CS3410: Computer Systems and Organization

LEC13: RISC-V Calling Convention

Professor Giulia Guidi
Wednesday, October 8, 2025

Credits: Bala, Bracy, Garcia, Guidi, Kao, Sampson, Sirer, Weatherspoon

GDB **not** on the prelim

Prelim

Our **prelim** is coming up **Thursday, October 9**, and here's how the room assignments will work:

- **Early Prelim:** 5:30 PM in Gates Hall G01 (everyone taking the alternate will meet here)
- **Regular Prelim:** 7:30 PM
  - **UPDATED** Kennedy Hall KND 116 — Last names **A–X**
  - **UPDATED** Kennedy Hall KND 213 — Last names **Y–Z**

Quick reminders:

- Please **use the restroom before the exam**
- During testing **no devices** (phones, calculators, watches, headphones, etc.) are permitted
- Don't forget to **bring your Cornell ID** since we'll use it to check you in and swap for your netID

Plan for Today

- RISC-V Calling Convention and Stack

```
#include <stdbool.h>


int main() {
    bool have_time = false; // Change this to true if time magically appears!
    if (have_time) {
        printf("Doing prelim review with Prof. Guidi!\n”);
    } else {
        printf(“Ok, self review + Ed\n");
    }
    return 0;
}
```

On to calling convention [not in the prelim]

PS: this is needed for **A7**

Purpose of Calling Convention

A calling convention is a set of rules that defines **how functions communicate**:

- E.g., how **arguments** are passed
- E.g., how **return values** are returned
- E.g., how **registers** and the **stack** are managed
- E.g., **who** is responsible for saving/restoring what

The essentially the "**contract**" between the caller (e.g., `main`) and callee, e.g., function `f()`

RISC-V Calling Convention

A calling convention is a set of rules that defines **how functions communicate**:

Register	Use
`x1 → ra`	The **return** address
`x2 → sp`	The **stack** pointer
`x5–x7 → t0–t2`	The **temporary** registers (caller-saved)
`x10–x17 → a0–a7`	The function **arguments** and **return** values (`a0–a1` only)
`x8–x9 → s0–s1`	The **saved** registers (callee-saved)

Caller-saved: Caller must save if it wants the value after the call

Callee-saved: Callee must preserve them across the call

RISC-V Calling Convention

A calling convention is a set of rules that defines **how functions communicate**:

Register	Use
`x1 → ra`	The **return** address — If `main` calls a function `add`, **`ra`** stores the address of the instruction following the call to `add` in `main`
`x2 → sp`	The **stack** pointer — **`sp`** stores the address in memory of the top of the **stack**
`x5–x7 → t0–t2`	The **temporary** registers (caller-saved) — The subroutine **can** modify these values
`x10–x17 → a0–a7`	The function **arguments** and **return** values (`a0–a1` only)
`x8–x9 → s0–s1`	The **saved** registers (callee-saved)

Caller-saved: Caller must save if it wants the value after the call

Callee-saved: Callee must preserve them across the call

RISC-V Calling Convention

I have **8** registers for the **arguments**. But what happens if my function has **10** arguments?

Register	Use
`x1 → ra`	The **return** address
`x2 → sp`	The **stack** pointer
`x5–x7 → t0–t2`	The **temporary** registers (caller-saved)
`x10–x17 → a0–a7`	The function **arguments** and **return** values (`a0–a1` only)
`x8–x9 → s0–s1`	The **saved** registers (callee-saved)

RISC-V Calling Convention

I have **8** registers for the **arguments**. But what happens if my function has **10** arguments?

Register	Use
`x1 → ra`	The **return** address
`x2 → sp`	The **stack** pointer
`x5–x7 → t0–t2`	The **temporary** registers (caller-saved)
`x10–x17 → a0–a7`	The function **arguments** and **return** values (`a0–a1` only)
`x8–x9 → s0–s1`	The **saved** registers (callee-saved)

The first 8 integer arguments → `a0–a7`

Remaining arguments → pushed onto the **stack**

RISC-V Calling Convention

A calling convention is a set of rules that defines **how functions communicate**:

Register	Use
`x1 → ra`	The **return** address — If `main` calls a function `add`, **`ra`** stores the address of the instruction following the call to `add` in `main`
`x2 → sp`	The **stack** pointer — **`sp`** stores the address in memory of the top of the **stack**
`x5–x7 → t0–t2`	The **temporary** registers (caller-saved) — The subroutine **can** modify these values
`x10–x17 → a0–a7`	The function **arguments** and **return** values (`a0–a1` only)
`x8–x9 → s0–s1`	The **saved** registers (callee-saved) — The subroutine e.g., `add` can touch these values, but it **must** restore them

Caller-saved: Caller must save if it wants the value after the call

Callee-saved: Callee must preserve them across the call

Poll

In the RISC-V calling convention, which registers must a callee (the called function) restore before returning?

PollEv.com /gguidi

Or send **gguidi** to **22333**

Function Call

Let's go through the `add` function **execution**:

```
int add(int x, int y) {
    int sum = x + y;
    return sum;
}


int main() {
    int a = 5, b = 7;   <-- PC
    int c = add(a, b);
    printf("%d", c);
}
```

1. The **caller** `main` prepares **arguments** and puts them in a place where the function `add` can access them

Step	Registers	Action
Caller prepares args	`a0 = 5, a1 = 7`	Put args in `a0`–`a1`

Function Call

Let's go through the `add` function **execution**:

```
int add(int x, int y) {
    int sum = x + y;
    return sum;
}

int main() {
    int a = 5, b = 7;
    int c = add(a, b);  <-- PC
    printf("%d", c);    <-- ra
}
```

2. The **caller** `main` transfers control the function `add`

Step	Registers	Action
Caller prepares args	`a0 = 5, a1 = 7`	Put args in `a0`–`a1`
Call `add`	`ra` = return addr	Jump to `add`

Function Call

Let's go through the `add` function **execution**:

```
int add(int x, int y) {
    int sum = x + y;
    return sum;
}

int main() {
    int a = 5, b = 7;
    int c = add(a, b);
    printf("%d", c);
}
```

PC ← 2. The **caller** `main` transfers control the function `add`

ra ←

Step	Registers	Action
Caller prepares args	`a0 = 5, a1 = 7`	Put args in `a0`–`a1`
Call `add`	`ra` = return addr	Jump to `add`

Function Call

Let's go through the `add` function **execution**:

```
int add(int x, int y) {
    int sum = x + y;
    return sum;
}

int main() {
    int a = 5, b = 7;
    int c = add(a, b);
    printf("%d", c);
}
```

