申请上海交通大学硕士学位论文

模型编译器及其在互连建模和信号完整性分析中的应用

学校：上海交通大学
院系：微电子学院
班级：B0421091
学号：1042109027
工学硕士生：洪杰
专业：计算机体系结构
导师 I：付宇卓 教授
导师 II：施国勇 教授

上海交通大学微电子学院
2007 年 1 月

A Dissertation Submitted to Shanghai Jiao Tong University for the Degree of Master

Model Compiler and the Application in Interconnect Modeling, Signal Integrity

Author: Hong Jie

Advisor I: Fu Yuzhuo

Advisor II: Shi Guoyong

School of Microelectronics
Shanghai Jiaotong University

摘要：在大规模集成电路发展迅速的今天，器件模型的研究也日新月异。传统的手工实现器件模型的方法难以跟上模型的变化。作为一种快速开发模型的工具，模型编译器被应时提出。

本文实现了一个自动化的模型编译器CAMC。CAMC能够接受使用Verilog-AMS等行为级语言描述的器件模型，并按照标准电路仿真器的接口产生C语言代码。这些代码经过C语言编译器，可以和电路仿真器协同工作。本文同时介绍了CAMC的具体实现、优化算法和语法扩展。

在高速电路设计中，反射等互连问题严重影响了信号完整性。本文尝试使用模型编译器进行互连建模，并验证一些互连设计中的影响因素。

本文的第一章讲述了模型编译器的历史背景和基本概念。第二章讲解了编译器实现的基础知识。第三章讲述了模型编译器的原理，一些电路仿真的原理和传输线模型的概念。第四章着重说明了模型编译器的前后端细节，以及一些验证信号完整性问题的实例。

关键字：模型编译器，电路仿真器，互连建模，信号完整性

Abstract: With the fast development of VLSI and more device models, traditional implementation method of device model in hand-coding can’t be satisfied, and module compiler is introduced at last, as a automatic tool for fast modeling.

This paper presents an automatic model compiler, CAMC, which accepts device models in behavioral language, such as Verilog-AMS, and generates C code according to standard circuit simulator programming interface. The code generated can be compiled with a circuit simulator after compiled by a C language compiler. This paper also gives the detailed implementation and optimization algorithms of CAMC.

Some interconnecting problems, reflection etc., heavily impact on signal integrity in the design of high speed circuit. This paper gives a way in probing some issues in interconnect design.

In Chapter I, the historical background of model compiler is introduced. In Chapter II, the basic concept of compiling is explained. In Chapter III, the algorithm of automatic model compiling is studied, the algorithm of circuit simulating and the model of transmission lines is also introduced. In Chapter IV, the main work of this dissertation is discussed, including the details of the front end and back end of CAMC, and the application of CAMC in Signal Integrity.

Keyword: module compiler, circuit simulator, interconnect modeling, signal integrity

目录

第一章　引言

1.1 背景

随着大规模集成电路设计进入深亚微米时代，电源完整性、信号完整性等问题日益突出。在工业生产中，产品的质量越来越依赖于工艺的进步；在研究实践中，对实验结果的精细估计往往也取决于模型的精确程度。因此新的器件模型被不断提出，从 BSIM3 到 BSIM4，再到 BSIM SOI，越来越复杂的模型被不断引入。

但是，从完成模型的理论描述，到将其应用于科学实践中，还有很长的路要走。在传统的模型实现过程中，模型本身和电路仿真器的结构混合在一起。因此，模型的开发者不仅需要对模型本身有着深刻的理解，还需要有着高超的编码能力和丰富的工程开发经验。这样跨学科的高要求，使得模型的开发过程艰难而缓慢。典型的，一个 BSIM SOI 的实现大约需要半年至一年的时间。更严重的是，在后期，对于代码可靠性的测试，往往耗时更巨。任何对模型的屑小改动，则需要重复先前的开发过程。

抛开在其中耗费的大量人力物力不谈，时间的损耗是最为关键的。无论在实验室中，还是在生产线上，研发人员总是希望使用最新的模型来验证他们的设计。比如在.09ns 的工艺下，漏电流对芯片功耗有着决定性的作用，而这不能从设计角度得到任何估计，必须在模型中验证。模型的延期交付，对工业生产的打击是致命性的。

模型编译器正式为了解决这一问题而提出。正如蒸汽机将纺织业从手工作坊推进到机械化流水线生产一样，模型编译器免除了模型开发者的编码重任，也将电路开发者从急缺模型的桎梏中解脱出来。模型编译器接受使用行为级语言（如 Verilog-AMS）描述的模型作为输入，并输出使用底层语言（如 C 语言）实现的代码。这些代码能够被编译，并链接入现有的模型编译器，进行协同工作。这样，模型编译器使得模型的自动化生成成为可能。模型编译器简化了模型的实现步骤，缩短了模型的交付时间，更为重要的是，模型编译器隔离了模型的研究、实现和使用这三个阶段，使得开发人员能够专注于他们自己的领域，大大提高了研发速度。

在实际的使用中，我们还发现，模型编译器还充当了一个标准化的角色。现有的电路仿真器在模型的实现过程中，总是会私自作一些修改，使得开发人员难以在不同的仿真器间进行比较和甄别。使用模型编译器，在各个仿真器间建立了一个标准化的接口，便于进行相应的比对。

从模型编译器的提出到现在，已经有了诸多相关工作[1][2][3][4][5][6][7]。但是，模型编译器还有很多困难，其应用领域也有待于进一步拓展。

1.2 模型编译器的工作流程

典型的，一个传统的的模型开发过程如下：

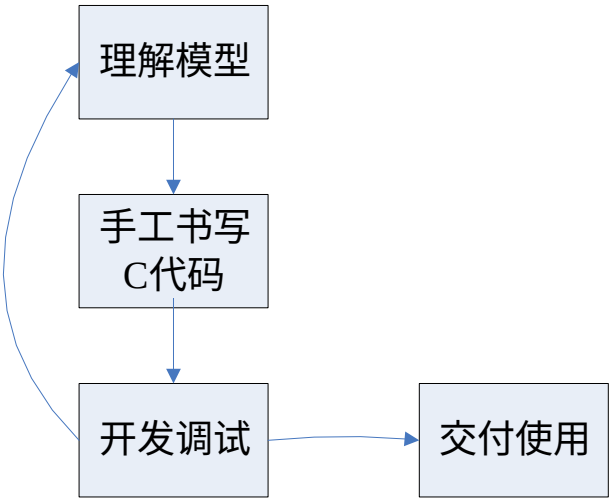


图 1 旧的模型开发过程

显然，模型的实现者是连接模型的开发者和模型的使用者之间的唯一桥梁。对于模型的生成和使用，以及在实现过程中的一些技术细节，模型的实现者必须有相当深刻的理解。而一个模型编译器的工作流程如下图所示：

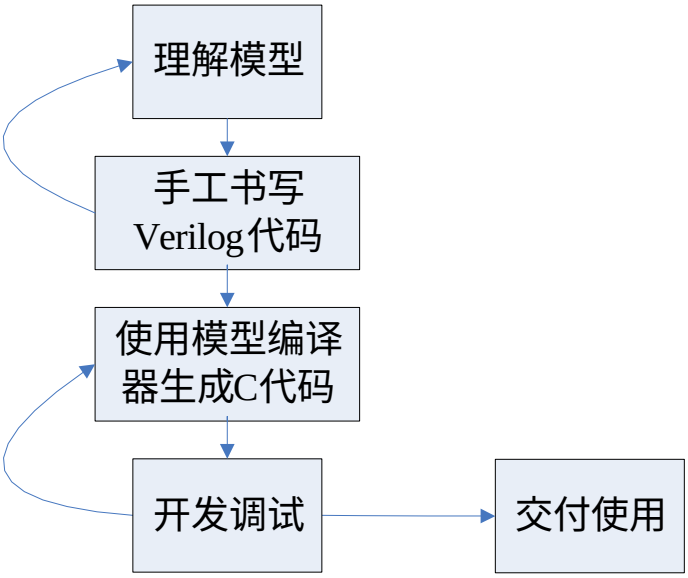


图 2 使用模型编译器的开发流程

在这个流程中，模型的实现者不再关心后端代码的实现细节，也不再需要了解仿真器的语言接口和工作方式，而是可以把精力集中在验证模型本身的正确性，以及现有的行为级描述是否和模型本身等价。

1.3 模型编译器的技术和应用

1.3.1 建模语言

现有的模型编译器，都接受 Verilog-AMS 或 VHDL-AMS 的输入。这些语言的初衷，都是面向模拟系统的设计和验证的。因此，使用这些语言描述和实现既有模型，和使用 C 语言建模，有较大的区别。这些语言，都有和模型相对应的模块、网络、信号、端口等概念，在结构上符合模拟系统的特征；在行为的描述上，这些语言都含有一些内建的算子，可以直接描述刻画模型行为的常微分方程(ODE)。比如，使用 Verilog-AMS 描述电阻模型的代码可以如下所示：

```
module resistor (a, b) ;
inout a, b ;
electrical a, b ; // access functions are V() and I()
parameter real R = 1.0 ;
analog
I(a,b) <+ V (a,b) / R;
endmodule
```

可以看到，这段代码中，对电阻的两个节点进行了定义，指出了电阻本身可供使用的电压和电流属性、电阻参数，以及它们之间的关系。这样，一个简单的电阻模型就被完整地定义了。

1.3.2 编译原理和工具

从本质上来讲，编译原理涵盖了所有从一种语言到另一种语言的翻译动作，比如从英语到汉语的自动化翻译动作。而从一种计算机语言到另一种计算机语言的翻译，也属于这一范畴。我们常用的 C 语言编译器，就是从 C 语言翻译成汇编语言，而 Verilog 综合工具，则是从 Verilog 语言翻译成网表语言。这些工具的基本特征是，是从一种高级语言翻译成一种低级语言。这是因为人们总是更善于使用具有高度归纳性的高级语言，而计算机只能处理面向机器的低级语言。

模型编译器的不同之处在于，它是在两种不同的高级语言之间进行翻译。这两种语言具有不同的起源、不同的背景和不同的应用，这就使得在这两种语言之间进行解释，比从一种高级语言编译到一种纯粹的低级语言更为艰难。

即便是这样，我们还是可以使用现有的编译原理和流行的辅助工具。在一般编译器的生成过程中，广为使用的是词法分析工具 Lex 和语法分析工具 Yacc。这两个工具使得使用者从繁琐的底层数据结构编写和纷繁复杂的语法整理中解脱出来，能够更专心于编译过程的实现。

1.3.2 电路仿真

和数字集成电路的仿真不同，模拟集成电路的数值仿真还局限于相对较小的规模。现有的电路仿真器也大多从改进节点法出发，列出电路分支方程，并求解这些方程，从而得到各个节点电压和分支电流的解。在求解过程中，因为方程组

的维数非常大，一般需要采取一些特定的方法，比如常用的稀疏矩阵的求法。

在列举电路方程组得时候，已有的电路仿真器通常采用所谓的“ 邮票法 ”(STAMP)。这种方法都是针对一些固定的器件模型，给出相应的节点导纳矩阵，并填充到整个电路方程组得矩阵中。这种方法将电路归纳为几个基本的类型，易于实现编程的自动化。而所谓的“ 邮票 ”，也是由开发人员依据现有的模型归纳实现的。

模型编译器的应用对象是各种新的期间模型，也有着各种不同的电路结构。因此不能局限于现有的“ 邮票 ”，而是要从给出的器件模型出发，求出各个分支电流对于各个节点电压的贡献，并依据基尔霍夫电流定律和基尔霍夫电压定律，列出平衡方程，供现有的电路仿真器求解。

1.3.3 互连设计和信号完整性

电路设计的基本思想是进行信号的通讯。各个模块、端口之间通过发送和接收电压、电流波莱传递信息。以前，互连本身对整个电路电气性能的影响是很小的。工程师常常可以忽略这些影响，或仅仅凭借经验法则来解决这些问题。但是，在 PCB 设计和 VLSI 中，随着时钟频率的提高、系统速度的增长，互连变得越来越重要。在这些系统中，互连的导体不能再被看作是一根简单的导线，而是呈现了高频效应的传输线，由此而导致的一些和互连相关的效应如信号崎变等将发生在所有级别的互联中。因此，对这些传输线的设计、对传输线模型的有效估计，在整个系统设计中占有重要的作用。

1.4 本文安排

本文章节安排如下：第二章介绍写编译器所需的编译基础理论，和编译器生成辅助工具 Lex 和 Yacc 的使用方法。第三章解释了电路仿真的一些基本原理、自动模型编译的算法，以及基本的传输线理论。第四章详细讲叙了模型编译器 CAMC 的前后端实现细节和其在互连线建模方面的实验。

第二章　编译原理

2.1 词法

词法，是语言最基本的组成部分。程序语言的文本，都是由一个个符号所构成。例如，程序的关键字，变量的标识符，以及算术运算符等特殊符号。这些符号总是以一定的模式组合在一起，而词法扫描的任务，就是用不同的模板来匹配输入的字符。如果有字符和某个模板匹配，就把它从输入文本中挑选出来，并分门别类加以整理。

因为词法扫描的主要过程是基于模式匹配的。我们常常用一些表达模式结构的理论，如正则表达式、有限自动机等等来表达一种语言的词法。正则表达式的组成符号都是取自字符表，但是，正则表达式本身提供了多种特定的符号，这些符号能够提供一些扩展的功能。比如竖直线，表示在两个符号中选择一个；星号，表示前一个符号的多次重复。这种通过基本符号的不同组合，正则表达式可以表示任何程序语言的文本。

虽然手工书写一个正则表达式是比较复杂的，但通常只需要写出符合文法的正则表达式，而由自动词法生成工具来生成词法解析器。

2.2 文法

任何编程语言，归根结底必须满足一定的语法。而语法，则可以定义为包含如下元素的集合：

1）终结符

2）非终结符

3）起始符，有时也看作一个非终结符，是所与语法推导的开始

4）产生式，作为语法的规则，规定了所有终结符和非终结符的可能的排列顺序。

我们常用一个长箭头表示一个产生式，即表示产生式左部可以产生右部，或称产生式右部可以归纳为左部。比如所有的整数可以表示为：

S→[-]E

E→[123456789][123456789]*

其中 S 是起始符，E 是非终结符，而-和 1234567890 这几个数字构成了所有了终结符。

根据 Chomsky 的语法理论，我们可以把所有的产生式归纳为四种类型，

0）短语语法，这种语法是没有限制的

1）上下文相关语法，即产生式可以形如 xSy→xAy，即一个非终结符的分解和它的上下文有关。

2）上下文无关语法，产生式形如 S→A，即非终结符的分解和它的上下文没有关系。

3）正则语法，即所有 S→aA 形式的语法。

迄今为止，我们所能完整处理的语法仅限于正则语法和上下文无关语法。幸

运的是，所有的编程语言都属于这两类。因此，对于一般的行为级描述语言，我们总可以作出正确的解析和处理。

通常情况下，我们使用 BNF(Backus-Naur Form)来表示上下文无关文法。比如整数的算术表达式可以表示为下列语法。

$$exp \longrightarrow exp\ op\ exp \mid (\ exp\ ) \mid \text{number}$$
$$op \longrightarrow + \mid - \mid *$$

在规定了一个语法以后，我们需要建立这个语法的解析器。通常的语法解析有“ 自上而下 ”和“ 自下而上 ”两张方式。

2.2.1 自上而下的解析方法

首先来介绍自上而下的解析方法。我们有语法：

$S \to (S)S \mid \varepsilon$

关于采用自上而下的解析过程，我们以下表举例表示。第一列显示了解析栈的内容，第二列则是输入的内容，结尾处的美元符表示输入的终止符，第三列是我们需要采取的动作。

Parsing stack	Input	Action
$S	( ) $	$S \longrightarrow (S)S$
$S) S (	( ) $	match
$S) S	) $	$S \longrightarrow \varepsilon$
$S)	) $	match
$S	$	$S \longrightarrow \varepsilon$
$	$	accept

图 3 自上而下的解析

自上而下的解析器首先将起始符放入解析栈中，终止时解析栈和输入都应当为空。自上而下的解析器按照 BNF 语法替换栈顶的非终结符，直到解析完成。在这期间，它只执行两种操作：

1）产生。按照语法规则 $A \to \alpha$ ，将栈顶的非终结符 A 替换成 α。

2）匹配。匹配栈顶的符号和下一个输入的符号。

在上表中，对于“ 产生 ”动作，我们用被使用的产生式标明。对于“ 匹配 ”动作，我们标注为 match，同时需要把弹出栈顶匹配的符号，并从输入中去除同样的符号。需要注意的是，执行产生动作时，压栈的产生式是逆序的。

显然，在自上而下的解析中，当栈顶是一个非终结符时，我们需要决定使用哪个产生式来替换它。常用的策略是构建一张表格，由栈顶的非终结符和输入的下一个终结符来决定“ 产生 ”动作。下图即是该文法的解析表。

M[N,T]	(	)	$
S	$S \longrightarrow (S) S$	$S \longrightarrow \varepsilon$	$S \longrightarrow \varepsilon$

这个表格可以使用下列规则生成：

1） 若产生式 $A \to \alpha$ 可能是一种选择，且存在一个产生式 $\alpha \Rightarrow *a\beta$，其中 a 是一个终结符，则将 $A \to \alpha$ 加入表项 M[A,a]。

2） 若产生式 $A \to \alpha$ 可能是一种选择，且存在产生式 $\alpha \Rightarrow *\varepsilon$ 和 $S\$ \Rightarrow *\beta A a \gamma$，其中 S 是起始符，a 是一个终结符，则将 $A \to \alpha$ 加入表项 M[A,a]。

如果一个文法按照如上的规则生成的表格，每个表项中只有一个产生式，则称这个文法为 LL(1)文法。

2.2.2 自下而上的解析方法

和“从上往下”的解析方法相对的，我们有“从下往上”的方法。“从下往上”的解析方法也需要一个栈来存放当前的状态。对于文法 $S \to (S)S \mid \varepsilon$ 和输入 $(S)S$，我们有如下的解析过程。

Stack	Input	Action
$	( ) $	shift
$ (	) $	reduce $S \longrightarrow \epsilon$
$ (S	) $	shift
$ (S)	$	reduce $S \longrightarrow \epsilon$
$ (S) S	$	reduce $S \longrightarrow (S) S$
$S	$	reduce $S' \longrightarrow S$
$S’	$	accept

图 4 自下而上的解析

在“从下往上”的解析过程中，有如下两种操作：

1）移进。移进操作是将下一个输入的终结符压入栈中。

2）归约。归约操作是把栈顶的一组符号根据产生式还原成一个非终结符。

因此，我们需要根据当前的状态，决定下一步是进行“移进”还是“归约”操作。通常这借助于有限自动机(Finite Automata)来作出决定。我们首先将每个产生式分解为几个渐变的状态，并找出所有状态间可能的状态切换的路线，综合成为一个有限自动机。但是，由文法直接得到的是非确定性有限自动机(Nondeterministic Finite Automata)，如下图：

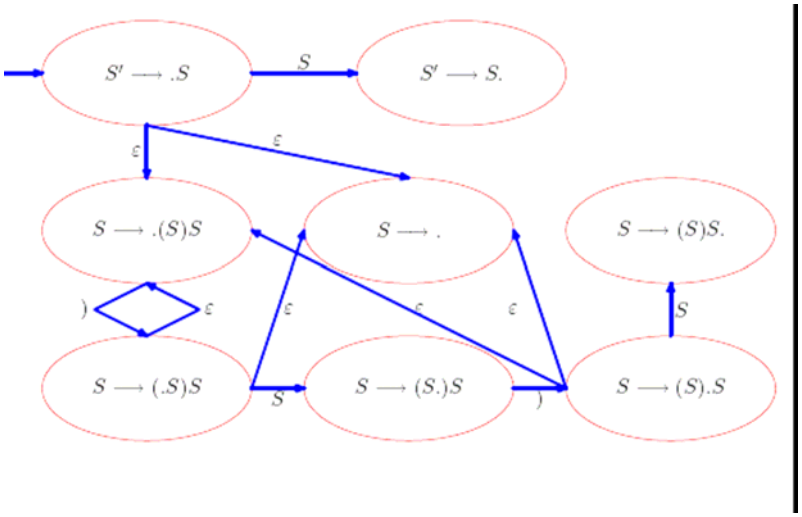


图 5 非确定性有限状态机

实际上，在将互相等价的状态归并以后，所有的非确定性有限自动机总可以转换为有限自动机(Deterministic Finite Automata)，如下图：

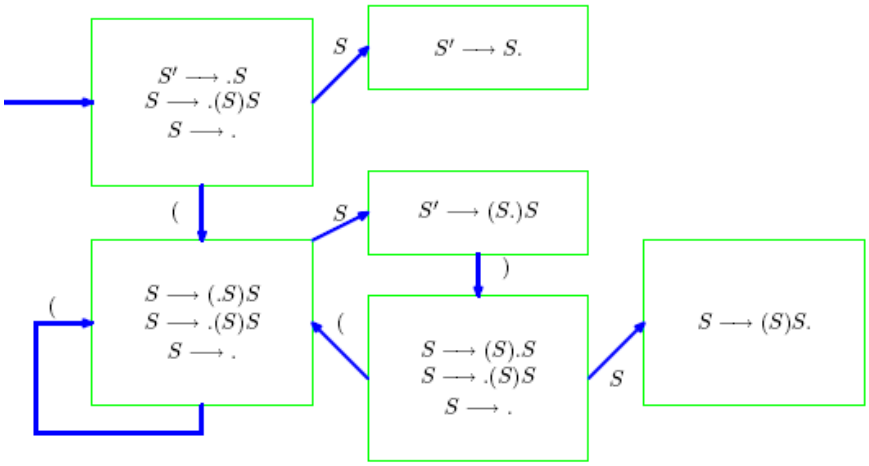


图 6 确定性有限状态机

得到了确定性有限自动机以后，解析器在每个状态下所采取的动作就已经确定了。

2.3Lex 和 Yacc

在模型编译器的实现过程中，Lex 和 Yacc[12]是两个基本的工具，分别对应词法分析和语法分析两个阶段。对于一些简单的语言，如电路仿真器的输入网表，能够直接使用 C 语言代码写出相应的词法和语法分析器。但是，对于 Verilog-AMS 或类似的高级语言，由于其丰富的表达形式、复杂的语法组成，纯粹的手工编写往往力不从心。Lex 和 Yacc 就是生成词法语法分析器的有力工具。

