C Refresher & C to RISC-V

Introduction to Computer Systems, Fall 2024

Instructors: Joel Ramirez Travis McGaha

Head TAs: Adam Gorka Daniel Gearhardt
Ash Fujiyama Emily Shen

TAs:

Ahmed Abdellah
Angie Cao
August Fu
Caroline Begg
Cathy Cao
Claire Lu
Eric Sungwon Lee
Ethan Weisberg
Garrett O'Malley Kirsch
Hassan Rizwan
Iain Li
Jerry Wang
Juan Lopez
Keith Mathe
Maya Huizar
Meghana Vasireddy
Perrie Quek
Sidharth Roy
Sydnie-Shea Cohen
Vivi Li
Yousef AlRabiah

Poll Everywhere

pollev.com/cis2400

- How are you? Any Questions?

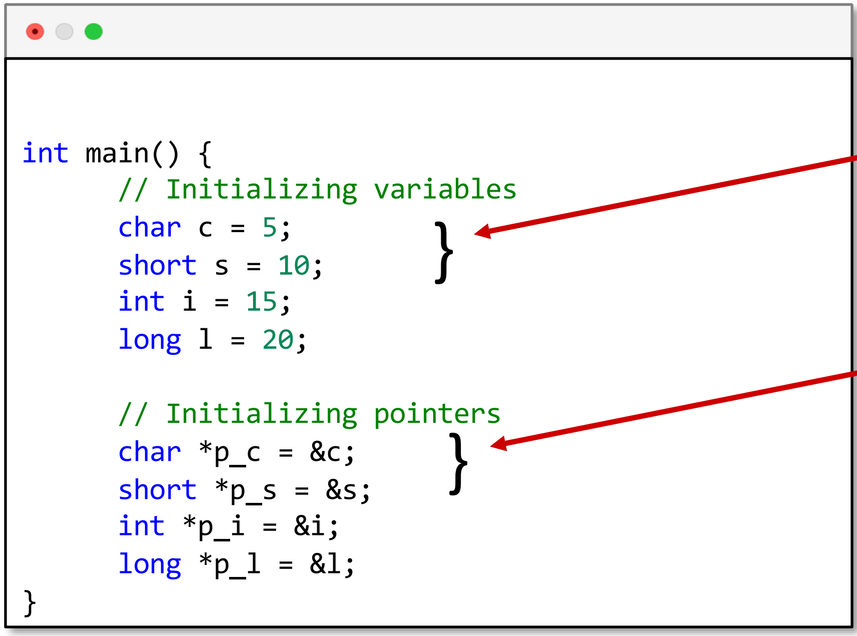
Logistics

Lecture Outline

- C Refresher
  - Pointers, Memory Allocation
  - Macros:
    - Constants vs. Function-like Macros
    - Evaluation Pitfalls
- C to RISC-V
  - Purpose of Translating C to Assembly
  - Performance, Memory Control
- Function Calls in RISC-V
  - The Registers
  - Argument Passing
  - Return Values
  - The Frame and Returning from Functions

Pointers Revisited

- Pointers are used to refer to ***locations in Memory***
- Pointers have a corresponding type ***if***
  - We know what is stored at the location in memory.
  - We know how ***large*** it is.

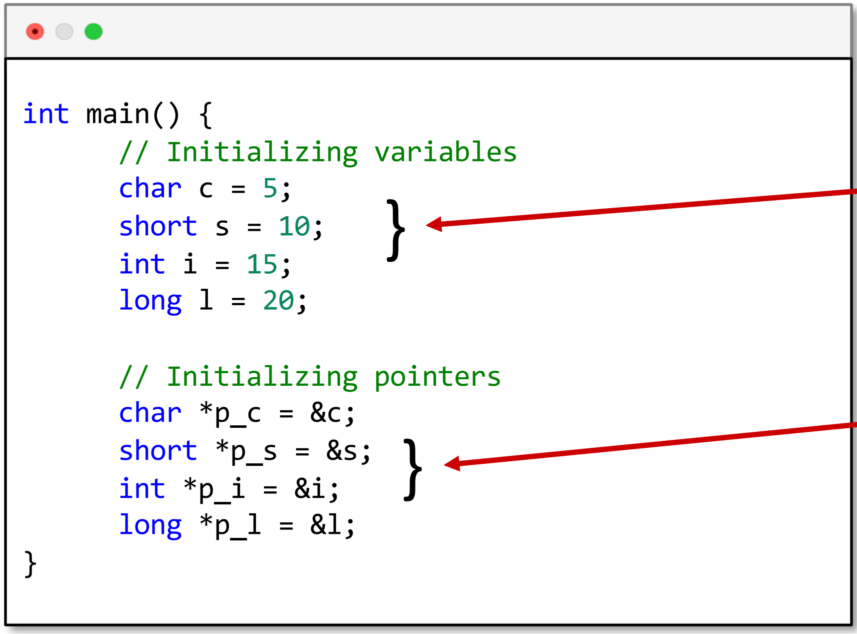


```
int main() {
      // Initializing variables
      char c = 5;           }
      short s = 10;
      int i = 15;
      long l = 20;

      // Initializing pointers
      char *p_c = &c;        }
      short *p_s = &s;
      int *p_i = &i;
      long *p_l = &l;
}
```

All types here have different sizes

All types here have the same size

Pointers Revisited



```c
int main() {
      // Initializing variables
      char c = 5;
      short s = 10;
      int i = 15;
      long l = 20;

      // Initializing pointers
      char *p_c = &c;
      short *p_s = &s;
      int *p_i = &i;
      long *p_l = &l;
}
```

All types here have different sizes

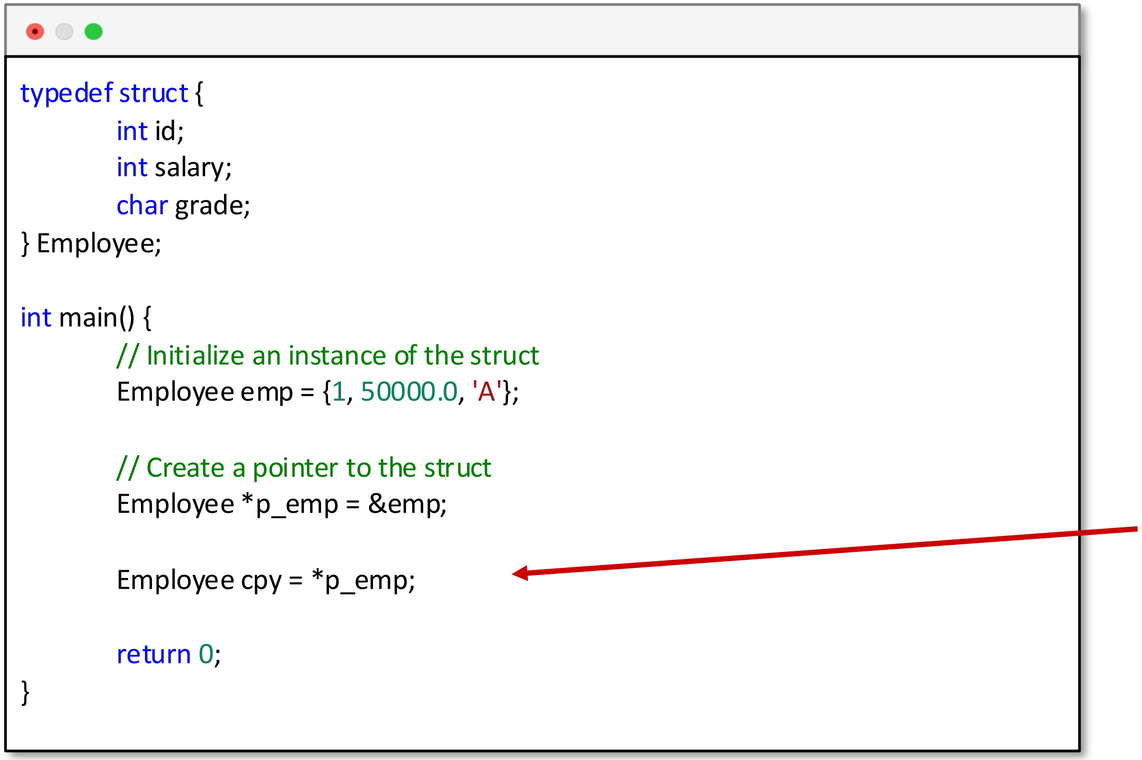
All types here have the same size

Why?

The ‘&’ operator returns the address of the corresponding variable.

On a given machine, all addresses are the same size—usually 32 or 64 bits, depending on the architecture. The difference arises when we dereference an address, as we need to know both how much memory to access and how the values are organized within that memory.

Pointers Revisited



```
typedef struct {
        int id;
        int salary;
        char grade;
} Employee;

int main() {
        // Initialize an instance of the struct
        Employee emp = {1, 50000.0, 'A'};

        // Create a pointer to the struct
        Employee *p_emp = &emp;

        Employee cpy = *p_emp;

        return 0;
}
```

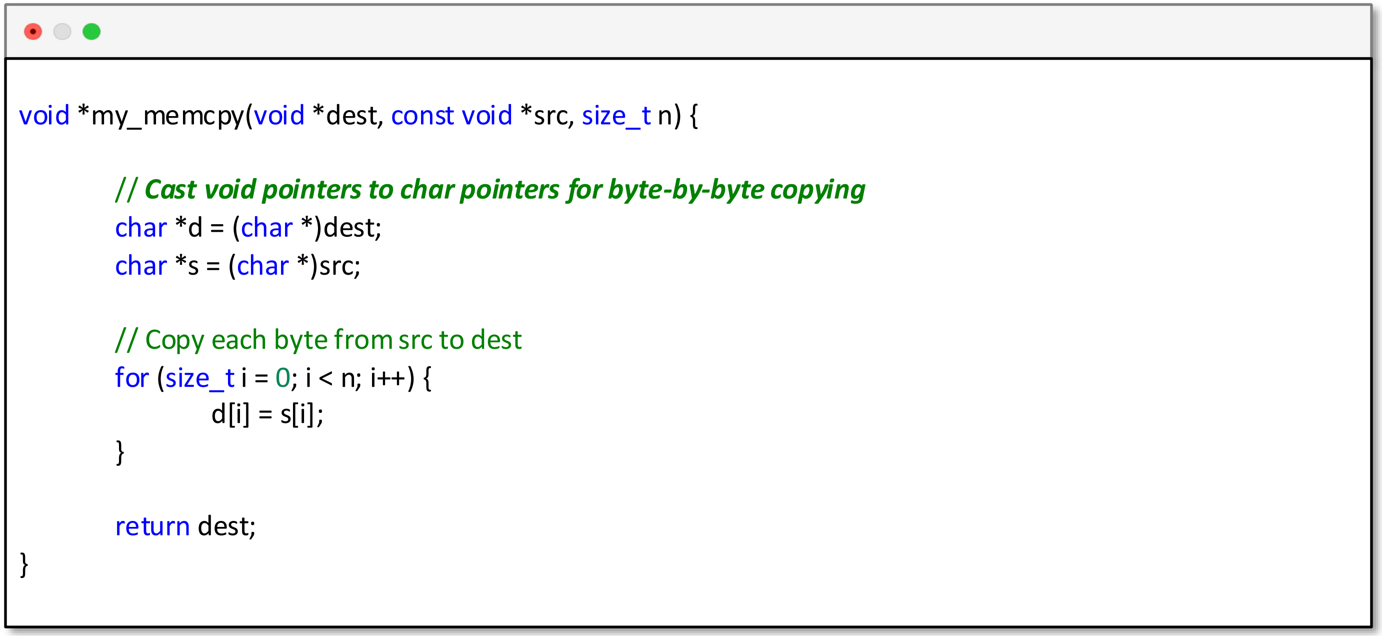
As an example, we might receive *9 bytes* **of data when we dereference this.**

Is `id` the first 4 bytes, `salary` the next 4, and `grade` the next byte? ***We need to know.***

Understanding how values are organized within memory is then imperative in understanding how structs and other larger data structures are populated when their memory is retrieved.

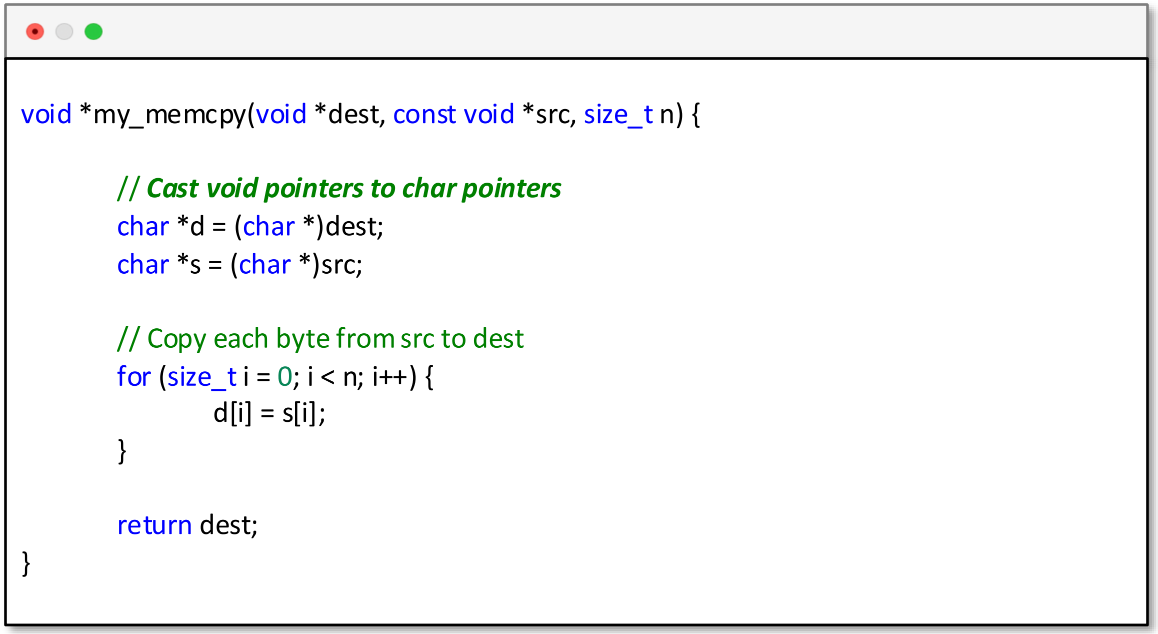
Pointers Revisited

- Pointers are used to refer to ***locations in Memory***
- Pointers ***do not*** have a corresponding type ***if***
  - we’re treating memory as just an array of bytes, without regard to the specific type of data stored there *(e.g. memcpy, duplicating a file)*
  - Or to support generics *(e.g. qsort, not important for you yet)*



```
void *my_memcpy(void *dest, const void *src, size_t n) {

    // Cast void pointers to char pointers for byte-by-byte copying
    char *d = (char *)dest;
    char *s = (char *)src;

    // Copy each byte from src to dest
    for (size_t i = 0; i < n; i++) {
        d[i] = s[i];
    }

    return dest;
}
```

