单位代码 10006
学　　号 35060107
分 类 号 TP302
密　　级

北京航空航天大學
BEIHANG UNIVERSITY

毕业设计(论文)

一种面向部件级模拟的体系结构描述语言的设计与实现

院（系）名称 计算机学院
专 业 名 称 计算机科学与技术
学 生 姓 名 吴兴博
指 导 教 师 高小鹏

2009 年 6 月

一种面向部件级模拟的体系结构描述语言的设计与实现
吴兴博
北京航空航天大学

一种面向部件级模拟的体系结构描述语言的设计与实现

吴兴博

北京航空航天大学

北京航空航天大学

本科毕业设计（论文）任务书

Ⅰ、毕业设计（论文）题目：

一种面向部件级模拟的体系结构描述语言的设计与实现

Ⅱ、毕业设计（论文）使用的原始资料（数据）及设计技术要求：

原始数据：
1. Lex 与 Yacc语言规范。
2. AADL语言基本特性。
3. Verilog HDL 语法定义。

设计技术要求：
1. 系统结构描述语言标准定义。
2. 实现将描述语言转化为模拟器的工具（程序）。
3. 语言分析有一定的正确性检查、给出有用的分析结果。
4. 生成的模拟器能够有较快的运行速度。
5. 具体的模拟器设计实例。

Ⅲ、毕业设计（论文）工作内容：

设计面向部件级模拟的体系结构描述语言，开发对应的处理软件。

Ⅳ、主要参考资料：

[1] 高小鹏, 牛建伟, 杨钦. 《加速部件及其应用》课题研制计划报告 [R]. 北京航空航天大学计算机学院, 2008

[2] Peter H. Feiler, David P. Gluch, John J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction [R]. Pittsburgh U.S.: Carnegie Mellon University, 2006

[3] bert hubert. Lex and YACC primer/HOWTO [M/OL]. (2002-04-20)[2009-2-23].

[4] David Pellerin, Scott Thibault. 使用C语言FPGA编程 [M]. 北京: 机械工业出版社, 2007.

[5] Clive Maxfield. FPGA设计指南：器件、工具和流程 [M]. 北京: 人民邮电出版社, 2007.

[6] AADL (Architecture Analysis & Design Language)[EB/OL]. (2007-02-09) [2009-2-23]. http://www.axlog.fr/aadl/aadl_en.html

计算机 学院（系） 计算机科学与技术 专业类 350601 班

学生 吴兴博

毕业设计（论文）时间：2009 年 3 月 1 日至 2009 年 6 月 1 日

答辩时间：2008 年 6 月 10 日

成　　绩：

指导教师：高小鹏

兼职教师或答疑教师（并指出所负责部分）：

计算机科学与技术系（教研室） 主任（签字）：

本人声明

我声明，本论文及其研究工作是由本人在导师指导下独立完成的，在完成论文时所利用的一切资料均已在参考文献中列出。

作者：吴兴博

签字：

时间：2009 年 6 月

一种面向部件级模拟的体系结构描述语言的设计与实现

学　　生：吴兴博

指导教师：高小鹏

摘　　要

现有的硬件系统设计流程主要包括代码编写、逻辑综合、行为仿真、布局与布线和时序仿真等几个步骤。在寄存器传输级描述系统复杂度较高，也比较耗时。为了较快地对系统进行原型设计，在很多设计中采用了编写模拟器的方式快速设计和验证系统结构。模拟器一般仅对功能模拟，以减少实现的难度。一些基于高级语言的设计方法提供了更加方便的建模与验证方法，但是这些方法主要是面向综合的，在验证上往往存在复杂度较高的问题。

本论文面向部件级的体系结构建模，以自动生成模拟器为目标，设计了一种体系结构描述语言及模拟器生成工具。这种语言分为行为描述和结构描述两种层次，能够通过简单的描述组织复杂的体系结构模拟器。由于将功能定位于模拟，在语言的设计中受到的限制较少，语言形式简单而灵活。模拟器生成工具有良好的用户界面，不仅能将用户描述转化为模拟器代码，还能提供多种便于设计和分析的反馈。将这套方法加入到现有的设计流程中，能够极大地简化原型系统的验证过程，从而提高设计质量。

本文首先介绍了 FPGA 设计和验证的相关技术，确定了研究目标，然后描述了体系结构描述语言的设计、模拟器生成工具的整体设计和详细的模块实现，最后是工具的测试以及结果的分析。

关键词：模拟器，验证，描述语言，代码生成

The Design and Implementation of A component-level-simulation Oriented Architecture Description Language

Author : Wu Xing-bo

Tutor : Gao Xiao-peng

Abstract

Popular hardware design flow always includes several steps such as coding, logic synthesis, behavior simulation, place & route, timing simulation. It is complex and time-consuming when describe a system in register transfer level. To speed up the design of a prototype system, simulators are always been used to design and verify the architecture. Generally, simulators only perform the functional features of the system to reduce the complicacy of the coding. Some approaches based on some kind of high level languages can perform better modeling and verification. But these approaches are always used for synthesis. It seems to be complex if you only use them to perform modeling.

This paper is aimed to component level architecture modeling and simulator auto-generation. An architecture description language has been designed with its tool. This language has two abstract levels, the behavioral level and the structural level. Using this language, a complex system can be organized by writing simple description code. Because of targeting the function to simulation, the form of the language is more simple and flexible than other languages. The tool has a well-designed user interface. It not only generates the user description to the code of a simulator, but also output feedbacks for design and analysis. It can optimize the verification and improve the design quality by adding this approach to popular design flow.

This paper first introduced the technology of FPGA design and verification, discussed the aim of research, then described the design of the Architecture Description Language, architecture of the simulator generator and its detailed implementation, finally described the testing and the analysis.

Keywords： Simulator, Verification, Description language, Code generation

目　　录

1 绪论

1.1 课题来源

本题目来源于 863 课题《千万亿次高效能计算机关键技术》中的《研究基于FPGA技术的应用加速器(FPGA-based Application Accelerator)体系结构》[1]子课题。其研究的应用包括：

1. 石油勘探领域中广泛用于“地震成像”的 PSDM（三维叠前深度偏移）算法。目前 PSDM 主要采用 Kirchhoff 积分法。该算法运行需要耗费大量计算时间和存储空间。
2. LINPACK 是目前国际上用于衡量高性能计算机性能指标的标准算法库。LINPACK 包含了大量的标准算法。研究其中 2 个具有典型代表性的算法的加速，在一定程度上能够有效的标明系统的性能指标。

在算法加速的研究中需要一套自动生成模拟器的方法辅助体系结构设计，本论文的目的是探讨一种自动生成模拟器的设计流程，包括描述语言和自动生成模拟器的工具。

1.2 研究背景

FPGA 设计过程漫长，设计与验证往往需要迭代进行。为了提高设计效率，缩短设计与验证的时间，对于一些设计可以采用编写模拟器的方法进行功能性验证。本文中的“模拟器”是指一类计算机程序，它通过模拟硬件的运行，实现硬件的功能，用于验证硬件设计。模拟器是进行硬件系统设计尤其是大规模硬件系统设计中必不可少的环节。为了提高设计的效率和提高设计速度，为系统编写模拟器是一项必要的工作。

在以往的设计中，模拟器一般由程序员直接编写计算机程序，人为地组织模拟器结构，设计硬件行为到软件功能上的对应关系和调度顺序。对于较大的系统，硬件的空间复杂度和时间复杂度使得部件的数量过多、调度排序的难度增大，模拟器的设计难度就会非常大。模拟器的编写如果和硬件一样复杂，则不能有效地在硬件设计开始前为系统提供评估和参考的作用。模拟器的编写如果能够被简化，整个系统的开发过程都会因其

而得到改善。

提高描述的抽象级别，即使用更少的描述刻画更多的实体。使用一种抽象级别更高的系统结构描述语言和相应的转换工具，自动地生成模拟器代码，能够简化编写模拟器的复杂度。用形式化的描述语言描述模拟器，和手工编写代码相比，从结构到实现的映射过程是自动的，人的工作更集中和精简，从而减少工作量，提高设计效率。

现有的一些提高抽象级别的设计方法具有一些模拟仿真的功能，但是由于工具的用途并非面向模拟，这些工具在生成模拟器这一功能上并不能满足项目的需求。

现有的模拟器设计研究较多地关注于处理器的设计，处理器具有粗粒度和结构复用少的特点，这类模拟器结构比较简单，不需要自动生成。而面对数学运算这类细粒度、部件数量多、部件类型少的设计，模拟器的自动生成功能才能明显地减少工作量，提高工作效率。

课题研究的应用加速器主要面向数学运算的加速，现有的硬件设计思想、工具或语言都处在发展阶段[21]。模拟器生成所需的功能，依靠现有的技术较难满足。设计一套具有描述、建模分析以及模拟验证功能的语言和配套工具是有必要进行的工作。

1.3 研究目标与内容

本论文的研究目标是设计一种可用于建模、分析和生成模拟环境的系统结构描述语言 ADL（Architecture Description Language），并且开发一套对应此语言的转化工具。用户使用 ADL 描述系统模型，然后使用转化工具来处理并生成对应的模拟器代码。通过将此工具加入到 FPGA 设计流程中，能够简化设计过程，将设计与实现过程分离，缩短设计阶段的验证周期，从而提高设计的速度。

研究内容包括：定义描述语言 ADL、设计与实现语言的处理和模拟器生成工具。

1.3.1 定义描述语言

HDL 语言的建模层次在寄存器传输级 RTL（Register Transfer Level）。这一层次的设计是面向实现的。直接在这一层次建模，使得设计、评估和实现成为一体，设计过程紧密耦合，使得设计、验证周期很长（涵盖整个流程）。为了快速进行设计、验证，快

速确定系统原型，需要简化实现的过程，最重要的研究切入点就在语言的抽象级别和描述能力上。

在 RTL 的描述层次进行设计，描述要具体到模块的实现。所以基于 RTL 的系统设计往往是自底向上的。但从系统的角度进行自顶向下的建模，设计者根据模块的外特性就可以进行必要的分析和评估。通过将模块表现为“功能”而非“实现”，就有可能减少实现内部逻辑所带来的工作负担。

设计一种新的体系结构描述语言，使设计者方便地定义系统中的模块，模块的接口、功能以及模块间的连接关系。在科学计算这一领域的硬件结构，有数据类型整齐、控制结构简单等特点。在语言的设计中可以利用这些特点简化用户的描述难度。

现有的硬件设计语言 Verilog HDL 和一般的建模语言 AADL 和 UML 等都是语言设计时可以参考的对象。另外，和一些常用的语言采用类似的设计也有利于使用者的学习。

1.3.2 语言处理和模拟器生成工具

语言的处理分为以下几个步骤：

1. 语言的扫描输入，词法分析、语法分析。
2. 模块基本分析，模块的连接关系、各种错误检测、模块依赖检测。
3. 模拟调度，包括模拟可行性分析、调度排序。
4. 模拟器生成。

模块（部件）是在空间中分布、连接的。如何在模拟环境中模拟这些模块的行为，保证还原它们的并发、时序特性也是本题目中很重要的一部分。需要将硬件中并行执行的模块映射为软件的调用顺序，将硬件中的触发、激励过程对应到软件的代码调用，如何使这一转化正确、高效是必须要考虑的问题。

通过语言分析，生成系统中模块的结构和分布，安排模拟的执行顺序和功能调用。对于模块的输入输出，需要建立数据传输的关系，如果是组合逻辑模块，其前导和后置模块的执行顺序也需要有序排列，检查是否出现了错误的连接，是否有闭环等各种可能的问题。

处理完成得到正确的模块关系，需要生成模拟器的代码，通过编译器的编译能够生

成可运行的模拟器程序。

1.3.3 处理工具的其它附加功能

在语言处理的主要流程中，为了方便用户得到设计所需要的信息，给用户提供足够的反馈也是工具中必要的部分。为了方便结构验证，系统需要反馈出系统生成的逻辑结构，以图形化的方式呈现给用户。调试功能的支持也是必不可少的，给用户呈现部件和端口的实时信息，能够便于发现设计的错误，快速修正设计。对于已经定义好的模块结构，可以为用户自动生成对应的 verilog 代码，方便用户在验证后直接对模块进行 HDL 实现。

1.4 本文组织结构

本各章内容安排如下：

- 第一章 绪论

  本章主要介绍了本论文的来源，分析了目前相关的研究背景，简要阐述了本论文的研究内容以及研究目标，列出了本文的组织结构。

- 第二章 相关技术分析

  本章主要介绍本论文在设计实现过程中相关各种技术、规范和工具。内容主要包括 FPGA 技术、HDL 设计方法、高级设计方法、C 语言、Lex 与 Yacc 工具和 Graphviz 工具等。

- 第三章 体系结构描述语言的设计

  本章介绍体系结构描述语言的设计，主要内容包括代码形式、预处理功能要求、词法规则语法规则等。

- 第四章整体结构设计

  本章介绍模拟器生成工具的整体结构设计，主要内容包括需求分析、目标语言选型和结构、功能模块设计等。

- 第五章 详细设计与实现

  本章介绍模拟器生成工具的详细设计与实现方法，按照功能模块为单位，详细

介绍每一个模块的设计与功能实现。

- 第六章 测试

  本章通过四个不同的测试用例，描述对模拟器生成工具的测试与结果，给出一些基本的分析。

- 总结

  本章将重新回顾本论文的开发过程，对论文进行总结，介绍本论文研究成果，提出不足之处，指出未来工作可能的研究方向。

2 相关技术分析

2.1 FPGA技术

FPGA[6][7]是英文Field-Programmable Gate Array的缩写，即现场可编程门阵列，它是在PAL、GAL、CPLD等可编程器件的基础上进一步发展的产物。它是作为专用集成电路ASIC（Application Specific Integrated Circuits）领域中的一种半定制电路而出现的，既解决了定制电路的不足，又克服了原有可编程器件门电路数有限的缺点。从1984年Xilinx公司发明了第一块FPGA开始，虽然产生不过二十多年，FPGA已经在电路设计领域占据了一席之地。FPGA技术因为成本较低、使用灵活，逐渐成为了主流的电路设计解决方案。从最初的在原型设计的辅助作用到现在的大量应用于各种解决方案，FPGA的发展前景非常乐观。

FPGA 采用了逻辑单元阵列 LCA（Logic Cell Array）这样一个概念，内部包括可配置逻辑模块 CLB（Configurable Logic Block）、输入输出模块 IOB（Input Output Block）和内部连线（Interconnect）三个部分。

FPGA 是由存放在片内 RAM 中的程序来设置其工作状态的，因此，工作时需要对片内的 RAM 进行编程。用户可以根据不同的配置模式，采用不同的编程方式。

加电时，FPGA 芯片将 EPROM 中数据读入片内编程 RAM 中，配置完成后，FPGA 进入工作状态。掉电后，FPGA 恢复成白片，内部逻辑关系消失，因此，FPGA 能够反复使用。FPGA 的编程无须专用的 FPGA 编程器，只须用通用的 EPROM、PROM 编程器即可。当需要修改 FPGA 功能时，只需换一片 EPROM 即可。这样，同一片 FPGA，不同的编程数据，可以产生不同的电路功能。因此，FPGA 的使用非常灵活。

2.2 基于HDL的设计方法

2.2.1 概述

FPGA 的基本设计方法是使用硬件描述语言 HDL（Hardware Description Language）进行设计，HDL 主要包括 Verilog HDL 与 VHDL 两种主流硬件描述语言，基于 HDL

的设计在 FPGA 设计中占 95%左右的比例，是现有的最成熟、最有效的设计方法。

2.2.2 Verilog HDL简介

Verilog HDL 是一种以文本形式来描述数字系统硬件的结构和行为的语言，用它可以表示逻辑电路图、逻辑表达式，还可以表示数字逻辑系统所完成的逻辑功能。

Verilog HDL 语言最初是于 1983 年由 Gateway Design Automation 公司为其模拟器产品开发的硬件建模语言，于 1990 年被推向公众领域，于 1995 年成为 IEEE 标准。

Verilog HDL 具有如下的特点：

1. 既能进行面向综合的电路设计，也可以用于电路的模拟仿真。
2. 能够在多个层次上对所设计的系统加以描述，从开关级、门级、寄存器传输级到行为级等。
3. Verilog HDL 语言具有混合建模能力，即在一个设计中，各个模块可以在不同的设计层次上建模和描述。
4. 灵活多样的电路描述风格，可以进行行为描述，也可以进行结构描述或者数据流描述。
5. Verilog 中的行为描述语句有条件语句、赋值语句和循环语句，类似于高级程序设计语言，便于学习和使用。

