Introduction to Computer Architecture
Chapter 2

Instructions: Language of the Computer - 3

Hyungmin Cho

Department of Computer Science and Engineering
Sungkyunkwan University

Procedure Calling

Caller

```
int main() {
    …
    …
    arg2 = val1 * 2;
    result = compute(arg1, arg2);
    …
    …
}
```

Callee

```
int compute(int p1, int p2) {
    int temp1;
    int temp2;
    …
    temp1 = p1 * p2;
    temp2 = temp1 + 10;
    …
    return temp2;
}
```

1. How to pass parameters (`arg1` and `arg2`) to the **callee** (function)
2. How to reserve memory space for the callee's **local variables**
3. How to transfer the result (return value) back to the **caller**
4. How to return to the **caller**

Procedure? Function?

- Are they the same? … Not really.

- Procedure is a group of instructions that are invoked to perform a designated task.

- There is no concept of “Function” in assembly language

- We use a special rule to support “C”-style function in assembly
  → calling convention

RISC-V Register Usage

Register Number	Name	Usage
`x0`	`zero`	Constant 0 (hardwired)
`x1`	`ra`	Return address
`x2`	`sp`	Stack pointer
`x3`	`gp`	Global pointer
`x4`	`tp`	Thread pointer
`x5-x7, x28-x31`	`t0 - t6`	Temporaries
`x8`	`s0 / fp`	Frame pointer
`x9, x18-x27`	`s1 - s11`	Saved registers
`x10-x11`	`a0 - a1`	Function arguments / results
`x12-x17`	`a2 - a7`	Function arguments

Argument Passing

- Set register `x10 – x17` with the arguments in order **before calling**
  (a0) (a7)
  the function.

- If the function takes more than 8 arguments, the rest of the arguments are passed to the callee using memory.

Return Value

- Use register `x10 – x11`
    - `x10 [31:0]`
    - `x11 [63:32]`
- If the return value is larger than 64-bit, the return value is passed through memory.

Procedure Call Instructions

- Procedure call: jump-and-link instruction

  `jal x1, ProcedureLabel`

  - Jumps to the target address
  - Save the address of the next instruction in **`x1`**
  - **`x1`**'s register name is **`ra`** (return address)

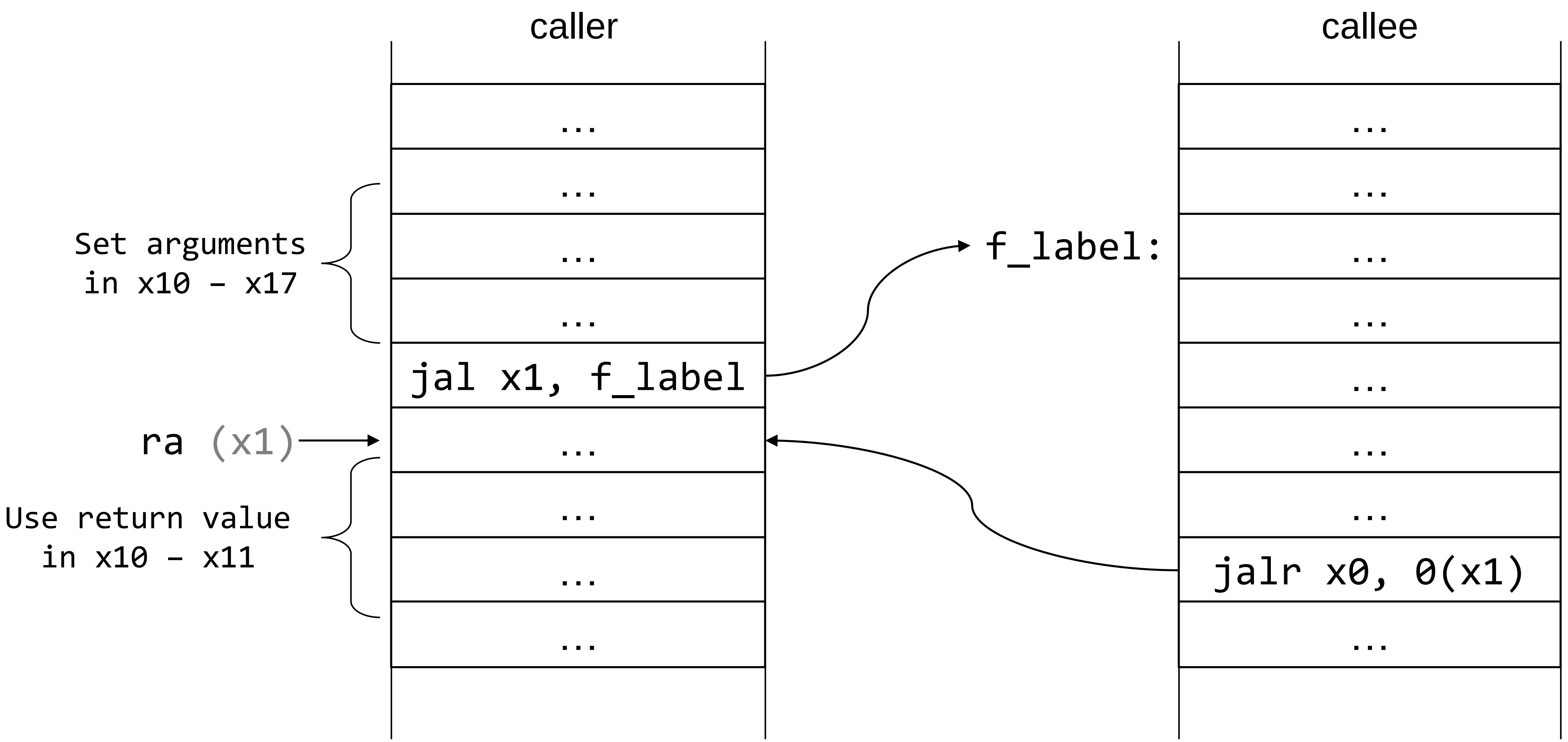
- Procedure return: jump-and-link register instruction

  `jalr x0, Offset(x2)`

  - Jump to offset + address in `x2`
  - Address of the next instruction → `x0`

  - Can also be used for computed jumps
    - e.g., for case/switch statements

RISC-V Procedure Call



Memory Space for Procedure (Function)

- A procedure needs its own memory space for the the local variables.

```
int func1(…) {
  int var1;
  int arr2[2];
  …
}
```

  - If all local variables can fit in the registers, a procedure may not need to use memory space (compiler optimization)

  - Even all local variables can fit in the registers, the values may need to be saved in the memory to call another function (procedure).

x5

```
int func1(…) {
  int var1;
  var1 = 123
  …
  func2(…)
  var2 = var1*2
  …
}
```

```
int func2(…) {
  int var10;
  …
  var10 = 456;
  …
}
```

x5

Memory Space for Procedure (Function)

- How to reserve space for each procedure?
  - Fixed location for each procedure **is not an ideal solution**
    - Can't know which procedures will be called in a program in advance
    - Recursion!

