编号_____

南京航空航天大学

毕业设计

题　目　X86-MIPS 二进制翻译器的设计与实现

学生姓名	金航
学　　号	161530115
学　　院	计算机科学与技术学院
专　　业	软件工程
班　　级	1615301
指导教师	冯爱民　副教授

二〇一九年六月

南京航空航天大学

本科毕业设计（论文）诚信承诺书

本人郑重声明：所呈交的毕业设计（论文）（题目：X86-MIPS 二进制翻译器的设计与实现）是本人在导师的指导下独立进行研究所取得的成果。尽本人所知，除了毕业设计（论文）中特别加以标注引用的内容外，本毕业设计（论文）不包含任何其他个人或集体已经发表或撰写的成果作品。

作者签名：　　年　月　日

（学号）：

X86-MIPS 二进制翻译器的设计与实现

摘 要

随着计算机软件和硬件技术的发展，计算机的用途和硬件层面的实现种类变得相当广泛，现今计算机市场上使用最为广泛的指令集体系架构为 x86 指令集。近年来，随着由中国科学院自主研发的基于 MIPS 架构的龙芯 CPU 逐渐进入硬件市场，国内对于 MIPS 系列的软件需求逐渐增大。由于大部分软件厂商通常只提供 x86 版本的闭源发行软件，导致包括 MIPS 在内的其他架构计算机的用户无法使用，对 MIPS 和龙芯的发展影响均较为不利。

本文设计了一个基于 Linux 系统的 x86-MIPS 二进制翻译器，通过对 x86 可执行程序采用程序头信息分析和反汇编分析等手段，提取原生 x86 程序的精确指令行为和数据信息并对应翻译成可在 MIPS 架构计算机上执行的汇编代码。实验证明，经过在 MIPS 目标机上对翻译后的汇编代码进行汇编和链接，可以得到运行表现和原生 x86 程序几乎等价的 MIPS 架构二进制可执行程序。

关键词：指令集，体系结构，汇编指令翻译，X86，MIPS

Design and Implementation of x86-MIPS Binary Translator

Abstract

With the development of computer software and hardware technology, ways to use computers and types of computer hardware are boosting nowadays. The most popular Instruction Set Architecture (ISA) in the market is x86, of course. But in recent years, with the MIPS-based Loongson ISA pouring into the field of hardware by the Chinese Academy of Sciences (CAS), needs for MIPS-based software are becoming larger than ever before. For the reason that most software providers only release the x86 version close source software, computer users of architectures other than x86 are unable to get access to them, which makes an extremely negative effect on the development of both MIPS and Loongson.

This article mainly designed a binary translator of x86-MIPS binaries. It analyzed the program headers and the disassembles of an ELF binary file in Linux platform by extracting the accurate operations of x86 instructions and data from the input file and convert it into MIPS-based assembly instructions which can be executed in MIPS-based computers. We have proofed that by assembling and linking translated MIPS codes, we are able to run the translated program as if running in the original architecture.

Key Words：Instruction Set; ISA; Translation of Disassembly; x86; MIPS

目 录

第一章 引 言

1.1 研究背景

在当今的计算机市场中，最为普遍的计算机计算模型当属冯·诺依曼所提出的计算体系，其中比较常见的指令集体系结构包括 x86、MIPS、ARM、RISC-V 等，它们分别有着自己的一套数据通路和体系架构。目前，处于领导地位的指令集体系结构仍然是 x86，不管是个人电脑还是服务器，x86 仍然占据着大量的市场。由于 x86 等复杂指令集计算机（CISC）指令系统具有种类复杂、指令形式不定、寻址方式多样、执行效率低等特点，其解码难度和效率都非常高，复杂指令所带来的速度的提升早已不及在解码上浪费的时间。

因此，以 ARM 和 MIPS 为代表的 RISC 指令集逐渐地在硬件市场中占据一席之地。人们逐渐发现 MIPS 指令集的体系结构具有数据通路结构简单、指令格式规范、硬件细节开放程度高等特点。和 ARM 相比，MIPS 的硬件设计开放程度更高，内部实现更加透明。X86 仍然存在的理由也是为了兼容大量的 X86 平台上的软件，而 MIPS 是高效精简指令集计算机（RISC）体系结构中最优雅的一种。它凭借设计简单、设计周期更短等优点，发展迅速，成为非常流行的一种 RISC 处理器指令集。

自 21 世纪初起，由中国科学院自主研发的基于 MIPS 架构的龙芯 CPU 逐渐进入国内硬件市场。时至今日，基于 MIPS 的龙芯 CPU 已经在国防、军事、卫星、商用服务器、开发者主机、个人桌面主机等方面都拥有了举足轻重的地位。然而，MIPS 和龙芯的生态环境都不如 x86 的发展情况乐观，较多闭源软件都没有提供 MIPS 或龙芯的发行版，导致部分应用只能在基于 x86 体系的计算机上运行，这对 MIPS 和龙芯的发展都相当不利。

在前期的英特尔 x86 指令集的影响下，MIPS 指令集的推广也需要兼顾到 x86 指令集的程序，来提高系统的兼容性。所以，以此为出发点，如果能够借助 x86 指令集二进制目标代码到 MIPS 指令集目标代码的翻译器来实现这个转换，就可以减少很多重复的工作，并且提供更好的系统兼容性。如何在未能取得软件源代码的情况下，将面向 x86 的程序移植到 MIPS 平台上成为一个有意义且亟待解决的问题。

就目前而言，比较成熟的 CPU 复用技术包括虚拟机和模拟器两类。其中虚拟机只能通过

CPU 虚拟化运行特性，实现多系统同时虚拟化运行，并不能实现软件跨体系运行。模拟器方面，使用最为广泛的则当属 QEMU 和 Bochs，此处列出的两个模拟器都具有轻量、模拟效率较高、仿真程度高等特点，可以模拟出包括 x86、x86_64、ARM、MIPS、PowerPC 等多种架构的 CPU[1]。虽然模拟器可以方便地在目标机上模拟出目标软件所需的计算机体系，但是需要极大的硬件资源和性能，其代价不容忽视。此外，对于一般用户而言，学习使用模拟器也将成为制约模拟器面向一般用户群体使用的一大瓶颈。因此，使用模拟器运行软件并不是一个长期可行的解决方案，也就由此提出了将原始软件翻译为目标平台上可以运行的二进制程序后执行的解决方案[5]。

现在，二进制翻译器层出不穷，但是因为两个指令集之间本身的差异，并不能很好的解决在指令转换时存储的管理[6][8]和过程调用的设计 [7][9]。所以，如果能够成功搭建 x86 到 MIPS32 指令集的二进制翻译器的框架，实现更进一步的优化，使 x86 程序在翻译后能够更好地完成对 MIPS32 指令集的契合，将可以省掉多余的操作，从而提高效率和系统兼容性[10]。

1.2 国内外研究现状

如今，二进制翻译技术已经得到了广泛的重视和研究。主要的二进制翻译技术分为三类：解释执行、静态执行和动态执行[2][3]。三类方法各有优缺点。所以在目前实际的开发中，一般会使用方法中的两种或三种。并且也推出了动态优化技术，是一种独立的代码优化技术，负责对程序的关键段进行必要的优化[11]。

根据所查阅的文献资料，列举一些有代表性的二进制翻译系统。

有特定目标的二进制翻译系统，如 DigitalBridge 是由中科院计算所开发的一个动静结合的二进制翻译系统，它采用动态翻译和静态翻译相结合的方式，使用模拟的标志寄存器进行二进制翻译，将 x86 的应用程序翻译到 MIPS 上执行，我们认为动态翻译导致程序运行效率会受到一定的影响。另外还有 DEC 公司 1996 年研发的 FX!32 系统和 HP 公司的 1999 年开发的 Aries 软件仿真器等。但是这些是需要在特定的操作系统环境下运行，着眼于应用程序的翻译。而 IBM 公司于 1996 年和 1999 年开发的 Daisy 和 BOA 系统，以及 Transmeta 公司在 2000 年开发的 Code Morphing 软件，则是对整个系统的翻译，涉及到硬件，即将翻译程序放在 ROM 中。Code Morphing 软件就是动态翻译运行 x86 代码，翻译全部程序包括 windows 的操作系统。我们的项目是与该系统类似，针对于 x86 的二进制文件，建立在硬件层面，但是我们的项目只涉及到框架的搭建。

同时也有可变源和目标的二进制翻译系统，即提供一个可重用的框架，在开发某个具体的二进制翻译器时，可重用这些公共的代码，加速定制的二进制翻译器的开发。昆士兰大学先后研究开发了可变源和目标的静态和动态二进制翻译系统框架，引导了二进制翻译研究中的一个新的研究方向。代表系统有 UQBT、UQDBT、Bintran、Dynamite 和 QuickTransit 等。但是还并没有相当成熟的框架。

那么目前，在二进制翻译器的研究中，能够完整的实现翻译（不仅是指令的简单执行，也包括在存储、效率等方面达到目标指令集的要求）并且能够针对目标指令集的特点，做进一步的优化仍然很难实现。

1.3 本文主要研究内容

本文设计了一种新型的 x86-MIPS 二进制翻译器，具有解析 x86 可执行二进制文件代码和数据、跨指令集体系结构翻译和目标指令集体系架构汇编语言代码生成等系列功能。本文提出了一种“一次信息收集、两次函数扫描、三次指令匹配”的翻译方案，确保各指令被正确匹配和翻译。着重于解决 x86 和 MIPS 两体系结构间程序流控制规范的不同，设计出了两体系架构间寄存器翻译对应关系、程序内分支循环等非顺序结构程序的翻译模式、函数（或过程）调用的翻译模式等。本翻译器使用 MIPS 指令集对 x86 指令予以独立表出，各指令翻译相对独立，不仅能够取得较高的翻译效率，亦使翻译器具有较好的可扩展性。设计时采取 x86 前端指令匹配和 MIPS 后端翻译分离的翻译方案。因此，基于本文提出的翻译器结构，可以做到仅修改翻译后端就能实现 x86 指令向任一指定指令集翻译的扩展功能。

1.4 论文组织结构

第一章：引言。介绍本文的研究背景、当前国内外的在本方向上的一些研究情况和本文要研究的主要内容等，最后介绍了本文的章节安排。

第二章：相关技术分析。介绍了本文中所提及的二进制解析技术、x86 和 MIPS 指令集的程序规范和本文研究中所使用到的开源库。

第三章：二进制翻译器的设计与实现。本章是本文的核心，介绍了本文所提出的二进制翻译器的程序结构设计、部分比较重要的 x86 指令的翻译模式和 MIPS 目标汇编代码生成和用户做进一步汇编链接的方法。

第四章：实验与结果分析。对若干组测试用例，通过仿真实验和真机实验与原生 x86 二进制程序行为相对比，得到实验结果和对比结果。

第五章：工作总结与展望。总结了论文的内容，提出了本文工作中的一些不足之处和可以扩展的方面。

第二章　相关技术分析

2.1　ELF 文件格式

2.1.1　文件格式

Linux 操作系统下的可执行目标文件、可重定位目标文件、动态链接库和静态链接库均采用可执行可链接格式（Excecutable and Linkable Format，ELF）组织程序内的程序信息、代码内容和数据信息。以 ELF 的可执行目标文件为例，其组织结构[12]如图 2.1 所示。

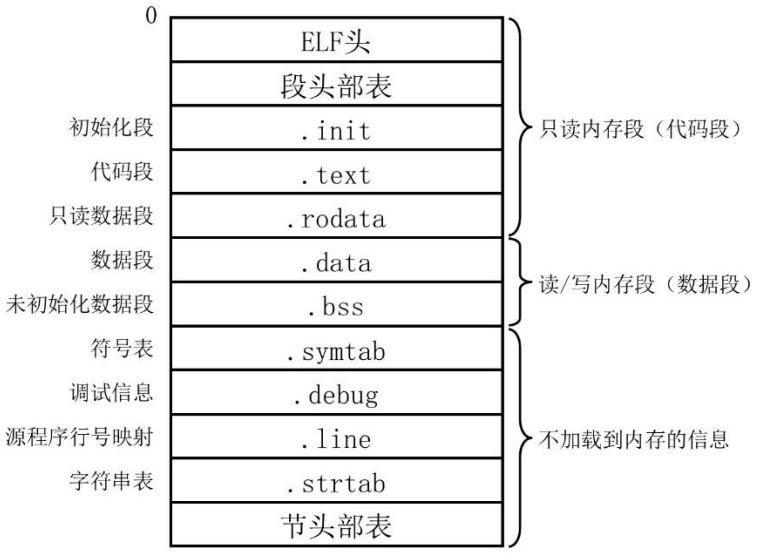

图 2.1 ELF 可执行目标文件结构

其中，段头部表中包含了程序中各个段在 ELF 文件中的存储情况和部分加载信息。只读内存段加载到内存中后仅允许程序对其进行访问，不允许修改，读/写内存段数据允许程序访问和修改，而“不加载到内存的信息”通常用于调试器调试或加载器在加载程序时使用。

2.1.2　节头部表和字符串表

ELF 文件的节头部表（section header table）中存储了每个节（section）的类型、地址、ELF 文件内位置和节大小等信息[12]，但并未直接以表项形式存储每个节的名称，于是引入了节头字符串表（.shstrtab）。该表是一连串连续的可读字符，每个字符串之间以一个'\0'字符隔开，当节头部表需要引用某个节的名称时，通过存储的本节名称的节头字符串表索引从

