存档日期： 存档编号：





NanChang Hangkong University

本科毕业设计

UNDERGRADUATE DESIGN

设 计 题 目：基于Flex和Bison提取Verilog网

表数据

姓 名： 金海林

专 业： 软件工程（东软班-嵌入式系统发）

班 级 、 学 号： 152042、15204218

指 导 教 师： 夏晓峰

学士学位设计原创性声明

本人声明，所呈交的设计是本人在导师的指导下独立完成的研究成果。除了文中特别加以标注引用的内容外，本设计不包含法律意义上已属于他人的任何形式的研究成果，也不包含本人已用于其他学位申请的论文或成果。对本文的研究作出重要贡献的个人和集体，均已在文中以明确方式表明。本人完全意识到本声明的法律后果由本人承担。

作者签名： 日期：2019年6月3日

学位设计版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位设计的规定，同意学校保留并向国家有关部门或机构送交设计的复印件和电子版，允许设计被查阅和借阅。本人授权南昌航空大学可以将本设计的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位设计。

作者签名： 日期： 年 月 日

导师签名： 日期： 年 月 日

摘要

人类发展史上产业革命分为三个阶段，农业革命，工业革命，知识革命。现阶段处于知识革命的初期——用信息数据来优化生产。伴随数据量和数据复杂度的巨增，人类迫切的需要功耗更低，速度和并发能力更强的芯片。随之而来是，芯片设计师对一套好的EDA工具的利好更为迫切。

芯片中晶体管数量直接决定着计算力，摩尔定律也是通过减少芯片线路的间距提高同等体积下的晶体管的数量，从22纳米，10纳米，7纳米，产业界都是选择压缩间距，换取计算力的提升。但是这条路越走会越艰难，因为从物理理论角度压缩的极限最小是1纳米，所以未来能压缩的空间极为有限。如果想要摩尔定律继续下去，产业界必须换一个方向，一个让摩尔定律持续下去，电子产业持续火热下去的方案。

现在产业界一个方向就是改变芯片架构，并且用更好的EDA软件，实现在同等晶体管量级的情况下达到更快的计算速度，更低的功耗。芯片设计软件（EDA），它可以辅助电路设计，提高效率。但遗憾的是，美国三家EDA企业几乎垄断了全球的EDA市场，据称，我们国产芯龙头海思每年为此付费在千万级别。

本项目做的就是EDA工具中数据入口处的重要一环。

EDA工具中的功耗分析和时序分析都需要用到芯片设计时产生的门级网表数据，该项目工作是用flex和bison按《IEEE 1364-2001Verilog Hardware Description Language》标准将Verilog语言描述的网表数据提取到预定义的数据结构中以供EDA工具做分析时调用。

关键词：芯片 摩尔定律 EDA工具 Verilog 门级网表 Flex Bison 数据提取

Abstract

In the history of human development, industrial revolution can be divided into three stages: agricultural revolution, industrial revolution and knowledge revolution. The present stage is at the beginning of the knowledge revolution-- Use information data to optimize production. With the huge increase of data volume and data complexity, people urgently need chips with lower power consumption, faster speed and stronger concurrency ability. As a result, chip designers are more eager for a good set of EDA tools.

The number of transistors in the chip directly determines the computing power. Moore's Law also increases the number of transistors in the same volume by reducing the pitch of the chip lines. From 22 nm, 10 nm, and 7 nm, the industry chooses compression spacing in exchange for computing power. Upgrade. But the harder it goes, the harder it is because the limit of compression from a physical theory is at least 1 nanometer, so the space that can be compressed in the future is extremely limited. If you want Moore's Law to continue, the industry must change direction, a plan to keep Moore's Law going, and the electronics industry continues to heat up.

One direction in the industry today is to change the chip architecture and use better EDA software to achieve faster computing speeds and lower power consumption with the same transistor level. Chip Design Software (EDA), which aids in circuit design and increases efficiency. But unfortunately, the three EDA companies in the United States have almost monopolized the global EDA market. It is said that our domestic core leader, Haisi, pays tens of millions of dollars each year.

What this project does is an important part of the data entry in the EDA tool.

Both power analysis and timing analysis in DEA tools require gate-level netlist data generated during chip design. This project uses flex &amp; bison to extract the network table data described by Verilog language into predefined data structure according to the standard of "IEEE 1364-2001 Verilog Hardware Description Language" for analysis by EDA tools.

Key Words：chip Moore's Law EDAtool Verilog gate level netlist Flex Bison Data extraction

**目 录**

[摘要 I](#_Toc9256247)

[Abstract II](#_Toc9256248)

[目 录 III](#_Toc9256249)

[图清单 V](#_Toc9256250)

[表清单 VI](#_Toc9256251)

[1 绪论 1](#_Toc9256252)

[1.1 课题的意义和目标 1](#_Toc9256253)

[1.2 国内外研究的现状 1](#_Toc9256254)

[1.3 论文的研究内容和主要工作 2](#_Toc9256255)

[1.4 论文组织结构 2](#_Toc9256256)

[2 系统分析 4](#_Toc9256257)

[2.1 可行性分析 4](#_Toc9256258)

[2.2 需求分析 6](#_Toc9256259)

[2.3 方案比选 7](#_Toc9256260)

[3 概要设计 9](#_Toc9256261)

[3.1 系统架构图 9](#_Toc9256262)

[3.2系统子模块结构图 10](#_Toc9256263)

[3.3 流程图 17](#_Toc9256264)

[3.4 类图 19](#_Toc9256265)

[4 系统详细设计与实现 22](#_Toc9256266)

[4.1 词法预处理器 22](#_Toc9256267)

[4.2词法分析 25](#_Toc9256268)

[4.3 语法分析 31](#_Toc9256269)

[5 系统运行与效果分析 34](#_Toc9256270)

[5.1 界面设计概要 34](#_Toc9256271)

[5.2运行效果分析 34](#_Toc9256272)

[6 系统测试 36](#_Toc9256273)

[6.1 测试方法 36](#_Toc9256274)

[6.2 测试方案及计划 36](#_Toc9256275)

[6.3 测试过程及结果分析 37](#_Toc9256276)

[7 总结与展望 42](#_Toc9256277)

[7.1 总结 42](#_Toc9256278)

[7.2 展望 42](#_Toc9256279)

[参考文献 43](#_Toc9256280)

[致谢 45](#_Toc9256281)

图清单

|  |  |  |
| --- | --- | --- |
| 图序号 | 图名称 | 页码 |
| 图1-1 | EDA公司发展现状 | 2 |
| 图3-1 | 系统架构图 | 9 |
| 图3-2 | 词法预处理结构图 | 10 |
| 图3-3 | 词法分析器结构图 | 11 |
| 图3-4 | Verilog语法类Source text | 12 |
| 图3-5 | Verilog语法类Declarations | 12 |
| 图3-6 | Verilog语法类Primitive instance | 13 |
| 图3-7 | Verilog语法类Module and generated instantiation | 13 |
| 图3-8 | Verilog语法类UDP declaration and instantiation | 13 |
| 图3-9 | Verilog语法类Behavioral statements | 14 |
| 图3-10 | Verilog语法类Specify section | 14 |
| 图3-11 | Verilog语法类Expression | 15 |
| 图3-12 | Verilog语法类General | 16 |
| 图3-13 | 系统前端流程图 | 17 |
| 图3-14 | 系统后端流程图 | 18 |
| 图3-15 | 类之间的组织关系图 | 19 |
| 图3-16 | 类Verilog\_p | 20 |
| 图3-17 | 类VerilogModule\_p | 20 |
| 图3-18 | 类VerilogGate\_p | 20 |
| 图3-19 | 类VerilogNet\_p | 20 |
| 图3-20 | 类VerilogBus\_p | 21 |
| 图4-1 | Verilog include预处理指令语法 | 23 |
| 图4-2 | Verilog number语法 | 27 |
| 图4-3 | Verilog module\_declaration语法 | 31 |
| 图5-1 | 程序运行效果图 | 39 |
| 图5-2 | 输出文件效果图 | 39 |
| 图5-3 | 程序内存消耗分析图 | 39 |
| 图6-1 | 原数据图1 | 43 |
| 图6-2 | 还原数据图1 | 44 |
| 图6-3 | 脚本测试比较图 | 44 |
| 图6-4 | 原数据图2 | 45 |
| 图6-5 | path分析报告 | 45 |
| 图6-6 | 还原数据图2 | 46 |
| 图6-7 | 内存检测 | 47 |

表清单

|  |  |  |
| --- | --- | --- |
| 表序号 | 表名称 | 页码 |
| 表3-1 | Cstruct表 | 16 |
| 表4-1 | Verilog关键字表 | 25 |
| 表4-2 | Verilog操作符 | 30 |
| 表6-1 | 测试进度安排表 | 42 |

1 绪论

1.1 课题的意义和目标

1.1.1 课题意义

芯片设计师得借助专门为芯片设计而编写的程序工具才能完成目前如此数据量级的芯片设计工作，就好比厨师做菜，越顶级的厨师处理越难处理的顶级食材越需要一套好的厨具。一道好菜，三者缺一不可。

中国芯片一直弱于外国，也有很大一部分原因是我们国家没有本土的EDA厂家可以合作。就好比越精密的电子器件需要精度越高的工业机床，中国的精密核心零部件很多依赖进口，也是因为中国没有加工这个精度级的机床，就算是有，使用和维护的成本也是非常高昂。中国的芯片设计如果真的想要强大，本土EDA工具肯定要发展起来，不然芯片设计将一直都将受制于国外。

人类已经进入到知识爆炸的时代，互联网也悄然而至，传统行业在互联网的加持下熠熠生辉。但是这也带来了新的时代问题，当传统行业在借力互联网的过程中将产生巨大的数据量，这在以前是无法想象的。这们多的数据势必需要更强大的计算能力才能驾驭。芯片制造伴随着14纳米10纳米7纳米的物理极限的逐步逼近，同等体积下的芯片能容纳的晶体管也越来越多，现在像国内最好的移动端芯片的麒麟980芯片晶体管的数量大约69亿左右。单凭人力已经不可能完成如此晶体管量级的设计工作，现实中是凭程序去分析如此数量如此复杂的数据也是有点不堪负重。所以EDA工具的本身的发展情况已经上升到左右是否能设计出一款性能和功耗都良好的芯片的关键之一。

1.1.2 课题目标

本课题的是EDA工具中的重要一环。EDA分析计算必须用到的网表数据的提取。

1.2 国内外研究的现状

1.2.1 国内发展现状

与国外相比，中国这一片领域几乎空白，目前国内只有华大九天一家公司在做。不是我们国家不想做，而是专业软件的门槛本身就高，再加上国内的芯片设计制造产业链本身就不完备和强大，没有制定该行业标准的能力，所以国内芯片设计软件公司也一直无法跻身于世界主流。相对于图1-1的这三家公司，本土的EDA公司，华大九天一直没有很大的主动权。中兴制裁事件国外的Cadence公司也有参与，他停止对中兴软件服务，意味着中兴要用到芯片设计软件相关的业务将停滞不前，从源头切断了CPU的设计生产，釜底抽薪。由此可见，我国的芯片EDA工具对我国的重要性。

1.2.2 国外发展现状

美国在EDA领域目前遥遥领先。目前绝大多数芯片设计公司用的都是国外这三家的软件产品：

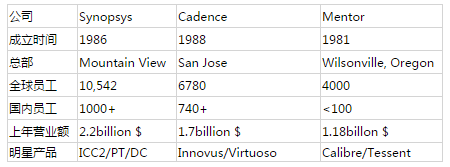


图1-1 EDA公司发展现状图

1.3 论文的研究内容和主要工作

本项目旨在将芯片设计的过程所需要的门级网表数据提取到预定义的数据结构中以供EDA工具做分析和计算时调用。

结合编译器原理的前端知识（预处理，词法分析，语法分析，语义分析）将Verilog这门硬件语言描述的数据打散成很小的粒度，再将其转移到对应的Cstruct结构树中，在经过相应的上层处理最终转移到预定的数据结构中。其中要保持数据量不变，但数据存储方式已经是转移到了适合EDA处理和计算的数据结构中。

1.4 论文组织结构

本文的主要内容是基于Flex和Bison提取Verilog网表数据的设计与实现进行描述，一共分为6个章节。具体章节安排如下：

第一章，绪论。本部分主要描述了EDA工具的现状与发展，研究意义和目标。

第二章，系统分析。本部分主要对系统进行可行性分析，需求分析和最后的方案比较。

第三章，系统概要设计。本部分主要对系统的总体架构，数据提取的主要流程以及类图进行分析和讲解。

第四章，系统的详细设计与实现。本部分主要重点讲解系统各功能模块的的算法设计以及核心代码的展示。

第五章，系统的运行效果分析。本部分主要展示系统的运行效果，并配有相关的文字进行介绍。

第六章，总结与期望。本部分主要介绍本系统的不足之处，以及优化的解决方案，和做这个思考。

2 系统分析

2.1 可行性分析

2.1.1 经济可行性

本系统对服务器配置和服务器环境要求不高，开周期在两个月左右，目前系统稳定性良好维护性尚可。

2.1.2 技术可行性

本系统的运行环境是CentOS7.6，词法分析器用的是Flex2.5.37，语法分析器用的是Bison3.0.4，g++编译器用的是4.8.5，编程语言底层用的是C，上层用的是C++。Verilog参考标准是IEEE 1364-2001。

现在服务器部署的大部分都是CentOS，因为它非常的稳定。同理Flex，Bison也是一门非常老的技术，并且经过历史的洗礼，已经变得非常的稳定。C和C++做为系统编程的不二之选，而且他们也历史久远且强大非常。经过多方的考量，Verilog的数据提取决定采用Flex和Bison，再结合其对C和C++极佳的支持性，最终决定采用这套技术提取Verilog数据。综合上述，此套方案技术可行。

