多目标 Forth 自生成器的研究与实现*

代红兵[1], 杨为民[2], 王丽清[1], 周永录[1]
(1. 云南大学 云南省电子计算中心 云南省高校数字媒体技术重点实验室, 昆明 650223; 2. 西南林业大学, 昆明 650023)

摘 要: 针对现有的 Forth 自生成器都与目标环境密切关联、缺少抽象层次、难以在异构新平台上有效生成新的 Forth 系统等问题,通过采用抽象 code 算法库、描述异构目标,并重构 Forth 虚拟机的方法,构建完成了一个面向嵌入式环境、具有多目标特性的 Forth 自生成器。该生成器简化了传统编译器复杂的前端和后端设计,依托 Forth 特有的解释执行状态和字典结构,可快速生成新的目标系统。实验结果表明,该自生成器代码生成质量和效率都较高,尤其适合资源有限的嵌入式环境。

关键词: 多目标; Forth 自生成器; 嵌入式环境
中图分类号: TN914 **文献标志码**: A **文章编号**: 1001-3695(2014)04-1109-06
doi:10.3969/j.issn.1001-3695.2014.04.037

Research and realization of multi-objective Forth self-generating system

DAI Hong-bing[1], YANG Wei-min[2], WANG Li-qing[1], ZHOU Yong-lu[1]
(1. *Digital Media Technology Key Laboratory of Universities in Yunnan, Yunnan Electronic Computing Center, Yunnan University, Kunming* 650223, *China*; 2. *Southwest forestry University, Kunming* 650023, *China*)

Abstract: Since the available Forth self-generating systems lacked common features for multi-operating environments, they were totally dependent on specific target environments and couldn't efficiently form new Forth system in a new operating environment. Therefore, it was important to design a novel Forth self-generating system that could be easily applied to multiple object-oriented target environments. This study proposed a new model of multi-objective self-generating system with common features for embedding-oriented multi-environments by using common code algorithm library, describing heterogeneous target, and refactoring Forth virtual machine. The model solved the problem of the traditional compiler with complex front-end and back-end design. Based on Forth has unique implementation of interpretation and the dictionary structure, the multi-objective Forth self-generator quickly generated a new target system. The results revealed that this self-generator sufficiently significantly improves quality and efficiency in code generation, especially suitable for embedded environments with limited resources.
Key words: multi-objective; Forth self-generating system; embedded environment

0 引言

与其他计算机语言相比,Forth 语言本身就是一种过程控制语言和一种快速开发环境,具有很强的交互性、构造性、移植性和自扩展能力,参数栈专门独立出来是其一大特色,这使得子程序的调用非常高效。其生成代码非常高效,甚至可以快速构造出一个实时多任务操作系统[1,2]。正是由于这些优良的特性,Forth 语言得到了迅速的推广和使用,从最早的 CCD 天文图像系统,逐步深入拓展到数据采集、过程控制、图像处理、仪器仪表、人工智能、计算技术等各领域,在许多尖端产品的关键部件或系统组成中都能找到 Forth 的身影[3,4]。近年来,基于 Forth 虚拟机思想而设计的 Forth 处理芯片日渐成为当今嵌入式多核微处理器的发展方向,广受业界关注,如 SEAforth 系列单芯 40 核处理器、100 核处理器[5~7]。最近, Forth 语言之父 C H. Moore 在 72 岁高龄之际,宣布推出 144 核的 CPU,运算能力达到 1 000 亿操作/s[8]。除基于 CMOS 的 Forth 专用芯片外,新出现的基于 FPGA 的 Forth 处理器已逐渐演变成为嵌入式系统的另一发展方向[9,10],同时形成了一批以 RTXcore 为代表的基于 FPGA 的 Forth IP 核[11,12]。为让 Forth 得以更好地推广和扩展,分布于世界各地的 Forth 小组先后推出了 Win32Forth、SwiftForth 等平台,可运行于 Windows、Linux 以及 Mac OS 等各种操作系统之上,并持续不断完善[13,14]。但无论是什么形态,Forth 处理器的结构基本没有大的改变,一般由 ALU(堆栈操作)、数据栈、返回栈、堆栈指针、指令指针等部件组成,提供 20 ~ 30 个机器指令(5 ~ 6 bit),其抽象出的高效目标描述对嵌入式多目标 Forth 编译器的研究提供了很好的借鉴[15]。但长期以来,如何高效、快速地构造 Forth 系统一直是 Forth 研究的难点,国内外的专家、学者为此开展了许多卓有成效的工作。其中,Brad Rodriguez 对相关研究[16~18]进行了总结,系统提出了三种构造 Forth 系统的方法[19]。近年来,面向多目标的 Forth 生成器研究出现了一些有价值的最新研究成果[20~22]。

目标 Forth 的编译过程与传统的编译模式有着本质的不同,并不存在严格区分和功能独立的词法分析、语法分析、语义分析、中间代码生成、目标代码生成及代码优化等传统编译过程,其中的任何一个成分也没有形成像 Lex、Yacc 一样的功能

收稿日期: 2013-07-24; **修回日期**: 2013-08-25 **基金项目**: 国家自然科学基金地区科学基金项目(61063010)
作者简介: 代红兵(1963-),男,江西人,正高级工程师,硕士,主要研究方向为嵌入式系统、数字电视技术等(hbdai_it@126.com);杨为民(1955-),男,云南人,教授,主要研究方向为遥感和地理信息系统、嵌入式系统;王丽清(1971-),女,云南人,高级工程师,硕士,主要研究方向为数字媒体技术、计算机应用;周永录(1965-),男,云南人,高级工程师,主要研究方向为嵌入式系统.

