此教程以软件工程师的视角介绍 MIPS 指令集，且为入门级教程，所以抛弃了协处理器(cp0）、浮点寄存器、Cache、TLB 等相关知识,而添加了更多应用程序可能需要的示例讲解：函数调用、汇编文件.S、内嵌汇编、ELF 文件格式、系统调用等。更多关于协处理器、cache 相关内容还请参看第二版《See MIPS Run》

最后修改时间：（2020 年 5 月 14 日星期四）
作　　者：孙国云　sunguoyun@loongson.cn

第 1章 计算机语言

计算机语言就是人和计算机之间交流的语言。计算机就是一组电子器件，要让它为我们完成特定的工作，就需要向它输入一组它所能识别和执行的语言或者叫指令。和人类语言一样，计算机语言也有一套标准的语法规范。而且计算机语言的种类很多，从使用层次的角度从上到下常被分成高级语言、汇编语言和机器语言三大类。简单来说，离计算机处理器最远的是高级语言，中间是汇编语言，最远的但可以被机器直接执行的是机器语言。世界上绝大多数的计算机工作人员使用高级语言编写程序，然后利用编译器、汇编器帮助我们把高级语言一步步转换成处理器可以识别的机器语言，然后指导处理器运行。这个流程如图 1-1 所示：

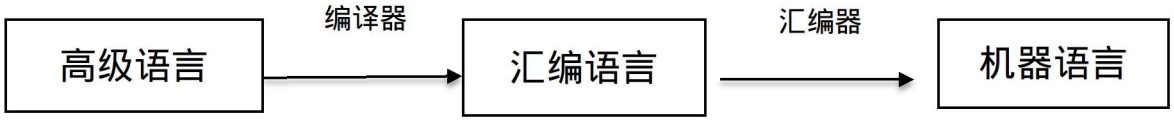


图 1-1 计算机语言转换过程

上图 1-1 描绘了高级语言到机器语言的转换过程，期间涉及到编译器和汇编器。简单来说，编译器负责帮助我们把高级语言（比如 c 语言）翻译成汇编语言，汇编器又帮助我们把汇编语言翻译成机器语言。机器语言就是最终可以被计算机识别和执行的语言。

下面分别介绍机器语言、汇编语言和高级语言。

1.1 机器语言

机器语言是计算机能直接识别和执行的程序语言，它的表示形式就是二进制。进制就是计数的一种方法，我们最为熟悉的十进制，就是用 0 到 9 共 10 个符号来表示，遵循“缝十进一”的进制规则。计算机的硬件作为一种电路元件，最容易用有电和没电状态来对外输入输出，也就是所说的高电平和低电平，分别对应到二进制数的“1”和“0”。每个数字被成为比特(bit)。

一条机器语言也叫一条指令，是计算机执行的基本单元。机器所具有的全部指令的集合就称为指令集或指令系统。指令系统是软件和硬件的接口层，反映了计算机所拥有的基本功能。指令系统有很多，x86、arm、mips、PowerPC 等等。中央处理器（CPU）是计算机中的核心配件。其功能主要是解释指令以及处理计算机软件中的数据。特定的 CPU 只能识别特定指令系统，比如 x86 指令集只能在采用 x86 指令集的处理器上，而不能运行在采用 ARM 或 MIPS 指令集的处理器上。一般而言指令集和体系结构是两个同义词，都包含了一组指令集和 一些寄存器的知识。

MIPS 指令系统中一条指令占用 32 个比特。比如我们要让 MIPS 处理器完成一个加法操作，它的指令如下面的样子：

0000 0000 0110 0010 0001 0000 0010 0001

这是让人很头疼的一串数字。因为我们很难直观读出这 32 个 0 和 1 组合代表什么意思。推想开来，我们要让计算机完成一个功能可能需要成千上万个长这样子的指令。编写难度可想而知。但是计算机发展的早期，工作人员就是这样干的。我们该向他们致敬。

1.2 汇编语言

汇编语言是机器语言的升级版。用一些容易理解和记忆的字母，单词来代替一个特定的机器指令。通过这种方法让我们更容易去阅读和理解程序正在执行的功能。比如还是 MIPS 架构下实现两个数的加法操作，其对应的机器指令和汇编指令如下：

机器指令： 0000 0000 0110 0010 0001 0000 0010 0001
汇编指令：add v0,v1,v0

一条汇编指令通常由助记符和操作数组成。助记符代表指令的功能，如上面的 add 就是助记符，功能是执行加法操作；操作数代表指令的计算对象，如上面的 v0、v1。v0，v1 是计算机中的两个寄存器（寄存器是计算机中临时存储数据的器件）。上面汇编指令“add v0,v1,v0”的意思是把寄存器 v0 和寄存器 v1 里存储的数做加法运算，结果存储到寄存器 v0 中。

通过这个例子可以看到，使用汇编语言可以使程序员不用关心这条指令对应的二进制数是多少，汇编器会帮助我们把它翻译成二进制的机器语言，编程效率得到很大提高。

汇编语言和机器语言一样都是和计算机体系架构强绑定的低级语言。也就是用 MIPS 指令集编写的汇编语言不可能运行到基于 X86 指令集或 ARM 指令集的 CPU 上。

同时为了便于人们的书写和计算，计算机领域也经常使用 16 进制的计数规则。16 进制就是用数字 0-9 和字母 A-F 共 16 个符号来表示，遵循“逢十六进一”的进制规则。比如用来做计算机数据存储的内存地址空间多用 16 进制表示，如下的一个地址：

```
0x120000af4
```

用“0x”标识这是一个 16 进制数。120000af4 值对应到 10 进制上就是 4831841012 字节。
在汇编指令对应的机器指令表示上也常用 16 进制。如上面的“add v0,v1,v0”对应的 16 进制就是
0x00621021

1.3 高级语言

高级语言是相对于低级语言而言，不再强依赖计算机处理器的硬件体系结构，表达方式更接近自然语言和数学公式的编程语言。高级语言并不特指某一个具体语言，而是很多语言的统称，比如 c 语言、c++、java 等等。这些语言都有自己的语法规则，但是都不能直接被计算机识别和执行，需要编译器和汇编器的翻译过程。这也是绝大多数编程人员所使用的语言。例如用 c 语言中实现两个数的加法运算基本格式如下：

```
int a=1,b=2;
a = a+b;
```

在上面的例子中，int 语句定义了两个整型变量 a、b，变量可以在保存寄存器中，也可以保存在内存中。a=a+b 一行以数学公式的表达方式对 a、b 执行求和计算，结果保存在 a 中。通过这个例子可以看到，C 语言比汇编语言更直观，更方便程序员编程。这两行 c 语言代码既可以在 MIPS 架构的计算机上编译运行，又可以在 x86 架构的计算机上编译运行，无需任何修改，这就是平台无关性。

高级语言设计思想发展历程经历了面向过程（c 语言为代表）、面向对象（java 语言为代表）、面向函数（Groovy、scala、Javascript 等）。未来的设计目标是面向应用，也就是说：只需要告诉程序你要干什么，程序就能自动生成算法，自动进行处理。高级语言设计思想的不断进化，让计算机语言越来越接近人类语言的思维方式，也更智能，编程效率也越来越高。

1.4 汇编语言的使用场景

很多使用高级语言编程的人员从业之初甚至从业多年都会有这样的疑问：现在大多数应用程序都可以使用高级语言进行编程学习汇编有什么用处？下面列举了几个汇编语言的使用场景：

场景 1：在应用程序出问题时，快速定位和分析

作为编程人员（特别是 c 语言程序员），平时应该都会遇到程序崩溃的问题，类似段错误、地址非法访问错误等。例如我们在写应用程序时都知道，做除法操作时是不允许被除数为 0 的，否则程序就会异常。我们可以在任意程序里，故意做一个除 0 的语句，类似于：

```
int a=3,b=0;
int c = a/b;
```

第 2 行语句在编译时是没有问题的，但是程序运行后就会异常。发生异常后，初级程序员可能通过在程序里加很多的 log 信息来定位问题。如果程序很大或者异常不是每次运行都会发生，那么定位这个问题将是一件很低效的过程。而具备汇编知识的高级程序员可能就会通过终端执行命令 dmesg 命令查看系统错误信息：

```
$ dmesg
CPU: 1 PID: 3394 Comm: hello Not tainted 3.10.84-22.fc21.loongson.10.mips64el #1
...
epc   : 0000000120000ab4 0x120000ab4
  Not tainted
ra    : 000000fff0dff1a4 0xfff0dff1a4
Status: e400ccf3    KX SX UX USER EXL IE
Cause : 10000008
BadVA : 0000000000000000
```

这段信息列出了出错应用程序的进程号、出错时刻应用程序用到的寄存器数值和 CPU 当前状态、错误原因、错误位置。我们通过寄存器 epc 给出值 0x120000ab4 来定位位置。首先用反汇编工具 objdump 帮助我们把可运行的应用程序翻译成可通过文本工具打开和查看的文件格式。比如应用程序是 a.out，那么 objdump 命令如下：

```
objdump -D a.out > a
```

这就实现了把 a.out 反汇编结果写入文件 a。然后像打开普通文本文件一样打开 a。从中查找 0x120000abe 所在位置和函数如下：

```
0000000120000a90 <main>:
 120000ab0:0062001a      div   zero,v1,v0
 120000ab4:004001f4      teq  v0,zero,0x7
 120000ab8:00001810      mfhi       v1
```

可以看出，出错函数是 main。错误位置是一个除法操作 div 。在第 3 章里我们还会更详细的介绍如何分析错误信息和定位问题。

场景 2：提升应用程序执行效率。

我们首先要明白，影响程序效率的因素有很多，合适的算法、编译器的优化、缓存、还有就是汇编优化。寄存器是离 CPU 最近的存储器，同时具有较高的读写速度，也是被 CPU 直接使用的存储器。 汇编语言直接操作的是寄存器，虽编写上会困难但是运行性能上要远优于高级语言。利用汇编提升性能的场景有很多，比如在游戏、音视频等领域，会经常遇到和图像处理相关的大数据量的数学运算。这里列举一个最简单的例子，用 c 语言实现一个 64M 的数据拷贝，基本语句如下：

```
#define SIZE 64*1024*1024
  for(int i=0;i<SIZE;i++){
    dest[i]=src[i];
  }
```

这段 c 语言代码使用 gcc 编译后对应的汇编指令如下：

```
loop:
beq t1,a2,exit      //判断 for 循环是否结束 。t1 代表 i ,a2 代表 SIZE
lb t2,0(a1)         //读 src[i]到寄存器 t2。a1 指向 src
sb t2,0(a0)         //写寄存器 t2 到 dest[i]。a0 指向 dest
addi t1,t1,1        //t1 自加 ++t1
addi a0,a0,1        //a0 自加 ++a0
addi a1,a1,1        //a1 自加 ++a1
j loop              //跳转到 loop:再次拷贝
```

上面这段指令对应了上面 c 语言里面的 for 循环语句。一次循环完成一个字节（Byte）的拷贝。在龙芯 3A3000 处理器上运行上述程序用时 210ms。下面我用汇编指令对上述功能做优化，核心过程如下：

```
loop:
beq t1,a2,exit      //判断 for 循环是否结束 。t1 代表 i ,a2 代表 SIZE
ld t2,0(a1)         //读 src[i]到寄存器 t2。a1 指向 src
sd t2,0(a0)         //写寄存器 t2 到 dest[i]。a0 指向 dest
addi t1,t1,1        //t1 自加 ++t1
addi a0,a0,8        //a0 自加 a0=a0+8
addi a1,a1,8        //a1 自加 a1=a1+8
j loop              //跳转到 loop:再次拷贝
```

这段汇编指令核心的改变是循环一次，拷贝 8 个字节。把编译器默认使用的 lb,sb 指令换成了 ld、sd。指令 lb 中的 b 代表 byte，意思是读一个 byte。而指令 ld/sd 中的 d 代表 double，意思是加载/存储 8 个字节。那么优化后的指令完成 64M 数据拷贝的时间是 59ms。

通过汇编指令的优化可以把性能提升近 4 倍。这对于那些运算量很大的程序来说确实是个很诱惑性的消息。

场景 3：完成高级语言无法实现的功能

比如获取当前进程运行在哪个处理器核上，可以使用如下汇编指令：

```
 int ret=0;
asm("rdhwr $2,$0       \n"
      "sw $2,%0          \n"
      :"+m"(ret)
 );
```

asm()是内嵌汇编，就是可以编写在 c 语言文件里的汇编程序，用来实现汇编语言和 c 语言的混合编程。MIPS 指令系统中，和处理器相关的控制寄存器是不能被用户程序直接读写。但是汇编指令 rdhwr 允许用户程序读取控制寄存器的一些信息到通用寄存器上。此处$0 是 MIPS 指令系统中专门存放处理器核信息的控制寄存器。$2 是编号为 2 的通用寄存器。通用寄存器可以被用户程序随便读写。指令”sw $2,%0”实现把处理器核信息写到 c 语言变量 ret。这里%0 代表了变量 ret。

汇编语言也是嵌入式设备上程序编写的理想工具。和一般的计算机处理器相比，嵌入式设备（比如电话、打印机、门禁设备等）的典型特征是没有大容量内存，这就要求其上的程序尽量短小。而汇编语言占用内存少，特别适合编写嵌入式程序。
在设备驱动程序（硬件的接口程序，用于控制硬件设备的工作）里也经常包含了大量的汇编语言代码。

场景 4：用于计算机体系结构原理教学

计算机体系结构常被定义为：程序员所看到的计算机的属性，即概念性结构与功能特性。它所研究的内容包括计算机基本工作原理、数据表示、寻址方式、寄存器定义、指令系统、中断机构、机器工作状态的定义和状态切换、机器级的输入、输出结构等涵盖计算机硬件、软件、算法和语言的综合概念和关系。计算机体系结构是计算机专业学生的必修课。在这门课程里需要使用汇编语言来讲解计算机的原理。

1.5 龙芯 CPU 介绍

龙芯 CPU 是中国科学院计算所自主研发的通用 CPU，采用自主 LoongISA 指令系统（兼容 MIPS 指令系统）。龙芯从 2001 年至今共开发了 1 号、2 号、3 号三个系列处理器和龙芯桥片系列，在政企、安全、金融、能源等应用场景得到了广泛的应用。 龙芯 1 号系列为 32 位低功耗、低成本处理器，主要面向低端嵌入式和专用应用领域；龙芯 2 号系列为 64 位低功耗单核或双核

系列处理器，主要面向工控和终端等领域；龙芯 3 号系列为 64 位多核系列处理器, 主要面向桌面和服务器等领域。

2019 年 12 月 24 日，龙芯 3A4000/3B4000 在北京发布，使用与上一代产品相同的 28nm 工艺，通过设计优化，实现了性能的成倍提升。使用龙芯公司最新研制的新一代处理器核 GS464V，主频 1.8GHz-2.0GHz，SPEC CPU2006 定点和浮点单核分值均超过 20 分，是上一代产品的两倍以上。具体性能参数如图 1-1 所示：

芯片	龙芯 3A4000
主频	1.8GHz–2.0GHz
峰值运算速度	128GFlops@2.0GHz
核心个数	4
处理器核	64 位超标量处理器核 GS464v； MIPS64 兼容； 支持 128/256 位向量指令； 四发射乱序执行； 2 个定点单元、2 个向量单元和 2 个访存单元
高速缓存	64KB 私有一级指令缓存、64KB 私有一级数据缓存； 256KB 私有二级缓存；共享 8MB 三级缓存
内存控制器	2 个 72 位 DDR4-2400 控制器，支持 ECC 校验
高速 I/O	2 个 16 位 HyperTransport 3.0 控制器； 支持多处理器数据一致性互连（CC-NUMA） 支持 2/4/8 路互连
其它 I/O	1 个 SPI、1 个 UART、2 个 I2C、16 个 GPIO 接口
制造工艺	28nm FDSOI 工艺
封装	37.5mm*37.5mm FC-BGA 封装
引脚数	1211
功耗管理	支持主要模块时钟动态关闭；支持主要时钟动态变频； 支持主电压域动态调压
典型功耗	<30W@1.5GHz　　<40W@1.8GHz　　<50W@2.0GHz

图 1-1 龙芯 3A4000 性能参数

龙芯 3 号 CPU 分为 3A 和 3B 系列。如 3A1000、3A2000、3A3000 和 3A4000。3A 开头系列的 CPU 主要用在桌面电脑。3B 系列如 3B1000、3B2000、3B3000 和 3B4000，主要用在多路服务器，多路意思是把多个 CPU 集成在一个主板上。

计算机性能评价的关键指标包含了主频、运算速度、CPU 核数、高速缓存。图 1-1 中对龙芯 3A4000CPU 关键性能指标都有列举。

主频就是 CPU 的时钟频率，即 CPU 每秒所产生的时钟脉冲，单位为赫兹（Hz）。主频越高，意味着 CPU 的工作节拍越快，运算速度也就越快。

运算速度衡量计算机工作能力的综合性指标，它取决于给定时间内 CPU 所能处理的数据量和 CPU 主频，单位用 MIPS（百万条指令/秒）和 MFLOPS（百万次浮点运算/秒）。MIPS 用于描述计算机定点运算能力而 MFLOPS 用于描述计算机的浮点运算能力。

CPU 核数即一个 CPU 由多少个核心组成。核心数越多，代表 CPU 并行处理事务的能力越强、性能越好。

高速缓存（Cache）属于高速存储器，是为了解决主内存存取速度一直比 CPU 操作速度慢得多，导致 CPU 的高速处理能力不能充分发挥而影响工作效率问题而出现的。高速缓存的容量一般只有主存储器的几百分之一，但它的存取速度能与中央处理器相匹配。原则上高速缓存容量越大越好，但是成本也会增加。

龙芯 CPU 使用的是兼容兼容 MIPS 指令系统的自主 LoongISA 指令系统，意味着只要是 MIPS 指令都可以在搭载龙芯 CPU 的计算机上运行。本书接下来章节介绍的都是 MIPS 指令集。

第 2 章　一窥 MIPS 指令风貌

MIPS 的意思是“无内部互锁流水级的微处理器”(Microprocessor without interlocked piped stages)，是一种采取精简指令集（RISC）的处理器架构。RISC 是相较于复杂指令集（CISC）而言，最大特点是寄存器更多，指令更少。目的是用更少的指令完成更多的任务，从而减少硬件设计的复杂程度，提高指令执行速度。 MIPS 最早是在 80 年代初期由斯坦福(Stanford)大学 Hennessy 教授领导的研究小组研制出来的。其指令系统经过通用处理器指令体系 MIPS I、MIPS II、MIPS III、MIPS IV 到 MIPS V，嵌入式指令体系 MIPS16、MIPS32 到 MIPS64 的发展已经十分成熟。以 MIPS 为代表的 RISC 处理器架构，相较与 x86 代表的 CISC 处理器架构，优点如下：

- 更多的通用寄存器

寄存器是 CPU 内暂时存放数据的单元，通用寄存器就是可供用户程序使用的寄存器。CPU 访问寄存器所用的时间要比访问内存的时间短。足量的寄存器使得程序运算的中间结果更多的暂存在 CPU 寄存器中，因而就减少了对内存的装入和存数，加快了运行速度。MIPS 采用 32 个通用 CPU 寄存器。

- 精简指令集

这里的精简包括更少的指令数、指令长度相等、更少的指令格式、和更少的寻址方式。大部分的指令都可以在一个 CPU 周期内完成。精简的指令集降低了硬件设计的复杂度、改善了处理器性能。MIPS 中的指令数不到 300 个？指令长度统一为 32 位、仅有 3 种指令格式和 1 种寻址方式。也正是源于这种精简的指令架构，MIPS 也更适合作为学习计算机体系结构的入门课程。

本章将通过一个简单的例子来对 mips 汇编有一个简单了解。同时作为基础知识，会介绍到 gcc 编译 c 语言的 4 个基本过程。

2.1 GCC 编译 C 语言的 4 个基本过程

GCC 原名为 GNU C 语言编译器（GNU C Compiler），就是负责把 c 语言的源程序转化成计算机可以执行的目标文件。之后 GCC 很快的扩展，可以支持 c 语言之外的 C++、Objective-C、Fortran、Java、Ada、go 等其他语言，称为 GNU 编译器组件（GNU Compiler Collection）。无论哪种语言，其编译原理和流程很类似，基本流程包括词法分析、语法分析、语义分析、中间代码生成、代码生成等阶段。下面介绍 GCC 编译 C 语言的 4 个基本过程。

首先我们编写一个简单的 c 程序 hello.c，内容如下：

```
#include <stdio.h>
#define STR “hello world!”

int main(){
        printf(“%s \n”，STR);
}
```

编写完成后，使用 gcc 命令编译上述代码:

```
$ gcc  -v --save-temps hello.c -o hello
  gcc 版本 4.9.4 20160726 (Red Hat 4.9.4-14) (GCC)
   /usr/libexec/gcc/mips64el-redhat-linux/4.9.4/cc1 -E -quiet -v hello.c -o hello.i
  /usr/libexec/gcc/mips64el-redhat-linux/4.9.4/cc1 -fpreprocessed hello.i  -o hello.s
   as -v -EL-o hello.o hello.s
 /usr/libexec/gcc/mips64el-redhat-linux/4.9.4/collect2  -o hello  crt1.o crti.o crtbegin.o hello.o
    crtend.o crtn.o
```

其中参数“-v”意为显示编译器的调用顺序、“--save-temps”意为不删除中间文件。为了便于接下来的分析和简洁，在这里对输出的信息做了一些删减和整理。同时你会发现在当前目录多了几个文件，具体如下：

```
$ ls
hello  hello.c hello.i  hello.o  hello.s
```

其中文件 hello 为最终可执行文件，名称通过参数“-o <file>”来指定 。hello.i、hello.s、hello.o 为 gcc 编译过程产生的中间文件。

从上面的输出信息可以清晰看出 gcc 编译过程中涉及到 3 个工具 cc1、as 和 collect2。cc1 是第一章提到的编译器，负责把高级语言（hello.c）进行预处理（hello.i）然后翻译成汇编语言（

hello.s）。as 是汇编器，负责把汇编语言翻译成机器语言（hello.o）。collect2 是链接器，负责将多个机器语言文件（*.o）组合成最终可在计算机上运行的可执行文件（hello）。我们可以通过下图直观显示这个过程：

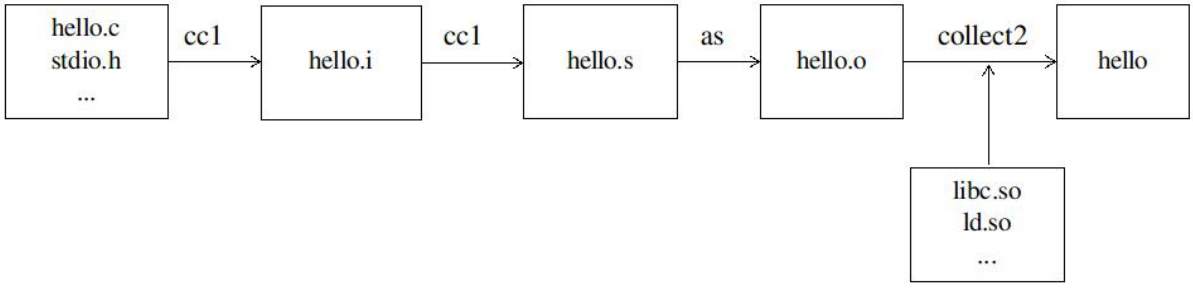


图 2-1 gcc 编译过程

图 2-1 中的 hello.o、libc.so、hello 文件都叫目标文件（Object File），这种文件采用 ELF（Executable Linkable Format）格式。ELF 格式的目标文件是非文本文件，一般里面包含两部分：指令代码和给链接器提供的编译信息。目标文件分为可重定向目标文件（Relocatable Object File）和可执行目标文件（Executable Object File）。hello.o、libc.so 就是可重定向目标文件，需要经过链接器重新组合、符号重定位后生成可执行目标文件 hello。

下面详细介绍 gcc 编译过程的 4 个阶段：

阶段 1：预处理

这是 gcc 编译的第一阶段，使用的工具是 cc1，产生的文件是 hello.i。 预处理后的 hello.i 文件内容大致如下：

```
# 1 "mips_book.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 1 "/usr/include/features.h" 1 3 4
# 365 "/usr/include/features.h" 3 4
# 1 "/usr/include/sys/cdefs.h" 1 3 4
# 402 "/usr/include/sys/cdefs.h" 3 4
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;
extern int fflush (FILE *__stream);

# 252 "/usr/include/stdio.h" 3 4
```

```
extern int fflush_unlocked (FILE *__stream);
# 266 "/usr/include/stdio.h" 3 4

extern FILE *fopen (const char *__restrict __filename,
     const char *__restrict __modes) ;

extern FILE *freopen (const char *__restrict __filename,
       const char *__restrict __modes,
       FILE *__restrict __stream) ;
...
int main(){
   printf("%s \n","hello world!");
}
```

预处理阶段主要是对各种预处理命令（以#开头的语句）进行处理，包括头文件包含、宏定义扩展、条件编译的选择等；主要处理规则如下：

1、将所有所有以“#”开头的语句，遇到#define 宏定义语句就进行扩展，之后删除#define;遇到#if 、#else、#endif 宏判断语句后就保存条件成立的部分，其他删除;

