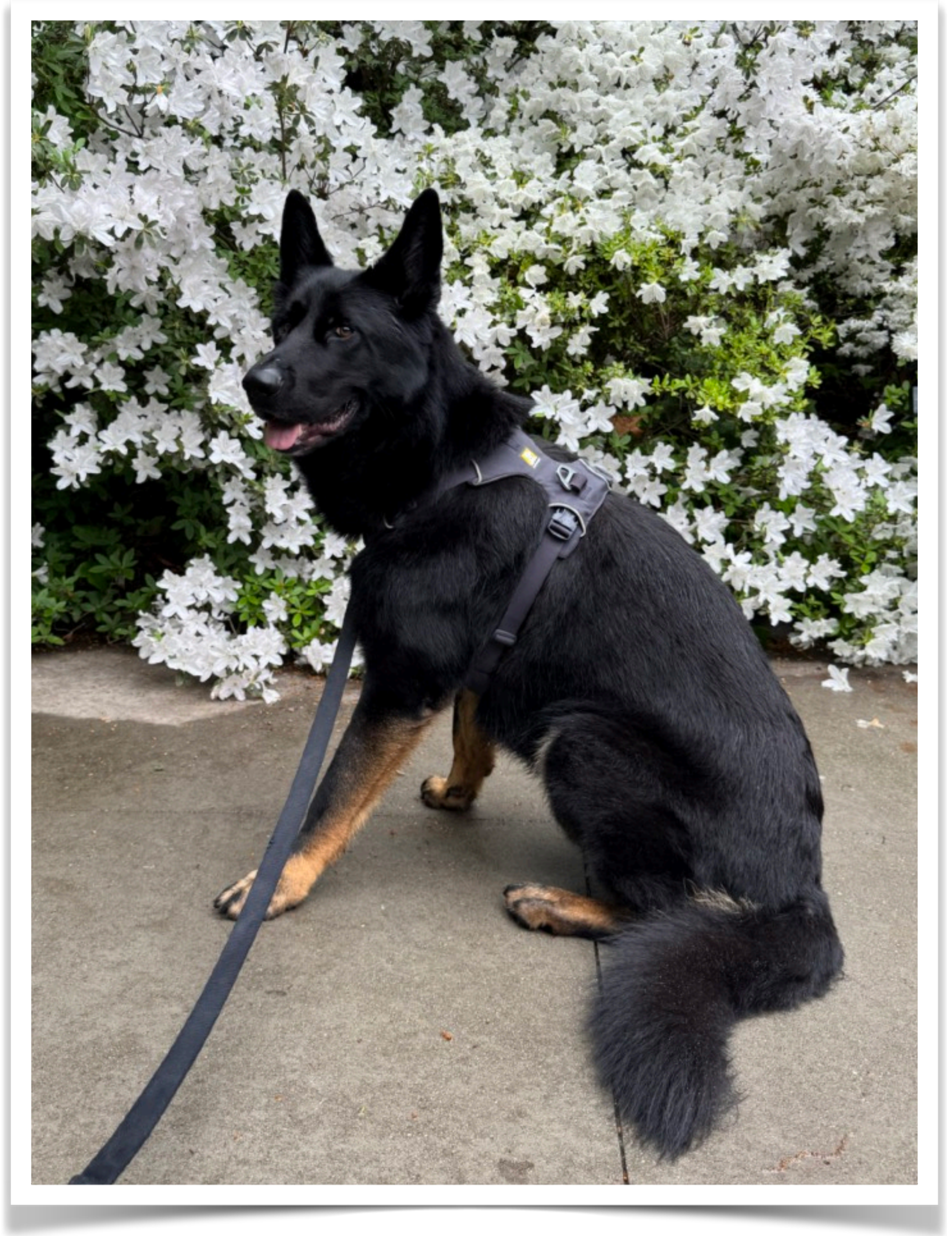
Cornell Bowers C·IS
Computer Science



CS3410: Computer Systems and Organization

LEC14: RISC-V Calling Convention (Vol. II)

Professor Giulia Guidi
Wednesday, October 15, 2025

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

On to a few announcements first

Prelim (Vol. II)

GDB **not** on the prelim

Our **prelim make-up** is coming up **Thursday, October 16**:

- 5:30 PM in Gates Hall G01 (everyone taking the makeup will meet here)

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

Prelim (Vol. II)

If you have taken the **regular** prelim, please see the **option to transfer 50% of prelim weight to final** on EdDiscussion:

- Regrade requests will be handled in bulk on Friday
- The weight transfer offer is **optional**
- If you decided to take the weight transfer offer, you must **let us know by October 22**
- If you decided to take the weight transfer offer, the **transfer is final and cannot be renegotiated**

> Option to transfer 50% of prelim weight to final — Response due Wednesday, October 22 #653
>
> Giulia Guidi STAFF
> 2 days ago in General
>
> UNPIN STAR WATCHING 484 VIEWS
>
> 3
>
> Hi everyone,
>
> We know that exams can be stressful, and sometimes you may feel that your prelim performance doesn't fully reflect your understanding of the material. To help with this, we introduced an option to adjust the weighting of your prelim and final.

Prelim (Vol. II)

If you have taken the **regular** prelim, please see the **option to transfer 50% of prelim weight to final** on EdDiscussion:

- Regrade requests will be handled in bulk on Friday
- The weight transfer offer is **optional**
- If you decided to take the weight transfer offer, you must **let us know by October 22**
- If you decided to take the weight transfer offer, the **transfer is final and cannot be renegotiated**

PS: I have a water bottle and a pair of AirPod still in my office. I've found them after the regular prelim in KND 116. If you think either of them is yours, please email me describing the object (e.g., color, case, etc.). I'll bring them to Gates Lost & Found on Monday.

Final Exam

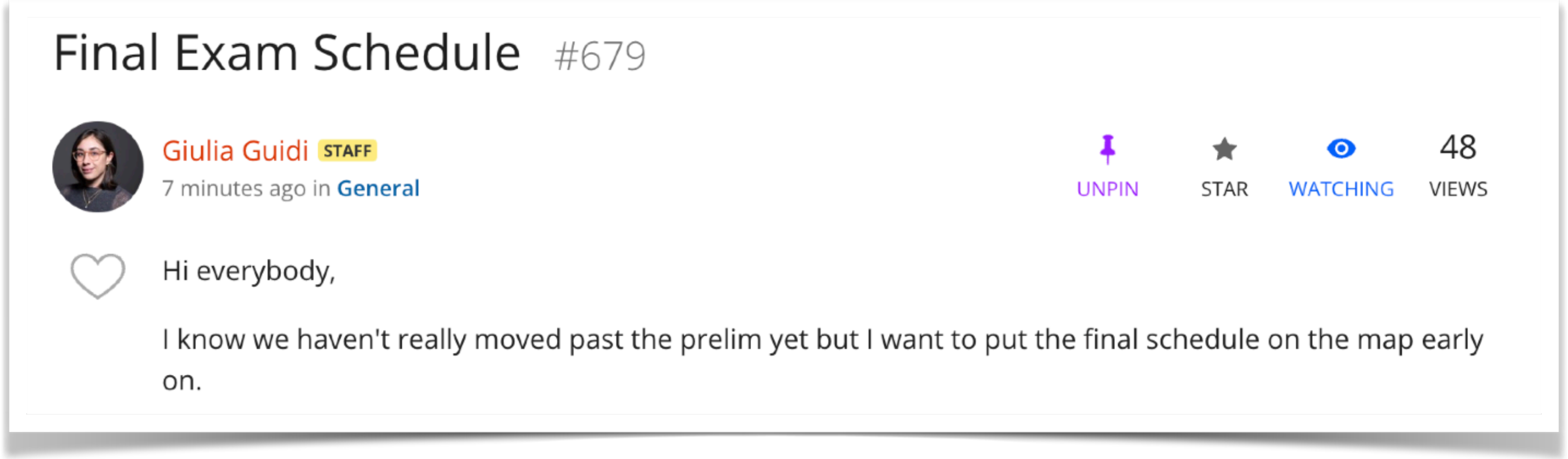
The **regular final** is on **Saturday, December 13, 7-9 PM**

The **early final** is on **Saturday, December 13, 4:30-6:30 PM**

The **make-up final** is on **Friday, Decemeber 12, 9-11 AM**

Check conflict early and let us know **by December 1** which exam you plan to take—**no** other make-up will be scheduled.

