

**硕 士 学 位 论 文**

**题 目： C语言实现基于V2V编码的符号分组熵编码**

**英文题目： C language implements symbol group entropy coding based on V2V coding**

**姓 名** **李旸** **学 号** **151809055**

**所在学院** **工学院** **导师姓名** **林建宇**

**专 业** **电子与通信工程**

**入学日期** **2018年9月** **答辩日期** **2021年7月**

**汕头大学学位论文原创性及学位论文使用授权声明**

**学位论文原创性声明**

本论文是我个人在导师指导下进行的工作研究及取得的研究成果。论文中除了特别加以标注和致谢的地方外，不包含其他人或其它机构已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在论文中以明确方式标明。本人完全意识到本声明的法律责任由本人承担。

作者签名： 日期： 年 月 日

**学位论文使用授权声明**

本人授权汕头大学保存本学位论文的电子和纸质文档，允许论文被查阅和借阅；学校可将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或其它复制手段保存和汇编论文；学校可以向国家有关部门或机构送交论文并授权其保存、借阅或上网公布本学位论文的全部或部分内容。对于保密的论文，按照保密的有关规定和程序处理。

作者签名： 导师签名：

日期： 年 月 日 日期： 年 月 日

# 摘要

近年来，视频编解码技术得到广泛应用，并且飞速发展。随着5G的时代到来，对于视频的清晰度的要求越来越高。不可避免的是视频清晰度越高，整个视频的数据量就越大。视频编解码技术一般包括下面几个部分：变换、量化、熵编码。其中熵编码是对于整个视频能否压缩到理想熵值起到至关重要的作用。

在熵编码技术发展到今天，目前发展的趋势仍然是大多数熵编码基于算数编码原理进行发展的。因为算数编码的理论可以无限接近于最佳编码效率。可是，由于算数编码在编码过程中要不断的更新概率，在计算概率的时候要涉及到大量的计算。从而增加了编码的时间，这对于软编码来说这是不可接受的。而实际使用的是二进制的算数编码算法，是使用逼近的方法来减少计算的复杂度，但不可避免的代价就是无法达到精准算数编码的压缩效率。

本文提出的熵编码方法是基于V2V编码的符号分组熵编码方法。并且用C语言实现，提高了编码的速度，达到了理想的压缩效率。对于独立同分布(iid)信源进行编码时,使用的是V2V编码结合改进的Golomb-Rice编码，是将概率区间划分成7个，每个子区间对应V2V编码构成的码表，改进的码表代替块编码的码表。使用块编码的熵编码方法的冗余率在1％左右，此熵编码方法冗余率大约保持在0.5％，取得了理想的压缩的效果。由于视频压缩技术中是将量化之后的系数进行熵编码，但是这些系数存在相关性，而V2V编码结合改进的Golomb-Rice编码只适用于独立同分布信源。因此，特意使用符号分组法处理量化系数，将其分为长游程组、短游程组、分组记录组，分组后的数组趋近于独立分布，然后在对三个组分别使用V2V编码结合改进的Golomb-Rice编码，从而可以应用到非独立的（non-iid）信源，使得本文提出的熵编码方法可以应用到视频编码中，提升视频的压缩效率。

**关键词：**视频压缩；熵编码；V2V编码；符号分组方法；

# Abstract

In recent years, video codec technology has been widely used and developed rapidly. With the advent of the 5G era, the requirements for the definition of video are getting higher and higher. Inevitably, the higher the video resolution, the larger the data volume of the entire video. Video coding and decoding technology generally includes the following parts: transformation, quantization, and entropy coding. Among them, entropy coding plays a vital role in whether the entire video can be compressed to the ideal entropy value.

In the development of entropy coding technology to today, the current development trend is still that most entropy coding is developed based on the principle of arithmetic coding. Because the theory of arithmetic coding can be infinitely close to the optimal coding efficiency. However, because the arithmetic coding has to constantly update the probability during the coding process, a lot of calculations are involved when calculating the probability. This increases the coding time, which is unacceptable for soft coding. The actual use is a binary arithmetic coding algorithm, which uses an approximation method to reduce the complexity of the calculation, but the inevitable cost is that the compression efficiency of precise arithmetic coding cannot be achieved.

The entropy coding method proposed in this paper is a symbol group entropy coding method based on V2V coding. And it is implemented in C language, which improves the coding speed and achieves the ideal compression efficiency. When coding independent and identically distributed (iid) sources, V2V coding combined with improved Golomb-Rice coding is used, which divides the probability interval into 7, and each sub-interval corresponds to a code table composed of V2V coding, an improved code table Instead of block coding code table. The redundancy rate of the entropy coding method using block coding is about 1%, and the redundancy rate of this entropy coding method is maintained at about 0.5%, achieving an ideal compression effect. Since the quantized coefficients are subjected to entropy coding in the video compression technology, these coefficients are correlated, and the V2V coding combined with the improved Golomb-Rice coding is only suitable for independent and identically distributed sources. Therefore, we deliberately use the symbol grouping method to process the quantized coefficients and divide them into long-run groups, short-run groups, and grouped record groups. The grouped arrays tend to be distributed independently, and then use V2V coding for the three groups to combine the improved Golomb-Rice coding can be applied to non-iid sources, so that the entropy coding method proposed in this article can be applied to video coding to improve video compression efficiency.

**Keywords:** Video compression, Entropy coding, V2V encoding, Symbol grouping method；

# 目 录

[摘要 I](#_Toc45663150)

[Abstract II](#_Toc45663151)

[目 录 III](#_Toc45663152)

[第1章 绪论 1](#_Toc45663153)

[1.1 集成电路发展背景和历史 1](#_Toc45663154)

[1.2 集成电路测试 2](#_Toc45663155)

[1.3 故障模拟的研究意义 3](#_Toc45663156)

[1.4 基于多核CPU故障模拟研究现状 5](#_Toc45663157)

[1.5 多核CPU与编程语言 6](#_Toc45663158)

[1.6 论文的主要工作 7](#_Toc45663159)

[1.7 论文结构 7](#_Toc45663160)

[第2章 故障模拟介绍 8](#_Toc45663161)

[2.1 引言 8](#_Toc45663162)

[2.2 故障模型（Fault Model） 8](#_Toc45663163)

[2.2.1 缺陷与故障模型 8](#_Toc45663164)

[2.2.2 故障模型的选择 10](#_Toc45663165)

[2.3 故障激活（Fault Sensitization）与故障传播（Fault Propagation） 10](#_Toc45663166)

[2.4 逻辑模拟（Logic Simulation）与算法 11](#_Toc45663167)

[2.5 故障压缩（Fault Collapsing） 11](#_Toc45663168)

[2.5.1 等价故障（Equivalent Fault） 11](#_Toc45663169)

[2.5.2 基于结构性等价的故障压缩 12](#_Toc45663170)

[2.6 故障模拟算法（Fault Simulation Algorithm） 13](#_Toc45663171)

[2.7 故障覆盖率（Fault Coverage） 15](#_Toc45663172)

[2.8 本章小结 15](#_Toc45663173)

[第3章 故障模拟模型设计 16](#_Toc45663174)

[3.1 引言 16](#_Toc45663175)

[3.2 故障模拟系统模型 16](#_Toc45663176)

[3.3 电路元件模型 17](#_Toc45663177)

[3.3.1 组合电路元件模型 17](#_Toc45663178)

[3.3.2 时序电路元件模型 18](#_Toc45663179)

[3.4 逻辑门节点值 21](#_Toc45663180)

[3.4.1 位并行 21](#_Toc45663181)

[3.4.2 四值逻辑 21](#_Toc45663182)

[3.5 电路的网表结构 22](#_Toc45663183)

[3.6 电路层级化处理 23](#_Toc45663184)

[3.7 本章小结 24](#_Toc45663185)

[第4章 基于单核CPU的故障模拟设计 25](#_Toc45663186)

[4.1 引言 25](#_Toc45663187)

[4.2 故障集生成 25](#_Toc45663188)

[4.2.1 基于电路结构的故障生成 25](#_Toc45663189)

[4.2.2 读取故障文件生成故障 29](#_Toc45663190)

[4.3 测试向量生成 29](#_Toc45663191)

[4.4 故障模拟器设计 29](#_Toc45663192)

[4.5 本章小结 33](#_Toc45663193)

[第5章 基于多核CPU的故障模拟设计 34](#_Toc45663194)

[5.1 引言 34](#_Toc45663195)

[5.2 OpenCL平台 34](#_Toc45663196)

[5.2.1 OpenCL编程流程 34](#_Toc45663197)

[5.2.2 平台模型 34](#_Toc45663198)

[5.2.3 执行模型 36](#_Toc45663199)

[5.2.4 内存模型 37](#_Toc45663200)

[5.3 基于多核CPU的数据结构改进 38](#_Toc45663201)

[5.4 基于多核CPU的故障模拟器设计 40](#_Toc45663202)

[5.5 本章小结 42](#_Toc45663203)

[第6章 实验及结果 43](#_Toc45663204)

[6.1 引言 43](#_Toc45663205)

[6.2 实验平台 43](#_Toc45663206)

[6.3 实验结果与分析 43](#_Toc45663207)

[6.3.1 故障生成与故障压缩验证 43](#_Toc45663208)

[6.3.2 可检测到故障正确性验证 44](#_Toc45663209)

[6.3.3 故障模拟结果 45](#_Toc45663210)

[6.4 本章小结 50](#_Toc45663211)

[第7章 总结与展望 51](#_Toc45663212)

[7.1 研究工作总结 51](#_Toc45663213)

[7.2 论文创新点 51](#_Toc45663214)

[7.3 下一步研究计划 51](#_Toc45663215)

[参考文献 52](#_Toc45663216)

[攻读学位期间主要研究成果 56](#_Toc45663217)

[致谢 57](#_Toc45663218)

[本文专业术语中英文对照表 58](#_Toc45663219)

# 第1章 绪论

## 1.1 熵编码发展背景和历史

1948年，Shannon发现了信息熵理论，并当时提出了一种简单的编码方法，即Shannon编码。Shannon编码理论是将信源符号依照其出现的概率进行降序排列，用符号序列累计概率的二进制作为对信源的编码，且理论上证明它的优越性。1952年，R.M.Fano再次提出了Fano编码。这一些早期的编码方法揭示基本的规律，并且取得了一定的压缩效果。对真正实用的压缩算法有着借鉴作用。

1952年，第一个真正实用的熵编码方法是由D.A.Huffman提出。年轻的Huffman发现这个编码也是纯属偶然，原因是为了不想参加某门功课的期末考试，才设计了这个看似简单，却有着深远意义的编码的方法。直到如今，Huffman编码也被广泛应用和讨论。Huffman编码在计算机的数据结构经常被提及。Huffman的编码效率比较高，编码运行速度快，同时实现的方式灵活，从1960年以后，Huffman编码在数据压缩领域开始得到广泛应用。例如:最初的UNIX系统上不被人注意的压缩程序COMPACT实际上就是Hffman编码的应用。1960年Peter Elias发现了不需要排序，只要编解码端使用相同的符号顺序即可，从而提出了算数的编码的概念。当时Peter Elias并没有立即公布算术编码的发现，由于当时算术编码是即使在数学理论上是成立的，但是不可能在现实条件中实现的。直到1976年，R.Pasco和J.Rissanen分别用定长的寄存器实现了有限精度的算术编码。到了1980年后，Huffman编码又在CP/M和DOS系统开始使用，其中代表程序叫SQ。今天的许多的著名的压缩工具和压缩经典算法，如WINRAR、GZIP和JPEG,都在使用Huffman编码。Huffman的缺点就是其编码所输出的编码长度只是对信息熵的计算结果的一种大概的近似，不能真正近似到信息熵的极限。因此，现在的压缩技术通常只将Huffman编码当做最终的编码手段,而非数据压缩算法的全部。

科学家们一直停止向信息熵极限挑战的脚步，1968年左右，P.Elias在Shannon和Fano的编码方法的基础上进行了延伸，构造出了从数学理论上看来更完善的Shannon-Fano-Elias编码。在延续这一编码方法的思路，1979年，G.G.Langdon将算术编码整合起来，并于1981年实现了算术编码的二进制的编码。直到发展到现在的基于上下文建模的算术编码，如JEPG2000的MQ编码算法和视频压缩标准的H264的CABAC编码算法，都达到了很好的压缩的效率。

## 1.2 研究目的与意义

长期以来，在我们的平常的生活中，通常可以通过图像或者是视频来认识客观世界，从而获得快速地我们所需要的信息，所以说图像或者是视频在我们如今的日常经济生活中不可或缺的载体。特别是近年来通信技术的高速发展，随之而来的要求就是对视频/图像的分辨率越来越高，带来的问题就是需要传输的数据量越来越大。例如现实生活比较的常见的1080p格式的，帧率为15,24位色的两小时的视频需要625G的存储空间。

可想而知，如果不使用压缩技术，直接将图像/视频不经过处理直接进行传输，那么会大量带宽，影响人们之间正常的信息的交流。同样的道理，如果直接进行存储，将会极大浪费计算机的存储资源。所以，视频/图像的压缩技术的诞生的目的就是节省传输的带宽和减少数据的存储空间。不言而喻，在这种巨大的技术需求下，图像和视频的压缩技术的开始蓬勃发展，如现在的音视频聊天和音视频会议都使用了视频的压缩的技术，才有了现在的在线教育，在线看病等领域的大发展。

熵编码是现代视频压缩技术的重要组成部分，它对整个数据量的压缩至关重要。所以自然而然成为压缩技术领域重点研究的一环。当前，熵编码的算法主流趋势是算术编码，例如：基于上下文建模的JEPG2000的MQ的编解码器和HEVC的CABAC编解码器。这种算法需要通过大量建模，才能达到良好的压缩效果，所以算术编码的复杂度比较高。本文提出来的熵编码的算法属于边长编码，与算术编码不同，其最大的有点就是算法复杂度相对简单，编解码速率较快，实用性较强，同时也达到了良好的压缩效果。

## 1.3 本文章节安排

第一章介绍了熵编码的发展背景和历史，并且详细解绍了熵编码的发展的历史的过程和熵编码的应用，以及目前主流的熵编码的算法。同时阐述了本文研究的目的和意义。

第二章主要是分为两个部分，一部分是此文研究的背景，第二部分是一些前期的工作。首先研究背景主要是介绍

第三章是算法的改进和实现，

第六章是对本文工作的进行总结和展望，

# 第2章 研究背景与前景工作

## 2.1 研究背景

### 2.1.1图像的基本概念

广义上来说，图像其实就是所有具有视觉效果的画面，它包括纸质介质上的、底片的或者照片上的、电视、投影仪或计算机屏幕上的。同时，图像视觉系统的基础，是物体世界的客观反映。我们作为人类可以通过身体生的感觉器官从外部世界获取各种各样的信息，其中，通过人眼获取的信息占比最大。通过相关调查可知，视觉器官在人获取的信息中占比75％，可见图像对于我们来说是多么的重要。图像是人们根据客观事物制作而成的，可以通过扫描、照相、摄像得到，也可以通过绘制得到。

图像可分为数字图像和模拟图像。模拟图像是指在图像处理中，像纸质照片、电视模拟图像等，这种通过某种物理量（如光、点等）的强弱变化来记录图像亮度信息的图像，其特点是物理量的变化是连续的。数字图像是指用一个数字阵列来表达客观物体的图像，是一个离散采样点的集合，每个点具有其各自的属性，其特点是它把连续的模拟图像离散化成规则网格，并用计算机以数字的方式来记录图像上的各网格点的亮度信息的图像。简而言之，一切你能肉眼看见的，都是模拟图像。而数字图像，本质上是一个存储数字的矩阵，是一团数据，你肉眼能看见的，都是模拟图像。而数字图像，本质上就是一个存储数字的矩阵，是一团数据，你肉眼直接看不见。

而在实际的应用中，为了方便对图像进行处理，需特定的方法将模拟图像转化为数字图像。数字图像是由像素组成的二维矩阵，对于灰色的图像，每个像素有8位比特位，故像素值的取值范围是在0到255之间，0为黑色，255为白色，它的数值变化范围即是从黑色到白色的渐变值。对于彩色的图像而言，大致有四种颜色模型：RGB模型、CYM模型、HIS模型、YUV模型，例如最常见的RGB模型，每个像素的都有24个比特位，即三个字节，由三个样本值组成的，为红、绿、蓝三种原色，彩色图像就是这样由这三个样本值按照特定的比例去组成的。

图像可以根据运动状态分为两大类：静止图像和运动图像。运动的图像的原理其实就是将一组静止的图像组合而成的，也就是我们常常遇见的视频。视频是指将一组图像按照时间的时序轴，进行组合。一个组中的每一张图像我们称为视频的每一帧。视频编码是由静止图像的编码发展过来的，静止图像的编码是其基础。

### 2.1.2C语言优势

C语言是一门面向过程的计算机编程语言，跟Java和Python等面向对象编程语言有所不同。C语言的设计目的是提供一种能以简易的的方法编译、处理低级的处理器、仅仅产生少量的机器码以及不需要任何运行环境支持的编程语言。C语言描述问题比汇编语言迅速、工作量小、可读性好、易于调试、修改和移植，而代码质量与汇编语言相当。C语言一般只比汇编语言代码生成的目标程序效率低于10％-20％。

因此，C语言有其优异的性能，所以本文提出的算法实现是由C语言实现，其编解码速度比MATLAB的编码速度更快，并且达到理论所要求的压缩的效果。

### 2.1.2图像压缩的基本原理

一张原始的图像是长是1920，宽是1080，如果每个像素都由32bit表示(RGBA)，那么图像需要的的内存储存空间的大小是大约为8M。这是万万不可接受的。同样的情况，如果是一个视频，1080p的，帧率为30fps,时长为一个小时，那么如果压缩的大概需要的内存是800多个G。所以说，我们需要图像压缩。图像之所以能够压缩，因为它有很多冗余信息。常见的图像、视频、音频数据中都存在的这几种冗余类型：空间冗余、时间冗余和视觉冗余。

空间冗余是指一幅图像表面上的各采样的颜色之间往往存在着空间的连贯性，比如一张图片的局部一部分颜色几乎是相同的，这些颜色相同的局部就可以压缩。例如，第一行像素基本都一样，假设亮度值Y是这样存的，[105 105 105 105 105……105],如果共100个像素，那需要100个字节。最简单的压缩就是：[105 100]，表示接下来100个像素的亮度都是105，那么只要2个字节，就能表示整行的数据了，这就达到压缩的效果了。空间冗余主要发生在单张图片，比如我们日常生活中的照片。

时间冗余主要是发生在视频中，运动图像一般为位于一时间轴区间的一组连续画面，其中的相邻帧往往包含相同的背景和移动物体，只不过移动物体所在的空间位置略有不同，所以后一帧的数据与前一帧的数据有许多共同的地方，这种共同性是由于相邻的帧记录了相邻的同一场景画面，所以称为时间冗余。

视觉冗余是指人类的视觉系统由于受生理特性的限制，对于图像场的注意是非均匀的，人对细微的颜色差异感觉不明显。例如，人类的视觉的一般分辨能力为26灰度等级，而一般的图像的量化采用的是28灰度等级，即存在视觉冗余。人类的听觉是对某些信号反映不太敏感，使得压缩后再还原允许范围的变化，人也感觉不出来。

### 2.1.3图像压缩的基本过程

图像压缩是指以较少的的比特有损或者无损的表示原来的像素矩阵的技术，其目的是减少表示数字图像时需要的数据量。那么图像压缩的基本过程主要分为三大部分：变换部分、量化部分、熵编码部分。

第一部分是图像压缩中的变换编码。其本质就是数学的变换，变换编码的作用是将空间域描述的图像信号变换到频率域，然后对变换后的系数进行编码处理。一般来说，图像在空间上具有较强的相关性，变换到频率域可以实现去相关和能量集中。常用的正交变换有离散傅里叶变换、离散余弦变换等等。数字视频压缩的过程中应用广泛的是离散余弦变换。

离散傅里叶变换是傅里叶变换在时限和频域上都呈离散的形式，将信号的时域采样变换为其DTFT的频域采样。在形式上，变换两端（时域和频域上）的序列是有限的长，而实际上这两组的序列都应当认为是离散周期信号的主值序列。即使对有限的长的离散信号作DFT,也应当将其看做周期延拓的变换。在实际应用中通常采用快速傅里叶变换的计算DFT.