Pointers Revisited



```
void *my_memcpy(void *dest, const void *src, size_t n) {

        // Cast void pointers to char pointers
        char *d = (char *)dest;
        char *s = (char *)src;

        // Copy each byte from src to dest
        for (size_t i = 0; i < n; i++) {
                d[i] = s[i];
        }

        return dest;
}
```

Void * pointers allow us to pass addresses around when we are agnostic about what they store

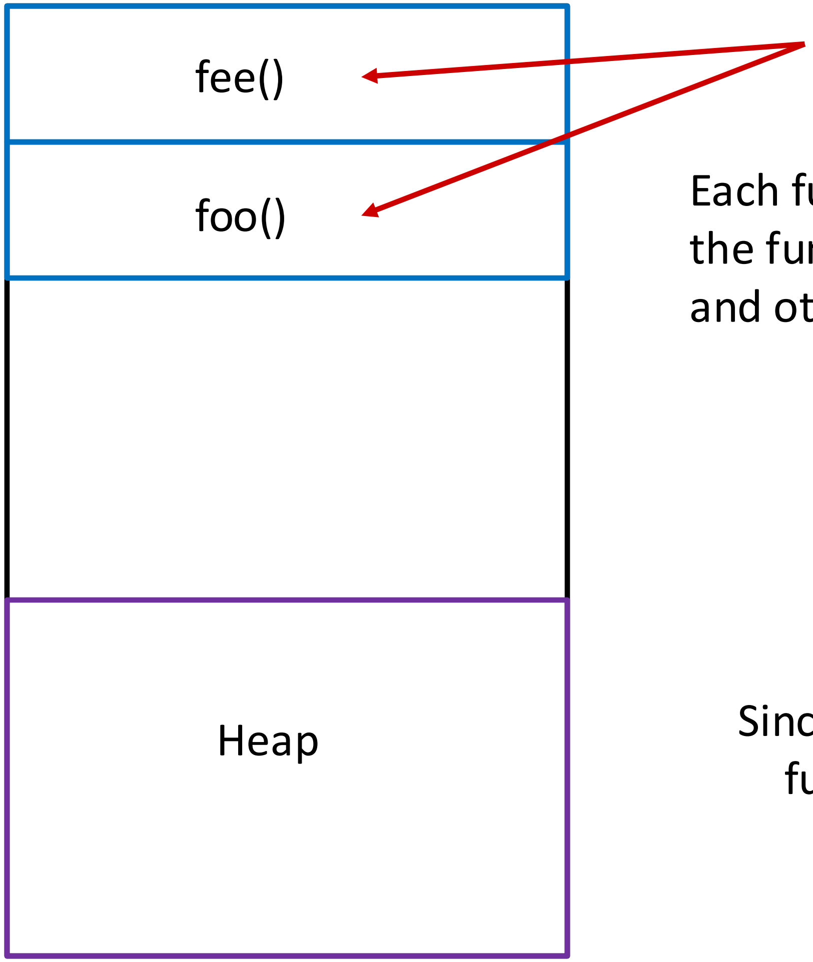
We’re limited by the “type system” when retrieving information, so we need to cast values to char when we want to access individual bytes and store them elsewhere.

Unfortunately, this limits us to perform ***memory moves a singular byte*** at a time.

Heap Revisited

- Functions like **`malloc`** and **`free`** let us manage memory directly.
  - If you need to store something outside ‘the stack’, you can use malloc to allocate space on the heap.
  - When you're finished with that memory, you use free to release it back to the system.

Heap Revisited



Each function has its ***own*** respective ***frame***.

Each functions frame holds, typically, arguments for the function, variables created within the function, and other state (which we'll see soon).

Since frames are temporary and cleared when a function returns, we store information that needs to persist in the heap.

Heap Revisited

- Functions like **`malloc`** and **`free`** let us manage memory directly.
- Functions the interact with the heap are:

malloc(), free(), calloc(), and realloc()

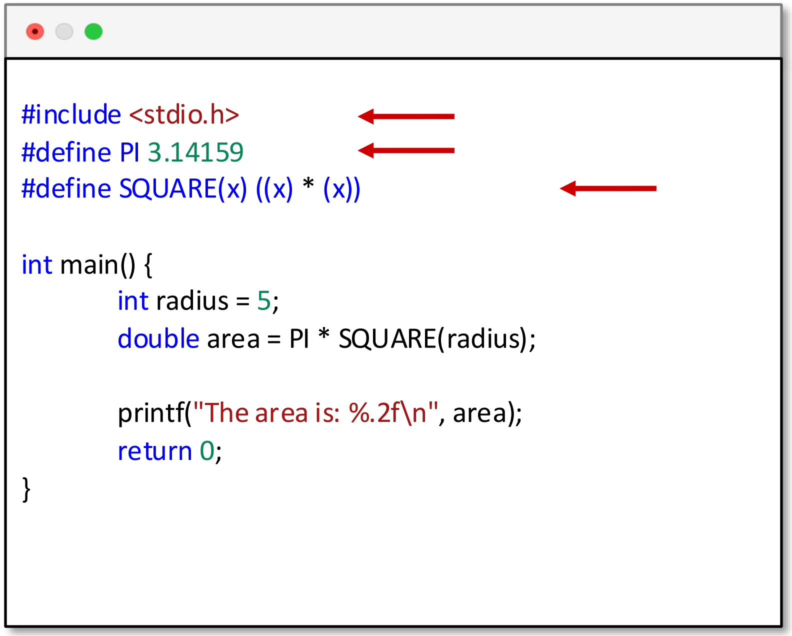
When you want all memory allocated to be ‘zero-d’ out.

When you want more memory after the heap gave you some already.

Macros

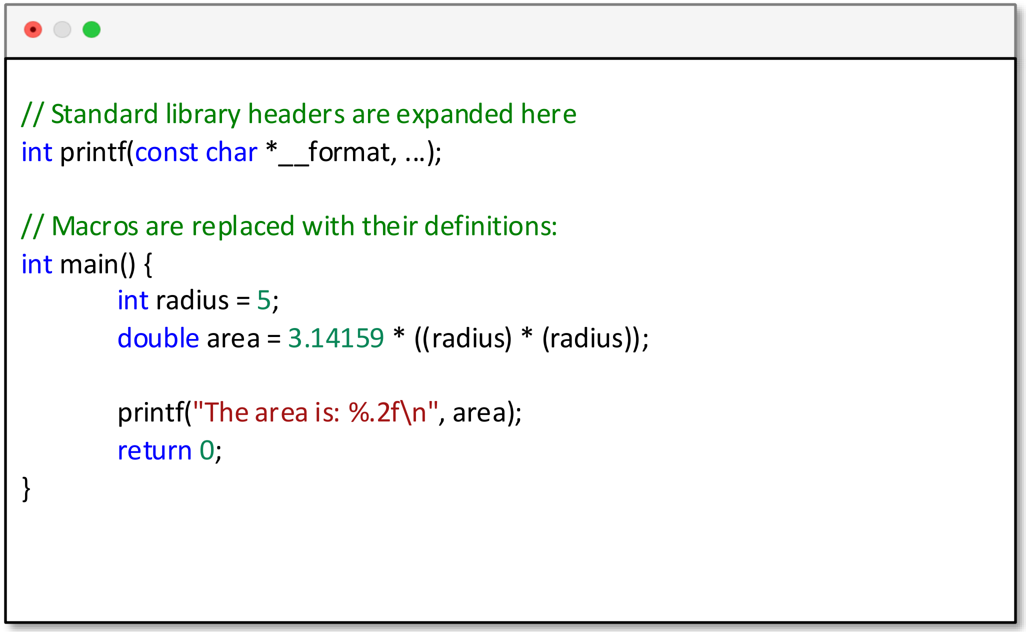
- We've seen how Macros are handled by the Pre-Processor.

`file.c`



```
#include <stdio.h>
#define PI 3.14159
#define SQUARE(x) ((x) * (x))

int main() {
        int radius = 5;
        double area = PI * SQUARE(radius);

        printf("The area is: %.2f\n", area);
        return 0;
}
```

clang-15 -E file.c -o file.i



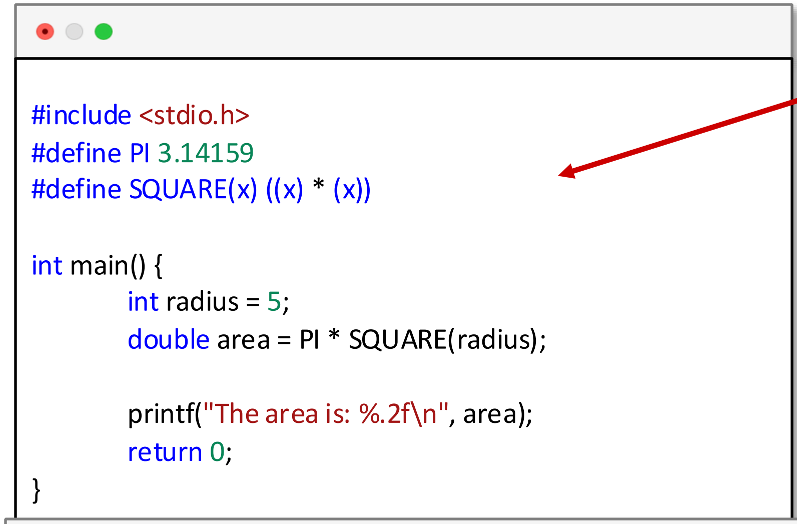
```
// Standard library headers are expanded here
int printf(const char *__format, ...);

// Macros are replaced with their definitions:
int main() {
        int radius = 5;
        double area = 3.14159 * ((radius) * (radius));

        printf("The area is: %.2f\n", area);
        return 0;
}
```

`printf` declaration is here now and `PI` is replaced by 3.14159

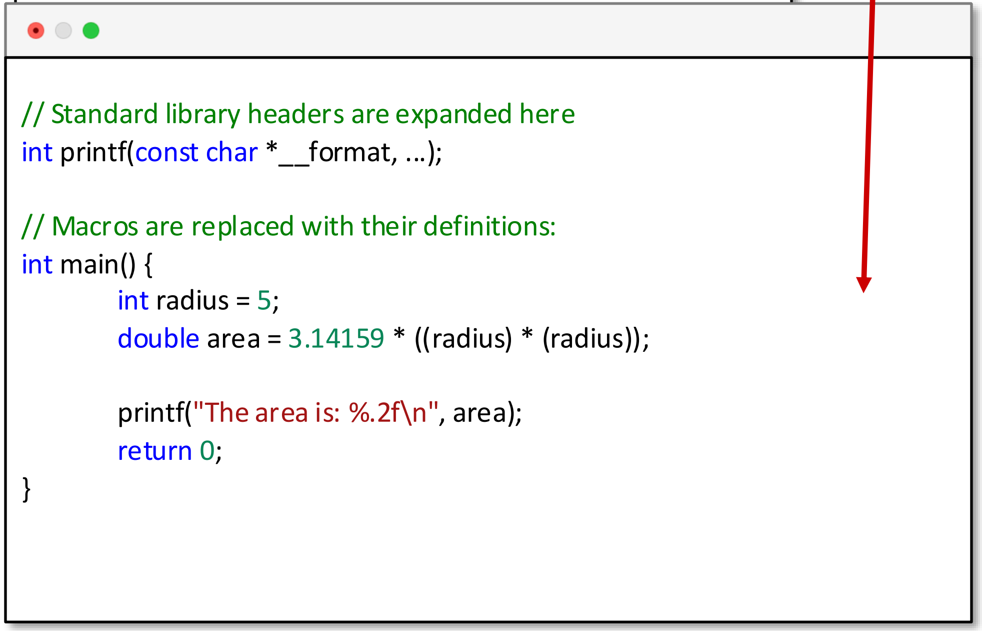
But this is a new type of macro for us:

```
#define SQUARE(x) ((x) * (x))
```

Function-like Macros



```
#include <stdio.h>
#define PI 3.14159
#define SQUARE(x) ((x) * (x))

int main() {
            int radius = 5;
            double area = PI * SQUARE(radius);

            printf("The area is: %.2f\n", area);
            return 0;
}
```



```
// Standard library headers are expanded here
int printf(const char *__format, ...);

// Macros are replaced with their definitions:
int main() {
            int radius = 5;
            double area = 3.14159 * ((radius) * (radius));

            printf("The area is: %.2f\n", area);
            return 0;
}
```

SQUARE(radius) is replaced with ***((***radius***) * (***radius***)).***

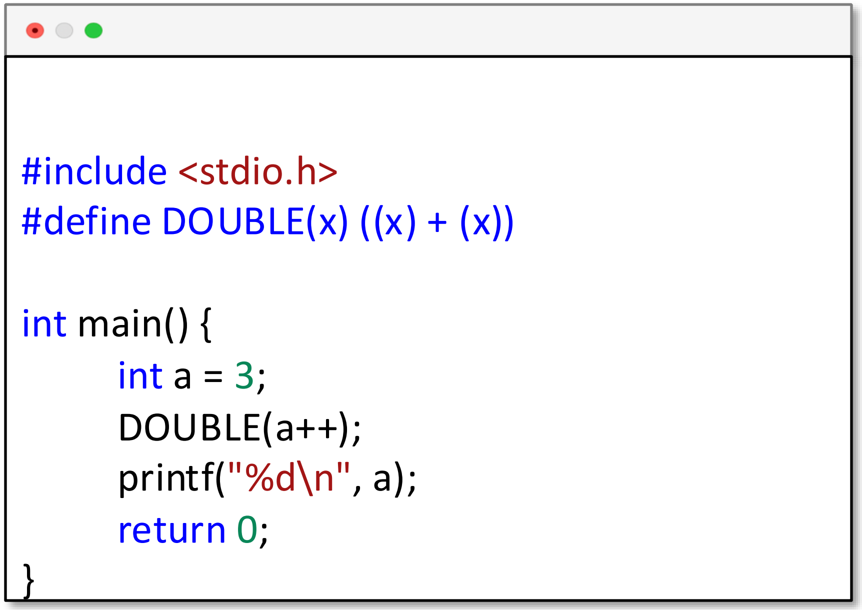
The macro expansion is text-based, meaning it’s a direct substitution rather than an actual function.

This might lead to head aches when writing them...

Poll Everywhere

pollev.com/cis2400

- What is the value of a that will be printed after the Function-Like Macro is executed?



```
#include <stdio.h>
#define DOUBLE(x) ((x) + (x))

int main() {
        int a = 3;
        DOUBLE(a++);
        printf("%d\n", a);
        return 0;
}
```

A) 3

B) 4

C) 5

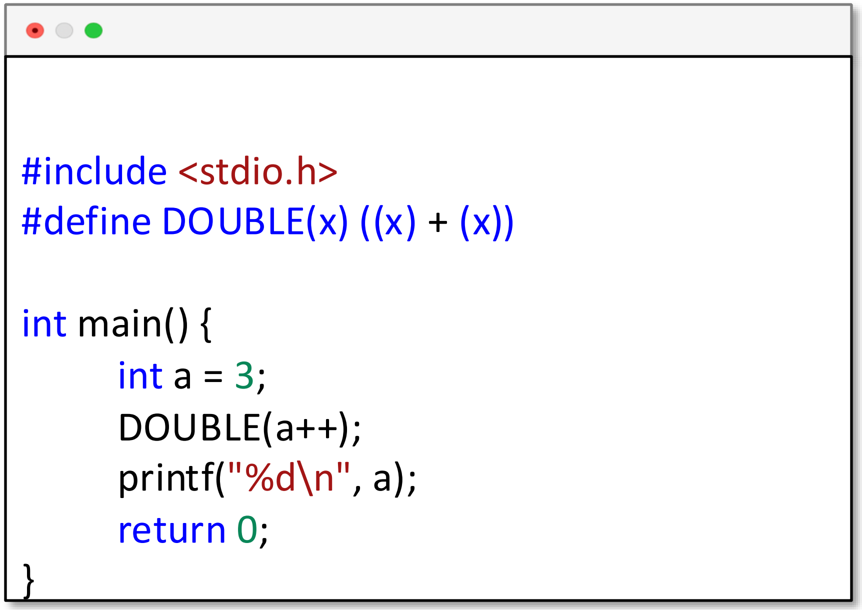
D) 6

E) Not sure.