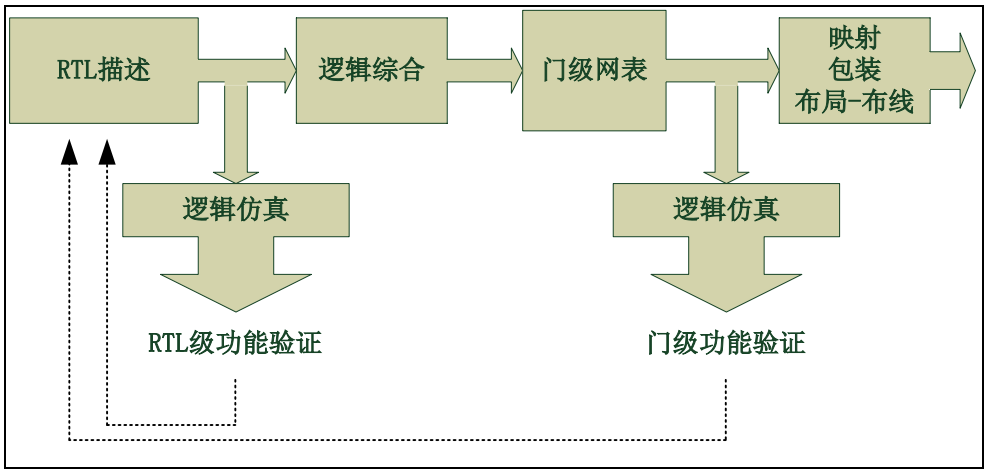


图 2.1 基于HDL的FPGA设计流程[6]

2.2.3 HDL设计方法的缺陷

以 Verilog HDL 为例，HDL 设计流程包括 RTL 代码编写、功能仿真、逻辑综合、布局布线、时序仿真和编程下载。每一个阶段都要进行验证，并且可能回溯到最开始的步骤。RTL 级描述的设计直接面对最终的实现，回溯过程包括了全部的设计步骤，所以 RTL 的编写受到较多因素的影响，开发周期较长。

HDL 设计是以 RTL 层面为基础的进行的，这是一个相对较低的层次，它有以下主要缺陷：

1. 编写 RTL 代码很耗时。
2. 验证 RTL 代码也很耗时。
3. 评估可替代方案很困难。
4. 规范变更很困难。

后两个缺陷部分源于前两个缺陷，而这些缺陷都有一个共同的原因——RTL 语言是面向实现的。这一特点可以用一个例子来解释：采用汇编语言实现系统，虽然能够给予程序员足够灵活的空间，但是在可移植性、编写和验证的难度上有着不可逾越的障碍，各种高级语言的成功也说明了汇编语言在大规模系统上描述能力的缺陷。

2.2.4 Verilog 程序设计语言接口

Verilog 程序设计语言接口 PLI（Programming Language Interface）是一种在 Verilog 中调用 C/C++函数的机制，通过使用 PLI，可以再 Verilog HDL 中实现行为级描述而不用编写 RTL 描述。

PLI 在描述能力方面存在两个问题，一是由于 PLI 的 C 代码在 Verilog 代码内部被调用，系统结构的建模依然在 Verilog 语言的层次上；二是由于使用 PLI 依然需要 Verilog 仿真平台的支持，程序设计语言不能完全发挥它本来的性能。

2.3 高级设计方法

2.3.1 概述

针对HDL设计存在的诸多问题，一些面向硬件电路设计的高级方法应运而生。这些方法致力于提高描述的抽象级别[18]，提高自动化程度，从而简化描述的复杂度，提高工作效率，改善工作质量。

现有的高级设计方法主要分为三类：

1. 基于高级程序设计语言的设计方法。这类方法主要基于 C/C++语言，以 System C、Catapult C 和 Impulse C 为代表。这类语言绝大多数都属于 C/C++语言的定制版本，通过对语言的一些约束和约定，使用 C/C++语言按照特定的规范编写程序，使其能够映射到硬件实现上，这类语言种类繁多，各具特色且都处于发展中阶段。在这些语言中，基于 C/C++语言一部分存在的最大问题就是由于语言本身的约束过于复杂，用户从对象出发，较难掌握描述与实现之间的对应关系，在一些实验中也出现过实现能力不足的问题。
2. 通用建模语言。以 AADL、UML 为代表。他们面向一般的系统建模，并不把描述的目标限定在硬件或软件上，可用于软件设计、硬件设计和软硬件协同设计。
3. 专用建模语言。这类语言抽象级别最高，描述能力也较有限，适合描述粗粒度且周期特性明显的系统，这类语言主要面向指令集体系结构 ISA（Instruction Set Architecture）的描述。

2.3.2 System C设计方法

System C[8]是基于C++语言的一种用于系统设计的语言，目的是为了提高电子设计的效率。本质上System C是一套C++的库和宏定义。

System C 利用 C++面向对象的特性，将硬件中的模块概念映射为类，将实体映射为对象。通过编写 C++程序，就能够进行硬件的模块化设计。

由于面向对象的设计思想本身就来源于真实世界的实体和分类，可以证明面向模块和面向对象这两种抽象的思想在实质上是相同的，所以 System C 的这一映射十分成功，

因此 System C 具有很强的描述能力和很好的描述形式。

而实际中 System C 的实现能力却受到很多因素的限制，如并行逻辑结构不能并行执行等，在此不一一列举。原因可能并不完全来自 System C 本身，因为面向对象思想和冯诺依曼计算机结构的串行执行之间并非同构，从面向对象模型到计算机程序之间的转化过程已经受到了一定的限制。

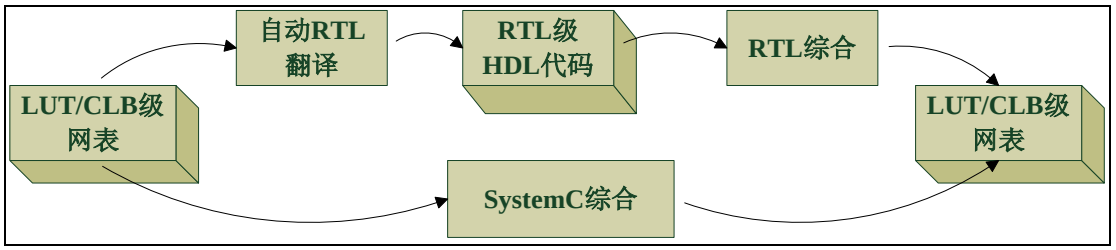


图 2.2 System C设计流程[6]

2.3.3 Catapult C综合技术

Catapult C[3]是Mentor Graphics公司研发的高级算法综合技术，能够从C++自动生成RTL硬件描述。

下图所示的是使用 Catapult C 技术的设计流程。

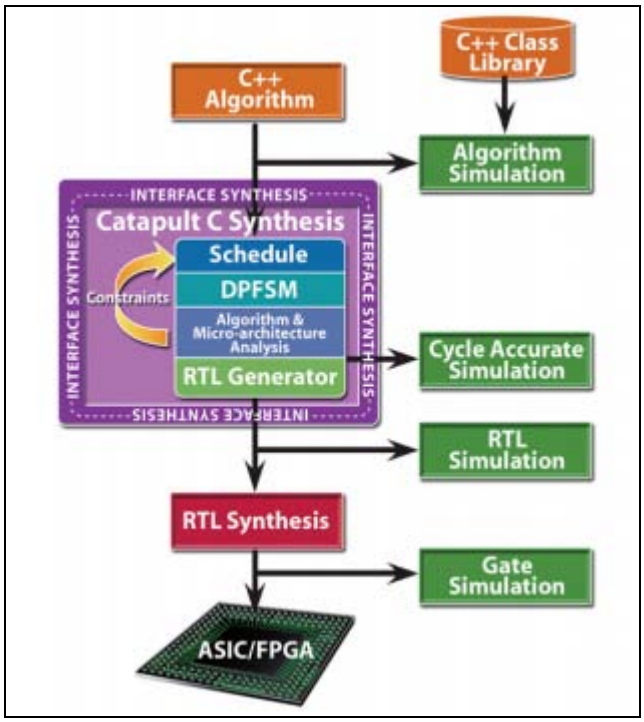


图 2.3 Catapult C设计流程[3]

Catapult C综合技术并不对C++进行扩展，而是支持了C++的一个子集，所以Catapult C 综合技术在代码层面的描述能力是受到限制的。它不支持动态内存分配，指针也必须指向静态变量即数组，程序的体积不能太大，并且程序结构要求简单，等等。这些限制使得 Catapult C 依然不能转化大多数的设计，其应用范围受到限制。

Catapult C综合技术不是系统建模技术，Catapult C综合技术主要解决的问题是实现而不是仿真，这两个应用范围是截然不同的[19]。

2.3.4 Impulse C设计方法

Impulse C[5]和System C类似，它也是一种以语言扩展库为基础的设计方法，Impulse C支持C语言描述的系统，具有和System C类似的建模、仿真和综合的能力。

由于 C 语言本身并不是面向对象的，所以 Impulse C 和 System C 相比，前者难以在概念的抽象级别上直接映射，模块、实体的描述形式较为冗长，编写和阅读都更加困难。由于 C 语言没有操作符重载、接口等一些和面向对象思想相关的结构，在 C 中实现信号的运算也是十分麻烦的，尤其是位操作。

2.3.5 统一建模语言

统一建模语言UML（Unified Modeling Language）[11]，最早于 1997 年由对象管理组织OMG（Object Management Group）发布。UML的目标之一就是为开发团队提供标准通用的设计语言来开发和构建计算机应用。UML提出了一套IT专业人员期待多年的统一的标准建模符号。通过使用UML，这些人员能够阅读和交流系统架构和设计规划，就像建筑工人多年来所使用的建筑设计图一样。UML侧重描述系统的软件体系结构，它在软件工程中占据非常重要的地位。

系统建模一般源于以下几个目的：

1. 按照实际情况或按照所需要的样式对系统进行可视化。
2. 规约系统的结构或行为。
3. 给出了指导构造系统的模板。
4. 对做出的决策进行文档化。

UML 能够帮助开发者进行建模，用统一的方法达到如上所述的四种目的。但是 UML 中抽象层次太高，它本身并不包括生成系统的转换规则，从 UML 模型到具体实现的转化依然需要用户手工完成，或自行编写模型转换的工具。

2.3.6 体系结构分析与描述语言

体系结构分析与描述语言AADL（Architecture Analysis and Description Language）[2][9]是一种面向软硬件协同设计的建模语言。

2004 年，美国汽车工程师协会 SAE(Society of Automotive Engineers)在 MetaH、UML 的基础上，提出嵌入式实时系统体系结构分析与设计语言，即 AADL，目的是提供一种标准、足够精确的方式，设计与分析嵌入式实时系统的软、硬件体系结构及功能与非功能属性，将系统设计、分析、验证、自动代码生成等关键环节融合于统一框架之下。AADL 语言包括软件、硬件和抽象系统组件。AADL 标准为描述和分析实用的系统结构的组件及其间的互相作用提供了形式化的建模理念。

AADL语言的抽象层次较高，难以直接进行具体设计，为了支持不同应用的建模需求，AADL可以通过语言扩展对具体系统进行描述。AADL提供两种扩展方式：引入新属性或符号以及子语言(Sub Language)扩展。后者的一般形式是AADL附件(Annex)，例如行为扩展（Behavior Annex）[10]、错误扩展（Error Annex）[17]。行为扩展存在于构件内部，对线程、子程序构件的执行进行更精确的描述。

目前的AADL工具具有一定的代码自动生成能力，如UCaG[15]，但是可转化的模型只限于线程、进程等AADL软件构件，对与硬件结构无法进行自动生成。

AADL 发展时间较短，大部分内容还在研究阶段，目前很难发现可用于从 AADL 语言描述生成模拟器的工具。

2.3.7 体系结构建模专用语言

这类语言的目的是为指令集体系结构进行设计和构件模拟器，主要有MIMOLA、nML[20]等。专用语言在结构化建模和行为级建模两方面各有不同的侧重，功能目标也各不相同。这些语言面向一般CPU结构的建模，设计因素包括指令、寄存器、地址等具体构件，而缺乏设计灵活性，不能自定义部件，难以应用于一般的硬件结构设计。

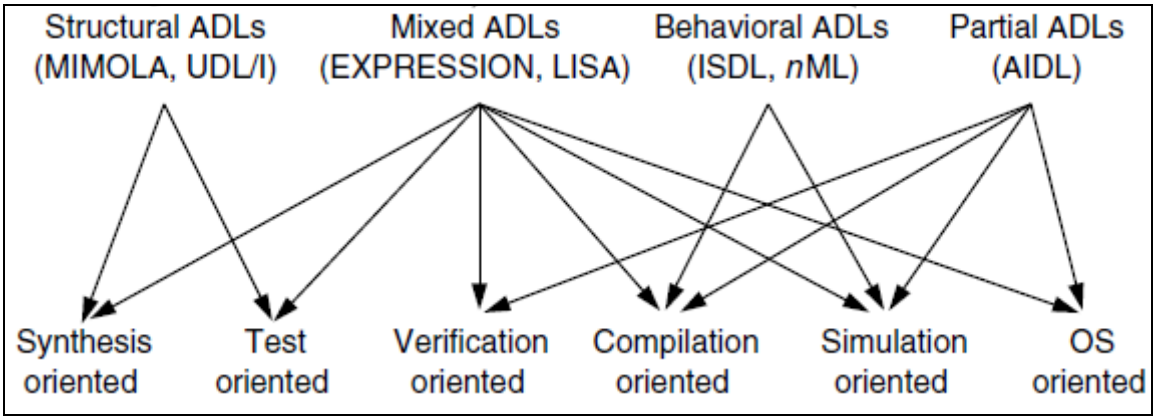


图 2.4 体系结构建模专用语言分类和特点[22]

2.3.8 对高级方法的分析

现有的语言、方法和工具较多地面向实现和建模两大领域，而很少有直接面向验证，特别是模拟器自动生成这一需求的工具。一般的工具都采用专用验证平台，即仿真器或解释器。这些工具大多数都存在仿真效率的问题，尤其是 RTL 仿真器的仿真速度较低。因此，开发一套专门面向验证，能够有效提升验证效率的设计方法是十分有必要的。

2.4 C程序设计语言

C，是一种通用的程序设计语言，它主要用来进行系统程序设计。具有高效、灵活、功能丰富、表达力强和移植性好等的特点，在程序员中备受青睐。1983 年，美国国家标准委员会（ANSI）对 C 语言进行了标准化，于 1983 年颁布了第一个 C 语言标准草案（83 ANSI C)，后来于 1987 年又颁布了另一个 C 语言标准草案（87 ANSI C）。最新的 C 语言标准是在 1999 年颁布并在 2000 年 3 月被 ANSI 采用的 C99。

C 语言作为一种十分流行的高级程序设计语言，它具有简单和高性能的特点。大量的 CPU 模拟器都采用 C 语言编写，在计算量庞大的硬件仿真领域，C 语言依靠其性能优势成为众多应用的首选语言。

本论文选用 C 语言作为生成模拟器的目标语言。GCC（GNU Compiler Collection）是被广泛使用的开放源代码的编译器集合，其中 C 语言的实现支持多种标准，并且 GCC 本身还提供了一些 C 语言的扩展特性。

2.5 Lex工具与Yacc工具

2.5.1 概述

Lex与Yacc[4][14]是UNIX系统中两个非常重要的工具，他们被用来自动生成编译器前端代码。在GNU/Linux中，这两个工具的实现版本分别是Flex和Bison。它们被大量的包含语言处理的软件所使用，其中包括著名的GCC（GNU Compiler Collection）。

使用 Lex 和 Yacc 构造编译器前端，是程序开发快速进入核心的处理过程，能够避免大量的重复工作。

2.5.2 Lex工具

Lex 是一种生成扫描器的工具。扫描器是一种识别文本中的词汇模式的程序，它完成编译器中词法分析。

词汇模式（或者正则表达式）在一种特殊的句子结构中定义。一项匹配的正则表达式可能会包含相关的动作。这一动作可能还包括返回一个标记。当 Lex 接收到文件或文本形式的输入时，它试图将文本与正则表达式进行匹配。

Lex 一次读入一个输入字符，直到找到一个匹配的模式。如果能够找到一个匹配的模式，Lex 就执行相关的动作，可能包括返回一个标记（token）。另一方面，如果没有可以匹配的常规表达式，将会停止进一步的处理，Lex 将显示一个错误消息。Lex 将输入文本转化为一个序列的标记（token）。

2.5.3 Lex语法规则

Lex 使用正则表达式对输入序列进行匹配，不同工具中的正则别打是都不相同，在这里有必要说明一下 Lex 使用的正则表达式。

Lex 脚本的主体以行为单位，每一行以一个正则表达式为开始，后面跟随 C 代码形式的相应操作，在 C 代码中使用“return”语句返回匹配的类型，无“return”语句则说明忽略当前匹配。

Lex 中用于元字符匹配的正则表达式如下所示：

表 2.1 Lex 正则表达式

符号	含义
.	匹配任意字符，除了‘\n’。
-	用来指定范围。
[ ]	字符集合。匹配括号内的任意字符。
*	匹配 0 个或者多个之前的模式。
+	匹配至少一个之前的模式。
?	匹配 0 或 1 个之前的模式。
$	匹配行末。
{ }	指定模式的出现次数的范围。
\	转义元字符。
^	否定。
|	表达式的逻辑“或”关系。
"<...>"	字符的字面含义。
/	前向匹配
( )	将一系列正则表达式分组。

如下图所示，这是一个简单的 Lex 程序实例，它以“%%”为开始和结束，每一行的左边是匹配的正则表达式，右边是要进行的操作，使用 C 语句编写。

```
%%
\n  ++num_lines; ++num_chars;
.   ++num_chars;
%%
```

