堆栈处理器汇编器的设计与实现

赵小东[1],施慧彬[2]
(1. 94860 部队,江苏 南京 210000;
2 南京航空航天大学 计算机科学与技术学院,江苏 南京 210006)

摘　要:堆栈处理器由于其快速的中断处理能力、极强的过程调用能力、代码尺寸小等优点,一直应用于工业控制和航空航天领域中。自堆栈处理器诞生以来,ALGOL、FORTH 这两种语言一直作为堆栈处理器的栈操作语言,并且成为区分第一代和第二代堆栈处理器的一条重要标准。尽管这两种语言在堆栈处理器领域应用广泛,但缺乏通用性。C 语言是传统的高级程序设计语言,其底层操作能力和通用性不言而喻。文中针对一款基于 FPGA 设计的 16 位堆栈处理器,设计并实现了一个汇编器。该处理器包含两个堆栈:执行数学表达式的数据堆栈和支持子程序调用的返回堆栈,其指令集含 35 条堆栈指令。汇编器可以将汇编代码转换成能够直接在 ModelSim 下仿真的内存文件。论文详细给出了汇编器的设计方法。最后简单做了一个测试验证汇编器的正确性。

关键词:堆栈处理器;汇编器;仿真;FPGA;ModelSim
中图分类号:TP314　　**文献标识码**:A　　**文章编号**:1673-629X(2021)0067-06

Design and Implementation of Assembler for Stack Processor

ZHAO Xiao-dong[1], SHI Hui-bin[2]
(1. 94860 PLA Troops, Nanjing 210000, China;
2. School of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210006, China)

Abstract: Stack processor, because of its fast interrupt handling, low procedure call overheads and small instruction format, is used in the field of indusry control and aerospace. ALGOL and FORTH have been the stack operating language of the stack processor, respectively respecting the first-generation stack computer language and the second-generaion stack computer language, to distinguish these two generation stack computer language. Both of these languages are widely used in the stack processor, while lacking is generality. C language is a traditional high level programming language, with good underlying operating capacity and generality. We introduce an assembler for 16-bit stack processor. The stack processor has two stacks: a data stack for evaluating mathematical expression and a return stack for calling subroutine. The instruction set includes 35 stack instructions. Assembly code can be assembled to memory file which can be used by ModelSim directly. We describe the design method to construct an assembler. Finally, we do a simple test to verify the correctness of the assembler.

Key words: stack processor; assembler; simulation; FPGA; ModelSim

0　引　言

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请参考文献[1]。此次研究的目标就是将一个符合 C 语法的源程序编译成能够在目标处理器上执行的指令。为了更好地实现这一目标,将研究分成三部分:编译器、汇编器、优化器。编译器负责将 C 代码转换成汇编器能够识别的汇编指令,这方面主要的支撑工具就是 LCC。LCC 可以通过重新书写目标机器描述文件的方式生成符合特定处理器汇编语法的汇编文件。有关 LCC 的详细信息请参考文献[2-3]。汇编器负责将汇编指令转换成能够直接在 ModelSim 上仿真的内存文件;优化器负责内存文件的优化工作,做到降低目标代码尺寸、提高执行效率。文中描述的是汇编器。首先介绍汇编文件和内存文件的格式,接着介绍汇编器的总体逻辑结构、重写信息表逻辑结构,在此基础上详细给

收稿日期:2021-02-25
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,工程师,研究方向为计算机系统结构;导师简介:施慧彬(1966-),男,副教授,研究方向为计算机系统结构、智能计算系统、边缘计算、编译优化技术等。

出关键模块的程序流程图。最后给出一个简单 C 代码,通过编译器和汇编器的转换,生成目标文件。ModelSim 仿真工具将目标文件作为目标处理器的内存文件载入,并通过波形图的方式给出实验结果。

1 设计与实现

1.1 汇编文件格式

汇编文件是汇编器的输入,文件的内容是汇编器能够识别的汇编代码,该汇编代码能够体现目标堆栈处理器的特点,符合该处理器的指令集,是目标处理器二进制代码的符号描述。同时,为了能够方便描述诸如全局变量、跳转等情况,汇编器还增加了对伪指令的支持。伪指令不产生目标代码,只是用来说明程序的控制结构和全局数据的定义。