节头字符串表中得到本节名称的字符串。图 2.2 和图 2.3 分别是示例程序 ELF 文件的节头表和节头字符串表的相关信息。

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        00000154 000154 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            00000168 000168 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            00000188 000188 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        000001ac 0001ac 000024 04   A  5   0  4
  [ 5] .dynsym           DYNSYM          000001d0 0001d0 000090 10   A  6   1  4
  [ 6] .dynstr           STRTAB          00000260 000260 0000aa 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          0000030a 00030a 000012 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         0000031c 00031c 000030 00   A  6   1  4
  [ 9] .rel.dyn          REL             0000034c 00034c 000048 08   A  5   0  4
  [10] .rel.plt          REL             00000394 000394 000008 08  AI  5  24  4
  [11] .init             PROGBITS        0000039c 00039c 000023 00  AX  0   0  4
  [12] .plt              PROGBITS        000003c0 0003c0 000020 04  AX  0   0 16
  [13] .plt.got          PROGBITS        000003e0 0003e0 000010 00  AX  0   0  8
  [14] .text             PROGBITS        000003f0 0003f0 000202 00  AX  0   0 16
  [15] .fini             PROGBITS        000005f4 0005f4 000014 00  AX  0   0  4
  [16] .rodata           PROGBITS        00000608 000608 000008 00   A  0   0  4
  [17] .eh_frame_hdr     PROGBITS        00000610 000610 00003c 00   A  0   0  4
  [18] .eh_frame         PROGBITS        0000064c 00064c 0000e8 00   A  0   0  4
  [19] .init_array       INIT_ARRAY      00001eec 000eec 000004 04  WA  0   0  4
  [20] .fini_array       FINI_ARRAY      00001ef0 000ef0 000004 04  WA  0   0  4
  [21] .jcr              PROGBITS        00001ef4 000ef4 000004 00  WA  0   0  4
  [22] .dynamic          DYNAMIC         00001ef8 000ef8 0000f0 08  WA  6   0  4
  [23] .got              PROGBITS        00001fe8 000fe8 000018 04  WA  0   0  4
  [24] .got.plt          PROGBITS        00002000 001000 000010 04  WA  0   0  4
  [25] .data             PROGBITS        00002010 001010 000008 00  WA  0   0  4
  [26] .bss              NOBITS          00002018 001018 000004 00  WA  0   0  1
  [27] .comment          PROGBITS        00000000 001018 00002d 01  MS  0   0  1
  [28] .symtab           SYMTAB          00000000 001048 000470 10     29  47  4
  [29] .strtab           STRTAB          00000000 0014b8 000263 00      0   0  1
  [30] .shstrtab         STRTAB          00000000 00171b 00010a 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)
```

图 2.2 示例程序的节头表

```
00001710  43 6c 6f 6e 65 54 61 62  6c 65 00 00 2e 73 79 6d  |CloneTable...sym|
00001720  74 61 62 00 2e 73 74 72  74 61 62 00 2e 73 68 73  |tab..strtab..shs|
00001730  74 72 74 61 62 00 2e 69  6e 74 65 72 70 00 2e 6e  |trtab..interp..n|
00001740  6f 74 65 2e 41 42 49 2d  74 61 67 00 2e 6e 6f 74  |ote.ABI-tag..not|
00001750  65 2e 67 6e 75 2e 62 75  69 6c 64 2d 69 64 00 2e  |e.gnu.build-id..|
00001760  67 6e 75 2e 68 61 73 68  00 2e 64 79 6e 73 79 6d  |gnu.hash..dynsym|
00001770  00 2e 64 79 6e 73 74 72  00 2e 67 6e 75 2e 76 65  |..dynstr..gnu.ve|
00001780  72 73 69 6f 6e 00 2e 67  6e 75 2e 76 65 72 73 69  |rsion..gnu.versi|
00001790  6f 6e 5f 72 00 2e 72 65  6c 2e 64 79 6e 00 2e 72  |on_r..rel.dyn..r|
000017a0  65 6c 2e 70 6c 74 00 2e  69 6e 69 74 00 2e 70 6c  |el.plt..init..pl|
000017b0  74 2e 67 6f 74 00 2e 74  65 78 74 00 2e 66 69 6e  |t.got..text..fin|
000017c0  69 00 2e 72 6f 64 61 74  61 00 2e 65 68 5f 66 72  |i..rodata..eh_fr|
000017d0  61 6d 65 5f 68 64 72 00  2e 65 68 5f 66 72 61 6d  |ame_hdr..eh_fram|
000017e0  65 00 2e 69 6e 69 74 5f  61 72 72 61 79 00 2e 66  |e..init_array..f|
000017f0  69 6e 69 5f 61 72 72 61  79 00 2e 6a 63 72 00 2e  |ini_array..jcr..|
00001800  64 79 6e 61 6d 69 63 00  2e 67 6f 74 2e 70 6c 74  |dynamic..got.plt|
00001810  00 2e 64 61 74 61 00 2e  62 73 73 00 2e 63 6f 6d  |..data..bss..com|
00001820  6d 65 6e 74 00 00 00 00  00 00 00 00 00 00 00 00  |ment............|
```

图 2.3 示例程序的节名字符串表

2.1.3 符号表

节.symtab 存储了程序中符号（symbol）的信息，称为符号表（symbol table）。在诸多符号表项目中，较为典型的符号类型包括全局变量和函数名称。存储的符号信息包括符号名称字符串表索引、符号类型、符号的值和符号大小等信息[12]，ELF 文件通过在符号表中记录相关符号信息定义一个符号。例如，图 2.4 是示例程序 ELF 文件的符号表信息。

```
Symbol table '.symtab' contains 71 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
    60: 00000590    93 FUNC    GLOBAL DEFAULT   14 __libc_csu_init
    61: 0000201c     0 NOTYPE  GLOBAL DEFAULT   26 _end
    62: 000003f0     0 FUNC    GLOBAL DEFAULT   14 _start
    63: 00000608     4 OBJECT  GLOBAL DEFAULT   16 _fp_hw
    64: 00002018     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
    65: 00000560    40 FUNC    GLOBAL DEFAULT   14 main
    66: 00000588     0 FUNC    GLOBAL HIDDEN    14 __x86.get_pc_thunk.ax
    67: 00000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
    68: 00002018     0 OBJECT  GLOBAL HIDDEN    25 __TMC_END__
    69: 00000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
    70: 0000039c     0 FUNC    GLOBAL DEFAULT   11 _init
```

图 2.4 示例程序的符号表（节选）

2.1.4 代码段和数据段

代码段的各个节中，最为重要的是.text 节（已编译的机器代码）和.rodata 节（只读数据）。.text 节中存放的是已经编译完成的机器代码,可以被加载器复制到内存中运行;.rodata 节存放的是一些只读的数据，如程序中使用到的字符串、常量等。

数据段由.data 节（已初始化的变量）和.bss 节（未初始化的变量）组成。其中，.data 节的数据会在执行程序时被加载器复制到内存中，而.bss 节仅记录变量的信息（如变量名、变量地址和变量大小等），并不在 ELF 文件中占据空间，而是在程序开始运行时由加载器从内存中为其分配空间。

2.2 程序生命周期

在 Linux 环境下,程序从写成的源代码开始,直到被送入内存执行,期间将依次经历（预）编译、汇编、链接和加载等四个过程。本节将就图 2.5 所示实例程序的 C 语言源代码而言，讨论程序在 Linux 环境下的一个完整的生命周期。

```
int main()
{
  int a = 1;
  int b = 2;
  return (a + b);
}
```

图 2.5 示例程序的 C 语言源代码

2.2.1 程序的编译

程序的编译包括预编译和编译两个阶段。预编译是在正式编译前，对源程序进行宏定义替换和外部文件引用等操作。在 Linux 下,可以使用命令(2.1)和(2.2)完成对程序的预编译，其中预编译结果（.i 文件）是一个经过宏替换和展开的文本文件。对于该预编译源代码，使用命令(2.3)或(2.4)即可完成编译操作，该过程包括一般编译步骤中的词法分析、语法分析和语义分析等过程，亦可在该过程中加入优化选项，要求编译器对生成代码过程进行编译优化。亦可以使用命令(2.5)一步完成源代码文件的编译。

```
gcc -E example.c -o example.i        (2.1)
cpp example.c -o example.i           (2.2)
gcc -S example.i -o example.S        (2.3)
```

```
ccl example.i -o example.S                    (2.4)
gcc -S example.c -o example.S                 (2.5)
```

编译阶段完成后，将生成一个以“.S”为扩展名的汇编语言文件（文本）。

2.2.2 程序的汇编

程序汇编过程的实质即是将以文本形式呈现的代码转换为二进制形式的机器代码，汇编过程的最终结果是一个 ELF 格式的可重定位目标文件。但是由于一个程序通常是多个程序模块的相互组合产生的，汇编过程中不可能完成每条指令和每个数据单元的最终地址的确定。各个程序模块之间的组合和重定位操作将是 2.2.3 节中讨论的内容。在 Linux 环境下，针对上一节所述的汇编文件，可以使用命令(2.6)或(2.7)完成对汇编语言文件的汇编操作，并生成一个以“.o”为扩展名的 ELF 可重定位目标文件。

```
gcc -c example.S -o example.o                 (2.6)
as example.S -o example.o                     (2.7)
```

2.2.3 程序的链接

程序的链接是指将程序的各个可重定位目标文件与库中的代码相互组合，确定各个指令和符号的最终存储器映像地址，生成 ELF 可执行文件的过程。程序的链接通常包括静态链接和动态链接两类。静态链接指的是将程序中所用到的函数及其相关数据以实体形式囊括到ELF可执行文件中，程序运行中调用各个函数时程序计数器总在本程序代码段中跳转。与之对应的是动态链接，即程序中只记录调用函数的动态链接符号名和被调用函数的动态链接地址，即可完成程序的链接，待程序运行时再载入相关的动态库函数。

通常来说，用户定义的函数采用静态链接的方式生成可执行文件，而库函数（标准库或其他的库）采用动态链接的方式。这样做的好处在于动态链接库通常可以被多个程序所复用，操作系统不需要重复加载相同的代码到内存中，常驻内存将提高内存的利用率，也有利于库的更新和升级。

在 Linux 环境下，可以使用命令(2.8)或(2.9)完成对单个目标文件的程序的链接操作，并生成一个 ELF 可执行目标文件。

```
gcc -o ./example example.o                    (2.8)
ld -o ./example example.o                     (2.9)
```

假如程序使用了外部的链接库，则需如命令(2.10)所示，用“-l”参数指定外部链接库

的名称。如命令(2.10)引用了一个名为“SDL2”的外部动态链接库。

```
gcc -o ./example example.o -lSDL2                    (2.10)
```

程序的链接过程中，允许开发人员使用参数“-T”指令.text 段、.data 段、.bss 段主动指令这三个段（或其中的某个段）的链接地址，如命令(2.11)则要求链接器将.text 段加载到指定位置 0x6000000，而不是加载到默认的存储器映像地址（i386 下通常为 0x8048000）。

```
gcc -o example.o -Ttext=0x6000000                    (2.11)
```

2.2.4 程序的加载

在 Linux 下，我们通常使用一个命令

