# FDU 计算机组成与体系结构 2. MIPS 体系结构

本文参考以下教材:

- Digital Design and Computer Architecture (D. M. Harris, S. L. Harris 2rd) Chapter $6$
- 数字设计和计算机体系结构 (D. M. Harris, S. L. Harris 2rd 陈俊颖译) 第 $6$ 章
- 计算机组成与系统结构 (袁春风、唐杰、杨若瑜、李俊) 第 $4$ 章

欢迎批评指正!

---

**体系结构** (architecture) 是程序员所见到的计算机,
它由**指令集** (instruction sets) (汇编语言) 和**操作数地址** (operand location) (寄存器、存储器) 来定义.

理解任何计算机体系结构的第一步是学习它的语言.
计算机语言中的单词叫作**指令** (instruction),计算机的词汇表叫作**指令集** (instruction set)
计算机指令包含需要完成的**操作** (operation) 和需要使用的**操作数** (operand) 两部分,
其中操作数来自存储器、寄存器或者指令自身.

计算机硬件只能理解二进制信息,
因此指令也被编码为二进制数,其格式称为**机器语言** (machine language)
可以读取并执行机器语言指令的数字系统称为**微处理器** (Micro-processors)
为了方便人类的阅读和理解,我们还使用符号格式来表示指令,称为**汇编语言** (assemble language)

我们将介绍 **MIPS 体系结构** (Microprocessor without Interlocked Piped Stages architecture),
(本门课程中,我们只介绍32位 MIPS 体系结构,不考虑64位 MIPS 体系结构)
并理解 MIPS 体系结构设计的 4 个准则:

- 简单设计有助于规整化 (simplicity favors regularity);
- 加快常见功能 (make the common case fast);
- 越小的设计越快 (smaller is faster);
- 好的设计需要好的折中方法 (good design demands good compromises);

## 2.1 MIPS 汇编的基本格式

MIPS 属于**精简指令集计算机** (RISC, Reduced Instruction Set Computer) 体系结构,
其指令集通过仅仅包含简单、常用的指令以使常见的情况能较快执行.
指令种类少,意味着用于指令译码的硬件比较简单、快捷.
更复杂但不常见的操作由若干条简单指令构成的序列执行.
具有复杂指令的体系结构,例如 $\text{Intel x86}$,
称为**复杂指令集计算机** (Complex Instruction Set Computer, CISC)
它们实现复杂指令的代价在于,增加了硬件复杂度,并且降低了简单指令的执行速度.

### 2.1.1 助记符 (Mnemonic)

```
# opcode destination, source1, source2
add a, b, c
```

汇编指令的第一部分 (例如 `add`) 称为**助记符**,它指明需要执行的操作.
在此例中,加法操作 `add` 基于**源操作数** (source operand) `b` 和 `c`,
运算结果写入**目的操作数** (destination operand) `a`

## 2.1.2 操作数 (Operand)

一条指令的操作需要基于**操作数** (operand).
操作数可以存放在**寄存器** (register) 或**存储器** (memory)，也可作为**常数** (constant) 存储在指令自身中.
计算机通过综合使用不同的位置存放操作数，以便在性能和存储容量之间作权衡，
使得程序可以以相对较快的速度访问大量的数据:

- 寄存器、常数形式的操作数的访问速度非常快，但它们只能包含少量数据；
- 存储器的容量更大，但访问速度更慢；

### (1) 寄存器 (Register)

MIPS 体系结构提供了 32 个通用寄存器，称为**寄存器堆** (register file)，用于存放常用的操作数.
其中 18 个寄存器用于存储变量:

- 以 `$s` 开头的寄存器 `$s0`～`$s7` 称为**保存寄存器** (saved register)
  按照 MIPS 的惯例，它们用于存储长期变量 (长期数据) 以及函数调用过程中应当被保留的数据.
- 以 `$t` 开头的寄存器 `$t0`～`$t9` 称为**临时寄存器** (temporary register)
  按照 MIPS 的惯例，它们用于存储临时变量 (临时数据) 以及函数调用过程中不需要被保留的数据.

例如 C 语句 `a = b + c - d;` 对应的 MIPS 汇编代码为:
(我们用 `$s0`～`$s3` 存储长期变量 `a,b,c,d`，用 `$t0` 存储临时变量 `t = c - d`)

```
# $s0 = a, $s1 = b, $s2 = c, $s3 = d
# $t0 = t
sub $t0, $s2, $s3  # t = c - d
add $s0, $s1, $t0  # a = b + t
```

其余的寄存器将在本文的后续内容介绍.

| Register Name | Software Name | Usage |
|---|---|---|
| $0 | | Always has the value 0 |
| $1 | at | Reserved for the assembler |
| $2 ～ $3 | v0 ～ v1 | Hold the integer type function result |
| $4 ～ $7 | a0 ～ a3 | Pass integer type arguments for function (caller-saved) |
| $8 ～ $15 | t0 ～ t7 | Temporary registers used for expression evaluations (caller-saved) |
| $16 ～ $23 | s0 ～ s7 | Callee-saved registers across procedure calls |
| $24 ～ $25 | t8 ～ t9(jp) | Temporary registers used for expression evaluations (caller-saved)<br>Sometimes, jp is Position-Independent Code (PIC) jump register |
| $26 ～ $27 | k0 ～ k1 | Reserved for the operating system kernel |
| $28 | gp | Global pointer. |
| $29 | sp | Stack pointer. |
| $30 | fp(s8) | Frame pointer (if needed), otherwise a saved register (like s0-s7). |
| $31 | ra | Return address |

### (2) 存储器 (Memory)

MIPS 采用**字节寻址存储器** (byte-addressable memory)
也就是说，存储器中的每一字节都有一个单独的地址.
一个 32 位的字 (word) 包含 4 个 8 位字节 (byte)，所以每一个字地址都是 4 的倍数.
按照惯例，我们用 16 进制表示 32 位字地址和数据值.

- **字节的排序方式:**
  大多数机器上，多字节对象都被存储为连续的字节序列，其地址为所使用字节中最小的地址.
  而关于字节的排序方式却存在两种规范:
  - ① **大端法 (big endian): 最高有效字节** (MSB, Most Significant Byte) 在最前端 (最低位地址);
  - ② **小端法 (little endian): 最低有效字节** (LSB, Least Significant Byte) 在最前端 (最低位地址);

  一个具体例子:
  考虑一个 `int` 类型的变量 `x=0x12345678`，地址为 `0x100`，
  其最高、最低有效字节分别为 `0x12`、`0x78`，
  地址范围 `0x100`~`0x103` 的字节顺序依赖于机器是大端还是小端:

大端法:

| ... | 0x100 | 0x101 | 0x102 | 0x103 | ... |
|---|---|---|---|---|---|
| ... | 12 | 34 | 56 | 78 | ... |

小端法:

| ... | 0x100 | 0x101 | 0x102 | 0x103 | ... |
|---|---|---|---|---|---|
| ... | 78 | 56 | 34 | 12 | ... |

  业界至今还没有在字节顺序这一问题上达成共识.
  有趣的是，术语 `big endian`、`little endian` 出自 `Gulliver's Travels` 的第一卷第四章，
  小人国的居民们因在吃鸡蛋时应该先打破大端还是小端而争论不休，甚至引发了战争.

  对于大多数应用程序员来说，机器所使用的字节顺序是完全不可见的，
  无论是大端还是小端都会得到相同的结果.
  然而有些时候，字节顺序的分歧可能会成为问题，
  例如通过网络二进制数据时，不同类型的机器之间传送的信息在对方看来是反序的.