2.3.1Lex

Lex 是一种生成扫描器的工具。扫描器是一种识别文本中词汇模式的工具。Lex 使用正则表达式匹配语言中的关键字或其他词汇，并返回一个相应的标记。如下是 Lex 的工作流程：

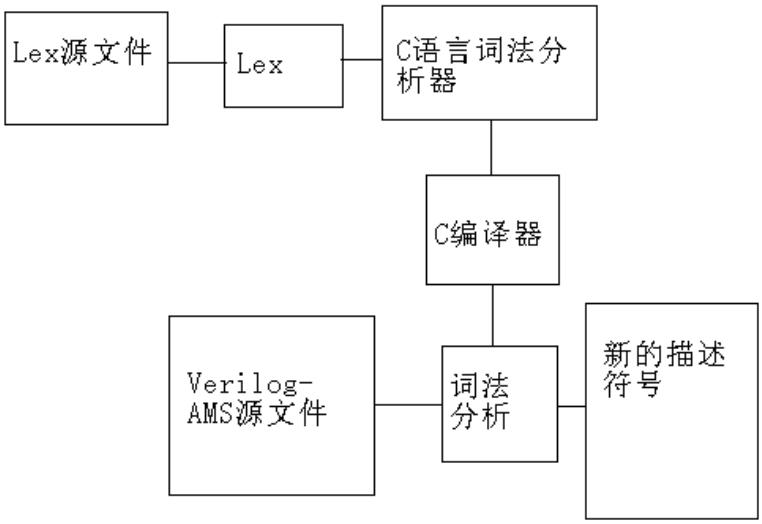


图 7Lex 工作流程

Lex 的源文件一般由三部分组成：

{定义}

%%

{规则}

%%

{自定义函数}

其中，定义部分的格式为

符号　　正则表达式

这表示在词法解析过程中，遇到匹配的表达式，则返回对应的符号。

规则部分的格式为

符号或正则表达式　执行动作

这表示若遇到已被识别的符号或比配对应的正则表达式时，就执行给出的动作。在纯粹由 Lex 生成的解析器中，可以直接在动作中解析源文件。而当与 Yacc 配合使用时，一般会返回一个 Yacc 能接受的符号。Lex 的识别能力非常强，通常的编程语言都在 Lex 的识别能力中。但在识别过程以后，Lex 只能做一些有限的动作，这使得它不能处理复杂的语法。这是，通常会把语法分析的部分独立交给 Yacc 完成。

Yacc(Yet Another Compiler Compiler)接收一个特定的文件作为输入，这个文件一般以(.y)为结尾，并且产生一个 C 语言文件作为输出。这个文件一般的文件名是 y.tab.c。

2.3.2Yacc

Yacc(Yet Another Compiler Compiler)接受 Lex 产生的标记，并作出相应的语

法动作。在 Yacc 中，程序员可以定义他需要的语法规则。Yacc 可以处理符合 LALR 文法的语言，这使得它能够胜任所有编程语言的解析工作。通常情况下，程序员只需要了解目标语言的语法，并写出和这些语法对应的规则以及对应的动作。

Yacc 的源文件和 Lex 类似，它包括下列部分:

{说明}

%%

{语法规则}

%%

{自定义程序}

可见，Yacc 的源文件由三部分组成，定义段、规则段和附加的程序段，它们之间用两个百分号来分隔。

一般来说，说明部分包含一些符号、数据类型和部分的语法。它也会包含一些直接编译到输出文件中的 C 语言代码，这些代码都会被附加到输出文件的开头，一般包含在语法解析过程中用到的常量或变量，有时还需要声明一些操作符的优先级、结合性等。然而，说明段有时是可以省略的。

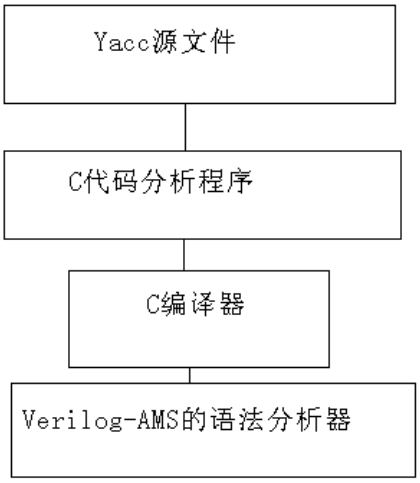


图 8Yacc 工作流程

规则段包含语法规则和对应的 C 语言执行动作。这些规则必须被改写成符合 LALR 文法的形式，并用一种类似于 BNF 的范式表示。这种范式和 BNF 的区别在于：

1）使用竖直线来组合可供选择的产生式。

2）在 Yacc 中，使用冒号代替箭头表示产生式中的推出符号。

3）在每个语法规则的结尾，使用分号来表示结束。

第三个部分，补充程序段，包含了一些必要的 C 语言程序。这些代码会被原封不动地附加到输出文件的结尾

Yacc 的语义分析依赖于“ 规则 ”的定义，这些“ 规则 ”，即是文法的产生式。在实现这些规则时，Yacc 使用的是 LR 方法，即在分析过程中，分析程序内部有两个堆栈，分析栈和状态栈。分析栈里存放了已经被解析的终结符和非终结符，这些符号可能是从 Lex 返回的，也可能是在执行 Yacc 的规则时产生的。Yacc 通过一个有限状态机来操作这两个栈。这个状态机包括下述动作：

1）移进

移进动作是指在当前状态下文本中下一个终结符是可以接受的，因此在将该终结符移入状态栈中，并移入新的状态

2）规约

栈顶的状态和某个产生式匹配，因此弹出状态栈中对应符号，将返回的非终结符重新压栈。在规约过程中，需要同步完成相应的动作，动作的对象放置在分析栈中，返回的对象也需要重新放回分析栈中。Yacc 为动作定义了一些伪变量，如$$指向动作返回的对象，$1,$2 等指向操作的对象，它们的顺序对应产生式右边的非终结符。

3）接受

遇到文本中的结束符，且栈中只剩下起始符，表明一次语法分析成功结束。

4）出错

输入的符号不能和任意一个产生式匹配，Yacc 会调用 YYError 函数尝试恢复错误，若无法恢复，则推出分析过程。

在产生式匹配的过程中，有时会产生冲突。Yacc 对这些冲突有默认的处理方法：

1）无优先级和结合性作用的情况下，出现移进/规约冲突时，执行移进动作。

2）无优先级和结合性作用的情况下，出现规约/规约冲突时，按照规则在 Yacc 源程序中出现的顺序，用先出现的规则进行规约。

3）如果在移进/规约冲突中，语法规则和输入单词均有优先级和结合性，那么 Yacc 按照优先级高的执行。如果优先级相同，则按照结合性处理，若是左结合则进行规约，若是有结合则进行移进。

如果 Yacc 的默认规则不能解决冲突，我们需要改写原先的语法规则来避免冲突，

2.3.3 抽象语法树

抽象语法树（Abstract Syntax Tree, AST），是一种内建的数据结构。抽象语法树作为语法分析的输出结果，能够完整表达经过语法分析以后输入代码所包含的语法结构和特定语义。和通常的解析树相比，抽象语法树本身具有简单明了、操作效率高等优点。例如一个表示两个数字相加的加法表达式可以写作：

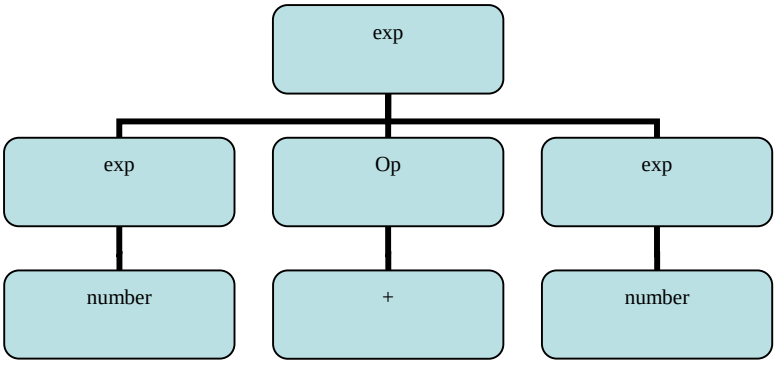


图 9 语法结构

而采用抽象语法树可以写作：

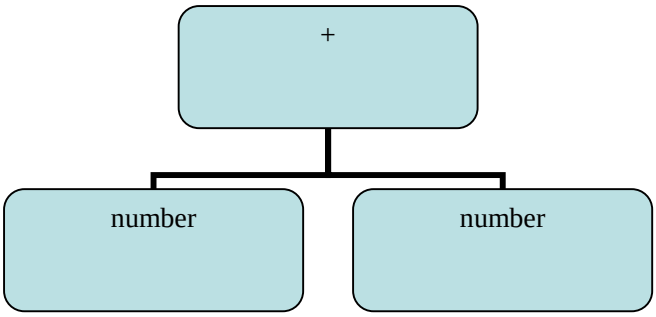


图 10 抽象语法树

显然，使用抽象语法树，省略了很多与最终结果无关的中间语法状态，使得后端的解析工作更加方便，同时使用操作符作为整个表达式的父节点，也使得结构更加简练。

和抽象语法树相比，一些现代化的中间语言（meta language），如 XML 等具有更好的编程接口和互操作性，但是也增加了开发难度。

第三章 自动化模型编译

3.1 使用行为级描述语言建模

使用模型编译器，研发人员可以快速实现一个使用低级语言描述的模型。但是，模型编译器仍然不可能直接从数学方程式中得到结论，一个高级的、行为级的模型仍然是必须的。通常情况下，我们会选择 Verilog-AMS[8] 或者 VHDL-AMS。这两种语言，是广泛使用的 Verilog 和 VHDL 的超集。它们具有一些良好的拓展特性，能方便的用来建立模拟和模数混合的模型。和 C 语言相比，使用这两种语言进行建模，无须复杂的编程技巧，也无关模型的一些特定细节。从 Verilog-AMS 或 VHDL-AMS 的模型出发，我们能够无歧义地转换成最终的 C 语言代码，保证模型的实现有效、唯一。

3.1.1Verilog-AMS 建模语言

在工业化设计和验证中，广泛使用 Verilog 作为标准语言。Verilog-AMS 是 Verilog 的一个超集。它保留了 Verilog 对数字信号的描述能力，增加了对模拟信号和模数混合信号的描述。更重要的是，Verilog-AMS 支持结构化、分层的建模。使用 Verilog-AMS 进行电子系统仿真设计的优点在于统一语言环境下可以对混合信号系统进行针体描述和仿真测试，以获得系统完整的测试结果以便于进行功能和性能分析，实现了模拟电路与数字电路描述和设计方法的统一。同时，Verilog-AMS 进一步扩大了 Verilog 的应用领域，除应用在数字系统设计外，在模拟电路、数模混合系统以及多学科系统设计中也可以实现代码复用，提高了工程设计的效率，缩短了开发周期。

在 Verilog-AMS 中，引入了系统的概念，系统被认为是一些相互连接的组件，它们互相产生激励和响应，协同工作。同时，这些组件也可以是一个子系统，这样，就体现了系统分层的概念。组件也可以是原子的，自身不再包含其它组件，这些组件被称为基本组件。

信号是在端口的节点间相互连接的网络。如果一个信号的量被定义在离散域上，被称为数字信号；如果信号的量被定义在连续域上，则被称为模拟信号；如果在离散域和连续域上都有定义，被称为混合信号。

和信号类似的，一个端口也可以被称为数字端口、模拟端口或混合端口。

如图所示，模块之间通过端口相互连接。

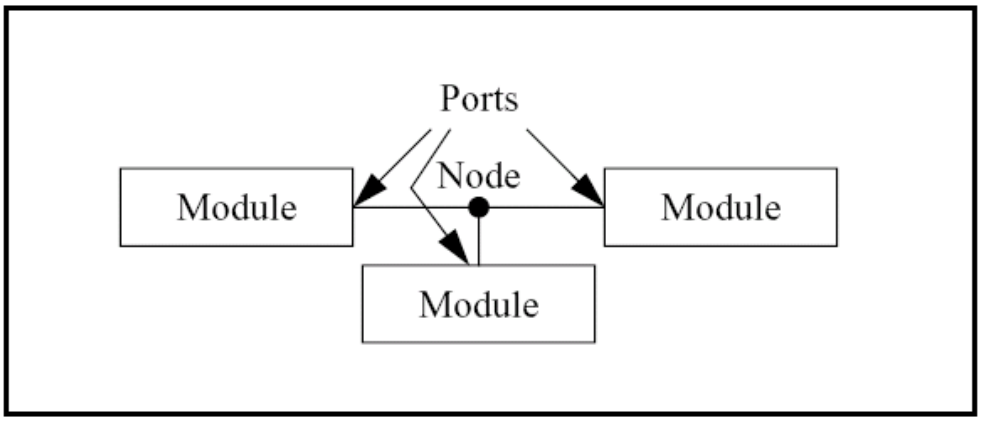


图 11 互连模块

如果一个信号是模拟的或混合的，我们需要为这个信号定义一个相关的节点，而纯数字信号没有相关的节点。这样，信号的模拟部分就可以用节点上定义的量来表示，这保证了模拟或混合信号相对于参考节点的势的唯一性。

当仿真一个系统时，需要得到对一个系统和它的组件的完整的描述。这些描述通常是结构性的。我们需要知道这些组件之间是如何连接的。此外，我们还要知道组件自身对激励的响应，这被称为行为级描述。所以行为级的描述，通常是对端口间信号的一些数学描述。

在 Verilog-AMS 中，一个很重要的概念是守恒系统。在一个守恒系统中，每个节点都被赋予两个值，节点的势（potential，即电子系统中的电压）和流（flow，即电子系统中的电流）。每个节点的势被相互直连的端口共享，因此，互连的端口拥有相同的势。而对于每个节点流入和流出的流，相加总为零。这样，每个节点都符合基尔霍夫势定律(Kirchhoff's Potential Law)和基尔霍夫流定律(Kirchhoff's Flow Law)。当一个组件和一个守恒的节点相连时，它既可能被这个节点的势所影响，也可能被从这个节点流入流出到组件的端口上的流影响。

单个节点上的势是相对于参考节点得到的。在电子系统中，参考节点被称为地，它的势总是为零，而任意一个节点都可以被作为参考地，这和我们熟知的电气定律也是相符的。

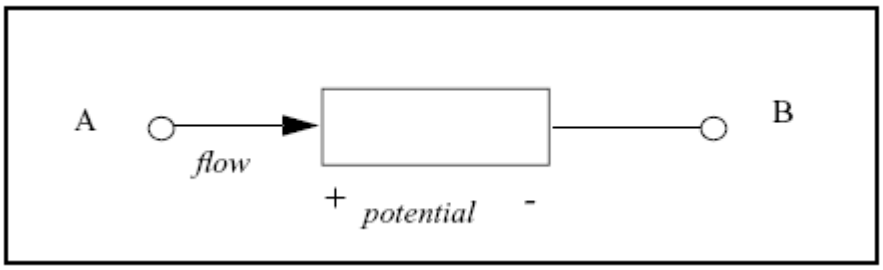


图 12 流和势

除了参考节点外，还需要给出参考方向。势的参考方向是用两端的正负符号表示的。给出了选定的参考方向以后，如果正号节点的势比负号节点的势大，则称势为正的，反之，则称势为负的。因此一个正的流总是从标记为正的节点流入，从标记为负的节点流出。

在表达连续的系统方程时，Verilog-AMS 使用两类不同的关系。第一类是描述每个内部行为的守恒关系。这些守恒关系通常在仿真器的内部实现，即被称为模型。第二类是网络之间的结构和互连关系。互连关系包含了组件之间相关连接

的信息，对应系统的拓扑结构。这些诗歌组件的自然属性无关的。

为了表达守恒系统，我们通常使用基尔霍夫定律来描述。虽然基尔霍夫定律可以描述相当广泛的系统，但在电子系统中，它们被特定称为基尔霍夫电压定律和基尔霍夫电流定律，而他们的推广形式，被称为基尔霍夫势定律和基尔霍夫流定律。

1）基尔霍夫流定律

每个节点在任意时刻流入流出的综合为零

2）基尔霍夫势定律

任意一个环路内势的总合为零。

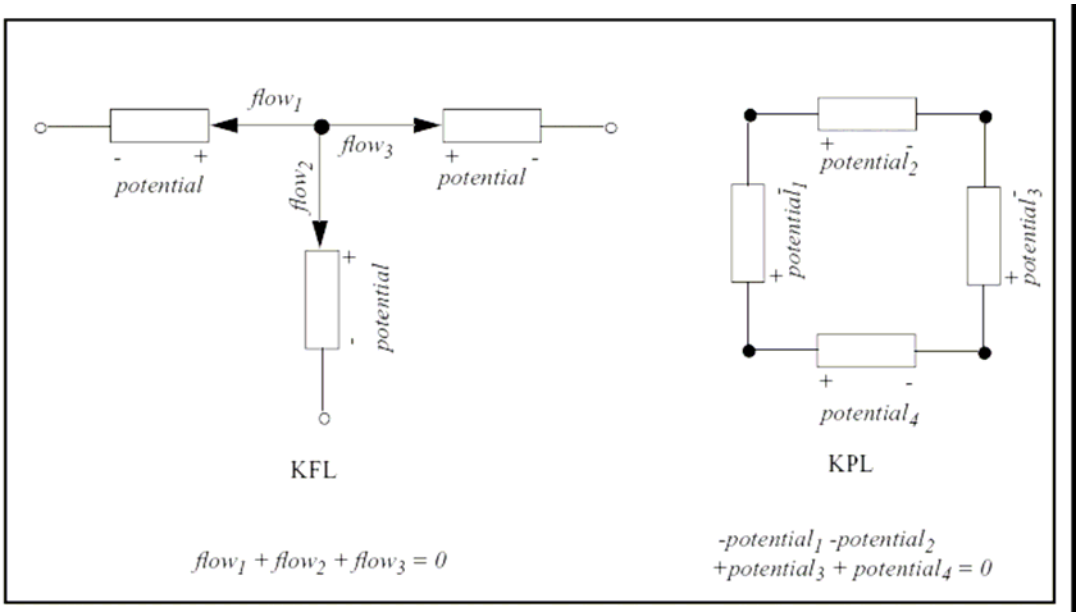


图 13 基尔霍夫定律

3.1.2 模型实例

在 Verilgo-AMS 中，定义一个模型，在结构上是很简单的。使用 module 标签，可以声明一个模型。模型可以拥有多个端口，每个端口可以独立地定义相关的电气特性。在模型内部，可以按照 Verilog-AMS 的语法对这些特性进行运算。一个模型的描述通常会从它的端口属性开始，紧接着会是一些常数或参数，从编程语言上来看，它们都可以视作为程序的普通常量和变量。和一般语言不同的是，这些变量的初始化时机不尽相同，有的需要在声明模型时初始化，有的则可以到实例化模型时再赋值。模型内部可能混杂着数字信号和模拟信号的行为描述。我们最关心的模拟行为模块需要用 analog 关键字标出，且可以多个并列。在 analog 块内部，就是对信号行为关系的具体表达。从形式上来讲，这些运算的表达非常接近于其数学原形。事实上，Verilog-AMS 内建了很多独特的算子，如 ddt,idt 等微分、积分算子，能方便地描述模型的数学公式。如下是一个电阻的模型：

```
module resistor (a, b) ;

inout a, b ;
```

electrical a, b ; // access functions are V() and I()

parameter real R = 1.0 ;

analog

I(a,b) <+ V (a,b) / R;

endmodule

从例子中可以看到，Verilog-AMS 不同于 Verilog-D，可以用来描述模拟量，对数学方程也有充分的表达能力。回忆一下在 C 语言中，对一个方程（组）的求解往往通过叠代得到，如果方程（组）非常复杂，程序员需要很高的编程技巧，并且足够仔细，以保证代码的质量。然而使用 Verilog-AMS，模型的描述非常接近其最初的数学形式，避免了开发者的额外负担，也便于验证和修改。