Poll Everywhere

pollev.com/cis2400

- What is the value of a that will be printed after the Function-Like Macro is executed?



```
#include <stdio.h>
#define DOUBLE(x) ((x) + (x))

int main() {
	int a = 3;
	DOUBLE(a++);
	printf("%d\n", a);
	return 0;
}
```

A) 3

B) 4

C) 5

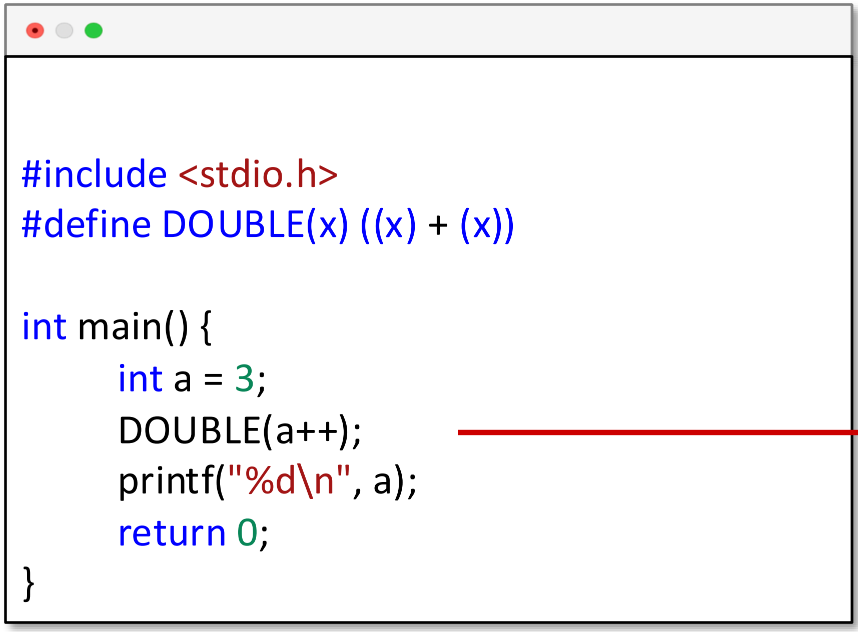
D) 6

E) Not sure.

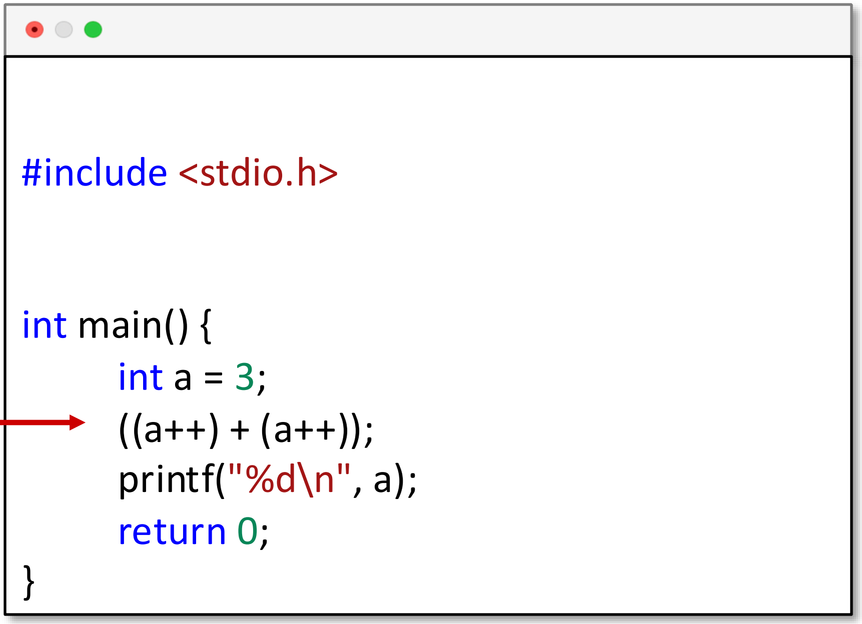
Poll Everywhere

pollev.com/cis2400

- What is the value of a that will be printed after the Function-Like Macro is executed?



```
#include <stdio.h>
#define DOUBLE(x) ((x) + (x))

int main() {
        int a = 3;
        DOUBLE(a++);
        printf("%d\n", a);
        return 0;
}
```



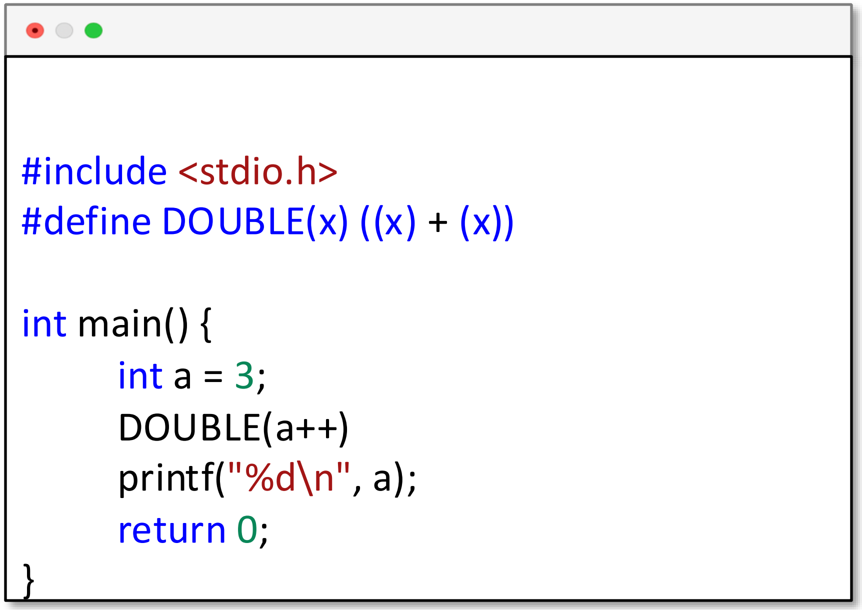
```
#include <stdio.h>


int main() {
        int a = 3;
        ((a++) + (a++));
        printf("%d\n", a);
        return 0;
}
```

Since a++ is evaluated twice, a is incremented twice in total!

Poll Everywhere

pollev.com/cis2400

- What is the value of a that will be printed after the Function-Like Macro is executed?



```
#include <stdio.h>
#define DOUBLE(x) ((x) + (x))

int main() {
        int a = 3;
        DOUBLE(a++)
        printf("%d\n", a);
        return 0;
}
```

A) 3

B) 4

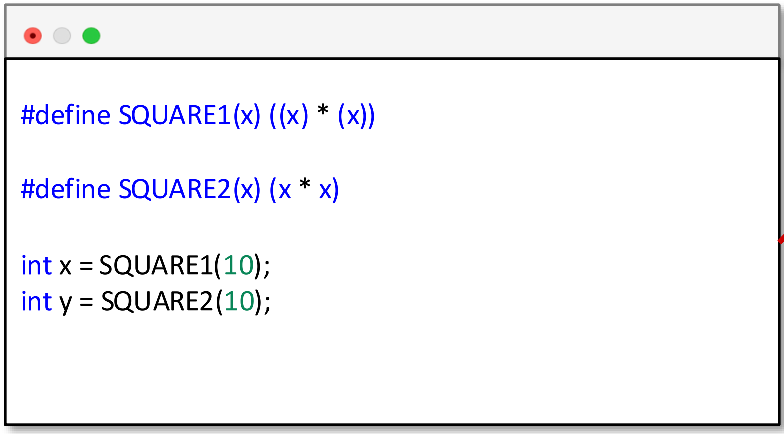
[✓ C) 5]

D) 6

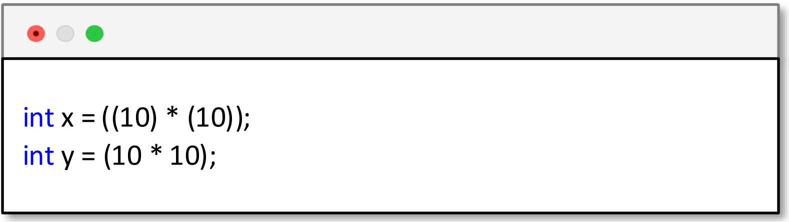
E) Not sure.

Macros and Multiple Evaluation

- Expansion of Macros can result in unexpected side affects.

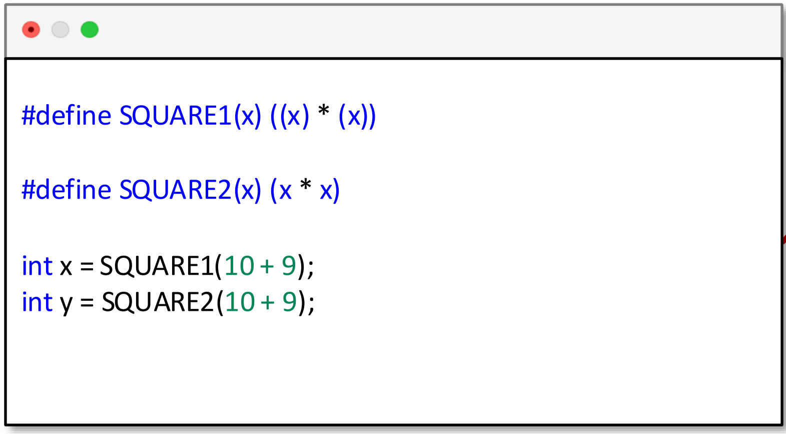


```
#define SQUARE1(x) ((x) * (x))

#define SQUARE2(x) (x * x)

int x = SQUARE1(10);
int y = SQUARE2(10);
```

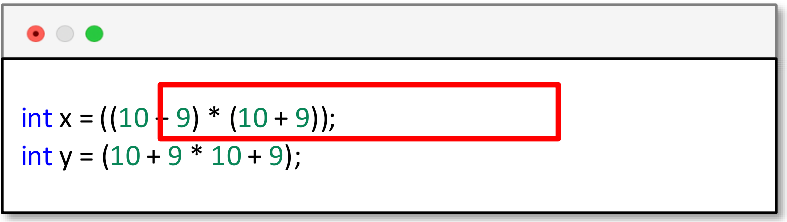


```
int x = ((10) * (10));
int y = (10 * 10);
```

These two ‘SQUARE’ Macros seem to do the same thing but only one it correct!



```
#define SQUARE1(x) ((x) * (x))

#define SQUARE2(x) (x * x)

int x = SQUARE1(10 + 9);
int y = SQUARE2(10 + 9);
```

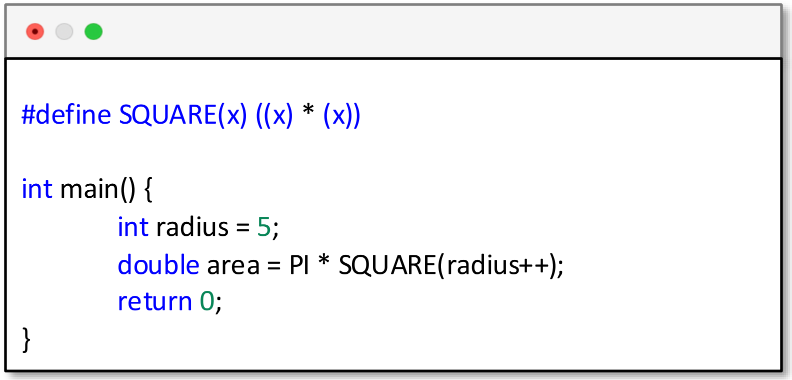


```
int x = ((10 + 9) * (10 + 9));
int y = (10 + 9 * 10 + 9);
```

Macros and Multiple Evaluation

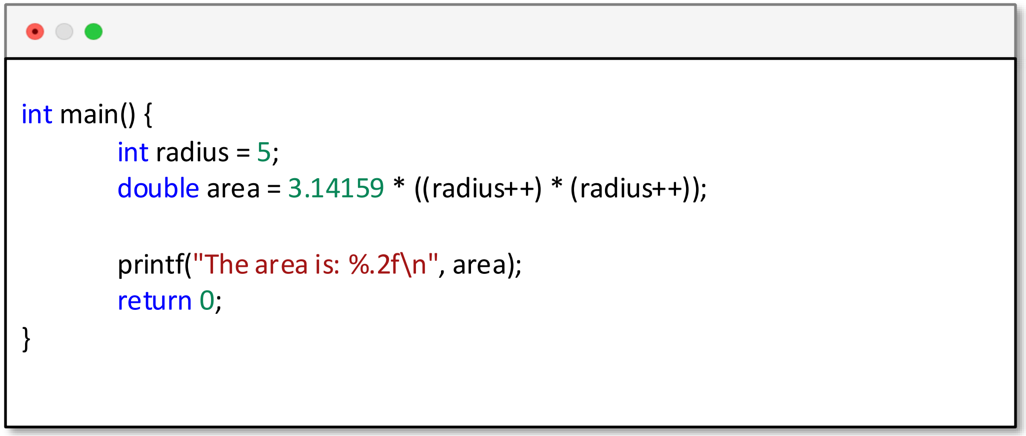
- Expansion of Macros can result in unexpected side affects.

file.c



```
#define SQUARE(x) ((x) * (x))

int main() {
        int radius = 5;
        double area = PI * SQUARE(radius++);
        return 0;
}
```

clang-15 -E file.c -o file.i



```
int main() {
        int radius = 5;
        double area = 3.14159 * ((radius++) * (radius++));

        printf("The area is: %.2f\n", area);
        return 0;
}
```

This is why it's essential to treat functions in C and macros as different things, especially when they contain expressions with parameters that could have side effects if evaluated multiple times.