**我们以大端存储器为例介绍 `lw` 和 `sw` 指令:**

| ⋮ | | | | | ⋮ |
|---|---|---|---|---|---|
| 0000 000C | A D | F F | C D | 0 7 | Word 3 |
| 0000 0008 | 2 0 | 1 6 | 0 1 | 1 1 | Word 2 |
| 0000 0004 | 2 0 | 0 3 | 0 3 | 2 0 | Word 1 |
| 0000 0000 | 1 2 | 3 4 | 5 6 | 7 8 | Word 0 |

```
lw $s0, 0($0)   # load word 0 (0x12345678) into $s0
lw $s1, 4($0)   # load word 1 (0x20030320) into $s1
lw $s2, 8($0)   # load word 2 (0x20160111) into $s2
lw $s3, 12($0)  # load word 3 (0xADFFCD07) into $s3 "AD" for "Albus Dumbledore"

sw $s0, 4($0)   # save $s0 (0x12345678) into word 1
sw $s1, 400($0) # save $s1 (0x20030320) into word 100
sw $s2, 0x20($0)# save $s2 (0x20160111) into word 8
```

`lw` 和 `sw` 指令指定内存**字地址** (word address) 为**基地址** (base address) 和**偏移量** (offset) 的和.
基地址为寄存器，写在括号内;
偏移量为常数，写在括号前面;
MIPS 体系结构要求**字地址**必须是**字对齐的** (word aligned)，即必须能被 4 整除.
字对齐的字地址称为**有效地址** (effective address)，否则称为**非法地址** (illegal address)

- `lw` 指令将有效地址指定的数据字加载到目的寄存器中;
- `sw` 指令将目的寄存器中的值存入有效地址指定的数据字;

MIPS 体系结构也提供了 `lb` 和 `sb` 指令来加载或存储单字节的数据 (将在后文 $2.3.2\ (5)$ 介绍)
它们使用的是字节地址，自然不用字对齐.

### (3) 常数(Constants)/立即数 (Immediates)

(2) 中 `lw` 和 `sw` 指令的示例也说明了 MIPS 体系结构中**常数**的用法，
因为常数的值可以被指令立即访问，而不需要通过访问寄存器或存储器来得到，
所以这些常数又称为**立即数** (Immediates)
在 MIPS 汇编指令中，除非另有指定 (例如加上了 `0x` 前缀)，立即数通常被解释为十进制数.

以 `addi` (加立即数) 指令为例:
C 语句 `a = a + 4;` 可以翻译成汇编代码 `addi $s0, $s0, 4`
指令中指定的立即数采用 16 位补码表示，其范围为 $-2^{15} = -32768 \sim 32767 = 2^{15} - 1$
由于减法相当于加上一个负数，故为了简单起见，MIPS 体系结构中没有 `subi` 指令.

## 2.2 MIPS 机器指令的基本格式

汇编语言方便人类阅读，但数字电路只能理解 `0,1`.
因此我们需要将汇编语言写的程序从符号格式转换为 `0,1` 序列表示的**机器语言** (machine language).

MIPS 使用 $32$ 位机器指令.
即使有些指令可能不需要所有 $32$ 位的编码，但可变长度的指令将增加太多的复杂性.
(简单的设计有助于规整化, simplicity favors regularity)
虽然简单化的设计原则鼓励使用单指令格式，但过于简单化将无法实现相应的功能，
所以 MIPS 采取了一个巧妙的折中，它定义了 $3$ 种指令格式: **R 类型、I 类型、J 类型**.

### 2.2.1 R 类型指令 (Register-type)

$$\text{R-type} \quad \text{rd}, \quad \text{rs}, \quad \text{rt}$$

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| (6) | (5) | (5) | (5) | (5) | (6) |

- R 型指令的 `op` (操作码, opcode) 字段为 `000000`
- R 型指令有 $3$ 个寄存器操作数: `rs`, `rt` 为源操作数， `rd` 为目的操作数
- `shamt` 字段仅用于移位指令，其二进制数值表示移位的位数.
  对于其他 R 型指令， `shamt` 为 `00000`
- R 型指令的 `funct` (函数, function) 字段决定 R 型指令的操作类型

### 2.2.2 I 型指令 (Immediate-type)

$$\begin{cases} \text{I-type} \quad \text{rt}, \quad \text{rs}, \quad \text{imm} & (\text{addi, andi, etc.}) \\ \text{I-type} \quad \text{rt}, \quad \text{imm(rs)} & (\text{lw, sw}) \end{cases}$$

| op | rs | rt | imm |
|---|---|---|---|
| (6) | (5) | (5) | (16) |

- I 型指令的操作类型由 `op` 字段唯一确定
  (不为 `000000` (R-type)、 `000010` (j, jump)、 `000011` (jal, jump and link))
- `rs` 为源操作数
- `rt` 通常用作目的操作数 (例如 `addi`, `sw` 等指令)，特殊情况用作源操作数 (例如 `lw` 指令)
- `imm` 为 16 位立即数，它会被扩展为 32 位立即数 `imm32` 以参与 32 位操作:
  - 算术操作中， `imm` 字段执行符号扩展 (以支持正、负立即数);
  - 逻辑操作 ( `andi`, `ori`, `xori` ) 中， `imm` 字段执行 $0$ 扩展;

## 2.2.3 J 型指令 (Jump-type)

J-type addr

| op | addr |
| --- | --- |
| (6) | (26) |

- `j` 指令的 `op` 字段是 `000010` (无条件跳转)
- `jal` 指令的 `op` 字段是 `000011` (无条件跳转并链接)
- `addr` 字段用于指定地址

## 2.2.4 机器指令与汇编指令的互译

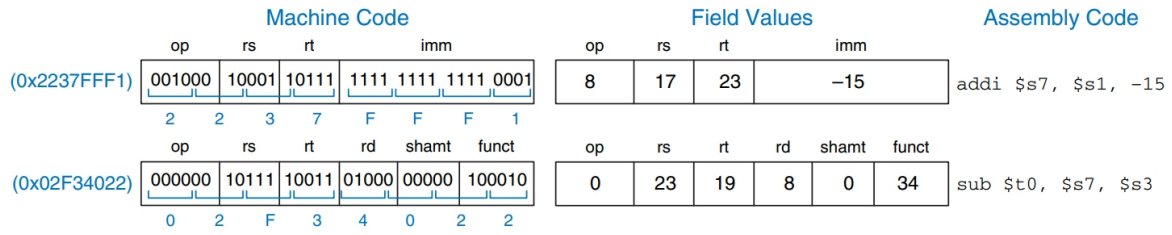



**Figure 6.12 Machine code to assembly code translation**

## 2.2.5 机器指令的存储

一条机器指令就是一个 32 位数，一个机器语言编写的程序就是一系列 32 位数.
与其他二进制数一样，机器指令被存储在存储器中，这就是**存储程序** (stored program) 的概念.
运行一个新的程序不需要对硬件进行重新装配或重新布线，
只需要将该程序写入存储器中，这使得计算器只需要改变存储程序就可以运行多种应用程序.
在 MIPS 体系结构中，指令一般从地址 `0x00400000` 开始存储，
每存入一条指令，指令地址递增 4 字节.

程序运行过程中，
存储程序中的机器指令从存储器中**取出** (fetch)，
交由处理器进行**解码** (decode) 和**执行** (execute).
当前指令的地址存储在**程序计数器** (PC, Program Counter) 中 (一个 32 位特殊寄存器)
每执行完一条指令，处理器将调整 `PC` 的值使之指向下一条指令，
接着取出并执行该指令，重复上述过程.

处理器的**体系结构状态** (architectural state) 代表程序的状态.
在 MIPS 体系结构中，体系结构状态由寄存器堆和 PC 构成.
操作系统可以在程序运行的某个时刻保存体系结构状态，
这样就可以中断该程序，转而去执行其他进程，
之后再恢复该状态，使得被中断的程序能够继续正确执行，而不知道它曾被中断过.