图 2.5 Lex 语法示例

2.5.4 Yacc工具

Yacc 将任何一种编程语言的所有语法翻译成处理此种语言的语法解析器。它用巴科斯范式(BNF, Backus Naur Form)来书写。Yacc 用于生成编译器前端中的语法分析模块。

Yacc 的输入是 Lex 的输出，Yacc 将输入的标记序列匹配 BNF 范式，每一次匹配成功都可以调用相应的动作。

Yacc 还包括很多必要的错误恢复功能，使用 Yacc 能非常容易地构造出稳定的编译器前端。

2.5.5 Yacc语法规则

Yacc 采用 BNF 范式进行模式匹配，但是 Yacc 中的 BNF 范式与一般 BNF 范式有所区别，格式更加简单，更便于使用。

下图所示的是 Yacc 语法的示例。在实际的应用中，还需要在匹配的内容后面添加 C 代码段以对内容进行相应的操作。

```
value:
                VERIABLE
              | NUMBER
expression:
                value ‘+’ value
              | value ‘-’ value
```

图 2.6 Yacc 语法示例

2.6 Graphviz工具

2.6.1 概述

Graphviz 可以用来将图或网络的结构信息转化为图形化的表现形式，这是一种自动绘图的功能，使用这种自动图形化的方式展示结构化信息能够用很小的开销换取很好的展示效果。

使用 Graphviz 工具将模拟器生成工具获取的模块间信息转化为图形化的表示方法，便于开发人员快速验证设计的结构正确性，更容易发现设计错误。

Graphviz 工具的输出支持很多种常用的格式，包括大量的图像格式和文档格式，Graphviz 工具甚至还能作为函数库添加到用户程序中，作为一个整体使用。

下图所示的是一个从结构描述到图形转化的示例。

```
digraph G {

        subgraph cluster_0 {
                style=filled;
                color=lightgrey;
                node
[style=filled,color=white];
                a0 -> a1 -> a2 -> a3;
                label = "process #1";
        }

        subgraph cluster_1 {
                node [style=filled];
                b0 -> b1 -> b2 -> b3;
                label = "process #2";
                color=blue
        }
        start -> a0;
        start -> b0;
        a1 -> b3;
        b2 -> a3;
        a3 -> a0;
        a3 -> end;
        b3 -> end;

        start [shape=Mdiamond];
        end [shape=Msquare];
}
```

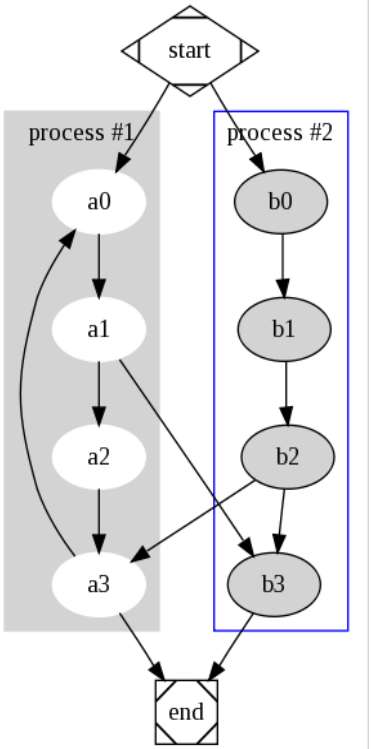


图 2.7 Graphviz 转化示例 DOT 语言

Graphviz工具的输入是DOT语言[13]所描述的结构化信息，使用Graphviz进行图形的自动生成，首先需要了解DOT语言的语法规则，将模块信息转化为DOT语言的形式。

DOT 语言的语法规则：

- *graph* : [**strict**] (**graph** | **digraph**) [*ID*] **'{'** *stmt_list* **'}'**
- *stmt_list* : [*stmt* [**';'**] [*stmt_list*]]
- *stmt* : *node_stmt* | *edge_stmt* | *attr_stmt* | *ID* '=' *ID* | *subgraph*
- *attr_stmt* : (**graph** | **node** | **edge**) *attr_list*
- *attr_list* : **'['** [*a_list*] **']'** [*attr_list*]
- *a_list* : *ID* ['=' *ID*] [**','**][*a_list*]
- *edge_stmt* : (*node_id* | *subgraph*) *edgeRHS* [*attr_list*]
- *edgeRHS* : *edgeop* (*node_id* | *subgraph*) [*edgeRHS*]
- *node_stmt* : *node_id* [*attr_list*]

- *node_id* : *ID* [*port*]
- *port* : **':'** *ID* [**':'** *compass_pt*] | **':'** *compass_pt*
- *subgraph* : [**subgraph** [*ID*]] **'{'** *stmt_list* **'}'**
- *compass_pt* : (**n** | **ne** | **e** | **se** | **s** | **sw** | **w** | **nw** | **c** | **_**)

其中粗体字表示关键字，各种元素都是与大小写无关的。

ID 可以是以下几种内容之一：

1. 一个由任意字母、数字、下划线组成的字符串，不以数字开头。
2. 一个数字。
3. 由双引号包含的任意字符串。
4. 一个 HTML 标记（<……>）。

edgeop 表示连接关系，有向图用“->”表示，无向图用“--”表示。

2.7 VI语法脚本

VI 是一种开放源代码的文本编辑器，它被众多程序设计者所欢迎。VI 和众多程序设计 IDE 或高级的文本编辑器类似，都具有语法高亮显示功能。通过添加自定义的设置，能够使 VI 支持新的语法格式。

本论文的目标是为开发者提供便于使用的模拟器生成平台，通过给用户提供有语法高亮[12]显示的编辑环境，能够更好地达到这一需求。

2.8 小结

本章介绍了本论文所涉及到的相关技术和工具，并对这些内容进行了分析。前面对硬件设计方法的介绍和分析，是设计一种新的描述语言和设计方法的重要参考依据。后面介绍了在开发中用到的主要工具。本章的内容是后文的描述语言设计和实现的基础。

3 体系结构描述语言的设计

3.1 概述

由于课题的目标是使用硬件（FPGA）对特定算法进行加速，所加速的算法属于科学运算，具有结构整齐、逻辑控制单一、计算量大等特点，现有的设计方法很大一部分是面向软硬件协同设计，软件的结构则较多地关心实时性、进程、逻辑控制等因素，和硬件关联并不大，而对于硬件的设计上往往更关心资源的分配。这些设计流程提供的方法对算法加速的研究意义较小，使用起来容易带来很多问题。面向科学计算领域的算法加速研究，更需要一种更直观面对硬件结构，更方便使用的专用模拟平台。

我们提出一种系统结构描述语言：ADL for Simulation。它比 Verilog HDL 更加简洁，使用高级程序设计语言描述部件的内部行为，可用于生成任意系统结构的模拟器，而不仅仅是一般的处理器结构。

ADL 描述以模块（Module）为单位。模块的内容包括外特性和内特性。模块的外特性包括模块的名称、周期属性和端口声明，其中端口类型包括输入、输出和双向端口 3 种。

模块的内特性有两种形式：结构描述和行为描述。结构描述中包括内部的模块和连接关系，可以看成是一些模块和连接的集合；行为描述则包括静态存储的声明和行为代码，行为代码要求符合相应程序设计语言的语法规则，在目前的实现中，采用 C 语言作为行为代码的语言。

描述语言的处理过程也被分成预处理、词法分析、语法分析、模型生成和模拟器生成等几个过程，前三个过程的功能需求在本章解释。

3.2 预处理

预处理是和语法规则关系紧密而又相对独立的内容，预处理过程的功能是为词法分析阶段准备合适的输入数据。由于本 ADL 语言也定义了预处理阶段的一些转化，为了便于理解本章中的内容，在这里首先介绍本语言的预处理功能。

在一般的编译器中，预处理阶段的功能并不是语言所要求的，以 GCC 为例，较常用的有引用文件（include）、宏替换（define）等。预处理阶段和编译本身是两个独立的过程，预处理阶段只进行文本处理，他的任务是组织代码，而不是理解代码，通过预处理阶段，原始文件被转换成编译器更易理解的形式。将预处理和编译阶段分离，既有利于简化编译器的设计，也方便用户检查代码中的错误。

本 ADL 语言的预处理阶段包括三种功能：

1. 外部文件引用。
2. ADL 描述语言和行为代码分离。
3. 行为代码中的引用转换。

3.2.1 外部文件引用

通过在代码中用添加以“#”开头的文件名作为单独的一行，可以引用相应的描述文件，如“#abc.adl”表示引用“abc.adl”文件。

外部文件引用机制和 C 语言中的“#include”类似，这种机制使得对系统的设计能够更具层次化，能够轻松的进行代码复用。

对于不存在的文件引用，和重复的文件引用，处理过程中应当给用户适当的反馈。

3.2.2 ADL描述语言与行为代码分离

由于行为代码部分采用标准的 C 语言，用户所要编写的 ADL 描述文件中将包含 ADL 描述和 C 语言代码两种内容，两种结构之间有引用关系。如果将 C 语言代码的语法分析工作纳入 ADL 分析工具，会极大的增加语法分析的复杂性，而对后期处理没有任何益处。因此这两种语言描述需要在预处理阶段进行分离操作。

两种语言混合编写的规则如下所述：行为代码块前后以“%%”标记进行包含，以区别描述和行为代码。预处理阶段的任务是将每一段 C 代码替换成单独的“%%”标记，并输出给编译器模块，那么在语法定义中，每一个“%%”标记都代表一个 C 代码块，被替换的 C 代码将在代码生成阶段被写入到目标文件中，如下图所示。

这样的设计也便于以后的扩展，ADL 语言的设计和目标语言耦合很小，使 ADL for

Simulation 容易被扩展来支持其他语言的模拟器生成。

```
module a(0){
  input[u32] a;
  input[u32] b;
  output[u32] o;
  ram[u32][3] rr;
%%
  op_add_U32($(o),$(a),$(b));
%%
}
```

处理前

```
module a(0){
  input[u32] a;
  input[u32] b;
  output[u32] o;
  ram[u32][3] rr;

  %%

}
```

处理后

图 3.1 预处理阶段功能示例

3.3 行为代码中的引用转换

在行为代码中必须要对模块的端口和存储器进行操作，必须要提供一种机制允许用户在 C 代码中使用模块中的端口和存储器。端口和存储器的概念会在后文中进行介绍。

预处理过程参考 Yacc 脚本的方法，采用标记的方法对模块中元素进行引用。

在 C 代码中，使用“$”符号加变量名对端口或存储器进行引用，通过预处理阶段的转化，C 代码中的这类引用被转化成目标端口或存储的地址。

在编写 C 代码的时候，只要将这种引用形式当成指针使用即可。例如引用名为“abc”的端口，在行为代码中的形式为 “$abc”。设定端口值的代码是“*($abc->value) = 1)”。引用数组形式的存储结构“xyz”的第二个元素，在行为代码中的形式为“$xyz[1]”，对其赋值的代码可以写成“op_assign_value(&($xyz[1]), 100)”，这里使用了一个模拟环境库的函数。

3.4 ADL语法定义

ADL for Simulation 的语法定义包括词法分析、语法分析阶段的形式定义，并不包括预处理阶段和代码生成阶段的转换规则。

3.4.1 词法规则

词法规则是由编译器前端的词法分析模块所识别的标记，词法标记在后续的语法分

析过程中作为序列输入。本 ADL 语言的词法标记较简单，主要包括标点符号、数字、标识符和关键字等。

词法分析模块采用 Lex 脚本格式编写，通过 Lex 工具自动转化为词法分析的代码。Lex 模块的输入不是用户所编写的 ADL 描述，而是经过预处理的中间形式。

下表所示的是词法分析部分的标记匹配列表。

表 3.1 ADL for Simulation 的标记匹配

匹配模式	标记名称	说明
‘(’	‘(’	左小括号
‘)’	‘)’	右小括号
‘;’	‘;’	分号
‘=’	‘=’	等于号
‘[’	‘[’	左方括号
‘]’	‘]’	右方括号
‘{’	‘{’	花括号
‘}’	‘}’	右花括号
[0-9]+	INTEGER	整数（无符号）
module	MODULE	module 关键字
input	INPUT	input 关键字
output	OUTPUT	output 关键字
inout	INOUT	inout 关键字
contain	CONTAIN	contain 关键字
ram	RAM	ram 关键字
link	LINK	link 关键字
\%\%	CODE	行为代码标记
[_a-zA-Z.][_0-9a-zA-Z.]*	TOKEN	标记，用于各种名字
[\t\r\n]+		空白字符

3.4.2 语法规则

语法规则是以 BNF 范式的形式给出的，它用来匹配词法分析模块输出的标记序列。如下所示是 ADL for Simulation 的语法，后文将对重要的部分详细解释：

- all: segments
- segments: /* 空 */
  | segments segment
- segment: ccode
  | module
- module: MODULE TOKEN cycle description
- cycle: /* 空 */
  |'(' INTEGER ')'
- description: '{' ports internal '}'
- ports: /* 空 */
  | ports port
- port: input_port
  | output_port
  | inout_port
- input_port: INPUT '[' TOKEN ']' TOKEN ';'
- output_port: OUTPUT '[' TOKEN ']' TOKEN ';'
- inout_port: INOUT '[' TOKEN ']' TOKEN ';'
- internal: structural
  | behavioral
- structural: entities
- entities: /* 空 */
  | entities entity
- entity: contain
  | link

- behavioral: rams ccode
- rams: /* 空 */
  | rams ram
- ram: RAM '[' TOKEN ']' '[' INTEGER ']' TOKEN ';'
  | RAM '[' TOKEN ']' TOKEN ';'
- contain: CONTAIN TOKEN TOKEN ';'
- link: LINK TOKEN '=' TOKEN ';'
- ccode: CODE

3.4.3 段（Segment）

和“段”相关的语法规则如下：

- all: segments
- segments: /* 空 */
  | segments segment
- segment: ccode
  | module

段（Segment）是语法分析过程中最基本的元素，一个 ADL 描述由任意数量的段组成。一个“段”可能是一个模块或一个代码标记。

在使用模拟器生成工具时，使用“-m”参数指定顶层模块的名称。

代码标记（ccode）的段结构使添加功能模块之内的 C 代码成为可能，例如行为代码中需要引用的一些库、一些子函数声明，都可以放在段中。段中的代码最终会被复制到目标代码的头部。“段”中的行为代码可以被看成是与模块功能无关的代码。

3.4.4 模块（Module）

和“模块”相关的语法规则如下：

- module: MODULE TOKEN cycle description
- cycle: /* 空 */

|'(' INTEGER ')'

- description: '{' ports internal '}'

模块是体系结构定义的基本单位，和 Verilog HDL 语言类似，整个体系结构的描述由模块组成。

模块以关键字“module”开始。后面紧接着是模块的名字，名字是一个 TOKEN 标记，类似于 C 语言中的变量名。然后是周期属性标记 cycle，它可以被省略，默认值为 1，并且 cycle 标记只在行为描述模块中有效。最后是描述部分，它包括了模块的端口声明和内特性描述。

周期属性是指一个模块的行为和时钟周期的关系，一个功能部件可能体现零周期、非零周期属性。

零周期模块等价于组合逻辑电路，它没有时钟信号输入，它的输出只和当时的输入相关，每当输入改变时，输出马上会随之改变。

非零周期模块等价于时序逻辑电路，它被一个时钟信号驱动，每当时钟发生一次翻转，他的输出才和此时的输入建立联系，发生改变。

非零周期模块还可以分为单周期模块和多周期模块。单周期模块是指一次时钟沿到来之前，它的输出只和上一次时钟来时的输入有关。多周期模块和单周期模块没有本质的区别，多周期模块代表一次输出和多次输入相关。多周期模块是一种方便使用者的机制，通过指定模块的周期属性，模拟器生成工具将自动地缓存多个周期的输入，这一机制将在端口定义部分详细解释。

描述部分由一对花括号包含，内部包含了端口信息和内部属性。

除了周期属性，描述部分包括所有的模块信息，包括端口、存储、行为等等。

3.4.5 端口（Port）

和端口相关的语法规则如下：

- ports: /* 空 */
  | ports port
- port: input_port

| output_port
| inout_port

- input_port: INPUT '[' TOKEN ']' TOKEN ';'
- output_port: OUTPUT '[' TOKEN ']' TOKEN ';'
- inout_port: INOUT '[' TOKEN ']' TOKEN ';'

端口包括三种类型：输入（input）、输出（output）和双向（inout）。

端口的声明以端口类型开始，可以是关键字“input”、“output”或“inout”。然后是用方括号包含的信号类型，目前支持的信号类型共有 9 种，它们将在后文介绍。最后是端口的名称，它也是 TOKEN 标记。分号“;”在最后，作为一个端口声明的结束标识。

