密级:_____

中国科学院大学
University of Chinese Academy of Sciences

博士学位论文

低功耗音频专用指令集处理器关键技术研究

作者姓名：薛金勇

指导教师：黑勇　研究员　中国科学院微电子研究所

学位类别：工学博士

学科专业：微电子学与固体电子学

培养单位：中国科学院微电子研究所

2013 年 5 月

Research on Key Technologies of ASIP for Low Power Audio Processing

By

Xue Jinyong

A Dissertation Submitted to

University of Chinese Academy of Sciences

In partial fulfillment of the requirement

For the degree of

Doctor of Engineering

Institute of Microelectronics of Chinese Academy of Sciences

May, 2013

声明

我声明本论文是我本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，本论文中不包含其他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

作者签名：[signature]　　日期：2013.6.1

论文版权使用授权书

本人授权中国科学院微电子研究所可以保留并向国家有关部门或机构送交本论文的复印件和电子文档，允许本论文被查阅和借阅，可以将本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、缩印或扫描等复制手段保存、汇编本论文。

（保密论文在解密后适用本授权书。）

作者签名：[signature]　　导师签名：[signature]　　日期：2013.6.3

摘要

专用指令集处理器（Application Specific Instruction set Processor，ASIP）面向应用定制处理器结构，兼具灵活性、良好的计算性能和低功耗等特点，随着嵌入式的迅速发展，被越来越多产品在开发时采用。本文基于 RISC 处理器核 FlexEngine，以音频数字助听器系统为应用，对低功耗专用指令集处理器设计展开了研究，主要的研究工作和取得的成果包括：

1. 提出了一种专用指令集处理器的设计方法。基于算法的汇编级代码分析，识别可能的扩展指令，将频繁使用的指令组合与指令序列以专用指令或者协处理器的形式添加到处理器的数据通路。针对数字助听器算法，添加了位反寻址、零开销循环、对数、开方、倒数、蝶形运算、睡眠模式、数据并行加载等专用指令；自定制的结构使处理器能够与应用匹配，更加高效的执行数字助听器算法，减少了算法 62.8%的运行时间。

2. 提出了专用指令集处理器的软件开发工具设计方法。基于 LLVM 与 Clang 的组合设计编译器，模块化、结构清晰、可移植性强、易于生成代码优化。ASIP 的软件开发工具，可用于 ASIP 设计时的应用代码分析与设计评估，同时极大的方便了基于 ASIP 的应用开发。对于面向不同应用的 ASIP 设计，需要指令扩展时，可以快速完成软件开发工具的修改，生成 ASIP 的软件工具。

3. 提出了专用指令集处理器的低功耗设计方法。在 ASIP 设计的软件级、系统结构级、寄存器传输级、以及逻辑级，综合运用算法优化、低功耗的工作模式、指令集优化、循环缓存、存储器分割、门控时钟、操作数隔离、等多种低功耗设计技术，有效的降低了系统的功耗。优化后的处理器在执行数字助听器程序时，运行时间减少了 67.8%，功耗降低了 79%。

本文以数字助听器系统作为音频处理的典型应用，以 FlexEngine 为基本指令集处理器核，主要研究了低功耗的 ASIP 设计。设计了 ASIP 的软件开发工具，用于设计评估与提供设计反馈；基于算法的汇编级代码分析，识别可能的扩展指令，并映射到 ASIP 的数据通路；基于标准单元，在 ASIP 设计的软件级、系统结构级、寄存器传输级、以及逻辑级，综合运用多种低功耗设计技术，完成了数

字助听器的低功耗 ASIP 设计。最终，基于 TSMC 130nm 工艺完成了 ASIP 设计流片，经过测试，该系统能够很好的完成数字助听器功能，并且低功耗效果显著，工作在 8MHz 频率、1.2V 工作电压时，处理器功耗约 0.963mW。

关键词：专用指令集处理器，软件开发工具，数字助听器，指令集优化，低功耗

Abstract

Xue Jinyong (Microelectronics and Solid-State Electronics)

Directed by Professor Hei Yong

Application Specific Instruction Set Processor (ASIP) can offer greater flexibility, higher computing performance and lower power consumption, through architectural customizations tailored to the applications. With the rapid development of embedded system, ASIPs are widely used in the development of products. This dissertation deals with research on the low-power ASIP design tailored to the application of digital hearing aid, based on FlexEngine, a DSP core using RISC. The main work and results are as follows:

1. An ASIP design method was proposed. Based on assembly level code profiling, possible instruction set extensions and accelerators were identified. Frequently used operations were merged into one customized instruction or accelerator and mapped to data path inside the ASIP hardware model. Through analyzing the algorithms of digital hearing aid system, customized instruction set extensions and dedicated hardware accelerator were added, for example bit-reverse addressing mode, zero-overhead looping, log, square root, reciprocal, butterfly, sleep mode, and parallel memory accessing. The customized architecture tailored to digital hearing aid system increased the power efficiency of FlexEngine and reduced 62.8% of the execution time.

2. A method to design software development tools for ASIP was proposed. Developing compiler based the combination of LLVM compiler framework and Clang frontend has many advantages, such as modular design, clear structure, high portability, and widely optimization strategies. The software development tools for ASIP is used for application analysis and design evaluation when ASIP designing, and provide convenience for application development based on ASIP. For different applications, when adding new instructions, the software development tools can be updated quickly.

3. A low power ASIP design method was proposed. Several ASIP power optimization techniques have been applied at the ASIP software level, the system level, the register transfer level, and also at the logic level. These power

optimization techniques are algorithm optimization, sleep mode, Instruction set optimization, loop cache, memory partitioning, clock-gating, and operand isolation. Optimized ASIP design reduced 67.8% of the execution time and decreased 79% of the power consumption when running the program of digital hearing aid system.

This dissertation mainly focused on the low power ASIP design, taking digital hearing aid system as the typical application of audio processing and based on the programmable base processor FlexEngine. The ASIP software development tools were developed, which were used to evaluate the design and provide design feedback. Possible instruction set extensions and accelerators were identified and mapped to the data path of ASIP, based on the assembly level code profiling. Power optimization techniques have been adopted at the ASIP software level, the system level, the register transfer level, and also at the logic level, based on the standard cell. Finally, the low power ASIP design was implemented in TSMC 130nm process. Measured results show that, the system can perform hearing aid well and consumes very low power. The power dissipation is 0.963mW at 1.2V supply voltage, when the system works at 8MHz clock frequency.

Key Words: ASIP, Software development tools, Digital hearing aids, Instruction set optimization, Low power

目录

第1章 绪论

1.1 研究背景与研究意义

目前，全世界大约有 12%[1]的人存在听力问题，我国大约有 7000 万[2]听障患者，60 岁以上老人中有 30%以上受到听力损失的困扰。助听器（Hearing Aid, HA）对听力损失患者进行听力补偿，是大部分听障患者恢复听力最有效的手段[2]，具有庞大的医疗市场，每年全球销售总额近 100 亿美元，且保持着高达 12%的年增长率。

过去，传统的助听器限于信号处理的功耗问题均采用模拟电路，随着数字信号处理理论的广泛应用以及集成电路的迅速发展，数字助听器[3]成为助听器市场的主流，以美国的助听器市场为例，至 2006 年数字助听器已经占据了市场的 93%以上[4]，如图 1-1。

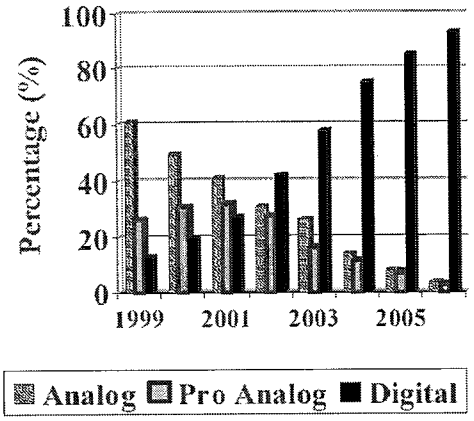


图 1-1 美国助听器市场，1996-2006[4]

数字助听器由单个 1.35V 锌空电池供电，电量一般为 100~200mAh，要求设计满足面积小于 25mm^2，至少持续供电 50 个小时[5]，因此数字助听器对功耗要求非常严格；同时新的算法层出不穷，功能的改进或者添加，需要设计提供良好的灵活性与计算性能。基于 ASIC 的设计功耗低，但无法满足系统快速升级的特点；基于通用处理器（General programmable processor，GPP）的设计灵活，但无法满足数字助听器低功耗的要求。因此，基于低功耗高性能 DSP 的数字助听器实现，因其灵活性成为研究的热点。专用指令集处理器（Application Specific

Instruction set Processor，ASIP）[6][7][8][9]具有精心设计的指令集，可以提供极大的灵活性，通过软件编程即可实现系统升级；通过针对数字助听器算法添加专用指令或者硬件加速单元，能够获得较好的性能和较低的功耗；因此专用指令集处理器能够很好的满足数字助听器的设计需要。

1.1.1 专用指令集处理器

ASIP 是为一种应用或一类应用专门设计的处理器，通常由一个基本指令集处理器、自定制的指令集扩展与硬件加速单元组成（如图 1-2）。精心设计的指令集，使得 ASIP 具有良好的可编程性；自定制的专用指令与硬件加速单元，使得 ASIP 具有较好的性能和较低的功耗。因此 ASIP 作为 ASIC 与 GPP 的折中（如图 1-3），兼具 GPP 的灵活性与 ASIC 的良好计算性能和低功耗特点，随着嵌入式的迅速发展，被越来越多产品在开发时采用。ASIP 可以有效地应用于多种嵌入式系统，如数字信号处理、视频监控、自动控制、以及医疗电子系统等[10]。图 1-4 显示了自 1995 年至 2010 年的 DSP 市场份额，可见 ASIP 的市场份额在逐年增长，且已经超过了 ASIC 与 GPP 的市场份额总和[11]。因此对于专用指令集处理器设计的研究具有极大的研究价值与应用前景，针对数字助听器的 ASIP 设计同样具有广阔的市场前景。

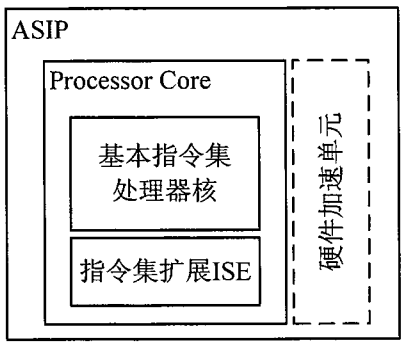


图 1-2 典型的 ASIP 结构

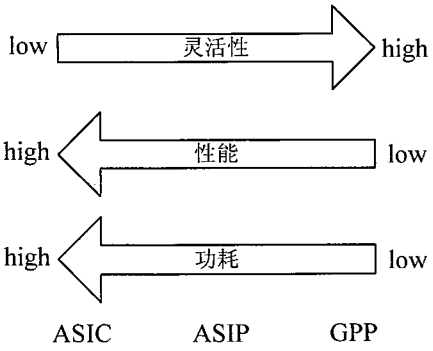
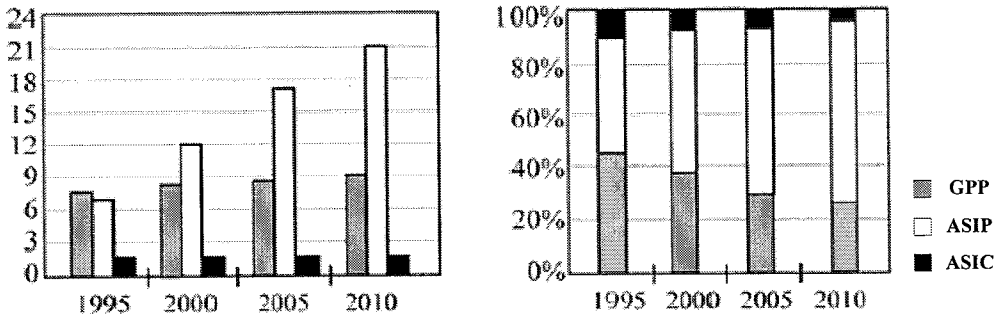


图 1-3 ASIC，ASIP 与 GPP



（a）DSP 销售额/十亿美元　　（b）DSP 市场份额

图 1-4 DSP 市场份额(FreehandDSP, Sweden)

1.2 专用指令集处理器的设计方法

ASIP 设计是一个软硬协同设计的迭代过程[12][13]，如图 1-5 所示。

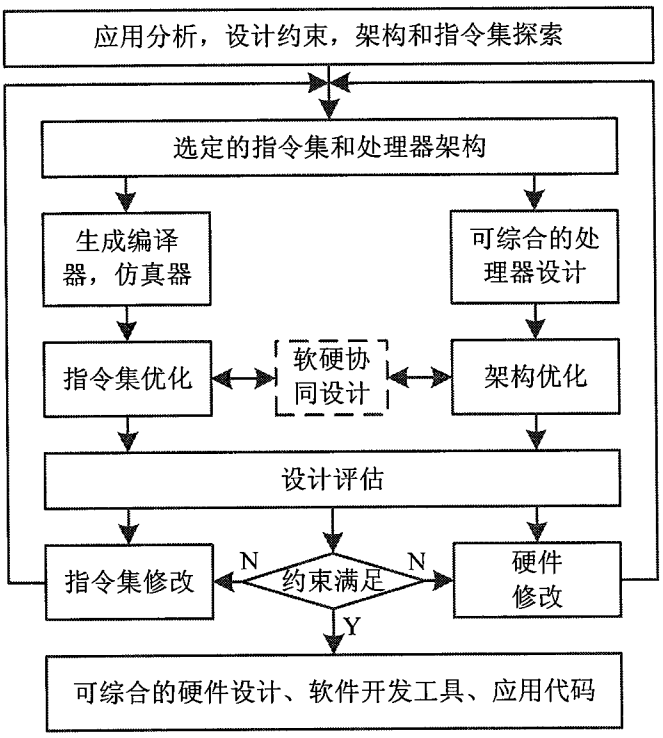


图 1-5 软硬协同的 ASIP 设计流程

首先通过对给定的应用与设计约束进行分析，选定合适的指令集与处理器结构，获得相应的软件开发工具以及可综合的处理器设计，通常不是最优设计，甚至不满足设计约束。通过对应用程序进行仿真与代码分析，进行指令集与处理器架构优化，并对设计进行评估。如此经过若干次迭代，直至设计满足约束，得到最终的软件开发工具、可综合的处理器设计以及可执行的应用程序代码。

ASIP 面向应用定制设计，可以获得良好的计算性能和较低的功耗，设计过程通常包括：应用分析，架构探索，指令集生成，代码综合和硬件综合[14]，如图 1-6 所示。应用分析以高级语言编写的应用程序作为输入，寻找程序的热点（hot spots），即消耗最多时间和功耗的部分。结构设计探索阶段根据应用分析结果和设计约束，评估处理器结构的性能，确定可行的处理器结构。指令集生成阶段利用应用分析结果，添加专用指令提高处理器的性能和效率。代码综合阶段，为处理器生成编译器，同时生成应用的可执行代码。硬件综合阶段生成设计的可综合 RTL 代码。

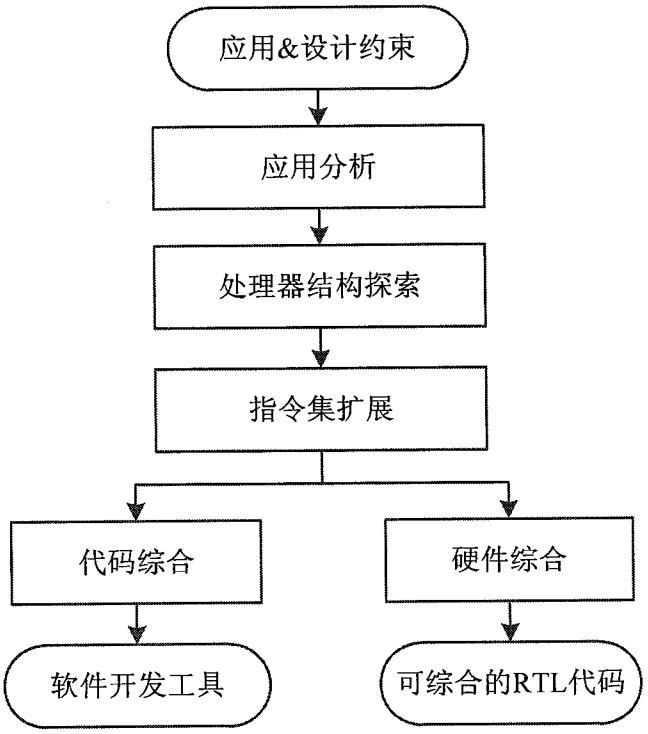


图 1-6 通常的 ASIP 设计流程

1.2.1 应用分析

应用分析是 ASIP 设计的第一步，以单个或者一组应用（通常以高级语言实现）作为输入，通过分析源代码结构，评估程序的执行时间以及存储器消耗，识别程序的热点，即消耗最多时间和功耗的程序代码。分析结果用来消除应用程序热点，增加处理器的性能，减少程序的运行时间和功耗。

应用分析中广泛使用代码分析（Code profiling）[15][16]，为了保证分析结果的有效性和 ASIP 的灵活性，源代码的选择需要典型，且要达到一定量，过多或过

少的代码都不利于代码的分析。代码分析揭示了程序的执行过程，如函数、循环体和子程序，同时给出程序的执行时间和硬件资源的利用率。执行时间通常以操作数衡量，需要根据选取的处理器结构换算为时钟周期。

代码分析可以在源代码级或者汇编级进行。源代码级的分析速度快，但是精度低，一行 C 代码可能包含多个操作，被映射为多条指令。一些汇编级的操作对源代码级的分析是不可见的，如地址计算，存储器访问以及类型转换等。同时 C 编译器会采用不同的编译器策略对源代码进行优化，源代码级的分析通常不会考虑这些因素。因此源代码级的分析对于指令集的设计是不够精确的。

基于处理器结构模型的汇编级代码分析对于指令集的设计足够精确，但是要求设计者已有适合应用的可用处理器设计，以其指令集作为设计的参考指令集，通过分析其与期待指令集的差别，获得新的指令集。

开源编译器架构的中间表示（Intermediate Representation，IR）是一种低级语言，与机器汇编语言类似。编译器前端可以将高级语言源程序转换为优化的可执行 IR，其中程序的所有的操作（如算术操作、逻辑操作、类型转换、地址计算、存储器访问）都是可见的。因此基于 IR 的分析结果足够的精确有效，同时不需要可用的处理器结构。

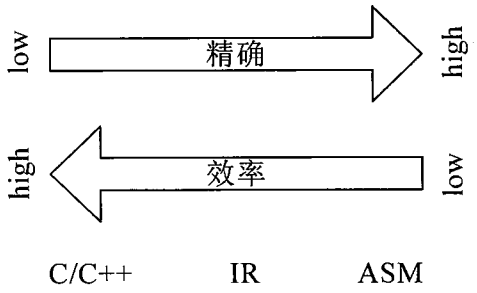


图 1-7 Comparing of Three Kinds of Code Profiling

应用分析识别热点代码后，潜在的优化和加速就可以被确定。代码识别装置（Code segment identification）[17]从程序的热点中寻找小的可重用的有向无环图（Directed Acyclic Graphs，DAG）[18]，并将其以专用指令添加到指令集。当处理器仍无法提供足够的计算性能时，通过代码识别装置从程序的热点中识别规模较大、规整、稳定的子图（sub-graphs），将其映射为专门的硬件加速单元，添加到专用指令集处理器，如 FFT 单元。

1.2.2 专用指令扩展

ASIP 设计中，一个主要的优化方法既是添加专用指令[19][20][21][22]。指令扩展依赖于识别装置[23][24][25]，典型的识别装置识别算法热点程序的数据流图（Data Flow Graphs，DFG），将多个操作合并为一条指令。识别装置的结果是一系列的可扩展指令 ISE，以 DAG 的形式表示，最终映射为硬件模型中的数据通路。

专用指令的添加可以明显的减小代码大小与执行时间，使得应用程序代码更加高效。运行时间的减少意味这功耗的减小。专用指令添加到处理器的数据通路，在处理器的指令执行级设计，执行的操作应该避免违反时钟周期，引起关键路径。如果违反不可避免，将指令划分为多条指令，或者多周期执行指令。如果多周期执行指令，需要对流水线进行合适的调度，以免引起流水线冲突。

对于需要大量访问存储器的应用，指令包含存储器访问可以减少 load 和 store 指令，从而减少因取值和译码以及寄存器访问带来的功耗。

指令集扩展能够最大化性能，同时满足应用对芯片尺寸以及功耗的需求。

1.2.3 硬件加速

如果指令集扩展不能满足计算性能或者功耗需求，可以添加硬件加速单元。硬件加速单元是专门的硬件模块，完成有限的计算任务，具有强大的计算性能，可以减少运行时间和功耗，代价是面积的增加与灵活性的下降。如果算法改变，加速单元可能会不再适用。因此硬件加速单元仅在处理器不能提供足够的计算性能或者指令集不能提供的功能，如计算密集的 FFT 单元、整数除法等特殊运算单元。

处理器应该提供一个良好的接口（interface）和控制装置（control mechanism）。接口在 ASIP 核和加速单元间传递数据，可以通过通用或者专用的寄存器实现。通常硬件加速单元比指令和专用指令需要更多的执行周期，控制装置通过可配置的寄存器开启加速单元，完成加速单元与处理器之间的程序流同步。当添加或者删除加速单元时，处理器的设计应该不做修改，以减少重设计带来的风险。

加速单元可以通过单条指令或者多条指令实现，指令只用来完成加速单元的

配置信息，如控制信号和操作数。

单指令加速单元用于实现功能较简单、以固定时钟周期完成的运算，如整数除法、平方根、对数、以及倒数运算等。

多指令加速单元用于实现相对规整的功能。基于其简单的指令流，通过指令的不同组合可以实现更多的功能[26]，因此多指令加速单元可以提供更大的灵活性。如多指令的 FFT 加速单元，每一层的 FFT 以一条指令执行，处理器负责配置 FFT 的指令流。

加速单元也可以是一个从处理器，主处理器仅发送任务代码至从处理器，初始化任务，从处理器在主处理器的控制下完成运算。由于从处理器是可编程的，因此能够执行多种复杂的任务，代价则是增加的芯片面积。由于主从处理器可以并行处理，所有系统具有良好的吞吐率，通过合理的应用划分可使得系统工作非常高效[27]。

1.2.4 代码综合

软件开发工具如编译器、汇编器、链接器和仿真器都是 ASIP 设计过程中必须的。ASIP 设计的代码综合阶段，即是生成软件开发工具和应用的可执行代码。ASIP 设计的可行性，要求软件开发工具能够在短时间实现。

优化的硬件实现是 ASIP 设计的关键，软件开发工具能够用来评估设计[28]以及提供设计反馈，因此设计者可以借助开发工具验证设计是否满足设计约束，更好的设计处理器结构。

软件实现如果能够有效的利用硬件资源可以有效的减少功耗。针对 ASIP 精心设计的编译器能够有效的将程序映射到专用指令和加速单元。同时，编译器使用多种编译优化策略，可以产生高效的目标代码，有效减少运行时间和功耗。功耗驱动的代码生成器基于指令集功耗模型选择指令，指令排序也可以减少连续指令的翻转从而减少功耗[29].

对算法进行优化可以减少功耗消耗[30]。通常需要更多运行时间和操作的算法消耗更多的功耗，复杂的算法需要更多的操作，不规则的算法需要更多的程序

控制流指令，程序控制流指令会因程序跳转而导致跳转惩罚。存储器访问通常消耗更多的功耗，因此算法应该尽量减少存储器访问和存储器容量，以提高能量效率。算法实现遵循编译器用户手册，达到高效使用硬件资源的目的，内联汇编与内函数能够帮助编译器将算法直接映射到硬件。

ASIP 设计是一个软硬件协同设计的过程，只有软硬件联合优化，才可以获得最优的低功耗 ASIP 设计。

1.2.5 硬件综合

当仿真结果满足设计约束时，专用指令和加速单元的描述将用来生成可综合的 RTL 设计。在硬件综合阶段，指令映射到硬件实现，安排到不同的流水线级。

指令映射到硬件模块时，复用技术可以通过复用硬件减小硬件资源消耗，但是可能会引起关键路径。

并行处理能够减小功耗消耗，对于流水线处理器，需要根据时钟周期约束指定合适的流水线深度，更高的工作频率要求更深的流水线，但会导致更多的流水线冲突，从而减少流水线的利用率。

存储器子系统通常消耗明显的功耗，存储器功耗随着存储器容量的增加而增加。因此分割存储器或者采用小容量的缓存能够减少存储器功耗[31]，面向应用的存储器系统也可以有效的减少存储器功耗。

1.2.6 本节小结

通过提取应用特征，添加专用指令或者硬件加速单元，完成自定制结构的硬件设计，协同优化的软件开发工具，将应用高效的映射到硬件资源，从而获得低功耗的设计。因此 ASIP 设计是一个迭代的软硬件协同设计过程，也是一个面积、性能与功耗折中的设计过程。

1.3 国内外研究现状

自十九世纪 70 年代 ASIP 概念提出，经过若干年的发展，专用指令集处理器的设计已由手工发展到自动化，在国外的研究中，更是涌现了大量的 ASIP 设

计工具以及解决方案，如表 1-1，ASIP 设计可以基于结构描述语言（Architecture Description Language，ADL）[32][33][34][35][36][38]或者已有的可配置处理器（configurable processor）[37][39]，其中不乏成功的商业案例[38][39]。

基于 ADL 的设计，使用抽象的高层 ADL 语言描述处理器行为，隐藏了具体的实现，通过设计环境辅助，可以自动生成软件开发工具集（如编译器、汇编器、链接器、以及仿真器）和可综合的硬件 RTL 代码。基于 ADL 的设计给设计者提供了足够的自由度，对应用的 ASIP 设计更加高效，但需要设计者更多的设计时间和工作。

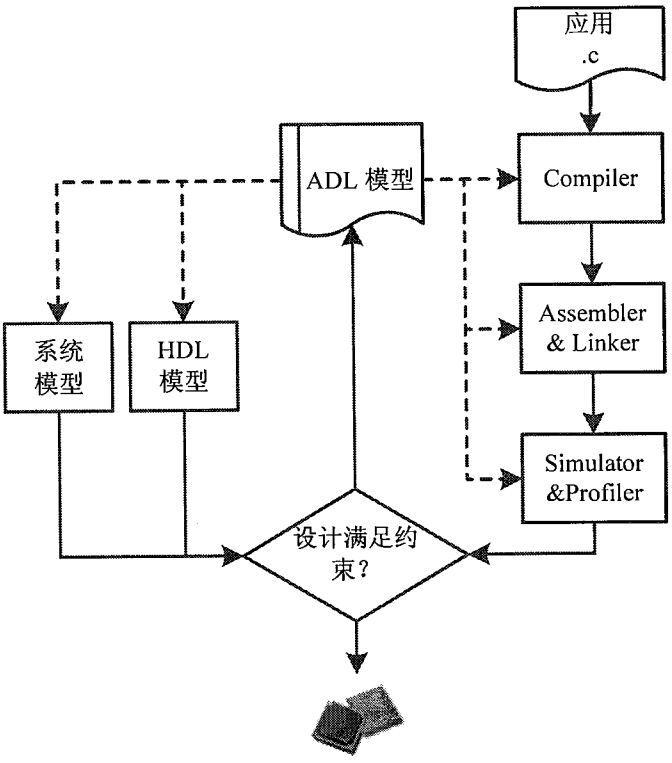


图 1-8 基于 ADL 的 ASIP 设计

基于可配置处理器的 ASIP 设计对设计者的要求较低，但是设计受到已有处理器的约束。首先需要在多个处理器模板中选择适合应用的的可配置处理器，进而通过仿真在其数据通路中添加指令扩展，并更新相应的软件开发工具。因此设计的 ASIP 往往并不是非常高效的，但是可以在短时间内获得满足系统需求的设计。

表 1-1 国外的相关研究

工具或解决方案	1	2	3	4	5	6	7	8
MIMOLA				Y			Y	N
Cathedral-II	Y			Y	Y		Y	N
Target	Y	Y	Y	Y	Y		Y	N
ARC		Y	Y	Y	Y		Y	Y
Tensilica	Y	Y	Y	Y	Y	Y	Y	Y
LISA	Y	Y	Y	Y	Y		Y	Y
MESCAL	Y		Y	Y	Y		Y	N
PEAS-III			Y	Y	Y		Y	Y
NoGAP	Y		Y	Y	Y		Y	N
…								

1 应用程序代码分析与架构探索
2 生成编译器
3 生成汇编器
4 生成仿真器
5 生成周期精确的行为级处理器模型
6 指令集与结构优化
7 生成可综合的 RTL 代码
8 有限的处理器结构

2000 年左右开始，国内陆续出现了关于 ASIP 的相关研究，如表 1-2 所示。

表 1-2 国内的相关研究

国内	设计描述/研究重点
国防科学技术大学	基于 TTA 体系结构面向 JPEG 解码的多核 ASIP 设计
清华大学	基于开源处理器面向人工耳蜗的 ASIP 设计
中国科学技术大学	xpADL 体系结构语言，A^2IDE 的设计环境
…	…

对比国内外的研究发现，国外对 ASIP 设计的研究已经比较完善，涵盖了 ASIP 设计的各个方面，相对成熟；但是国内却多是处于起步阶段，在已有或者开源的处理器结构上进行的探索性工作。

虽然 ASIP 的设计环境[38][39][40][41]越来越高效，但是 ASIP 设计方法仍面临许多问题，如指令集定制、体系结构优化、软件开发工具的可重定向生成等。因此，完善 ASIP 设计方法，以高性能、低功耗为设计目标，设计可扩展的 ASIP 处理器体系结构，研究专用指令与加速单元的设计，以及对其性能、功耗进行评估，具有重要的研究意义和广阔的应用前景。

1.4 论文的内容与组织结构

1.4.1 论文的内容

数字助听器具有庞大的市场，与人类的生活质量密切相关，因此本文以数字助听器系统作为音频处理的典型应用，以 FlexEngine 为基本指令集处理器核，主要研究了低功耗的 ASIP 设计。设计了 ASIP 的软件开发工具，用于设计评估与提供设计反馈；基于算法的汇编级代码分析，识别可能的扩展指令，并映射到 ASIP 的数据通路；基于标准单元，在 ASIP 设计的软件级、系统结构级、寄存器传输级、以及逻辑级，综合运用多种低功耗设计技术，完成了数字助听器的低功耗 ASIP 设计。本论文的主要贡献有：

（1）建立精确到指令周期级的音频程序分析方法，利用分析结果制定处理器扩展专用指令和加速硬件策略，快速、准确的完成处理器指令集设计迭代。通过对音频算法进行汇编级代码分析，识别可能的扩展指令，将频繁使用的指令组合与指令序列以专用指令或者协处理器的形式添加到 ASIP 的数据通路。针对数字助听器算法，添加了位反寻址、零开销循环、对数、开方、倒数、蝶形运算、睡眠模式、数据并行加载等专用指令。自定制的处理器结构与应用程序高度匹配，大大提升处理器的运算效率。

（2）完成基于 LLVM 与 Clang 平台的编译器设计，具有模块化、结构清晰、可移植性强、易于生成代码优化等优点，为处理器开发提供工具链支持。处理器相关的编译器代码优化能够有效的将应用程序映射到专用指令和加速单元。ASIP 的软件开发工具，可用于 ASIP 设计时的应用代码分析与设计评估，同时极大的方便了基于 ASIP 的应用开发。对于面向不同应用的 ASIP 设计，需要指令扩展时，可以快速完成软件开发工具的修改，生成 ASIP 的软件工具。

（3）在多个设计层级对处理器功耗进行优化，实现了面向音频应用的低功耗专用指令集处理器，功耗小于 1mW。本文以数字助听器系统作为音频处理的典型应用，综合运用算法优化、低功耗的工作模式、指令集优化、循环缓存、存储器分割、门控时钟、操作数隔离、等多种低功耗设计技术，大大的降低了系统功耗。最终，基于 TSMC 130nm 工艺完成了 ASIP 设计流片，经过测试，该系统能够很好的完成数字助听器功能，并且低功耗效果显著，工作在 8MHz 频率、1.2V 工作电压时，处理器功耗约 0.963mW。

1.4.2 论文的组织结构

论文的组织结构如下：

第一章介绍了本文的研究背景与研究意义、研究方法、以及国内外的研究现状，确定了本文的研究方向和主要工作。

第二章介绍了本文低功耗音频专用指令集处理器研究的目标应用数字助听器系统，确定了本文 ASIP 设计的基本指令集处理器核 FlexEngine，并对 FlexEngine 的体系结构设计进行了详细的说明。

第三章介绍了 FlexEngine 处理器的软件开发工具设计，包括编译器、二进制工具集、以及仿真器。

第四章介绍了低功耗音频专用指令集处理器设计的具体过程。以数字助听器系统为应用，展开低功耗的 ASIP 设计，详细介绍了汇编级代码分析、专用指令的扩展、硬件加速单元的添加、软件开发工具的专用指令扩展、以及在 ASIP 设计的各阶段所采用的低功耗技术。

第五章介绍了低功耗数字助听器 ASIP 的物理实现与测试结果。

第六章为本文的工作总结与展望。

参考文献

[1]Dijkmans E C. Hearing instruments go digital[C]//Solid-State Circuits Conference, 1997. ESSCIRC'97. Proceedings of the 23rd European. IEEE, 1997: 16-28.

[2]肖宪波, 谷子, 胡广书, 等. 数字助听器算法开发平台 pDHA 的构建和测试[J]. 北京生物医学工程, 2006, 25(1): 55-58.

[3]Kates J M. Digital hearing aids[M]. Plural Pub., 2008.

[4]Fabry D. "Acoustic Scene Analysis" and Digital Hearing Aids[J].

[5]Benesty J, Huang Y. Adaptive signal processing: applications to real-world problems[M]. Springer Verlag, 2003.

[6]Jain M K, Balakrishnan M, Kumar A. ASIP design methodologies: Survey and issues[C]//VLSI Design, 2001. Fourteenth International Conference on. IEEE, 2001: 76-81.

[7]Orailoglu A, Veidenbaum A. Guest editors' introduction: application-specific microprocessors[J]. IEEE Design & Test, 2003: 6-7.

[8]Gries M, Keutzer K. Building ASIPs: The Mescal Methodology[M]. Springer, 2005.