PC

3. Obtain (local) storage resources `x10` and `x11` as needed for the function and perform the desired task `x + y`

Step	Registers	Action
Caller prepares args	`a0 = 5`, `a1 = 7`	Put args in `a0`–`a1`
Call `add`	`ra` = return addr	Jump to `add`
In `add`	`x10 = a0`, `x11 = a1`	If needed, allocate `sum` on stack

Function Call

Let's go through the `add` function **execution**:

```
int add(int x, int y) {
    int sum = x + y;
    return sum;                ← PC
}

int main() {
    int a = 5, b = 7;
    int c = add(a, b);
    printf("%d", c);           ← ra
}
```

4. Put the return value in a place where the calling code `main` can access it and restore any registers you used; release local storage

Step	Registers	Action
Caller prepares args	`a0 = 5`, `a1 = 7`	Put args in `a0`–`a1`
Call `add`	`ra` = return addr	Jump to `add`
In `add`	`x10 = a0`, `x11 = a1`	If needed, allocate `sum` on stack
Return	`a0 = sum`	Jump back using `ra`

Function Call

Let's go through the `add` function **execution**:

```
int add(int x, int y) {
    int sum = x + y;
    return sum;
}

int main() {
    int a = 5, b = 7;
    int c = add(a, b);
    printf("%d", c);      <-- PC
}
```

5. Return **control** to the point of origin, since a function can be called from several points in a program

Step	Registers	Action
Caller prepares args	`a0 = 5`, `a1 = 7`	Put args in `a0`–`a1`
Call `add`	`ra` = return addr	Jump to `add`
In `add`	`x10 = a0`, `x11 = a1`	If needed, allocate `sum` on stack
Return	`a0 = sum`	Jump back using `ra`
Caller resumes	`c = a0`	Retrieve return value

Helpful RISC-V Assembler Features

- Register symbolic names
  - E.g., **`a0–a7`** for argument registers (**`x10–x17`**) for function calls
  - E.g., **`zero`** for **`x0`**

- Pseudo-instructions; the **shorthand syntax** for common assembly idioms
  - E.g., **`mv rd, rs = addi rd, rs, 0`**
  - E.g., **`li rd, 13 = addi rd, x0, 13`**
  - E.g., **`nop = addi x0, x0, 0`**

Instruction Support for Function

C:

```
/* a, b:s0, s1 */
… sum(a, b); …
}

int sum(int x, int y)
{
  return x + y;
}
```

RISC-V:


```
address (shown in decimal)
1000
1004
1008
1012
1016
…
2000
2004
```

let us assume instructions are 4 bytes

Instruction Support for Function

C:

```
/* a, b:s0, s1 */
… sum(a, b); …
}

int sum(int x, int y)
{
  return x + y;
}
```

RISC-V:

```
address (shown in decimal)
1000 mv   a0, s0 # x = a
1004 mv   a1, s1 # y = b
1008 addi ra, zero, 1016 # ra = 1016
1012 j    sum    # jump to sum
1016 …           # next instruction
…
2000 sum: add a0, a0, a1
2004 jr   ra     # new instr
```

PC = R[ra]

Uncoditional jump to address R[ra]

Instruction Support for Function

C:

```
/* a, b:s0, s1 */
… sum(a, b); …
}

int sum(int x, int y)
{
  return x + y;
}
```

RISC-V:

```
address (shown in decimal)
1000 mv   a0, s0 # x = a
1004 mv   a1, s1 # y = b
1008 addi ra, zero, 1016 # ra = 1016
1012 j    sum    # jump to sum
1016 …           # next instruction
…
2000 sum: add a0, a0, a1
2004 jr   ra     # new instr
```

Question: why use `jr` here instead of `j`?

Instruction Support for Function

A single instruction to jump and **save return address**: jump and link: `jal`

- Before:
  - `1008 addi ra, zero, 1016  # ra = 1016`
  - `1012 j sum                # goto sum`

This is **usually bad practice:** If you move your code or reload it somewhere else in memory, those fixed addresses **might no longer be valid**, and your program could break.

- Then:
  - `1008 jal ra, sum          # ra = 1012, goto sum`

`jal` = “Jump and Link”

`PC + 4`

`ra` is the destination register (reiceives the return address, i.e., address of the **next** instruction)

`sum` is the target label (address of the function or code to jump to)

Instruction Support for Function

A single instruction to jump and **save return address**: jump and link: **jal**

- Before:
  - `1008 addi ra, zero, 1016  # ra = 1016`
  - `1012 j sum                # goto sum`
- Then:
  - `1008 jal ra, sum          # ra = 1012, goto sum`

Question: why have **jal**?

- To make the common case fast: function calls very common
- Reduce program size
- Don’t have to know where code is in memory with **jal**!

RISC-V Function Calling Convention

- Invoke function: **jump and link** instruction `jal`
  - Jumps to address and **simultaneously** saves the address of the following instruction in generic register `rd`

```
jal rd, FunctionLabel
```

- Return from function: **jump register** instruction `jr`
  - Unconditional jump to address specified in register: `jr rd`
  - The assembler shorthand: `ret = jr ra`

Instruction Support Summary

- Only two instructions:
  - Jump-and-link: `rd = pc + 4; pc += imm`

    `jal  rd, Label`

  - Jump-and-link register: `rd = pc + 4; pc = R[rs1] + imm`

    `jalr rd, rs, imm`

    ```
    R[rd] = PC + 4
    PC = R[rs1] + imm
    ```

The problem with `jal` + `label` is that there might **not** be enough bits left for `label` to go as far as we want to jump

It means: write `PC + 4` to register `rd` + **unconditional** jump: go to statement at address `R[rs1] + imm`

Instruction Support Summary

- Only two instructions:
  - Jump-and-link: `rd = pc + 4; pc += imm`

    `jal  rd, Label`

  - Jump-and-link register: `rd = pc + 4; pc = R[rs1] + imm`

    `jalr rd, rs, imm`

The problem with `jal` + `label` is that there might **not** be enough bits left for `label` to go as far as we want to jump

`j`, **`jr`** and **`ret`** are pseudo-instructions!

`j = jal  x0, Label`

But where are old register values saved to restore them after a function call?

The Place to Be?

