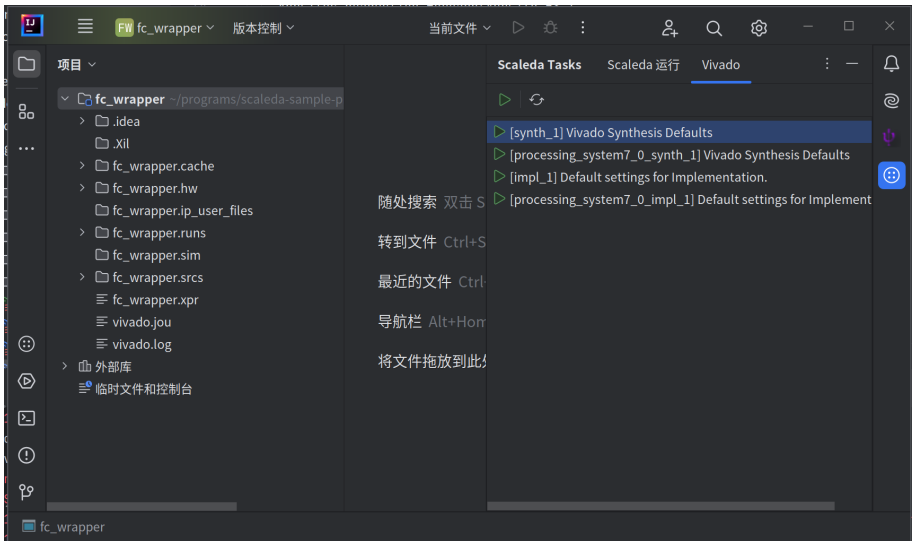


图 2-14　Vivado 集成模式

如图 2-14 所示，打开项目时若检测到有效的 `*.xpr` 文件，Scaleda 会为项目启用 Vivado 集成模式。Vivado 集成模式下，Scaleda 将加载并解析 Vivado 的 `.xpr` 项目文件，并为能够执行的仿真、综合等任务提供任务列表，用户可以直接在 Scaleda 中执行这些任务。

2.8 项目和全局设置

上文已经提到，Scaleda 通过在用户目录下的 `.scaleda` 目录中存储用户的配置文件、秘钥、解压的可执行程序等，来实现全局设置。本小节将介绍 Scaleda 全局设置实现的一些细节。

当前已经安装并使用的 `.scaleda` 目录结构大致如下：

```
$ tree ~/.scaleda
/home/chiro/.scaleda
|-- authorization.yml
|-- bin
|   |-- rvcd
|   |-- rvcd.exe
|   |-- surfer
|   |-- surfer.exe
|   |-- verible
|   |   |-- lin64
|   |   |   |-- README.txt
|   |   |   `-- verible-verilog-format
|   |   |-- README.txt
|   |   `-- win64
|   |       `-- verible-verilog-format.exe
|   `-- version
```

```
|-- config.yml
|-- ip
|   |-- clocking-wizard
|   |   `-- scaleda.yml
|   `-- scaleda_bram
|       |-- bram2.v.j2
|       `-- scaleda.yml
|-- scripts
|   `-- vivado_call.tcl
`-- toolchains
    |-- bsc.yml
    `-- yosys.yml

10 directories, 17 files
```

- `authorization.yml`：用户的授权文件，用于存储用户的鉴权信息。
- `bin`：解压好的可执行文件目录，包含外置格式化工具、波形显示器等。
- `config.yml`：用户的全局配置文件，用于存储用户的全局配置信息。
- `ip`：Scaleda 安装时自动释放的内置 IP 核，用户可以直接导入到项目中使用。
- `scripts`：存储一些脚本文件，用于辅助 Scaleda 的运行。
- `toolchains`：存储用户配置的 EDA 工具链的配置文件。

当 Scaleda 检测到 `bin` 目录下的 `version` 比内置打包好的版本新时，会自动更新内置的可执行文件，这样用户就不需要手动下载、解压这些文件了。

`toolchains` 下的配置文件是用户配置的 EDA 工具链的配置文件，用户可以在该目录下配置 Vivado、Icarus Verilog、Yosys 等工具链的配置文件。这些配置文件也是相对固定的，在授课场景下教师可以提前配置好，学生只需要导入配置文件即可使用。

2.9 命令行接口

在 Scaleda 中，用户可以通过命令行接口来执行项目的任务，这样用户就可以不需要打开 IntelliJ IDEA，也能够执行项目的任务。由此，Scaleda 也能适用于纯命令行的应用场景中，如服务器执行或者持续集成测试等。

Scaleda 在代码结构设计时，就考虑到了命令行接口的实现，将内部关键逻辑等分离到 `scaleda-kernel` 模块中，从而编译生成的 `scaleda-kernel.jar` 可以在命令行中执行。

2.9.1 命令行参数

在 Scaleda 的命令行接口中，命令被分为不同的命令组，每个命令组下有不同的子命令，每个子命令下有不同的参数。当让 `scaleda` 指向 `java -jar /path/to/scaleda-kernel.jar`，输入`scaleda --help`显示帮助信息如下：

```
$ scaleda --help
05:29:33.364 [main] INFO scaleda-kernel - [ScaledaShellMain.scala:78 main] This is
    Scaleda-Shell, an EDA tool for FPGAs
05:29:33.659 [main] INFO scaleda-kernel - [ScaledaShellMain.scala:105 main] No project
    config detected!
scaleda
Usage: scaleda [remote|list|manage|create|run|tools] [options]

   --help               Prints this usage text
   -C, --workdir <value> Working directory
   -h, --host <value>   Set server host for RPC
   -u, --username <value> Specify username
   -p, --password <value> Specify password
Command: remote [serve|login|register|refresh]

Command: remote serve
       Run as server
Command: remote login
       Login into server
Command: remote register [options]
       Create account in server
   -n, --nickname <value> Specify nickname
Command: remote refresh
       Refresh token
Command: list [profiles|tasks|configurations]

Command: list profiles
       Show loaded profiles
Command: list tasks
       Show loaded tasks
Command: list configurations
       Show all configurations to run
Command: manage [install|clean]

Command: manage install
       Install necessary binaries force
Command: manage clean
       Clean all data on device
Command: create [options]
```

```
Create Scaleda Project
   -n, --name <value>    Specify project name, default is 'scaleda-rtl'
   --empty <value>       Create empty project, do not detect project structure
Command: run [options]
Run task
   -c, --config <value>  Specify configure
   -t, --task <value>    Specify the task
   -r, --target <value>  Specify the target, otherwise will auto fill
   -p, --profile <value> Specify profile name, otherwise will auto fill
   -e, --environment <value>
                         Specify environment, example ENV_A=a;ENV_B=b
Command: tools [ast]
Useful tools
Command: tools ast [options]
Parse & show AST of a file
   -f, --file <value>    Specify file
   -t, --type <value>    Specify file language type, available: Verilog, SystemVerilog,
       Bluespec
```

Scaleda Kernel 的命令行接口支持的命令有 `remote`、`list`、`manage`、`create`、`run`、`tools` 等，其中：

- `remote`：远程服务器相关命令，用于远程服务器的登录、注册、刷新等操作，或是直接作为远程服务器启动。
- `list`：列出当前项目的配置、任务、配置文件等信息。
- `manage`：管理当前项目的安装、清理等操作。
- `create`：创建一个新的 Scaleda 项目。
- `run`：执行当前项目的任务。
- `tools`：一些实用工具，如解析 HDL 文件的 AST 等。

命令行参数解析是依靠 `scopt` 库实现的，实现逻辑较为简单，不再赘述。

2.10 远程服务

上文提到 Toolchain API 抽象出的 EDA 工具链调用过程，将工具链的执行过程抽象为一个任务的执行过程，而一个准备好的任务执行环境，实际上既可以在本地执行，也可以在远程服务器上执行。

于是 Scaleda 实现了一种远程调用服务的支持，用户可以在本地配置好远程服务器的地址、用户名、密码等信息，然后通过命令行接口将任务发送到远程服务器上执行。

此功能是为了解决厂商 FPGA EDA 工具链安装复杂、资源消耗大等问题，用户将任务发送到远程服务器上执行，这样就可以在本地节省资源，而只在远程服

务器上执行任务。这样的设计十分契合有校园内网、校园服务器集群的情况，同时也能让宝贵的 FPGA EDA 许可证得到更好的利用。有了这个功能，数字电路设计的初学者，或者是没有太多资源的个人用户，都可以不经过本地工具链的安装，就能够使用 FPGA EDA 工具链，达成真正的开箱即用的配置效果。

2.10.1 鉴权系统

为了保证远程服务器的安全性，Scaleda 实现了一个简单的鉴权系统。为了契合多任务、任务之间关联性不强的特性，Scaleda 的鉴权系统是基于 JWT（JSON Web Token）的，用户在登录时会生成一个 JWT，然后在后续的请求中携带这个 JWT，服务器会验证 JWT 的有效性，从而判断用户是否有权限执行任务。

由于 JWT 是基于 Json 的，并不特别依附于传统的 HTTP 等用于网页浏览的协议，因此 Scaleda 的 JWT 鉴权系统被加在 gRPC 的请求头中，用户的远程调用通过 gRPC 协议进行。

未注册或登录的用户，只能执行一些简单的任务，如查看配置、查看任务等，而登录后的用户可以执行更多的任务，如执行任务、查看服务器配置等。

2.10.2 远程调用

Scaleda 以及 Rvcd 的主要实现语言是不同的，于是这里也选择了一种跨语言的 RPC 框架，gRPC。

gRPC 是一个高性能、开源和通用的 RPC 框架，面向多种语言。其使用 protobuf 作为消息的序列化工具，使用 HTTP/2 作为底层传输协议。相较于使用 Json 作为消息序列化工具的 RESTful API，gRPC 的性能更好，更适合用于高性能、低延迟的场景。

除了性能更好外，gRPC 还能够支持流式传输、双向流式传输等特性，这些特性在 Scaleda 的远程调用中也有所体现，而且也是 Scaleda 实现设计的远程调用功能不可或缺的。

Scaleda 中使用的 gRPC 接口定义文件位于 `scaleda-kernel/src/proto` 目录下，包括：

- `remote.proto`：远程调用服务的接口定义，包括运行任务、用户管理、拉去服务器配置等接口。
- `scaleda.proto`：Scaleda 项目的接口定义，包括了源码跳转接口，用于与波形查看器联动，被波形查看器所调用。
- `rvcd.proto`：波形查看器的接口定义，包括了波形查看器的启动、打开文件、波形跳转等接口。

- `fs.proto`：文件系统的接口定义，负责实现远程 FUSE 文件系统传输。

当 Scaleda 被编译的时候，这些接口定义文件会被 `ScalaPB` 编译成对应的 Scala 文件，然后被引入到 Scaleda 的代码中。`ScalaPB` 是一个 Scala 语言的 Protocol Buffers 编译器，能够将 Protocol Buffers 文件编译成 Scala 代码。

2.10.3 远程文件系统

在远程调用中，用户需要将本地和远程的文件进行同步，才能够在远程服务器上执行任务。为了实现这个功能，Scaleda 实现了一个远程文件系统，使用 gRPC 的流式传输功能，将本地文件传输到远程服务器上。

FUSE 是一个允许非特权用户在用户空间实现文件系统的 Linux 内核模块，用户可以通过 FUSE 实现自己的文件系统，而不需要修改内核代码。FUSE 的设计思路是将文件系统的 API 消息通过一个自定义的信道传输到用户空间的文件系统程序中，然后由文件系统程序来处理这些消息。也就是说，FUSE 能让用户在用户空间使用自定义的方式来实现并挂载自定义的文件系统。其逻辑如图 2-15 所示。

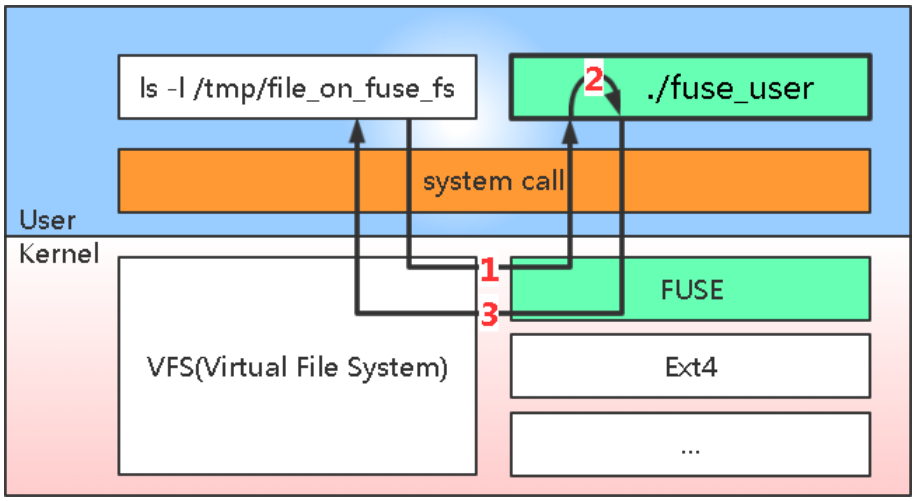


图 2-15　FUSE 文件系统工作原理

类别其他一些文件系统，如 sshfs，也是将 FUSE 相关的 API 消息通过自定义的 ssh 信道传输到远程服务器上，Scaleda 也采用了类似的设计。其系统功能示意图如图 2-16 所示。

此处的一个设计细节是，由于本地计算机和远程服务器之间不一定是直接连接的，可能会经过 NAT 网络，因此 Scaleda 为了解决远程服务器到本地计算机之间的主动连接，使用了 gRPC 的双向流式传输功能。本地计算机主动连接远程服务器后打开一个流，接收远程服务器端发来的请求，然后根据请求的内容，再向远程服务器端发送 FUSE 相关的系统调用的结果。

