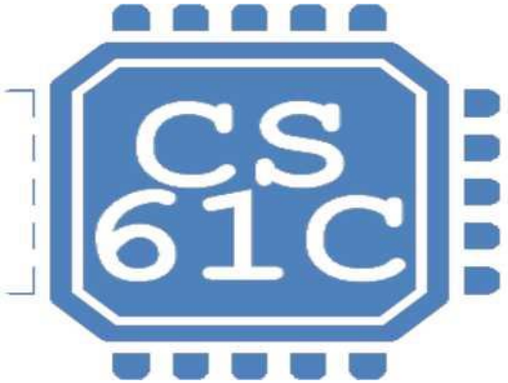


Great Ideas in Computer Architecture

更多 RISC-V、RISC-V 功能

讲师：肖恩·法哈

高风险

面积

上堂课复习（1/2）

- RISC 设计原则
  — 越小越快：32 个寄存器，指令更少
  - 保持简单：严格语法
- RISC-V 寄存器：s0-s11、t0-t6、x0
  - 没有数据类型，只有原始位，操作决定如何解释它们
- 内存为字节寻址

- 无类型 ^ 无自动指针算法

上堂课复习（2/2）

- RISC-V 指令
  - 算术：add、sub、addi、mult、div
  - 数据传输：lw、sw、lb、sb、lbu
  - 分支：beq, bne, bge, blt, jal, j, jalr, jr;
  - 按位：和、或、xor、andi、ori、xori
  - 换档：sll、srl、sra、slli、srli、srai

i ="立即"（常量整数）

RISC-V 议程

- **标志扩展练习**
- 伪指令
- C 到 RISC-V 实践
- 程序集中的函数
- 函数调用约定

符号 2 的补码

- 我们如何知道二进制二的补码数是否是负的?

符号 2 的补码

- 我们如何知道二进制二的补码数是否是负的?

二进制：0b10000010 截止日期 0b11110000

十六进制：0x82 0x7F 0xF0

符号 2 的补码

- 我们如何知道二进制二的补码数是否是负的?
  — 查看最重要的位置!

二进制：0b10000010 截止日期 0b11110000

十六进制：	0x7F	0xF0
0x82	阳性	阴性

标志扩展

- 如果我们想取一个 8 位二进制补码数，并使其成为 9 位数字，我们如何做到这一点?

0b0000 0010（+2）-> 0b0 0000 0010（2）

0b1111 1110（-2）-> 0b1 1111 1110（-2）

我们复制了最重要的部分!

算术符号扩展

计算时，立即数是符号扩展的 addi t0, x0,-1 == addi t0, x0, OxFFF

tO -> [-1] -> [OxFFFFFFFF]

为什么在这些 addi tO、xO、OxOFF tO -> [OxOOOOOOOFF] 指令中，我们只对立即数使用 12 位?查找下节课结束!

addi tO, xO, OxF77 tO -> [OxFFFFFF77]

- 对于汇编，当我们从内存中取出数据时，会发生这种情况

正在加载标志扩展

- 内存中的字节：

  0b11111110（-2）

- 加载字节-> 寄存器内容：

  0 b XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX 1111 1110

  我们如何处理 X 值?

- 对于汇编，当我们将数据从内存中取出时，就会发生这种情况

正在加载标志扩展

- 内存中的字节：
  0b11111110（-2）
- 加载字节-> 寄存器内容：
  0b 11111111111111111111111111111111

  我们如何处理 X 值?**签署扩展!**

正常（有符号）加载符号扩展最高有效位内存：0b1000 1111

正在加载标志扩展

加载字节-> | 0b1111 1111 1111 1111 1111 1111 1000 1111

内存：0b0000 1111

加载字节-> | 0b0000 0000 0000 0000 0000 0000 0000 1111

偏移荷载也表示延伸：

内存 = [0x00008011]（s0 中的地址）

假设系统是小端

正在加载标志扩展

磅 t0, 0（s0）-> 加载 0b00010001

0b0000 0000 0000 0000 0000 0000 0001 0001

磅 t0,1（s0）-> 加载 0b10000000

0b1111 1111 1111 1111 1111 1111 1111 1000 0000

无符号载荷不带符号扩展，而是用零填充：

内存 = [0x00008011]（s0 中的地址）

假设系统是小端

正在加载标志扩展

lbu t0,1（s0）-> 正在加载 0b10000000

l0b0000 0000 0000 0000 0000 0000 1000 0000 0000 0000

- 标志扩展练习
- **伪指令**
- C 到 RISC-V 实践
- 程序集中的函数
- 函数调用约定

组装说明

- 一种低级程序设计语言，其程序指令与特定体系结构的操作相匹配。
  — 代码可以编译为不同的汇编语言，但汇编语言只能在支持它的硬件上运行

- 但是，有时候，为了程序员的利益，提供硬件没有真正实现的附加指令是有用的
  — 翻译成真实的指令

- 示例：　　`mv dst,reg1`

  转换为 addi dst,reg1,0
  - 立即加载（li）

更多伪指令

- `li dst, imm`
- 将 32 位立即数加载到 dst
- 利用：addi、lui

- 加载地址（la）
  - `la dst,标签`
  - 将指定标签的地址加载到 dst
  - 转换为：AUIPCDST,<offset to label>
- 无操作（nop）
  - 诺普
  - 不执行任何操作
  - 转换为：addi x0, x0, 0