- The ideal place is the **stack**: last-in first-out (LIFO) queue (e.g., stack of plates)
  - **Push:** placing data onto the stack
  - **Pull:** removing data from the stack

- The stack is in **memory**
  - Thus, we need a **register to point to it**

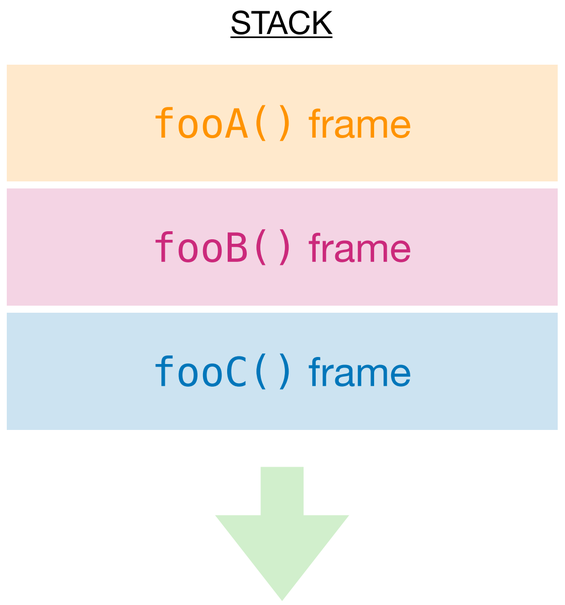
- `sp` is the stack pointer in RISC-V
- Convention is grow stack down from high to low addresses

Stack

A stack is like a vertical stack of boxes: You add (**push**) boxes on top and take (**pop**) boxes from the top

Every time a function is called, a new **stack frame** is allocated on the stack

```
fooA() { fooB(); }
fooB() { fooC(); }
fooC() { … }
```



Each stack **frame** includes:

- Return “instruction” address
- Parameters (arguments)
- Space for other local variables

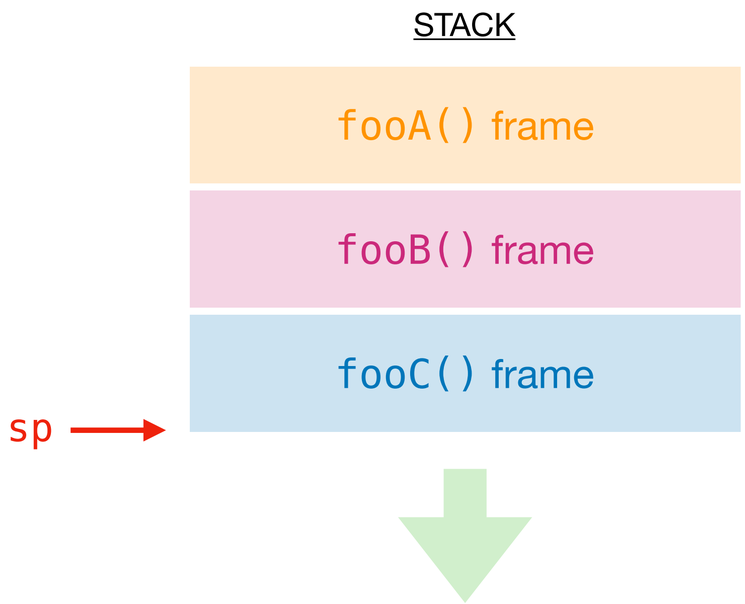
Stack

A stack **frame** constitutes a **contiguous** block of memory

The stack pointer `sp` tells us the “top of the stack,” i.e., the start of the current stack frame

```
fooA() { fooB(); }
fooB() { fooC(); }
fooC() { … }
```

If `fooC()` terminates:



Each stack **frame** includes:

- Return “instruction” address
- Parameters (arguments)
- Space for other local variables

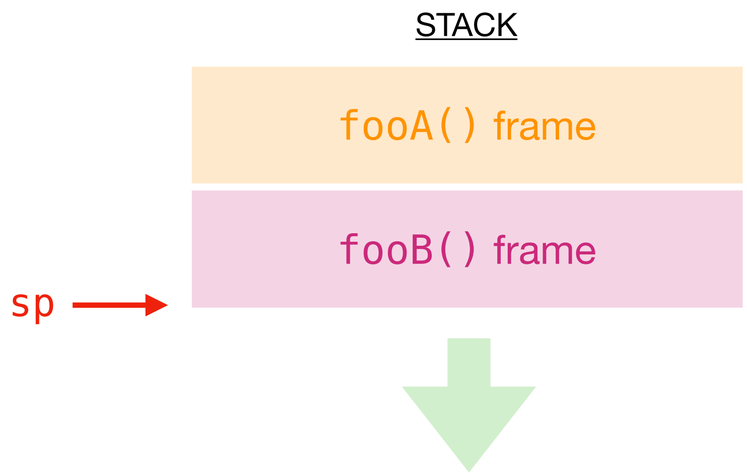
Stack

So when the function **ends**, the stack frame is “tossed off the stack”

The stack pointer `sp` tells us the “top of the stack,” i.e., the start of the current stack frame

```
fooA() { fooB(); }
fooB() { fooC(); }
fooC() { … }
```

If `fooC()` terminates:



Each stack **frame** includes:

- Return “instruction” address
- Parameters (arguments)
- Space for other local variables

Stack

```
int main ()
{ a(0);  …
}
void a (int m)
{ b(1);
}
void b (int n)
{ c(2);
}
void c (int o)
{ d(3);
}
void d (int p)
{
}
```

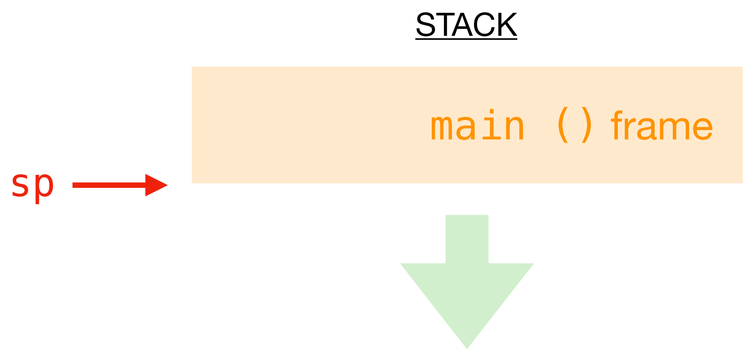
0xFFFF FFFF

The **stack pointer** sp is what determines “allocating” and “deallocating/freeing” stack frames!