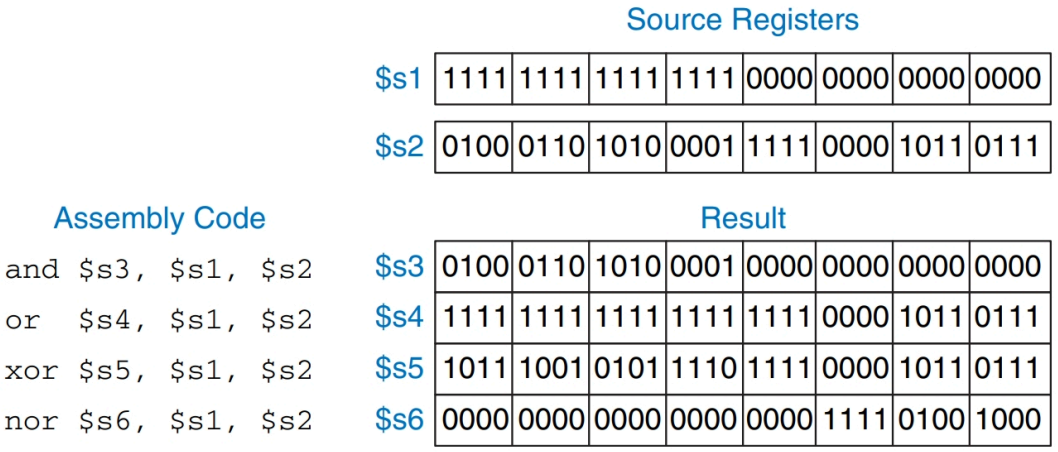
# 2.3 编程 (Programming)

高级编程语言使用算术/逻辑操作、if/else 语句、for 循环和 while 循环、数组下标和函数调用等软件结构，
本节我们将探讨如何将这些高级语言结构转换为 MIPS 汇编代码.

## 2.3.1 算术/逻辑指令 (Arithmetic/Logical Instructions)

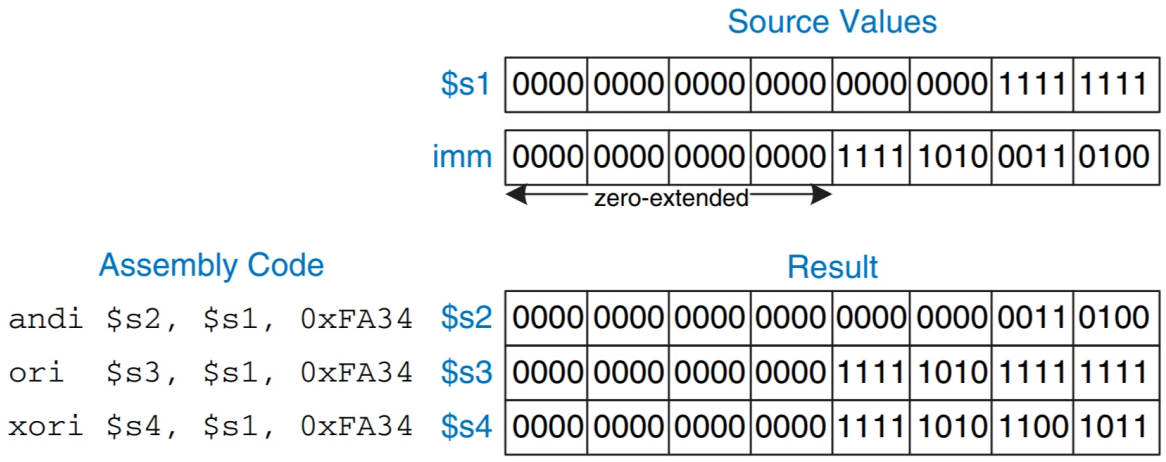
### (1) 逻辑指令 (Logical Instructions)

MIPS 逻辑操作包括 `and`, `or`, `xor`, `nor`
这些 R 型指令对两个源寄存器和一个目的寄存器进行**按位操作** (bitwise operation)




- `and` 指令可用于**屏蔽** (mask) 某些不需要的位 (设置为 0)
  寄存器位的任意子集都可以被屏蔽.
  例如图中的 `and` 指令通过 `$s1 = 0xFFFF0000` 屏蔽了 `$s2` 最低的两个字节,
  并将未被屏蔽的高两个字节 `0x46A1` 写入目的寄存器 `$s3`
- `or` 指令可用于**组合** (combine) 来自两个寄存器的位.
  例如 `0x12340000 OR 0x00005678 = 0x12345678`
- MIPS 不提供 `not` 指令，但是 `A NOR $0 = NOT A`
  因此 `nor` 指令可以代替 `not` 指令.
  其中 `nor` 指令的含义为 `rd = ~(rs | rt)`
- `andi`, `ori`, `xori` 指令也可以对立即数进行逻辑操作.
  MIPS 不提供 `nori` 指令，因为 `nori $s2, $s1, imm` 指令可由以下指令轻松实现:

```
ori $t0, $s1, imm
nor $s2, $t0, $0
```



## (2) 移位指令 (Shift Instructions)

移位指令可将寄存器中的值左移或右移至多 31 位.
移位操作可将操作数乘或除以 2 的整数次幂.
MIPS 移位指令包括**逻辑左移指令** `sll`、**逻辑右移指令** `srl`、**算术右移指令** `sra`

| Assembly Code | op | rs | rt | rd | shamt | funct | Machine Code | |
|---|---|---|---|---|---|---|---|---|
| | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits | | |
| `sll $t0, $s1, 4` | 0 | 0 | 17 | 8 | 4 | 0 | 000000 00000 10001 01000 00100 000000 | (0x00114100) |
| `srl $s2, $s1, 4` | 0 | 0 | 17 | 18 | 4 | 2 | 000000 00000 10001 10010 00100 000010 | (0x00119102) |
| `sra $s3, $s1, 4` | 0 | 0 | 17 | 19 | 4 | 3 | 000000 00000 10001 10011 00100 000011 | (0x00119903) |

**Figure 6.16** Shift instruction machine code

**Figure 6.17** Shift operations

Source Values

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $s1 | 1111 | 0011 | 0000 | 0000 | 0000 | 0010 | 1010 | 1000 |
| shamt | | | | | | | | 00100 |

| Assembly Code | Result | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| `sll $t0, $s1, 4` | $t0 | 0011 | 0000 | 0000 | 0000 | 0010 | 1010 | 1000 | 0000 |
| `srl $s2, $s1, 4` | $s2 | 0000 | 1111 | 0011 | 0000 | 0000 | 0000 | 0010 | 1010 |
| `sra $s3, $s1, 4` | $s3 | 1111 | 1111 | 0011 | 0000 | 0000 | 0000 | 0010 | 1010 |

MIPS 也提供**可变移位指令** (variable-shift instructions):
**可变逻辑左移指令** `sllv`、**可变逻辑右移指令** `srlv`、**可变算术右移指令** `srav`
其 `rs`, `rt` 顺序与大多数 R 型指令相反，为 `sllv rd, rt, rs`
其中 `rt` 保存待移位的值，
而 `rs` 的低 5 位给出了位移的位数，(`shamt` 字段为全 0 且被忽略)
移位结果存放在 `rd` 中.

| Assembly Code | op | rs | rt | rd | shamt | funct | Machine Code | |
|---|---|---|---|---|---|---|---|---|
| | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits | | |
| `sllv $s3, $s1, $s2` | 0 | 18 | 17 | 19 | 0 | 4 | 000000 10010 10001 10011 00000 000100 | (0x02519804) |
| `srlv $s4, $s1, $s2` | 0 | 18 | 17 | 20 | 0 | 6 | 000000 10010 10001 10100 00000 000110 | (0x0251A006) |
| `srav $s5, $s1, $s2` | 0 | 18 | 17 | 21 | 0 | 7 | 000000 10010 10001 10101 00000 000111 | (0x0251A807) |