There's **no** weight transfer for the final or make-up final.



Course eval mid-semester survey due 10/17

Course eval mid-semester survey due October 17 #655

Giulia Guidi STAFF
2 days ago in General

UNPIN STAR WATCHING 233 VIEWS

Hi (again) everybody,

The **mid-semester course evaluation** survey is running from October 8 to October 17. You should have received the survey link by email. Per syllabus, **you can receive credit for completing it**.

It is common that most feedback can only be implemented between semesters, but we always look forward to reading your comments and seeing what we can improve for you as the semester progresses. Your feedback helps us make CS 3410 better, not just for future students, but also for your own learning experience right now.

Thank you for sharing your thoughts. We look forward to reading them!

Best,
Prof. Guidi

Participation

The "participation" segment of your grade has three main components:

- 4% for Lecture attendance, as measured by occasional Poll Everywhere polls.
- 4% for lab attendance, as recorded by the lab's instructors.
- 2% for surveys:
  - The introduction survey (on Gradescope) in the first week of class.
  - The mid-semester feedback survey.
  - The semester-end course evaluation.

We know that life happens, so you can miss up to 3 lab sections and 5 lectures without penalty.

Plan for Today

- Review of RISC-V Calling Convention and Stack
- Introduction to Cache, maybe?

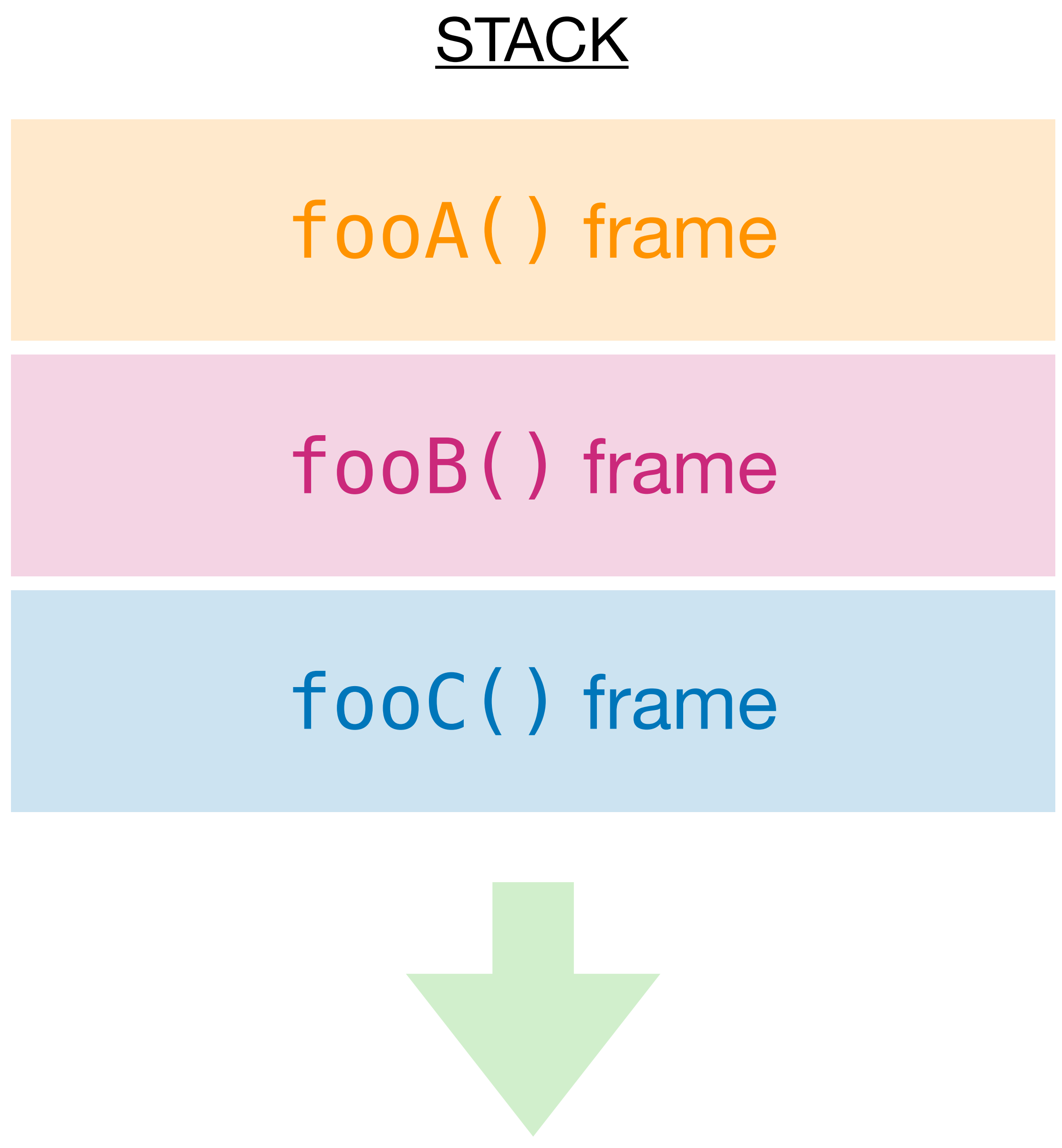
Review of stack

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

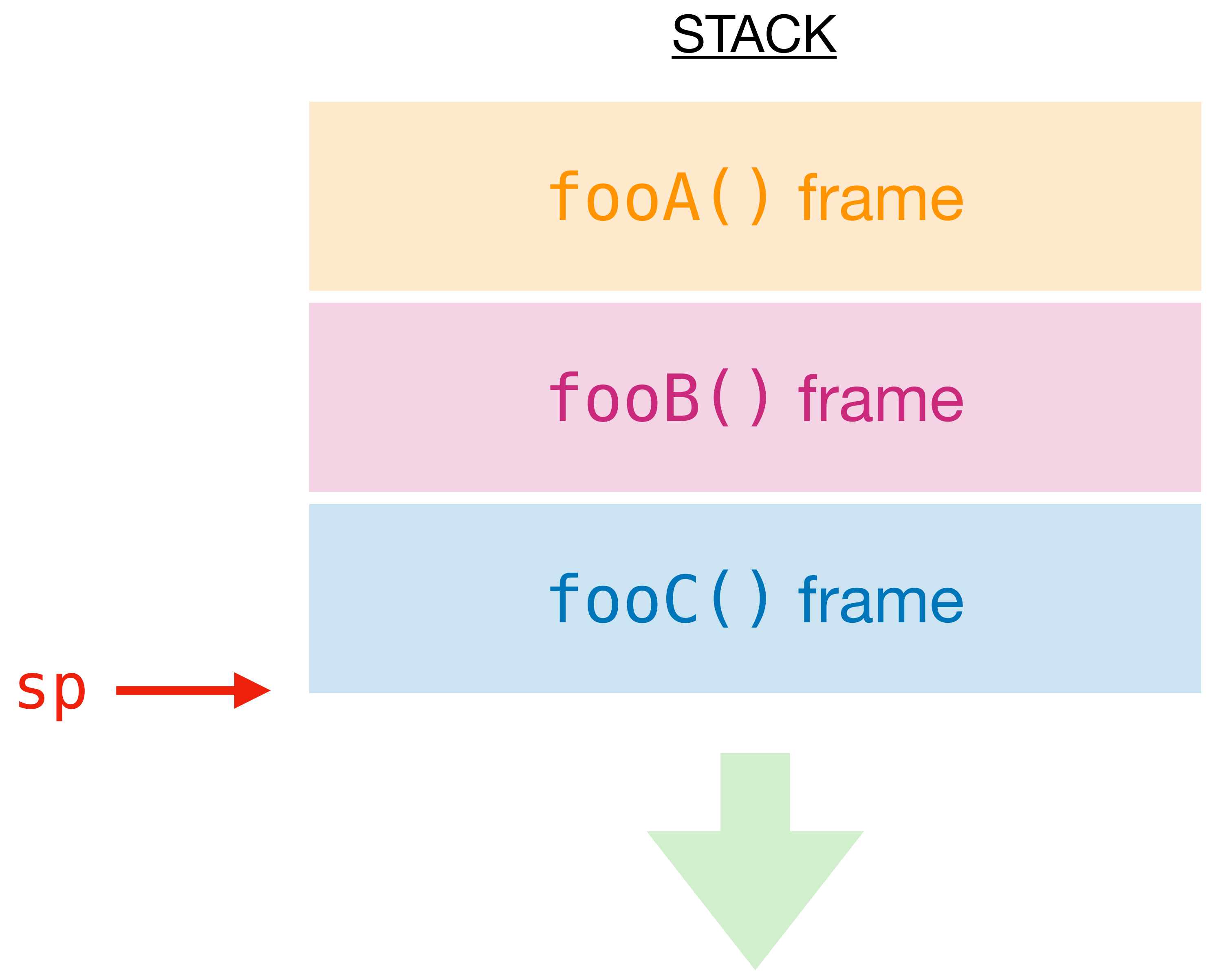
The stack pointer `sp` tells us the "top of the stack," i.e., the start of the current stack frame