Stack

```
int main ()
{ a(0);  …
}
void a (int m)
{ b(1);
}
void b (int n)
{ c(2);
}
void c (int o)
{ d(3);
}
void d (int p)
{
}
```

PC

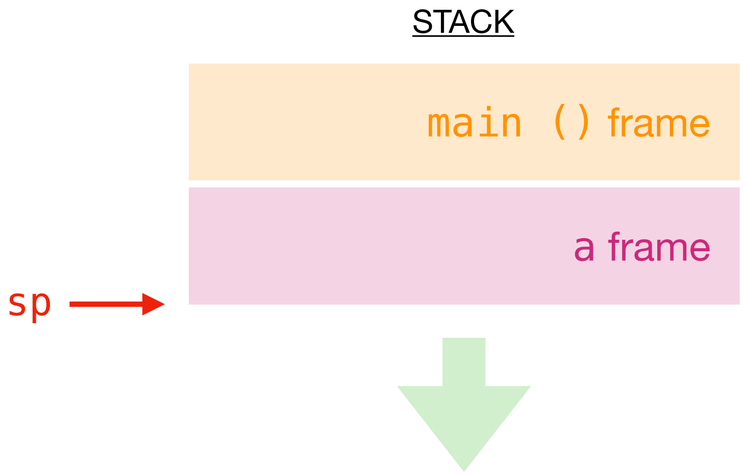


The **stack pointer** `sp` is what determines “allocating” and “deallocating/freeing” stack frames!

Stack

```
int main ()
{ a(0);  …
}
void a (int m)
{ b(1);
}
void b (int n)
{ c(2);
}
void c (int o)
{ d(3);
}
void d (int p)
{
}
```

PC

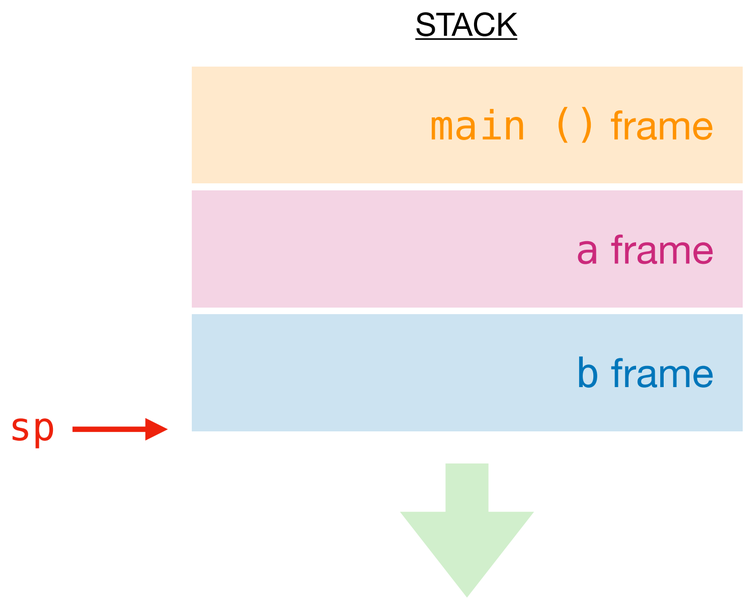


The stack grows **downward;** `a()`'s local variables have **lower** byte addresses than `main()`'s, and so on

Stack

```
int main ()
{ a(0);  …
}
void a (int m)
{ b(1);
}
void b (int n)
{ c(2);
}
void c (int o)
{ d(3);
}
void d (int p)
{
}
```

PC



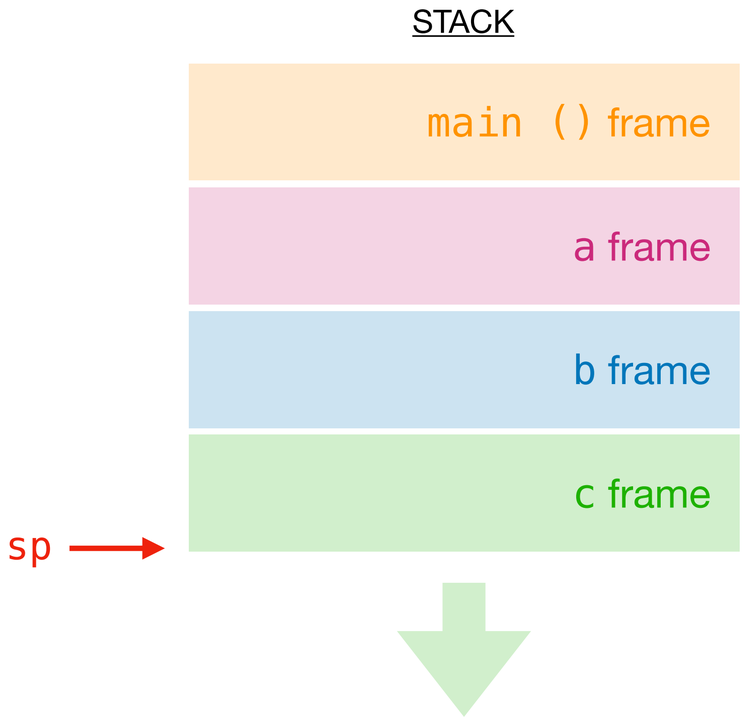
The stack grows **downward;** `a()`'s local variables have **lower** byte addresses than `main()`'s, and so on

`b()`'s local variables have **lower byte addresses** than `a()`'s, and so on

Stack

```
int main ()
{ a(0);  …
}
void a (int m)
{ b(1);
}
void b (int n)
{ c(2);
}
void c (int o)
{ d(3);
}
void d (int p)
{
}
```

PC



0xFFFF FFFF

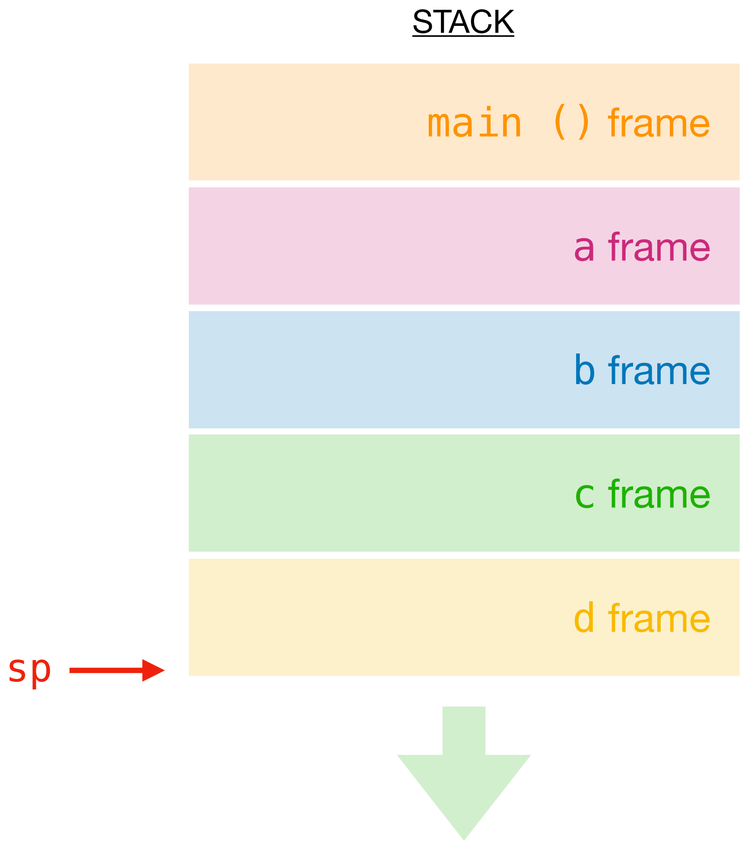
The stack grows **downward;** `a()`'s local variables have **lower** byte addresses than `main()`'s, and so on