**Figure 6.18** Variable-shift instruction machine code

Source Values

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $s1 | 1111 | 0011 | 0000 | 0100 | 0000 | 0010 | 1010 | 1000 |
| $s2 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | **1000** |

| Assembly Code | Result | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| `sllv $s3, $s1, $s2` | $s3 | 0000 | 0100 | 0000 | 0010 | 1010 | 1000 | 0000 | 0000 |
| `srlv $s4, $s1, $s2` | $s4 | 0000 | 0000 | 1111 | 0011 | 0000 | 0100 | 0000 | 0010 |
| `srav $s5, $s1, $s2` | $s5 | 1111 | 1111 | 1111 | 0011 | 0000 | 0100 | 0000 | 0010 |

**Figure 6.19** Variable-shift operations

## (3) 生成常数 (Generating Constants)

- `addi` 指令可用于 16 位常数赋值，
  例如 `int a = 0x1234` 可表示为 `addi $s0, $0, 0x1234`
- `lui` 和 `ori` 指令可用于 32 位常数赋值，
  即先使用一条装入高位立即数指令 `lui` 装入，
  接着使用一条 OR 立即数指令 `ori` 合并低 16 位.
  例如 `int a = 0x12345678` 可表示为:

```
lui $s0, 0x1234
ori $s0, $0, 0x5678
```

### (4) 乘法和除法指令 (Multiplication and Division Instructions)

`multi $s0, $s1` 将两个 32 位数相乘，产生一个 64 位乘积，
高 64 位存放在 `hi` 中，低 32 位存放在 `lo` 中.
`div $s0, $s1` 计算 `$s0/$s1`，余数存放在 `hi` 中，商存放在 `lo` 中.

- (move from hi) `mfhi $s2` 指令将 `hi` 中的值复制到 `$s2` 中
- (move from lo) `mflo $s3` 指令将 `lo` 中的值复制到 `$s3` 中

MIPS 还提供另一种乘法指令，
`mul $s1, $s2, $s3` 生成 32 位结果，并存储在 `$s1` 中.

## 2.3.2 分支 (Branching)

为了顺序执行指令，程序计数器 (PC) 执行一条指令后递增 4
分支指令改变程序计数器的值，跳过某段代码或返回到先前的代码.

- **条件分支** (condition branch) 指令执行一次测试，当测试结果为 `true` 时才执行分支语句
- **无条件分支** (uncondition branch) 指令总是执行分支语句，称为**跳转** (jump) 指令

### (1) 条件分支 (Conditional Branches)

MIPS 提供两种分支指令: `beq` (Branch if EQual) 和 `bne` (Branch if Not Equal)

- `beq $rs, $rt, imm` (这里 `$rs` 为第一个源寄存器，这种顺序与大部分 I 型指令相反)
  当两个寄存器值相等时，`beq` 执行分支语句；
  其中 `imm` 为指令地址.
- `bne $rs, $rt, imm`
  当两个寄存器值不相等时，`bne` 执行分支语句；

```
    addi $s0, $0, 4         # $s0 = 0+4 = 4
    addi $s1, $0, 1         # $s1 = 0+1 = 1
    sll $s1, $s1, 2         # $s1 = 1 << 2 = 4
    beq $s0, $s1, target    # $s0 == $s1, so branch is taken
    addi $s1, $s1, 1        # not executed
    sub $s1, $s1, $s0       # not executed
target: # 为突出标号，只缩进代码而不缩进标号
    add $s1, $s1, $s0       # $s1 = 4+4 = 8
```

其中标号 `target` 将转换为指令地址
MIPS 汇编语言标号后面跟着一个冒号 `:`，标号的名称不能使用指令助记符等保留字.
按照惯例，为突出标号，只缩进代码而不缩进标号.

### (2) 跳转指令 (Jump)

MIPS 提供三种跳转指令:
**跳转指令 `j`、跳转和链接指令 `jal`、跳转寄存器指令 `jr`**
`j` 指令直接跳转到标号指定位置的指令;
`jal` 指令与 `j` 指令类似 (它们都是 J 型指令)，但它保存返回地址;
`jr` 指令跳转到寄存器所保存的地址，它是 R 型指令，但只用到了 `rs` 寄存器;

- 使用 `j` 指令无条件跳转:

```
    addi $s0, $0, 4     # $s0 = 4
    addi $s1, $0, 1     # $s1 = 1
    j target            # jump to target
    addi $s1, $s1, 1    # not executed
    sub $s1, $s1, $s0   # not executed
target:
    add $s1, $s1, $s0   # $s1 = 1+4 = 5
```

- 使用 `jr` 指令无条件跳转:

```
0x00002000 addi $s0, $0, 0x2010 # $s0 = 0x00002010
0x00002004 jr $s0               # jump to 0x00002010
0x00002008 addi $s1, $0, 1      # not executed
0x0000200c sra $s1, $s1, 2      # not executed
0x00002010 lw $s3, 44($s1)      # executed after jr instruction
```

### (3) 条件语句 (If/Else Statements)

- **if 语句:**
  if 语句汇编代码的检测条件与高级语言代码相反.
  也就是说，若高级语言检测 `i == j`，则汇编代码使用 `bne` 检测 `i != j`
  例如以下 C 代码:

```
if (i == j)
    f = g + h;
f = f - i;
```

  翻译成 MIPS 汇编代码为:

```
# $s0 = f, $s1 = g, $s2 = h, $s3 = i, $s4 = j
    bne $s3, $s4, L1    # if i != j, skip if block
    add $s0, $s1, $s2   # if block: f = g + h
L1:
    sub $s0, $s0, $s3   # f = f - i
```

- **if/else 语句:**
  与 if 语句一样，if/else 语句汇编代码的检测条件与高级语言代码相反.
  例如以下 C 代码:

```
if (i == j)
    f = g + h;
else
    f = f - i;
```

翻译成 MIPS 汇编代码为:

```
# $s0 = f, $s1 = g, $s2 = h, $s3 = i, $s4 = j
    bne $s3, $s4, else  # if i != j, branch to else
    add $s0, $s1, $s2   # if block: f = g+h
    j L2                # skip past the else block
else:
    sub $s0, $s0, $s3   # else block: f = f – i
L2:
```

- **switch/case 语句:**
  switch/case 语句根据条件执行多块代码中的一块.
  如果不能满足条件，则执行 `default` 块.
  它相当于多条嵌套的 `if/case` 语句.
  例如以下 C 代码:

```
switch (amount) {
    case 20: fee = 2; break;
    case 50: fee = 3; break;
    case 100: fee = 5; break;
    default: fee = 0;
}

// equivalent function using if/else statements
if (amount = = 20) fee = 2;
else if (amount = = 50) fee = 3;
else if (amount = = 100) fee = 5;
else fee = 0;
```

翻译成 MIPS 汇编代码为:

```
# $s0 = amount, $s1 = fee
case20:
    addi $t0, $0, 20        # $t0 = 20
    bne $s0, $t0, case50    # amount = = 20? if not, skip to case50
    addi $s1, $0, 2         # if so, fee = 2
    j done                  # and break out of case
case50:
    addi $t0, $0, 50        # $t0 = 50
    bne $s0, $t0, case100   # amount = = 50? if not, skip to case100
    addi $s1, $0, 3         # if so, fee = 3
    j done                  # and break out of case
case100:
    addi $t0, $0, 100       # $t0 = 100
    bne $s0, $t0, default   # amount = = 100? if not, skip to default
    addi $s1, $0, 5         # if so, fee = 5
    j done                  # and break out of case
default:
    add $s1, $0, $0         # fee = 0
done:
```

### (4) 循环语句 (Getting Loopy)

循环根据某个条件重复地执行一块代码语句.

