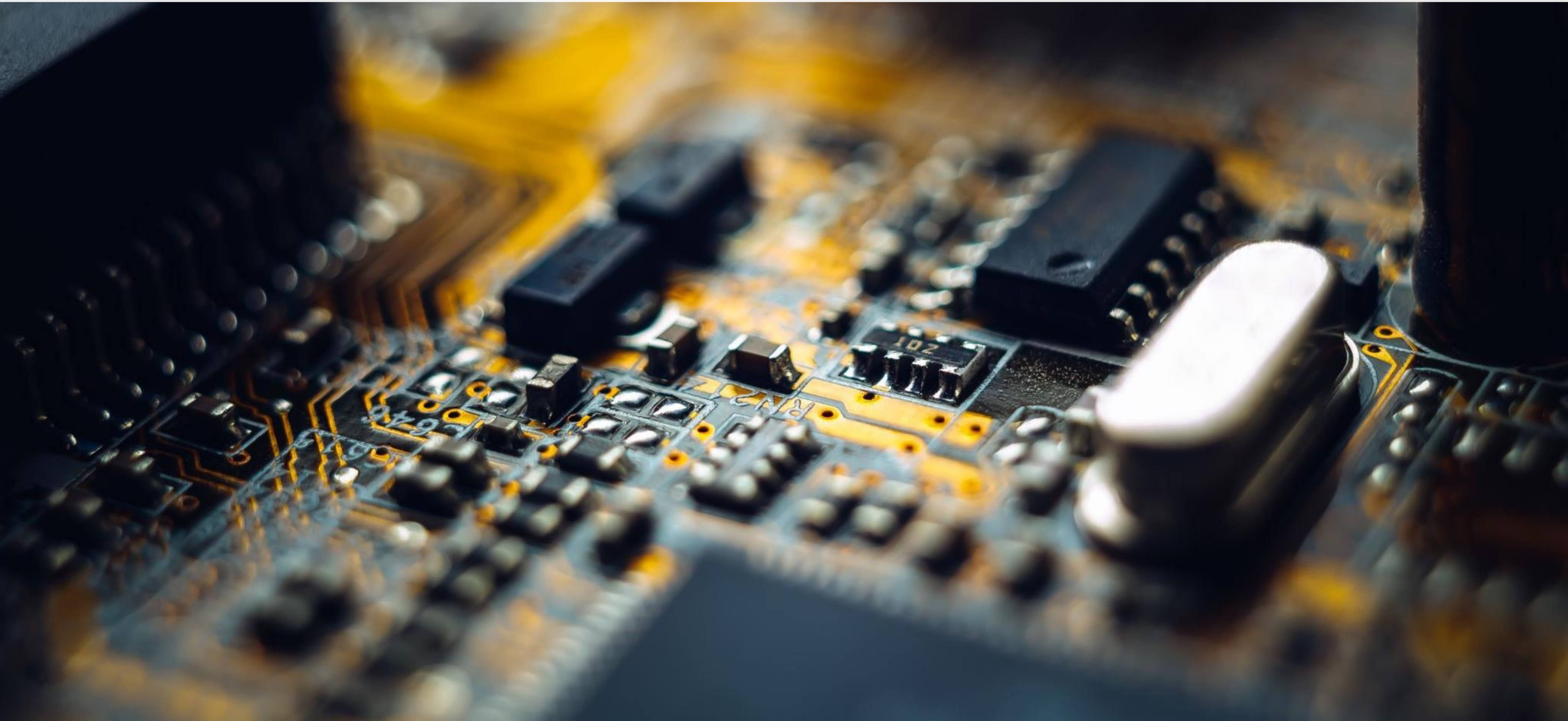
High Performance Computing in Julia

from the ground up.

SIMD & The Stack and the Heap

2/10

SIMD - Vector Instruction Sets



SIMD – Single Instruction Multiple Data

- Modern CPUs now have special, **wide**, registers that can store multiple numbers.
- A 256-bit register can **pack** 4 64-bit values or 8 32-bit values together for a **single** load operation.
- ALU contains special circuits to process all packed numbers at the **same** time.
- This is **hardware level** parallelism, allowing the processing of multiple elements at the same time.

Example: Vector Addition

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \end{bmatrix}$$

- Each operation is **independent** and can be done at the same time
- Can load both numbers of each vector in a single operation
- Addition of all numbers happens in **one** clock cycle
- Storage back in memory happens in **one** cycle

SSE and AVX (x86 architecture)

- Streaming SIMD Extensions (SSE) supported on most modern x86 CPUs
- Advanced Vector Extensions (AVX) expands on modern chips with AVX-512 for larger vectors
- Usually included on higher end processors, typically workstation/server processors like Xeon, Epyc, Threadripper etc.

Example SIMD in Julia

Traditional `for` Loop

```julia
function custom_sum(numbers)
    total = 0
    for x in numbers
        total += x
    end
    return total
end
```

Example SIMD in Julia

Traditional `for` Loop

```
function custom_sum(numbers)
    total = 0
    for x in numbers
        total += x
    end
    return total
end
```

`@simd` Macro

```
function custom_sum_simd(numbers)
    total = 0
    @simd for x in numbers
        total += x
    end
    return total
end
```

Macro: A metaprogramming techniques which takes existing code as an input, and manipulates it to produce more code.

Example SIMD in Julia

Traditional `for` Loop

```
function custom_sum(numbers)
    total = 0
    for x in numbers
        total += x
    end
    return total
end
```

```
julia> numbers = rand(Float32, 128);
julia> custom_sum(numbers)
65.52533f0
```

`@simd` Macro

```
function custom_sum_simd(numbers)
    total = 0
    @simd for x in numbers
        total += x
    end
    return total
end
```

```
julia> custom_sum_simd(numbers)
65.52534f0
```

Example SIMD in Julia

Traditional `for` Loop

```
function custom_sum(numbers)
    total = 0
    for x in numbers
        total += x
    end
    return total
end
```

`@simd` Macro

```
function custom_sum_simd(numbers)
    total = 0
    @simd for x in numbers
        total += x
    end
    return total
end
```

*Floating point arithmetic is **not** associative!

$$(a + b) + c \neq a + (b + c)$$

```
julia> numbers = rand(Float32, 128);
julia> custom_sum(numbers)
65.52533f0
```

$\neq$

```
julia> custom_sum_simd(numbers)
65.52534f0
```

Benchmarking SIMD

*Functions benchmarked here are **slightly** different

```
julia> using BenchmarkTools
julia> @btime custom_sum($numbers)
 84.265 ns (0 allocations: 0 bytes)
65.52533f0
julia> @btime custom_sum_simd($numbers)
 7.500 ns (0 allocations: 0 bytes)
65.52534f0
```

- Around 11x faster for adding 5 characters.