部件。Forth 特有的堆栈和字典式结构决定了编译过程的特殊性。在 Forth 自生成器中,以往的编译器前端(front end)结构被宿主和目标字典的搜索完全取代,其对应的后端(back end)也没有传统编译器经常使用的树型模式匹配算法。按照现有自生成器模型,尽管 Forth 自生成器的代码生成非常快捷,但几乎所有构成部件都与目标环境密切关联,缺乏抽象层次,难于复用共享,难以在异构新平台上有效生成新的 Forth 系统,如果要更改一个新的目标环境,所有工作都将推倒重来。本文通过对框架抽象表达、异构目标描述、Forth 虚拟机重构、代码生成过程、编译控制过程等关键问题的研究,提出一种面向嵌入式环境、具有多目标特性的 Forth 自生成器框架模型及实现算法,为解决异构目标平台上 Forth 系统的快速构建提供理论和方法支撑。

1 多目标 Forth 自生成器框架模型

多目标 Forth 自生成器框架由目标系统模型、应用系统模型、虚拟机模型、Meta 编译器、code 算法库和可重构的编译程序组成,如图 1 所示。

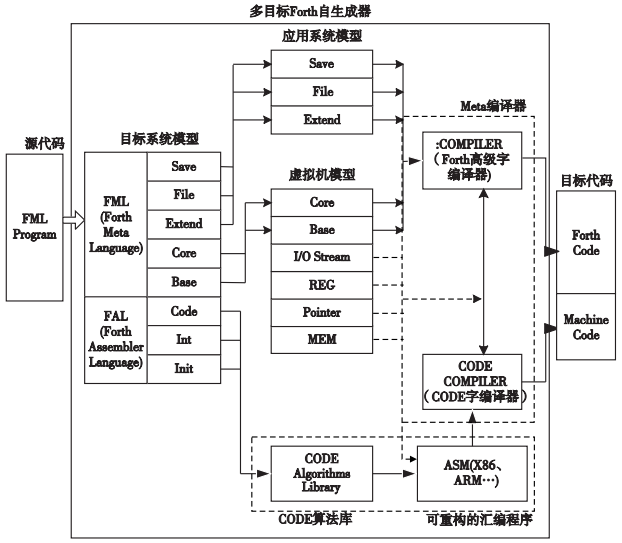

图1 多目标Forth自生成器框架

要生成新的目标 Forth 系统,按照自生成器框架提供的目标系统模型,用其提供的 Forth 元语言(Forth meta language, FML)编写 Forth 源代码程序,经自生成器编译后,生成新的目标代码。

1.1 CODE 算法库

CODE 算法库是针对目标处理器的 CODE 字预定义。对 Forth 系统来说,CODE 字实际就是一种目标系统的中间描述或抽象表达,因此用 FML 编写的 Forth 程序就是一个与任何目标平台无关的可移植软件。

1.2 应用系统模型

应用系统模型是目标系统模型针对具体应用的字描述,包括扩展字(Extend)描述显示工作状态和调试手段的 Forth 字,文件系统(File)描述外存设备文件管理和外存数据存取,保存部分(Save)保存目标系统映像。

1.3 目标系统模型

从逻辑看,目标系统模型涵盖了 CODE 算法库、虚拟机模型和应用系统模型。一个完整的 Forth 目标系统模型由以下部分组成:

a)初始化(Init)。它是新的目标系统开始执行的引导部分,主要是完成对整个系统进行初始化。对于作为应用程序的 Forth 目标系统,整个程序会被操作系统正确地加载各个内存区域,因此初始化的工作较少,主要是完成各种指针和系统变量的初始化。但是如果目标系统是被 BIOS 从磁盘上或者从 Flash BOOT 区加载的,系统板通常只加载一段很小的引导 BOOT 代码,因此该部分还要包括将自己整个系统从磁盘、Flash 或者 SD 卡等外存设备正确加载到内存,也要自己完成 CPU、MMU 和各种外设的初始化,如果有必要,还要完成对整个硬件系统的检查,这种自引导的 Forth 系统初始化工作就很多。初始化部分的代码运用 Forth 汇编语言(Forth assembler language,FAL)编写,因为这时虚拟机尚未开始工作,初始化部分最后是初始化虚拟机的指针和状态,然后跳到指定的 Forth 字开始执行,虚拟机开始工作。初始化部分是目标系统的必须组成部分,其中只有启动方式、系统检查和外设初始化的少量代码可以裁剪。

b)中断码部分(Int)。它是新的目标系统的中断处理程序。如果新目标系统是一个应用程序,这个部分的内容是如何访问系统的中断。如果新系统是一个独立引导的操作系统,则全部中断处理程序得自己写;如果新系统是一个控制程序,那么有些中断处理是访问系统的,有些中断处理是自己写的,有些中断处理是挂钩在系统的中断上。