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

fooB() { fooC(); }

fooC() { … }
```

If `fooC()` terminates:



Each stack **frame** includes:

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

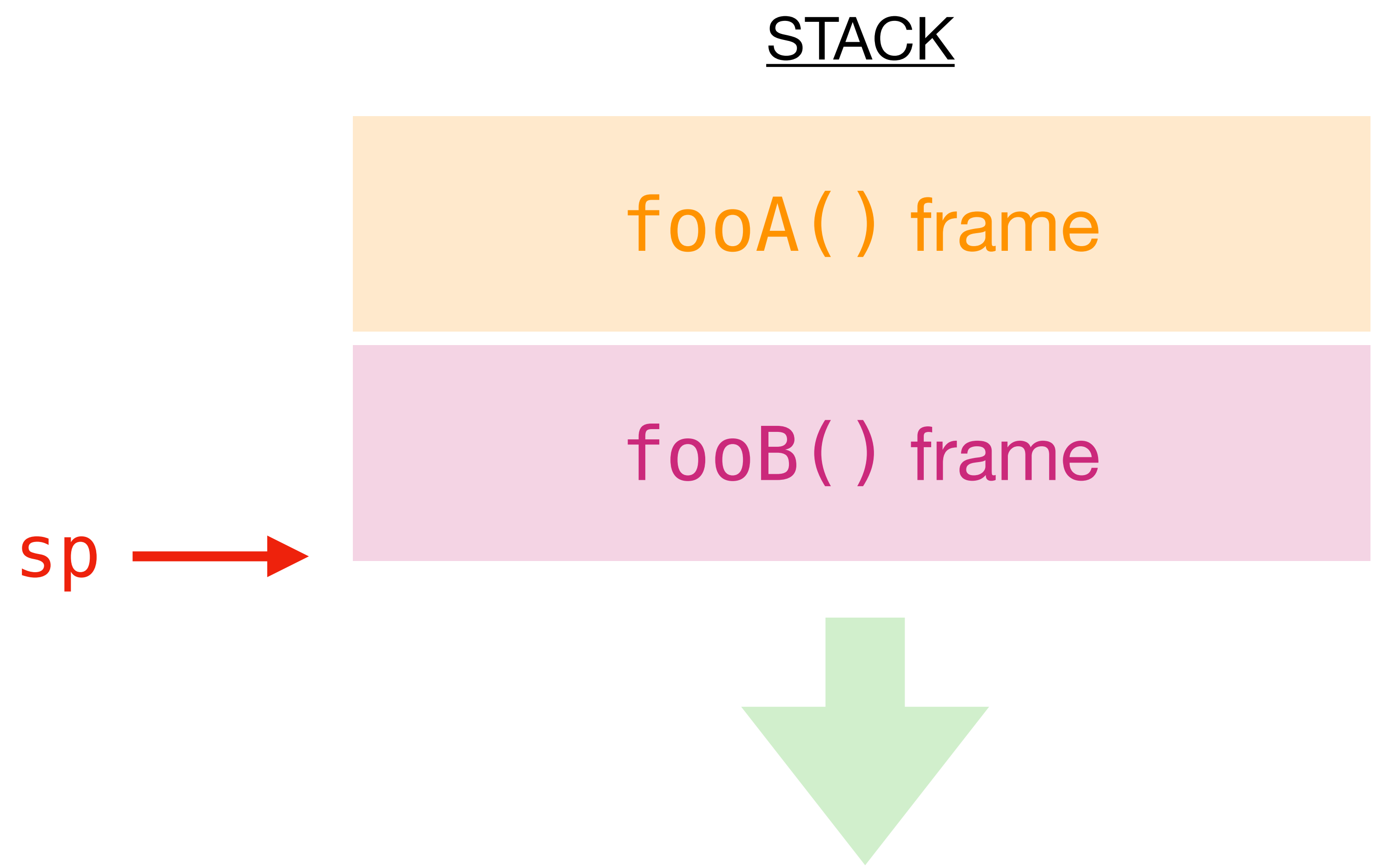
Stack

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

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