（1）Flex技术分析

Flex概述

Flex 是一个产生词法分析器的工具。词法分析器能够识别出文本中的词法模式。

Flex程序读取给的输入文件，或者没有文件名的标准输入，从而生成一个词法分析器。这个词法分析器由正则表达式和C语言代码组成，称之为规则。Flex生成的词法分析器以一个C文件呈现，默认的文件名是lex.yy.c，文件里定义了一个例程yylex()。这个文件能和flex例程库编译链接生成一个可执行程序。当这个可执行程序执行的时候，它分析会用已定义的正则表达式去分析输入。当它发现一个匹配的时候，便会执行与之对应的C代码。

Flex工作原理

Flex分为三部分，预定义，规则，和用户自定义的代码。

第一部分：预定义

预定义中用户可以添加自己需要的头文件，自己需要的函数声明，还有对应的正则表达式的别名，和一些模式名。

第二部分：规则

每个正则表达式后面对用了一段C代码，词法解析器把输入流分成若干个片段每个片段，每个片段被其中的某个或多个正则表达式匹配成功，匹配成功后其对应的C代码块就会执行。

第三部分：用户自定义代码块

用户可以在这个部分去定义前面声明的函数。

（2）Bison技术分析

Bison概述

Bison是一个通用的语法分析器的生成程序，它能把一个上下文无关的语法转化成一个自左向右查看的语法解析表。一旦你对Bison熟练，你能用它写大都数语言的解析器，从简单的左面计算器到复杂的语言他都能胜任。

Bison工作原理

Bsion分为三部分，预定义，规则，和用户自己定义的代码。

第一部分：预定义

用户可以在预定义部分添加自己需要的头文件，声明函数，声明记号和非终结符。

第二部分：规则

规则的结构用的BNF语法结构，一个或多个记号，非终结符组成一个非终结符，这些记号和非终结符组成的逻辑组合最终构成了一个语法树。这棵语法树上最小的分支是由非终结符和记号组成。语法分析器持续的调用词法分析器，每次调用词法分析器，词法分析器都返回一个记号给语法分析器。Bison分析这些记号基于语法分支上的记号顺序进行匹配，一旦匹配完成就归约。这个语法分支后面对应的C语言代码块就会执行。每个C语言代码块也可以称之为一个动作，动作做的事情就是语义分析。

第三部分：用户自定义代码块

用户可以在这部分定义自己的代码块。

2.1.3 法律可行性

本项目提取的是Verilog语法格式的网表数据，1995年12月，Verilog成为IEEE标准，这意味着我们可以免费使用它。

提取数据的工具是Flex和Bison，Flex是unix系统下的一个免费程序，而且它本身也是开源。Bison是GNU中的一个自由软件。所以用这两款工具开发的程序是不需要考虑用这两个程序开发程序造成的版权问题。

2.1.4 可持续发展可行性

该程序是公司功耗分析的一部分，会随着公司产品的发布逐渐完善，直到达到极为稳定和健壮的程度才会停止优化。

程序的接口本身也非常丰富可以进一步改进从而实现RTL级的Verilog文件解析。数据提取方面，程序提出来的数据全部放在一个过度类中，避开了和计算类的直接耦合，当计算类发生变动时，整个程序并不需要做很大的改动就可以运行。

2.2 需求分析

2.2.1 需求描述

EDA作为芯片设计的最上游，其重要性毋庸置疑。像EDA中的功耗分析软件其需要的输入文件大概在10种左右，每种数据都需要提取，提取的方法也是因文件的内容而异。但他们都有一个共同之处，每种数据对于EDA功耗分析都是非常重要的，Verilog中的网表数据更是其需要的常用数据。

（1）任务需求

本项目所需要实现的是从芯片行为设计描述经过Design Compiler综合得到的门级设计描述中将其中的网表数据提取处理出，供EDA工具调用。在提出完成后保证提取器提取过程中申请的内存全部释放，避免内存泄漏。剩下的内存空间消耗全部来自预定义的数据结构。

（2）功能需求

预处理模块

Verilog硬件描述语言，和C语言有很多的共通之处。文件预处理就是其中之一，在对verilog文件做正式的处理之前将verilog文件中的包含头文件的语句替换成完整的头文件的内容。除此之外还要对verilog语言中的其他宏指令进行处理。

词法分析模块

编写的词法分析语法通过flex编译生成的词法分析器，将对Verilog文件内容逐个字符的扫描，每匹配到已定义的词法时有选择地返回一个标记给语法分析器。

语法分析模块

编写的verilog语法树通过bison编译生成语法分析器，根据词法分析返回的标记去触发具体的verilog语法树分支，当某个分支匹配完成时，归约。以此类推最终返回到根节点

语义分析模块

每个语法树的分支都可以插入C或C++语法的语句，语法树的每个分支被归约之后执行对应的代码块，代码块将这个语法分支上的数据插入到这个分支对应的C语言的结构体中。文件解析完成时，整个Verilog中的数据将转移到由多个Cstruct构成的数据结构构成的结构树中。

门级数据校验模块

从Cstruct构成的数据树中，取出门级的Verilog数据，根据其对应的语法规则将其还原成门级verilog文件，用python脚本将其和输入的verilog文件进行比对，校验数据。

临时过渡数据模块

将Cstruct构成的数据树中的门级verilog数据转移到C++的模板容器类中，便于后期对数据的进一步处理。数据全部转移完成之后，一次性将Cstruct构成的数据树占用的内存全部释放。

待调用数据模块

将C++模板容器类中的数据打散，将其拆分成方便处理的数据粒度和格式，再一次存入新创建C++模板容器类中，数据全部拆分完成之后，将临时过渡的模板容器类申请的内存释放。一个静态类指针指向拆分后的模板容器类地址，调用者通过该静态指针调用拆分后的数据。

2.2.3 系统非功能需求

本系统对性能和代码的工程质量和控制都有要求，再性能方面要求用系统编程语言这也就限制了只有C和C++了。

本项目的底层是纯C实现的，上层为了计算调用的方便同意转成了C++。这虽然给系统带来了强大的性能但意味着它的维护性的下降。

版本控制采用的是SVN，集中式版本控制器。本项目的所有代码都放在服务器的仓库中进行统一的管理。

帮助文档采用的是doxygen的工具，只要采用统一符合一定规范的代码注释风格就能生成很漂亮的网页版帮助文档。对于程序员来说是非常友好的一款软件。

2.3 方案比选

2.3.1 方案一：

采用python提取：

Python是一个脚本解释型语言，网上很多爬虫用python来写，但是网络的上的网页数据并不会嵌套的很深，对于语言型的数据并没有好的解决方案。

2.3.2 方案二：

采用Flex和Bison，Flex是通用且快速的词法分析器，其核心是正则表达式的匹配，时间复杂度很低。Bison是目前非常成熟的语法分析器，往往和Flex一起使用。而且从友商的策略来看，他们选择的也都是Flex和Bison的组合。

2.3.3 必选结论

通过两个方案的对比，可以发现，除了Flex和Bison这两个工具，很少用工具能做到Verilog这种复杂型语言的数据解析。所以最终选择方案二。

3 概要设计

3.1 系统架构图

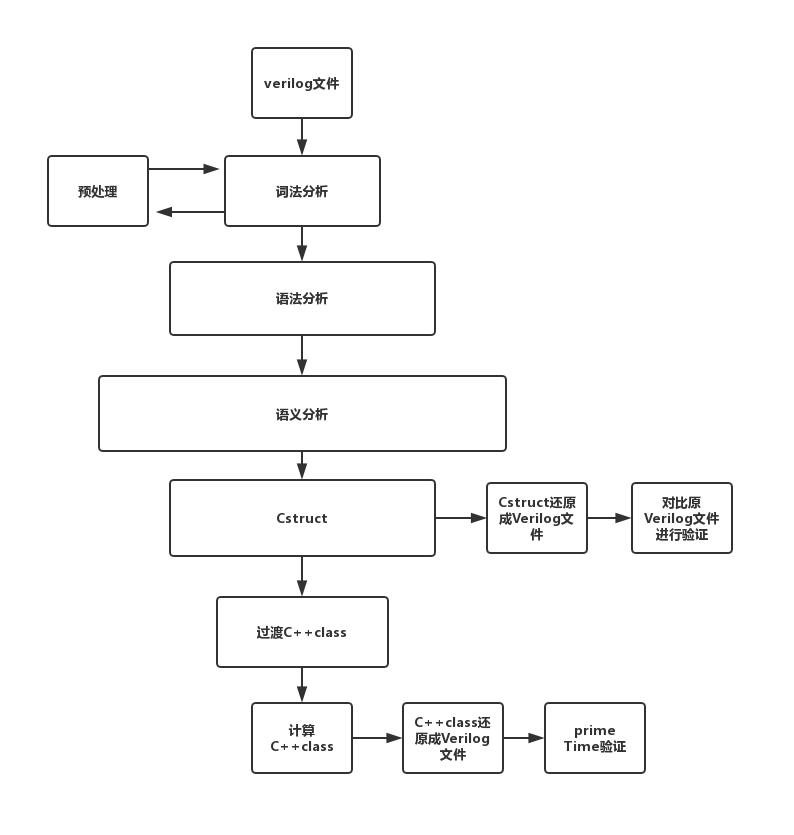


图3-1 系统架构图

图3-1描述的是整个系统的架构图，图中出现的矩形越宽代表这个矩形所代表的模块里面的接口越复杂。整个架构图的就像两个拼起来的树，一个倒的一个正的。一个文件被拆成若干的数据流片段，越往下走越拆的越散，最终都散开后被存储在Cstruct构成的树中，等待被组装，组装的方式和拆开的方式截然不同。组装的过程中是从Cstruct到C++class的这么一个过程，数据从很散的状态再次聚合，聚在另一个树中，C++class构成树中。经过一系列的处理最终达到计算分析能处理的数据粒度和要求。前提是有用的信息数据0丢失。

3.2系统子模块结构图

3.2.1 预处理器结构

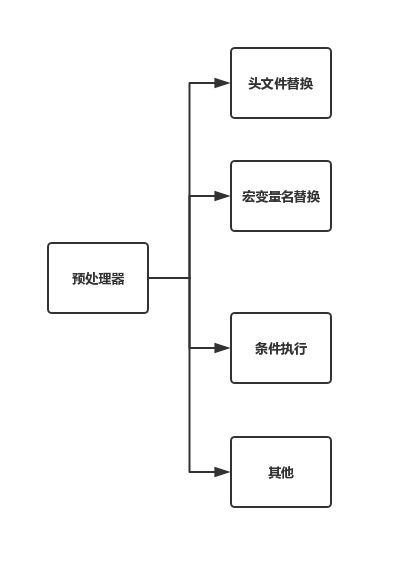


图3-2 词法预处理器结构图

如图3-3所示，预处理器是嵌在词法分析器内部，因为其也需要词法分析，但其又必须得和Verilog词法分析独立开，因为预处理器的指令不属于Verilog标准里面的一部分。一门语言的其对应的预处理指令，往往扮演着程序的好帮手的角色，所以虽然不是语言标准的一部分，但是却往往会伴生出现。所以程序也必须考虑预处理指令的解析。如图3-2所示，预处理指令最常被使用的三种指令就是头文件替换，宏变量名替换，条件执行。所以本程序也只是处理这三种指令，其他的指令要么匹配不处理要么不匹配。