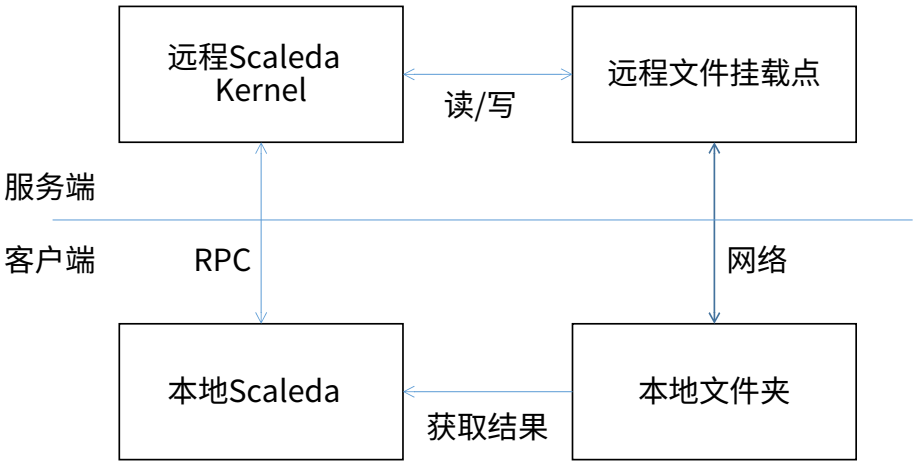


图 2-16　远程文件系统示意图

由此达成的效果是，尽管远程服务器端可能无法直接访问本地计算机，但是远程服务器端还是可以在服务器的文件系统中挂载本地计算机的文件内容，实现无需上传文件，直接在远程服务器上执行任务的效果。

然而，由于 FUSE 文件系统的实现比较复杂，目前不同操作系统之间的远程 FUSE 访问还是有可能出现问题，又或是可能出现不支持的文件系统扩展造成 EDA 工具执行异常等问题，因此 Scaleda 的远程文件系统功能还是一个实验性的功能，用户在使用时需要注意。

2.10.4 远程任务执行

通过上述说明，远程任务的执行逻辑已经比较清晰明了了。以下是 `remote.proto` 中的一部分定义：

```
syntax = "proto3";

package top.scaleda.kernel.net;

service Remote {
    rpc Run(RunRequest) returns (stream RunReply) {}
    rpc GetProfiles(Empty) returns (ProfilesReply) {}
    rpc GetRemoteInfo(Empty) returns (RemoteInfo) {}
    rpc StopCommand(StopCommandRequest) returns (StopCommandReply) {}
}

message Empty {}

message StringTriple {
    string a = 1;
    string b = 2;
}
```

```
message RunRequest {
    repeated string commands = 1;
    string path = 2;
    repeated StringTriple envs = 3;
    string projectBase = 4;
    string runId = 5;
}

enum RunReplyType {
    RUN_REPLY_TYPE_RETURN = 0;
    RUN_REPLY_TYPE_STDOUT = 1;
    RUN_REPLY_TYPE_STDERR = 2;
    RUN_REPLY_TYPE_ERR_AUTH = 3;
    RUN_REPLY_TYPE_COMMAND_ID = 4;
}

message RunReply {
    RunReplyType reply_type = 1;
    string str_value = 2;
    int32 int_value = 3;
    bool finishedAll = 4;
    bool meetErrors = 5;
}

message StopCommandRequest {
    string commandId = 1;
    uint64 timeoutMs = 2;
}
message StopCommandReply {
    bool ok = 1;
    string reason = 2;
}

// Note that these value cannot be null!!!
message RemoteProfile {
    string profileName = 1;
    string toolchainType = 2;
    string path = 3;
    string iverilogPath = 4;
    string vvpPath = 5;
    string iverilogVPIPath = 6;
}
message ProfilesReply {
    repeated RemoteProfile profiles = 1;
}
```

```
enum RemoteOsType {
    REMOTE_OS_TYPE_LINUX = 0;
    REMOTE_OS_TYPE_WINDOWS = 1;
    REMOTE_OS_TYPE_MACOS = 2;
}
message RemoteInfo {
    RemoteOsType os = 1;
    string tempPrefix = 2;
}
```

客户端首先拉取远程服务器的服务器信息和配置文件，然后发送任务执行请求，远程服务器接收到请求后执行任务，将任务的输出流式传输给客户端，客户端接收到输出后进行消息输出相关的处理。

此处也有许多设计细节，例如在远程服务器上的执行路径并不是固定的，而是一个随机生成的临时路径，用于避免不同用户的任务之间的干扰；同时返回的消息的解析也要进行相应的路径替换，才能在本地正确显示。

当然，除了使用 FUSE 进行远程文件传输的连接，Scaleda 也提供了直接打包复制文件的方式，以提升远程调用的兼容性。

通过远程任务执行功能，用户可以在本地计算机上执行任务，而将任务发送到远程服务器上执行，实现了基本无感知地在本地直接执行了远程服务器上的任务，用户无需安装也无需配置环境，大大降低了用户的学习成本和使用门槛。

2.11 波形交互

在数字电路设计中，波形查看器是一个非常重要的工具，用户可以通过波形查看器查看仿真的波形，从而判断设计的正确性。但是，除了厂商 FPGA EDA 内部提供的波形查看器能够获取到波形数据以外的信息来辅助调试，其他开源的波形查看器往往并不能获取到波形数据以外的信息，这就导致了用户在仿真数字电路时需要频繁地切换窗口。HDL 代码与仿真波形之间存在很难逾越的鸿沟，使得数字电路设计仿真过程中的调试变得非常困难。

为了解决这个问题，本项目实现了一个波形查看器 Rvcd。Rvcd 的具体实现逻辑在后文中会详细介绍，这里主要介绍 Rvcd 与 Scaleda 之间的交互。

2.11.1 代码与波形之间的跳转

当仿真结束时，用户可以选择打开生成的波形文件，由 Rvcd 完成波形文件的解析和显示。

而在打开 Rvcd 窗口后，Scaleda 发现监听的 gRPC 端口能够访问到 Rvcd 的服

图 2-17　由源代码跳转到波形

务器，于是将波形跳转功能设置为可用。此时用户在源代码的语法元素上点击右键，点击跳转到波形，就能够在 Rvcd 中跳转到对应的波形位置。

由于 Verilog 的模块结构是树状的，所以用户选择的语法元素可以指定为语法树上的某个节点到语法树树根的路径，Scaleda 将这个路径发送给 Rvcd，Rvcd 根据当前打开的项目重新解析波形文件，找到对应的波形位置，然后跳转到对应的波形元素上。

而 VCD 文件的信号标记树也是树状的，所以 Rvcd 也可以根据用户选择的波形元素，找到对应的波形位置，然后跳转到对应的源代码位置上。Rvcd 内置有一个源码查看窗口，用户可以选择是跳转到内置的源码查看窗口，还是跳转到外部的编辑器中。

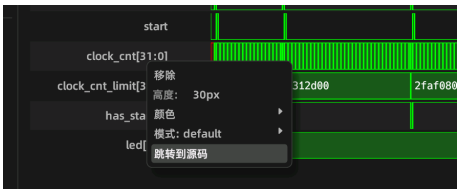


图 2-18　由波形跳转到源代码

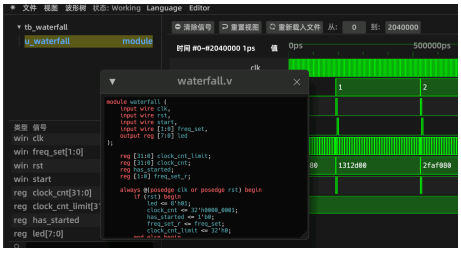


图 2-19　在 Rvcd 中显示跳转到的源代码

当前，有些仿真器生成的波形，其信号路径与源代码的路径不一定完全一致，所以此处还实现了一个简单的路径匹配算法，跳转时寻找匹配路径最长的波形元素，以提高跳转的准确性。

2.11.2 波形查看器在 IDEA 窗口中的集成

Rvcd 是一个独立的波形查看器，用户可以单独打开 Rvcd 窗口来查看波形，但是这样的设计并不友好，用户还是需要频繁地切换窗口。为了解决这一问题，Scaleda 与 Rvcd 同时还实现了窗口投射功能。用户在 Scaleda 内，能够直接在 IDEA 窗口中启动 Rvcd 查看波形，而不需要再在多个窗口之间切换，效果如图 2-20 所示。

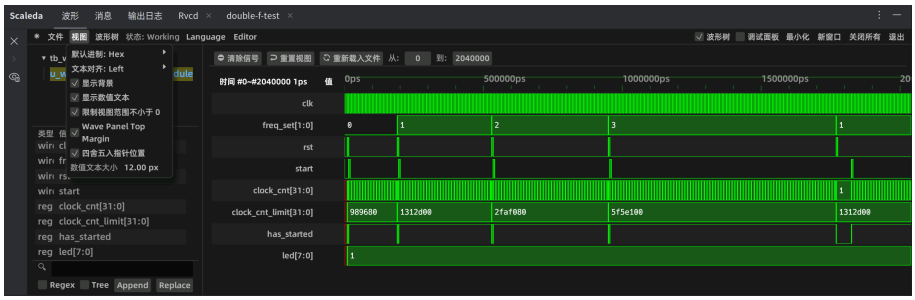


图 2-20　Rvcd 在 IDEA 窗口中的集成

此处的实现原理，将在后文中详细介绍。

2.11.3 使用其他波形查看器

除了 Rvcd，用户还可以使用其他波形查看器，如 GTKWave、Surfer 等。通过设置，Scaleda 可以使用其他波形查看器打开波形，实现了波形查看器的可插拔性。但是，其他波形查看器并不能实现代码与波形之间的跳转等功能。

目前 Rvcd 以及 Surfer 已经被编译并内置到 Scaleda 中，随着用户更新 Scaleda，这些波形查看器也会被更新，而 GTKWave 等其他查看器则需要用户自行安装。

2.12 国际化支持

为了适应不同国家和地区的用户，Scaleda 实现了国际化支持，显示的文本信息、提示消息、错误信息等都以当前用户的语言显示。这是通过 IntelliJ 的 `DynamicBundle` 实现的，Scaleda 需要显示文本时都会从此 Bundle 中获取字符串并进行模式匹配。当前已支持的语言有简体中文和英文。

2.13 本章小结

本章主要介绍了 Scaleda 的核心功能，包括 EDA 工具链调用、Vivado 集成模式、项目和全局设置、命令行接口、远程服务、波形交互等功能的实现细节。

第 3 章 Rvcd 波形查看器设计与开发

本章主要介绍了 Rvcd 波形查看器的设计思路、设计细节与开发过程。

3.1 概述

Rvcd 是本项目中实现的一个波形查看器，主要功能即为查看波形文件。而除了查看波形文件外，Rvcd 还支持了波形和源代码之间的交互、HDL 代码解析、与 Scaleda 集成等多种功能。

Rvcd 是针对高性能、高可靠性、高可扩展性设计的波形查看器，通过使用 Rust 语言进行全异步程序的开发，使得它能在具有较高性能的同时，能够在 Linux、Windows、macOS 等多种操作系统上运行，甚至能够编译到 WebAssembly 并运行在浏览器中。

Rvcd 是 Rust VCD Viewer 的缩写，VCD 即 Value Change Dump，是一种用于描述数字电路仿真波形的文件格式，是一种非常常见的波形文件格式，几乎所有的数字电路仿真工具都支持 VCD 格式的波形文件。

Rvcd 选择使用 Rust 语言进行开发，是因为 Rust 语言具有内存安全、并发安全、无数据竞争、无垃圾回收等诸多优点，同时 Rust 语言的异步编程模型也非常适合于波形查看器这种 IO 密集型的应用。另外，Rust 具有非常好的跨平台支持，能够在多种操作系统上运行，以至于能够编译到 WebAssembly 并运行在浏览器中，具有非常好的可移植性和扩展性。

功能模块设计方面，Rvcd 主要包括以下几个部分：

- 波形文件解析：支持 VCD 格式的波形文件解析，并进行存储空间和数据索引优化。
- 波形查看：支持渲染波形文件，并支持波形的缩放、平移、选择、标记等操作，注重性能优化和用户体验。
- HDL 代码解析：支持解析 HDL 代码，支持查看 HDL 代码的层次结构、查看 HDL 代码的波形。
- 波形交互：支持波形和源代码之间的交互，支持点击波形跳转到源代码、点击源代码跳转到波形。
- Scaleda 集成：支持与 Scaleda 集成，支持在 Scaleda 中直接查看波形，并调用相关互动功能。

3.2 设计思路

Rvcd 作为一个独立的异步应用程序，其内部主要有以下几类协程：

- 主协程：负责用户界面渲染、事件处理、波形渲染等。
- 波形文件读取协程：负责波形文件数据读取，并向主协程报告读取进度，向波形文件解析协程发送数据。
- 波形文件解析协程：将原数据解析压缩为波形数据，建立适合波形查看器的数据结构，排序建立数据索引。
- gRPC 处理协程：负责处理 gRPC 请求，与 Scaleda 通信。
- 图像传输协程：负责启动服务器，将渲染的波形图像传输给 Scaleda。

由于执行后端可能不同，于是协程的实际实现也可能不同，既可能是在同一个线程中实现，也可能是在不同的线程中实现，所以协程之间完全使用消息队列的方式来进行通信。

Rvcd 使用 egui 作为用户界面库，egui 是一个为立即模式 GUI 设计的 Rust 库，它具有简单易用、高性能、跨平台等特点，非常适合用于波形查看器这种应用。立即模式 GUI 是一种 GUI 编程模式，它不需要保存 GUI 的控件，而是在每一帧都重新绘制整个 GUI。这种模式接近于 React 一类的前端框架设计思想，将程序的所有状态保存为一个结构体，从而减少了状态管理的复杂性。

egui 本身只是一个 GUI 库，它并不提供窗口管理、事件处理等功能，所以 Rvcd 还使用了 eframe 作为渲染器，具体使用的渲染后端支持 OpenGL、WebGL 等，跨平台特性优越。

