

堆栈处理器代码生成器的设计与实现

赵小东,施慧彬

(南京航空航天大学 计算机科学与技术学院,江苏 南京 210016)

摘要:堆栈处理器是一种面向嵌入式控制领域的处理器,其执行过程不依赖于通用寄存器而是硬件堆栈。文中以一款基于 FPGA 设计的 16 位堆栈处理器为平台,研究如何将一个 C 源代码转换成能够被目标处理器汇编器识别的汇编指令。为了能够迅速有效地实现代码生成器,文中选用 LCC。LCC 是一款可变目标的 C 编译器,通过重新书写目标处理器的描述文件,LCC 可以生成特定处理器的汇编指令。文章的最后通过一个简单的测试证明了 C 语言是可以移植到堆栈处理器上的。

关键词:堆栈处理器;代码生成器;LCC

中图分类号:TP314

文献标识码:A

文章编号:1673-629X(2013)04-0163-05

doi:10.3969/j.issn.1673-629X.2013.04.040

Design and Implementation of Code Generator for Stack Processor

ZHAO Xiao-dong,SHI Hui-bin

(College of Computer Science and Technology,Nanjing University of Aeronautics & Astronautics,
Nanjing 210016,China)

Abstract:Stack processor is a processor facing to the field of embedded control,and its implementation process does not depend on general register but hardware stack. Take the 16 stack processor based on FPGA as the platform,research how a C source code can be converted into assembly instruction by target processor assembler recognition. In order to quickly and effectively achieve the code generator,choose LCC. LCC is one variable target C compiler,through rewriting the description file of the target processor,LCC can generate a particular processor assembly instruction. Finally through a simple test proved that C language can be transplanted to stack on the processor.

Key words:stack processor;code generator;LCC

0 引言

文中针对的目标处理器是一个基于 FPGA 开发的 16 位堆栈处理器。该处理器的相关信息可以参考文献[1]。使用的工具是 LCC 和 LBURG。有关 LCC 的详细信息可以参考文献[2~4]。LBURG 是基于 BURS^[5~8]理论设计出来的一款代码生成器的自动生成工具。该工具能够将目标机器的描述文件自动转换成 C 文件,是 LCC 能够实现可变目标的主要支撑工具。为了能够将 LCC 进行重定向,需要重新书写机器描述文件 STACK16.md。

1 设计与实现

1.1 终结符与非终结符

利用 LBURG 工具生成代码生成器,第一步工作就

是根据目标处理器的指令集和 LCC 中间代码格式设计出符合 LBURG 规范的终结符和非终结符。目标处理器的指令集和 LCC 中间代码格式分别在文献[1]和文献[2]有详细的描述。表 1 归纳了所有用到的非终结符。

表 1 非终结符

名字	匹配对象
acon	地址常量
addr	针对内存读写指令的地址计算
addrj	针对跳转指令的地址计算
con	常量
con1	整型常量 1
reg	计算结果在数据堆栈 T 中的运算
stmt	匹配根

终结符就是 LCC 中间代码 DAG 节点对应的整数表示。对于所有可能产生的 DAG 节点,LCC 都使用一个整数与之相对应。文献[9]总结了 LCC 4.x 支持的所有节点操作符。表 2 归纳一些常见的操作符及其类型后缀、类型大小和操作。

收稿日期:2012-08-08;修回日期:2012-11-13

基金项目:南京航空航天大学引进人才科研启动基金(S1028-042)

作者简介:赵小东(1980-),男,硕士研究生,研究方向为计算机系统结构;施慧彬,博士,副教授,研究方向为计算机系统结构。

表 2 LCC 支持的节点操作符

操作符	类型后缀	类型大小	操作
ADDRF	P	p	参数地址
ADDRG	P	p	全局地址
ADDRL	P	p	局部地址
CNST	FIUP	fdx csilh csilh p	常量
INDIR	FIUPB	fdx csilh csilh p	取
ASGN	FIUPB	fdx csilh csilh p	赋值
ARG	FIUPB	fdx ilh ilh p	实际参数
CALL	FIUPVB	fdx ilh ilh p	函数调用
RET	FIUPV	fdx ilh ilh p	从函数返回
JUMP	V		无条件转移

