第37卷第2期 电 子 与 信 息 学 报 Vol.37 No.2
2015年2月 Journal of Electronics & Information Technology Feb. 2015

一种基于后缀排序快速实现 Burrows-Wheeler 变换的方法

李 冰 龙冰洁* 刘 勇
(东南大学集成电路学院 南京 210000)

摘 要：近年来，Bzip2 压缩算法凭借其在压缩率方面的优势，得到了越来越多的应用，Bzip2 的核心算法是 Burrows-Wheeler 变换(BWT)，BWT 能有效的将数据中相同的字符聚集到一起，为进一步压缩创造条件。在硬件实现 BWT 时，常用的基于后缀排序的算法能有效克服 BWT 消耗存储资源大的问题，该文对基于后缀排序实现 BWT 的方法进行了详细分析，并且在此基础上提出了一种快速实现 BWT 的方法——后缀段算法。仿真结果表明后缀段算法在处理速度上比传统的基于后缀排序的算法有很大的提高。

关键词：信号处理；数据压缩；Bzip2；Burrows-Wheeler 变换；后缀排序
中图分类号：TN492 **文献标识码**：A **文章编号**：1009-5896(2015)02-0504-05
DOI: 10.11999/JEIT140232

A Fast Algorithm for Burrows-Wheeler Transform Using Suffix Sorting

Li Bing Long Bing-jie Liu Yong
(*College of Integrated Circuit*, *Southeast University*, *Nanjing* 210000, *China*)

Abstract: Bzip2, a lossless compression algorithm, is widely used in recent years because of its high compression ratio. Burrows-Wheeler Transform (BWT) is the key factor in Bzip2. This method can gather the same symbols together. The traditional methods which are based on suffix sorting used in implement of BWT in hardware can solve the problem of memory consumption effectively. Detail analysis of BWT algorithm based on suffix sorting is given and a new method — Suffix segment method is presented in this paper. Experimental results show that the proposed method can much decrease BWT time consumption without increasing memory consumption much.

Key words: Information processing; Data compress; Bzip2; Burrows-Wheeler Transform (BWT); Suffix sorting

1 引言

数据压缩在信息技术中占有很重要的地位，传统的 LZ 系列和 ZIP 系列压缩算法利用了数据内部的重复性，对数据重复性进行记录，然后对数据进行编码处理，从而得到压缩数据。这些压缩算法的压缩率在一定程度上取决于数据内部的重复性。Bzip2 与这些传统的算法不同，Bzip2 的核心变换算法 Burrow-Wheeler 变换 (Burrows-Wheeler Transform, BWT)是一种不依赖于数据内部重复性的变换方法，它能将数据内部相同的字符聚集到一起，这使得 Bzip2 的压缩率基本不会受到数据内部重复性的影响，Bzip2 压缩数据主要由游程编码(Run-Length Encoding, RLE)，BWT，前移变换(Move-To-Front, MTF)以及霍夫曼(Huffman)编码 4 个步骤组成，Bzip2 的压缩性能比其它传统的压缩算法都要高，但是耗费的压缩时间比较长[1–4]。

Bzip2 压缩数据过程中耗时最多的是 BWT，快速实现 BWT 能有效地减少 Bzip2 压缩数据的时间。BWT 由 Burrows 和 Wheeler 在 1994 年提出[5]，这种算法的核心思想是对字符串轮转后得到的字符矩阵进行排序和变换，得到的结果序列中相同的字符在很大程度上聚集到了一起，这样的特性很适合使用通用的统计压缩模型(Huffman 编码、PPM 算法等)进行压缩，并得到理想的压缩率[5–7]。BWT 原始算法是通过矩阵完成的，需要占用大量存储资源，交织(Weavesort, WS)编码算法的出现解决了这个问题，但是这种算法处理的速度太慢[8–11]，基于后缀排序的算法(Suffix sorting)比 WS 算法在速度上提高了很多，其中双向搜索算法(Bi-directional search)进一步提高了处理速度[12–16]。本文通过对基于后缀排序实现 BWT 算法的研究，结合后缀排序以及 BWT 自身的特点，提出了一种快速实现 BWT 的算法，称其为后缀段算法，仿真结果证实了这种算法能在存储资源消耗与双向搜索算法基本相当的情况下大大减少 BWT 的时间。