端口是部件与部件之间通信的渠道，不同类型的端口完成不同的通信任务。

“inout”端口在本语言中作为一种总线结构。多个 inout 端口之间能够构成一种对等的网络关系，同一时刻可以有一个发送者，多个接收者。

本语言提出了一种输入历史缓冲区的结构，对于周期属性大于 1 的模块，其 input 端口和 inout 端口都具有输入历史缓冲区。假设模块的周期属性为 n，缓冲区里按时间倒序排列了从当前周期到过去一共 n 周期的输入值，用户在行为代码中只要简单地使用数组的形式访问缓冲区即可获得 n 周期内的任意输入。这一机制有助于简化多周期输入、队列缓冲等功能的实现，使用这种机制，用户不需要再编写复杂的输入缓冲，而直接在周期属性中设置值来让系统自动生成缓冲区及维护缓冲区的代码。

一个目前已知的缺陷是，用户定义输入输出端口的时候不能以数组的形式声明，这在构造模块接口的时候可能会带来很多冗余的代码。数组形式的端口在实现形式上可能和缓冲区结构类似，但是并不与其冲突，在今后的改进中考虑采用二维数组的形式实现端口的数组声明形式。

3.4.6 内部属性（Internal）

和内部属性相关的语法规则如下：

- internal: structural
  | behavioral

模块的内部属性有结构描述（structural）和行为描述（behavioral）两种类型。结构描述类型的模块是若干个子模块的集合，结构描述模块的内部属性包括子模块声明和连接声明。行为描述模块不能包含子模块，它可以包含存储结构和功能代码。

3.4.7 结构描述（Structural）

和结构描述相关的语法规则如下：

- structural: entities
- entities: /* 空 */
  | entities entity
- entity: contain
  | link

抽象地来看，一个结构描述模块内部包含若干实体，实体分为“内含子模块”和“连接”两类。结构描述的模块逻辑简单清晰，便于按层次结构组织系统中的部件。

3.4.8 内含子模块（Contain）

和内含子模块相关的语法规则如下：

- contain: CONTAIN TOKEN TOKEN ';'

声明一个内含子模块需要使用“contain”关键字，后面依次指定子模块的模块类型和实体名称，最后以分号结束。

一个结构描述模块可以包含其他在设计中声明过的模块，被包含的模块在设计中的任意位置声明均可，并不要求引用与被引用的先后位置。子模块包含声明是不允许重入的，任何模块都不能直接或间接地包含本身。

3.4.9 连接（Link）

和连接相关的语法规则如下：

- link: LINK TOKEN '=' TOKEN ';'

连接属于结构描述中的实体，它表示两个端口之间的通信关系。

声明连接需要使用“link”关键字，后面跟随的第一个标识符是连接的目的端口，然后是等于号“=”以及代表源端口的标识符，最后以分号结尾。

连接关系表示从源端口到目的端口有数据连通关系，连接中能够出现的端口可能是模块本身的端口，也可能是包含的子模块的端口，端口之间的数据传递方向必须符合要求。

使用模块本身的端口，直接使用端口的名字即可。引用子模块的端口，要使用“子模块名”+“.”+“端口名”的形式进行引用。

下表给出了连接定义中，源、目的端口的有效搭配关系。

表 3.2 有效的源端口和目的端口搭配

目的端口	源端口
本体的 output 端口	本体的 input 端口
子模块的 input 端口	本体的 input 端口
本体的 output 端口	子模块的 output 端口
子模块的 input 端口	子模块的 output 端口
任意模块的 inout 端口	任意模块的 inout 端口

对于表格中的前 4 项，数据在连接上的移动方向是单向的，只能从源端口流向目的端口。数据的移动过程是自动生成的，不需要用户维护。

inout 端口的连接则是双向的，并且可以用多组连接声明将多个 inout 端口连接在同一网络上。每次 inout 连接传送数据时，同一网络上只能有一个 inout 端口发送数据，其他端口则要被置为“高阻”状态。

由于在 FPGA 设计中并不推荐使用 inout 端口。简化起见，在目前的设计中暂不支持 inout 端口和另两种类型端口的互联。

3.4.10 行为描述（Behavioral）

和行为描述相关的语法规则如下：

- behavioral: rams ccode

行为描述的核心是行为代码，本工具目前支持用 C 语言编写的行为代码，为了在模块中声明持久化的存储器，存储器需要和功能代码相互独立，作为一个单独的语法成分在行为代码之前声明。

行为描述的结构首先包含存储声明，然后是行为代码。

3.4.11 存储（RAM）

和存储相关的语法规则如下：

- ram: RAM '[' TOKEN ']' '[' INTEGER ']' TOKEN ';'
  | RAM '[' TOKEN ']' TOKEN ';'

存储的声明格式和输入输出端口类似，区别是存储还能够以数组的形式声明。

存储的声明首先以“ram”关键字开始，然后是被方括号包含的数据类型，数组宽度是可选的，它是被方括号包含的一个整数，最后是存储的名称和‘;’。如果没有声明数据宽度，则默认的宽度就是一单位的数据。

3.4.12 行为代码（C code）

行为代码是一段 C 代码，它负责模拟模块的行为，通过预处理阶段的转化，这段代码在词法分析阶段的形式是一个简单的标记“%%”，而具体的代码已经被转移到单独的临时文件中。

用户编写行为代码块，需要在代码块的前后各用单独一行的“%%”标记包含。在 C 代码中引用端口和存储的方式是在端口名或存储器名前添加“$”符号的标记，这部分内容在预处理部分已经进行了介绍。

3.4.13 基本数据类型

基本数据类型是端口和存储器的一项基本属性。

在 Verilog HDL 中，端口只有宽度而没有数据类型，在 C 语言中只有和 CPU 寄存器大小相关的几种数据类型，对任意宽度的数据类型进行操作较为复杂。综合这两种语言的特点，同时为了方便行为代码的编写，本语言目前提出了一种混合式的数据类型定义。

目前的数据类型定义包括类似于 C 语言的基本数据类型，和用于一般信号传递的一位宽数据类型。

由于目前的需求中对任意位宽的数据类型没有要求，并且使用组合多个一位数据的形式也能够实现任意宽度的数据操作，所以目前的设计中并不直接支持任意宽度的数据类型。

本 ADL 语言支持的数据类型如下表所示。

表 3.3 ADL 语言支持的基本数据类型

数据类型	说明
I32	对应于 C 语言中的 int 类型
I16	对应于 C 语言中的 short 类型
I08	对应于 C 语言中的 char 类型
U32	对应于 C 语言中的 unsigned int 类型
I16	对应于 C 语言中的 unsigned short 类型
I08	对应于 C 语言中的 unsigned char 类型
F32	对应于 C 语言中的 float 类型
F64	对应于 C 语言中的 double 类型
W01	一位二值数据信号，仅具有 1 和 0 两种取值

模拟器中的基本数据类型实际是一个结构体，它包含两个域：option 和 value。option 域信号状态参数，可能的值如下表所示：

表 3.4 基本数据类型的 option 域取值

option 域取值	含义
OPTION_VALID	信号有效
OPTION_X	信号未知
OPTION_Z	高阻态

3.5 小结

本章详细描述了一种面向部件级模拟的体系结构描述语言的语法定义，说明了这种语言对预处理功能的要求，给出了语言的词法结构和语法结构，并解释了各个语法元素的含义。本章的内容是后文进行工具设计与实现的重要参考依据。

4 模拟器生成工具的整体结构设计

4.1 设计要求

4.1.1 基本功能要求

模拟器生成工具的设计目标是实现将 ADL for Simulation 语言的描述自动转化为模拟器的功能。通过将此工具加入到 FPGA 设计流程中，能够简化设计过程，将设计与实现过程进行分离，提高设计阶段的验证周期，从而提高设计的速度。

下图所示的是将模拟器生成工具加入到 FPGA 设计流程中的示意图。

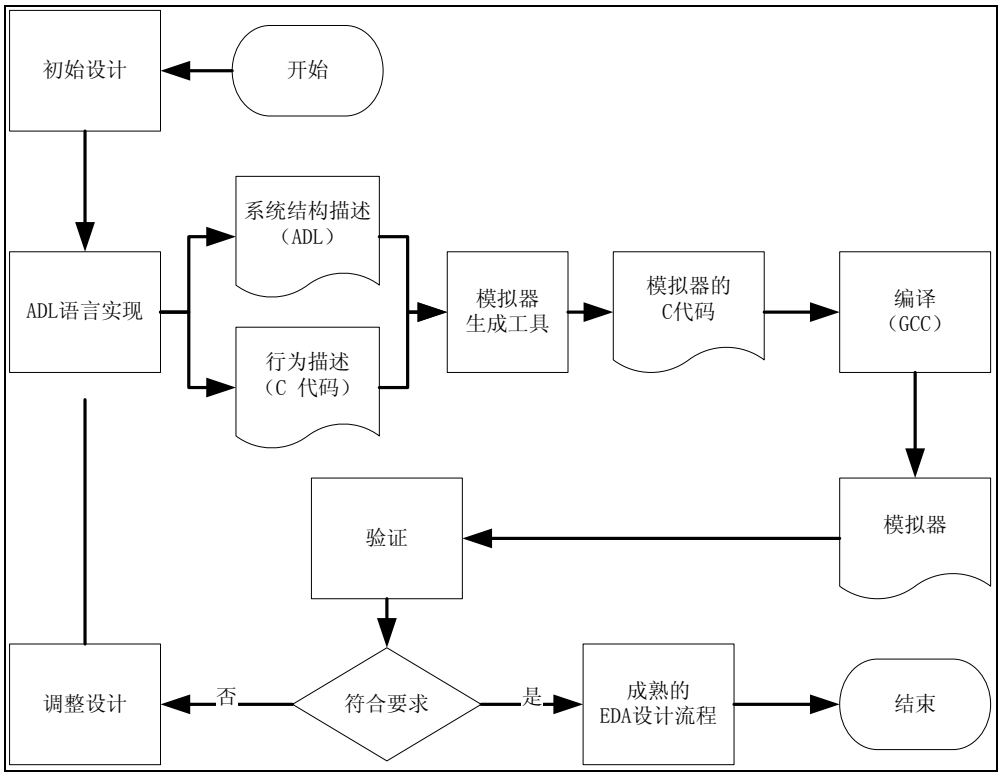


图 4.1 使用模拟器生成工具的 FPGA 设计流程

模拟器生成工具所完成的任务是从用户描述到模拟器代码转化的过程，这一过程要求结果正确，生成的模拟器具有合理的性能，并且能够提供给用户必要的反馈。目标代码也要具有充分的可扩展性，便于和其他工具组合使用。

4.1.2 附加功能要求

为了给体系结构设计者提供便利，转换工具除了完成主体功能之外，还需要附加信息反馈功能和对其他模型的转换支持。在代码生成阶段也需要加入一些基本的调试功能。模拟器需要实现的附加功能包括系统结构图生成和 Verilog HDL 代码框架生成。

系统结构图生成功能中引用了 Graphviz 工具，使用 Graphviz 工具将模拟器生成工具输出的 dot 语言描述转化成图形化表示。

Verilog HDL 框架生成功能能够自动将结构模块以及行为描述模块的外部接口转化为 Verilog HDL 描述，使得用户完成模拟器设计后能够更快地进行 HDL 设计，直接关注模块的内部功能实现。

4.1.3 输入与输出

综合前面所述的功能要求，模拟器生成工具需要具备输出一下几种内容的功能：

1. 模拟器源代码。
2. 系统结构图。
3. HDL 框架（Verilog HDL）。

4.2 目标语言选型

计算机程序可以分为两大类，即编译执行和解释执行。这两种程序类型在性能上存在较大差别，尤其是在计算密集的程序中。多数比较权威的 CPU 模拟器都采用 C 语言编写，C 语言描述的程序具有很高的性能，易于进行性能分析，非常适合在模拟仿真领域使用。

由于 C 语言在具体的编译器环境下还可能存在兼容性问题，所以自动生成的 C 语言以在 GCC 下可编译为标准。

4.3 系统结构设计

模拟器生成工具类似于编译器，工作流程中包括参数解析、预处理、词法分析、语

法分析、语义分析等步骤，后端功能涉及模型转换、调度、代码生成等内容，整个处理流程完成一套从用户描述到目标代码的转化过程。

在程序结构上，在整体上将处理过程分为三个主要步骤，每个步骤内部又分为具体的子过程。系统的处理流程如下图所示。

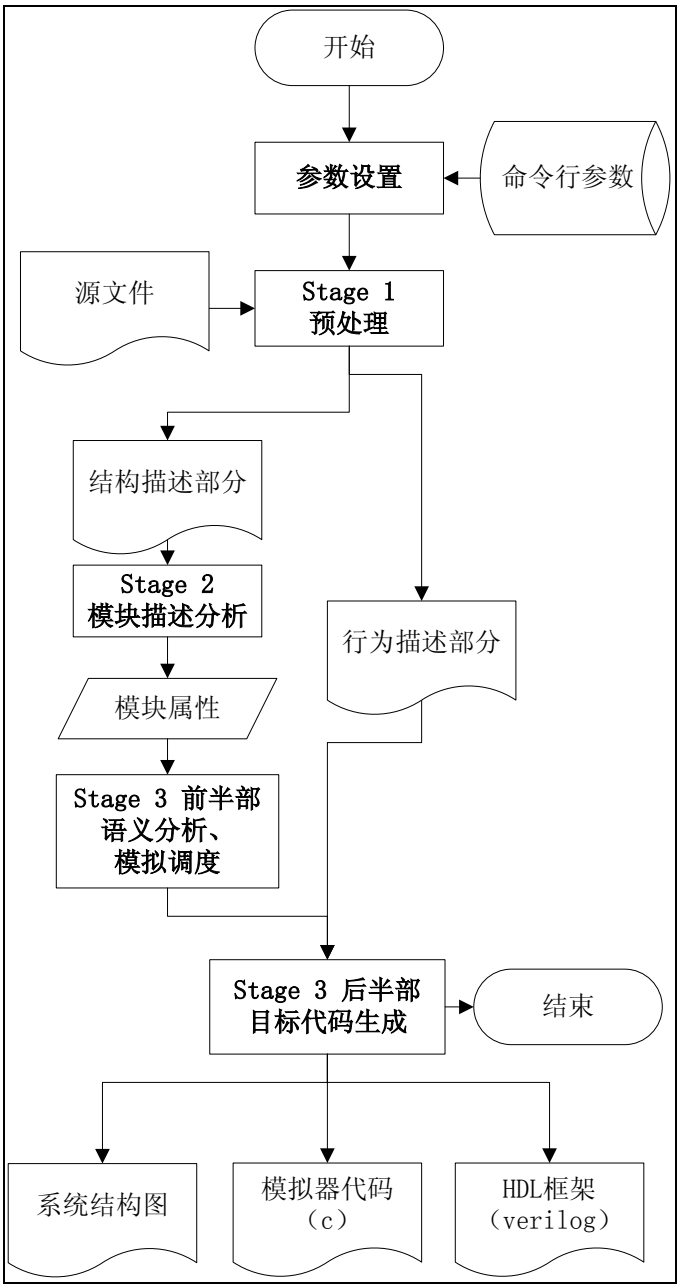


图 4.2 模拟器生成工具的处理流程

4.4 功能模块设计

根据系统结构的设计要求，系统按照功能划分为如下几个模块：

- 通用功能（common）：
  通用功能模块主要包括在整个工具中用到的各种模型结构定义，以及对模块进行操作的各种基本函数和错误消息函数，提供了对模块进行操作的基本机制。
- 一般功能（util）：
  一般功能模块包括了和模块无关的一些功能函数，主要包括一些字符串处理函数。
- 参数配置（config）：
  在模拟器生成工作开始前，首先要使用配置模块进行必要的参数设置，参数配置模块按照用户输入的参数，设置各个处理阶段的功能选项。
- 预处理（pre）：
  预处理模块对用户的输入文件进行分离，输出结构描述和行为描述两个部分供后面的语言扫描模块使用。
- 处理过程控制器（stage）：
  本模块对模拟器生成的整体过程按阶段（stage）进行控制，模拟器生成的处理流程主要分为 3 个阶段：预处理阶段、模块扫描阶段和模拟器生成阶段。
- 词法分析（lex）：
  词法分析模块将预处理阶段的结果——结构描述部分作为输入，进行词法分析，然后输出到后级的语法分析模块。本模块采用 lex 脚本编写，使用 lex 工具生成 C 代码。
- 语法分析（parse）：
  语法分析模块的输入是词法标记序列，按照 Yacc 的 BNF 范式进行语法匹配，然后将匹配的内容生成模块结构体，供后端的模块使用。语法分析模块分为两个部分，Yacc 脚本和功能函数。
- 模块分析（module）：

模块分析模块进行模块信息的基本语义分析，它的输入是语法分析模块生成的 module 结构。本模块对模块的内容进行有效性检查，将独立的模块信息组合成实体树和依赖树的结构，并且在此过程中查找各种可能的错误。