3.2.2 词法分析器结构图

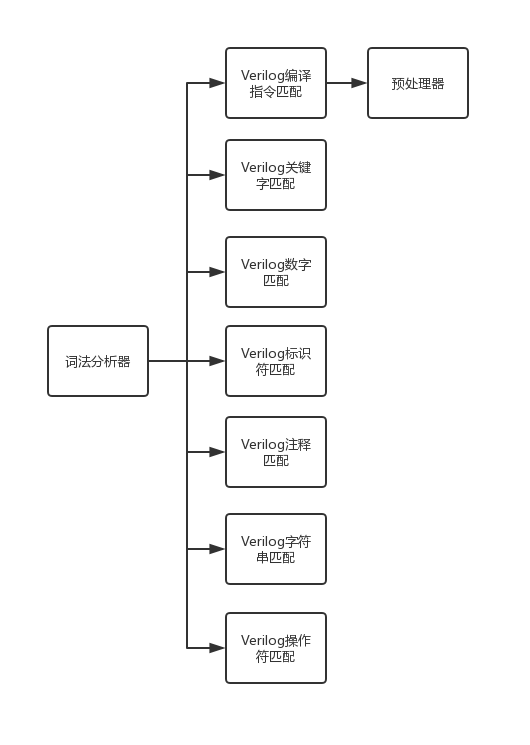


图3-3 词法分析器结构图

如图3-2中所示的，词法分析将Verilog词法分成七大类，便于对它们进行独立管理。

3.2.3 语法分析器结构图

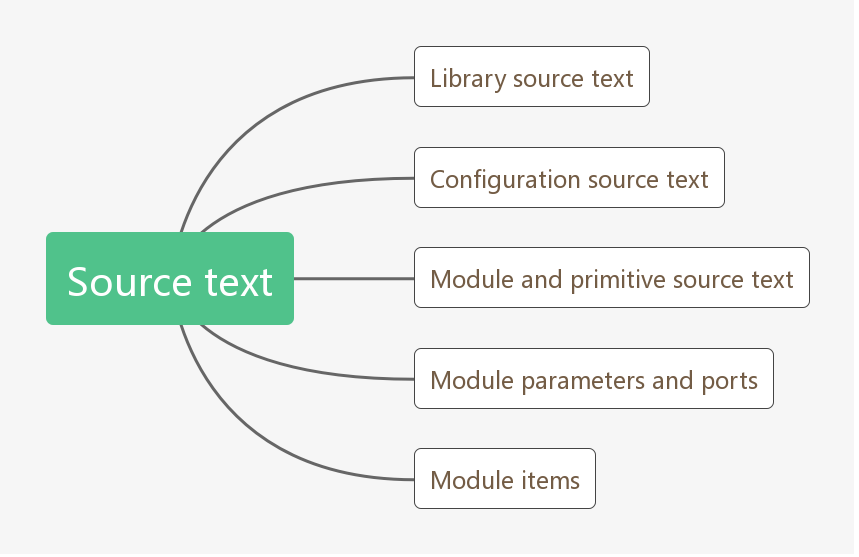


图3-4 Verilog语法树第一层Source text

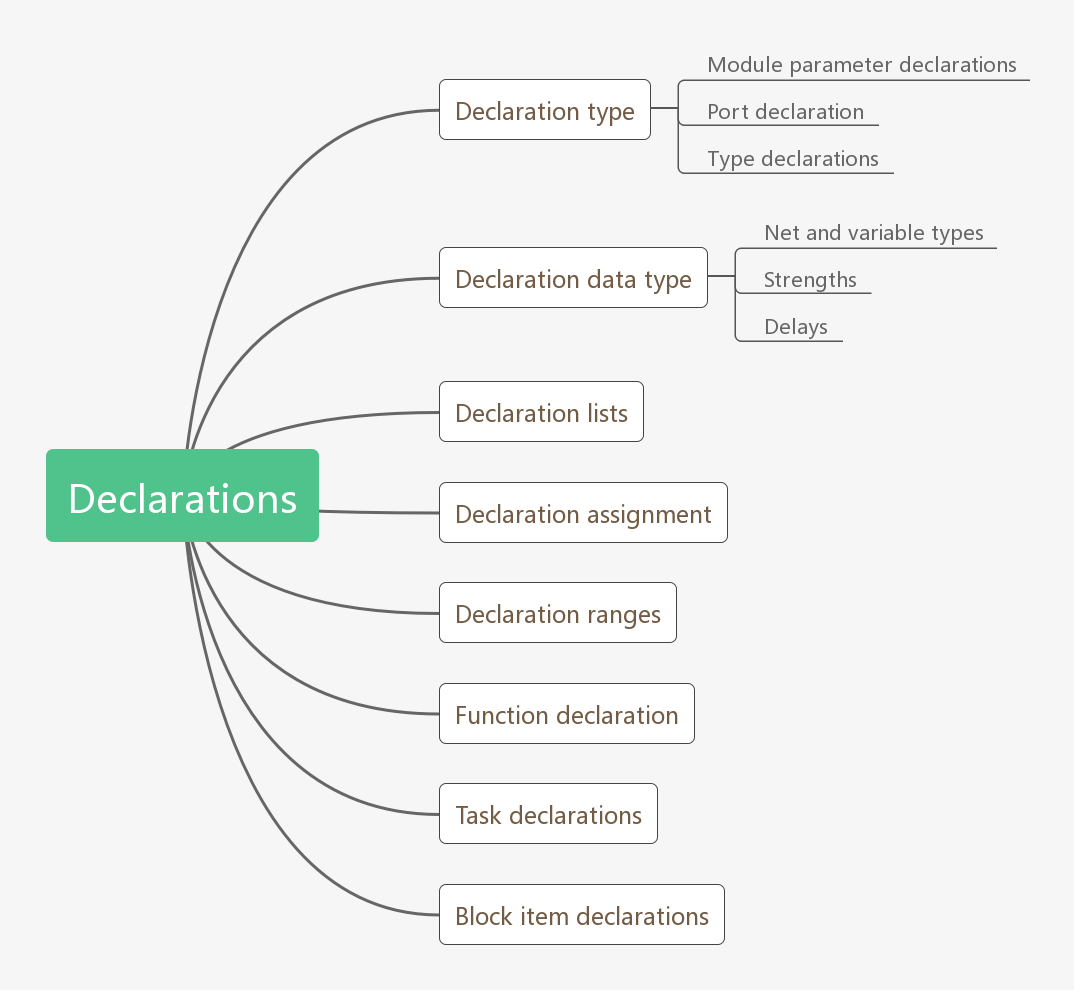


图3-5 Verilog语法树第二层Declarations

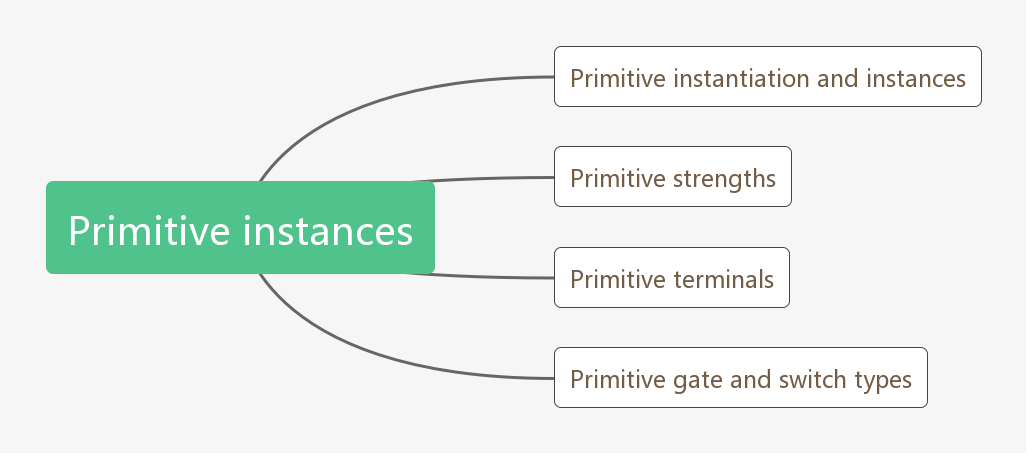


图3-6 Verilog语法树第三层Primitive instance



图3-7 Verilog语法树第四层Module and generated instantiation

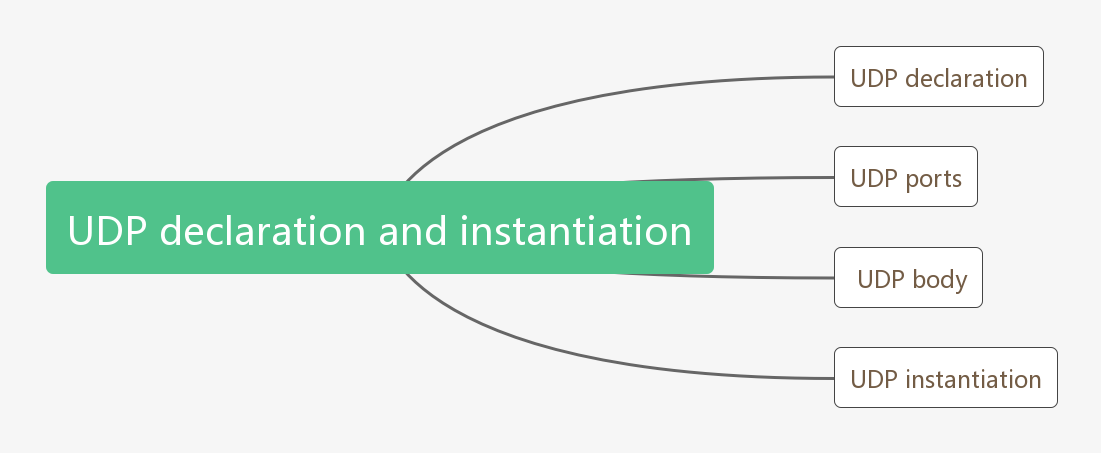


图3-8 Verilog语法树第五层 UDP declaration and instantiation

语法分析的是参考IEEE制定Verilog标准BNF格式的语法树来进行的，在IEEE标准中，其将Verilog语法树分为九个大层，相当于Verilog这棵树上有九根粗大的枝干，这每个枝干上又有很多小的枝干，图4-12，描述的就是每个枝干和其对应的子枝干。这几张图中并没有出现叶子节点，因为叶子节点很多，需要用特定的语法格式描述比如BNF范式来描述，不适合用文字的形似来表达，所以在本论文中并没有体现。词法分析器将Verilog拆分成一个个字符或词后以语法分析器预先定义的记号形式返回给语法分析器。这些记号是枚举类型。这些记号就是这颗巨大语法树上的叶子节点。不同分支的末梢最小的树枝上长得叶子都不同。而最小树枝可以任意顺序组合接在比他大一级的树干上，最终长成Verilog这棵语法树。单纯语法树用编程实现工程量很大，但还不是语法分析的最难点，最难的是语法分析对应的语义分析。要能够熟练应用Verilog这门语言必须得有微电子方面得基础，对电路，芯片有一定得了解，不然语义分析无从谈起。

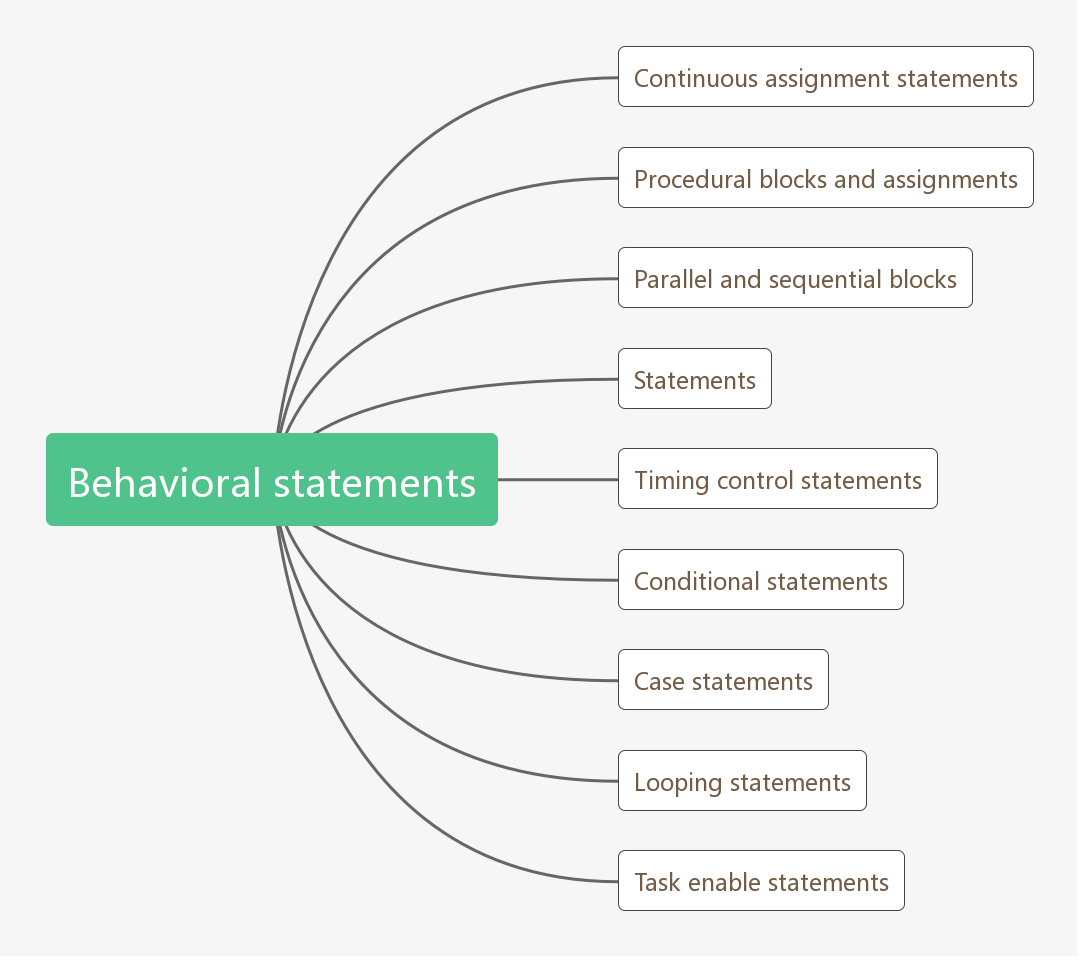


图3-9 Verilog语法树第六层Behavioral statements

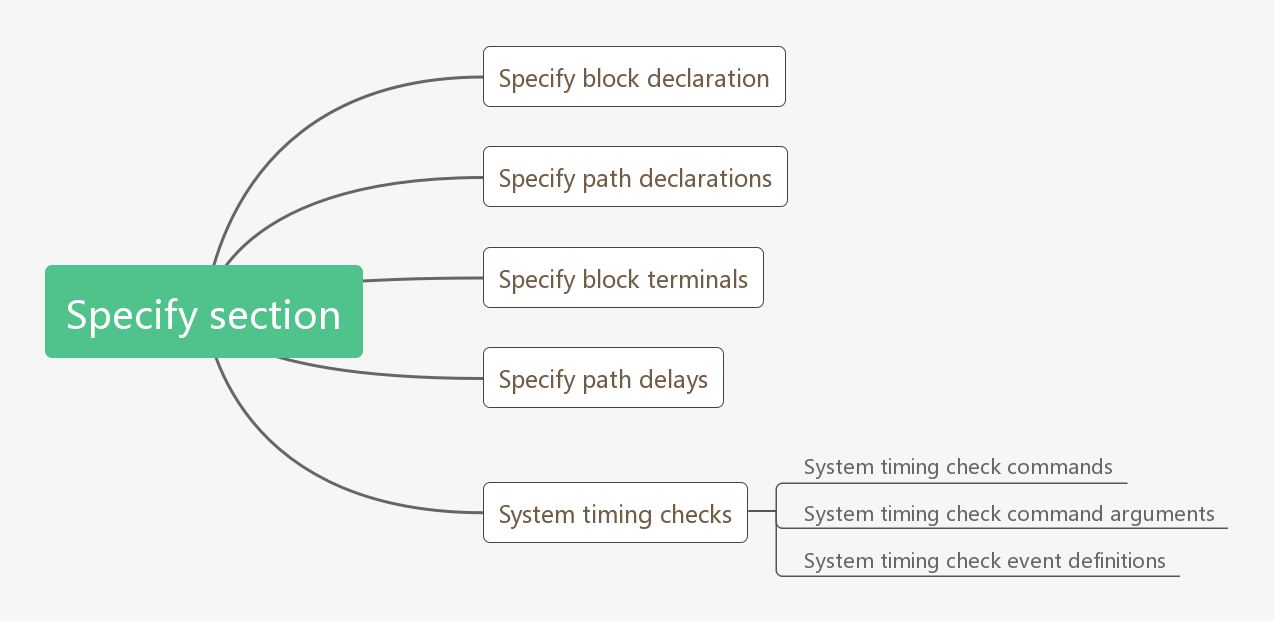


图3-10 Verilog语法树第七层Specify section

语义分析就是对每个语法分支，用自己的思维逻辑去理解，这个分支这些记号为什么要有这样的顺序组合，这个顺序的组合我可以怎样去利用其形成自己的数据组合，形成具体含义。这就是语义分析要做的事。本项目的语义分析做的也是这样一件事，对需要的语法分支根据实际需要将其组合存放进对应的Cstruct中。