3.3 波形文件解析

波形文件解析是 Rvcd 的核心功能之一，目前仅支持 VCD 格式的波形。它负责将 VCD 格式的波形文件解析为波形数据，并建立适合波形查看器的数据结构，排序和建立数据索引。

VCD（Value Change Dump）格式的波形文件是一种文本文件，它一般会在文件的头部位置保存创建时间、仿真时间信息、仿真器参数等信息，然后声明信号的类型、名称、宽度等，最后保存信号的值变化信息。

VCD 文件中信号的声明一般类似这样：

```
$var reg 32 "a commits_0_data [31:0] $end
```

其中 `reg` 表示信号的类型，`32` 表示信号的宽度，`commits_0_data` 表示信号的名称，`[31:0]` 表示信号的位，而中间的 `"a` 则为信号分配了一个唯一的 ID，用于后续的值变化信息记录。

其实际数据记录方式与其 Value Change Dump 的名字一样，即只记录信号值的变化，如 `#1000` 表示仿真时间为 1000 时刻，`b1010 "a` 表示在当前时刻下，ID 为 `"a` 的信号的值变化为 `b1010`。

3.3.1 数据结构设计

VCD 设计为以文本存储，这样方便人类阅读，但是对于机器来说，这种格式并不是最优的，因为它的数据量很大，而且数据的读取和解析都是串行的，所以 Rvcd 在解析 VCD 文件时，会将其解析为二进制格式，并建立适合波形查看器的数据结构。

然而，将 VCD 文件解析为二进制格式也面临一些挑战。针对这些挑战，Rvcd 采取了对应的优化策略。

VCD 文件中，一个信号的位的可能的长度是没有限制的，但是在普通的计算机中，单个数据的长度是有限的，使用普通的数据结构无法直接存储。同时，改变的数据不一定是可取值的，可能在某一时刻某些位变成了 X、Z 等值，这些值也需要存储。所以，Rvcd 采用了一种灵活的方式来存储这种数据：

```
#[derive(Serialize, Deserialize, Debug, Clone)]
pub enum WaveDataValue {
    /// when vec empty, invalid
    Comp(Vec<u8>),
    Raw(Vec<WireValue>),
}
```

每个 `WaveDataValue` 表示一次变化数据的记录，其中 `Comp` 表示压缩后的数据，`Raw` 表示原始数据。在渲染波形时，两种数据都会被渲染，而数据在加载时，会根据当前是否能够合法压缩数据来选择是否压缩。此处的数据压缩指的是将原本的基于文本的数据放入二进制存储从而减少存储空间，而不是对数据进行压缩算法的压缩。

`Comp` 内的 `Vec<u8>` 存储的是 `BigUInt` 的二进制存储数据。`BigUInt` 是 Rust crate 中的一个大数运算库，支持任意长度的大数运算，非常适合用于存储波形数据。

此处使用一个大小为 196.4MiB 大小的 VCD 文件 `testbench2.vcd` 进行测试，Rvcd 与 GTKWave 同时打开并将所有波形数据加载到窗口中，Rvcd 的内存占用为 1.2GiB，而 GTKWave 的内存占用为 1.1GiB。可以看出，Rvcd 的内存占用与 GTKWave 相比基本一致。

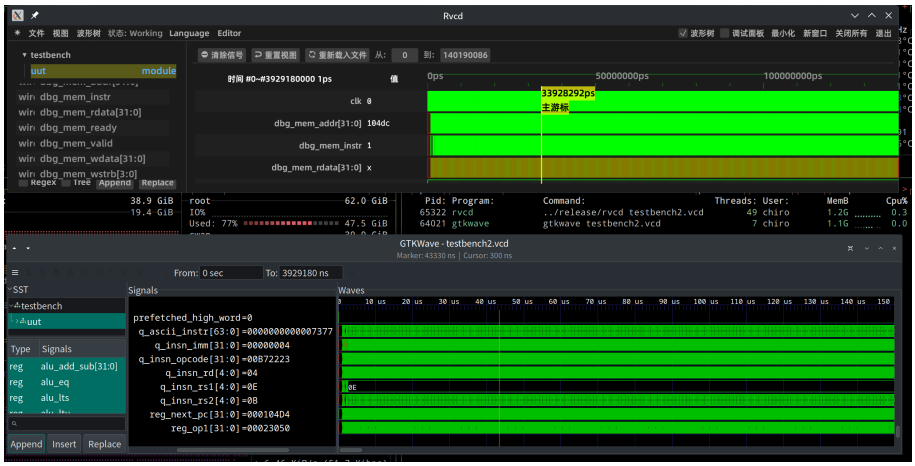


图 3-1　Rvcd 和 GTKWave 内存占用对比

除了内存占用外，Rvcd 对 VCD 文件的解析速度也进行过对应的优化。此处简单测试了 Rvcd 和 GTKWave 对同一个大小为 221MiB 的 VCD 文件的解析速度，当程序对文件解析完毕后立刻退出，结果如下：

```
$ time gtkwave ./chipyard-new/generators/picorv/src/main/resources/vsrc/picorv32/
    dhrystone/testbench.vcd

GTKWave Analyzer v3.3.118 (w)1999-2023 BSI

[0] start time.
[3929180000] end time.
WM Destroy
gtkwave  1.45s user 0.11s system 73% cpu 2.118 total
$ time ~/programs/rvcd/target/release/rvcd ./chipyard-new/generators/picorv/src/main/
    resources/vsrc/picorv32/dhrystone/testbench.vcd
2024-05-07T15:24:21.552734Z INFO rvcd::app: init rvcd app
// ...
~/programs/rvcd/target/release/rvcd 2.39s user 0.91s system 112% cpu 3.197 total
```

由于 Rvcd 的 VCD 解析还是相对简单的，在解析速度上比 GTKWave 稍慢。

3.3.2 数据索引设计

由于 VCD 中的信号逻辑结构是树状结构，所以 Rvcd 在解析 VCD 文件头时，会将其解析为 `Tree<WaveTreeNode>`。由于需要保证树的子树之间的顺序性，所以这里并没有使用类似 `HashMap` 这样的无序数据结构，而是使用了 Crate `tree` 中的 `Tree` 来存储子树。`WaveTreeNode` 结构体如下：

```rust
#[derive(Serialize, Clone, Default, Debug, PartialEq)]
pub enum WaveTreeNode {
```

```
    #[default]
    WaveRoot,
    WaveScope(WaveScopeInfo),
    WaveVar(WaveSignalInfo),
}
```

WaveScope 是波形数据树的树枝节点，而 WaveVar 是波形数据树的叶子节点。叶子节点中存储了信号的名称、类型、宽度等信息，并且记录一个 u64 类型的 ID，用于在实际存储数据的波形树中索引波形数据。

在存储波形数据的数据结构 Wave 中，数据保存在 HashMap<u64, Vec<WaveDataItem>> 中，其中 HashMap 的索引 u64 是读取 VCD 文件时生成的数字格式的 ID。

```
/// loaded wave data in memory
#[derive(Clone)]
pub struct Wave {
    pub info: WaveInfo,
    pub data: HashMap<u64, Vec<WaveDataItem>>,
}

/// item struct in data list
#[derive(Default, Serialize, Deserialize, Clone, Debug)]
pub struct WaveDataItem {
    // pub id: u64,
    pub value: WaveDataValue,
    pub timestamp: u64,
}
```

选择将 Scope 相关数据和 Var 相关数据分开存储的原因是，Scope 相关数据是树状结构，而 Var 相关数据能以 ID 为索引直接存储为 HashMap，这样能够在保留 Scope 信息的情况下，更好地利用 Rust 的数据结构特性，提高数据的访问效率。

在不同的波形之间查找的时候，使用的是 HashMap 中由 ID 建立的索引，而在同一个波形中查找某时刻数据时，使用的是二分查找。在将解析的数据插入到 HashMap 时，会先将数据按照时间戳排序，以保证数据的有序性。由于二分查找的时间复杂度为 $O(\log n)$，所以在查找数据时，仍然能够保证较高的效率。

Rvcd 对波形数据的解析、存储进行了一些设计，不过仍有改进空间，例如可以将解析任务分配给多个核心、使用新的检索算法等[19]。

3.4 波形数据渲染

波形数据渲染是 Rvcd 的另一个核心功能，它负责将解析的波形数据渲染为波形图像以向用户正确地展示出来，并支持波形的缩放、平移、选择、标记等操作，同时还需支持游标等用户交互功能。

由于记录的数据仍然是“数据的变化情况”，所以要将其与“当前时刻的数据的值”对应起来，仍然需要比较复杂的处理流程。

3.4.1 基础波形数据渲染

在用于保存当前波形视图的 `WaveView` 结构体中，保存了当前视图的范围信息 `range: (f64, f64)`，表示当前视图中查看的数据在整个波形数据中以 VCD 文件规定的最小精度为单位的范围。当用户进行缩放、平移等操作时，会改变这个范围，从而改变当前视图的显示。这个范围是浮点数，从而可以实现更加精细的缩放和平移操作，而在数据寻址时，会将其转换为整数，以便于在数据中查找。

`WaveView` 结构体中有两个函数，负责将 VCD 中的时间单位与屏幕上的像素位置互相转换：

```
/// Convert paint pos to wave position
/// * `x`: x position to wave panel
pub fn x_to_pos(&self, x: f32) -> u64 {
    ((x as f64 * (self.range.1 - self.range.0) / self.wave_width as
       f64) + self.range.0) as u64
}
/// Convert wave position to paint pos
/// * `pos`: wave position defined in wave info
pub fn pos_to_x(&self, pos: u64) -> f32 {
    ((pos as f64 - self.range.0) * self.wave_width as f64 / (self.
       range.1 - self.range.0))
       as f32
}
```

`WaveView.ui_signal(...)` 进行 UI 渲染时，首先会根据当前视图范围的最左侧，找到当前视图范围内的第一个数据，再从这一个数据的前一个数据开始绘图，直到当前视图范围内的最后一个数据。这样，就能够保证在当前视图范围内的数据都能够被正确地绘制出来。

绘图的过程是，首先在绘图前保存上次绘制的数据点的波形数据索引位置，然后联系当下的数据点的波形数据索引位置进行判断，当数据相同时且非第一次绘制，则忽略此次绘制。当数据不同时，根据数据的变化情况，绘制出相应的波形。

当需要绘制的数据宽度为 1 且有效时，绘制的形状应该为一条平直直线和可选的一条竖直直线，根据数据是高电平还是低电平，绘制出相应的直线；当需要绘制的数据宽度大于 1 时，绘制的形状应该为一个矩形，根据数据是高电平还是低电平，绘制出相应的矩形。而当数据为 X、Z 等无效数据时，将会选择红色进行绘制。

总之，波形数据渲染的过程是一个非常复杂的过程，需要根据数据的变化情况、数据的宽度、数据的有效性等多个因素进行判断，从而绘制出正确的波形图像。

3.4.2 波形数据渲染优化

除了需要保证绘制的波形是正确的，还需要保证绘制的性能。当当前视图范围内的数据量非常大时，绘制的性能就会成为一个问题。为了保证绘制的性能，Rvcd 采用了一些优化策略：

若在一段像素长度内需要连续绘制数据，则可以将这段数据合并为一个矩形，从而减少绘制的次数；若当前帧需要计算的帧与上一帧相同，则可以直接使用上一帧的绘制结果，从而进一步减少绘制的次数。

通过这些优化策略，Rvcd 能够在保证绘制正确性的同时，保证绘制的性能。即使打开上 GiB 的波形文件，Rvcd 的帧生成时间也在 10ms 以内，能够轻松跑满 120fps 高刷新率显示器。

3.4.3 视图移动和游标功能

在波形数据渲染的过程中，用户可能需要对波形进行缩放、平移、选择、标记等操作，这就需要在波形数据渲染的基础上，增加一些用户交互功能。

在 Rvcd 中，用户可以使用很多方式方便地移动视图和操作游标，包括：

- 使用鼠标滚轮滚动，可以上下移动视图。
- 使用带副滚轮的鼠标移动，或是按住 Shift 键的同时滚动鼠标滚轮，或是双指滑动触摸板，可以左右移动视图。

- 按住 Ctrl 键的同时滚动鼠标滚轮，或是双指捏合/张开触摸板，可以缩放视图。
- 使用鼠标左键点击与拖动，可以移动主游标。
- 鼠标左键在时间条上拖动，可以移动其他的游标。
- 使用鼠标右键拖动，选定新的视图范围。
- 使用鼠标中键拖动，可以直接水平和垂直移动视图。

相比于传统的波形查看器，Rvcd 的用户交互功能更加丰富，更加方便，更加易用。例如，GTKWave 的鼠标滚轮事件几乎没有得到正确的处理，无法直接上下滚动视图，也无法处理横向的鼠标滚动事件；处理这些鼠标事件也几乎都是导致离散的视图移动，而不是连续的视图移动。

游标是 Rvcd 视图中的标记，标记在波形中的某个时刻，用于对特殊时间进行标记，也可以用于时间的测量。有关游标的操作除了上述几种，还有：

- 在时间条上右击，可以添加或删除游标。
- 在游标上右击，可以设置是否测量其到其他游标之间的时间距离。
- 在移动主游标时，会自动测量主游标移动前后距离。

3.4.4 内部多窗口模式

由于立即式 GUI 的特性，Rvcd 的 GUI 逻辑相当简单，于是可以将文件和波形相关的状态抽取出来，单独保存在一个列表中，从而实现多窗口模式。窗口操作栏由最大化的窗口控制，可以最大化、最小化、关闭窗口，也可以将显示内容与其他窗口直接替换。利用多窗口模式，Rvcd 可以方便地进行波形数据对比、波形视图切换等操作，提高了用户的使用体验。

