INTRODUCTION TO RISC-V

# RISC-V入门教程

ISA | 汇编指令 | 系统编程 | 组成原理 | 嵌入式应用

## 函数


主讲 邢建国



中国开放指令生态（RISC-V）联盟 | 浙江中心
China RICS-V Alliance | Zhejiang Center

浙江图灵算力研究院
ZHEJIANG TURING INSTITUTE

RISC-V®

# 函数

- 函数调用和返回
- 程序内存布局和堆栈
- ABI和函数参数传递、返回
- 实现RISC-V ilp32兼容的函数
- 函数示例

## 函数调用和返回

函数在汇编语言中由一个标号和一段代码定义，标号定义了函数的入口点。

调用函数只要跳转到其入口点即可。但是，在调用（跳转到）函数之前，保存返回地址，以便函数在执行后可以返回调用点。在RISC-V中，可以使用JAL指令来实现保存返回地址和跳转，通常返回地址放在寄存器ra中：JAL ra, func

函数定义

```
1 # The update_x routine
2 update_x:
3   la   t1, x
4   sw   a0, (t1)
5   ret
```

函数调用

```
1 .data
2 x: .skip 4
3
4   li  a0, 42     # loads 42 into a0
5   jal update_x   # invoke the update_x routine
```

注意：返回地址由指令自动存储在寄存器ra中。此操作破坏了寄存器ra先前的值，因此，在调用函数之前，可能需要保存此寄存器的内容，以便以后可以恢复。

## 函数调用和返回

函数调用几个问题：

如何保存返回地址？

jal指令，返回地址在ra寄存器中

如何向函数传递参数？

放在约定的寄存器中，如a0

如何从函数中返回值？

放在约定的寄存器中，如a0

如何返回调用点？

ret（jalr ra）

参数和返回值放在哪里是一个约定问题，通常由应用应用二进制接口或ABI定义。RISC-V ABI定义函数必须通过参数放在a0-a7中，将返回值存储在寄存器a0中，返回地址在ra中。

## 程序内存布局

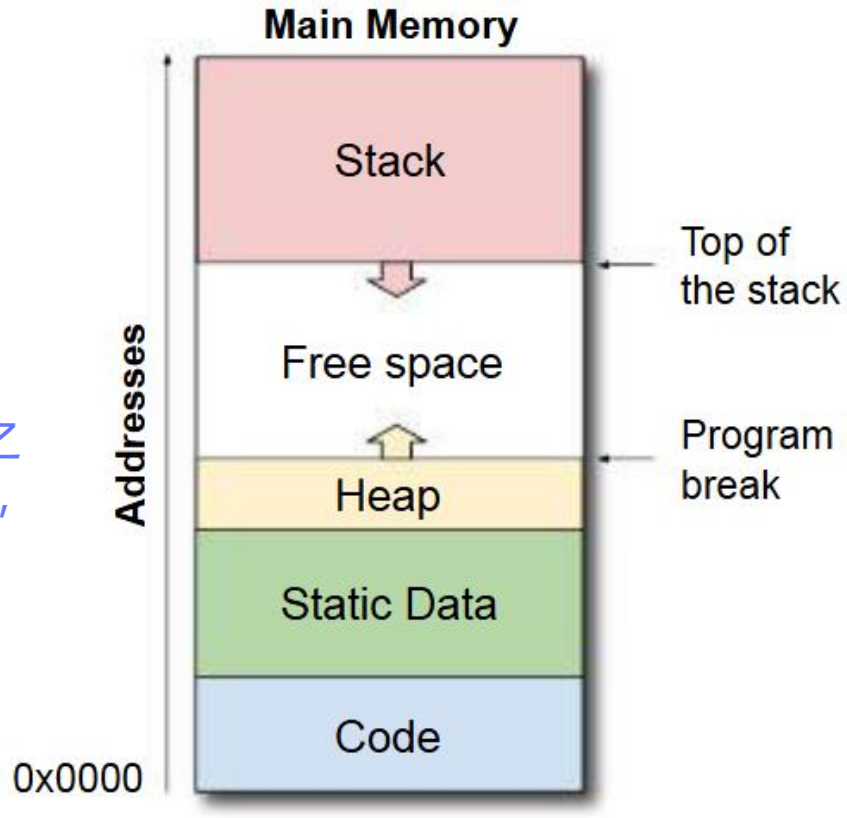
冯·诺依曼体系结构是一种将数据和代码存储在同一地址空间的计算机体系结构。大多数现代计算机体系结构都是冯·诺依曼体系结构。

代码空间：是存储程序代码的内存空间，通常首先放在最低的地址中。

静态数据空间：是存储程序静态数据（全局变量）并放在代码之后的内存空间。

堆空间：是由内存分配库1管理的内存空间，在静态数据之后分配。堆开始很小，每当内存分配函数需要更多空间时，它就调用操作系统来增加堆空间。

堆栈空间：是程序堆栈，通常放置在内存的末尾（高地址）。



## 程序堆栈

活动函数是被调用但尚未返回的函数。在执行的给定点可能有多个活动函数。

每当调用函数时，活动函数就会增加，而函数返回时就会减少。函数以先进后出的方式激活和停用，最后一个被激活的必须是第一个被停用的。

因此，跟踪活动函数的最自然的数据结构是堆栈。

```
1  int a = 10;
2
3  int main()
4  {
5      return bar() + 2;
6  }
7  int bar()
8  {
9      return fun() + 4;
10 }
11 int fun()
12 {
13     return a;
14 }
```