伪指令非常有用

- 甚至 j 指令实际上也是一个伪指令——我们将在这节课的后面看到它转换成什么
- 伪指令是编写 RISC 汇编代码的核心，在您阅读的任何 RISC 汇编代码中都会看到它们

支持 RISC-V 的伪指令的完整列表位于绿表中

RISC-V 议程

- 标志扩展练习
- 伪指令
- **C 到 RISC-V 实践**
- 程序集中的函数
- 函数调用约定

C 到 RISC-V 实践

- 让我们将所有新的 RISC-V 知识用在一个示例中：“快速字符串复制”
- C 代码如下：

```
/* 将字符串从 p 复制到 q */
字符 *p, *q;
while ( (*q++ = *p++) != '\0' );
```

- 我们对其结构了解多少?
  - 单 while 循环
  - 退出条件是等同性测试

C 到 RISC-V 实践

- 从代码框架开始：

```
# 将字符串 p 复制到 q
# ——sQ,q^——si（字符*指针）
循环：
                                    # tQ = *p
                                    # *q = tQ
                                    # p = p + 1
                                    # q = q + 1
                                    # 如果*p==Q,转到退出 j 循环
        R 转到循环
```

退出：

C 到 RISC-V 实践

- 成品代码：

```
# 将字符串 p 复制到 q
# ——sQ,q^——si（字符*指针）
循环：  磅 tQ,Q（sQ）
        sb tQ,Q（s1）
        addi sQ,sQ,1
        addi si,si,1
        beq tQ,xQ,Exit
        j Loop
```

```
# tQ = *p
# *|q = tQ
# p = p + 1
# q = q + 1
# 如果 *p==Q,则退出
继续
# 转到循环
```

退出：# p => N*6 指令中的 N 个字符

C 到 RISC-V 实践

- 成品代码：

```
# copy String p to
q # p >s0, q >s1
 (char**)
循环：lb* t0,0(s0)
      sb t0,0(s1)
     addi s0,s0,1
     addi s1,s1,1
     beq t0,x0,Exit
     j 循环
退出：#N 字符，p = 2020
```

如果 lb 符号
^^extend 扩展怎么办?

指针）

不成问题，因为某人只写一个字节。

（符号扩展名被忽略）

=> N*6 条指令

C 到 RISC-V 实践

- 使用 bne 的备用代码：

```
# 将字符串 p 复制到 q
# ——sQ,q^——si（字符*指针）
回路：lb tQ,Q（sQ）I # tQ = *p
      sb tG,G（s1）l # *q I= tQ
```

```
addi sQ,sQ,1 #p =          p+1
addi s1,s1,1B E K =        q+1
bne tQ,xQ,LoopB # if
```

退出：p 中的 N 个字符数 => N*5 ii *p一: Q,转到循环指令

问题：哪个 C 代码可以正确填写下面的空白？

做 --;} while (_);

```
循                                编i^s0,j^s1
addi   s0,s0,-1                   编i = i - 1
索尔   t0,s1,2                    编t0 = (j < 2)
贝恩   t0,x0 循环                   # goto 循环（如果 t0!=0）
slt    t0,s1,s0                   编t0 = (j < i)
贝恩   t0,x0,循环                  编如果 t0!=0,则 goto 循环
```

(A) `j <2 || j < i`

(B) `j >2 && j < i`

(C) `j <2 || j > i`

(D) `j <2 && j > i`

问题：哪个 C 代码可以正确填写下面的空白？

做 --;} while (_);

循环:

```
addi   s0,s0,-1      编i^s0,j^s1
索尔蒂 t0,s1,2       编i = i - 1
贝恩   t0,x0 循环    编t0 = (j < 2)
slt    t0,s1,s0        # goto 循环（如果
贝恩   t0,x0,循环    编t0 = (j < i)
                     编如果 t0!=0,则 goto 循
```

(A) j <2 || j < i

(B) j >2 && j < i

(C) j <2 || j > i

(D) j <2 && j > i

RISC-V 议程

- 标志扩展练习
- 伪指令
- C 到 RISC-V 实践
- **程序集中的函数**
- 函数调用约定

函数调用的六个步骤

1. 将参数放在函数可以访问它们的位置
2. 将控制权转移给功能

3. 该功能将获取所需的任何（本地）存储资源
4. 该功能执行其所需任务
5. 该函数将返回值放在可访问的位置并“清除”
6. 控制归还给您

1 和 5：我们应该把论点放在哪里和返回值?