2014-02-24 收到，2014-07-17 改回
十二五国家科技支撑计划(2013BAJ05B03)资助课题
*通信作者：龙冰洁 longbj1107@sinacom

2　BWT 与后缀排序

对于一个包含 N 字节的字符串 S，对其进行 BWT 由 3 个步骤组成：

步骤 1　构造一个 $N \times N$ 矩阵 $\boldsymbol{M}$ ($\boldsymbol{M}$ 中的每一行分别是 S 左移 $0,1,\cdots,n-1$ 个字符)；

步骤 2　对矩阵 $\boldsymbol{M}$ 行向量按字典顺序进行升序排序，得到新的矩阵 $\boldsymbol{M}^{\mathrm{T}}$；

步骤 3　输出 $\boldsymbol{M}^{\mathrm{T}}$ 中最后一列(记为 L)以及原字符串 S 在 $\boldsymbol{M}^{\mathrm{T}}$ 中的行号 index。

以 abraca 为例，其 BWT 变换过程如图 1 所示。

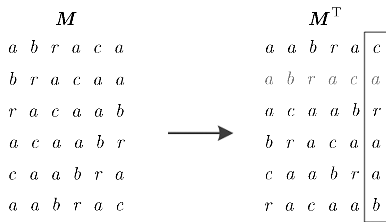

图 1　BWT 实例

abraca 的 BWT 结果 $L=$ caraab , index $=1$ 是用于进行 BWT 反变换的。BWT 之后，数据长度并没用缩短，但相同字符很大程度上被排列到了一起。

对于字符串 $S: X_1X_2\cdots X_N$，将它的子字符串 $X_iX_{i+1}\cdots X_N$ 称为后缀 S_i，后缀之间存在着大小关系，假设有后缀 X_i 与 X_j，定义其大小关系为：

(1)如果 $X_i > X_j$，则 $S_i > S_j$；

(2)如果 $X_i < X_j$，则 $S_i < S_j$；

(3)如果 $X_i = X_j$，则继续比较 S_{i+1} 和 S_{j+1}，比较规则同(1)和(2)。

对于字符串 $S: X_1X_2\cdots X_N$，在其后面添加一个 \$，得到 $S': X_1X_2\cdots X_N\$$ 其中 \$ 需要满足条件 $\$ > X_t, t=1,2,\cdots,N$，即 \$ 要比 S 中所有字符的字典序都要大，将 S' 的所有后缀进行升序排序，同样以 abraca 为例，首先在其后面添加一个 \$，这样这个字符串就有 7 个后缀，分别是 S_1 : abraca\$, S_2 : braca\$, S_3 : raca\$, S_4 : aca\$, S_5 : ca\$, S_6 : a\$, S_7 : \$ 根据后缀大小的比较规则，这些后缀的升序排列结果为 $S_1S_4S_6S_2S_5S_3S_7$，现在用每个后缀在 S' 中的前一个字符代替，得到 $X_7X_3X_5X_1X_4X_2X_6$，即 \$rcaaba，同样，参照图 1 所示的方法对 S' 进行 BWT，可以发现其 BWT 结果也是 \$rcaaba，也就是说只要进行 BWT 的字符串满足最后一个字符的字典序比其它出现在字符串中的字符的字典序都大这个条件，那么就可以用后缀排序来得到它的 BWT 结果，实际上满足这个条件的字符串很少，但是一般在读入数据流的时候，都会默认添加一个文件结束符(End of File, EOF)，而 EOF 恰好是满足这个条件的，所以后缀排序的时候带上 EOF 就可以了。

对于一个字符串 $S: X_1X_2\cdots X_N$ 对其后缀进行排序，从第 1 个后缀 $S_1(S)$ 开始处理，直到最后一个后缀 $S_N(X_N)$，处理后缀 S_i 相当于是都在 $S_1S_2\cdots S_{i-1}$ 的升序排列集合中插入 S_i，用 T_{i-1} 来表示这个升序排列集合，将这样的集合称为后缀链表。当所有后缀完成排序后，得到后缀链表 T_N，将 T_N 中的所有后缀用它在 S 中的前一个字符(图 1 中矩阵 $\boldsymbol{M}^{\mathrm{T}}$ 的最后一列)替换，即后缀 S_i 用字符 X_{i-1} 替换(S_1 用 X_N 替换)，这样得到的字符串就是 S 的 BWT 结果，这就是后缀排序实现 BWT 的原理，也是基于后缀排序实现 BWT 的基本算法。