Fixed address?

```
int func1(…) {
  int var1;
  int arr2[2];
  …

}
```

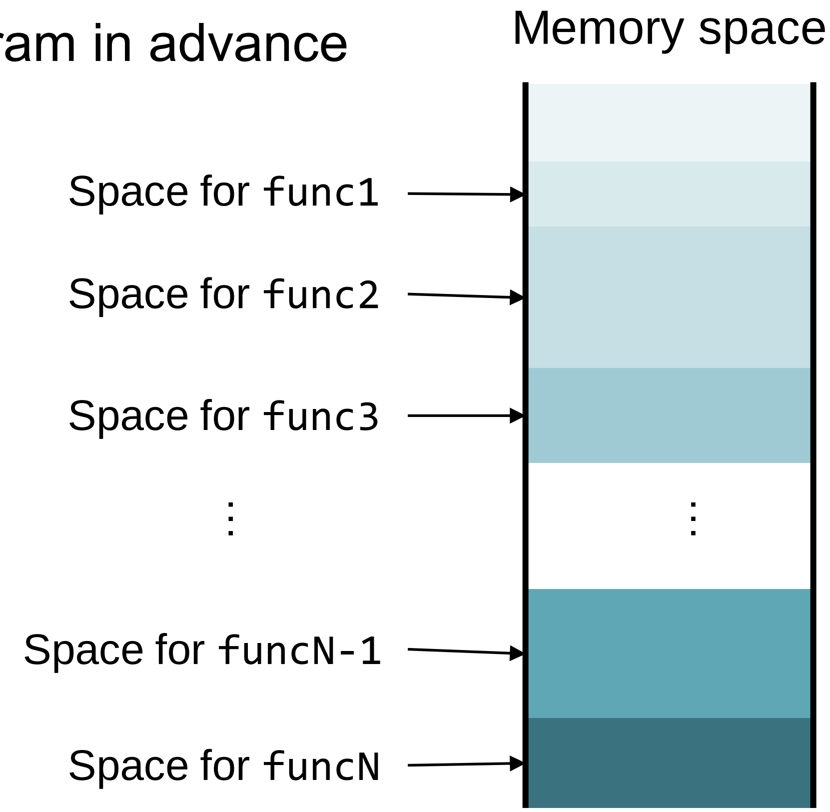
0x1000

0x1004

```
int func2(…) {
  int arr20[100];
  int var10;
  …

}
```

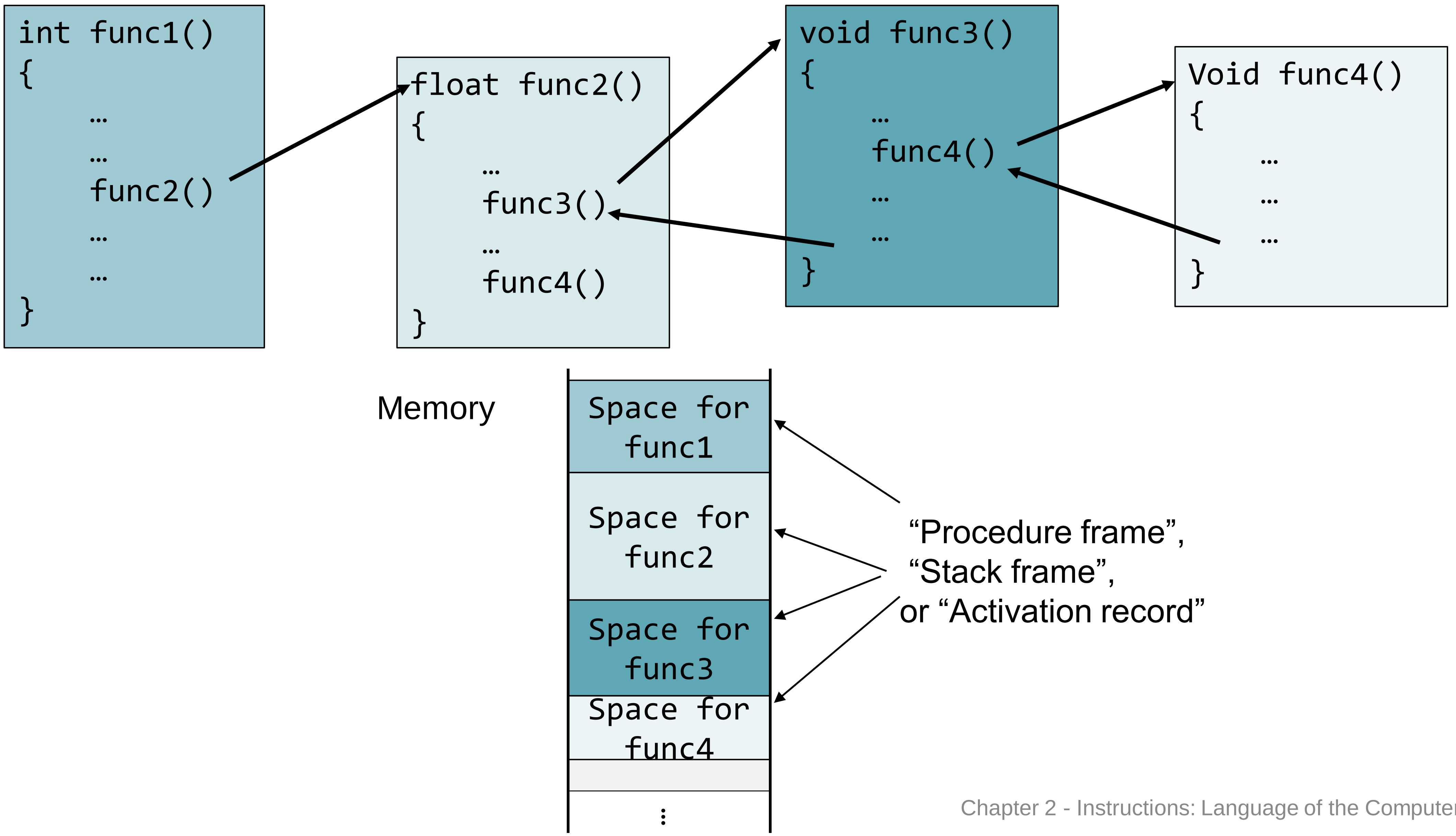