```
// incorrect
char *foo() {
   char string[32]; ...;
   return string;
}
```

because of how the stack **operates**!

Stack

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

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

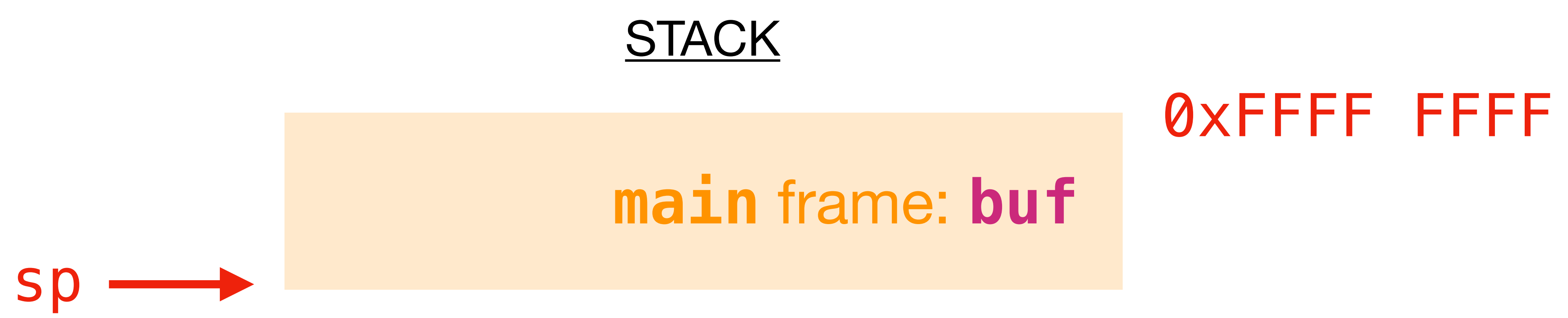
STACK

sp → 0xFFFF FFFF

Stack

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

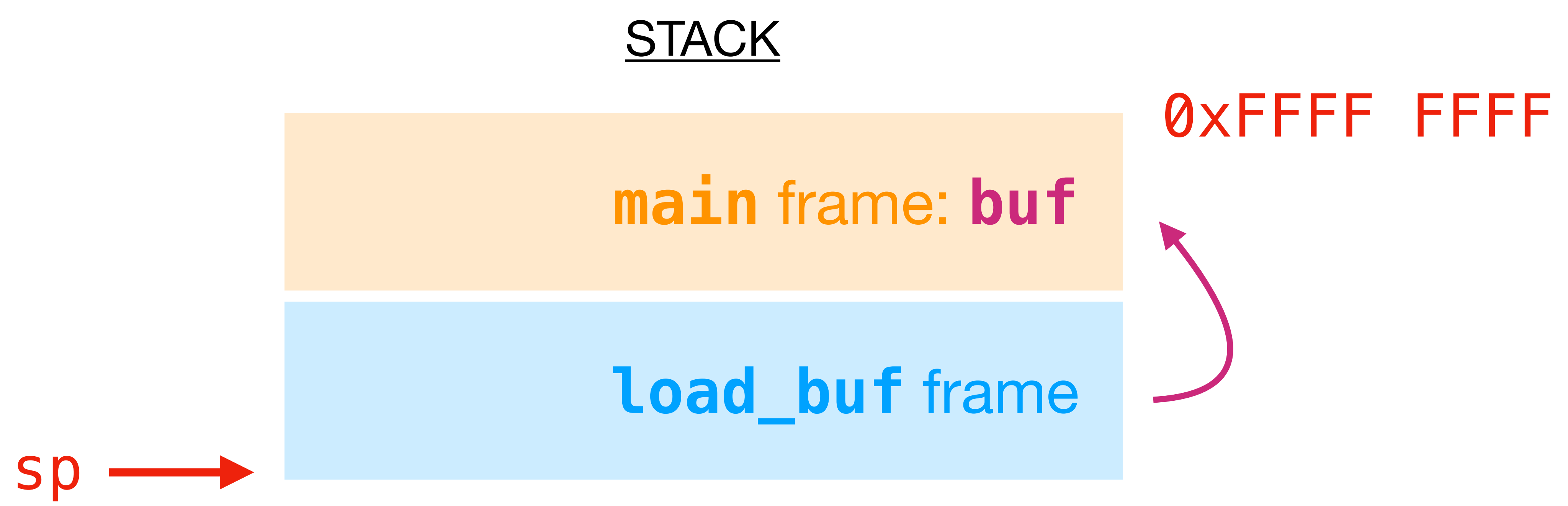
```
void load_buf(char *ptr,
              …) {
  …
};
int main() {
  …
  char buf[...];          PC
  load_buf(buf, BUFLEN);
  …
}
```



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

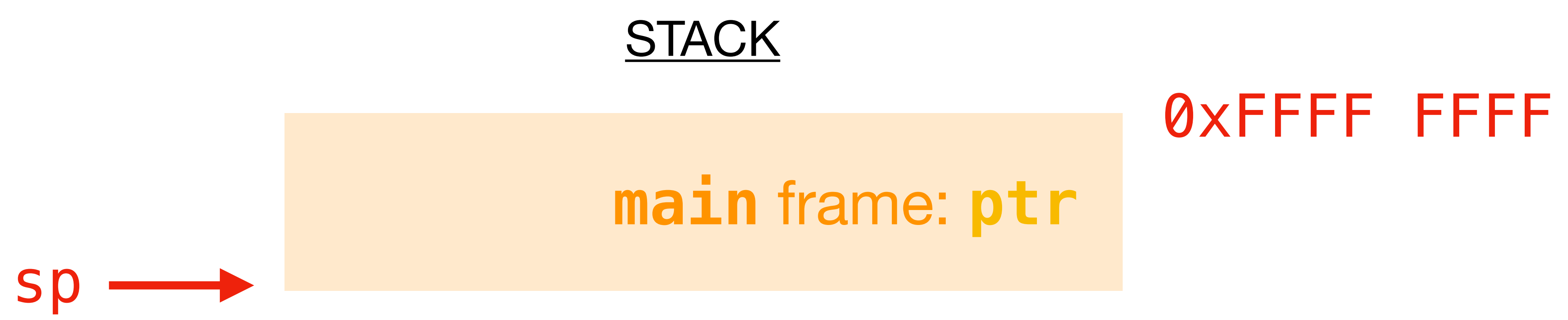
sp →

0xFFFF FFFF

Stack

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

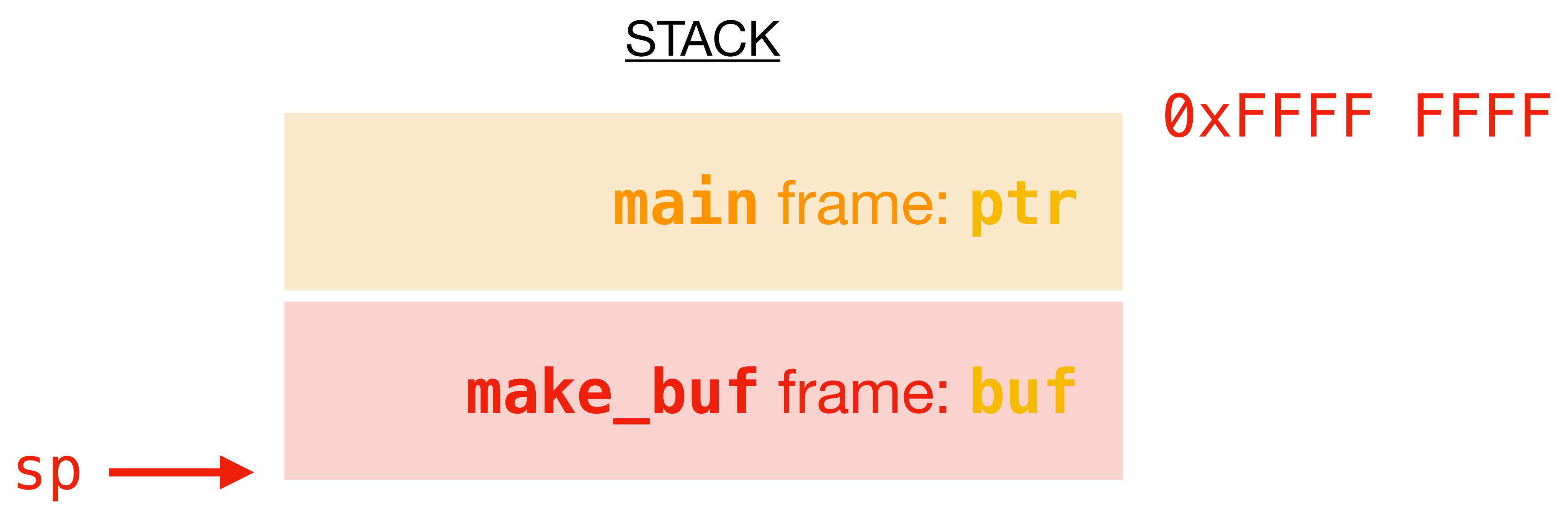
```
char *make_buf() {
    char buf[50];
    return buf;
}
void foo(…) {…}
int main(){
   char *ptr = make_buf();     ← PC
   foo(ptr);
   …
}
```



Stack

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

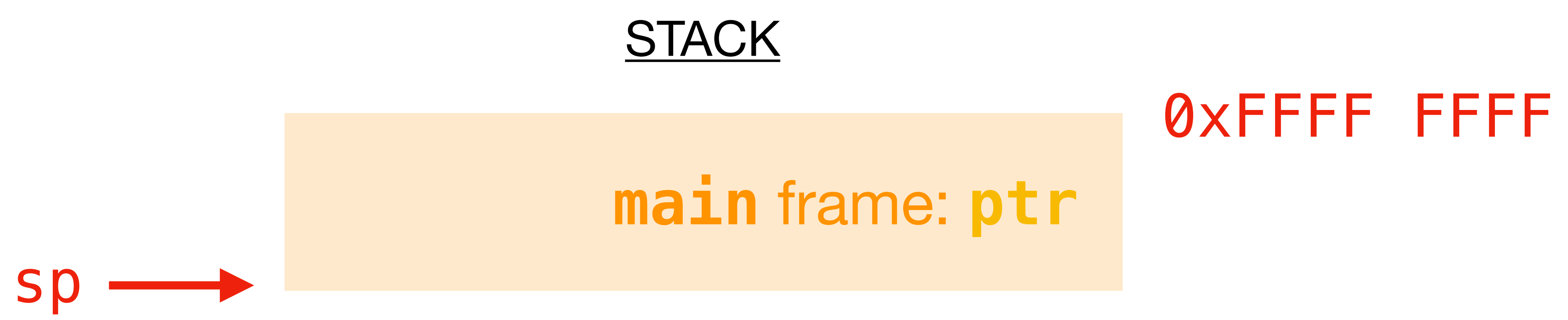
```
char *make_buf() {
    char buf[50];        <- PC
    return buf;
}
void foo(…) {…}
int main(){
   char *ptr = make_buf();
   foo(ptr);
   …
}
```



Stack

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

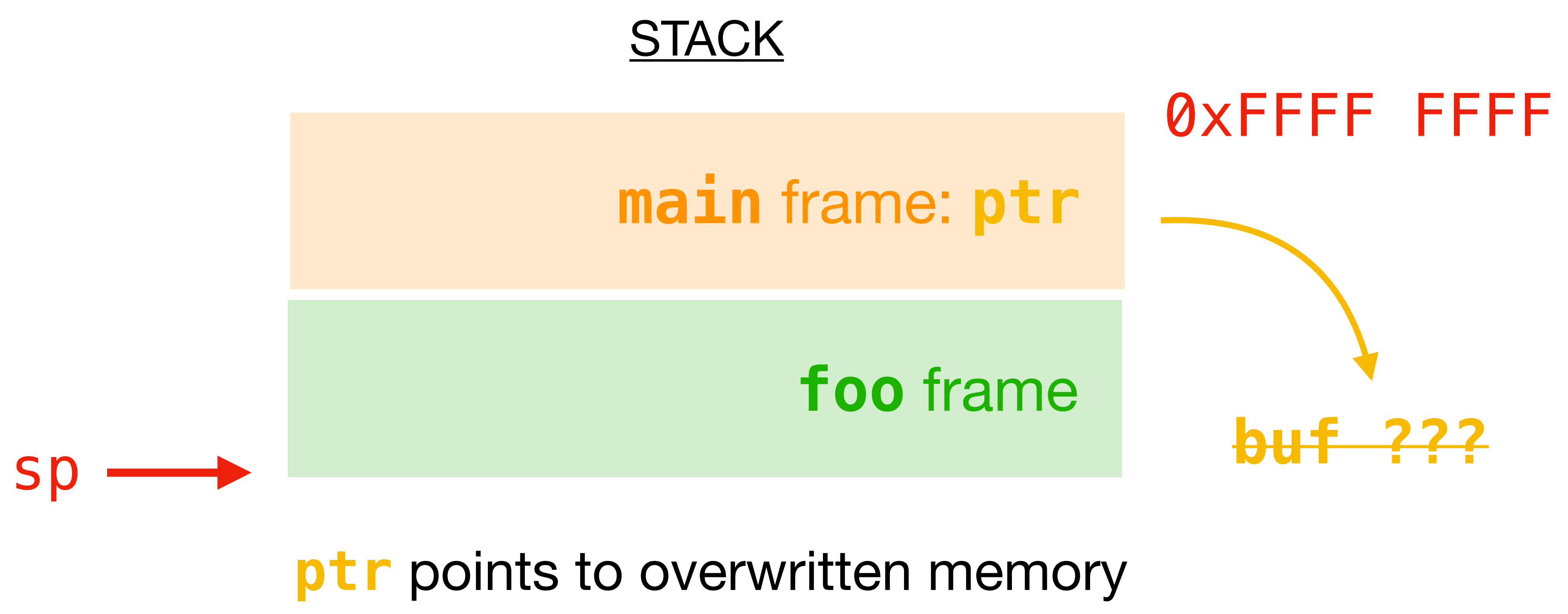
```
char *make_buf() {
    char buf[50];
    return buf;
}
void foo(…) {…}
int main(){
   char *ptr = make_buf();     <—— PC
   foo(ptr);
   …
}
```



Stack

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

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



Poll: The Stack Café

Given the C code below, if **`makeCoffee(3)`** is called, what happens on the stack?