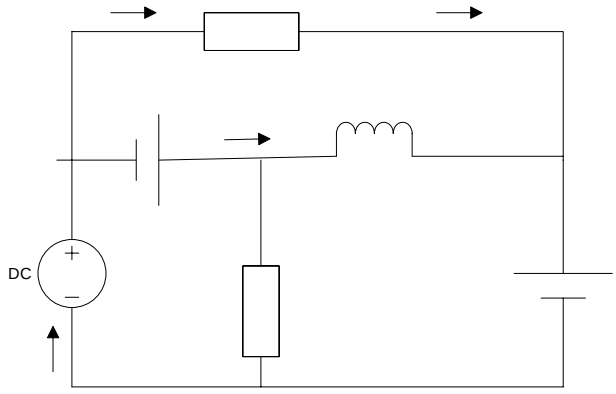
3.2 电路仿真原理

在本章中，我们试从常用的电路仿真原理入手，并逐步归纳出在模型编译器中使用的自动编译算法。在一般的电路求解中，广为人知的是改进节点法[10][11]。

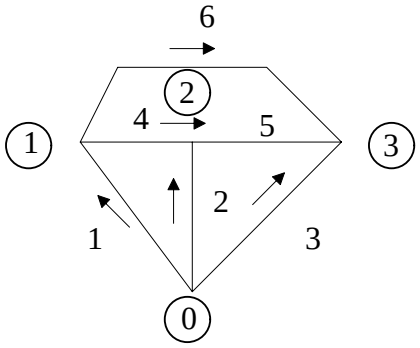
3.2.1 改进节点法和“ 邮票法 ”

3.2.1.1 关联矩阵和割集矩阵

在自动化的电路分析中，借鉴的是图论中的分析方法。因此，我们首先将元件抽象为一条支路，并定义各个节点上的电压和支路上的电流。例如下述电路



我们可以简化为



并根据基尔霍夫电流定律，可以写出各个分支电流间的关系

$$\begin{cases} -i_1 + i_4 + i_6 = 0 \\ -i_2 + i_4 + i_5 = 0 \\ -i_3 - i_5 - i_6 = 0 \end{cases}$$

或写成矩阵形式为：

$$A \cdot i = 0$$

其中 i 称为支路电流列向量，A 称为为节点与支路的关联矩阵。

设 n 为独立节点数，b 为支路数，A 为 0-1 矩阵，其中有 n 行 b 列，A 中各元素的取值按如下规则确定

$$a_{kj} = \begin{cases} 1\text{当支路j与节点k关联且方向离开节点k} \\ -1\text{当支路j与节点k关联且方向指向节点k} \\ 0\text{当支路j与节点k无关联} \end{cases}$$

和电流类似的，分支电压与节点电压也可以用关联矩阵联系起来。

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ 1 & -1 & 0 \\ 0 & 1 & -1 \\ 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} u_{n1} \\ u_{n2} \\ u_{n3} \end{bmatrix}$$

或写成矩阵形式为

$$A^T \cdot u_n = u$$

实际上，这也就是基尔霍夫电流定律和基尔霍夫电压定律的矩阵形式。

如果我们从电路图中抽取一颗生成树，然后对树枝和连枝进行编号。我们选取单树枝割集，称为基本割集。定义基本割集矩阵

$$Q_f = \{q_{kj}\}$$

其中各元素按如下规则确定

$$q_{kj}=\begin{cases}1\text{支路j与割集k关联，且方向相同}\\-1\text{支路j与割集k关联，且方向相反}\\0\text{支路j与割集k无关联}\end{cases}$$

则支路电压与树枝电压的关系可以表示为

$$u=Q_f^T u_t$$

其中u_t为树枝电压，u为支路电压。

同理，也可以列出割集上的基尔霍夫电流定律

$$Q_f i=0$$

3.2.1.2 改进的节点法

在节点法中，第 k 个无源元件支路的电压、电流关系可写成：$Y_k U_k=I_k$。对图，可以表示为

或简写为$Y_b U_b=I_b$

式中Y_b为对角矩阵，称为无源支路导纳矩阵，U_b和I_b分别为无源元件的支路电压向量和支路电流向量。

这样，我们可以把基尔霍夫电流定律表示为

$$A_a I=\begin{bmatrix}A & A_i\end{bmatrix}\begin{bmatrix}I_b\\I_s\end{bmatrix}=0$$

其中，A 对应无源支路，Ai 对应电流源支路，Is 对电流源支路的电流向量。即

$$AI_b=-A_i I_s$$

代入$Y_b U_b=I_b$得到

$$AY_b U_b=-A_i I_s$$

又由基尔霍夫电压定律有

$$U_b=A^T U_n$$

因此

$$AY_b A^T U_n=-A_i I_s$$

或简写为

$$YU_n=J_n$$

式中，Y 称为节点导纳矩阵，Jn 成为等效的节点电流源向量。这就是所谓的“ 节点法 ”。

但是，节点法不能用来处理 VCVS，CCVS 等元件，所以提出了“ 改进的节点法 ”。改进节点法对导纳支路以节点电压为变量，而对其他支路以支路电流为变量，将支路分为类，一组为导纳形式，另一组为阻抗形式，则基尔霍夫电流方程可写为

$$\begin{bmatrix} A_1 & A_2 & A_3 \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \\ I_s \end{bmatrix} = 0$$

式中，I1 对应于用导纳表示的支路电流向量，I2 对应于用阻抗表示的支路电流向量，Is 为电流源向量。

因此

$$A_1 I_1 + A_2 I_2 = -A_3 I_s$$

代入相应的导纳支路关系，

$Y_1 U_1 = I_1$，或 $Y_1 A_1^T U_n = I_1$，代入得

$$A_1 Y_1 A_1^T U_n + A_2 I_2 = -A_3 I_s$$

更一般的支路关系可表示为

$$Y_2 A_2^T U_n + Z_2 I_2 = W_2$$

其中，W2 中可以包括独立源、电感或电容的初始条件。

将上式写成矩阵形式，就得到了改进的节点方程

$$\begin{bmatrix} A_1 Y_1 A_1^T & A_2 \\ Y_2 A_2^T & Z_2 \end{bmatrix} \begin{bmatrix} U_n \\ I_2 \end{bmatrix} = \begin{bmatrix} -A_3 I_s \\ W_2 \end{bmatrix}$$

或简写成

$$\begin{bmatrix} Y_{n1} & A_2 \\ Y_2 A_2^T & Z_2 \end{bmatrix} \begin{bmatrix} U_n \\ I_2 \end{bmatrix} = \begin{bmatrix} J_n \\ W_2 \end{bmatrix}$$

式中，Yn1 为节点导纳矩阵，Jn 为等效的节点电流源向量。

3.2.2.3 “ 邮票法 ”

在实用的电路分析程序中，并不需要根据分支方程来求出节点导纳矩阵，而是从元件的性质和节点位置直接填充导纳矩阵的。我们试给出一些基本元件的矩阵形式：

1）电流源

$$\begin{matrix} i \\ j \end{matrix} \begin{bmatrix} -I_s \\ I_s \end{bmatrix}$$

2）导纳

$$\begin{array}{c} \\ i \\ j \end{array}\begin{array}{c} \begin{array}{cc} u_i & u_j \end{array} \\ \begin{bmatrix} y & -y \\ -y & y \end{bmatrix} \end{array}$$

3）VCCS

$$\begin{array}{c} \\ m \\ n \end{array}\begin{array}{c} \begin{array}{cc} u_i & u_j \end{array} \\ \begin{bmatrix} g & -g \\ -g & g \end{bmatrix} \end{array}$$

4）CCVS

$$\begin{array}{c} \\ i \\ j \\ m \\ n \\ \\ \\ \end{array}\begin{array}{c} \begin{array}{cccccc} u_i & u_j & u_m & u_n & I_1 & I_2 \end{array} \\ \begin{bmatrix} & & & & 1 & \\ & & & & -1 & \\ & & & & & 1 \\ & & & & & -1 \\ 1 & -1 & & & & \\ & & 1 & -1 & -r & \end{bmatrix} \end{array}$$

3.3 模型自动编译原理

在很多模型中，电路通常是非线性的，要求解非线性方程组，通常使用 Newton-Raphson 方法。设有分支方程，

$$I_i = f_i(x_1, x_2, \ldots, x_n)$$

其中 x 为所有未知的分支电压或分支电流。其中，f 为非线性函数。作一阶泰勒展开，我们有

$$I_i^{j+1} = f_i(x_1^j, x_2^j, \ldots, x_n^j) + \frac{\partial f_i}{\partial x_1^j}(x_1^{j+1} - x_1^j) + \ldots + \frac{\partial f_i}{\partial x_n^j}(x_n^{j+1} - x_n^j)$$

令

$\hat{F} = [f_1, f_2, \ldots, f_n]^T$ $\hat{x} = [x_1, x_2, \ldots, x_n]^T$，$J = \dfrac{\partial \hat{F}}{\partial \hat{x}}$，且设 $I_i^{j+1} = 0$，则

$$J \cdot \delta \hat{x} = J \cdot (\hat{x}^{j+1} - \hat{x}^j) = -\hat{F}(x_1^j, x_2^j, \ldots, x_n^j)$$

其中，矩阵 J 被称为 Jacobi 矩阵，而 - F 被称为 RHS（Right Hand Side）。显然，这里的 Jacobi 矩阵和 RHS，就是上文所称的“ 邮票 ”。因此，在自动编译技术中，构建电路方程的原理和传统的电路仿真类似，但具体的过程却是恰恰相反。在传统的仿真器中，我们得到的是模型的名称，相对应的，对于所有可能的输入模型，我们都已经有了它们的数学表达，以及从这些数学表达中固定抽象出来的“ 邮票 ”，因此，剩下的工作只是按部就班的“ 黏贴 ”。而在自动编译技术中，由于输入的复杂性，难以归纳出一一对应的“ 邮票 ”，因此，我们需要从最原始的

电路方程出发，通过一些确定的步骤，来自动生成“ 邮票 ”。简而言之，自动编译技术的核心，即是如何生成模型的“ 邮票 ”。和改进节点法的思想类似，我们将顺序推导下列三种情况。

3.3.1 不含独立电压源或流控电压源的分支

首先我们按照基尔霍夫电流定律写出分支电路方程

$$Curr = f(x_1, x_2 \ldots x_n)$$

其中，x 为待求解的节点电压等。按照 Newton-Raphson 方法，对方程左右两边做一阶 Taylar 展开，即

$$Curr^{n+1} = f(x_1^n, x_2^n \ldots x_m^n) + \frac{\partial f}{\partial x_1}(x_1^{n+1} - x_1^n) + \frac{\partial f}{\partial x_2}(x_2^{n+1} - x_2^n) + \ldots + \frac{\partial f}{\partial x_m}(x_m^{n+1} - x_m^n)$$

其中

$$f(x_1^n, x_2^n \ldots x_m^n) = Curr^n$$

因此，我们有

$$Curr^{n+1} \approx \frac{\partial f}{\partial x_1}x^{n+1} + \frac{\partial f}{\partial x_2}x^{n+1} + \ldots\frac{\partial f}{\partial x_m}x^{n+1} + (Curr^n - \frac{\partial f}{\partial x_1}x^n - \frac{\partial f}{\partial x_2}x^n - \ldots\frac{\partial f}{\partial x_m}x^{n+1})$$

从这个形式上，我们可以直观地得到结果

	x_1	x_2	$\cdots$	x_m	RHS
STAMP	$\frac{\partial f}{\partial x_1}$	$\frac{\partial f}{\partial x_2}$	$\cdots$	$\frac{\partial f}{\partial x_m}$	$Curr^n - \frac{\partial f}{\partial x_1}x^n - \frac{\partial f}{\partial x_2}x^n - \ldots\frac{\partial f}{\partial x_m}x^{n+1}$

3.3.2 需要输出电流值的支路

为了得到输入支路电流，需要将其引入，作为一个新的未知量，则

$$0 = f(x_1, x_2 \ldots x_n, Curr)$$

同上，我们对这个方程作一阶 Taylar 展开

$$0 \approx \frac{\partial f}{\partial Curr}Curr + \frac{\partial f}{\partial x_1}x^{n+1} + \frac{\partial f}{\partial x_2}x^{n+1} + \ldots\frac{\partial f}{\partial x_m}x^{n+1} + (f(x_1^n, x_2^n, \ldots x_m^n, Curr) - \frac{\partial f}{\partial Curr}Curr - \frac{\partial f}{\partial x_1}x^n - \frac{\partial f}{\partial x_2}x$$

从而得到下表

	x_1	x_2	$\cdots$	x_m	Curr	RHS
N+					+1	

N-					-1	
STAMP	$\frac{\partial f}{\partial x_1}$	$\frac{\partial f}{\partial x_1}$	…	$\frac{\partial f}{\partial x_m}$	$\frac{\partial f}{\partial Curr}$	$f(x_1^n, x_2^n, \ldots x_m^n, Curr) - \frac{\partial f}{\partial Curr} Curr$ $- \frac{\partial f}{\partial x_1} x^n - \frac{\partial f}{\partial x_2} x^n - \ldots \frac{\partial f}{\partial x_m} x^{n+1})$

其中，N+，N-分别是该分支的流入节点和流出节点

3.3.3 带有可控电压源的分支

我们需要分别引入该电压源的电压和电流作为未知变量，得到下述方程

$V = f(x_1, x_2, \ldots x_m, i)$

同样的，做一阶 Taylar 便展开后得到

	x_1	x_2	…	x_m	V	i	RHS
N+						+1	
N-						-1	
STAMP	$\frac{\partial f}{\partial x_1}$	$\frac{\partial f}{\partial x_1}$	…	$\frac{\partial f}{\partial x_m}$	$\frac{\partial f}{\partial V}$	$\frac{\partial f}{\partial i}$	$f(x_1^n, x_2^n, \ldots x_m^n, V, i) - \frac{\partial f}{\partial V} V - \frac{\partial f}{\partial i} i$ $- \frac{\partial f}{\partial x_1} x^n - \frac{\partial f}{\partial x_2} x^n - \ldots \frac{\partial f}{\partial x_m} x^{n+1})$

3.4 传输线模型和信号完整性

当一个导体必须被看作一系列的分布电容和电感时，它就被称为是一条传输线。一般来说，当电路尺寸接近信号中我们关心的最高频率的波长时，就应该将导线按照传输线来对待。在实践中，如金属导线、波导、同轴线以及 PCB 板上和芯片内部的走线等等，都可以被看作是传输线。由于许多的互连结构都是在完整的地平面上的走线构成，所以可以将它们准确地模拟为传输线，信号完整性的分析大多都建立在这个基础上。

信号完整性是指信号在传输线上的质量，即信号在电路中能以正确的时序和电压做出响应的能力。当电路中的信号能以要求的时序、持续时间和电压幅度到达接收端时，就表明该电路具有很好的信号完整性。反之，我们就说信号完整性较差。

3.4.1 传输线方程

传输线上的电压、电流不仅是时间的函数，也是空间位置的函数，这是由于线上的分布参数起作用的。这些分布参数是由传输线的结构形状、几何尺寸、导体材料及介质特性决定的。当传输线的分布参数沿轴向不发生变化时，我们称这样的传输线为均匀传输线，用 R1、L1、G1、C1 表示其单位长度上的分布电阻、

分布电感、分布电导和分布电容。

对于一段有限长的传输线，可以看成由许许多多长为△z 的小线元所组成，当△z 足够小时，可以把它们看作分立的电路元件来处理。如图，可以把传输线当作一个电路网络来推导。

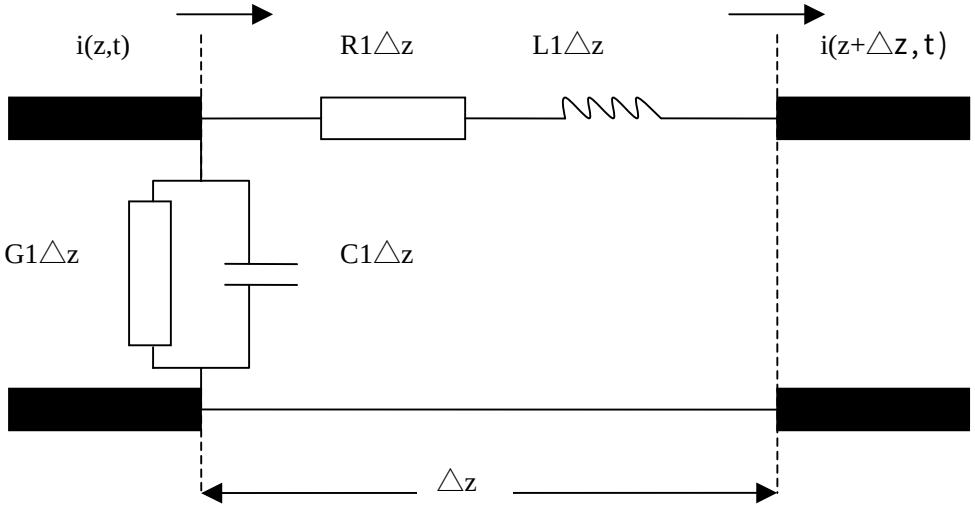


图 14 传输线分布参数模型

考虑小线元△z 处的等效电路。设两端的电压、电流分别为 v(z,t)、i(z,t)、和 v(z+△z,t)、i(z+△z,t)。由基尔霍夫定律列出方程

$$v(z+\Delta z,t)-v(z,t)=-\Delta v(z,t)=R_1\Delta z i(z,t)+L_1\Delta z\frac{\partial i(z,t)}{\partial t}$$

$$i(z+\Delta z,t)-i(z,t)=-\Delta i(z,t)=G_1\Delta z v(z,t)+C_1\Delta z\frac{\partial v(z,t)}{\partial t}$$

将上式两边除以△z，并令△z 趋近于零，则有

$$-\frac{\partial v(z,t)}{\partial z}=R_1 i(z,t)+L_1\frac{\partial i(z,t)}{\partial t}$$

$$-\frac{\partial i(z,t)}{\partial z}=G_1 v(z,t)+C_1\frac{\partial v(z,t)}{\partial t}$$

此式即为均匀传输线方程，又称电报方程。当传输线始端接简谐震荡源时

$$v(z,t)=\mathrm{Re}[V(z)e^{j\omega t}]$$

$$i(z,t)=\mathrm{Re}[I(z)e^{j\omega t}]$$

代入可得

$$-\frac{dV(z)}{dz}=(R_1+j\omega L_1)I(z)=Z_1 I(z)$$

$$-\frac{dI(z)}{dz}=(G_1+j\omega C_1)V(z)=Y_1 V(z)$$

其中，$Z_1=R_1+j\omega L_1$，$Y_1=G_1+j\omega C_1$分别为传输线单位长度上的串联阻抗和并联阻抗。

3.4.2 传输线方程的解和特性参数

在传输线方程两端对 z 微分，得到

$$\frac{d^2V(z)}{dz^2}-Z_1Y_1V(z)=0$$

$$\frac{d^2I(z)}{dz^2}-Z_1Y_1I(z)=0$$

令 $\gamma^2=Z_1Y_1$

方程的通解可以写成

$$V(z)=A_1e^{-e\gamma z}+A_2e^{\gamma z}$$

$$I(z)=\frac{1}{Z_0}(e^{-e\gamma z}+A_2e^{\gamma z})$$

式中

$$Z_0=\frac{Z_1}{\gamma}=\sqrt{\frac{Z_1}{Y_1}}$$

这表明，传输线上的电压、电流是两个波的合成，一个是由信号源向负载端传输的“ 入射波 ”，另一个是由负载端向信号源传输的“ 反射波 ”。若分别用“ + ”、“ - ”标示，则

$$V(z)=V_+(z)+V_-(z)$$

$$I(z)=I_+(z)+I_-(z)=\frac{1}{Z_c}[V_+(z)-V_-(z)]$$

传输线的特征阻抗 Z_0 定义为线上任意点的电压和电流的比率，即 $Z_0=\frac{V_+}{I_+}=\frac{V_1}{I_-}=\frac{Z_1}{\gamma}=\sqrt{\frac{Z_1}{Y_1}}$。传播速度（相速）v 是指电信号在传输线上的传播速度，即 $v_p=\frac{\omega}{\beta}=\frac{1}{\sqrt{L_1C_1}}$。单位长度的传播延迟 t_{pd} 是传播速度的倒数。

3.4.3 传输线上的信号完整性问题

在传输线上，影响信号完整性的最显著的问题就是反射和串扰。

3.4.3.1 反射

在高速电路中想要把信号能量从源端全部传递给负载，必须使传输线特征阻抗与信号的源端阻抗和负载阻抗匹配，否则信号会发生反射，导致信号波形发生畸变。

信号反射会产生过重合下冲现象，过充是指信号跳变的第一个峰值，它是在电源电平之上或参考地电平之下的额外电压效应；下冲是指信号跳变得下一个谷值。如果信号在源端和负载之间来回多次反射，就会发生震荡现象，增加了信号稳定所需要的时间。