c)汇编字(Code)。它包括了大部分要用机器码实现的 Forth 字的机器指令,如算术操作的 Forth 字“+、-、*、/”的实现,算术堆栈操作的 Forth 字“Drop、Dup、Swap、Rot”的实现,返回堆栈操作的 Forth 字“>R、R>、R@”的实现,以及各种关系运算、逻辑运算 Forth 字的实现,包括各个段内存存取 Forth 字的实现等。Forth 虚拟机的字指令是需要物理机的机器码来实现的,在多目标 Forth 自生成器框架中,为对具体的目标处理器进行抽像,其实现是通过可裁剪 CODE 算法库来完成的。

d)基本字(Base)。它是在新的目标系统中提供 Forth 语言编程能力基本 Forth 字,包括常数和变量的定义字,包括冒号定义和其他一些建立新 Forth 字的方法,包括 Forth 字典的维护等。最重要的是包括实现各种结构化编程的 Forth 字。由于 Forth 是一个扩展性极强的语言,尤其是使用“<Build”和“Does>”这样的构造字后,Forth 不仅可以构造新类型的数据结构,还可以构造新语法,因此这部分也是可以裁剪的。

e)核心字(Core)。它包括实现 Forth 虚拟机、Forth 语言解释器、主循环和输入/输出流控制的 Forth 字,以及多任务分配和热启动处理的 Forth 字。由于 Forth 是一个自扩展系统,因此为了实现扩展,这个部分的许多关键字都采用矢量字的形式。例如,输入流可以通过重定向功能从键盘、串口、磁盘和网络输入 Forth 程序,输出流可以通过重定向功能将信息送到显示器、串口、磁盘和网络。同样目标系统的主循环、解释器和多任务分配都应该是矢量字,以备将来升级的需要。这个部分是目标系统的核心,它包括的核心字互相关联,支持着 Forth 系统的工作。但是如何实现这些功能,不同的程序员有不同的方式。例如,在同一个目标系统要求下,对于 X86 或 51 单片机,就有 N 个的版本 Forth,其主要差别就在这个部分。

f)扩展字(Extend)。它是为新目标系统增加一些显示工作状态和调试手段的 Forth 字。比如显示堆栈内容、内存实际分配情况、内存信息、端口信息、系统的工作状态和反编译已有的 Forth 字。这个部分与应用需求有关,因此可以进行裁剪或

者全部省略。

g)文件系统(File)是为新的目标系统增加外存设备文件管理和外存数据存取功能。对于从 Flash 引导起来的嵌入式系统,可以全部省略,即使要扩展也可以通过预留的矢量字来实现。但是对于目标系统是应用程序和磁盘操作系统的情况,则这个部分是必须的。

h)保存部分(Save)将已经完成的系统的各种指针赋以正确的值,使得系统启动时能够正确工作,然后将目标系统映像以合适格式保存到磁盘文件中。

1.4　虚拟机模型

虚拟机模型是 Forth 虚拟机与目标物理机的对应关系,包含内存分配、指针定义、寄存器分配、输入/输出流的定义,以及之前描述过的基本字(Base)和核心字(Core)。由于不同目标机应用的 CPU 字长不同、寄存器数量不同、被允许内存总量不同,以及新 Forth 系统是应用软件还是独立的操作系统的类型不同,因此需要建立起一个模型,才能生成新的目标系统。对于多目标 Forth 自生成器,虚拟机模型可以采用共性的模型,也可以是专门设计的模型。目前 Forth 一般是作为操作系统下的应用程序调入 RAM 内存中运行,各种代码、字典和数据混杂在同一个 RAM 区域中,很难实现内存管理和多任务。Forth 的程序和数据有四类:a)CODE 字(本机码)是二进制的目标机器码程序;b)高级字是 Forth 虚拟机的二进制“机器码”;c)用户数据和虚拟机的堆栈数据;d)用于中断和任务切换的堆栈数据。为此,在 Forth 虚拟机模型中,把 Forth 内存分为代码段、数据段、虚拟段、堆栈段四个逻辑上独立的段。

1)代码段 CS　用于存放机器指令和与硬件有关的系统数据。

2)数据段 DS　用于存放用户数据和虚拟机的堆栈等数据。现有的 Forth 一般是单用户单任务的版本,其为了方便,直接将执行中断的系统堆栈作为计算用的数据堆栈用同一个 CPU 堆栈(直接使用 CPU 的 SP 堆栈指针寄存器)实现,这对于中断处理和实现多任务带来不便或者存在风险。因此虚拟机模型单独设立了算术堆栈空间和虚拟指令堆栈空间,并且不使用 SP 为算术堆栈指针。

3)虚拟段 VS　用于存放高级字(虚拟机指令)、字典和目录。Forth 的核心是字指令,完成虚拟机的一个特定操作,通常用一个 16 bit 的地址码(穿线码)作为虚拟机指令来表示,这个地址码就像汇编语言的 CALL 指令所带的目标地址一样,指向该字在虚拟机字典中的位置。把 Forth 字的名称与地址码对应成为目录。目录对于把高级计算机语言编译为机器码是必要的,当输入新的命令流后,Forth 就会检索这个目录,找到每个字对应的虚拟机指令地址,然后去执行它。