- 全局分析（global）：
  全局分析模块使用模型处理模块生成的实体树结构，以全局的角度对模块进行分析，将结构内的连接扩展成全局连接，然后进行模拟调度分析。
- 模拟器代码生成（code）：
  模拟器代码生成模块使用模块实体信息、连接关系和模拟调度信息，生成模拟器代码，输出到 C 文件中。
- 结构图描述生成（graph）：
  结构图描述生成模块将模块的实体树结构转化成 DOT 语言形式的脚本文件，用户可以使用 Graphviz 工具将这个脚本转化成各种图像形式。
- Verilog HDL 代码框架生成（verilog）：
  本模块将所有的结构描述信息转化成 verilog 语法描述的形式，对于行为代码的部分，用户需要自行填写相应的 Verilog 代码。对于行为代码的自动转化已经超出了本论文所探讨的内容。
- 模拟环境库：
  模拟环境库是和模拟器生成工具相对独立的一个部分，它也是一些 C 代码，用于在模拟器中提供常用的功能调用，通过使用模拟环境库，能够简化模拟器生成的工作负担，将一些重复的和常用的功能统一管理，减少目标代码的复杂度。

4.5 小结

本章分析了模拟器生成工具的需求，明确了工具的功能要求，给出了软件的体系结构定义，并给出了软件的功能模块划分。

5 模拟器生成工具的详细设计与实现

5.1 通用功能

在本模块里定义了和模拟器系统模型相关的数据结构与处理函数，它为处理流程提供便利的操作机制。

5.1.1 列表项属性枚举类型

如上图所示，这是一个枚举类型的定义，它定义了一些不同的名字，用来区别相同结构形式的不同含义。后文将介绍具体用途。

```
typedef enum __list_type_enum {
        type_input,
        type_output,
        type_inout,
        type_ram,
        type_contain,
        type_link,
        type_global,
        type_group,
        type_sorted,
        type_bus,
        type_schedule,
        type_unknown
} LIST_TYPE_E;
```

图 5.1 列表项属性枚举类型

5.1.2 列表项结构体

下图中代码定义的是一种链表形式的列表结构。

```
typedef struct __sim_list {
        struct __sim_list *next;
        LIST_TYPE_E type;
        char *name1;
        char *name2;
        int value1;
} SIM_LIST;
```

图 5.2 列表项结构体定义

结构中有 LIST_TYPE_E 域，用来标记列表项的类型，name1、name2 和 value1 用

来存放相关数据，对于不同的列表项，相关数据的含义也各不相同。下表显示了具体的列表项属性和内容的关系。

表 5.1 列表项类型

类型名称	含义	name1	name2	value1
input	输入端口	端口名称	数据类型	（空）
output	输入端口	端口名称	数据类型	（空）
inout	输入端口	端口名称	数据类型	（空）
ram	存储	变量名称	数据类型	数据宽度
contain	子模块包含	实体名称	模块类型	（空）
link	一般连接	目的端口	源端口	是否为总线
global	全局连接	目的端口	源端口	是否为总线
group	分组模块	实体名称	（空）	分组编号
sorted	已排序的分组	实体名称	（空）	分组编号
bus	总线连接	端口 1	端口 2	总线编号
schedule	模块调度关系	后级模块	前级模块	（空）

5.1.3 模块结构体

上图所示的是模块在程序中的结构定义，用来描述用户所描述的一个模块的所有属性，它也是链表结构的，一个设计中的所有模块信息都写入到一个模块结构链表中。

```
typedef struct __sim_module {
        struct __sim_module *next;
        char *name;
        int cycle;
        int behavior;
        SIM_LIST *input;
        SIM_LIST *output;
        SIM_LIST *inout;
        SIM_LIST *ram;
        SIM_LIST *contain;
        SIM_LIST *link;
        int flag;
} SIM_MODULE;
```

图 5.3 模块结构体定义

cycle 域是模块的周期属性。behavior 域是模块描述形式的标记，取值为“0”代表模块是结构描述的，取值为“1”代表模块是行为描述的。flag 域与模块属性无关，用来在分析时设置一些标记值。能够通过名字了解模块结构体中其它域的含义，不赘述。

5.1.4 树节点结构体

树节点结构体用来构造模拟器模块的实体树和依赖关系树，定义如下图所示。

```
typedef struct __sim_tree {
        struct __sim_tree *father;
        struct __sim_tree *brother;
        struct __sim_tree *child;
        char *name;
        int group;
        SIM_MODULE *module;
} SIM_TREE;
```

图 5.4 树节点结构体定义

每一个树节点都记录一个父节点（father 域）、一个子节点（child 域）和一个兄弟节点（brother 域）。

name 域表示这一节点的名字，在实体树中，名字代表这一模块在其父节点中被引用的名字；在依赖关系树种，名字就是模块类型本身。

group 域用来在模拟调度阶段存放分组信息，和结构本身无关。

module 域指向这一节点对应的模块结构体。

5.1.5 功能函数

本模块提供了对上述几种结构进行操作的功能函数，以及用于提示信息显示的两个函数。下面对这些功能函数进行简要介绍：

- void common_error(const char *format, ...);
  报告一个错误，并对错误进行计数。
- void common_warning(const char *format, ...);
  报告一个警告，提示可能的隐患，但并不影响处理过程。
- void common_error_list_element(SIM_LIST * element);
  用于报告一个模块中的错误，在错误消息中显示模块属性。

- int common_type_check(char *type);
  检查字符串形式的数据类型名是否合法。
- SIM_LIST * common_module_find_name1(SIM_MODULE * module, char *name);
  在一个模块中寻找 name1 域匹配 name 字符串的列表元素。
- SIM_MODULE * common_module_list_find(SIM_MODULE * list, char *name);
  在模块列表中寻找名为 name 的模块。
- void common_module_print(SIM_MODULE * module);
  输出一个模块的相关信息。
- void common_module_list_print(SIM_MODULE * head);
  输出列表中所有模块的相关信息。
- void common_module_list_free(SIM_MODULE * root);
  释放一个模块列表。
- SIM_LIST * common_list_find_name1(SIM_LIST * head, char * name1);
  在列表中寻找一个匹配 name1 域的一个项。
- SIM_LIST * common_list_find_name2(SIM_LIST * head, char * name2);
  在列表中寻找一个匹配 name2 域的一个项。
- SIM_LIST * common_list_find_both(SIM_LIST * head, char * name1, char *name2);
  在列表中寻找同时匹配 name1 和 name2 的一个项。
- SIM_LIST * common_list_find_any(SIM_LIST * head, char *name);
  在列表中寻找匹配 name1 或 name2 的一个项。
- void common_list_print(SIM_LIST * list);
  输出整个列表的相关信息。
- void common_list_print_node(SIM_LIST * node);
  输出一个列表项的相关信息。
- void common_list_add(SIM_LIST ** head, LIST_TYPE_E type, char *name1, char *name2, int value1);
  添加一个列表项到列表中，追加在末尾。

- void common_list_delete(SIM_LIST ** phead, SIM_LIST * node);
  在列表中删除一项。
- void common_list_free(SIM_LIST * root);
  释放整个列表。
- void common_tree_print(SIM_TREE * root);
  输出整个树结构的信息。
- void common_tree_print_node(SIM_TREE * node);
  输出一个树节点的信息。
- SIM_TREE * common_tree_add_child(SIM_TREE * father, SIM_MODULE * child, char *name);
  为一个树节点添加一个新的子节点，并返回这个新的节点。
- SIM_TREE * common_tree_find_node(SIM_TREE * root, char *name);
  在树结构中按名字查找节点。
- SIM_MODULE * common_tree_find_module(SIM_TREE * root, char *name);
  在树结构中按模块类型名字查找节点。
- void common_tree_free(SIM_TREE * root);
  释放整个树结构。
- LIST_TYPE_E common_parse_type(const char * const type_name);
  识别数据类型，转化成内部的枚举类型。
- char * common_type_to_string(LIST_TYPE_E type);
  将数据类型的枚举类型形式转化为字符串形式。
- int common_data_type_to_width(char * data_type);
  获取一个数据类型的宽度。

5.2 一般功能

本模块主要包括了几个用于字符串处理的函数，实现在分析过程中常用的一些字符串操作。包含的功能函数如下：

- char * util_join(char *string1, int free1, char *string2, int free2);
  合并两个字符串，free1 和 free2 参数用来选择是否释放源字符串占用的内存。
- char * util_trim(char *buffer);
  删除字符串前后端的空白字符。
- char * util_trim_front(char *buffer);
  删除字符串前端的空白字符。
- char * util_split_first(const char * original, char c);
  以第一次出现字符 c 的位置分割字符串，返回前半部。
- char * util_split_last(const char *original, char c);
  以最后一次出现字符 c 的位置分割字符串，返回后半部。

5.3 参数配置

参数配置模块为所有的可配置参数提供统一的管理，包括处理过程中的文件名称、一些处理选项、生成代码选项等。

5.3.1 参数信息结构体

参数信息结构体包含了所有的可配置信息，结构定义如下图所示。

```
typedef struct __CONFIG {
        int run_cycle;
        int graph;
        int verilog;
        int preprocess_only;
        int verbose;
        int keep_arch;
        int keep_code1;
        int keep_code2;
        int initial_value;
        int trace;
        char * top_module;
        char * file_name_input;
        char * file_name_sim_h;
        char * file_name_sim_c;
        char * file_name_arch;
        char * file_name_code1;
        char * file_name_code2;
        char * file_name_script;
        char * file_name_verilog;
} CONFIG;
```

图 5.5 参数信息结构体定义

下面对所有的可配置信息进行解释：

- run_cycle：

  整型值，表示生成的模拟器运行需要运行多少个周期。

- graph：

  是否生成 dot 格式的结构描述。

- verilog：

  是否生成 verilog 语言代码框架。

- preprocess_only：

  是否仅进行预处理。

- verbose：

  是否输出大量处理过程信息。

- keep_arch：

  是否保留临时文件 arch。

- keep_code1：

  是否保留临时文件 code1。

- keep_code2：

  是否保留临时文件 code2。

- initial_value：

  全局初始化值，这是一个字符值，有‘0’、‘x’和‘z’三种取值。

- trace：

  是否输出 trace 信息。

- top_module：

  顶层模块名称。

- file_name_*：

  各种临时文件的名称。

5.3.2 参数配置函数

使用参数配置功能前，首先要调用 config_initial 函数进行初始化，设置默认值。

参数配置函数与参数信息结构体的各个域一一对应，每一个参数都有一对获取（get）和设置（set）的函数，具体名称不赘述。

5.3.3 命令行参数格式

命令行参数的内容以空格分隔，每个参数以减号‘-’作为前缀，后面跟随一个字符的选项名，如果这个参数有附加参数，则跟在后面继续输入，附加参数和参数之间可以不含空格。

下表所示的是模拟器生成工具识别的参数，参数中的冒号表示有附加参数。

表 5.2 命令行参数

参数	含义
-h	显示帮助信息
-v	verbose 模式，显示大量的处理过程信息
-g:	生成 dot 和 verilog 代码，默认不生成。“g”代表图形，“v”代表 verilog
-p	只进行预处理
-m:	指定顶层模块名，默认为“top”
-k:	保留临时文件，附加信息是 1、2 或 a，各代表两阶段的行为代码和描述代码
-r:	设置生成的模拟器运行几个周期后结束
-i:	设置初始化的值，可选则“0”、“x”或“z”，默认为“0”
-t	生成的模拟器是够输出 trace 信息。trace 信息可用于非交互式调试

5.4 预处理

预处理模块的主要功能是对输入文件进行简单的文本操作，方便后端程序的读取。预处理阶段的功能如下所示：

- 外部文件引用。

- ADL 描述语言和行为代码分离。
- 记录预处理后的代码行号变动。

5.4.1 外部文件引用

使用“#”符号作为一行的开始，后面跟随一个文件名称，就会被预处理模块识别为外部文件应用，这一项处理不会和 C 代码中的预处理操作符进行匹配。

每识别到一个引用文件，首先将被引用的文件读取到临时文件中，然后再继续处理当前的文件。引用的处理过程是递归的，在被引用的文件中也能引用更多的文件。

5.4.2 ADL描述语言和行为代码分离

ADL 描述文件中以“%%”符号分隔描述语言部分和行为代码，这个标记不会和 C 语言中的语法冲突。在分离的代码中，描述部分的 C 引用被替换成一个“%%”符号，而行为代码部分则分段按顺序存放，由于此时还没进入语法分析阶段，不能将模块信息和行为代码建立联系，在语法分析过程后，再对行为描述代码重新按顺序与模块建立联系。

5.4.3 记录预处理后的代码行号变动

由于经过预处理，词法分析模块所获取的输入流已经和源文件有所区别，在预处理阶段需要进行原始文件和目标文件的行号变动对照关系，以便在报告错误、警告的时候显示行号，让用户快速找到问题的所在。

行号对照的记录采用每行一个记录的形式，将预处理过后的文件的每一行对应到一个原始文件中的一行，在输出行号相关信息的时候只要在对照表中直接查询即可。

5.5 处理过程控制器

本模块的功能是对整个处理流程进行控制。在本模块的最开始，首先调用参数配置模块（config）进行相应设置。然后按照语言处理的三个阶段，将各阶段的功能分阶段执行。每个阶段的功能函数都包括初始化、主体过程和清理子过程函数。功能函数的声明如下所示，其中<n>符号代表 1、2 或 3：

- static int stage_<n>_initial(void);
- static int stage_<n>_process(void);
- static void stage_<n>_finish(void);

initial 函数负责打开本阶段使用的输入、输出文件，process 阶段负责主要的处理，finish 阶段对临时文件进行关闭或者删除。

根据配置设置，可以选择只进行预处理（阶段一）的工作。

5.5.1 阶段一

本阶段的功能是对输入文件进行预处理。打开用户描述文件用于读取，打开两个文件用于输出分离后的描述部分和代码部分，然后调用预处理模块进行相关操作，最后关闭所有文件。

5.5.2 阶段二

本阶段的功能是词法分析和语法分析，打开的文件有预处理阶段输出的两部分，还有一个用于输出有模块标记的行为代码。语法分析和词法分析会在相关章节中介绍。

5.5.3 阶段三

本阶段的功能是对模块信息进行必要的分析和转化，生成最终的模拟器代码。打开上一阶段输出的行为代码文件作为一部分输入，输出到目标代码的“C”和“H”文件中。按照配置的设置，可能还要打开生成 verilog 代码和 dot 脚本的文件。

阶段三所调用的模块包括模块分析、全局分析、代码生成、verilog 代码生成和 DOT 代码生成。后两个部分按照相关配置判断是否执行。

5.6 词法分析

词法分析模块使用 Lex 脚本编写，通过 lex 工具转化为词法分析的代码，与 Lex 有关的内容和本 ADL 语言的词法定义已经在前文中介绍过，这里只说明一下词法分析脚本的编写。

对于词法中定义的几种标记类型，大多数只要直接返回类型即可，对于数字和字符串还需要在名为“yylval”的变量中设置对应的内容。

下图所示的是词法分析的脚本实现。

```
%{
#include "front.tab.h"
%}
%%
[0-9]+                  {yylval.number=atoi(yytext); return INTEGER;}
\[                      {return '['; /* OBRACKET; */}
\]                      {return ']'; /* EBRACKET; */}
\{                      {return '{'; /* EBRACE; */}
\}                      {return '}'; /* EBRACE; */}
\(                      {return '('; /* OPAREN; */}
\)                      {return ')'; /* EPAREN; */}
\;                      {return ';'; /* SEMICOLON; */}
\=                      {return '='; /* EQUAL; */}
module                  {return MODULE;}
input                   {return INPUT;}
output                  {return OUTPUT;}
inout                   {return INOUT;}
contain                 {return CONTAIN;}
ram                     {return RAM;}
link                    {return LINK;}
\%\%                    {return CODE;}
[_a-zA-Z.][_0-9a-zA-Z.]*    {yylval.string=strdup(yytext); return TOKEN;}
[ \t\r\n]+                  {}
%%
```

图 5.6 语法分析 Lex 脚本实现

5.7 语法分析

语法分析模块分为 Yacc 脚本和处理函数两部分。Yacc 脚本的相关内容和语法定义已经在相关技术分析一章做过介绍，在此不再重复。

5.7.1 Yacc脚本编写

Yacc 脚本中需要在语法识别的过程中调用功能函数，调用函数的过程直接用 C 代码的形式在语法结构内部编写即可，C 代码块要用花括号包含。

下图所示的是 Yacc 脚本中识别“module”非终结符的代码。其中“$2”表示语法中第二个元素的内容，即 TOKEN 标记的内容。其中以“parse”开头的函数是语法处理函数。一个语法范式中可以包含多个位置的处理函数。

```
module:
        MODULE TOKEN
        {
                parse_module_start($2);
                free($2);
                if(common_error_count > 0) YYABORT;
        }
        cycle description
        {
                parse_module_finish();
                if(common_error_count > 0) YYABORT;
        }
;
```

图 5.7 语法分析 Yacc 脚本实现示例

5.7.2 语法处理函数

功能函数将语法分析识别出来的内容按照结构保存在一个模块结构体的链表中，在这些函数中并不对语法结构进行错误分析。它可以看成是 yacc 脚本的子模块。

5.8 模块分析

这一模块属于整个流程的第三阶段，这一阶段的任务是对模块进行内容检查，保证模块信息能够进入下一阶段的处理。具体任务包括子模块包含检查、构造依赖关系树、内部连接检查、构造实体树。在分析过程中，某一步骤检查到了错误，都会中断分析过程。