- 寄存器的速度比内存快得多，因此请尽可能使用它们
- a0-a7：传递参数的八个自变量寄存器
- a0-a1：两个参数寄存器也用于返回值
  - 争论的顺序很重要
  - 如果需要额外空间，请使用内存（堆栈!）

示例：程序集中的函数

空干线（void）{
a = 3;
b=a+1;
a = add
（a,b）;

主
峰：
```
addi a0, x0, 3
addi a1, a0, 1
jal ra, add
```

整数
```
add（int a, int
b）{ return a+b;
```

添加
```
加 a0,a0,a1 jr
ra
```

更多寄存器

- a0-a7：传递参数的八个自变量寄存器
- a0-a1：两个寄存器返回值
- sp：“堆栈指针”

— 保存堆栈“底部”的当前内存地址

2 和 6：如何进行转移控制?

- 跳跃（j）
  — j 标签
- Jump and Link（jal）*
  - jal dst 标签用于调用
- 跳转和链接寄存器（jalr）功能
  —jalr dst src imm
- “和链接”：在跳转之前保存指令在寄存器中的位置
- 跳转寄存器（jr）用于从函数返回
  — jr src（src = ra）
- ra = 返回地址寄存器，用于保存调用函数的位置，以便返回

函数调用示例

```
...和（a,b）;                          /* a—sQ,b-s1 */
int sum（intx,inty）{ 返回 x+y ;
}
```

RISC-V

地址（十进制）

```
第  1   addi aQ,sQ,Q
第  1   addi a1,s1,Q
2008    addi ra,xQ,1Q16
2012    I    总和，/
...
2016 我们以前会知道吗
...  编译?
2QQQ    总    添加 aQ,aQ,a1
第  2   焦耳  ra  z
```

```
# x = a
# y = b
ra 数量=2016 年
第 1 季度
# 跳转求和
```

否则我们不知道我们在哪里

```
Z#返回
```

来自

函数调用示例

```
...和（a,b）;.../* a—sQ,b-s1 *
int sum（intx,inty）{ 返回 x+y ;
}
第 2 季度第 4                    #返回
季度
第 1 季 addi aQ,sQ,Q          编 x = a
第  1 addi a1,s1,Q          编号 y = b
2008  贾尔 拉萨姆             编号 ra=12 季度，转
2012
```

地址（十进制）

```
2QQQ 总和：添加 vQ,aQ,a1
```

第/页

C RISC-V

总和

J 是解释的伪指令

- `jal` 语法：`jal dst label`
- 您提供了用于链接的寄存器
  — 调用函数时，使用 ra
- 如果指定 x0 会发生什么？
  - `jal x0` 标签
  - x0 始终包含 0,因此尝试写入时不执行任何操作
  — 所以 jal x0 标签只是跳转，没有链接
- j 标签是 jal x0 标签的伪指令
  — 类似地，jr 是遵循相同思想的 jalr 的伪指令

审查问题

RET 是可用于从函数返回的伪代码指令。您将使用哪条实际指令创建 ret?

描述：PC = R[1]

[A] jal x0,ra

[B] beq x0,x0,ra

[c]jalr x0,ra,0

杰拉

[E] jalr ra,ra,0

审查问题

RET 是可用于从函数返回的伪代码指令。您将使用哪条实际指令创建 ret?

描述：PC = R[1]