## 程序堆栈

函数通常需要内存空间来存储重要信息，例如局部变量、参数和返回地址。因此，每当调用函数（并变得活跃）时，系统都需要分配内存空间来存储与函数相关的信息。一旦它返回（停用），就不再需要与函数调用相关的所有信息，必须释放这些内存空间。

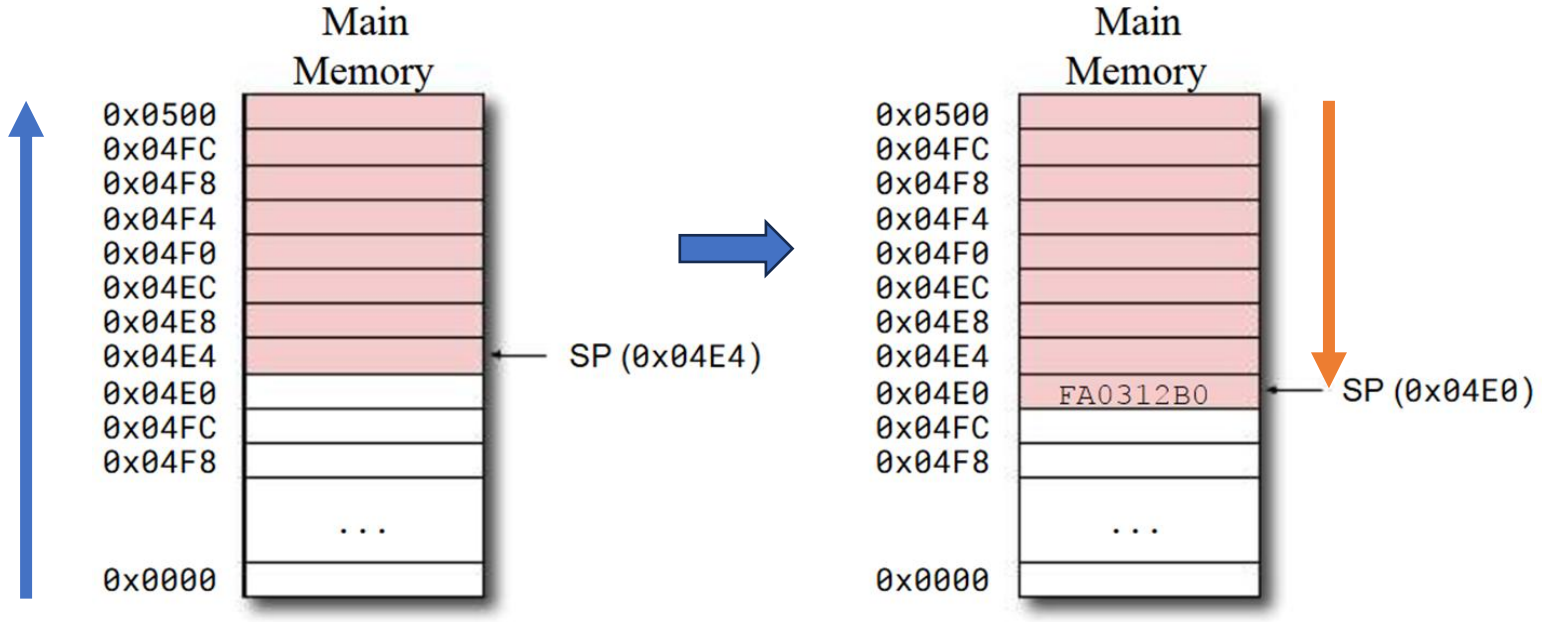
程序堆栈是一种堆栈数据结构，它存储有关活动函数的信息，例如局部变量、参数和返回地址。程序堆栈存储在内存中，每当调用函数时，有关函数的信息就会被推到堆栈顶部，使得栈增长。而当函数返回时，有关函数的信息就会通过出栈被丢弃，使得栈缩小。

## 程序堆栈

堆栈指针是指向堆栈顶部的指针，它存储堆栈顶部的地址。堆栈的增长或收缩是通过调整堆栈指针来执行的。

在RISC-V中，堆栈指针由寄存器sp存储。此外，在RISC-V中，堆栈向低地址增长，因此，堆栈的增长（或分配空间）可以通过减小寄存器（堆栈指针）的值来执行。

（a）堆栈向低地址增长
（b）堆栈指针指向压入堆栈中的最后一个元素



## 程序堆栈

下面的代码展示了如何将数据压入堆栈和弹出堆栈。

入栈：首先，减少堆栈指针sp来分配空间（4字节），然后，用sw指令将寄存器a0（4字节）的内容存储在堆栈的顶部

```
addi sp, sp, -4   # allocate stack space
sw   a0, 0(sp)    # store data into the stack
```