- **while 循环:**
  与 if/else 语句类似，while 循环汇编代码的测试条件与高级语言代码相反.
  以下 C 代码将 `pow` 值与 $128$ 进行比较:
  如果相等，就退出循环;
  否则，它将 `pow` 值乘以 $2$ (左移 $1$ 位)，递增 $x$，然后跳转到 while 循环的开始.

  ```
  int pow = 1;
  int x = 0;
  while (pow != 128)
  {
      pow = pow * 2;
      x = x + 1;
  }
  ```

  翻译成 MIPS 汇编代码为:

  ```
  # $s0 = pow, $s1 = x
      addi $s0, $0, 1         # pow = 1
      addi $s1, $0, 0         # x = 0
      addi $t0, $0, 128       # t0 = 128 for comparison
  while:
      beq $s0, $t0, done      # if pow = = 128, exit while loop
      sll $s0, $s0, 1         # pow = pow * 2
      addi $s1, $s1, 1        # x = x+1
      j while
  done:
  ```

- **for 循环:**
  与 while 循环类似，for 循环重复执行一段代码，直到某个条件不再满足.
  但是 `for` 循环增加了对**循环变量** (loop variable) 的支持，它跟踪循环执行的次数.
  例如以下 C 代码将数字从 $0$ 到 $9$ 相加:

  ```
  int sum = 0;
  for (i = 0; i != 10; i = i + 1)
      sum = sum + i ;

  // equivalent to the following while loop
  int sum = 0;
  int i = 0;
  while (i != 10) {
      sum = sum + i;
      i = i + 1;
  }
  ```

  翻译成 MIPS 汇编代码为:

```
# $s0 = i, $s1 = sum
    add  $s1, $0, $0    # sum = 0
    addi $s0, $0, 0     # i = 0
    addi $t0, $0, 10    # $t0 = 10
for:
    beq  $s0, $t0, done # if i == 10, branch to done
    add  $s1, $s1, $s0  # sum = sum + i
    addi $s0, $s0, 1    # increment i
    j for
done:
```

- **量值比较 (Magnitude Comparison):**
  到目前为止，我们只使用了 `beq` 和 `bne` 指令执行相等或不相等的比较和分支.
  为应对分支条件为量值比较的情况，MIPS 提供了**小于设置指令** `slt` (Set on Less Than)
  `slt $rd, $rs, $rt`
  当 `$rs < $rt` 时，`slt` 将 `$rd` 设置为 1；否则，将 `$rd` 设置为 0
  例如以下 C 代码:

  ```
  int sum = 0;
  for (i = 1; i < 101; i = i * 2)
      sum = sum + i;
  ```

  翻译成 MIPS 汇编代码为:

  ```
  # $s0 = i, $s1 = sum
      addi $s1, $0, 0     # sum = 0
      addi $s0, $0, 1     # i = 1
      addi $t0, $0, 101   # $t0 = 101
  loop:
      slt $t1, $s0, $t0   # if (i < 101) $t1 = 1, else $t1 = 0
      beq $t1, $0, done   # if $t1 == 0 (i >= 101), branch to done
      add $s1, $s1, $s0   # sum = sum + i
      sll $s0, $s0, 1     # i = i*2
      j loop
  done:
  ```

## (5) 数组 (Arrays)

数组用于访问大量类似的数据.
数组按照存储器中顺序数据地址组织，每一个数组元素由**下标** (index) 区分.
数组中元素的个数称为数组的**长度** (size)

- **数组下标 (Array Indexing):**
  在主存中，数组从**基地址** (base address, 即 array[0] 的地址) 开始连续地存储.
  访问数组元素的第一步便是将数组的基地址装入寄存器 (可以使用 `lui` 和 `ori` 指令)
  例如以下 C 代码使用 `for` 循环将基地址为 `0x23B8F000` 的数组中的所有 1000 个元素乘以 8:

  ```
  int i;
  int array[1000];
  for (i = 0; i < 1000; i = i + 1)
      array[i] = array[i] * 8
  ```

  翻译成 MIPS 汇编代码为:

```
# $s0 = array base address, $s1 = i
# initialization code
    lui $s0, 0x23B8         # $s0 = 0x23B80000
    ori $s0, $s0, 0xF000    # $s0 = 0x23B8F000
    addi $s1, $0, 0         # i = 0
    addi $t2, $0, 1000      # $t2 = 1000
loop:
    slt $t0, $s1, $t2       # i < 1000?
    beq $t0, $0, done       # if not, then done
    sll $t0, $s1, 2         # $t0 = i * 4 (byte offset)
    add $t0, $t0, $s0       # address of array[i]
    lw $t1, 0($t0)          # $t1 = array[i]
    sll $t1, $t1, 3         # $t1 = array[i] * 8
    sw $t1, 0($t0)          # array[i] = $t1 = array[i] * 8
    addi $s1, $s1, 1        # i = i + 1
    j loop                  # repeat
done:
```

- **访问字符数组 (Character Array):**
  MIPS 使用 `lb` 和 `sb` 访问字符数组，其数组元素之间的地址变化是 1 个字节而不是 4 个字节.
  例如以下 C 代码意图将数组中的小写字母减去 32，以转化为大写字母:

```
char chararray[10];
int i;
for (i = 0; i != 10; i = i + 1)
    chararray[i] = chararray[i] - 32;
```

翻译成 MIPS 汇编代码为:

```
# MIPS assembly code
# $s0 = base address of chararray, $s1 = i
    addi $s1, $0, 0         # i = 0
    addi $t0, $0, 10        # $t0 = 10
loop:
    beq $t0, $s1, done      # if i == 10, exit loop
    add $t1, $s1, $s0       # $t1 = address of chararray[i]
    lb $t2, 0($t1)          # $t2 = array[i]
    addi $t2, $t2, -32      # convert to upper case: $t2 = $t2 - 32
    sb $t2, 0($t1)          # store new value in array: chararray[i] = $t2
    addi $s1, $s1, 1        # i = i + 1
    j loop                  # repeat
done:
```

一个字符序列称为**字符串** (string)
其长度可变，C 语言中使用空字符 `0x00` 代表字符串的结束.
例如字符串 `"Hello!"` (0x48 65 6C 6C 6F 21 00) 在小端机器中的存储形式为:

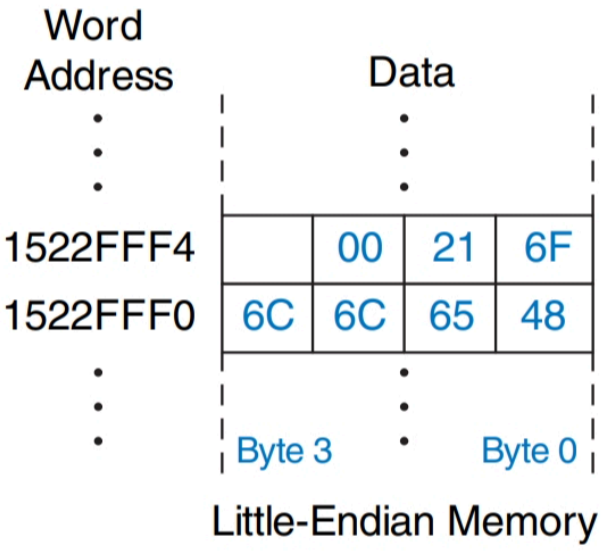



**Figure 6.23 The string "Hello!" stored in memory**

## 2.3.3 函数调用 (Function Calls)

函数，又称为**过程** (procedure)，
它能方便我们重用经常使用的代码，并使程序更加模块化和可读.
函数的输入和输出分别称为**参数** (arguments) 和**返回值** (return value)

当一个函数调用其他函数时，
**调用函数** (caller) 和**被调用函数** (callee) 必须在参数和返回值上保持一致.
MIPS 汇编的惯例是:
调用函数在调用前要将参数存放在 `$a0` ~ `$a3` 中，
被调用函数在完成前要将返回值存放在 `$v0` ~ `$v1` 中.