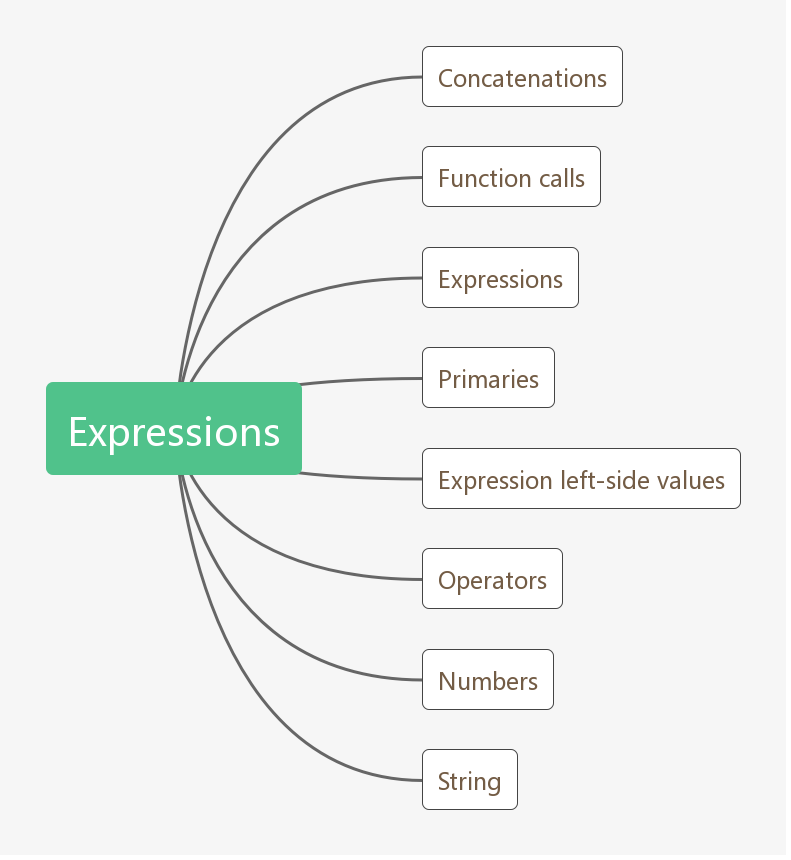


图3-11 Verilog语法树第八层Expression

Cstruct是个简化版的Verilog语法树，因为Verilog里很多语法分支可以用同一个Cstruct来存储。Verilog语法的上层很多都是以迭代的形式出现，不同的迭代其实都可以用C语言中的链表来统一存储。

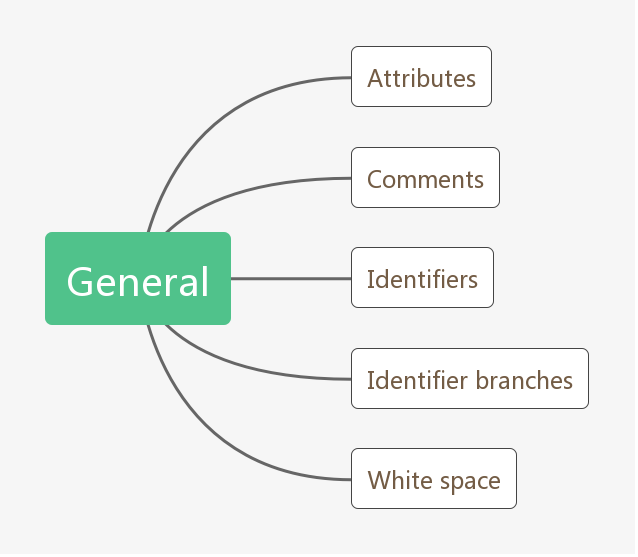


图3-12 Verilog语法树第九层General

3.2.4 Cstruct

|  |  |  |
| --- | --- | --- |
| ast\_source\_item | ast\_module\_declaration | ast\_new\_source\_item |
| ast\_udp\_declaration | ast\_node\_attributes | ast\_identifier |
| ast\_list | ast\_parameter\_declarations | ast\_port\_direction |
| ast\_universal\_struct | ast\_expression | ast\_mode\_item |
| ast\_port\_declaration | ast\_assignment | ast\_module\_instantiation |
| ast\_type\_declaration | ast\_net\_type | ast\_range |

|  |  |  |
| --- | --- | --- |
| ast\_drive\_strength | ast\_charge\_strength | ast\_delay3 |

|  |  |  |
| --- | --- | --- |
| ast\_single\_assignment | ast\_lvalue | ast\_delay\_value |
| ast\_port\_connection | ast\_concatenation | ast\_function\_call |

|  |  |  |
| --- | --- | --- |
| ast\_operator | ast\_primary |  |

表3-1 Cstruct表

Cstruct是一个非常核心的一部分，是程序的中间部分，程序的核心，这个程序的好坏也体现在Cstruct树的设计上。每个Cstruct由两大部分组成。一个是用来存储数据的变量，这个变量可以是一个具体的变量，也可以是指向另一个结构体的指针。另一部分是枚举类型，这个枚举类型反应的是这些变量的组合顺序，这些顺序跟上文中提到记号和细小的语法分枝组合顺序相对应。每个枚举类型反应一个组合顺序。一个结构体中枚举类型里面枚举值的个数都不同，如果这是一个大的枝干其上接这很多个小分支，这个结构体正好用来存储这个大树干中出现的值那么，这个结构体里枚举类型的枚举值就很多，以便对应这些分支。如表3-1列出的29个结构体，构成一个Cstruct树，通过语义分析与语法分析接触，从语法树分支中获取数据。中上层的Cstruct中里面或许会出现链表指针，存储同一类型的Cstruct的多个变量。

3.3 流程图

3.3.1系统前端流程图

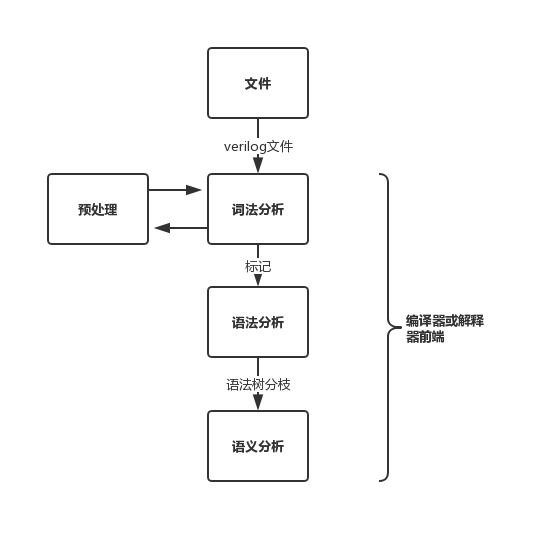


图3-13 系统前端流程图

流程图有两张图分别反应了系统前端和后端的程序大致流程。

图3-13反应的是系统前端的流程图，可以看到程序的输入是verilog文件，文件经过词法分析器的拆解编程细碎的记号流。这些记号流根据流的顺序不断的触发语法树构成的语法树上的语法分支，每触发一个分支，其语法分支对应动作就会执行用来处理这个语法分支触发后的操作。

图3-14反应的是系统的后端图，后端负责将语法树里的数据整合，分门别类的存放在其对应的数据结构中，也就是Cstruct中。当所有数据都存放在Cstruct后，将其做进一步处理，将其转移到对应的C++class中，这也可以看成一棵树，但这个树要比Cstruct构成的树要简单的多。数据就经过这样散到聚的过程中达到计算要求。就像一台大的机器，将其拆开，将其中需要的零部件拿出最终组装成自己的机器。将原来那台大机器中的不必要的部分全部剃掉。其拆分得到得那些零件在组装自己机器得时候，并不能直接用，还要对其进行相应得切割和打磨。切割和打磨得过程就是算法具现的过程。

3.3.2系统后端流程图

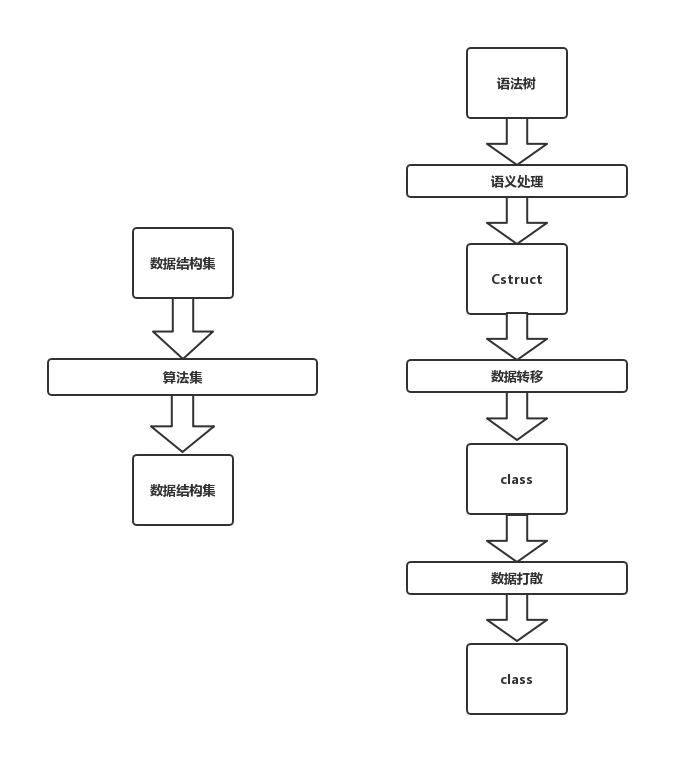


图3-14 系统后端流程图

前端负责将数据拆开，后端负责将数据组装。拆分用的是词法分析和语法分析。组装用的是C++编程语言。通过前后端的通力配合，得到的最终果实就是一个待调用的数据集。

解释器：

解释性语言用的是解释器，解释器的结构是：

词法分析 --- 语法分析 --- 语义分析 --- 执行

常见的解释型语言有java，python

编译器：

编译型语言用的是编译器，编译器的结构式：

词法分析 --- 语法分析 --- 语义分析 --- 中间代码生成 --- 中间代码 --- 代码优化 --- 目标代码生成

因为该项目做的就是从硬件描述语言（和C语言的语法风格有一半的相似之处）描述的文件中提取数据，针对语言类数据的多变和复杂性，所以采用了和解释器和编译一样的前端框架：