对于表 2,这里作一个解释。首先,类型后缀中:I 代表 INT, P 代表 POINT, F 代表 FLOAT, B 代表 STRUCT, U 代表 UNSIGNED, V 代表 VOID。同时, LCC 规定:I 等于 5, P 等于 7, F 等于 1, B 等于 9, U 等于 6, V 等于 8。其次,类型大小用 csilhf d xp 这 9 个小写字母表示:c 代表 char, s 代表 short, i 代表 int, l 代表 long, h 代表 longlong, f 代表 float, d 代表 double, x 代表 double long, p 代表 point。这 9 个字母对应的整数由目标处理器决定。

节点操作符+类型后缀+类型的大小共同组合成 LCC 中间代码的 DAG 节点。比如 ASGNi1 表示一个针对单字节整型变量赋值的操作,对应于 C 代码就是对用 unsigned char 修饰的变量进行赋值。对于 32 位的目标机器而言,整型数据的赋值有 3 种,分别是 AS-GNi1、ASGNi2、ASGNi4。其中,ASGNi1 代表 8 位整型数据的赋值,ASGNi2 代表 16 位整型数据的赋值,AS-GNi4 代表 32 位整型数据的赋值。

最新版本的 LCC(LCC4.2)是这样计算 ASGNi1 对应的整数:首先,将 ASGN 定义成用整数 3 左移 4 位得到的整数,即 $3 * 2^4$, 等于 48;然后,将 I 定义为 5;接着,定义类型的大小。对于 32 位的目标机器而言, c 等于 1, s 等于 2, i 等于 4, l 等于 4。这里, ASGNi1 的类型大小就等于 1。最后,应用 ASGN+I+类型大小<<10 这个计算公式得出 ASGNi1 节点对应的整数,即 $48+5+1024$, 等于 1077。于是,在 LBURG 规范中定义终结符的部分可以按照 <% term ASGNi1 = 1077> 的形式添加终结符 ASGNi1。尖括号只是为了将 LBURG 规范和其他文字区别开来,实际代码的书写当中是没有这对尖括号的。

如果所有终结符的定义都手动计算的话,这将是一件工作量不小的工作。由于 F、I、U、P、V、B 的值是确定的,每个操作符的值也是确定的,比如 CNST 等于 1 左移 4 位, ARG 等于 2 左移 4 位, INDIR 等于 4 左移 4 位,类型大小更是很明确地写在最后,因此可以简单写个程序打印出所有可能的<% term 终结符=对应的

整数>。这里需要注意的是,由于目标堆栈处理器是 16 位的,且不支持浮点运算,要生成符合目标处理器特点的终结符必须完成两个工作:首先,重新定义 csilhf d xp 的大小。为了使问题简单化,文中规定 c 等于 1, silhf d xp 都等于 2;其次,去除类型后缀是 F 的终结符。

1.2 规 则

1.2.1 栈帧的处理

传统的 C 函数编译器是利用栈帧来保存函数调用时需要的所有状态信息,包括自动变量、返回地址以及保存的寄存器。栈中为每一个函数调用保存了一个帧,这个栈是向下增长的,即向低地址方向增长。图 1 描述了 main 函数调用 proc,而 proc 又递归调用自身一次时的栈结构。

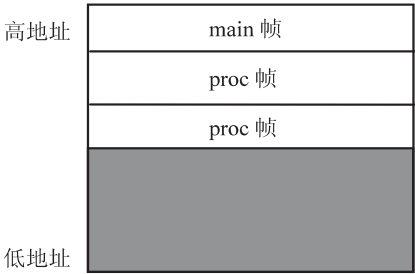


图 1 栈结构

逻辑帧指针指向帧的底部。一般而言,相对与帧指针来说,局部变量具有负的偏移量,形式参数和其他数据按照目标处理器的约定可能有正或者负的偏移量。

有些目标处理器把帧指针保存在一个物理寄存器中,比如 X86 的帧指针存储在寄存器 ebp 中。有些目标处理器仅保存栈指针,帧指针表示为栈指针和某个常量的和,比如 MIPS 处理器,如果某个程序的帧有 60 个字节,它的虚拟帧指针就是地址 $60(\$ sp)$,即 60 加上栈指针 $\$ sp$ 的值。而 $-8+60(\$ sp)$ 引用了偏移量为 -8 的局部变量^[3]。