在后缀排序中将后缀 S_i 插入到后缀链表 T_{i-1} 的过程中，需要将 S_i 与 T_{i-1} 中所有后缀进行比较，从而确定 S_i 在 T_{i-1} 中的位置，称这个过程为搜索。以降序搜索为例，当 T_{i-1} 中所包含的后缀很多，并且 S_i 比 T_{i-1} 中绝大多数后缀都要小的时候，搜索过程耗费的时间是很长的，这种情况下采用升序搜索就能很快确定 S_i 在后缀链表中的位置，于是基于后缀排序实现 BWT 的双向搜索算法也很自然地被提出，就是在后缀链表中升序搜索和降序搜索同时进行，只有有一个方向上找到 S_i 对应的位置，即停止搜索，并插入 S_i，然后更新相应的信息。但是当数据块很大，处理过程中 T_{i-1} 包含的后缀越来越多，而且 S_i 处于 T_{i-1} 的中间位置的时候，即使采用双向搜索算法，耗费的时间也是很长的。

3　后缀段算法

仔细分析一下后缀排序的特点，可以发现有很多特点是可以利用的。

对于字符串 S，定义集合 $P_i: \{S_j : j<i,\ X_i = X_j, S_{i+1} < S_{j+1}\}$，它表示当前后缀链表 T_{i-1} 中首字符与 S_i 的首字符 X_i 相同并且比 S_i 小的所有后缀集合，在插入 S_i 的过程中会出现以下 3 种情况：

(1) T_{i-1} 中存在以 X_i 为首字符的后缀 S_i，并且也满足 $X_j = X_i$, $S_{j+1} < S_{i+1}$，这时 P_i 不是空集；

(2) T_{i-1} 中存在以 X_i 为首字符的后缀 S_i，但是不满足 $X_j = X_i$, $S_{j+1} < S_{i+1}$，这时 P_i 是空集；

(3) T_{i-1} 中不存在以 X_i 为首字符的后缀，这时 P_i 也是空集。

以降序搜索为例，在后缀排序过程中，情况(1)出现的可能性最大，出现情况(1)时，需要在 P_i 中搜索并插入 S_i；对于情况(2)和情况(3)，需要有更进一步的分析，参照双向搜索算法的原理以及集合 P_i，同样可以定义集合 $P_i: \{S_j : j<i, X_i = X_j, S_{i+1} >$

$S_{j+1}\}$，它表示当前后缀链表 T_{i-1} 中首字符与 S_i 的首字符 X_i 相同并且比 S_i 大的所有后缀集合，出现情况(2)时，检测 Q_i 是否为空集，如果 Q_i 不是空集，则在 Q_i 中搜索并插入 S_i 。

假设 P_i 和 Q_i 中的后缀都是按照升序排列的，定义集合 $R_i:\{P_i,Q_i\}$，表示 T_{i-1} 中首字符与后缀 S_i 的首字符相同的所有后缀的集合，称其为后缀段，并且将 S_i 的首字符 X_i 称为后缀段 R_i 的段首字符。显然，根据后缀大小的判断规则，在后缀排序的过程中，后缀链表由0~256个后缀段(假设文件是以ASCⅡ码的形式读入)按照段首字符升序排列组成。当出现情况(1)和情况(2)时，R_i 不是空集，在插入后缀 S_i 时，只要能确定以 X_i 为段首字符的后缀段 R_i 在后缀链表中的位置，接下来在 R_i 中搜索并插入 S_i 就可以了。当出现情况(3)时，R_i 是空集，这时需要做的就是找到段首字符与 X_i 距离最近的段，然后根据大小情况将 S_i 插入到对应的位置。

通过以上分析可以知道在后缀排序过程中，有3条信息是可以利用的：(1) R_i 是否已经存在于当前的后缀链表 T_{i-1} 中，这个信息可以用一个数组appear 记录，appear[i]=1 表示 R_i 已经存在于后缀链表中，appear[i]=0 表示 R_i 还不存在；(2)当前 R_i 中最大的后缀 max[i], max[i]里面存放的是当前 R_i 中最大的后缀在后缀链表中的地址；(3)当前 R_i 中最小的后缀 min[i], min[i] 里面存放的是当前 R_i 中最小的后缀在后缀链表中的地址，信息(2)和(3)用于确定 R_i 在后缀链表中的位置。在后缀排序过程中后缀链表结构如图2所示。