2、遇到#include 语句后，将包含的文件插入到该位置，然后删除此语句。比如上面 hello.i 里面已经看不到 STR 的定义，它已经被扩展成语句：

```
printf("%s\n","hello world");
```

3、删除所有的注释//和 /**/。比如上面 hell.i 里面已经看不到/*this is my test file*/

4、添加行号和文件名标识，比如“# 1 "/usr/include/stdio.h" 1 3 4”。这个用于编译阶段产生调试用的行号信息及产生错误或警告时的显示行号。

5、保留所有的#pragma 编译器指令给编译器使用。

阶段 2：编译

编译阶段使用的工具还是 cc1。这阶段主要是将预处理得到的源代码文件 hello.i 进行“翻译转换”，产生机器语言的汇编源文件（hello.s）。hello.s 文件是汇编源文件（assembler source），就是将要输入给汇编器使用的文本文件。打开 hello.s 文件里面内容如下：

```
.file 1 "hello.c"
.section .mdebug.abi64
.LC0:
```

```
.ascii	"hello world\000"
.globl	main
main:
.frame	$fp,32,$31		# vars= 0, regs= 3/0, args= 0, gp= 0
daddiu	$sp,$sp,-32
gssq $31,$fp,16($sp)
sd	$28,8($sp)
move	$fp,$sp
lui	$28,%hi(%neg(%gp_rel(main)))
daddu	$28,$28,$25
daddiu	$28,$28,%lo(%neg(%gp_rel(main)))
ld	$2,%got_page(.LC0)($28)
daddiu	$4,$2,%got_ofst(.LC0)
ld	$2,%call16(puts)($28)
move	$25,$2
.reloc	1f,R_MIPS_JALR,puts
...
```

可以看到，hello.s 里面已经不再有 c 语言的语句。实际上 hello.s 内容包含两部分：MIPS 汇编指令和伪指令。比如“daddiu$sp,$sp,-32”、“move　$fp,$sp”等都是 MIPS 汇编指令，“ .file 1 "hello.c"”、“ .section .mdebug.abi64”等都是伪指令。伪指令就是为了方便软件编程，由编译器定义的命令，在编译时转换为 CPU 实际执行的机器指令 ，用于指导汇编器如何翻译。

阶段 3：汇编

汇编阶段使用的工具是 as 。这阶段是将汇编指令翻译成了机器指令，生成可重定位的目标文件 hello.o。我们可以使用 file 查看一下 hello.o 文件类型。

```
$ file hello.o
hello.o: ELF 64-bit LSB relocatable, MIPS, MIPS64 rel2 version 1 (SYSV), not stripped
```

“relocatable”就表明了此文件是可重定位目标文件。重定位就是为了使二进制代码能够在不同存储器位置执行而执行进行的转换过程。可重定位文件就是文件里面已经包含了机器可执行的二进制指令和数据，但是里面所有段的起始地址都是 0x0，需要链接过程的地址重定义后形成最终可执行的文件。而且 ELF 信息说明了 hello.o 是标准的 ELF 文件格式，我们可以使用 readelf 命令查看 hello.o 文件的头信息，具体命令如下：

```
$ readelf -h hello.o
ELF 头：
```

Magic： 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Class: ELF64
Data: 2's complement, **little endian**
ABI Version: 0
Type: **REL (可重定位文件)**
Machine: MIPS R3000
Version: 0x1
入口点地址： 0x0
程序头起点： 0 (bytes into file)

“little endian”表示 hello.o 文件的字节序是小尾端。尾端是计算机中数据存储的一种模式，分为大尾端和小尾端两种。大尾端就是高位存放在低地址。小尾端就是地位存放在低地址。比如数据 0xeeffbbaa。它的存放方式如图 2-2 所示。

图 2-2 大尾端和小尾端的数据存放顺序

目标文件不是文本文件，不能直接打开。如果要查看里面具体的内容，可以使用反汇编工具 objdump。objdump 工具可以把目标文件转化成人类可读的 UTF-8 编码格式格式的文本文件，命令格式为“objdump -D 目标文件 > 文本文件”。参数“-D ”意思是显示所有的文件内容，“>”是重定向符号，意思是把 objdump 工具转化的结果输出到一个文件。例如我们反汇编文件 hello.o

```
$ objdump -D hello.o > a
```

打开文件 a 里面的内容如下：

```
0000000000000000 <main>:
  0: 67bdffe0      daddiu   sp,sp,-32
  4: ffbf0018 sd   ra,24(sp)
  8: ffbe0010      sd   s8,16(sp)
  c: ffbc0008      sd   gp,8(sp)
 10: 03a0f02d      move     s8,sp
 14: 3c1c0000      lui   gp,0x0
  ...
```

hello 文件中的内容实际上是一堆不可读的 0、1 二进制机器指令和 ELF 文件本身信息。为了便于阅读，反汇编工具 objdump 把这些指令按 3 列显示。第 1 列是每条机器指令将来运行时在内存的地址。第 2 列是真正的二进制机器指令的 16 进制表示形式。第 3 列是机器指令对应的汇编指令形式。这里我们发现 main 函数的入口地址是 0。实际上链接前的所有目标文件函数的入口地址都是 0。

阶段 4：链接

链接阶段使用的工具是 collect2（collect2 可以看作是 ld 链接器的封装，里面还是调用了 ld 来完成链接工作），生成最终的可执行目标文件 hello 。这个阶段主要内容就是将各个可重定向目标文件相互引用的部分正确的衔接起来形成一个文件。对于不能衔接进来的动态库（例如 libc.so），也要在 hello.o 中引用它的位置计算好地址，待程序运行时可以正确找到并动态加载它。衔接过程有 2 个核心工作就是符号解析和重定位。符号包括函数名和变量名，符号解析就是将每个符号定义和符号的引用确定关联，比如 main 函数中调用了 printf，printf 的定义并没有在 hello.c 里，那么符号解析过程就是帮助 main 函数在使用 printf 时可以正确找到它所在的真实位置。重定位就是负责把所有输入文件中的信息重新排列并重新计算里面符号的位置，比如把多个输入文件中的代码统一放在一个代码区、所有的静态变量、全局变量放在统一放在一个数据区并重新计算它们在程序运行时存放的地址，等等。

我们使用 objdump 反汇编 hello 文件，显示结果如下：

```
0000000120000ab0 <main>:
  120000ab0:67bdffe0       daddiu   sp,sp,-32
  120000ab4:ffbf0018 sd    ra,24(sp)
  120000ab8:ffbe0010       sd    s8,16(sp)
  120000abc: ffbc0008      sd    gp,8(sp)
  120000ac0: 03a0f02d      move     s8,sp
  120000ac4: 3c1c0002      lui   gp,0x2
   ...
```

链接后的 main 函数地址在 0x120000ab0。这个地址是 hello 程序运行过程中，main 函数在内存中的真实虚拟地址。

2.2 一窥 MIPS 指令风貌

通过上面的汇编源文件 hello.s 里面的内容，可以看到 MIPS 汇编语言的编写风格和其他架构大致相同，都是由汇编指令和伪指令组成。指令都是以行为单位，”#”之后的内容为注释内容，不参与编译和执行。单词后面跟一个“：”号代表一个标号，标号用来定义代码中的入口点或者数据的存储位置。例如 hello.s 里面的：

```
main:
    .frame    $fp,32,$31          # vars= 0, regs= 3/0, args= 0, gp= 0
```

```
daddiu   $sp,$sp,-32
```

“main:”代表了 main 函数的入口。.frame 为伪指令，用于汇编器调试使用，无实际作用。“daddiu　　$sp,$sp,-32”是 MIPS 汇编指令，实现寄存器 sp 和常数-23 的加法运算，结果存回 sp。寄存器 sp 是 MIPS 里比较特殊的一个寄存器，称做栈指针。它在程序运行之初指向了系统为本程序分配的一块固定空间地址，之后在程序里每个函数的入口和出口处，通过 sp 的移动来实现函数栈的空间申请。

2.2.1 流水线技术

流水线技术是一种将指令分解为多步，并让不同指令的各步操作重叠，从而实现几条指令并行处理，以加速程序运行过程的技术。一条指令分解成 3 步就称为 3 级流水线设计，分解成 5 条就称为 5 级流水线设计，甚至还有 12 级流水、18 级流水线设计。流水线技术不是 MIPS 处理器特有，现在几乎所有的处理器都采用流水线设计。MIPS 采用的是多级流水线技术。下面以典型的 5 级流水为例（IF 从 I-cache 取址、RD 读寄存器、ALU 算数/逻辑运算单元、MEM 读写 D-cache、WB 写寄存器），如图 2-3 所示。

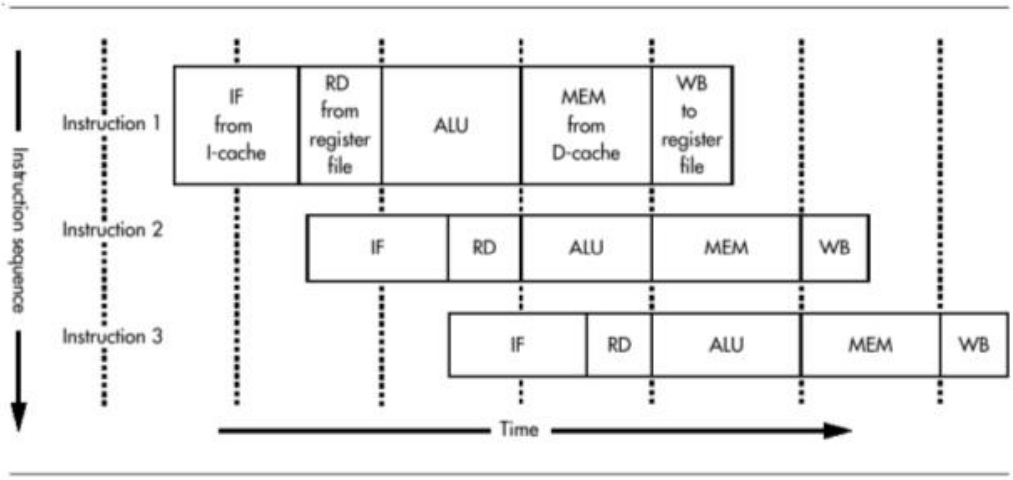


图 2-3 MIPS 体系架构的 5 级流水线

从图 2-3 可以看出，流水线中的每一级都占用固定的时间，所有指令也都经过严格定义来适应流水线的过程。流水线设计使得 CPU 在同一时刻同时执行多条指令，大大提升程序运行的性能。

2.2.2 MIPS 体系架构特点

MIPS 指令相较于其他架构的汇编指令，有如下典型特点：

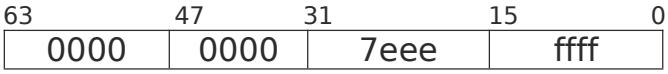
特点 1：MIPS 指令长度都是 32 位

CISC 里是变长指令，指令可以是 2 个字节、3 个字节或 4 个字节。MIPS 架构中寄存器有 32 位和 64 位之分，但是所有指令长度却都是 32 位（4 个字节）。固定的指令长度使得流水线设计过程中第一阶段 IF（取指令）时间对每条指令来说都是完全相同的。但是 MIPS 的二进制文件却比典型的 CISC 占用空间大出约 25%（CISC 中比如 x86 平均指令长度是 3 个字节）。

MIPS 指令长度为 32 位，指令的最高 6 位均为操作码，故不可能用一条指令完成 32 位或者 64 位的常数加载。比如我们要加载一个 32 位的常数 0x7eeeffff，需要两条指令才能完成：

```
lui  a1,0x7eee
ori  a1,a1,0xffff
```

先是通过指令 lui 把常数 0x7eee（0x7eeeffff 的高 16 位）存入寄存器 a1 的高 16 位，然后通过指令 ori 将寄存器 a1 和常数 0xffff 做与运算。在 MIPS 64 位处理器上最后 a1 的结果是：



特点 2：指令类型分 R、I 和 J 型

MIPS 指令长度是 32 位。指令的最高 6 位用来标识这个指令的功能，又称操作码（OPcode），剩下的 26 位可以将指令分为 3 种类型：寄存器指令 R-型、立即数指令 I-型和跳转指令 J-型。R-型指令特点是操作数和运行结果均保存在寄存器中。I-型指令特点是一个操作数为一个 16 位的立即数。J-型指令特点是用 26 位二进制码表示跳转目标的指令地址。这 3 种指令类型格式如图 2-4 所示。

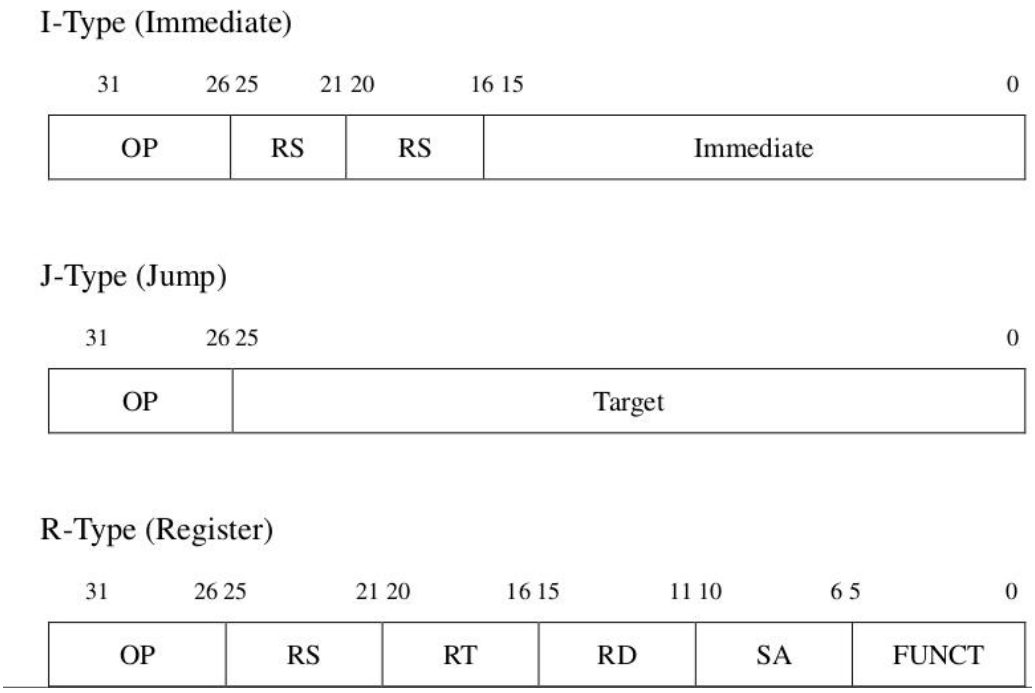


图 2-4 mips 指令格式

在图 2-4 中， OP（OPcode）代表操作码。RS（Regitser Source）代表第一个源操作数寄存器。RT（Register Target）即可以代表 第二个源操作数寄存器，又可以代表目标寄存器，要依具体指令而定。RD（Register Destination）代表 目标寄存器，用于存放操作结果。Immediate 代表 16 位立即数（用在指令里的常数）。Target 为 26 位 跳转目标地址。FUNCT 为 6 为功能域这个字段选择 Opcode 操作某个特定变体。

典型的 R-型指令包括一些运算指令、逻辑运算、跳转指令，如 ADD、SUB、MUL、AND、OR、JR、JALR 等。比如加法指令：

```
add       a1,a0,a2
```

上面指令 add 实现了寄存器 a0 和 a2 的加法运算，结果存入寄存器 a1。

典型的 I 型指令包括一些运算指令、加载指令和条件分支指令，如：ADDI、DADDIU、LD、LUI、B、BAL、BEQ。这些指令中的 I 代表 Immediate。比如带立即数的加法指令：

```
addi      a1,a0,0x2
```

上面指令 addi 实现了寄存器 a0 和立即数 0x2 的加法运算，结果存入寄存器 a1。

从 R-型和 I-型可以看出，MIPS 指令中算数/逻辑运算都是三操作数指令。

典型的 J 型指令包括 J、JAL。比如要实现函数调用，可以写成：

```
J test
```

表示跳转到符号 test 所在地址。这里指令 J 能跳转的范围是有限的，因为它只有 26 位用来存放地址。如果要想跳转到更远的地址可以使用 I-型的 JR 或 JALR 指令。

```
jr    t9
```

指令 jr 是跳转到寄存器。在 MIPS64 位处理器上，寄存器 t9 是 64 位，也就意味着 jr 可以跳转到任何有效的地址。

特点 3：没有条件码

条件码是 CPU 根据运算结果由硬件设置的位，体现当前指令执行结果的各种状态信息。例如算术运算产生的正、负、零或溢出等的结果。许多体系结构里面都会有相应的条件码寄存器，但 MIPS 体系架构中确实没有它， 为何设计成没有条件码？但 MIPS 对此有自己的处理方式。比如 MIPS 指令中算术运算分为“有溢出检测”和“无溢出检测”。比如指令：

```
add     a1,a0,a2
addu    a1,a0,a2
```

这两条指令都是 32 位加法运算，指令 add 执行过程会判断 a0+a2 结果的高 31 位和高 32 位是否相等，如果不相等说明数据有溢出，会报整型溢出异常（IntegerOverflow）。指令 addu 中额 u 代表“无溢出检测”，就是计算 a0+a2 的结果不做判断，直接赋值给 a1。

特点 4：流水线效应：分支延迟槽

如果程序所有的指令都是简单的数学运算、逻辑运算，那么流水线技术对用户程序来说是透明的，我们不用去关心它。但是程序中终会有条件分支、跳转等操作打断流水线的正常运作，这时就会有一些注意事项要我们遵守。

当前一条指令 Instruction1 是一条跳转指令，在 5 级流水的 ALU 阶段完成逻辑判断并修改计数器程序值准备跳转，指令 Instruction2 和 Instruction3 都已经开始执行但是都未完成。MIPS 对此情况的规定是：跳转指令后的第一条指令（Instruction2）总是在跳转指令之前完成。例如下面指令：

```
jal test
add a0,a0,a1   # a0=a0+a1;
nop
```

这里 jal test 表示跳转到 test 子程序。紧跟在它后面的指令”add a0,a0,a1”是实现 a0+a1 结果存入 a0。如果 jal 跳转成功，CPU 也是保证”add a0,a0,a1”指令执行完成之后，才会执行 test 子程序。test 执行返回后程序开始的位置就在第 3 条指令 nop 处。

这和我们眼睛看到的指令顺序和心里预期都不太一样。如果我们期望 test 中修改 a0 或者 a1 的值，然后再对它们进行加法操作，那么我们写指令时应该是这样子：

```
jal test
nop
add a0,a0,a1   # a0=a0+a1;
```

指令 nop 的意思是什么都不做。MIPS 汇编中经常使用这个指令在分支、跳转的语句后面充当分支延迟槽作用。分支延迟槽定义是紧跟在分支指令后面的那条指令。如果你能确保分支后面的指令不依赖分支跳转后的数据，那么任何指令都可以做延迟槽，否则就使用 nop。

特点 5：只有一种数据寻址方式

MIPS 中几乎所有的内存数据的加载/存储操作都是通过一个寄存器做基址加上一个 16 位的常数偏移量完成。例如：

```
ld   s8,24(sp)
```

这里指令 ld 是要完成数据从内存到寄存器的加载操作。数据所在内存的地址是通过寄存器 sp 和常数 24 确定，即 sp+24。地址确定后取值到寄存器 s8。

简单的数据寻址方式更有利于流水化的设计？

特点 6：没有硬件对堆栈的支持

栈（Stack）本身是一种先进后出的数据结构,在体系架构中就是内存中的一段空间，用于对子程序或函数的信息保存，使用上还是采用先进后出的原则。一般都是由 CPU 硬件支持，通常使用指令 PUSH（入栈）和 POP（出栈）操作。MIPS 体系架构中认为硬件支持的堆栈每次操作要 2 条指令完成（比如入栈需要一条指令将数据写入内存，另一条指令完成堆栈指针移动），不适和流水，所以 MIPS 中也有堆栈但不是硬件支持的。用户程序使用堆栈时可以直接通过通用寄存器 sp 的移动来完成栈管理。在子程序/函数开始都会通过 sp 加上一个固定值来申请栈空间（内存空间向下生长），使用时把 sp 做基址，通过偏移量完成对栈空间数据的读写，函数结束后通过 sp 减去这个固定值来释放栈空间。例如：

```
daddiu  $sp,$sp,-32      # sp = sp-32 申请 32 个字节的栈空间
sd      ra,24(sp)        # 寄存器 ra 存入栈 sp+24 的位置
sd      a0,24(sp)        # 取栈 sp+24 位置的数据到寄存器 a0
daddiu  $sp,$sp,+32      # sp = sp+32 释放 32 个字节的栈空间
```

第 3章　MIPS 寄存器

存储器（Memory）是计算机 3 个核心部件（控制器、运算器、存储器）之一，是用来存储程序和各种数据信息的记忆部件。当代计算机系统通常按照不同的存储容量、存储速度和价格把存储器分成如图 3-1 所示的层次结构：

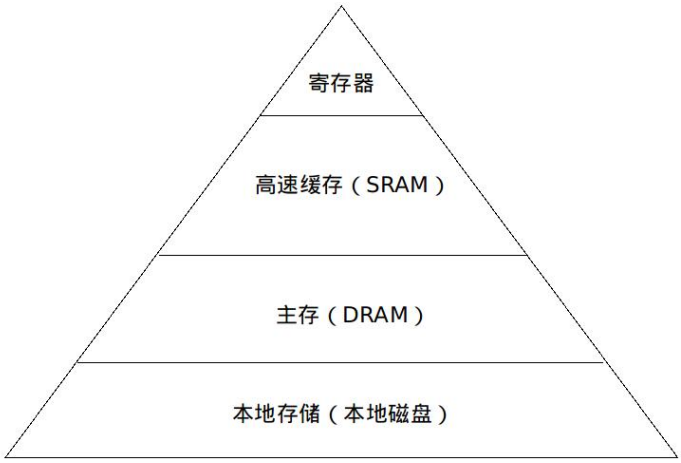


图 3-1 计算机存储器的典型层次结构

图 3-1 用金字塔形式表述了存储器的层次结构来表明越往上存储容量越小，但是存储速度也越快，离 CPU 越近，价格也越高。寄存器是 CPU 内部用来暂时存储数据的一些小型存储区域，用来暂时存放的数据包括参与运算的数据和运算结果、指令和地址。从图 3-1 可以看出寄存器是最靠近 CPU、读写速度最高的存储器。

不同的计算机架构中寄存器的种类和数量也不相同。MIPSmips 中用到的寄存器按照功能分为有通用寄存器、协处理器 0、浮点寄存器、乘法部件寄存器。通用寄存器共 32 个，是没有特殊限制，一般程序员可以使用的寄存器。协处理器 0 寄存器也叫控制寄存器，共 32 个，用来控制并管理 CPU。浮点寄存器和乘法部件寄存器都是专用寄存器。浮点寄存器也叫协处理器 1 寄存器，共 32 个，用来存储和浮点计算相关的数据。乘法部件寄存器共 2 个，用来存储和乘除法相关的数据。

寄存器的长度是由机器字长决定的，机器字长是指计算机进行一次整数运算所能处理的二进制数据的位数。龙芯 3A/3B 系列计算机的字长为 64 位，所以寄存器的长度也是 64 位。也叫 64 位寄存器。

3.1 32 个通用寄存器

MIPS 可供程序使用的通用寄存器（general-purpose register 简称 GPR）共 32 个，编号从 $0-$31。各个通用寄存器的别名和功能如表 3-1 所示。

表 3-1 n64 通用寄存器的别名和功能

寄存器编号	别名	功能
$0	zero	常量寄存器，其值永远为0
$1	at	汇编暂存（Assembly Temporary保留给汇编器使用）
$2-$3	v0,v1	用于存储子程序的返回值
$4-$11	a0-a7	子程序调用的前8个参数
$12-$15	t0-t3	临时变量（Temporaries），此程序使用时无需保存
$24-$25	t8,t9	临时变量（Temporaries），此程序使用时无需保存
$16-$23	s0-s7	子程序寄存器变量
$26,$27	k0,k1	保留给中断和自陷程序使用
$28	gp	全局指针（Global Pointer）
$29	sp	堆栈指针（Stack Pointer）
$30	s8/fp	帧指针（Frame Pointer）
$31	ra	函数返回地址（return address）

n64 代表的是用于 64 位处理器上的 MIPS ABI。在表 3-1 中列出了 n64 上的 32 个通用寄存器的别名和功能简介。

3.1.1 MIPS ABI

ABI（Application Binary Interface）：应用程序二进制接口。描述了应用程序与操作系统之间的底层接口，包括目标文件格式、数据类型、数据对齐方式、函数调用约定（应用程序如何使用内存地址和使用寄存器的约定）等。MIPS 历史上重要的 3 个 ABI 如下：

- o32 “o”代表 old。传统的 32 位处理器的 MIPS ABI 约定。指针和 long 类型为 32 位\
- n64 “n”代表 new。新的用于 64 位处理器的 MIPS ABI 约定。新的 ABI 约定了指针和 long 类型都为 64 位。同时改变了寄存器和参数传递的规则。将更多的参数用寄存器传递（a0-a8）。