4)堆栈段 SS　用于存放系统堆栈和多任务系统数据等。这个段一般使用系统 CPU 的堆栈指针 SP 来访问,在中断和任务切换时保存现场、保存各种任务表等。这个段只应该由操作系统访问,配合新一代单片机的内存管理单元 MMU 就可以保证大型嵌入式系统工作的安全性。

这四个段的具体大小在进行自生成时根据目标机的配置和要完成的任务再进行定义。例如,对于 8086 的实模式,简单地令每个段为 64 KB,直接用 CS、DS、ES 和 SS 四个段寄存器来控制访问。

作为物理机有两个使用最频繁的指针,即程序指针 PC 和堆栈指针 SP。这在 CPU 核中有专门的寄存器来实现,而且有一系列的专门操作机器指令。同样 Forth 虚拟机有许多专门的指针,这些指针在虚拟机工作时,其使用频率大不相同,虽然这些指针都可以用一个内存指针变量来实现,但是将最常用的虚拟机指针用 CPU 中的寄存器来实现,将大大提高虚拟机执行速度,只是做指针用的寄存器必须能进行寄存器间接寻址。

Forth 虚拟机运行时使用频率最高的指针是指令指针、字指针、返回栈指针、算术堆栈指针。指令指针 IP 相当于物理机的 PC 指针,指向下一条要执行的虚拟机指令(地址码)。字指针 WP 指向当前执行 Forth 字的首地址,通常它指向一段物理机代码,这段机器指令代码负责实现虚拟机的虚拟指令。返回栈指针 RP 相当于物理机的 SP 指针。虚拟机返回堆栈的功能与物理机堆栈的使用十分相似,例如在执行虚拟机字指令时就像物理机访问一个新过程一样要把当前的 IP 推入返回栈以便字指令完成返回后恢复 IP,正确执行下一条虚拟机指令。算术堆栈指针 AP 用于 Forth 的各种运算。由于 Forth 虚拟机不需要指定被操作数,所有的操作都是针对算术堆栈顶部的数据,所以 Forth 虚拟机指令是定长度的地址码,这个地址码就是存放该 Forth 字的内存地址。

显然这些指针最好用寄存器来实现,以提高虚拟机的速度。例如,在 X86 的 CPU 上,可使用 SI、BX、BP 和 DI 作为 IP、WP、RP 和 AP 指针寄存器,正好这些都支持寄存器间接寻址,便于实现堆栈操作。对于 X86 和 ARM 类有三个以上支持间接寻址寄存器的 CPU 核目标机,都可以采用这个模型将四个指针都用 CPU 寄存器实现。但增强 51 类 CPU 只有一个可以在整个地址空间间接寻址的寄存器 DPTR,所有的指针只能作为内存变量,在使用时装入 DPTR 寄存器中。如果 CPU 有两个可以在整个地址空间间接寻址的寄存器,则将字指针 WP 用寄存器实现,这有利于用物理机指令实现虚拟机指令,而另外三个指针共用另一个寄存器。如果 CPU 有三个可以在整个地址空间间接寻址的寄存器,则将字指针 WP 和 AP 用寄存器实现,另外两个指针共用另一个寄存器。这是因为算术堆栈是一个任务的全程堆栈,执行虚拟机指令不需要改变 AP 的值,这样就减少了频繁切换寄存器的操作。

虚拟机到物理机的转换算法:首先对每个虚拟指令(Forth 字)设置一个代码场(CFA),在其中存放实现该虚拟指令的机器码的开始地址,在执行虚拟指令时字指针 WP 已经指向 CFA 了,这时只要执行指针间接寻址跳转机器码指令[WP] JMP,就可以将控制权交给物理机执行。与之相反,物理机到虚拟机的转换算法是 Forth 的 NEXT 过程。

1.5　可重构的汇编程序

为快速开发新的目标系统,就需要对目标硬件进行抽象表达,对现有的 Forth 汇编程序过程进行算法改进,构造出可适应不同目标 CPU、可快速重构的汇编程序。为避开传统多目标编译器复杂的词法分析、语法分析、语义分析、代码生成等过程,可重构的汇编程序充分利用了 Forth 解释执行的特性,每条指令助记符、寄存器、寻址方式等汇编语句元素都被描述定义为 ASSEMBLER 字典中的一个 Forth 字或常数,因此,编译过程就成为这些定义的执行过程。由此可见,这种方式并不存在传统编译器复杂的前端,其后端也不需要使用传统的树型模式匹配算法。

1.6　Meta 编译器

Meta 编译器是一个用 Forth 语言编写的交叉编译器,在宿

主机上运行后,对目标系统的源文件进行编译,生成目标系统的机器码和 Forth 虚拟机码。Meta 编译器由:-COMPILER(FOTH 高级字编译器)和 CODE-COMPILER(CODE 字编译器)两个关键核心过程组成。Forth 采用逆波兰和字为基本元素,这使得 Forth 编译器工作原理和算法清晰易懂,用 Forth 实现的方法简单巧妙,这正是 Forth 语言的迷人之处。其实,像 GCC 等 C 语言的编译器都是将原始的高级语言转换为 RTL 的一种内部的中间语言,这种转换过程由于语法的原因非常复杂,甚至要多遍扫描,而转换后 RTL 中间语言恰巧就是逆波兰和字单元的。由于 Forth 语言的优点,实现 Meta 编译器的算法比较简明,其有宿主机运行态和目标机编译态两种工作状态。采用 FML 编写的源文件中的 Forth 字也有作为编译命令的宿主机 Forth 字和新目标系统的 Forth 字两种类型。像所有的编译器一样,Meta 编译器在编译过程中维护宿主机 Forth 和目标机 Forth 两个字典。依靠这两个字典对源文件的每一个字进行处理,最终编译成为目标代码。