The Stack and the Heap

A Stack

- A stack is an extremely common data structure used throughout CS
- **Linear data structure** where you can only access items from the top

Example: Computing algebraic expression

How **write** the addition of a and b?

$$\mathrm{add}(a, b) \qquad a + b \qquad a\, b\, +$$

Prefix **Infix** **Postfix**

Example: Postfix evaluation with a stack

- Take the expression in **infix:**

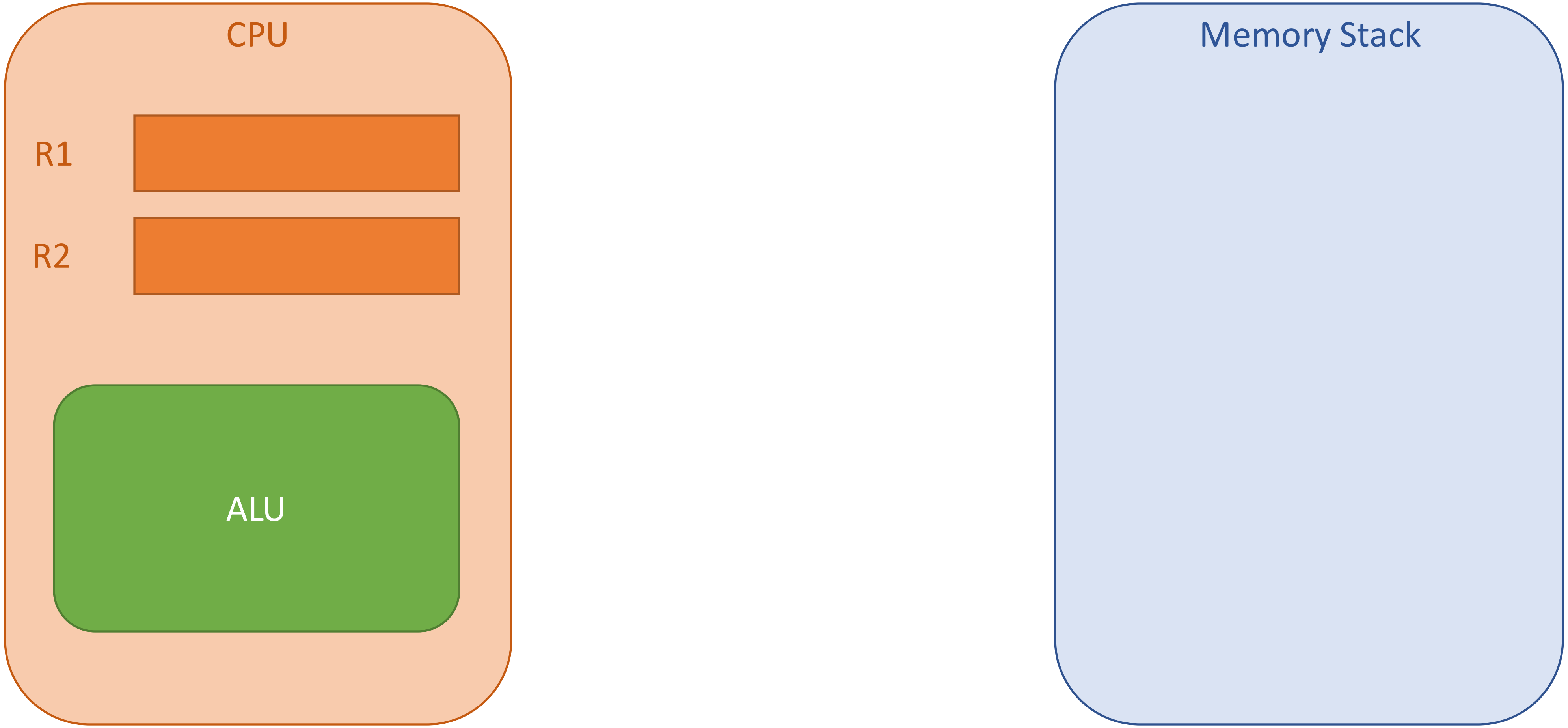
$$5x(x + 2) - 1$$

- In **postfix** we get:

$$1\ 5\ x\ 2\ x + \times \times -$$

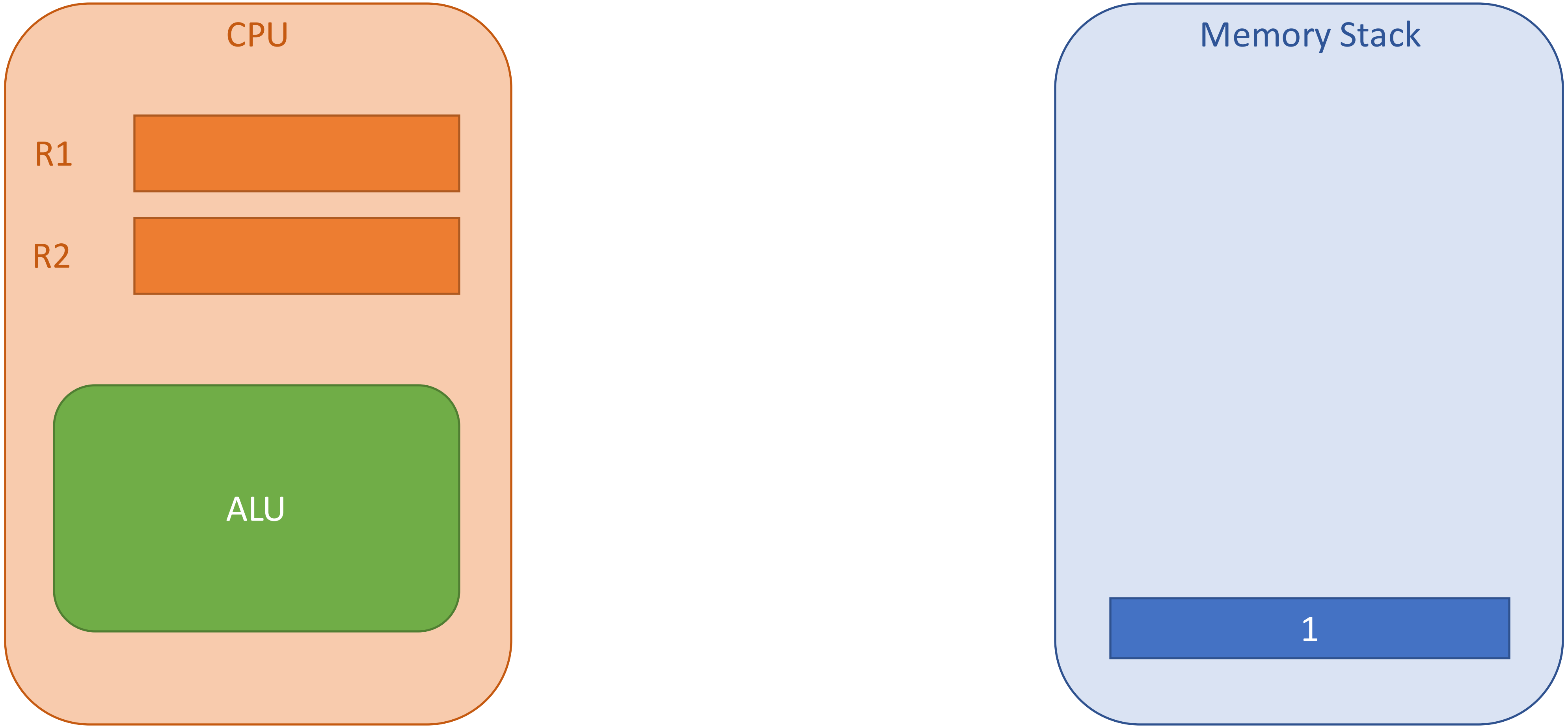
Example: Postfix evaluation with a stack