5.8.1 子模块包含检查

这一阶段仅检查结构描述模块中引用的子模块包含是否存在，并且检查参数配置中存放的顶层模块名是否存在。

5.8.2 构造依赖关系树

按照模块的包含关系，使用树节点结构体构造依赖关系树，递归的构造过程如下：

1. 以顶层模块作为当前模块，建立树节点，节点名称是模块的类型名。
2. 继续遍历当前模块的子模块，如果子模块的类型第一次出现，则转向（3）。如果遍历结束，则进入（5）。

3. 为此模块类型在当前节点下建立新的子节点，从子节点一直遍历到顶端，检查这一序列中是否存在相同模块类型。若有相同类型，则说明存在模块依赖，此时停止检查，返回错误信息。若检查无误，进入（4）。
4. 以子节点为当前节点，进入步骤（2）。
5. 如果当前节点存在父节点，则将父节点作为当前节点，进入（2），否则进入（6）。
6. 完成。

5.8.3 内部连接检查

本阶段进行模块内部的连接合法性检查。连接的源、目的端口应该符合的要求在前文中已经说明，这一阶段按照这一规则进行检查。

首先检查端口是否存在。模块本身的端口直接用端口名进行引用，子模块的端口用“子模块名”+“.”+“端口名”的形式声明。对于后者，在分析过程中首先要拆分这一字符串，按照子模块名查找这一模块的声明，然后在对应的模块中查找端口。然后按照语法定义中的规则检查端口的类型是否符合要求。

5.8.4 构造实体树

这一步骤和构造依赖树的过程类似。它们之间的区别是构造实体树的过程并不进行循环依赖的检查，节点名称取模块的实体名，并且每个实体都存在与其对应的模块，并不将同类的兄弟模块消除。

构造实体树的过程结束后，模块的关系已经形成了一个全局的树状网络，模块内部的各种检查已经完成。

5.9 全局分析

全局分析过程以顶层模块的实体树为基础，进行全局连接扩展和模拟过程调度。调度过程又分为初始化、分组、排序、任务组合、总线分组几个步骤。经过全局分析后，处理工具已经掌握了进行模拟器生成的全部信息。

5.9.1 全局连接扩展

由于连接的描述在模块内部，对于结构描述模块来说，它的功能是对行为描述模块进行组织，结构描述中的连接最终都实现为某两个行为模块之间的连接，而结构描述模块内的端口只是作为一种中转的形式。顶层模块的端口情况特殊，它可以看成是未完成的连接，但是在模拟器的功能中，顶层模块的端口是不推荐使用的，在此不多分析。

连接扩展的规则如下。对于两个前后连续的连接，如果承接的端口属于一个结构描述模块，那么它们将被合并成一个连接。如下图所示。

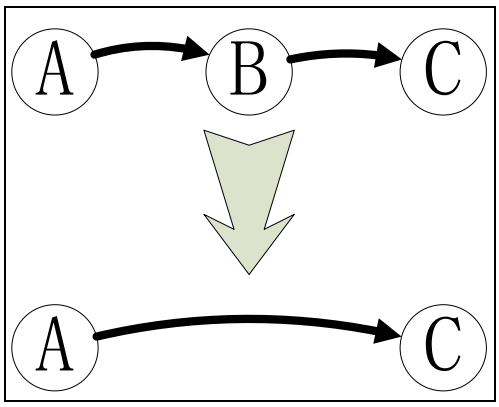


图 5.8 连接扩展示意

对于有多个输出的结构端口，每一条连接路径都被扩展成一条连接。如下图所示。

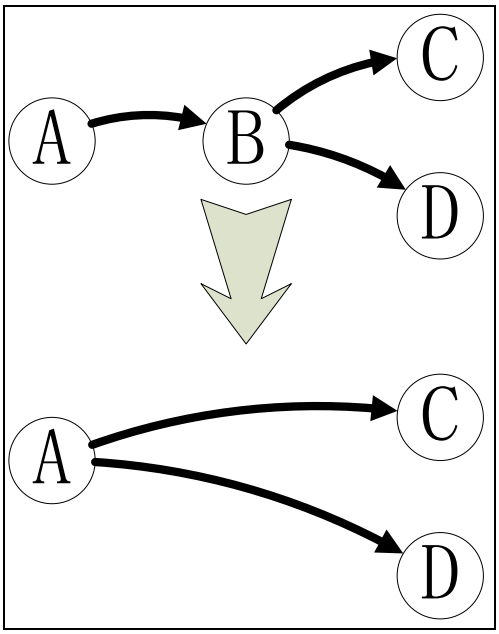


图 5.9 多扇出的连接扩展

连接扩展的操作以结构描述模块的端口为顺序，按照扩展规则，每次处理一个中转端口的连接，相当于每次消除一个中间端口，最终所有的连接都成为行为描述之间的连接。

最后还要对所有的端口进行扇入检查，原则是每一个端口只能有一个对其赋值的连接。

5.9.2 调度初始化

全局连接体现行为模块之间的数据依赖关系，同一运行周期中存在数据依赖的模块要以正确的顺序先后执行。对于一个连接，如果其目的端口属于一个 0 周期模块，那么他们之间存在数据依赖，源模块的行为要先于目的模块执行。

调度初始化的任务是将这类体现数据依赖关系的连接进行提取，为下一阶段使用。具体过程是：扫描全局连接列表，将所有目的端口是 0 周期模块的连接复制到一个列表中。

5.9.3 调度分组

分组的目的是将具有多级数据依赖的相关链接进行编号。一个设计中可能存在多个数据依赖网络，进行调度排序前要划分依赖网络，分组的编号从 1 开始依次增加。对于与其他模块不存在依赖关系的模块，被看成是不需要排序的一组，所有的总线端口也被统一编成另一组。假设一共有 n 组依赖网络，那么需要被排序的模块一共有 n 组（1 到 n），不需要排序的模块属于 0 号组，而总线端口都属于 n+1 号组。

分组的具体过程是以全局连接为单位，遍历全局连接列表。对于每一个连接，仅当目的端口所属的模块是 0 周期模块时，按如下规则操作：

- 如果连接两端的实体均未被分组，则使用一个新的分组号对这两个实体编号。
- 如果仅链接一端的实体被分组，则将另一端未被分组的实体加入这一组。
- 如果两端均已被分组，且属于不同的组，则将这两组合并，使用其中一个组号。
- 如果两端均已被分组，且属于同一组，什么都不做。

按照如上的步骤遍历所有的连接后，剩下的模块，由于可能有组合并的操作，最终

的分组号码可能是不连续的，最后还要对组号进行调整，使组号连续。分组规则的示意如下图，其中数字表示组号。

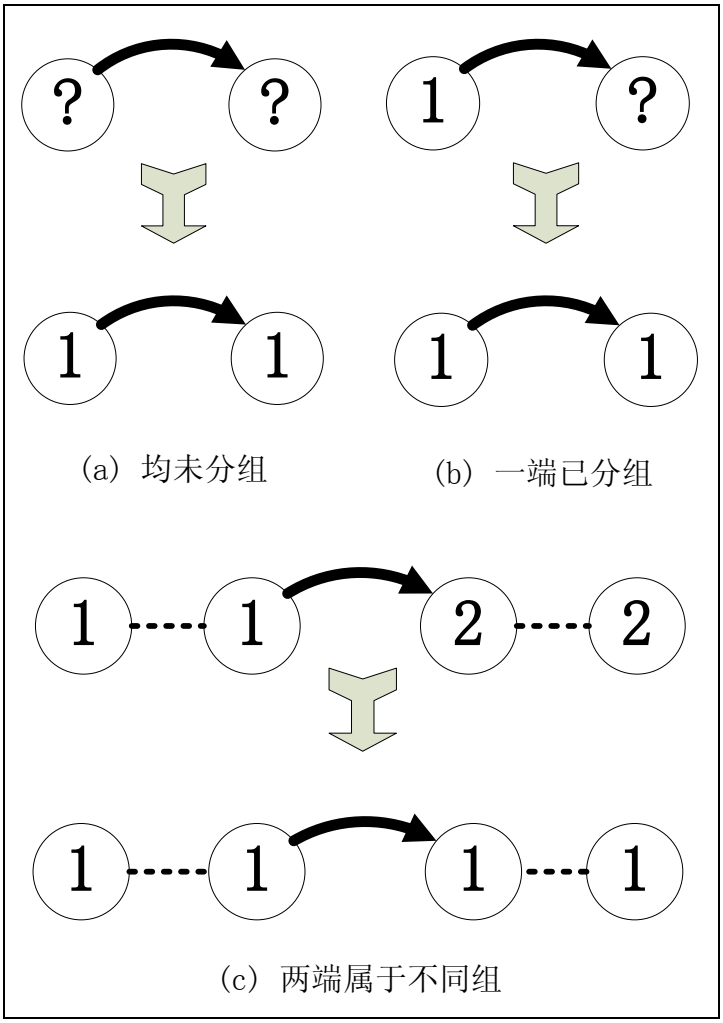


图 5.10 调度分组示意

5.9.4 调度排序

这一阶段的任务是对 n 个分组的依赖网络进行排序。

一般的线性排序算法按照一个比较方法判断出两个元素之间的先后关系，然后调整其位置，最终达到任意两个元素的位置关系符合前后关系。对于这种图结构体的排序，模块的排序依据来源于他们的连接关系，在同一网络中的两个模块之间不一定存在单向通路，这意味着不能对所有的模块之间给出确定的先后次序。

一个点与其所处的任意一条单向路径上的其它点存在次序关系，这个点与在这些路径之外的点的关系是任意的，即在逻辑上可以并行进行。因此无关系的两个点实际能够

接受任意的先后顺序。那么，可以认为对于这样的两个点，存在至少一个有效的排序关系。因此，使用一般的排序算法能够对这样的网络进行线性的排序。

排序的方法采用直接选择排序法，每次选择出一个最先执行的实体。对于无法比较的两个实体，则认为当前的最先实体先于被遍历到的实体，不进行交换。

调度排序的示例将在下一节的示例中出现。

5.9.5 任务组合

排序过程只包括行为描述模块本身的顺序，不包括连接中的值传递，这一步骤的任务是把相关的连接插入到模块序列中，形成包含模块行为和连接的任务序列。另外，把与调度无关的连接也加入到无排序任务序列中，将总线操作加入到第 n+1 组。

任务组合的过程是，按顺序遍历每个分组，将连接插入其目的端口对应的实体之前，构成先赋值再执行的行为过程。

下图所示的是任务调度和任务组合的示例。

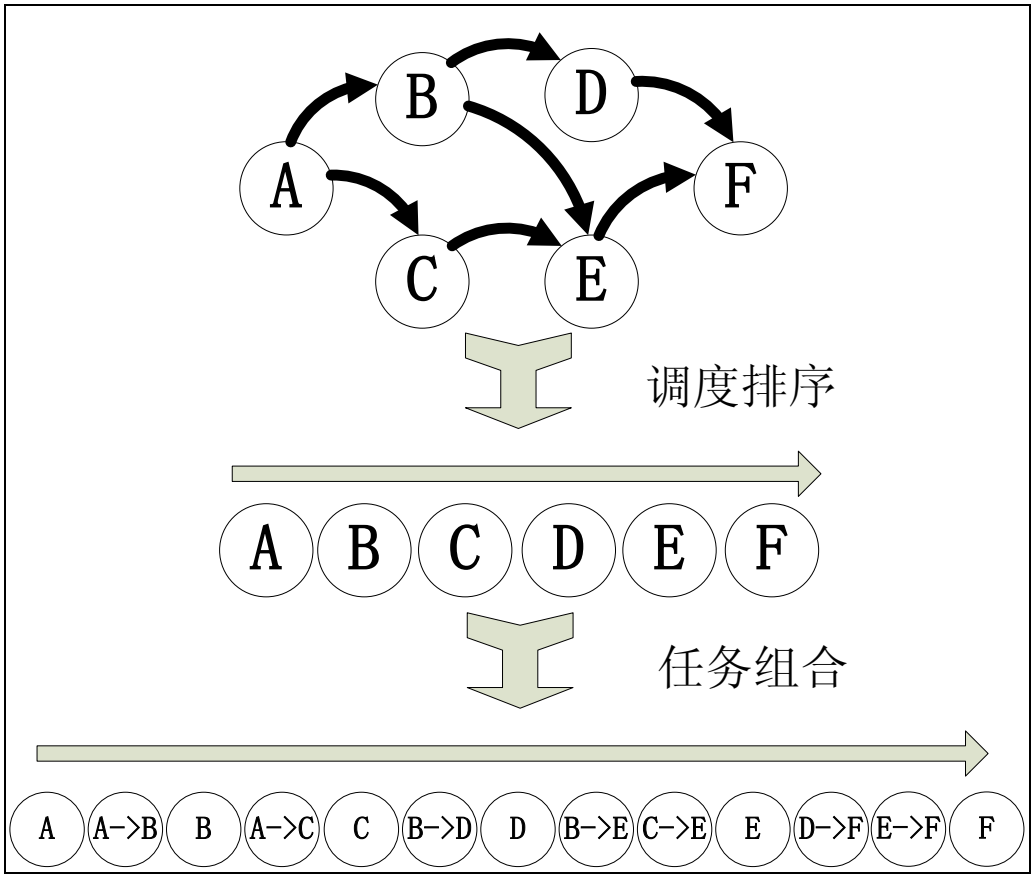


图 5.11 任务调度与组合示意图

5.9.6 总线分组

对于总线的处理是独立的，前面的步骤已经完成了模块、input 端口和 output 端口有关的所有处理，这一阶段只负责处理 inout 端口的连接。

inout 端口在描述中直接以一种总线的机制运作，而不是在 verilog 中只具有读写属性，将这里的 inout 端口理解为一种总线终端设备更便于理解。语言规范中对 inout 端口已经介绍过，为了设计的简化起见，inout 端口不能与 input、output 端口连接，是一种独立的结构。

多个 inout 的连接声明能够将多个终端挂在一个总线上，本阶段的内容就是识别这样的网络，并且进行编号整理，供后端功能使用。

总线分组的过程和连接分组类似，这里不详细描述，仅给出示意图。图中所示的是对三项 link 语句声明的 inout 连接进行组合的过程。

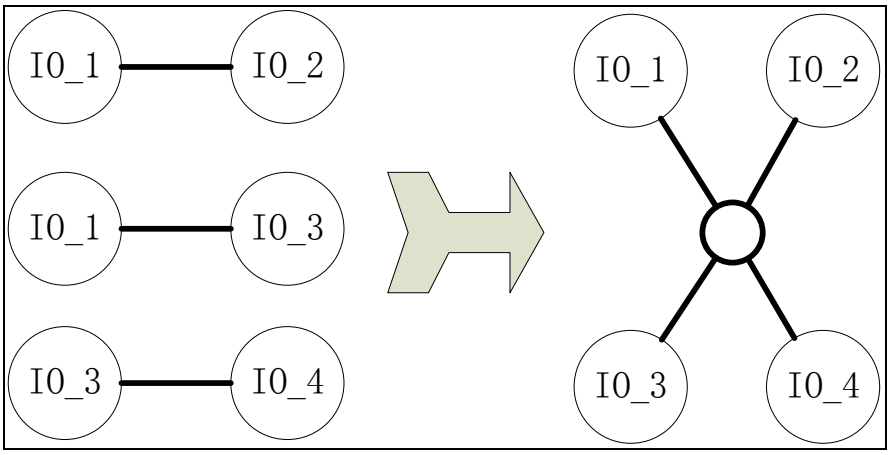


图 5.12 总线分组示意图

5.10 模拟器代码生成

自动生成的模拟器代码应该能被编译器编译成可执行文件，直接运行，为模块化设计提供支持，生成的代码也应该具有一般软件包的结构，能够方便地被其他程序引用。生成的模拟器代码综合各种考虑，设计为生成一组代码和头文件的形式，在代码文件中包含必要的功能，在头文件中提供清晰的接口。

代码生成阶段使用前端功能所分析出的结果，按照要求进行简单的转化。转化过程

比较凌乱，但是逻辑十分清晰，大部分都是非常独立的映射过程。第一节介绍生成头文件的内容，后面几节均为生成 C 代码文件的内容。

5.10.1 头文件生成

头文件的内容包括：

- 模块结构的结构体类型定义：
  按照模块列表的信息，为所有模块类型建立结构体，结构体包括模块的端口和存储，每一个模块都对应一个结构体实体，通过操作结构中的值来对端口和存储进行输入输出的操作。所有的端口和存储都采用指针的形式，这样的设计使输入缓冲和存储的实现更加方便，并且不会让结构体的声明太大。
- 接口函数声明
  这里只声明了两个函数，初始化函数和模拟执行函数。在早期的实现中，每个模块的初始化函数和行为函数都在此声明，后来考虑到接口的简洁性，将所有子模块的声明改为 static 类型，只对用户呈现顶层入口，避免对使用方法产生误解。使用任意的模块接口，只需要将其作为顶层模块进行生成即可。