汇编文件采用后缀名为 ASM 的文件,文件的内容主要分成两个部分,具体如下所示。

```
_DATA segment
… … … …
_DATA ends
_TEXT segment
… … … …
_TEXT ends
… … … …
end
```

容易看出,汇编文件的内容由许多 DATA 段和 TEXT 段组成,end 标识文件内容的结束。由于文中是通过重新书写目标机器描述文件的方法生成汇编文件的,所以不得不提一下 LCC 中定义的四个段 CODE、LIT、BSS、DATA。LCC 前端管理这 4 个逻辑段,将可执行代码发送到 CODE 段,将未初始化的变量定义到 BSS 段,将已初始化的变量定义到 DATA 段并初始化,常量则定义到 LIT 段。这里,CODE 和 LIT 段可以映射到目标机器只读的段上,BSS 和 DATA 段必须映射到目标机器可读写的段上。文中将 CODE 和 LIT 段映射到只读的段 TEXT,BSS 和 DATA 段映射到可读写的段 DATA。

汇编指令包括两部分,一部分是目标处理器能够执行的指令,另一部分是目标处理器不能够执行的指令。能够执行的指令总结在表 1 中。汇编指令中用到的伪指令主要分成三类,一类用于段的定义;一类用于符号的定义或内存的初始化;一类用于程序的控制结构。表 2 归纳了所有用到的伪指令。

表 1 可执行汇编指令

指令名称	目标代码	指令功能	指令名称	目标代码	指令功能
R>	04	弹出 R 中元素,压入数据堆栈	+	2C	弹出 N1 中元素与 T 中元素相加
OVER	05	复制 N1 中元素,压入数据堆栈	LSL	31	T 中元素逻辑左移
DUP	06	复制 T 中元素,压入数据堆栈	LSR	32	T 中元素逻辑右移
>R	08	弹出 T 中元素,压入返回堆栈	ASR	34	T 中元素算术右移
DROP	09	弹出 T 中元素并抛弃	SHLD	39	实现除法功能
NIP	0A	弹出 N1 中元素并抛弃	1MIN	3A	T 中元素减 1
NOP	10	空操作	–	3C	弹出 N1 中元素与 T 中元素相减
SWAP	11	交换 T 与 N1 中元素	LIT	40	访存取出指令存储器中的字常量压入 T
DRJNE	7A	将 R 中元素减 1,如果不为 0,则跳转,否则不跳转	@	41	以 T 中元素为地址访存取出字数据后替换 T 中已有元素
ZERO	13	置 T 中所有比特位为全 0	!	42	弹出 T 中元素为地址,弹出 N 中字数据存入数据存储器
ROT	14	旋转交换 T、N1 和 N2 中内容	LITC	50	将字节常量扩展为 16 位压入数据堆栈
AND	21	弹出 N1 中元素与 T 中元素相与	C@	51	以 T 中元素为地址访存取出字节数据后替换 T 中已有元素
XOR	22	弹出 N1 中元素与 T 中元素相异或	C!	52	弹出 T 中元素为地址,弹出 N 中字节数据存入数据存储器
OR	24	弹出 N1 中元素与 T 中元素相或	CALL	MSB = 1	执行子程序调用
MPP	29	实现乘法功能	RET	60	弹出 R 中元素,送入 PC 寄存器
1PLUS	2A	T 中元素加 1	JMP	61	无条件跳转
JNC	79	若进位寄存器为 0,则跳转,否则不跳转	JZ	78	若 T 中元素为 0,则跳转,否则不跳转
ONE	12	置 T 中所有比特位为全 1			

表 2 伪指令

指令类型	指令名称	指令功能
段的定义	_DATA segment	数据段的开始
	_DATA ends	数据段的结束
	_TEXT segment	文本段的开始
	_TEXT ends	文本段的结束

续表 2

指令类型	指令名称	指令功能
内存初始化	public 模块名称	输出模块
	align n	对齐字节数为 n
	变量名或标号 label byte	初始化标号或变量所对应内存中的内容
	db 数据	后面的数据是字节对齐的
	dw 数据	后面的数据是字对齐的
	dup 数据	重复,比如 db 2 dup (0) 代表用 0 初始化当前位置的两个字节
程序结构控制	变量名、标号名或模块名后面接一个冒号	模块或变量定义的开始,标号指向的位置。
	end	汇编程序的结束