`b()`'s local variables have **lower byte addresses** than `a()`'s, and so on

Stack

```
int main ()
{ a(0);  …
}
void a (int m)
{ b(1);
}
void b (int n)
{ c(2);
}
void c (int o)
{ d(3);
}
void d (int p)
{
}
```

PC



0xFFFF FFFF

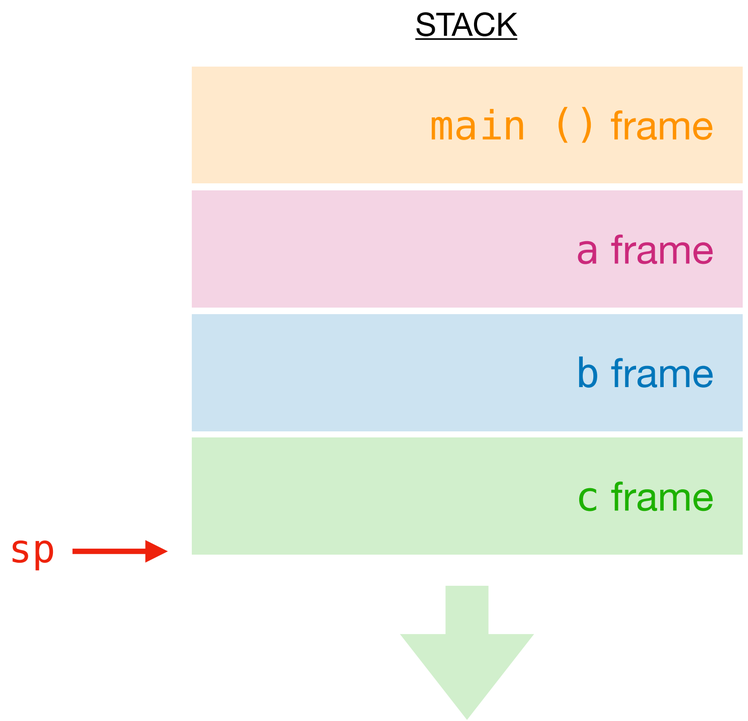
The stack grows **downward;** `a()`'s local variables have **lower** byte addresses than `main()`'s, and so on

`b()`'s local variables have **lower byte addresses** than `a()`'s, and so on

The **stack pointer** `sp` is what determines "allocating" and "deallocating/freeing" stack frames!

Stack

```
int main ()
{ a(0);  …
}
void a (int m)
{ b(1);
}
void b (int n)
{ c(2);
}
void c (int o)      ← PC
{ d(3);
}
void d (int p)
{
}
```



0xFFFF FFFF

The stack grows **downward;** `a()`'s local variables have **lower** byte addresses than `main()`'s, and so on

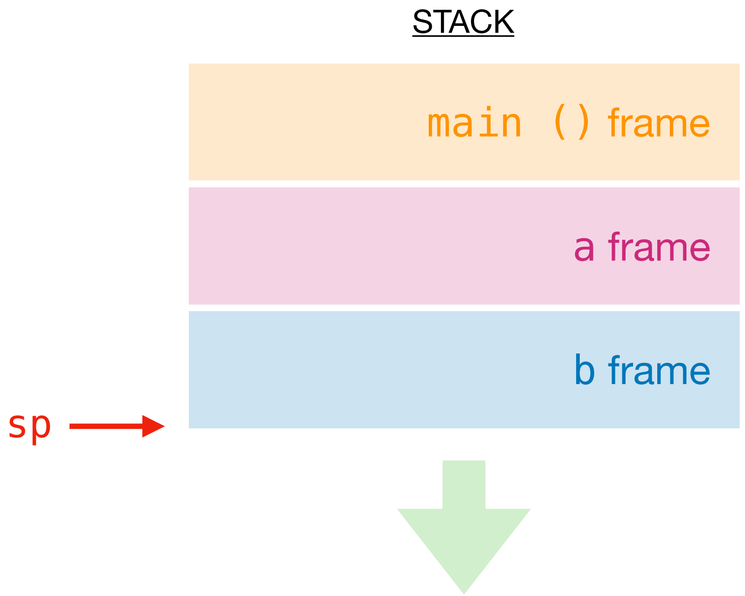
`b()`'s local variables have **lower byte addresses** than `a()`'s, and so on

The **stack pointer** `sp` is what determines "allocating" and "deallocating/freeing" stack frames!

Stack

```
int main ()
{ a(0);  …
}
void a (int m)
{ b(1);
}
void b (int n)
{ c(2);
}
void c (int o)
{ d(3);
}
void d (int p)
{
}
```

← PC



0xFFFF FFFF

The stack grows **downward;** `a()`'s local variables have **lower** byte addresses than `main()`'s, and so on

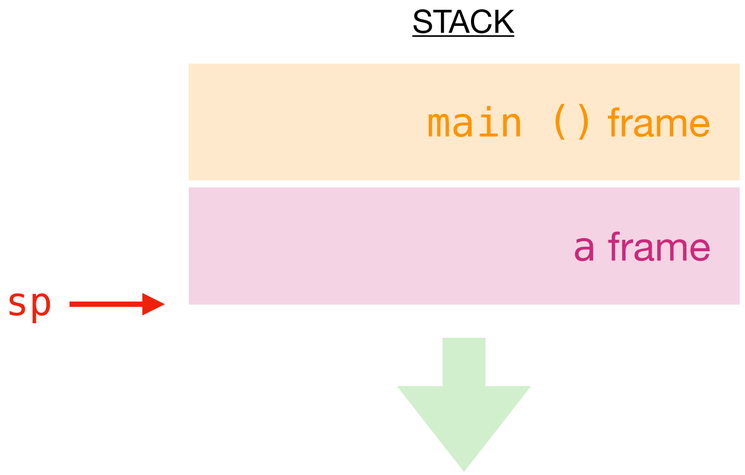
`b()`'s local variables have **lower byte addresses** than `a()`'s, and so on

The **stack pointer** `sp` is what determines "allocating" and "deallocating/freeing" stack frames!