现在面对的是一个 16 位的堆栈处理器,该处理器不提供任何通用寄存器。处理器内部提供两个物理堆栈:一个是数据栈,主要用来完成数学表达式的计算;一个是返回栈,用来保存子程序调用的返回地址。其中,对数据栈而言,处理器的操作仅针对栈顶寄存器 T 及紧接其后的两个连续的栈寄存器 N1 和 N2;对返回栈而言,处理器的操作仅仅是将返回栈栈顶寄存器 R 的值弹出保存到 T 中,或者弹出 T 的值保存到 R 中。如何在这样的硬件条件下支持 C 语言的函数调用?

为了解决这个问题,首先需要了解的是堆栈处理器支持传统高级语言的一个主要难点。前面已经描述了传统高级语言利用栈帧来保存函数调用所需要的状态信息,每个调用对应一个帧。程序员可以很方便地通过“esp+偏移量”的方式随时获取局部变量、参数的

值。由此可以看出,这个所谓的栈和真正意义上的物理堆栈是有明显区别的。真正意义上的堆栈是不允许自由访问的,访问必须按照 LIFO 的次序进行。

首先想到的解决方案是:能否将栈帧放到物理堆栈中,比如数据栈。或者如果目标堆栈处理器有两个数据栈,可以将其中一个用来进行表达式的计算,另一个用来存储栈帧。这样做的后果就是局部变量的访问变得非常地缓慢。因为物理堆栈是不允许按地址随意访问的,要取出保存在物理堆栈的局部变量,需要做一系列的 SWAP 操作直至需要的数据被放置在 T 中。这么低的效率与堆栈处理器设计的初衷已经背道而驰。

文献[10]提出了一个折中的方案,该方案通过把程序存储器驻留与硬件堆栈数据驻留组合起来处理栈帧问题。处理过程如下:所有的函数调用都把参数放到硬件堆栈上,因为这些参数或者用于计算或者用于通过硬件数据堆栈在不同的栈帧间移动。然后进行函数调用,被调用函数从硬件堆栈上赋值参数到分配的栈帧位置处作为局部变量,只留下一、两个参数。编译器保证留在数据堆栈上的参数不使用地址引用,这可以通过声明这是一个寄存器变量来做到。最坏的情况是把所有的参数都赋值到存储器中去。当函数结束时,返回值被放到数据堆栈上。显然,这种方案不能应用到文中的 16 位堆栈处理器上,因为硬件上要提供一个帧指针。

通过对目标堆栈处理器的仔细研究,文中找到了一个适合于该处理器的解决方案。该方案的基本思想是:由于返回栈在整个处理器的执行过程中只保存返回地址,且在一个函数的执行过程中,如果该函数没有执行子函数调用,返回栈中的内容是固定不变的。因此,可以利用返回栈来保存帧指针。具体方法如下:

1)把 64K 的内存空间划分成两个区域,地址 0000H 开始的往高地址方向的内存空间保存程序代码,地址 FFFFH 开始的往低地址方向的内存空间保存栈帧;

2)给出 16 位堆栈处理器的栈帧结构如图 2 所示;



图 2 16 位目标堆栈处理器的栈帧结构

3)在函数被调用之前,需要计算被调用函数的帧指针,并放到 T 中;在函数被调用时,返回地址首先被压入返回栈,这个动作是由 CALL 指令执行的,不需要额外的指令。CALL 指令执行后,在函数内部第一个执行的动作是将当前函数的帧指针压入返回栈;在函数执行结束返回时,首先需要将返回值放到 T 中,其次,需要将帧指针弹出返回栈。由于返回栈的深度是