词法分析 ---- 语法分析 ---- 语义分析

词法分析：

词法分析将用户输入的Verilog文件转化为记号流返回语法分析器

语法分析：

语法分析将词法分析返回的记号匹配对应的语法树分支。

语义分析：

当某个语法树分支的被触发且归约时，将该语法树分支里的部分或全部数据拷贝存储在其对应的Cstruct中。

后端框架采用的架构是：

Cstruct ---- C++模板容器类。

Cstruct：

Cstruct构成的数据结构树对应语法分析里的语法树，语法树中被触发的且有用的数据全部被拷贝存储在该数据结构树中。

C++模板容器类：

将Cstruct中的数据转移到带对应的链表或图中，以备计算分析是调用。

3.4 类图

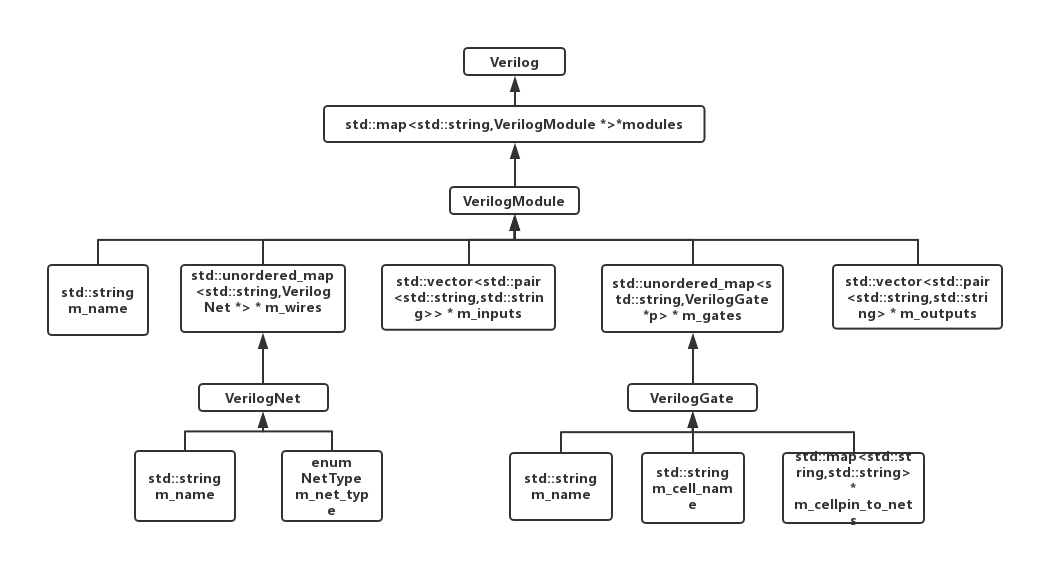


图3-15 类之间的组织关系图

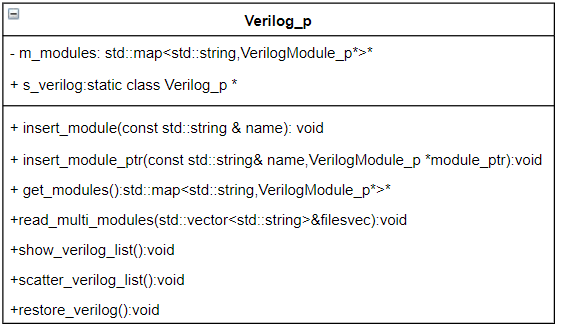


图3-16 类Verilog\_p

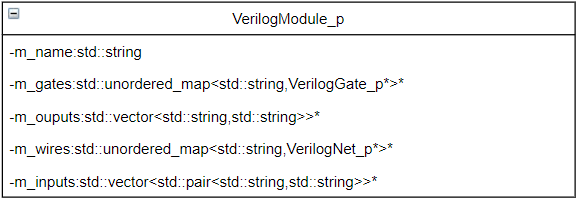


图3-16 类VerilogModule\_p

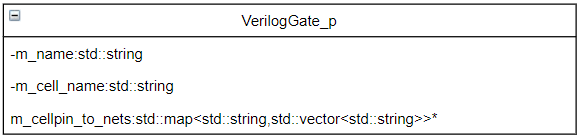


图3-17 类VerilogGate\_p

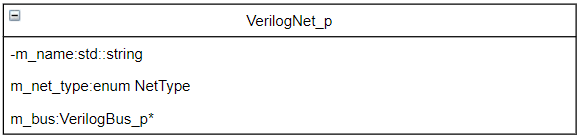


图3-18 类VerilogGate\_p

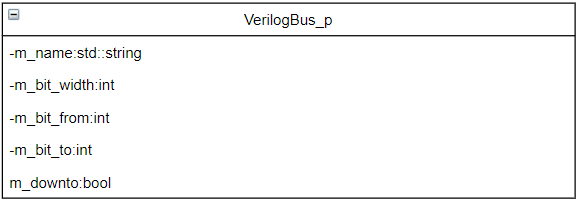


图3-19 类VerilogBus\_p

4 系统详细设计与实现

4.1 词法预处理器

词法预处理器的谷歌定义：词法预处理器是最低级的预处理器，因为它们只需要词法分析。也就是说，它们处理源文本在任何语法分析之前。根据用户自定义的规则，替换掉其他的标记序列。它们通常执行宏替换，头文件替换为原文，和条件编译或包含。

(1)算法分析

宏替换指令：`define

使用说明：指令`define为文本替换创建了一个宏。该指令可以在模块定义的外部和内部使用。当一个文本宏被定义之后，它可以使用源描述通过使用（`）字符，紧跟着宏变量名。预处理器将`text\_macro\_name替换为文本和任何紧跟在它后面的参数。应考虑所有编译器指令与预定义宏名称的冲突，将编译器指令重新定义为宏名称是非法的。

（2）代码实现

{CD\_DEFINE} {BEGIN(in\_define);}

<in\_define>{SIMPLE\_ID} {

yy\_preproc -> scratch = ast\_strdup(yytext);

BEGIN(in\_define\_t);

}

<in\_define\_t>{MACRO\_TEXT} {

// only a '\n'

if(yyleng == 1)

{

// Macro has no value, and is just a newline character.

verilog\_preprocessor\_macro\_define(

yylineno-1,

yy\_preproc -> scratch,

NULL,

0); // -1 to avoid including the newline.

}

else

{

// Macro has a proper value.

verilog\_preprocessor\_macro\_define(

yylineno-1,

yy\_preproc -> scratch,

yytext+1,

yyleng-2); // -1 to avoid including the newline.

}

BEGIN(INITIAL);

}

头文件替换指令：`include

（1）算法分析

使用说明：文件包含（`include）编译指令用于将源文件的全部内容插入另一个编译期间的文件。这个结果就好像包含的头文件的内容出现在`include指令出现的地方。`include编译指令可用于全局或通用的使用定义和任务而不再模块边界内封装的重复代码。

语法说明：

include\_compiler\_directive ::=

`include "filename"

图4-1 Verilog include预处理指令语法

（2）代码实现

{CD\_INCLUDE} {

BEGIN(in\_include);

}

<in\_include>{STRING} {

YY\_BUFFER\_STATE cur = YY\_CURRENT\_BUFFER;

verilog\_include\_directive \* id =

verilog\_preprocessor\_include(yytext,yylineno);

// Now, we need to look for the file, open it as a buffer, and then

// switch to it.

if(id -> file\_found == AST\_TRUE)

{

FILE \* file = fopen(id -> filename, "r");

YY\_BUFFER\_STATE n = yy\_create\_buffer(file, YY\_BUF\_SIZE);

cur -> yy\_bs\_lineno = yylineno;

yy\_switch\_to\_buffer(cur);

yypush\_buffer\_state(n);

BEGIN(INITIAL);

}

else

{

printf("ERROR - Could not find include file %s on line %d\n",

id -> filename, id-> lineNumber);

printf("\tExpect stuff to break now.\n");

}

BEGIN(INITIAL);

}

条件指令：`ifdef，`else，`elsif，`endif，`ifndef

（1）算法分析

使用说明：这些条件编译器编译指令用于包含可选行的Verilog HDL源描述在编译期间。`ifdef 编译指令检查宏变量名的定义，如果宏变量名已经被定义，`ifdef接下的行将被包含，如果宏变量名没有被包含，并且`else指令存在，`else后面的源将被编译。`ifndef编译指令检查宏变量名的定义，如果宏变量名没有被定义，`ifndef指令后面包含的行将被包含。如果宏变量名已经被定义，并且`else指令存在，则`else后面包含的行将被编译。

如果`elsif指令存在（取代`else）,预处理器检查宏变量的定义情况，如果名字存在，`elsif指令后面所包含的行将被编译。`elsif指令是和`else `ifdef…`endif指令平级的。该指令不需要一个与之对应的`endif指令。该指令前面应有`ifdef或`ifndef指令。

（2）代码实现

{CD\_IFDEF} {

BEGIN(in\_ifdef);

}

<in\_ifdef>{SIMPLE\_ID} {

verilog\_preprocessor\_ifdef(yytext,yylineno,AST\_FALSE);

BEGIN(INITIAL);

}

{CD\_IFNDEF} {

BEGIN(in\_ifndef);

}

<in\_ifndef>{SIMPLE\_ID} {

verilog\_preprocessor\_ifdef(yytext,yylineno,AST\_TRUE);

BEGIN(INITIAL);

}

{CD\_ELSIF} {

BEGIN(in\_elseif);

}

<in\_elseif>{SIMPLE\_ID} {

verilog\_preprocessor\_elseif(yytext, yylineno);

BEGIN(INITIAL);

}

{CD\_ELSE} {

verilog\_preprocessor\_else(yylineno);

}

{CD\_ENDIF} {

verilog\_preprocessor\_endif(yylineno);

}

其他指令 ：

`begin\_keywords `celldefine `default\_nettype `end\_keywords `endcelldefine `line `nounconnected\_drive `pragma `resetall `timescale `unconnected\_drive `undef

这指令对应的处理目前还没完善。所划分到其他指令着一块。

4.2词法分析

4.2.1Verilog关键字匹配

（1）算法分析

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| always | and | assign | automatic | begin |
| buf | bufif0 | bufif1 | case | casex |
| casez | cell | comos | config | deassign |
| default | defparam | design | disable | edge |
| else | end | endcase | endconfig | endfunction |
| endgenerate | endmodule | endprimitive | endspecify | endtable |
| endtask | event | for | force | forever |
| fork | function | generate | genvar | highz0 |
| highz1 | if | ifnone | incdir | include |
| initial | inout | input | instance | integer |
| join | large | liblist | library | localparam |
| macromodule | medium | module | nand | negedge |
| nmos | nor | notif0 | notif1 | or |
| output | parameter | pmos | posedge | primitive |
| pull0 | pull1 | pulldown | pullup | pulldown |
| pullup | pulsestyle\_onevent | pulsestype\_ondetect | rcmos | real |
| realtime | reg | release | repeat | rnmos |
| rpmos | rtran | rtranif0 | rtranif1 | scalared |
| showcancelled | signed | small | specify | specparam |
| strong0 | strong1 | supply0 | supply1 | table |
| task | time | tran | tranif0 | tranif1 |
| tri | tri0 | tri1 | triand | trior |
| trireg | unsigned | use | uwire | vectored |
| wait | wand | weak0 | weak1 | while |
| wire | wor | xnor | xor |  |

表4-1 Verilog关键字表

在词法分析需要对Verilog关键字表中出现的关键字进行匹配，每匹配到一个关键字将其对应的标记返回给语法分析器。

以module关键字为例：

在词法分析的定义部分定义关键字module的别名为MODULE

在词法分析的规则部分，定义模式{MODULE} 和其对应的动作{返回标记KW\_MODULE}，该标记会被返回给语法分析器。

（2）代码实现

同样以module关键字为例：

flex定义部分的代码：

MODULE “mdoule”

flex规则部分的代码：

{MODULE} {yylval.keyword = yytext;\ EMIT\_TOKEN(KW\_MODULE);}

4.2.2数字匹配

（1）算法分析

number ::=

decimal\_number

| octal\_number

| binary\_number

| hex\_number

| real\_number

real\_numbera ::= unsigned\_number . unsigned\_number

| unsigned\_number [ . unsigned\_number ] exp [ sign ] unsigned\_number exp ::= e | E

decimal\_number ::= unsigned\_number

| [ size ] decimal\_base unsigned\_number

| [ size ] decimal\_base x\_digit { \_ }

| [ size ] decimal\_base z\_digit { \_ }

binary\_number ::= [ size ] binary\_base binary\_value

octal\_number ::= [ size ] octal\_base octal\_value

hex\_number ::= [ size ] hex\_base hex\_value

sign ::= + | -

size ::= non\_zero\_unsigned\_number

non\_zero\_unsigned\_numbera ::= non\_zero\_decimal\_digit { \_|decimal\_digit}

unsigned\_numbera ::= decimal\_digit { \_ | decimal\_digit }

binary\_valuea ::= binary\_digit { \_ | binary\_digit }

octal\_valuea ::= octal\_digit { \_ | octal\_digit }

hex\_valuea ::= hex\_digit { \_ | hex\_digit }

decimal\_basea ::= '[s|S]d | '[s|S]D

binary\_basea ::= '[s|S]b | '[s|S]B

octal\_basea ::= '[s|S]o | '[s|S]O

hex\_basea ::= '[s|S]h | '[s|S]H

non\_zero\_decimal\_digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

decimal\_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

binary\_digit ::= x\_digit | z\_digit | 0 | 1

octal\_digit ::= x\_digit | z\_digit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

hex\_digit ::= x\_digit | z\_digit | 0 | 1| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

| a | b | c | d | e | f | A | B | C | D | E | F

x\_digit ::= x | X z\_digit ::= z | Z | ?