**被调用函数不能影响调用函数的功能.**
这意味着被调用函数必须知道当它完成后要返回到哪里，
而且它不能破坏调用函数用到的寄存器和内存.

- 调用函数将**返回地址** (return address) 存储在 `$ra` 寄存器中，
  并使用 `jal` 指令跳转到**被调用函数入口**.
- 被调用函数必须保证**存储寄存器** `$s0` ~ `$s7` 和 `$ra` 以及用于存放临时变量的**栈** (stack) 不被修改.

### (1) 函数调用和返回 (Function Calls and Returns)

MIPS 使用 `jal` 指令调用一个函数，使用 `jr` 指令从函数返回.
例如:

```
0x00400200 main: jal simple     # call function
0x00400204  ...
    ...     ...
0x00401020 simple: jr $ra       # return
```

`main` 函数通过执行 `jal` 指令调用 `simple` 函数.
`jal` 跳转到 `simple` 标号，同时将 `0x00400204` 存储到 `$ra` 寄存器中.
`simple` 通过执行 `jr $ra` (跳转到 `$ra` 寄存器保存的指令地址) 立即返回.
`main` 函数从地址 `0x00400204` 处继续执行.

## (2) 输入参数和返回值 (Input Arguments and Return Values)

根据 MIPS 惯例:
调用函数在调用前要将参数存放在 `$a0`～`$a3` 中，(多于 $4$ 个的参数放入栈中)
被调用函数在完成前要将返回值存放在 `$v0`～`$v1` 中 (返回 $64$ 位值的函数将使用两个返回寄存器).
例如:

```
# $s0 = y
main:
        ...
    addi $a0, $0, 2     # argument 0 = 2
    addi $a1, $0, 3     # argument 1 = 3
    addi $a2, $0, 4     # argument 2 = 4
    addi $a3, $0, 5     # argument 3 = 5
    jal diff_of_sums    # call function
    add $s0, $v0, $0    # y = returned value
        ...
# $s0 = result
# $a0 = f, $a1 = g, $a2 = h, $a3 = i
diff_of_sums:
    add $t0, $a0, $a1   # $t0 = f + g
    add $t1, $a2, $a3   # $t1 = h + i
    sub $s0, $t0, $t1   # result = (f + g) - (h + i)
    add $v0, $s0, $0    # put return value in $v0
    jr $ra              # return to caller
```

注意这里的 `diff_of_sums` 函数并不符合 MIPS 惯例:
它没有保存和复原寄存器 `$s0`，`$t0` 和 `$t1` 的值，从而可能对 `main` 函数产生负面影响.
我们会在 $(3)$ 中改正 `diff_of_sums` 函数的写法.

## (3) 栈 (Stack)

**栈**是用于存储函数中局部变量的存储器.
当处理器需要更多空间时，栈会扩展 (使用更多内存);
当处理器不再需要存在栈中的变量时，栈会缩小 (使用较少的内存)

**栈**是一个**后进先出队列** (LIFO, Last-In-First-Out Queue),
它在内存中是**向下增长**的 (grows down),
当一个程序需要更多的空间时，栈空间会向内存中地址较低的方向扩展.
**栈指针** (stack pointer) 指向**栈顶** (top of stack),
即最后分配的空间，也即栈的最低可访问内存.
栈指针 `$sp` 开始于一个高内存地址，通过地址的递减来扩展栈空间.
如图所示:

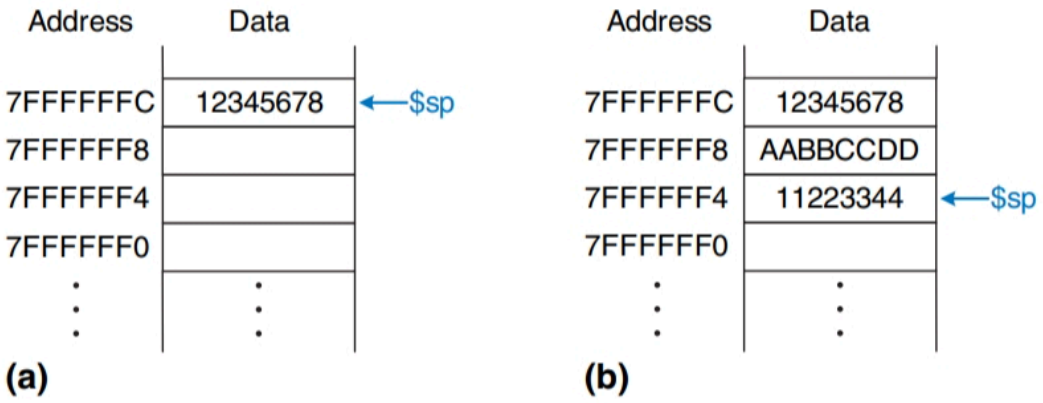



**Figure 6.24 The stack**

**栈的一个重要应用就是保存和恢复调用函数使用的寄存器.**
被调用函数应该计算返回值，但不应该产生其他负面影响.
除了包含返回值的寄存器 `$v0` (和 `$v1`，如果结果为 64 位数) 外，
其他任何寄存器都不能被修改，即使被修改了也必须在调用结束后复原.

我们现在给出 (2) 中 `diff_of_sums` 函数的改进版本，它保存和恢复 `$s0`, `$t0`, `$t1`:
(后面我们会看到，由于 `$t0`, `$t1` 属于不受保护的寄存器，对它们的保存和恢复实际上是无用的操作)

```
# $s0 = result
diff_of_sums:
    addi $sp, $sp, -12      # make space on stack to store three registers
    sw $s0, 8($sp)          # save $s0 on stack
    sw $t0, 4($sp)          # save $t0 on stack
    sw $t1, 0($sp)          # save $t1 on stack

    add $t0, $a0, $a1       # $t0 = f+g
    add $t1, $a2, $a3       # $t1 = h+i
    sub $s0, $t0, $t1       # result = (f + g) - (h + i)
    add $v0, $s0, $0        # put return value in $v0

    lw $t1, 0($sp)          # restore $t1 from stack
    lw $t0, 4($sp)          # restore $t0 from stack
    lw $s0, 8($sp)          # restore $s0 from stack
    addi $sp, $sp, 12       # deallocate stack space
    jr $ra                  # return to caller
```

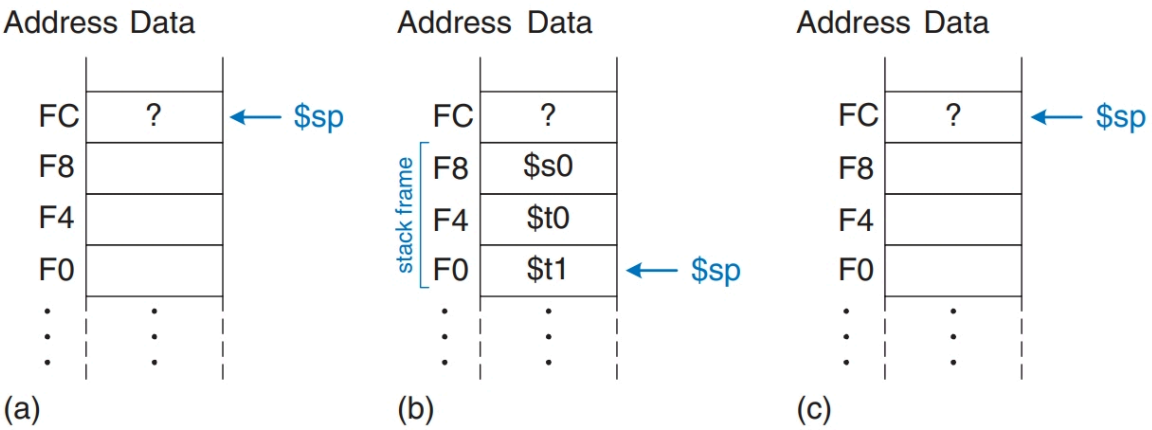



**Figure 6.25 The stack (a) before, (b) during, and (c) after `diffofsums` function call**

### (4) 受保护寄存器 (Preserved Registers)

MIPS 将寄存器划分为**受保护类型** (preserved) 和**不受保护类型** (nonpreserved)