```
void makeCoffee(int scoops) {
    int waterMl  = scoops * 100;     // 100 mL of water per scoop of coffee
    int brewTime = waterMl / 50;     // 1 second per 50 mL
    pourCoffee(brewTime);            // call another function
}
```

[PollEv.com /gguidi](https://PollEv.com/gguidi)

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

Poll: The Stack Café

Given the C code below, if `makeCoffee(3)` is called, what happens on the stack?

```
void makeCoffee(int scoops) {
    int waterMl  = scoops * 100;     // 100 mL of water per scoop of coffee
    int brewTime = waterMl / 50;     // 1 second per 50 mL
    pourCoffee(brewTime);            // call another function
}
```

→ `scoops`, `waterMl`, and `brewTime` have space reserved on the stack

- In **C**, space is allocated on the stack for every local variable, even if the compiler later might keep it in a register
- In hand-written assembly, you allocate stack space only when needed, such as saving the return address `ra` in a non-leaf function

Poll: The Stack C

Given the C code below, if **makeCoffee(3)** is called,

```
void makeCoffee(int scoops) {
    int waterMl  = scoops * 100;      // 100 mL of water p
    int brewTime = waterMl / 50;      // 1 second per 50 m
    pourCoffee(brewTime);             // call another func
}
```

→ `scoops`, `waterMl`, and `brewTime` have space reserved on th

Review of calling convention

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

Function Call

Let's go through the addOne function **execution**:

```
int addOne(int i) {
  return i + 1;
}
```

I'm **not** enforcing the 16-byte alignment

```
# prologue
addi sp, sp, -8  # push the stack frame aka allocate space on the stack to store 8 bytes
sd   ra, 0(sp)   # save the caller return address onto the stack
```

If we assumer 64-bit architecture, then ra takes 8 bytes

```
# body
addi a0, a0, 1   # i + 1 (i passed to addOne in a0) and place return value in a0
```

Can find this example in Lab 7

Function Call

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

```
int addOne(int i) {
  return i + 1;
}
```

I'm **not** enforcing the 16-byte alignment

```
# prologue
addi sp, sp, -8  # push the stack frame aka allocate space on the stack to store 8 bytes
sd   ra, 0(sp)   # save the caller return address onto the stack
```

If we assumer 64-bit architecture, then ra takes 8 bytes

```
# epilogue
ld   ra, 0(sp)   # restore the return address aka load the value from memory (stack) into ra
addi sp, sp, 8   # pop the stack frame aka deallocate space on the stack
ret              # return to caller, remember "ret = jr ra = jalr x0, 0(ra)"
```

Can find this example in Lab 7 (assuming 64-bit architecture)

Do I really need to store `ra` on the stack in `addOne`?

Function Call

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

```
int addOne(int i) {
  return i + 1;
}
```

`addOne` is a leaf function, meaning it does **not** call another fuction, so we do **not** actually need to store `ra` on the stack

The assembly can be just:

```
# body
addi a0, a0, 1   # i + 1 (i passed to addOne in a0) and place return value in a0
ret
```

Function Call

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