0x2000

0x2190

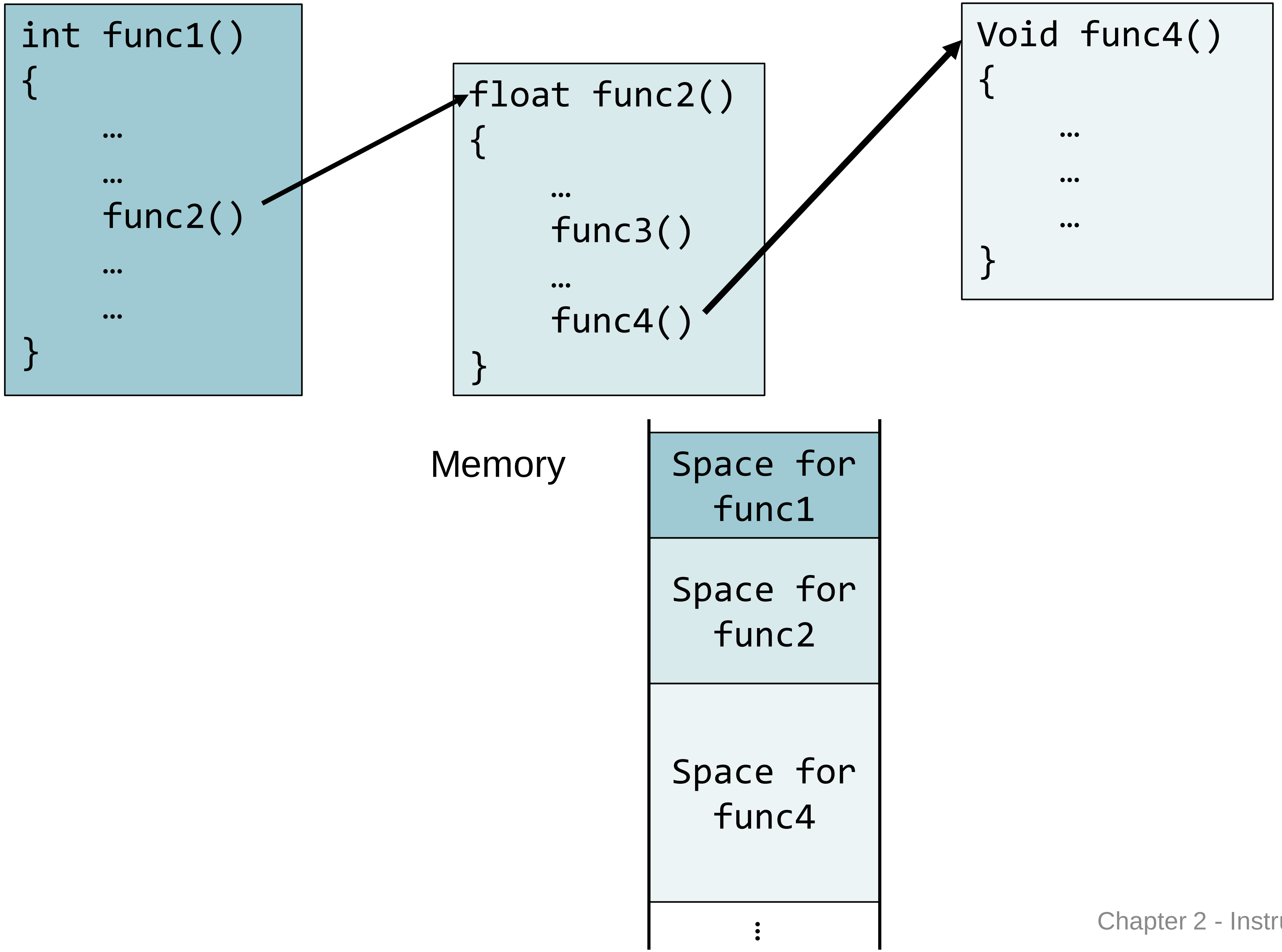


- Solution?
  - "Allocate" the space when the function is called / "Deallocate" when returns
  - Use relative addressing within the allocated space.