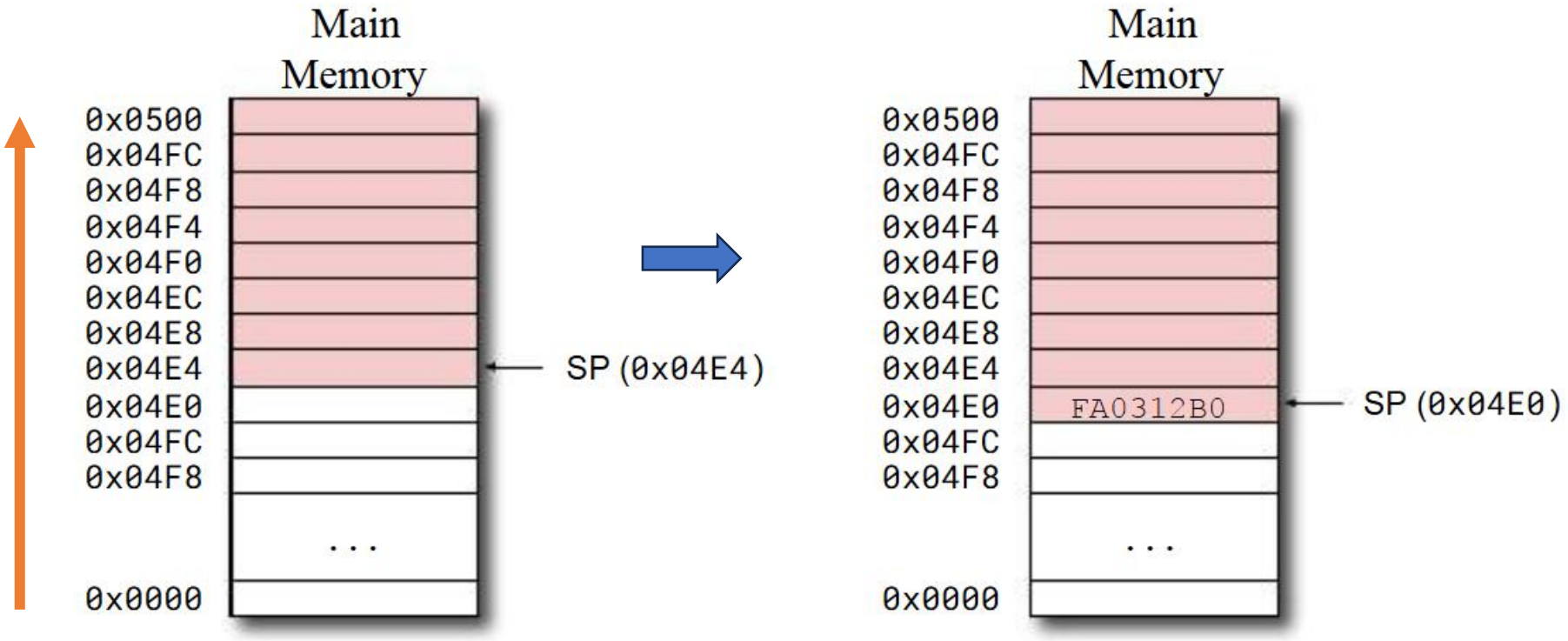
出栈：首先，使用lw指令将堆栈顶部的值加载到a0寄存器（4字节）中。然后，堆栈指针sp增加以释放空间（4字节）

```
lw   a0, 0(sp)    # retrieve data from stack
addi sp, sp, 4    # deallocate space
```

## 程序堆栈

入栈示例

```
1 li   a0, 0xFA0312B0
2 addi sp, sp, -4      # allocate stack space
3 sw   a0, 0(sp)       # store data into the stack
```



## 在堆栈中压入多个值

在许多情况下，程序可能需要将多个值推送或弹出到堆栈中或从堆栈中弹出。例如，程序可能需要在堆栈上保存一组寄存器值。

在这些情况下，可以通过仅调整（增加或减少）堆栈指针一次来优化代码。

下面的代码显示了如何将四个值从寄存器a0、a1、a2、a3中压入堆栈。堆栈指针只调整了一次，使用sw指令的立即数字段来选择适当的位置来存储每个值。压入堆栈中的最后一个值是存储在寄存器 a3中的值。

```
1 addi sp, sp, -16   # allocate stack space (4 words)
2 sw   a0, 12(sp)    # store the first value (SP+12)
3 sw   a1,  8(sp)    # store the second value (SP+8)
4 sw   a2,  4(sp)    # store the third value (SP+4)
5 sw   a3,  0(sp)    # store the fourth value (SP+0)
```

## 在堆栈中压入多个值

在许多情况下，程序可能需要将多个值推送或弹出到堆栈中或从堆栈中弹出。例如，程序可能需要在堆栈上保存一组寄存器值。

在这些情况下，可以通过仅调整（增加或减少）堆栈指针一次来优化代码。

下面的代码显示了如何将四个值从寄存器a0、a1、a2、a3中压入堆栈。堆栈指针只调整了一次，使用sw指令的立即数字段来选择适当的位置来存储每个值。压入堆栈中的最后一个值是存储在寄存器 a3中的值。

```
1 addi sp, sp, -16   # allocate stack space (4 words)
2 sw   a0, 12(sp)    # store the first value (SP+12)
3 sw   a1,  8(sp)    # store the second value (SP+8)
4 sw   a2,  4(sp)    # store the third value (SP+4)
5 sw   a3,  0(sp)    # store the fourth value (SP+0)
```

## 从堆栈中弹出多个值

```
addi  sp, sp, -32
sd    x5, 24(sp)
sd    x6, 16(sp)
sd    x7, 8(sp)
sd    x28, 0(sp)
...
ld    x28, 0(sp)
ld    x7, 8(sp)
ld    x6, 16(sp)
ld    x5, 24(sp)
addi  sp, sp, 32
```

## 堆栈指针初始化

在执行程序之前，堆栈指针寄存器必须初始化到程序堆栈的底部。

当在没有操作系统支持的情况下运行程序时（例如，在嵌入式系统中），堆栈指针必须由系统初始化代码初始化。

当在操作系统上运行程序时，执行环境（操作系统，例如内核）通常在跳转到程序入口点之前初始化堆栈指针。

在RARS中，堆栈指针在运行前被设置为0x7fffeffc（也可以自行设置）

## 函数调用

函数调用过程通常分为6个阶段：

1. 将参数存放到函数可访问的位置；
2. 跳转到函数入口（使用RV32I的 jal指令）；
3. 获取函数所需局部存储资源，按需保存寄存器；
4. 执行函数功能；
5. 将返回值存放到调用者可访问的位置，恢复寄存器，释放局部存储资源；
6. 由于程序可从多处调用函数，故需将控制权返回到调用点（使用 ret指令）。