[A] 无效语法

[B] Invalid Syntax

X

[c]jalr x0,ra,0

[D] Invalid Syntax

将正确返回，但执行此操作后[E]会覆盖 ra

3: 变量的本地存储

- 堆栈指针（sp）保存堆栈底部的地址

— 递减（召回堆栈向下增长）
— 然后使用存储字写入变量
— 要“清理”，只需增加堆栈指针

将 t0 存储到堆栈
附加 `sp, sp, -4 sw t0, 0（sp）`

RISC-V 议程

- 标志扩展练习
- 伪指令
- C 到 RISC-V 实践
- 程序集中的函数
- **函数调用约定**

函数调用的六个步骤

*将参数放在函数可以**访问**的位置*

将控制权转移给功能

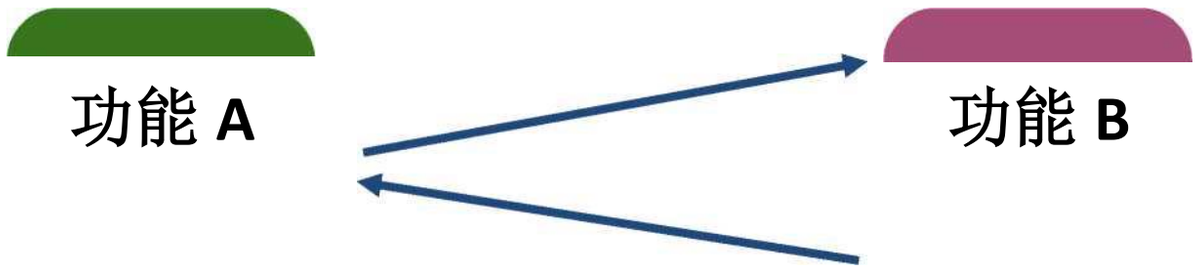
该功能将获取所需的任何（本地）存储资源

4.该功能执行其所需任务

该函数将返回值放在可访问的位置并“清除”，将质控品返还给您

我们可以使用哪些寄存器?

- 问题： 函数如何知道哪些寄存器可以安全使用?



函数 A 可能在调用函数 B 时使用了 t0

示例： sumSquare

```
int sumSquare(int x, int y) { 返回 mult
    (x,x) + y;}
```

- 我们需要节省什么?
  - — 调用 mult 将覆盖 ra,因此保存它
  - — 重用 al 将第 2 个参数传递给 mult,但以后需要当前值（y），因此保存 al

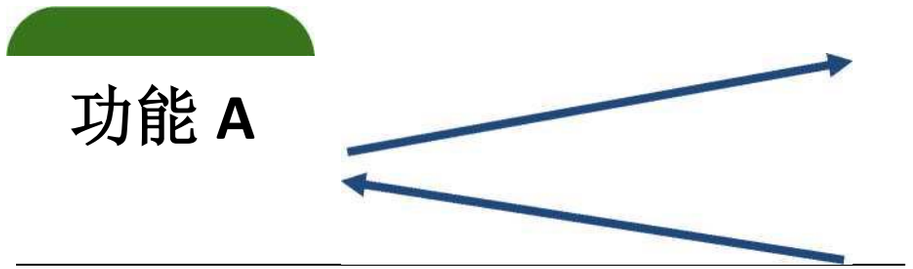
调用约定

- **CalleR：调用函数**
- **CalleE：被调用的函数**

- 寄存器惯例：一组公认的规则，规定在过程调用（jal）

后，哪些寄存器不会改变，哪些寄存器可能已经改变。

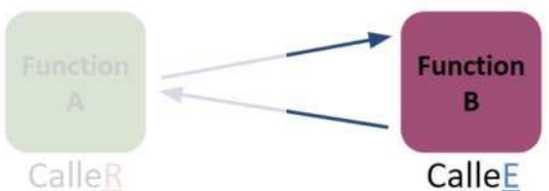
呼叫者和被呼叫者




```
void 函数 A（void）{ // 执行填充函数 B
        （void）;// 多做材料返回;}
```

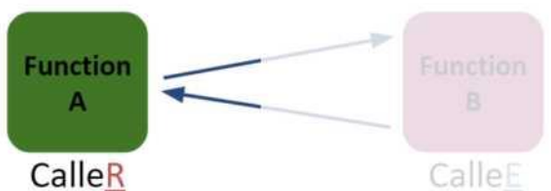
保存的寄存器（被叫方保存）

- 函数调用前后，这些寄存器应相同
  —如果 cal leE 使用它们，则必须在返回之前恢复值
  —这意味着保存旧值，使用寄存器，然后将旧值重新加载到寄存器中
- `s0-s11`（保存的寄存器）
- `sp`（堆栈指针）
  - 如果不在同一位置，调用方将无法正确访问自己的堆栈变量



易失性寄存器（呼叫者已保存）

- 这些寄存器可由调用者自由更改
  — 如果 calleR 需要这些值，则必须在调用过程之前保存这些值
- `t0-t6`（临时寄存器）
- `a0-a7`（返回地址和参数）
- `ra`（返回地址）
  — 如果 calleE 调用另一个函数（嵌套函数意味着 calleE 也是一个 calleR），则这些参数将发生变化。



寄存器约定

每个寄存器有两种类型：

- 呼叫方已储存
  - — 被调用者功能可以自由使用
    （如果需要，调用者必须在调用之前保存它们，并在之后恢复它们）
- 被叫方已保存
  - — 被调用方函数修改前必须保存，恢复后返回
    （完全避免使用，无需保存）