```
int addTwo(int i) {
  i = incrementOne(i)
  return i + 1;
}
```

```
int incrementOne(int x) {
  x = x + 1;
  return x;
}
```

3

```
# incrementOne body (leaf)
addi a0, a0, 1 # x = x + 1
ret            # leaf, ra untouched
# remember "ret = jr ra = jalr x0, 0(ra)"
```

1

```
# addTwo prologue
addi sp, sp, -8   # make stack space
sd   ra, 0(sp)    # save the caller return address onto the stack
```

2

```
# addTwo body (non-leaf)
call incrementOne  # overwrites ra
addi a0, a0, 1     # place return value in a0
```

4

```
# addTwo epilogue
ld   ra, 0(sp) # non-leaf, ra restored
addi sp, sp, 8
ret
```

I assumed 64-bit architecture and I did not enforce 16-byte alignment

Stack

```
int incrementOne(int x) {
  x = x + 1;
  return x;
}
int addTwo(int i) {
  i = incrementOne(i)
  return i + 1;
}
int main() {
    int z = 5;
    int y = addTwo(z);
    printf("%d\n", y);
    return 0;
}
```

sp →

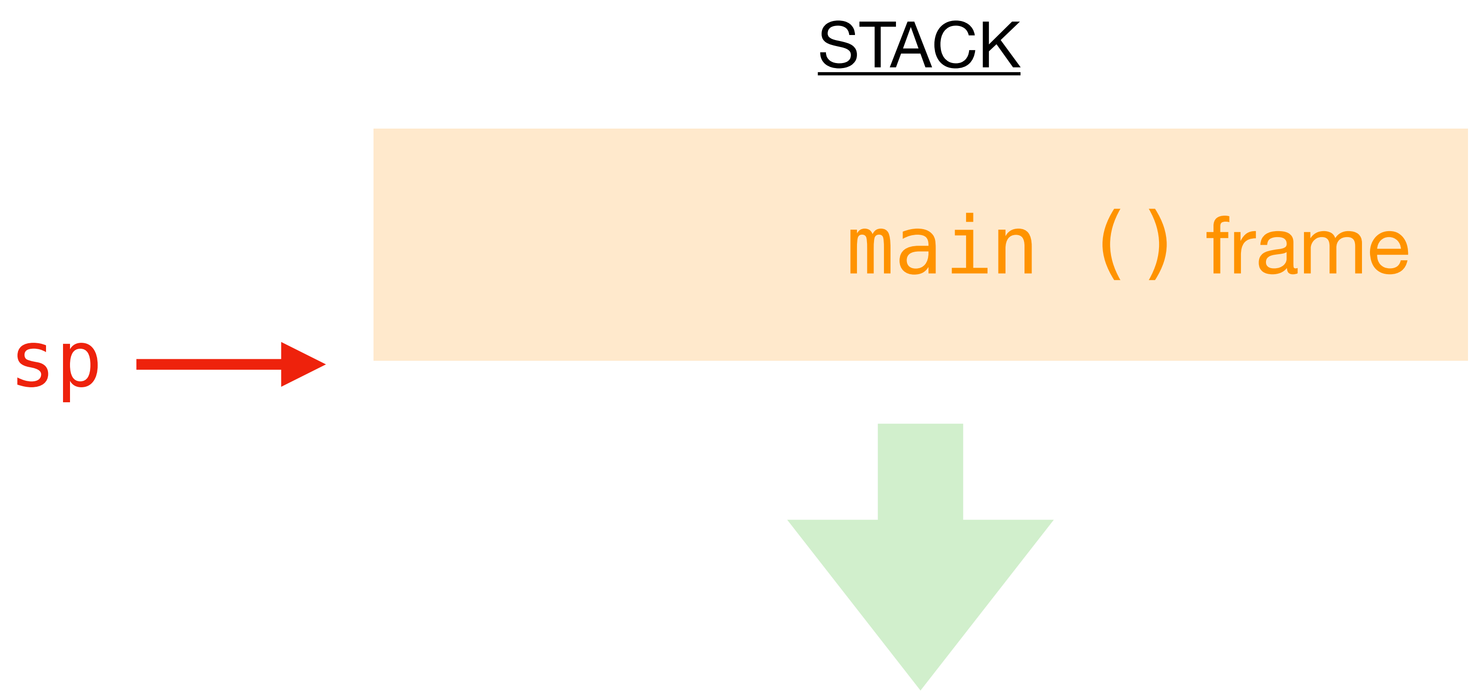
STACK

0xFFFF FFFF

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

Stack