[9]Fisher J A. Customized instruction-sets for embedded processors[C]//Proceedings of the 36th annual ACM/IEEE Design Automation Conference. ACM, 1999: 253-257.

[10]Glökler T, Meyr H. Design of energy-efficient application-specific instruction set processors[M]. Springer, 2004. pp. 46-47.

[11]Liu D. Embedded DSP processor design: Application specific instruction set processors[M]. Morgan Kaufmann, 2008.

[12]De Micheli G. Computer-aided hardware-software codesign[J]. Micro, IEEE, 1994, 14(4): 10-16.

[13]Gschwind M K. Hardware Software co-evaluation of instruction sets[M]. 1996.

[14]Jain M K, Balakrishnan M, Kumar A. ASIP design methodologies: Survey and issues[C]//VLSI Design, 2001. Fourteenth International Conference on. IEEE, 2001: 76-81.

[15]Liu D. Embedded DSP processor design: Application specific instruction set processors[M]. Morgan Kaufmann, 2008. pp. 217-220

[16]Karuri K, Al Faruque M A, Kraemer S, et al. Fine-grained application source code profiling for ASIP design[C]//Design Automation Conference, 2005. Proceedings.

42nd. IEEE, 2005: 329-334.

[17]Henkel J, Parameswaran S. Designing embedded processors: a low power perspective[M]. Springer, 2007. pp. 12-14

[18]Karuri K, Leupers R, Ascheid G, et al. A Generic Design Flow for Application Specific Processor Customization through Instruction-Set Extensions (ISEs)[J]. Embedded Computer Systems: Architectures, Modeling, and Simulation, 2009: 204-214.

[19]Atasu K, Pozzi L, Ienne P. Automatic application-specific instruction-set extensions under microarchitectural constraints[J]. International Journal of Parallel Programming, 2003, 31(6): 411-428.

[20]Biswas P, Banerjee S, Dutt N, et al. ISEGEN: Generation of high-quality instruction set extensions by iterative improvement[C]//Design, Automation and Test in Europe, 2005. Proceedings. IEEE, 2005: 1246-1251.

[21]Yu P, Mitra T. Scalable custom instructions identification for instruction-set extensible processors[C]//Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems. ACM, 2004: 69-78.

[22]Clark N, Zhong H, Mahlke S. Processor acceleration through automated instruction set customization[C]//Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2003: 129.

[23]Karuri K, Leupers R, Ascheid G, et al. A Generic Design Flow for Application Specific Processor Customization through Instruction-Set Extensions (ISEs)[J]. Embedded Computer Systems: Architectures, Modeling, and Simulation, 2009: 204-214.

[24]]Verma A K, Brisk P, Ienne P. Rethinking custom ISE identification: A new processor-agnostic method[C]//Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems. ACM, 2007: 125-134.

[25]Glokler T, Bitterlich S. Power efficient semi-automatic instruction encoding for application specific instruction set processors[C]//Acoustics, Speech, and Signal Processing, 2001. Proceedings.(ICASSP'01). 2001 IEEE International Conference on. IEEE, 2001, 2: 1169-1172.

[26]Liu D. Embedded DSP processor design: Application specific instruction set processors[M]. Morgan Kaufmann, 2008. pp. 601-602

[27]Liu D. Embedded DSP processor design: Application specific instruction set

processors[M]. Morgan Kaufmann, 2008. pp. 602-603

[28]Gong J, Gajski D D, Nicolau A. Performance evaluation for application-specific architectures[J]. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 1995, 3(4): 483-490. pp. 483-490.

[29]Tiwari V, Malik S, Wolfe A, et al. Instruction level power analysis and optimization of software[J]. The Journal of VLSI Signal Processing, 1996, 13(2): 223-238.

[30]Landman P E. Low-power architectural design methodologies[D]. University of California, 1994. pp. 63-68.

[31]Henkel J, Parameswaran S. Designing embedded processors: a low power perspective[M]. Springer, 2007. pp. 131-141

[32]Halambi A, Grun P, Ganesh V, et al. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability[C]//Proceedings of the conference on Design, automation and test in Europe. ACM, 1999: 100.

[33]Fauth A, Van Praet J, Freericks M. Describing instruction set processors using nML[C]//European Design and Test Conference, 1995. ED&TC 1995, Proceedings. IEEE, 1995: 503-507.

[34]Halambi A, Grun P, Ganesh V, et al. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability[C]//Proceedings of the conference on Design, automation and test in Europe. ACM, 1999: 100.

[35]Hoffmann A, Meyr H, Leupers R. Architecture exploration for embedded processors with LISA[M]. Springer, 2002.

[36]Hoffmann A, Kogel T, Nohl A, et al. A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language[J]. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 2001, 20(11): 1338-1354.

[37] ARC International, http://www.arc.com

[38] CoWare LISATek Tools, http://www.coware.com/

[39] Tensilica, http://www.tensilica.com/

[40]Glökler T, Meyr H. Design of energy-efficient application-specific instruction set processors[M]. Springer, 2004. pp. 6-10.

[41]Hohenauer M, Scharwaechter H, Karuri K, et al. A methodology and tool suite for c compiler generation from adl processor models[C]//Proceedings of the conference

on Design, automation and test in Europe-Volume 2. IEEE Computer Society, 2004: 21276.

第2章 音频专用指令集处理器

作为 ASIP 设计的第一步，首先确定应用程序以及设计约束，探索基本指令集处理器，作为 ASIP 设计的起点。本章对设计面向的应用数字助听器系统、以及所采用的基本指令集处理器核 FlexEngine 进行了介绍。

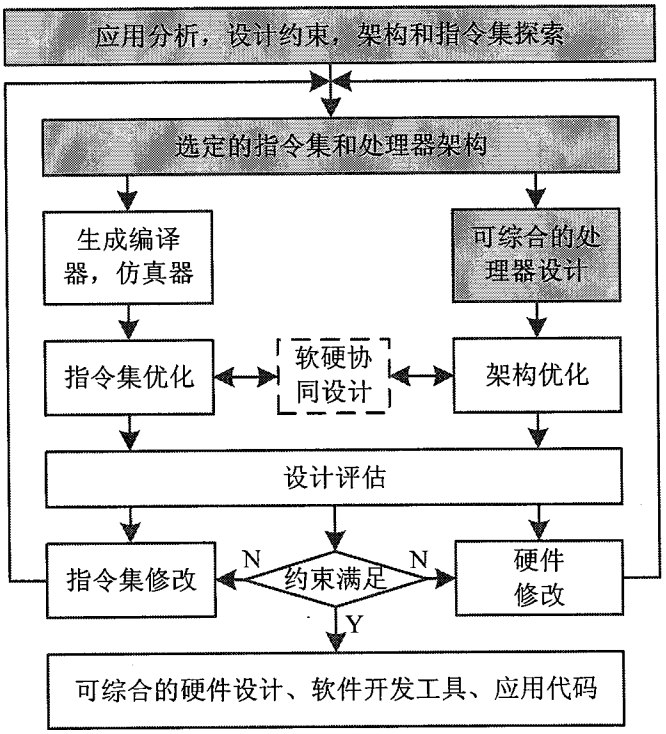


图 2-1 软硬协同的 ASIP 设计流程

本文的 ASIP 设计面向音频处理系统，音频处理算法是数据密集型运算，通常包含加窗、滤波运算、时域频域变换、语音增强、噪声消除、反馈抑制、频率移动、语音识别等，这些算法有以下特点：（1）包含大量特殊运算，如开方、对数、倒数等；（2）执行过程以小循环体为主，占大部分执行时间，对程序控制要求较高；（3）精度要求数据位宽以 24bit 为宜，也是当前语音处理器的主流位宽；（4）频繁使用累加操作，要求能够提供高精度的中间计算过程；（5）寻址模式灵活多样。

针对音频处理算法的以上特点，本文选取了其典型应用数字助听器做为 ASIP 设计的目标应用，对低功耗音频 ASIP 设计开展了相关研究。首先对数字助听器系统做简单的介绍。

2.1 数字助听器系统

数字助听器系统对经过 A/D 采样的语音信号进行数字信号处理，处理后的语音信号通过 D/A 转换为模拟信号驱动扬声器。数字信号处理的核心功能是对输入语音信号进行听力补偿，为了提高语音的可辨识度与聆听舒适感，数字助听器系统通常会添加更多更复杂的数字信号处理模块。图 2-2 显示了一个高端数字助听器系统所包含的主要信号处理单元[1]，语音信号首先被三路方向性麦克风采集，经过波束整形变为一路语音信号，进而分析滤波器组将其分解为多个频率带，不同的频率带经过噪声消除与动态压缩，最终综合为一路语音信号。

由于助听器体积小，麦克风与扬声器距离较近，扬声器发出的一部分语音信号回馈到麦克风，通过助听器再次采集放大，如此循环反复，便会产生啸叫。反馈抑制模块对输入信号中的反馈分量进行抑制，避免产生啸叫。

在不同的环境下，助听器的各个模块通常需要采取不同的算法才能达到最佳的聆听效果。场景识别模块对环境进行分类，进而选择与之相适应的算法。在双耳助听器系统中，两耳间的无线数据交互可以显著的提升场景识别的效果。

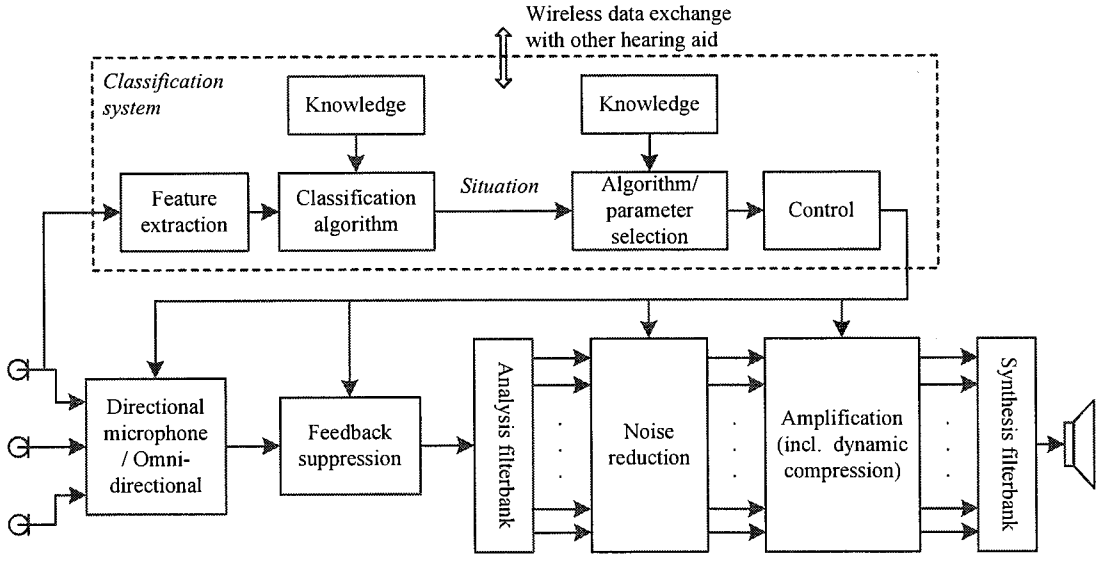


图 2-2 高端数字助听器系统

作为本文 ASIP 设计的目标应用，算法实现了数字助听器的核心功能模块多通道分离[2][3]、听力补偿[2]和噪声消除[2][3]，系统框图如图 2-3。功能模块划分为分块操作、多通道分离、听力补偿和噪声消除。

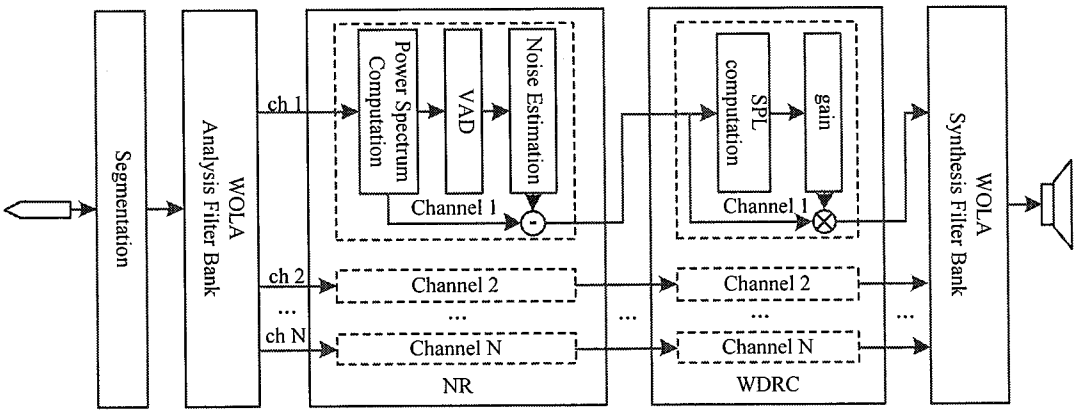


图 2-3 数字助听器系统框图

1）分块操作

数字助听器主要使用两种方法进行信号处理和频率分析，第一种利用滤波器组将输入语音信号划分为若干不等频带的通道，对每一个采样点单独进行操作。第二种利用快速傅立叶（Fast Fourier Transformation , FFT）原理将输入语音信号划分为若干等频带的通道，对采样点进行分块操作[4]，也是本文采用的方法。

2）多通道分离

听力损失具有很强的频率相关性，且不同频带的语音信号具有不同的特性，因此在数字助听器系统中，多采用多通道信号处理。WOLA（weighted overlap-add）滤波器组[5]是 DFT 滤波器组的一种高效实现，WOLA 分析滤波器组（Analysis Filter Bank, AFB）将输入信号分解为多个通道，是后续多通道听力补偿与噪声消除的基础。WOLA 分析滤波器组的表达式如(2.1)：

$$X_k(m) = W_K^{-kmM} \sum_{r=0}^{K-1} \sum_{l=-\infty}^{\infty} h(-r-lK)x(r+lK+mM)W_K^{-kr}, k = 0,1,...,K-1 \qquad (2.1)$$

其中，K 为 FFT 点数，通道数 N=K/2，反映了助听器对声音信号按频段处理的细致程度．M 为降采样因子，h (n)为分析窗。WOLA 分析滤波器组的实现框图如图 2-4 所示。

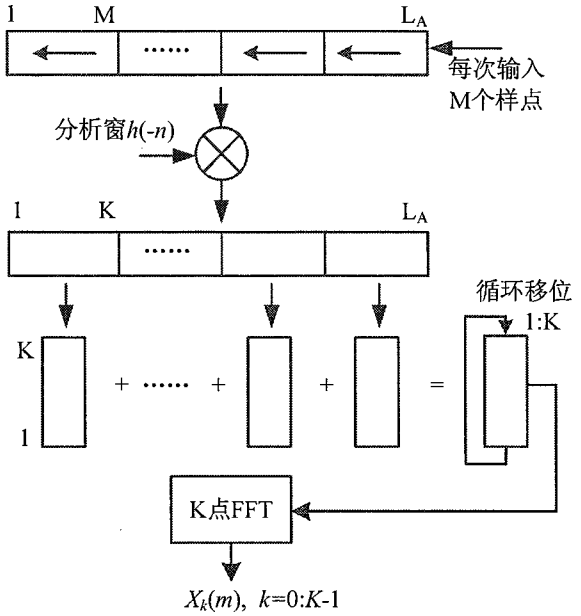


图 2-4 WOLA 分析滤波器组

WOLA 综合滤波器组（Synthesis Filter Bank, SFB）是分析滤波器组的逆过程，将经过处理的多通道信号综合为一路信号。WOLA 综合滤波器组的表达式如(2.2)：

$$y(n+mM)|_{m=m_0} = f(n)\frac{1}{K}\sum_{k=0}^{K-1} Y_k(m)W_K^{kmM}W_K^{kn}|_{m=m_0} + (m \neq m_0\text{项}) \tag{2.2}$$

其中，f(n)是为综合窗，实现框图如图 2-5 所示。

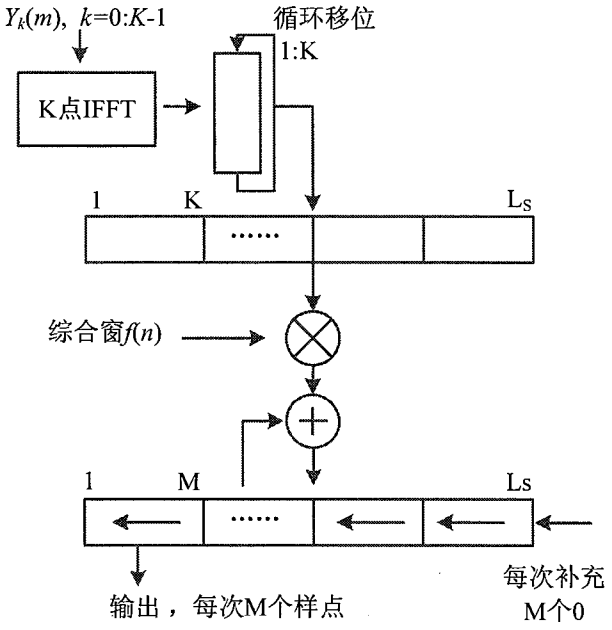


图 2-5 WOLA 综合滤波器组

3）听力补偿

多通道宽动态范围压缩（Wide Dynamic Range Compression，WDRC）算法根据听力障碍人士的听力损失曲线，对输入语音信号在各个通道进行听力补偿，从而将正常人听觉动态范围的声音映射到听障患者的听觉动态范围[2][3]。听力补偿是数字助听器系统中最基本，也是最重要的功能。

听力损失人群的听阈较正常人窄，并且在不同频段的听力损失程度也不相同，多通道宽动态范围压缩算法将输入语音在不同频段进行压缩或者放大，将正常人听力范围内的声音映射到听力损失患者的听觉动态范围，从而提高患者的语音辨识度。如图 2-6 所示，THR（threshold）代表听阈，UCL（uncomfortable level）代表痛阈，听阈与痛阈之间为听觉范围。

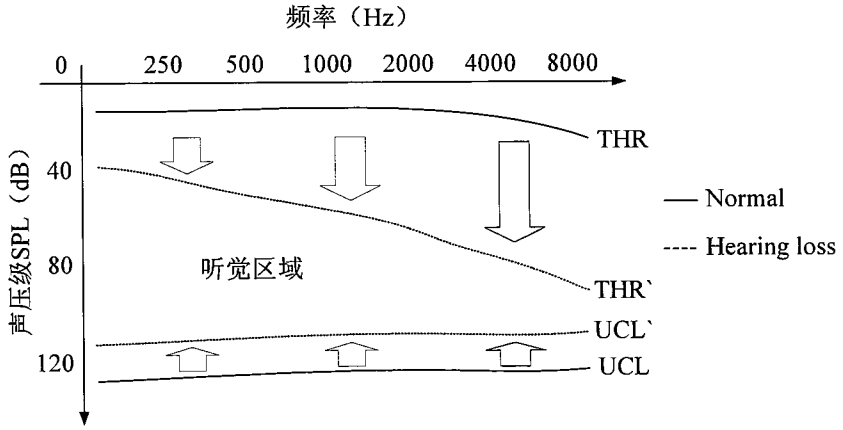


图 2-6 听觉区域与 WDRC 原理

WDRC 算法（框图如图 2-7）通过峰值检测计算输入信号的声压级（SPL），根据声压级和听力补偿曲线计算增益，通过增益对输入信号进行放大或者压缩[2][6][7]。

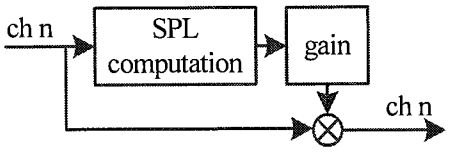


图 2-7 WDRC 实现框图

输入语音信号被分析滤波器组划分为若干频率等宽的通道，对每一个通道进行 WDRC 宽动态压缩，实现多通道的听力补偿。

4）噪声消除

噪声消除（Noise Reduction，NR）用来降低输入语音中的背景噪声，提高带噪语音的可辨识度及听觉舒适度。设计采用多子带谱相减算法，首先将各通道根据人耳听觉特性进行合并、划分为若干子带，通过估计获得每个子带的噪声谱，然后在子带内进行谱相减噪声消除[3][8]，如图 2-8。

谱相减法基于噪声和语音不相关的基本假设，同时利用了人耳对相位信息的不敏感性。它将噪声谱直接从带噪语音谱中减去，并利用带噪语音的相位重建消噪后的纯净语音。

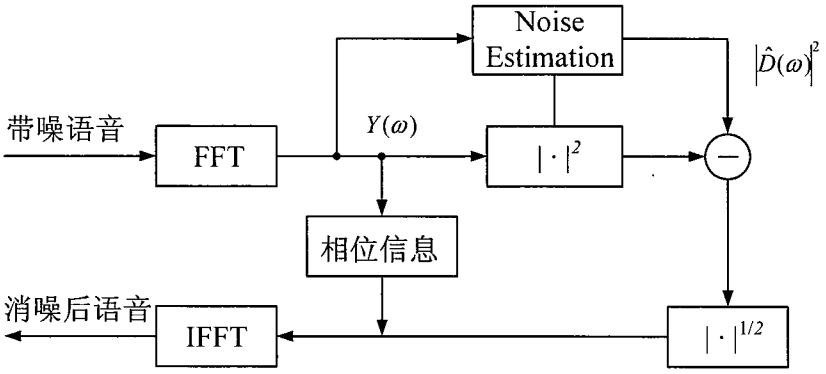


图 2-8 谱相减噪声消除

2.2 FlexEngine

通过对音频处理算法的分析，针对数字助听器应用选择合适的基本指令集处理器核做为 ASIP 设计的起点。FlexEngine 是项目组自主设计的处理器核，适用于音频设计，满足设计的要求，以下对 FlexEngine 的结构设计进行介绍。

FlexEngine 为 24 位低功耗数字音频信号处理器，采用改进的哈佛结构，一条指令总线和两条数据总线的三总线结构，RISC 指令集，多级流水线。FlexEngine 的系统框图如图 2-9，主要包含算术逻辑运算单元（arithmetic and logic unit，ALU），乘累加单元（multiplication and accumulation unit，MAC），寄存器文件（register file，RF），程序控制单元（program control unit，PCU），地址产生单元（address generation unit，AGU），以及外围设备等。

FlexEngine 有 32 个 24-bit 通用寄存器(General register) r0 - 31，做为数据的临时存储与操作数的来源用于 ALU、MAC 和存储指令；FlexEngine 指令集根据功

能可以分为数据传输类、算术运算类、逻辑运算类、流程控制类、双精度运算类指令。

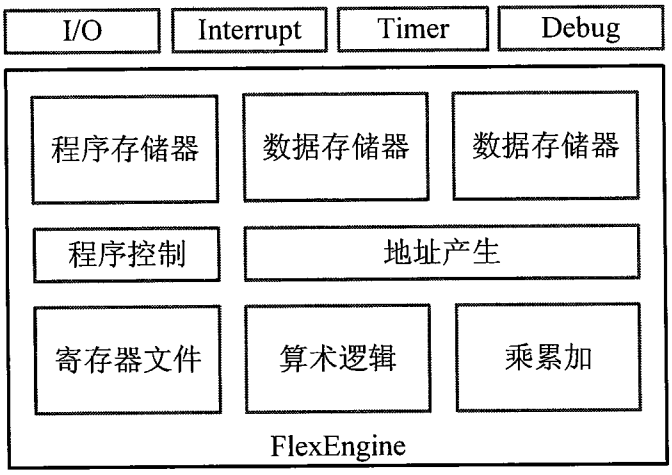


图 2-9 FlexEngine 系统框图

2.2.1 流水线结构

FlexEngine 采用多级流水线，包括取指(FE)、译码(DE)、取操作数(OF)、执行(IE)、回写(WB)5 个执行过程。ALU 指令是单周期执行指令，MAC 指令则需要两个时钟的执行周期，而以存储器数据作为操作数的指令需要额外的一个时钟周期完成存储器取操作数。图 2-10 是 FlexEngine 的流水线结构示意图。

FE 为取指级，根据程序控制单元给出的程序地址从程序存储器中取出将要执行的指令；DE 为译码级，对指令进行译码，得到指令包含的操作码、源操作数、目的操作数、执行条件及控制信号等；OF 为取操作数级，根据 DE 得到的控制信号，读取相应位置的源操作数，准备进行运算；IE 为执行级，表示根据 DE 阶段的操作码和控制信号，对 OF 阶段准备的数据进行运算，得到执行结果；WB 为回写级，表示将 IE 的执行结果，存回 DE 阶段指定的目的操作数地址。

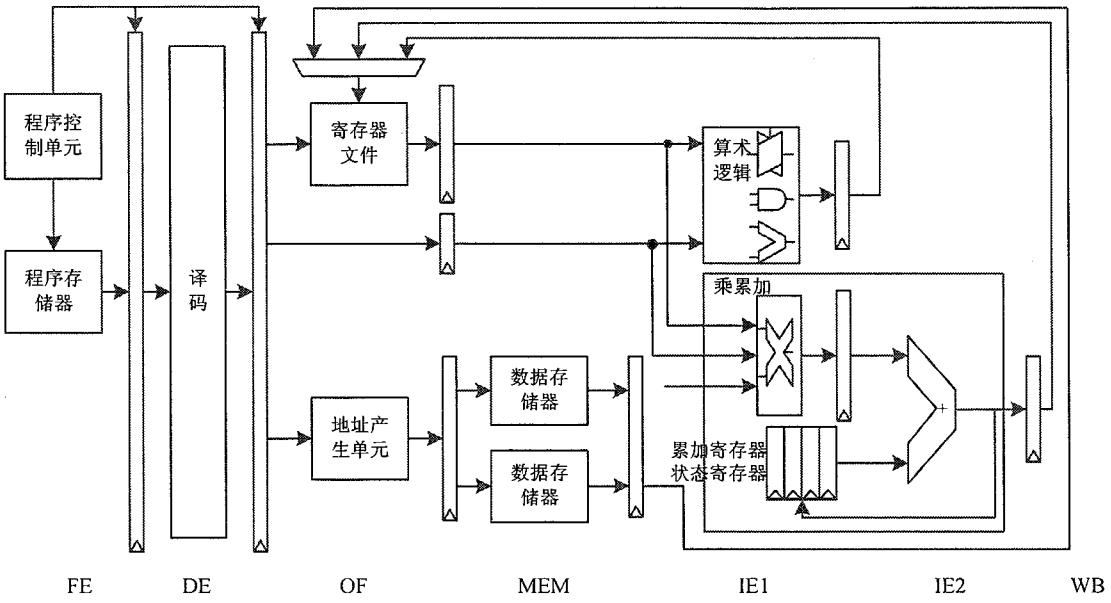


图 2-10 FlexEngine 的流水线结构图

对于流水线结构，会不可避免的产生流水线冲突[9][10]，通过数据前推（data forwarding/bypass）[11]将结果在 WB 级回写前推送至执行单元，可以减少由于流水线冲突引起的数据竞争造成的错误。

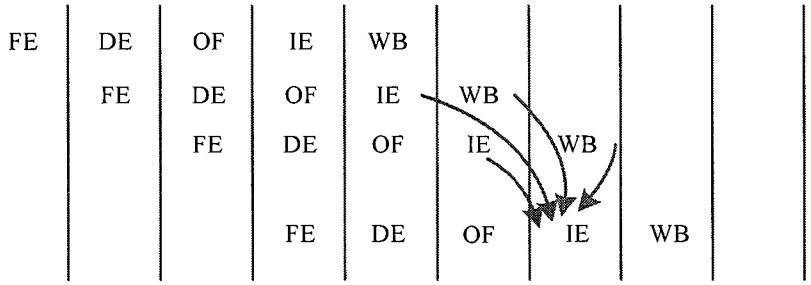


图 2-11 数据前推

2.2.2 ALU 单元

ALU 单元是处理器的关键运算单元，可以完成算术、逻辑、移位、位操作以及特殊运算等诸多运算，操作数来源于寄存器文件或者指令中的立即数，结果写回至寄存器文件。操作数在运算前可能需要进行预处理，如插入保护位、位反、进位位等；运算后进行后处理，如饱和处理、标志位更新（零标志位、负标志位、进位标志位、溢出标志位）。ALU 单元可以在单个时钟周期内完成加法、移位、与、或、非等运算中的任何一种运算。结构如图 2-13。

FlexEngine 的 ALU 运算类指令支持条件执行，其运算结果是否写回，会根

据状态寄存器的值进行判断，如果条件满足则将结果写回，否则丢弃而不写回结果。通过在程序中使用条件执行指令，可以减少条件转移基本块，从而节省条件转移开销。

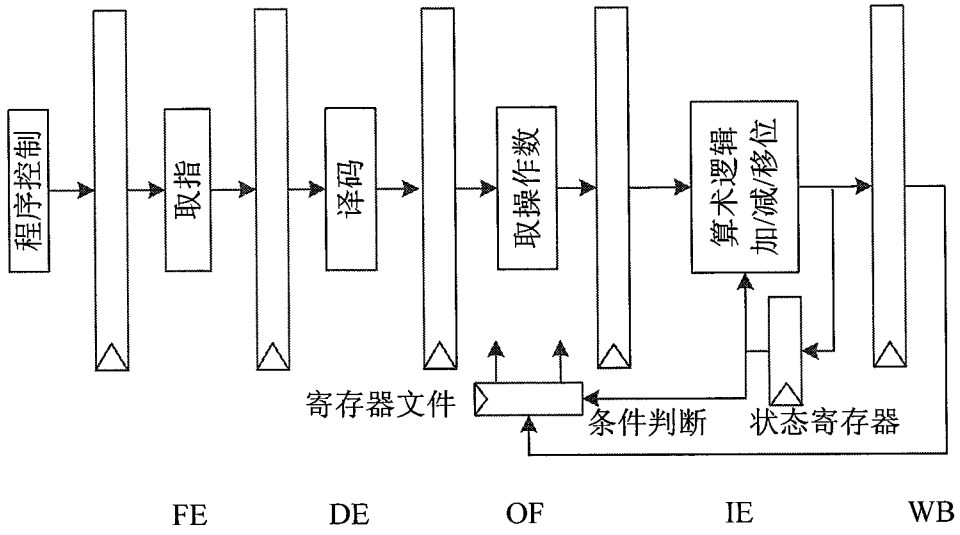


图 2-12 ALU 单元指令流水线

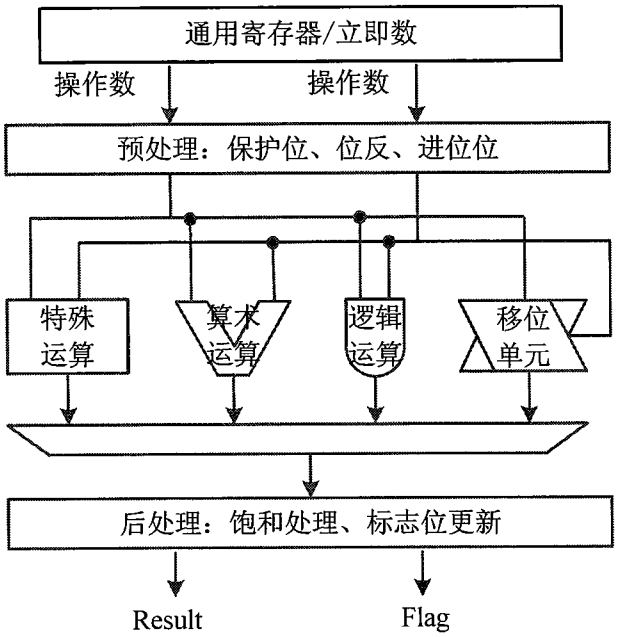


图 2-13 ALU 单元

通过在程序中使用条件执行指令，减少条件转移基本块，节省条件转移开销。如数字助听器算法中过减因子 α_i 的计算过程，如式(2.3)：

$$\alpha_i = \begin{cases} 4.75 & SNR_i < -5 \\ 4 - \frac{3}{20} \cdot SNR_i & -5 \le SNR_i \le 20 \\ 1 & SNR_i > 20 \end{cases} \tag{2.3}$$

伪指令表示：

$\alpha_i = 4 - \frac{3}{20} \cdot SNR_i$;设置初值 $\alpha_i = 4 - \frac{3}{20} \cdot SNR_i$

cmp SNR_i, *-5*

add *{Condition LT}* α_i, *r0, 4.75* ;如果 $SNR_i < -5$，则 $\alpha_i = 4.75$

cmp SNR_i, *20*

add *{Condition GT}* α_i, *r0, 1* ;如果 $SNR_i > 20$，则 $\alpha_i = 1$

2.2.3 MAC 单元

MAC 单元是数字信号处理器的重要单元，完成一次乘累加运算需要两个周期，可以显著的增强其数据处理能力。MAC 单元支持乘法、乘累加\减、双精度加减运算等，能够快速完成滤波、相关、变换以及双精度运算等操作。MAC 单元主要包括乘法器、移位器、累加器、4 个 56bit 乘累加寄存器，支持截断、饱和处理。结构如图 2-15。操作数来源于寄存器或者存储器，防止多次乘累加可能造成的溢出，累加寄存器进行了 8bit 的保护位扩展，最终可以选择性的对计算结果进行饱和处理或者取整（四舍五入）处理。

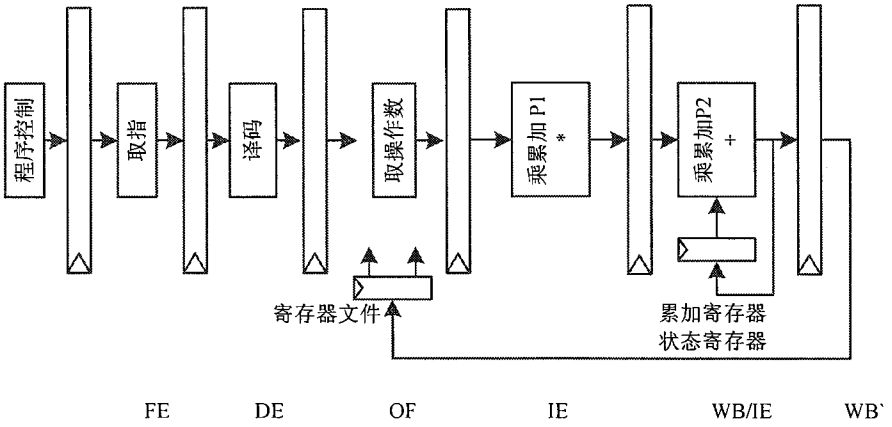


图 2-14 MAC 单元指令流水线

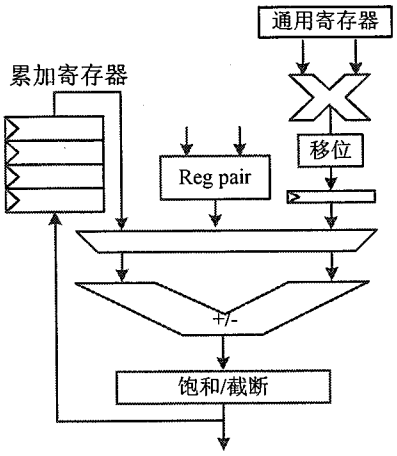


图 2-15 MAC 单元

2.2.4 AGU 单元

数据存取类指令完成程序对数据存储器的访问，包括数据读取、数据写回，此外还支持单数据、多数据的访问。数据存取指令的核心在于访问地址产生单元 AGU。FlexEngine 采用改进的哈佛结构，包括两个独立的数据存储器， AGU 单元根据译码信号产生数据存储器的访问地址，有两组总线分别连接到处理器的两块数据存储器，允许同时对它们进行访问，在一个时钟周期内存取两个数据。