离散余弦变换是与傅里叶变换相关的一种的变换，类似于离散的傅里叶变换，但是只使用实数。离散余弦变化相当于一个长度大概是它的两倍的离散傅里叶变换，这个离散傅里叶变换是对一个实偶函数进行的(因为一个实偶函数的傅里叶变换仍然是一个实偶函数)，在有些变形里面需要将输入或者输出的位置移动半个单位（DCT有8种标准类型，其中4种是常见的）。离散余弦变换经常被信号处理和图像处理使用，用于对信号和图像（包括静止图像和压缩图像）进行有损数据压缩。这是由于离散余弦变换具有很强的“能量集中特性”：大多数的自然信号（包括声音和图像）的能量都集中在离散余弦变换后的低频部分，而且当信号具有接近马尔可夫过程的统计特性时，离散余弦变换的去相关性接近于K-L变换的性能。例如，在静止图像编码标准JEPG中，在运动图像编码标准MJPEG和MPEG的各个标准中都使用了离散余弦变换。在这些标准中都使用了离散余弦变换。

第二部分是图像压缩中的量化编码，图像压缩中的量化就是用更小的集合表示更大的集合过程，例如0~10之间的数，我们可以把小于5的数记为0，大于或等于5的数记为1，即用{0,1}这个小集合表示了0到10之间的数。量化其实就是就是一中对信号源的有限近似，并且它在图像压缩中是一种有损过程，基本上所有的量化操作都是有损的，会产生量化误差，除非量化的系数是整数，并且步长是1。

为何要在图像压缩中要使用量化，主要原因如下：一是由于人眼的视觉特性中，人眼的分辨能力有一定的限度，我们可以通过一定的量化来减小变换后系数的精确度，这样可以使人们对图像的主观评价并不降低，但是可以使图像得到压缩，我们需要根据不同的图像的各种的特征、人们的主观要求等因素来设计量化的步长。二是可以把频繁出现的系数集合量化成码字较小的符号，这样使得压缩率更高。

图像压缩中的量化如果按照量化维数来分，可以量化分为标量量化和矢量量化。标量量化是对需要量化的系数一个一个进行量化，将需要量化的系数取值范围被分为若干的小区间，每个小区间有一个代表值，量化时小区间的信号值就用这个代表值代替，或者叫被量化为这个代表值，这时的信号量是一维的。而矢量量化，就是将需要的量化的系数组成一个矢量，矢量量化是对矢量进行量化，和标量量化一样，它把矢量空间分成若干个小区域，每个小区域寻找一个代表的矢量，量化时落入小区域的矢量就用这个矢量代替，或者叫做被量化为这个代表矢量。矢量量化就是标量量化的多位扩展，标量量化是矢量量化的特殊情况。

图像压缩中的量化按照量化级的划分可分为均匀量化和非均匀量化，在图像压缩中将需要量化的系数的取值范围平均分成多少等分，步长都是一样的，这种量化叫做均匀量化。对于像素灰度值在黑-白范围内较均匀分布的图像，这种量化方法可以得到较小的量化误差。非均匀量化是依据一幅图像具体的灰度值分布的概率密度函数，按总的量化误差最小的原则进行的量化。具体做法是对图像中的像素灰度值分布频繁出现的灰度值范围，量化间隔取得小一些，而对于那些像素灰度值较少出现的范围，则量化的间隔大一些。这种量化方法可以较少丢失图像信息。由于灰度值的概率分布因图像的不同而不同，所以不可能找到一个适合各种图像的最佳非均匀量化方案，实际上一般都采用均匀量化。

第三部分是图像压缩中的熵编码。熵编码是一种独立于介质的具体特征的进行无损数据压缩的方案。一种主要类型的熵编码方式是对输入的每一个符号，创建并分配一个唯一的前缀码，然后，通过将每个固定长度的输入符号替换成相应的可变长度前缀无关的输出码字替换，从而达到数据压缩的目的。每个码字的长度近似于概率的负对数成正比例。因此，最常见的符号使用最短的码。根据香农的信源编码原理，一个符号的最佳码长是，其中b是用来输出的码的数目，p是输入符号出现概率。霍夫曼编码和算术编码是两种常见的熵编码技术。如果预先已知数据流的的近似熵特性(尤其是对于压缩信号)，可以使用简单的静态码。这些静态码，包括通用密码（如斐波那契编码）和哥伦布编码（如元编码或Rice编码）。

现在的熵编码技术大多都是结合了上下文建模的方法，这样了编码的过程中，可以利用上下文信息，对信源选择更为合适的码表进行编码，从而提高编码的效率。在编码过程中，利用上下文信息，可以更新信源的符号的条件概率，达到自适应的效果。这种结合上下文的建模方法挖掘了高阶条件熵，使得平均码长能够接近理论的标准熵。但是丰富的上下文，会带来计算的复杂度，减慢编码的速度。

## 2.2 前期工作

### 2.2.1 基于游程的Golomb编码

本文的前期工作使用的熵编码方法是基于游程的熵编码，它是一种无损的熵编码。普通的游程编码是基于使用变动的长度的码来取代连续重复出现的原始资料来实现。距离来说，一组符号串为“AAAABBBCCDEEE”，由4个A、3个B、2个C、1个D、4个E组成，经过变动长度编码方法可将此符号串压缩为4A3B2C1D4E，由14个需要编码单位转成10个编码单位。简言之，普通的游程编码的优点在于将重复性高的符号串压缩成小的单位；然而，其缺点在于一但该符号串中的符号出现的频率不高，就可能导致压缩的所需的编码长度大于原始符号串所需的编码的长度。例如：符号串“ABCDE”，压缩结果为“1A1B1C1D1E”，编码所需的长度编程符号串不压缩的长度的两倍。

基于游程的Golomb编码是它针对符号只有类型，即0和1进行编码。此方法是由Golomb编码演变而来。对于给定的一个idd的二进制信号源“0”和“1”的序列。我们先定义p为

1. 单固定型故障（single stuck-at fault）模型

单固定型故障模型是最常见的故障模型，最早由Poage提出，指任何时候电路中只存在一条信号线只取逻辑值1或0，用来对应表示节点与电源直接连接或者接地的缺陷，该模型由于简易和代表了实际中最常见的缺陷，被广泛应用于学术研究。

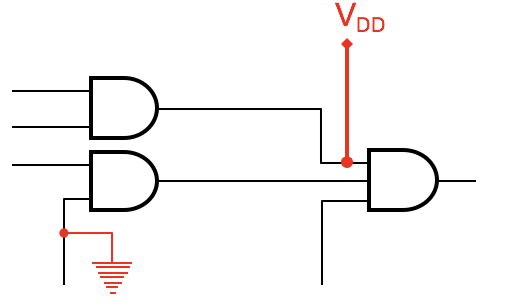


图2-1 单固定型故障

1. 桥接故障（bridging fault）模型

桥接故障模型是CMOS电路中最常见的故障类型，它是指电路节点之间允许短路，根据具体情况又可以分为输入型桥接，即同一个门的两个或多个输入端发生短路；其他情况属于非输入型桥接。桥接故障可能在电路中产生反馈（feedback），从而将组合电路转换成时序电路。

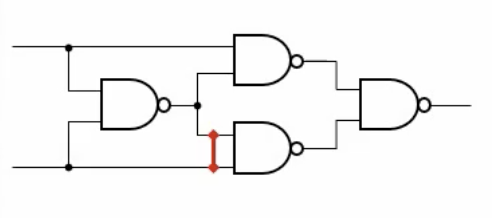


图2-2 桥接故障

1. 延时故障（delay fault）模型

延时故障模型又分为门延时故障和路径延时故障，门延时故障指单个门的输入或输出跳变（从0转换到1或从1转换到0）时行为缓慢，一般影响范围较小；而路径延时故障是由于电路中大范围的参数跳变，严重时可导致某些信号无法沿传播路径到达电路主输出从而被检测到，由于延迟的累积性，即使很小的门延时故障通过累积也可能导致不可接受的路径延时故障。

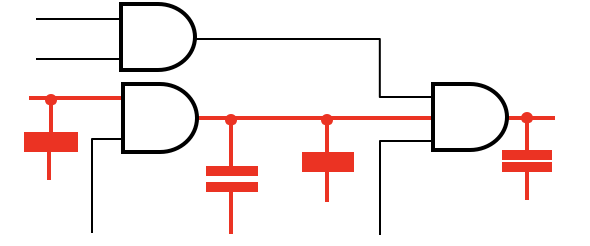


图2-3 延时故障

除了以上三种常见的故障外，还有多重固定型故障、转换故障等，因本文不涉及所以这里不再赘述。

### 2.2.2 故障模型的选择

本文研究选择单固定型故障，主要基于以下几点原因。一是根据实际工业生产测试经验所显示，在单固定型故障下进行的故障模拟如果表现出较高的故障覆盖率那么对其他模型同时也能表现较高的缺陷覆盖率[[[1]](#endnote-2)]。二是对于逻辑电路来说，假如电路有n个节点，单固定型故障的数量则为2n对于测试来说是完全可行的，而且单固定型故障模型有独立的技术具备良好的移植性和发散型。

## 2.3 故障激活（Fault Sensitization）与故障传播（Fault Propagation）



图2-4 故障激活

故障激活和故障传播是故障检测的必要条件，如图2-4所示，当需要检测在节点a处的单固定型故障stuck-at 1时，假定门输入b=0，节点d恒为零0，即节点a的故障被屏蔽掉了，a=1，b=1时故障同样被屏蔽，只有当输入值置为a=0，b=1时，此时d的值从理论值0变成1，这种情况称为故障激活。



图2-5 故障传播

另假设图2-5是一个完整的组合电路，节点a、b、c为原始输入，g为原始输出。目标是检测节点a处的故障stuck-at 1，首先将输入a、b值置为a=0、b=1，这样故障被激活，节点d的值为d=1，当我们把输入c置为0时会发现输出g恒定为0，故障无法被检测到，此时故障只能传播到节点e处；只有当把输入c置为1时，g的值为1，与输入向量a=0、b=1、c=1下的理论值g=0不同，根据1.3节可知故障能被检测到，这种情况说明故障能传播到原输出g，而故障成功传播到主输出的路径a->d->e->g又称为敏化路径（sensitized path），因此本文进行故障模拟时判断指定故障是否能被检测到即判断该故障是否能被激活且传播到主输出[[[2]](#endnote-3)]。

## 2.4 逻辑模拟（Logic Simulation）与算法

逻辑模拟是对逻辑电路的逻辑值进行计算的过程，也是故障模拟的基础。一般来说，逻辑模拟可分为顺序逻辑模拟（In-order Logic Simulation）和事件驱动逻辑模拟（Event-driven Logic Simulation）。顺序逻辑模拟是按照门的顺序依次模拟，对于每更新一组输入向量激励，整个电路的门又会被重新模拟一次。而事件驱动逻辑模拟的核心思想是事件驱动算法，即当某种改变产生事件时才会驱动相应的步骤进行处理，它也在其他领域被广泛应用[[[3]](#endnote-4)]。对于逻辑电路来说，事件驱动逻辑模拟即当上一次模拟完成后电路各节点的值会被保留下来，当下一次模拟时针对某个节点的值改变判断是否触发事件来进行相应的计算。



图2-6 两与门电路

如图2-6所示是一个由两与门组成的电路，在输入向量abc为101的激励下，通过逻辑模拟，电路中各个节点的值能够被计算并保留，插入节点d的故障stuck-at 1，事件驱动被激活。其中先对电路进行逻辑模拟的过程称为无故障模拟（Fault-free Simulation），以保留无故障电路的理论值，插入故障后进行的故障模拟本质上在无故障模拟的基础上进行的事件驱动逻辑模拟。

## 2.5 故障压缩（Fault Collapsing）

### 2.5.1 等价故障（Equivalent Fault）

等价故障又分为功能性等价故障和结构性等价故障：

（1）功能性等价，用hu（x）和hv（x）分别表示在故障u和v下的电路输出响应，如果有

|  |  |  |
| --- | --- | --- |
|  |  | （2-1） |

则称故障u和v在功能上是等价的。如图2-7所示，对于两个不同节点上的stuck-at 1，只要它们的输出响应相等，说明它们在功能上等价。功能性等价将一个电路所有故障划分为功能等价类，每一个等价类只考虑一个有代表性的错误即可。值得注意的是，功能性等价的故障并不局限于相同的故障模型，比如u可以是单固定型故障，v可以是桥接故障，但是通常将等价关系的分析限制在相同类型的故障之间。

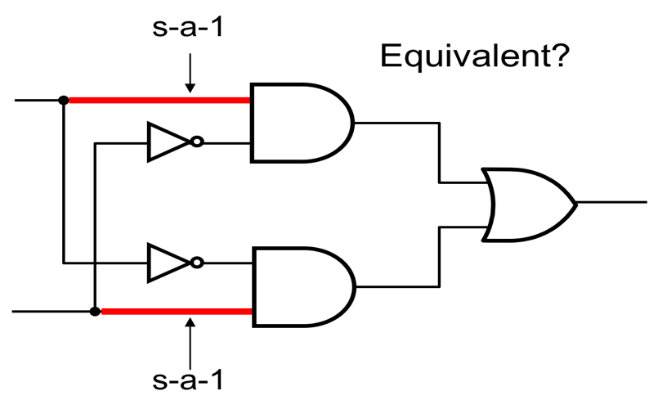


图2-7 等价故障

（2）结构性等价，Nu、Nv分别代表两个故障电路，故障为u、v，故障的存在产生了一组具有常数值的线路，除去所有这些线路(除了主要的输出)可得到一个简化的电路S(Nu)、 S(Nv)，如果满足

|  |  |  |
| --- | --- | --- |
|  |  | (2-2) |

则两个故障u和v在结构上是等价的[[[4]](#endnote-5)]。如图2-8所示，可以忽略输入向量的影响，由两个单固定性故障得到的简化电路相同，说明两个故障在结构上等价。

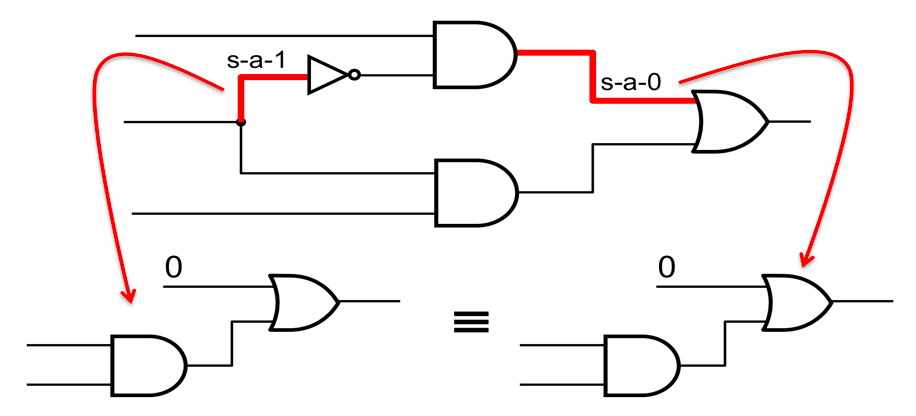


图2-8 结构性等价故障

结构性等价的故障一定是功能性等价的，反之不一定；另外功能性等价需要有输入激励经过计算，而结构性等价可以忽略输入向量直接得出。基于以上两点，在故障模拟之前我们选择利用结构性故障来进行故障压缩，故障等价性是测试生成和故障模拟中用来减少处理时间的关键步骤之一。

### 2.5.2 基于结构性等价的故障压缩

本文故障模型基于单固定型故障，对于一个指定的电路，假设电路中有n个节点（主输入、主输出、门输入、门输出），那么所有可能的单固定型故障数量为2n个，根据结构性等价的定义可以将2n个初始故障划分为w个等价故障类（Equivalent Fault Class），每个类里的所有故障都是结构性等价的，在每一个等价类选一个具有代表性的故障（一般选靠近原始输入或者靠近原始输出的），便可以得到由w个代表性故障组成真正用于故障模拟的故障集，这个过程称为故障压缩[[[5]](#endnote-6)]。由于结构性等价只能用于电路中的若干个无扇出区（Fan-Out Free Region），所以基于结构性等价的故障压缩得到的等价类不一定是最优解，尽管如此，这个过程仍然实现了初始故障列表约35%～45%的缩减[[[6]](#endnote-7)]。

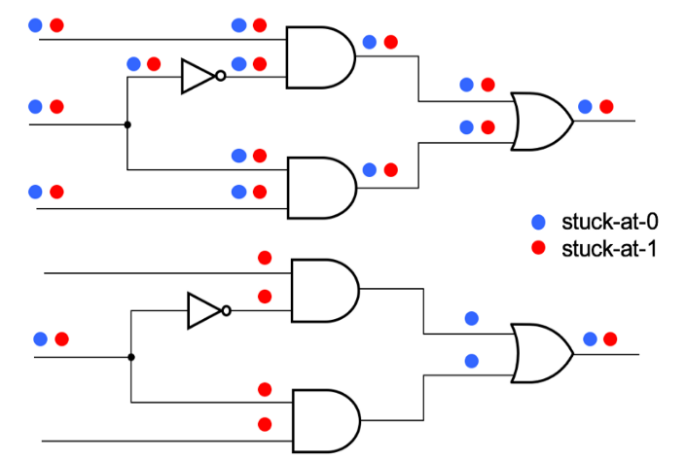


图2-10 基于结构性等价的故障压缩过程示意图

如图2-10所示是一个无扇出区（或完整电路），由原始输入、原始输出、门扇入（Fan-in）、门扇出（Fan-out）组成的所有节点，在每个节点分别插入stuck-at 0和stuck-at 1，图中蓝点表示stuck-at 0红点表示stuck-at 1，按照以下三个步骤：

1.对于没有分支的节点线上（Fan-out Free）它的原始输入或门扇出等价于门扇入；

2.对于具有决定值（如与门的决定值是输入0、或门的决定值是输入1）的扇入直接等价于相应的扇出；

3.对于反相器，扇入上的stuck-at 0（stuck-at 1）等价于扇出上的stuck-at 1（stuck-at 0）。

最终可以得到图中下方的代表性等价故障，以此用于故障模拟。

## 2.6 故障模拟算法（Fault Simulation Algorithm）

故障模拟算法有很多种，这里介绍三种常见的算法，并说明本文选择的算法：

（1）顺序故障模拟算法（In-order Fault Simulation）：

顺序故障模拟算法即先在指定测试向量集下进行无故障模拟保存输出理论值，接着从故障集引入单个故障，按序进行模拟，每模拟一个向量与输出理论值进行比较，不同的话报告检测到故障并停止模拟，重复上述过程直至故障集的每一个故障都被模拟过。