Forth 编译器工作流程是一个循环:从源文件输入流取出下一个字,然后在宿主机字典中查找该字并执行该字。作为 Forth 语言的特点,正在执行的宿主机 Forth 字可以不停地用“WORD”这个 Forth 字来获取源文件中的下一个字进行处理,如查找这个新字是否在目标机或者宿主机字典中存在,然后作出相应的处理。这个编译循环一直进行到该文件结束,或者碰到一个终止命令为止。

2 多目标 Forth 自生成器的关键实现算法

在关键算法中,为了不失一般性,这些算法没有刻意地突出不同目标硬件之间的差异,只是用不同的指针来说明算法的原理。

2.1 目标代码存取算法

由于一般 Forth 目标系统的核心规模较小,16 bit 的系统不超过 64 KB,32 bit 的系统也不超过 256 KB,因此可以直接在宿主机 Forth 系统管理的内存中划出一块高内存区域。如 P! 和 P@ 是原内存的写入和读取字,目标代码区域的起点是 XXXX,则目标代码的存取字可以简单定义为下面的算法:

```
XXXX CONSTANT T-Origin
// 定义常数 T-Origin,它的值是目标代码区域的起点
: T-! T-Origin + P! ;: T-@ T-Origin + P@ ;
// 定义目标区的数据存取字
```

同样也可以产生按字节的存取字 T-C! 和 T-C@ 。

2.2 机器码写入算法

汇编语言产生的目标机机器码是顺序存放的,其中许多跳转指令也是以代码当前位置参考计算的,因此,设置一个当前代码地址指针 CP 就可以实现:

```
Variable CP;T-Here  CP @ ;
//T-Here 字给出下一个机器码存放的地址
:T-C,T-Here T-C! CP @ 1 + CP ! ;
//放 1 Byte 的机器码到目标区
:T-,T-Here T-! CP @ 2 + CP ! ;
//放一个 16 bit 字的机器码到目标区
```

有了这些 Forth 字,可以简单到无须助记符和汇编,直接把二进制代码一个字节一个字节地生成目标码。

2.3 向前跳转地址算法

机器码中最重要的指令就是根据某种条件进行转移的指令。由于 Forth 语言的工作过程是按照输入流一个字一个字解释执行的,因此向前转移时目的地址是已知的,实现起来比较简单:以目标地址为值,定义一个常数,然后在生成转移指令时引用它。

```
: T-L:T-Here Constant;
// 以当前地址定义一个常数
```

配合前面的 JMP 汇编指令,向前跳转的汇编语言规范就可以实现,如:

```
T-L: L_11 … 一些代码 … L_111 JMP
```

显然,向前跳转的标号可以多次引用,根据条件跳转到同一个标号。

2.4 向后跳转地址算法

算法原理是定义一个 Forth 字,在跳转地址处先填一个数,记下该地址:

```
: T-LB: <Build T-Here +1 ( 这将是 JMP 指令中存放目的地址的地址 ) , 0 Does > @ T-Here SWAP T-! ;
```

其中:<Build 的功能是定义一个新的 Forth 字,然后把地址保存到新字里,最后在堆栈中留了一个数 0,让 JMP 指令先存放进去。在后面到了要跳转的目的地址,执行这个新的字时,宿主机 Forth 从 Does > 后面开始执行,首先取出前面保存的指令中的跳转地址,然后得到现在要跳转的地址,再把这两个地址交换一下,把目的地址正确地存放到机器码指令中。算法例子如下:

```
T-LB: L_222 JMP … 一些代码 … L_222
```

当然如果要考虑一个地址有多个向后跳转时,算法要更复杂一些,但其原理是一样的。

2.5 定义目标机 Forth 字结构算法

一个 Forth 字依次包括以下部分:链接场(LFA)、名字场(NFA)、代码场(CFA)和参数场(PFA)。除了参数场其余部分都是一个整数字的长度(16 或者 32 bit),括号里是每个字段地址的名字。链接场将一个个 Forth 链接形成链表结构。名字场存放该 Forth 字的名字,Forth 在解释执行 Forth 字时自链尾向链头进行搜索,找到就执行它。代码场存放实现该虚拟机指令机器码的开始地址。参数场存放 Forth 字的内容:数据、机器码或者组成它的基础 Forth 字。给一个名字在目标区域产生一个字的头两个部分的编译命令为 T- < Build,可以用产生链表的算法实现。

2.6 定义目标机常数和变量算法

由于编译命令也是 Forth 字,当然也可以从已有的编译命令中产生出新的编译命令,这是 Forth 自生成编译器的一个特点。例如在目标系统中定义一个常数的算法是:

```
: T-Constant T- < Build T-[ Const ]  T-, T-, ;
```