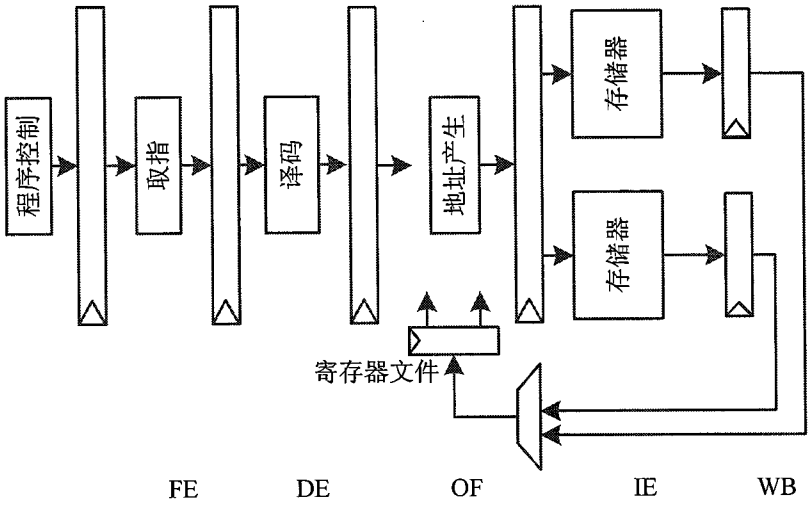


图 2-16 数据存取指令流水线

AGU 的作用是根据指令解析的数据和寻址方式，计算数据访问的地址。它是七种寻址模式的执行单元，包括立即寻址（Direct）、寄存器间接寻址（Register indirect）、偏移量寻址（Offset）、基址变址寻址（Based indexed）、地址后加（Post-inerease）、地址先减(Pre-subtract)、和模寻址(Modulo)。地址产生单元内含

所有地址计算相关的特殊寄存器，统称为地址寄存器，包括：基址寄存器、栈底地址寄存器、栈顶地址寄存器、步长寄存器、堆栈指针。地址产生单元的具体结构，如图 3.4 所示。

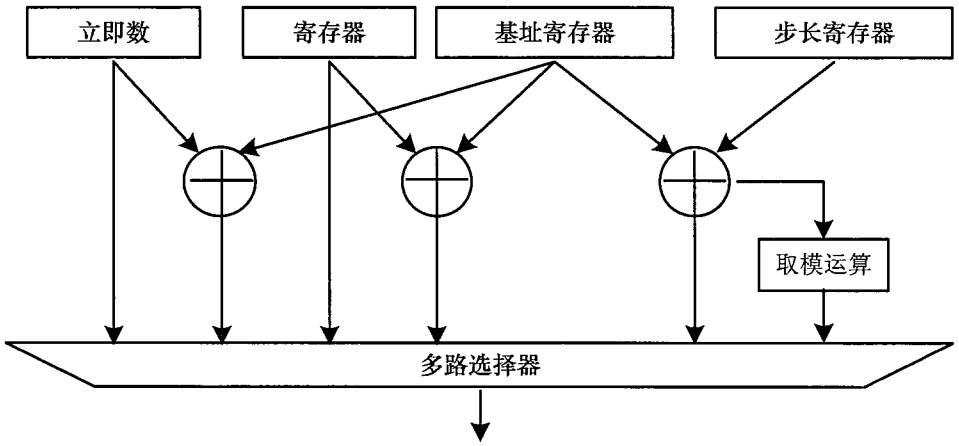


图 2-17 地址产生单元的结构框图

如数字助听器算法中的复数以及功率谱双倍存储器字长，通过将复数的实部虚部以及功率谱的高位与低位分别存储在两块数据存储器的相同地址，通过并行数据存取指令，可以在单周期完成复数以及功率谱双字长数据的存取。并行数据存取指令也可以在单周期存取两个数据。

FlexEngine 的 7 种寻址模式，能够灵活的满足语音处理算法中的寻址需求。地址先减或后加在进行存取指令之前或之后自动改变地址寄存器的值，能加快数组索引操作。FlexEngine 支持两个循环缓存地址寄存器，能定义两个以任意起点，任意长度的循环缓冲区，加速 IIR、FIR 滤波、FFT 变换等 DSP 算法。

2.2.5 PCU 单元

程序控制类指令，主要包括跳转、子函数调用、子函数返回，控制整个算法流程的变化。

程序控制类指令的执行依赖于程序控制单元 PCU。PCU 根据 PC 状态机产生下一条要执行指令的程序存储器访问地址。程序地址产生单元的结构，如图 3.11 所示，其核心就是一个多路选择器，选择控制信号由程序状态机单元，通过检测处理器执行状态产生；程序地址产生单元在选择信号的控制下，从多种地址

产生的方式中选择合适的地址，作为程序执行的下一条指令地址，送去访问程序存储器。从而实现了算法程序中指令执行顺序的多样性。

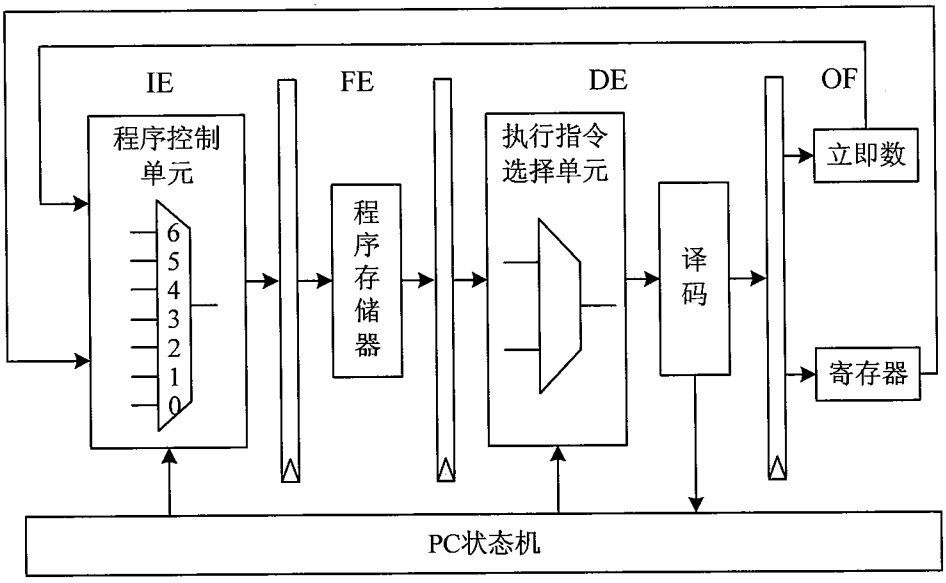


图 2-18 程序控制类指令的执行流程

程序流程控制指令 j/call/ret 会引起程序指针跳转，跳转一旦发生，会带来跳转惩罚。指令延时槽紧跟在程序控制指令后，程序控制指令在跳转至目的地址前，需要执行完延时槽内的指令，默认指令延时槽由硬件自动填充 3 条空指令。因此可以将程序跳转前一些不影响程序控制流与结果的指令放在跳转指令后内的延时槽，程序控制指令在执行完延时槽指令后跳转到目标地址。

2.3 FlexEngine 的性能优势

综上所述，FlexEngine 的指令集包括了算术逻辑运算、数据传送、程序控制等多种基本指令，以下特性使 FlexEngine 适用于音频处理应用：

（1）很好的支持密集乘法运算。FlexEngine 具有独立的 MAC 单元，可以在两个周期完成乘累加运算，保护位扩展的累加寄存器可以防止多次乘累加运算造成的结果溢出，因此 FlexEngine 非常适合音频处理中的滤波运算，如 FIR、IIR 等。

（2）加速存储器数据访问。FlexEngine 采用改进的哈佛结构，程序和数据存储器分离，其中数据存储器又分为两块，可以在一个周期实现两个数据的读/

写操作。

（3）灵活多样的寻址模式能够满足音频处理算法对寻址方式的较高需求。FlexEngine 的地址产生单元 AGU 支持模寻址等 7 种寻址模式，可以快速的实现音频处理算法，例如 IIR、FIR 滤波、FFT 变换等 DSP 算法。

（4）条件执行指令可以减少程序中的跳转指令，减小流水线的跳转开销。

2.4 FlexEngine 的低功耗 ASIP 设计

由于 FlexEngine 是一个基本指令集处理器核，要满足数字助听器的低功耗需求，仍需要对 FlexEngine 进行以下低功耗 ASIP 改进，这些问题会在后续章节进行介绍：

（1）完善其软件开发工具，将高级语言实现的应用程序编译为处理器的可执行目标代码，用来快速进行应用仿真，并对仿真过程进行分析与数据统计，提供 ASIP 设计反馈，指导专用指令集处理器 FlexEngine 的开发；

（2）基于数字助听器应用程序仿真，提取算法特征，针对程序热点代码添加专用指令和硬件加速单元，提高处理器的运行效率，减少执行时间，从而节省功耗；

（3）在 FlexEngine ASIP 设计的各阶段综合使用低功耗技术，实现数字助听器的低功耗 ASIP 设计。

2.5 本章小结

本章主要介绍了本文专用指令集处理器设计的目标应用数字助听器系统、以及基本指令集处理器核 FlexEngine，至此形成了本文低功耗数字助听器 ASIP 设计的平台。随后将在专用指令集处理器设计方法的指导下，针对数字助听器应用，对 FlexEngine 处理器核进行指令集优化与低功耗设计，最终获得低功耗数字助听器 ASIP 与软件开发工具。

参考文献

[1]Hamacher V, Chalupper J, Eggers J, et al. Signal processing in high-end hearing aids: state of the art, challenges, and future trends[J]. EURASIP Journal on Applied Signal Processing, 2005, 2005: 2915-2929.

[2]于增辉，黑勇，薛金勇，等. 助听器多通道宽动态范围压缩的低功耗硬件实现[J]. 哈尔滨工程大学学报，2012，33(1)：106-111.

[3]于增辉, 黑勇, 陈黎明, 等. 多通道数字助听器算法及低功耗 VLSI 设计[J]. 微电子学与计算机, 2012, 29(004): 14-18.

[4]Herbig R. Research on Hearing Aid Processing Delays[J]. Hearing Review, 2010.

[5]Brennan R, Schneider T. A flexible filterbank structure for extensive signal manipulations in digital hearing aids[C]//Circuits and Systems, 1998. ISCAS'98. Proceedings of the 1998 IEEE International Symposium on. IEEE, 1998, 6: 569-572.

[6]Kates J M. Digital hearing aids[M]. Plural Pub., 2008.

[7]Ngo K, Doclo S, Spriet A, et al. An integrated approach for noise reduction and dynamic range compression in hearing aids[C]//Proc. 16th European Signal Processing Conference (EUSIPCO). 2008.

[8]Loizou P C. Speech enhancement: theory and practice[M]. CRC, 2007.

[9]Hennessy J L, Patterson D A. Computer architecture: a quantitative approach[M]. Morgan Kaufmann, 2011.

[10]Glökler T, Meyr H. Design of energy-efficient application-specific instruction set processors[M]. Springer, 2004. pp. 45-46

[11]Henkel J, Parameswaran S. Designing embedded processors: a low power perspective[M]. Springer, 2007.

第3章 处理器软件开发工具设计

在 ASIP 设计中，软件开发工具如编译器（compiler）、汇编器（assembler）、链接器（linker）和仿真器（simulator）都是必需的[1][2]。应用程序通过软件开发工具编译为可执行的目标程序，经过仿真，提取应用程序的特点，指导 ASIP 的指令集优化与架构优化。在 ASIP 设计完成后，软件开发工具可以极大的方便基于 ASIP 的应用开发。本章对处理器的软件开发工具设计进行了详细的介绍，包括编译器的设计、二进制工具集的设计、以及仿真器的设计。

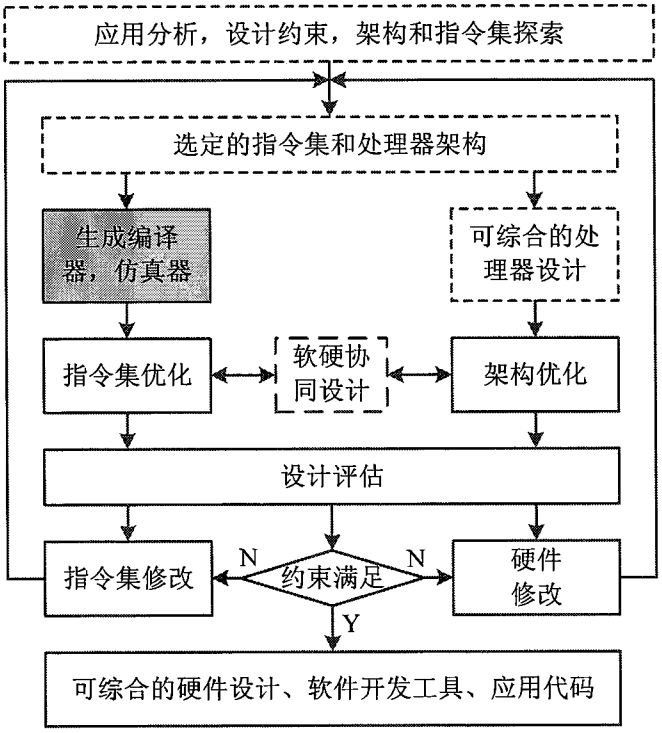


图 3-1 软硬协同的 ASIP 设计流程

处理器的软件开发工具包括编译器、二进制工具集（binary utilities）和仿真器，如图 3-2 所示。编译器执行高级语言到汇编语言的转换；汇编器将用户编辑的或编译器输出的汇编语言源文件转换为可重新分配地址的机器语言目标文件；链接器按照用户指定的程序和数据存放地址，把具有浮动地址的目标文件重新定位链接，映射到实际物理存储器中，输出可执行的目标文件；仿真器非实时的模拟程序的运行过程，方便用户调试。

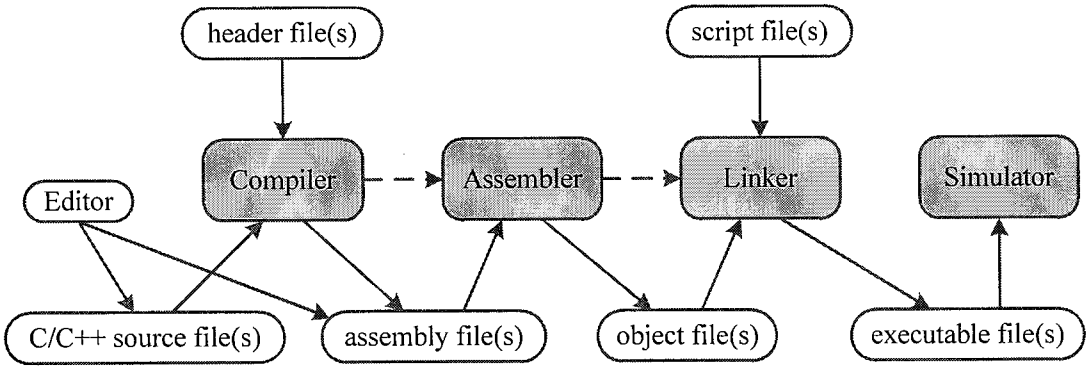


图 3-2 处理器软件开发工具

3.1 编译器设计

编译器运行在一个主机平台上，为另一个不同的目标机平台产生可执行代码。有了编译器，就可以在通用主机平台(如 Linux、windows 等)上进行嵌入式系统的应用开发，生成的可执行代码在嵌入式目标平台上执行。整个编译过程包含若干步骤，每个步骤把源程序的一种表示方式转换成另一种表示方式，一个典型的编译过程如图 3-3[3]。

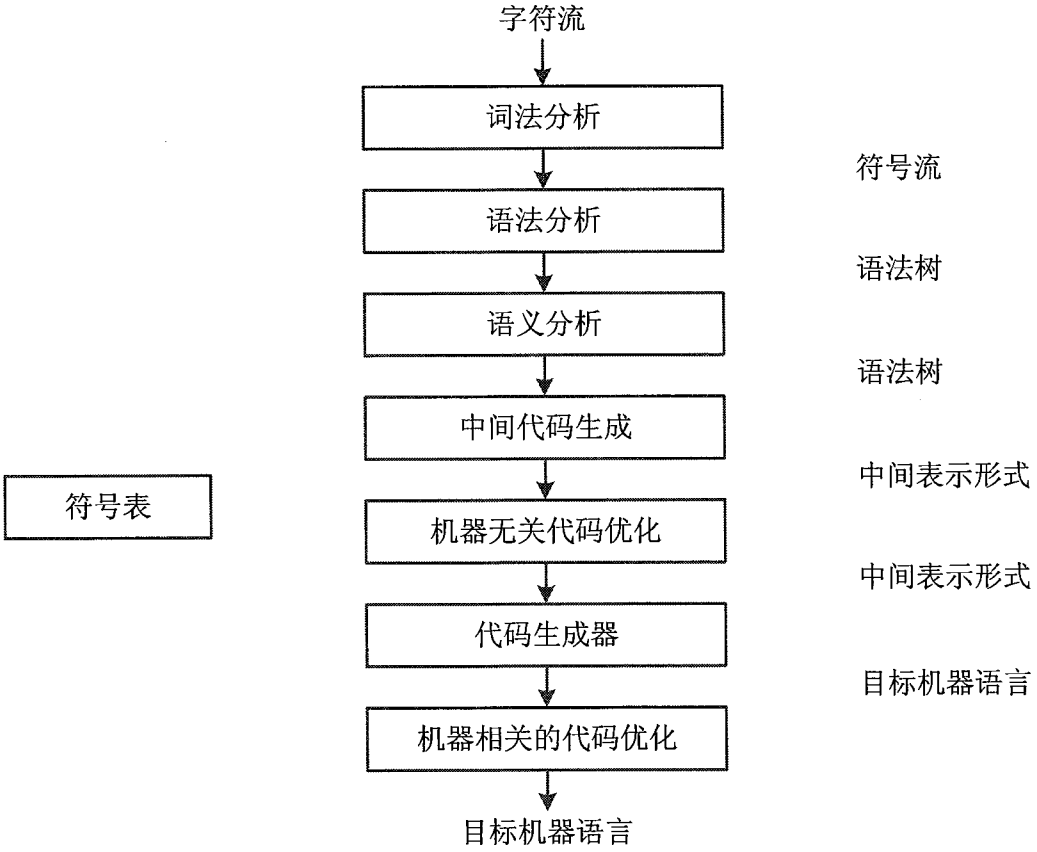


图 3-3 一个编译器的各个步骤

词法分析读入组成源程序的字符流，将其组织成有意义的词素序列；语法分析使用词法分析器生成的词法单元创建树形的中间表示（Immediate Representation，IR），描述程序的语法结构；语义分析使用语法树和符号表信息检查源程序是否和语言定义的语义一致，如类型检查等；中间代码生成将源程序翻译成一种中间表示，即一个明确的低级或者类机器语言，该中间表示易于生成且能够被轻松转换为目标机语言；机器无关的代码优化试图改进中间代码，生成等价的优化的中间代码，优化意味着更快、更短或者功耗更低的目标代码；代码生成器将源程序的中间表示映射到目标机器语言，为程序使用的每个变量分配寄存器或者内存，将中间指令翻译为完成相同任务的机器指令序列；符号表记录源程序中使用的变量的名字，并收集变量的各种属性信息，如类型、作用域、参数数量和类型、参数的传递方式、以及返回类型等。

现代化结构的编译器为了便于扩展，通常将设计分为前端(front end)和后端（back end）两部分，如图 3-4。前端进行机器无关的词法分析、语法分析、语义分析、目标机无关的代码优化，生成源程序的中间表示，通过添加前端可以使编译器编译多种高级语言；后端根据机器特性进行寄存器分配、指令排序、目标代码生成与优化，将语法树映射为目标机器代码，扩展后端则可以使编译器生成更多种目标机的机器语言。

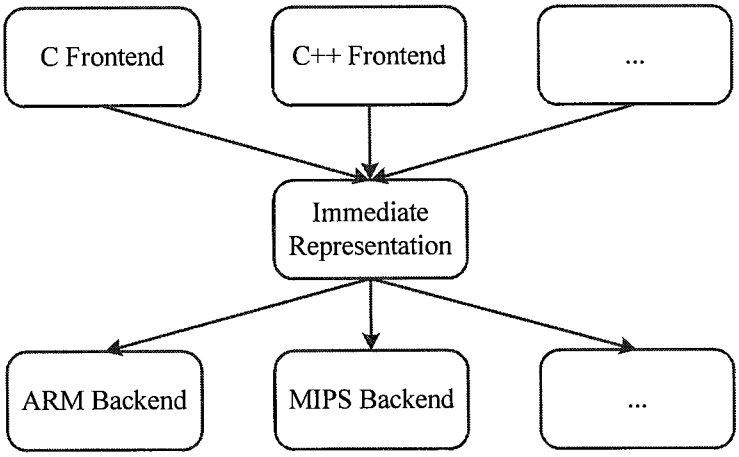


图 3-4 编译器架构

编译器原理复杂、开发周期长、成本高，开发全新编译器需要付出高昂的代价，满足不了嵌入式系统开发对市场快速变化的要求。利用成熟的可重定位编译器，针对目标处理器的体系结构进行移植，生成所需的编译器，是目前开发编译

器比较合理的方法。国外对编译器的架构进行了大量的研究，并针对并行、多发射等进行代码优化，产生了很多优秀的编译器架构，如 GCC[4]，Open64[5]，LLVM[6]，IMPACT[7]，LCC[8]，SUIF[9]，Trimaran[10]，Zephyr[11]等。

国内在编译器架构上也进行了一些研究，但主要是在可重定向编译器架构基础上进行研究和应用。如 Intel 公司和中国科学院计算所合作开发的 ORC[12]编译器，基于开源编译器 open64，针对 IA-64 的结构特点，以 IPF(Itanium Processor Family)处理器为目标，其主要优点是机器代码级的明确并行、预测执行、控制和数据的推测以及软件流水等。

表 3-1 国内在编译器架构基础上的研究与应用

单位	年份	编译器框架	研究应用
国防科学技术大学	2006	IMPACT	VLIW DSP 后端移植
国防科学技术大学	2007	SUIF	多媒体 SIMD 处理器后端移植
哈尔滨工程大学	2006	GCC	嵌入式微处理器 C*CORE 后端移植
上海交通大学	2007	LLVM	以 ARM 为例，探索工具链的自动生成
清华大学	2007	ORC	ASIP 后端移植
清华大学	2010	Open64	以 PowerPC 为例，探索后端移植方法
中科院声学研究所	2009	IMPACT	VLIW DSP(SUPERV)后端移植
浙江大学	2009	LCC	ASIP(SPOCK)后端移植
哈尔滨工业大学	2010	LLVM	方舟 ARCA3 后端移植
哈尔滨工业大学	2010	GCC	方舟 ARCA3 后端移植
中国科学院计算所	--	GCC/ORC	龙芯后端移植

GCC、Open64、LLVM 是工业界较多使用的编译器。GCC 在嵌入式领域应用最广泛，源代码开放，结构清晰，支持平台众多，是工业界普遍认可的编译器，可重定向的主要目标包括 CISC、RISC，GCC 容易上手，可能会比较快地开发出一款适合的编译器；但是 GCC 为了可扩展性、健壮性等方面的目的，缺乏较多的程序分析和优化工作。

Open64 前端和 GCC4 兼容，支持多种高级语言，后端支持多种体系结构，采用了五层中间表示，有非常好的优化支持，便于各种优化算法，可以生成非常高质量的代码，工业界普遍认可，在通用处理器平台上有更好的性能表现，是

AMD 的产品级编译器；但其结构复杂，上手较为困难。

LLVM 与 GCC 兼容，支持多阶段优化策略，提供最大化的性能，设计高度模块化，具有优良的代码接口，已经应用到 Apple 公司的产品开发中。LLVM 的目标处理器包括 CPU、SPU、GPU，无论是要添加一个优化到编译器中，还是要为编译器添加一个后端，都有规范的文档可以遵循，比 GCC 更加容易上手。

本文即是在开源编译器架构 LLVM 的基础上进行 FlexEngine 的后端移植，利用 LLVM 架构的可重用部件，通过添加目标机器后端获得专用指令集处理器 FlexEngine 的编译器，也是研究编译器自动生成的基础。

3.1.1 LLVM 编译器框架

LLVM（Low Level Virtual Machine）是美国伊利诺斯大学开发的开放源代码编译器架构，程序设计使用 C++语言，致力于程序整个生命周期（编译时，链接时，执行时等阶段）的优化。LLVM 最初的设计只支持 C/C++语言，而后随着 LLVM 的成功出现了众多的语言前端，包括 Objective-C、Fortran、Ada、Haskell、Java bytecode、Python、Ruby、ActionScript、GLSL、Clang 等，如今正逐渐被越来越多的科研机构和商业项目使用。LLVM 编译器架构对开发者透明，具备标准编译器系统全部的功能，提供大量用于建立编译器的可重用模块，可帮助减少开发新编译器的工作量。

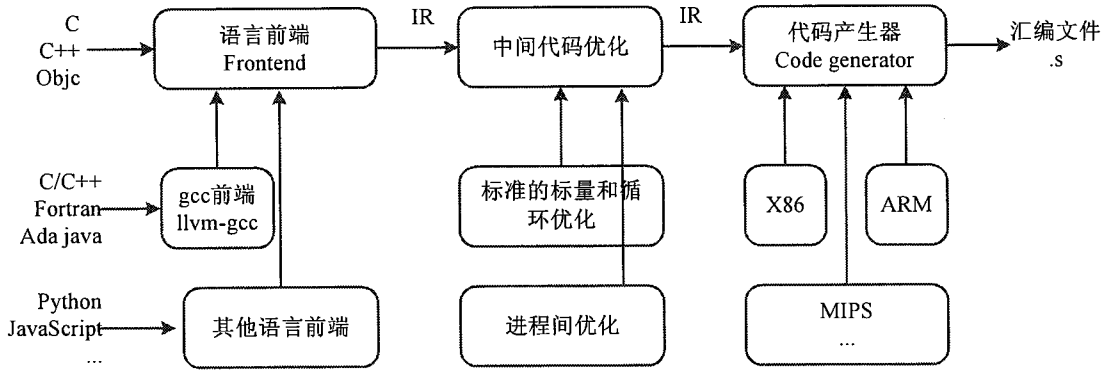


图 3-5 LLVM 编译器框架

3.1.1.1 语言前端

LLVM 最初重用了 GCC 的前端，针对 GCC 进行了代码优化，使整个程序

的全局优化成为可能。由于 LLVM 的流行，越来越多的语言前端正在不断开发，通过使用这些不同的语言前端，LLVM 目前可以编译 C/C++、Ada、D、Fortran、Objective-C 等语言。

语言前端的功能是将源语言程序转换为 LLVM 虚拟指令集，主要完成以下功能：

(1) 执行针对程序语言的优化。

(2) 转换源程序至 LLVM 中间表示

(3) 在模块级（module），调用 LLVM 遍处理进行全局优化。LLVM 优化以库的形式存在，易于前端使用。

clang[13]是一个全新的 C/C++/Objective-C 语言编译器，利用 C++代码编写，基于 LLVM 发布于 LLVM BSD 许可证下，与 GCC 兼容性。但是 GCC 的代码陈旧，整个编译过程的中间信息重用性差，不易集成，开发难度比较大；鉴于此，在 clang 的设计过程中，采用了基于库的模块化设计，结构清晰简单、易于 IDE 集成及其他用途的重用，并且 clang 具有比 GCC 更加良好的错误诊断信息。Clang 和 LLVM 的组合，可以用来取代 GCC 的 C/C++/Objective-C 编译器，且已经应用于 Apple 公司的产品开发中。

3.1.1.2 中间表示

LLVM IR（LLVM Immediate Representation）[14]是 LLVM 编译各阶段的通用中间代码表示。LLVM IR 容易定义与生成、易于理解，与源程序语言和代码生成器的目标机器无关，提供安全的数据类型，通用（universal）的低级操作（low-level operations）使得 LLVM IR 具备灵活表示所有高级语言的能力。

LLVM IR 有三种等价的存在形式：

（1）可以阅读的汇编代码，方便开发人员进行代码调试；

（2）存在磁盘的二进制字节码，用于 JIT（Just-In-Time）编译器在本机执行应用程序代码；

（3）存在内存的编译器中间表示形式，为编译器分析与转换过程提供高效

的中间代码。

LLVM 中间表示是一种三地址代码(three-address code)[15]，由一组类似于RISC(RISC-like)的汇编语言指令组成，每个指令具有三个运算分量，每个运算分量都像一个寄存器，右部最多只有一个运算符，编译器为每个指令计算得到的值生成一个临时名字存放。

1） LLVM 的类型系统

LLVM 类型系统是中间表示的重要组成部分，包含了 LLVM 系统所有支持数据类型的定义，由原子类型（Primitive Types）以及衍生类型（Derived Types）两部分组成（表 3-2）。原子类型是 LLVM 类型系统的基础，衍生类型是原子类型的扩展。

表 3-2 LLVM type system

Primitive Types	
Integer Types	i1, i2, i3, ... i8, ... i16, ... i32, ... i64, ...
Floating Point Types	float, double, x86_fp80, fp128, ppc_fp128
Void Types	
Label Types	
Metadata Types	
Derived Types	
Aggregate Types	array, structure, packed structure, union, vector
Function Types	
Pointer Types	
Opaque Types	

2） LLVM 的程序结构

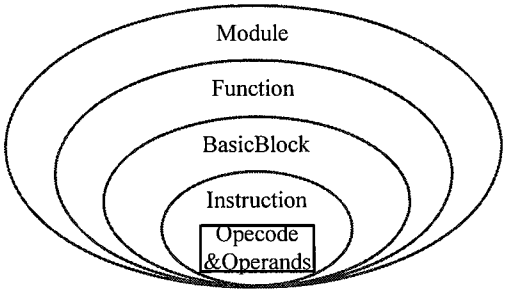


图 3-6 LLVM 的程序结构

LLVM 程序由若干模块（module）组成，高级语言源程序以模块为单位转换为 LLVM 程序。每个模块由函数、全局变量和符号表入口组成，是编译、分析

与优化的基本单位。

函数（Function）通常对应于 C 语言中的函数，由若干基本块组成，基本块构成了函数的控制流图。

基本块（BasicBlock）由若干指令组成，以终结指令（Terminator Instruction）结束，如跳转指令或者函数返回指令。

指令（Instruction）由操作码和操作数向量构成，包含操作数与结果的类型信息。LLVM 虚拟指令集包含终结指令、算术指令、逻辑指令、移位指令、向量指令、存储器访问指令、类型转换指令和其他指令等（表 3-3）。

表 3-3 LLVM 虚拟指令集

Terminator Instructions: ret, br, switch, indirectbr, invoke, unwind, unreachable
Binary Operations: add, fadd, sub, fsub, mul, fmul, udiv, sdiv, fdiv, urem, srem, frem
Bitwise Binary Operations: shl, lshr, ashr, and, or, xor
Vector Operations: extractelement, insertelement, shufflevector
Aggregate Operations: extractvalue, insertvalue
Memory Access and Addressing Operations: alloca, load, store, getelementptr
Conversion Operations: , trunc .. to, zext .. to, sext .. to, fptrunc .. to, fpext .. to, fptoui .. to, fptosi .. to, uitofp .. to, sitofp .. to, ptrtoint .. to, inttoptr .. to, bitcast .. to
Other Operations: icmp, fcmp, phi, select, call, va_arg

3.1.1.3 代码生成

LLVM 目标无关的代码生成器（Code Generator）可以为基于标准寄存器设计的处理器生成高效的代码，代码生成过程如图 3-7。代码生成器主要包含 3 个

任务：指令选择、寄存器分配、以及指令排序。指令选择使用合适的目标机指令实现 LLVM IR 语句；寄存器分配为目标机指令的操作数与结果分配具体的寄存器；指令排序安排指令的执行顺序；最后，经过代码发射将目标机指令按照汇编语言格式输出。

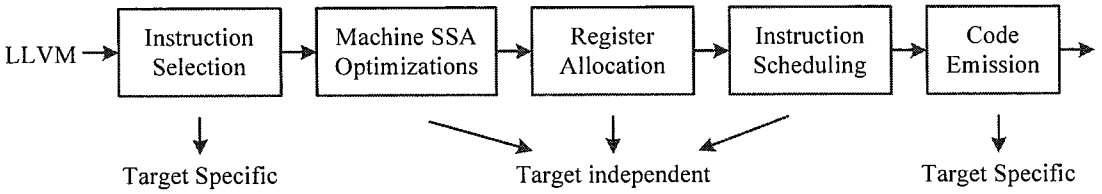


图 3-7 代码生成过程

LLVM 为代码生成器提供一系列的可重用组件，转换 LLVM 中间表示至特定目标机器的机器码，主要包括以下几个组件：

（1）抽象的目标机描述接口，如表 3-4，用于描述目标机的重要特征，如指令集、寄存器文件，与特定目标机无关。

表 3-4 抽象的目标机描述接口

Class	Description
TargetMachine	提供虚方法，用于访问不同的目标机描述类实例
TargetData	描述目标机的数据类型，对齐，指针类型，数据尾端
TargetRegisterInfo	描述寄存器文件
TargetInstrInfo	描述指令集
TargetFrameInfo	描述目标机的堆栈布局
TargetLowering	描述 LLVM IR 到目标机指令的转换
TargetSubtarget	描述目标机的子系列