1.2　目标代码的格式

目标文件是一个 DAT 文件。在 verilog 中,系统任务 $ readmemb 和 $ readmemh 读取 DAT 文件的数据并映射到存储器中。如果数据是二进制,使用 $ readmemb;如果数据是十六进制,使用 $ readmemh。这两个任务都是从低端开始读数据到内存的低地址处。由于目标 16 位堆栈处理器的存储器是高低 8 位分开来的,因此需要生成两个内存文件,高 8 位内存文件名称是 ramh. dat,低 8 位内存文件名称是 raml. dat。内存文件的格式如下所示:

@00 字节数据
@01 字节数据
@02 字节数据
@03 字节数据
@04 字节数据
… … … …

从上面可以看出,内存文件其实非常简单,@ 声明一个地址,比如“@00 字节数据”表示该字节数据保存在地址 00 处。由于有高低 8 位的区别,所以地址 00 应称为字地址。

1.3　汇编器的设计

汇编器共包含 12 个模块,分别是 main,main_init,translate,data,text,code,rewrite,output,fillbuf,maininit,process,printcode。功能上将这 12 个模块分成 4 部分:总控部分、输入部分、转换部分、输出部分。下面首先介绍汇编器的总体逻辑结构,然后介绍二次填写法和重写信息表,最后详细介绍实现汇编器的几个关键模块。

1.3.1　汇编器总体逻辑结构

汇编器的总体逻辑结构如图 1 所示。从图中可以看出,汇编器由总控部分、输入部分、转换部分、输出部分组成。总控部分负责协调输入部分、转换部分和输出部分,使之能够按照正确地逻辑运行。输入部分负责从 ASM 文件中读取数据;转换部分负责将取到的数据送到 data 模块、text 模块、rewrite 模块进行数据转换,其中:data 模块负责处理 DATA 段里的内容,text 模块负责处理 TEXT 段里的内容,rewrite 模块负责将全局变量及函数的定义地址、所有 CALL 指令对应的 16 进制代码重新填入到输出文件的正确位置当中;输出部分负责生成能够直接应用到 ModelSim 进行仿真的高低 8 位内存文件。当 ASM 文件所有的数据都读取完后,总控部分调用输出部分。

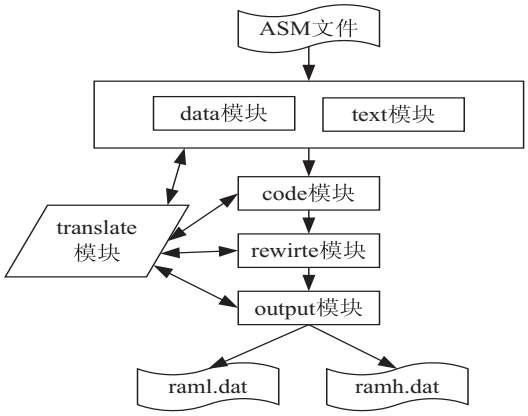

图 1　汇编器总体逻辑结构

1.3.2　二次填写法与重写信息表

汇编器能够生成目标代码需要解决几个问题:一、建立可执行汇编指令与目标代码的一一对应关系,比如 RET 与十六进制数 60 对应;二、由于目标处理器一次从存储器中取 2 个字节,所以必须处理好诸如 LIT、JMP、JNC、JZ、DRJNE 这类有效指令字节数是 3 的情况;三、CALL 的特殊处理,因为目标处理器是这样处理 CALL 指令的:当从存储器中取出的 16 位数最高位是 1,则将这 16 位数看成一条 CALL 指令,去掉最高位 1,将剩下的 15 位数左移 1 位得到被调用函数的地址。从处理器执行的过程来看,函数的地址只能是字地址,而且 CALL 指令只占 2 个字节。函数的地址只有遇到函数的定义才能知道,所以只有在所有的 ASM 文件读完后才能确定被调用函数的地址,且汇编器必须做特殊的处理,生成 2 个字节的 CALL 指令。

为了能够正确生成目标代码,汇编器采用了二次填写的方法:第一次填写负责生成除 CALL 以外没有操作数的指令;第二次填写负责 CALL 指令的生成,当 JMP、LIT、JNC、JZ、DRJNE 后面接符号而不是 16 位明确的数据时,还需要填入该符号对应的地址。code 模块完成第一次填写任务,rewrite 模块完成第二次填写任务。