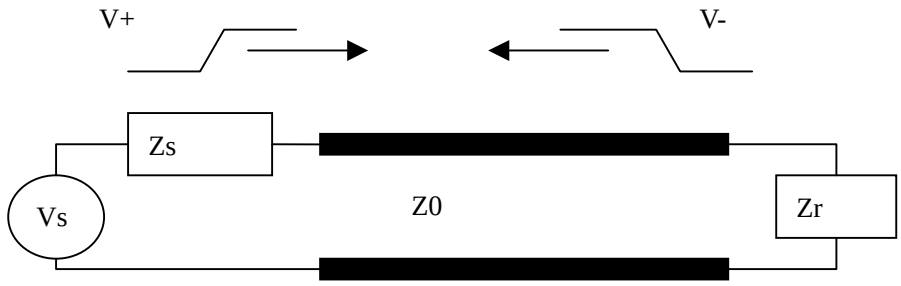


图 15 反射现象

从源端发射一个信号进入传输线时，从源端看到的初始电压 Vi 取决于驱动源电压，内阻和传输线特征阻抗，为 VsZ0/(Z0-Zs)。这个初始电压将沿着传输线传播到负载端。如果传输线末端负载阻抗和线的特征阻抗相等，则 Vs 到达负载端并稳定在此值直到信号源状态改变；否则，如果负载阻抗与传输线特征阻抗不相等，当入射波到达负载 Zr 时，信号的一部分 ρV_i 被反射回源端，并与入射波叠加在线上产生了一个总的幅值，反射回来的能量到达源端时，若源端阻抗不等于传输线阻抗，就将产生第二次反射。这样，信号就会在源端和负载端来回多次反射，直到最后达到直流稳态。其中，ρ 是反射系数，被定义为给定节点上的反射电压和入射电压的比值，它由传输线特征阻抗和负载阻抗之间的比例关系决定，即

$$\rho = \frac{V_-}{V_+} = \frac{Z_r - Z_0}{Z_r + Z_0}$$

3.4.3.2 串扰

串扰是指当信号在传输线上传播时，因电磁能量耦合对相邻的传输线产生的不期望的噪声干扰，它是由不同结构引起的电磁场 在同一区域里的相互作用而产生的。串扰会改变总线中单根传输线的性能，比如传输线特征阻抗和传输速度等，同时，串扰会将噪声感应耦合到其他的传输线上，从而对信号完整性产生较大的影响。

由于串绕都是由电磁耦合形成的，所以可分为由互容 Cm 引起的电场耦合和由互感 Lm 引起的磁场耦合。从串扰的位置来看，我们又把受绕线上靠近侵害线

源端的串扰称为近端串扰，远离侵害线源端的串扰成为远端串扰。因为互容引起的电流在两个端口处方向相反，而互感产生的电流总是与源电流相反，因此，我们有

$$\begin{cases} I_{near} = I(L_m) + I_{near}(C_m) \\ I_{far} = I_{far}(C_m) - I(L_m) \end{cases}$$

如图

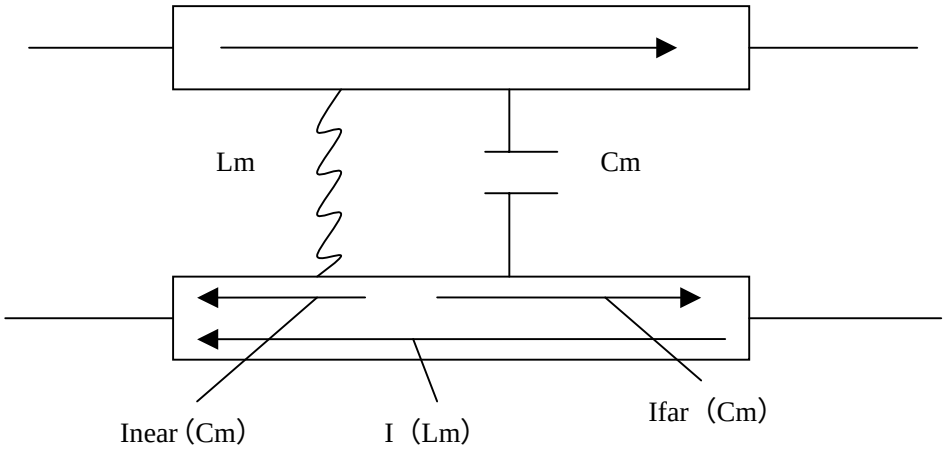


图 16 串扰现象

3.5 其他相关算法

在 Newton-Raphson 的求解过程中，迭代的每一步都是求解一个线形方程。在复杂的电路中，这个方程的维数非常高。通常情况下，我们使用 LU 分解来求解。

首先我们把 Jacobi 分解为两个矩阵 L 和 U 的乘积，其中，L 为下三角矩阵，U 为上三角矩阵。则我们有

$$A \cdot x = (L \cdot U) \cdot x = L \cdot (U \cdot x) = b$$

从而我们可以分步求解该方程组，即首先有

$$L \cdot y = b$$

再有

$$U \cdot x = y$$

因为上述都是三角矩阵，因此，求解这两个方程是非常简单的，我们有：

$$y_1 = \frac{b_1}{a_{11}}$$

$$y_i = \frac{1}{a_{ii}}[b_i - \sum_{j=1}^{i-1} a_{ij} y_j], i = 2,3,\ldots,N$$

和

$$x_N = \frac{y_N}{\beta_{NN}}$$

$$x_i = \frac{1}{\beta_{ii}}[y_i - \sum_{j=i+1}^{N} \beta_{ij} x_j], i = N-1, N-2, \ldots, i$$

因此，问题的关键是把 Jacobi 矩阵写成 LU 分解的形式。
观察矩阵的代数形式：

$$\begin{bmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

观察得：

$$i < j : \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \ldots + \alpha_{ii}\beta_{ij} = a_{ij}$$
$$i = j : \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \ldots + \alpha_{ii}\beta_{jij} = a_{ij}$$
$$i > j : \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \ldots + \alpha_{ij}\beta_{ij} = a_{ij}$$

我们有 Crout 算法

1） 令 $\alpha_{ii} = 1, i = 1, \ldots, N$

2） 对于 j 从 1 到 N，重复下述两个步骤：
首先，对 i 从 1 到 j，我们有

$$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj}$$

其次，对 i 从 j+1 到 N，我们有

$$\alpha_{ij} = \frac{1}{\beta_{jj}}(a_{ij} - \sum_{k=1}^{j-1} \alpha_{ik}\beta_{kj})$$

重复上述步骤，直到得到所有 L 和 U 的分量。
从下图中，可以看出 Crout 算法的优点是只需占用额外的 N 个存储空间

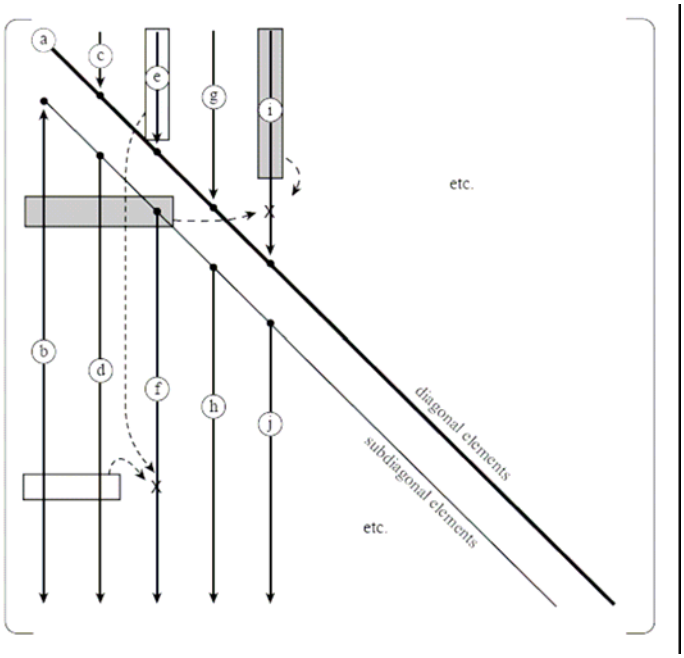

a
c
e
g
i
etc.
b
d
f
h
j
diagonal elements
subdiagonal elements
etc.

图 17Crout 算法

第四章　CAMC 的实现与应用

CAMC(***C***amc is ***A***nother ***M***odel ***C***ompiler)是一个新的模型编译器。CAMC 能够将 Verilog-AMS 描述的器件模型编译成一个动态链接库。这种动态链接库能被电路仿真器 zspice[13][14]加载，并由输入网表的参数实例化，配置在电路中进行仿真。CAMC 的使用流程如下图所示：

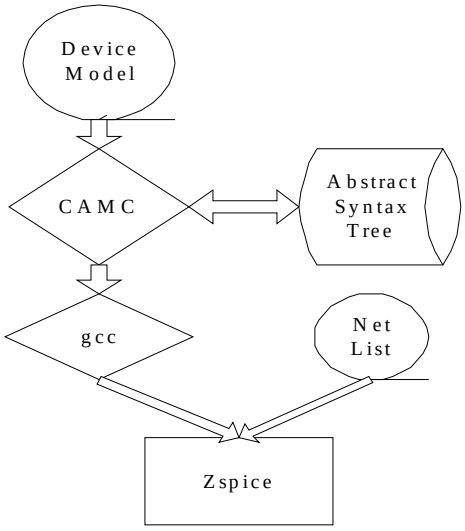


图 18CAMC 工作流程

图中，gcc 作为标准的 C 语言编译器，把 CAMC 的输出代码编译成可以被 zspice 加载的动态链接库。Zspice 在加载了这些动态链接库以后，就能仿真我们通过 CAMC 编译出来的模型。具体电路的实例化工作，需要额外的网表(Net List)文件指定。

在 CAMC 的整体结构中，抽象语法树起到了一个很重要的角色。抽象语法树保存了所有从输入模型中解析出来的必要的信息，使得接下来的代码生成工作不再依赖于前面的变化和改动。因此，以抽象语法树的生成为分界线，我们可以把 CAMC 的实现分为前端和后端两个部分。

4.1 前端分析

CAMC 的前端被称为 VP(Verilog Parser)。VP 部分是对 Verilog-AMS 语法进行解析，并生成和后端无关的抽象语法树。这部分的工作使用 Lex 和 Yacc 实现。

4.1.1 符号

在前文中，我们已经对 Lex 的基本使用有了大致的了解。虽然 Lex 也有一些语法的解析能力，但在 CAMC 的实现中，我们可以完全使用 Yacc 来进行 Verilog-AMS 的语法解析。因此，Lex 的作用仅仅是对词法的识别，也就是从 Verilog-AMS 源代码中提取出具有一定模式特征的**符号**。根据 Verilog-AMS 的词

法，我们所定义的符号有：

1）数字。

包括整数，浮点数等。考虑到整数的不同进制，我们要定义各种进制的数字模式：

二进制　binteger b[1-9][1-9]*

八进制　ointeger 0[0-9]*

十六进制　　hinteger 0[xX][0-9A-Fa-f]+

十进制　dinteger [1-9][1-9]*

而浮点数得模式为

浮点数　float　　[0-9]*\.[0-9]+([eE][+-]?[0-9]+)?|[0-9]+\.?([eE][+-]?[0-9]+)?

需要注意的是，我们并没有定义负数。因此负号既可能是表明数字的正负，也可能作为一个减号运算法来使用。为了避免解析的歧义，我们统一把负号作为一个算术运算符，在稍后的语法解析中来对数字进行操作。

2）算术符号。

包括加减乘除等各种算术运算符：加、减、乘、除等等

3）结构性符号。

如模块的声明关键字 module 等等。这些仅需要匹配相关字符串即可。

4）程序逻辑控制符号。

如 if, else 等常见的编程符号。这种符号和前述的结构性符号通常一起被称为程序语言的关键字。

5）其他一切特殊符号。

如 idt, ddt 等 Verilog-AMS 特有的符号。

对于一些复杂的词法，如数字等，我们在 Lex 的“ 定义段 ”给出其详细模式，且在“ 规则段 ”对这些定义返回一个既定的非终结符。对于简单的词法，如关键字等，我们只在“ 规则段 ”中处理和返回既定值。因为这些符号需要在 Yacc 中能被识别，因此，我们必须使用在 Yacc 已定义的非终结符。Yacc 工具能自动导出包含所有非终结符的头文件，y.tab.h，在 Lex 的源文件中包含即可。因为不是单独使用 Lex，附加的“ 程序段 ”不需要编写。

4.1.2 规则

在 Yacc 中，需要严格按照 Verilog-AMS 的语法产生式制定各种规则，每个规则，都是对应一个或多个产生式。在 Yacc 中，需要为这些产生式定义相应的动作。在 CAMC 中，即是构建输入代码的抽象语法树。对于每个产生式，基本都会有一个或多个抽象的数据结构，这样，每一个归约动作，就需要为这个动作创建一个新的节点。这个节点的各个子树分别指向归约的内容，指向节点本身的指针则返回给下一个规约动作。

这些规则可以做如下的分类：

1）结构性的产生式。

这部分包括整个文本的定义 source_text，这也就是语法解析的起始符。还包括模块的声明 module_declaration 和模块的内容子块 module_items 等。

举例而言，产生式

module_declaration　::=

module_keyword module_identifier [digital_list_of_ports];

[module_items]
endmodule
在 Yacc 中的规则可以写作
module_declaration :
module_keyword module_identifier CC_LPARENTHESIS digital_list_of_ports CC_RPARENTHESIS CC_SEMICOLON module_items CC_ENDMODULE
其中，CC_LPARENTHESIS，CC_RPARENTHESIS，CC_SEMICOLON，CC_ENDMODULE 都是由 Lex 返回的非终结符，分别对应左括号、右括号、分号和关键字 *endmodule*。
而我们创建的抽象数据结构为
typedef struct module_declaration {
identifier *pModuleIdentifier;
identifier *pDigitalListOfPorts;
module_items *pModuleItems;
}module_declaration;
显然，对于产生式右端的非终结符，或有具体意义的终结符（如数字），我们总是需要在产生式对应的抽象数据结构中为其定义相应的变量。

2）属性的产生式。
这部分通常定义了各种可供访问的电气属性。例如产生式：
nature_declaration :=
nature nature_name
endnature

3）声明的产生式。
这部分包括各种常量和变量列表的声明、模块参数的声明和模块节点的类型声明。
例如浮点数变量的声明规则：
real_declaration :
CC_REAL list_of_identifiers CC_SEMICOLON
整数变量的声明规则：
integer_declaration :
CC_INTEGER list_of_identifiers CC_SEMICOLON
模块输入输出节点的声明则包括下述两条规则：
inout_declaration :
CC_INOUT list_of_inout_identifiers CC_SEMICOLON
以及模块参数的声明
parameter_declaration :
CC_PARAMETER opt_type list_of_param_assignments CC_SEMICOLON
通常而言，程序变量的初始化可以和声明在一起。为了简便起见，我们规定仅有模块的参数可以在声明时被初始化。也就是说，因为模型的参数常常有对应的物理意义，因而必然会具有一个默认的值。而其他的模型内部变量，则需要另行赋值。因此，我们仅对模型参数定义赋初值的产生式：
declarator_init :
parameter_identifier CC_ASSIGN constant_expression
其中 CC_ASSIGN 是由 lex 返回的一个非终结符，它对应等号。

创建的抽象数据结构为

```
typedef struct declarator_init {
struct identifier *pParameterIdentifier;
constant_expression *pConstantExpression;
struct declarator_init *pNext;
}declarator_init;
```

注意到这个结构最后有一个指向下一个声明初始化语句的指针，但在这条规则中，并没有相应的对应项。这是因为存在另一条规则

list_of_param_assignments : declarator_init | declarator_init CC_COMA list_of_param_assignments

这是一个右递归的产生式，它将一些连续的声明初始化语句归约成一个非终结符。但是，从语义上讲，单个的声明语句和多个声明语句并没有什么不同。为了在生成的语法树种尽量减少不必要的语法层次，对于这两个规则，我们只分配了一个数据结构，并对这个结构进行扩充，使其拥有一个指向下一条语句的指针，从而在右递归的归约中，把它们串成一个链表，并返回链表的头指针。即，对模型参数的语法定义如下：

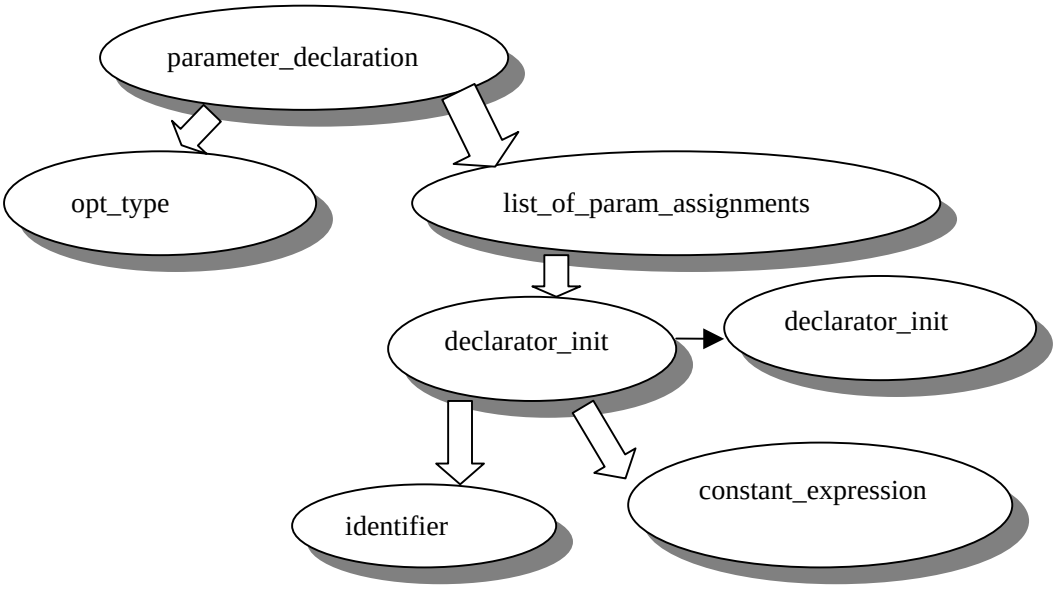


图 19parameter_declaration

另外需要注意的是，由于 Yacc 使用的是“ 自下而上 ”的分析方法，因此只能解决右递归。如果我们把上面的规则改写为

List_of_param_assignments : Declarator_init | list_of_param_assignments CC_COMA declarator_init

就成了一个左递归。虽然在语法的表达能力上是相同的，但是 Yacc 并不能正确实现，必须写成右递归的形式。

4）实例化的产生式。

通常是对一些模块的实例化语句

5）混合信号的产生式。

关于一些混合信号相关的内容

6）行为语句的产生式，对信号行为级的描述

比如包含模拟行为的语句：

Analog_block :

CC_ANALOG analog_statement

其中 CC_ANALOG 表示关键字 analog。表达式对应的数据结构为

typedef struct analog_block {

analog_statement *pAnalogStatement;

struct analog_block *pNext;

}analog_block;

和上例类似，这是因为在一个模块中，也可以由多个描述模拟电路行为的流程块并列。因此需要在数据结构中保留指向下一块的指针。实际上，我们还可以找到这条规则

Module_items :

| module_item module_items | analog_block module_items

这条规则显示了 Yacc 有能力处理从Φ中归约的形式。和前例相比，我们并不能因为 analog_block 可以右递归生成 module_items 而对这两个产生式公用一个结构。但是，我们不需要再每次递归过程中都生成一个新的 module_items 结构，而是只需有一个 module_items 结构，它管理着两个子队列 module_item 和 analog_block，在每次递归时把 module_item 和 analog_block 分别插入到这两个队列中就可以了。这样，归约会形成如下的语法结构：

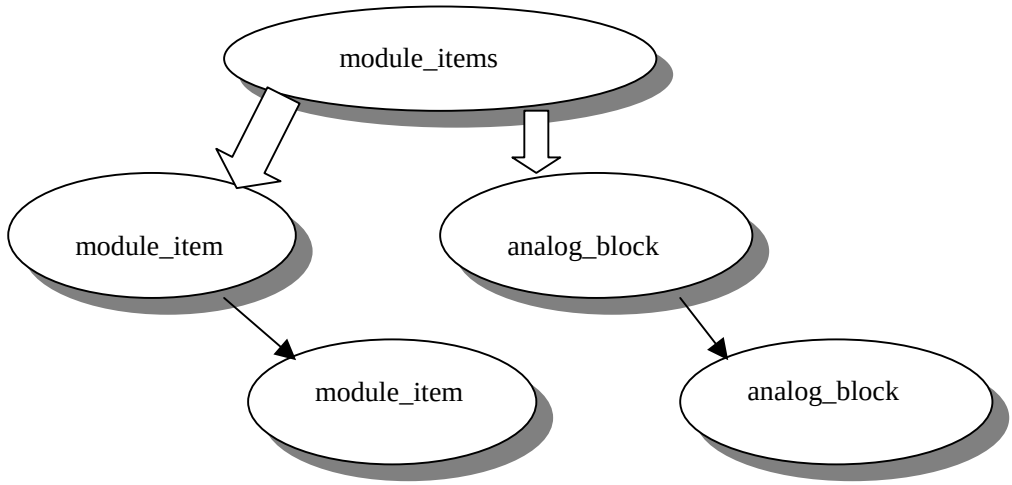


图 20module_items

更为常见的是一些分支控制的语句，如 IF，WHILE, SWITCH 等，为这些分支控制语句建立的抽象数据结构就直接指向不同的分支。但是因为程序员可能在任一分支上都留为空，我们必须改写一下语法规则。典型的 IF 语句会包含三个规则：