Stack

```
int main ()
{ a(0);  …
}
void a (int m)
{ b(1);
}
void b (int n)
{ c(2);
}
void c (int o)
{ d(3);
}
void d (int p)
{
}
```

PC ⟵ `void a (int m)`



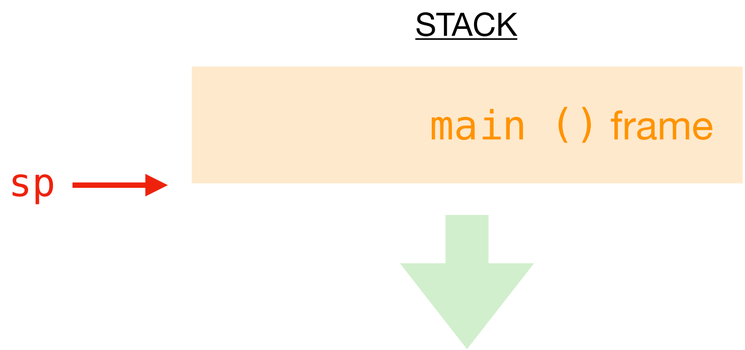
The stack grows **downward;** `a()`'s local variables have **lower** byte addresses than `main()`'s, and so on

The **stack pointer** `sp` is what determines "allocating" and "deallocating/freeing" stack frames!

Stack

```
int main ()
{ a(0);  …
}
void a (int m)
{ b(1);
}
void b (int n)
{ c(2);
}
void c (int o)
{ d(3);
}
void d (int p)
{
}
```

PC



The **stack pointer** `sp` is what determines “allocating” and “deallocating/freeing” stack frames!

Stack

```
int main ()
{ a(0);  …
}
void a (int m)
{ b(1);
}
void b (int n)
{ c(2);
}
void c (int o)
{ d(3);
}
void d (int p)
{
}
```

0xFFFF FFFF

The **stack pointer** sp is what determines “allocating” and “deallocating/freeing” stack frames!

Poll

Consider these function calls: **`profguidi() -> mahler() -> nina()`**: at the end, which function is at the **top** of the stack?

PollEv.com /gguidi

Or send **gguidi** to **22333**

If I call a function in C, where do its variables go?

The table summarizes **where** different types of **variables** live in memory, **how long they exist**, and **when** they are automatically or manually released

The storage duration	Declared in?	The memory area	The lifetime	Freed when?
static	global or static	Data	entire program	program exit
automatic	local variables	Stack	function entry → exit	return
allocated	via `malloc`	Heap	until `free`	explicit free

Stack

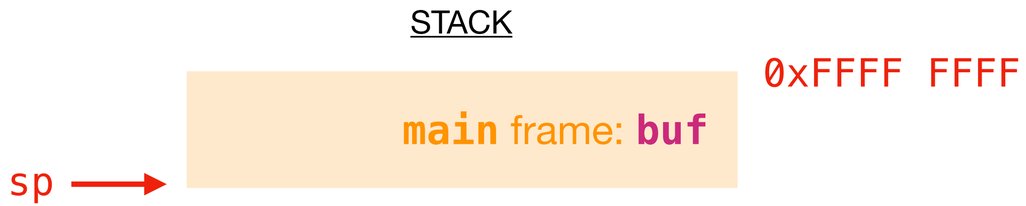
Declared arrays are only allocated while the **scope is valid**

```
void load_buf(char *ptr,
              …) {
  …
};
int main() {
  …
  char buf[...];
  load_buf(buf, BUFLEN);
  …
}
```

Stack

Declared arrays are only allocated while the **scope is valid**

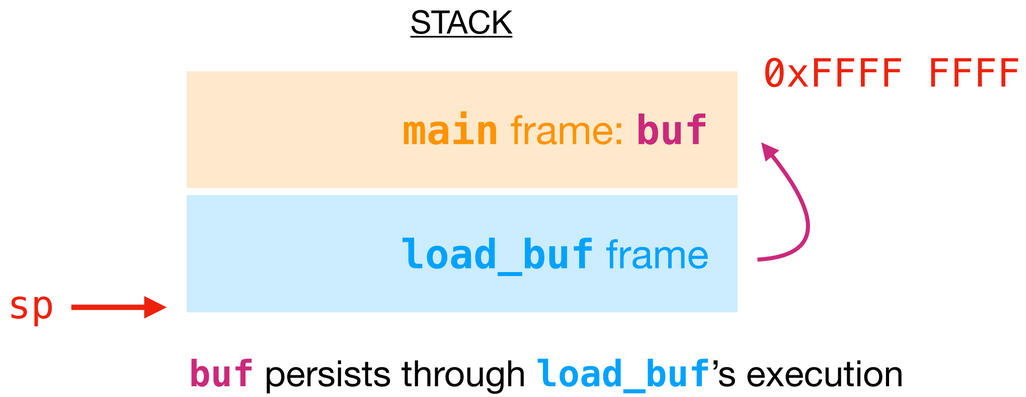
```
void load_buf(char *ptr,
              …) {
  …
};
int main() {
  …
  char buf[...];                ← PC
  load_buf(buf, BUFLEN);
  …
}
```



Stack

Declared arrays are only allocated while the **scope is valid**

```
void load_buf(char *ptr,        ← PC
              …) {
  …
};
int main() {
  …
  char buf[...];
  load_buf(buf, BUFLEN);
  …
}
```



buf persists through **load_buf**'s execution

Stack

Declared arrays are only allocated while the **scope is valid**

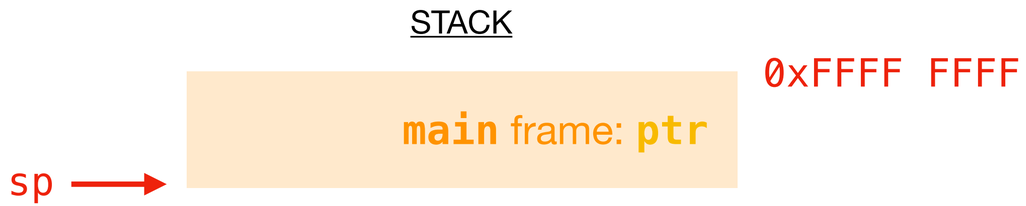
```
char *make_buf() {
    char buf[50];
    return buf;
}
void foo(…) {…}
int main(){
   char *ptr = make_buf();
   foo(ptr);
   …
}
```