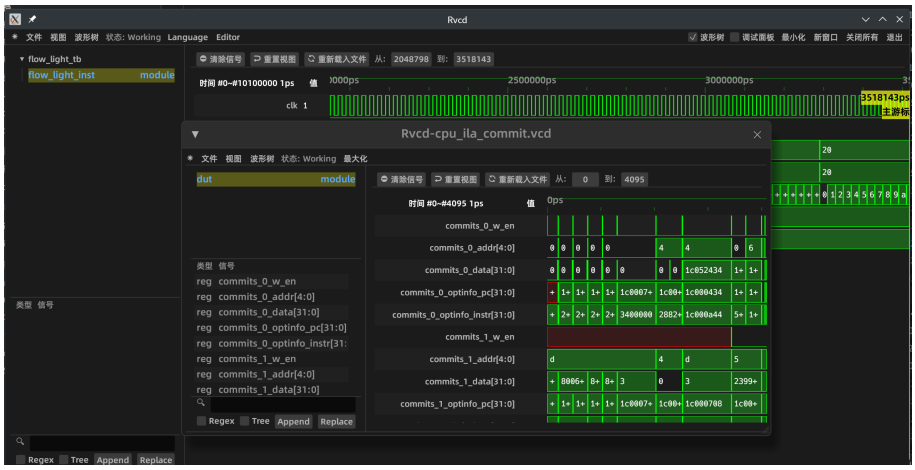


图 3-2　Rvcd 多窗口模式

3.4.5 其他渲染结构和细节

为了提升 Rvcd 的实用程度，Rvcd 还支持了一些其他的渲染结构和细节。

左下角是波形树窗口，用于显示当前 Scope 下的所有信号，用户可以通过点击信号来选择显示的波形，还提供了按文本和正则进行波形搜索和批量添加的功能，以及可以对当前视图的波形进行附加、替换等。

左上角是波形结构窗口，显示当前波形的模块结构。为了特殊场景需要，它也可以选择显示其中的树形波形。当双击其中的结构时，如果对应的操作是将结构中的波形全部添加到视图中，则当前结构会变成亮黄色背景，以提醒当前的波形来源。右击波形结构，可以选择添加内部波形或者递归地添加所有波形等。

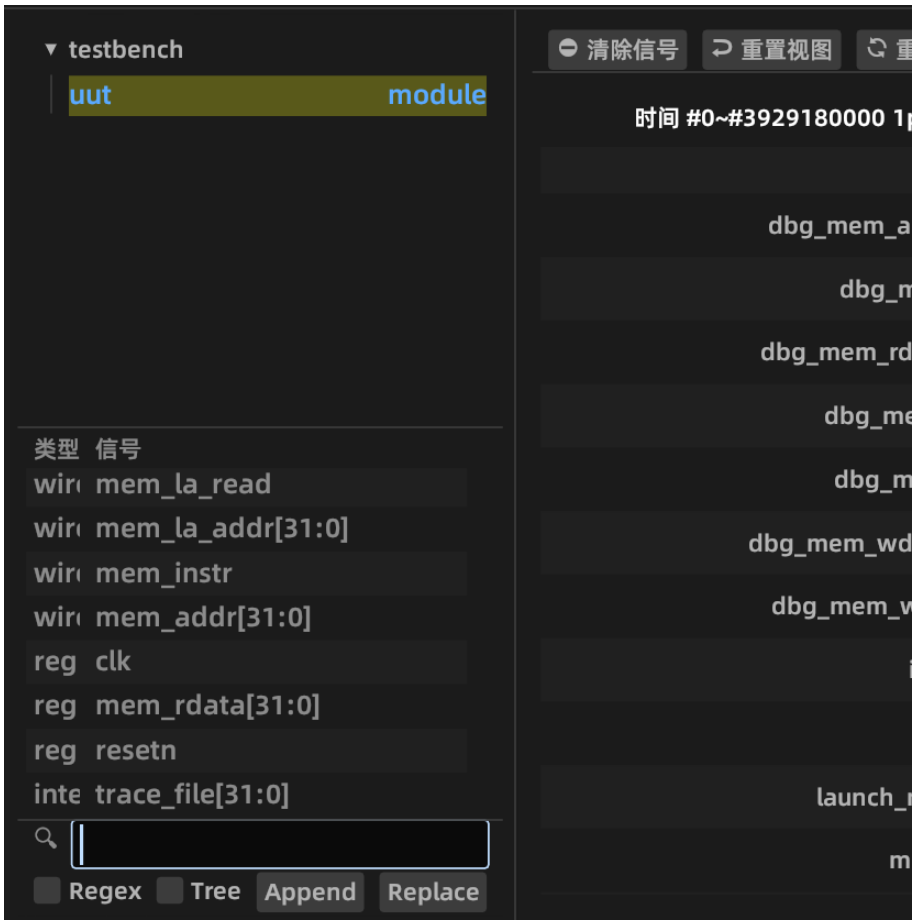


图 3-3　Rvcd 波形树窗口

主波形视图窗口也有许多可以动态配置的选项。在波形名称上右击，可以选择波形的显示高度、显示颜色、显示模式、数据格式等。在左上角的视图设置中，还能设置各个波形显示的默认值。波形的显示模式可以选择数值、模拟拟合、模拟台阶等，以适应不同种类的波形数据。

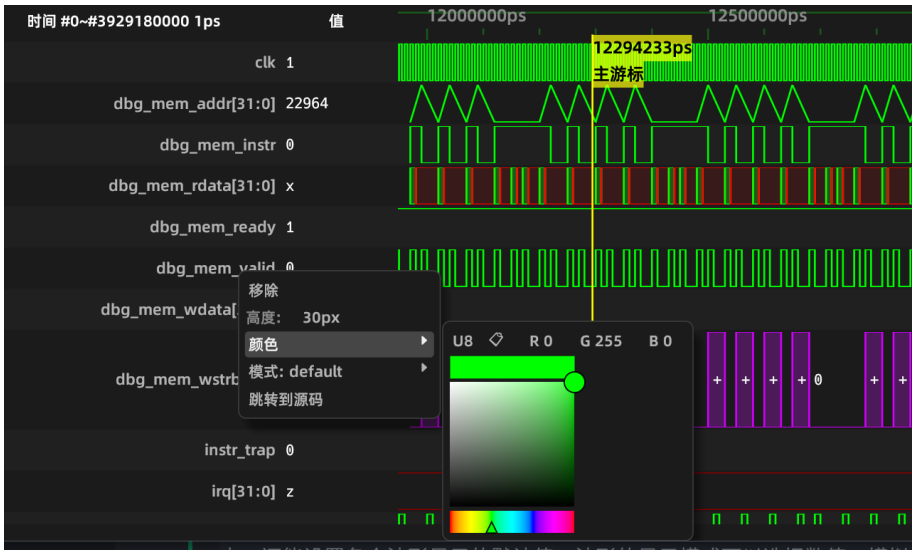


图 3-4　Rvcd 波形显示设置

3.5 HDL 代码解析

由于 ANTLR4 是跨平台的，相同的语法文件可以生成多种不同的目标编程语言结构，所以 Rvcd 同样选择了使用 ANTLR4 来解析 HDL 代码，不仅能够复用为 Scaleda 编写的各种语法文件，还能增强 Rvcd 作为独立波形查看器的功能。

不过 Rvcd 作为波形查看器，代码查看和编辑不应作为其主要功能，于是 Rvcd 虽然有 HDL 代码的解析、显示和编辑功能，但是仅支持了最基础的语法高亮显示和文本编辑，其代码解析功能主要用于波形和源代码互相索引。

3.6 与 Scaleda 集成

在前面的章节已经大致展示了 Rvcd + Scaleda 的集成效果，这里再详细介绍一下在 Scaleda 中集成 Rvcd 的过程。

3.6.1 Java2D 中的 OpenGL 渲染

因为使用 egui 和 eframe 的 Rvcd 在渲染时默认使用的是 OpenGL，而 Scaleda 使用的是 Java2D 和 AWT，在项目初期探索过直接使用 Java2D 进行 OpenGL 渲染的方式。

首先，将 Rvcd 编译为一个 Library，然后在 Scaleda 中使用 JNA 或 JNI 调用 Rvcd 的 Library，向其中传入特定的渲染所需函数，然后启动 Rvcd 的渲染器。理论上，这样就可以在 Scaleda 中直接使用 OpenGL 渲染波形。

但是，由于 OpenGL 等图形库的特殊性，这种方式在实际操作中遇到了很多

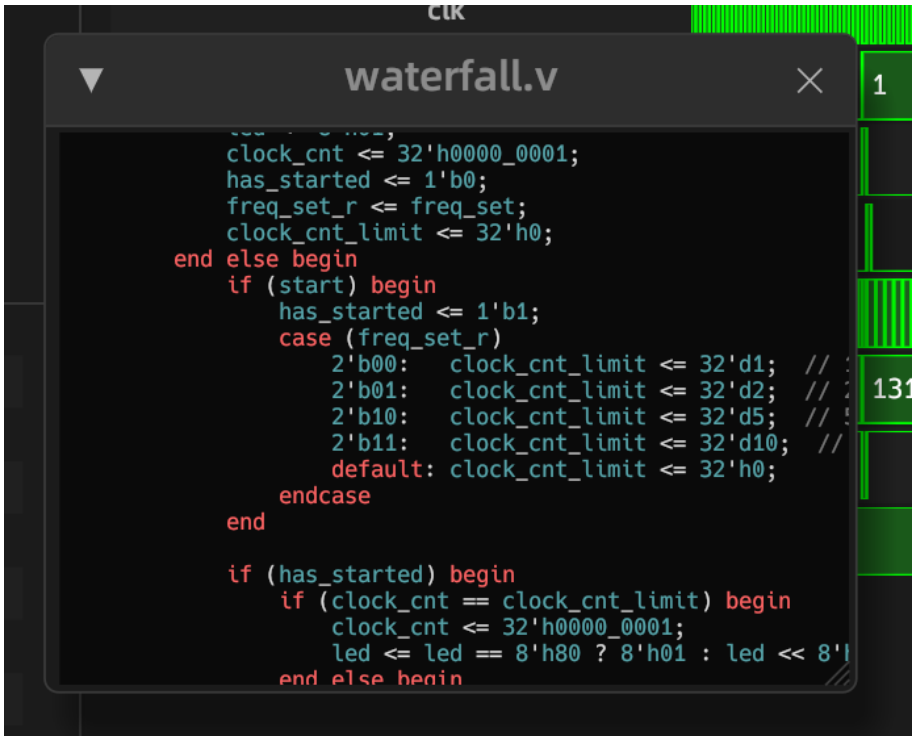


图 3-5　Rvcd Verilog 高亮显示

问题，例如 OpenGL 的线程安全性、OpenGL 的资源管理、OpenGL 的跨平台性等问题，导致这种方式的实现非常困难。

3.6.2 使用 WebAssembly 进行跨平台集成

在前文中已经提到，Rvcd 除了能够在 Linux、Windows、macOS 等多种操作系统上运行，还能够编译到 WebAssembly 并运行在浏览器中。所以，Rvcd 同时还可以使用嵌入 WebAssembly 的方式来集成到 Scaleda 的浏览器窗口中。

Rvcd 能够使用 `trunk` 工具将其编译为 WebAssembly，禁用后台线程、gRPC、FrameBuffer 服务器等功能，其功能仅限于波形查看，如图 3-6 所示。

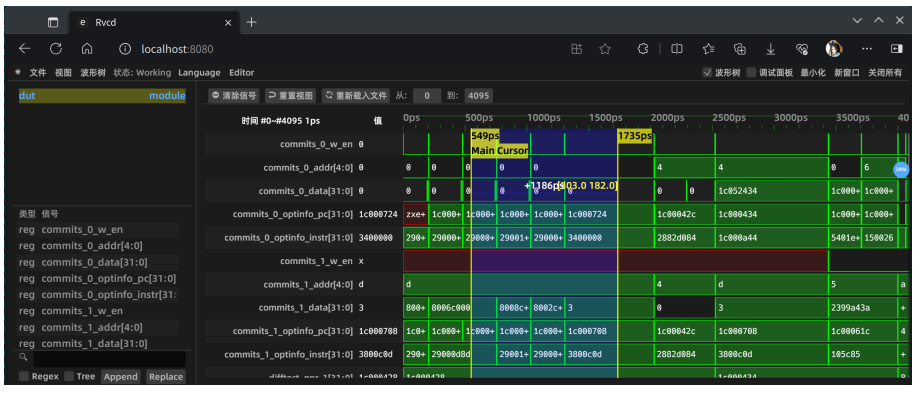


图 3-6　在 Edge 浏览器中使用 Rvcd WebAssembly 版本

经过一些尝试，Rvcd 能够在 IDEA 的浏览器窗口中打开并运行。但是，由于 WebAssembly 运行时相当于在一个沙盒中运行，所以在 WebAssembly 中无法直接访问浏览器的 DOM、无法直接访问浏览器的文件系统等，这就导致了 Rvcd 无法直接访问 Scaleda 的文件系统，也无法启动 gRPC 服务器以与 Scaleda 通信。于是此方式也被放弃。

3.6.3 使用渲染帧传输进行集成

由于前两种方式都无法实现 Rvcd 和 Scaleda 的集成，于是最终选择了使用渲染帧传输的方式进行集成。

因为 Rvcd 使用 egui 这样的立即式 GUI 库，所以 Rvcd 的渲染是非常快速的，于是渲染出的图像可以通过 gRPC、TCP 等多种途径到达 Scaleda。在 Scaleda 中，只需要接收到 Rvcd 发来的图像帧，然后将其显示在 Scaleda 的窗口中，最后再将用户的操作传递给 Rvcd 即可。