# ■ ABI (Application Binary Interface)

ABI：应用程序二进制接口是为了方便软件的组合而定义的一组约定。

例如，ABI定义了调用约定，它决定了参数必须如何和在哪里传递给函数，以及函数必须如何和在哪里返回值。

假设一个名为jsort的函数，它接受两个参数：一个指向字符数组的指针和一个指示数组大小的整数。

```
void jsort(char* a, int n);
```

如果要调用该函数，可以通过执行一条jal jsort指令。

调用者应该将函数参数（指向数组a的指针和大小n）放在哪里?
这取决于函数jsort中的代码期望参数的位置。例如，如果函数期望将第一个参数放在寄存器a中，第二个参数放在寄存器中a1中，那么调用者必须将这两个参数放在这两个寄存器中，否则jsort将无法正常工作。

由ABI定义的调用约定定义了函数参数必须传递的位置，而调用者和函数开发者遵循相同的ABI。

## ■ ABI寄存器约定

| 寄存器 | ABI 名称 | 描述 | 调用前后是否一致? |
|---|---|---|---|
| x0 | zero | 硬连线为 0 | — |
| x1 | ra | 返回地址 | 否 |
| x2 | sp | 栈指针 | 是 |
| x3 | gp | 全局指针 | — |
| x4 | tp | 线程指针 | — |
| x5 | t0 | 临时寄存器/备用链接寄存器 | 否 |
| x6–7 | t1–2 | 临时寄存器 | 否 |
| x8 | s0/fp | 保存寄存器/帧指针 | 是 |
| x9 | s1 | 保存寄存器 | 是 |
| x10–11 | a0–1 | 函数参数/返回值 | 否 |
| x12–17 | a2–7 | 函数参数 | 否 |
| x18–27 | s2–11 | 保存寄存器 | 是 |
| x28–31 | t3–6 | 临时寄存器 | 否 |
| f0–7 | ft0–7 | 浮点临时寄存器 | 否 |
| f8–9 | fs0–1 | 浮点保存寄存器 | 是 |
| f10–11 | fa0–1 | 浮点参数/返回值 | 否 |
| f12–17 | fa2–7 | 浮点参数 | 否 |
| f18–27 | fs2–11 | 浮点保存寄存器 | 是 |
| f28–31 | ft8–11 | 浮点临时寄存器 | 否 |

# ■ ABI (Application Binary Interface)

一个计算机架构可能定义了多个不同的ABI。例如，x86就是这种情况，为不同的操作系统定义了不同的ABI。

RISC-V联盟定义了几个ABIs。除非另有说明，在本教程中，我们将使用RISC-V ilp32 ABI，它定义了int、long、pointer的大小为32位长，还定义了64位long long类型，8位的char，16位短
整数short。

注意：只有为同一ABI生成的代码才能通过链接器链接在一起。

当用GCC生成代码时，可以使用-mabi标志指定ABI。以下命令使用ABI ilp32来编译程序：

```
gcc-c Program.c-mabi=ilp32-o Program.o
```

## ■ 调用函数如何传递参数

调用函数时，参数可以放在寄存器中，也可以放在堆栈上。

RISC-V ilp32 ABI定义了一组将参数传递给寒暑假的约定。这些约定指定了如何在寄存器或堆栈上传递不同类型的值（char、integer、structs...）。下面以用32位或更少位表示的参数为例，这些参数可以是(unsigned）char、（unsigned）short、（unsigned ）int或指针类型。

函数的前8个参数通过寄存器a0-a7传递，即每个寄存器一个参数。如果函数的参数少于9，则所有参数都通过寄存器传递。小于32位（char、short）的参数扩展为32位。

如果超过8个参数，则剩余的参数9到参数N，将通过堆栈传递。在这种情况下，必须将参数压入程序堆栈中。最后一个参数N，必须首先压入，第9个参数必须最后压入。这些参数必须稍后通过将它们推送到堆栈的相同例程从堆栈中删除。同样，小于32位（例如char或short）的标量参数扩展为32位。

## 调用函数如何传递参数

在调用例程之前，调用方必须根据ABI设置参数。

假设函数sum10对传入的10个参数（int类型）求和并返回结果。

```
int sum10(int a, int b, int c, int d, int e,
          int f, int g, int h, int i, int j);
```

调用sum10时，前8个参数放在寄存器a0-a7中，第9个和第10个参数i、j放在堆栈上（先压入第10个参数j，再压入第9个参数i）。

```
# sum10(10,20,30,40,50,60,70,80,90,100);
main:
  li a0, 10          # 1st parameter
  li a1, 20          # 2nd parameter
  li a2, 30          # 3rd parameter
  li a3, 40          # 4th parameter
  li a4, 50          # 5th parameter
  li a5, 60          # 6th parameter
  li a6, 70          # 7th parameter
  li a7, 80          # 8th parameter
  addi sp, sp, -8    # Allocate stack space
  li t1, 100         # Push the 10th parameter
  sw t1, 4(sp)
  li t1, 90          # Push the 9th parameter
  sw t1, 0(sp)
  jal sum10          # Invoke sum10
  addi sp, sp, 8     # Deallocate the parameters from stack
  ret
```