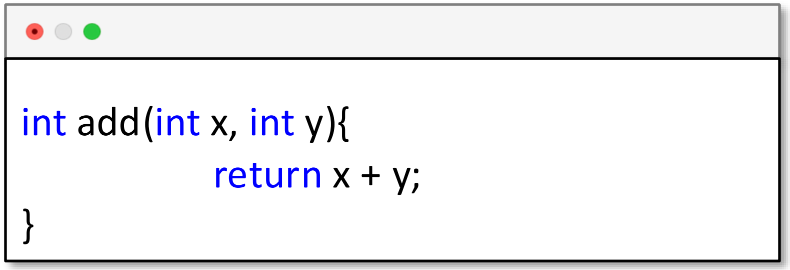
Lecture Outline

- C Refresher
  - Pointers, Memory Allocation
  - Macros:
    - Constants vs. Function-like Macros
    - Evaluation Pitfalls
- C to RISC-V
  - Purpose of Translating C to Assembly
  - Performance, Memory Control
- Function Calls in RISC-V
  - The Registers
  - Argument Passing
  - Return Values
  - The Frame and Returning from Functions

C to RISC-V

- **Why learn code translation if compilers do it for us?**
  - Systems programming demands ***precise control*** over memory and speed. We need to be confident that our code does what we expect it to.
  - Knowing how compilers make translation decisions is ***essential.***
    - ***Allows you to use the compiler to your advantage.***
  - Familiarity with conventions for saving state, passing, and returning values across functions is critical.

C to RISC-V



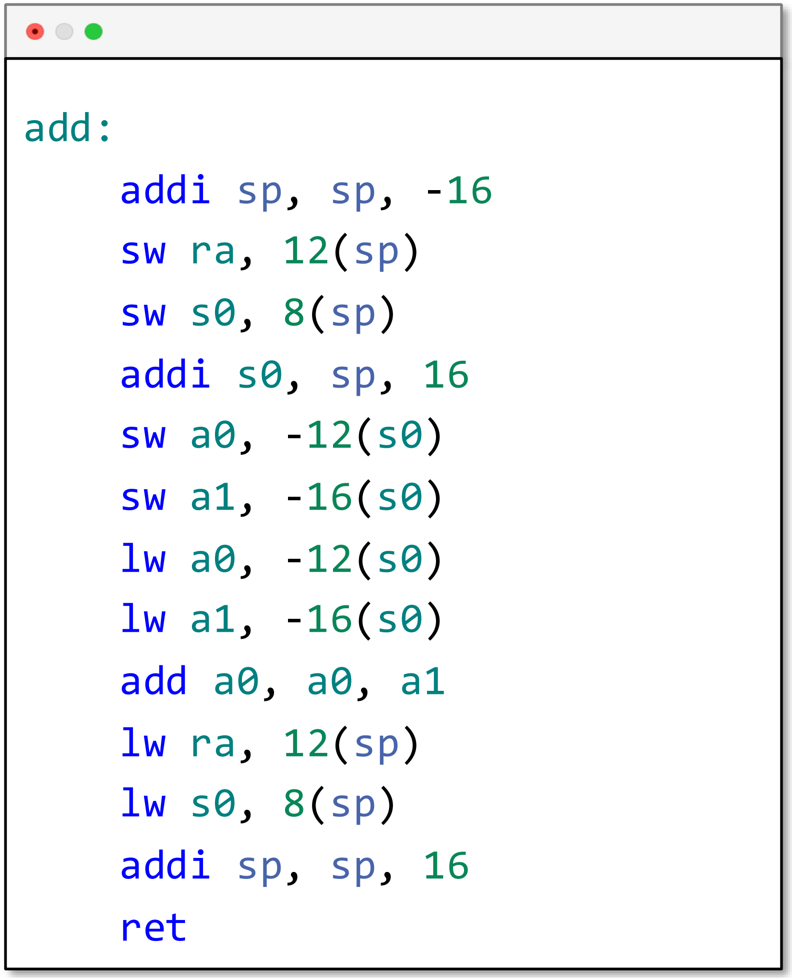
```c
int add(int x, int y){
        return x + y;
}
```

As an example, lets' see why one line of C become 13 lines of RISC-V.

Quick Refresher:

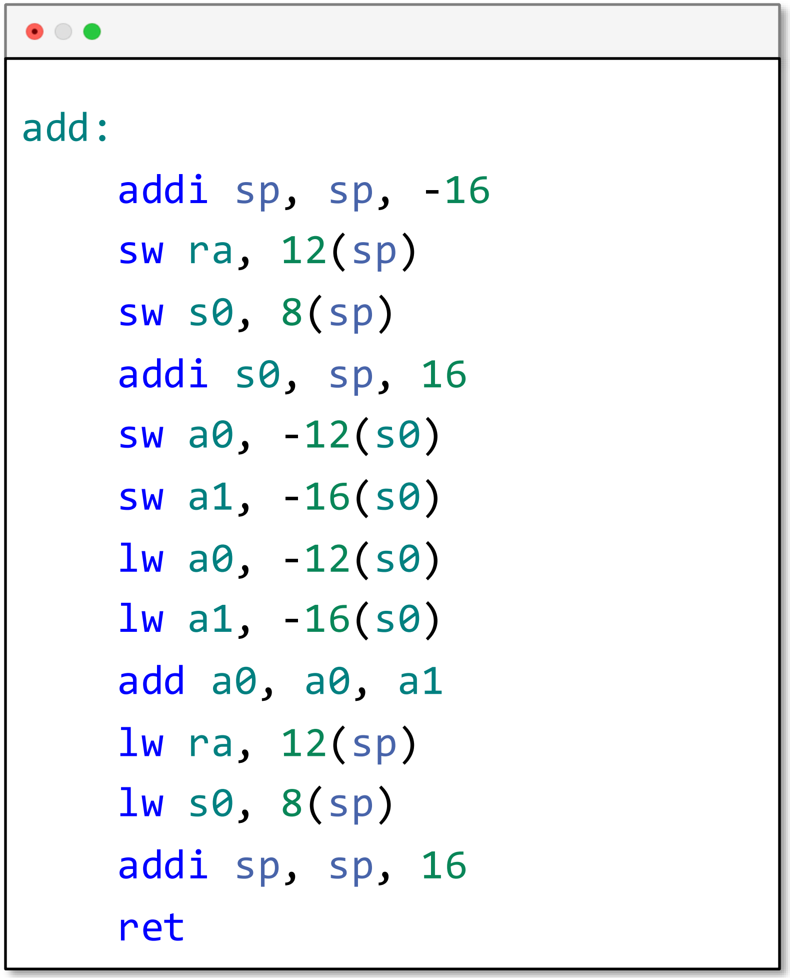
1. Caller
   - The function who called the function

2. Callee
   - The function called.



```
add:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    addi s0, sp, 16
    sw a0, -12(s0)
    sw a1, -16(s0)
    lw a0, -12(s0)
    lw a1, -16(s0)
    add a0, a0, a1
    lw ra, 12(sp)
    lw s0, 8(sp)
    addi sp, sp, 16
    ret
```

C to RISC-V



```
add:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    addi s0, sp, 16
    sw a0, -12(s0)
    sw a1, -16(s0)
    lw a0, -12(s0)
    lw a1, -16(s0)
    add a0, a0, a1
    lw ra, 12(sp)
    lw s0, 8(sp)
    addi sp, sp, 16
    ret
```

Questions to consider…

Why is `sp` used at the start and at the end?

Why do we store `ra` and `s0` and then load them again at the end?

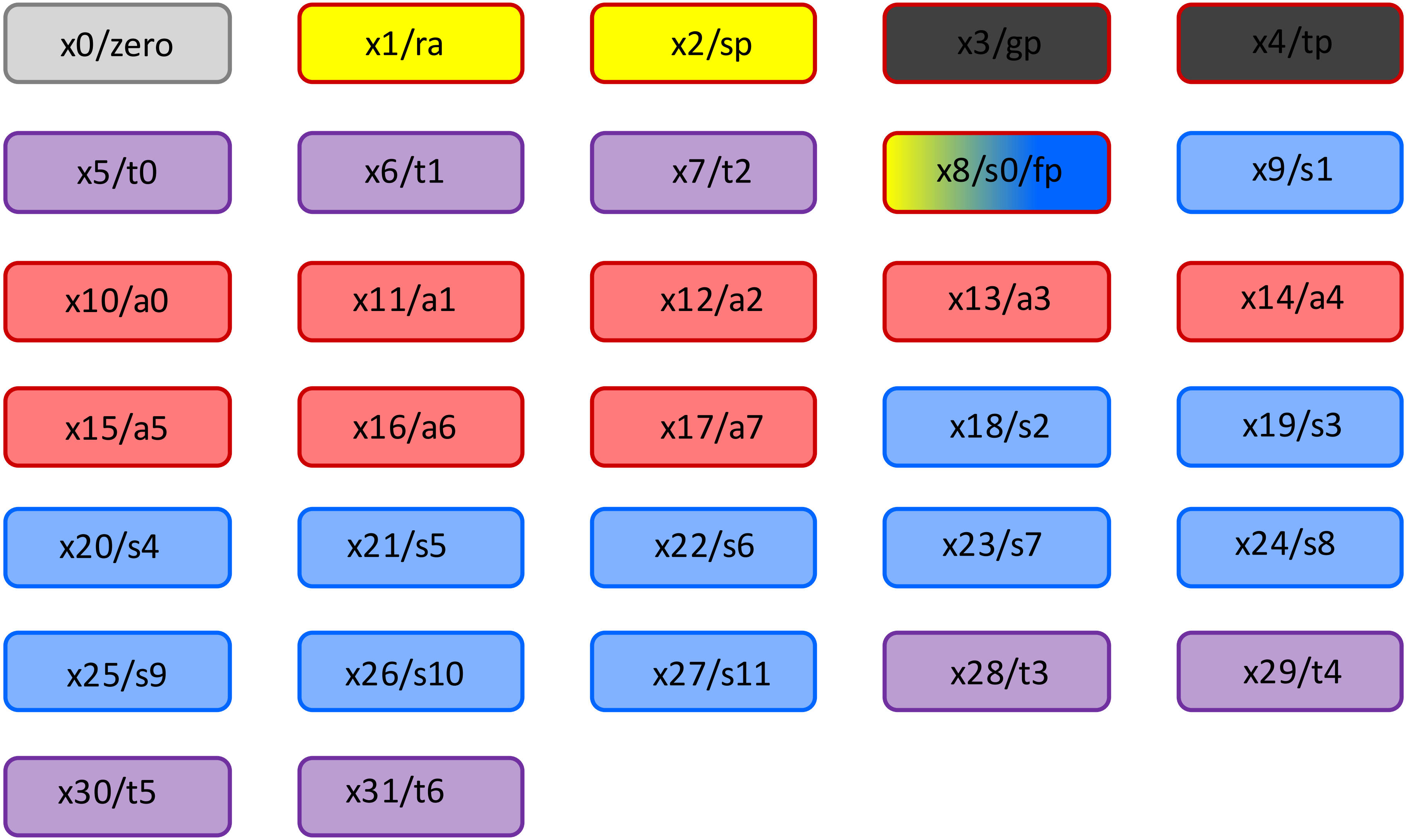
What really is `ra` and `s0`?

What really is `a0` and `a1`?

How are values returned from functions?

Registers

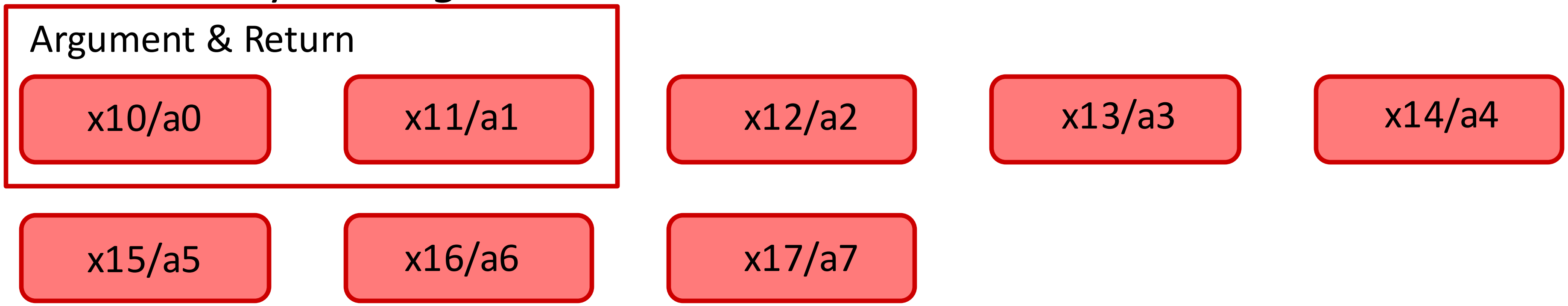
Yes, we don’t care about these two.



x0/zero	x1/ra	x2/sp	x3/gp	x4/tp
x5/t0	x6/t1	x7/t2	x8/s0/fp	x9/s1
x10/a0	x11/a1	x12/a2	x13/a3	x14/a4
x15/a5	x16/a6	x17/a7	x18/s2	x19/s3
x20/s4	x21/s5	x22/s6	x23/s7	x24/s8
x25/s9	x26/s10	x27/s11	x28/t3	x29/t4
x30/t5	x31/t6			

Registers

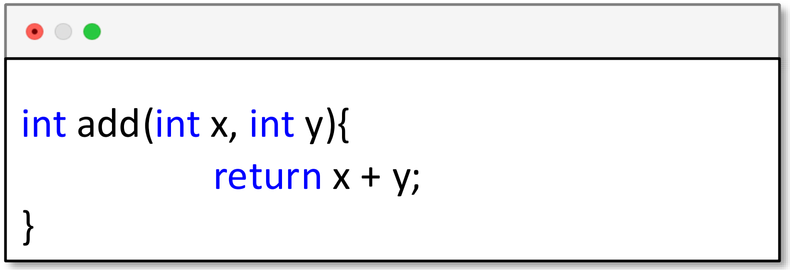
- Argument Registers
  - These *pass* arguments to functions
  - Only ***two registers*** *'return'* values.



Argument & Return

x10/a0 x11/a1 x12/a2 x13/a3 x14/a4

x15/a5 x16/a6 x17/a7

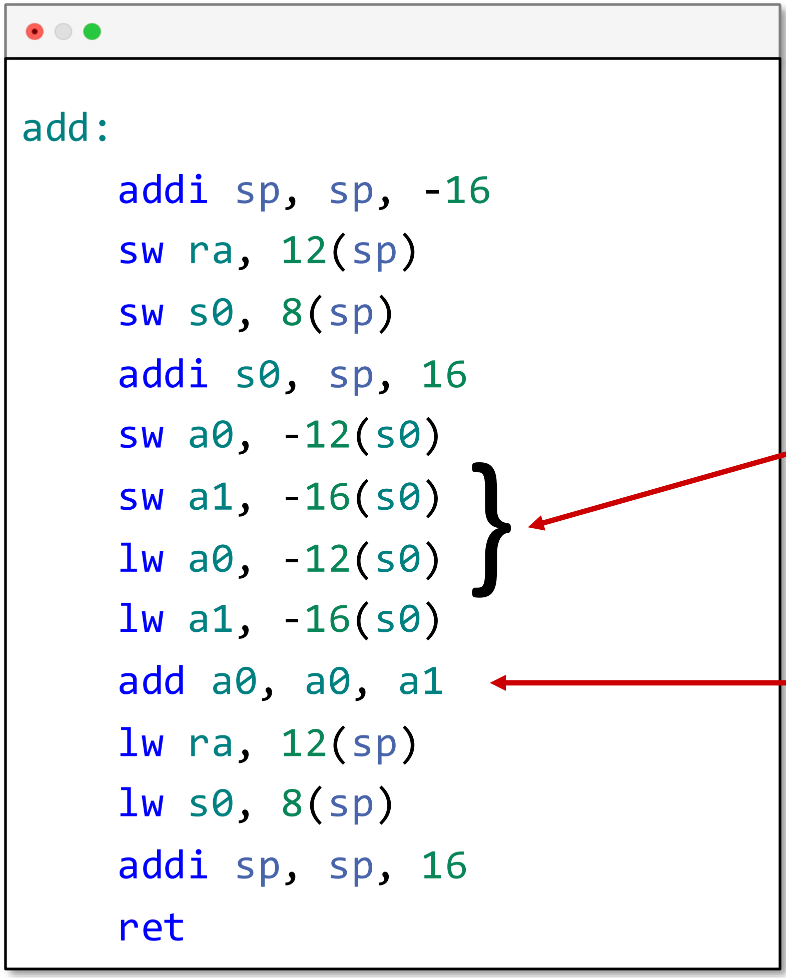


```
int add(int x, int y){
        return x + y;
}
```

When we enter the add routine, we expect x in a0 and y in a1.