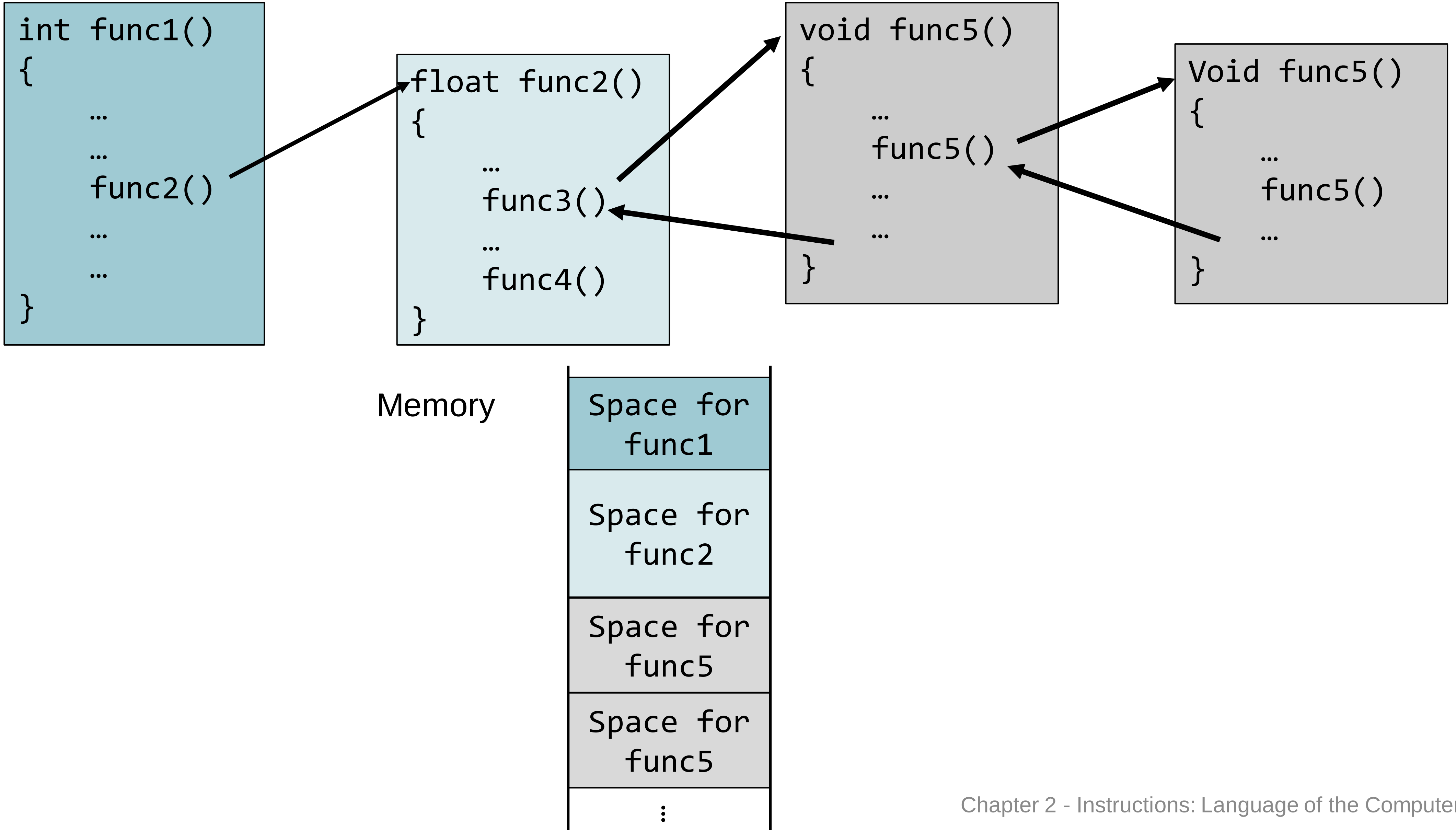
Stack


int func1()
{
…
…
func2()
…
…
}
float func2()
{
…
func3()
…
func4()
}
void func3()
{
…
func4()
…
…
}
Void func4()
{
…
…
…
}
Memory
Space for func1
Space for func2
Space for func3
Space for func4
⋮
"Procedure frame",
"Stack frame",
or "Activation record"

Stack

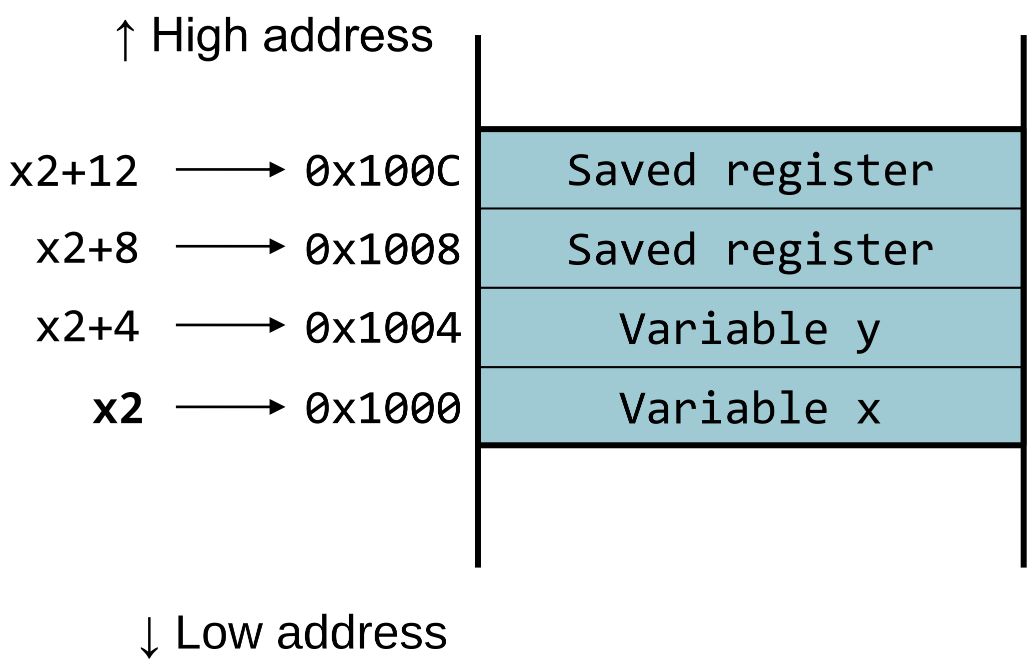

int func1()
{
…
…
func2()
…
…
}
float func2()
{
…
func3()
…
func4()
}
Void func4()
{
…
…
…
}
Memory
Space for func1
Space for func2
Space for func4
⋮

Stack


int func1()
{
…
…
func2()
…
…
}
float func2()
{
…
func3()
…
func4()
}
void func5()
{
…
func5()
…
…
}
Void func5()
{
…
func5()
…
}
Memory
Space for func1
Space for func2
Space for func5
Space for func5
⋮

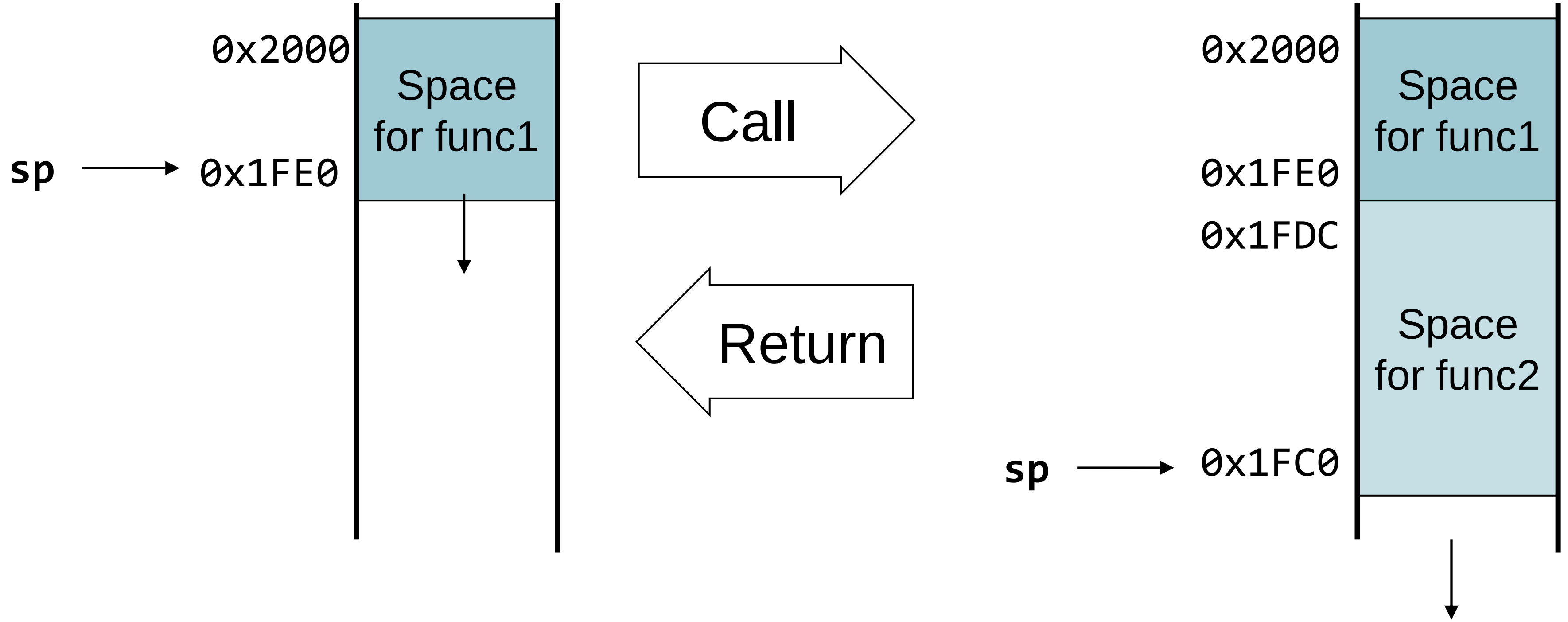
Managing & Accessing Stack (1)

- A stack frame can be placed at any location in memory
  - Relative addressing is required
- Stack pointer register x2 (sp) points to the lowest address in the current stack frame



Managing & Accessing Stack (2)

- Stack grows from higher address to lower address

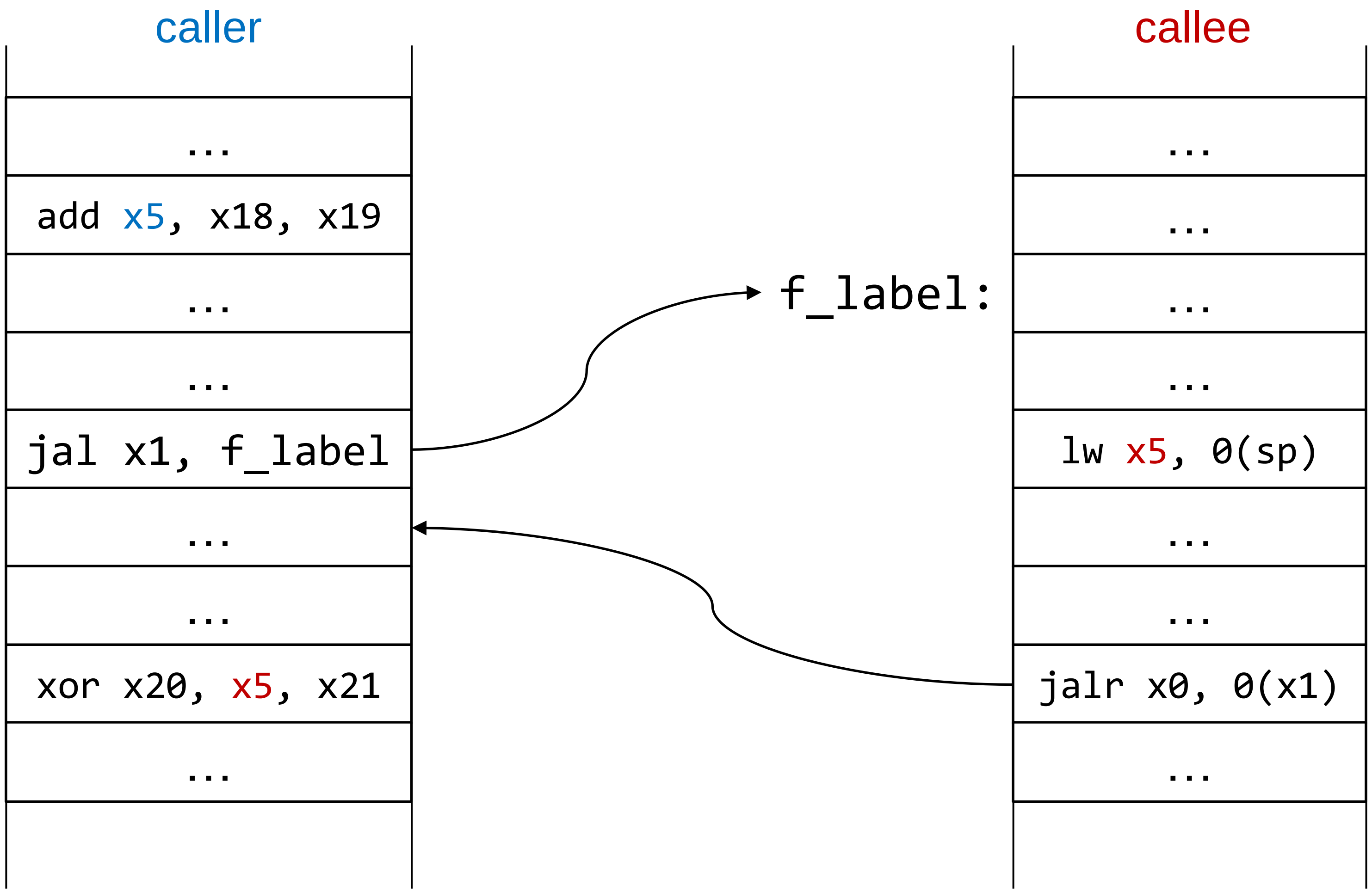


Managing & Accessing Stack (3)

- Each procedure needs to know how much stack space it needs
  - High-level language: the compiler automatically computes the required space
  - Assembly language: the programmer need to determine the required size

- Suppose a function requires 20 bytes of space (e.g., 5 x 4-byte values)…
  - When the function starts, **decrement** `sp` by 20
    - `addi sp, sp, -20`
  - To access any stack value, use `sp` as the base register
    - `sw x18, 4(sp)`
  - When the function returns, **increment** `sp` by 20
    - `addi sp, sp, 20`

Saving Registers



Register Saving Convention

Register Number	Name	Usage
x0	zero	Constant 0 (hardwired)
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-x7, x28-x31	t0 - t6	Temporaries
x8	s0 / fp	Frame pointer
x9, x18-x27	s1 - s11	Saved registers
x10-x11	a0 - a1	Function arguments / results
x12-x17	a2 - a7	Function arguments

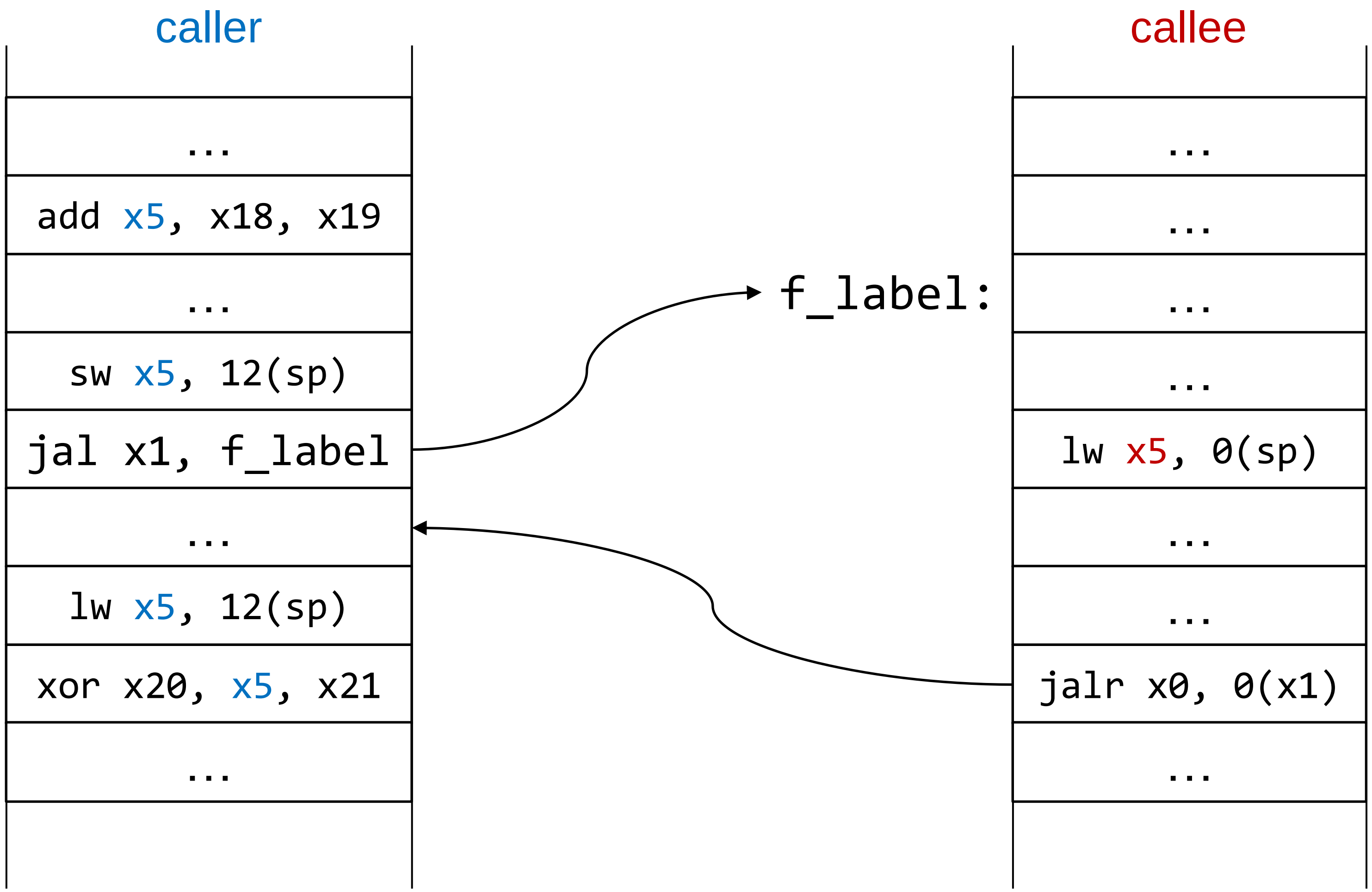
Caller-save registers

If the caller wants to **use value in these registers** *after* calling a function,
→ caller need to store these registers in its stack frame before calling the function

Callee-save registers

If the callee wants to **use these registers**,
→ callee need to store these registers in its stack frame and restore those before the return

Caller-save Registers



Callee-save Registers

caller

```
...
ori x19, x20, 0xF
...
...
jal x1, f_label
...
...
xor x21, x19, x22
...
```

callee

```
         ...
         ...
f_label: addi sp, sp, -12
         ...
         sw x19, 8(sp)
         mul x19, x5, x6
         ...
         lw x19, 8(sp)
         addi sp, sp, 12
         jalr x0, 0(x1)
```

Avoiding Unnecessary Register Save & Restore

- If a value is used for a short period, use the “Temporary” registers (caller-save) instead of the “Saved” registers (callee-save)

- Use “Saved” registers if a value is created before a function call and used after the function call.

- Use “Argument” and “Results” registers as temporary registers if possible

Leaf Procedure Example

- C code:

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

  - Arguments `g, …, j` in `x10, x11 …, x13 (a0, a1, …, a3)`
  - `f` in `x20 (s4)`
    - hence, need to save the caller's `x20` on stack
  - Temporaries `x5 (t0), x6 (t1)`
  - Return value in `x10 (a0)`

Leaf Procedure Example

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

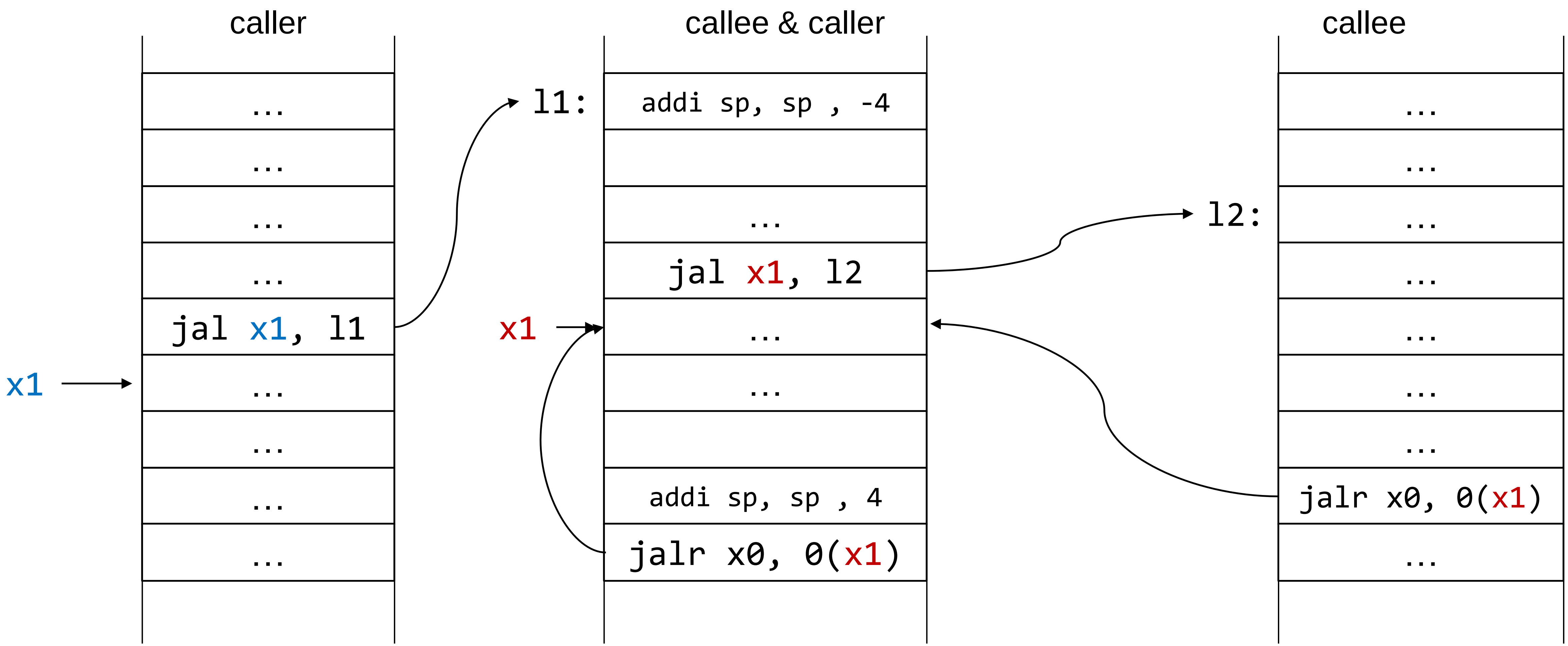
```
leaf_example:
  addi sp, sp, -4        Allocate stack space
  sw   x20, 0(sp)        Save x20 on stack
  add  x5, x10, x11      
  add  x6, x12, x13      Procedure body
  sub  x20, x5, x6       
  addi x10, x20, 0       Set return value
  lw   x20, 0(sp)        Restore x20
  addi sp, sp, 4         De-allocate stack space
  jalr x0, 0(x1)         Return
```

Leaf Procedure Example (2)

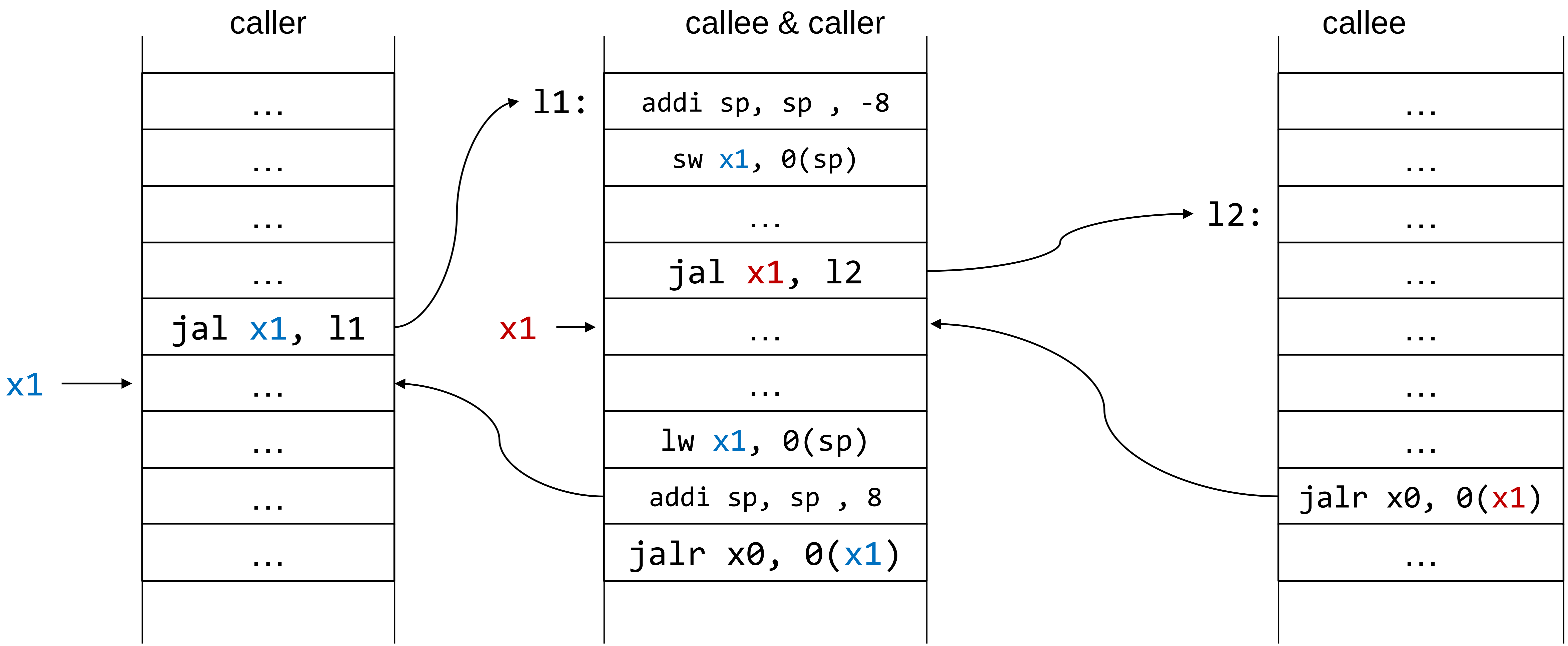
```
int leaf_example (int g, h, i, j)
{
  int f;
  f = (g + h) - (i + j);
  return f;
}
```

```
leaf_example:
  add  x5, x10, x11
  add  x6, x12, x13
  sub  x10, x5, x6
  jalr x0, 0(x1)
```