- n32 “n”代表 new。用于 64 位处理器上兼容 32 位程序。指针和 long 类型还是使用 32 位。

3.1.2 寄存器别名

为了便于记忆和阅读，每个寄存器都有一个别名。别名多以寄存器功能的英文首字母+数字或字母缩写表示。例如寄存器 2 的别名 a0 开头的 a 代表 arguments。寄存器 29 的别名 sp 代表 stack pointer 。在实际汇编语言编写过程中，我们也更推荐使用寄存器的别名方式，使用 MIPS 别名时需要包含头文件 regdef.h（例如 #include <sys/regdef.h>）,里面已经定义好了寄存器编号和别名的对应。下面这两句指令是完全相等的。

```
daddiu  $29,$29,32
daddiu sp,sp,32
```

在 MIPS 指令中使用任何寄存器编号时都要加上符号”$”。上述两条指令是完全相同的，都是实现$29+32，结果存入$29 的功能。使用$29 和 sp 是完全相同的。不过使用 sp 更让我们是在对堆栈指针的操作。

3.1.3 通用寄存器功能介绍

在表 3-1 中简单介绍了各通用寄存器的功能，接下来对其进行详细介绍。

- **伪指令与寄存器 zero、at**

寄存器 zero($0)是常量寄存器，即不管存入什么值，永远返回 0。其对我们平时的汇编语言编写用处不大，但是对一些合成指令的作用还是很大，它为我们提供了一种更简洁的编码方式。比如 MIPS 中常常用到的一些伪指令：

```
//伪指令        汇编器        //汇编指令
move t0,t1                  or t0,t1,zero
```

简单解释一下伪指令，伪指令就是为了方便软件编程，由编译器定义的命令，在编译时转换为 CPU 实际执行的机器指令 。也就是说伪指令并不是 CPU 最终可执行的机器指令，而是给汇编器看的指令。汇编器负责将伪指令翻译成正式的机器指令。有的地方称伪指令为合成指或宏指令。
MIPS 汇编中没有两个寄存器之间数据拷贝的指令。要实现这个功能可以通过 or 指令实现。上面的汇编指令 or t0,t1,zero 意思是寄存器 t1 和寄存器 zero 进行或运算 t1 | zero，结果存入 t0。寄存器 zero 里面值都为 0。所以 t1|zero 的结果还是 t1，然后存入 t0。这就实现了一个寄存器 t1 到另一个寄存器 t0 的数据拷贝。但是这个写法还不是很直观，从“or”名字上很难看出这是个拷贝功能。所以汇编器对此实现了与此功能相同的伪指令 move，要实现寄存器 t1 到 t0 的拷贝，只

要写成 move t0,t1 即可。伪指令 move t0,t1 就是告诉汇编器拷贝寄存器 t1 的值到寄存器 t0。汇编器帮助我们把这条指令翻译成 CPU 可识别和执行的机器指令 or t0,t1,zero。

通用寄存器 at（assembly temporary），为汇编器所保留。基本可以认为通用寄存器 at 是伪指令的中间变量。之前提到过 I 型指令的立即数字段只有 16 位，所以在加载大常数（大于 16 位的数）时，编译器或汇编程序需要把大常数拆开，然后重新组合到寄存器里。比如加载一个 32 位立即数需要 lui（装入高位立即数）和 addi 两条指令。这个过程由汇编程序来完成。这时汇编器就需要一个临时寄存器 at 来重新组合大常数。

既然寄存器 at 是为汇编器所保留的，那么我们在编写汇编程序时就要注意避免使用此寄存器。当然如果你确实想在自己的汇编程序中使用这个寄存器，可以通过伪指令.set noat 来通知汇编器。但是这样一来汇编器中用 at 做中间变量的宏指令就不能再使用了，比如指令 rol（循环左移）、指令 rem（有符号整数除余）、指令 jal（跳转）等。

- **函数调用与寄存器 v0、v1、a0-a7、ra**

所有的高级语言中都有子程序或者函数这个概念。子程序或者函数就是可以独立实现一个特定功能的程序块。在这里我更愿意使用函数。函数基本由返回值、函数名、参数、函数体组成。格式如下：

返回值类型 函数名称（参数列表）{函数体};

在使用一个函数时需要关心的是参数、返回值和返回地址。MIPS n64 约定函数调用时使用寄存器 a0 至寄存器 a7 来传递前 8 个参数，用寄存器 v0 存放子程序的返回值（整数或者指针），寄存器 v1 保留，用寄存器 ra 保存返回地址。

比如我们编写一个有 2 个整型参数，一个整型返回值子程序的 c 语言代码如下：

```
int ret = add(2,3);  //c 语言代码
```

经过编译后对应的汇编指令如下：

```
//汇编指令
li a0,0x2
li a1,0x3
bal  add
nop
sw v0,32(gp)
```

第 1 行和第 2 行分别将两个整型参数 2、3 存入寄存器 a0、a1。第 3 行的 bal 指令完成跳转到函数 add。第 4 行的 nop 是空指令，意思是什么都不做，充当延迟槽作用。指令 bal 实现到函数 add 的跳转，在跳转到 add 之前，指令 bal 会负责保存返回地址（指令 sw 所在位置）到寄存器 ra（$31）。这样在函数 add 返回可以通过 jr ra 完成。汇编器把变量 ret 的地址保存在 gp+32 的位置。那么就可以通过最后的 sw 指令把返回值 v0 写到 ret。

- **临时寄存器 t0-t3、t8、t9**

通用寄存器 t0 至 t3 和 t8、t9 共 6 个寄存器在子程序中充当临时变量的作用，这里”t”表示 temporaries。临时变量就是在子程序中使用的变量。子程序结束后，这几个寄存器的值就无效。所以一个程序中不必保存而自由使用这 6 个寄存器。其中，t9（$25）经常被汇编器用来保存子程序的地址，然后执行子程序跳转功能，类似于如下写法：

```
jalr    t9
```

指令 jalr t9 意思是跳转到寄存器 t9 所存储的地址。

- **寄存器变量 s0-s7**

s0-s7 中的”s”代表 saved。这 8 个寄存器是保存寄存器，如果程序中使用了这组里面的寄存器，在发生函数调用之前需要对该寄存器做保存。也称子程序寄存器变量。这里解释一下什么是寄存器变量。通常我们的程序是保存在内存或者外存上的，需要时才会加载到寄存器。如果一个变量在程序中频繁使用，例如循环变量，那么，系统就必须多次访问内存中的该单元，影响程序的执行效率。因此，C 语言\C++语言还定义了一种变量，不是保存在内存上，而是直接存储在 CPU 中的寄存器中，这种变量称为寄存器变量。寄存器变量的定义格式为 “register 类型标识符 变量名”。例如我们使用 c 语言定义如下两个寄存器变量：

```
register int a=2;
register int b=3;
```

上面两行语句经过 gcc 编译后，对应的汇编指令如下：

```
li    s1,2
li    s0,3
```

上面的 2 条汇编指令 li 是加载立即数（常数）到寄存器 s1 和 s0。使用时就无需去内存加载。s0-s7 在使用上和 t0-t9 恰恰相反，s0-s7 在子程序的执行过程中，需要将它们存储在堆栈里，并在子程序结束前恢复。从而在调用函数看来这些寄存器的值没有变化。

- **系统保留寄存器 k0、k1**

k0 和 k1 是为系统所保留，专门保留给系统发生中断时程序使用的寄存器。这里的”k”代表 keep。中断是指计算机运行过程中，出现某些意外情况需主机干预时，机器能自动停止正在运行的程序并转入处理新情况的程序，处理完毕后又返回原被暂停的程序继续运行。
MIPS 架构的系统在这个过程中就是通过 k0,k1 引用一段可以保存其他寄存器的内存空间来实现正在运行程序的环境保存。关键汇编程序如下：

```
mfc0    k1,CP0_CAUSE
andi    k1,k1,0x7c
ld      k0,exception_handlers(k1)
```

```
    jr      k0
```
上述指令中，首先通过指令 mfc0 把控制寄存器 CP0_CAUSE 值拷贝到通用寄存器 k1。CP0_CAUSE 保存了这次中断发生的原因。用 andi 指令把 k1 的值加上 0x7c，然后用指令 ld 加载具体一个中断处理程序的入口地址 exception_handlers(k1)。exception_handlers（）记录了很多中断处理程序，根据 k1 的值找到当前系统需要执行哪个中断处理程序。然后使用指令 jr 跳转到这个中断处理程序。exception_handlers(k1)里将完成被中断的程序环境保存。
备注：这段指令引自内核代码 genex.S

- **相对于 gp 的寻址**

通用寄存器 gp 可以看做是为汇编器保留。gp 代表 global pointer，用于快速的存取 static 和 extern 类型的变量。

MIPS 指令集中，加载指令都是 I-型指令，也就是用 16 位来存储地址偏移量。所以要对一个常数或者变量地址的加载最少需要 2 条指令才能完成。比如要加载 32 位的变量地址就先需要一条指令加载变量地址的高 16 位到临时寄存器（通常是通用寄存器 at），然后是一条以该变量地址的低 16 位为偏移量的加载指令。例如：

```
    lui     at,%hi(addr)
    lw      v0,%lo(addr)(at)
```
上面两条指令中 lui at,%hi(addr)就代表了取变量 addr 的高 16 位，赋值给 at。%lo(addr)就是取变量 addr 的低 16 位做偏移量，at 为基址。%lo(addr)(at)相当于 at+%lo(addr)。指令 lw 就完成了 addr 地址加载到寄存器 v0。

c 语言程序中包含了很多 static 和 extern 类型的变量。对这些类型变量的加载和存储需要至少 2 条指令来完成确实开销很大。对此编译器利用寄存器 gp 来做了优化，可以使用 1 条指令完成对这类变量的加载和存储。做法编译器会统一把 static 和 extern 类型变量放在一个 64KB 大小的内存区域。然后让寄存器 gp 指向这块内存区域的中间位置，接下来的变量寻址都以 gp 为基址，加上特定偏移量就可定位到某个变量所在地址并完成加载或者存储。例如：

```
    lw      v0,offset(gp)
```
gp+offset 就可以定位到一个 static 或 extern 类型变量的地址，然后通过指令 lw 加载到寄存器 v0。

注意：这里的变量不包括函数内部定义的局部变量。函数内部的局部变量是通过寄存器或堆栈实现，不需要 gp。

- **函数栈和寄存器 sp、fp**

在计算机领域，栈（stack）是允许在同一端进行插入和删除操作的动态存储空间。它按照先进后出的原则存储数据，先进入的数据被压在栈底，最后的数据在栈顶。函数栈就是在程序运行时动态分配，用来保存一个函数调用时需要维护的信息。这些信息包括函数的返回地址和参数，临时变量、栈位置。一个典型的函数栈如图 3-2 所示：

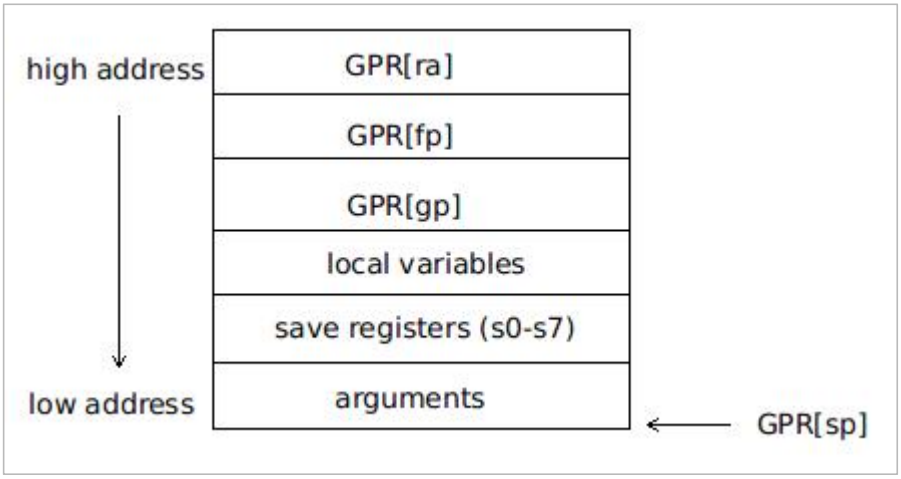


图 3-2 典型的函数栈

MIPS 架构中，栈是向下增长的，也就是栈底在高地址，栈顶在低地址。在图 3-2 中，GPR[sp]代表通用寄存器 sp（$29）指向栈顶，又称 sp 为栈指针（stack pointer）。每次函数开始的时候 sp 都会向下移动 n 字节，sp 预先指定了一块存储区域。这 n 个字节就是此函数的栈空间。这里要求 n 要求必须是 16 的倍数。典型的分配函数栈指令：

```
daddiu  sp,sp,-48
```

指令 daddiu 是带常数的加法指令，上面指令相当于 c 语言表达式 sp = sp-48。相当于把 sp 指针向下移动 48 字节，也就是申请一个 48 字节的栈。当函数返回时就通过 sp 指针向上移动 48 字节恢复栈，指令如下：

```
daddiu  sp,sp,48
```

通常通过 sp 栈指针分配空间后，我们就可以根据需要操作函数信息的保存和恢复。一般都会先把通用寄存器 ra 和 fp 入栈，如果有 gp 的操作，那么也会把 gp 入栈。当此函数要调用其他函数时，那么此函数内部的局部变量、寄存器变量、参数等也要做入栈保存。

栈空间的分配是以进程为单位的。进程是系统进行资源分配和调度的基本单位。系统会在进程启动时指定一个固定大小的栈空间，用于该进程的函数参数和局部变量的存储。sp 的初始值就指向了这个固定大小栈的栈底。该进程中的每一次函数调用，都会通过 sp 指针的移动来为函数在此空间划分出一块空间用作函数栈，sp 指向栈顶。函数栈又被称作栈帧（stack frame）,用通用寄存器 fp（$30）指向当前函数栈的栈底。fp 又被称作帧指针（frame pointer）。一个进程典型的栈空间如图 3-3 所示。

每次调用一个函数，都要为该次调用的函数实例分配栈空间，用来局部变量的分配和释放传递函数参数，存储返回值信息，保存寄存器、以供恢复调用前处理机状态。mips 下栈空间采用的是向下增长的方式（上面为高地址，下面为低地址）。SP(stack pointer) 就是当前函数的栈指针，它指向的是栈底的位置。

进入一个函数时需要将当前栈指针向下移动 n 字节，这个大小为 n 字节的存储空间就是此函数的栈的存储区域，然后在函数返回时再将栈指针加上这个偏移量恢复栈现场。例如下面函数部分指令：

```
//通过 objdump 工具 获取到的 main 函数

0000000120000b38 <main>:
  120000b38:   67bdffd0    daddiu  sp,sp,-48  //开辟 48 字节空间的栈
  120000b3c:   ebbe00bf    gssq    ra,s8,32(sp) //ra，s8 寄存器值入栈
```

```
120000b40:   ffbc0018    sd   gp,24(sp)          // gp 寄存器值入栈
...

120000ba0:   67bd0030   daddiu  sp,sp,48     //恢复栈现场
120000ba4:   03e00008   jr    ra                 //函数返回
120000ba8:   00000000   nop
```

从上面的例子来看，我们似乎只用一个 sp 寄存器就管理了整个函数栈，完全不需要 fp（s8）寄存器。那么 fp 什么时候会用到呢？

首先我们要知道每个进程都有自己的栈。所有函数都在这同一个栈上分配空间来存储和本函数相关的信息。每个函数所使用的那部分空间就叫栈针（stack frame）。sp 和 fp 就限定了每个函数的栈边界，如图 3-3 所示。

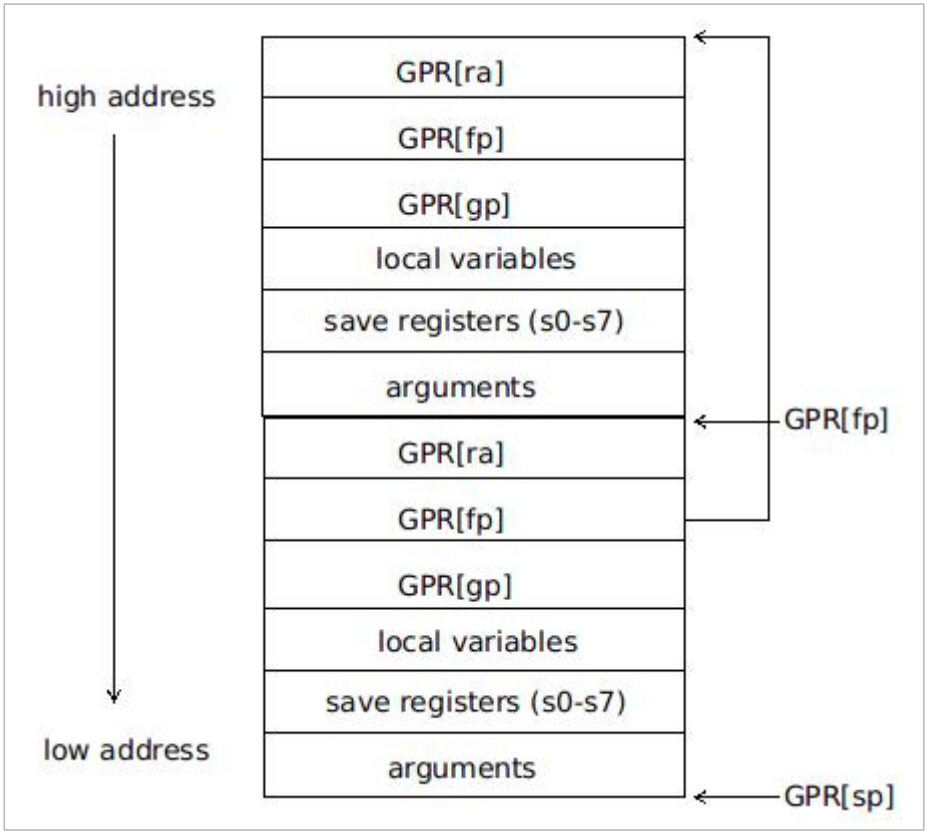


图 3-3 进程栈空间

在图 3-3 中，所有的函数栈都是在当前进程的栈空间中按照向下增长的方式分配出来的一段空间。有了 fp 和 sp 就界定了每个函数栈的边界。每个函数栈内部的 fp 都指向了调用者函数的栈顶。fp 的好处就在于当我们需要回溯函数调用关系或者动态堆栈管理时，通过当前函数里的 sp 和 fp，就可以得到上一个函数的 sp 和 fp，以此类推直到第一个函数。

3.2 整数乘法寄存器 HI、LO

MIPS 中和整数乘除法相关的寄存器有两个：HI 寄存器和 LO 寄存器。在 64 位处理器上，对于两个 32 位做乘法运算后，可以产生 64 位结果。可以临时将结果的低 32 位放在 LO 寄存器，高 32 位放在 HI 寄存器。例如下面的指令：

```
mult    t0,t1
mflo    a4
mfhi    a5
```

上面的指令”mult t0,t1”实现的是寄存器 t0 和寄存器 t1 的乘法操作，结果的低 32 位存放在寄存器 LO，结果的高 32 位存放在寄存器 HI。mult 中的 mul 代表乘法，t 代表临时存储（temply）。HI 和 LO 是特殊的寄存器，不允许程序直接使用，里面的结果要通过指令 mflo 和 mfhi 获取。“mflo a4”就是拷贝寄存器 LO 的值到通用寄存器 a4，“mfhi a5”就是拷贝寄存器 HI 的值到通用寄存器 a5。

HI 和 LO 寄存器也用于除法运算结果的临时保存。除法运算结果商临时存放在寄存器 LO，余数存放在寄存器 HI。使用实例如下：

```
div     t0,t1
mflo    a4
mfhi    a5
```

上面指令“div t0,t1”实现的是寄存器 t0 和 t1 的除法运算，运算结果的商存放在寄存器 LO，余数存放在寄存器 HI。类似于 LO = t0/t1，HI = t0%t1。在 HI 和 LO 内的数据还是要通过指令 mflo 和指令 mfhi 拷贝到通用寄存器才可以使用。

3.3 协处理器 CP0

在 MIPS 体系结构中，可支持多个协处理器(Co-Processor)。其中，协处理器 0（简称 CP0）是体系结构中必须实现的。CP0 就是系统控制处理器，它起到控制 CPU 的作用，比如 CPU 配置、高速缓存控制、异常中断控制、存储单元控制、定时器、错误检测等。CP0 包含 32 个寄存器，本节中只介绍有助于我们调试程序的几个关键寄存器。

注意：和 CP0 相关的寄存器和指令在用户态是没有权限使用的。用户态指的是非特权状态，在此状态下，执行的代码被硬件限定而不能进行某些操作。与此相对的是内核态（特权状态），在此状态下程序可以进行任何操作。通常我们的应用程序都是工作在用户态。

3.3.1 EPC 寄存器

异常程序计数器（Exception Program Counter 简称 EPC）。EPC 是一个 64 位可读写寄存器,其存储了异常处理完成后继续开始执行的指令的 地址。在 MIPS 体系架构中，中断、自陷、系统调用、程序错误等事件都称为异常。回到我们第一章程序崩溃时捕获的例子：

potentially unexpected fatal signal 8.

CPU: 1 PID: 10132 Comm: exception Not tainted 3.10.84-22.fc21.loongson.10.mips64el #1

Hardware name: /Loongson-3A5-780E-1w-V1.1-demo, BIOS Loongson-PMON-V3.3-

\x9c\x9f\xffffffe4\xffffffb8

task: 980000017d3e6d00 ti: 9800000174afc000 task.ti: 9800000174afc000

$ 0 : 0000000000000000 0000000000000001 0000000000000000 000000000000000a

$ 4 : 0000000000000001 000000ffff8303e8 000000ffff8303f8 0000000000000000

$ 8 : 000000ffee7f3820 000000ffee81fbe8 000000ffff8303e0 0080000000000000

$12 : 000000ffee631140 0000000000000003 00000000f63d4e2e 000000ffee841e28

$16 : 000000ffee7f1cc8 0000000120000b90 0000000000000000 0000000000000000

$20 : 0000000126d8ad60 0000000126c96170 0000000000000000 0000000120158008

$24 : 0000000000000000 0000000120000ad0

$28 : 0000000120019010 000000ffff830260 000000ffff830260 0000000120000b78

Hi   : 0000000000000000

Lo   : 0000000000000000

epc : 0000000120000b04 0x120000b04

  Not tainted

ra   : 0000000120000b78 0x120000b78

Status: e400ccf3 KX SX UX USER EXL IE

Cause : 10000034

PrId : 0014630d (ICT Loongson-3)

这里面“PID: 10132”表明此程序运行的进程号为 10132。从$0 至$28 行分别记录了发生异常时 32 个通用寄存器的值。Hi 和 Lo 记录了发生异常时的寄存器 HI 和 LO 的数据结果。“unexpected fatal signal 8”可知引起此次程序异常的信号值为 8,对应的信号名可以通过/usr/include/asm/signal.h 里查找到 8 即为 SIGFPE（浮点数异常）。这里 EPC 的值为 0x120000b04，通过 objdump 反汇编程序后，找到 0x120000b04 附近的指令如下：

0000000120000ad0 <test>:

120000ad0: 67bdffd0 daddiu sp,sp,-48

120000ad4: ebbe00bf gssq ra,s8,32(sp)

120000ad8: ffbc0018 sd gp,24(sp)

...