When we return from the add routine, we expect the result in a0.

C to RISC-V



```
add:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    addi s0, sp, 16
    sw a0, -12(s0)
    sw a1, -16(s0)
    lw a0, -12(s0)
    lw a1, -16(s0)
    add a0, a0, a1
    lw ra, 12(sp)
    lw s0, 8(sp)
    addi sp, sp, 16
    ret
```

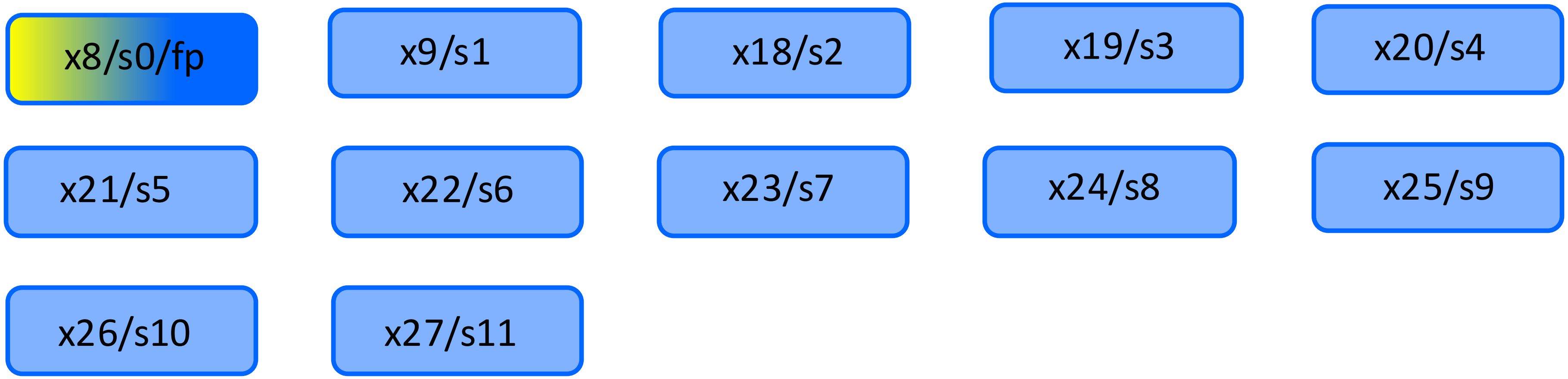
What really is `a0` and `a1`?

Now we know, here we are storing and re-'loading' the arguments passed into the routine, add.

Here, we store the result in `a0` because that's where the *caller* expects it.

Registers

- S Registers (Callee-Saved)
  - These registers ***must*** be restored to their original values when a function returns.

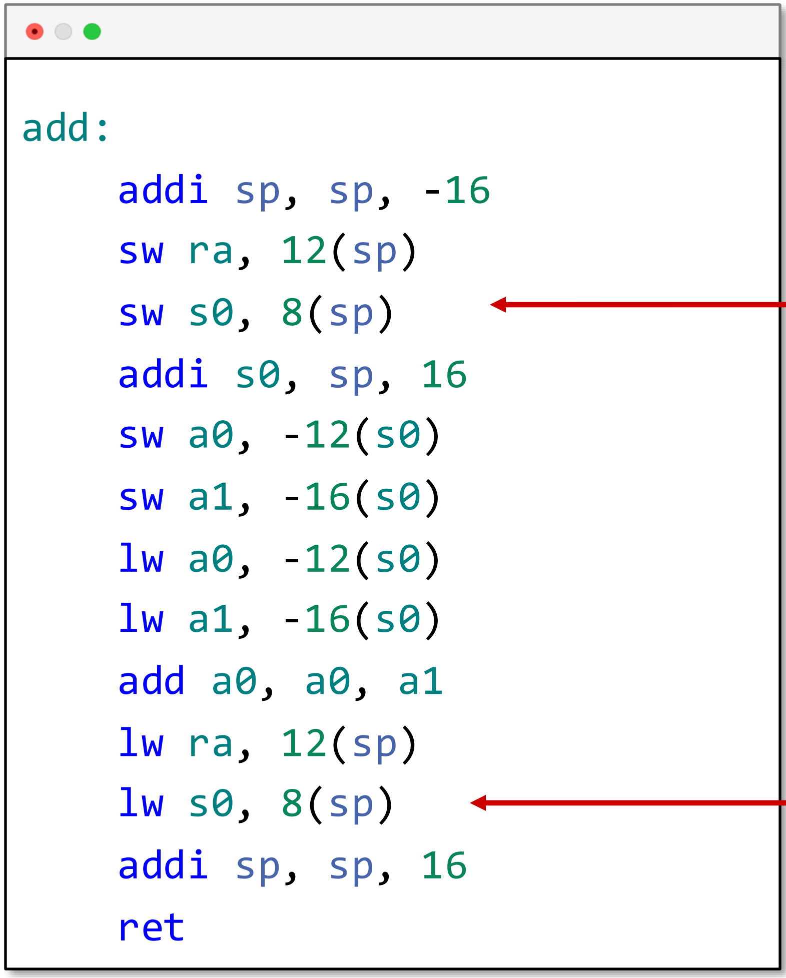


x8/s0/fp x9/s1 x18/s2 x19/s3 x20/s4

x21/s5 x22/s6 x23/s7 x24/s8 x25/s9

x26/s10 x27/s11

C to RISC-V



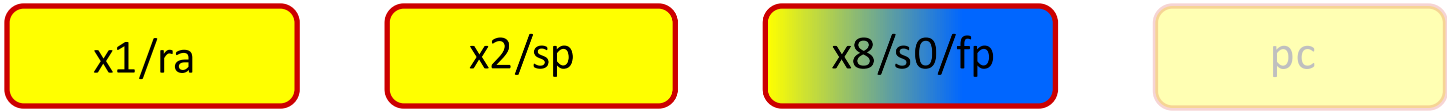
```
add:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    addi s0, sp, 16
    sw a0, -12(s0)
    sw a1, -16(s0)
    lw a0, -12(s0)
    lw a1, -16(s0)
    add a0, a0, a1
    lw ra, 12(sp)
    lw s0, 8(sp)
    addi sp, sp, 16
    ret
```

`s0` is saved @ `8(sp)`

If this function were to modify any others `s` registers, it would have to do the same thing we see here.

`s0` is loaded from @ `8(sp)`

Registers



x1/ra

x2/sp

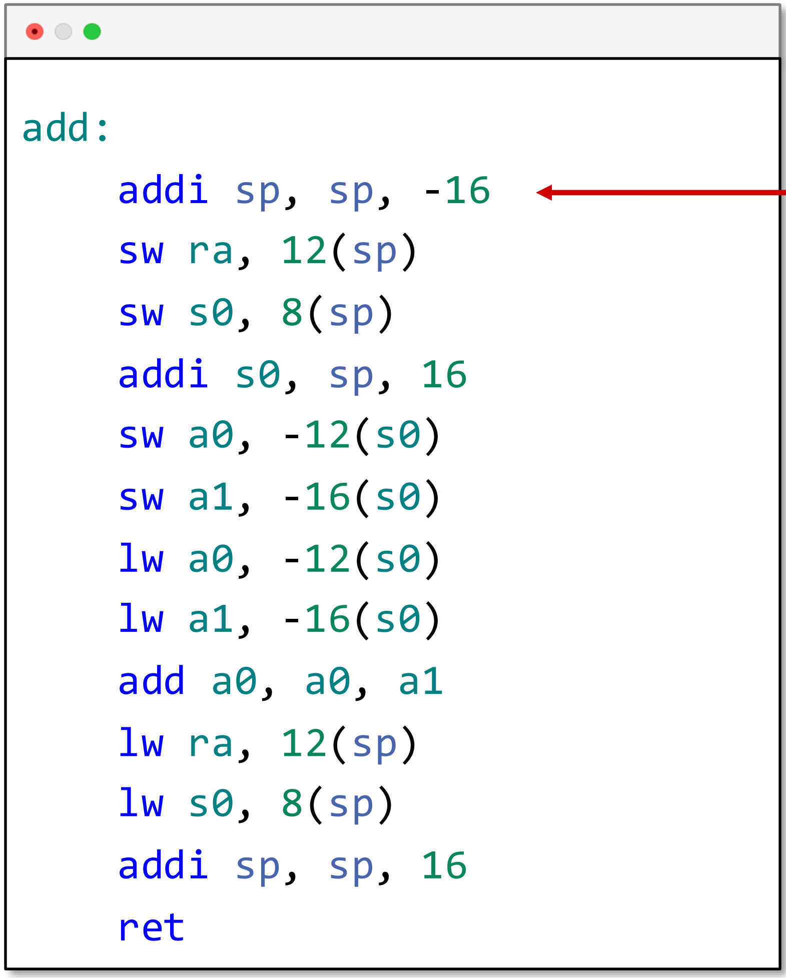
x8/s0/fp

pc

It’s transparent because we can’t interact with it ‘directly’.

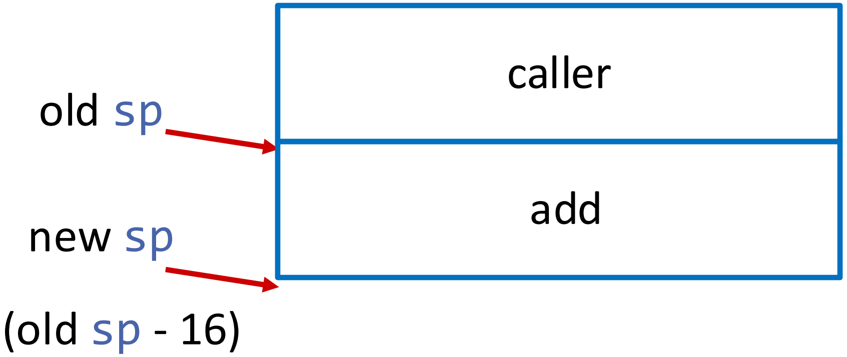
- Return Address (ra)
  - Tells us what we should set the PC to when we are done with a routine.
  - It is what makes “returning” from a function possible.
- Stack Pointer (sp)
  - Points to the bottom of the stack.
  - Allows us to do memory accesses (loads/stores) *relative* to the ‘bottom’ of the stack.
- Frame Pointer (s0, fp)
  - Points to the ***beginning*** of a function's “stack” or “Frame”.

C to RISC-V

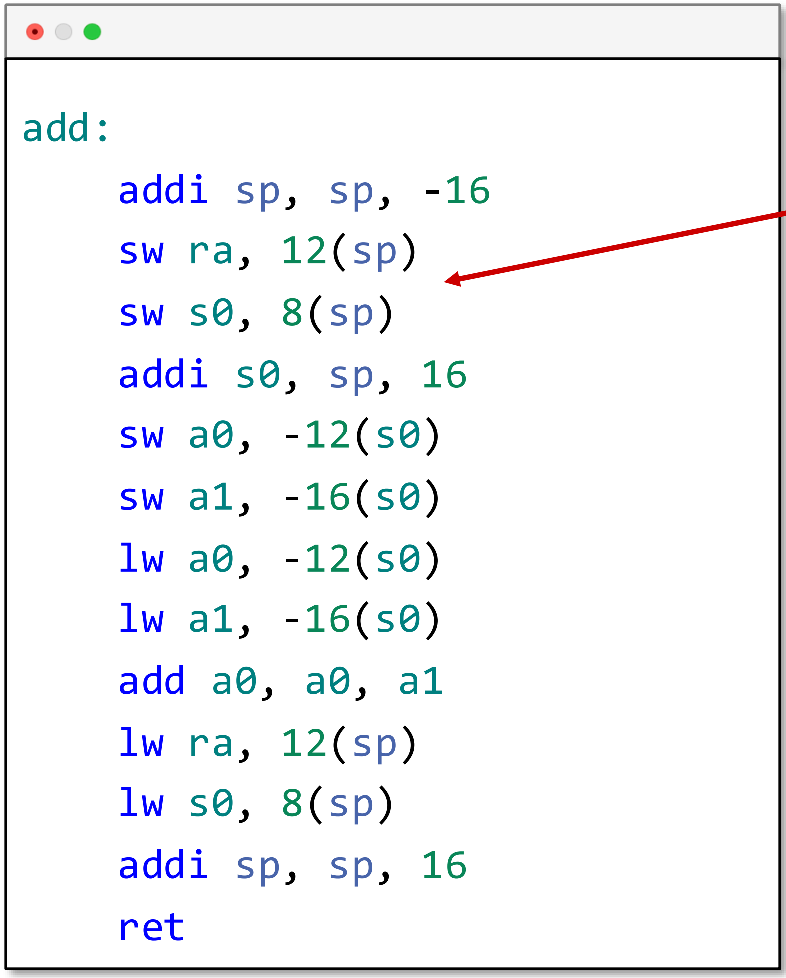


```
add:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    addi s0, sp, 16
    sw a0, -12(s0)
    sw a1, -16(s0)
    lw a0, -12(s0)
    lw a1, -16(s0)
    add a0, a0, a1
    lw ra, 12(sp)
    lw s0, 8(sp)
    addi sp, sp, 16
    ret
```

The first thing this procedure does is make space for it's stack. The stack 'grows down' so we subtract from the `sp`.