## 调用函数如何传递参数

sum10函数中，从a0-a7得到前8个参数，从堆栈中得到后面两个参数：

```
# sum10(10,20,30,40,50,60,70,80,90,100);
main:
 li a0, 10          # 1st parameter
 li a1, 20          # 2nd parameter
 li a2, 30          # 3rd parameter
 li a3, 40          # 4th parameter
 li a4, 50          # 5th parameter
 li a5, 60          # 6th parameter
 li a6, 70          # 7th parameter
 li a7, 80          # 8th parameter
 addi sp, sp, -8    # Allocate stack space
 li t1, 100         # Push the 10th parameter
 sw t1, 4(sp)
 li t1, 90          # Push the 9th parameter
 sw t1, 0(sp)
 jal sum10          # Invoke sum10
 addi sp, sp, 8     # Deallocate the parameters from stack
 ret
```

```
sum10:
 lw  t1, 0(sp)    # Loads the 9th parameter into t1
 lw  t2, 4(sp)    # Loads the 10th parameter into t2
 add a0, a0, a1   # Sums all parameters
 add a0, a0, a2
 add a0, a0, a3
 add a0, a0, a4
 add a0, a0, a5
 add a0, a0, a6
 add a0, a0, a7
 add a0, a0, t1
 add a0, a0, t2   # Place return value on a0
 ret              # Returns
```

# 从函数中返回结果

RISC-V ilp32 ABI定义了应该在寄存器a0中返回值。如果返回的值为64位长，那么低32位放在寄存器a0中，高32位放在寄存器a1中。

```
1  sum10:
2   lw   t1, 0(sp)    # Loads the 9th parameter into t1
3   lw   t2, 4(sp)    # Loads the 10th parameter into t2
4   add  a0, a0, a1   # Sums all parameters
5   add  a0, a0, a2
6   add  a0, a0, a3
7   add  a0, a0, a4
8   add  a0, a0, a5
9   add  a0, a0, a6
10  add  a0, a0, a7
11  add  a0, a0, t1
12  add  a0, a0, t2   # Place return value on a0
13  ret               # Returns
```

## 值和引用

参数可以是值或对变量的引用。值参数是包含值本身的参数，值直接放置在寄存器或程序堆栈中。

以下代码显示了一个函数，该函数期望其参数按值传递。根据RISC-V ilp32 ABI调用约定，参数v需要通过寄存器a0传递。由于它是作为值传递的，因此寄存器将包含值本身。

```
1 int pow2(int v)
2 {
3     return v*v;
4 }
```

```
1 pow2:
2   mul a0, a0, a0 # a0 = a0 * a0
3   ret            # return
```

```
1 main:
2   li  a0, 32  # set the parameter with value 32
3   jal pow2    # invoke pow2
4   ret
```

## 值和引用

“引用”是变量的存储器地址（指针）。函数可以使用这个地址来读取或更新变量值。下面的代码显示了一个参数通过引用方法传递的函数。

根据RISC-V ilp32 ABI，这个参数要通过寄存器a0传入。由于它是作为引用传递的，寄存器a0将包含变量v的地址。

```
1 void inc(int* v)
2 {
3     *v = *v + 1;
4 }
```

```
1 inc:
2   lw   a1, (a0)  # a1 = *v
3   addi a1, a1, 1 # a1 = a1 + 1
4   sw   a1, (a0)  # *v = a1
5   ret
```

```
1 .data
2 y: .skip 4
3
4 .text
5 main:
6   la   a0, y  # set the parameter with the address of y
7   jal  inc    # invoke inc
8   ret
```

## 值和引用

引用参数可用于将信息传入和传出函数。

由于引用本质上是一个内存地址，因此传入或传出函数的信息必须位于内存中。

# 全局变量和局部变量

在高级语言中，例如“C”，全局变量是在函数之外声明的变量，可以在程序中的任何函数中访问。

全局变量由汇编器在静态数据空间（.data）上分配，通常借助汇编命令在汇编程序里声明。

```
1  int x;
2
3  int main()
4  {
5      return x+1;
6  }
```

```
1  .data
2  x:
3      .skip 4
4
5  .text
6  main:
7      la   a0, x       # Loads the address of variable x
8      lw   a0, 0(a0)   # Loads the value o x
9      addi a0, a0, 1   # Increments the value
10         ret              # Return
```

# 局部变量

在高级语言中，例如在“C”中，局部变量是在函数中声明的变量，只能在声明它的函数中使用。

理想情况下，局部变量应该在寄存器上分配。以下代码包含一个名为tmp的局部变量，可以使用寄存器a2来存储。

```
1 void exchange(int* a, int* b)
2 {
3     int tmp = *b;
4     *b = *a
5     *a = tmp;
6 }
```

```
1 exchange:
2   lw a2, (a1)    # tmp = *b
3   lw a3, (a0)    # a3 = *a
4   sw a3, (a1)    # *b = a3
5   sw a2, (a0)    # *a = tmp
6   ret
```

## 在内存中分配局部变量

有几种情况下必须在内存中分配局部变量，

- 当函数有很多局部变量，没有足够的寄存器来分配时；
- 当局部变量是数组或结构时；
- 当代码需要局部变量的地址时（当传递局部变量作为对其他函数的引用参数时可能会出现这种情况）。

在内存中分配的局部变量，每当调用函数就会在程序堆栈上分配，并在函数返回时释放。

这些变量必须由定义它们的函数分配和释放。它们必须在函数进入时分配，并在从函数返回之前释放。

程序堆栈的空间分配是通过将堆栈指针的值减少需要分配的字节数来执行的。

## 在内存中分配局部变量

```
1 int foo()
2 {
3     int userid;
4     get_uid(&userid);
5     return userid;
6 }
```

引用

```
1 foo:
2   addi sp, sp, -4      # Allocate userid
3   mv   a0, sp          # a0 = address of userid (&userid)
4   jal  get_uid         # Invoke the get_uid routine
5   lw   a0, (sp)        # a0 = userid
6   addi sp, sp, 4       # Deallocate userid
7   ret
```

```
1 int bar()
2 {
3     int my_array[8];
4     init_array(my_array);
5     return my_array[4];
6 }
```