1．conditional_statement :

CC_IF CC_LPARENTHESIS expression CC_RPARENTHESIS statement_or_null conditional_statement_else

2．conditional_statement_else:

| CC_ELSE statement_or_null

3．statement_or_null :

CC_SEMICOLON | statement

产生的结构如图：

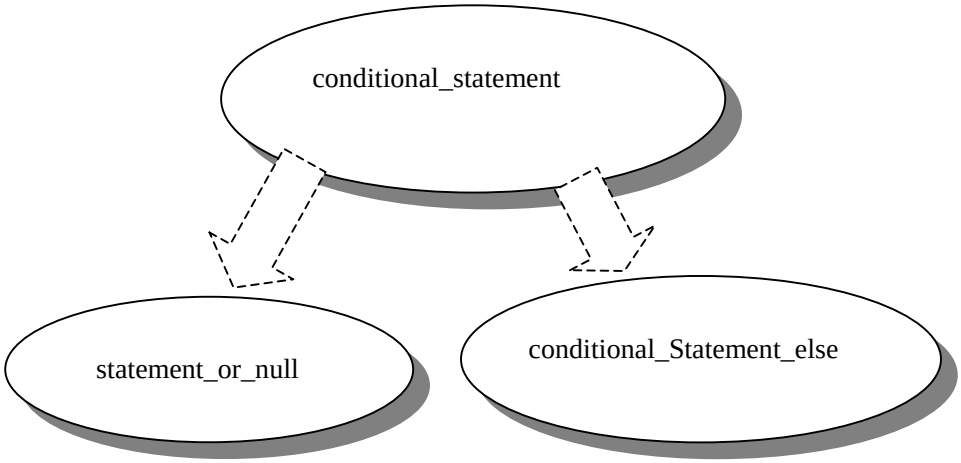


图 21conditional_statement

在稍后的遍历过程中，也必须很小心的检查分支是否为空。

7）模拟表达式的产生式。

这包括一些作用于模拟信号的特定表达式，比如模拟赋值语句：

analog_branch_contribution :

bvalue CC_ANALOG_ASSIGN analog_expression CC_SEMICOLON

其中 CC_ANALOG_ASSIGN 表示模拟赋值符号<+。

8）通用表达式的产生式。

为了减少抽象语法结构的类型，对于一般的运算表达式，我们仅定义了一个结构：

```
typedef struct expression {
primary *pPrimary;
operator *pUnaryOperator;
operator *pBinaryOperator;
struct expression *pExpression1;
struct expression *pExpression2;
function_call *pFunctionCall;
struct built_in_function *pBuiltInFunction;
struct expression *pNext;
}expression;
```

这样，所有由非终结符 expression 产生出来的符号，包括单独的变量和常量、一元和二元算术运算、函数调用和内建函数的调用，都可以用一个结构来表示。程序通过读取各个结构子项是否为空，来判断当前表示的是何种类型的表达式。

9）其他，程序语言中所需的一些其他杂项

基于以上的规则，我们书写 Yacc 的 “ 规则段 ”。在每个规则的动作中，我们引用 Yacc 的默认变量，$$、$1、$2 等等，分别表示规则中左端的非终结符和右端的第一、第二个终结符，并以此类推。Yacc 会自动保存返回的指针$$，在下次归约中作为被规约的终结符传递回来。因此，规约过程中创建的节点最终会形成一棵树。例如，对表达式 if (Vgs –Vth <= Vds)创建的语法树为

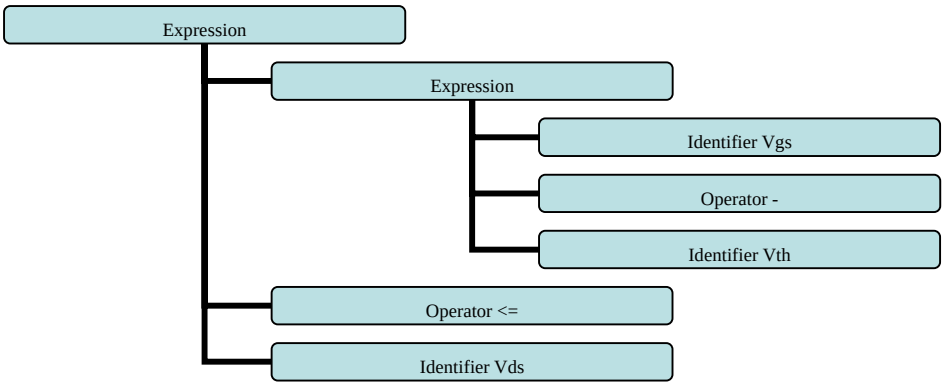


图 22 抽象语法树

在 CAMC 中，Yacc 的“ 定义段 ”包含了一些辅助性的宏定义，如最常用的宏

```
#define NEW(pointer,type) \
{\
    (pointer)=(type*)malloc(sizeof(type));\
    memset ((pointer), 0, sizeof(type));\
}
```

可以方便地分配一块新的结构。这一段还包括

1）Yacc 中用到的内置变量，声明为 extern 的变量

2）符号表的头指针

3）后端函数，在前端的主函数中调用

CAMC 中 Yacc 的“ 程序段 ”完成程序参数的提取和输入文本的读取工作。

4.1.3 符号表

在创建语法树的过程中，一个需要独立实现的过程是符号表。所谓符号表，即是保存了输入代码中所有自定义符号，比如变量、常量等的一个表格。典型的，当程序中声明了一个变量以后，就需要在符号表中保存这个符号。以后在程序中引用这个变量，如读取这个变量的值，或是对这个变量重新赋值时，都需要从符号表中查找这个变量的信息，这样才能保证同一个符号引用的是同一个变量，保证程序逻辑的正确性。因此，符号表的一个基本特征是能够实现快速的查找和插入。

使得符号表变得复杂的一个原因是每个变量都会有一定的作用域。这就是说，同样的一个符号，虽然在同一段程序文本中出现，但由于上下文的不同，完全可能代表不同的变量，比如在这段程序中

```
//定义域 1
real I;
I = 3.1;
……
{
    //定义域 2
    integer I;
```

```
        I = 1;
    }
    ……
```

I 的前后三次出现代表了两个无关的变量。

为了区分不同的定义与，在 CAMC 中，我们使用分段式的符号表来管理定义域。每个定义域都有独立的符号表，被嵌套的定义域的符号表保存指向其外部定义域的符号表的指针。查找的时候，总是从当前的定义域开始，逐步向外搜索。这样，我们得到的符号就是最近作用域内的符号。这是符合程序的语义的。

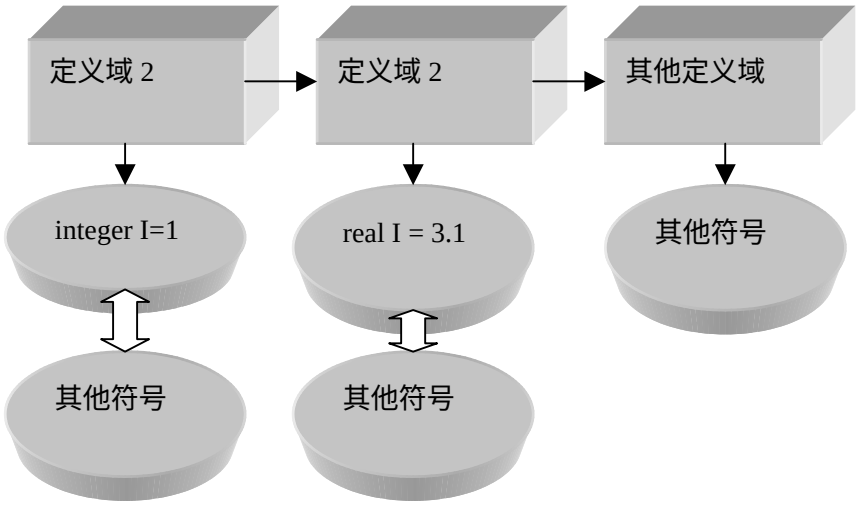


图 23 符号表

在每个定义域的内部，使用双向链表来管理符号表，能够做到动态的插入。插入时，我们保持这个链表按照字母序有序；查找时，只需要使用顺序查找法搜索。链表的插入和查找的代价都是 O(n)，考虑到模型中的符号数量通常不是很大，这样的代价是可以接受的，实践中也没有遇到性能瓶颈。在后端的结构化代码生成和 Jacobi 矩阵生成过程中，也需要使用一些特殊的符号表，或借用符号表的结构实现其他的记录功能，这些我们将在后端的实现中详细介绍。

4.1.4 前端代码结构

```
VP_____                         前端根目录
    |_____  vcLex.l             词法分析
    |_____  vcYacc.y            语法分析
    |_____  abytree.h           抽象数据结构
    |_____  table.h, table.c    符号表
    |_____  Makefile            编译配置文件
```

4.2 后端实现

CAMC 的后段被称为 CP(Code Printer)。CP 负责从语法树中生成 C 语言代码。为了适应不同的仿真器，CAMC 在后端的实现上采用面对对象的技巧，定义了一组和仿真器相关的虚拟接口，这样，不同的仿真器被虚拟化成不同的对象。代码生成时，主线程只需要调用相应对象的接口，而无需关系具体的接口实现细节，

从而尽可能做到了仿真器间的可移植性。

在生成 C 语言代码的过程中，大致可以分为两部分工作。

4.2.1 结构性代码

首先是结构性的代码。这部分代码的生成和电路的运算无关，只是针对仿真器的 API 生成必要的结构性代码框架。这部分代码的形式相对比较固定，需要根据模块的输入输出节点、参数列表等生成符合 API 的代码。在现有的 CAMC 中，只实现了针对仿真器 zspice 的代码，但是，也可方便地移植到其他类似的电路仿真器上。

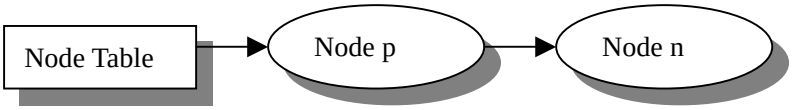
要能被 zspice 调用，必须提供如下的接口：

1）注册模型
zspice_module_register
2）获取模型信息
module_get_info
3）生成模型
module_model_new
4）模型初始化
module_model_initialize
5）查询模型参数
module_model_ask_parameter
6）设定模型参数的默认值
module_model_default_parameter
7）设定模型参数的值
module_model_set_parameter
8）设定模型的扫描参数
module_model_set_sweep_parameter
9）实例化模型
module_instance_new
10） 模型实例初始化
module_instance_initialize
11） 加入模型实例中的一个新节点
module_instance_set_node
12） 按照节点序号查找节点名称
module_instance_get_node
13） 按照节点名称获得节点序号
module_instance_get_node_index
14） 查询模型实例的参数
module_instance_ask_parameter
15） 设定模型实例的参数默认值
module_instance_default_parameter
16） Jacobi 矩阵的填充
module_instance_evaluate

其中接口 1）到接口 15）的都仅仅使用了模型名称、输入输出节点和模型参

数。因此，对这些符号，我们在前端都建立了单独的符号表 ModuleTable，NodeTable 和 ParamTable。这样，产生结构代码的时候，就不再需要遍历整个抽象语法树，而是只需要在这三个符号表中依次遍历各个符号就可以了。

假设模型有两个节点 p 和 n，则我们有节点符号表



因此，接口 13）生成如下：

```
module_instance_get_node_index (MODULENAME_instance_get_node_index)
{
    p_MODULENAMEinstance instance =
            (p_ MODULENAMEinstance) myInstance->_instance;
    if (!strcmp(name,"GND"))    {
        return E_RHS_NODE_GND;
    }
    else if (!strcmp(name,"p"))    {
        return E_RHS_NODE_p;
    }
    else if (!strcmp(name,"n"))    {
        return E_RHS_NODE_n;
    }
    else {
        return 0;
    }
}
```

其中，“GND”是地，不需要显式声明，”p”和”n”则是从节点符号表中读取而出。

所有的接口代码都会输出到一个 C 文件中。其中接口 16）虽然也会在这里定义，但其主要的计算内容将在另外的.include 文件中实现，仅在作 C 编译时被这个文件包含。

4.2.2 头文件生成

目标 C 语言所使用的头文件作为一个单独的功能输出。这部分的功能比较简单，头文件中所包含的主要是两部分信息：

1）zspice 需要的辅助性宏定义和变量

2）对应于模型的数据结构、模型实例的数据结构、参数的变量实例化。

和结构性代码的输出类似，头文件生成也只需要遍历模型、节点和参数的符号表就可以完成。

4.2.3Jacobi 矩阵生成

模型编译器的后端中最重要的模块就是求解 Jacobi 矩阵。在上文中我们已经知道，Jacobi 就是由各个支路方程对其相关节点的贡献组合而成。比如，在 $I_{ds} = Beta \cdot ((V_{gs} - V_{th}) \cdot V_{ds} - \frac{V_{ds}^2}{2})$，显然这个支路和节点 g，d，s 相关，因此，需要分别求得 I_{ds} 对 V_g, V_d, V_s 的偏导。在真正的模型实现中，因为一些变量经过复杂的运算，相关联的节点也会不断增加。 我们定义一个变量的关联节点集合由如下的规则生成：

1）如果 x = f(Va)，则 NodeSet(x) = {a}

2）如果 x = y ***op*** z，则 NoteSet(x) = NodeSet(y)∪NodeSet(z)

可以看出，变量的相关节点的集合只有一种归并操作，是一种只加不减的集合。考虑到为满足基尔霍夫电压定律，对节点电压的引用总是以其差值的形式出现的，我们在关联节点集合的数据结构定义上，使用一个双元组(x,y)来记录一对相关节点。即有结构

```
typedef struct vbranch {
    char name[128];
    int direct;
    identifier *pLeftNode;
    identifier *pRightNode;
    struct vbranch *pNext;
}vbranch;
```

对于每个变量而言，我们将其关联的节点对串成一个链表。在这个链表上，我们定义了

归并　　vbranch_merge

和克隆　　vbranch_clone

两种操作。为了加快归并的速度，我们希望这些节点对至少是偏序的。因此，我们规定其中左节点按字母序小与右节点，比如 $V_d < V_g, V_g < V_s$，如果需要交换左右节点，则用“ 方向 ”标志位标记。节点对相连时，按照左节点的字母序排列。在节点组中，本身并不存储节点的信息，而是直接指向节点符号表中相应的节点。

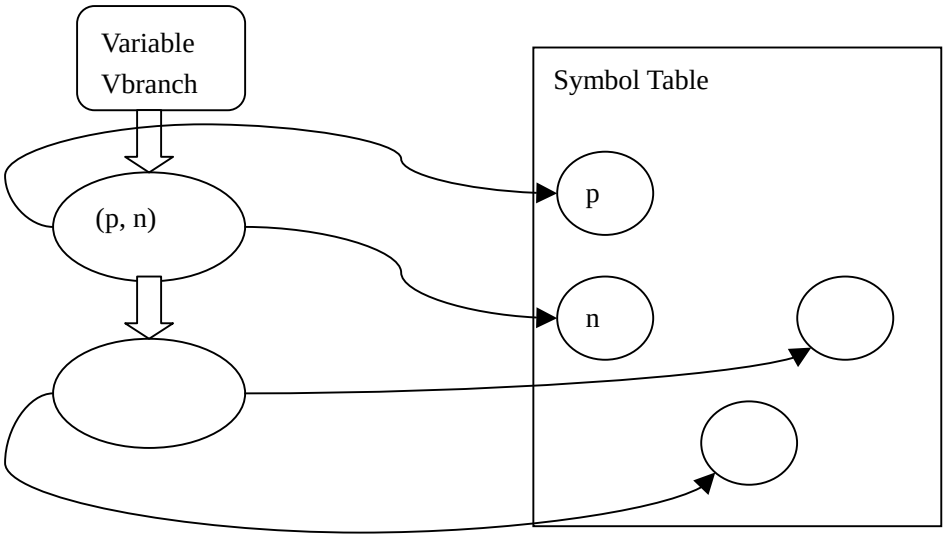


图 24 节点符号表

在上例中，变量V_{gs}相关的节点符号表是(V_g, V_s)，变量V_{ds}相关的节点符号表是(V_d, V_s)，则需要在生成变量I_{ds}，我们需要先克隆V_{gs}和V_{ds}的节点符号表，然后归并为一个节点符号表，从而生成一个新的属于变量I_{ds}的符号表。需要注意的是，由于节点符号表是偏序而非全序。归并的代价是 O(n^2)级的。

在得到了相关联的节点以后，我们就可以依次对这些节点求导。事实上，我们所依据的运算无碍乎如下规则：

$$\frac{\partial(f+g)}{\partial x} = \frac{\partial f}{\partial x} + \frac{\partial g}{\partial x}$$

$$\frac{\partial(f \cdot g)}{\partial x} = \frac{\partial f}{\partial x} g + f \frac{\partial g}{\partial x}$$

$$\frac{\partial(\frac{1}{f})}{\partial x} = -\frac{1}{f} \cdot \frac{\partial f}{\partial x}$$

等等。

因为求导是一个递归的过程，我们可以通过遍历抽象语法树来实现。遍历时，有时需要求解表达式的原值，有时则需要求导，因此，需要区别这两种操作。对于第二种操作，还需要知道当前变量的相关节点符号表。例如，对于乘法操作，我们有

```
if (!strcmp (pExpression->pBinaryOperator->OpName, "*")) {
        output_expression (_DERIVATE_,
            pExpression->pExpression1, pVBranch);
        ccoutput ("*");
        output_expression (_STATIC_,
            pExpression->pExpression2, NULL);
        ccoutput ("+");
```

```
            output_expression (_STATIC_,
                pExpression->pExpression1, NULL);
            ccoutput ("*");
            output_expression (_DERIVATE_,
                pExpression->pExpression2, pVBranch);
    }
```

其中，_DERIVATE_和_STATIC_表示求导或求原值，对于_DERIVATE_类型的输出，还需要传递当前变量的 VBranch 链表。

例如，若我们有如下的表达式

$$Vgs = V(g,s)$$

$$Vds = V(d,s)$$

$$Ids < +Beta*((Vgs-Vth)*Vgs-((Vds^2)/2))$$

则对应的RHS即为

$$RHS = Beta*((Vgs-Vth)*Vgs-((Vds^2)/2))$$

同时，我们已经记下相关节点对为(g,s)和(d,s)，则可以递归生成相应的Jacobian矩阵：

$$\frac{\partial Ids}{\partial V(g,s)} = Beta*(1*Vgs+(Vgs-Vth)*1)$$

$$\frac{\partial Ids}{\partial V(d,s)} = Beta*(-Vds)$$

最终，填充到整个电路的矩阵中，得到

$$\begin{array}{c} \\ d \\ s \end{array} \begin{array}{c} \begin{array}{ccc} g & d & s \end{array} \\ \begin{bmatrix} Beta*(Vgs+(Vgs-Vth)) & Beta*(-Vds) & -Beta*(Vgs+(Vgs-Vth))-Beta*(-Vds) \\ -Beta*(Vgs+(Vgs-Vth)) & -Beta*(-Vds) & Beta*(Vgs+(Vgs-Vth))+Beta*(-Vds) \end{bmatrix} \end{array}$$

需要指出的是，只有出现了 analog_branch_contribution 语句的时候，才会定义一个分支方程。因此，所有的 Jacobi 矩阵的填充动作都是在遍历到 analog_branch_contribution 语句时才发生的。但是，各个变量赋值和运算的表达式是和 analog_branch_contribution 语句并列的，对于某个变量的原值和导数会在不同的地方被引用和赋值。因此，在最终的输出代码中，我们需要不断保存和读取变量的值。实际上，仿真器在实例化模型的时候，会为每个变量开辟空间，因此只需要读取这些值就可以了。而对于微分值，在仿真器中并没有声明，因此我们需要显式地声明局部变量 VariableName_Vp_n 表示变量对节点对(p,n)的微分。在抽象数据结构 primary 对应任意变量的标识符，相应的后端代码中有如下形式：

```
        if (flag == _DERIVATE_) {
            int vb = 0;
            pIdTable = table_look_up_id (&IdTable, pPrimary->pIdentifier);
```

```
            pVB = pIdTable->pId->pVBranch;
            while (pVB) {
                if (pVB->pLeftNode->index == pVBranch->pLeftNode->index
            && pVB->pRightNode->index == pVBranch->pRightNode->index)
                    vb = 1;
                pVB = pVB->pNext;
            }
            if (vb)
            ccoutput ("%s_V%s_%s", pPrimary->pIdentifier->name,
                pVBranch->pLeftNode->name,
                pVBranch->pRightNode->name);
            else
                ccoutput ("0");

        }
        else if (flag == _STATIC_) {
            if (type == PARAMETER)
                ccoutput ("_ipv(%s)", pPrimary->pIdentifier->name);
            else
                ccoutput ("%s", pPrimary->pIdentifier->name);
        }
```