（2）类，用于描述目标机的机器语言。这些类的设计应该足够抽象，以至于可以用来描述任意目标机。

表 3-5 类

类	描述
MachineInstr	目标机指令
MachineBasicBlock	目标机基本块
MachineFunction	目标机函数

（3）目标机无关的算法，用于目标代码产生的各个阶段，如寄存器分配、指令排序、堆栈表示等。

（4）特定目标机器的代码生成器实例化。利用 LLVM 提供的组件描述特定的目标机器，添加遍处理，为特定目标机器建立完整的目标代码生成器。

3.1.2 编译器实现

本文基于 Clang 与 LLVM 组合，利用 LLVM 系统的代码生成器组件，实现 FlexEngine 的编译器，将 C 语言源程序转换为 FlexEngine 的目标机器语言，过程如图 3-8。

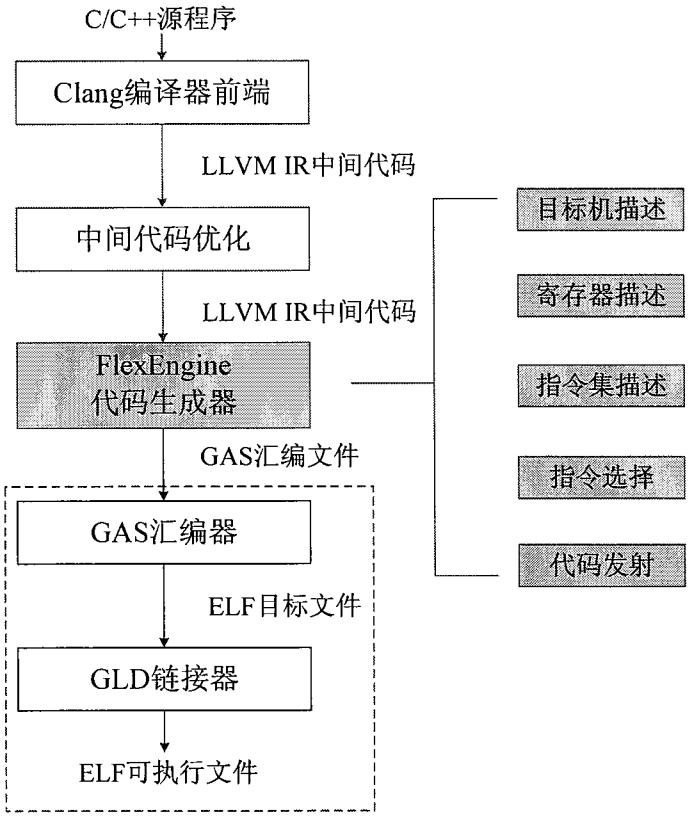


图 3-8 基于 LLVM 移植编译器

设计的开发环境采用 Microsoft Visual Studio 2005，使用目标描述语言(target description language，TD)和 C++。TD 语言用于抽象地描述目标机，可以减少 C++代码量，简化代码、降低系统的复杂性。但是 TD 语言仍在不断完善中，不能完全描述代码生成器，因此开发过程仍需要 C++代码，主要用于指令选择以及 TD 描述不能完成的工作。

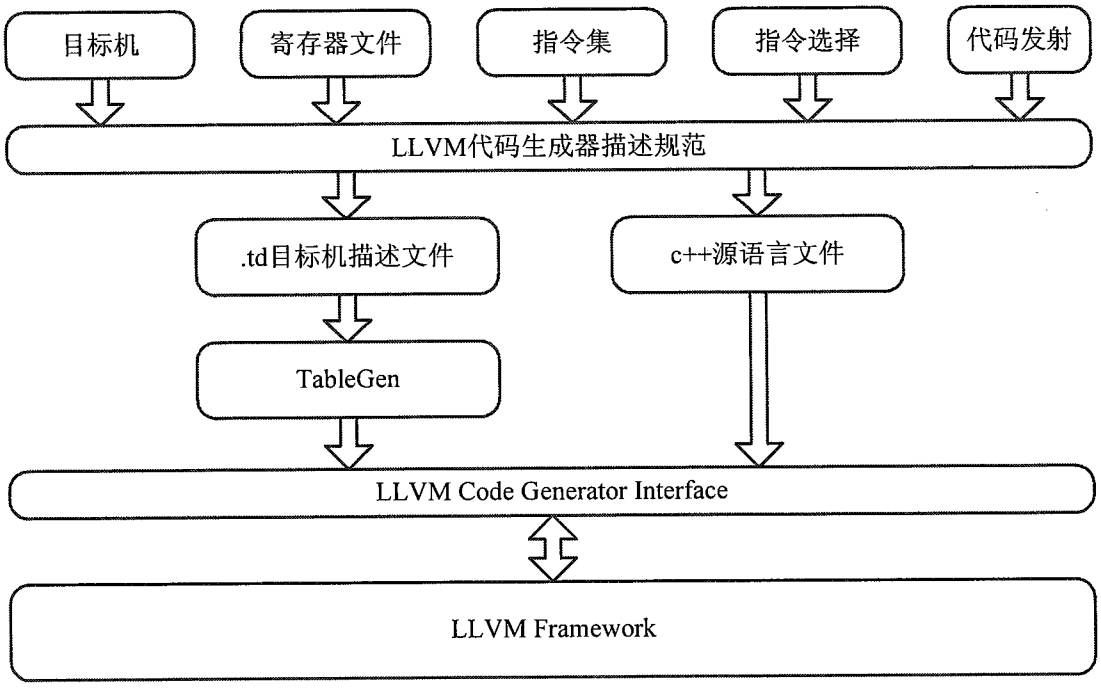


图 3-9 LLVM 代码生成器移植结构框图

1） TableGen 工具

TableGen[16]是一套用来描述信息记录（record）的数据结构和语法规范，可以在众多信息记录中提取公共特征，进而用更加抽象但简单灵活的声明来替代逐条的信息记录，这些抽象的声明最终经 TableGen 解析为逐条信息记录实例。所以 TableGen 主要用于组织大量的信息记录，解决信息记录描述过程中的重复性工作，同时减少出错的概率。

TableGen 文件由记录组成，每个记录具有唯一的名称与数据，数据包含了 TableGen 生成记录实例的全部信息。记录包含类（class）和定义（definition）两种存在形式：类是抽象的记录，用来描述记录的抽象数据结构，如目标机的"Register"，"RegisterClass"和 "Instruction"等；定义是记录的具体实现形式，由关键字‘def’标识，不包含未定义的数据。

TableGen 拥有一个完备的数据类型结构，如低级类型‘bit’、高级类型‘dag’，其灵活性可以方便有效的描述不同领域内的信息记录。

表 3-6 TableGen 的数据类型

类型	描述
bit	bool 类型
int	32-bit 整数类型
string	字符串类型
bits<n>	任意 n bit 整数类型
list<ty>	ty 类型的列表容器；ty 可以是任意类型，甚至列表类型本身
Class	在类型的定义中，标识数值类型为类的部分
dag	可嵌套的 DAG
code	大块代码，不需要转义字符

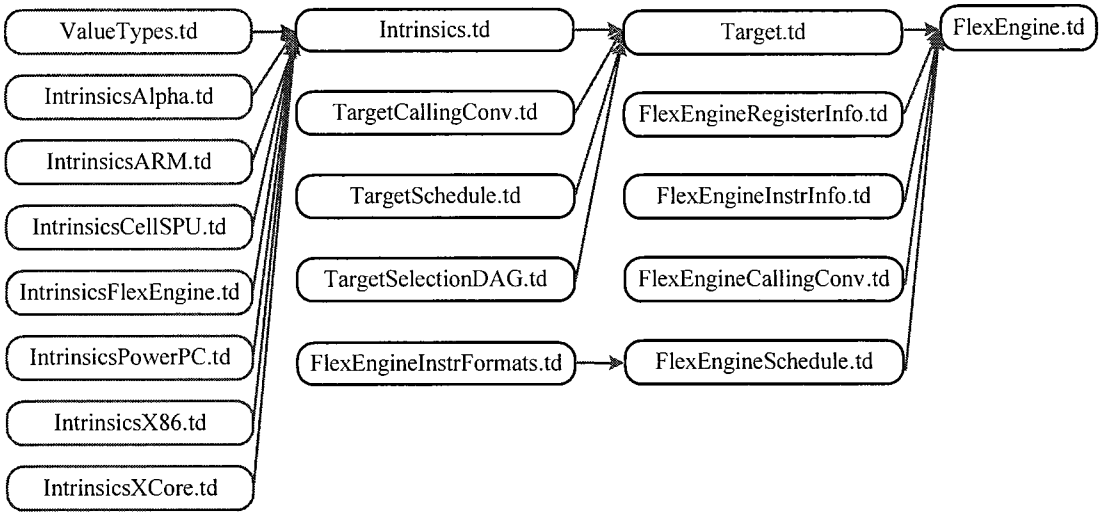


图 3-10 .td 文件结构

TD 语言即是遵循 TableGen 语法规范的目标机描语言，用来描述目标机的抽象属性，描述文件一般保存为.td 文件，文件中可以使用关键字 include 包含其他文件。.td 文件经 TableGen 解析为等价的 C++头文件（.inc 文件），嵌入到 C++源文件中。描述 FlexEngine 的.td 文件（图 3-10），通过以下配置生成对应的 C++代码片段文件，嵌入到 C++源文件中，完成目标处理器的抽象接口实现（图 3-11）：

```
set(LLVM_TARGET_DEFINITIONS FlexEngine.td)
//寄存器描述
tablegen(FlexEngineGenRegisterInfo.h.inc    -gen-register-desc-header)
tablegen(FlexEngineGenRegisterNames.inc    -gen-register-enums)
tablegen(FlexEngineGenRegisterInfo.inc    -gen-register-desc)
//指令集描述
tablegen(FlexEngineGenInstrNames.inc    -gen-instr-enums)
tablegen(FlexEngineGenInstrInfo.inc    -gen-instr-desc)
```

tablegen(FlexEngineGenAsmWriter.inc -gen-asm-writer)

tablegen(FlexEngineGenDAGISel.inc -gen-dag-isel)

//调用约束描述

tablegen(FlexEngineGenCallingConv.inc -gen-callingconv)

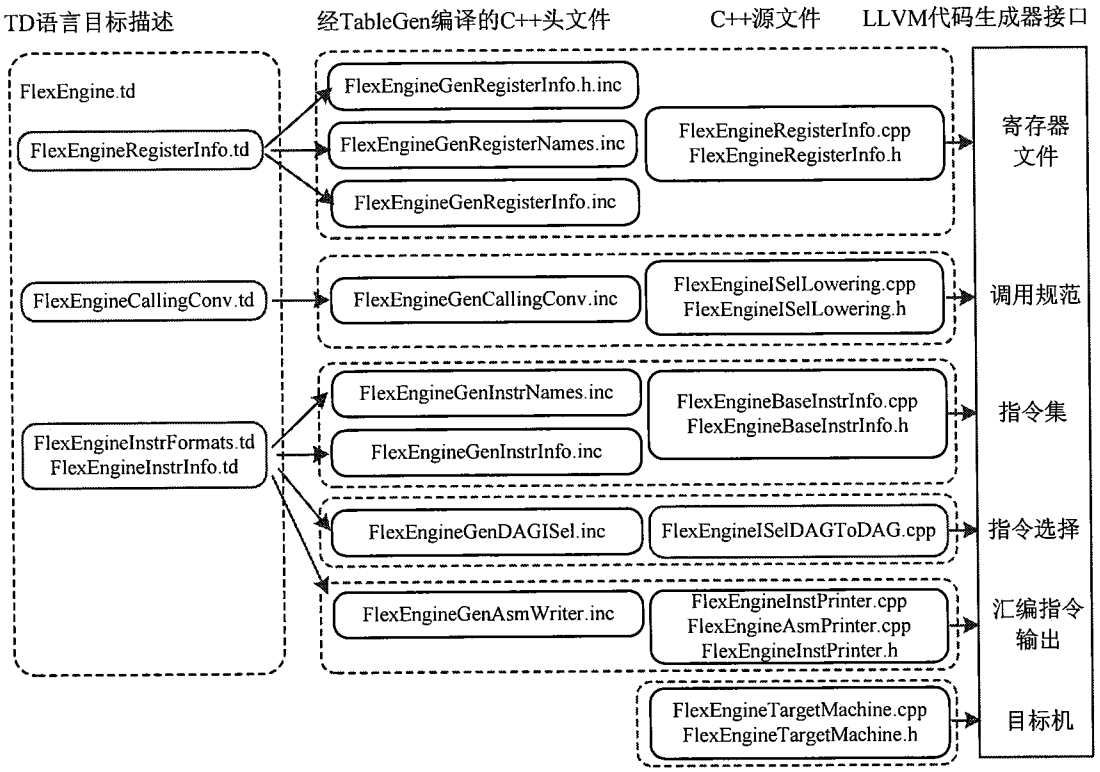


图 3-11 LLVM 代码生成器项目构成

利用 Cmake[17]，在 Microsoft Visual Studio 2005 中创建 LLVM 工程，添加 LLVM FlexEngine 代码生成器的源代码（图 3-11），实现过程如下：

（1）创建 TargetMachine 类的子类 FlexEngineTargetMachine，描述 FlexEngine 的特征，提供访问 FlexEngine 描述类的方法。

（2）描述 FlexEngine 的寄存器文件。利用 TableGen 将寄存器描述的 TD 文件 FlexEngineRegisterInfo.td 转换为包含寄存器定义、寄存器类、别名的 C++代码头文件；编程实现类 FlexEngineRegisterInfo，描述用于寄存器分配的寄存器类、寄存器之间的相互作用。

（3）描述 FlexEngine 的指令集。利用 TableGen 将指令集描述的 TD 文件 FlexEngineInstrFormats.td 和 FlexEngineInstrInfo.td 转换为包含指令操作码、操作数的 C++代码文件；编程实现类 FlexEngineInstrInfo，描述目标机器支持的机器

指令。

（4）描述 LLVM IR 到 FlexEngine 指令的选择与转换，将以 DAG 表示的 LLVM IR 转换为 FlexEngine 指令。FlexEngineInstrInfo.td 中描述了 FlexEngine 的指令集，每条指令的最后一个元素定义了该指令所能匹配的 LLVM IR 模式，利用 TableGen 将其解析为包含模式匹配的 C++ 代码；编程 FlexEngineISelDAGToDAG.cpp 文件，完成模式匹配，以及 DAG-to-DAG 的指令选择；编程 FlexEngineISelLowering.cpp 文件，替换或者删除 LLVM IR 中不支持的操作和数据类型。

（5）完成 LLVM IR 至 FlexEngine GAS 汇编格式转换的描述。在 FlexEngineInstrInfo.td 的指令定义中添加指令的汇编语言描述，编程 Asmprinter 子类 FlexEngineAsmprinter，完成 LLVM-to-assembly 的转换。

3.1.2.1 添加与注册目标机

LLVMTargetMachine 是 LLVM 代码生成器目标机实现的基类，是 TargetMachine 的子类，须由具体的目标机类实例化，实现目标机的各种虚方法，用来获取目标机各组成部件的信息。因此添加 FlexEngine 代码生成器后端，需要实现 LLVMTargetMachine 的目标机子类 FlexEngineTargetMachine，创建 FlexEngineTargetMachine.h 和 FlexEngineTargetMachine.cpp 文件。

（一）实现目标机各组件的访问方法（get*Info），如表 3-7。

表 3-7 目标机虚方法

虚方法	描述
getInstrInfo()	指令集信息访问接口
getRegisterInfo()	寄存器文件信息访问接口
getFrameInfo()	堆栈布局信息访问接口
getTargetData()	目标机数据类型信息访问接口
getSubtargetImpl()	子目标机系列信息访问接口

（二）为 FlexEngineTargetMachine 构造函数指定目标机描述字符串，指定以下信息：

（1）大小尾端：‘E’表示大尾端，‘e’代表小尾端。

（2）指针信息：‘p’后为指针信息，包括大小、ABI 对齐、优先对齐，如果

只有两个数据，则第二个数据同时表示 ABI 对齐与优先对齐。

（3）数据类型对齐：‘i’，‘f’，‘v’，‘a’，分别对应整数、浮点、向量、聚合体。‘i’、‘v’、‘a’后接 ABI 对齐与优先对齐，‘f’后接 long double 大小、ABI 对齐与优先对齐。

```
DataLayout(std::string("e-p:32:32-i64:32:32-n32"))
```

（三）通过 TargetRegistry 模板注册目标机：

```
Target llvm::TheFlexEngineTarget;
extern "C" void LLVMInitializeFlexEngineTargetInfo() {
  RegisterTarget<Triple::flexengine, /*HasJIT=*/false>
    X(TheFlexEngineTarget, "flexengine", "FlexEngine");
}
```

（四）配置 FlexEngine 的遍（Pass）信息。LLVM Pass 结构是 LLVM 系统的重要部分，执行编译转换和优化，编译过程中进行一系列的遍处理。通过在 FlexEngine 后端的实现过程中添加自定义遍处理，可以针对编译器和目标机的特殊结构进行编译转换和优化，获得高效的目标代码。在 FlexEngineTargetMachine.h 和 FlexEngineTargetMachine.cpp 文件中，需要将代码生成过程中执行的所有遍，添加到遍管理组件（Pass Manager）。LLVM 提供了多种不同的方法用以添加遍，如表 3-9。

表 3-8 4 种不同类型的遍处理

Four types of Pass	
ModulePass	通常的函数间（interprocedural）遍处理
CallGraphSCCPass	自下至上的调用树遍处理
FunctionPass	函数遍处理
BasicBlockPass	基本块遍处理

表 3-9 LLVM 添加遍的方法

方法	描述
addInstSelector	添加任意用于 LLVM IR 的转换遍处理；然后添加指令选择遍转换 LLVM IR 至目标机指令
addPreRegAlloc	添加遍处理，在寄存器分配前执行
addPostRegAlloc	添加遍处理，在寄存器分配后、prolog-epilog 插入前执行
addPreSched2	添加遍处理，在 prolog-epilog 插入后、第二遍指令排序前执行
addPreEmitPass	添加遍处理，在指令发射前执行
addCodeEmitter	添加遍用于代码发射

如在 addInstSelector 方法中添加 FlexEngine 的指令选择遍处理：

```
bool FlexEngineBaseTargetMachine::addInstSelector
(PassManagerBase &PM,CodeGenOpt::Level OptLevel)  {
  PM.add(createFlexEngineISelDag(*this, OptLevel));
  return true;
}
```

3.1.2.2 寄存器描述

寄存器描述用于描述 FlexEngine 的寄存器文件，以及寄存器之间的交互信息。所有描述包含在 FlexEngineRegisterInfo.td、FlexEngineRegisterInfo.cpp 、以及 FlexEngineRegisterInfo.h 文件中。

在 FlexEng ineRegisterInfo.td 中，定义了 FlexEngine 的寄存器，并对寄存器进行分类，以便在指令集的描述中，根据操作数的类型选则相应的寄存器类用于寄存器分配。通常可以将寄存器划分为整数寄存器、浮点寄存器或者向量寄存器。根据 FlexEngine 的特征定义了 32 个通用寄存器、32 个特殊寄存器、4 个累加寄存器，同时对其进行寄存器分类：通用寄存器类、累加寄存器类、地址寄存器类。通过 MethodProtos 和 MethodBodies 方法将通用寄存器类作为指令的操作数，参与寄存器分配。

FlexEngineRegisterInfo.td 通过 TableGen 解析生成头文件 FlexEngineGenRegisterInfo.h.inc 和 FlexEngineGenRegisterInfo.inc。

下面给出了对通用寄存器类以及通用寄存器 R0 的定义：

```
class FlexEngineReg<bits<5> num, string n> : Register<n> {
  field bits<5> Num;
  let Namespace = "FlexEngine";
}
def R0   : FlexEngineReg< 0, "r0">,   DwarfRegNum<[0]>;
……
def GPR : RegisterClass<"FlexEngine", [i32], 32, [ R0, R1, R2, R3,……
                                                  //Non-allocatable regs:
                                                  ……]> {
```

```
let MethodProtos= [{iterator allocation_order_end(const MachineFunction &MF)
    const;}];
let MethodBodies = [{GPRClass::iterator GPRClass::allocation_order_end(const
    MachineFunction &MF) const{return end()-1} }];//Don't allocate special
    registers
}
```

FlexEngineRegisterInfo.cpp 文件实例化 TargetRegisterInfo 类，实现寄存器文件的接口函数。

表 3-10 寄存器描述接口函数

接口函数	描述
getCalleeSavedRegs	返回被调用函数保存的寄存器列表
getCalleeSavedRegClasses	返回被调用函数保存的寄存器所属寄存器类
getReservedRegs	返回保留寄存器
hasFP	描述目标机是否具有帧指针寄存器（FP）
emitPrologue	在函数入口处插入入栈指令（prolog code）
emitEpilogue	在函数出口处插入出栈指令（epilog code）
eliminateCallFramePseudoInstr	使用目标机指令实现帧 setup/destroy 伪指令
eliminateFrameIndex	使用目标机帧栈指针实现抽象的虚拟帧指针

3.1.2.3 指令集描述

指令集描述用于完整的描述 FlexEngine 的指令集信息，包括每一条指令的操作码、结果类型、操作数个数、操作数类型、汇编语句格式、以及用于指令选择的模式匹配等。所有描述包含在 FlexEngineInstrFormats.td、FlexEngineInstrInfo.td、FlexEngineInstrInfo.h、以及 FlexEngineInstrInfo.cpp 文件中。

FlexEngine 指令集根据功能分为数据传输、算术运算、逻辑运算、流程控制、双精度运算、及自定制指令几类。同类型的指令具有相同的指令类型码、相似的操作数列表，因此使用 TD 语言按照指令分类描述指令集，首先为每类指令定义指令类，然后使用类定义实例化类内的指令。

下面给出了对算术类指令以及指令 ADD 的定义：

```
multiclass AsI1_arith<string opc, PatFrag opnode,  bit Commutable = 1>
{
  def ri : AInoP<(outs GPR:$dst), (ins GPR:$a, simm12:$b), DPFrm, IIC_iALU,
                 !strconcat(opc, "i"), "\t$dst, $a, $b",
```

```
                [(set GPR:$dst, (opnode GPR:$a, simm12:$b))]>
    def rr : AsI1< (outs GPR:$dst), (ins GPR:$a, GPR:$b), DPFrm, IIC_iALU,
                opc, "\t$dst, $a, $b",
                [(set GPR:$dst, (opnode GPR:$a, GPR:$b))]>
}
defm ADD:AsI1_arith<"add",BinOpFrag<(add node:$LHS,node:$RHS)>, 1>;
```

FlexEngineInstrInfo.cpp 文件实例化 TargetInstrInfoImpl 类，实现指令集描述的接口函数。

表 3-11 指令集描述接口函数

接口函数	描述
isMoveInstr	如果是寄存器到寄存器的数据拷贝，返回真值、源寄存器类、目的寄存器类
isLoadFromStackSlot	如果指令直接从堆栈加载数据，返回目的寄存器与堆栈帧索引
isStoreToStackSlot	如果指令直接向堆栈存放数据，返回源寄存器与堆栈帧索引
copyRegToReg	发射目标机指令完成寄存器间的数据拷贝
storeRegToStackSlot	添加指令完成寄存器到堆栈的存放
loadRegFromStackSlot	添加指令完成从堆栈加载到寄存器

3.1.2.4 指令选择

在代码生成的早期，LLVM IR 被转换为带有结点的 SelectionDAG，其结点是 SDNode 类的实例，利用寄存器文件与指令集描述的信息，通过模式匹配部分结点用目标机指令表示。尚未用目标机指令表示的 SDNode 结点，将在指令选择时转换为目标机指令。FlexEngineISelDAGToDAG.cpp 文件通过重载函数 select 进行更加复杂的模式（pattern）匹配，执行 DAG-to-DAG 的指令选择。而后，对于 SelectionDAG 中仍然不支持的操作和数据类型，在 FlexEngineISelLowering.cpp 文件中通过重载 TargetLowering 将其替换或者删除。

1) SelectionDAG 合法化

合法化阶段（Legalize phase）将 DAG 转化为目标机支持的数据类型和操作。对于目标机不支持的数据类型和操作，需要在 FlexEngineTargetLowering 类中将其转换为支持的数据类型和操作。首先，在 FlexEngineTargetLowering 的构造函

数中，使用 addRegisterClass 方法指定目标机支持的数据类型和相关联的寄存器类。FlexEngine 的构造器实现：

addRegisterClass(MVT::i32, FlexEngine::GPRRegisterClass);

在 include/llvm/CodeGen/SelectionDAGNodes.h 文件中，ISD 命名域下定义了所有的结点类型，一些结点通过指令描述的模式匹配映射到目标机指令，一些结点通过重载函数 select 执行的更加复杂的模式匹配映射为目标机指令，但是存在一些结点在目标机中找不到可以匹配的目标机指令，对于这些结点需要在 FlexEngineTargetLowering 的构造函数中添加 callback 方法，指示指令选择如何处理。TargetLowering 类定义了 callback 方法：

表 3-12 callback 方法

callback	描述
setOperationAction	General operation
setLoadExtAction	Load with extension
setTruncStoreAction	Truncating store
setIndexedLoadAction	Indexed load
setIndexedStoreAction	Indexed store
setConvertAction	Type conversion
setCondCodeAction	Support for a given condition code

callback 方法定义了三个参数，分别是操作、数据类型、LegalAction 类型的处理方式（Promote, Expand, Custom, 或者 Legal）。

（1）Promote：操作不支持某种数据类型，但是可以通过将数据类型提升至目标机支持的较大的数据类型来实现。如 FlexEngine 不支持 i1 类型带符号扩展的存取操作，可以将 i1 类型在存取前改变至更大的数据类型实现。

setLoadExtAction(ISD::SEXTLOAD, MVT::i1, Promote);

（2）Expand：操作不支持某种数据类型，但是可以通过将操作扩展来实现，即通过组合其他的操作来实现相同的操作，这些扩展的组合操作已经由 LLVM 框架给出。如 FlexEngine 不支持 i32 类型的符号数除法，可以将符号数除法通过其他操作的组合实现。

setOperationAction(ISD::SDIV, MVT::i32, Expand);

（3）Custom：对于一些操作，通过类型提升或者操作扩展无法实现，可以将

这些操作设置为 custom 类型，通过函数实现。

```
setOperationAction(ISD::SHL_PARTS, MVT::i32, Custom);
setOperationAction(ISD::SRA_PARTS, MVT::i32, Custom);
setOperationAction(ISD::SRL_PARTS, MVT::i32, Custom);
```

在 LowerOperation 方法中，对每一个 Custom 操作增加一个 case 语句，指定处理该操作的函数。

```
SDValue FlexEngineTargetLowering::LowerOperation
(SDValue Op, SelectionDAG &DAG) {
  switch (Op.getOpcode()) {
  default: llvm_unreachable("Don't know how to custom lower this!");
  ......
  case ISD::SHL_PARTS:        return LowerShiftLeftParts(Op, DAG);
  case ISD::SRL_PARTS:
  case ISD::SRA_PARTS:        return LowerShiftRightParts(Op, DAG);
  ......
  }
}
```

（4）Legal：对于目标机支持的结点，其操作不需要通过上述三种方法就可以完成指令选择，操作类型应该设置为 Legal，Legal 作为结点默认的属性通常省略。如 FlexEngine 支持 i32 的加减操作，将其属性设置为 Legal：

```
setOperationAction(ISD::ADD,    MVT::i32, Legal);
setOperationAction(ISD::SUB,    MVT::i32, Legal);
```

2) 调用约束

堆栈是内存中的一块连续存储空间，用来传递函数参数、存储局部变量、存储函数返回值、保存寄存器的值以供恢复使用。应用程序借用栈来支持函数（又称过程）调用，变量的存储按后进先出（LIFO）的方式进行。

FlexEngine 堆栈位于数据存储器，8 字节对齐，特殊寄存器 SP 为堆栈指针，规定为空升序栈，既 SP 指向第一个空闲单元，在内存中正向增长。压堆栈时，先将数据存入 datamem [SP]，后 SP 地址加 1；弹出堆栈时，先 SP 地址减 1，然后将数据从 datamem [SP]取出。函数调用和中断程序产生时，硬件自动将函数返

回地址压入堆栈，当函数调用结束或者中断返回时将，返回地址由硬件自动从堆栈弹出赋值给 PC，任意函数调用前或者中断产生前必须初始化 SP 寄存器。

调用约束（Calling Convention）规定：FlexEngine 的函数调用时通过通用寄存器 r0-r3 传递参数与返回值，被调用函数需要保存通用寄存器 r4-r11。

FlexEngineGenCallingConv.td 使用 TargetCallingConv.td 定义的接口，描述了 FlexEngine 的函数调用约束。TableGen 将其编译为 FlexEngineGenCallingConv.inc，包含在 FlexEngineISelLowering.cpp 中。调用约束需要指定的信息包括参数分配的顺序、参数和返回值的存放位置（寄存器或者堆栈）、可使用的寄存器、调用函数或者被调用函数是否展开堆栈。

如 FlexEngine 的返回值调用约束 RetCC_FlexEngine 和 C 调用约束 CC_FlexEngine：

```
def RetCC_FlexEngine_24Bit : CallingConv<[ //传递返回值
  CCIfType<[i8, i16], CCPromoteToType<i32>>,
  CCIfType<[i32],CCAssignToReg<[R0,R1,R2,R3]>>,//通过寄存器传递返回值
  CCIfType<[i64], CCAssignToRegWithShadow<[R0,R2], [R1,R3]>>
]>;
def CC_FlexEngine_24Bit : CallingConv<[//传递参数
  CCIfType<[i8, i16], CCPromoteToType<i32>>,
  CCIfType<[i32], CCAssignToReg<[R0,R1,R2,R3]>>,//通过寄存器传递参数
  CCIfType<[i64], CCAssignToRegWithShadow<[R0, R2], [R1, R3]>>,
  CCIfType<[i32], CCAssignToStack<4, 4>>//通过堆栈传递参数
]>;
```

指令选择阶段需要实现的目标机接口：

表 3-13 指令选择接口函数

接口	描述
createFlexEngineISelDag	建立遍，转换 DAG 至 FlexEngine-GAG
Select	完成复杂的模式匹配，转换 LLVM IR 至目标机指令
LowerCall	转换 call 为 DAG：callseq_start<-FlexEngineISD:CALL<-callseq_end，添加输入输出参数结点
LowerFormalArguments	转换形参至 DAG
LowerReturn	转换返回值至 DAG
LowerX	目标机不支持结点 X 的函数实现

3.1.2.5 汇编输出

在代码发射阶段，代码生成器利用 LLVM 遍输出汇编代码，将 LLVM IR 转换为 GAS 格式的汇编语言。

FlexEngineInstrInfo.td 中的指令定义添加了汇编字符串，通过 TableGen 生成 FlexEngineGenAsmWriter.inc 文件，其中包含了所有指令的汇编打印输出函数 printInstruction。FlexEngineAsmPrinter.cpp 文件中的 runOnMachineFunction 函数使用 printInstruction 打印输出每条指令的汇编语句。FlexEngineAsmPrinter 还重载了 printOperand、printMemOperand、PrintAsmOperand 和 PrintAsmMemoryOperand 等函数，实现不同指令操作数的打印输出。

def XOR : AsI1 < (outs GPR:$dst), (ins GPR:$a, GPR:$b), IIC_iALU, "xor", "\t$dst, $a, $b", [(set GPR:$dst, (xor GPR:$a, GPR:$b))]>;

3.1.2.6 LLVM 系统扩展

LLVM 支持内函数（Intrinsic Functions），在不改变 LLVM 语言转换过程的情况下扩展 LLVM 语言。内函数是 LLVM 语言的一部分，以保留字“llvm.”作为名称前缀，以函数形式存在，编译时被映射为一条或若干条指令。通过内函数重载可以使内函数支持不同的数据类型。

在专用指令集的设计过程中，针对应用程序频繁使用的指令或指令序列添加专用指令，需要编译器能够高效的将应用程序映射到专用指令。模式匹配可以将 LLVM IR 指令或者指令序列映射为专用指令。对于那些很难通过模式匹配完成指令选择的专用指令，可以通过添加内函数扩展 LLVM，将功能单元以函数调用的形式表示，直接映射为专用指令。如添加带饱和的加法运算内函数 __builtin_adds(int a, int b)，实现将功能单元 adds()函数直接映射为 FlexEngine 的 adds 指令。

```
int adds(int var1, int var2)
{
   Long long L_sum;                    __builtin_adds(int a, int b)
   int swOut;                                  ⟶          adds   rD, rB, rA
   L_sum = (Long long) var1 + var2;
   swOut = saturate(L_sum); //饱和处理
   return (swOut);
}
```

图 3-12 内函数实例

表 3-14 FlexEngine 的内函数

内函数	描述
__builtin_adds(int a, int b)	带饱和处理的加法运算
__builtin_subs(int a, int b)	带饱和处理的减法运算
__builtin_ max(int a, int b)	最大值运算
__builtin_min(int a, int b)	最小值运算
__builtin_sabs(int a, bool saturation)	绝对值运算（带饱和处理）
……	

3.1.3 编译器环境配置

为了使 FlexEngine 后端描述文件能够在 LLVM 中正常工作，需要在 LLVM 编译系统中做如下修改：

（1）将 FlexEngineAsmPrinter.cpp 放到 lib/Target/FlexEngine/asmprinter 目录下，其余的后端描述文件（.td，.h，.cpp）放到 lib/Target/FlexEngine 目录下。

（2）添加 makefile 系统支持。在 lib/Target/FlexEngine 目录下创建 makefile 文件，该文件包含变量 LEVEL、LIBRARYNAME、TARGET、BUILT_SOURCES 与 DIRS，分别指定了目标的编译层次、库文件名称、目标名称、TableGen 生成文件与该目录下的文件子目录。最后将顶层目录中的 Makefile.common 文件包含进来，使得目标可以被编译。

（3）修改 LLVM 顶层目录中的 configure 脚本文件，将 FlexEngine 目标添加到 TARGETS_TO_BUILD 变量中。

（4）运行配置脚本文件，编译整个 LLVM 项目，使得 LLVM 系统可以编译并链接 FlexEngine 目标。

表 3-15 配置文件列表

./CMakeLists.txt
./configure
./autoconf/configure.ac
./cmake/config-ix.cmake
./cmake/modules/LLVMLibDeps.cmake
./include/llvm/ADT/Triple.h
./lib/Support/Triple.cpp