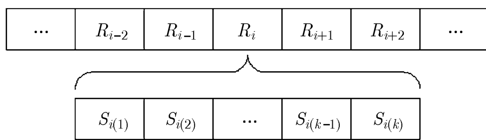

图2 后缀链表结构

综合以上对后缀排序特点的分析，可以总结出后缀段实现 BWT 的步骤：首先初始化后缀链表为空，接下来从后缀 S_i 开始，一直到后缀 S_N，每次向后缀链表中插入一个后缀。插入后缀 S_i 的步骤如图3所示：首先检查 appear[i]的值，如果 appear[i]=1，说明对应的 R_i 已经存在，接下来在 R_i 内进行双向搜索并插入 S_i；如果 appear[i]=0，说明对应的 R_i 不存在，这时需要查找距离最近的后缀段，根据最近后缀段的大小确定 S_i 所在的位置并更新后缀链表和相应信息。

查找距离最近的段的步骤如图4所示。查找距

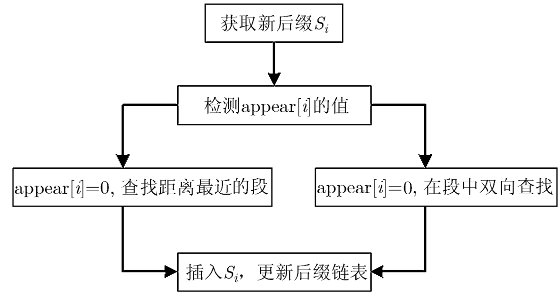

图3 插入后缀 S_i 的流程

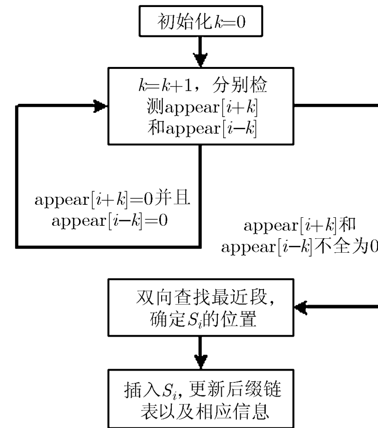

图4 查找距离最近的段流程

离最近的段是一个双向的过程，从 appear[i+1]和appear[i-1]开始依次往两边扩散，直到 appear[t]的值为1，如果 $t>i$，S_i 就应该插入到段 R_i 中最小后缀的左边；如果 $t<i$，S_i 就应该插入到段 R_i 中最大后缀的右边。插入 S_i 后，需要更新后缀链表以及相应的 max[i]和 min[i]的值，且令 appear[i]=1。

在段 R_i 中双向搜索的步骤如图5所示：首先根据 max[i]min[i]的值确定 R_i 在后缀链表中的位置，接下来从 max[i]和 min[i]开始依次在两个方向上往段内取相邻后缀与 S_i 进行比较，直到确定 S_i 在 R_i 中的位置，插入 S_i，然后更新后缀链表以及相应的max[i]和 min[i]的值。

在所有后缀都已经插入到后缀链表后，把后缀链表中的所有后缀用它在 S 中的前一个字符替换，这就是基于后缀段实现 BWT 的方法。

对于硬件实现来说，存储资源是十分宝贵的资源，用硬件实现很多经典的算法时，存储资源往往成为瓶颈，一个算法是否能够很好地用硬件实现也很大程度上与其所消耗的存储资源有关，第2节曾提到用基于后缀排序的算法来实现 BWT 比起原始算法最大的优势就是节省了大量的存储资源，但是在变换速度上仍然显得不够，后缀段算法利用了后缀排序算法的特点，在增加了一些存储资源的情况下，大大提高了变换速度。