120000afc: 8fc20004 lw v0,4(s8)

120000b00: 0062001a div zero,v1,v0

120000b04: 004001f4 teq v0,zero,0x7

120000b08: 00001810 mfhi v1

120000b0c: 00001012 mflo v0

这就可以看出异常发生在 test 函数内。发生异常的是一条除 0 指令“div zero,v1,v0”。注意异常指令执行后，EPC 已经指向下一条指令。此时你就可以去程序代码中找原因去了。上述异常我使用的 C 语言实例如下：

```
void test(){
   int a = 10，b = 0;
  int c = a/b;    //非法除 0操作
}
```

3.3.2 无效地址寄存器 BadVaddr

BadVAddr 寄存器是一个 64 位只读寄存器,这个寄存器保存引发异常的地址。如果程序中发生了非法或无效地址访问、地址没有正确对齐时，该寄存器都会被设置。比如我故意编写了一段地址错误的语句：

```
int* ep = NULL;
int c = *ep; //无效地址访问
```

包含这段语句的程序编译运行后会报错如下：

potentially unexpected fatal signal 11.
CPU: 0 PID: 13610 Comm: exception Not tainted 3.10.84-22.fc21.loongson.10.mips64el #1
...
epc : 0000000120000ac4 0x120000ac4

Not tainted
ra : 0000000120000b0c 0x120000b0c
Status: e400ccf3 KX SX UX USER EXL IE
Cause : 10000008
BadVA : 0000000000000000
PrId : 0014630d (ICT Loongson-3)

这里“unexpected fatal signal 11”指明了异常信号为 SIGSEGV。“BadVA : 0000000000000000”异常地址为 0。也就是 NULL 指针，出错位置通过 epc 可以看出在 0x120000ac4

3.4 浮点寄存器

浮点寄存器也称协处理器 1（Co-Processor 1 简称 CP1）。MIPS 拥有 32 个浮点寄存器，记为$f0-$f31。每个浮点寄存器为 64 位。这 32 个浮点寄存器在使用约定（ABI）上和通用寄存器一样，也有自己一套说明。说明如表 3-2

表 3-2 n64 浮点寄存器 ABI

浮点寄存器编号	功能简介
$f0,$f2	用作函数返回值
$f12-$f19	用作传递参数
$f24-$f31	寄存器变量，发生函数调用时要保存
$f1、$f3-$f11、$f20-$23	用作临时变量

在表 3-2 中，已经对 n64 的浮点寄存器使用约定做了简单介绍。功能的分配基本上和通用寄存器相同。比如一个函数返回值为整数或指针时，使用通用寄存器的 v0，返回值为浮点类型时，就使用浮点寄存器 f0。函数调用时的参数是整型就用通用寄存器 a0-a7 传递，如果是浮点类型就用 $f12-$f19 传递。对于函数返回地址、栈指针等还是使用通用寄存器的 ra、sp。

比如我们要实现两个浮点数的加法运算，用 c 语言实现就是：

```
float c = fa+fb;
```

同样的功能，对应的汇编指令如下：

```
add.s    $f0,$f0,$f1
```

指令 add.s 实现的是对两个浮点寄存器的加法操作。上面的“add.s $f0,$f0,$f1”意思是浮点寄存器$f0 和$f1 的加法运算，结果存入$f0。相当于$f0=$f0+$f1。由于浮点运算的性能和功耗会低于整数，平时使用又不是很多，所以浮点寄存器的使用不做展开介绍，阅读 mips 汇编代码时能认出浮点寄存器就可以了。

第 4 章　MIPS 指令集

计算机指令就是指挥计算机工作的命令。程序就是一系列按一定顺序排列的指令，执行程序的过程就是计算机的工作过程。通常一条指令包括两方面的内容：操作码和操作数。操作码决定要完成的操作，操作数指参加运算的数据及其所在的单元地址。在第 2 章时我们介绍了 MIPS 所有指令都是 32 位的，操作码占用指令的高 6 位（bit31-bit26）表示，操作数占用指令的低 26 位（bit25-bit0）。MIPS 指令按格式划分为 R-型、I-型和 J-型指令，按功能划分为运算指令、访存指令、跳转指令、控制指令等。在第 2 章介绍过了 MIPS 的 R-型、I-型和 J-型指令。本章将按 MIPS 指令的功能划分，分别介绍运算指令、访存指令、跳转和分支指令、控制指令、其他指令和原子指令。

同时在学习 MIPS 指令时，需要了解 64 位处理器中，CPU 一次可以从主存加载的操作数可以有 1 字节、2 字节、4 字节和 8 字节。c 语言中的基本数据类型和 MIPS 汇编中的类型对应对应关系和汇编助记符如下表 4-1 所示。

表 4-1 c 语言数据类型大小和汇编助记符

c 语言	MIPS 名称	大小（字节）	汇编助记符
char	byte	1	lb 中的“b”
short	halfword	2	lh 中的“h”
int	word	4	lw 中的“w”
long(long long)	dword	8	ld 中的“d”

理解了表 4-1 后，在接下来的汇编语言学习中，加载数据时就能清楚地知道该使用那种指令。

4.1 运算指令

运算指令用于完成寄存器值的加减、乘除、逻辑、移位操作。运算指令包含了寄存器指令格式(R 型)和立即数指令格式(I 型)，运算方式有。 下面我们分别详细介绍。

4.1.1 加减运算

加减运算就是加法和减法运算，都是数学基本的四则运算（加减乘除）之一，也是汇编语言使用频率最高的运算指令。加减运算指令包括 32 位操作数加法和减法指令、64 位操作数的加法和减法指令、带立即数的加减法指令。同时加减法指令又分为有溢出检测和无溢出检测运算。具体的指令如表 4-2 所示。

表 4-2　加法/减法运算指令集

指令	格式	功能概述

ADD	ADD rd,rs,rt	加法
ADDU	ADDU rd, rs, rt	无溢出检测加法
ADDI	ADDI rd,rs,immediate	带立即数的加法
ADDIU	ADDIU rt, rs, immediate	无溢出检测的带立即数的加法
DADD	DADD rd, rs, rt	64 位加法
DADDU	DADDU rd, rs, rt	64 位无溢出检测加法
DADDI	DADDI rt, rs, immediate	64 位带立即数加法
DADDIU	DADDIU rt, rs, immediate	64 位无溢出检测立即数加法
SUB	SUB rd, rs, rt	减法
SUBU	SUBU rd, rs, rt	无溢出检测的减法
DSUB	DSUB rd, rs, rt	64 位减法
DSUBU	DSUBU rd, rs, rt	64 位无溢出检测的减法

在表 4-2 中，指令最前面的 D 代表 Double，即表示 64 位操作数的指令，否则就是 32 位指令。。格式中的 rd 代表目标操作数、rs 代表第一个源操作数、rt 代表第二个操作数。rd,rs,rt 可以是通用寄存器中的任意一个。指令最后面的“U”代表“无溢出检测”，否则就是有溢出检测。比如下面例子：

```
add t0,a0,a1    # t0=a0+a1
addu t0,a0,a1  # t0=a0+a1
```

这两句都实现的是 a0+a1,结果存入 t0。但是 add 指令对加结果有溢出检测（仅支持有符号运算），如果 a0+a1 结果有溢出的话，那么 t0 不会被改变，程序会报“整型溢出异常”。而 addu 指令表示“无溢出检测”，a0+a1 的结果直接存入 t0。
c 和 c++语言不支持“整型溢出异常”，所以总是使用带”U”的指令。

这里的“I”代表立即数。立即数就是嵌入在指令中的常数。加法运算中的立即数都占用 16 位，所以都需要有符号扩展到 32 位或者 64 位，然后再参与加法运算，比如下面的指令：

```
addiu t0,a0,0x3     # t0=a0+sign_extend(0x3)
daddiu t0,a0,0x3   #t0=a0+sign_extend(0x3)
```

同样是实现寄存器 a0 和立即数 0x3 的加法运算，但是 addiu 首先把立即数 0x3 进行有符号扩展(sign_extend)到 32 位，然后和寄存器 a0 进行加法操作，正确结果被存入 t0。而 daddiu 实现的是 64 位加法运算，所以需要先把立即数 0x3 进行有符号扩展(sign_extend)到 64 位，然后和寄存器 a0 进行加法操作，正确结果被存入 t0。

这里解释一下为何要对立即数进行有符号扩展。下图是 addiu 指令的具体格式：

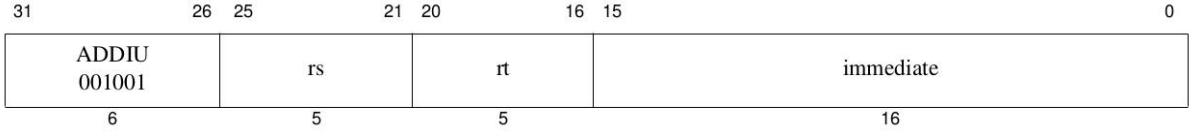


图 4-2 addiu 指令格式

从上图可以看出，addiu 是 I 型指令，立即数占用 16 位。而 addiu 实现的是 32 位加法操作，所以需要把立即数进行符号扩展到 32 位，然后和 rs(a0)进行加法操作，结果存入 rt。

在介绍一下符号扩展。符号扩展是指计算机对于小字节转换成大字节的规则。比如要 16 个字节的数转换成 32 字节，多出来的 16 个字节到底怎么填充？mips 指令集中会经常需要将其中的立即数进行符号扩展。符号扩展分无符号扩展和有符号扩展，区别如下：

16 位立即数	0x8000	0xe000
有符号扩展到 32 位	0xffff8000	0x0000e000
无符号扩展到 32 位	0x00008000	0x0000e000

表 4-3 符号扩展方式

从上表可以看出，有符号扩展就是高位填充的值取决于这个数扩展前的最高位。扩展前的数高位为 1,那么扩展出来的高位就都是 1，扩展前的数高位为 0，那么扩展出来的高位就都是 0。而无符号扩展就是无论扩展前的数是多少，扩展出来的高位都是 0。

减法指令中没有带立即数的操作，只区别是 32 位和 64 位减法，减法是否有溢出。比如：

```
sub t0,a0,a1        # t0 = a0-a1
dsubu t0,a0,a1      # t0 = a0-a1
```

都是实现 a0 减 a1，结果存入 t0 操作。dsubu 支持 64 位的减法，而且没有整数溢出异常。

4.1.3 乘除运算

MIPS 中乘法和除法指令如下表：

指令	格式	功能简述
MUL	MUL rd, rs, rt	乘法运算
MULT	MULT rs, rt	乘法运算，结果低 32 位存入 LO，高 32 位存入 HI
MULTU	MULTU rs, rt	无符号乘法运算，结果低 32 位存入 LO，高 32 位存入 HI
DMULT	DMULT rs, rt	64 位乘法运算，结果低 64 位存入 LO，高 64 位存入 HI
DMULTU	DMULTU rs, rt	64 位无符号乘法运算，结果低 64 位存入 LO，高 64 位存入 HI
MADD	MADD rs, rt	乘加运算，(HI,LO) ← (HI,LO) + (GPR[rs] × GPR[rt])
MADDU	MADDU rs, rt	无符号的乘加运算
MSUB	MSUB rs, rt	乘减运算，(HI,LO) ← (HI,LO) - (GPR[rs] × GPR[rt])
MSUBU	MSUBU rs, rt	无符号乘减运算
DIV	DIV rs, rt	除法运算，商存入 LO，余数存入 HI
DDIV	DDIV rs, rt	64 位除法运算，商存入 LO，余数存入 HI

DIVU	DIVU rs, rt	无符号除法运算，商存入 LO，余数存入 HI
DDIVU	DDIVU rs, rt	64 位无符号除法运算，商存入 LO，余数存入 HI
MFHI	MFHI rd	拷贝 HI 到 rd
MTHI	MTHI rs	拷贝 rs 到 HI
MFLO	MFLO rd	拷贝 LO 到 rd
MTLO	MTLO rs	拷贝 rs 到 LO

表 4-4 乘法/除法指令

上表中完成了和乘法/除法相关的算术运算。运算过程中，操作数（rs,rt）默认按有符号计算，如需完成无符号数的乘法/除法操作，指令里面要带“U”，所以这张表里面的“U”意为”无符号”。”T”代表”结果临时存储“，也就是需要特殊寄存器 HI/LO 存储运算结果。

1.乘法运算指令

下面举例说明：

```
mul t0,a0,a1        #t0 = a0*a1
mult a0,a1          # (HI,LO) ← (a0*a1)
multu a0,a1         # (HI,LO) ← (a0*a1)   a0 和 a1 均为无符号数
dmult a0,a1         # (HI,LO) ← (a0*a1) 64 位运算
```

mul 实现了 2 个 32 位有符号数乘法运算，得出可以是 64 位的结果并存入 t0。mult 也实现两个 32 位有符号数乘法运算，得出可以是 64 位的结果，但是结果的低 32 位有符号扩展到 64 位后存入寄存器 LO，高 32 位有符号扩展到 64 位后存入寄存器 HI。multu 同 mult 指令，区别是 a0 和 a1 按 32 位无符号数运算。

dmult 实现的是两个 64 位有符号数乘法运算，可以得出一个 128 位的结果。结果的低 64 位存入 LO。高 64 位存入 HI。

2.乘加和乘减运算指令

接下来在看另一组运算：

```
madd a0,a1          #  (HI,LO) ← (HI,LO) + (a0*a1)
maddu a0,a1         #  (HI,LO) ← (HI,LO) + (a0*a1)  a0 和 a1 均为无符号数
msub a0,a1          #  (HI,LO) ← (HI,LO) - (a0*a1)
msubu a0,a1         #   (HI,LO) ← (HI,LO) - (a0*a1) a0 和 a1 均为无符号数
```

madd 实现了两个 32 位寄存器的有符号乘法运算后,运算结果的低 32 位累加到 LO 寄存器，高 32 位累加到 HI 寄存器。maddu 功能同 madd，区别是操作数 a0 和 a1 为无符号数。

msub 实现了两个 32 位寄存器的有符号乘法运算后,结果的低 32 位累减到 LO 寄存器，高 32 位累减到 HI 寄存器。msubu 功能同 msub，区别是操作数 a0 和 a1 为无符号数。

3.除法运算指令

下面再看除法运算：

```
div a0,a1       # (HI, LO) ← a0 / a1
```

```
divu a0,a1     # (HI, LO) ← a0 / a1 ,a0 和 a1 均为无符号数
ddiv  a0,a1    # (HI, LO) ← a0 / a1
ddivu a0,a1    # (HI, LO) ← a0 / a1 ,a0 和 a1 均为无符号数
```

div 实现两个有符号 32 位寄存器的除法运算，商有符号扩展到 64 位存放在 LO 寄存器，余数有符号扩展到 64 位后存放在 HI 寄存器。divu 功能同 div，区别是操作数 a0 和 a1 为无符号数。

ddiv 实现两个有符号 64 位寄存器的除法运算，商存放在 LO 寄存器，余数存放在 HI 寄存器。ddivu 功能同 ddiv，区别是操作数 a0 和 a1 为无符号数。

4.HI、LO 相关指令

上面介绍的和乘法/除法相关的运算时涉及到两个寄存器 HI、LO。这两个寄存器不是通用寄存器。所以需要把里面的结果复制到通用寄存器才可以使用。下面介绍和 HI、LO 寄存器相关的指令：

```
mfhi t0   #t0 ← HI
mflo t0   #t0 ← LO
mthi t0    #HI ← t0
mtlo t0    #LO ← t0
```

上面 4 条指令完成了 HI、LO 和通用寄存器的互拷贝操作。指令中“f”意为 from，“t”意为 to。mfhi 实现把 HI 寄存器值拷贝到 t0；mflo 实现把 LO 寄存器值拷贝到 t0。mthi 实现把 t0 拷贝到 HI 寄存器，mtlo 实现把 t0 拷贝到 LO 寄存器。

4.1.4 逻辑运算

MIPS 中逻辑运算指令包括与、或、非、条件设置等，具体如下表所示：

指令	格式	功能简述
SLT	SLT rd, rs, rt	小于设置 if GPR[rs] < GPR[rt] then GPR[rd]=1 else GPR[rd]=0
SLTI	SLTI rt, rs, immediate	小于设置 if GPR[rs] < immed then GPR[rd]=1 else GPR[rd]=0
SLTU	SLTU rd, rs, rt	无符号小于设置
SLTIU	SLTIU rt, rs, immediate	无符号小于立即数设置
AND	AND rd, rs, rt	与运算 rd = rs and rt
ANDI	ANDI rt, rs, immediate	与立即数 rt = rs and immediate
OR	OR rd, rs, rt	或运算 rd = rs or rt
ORI	ORI rt, rs, immediate	或立即数 rt = rs or immediate
XOR	XOR rd, rs, rt	异或 rd = rs ^ rt
XORI	XORI rt, rs, immediate	异或立即数 rt = rs ^ immediate
NOR	NOR rd, rs, rt	或非 rd = rs nor rt
LI	LI rt, immediate	取立即数到寄存器。
LUI	LUI rt, immediate	取立即数到高位 GPR[rt] ← sign_extend(immediate || 0 16)
CLO	CLO rd, rs	字前导 1 个数

DCLO	DCLO rd, rs	双字前导 1 个数
CLZ	CLZ rd, rs	字前导 0 个数
DCLZ	DCLZ rd, rs	双字前导 0 个数
WSBH	WSBH rd, rt	半字中字节交换
DSHD	DSHD rd, rt	字中半字交换
DSBH	DSBH rd, rt	双字中的半字中字节交换
SEB	SEB rd, rt	字节符号扩展
SEH	SEH rd, rt	半字符号扩展
INS	INS rt, rs, pos, size	位插入， pos 取值 0-31 size 取值 1-32 。pos+size 取值 (1-32)
DINS	DINS rt, rs, pos, size	双字位插入 pos 取值 0-31 size 取值 1-32 pos+size 取值 (1-32)
DINSM	DINSM rt, rs, pos, size	双字位插入 pos 取值 0-31 size 取值 2-64，pos+size 取值 (33-64)
DINSU	DINSU rt, rs, pos, size	双字位插入 pos 取值 32-63, size 取值 1-32，pos+sizee 取值 (33-64)
EXT	EXT rt, rs, pos, size	位提取 pos 取值 0-31, size 取值 1-32，pos+size 取值 (1-32)
DEXT	DEXT rt, rs, pos, size	双字位提取 pos 取值 0-31, size 取值 1-32，pos+size 取值 (1-63)
DEXTM	DEXTM rt, rs, pos, size	双字位提取 pos 取值 0-31, size 取值 33-64，pos+size 取值 (33-64)
DEXTU	DEXTU rt, rs, pos, size	双字位提取 pos 取值 32-63, size 取值 1-32，pos+size 取值 (33-64)

表 4-5 逻辑运算

这张表中的 U 代表无符号数。

1.条件设置指令

条件设置指令就是指根据条件判断的结果来决定指令的执行，具体举例如下：

```
slt a3,t0,t1        #if(t0<t1)  a3=1 else a3=0
slti a3,t0,0x3      #if(t0<0x3)  a3=1 else a3=0
sltu a3,t0,t1       #if(t0<t1)  a3=1 else a3=0
sltiu  a3,t0,0x3    #if(t0<0x3)  a3=1 else a3=0
```

slt 指令比较两个操作数寄存器 t0 和 t1 的大小，结果为真则目标寄存器 a3 置 1,结果为假则目标寄存器 a3 置 0。slti 功能同 slt，只是操作数之一是个立即数。sltu 功能同 slt，就是要求两个操作数必须是无符号数。sltiu 功能也同 slt，就是要求操作数之一是立即数，且两个操作数都为无符号数。

与、或、非等相关的指令比较简单，这里不展开解释。

LI、LUI 都是加载立即数到寄存器。立即数大小是 16 位。如果超过 16 位将会被编译拆分成多条指令。指令 LUI（Load Upper Immediate），这里的 U 代表 Upper。就是加载一个 16 位的立即数放到目标寄存器 rt 的高 16 位上，rt 低 16 为为 0。例如我在内嵌汇编里面写了下面语句：

```
li v1,0x3ffff
```

gcc 编译后会把它转化成：

```
lui   v1,0x3   # v1 = 0x30000
ori   v1,v1,0xffff # v1 = 0x30000 | 0xffff
```

指令 CLO（Count Leading Ones）对地址为 rs 的通用寄存器的值，从其最高位开始向最低位方向检查，直到遇到值为“0”的位，将该位之前“1”的个数保存到地址为 rd 的通用寄存器中，如果地址为 rs 的通用寄存器的所有位都为 1（即 OxFFFFFFFF），那么将 32 保存到地址为 rd 的通用寄存器中。而指令 CLZ（Count Leading Zeros）和 CLO 相反，对地址为 rs 的通用寄存器的值，从其最高位开始同最低位方向检查，直到遇到值为“1”的位，将该位之前“0”的个数保存到地址为 rd 的通用寄存器中，如果地址为 rs 的通用寄存器的所有位都为 0（即 0x00000000），那么将 32 保存到地址为 rd 的通用寄存器中。DCLO 和 DCLZ 分别是 CLO 和 CLZ 的 64 位运算形式。

指令 INS（Insert Bit Field）实现的是 32 位寄存器的域值插入，就是将操作数 rs 的低 size 位，插入到目标操作数的（pos 到 pos+size）位置。此处 rs,rt 均为 32 位数。举例说明：

```
ins a3,t0,0x5,0x3
```

上面 INS 指令的意思是将寄存器 t0 的低 3 位，放到目标寄存器 a3 的第 5 位到第 7 位（bit5-bit7）。DINS 指令功能同 INS，都实现的是目标操作数 rt 的低 32 位的域值设置。区别在于 rs,rt 可以是 64 位数。而 DINSM 和 DINSU 中的 M 代表 Middle,U 代表 Upper,实现的是对目标操作数 rt 的高 32 位的域值设置。通常情况我们只要使用 DINS 指令就可以，汇编器会根据需要选择是 DINSU 还是 DINSM。

指令 EXT 实现 32 位寄存器的域值提取，就是将操作数 rs 的第 pos 位到第（pos+size-1）位提取出来，结果保存到目标寄存器 rt 中。例如：

```
ext a3,t0,0x5,0x3
```

上面的 ext 指令的意思是将寄存器 t0 的第 5 位到第 7 位提取出来存放到寄存器 a3。相关指令还有 DEXT,实现的是 64 位操作数的域值部分提取。而 DEXTM 和 DEXTU 分别完成 64 位操作数 rs 的高 32 位（bit32-bit63）域值的部分提取。通常情况我们只要使用 DEXT 指令就可以，汇编器会根据需要选择是 DEXTU 还是 DEXTM。

指令 SEB 和 SEH 实现的是符号扩展，B 代表 Byte（8 位）、H 代表 HalfWord（16 位）。命令比较简单，这里不做举例。

指令 WSBH 实现一个 32 位操作数 rt 内的两个半字内部进行字节交换，结果存入 rd。形式类似于：

```
rd =  rt( 23..16 )  || rt( 31..24 ) || rt( 7..0) || rt( 15..8 );
```

DSBH 功能类似于：

rd = rt(55.48) || rt(63..56) || rt(39..32) || rt(47..40) ||
rt(23..16) || rt(31..24) || rt(7..0)|| rt(15..8);

DSHD 功能类似于：

rd = rt(15..0) || rt(31..16) || rt(47..32) || rt(63..48);

4.1.5 移位运算

移位操作指令是一组经常使用的指令，属于汇编语言逻辑指令中的一部分，它包括逻辑左移、逻辑右移动、算数右移、循环移位。其功能为将目的操作数的所有位按操作符规定的方式移动 n 位结果送入目的地址。mips 中移位相关指令如下表：

指令	格式	功能简述
SLL	SLL rd, rt, sa	逻辑左移GPR[rd] ← GPR[rt] << sa， sa 为常量 取值 0-31
SLLV	SLLV rd, rt, rs	可变的逻辑左移 GPR[rd] ← GPR[rt] << GPR[rs] ， rs 取值 0-31
DSLL	DSLL rd, rt, sa	双字逻辑左移，0<=sa<32
DSLLV	DSLLV rd, rt, rs	可变的双字逻辑左移
DSLL32	DSLL32 rd, rt, sa	移位量加 32 的双字逻辑左移 GPR[rd] ← GPR[rt] << (sa+32)
SRL	SRL rd, rt, sa	逻辑右移
SRLV	SRLV rd, rt, rs	可变的逻辑右移
DSRL	DSRL rd, rt, sa	双字逻辑右移
DSRLV	DSRLV rd, rt, rs	可变的双字逻辑右移
DSRL32	DSRL32 rd, rt, sa	移位量加 32 的双字逻辑右移 GPR[rd] ← GPR[rt] << (sa+32)
SRA	SRA rd, rt, sa	算术右移
SRAV	SRAV rd, rt, rs	可变的算数右移
DSRA	DSRA rd, rt, sa	双字算术右移
DSRAV	DSRAV rd, rt, rs	可变的双字算术右移
DSRA32	DSRA32 rd, rt, sa	移位量加 32 的双字算术右移 GPR[rd] ← GPR[rt] >> (sa+32)
ROTR	ROTR rd, rt, sa	循环右移 temp ← GPR[rt] sa-1..0 || GPR[rt] 31..sa rd ← sign_extend(temp)
ROTRV	ROTRV rd, rt, rs	可变的循环右移
DROTR	DROTR rd, rt, sa	双字循环右移
DROTR32	DROTR32 rd, rt, sa	移位量加 32 的双字循环右移
DROTRV	DROTRV rd, rt, rs	双字可变的循环右移

表 4-6 移位运算

逻辑左移 SLL（Shift Logical Left），就是 rt 向左（高位）移动 sa 位，sa 是常量，取值范围在 0-31 之间（SLL 指令是 R 型，sa 占用 5bit），空出的右边（低位）补 0。和 SLL 相关的指令运算举例如下：

```
sll a3,t1,0x3        #a3 = t1 << 0x3
sllv a3,t1,t0        #a3 = t1 << t0 ， 0<=t0<31
dsll a3,t1,0x3       #a3 = t1 << 0x3  t1 为 64 位
dsll32 a3,t1,0x3     #a3 = t1 << （0x3 +32）
dsllv a3,t1,t0       #a3 = t1 << t0 ， 0<=t0<31 t1 为 64 位
```

这个例子中 sll 就是把一个 32 位操作数 t1 左移 3 位，低 3 位添 0，结果符号扩展到 64 位后存入 a3 寄存器。sllv 功能同 sll，区别在于 sllv 中的 v 意为 variable，就是要移位的位数不是常量，而是寄存器。dsll 实现的是一个 64 位操作数 t1 左移 3 位，低 3 位添 0，高位溢出后丢弃，结果存入 a3 寄存器。dsll32 功能同 dsll，区别在于 sa 需要加 32。dsllv 功能同 sllv，区别在于 t1 表示 64 位数。

逻辑右移 SRL（Shft Right Logical）,就是 rt 向右（低位）移动 sa 位，sa 取值范围在 0-31，空出的左边（高位）补 0。和 SRL 相关的指令运算举例如下：

```
srl a3,t1,0x3        #a3 = t1 >> 0x3
```

这个例子中 srl 就是把一个 32 位操作数 t1 右移 3 位，空出的高 3 位添 0，结果符号扩展到 64 位存入寄存器 a3。其他几组 srlv、dsrl、dsrlv、dsrl32 功能相信读者可以自行推倒出来。