- 受保护寄存器包括 `$s0` ~ `$s7`, `$ra` 和 `$sp` (又称**被调用者保存** (callee-save) 的寄存器)
  (注意，栈指针 `$sp` 是自动保护的，因为被调用函数在返回前需要回收自己的栈空间)
- 不受保护寄存器包括 `$t0` ~ `$t9`, `$a0` ~ `$a3` 和 `$v0` ~ `$v1` (又称**调用者保存** (caller-save) 的寄存器)
  (注意，`$a0` ~ `$a3` 必须由调用函数保存，因为它们可能包含调用函数的自身参数)

我们规定:
**函数必须保存和恢复任何需要使用的受保护寄存器，但是可以随意改变不受保护寄存器.**

我们现在给出 (3) 中 `diff_of_sums` 函数的进一步改进版本，
它只保存和恢复 `$s0`，而无需保存 `$t0`, `$t1`:

```
# $s0 = result
diff_of_sums:
    addi $sp, $sp, -4       # make space on stack to store one register
    sw $s0, 0($sp)          # save $s0 on stack

    add $t0, $a0, $a1       # $t0 = f+g
    add $t1, $a2, $a3       # $t1 = h+i
    sub $s0, $t0, $t1       # result = (f + g) - (h + i)
    add $v0, $s0, $0        # put return value in $v0

    lw $s0, 0($sp)          # restore $s0 from stack
    addi $sp, $sp, 4        # deallocate stack space
    jr $ra                  # return to caller
```

### (5) 递归函数调用 (Recursive Function Calls)

不用调用其他函数的函数称为**叶子函数** (leaf function)
需要调用其他函数的函数称为**非叶子函数** (nonleaf function)
非叶子函数在调用其他函数之前，需要将**调用者保存**的寄存器保存到栈中，在调用后再恢复这些寄存器.

**递归函数** (recursive function) 是调用自己的非叶子函数.
一个典型的例子就是**阶乘函数** (factorial function):

$$\text{factorail}(n) = \begin{cases} 1, & n \leq 1 \\ n \times \text{factorial}(n-1), & n \geq 2 \end{cases}$$

其 C 代码为:

```c
int factorial(int n)
{
    if (n <= 1)
        return 1;
    else
        return (n * factorial(n - 1));
}
```

其 MIPS 汇编代码为:

```
0x90 factorial:
        addi $sp, $sp, -8   # make room on stack
0x94    sw $a0, 4($sp)      # store $a0
0x98    sw $ra, 0($sp)      # store $ra
0x9C    addi $t0, $0, 2     # $t0 = 2
0xA0    slt $t0, $a0, $t0   # n <= 1 ?
0xA4    beq $t0, $0, else   # no: goto else
0xA8    addi $v0, $0, 1     # yes: return 1
0xAC    addi $sp, $sp, 8    # restore $sp
0xB0    jr $ra              # return
0xB4 else:
        addi $a0, $a0, -1   # n = n - 1
0xB8    jal factorial       # recursive call
0xBC    lw $ra, 0($sp)      # restore $ra
0xC0    lw $a0, 4($sp)      # restore $a0
0xC4    addi $sp, $sp, 8    # restore $sp
0xC8    mul $v0, $a0, $v0   # n * factorial(n-1)
0xCC    jr $ra              # return
```

以 `factorial(3)` 为例展示递归调用期间栈的变化:

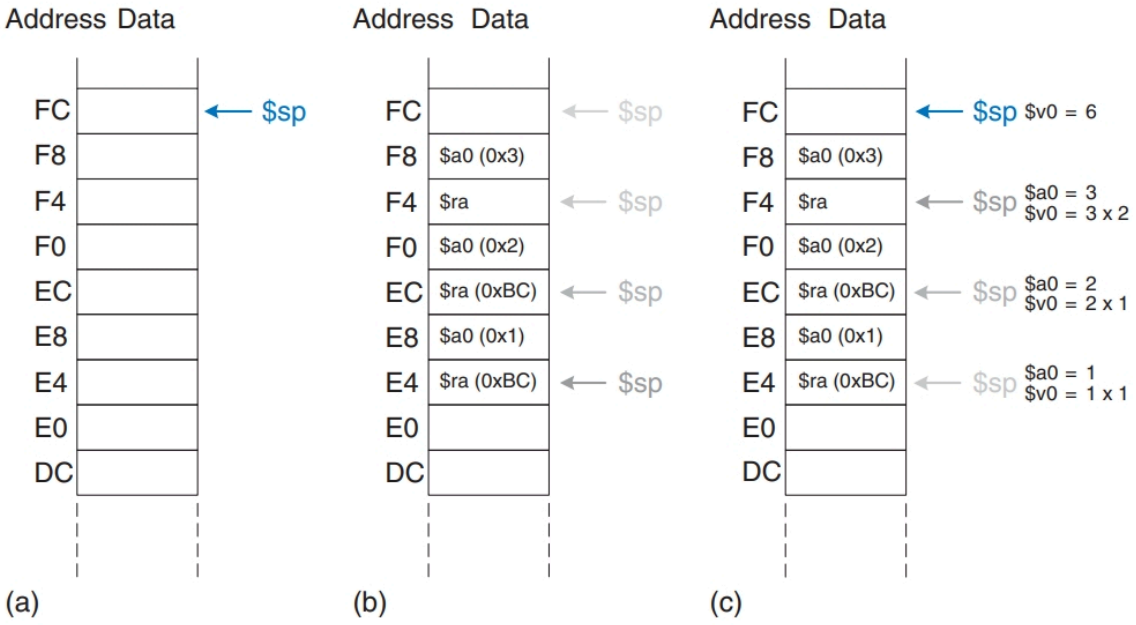



**Figure 6.26** **Stack during** `factorial` **function call when** $n = 3$**:**
**(a) before call, (b) after last recursive call, (c) after return**

## (6) 附加参数和局部变量 (Additional Arguments and Local Variables)

**函数可能有多于 4 个的参数.**
根据 MIPS 惯例，如果一个函数有 4 个以上的参数，
则前 4 个参数像往常一样存储在参数寄存器 `$a0` ~ `$a3` 中，额外的参数保存在栈中.

**函数也可以声明局部变量或数组.**
局部变量 (local variable) 在一个函数内部定义并且只能在该函数内部使用，
一般存储在保存寄存器 `$s0` ~ `$s7` 中
如果有多于 8 个的局部变量，则也存储在栈中.
特别地，局部数组存储在栈中.

被调用函数可以从调用函数的**栈帧** (stack frame) 中找到额外的输入参数和局部变量 (数组)
这种情况下，函数可以访问不属于自己栈帧中的内容.