图4-2 Verilog Number语法

Verilog数字语法描述

从该语法树中可以得知Verilog数字语法树中出现的数字格式比较复杂，以下举几个例子：

可变位数常数：

659 //十进制数

‘h 837FF //16进制数

‘07460 //8进制数

固定位数常数：

4’b1001 //一个4-bit 二进制数

5 ‘D 3 //一个5-bit十进制数

3’b01x //一个3-bit数字，但最小的比特位状态未知

12’hz //一个12-bit位都未知的数

16’hz //一个16-bit位都是高阻态的数

对Verilog数字的词法分析参照Verilog数字语法描述来写，拿8进制数词法分析为例：

词法分析的定义部分：

8进制的前缀用正则表达式：’[sS]?[oO] 来表达，数字用正则表达式：0-7]|{X|{Z}

词法分析的规则部分，

词法分析的规则部分，模式用的是正则表达式的别名，动作部分的实现是将规则部分匹配到的部分放在联合体变量中，再返回记号。

（2）代码实现

词法分析定义部分：

BASE\_OCTAL ‘[sS]?[oO]

DIGIT\_OCTAL [0-7]|{X}|{Z}

词法分析规则部分：

{BASE\_OCTAL} {BEGIN(in\_oct\_val);

yylval.string\_char = strdup(yytext);

EMIT\_TOKEN(OCT\_BASE);}

<in\_oct\_val>{OCT\_VALUE} {BEGIN(INITIAL); yylval.string\_char = strdup(yytext); EMIT\_TOKEN(OCT\_VALUE);}

4.2.3 注释和空格匹配

（1）算法分析

注释匹配：

Verilog硬件描述语言有两种注释格式，一种是行注释，以两个字符（//）开始，以换行符结束。一个段注释，以/\*开始以\*/结束。段注释不能嵌套使用。在段注释中//没有注释的含义，只有普通的两个字符。

以行注释为例：

词法分析的定义部分，其对应的则表达式为：”//”.\*\n

词法分析的规则部分，其对应的规则为其正则表达式的别名。不做任何动作，将注释忽略。

空格匹配：

空格包含的字符有空格符，制表符，换行符，换页符。这些字符一般都进行忽略，除非它们出现在其他的词法标记中。

以换行符为例：

词法分析的定义部分，其对应的正则表达式为NEWLINE “\n”|”\r\n”，NEWLINEWEI为正则表达式”\n”|”\r\n”的别名。

词法分析的规则部分，规则部分为其别名，动作忽略。

（2）代码实现

注释匹配：

词法分析定义部分：COMMENT\_LINE “//”.\*\n

词法分析规则部分：{COMMENT\_LINE} {/\* IGNORE \*/}

4.2.4 字符串匹配

（1）算法分析

字符串由双引号（“”）括起来并包含在一行中的字符序列。字符串在表达式和赋值中用作操作数应被视为无符号常数通过一系列8位ASCII值，一个8位ASCII值代表一个字符。

词法分析定义部分：STRING \”.\*\” ,STRING为\”.\*\”的别名

词法分析规则部分：匹配STRING模式 ，动作代码中执行将正则表达式匹配到的内容放入联合体数据类型中，并返回记号STRING。

（2）代码实现

定义部分代码：STRING \”.\*\”

规则部分代码：{STRING} {yylval.string\_char = yytext;EMIT\_TOKEN(STRING);}

4.2.5 操作符匹配

（1）算法分析

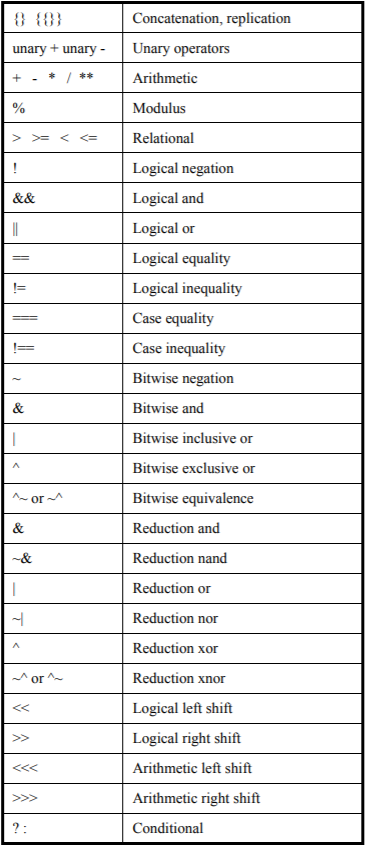


表4-2 Verilog操作符

这些操作符的用法和C语言类似，在词法分析中需要一一对其进行词法匹配并将其返回标记给语法分析器。

以加号为例：

定义部分，STAR “+” ，STAR是+的别名。

规则部分，需要将+的值放在其对应的联合体变量中，再将其对应的标记返回给语法分析器。

（2）代码实现

定义部分实现：PLUS “+”

规则部分：{PLUS} {yylval.enum\_operator=OPERATOR\_PLUS;EMIT\_TOKEN(PLUS);}

4.3 语法分析

Verilog语法总共有九个大类，以下分别以第一类和第八类的一小部分为例进行分析。

4.3.1 module\_declatation

module\_declatation是Source text（图3-4）这个大类下的Module and primitive source text小类下的一部分。module\_declaration代表的是Verilog模块的语法描述。

（1）算法分析

module complexity (inp1,inp2,inp3,out1,out2);

以上是Verilog一个模块的定义部分，用module关键字声明这是一个module，complexity是这个模块的名字，括号里声明的是该模块的输入和输出端口名。

图4-3是module\_declaration的语法树描述：

module\_declaration ::=

{ attribute\_instance } module\_keyword module\_identifier

[ module\_parameter\_port\_list ] list\_of\_ports ; { module\_item }

endmodule

module\_keyword ::= module | macromodule

图4-3 Verilog module\_declaration语法

图4-3中的出现的表示符除module和macromodule可以作为记号之外，其他的标识符都是非终极符。记号（也可成为终结符）代表的是某个正则表达式或Verilog中的关键字，非终结符可以代表多种记号的顺序组合，多种记号和非终结符的组合，多个非终结符的组合。

（2）代码实现

/\*

ast\_module\_declaration \* module\_declaration

module\_declaration module\_decalration

ast\_node\_attributes \* node\_attributes

node\_attributes attribute\_instances

char \* keyword

keyword KW\_MODUL

ast\_identifier identifier

identifier module\_identifier

ast\_list \* list

list module\_parameter\_port\_list

list list\_of\_port\_declarations

char \* string

string SEMICOLON

list non\_port\_module\_item\_os

keyword KW\_ENDMODULE

ast\_module\_declaration \*ast\_new\_module\_declaration(

ast\_node\_attributes \*attributes,

ast\_identifier identifier,

ast\_list \*parameters,

ast\_list \*ports,

ast\_list \*constructs)

list list\_of\_ports

list module\_item\_os

\*/

module\_declaration :

attribute\_instances

module\_keyword

module\_identifier

module\_parameter\_port\_list

list\_of\_ports

SEMICOLON

module\_item\_os

KW\_ENDMODULE{

$$ = ast\_new\_module\_declaration\_module\_item($1,$3,$4,$5,$7);

}

代码实现分为两部分第一部分注释，该部分对语法种用到的非终结符的类型进行了描述。第二部分是module\_declaration的语法实现。

4.3.2 number

如图3-11所示，Numbers是Expressions这个大类中的一个小类。该类下的非终结符number代表Verilog中的所有数字类型。

（1）算法分析

如上图4-2所示，冒号右边出现的标识符有终结符也有非终结符。这些终结符通过词法分析器返回的记号触发。记号流的某一段对应一组记号顺序组合。而冒号左边出现的非终结符一般被其上级的语法分支调用，作为上一级语法分支的一部分。

(2)代码实现

/\*

ast\_number \* number

number number

char \* string

string NUM\_REAL

enum ast\_number\_base BASE\_DECIMAL

enum ast\_number\_representation REP\_BITS

string BIN\_BASE

string BIN\_VALUE

string HEX\_BASE

string HEX\_VALUE

string OCT\_BASE

string OCT\_VALUE

string DEC\_BASE

string UNSIGNED\_NUMBER

number unsigned\_number

ast\_number \* ast\_new\_number (ast\_number\_base base, ast\_number\_representation representation, char \*digits)

\*/

number :

NUM\_REAL{

$$ = ast\_new\_number(BASE\_DECIMAL,REP\_BITS,$1);

free($1);

}

| BIN\_BASE BIN\_VALUE {

//$$ = ast\_new\_number(BASE\_BINARY, REP\_BITS, $2);

$$ = ast\_new\_number\_string(NULL,$1,$2);

free($1);

free($2);

}

| HEX\_BASE HEX\_VALUE {

//$$ = ast\_new\_number(BASE\_HEX, REP\_BITS, $2);

$$ = ast\_new\_number\_string(NULL,$1,$2);

free($1);

free($2);

}

| OCT\_BASE OCT\_VALUE {

//$$ = ast\_new\_number(BASE\_OCTAL, REP\_BITS, $2);

$$ = ast\_new\_number\_string(NULL,$1,$2);

free($1);

free($2);

}

| DEC\_BASE UNSIGNED\_NUMBER{

//$$ = ast\_new\_number(BASE\_DECIMAL, REP\_BITS, $2);

$$ = ast\_new\_number\_string(NULL,$1,$2);

free($1);

free($2);

}

| UNSIGNED\_NUMBER BIN\_BASE BIN\_VALUE {

//$$ = ast\_new\_number(BASE\_BINARY, REP\_BITS, $3);

$$ = ast\_new\_number\_string($1,$2,$3);

free($1);

free($2);

free($3);

}

| UNSIGNED\_NUMBER HEX\_BASE HEX\_VALUE {

//$$ = ast\_new\_number(BASE\_HEX, REP\_BITS, $3);

$$ = ast\_new\_number\_string($1,$2,$3);

free($1);

free($2);

free($3);

}

| UNSIGNED\_NUMBER OCT\_BASE OCT\_VALUE {

//$$ = ast\_new\_number(BASE\_OCTAL, REP\_BITS, $3);

$$ = ast\_new\_number\_string($1,$2,$3);

free($1);

free($2);

free($3);

}

| UNSIGNED\_NUMBER DEC\_BASE UNSIGNED\_NUMBER{

//$$ = ast\_new\_number(BASE\_DECIMAL, REP\_BITS, $3);

$$ = ast\_new\_number\_string($1,$2,$3);

free($1);

free($2);

free($3);

}

| unsigned\_number {$$ = $1;

}

;

/\*

ast\_number \* number

number unsigned\_number

char \* string

string UNSIGNED\_NUMBER

enum ast\_number\_base BASE\_DECIMAL

enum ast\_number\_representation REP\_BITS

ast\_number \* ast\_new\_number (ast\_number\_base base, ast\_number\_representation representation, char \*digits)

\*/

unsigned\_number :

UNSIGNED\_NUMBER {

$$ = ast\_new\_number(BASE\_DECIMAL, REP\_BITS, $1);

free($1);

}

;

冒号右边的记号和非终结符，冒号左边的非终结符都有类型，它们只可以存与他们类型一致的变量值。因为冒号左边只有一个变量，而冒号右边有多个变量，所以冒号左边的变量类型必须是一个联合的变量类型。在该项目中，冒号左边的变量大部分都是结构体类型。这样才能存储冒号右边不同类型不同顺序组合的变量值。冒号右边的花括号存放的就是，语义动作。语义动作就是语义分析，负责将冒号右边的非终结符和记号值（记号编号和记号值是不同的，记号编号是枚举类型，记号值的类型由自己定义），放在冒号右边的变量中。冒号左边的变量可以出现在其他的语法分支的冒号右边，作为待调用的变量。

4.4Cstruct

如表3-1所示，用到的有29个结构体，每个结构体的设计跟其对应的语法树的某个或多个非终结符相对应。但因为每个非终结符代表的记号和记号顺序，非终结符顺序都不同，所以每个结构体的设计都是不同的。现在如4.3所示的介绍一个Source text大类中的结构体，一个Expressions大类中的结构体。

4.4.1 ast\_module\_declaration

struct ast\_module\_declaration\_t{

ast\_metadata meta; //!< Node metadata.

ast\_node\_attributes \* attributes; //!< Tool specific attributes.

ast\_identifier identifier; //!< The name of the module.

ast\_list \* list\_of\_ports;//list of port

ast\_list \* list\_of\_port\_declarations;

ast\_list \* module\_ports; //!< ast\_port\_declaration

ast\_list \* always\_blocks; //!< ast\_statement\_block

ast\_list \* continuous\_assignments; //!< ast\_single\_assignment

ast\_list \* event\_declarations; //!< ast\_var\_declaration

ast\_list \* function\_declarations; //!< ast\_task\_declaration

ast\_list \* gate\_instantiations; //!< ast\_gate\_instantiation

ast\_list \* genvar\_declarations; //!< ast\_var\_declaration

ast\_list \* generate\_blocks; //!< ast\_generate\_block

ast\_list \* initial\_blocks; //!< ast\_statement\_block

ast\_list \* integer\_declarations; //!< ast\_var\_declaration

ast\_list \* local\_parameters; //!< ast\_parameter\_declaration

ast\_list \* module\_instantiations; //!< ast\_module\_instantiation

ast\_list \* module\_parameters; //!< ast\_parameter\_declaration

ast\_list \* net\_declarations; //!< ast\_net\_declaration

ast\_list \* parameter\_overrides; //!< ast\_single\_assignment

ast\_list \* real\_declarations; //!< ast\_var\_declaration

ast\_list \* realtime\_declarations; //!< ast\_var\_declaration

ast\_list \* reg\_declarations; //!< ast\_reg\_declaration

ast\_list \* specify\_blocks; //!< Not Supported

ast\_list \* specparams; //!< ast\_parameter\_declaration

ast\_list \* task\_declarations; //!< ast\_task\_declaration

ast\_list \* time\_declarations; //!< ast\_var\_declaration

ast\_list \* udp\_instantiations; //!< ast\_udp\_instantiation

} ;

typedef struct ast\_module\_declaration\_t ast\_module\_declaration;

从以上的代码可以给看出，ast\_module\_declaration这个Cstruct包含结构体，结构体指针和链表指针。每个变量都有其各自的作用。链表指针后的注释是说明这个链表中存的是哪个结构体类型的变量。从上一小节的内容可以得知module\_declation的数据类型就是ast\_module\_declaration。module\_declaration中的值来自其冒号右边的终结符和非终结符。

4.4.2 ast\_number

typedef struct ast\_number\_t ast\_number

typedef enum ast\_number\_base\_e{

BASE\_BINARY,

BASE\_OCTAL,

BASE\_DECIMAL,

BASE\_HEX

} ast\_number\_base;

typedef enum ast\_number\_representation\_e{

REP\_BITS, //!< For numbers specified per digit.

REP\_INTEGER, //!< For "Integer" typed numbers"

REP\_FLOAT //!< For "real" typed numbers.