为了能够方便地实现上面提出的二次填写法,需要设计一个数据结构能够包含所有有关重新填写的信息。全局变量和标号对应的地址需要重新填写,整型变量 node. addr 保存这个地址,结构 cite 保存目标文件中所有需要重新填写对应地址的位置;函数名对应的是 CALL 指令的目标代码,该信息需要在出现调用该函数的地方进行重新填写,整型变量 node. call 保存该函数名对应 CALL 指令的目标代码,结构 func 保存目标文件中需要重新填写 CALL 指令对应目标代码的位置。结构 node 保存 ASM 文件中所有用到的全局变量名、函数名及标号。这三个结构共同构建了一张重写信息表,该表的逻辑结构如图 2 所示。

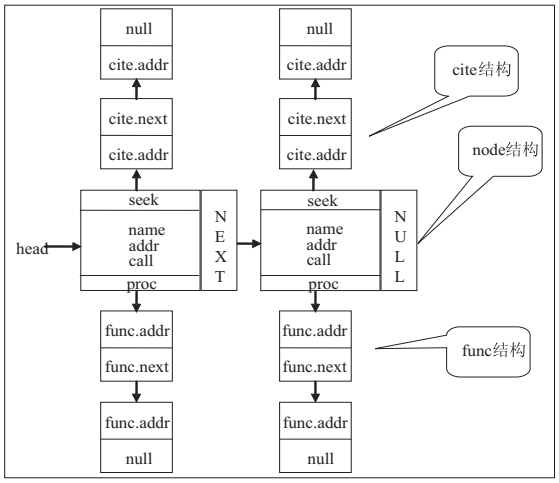

图 2 重写信息表的逻辑结构

第二次填写时,汇编器只需要遍历这张重写信息表。如果当前符号是全局变量或标号,则遍历 seek 指针所指向的链表,根据每个链表节点的 addr 域在输出文件正确的位置处填写该全局变量或标号对应的内存地址。如果当前符号是函数名,则遍历 proc 指针所指向的链表,根据每个链表节点的 addr 域在输出文件正确的位置处填写正确的 CALL 指令。

1.3.3 关键模块的实现方法

汇编器的主要部分就是转换和输出,转换涉及 code 模块、rewrite 模块、text 模块、data 模块,输出涉及 process 模块。下面将详细叙述这 5 个模块的实现方法。

code 模块负责第一次填写,所以其实现相对简单,只是在遇到 CALL、LIT、LITC、JMP、JZ、JNC、DRJNE 时需要做特殊处理。其程序流程如图 3 所示。

从流程图中可以看出,code 模块首先判断当前汇编代码是否是 CALL。如果是,且当前处于高 8 位,则往目标文件发送“10”。如果当前处于低 8 位,则不需要发送“10”。为什么要这样做呢?

处理器提供的可执行指令按指令字节数来划分共有三类,分别是单字节指令、双字节指令、三字节指令。由于目标处理器一次取指的宽度是 16 位,所以双字节指令必须处于处理器某次提取的 16 位指令中,不会出现某条双字节指令的低 8 位处于当前处理器分析指令的高 8 位,双字节指令的高 8 位处于下一个指令字的低 8 位这种不连续的现象。所以,在往目标文件发送代码时,必须考虑当前是处于低 8 位还是处于高 8 位。对于三字节指令 XX AA BB 而言,如果当前处于低 8 位,则首先往目标文件发送 XX NOP,然后发送三字节指令后面的 16 位数据 AA BB,其中,NOP 的 16 进制代码就是“10”。