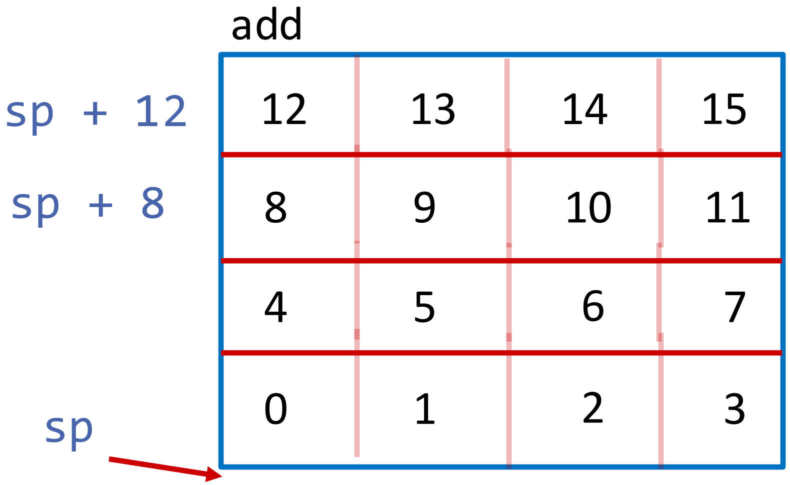


C to RISC-V



```
add:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    addi s0, sp, 16
    sw a0, -12(s0)
    sw a1, -16(s0)
    lw a0, -12(s0)
    lw a1, -16(s0)
    add a0, a0, a1
    lw ra, 12(sp)
    lw s0, 8(sp)
    addi sp, sp, 16
    ret
```

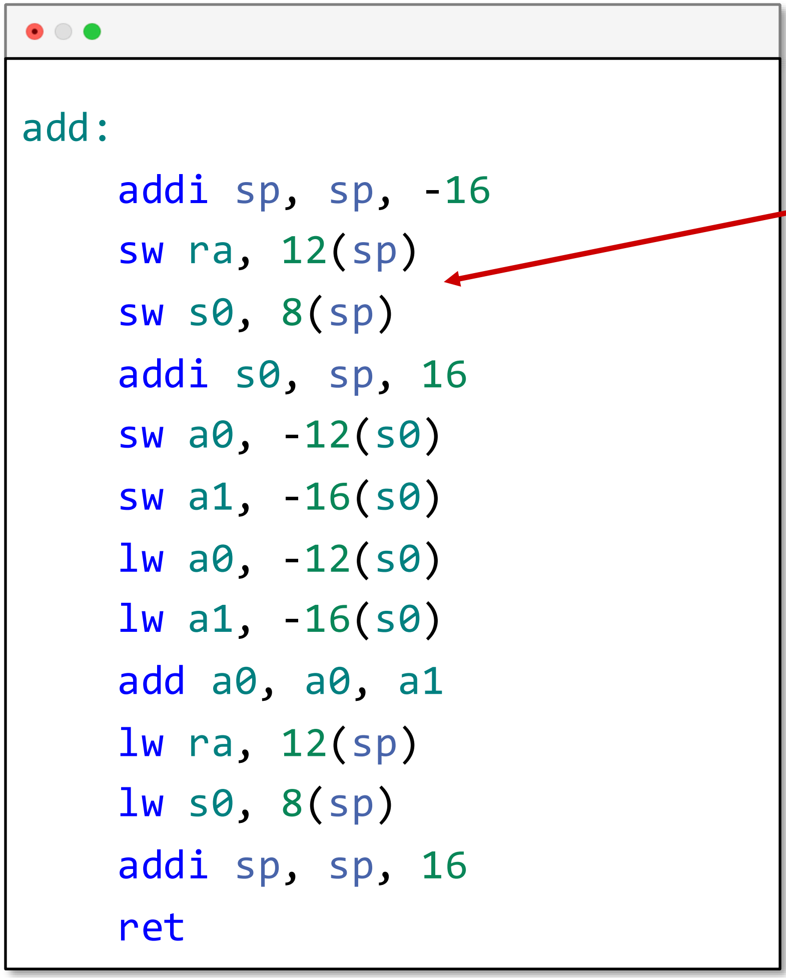
Now, we store these registers `ra & s0` relative to the new new `sp`!



12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

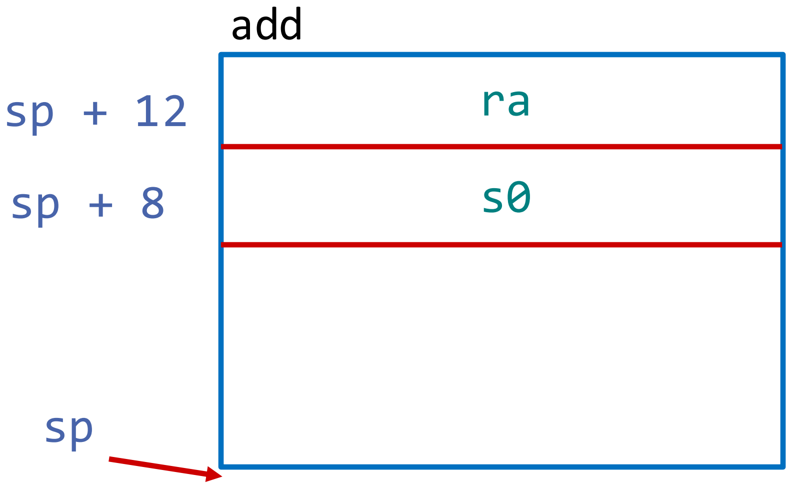
This shows you what byte is "relative" to sp.

C to RISC-V

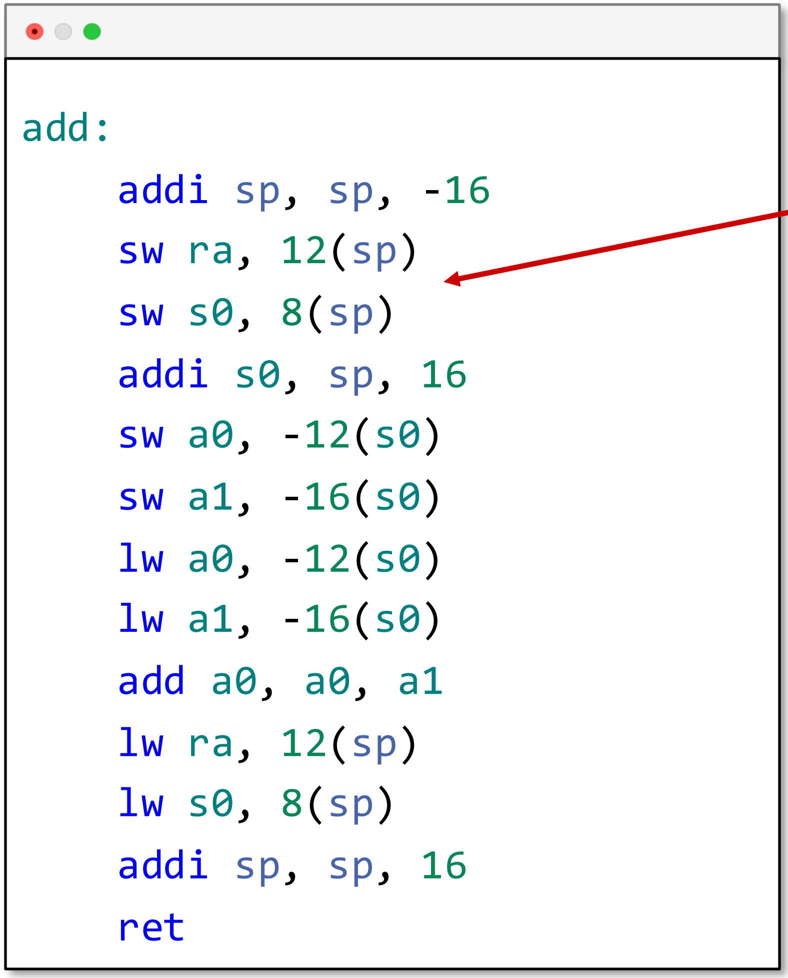


```
add:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    addi s0, sp, 16
    sw a0, -12(s0)
    sw a1, -16(s0)
    lw a0, -12(s0)
    lw a1, -16(s0)
    add a0, a0, a1
    lw ra, 12(sp)
    lw s0, 8(sp)
    addi sp, sp, 16
    ret
```

Now, we store these registers ra & s0 relative to the new new sp!

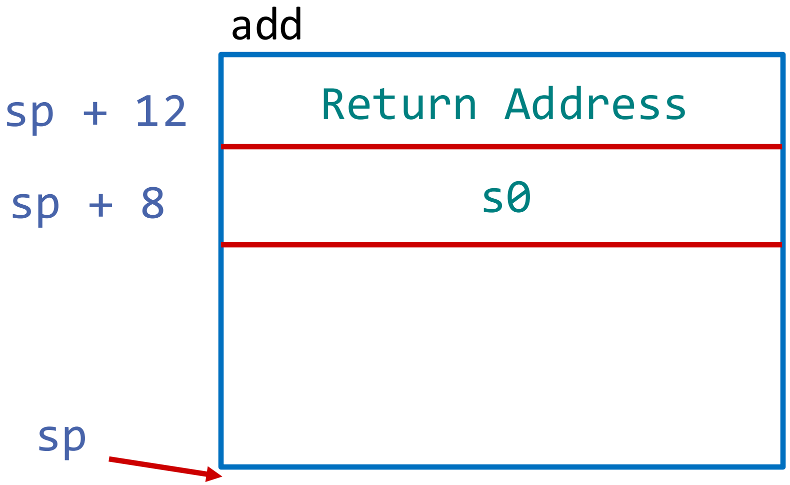


C to RISC-V



```
add:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    addi s0, sp, 16
    sw a0, -12(s0)
    sw a1, -16(s0)
    lw a0, -12(s0)
    lw a1, -16(s0)
    add a0, a0, a1
    lw ra, 12(sp)
    lw s0, 8(sp)
    addi sp, sp, 16
    ret
```

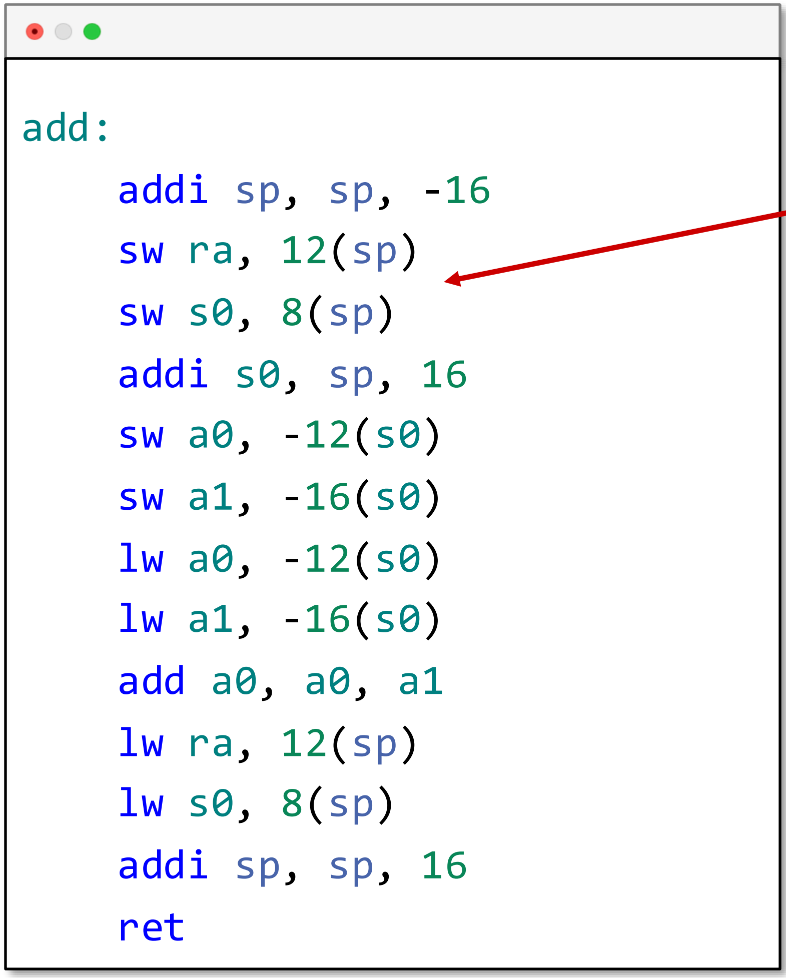
Now, we store these registers `ra & s0` relative to the new new `sp`!



If `s0` is a frame pointer and we save it - who's frame pointer are we saving?

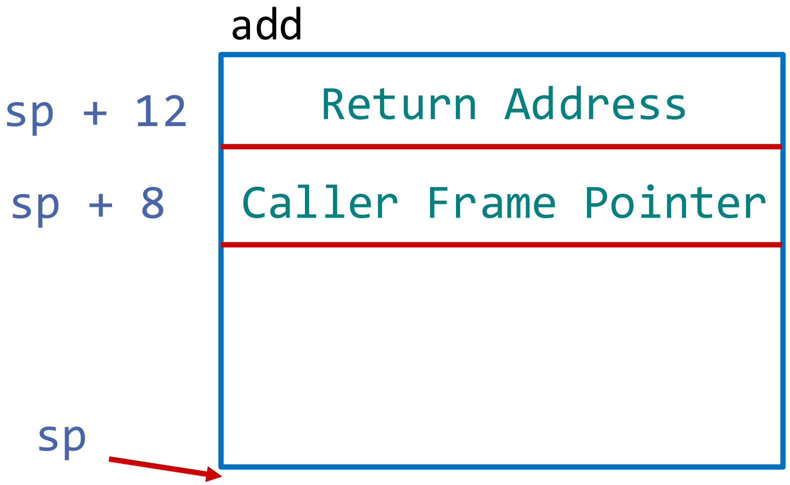
The callers frame pointer!

C to RISC-V



```
add:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    addi s0, sp, 16
    sw a0, -12(s0)
    sw a1, -16(s0)
    lw a0, -12(s0)
    lw a1, -16(s0)
    add a0, a0, a1
    lw ra, 12(sp)
    lw s0, 8(sp)
    addi sp, sp, 16
    ret
```

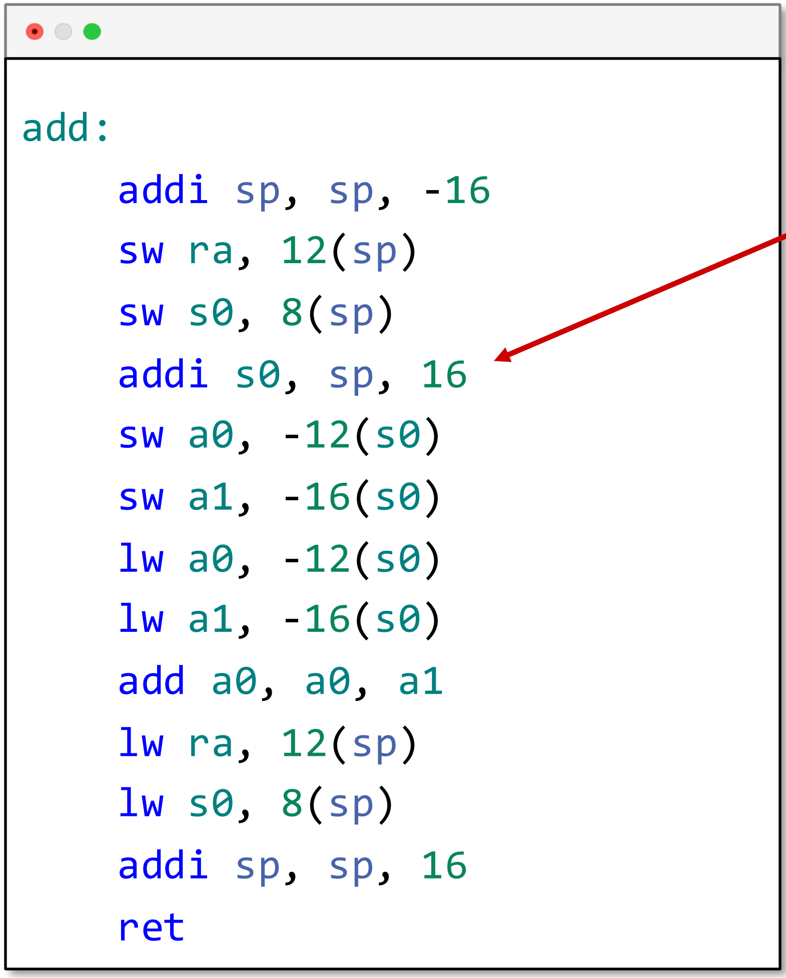
Now, we store these registers ra & s0 relative to the new new sp!