（2）并行故障模拟算法（Parallel Fault Simulation）：

并行故障模拟算法是假设电路只包含逻辑门且所有门的延迟都相同，并充分利用位并行和逻辑操作。如图2-12所示假设用来储存各节点值的数据类型长度为3个bit，分别编号Bit 0、Bit1、Bit2，从输入向量开始，所有节点的Bit0用来存储无故障电路（fault-free circuit）的节点值，所有节点的Bit1用来存储在故障c stuck-at 0下的节点值，所有节点的Bit2用来存储在故障f stuck-at 1下的节点值，每一位代表故障影响下的节点值，在当前输入向量下进行逻辑模拟，可以在输出节点g处得到3位输出逻辑值，用代表故障电路输出即节点g的Bit1、Bit2与代表无故障电路输出即节点g的Bit0相比较，可以得出故障c stuck-at 0可以被检测到，接着从向量集读取下一个输入向量，重复上述步骤直到每一位上的故障都被检测到[[[7]](#endnote-8)]。



图2-12 并行故障模拟

（3）并行向量单故障传播算法（Parallel-Pattern Single-Fault Propagation）：

并行向量单故障传播算法同样也是利用位并行和逻辑操作，与并行故障算法不同的是它是将测试向量进行并行处理。如图2-13所示，假设储存各节点值的数据类型长度为4个bit，第一步从故障集取第一个故障，然后读取4个测试向量，将向量的值分别放置在输入的4个bit上，进行无故障模拟得到输出的理论值；接着插入选定的故障，如图插入的故障为a stuck-at 0，将节点a处的每一位上的值固定为0，然后重新进行逻辑模拟可以得到故障电路输出的值，与无故障电路输出按位比较，只要有一位不同则可以判断故障a stuck-at 0可以被检测到，最后从故障集按序注入故障，重复上述步骤。需要注意的是，假如对于某个故障来说，所有的测试向量都无法检测到该故障，称之为不可检测到的故障[[[8]](#endnote-9)]。



图2-13 并行向量单故障传播

并行故障模拟和并行向量单故障传播这两种算法都依赖于计算机固有的位并行性来减少故障模拟的时间，一个是并行处理故障一个是并行测试向量。与并行故障模拟算法适合基于数据并行相比，并行向量单故障传播算法一方面适合大规模测试向量的故障模拟，另一方面更适合基于任务并行，这些特点适合于多核CPU体系架构[[[9]](#endnote-10)]。

## 2.7 故障覆盖率（Fault Coverage）

在测试中一个重要问题就是测试评估，即确定测试的有效性或质量，在故障模拟阶段测试评估通常是在指定故障模型的背景下进行的，用故障模拟可检测到的故障数量Df与假定的故障列表内的故障总数Af之间的百分比值来衡量，称为故障覆盖率Fc。

|  |  |  |
| --- | --- | --- |
|  |  | （2-3） |

## 2.8 本章小结

本章详细介绍了故障模型、故障激活、逻辑模拟、故障压缩和故障仿算法的概念，并解释了选取单固定型故障模型、采用基于结构性等价的故障压缩、选用并行向量单故障传播算法的优点以及故障覆盖率。

# 第3章 故障模拟模型设计

## 3.1 引言

本文作为学术研究，研究对象采用的是国际通用的基准电路，分别是是电路与系统国际研讨会颁布的ISCAS`89基准电路和德州大学颁布的ITC`99基准电路，这两个基准电路标准将时序电路中的D触发器元件进行了简化，忽略了时钟信号输入，而本文中需要考虑到带有时钟信号的时序电路[[[10]](#endnote-11)]，本章中主要介绍了如何改进ISCAS`89和ITC`99基准电路模型使得它们可以支持带时钟信号的时序电路、位并行、四值逻辑和如何改进电路层级化处理。

在介绍本章之前，以下几个概念需要说明一下：

* 电路输入（Primary Input，简写是PI）：电路的原始输入，作为特殊的门类型，其值是给定的测试向量决定的；
* 电路输出（Primary Output，简写是PO）：电路的原始输出，作为特殊的门类型，其值是作为整个电路的输出结果；
* 门输入（Fan-In）：对于除电路输入和电路输出以外的门类型，每个逻辑门都至少包含一个逻辑门输入，是门进行逻辑计算的输入值；
* 门输出（Fan-Out）：对于除电路输入和电路输出以外的门类型，每个逻辑门通过对门输入值进行逻辑运算得到的结果为门输出值，逻辑门的输出值只有一个，但是逻辑门的输出可以连接到多个其它门的输入。

## 3.2 故障模拟系统模型

逻辑电路由若干个逻辑门组成，包括多个电路输入和电路输出，通过读取一组输入向量赋值给电路输入来给电路相应的输入激励值，电路输入的个数决定了输入向量的宽度；经过逻辑模拟后的电路可以得到所有电路输出的值代表电路的响应，电路输出的个数决定电路输出值的宽度[[[11]](#endnote-12)]。系统通过读取标准的电路文件，将电路信息存储在经过设计的电路网表（Netlist）中。

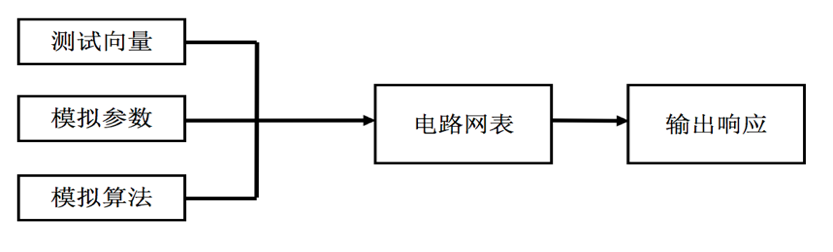


图3-1 故障模拟系统模型

如图3-1所示是故障模拟系统模型，其中测试向量可以利用系统随机数程序来生成或者读取外部的测试向量值，模拟算法本文选择并行向量单故障传播，模拟参数包括时钟周期（time cycle）和时帧（time frame）。在时序电路中存在一些时钟信号沿触发的触发器，它们在时钟信号发生跳变（高变低、低变高）的时候才会被激活[[[12]](#endnote-13)]。为了较准确地处理触发器元件，本文引入时帧的设置，将一个时钟周期分为多个时帧，这样可以模拟出上升沿触发和下降沿触发的过程。如图3-2所示描述了3时帧的时钟信号变化图，假设每个时钟周期的起始值为0，第一个时帧为信号保持状态0，第二个时帧为完整的信号保持状态1，第三个时帧为部分信号保持状态0，这样一个时钟周期就被划分为了3个时帧，每个时钟周期的时钟信号值变化为010。在每个周期内，从第一个时帧变到第二个时帧，由0跳变1将激活上升沿触发的触发器，从第二个时帧变到第三个时帧，由1跳变0将激活下降沿触发的触发器。同样地，也可以将3时帧时钟周期划分为101，这样时钟信号便会从第一个时帧到第二个时帧时激活下降沿触发的触发器，从第二个时帧到第三个时帧时激活上升沿触发的触发器。

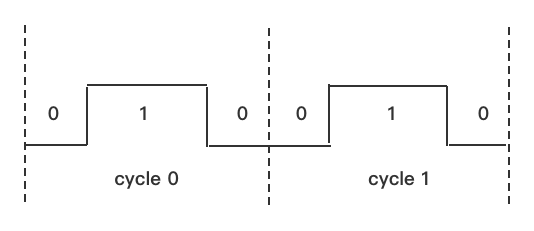


图3-2 3frame的时钟信号

## 3.3 电路元件模型

### 3.3.1 组合电路元件模型

组合电路中门级电路模型包括常见的与门（AND）、或门（OR）、非门（NOT）、与非门（NAND）、或非门（NOR）、同或门（XNOR）和异或门（XOR）。

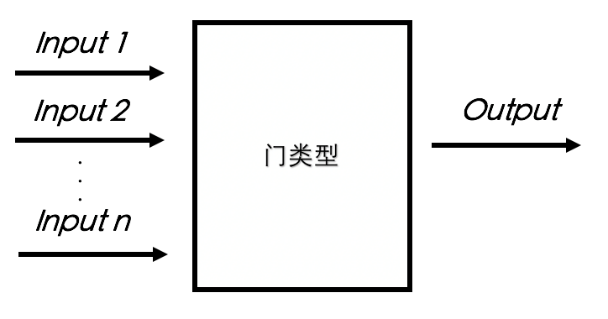


图3-3 组合电路元件模型

图3-3所示是组合电路元件模型，假设组合电路元件X输出为：

|  |  |  |
| --- | --- | --- |
|  |  | (3-1) |

Gate为指定门类型对应的逻辑运算，一般地，根据元件X的门类型有两种方法得出门的输出值，一种是查表法，一种是逻辑运算。查表法是当有了门类型和具体的门输入数量和各个门输入的值，可以通过查看真值表来得出门输出值；真值表是预先储存下来，值得注意的是，真值表细化地包含了各种门的类型下若干数量门输入的值，例如二输入或门、三输入或门一直到n输入或门。查表法的优点在于不用计算可以直接提取现成的值，但是寻找对应数量输入的门类型成了查表法的主要时间开销。查表法适用于小型、门类型单一的电路，对于大规模、门类型多样化的电路，不仅真值表需要储存庞大的数据，寻找对应的门类型的时间也大大增加，造成存储空间和时间上的浪费。逻辑运算方法不需要花费内存来存储门类型真值表，当已知具体的门类型和若干数量门输入的值，每次需要对门进行模拟时，通过具体的门所代表的布尔代数关系，对该门进行相应的逻辑运算可以得到门输出的值。通过逻辑运算的方法，不仅可以省略存储门类型真值表的内存开销，也适用于门类型多样化的电路，更关键的是可以支持位并行的并行算法[[[13]](#endnote-14)]。为了实现并行向量单故障传播算法，本文采用逻辑运算的方式。

### 3.3.2 时序电路元件模型

如图3-4所示时序电路是有组合电路和记忆元件组合成的，常见的时序电路记忆元件有触发器（Flip-Flop）、锁存器（Latch），常见的触发器类型有D触发器、JK触发器，常见的锁存器类型有D锁存器、SR锁存器。

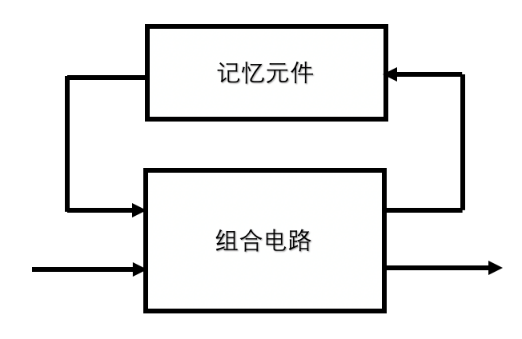


图3-4 时序电路模型

记忆元件由于结构特性，它的输出值不仅取决于输入值还取决于当前时刻的状态，因此具有记忆性的功能。图3-5是D锁存器模型图，D为锁存器输入信号，Q为锁存器输出信号，当时钟信号C为低电平即C=0时，锁存器为保持状态，Q的值保持不变；当时钟信号C为高电平即C=1时，锁存器为传输状态，Q=D。

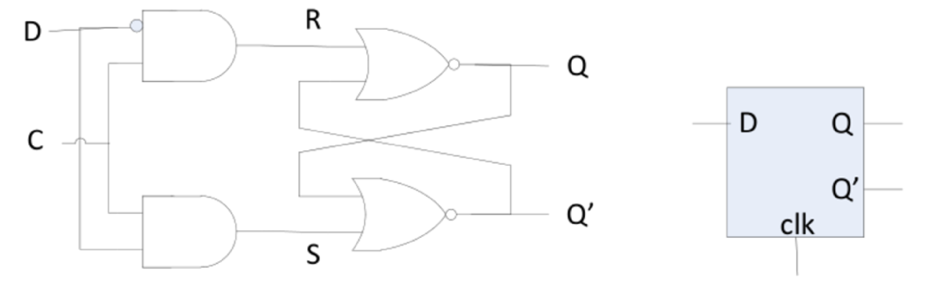


图3-5 D Latch模型

图3-6为D触发器模型图，D触发器在结构上是由两个D锁存器（主锁存器、从锁存器）加上Clock时钟控制信号构成的，在功能上与D锁存器类似，图中当时钟信号Clock=1时，主锁存器激活，数据传输D->Qm，随后Clock跳变到0，主锁存器关闭，从锁存器激活，数据传输Qm->Qs，且此时D端不再读取数据；同样地还存在对时钟信号从0跳变到1敏感的触发器。D触发器的特性方程为：

|  |  |  |
| --- | --- | --- |
|  |  | (3-2) |

方程中Q（t+1）表示下一个时钟状态下输出值，当时钟信号下降沿（或上升沿）到来时，输出Q等于数据输入端D的值，在其它时刻，输出Q的值保持不变。与D锁存器类似，在功能上D触发器具有保持和传输的状态，但不同的是D触发器的触发条件变成了时钟信号发生跳变。对于电平敏感的锁存器来说，当时钟信号保持为1时，输出信号随着输入信号变化一直随着改变，这样不容易捕捉元件某时刻的状态，而边缘敏感的触发器则可以避免这个问题[[[14]](#endnote-15)]。

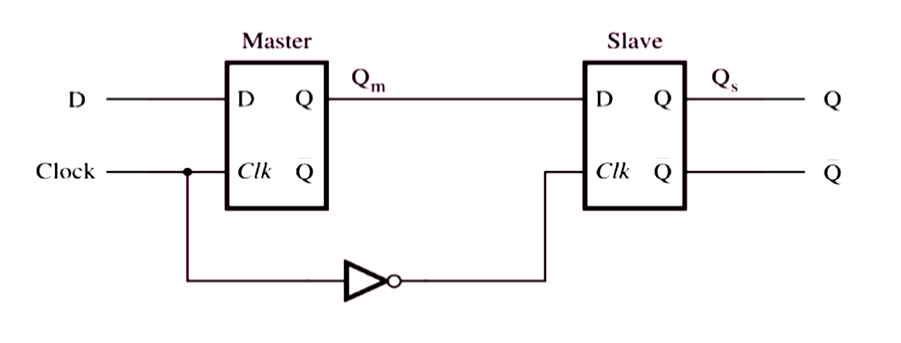


图3-6 D Flip-Flop模型

本文采用的学术界广泛应用的ISCAS`89基准电路和ITC`99基准电路，该标准定义的D触发器模型仅有一个数据输入，输出等于数据输入，缺少时钟信号的输入，这种模型就无法直接运用在带有时钟信号的电路中，本文对ISCAS`89基准电路和ITC`99基准电路中的触发器模型做了改良，以此能更好地贴近实际电路。

因为时序电路存在着反馈回路，导致在划分电路结构时时序元件可能处于较高的层级，但它的输出又作为较低层级逻辑门的输入，为了解决回路对确定电路中逻辑门的执行顺序造成的影响，以便于更准确地模拟时序电路的运行，本文将时序电路中的时序元件用组合逻辑门来表示。因此这里引入一种新的时序元件，称为延时元件（Delay Element，简称DE），主要作用是将数据储存一个时钟周期后再输出。

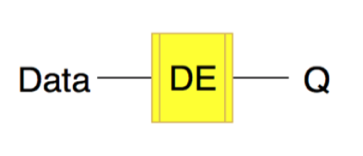


图3-7 DE模型

如图3-7中所示，延时元件为单输入单输出、作为唯一具有记忆功能的元件。在当前周期下，输出的是上一周期Data端的数据，当前周期Data的数据保存在延时元件中作为下一周期输出的新值，如表3-1中所示，Q为当前周期输出的值，Qn+1为当前周期保存的值也是下个周期的输出值。定义了唯一具有记忆功能的元件，其他的时序元件可以通过延时元件盒组合逻辑门来实现。在读取电路结构后对电路进行层级化处理的时候，本文将延时元件置于特定的层级，保证即使它与其他组合逻辑门形成回路也不会影响电路的层级化处理。

表3-1 DE特性表

|  |  |  |
| --- | --- | --- |
| D | Q | Qn+1 |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

如图3-8所示，引入了DE后，锁存器可以用一个数据选择器（MUX）和一个延时元件DE来实现。图中a部分为两位数据选择器模型，输入为两个数据信号端A、B和控制信号端S，则输出F=SB+S’A。图中c部分为D锁存器模型，D作为数据输入端，EN作为控制信号端，Q为输出信号端，通过数据选择器和延时元件的回路，实现了D锁存器传输和保持的功能。

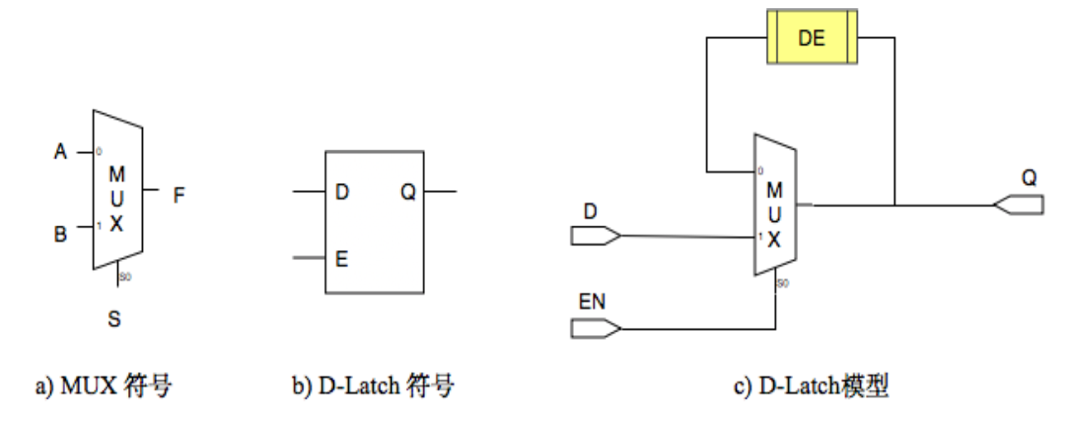


图3-8 D Latch新模型

由于D触发器是由两个锁存器构成，这里将新的锁存器模型应用到D触发器，可以得到新的D触发器的结构模型如图3-9所示，通过两个数据选择器和两个延时元件加一个非门构成，其中D为数据输入端，Clk为时钟信号输入端，Q为状态输出端，通过数据选择器和延时元件的回路，实现了D触发器边缘敏感、传输和保持的功能。

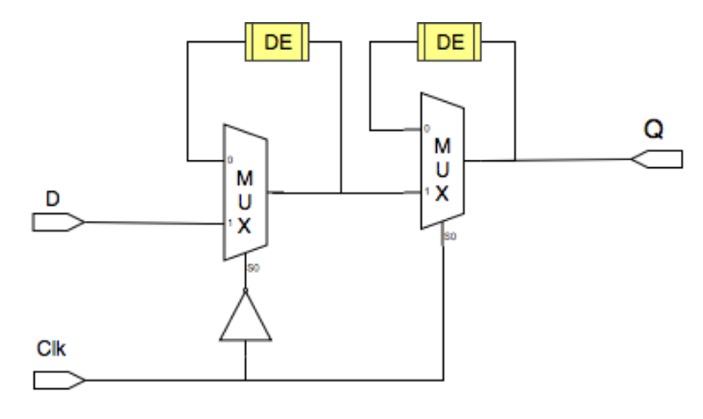


图3-9 D Flip-Flop新模型

## 3.4 逻辑门节点值

### 3.4.1 位并行

位并行是最常见的一种并行方式，一般来说电路中的输入向量只有高电平和低电平即用数值为1和0，在计算机里可以用无符号长整型变量来储存输入向量和电路中各逻辑门的节点值。