在 Rvcd 侧，如果遇到在 Rpc 的请求消息队列中有遇到了帧数据请求，则会向 egui 系统发送 `ScreenShot` 请求。egui 在当前帧渲染完后便不会抛弃当前帧，而是将当前帧的图像数据保存下来。在处理下一个帧时，会将上一个帧的图像数据发送给 Scaleda。

发送图像时，原图像信息是以 RGBA 格式的字节数组存储的，数据相对较大，对传输并不友好。于是 Rvcd 在发送图像之前，会首先将图像数据转换压缩，从每个像素 32bit 转换到 RGB656 格式，让每个像素仅占用 16bit，从而减少传输的数据量。

Rvcd 的 FrameBuffer 服务器支持三种传输方式：gRPC、TCP 和 UnixSocket。gRPC 速度和延迟最差，TCP 好一些，UnixSocket 最好。在 Scaleda 中，使用 UnixSocket 作为 FrameBuffer 服务器的传输方式，能够保证最低的延迟和最快的速度。

Scaleda 到 Rvcd 的消息也是通过 FrameBuffer 服务器进行传输的，复用了 FrameBuffer 服务器的传输通道。

在 Scaleda 中通过 UnixSocket 显示 Rvcd 的波形图像，可以达到 60fps 的帧率，延迟在 100ms 以内，用户体验相对稳定。效果如图 2-20 所示。

3.7 国际化支持

与 Scaleda 一样，Rvcd 也支持国际化，支持多种语言的切换。Rvcd 使用了 Rust 的 `rust-i18n` 库来实现国际化支持，目前已经支持了简体中文和英文。

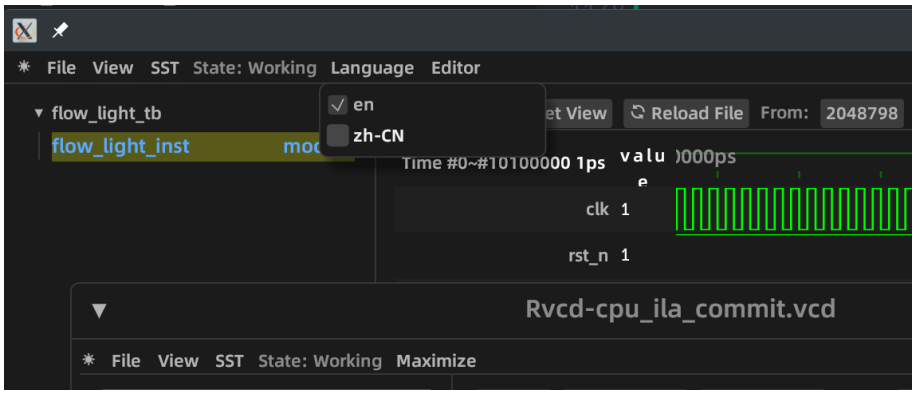


图 3-7　Rvcd 显示语言切换效果

3.8 本章小结

本章主要介绍了 Rvcd 波形查看器的设计思路、设计细节与开发过程。

Rvcd 作为一款全新开发的波形查看器，具有高性能、高可靠性、高可扩展性和可移植性等特点。其不仅能作为一个独立的波形查看器使用，还能够与 Scaleda 集成，支持在 Scaleda 中直接查看波形，并调用相关互动功能。

在 Rvcd 的编写过程中，参考了许多现有的波形查看器，如 GTKWave、Surfer[20] 等，从中吸取了许多经验，同时也在性能优化、用户体验等方面进行了许多创新性的工作。

第 4 章 使用本项目软件进行设计实践

本章将简要介绍如何使用本项目的软件进行数字电路设计实践。

4.1 简单流水灯设计

流水灯是一种常见的数字电路设计实践，本节将从最基础的流水灯电路出发，逐步介绍使用本项目软件进行设计实践的过程。

4.1.1 流水灯电路设计

在本示例设计中，可以使用计数器和移位寄存器来实现一个简单的流水灯电路。初步 Verilog 设计如下：

```
module flow_light
#(
    parameter WIDTH = 8,
    parameter PERIOD = 1000000
) (
    input wire clk,
    input wire rst_n,
    output wire [WIDTH-1:0] led
);
    reg [WIDTH:0] led_r;
    assign led = led_r[WIDTH-1:0];
    reg [$clog2(PERIOD)-1:0] cnt;
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            led_r <= 8'b00000001;
            cnt <= 0;
        end
        else begin
            if (cnt == PERIOD - 1) begin
                led_r <= (led_r << 1) | led_r[WIDTH];
                cnt <= 0;
```

```
            end else begin
                cnt <= cnt + 1;
            end
        end
    end
endmodule
```

在此模块中，本设计使用了一个计数器 cnt 来控制流水灯的移位速度，使用一个移位寄存器 led_r 来存储当前流水灯的状态。在每个时钟周期中，如果计数器 cnt 达到了设定的周期 PERIOD，则将流水灯状态左移一位，并将最低位的值赋给最高位，从而实现流水灯的效果，每当计数器 cnt 达到 PERIOD 时，流水灯状态循环左移一位。

使用者可以首先使用 Scaleda 创建一个新的项目 flow_light（2-7），并在项目中创建一个新的源文件 flow_light.v，将上述代码粘贴到源文件中。从图 4-1 可以看到，代码被 Scaleda 成功地解析，在结构窗口中显示了当前模块的结构，包括模块、端口、参数、驱动等信息。

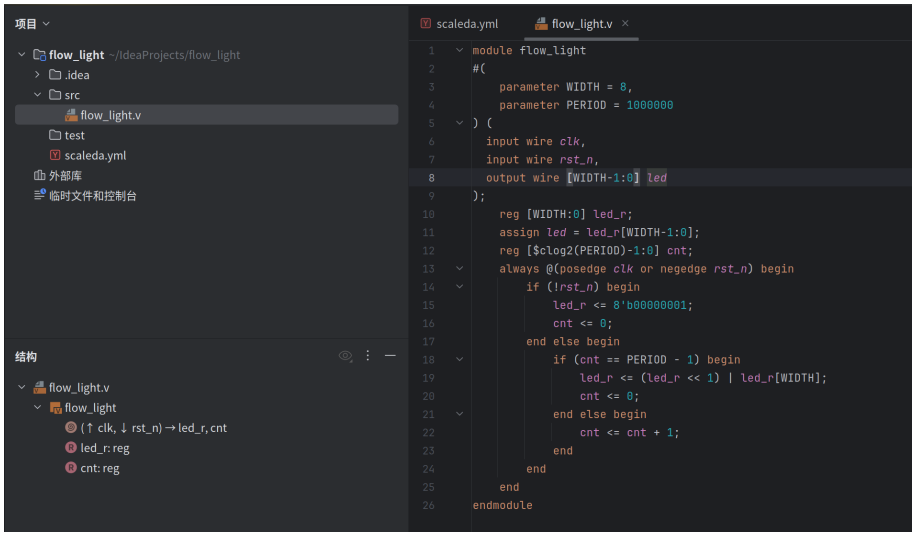


图 4-1　流水灯代码

成功创建项目后，可以发现右下角弹出窗口提示还未添加工具链（4-2）。点击“前往设置”以使用 GUI 设置可能用到的工具链。

在输入或选择工具链路径后，Scaleda 将调用工具链验证功能，以保证工具链设置正确。此处添加了可能用到的以下几个工具链。

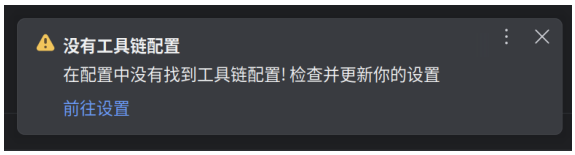


图 4-2　工具链添加提示

(a) 添加工具链：Icarus Verilog

(b) 添加工具链：Xilinx Vivado

(c) 添加工具链：Yosys

(d) 添加工具链：Bluespec Compiler

图 4-3　添加本地工具链

4.1.2 使用 Icarus Verilog 进行快速仿真

Icarus Verilog 由于其轻量化、开源等特点，是一个非常适合用于对较小的设计进行快速仿真验证的工具。

首先为 `flow_light` 模块编写一个简单的测试文件 `flow_light_tb.v`，放在 `tests` 目录下。可以在 Scaleda 的文件树中右键点击 `tests` 目录，选择“新建文件”→“新建 Verilog 文件”来创建一个新的测试文件。测试文件的内容如下：

```
module flow_light_tb;
    reg clk;
    reg rst_n;
    wire [7:0] led;
    flow_light #(
        .WIDTH(8),
        .PERIOD(32)
    ) flow_light_inst (
        .clk(clk),
        .rst_n(rst_n),
        .led(led)
```

```
    );
    initial begin
        clk = 0;
        rst_n = 0;
        #100;
        rst_n = 1;
        repeat (1000) begin
            #10 clk = ~clk;
        end
        $finish;
    end
endmodule
```

在“Scaleda Tasks”侧边栏中，点击项目编辑器，打开项目编辑界面。首先点击项目根，添加一个“Icarus Verilog”类型的目标（4-4），然后点击目标，再添加一个仿真任务（4-5）。在任务编辑界面，可以直接点击选择顶层模块，然后在弹出的列表中选择 `flow_light_tb`。设置任务名称，再将任务类型选择为 `Simulation`，即可完成任务的设置。此时项目中的 `scaleda.yml` 被自动更新，侧边栏的任务树也添加了新的任务如图 4-6 所示。

图 4-4　添加 Icarus Verilog 目标

双击或按下回车或点击左上角的执行任务，将会在 IDEA 的 Run Configuration 处自动生成一个新的配置，并开始仿真。此时在运行窗口中可以看到仿真的输出信息（4-7）。再打开左侧的“Scaleda”边栏，同时可以查看仿真的输出消息列表、仿真输出文本信息，以及启动了的波形查看器窗口（4-8）。在波形查看器中添加感兴趣的信号，并设置相应的展示参数。仿真时是否生成波形，以及仿真成功后是

图 4-5　添加 Icarus Verilog 任务

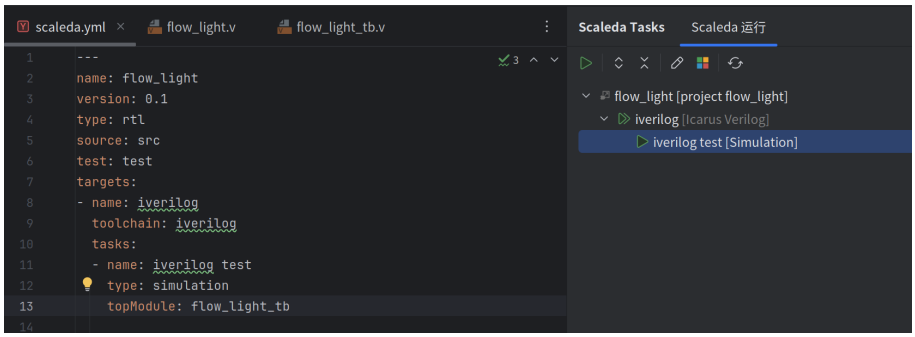


图 4-6　侧边栏任务树

图 4-7　仿真输出

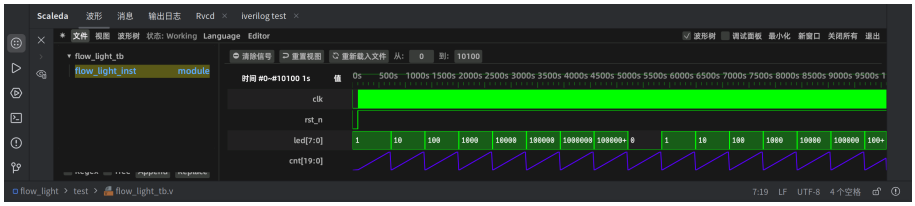


图 4-8　波形查看器查看仿真波形

否自动打开、自动重新加载波形文件，都可以在 IDEA 的设置中进行配置，存储在全局配置中。

经过 Icarus Verilog 仿真验证，可以确定本设计中的流水灯电路设计是正确的。接下来将使用 Xilinx Vivado 进行之后的 FPGA 设计流程。

4.1.3 使用 Xilinx Vivado 进行 FPGA 设计

首先，使用 Vivado 对刚才的设计进行仿真。与添加 Icarus Verilog 的仿真任务类似，需要首先添加一个 Xilinx Vivado 目标，然后添加综合任务。在目标编辑界面中，输入“xc7z010clg400-2”作为本次设计的 FPGA 型号（4-9），然后添加对应的仿真任务，顶层模块仍然填写 `flow_light_tb`。

图 4-9　添加 Xilinx Vivado 目标

运行此 Vivado 仿真任务，仿真成功后，可以在左侧的“Scaleda”边栏中查看仿真输出信息、波形等（4-10、4-11）。如果仿真失败，则可以在仿真输出信息中查看详细的错误信息（4-12），以便进行调试。