```c
int incrementOne(int x) {
  x = x + 1;
  return x;
}
int addTwo(int i) {
  i = incrementOne(i)
  return i + 1;
}
int main() {          ← PC
    int z = 5;
    int y = addTwo(z);
    printf("%d\n", y);
    return 0;
}
```

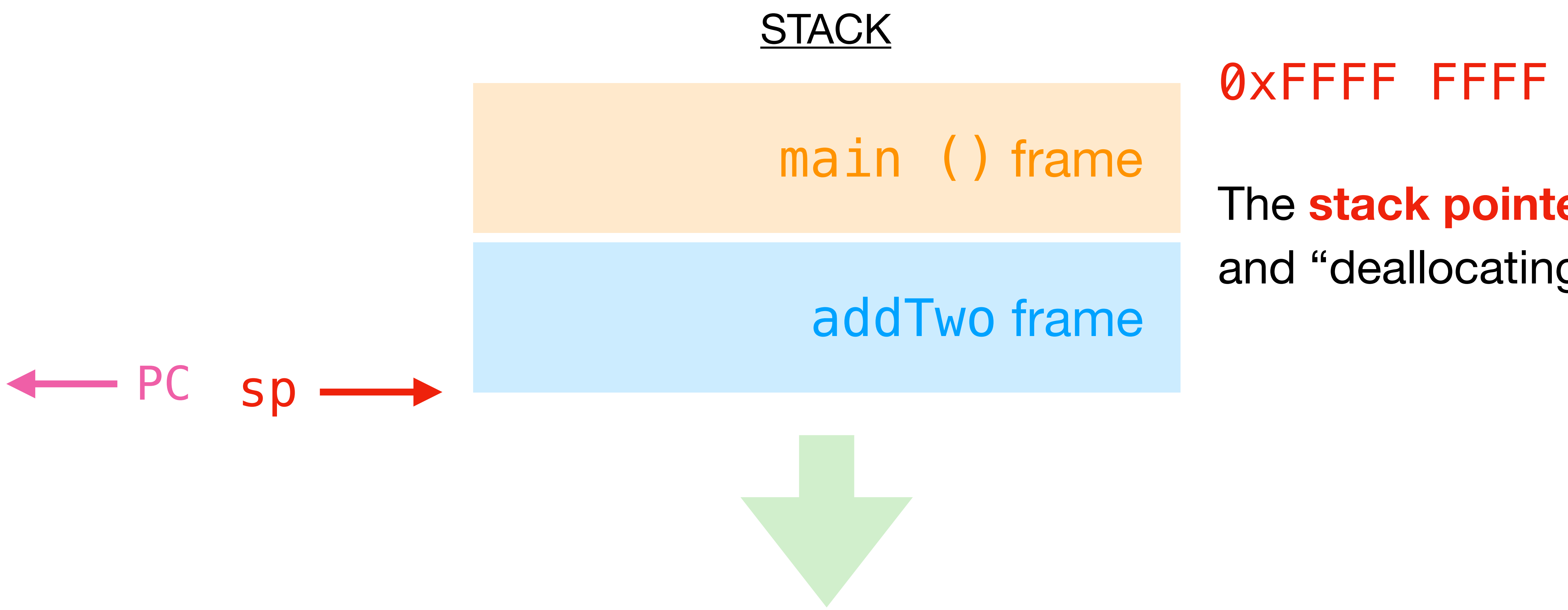


0xFFFF FFFF

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

Stack

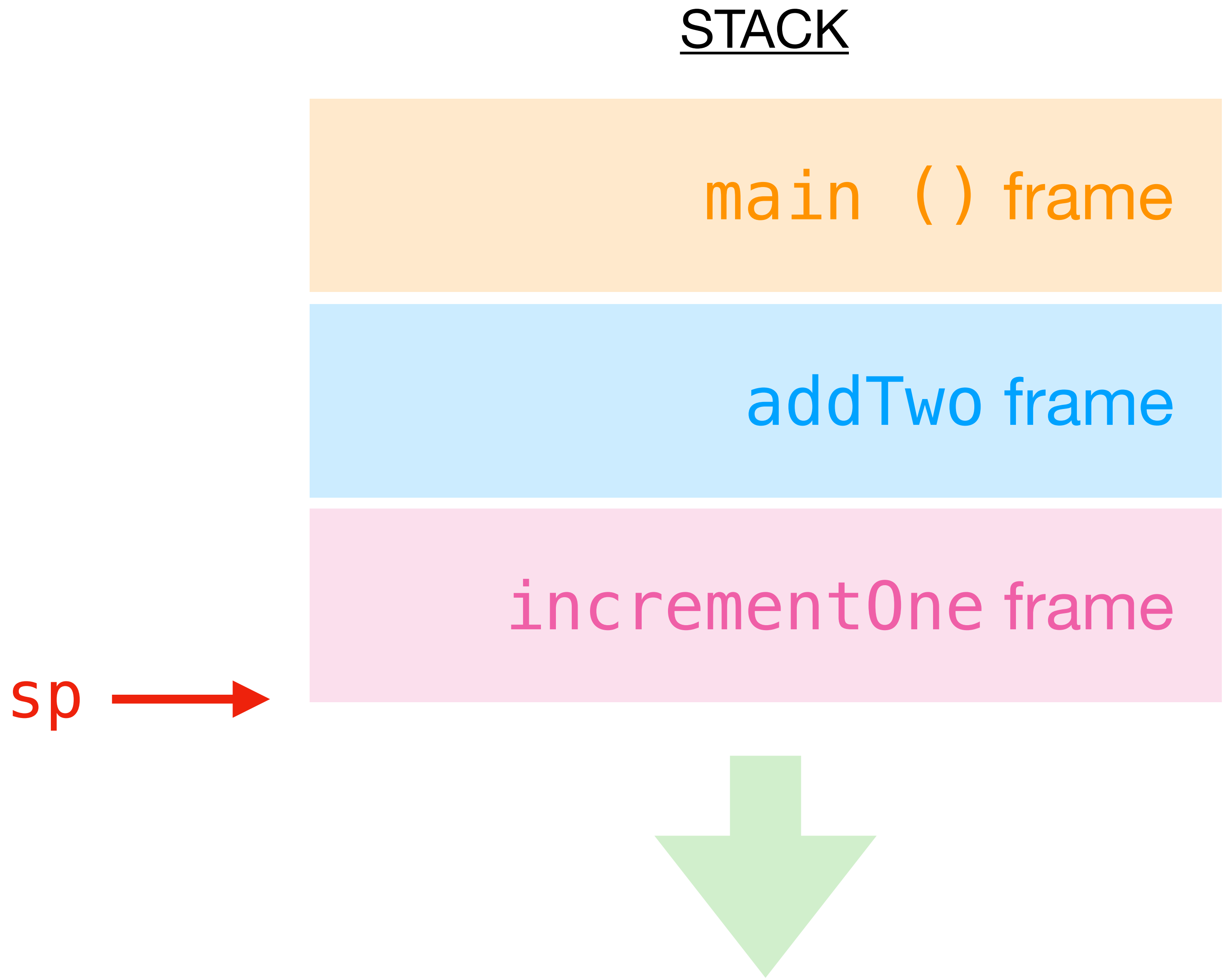
```
int incrementOne(int x) {
  x = x + 1;
  return x;
}
int addTwo(int i) {
  i = incrementOne(i)
  return i + 1;
}
int main() {
    int z = 5;
    int y = addTwo(z);
    printf("%d\n", y);
    return 0;
}
```



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

Stack

```
int incrementOne(int x) {    ← PC
  x = x + 1;
  return x;
}
int addTwo(int i) {
  i = incrementOne(i)
  return i + 1;
}
int main() {
    int z = 5;
    int y = addTwo(z);
    printf("%d\n", y);
    return 0;
}
```



0xFFFF FFFF

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

Function Call Example

a0 a1 a2 a3

g, h, i, and j are **arguments** so they are stored in a0–a7 (a0–a3, in this case)

```
int Leaf (int g, int h, int i, int j)
{
  int f;  → s0
  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
```

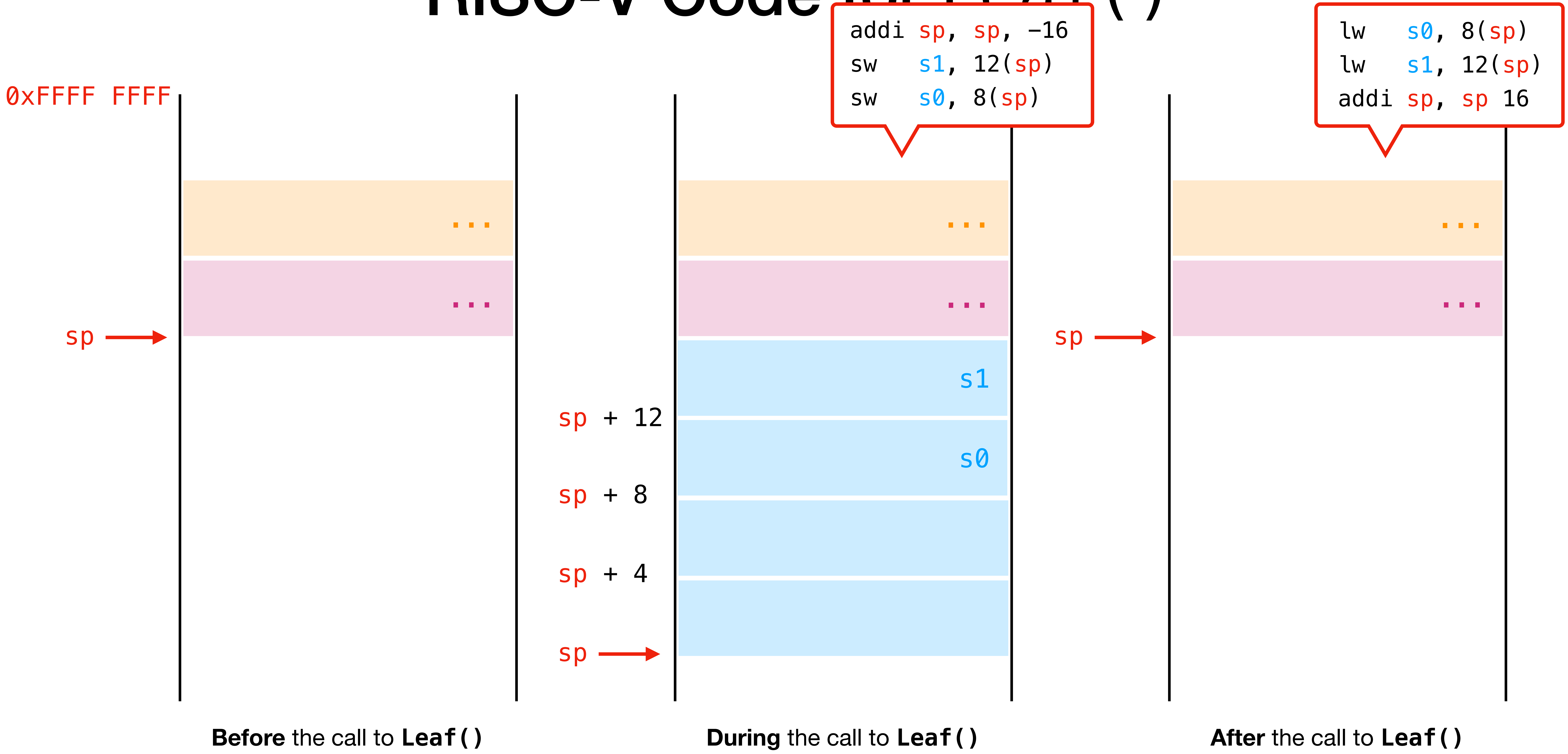
I'm **enforcing** the 16-byte alignment

callee-saved

callee-saved

Can find this in lecture 13 (assuming 32-bit architecture)

RISC-V Code for `Leaf()`



Can find this in lecture 13 (assuming 32-bit architecture)

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             caller-saved
    add   t1, a2, a3   # t1 = i + j             caller-saved
    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

Poll

Given the C code below, how many **return addresses** **`ra`** exist during execution?