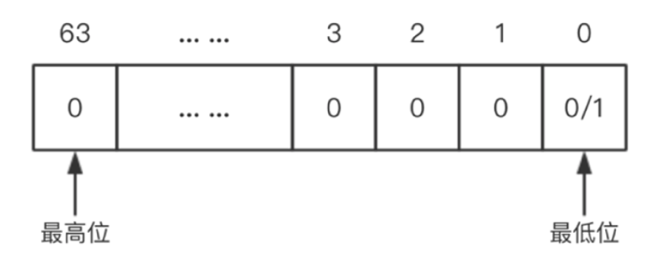


图3-10 位并行

如图3-10所示，在计算机中定义一个数值为0或1的64位无符号长整型变量，可以看到只有最低位是有效的，其余的63位都为0，造成资源浪费。位并行就是充分利用变量的每一位，每一位都储存一个输入向量或门节点的值，这样对于一个64位无符号长整型变量最多则可以储存64个数值，实现了同时对64组数值进行模拟计算[[[15]](#endnote-16)]。位并行同时适用于单线程的顺序执行模拟和多线程并行执行模拟，不仅大幅减少模拟次数还节约了用来储存门节点值的存储空间。

### 3.4.2 四值逻辑

在ISCAS`89基准电路和ITC`99基准电路标准下，对于时序电路来说需要考虑到带时钟信号的D触发器、D锁存器和总线等门类型。对于D触发器和D锁存器的初始状态通常是未知的，而总线在工作之前是保持高阻状态。对于组合电路来说，只有高电平和低电平即逻辑门的节点值为1或0，用一个变量就可以表示这两个值，称为二值逻辑；而在处理时序电路时还需要另外引入未知态（X）和高阻态（Z）即逻辑门的节点值可能为0、1、X、Z总共四个值，利用二进制编码的方法，需要两个变量来表示四种值，称为四值逻辑[[[16]](#endnote-17)]。

本文要同时考虑到组合电路和时序电路，为了能在电路故障模拟中兼容二值和四值同时支持位并行的方式，引入两个变量GMV0、GMV1来表示电路逻辑门的节点值，如表3-2所示，当GMV0与GMV1为00时，代表低电平状态，当值为11时代表高电平状态，当值为01时为未知态，当值为10时为高阻态，而当只需要用到二值逻辑时，GMV0则与GMV1相等。

表3-2 逻辑门节点值表示方法

|  |  |  |
| --- | --- | --- |
| GMV1 | GMV0 | 代表状态 |
| 0 | 0 | 低电平 |
| 1 | 1 | 高电平 |
| 0 | 1 | 未知态 |
| 1 | 0 | 高阻态 |

## 3.5 电路的网表结构

电路的网表结构记录的是整个电路的信息，包括目标电路门的数量以及每个门的具体信息（包括门与门之间的连接关系）。本文基本电路网表数据结构沿用的ISCAS`89和ITC`99基准电路标准，如图3-11所示每个门的信息包括门节点的编号、门的类型、所在层级、扇入信息、扇出信息以及该门的值等。

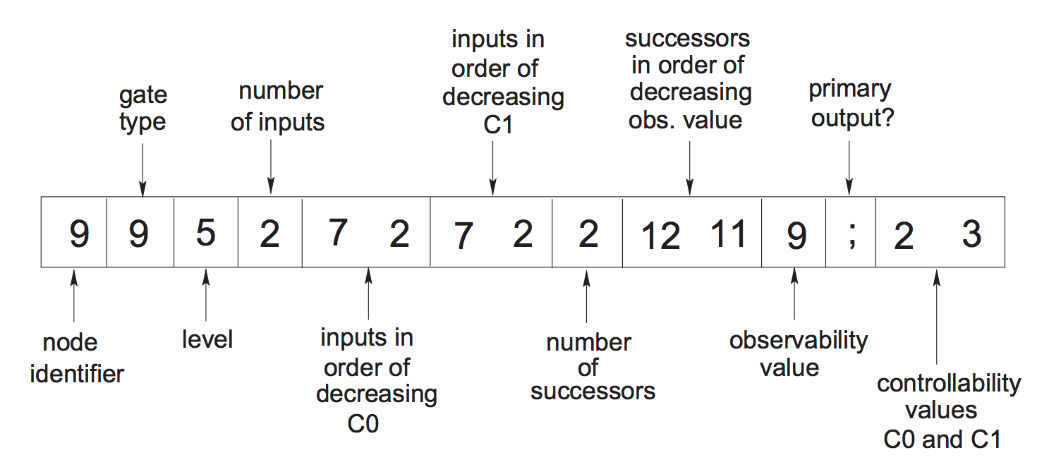


图3-11 门节点的信息

本文储存电路网表的数据结构设计如下：

* gate\_id：储存门节点的ID，ID唯一；
* gate\_type：储存门的类型；
* level：储存门所在层级信息；
* num\_fin：储存门扇入的数量；
* fan\_in [ ]：用一位数组储存扇入的节点ID，数组长度等于扇入的数量；
* num\_fout：储存门扇出的数量；
* fan\_out [ ]：用一位数组储存扇出的节点ID，数组长度等于扇出的数量；
* value：储存门的节点值，对于组合元件来说表示模拟后的门输出值，对于时序元件来说表示内部存储的值。

## 3.6 电路层级化处理

电路层级化处理的概念很早就已经被提出，其核心思想是将电路结构进行层级划分，将电路中从原始输入到原始输出的所有门进行排序，这样的做的主要目的是避免了同一层级的所有逻辑门值在模拟时的相互干扰和混乱[[[17]](#endnote-18)]。本文采用的ISCAS`89基准电路标准和ITC`99基准点路标准已经将电路进行了层级划分，其规则是将原始输入和D触发器统一归为第一个层级（level 1），将原始输出放在最末的层级（level n），而中间层级的任意一个逻辑门G的层级由它的所有扇入中层级最高的决定：

|  |  |  |
| --- | --- | --- |
|  |  | (3-3) |

一般顺序模拟的顺序就是从低层级到高层级逐级模拟，直到最后一级的原始输出模拟完毕，如图3-12所示。

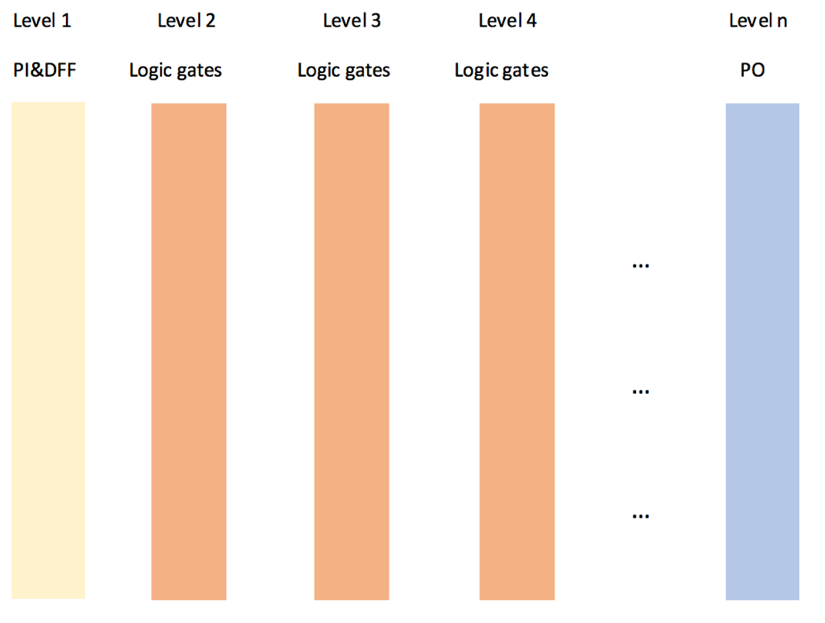


图3-12 传统层级化处理示意图

在3.3.3中阐述过ISCAS`89基准电路和ITC`99基准电路标准因为缺少时钟信号，所以引入了新的时序元件，相应地这里对电路层级划分也做一定的改善，因为时序元件在当前周期输出的是上一周期保存的值，所以本文将所有组合电路元件都模拟完后再进行时序元件的值的更新，这样的话就可以把电路中的D触发器、D锁存器等时序原件统一放到原始输出的上一层级第n-1层，如图3-13所示。

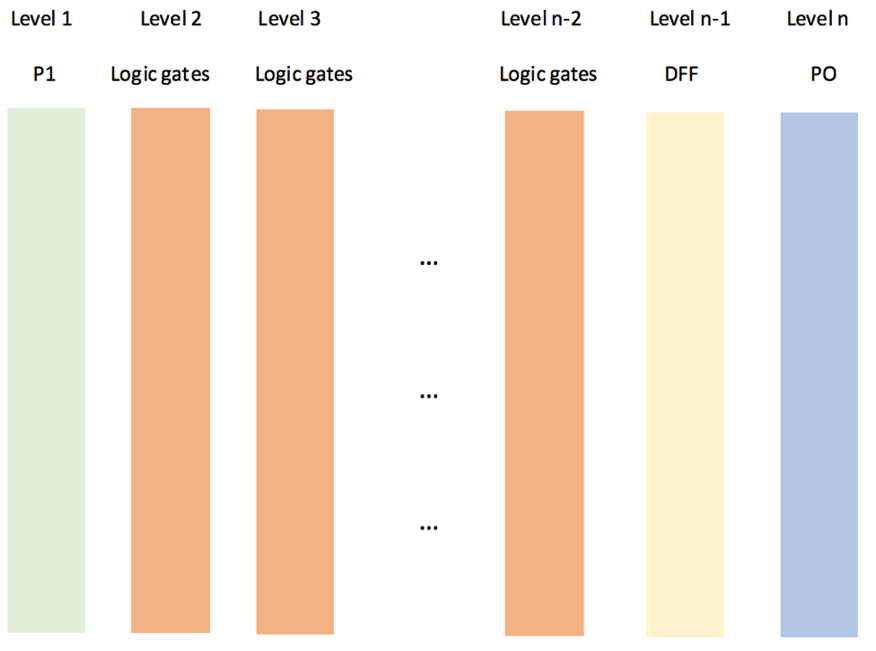


图3-13 改进后的层级化处理示意图

## 3.7 本章小结

本章结合工业实际电路，介绍了组合电路和时序电路元件模型，并对时序电路模型进行了改良，解决了ISCAS`89基准电路和ITC`99基准电路标准在模拟时缺少时钟信号导致时序元件无法驱动的问题，同时将时钟周期分为多个时帧实现了时钟沿信号激活边沿敏感触发器的功能；另外介绍了位并行的概念，并通过引入四值逻辑解决了模拟过程中对逻辑门节点值如何处理的问题；最后对ISCAS`89和ITC`99基准电路比标准下传统的层级化处理方法进行了改良，统一将时序元件放在了电路最后一个层级。

# 第4章 基于单核CPU的故障模拟设计

## 4.1 引言

前文介绍了本研究是在ISCAS`89基准电路和ITC`99基准点路标准下进行改进，针对逻辑电路包括组合和时序电路的单固定型故障的模拟研究，其中时序电路主要是同步时序电路。本章主要介绍基于单核CPU上的故障模拟设计，主要内容包括故障集生成、测试向量集生成、故障模拟器程序设计[[[18]](#endnote-19)]。

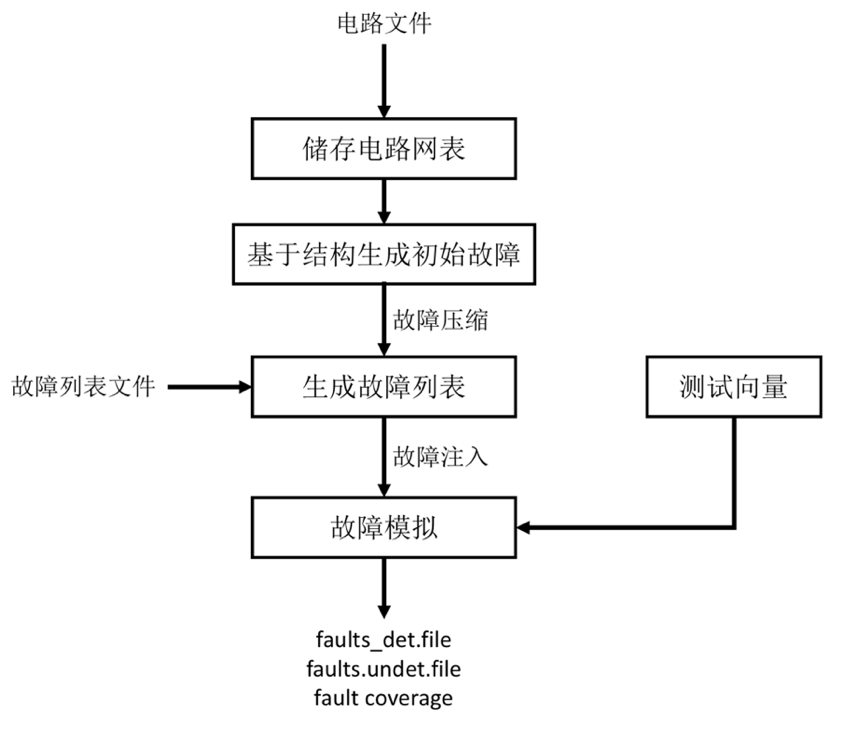


图4-1 故障模拟设计流程图

图4-1所示为基于单核CPU的故障模拟设计，在介绍本章之前需要说明一下，本文故障模拟的设计背景是使用TCL脚本语言，在已有项目程序框架的基础上完成的故障模拟研究部分，tcl命令命名为add\_faults和simulate\_pattern，add\_faults支持基于结构产生故障、读取故障文件和打印故障报告，而simulate\_pattern支持逻辑模拟和故障模拟。

## 4.2 故障集生成

### 4.2.1 基于电路结构的故障生成

本文研究是基于单固定型故障，第一步是读取电路结构后生成进行故障模拟需要的故障集。使用add\_faults命令添加参数--all代表基于电路结构产生故障集，第一步就是生成初始故障集，当读取电路信息后系统储存了电路所有节点的信息，根据2.5节，假设电路有n个节点，对于单固定型故障只有0和1两种故障，我们可以得到2n个初始故障，即所有节点统一添加stuck-at 0和stuck-at 1。

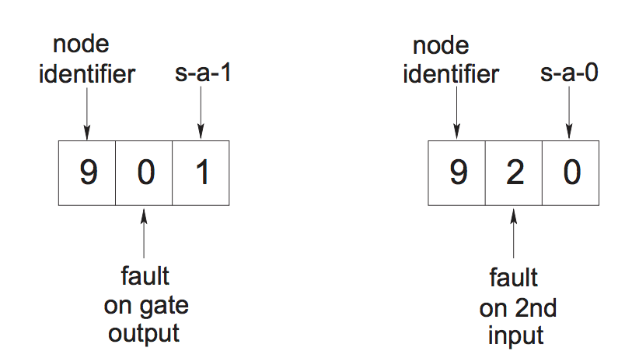


图4-2 故障格式标准

本文用三个变量来储存每个故障，如图4-2所示第一个变量代表节点的ID；第二个变量的范围为不大于节点所在门的扇入的个数，且0代表节点位于门输出上，1～n代表节点位于门的第1～n个门输入上；第三个变量的范围为0或者1，分别代表stuck-at 0和stuck-at 1。本文建立以下一个标准类命名为Fault来储存故障信息,定义了4个整形变量:

* gateid ：用来储存故障节点所在门的id；
* value ：用来储存故障节点类型，门输入或者门输出；
* type ：用来储存故障类型，stuck-at 0或stuck-at 1；
* flag ：用来储存标志位，用以判断是否等效过。

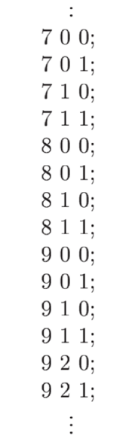


图4-3 初始故障示例

按照上述标准可以得到初始故障列表，如图4-3所示为标准电路S27.bench的初始故障示例，同样地用一个标准类命名为FaultClass来储存电路故障集的信息，本文用容器vector来储存：

* vector init\_faults：储存基于电路结构产生的初始向量；
* vector <vector> equclass：储存经过故障压缩后的等价故障集；
* vector faultlist：储存进行故障模拟时的需要的故障列表。

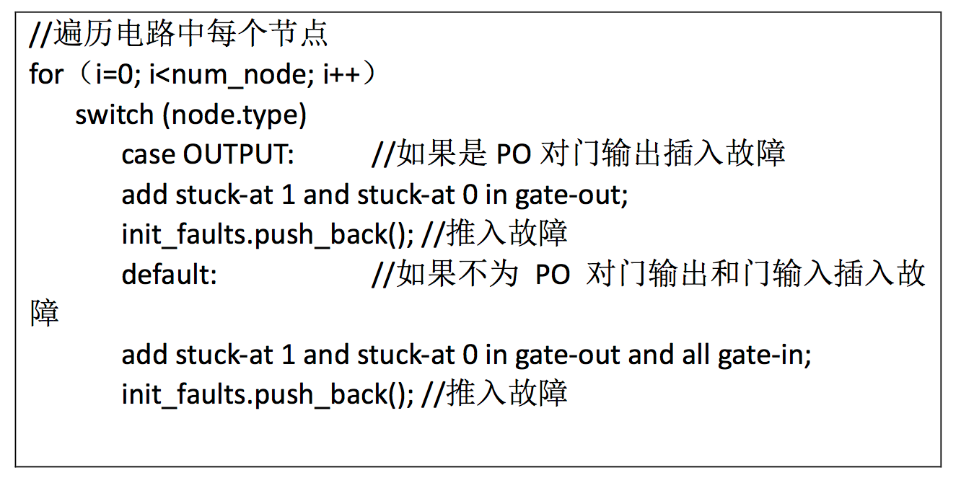
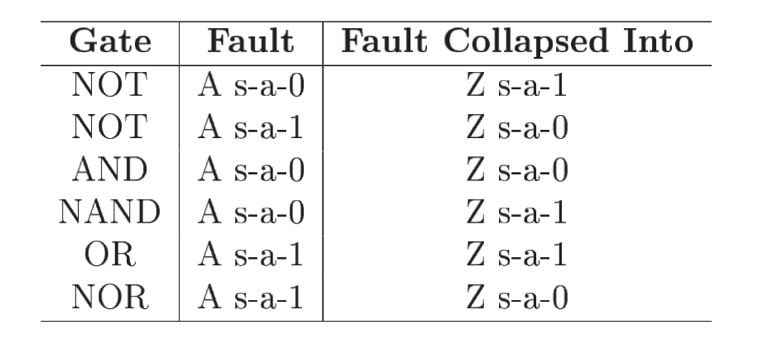


图4-4 生成初始故障伪代码

如图4-4为生成初始故障伪代码，得到初始故障后，需要对故障进行基于结构的故障压缩如表4-1所示，其中A代表该门的一个输入，Z代表该门的输出。对于两个门之间的连线，假如这条连线没有分支那么前一级门的输出上的故障完全等价于后一级对应门输入上的故障；对于时序电路中的D触发器的门类型，门输入上的故障等价于门输出上相应的故障。

表4-1 故障等价规则表



如图4-5为故障压缩伪代码，值得注意的是在做故障压缩时需要分别对门输入和门输出两类故障位置进行判断和处理，如果是门输出则需要判断其有没有分支，如果是门输入则需根据具体的门类型和该门其他输入值来考虑。另外对于D触发器门输入上的故障处理即将D触发器门当作一个缓冲器，其门输出上的故障等价于门输出上的对应故障。