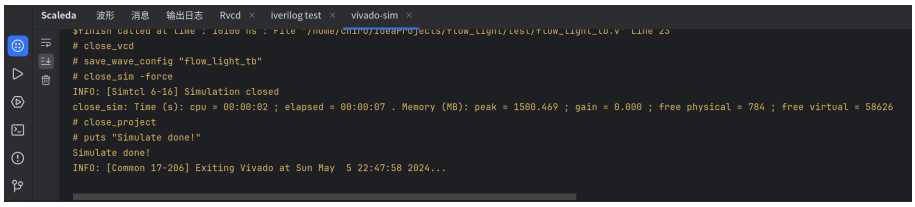


图 4-10　Vivado 仿真输出

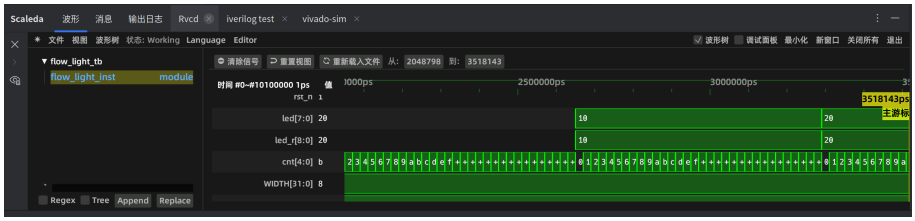


图 4-11　Vivado 波形查看器查看仿真波形

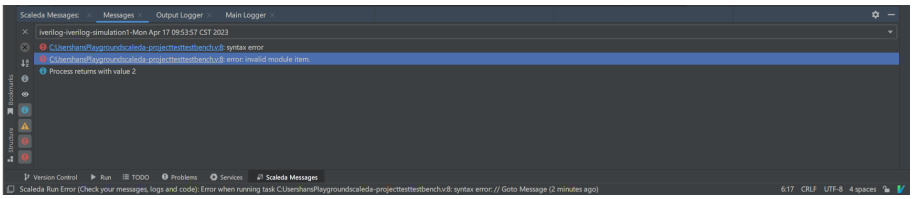


图 4-12　Vivado 仿真输出错误消息

接下来，用户能够通过 Scaleda 使用 Vivado 进行综合、布局布线、生成比特流文件等操作，以便将设计下载到 FPGA 开发板上进行验证。

由于本设计中使用的是一个简单的 Zynq 7010 FPGA，开发板是一块廉价的 EBAZ4205 矿板，外观如图 4-13，并没有外置晶振，所以本设计还需要在 Vivado 中添加额外的 PS 端 IP 核，用于生成时钟信号。

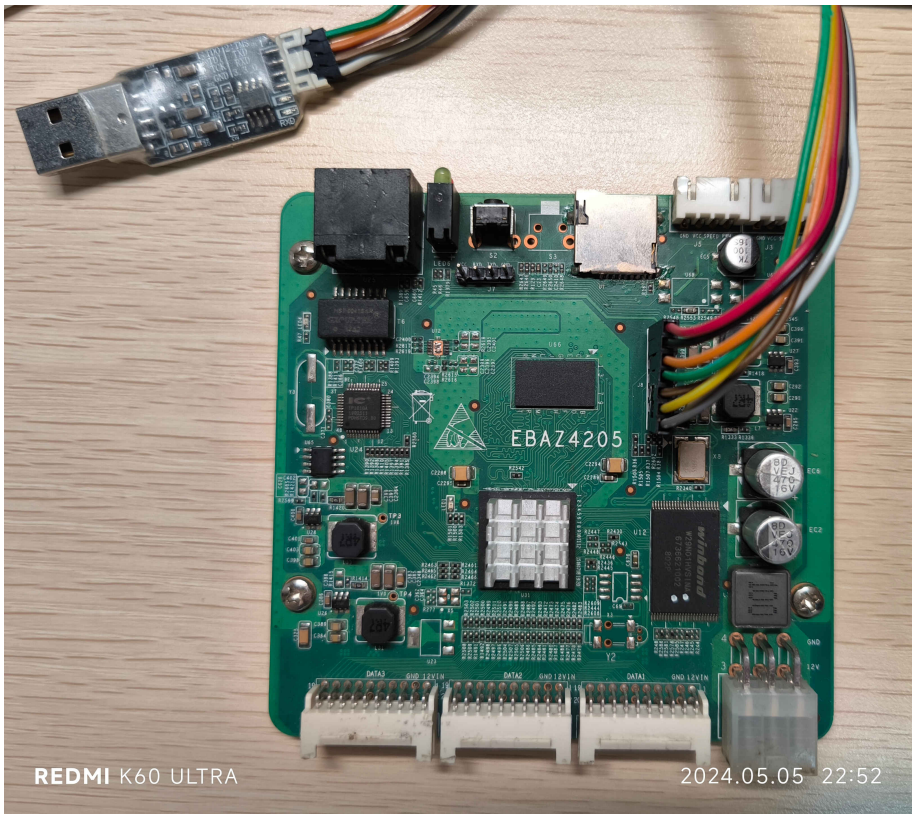


图 4-13　EBAZ4205 开发板

由于此 ZYNQ Processing System IP 核是 Vivado 的专有 IP 核，且 Scaleda 目前暂不支持这种复杂的 IP 核的设计，所以本设计要求用文件的方式将此 IP 核导入到 Scaleda 中。

首先，打开 Vivado 的 GUI，在其中新建一个项目，添加 “ZYNQ Processing System” IP 核，然后选择 “Generate Output Product”，导出 `processing_system7_0.xci` 文件，将此文件复制到当前项目的 `ip` 目录下，然后在 `scaleda.yml` 的 `ipFiles` 中

添加 ip/processing_system7_0.xci。具体目录结构并不是固定的，可以根据实际情况进行调整。

在 flow_light 外层添加一个新的模块 flow_light_top 用于连接 PS 端的时钟信号，然后在 flow_light_top 中实例化 flow_light 和 processing_system7_0，并将二者连接起来。在编辑 flow_light_top 的时候，可以体验 Scaleda 的端口连接检查、自动端口和参数补全等功能，如图 4-14、图 4-15。

图 4-14　端口连接检查

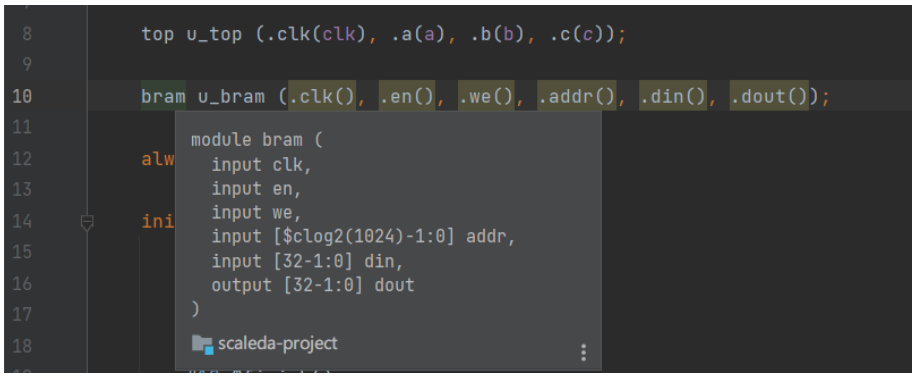


图 4-15　端口自动补全

添加的新的模块 flow_light_top 如图 4-16 所示。其中，processing_system7_0 标红的原因是 Scaleda 在 Vivado 还未开始处理时还无法解析 Vivado 的 IP 核，所以无法显示其结构。当 Vivado 对 IP 进行生成操作后，Scaleda 将自动从 Vivado 生成的文件中解析出 IP 核的结构，并在 Scaleda 中显示。

再次打开项目编辑器，在 Vivado 的目标下添加添加一个“Programming”任务，此任务将完成综合、布局布线、生成比特流文件等一系列操作，并最后将比特流文件下载到 FPGA 开发板上进行验证。

在 constraints/ 目录下添加约束文件 flow_light_top.xdc，用于设置本设计的约束。在项目编辑器的任务编辑设置中添加此约束文件。由于此开发板上只

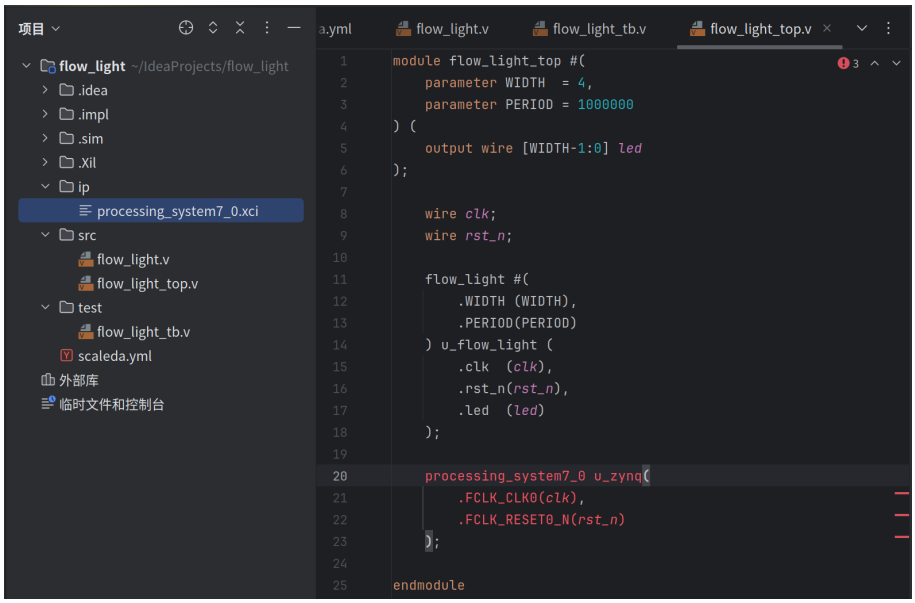


图 4-16　添加 IP 核后的顶层模块设计

有两个可用 LED，于是将 `led[1:0]` 信号连接到这两个 LED 上，其他信号分配任意引脚。

```
set_property -dict { PACKAGE_PIN W13 IOSTANDARD LVCMOS33 } [get_ports { led[0] }]; #
    green led
set_property -dict { PACKAGE_PIN W14 IOSTANDARD LVCMOS33 } [get_ports { led[1] }]; # red
    led
set_property -dict { PACKAGE_PIN M19 IOSTANDARD LVCMOS33 } [get_ports { led[2] }];
set_property -dict { PACKAGE_PIN N20 IOSTANDARD LVCMOS33 } [get_ports { led[3] }];
```

运行此任务，Vivado 完成综合、布局布线、生成比特流文件等操作后，自动将比特流文件下载到 FPGA 开发板上进行验证。如果验证成功，开发板上的 LED 将会按照流水灯的设计进行闪烁。

经过一段时间的等待，任务正常结束，观察到开发板上的 LED 按照设计进行闪烁，如图 4-17 ，说明设计验证成功。

4.2 使用远程工具链

上一节主要演示了如何使用本地工具链进行 FPGA 设计，因为使用的 FPGA 型号为 Xilinx Zynq7010，所以需要首先生成 PS 侧 IP 核，用于生成时钟信号。而在实际使用过程中，用户即使没有本地的 EDA 工具链环境，也可以使用远程工具链进行 FPGA 设计。

采取上一节的流水灯设计为例，用户可以使用远程工具链而非本地的 Vivado 进行 FPGA 设计。

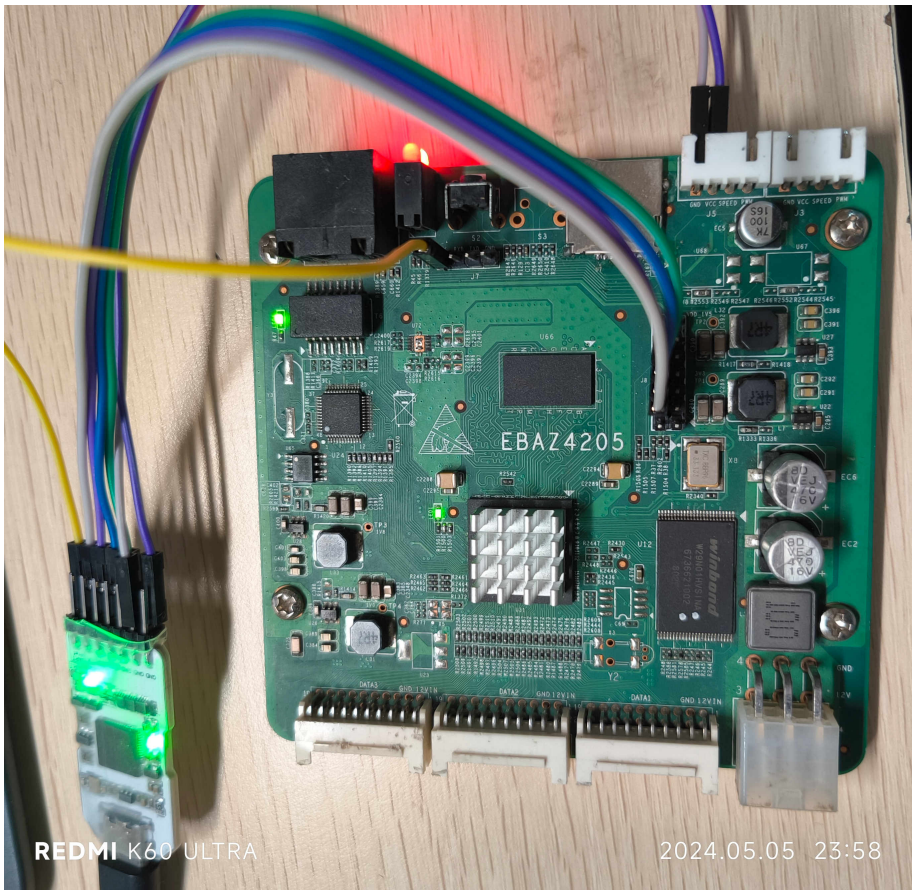


图 4-17　上板验证成功

首先，在 Jvm 环境中启动 `scaleda-kernel.jar`，使用 Scaleda 的命令行功能启动服务器监听。

```
$ scaleda remote serve
07:28:42.076 [main] INFO scaleda-kernel - [ScaledaShellMain.scala:78 main] This is
    Scaleda-Shell, an EDA tool for FPGAs
07:28:42.327 [main] INFO scaleda-kernel - [ScaledaShellMain.scala:105 main] No project
    config detected!
07:28:42.569 [server-main-rpc-server-thread] INFO scaleda-kernel - [RpcPatch.scala:59
    getStartServer] scaleda grpc server serve at port 20051
```