} ast\_number\_representation;

/\*!

struct ast\_number\_t{

ast\_metadata meta; //!< Node metadata.

unsigned int width; //!< Width of the number in bits.

ast\_number\_base base; //!< Hex, octal, binary, decimal.

ast\_number\_representation representation; //!< How is it expressed?

union{

char \* as\_bits;

float as\_float;

int as\_int;

};

};

ast\_number是Expressions大类中的一个结构体，参考图4-2，ast\_number是与非终结符number相对应的Cstruct。枚举类型ast\_number\_base\_e代表的是数字进制，十进制，十六进制等。枚举类型ast\_number\_representation\_e代表的是数字类型。这样图4-2语法树的分支里的数据都可以存在ast\_number中。及保存其数据的同时有保存其格式。

5 系统运行与效果分析

5.1 界面设计概要

因为该项目完成的任务是运行在服务器上的软件的子模块，功能主要是为软件提供运算数据。没有qt和网页界面只有黑窗口显示的数据。

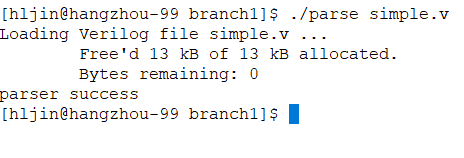


图5-1 程序运行效果图

从图5-1中可以看出，可执行程序为parse，输入文件为一个Verilog语言描述的输入文件simple.v。数据提取完成后程序会将提取过程申请的13kB内存给释放掉，然后提示解析成功。但看到解析成功字样时，所有的数据全部存到一个静态指针指向的内存块中。

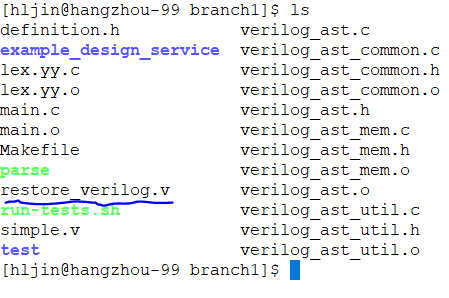


图5-2 输出文件效果图

为了验证数据正确性，程序将存在待调用静态指针指向的内存中的数据还原成一个Verilog格式的文件，这个文件被命名为restore\_verilog.v。

5.2运行效果分析

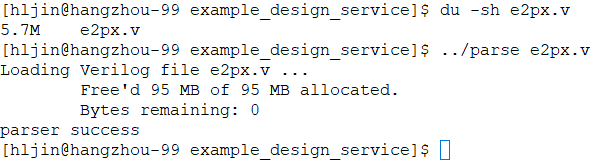


图5-3 程序内存消耗分析图

程序在运行过程需要消耗文件大小是几倍到十几倍的内存，而且这些内存必须得是数据全部提取完毕统一释放的，如果输入文件是以GB为单位的文件，那么程序在内存这方面得开销会很大，对本程序的其他软件模块会造成不好影响。

目前的一个优化方案就是提取一部分释放一部分这样才能减轻整个软件系统运行的负担。但是这样又会带来一个工程上的问题，用于存放Verilog文件的Cstruct嵌套层数有五层之多，而且里面除了底层之外都是指针调用，完成其内存释放函数的难度会很大。所以目前的这一块的工作打算放在开发周期的后期的代码优化那一个时间段来做。

6 系统测试

6.1 测试方法

在专业软件中，特别是芯片设计这样的软件，其数据格式和硬件知识结核紧密，需要对芯片设计相对熟练的人才能知晓对应数据格式下的数据的含义。芯片设计软件中对芯片数据的完整度和准确度要求极高。比如Verilog网表数据，代表的是芯片内部分连线数据，如果代表某一跟连线的数据出错，那么整个功耗软件跑下来的结果可能会错的离谱。所以势必确保每一个比特位的数据都不能错。提取之后的数据需要经过严格的测试，确保这个数据提取模块提取数据完全正确才能接入其他模块以供调用，不然接入也没有意义。失之毫厘，差之千里。

该项目采取测试分为两个方面或者两个方法：

一个是从提取得到的数据的正确性来确定程序本身的正确性。一个是不看程序跑的结果看程序本身的工程质量，来确定程序的正确性。

测试方法1：数据还原测试法。

该方法使用了两次：

第一次测试是将Cstruct（原数据，去除了标点空格等助记符），Verilog语法格式反向还原为Verilog文件（还原的过程中要加入标点），用python脚本对比还原的Verilog文件和原文件。脚本会自动忽略文件中的空格，只按顺序对比文件中字符是否相同。

第二次测试是将C++class中的数据（已经打散的数据），还原成打散的Verilog文件。这已经打散的文件和原文件里面蕴含的信息量虽然一样，但格式已经完全不同，所以不能用脚本来测试。我们采用的是用友商的软件（PrimeTime 是Synopsys公司旗下的静态时序分析软件，一般芯片设计工程师都是用这款软件做静态时序分析）。因为打散和原文件的数据信息量是一样的而且它们的语法标准都是经过综合工具得到的门级Verilog数据，理论上用这两个文件跑出来的分析数据都是一样的。跑出来的分析结果我们分析的是其中的path数据，如果这个数据相同那就能表明输入文件的信息量一样的。

测试方法2：内存使用情况检测

使用valgrind内存使用情况检测工具来检测，程序在内存使用上的问题。因为程序的核心都是指针和结构体，嵌套了很多层。内存问题会很突出。需要重点关注。

6.2 测试方案及计划

6.2.1 系统功能介绍

该系统的功能是将Verilog文件里的数据提取打散，将其存入公司自定义的数据结构中。

数据提取分为前端和后端：

提取前端是编译器或解释器的前端，有预处理模块，词法分析模块，语法分析模块。

提取后端的核心是数据结构到数据结构的转移，并且在转移的过程将对数据做加工。

6.2.2 测试目的

芯片设计过程中，芯片内部每一条布线都不能错，电源线，地线，信号线。这些线是芯片内部每一计算存储单元的生命线，有一条线出错就意味和这条线线相关联的单元将不能正常工作，连锁反应，整块芯片可能都因此而无法正常运行。比这更严重的是本来是错的数据没有经过严格的测试当成正确的数据用，那么软件后面的模块不管怎们分析都得不到正常的数据。这将会拉长开发周期，而且这些时间是没有必要损耗的。

所以该项目对于数据的内容的完整和正确要求极高，多以其对应的测试必须到位，这样才能确保数据的正确性，为后面的模块开发提供扎实的数据支撑。

6.2.3 测试范围

测试的数据是经过IC Compiler综合得到的门级网表数据。综合得到的Verilog文件具有统一的语法格式。

6.2.4 测试进度安排

|  |  |  |  |
| --- | --- | --- | --- |
| 应用程序名称 | 模块名称 | 描述 | 优先级 |
| Verilog网表数据提取器 | Cstruct | 从Cstruct将数据还原成Verilog文件，用Python脚本对比还原文件与源文件的差异，验证数据在Cstruct结构体中数据的正确性。 | 1 |
| C++class | 将已经打散的数据从C++class数据结构中还原成Verilg文件。用PrimeTime工具分别跑还原的文件和原文件比较分析得到的path数据。比较数据的差异，如果数据一样，数据打散成功。 | 1 |

表6-1 测试进度安排表

6.3 测试过程及结果分析

6.3.1 测试用例设计

（1）比较从Cstruct中还原的Verilog数据和原Verilog数据，

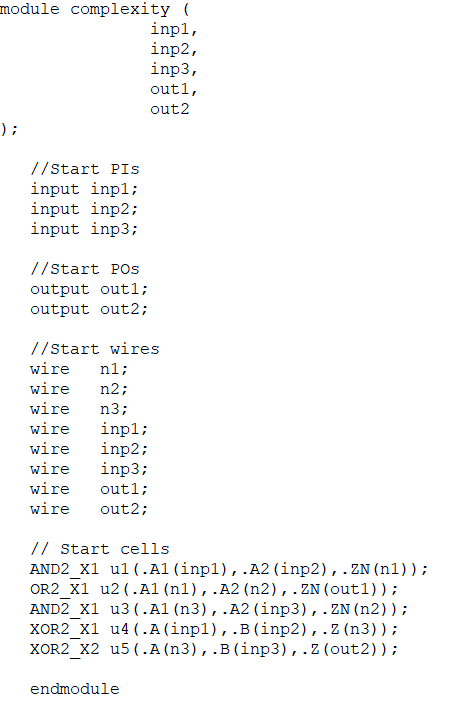


图6-1原数据图1

从图6-1可以看出，输入端口有三个，输出端口有两个。wire声明的是net类型也就是网线类型。是cell和cell之间的连线。// Start cells注释以下的部分是各cell之间的连接关系。AND\_X1 u1(.A1(inp1),.A2(inp2),ZN(n1));AND\_X1代表这个cell是与门，把这个cell命名为u1，括号内部表达是u1这个cell上的pin与线网的连接关系。本项目的任务就是将这个文件作为输入文件从这个文件中提取数据。Verilog是以module为单位存取网表数据的，图6-1所示的就是一个module，这个module会通过端口和线网与其他的module进行拼接，最终形成芯片内部各cell的连接关系数据，也就是网表数据。图6-1是一个简单module的示例，实际要提取的例子module数是以万为单位的。在Cstruct结构体中，并不关心各module之间的关系，只是负责存取各module的数据，在C++class才会开始考虑各module之间的连接关系。

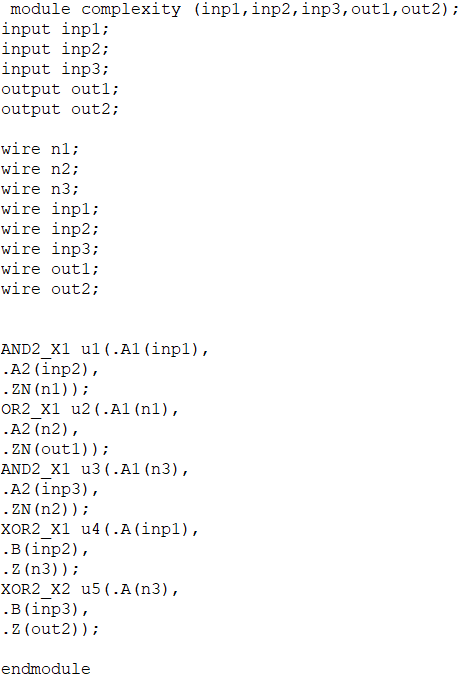


图6-2 还原数据图1

图6-2所示的是从Cstruct还原的数据。

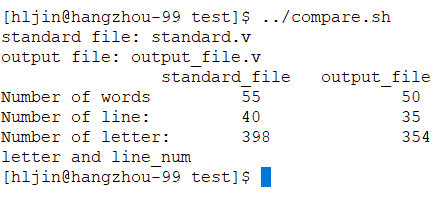


图6-3 脚本比较测试图

从脚本跑的结果来看，原文件只比还原文件多了注释，除此之外，两文件的信息量是一样的。

（2）比较从C++class中还原的数据和原数据的信息量

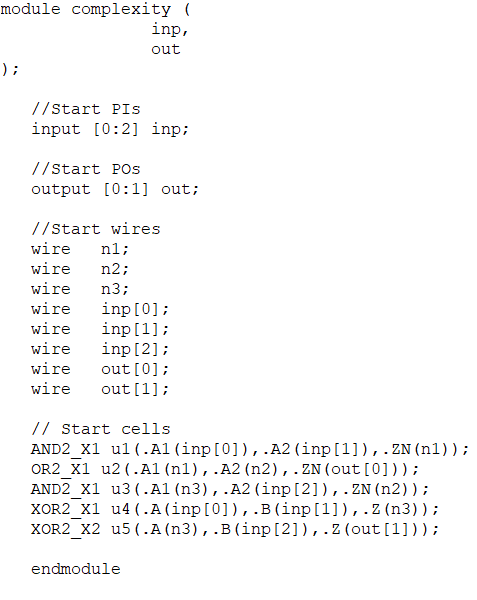


图6-4 原数据图2

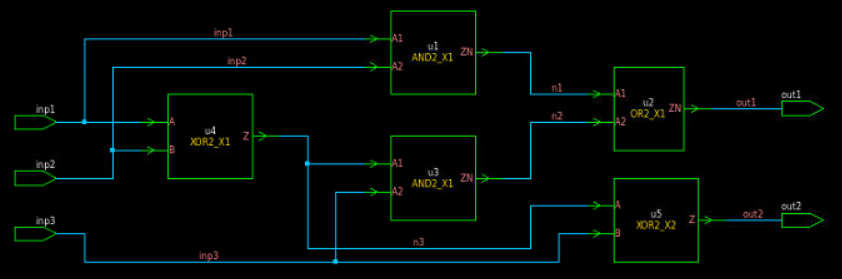


图6-5 网表数据展示图

图6-4的信息数据量和图6-1的信息数据量相同，但是格式变得更复杂了一点。input和output类型变成了bus类型。程序需要将图6-4的数据还原成图6-1的格式。还原后的结果如图6-5所示。可以发现图6-5中的中括号里的数据已经全部打散。图6-5的数据就不再是从Cstruct中还原数据了而是要从C++class中还原数据。如果还是从Cstruct还原数据的话，只能还原成和图6-4一样的数据，只是没有了注释。图6-4中的数据经过特定算法的处理最终可以得到图6-5所示数据格式。打散的算法全部在C++class这个模块中。