$$1\ 5\ x\ 2\ x + \times \times -$$



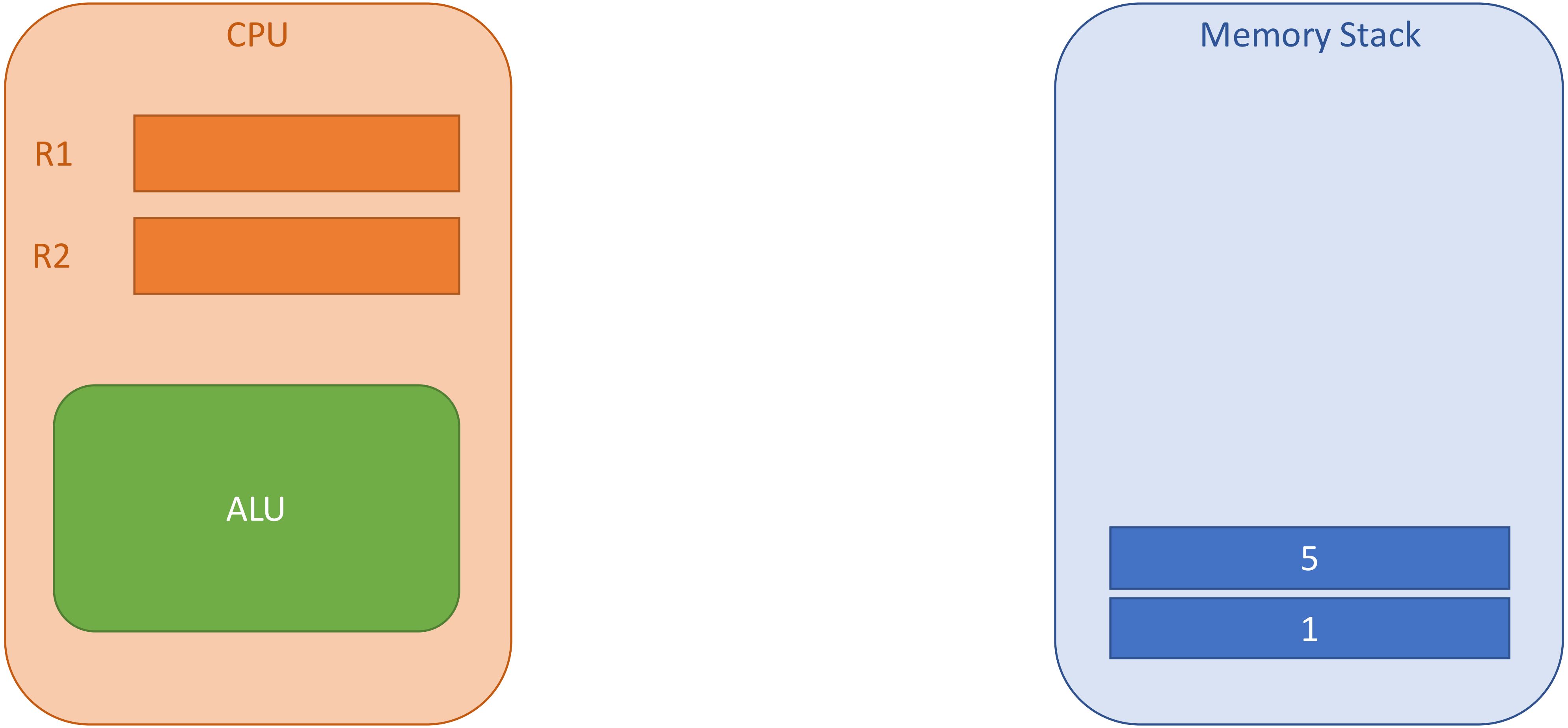
Example: Postfix evaluation with a stack

$1\ 5\ x\ 2\ x + \times \times -$



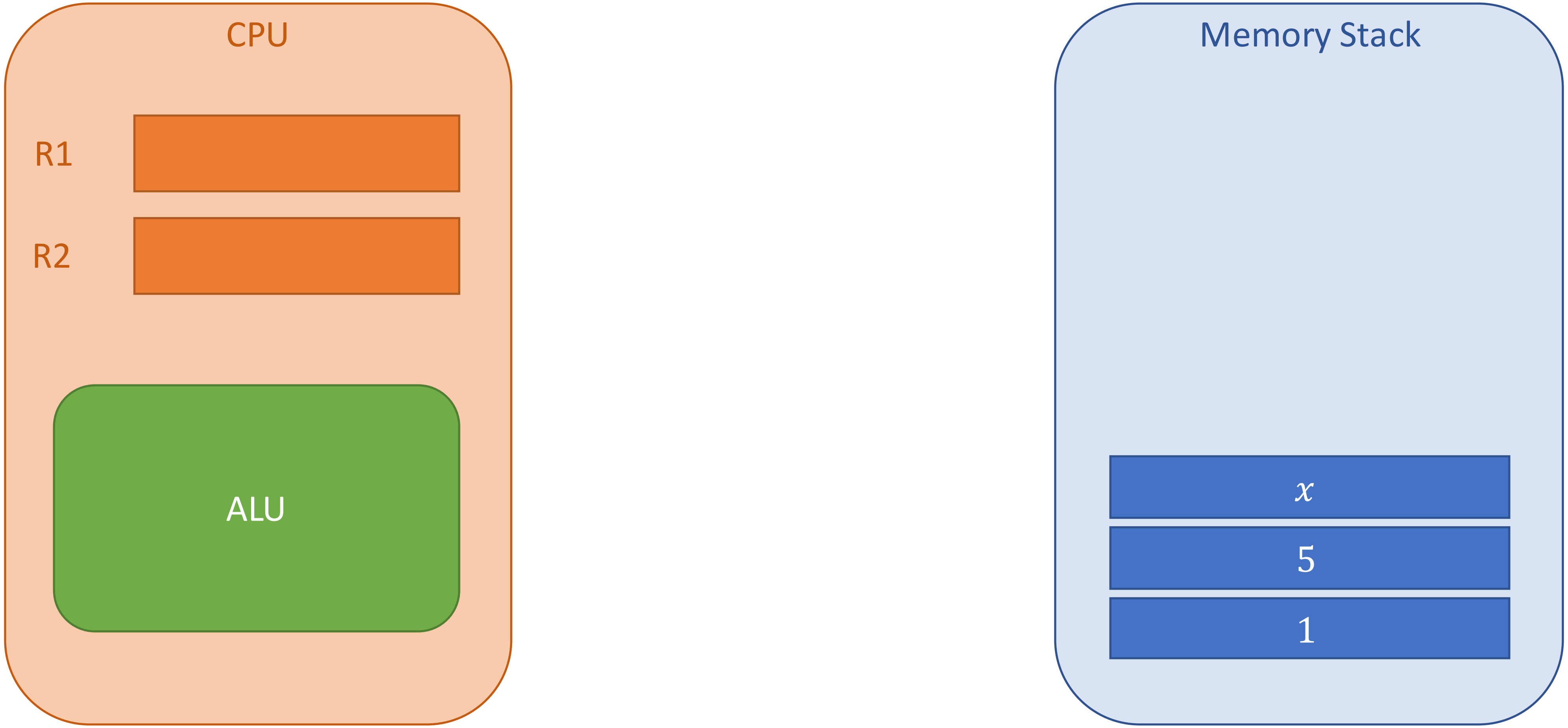
Example: Postfix evaluation with a stack