在_STATIC_情况下，即引用原值时，使用仿真器内部定义的宏_ipv 来读取变量值；在_DERIVATE_情况下，即引用微分值时，首先查找这个变量相关的节点对中有没有微分的节点对，有的话输出定义的局部变量，否则为 0。

4.2.4 后端代码结构

```
CP_____                                  后端根目录
    |_____  printtree.h, printtree.c     打印抽象语法树
    |_____  printh.h, printh.c           头文件生成
    |_____  printc.h, printc.c           接口文件生成
    |_____  printevaluate.h, printevaluate.c  Jacobi 矩阵生成
    |_____  Makefile                     编译配置文件
```

4.3 编译器优化和测试

1) 冗余消除(Reduntance Elimination)：在求解Jacobian矩阵的过程中，如果某个变量和一个分支电压无关，则求得的偏微分值必为零。因此，在初始结果中，包含了大量的零节点。去除零节点和与之做乘法等运算的节点，可以减少不必要的计算时间。
2) 常量替代：如果某个节点的子节点都是常量，则可以在编译过程中先行计算该节点的值，并舍弃所有子节点。
3) 节点共享(Node Sharing)：CAMC采用哈希(Hash)查找法来匹配相同的计算节

点。对于已有的计算节点，只需指向原先的节点即可。

4） 循环优化(Loop Optimization)：如前所述，电路仿真器使用迭代求解结果。因此，减少循环中的重复计算，能最有效地减少仿真时间。CAMC对于仅依赖于输入参数的节点设置一标志位，使之只在第一次迭代循环中计算子节点的值，以后则可以直接引用该值。

事实上，对于打开足够优化选项的C语言编译器，也可以再次对CAMC的输出代码做算法1、2、3的优化。但是，因为C语言编译器并不知道这些代码会被循环计算，所以算法4的优化是不可替代的，也是最有效果的。

使用优化以后的CAMC，参照BSIM Level 1模型（对应spice Level 4），我们试验了用于MOS管直流仿真。图4给出了一个NMOS管的Vgs—Ids曲线。

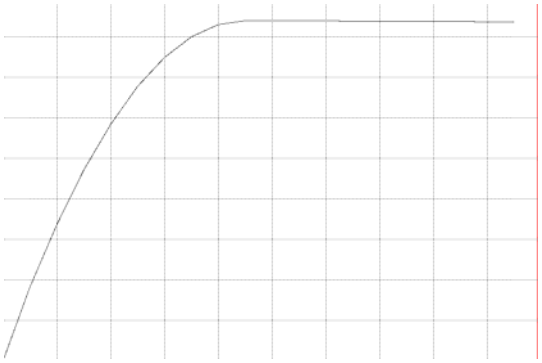


图 25 BSIM1 MOSFETVs-Ids 特性曲线

VFB=-0.6 PHI=0.7 K1=0.0 K2=0 ETA=0 MUZ=600 DL=0 DW=0 U0=0 U1=0 X2MZ=0 X2E=0 X3E=0 X2U0=0 X2U1=0 MUS=600 X2MS=0 X3MS=5.0 X3U1=0 TOX=0.02 TEMP=30 VDD=5 CGDO=1.5e-9 CGSO=1.5e-9 CGBO=2.0e-10 XPART=1.0 N0=0.5 NB=0 ND=0 RSH=0 JS=0 PB=0.8 MJ=0.5 PBSW=0.8 MJSW=0.33 CJ=4.5e-5 CJSW=0 WDF=0 DELL=0

4.4 在互连线建模中的应用

4.4.1 瞬态分析

在 CAMC 的开发过程中，zspice 简单的接口为我们提供了很大的便利。但是，在互连线建模的应用中，比较多的需要进行瞬态分析，以得到波形的崎变和时延的估计，而 zspice 本身没有提供瞬态仿真的功能。因此，我们对 zspice 进行了必要的改进，以使其支持瞬态仿真。

zspice 的代码结构可以分为

1）模型加载

2）网表加载

3）电路网络的数据结构和相关操作

4）仿真主程序

5）矩阵方程解析器

我们需要改写的仅是网表的加载和仿真主程序两个模块。

zspice 能支持对多个参数的 DC 扫描，在读取网表时，需要把各个扫描参数链接成一个链表。为了支持 AC 扫描，zspice 有一个内置的参数 freq，仿真函数

在遇到这个参数时，就进行 AC 扫描。在这个基础上，我们增加了一个内置的变量 time。当在网表中读取到 TRAN 语句时，就把 time 变量加到扫描链表中，表示需要做瞬态扫描，同时记录扫描的区段、步长等信息。

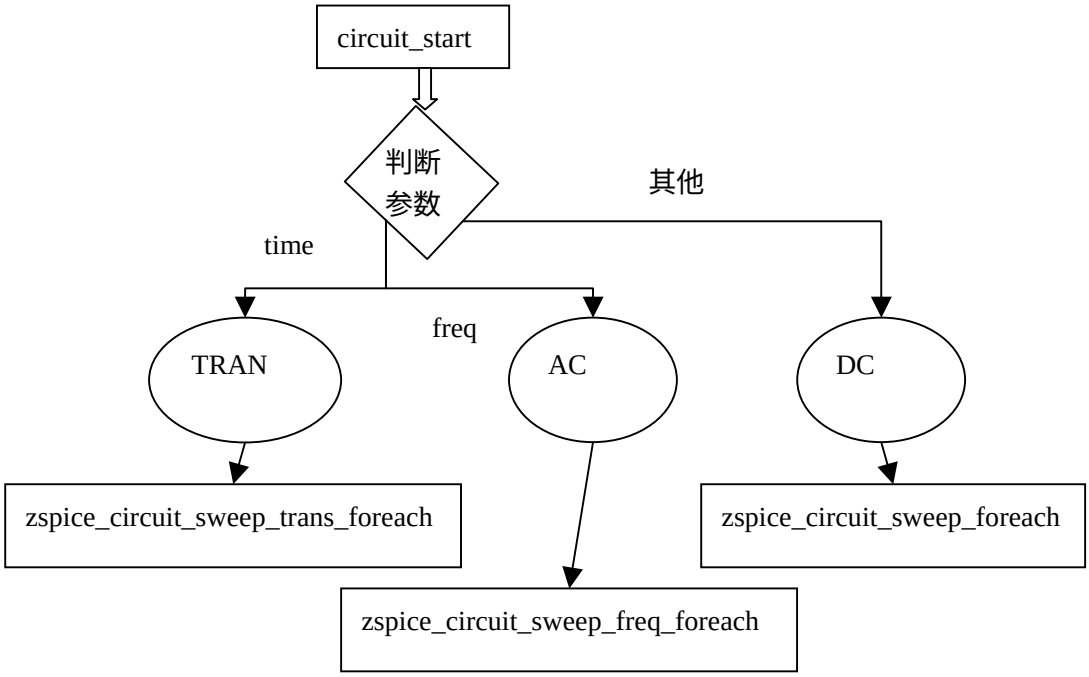


图 26zspice 仿真功能

在 zspice_circuit_sweep_trans_foreach 函数中，我们采用了最简单的前向一步欧拉方法[20]。在欧拉方法中，每次的迭代需要用到前一次各变量的值，在尽量不改动 zspice 内部数据结构的情况下，我们选择为每个变量增加一个相关的局部变量 old_VraiableName。这些 old_*的变量被声明为 static，可以在下次重入时保存旧值。参考前文对 primary 数据结构的输出代码，我们需要添加：

```
else if (flag == _TRANS_) {
            if (type == PARAMETER)
                ccoutput ("old_ipv(%s)", pPrimary->pIdentifier->name);
            else
                ccoutput ("old_%s", pPrimary->pIdentifier->name);
        }
```

对 ddt，idt 等时域相关的算子，我们要按照欧拉方法将微分改写成差分，将积分改写成求和，如 ddt 需要输出：

```
    if (pAnalogExpression->AnalogOperator == DDT) {
        if (flag == _DERIVATE_) {//derivative
            ccoutput("(");
            output_analog_operator_argument(1,
                    pAnalogExpression->pAnalogOperatorArgument, pVBranch);
            ccoutput("/step)");
        }
        else if (flag == _STATIC_) {
            ccoutput("((");
```

```
            output_analog_operator_argument(_STATIC_,
                    pAnalogExpression->pAnalogOperatorArgument, NULL);
            ccoutput("-");
            output_analog_operator_argument(_TRANS_,
                    pAnalogExpression->pAnalogOperatorArgument, NULL);
            ccoutput(")/step)");
        }
    }
```

4.4.2 传输线模型编译

对于传输线的时域仿真，最简单的方法就是把传输线离散成由电感和电容元件表示的等效电路，然后再对这个电路网络进行求解。按照前文的分析，可以把传输线分解成门型网络。

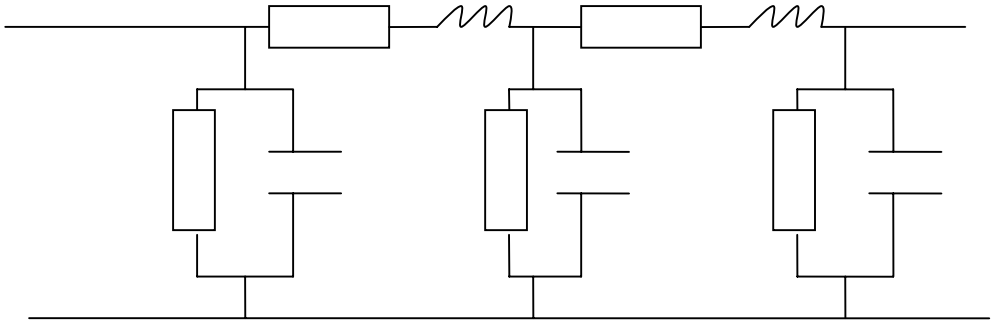


图 27 传输线分布参数模型

采用这种RCLG分离的方法非常直观和简单、物理含义明确，主要缺点是效率低下[15][16][17][18]。在为了保证仿真结果的精度，一般要求每一步时间的步长都至少应该小于信号在单元长度上时延的的1/20，单元长度上的时延至少应该小于激励信号上升下降时间的1/5。

近些年来，也有一些新的传输线离散模型被提出[19]。但在下文的讨论中，我们都使用 RCLG 模型来验证传输线对信号的影响。如图给出了一个正弦信号经过两个反向器和一个传输线以后的波形变化

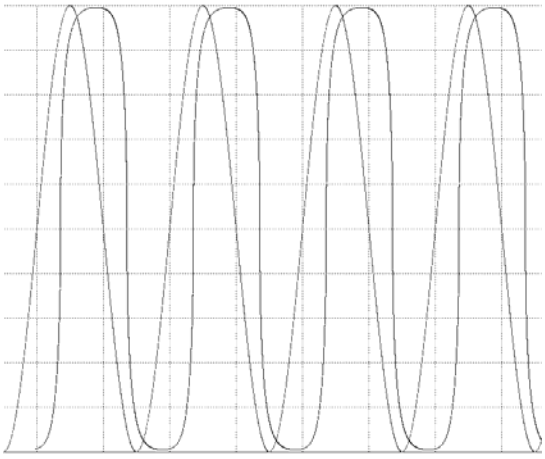


图 28 传输线信号曲线

在 CAMC 中，我们尝试通过类似于 spice 的形式，来定义传输线的模型参数，并由 CAMC 自动转换为分段式的 RCLG 模型。我们有宏：

LOSSLESS_TLINE[a,b,c,d](Z0, TD);

作为内置的一个模型，CAMC 会在编译过程中将其转换为分段式的电容和电感的级联网络。这个模型可以以宏的形式嵌入其他的模型内部，输入的四个节点必须是在其他模型中已经声明的内部或外部的节点，而 Z0 和 TD 两个参数必须是已声明的模型参数。

从 Z0 和 TD 到 L 和 C 的计算非常简单。假设我们把传输线分成 i 段，从上文已知：

$$Z0=\sqrt{\frac{L}{C}} \text{ 和 } TD=i\cdot\sqrt{L\cdot C}$$

因此，我们有：

$$L=\frac{Z0\cdot TD}{i}\text{，}\quad C=\frac{TD}{Z0\cdot i}$$

其中，分段的次数可以在模型中制定，也可以预先设定好，但是从下图中可以看出，在下面的实验中，我们取 Z0=1,TD=1,i 分别取 10,20,100,200，得到的结果基本是一致的。因此，在以后的实验中，我们一致取 i = 10。

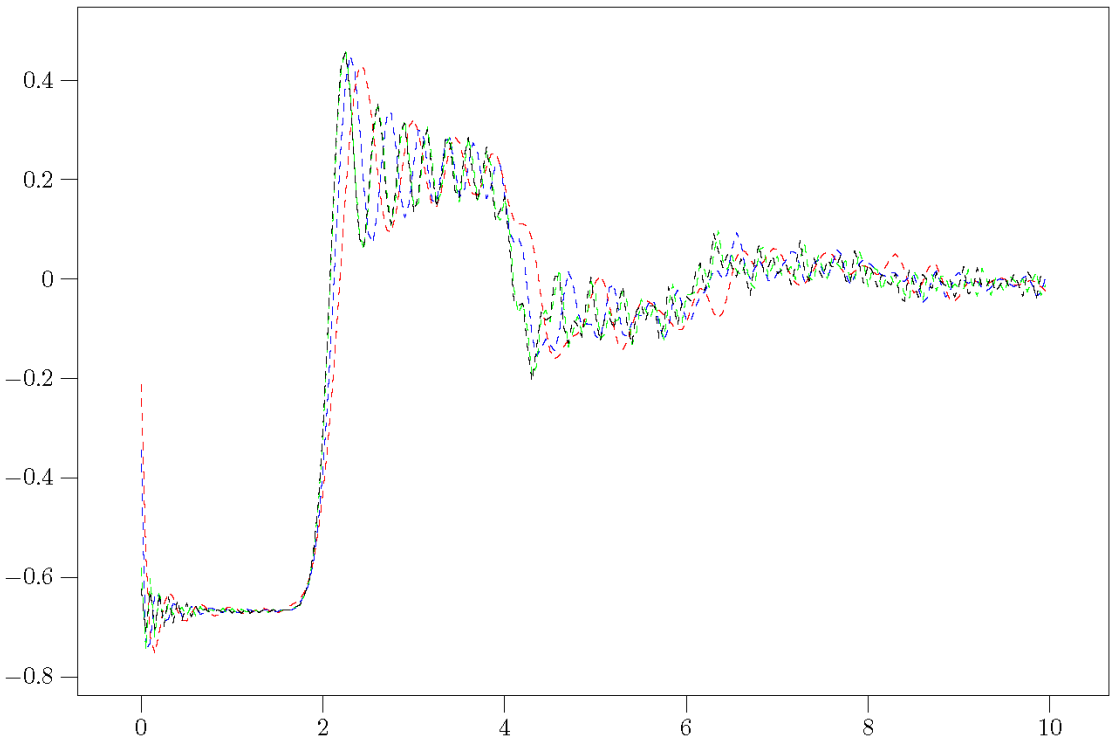


图 29 分段次数比较

和通常的模型不同，分段以后，由于产生了大量的 L 和 C，需要额外的节点来连接。因此，CAMC 需要声明一些模型的内部节点，这些节点无需在电路仿真时的输入网表中定义，而是在导入模型时就同时生成了。实际上，我们编译以后的模型，就是由这些内部节点，以及这些内部节点间的电流、电压关系所构成。

在下面的实验中，我们使用 CAMC 的自动编译功能，简单看一些信号在传输线上遇到的信号完整性问题。在传输线连结的系统中，阻抗匹配是一个重要的问题，它关系到负载所能吸收的功率、传输的效率和信号的稳定性等。下面我们分别从信号源和传输线的匹配，及负载和传输线之间的匹配中来讨论传输线上的信号完整性。

4.4.3 欠载和过载传输线上的信号完整性

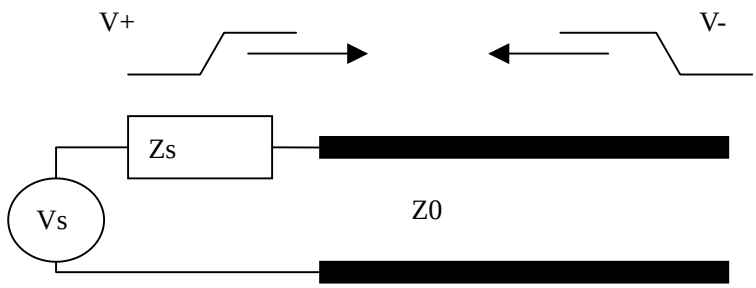


图 30 欠载和过载

当源端阻抗 Zs 比传输线阻抗 Z0 大时，称为欠载传输线。例如，Vs=1V，Zs=2 欧姆，Z0=1 欧姆，负载开路 ，则负载端反射系数为(Zr-Z0)/(Zr+Z0)=-1，源端反射系数(Zs-Z0)/(Zs+Z0)=0.33。当信号从源端进入传输线时，传输线上的初始电压

为 Vinitial=Vs*Z0(Z0+Zs)=0.33V。初始信号沿着传输线向负载端传播，经过延时 TD 后到达负载，因为负载开路，导致反射系数为 1，整个信号被全部反射回源端，这时，负载端的信号为 0.33+0.33=0.66V。经过两倍的延时以后，反射信号到达源端，并产生新的反射分量 0.66V*0.33=0.22V。这时源端的电压是 0.33+0.33+0.22=0.88V。因此，欠载的传输线导致信号产生阶梯状的延时。我们使用 zspice 的方针结果验证这一现象。在下表中，我们选录了一部分实验数据，并和 spice 的输出结果相对比，在 spice 中，我们分别使用了等价参数的 RCLG 网络和相同参数的 spice 内置无损传输线模型，分别得到了(I)和(II)的数据。

zspice		Spice(I)		Spice(II)	
0.00000000	0.000000000	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.050000000	0.289570889	1.000000e-03	9.803941e-03	1.000000e-03	3.333333e-03
0.10000000	0.242526578	1.042072e-03	1.021606e-02	2.000000e-03	6.666667e-03
0.15000000	0.299201901	1.126215e-03	1.103962e-02	4.000000e-03	1.333333e-02
0.20000000	0.350983109	1.294502e-03	1.268257e-02	8.000000e-03	2.666667e-02
0.25000000	0.365152929	1.631077e-03	1.595194e-02	1.600000e-02	5.333333e-02
0.29999999	0.350153522	2.304225e-03	2.242513e-02	3.200000e-02	1.066667e-01
0.34999999	0.328473158	3.440042e-03	3.315219e-02	6.400000e-02	2.133333e-01
0.39999999	0.317527703	4.672715e-03	4.452180e-02	1.000000e-01	3.333333e-01
0.44999999	0.321370052	6.319423e-03	5.928061e-02	1.064000e-01	3.333333e-01
0.49999999	0.332874004	8.195753e-03	7.551724e-02	1.192000e-01	3.333333e-01

绘制成图，我们得到了下图。其中，红色虚线标示出了经过 CAMC 编译后模型在 zspice 中的输出结果，蓝线和绿线分别表示 spice(I)和 spice(II)的输出结果。可以看出，结果是基本一致的：