这是所有职能部门都同意的合同

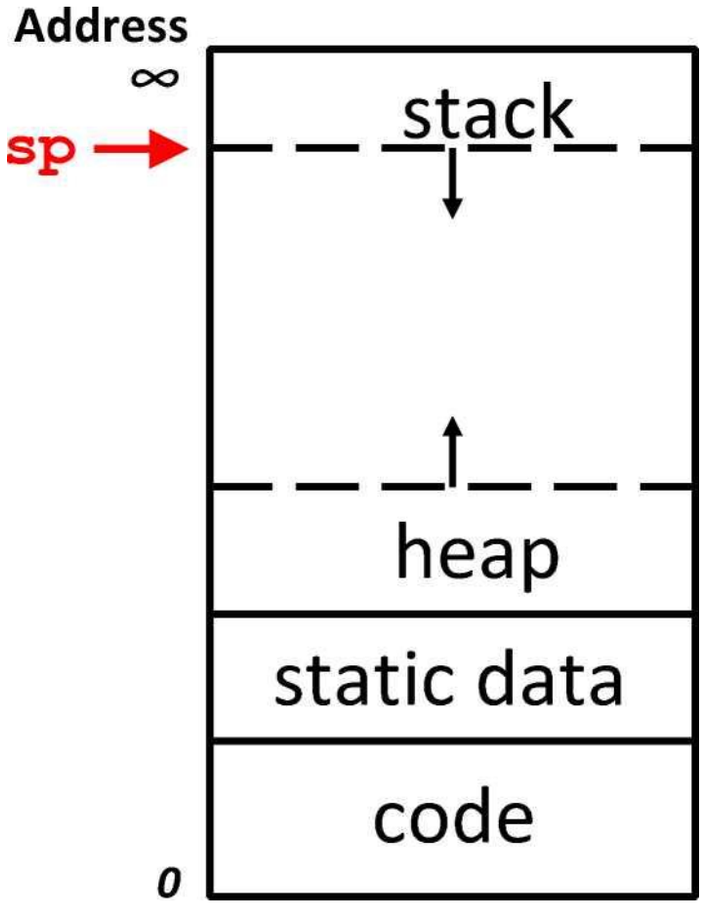
绿卡调用约定

注册名称、用途、召集公约（4）

注册	姓名	用途	拯救者
x0	零	I'he常数值0	不适用
按屋调	零	寄件人地址	呼叫者
x2	gp	堆栈指针	被呼叫者
x3	总	全局指针	—
x4	灶	螺纹指针	—
x5 x7	t0	临时	呼叫者
x8	s0 /	保存的寄存器/帧指针	被呼叫者
x9	对	已保存的寄存器	被呼叫者
L L L	a0	函数参数/返回值	呼叫者
X12	a2	函数参数	呼叫者
x18	s2	已保存的寄存器	被呼叫者
28 不	t3	临时	呼叫者
f0 f7	英口	FP临时工	呼叫者
f8 f9	fs0	FP 保存的寄存器	被呼叫者
浮 进	fa0	FP 函数参数/返回值	呼叫者
f12	fa2	FP 函数参数	呼叫者
f18	fs2	FP 保存的寄存器	被呼叫者
f28	ft8	R|rdl = Rfrsl 1 4- R[rs2]	呼叫者

gp 和 tp 是特殊的寄存器，在本课程中我们不会担心

我们如何保存寄存器?这些堆叠!