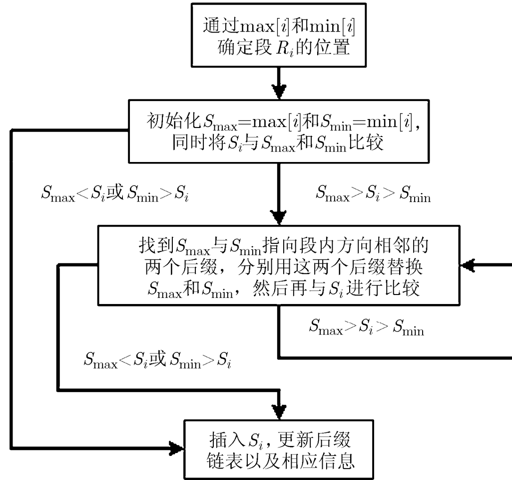

图 5 在段 R_i 中双向搜索流程图

对于一个长度为 N 字节，包含 T 个不同的字符的字符串 S，采用后缀段算法实现 BWT 所消耗的主要存储资源为(假设文件都是以 ASCⅡ码形式读入)：

(1)位宽为 8，深度为 N 的数组，用于存放字符串 S；

(2)位宽为$\lceil \log_2 N \rceil$，深度为 N 的数组，用于存放后缀链表(后缀链表的每个位置上存放的是处于该位置的后缀在字符串 S 中的地址)；

(3)位宽为 1，深度为 256 的数组，用于记录以 ASCⅡ值为 0~255 的字符为段首字符的后缀段是否存在于后缀链表中；

(4)2 个位宽为$\lceil \log_2 N \rceil$，深度为 256 的数组，分别用于记录每个后缀段中的最大后缀 max[i]和最小后缀 min[i](存放的是该后缀在后缀链表中的地址)。

4　实验结果分析

对基于后缀排序实现 BWT 的 3 种算法进行了仿真测试，测试代码是用 Verilog 硬件语言进行设计的，在 Modelsim 软件下面进行仿真。用于测试的数据是一系列长度不同的 0~255 的随机数，3 种基于后缀排序实现 BWT 的算法在处理这些数据时所消耗的主要存储资源如表 1 所示，完成 BWT 的时间如表 2 所示。

表 1 给出了 3 种基于后缀排序实现 BWT 算法

表 1　3 种算法实现 BWT 消耗的存储资源(bit)

算法	主要消耗的存储资源
基本算法	$(8+\lceil \log_2 N \rceil)\times N$
双向算法	$(8+\lceil \log_2 N \rceil)\times N$
后缀段算法	$(8+\lceil \log_2 N \rceil)\times N+\lceil \log_2 N \rceil\times 512$

表 2　3 种算法完成 BWT 的时间(ms)

数据长度(byte)	基本算法	双向算法	后缀段算法
2k	1.10	0.90	0.37
4k	4.50	3.40	0.86
8k	17.80	14.00	2.30
16k	–	54.00	6.80
32k	–	270.00	23.00
64k	–	900.00	81.00

的存储资源，3 种算法的主要消耗在于后缀链表中记录该处后缀在 S 中的地址，这是区分后缀的唯一标识，另外，在后缀段算法中，增加了$\lceil \log_2 N \rceil\times 512$ bit 的消耗，这个消耗用于记录后缀段在后缀链表中的位置，是提高排序速度必不可少的。

表 2 给出了 3 种基于后缀排序实现 BWT 算法的速度结果，表中数据长度的单位是字节，后面 3 列数据表示该算法处理对应大小的数据时所用的时间，单位是 ms。这里由于采用基本算法处理 16k 字节以上大小的文件时耗时太长，所以没有给出具体数据。

从测试结果数据上可以得出一些结论：当数据的长度增加一倍时，基本算法完成后缀排序的时间大概会增加 3 倍；双向算法所用的时间大概也会增加 3~4 倍；后缀段算法所用的时间大概会增加 2~3 倍。在数据长度相同的情况下，双向算法完成后缀排序的时间比基本算法减少了 20%，后缀段算法完成后缀排序的时间比起双向算法则有了更加明显的减少，随着数据长度的增加，后缀段算法完成后缀排序的时间比起双向算法减少的幅度也在增加，当数据长度为 32k 时，后缀段算法完成后缀排序所用的时间只是双向算法的十分之一不到，而且可以推测当数据长度进一步增加时，后缀段算法的优势会更加明显。