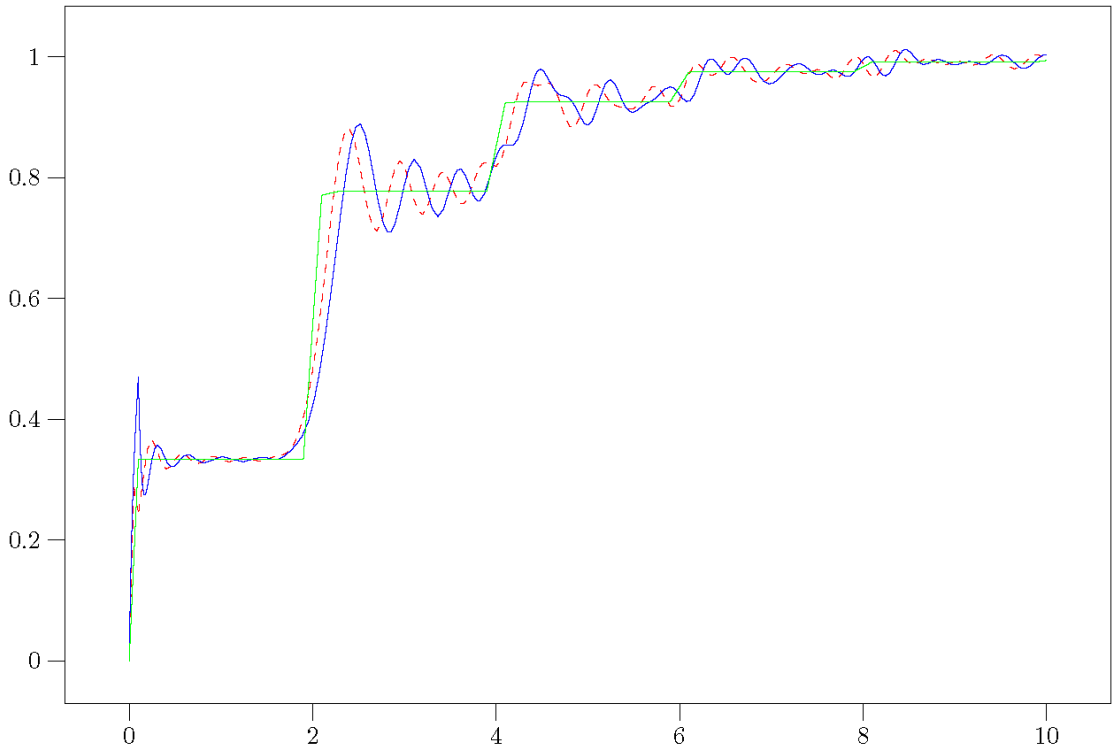


图 31 欠载传输线

当源端阻抗Zs比传输线阻抗Z0小时，成为过载传输线。例如，Vs=1V，Zs=0.5欧姆，Z0=1欧姆，负载开路，则负载端反射系数(Zr-Z0)/(Zr+Z0)=1，源端反射系数(Zs-Z0)/(Zs+Z0)=0.333。当信号从源端进入传输线时，传输线上的初始电压为Vinitial=Vs*Z0/(Z0+Zs)=0.66。初始信号沿着传输线向负载端传播，经过延时TD后到达负载，因为负载开路，整个信号被全部反射，这时负载端的信号为0.66+0.66=1.32V。经过两倍的延时以后，反射信号又到达源端，并产生新的反射分量。经过多次反射后，信号达到稳定。仿真结果如下图，和上例类似的，我们用红色虚线表示zspice的输出结果，蓝色和绿色表示spice(I)和spice(II)的输出结果：

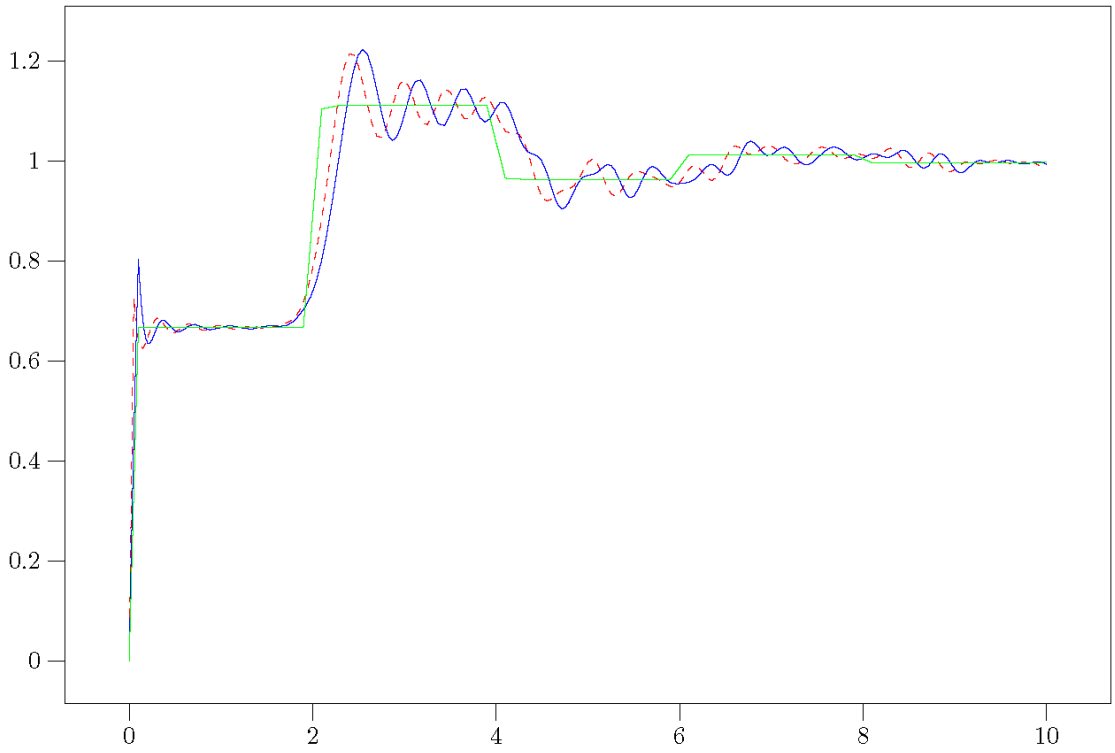


图 32 过载传输线

4.4.4 电抗性负载

当传输线终端连结一个容性负载时，波形有很大的变化。事实上，电容是时间相关的负载，当信号到达电容时考试看起来是短路，而当电容完全充电之后看起来是开路。

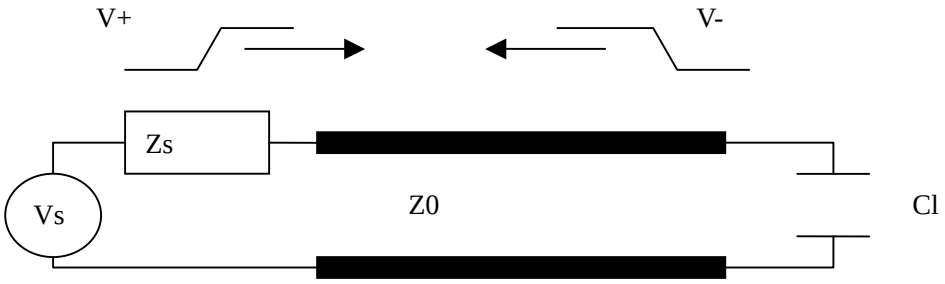


图 33 容性负载

例如，我们取负载电容 Cl 为 1F，驱动源和传输线阻抗都是 1 欧姆。源端的初始电压为 Vinitial=Vs*Z0/(Z0+Zs)=0.5V，源端反射系数为 0。信号到达电容时，负载端反射系数为-1，当电容充满以后，负载端反射系数为 1。因此，一开始信号波形被反射离开负载。此后电容被逐渐充电，负载端的电压也逐渐增加。在源端，过了两倍的延时以后，信号被完全反射，波形被完全抵消，然后随着电容充电，电压逐步上升到稳态。

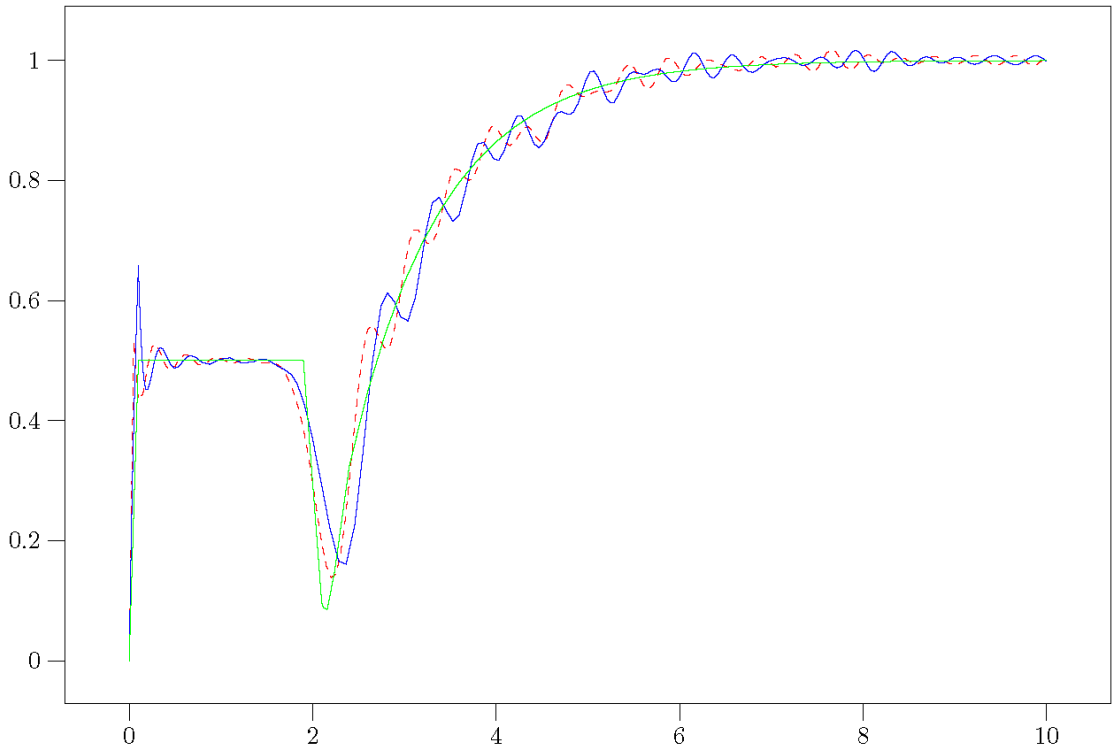


图 34 容性负载

当一个串联电感出现在传输线的负载上时，它也是一个时间相关的负载。一开始，电感好像是开路的，这使得反射系数为 1。电感值的大小决定反射系数保持为 1 的时间，最后，电感以决定 LR 电路的时间常数的速度释放能量。

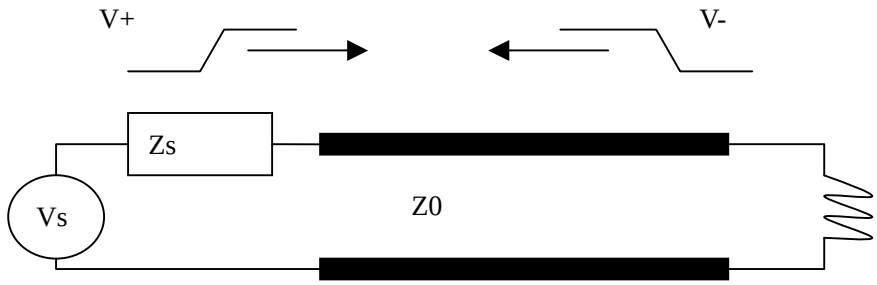


图 35 感性负载

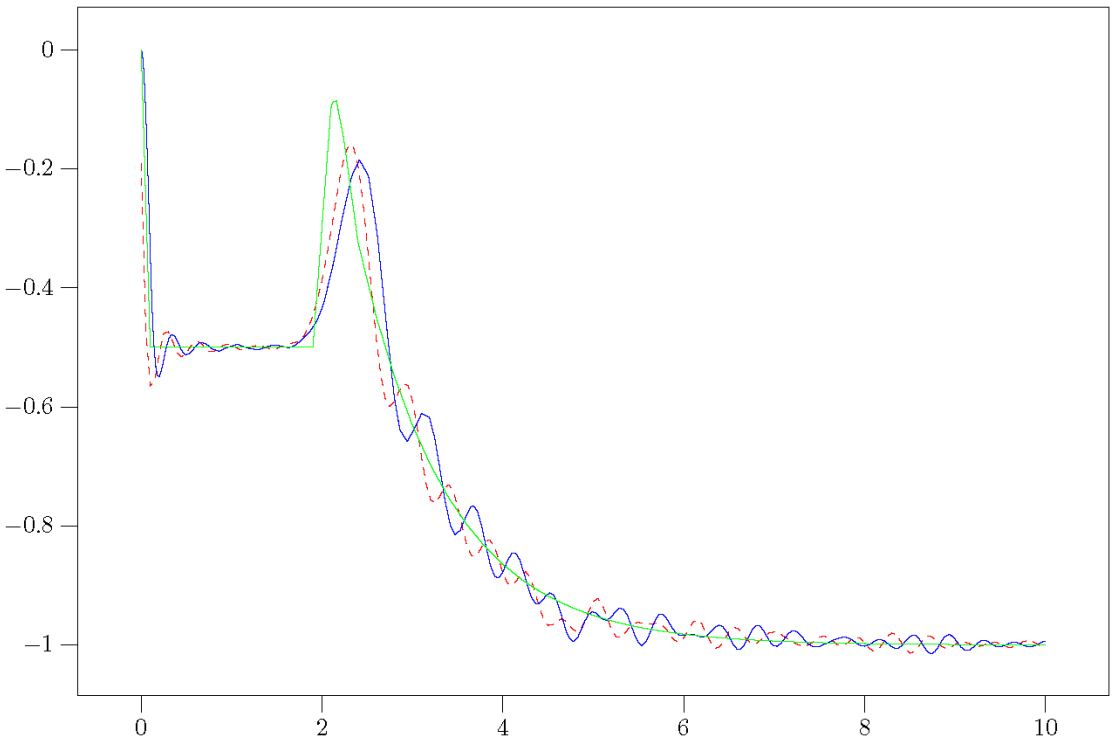


图 36 感性负载的电流响应

4.5 小结

本文给出了一个模型编译器的具体实现，并探讨了模型编译器在互连线模型中的应用。实验表明，使用模型编译器可以方便地生成用于标准仿真器的仿真模型，并且在进行了一些局部的优化以后，不会在性能方面有太大的损失。在应用领域上，模型编译器不仅能应用于传统的器件模型，在互连线的建模和相关的信号完整性，电源完整性等热点领域也有着充分的应用前景。

本文所进行的主要研究工作：

1 研究使用Verilog-AMS对模型进行建模的方法

2 抽取一个Verilog-AMS的语法子集，创建了一个能够编译这个子集的编译器CAMC。

3 为编译器的后端创建符合zspice仿真器接口的生成代码，将器件模型转换为C语言代码

4 在互连线模型等应用中使用模型编译器CAMC进行了初步的研究和分析。

随着集成电路技术继续向高密度、高频率、高速度的方向发展，信号完整性、电源完整性等问题已经成为电路设计的主要难点。使用模型编译器来验证互连设计中的信号完整性问题，无疑为解决这些难题提供了一个新的思路。但是，本文在这方面仅仅做了初步的探索，对于更加复杂的电路结构和更新的器件模型，还有待于进一步实践。

附录 I　MOSFET 器件模型举例

我们试从最简单的 Berkeley Short-Channel IGFET Model(BSIM) Level 1[9]模型中，推倒器件模型描述方式。

首先定义下列参数：

V_{FB} 平带电压

ϕ_S 表面反向电位

K_1 体效应系数

K_2 源漏电荷分享系数

η 漏极引发能障衰退系数

U_0 垂直下降系数

U_1 速度饱和系数

μ_0 载体迁移率

并定义阈值电压为

$$V_{th} = V_{FB} + \phi_S + K_1\sqrt{\varphi_S - V_{BS}} - K_2(\phi_S - V_{BS}) - \eta V_{DS}$$

1. 截止区$V_{GS} \leq V_{th}$]

$$I_{DS} = 0$$

2. 三极管区[$V_{GS} > V_{th}.and.0 < V_{DS} < V_{DSAT}$]

$$I_{DS} = \frac{\mu_0}{[1+U_0(V_{GS}-V_{th})]} \cdot \frac{C_{ox}\frac{W}{L}}{(1+\frac{U1}{L}V_{DS})}((V_{DS}-V_{th})V_{DS} - \frac{a}{2}V_{DS}^2)$$

其中 $a = 1 + \frac{gK_1}{2\sqrt{\phi_S - V_{BS}}}$

$$g = 1 - \frac{1}{1.744 + 0.8364(\phi_S - V_{BS})}$$

3. 饱和区[$V_{GS} > V_{th} .and. V_{DS} \geq V_{DSAT}$]

$$I_{DS} = \frac{\mu_0}{[1+U_0(V_{GS}-V_{th})]} \cdot \frac{C_{ox}\frac{W}{L}(V_{GS}-V_{th})^2}{2aK}$$

其中 $K = \frac{1+v_c+\sqrt{1+2v_c}}{2}$

$$V_{DSAT} = \frac{V_{GS}-V_{th}}{a\sqrt{K}}$$

$$v_c = \frac{U_1}{L} \cdot \frac{(V_{GS}-V_{th})}{a}$$

如果需要考虑亚阈值效应，则需要修正为

$$I_{DS,total} = I_{DS,W} + I_{DS}$$

其中 $I_{DS,W} = \frac{I_{\exp} \cdot I_{\lim it}}{I_{\exp} + I_{\lim it}}$

且 $I_{\exp} = \mu_0 C_{ox} \frac{W}{L} (\frac{kT}{q})^2 \cdot e^{1.8} e^{(q/kT)(V_{GS}-V_{th})/n} [1-e^{-V_{DS}(q/kT)}]$

和 $I_{\lim it} = \frac{\mu_0 C_{ox}}{2} \cdot \frac{W}{L} \cdot (3\frac{kT}{q})^2$

不考虑亚阈值效应，我们可以简单得到一个 Verilog-AMS 的模型行为为

```
analog
begin

    Vgs = V(g, s);
    Vds = V(d, s);
    Vbs = V(b, s);

    Vth = Vfb + PHI + K1*(sqrt(PHI-Vbs)) - K2*(PHI-Vbs);
    g0 = 1 - (1/(1.744 + (0.8364*(PHI-Vbs))));
    a = 1 + ((g0*K1)/(2*sqrt(PHI-Vbs)));
    vc = (U1/L)*(Vgs-Vth)/a;
    K = ((1+vc)+sqrt(1+(2*vc)))/2;
    Vdsat = (Vgs - Vth)/(a*sqrt(K));

    if (Vgs <= Vth) //Cutoff
```

```
        Ids = 0.0;
    else
        begin
        if (Vds < Vdsat)        //Trinode
            Ids                                                                    =
            (MU0/(1+(U0*(Vgs-Vth))))*((Cox*W/L)/(1+(U1*Vds/L)))*(((Vgs-Vth)
            *Vds)-((a/2)*Vds*Vds));
        else                        //Saturation
            Ids                                                                    =
            (MU0/(1+(U0*(Vgs-Vth))))*((Cox*W/L)/(2*a*K))*(Vgs-Vth)*(Vgs-Vt
            h);
        end

     I(d,s) <+ Ids;
end
```