算数右移 SRA(Shift Right Arithmetic),就是实现 rt 向右移位，空出来的左边（高位）是补 0 还是 1 取决于 rt 的最高位（32 位 rt 就取 bit31，64 位 rt 就取 bit63），可移动的范围为 0-31。

```
sra a3,t1,0x3        #a3 = t1 << 0x3
```

sra 实现一个 32 位操作数 t1 的右移 3 位，空出来的高 3 位填充 t1 的 bit31 值，结果符号扩展后存入寄存器 a3。和算数右移相关的还有指令 srav、dsra、dsrav、dsra32。

循环右移就是操作数向右移动 n 位，然后把移出的 n 位顺序填充左边 n 位。比如：

```
rota a3,t1,0x3
```

rota 实现一个 32 位操作数向右循环移位，循环量是 3。就是把 t1 低 3 位放入 t1 的高 3 位，t1 的高 29 位向右移动 3 位。ROTRV 功能同 ROTA，只是移位量不是常数而是寄存器，不过寄存器取值范围同 SA（0-31）。

4.2 访存指令

访存指令就是实现对主存中的数据进行访问或存储。MIPS 体系结构采用 load/store 架构。所有运算都在寄存器上进行,只有访存指令可对主存中的数据进行访问。访存指令包含各种宽度数据的读写、无符号读、非对齐访存和原子访存等。MIPS 中访存指令如下：

指令	格式	功能简述
LB	LB rt, offset(base)	8 位加载，
LBU	LBU rt, offset(base)	8 位加载，结果 0 扩展
LH	LH rt, offset(base)	加载半字
LHU	LHU rt, offset(base)	加载半字，结果 0 扩展
LW	LW rt, offset(base)	加载字
LWU	LWU rt, offset(base)	加载字，结果 0 扩展
LWL	LWL rt, offset(base)	加载字的左部
LWR	LWR rt, offset(base)	加载字的右部
LD	LD rt, offset(base)	加载双字
LDL	LDL rt, offset(base)	加载双字左部
LDR	LDR rt, offset(base)	加载双字右部
LL	LL rt, offset(base)	加载标志处地址
LLD	LLD rt, offset(base)	加载标志处双字地址
SB	SB rt, offset(base)	存储节
SH	SH rt, offset(base)	存储半字
SW	SW rt, offset(base)	存储字
SWL	SWL rt, offset(base)	存储字的左部
SWR	SWR rt, offset(base)	存储字的右部
SD	SD rt, offset(base)	存储双字
SDL	SDL rt, offset(base)	存储双字左部
SDR	SDR rt, offset(base)	存储双字右部
SC	SC rt, offset(base)	带条件的存储
SCD	SCD rt, offset(base)	带条件的存储双字

表 4-7 访存指令

上表中 L 开头的都是从主存加载数据操作，S 开头的都是存储数据操作。操作数据的大小有 B（Byte 8bit）、H（Halfword 16bit）、W（Word 32bit）、D（Double 64bit）。加载地址都是 base+offset 方式，offset 取值范围在在-32767 至 32768（2 的 16 次方）。这一点可以从任意加载指令格式可以看出。

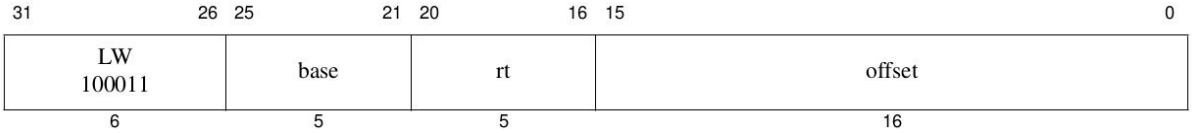


图 4-4 LW 指令格式

从图 4-4 可以看出 offset 域占用 16bit 的大小。所以 load 相关指令的可寻址范围在 base-32767 到 base+32768 之间。下面举个例子说明加载指令的使用：

```
ld    v0,-32688(gp)   # v0 ← memory[gp-32688]
lwu  v1,32(gp)        #v1 ← memory[gp + 32]
```

这里 ld 指令意思是从地址值为 gp-32688 的内存位置，加载一个 64 位的数据赋给寄存器 v0。如果地址无效，程序会收到异常信号。lwu 指令意思是从地址为 gp+32 的内存位置加载一个 32 位数据，结果无符号扩展到 64 位后存入 v1 寄存器。

存储指令和加载指令功能上恰好相反，负责把寄存器数据存储到主存。举例说明：

```
sd    ra,-352(ra)      # memory[ra -352] ← ra
sw    ra,0(s8)         # memory[s8 +0] ← v0
```

这里 sd 指令意思是把寄存器 ra 里的 64 位数写出到内存地址为（ra-352）的位置。如果地址无效，程序会收到异常信号。sw 指令意思是把寄存器 ra 里的 32 位数写出到内存地址为 s8+0 的位置。

上表中有两个指令 ll 和 sc 比较特殊，可以完成多条指令的原子操作。在本章最后一节会单独介绍。

4.3 跳转和分支指令

跳转和分支指令可改变程序的控制流。mips 中指令如下表：

指令	格式	功能简述
J	J target	绝对跳转
JR	JR rs	寄存器绝对跳转，寻址空间大于 256M
JAL	JAL target	子程序调用
JALR	JALR rs	寄存器子程序调用，寻址空间大于 256M
B	B lable	相对 PC 的跳转，寻址范围小于 256K
BAL	BAL lable	相对 PC 的子程序调用，寻址空间小于 256K
BEQ	BEQ rs, rt, lable	相等则跳转 if (rs == rt) goto lable
BNE	BNE rs, rt, lable	不等则跳转 if (rs = rt) goto lable
BLEZ	BLEZ rs, lable	小于等于 0 跳转 if (rs <= 0) goto lable
BGTZ	BGTZ rs, lable	大于 0 跳转 if (rs > 0) goto lable
BLTZ	BLTZ rs, lable	小于 0 跳转 if (rs < 0) goto lable
BGEZ	BGEZ rs, lable	大于等于 0 跳转 if (rs >= 0) goto lable
BLTZAL	BLTZAL rs, lable	带条件的相对 PC 子程序调用 if(rs<0) lable()
BGEZAL	BGEZAL rs, lable	带条件的相对 PC 子程序调用 if(rs >=0) lable()

表 4-8 跳转和分支指令

MIPS 体系架构中，跳转、分支和子程序调用的命名规则如下：

- 和 PC 相关的相对寻址指令（地址计算依赖于 PC 值）称为“分支”(branch)，助记词以 b 开头。绝对地址指令（地址计算不依赖 PC 值）称为“跳转”（Jump），助记词以 j 开头。
- 子程序调用为“跳转并链接”或“分支并链接”，助记词以 al 结尾。

上表中的第一个指令 J 表示无条件跳转到一个绝对目标地址，地址可寻址范围在 256MB（原因可以参阅第 5 章）。使用上可以是如下形式：

```
j test      #跳转到 test 标识
```

如果要跳跃到大于 256MB 的地址，可以使用 JR 指令（寄存器指令跳转）。JR 的操作数是寄存器，可以表达更大的地址空间。比如：

```
jr t9 #跳转到寄存器 t9 所存的地址
```

JAL、JALR 实现了直接和间接的子程序调用。这两个指令和上面的 J、JR 指令的区别在于，JAL、JALR 在跳转到指定地址的同时，还要保存返回地址（PC+8）到寄存器 ra（$31）。这样在函数返回可以通过 JR ra 完成。常见使用如下：

```
main:
    jal test    #跳转到 test 标识或者子程序
    nop
    ld v0 ,32696(gp)
...
test:
...
    jr ra
    nop
```

这里 jal test 指令执行时，首先会把 pc+8 的地址赋值给 ra。那么 ra 就指向了“ld v0 ,32696(gp)”的位置。然后程序跳转到 test 程序运行。test 子程序最后会通过“jr ra”指令返回到“ld v0 ,32696(gp)”的位置。

如果要跳转的子程序超过了 258M，可以使用 JALR 指令完成。这里不做介绍。

相对寻址（PC-relotive）的子程序调用可以用 BAL。BAL 指令使用 16 位存储 offset，所以可寻址范围特别小，仅为 256K。所以平时很少使用到。

BEQ、BNE、BLEZ、BGTZ、BLTZ、BGEZ 的功能都是带条件的分支跳转，即当操作数寄存器 rs 和寄存器满足一定条件时，才会跳转到 label。以 BEQ 为例：

```
    beq $8,$4,again    #if($8==$4) goto again
    nop
```

当寄存器$8 和$4 值相等时，跳转到标识符 again 位置。这里 again 的地址计算是依赖于程序计数器 PC，计算过程由汇编器帮助我们完成。我们所要注意的就是 again 的范围必须在（PC-127K）至（PC+128K）之间，否则分支跳转就会失败。跳转失败不会有异常。

BLTZAL、BGEZAL 指令 实现的是带条件的子程序调用。以 BLTZAL 指令为例：

```
bltzal a0,test  #if(a0<0) test()
nop
```

bltzal 指令意思是如果寄存器 a0 的值小于 0，那么 PC 跳转到子程序 test。这里要注意的是无论调用是否发生，返回地址一律保存到寄存器 ra 中。

4.4 协处理器 0 指令：CPU 控制指令

MIPS 的协处理器 0 指令就是 CPU 控制指令，包括数据在通用寄存器和 CP0 寄存器间的读写、管理内存、处理异常等。具体指令如下表所示：

指令	格式	功能简述
MFC0	MFC0 rt,cs	从 CP0 寄存器取字
MTC0	MTC0 rt, cs	往 CP0 寄存器写字
DMFC0	DMFC0 rt,cs	从 CP0 寄存器取双字
DMTC0	DMTC0 rt, cs	往 CP0 寄存器写双字
ERET	ERET	异常返回，返回 EPC 中保存的地址
CACHE	CACHE k,addr	Cache 操作
TLBR		读索引的 TLB 项
TLBWI		写索引的 TLB 项
TLBWR		写随机的 TLB 项
TLBP		在 TLB 中搜索匹配项

表 4-9　cpu 控制指令

MFC0、MTC0、DMFC0、DMTC0 可以完成通用寄存器和 CP0 寄存器之间的数据传送。比如我们可以通过下面指令来获取 EPC 寄存器的值。

```
mfc0 t1,$14  # 从 CP0 取 epc 寄存器（$14）数据，存到通用寄存器 t0
```

mfc0 完成从 cp0 寄存器获取一个 32 位数据（如果 cp0 寄存器是 64 位数据则只取其低 32 位。）到通用寄存器，如果要获取的是 64 位数据则需要使用 dmfc0 指令。

注意：和 CP0 相关的指令执行都需要特权模式，比如上面的 mfc0 指令的运行需要 root 权限或者 sudo 才可以运行。所以在一般用户程序中很少见到这些 CP0 相关的控制指令。

4.5 特殊指令

mips64 中，除了本章列举的运算指令、控制指令、跳转指令等，还有一些无法归类的指令，这里统称为其他指令，具体如下表：

指令	格式	功能简述
SYSCALL	SYSCALL	系统调用
BREAK	BREAK	断点
SYNC	SYNC	同步
RDHWR	RDHWR rt,rd	用户态读取硬件寄存器
WAIT	WAIT	等待指令
TEQ	TEQ rs, rt	条件自陷 if(rs == rt) exception(trap)
TNE	TNE rs, rt	条件自陷 if(rs != rt) exception(trap)
MOVE	MOVE rd, rs	移动 rd = rs
MOVZ	MOVZ rd, rs, rt	条件移动 if (rt == 0) rd = rs
MOVN	MOVN rd, rs, rt	条件移动 if (rt != 0) rd = rs

表 4-9　其他指令

SYSCALL 指令可以产生一个引发系统调用的异常，相当于 x86 的 “int 0x80”指令。它的使用方式在本书后面第 6 章会有专门的讲解。

BREAK 指令也称断点指令，它可以产生一个“断点”类型的异常。在我们调试代码时，break 指令是很有用的调试手段了。比如我们在 c 语言中插入一个 break 指令：

```
asm(“break”);
```

那么程序执行到这条指令时，就会收到一个 SIGTRAP 信号，提示信息为“Trace/breakpoint trap”。

TEQ、TNE 都是条件自陷指令。如果条件成立就会引发一个自陷（Trap）异常。自陷指令又叫做访管指令，出现在计算机操作系统中，用于实现在用户态下运行的进程调用操作系统内核程序，即当运行的用户进程或系统实用进程欲请求操作系统内核为其服务时，可以安排执行一条陷入指令引起一次特殊异常。通常自陷指令是给编译器和解释器用的，可以实现运行时数组边界检查之类的操作。比如我用 c 语言写了一个除法语句：

```
int c = a/b;
```

gcc 编译完成后，对应的汇编指令就是下面的样子：

```
div   zero,v1,v0     # lo = v1/v0   hi = v1%v0
teq   v0,zero,0x7    # if(v0 == zer0) trap
mfhi      v1         # v1 = hi
mflo      v0         # v0 = lo
```

上面的 div 实现两个有符号数的除法运算，商存入 lo，余数存入 hi。下面第二句 teq 就是判断 v0（代表 c 语言里的变量 b）是否为 0。如果为 0，那么程序自陷，0x7 会被保存到 Cause 寄存器。程序会收到信息“浮点数例外”，后面的 mfhi、mflo 将不再执行。

SYNC 是用在多核处理器的内存操作(load/store 寄存器)同步指令，用来保证 sync 之前对于内存的操作能够在 sync 之后的指令开始之前完成。比如下面指令：

```
sw v1,8208(v0) #写 v1 寄存器数据到内存地址 v0+8208
sync           # 同步
cache 21,8208(v0) #刷新缓存
```

RDHWR 指令允许用户态读取硬件寄存器（CP0 寄存器）。当然并不是所有硬件寄存器都可以读取的，而是受到 CP0 寄存器 HWREna 限制的。HWREna 决定了哪些硬件寄存器可以被用户态程序访问。当前支持可读的寄存器信息如下表：

寄存器编号（rd 值）	助记符	功能简介
0	CPUNum	用于获取当前进程正在运行在哪个 CPU 上，实际是访问 CP0 寄存器 Ebase 的 CPUNUM 域
1	SYNCI_Step	用于获取一级高速缓存行的有效宽度
2	CC	读取 CP0 的 Count 寄存器
3	CCRes	略
29	ULR	用于获取线程本地存储（TLS）的位置

表 4-10 RDHWR 寄存器说明

在第一章我们已经举例了用 RDHWR 指令获取 TLS 地址方法。这里再举例说明如何获取当前进程正在运行在那个 CPU 核上，指令如下：

```
rdhwr $2,$0
```

这里就是从 RDHWR 的 0 号寄存器获取 CPUNum 信息到通用寄存器 v0（$2）。

MOVE、MOVZ、MOVN 都实现了寄存器到寄存器的复制。MOVE 是无条件复制，例如：

```
move a1,a2   # a1=a2
```

move 就是把寄存器 a2 的值复制到寄存器 a1。这条指令和下面的指令相等：

```
movz a1,a2,zero
```

4.6 原子指令 LL/SC

mips 用 ll（load-linked）和 sc（store-conditional）两条指令实现多进程的互斥锁。也就是说用这两条指令可以实现原子操作。例如内核代码里的原子自加操作 atomic_inc(&count) ,每次调用此函数,变量 count 加 1,汇编指令如下：

```
atomic_inc:
    ll v0,0(a0)  #v0 指向了变量 count 位置（0(a0)）
    addu v0,1
    sc v0,0(a0)
    beq v0,zero,atomic_inc #如果 sc 指令失败（v0==0）,跳到 atomic_inc 重试
    nop
    jr ra
    nop
```

ll 和 sc 指令具体介绍如下图：

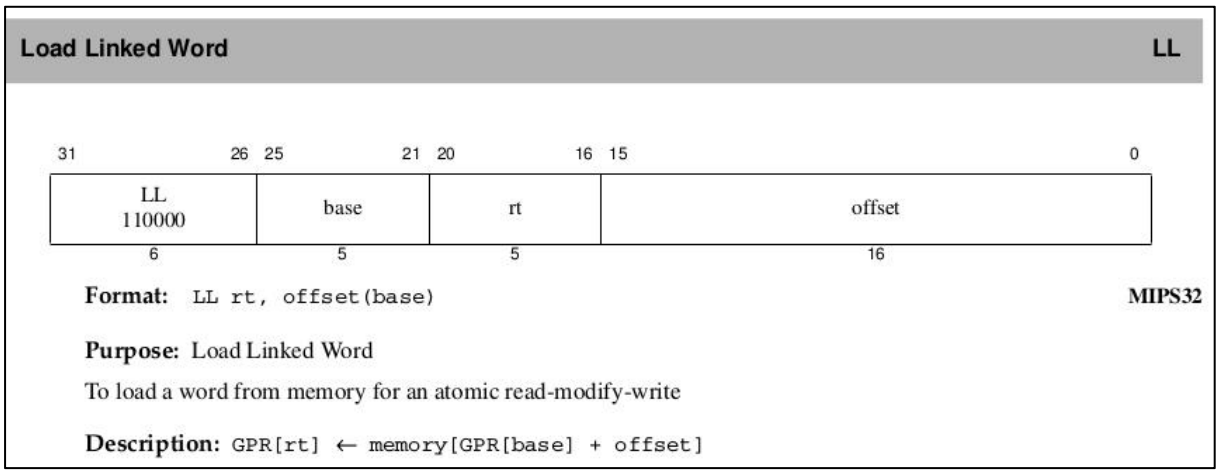


图 4-3　LL 指令格式

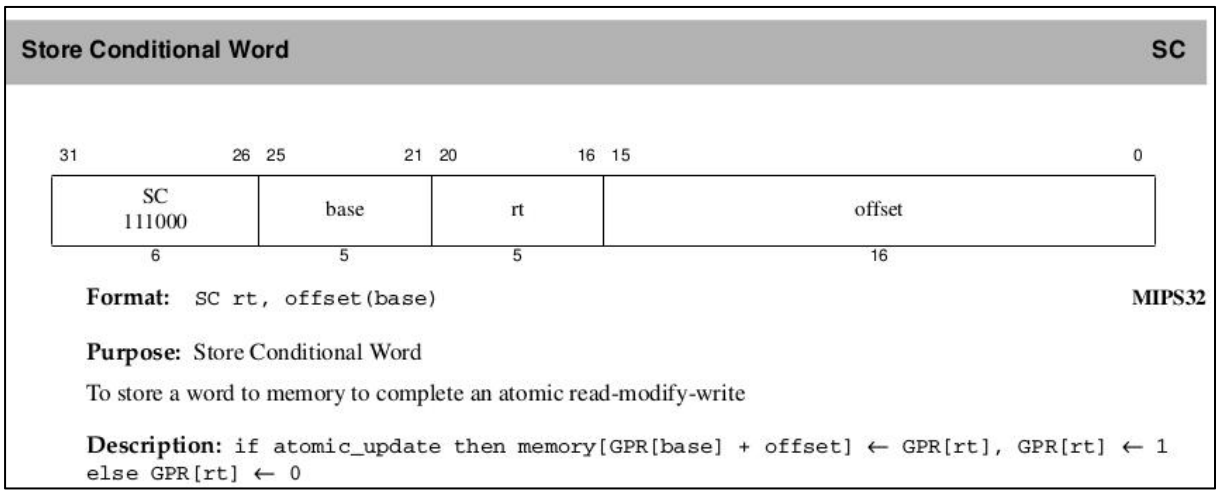


图 4-4 sc 指令格式

从上图对 ll 和 sc 描述（Description）就可以看出 ll/sc 实现原子操作的原理：指令 LL rt offset(base)从地址(base+offset)执行 32 位的加载，同时把加载的地址保存到 LLAddr 寄存器。当 SC rt,offset(base)执行时，首先会检查是否可以保证从上次执行的 LL 开始以来的读写操作能否原子性的完成，如果能,rt 值被成功存储同时 rt 被置 1。否则 rt 被置 0。所以上面实例中 sc 指令完成后，会有个对 v0 值的判断“beq v0,zero,atomic_inc”，如果 v0 值为 0 说明原子操作失败，需要返回 atomic_inc 函数入口重试。

第 5章 函数调用和寻址方式

每一种体系架构都对函数调用有自己的规则。比如参数如何传递、返回值如何保存、函数符号如何寻址等。前面的章节已经列举的 MIPS 的寄存器和指令集。本章将通过学习如何使用这些寄存器和相关指令实现一个函数和函数调用。

5.1 如何实现一个函数调用

先介绍什么是函数调用。为什么分成固定参数的函数调用和不定参数的函数调用？两者的定义分别是什么？有什么区别？

MIPS 中和函数调用相关的寄存器有 a0-a3（用于参数传递）、v0/v1（用于函数返回值）、ra（用于函数的返回地址）、函数调用使用寄存器 t9。下面通过两个实例来讲解固定参数的函数调用方法和不定参数的函数调用方法。

5.1.1 固定参数的函数调用

下面 func_test.c 文件里面通过一段内嵌汇编实现有 2 个参数的 test 函数的调用方法。

```
/*fuc_test.c*/
#include <stdio.h>
void test(const char * str，const int num){
   printf("%s %d \n",str,num);
}
int main()
{
   void* addr = test;
   char* str = "function test";
   int num = 5;
   int ret = 0;

   __asm__ __volatile__(
```

```
        "move    $25,%1\n\t"  //move t9 , addr
        "move    $4,%2\n\t"   // move a0,str
        "move    $5,%3\n\t"   //move a1,num
        "jalr      $25 \n\t"    //jalr t9 调用 test()函数
        "nop \n\t"
        "sw    $2,%0\n\t"     //sw v0,ret
        :"=m"(ret)
        :"r"(addr),"r"(str),"r"(num)
        :"$4","$5","$25","memory","$2","$31");

    printf("return value ret=%d\n",ret);
    return 0;
}
```

这个文件比较简单，里面的 void test(const* str)函数的参数是常量字符串 str，没有返回值。功能就是仅仅打印出字符串 str 的信息。int main()函数是程序入口函数。通常在 c 语言中我们要实现对 test 函数的调用，只需要一句“test(str);”即可实现。但是为了描述 MIPS 汇编语言的使用，这里我用了内嵌汇编来完成 test 函数的调用。

内嵌汇编实现了在 c 语言中使用汇编指令的功能，它有一套自己的语法规则，格式如下：

__asm__(

汇编语言部分

：输出部分

：输入部分

：破坏描述部分

);

关于内嵌汇编详细的讲解，可以参考第 8 章。这里我们主要关心和函数调用相关的汇编语言部分。相对应的，此处代码中，%0、%1、%2、%3 分别对应输出部分和输入部分的变量“ret”、“addr”、“str”和“num”。“move　$25,%0”实现了把 addr 指针(指向了 test 函数的地址)存入寄存器 t9。“move　$4,%1”实现了第一个参数的传递（str 放在 a0 上）。“move　$5,%2”实现了第 2 个参数的传递（num 放在 a1 上）。“jalr　$25 ”、“nop”实现了到对函数 test 的调用。“sw　$2,%0”实现了把 v0 存入到变量 ret。

这样我们就通过 a0 和 a1 完成函数参数的传递、t9 完成函数调用并通过 v0 获取到函数返回值。由于 MIPS 体系架构（n64 标准）规定 a0-a7 用于函数参数传递，所以当子函数的参数个数超过 8 个时，只能通过堆栈的方式实现。

5.2.2 不定参数的函数调用

不定参数就是函数的形式参数不是固定的，可以是 2 个，可以是 3 个等等。比如 c 语言里面常常用到的 printf。printf 的定义是“int printf (const char *__restrict __format, ...);”。那么这样的函数我们怎么调用呢？下面是通过汇编指令实现的不定参数 printf 函数的调用：

/* printf(“%s,%d \n”,str,num) */

```
int printf_test(){
   char* pFormat = "%s,%d \n";
   char* str = "function test";
int num = 5;
   int ret=0;

   __asm__ __volatile__(
      ".set noreorder \n\t"
      "move    $25,%1\n\t"
      "move    $4,%2\n\t"  //参数 1 ： “%s,%d \n”
      "move    $5,%3\n\t"  //参数 2: str
      "move    $6,%4\n\t"  //参数 3: num
      "jalr    $25 \n\t"       //调用 libc 库的 printf()函数
      "nop \n\t"
      "sw      $2,%0 \n\t"
      :"=m"(ret)
      :"r"(printf),"r"(pFormat),"r"(str),"r"(num)
      :"$4","$5","$25","memory","$31","$6");

   return 1;
}
```

在这个函数里，实现的就是调用函数 printf 打印数据（str 和 num）。但是 printf 是参数是不定的。所以我们调用它时需要将格式"%s,%d \n"也要当参数传递过去，参数存放顺序就是 a0 存放格式字符串"%s,%d \n"的地址，a1-a7 用来存放具体的参数，比如上面 a1 存放第一个参数字符串 str 的地址、a2 存放第二个参数 num 的值 5。这样就实现了不定参数的传递过程。

所以它的参数栈如图 5-1 所示。

寄存器	里面内容
a0	"%s,%d \n" 的地址
a1	str的地址
a2	num值5
a3	undefine
a4	undefine
a5	undefine
a6	undefine
a7	undefine

图 5-1 参数传递

5.2 MIPS 寻址方式

寻址方式就是处理器根据指令中给出的地址信息来寻找有效地址的方式，是确定本条指令的数据地址以及下一条要执行的指令地址的方法。寻址方式分为指令寻址方式和数据寻址方式两大类。