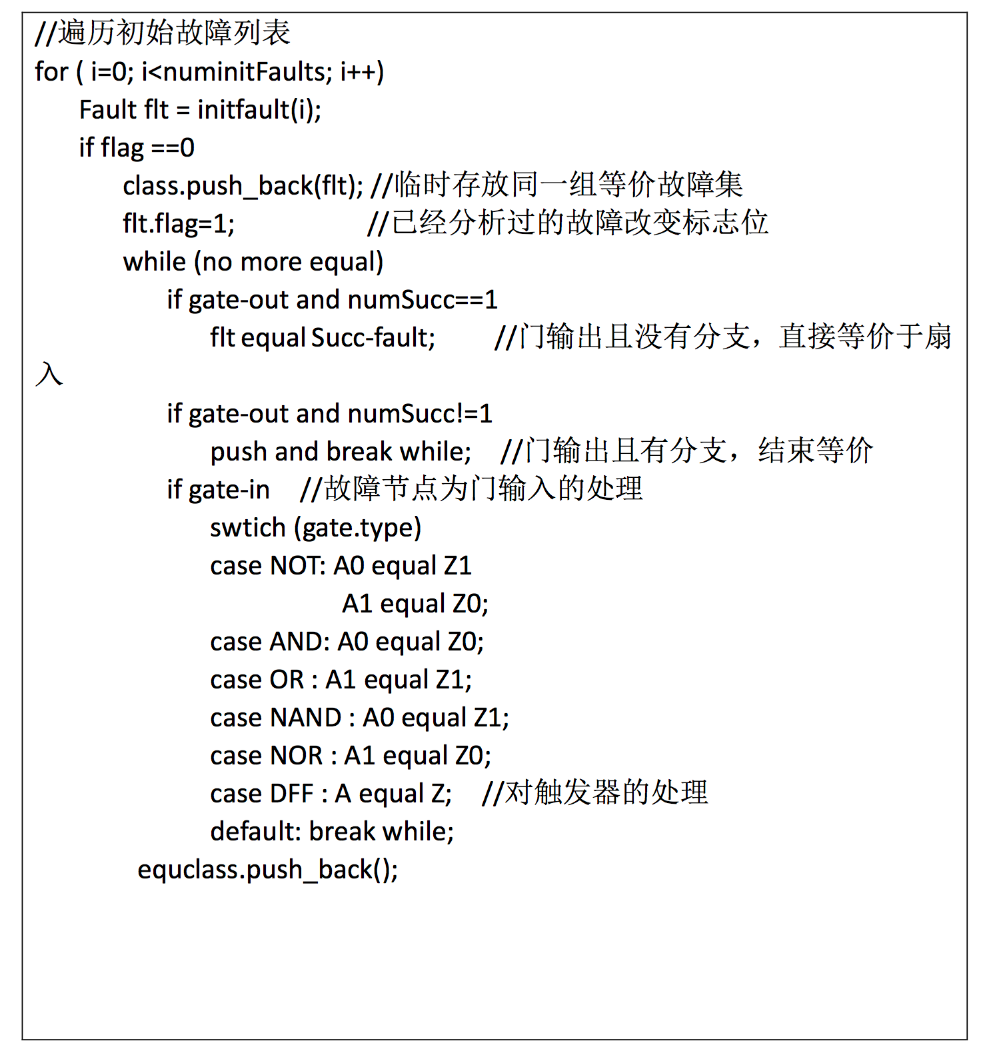


图4-5 故障压缩伪代码

经过故障压缩处理后，可以得到若干个等价故障集，如图4-6所示为标准电路S27.bench的等价故障集示例，本文在每一组等价故障集里选取离原始输出最近的一个故障作为本组等价故障的代表，组成进行故障模拟用的故障列表，因为已有大量研究发现选取离原始输出最近的故障作代表的话得到故障覆盖率更加优秀。

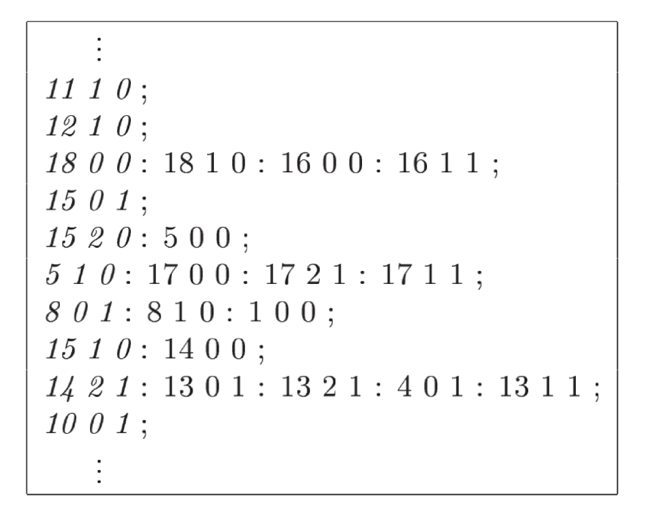


图4-6 等价故障集示例

### 4.2.2 读取故障文件生成故障

除了基于电路结构生成故障之外，还可以使用add\_faults命令添加参数--file来读取指定的故障文件，故障文件格式如4.2.1所述的故障格式标准，值得注意的是，通过读取故障文件直接生成的就是故障模拟用的故障列表，无需再进行故障压缩。

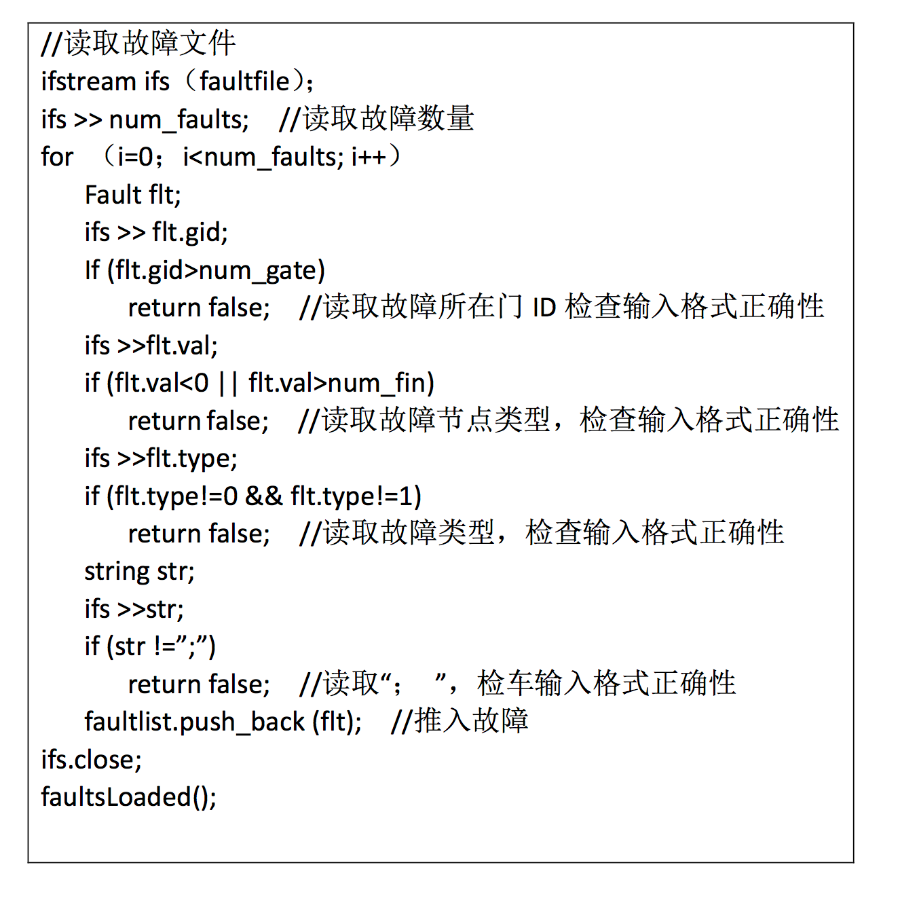


图4-7 读取故障文件伪代码

## 4.3 测试向量生成

获取故障列表之后，还需要生成测试向量，本文设计生成测试向量时考虑支持随机生成测试向量和读取测试向量文件两种方式。随机生成测试向量利用C++的随机函数，命令命名为set\_random\_patterns，可以指定生成的向量的数量；读取测试向量文件通过执行命令read\_pattrens直接读取指定向量文件，读取后用bitstream的结构来储存向量信息。

## 4.4 故障模拟器设计

一般地，故障模拟器可以作为一个单独的程序来评估指定一组测试向量集的故障覆盖率，也可以与测试向量自动生成一起运行，通过每一次的模拟结果去指导下一步的测试向量生成[[[19]](#endnote-20)]。对于故障模拟器的设计，同样地本文用一个标准类命名为FaultSim来储存故障模拟器的信息，需要储存的一些基本信息有：

* gmv0[]:与gmv1一起用来储存门节点的值，用大小为门节点数量的一维向量储存；
* gmv1[]:与gmv0一起用来储存门节点的值，用大小为门节点数量的一维向量储存；
* goodval0[]:与goodval1一起用来储存goodmachine的值，用大小为门节点数量的一维向量储存；
* goodval1[]:与goodval0一起用来储存goodmachine的值，用大小为门节点数量的一维向量储存；
* detectedfaults[]:储存能够被检测到的故障；
* undetectedfaults[]:储存未被检测到的故障，与检测到的故障的数量之和等于故障列表的长度；
* faultcoverage:储存故障模拟的故障覆盖率；
* runningtime:储存程序运行时间。

在3.4.1介绍过位并行，在故障模拟器设计里本文定义8字节的无符号长整型变量为位并行的基本结构，它的最大长度为64位，即每一次位并行模拟能同时支持64个测试向量。 在2.4介绍过故障模拟是在逻辑模拟即无故障模拟的基础上，由插入的故障点引起事件，激活事件驱动的过程，这里用一个标准类命名为Event来储存事件驱动的信息，需要储存的一些基本信息有：

* events[]：用二维向量来储存事件信息，包含事件所在的层级以及发生事件的门id；
* inQueue[]：用一维向量来储存电路中每个门是否进入向量队列的信息，大小为门节点数量；
* curLevle：用来表示目前所在的层级情况，当curLevel=-1时代表所有事件队列为空，当curLevel>0时代表电路中某一层级存在事件。

在设计故障模拟器时，当读取了故障集和测试向量集，先进行无故障模拟，将无故障模拟的结果储存下来，然后插入故障进行故障模拟，比较故障模拟后与无故障模拟的所有主输出和触发器的值，可以判断该故障能否被检测到。对于时序电路来说，在每一组位并行向量（每一组paralell）的激励下，根据设置的时钟周期和时帧数使用两个循环来实现时钟信号下的模拟；对于组合电路，需将时钟周期和时帧数都置为0。

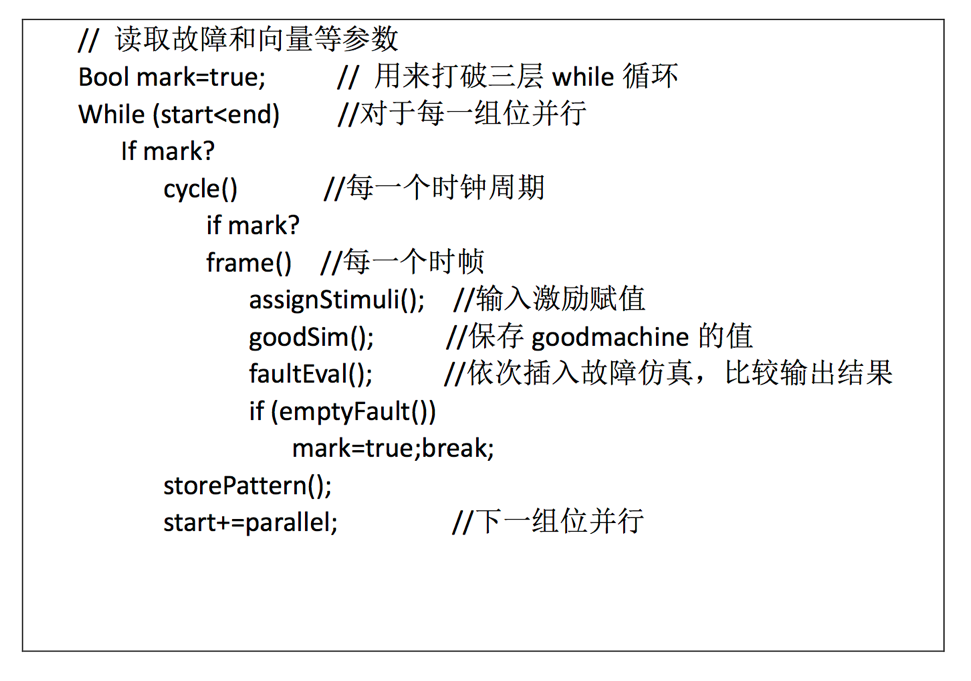


图4-8 故障模拟器伪代码

如图4-8所示为故障模拟器伪代码，其中变量mark的作用是当所有故障都被检测到时立即停止程序运行，assignStimuli（）函数用于提取测试向量给电路输入赋值，goodSim（）函数即对电路进行无故障模拟并保存输出和触发器的值，faultEval（）函数则是从故障集中选取故障依次插入电路并进行故障模拟，将模拟后的值与无故障模拟储存的值比较，倘若两次任意主输出或触发器的值不同则证明该故障能被检测到。

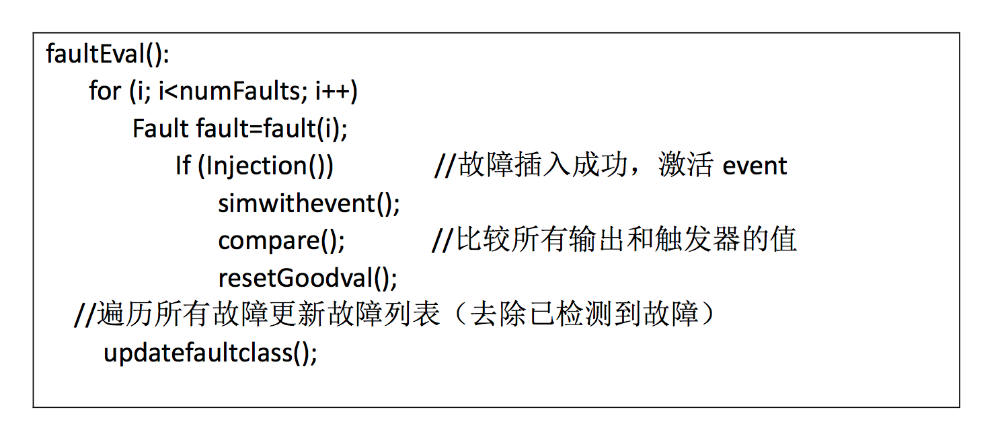


图4-9 故障分析函数伪代码

如图4-9为faultEval（）函数的伪代码，其中函数Injection（）为判断该故障是否能激活事件驱动，如果可以则进行事件驱动模拟，然后比较所有输出和触发器的值，最后复原无故障模拟的值并更新故障列表[[[20]](#endnote-21)]。在函数Injection（）判断故障能否激活事件驱动的时，要考虑到几种具体的情形：

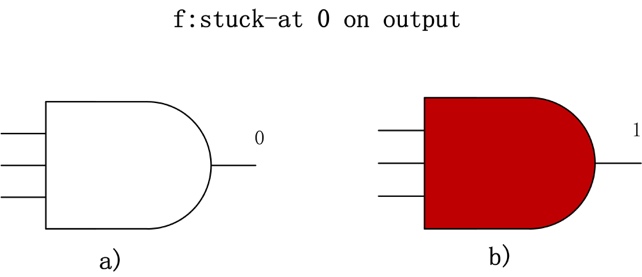


图4-10 门输出故障示例

当插入故障节点为门的输出时，如图4-10所示，假设插入的故障f为三输入与门输出上的stuck-at 0，只比较故障的值和无故障模拟时门输出上的值，如果不相同则可以激活故障模拟，如图中b）红色代表激活事件驱动。

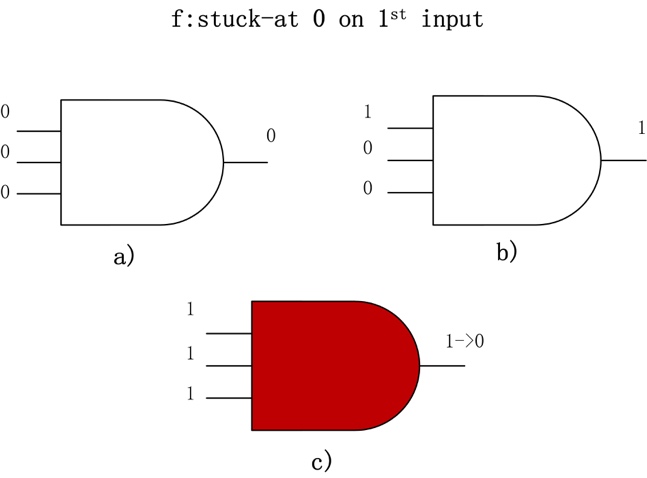


图4-11 门输入故障示例

当插入故障节点为门的输入时，如图4-11所示，假设插入的故障f为三输入与门第一个输入上的stuck-at 0，此时除了要比较无故障模拟时的第一个输入的值与故障值，还要考虑其他两个门输入的值；如a）与b）图中两种情况则不能激活事件，只有当图c）中所示的情况才能激活事件驱动，此时将与门第一个输入上的stuck-at 0故障可以等价为门输出上stuck-at 0故障，然后完成事件驱动模拟。因此，当插入故障节点为门的输入时，首先判断该故障是否能激活事件驱动，倘若能激活则将该故障等效为门输出上的相对应故障。



图4-12 门输入故障处理

上文阐述了一个有效位故障插入的处理，在本研究故障模拟器设计中采用的是最大宽度为64位的位并行模拟，当插入故障时需要将该节点上的每一位都置为故障值，如果故障节点位置为门输出，同样地只需与无故障模拟的值按位比较；如果故障节点位置为门输入，采用通过门值计算得到的门输出的值与无故障模拟的值比较，判断是否激活事件驱动模拟。如图4-12所示，注入故障f为第一个门输入上的stuck-at 0，将第一个输入节点每一位置0，通过门值计算得到输出的故障值，与无故障模拟的值按位比较，判断是否激活事件驱动模拟。

使用write\_faults命令将所有已检测到的故障和未检测到的故障打印到指定文件，通过比较故障文件，可以确保指定电路在相同测试向量不同参数设置下的故障模拟能检测到的故障结果的正确性。

## 4.5 本章小结

本章实现了基于单核CPU的故障模拟设计，详细阐述了故障生成算法和故障模拟器的实现，在故障生成中基于电路结构进行故障等效压缩，设计故障模拟器时利用了位并行和事件驱动算法，并利用将时钟周期分为多个时帧的方式，实现了对时序电路的故障模拟。

# 第5章 基于多核CPU的故障模拟设计

## 5.1 引言

在第四章介绍了基于单核CPU的故障模拟的具体流程和故障模拟器的设计，本章首先介绍了基于多核CPU编程使用的OpenCL平台，然后基于OpenCL的平台编程特性在单核CPU的基础上改进数据结构以便于更加适应多核CPU编程，最后完成了基于多核CPU的故障模拟器设计。

## 5.2 OpenCL平台

### 5.2.1 OpenCL编程流程

使用OpenCL进行多核CPU编程有以下几个步骤：

1. 获取OpenCL平台上所有设备组成信息；
2. 选取要使用的设备并创建上下文（context）和命令队列（command queue）；
3. 创建内核命令（kernel）并分配储存对象（memory object）；
4. 从主机（host）复制数据到OpenCL平台；
5. 设备按命令队列顺序执行内核程序；
6. 从OpenCL平台拷贝计算结果到主机上
7. 释放OpenCL平台资源。