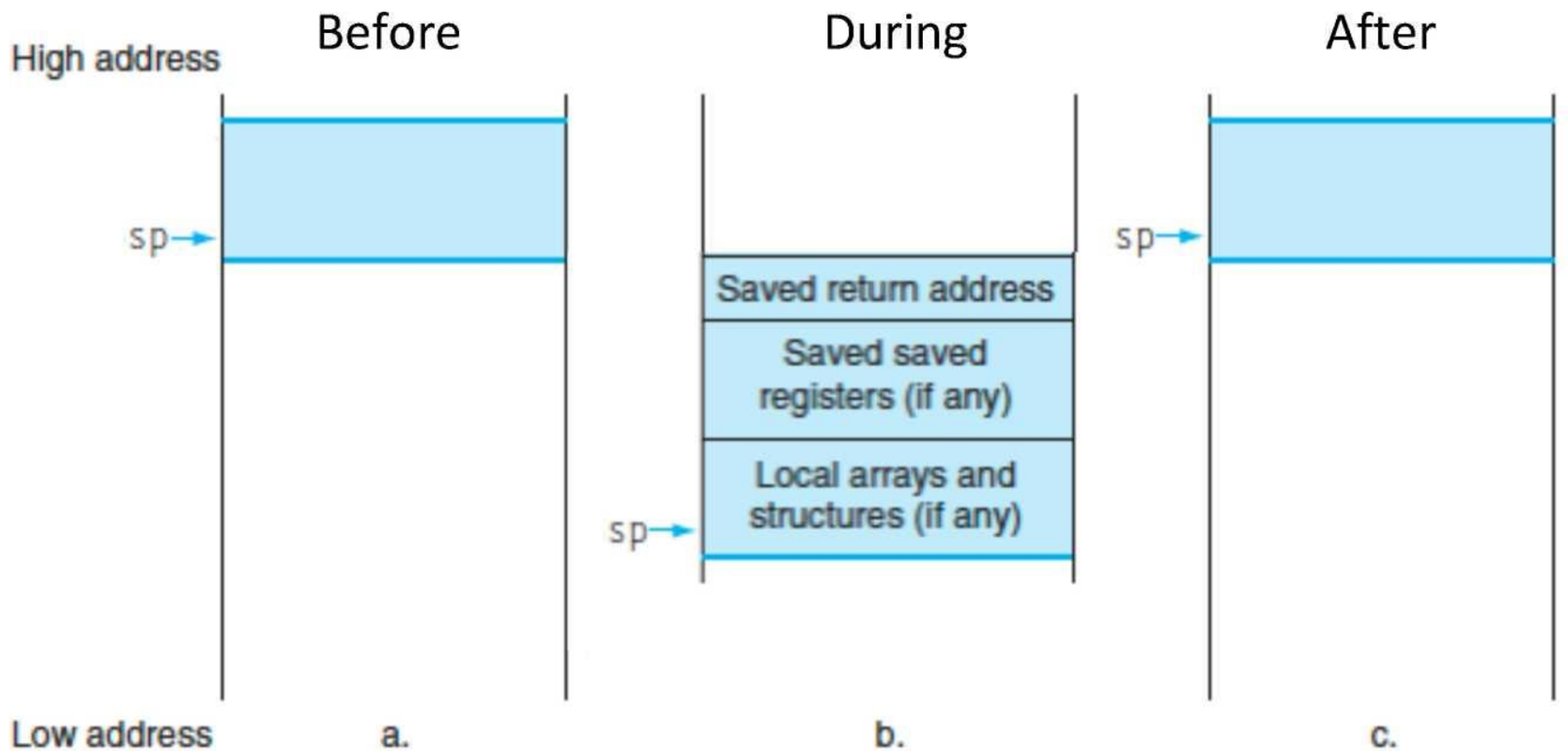


局部变量和保存的寄存器

动态分配的空间全局变量、字符串文本

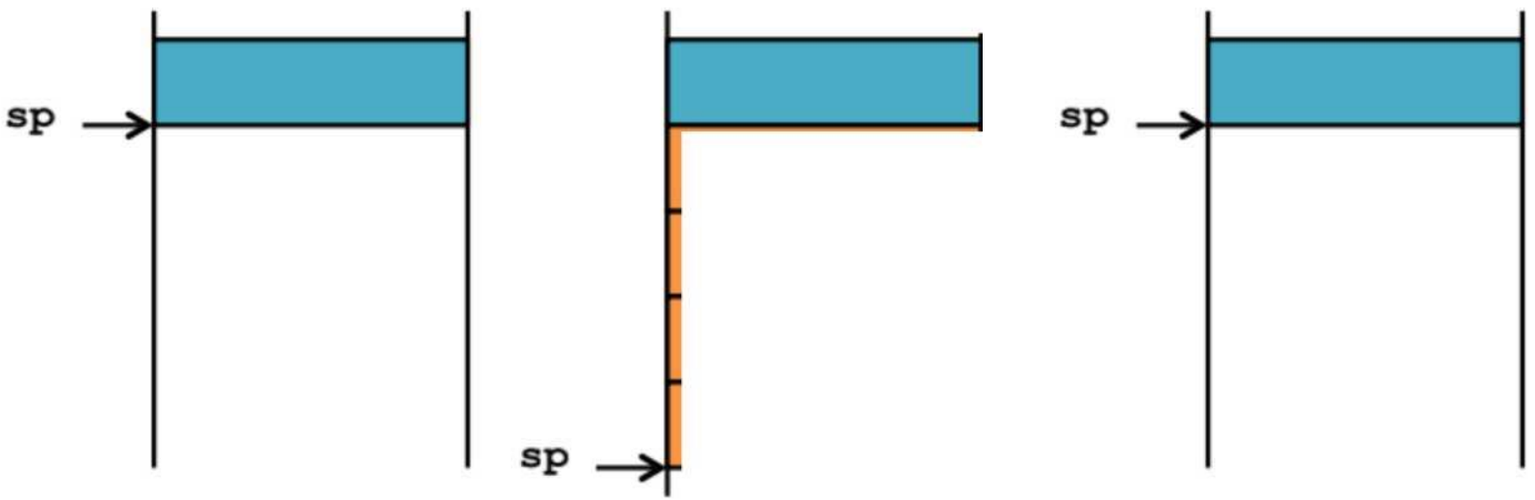
程序说明

调用前、调用中、调用后堆栈



调用前、调用中、调用后堆栈

之前　　期间　　之后



保存的退货

地址（如需要）

已保存参数

寄存器（如有）

已保存已保存

寄存器（如有）

局部变量

```
int sumSquare（int x, int y）
                    {
    返回 mult（x,x）+ y;}
```