```
./example
```

命令操作系统执行给定的可执行文件，亦可在执行时附加额外的字符串作为程序的参数：

```
./example argu1 argu2 ...
```

这个过程中，程序将根据链接后确定的各个符号和指令的最终存储器映像地址，将程序加载到虚拟地址空间中的对应位置上，并完成程序运行时环境的初始化（如设置堆栈和载入动态链接库等）操作。初始化完毕后，操作系统将为执行的程序分配执行进程，参与操作系统的各项调度机制直至程序主动退出运行或被杀死。

2.3 X86 程序规范

X86 为英特尔公司所开发的 80286、80386、80486 等等基于 i386 体系的处理器的统称，在本文中认为 x86 和 i386 名称等价，为 32 位的指令集体系架构，区别于英特尔公司后来提出的 x86_64 的 64 位体系架构。

英特尔公司于 1986 年发布了《Intel 80386 Programmer’s Reference Manual》一书作为最为权威的官方指令集文档[13]，定义了 x86 下寄存器的组织、指令集细节等详细信息。

2.3.1 通用寄存器组

在 x86 体系中，有 8 个通用寄存器（general registers）和 4 个段寄存器（segment registers）允许程序员编程使用，另外包括一个指令指针（instruction pointer，即程序计数器）和一个标志寄存器（system flags）。他们的名称如图 2.6 所示。

现代操作系统中，通常采用“扁平”的保护模式，即使所有段寄存器指向某个相同的位置，设置段的大小为整个内存的大小，相当于各个段拥有完全相同的地址空间。

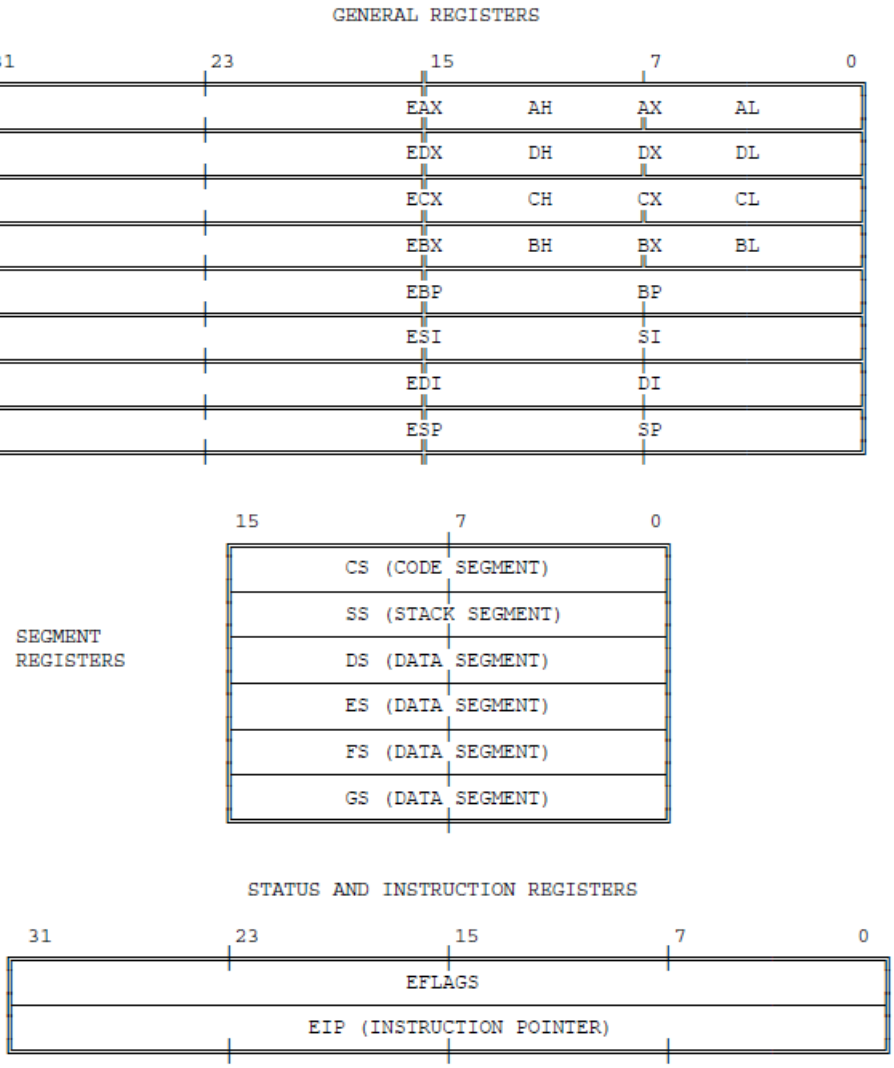

图 2.6 x86 体系下的寄存器体系

2.3.2 指令集

根据文档，x86 体系下一共有百余条功能不同的指令（不包括 x87 浮点指令），指令形式多样，构成要素复杂。其指令的形式如图 2.7 所示。

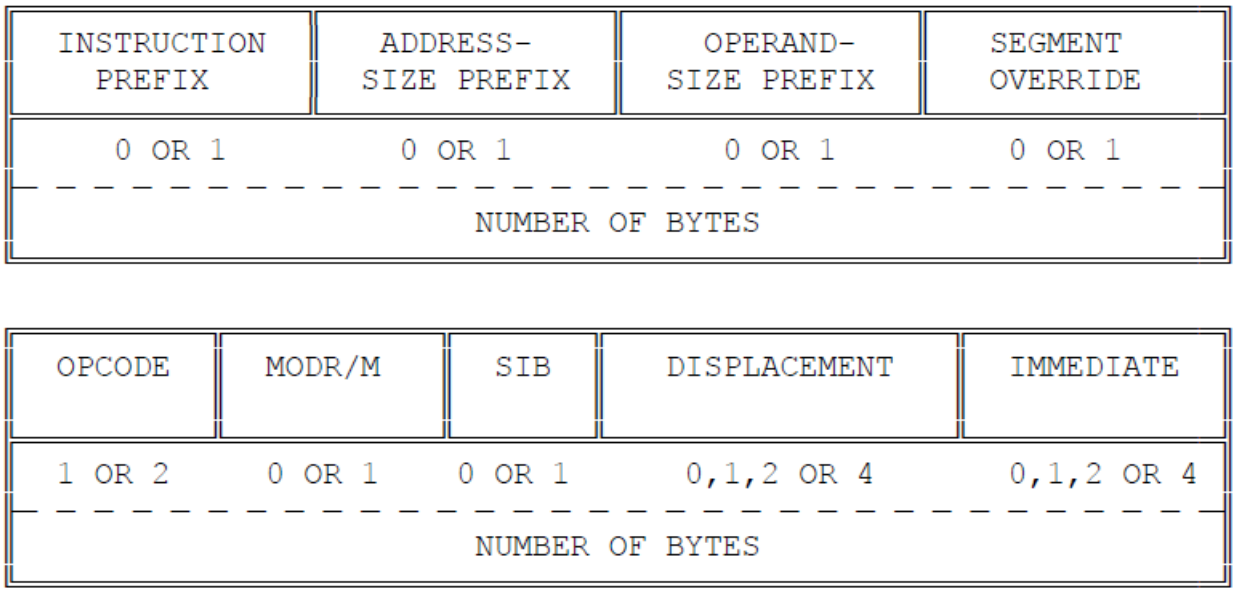

图 2.7 i386 指令格式

根据 x86 指令集格式可以得知，x86 下一条指令的解码将经历极其复杂的步骤，包括译码过程中从内存取数据（即指令长度不定）、解析 ModR/M 字节、解析 SIB 字节等，译码效率相对低下。

2.3.3 分支和循环结构

根据 x86 官方手册和 System V ABI[14]，程序中的分支结构和循环结构通过 cmp、test 等指令设置标志寄存器相关标志位（如借位标志、溢出标志、零标志、符号标志等），之后使用 jmp、jcc 系列指令跳转到当前函数（或过程）中的某个位置实现分支结构和循环结构。

2.3.4 过程调用

根据 x86 官方手册和 System V ABI[14]，程序中的过程调用和函数调用将满足指令集规范。规范要求，程序的参数一律由调用者依次存入栈空间，并保存返回后的程序计数器的值。被调用者则要求在进入时设置自己的栈空间，包括更新栈底指针（ebp）和栈顶指针（esp），并保护将要用到的寄存器的值，以便在返回时恢复，避免影响调用者函数的运行。图 2.8 中列举出了根据上述调用过程调用函数或过程所形成的一个函数栈帧的结构。

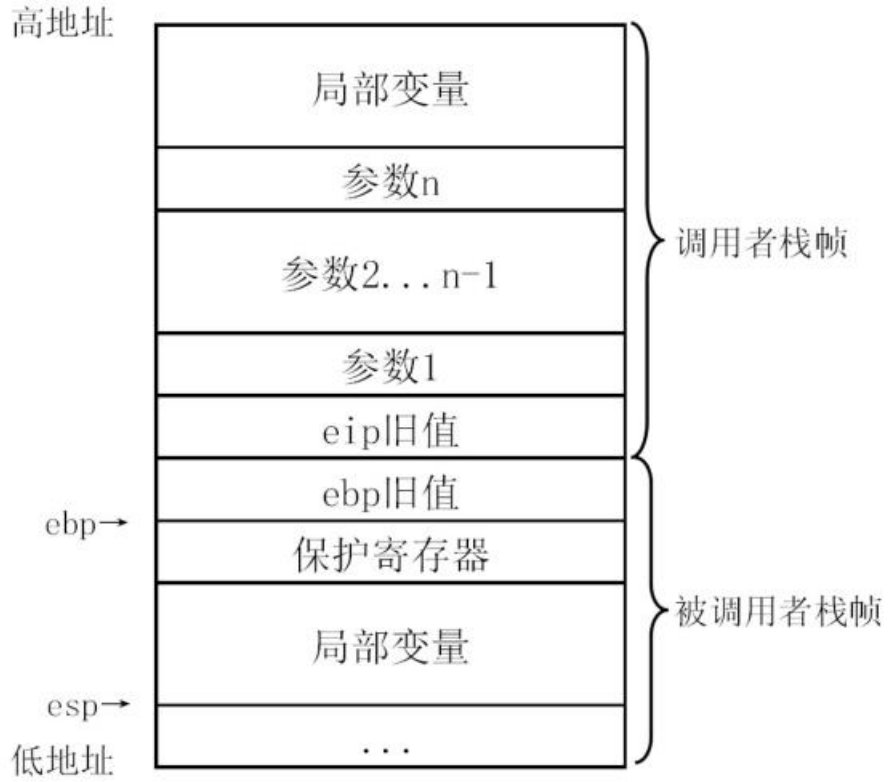

图 2.8 x86 中一次函数调用所形成的栈帧结构

2.4 MIPS 程序规范

从 MIPS 官方网站[15]上可以得到 MIPS32 指令集的官方指令集手册，和 80386 手册类似，记录有 MIPS32 指令集的指令集细节[16]和编程接口规范[17]等相关信息。

2.4.1 通用寄存器组

MIPS 体系下含有 32 个用户可编程使用的寄存器，他们的名称和作用如图 2.10 所示[19]。

Register Name	Software Name (from regdef.h)	Use and Linkage
$0		Always has the value 0.
$at		Reserved for the assembler.
$2..$3	v0-v1	Used for expression evaluations and to hold the integer type function results. Also used to pass the static link when calling nested procedures.
$4..$7	a0-a3	Used to pass the first 4 words of integer type actual arguments, their values are not preserved across procedure calls.
$8..$15	t0-t7	Temporary registers used for expression evaluations; their values aren't preserved across procedure calls.
$16..$23	s0-s7	Saved registers. Their values must be preserved across procedure calls.
$24..$25	t8-t9	Temporary registers used for expression evaluations; their values aren't preserved across procedure calls.
$26..27 or $kt0..$kt1	k0-k1	Reserved for the operating system kernel.
$28 or $gp	gp	Contains the global pointer.
$29 or $sp	sp	Contains the stack pointer.
$30 or $fp	fp	Contains the frame pointer (if needed); otherwise a saved register (like s0-s7).
$31	ra	Contains the return address and is used for expression evaluation.

图 2.10 MIPS 体系下的寄存器体系

2.4.2 指令集

MIPS 指令集所有指令长度均为 32 比特，可以根据指令操作数种类和行为分为为立即数型（I-Type）、跳转型（J-Type）和寄存器型（R-Type）三种。三类指令格式如图 2.11 所示[4]。

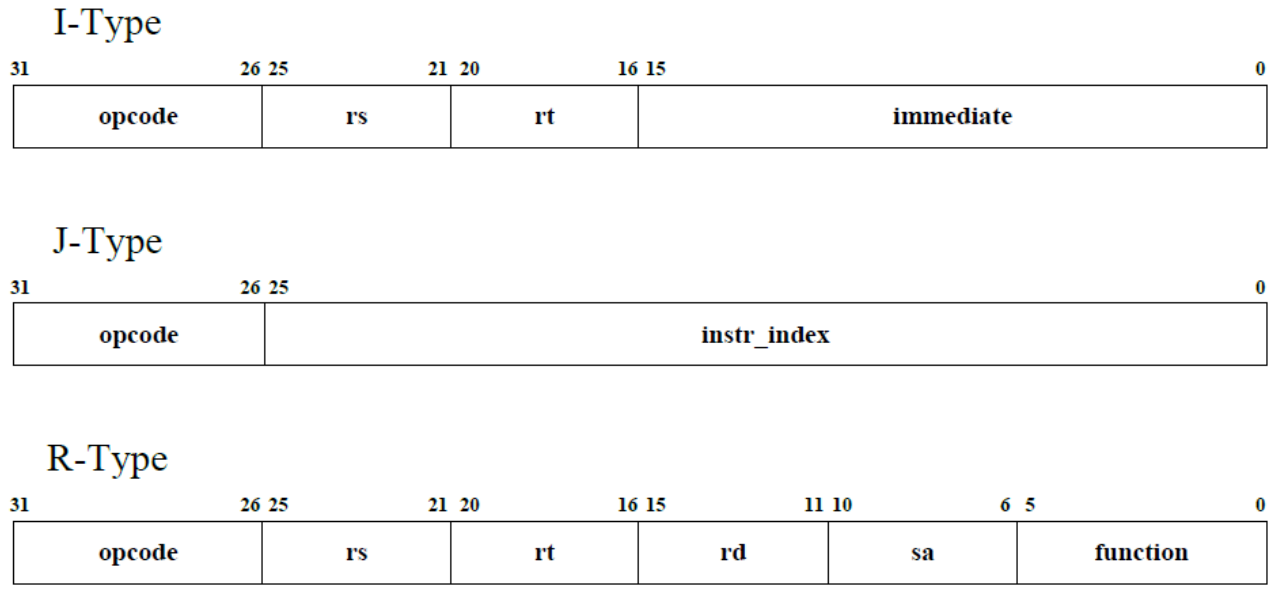

图 2.11 MIPS 指令格式

2.4.3 分支和循环结构

MIPS 体系下没有标志寄存器，因此程序非顺序结构的流程控制 SLT 等条件设置指令根据运算结果设置某个寄存器的值，此后采用 B 系列条件跳转指令（如 BEQ、BNE 等）对于给定寄存器进行检验，若符合本指令跳转条件则转移到给定偏移处。同时，也采用了 J 系列无条件跳转指令（如 J、JAL 等）跳转到目标地址后执行指令序列[4]。

2.4.4 过程调用

MIPS 体系下的过程调用通过 J 系列指令（无条件直接跳转指令）进行函数（或过程）之间的转移，包括从调用者跳转到被调用者，亦包括从被调用者转移回调用者。调用者在调用函数前，将装载函数的参数值，其中前 4 个参数依次保存于 a0-a3 这四个寄存器中，并在栈中为其保留相应长度的空间，第 5 个参数起则依次保存在栈中。参数保存完毕后则跳转到被调用者的首地址继续执行指令。进入被调用者后，首先由被调用者保存返回地址和需要使用的 sx 寄存器，随后由被调用者逐渐形成自己的栈帧。MIPS 体系下的函数调用栈帧结构如如图 2.12 所示[19]。

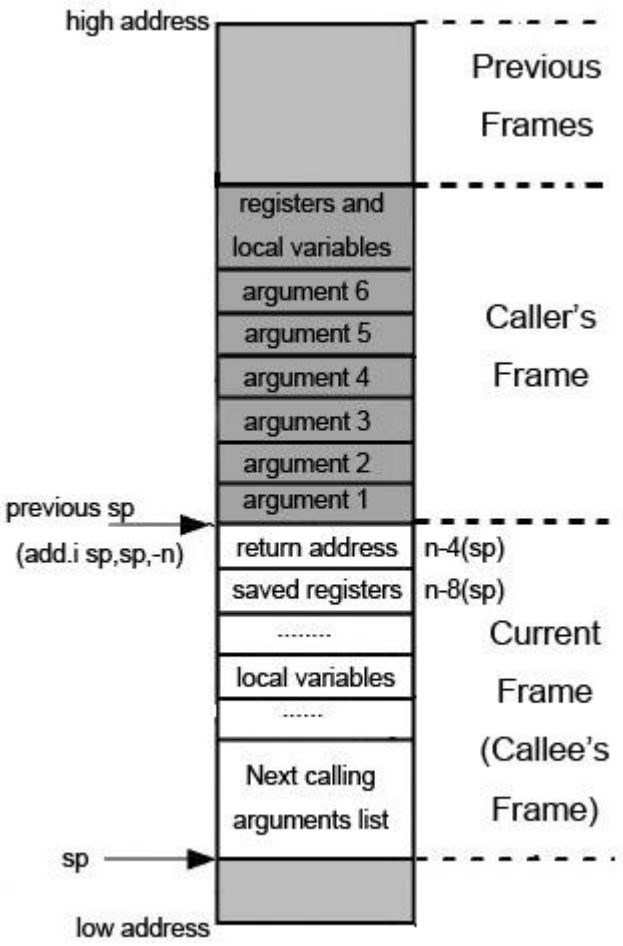

图 2.12 MIPS 的函数栈帧

2.5 SDL2 图形库

2.5.1 SDL2 图形库概述

SDL 的全称为简易直接媒体层[18](Simple DirectMedia Layer)，是一个跨平台的开发库，用于向应用程序提供包括音频、键盘、鼠标、游戏手柄和基于 OpenGL 和 Direct3D 的图形硬件的底层访问接口。SDL 被广泛利用与视频播放软件、模拟器和各种游戏开发中。SDL 库使用 C 语言写成，在硬件上的运行效率较高。

SDL 的最新发行版为 SDL1.2，此后 SDL 官方发布了其 SDL 的第二个大版本，即 SDL2。SDL2 和 SDL1.2 相比，最显著的变化在于其代码简洁高效，编程理念清晰易用。本文使用了一个基于 SDL2 的测试用例，用以测试本文所提出的二进制翻译器对于来自外部的动态链接库的库函数的翻译。

2.5.2 基于 SDL2 图形库的编程

图 2.13 中列举了一个简单的基于 SDL2 的应用程序，用以显示出一张静态图片，其运行结果如图 2.14 所示。

```
#define SDL_MAIN_HANDLED

#include "SDL2/SDL.h"
#include <stdio.h>

int main(int argc, char *args[])
{
  const int Window_WIDTH = 640;
  const int Window_HEIGHT = 480;
  char error[20] = "error! %d\n";
  char title[20] = "hello!";
  char file[20] = "hello.bmp";
  char mode[10] = "rb";

  SDL_Window* window = NULL;
  SDL_Renderer* render = NULL;
  SDL_Texture* tex = NULL;

  //init
  window = SDL_CreateWindow(title,
    SDL_WINDOWPOS_UNDEFINED,
    SDL_WINDOWPOS_UNDEFINED,
    Window_WIDTH,
    Window_HEIGHT,
    SDL_WINDOW_SHOWN);
  if (window == NULL)
  {
    printf(error, 1);
    return 1;
  }
  render = SDL_CreateRenderer(window, -1, 0);
  if (render == NULL)
  {
    printf(error, 2);
    return 1;
  }

  //load image
  SDL_Surface* hello = NULL;
  puts(file);
```

```
  hello = SDL_LoadBMP_RW(SDL_RWFromFile(file, mode), 1);
  if (hello == NULL)
  {
    printf(error, 3);
    return 1;
  }
  tex = SDL_CreateTextureFromSurface(render, hello);
  SDL_FreeSurface(hello);
  if (tex == NULL)
  {
    printf(error, 4);
    return 1;
  }

  //put image
  SDL_RenderCopy(render, tex, NULL, NULL);
  SDL_RenderPresent(render);
  SDL_Delay(1000);

  //quit
  SDL_DestroyWindow(window);
  SDL_DestroyRenderer(render);
  SDL_DestroyTexture(tex);
  SDL_Quit();

  return 0;
}
```

图 2.13 一个简单的 SDL2 应用程序

图 2.14 简单 SDL2 应用程序的运行结果

2.6 本章小结

本章主要介绍了一些在本文所设计的 x86-MIPS 二进制翻译器的过程中将用到的一些相关技术和概念以及部分测试用例中所涉及的图形库 SDL2。在本文余下的内容中，将结合本章所提及的技术和概念，进行综合性的应用。

第三章　二进制翻译器的设计与实现

3.1　X86 二进制程序解析

3.1.1　X86 二进制程序解析简述

翻译器运行伊始，需要对待翻译的源文件进行代码和符号位置分析，该过程需要按照一般 ELF 文件信息过程进行。即如图 3.1 所示的翻译过程，对 ELF 可执行文件进行解析、获取.text 节地址，并从该节中解析出所有用户函数的起始位置和长度，供下一步翻译环境使用。

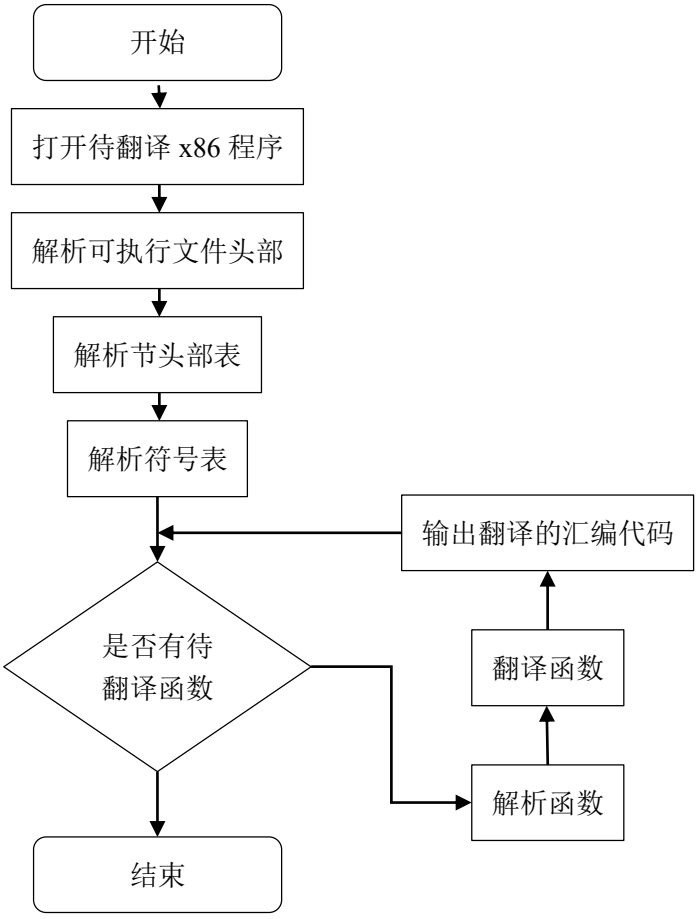

图 3.1 X86-MIPS 二进制程序翻译流程

3.1.2　ELF 可执行文件头部解析

ELF 可执行文件的头部魔数（magic number）规定为“.ELF”，即十六进制下的“0x7f 0x45 0x4c 0x46”。翻译器通过读入一个 Elf32_Ehdr（ELF 头部数据结构，在 elf.h 中定义）大小的头部信息，并逐字节比对 4 个字节的魔数是否对应，若对应则说明本文件是一个 ELF 文件，可以进行下一步解析工作，否则翻译器将报错并中止运行。

3.1.3 节头部表解析

ELF 文件头部中，有一个数据成员 e_shoff 指示着节头部表在 ELF 文件中的存储位置（section headers offset），以及另一个数据成员 e_shnum 指示着节头部表中的项目数量。翻译器需要根据 e_shnum 所记载的数量读取 e_shnum 个表项单位大小的数据，存储为节头部表信息。

读取节头部表完成后，需要确定每个表项所对应的节名称。根据上一章节所述，每个节的名称存储在字符串表中。由于目前尚不知晓字符串表所在的节头部表项为哪一个，此处则根据节头部表中所指示 e_shstrndx 数据成员确定节头部字符串表表项的索引（section headers string index），从该表项中获取字符串表的起始位置和总长度，并从文件中存储到翻译器内部以备后续使用。

获取到表项名称后，就可以遍历整个节头部表，得到每个节的索引、名称、类型、链接地址、文件内偏移和节大小等信息。

3.1.4 符号表解析

读入程序的节头部表之后，就可以很容易地定位到符号表和字符串表的表项。遍历符号表即可读入程序中所有符号的各项信息，这些信息包括：编号、值、大小、类型、索引和名称。符号表读入完成后，程序中所有全局变量和函数的名称信息就完全被翻译器所掌握，ELF 文件信息的解析也就此告一段落。

3.1.5 函数解析

函数解析的过程和翻译过程是同时进行的，本节只讨论函数的解析过程，具体每个指令的翻译模式将在下一节中详细介绍。

本翻译器采用一个栈结构（称为待翻译函数栈）维持程序中用户函数的翻译过程。一般来说，用户函数（区别于动态链接的库函数）都是静态存储在 ELF 可执行文件中的，可以在扫描某个函数时通过其中的函数调用指令 call 得知调用了哪些函数。具体的栈式函数解析过程如算法 3.1 所示。

算法 3.1（栈式函数解析算法）：

（1） 初始化目标指令集汇编代码头部；

（2） 将“main”函数压入待翻译函数栈中；

（3） 判断待翻译函数栈是否为空，若为空则进入第（12）步，否则进入第（4）步；

（4） 从待翻译函数栈中弹出位于栈顶的函数名；

（5） 判断该函数是否已经被解析，若是则进入第（3）步，否则进入第（6）步；

（6） 从符号表中取得当前函数的符号信息，调用 objdump 工具产生其 x86 反汇编代码；

（7） 向目标指令集汇编代码中输出函数头部代码；

（8） 扫描并翻译该函数，具体算法如算法 3.2 所示；

（9） 向目标指令集汇编代码中输出函数尾部代码；

（10）将该函数状态记录为已解析；

（11）当前函数解析完成，进入第（3）步；

（12）算法结束。

算法 3.2（当前函数扫描算法）：

（1） 打开当前函数的反汇编代码文件；

（2） 从文件中读入一行反汇编代码；

（3） 若当前文件已经读取完毕，进入第（7）步，否则进入第（4）步；

（4） 将读入的一行反汇编代码的各个部分分离（指令地址、指令名、指令参数、额外参数）；

（5） 若该指令是 x86 下的 JMP 或 JCC 系列跳转指令，则将指令参数所指示目标跳转地址放入标号表中，并生成该跳转地址的唯一标号；

（6） 进入第（2）步；

（7） 重新打开当前函数的反汇编代码文件；

（8） 从文件中读入一行反汇编代码；

（9） 若当前文件已经读取完毕，进入第（13）步，否则进入第（10）步；

（10）将读入的一行反汇编代码的各个部分分离；

（11）将分离后的指令及其及其参数传入翻译控制算法中，具体算法如算法 3.3 所示；

（12）进入第（9）步；

（13）算法结束。

算法 3.3（翻译控制算法）：

（1） 根据当前指令地址获取当前地址标号；

（2） 若当前指令为 CALL 指令，进入下一步，否则进入第（6）步；

（3） 判定当前指令调用的函数是否在.plt 段中，若是则进入第（5）步；

（4） 将当前翻译指令调用的函数名压入待翻译函数栈；

（5） 翻译当前指令（见算法 3.4，参数为当前地址标号、当前指令、指令参数和附加参数），算法结束；

（6） 若当前指令为 JMP 或 JCC 系列跳转指令，进入下一步，否则进入第（9）步；

（7） 根据当前跳转指令的跳转地址获取目标跳转点的唯一地址标号；

（8） 将该标号作为额外参数翻译当前指令，算法结束；

（9） 翻译当前指令，算法结束。

由于用户定义的函数通常存放于.text 段，而动态链接的库函数通常只存放一个动态链接标志在.plt 段。翻译器在算法 3.3 中将通过对函数调用指令（CALL）所调用的函数名从两个段中进行搜索，确定函数的种类（用户函数或库函数），并采取不同的方式相应。对于用户函数，需要由翻译器进行二进制转换，因此需要将扫描到的用户函数加入待翻译函数栈中。相反地，对于库函数，则仅仅只需记录其名称，将其名称直接对应到目标指令集中的函数调用指令即可。从效率上考量，翻译器不对库函数进行翻译，而是令转换后的 MIPS 程序直接动态调用库函数。

3.2 翻译模式

3.2.1 指令翻译流程

本节将详细介绍指令的翻译流程，该流程如算法 3.3 所示。其中每条待翻译的指令的参数包括：当前指令的标号（若有）、当前指令名（可能含后缀）、指令参数、额外参数（通常是 CALL 指令的被调用函数名或 JMP/JCC 指令的目标跳转地址）。

本翻译器的实现中，每条 x86 指令均对应一个翻译工人函数（translating worker），其参数列表为：

（1）数据传送模式，以字符串形式给出源操作数和目标操作数，可能是 0 位、1 位或 2 位，各种组合的意义如表 3.1 所示；

表 3.1 数据传送模式组合

数据传送模式	意义
（无）	该指令无特定的数据传送模式
i	仅一个操作数，为立即数
r	仅一个操作数，为寄存器名称
m	仅一个操作数，为内存地址
ir	源操作数为立即数，目标操作数为寄存器名称
rr	源操作数和目标操作数均为寄存器名称
rm	源操作数为寄存器名称，目标操作数为内存地址
mr	源操作数为内存地址，目标操作数为寄存器名称

（2）指令参数；

（3）指令后缀，可能是 0 位、1 位或 2 位，可能的指令后缀形式如表 3.2 所示。

表 3.2 可能的指令后缀形式

指令后缀	意义	举例
（无）	该指令不需要通过指令后缀区分数据宽度	call, jmp, jne, mov, cmp, ret
b	该指令操作数据宽度为8位（字节）	movb
w	该指令操作数据宽度为16位（字）	movw
l	该指令操作数据宽度为32位（双字）	movl
bl	该指令操作数据宽度为从8位到32位	movzbl

算法 3.4（指令翻译算法）：

（1） 若给予了本指令跳转标号，则输出到汇编文件中；

（2） 分割参数并得到参数个数和每个参数的类型（具体到源参数类型和目标参数类型）；

（3） 若有额外参数，则将额外参数放入翻译工人参数列表首位；

（4） 按无指令后缀匹配指令名称，若匹配上则传入对应工人函数中翻译；

（5） 按 1 个字母长度后缀匹配指令名称，若匹配上则传入对应工人函数中翻译；

（6） 按 2 个字母长度后缀匹配指令名称，若匹配上则传入对应工人函数中翻译；

（7） 若（4）至（6）步中均未成功匹配，说明不支持本条指令，算法结束并报错；

（8） 算法结束。

3.2.2 寄存器的翻译

X86 体系和 MIPS 体系下寄存器设计情况如第二章所述，可以看出两个体系下寄存器差异巨大。其差异不仅仅体现在数量差异较大，且 MIPS 不含有段寄存器，MIPS 拥有 t0-t9 等 10 个可以任意使用且不必保护的临时寄存器，而 x86 下所有寄存器在使用前都应保存，然这一点又和 MIPS 下 s0-s7 等 8 个寄存器特性相似。

鉴于上述特性，本文计划采用表 3.3 所示方案对寄存器进行对应。

表 3.3 两体系寄存器的对应方案

X86寄存器	MIPS寄存器	说明
eax	s0	-
ecx	s1	-
edx	s2	-
ebx	s3	-
esp	sp	-
ebp	s5	栈底指针
esi	s6	-
edi	s7	-
eip	pc	程序计数器
eflags	-	不模拟标志寄存器
cs	-	MIPS无需使用段寄存器
ds	-	
ss	-	
ds	-	
es	-	
gs	-	
-	t0-t9	可任意使用的临时寄存器
-	ra	返回地址寄存器
-	fp	栈帧指针
-	zero	恒零寄存器
-	v0, v1	子函数调用返回的结果

	a0-a3	子函数调用的前4个参数

3.2.3 寻址的翻译

X86 使用了多种寻址方式，X86 和 MIPS 的寻址方式对应方案如表 3.4 所示。

表 3.4 X86 和 MIPS 寻址方式的对应

X86寻址方式	对应MIPS寻址方式	说明
立即寻址	立即寻址	直接给出立即数
寄存器寻址	寄存器寻址	直接给出寄存器
位移	基址寄存器加偏移量	位移放入基址寄存器，偏移为0
基址寻址		基址放入基址寄存器
基址加位移		根据x86的寻址方式计算出目标地址并放入某个临时寄存器作为基址寄存器，将偏移量设置为0
比例变址加位移		
基址加变址加位移		
基址加比例变址加位移		
相对寻址		

3.2.4 函数内跳转的翻译

函数内跳转在 x86 体系中通常是使用 JMP 指令（无条件跳转）和 JCC 系列指令实现的（有条件跳转）。其中，有条件跳转的实现方式为通过 CMP 等指令做比较运算，并设置标志寄存器相关位。随后紧跟一条 JCC 系列指令，验证是否满足本条 JCC 指令的跳转条件，满足则跳转，不满足则执行下一条指令。然而，本翻译器在设计时并未实现标志寄存器的模拟，有以下原因：

（1） 模拟标志寄存器取特定位需要额外生成算术运算指令进行位操作模拟，效率不高；

（2） 其他所有涉及标志寄存器指令在翻译时均需执行额外的 MIPS 指令以设置特定位；

（3） MIPS 原生程序中并不需要标志寄存器即可实现数学或逻辑条件的比较。

表 3.5 给出了 x86 下具体每个 JCC 指令所对应的跳转条件，以及对应的翻译方案。

表 3.5 跳转指令翻译方案

X86指令	跳转条件	对应MIPS指令	说明
cmp	-	sub	作差，结果存入$t8
ja/jnbe/jg/jnle	大于	bgtz	$t8>0时跳转

jnc/jae/jnb/jge/jnl	大于或等于	bgez	$t8>=0时跳转
jc/jb/jnae/jl/jnge	小于	bltz	$t8<0时跳转
jbe/jna/jle/jng	小于或等于	blez	$t8<=0时跳转
jz/je	等于	beq	令rt=$zero，$t8=0时跳转
jnz/jne	不等于	bne	令rt=$zero，$t8≠0时跳转

按照上表的对应关系，可以很方便地根据两数相减的结果选择对应指令进行相应条件的跳转，且满足指令间无关原则，不需成对翻译，仅约定将作差结果存入$t8 中，跳转指令也使用$t8 进行判断、当遇到某条反汇编时可快速翻译成对应 MIPS 指令。

3.2.5 过程调用的翻译

本翻译器在处理函数调用时，根据被调用函数的性质不同，采用不同的翻译方案。

对于用户函数，根据指令间无关原则，各函数在进入后将继续按 x86 模式进行 ebp 保护和各个通用寄存器保护等措施，寻址也有通过 esp 或 ebp 来进行栈内寻址。通过寄存器对应，使 ebp 寄存器对应$s5，esp 寄存器对应$sp，使用这两个寄存器对用户的栈空间进行管理。由于用户函数均是套用 x86 栈管理模式，传递参数仿照 x86 模式，一律通过栈来传递参数，返回值则无关翻译 x86 指令，依据 x86 模式存储到代表 eax 寄存器的$s0 中带回。

考虑到 x86 和 MIPS 都需要保存返回地址，虽然返回地址的使用方法有所差异，但为了使 MIPS 的 x86 形式的参数寻址和 x86 保持一致，在调用 jal 指令前将$ra 压栈存储，返回后从栈中弹出。该对应关系中，为了使 x86 模式和 MIPS 参数传递模式一致，强制将$sp 起的 4 个 4 字节空间内容分别存入$a0 到$a3 寄存器中，保存$ra 的内容到栈中，以便返回后恢复其状态。调用库函数则相对麻烦，这是由于我们需要人工处理拟 x86 管理模式到返回原生 MIPS 管理模式。

函数调用的翻译关系如表 3.6 所示。

表 3.6 函数调用的翻译模式

X86指令	对应MIPS指令序列	说明
call func (位于.text段的用户函数)	li $a0,0($sp) li $a1,4($sp) li $a2,8($sp) li $a3,12($sp)	将$sp所指示地址起连续的4个4字节数据存入a0-a3中

	addi $sp,$sp,-4	将当前返回地址$ra保存到栈中
	sw $ra,0($sp)	
	.option pic0	跳转到func的入口地址处
	jal func	
	.option pic2	
	li $ra,0($sp)	从栈中弹出被保护$ra的值
	addi $sp,$sp,4	
call func (位于.plt段的动态链接函数)	li $a0,0($sp)	将$sp所指示地址起连续的4个4字节数据存入a0-a3中
	li $a1,4($sp)	
	li $a2,8($sp)	
	li $a3,12($sp)	
	lui $gp,%hi(__gnu_local_gp)	设定全局指针
	addiu $gp,$gp,%lo(__gnu_local_gp)	
	move $s4,$ra	使用$s4急促请你保护返回地址$ra
	lw $v0,%call16(func)($gp)	跳转到func的入口地址处
	move $t9,$v0	
	.option pic0	
	jal $t9	
	.option pic2	
	move $ra,$s4	还原被保护的返回地址$ra的值
	move $s0,$v0	将函数返回值送入对应$eax的$s0

值得一提的是，本翻译器只对用户函数进行二进制级别的翻译。对于库函数则仍令程序在翻译后采用动态调用的方式，这样做的好处在于可以充分利用某些跨平台的库，不必使用额外的翻译开销翻译库函数（也没有必要这样做），是一种运行效率和正确性上的考量。

3.2.6 指令翻译模式

本翻译器在设计时，把握“指令间无关”原则，即每一条指令的不同形式分别对应着一段特定的翻译序列，翻译时不会因为其前方指令不同而改变翻译策略。这样做的优点在于解除耦合，便于程序的维护和二次开发，具体发展前景将在第五章中讨论。

由于本翻译器采取的翻译策略为直接用 MIPS 指令表出 x86 指令，因此存在两体系间细节部分代码行为的差异。本翻译器设计时把握“进出一致”原则，将程序运行空间划分为系统区和用户区分别管理，其中系统区的定义为：程序在进入 main 函数之前的运行环境初始化阶段、程序退出 main 函数之后的运行环境销毁阶段和库函数内部。相对地可以提出用户区的定义，即：程序在用户所定义的函数的内部。

在用户区和系统区之间转换时，程序需要保证“进出一致”，即从系统区进入用户区时的状态必须和对应的从用户区返回到系统区的状态相同，此处所述的状态包括：栈空间数据排布、MIPS 下要求保护的寄存器和系统状态寄存器（如 status、ra、fp、sp、gp）等。实验证明，只要保证“进出一致”，翻译后的程序将不会出现系统错误或运行时错误。

3.3 MIPS 目标程序生成

3.3.1 代码的生成

根据 3.2 节所述翻译流程和翻译模式，翻译器将输入的 x86 程序的用户函数逐个翻译并以对应的函数名的 MIPS 汇编代码的形式存入输出文件中，该文件是一个 MIPS 下的汇编代码文件，可以直接使用 GCC 等编译汇编工具进行汇编和链接。

3.3.2 数据的转移

由于程序中存储的数据和平台无关，因此对于数据段和只读数据段中的数据，均采用直接将整个段的二进制数据原封不动地复制到 MIPS 汇编代码文件中，使用汇编伪指令将数据定义到对应段中。若输入的 x86 程序中含有数据段数据，翻译器将计算出翻译后 MIPS 程序的目标数据段链接地址并显式告知用户，用户需要根据给出的数据段链接地址对汇编后的程序进行链接以保证数据位置对应。

3.3.3 目标程序的汇编

经本翻译器翻译后的 MIPS 程序直接以汇编代码（.S 文件）形式输出存储，用户只需通过在 MIPS 环境下使用类似命令 3.1 或 3.2 的 Linux 命令对汇编代码进行汇编，生成可 ELF 可重定位目标文件。

```
gcc -c example.S -o example.o        (3.1)
as example.S -o example.o            (3.2)
```

3.3.4 目标程序的链接

经过上述汇编，已经可以得到 ELF 格式可重定位目标文件，若要将该可重定位目标文件

链接生成可执行文件，则需要使用适当链接参数显式指定数据段链接地址（gcc -Tdata=address）。若输入的 x86 程序中调用了非 C 标准库，则还需要显式指定链接库名称（gcc -lname）并保证当前系统中已经妥善安装了该库以便动态调用。如命令 3.3 所示，该命令行用于汇编链接一个调用了外部 SDL2 库，并具有数据段（数据段计算地址为 0x8049058）的 x86 程序。

```
gcc -o ./example example.o -Tdata=0x8049058 -lSDL2          (3.3)
```

链接结束后，将生成目标程序“./example”，可以直接使用命令 3.4 执行。

```
./example                                                   (3.4)
```

至此，一个程序的翻译过程全部结束。

3.4 本章小结

本章节中主要描述了本翻译器的完整翻译流程和翻译策略，介绍了翻译过程中需要使用者进行的外部操作。本翻译器具有 x86 可执行文件的解析功能和 x86-MIPS 的指令翻译功能，并能够将解析的段地址告知用户，便于用户根据指示进行下一步链接操作。本翻译器采取的翻译策略为用户区代码采用翻译器翻译，系统区代码则构造入口参数后调用，保证了翻译后程序的运行效率。

第四章　实验与结果分析

4.1　整体系统仿真实验

本节将采用 QEMU 模拟器模拟出 MIPS 体系架构的模拟计算机进行仿真实验，仿真环境如表 4.1 所示。

表 4.1 仿真实验运行环境

项目	值
QEMU版本	2.5.0
QEMU模拟架构	qemu-system-mipsel
虚拟CPU核数	1
虚拟内存大小	2GB
模拟器操作系统	Debian 8
操作系统内核版本	Linux debian 3.16.0-6-4kc-malta

4.1.1　顺序结构程序翻译

顺序结构为程序设计中最为简单的一种结构，本节将使用图 4.1 所示的 C 语言编写的测试用例甲进行翻译测试，甲程序中只使用了顺序结构，将一次加法结果利用 main 函数的返回值带回操作系统，再使用特定命令显示该结果。甲程序经过编译后的 x86 可执行文件的反汇编代码如图 4.2 所示。

```
int main()
{
  int a = 1;
  int b = 2;
  return (a + b);
}
```

图 4.1 甲程序源程序代码

```
080483db <main>:
 80483db: 55                    push   %ebp
 80483dc: 89 e5                 mov    %esp,%ebp
 80483de: 83 ec 10              sub    $0x10,%esp
 80483e1: c7 45 f8 01 00 00 00  movl   $0x1,-0x8(%ebp)
 80483e8: c7 45 fc 02 00 00 00  movl   $0x2,-0x4(%ebp)
 80483ef: 8b 55 f8              mov    -0x8(%ebp),%edx
 80483f2: 8b 45 fc              mov    -0x4(%ebp),%eax
 80483f5: 01 d0                 add    %edx,%eax
 80483f7: c9                    leave
 80483f8: c3                    ret
 80483f9: 66 90                 xchg   %ax,%ax
 80483fb: 66 90                 xchg   %ax,%ax
 80483fd: 66 90                 xchg   %ax,%ax
 80483ff: 90                    nop
```

图 4.2 甲程序 x86 反汇编代码

将该程序的 x86 可执行文件作为二进制翻译器的输入文件进行翻译，得到其 MIPS 下的汇编语言文件，其内容如图 4.3 所示。

```
 .text
  .globl main
  .ent main
  .type main, @function
main:
  addi  $sp,$sp,-4
  sw  $s5,0($sp)
  move  $s5,$sp
  li  $t8,-16
  add  $sp,$sp,$t8
  li  $t9,-0x8
  addu  $t9,$s5,$t9
  li  $t8,0x1
  sw  $t8,0($t9)
  li  $t9,-0x4
  addu  $t9,$s5,$t9
  li  $t8,0x2
  sw  $t8,0($t9)
  li  $t9,-0x8
  addu  $t9,$s5,$t9
  lw  $s2,0($t9)
  li  $t9,-0x4
  addu  $t9,$s5,$t9
  lw  $s0,0($t9)
  add  $s0,$s2,$s0
  move  $sp,$s5
  lw  $s5,0($sp)
  addi  $sp,$sp,4
  move  $v0,$s0
  jr  $ra
.end main
```

图 4.3 翻译后的甲程序汇编代码

对上述汇编语言在 MIPS 仿真环境中使用图 4.4 中的命令进行汇编和链接，得到 MIPS 下的可执行文件，该可执行文件的执行情况如图 4.4 所示。

图 4.4 甲程序在仿真环境下的运行情况

如图 4.4 所示，程序正常运行至程序退出，未出现任何系统异常，且返回操作系统的值正确。甲程序翻译情况符合实验预期。

4.1.2 分支循环结构程序翻译

分支循环结构程序主要用于测试 x86 下跳转指令的翻译情况，本节将使用图 4.5 所示的 C 语言编写的测试用例乙进行翻译测试，乙程序中使用一个循环结构求得一个等差数列的累加值，将循环累加的结果利用 main 函数的返回值带回操作系统，再使用特定命令显示该结果。乙程序经过编译后的 x86 可执行文件的反汇编代码如图 4.6 所示。

```
int main()
{
  int i;
  int sum = 0;
  for (i=1; i<=10; ++i)
  {
    sum = sum + i;
  }
  return sum;
}
```

图 4.5 乙程序源程序代码

```
080483db <main>:
 80483db: 55                    push   %ebp
 80483dc: 89 e5                 mov    %esp,%ebp
 80483de: 83 ec 10              sub    $0x10,%esp
 80483e1: c7 45 fc 00 00 00 00  movl   $0x0,-0x4(%ebp)
 80483e8: c7 45 f8 01 00 00 00  movl   $0x1,-0x8(%ebp)
 80483ef: eb 0a                 jmp    80483fb <main+0x20>
 80483f1: 8b 45 f8              mov    -0x8(%ebp),%eax
 80483f4: 01 45 fc              add    %eax,-0x4(%ebp)
 80483f7: 83 45 f8 01           addl   $0x1,-0x8(%ebp)
 80483fb: 83 7d f8 0a           cmpl   $0xa,-0x8(%ebp)
 80483ff: 7e f0                 jle    80483f1 <main+0x16>
 8048401: 8b 45 fc              mov    -0x4(%ebp),%eax
 8048404: c9                    leave
 8048405: c3                    ret
 8048406: 66 90                 xchg   %ax,%ax
 8048408: 66 90                 xchg   %ax,%ax
 804840a: 66 90                 xchg   %ax,%ax
 804840c: 66 90                 xchg   %ax,%ax
 804840e: 66 90                 xchg   %ax,%ax
```

图 4.6 乙程序 x86 反汇编代码

将该程序的 x86 可执行文件作为二进制翻译器的输入文件进行翻译，得到其 MIPS 下的汇编语言文件，其内容如图 4.7 所示。

```
  .text
  .globl main
  .ent main
  .type main, @function
main:
  addi  $sp,$sp,-4
  sw  $s5,0($sp)
  move  $s5,$sp
  li  $t8,-16
  add  $sp,$sp,$t8
  li  $t9,-0x4
  addu  $t9,$s5,$t9
  li  $t8,0x0
  sw  $t8,0($t9)
  li  $t9,-0x8
  addu  $t9,$s5,$t9
  li  $t8,0x1
```

```
  sw  $t8,0($t9)
  j  $L0
$L1:
  li  $t9,-0x8
  addu  $t9,$s5,$t9
  lw  $s0,0($t9)
  li  $t9,-0x4
  addu  $t9,$s5,$t9
  lw  $t8,0($t9)
  add  $t8,$s0,$t8
  sw  $t8,0($t9)
  li  $t8,0x1
  li  $t9,-0x8
  addu  $t9,$s5,$t9
  lw  $t4,0($t9)
  add  $t8,$t8,$t4
  sw  $t8,0($t9)
```

```
$L0:
  li  $t8,0xa
  li  $t9,-0x8
  addu  $t9,$s5,$t9
  lw  $t4,0($t9)
  subu  $t9,$t4,$t8
  sub  $t8,$t4,$t8
  blez  $t8,$L1
  li  $t9,-0x4
  addu  $t9,$s5,$t9
  lw  $s0,0($t9)
  move  $sp,$s5
  lw  $s5,0($sp)
  addi  $sp,$sp,4
  move  $v0,$s0
  jr  $ra
  .end main
```

图 4.7 翻译后的乙程序汇编代码

对上述汇编语言在 MIPS 仿真环境中使用图 4.8 中的命令进行汇编和链接，得到 MIPS 下的可执行文件，该可执行文件的执行情况如图 4.8 所示。

图 4.8 乙程序在仿真环境下的运行情况

如图 4.8 所示，程序正常运行至程序退出，未出现任何系统异常，且返回操作系统的值正确。乙程序翻译情况符合实验预期。

4.1.3 含过程调用的程序翻译

过程调用的程序翻译包括用户函数调用的翻译和库函数调用翻译两类，首先将测试如图 4.9 所示的程序丙，该程序使用定义的一个函数进行递归，通过递归方法求得一个等差数列的累加值。计算结果利用 main 函数的返回值带回操作系统，再使用特定命令显示该结果。丙程序经过编译后的 x86 可执行文件的反汇编代码如图 4.10 所示。

```
int get_sum(int n)
{
  if (n == 0)
  {
    return 0;
  }
  return n + get_sum(n-1);
}

int main()
{
  int sum = 0;
  sum = get_sum(10);
  return sum;
}
```

图 4.9 丙程序源程序代码

```
080483db <get_sum>:
 80483db: 55                    push   %ebp
 80483dc: 89 e5                 mov    %esp,%ebp
 80483de: 83 ec 08              sub    $0x8,%esp
 80483e1: 83 7d 08 00           cmpl   $0x0,0x8(%ebp)
 80483e5: 75 07                 jne    80483ee <get_sum+0x13>
 80483e7: b8 00 00 00 00        mov    $0x0,%eax
 80483ec: eb 19                 jmp    8048407 <get_sum+0x2c>
 80483ee: 8b 45 08              mov    0x8(%ebp),%eax
 80483f1: 83 e8 01              sub    $0x1,%eax
 80483f4: 83 ec 0c              sub    $0xc,%esp
 80483f7: 50                    push   %eax
 80483f8: e8 de ff ff ff        call   80483db <get_sum>
 80483fd: 83 c4 10              add    $0x10,%esp
 8048400: 89 c2                 mov    %eax,%edx
 8048402: 8b 45 08              mov    0x8(%ebp),%eax
 8048405: 01 d0                 add    %edx,%eax
 8048407: c9                    leave
 8048408: c3                    ret

08048409 <main>:
 8048409: 8d 4c 24 04           lea    0x4(%esp),%ecx
 804840d: 83 e4 f0              and    $0xfffffff0,%esp
 8048410: ff 71 fc              pushl  -0x4(%ecx)
 8048413: 55                    push   %ebp
 8048414: 89 e5                 mov    %esp,%ebp
 8048416: 51                    push   %ecx
 8048417: 83 ec 14              sub    $0x14,%esp
 804841a: c7 45 f4 00 00 00 00  movl   $0x0,-0xc(%ebp)
 8048421: 83 ec 0c              sub    $0xc,%esp
 8048424: 6a 0a                 push   $0xa
 8048426: e8 b0 ff ff ff        call   80483db <get_sum>
 804842b: 83 c4 10              add    $0x10,%esp
 804842e: 89 45 f4              mov    %eax,-0xc(%ebp)
 8048431: 8b 45 f4              mov    -0xc(%ebp),%eax
 8048434: 8b 4d fc              mov    -0x4(%ebp),%ecx
 8048437: c9                    leave
 8048438: 8d 61 fc              lea    -0x4(%ecx),%esp
 804843b: c3                    ret
 804843c: 66 90                 xchg   %ax,%ax
 804843e: 66 90                 xchg   %ax,%ax
```

图 4.10 丙程序 x86 反汇编代码

将该程序的 x86 可执行文件作为二进制翻译器的输入文件进行翻译，得到其 MIPS 下的汇编语言文件，其内容如图 4.11 所示。

```
  .text
  .globl main
  .ent main
  .type main, @function
main:
  li  $t9,0x4
  addu  $t9,$sp,$t9
  move  $s1,$t9
  li  $t8,0xfffffff0
  and  $sp,$sp,$t8
  addi  $sp,$sp,-4
  li  $t9,-0x4
  addu  $t9,$s1,$t9
  lw  $t8,0($t9)
  sw  $t8,0($sp)
  addi  $sp,$sp,-4
  sw  $s5,0($sp)
  move  $s5,$sp
  addi  $sp,$sp,-4
  sw  $s1,0($sp)
  li  $t8,-20
  add  $sp,$sp,$t8
  li  $t9,-0xc
  addu  $t9,$s5,$t9
  li  $t8,0x0
  sw  $t8,0($t9)
  li  $t8,-12
  add  $sp,$sp,$t8
  addi  $sp,$sp,-4
  li  $t8,0xa
  sw  $t8,0($sp)
  lw  $a0,0($sp)
  lw  $a1,4($sp)
  lw  $a2,8($sp)
  lw  $a3,12($sp)
  addi  $sp,$sp,-4
  sw  $ra,0($sp)
  .option pic0
  jal get_sum
  .option pic2
  lw  $ra,0($sp)
  addi  $sp,$sp,4
  li  $t8,0x10
  add  $sp,$sp,$t8
  li  $t9,-0xc
  addu  $t9,$s5,$t9
  sw  $s0,0($t9)
  li  $t9,-0xc
  addu  $t9,$s5,$t9
  lw  $s0,0($t9)
  li  $t9,-0x4
  addu  $t9,$s5,$t9
  lw  $s1,0($t9)
  move  $sp,$s5
  lw  $s5,0($sp)
  addi  $sp,$sp,4
  li  $t9,-0x4
  addu  $t9,$s1,$t9
  move  $sp,$t9
  move  $v0,$s0
  jr  $ra
  .end main
  .globl get_sum
  .ent get_sum
  .type get_sum, @function
get_sum:
  addi  $sp,$sp,-4
  sw  $s5,0($sp)
  move  $s5,$sp
  li  $t8,-8
  add  $sp,$sp,$t8
  li  $t8,0x0
  li  $t9,0x8
  addu  $t9,$s5,$t9
  lw  $t4,0($t9)
  subu  $t9,$t4,$t8
  sub  $t8,$t4,$t8
  bne  $t8,$zero,$L0
  li  $s0,0x0
  j  $L1
$L0:
  li  $t9,0x8
  addu  $t9,$s5,$t9
  lw  $s0,0($t9)
  li  $t8,-1
  add  $s0,$s0,$t8
  li  $t8,-12
  add  $sp,$sp,$t8
  addi  $sp,$sp,-4
  sw  $s0,0($sp)
  lw  $a0,0($sp)
  lw  $a1,4($sp)
  lw  $a2,8($sp)
  lw  $a3,12($sp)
  addi  $sp,$sp,-4
  sw  $ra,0($sp)
  .option pic0
  jal get_sum
  .option pic2
  lw  $ra,0($sp)
  addi  $sp,$sp,4
  li  $t8,0x10
  add  $sp,$sp,$t8
  move  $s2,$s0
  li  $t9,0x8
  addu  $t9,$s5,$t9
  lw  $s0,0($t9)
  add  $s0,$s2,$s0
$L1:
  move  $sp,$s5
  lw  $s5,0($sp)
  addi  $sp,$sp,4
  move  $v0,$s0
  jr  $ra
  .end get_sum
```

图 4.11 翻译后的丙程序汇编代码

对上述汇编语言在 MIPS 仿真环境中使用图 4.12 中的命令进行汇编和链接，得到 MIPS 下的可执行文件，该可执行文件的执行情况如图 4.12 所示。

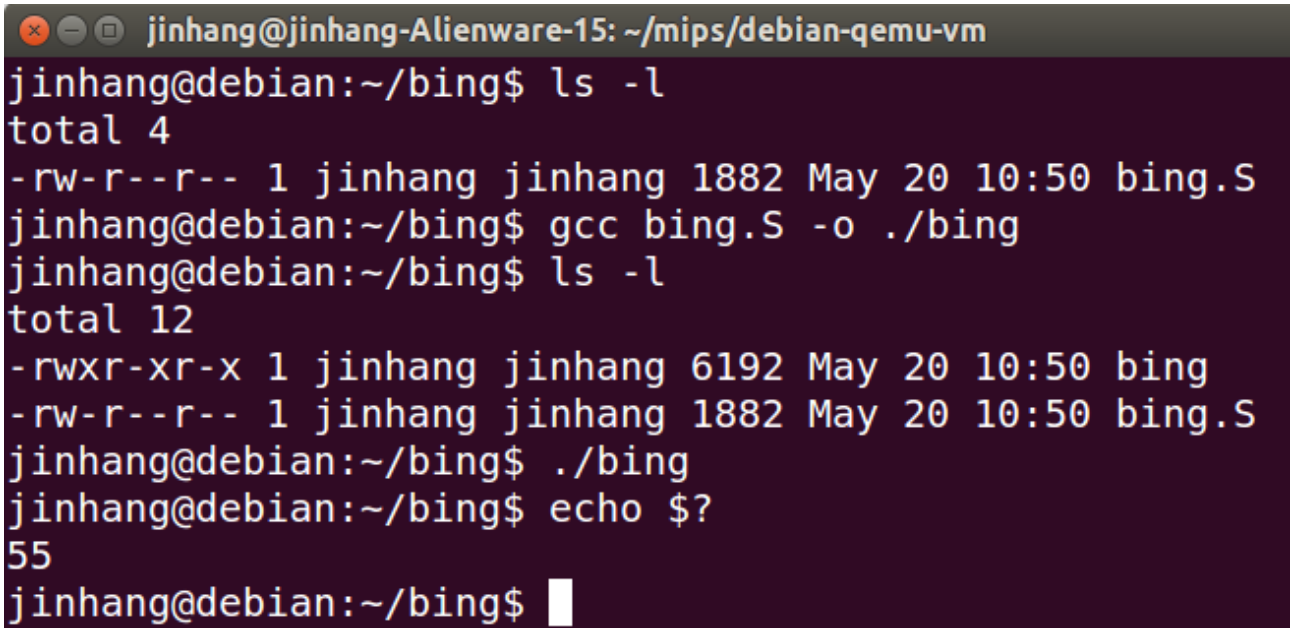

图 4.12 丙程序在仿真环境下的运行情况

如图 4.12 所示，程序正常运行至程序退出，未出现任何系统异常，且返回操作系统的值正确。丙程序翻译情况符合实验预期。

下面通过丁程序测试库函数调用。丁程序源程序代码如图 4.13 所示，使用简单的格式化打印字符串函数 printf 打印若干个加法算式的结果。戊程序的 x86 反汇编代码如图 4.14 所示。

```
#include <stdio.h>

int main(int argc, char const *argv[])
{
  char fmt[50] = "%d+%d=%d\n%d+%d=%d\n";
  int a = 1;
  int b = 2;
  int c = a + b;
  int d = 4;
  int e = 5;
  int f = d + e;
  printf(fmt, a, b, c, d, e, f);
  return 0;
}
```

图 4.13 丁程序源程序代码

```
0804846b <main>:
 804846b: 8d 4c 24 04           lea    0x4(%esp),%ecx
 804846f: 83 e4 f0              and    $0xfffffff0,%esp
 8048472: ff 71 fc              pushl  -0x4(%ecx)
 8048475: 55                    push   %ebp
 8048476: 89 e5                 mov    %esp,%ebp
 8048478: 53                    push   %ebx
 8048479: 51                    push   %ecx
 804847a: 83 ec 60              sub    $0x60,%esp
 804847d: 89 c8                 mov    %ecx,%eax
 804847f: 8b 40 04              mov    0x4(%eax),%eax
 8048482: 89 45 a4              mov    %eax,-0x5c(%ebp)
 8048485: 65 a1 14 00 00 00     mov    %gs:0x14,%eax
 804848b: 89 45 f4              mov    %eax,-0xc(%ebp)
 804848e: 31 c0                 xor    %eax,%eax
 8048490: c7 45 c2 25 64 2b 25  movl   $0x252b6425,-0x3e(%ebp)
 8048497: c7 45 c6 64 3d 25 64  movl   $0x64253d64,-0x3a(%ebp)
 804849e: c7 45 ca 0a 25 64 2b  movl   $0x2b64250a,-0x36(%ebp)
 80484a5: c7 45 ce 25 64 3d 25  movl   $0x253d6425,-0x32(%ebp)
 80484ac: c7 45 d2 64 0a 00 00  movl   $0xa64,-0x2e(%ebp)
 80484b3: 8d 45 d6              lea    -0x2a(%ebp),%eax
 80484b6: b9 1e 00 00 00        mov    $0x1e,%ecx
 80484bb: bb 00 00 00 00        mov    $0x0,%ebx
 80484c0: 89 18                 mov    %ebx,(%eax)
 80484c2: 89 5c 08 fc           mov    %ebx,-0x4(%eax,%ecx,1)
 80484c6: 8d 50 04              lea    0x4(%eax),%edx
```

```
80484c9: 83 e2 fc              and    $0xfffffffc,%edx
80484cc: 29 d0                 sub    %edx,%eax
80484ce: 01 c1                 add    %eax,%ecx
80484d0: 83 e1 fc              and    $0xfffffffc,%ecx
80484d3: 83 e1 fc              and    $0xfffffffc,%ecx
80484d6: b8 00 00 00 00        mov    $0x0,%eax
80484db: 89 1c 02              mov    %ebx,(%edx,%eax,1)
80484de: 83 c0 04              add    $0x4,%eax
80484e1: 39 c8                 cmp    %ecx,%eax
80484e3: 72 f6                 jb     80484db <main+0x70>
80484e5: 01 c2                 add    %eax,%edx
80484e7: c7 45 a8 01 00 00 00  movl   $0x1,-0x58(%ebp)
80484ee: c7 45 ac 02 00 00 00  movl   $0x2,-0x54(%ebp)
80484f5: 8b 55 a8              mov    -0x58(%ebp),%edx
80484f8: 8b 45 ac              mov    -0x54(%ebp),%eax
80484fb: 01 d0                 add    %edx,%eax
80484fd: 89 45 b0              mov    %eax,-0x50(%ebp)
8048500: c7 45 b4 04 00 00 00  movl   $0x4,-0x4c(%ebp)
8048507: c7 45 b8 05 00 00 00  movl   $0x5,-0x48(%ebp)
804850e: 8b 55 b4              mov    -0x4c(%ebp),%edx
8048511: 8b 45 b8              mov    -0x48(%ebp),%eax
8048514: 01 d0                 add    %edx,%eax
8048516: 89 45 bc              mov    %eax,-0x44(%ebp)
8048519: 83 ec 04              sub    $0x4,%esp
804851c: ff 75 bc              pushl  -0x44(%ebp)
804851f: ff 75 b8              pushl  -0x48(%ebp)
8048522: ff 75 b4              pushl  -0x4c(%ebp)
8048525: ff 75 b0              pushl  -0x50(%ebp)
8048528: ff 75 ac              pushl  -0x54(%ebp)
804852b: ff 75 a8              pushl  -0x58(%ebp)
804852e: 8d 45 c2              lea    -0x3e(%ebp),%eax
8048531: 50                    push   %eax
8048532: e8 f9 fd ff ff        call   8048330 <printf@plt>
8048537: 83 c4 20              add    $0x20,%esp
804853a: b8 00 00 00 00        mov    $0x0,%eax
804853f: 8b 5d f4              mov    -0xc(%ebp),%ebx
8048542: 65 33 1d 14 00 00 00  xor    %gs:0x14,%ebx
8048549: 74 05                 je     8048550 <main+0xe5>
804854b: e8 f0 fd ff ff        call   8048340 <__stack_chk_fail@plt>
8048550: 8d 65 f8              lea    -0x8(%ebp),%esp
8048553: 59                    pop    %ecx
8048554: 5b                    pop    %ebx
8048555: 5d                    pop    %ebp
```

```
8048556: 8d 61 fc               lea    -0x4(%ecx),%esp
8048559: c3                     ret
804855a: 66 90                  xchg   %ax,%ax
804855c: 66 90                  xchg   %ax,%ax
804855e: 66 90                  xchg   %ax,%ax
```

图 4.14 丁程序 x86 反汇编代码

将该程序的 x86 可执行文件作为二进制翻译器的输入文件进行翻译，得到其 MIPS 下的汇编语言文件，其内容如图 4.15 所示。

```
  .text
  .globl main
  .ent main
  .type main, @function
main:
  li  $t9,0x4
  addu  $t9,$sp,$t9
  move  $s1,$t9
  li  $t8,0xfffffff0
  and   $sp,$sp,$t8
  addi  $sp,$sp,-4
  li  $t9,-0x4
  addu  $t9,$s1,$t9
  lw  $t8,0($t9)
  sw  $t8,0($sp)
  addi  $sp,$sp,-4
  sw  $s5,0($sp)
  move  $s5,$sp
  addi  $sp,$sp,-4
  sw  $s3,0($sp)
  addi  $sp,$sp,-4
  sw  $s1,0($sp)
  li  $t8,-96
  add   $sp,$sp,$t8
  move  $s0,$s1
  li  $t9,0x4
  addu  $t9,$s0,$t9
  lw  $s0,0($t9)
  li  $t9,-0x5c
  addu  $t9,$s5,$t9
  sw  $s0,0($t9)
  li  $s0,0x14
  li  $t9,-0xc
  addu  $t9,$s5,$t9
  sw  $s0,0($t9)
  xor  $s0,$s0,$s0
  li  $t9,-0x3e
  addu  $t9,$s5,$t9
  li  $t8,0x252b6425
  sw  $t8,0($t9)
  li  $t9,-0x3a
  addu  $t9,$s5,$t9
  li  $t8,0x64253d64
  sw  $t8,0($t9)
  li  $t9,-0x36
  addu  $t9,$s5,$t9
  li  $t8,0x2b64250a
  sw  $t8,0($t9)
  li  $t9,-0x32
  addu  $t9,$s5,$t9
  li  $t8,0x253d6425
  sw  $t8,0($t9)
  li  $t9,-0x2e
  addu  $t9,$s5,$t9
  li  $t8,0xa64
  sw  $t8,0($t9)
  li  $t9,-0x2a
  addu  $t9,$s5,$t9
  move  $s0,$t9
  li  $s1,0x1e
  li  $s3,0x0
  sw  $s3,0($s0)
  li  $t9,0x4
  addu  $t9,$s0,$t9
  move  $s2,$t9
  li  $t8,0xffffffff
  and   $s2,$s2,$t8
  sub   $s0,$s0,$s2
  add  $s1,$s0,$s1
  li  $t8,0xffffffff
  and   $s1,$s1,$t8
  li  $t8,0xffffffff
  and   $s1,$s1,$t8
  li  $s0,0x0
$L0:
  li  $t8,0x4
  add  $s0,$s0,$t8
  sub   $t8,$s0,$s1
  subu  $t9,$s0,$s1
  bltz  $t9,$L0
  add  $s2,$s0,$s2
  li  $t9,-0x58
  addu  $t9,$s5,$t9
  li  $t8,0x1
  sw  $t8,0($t9)
  li  $t9,-0x54
  addu  $t9,$s5,$t9
  li  $t8,0x2
  sw  $t8,0($t9)
  li  $t9,-0x58
  addu  $t9,$s5,$t9
  lw  $s2,0($t9)
  li  $t9,-0x54
  addu  $t9,$s5,$t9
  lw  $s0,0($t9)
  add  $s0,$s2,$s0
  li  $t9,-0x50
  addu  $t9,$s5,$t9
  sw  $s0,0($t9)
```