32,所以这种方法只支持最大嵌套深度为 16 的函数递归调用:

4)对函数内局部变量和参数值的调用可以通过栈指针+偏移量的方式得出对应的内存地址,然后执行@ 或者! 指令。当然,如果值是单字节的,则执行 C@ 或者 C! 指令。

1.2.2 STACK16 的规则

常量节点直接发送该常量对应的值。

con: CNSTI1 " % a"

若常量的值为 1,则对应非终结符 con1。

con1: CNSTI2 "1" range(a, 1, 1)

非终结符 addr 对应局部变量地址和参数地址,addrj 对应跳转地址。

addr: ADDRLP2 " % a"

addrj: ADDRGP2" % a"

对于类似 reg: OP(reg, reg) 这样的树模式,可以看成是 reg= regOPreg 这样的运算。由于堆栈处理器不存在通用寄存器,所以不需要考虑输出结果是否破坏寄存器中的值。对于处理器而言,OP 前的 reg 是 T 中的元素,OP 后的 reg 是 N1 中的元素。结果放在 T 中。于是可以这样描述规则:

reg: ADDI2(reg, reg) " +\n"

目标处理器指令集中未提供位补码和取反操作,如何为 BCOM 和 NEG 节点描述规则? 这里,可以使用一个比较迂回地做法,就是将 T 中的内容与 FFFFH 进行异或操作,结果就是对 T 进行取反。在这个基础上再加上 1 就等于 T 的补。于是,可以按如下的方式描述该规则:

reg: BCOMI2(reg)

" LIT 65535\nXOR\n"

根据前面描述栈帧解决的机制,CALL 节点发送的代码由 emit2 决定。

由于需要通过计算每一个输入参数的大小得出被调用函数的帧指针,所以 ARG 节点发送的目标代码由 emit2 函数决定。

对于返回值保存在 T 中的规则,还需要另外添加一条规则,使得模式 reg 能够匹配起始非终结符 stmt。

stmt: reg ""

1.3 接口函数的实现

LCC 接口 Interface 提供 18 个接口函数,该接口内包含 Xinterface 扩展接口,这个扩展接口提供 11 个接口函数。每个函数都起一个固定的作用。当然,这些函数可以根据自己的需求进行改写,只要保证函数的逻辑和功能不变就行了。就设计的 MD 文件而言,重点需要重新书写的是 function 函数,progbeg 函数和 emit2 函数。

1.3.1 function 函数

编译前端调用 function 函数为一个例程生成目标代码。首先,function 决定如何接收和存储形式参数;然后,调用前端的 gencode。gencode 调用后端的 gen 处理代码表中的各个森林。当 gencode 结束返回的时候,后端已经看到整个程序,并计算了栈空间的大小。接着,function 开始产生函数的头代码,调用前端的 emitcode,emitcode 为代码表中的每个森林调用后端的 emit。emitcode 返回时,function 开始产生程序尾代码并返回^[3]。

以上是 function 函数的执行过程,在重写该函数时应保证函数的基本逻辑结构不变。function 的基本逻辑结构如下所示。

```
void function( symbol f, symbol caller[], symbol callee[], int
ncalls ) {
    <初始化部分>
    gencode( caller, callee );
    <发送代码的起始部分>
    emitcode();
    <发送代码的结尾部分>
}
```

初始化部分,function 首先往目标文件打印函数名,格式是“_函数名:\n”。接着判断函数名是否是_main。如果是,则往目标文件打印“LIT 65535\n>R\n”,目的是将 main 函数的帧指针压入返回栈。65535 对应 FFFFH,即内存的最高地址处。由于函数被调用前,该函数的帧指针被保存在 T 中,因此,如果当前函数名不是_main,则直接往目标文件打印“>R\n”。>R 指令将当前函数的帧指针压入返回栈。接着将 offset 和 maxoffset 赋初值为 0,offset 记录最后一个局部变量栈偏移量的绝对值,maxoffset 记录 offset 的最大值。