平方和：

“推送”

```
附加 sp,sp,-8
sw ra, 4
（sp）sw a1,
0（sp）加
贾尔多
a1,a0,x0
```

“流行”

```
  * lw a1, 0
   （sp）/ add
  a0,a0,a1 lw
   ra, 4（sp）
* 瑞拉加 sp,sp,8
```

mult：

```
编在堆栈上留出空间
编保存检索地址
编保存 y
编设置第二个多参数
编呼叫多重
编恢复 y
编ret val = mult（x,x）
编获取 ret 地址
编恢复堆栈
```

函数的基本结构

开场白

```
func_label:
addi sp,sp, -framesize
SWRA,<framesize-4>（sp）
#存储其他被调用方保存的寄存器
#如果需要，请保存其他注册表
```

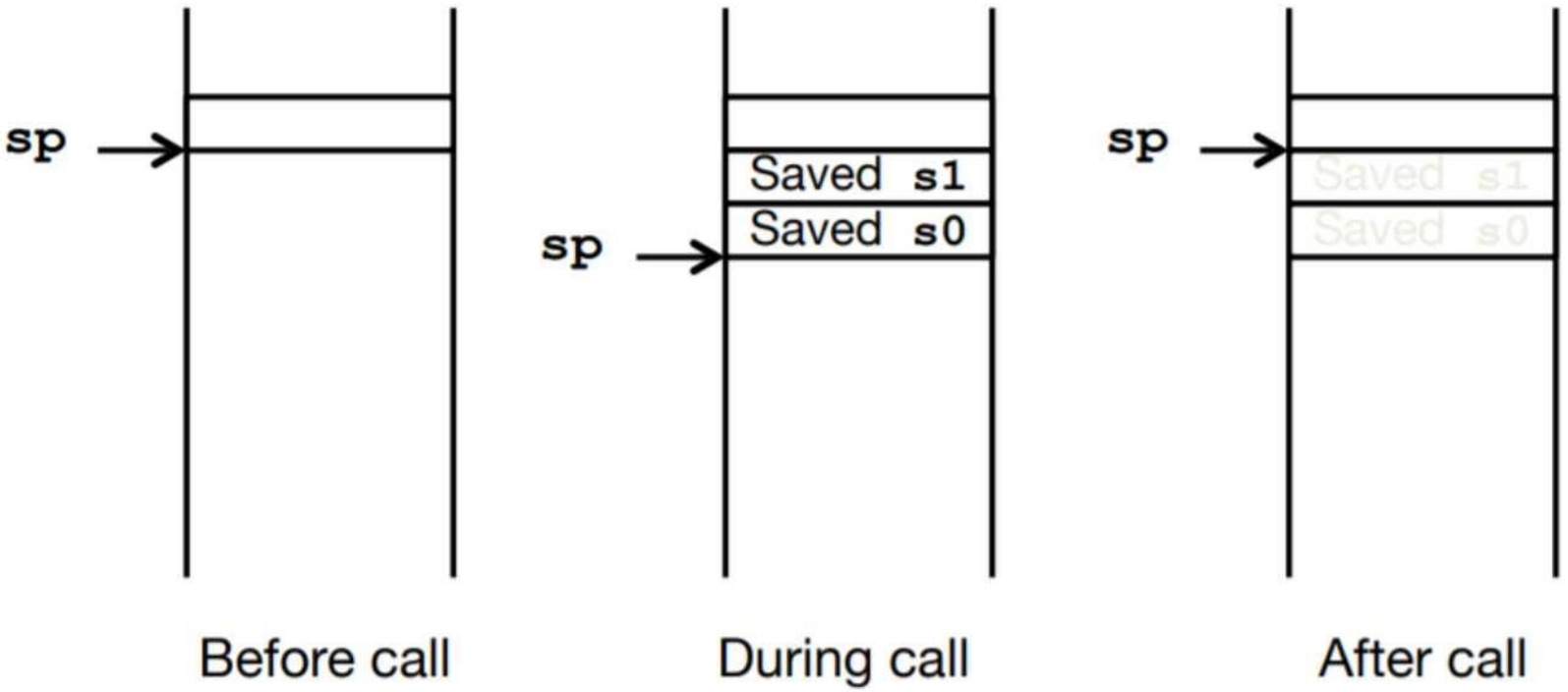
正文（调用其他功能...）

结语

```
#根据需要恢复其他 reg
#恢复其他被调用方保存的寄存器
LWRA,<framesize-4>（sp）
添加 sp,sp, 帧大小 I
```

函数执行过程中的堆栈

- 需要保存 s0 和 si 的旧值



示例：使用保存的寄存器

```
myFunc:  # 使用 s0 和 si
    addi        sp,sp,-i2   编
    SW          ra,8（sp）   编
    SW          s0,4（sp）
    SW          硅，0（sp）
    • ••                    编
    贾尔        函数         编
    • ••                    编
    贾尔        函数 2       编
    • ••                    编
    长波        硅，0（sp）   编
    长波        s0,4（sp）   编
    长波        ra,8（sp）
    addi        sp,sp,i2
    焦耳        雷           编
```

这是开场白

保存已保存的寄存器

用 s0 和 si 做事

funci 和 func2 会遵守约定，所以我们不在乎它们使用 s0 还是 si,我们可以正常使用

用 s0 和 si 做事

这是后记

恢复已保存的寄存器

返回

示例：使用易失性寄存器

```
myFunc: 使用次数 t0
addi    sp,sp,-4    这是开场白
SW      ra,0        #  保存已保存的寄存器
• ••                #  与 t0 做些事情
addi    sp,sp,-4    #  保存易失性寄存器
SW      t0,0        #  调用函数前
贾尔    功能 1      #  功能可更改  t0
长波    t0,0        #  恢复易失性寄存器
addi    sp,sp,4     #  在您再次使用它们之前
• ••                #  与 t0 做些事情
长波    ra,0        这是结尾
addi    sp,sp,4     #  恢复已保存的寄存器