$$1\ 5\ x\ 2\ x + \times \times -$$



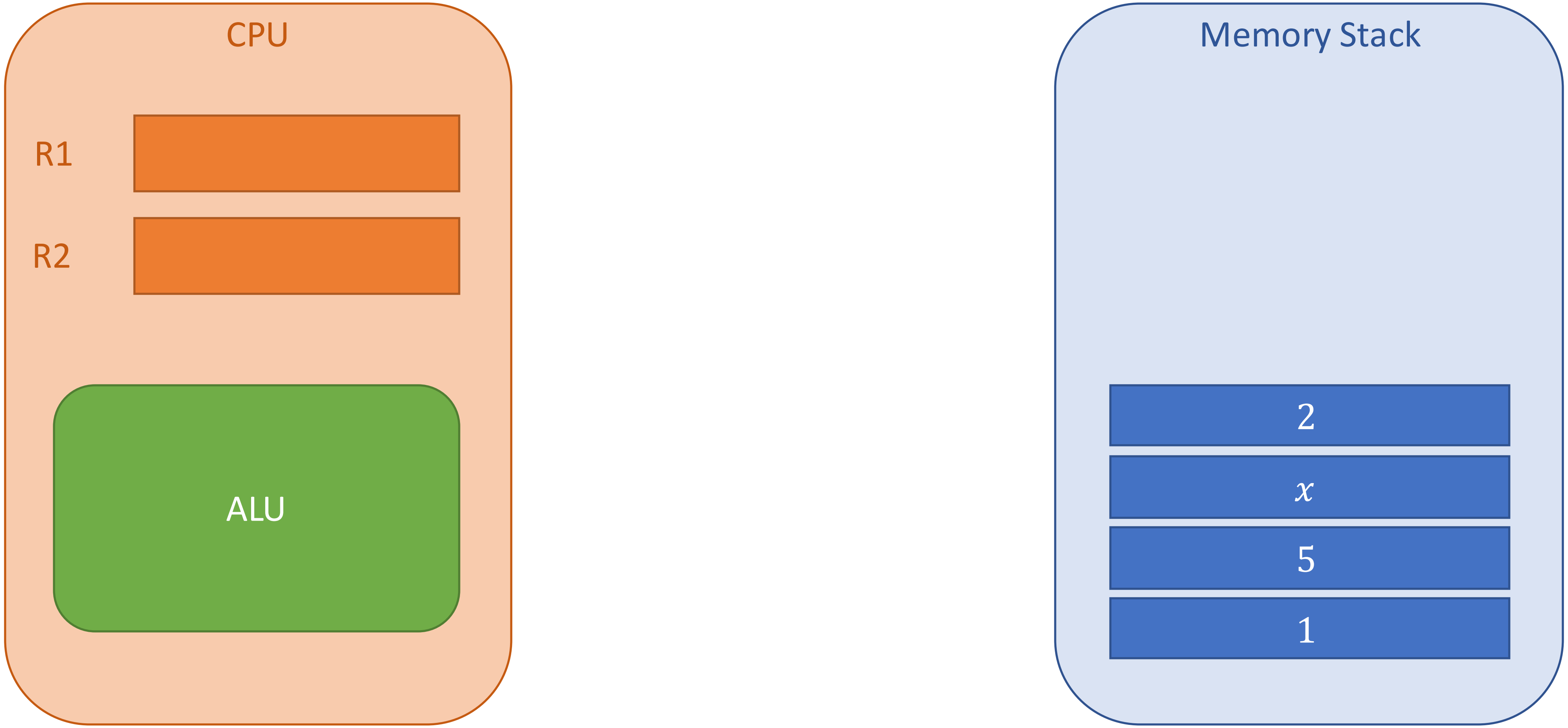
Example: Postfix evaluation with a stack

$1\ 5\ x\ 2\ x + \times \times -$



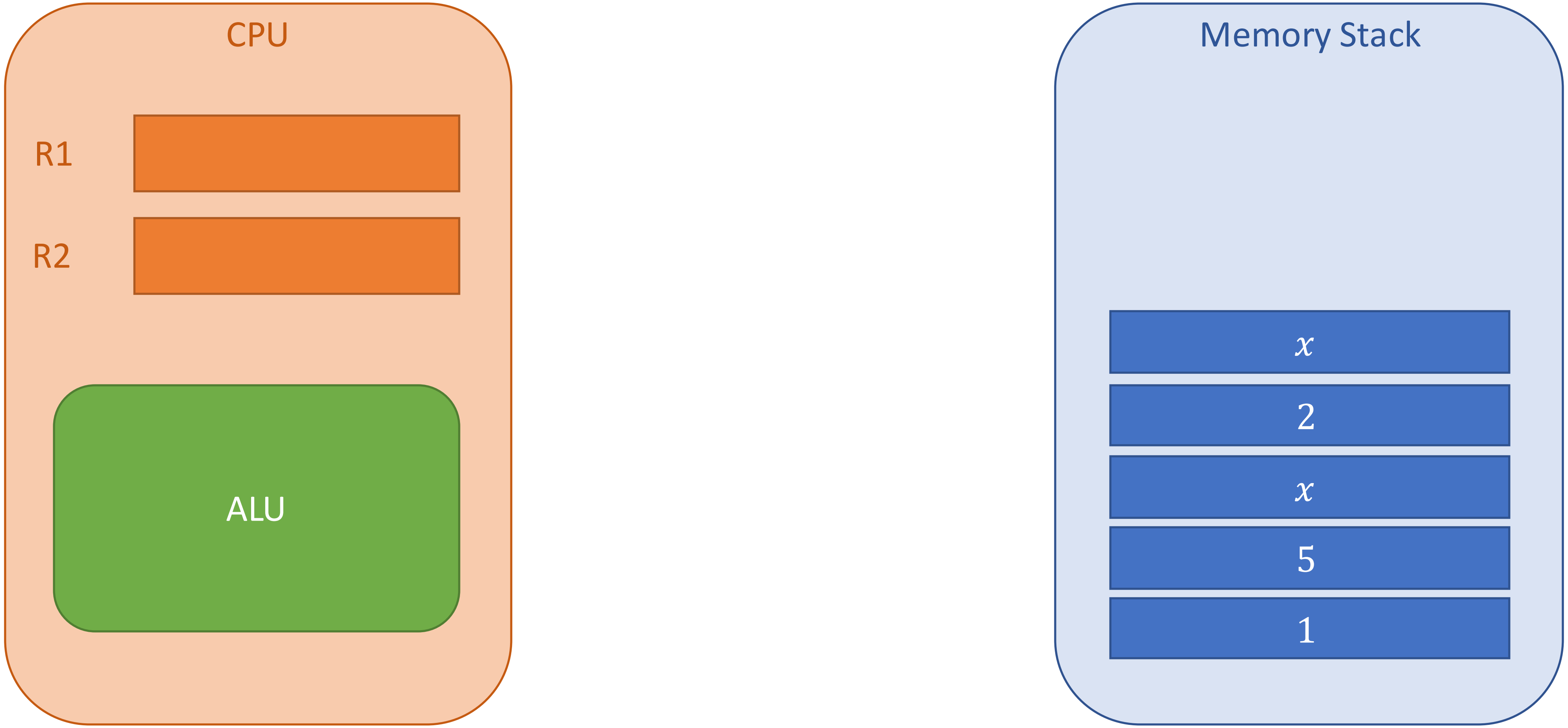
Example: Postfix evaluation with a stack