数组

```
1 bar:
2   addi sp, sp, -32     # Allocate my_array
3   mv   a0, sp          # a0 = address of my_array
4   jal  init_array      # Invoke the init_array routine
5   lw   a0, 16(sp)      # Load my_array[4] into a0
6   addi sp, sp, 32      # Deallocate my_array
7   ret
```

## 在内存中分配局部变量

```
1  typedef struct
2  {
3    int year;
4    int month;
5    int day;
6  } date_t;
7
8  int get_current_day()
9  {
10     date_t d;
11     init_date(&d);
12     return d.day;
13 }
```

结构体

```
1  get_current_day:
2    addi sp, sp, -12    # Allocate d
3    mv   a0, sp         # a0 = address of d
4    jal  init_date      # Invoke the init_date routine
5    lw   a0, 8(sp)      # Load d.day into a0
6    addi sp, sp, 12     # Deallocate d
7    ret
```

# 寄存器使用策略

在程序中经常使用寄存器来保存变量和临时值，从函数返回值，并将参数传递给函数。事实上，寄存器是非常宝贵的资源，由程序的不同部分共享。

因此在使用寄存器之前，可能需要将其内容保存到内存中，以便以后可以恢复。

```
1 mix:
2   lw  a2, (a0)    # load important information into a2
3   la  a0, x       # Sets parameter 0 with address of var. x
4   la  a1, y       # Sets parameter 1 with address of var. y
5   jal exchange    # Invokes exchange to swap x an y values
6   mv  a0, a2      # Move important information into a0 to return
7   ret
```

a2内容被修改

```
1 exchange:
2   lw a2, (a1)     # tmp = *b
3   lw a3, (a0)     # a3 = *a
4   sw a3, (a1)     # *b = a3
5   sw a2, (a0)     # *a = tmp
6   ret
```

X

## ■ 寄存器使用策略

因此在使用寄存器之前，可能需要将其内容保存到内存中，以便以后可以恢复。

调用者保存

保存

恢复

```
1  mix:
2    lw   a2, (a0)    # load important information into a2
3    addi sp, sp, -4  # Saves a2: Allocate stack space
4    sw   a2, (sp)    #             Store a2 into the stack
5    la   a0, x       # Sets parameter 1 with address of var. x
6    la   a1, y       # Sets parameter 1 with address of var. y
7    jal  exchange    # Invokes exchange to swap x an y values
8    lw   a2, (sp)    # Restores a2: Loads a2 from the stack
9    addi sp, sp,  4  #                Deallocate the stack space
10   mv   a0, a2      # Move important information into a0 to return
11   ret
```

## ■ 寄存器使用策略

因此在使用寄存器之前，可能需要将其内容保存到内存中，以便以后可以恢复。

被调用者保存

保存

恢复

```
1  exchange:
2    addi sp, sp, -8 # Allocate stack space
3    sw   a2, 4(sp)  # Save contents of a2
4    sw   a3, 0(sp)  # Save contents of a3
5    lw   a2, (a1)   # tmp = *b
6    lw   a3, (a0)   # a3 = *a
7    sw   a3, (a1)   # *b = a3
8    sw   a2, (a0)   # *a = tmp
9    lw   a3, 0(sp)  # Restore contents of a3
10   lw   a2, 4(sp)  # Restore contents of a2
11   addi sp, sp, 8  # Deallocate stack space
12   ret
```

## ■ 调用方保存 vs. 被调用方保存

ABI定义了在调用函数时，哪些寄存器必须由调用方保存，以及哪些寄存器必须由被调用方。

RISC-V ilp32 ABI定义了寄存器t0-t6、a0-a7、ra由调用方保存，寄存器s0-s11由被调用方保存。

保存

恢复

```
1  mix:
2    lw   a2, (a0)    # load important information into a2
3    addi sp, sp, -4  # Saves a2: Allocate stack space
4    sw   a2, (sp)    #              Store a2 into the stack
5    la   a0, x       # Sets parameter 1 with address of var. x
6    la   a1, y       # Sets parameter 1 with address of var. y
7    jal  exchange    # Invokes exchange to swap x an y values
8    lw   a2, (sp)    # Restores a2: Loads a2 from the stack
9    addi sp, sp,  4  #                 Deallocate the stack space
10   mv   a0, a2      # Move important information into a0 to return
11   ret
```

在调用函数之前，只需要保存那些可能被被调用方破坏的寄存器。

# 保存和恢复返回地址

每当调用函数时，返回地址都存储在返回地址寄存器ra中。每次调用函数，ra都会更新一个新值，并且它以前的内容会被破坏。因此，如果调用函数之后还需要寄存器ra的内容，那么在函数调用前必须要先保存寄存器ra，调用结束后再恢复。

因为调用函数的代码通常属于另一个函数，因此，它可能需要返回地址才能将其执行返回给调用方。

保存

请注意，伪指令ret使用寄存器ra的值以返回到正确的位置。

调用方函数负责保存寄存器ra的内容

恢复

```
1  mix:
2    addi sp, sp, -4 # Saves ra: Allocate stack space
3    sw   ra, (sp)   #            Store ra into the stack
4    lw   a2, (a0)   # load important information into a2
5    addi sp, sp, -4 # Saves a2: Allocate stack space
6    sw   a2, (sp)   #            Store a2 into the stack
7    la   a0, x      # Sets parameter 1 with address of var. x
8    la   a1, y      # Sets parameter 1 with address of var. y
9    jal  exchange   # Invokes exchange to swap x an y values
10   lw   a2, (sp)   # Restores a2: Loads a2 from the stack
11   addi sp, sp,  4 #                Deallocate the stack space
12   mv   a0, a2     # Move important information into a0 to return
13   lw   ra, (sp)   # Restores ra: Loads ra from the stack
14   addi sp, sp,  4 #                Deallocate the stack space
15   ret
```