其中:T-[Const]是一个常数,指向实现常数虚拟字的机器码的开始地址。显然这与定义普通变量方法类似。实现在目标系统中定义一个用户变量的算法为:

```
:T-Uvariable T-UDP T-@ Dup T-Constant 2 + T-UDP T-!;
```

其中:T-UDP 是用户变量指针。

2.7 CODE-COMPILER 核心算法

可重构的 Forth 汇编程序既要适合 CISC,也要经重构后面向 RISC。从汇编程序的实现看,可变长 CISC 的编译过程要比固定指令长度的 RISC 复杂得多。通过对多种嵌入式微处理器指令系统的分析,可以提炼得出一套通用的 Forth 汇编算法,这正是实现可重构 Forth 汇编程序的基础,实现多目标实质就是对目标硬件进行重新描述。重构描述如表 1 所示。

表 1　重定义描述

描述项目	内容说明	描述方式	示例(X86)
存取操作	编译指针、存取操作	定义 Forth 字	ADP、AB!、AB, …
寄存器	寄存器编码	定义常数	C3 CONSTANT BX …
条件码	条件码编码	定义常数	F CONSTANT GT, …
寻址方式	寻址方式编码	定义常数	40 CONSTANT +B) …
语法描述	合法性检查	定义 Forth 字	? S#、? D#、? REG …
编译类型	操作数寻址方式类型	定义 Forth 字	00TY、11TY、21TY …
编译控制	控制代码生成过程	定义常数	00h-FFh
指令描述	汇编语言助记符及编码	建立助记符与指令码表	00 C7 89 21TY MOV …
代码生成	机器码组合/合成	定义 Forth 字 00TY…DOES > B@ AB, ; …	

对不同的目标平台,除上述特定描述之外,可重构的 Forth 汇编程序可共享的部件或内容包括缓冲区管理、基本操作、编码规则、核心过程等。其中,存取操作描述并未使用宿主 Forth 的字典指针 DP 及编译字“,”而是基于虚拟编译指针 AVDP 构造出可寻址整个存储空间的全局编译字 A,和 AB,。与其他描述相比,编译类型描述较为复杂,其规则是按所有指令操作数寻址方式进行划分,将能一并处理的指令合并为一个编译类型。以 X86 为例,单操作数有 8 种,双操作数有 4 种,无操作数有 1 种,共计 13 种类型。以 X86 的寄存器的操作数寻址方式为例(适用于 ADD、SUB、MOV、OR、XOR、CMP、TEST 等指令),其对应的编译类型描述示例为:

```
:21TY CREATE B, B, B, DOES >
  A[2   //将操作数或编码送入操作数缓冲区
  ? D# 3 AERR OR ? DSR 4 AERROR  ? SSR 4 AERROR
//语法检查
? S#   //源操作数是立即数?
  IF  2 @1OP S2OP. ADR B! ? #BIT 0 ? OFFBIT -1 1
//产生编译控制码
  ELSE  ? DREG
    IF 0 @1OP FE AND 84 〈〉
      IF 02 [ +OP] B!
      ENDIF [S]〈〉[D]
    ENDIF 0 0 ? OFFBIT -1 0
ENDIF ]A,  //代码生成
;
```

以# REG MOV 指令为例,MOV 的指令描述为 00 C7 89 21TY MOV。当可重定义的 Forth 汇编程序被装载执行时,21TY 定义中的 CREATE 创建了 MOV 字。当 T-CODE 定义中出现 MOV 时,21TY 定义中 DOES〉之后的语句被执行,即完成 MOV 指令的编译。由此可见,与其他汇编程序不同,实现可重定义的 Forth 汇编程序的关键在于利用了 Forth 的解释执行状态来完成编译过程。

在 Forth 的解释执行状态下,整个编译过程(一次性解释扫描编译算法)分为两个步骤:a)扫描堆栈中的指令操作数,结果送入操作数缓冲区;b)依据操作数缓冲区的信息,提取编译控制码,完成指令码的组合编译。T-CODE 编译算法如图 2 所示。

2.8　:-COMPILER 核心算法

如同冒号字是 Forth 的核心字,它可以把已有的 Forth 字组合起来生成新的 Forth 字一样,编译命令 T-:也是自生成编译器的核心命令,通过它可以把已经编译产生在目标系统中的 Forth 字组合成为一个新的目标系统 Forth 字。与 T-:配对的编译命令 T-;,它结束 T-:编译命令。实现核心算法 T-:的方法不是唯一的,本文提出一种特别符合 Forth 风格的一次性解释扫描编译算法,如图 3 所示。