```c
int foo() {
    return 1;
}
int bar() {
    foo();
    return 2;
}
int main() {
    bar();
    return 0;
}
```

[PollEv.com /gguidi](https://PollEv.com/gguidi)

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

Return Address

Given the C code below, **fill in** the table below**:**

```
int foo() {
    return 1;
}
int bar() {
    foo();
    return 2;
}
int main() {
    bar();
    return 0;
}
```

#	**Program execution**	**ra points to:**
1	`main()` calls `bar()`	
2	`bar()` calls `foo()`	
3	`foo()` returns	
4	`bar()` returns	
5	`main()` returns	

Return Address

Given the C code below, **fill in** the table below**:**

```
int foo() {
    return 1;
}
int bar() {
    foo();
    return 2;
}
int main() {
    bar();
    return 0;
}
```

#	**Program execution**	**ra points to:**
1	`main()` calls `bar()`	The address after `call bar` (return to `main`)
2	`bar()` calls `foo()`	
3	`foo()` returns	
4	`bar()` returns	
5	`main()` returns	

Return Address

Given the C code below, **fill in** the table below**:**

```
int foo() {
    return 1;
}
int bar() {
    foo();
    return 2;
}
int main() {
    bar();
    return 0;
}
```

#	**Program execution**	**ra points to:**
1	`main()` calls `bar()`	The address after `call bar` (return to `main`)
2	`bar()` calls `foo()`	The address after `call foo` (return to `bar`)
3	`foo()` returns	
4	`bar()` returns	
5	`main()` returns	

Return Address

Given the C code below, **fill in** the table below**:**

```c
int foo() {
    return 1;
}
int bar() {
    foo();
    return 2;
}
int main() {
    bar();
    return 0;
}
```

#	Program execution	ra points to:
1	`main()` calls `bar()`	The address after `call bar` (return to `main`)
2	`bar()` calls `foo()`	The address after `call foo` (return to `bar`)
3	`foo()` returns	Back to `bar`
4	`bar()` returns	
5	`main()` returns	

Return Address

Given the C code below, **fill in** the table below:

```
int foo() {
    return 1;
}
int bar() {
    foo();
    return 2;
}
int main() {
    bar();
    return 0;
}
```

#	Program execution	ra points to:
1	`main()` calls `bar()`	The address after `call bar` (return to `main`)
2	`bar()` calls `foo()`	The address after `call foo` (return to `bar`)
3	`foo()` returns	Back to `bar`
4	`bar()` returns	Back to `main`
5	`main()` returns	

Return Address

Given the C code below, **fill in** the table below:

```
int foo() {
    return 1;
}
int bar() {
    foo();
    return 2;
}
int main() {
    bar();
    return 0;
}
```

#	Program execution	ra points to:
1	`main()` calls `bar()`	The address after `call bar` (return to `main`)
2	`bar()` calls `foo()`	The address after `call foo` (return to `bar`)
3	`foo()` returns	Back to `bar`
4	`bar()` returns	Back to `main`
5	`main()` returns	Program ends

Return Address

Given the C code and assembly below, what **if we comment out** lines 1–2 and 4–5?

```c
int foo() {
    return 1;
}
int bar() {
    foo();
    return 2;
}
int main() {
    bar();
    return 0;
}
```

```
bar:
1     addi sp, sp, -8
2     sd   ra, 0(sp)
3     call foo
4     ld   ra, 0(sp)
5     addi sp, sp, 8
6     ret
```

In this slide, we assume we are working on a 64-bit architecture

Return Address

Given the C code and assembly below, what **if we comment out** lines 1–2 and 4–5?

```c
int foo() {
    return 1;
}
int bar() {
    foo();
    return 2;
}
int main() {
    bar();
    return 0;
}
```

```
bar:
1     addi sp, sp, -8
2     sd   ra, 0(sp)
3     call foo
4     ld   ra, 0(sp)
5     addi sp, sp, 8
6     ret
```

The call to `foo()` overwrite `ra` (which was holding the return address for `main`)

Thus, when `bar()` executes `ret`, it jumps to the <u>wrong</u> place. Can crash or runs garbage

In this slide, we assume we are working on a 64-bit architecture

Return Address

Given the C code and assembly below, what **if we comment out** lines 1–2 and 4–5?

```c
int foo() {
    return 1;
}
int bar() {
    foo();
    return 2;
}
int main() {
    bar();
    return 0;
}
```

```
bar:
1    addi sp, sp, -8
2    sd   ra, 0(sp)
3    call foo
4    ld   ra, 0(sp)
5    addi sp, sp, 8
6    ret
```

Key Takeaway:

Every time you call a function (correctly), ra changes — unless it's a **leaf** fuction

If your function calls another, you must save ra or you'll lose your way back

In this slide, we assume we are working on a 64-bit architecture

Nested Procedures

```
int main() {
    int result = sumSquare(3, 4);
    return 0;
}

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

main:
  li   a0, 3          # x = 3
  li   a1, 4          # y = 4
  jal  ra, sumSquare  # a0 contains
result (3*3 + 4*4 = 25)
  ret
```

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

Nested Procedures

```
int main() {
    int result = sumSquare(3, 4);
    return 0;
}

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

```
main:
  li    a0, 3             # x = 3
  li    a1, 4             # y = 4
  jal   ra, sumSquare     # a0 contains result (3*3 + 4*4 = 25)
  ret
```

`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 # jump to mult(x, x)
    …
    jr  ra       # return to main

mult:
    # mult function defined elsewhere
```

Nested Procedures

I need to save **sumSquare** return address on the stack before call to **mult**

```
sumSquare:
     addi sp, sp, -16       # Get some space on the stack
     sw   ra, 12(sp)        # Can store ra (poiting to main) 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
```

In this slide, we assume a 32-bit architecture and we're enforcing 16-byte alignment

Nested Procedures

I need to save **`sumSquare`** return address on the stack before call to **`mult`**

```
sumSquare:
      addi sp, sp, -16        # Get some space on the stack
      sw   ra, 12(sp)         # Can store ra on the stack
      sw   a1, 8(sp)          # Can store y on the stack (I could have copied it into t0 directly)

      mv   a1, a0             # Set a1 = a0 so mult(x, x) gets correct second argument
      jal  ra, mult           # Call mult(x, x) takes a1 and a0 as input

      # now a0 contains (x * x)

      lw   t0, 8(sp)          # Pop y from the stack into temporary reg t0
      add  a0, a0, t0         # a0 = (x * x) + y; (x * x) is already stored in a0

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

mult:
    # mult function defined elsewhere
```

In this slide, we assume a 32-bit architecture and we're enforcing 16-byte alignment

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