If s0 is a frame pointer and we save it - who's frame pointer are we saving?

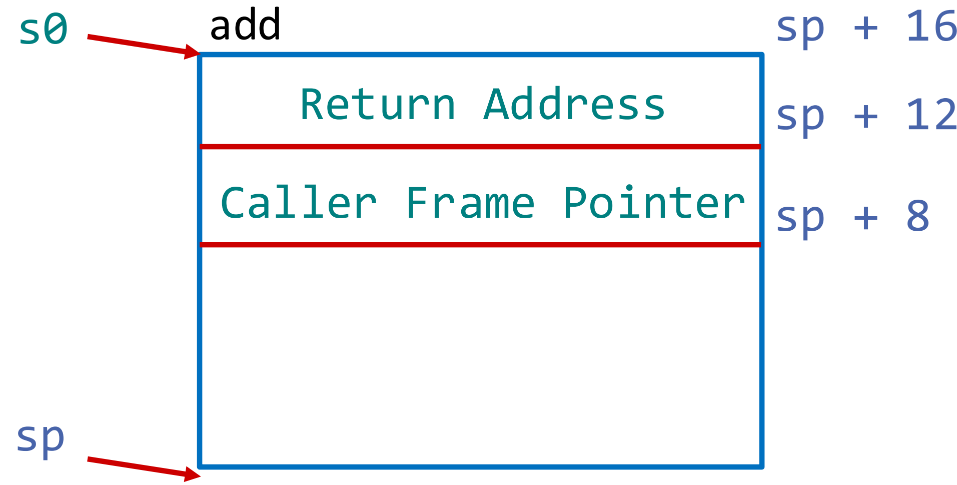
The callers frame pointer!

C to RISC-V



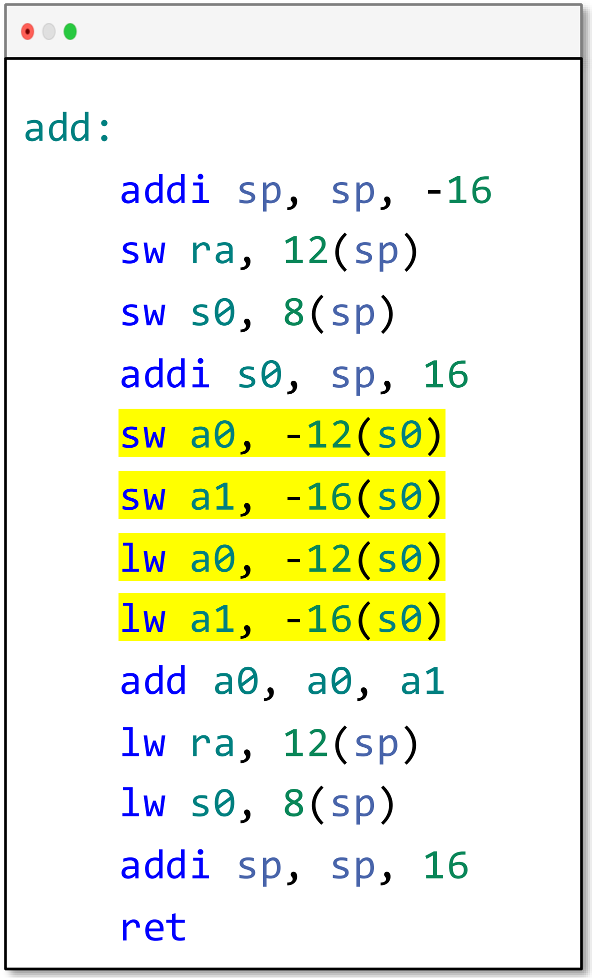
```
add:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    addi s0, sp, 16
    sw a0, -12(s0)
    sw a1, -16(s0)
    lw a0, -12(s0)
    lw a1, -16(s0)
    add a0, a0, a1
    lw ra, 12(sp)
    lw s0, 8(sp)
    addi sp, sp, 16
    ret
```

sp + 16 is the the start of our frame!

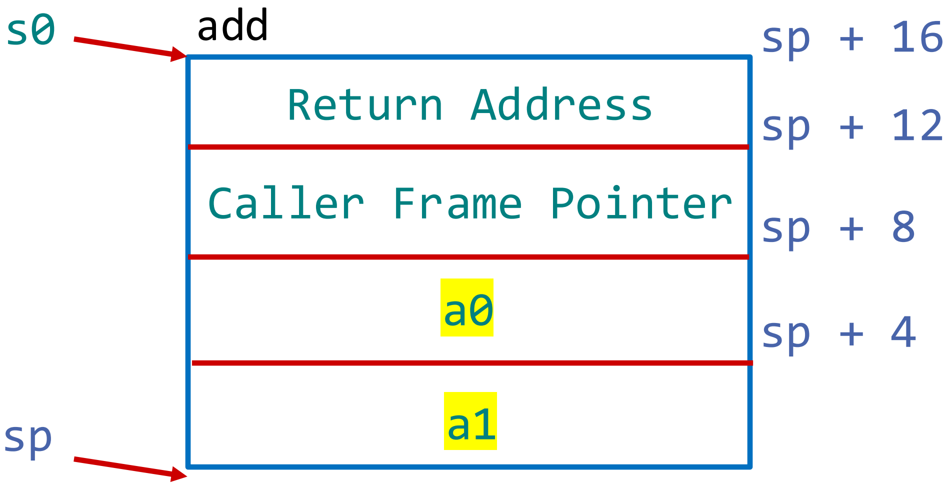


s0 is updated to be add’s frame pointer

C to RISC-V



```
add:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    addi s0, sp, 16
    sw a0, -12(s0)
    sw a1, -16(s0)
    lw a0, -12(s0)
    lw a1, -16(s0)
    add a0, a0, a1
    lw ra, 12(sp)
    lw s0, 8(sp)
    addi sp, sp, 16
    ret
```

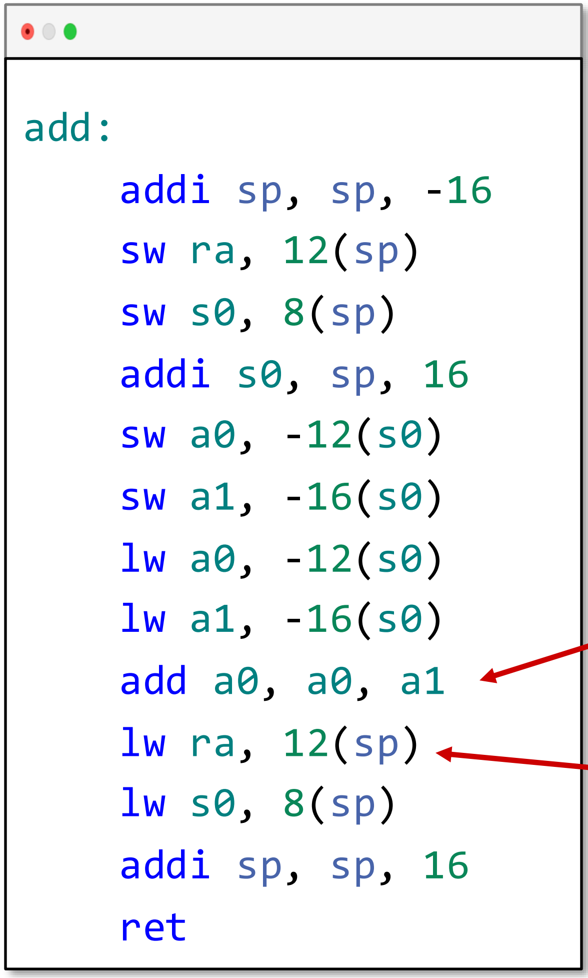


Although, we are now using memory accesses relative to `s0`, let's rewrite them using `sp` for simplicity.

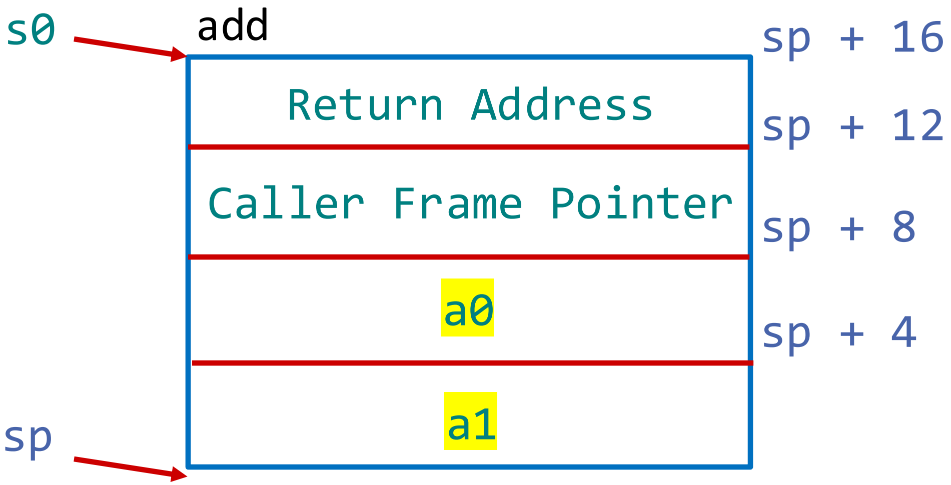
```
sw a0, -12(s0)        sw a0, 4(sp)
sw a1, -16(s0)   →    sw a1, 0(sp)
```

Using this, we can see where `a0` and `a1` are stored more easily.

C to RISC-V



```
add:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    addi s0, sp, 16
    sw a0, -12(s0)
    sw a1, -16(s0)
    lw a0, -12(s0)
    lw a1, -16(s0)
    add a0, a0, a1
    lw ra, 12(sp)
    lw s0, 8(sp)
    addi sp, sp, 16
    ret
```



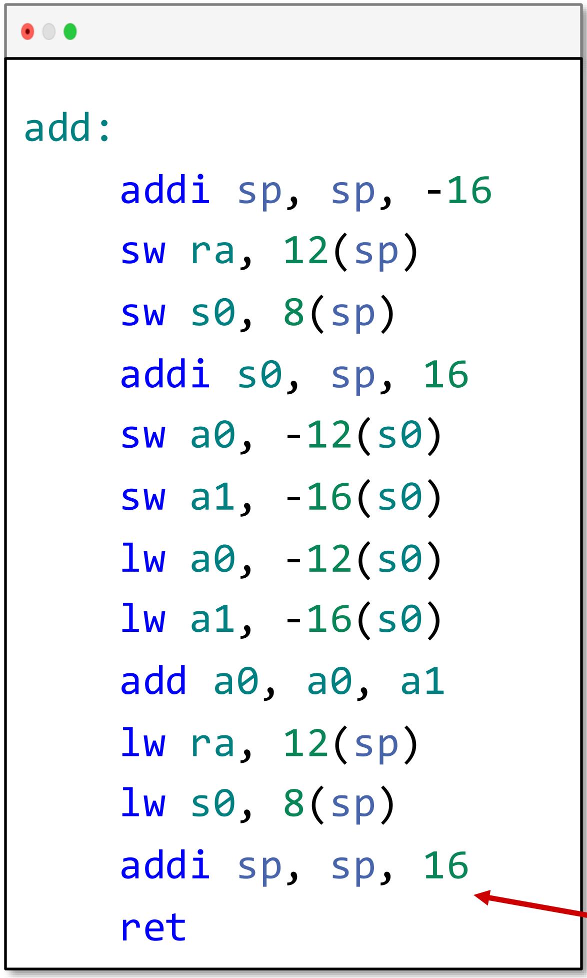
Now, a0 holds the result.

We restore the ra.

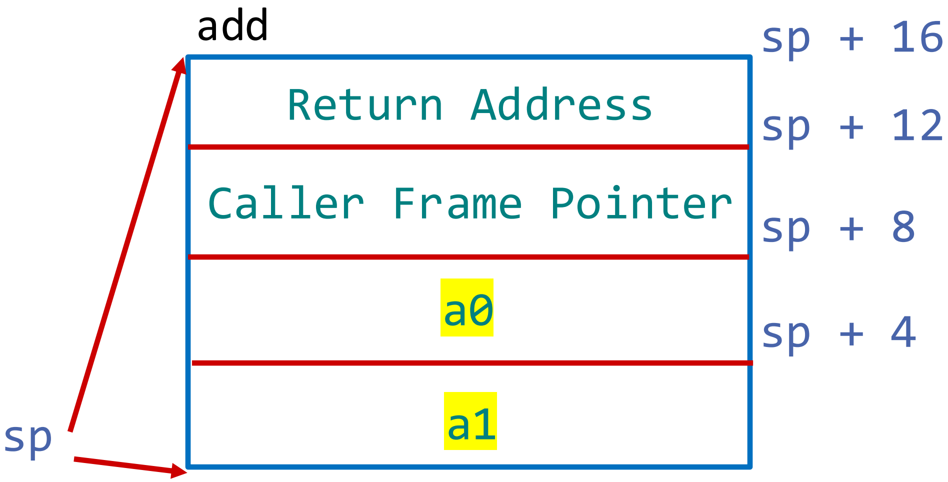
We restore the s0.

Quick note:
Why would we ever need to 'restore the ra'?