$$1\ 5\ x\ 2\ x + \times \times -$$



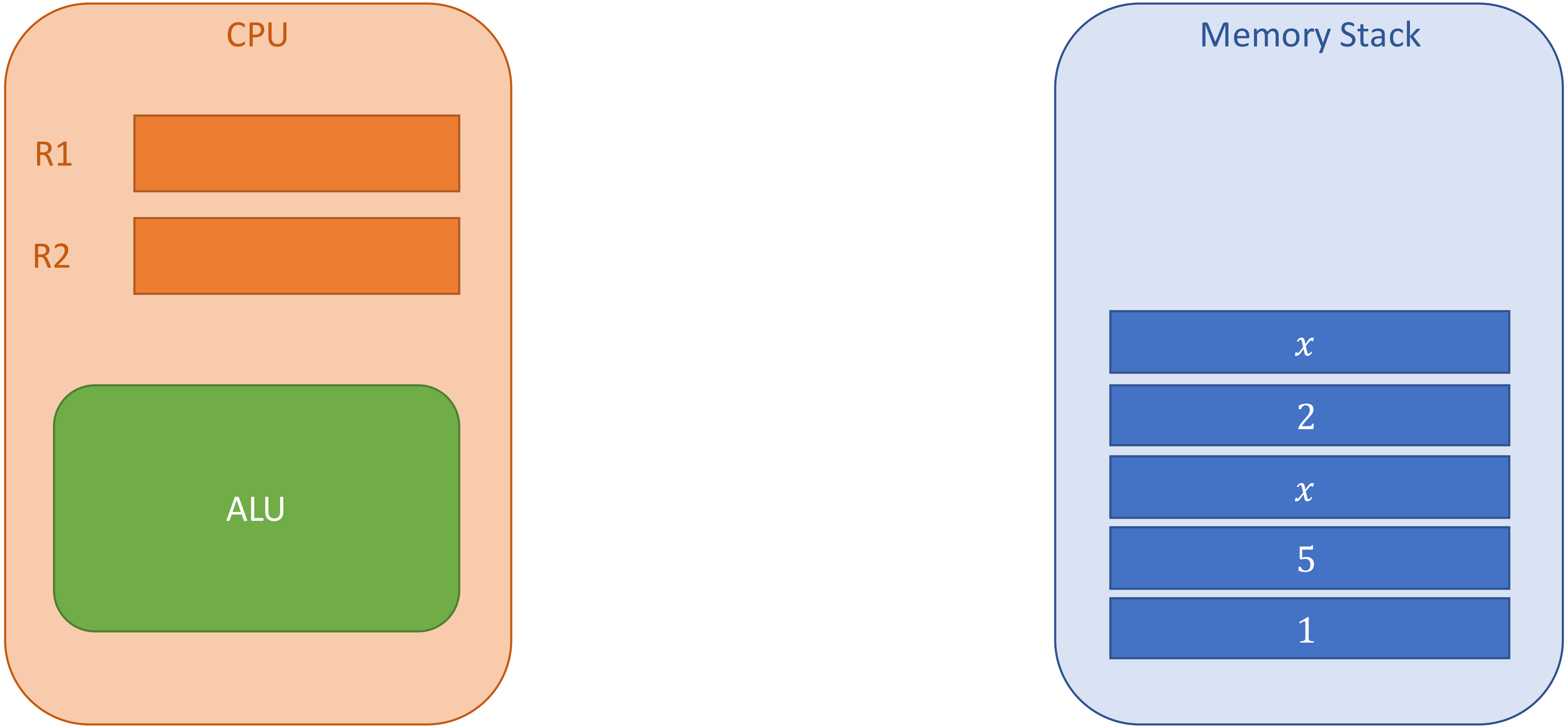
Example: Postfix evaluation with a stack