OpenCL编程模型可以分为平台模型（Platform model）、执行模型（Execution model）、内存模型（Memory model）。

### 5.2.2 平台模型

OpenCL的平台模型指定了用作统一协调执行过程的主机和一个或多个能够执行内核的设备，平台可以被视为每个厂商特定的OpenCL API的实现，而设备则是每一个具体硬件的映射[[[21]](#endnote-22)]。假设某台计算机中有一个Intel的CPU和两个NVIDIA的GPU，则需要同时装载Intel OpenCL和NVIDIA OpenCL的API接口，当运行OpenCL程序时就可以检测到两种平台三个设备，如图5-1所示。这就是OpenCL可以作为异构平台的特点，CPU开发人员和GPU开发人员可以在同一个系统上分别定义自己的OpenCL框架，本研究选用OpenCL来实现多核CPU编程也是因为具有很好的可移植性。因为本文使用的是Intel i5-4210M的多核CPU，所以需要提前配置好Intel OpenCL的平台环境。

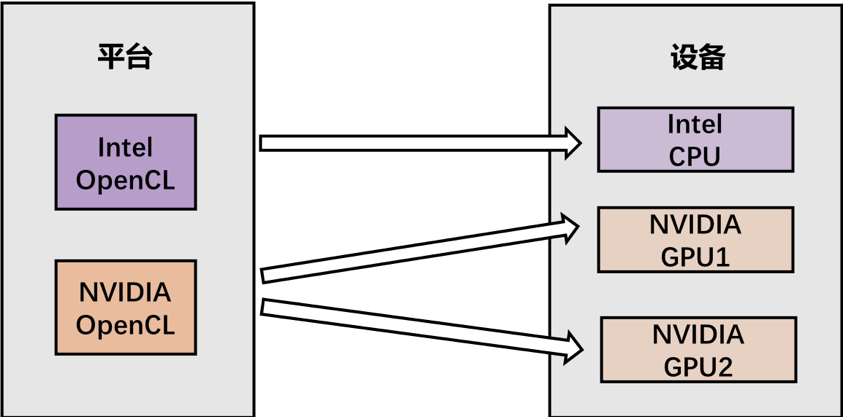


图5-1 平台与设备的关系

OpenCL平台模型中，通常是由一个CPU担任的主机用来控制和调度若干由CPU或GPU等硬件担任的设备，OpenCL平台模型是由GPU的硬件模型映射过来的，所以解释时需要用到一些GPU中的术语，在每个设备中由OpenCL定义的提供给用户执行内核程序的最小计算单元可称为线程（thread），若干个执行相同内核指令的线程组成线程块（thread block），这种执行方式称为单指令多数据（SIMD）；单个设备中又可存在若干个线程块，称为网格（grid），不同的线程块又可以执行不同的指令，又称为多线程任务并行。如图5-2为平台模型示意图，主机可以控制多台不同的设备，每个设备至少包含一个线程块，每个线程块又至少包含一个线程。

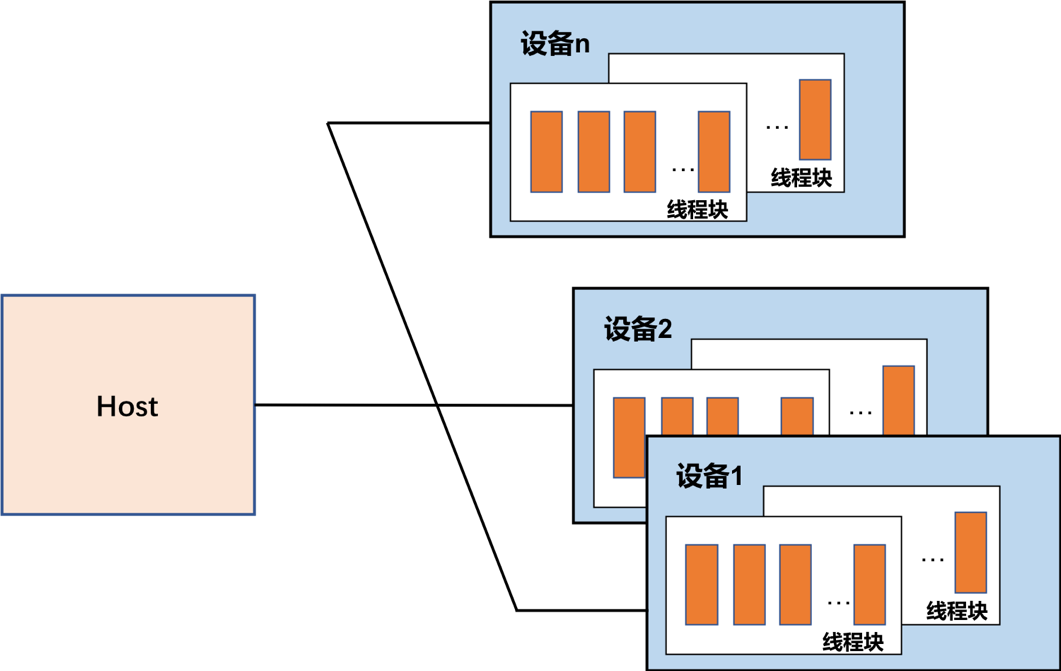


图5-2 平台模型示意图

### 5.2.3 执行模型

OpenCL执行模型是应用程序通过主机端对设备端上的内核程序进行管理，主要包含主机程序和内核程序（类似于C语言中的函数）两部分，主机通过定义上下文来实现对OpenCL资源调配和对设备的控制，上下文包含设备信息、储存对象和命令队列三部分。根据OpenCL定义的内核编程语言用户可以编写内核程序，它负责完成OpenCL中的实际工作，也是线程执行的具体动作，在部分OpenCL实现中用户可以跟其他语言编写的原生内核实现交互，但大多数情况下内核是需要用户使用内核编程语言编写实现的，OpenCL中的内核编程语言是OpenCL C编程语言[[[22]](#endnote-23)]。内核程序执行的实例称为工作项，所有工作项可以同时执行，在执行内核程序之前，先要建立一个索引空间来对设备里的每个工作项进行标识，每个工作项都将执行相同的内核程序，它们不仅在全局里有全局ID，并且在每个工作组里各个工作项都有一个局部ID，这个索引空间在OpenCL称为NDRange。如图5-3为单个设备执行模型示意图，其中当主机指挥设备工作时，通过执行程序将命令提交到命令队列中，命令队列中的命令可以按照先进选出顺序执行也可非顺序执行，另外主机可以为单个设备创建多个命令队列，这样就可以实现任务的并行性。

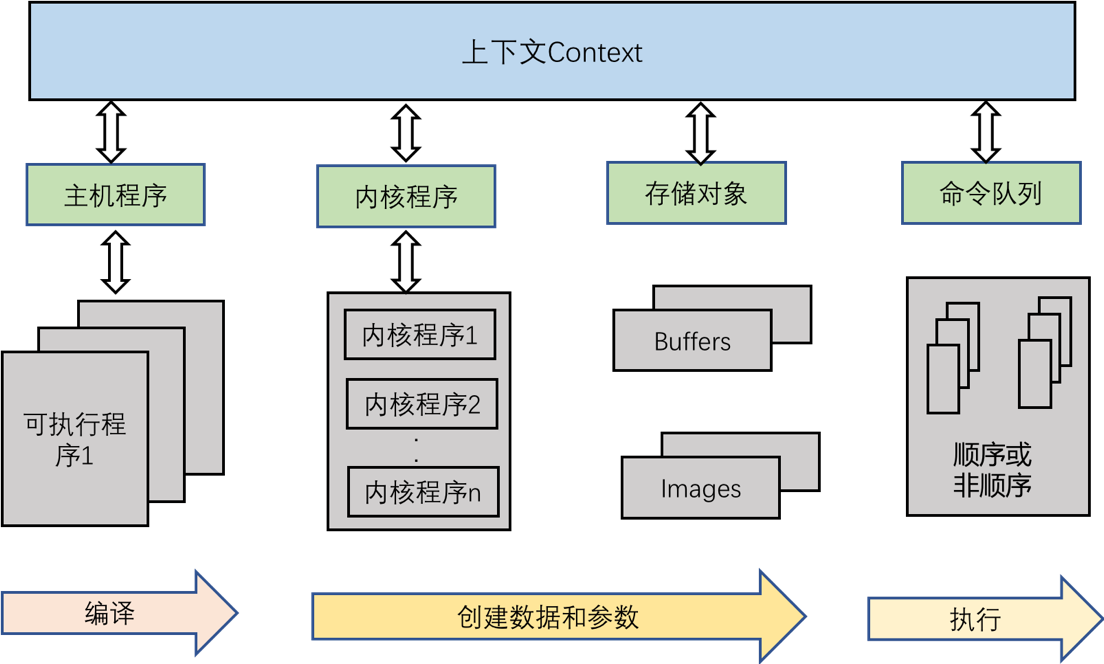


图5-3 单设备执行模型示意图

关于内核函数的执行涉及以下几个概念：

（1）工作项（work item）：执行内核程序的基本单元，对应硬件设备上的一个计算单元，是执行单指令多数据的并行基本单位，所有的工作项都可以同时执行，每个工作项会被分配唯一且固定的全局ID（global id）；

（2）工作组（work group）：由一个或多个工作项组成，在一个工作组内的工作项执行相同的内核程序只是数据不相同，同一个工作组内的工作项可以互相通信和共享数据，且工作组内的工作项会被另外分配组内局部ID，每个工作组会被分配唯一的工作组ID（workgroup id）；

（3）多维索引（NDRange）：工作组被划分标识组成的更大的结构，多维索引一般从可以从一维到三维空间表示，本文统一采用的三维空间。

### 5.2.4 内存模型

OpenCL定义的内存模型是从GPU的硬件内存模型映射过来的，同样适用于多核CPU处理器，分为主机内存、私有内存、局部内存、全局内存、常量内存和纹理内存，图5-4为OpenCL内存模型示意图，其中私有内存仅对一个工作项可见，局部内存对于工作组中的所有工作项是可见的，全局内存、常量内存和纹理内存对整个多维索引空间中的所有工作项都是可见的[[[23]](#endnote-24)]。

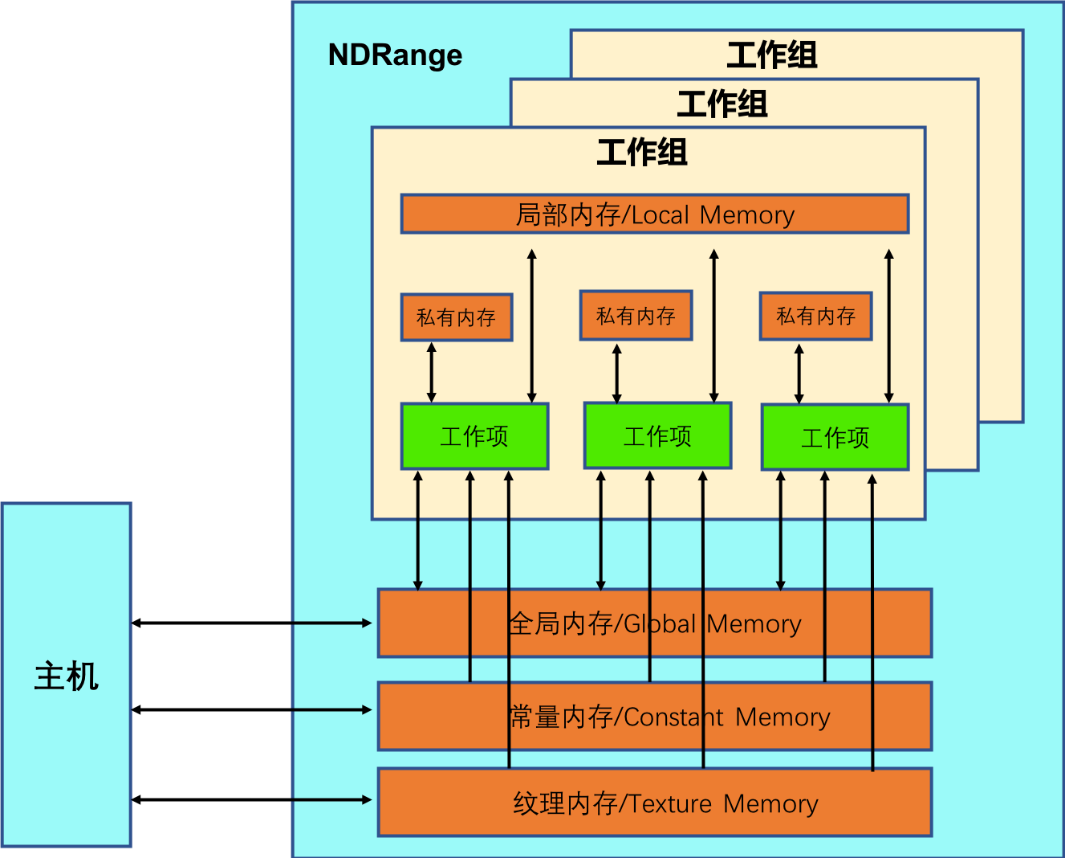


图5-4 内存模型示意图

1. 主机内存（Host Memory）：只有主机可见，可以与OpenCL的全局内存、常用内存和纹理内存互相通信进行数据传输；
2. 全局内存（Global Memory）：能够被整个索引空间的所有工作项进行读写操作，访问速度在所有内存中最慢，且内存大小比CPU小的多；
3. 常量内存（Constant Memory）：能够被整个索引空间的所有工作项进行读操作，常量内存所存数据不允许被修改；
4. 纹理内存（Texture Memory）：对图像单位进行缓存，在大多数情况下比全局内存访问速度更快；
5. 局部内存（Local Memory）：能够被同一个工作组的所有工作项进行读写操作，为同组工作项所共享的资源；
6. 私有内存（Private Memory）：只能被对应的单个工作项进行读写操作，是访问速度最快的内存。

## 5.3 基于多核CPU的数据结构改进

在第四章中定义的数据结构是基于单核CPU编程而设计的，因为内存相对充足而且是单线程执行的，所以我们定义了较多的变量来减少计算过程。而对于多核CPU编程来说，考虑到OpenCL的内存模型特点，为了减少主机与OpenCL数据交流的时间花销且不会对每个工作项的执行产生较大影响，本文在第四章单核CPU数据结构的基础上尽可能压缩数据采用更少的变量来描述电路信息：

* gate\_type[]:用大小为电路门节点数量的数组来储存电路门类型信息，按照节点编号的顺序依次存储；
* level[]：用大小为电路门节点数量的数组来储存各个门所在层级信息，按照节点编号的顺序依次存储；
* fanin[]：用大小为所有门扇入总量的数组来储存各个门扇入的信息，按照门节点编号的顺序依次存储；
* fanout[]：用大小为所有门扇出总量的数组来储存各个门扇出的信息，按照门节点编号的顺序依次存储；
* first\_fanin\_index[]：用大小为电路门节点数量的数组来储存每个门节点的第一个扇入在数组fanin[]中的位置，则对于门节点n的扇入个数为first\_fanin\_index[n+1]-first\_fanin\_index[n]，在fanin []中门节点n的输入信息就储存在第first\_fanin\_index[n]到first\_fanin\_index[n+1]的位置；
* first\_fanout\_index[]：用大小为电路门节点数量的数组来储存每个门节点的第一个扇出在数组fanout[]中的位置，则对于门节点n的扇出个数为first\_fanout\_index[n+1]-first\_fanout\_index[n]，在fanout[]中门节点n的输入信息就储存在第first\_fanout\_index[n]到first\_fanout\_index[n+1]的位置。

因为故障模拟是在无故障电路的基础上进行的事件驱动模拟，在4.4章节本文介绍了关于事件驱动的数据结构，基于多核CPU编程为了更加贴合OpenCL平台，使事件驱动更加高效，本文对事件驱动的数据结构做如下优化：

* level\_flag[]：用类型为布尔类型的一维向量来储存每一层级是否有事件存在的标识，长度为电路层级的数量，按层级顺序储存，如果某一层级标识为0表示该层级上没有等待模拟的门节点，如果标识为1则表示有需要模拟的门；
* gate\_flag[]：用类型为布尔类型的一维向量来储存电路中所有门的状态，长度为电路门节点的数量，按门节点id的顺序储存，如果某个门节点的标识为1表示该门节点被激活需要进行模拟，如果某个门节点的标识为0则表示该门节点不需要进行模拟。

该结构优化是基于多核CPU编程单指令多数据的特点，不需要再像4.4章节中利用事件队列和标识数组来记录事件信息，而是只需要利用层级标识和门标识两个一维向量来记录电路中所有的事件信息[[[24]](#endnote-25)]，如图5-5是事件驱动伪代码。

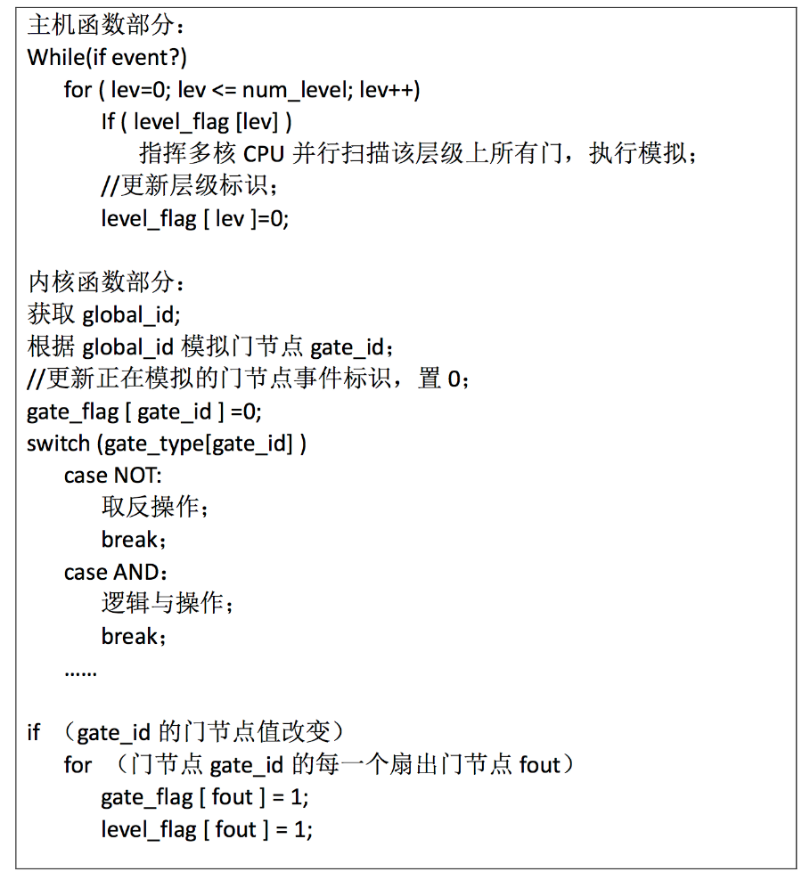


图5-5 基于多核CPU的事件驱动伪代码

## 5.4 基于多核CPU的故障模拟器设计

如图5-6为基于多核CPU故障模拟流程图，其中读取电路、生成故障列表和生成测试向量几个步骤与章节4.4的步骤相同[[[25]](#endnote-26)]。在无故障模拟环节，电路赋值后通过OpenCL指挥多核CPU进行并行模拟，这里同样采用64位并行向量单故障传播算法，多核CPU并行过程主要是实现了一次性模拟每个层级上的所有门节点；在故障模拟环节，从故障列表顺序插入故障，每一次插入故障后判断是否激活事件，如果事件被激活则通过OpenCL指挥多核CPU进行并行模拟，然后并行比较所有电路主输出（PO）和D触发器（DFF）的值来判断当前故障能否被检测到。