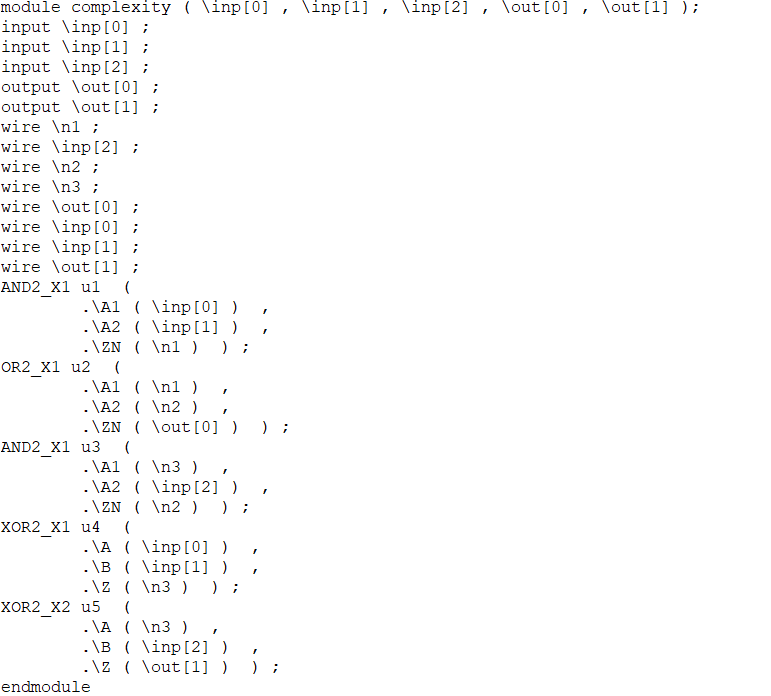


图6-5 还原数据图2

比较这两个文件可以发现，文件中[2:0]等中括号里的内容全部打散。这是后就不能用上一个测试中的python脚本来跑了。得用第三方软件PrimeTime来分别跑这两个文件然后比较这两个文件跑出的分析报告。数据打散成功。

（3）内存测试

使用valgrind内存检测工具来检测本程序在运行的过程存在的内存漏洞，可以看出程序存在内存问题。测试过程需要将Cstrcut和C++class分开来单独测试。因为这两个模块都有大量的内存申请，其中C++class的最终的计算C++class模块是不能释放的得其他模块调用完才能释放。所以需要独立开测试，先测试Cstruct，确保Cstruct这个模块中没有问题在接入C++class模块。这和既要金山银山又要绿水青山得思想相类似。金山银山代指提取得到的正确结果，内存和其他资源消耗代指对环境的负担。最好的结果就是消耗的资源，系统能轻松提供且不会给系统带来负担，得到结果也是达到工程要求的。现在的工程进度是已经是金山银山有了，但绿水青山有没有达到还要进一步测试。图6-6是目前系统用valgrind初步的检测报告详细的检测报好，还在推进中。

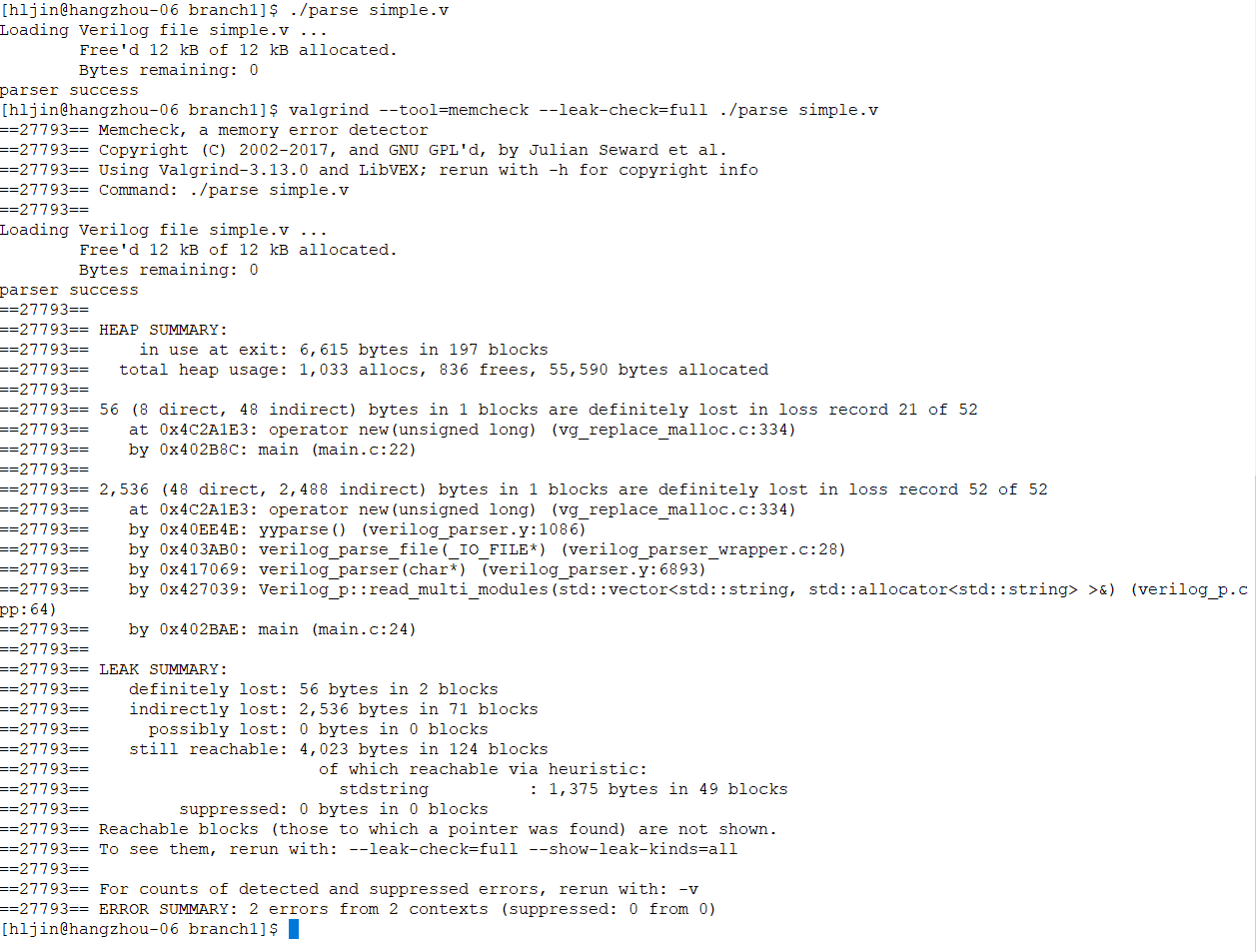


图6-6 内存检测

6.3.2 测试结果分析

通过数据还原测试和内存测试，可以抓出系统绝大数bug，一些隐藏的bug需要更细致的测试。单元测试和极限测试，健壮性测试，暴力测试，在后期都要进行。

从目前测试的结果来看，结果是正确的，但是其他的一些隐性的细节问题随着项目的推进和测试工作的展开，程序会逐步走向成熟完善。

7 总结与展望

7.1 总结

该项目锻炼的是多方位的能力。

新的业务的熟悉：

刚进入到行芯科技这个公司时我对芯片设计这个领域的知识是陌生的，不知道芯片设计的流程是什么，每个流程需要哪些知识，哪些开发工具都一无所知。慢慢的摸索到现在已经有了些模糊的认识，也对芯片设计过程中产生的需要的各种文件格式有了一定的了解。

新技术的熟悉：

网表数据的数据量和数据的复杂度用python和其他脚本或语言都不是很合适，必须得借助词法语法分析器，而且最好是C实现的词法和语法分析器，这样在性能上才能达到要求。最终选定了Flex和Bison，刚开始这对于我来说完全是一个全新的技术和工具，再此之前我只知道有编译器和编译原理这么一个东西，但是并没有什么亲密的接触。但这个东西确实重要，也避不开，所以慢慢学，先从简单的例子开始一点点的积累，到现在能够对一门语言进行分解并获得数据。

现在回过头来看这些业务知识和编程和工具用法，都是磕磕碰碰走了很多弯路，但是弯路只要能能及时的走过去，弯路也会带来价值。所以自己也一直和自己说，熬一熬拖一拖，在争取到的时间和空间中快速成长，最终正面直接和问题交手。中国和美国的关系是这样，个人的成长也是这样。碰到陌生的东西，遇到某个自己无从下手的问题，不要慌张，在别人的怀疑和各种压力面前撑起一个给自己成长的时间和空间，坚持住，希望就会来临，最终问题慢慢消散，好像没有发生一样。

7.2 展望

未来风雨未可期，只有心中舟一许。

不断的解决问题，学习，让它不在是问题，循环往复。新的问题不断产生，旧的问题不断的被消灭，价值也在缓慢积累。

未来，希望自己能够在这个波澜壮阔的时代里做一个船夫，一个技术高超的船夫，在任何浪头面前撑好船，保护船上的人。度他们过河，也度自己过河。

参考文献

[1]flex和bison在软PLC编译器中的应用[J]. 田文琦,于东,高伟,纪元. 微计算机信息. 2009(19)

[2]Linux环境中使用Flex、Bison进行SQL语法分析[J]. 孙兆玉,朱鸿宇,黄宇光. 电脑编程技巧与维护. 2007(02)

[3]混合信号边界扫描系统EDIF网表文件解析器的实现[J]. 颜学龙,李艳. 计算机测量与控制. 2006(06)

[4]基于Lex&Yacc的电火花加工译码系统[J].郑君民,王振龙,赵万生.电加工与模具. 2005(06)

[5]在Visual C++集成开发环境下应用flex和bison开发编译器[J].熊伟,戴果.战术导弹控制技术. 2004 (04)

[6][微纳级SRAM器件单粒子效应理论模拟研究](http://kns.cnki.net/kcms/detail/detail.aspx?filename=1014042564.nh&dbcode=CDFD&dbname=CDFD2014&v=" \t "http://kns.cnki.net/kcms/detail/frame/kcmstarget)[D].耿超.中国科学院研究生院（近代物理研究所）2014

[7]基于Lex&Yacc的PLC指令编译器的研究与实现[J].张少坤,游有鹏.可编程控制器与工厂自动化. 2009(05)

[8]基于flex和bison的编译器开发[J].熊伟,戴果.兵工自动化. 2004(01)

[9]编译原理及编译程序构造[M].北京航空航天大学出版社,高仲仪,金茂忠编, 1990

[10][可逆逻辑电路综合方法研究](http://kns.cnki.net/kcms/detail/detail.aspx?filename=1011292184.nh&dbcode=CMFD&dbname=CMFD2011&v=" \t "http://kns.cnki.net/kcms/detail/frame/kcmstarget)[D].冯冉南京航空航天大学2011

[11]可逆逻辑综合[M].科学出版社,管致锦, 2011

[12]EDA技术实用教程[M]. 科学出版社 , 潘松, 2006

[13]Compilers:Principles, Techniques, and Tools. Alfred V Aho,Monica S.Lam,Ravi Sethi. . 2006

[14]Error-Rate Estimation Combining SEE Static Cross-section Predictions and Fault-Injections Performed on HDL-Based Designs.W.Mansour,R.Velazco,G.Hubert. IEEE Transactions on Nuclear Science.2013

[15]A low-cost radiation hardened flip-flop. Lin Y,Zwolinski M,Halak B. Design Automation and Test in Europe Conference and Exhibition (DATE) 2014

[16]Gallier J H Logic for computer science: foundations of automatic theorem proving[M].Courier Dover Pulications, 2015

[17]Zhang N,Duan z,Tian C.A mechanism of function calls in MSVL[J].Theoretical Computer Science,2016,654:11-25.

[18]Zhang N,Duan z,Tian C.Model checking concurrent systems with MSVL[J].Science China Information Sciences,2016,59(11):118101

[19]Wang M, Tian D,Duan Z.Full regular temporal property verification as dynamic program execcution[C]//Software Engineering Companion(ICSE-C),2017 IEEE/ACM 39th International Conference on.IEEE,2017:2006-2008.

[20]Jobn Levine.Flex & Bison.USA:O’Reilly,2009,19-21

[21]”IEEE Std 1364-2001”,IEEE Standard for Verilog Hardware Description Language.

[22]”IEEE Std 1364-2005”,IEEE Standard for Verilog Hardware Description Language

致谢

感谢母校！南昌航空大学，家乡很多人都以为我是造飞机的，但是我知道，我不会但我学校以前会，我也因此而自豪。刚进学校的时候对于我们学校的校训“日新自强，知行合一”没什么感觉，但现在临近毕业，也在外面工作了六个来月，回过头来看这句话，发现自己骨子里已经打上了校训的烙印。日日要学习更新更正自己，虽然不够强大但要自己要有一个让自己强大的心，自新和自强，知道也要做到。这样才能有韧性和潜力。谢谢您，母校，谢谢您给了每一个真正感受到您的教诲昌航学子一个强大的法宝。

感谢昌航的每一位老师！在昌航读书的那两年，可以真心感觉出昌航的老师都很务实，知识的授予勤恳踏实。从主课到副课到选修，每一个老师给人的感觉都很好，很喜欢上他们的课。主课的每一个老师，都巴不得我们有多少学多少，一副怕我们在社会上吃亏的样子。副课的老师不管同学们听课的态度怎么样，都很认真，可以说用一丝不苟来形容。选修课的老师都比较慢节奏，会教一些情感生活上的东西，教我们怎么把生活过的有趣。谢谢你们！

感谢东软的每一位老师！东软的老师带我们做了很多的小项目，在做这些小项目的时候发生过很多个小故事，有自己与同学之间的，也有老师与我们之间的，很有趣，生活中充满了可爱的人，但需要一双发现美的眼睛。

最后感谢的是行芯科技的每一位同事，当然还有青哥（boss）。离开了学校，进入了公司和一群志同道合的人在一起用技术擦出火花。工作中所有的问题都会回到原点，怎么解决它。每个人都会有自己冲锋进攻的垛口，当确实碰到瓶颈或需要帮助的时候公司的每一位同事又会从他的工作中挤出时间给你必要的帮助。这个项目也是经过老师（公司里的George师傅）耐心培养，指导和三个浙大的同事通力完成的。谢谢你们！