焦耳    雷          #返回
```

寄存器惯例摘要

- 再来一次运气：
  - 在进行过程调用之前，CalleR 必须将其使用的任何易失性寄存器保存到堆栈中
  - CalleR 可以信任保存的寄存器来维护值
  - 被调用者必须“保存”它打算使用的任何已保存寄存器，方法是在覆盖其值之前将其放入堆栈中
- 注释：
  - CalleR 和 calleE 只需保存其使用的相应寄存器（并非全部!）
  - 不要忘记稍后恢复值

RISC-V 议程

- 标志扩展练习
- 伪指令
- C 到 RISC-V 实践
- 程序集中的函数
- 函数调用约定
- **总结**

示例函数带呼叫惯例

```
int Leaf（int g, int h,
         int i, int j）{

        intf;
        f=（g+h）-（i+j）;返
        回 f;
}

Leaf:  addi sp,sp,-8 #
allocate stack ss1,4（sp）
sw s0,0（sp）add s0,a0,a1
add s1,a2,a3
sub a0,s0,s1 #
返回值

lw s0,0（sp）
lw s1,4（sp）
附加 sp,sp,8
jr ra
```

```
# 保存 s1
# 保存 s0

# s0 = g+h
# s1 = i+j


a0 = s0-s1

# 恢复 s0
# restore s1
# free stack
# return
```

选择您的寄存器

- 最小化寄存器占用空间
  - 通过选择函数中使用的寄存器进行优化，以减少需要保存的寄存器数量
  - 仅在绝对需要时保存
- 函数不调用其他函数
  - 仅使用 t0-t6,没有可保存的内容!
- 函数调用其他函数
  - 您需要的值始终在 s0-s11 中，其他值在 t0-t6 中
  - 在每次函数调用时，检查数字参数和返回值，以确定是否需要保存

不同寄存器选择可以减少工作量

```
int Leaf（int g, int h, int
        i, int j）{

    intf;
        f=（g+h）-（i+j）;返回
        f;
}
```

叶片：

```
# stack add t0,a0,a1 #
t0 = g+h add t1,a2,a3 # t1 =
i+j sub a0,t0,t1
# 返回值 a0 = t0-t1
```

#返回

#瑞投有要从堆栈恢复的内容

懒惰点!使用可以最大限度地减少保存到堆栈的寄存器选择。它使您的程序也变得更快...

总结（1/2）

- 伪指令
- 函数是程序集
  - 调用函数的六个步骤
    1. 放置参数
    2. 跳转到函数
    3. 创建本地存储（Prologue）
    4. 执行预期任务
    5. 放置返回值并清理存储（Epilogue）
    6. 跳回呼叫方

总结（2/2）

- 调用约定
  — 需要一种方法来了解在函数调用中可以信任哪些寄存器
  - 呼叫方保存的寄存器（易失性寄存器）
    - 如有需要，由呼叫者保存
    - 被调用方可免费使用
  - 被叫方保存的寄存器（保存的寄存器）
    - 如果需要，由被调用方保存
    - 跨调用方的函数调用安全