## ■ 寄存器使用策略

叶子函数是不调用其他例程的例程。

由于它们不调用其他函数，寄存器的内容不会被修改。因此，在实现叶子函数时，不需要将寄存器ra保存在堆栈上。

标准ABI规定函数不应修改寄存器tp和gp，因为汇编器可能依赖于它们的值。

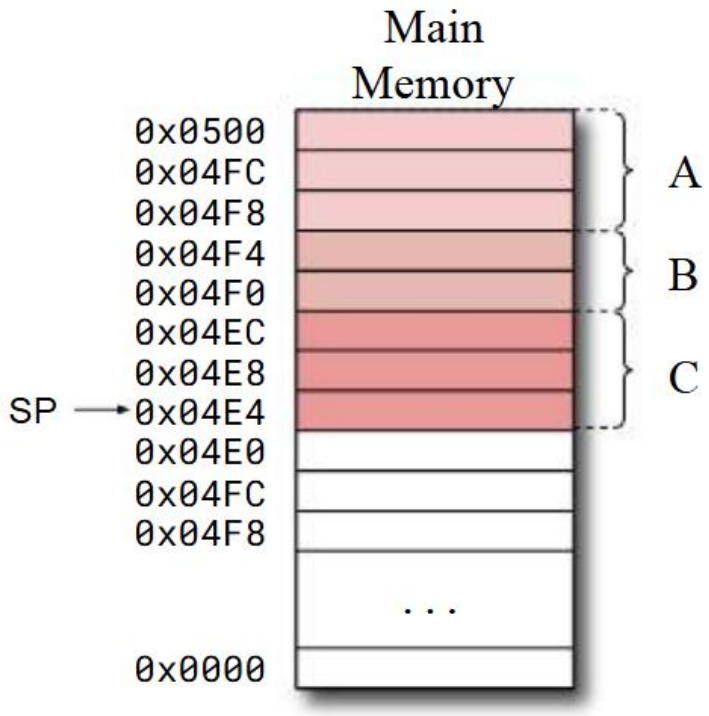
## ■ 堆栈帧(stack frame)和帧指针(frame pointer)

所有活动的函数都可能包含程序堆栈中的信息。此外，这些信息自然地根据调用顺序进行分组。

例如，让我们假设函数A调用函数B，函数B调用函数C，函数C正在执行。请注意，此时函数A、B仍然是活动的。堆栈上函数A添加的内容放在函数B的内容之前。函数B添加的内容放在函数C的内容之前。

堆栈帧是程序堆栈上存储活动函数信息的连续数据段。

在前面的例子中，在函数C执行时有三个堆栈帧。
函数A的堆栈帧位于内存区间0x0500-0x04F8。



## ■ 帧指针(frame pointer)

每当向程序堆栈添加新信息时，堆栈指针都会调整。

那么如何引用之前函数栈帧数据?

如下面函数addijx，它接受十个参数，调用函数get_x，将get_x返回值与参数9、10的和作为结果返回。

```
int addijx(int a, int b, int c, int d, int e,
           int f, int g, int h, int i, int j)
{
    return get_x() + i + j;
}
```

```
addijx:
  addi sp, sp, -4 # Saves the
  sw   ra, (sp)   #  return address
  jal  get_x      # Invoke the get_x routine
  lw   a1, 4(sp)  # Loads i from the program stack
  lw   a2, 8(sp)  # Loads j from the program stack
  add  a0, a1, a1 # a0 = get_x() + i
  add  a0, a2, a2 # a0 = get_x() + i + j
  lw   ra, (sp)   # Restore the
  addi sp, sp, 4  #  return address
  ret             # Returns
```

## ■ 帧指针(frame pointer)

注意，返回地址保存到（第2行和第3行）并从（第9行和第10行）程序堆栈恢复。在入口点，堆栈指针指向第9个参数。

但是，在返回地址压入堆栈后，堆栈指针sp指向返回地址。因此，要访问这一点之后的参数9，代码必须在当前堆栈指针上添加4

```
1  addijx:
2    addi sp, sp, -4 # Saves the
3    sw   ra, (sp)   #  return address
4    jal  get_x      # Invoke the get_x routine
5    lw   a1, 4(sp)  # Loads i from the program stack
6    lw   a2, 8(sp)  # Loads j from the program stack
7    add  a0, a1, a1 # a0 = get_x() + i
8    add  a0, a2, a2 # a0 = get_x() + i + j
9    lw   ra, (sp)   # Restore the
10   addi sp, sp, 4  #  return address
11   ret              # Returns
```