发送代码的起始部分,只需要将 gencode 内得到的 offset 最大值 maxoffset 对齐后赋值给变量 framesize。framesize 记录帧的大小。

发送代码的结尾部分,往目标文件打印“R>\nRET\n”。

1.3.2 progbeg 函数

在编译的开始,前端调用 progbeg 函数处理前端不能识别的程序参数,这些参数是与特定目标机器相关的选项。progbeg 处理这些选项并初始化后端。progbeg 的基本逻辑结构如下所示。

```
static void progbeg(int argc, char * argv[]) {
    int i;
    <共享部分>
    parseflags( argc, argv );
    <初始化寄存器符号>
    <与特定后端有关的部分>
```

```
}
```

由于 STACK16 不需要考虑寄存器,而共享部分对于所有的后端都是一样的,所以 progbeg 函数只需要重新书写与特定后端有关的部分。这部分只有一句代码,用来对全局静态变量 sz 进行初始化。该变量保存当前函数输入参数的总大小。

1.3.3 emit2 函数

emit2 函数处理那些不能通过指令模板来发送代码的情况。如果需要调用 emit2 函数来发送代码,规则部分必须用#开头的字符串作为指令模板。比如:stmt: RETI2(reg) "# ret\n"。对 STACK16 来说,需要处理的有 ARG 类节点、RET 类节点和 CALL 类节点。

对于 RET 类节点,emit2 不做任何处理,因为 function 函数已经处理过函数的返回了。

对于 ARG 类节点,emit2 函数判断节点操作符是否是 ARG1,ARGP,ARGU,如果是,则计算出该参数的大小,并将该参数的值保存到栈帧中。

对于 CALL 类节点,emit2 首先负责在函数被调用前将帧指针的值保存到 T 中,其次负责发送 CALL 指令,最后负责给全局变量 sz 赋零,该变量保存当前函数输入参数的总大小。

2 实验结果

文中主要为指定的 16 位堆栈处理器生成代码生成器。通过和 LCC 配合,将 C 代码转换为目标处理器的汇编指令。

2.1 待编译的 C 代码

下面给出了待编译的 C 代码,内容非常简单,不牵涉到库函数的调用。

```
int add(int, int);
void main() {
    int i=5,j=8,sum;
    sum=add(i,j);
}
int add(int i,int j) {
    int sum;
    sum=i+j;
    return sum;
}
```

2.2 生成的指令

将生成的指令按照从上到下、从左向右的顺序以表格的方式列出,如表 3 所示。

3 结束语

文中以 LCC 和 LBURG 工具为基础,根据 16 位目标堆栈处理器的特点重新设计了一个后端。该方式对于快速生成特定处理器的 C 代码编译器非常有效。

当然,除了 LCC 外还有其他的解决方案,比如,文献 [11,12]就归纳了许多可重定向的编译器。今后需解决两个问题:一个是目前的后端不能支持 C 中的结构;另一个是代码优化。

表3 生成的汇编指令

public _main	>R	R>	LIT -6	>R	DROP
_TEXT segment	+	DUP	R>	+	LIT -2
_main;	!	>R	DUP	@	R>
LIT 65535	DROP	+	>R	LIT 2	DUP
>R	LIT -4	@	+	R>	>R
LIT 5	R>	LIT -10	!	DUP	+
LIT -2	DUP	R>	DROP	>R	@
R>	>R	DUP	L1;	+	L2;
DUP	+	>R	R>	@	R>
>R	@	+	DROP	SWAP	DROP
+	LIT -8	!	RET	+	RET
!	R>	R>	public _add	LIT -2	_TEXT ends
DROP	DUP	DUP	_add;	R>	end
LIT 8	>R	>R	>R	DUP	
LIT -4	+	LIT -10	LIT 0	>R	
R>	!	+	R>	+	
DUP	LIT -2	CALL _add	DUP	!	