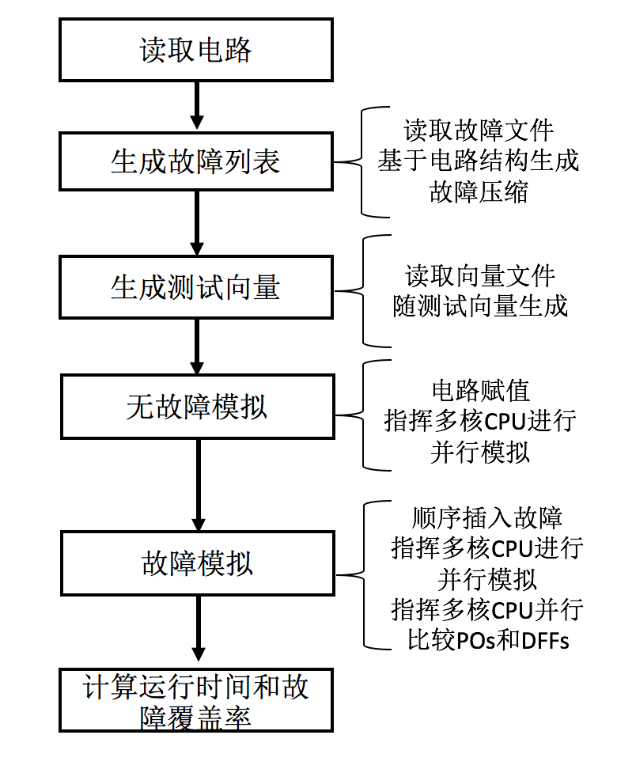
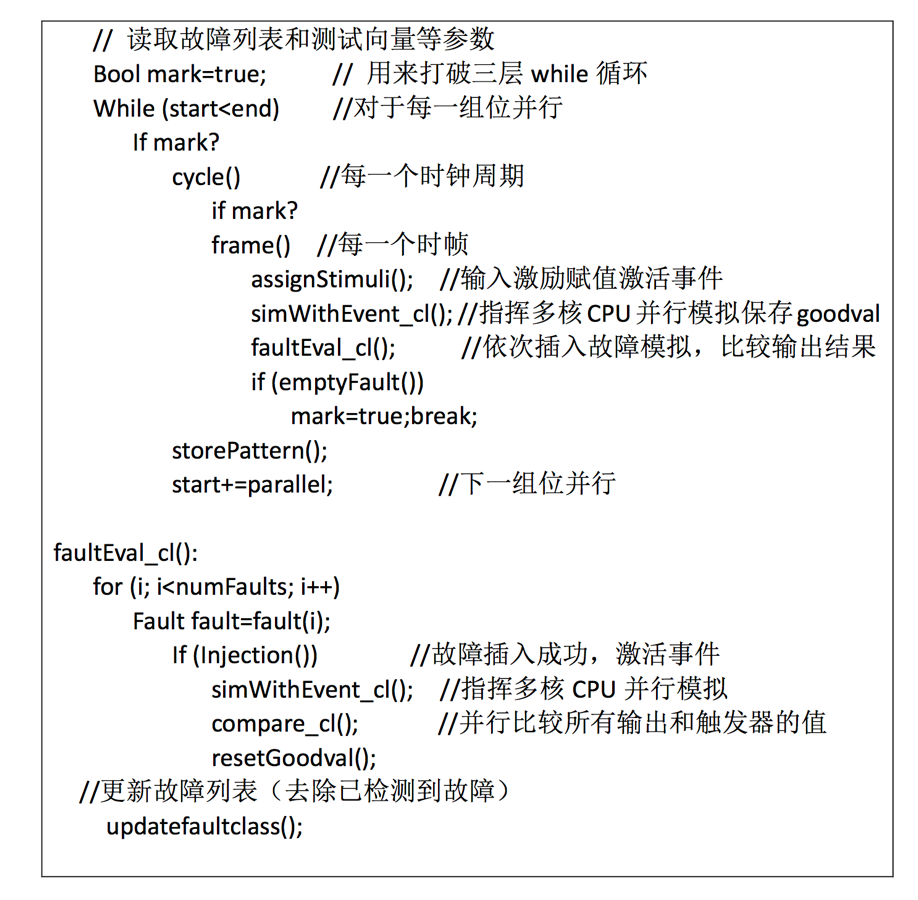


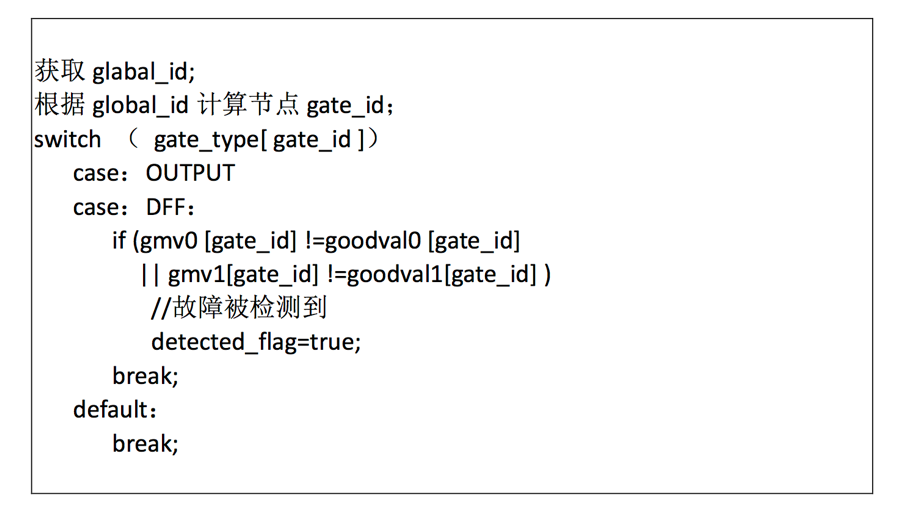
图5-6 基于多核CPU故障模拟流程图

图5-7为基于多核CPU的故障模拟算法伪代码，事件驱动算法在5.3节中已给出，图b为并行比较电路主输出和D触发器的算法伪代码。其中图a主机函数部分中函数assignStimuli()给电路输入赋值并激活事件，函数simWithEvent\_cl()主要利用事件驱动内核函数完成无故障模拟并将无故障电路的门节点值保存下来，然后在函数faultEval\_cl()中依次插入故障列表中的故障进行故障模拟,图a下方为faultEval\_cl()函数实现的伪代码。

本章主要通过基于任务并行的方式，实现了利用多线程同时对多个门值进行计算和比较。在无故障模拟阶段，按层级顺序在同一层级上的同时模拟多个门节点；在故障模拟阶段，每次插入故障后会激活事件，故障模拟器对事件所在层级的多个门节点并行模拟，接着对受事件影响的后续事件所在的层级依次进行并行模拟[[[26]](#endnote-27)]。基于任务并行既保证了逻辑电路故障模拟的正确性，同时充分利用了多线程并行性加速模拟。



a) 主机函数伪代码



b) 内核函数伪代码

图5-7 基于多核CPU故障模拟算法

图5-8为具体程序结构图，在故障模拟完成后可以用write\_faults命令将所有已检测到的故障和未检测到的故障打印到指定文件，通过比较故障文件，就可以确保指定电路在相同测试向量不同参数设置下的故障模拟能检测到的故障结果的正确性。



图5-8 具体程序结构

## 5.5 本章小结

本章实现了基于多核CPU的故障模拟设计，详细介绍了多核CPU编程平台以及在单个CPU故障模拟的基础上对数据结构进行了改进优化，最后给出了多核CPU故障模拟设计的具体实现。

# 第6章 实验及结果

## 6.1 引言

本章通过实验证明基于多核CPU对逻辑电路故障模拟的加速效果，6.2节为实验平台的介绍，6.3节为实验结果展示以及对实验结果进行分析，6.4节为本章小节。

## 6.2 实验平台

本文旨在通过实验证明基于多核CPU对逻辑电路故障模拟的加速效果，实验电路使用了ISCAS`89基准电路以及ITC`99基准电路，并与Atalanta故障模拟软件进行对比。本文的实验是在支持OpenCL2.1的Ubuntu16.40系统上运行的，计算机硬件配置的是型号为Intel i5-4210M的双核四线程CPU以及8G内存，处理器基本频率为2.6GHz，睿频可达3.2GHz，表6-1为实验平台设备型号和参数。

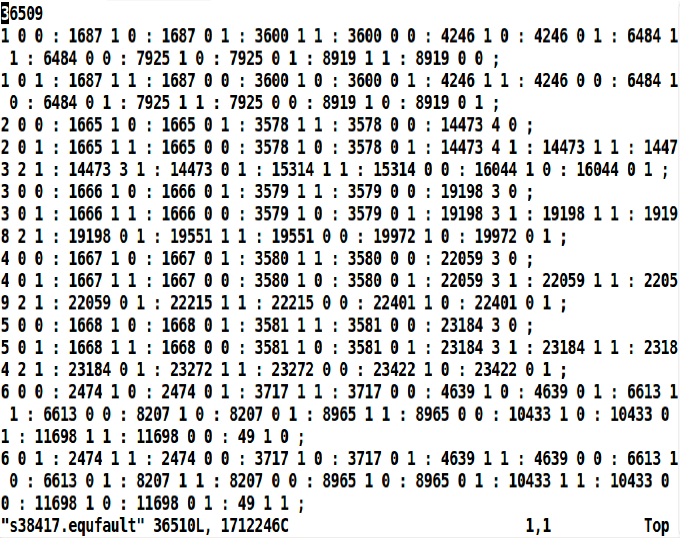
表6-1 实验平台型号与参数

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 多核编程平台 | CPU型号 | 频率 | 线程数 | 全局内存 | 全局缓存 | 局部内存 |
| OpenCL  2.1 | Intel  i5-4210M | 2.60GHz | 4 | 8G | 64KB | 32KB |

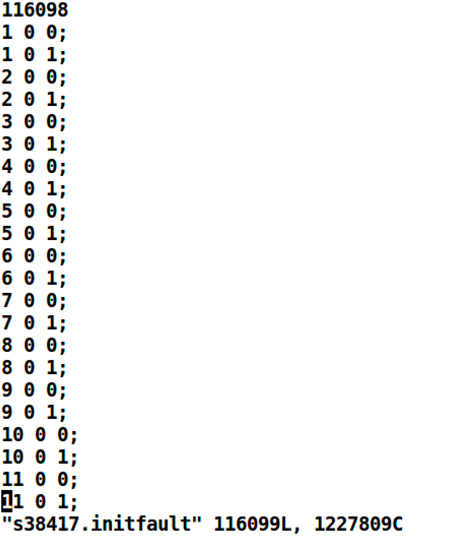
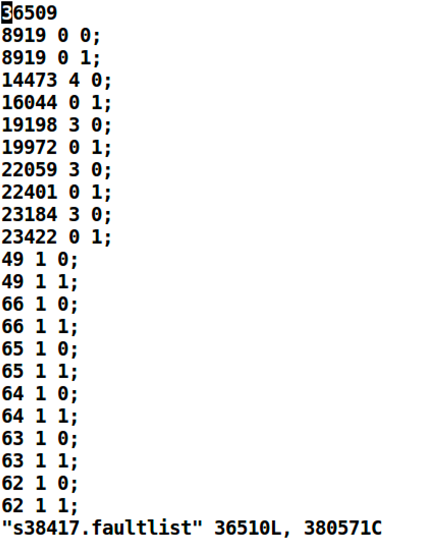
## 6.3 实验结果与分析

### 6.3.1 故障生成与故障压缩验证

在4.2节介绍过故障生成和故障压缩的设计，本实验在设计故障模拟的添加故障过程中可以使用add\_faults命令的dump参数将基于电路结构生成的初始故障、故障压缩后的等价故障集以及最终用于故障模拟的故障列表打印成文本文件，以此来验证故障生成和故障压缩环节的正确性。以s38417电路为例，如图6-1为初始故障文件、等价故障集文件和故障列表文件的对比，s38471电路有58049个电路节点，从图b可知生成了116098个初始故障是电路节点值的两倍，与理论相符（见2.5.2节）；将图a等价故障集与图c故障列表作对比，等价故障集的数量与故障列表中故障数量相同，且故障列表中每一个故障都是从每一个等价故障集中选取离电路主输出最近的代表故障，与理论相符（见4.2.1节）；我们保证了故障生成和故障压缩阶段的正确性。



图a) 等价故障集dump 文件

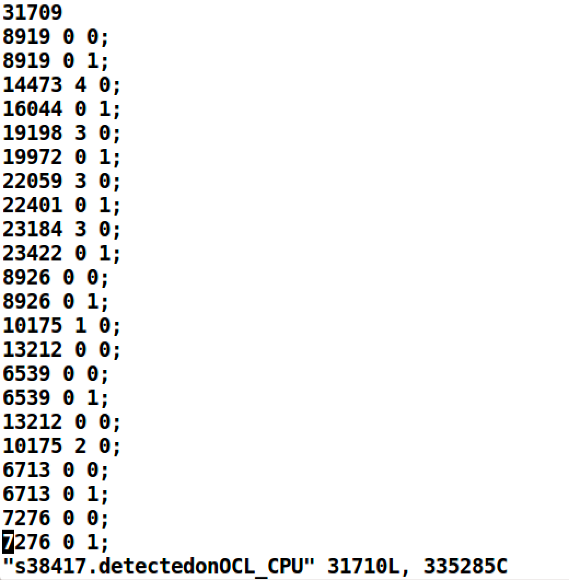
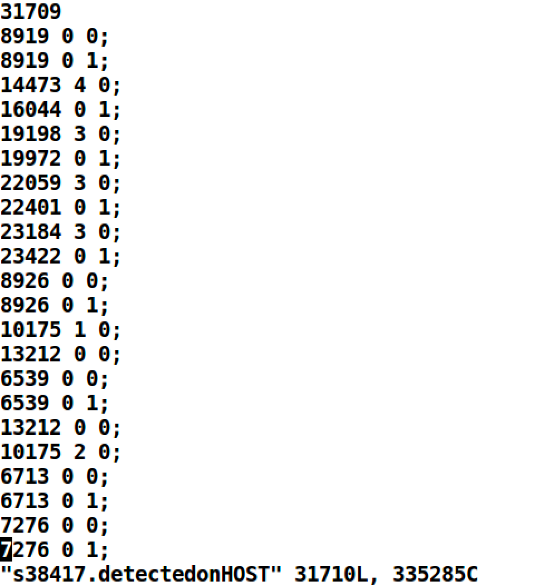
 

图b) 初始故障dump文件 图c) 故障列表dump文件

图6-1 故障生成与故障压缩验证

### 6.3.2 可检测到故障正确性验证

在4.1节介绍过故障模拟的流程，在故障模拟完成后可以得到可检测到故障、无法检测到故障和故障覆盖率等信息，本实验在故障模拟完成后可以使用write\_faults命令指定将可检测故障或无法检测故障打印成文本文件，以此来验证基于单核CPU和基于多核CPU两次故障模拟检测故障结果的正确性。同样以s38417为例，如图6-2为基于单核CPU故障模拟可检测故障文件与基于多核CPU可检测故障文件的对比，两次模拟可检测到故障的数量都是31709且每个故障信息都相同，说明两次模拟结果一样，这样我们便保证了电路模拟阶段的正确性。

图a) 基于多核CPU可检测故障文件 图b) 基于单核CPU可检测故障文件

图6-2 可检测到故障正确性验证

### 6.3.3 故障模拟结果

本研究共选取了6个时序电路作为实验对象，分别是ITC`99基准电路中的b17、b20、b21和b22电路，以及ISCAS`89基准电路中的b38417和b38584电路，电路信息如表格6-2所示，故障压缩率平均值为41.83%，符合2.5节中的理论预期。

表6-2 实验电路信息

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 电路名称 | 逻辑门数量 | 时序元件数量 | 初始故障数量 | 压缩后故障 | 压缩率 |
| b17 | 37446 | 1415 | 154488 | 76942 | 49.80% |
| b20 | 22557 | 490 | 117858 | 51838 | 43.98% |
| b21 | 23100 | 490 | 120108 | 52677 | 43.86% |
| b22 | 33569 | 735 | 174894 | 77144 | 44.11% |
| s38417 | 24167 | 1636 | 116098 | 36509 | 31.45% |
| s38584 | 21149 | 1452 | 111022 | 41980 | 37.81% |

完成基于单核CPU的故障模拟设计后，首先在5000个测试向量下将本研究设计与成熟的故障模拟软件Atalanta进行对比，结果如表6-3。通过实验结果看出本研究设计离对比工具还有不小的差距，主要原因是由于本人能力有限，本文设计在算法和数据结构上主要由自己独立完成，与已经十分成熟的软件相比还有许多优化的空间。

表6-3 对比实验

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 电路标准 | 电路名称 | 本文设计 | | | Atalanta | | |
| 故障数 | Run time  （s） | Fault coverage | 故障数 | Run time  （s） | Fault coverage |
| ITC`99 | b17 | 76942 | 223.51  87.26  95.34  261.13 | 84.26% | 76625 | 129.93  43.55  45.91  113.63 | 95.44% |
| b20 | 51838 | 96.16% | 45459 | 97.19% |
| b21 | 52677 | 95.57% | 46154 | 96.15% |
| b22 | 77144 | 94.81% | 72339 | 97.43% |
| ISCAS`89 | s38584 | 41980 | 61.70  49.82 | 87.37% | 36595 | 17.79  11.37 | 94.86% |
| s38417 | 36509 | 86.54% | 31348 | 93.80% |

在完成基于多核CPU故障模拟设计后，本研究利用控制变量分别测试了在相同时钟周期内不同随机测试向量数量以及在相同随机测试向量数量下不同时钟周期，对故障模拟运行时间、故障覆盖率以及加速效果的影响。首先我们设定时钟周期为1个cycle，用6个基准电路作为实验对象，分别在不同数量随机测试向量下运行故障模拟，这里需要说明的是在同一组故障模拟比较实验中随机测试向量集是相同的，我们选取系统运行的真实时间（Wall time）来作为评估故障模拟的时间。表格6-4为各实验电路故障模拟的实验结果，表中第三列为随机生成测试向量的数量，我们以1024个测试向量为对比基数，第四大列和第五大列分别为基于单核CPU和基于多核CPU的故障模拟运行时间和故障覆盖率结果。

从运行时间上来看，在同一时钟周期下随着随机测试向量数量的倍增，同一电路下的故障模拟运行时间有增加，故障覆盖率也呈上升趋势，整体基于多核故障模拟与基于单核故障模拟时间平均加速比为2.59，故障覆盖率平均值为90.11%。因为本实验计算机设备的线程数为4，考虑到在整个故障模拟算法执行过程中包含不少仍然需要依序串行执行的的部分（如故障注入的过程），并没有完全实现并行化，所以我们的理论预期为趋向于3倍左右的加速，实验结果符合预期；另外在同一电路下随着随机测试向量数量增加，故障覆盖率呈现上升趋势，因为更多的测试向量会增加对故障的检测能力，90.11%的平均故障覆盖率也表现出了较好的测试质量。