## ■ 帧指针(frame pointer)

添加到堆栈中的信息越多，就越难跟踪整个例程中所有参数和局部变量的地址。

解决这个问题的一种方法是保留一个指向堆栈的固定指针，以便可以使用这个指针加上一个固定的函数集访问所有参数和局部变量。

帧指针指向当前执行例程的堆栈帧的开头。它在函数的执行过程中提供了一个指向堆栈的固定指针，并且可以用作访问参数和局部变量的固定引用。

在RISC-V ilp32 ABI中，帧指针为帧指针寄存器fp。寄存器fp必须在函数开始时初始化，但是，必须保存其先前的内容，以便在从函数返回之前恢复。

此外，可以在函数开始时分配栈空间，并在返回前释放栈空间。

## ■ 帧指针(frame pointer)

分配栈空间
保存ra
保存调用函数的fp
调整fp指向调用函数栈顶

恢复fp
恢复ra
释放栈空间

```
1  addijx:
2    addi sp, sp, -8 # Allocates the stack frame
3    sw   ra, 4(sp)  # Saves return address
4    sw   fp, 0(sp)  # Saves previous frame pointer
5    addi fp, sp, 8  # Adjust frame pointer.
6
7    jal  get_x      # Invoke the get_x routine
8    lw   a1,  (fp)  # Loads i from the program stack
9    lw   a2, 4(fp)  # Loads j from the program stack
10   add  a0, a1, a1 # a0 = get_x() + i
11   add  a0, a2, a2 # a0 = get_x() + i + j
12
13   lw   fp, 0(sp)  # Restore previous frame pointer
14   lw   ra, 4(sp)  # Restore return address
15   addi sp, sp, 8  # Deallocate the stack frame
16   ret              # Returns
```

## 保持堆栈指针对齐

RISC-V ilp32 ABI指定堆栈指针在函数进入时始终与128位（16字节）边界对齐。

标准ABI中，堆栈指针必须在整个过程执行过程中保持对齐。非标准ABI代码必须在调用标准ABI过程之前需要重新对齐堆栈指针。

确保堆栈指针在整个函数执行过程中始终对齐的一种方法是始终以16的倍数增加和减少它。因此程序员（或编译器）应使用16字节的倍数来分配堆栈帧。

## ■ 实现RISC-V ilp32兼容的函数

- 包含一个标号来定义函数入口点。将C代码转换为汇编代码时，标号必须和C函数名相同；
- 使用返回指令ret从函数返回。该指令跳转到存储在返回地址寄存器ra的地址；
- 参数必须根据RISC-V ilp32 ABI进行访问。参数要小于或等于32位，前八个参数放在在寄存器a0-a7中，其余的在堆栈上。小于32位参数要根据其类型扩展为32位；
- 传递到堆栈上的参数被组织成最后一个参数，即第N个参数，首先被压入堆栈，而第9个参数最后压入堆栈。在函数入口时，堆栈指针指向第9个参数，sp+4指向第10个参数，依此类推。参数由调用方在堆栈上分配，也必须由调用方函数释放。被调用方不得释放调用方分配的参数；
- 如果函数需要在程序堆栈上存储信息，应该在函数的开头分配一个堆栈帧，并在返回之前释放。堆栈帧的大小必须是16的倍数，以确保堆栈指针保持与128位边界对齐，这是标准ABI的要求；

# ■ 实现RISC-V ilp32兼容的函数

函数可以使用寄存器，但是，被函数修改的被调用方保存的寄存器必须保存在函数的开头，并在从函数返回之前恢复。这些寄存器必须保存在堆栈帧中；

被调用方可以修改和使用调用方保存的寄存器而不保存它们，调用方必须在调用点之前（之后）保存（恢复）其内容。调用其他函数的函数（非叶子函数）必须保存和恢复返回地址寄存器ra。这些寄存器必须保存在栈帧上；

局部变量可以分配在寄存器或内存中。需要在内存中分配的局部变量必须在栈帧上分配；

可选地，帧指针寄存器fp可用于保留指向栈帧开头的指针，并提供对调用参数和局部变量的固定引用。在这种情况下，从函数返回时必须恢复前一个帧指针。帧指针fp的内容必须保存在函数开头的栈帧中，并在返回前恢复。

标准ABI规定例程不应修改寄存器tp、gp。

## 示例：递归函数

递归函数是自己调用自己的函数。

```
1 int factorial(int n)
2 {
3   if (n>1)
4     return n * factorial(n-1);
5   else
6     return 1;
7 }
```

```
1  factorial:
2    addi sp, sp, -16    # Allocates the routine frame
3    sw   ra, 0(sp)      # Saves the return address
4    li   a1, 1
5    ble  a0, a1, else   # if (n>1)
6    sw   a0, 4(sp)      # Saves n (a0) on the routine frame
7    addi a0, a0, -1     # Set the parameter (n-1)
8    jal  factorial      # Perform the recursive call
9    lw   a1, 4(sp)      # Loads n from the routine frame (into a1)
10   mul  a0, a0, a1     # a0 = factorial(n-1) * n
11   j    fact_end       # Jumps to end
12 else:
13   li   a0, 1          # Set the return value to 1
14 fact_end:
15   lw   ra, 0(sp)      # Restores the return addres
16   addi sp, sp, 16     # Deallocate the routine frame
17   ret                 # Return
```

## ■ 示例：标准的“C”库系统调用函数

用户程序通常调用操作系统服务来执行输入和输出操作。这种操作称为系统调用(syscall)

在RISC-V中通过执行指令ecall来实现。程序调用必须用正确的系统调用号设置寄存器a7。标准的“C”库提供函数来封装系统调用。

```
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

```
write:
  addi sp, sp, -16   # Allocates the stack frame
  sw   ra, 12(sp)    # Saves the return address
  li   a7, 64        # Sets the syscall code (64 = write)
  ecall              # Invokes the operating system
  lw   ra, 12(sp)    # Restores the return address
  addi sp, sp, 16    # Deallocates the stack frame
  ret                # Returns
```

RISC-V®

# 函数

- 函数调用和返回
- 程序内存布局和堆栈
- ABI和函数参数传递、返回
- 实现RISC-V ilp32兼容的函数
- 函数示例