$$1\ 5\ x\ 2\ \boxed{x}\ +\ \times\ \times\ -$$



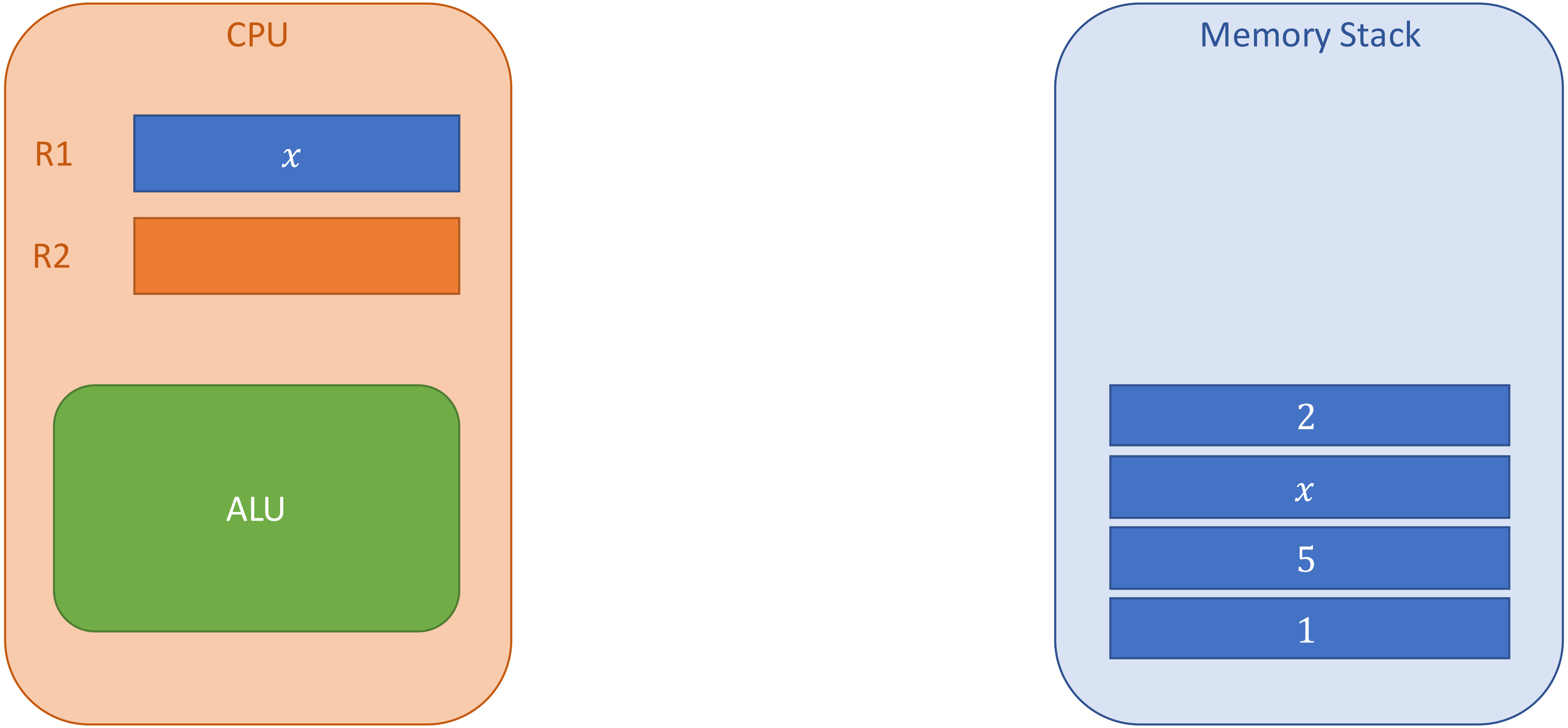
Example: Postfix evaluation with a stack

$$1\ 5\ x\ 2\ x + \times \times -$$



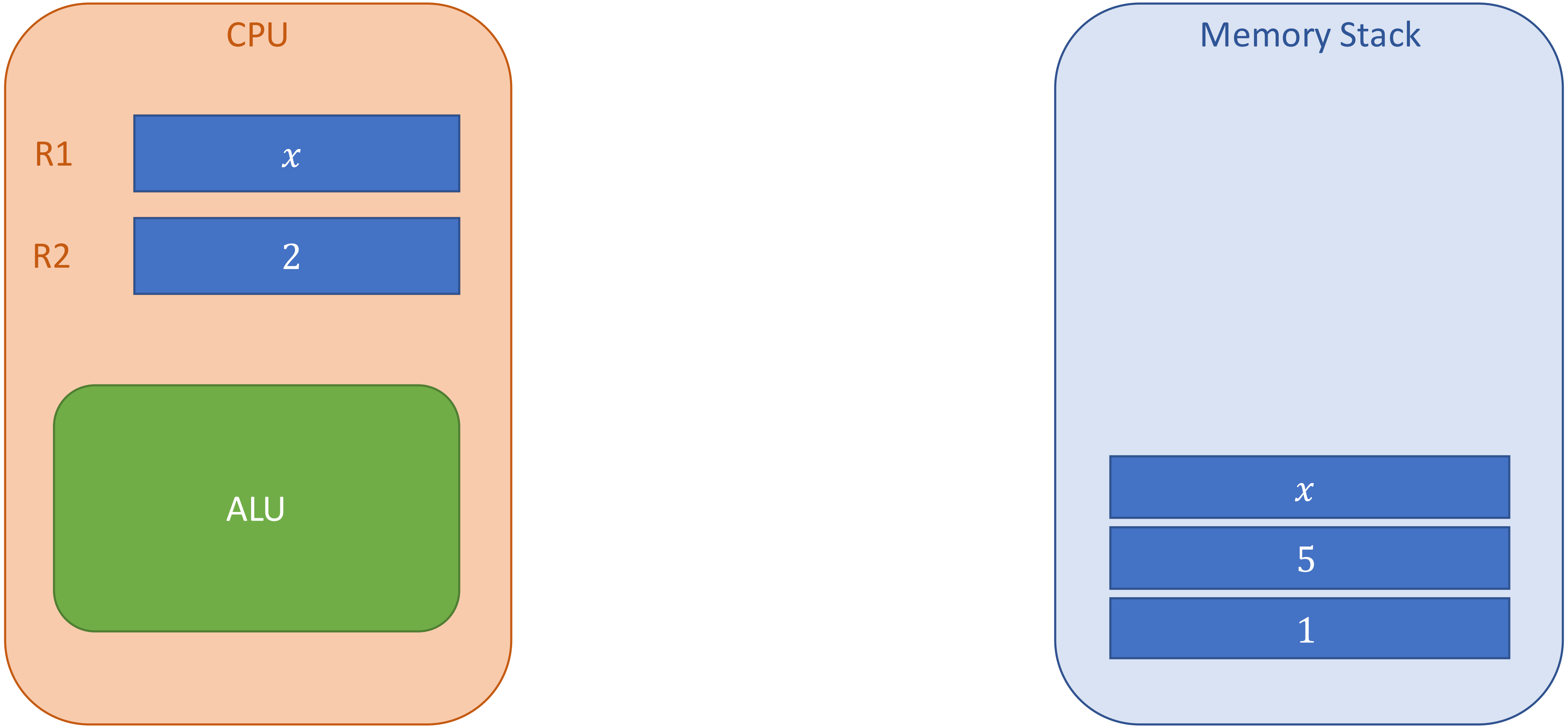
Example: Postfix evaluation with a stack