然后在 IDEA 中，打开 Scaleda 的项目，编辑当前 Vivado 的 RunConfiguration，为其设定服务器地址，如图 4-18 所示。设定服务器地址时，Scaleda 将会自动拉取远程服务器上的可用工具链配置等信息。当因为 JWT Token 过期等原因导致连接失败时，Scaleda 会自动提示用户在设置中重新登录，如图 4-19。

在设置好远程工具链后，可以直接运行 Vivado 的 RunConfiguration。此任务将会在远程服务器上运行 Vivado，与在本地运行 Vivado 的行为和输出几乎无异。

Scaleda Server 运行时的一些日志如图 4-20 所示，主要记录连接记录、命令输出和 FUSE 日志等。

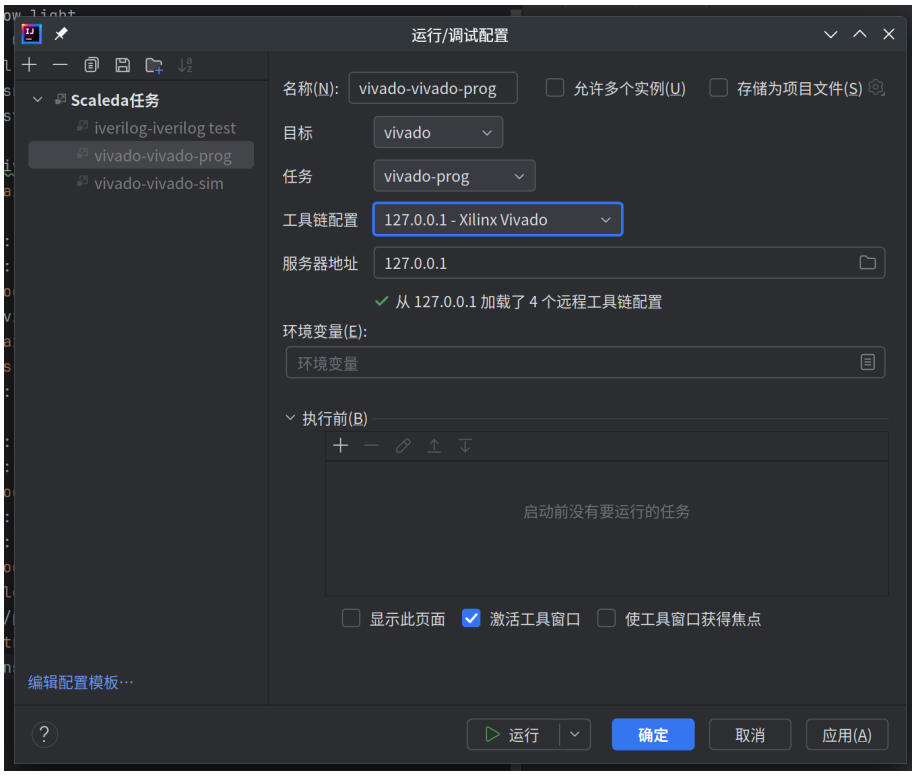


图 4-18　远程工具链设置

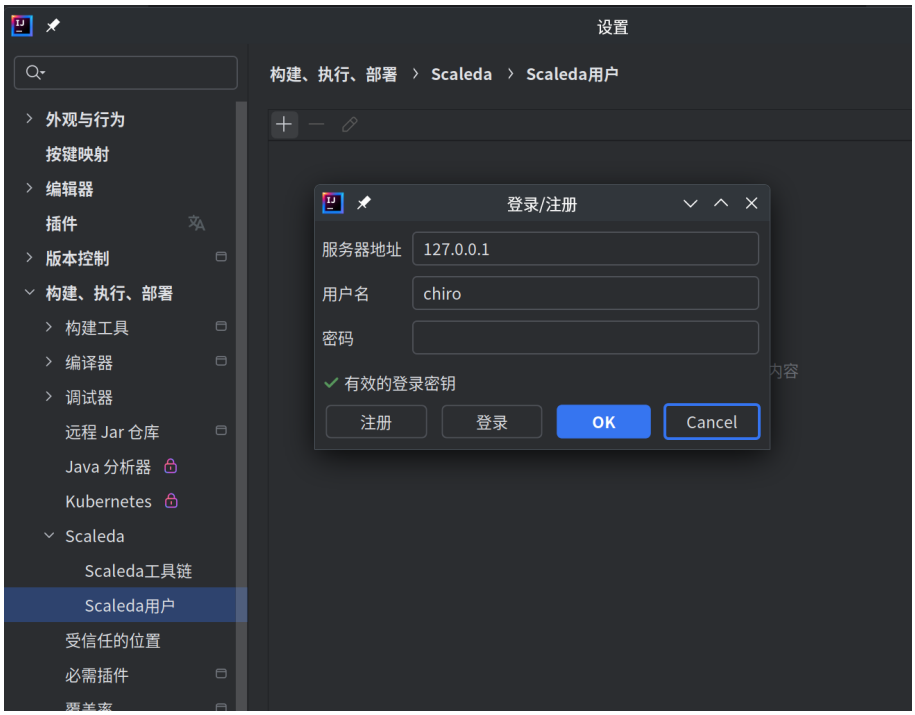


图 4-19　登录界面

图 4-20　服务器日志

4.3 全连接神经网络设计

为了演示本项目软件在更复杂的设计中的应用，本章节将使用 Bluespec Verilog 设计一个简单的全连接神经网络，使用 MNIST 数据集进行测试，在仿真环境中进行验证，并将其部署到 Xilixn FPGA 上。

Bluespec Verilog 是一门高级硬件描述语言，与 Verilog 等一样，被用于 FPGA 或 ASIC 的设计和验证。BSV 于 2003 年被 Bluespec 公司开发，期间是商业收费工具，到 2020 年它的编译器被其所有的公司开源，这才给了外界接触它的机会。

Bluespec Verilog 是一门全新的硬件描述语言，它的语法和 Verilog 有很大的不同。它通过模块、接口、规则、方法等结构，以及复杂的类型系统，提供了一种更加高效、更加抽象的硬件设计方法。BSV 的设计思想是将硬件设计看作是一种并发的、规则驱动的、函数式的编程，而不是传统的顺序执行的过程。

4.3.1 基于 Bluespec Verilog 的全连接神经网络设计

全连接神经网络是一种最简单的神经网络结构，每个神经元都与上一层的所有神经元相连。全连接神经网络的设计是神经网络设计中的基础，也是最简单的一种设计，其计算过程也是最直观的矩阵乘法和激活函数的组合。

本示例项目也是本文作者的一个开源项目，项目地址为 https://github.com/chiro2001/bsv-cnn，其中包含了一个简单的全连接神经网络的设计，以及一个简单的 MNIST 数据集的测试。项目的 HDL 部分使用 Bluespec Verilog 进行设计，并且项目开发时使用 Scaleda 进行 HDL 编写、仿真、综合、布局布线等操作。

此设计中的模型使用 PyTorch 构造，其结构和代码如下。`name` 属性用于后续直接生成 FPGA 综合时的 Block Memory 内的数据。

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from config import *

class FcNet(nn.Module):
    def __init__(self) -> None:
    super().__init__()
    self.name = "fc"
    self.fc1 = nn.Linear(784, 64)
    self.fc2 = nn.Linear(64, 10)

    def forward(self, x):
    x = torch.flatten(x, 1)
    x = self.fc1(x)
    x = F.relu(x)
    x = self.fc2(x)
    output = F.log_softmax(x, dim=1)
    return output
```

通过 torchsummary 工具，得知当前模型为两层全连接层结构，隐藏层大小 64，有 50,890 个参数，大小约为 0.20 MB。设计较小的神经网络，是为了在 FPGA 上进行验证时，能够在有限的资源内完成验证，并在较短的时间内完成综合、布局布线等操作。

```
----------------------------------------------------------------
       Layer (type)           Output Shape      Param #
================================================================
          Linear-1               [-1, 64]       50,240
          Linear-2               [-1, 10]          650
================================================================
Total params: 50,890
Trainable params: 50,890
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.19
Estimated Total Size (MB): 0.20
----------------------------------------------------------------
```

```
Train Epoch: 0 [0/60000 (0%)] Loss: 2.290047
Train Epoch: 0 [6400/60000 (11%)] Loss: 0.845264
Train Epoch: 0 [12800/60000 (21%)] Loss: 0.269843
// ...
Train Epoch: 2 [44800/60000 (75%)] Loss: 0.275238
Train Epoch: 2 [51200/60000 (85%)] Loss: 0.341927
Train Epoch: 2 [57600/60000 (96%)] Loss: 0.132406

Test set: Average loss: 0.2542, Accuracy: 9292/10000 (93%)
```

模型使用 MNIST 数据集进行训练。经过 3 个 epoch 的训练，模型在测试集上的准确率达到了 93%。

模型在处理和计算过程中使用了相对简单的量化方案，以降低处理复杂度，减少资源占用。模型采用 PTQ（Post-training Quantization，训练后量化）方案，量化后数据类型为小数位数占 20 位、符号位占 1 位的 32bit 有符号定点数。

训练后的模型被分别保存为量化以及非量化的 PyTorch 模型文件，并进行了简单的量化误差测试。

```
fc fc1.weight torch.Size([32, 784]) max 1.553686 min -1.8483458 int32 max 4294967287
    int32 min 8 ../data/fc-fc1.weight.hex
restore diff mean -1.0933673e-08 max 9.5320866e-07 min -9.52743e-07
fc fc1.bias torch.Size([32]) max 0.07604257 min -0.16778715 int32 max 4294958541 int32
    min 7307 ../data/fc-fc1.bias.hex
restore diff mean 9.636278e-08 max 9.331852e-07 min -9.275973e-07
fc fc2.weight torch.Size([10, 32]) max 0.2685824 min -0.2566016 int32 max 4294967189
    int32 min 1203 ../data/fc-fc2.weight.hex
restore diff mean -4.848989e-08 max 9.4622374e-07 min -9.49949e-07
fc fc2.bias torch.Size([10]) max 0.798702 min -1.0660744 int32 max 4294592164 int32 min
    251099 ../data/fc-fc2.bias.hex
restore diff mean -4.0233136e-08 max 6.2584877e-07 min -7.1525574e-07
Test Q: Accuracy: 9021/10000 (90%)
```

将两个模型的计算中间值进行比较，量化误差较小，且在测试集上的准确率也较高，说明量化此量化方案大致可行。量化后的模型参数被保存为十六进制的 `.hex` 文件，以便综合器读取。

在 Bluespec Verilog 的代码实现中，每一层的输入被定义为 `Layer` 接口。主要接口和模块定义如下，包括全连接层、数据加载层、激活函数层、Softmax（实际为 ArgMax）层等。

```
interface Layer#(type in, type out);
    method Action put(in x);
    method ActionValue#(out) get;
```

```
endinterface

module mkLayerData#(parameter String model_name, parameter String
    layer_name)(LayerData_ifc#(td, lines, depth))
    provisos (
      Bits#(td, sz),
      Literal#(td),
      Log#(depth, depth_log),
      Log#(lines, lines_log),
      Mul#(lines, sz, lines_bits)
    );
// ...
endmodule

module mkReluLayer(Layer#(in, out))
  provisos (
    Bits#(in, input_bits),
    Mul#(input_size, 32, input_bits),
    PrimSelectable#(in, Int#(32)),
    Bits#(out, output_bits),
    Mul#(output_size, 32, output_bits),
    PrimSelectable#(out, Int#(32)),
    Add#(input_bits, 0, output_bits),
    PrimUpdateable#(out, Int#(32))
  );
// ...
endmodule

module mkSoftmaxLayer(Layer#(in, out))
  provisos (
    Bits#(in, input_bits),
    Mul#(input_size, 32, input_bits),
    PrimSelectable#(in, Int#(32)),
```