经过编译得到的 Jacobi 矩阵生成结果为：

```
#if defined(_STATIC)
double Ids;
#if defined(_DERIVATE)
double Ids_Vg_s=0.0;
double Ids_Vb_s=0.0;
double Ids_Vd_s=0.0;
#endif /*_DERIVATE*/
#endif /*_STATIC*/
#if defined(_STATIC)
double Vdsat;
#endif /*_STATIC*/
#if defined(_STATIC)
double K;
#if defined(_DERIVATE)
double K_Vg_s=0.0;
double K_Vb_s=0.0;
#endif /*_DERIVATE*/
#endif /*_STATIC*/
#if defined(_STATIC)
double vc;
#if defined(_DERIVATE)
double vc_Vg_s=0.0;
double vc_Vb_s=0.0;
#endif /*_DERIVATE*/
#endif /*_STATIC*/
#if defined(_STATIC)
double a;
#if defined(_DERIVATE)
```

```
double a_Vb_s=0.0;
#endif /*_DERIVATE*/
#endif /*_STATIC*/
#if defined(_STATIC)
double g0;
#if defined(_DERIVATE)
double g0_Vb_s=0.0;
#endif /*_DERIVATE*/
#endif /*_STATIC*/
#if defined(_STATIC)
double Vth;
#if defined(_DERIVATE)
double Vth_Vb_s=0.0;
#endif /*_DERIVATE*/
#endif /*_STATIC*/
#if defined(_STATIC)
double Vbs;
#if defined(_DERIVATE)
double Vbs_Vb_s=0.0;
#endif /*_DERIVATE*/
#endif /*_STATIC*/
#if defined(_STATIC)
double Vds;
#if defined(_DERIVATE)
double Vds_Vd_s=0.0;
#endif /*_DERIVATE*/
#endif /*_STATIC*/
#if defined(_STATIC)
double Vgs;
#if defined(_DERIVATE)
double Vgs_Vg_s=0.0;
#endif /*_DERIVATE*/
#endif /*_STATIC*/
#if defined(_STATIC) && defined(_DERIVATE)
Vgs_Vg_s=1.0;
#endif /*_STATIC && _DERIVATE*/
#if defined(_STATIC)
Vgs=BP(g,s);
#endif /*_STATIC*/
#if defined(_STATIC) && defined(_DERIVATE)
Vds_Vd_s=1.0;
#endif /*_STATIC && _DERIVATE*/
#if defined(_STATIC)
Vds=BP(d,s);
```

```
#endif /*_STATIC*/
#if defined(_STATIC) && defined(_DERIVATE)
Vbs_Vb_s=1.0;
#endif /*_STATIC && _DERIVATE*/
#if defined(_STATIC)
Vbs=BP(b,s);
#endif /*_STATIC*/
#if defined(_STATIC)
{
double __sqrt_0=0.0;
#if defined(_DERIVATE)
double __d_sqrt_0=0.0;
#endif /* _DERIVATE */
#if defined(_DERIVATE)
_d_sqrt(__sqrt_0,__d_sqrt_0,((_ipv(PHI)-Vbs)))
#else
_sqrt(__sqrt_0,((_ipv(PHI)-Vbs)))
#endif
#if defined(_STATIC) && defined(_DERIVATE)
Vth_Vb_s=((+(_ipv(K1)*(-Vbs_Vb_s)*__d_sqrt_0))-(_ipv(K2)*(-Vbs_Vb_s)));
#endif /*_STATIC && _DERIVATE*/
#if defined(_STATIC)
Vth=(((_ipv(Vfb)+_ipv(PHI))+(_ipv(K1)*__sqrt_0))-(_ipv(K2)*(_ipv(PHI)-Vbs)));
#endif /*_STATIC*/
}
#endif /* _STATIC */
#if defined(_STATIC) && defined(_DERIVATE)
g0_Vb_s=(-(-(+(0.8364*(-Vbs_Vb_s)))/((1.744+(0.8364*(_ipv(PHI)-Vbs)))*(1.744+(
0.8364*(_ipv(PHI)-Vbs))))));
#endif /*_STATIC && _DERIVATE*/
#if defined(_STATIC)
g0=(1-(1/(1.744+(0.8364*(_ipv(PHI)-Vbs)))));
#endif /*_STATIC*/
#if defined(_STATIC)
{
double __sqrt_0=0.0;
#if defined(_DERIVATE)
double __d_sqrt_0=0.0;
#endif /* _DERIVATE */
#if defined(_DERIVATE)
_d_sqrt(__sqrt_0,__d_sqrt_0,((_ipv(PHI)-Vbs)))
#else
_sqrt(__sqrt_0,((_ipv(PHI)-Vbs)))
#endif
```

```
#if defined(_STATIC) && defined(_DERIVATE)
a_Vb_s=(+(g0_Vb_s*_ipv(K1)*(2*__sqrt_0)-(g0*_ipv(K1))*(2*(-Vbs_Vb_s)*__d_s
qrt_0))/((2*__sqrt_0)*(2*__sqrt_0)));
#endif /*_STATIC && _DERIVATE*/
#if defined(_STATIC)
a=(1+((g0*_ipv(K1))/(2*__sqrt_0)));
#endif /*_STATIC*/
}
#endif /* _STATIC */
#if defined(_STATIC) && defined(_DERIVATE)
vc_Vb_s=(((_ipv(U1)/_ipv(L))*(-Vth_Vb_s))*a-((_ipv(U1)/_ipv(L))*(Vgs-Vth))*a_
Vb_s)/(a*a);
vc_Vg_s=((_ipv(U1)/_ipv(L))*Vgs_Vg_s)/a;
#endif /*_STATIC && _DERIVATE*/
#if defined(_STATIC)
vc=(((_ipv(U1)/_ipv(L))*(Vgs-Vth))/a);
#endif /*_STATIC*/
#if defined(_STATIC)
{
double __sqrt_0=0.0;
#if defined(_DERIVATE)
double __d_sqrt_0=0.0;
#endif /* _DERIVATE */
#if defined(_DERIVATE)
_d_sqrt(__sqrt_0,__d_sqrt_0,((1+(2*vc))))
#else
_sqrt(__sqrt_0,((1+(2*vc))))
#endif
#if defined(_STATIC) && defined(_DERIVATE)
K_Vb_s=((+vc_Vb_s)+(+(2*vc_Vb_s))*__d_sqrt_0)/2;
K_Vg_s=((+vc_Vg_s)+(+(2*vc_Vg_s))*__d_sqrt_0)/2;
#endif /*_STATIC && _DERIVATE*/
#if defined(_STATIC)
K=(((1+vc)+__sqrt_0)/2);
#endif /*_STATIC*/
}
#endif /* _STATIC */
#if defined(_STATIC)
{
double __sqrt_0=0.0;
#if defined(_DERIVATE)
double __d_sqrt_0=0.0;
#endif /* _DERIVATE */
#if defined(_DERIVATE)
```

```
_d_sqrt(__sqrt_0,__d_sqrt_0,(K))
#else
_sqrt(__sqrt_0,(K))
#endif
#if defined(_STATIC)
Vdsat=((Vgs-Vth)/(a*__sqrt_0));
#endif /*_STATIC*/
/* variable `Vdsat': never used in source expression -> computation of deriv skipped
*/
}
#endif /* _STATIC */
if
((Vgs<=Vth))
{
#if defined(_STATIC)
Ids=0.0;
#endif /*_STATIC*/
#if defined(_STATIC) && defined(_DERIVATE)
Ids_Vg_s=0.0;
Ids_Vb_s=0.0;
Ids_Vd_s=0.0;
#endif /*_STATIC && _DERIVATE*/
}
else
{
if
((Vds<Vdsat))
{
#if defined(_STATIC) && defined(_DERIVATE)
Ids_Vd_s=((((_ipv(MU0)/(1+(_ipv(U0)*(Vgs-Vth))))*(-(((_ipv(Cox)*_ipv(W))/_ipv(
L))*(+(_ipv(U1)*Vds_Vd_s)/_ipv(L)))/((1+((_ipv(U1)*Vds)/_ipv(L)))*(1+((_ipv(U1
)*Vds)/_ipv(L))))))*(((Vgs-Vth)*Vds)-(((a/2)*Vds)*Vds)))+(((_ipv(MU0)/(1+(_ipv(
U0)*(Vgs-Vth))))*(((_ipv(Cox)*_ipv(W))/_ipv(L))/(1+((_ipv(U1)*Vds)/_ipv(L)))))*
(((Vgs-Vth)*Vds_Vd_s)-((((a/2)*Vds_Vd_s)*Vds)+(((a/2)*Vds)*Vds_Vd_s)))));
Ids_Vb_s=(((-(_ipv(MU0)*(+(_ipv(U0)*(-Vth_Vb_s))))/((1+(_ipv(U0)*(Vgs-Vth)))*
(1+(_ipv(U0)*(Vgs-Vth)))))*(((_ipv(Cox)*_ipv(W))/_ipv(L))/(1+((_ipv(U1)*Vds)/_i
pv(L))))*(((Vgs-Vth)*Vds)-(((a/2)*Vds)*Vds)))+(((_ipv(MU0)/(1+(_ipv(U0)*(Vgs-
Vth))))*(((_ipv(Cox)*_ipv(W))/_ipv(L))/(1+((_ipv(U1)*Vds)/_ipv(L)))))*((-Vth_Vb
_s)*Vds-a_Vb_s/2*Vds*Vds)));
Ids_Vg_s=(((-(_ipv(MU0)*(+(_ipv(U0)*Vgs_Vg_s)))/((1+(_ipv(U0)*(Vgs-Vth)))*(1
+(_ipv(U0)*(Vgs-Vth)))))*(((_ipv(Cox)*_ipv(W))/_ipv(L))/(1+((_ipv(U1)*Vds)/_ipv
(L))))*(((Vgs-Vth)*Vds)-(((a/2)*Vds)*Vds)))+(((_ipv(MU0)/(1+(_ipv(U0)*(Vgs-Vth
))))*(((_ipv(Cox)*_ipv(W))/_ipv(L))/(1+((_ipv(U1)*Vds)/_ipv(L)))))*Vgs_Vg_s*Vd
s));
```

```
#endif /*_STATIC && _DERIVATE*/
#if defined(_STATIC)
Ids=(((_ipv(MU0)/(1+(_ipv(U0)*(Vgs-Vth))))*(((_ipv(Cox)*_ipv(W))/_ipv(L))/(1+((
_ipv(U1)*Vds)/_ipv(L)))))*(((Vgs-Vth)*Vds)-(((a/2)*Vds)*Vds)));
#endif /*_STATIC*/
}
else
{
#if defined(_STATIC) && defined(_DERIVATE)
Ids_Vb_s=(((((((-(_ipv(MU0)*(+(_ipv(U0)*(-Vth_Vb_s))))/((1+(_ipv(U0)*(Vgs-Vth)
))*(1+(_ipv(U0)*(Vgs-Vth)))))*(((_ipv(Cox)*_ipv(W))/_ipv(L))/((2*a)*K)))+((_ipv(
MU0)/(1+(_ipv(U0)*(Vgs-Vth))))*(-(((_ipv(Cox)*_ipv(W))/_ipv(L))*(((2*a_Vb_s)*
K)+((2*a)*K_Vb_s)))/(((2*a)*K)*((2*a)*K)))))*(Vgs-Vth))+(((_ipv(MU0)/(1+(_ipv(
U0)*(Vgs-Vth))))*(((_ipv(Cox)*_ipv(W))/_ipv(L))/((2*a)*K)))*(-Vth_Vb_s)))*(Vgs
-Vth))+((((_ipv(MU0)/(1+(_ipv(U0)*(Vgs-Vth))))*(((_ipv(Cox)*_ipv(W))/_ipv(L))/(
(2*a)*K)))*(Vgs-Vth))*(-Vth_Vb_s)));
Ids_Vg_s=(((((((-(_ipv(MU0)*(+(_ipv(U0)*Vgs_Vg_s)))/((1+(_ipv(U0)*(Vgs-Vth)))
*(1+(_ipv(U0)*(Vgs-Vth)))))*(((_ipv(Cox)*_ipv(W))/_ipv(L))/((2*a)*K)))+((_ipv(M
U0)/(1+(_ipv(U0)*(Vgs-Vth))))*(-(((_ipv(Cox)*_ipv(W))/_ipv(L))*((2*a)*K_Vg_s))
/(((2*a)*K)*((2*a)*K)))))*(Vgs-Vth))+(((_ipv(MU0)/(1+(_ipv(U0)*(Vgs-Vth))))*(((
_ipv(Cox)*_ipv(W))/_ipv(L))/((2*a)*K)))*Vgs_Vg_s))*(Vgs-Vth))+((((_ipv(MU0)/(
1+(_ipv(U0)*(Vgs-Vth))))*(((_ipv(Cox)*_ipv(W))/_ipv(L))/((2*a)*K)))*(Vgs-Vth))*
Vgs_Vg_s));
#endif /*_STATIC && _DERIVATE*/
#if defined(_STATIC)
Ids=((((_ipv(MU0)/(1+(_ipv(U0)*(Vgs-Vth))))*(((_ipv(Cox)*_ipv(W))/_ipv(L))/((2*
a)*K)))*(Vgs-Vth))*(Vgs-Vth));
#endif /*_STATIC*/
#if defined(_STATIC) && defined(_DERIVATE)
Ids_Vd_s=0.0;
#endif /*_STATIC && _DERIVATE*/
}
}
   _load_static_residual2(d,s,Ids)
   _load_static_jacobian4(d,s,d,s,Ids_Vd_s)
   _load_static_jacobian4(d,s,b,s,Ids_Vb_s)
   _load_static_jacobian4(d,s,g,s,Ids_Vg_s)
```

附录 II　传输线编译结果举例

对于 Z0=1, TD=1 的无损传输线，简单的分为三段，则经过 CAMC 编译得到的代码如下：

```
#if defined(_STATIC)
static double v_tline_0_l_0;
static double v_tline_0_c_0;
static double i_tline_0_l_0;
static double i_tline_0_c_0;
static double old_v_tline_0_l_0=0.0;
static double old_v_tline_0_c_0=0.0;
static double old_i_tline_0_l_0=0.0;
static double old_i_tline_0_c_0=0.0;
if (tran_reset == 1) {
    old_v_tline_0_l_0 = v_tline_0_l_0;
    old_v_tline_0_c_0 = v_tline_0_c_0;
    old_i_tline_0_l_0 = i_tline_0_l_0;
    old_i_tline_0_c_0 = i_tline_0_c_0;
}
static double v_tline_0_l_1;
static double v_tline_0_c_1;
static double i_tline_0_l_1;
static double i_tline_0_c_1;
static double old_v_tline_0_l_1=0.0;
static double old_v_tline_0_c_1=0.0;
static double old_i_tline_0_l_1=0.0;
static double old_i_tline_0_c_1=0.0;
if (tran_reset == 1) {
    old_v_tline_0_l_1 = v_tline_0_l_1;
    old_v_tline_0_c_1 = v_tline_0_c_1;
    old_i_tline_0_l_1 = i_tline_0_l_1;
    old_i_tline_0_c_1 = i_tline_0_c_1;
}
static double v_tline_0_l_2;
static double v_tline_0_c_2;
static double i_tline_0_l_2;
static double i_tline_0_c_2;
static double old_v_tline_0_l_2=0.0;
static double old_v_tline_0_c_2=0.0;
static double old_i_tline_0_l_2=0.0;
static double old_i_tline_0_c_2=0.0;
if (tran_reset == 1) {
```

```
    old_v_tline_0_l_2 = v_tline_0_l_2;
    old_v_tline_0_c_2 = v_tline_0_c_2;
    old_i_tline_0_l_2 = i_tline_0_l_2;
    old_i_tline_0_c_2 = i_tline_0_c_2;
}
#endif
#if defined(_DERIVATE)
double v_tline_0_l_V0=1/0.333333;
double v_tline_0_c_V0=1*0.333333;
double i_tline_0_l_V0;
double i_tline_0_c_V0;
double v_tline_0_l_V1=1/0.333333;
double v_tline_0_c_V1=1*0.333333;
double i_tline_0_l_V1;
double i_tline_0_c_V1;
double v_tline_0_l_V2=1/0.333333;
double v_tline_0_c_V2=1*0.333333;
double i_tline_0_l_V2;
double i_tline_0_c_V2;
#endif
#if defined(_STATIC)
v_tline_0_l_0=(BP(b,tline_node_0_0))/0.333333;
v_tline_0_c_0=(BP(tline_node_0_0,a))*0.333333;
i_tline_0_l_0=0.5*global_step*(v_tline_0_l_0+old_v_tline_0_l_0)                +
old_i_tline_0_l_0;
i_tline_0_c_0=2.0*(v_tline_0_c_0-old_v_tline_0_c_0)/global_step                -
old_i_tline_0_c_0;
#if defined(_DERIVATE)
i_tline_0_l_V0=0.5*global_step*(v_tline_0_l_V0);
i_tline_0_c_V0=2.0*v_tline_0_c_V0/global_step;
#endif
_load_static_residual2(b,tline_node_0_0,i_tline_0_l_0)
_load_static_jacobian4(b,tline_node_0_0,b,tline_node_0_0,i_tline_0_l_V0)
_load_static_residual2(tline_node_0_0,a,i_tline_0_c_0)
_load_static_jacobian4(tline_node_0_0,a,tline_node_0_0,a,i_tline_0_c_V0)
v_tline_0_l_1=(BP(tline_node_0_0,tline_node_0_1))/0.333333;
v_tline_0_c_1=(BP(tline_node_0_1,a))*0.333333;
i_tline_0_l_1=0.5*global_step*(v_tline_0_l_1+old_v_tline_0_l_1)                +
old_i_tline_0_l_1;
i_tline_0_c_1=2.0*(v_tline_0_c_1-old_v_tline_0_c_1)/global_step                -
old_i_tline_0_c_1;
#if defined(_DERIVATE)
i_tline_0_l_V1=0.5*global_step*(v_tline_0_l_V1);
i_tline_0_c_V1=2.0*v_tline_0_c_V1/global_step;
```

```
#endif
_load_static_residual2(tline_node_0_0,tline_node_0_1,i_tline_0_l_1)
_load_static_jacobian4(tline_node_0_0,tline_node_0_1,tline_node_0_0,tline_nod
e_0_1,i_tline_0_l_V1)
_load_static_residual2(tline_node_0_1,a,i_tline_0_c_1)
_load_static_jacobian4(tline_node_0_1,a,tline_node_0_1,a,i_tline_0_c_V1)
v_tline_0_l_2=(BP(tline_node_0_1,d))/0.333333;
v_tline_0_c_2=(BP(d,a))*0.333333;
i_tline_0_l_2=0.5*global_step*(v_tline_0_l_2+old_v_tline_0_l_2)          +
old_i_tline_0_l_2;
i_tline_0_c_2=2.0*(v_tline_0_c_2-old_v_tline_0_c_2)/global_step          -
old_i_tline_0_c_2;
#if defined(_DERIVATE)
i_tline_0_l_V2=0.5*global_step*(v_tline_0_l_V2);
i_tline_0_c_V2=2.0*v_tline_0_c_V2/global_step;
#endif
_load_static_residual2(tline_node_0_1,d,i_tline_0_l_2)
_load_static_jacobian4(tline_node_0_1,d,tline_node_0_1,d,i_tline_0_l_V2)
_load_static_residual2(d,a,i_tline_0_c_2)
_load_static_jacobian4(d,a,d,a,i_tline_0_c_V2)
#endif
```

附录 III CAMC 使用手册

CAMC 的目录结构为

VP_____		前端根目录
|_____	vcLex.l	词法分析
|_____	vcYacc.y	语法分析
|_____	abytree.h	抽象数据结构
|_____	table.h, table.c	符号表
|_____	Makefile	编译配置文件
CP_____		后端根目录
|_____	printtree.h, printtree.c	打印抽象语法树
|_____	printh.h, printh.c	头文件生成
|_____	printc.h, printc.c	接口文件生成
|_____	printevaluate.h, printevaluate.c	Jacobi 矩阵生成
|_____	Makefile	编译配置文件

在 CP 和 VP 目录下分别运行 make，就可以编译出 camc 的可执行文件 vm.exe。

vm.exe 的使用方法为：

vm.exe <modulename>

其中，<modulename>为包含模型的 Verilog-AMS 文件。这一步的输出经过 gcc 编译以后，就是可被 zspice 加载的模型动态库。

在 camc 的 TEST 目录下，包含了测试所需的模型文件、网表和 Makefile，可以简单使用 make 得到输出结果。

参考文献

[1] Ken Kundert, “Automatic Model Compilation – An Idea Whose Time Has Come”, The Designer’s Guide,pp. 1310-1315, vol. 2, 2002

[2] Laurent Lemaitre, Colin Mcandrew and Steve Hamm, “ADMS – Automatic Device Model Synthesizer” IEEE Custom Integrated Circuits Conference, pp. 27-30, 2002

[3] Bo Wan, Bo P. Hu, Lili Zhou, and C.-J. Richard Shi, “MCAST: An Abstract-Syntax-Tree based Model Compiler for Circuit Simulation”, IEEE Custom Integrated Circuits Conference, pp. 249-252, 2003

[4] R. V. H. Booth, Agere Systems, “An Extensible Compact Model Description Language and Compiler”, IEEE International Workshop on Behavioral Modeling and Simulation, pp. 39-44, 2001

[5] Albert Davis, “The Gnucap Model Compiler”, IEEE International Workshop on Behavioral Modeling and Simulation, pp. 112 -117, 2002

[6] Vivek Chaudhary, Matt Francis, Wei Zheng, Alan Mantooth, Laurent Lemaitre, “Automatic Generation of Compact Semiconductor Device Models using Paragon and ADMS”, IEEE International Workshop on Behavioral Modeling and Simulation, pp. 107 -112, 2004

[7] Bo Wan, Phd dissertation, University of Washington

[8] Verilog-AMS Language Reference Manual, Open Verilog International, 1999， http://www.verilog.org/verilog-ams/htmlpages/public-docs/lrm/2.0/AMS-LRM-2-0.pdf

[9] Bing J. Sheu, Donald L. Scharfetter, Ping-Keung Ko and Min-Chie Jeng, “BSIM: Berkeley Short-Channel IGFET Model for MOS Transistors”, IEEE Journal of Solid-State Circuits, pp. 558-566, vol. 22, issue 4, 1987

[10]Franklin H. Branin, JR., “Computer Methods of Network Analysis”, Proceedings of the IEEE, vol. 55, issue 11, 1967

[11] Chung-Wen Ho, Albert E. Ruehli and Pierce A. Brennan, “The Modified Nodal Approach to Network Analysis”, IEEE Trans. On Circuits and Systems, pp. 504-509, vol. 22, issue 6, 1975

[12] John R.Levine， Tony Mason and Doug Brown, “LEX & YACC”, O’Reilly, 1992

[13] Adms, http://mot-adms.sourceforge.net

[14] Zspice, http://mot-zspice.sourceforge.net

[15] Jr. F. H. Branin, “Transient analysis of lossless transmission lines”, Proceedings of the IEEE, vol.55, issue. 11, 1967

[16] J. S. Roychowdhury, D. O. Pederson, “Efficient transient simulation of lossy interconnect”, Design Automation Conference, 1991

[17] M. Celik, A. C. Cangellaris, A. Yaghnour, “An all-purpose transmission-line model for interconnect simulation in SPICE”, IEEE transactions on Microwave Theory and Techniques, vol. 45, issue. 10, pp 1857-1867, 1997

[18] A. R. Djordjevic, T. K. Sarkar and R. F. Harrington. “Time-domain response of multiconductor transmission lines”, Proc. IEEE, vol. 75, issue 6, pp 743-764, 1987

[19] 郭裕顺， “传输线瞬态分析中基于电报方程时-空离散的有效方法”，电子学报，vol. 29, pp 373-377, 2001

[20] 李庆扬，王能超，易大义，《数值分析》

致谢

在完成论文之际，首先向给与我关心和教导的付宇卓教授、施国勇教授致以衷心的感谢。感谢他们在学业上对我的严格要求，在生活中给予我的言传身教。他们渊博的知识和研究求实的治学态度将使我终生受益。同时，我要感谢邓立群和陈微微同学，感谢他们在共同的学习研究中对我的帮助。特此致谢！

攻读硕士期间已发表或已录用的论文

[1] 洪杰，施国勇，“ 模型编译器的研究与应用 ”，微电子学与计算机，2007