表6-4 不同数量测试向量下故障模拟结果

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 电路标准 | 电路名称 | 测试向量  数量 | 单核CPU | | 多核CPU | | 加速比 |
| Wall time（s） | Fault coverage | Wall time（s） | Fault coverage |
| ITC`99 | b17 | 1024 | 197.58 | 80.27% | 53.84 | 80.27% | 3.67 |
| 2048 | 202.44 | 82.11% | 57.19 | 82.11% | 3.54 |
| 4096 | 214.26 | 83.34% | 64.73 | 83.34% | 3.31 |
| 8192 | 225.53 | 84.71% | 73.22 | 84.71% | 3.08 |
| b20 | 1024 | 71.09 | 94.56% | 25.30 | 94.56% | 2.81 |
| 2048 | 78.37 | 95.44% | 28.81 | 95.44% | 2.72 |
| 4096 | 81.75 | 95.84% | 30.97 | 95.84% | 2.64 |
| 8192 | 89.28 | 96.39% | 34.20 | 96.39% | 2.61 |
| b21 | 1024 | 79.20 | 94.26% | 29.77 | 94.26% | 2.66 |
| 2048 | 81.32 | 94.88% | 32.14 | 94.88% | 2.53 |
| 4096 | 88.31 | 95.13% | 37.26 | 95.13% | 2.37 |
| 8192 | 96.57 | 96.60% | 39.90 | 96.60% | 2.42 |
| b22 | 1024 | 237.59 | 93.42% | 97.77 | 93.42% | 2.43 |
| 2048 | 246.24 | 94.22% | 103.03 | 94.22% | 2.39 |
| 4096 | 254.95 | 94.50% | 111.82 | 94.50% | 2.28 |
| 8192 | 270.84 | 95.20% | 123.11 | 95.20% | 2.20 |
| ISCAS`89 | s38584 | 1024 | 52.04 | 85.57% | 21.51 | 85.57% | 2.46 |
| 2048 | 54.09 | 86.14% | 24.04 | 86.14% | 2.25 |
| 4096 | 55.94 | 87.20% | 26.51 | 87.20% | 2.11 |
| 8192 | 63.51 | 88.82% | 29.54 | 88.82% | 2.15 |
| s38417 | 1024 | 41.97 | 85.37% | 16.65 | 85.37% | 2.52 |
| 2048 | 45.56 | 85.64% | 19.22 | 85.64% | 2.37 |
| 4096 | 46.66 | 86.16% | 20.92 | 86.16% | 2.23 |
| 8192 | 52.35 | 86.78% | 22.86 | 86.78% | 2.29 |

第二步我们把随机测试向量的数量设定在1024的基数下，同样用6个基准电路作为实验对象，分别在不同的时钟周期下运行故障模拟，在每一组同周期的对比实验中随机测试向量集是相同的。表格6-5为各实验电路故障模拟的运行结果，表中第三列为设定的时钟周期数，同样第四大列和第五大列分别为基于单核CPU和基于多核CPU下的故障模拟运行时间和故障覆盖率结果。

从运行时间上来看，在相同数量随机测试向量下随着时钟周期的倍增，同一电路下的故障模拟运行时间会增加，故障覆盖率同样也有缓慢增加，整体基于多核故障模拟与基于单核故障模拟时间平均加速比为2.65，故障覆盖率平均值为90.59%。2.65的故障模拟加速比也符合预期，在相同数量随机测试向量下随着时钟周期倍增，运行时间增加是因为周期增加会增加故障模拟运行次数，故障覆盖率也有缓慢增加，这是因为在4.4节和5.4节设计故障模拟器中，对于多个时钟周期的故障模拟算法是在第一个周期顺序往电路插入故障，然后在每一个故障电路下用所有的测试向量进行模拟，所以在第一个时钟周期我们可以检测完所有的组合电路部分的故障，而在之后的周期模拟完成后如果触发器的值发生改变则可检测到触发器门输出节点的故障。

总的来说，更多的测试向量会带来更好的故障覆盖率，增加时钟周期对故障覆盖率影响相对较小，基于多核CPU的逻辑电路故障模拟确实表现出一定的加速效果，改变测试向量数量或改变时钟周期对加速效果影响不大，其加速效果主要受限于故障模拟算法和多核CPU的线程数。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 电路标准 | 电路名称 | 时钟周期  （cycle） | 单核CPU | | 多核CPU | | 加速比 |
| Wall time（s） | Fault coverage | Wall time（s） | Fault coverage |
| ITC`99 | b17 | 1 | 197.58 | 80.27% | 53.84 | 80.27% | 3.67 |
| 2 | 207.11 | 83.63% | 58.34 | 83.63% | 3.55 |
| 4 | 215.47 | 84.31% | 62.46 | 84.31% | 3.45 |
| 8 | 230.96 | 85.74% | 68.53 | 85.74% | 3.37 |
| b20 | 1 | 71.09 | 94.56% | 25.30 | 94.56% | 2.81 |
| 2 | 74.93 | 95.93% | 27.05 | 95.93% | 2.77 |
| 4 | 79.85 | 96.17% | 30.02 | 96.17% | 2.66 |
| 8 | 83.04 | 96.37% | 33.48 | 96.37% | 2.48 |
| b21 | 1 | 79.20 | 94.26% | 29.77 | 94.26% | 2.66 |
| 2 | 84.14 | 94.89% | 32.24 | 94.89% | 2.61 |
| 4 | 88.11 | 95.41% | 34.28 | 95.41% | 2.57 |
| 8 | 95.51 | 96.32% | 37.90 | 96.32% | 2.52 |
| b22 | 1 | 237.59 | 93.42% | 97.77 | 93.42% | 2.43 |
| 2 | 239.68 | 94.32% | 101.56 | 94.32% | 2.36 |
| 4 | 241.55 | 94.78% | 105.02 | 94.78% | 2.30 |
| 8 | 253.28 | 95.25% | 106.42 | 95.25% | 2.38 |
| ISCAS`89 | s38584 | 1 | 52.04 | 85.57% | 21.15 | 85.57% | 2.46 |
| 2 | 56.07 | 86.78% | 24.59 | 86.78% | 2.28 |
| 4 | 61.46 | 87.06% | 27.32 | 87.06% | 2.25 |
| 8 | 65.89 | 88.21% | 28.28 | 88.21% | 2.33 |
| s38417 | 1 | 41.97 | 85.37% | 16.65 | 85.37% | 2.52 |
| 2 | 44.76 | 87.66% | 18.20 | 87.66% | 2.46 |
| 4 | 49.95 | 88.39% | 20.90 | 88.39% | 2.39 |
| 8 | 57.37 | 89.57% | 24.84 | 89.57% | 2.31 |

表6-5 不同时钟周期下故障模拟结果

## 6.4 本章小结

本章通过实验首先验证了故障模拟流程中故障生成、故障压缩以及可检测到故障的运行结果，确保了本研究故障模拟设计整体的正确性；然后以ISCAS`89和ITC`99标准下的电路为实验对象，通过模拟实验证明了基于多核CPU的故障模拟算法可以表现出符合理论预期的加速效果。

# 第7章 总结与展望

## 7.1 研究工作总结

本文主要对基于并行向量单固定型故障传播算法的逻辑电路故障模拟进行学习和研究，通过理论学习然后在Linux下搭建编程环境用C++语言代码实现了逻辑电路故障模拟，并将传统的故障模拟算法映射到了多核CPU体系架构，并利用OpenCL平台实现了基于多核CPU的逻辑电路故障模拟。最后在保证了本研究故障模拟算法各流程运行结果正确的前提下，通过实验验证了基于多核CPU的逻辑电路故障模拟加速效果，结果符合理论预期。本研究的不足之处在于由于自身学习能力有限，只研究了多核CPU体系结构，还未来得及研究在众核GPU体系结构下的加速效果。

## 7.2 论文创新点

本研究主要创新点在于采用基于任务并行方式，改进逻辑电路故障仿真算法，减少了内存开销，实现了对单个故障的并行模拟；在对电路进行层级化处理时将触发器等时序元件放在最后一个层级方便处理；针对将单线程算法映射到多核体系架构的问题，对已有数据结构进行改进，以便于更好地贴合多核体系架构；选用OpenCL作为多核编程平台，OpenCL是支持多种硬件设备的异质架构平台，具有很好的移植性。

## 7.3 下一步研究计划

在本文的研究基础上，可以进一步开展如下工作：

1.对故障模拟算法的各个环节进一步精细优化，减少模拟时间。

2.OpenCL平台是异质架构平台，即可以支持CPU和GPU等不同的硬件设备协同工作进行计算，下一步可以继续在GPU上实现故障模拟的并行，最后可以接着探索如何通过利用CPU与GPU进行协同工作达到对故障模拟最好的加速效果。

由于本人的专业水平有限，加之集成电路故障模拟问题研究需要大量的专业背景知识，具有不小的复杂性和难度，所以在本课题的研究中存在一些不足之处，望批评指正。

# 参考文献

# 攻读学位期间主要研究成果

一、发表论文

[1] 黄宇翔.通用图形处理器GPGPU在自动测试生成中的应用[J].集成电路应用,2020,37(05):34-35.

二、获奖情况

[1] “兆易创新杯”第十三届中国研究生电子设计大赛华南赛区一等奖

# 致谢

转眼间三年的研究生生活即将画上句号，我终于要从一名学生走向社会走向职场，将面临更多的人生挑战，但是我却感到充实和坚定，研究生三年我学到的不仅是知识，更学到了做人做事和面对困难的勇气。

首先，我要感谢我的导师赖李洋教授的悉心教育之恩。在我刚刚开启我的研究生生涯时，由于是跨专业报考，本专业的基础知识储备比较薄弱，是赖老师手把手教我如何去学习，首先督促我们养成早睡早起的良好作息习惯，从助教做起跟着本科师弟师妹们一同学习基础课程和做相关实验，为做课题研究打下了良好的基础，其次坚持学习弟子规和英语也磨去了我浮躁的心性，让我能够在实验室踏实地学习。在学习编程方面，赖老师也是悉心教导，让我养成了良好的编程习惯以及熟练地使用和理解了分布式版本控制系统，同时学到了很多Linux基础操作和运维知识，“工欲善其事，必先利其器”的思维习惯深刻地影响了我做研究的态度。在动手实践方面，赖老师会指导我们动手装系统和服务器相关设备，让我们观察晶圆实物，和自己录教学视频等；除此之外每周定期的学习汇报组会也锻炼了我们做演讲汇报的表达能力。赖老师是我走进学术研究的引导者，也是我的人生导师，在三年的学习时光中，赖老师谦逊的为人与严谨的治学态度深深地影响了我，我会继续努力不断提升自己做人做事的修养。

其次，我要感谢范衠老师、陈耀文老师、崔岩老师、魏楚亮老师和林建宇老师，感谢研究生院王风奇老师在我参加研会期间和生活上的帮助。我还要感谢史新宝师兄和张绮婷师姐，史新宝师兄无论是生活上还是求职上都给了我很大的支持，张绮婷师姐也在科研和生活上经常帮助和关心我。在三年的研究生生活中，我也认识了很多朋友，感谢同学王春臣、樊广文、程誉、魏思冉、周园园等，师弟邱立靖、吴观生、林晓泽，谢谢你们在学习和日常生活中给予的关心和支持。再者，我要感谢我的家人，他们总是在背后默默地支持和包容我，感谢父母的养育之恩，是你们从小教育我要为人善良、学有所得。我还要特别感谢方奕，谢谢从相识以来一直的陪伴、包容和支持。

最后，感谢汕头大学给予的成长环境，很幸运在这里遇见这么多美好同时见证了自己的成长，我自知自己身上仍有很多的缺点，但三年的研究生生活让我看到了可以凭借努力去一步步完善自己，让自己变得更好。即将进入职场和社会，随之会面对各种新的挑战和机遇，今年突遇疫情困在家的五个月也让自己有了更多的人生体会和思考，发现自己还有很多地方做的不够，我想借用毛主席的一句诗勉励自己：“雄关漫道真如铁，而今迈步从头越”。

# 本文专业术语中英文对照表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 中文 | 英文 | 中文 | 英文 | 中文 | 英文 |
| 故障模拟 | fault simulation | 自动测试向量生成 | ATPG | 时钟周期 | time cycle |
| 数字暴政 | tyranny of numbers | 故障字典 | fault dictionary | 时帧 | time frame |
| 无工厂芯片供应商 | fabless | 无故障电路 | good machine | 触发器 | flip-flop |
| 代工厂 | foundry | 中央处理单元 | CPU | 锁存器 | latch |
| 晶圆测试 | probe test | 图形处理单元 | GPU | 延时元件 | delay element |
| 封装测试 | board test | 加速处理单元 | APU | 位并行 | bit parallel |
| 系统级测试 | system test | 故障模型 | fault model | 上下文 | context |
| 单固定型故障 | single stuck-at fault | 桥接故障 | bridging fault | 命令队列 | command queue |
| 延时故障 | delay fault | 故障激活 | fault sensitization | 内核命令 | kernel |
| 故障传播 | fault propagation | 敏化路径 | sensitized path | 储存对象 | memory object |
| 顺序逻辑模拟 | in-order logic simulation | 事件驱动逻辑模拟 | event-driven logic simulation | 单指令多数据 | SIMD |
| 故障压缩 | fault collapsing | 等价故障 | equivalent fault | 线程 | thread |
| 等价故障集 | equivalent fault class | 无扇出区 | fan-out free region | 线程块 | thread block |
| 门输入 | fan-in | 门输出 | fan-out | 主机 | host |
| 顺序故障模拟算法 | in-order fault simulation algorithm | 并行故障模拟算法 | parallel fault simmulation algorithm | 并行向量单故障传播算法 | parallel-pattern single-fault propagation |
| 网格 | grib | 故障覆盖率 | fault coverage | 工作项 | work item |
| 电路主输入 | primary input | 电路主输出 | primary output | 工作组 | work group |
| 电路网表 | netlist | 主机内存 | host memory | 局部内存 | local memory |
| 多维索引 | NDRange | 全局内存 | global memory | 私有内存 | private memory |
| 常量内存 | constant memory | 纹理内存 | texture memory | 平台模型 | platform model |
| 执行模型 | execution model | 内存模型 | memory model |  |  |

1. #### [21] Rogers W A, Abraham J A. High level hierarchical fault simulation techniques[C]. conference on scientific computing, 1985: 89-97.

   [↑](#endnote-ref-2)
2. #### [22] Ghalaty N F, Aysu A, Schaumont P, et al. Analyzing and eliminating the causes of fault sensitivity analysis[C]. design, automation, and test in europe, 2014: 1-6.

   [↑](#endnote-ref-3)
3. #### [23] Gerd Meister. A Survey on Parallel Logic Simulation[J]. university of saarland department of computer science misra j, 1993.

   [↑](#endnote-ref-4)
4. #### [24] Prasad A V, Agrawal V D, Atre M V, et al. A new algorithm for global fault collapsing into equivalence and dominance sets[C]. international test conference, 2002: 391-397.

   [↑](#endnote-ref-5)
5. #### [25] Eslami Mohammad,Ghavami Behnam,Raji Mohsen,Mahani Ali. A survey on fault injection methods of digital integrated circuits[J]. Integration,2020,71(C).

   [↑](#endnote-ref-6)
6. #### [26] Pomeranz I, Reddy S M. Level of similarity: a metric for fault collapsing[C]. design, automation, and test in europe, 2004.

   [↑](#endnote-ref-7)
7. #### [27] Ishiura N , Yajima S . Linear time fault simulation algorithm using a content addressablememory[C]. European Design Automation Conference. 1992.

   [↑](#endnote-ref-8)
8. #### [28] Lee H K, Ha D S. an efficient, forword fault simualtion algorithm based on the parallel pattern single fault propagate[C]. international test conference, 1991: 946-955.

   [↑](#endnote-ref-9)
9. #### [29] Fink F, Fuchs K, Schulz M H, et al. Robust and nonrobust path delay fault simulation by parallel processing of patterns[J]. IEEE Transactions on Computers, 1992, 41(12): 1527-1536.

   [↑](#endnote-ref-10)
10. #### [30] Massengill L W , Baranski A E . Analysis of single-event effects in combinational logic-simulation of the AM2901 bitslice processor[J]. IEEE Transactions on Nuclear ence, 2000, 47(6):2609-2615.

    [↑](#endnote-ref-11)
11. #### [31] Chang K H , Browy C . Parallel Logic Simulation: Myth or Reality?[J]. Computer, 2012, 45(4):67-73.

    [↑](#endnote-ref-12)
12. #### [32] Brglex F . Combinational Profiles of Sequential Benchmark Circuits[J]. Proc.ieee Int.symp.circuit & System, 1989.

    [↑](#endnote-ref-13)
13. #### [33] Murakami T, Wada K, Okano S, et al. Parallel logic simulation on a workstation cluster[C]. pacific rim conference on communications, computers and signal processing, 1995: 268-271.

    [↑](#endnote-ref-14)
14. #### [34] Burch J R, Clarke E M, Mcmillan K L, et al. Sequential circuit verification using symbolic model checking[C]. design automation conference, 1990: 46-51.

    [↑](#endnote-ref-15)
15. #### [35] 龙川,宁涛.数字电路门级并行逻辑模拟[J].计算机工程与应用,2008(13):63-66.

    [↑](#endnote-ref-16)
16. #### [36] Gonsiorowski E, Carothers C D, Tropper C, et al. Modeling Large Scale Circuits Using Massively Parallel Discrete-Event Simulation[C]. modeling, analysis, and simulation on computer and telecommunication systems, 2012: 127-133.

    [↑](#endnote-ref-17)
17. #### [37] Alpert C J , Kahng A B . Recent directions in netlist partitioning: a survey[J]. Integration the Vlsi Journal, 1995, 19(1-2):1-81.

    [↑](#endnote-ref-18)
18. #### [38] 戴金玲,许爱强,王栋,唐小峰.针对逻辑门功能异常的VLSI故障模拟方法研究[J].国外电子测量技术,2016,35(09):24-28.

    [↑](#endnote-ref-19)
19. #### [39] Hadjitheophanous S, Neophytou S N, Michael M K, et al. Scalable parallel fault simulation for shared-memory multiprocessor systems[C]. vlsi test symposium, 2016: 1-6.

    [↑](#endnote-ref-20)
20. #### [40] Heyszl J, Nyberg R, Heinz D, et al. Enhancing Fault Emulation of Transient Faults by Separating Combinational and Sequential Fault Propagation[C]. great lakes symposium on vlsi, 2016: 209-214.

    [↑](#endnote-ref-21)
21. #### [41] Elangovan V K, Badia R M, Parra E A, et al. OmpSs-OpenCL Programming Model for Heterogeneous Systems[C]. languages and compilers for parallel computing, 2012: 96-111.

    [↑](#endnote-ref-22)
22. #### [42] Gaster B R , Howes L . OpenCL C++[M]. 2013.

    [↑](#endnote-ref-23)
23. #### [43] Lai L, Tsai K, Li H, et al. GPGPU-Based ATPG System: Myth or Reality?[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2020, 39(1): 239-247.

    [↑](#endnote-ref-24)
24. #### [44] Baker W I , Mahmood A , Carlson B S . Parallel event-driven logic simulation algorithms: tutorial and comparative evaluation[J]. IEEE Proceedings-Circuits, Devices and Systems, 1996, 143(4):177-0.

    [↑](#endnote-ref-25)
25. #### [45] Voorakaranam R, Chakrabarti S, Hou J, et al. Hierarchical specification-driven analog fault modeling for efficient fault simulation and diagnosis[C]. international test conference, 1997: 903-912.

    [↑](#endnote-ref-26)
26. #### [46] Kochte M A, Schaal M, Wunderlich H, et al. Efficient fault simulation on many-core processors[C]. design automation conference, 2010: 380-385.

    [↑](#endnote-ref-27)