MIPS Procedure Calls – Non-Leaf



MIPS Procedure Calls – Non-Leaf



Non-Leaf Procedure Example

- C code:

```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

  - Argument `n` in `x10`
  - Result in `x10`

  - Need to store return address (x1) in stack
  - Need to store argument (`x10`) in stack
    - `n` is used again after calling `fact(n-1)`

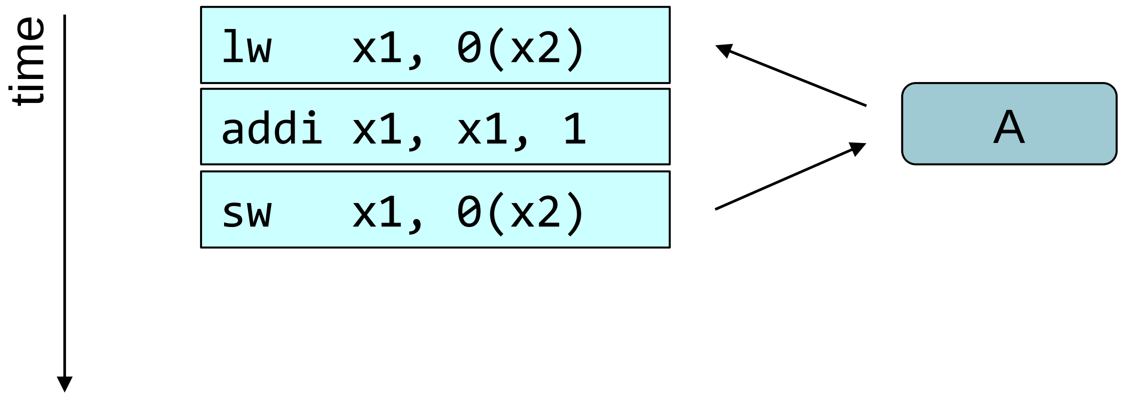
Non-Leaf Procedure Example

```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

```
fact:
     addi sp,  sp, -8       # allocate stack for two 4-byte items
     sw   x1,  4(sp)        # save return address
     sw   x10, 0(sp)        # save argument (n)

     addi x5, x0, 1
     bge  x10, x5, L1       # test if n < 1

     addi x10, x0, 1        # if n < 1, result is 1
     addi sp, sp, 8         # de-allocate stack space
     jalr x0, 0(x1)         # return

L1:  addi x10, x10, -1      # if n >= 1 set n-1 as argument
     jal  x1, fact          # recursive call

     lw   x5, 0(sp)         # restore original n
     mul  x10, x5, x10      # multiply to get result

     lw   x1, 4(sp)         # retore return address
     addi sp, sp, 8         # de-allocate stack space
     jalr x0, 0(x1)         # return
```

Synchronization

- A = A + 1;



Synchronization

- A = A + 1;

time

```
lw   x1, 0(x2)
addi x1, x1, 1
sw   x1, 0(x2)
```

A

- A = A + 2;

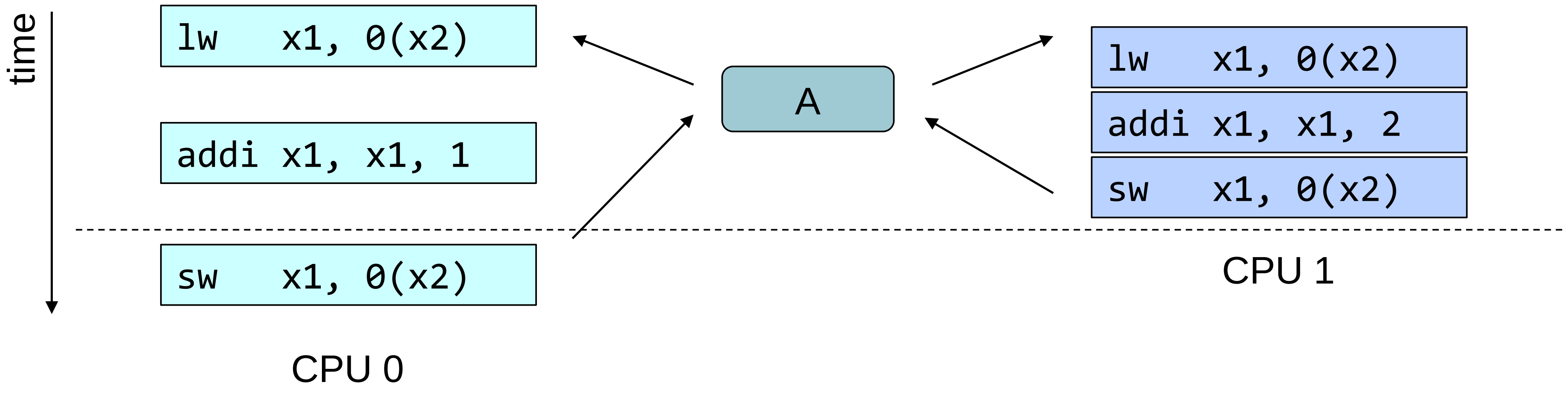
```
lw   x1, 0(x2)
addi x1, x1, 2
sw   x1, 0(x2)
```

CPU 1

A = A + 3

Synchronization: Race Condition

- A = A + 1;
- A = A + 2;



A = A + 1 ??

Data “race”

Synchronization: Atomic Operation (1)

- Special memory instructions (RISC-V “A” extension)

`lw   x1, 0(x2)`	→	`lr.w   x1, (x2)`
`addi x1, x1, 1`		`addi x1, x1, 1`
`sw   x1, 0(x2)`	→	`sc.w   x1, (x2)`

  - `lr.w` : load word **reserved**
    - Load from address in x2 to x1 (no offset)
  - `sc.w` : store word **conditional**
    - Store from x1 to address in x2 (no offset)

- `sc.w` writes to the memory only if the target location wasn’t modified after `lr.w`
  - → x1 is set to 0
- If the memory location was changed by someone else, store does not happen
  - → x1 is set to a non-zero value

Synchronization: Atomic Operation (2)

- A = A + 1;

```
lr.w   x1, (x2)
addi x1, x1, 1
sc.w   x1, (x2)
```

CPU 0

- Write does not happen
- x1 becomes non-zero

- Your program may choose to retry the task if x1 is non-zero.

- A = A + 2;

A

```
lr.w   x1, (x2)
addi x1, x1, 2
sc.w   x1, (x2)
```

CPU 1

- Write happens
- x1 becomes 0

- Your program can proceed to the next task if x1 is 0.

Synchronization

- In high-level languages, you can use these basic synchronization methods through library functions

- In Linux, POSIX pthread library
  - `pthread_mutex_lock()`
  - `pthread_mutex_unlock()`