```
    Bits#(out, output_bits),
    PrimIndex#(out, a__)
  );
// ...
endmodule

module mkConvLayer#(parameter String layer_name)(Layer#(in, out))
// now assuming that stride == 1
  provisos (
    // ...
  );
// ...
endmodule
```

通过 Bluespec Verilog 的 FIFO 功能，各层之间通过 FIFO 进行数据传输，从而实现了简单的流水级并行技术处理。代码中还含有卷积层，但其显得过于复杂，故不在此展示。

Bluespec Verilog 也是 Scaleda 支持的一种硬件描述语言，当打开 Bluespec Verilog 源文件时，Scaleda 会自动解析文件，将解析得到的模块、接口、规则等信息显示在结构窗口中，如图 4-21 所示。

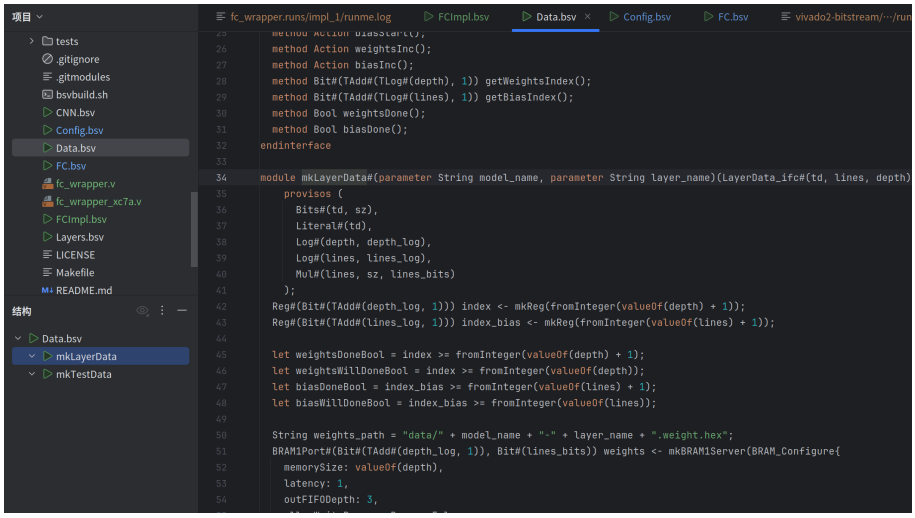


图 4-21　全连接神经网络代码

4.3.2 使用 Bluespec Compiler 进行仿真验证

在设计完成后，本章节能够使用 Bluespec Compiler 进行仿真验证。在

scaleda.yml 中添加 Bluespec 相关目标和任务，并添加 Vivado 相关目标和任务，得到项目结构如图 4-22 所示。

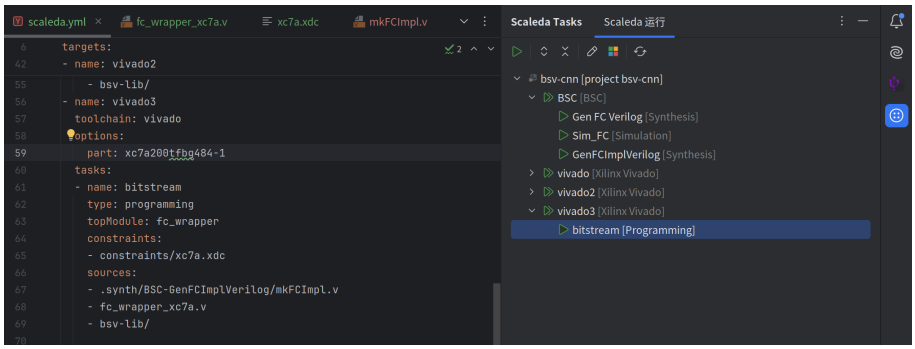


图 4-22　全连接神经网络设计项目结构

执行任务 Sim_FC，Scaleda 将调用 Bluespec Compiler 编译 FC.bsv 到可执行文件，并调用此可执行文件进行仿真。由仿真结果图 4-23 可知，在 1000 个测试样本中，有 931 个样本被正确分类，准确率为 93%。整个网络推理过程是流水化的，最高时延由最大的矩阵相乘决定，为约 786 个周期。

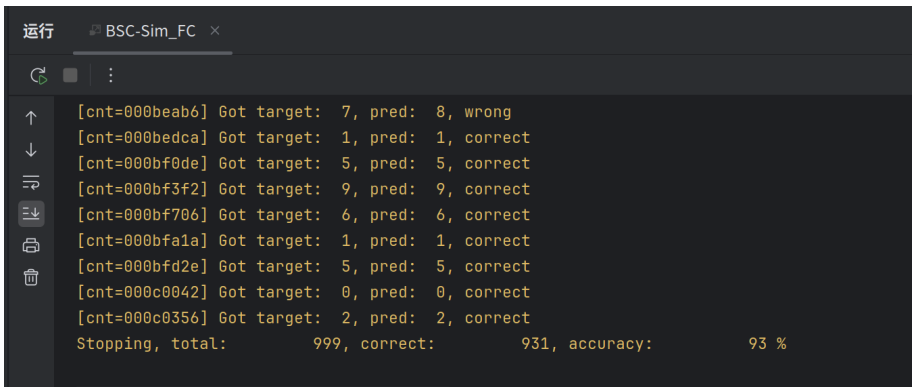


图 4-23　Bluespec Compiler 仿真结果

4.3.3 使用 Xilinx Vivado 进行 FPGA 部署

在设计验证成功后，本设计将继续使用 Vivado 进行 FPGA 部署。执行图 4-22 中的 vivado3-bitream 任务，Scaleda 将调用 Vivado 对 Bluespec 编译生成的 Verilog 文件进行综合、布局布线、生成比特流文件等操作。由于此电路设计还有许多能够优化的地方，所以此处选择 xc7a200tfbg484-1 作为目标 FPGA，以免出现资源不足的情况。

经过一段时间的综合、布局布线等操作，任务正常结束。使用 Vivado 打开生成的项目文件，能够查看当前电路的资源占用情况，如图 4-24 所示。

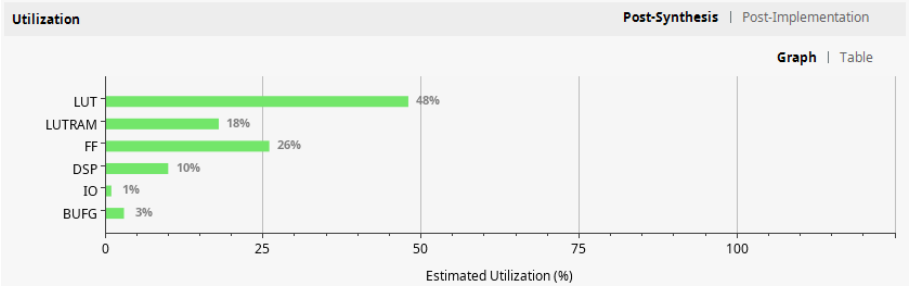


图 4-24　全连接神经网络电路在 Vivado 中的资源占用情况

4.4 本章小结

本章主要介绍了 Scaleda 的使用方法，以及在实际 FPGA 设计中的应用。通过一个简单的流水灯设计，演示了 Scaleda 的基本功能，包括项目创建、工具链设置、仿真、波形查看等，并演示了远程工具链调用。然后，通过一个全连接神经网络设计，演示了 Scaleda 在复杂设计中的应用，包括 Bluespec Verilog 的设计、仿真、综合、布局布线、设计部署等。

由此体现了 Scaleda 面向学生与数字电路初学者，灵活、高效、易用的特点，能为用户提供一个全面而灵活的 FPGA 设计解决方案，辅助用户进行 FPGA 设计与验证。

结　论

本文提出并完成了一款 IDEA 插件 Scaleda，以及一个全新开发的波形查看器 Rvcd。Scaleda 在为 Verilog 等 HDL 代码提供代码解析、编辑提示等功能的基础上，还提供了项目管理、EDA 工具链集成等功能，并且创新性地完成了多种 EDA 工具链的功能集成方案、开箱即用的远程开发环境，以及面向开源社区的 IP 仓库等功能。Rvcd 实现了波形文件的解析、查看等功能，为用户提供了跨平台、高性能、更易用的波形查看和辅助开发工具。同时，两个工具能够无缝集成，为用户提供了更加全面、易用的数字电路设计学习工具。

本论文的主要创造性工作总结归纳如下：

1. 提出并设计了一款面向 FPGA 设计的灵活、通用的课程实践软件 Scaleda，为学生和数字电路设计初学者提供了多功能且实用的数字电路设计学习工具，促进了数字电路设计教学的发展。

2. 提出并完成了一个全新开发的波形查看器 Rvcd，为用户提供了跨平台、高性能、更易用的波形查看和辅助开发工具。

3. 完成了多种 EDA 工具链的功能集成方案、开箱即用的远程开发环境，以及面向开源社区的 IP 仓库等功能，促进了 FPGA 领域开源工具的发展，推动了数字电路设计教育的开放与创新。

今后还应在以下几个方面进行深入研究：

1. 完善 Scaleda 的功能，提供更多的 HDL 语言支持、更多的 EDA 工具链集成方案，提高远程开发环境的性能和易用性，丰富 IP 仓库的内容，提升软件的稳定性和可靠性。

2. 完善 Rvcd 的功能，提供更多的波形格式查看和辅助协同开发功能，提高波形查看器的性能和易用性，为用户提供更加全面、易用的波形查看和辅助开发工具。

3. 在软件开发领域克服更多的技术难题，使用更多软件开发技术和工具，完善软件的测试和发布，提高软件开发的效率和质量。

4. 将软件产品投入实际使用，收集用户反馈，不断改进软件产品，提高软件产品的用户体验和用户满意度。

5. 结合更多已有的开源 FPGA 和数字电路设计工具，进一步推动 FPGA 领域开源工具的发展。

参考文献

[1] AHMED S Z，SASSATELLI G，TORRES L，et al. Survey of New Trends in Industry for Programmable Hardware: FPGAs, MPPAs, MPSoCs, Structured ASICs, eFPGAs and New Wave of Innovation in FPGAs ［C/OL］ // 2010 International Conference on Field Programmable Logic and Applications，［S.l.］，2010：291-297. http://dx.doi.org/10.1109/FPL.2010.66.

[2] 2023 年全球及中国 FPGA 芯片行业现状及竞争格局分析，应用场景和发展潜力广阔 ［EB/OL］. 2023. https://www.huaon.com/channel/trend/915648.html.

[3] WOLF C，GLASER J，KEPLER J. Yosys-a free Verilog synthesis suite ［C］ // Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)：Vol 97，［S.l.］，2013.

[4] WILLIAMS S，BAXTER M. Icarus verilog: open-source verilog more than a year later ［J］. Linux Journal，2002，2002（99）：3.

[5] SNYDER W. Verilator: Open simulation-growing up ［J］. DVClub Bristol，2013.

[6] YADAV G，SHUKLA N K. Pre-Eminance of Open Source Eda Tools and Its Types in The Arena of Commercial Electronics ［J/OL］. International Journal of Advanced Computer Science and Applications，2013，4（12）. http://dx.doi.org/10.14569/IJACSA.2013.041225.

[7] ROSE J，LUU J，YU C W，et al. The VTR project: architecture and CAD for FPGAs from verilog to routing ［C］ // Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays，［S.l.］，2012：77-86.

[8] ALIIANCE C. F4PGA - the GCC of FPGAs ［EB/OL］. 2024. https://f4pga.org/.

[9] NITCLOUD，LSTM-KIRIGAYA，KYLIN. Digital-IDE ［EB/OL］. GitHub，2024. https://github.com/Digital-EDA/Digital-IDE.

[10] Verilog-HDL/SystemVerilog/Bluespec SystemVerilog ［EB/OL］. GitHub，2024. https://github.com/mshr-h/vscode-verilog-hdl-support.

[11] An abstraction library for interfacing EDA tools ［EB/OL］. GitHub，2024. https://github.com/olofk/edalize.

[12] A universal utility for programming FPGAs ［EB/OL］. 2024. https://trabucayre.github.io/openFPGALoader.

[13] SHAH D，HUNG E，WOLF C，et al. Yosys+nextpnr: An Open Source Framework from Verilog to Bitstream for Commercial FPGAs [C/OL] //2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)，[S.l.]，2019：1-4. http://dx.doi.org/10.1109/FCCM.2019.00010.

[14] PARR T. The definitive ANTLR 4 reference [J]. The Definitive ANTLR 4 Reference，2013：1-326.

[15] ANON. IEEE Standard for Verilog Hardware Description Language [J/OL]. IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)，2006：1-590. http://dx.doi.org/10.1109/IEEESTD.2006.99495.

[16] Bluespec SystemVerilog Language Reference Guide [EB/OL]. GitHub，2024. https://github.com/B-Lang-org/bsc/releases/latest/download/BSV_lang_ref_guide.pdf.

[17] 万其琛. 基于强化学习的 FPGA 布局布线算法研究 [C] //，[S.l.]，2024.

[18] SYNTHESIS L，GROUP V. ABC: A System for Sequential Synthesis and Verification [EB]. 2012.

[19] ZHENG X，ZENG S，ZHONG Y，et al. An Efficient VCD Parser for Dynamic Power Estimation of Digital Integrated Circuits [J/OL]. IEEE Embedded Systems Letters，2024：1-1. http://dx.doi.org/10.1109/LES.2024.3380048.

[20] A waveform viewer with a focus on a snappy and extensibility usable interface [EB/OL]. Gitlab，2024. https://gitlab.com/surfer-project/surfer.

哈尔滨工业大学本科毕业论文（设计）原创性声明和使用权限

本科毕业论文（设计）原创性声明

本人郑重声明：此处所提交的本科毕业论文（设计）《面向 FPGA 设计的课程实践软件开发》，是本人在导师指导下，在哈尔滨工业大学攻读学士学位期间独立进行研究工作所取得的成果，且毕业论文（设计）中除已标注引用文献的部分外不包含他人完成或已发表的研究成果。对本毕业论文（设计）的研究工作做出重要贡献的个人和集体，均已在文中以明确方式注明。

作者签名：　　日期：　　年　　月　　日

本科毕业论文（设计）使用权限

本科毕业论文（设计）是本科生在哈尔滨工业大学攻读学士学位期间完成的成果，知识产权归属哈尔滨工业大学。本科毕业论文（设计）的使用权限如下：

（1）学校可以采用影印、缩印或其他复制手段保存本科生上交的毕业论文（设计），并向有关部门报送本科毕业论文（设计）；（2）根据需要，学校可以将本科毕业论文（设计）部分或全部内容编入有关数据库进行检索和提供相应阅览服务；（3）本科生毕业后发表与此毕业论文（设计）研究成果相关的学术论文和其他成果时，应征得导师同意，且第一署名单位为哈尔滨工业大学。

保密论文在保密期内遵守有关保密规定，解密后适用于此使用权限规定。

本人知悉本科毕业论文（设计）的使用权限，并将遵守有关规定。

作者签名：　　日期：　　年　　月　　日

导师签名：　　日期：　　年　　月　　日

致　谢

感谢我的导师徐勇教授对我的指导和帮助。徐老师在我准备毕业设计期间给予了我很多的帮助和支持，不仅在学术上给予了我很多的指导，还在生活上给予了我很多的关心和帮助。徐老师严谨的治学态度、丰富的科研经验、耐心的指导和关心学生的工作作风，都让我受益匪浅。

感谢同年级的万其琛同学在本项目初期的合作。万其琛同学在项目的初期阶段为本项目的设计和开发提供了很多的帮助和支持，完成前期的许多 UI 设计、相当一部分创新点的提出和部分程序设计，为本项目的顺利进行打下了良好的基础。

感谢哈工大（深圳）-中微半导人工智能芯片联合实验室为我毕业设计提供了选题、灵感和实际试用支持，促进了本项目的落地效果和实际应用。

最后，感谢我的家人和朋友们对我的支持和鼓励。感谢他们在我学习和生活中的陪伴和支持，让我能够顺利完成学业。