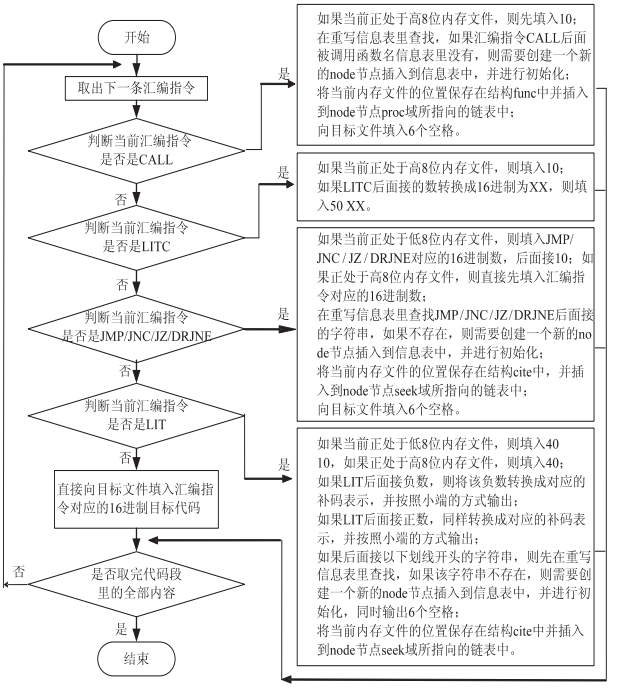

图 3 code 模块程序流程

rewrite 模块负责第二次填写,填写的依据就是重写信息表。该模块遍历重写信息表,对于每一个符号,都遍历 seek 和 proc 指向的链表。node. addr 保存了该符号对应全局变量或标号的地址,seek 指向的链表保存了所有需要用此地址重写的目标文件的位置。node. call 保存了该符号对应 CALL 指令的目标代码,proc 指向的链表保存了所有需要用此目标代码重写的目标文件的位置。

text 模块负责处理 TEXT 段。TEXT 段是只读的段,段中的内容有两种情况,一种是代码,一种是只读内存的初始化,比如全局字符串指针指向的内存。

text 模块首先判断当前是代码段还是只读内存的初始化。如果是代码段,则调用 code 模块进行处理;如果是只读内存的初始化,则将指向该内存的标号进行初始化并添加到符号表中。当然,在处理代码段时,code 模块同样需要对声明或定义的符号进行初始化工作,并添加到符号表中。

该模块的程序流程如图 4 所示。

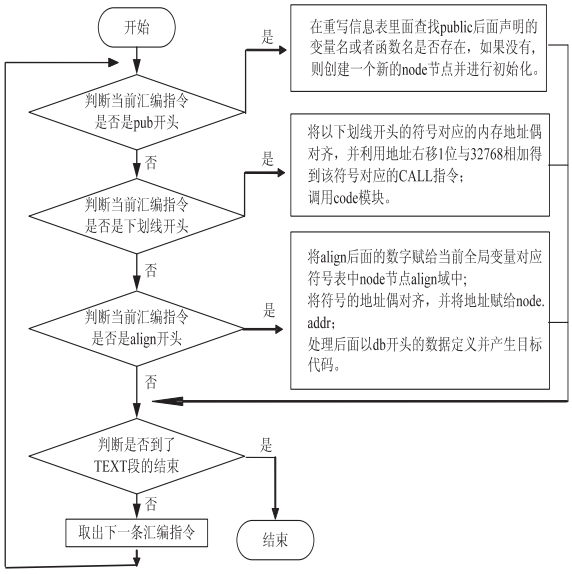

图 4　text 模块的程序流程

data 模块负责处理 DATA 段。DATA 段是可读写的段,主要是局部变量和参数对应内存的初始化。思路与 text 模块基本相同,所以就不再画出该模块的程序流程图了。

process 模块负责内存文件的生成。在描述这个模块之前,首先需要交代的是在执行 process 模块之前,汇编器已经生成了一个符合 verilog 输入文本文件要求的内存文件,该文件并没有将一个字的高低 8 位分开存放,而是放在一个文件当中,文件的类型是 TXT。该文件的格式如下所示。

```
@0000//说明下面的内容从地址 0000 开始
XX XX XX XX XX XX XX XX
XX XX XX XX XX XX XX XX
@0010
XX XX XX XX XX XX XX XX
XX XX XX XX XX XX XX XX
XX XX XX XX XX XX XX XX
@0028
XX XX XX XX XX XX XX XX
XX XX XX XX XX XX XX XX
@0038
XX XX XX XX XX XX XX XX
```

从 TXT 内存文件的格式可以看出,这个文件与 DAT 内存文件的区别是没有将每一个字节数据的前面都注明其内存地址。内存文件是以段的方式组织起来的,不管是全局变量和全局指针对应内存空间的初始化,还是每一个函数的定义,都可以看成一个段,段的起始就是一个地址声明标志。虽然 TXT 文件并不能直接应用到 ModelSim 进行仿真,但文件内容思路清晰,并保留着汇编文件内容的痕迹,所以比较适合阅读并可以大致判断汇编器生成目标文件的正确性。该文件还一个用处就是作为 process 模块的输入。如果将 TXT 内存文件作为 process 模块的输入,存在一个问题:process 模块如何得知当前是处于代码段还是数据段？因为同样是 60,如果看成代码的话,就是 RET 指令;如果看成数据的话,就是 ASCII 的'<'。于是,对 TXT 文件进行了简单地修改,凡是以@ 开头声明地址的,就是数据段的开始,凡是以#开头声明地址的,就是代码段的开始。图 5 详细描述 process 模块的程序流程。