通过编译 LLVM 系统源代码得到一系列可执行文件，通过以下命令行生成 FlexEngine 的汇编代码：

clang xxx.c -O2 -emit-llvm -S -o xxx.ll
llc xxx.ll -march=flexengine -o xxx.s

3.1.4 本节小结

基于 Clang 与 LLVM 组合为专用指令集处理器 FlexEngine 生成编译器，模块化的 LLVM 结构使得移植工作结构清晰；自定义添加遍能够灵活的针对 FlexEngine 进行编译转换和优化，从而获得高质量的代码；通过添加内函数或者新的结点对 LLVM 进行扩展，可以使得编译器更好的支持专用指令的添加与指令选择。

3.2 二进制工具集设计

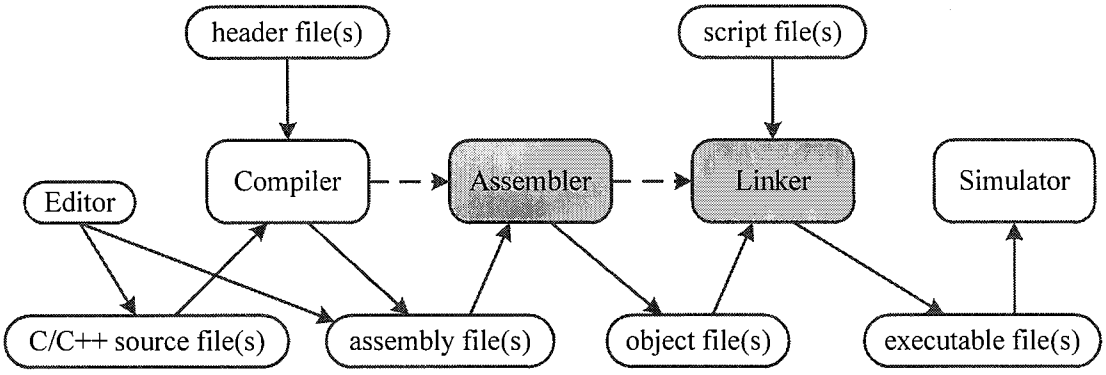


图 3-13 软件开发工具

面向 FlexEngine 处理器移植 GNU binutils，可以在短时间内获得兼容性较好

的二进制工具集。GNU binutils 是 GNU 的二进制工具集，源码公开，遵循 GPL 协议，可移植性强，已经被移植到几十种处理器，采用前后端设计，通过修改已有的目标机或者添加新的目标机即可实现二进制工具的移植。移植 GNU binutils 有很多优点：

（1）重用 GNU binutils 中目标机无关的代码；

（2）代码的重用可以节省工具集的开发时间；

（3）方便熟悉 GCC 编译工具链的开发者。

GNU binutils（Binary Utilities）是一个处理目标代码的软件工具集，同编译器（compiler）和仿真器（simulator）一起用于软件开发，如图 3-13。GNU binutils 主要包含的工具如表 3-16，其中最主要的是汇编器和链接器[18]：

表 3-16 GNU binutils 工具集

Utility	**Description**
assembler (as)	将汇编语言源文件转换为目标文件
linker (ld)	将多个目标文件或者库文件组合为可执行文件
archiver (ar)	创建、修改归档文件，从归档文件中抽取文件
archive indexer(ranlib)	生成索引以加快对归档文件的访问
object file copier (objcopy)	复制二进制文件，或者进行变换
object file stripper (strip)	从文件中删除符号和段
object file symbols (nm)	显示目标文件中的符号
object file viewer (objdump)	显示目标文件的信息
object file size (size)	显示二进制文件中段的信息
object file strings (strings)	显示目标文件中可打印的字符串
address converter (addr2line)	将地址转换成文件名/行号对
c++ filter (c++filt)	将改编的 C++符号还原
ELF file viewing (readelf)	显示 ELF 格式文件内容的信息
ELF file editing (elfedit)	编辑 ELF 格式文件

3.2.1 移植 GNU 二进制工具集

GNU binutils 源代码可以通过 GNU 网站免费获得，其根目录的组织如图 3-14：

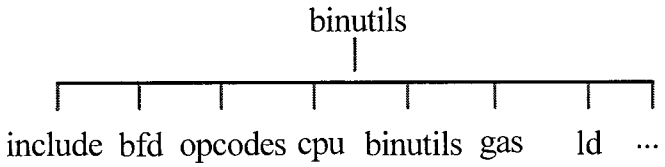


图 3-14 GNU binutils 文件结构

include 包含公共头文件，其中目标机文件定义了目标文件格式信息和 opcode 信息，分别为文件 include/{file-format}/{target}.h 与 include/opcode/{target}.h。

bfd 包含 BFD 文件，支持 ELF，COFF，SREC 等目标文件格式，如果支持新的目标文件格式需要在此添加相应的代码。BFD 使用不同的 bfd_architecture 枚举值表示处理器体系结构，cpu-{target}.c 定义了目标机的体系结构信息。{file-format}-{target}.c 定义了目标机重定位信息，reloc_map 映射 BFD 重定位至目标机重定位，reloc_howto_type 描述了目标机的重定位操作。

opcodes 定义了指令集的汇编与反汇编信息。{target}-opc.c 用于汇编器。{target}-dis.c 用于反汇编，匹配操作码和操作数，并且将指令按照一定格式输出。

cpu 用于 CGEN[19]自动生成目标相关的 opcodes 源文件。

binutils 包含了 objcopy，objdump 和 readelf 等 binutils 工具源文件。文件夹不需要添加任何目标机文件，但需要修改配置信息。

gas 包含了汇编器的源代码。config 目录下的 tc-{target}.c 和 tc-{target}.h 包含了目标机汇编器代码，定义了指令的操作数错误类型，注释符，单行注释符，行内分隔符，目标机的命令行参数解析函数 md_parse_option()，目标机帮助信息 md_show_usage()，汇编器指示 md_pseudo_table，汇编器初始化函数 md_begin()，汇编单条指令解析函数 md_assemble()，指令的操作数解析函数 parse_operand()等。其中 md_assemble()是汇编器移植的关键。

ld 包含了链接器的源代码。{target}.sc 定义了链接器默认的链接脚本。{target}.sh 定义参数用来修改默认的链接脚本。

gprof 包含了 GNU profiler 的源代码，其代码与目标机无关。另外几个文件夹，其中的文件用于 binutils 的编译过程，并不是 binutils 的源文件，如 intl 文件夹包含 GNU gettext 库，Libiberty 是 GNU 项目的标准函数库。

GNU binutils 中最主要的是汇编器和链接器。移植工作也主要是汇编器、链接器和反汇编（objdump 的一部分）的移植。由于 GNU binutils 工具通过 BFD 来操作目标文件，而且 BFD 规范不能以基本接口表示所有的目标文件格式的全部信息，部分目标文件的特殊信息还需要通过添加一些 hook 到 BFD，因此移植工作还包括 BFD 的移植。

移植 GNU binutils 可以通过修改一个相似的 target 文件，或者添加 target 相关的文件[20]。方法则可以通过复制与目标机结构相似的已有目标机文件，或者通过 CGEN(Cpu tools GENerator)辅助生成 GNU binutils 的目标机文件[21]。CGEN 是一个独立的项目，提供框架和工具集辅助生成 cpu 开发工具，如汇编器、反汇编器和仿真器等，通过 cpu 描述文件，可以为 GNU binutils 生成目标机文件，如表 3-17，这些文件的内容也正是移植 GNU binutils 的关键所在：

表 3-17 CGEN 生成的目标机文件

BFD	
bfd/{file-format}-{target}.c	目标机重定位信息
bfd/cpu-{target}.c	目标机体系结构
Opcodes	
include/opcodes/{target}.h	目标机指令集编码公共头文件
opcodes/{target}-dis.c	反汇编程序
opcodes/{target}-opc.c	指令集编码
GAS	
gas/config/tc-{target}.h	汇编程序公共头文件
gas/config/tc-{target}.c	汇编程序
include/{file-format}/{target}.h	汇编程序公共头文件
LD	
ld/scripttempl/{target}.sc	默认的链接脚本
ld/emulparams/{target}.sh	链接脚本自定义参数

在整个的 binutils 移植过程中，用户可以利用已定义的宏和接口函数，或者定义新的宏和函数，实现所需要的功能。然后修改配置信息文件，如表 3-18，添加目标机的相关信息，完成目标机的注册和文件依赖关系，便可以在 linux 平台或者 windows 的 linux 开发环境(如 cygwin、mingw)下编译安装 GNU binutils，获得目标机的二进制开发工具集。

表 3-18 GNU binutils 配置文件

配置信息文件	
opcodes/configure.in	gas/configure.in
opcodes/disassemble.c	gas/configure.tgt
opcodes/Makefile.am	gas/Makefile.am
bfd/configure.in	ld/configure.in
bfd/config.bfd	ld/makefile.am
bfd/archures.c	include/dis-asm.h
bfd/reloc.c	binutils/readelf.c
bfd/targets.c	binutils/Makefile.am
bfd/Makefile.am	include/elf/common.h

3.2.2 二进制文件描述库

BFD（Binary File Descriptor library）[22]是一个独立的通用工具库，提供操作目标文件的各种函数和接口，为 GNU binutils 等工具提供了一组统一的接口，如图 3-15。BFD 对目标文件进行抽象，使用统一的规范格式管理信息，当 BFD 读取文件时将数据转换为规范格式，当写文件时再将规范格式转换为目标文件格式。BFD 的设计分为前端和后端两部分；前端为应用提供用户接口，定义了管理内存和规范格式数据结构的各种接口，决定使用哪一种后端以及何时使用后端函数。BFD 的每一个后端支持一种目标文件格式，提供各种函数用于规范格式和目标文件格式的转换。BFD 的设计为不同体系结构和不同目标文件格式提供了一种统一的操作接口，用户能够使用相同的函数对多种目标文件格式进行操作，因此 BFD 能够实现不同目标文件格式之间的转换，通过添加 BFD 后端可以使 BFD 支持新的目标文件格式。

nm、objdump、objcopy 等工具使用 BFD 以规范格式读取 object 文件，基本的 BFD 接口就能满足要求。as、objcopy 等工具使用 BFD 创建 object 文件，基本的 BFD 接口也已经能够满足要求；但是如果 target 具有一些特殊的 object 文件格式，则需要添加特殊的 BFD 接口。ld 使用通用的 BFD 结构将多个目标文件或者库连接，生成一个 object 文件。

BFD 以指向 bfd 类型的指针表示文件，其中包含了文件的各种信息，如目标文件格式信息、起始地址、处理器类型信息、段信息和符号表。目前的 BFD 规范不能以基本接口表示所有的目标文件格式的全部信息（information loss），但是

简化了处理目标文件的过程，部分目标文件的特殊信息可以通过添加一些 hook 到 BFD，或者通过汇编器和链接器去处理。

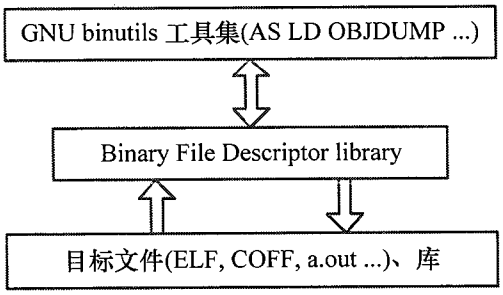


图 3-15 Binary File Descriptor library

BFD 使用 bfd_architecture 枚举值描述不同的 CPU 体系结构，bfd_flexengine_arch 描述了 flexengine 处理器的体系结构。

```
const bfd_arch_info_type bfd_flexengine_arch =
{
    32,                                 /* 32 bits in a word. */
    32,                                 /* 32 bits in an address. */
    8,                                  /* 8 bits in a byte. */
    bfd_arch_flexengine,                /* bfd_architecture */
    0,                                  /* Only 1 machine. */
    "flexengine",                       /* architecture name; */
    "flexengine",                       /* printable name; */
    4,                                  /* Section alignment power. */
    TRUE,                               /* The one and only. */
    bfd_default_compatible,
    bfd_default_scan,
    0                                   /* next pointer is null. */
};
```

BFD 访问目标文件和库文件，通常能够执行大部分的链接操作，如处理符号表、参考符号、重定位 relocation 和创建最终的输出文件。reloc_howto_struct 结构描述了符号的重定位信息，重定位类型指向不同的目标机重定位，flexengine_elf_howto_table 描述了 flexengine 的重定位信息。

```
static reloc_howto_type flexengine_elf_howto_table[] = {
    ...
    HOWTO(FLEXENGINE_LO_16,
            2,                          /* rightshift */
            2,                          /* size = long */
            16,                         /* bitsize */
```

```
        FALSE,                          /* pc_relative */
        6,                              /* bitposition */
        complain_overflow_unsigned,
        bfd_elf_generic_reloc,
        "FLEXENGINE_LO_16",
        FALSE,
        0,
        0x003FFFC0,
        FALSE),
    ...
}
```

3.2.3 汇编器

汇编器将汇编语言源文件转换为可链接的目标文件，其中包含汇编后的二进制机器码，链接器创建可执行文件需要的链接信息，以及仿真器需要的符号信息。GNU as（GAS）[23]是 GNU binutils 中的汇编器，采用一遍扫描处理。首先，汇编器通过调用初始化子函数初始化。对于每个源文件，汇编器调用 read_a_source_file()函数读入文件并解析。对于源文件的每一行，如果是 label，汇编器调用 colon()函数处理；如果是汇编器指示，则通过查找指示哈希表 po_hash，执行相应的指示处理函数。指令则通过 target 相关的子函数 md_assemble( )转换为二进制编码。汇编器在解析源文件时创建 frag 存储数据，当指示或者指令产生二进制编码时，汇编器会在生成的 frag 中，通过 frag_more( )分配空间并将其保存在 frag 中。若编码中存在不能解析的地址信息（如外部全局变量或者后文中的程序 label），则通过 fix_new( )或者 fix_new_exp( )生成 fixup。当输入文件处理完时，write_object_file()给所有的 frag 分配地址，解析 fixup，仍不能解析的 fixup 将保存为目标文件的 relocation，然后输出可链接的目标文件。

md_assemble()函数将单条指令汇编为二进制编码，是汇编器设计的核心，如图 3-16。FlexEngine 采用了一个指令 hash 表，每一项主要包含 3 个 field，分别为指令助记符、指令的操作数和指令编码，在汇编器初始化时候根据 flexengine 的指令编码表 flexengine_opcodes 构建，如{"ADD", "[ccC], rD, rA, rB", "10101000 0100DDDD DAAAAABB BBBCCCCC"}，md_assemble( )首先解析指令的助记符，如发现存在于指令 hash 表，则解析操作数，得到操作数回填的编码。

flexengine 的指令编码表 flexengine_opcodes：

```
const  struct  flexengine_opcode  flexengine_opcodes[]  =
{
    ...
    {"add",  "[ccC],rD,rA,rB", "10101000 0100DDDD DAAAAABB BBBCCCCC"},
    {"and",  "[ccC],rD,rA,rB", "10100000 0000DDDD DAAAAABB BBBCCCCC "},
    ...,
    { "", "", ""}   // The end of the table
};
```

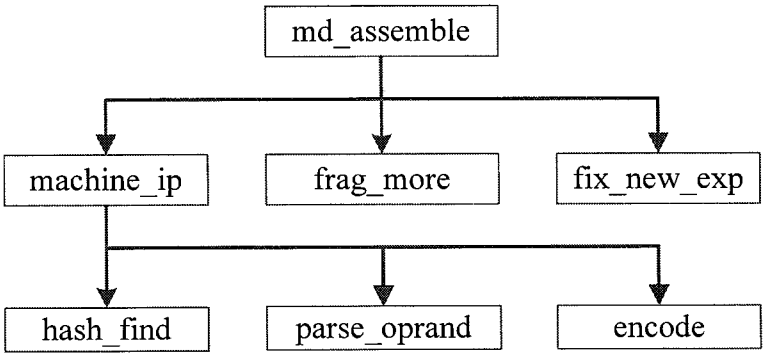


图 3-16 函数 md_assemble()

3.2.4 链接器

链接器 ld 组合若干目标文件或者库文件，为数据和程序分配存储空间，处理符号表、参考符号和重定位 relocation，生成可执行文件[24]，如图 3-17。链接器通过链接脚本和 BFD 完成整个链接过程。链接脚本规定了输出文件的存储器映射，描述如何将输入文件内的各个段放入输出文件，控制输出文件内各部分在存储器空间的布局。BFD 访问目标文件和库文件，通常能够执行大部分的链接操作，如处理符号表、参考符号、重定位 relocation 和创建最终的输出文件。

BFD 打开目标文件，检测输入目标文件的格式，创建 BFD 描述符 descriptor，其中包含了处理目标文件数据的各种子程序，然后根据连接器脚本的内容将所有的输入段映射到输出段。如果一些段不能被映射，则调用函数 ldemul_place_orphan，计算应其放置的位置。lang_process 计算所有符号的值，然后任意未定义的 symbols 都会导致连接错错误。Ldwrite 调用 final_link，处理

所有的 relocation，输出至文件。

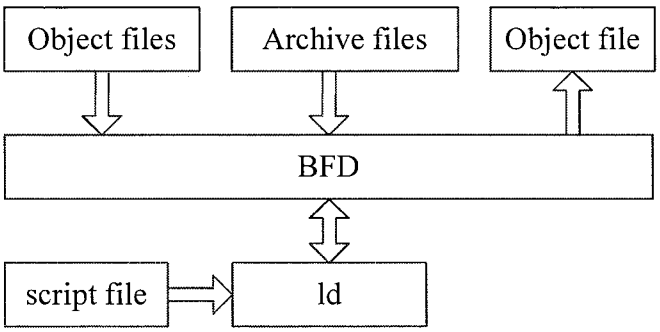


图 3-17 linker

3.2.5 反汇编工具

objdump 可以将二进制编码反汇编为汇编语言，便于用户调试程序。Opcode 库提供了实现这种映射的 BFD 接口函数。反汇编的实现一般由{target}-dis.c 和{target}-opc.c 组成。{target}-opc.c 描述了指令集的编码方式，同时包含一些处理函数。{target}-dis.c 实现了反汇编 disassemble 的接口函数，如图 3-18。反汇编器通过函数 bfd_get_arch()得到 CPU 体系类型，调用不同的反汇编函数，print_insn()从缓冲区读入二进制编码，遍历 flexengine 的指令编码表 flexengine_opcodes，得到匹配项，从中解析出指令助记符、操作数，并将这些信息按照一定格式打印。

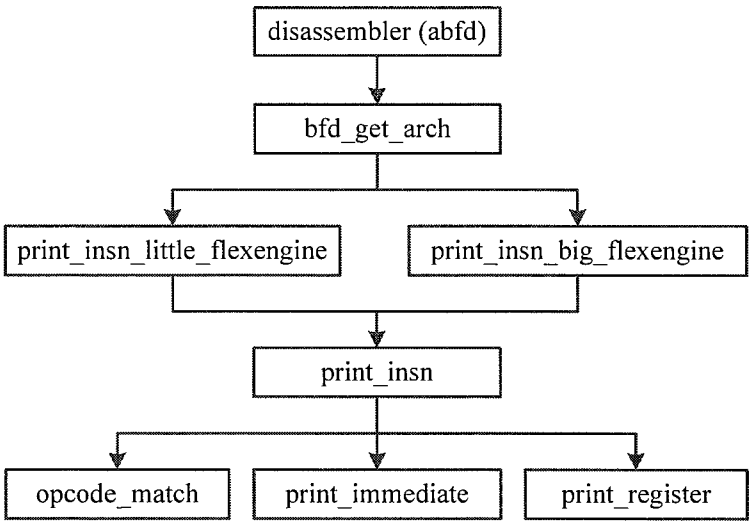


图 3-18 函数 disassemble()

3.2.6 本节小结

移植 GNU binutils 可以在短时间内获得支持 FlexEngine 的全部 GNU binutils 工具集，包括 as、ld、objdump、objcopy、ar 等，兼容性好，是为处理器开发高质量二进制工具集的首选方法。同时，基于指令哈希表的设计，易于设计的重定位与扩展。

3.3 仿真器设计

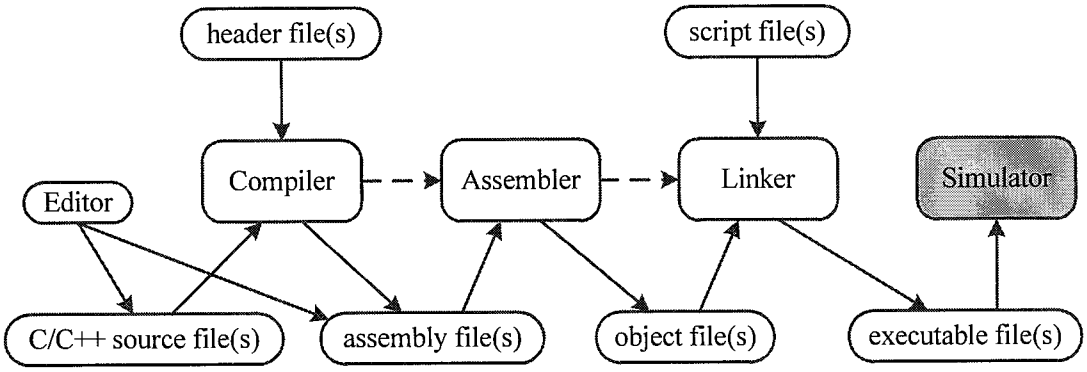


图 3-19 软件开发工具

指令集仿真器是最基本的软件仿真工具，解释型指令集仿真器通过在内存中建立数据结构来模拟目标处理器的状态，然后对每一条指令进行译码、执行。如图 3-20 所示。取指单元从程序存储器中取出一条指令；译码单元通过相应的掩码取出指令中的操作码和操作数，然后根据操作码选择对应的执行函数；执行单元通过执行函数对操作数进行运算，从而完成指令仿真。

解释型指令集仿真器能够精确地模拟指令执行过程中各个部件的状态，能有效地对存储器、流水线及运算单元进行模拟；具有良好的扩展性，可以方便地实现程序调试，对程序仿真过程进行数据统计，用于硬件设计评估，软件开发、调试以及编译器测试。

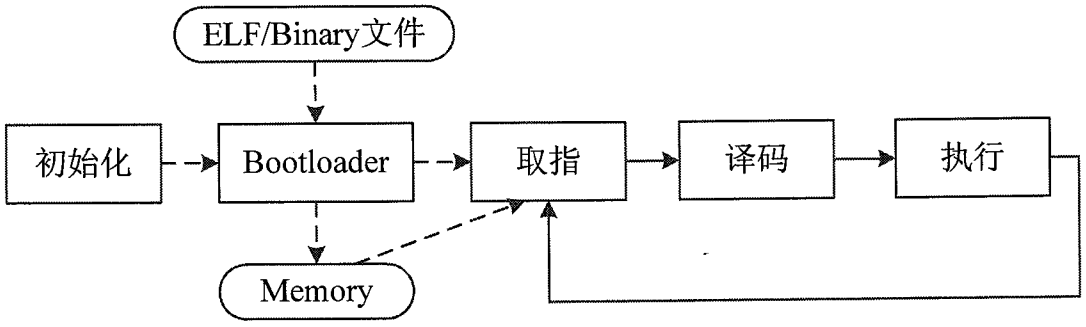


图 3-20 解释型仿真器

本文的仿真器主要分为 5 个模块：初始化模块、加载模块、取指模块、译码模块和执行模块。

3.3.1 初始化模块

初始化模块在仿真器启动时，对各模块进行初始化，主要完成以下功能：

（1）初始化寄存器文件和存储器。

（2）初始化译码模块，建立指令索引列表，以及指令操作数列表、指令掩码。

（3）初始化执行模块，如处理器的状态寄存器等。

3.3.2 加载模块

初始化模块运行结束后，加载模块解析 ELF 可执行文件，将解析出的数据和指令分别存储到指定地址的数据存储器和程序存储器。

3.3.3 译码模块

译码模块提取指令的操作码和操作数，由操作码确定相应的指令索引号，并将指令序号和操作数保存在操作数结构体中。

1）将指令和指令类型掩码依序进行掩码操作，判断指令类别，然后根据指令的类别，通过指令特征码完成指令识别，获得指令索引号。

2）根据指令的操作数列表，以及在指令编码中的位置信息，从指令中提取出操作数，并将操作数按序保存在定义的操作数向量 OperandList 中。

如算术操作指令 add，其译码过程如下：

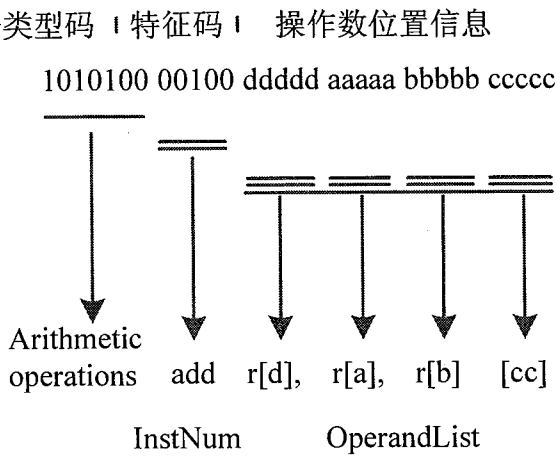


图 3-21 译码单元

定义如下的结构体：

```
struct Operands
{
    int InstructionIndex;                    //指令序号
    vector<unsigned int>OperandList;         //操作数列表
};
```

该结构体包括指令索引和操作数列表信息。其中操作数列表中包含了从指令中提取的操作数信息。

在执行模块定义了每条指令的执行函数，完成对于操作数的运算。通过定义函数指针数组，译码模块由指令索引选择指令的执行函数。该函数指针数组如下：

```
Int(* ExecuteFunction [InstNum]) (struct Operands &);
```

其中* ExecuteFunction 为定义的执行函数指针数组，InstNum 为指令索引。

3.3.4 执行模块

执行模块是一个执行函数集合，通过执行函数实现指令，根据指令定义对相应的数据结构进行访问或修改。这些函数均以译码模块传递的指令操作数向量作为参数。

```
int instruction0(struct Operands operand &);

int instruction1(struct Operands operand &);

… …
```

3.3.5 本节小结

解释型指令集仿真器能够精确地模拟指令执行过程中各个部件的状态，能有效地对存储器、流水线及运算单元进行模拟；具有良好的扩展性，在仿真过程中可以设置与添加断点，方便地实现程序调试，对程序仿真过程进行数据统计，如程序的总执行周期、函数调用次数与执行周期、单条指令的执行次数，统计结果可以用于设计评估。

3.4 本章小结

本章主要介绍了处理器的软件开发工具实现：基于 LLVM 与 Clang 的组合设计编译器，模块化、结构清晰、可移植性强、易于生成代码优化；移植 GNU binutils，可以获得 FlexEngine 的全部 GNU binutils 工具集，设计周期短、兼容性好，同时基于哈希表的设计易于扩展；解释型指令集仿真器具有良好的扩展性，可以方便地实现程序调试，对程序仿真过程进行数据统计，用于硬件设计评估，软件开发、调试以及编译器测试。软件开发工具的可扩展性，能够很好的支持专用指令集处理器对于指令集扩展的需求。

软件开发工具如编译器、汇编器、链接器和仿真器都是 FlexEngine 面向数字助听器 ASIP 设计过程中必须的。在 ASIP 设计过程中，软件开发工具能够用来评估设计以及提供设计反馈，验证设计是否满足设计约束；有效地提取应用的特点，设计优化的硬件实现。在 ASIP 设计结束时，软件开发工具能够用于应用的开发与调试。针对 ASIP 精心设计的编译器能够发挥专用指令集的优势，有效的将程序映射到专用指令和加速单元，充分利用硬件资源从而有效的减少功耗；同时，编译器使用多种编译优化策略，产生高效的目标代码，减少运行时间和功耗。仅当 ASIP 的软件和硬件联合优化，才可以获得最优的低功耗 ASIP 设计。

ASIP 设计过程中，快速生成软件开发工具是很重要的，本文对软件开发工

具设计进行了探索，也是研究其自动生成机制的基础。

参考文献

[1]Aho A V, Lam M S, Sethi R, et al. Compilers: principles, techniques, and tools[M]. Pearson/Addison Wesley, 2007.

[2]Liem C, Breant F, Jadhav S, et al. Embedded tools for a configurable and customizable DSP architecture[J]. Design & Test of Computers, IEEE, 2002, 19(6): 27-35.

[3]Aho A V, Lam M S, Sethi R, et al. Compilers: principles, techniques, & tools[M]. Pearson/Addison Wesley, 2007.

[4]Team G C C. Gnu compiler collection homepage[J]. 2006.

[5]Developers O. Open64 compiler and tools[J]. 2001.

[6]Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation[C]//Code Generation and Optimization, 2004. CGO 2004. International Symposium on. IEEE, 2004: 75-86.

[7]Chang P P, Mahlke S A, Chen W Y, et al. IMPACT: an architectural framework for multiple-instruction-issue processors[M]. ACM, 1991.

[8]Hanson D R, Todd A P. A retargetable C compiler[C]//Design and Implementation. Benjamin/Cummings Publishing. 1995.

[9]SUIF Group. Suif compiler system[J]. URL: http://suif. stanford. edu, 1999.

[10]Chakrapani L, Gyllenhaal J, Hwu W, et al. Trimaran: An infrastructure for research in instruction-level parallelism[J]. Languages and Compilers for High Performance Computing, 2005: 922-922.

[11]Appel A, Davidson J, Ramsey N. The Zephyr compiler infrastructure[J]. Distributed at Supercomputing, 1998, 98.irginia.edu/zephyr

[12]Ju R, Chan S, Wu C. Open research compiler for the Itanium family[C]//Tutorial at the 34th Annual International Symposium on Microarchitecture. 2001.

[13]Lattner C. LLVM and Clang: Next generation compiler technology[C]//The BSD Conference, Ottawa, Canada. 2008.

[14]Lattner C, Adve V. LLVM language reference manual[J]. 2006.

[15]Louden K C. Compiler construction[M]. PWS Publishing Company, 1997.

[16]TableGen Fundamentals. http://llvm.org/docs/TableGenFundamentals.html

[17]CMake, http://www.cmake.org/cmake/help/documentation.html

[18]Visscher P. GNU Binary Utilities[J]. documentation for the GNU binary utilities,

collectively version, 2(1): 61.

[19]The Cpu tools GENerator, CGEN [M]. Red Hat, 2009

[20]Free Software Foundation. Binutils Porting Guide To A New Target Architecture [EB/OL]. http://sourceware.org/binutils/binutils-porting-guide.txt, 2009-12-1

[21]Porting [EB/OL]. http://sourceware.org/cgen/docs/cgen_5.html, 2010-1-28

[22]Taylor I L. BFD Internals[J]. Free Software Foundation Inc.

[23]Elsner D, Fenlason J. Using as[J]. Assembler Internals Section, Free Software Foundation, 2001.