指令寻址方式就是确定 CPU 下一条要执行的指令地址的方法。指令寻址方式又被分为顺序寻址和跳跃寻址方式。顺序寻址方式就是程序执行时，一条指令接着一条指令地顺序执行，这个过程借助于程序计数器 PC 来完成，当前指令执行时，PC 通过 PC+8（此处为 64 位寻址）来指向下一条指令。跳跃寻址就是下一条指令的地址不由 PC+8 给出，而是由本条指令确定新的地址，然后修改 PC 值，完成指令的跳转。跳跃寻址方式又分为绝对寻址方式和相对寻址方式。绝对寻址就是在指令格式的地址的字段中直接指出操作数在内存的地址。由于操作数的地址直接给出而不需要经过某种变换，所以称这种寻址方式为直接寻址方式。MIPS 文档中描述为 not PC-relative，即 PC 无关寻址，mips 中的 JAL 指令就是这一类。相对寻址以程序计数器 PC 的当前值（指向当前指令所在位置）为基地址，指令中的地址标号作为偏移量，将两者相加后得到操作数的有效地址。MIPS 文档中描述为 PC-relative，即 PC 相关寻址，mips 中的 bal 指令就是这一类。

形成操作数的有效地址的方法称为操作数的寻址方式，又叫数据寻址方式，目的是如何完成寄存器和内存之间数据的正确加载和存储。MIPS 下数据寻址可以认为就有一种：基址+偏移的方式，类似 lw $1,offset($2)。

上面的寻址方式可以总结为如下图：

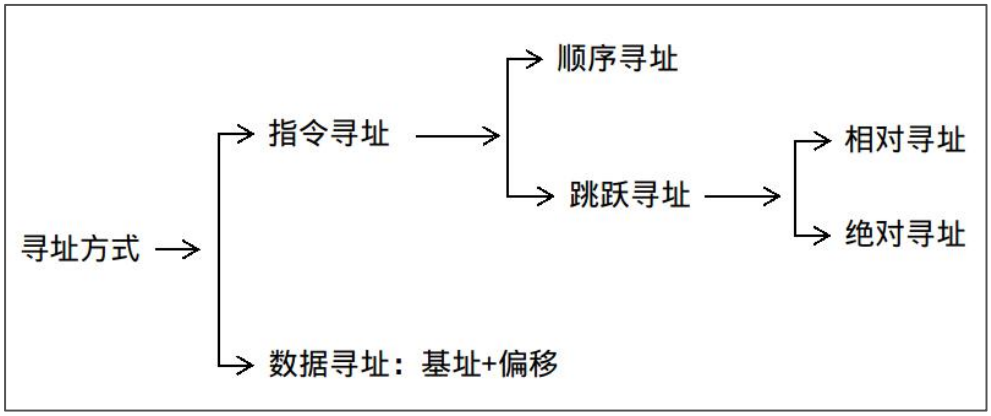


图 5-2 MIPS 寻址方式

下面将详细介绍 mips 中的相对寻址方式、绝对寻址方式和数据寻址方式。

5.2.1 MIPS 跳跃寻址方式和跳转指令

上面提到，跳跃寻址方式分为相对寻址方式和绝对寻址方式，对应 MIPS 的 I 型跳转指令（比如指令 B、BAL、BEQ、JALR 等）和 J 型跳转指令（比如 J、JAL 等）。在此以 bal 和 jal 指令讲解 mips 的指令寻址计算过程。

BAL 指令实现的相对寻址

bal 指令的格式如下图：

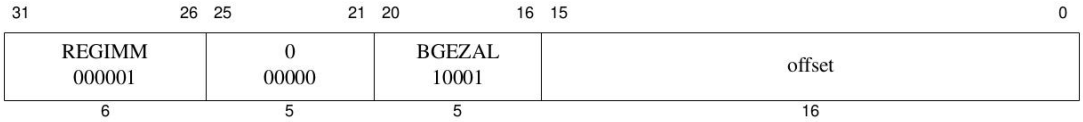


图 5-3 bal 指令格式

bal 指令用低 16 位（bit0-bit15）存储 18 位的有符号地址偏移（偏移地址右移 2 位后存储），所以 bal 可以实现的寻址范围在-128KB~+128KB。我们可以把 5.1 小节中的 test 函数调用指令“jalr $25”换成指令“bal test”,然后使用命令“gcc fun_test.c -o fun_test”来重新编译 fun_test.c 文件。通过“objdump -D fuc_test > func_test_objump”得到返汇编文件 func_test_objump，打开此文件后，会看到大致下列信息：

```
0000000120000b50 <test>:
  120000b50:67bdffd0      daddiu   sp,sp,-48
...
0000000120000cac <main>:
...
  120000d00:0411ff93      bal  120000b50 <test>
  120000d04:00000000      nop
```

我们开始使用”bal test”时并不知道 test 函数的地址， gcc 编译过程的链接器 collect2 会实现地址定位计算。计算后 bal 的二进制机器码为 0411ff93，这里的 0411 对应 bal 指令格式的 bit16-bit31 位，0xff93 对应低 16 位的 offset。bal 的计算过程如图：

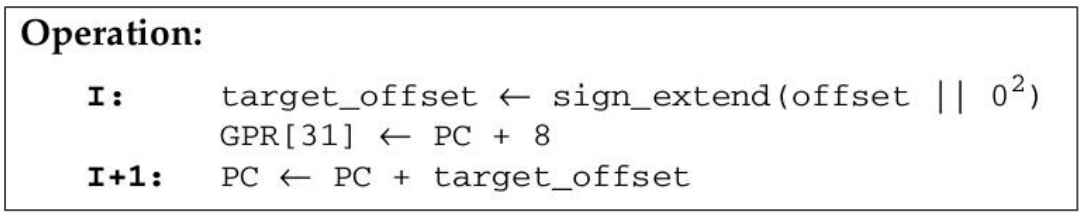


图 5-4 bal 寻址过程

按照上图过程，对 0xff93 左移 2 位并进行有符号扩展后得到 target_offset 结果为负数 0x1B4。当前 PC 的下一条指令地址为 0x120000d04，加上 target_offset 结果后就得到 test 的地址为 0x120000b50 。

JAL 指令实现的绝对寻址

jal 指令的格式如下：

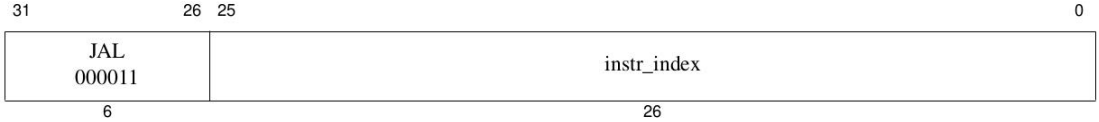


图 5-5 JAL 指令格式

jal 用低 26 位（bit0-bit25）来存放 28bits 的地址（存放时地址右移 2 位），可寻址范围在 256MB(2 的 28 次方)。它的计算方式如下图：

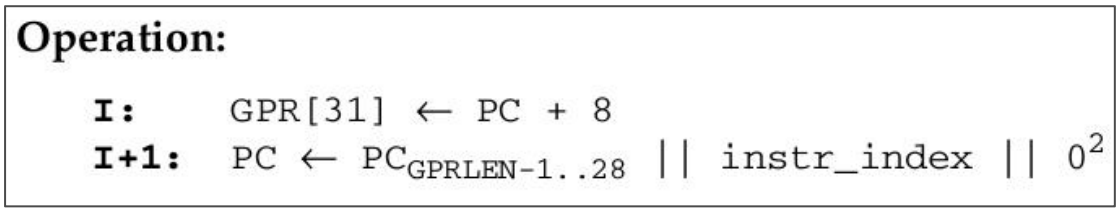


图 5-6 JAL 寻址过程

在 MIPS 64 位架构中，GPRLEN 值为 64。那么上图计算过程是先把 PC+8 的地址存入寄存器 ra（$31），然后把指令的低 26 位左移 2 位。剩下的高 36 位（bit63...bit28）地址使用当前 PC 的高 36 位值。

我们把程序中函数调用改成“jal test”后的反汇编结果大致如下：

```
...
  120000030:0c000118      jal    120000460 <test>
...
0000000120000460 <test>:
  120000460: 67bdffe0       daddiu    sp,sp,-32
```

按照 JAL 寻址过程，pc 的高 36 位（bit63-bit28）不变，低 28 位的计算为：指令码 67bdffe0 的低 26 位（0x118）左移 2 位后值为 0x000460，结果为 0x120000460 即为要跳转 test 函数的地址。这里 GPRLEN 意为通用寄存器的长度，在 64 位系统下此值为 64。

从上可以总结出 bal 和 jal 的寻址差别如下表：

指令	指令格式	寻址范围	寻址方式	重定位类型
bal	I 格式（6/5/5/16）	128KB（2 的 16 次方 ×4）	相对寻址	R_MIPS_PC16
jal	J 格式（6/26）	256MB(2 的 26 次方×4)	绝对寻址	R_MIPS_26

表 5-1 bal 和 jal 寻址差别

5.2.2 数据寻址

MIPS 体系架构中对于 load/store 等指令，如 ld、lw、sd、sw 等存取操作的数据寻址方式使用的是基址+偏移的方式。以 lw 为例，它的格式如下图：

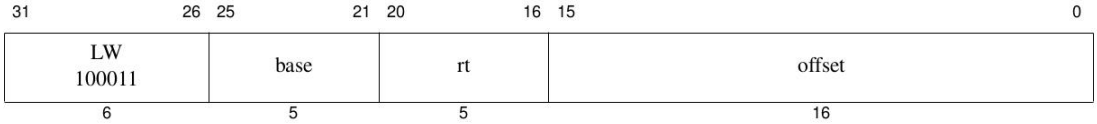


图 5-7 LW 指令格式

lw 形式为 lw rt,offset(base)。下面的汇编语句实现使用 lw 指令加载一个整数 args（初始值为 2）到 a0（$4）寄存器，并作为 test 函数的参数：

```
lw   $4, args
jal  test
args:
.long 2
```

经过 gcc 编译后的这段指令对应的反汇编结果如下：

```
120000000:3c040000     lui   a0,0x0
120000004:3c012000     lui   at,0x2000
120000008:64840001     daddiu   a0,a0,1
12000000c:0004203c     dsll32   a0,a0,0x0
120000010:0081202d     daddu    a0,a0,at
120000014:8c840038     lw    a0,56(a0)
120000030:0c000118     jal   120000460 <test>
120000034:03e00008     jr    ra

0000000120000038 <args>:
120000038:00000002     srl   zero,zero,0x0  // args 是位于.data 段的符号，这里 srl 无效
12000003c:00000000     nop                  // nop 无效
```

其中前面的 5 条指令“lui a0,0x0”、“lui at,0x2000”、“daddiu a0,a0,1”、“dsll32 a0,a0,0x0”和“daddu a0,a0,at”共同完成了基址 a0 的计算，结果即为 0x120000000。感兴趣的同学可以对照 mips 官方手册计算一下，这里不再展开介绍。
接下来的”lw a0,56(a0)”，56 是 0x38 的十进制形式。所以 lw 的结果就是加载 0x120000038（0x120000000+0x38）地址的数据到 a0。

在这里要注意在 0x120000038 处的 srl 和 nop 都是无效指令，这里实际上存储的数据 2。objdump 工具在反汇编时没有区分数据和指令。

第 6 章 一个最“小”程序

绝大多数情况下，我们编写的 c 语言程序都是依赖很多系统库才能运行的，例如本书前面章节介绍的 hello.c，该程序仅通过调用 printf 函数实现“hello world”的打印，其中必不可少的就有

crt1.o、 crti.o、crtn.o、crtbegin.o、crtend.o 和 libc 库。其中，libc 库包含了数学函数、字符串处理、I/O、网络、线程等最基本和最常用的函数。crt1.o、 crti.o、crtn.o、crtbegin.o、crtend.o 则提供了代码的运行入口与全局构造和解析函数。如果我们的程序使用了动态链接，那么还得需要 ld 库帮助我们在程序运行时实现其他动态库的加载。所有这些大部分时间对程序员来说是透明的。本章我们将编写一个最“小”程序，同样实现“hello world”的打印，但是跳过对那些.o 文件和 libc 库的依赖。这里的“小”是指这个程序最终文件会很小。之前依赖系统库实现的“hello world”打印功能的程序文件大小近 22KB，而本章实例完成同样的功能的程序大小将会不到 1KB。

6.1 逃离 libc 库的程序

libc 库很重要的一个功能就是实现了内核很多功能的封装。例如，libc 库中的 printf 函数内部就是通过系统调用实现数据的打印输出（关于系统调用的解释请看 6.4 节）。所以要实现不通过 libc 库就可以完成“hello world”的输出打印，那么就不能使用 libc 库提供的 printf 函数，而是直接使用汇编语言做系统调用。接下来详细描述实现过程。

首先我们要编写的是 main.c 文件，里面内容如下：

```
//main.c
#define STR "Hello world \n"

void printf(char* str,int len){
   asm(
".set noreorder \n\t"
"li    $2,5001\n\t"        //系统调用函数 write() 的 ID 号是 5001 ,放在 v0
"li    $4,0\n\t"           //参数 1：stdout 文件描述符是  1
"move    $5,%0\n\t"      // 参数 2：
"move    $6,%1\n\t"    // 参数 2：
"syscall     \n\t"      // 系统调用指令
:
:"r"(str),"r"(len)
:"$2","$4","$5","$6");
}

void exit(){
   asm(
      "li    $2,5058\n\t"  // syscall id of exit is 5058 ,load to  v0
      "li    $4,0\n\t"     //exit code is 0,load to a0
      "syscall     \n\t");
}

int main(){
   printf(STR,13);
```

```
    exit();
}
```

注意：建议提前阅读 6.4 系统调用小节。这里要清楚知道 mips 中系统调用通过 syscall 指令实现，系统调用号通过 v0 寄存器（$2）传递，其他参数通过 a0-a7 ($4-$11)传递。

在 main.c 里定义了 2 个函数 printf 和 exit，用于打印信息和程序正常退出。自定义的 printf 函数通过对系统调用 write 实现了字符串 STR 的输出。write 系统调用的定义在内核代码 fs/read_write.c 中，声明如下：

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf, size_t, count){
return ksys_write(fd, buf, count);
}
```

可以看出 write 是 3 个参数的系统调用。

所以上面代码先是不把 write 系统调用号 5001 存入 v0（$2），然后参数文件描述符 fd 的值 1 写入寄存器 a0（$4），字符串 str 写入寄存器 a1（$5）、字符串 str 长度写入寄存器 a2（$6）。fd 值为 1 意味要写入的文件是/dev/stdout，它的描述符可通过下面命令得出：

```
# ls -l /dev/stdout
lrwxrwxrwx 1 root root 15 2 月  19 11:38 /dev/stdout -> /proc/self/fd/1
```

然后 syscall 指令使程序陷入内核态，并完成功能调用。

exit()函数系统调用方式相同，就是参数要传递的退出状态码是 0，表示成功退出。

6.2 链接脚本

本书第 2 章提到了 gcc 编译的过程，其中阶段四链接过程实现了把多个目标文件链接成一个可执行文件。其过程中需要一个链接脚本来帮助指定链接规则，例如规定如何把输入文件内的 section 放入输出文件内, 并控制输出文件内各部分在程序地址空间内的布局等。这里的输入文件就包括了 crt1.o、crtbegin.o libc.so 等目标文件。输出文件通常就是可执行文件或者动态库文件。在这里，我们要实现独立的最小程序时，为了避免这个默认链接脚本对一些输入文件的依赖，所以需要重写一个链接脚本。具体内容如下：

```
//ld.lds 文件
OUTPUT_ARCH(mips)
ENTRY(main)
SECTIONS
{
. = 0x120000000 + SIZEOF_HEAEDERS;
```

```
.text : {
        *(.head.text)
                *(.text*)
}

.rodata : {
        *(.rodata*)
        *(.got*)
}

.data : {
        *(.data*)
        *(.bss*)
        *(.sbss*)
}

/DISCARD/ : {
*(.comment)
*(.pdr) /*debug used*/
*(.options)
*(.gnu.attributes)
*(.debug*)
*(.MIPS.options)
}
}
```

链接脚本的语法并不复杂。可以通过内容逐步介绍。
第一行的 OUTPUT_ARCH(mips)指定了体系架构为 mips。ENTRY(main)指定程序入口函数为 main()。SECTIONS{ …}为链接脚本主体，里面包含了 SECTIONS 的变换规则。

- .=0x120000000 + SIZEOF_HEAEDERS; 这是一条赋值语句，意思是将当前虚拟地址设置成 0x120000000 + SIZEOF_HEAEDERS。”.”代表了当前虚拟地址。 SIZEOF_HEAEDERS 为输出文件的文件头大小。链接脚本里面的语句分为赋值语句和命令语句，上面的 OUTPUT—ARCH 和 ENTRY 就属于命令语句，可以用换行代替“;”赋值语句必须使用“;”结尾。
- .text:{*(head.text) *(.text*)} 这是个段转换规则。意为将所有输入文件中名字为“.head.txt”和“.text.*”的段依次合并到输入文件的.text 段。段是 ELF 文件格式中的一个概念。可以查看本书最后一章对 ELF 格式介绍。
- /DISCARD/ : { *(.comment) *(.pdr)…*(.MIPS.options) } 意为将所有输入文件中的.comment 段、.pdr 段、.MIPS.options 段丢弃，不保存在输出文件中。
- /*debug used*/ 链接文件中需要使用/**/作为注释。

6.3 最“小”程序的运行

上面实现了 main.c 和 ld.lds 链接脚本，那么我们的最“小”程序就可以使用下面命令编译运行：

```
$ gcc -c -fno-builtin -mno-abicalls main.c
$ ld -T ld.lds main.o -o main
$ ./main
Hello world
```

- -c 表示“编译、汇编到目标代码，不进行链接”。
- -fno-builtin 表示“关闭 gcc 内置函数功能”。
- -mno-abicalls 表示“do not generate SVR4-style position-independent code. -mabicalls is the default for SVR4-based systems.”。
- -T ld.lds 表示使用链接脚本 ld.lds。如果不指定-T 项，那么 ld 会使用系统默认的链接脚本。
- -o main 表示输出可执行文件名为 main。

这个程序没有使用任何其他系统库，通过系统调用就完成了“hello world”的输出。我们可以查看一下这个 main 程序的大小：

```
$ ls -lh main
-rwxrwxr-x 1  985 2 月  19 16:24 main
```

可以看到 main 大小仅为 985 的字节，这的确算得上是最“小”程序。

6.4 系统调用

从用户程序的角度看，内核是一个透明的系统层，因为用户程序都是通过 libc 库运行。内核是一个操作系统的核心，负责管理系统的进程、内存、设备驱动程序、文件和网络系统等，同时她也是计算机硬件的第一层软件扩充，对上提供操作系统的应用编程接口（Application Programming Interface，API）。这些 API 就叫系统调用。通常 libc 库封装了这些系统调用的接口，被看作是用户程序和内核的中间层。例如 printf 的调用过程如下图所示：

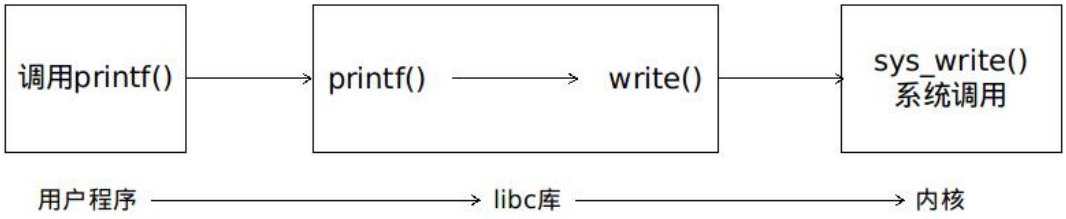


图 6-1 printf()的调用过程

从上图可以看出，内核里才是真正实现数据的输出显示。而 libc 库就是对内核提供此功能接口的封装。有了这层封装，用户程序就不用去关心过多的系统底层细节，并确保用户程序具有更好的兼容性和移植性。

6.4.1 系统调用号

在 Linux 中，每个系统调用被赋予一个系统调用号，它就像是人的身份证号码，用于唯一的标识一个系统调用接口。当用户空间的程序执行一个系统调用时，就会用到这个系统调用号还指明到底是要执行哪个系统调用。

内核记录了所有已经注册过的系统调用的列表。每一种体系结构中都明确定义了这个表。在 mips64 中,它的定义在内核代码的 arch/mips/kernel/syscalls/目录下，下面有 syscall_n64.tbl、syscall_n32.tbl、syscall_o32.tbl 等。在龙芯 3000 系列的处理器上使用是 syscall_n64.tbl，里面部分 ID 号定义如下：

```
0     n64 read                sys_read
1     n64 write               sys_write
2     n64 open                sys_open
3     n64 close               sys_close
4     n64 stat                sys_newstat
5     n64 fstat               sys_newfstat
6     n64 lstat               sys_newlstat
```

其中第一列就是系统调用 ID 号。比如第一行的 ID 号 0 分配给了系统调用函数 sys_read。

6.4.2 系统调用方法

系统调用运行于内核态，必须通过软中断机制才能进入系统核心，然后才能转向相应的命令处理程序。内核态是操作系统内核所运行的模式，运行在该模式的代码，可以无限制地对系统存储、外部设备进行访问。和内核态相对的就是用户态，用户态只能受限的访问系统资源。用户态到内核态的切换只能通过系统调用。不同的体系结构有不同的指令来实现这个软中断，mips 下这个指令是 syscall。调用规则和本书前面将的函数调用很像，大致如下：

- 系统调用号放在 v0 中。
- 参数放在 a0-a7 寄存器中传递。
- 系统调用的返回值位于 v0 中。

例如我们通过系统调用来实现 libc 库中的 open 函数功能，具体例子如下：

```
int open(const char* fileName,unsigned long flags, unsigned long mode){
   int fd = 0;

   asm(
"li    $2,5002\n\t"    //sys_open syscall id is 5002 -> v0
"move    $4,%1\n\t"      //filename          -> a0
"move    $5,%2\n\t"      // flags          -> a1
"move    $6,%3\n\t"      // mode          -> a2
"syscall     \n\t"
"sw    $2,%0\n\t"
```

```
:"=m"(fd)                    //输出 对应于%0
:"r"(fileName),"r"(flags),"r"(mode)  //输入 对应于%1 %2 %3
:"$2","$4","$5","$6");
   return fd;
}
```

其中参数 fileName 表示函数名称，可以传入类似于“/tmp/file.txt”等字符串。flags 表示访问模式（是否可读、可写、、可创建、可执行等），例如可以传入 O_RDWR | O_CREAT 标识文件打开方式为可读可写且文件不存在时创建。mode 表示文件访问权限，例如可以设置成 0644 表示用户具有读写权限而其他用户具有只读权限。

open 系统调用定义在内核代码 fs/open.c 里面：

```
SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
```

实际上所有的系统调用声明都类似 SYSCALL_DEFINEn 开始，其中 n 代表参数个数，这里 n 为 3 就说明 open 具有 3 个参数 filename、flags 和 mode。里面的 open 就是接口名称。

open 对应的系统调用号通过 syscall_n64.tbl 查得为 5002,赋值给寄存器 v0（$2）。三个参数 fileName、flags、mode 分别赋值给 a0（$4）、a1（$5）、 a2（$6），然后通过指令 syscall 实现系统调用，因为返回值被存在 v0 中，所以通过“sw $2，%0”实现 v0（$2）写入变量 fd。

6.5 汇编程序源文件.S

上面的最“小”程序是通过 c 文件里面使用内嵌汇编实现的。实际上我们还可以直接编写汇编程序源文件来实现。这个文件通常以.S 做后缀，具体实现如下：

```
#define __ALIGN         .align 4

#define ENTRY(name)        \
.globl name;         \
__ALIGN;       \
.type name, @function;      \
name:

#define END(sym)       \
.size sym, . - sym

.section .head.text, "ax"
ENTRY(main)
.set noreorder
```

```
    li $2,5001
    li $4,2
    dla $5,str
    lw $6,length
    syscall
    nop
    break

    length:
    .word 12
    str:
    .string "Hello World\n"
    END(main)
```

- .global name
- dla $5,str dla 为加载 64 位地址指令，这里是加载 str 地址到寄存器 a1($5)。

运行方式如下：

```
$ gcc -c -fno-builtin -mno-abicalls test.S
$ ld -T ld.lds test.o -o main
$ ./main
Hello World
```

第 7 章 编写 MIPS 汇编程序

在本书的第 3 章和第 4 章分别系统地介绍了 MIPS 寄存器和指令。有了这些，我们编写汇编程序还需要一个“框架”,用“框架”的目的是让我们更加容易的编写汇编程序，把更复杂的任务，比如符号解析、地址重定向、对齐等工作交给工具链去完成。这个框架有 2 种方式，汇编源代码文件(以.S 为后缀)和内嵌汇编（在.c 文件中嵌入汇编语言的方式）。内嵌汇编前面也接触了一些，后面还将有专门一节来介绍。本章介绍第一种“框架”方式。

我们可能经常遇到以.S 后缀结尾和.s 后缀结尾的文件。它们都是汇编源文件（可以作为 gcc as 汇编器的输入）。区别在于.S 是 GCC 编译的汇编源代码文件。编译后生成的输出文件就是.s。在第二章介绍 GCC 编译过程时我们知道一个.c 文件的编译过程。那么如果我们编写了.S 文件，gcc 就就省略了.c 文件到.i 文件的过程，可以直接对.S 进行编译产生对应的.s 文件。流程就如下图所示：

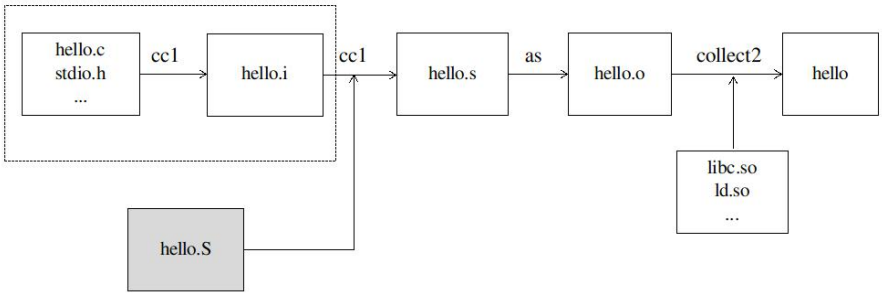


图 7-1 汇编源文件的编译过程

本章将通过一个实现内存数据拷贝功能的 memcpy.S 文件来了解 MIPS 汇编程序的编写规则。

7.1 汇编程序实例：memcpy.S

我们平时编程时经常会使用到 libc 库中的内存拷贝功能，接口如下：

void *memcpy(void *dest, const void *src, size_t n);

函数的功能是从源内存地址 src 的起始位置开始拷贝 n 个字节到目标内存地址 dest 中。这里编写 MIPS 汇编程序实现如下：

```
//memcpy.S
#include <sys/asm.h>
#include <sys/regdef.h>
.section .text
LEAF(my_memcpy)
.set push
    .set mips64
dadd v0,zero,a0
daddiu t1,zero,0
loop:
beq t1,a2,exit
nop
lb t2,0(a1)
sb t2,0(a0)
daddiu t1,t1,1
daddiu a0,a0,1
daddiu a1,a1,1
j loop
nop
exit:
jr ra
nop
    .set pop
END(my_memcpy)
```

上面的第一行引用的头文件 asm.h 里定义了宏 LEAF 和 END，分别代表了一个函数的开始和结束。具体定义如下：

```
#define  LEAF(symbol)                    \
        .globl      symbol;                 \
        .align      2;                      \
        .type       symbol,@function;       \
        .ent symbol,0;                  \
symbol:         .frame    sp,0,ra;

# define END(function)              \
        .end function;                  \
        .size function,.-function
```

以“.”开头的都是仅仅给汇编器看的汇编指令，具体解释参考 5.2 小节。这里解释一下叶子函数（LEAF）：不再调用其他子函数的函数。

上面第二行引用的头文件 regdef.h 里面定义了 t2、a0 等寄存器的习惯名称。

程序中的“loop :”代表的是标号 。标号都以“:”结尾，标号用于定义函数的入口点、中间的分支、数据存储位置。这里标号 loop 和下面的标号 exit 分别代表分支。

程序中的指令“beq t1,a2,exit”。其中 a2 就是参数 n，表示要拷贝的字节数。t1 功能是计数器，初始值为 0（daddiu t1,zero,0）。如果 t1 和 a2 相等，那么指令跳转到 exit 标识处执行返回操作。否则就要拷贝 src（a1）中的一个字节到 dest（a0）中，然后地址移位。就是指令：

```
lb t2,0(a1)
sb t2,0(a0)
daddiu a0,a0,1
daddiu a1,a1,1
```

复制一个字节后,通过”j loop”跳转到 loop 标识处重新判断、拷贝字节。随着 t1 在后面的不断累加(daddiu t1,t1,1)到 a2 大小，指令就会执行到 exit。

这个 memcpy.S 写好后，我们可以通过编写一段 c 代码调用测试它。c 代码如下：

```
//main.c
#include <stdio.h>
int main(){
   char* str = "function test \n";
   char dest[100];
   my_memcpy(dest,str,13);
   printf("%s \n",dest);
      return 0;
}
```

编译运行命令如下：

```
# gcc main.c memcpy.S -o out
```

```
# ./out
function test
```

7.2 汇编指令

汇编指令（Assembler Directives），是汇编语言中使用的一些操作符和助记符,还包括一些宏指令(如 nop、dla、li 等)。用于告诉汇编程序如何进行汇编，它既不控制机器的操作也不被汇编成机器代码，只能为汇编程序所识别并指导汇编如何进行。

所有汇编程序指令的名称都以句点（'.'）开头，以“;”或者换行结尾。这些名称对大多数目标都不区分大小写，通常用小写字母书写。一些常用的的指令分类和功能如下：

符号/数据定义相关的指令：

- .global symbol 声明 symbol 为全局变量。
- .lobal symbol 声明 symbol 为局部变量。作用范围仅在当前文件内。
- .extern symbol 声明一个外部符号。
- .set symbol ,expression 常量设置。比如
  “.set mark,0x3”设定常数，类似 c 语言中的宏定义。
- .byte 定义一个字节（8 位）的地址空间。类似的指令还有.int、.long 、.word。空间大小依赖具体系统。
- .string “STR”也是用于字符串定义。但是有尾端区分。
- .ascii “string”... 将字符串组合成连续的地址。还有一个“.asciz”。.asciz 会在字符串后自动添加结束符\0。
- .rept count 重复 count 次扩展后面的指令，以.endr 结尾。例如：
  ```
  .rept 3
  .int 0
  .endr
  ```
  这就相当于目标文件中会分配 12 个 Byte 的空间（int 大小为 4Byte*3 次）。
- .macro name args 宏定义，name 为宏名称，args 为参数，以.endm 结尾。例如：
  ```
  .macro label l
      \l:
  .endm
  ```

汇编控制相关的指令：

- .set symbol 汇编语言控制指令。例如：
  “.set push”和“.set pop”分别用于设置的保存和恢复，表明其间的.set mips64 设 置仅仅对当前这段代码起效。
  “.set noat” 防止汇编器将汇编代码翻译成用到 at（$1） 寄存器的指令序列。

“.set nomacro” 防止汇编器将单个汇编语句翻译成多个指令。

“.set norecorder” 防止汇编器打乱代码次序。通常和“.set reorder”成对出现。

- .text告诉汇编器，把此后产生的代码放到目标文件中的“.text”段。
- .section name 告诉汇编器，把此后产生的代码放到目标文件中的 name 段。比如：
  .section .text 表明接下来的这段代码放到目标文件的.text 段（代码段）。
  .section .data 表明接下来的这段代码放到目标文件的.data 段（数据段）。
  关于段的概念请参考 ELF 文件格式说明，在本书最后一张将有介绍。
- .ent 标识函数的起始点。
- .end 标识函数的结尾，和.ent 一样仅仅用于调试。
- .size 表示在函数表中，function 和所用指令的字节数一同列出。
- .align n 对指令或者数据的存放地址指定对齐方式。n 取值因系统而异。
- .type symbol @function 标识 symbol 为函数名称。

7.3 MIPS 相关的汇编指令

上面 7.2 列举的无论 GNU 汇编程序的目标机器配置如何都可用的指令。一些机器配置提供了附加指令。

- .sym32 设置加载地址为 32 位

  比如以 MIPS 的“dla $4,sym”宏指令为例。这句话的意思是加载 sym 地址到寄存器$4。在用 n64 汇编器汇编出来的结果是：

```
lui    $4,%highest(sym)
lui    $1,%hi(sym)
daddiu  $4,$4,%higher(sym)
daddiu  $1,$1,%lo(sym)
dsll32  $4,$4,0
daddu   $4,$4,$1
```

  也就是把 sym 地址按 64 位来处理，%highest(sym)获取的是 sym 的高 16 位（bit63-bit48）、%higher(sym)获取的是 sym 的 bit47-bit32。%hi(sym)获取的是 sym 的 bit31-bit16、%lo(sym)获取的是 sym 的低 16 位（bit15-bit0）。而如果要想指定 sym32 位处理，那么需要添加”.set sym32”指令，结果就是：

```
lui    $4,%hi(sym)
daddiu  $4,$4,%lo(sym)
```

- .set mipsn 兼容的指令版本

  n 是一个从 0 到 5 的数字，或是数字 32 或 64。1 到 5，32 或 64 使汇编器从源程序中的这一点开始接受相应 ISA 级别的指令。比如.set mips3 告诉汇编器下面的指令是 MIPS IV（64 位指令集，兼容 32 位指令）中的指令。
- .cpsetup .cpload
- .set gp=64 和.set fp=64 允许指定目标文件中寄存器的大小。默认情况是.set gp=default .set fp=default。
- .set hardfloat 开启使用硬件浮点指令。与其对应的是.set softfloat

-

更多的汇编指令可以参考 GNU 汇编器开源社区 https://sourceware.org/binutils/docs/as/的第 7 部分 Assembler Directives。

第 8 章 内嵌汇编

之前提到过我们编写汇编程序还需要一个“框架”。这个框架有 2 种方式，汇编源文件和内嵌汇编。汇编源文件.S 框架格式的汇编程序已经介绍过了，本章介绍内嵌汇编的语法规则。

内嵌汇编（Assembly）是可以直接插入在 c/c++语言中汇编程序。它实现了汇编语言和高级语言的混合编程。当在高级语言中要实现一些高级语言没有的功能，或者提高程序局部代码的执行效率时，都可以考虑内嵌汇编的方式。

内嵌汇编标识为 asm()。asm 是 c/c++中的内嵌汇编关键字，或称模板。用于通知编译器，接下来的()内的代码是内嵌汇编程序，需要特殊处理。()内部的有自己专门的语法格式。内嵌汇编实现了 c/c++语言和汇编语言的混合编程。比如你可以在一个 c 语言文件中你可以使用 MIPS 汇编指令 MOVE 完成两个数/地址的拷贝：

```
asm("move %0,%1\n\t" :"=r"(ret) :"r"(src) );
```

这里 src 和 ret 都是 c 语言中的变量。src 在内嵌汇编中作为输入操作数。ret 在内嵌汇编中作为输出操作数。“=g”中的“=”符号表明这是个输出操作数。%0,%1 称为占位符（顾名思义，占位符就是先占住一个固定的位置，等着你再往里面添加内容的符号），代表指令的操作数，分别代表 c 语言变量的 ret 和 src。 内嵌汇编指令"move %0,%1\n\t"中的 move 还不是真正的 MIPS 汇编指令，MIPS 中的 move 指令的 2 个操作数是寄存器，而此处 move 的操作数是 c 语言中的变量。C 变量与寄存器的对应关系由 GCC 编译器自动处理，处理后的结果就是 2 条 load 指令加载变量到某 2 个寄存器，然后再执行 move 指令操作。扩展后的真实汇编指令大概是下面这样：

```
lw    t1,src
lw    t2,ret
move t2,t1
```

可以看出使用内嵌汇编，我们就省去了加载变量到寄存器的过程，也不用考虑使用哪个寄存器的问题。方便我们更快捷的编写程序。

接下来我们就详细介绍内嵌汇编的格式。

8.1 内嵌汇编基本格式

前面章节介绍过内嵌汇编的基本格式：

```
asm(
内嵌汇编指令
: 输出操作数
: 输入操作数
: 破坏描述
);
```

可以看出，内嵌汇编以 asm();格式表示，里面分成 4 个部分，内嵌汇编指令、输出操作数、输入操作数、破坏描述。各部分之间使用“:”分割。其中内嵌汇编指令是必不可少的，但可以为空。其他 3 部分根据程序需要可选。如果只有内嵌汇编指令时，后面的“:”可以省略。例如：

```
asm("break" );
```

asm 是__asm__的别名，所以上面语句也可以写成

```
__asm__(“break”);
```

注意：如果使用了后面部分，前面部分为空，也需要“:”分割符。例如程序中没有输出部分，但是有输入部分，那么输出部分的“:”不能省略。同时 asm 模板里面可以使用/**/或者//添加注释。

再看下面的内嵌汇编：

```
asm("daddu %0,%1,%2\n\t" :"=r"(ret) :"r"(a),"r"(b));
```

其中"daddu %0,%1,%2\n\t" 就是内嵌汇编指令，指令由指令操作符和指令操作数组成。操作符就使用 MIPS 汇编指令中的助记符，操作数可以是%0,%1,%2 形式的占位符，来表示 c 语言中的表达式或变量，在这里分别代表操作数 ret、a 和 b。指令操作数也可以是寄存器。使用寄存器做指令操作数时，寄存器前面需要符号$。例如：

```
asm("move $31,%0\n\t" : :"r"(a));
```

上面这条指令实现了把 c 语言变量 a 的值存入通用寄存器 ra（$31）。

asm 模板里可以有多条内嵌汇编指令。每条指令都以" "为单位。多条指令可以使用" ;"号、\n\t 或者换行来分割。

```
asm("dadd %0,%1\n\t"
  "dadd %0,%2\n\t"
```

```
        :"=r"(ret)
         :"r"(a),"r"(b));
```

8.1.1 输入操作数和输出操作数

操作数包括输出操作数和输入操作数，输出操作数和输入操作数里的每一个操作数都由一个约束字符串和一个带括号的 c 语言表达式或变量组成，比如“r”(src)。多个操作数之间使用“,”分割。内嵌汇编指令中使用%num 的形式依次表示每一个操作数，num 从 0 开始。比如：

```
asm("daddu %0,%1,%2\n\t"
   :"=r"(ret)              //输出操作数，也是第 0 个操作数
   :"r"(a),"r"(b)    /*输入操作数，也是第 1 个操作数和第 2 个操作数*/
);
```

这里使用了 daddu 指令实现了 c 语言中 ret=a+b 的操作。两个输入操作数"r"(a)和"r"(b)之间使用“,”分割。%0 代表操作数"=r"(ret)、%1 代表操作数"r"(a)、%2 代表操作数"r"(b)。

每个操作数前面的约束字符串表示对后面 c 语言表达式或变量的限制条件。GCC 会根据这个约束条件来决定处理方式。比如"=r"(ret)中的"=g"表示有两个约束条件，"="表明此操作数是输出操作数，"r"(b)中的"r"表示将变量 b 放入通用寄存器（relation 相关联）。约束字符还有很多，有些还与特定体系结构相关，在下一节会详细列举。

输入操作数通常是 c 语言的变量，但是也可以是 c 语言表达式。比如：

```
asm("move %0,%1\n\t"
   :"=r"(ret)
   :"r"(&src+4));
```

这里输入操作数&src+4 就是 c 语言表达式。执行的结果就是把&src+4 的地址赋给 ret。

输出操作数必须是左值，GCC 编译器会对此做检查。简单地说，以赋值符号 = 为界，= 左边的就是左值，= 右边就是右值。输入操作数可以是左值也可以是右值。所以输出操作数必须使用"="标识自己。同时默认情况下输出操作数必须是只写（write-only）的，但是 GCC 不会对此做检查。这个特性有时会给我们带来麻烦。如果你要在内嵌汇编指令里把输出操作数当右值来操作，GCC 编译时不会报错，但是程序运行后你可能无法得到你想要的结果。为此我们可以使用限制符"+"来把输出操作符的权限改为可读可写。例如：

```
asm("daddu %0,%0,%1\n\t"
        :"+r"(ret)
   :"r"(a));
```

这就实现了 ret = ret+a 的操作。 "+r"中的"+"就表示 ret 为可读可写。同时我们也可以使用数字限制符"0"达到修改输出操作符权限的目的。

```
    asm("daddu %0,%1,%2\n\t"
:"=r"(ret)
:"0"(ret),"r"(a));
```

这里数字限制符"0"意思是第 1 个输入操作数 ret 和第 0 个输出操作数使用同样的地址空间。数字限制符只能用在输入操作数部分，而且必须指向某个输出操作数。

8.1.2 破坏描述

破坏描述部分就是声明内嵌汇编中有些寄存器被改变。通常内嵌汇编程序中会使用到一些寄存器，并对其做修改。如果在破坏描述部分不做说明，那么 gcc 编译内嵌汇编时不会做任何的检查和保护。这可能就会导致程序出错或致命异常。例如：

```
    asm(
            "dadd %0,%1,%2\n\t"
        "move $31,%0\n\t"
        :"=g"(ret)
        :"r"(a),"r"(b)
);
```

上面程序完成 ret=a+b，然后 ret 的值写入寄存器 ra($31)。我们知道寄存器 ra 被用来做函数返回的。但是 ra 被改变，将导致函数无法正常返回。这时就需要在破坏描述部分添加声明来告诉编译器此寄存器的值被改变。MIPS 的内嵌汇编中寄存器的使用以$num 形式，num 代表寄存器编号。在破坏部分声明就使用"$num"形式，多个声明之间使用“,”分开。例如：

```
 asm(
            "dadd %0,%1,%2\n\t"
        "move $31,%0\n\t"
        :"=g"(ret)
        :"r"(a),"r"(b)
            :"$31"
);
```

破坏描述符除了寄存器还有“memory”。它的作用见本章最后一节。

8.1.3 有名操作数和指定寄存器

从 gcc 的 3.1 版本之后，内嵌汇编支持有名操作数。就是可以在内嵌汇编中为输入操作数、输出操作数取名字，名字形式是[name]，放在每个操作数的前面，然后汇编程序模板里面就可以使用%[name]的形式，而不是上面%num 形式。例如：

```
asm("daddu %[out],%[in1],%[in2]\n\t"
    :[out]"=g"(ret)
    :[in1]"r"(a),[in2]"r"(b)
);
```

其中 name 可以是大小写字母、数字、下划线等。但是你必须确保同一汇编程序中没有两个操作数使用相同的符号名。

当然，你可以仅输出或者输入中的部分操作数去名字。例如：

```
asm("daddu %[out],%1,%2\n\t"
    :[out]"=g"(ret)
    :"r"(a),"r"(b)
);
```

这里我只给第 0 个操作数"=g"(ret)取名字。那么后面的第 1 个操作数和第 2 个操作数在使用时还是序列号形式%1、%2。

有时候我们需要在指令中使用指定的寄存器；比如系统调用时需要将系统调用号放在 v0 寄存器，参数放入 a0-a7 寄存器。那么这是我们可以在 c 语言声明变量时使用指定寄存器功能。例如：

```
register int sys_id asm("$2") = 5001;
```

register 声明了一个寄存器变量 sys_id。通知 GCC 编译器使用$2（v0）寄存器来加载 sys_id 变量。

8.2 约束字符

约束字符就是输入操作数和输出操作数前面的修饰符。约束字符可以说明操作数是否可以在寄存器中，以及哪种寄存器；操作数是否可以是内存引用，以及哪种地址；操作数是否可以是立即常数，以及它可能具有的值。本节介绍常用的约束字符信息。

- “r” 通知汇编器可以使用通用寄存器中的任意一个来加载操作数。最常用的一个约束。
- “g” 允许使用任何通用寄存器、内存或立即整数操作数。
- “i”通知汇编器这个操作数是个立即数（一个具有常量值）。例如：

  ```
  #define DEFAULT         1
  ```

asm("li %0,%1\n\t" :"=r"(ret) :"**i**"(-TCP_MSS_DEFAULT));
此处的"**i**"也可以使用"g"代替。

- “n”同约束字符“i”。
- “m”内存操作数，用在访存指令的地址加载和存储。例如：
  int* p = &a;
  asm("ld $2,%[addr]\n\t" : :[addr]"**m**"(p));
- “o”内存操作数，用在访存指令的地址加载和存储。在 MIPS 架构中功能同“m”。
- “+”修改操作数的权限为可读可写，通常只修饰输出操作数。
- “&”
- “I”在 MIPS 架构下用于标识此操作数是一个有符号 16 位的常数。“I”用于算术指令。例如：
  asm("daddiu $2,$2,%0\n\t" : :"I"(0x3));

8.3 理解 asm volatile("": : :"memory")的含义

asm volatile("": : :"memory")是我们平时经常遇到的内嵌汇编格式。其中有一个关键字 volatile 和一个破坏描述“memory”。当然这两个关键字不是必须同时出现的，使用时要根据情况。

首先我们要知道的是 MIPS 是多级流水线架构。编译器优化就会依赖这个特性在编译时调整指令顺序，让没有相关性的指令可以乱序执行，以充分利用 CPU 的指令流水线，提高执行速度。如果内嵌汇编的指令中直接使用了某些寄存器或内存。GCC 编译器在优化后很可能带来错误。这种情况我们可以使用 volatile 关键字来修饰。volatile 用于告诉编译器，严禁将此处的 asm 汇编语句与它之前和之后的 c 语句在编译时重组合。

如果你的内嵌汇编中使用了一段未知大小的内存，或者使用的内存用于在多线程。那么请务必使用约束字符“memory”。“memory”就是通知 GCC 编译器，此段内嵌汇编修改了 memory 中的内容，asm 之前的 c 代码块和之后的 c 代码块看到的 memory 可能是不一样的，对 memory 的访问不能依赖之前的缓存，需要重新加载。

第 9 章 MIPS 空间地址分配和符号解析

9.1 虚拟地址空间和页大小

在龙芯 3A/3B 处理器上，进程执行在一个 40bit 的虚拟地址空间，其地址可以从 0 到 2^{40}-1。内存管理硬件（Memory Management Unit 简称 MMU）负责把虚拟地址转换成物理地址，向程序隐藏物理地址，从而使得进程可以运行在系统的任何空间。进程可以使用的空间通常按段（Segments）来分割，典型的进程包括至少 3 个段空间，分别是代码段（也叫 text 段）、数据段（也叫 data 段）和栈空间（也叫 stack）。有动态链接的进程可以动态创建更多的段空间。

内存是按照页组织的，页是 Linux 系统下内存管理的最小单元。不同的系统页大小可能不同，这取决于处理器、MMU 和系统配置。我们可以通过如下命令查看当前系统的页大小：

```
$ getconf PAGE_SIZE
16384
```

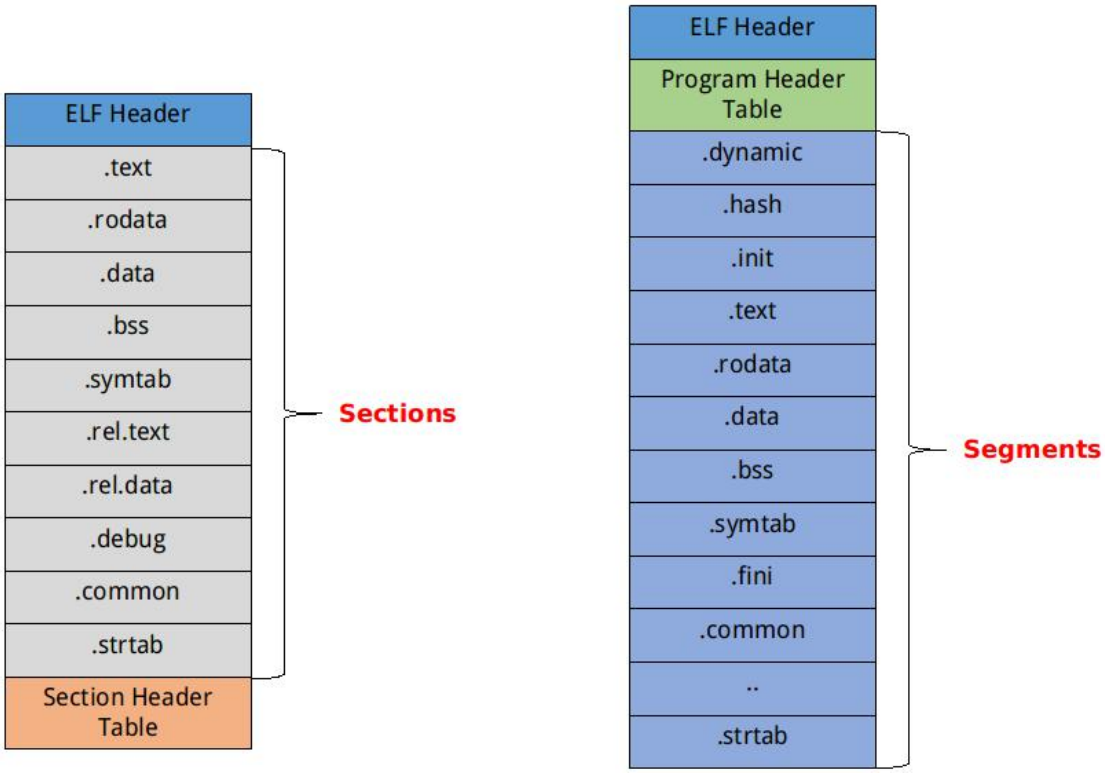
上面 getconf 命令显示当前系统的页大小是 16384Byte ,也就是 16K。

9.1 ELF 文件

ELF 文件（Executable and Linking Format）是用在 Linux 系统下的一种目标文件（object file）存储格式。典型的目标文件有如下 3 类：

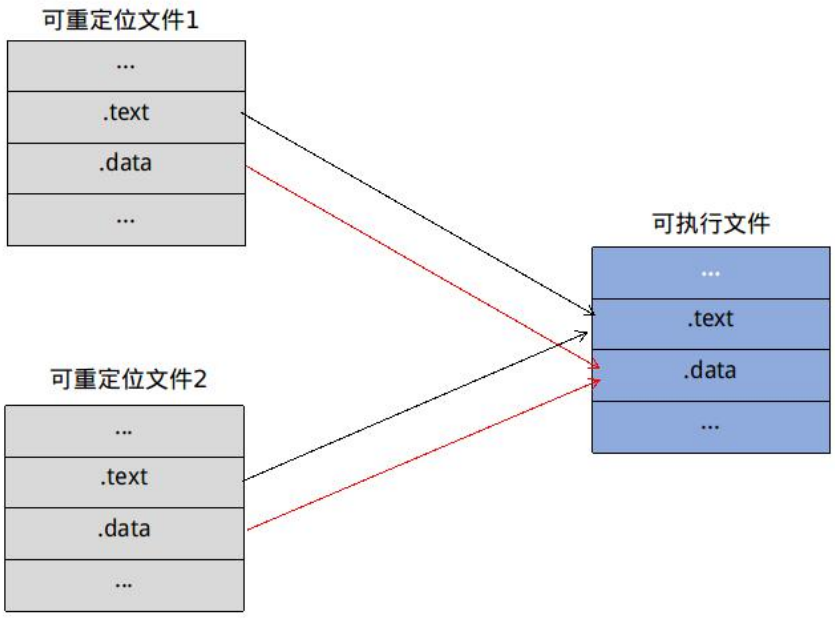
- 可重定向文件（relocatable file）可重定向文件里面包含了代码和数据，用于和其他可重定向文件一起链接形成一个可执行文件或者动态库。
- 可执行文件（executable file） 可执行文件里面包含了可以运行的程序代码。
- 动态库文件（shared object file） 动态库文件里面也包含了可用于链接的代码和程序。它用于 2 个过程，首先链接器把它和其他可重定向文件、动态库一起链接形成一个可执行文件。然后程序运行时，动态链接器负责在需要时动态加载动态库文件。

由汇编程序和链接编辑器生成的目标文件是可以在 CPU 上执行的二进制表示程序。目标文件统一遵守 ELF 格式。但是不同的系统架构，ELF 里面的格式和数据处理方式会略有不同，但是基本格式如下：



可重定向目标文件格式（左图）和可执行目标文件格式（右图）

ELF 文件格式根据文件类型不同有不同的组成方式。可重定向文件的格式基本由 ELF Header 、Sections、Section Header Table 组成。可执行文件的格式基本由 ELF Header、Segments、Program Header Table 组成。 其中可重定向文件中的节（Section）和可执行文件中的段（Segment）都是存储了程序的代码部分、数据部分等，区别是可执行目标文件中的某个段就是结合了很多可重定向目标文件中的相关节。如下图所示：



...

所以平时工作中，很多中国程序员并不会过多区分 section 和 segment，基本都称作段。甚至很多教材中也不做过多区分。本章介绍中也不会过多区分他们。

9.1.1 ELF 文件头

ELF 文件头描述了一个目标文件的组织，是对文件基本信息的描述，包括字的大小和字节次序（尾端）、ELF 文件头的大小、目标文件类型、机器类型、节头表/段头表的大小和数量、程序入口点等。ELF 文件头必须位于文件的最开始。我们可以使用命令“readelf -h 目标文件”来查看一个可执行目标文件的 ELF 头信息：

```
$ readelf -h hello
ELF 头：
 Magic：  7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
 Class:                             ELF64
 Data:                              2's complement, little endian
 Version:                           1 (current)
 OS/ABI:                            UNIX - System V
 ABI Version:                       0
 Type:                              REL (可重定位文件)
 Machine:                           MIPS R3000
 Version:                           0x1
 入口点地址：               0x0
 程序头起点：               0 (bytes into file)
 Start of section headers:          440 (bytes into file)
 标志：             0x80a20007, noreorder, pic, cpic, gs464, mips64r2
 本头的大小：       64 (字节)
 程序头大小：       0 (字节)
 Number of program headers:         0
 节头大小：         64 (字节)
 节头数量：         13
 字符串表索引节头： 10
```

参数“-h”就是代表 head。Magic 行的前 4 个字节“7f 45 4c 46”就标识了这是一个 ELF 格式的文件。第 5 个字节“02”代表文件运行在 64 位体系结构（“01”代表 32 位）。第 6 个字节“01”表示小尾端（02 代表大尾端字节序列）。第 7 个字节“01”代表 ELF 版本。后面的 9 个字节未定义。”Type:”显示文件是可重定位类型文件，程序头是可选项，这里程序头大小为 0 字节，表示没有程序头。

9.1.2 .O 文件中的段（section）和段表（section header tables）

一个可重定向目标文件中（一般以.o 为后缀的文件），除了 ELF 头、程序头和段表之外，所有的其他信息都包含在节（section）中。我们可以使用 readelf 带参数“-S”查看段信息（readelf -S 信息要全面于 objdump -h）：

]# readelf -S hello.o

共有 14 个节头，从偏移量 0x1f8 开始：

节头：

```
  [Nr] Name              Type            Address          Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS        0000000000000000  00000040
       00000000000000a0  0000000000000000  AX       0     0     16
  [ 2] .rela.text        RELA            0000000000000000  000006b0
       0000000000000078  0000000000000018          12     1     8
  [ 3] .data             PROGBITS        0000000000000000  000000e0
       0000000000000000  0000000000000000  WA       0     0     16
  [ 4] .bss              NOBITS          0000000000000000  000000e0
       0000000000000000  0000000000000000  WA       0     0     16
  [ 5] .MIPS.options     MIPS_OPTIONS    0000000000000000  000000e0
       0000000000000028  0000000000000001  Ao       0     0     8
  [ 6] .pdr              PROGBITS        0000000000000000  00000108
       0000000000000040  0000000000000000           0     0     4
  [ 7] .rela.pdr         RELA            0000000000000000  00000728
       0000000000000030  0000000000000018          12     6     8
  [ 8] .mdebug.abi64     PROGBITS        0000000000000000  00000148
       0000000000000000  0000000000000000           0     0     1
  [ 9] .comment          PROGBITS        0000000000000000  00000148
       000000000000002e  0000000000000001  MS       0     0     1
  [10] .gnu.attributes   LOOS+ffffff5    0000000000000000  00000176
       0000000000000010  0000000000000000           0     0     1
  [11] .shstrtab         STRTAB          0000000000000000  00000186
       0000000000000070  0000000000000000           0     0     1
  [12] .symtab           SYMTAB          0000000000000000  00000578
       0000000000000120  0000000000000018          13    11     8
  [13] .strtab           STRTAB          0000000000000000  00000698
       0000000000000016  0000000000000000           0     0     1
Key to Flags:
```

W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)

上面的信息看出：这个段表里面包含了 14 个段。每个段的段表结构如下:

（1）段名 Name：段名是个字符串，位于“.shstrtab”的段中。每个段的功能都不同，比如.text 用于存放代码、.data 段用于存放数据等，具体如下表所示。

（2）段类型 Type：目前有 11 中类型。程序中段类型以 SHT_开头，如 SHT_NULL 代表无效段、SHT_SYMTAB 代表符号表等，但是 readelf 中省略了 SHT_。具体段类型和含义如下表：

段类型	含义
SHT_NULL	无效段，忽略
SHT_PROGBITS	程序段。包括代码段、数据段、调试信息等，对应的段名有.text .data .pdr .common 等。
SHT_SYMTAB	符号表段。对应的段名是.symtab，里面存放了链接过程需要的所有符号信息。后面有详细介绍。
SHT_STRTAB	字符串表段。包括.strtab 和.shstrtab。.strtab 用来保存 ELF 文件中一般的字符串，如变量名、函数名等。.shstrtab 用于保存段表中用到的字符串，如段名 Name。
SHT_RELA	重定位表段（with explicit addends）。存放那些代码段和数据段中有绝对地址引用的相关信息，用于链接器的重定位。对应的段名有.rela.text .rela.data 等
SHT_HASH	符号表的哈希表
SHT_DYNAMIC	动态链接信息
SHT_NOTE	提示性信息
SHT_NOBITS	表示该段在文件中无内容，比如.bss 段
SHT_REL	重定位表段（without explicit addends）。对应的段名有.rel.text .rel.data 等
SHT_SHLIB	保留
SHT_DNYSYM	动态链接的符号表

（3）段的虚拟地址 Address：

段被加载到进程地址空间中的虚拟地址值；上面的信息读取的是可重定向目标文件 mips_book.o，在进程中的位置还不确定，所以所有段的虚拟地址都为 0。如果我们读取 mips_book.o 最终的可执行文件，那么段的虚拟地址都将是有效的，对应地址都是最终在内存中的虚拟地址，如下：

```
[10] .text             PROGBITS        00000001200008f0  000008f0
     0000000000000270  0000000000000000  AX       0     0     16
```

这里地址 0x1200008f0 就是.text 代码段最终在内存中的虚拟地址。

（4）段的偏移 Offset：

表示该段在 ELF 文件中的偏移。上面读取 hello.o 符号表信息中.text 段的 Offset 值为 0x40（64 字节），而“readelf -h hello.o”显示 hello.o 文件的头大小为 64 字节（如上面 **本头的大小： 64 (字节)**），说明 hello.o 的 ELF 文件中，头信息之后紧接着就是.text 段。

（5）段大小 Size：

表示该段的大小。上面信息中.text 的段大小为 0xa0，即代码段占用 160 个字节。

（6）段 Ensize：

…

（7）段的标志位 Flag：

表示该段在进程虚拟空间中的属性，比如是否可读、可写、可执行。比如代码段.text 的标志位为 AX 代表 alloc+execute，表示该段需要在内存开辟空间并且权限为可执行。如果你的程序里指针错误的往此段写数据，那么你肯定会收到一个没有写权限的段错误。数据段.data 的标志位为 WA 代表 write_alloc，表示该段权限为可写段并且需要在内存开辟空间。

（8）段的链接信息 Link Info：

如果段的类型与链接相关，比如重定位段、符号表段等，那么。。。

（9）段地址对齐 Align：

如果该段有对地址有要求，那么 Align 就指定了对齐方式。比如.text 段的 Align 值为 16 就代表该段在内存的必须以 16 对齐存放，即存放该段的内存起始地址必须可以被 16 整除。如果 Align 其值 0 和 1 没有意义。

目标文件的段满足如下条件：

- 每个节（section）都对应一个节头。目标文件中可以存在没有节信息的节头。
- 每个节（section）在目标文件中都占用一个连续的空间，这个空间可以是 0。
- 一个目标文件中的多个节（section）不能重叠。一个文件中没有一个字节位于多个节中。

ELF 中的各个部分是预定义的，并保存程序和控制信息。这些部分由操作系统使用，对于不同的操作系统具有不同的类型和属性。 下表 9-1 列举了 ELF 中定义的段名和功能介绍：

段名	功能描述
.text	常称为代码段，用于存放程序的可执行机器指令
.data 和.data1	数据段，用于存放程序中已经初始化的全局静态变量和局部静态变量
.rodata 和 .rodata1	只读数据段，用于存放只读数据，如 const 类型变量和字符串常量
.bss	用于存放未初始化的全局变量和局部静态变量
.common	用于存放编译器的版本信息
.hash	符号哈希表
.dynamic	动态链接信息
.strtab	字符串表，用来存放变量名、函数名等字符串。

.symtab	符号表，用于保存变量、函数等符号值。
.shstrtab	段名表，用于保存段名信息，如“.text” “.data”等
.plt 和 .got	动态链接的跳转表和全局入口表
.init 和 .fini	程序初始化和终结代码段

表 9-1 ELF 常见的段和段功能

还有一些用于调试的段，比如.debug 、.line 和存放编译器信息的段.note

9.1.3 可执行文件和动态库中的段（segment）程序头表（Program Header Table）

可执行文件和动态库（.so）文件是存在段（segment）和程序头表的。前面说过 segment 可以看作是多个.o 文件中的相似段（section）的合并，即一个 Segment 包含一个或多个属性相似的 Section。这里的属性相似更多是指权限，比如链接器会把多个.o 文件中的都具有可读可执行的 .text 和.init 段都放在最终可执行文件中的一个 Segment 段内，这样的好处就是可以更多的节省内存空间。这个原因是由于 ELF 文件被加载时，是以系统的页长度为单位，如果一个 Segment 的长度小于一个页大小，那么这个 Segment 也要占据整个页大小。连接器对具有相同权限的段 Section 合并到一个段 Segment 后，就可以尽可能的减少内存碎片。

描述 Section 属性的结构叫段表（Section Header Table）,描述 Segment 结构的叫程序头（Program Header Table），它 指导系统如何把多个 segment 段加载到内存虚拟空间。我们可以使用 readelf -l 来查看程序头表。

```
# readelf -l hello

Elf 文件类型为 EXEC (可执行文件)
入口点 0x1200008f0
共有 10 个程序头，开始于偏移量 64

程序头：
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000120000040 0x0000000120000040
                 0x0000000000000230 0x0000000000000230  R E    8
  INTERP         0x0000000000000bd0 0x0000000120000bd0 0x0000000120000bd0
                 0x000000000000000f 0x000000000000000f  R      1
    [正在请求程序解释器：/lib64/ld.so.1]
  LOAD           0x0000000000000000 0x0000000120000000 0x0000000120000000
                 0x0000000000000c94 0x0000000000000c94  R E    10000
```

```
  LOAD           0x0000000000000fe8 0x0000000120010fe8 0x0000000120010fe8
                 0x00000000000000e8 0x00000000000000f8  RW     10000
  DYNAMIC        0x0000000000000400 0x0000000120000400 0x0000000120000400
                 0x00000000000001f0 0x00000000000001f0  RWE    8
  NOTE           0x00000000000003d8 0x00000001200003d8 0x00000001200003d8
                 0x0000000000000024 0x0000000000000024  R      4
  NOTE           0x0000000000000c74 0x0000000120000c74 0x0000000120000c74
                 0x0000000000000020 0x0000000000000020  R      4
  GNU_EH_FRAME   0x0000000000000be0 0x0000000120000be0 0x0000000120000be0
                 0x000000000000001c 0x000000000000001c  R      4
  GNU_RELRO      0x0000000000000fe8 0x0000000120010fe8 0x0000000120010fe8
                 0x0000000000000018 0x0000000000000018  R      1
  NULL           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000         8

 Section to Segment mapping:
  段节...
   00
   01     .interp
   02     .MIPS.options .note.gnu.build-id .dynamic .hash .dynsym .dynstr .gnu.version
.gnu.version_r .init .text .MIPS.stubs .fini .rodata .interp .eh_frame_hdr .eh_frame .note.ABI-tag
   03     .init_array .fini_array .jcr .data .rld_map .got .sdata .bss
   04     .dynamic
   05     .note.gnu.build-id
   06     .note.ABI-tag
   07     .eh_frame_hdr
   08     .init_array .fini_array .jcr
   09
```

从上面可以看出，程序表中包含 10 个 Segment。每个 Segment 包含如下属性：

- Segment 类型 Type :类型分为 LOAD、DYNAMIC、NOTE、NULL 等。LOAD 类型的 Segment 是需要被加载到内存的。
- Segment 偏移 Offset：表示此 Segment 在 ELF 文件中的偏移。
- Segment 虚拟地址 VirtAddr：表示此 Segment 在进程内存中的起始地址。
- Segment 物理地址 PhysAddr：？
- Segment 在文件中的大小 FileSiz：此 Segment 在 ELF 文件中所占空间的长度。
- Segment 在内存中的大小 MemSiz：此 Segment 在进程内存中所占的长度。对于代码段，此值和 FileSiz 相等，但是对于数据段，此值可能大于 FileSiz。

- Segment 权限属性 Flags：权限属性包括可读 R、可写 W 和可执行 X。

- Segment 对齐属性 Align：只此 Segment 在内存加载时的对齐方式。其值为 2 的 Align 次方。比如上面的 Align 值为 4，那么对齐要求就是 16。

从上面也可以看出。所有相同属性的 Section 被归类到一个 Segment 中，比如：

```
03     .init_array .fini_array .jcr .data .rld_map .got .sdata .bss
```

9.1.4 符号（Symbol）和符号表(Symbol Table)

在链接器中，函数和变量统称为符号（Symbol）,函数名和变量名称为符号名（Symbol Name）。符合可以分为如下种类：

- 局部符号：类似于函数内部定义的静态局部变量，类似于下面的变量 a,b。这类符号只在编译单元内部可见。

```
int fun(){
      static int a,b;
}
```

- 全局符号：定义在本目标文件的全局符号，可以被其他文件引用。例如下面的 global_var 、main：

```
int global_var;
int main(int arg ,char* arg[]){}
```

- 外部符号(External Symbol)：在本目标文件中引用的全局符号。比如我们经常使用的 printf 函数，它是定义在模块 libc 内的符号。

- 段名：由编译器产生的.text、.data 等段名符号。

每一个可重定向目标文件中都会有一个名为.symtab 的 Section 用来存放符号表（Symbol Table）。如果是可执行文件，还会有一个段.dynsym 存放动态符号表。符号表里面记录了目标文件中所有用到的符号和符合信息。我们可以使用“readelf -s 文件名”来查看一个目标文件中的符号表信息：

```
[root@localhost sunguoyun]# readelf -s hello.o

Symbol table '.symtab' contains 17 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS hello.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT   10
     7: 0000000000000000     4 OBJECT  LOCAL  DEFAULT    4 static_a.1959
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    9
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
    10: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
    11: 0000000000000000     0 SECTION LOCAL  DEFAULT   12
    12: 0000000000000000     0 SECTION LOCAL  DEFAULT   13
    13: 0000000000000000     8 OBJECT  GLOBAL DEFAULT   10 str
    14: 0000000000000004     4 OBJECT  GLOBAL DEFAULT  COM global_var
    15: 0000000000000000   136 FUNC    GLOBAL DEFAULT    1 main
    16: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
```

从上图对应的 C 代码如下：

```
#include <stdio.h>
char* str="HELLO";
int global_var;
int main(){
   int a = 0,b = 0;
   static int static_a=0;

   printf("%s %d \n",str,a+b+static_a);
   return 0;
}
```

从上图可以看出，hello.o 文件中共有 16 个符号，Num 从 0 到 15。每个符号都有如下属性：

- 符号名（Name）：如上图的最后一列。hello.c、static_a、str、main 和 printf。没有名字的符号是段，从列 Type 的 SECTION 可以看出。

- 符号值（Value）：每个符号都有一个对应的值，如果此符号是一个函数、变量，那么符号的值就是函数和变量的地址。上面 hello.o 是未做重定向的 ELF，所以符号值都为 0。可执行文件中，符号值可能是符号的虚拟地址、符号所在函数偏移等。

- 符号大小（Size）：对于变量，符号大小就是数据类型的大小，比如上面的 static_a 是 int 类型，所以 size 为 4。对于函数，符号大小就是该函数中所有指令的字节数，例如 main 函数中 c 程序对应的 MIIPS 指令行数为 33 行，每条指令占 4 个字节，所以 main 符号的的 size 为 136。

- 符号类型和绑定信息（Type & Bind）：符号类型（Type）分为如下种类：
  NOTYPE： 未知符号类型
  OBJECT： 表示该符号是个数据对象，比如变量、数组等。
  FUNC： 表示该符号是个函数或其他可执行代码。

SECTION：表示该符号为一个段。
FILE：　　该符号表示文件名。

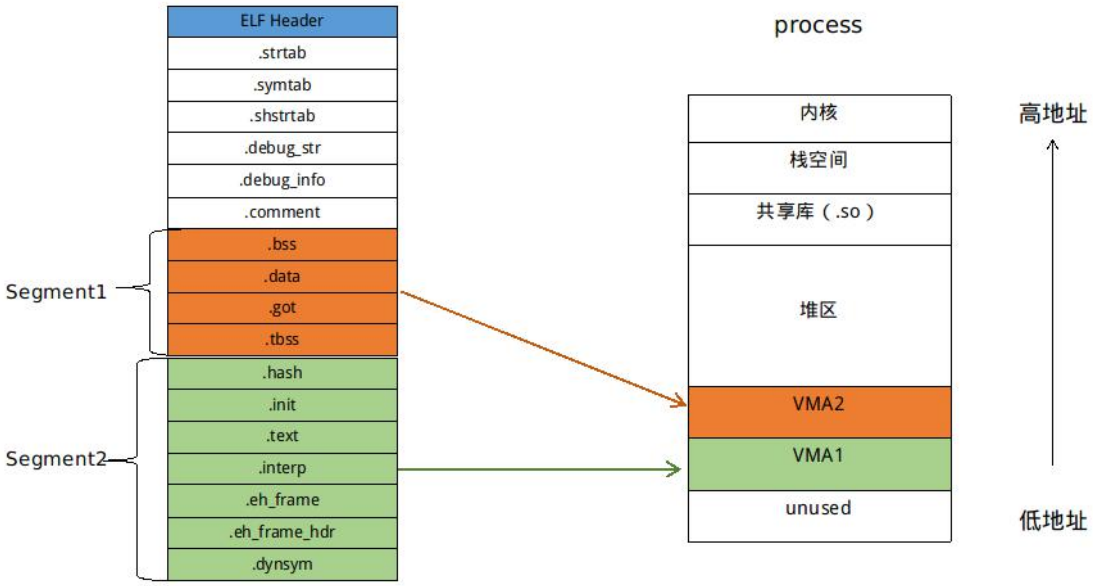
符号绑定信息分为如下种类：
LOCAL：　局部符号。
GLOBAL：全局符号，如上面定义的全局变量 global_var、函数 main、printf
WEAK：　弱引用符号。在这里没有体现。对于 C/C++语言，编译器默认函数和已经初始化的全局变量为强符号。而未初始化的全局变量和使用__attribute__((weak))定义的变量为弱符号。

- 符号所在段（Ndx）：如果符号定义在本目标文件中，那么这个成员表示符号所在的段在段表中的下标。比如静态变量 static_a 所在的段标为 4。Ndx 还有如下 3 中特殊值：

  UND：就表示这是个外部符号，不再本目标文件中定义，如 printf。
  ABS：表示该符号包含了一个绝对值，比如符号 hello.c。
  COM：表示该符号是个未初始化的全局符号。

9.2 可执行文件与进程虚拟空间映射关系

系统启动一个进程后，加载 ELF 文件的方式如下图所示：



进程与ELF文件的映射关系

从上图可以看出，ELF 格式的文件在映射入进程时，ELF 头和一个调试信息是不再需要的。只需要加载部分段（Segment）到内存，具体就是加载类型为 LOAD 的段即可。VMA 代表 Virtual Memorry Address，即虚拟地址空间。

实际开发中，我们还可以通过查看“/proc/pid/maps”节点来查看一个进程的虚拟空间分布：

```
[root@localhost testdir]# cat /proc/14330/maps
120000000-120004000 r-xp 00000000 fd:02 919128                           /home/sunguoyun/work/testdir/main_l
120010000-120014000 rwxp 00000000 fd:02 919128                           /home/sunguoyun/work/testdir/main_l
fff56b4000-fff5880000 r-xp 00000000 fd:01 1192731                        /usr/lib64/libc-2.20.so
fff5880000-fff588c000 ---p 001cc000 fd:01 1192731                        /usr/lib64/libc-2.20.so
fff588c000-fff5890000 r-xp 001c8000 fd:01 1192731                        /usr/lib64/libc-2.20.so
fff5890000-fff5898000 rwxp 001cc000 fd:01 1192731                        /usr/lib64/libc-2.20.so
fff5898000-fff589c000 rwxp 00000000 00:00 0
fff58a8000-fff58b8000 rwxp 00000000 00:00 0
fff58b8000-fff58e0000 r-xp 00000000 fd:01 1192528                        /usr/lib64/ld-2.20.so
fff58e8000-fff58ec000 rwxp 00000000 00:00 0
fff58ec000-fff58f0000 rwxp 00024000 fd:01 1192528                        /usr/lib64/ld-2.20.so
fffb9a4000-fffb9c8000 rwxp 00000000 00:00 0                              [stack]
fffbff4000-fffbff8000 r-xp 00000000 00:00 0
fffdec0000-fffdec4000 r--p 00000000 00:00 0                              [vvar]
fffdec4000-fffdec8000 r-xp 00000000 00:00 0                              [vdso]
```

其中第一列为 VMA 的地址范围。第二列为 WMA 的权限，分为可读 r、可写 w、可执行 x、私有 p、可共享 s。第三列为 WMA 对应的 Segment 在映射文件中的偏移。第四列表示映像文件所在设备的主设备号和次设备号。第五列表示映像文件的节点号。最后一列是映像文件的路径。

上面图中显示此进程共占用 15 个 VMA。前两个 VMA（12000000-120004000、120010000-120014000）映射到 ELF 文件的两个 Segment。程序用到的动态库 libc-2.20.so 被映射到了接下来的 4 个 VMA。接下来的 2 个 VMA 区没有显示映像文件的路径，实际上是堆区（Heap）,程序中可以通过类似 malloc 函数来申请使用。 ld-2.20.so 是 Linux 下的动态链接器，负责 libc-2.20.so 中和绝对地址相关的代码和数据的动态重定向过程，它占据了 2 个 VMA。后面的 2 个 VMA 被 vvar 和 vdso 模块占用，这两个模块是内核映射出来的 2 个区域，用于程序可以绕过系统调用 syscall 而直接和内核快速通信的一些接口，比如在大部分的体系架构中，gettimeofday 都是通过 vdso 直接实现来提高运行速度。

第三部分 汇编调试手段和移植事项（未完成）

第 10 章 汇编调试手段

10.1 break 指令和 beqz 指令
10.2 使用 ejtag 调试程序

第 11 章 应用程序移植注意事项