```
  li  $t9,-0x4c
  addu  $t9,$s5,$t9
  li  $t8,0x4
  sw  $t8,0($t9)
  li  $t9,-0x48
  addu  $t9,$s5,$t9
  li  $t8,0x5
  sw  $t8,0($t9)
  li  $t9,-0x4c
  addu  $t9,$s5,$t9
  lw  $s2,0($t9)
  li  $t9,-0x48
  addu  $t9,$s5,$t9
  lw  $s0,0($t9)
  add  $s0,$s2,$s0
  li  $t9,-0x44
  addu  $t9,$s5,$t9
  sw  $s0,0($t9)
  li  $t8,-4
  add  $sp,$sp,$t8
  addi  $sp,$sp,-4
  li  $t9,-0x44
  addu  $t9,$s5,$t9
  lw  $t8,0($t9)
  sw  $t8,0($sp)
  addi  $sp,$sp,-4
  li  $t9,-0x48
  addu  $t9,$s5,$t9
  lw  $t8,0($t9)
  sw  $t8,0($sp)
  addi  $sp,$sp,-4
  li  $t9,-0x4c
  addu  $t9,$s5,$t9
  lw  $t8,0($t9)
  sw  $t8,0($sp)
  addi  $sp,$sp,-4
  li  $t9,-0x50
  addu  $t9,$s5,$t9
  lw  $t8,0($t9)
  sw  $t8,0($sp)
  addi  $sp,$sp,-4
  li  $t9,-0x54
  addu  $t9,$s5,$t9
  lw  $t8,0($t9)
  sw  $t8,0($sp)
  addi  $sp,$sp,-4
  li  $t9,-0x58
  addu  $t9,$s5,$t9
  lw  $t8,0($t9)
  sw  $t8,0($sp)
  li  $t9,-0x3e
  addu  $t9,$s5,$t9
  move  $s0,$t9
  addi  $sp,$sp,-4
  sw  $s0,0($sp)
  lw  $a0,0($sp)
  lw  $a1,4($sp)
  lw  $a2,8($sp)
  lw  $a3,12($sp)
  lui
$gp,%hi(__gnu_local_gp)
  addiu
$gp,$gp,%lo(__gnu_local_gp)
  move  $s4,$ra
  lw
$v0,%call16(printf)($gp)
  move  $t9,$v0
  .option pic0
  jal $t9
  .option pic2
  move  $ra,$s4
  move  $s0,$v0
  li  $t8,0x20
  add  $sp,$sp,$t8
  li  $s0,0x0
  li  $t9,-0xc
  addu  $t9,$s5,$t9
  lw  $s3,0($t9)
  li  $t8,0x14
  xor  $s3,$s3,$t8
  beq  $t8,$zero,$L1
  nop
$L1:
  li  $t9,-0x8
  addu  $t9,$s5,$t9
  move  $sp,$t9
  lw  $s1,0($sp)
  addi  $sp,$sp,4
  lw  $s3,0($sp)
  addi  $sp,$sp,4
  lw  $s5,0($sp)
  addi  $sp,$sp,4
  li  $t9,-0x4
  addu  $t9,$s1,$t9
  move  $sp,$t9
  move  $v0,$s0
  jr  $ra
  .end main
```

图 4.15 翻译后的丁程序汇编代码

对上述汇编语言在 MIPS 仿真环境中使用图 4.16 中的命令进行汇编和链接，得到 MIPS 下的可执行文件，该可执行文件的执行情况如图 4.16 所示。

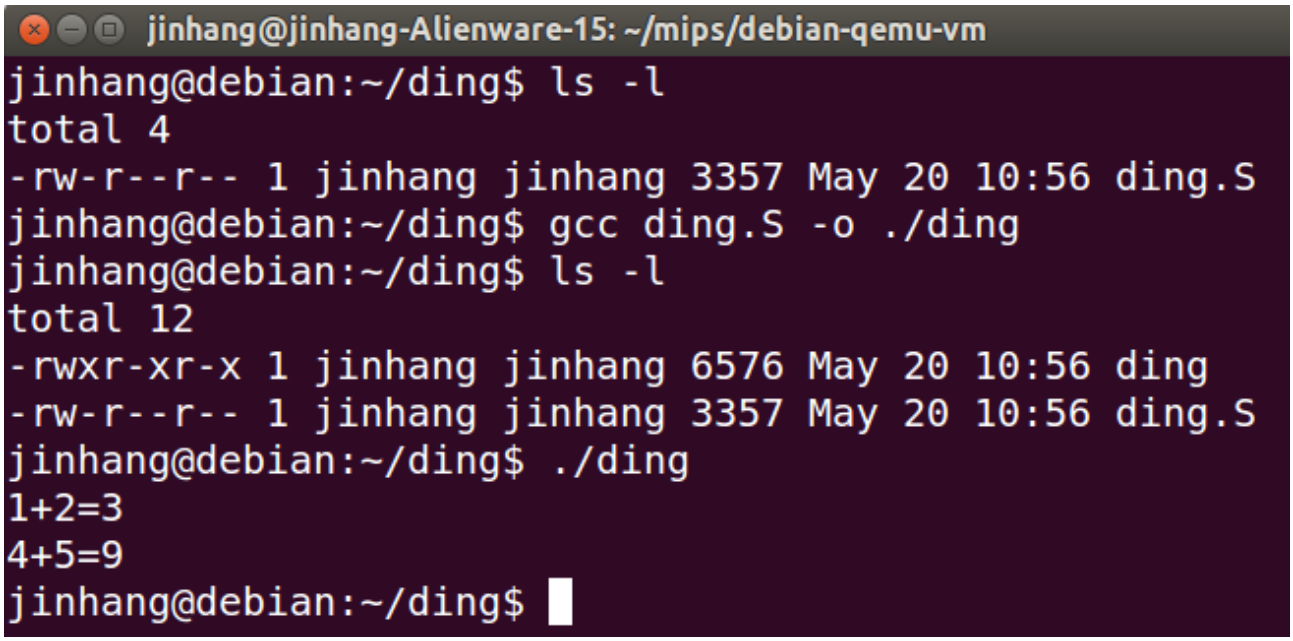

图 4.16 丁程序在仿真环境下的运行情况

如图 4.16 所示，程序正常运行至程序退出，未出现任何系统异常，且显示到屏幕上的运算情况和输出的字符串均符合预期。丁程序翻译结果正确。

4.1.4 引入 SDL2 库的程序翻译

SDL2 库是一个开源的、跨平台的图形库，已经在第二章中做了相应介绍，下面将使用第二章中所提及的简单 SDL2 程序显示一幅图片在屏幕上，其代码如图 2.13 所示，运行预期结果如图 2.14 所示。

将该程序的 x86 可执行文件作为二进制翻译器的输入文件进行翻译，得到其 MIPS 下的汇编语言文件。对该汇编语言代码在 MIPS 仿真环境中使用图 4.17 中的命令进行汇编和链接，得到 MIPS 下的可执行文件，该可执行文件的执行情况如图 4.17 所示。

图 4.17 简单 SDL2 程序在 MIPS 仿真环境下的运行情况

如图 4.17 所示，程序能够正常显示出如预期结果所示的图片，未出现任何系统异常，翻译结果正确。

4.2 基于龙芯架构的实验

龙芯中科公司于 2018 年初发布了新的嵌入式开发平台 2K 龙芯派，本实验将基于该 SoC 平台开展，试图利用该平台运行经由本翻译器翻译后的 MIPS 程序。运行程序以上述丁程序为例，将翻译后的丁程序在 MIPS 环境下汇编和链接，生成 MIPS 版本的丁程序可执行目标文件，将该目标文件下载到开发板上。

在 2K 龙芯派上执行该程序，运行情况如图 4.18 所示。可以看到，程序在屏幕上输出了与仿真环境下相同的结果，表明丁程序在龙芯真机的运行正常。

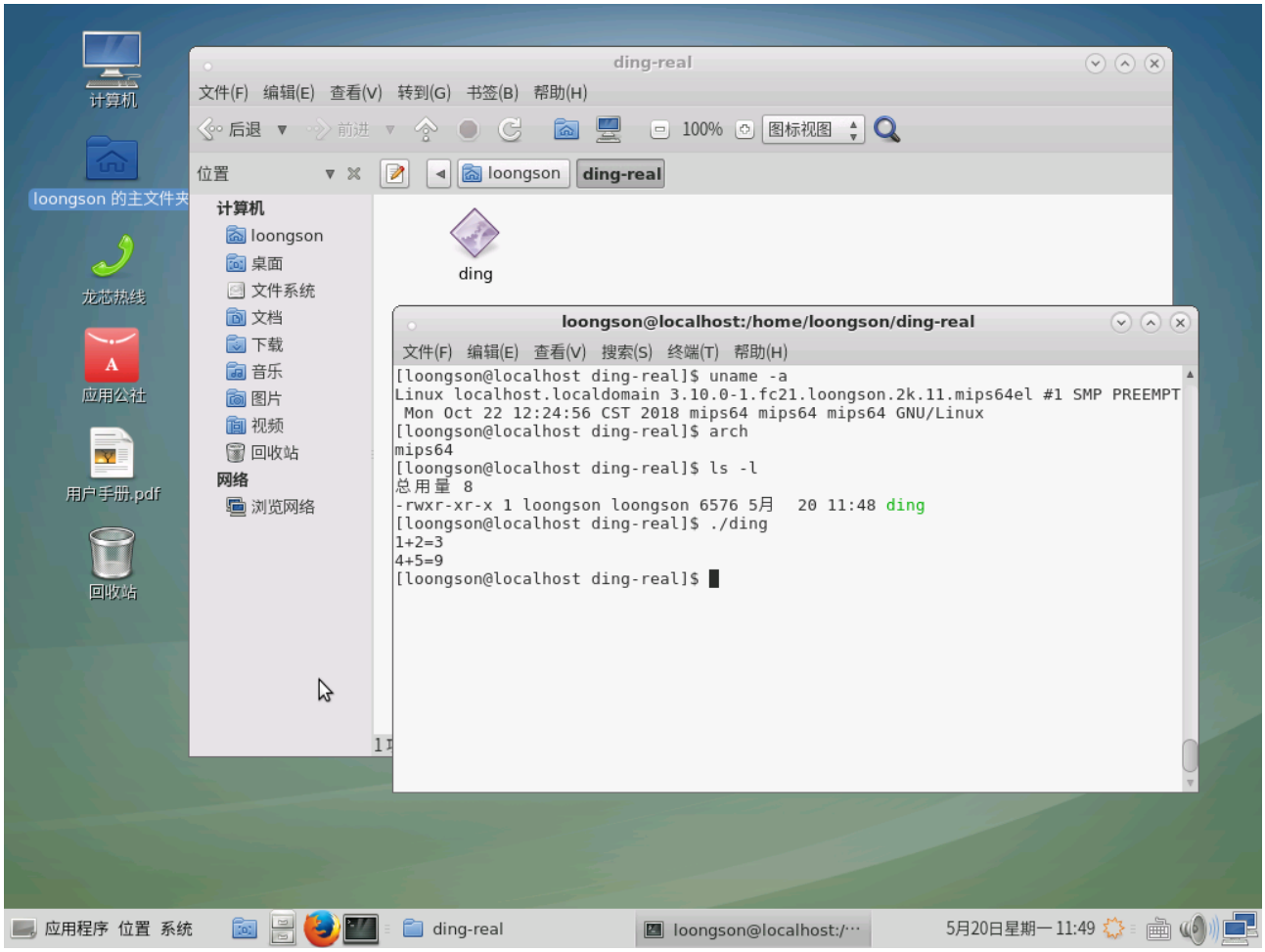

图 4.18 丁程序在龙芯开发板上的运行情况

4.3 实验分析

经过上述甲乙丙丁等四个测试用例的实验，表明本翻译器在翻译一般的由 C 语言编写的 x86 程序时表现基本正常，四个测试用例在种类上覆盖了常见的 C 语言程序结构，如顺序结构、分支结构、循环结构、过程调用和递归结构，其中过程调用又包括用户函数调用和库函数调用。测试在程序形式的覆盖上基本完全，在仿真环境下运行表现也符合实验预期。

除此以外，测试用例中还引入了比较常用的 SDL2 图形库进行实验，表明本翻译器在翻译含图形的程序以及引入了外部链接库的程序时也能得到正常的表现，符合实验预期。

另外，还选取了一个较为有代表性的程序在龙芯 2K 龙芯派开发板上进行了真机测试，测试结果表明经本翻译器所翻译的 MIPS 程序具有在真机上运行的能力，具有良好的应用前景。

4.4 本章小结

本章对本文提出的 x86-MIPS 二进制翻译器进行了仿真实验和真机测试，通过设定实验预

期和实验结果对比验证实验的正确性，并通过实验的正确性直接反映出本翻译器翻译功能的正确性。使用本翻译器翻译的程序不仅能够在仿真环境下顺利运行，也能在满足运行条件的真机上运行，说明本翻译器同时还具有一定的实用性。

第五章　工作总结与展望

5.1　全文总结

目前对于二进制程序跨平台通常采用虚拟机或模拟器作为解决方案。然而用户程序通常会依赖于操作系统提供的特定运行时环境，如系统文件和设备、目标指令集体系架构的静态链接库和动态链接库等。这就决定了模拟器不能直接将部分有所依赖的用户程序直接载入模拟器内存运行，目前比较成熟的方案是在模拟器内安装一个目标指令集体系架构的操作系统，将程序下载到模拟器中运行的操作系统中运行，然而这样就会造成较多问题。

首先，由于模拟器本身运行效率不高，在其中运行操作系统更会使运行效率大打折扣，因此才提出了翻译器的方案，将 x86 平台下的二进制程序在汇编代码层面上翻译到目标指令集体系架构下，可以减少中间的模拟层，更好地利用计算机的硬件资源运行二进制程序。

在翻译器的设计中，本文秉承“指令间无关”“进出一致”“地址统一”等基本思想，利用现代 x86 体系“扁平化”程序运行空间的思想屏蔽段寄存器，使得翻译后的程序在没有段机制的 MIPS 体系下运行的过程中也是“扁平化”的。经过本文的调研，只要保证程序在运行过程中不破坏系统区环境，平滑地在系统区和用户区之间过渡，程序就能在合理范围内使用用户地址空间而不必关心翻译后的程序和直接利用高级语言源代码在 MIPS 环境中编译得到的代码的差异，即寻求功能一致。在功能一致的前提下，本文又尽量考虑了程序效率，在翻译指令时尽可能采用更少的指令条数翻译一条复杂的 x86 指令，一定程度上减少了翻译造成的效率损失。

在测试用例的的选取上，本文本着尽可能覆盖更多程序样态的目的，从简到繁设计出若干测试用例在仿真环境下实验。在此基础上，又选取了最具代表性的测试用例下载到龙芯 2K 龙芯派平台上运行，尝试本翻译器所翻译程序在真机环境下运行的可能性。经过多次实验验证，以上选取的测试用例运行情况均和预期相符，本翻译器的设计和实现取得了具有建树性的成果。

5.2　研究展望

本文中所设计的二进制翻译器方案，适用于无论是否取得待翻译程序的高级语言源代码

的条件下，均能够将输入程序翻译到MIPS平台上，经过测试实验其运行结果符合预期。经过本文作者的考虑，结合当下计算机体系架构发展趋势，本文所提出二进制翻译器方案尚有以下几点继续研究的可能性：

1. 本文所设计的二进制翻译器方案在翻译后端采取“指令间无关”原则，使翻译过程中各个指令不具有很强的关联性，且每个 x86 指令对应一条“翻译工人函数”，若要将本翻译器用作 x86 指令集向其他指令集（如 RISC-V）的翻译，只需要根据其他指令集的指令行为对 x86 指令进行功能等价表出，仅修改“翻译工人函数”的代码就能够实现向其他指令集的翻译；
2. 由于本文在设计之初面向的是比较传统的 32 位版本的 x86 指令集和 MIPS 指令集，而现今的计算机系统通常采用 64 位（即 x86_64），龙芯公司的最新产品亦全面升级至 64 位版本 MIPS 指令集，这就给我们一个明确的方向：升级到 64 位。对本翻译器而言，需要修改翻译前端，令 ELF 二进制文件解析符合 64 位的解析规范；修改翻译后端，将目前生成的 32 位版本 MIPS 指令全面升级到 64 位版本，即可实现对 64 位程序翻译的支持；
3. 本翻译器目前的翻译方案中，仅仅翻译到用户函数内部，没有对库函数内部进行翻译，因此需要保证目标运行环境中安装有程序运行时需要的动态链接库；在一些嵌入式情形中，往往包括库函数在内均采用静态链接方案，若要使本翻译器能够在嵌入式领域亦占据一席之地，则还要考虑将库函数内部同样进行翻译，这就势必会涉及到操作系统接口层面的翻译，典型的如系统设备和系统调用的翻译；
4. 本翻译器采用的翻译方案为针对 Linux 下的 x86 可执行文件，然而 Linux 对于非计算机领域的用户而言应用不如普及度较高的 Windows 系统广泛，那么让本翻译器支持 Windows 下 PE 格式可执行文件，也就成为了一个有待继续研究的课题；其实本方面的实现并不较难，只需要修改翻译器前端，使翻译器能够解析 PE 格式可执行文件的各个段的信息，采用现有的翻译后端对指令集进行翻译即可。

从以上方向进行考虑，本文所设计翻译器仍有着较为可观的应用前景和发展前景。

参考文献

[1]蔡嵩松,刘奇,王剑,刘金刚.基于龙芯处理器的二进制翻译器优化[J].计算机工程,2009,35(07):280-282.
[2]李男,庞建民,单征.一种基于频度统计的动态二进制翻译优化方法[J].计算机工程与科学,2018,40(04):602-608.
[3]戴涛,单征,卢帅兵,石强,谭捷.基于优先级动态二进制翻译寄存器分配算法[J].浙江大学学报(工学版),2016,50(07):1338-1346.
[4](英)Dominic Sweetman 著,李鹏,等译.MIPS 体系结构透视[M].原书第 2 版,北京:机械工业出版社,2008.5.
[5]武成岗,马湘宁,崔慧敏.二进制翻译技术研究[J]. 信息技术快报, 2005.
[6]李晖,王振华,靳国杰.基于双 TLB 的二进制翻译访存性能优化[J].计算机工程,2015,41(12):75-81.
[7]远翔,武成岗,王振江.二进制翻译系统中信号处理机制的研究[J].高技术通讯,2015,25(06):543-551.
[8]秦焕青,刘敏,马刘杰.基于动态二进制翻译的关键内存防护[J].上海船舶运输科学研究所学报,2018,41(03):80-82+96.
[9]丁松阳,赵荣彩.静态二进制翻译中回调函数逆向恢复技术研究[J].计算机应用,2008(03):782-784.
[10]丁松阳,张墨华.二进制翻译形式化模型[J].电脑与电信,2007(09):9-10+28.
[11]魏振方.针对 x86_64 的二进制翻译若干关键技术研究[D].解放军信息工程大学,2011.
[12](美)Randal E. Bryant, David R. O'Hallron 著,龚奕利,等译.深入理解计算机系统[M].原书第 3 版,北京:机械工业出版社,2016.7.483,467,469,
[13]INTEL 80386 PROGRAMMER'S REFERENCE MANUAL[S].Intel,1986.
[14]SYSTEM V APPLICATION BINARY INTERFACE Intel386™.Architecture Processor Supplement[S]. Fourth Edition. The Santa Cruz Operation, 1997.
[15]MIPS-Market-leading RISC CPU IP processor solutions [EB/OL]. https://www.mips.com/.
[16]MD00086, MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual[S].MIPS, 2016.5.
[17]MD00741, MIPS® Architecture For Programmers Volume I-B: Introduction to the microMIPS32™ Architecture[S].MIPS, 2015.8.
[18]Simple DirectMedia Layer - Homepage [EB/OL]. http://www.libsdl.org/.
[19]ASM-01-DOC, MIPS Assembly Language Programmer's Guide[S]. Silicon Graphics, 2011.
[20]胡伟武,靳国杰,汪文祥,张晓春,王焕东.龙芯指令系统融合技术[J].中国科学:信息科学,2015,45(04):459-479.

致 谢

江宁之畔，金陵孟夏；芭蕉新绽，日渐炎热；受业四载，卧薪尝胆；回首相望，已为往事；新文初成，思虑不绝。

恩师冯爱民先生，实乃真学者也；授我以智慧，传我以身教；导我于狭路，示我以明途；感苍天有恩，予我与先生相会之契机；时值卒业，有感于先生之授教，铭而致谢。

工欲善其事，必先利其器；敝文得以成就，亦不乏龙芯石南先生之鼎力相助；研究条件，实为重要；协助之恩，定无相忘。

学士之位，博得不易；吾自入南航母校至今，遇良师无数；一日为师，终身为父；教育之恩，当以涌泉相报。群师之中，尤有良师李博涵先生，引我入兴趣之途；陈丹先生，予我以实践之机；高航先生，掘我于众人之中；刘绍翰先生，携我当启蒙之时；唐志文先生，则关怀入微，助我成长。

高山流水，知音难觅；良友难得，实为造化。后辈王山岳、秦瑞哲、蔡益武诸君尝于学途之路给予援手，感激涕零。团队之力，功不可没；事无巨细，当以感恩。

身体皮肤，受之父母；人之初成，不可忘恩。若比教师以学术之师，则比父母为人生之师。二师之一，尚不可缺；若无其一，不能为人。二十二载，光阴如梭；任劳任怨，良苦用心；万爱千恩百苦，疼我孰知父母。

大学四载，看似虽长；时至今日，实为短暂；忧从中来，不可断绝；惜今日余下之时光，期明日“研”途之风景；虽有不舍，羊城深造。

感南航予我学术之路，谢南航借我舒适之氛围。母校之情，铭记于心；诸先生之恩，定当终身难忘。

附 录

1．本文所设计翻译器方案，已经形成代码，并发布到 GitHub 之上，链接：https://github.com/jinhang1997/x86-mips-translator。