参考文献:

[1] 储昭贤,施慧彬. 基于 FPGA 的 16 位堆栈处理器的设计[J]. 微电子学与计算机,2012,29(2):22-26.

[2] Fraser C W, Hanson D R. A retargetable C compiler; design

(上接第 162 页)

在航天地面应用支持系统中得到广泛应用提供基础。在未来的工作中,执行环境还有很多完善的地方,比如执行器以 workflow 形式显示和执行,以及图形化编辑器设计等,从而提高系统的可用性。

参考文献:

[1] 杨震. 地面测控系统支持“遥操作”研究[C]//中国空间科学学会空间探测专业委员会第十九次学术会议论文集:下册. 出版地不详;出版者不详,2006.

[2] 李征宇,刘建,李绪志. 一种新型卫星测试操作工具软件的设计与实现[J]. 计算机工程与应用,2005(4):116-118.

[3] Terence P. 编程语言实现模式[M]. 李袁奎,尧飘海译. 武汉:华中科技大学出版社,2012.

and implementation [M]. [s. l.]: Benjamin/Cummings Pub. Co., 1995.

[3] Fraser C W, Hanson D R. 可变目标 C 编译器-设计与实现[M]. 王挺,黄春译. 北京:电子工业出版社,2005.

[4] 张红光,赵彩云,李海丰,等. 可重定向 C 编译器中 DAG 及归约规则[J]. 计算机工程,2008,34(17):74-76.

[5] Pelegri-Llopert E. Rewrite Systems, Pattern Matching, and Code Generation [D]. Berkeley: University of California, 1987.

[6] Pelegri-Llopert E, Graham S L. Optimal code generation for expression trees; An application of BURS theory [C]//Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York: ACM, 1988: 294-308.

[7] Fraser C W, Henry R R, Proebsting T A. BURG-Fast Optimal Instruction Selection and Tree Parsing[J]. SIGPLAN Notices, 1992,27(4):68-76.

[8] 胡伟平. 可扩展编译系统的关键技术研究[D]. 北京:中国科学院计算技术研究所,1998.

[9] Fraser C W, Hanson D R. The lcc 4. x code-generation interface[R]. Redmond, WA: [s. n.], 2001.

[10] Koopman P J. Stack computers: The new wave [M]. California: Ed. Mountain View Press, 1989.

[11] 王民华,张素琴,田金兰. 基于类库的可重定向编译后端设计与实现[J]. 计算机工程与应用,2003,38(9):115-118.

[12] 戴桂兰,张素琴,田金兰,等. 编译基础设施中多目标编译技术探讨[J]. 计算机研究与发展,2003,37(2):312-317.

[4] 杨永安,余培军,张武光,等. 面向航天器控制的专用语言及编译程序设计[J]. 计算机工程,2006,32(12):247-249.

[5] ECSS Secretariat. ECSS-E-ST-70-32C-test and operations procedure language[S]. 2008.

[6] Reid S, Pascoe A, Dankiewicz I. Learning from the Experience of Spacecraft Operations Automation [C]//SpaceOps 2006 Conference. [s. l.]: [s. n.], 2006.

[7] 张素琴,戴桂兰,田金兰,等. 面向对象编译类库构造[J]. 清华大学学报,2003,43(7):964-967.

[8] 严坤,倪桂强,姜劲松,等. 基于 AOM 和插件模式的轻量级框架研究[J]. 计算机技术与发展,2010,20(10):54-57.

[9] Chaudhri G, Hollander S. Ground Systems - The Need for Standardization [C]//SpaceOps 2004 Conference. [s. l.]: [s. n.], 2004.

堆栈处理器代码生成器的设计与实现

作者: [赵小东, 施慧彬](#)
作者单位: [南京航空航天大学 计算机科学与技术学院, 江苏 南京 210016](#)
刊名: [计算机技术与发展](#)
英文刊名: [Computer Technology and Development](#)
年, 卷(期): 2013(4)

本文链接: http://d.g.wanfangdata.com.cn/Periodical_wjfz201304042.aspx