$1\ 5\ x\ 2\ x\ +\ \times\ \times\ -$



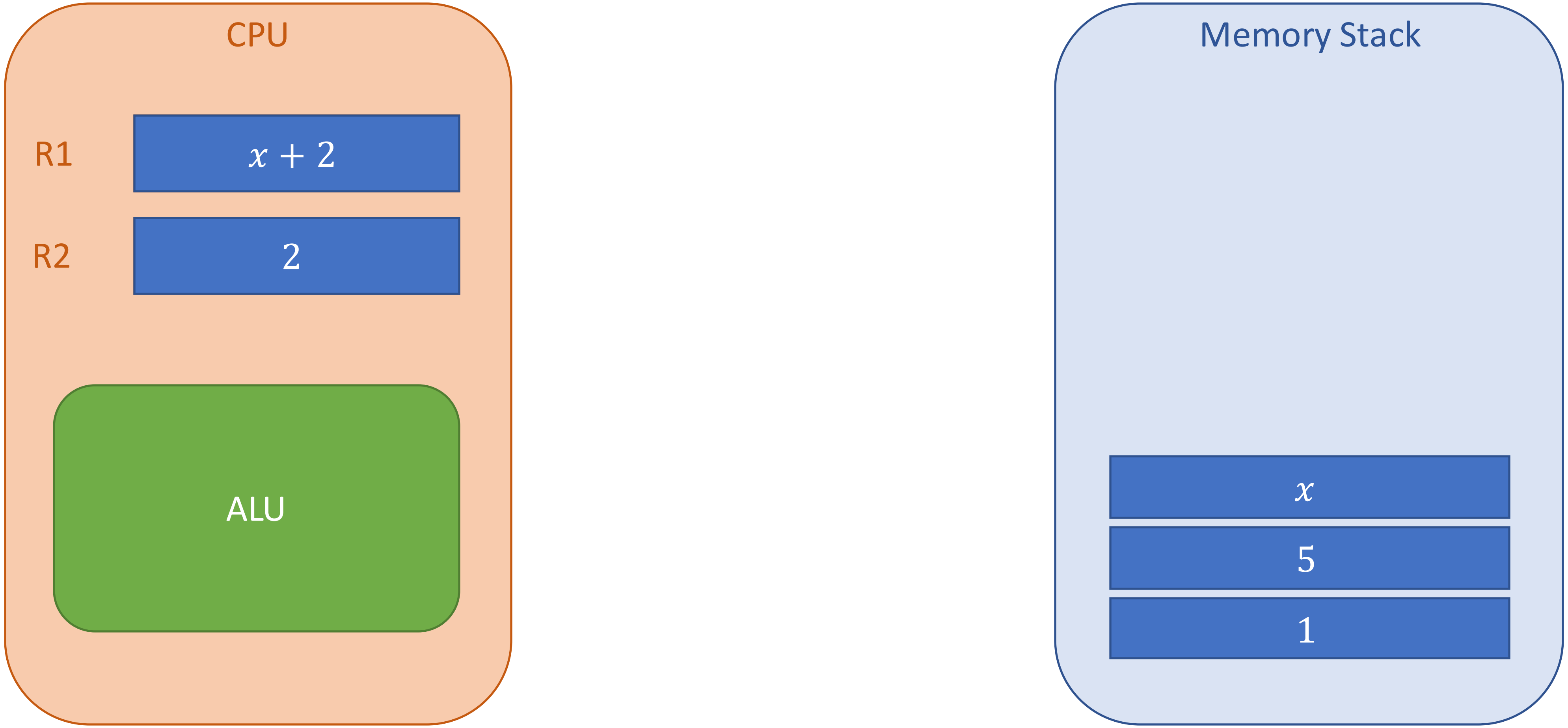
Example: Postfix evaluation with a stack

$$1\ 5\ x\ 2\ x\ +\ \times\ \times\ -$$



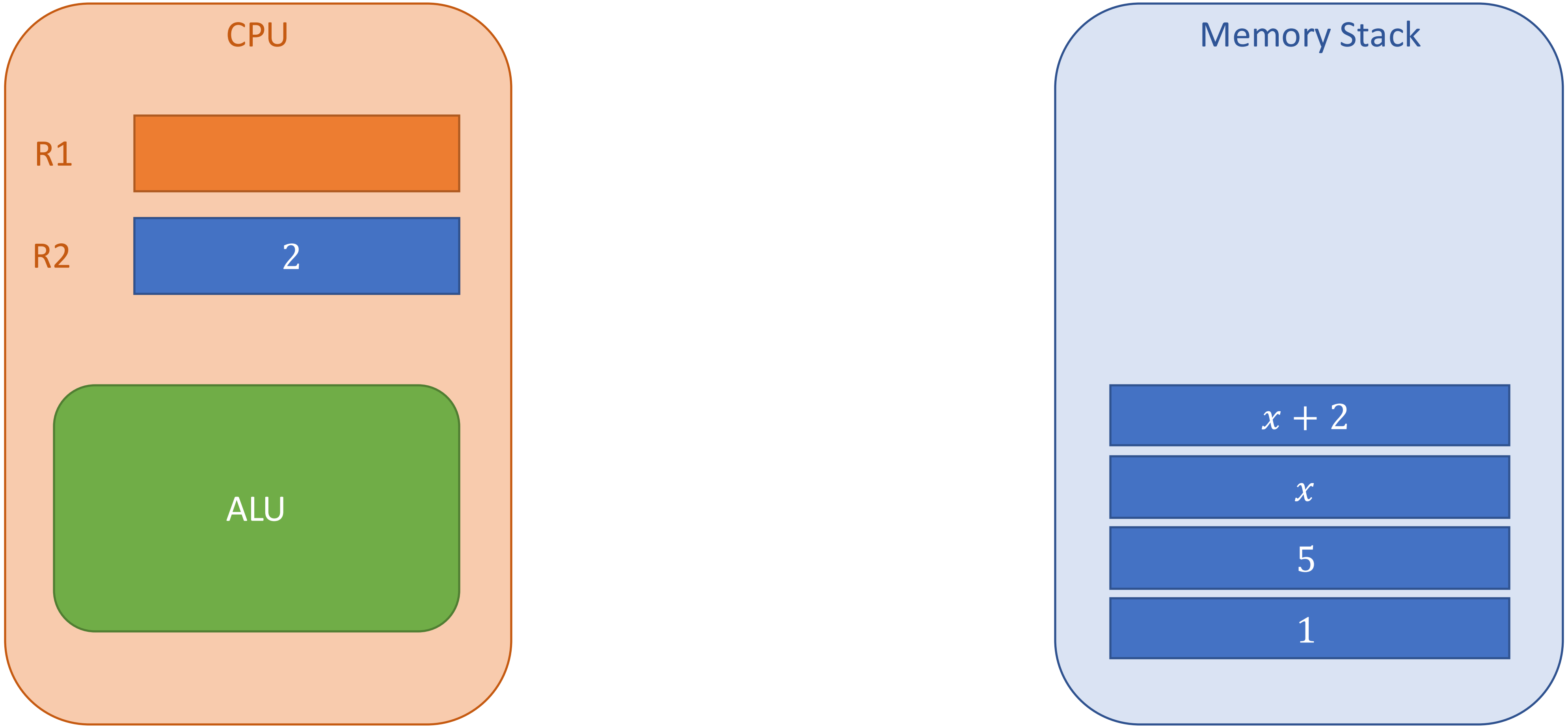
Example: Postfix evaluation with a stack