5　结束语

本文针对 Bzip2 压缩算法里面的 BWT 部分，对其原理算法进行了介绍，对基于后缀排序的 BWT 算法进行了详细的分析，利用后缀排序中后缀链表的组成特点提出了后缀段算法，虽然比传统的基本算法和双向搜索算法消耗更多的存储资源，但是在处理速度上得到了明显的提高。测试结果表明在处理同样长度的数据时，后缀段算法能明显减少处理时间，在数据长度从 2k 增加到 64k 时，后缀段算法与双向算法完成后缀排序所用的时间比从 1:3 逐渐减小到 1:10 不到，并且时间减少的幅度在不断增加。在后缀段中进行双向搜索的优化是进一步研究的重点。

参 考 文 献

[1] Julian Seward, Bzip2[OL]. http://en.wikipedia.org/wiki/Bzip2, 2010.9.20

[2] Szecówka P M, and Mandrysz T. Towards hardware implementation of bzip2 data compression algorithm [C]. Proceedings of the Mixed Design of Integrated Circuits and Systems, Lodz, Polland, 2009: 337–340.

[3] Sun Wei-feng, Zhang Nan, and Mukherjee A. Dictionary-based fast transform for text compression[C]. Proceedings of the International Conference on Information Technology: Computers and Communications, Nanjing, China, 2003: 176–182.

[4] Baloul F M, Abdulah M H, and Babikir E A. ETAOSD: static dictionary-based transformation method for text compression [C]. Prioeedings of the International Conference on Computing, Electrical and Electronic Engineering, Khartoum, Sudan, 2013: 384–389.

[5] Burrows M and Wheeler D. A block-sorting lossless data compression algorithm[R]. SRC Research Report 124, Digital Systems Research Center, Palo Alto, CA, USA, 1994.

[6] Arnsvut Z. Move-to-front and inversion coding[C]. Data Compress Conference, Snowbird, USA, 2000: 193–202.

[7] Effros M. PPM performance with BWT complexity: a fast and effective data compress algorithm[J]. *Proceedings of the IEEE*, 2000, 88(11): 1703–1712.

[8] Martínez J, Cumplido R, and Feregrino C. A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation[C]. Proceedings of the International Conference on Reconfigurable Computing and FPGAs, Puebla City, Mexico, 2005: 28–30.

[9] Kong J M, Ang L M, and Seng K P. Low-complexity two instructions set for suffix sort in Burrows-Wheeler transform[C]. Proceedings of the International Conference on Advanced Computer Science Applications and Technologies, Kuala Lumpur, Malaysia, 2012: 181–186.

[10] Arming S, Fenkhuber R, and Handl T. Data compression in hardware — the Burrows-Wheeler approach[C]. Proceedings of the Design and Diagnostics of Electronic Circuits and Systems (DDECS), Vienna, Austria, 2010: 60–65.

[11] Grajeda V Z, Uribe C F, and Parra R C. Parallel hardware/aoftware architecture for the BWT and LZ77 lossless data compression algorithms[J]. *Computation System*, 2006, 10(2): 172–188.

[12] Sadakane K. A modified Burrows-Wheeler transformation for case-insensitive search with application to suffix array compression[C]. Proceedings of the Data Compression Conference, Snowbird, USA, 1999: 29–31.

[13] Hayashi S and Taura K. Parallel and memory-efficient Burrows-Wheeler transform[C]. Proceedings of the IEEE International Conference on Big Data, Silicon Valley, USA 2013: 43–50.

[14] Zhao Zhi-heng, Yin Jian-pin, and Xiong Wei. GPU-accelerated Burrows-Wheeler transform for genomic data[C]. Proceedings of the 5th International Conference on BioMedical Engineering and Informatics, Chongqing, China, 2012: 889–892.

[15] Cheema U I and Khokhar A A. High performance architecture for computing Burrows-Wheeler transform on FPGAs[C]. Proceedings of the International Conference on Reconfigurable and FPGAs, Cancun, Mexico, 2013: 1–6.

[16] Baron D and Bresler Y. Antisequential suffix sorting for BWT-based sata sompression[J]. *Computers*, 2005, 54(4): 385–397.

李 冰： 男，1963 年生，教授，博士生导师，研究方向为现场集成系统与信息专用集成电路设计、数据压缩等.

龙冰洁： 男，1988 年生，硕士生，研究方向为嵌入式系统、数据压缩.

刘 勇： 男，1979 年生，博士生，研究方向为集成电路设计.