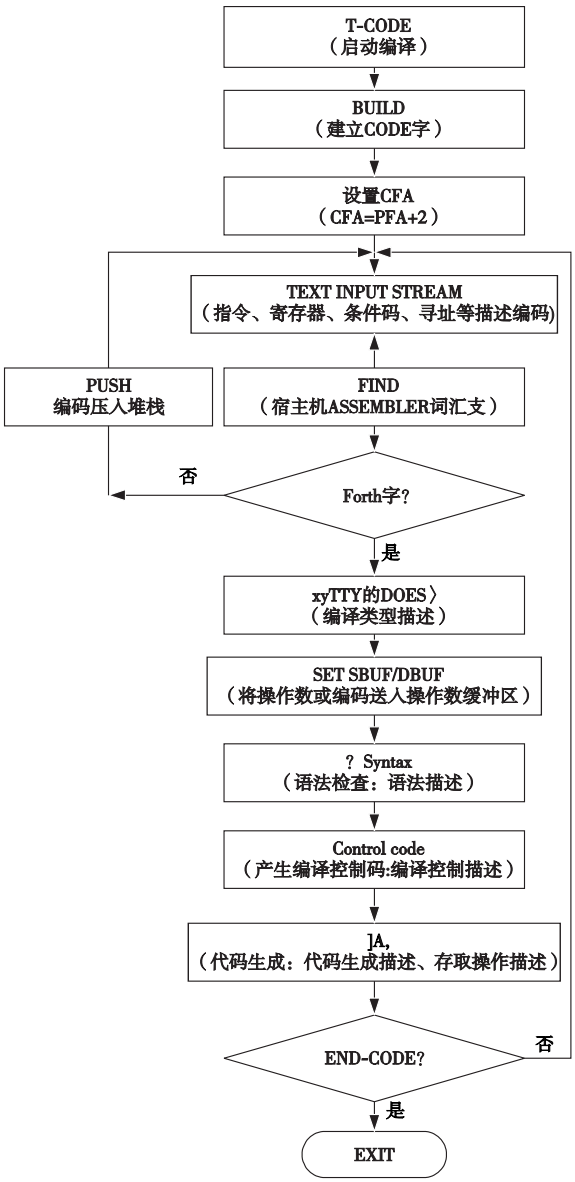

图2　T-CODE编译算法

3　实验评估

由于 Forth 是一个开放性的语言,所以对于应该有哪些编译命令,这些编译命令应该如何实现,甚至编译命令的名字应该取什么都没有统一要求,只有一些参考标准。所以,如何实现一个高效率和高速度的算法,就靠开发人员自己去发挥了。作为参考,笔者在现在的 2G 4 核性能的 PC 机上对 100 KB 的 Forth 源文件进行了自生成,整个编译过程不超过 3 s。

表 2 给出了利用本文的自生成器模型进行实际两个目标生成测试的结果,其中 X86 是针对 PC 架构嵌入式系统的 DOS 运行环境生成的新的单任务 Forth 系统,X51 是针对 C8051F020 嵌入式单片机开发板硬件环境生成的新的单任务 Forth 系统。首先,由于两者都没有自己的文件系统,且是单任务的,因此堆栈段长度 X86 是指定的,而 X51 是硬件决定的。其次,对于 X51,由于其目标模型中不需要关于文件操作等扩展功能(也就没有相应的文件缓冲区等数据),所以它的数据段和虚拟段长度明显少于 X86。最后,由于 X51 是 8 bit 的 CPU 核,实现 16 bit 操作的机器码要比 X86 长许多,但在具体实现 X51 汇编算法时采用了减小代码总长度优先的标准,首先编写了大量的公共子程序,因此比起 X86 来,X51 代码段增加的长度不多。

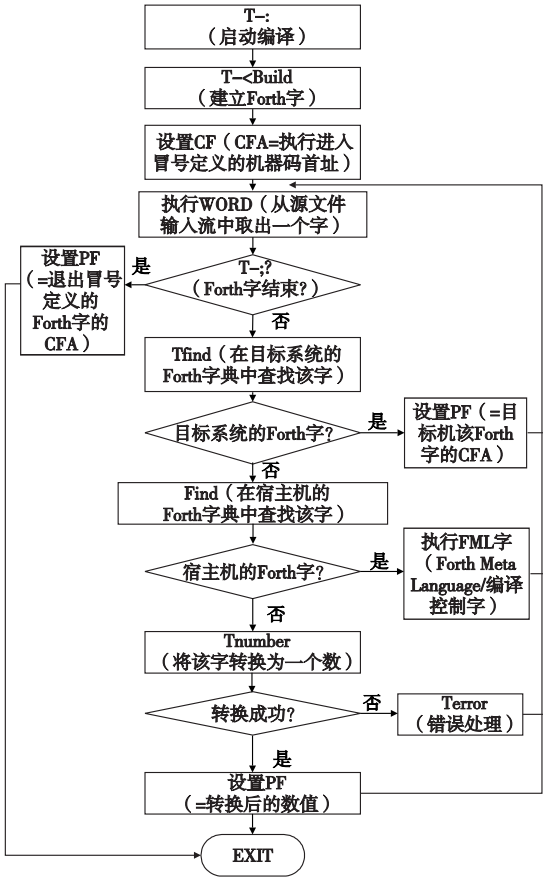

图3 T–:编译算法

表 2 Forth 自生成器性能测试

指标	单位	X86	X51
目标机字长		16 bit	16 bit
目标模型代码 FML 源文件长度	字符数	41 598	40 115
目标模型代码 FML 源文件字数	字数	6 588	5 148
目标模型代码 FAL 源文件长度	字符数	55 371	63 470
目标模型代码 FAL 源文件字数	字数	9 773	11 380
编译软件平台		Win32Forth	Win32Forth
编译硬件平台	CPU	2G 4 核	2G 4 核
编译时间	s	2.2	2.8
目标代码段长度	Byte	3 521	4 360
目标数据段长度	Byte	9 452	4 824
目标虚拟段长度	Byte	7 512	4 322
目标堆栈段长度	Byte	2 048	128