[24]Bothner P, Chamberlain S, Taylor I L.GNU Linker Internals [EB/OL]. http://sunsite.ualberta.ca/Documentation/Gnu/binutils-2.9.1/html_mono/ldint.html

第4章 低功耗数字助听器 ASIP 设计

在明确了 ASIP 设计的应用与设计约束，并针对应用程序特点选择了合适的基本指令集处理器后，借助处理器的软件开发工具对应用程序进行仿真，分析程序的行为，进而优化指令集与处理器架构，是 ASIP 设计的关键之处。本章主要介绍了 ASIP 的指令集迭代优化过程、软件开发工具的专用指令扩展、以及设计中采用的低功耗技术。

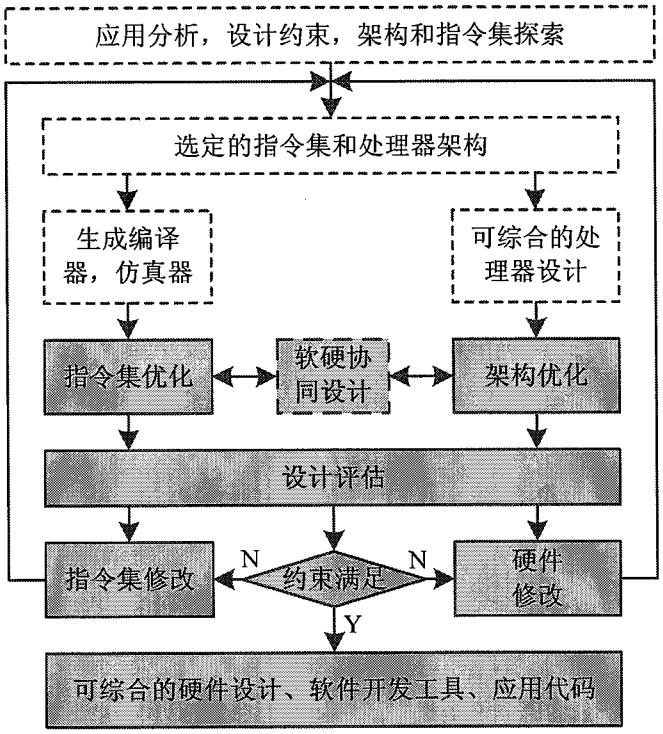


图 4-1 软硬协同的 ASIP 设计流程

4.1 功耗分析

当基本指令集处理器满足所有的时序要求后，功耗优化可以有效的降低处理器的功耗。对于给定的算法，处理器的能耗可以划分为三部分[1]：

（1）固有能耗（intrinsic energy）：完成算法全部计算任务所需要消耗的最小能量，仅仅取决于算法的操作量与实现工艺。

（2）路由能耗（routing energy）：功能单元间数据交互所消耗的能量。

（3）无效能耗（overhead energy）：由无效的逻辑翻转、时钟树分布等产生

的能耗浪费。

性能优化的同时也会带来功耗的优化，如专用指令的添加通常不会减少计算量，但是缩短了程序的执行时间，因此减少了无效能耗。如果结构的改变对数据的交互没有太大影响，路由能耗的减少通常不明显。如果算法保持不变，固有能耗也通常保持不变。对于给定的计算任务进行功耗优化，从算法行为实现到标准单元网表，不同级别上降低功耗的比例是不同的，级别越高，可降低的功耗也越多，如图 4-2。

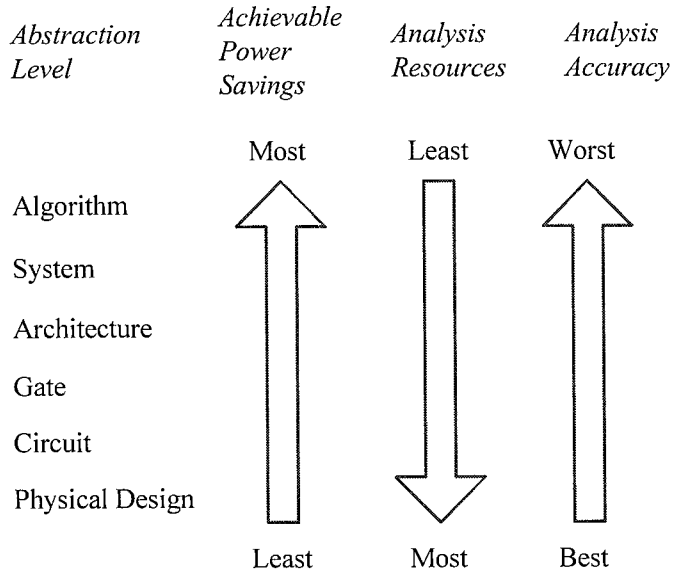


图 4-2 设计的不同抽象级优化对功耗的影响

基于标准单元的 ASIP 实现，主要是在设计的 ASIP 软件级、系统级、寄存器传输级与逻辑级采用低功耗的设计技术。在设计的各阶段综合运用以下技术降低功耗：

（1）ASIP 软件级（ASIP Software Level）

- 应用算法优化；
- 汇编代码级优化。

（2）系统级（Architectural System Level）

- 根据算法需要确定程序存储空间和数据存储空间，较小的存储空间可以有效的减少存储器功耗。ASIP 片上集成了三块双端口 SRAM，分别是 1K×32bits 的程序存储器，和 2 块 2K×24bits 的数据存储器；
- 存储器功耗优化，包括存储器分割、循环缓存。

（3）寄存器传输级（RT Level）

- 专用指令与加速单元，提高程序执行效率；
- 处理器睡眠模式减少空闲状态下的动态功耗；
- 操作数隔离，减少数据通路的无效翻转；

（4）逻辑级（Logic Level）

- 门控时钟

4.2 指令集扩展

通过指令集扩展优化指令集，是专用指令集处理器设计最关键的部分，因此首先介绍面向数字助听器应用的自定制指令扩展。自定制指令通常可以针对两种情况：频繁使用的指令或者指令序列，以专用指令添加到指令集；虽不是频繁使用的操作，但是硬件实现简单，非常容易添加到数据通路[2]。频繁使用的指令可以通过仿真仔细的分析运行时汇编程序获得。

4.2.1 位反寻址模式

通过软件开发工具编译源程序，得到可执行代码，进行仿真，对不同的功能模块进行执行周期统计，如表 4-1。

表 4-1 程序执行周期统计

功能模块	优化前运行周期	%
IN	224	0.9
OUT	256	1.0
WOLA AFB	1344	5.2
FFT	4173	16.2
NOISE REDUCE	9579	37.1
WDRC	2157	8.4
WOLA SFB	1728	6.7
IFFT	4173	16.2
BitReverse	2176	8.4
TOTAL	25810	100

对程序进行分析，BitReverse 仅仅实现地址的位反操作，用于对 FFT 与 IFFT 的输入数据位反存储，但是其占据了程序执行时间的 8.4%。

对于 FFT 的输入数据位反存储，则经过 FFT 处理得到正序的数据结果。以

8 点基 2 时间抽取 FFT 为例，输入数据位反存储得到位反序列，如表 4-2；对位反序列数据进行 FFT 变换，得到顺序结果输出，如图 4-3。

表 4-2 位反寻址存储获得的位反序列

Sequential order		Bit-reverse order	
输入数据	binary	binary	位反存储序列
$x(0)$	000	000	$x(0)$
$x(1)$	001	100	$x(4)$
$x(2)$	010	010	$x(2)$
$x(3)$	011	110	$x(6)$
$x(4)$	100	001	$x(1)$
$x(5)$	101	101	$x(5)$
$x(6)$	110	011	$x(3)$
$x(7)$	111	111	$x(7)$

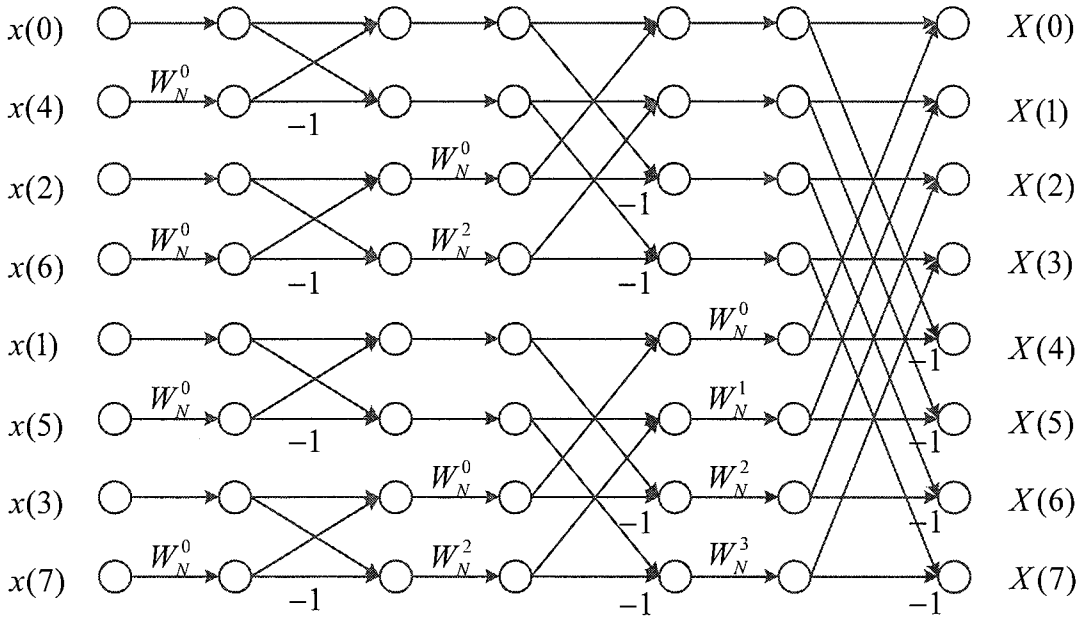


图 4-3 8 点基 2 时间抽取 FFT 信号流图

同理，对于 IFFT 的输入数据位反存储，则经过 IFFT 处理得到正序的数据结果。

4.2.1.1 硬件实现

下面给出了 C 位反程序，实现对地址 addr 进行 num 位的位反转：

```
unsigned short bit_rev(unsigned short addr,unsigned short num) {
    unsigned short addr_temp=0;
    unsigned short i=0;
    addr = addr<<(16-num);
```

```
    for (i=0;i<num;i++) {
        if (addr&0x8000) {
            addr_temp = addr_temp | (1<<i);
        }
        addr = addr<<1;
    }
    return addr_temp;
}
```

可见实现位反寻址是很麻烦的，需要多次移位操作。音频处理算法中通常需要时频变换，FFT 与反变换最多使用。在本文的 32 通道的数字助听器系统中，需要实现 64 点 FFT 与反变化。对于嵌入式处理器如果不支持位操作，通常很难实现位反寻址，这对于低功耗的便携式语音设备是不能忍受的。由于 FlexEngine 不支持位反寻址操作，因此在本文的 ASIP 设计中，通过在地址寻址单元添加位反寻址模式实现音频处理中常见的位反寻址。

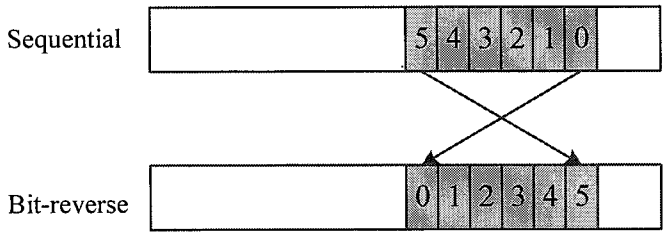


图 4-4 位反寻址

地址产生单元的具体结构，如图 4-5 所示，设置专门的位反寄存器，指示位反寻址的位数，实现位反转。

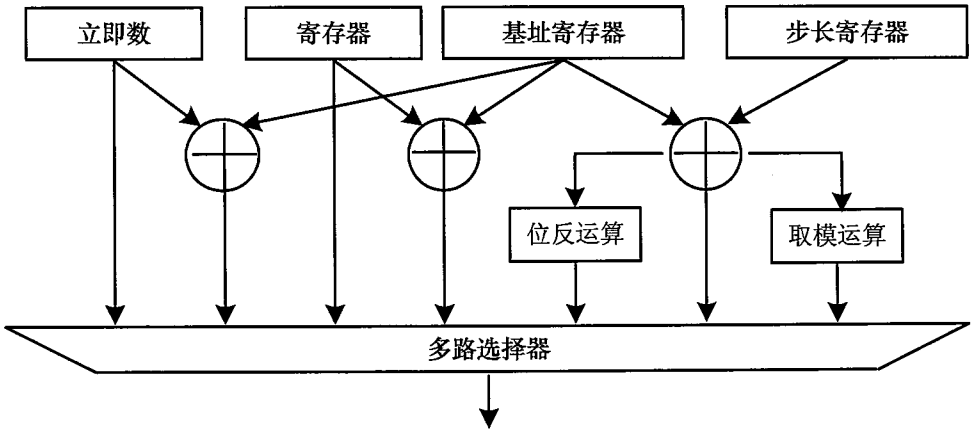


图 4-5 地址产生单元的结构框图

4.2.1.2 软件开发工具设计

首先，添加内函数将位反寻址模式作为 LLVM 系统的语言扩展：

int __builtin_reverse (int)

在指令集 TD 描述文件中，实现位反寻址模式，描述如下：

def addrmode8:Operand<i32>,ComplexPattern<i32,2,"SelectAddrMode8", []>;

*bool SelectAddrMode8(SDNode *Op, SDValue N, SDValue &Base, SDValue &Offset);*

def LD8 : INSTLD<(outs GPR:$dst), (ins addrmode8:$addr), "revload ", "\t$dst, ($addr)", [(set GPR:$dst, (load addrmode8:$addr))]>;

函数 *SelectAddrMode8*，识别位反寻址模式，给出基址和偏移寄存器，需要在文件“FlexEngineISelDATToDAG.cpp”中实现。最终位反寻址通过模式匹配映射到位反寻址存取指令。

位反寻址模式可以显著加速 FFT、DCT 等蝶形变换中的地址计算，添加指令前后对程序进行仿真，结果如表 4-3。

表 4-3 程序执行周期统计

功能模块	优化后运行周期	优化前运行周期
IN	224	224
OUT	256	256
WOLA AFB	1216	1344
FFT	4173	4173
NOISE REDUCE	9579	9579
WDRC	1775	2157
WOLA SFB	1728	1728
IFFT	4173	4173
BitReverse	0	2176
TOTAL	23124	25810
		2686 cycles saved

4.2.2 零开销循环指令

至此，使用 ASIP 实现数字助听器功能，并对仿真过程进行分析，得到程序中各函数的运行周期以及所占总运行时间的比例，如表 4-4。

表 4-4 程序执行周期统计

功能模块	运行周期	%
IN	224	1.0
OUT	256	1.1
WOLA AFB	1216	5.3
FFT	4173	18.0
NOISE REDUCE	9579	41.4
WDRC	1775	7.7
WOLA SFB	1728	7.5
IFFT	4173	18.0
TOTAL	23124	100

表 4-5 指令使用率统计

指令	次数	使用率
add	2867	12.4
cmp	2012	8.7
j	1966	8.5
mul	1757	7.6
lsft	1411	6.1
mov	1341	5.8
hext	1341	5.8
sub	1295	5.6
mla	1109	4.8
wadd	994	4.3
nop	902	3.9
clz	832	3.6
shr	832	3.6
store	624	2.7
wstore	578	2.5
shl	555	2.4
load	555	2.4
lext	532	2.3
or	301	1.3
wload	256	1.1
wsub	231	1.0
...	...	...

由表 4-4 可见 FFT 变换与反变换占据了整个运行时间的 36%，需要大量的乘累加/减、数据存取操作，但其计算密度大、规整，一般考虑添加硬件加速实现。对 FFT/IFFT 以外的程序进行指令使用率统计，如表 4-5。对使用率最高的指令进行分析，发现 add/cmp/j 指令与音频算法中大量使用小循环体的特点相符

合。对于循环体，程序执行过程中通常需要进行循环体执行计数（add）、循环条件判断（cmp）、跳转指令（j）转移至循环体首或循环体外。因此有必要针对音频处理多数时间执行小循环体的特点进行硬件优化。零开销循环指令即是为解决这一问题而添加到 ASIP 中，循环控制单元是循环指令实现的关键部件。

4.2.2.1 硬件设计

循环控制单元设置了 3 个特殊寄存器分别记录着循环体的循环次数、循环体首地址、循环体尾地址。在进入循环体之前，循环指令会对这三个变量进行初始化，记录循环体执行次数、循环体首地址、循环体尾地址。在接下来循环体执行的过程中，循环体首地址、尾地址两个变量都不会发生变化，但是循环次数会随着每次循环体完整的执行一遍而减小。而循环体一次执行结束的标志，也就是程序执行地址与循环体尾地址相等时。此时，除了循环次数减 1 之外，还需要通知程序状态机单元，产生控制信号，控制程序地址产生单元，选择循环体首地址作为接下来的取指地址。这个过程会一直重复，直到循环次数减到 0，表征循环次数已经被执行满足，此时应该跳出循环体，执行后续指令。循环控制单元的结构如图 4-6 所示。

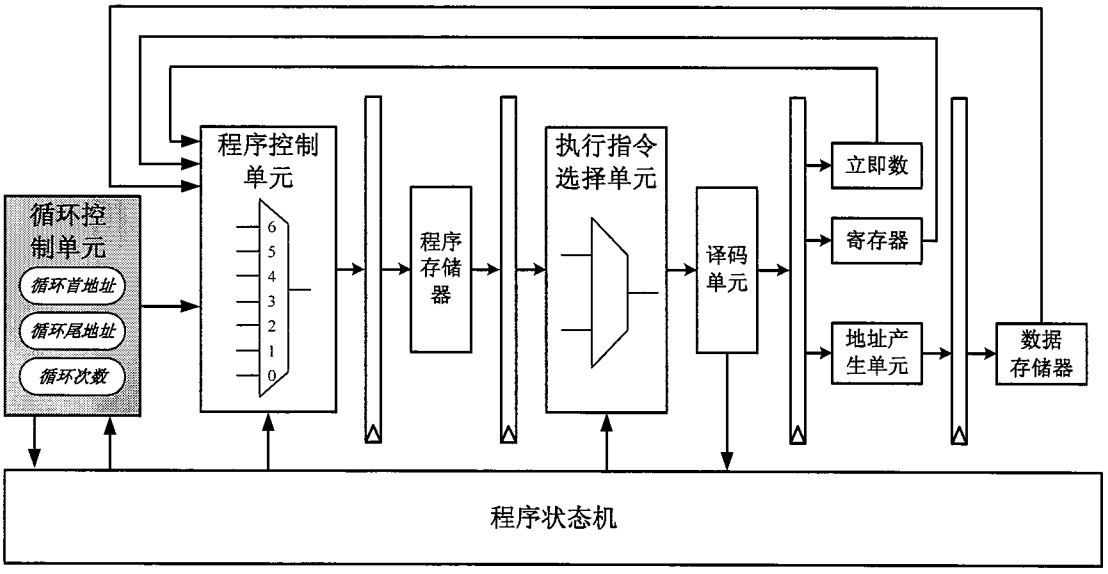


图 4-6 程序控制类指令的执行流程

程序地址	流水线									
loop	FE	DE	OF	IE	WB					
loop+1		FE	DE	OF	IE	WB				
⋮			⋮	⋮	⋮	⋮	⋮			
loop end				FE	DE	OF	IE	WB		
loop					FE	DE	OF	IE	WB	
loop+1						FE	DE	OF	IE	WB

图 4-7 循环指令流水线

4.2.2.2 软件开发工具实现

每个程序都会花很多时间执行循环，因此编译器为循环生成优良的代码就很重要。因此在 ASIP 的编译器设计时，添加循环体优化遍（FlexEngineLoopOptPass），以函数为分析单位，分析中间表示中的循环信息，并发射 loop 指令。

PM.add(createFlexEngineLoopOptPass());

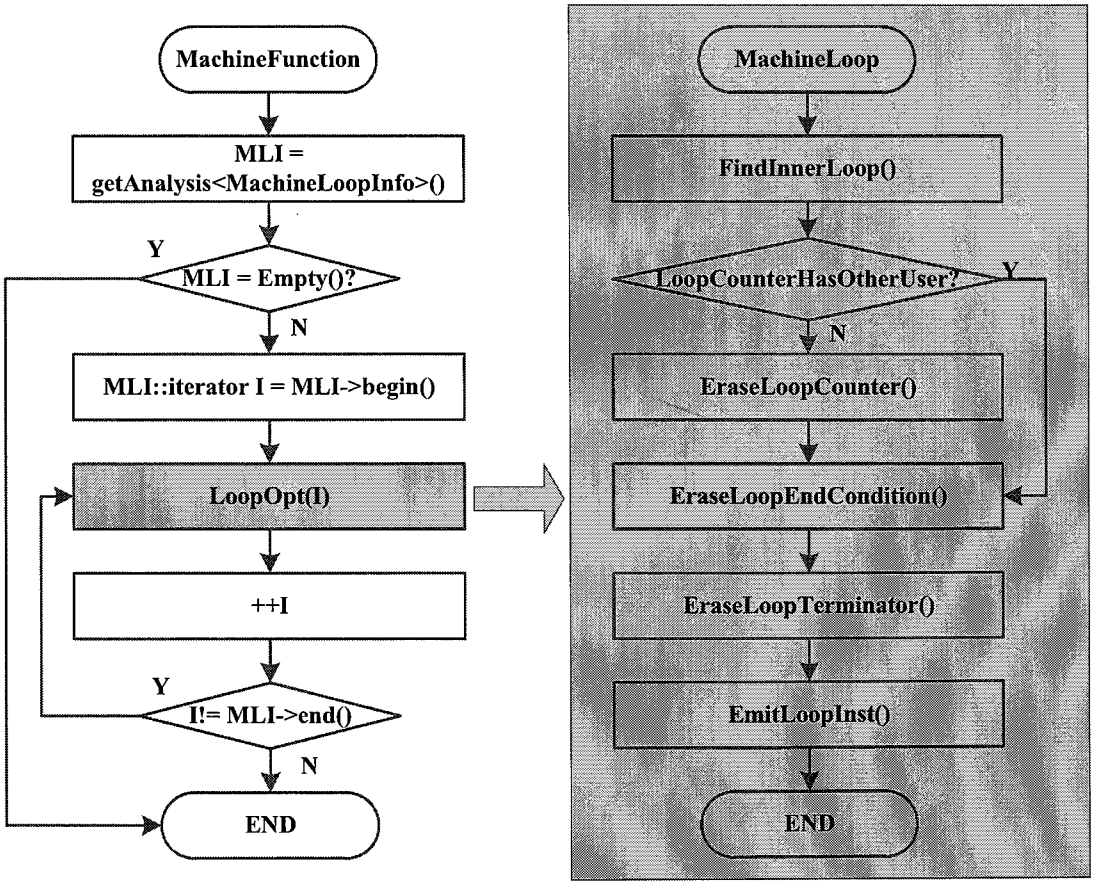


图 4-8 FlexEngineLoopOptPass 流程图

图 4-8 显示了 FlexEngineLoopOptPass 的执行过程，首先利用循环分析方法

getAnalysis<MachineLoopInfo>()，得到函数内的所有循环体；然后对每个循环体进行循环优化 LoopOpt(I)；循环优化的过程即是得到循环体的最内层循环，以 loop 指令替换循环计数指令、终止条件判断指令、与循环跳转指令。

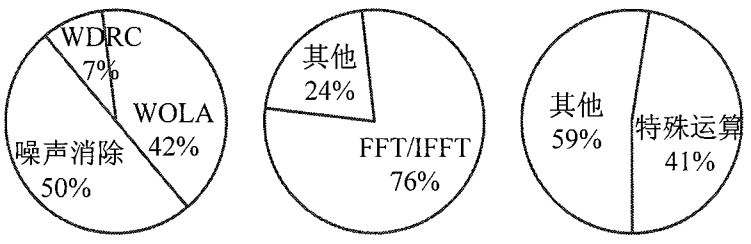
零开销循环适合音频处理算法大部分时间运行小循环体的特点，可以有效提高短循环体的执行效率，大大降低语音处理算法的执行周期。添加 loop 指令后，再次利用 ASIP 的仿真器对数字助听器算法进行仿真分析，对程序仿真过程进行分析，得到优化前后各函数的运行周期，如表 4-6。

表 4-6 程序执行周期统计

功能模块	优化后运行周期	优化前运行周期
IN	32	224
OUT	64	256
WOLA AFB	832	1216
FFT	3021	4173
NOISE_REDUCE	9459	9579
WDRC	1397	1775
WOLA SFB	1344	1728
IFFT	3021	4173
TOTAL	19170	23124
		3954 cycles saved

4.2.3 特殊运算

对表 4-6 中优化后的数据进行进一步分析，得到结果如图 4-9，WOLA（WOLA AFB/FFT/WOLA SFB/IFFT）约占整个程序执行时间的 42%，噪声消除模块约占整个程序执行时间的 50%。对模块单独进行分析，发现 WOLA 滤波器组的快速傅立叶变换（FFT）及其反变换占据了 76%的 WOLA 运行时间；消噪算法中的除法运算以及非线性运算（如开方，对数运算）占据了 41%的噪声消除运行时间（表 4-7）。



Total cycles 18914　　WOLA　　噪声消除

图 4-9 不同模块的周期使用率

表 4-7 噪声消除模块仿真数据

NOISE_REDUCE			
功能模块	运行周期	%	
log	1216	12.9	
recipe	1200	12.7	
sqrt	1472	15.6	
其他	5571	58.9	
	TOTAL	9459	100

对于 FFT/IFFT 变换，其运算规整、计算密度大，暂时不考虑通过专用指令加速其执行。但对于音频处理算法噪声消除（Noise Reduction，NR）、语音增强等涉及到的大量非线性或者特殊运算[3]，如开方 sqrt(x)、除法、对数 log2(x)和倒数 1/x 运算，以库函数实现，虽然计算精度高，但是计算量大，不适合音频处理系统的低功耗要求，考虑添加专门的硬件加速。

4.2.3.1 硬件设计

音频处理算法中几乎所有的非线性运算，都是仅要求对数据进行估计[4][5]。采用函数迭代算法（如牛顿迭代法或者级数展开），其执行时间通常需要多个时钟周期，但是 ASIP 的 ALU 执行周期为 1，MAC 执行周期为 2，如果将其添加到数据通路的执行单元，会破坏已有的流水线结构。因此本文采用更加简单快速的查表法，根据系统的变量范围与精度要求制作表格，适当降低运算精度。特殊运算具体实现如下：

对数计算过程：

$$x = a * 2^b = 2a * 2^{b-1},\ a \in [0.5,1) \tag{4.1}$$

$$\log_2 x = \log_2(2a) + (b-1) \tag{4.2}$$

$$\log_{10} x = \log_2 x / \log_2 10 \tag{4.3}$$

开方计算过程：

$$x = a * 2^b,\ a \in [0.5,1) \tag{4.4}$$

$$x^{1/2} = a^{1/2} * 2^{b/2},\ b\text{为偶数} \tag{4.5}$$

$$x^{1/2} = a^{1/2} * 2^{(b-1)/2} * \sqrt{2},\ b\text{为奇数} \tag{4.6}$$

除法可以通过除数的倒数与被除数相乘获得，倒数的计算过程：

$$y / x = y \cdot x^{-1} \tag{4.7}$$

$$x = a * 2^{b},\ a \in [0.5,1) \tag{4.8}$$

$$x^{-1}=a^{-1}*2^{-b} \tag{4.9}$$

通过查找输入数据 x 的最高非零位确定数值 b，然后向低位连续取 L 位（低位不足补零）作为查找表的地址索引，由查找表获得 $\log_2(2a)$、$a^{1/2}$ 与 a^{-1}，如图 4-10，表格大小由计算精度确定。

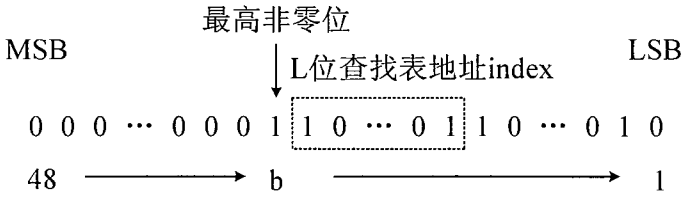


图 4-10 最高非零位与查找表地址

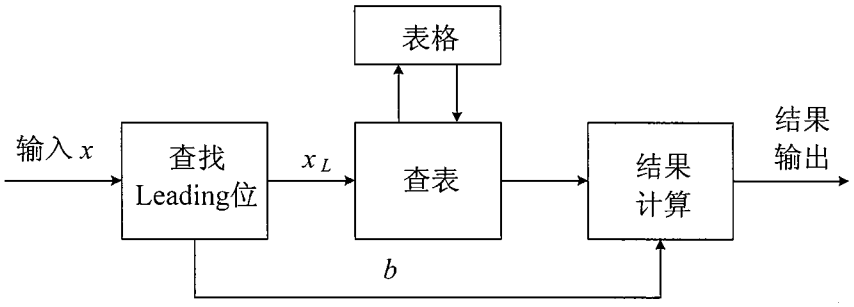


图 4-11 特殊运算的查表法实现

由于查表方法结构简单，因此可以将 log10/sqrt/recipe 作为指令集扩展（ISE）添加到处理器的数据通路 MAC 单元，添加扩展指令集后的处理器数据通路如

图 4-12，扩展运算执行单元分为 P1、P2 两级，在执行单元 P2 级后将运算结果写回。

基于查找表的计算广泛的应用在音频处理中，如数字助听器设计中广泛采用的宽动态压缩（WDRC）算法。wclz 指令返回数据的最高非零位，加速类似 WDRC 算法采用的查找表计算[6]，在硬件设计中复用特殊运算的执行单元 P1 级。

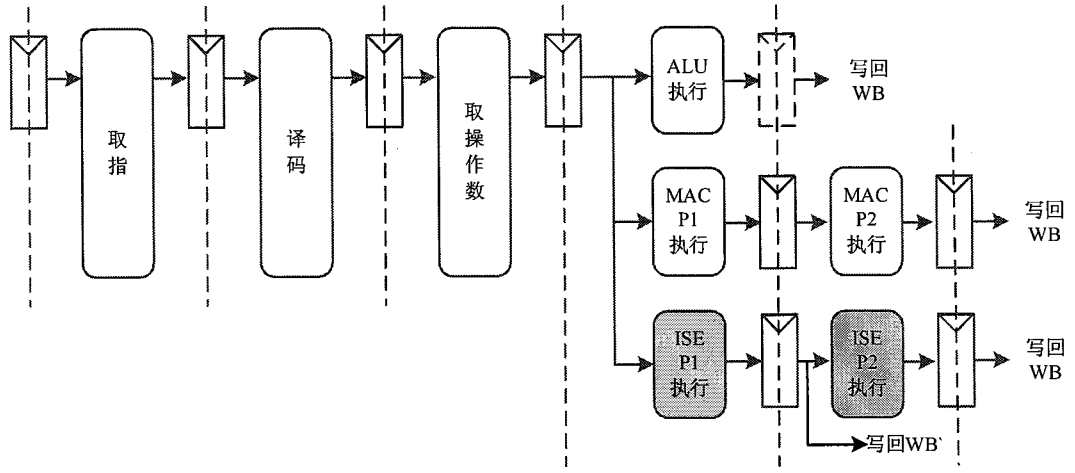


图 4-12 ASIP 的数据通路

4.2.3.2 软件开发工具实现

添加特殊运算指令后，利用 LLVM 内函数（Intrinsic Fuction）[9]，将特殊运算功能单元以函数调用的形式表示，直接映射为专用指令，在不改变 LLVM 语言转换过程的情况下扩展 LLVM 语言。

表 4-8 特殊运算内函数

内函数	LLVM IR	指令
int __builtin_FElog10(long long)	i32 @llvm.FElog10.i64(i64)	log
int __builtin_FEsqrt(long long)	i32 @llvm.FEsqrt.i64(i64)	sqrt
int __builtin_FErecipe(long long)	i32 @llvm.FErecipe.i64(i64)	recipe
int __builtin_FEwclz (long long)	i32 @llvm.FEwclz.i64(i64)	wclz

表 4-9 程序执行周期统计

功能模块	优化后运行周期	优化前运行周期
IN	32	32
OUT	64	64
WOLA AFB	832	832
FFT	3021	3021
NOISE_REDUCE	3400	9459
WDRC	1205	1397
WOLA SFB	1344	1344
IFFT	3021	3021
TOTAL	12919	19170
		6251 cycles saved

添加的特殊运算指令 log10/sqrt/recipe/wclz 提高了音频处理系统的效率，非常适合应用于实时音频处理系统中。对数字助听器算法进行优化后仿真，数据如

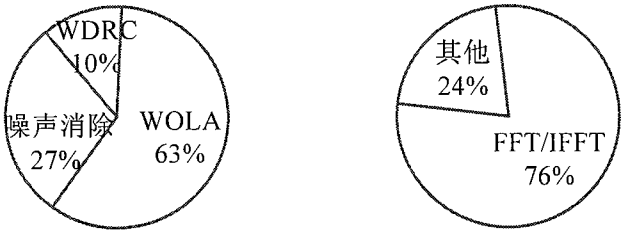
表 4-9。

4.2.4 蝶形运算单元

当系统的计算性能不能满足系统设计需求时，可以考虑添加硬件加速单元，但是硬件加速单元通常不满足指令集的正交性，同时会降低设计的可维护性。因此在添加硬件加速单元时，应该不引起设计变化，以免对设计进行重新验证，同时要保证设计的灵活性。

4.2.4.1 硬件设计

当前的多通道语音处理大多基于等宽通道划分方案，其中用得最为普遍的是WOLA 滤波器组[7]。WOLA 滤波器组中的 FFT 及其反变换对处理器的计算性能要求比较高，且运算密集规整，可考虑硬件优化。以数字助听器为例，由表 4-9 可得整个 WOLA 模块数据统计，如图 4-13，FFT 及其反变换占据了 WOLA 功能模块的 76%的执行时间，占据了整个程序 49%的执行时间。但由于 FFT 结构较复杂，不能以专用指令的形式添加到处理器的数据通路，添加硬件加速单元，则可以在提高系统计算性能的同时降低系统功耗。虽然加速单元会降低系统的灵活性，但是 FFT 及其反变换广泛应用在音频处理的时频变换中，硬件加速 FFT 可以显著提高系统的计算性能，进而提高系统的吞吐量。对于助听器系统而言，则可以使得处理器在一个帧处理期间完成更多更复杂的功能，如反馈抑制、移频压缩、场景识别等，这都得益于处理器强大的计算性能。



Total cycles 12663　　WOLA

图 4-13 WOLA 模块数据统计

蝶形运算（Butterfly）单元是快速傅立叶变换的核心操作，基 2 时间抽取 FFT 及其反变换蝶形单元（如图 4-14）表达式为：

$$\begin{aligned} X_{m+1} &= X_m + W_N^r \bullet Y_m \\ Y_{m+1} &= X_m - W_N^r \bullet Y_m, m = 0,1,.....,M-1 \end{aligned} \tag{4.10}$$

X，Y 是蝶形运算单元的输入，W 为旋转因子，M 为 log_2N，N 为 FFT 点数。通过精心设计的汇编指令每个蝶形运算仍需要 13 条指令，对于 N 点的 FFT，需要计算 $log_2N*N/2$ 次蝶形运算，因此计算非常耗时。将蝶形运算单元以加速单元添加到处理器，如图 4-15，通过指令 butterfly 执行加速单元，输入输出数据地址采用默认地址寄存器，由于蝶形运算的计算包含数据载入、复数乘法、复数加法、结果存储，因此需要 5 个时钟周期，执行 butterfly 指令，需要流水线暂停等待 butterfly 指令执行完成。在进行 FFT 运算时，通过配置 FFT 的点数、蝶形运算单元输入数据、旋转因子以及输出数据的地址与寻址方式即可完成任意点数基 2 时间抽取的快速傅立叶变换。FFT 蝶形加速单元可广泛的应用于音频应用的 FFT 滤波器组的设计中，从而提高系统效率。