C to RISC-V

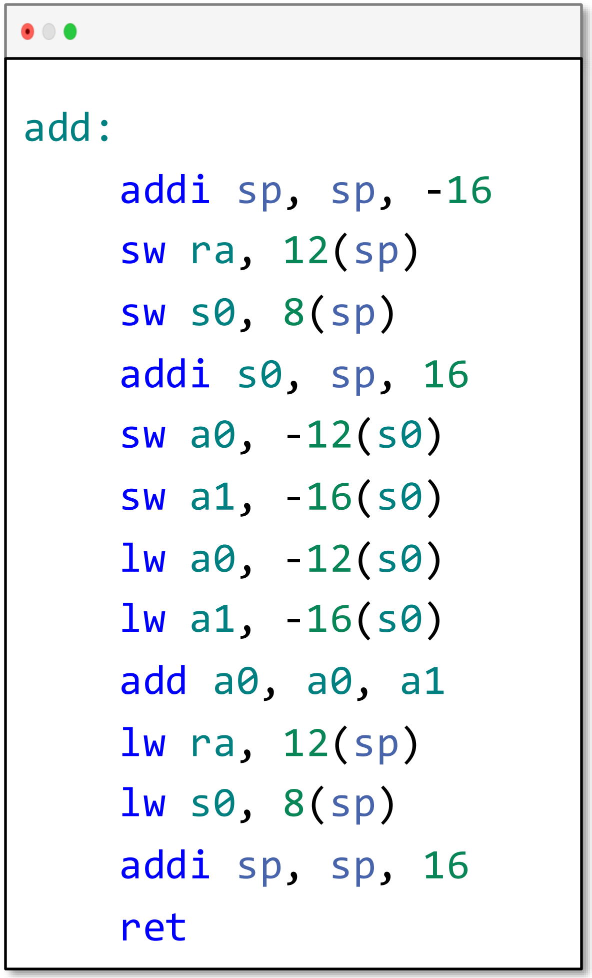


```
add:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    addi s0, sp, 16
    sw a0, -12(s0)
    sw a1, -16(s0)
    lw a0, -12(s0)
    lw a1, -16(s0)
    add a0, a0, a1
    lw ra, 12(sp)
    lw s0, 8(sp)
    addi sp, sp, 16
    ret
```

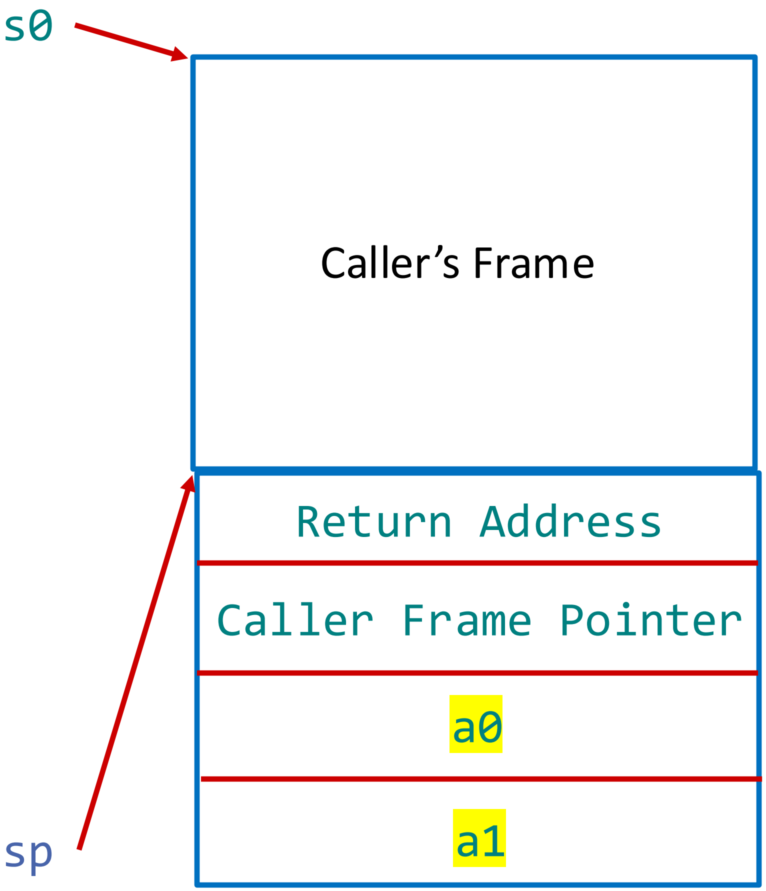


We restore the old sp.

C to RISC-V

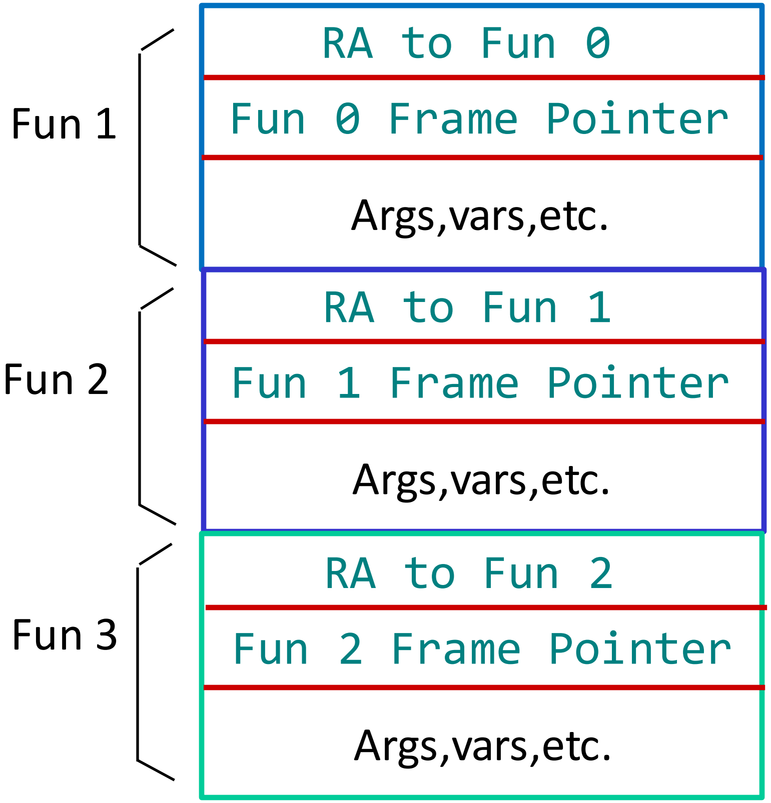


```
add:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    addi s0, sp, 16
    sw a0, -12(s0)
    sw a1, -16(s0)
    lw a0, -12(s0)
    lw a1, -16(s0)
    add a0, a0, a1
    lw ra, 12(sp)
    lw s0, 8(sp)
    addi sp, sp, 16
    ret
```



And now, we're done and return.

C to RISC-V



Fun 1

RA to Fun 0

Fun 0 Frame Pointer

Args,vars,etc.

Fun 2

RA to Fun 1

Fun 1 Frame Pointer

Args,vars,etc.

Fun 3

RA to Fun 2

Fun 2 Frame Pointer

Args,vars,etc.

Function Frames:

- store the caller’s Frame Pointer
- store the return address

How does the callee know what the correct return address is?

jal rd,targ20

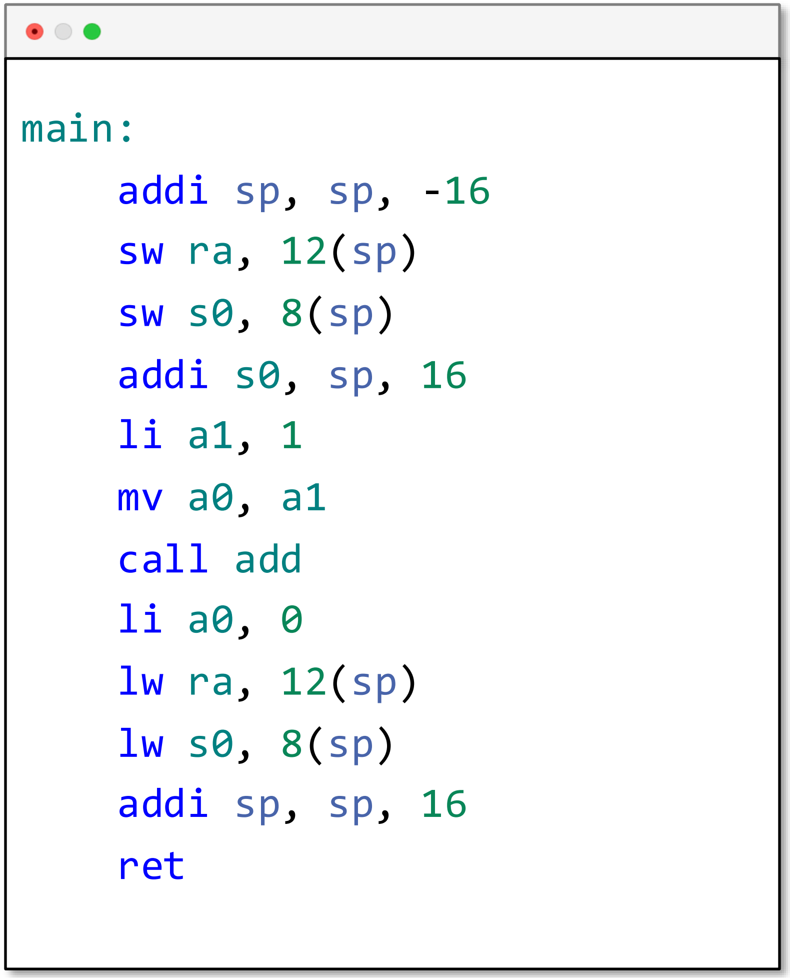
jalr rd,imm12(rs1)

These instructions save pc + 4 into rd.

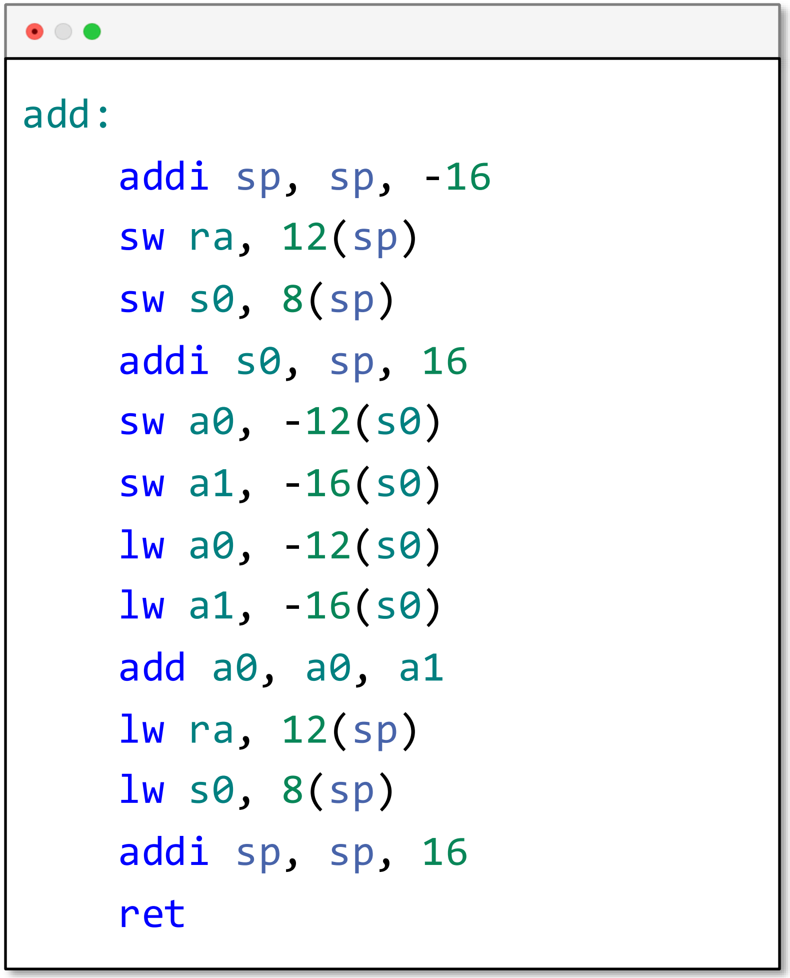
When we enter fun4, `ra` holds the address of the instruction after this `jal` in the callers routine.

```
jal ra, fun4
//next instruction
```

For those who want to practice…

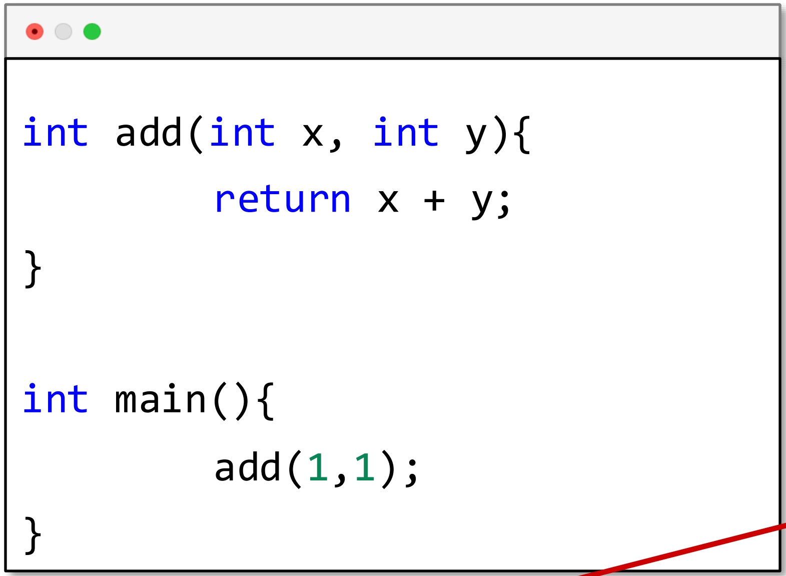


```
main:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    addi s0, sp, 16
    li a1, 1
    mv a0, a1
    call add
    li a0, 0
    lw ra, 12(sp)
    lw s0, 8(sp)
    addi sp, sp, 16
    ret
```

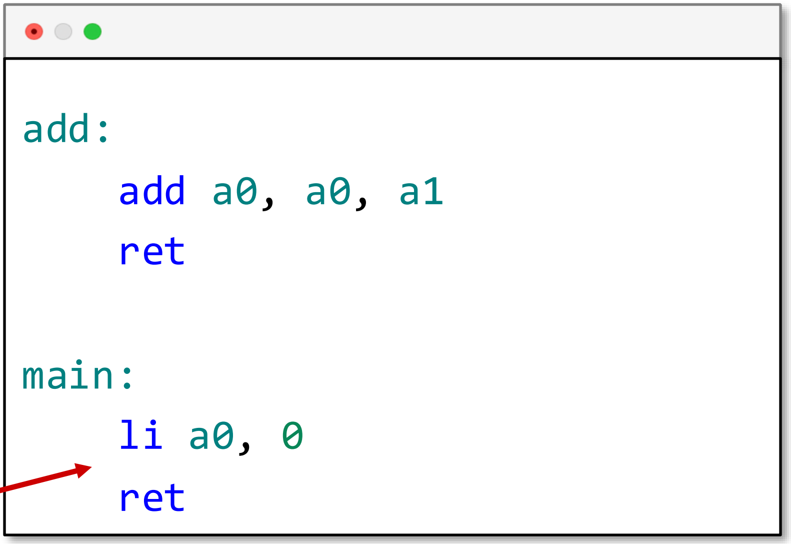


```
add:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    addi s0, sp, 16
    sw a0, -12(s0)
    sw a1, -16(s0)
    lw a0, -12(s0)
    lw a1, -16(s0)
    add a0, a0, a1
    lw ra, 12(sp)
    lw s0, 8(sp)
    addi sp, sp, 16
    ret
```

What does the compiler say?



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

int main(){
        add(1,1);
}
```



```
add:
    add a0, a0, a1
    ret

main:
    li a0, 0
    ret
```

Why?

Well, we never use the result of add! So why waste time calling it?

Why do we load 0 into `a0`?

Take a look at this Macro defined in stdlib.h #define EXIT_SUCCESS 0

Yup.

Next time!

- More C to RISC-V!