Stack

Declared arrays are only allocated while the **scope is valid**

```
char *make_buf() {
    char buf[50];
    return buf;
}
void foo(…) {…}
int main(){
   char *ptr = make_buf();
   foo(ptr);
   …
}
```

PC

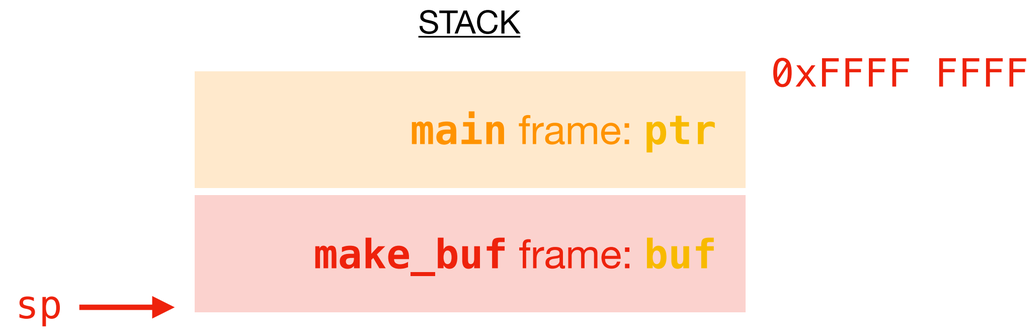


Stack

Declared arrays are only allocated while the **scope is valid**

```
char *make_buf() {
    char buf[50];
    return buf;
}
void foo(…) {…}
int main(){
   char *ptr = make_buf();
   foo(ptr);
   …
}
```

PC

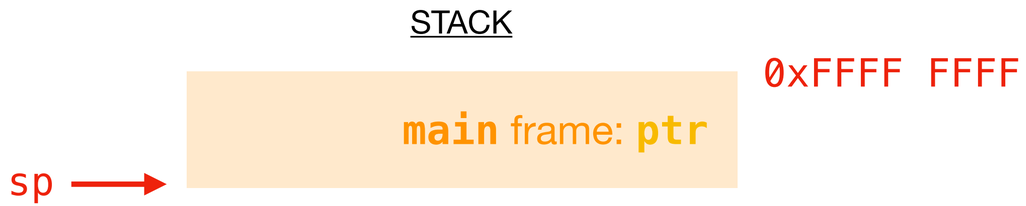


Stack

Declared arrays are only allocated while the **scope is valid**

```
char *make_buf() {
    char buf[50];
    return buf;
}
void foo(…) {…}
int main(){
   char *ptr = make_buf();
   foo(ptr);
   …
}
```

PC

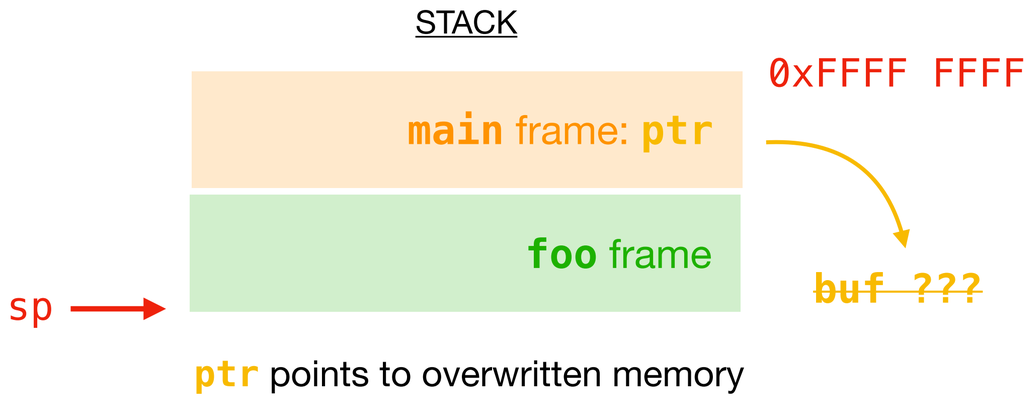


Stack

Declared arrays are only allocated while the **scope is valid**

```
char *make_buf() {
    char buf[50];
    return buf;
}
void foo(…) {…}
int main(){
   char *ptr = make_buf();
   foo(ptr);
   …
}
```

PC



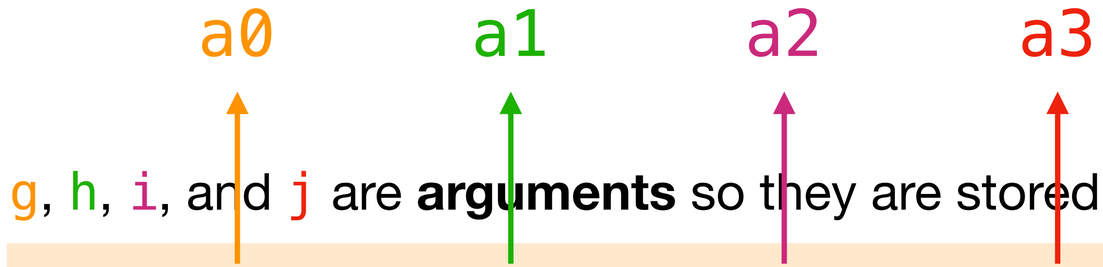
Back to RISC-V

Function Call Example

g, h, i, and j are **arguments**

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

Function Call Example




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


callee-saved

In this example, I'm using `s0` as a temporary register to store the result of the computation

RISC-V Code for Leaf()