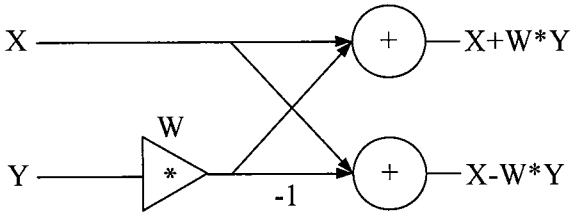


图 4-14 基 2 时间抽取的 FFT 蝶形运算单元

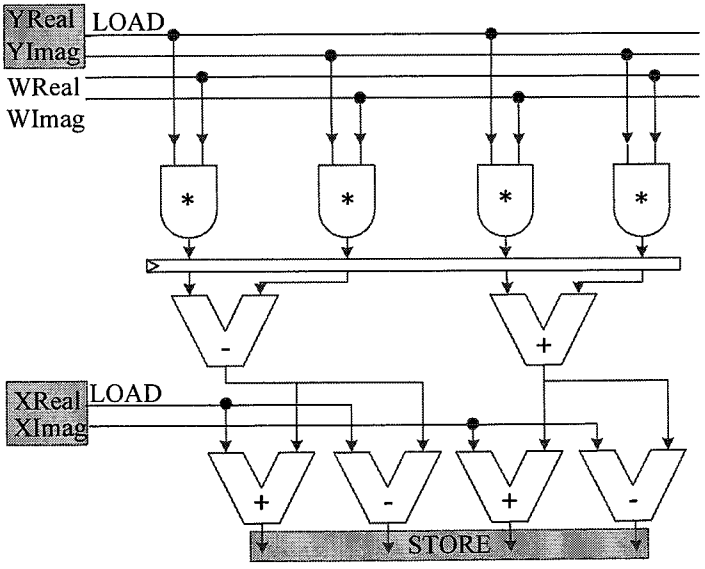


图 4-15 基 2 时间抽取的 FFT 蝶形加速单元

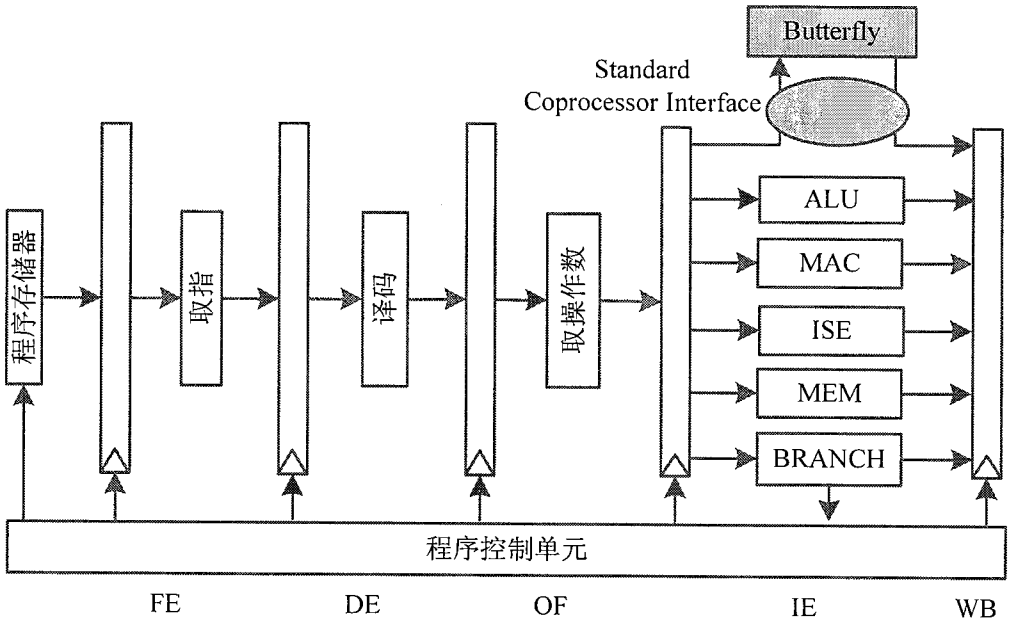


图 4-16 硬件加速单元流水

4.2.4.2 软件开发工具实现

由于 butterfly 指令特殊，通过编译器将程序映射到指令比较困难，所以一般通过内联汇编的形式使用指令。对于 DSP 算法常用的密集运算单元，以库函数（library function）实现。由于库函数实现充分的利用了处理器的指令集，所以在程序设计中通过调用这些库函数即可高效的完成操作。如 FFT/IFFT 变换，通过设置 FFT/IFFT 的点数、旋转因子地址、输入输出数据地址，调用库函数 *fft()* 即可实现：

*int fft(int N, COMPLEX *X, COMPLEX *Y, COMPLEX * W);*

其中 N 代表变换点数，X、Y、W 分别为输入、输出和旋转因子的地址。

表 4-10 WOLA 模块优化结果

功能模块	优化后运行周期	优化前运行周期
IN	32	32
OUT	64	64
WOLA AFB	832	832
FFT	1485	3021
NOISE_REDUCE	3400	3400
WDRC	1205	1205
WOLA SFB	1344	1344
IFFT	1485	3021
TOTAL	9847	12919
		3072 cycles saved

WOLA 滤波器组中的快速傅立叶变换及其反变换对处理器的计算性能要求比较高，添加 butterfly 加速单元，WOLA 模块优化前后的程序运行时间比较如表 4-10。

4.2.5 并行存储器加载指令

指令在计算的同时执行数据存储器的数据加载操作，可以达到隐藏访问存储器操作延迟的目的。并行存储器存取指令容易添加到硬件的数据通路，符合语音算法多滤波运算的特点。

4.2.5.1 硬件设计

由于 ASIP 的 MAC 单元与存储器访问相关的部件可以并行执行，在执行乘累加/减运算的同时加载存储器数据至寄存器，硬件实现非常简单，非常容易添加到数据通路，因此添加并行存储器加载指令，合并 MAC 指令与存储器加载指令。如图 4-17，译码单元在译码时刻分别掩码得到 MAC 单元的操作数以及地址产生单元的操作数，在执行级完成 MAC 运算与存储器数据加载操作。

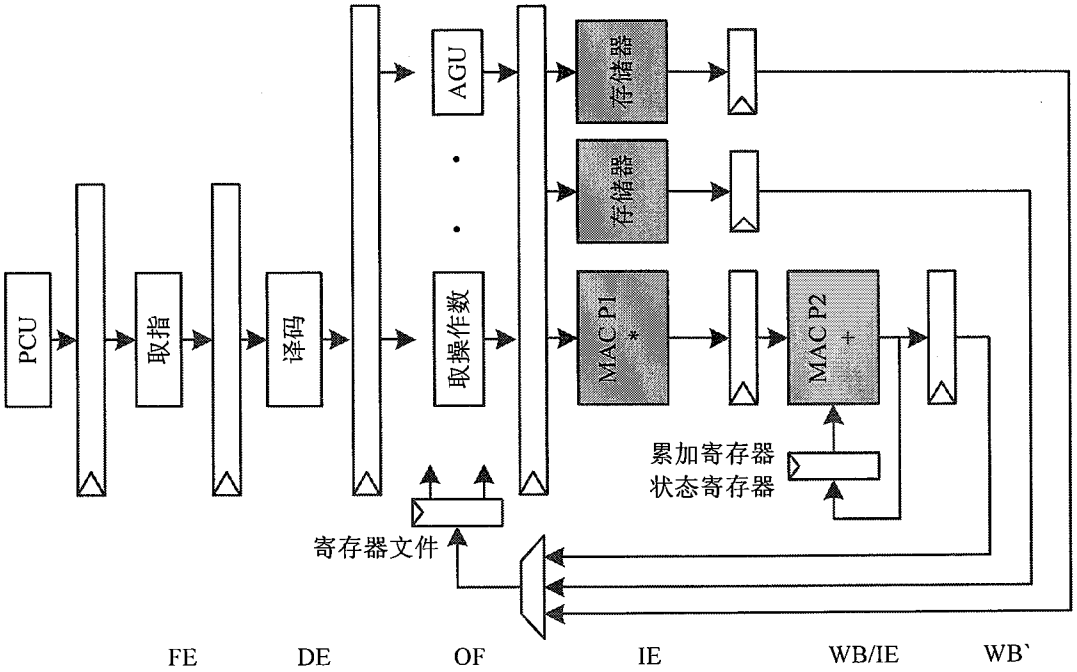


图 4-17 并行加载指令流水线

对于滤波运算，需要在存储器中加载滤波器系数与输入数据，通过乘累加实现滤波运算。音频处理算法中涉及到大量的滤波运算，因此在 ASIP 中添加 fir

指令，其两个操作数分别来源于两块数据存储器，执行乘累加/减运算。fir 指令合并了存储器访问指令与 MAC 指令，减少了指令数目，因此减少了译码单元与控制逻辑的翻转；将操作数从存储器直接送至 MAC 单元，避免了存储器与通用寄存器的数据交互，加速了指令执行。

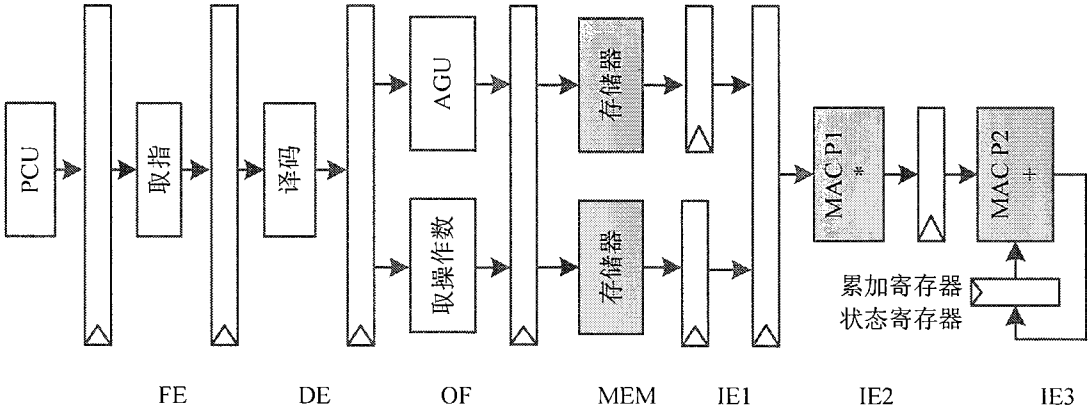


图 4-18 fir 指令流水线

4.2.5.2 软件开发工具实现

在编译器中添加遍处理（ParallelMemAccessingPass），在基本块内检查指令，检测可以并行的存储器加载指令和 MAC 单元指令，发射并行存储器加载指令，并将原存储器加载指令与 MAC 指令删除。

PM.add(create ParallelMemAccessingPass());

ParallelMemAccessingPass 的执行过程如图 4-19，以函数为遍处理单位，遍历其中的每一个基本块，进行存储器并行访问指令优化。优化过程即是遍历基本块内的目标机指令，寻找可以并行的 MAC 指令与存储器访问指令，如果可以合并，则发射存储器并行访问指令，同时将原来的指令在基本块内移除。

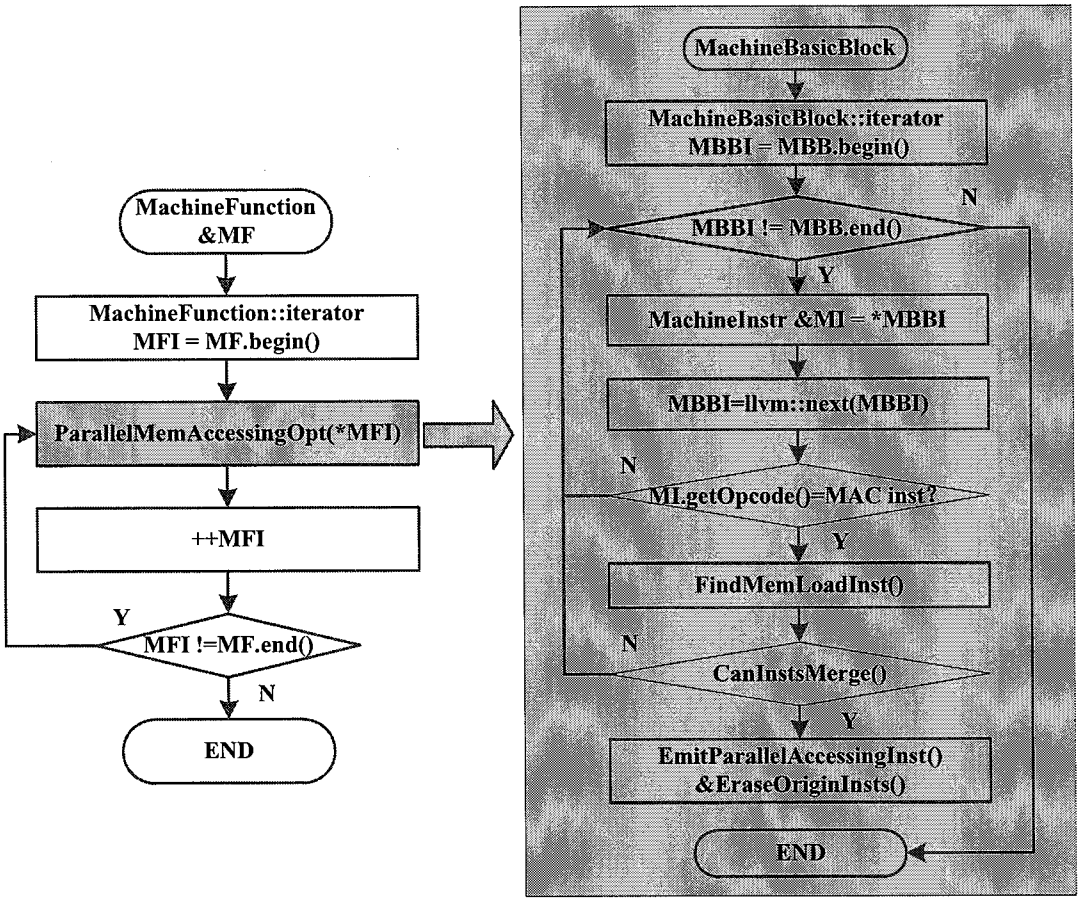


图 4-19 ParallelMemAccessingPass 流程图

添加并行存储器加载指令前后，对应用程序进行仿真，仿真结果如表 4-11。

表 4-11 程序执行周期统计

功能模块	优化后运行周期	优化前运行周期
IN	32	32
OUT	64	64
WOLA AFB	576	832
FFT	1485	1485
NOISE REDUCE	3400	3400
WDRC	1205	1205
WOLA SFB	1344	1344
IFFT	1485	1485
TOTAL	9591	9847
		256 cycles saved

4.2.6 本节小结

基于数字助听器算法的汇编级代码分析，频繁使用的指令组合与指令序列被识别，并以专用指令或者协处理器的形式添加到 ASIP 的数据通路，如位反寻址、

零开销循环、对数、开方、倒数、蝶形运算、并行存储器加载等专用指令。自定制的结构有效的提高了程序的运行效率，优化后的运行时间约为优化前运行时间的 37.2%，如表 4-14。

表 4-12 不同功能模块的周期消耗

	优化前	优化后
块操作	480	96
WOLA	12506	4890
噪声消除	9579	3400
WDRC	3245	1205
Total cycles	25810	9591

4.3 低功耗设计

本文基于标准单元进行低功耗的数字助听器 ASIP 设计，所能采用的低功耗技术通常是在设计的软件级、系统级、寄存器传输级与逻辑级，这些功耗的优化策略也是 ASIP 设计中普遍适用的。以下分别对本文中采用的低功耗技术进行介绍。

4.3.1 ASIP 软件级

处理器周期性的处理语音帧数据，通常会有大部分时间是处于空闲状态，有效的使用睡眠模式可以很好的降低处理器空闲状态时的动态功耗。通过软件编程实现中断唤醒、数据同步、语音处理、睡眠模式的状态转换，可以有效的节省处理器的动态功耗。

汇编优化可以有效的使用处理器的寄存器，从而减少对数据存储器的访问，有效的降低存储器的功耗；高效的使用指令集，可以有效的提高处理器的运算能力，减少运行时间。除此之外，在 ASIP 的汇编代码阶段，还进行了控制流优化与强度消减，把代价比较高的运算替换为目标机上代价比较低的指令，有效的提高了处理器的执行效率。

4.3.1.1 应用算法优化

对于给定的应用，功耗取决于操作量（加法、乘法、逻辑操作等）、存储器访问量以及存储器空间。对应用进行算法选择与优化，可以有效的降低系统功耗，

如简单的算法通常消耗更少的功耗；优化的算法可以减少操作、减少对存储器的访问、减少程序控制，从而节约功耗。在数字助听器的实现过程中，算法的优化也是非常重要的。

1）基于查表法实现 WDRC

宽动态压缩算法的关键部分即是根据信号的声压级和听力补偿曲线计算增益，但是通常增益的计算过程过于复杂，不适合嵌入式的低功耗实现，因此对算法进行优化是低功耗实现的关键。

基于查表法获得增益的 WDRC 实现可以有效的避免过于复杂的 gain 计算，快速获得增益 *gain* 。首先，把助听器每个通道输入信号的 *SPL* 范围均分为 48 个区间，以每个区间中心的 *SPL* 值对应的增益，作为该区间的增益。预先根据患者的听力补偿 I/O 曲线将每一个通道的 48 个增益值计算出来，并作为表格存放在存储器中。通过查找平均能量 $p(n)$ 的最高非零位 M，在其对应通道 n 内通过查找表获得增益 *gain* ，如图 4-21。通过使用添加的专用指令 wclz，基于查表法的 WDRC 算法计算复杂度和计算量都大大减少。

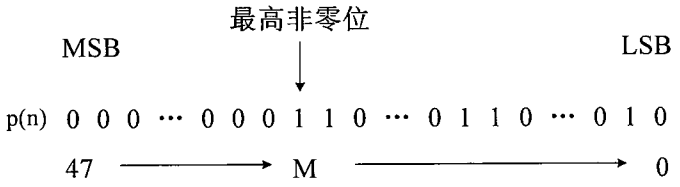


图 4-20 查找最高非零位

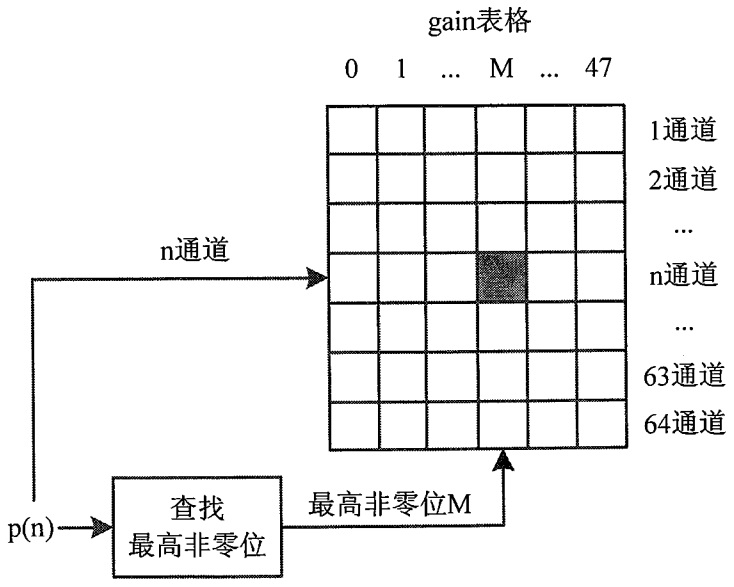


图 4-21 WDRC 算法的增益查表实现

2）利用 K/2 点复序列 FFT 计算一个 K 点实序列 FFT

由于 WOLA 滤波器组的 K 点 FFT 输入数据均为实数，因此可以采用 K/2 点的 FFT 与反变换实现 K 点 FFT 及其反变换[8]，从而减少约一半的运算量与执行时间。以 FFT 为例做简单介绍，反变换可参考 FFT 的逆过程，如对 K 点实序列 $x(n)$ 进行 FFT 变换，首先构造 K/2 点复序列 $z(n)$：

$$z(n)=g(n)+jh(n),0\le n<K/2 \tag{4.11}$$

其中 $g(n)=x(2n)$ ，$h(n)=x(2n+1)$，则有：

$$g(n)=\frac{1}{2}(z(n)+z^*(n)),0\le n<K/2 \tag{4.12}$$

$$h(n)=\frac{1}{2j}(z(n)-z^*(n)),0\le n<K/2 \tag{4.13}$$

假设 $Z(k)$ 为 $z(n)$ 的 K/2 点 FFT 变换，推导得到 $g(n)$ 和 $h(n)$ 的 K/2 点 FFT 变换如下：

$$G(k)=\frac{1}{2}(Z(k)+Z^*(<-k>_{K/2})),0\le k<K/2 \tag{4.14}$$

$$H(k)=\frac{1}{2j}(Z(k)-Z^*(<-k>_{K/2})),0\le k<K/2 \tag{4.15}$$

则 $x(n)$ 的 K 点 FFT 变换可表示为：

$$\begin{aligned}X(k)&=\sum_{n=0}^{K-1}x(n)W_K^{nk}=\sum_{n=0}^{K/2-1}x(2n)W_K^{2nk}+\sum_{n=0}^{K/2-1}x(2n+1)W_K^{(2n+1)k}\\&=\sum_{n=0}^{K/2-1}g(n)W_{K/2}^{nk}+\sum_{n=0}^{K/2-1}h(n)W_{K/2}^{nk}W_K^k\\&=\sum_{n=0}^{K/2-1}g(n)W_{K/2}^{nk}+W_K^k\sum_{n=0}^{K/2-1}h(n)W_{K/2}^{nk}\\&=G(<k>_{K/2})+W_K^kH(<k>_{K/2}),0\le k<K\end{aligned} \tag{4.16}$$

则 $x(n)$ 的 K 点 FFT 变换 $X(k)$ 为：

$$\begin{aligned}X(k)=&\frac{1}{2}(Z(<k>_{K/2})+Z^*(<-k>_{K/2}))+\\&W_K^k*\frac{1}{2j}(Z(<k>_{K/2})-Z^*(<-k>_{K/2})),0\le k<K\end{aligned} \tag{4.17}$$

其中<>表示求模运算。

综上，用 K/2 点复序列 FFT 计算一个 K 点实序列 FFT，其实现过程如下：

首先，将 K 点实序列$x(n)$按偶数项和奇数项分解为两个序列$g(n)$和$h(n)$，分别作为复序列的实部序列和复部序列，构造出 K/2 点复序列$z(n)$；

计算 K/2 点复序列$z(n)$的 FFT $Z(k)$；

最后，根据频域变换性质，利用式(4.17)，以 K/2 点复序列 FFT $Z(k)$ 实现 K 点实序列 FFT $X(k)$。

根据 WOLA 滤波器组算法特点，在综合阶段，IFFT 运算模块的输出必定是实序列，可以采用与上文类似的简化，以 K/2 点复序列 IFFT 实现 K 点实序列 IFFT。直接给出结果：

设频域序列$Y(k)$，$0 \le k < K$，构造 K/2 点新序列$V(k)$：

$$\begin{aligned} &V(k) = (Y_1(k) + Y_2(k)) + W_K^{-k} * \frac{j}{2} (Y_1(k) - Y_2(k)) \\ &Y_1(k) = Y(k) \\ &Y_2(k) = Y(k + N/2), 0 \le k < K/2 \end{aligned} \tag{4.18}$$

设$v(n)$为$V(k)$的 K/2 点 IFFT 变换，则$v(n)$为复序列，则得到$Y(k)$的 K 点实序列 IFFT 的结果$y(n)$：

$$\begin{aligned} &y(2n) = real(v(n)) \\ &y(2n+1) = imag(v(n)), 0 \le n < K/2 \end{aligned} \tag{4.19}$$

IFFT 的计算过程与 FFT 的逆过程类似，其过程如下：

首先，利用频域变换性质，将 N 点实序列 FFT 整合为 N/2 点复序列 FFT，过程可参考分离。

然后，计算 N/2 点复序列的 IFFT。

最后，将 N/2 点复序列 IFFT 的实部作为偶数项，复部作为奇数项，构建出 N 点实序列（项数从零开始）。

将 K 点的实序列(I)FFT 变换转化为 K/2 点的复数序列(I)FFT，计算量减少将近一半，缩短程序运行时间，同时减少(I)FFT 对数据存储器的访问。

表 4-13 FFT/IFFT 优化仿真对比

	优化前	优化后
FFT+IFFT	2970	1685

4.3.2 系统级

在系统级，算法被映射到可编程处理器或者硬件加速单元。算法的软硬件划分显得尤为重要，既要满足系统的功耗要求，又要保证数据处理的实时性。算法中运算密度大、规整的部分在保证系统不失灵活性的前提下，可以通过专用的或者可配置的硬件加速器实现，否则只能通过处理器编程实现。系统的灵活性使得当算法改变时，通过编程即可实现系统升级。

4.3.2.1 存储器功耗优化

图 4-22 给出了 ASIP 的内核和存储器的功耗分布，可见存储器功耗占了整个系统功耗的 63.4%。因此有必要分析存储器的功耗来源，并对其进行功耗优化。动态功耗是存储器功耗的主要构成部分，其表达式如式(4.20)：

$$P_{dymatic}=\partial\cdot f\cdot C\cdot V_{dd}^{2} \tag{4.20}$$

其中∂为活动因子，f 为时钟频率，C 为负载电容，V_{dd}为工作电压。对于采用商用 SRAM IP 的设计，降低动态功耗只有尽可能的减少对存储器的访问，从而降低存储器的活动因子∂；或者缩小存储器的容量，从而降低存储器的电容值C。优化的算法通常可以减少对存储器的访问，如通过 K/2 点的复序列 FFT 实现 K 点实序列 FFT；当算法确定后，存储器的容量与访问次数也随之确定。根据算法的需要确定处理器的程序存储空间和数据存储空间，尽量减小存储器容量，可以有效的减少存储器的功耗。通过存储器使能关断，只在有效的存储器访问时使能存储器，可以有效减少无效访问带来的功耗损失。

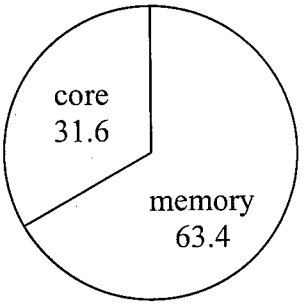


图 4-22 ASIP 功耗分布

1）存储器子系统

为了有效的降低存储器系统的功耗，需要在系统级根据算法确定程序存储器和数据存储器的容量，存储器容量越小，存储器的功耗就相应越小。本文的设计中采用了 3 块双端口的片上集成存储器，一块 768x32 比特的程序存储器，两块 1Kx24 比特的数据存储器。

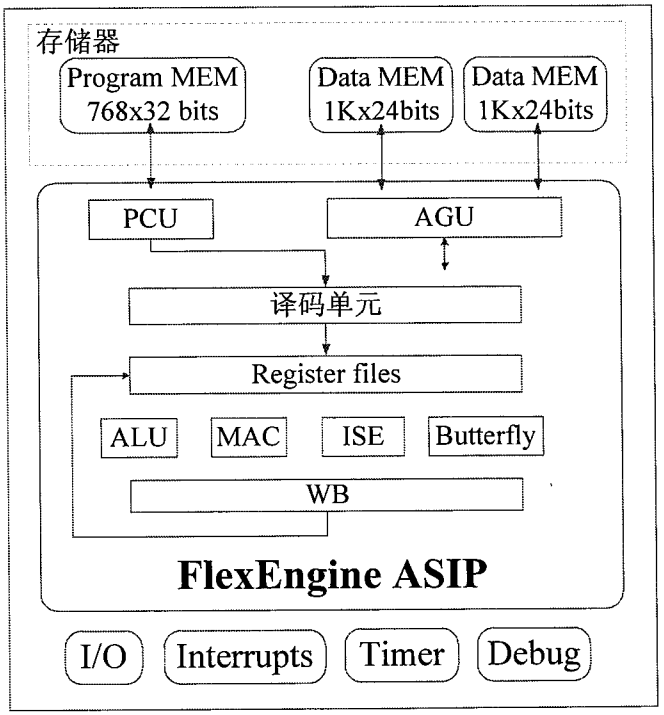


图 4-23 ASIP 结构框图

2）存储器分割

通过将存储器分割为若干块，每块存储器具有更小的容量与电容 C，从而减少了存储器访问的有效电容，但代价是面积的增加以及更加不规则的版图布局。

根据功能划分存储器可以获得较好的效果，统计不同地址的访问次数，根据访问次数将存储器划分为若干不等的存储器段，其中访问越密集的地址段划分容量越小。当数据不可统计或者应用不断更新时，可以在面积增加与版图的折中下将存储器均等划分。

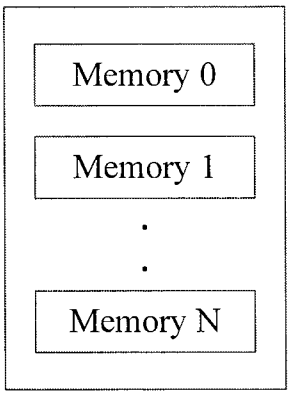


图 4-24 存储器分割

3）循环缓存

当主程序包含许多小的循环体，且循环指令执行周期占总程序执行周期比例较大时，可将循环体程序在首次执行时存入缓存空间，而后取指单元访问缓存空间而非程序存储空间，若存储空间与缓存空间容量相差较大，可以节省较多的功耗。

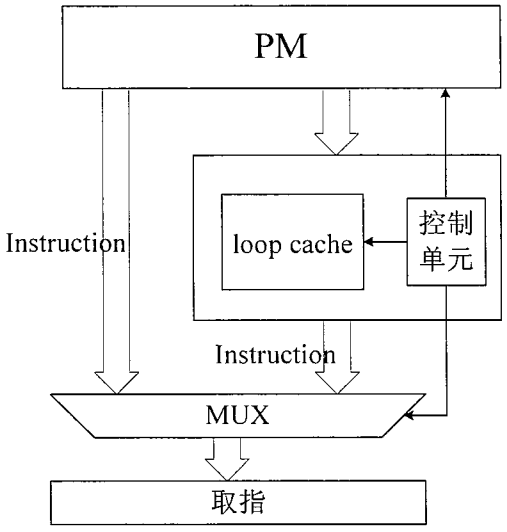


图 4-25 循环缓存

4.3.3 寄存器传输级

（1）针对指令集架构（instruction set architecture，ISA），添加专用指令与加速单元，提高程序执行效率；

（2）添加顶层时钟门控将处理器空闲状态时的时钟关掉，引入处理器睡眠模式，减少空闲状态下的动态功耗；

（3）采用操作数隔离，减少数据通路的无效翻转；

4.3.3.1 睡眠模式指令

在专用指令集处理器的设计中，通过添加专用指令与加速单元，可以显著减少处理器的运行时间，从而使处理器在更多的时间内处于空闲状态。配合睡眠模式可以很好地减少动态功耗，将运行时间的减少体现为硬件功耗的降低。

在睡眠模式下，处理器核时钟被关断，理想情况下，处理器核的动态功耗为零，即其处于低功耗状态。因此当处理器执行完应用程序，将空闲状态的处理器转入睡眠模式可以减少不必要的功耗开销。

1）硬件设计

在 ASIP 中引入睡眠模式，通过 idle 指令开启睡眠模式，处理器时钟关断，外围设备模块保持工作，中断模块检查外部中断输入，当有中断发生时，唤醒处理器，如图 4-26。

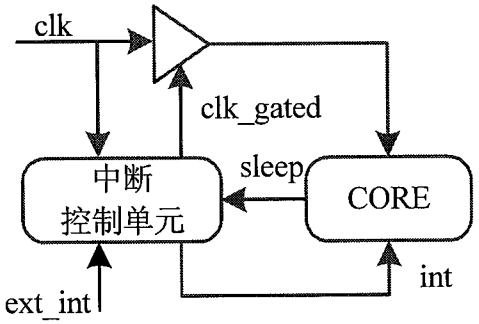


图 4-26 睡眠模式顶层时钟门控

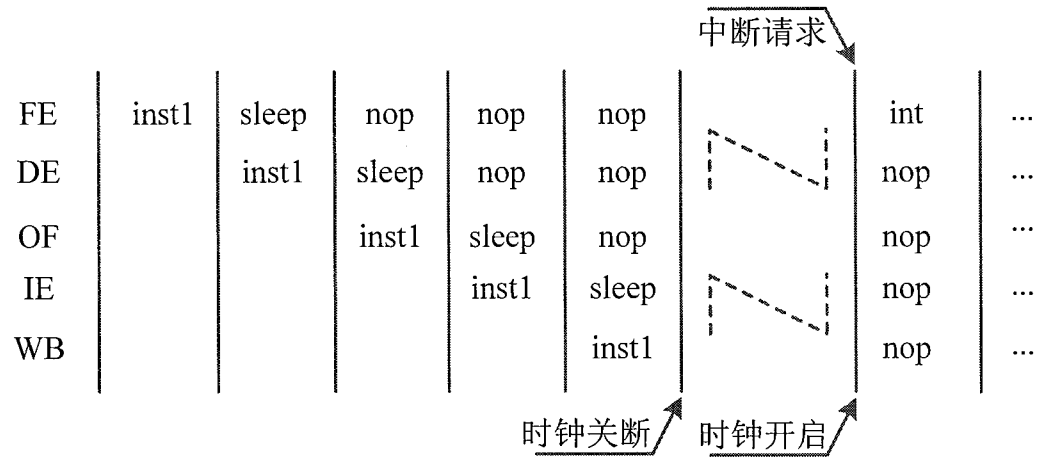
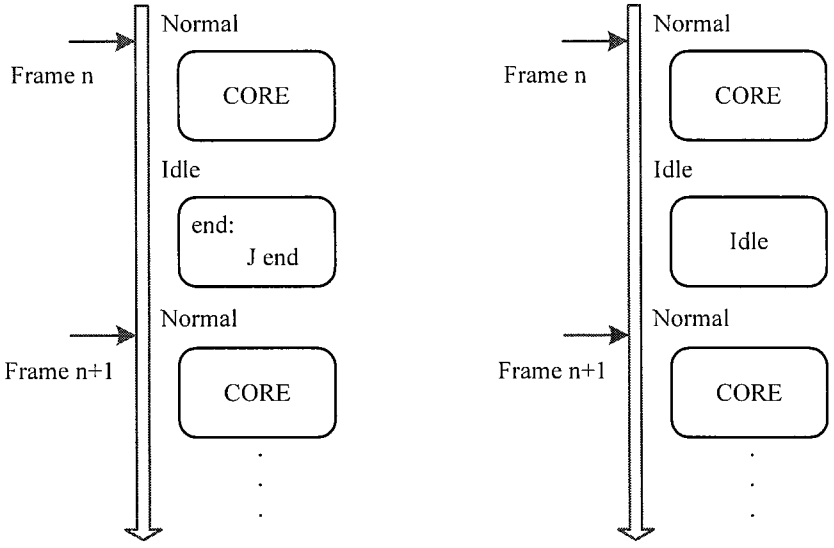


图 4-27 睡眠模式指令流水线

如图 4-27 所示，ASIP 在译码级检测到 idle 指令，然后停止取指，并将后续指令置为空指令（nop），同时向中断控制单元发出 idle 指示信号。当 idle 进入执行级，idle 前的指令都已经执行完毕，中断控制单元发出时钟关闭信号 clk_gated，关闭处理器核的时钟，处理器核进入低功耗状态，中断控制单元以及相关外围保持正常运行。直到外部中断请求出现，处理器核的门控时钟信号 clk_gated 变为无效，处理器核时钟恢复，继而从低功耗模式进入工作状态。

在添加睡眠模式前，处理器处于空闲状态时，以跳转指令 j $将程序维持在结束地址，等待中断请求。添加睡眠模式后，在程序结尾处使用 idle 指令，将空闲时的处理器转入睡眠模式，可以有效的避免处理器空闲状态时的动态功耗，如图 4-29。