5.10.2 C代码头部

这一阶段的任务是将用户定义的外部代码段复制到文件开始，以及添加必要的头文件引用声明，包括对模拟环境库的引用和对生成的头文件的引用。

5.10.3 全局实体声明

这一阶段生成的内容包括：

- 所有的实体对应的结构体变量声明。
- 相应端口、存储所对应的变量声明。多周期模块的 input 端口和 inout 端口被声明为数组，在功能函数中能够以数组的形式访问多周期的历史输入缓冲区。
- 每个模块实体所对应的初始化函数。初始化函数负责将实体结构中的指针指向正确的端口、存储变量，并且为这些端口初始化，初始值从参数配置模块获得。
- 顶层初始化函数，它调用所有的实体初始化函数。它也作为外部的接口在头文

件中声明。

5.10.4 功能函数实现

这一阶段生成每个模块类型的功能函数实现，功能函数的内容包括：

- 按照参数配置的设置，生成产生 TRACE 信息的代码。
- 用户定义的行为代码。行为代码中的端口、存储引用形式也要在此时进行转换，以“$”开始的名字将被替换成引用端口或存储变量的指针。
- 对于多周期模块，生成输入缓冲区更新的代码。

5.10.5 调度函数实现

这一阶段生成模拟调度的函数，按照全局分析阶段产生的调度信息，生成顺序的模块调用代码。

调度信息按如下顺序被转化为函数调用：

1. 顶层模块端口相关的连接的传值操作。
2. 以组为单位，按顺序输出每组的行为，包括行为代码调用和连接的传值操作。
3. 未分组模块的行为代码调用和连接的传值操作。

5.10.6 模拟器主函数

最后阶段输出主函数，也就是整个模拟器的入口，它的形式如下图所示。生成的主函数中的执行次数可以通过命令行参数进行配置。

```
int
main(int argc, char *argv[])
{
        int i;
        sim_main_initial__();
        for( i = 0; i < 10; i++ ){
                sim_main_schedule__x();
        }
        return 0;
}
```

图 5.13 模拟器主函数

5.11 结构图描述脚本生成

结构图生成和模拟器生成过程类似，也是一种比较直接的映射过程，并且更加简单，在此不详细分析。生成的目标文件只需要使用 graphviz 工具进行转化，就能生成结构图的图像。

生成的结构图中包括模块、端口和连接。结构图中以颜色区别不同的端口，红色代表 output 端口，蓝色代表 input 端口，绿色代表 inout 端口，顶层模块的端口统一用褐色表示。

下面是结构图生成代码和图形的示意。例子是一个简单的乘加器的实现，源代码略。

```
digraph ma {
        fontname="monospace";
        fontsize=40;
        style=rounded;
        node [fontname="monospace",shape=box,fontsize=32];
        labelloc = t;
        label = "ma";
        node [label = "a",shape=box,] ma_a;
        node [label = "b",shape=box,] ma_b;
        node [label = "c",shape=box,] ma_c;
        node [label = "o",shape=box,] ma_o;
        subgraph cluster_ma_a1 {
                labelloc = t;
                label = "ma_a1";
                node [label = "a",shape=box,] ma_a1_a;
                node [label = "b",shape=box,] ma_a1_b;
                node [label = "o",shape=box,] ma_a1_o;
        }
        subgraph cluster_ma_m1 {
                labelloc = t;
                label = "ma_m1";
                node [label = "a",shape=box,] ma_m1_a;
                node [label = "b",shape=box,] ma_m1_b;
                node [label = "o",shape=box,] ma_m1_o;
        }
        ma_a ->ma_m1_a [arrowhead=normal, arrowtail=inv];
        ma_b ->ma_m1_b [arrowhead=normal, arrowtail=inv];
        ma_m1_o ->ma_a1_a [arrowhead=normal, arrowtail=inv];
        ma_c ->ma_a1_b [arrowhead=normal, arrowtail=inv];
        ma_a1_o ->ma_o [arrowhead=normal, arrowtail=inv];
}
```

图 5.14 自动生成的 dot 脚本

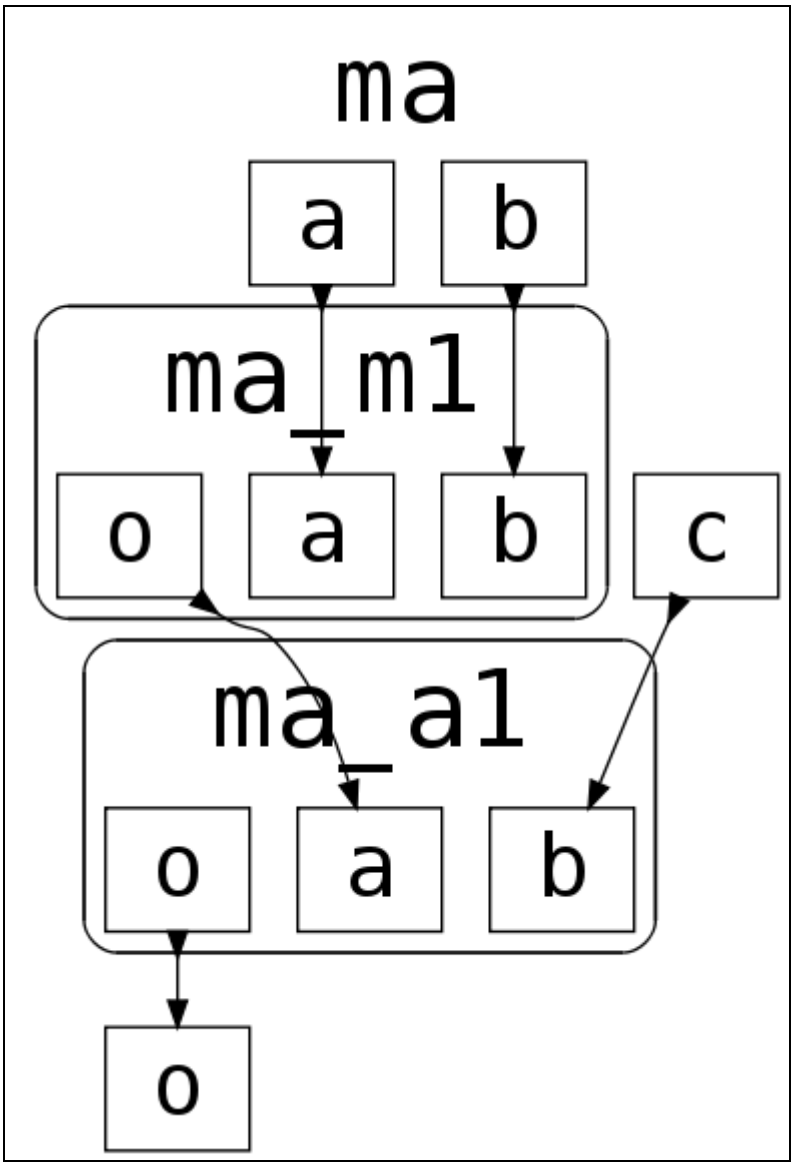


图 5.15 用 graphviz 工具转化生成的图像

5.12 verilog语言框架生成

这一功能被称为框架，是因为转化只涉及到结构描述部分，具体的行为代码并不进行转化，由于是相同结构间的转换，这一过程也非常简单，进行基本的映射即可完成，在此不详细分析。

下面给出一个转换成 Verilog 代码的实例。

```
module ma( a, b, c, o, clock);
  input [31:0] a;
  input [31:0] b;
  input [31:0] c;
  output [31:0] o;
  input clock;

  wire [31:0] a1_o;
  wire [31:0] m1_o;

  a a1(.a(m1_o), .b(c), .o(a1_o), .clock(clock));
  m m1(.a(a), .b(b), .o(m1_o), .clock(clock));

endmodule
```

图 5.16 Verilog 代码框架生成

5.13 模拟环境库

模拟环境库是与转换工具独立的一个程序包，它包含了模拟器中常用的一些信号操作、总线操作和调试等功能，开发模拟环境库的目的是简化模拟器行为代码的编写，并且减少生成代码的数量。

目前模拟环境库的内容主要包括：

- 模拟环境基本数据类型定义。
- 基本数据类型的赋值、四则运算、逻辑运算函数。
- 可用于非交互式调试的 trace 功能函数。
- 进行总线操作的功能函数。

模拟环境库作为一个静态链接库使用，文件名为“libsim.a”，在编译模拟器代码时使用“-lsim”参数连接静态库即可使用。

5.14 小结

本章以模块为单位，详细介绍了模拟器生成工具的设计与实现，从基本的数据结构，到具体的处理过程，完整地说明了工具的工作原理。

6 测试与验证

6.1 测试环境

测试机为 Intel x86 架构，CPU 主频为 3.25GHz，安装 Windows XP sp3 操作系统，运行环境是使用 Vmware 6.5 虚拟机构件的 Ubuntu 8.04 Linux 操作系统，内核版本为 2.6.24-24 generic。模拟器的编译使用 GCC 4.2.4，在编译时使用“-O0”选项关闭优化。

6.2 测试样例

在本项目的开发过程中，代码编写和测试的过程是一同进行的，测试对开发有很好的指导作用，能够使开发更加高效。测试样例的编写也是随着开发逐渐完善的，从最初的用于语法分析的测试样例，到后期的完整应用测试，几个侧重点不同的测试对这一工具的转化能力进行了比较完善的验证。

6.2.1 测试样例一

本测试样例在开发阶段使用，测试文件中的模型结构尽量包含各种语法元素与形式，而较少存在重复，目的是验证语言分析过程的正确性。

测试代码包括一个顶层模块，内含三个结构描述的子模块。第一个子模块由一个零周期的加法器和一个零周期的乘法器组成一个乘加器，目的是测试零周期模块的转化与组合。第二个子模块由一个多周期的加法器和一个多周期的乘法器组成，目的是测试多周期模块的转化与组合。第三个子模块是一组总线连接的模块，目的是测试总线的综合。测试样例中还包括了一些未使用端口，用于测试相关的警告信息功能。

这个测试样例一直作为开发过程中的基准测试程序，每个模块的实现都以正确处理这一模块为标准。

6.2.2 测试样例二

本测试样例是由多个加法器组成的一个 8 位输入 16 位输出的乘法器。在这个模块中涉及到较多的位操作，这些操作都需要单独的位操作的模块，整个模块的实现结构较

复杂。在 Verilog 语言的描述中，位拼接操作能够为这样的按位连接提供很好的支持，极大地简化描述难度，在今后的改进中，可以考虑支持位拼接形式的连接声明。

在测试中，对加法单元的周期属性进行调整，以变换模块的执行周期，测试多种周期特性下的模块行为，均得到了正确的结果。

6.2.3 测试样例三

本测试样例实现了 PSDM 算法叠前偏移部分的四个运算阵列模块。这四个模块在所属的《应用加速器》课题中作为 FPGA 实现的部分，这些运算模块以乘法器和加法器为主要成分，还包括了一些数学运算单元，如三角函数运算、浮点数取整等模块，每个运算阵列包含 20~30 个基本运算模块。后面的分析主要以第一个阵列为主。

下图所示的是第一个运算阵列的结构示意图。

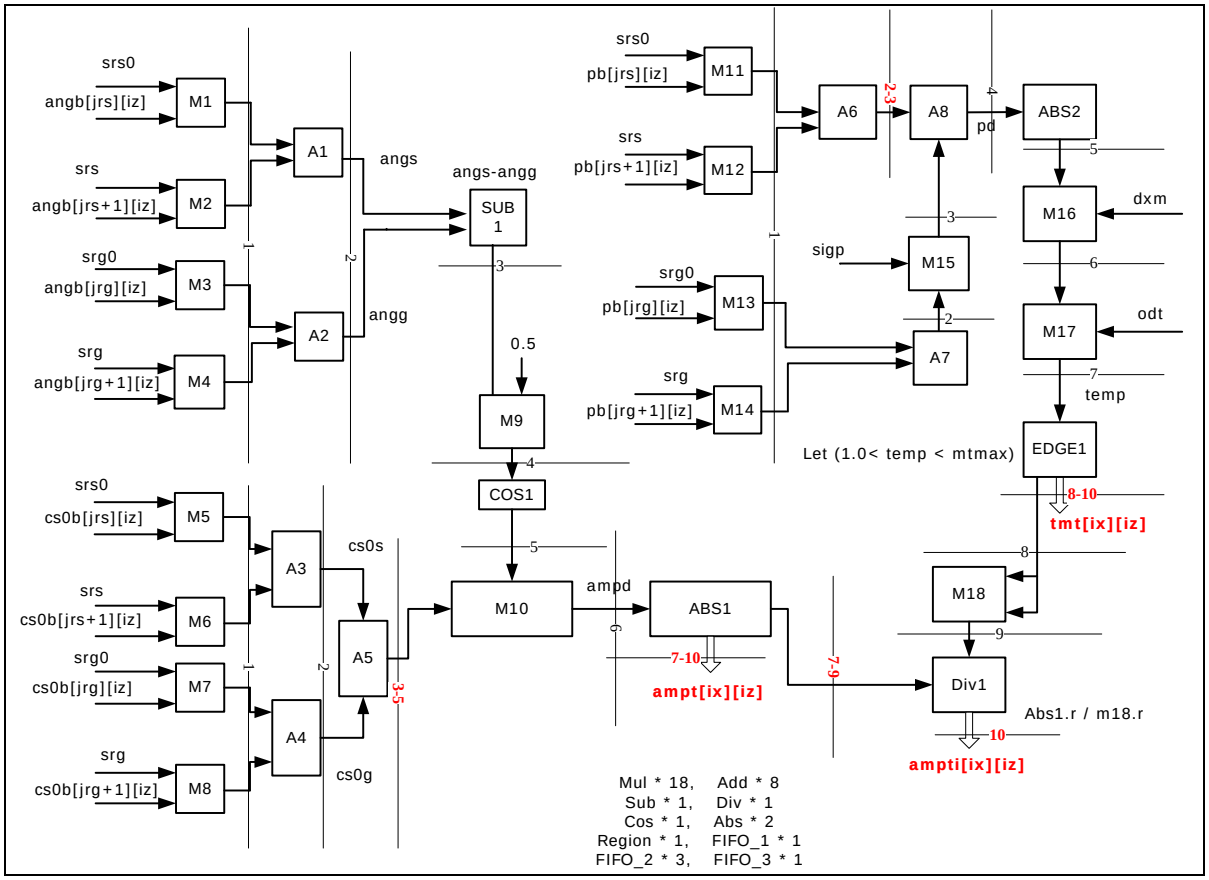


图 6.1 PSDM 叠前偏移第一阵列结构图

这项测试能够验证模拟器生成工具对真实系统进行建模的能力。将模拟器代码和原始程序的代码结合，还能说明这一工具在软硬件协同设计上的验证能力。运算阵列的原

始程序由 C 语言编写，在测试中，通过在原始程序中加入采样代码，将阵列所对应的代码段的输入输出值进行提取。在 ADL 描述代码中，使用测试模块将采样的输入值传送给模拟阵列进行运算，然后获取输出值，和采样的输出值进行比较。测试一共进行了 50 组，均得出了正确的输出，说明生成的模拟器能够正确地实现所描述的功能。

6.2.4 测试样例四

本测试样例是AES（Advanced Encryption Standard）[16]的 128 位密钥的加密部分。

AES 的算法是 Rijndael，由比利时密码学家 Joan Daemen 和 Vincent Rijmen 设计，是一种对称密钥的块加密算法，AES 于 2001 年由美国国家标准与技术研究院发布。AES 的密钥长度可以是 128 位（16 字节）、192 位（24 字节）或 256 位（32 字节），分别对应的加密的轮数为 10、12 和 14 轮，加密的过程分为加密主体过程和密钥生成两个部分，测试所实现的是一个多周期的 128 位密钥版本的加密功能，不包含解密的过程。