作为描述目标系统模型的源代码，FML 虚拟机部分差别不大，但是物理机实现部分两者有较大的差异，使得 X51 的 FAL 源文件在字符数和字数方面相差较大。由于字是 Forth 元语言的基本元素，字数的比较更具有意义。

4 结束语

本文通过对 Forth 多目标编译理论和技术中关键问题的研究，提出一种面向嵌入式环境、具有多目标特性的 Forth 自生成器框架模型及实现算法，并通过实际构建多目标自生成器对其进行验证和评估，为嵌入式领域的应用开发提供一种高效率、可重构、可组装、规范化、可移植的 Forth 语言编译及系统生成方案，支持从核心 Forth、基本 Forth 的实现，直到 Forth 操作系统，乃至 Forth 专用系统的实现，以满足嵌入式领域对 Forth 技术的应用需求，推动 Forth 技术和编译技术的创新发展，为嵌入式 Forth 系统的大规模应用提供必要的理论支撑和实现参考。

参考文献：

[1] 代红兵. 新型、高效微机 Forth 语言的研制[J]. 中国科学院研究生院学报，1993，10(1)：62-69.

[2] JinXfei. Forth 语言[EB/OL]. (2010-07-15)[2013-07-16]. http://blog.csdn.net/jinxfei/article/details/5738494.

[3] Forth Inc. Featured forth applications[EB/OL]. (2009-01-01)[2013-07-16]. http://www.forth.com/resources/app Notes.

[4] JAMES R. Space-related applications of forth[EB/OL]. (2006-09)[2013-07-16]. http://forth.gsfc.nasa.gov/.

[5] GUZEMAN D G. A 21st century sea change taking place in embedded micro processors[EB/OL]. (2006-12)[2013-07-16]. http://www.intellasys.net/templates/trial/content/WPEmb Micro.pdf.

[6] IntellaSys, A TPL Group Enterprise. SEAforth 40C18 scalable embedded array processor[EB/OL]. (2008-01-01)[2013-04-26]. http://www.intellasys.net/templates/trial/content/SEK_40C18_DataSheet_1.1.pdf.

[7] LaFOREST C E. Second-generation stack computer architecture[EB/OL]. (2007-08-01)[2013-02-10]. http://is.uwaterloo.ca/Eric_LaForest_Thesis.pdf.

[8] CSDN. Forth 语言之父的神品[EB/OL]. (2010-02-07)[2013-07-16]. http://soft.zdnet.com.cn/software_zone/2010/0207/1627096.shtml.

[9] HJRTLAND E, LI Chen. EP32-a 32-bit Forth micorprocessor[C]// Proc of Canadian Conference on Electrical and Computer Engineering. 2007：518-521.

[10] HASKELL R E, HANNA D M. Implementing a forth engine microcontroller on a Xilinx FPGA[J]. The IEEE Computer Society's Student Newsletter, 2000, 8：32-36.

[11] MicroProcessor Engineering. RTXcore[EB/OL]. (2006-09-25)[2012-04-10]. http://www.Mpeforth.com/arena/rtxcore_brief.pdf.

[12] HANNA D M, HASKELL R E. Implementing software programs in FPGAs using flowpaths[C]//Proc of International Conference on Embedded Systems and Applications. 2004：76-82.

[13] Forth Develop Group. SwiftForth integrated development environment. [EB/OL]. (2013-07-21)[2013-04-10]. http://www.forth.com/swiftforth/index.html.

[14] Win32 Forth Group. Introduce[EB/OL]. (2013-01-01)[2013-07-16]. http://tech.groups.yahoo.com/group/win32forth/.

[15] HARRIS A J, HAYES J R. Functional programming on a stack-based embedded processor[C]//Proc of the 2nd IEEE International Conference on Space Mission Challenges for Information Technology. Washington DC：IEEE Computer Society, 2006：418-424.

[16] 代红兵. Forth 自生成器技术[J]. 中国科学院研究生院学报，1992，9(4)：391-398.

[17] Taygeta Scientific Inc. forth compilers on taygeta[EB/OL]. (2002-10-08)[2013-07-16]. http://www.tayge ta.com/forthcomp.html.

[18] MENTINK B. graspForth--a 32-bit Forth[EB/OL]. (2004-02)[2013-06]. ftp://ftp.taygeta.com/pub/Forth/Compilers/native/misc/graspForth/graspforthreadme.

[19] RODRIGUEZ B. Moving Forth[J]. The Computer Journal, 1993, 59(1)：27-32.

[20] ERTL M A. Ways to reduce the stack depth[C]//Proc of the 27th EuroForth Conference. 2011.

[21] STODDART B, RITCHIE C, DUNNE S. Forth semantics for compiler veri_cation[EB/OL]. (2012-09-13)[2013-07-1]. http://www.complang.tuwien.ac.at/anton/euroforth/ef12/papers/stoddart.pdf.

[22] SPRINGE W S. Connection of a Forth target with a Forth host[EB/OL]. (2012-09-12)[2013-07-10]. http://www.complang.tuwien.ac.at/anton/euroforth/ef12/papers/stricker.pdf.