(a)j 指令将程序维持在结束地址　　(b)idle 指令的睡眠模式

图 4-28 处理器工作状态

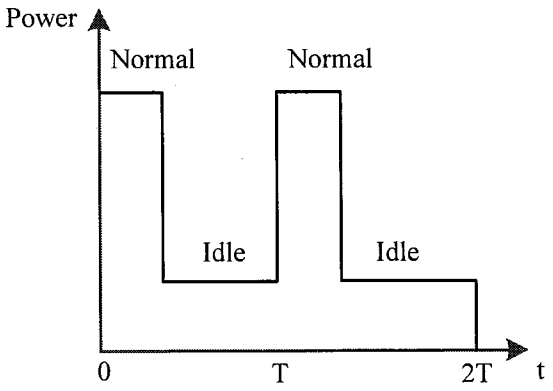


图 4-29 睡眠模式功耗

2）软件开发工具实现

在 C 语言源程序中以内联汇编（Inline Assembly）[9][10]的形式使用 idle 指令，在程序运行的结尾将处理器转入低功耗的工作状态。

在编译器的目标机头文件‘flexengine.H’中定义宏：

```
#define SLEEP() asm ("idle")
```

在源程序文件中通过使用宏 SLEEP()内联 idle 指令：

```
int main(void){
...
SLEEP();
...
}
```

对于实时音频处理系统，处理器周期性的处理外部语音采样信号。以数字助听器为例，处理器的输入缓冲器缓存采样的语音信号，接收一帧即向处理器发出中断信号，请求进行数据处理，处理结束后处理器便处于空闲状态，等待下一帧的中断处理信号。因此睡眠模式很好的满足了音频处理算法的这一特点。

4.3.3.2 操作数隔离

在处理器中，存在多个执行单元，如地址产生单元 AGU、算术逻辑单元 ALU、乘累加单元 MAC 以及指令集扩展单元，不同的执行单元之间通常存在相同的激励，因此当某一执行单元工作时会不可避免的引起其他执行单元的信号翻转，操

作数隔离可以有效的避免系统冗余翻转，即在不同的执行单元之间或者内部添加透明锁存器和使能控制电路，如图 4-30。

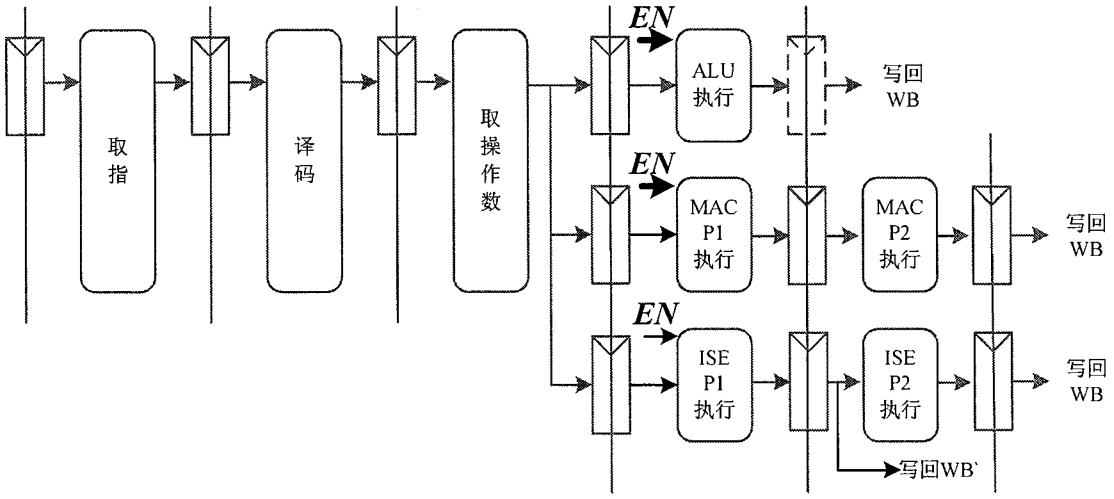


图 4-30 操作数隔离

4.3.4 逻辑级

在逻辑级，插入时钟门控，可以将不工作的功能单元及时关断，关断后这些模块内部的时钟不翻转，寄存器的值保持不变，理想情况下将不会产生动态功耗。在设计阶段或者综合阶段都可以手动或者自动的插入时钟门控，以降低系统的动态功耗。

4.3.5 本节小结

综合运用低功耗的 ASIP 设计技术可以有效地降低系统的功耗。通过指令集优化、与算法优化，可以有效的提高程序的运行效率，优化后的运行时间约为优化前运行时间的 32.2%，如表 4-14。

表 4-14 不同功能模块的周期消耗

	优化前	优化后
块操作	480	96
WOLA	12506	3605
噪声消除	9579	3400
WDRC	3245	1205
Total cycles	25810	8306

对于本文实现的数字助听器系统，算法要求每桢数据的处理时间为 19456

个时钟周期。由于 ASIP 设计开始阶段，算法的实现指令繁琐，执行时间超过了一帧音频数据的处理时间，不满足系统的实时性。因此对数字助听器功耗的仿真从添加特殊运算指令开始，不同方法对系统功耗的优化结果如图 4-31，优化后的功耗为优化前的 21%，ISE 指特殊运算指令。

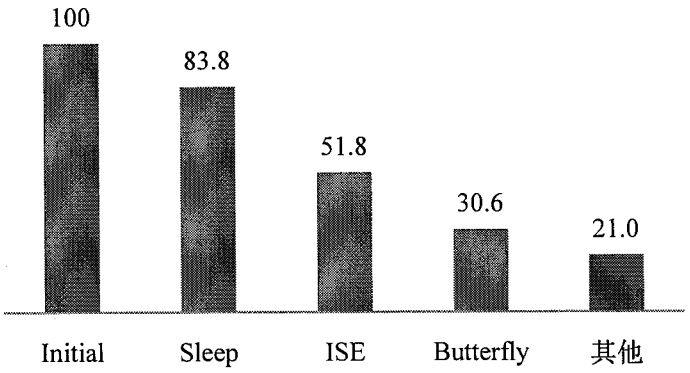


图 4-31 归一化的功耗

4.4 本章小结

本章主要介绍了低功耗数字助听器 ASIP 的指令集优化、处理器相关的编译器代码优化、与低功耗设计。

针对音频信号处理应用，以数字助听器关键算法为例，进行汇编级的代码分析，识别频繁使用的指令序列与指令组合，并将其以专用指令或者加速单元的形式添加到 ASIP 处理器的数据通路，如睡眠指令、位反寻址、零开销循环、开方、除法、log、蝶形运算、并行数据加载等专用加速指令。自定制的结构使得 ASIP 同时具备灵活、高效的运算特点，非常适合高性能、低功耗的音频处理应用，如助听器、便携音频设备等。

由于便携式语音设备对功耗要求非常高，所以在 ASIP 设计的各个阶段都针对功耗进行了优化设计。以数字助听器的 ASIP 设计为例，本文在软件级，进行了算法优化，减少了算法的复杂度与存储器访问。在系统级，根据算法的需要确定处理器的程序存储空间和数据存储空间，尽量减小存储器容量；进行存储器分割与循环缓存减小存储器的访问电容，降低了存储器的功耗。在寄存器传输级，采用操作数隔离，减少数据通路的无效翻转；添加顶层时钟门控将处理器空闲状

态时的时钟关掉，引入处理器睡眠模式，减少空闲状态下的动态功耗；添加专用指令与加速单元，提高程序执行效率。在逻辑级，插入时钟门控。同时在综合阶段采用低功耗的设计约束。以上设计方法都可以有效的减少系统的功耗，从而使处理器获得更多的待机时间。

参考文献

[1]Glokler T, Meyr H. Power reduction for ASIPs: A case study[C]//Signal Processing Systems, 2001 IEEE Workshop on. IEEE, 2001: 235-246.

[2]Glokler T, Bitterlich S, Meyr H. Increasing the power efficiency of application specific instruction set processors using datapath optimization[C]//Signal Processing Systems, 2000. SiPS 2000. 2000 IEEE Workshop on. IEEE, 2000: 563-570.

[3]Kates J. Signal processing for hearing aids[J]. Applications of Digital Signal Processing to Audio and Acoustics, 2002: 235-277.

[4]Fastl H, Zwicker E. Psychoacoustics: Facts and models[M]. Springer, 2006.

[5]Chang K C, Kuo Y T, Lin T J, et al. Complexity-effective dynamic range compression for digital hearing aids[C]//Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on. IEEE, 2010: 2378-2381.

[6]于增辉，黑勇，薛金勇，等. 助听器多通道宽动态范围压缩的低功耗硬件实现[J]. 哈尔滨工程大学学报，2012，33(1)：106-111.

[7]Crochiere R E, Rabiner L R. Multirate Digital Signal Processing[M]. Englewood Cliffs: Prentice Hall, 1983.

[8]Mitra S K. Digital Signal Processing - A Computer-based Approach [M]. Second Edition. Beijing: Tsinghua University Press, 2006. pp147-149.

[9]Hohenauer M. C compilers for ASIPs[M]. Springer, 2010.

[10]Coleman C L. Using Inline Assembly with gcc[J]. 1988.

第5章 数字助听器 ASIP 的测试验证

本章介绍了数字助听器 ASIP 实现的系统功能仿真、FPGA 验证、以及物理实现，并对测试结果进行了分析。

5.1 系统功能仿真与 FPGA 验证

基于数字助听器 ASIP 设计 32 通道的数字助听器，数字助听器算法通过软件开发工具编译为可执行代码，在系统上电时加载至 ASIP 的程序存储器，通过处理器执行程序完成数字助听器的听力补偿与噪声消除功能。设计的基本参数如表 5-1。

表 5-1 数字助听器系统特征参数

采样率 FS/ kHz	16
分析窗长度 LA	256
综合窗长度 LS	256
降采样因子 M	32
通道数	32
功能	听力补偿 WDRC、噪声消除 NR

5.1.1 系统功能仿真

论文采用 Modelsim 和 VCS 进行 RTL 级代码功能验证。系统的实现框图如图 5-1，其中 Codec 模拟 ADC/DAC 串行接口，Flash 为 SPI 串行 EEPROM 模型，Clk_gen 为系统时钟发生器，整个系统采用单一的全局时钟。Rst_gen 为复位信号发生模块，低电平有效。UART、INT 和 GPIO 是 ASIP 的外围设备。

16KHz 采样率音频文件定点化为 16bit 声音数据，以 10 进制格式保存在 veri.dat 文件中，作为输入音频信号，用来验证 ASIP 语音处理程序的正确性，仿真时该文本通过 CODEC 模块以采样点的形式送入处理器，继而进行分块、多通道分离、噪声消除与听力补偿。

memory.txt 文本保存 ASIP 的可执行程序，仿真开始时该文本被自动加载至片外 Flash，ASIP 中的 bootload 模块将 Flash 里面的程序加载至内部程序存储器，然后开始执行可执行程序。通过修改 memory.txt 文本，ASIP 可以运行不同的程

序，实现处理器的软件编程。

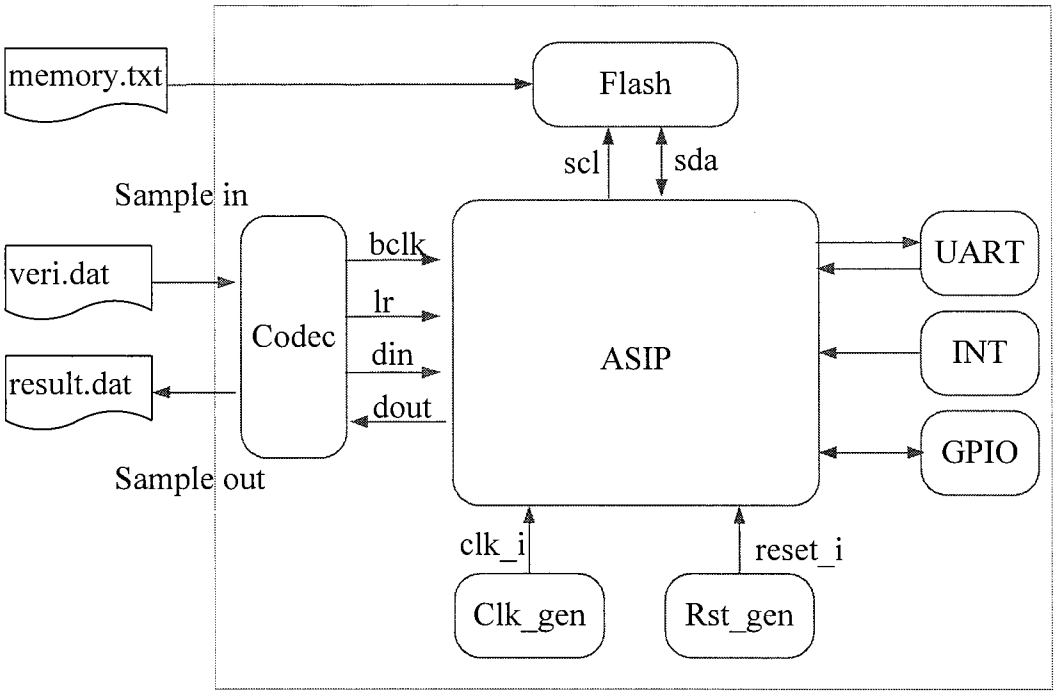


图 5-1 系统实现框图

经过处理器 RTL 实现处理的 16bit 音频数据，通过 CODEC 模块输出至文件 result.dat，与 C 语言级程序的结果进行比较，结果一致即验证了系统功能的正确性。通过 CoolEdit 可以将输出.dat 文件转换为音频文件，进而验证处理器的助听器功能。经过仿真，C 语言级程序运行结果与处理器的执行结果一致，验证了系统的正确性。

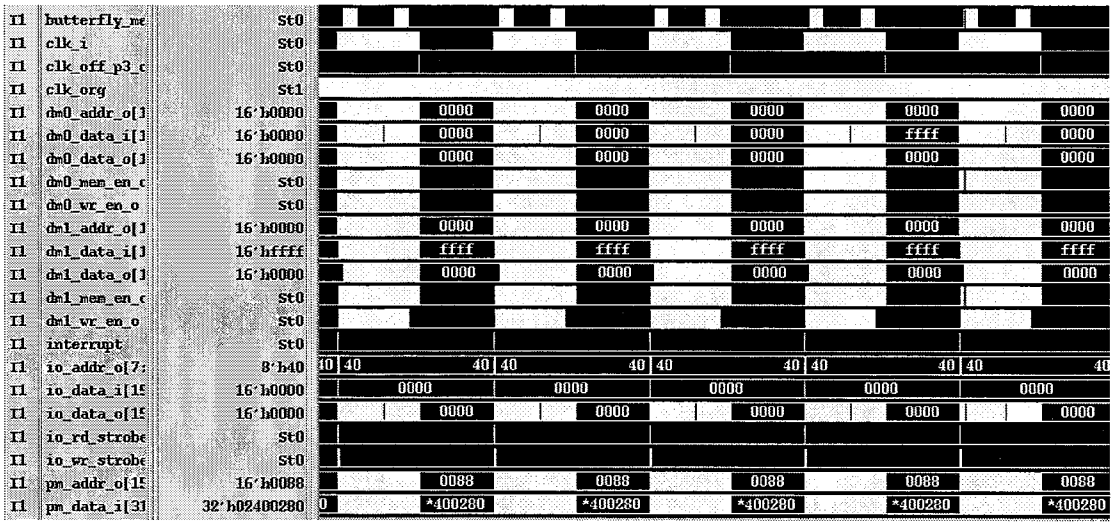


图 5-2 ASIP 数字助听器程序仿真

5.1.2 FPGA 功能验证

系统的 FPGA 功能验证如图 5-3 所示，主要包括三个部分：Stratix II EP2S180F1020C3 开发板[1]，TI 的 TLV320AIC23 系列 CODEC 音频解码器（AIC23B）[2]和 Winbond 的 W25X10AV Flash 存储器[3]。其中，FPGA 开发板为主体，实现数字助听器 ASIP 的音频处理功能；TLV320AIC23 芯片内置 A/D 和 D/A，作为音频 CODEC；Flash 存储器为外置存储器件，用来保存处理器中程序存储器所需要执行的代码和所需要配置 CODEC 芯片的寄存器参数，烧写 Flash 存储器可以更新处理器的执行程序，完成处理器软件编程。

ASIP 的 RTL 代码在 Quartus 中综合完成后，通过 USB Blaster 下载到 FPGA 中。Flash 中烧录着语音处理算法程序的可执行代码，该程序实现了数字助听器的多通道分离、听力补偿和噪声消除。上电后可执行程序由 Flash 加载至处理器的程序存储器中，之后处理器进入正常工作状态，外界模拟音频信号经过 CODEC 中的 A/D 转换成 16 位数字信号，通过串行比特流的形式送入 ASIP，经过 ASIP 处理后的数据送入 CODEC 芯片，由芯片内部的 D/A 转换成模拟音频信号输出。

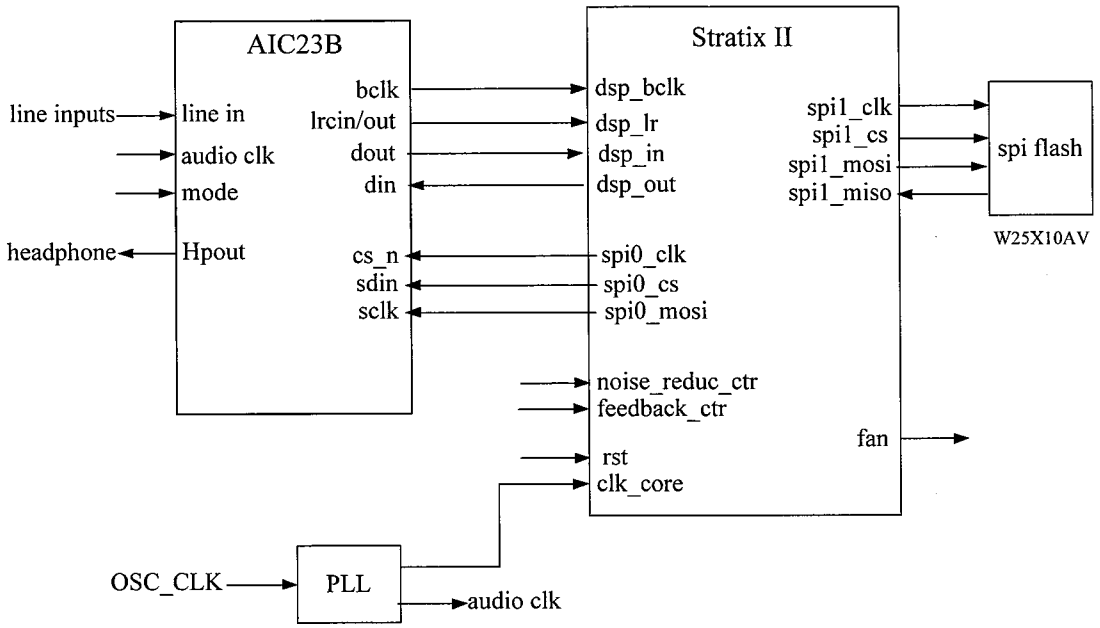


图 5-3 FPGA 功能验证结构框图

系统测试中，测试数据是一段带有背景噪声的声音数据，ASIP 通过执行程序存储器中的机器码实现多通道分离、听力补偿和噪声消除。经验证处理后的声音背景噪声得到抑制，声音变得清晰可辨。此外，通过 SignalTap 显示处理器内部的信号波形，证明了程序加载过程和程序流的正确性，进一步验证了 ASIP 的

功能。

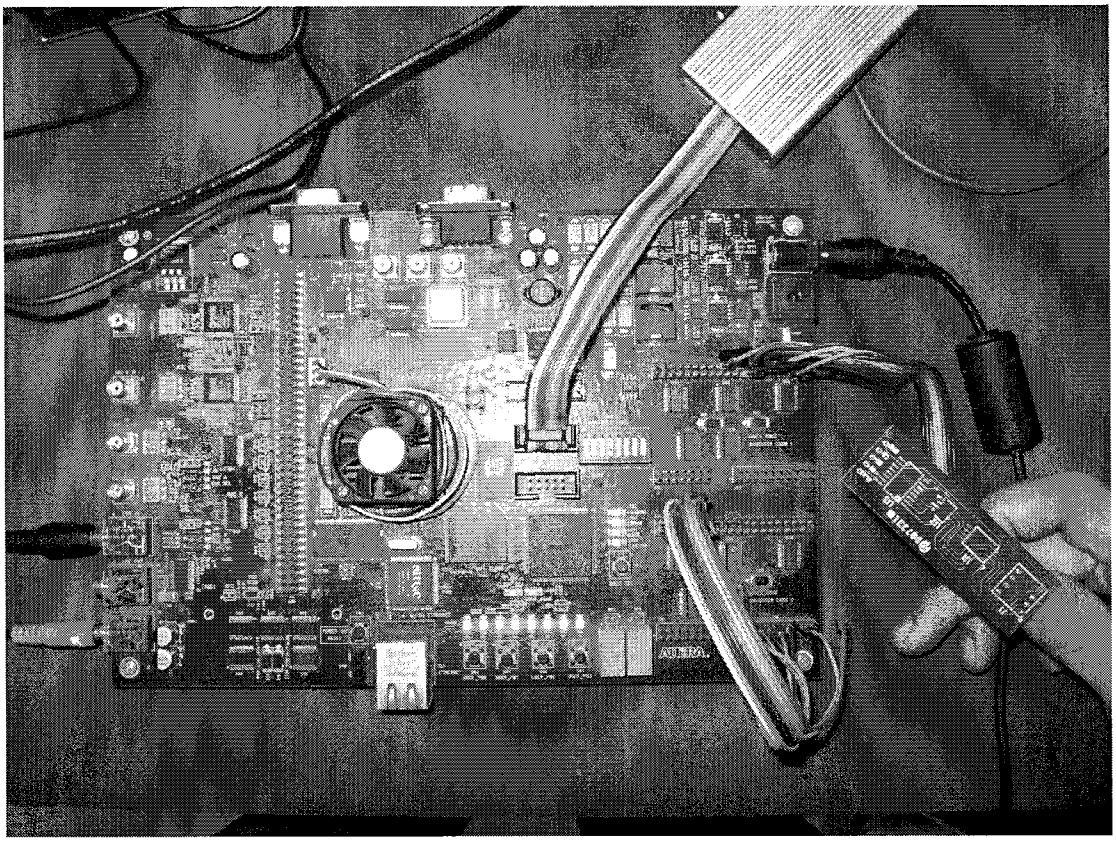


图 5-4 FPGA 功能验证

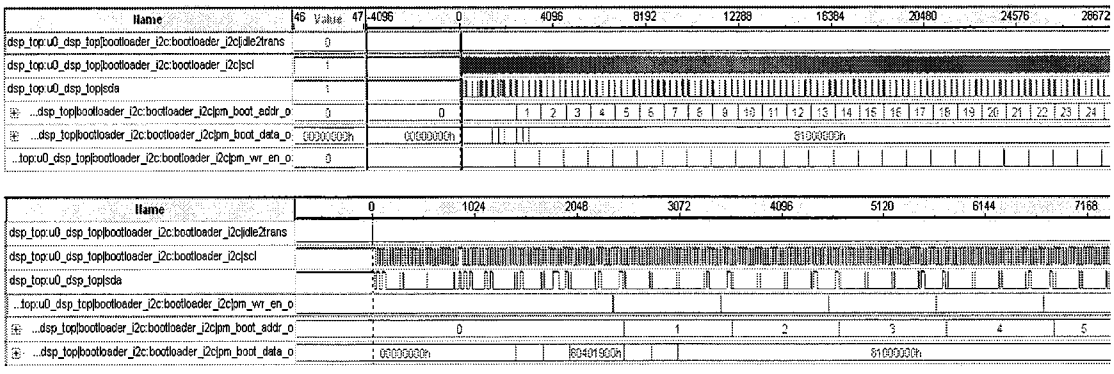


图 5-5 ASIP 从片外 Flash 加载程序过程

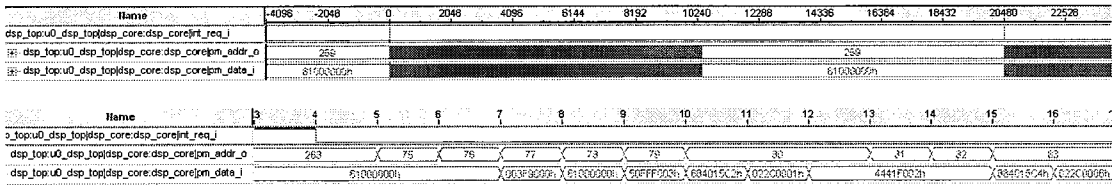


图 5-6 ASIP 程序执行过程

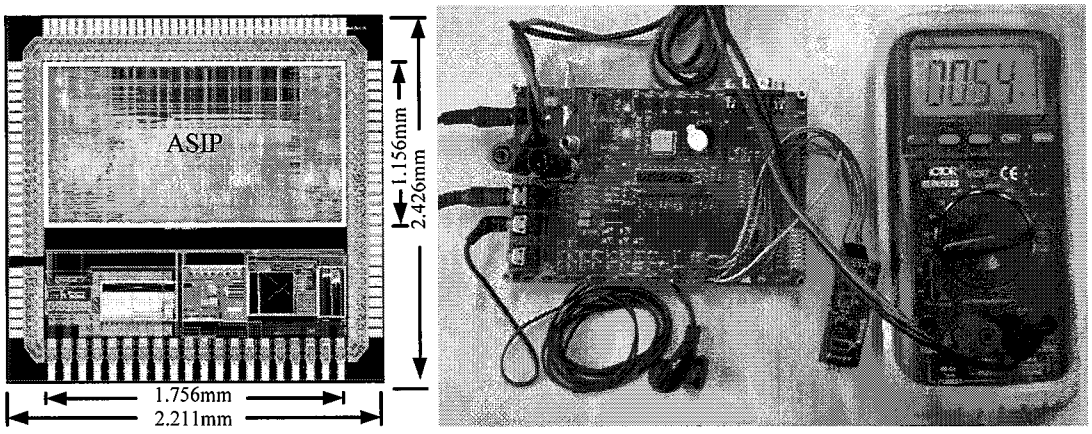
5.2 系统物理实现与结果分析

最后，本文基于 TSMC 130nm 工艺完成设计流片，图 5-7 为芯片的版图与芯片测试图片。经过测试，ASIP 的最高工作频率可以达到 100MHz，由于指令大部分是单周期的，因此运算能力接近 100MIPS，可以满足大部分音频系统的处理要求。ASIP core 的面积约 0.96mm^2，对于大部分便携式音频系统来说也是可以接受的。

表 5-2 ASIP core 设计参数

内核功耗	
本设计	70μW/MHz@130nm
国芯 C305	120 μW/MHz@130nm
ARM Cortex-M0	90 μW/MHz@180nm
NXP CoolFlux	80 μW/MHz@130nm

将数字助听器的目标代码通过 flash 烧写，在系统上电时由 bootloader 加载至程序存储器，验证了该系统能较好的实现数字助听功能，工作在 8MHz 频率、1.2V 工作电压时，处理器功耗约 0.963mW，设计参数如表 5-3。



(a)芯片版图　　(b)芯片测试

图 5-7 芯片版图与测试

表 5-3 设计参数

工艺	TSMC130nm
面积	SOC 5.36mm^2；　ASIP 2.03mm^2（含片上存储器）
时钟	8MHz
电压	1.2V
功耗	0.963mW

由于数字助听器系统的算法多样，不同的实现具有不同的特征参数，如采样率、通道数，包含的功能模块不同，相同的功能模块采用的算法也不相同，而且

很多研究只关注单个模块的算法与实现，对于整个数字助听器系统的实现比较少，因此很难将本文的设计直接与其他工作对比。表 5-4 给出了本文与已发表文献的对比，文献[4][5]均采用了更高的工艺、更低的工作电压，虽然文献[4]功能略多，文献[5]采样率较高，但是本文的通道数明显多于文献[4] [5]，系统的计算任务量仍可比于甚至多于文献[4] [5] ，因此本文的流片结果与文献[4] [5]的结果相比具有可比性和优势。为方便比较引入归一化的比较指标，归一化面积[6]和归一化功耗，$Area_0$ 表示本设计的面积，$technology_0$ 表示本设计采用的工艺，可以看出本文的设计在面积上有一定的可比性甚至优势，同时具有较低的功耗。

$$Normalized\ Area = \frac{Area}{Area_0} \times \left(\frac{technology_0}{technology} \right)^2 \tag{5.1}$$

$$Normalized\ Power = \frac{power}{power_0} \times \left(\frac{voltage_0}{voltage} \right)^2 \times \frac{channel_0}{channel} \times \frac{sample\ rate_0}{sample\ rate} \times \frac{NR\ Cycles + WDRC\ Cycles}{Total\ Cycles} \tag{5.2}$$

表 5-4 与已发表文献比较

	本文	[4]	[5]
功能	NR，WDRC	波束成形，反馈消除，NR，WDRC	NR，WDRC
通道数	32	17	18 NR，3WDRC
采样率/KHz	16	16	24
时钟/MHz	8	11	3，6，8
实现	ASIP	ASIP	ASIC
工艺/nm	130	65	90
电压/V	1.2	0.8	1.0
面积/mm2	2.0	0.49	3.13
归一化的面积	1	0.98	3.27
功耗/mW	0.963	0.334	1.095
归一化的功耗	1	1.03	1.94

5.3 本章小结

本章介绍了基于 ASIP 的数字助听器系统的功能仿真、FPGA 验证、以及物

理实现，并对芯片实现进行了测试结果分析。经过验证和测试，证明了本文的 ASIP 设计达到了数字助听器应用的要求，能较好的实现数字助听功能，工作在 8MHz 频率、1.2V 工作电压时，处理器功耗约 0.963mW，与同类型设计相比具有一定的优势。同时也证明了本文的 ASIP 设计是可行的，能够满足系统对功耗、面积、以及灵活性的需求，可以作为其他设计的参考。

参考文献

[1]Stratix II EP2S180 DSP Development Board Reference Mannual s[EB/OL]. Altera, 2005.

[2]TMS320AIC23B Data Mannual s[EB/OL]. Texas Instruments Incorporated, 2003.

[3]Winbond W25X10AV SPIFlash Data Sheet s [EB/OL]. Windbond, 2004.

[4]Peng Qiao, Corporaal H, Lindwer M. A 0.964mW digital hearing aid system[C]. //Design, Automation and Test in Europe Conference and Exhibition, Grenoble, France, 2011:1-4.

[5]Cheng-Wen Wei, Yu-Ting Kuo, Kuo-Chiang Chang, et al. A Low-Power Mandarin-Specific Hearing Aid Chip[C].//IEEE Asian Solid-State Circuits Conference. Beijing, China, 2010:1-4.

[6]Baas B M. A low-power, high-performance, 1024-point FFT processor[J]. Solid-State Circuits, IEEE Journal of, 1999, 34(3): 380-387.

第6章 总结与展望

6.1 工作总结

目前大多数嵌入式产品在设计时采用专用指令集处理器，专用指令集处理器的市场已经超过了 ASIC 与通用处理器的总和，因此对于 ASIP 的设计进行研究具有广阔的市场空间与应用价值。ASIP 设计的过程是一个软硬件协同设计的过程，包含了构体系结构与软件开发工具的设计。本文对专用指令集处理器的体系结构与软件开发工具的设计进行了研究，取得了如下成果：

（1）本文研究了专用指令集处理器设计的一般流程，以数字助听器关键算法为应用，提取应用特征，添加专用指令与硬件加速单元，提出了一种面向音频处理的专用指令集处理器。

（2）设计了语音专用指令集处理器的软件开发工具，可用于 ASIP 设计时的应用分析、设计评估，缩短 ASIP 的设计周期，同时极大的方便了基于 ASIP 的软件开发。对于不同的应用，需要指令扩展时，可以快速完成开发工具的修改，生成配套的软件工具。对于软件开发工具的研究也是进一步研究软件开发工具自动生成的基础。

（3）基于提出的音频专用指令集处理器研究了数字助听器的低功耗 ASIP 设计，通过添加专用指令与硬件加速单元，综合运用低功耗的工作模式、门控时钟、操作数隔离、循环缓存、存储器分割等多种低功耗设计技术，有效的提高了程序的执行效率，降低了系统的功耗。

（4）基于 TSMC 130nm 工艺完成设计流片，经过测试，该系统能较好的实现数字助听器功能，工作在 8MHz 频率、1.2V 工作电压时，处理器功耗约 0.963mW。本文工作对于设计其他面向特定应用的 ASIP 系统有着一定的借鉴意义。

6.2 工作展望

本文对于 ASIP 设计的研究工作取得了一些成果，但由于 ASIP 设计过程包

含内容广泛，在已有的工作基础上，有必要进行继续研究：

（1）ASIP 设计过程是一个软硬件协同设计的过程，优化的体系结构设计基于对应用特征的提取，软件开发工具快速生成能够对 ASIP 设计进行评估、提供设计反馈，研究软件开发工具的自动生成可以快速生成软件开发工具。

（2）建立指令集功耗模型，在编译器中添加低功耗的优化策略，以功耗驱动进行指令选择与指令排序，获得低功耗的代码；在仿真器中添加指令集功耗模型，基于仿真过程对设计的功耗进行评估。

（3）基于 LLVM 的 IR 对应用进行代码分析，设计自动的指令扩展识别装置。

（4）编译器的设计是一个复杂繁琐的过程，优化的编译器可以获得高质量的代码，对编译器优化策略进行进一步研究可以获得代码更小、功耗更低、抑或执行效率更高的代码。同时编译器的验证需要大量的测试程序集与验证时间，对编译器进行完整的验证是需要进一步的研究。

（5）本文的 ASIP 设计基于标准单元，采用单电源单时钟，研究多电压与多时钟域设计，根据不同功能模块的需求采用不同的电压与时钟，可以有效降低动态功耗。除此之外，采用电源门控，关闭空闲模块的电源，可以有效的降低系统的静态功耗。

攻读博士学位期间发表的学术论文及申请的发明专利

学术论文：

[1] 薛金勇，黑勇，陈黎明，“快速高效无损图像压缩算法的低功耗硬件实现”，哈尔滨工程大学学报，EI

[2] 薛金勇，黑勇，陈黎明，于增辉，“面向数字助听器的低功耗 ASIP 设计”，微电子学与计算机，中文核心

发明专利：

[1] 薛金勇，黑勇，徐欣锋，陈黎明，Golomb-Rice 编码方法及装置、FELICS 图像压缩方法及系统，申请号：201110103853.4