下图所示的是所实现的结构示意图，较大的箭头表示顶层模块的端口名。

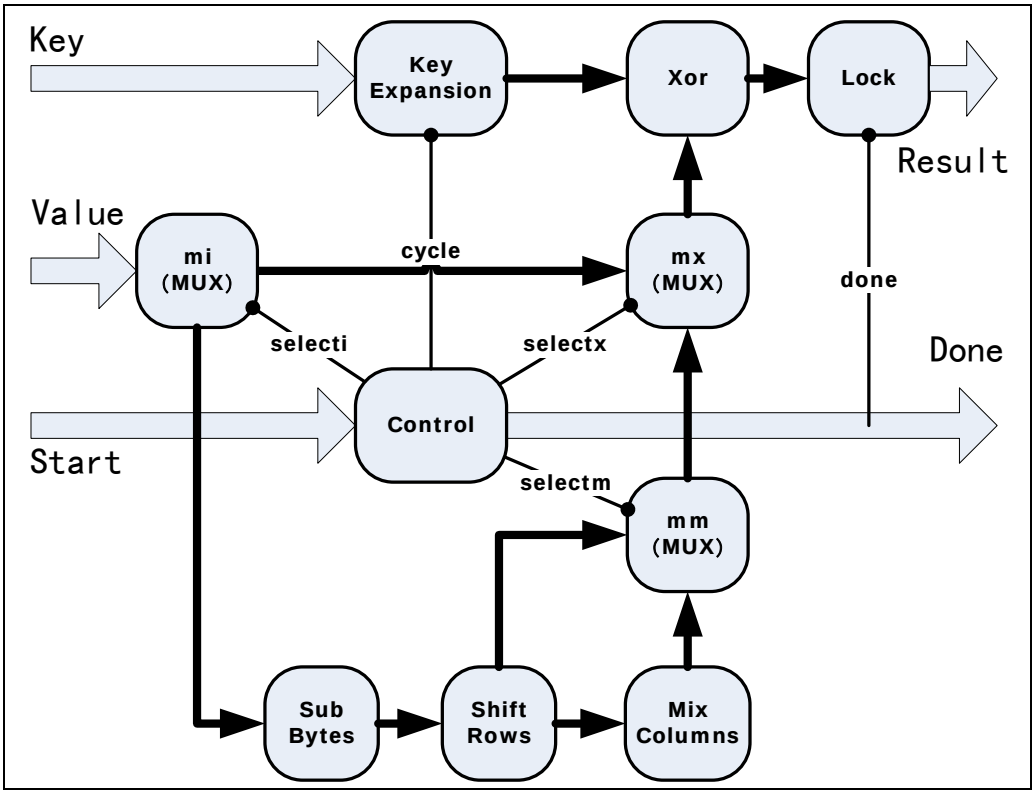


图 6.2 AES-128 加密实现结构图

加密模块和一个测试模块对接，输入明文、密钥和启动信号，等待 10 个周期后得

到完成信号和密文，经验证密文正确。AES 算法作为一个真实的应用，对其正确的实现已经能够说明工具的建模能力和模拟器构造能力。

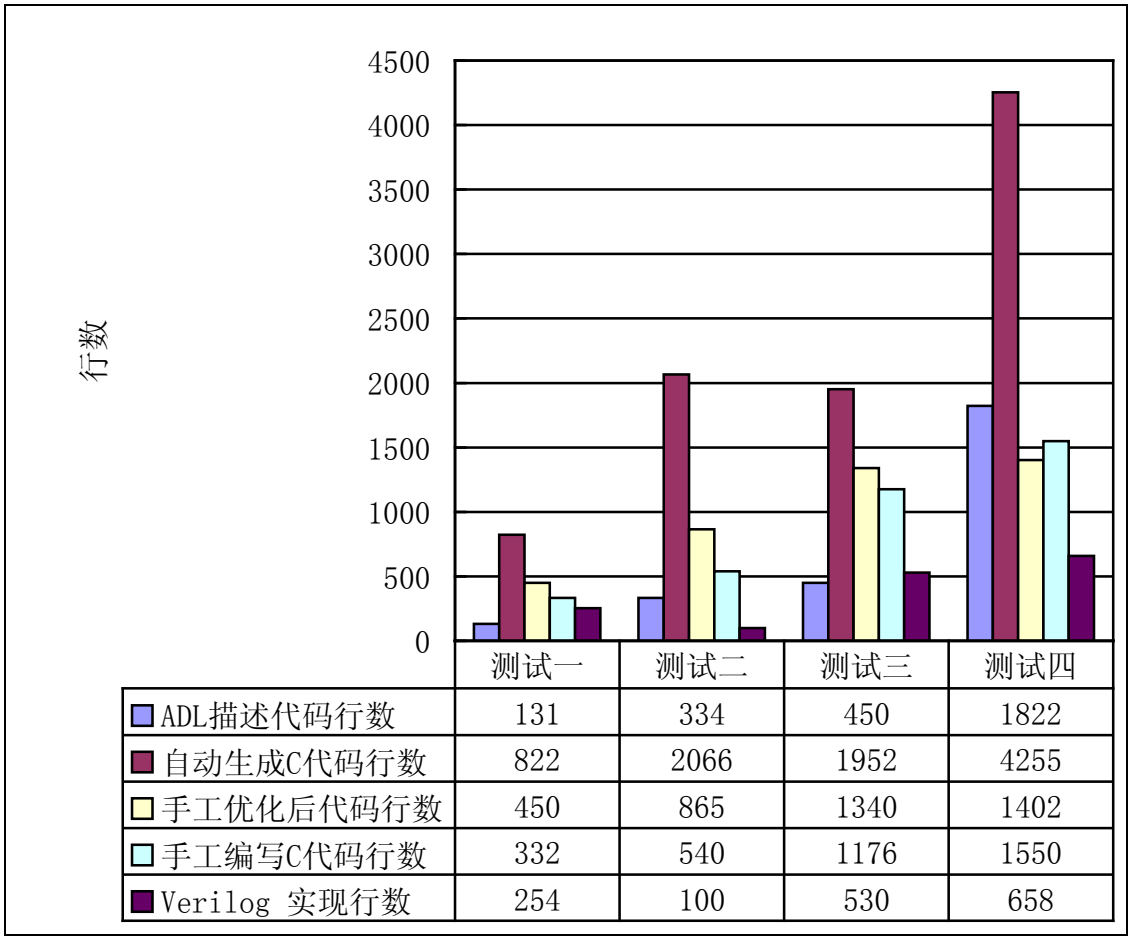
在对算法的实现过程中，由于数据为矩阵形式，大部分各种端口是以一组 16 个的形式出现，在本描述语言中还不支持数组形式的端口，因此编写端口声明和连接的时候总要对一组连接分 16 项声明，在以后的设计中可以考虑对数组形式的端口的支持。

6.3 结果分析

前文的几个测试样例从不同的角度对工具进行了测试。下文将从从具体分析代码生成的效果与引发问题的原因，以尽量说明描述语言与转化工具的实际能力。

6.3.1 代码量分析

下图给出了有关代码量的相关统计数据，其中测试三仅分析第一个阵列的数据。



	测试一	测试二	测试三	测试四
ADL描述代码行数	131	334	450	1822
自动生成C代码行数	822	2066	1952	4255
手工优化后代码行数	450	865	1340	1402
手工编写C代码行数	332	540	1176	1550
Verilog 实现行数	254	100	530	658

图 6.3 测试样例代码量统计

首先要说明，不同类型的代码抽象级别不同，所用的描述也不同，并且每一行代码的信息量和需要的工作量均不相同，并不能简单地从代码行数比较不同描述语言的优劣。不同的测试样例在数据的走向上也不各不相同，这是因为它们的描述内容不同。

本论文设计的 ADL 语言结构简单，并且语法形式单一，每一行描述只包含少量的信息。但是连接和包含等描述均不用说明具体的传递过程。C 代码由于包含很多的底层处理操作，包括初始化、数据拷贝传递等，所以在描述上要多于 ADL 描述。

Verilog 代码注重实现，代码的凝聚性很高，而且 Verilog 是面向实现的，在描述上只具有相同的粒度。而 ADL 对系统进行描述的代码量与实现的粒度具有很大关系，对于越高层次抽象的系统，ADL 描述和 Verilog 相比，越能节省代码。

对于测试样例一，其描述粒度以加法器和乘法器为最小元素，这种粒度与两外两种描述形式具有类似的复杂度，其代码量的比例关系比较符合数学运算结构的建模特点。

对于测试样例二，其 Verilog 代码数量非常少。由于描述中涉及到大量的位拼接操作，ADL 语言在这方面能力较弱，需要编写专门的模块来处理位拼接，而 Verilog 中则在连接中直接使用拼接操作即可轻松完成。

对于测试样例三，和样例一在描述粒度上相同，代码上也具有类似的走向。

对于测试样例四，它的 ADL 描述偏高，原因在于这个设计中端口数量很大，每个模块都有大约三组 4 行 4 列的矩阵输入或输出，由于 ADL 语言的描述还不支持以数组形式声明端口，每组端口需要用 16 行代码声明，这造成了 ADL 描述的代码量过大。

自动生成的代码行数较多，有如下几个主要原因：

- 为自动处理过程生成了较多的注释，优化后的代码和手写的代码没有注释。
- 为了使结构清晰，对代码块、函数和变量声明添加了很多空行。
- 若值全初始化为 0，在初始函数中也采用多条语句为结构体赋值。而在手写代码中，如果全初始化为零，则可以使用“bzero”这类函数简单的进行清零。

以 AES 的模拟器精简过程为例，注释的行数为 527 行，空行约为 700 行，用于变量初始化的代码为 2000 行，由于设定初始值为 0，这一过程可以通过使用编程技巧进行简化，初始化的代码能够精简为 200 行以内，总代码行数缩减了近 3000 行，但是优化

后的代码的执行效率并没有变化，而可读性明显下降了。手工编写代码还能够应用一些编程技巧，这对自动生成功能来说难度较大，但是可以在以后的改进中逐渐研究并实现。

6.3.2 模拟器执行效率

对模拟器的执行效率进行了测试，结果如下表所示。模拟器的运行效率约为每秒运行百万周期，完全能满足一般的模拟需要。

表 6.1 模拟器运行效率

编号	运行一百万周期耗时（秒）
1	0.196
2	0.570
3	0.352
4	1.188

对于手工优化的模拟器代码，其运行效率具有极小的提升空间。自动生成的代码主要在形式上存在较多冗余，在数据流向上和手工编写的代码是相同的。代码形式上的冗余完全可以通过编译器的优化技术来消除。现代编译技术中的无用代码消除、内联函数等方法能够很好地化解自动生成的代码在结构上带来的计算负担。

使用 Modelsim 对测试四的 Verilog HDL 实现进行仿真，并统计仿真时间。以周期为单位，将模拟时间精确度调到最低（使速度最快），使用 modelsim 仿真 AES 模块一百万个周期需要大约 30 秒。由于 Modelsim 工具采用数据流驱动的仿真方式，这是一种用于 RTL 级电路仿真的方法，可以理解为更少的数据改动会更加节省仿真时间。由于没有足够长的测试用例，在漫长的运行周期中的绝大部分时间都没有实际的数据流动，使得仿真速度很快。在以往的经验中，使用 Modelsim 对类似的系统进行一百万周期仿真会消耗很长时间。使用模拟器进行仿真，即使是在数据流改动很小的情况下，也能比 RTL 级仿真快至少一个数量级。

6.4 小结

根据前文中所述的一些语言的设计缺陷和工具的功能缺陷来看，语言的描述能力和工具的转换能力还存在很大的提升空间，现在的测试对以后的改进给出了很好的参照。

结 论

本论文完成了一种面向模拟器生成的描述语言的设计，并且实现了一套相应的模拟器生成工具。在语言的设计过程中，参考了多种相关的设计语言，包括 C 语言、Yacc 语法、Verilog HDL、AADL、UML、System C 等，并且在所涉及的语言中引用了它们的一些优秀特性。在语言的描述能力上，语法本身的结构设计以及对预处理功能的支持使用户能够方便地组织代码，或者与其他程序交互使用模拟器。语言中提出了多周期模块的历史输入缓冲区概念，为构造复杂逻辑的模块提供了一定程度的便利。在模拟器生成的功能上，除了能够准确无误地处理所有测试样例外，还能够生成用于评估、分析的结构图和 HDL 代码。

在课题的实际应用中，这一流程已经被应用到加速部件设计的任务中，并且已经给出了一些有价值的反馈，起到了一定的参考作用。本方法能够提供对体系结构建模、模拟和评估的支持，简化了系统结构设计的深度和难度，达到了预期的设计目标。

模拟器转化工具与模拟环境库的代码共计七千余行，用于测试的 ADL 描述文件共计三千余行。

在设计、开发和测试的过程中，也不断的遇到困难，出现问题。从开始到最后，对语言规范和功能的修正和增减也一直在进行，发现的各种错误均已被修正，语言设计和功能设计上的缺陷和不足也进行了一定的完善，还有一些问题有待今后解决。

有待解决或改进的内容主要有：

- 端口的声明形式若能支持数组的形式，将极大地简化用户描述的难度。
- 数据类型的支持比较薄弱。问题主要表现为不能声明任意信号宽度的端口，不能进行“位拼接”形式的连接声明。
- 支持以组为单位声明端口。两个器件间的对应的多个连接与端口可以用更简化的形式进行声明，而不是每一个端口和每一个连接都单独声明。
- 生成的 C 代码在性能上还有提升空间，包括缓冲区自动维护的代码、初始化的方式和代码等。进行一定的调整后，生成代码的数量和效率将会有所提高。
- 对交互性调试功能的支持，现有的 trace 信息输出只能支持非交互式调试，并不

支持在运行时修改端口和存储器的值。交互式调试将使验证能力更加强大。

- 加入图形化的设计方法，是结构组织和分析更加直接，也更加清晰。自动生成的逻辑结构图的组织排列很难符合人的思维方向，直接以图形的方式进行设计更方便理解。

致 谢

我要特别感谢我的指导教师高小鹏老师。高老师知识渊博，经验丰富，在我的毕业设计阶段和以往的实习中都给予我很大的关心与指导，在学术研究上为我答疑解惑，在思想德育和人生规划上与我分享心得，遇到挫折的时候鼓励我坚持，使得我能够不断进步。

我也要特别感谢我未来的导师龙翔老师。龙老师在科学研究上的深厚造诣，以及严谨的治学作风一直影响着我，使我一直以来的榜样。在我的求知道路上，龙老师一直给予我极大的帮助与支持，使得我能有良好的学习环境。

感谢大学里所有的授课老师，从你们的课程中我学到了丰富而广泛的知识，掌握了更多的学习方法，明白了更多的道理。

我还要感谢和我一起工作过的同学、师兄、师姐们，他们在我遇到困难时都毫不犹豫地伸出了援助之手，帮助我解决了种种困难，能和他们在一起学习工作是我的荣幸，我永远也不会忘记这段共同奋斗的日子。

感谢 3506 大班，尤其是 350601 小班的同学们，以及班主任欧阳元新老师、曹远和张莉两位辅导员。四年来你们在学习和生活上给予我的帮助与包容，令我十分感动。

将最诚挚的感谢献给我的父母，是你们由衷的关心、支持和无私的爱，使我不断奋发和前进，祝你们永远身体健康。

参考文献

[1] 高小鹏, 牛建伟, 杨钦. 《加速部件及其应用》课题研制计划报告 [R]. 北京航空航天大学计算机学院, 2008.

[2] Peter H. Feiler, David P. Gluch, John J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction [R]. Pittsburgh U.S.: Carnegie Mellon University, 2006.

[3] 算法综合技术 Catapult C Synthesis [M/OL]. http://www.acconsys.com

[4] bert hubert <bert@powerdns.com>. Lex and YACC primer/HOWTO [EB/OL]. (2002-04-20)[2009-2-23].

[5] David Pellerin, Scott Thibault. 实用 C 语言 FPGA 编程 [M]. 北京: 机械工业出版社, 2007.

[6] Clive Maxfield. FPGA 设计指南：器件、工具和流程 [M]. 北京: 人民邮电出版社, 2007.

[7] 薛小刚, 葛毅敏. Xilinx ISE 9.X FPGA/CPLD 设计指南 [M]. 北京: 人民邮电出版社, 2007.

[8] 陈曦, 徐宁仪. SystemC 片上系统设计 [M]. 北京: 科学出版社, 2004.

[9] AADL (Architecture Analysis & Design Language)[EB/OL]. http://www.axlog.fr/aadl/aadl_en.html. 2007-02-09.

[10] SAE AS5506 Annex: Behavior_Specification V2.0 September 20, 2007.

[11] Donald Bell. UML 基础：统一建模语言简介 [EB/OL]. http://www.ibm.com/developerworks/cn/rational/r-uml/index.html

[12] Bram Moolenaar. VIM 用户手册_自定义语法高亮 [EB/OL]. http://vimcdoc.sourceforge.net/doc/usr_44.html. 2006.

[13] The DOT Language [EB/OL]. http://www.graphviz.org/doc/info/lang.html.

[14] Peter Seebach. 使用 lex 和 yacc 编译代码，第 1 部分：介绍 [EB/OL]. http://www.ibm.com/developerworks/cn/linux/l-lexyac.html. 2004-9-23.

[15] Shenglin Gui, et al. UCaG: An Automatic C Code Generator for AADL Based Upon DeltaOS[A]. International Conference on Advanced Computer Theory and Engineering,2008:346-350.

[16] Federal Information Processing Standards Publication 197, Announcing the ADVANCED ENCRYPTION STANDARD (AES)[S]. 2001.

[17] Lars Grunske and Jun Han. A Comparative Study into Architecture-Based Safety Evaluation Methodologies using AADL's Error Annex and Failure Propagation Models[A]. IEEE High Assurance Systems Engineering Symposium. 2008:283:292.

[18] Michael Butts, Anthony Mark Jones, Paul Wasson. A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing[A]. International Symposium on Field-Programmable Custom Computing Machines. 2007:55-64.

[19] GREG STITT and FRANK VAHID. Binary Synthesis[J]. ACM Transactions on Design Automation of Electronic Systems, Vol.12, No.3,Article 34, Publication date:August 2007.

[20] Souvik Basu, Rajat Moona. High Level Synthesis from Sim-nML Processor Models[A]. Proceedings of the 16th International Conference on VLSI Design, 2003.

[21] YONGJIN AHN, et al. SoCDAL: System-on-Chip Design AcceLerator [R]. ACM Transactions on Design Automation of Electronic Systems,Vol. 13,No. 1,Article 17, Pub. date: January 2008.

[22] Louis Scheffer, et al. 集成电路系统设计、验证与测试[M]. 北京：科学出版社, 2008.