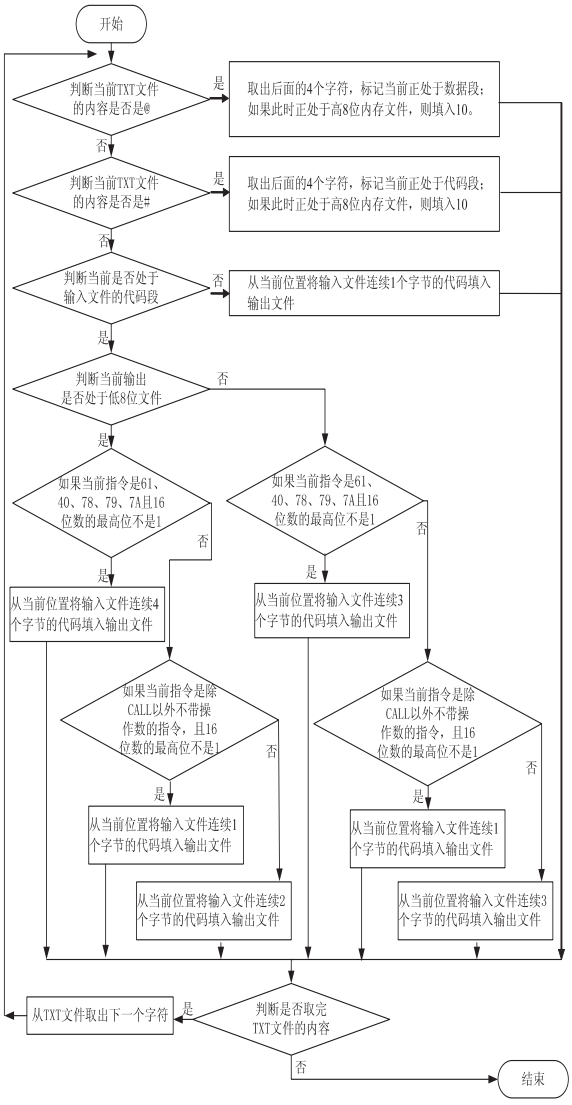

图 5　process 模块程序流程

2　实验结果

由于目标处理器是通过 FPGA 设计完成的,所以验证平台选用了仿真工具 ModelSim6.5。如果能成功加载内存文件,就可以通过波形图来验证代码执行的正确性。

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 仿真结果

从 C 代码中可以看出,当 add 函数返回时,T 中应保存该函数的返回值 13,其 16 进制数是 0D。此时已经执行完返回指令 60,且 T 中的内容是 0D。

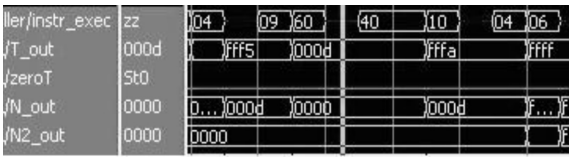

图 6 仿真波形

3 结束语

文中设计的汇编器,虽然功能上是正确的,但生成的目标代码还有许多可以优化的空间。下一步研究的重点是目标代码的优化工作,设想通过增加一个优化器,使得生成的目标代码体积更小,执行效率更高。

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 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-LLOPART E. Rewrite systems,pattern matching,and code generation[D]. California: University of California,Berkeley,1987.

[6] PELEGRI-LLOPART 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]. [s. l.]:[s. n.],2001.

[10] KOOPMAN P J. Jr. 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.

[13] 吴元斌. C 语言程序的理解与编译优化[J]. 现代计算机,2020(18):93-96.

[14] 水 颖. 一种新型粗颗粒度可重构架构[J]. 科学技术创新,2020(15):76-77.

[15] 高国军,任志磊,张静宣,等. 编译优化序列选择研究进展[J]. 中国科学:信息科学,2019,49(10):1267-1282.