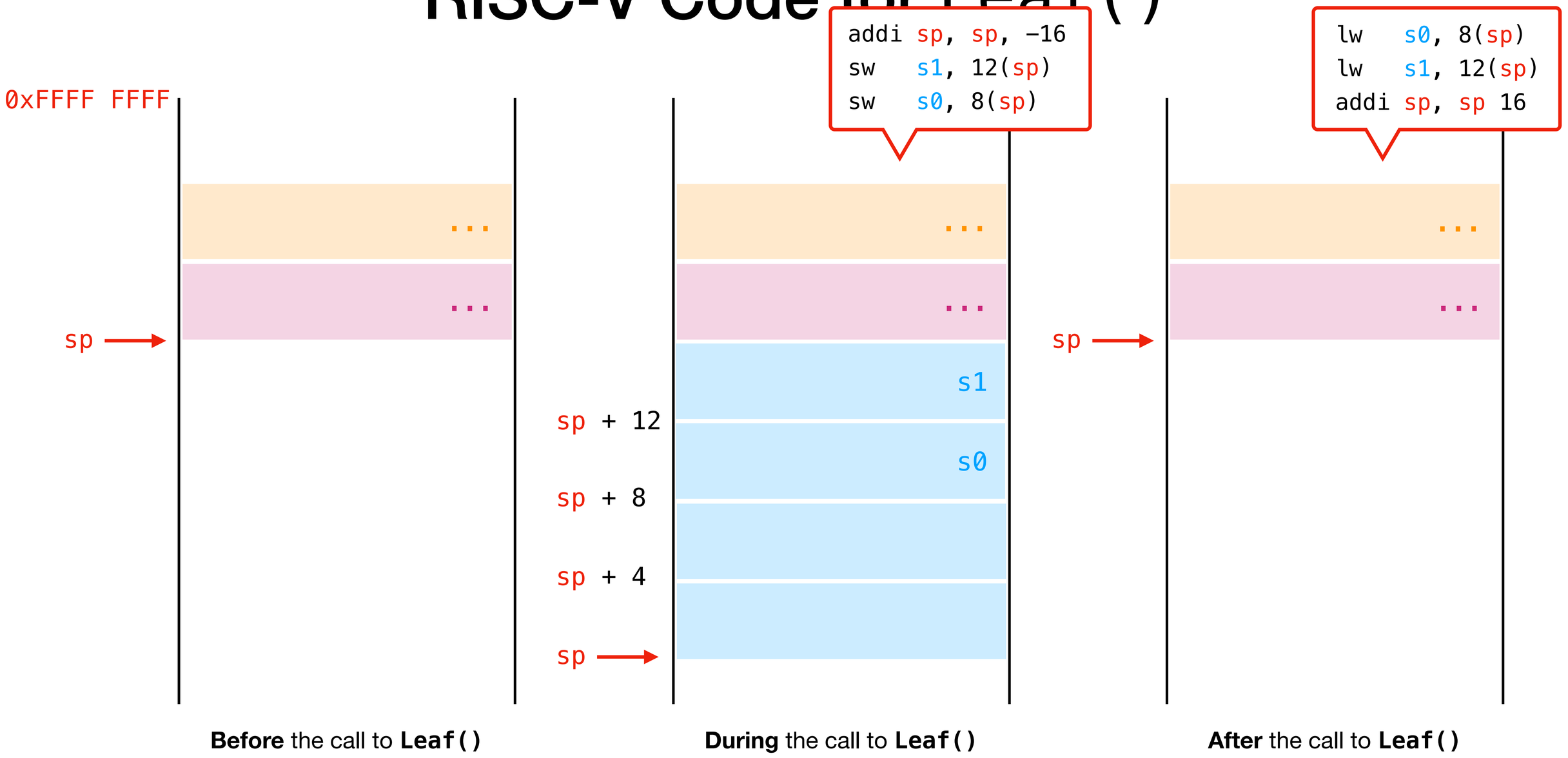
```
# int Leaf(int g, int h, int i, int j)
# a0 = g, a1 = h, a2 = i, a3 = j
# return in a0

Leaf:
    addi sp, sp, -16 # stack allocates 16 bytes (for s0 and s1)
    sw   s1, 12(sp)  # save s1 for use afterward
    sw   s0, 8(sp)   # save s0 for use afterward
    add  s0, a0, a1, # f  = g + h
    add  s1, a2, a3, # s1 = i + j
    sub  a0, s0, s1, # return value (g + h) – (i + j)

    lw   s0, 8(sp)   # restore register s0 for caller
    lw   s1, 12(sp)  # restore register s1 for caller
    addi sp, sp 16   # stack deallocates 16 bytes
    jr   ra          # jump back to calling routine
```

RISC-V, like many modern architectures, mandates a **16-byte alignment** for the stack pointer (**sp**) at function call boundaries

callee-saved

callee-saved

RISC-V Code for Leaf()



Before the call to **Leaf()**

During the call to **Leaf()**

After the call to **Leaf()**

RISC-V Code for `Leaf()`

```
# int Leaf(int g, int h, int i, int j)
# a0 = g, a1 = h, a2 = i, a3 = j
# return in a0

Leaf:
    add   t0, a0, a1   # t0 = g + h
    add   t1, a2, a3   # t1 = i + j
    sub   a0, t0, t1   # a0 = (g + h) - (i + j)
    jr    ra           # return to caller
```

Do **not** need to store the current value of `t0` and `t1` on the stack—the caller was responsible for saving the content, if needed

caller-saved

caller-saved

But what if a function calls a function? Recursive function calls?

Nested Procedures

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

`main()` calls `sumSquare()` and write the **return address** to `main()` to **`ra`**

Nested Procedures

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

`main()` calls `sumSquare()` and write the **return address** to `main()` to `ra`

but then `mult()` is called in `sumSquare()` and `ra` is **overwritten**

```
sumSquare:
    …
    jal ra, mult              # Call mult(x, x)
    …
    jr  ra                    # Return

mult:
    # mult function defined elsewhere
```

Nested Procedures

I need to save **sumSquare** return address before call to **mult**: use **stack**!

```
sumSquare:
    addi sp, sp, -16        # Get some space on the stack
    sw  ra, 12(sp)          # Can store ra on the stack
    …
    jal ra, mult            # Call mult(x, x), ra overwritten
    …
    lw  ra, 12(sp)          # Restore ra from the stack
    …
    jr  ra                  # Return

mult:
    # mult function defined elsewhere
```

Nested Procedures

I need to save **sumSquare** return address before call to **mult**: use **stack**!

```
sumSquare:
      addi sp, sp, -16        # Get some space on the stack
      sw  ra, 12(sp)          # Can store ra on the stack
      sw  a0, 8(sp)           # Put argument x on the stack
      sw  a1, 4(sp)           # Put argument y on the stack

      lw  a0, 8(sp)           # Load x into a0
      lw  a1, 8(sp)           # Load x into a1

      jal ra, mult            # Call mult(x, x)

      mv  t2, a0              # a0 in temporary reg t2
      lw  t3, 4(sp)           # Load y into temporary reg t3
      add t2, t2, t3          # t2 = (x * x) + y

      lw  ra, 12(sp)          # Restore ra from the stack
      addi sp, sp, 16         # Deallocate stack space
      jr  ra                  # Return

mult:
    # mult function defined elsewhere
```

RISC-V Symbolic Register

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	-
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	-
x4	tp	Thread pointer	-
x5	t0	Temporary/Alternate link register	Caller
x6-7	t1-2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function arguments/Return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller