2.4. 硬件对过程的支持

本节新使用到的寄存器有$v0~$v1（值寄存器），$a0~$a3（参数寄存器），$gp（全局指针寄存器），$sp（栈指针寄存器），$fp（帧指针寄存器）和$ra（返回地址寄存器）共 10 个通用寄存器。以及，一个专用寄存器 PC（程序计数器）。

此外，会对保留寄存器和临时寄存器有进一步认识。

过程（procedure）是根据提供的参数执行一定任务的子程序，有助于提高程序的理解性和代码的可重用性。函数是一种典型的过程。

2.4.1. 过程的执行

在过程运行中，程序必须遵循以下 6 个步骤：

(1) 主程序（调用者，caller）将参数放在过程（被调用者，callee）可以访问的位置；

(2) 主程序将控制权转交给过程；

(3) 过程申请并获得其执行所需的存储资源；

(4) 过程执行需要的任务；

(5) 过程将结果的值放在主程序可以访问的地方；

(6) 过程把控制权交还给主程序，即返回调用点并执行主程序的下一条指令。

以上 6 个步骤提出了 3 个问题：

(1) 主程序和过程怎样访问彼此提供的值？

主程序的参数：参数寄存器（$a0~$a3）；过程的结果：值寄存器（$v0~$v1）。

(2) 控制权怎样移交？

在 MIPS 中，控制权的移交是通过跳转指令实现的。

(3) 怎么继续执行主程序？

这就需要一个寄存器（instruction address register）来保存当前正在被执行指令的地址，该寄存器是一个专用寄存器，被称为程序计数器（program counter, PC）。此外，还需要返回地址寄存器（$ra）来保存主程序下一条指令的地址。

jal-jr 指令对

跳转和链接（jump-and-link, jal）指令：

```
jal Label
```

由主程序执行，分为两步：首先无条件跳转到标签的位置，接着将该 jal 指令的下一条指令的地址（返回地址）保存到寄存器$ra 中（jal 指令实际上将 PC+4 保存在寄存器$ra 中，从而链接到下一条指令）。

寄存器跳转（jump register, jr）指令：

```
jr rs
```

由过程执行，无条件跳转到寄存器所保存的地址（32 位）。jr 指令是将 rt 和 rd 置 0 的 R 类型指令。

通过以上设计，可以实现过程的执行：调用者将参数值放在参数寄存器中，然后使用

```
jal X
```

跳转到被调用者 X，控制权交给了被调用者；被调用者执行任务，将结果放在值寄存器中，然后使用

```
jr $ra
```

将控制权返回给调用者，跳转到调用者的下一条语句。

2.4.2. 寄存器的值的保护

执行一个过程（函数）时，除了传入的参数使用的参数寄存器和存放过程结果的值寄存器外，还需要其他寄存器来保存过程执行的中间变量。

Note

此处可以联想 C 语言中函数的一般变量只能在函数执行过程中生存，一经调用结束就消失了。毕竟，函数也是一种过程。

2.4.2.1. 栈与寄存器的保护

过程如果要调用主程序使用的寄存器，需要先将寄存器在主程序执行中保存的值先放入内存的一段空间中。等过程执行完毕，清除过程执行中寄存器保存的值，再将主程序中原本的值再对应地恢复给这些寄存器。这样，对主程序使用的寄存器的值进行了“保护”。

这种“主程序-过程-过程-主程序”的“先入后出”数据结构，用栈来实现最合适。栈在内存中的增长是按照地址从高到低进行的，即栈顶为低地址、栈底为高地址。

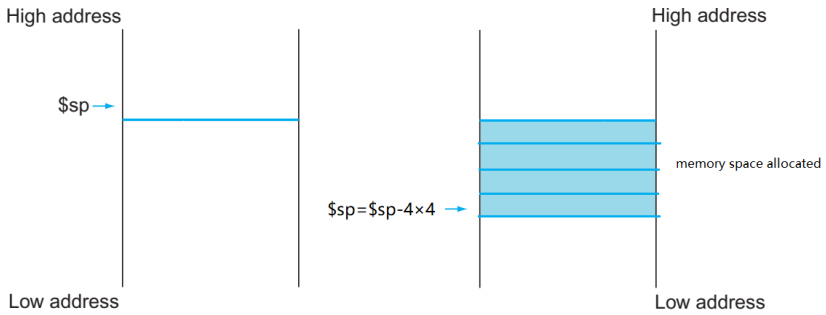

Figure 2-10 How $sp Works

$sp 里存放着内存中栈的“原址”，当使用$sp 时，需要预先由高到低分配一段空间；然后和使用数组一样进行地址的偏移，同样是 1 个字（4 个字节）的偏移量。

Note

此处应当区分不同地方对栈顶的规定，有的是指向栈顶元素（MIPS），有的是指向栈顶+1。

$sp 是栈指针寄存器，顾名思义，其使用和高级语言的指针方法相同，即指针本身是一个地址，指向地址存放的值。

$sp 永远指向栈顶元素，因此其作为基址，加上由高到低的栈增长，使栈像数组一样是正的偏移量而非负的。

栈在内存中，使用时需要用 lw 和 sw 指令。

因此，将数据压栈时，栈长度增大，栈指针减小；将数据弹栈时，栈长度减小，栈指针增大。

2.4.2.2. 保留和非保留寄存器

相比于访问寄存器，访问内存会慢很多，我们需要尽可能减少内存的访问，也就是减少对栈的使用。

我们约定，非保留寄存器在过程中不必由被调用者保存，保留寄存器在过程中必须被保存（一旦过程需要使用保留寄存器，必须由被调用者保存和恢复，相当于必须要保护保留寄存器里的值）。

编译器因此也不会对非保留寄存器进行栈操作来保护，如果需要(尽量不要)，则需要程序员自己编写相关代码。所以要在过程中尽可能使用非保留寄存器。

Note

对于保留寄存器，过程必须假定主程序需要使用这些寄存器，因而必须进行压栈和弹栈的操作；而对于非保留寄存器，过程不必进行假定，具体是否要保护临时寄存器里的值，要看主程序是否在调用过程结束后仍需要这些临时寄存器里的值。

保存寄存器就是保留寄存器的一种，临时寄存器时是非保留寄存器的一种。

Example 2.10

请写出下列 C 过程：

```
int leaf_exmaple (int g, int h, int i, int j)
{
    int f;
    f=(g+h)-(i+j);
    return f;
}
```

编译后的 MIPS 汇编代码。假定 g, h, i, j 保存在参数寄存器$a0, $a1, $a2, $a3，f 保存在$s0。

首先，我们考虑临时寄存器在调用过程后仍需被使用的情况：

```
leaf_example:
  addi $sp, $sp, -12 # adjust stack to make room for 3 items
  # push items
  sw $t1, 8($sp)
  sw $t0, 4($sp)
  sw $s0, 0($sp)
  # execute the procedure
  add $t0, $a0, $a1
  add $t1, $a2, $a3
  sub $s0, $t0, $t1
  add $v0, $s0, $zero # move $v0, $s0
  # pop items
  lw $s0, 0($sp)
  sw $t0, 4($sp)
  sw $t1, 8($sp)
  add $sp, $sp, 12 # recover
  # 将控制权还给主程序
  jr $ra
```

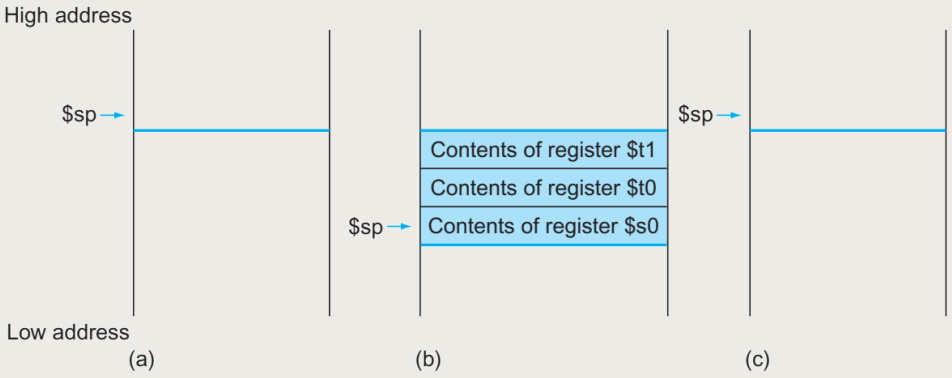

Figure 2-11 The Stack (a) Before, (b) During, and (c) after the procedure call

一般地，我们只需保护保留寄存器的值，故只需要为$s0 分配一个字空间即可，不必对临时寄存器（非保留寄存器）进行栈保护。

Note

为了减少对内存的访问，我们尽可能避免在过程执行时使用保留寄存器，因此上述代码可以优化为（假设 f 保存在$t0）：

```
leaf_example:
  # execute the procedure
  add $t0, $a0, $a1
  add $t1, $a2, $a3
```

```
sub $v0, $t0, $t1 # 减少了再把 f 放到值寄存器的过程
jr $ra # 将控制权还给主程序
```

开头的过程标签和结束的 jr 指令是任何一个过程所必须的。

2.4.3. 嵌套过程

不调用其他过程的过程称为叶（leaf）过程，否则称为嵌套（nested）过程。

在嵌套过程中，除了必须保护保存寄存器里的值，还必须需要保护返回寄存器的值（以返回上一个过程的下一条指令）以及栈指针寄存器和栈指针之上的栈（可以恢复被保存的值），这些都是和保存寄存器一样的保留寄存器。

至于返回值和参数寄存器，则和临时寄存器一样根据需要选择是否保留，都属于非保留寄存器。

Example 2.11

下面是一个计算阶乘的递归过程：

```
int fact (int n)
{
    if (n<1) return 1;
    else return n*fact(n-1);
}
```

请写出该过程的 MIPS 汇编代码。

参数 n 对应的参数寄存器为$a0

每次递归，我们都需要向栈申请 2 字空间，来保存当前过程的参数 n 和当前过程被上一过程（最顶级的上一过程是主程序）jal 指令调用时的跳转链接。

整个递归过程应当如 Figure 2-12 所示：

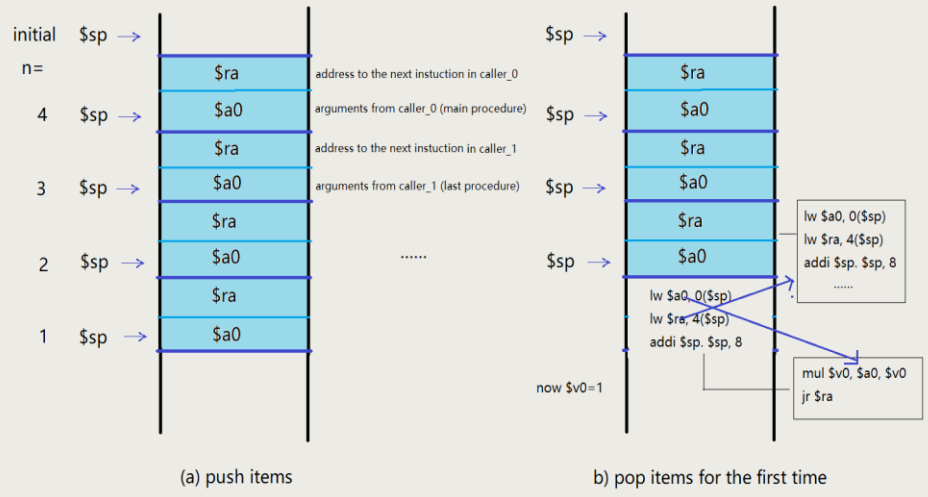

Figure 2-12 Recursion when n=4 from the Caller

```
fact:
  # protect/save stack
  addi $sp, $sp, -8
  sw $ra, 4($sp)
  sw $a0, 0($sp)
  slti $t0, $a0, 1 # 执行过程尽可能充分利用临时寄存器
  beq $t0, $zeros, L1
  addi $v0, $zero, 1
  jr $ra
L1:
  addi $a0, $a0, -1
  jal fact
  # recover
  lw $a0, 0($sp)
  lw $ra, 4($sp)
  addi $sp, $sp, 8
  mul $v0, $a0, $v0 # 乘法指令
  jr $ra
```

递归与迭代

递归和迭代都是程序中常用的两种循环方式。递归是指程序调用自身的编程思想，即一个函数调用本身；迭代是利用已知的变量值，根据递推公式不断演进得到变量新值的编程思想。两者的区别在于递归是一种**调用机制**，而迭代是一种**循环机制**。

调用机制使递归会产生在栈上的额外开销，一些递归过程可以使用迭代来避免这些开销，从而显著提高性能。

例如求 n!时，递归过程如上所示，迭代过程为：

```
for (int acc=1; n>0; --n) acc*=n;
```

有一种被称为尾调用（tail call）的递归调用，这种递归函数的参数多了一个表示迭代结果的参数。因此，尾递归（tail recursion）不需要在栈上产生额外开销，实际上是一种以递归形式表示的迭代。

求 n!的尾递归形式为：

```
int fact (int n, int acc)
{
  if (n>0) return fact(n-1, acc*n);
  else return acc;
}
```

假设 n=3，初始化 acc=1，调用 fact(3, 0)的过程为：fact(2, 3)、fact(1, 6)和 fact(0, 6)。然后将结果 6 进行 4 次返回操作，将控制权还给主程序。实际上，这和 for 循环迭代的本质相同。

编译为 MIPS 汇编语言为：

```
fact:
  slti $t0, $a0, 1
  bne $t0, $zero, fact_exit
  mul $a1, $a1, $a0
  addi $a0, $a0, -1
  j fact
fact_exit:
  add $v0, $a1, $zero # return value acc
  jr $ra # return to caller
```

Note

递归和迭代都有各自的优缺点，递归的优点是代码简洁，缺点是递归深度过大会导致栈溢出；迭代的优点是效率高，缺点是代码复杂。

2.4.4. 程序和数据的 MIPS 内存分配

内存中，低端地址的第一部分是保留的；接着是**代码段（text segment）**，存储 MIPS 机器代码；代码段之上的是**静态数据段（static data segment）**，存储常量和其他静态变量；静态数据段之后是**堆（heap）**和栈。

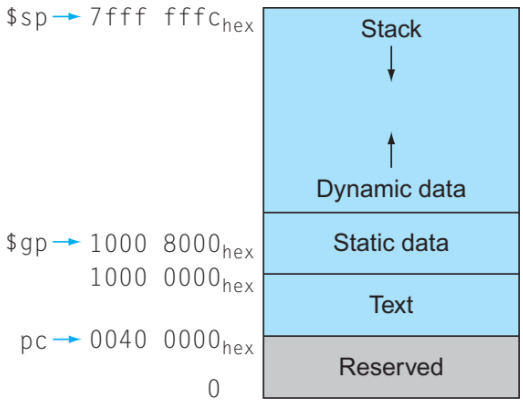

Figure 2-13 The MIPS Memory Allocation for Program and Data

下面从低地址到高地址说明各内存部分（被保留的部分除外）的一些细节。

2.4.4.1. 代码段

访问代码段借助的是专用寄存器 PC，它指向代码段的底端（最低地址）。

2.4.4.2. 静态数据段

C 语言包括两种存储变量的方式：**动态的（automatic）**和**静态的（static）**。动态变量位于过程中，当过程结束调用时失效；静态变量在进入和退出过程时始终存在。在所有过程之外声明的变量（如全局变量和常量），以及声明时在 C 语言用“static”关键字的变量，都被视为静态变量，其余则被视为动态变量。

为了简化静态数据的访问，MIP 保留了一个成为全局指针的寄存器$gp，指向静态数据段的中间地址（0x1000 8000），通过对$gp 正负 16 的偏移量就可以访问 0x1000 0000 到 0x1000 ffff 之间的内存空间。

$gp 也是一个保留寄存器。

2.4.4.3. 堆

动态数据存放在堆中，堆在静态数据段之后，与栈在同一部分。栈从高到低增长，堆从低到高增长，二者此消彼长的过程中可以达到内存的高效使用。

2.4.4.4. 栈

栈除了要保存一些需要被保护的寄存器的值，还需要存储过程中不适合用寄存器存储的局部变量（如数组和结构体）。栈中包含过程所保存的寄存器和局部变量的片段成为**过程帧（process frame）**或**活动记录（activation record）**。

Note

栈中存放需要保存的寄存器以及不适宜用寄存器保存的局部变量，堆中存放动态数据。

高级语言中，栈由系统进行分配和释放，里面的数据是从高到低的连续存储；堆由开发人员申请分配和释放（例如 C 的 malloc()和 C++的 new），里面的数据可以不连续存储，会因为忘记释放空间而造成内存泄漏。

一般栈所占的内存远比堆小。

某些 MIPS 软件使用帧指针寄存器$fp 一个固定的基址。因为栈指针在过程中可能会改变（被保护内容的出栈和入栈），所以使用固定的帧指针引用变量会更为简单。

如果一个过程中栈内无局部变量（允许有被保护的寄存器，栈指针只是方便引用局部变量，被保护的寄存器一般不需要引用），编译器可以不设置和恢复帧指针，以此节省时间。

帧指针不是必需的，GNU MIPS C 编译器使用帧指针，一些编译器（the C compiler from MIPS）没有帧指针寄存器，而是将 30 号寄存器用作另一个保存寄存器$s8。

当一个过程使用$fp 时，需要先对调用者的栈底（帧指针）进行压栈，以保存调用者的信息，然后令$fp 指向此地址。再用$fp 对$sp 进行初始化，开始利用$sp 进行被调用者的需要保留的寄存器和局部变量的栈空间分配，$fp 固定。

如果过程需要超过 4 个参数，MIPS 约定将参数寄存器之外的参数放在栈中帧指针的上方，通过帧指针在内存中寻址获得这些参数。

Figure 2-14 展示了使用帧指针的调用前和调用中的栈中情况：

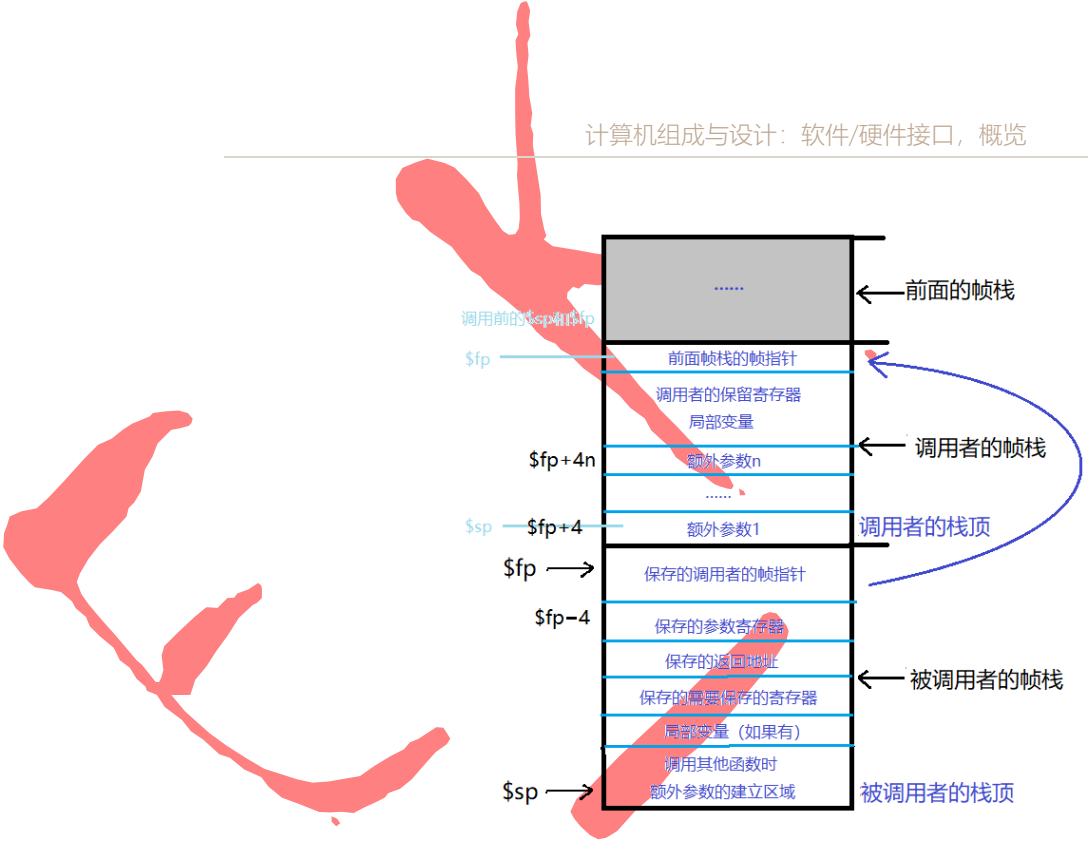

Figure 2-14 The Stack Allocation before and during the procedure call with $fp

当结束调用时，可以利用$fp 恢复$sp（addi $sp, $fp, -4），同时$fp 也可以恢复到调用者的帧指针（lw $fp, 0($fp)）。

Note

$fp 是寄存器，存放的是内存的一个地址，该地址存放的是调用者的帧指针的地址。所以恢复$sp 时，需要的是$fp 存放的那个内存地址，在寄存器中用add/addi；恢复$fp 时，需要的是$fp 存放的那个内存地址存放的值（上一个帧指针的地址），在内存中用 lw。

Figure 2-15 展示了过程调用时的保留和非保留寄存器。

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

Figure 2-15 MIPS Register Conventions

除此之外，称为$at 的 1 号寄存器被汇编器所保留，称为$k0~$k1 的 26~27 号寄存器被操作系统所保留。