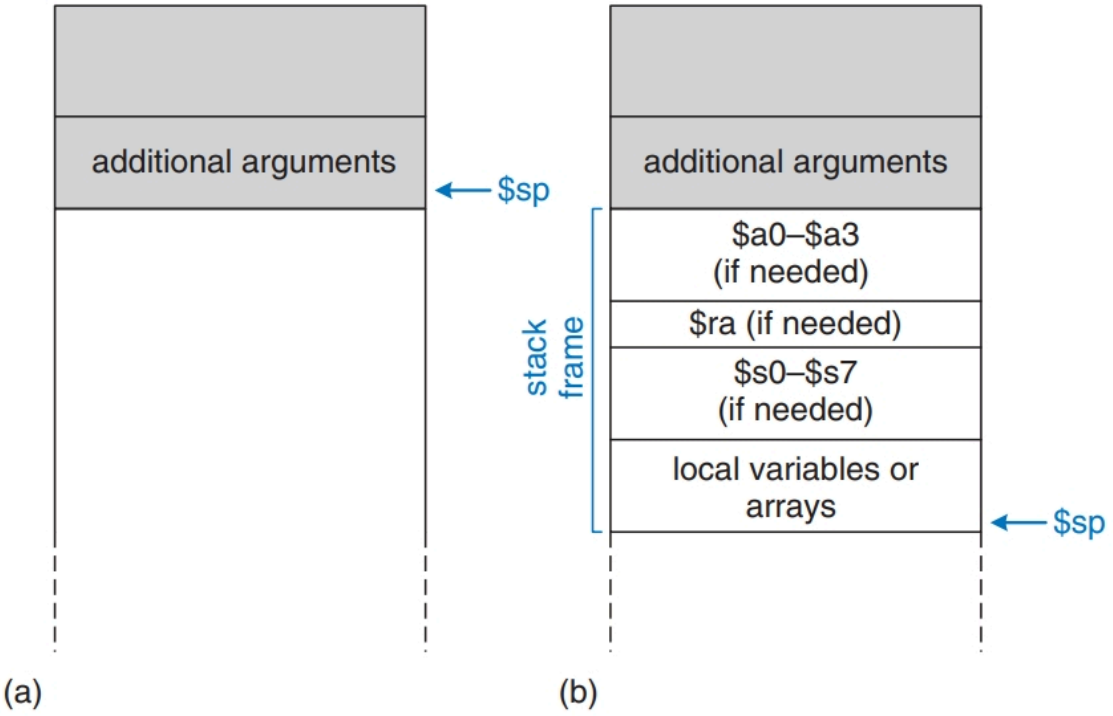



**Figure 6.27 Stack usage: (a) before call, (b) after call**

## 2.4 寻址方式 (Address Mode)

MIPS 使用 5 种寻址方式:
寄存器寻址 、立即数寻址、基地址寻址、PC 相对寻址和伪直接寻址
前 3 种方式定义读/写操作数的模式，
后 2 种寻址方式 (PC 相对寻址和伪直接寻址) 定义写程序计数器 PC 的方式.

- **寄存器寻址 (register-only addressing):**
  使用寄存器存储所有源操作数和目的操作数，所有 R 型指令都采用寄存器寻址.
- **立即数寻址 (immediate addressing):**
  使用 16 位立即数 (存储于指令中) 和寄存器作为操作数，部分 I 型指令 (例如 `addi`) 采用立即数寻址.
- **基地址寻址 (base addressing):**
  存储器访问指令 (例如 `lw` 和 `sw`) 都使用基地址寻址.
  存储器操作数的有效地址由寄存器 `rs` 中的基地址与立即数字段的符号扩展的 16 位偏移量相加得到.
- **PC 相对寻址 (PC-Relative Addressing):**
  条件分支指令在进行分支时使用 PC 相对寻址来确定 PC 的新值.
  **分支目标地址** (BTA, Branch Target Address) 与当前 PC 值相关.
  16 位立即数字段给出**分支目标地址**与**分支指令后一条指令** (即 `PC + 4` 处的指令) 之间的指令数.
  以下面的代码为例:
  (分支目标地址 `0xB4` 与 `PC+4 = 0XA8` 之间有 3 条指令，因此 `beq` 指令的立即数字段为 3)

```
0xA4    beq $t0, $0, else   # 分支指令
0xA8    addi $s0, $0, 1     # 分支指令后一条指令
0xAC    addi $s1, $0, 2
0xBO    addi $s2, $0, 3
0xB4 else:                  # 分支目标地址
        addi $s0, $0, -1
```

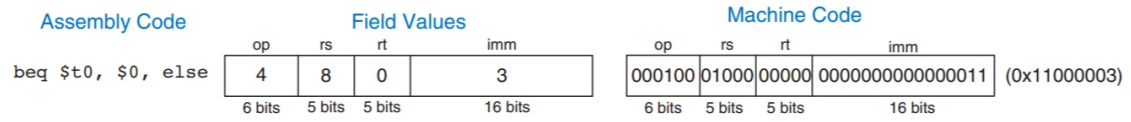



**Figure 6.28** Machine code for `beq`

因此计算**分支目标地址**的方法是:
符号扩展 16 位立即数并乘以 4 (单位从字转化为字节)，然后与 `PC+4` 相加.
我们再看一个例子:
(分支目标地址 `0x40` 与 `PC+4 = 0x58` 之间有 $-6$ 条指令，因此 `beq` 指令的立即数字段为 $-6$)

```
0x40 loop:                  # 分支目标地址
        add $t1, $a0, $s0
0x44    lb $t1, 0($t1)
0x48    add $t2, $a1, $s0
0x4C    sb $t1, 0($t2)
0x50    addi $s0, $s0, 1
0x54    bne $t1, $0, loop   # 分支指令
0x58    lw $s0, 0($sp)      # 分支指令后一条指令
```


**Figure 6.29** `bne` machine code

- **伪直接寻址 (Pseudo-Direct Addressing):**
  在**直接寻址** (direct addressing) 中 (例如 R 型指令 `jr rs` 跳转到 `rs` 保存的 32 位值),
  32 位**跳转目标地址** (JTA, Jumping Target Address) 在指令中是直接给出的.
  然而 J 型指令 ( `j` 和 `jal` ) 没有足够的位数 (只有 26 位) 来表示 32 位的跳转目标地址
  跳转目标地址 JTA 的最低两位 $\mathrm{JTA}_{1:0}$ 总是 0，因为指令是字对齐的;
  紧跟着 26 位 $\mathrm{JTA}_{27:2}$ 由指令的 `addr` 字段给出;
  最高 4 位 $\mathrm{JTA}_{31:28}$ 由 `PC+4` 的最高 4 位给出.
  这种寻址方式称为**伪直接寻址** (Pseudo-Direct Addressing)
  其跳转范围受限，为 $2^{26} \times 4\mathrm{KB} = 256\mathrm{MB}$，而直接寻址的跳转范围为 $2^{32}\mathrm{KB} = 4096\mathrm{MB}$

  反过来，在装入指令时，跳转目标地址的最高 4 位和最低 2 位被丢弃，剩下的 26 位存储于 `addr` 字段.
  以下面的代码为例:
  (跳转目标地址为 `0x004000A0`，掐头去尾得到 26 位 `addr` 字段 `0x0100028` )

```
0x0040005C      jal sum
    ...
0x004000A0 sum:
                add $v0, $a0, $a1
```

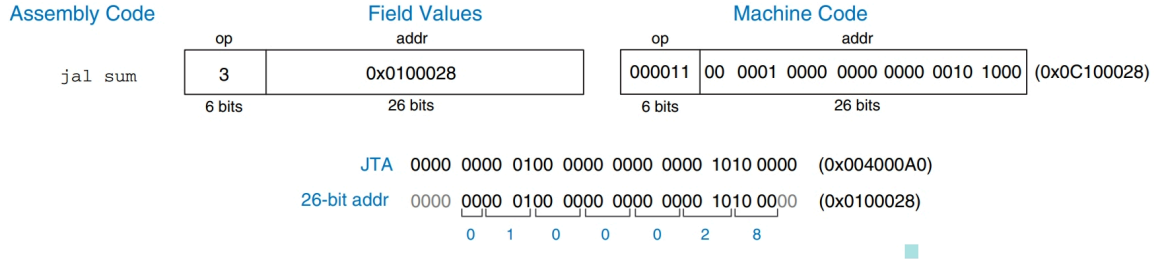



**Figure 6.30** `jal` machine code

**The End**