$$1\ 5\ x\ 2\ x + \times \times -$$



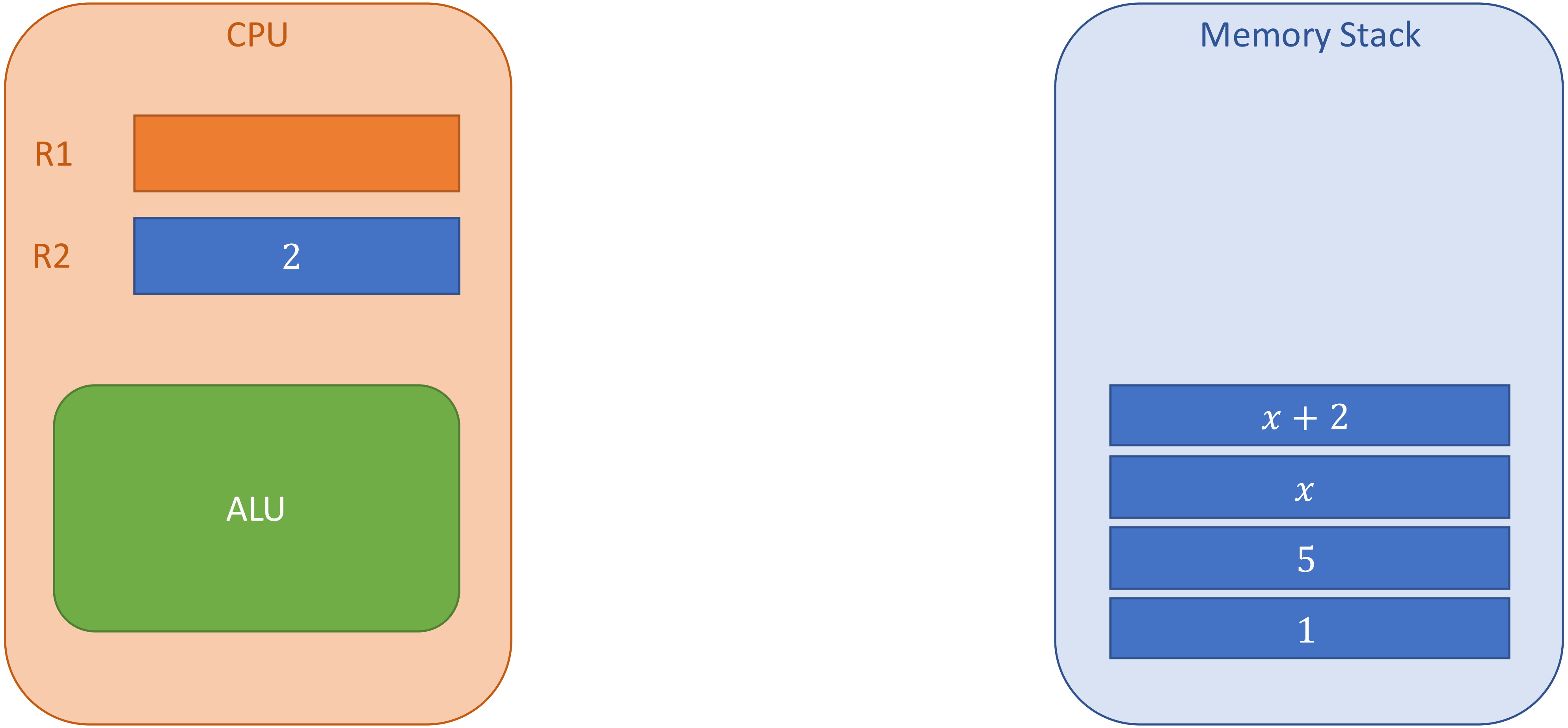
Example: Postfix evaluation with a stack

$1\ 5\ x\ 2\ x + \times \times -$



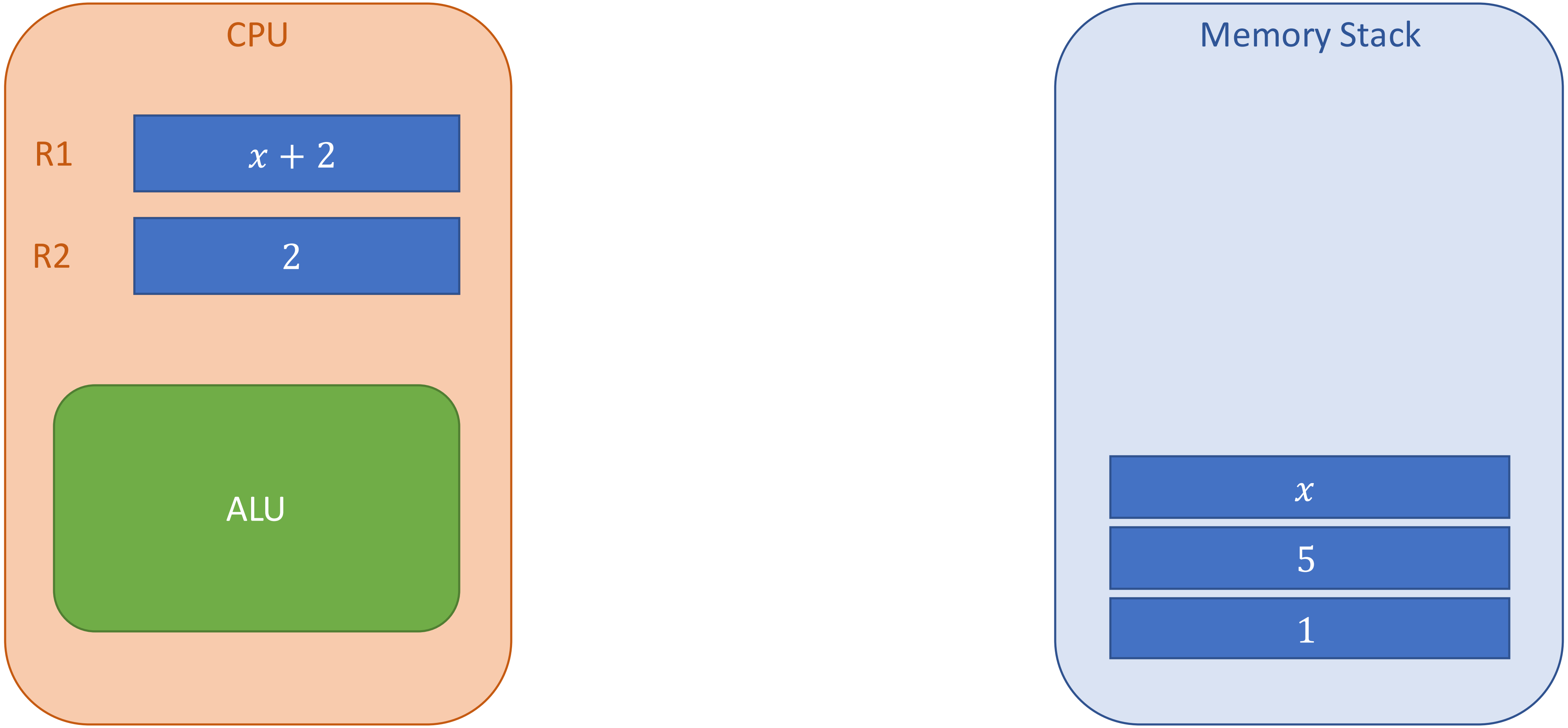
Example: Postfix evaluation with a stack

$1\ 5\ x\ 2\ x + \times \times -$



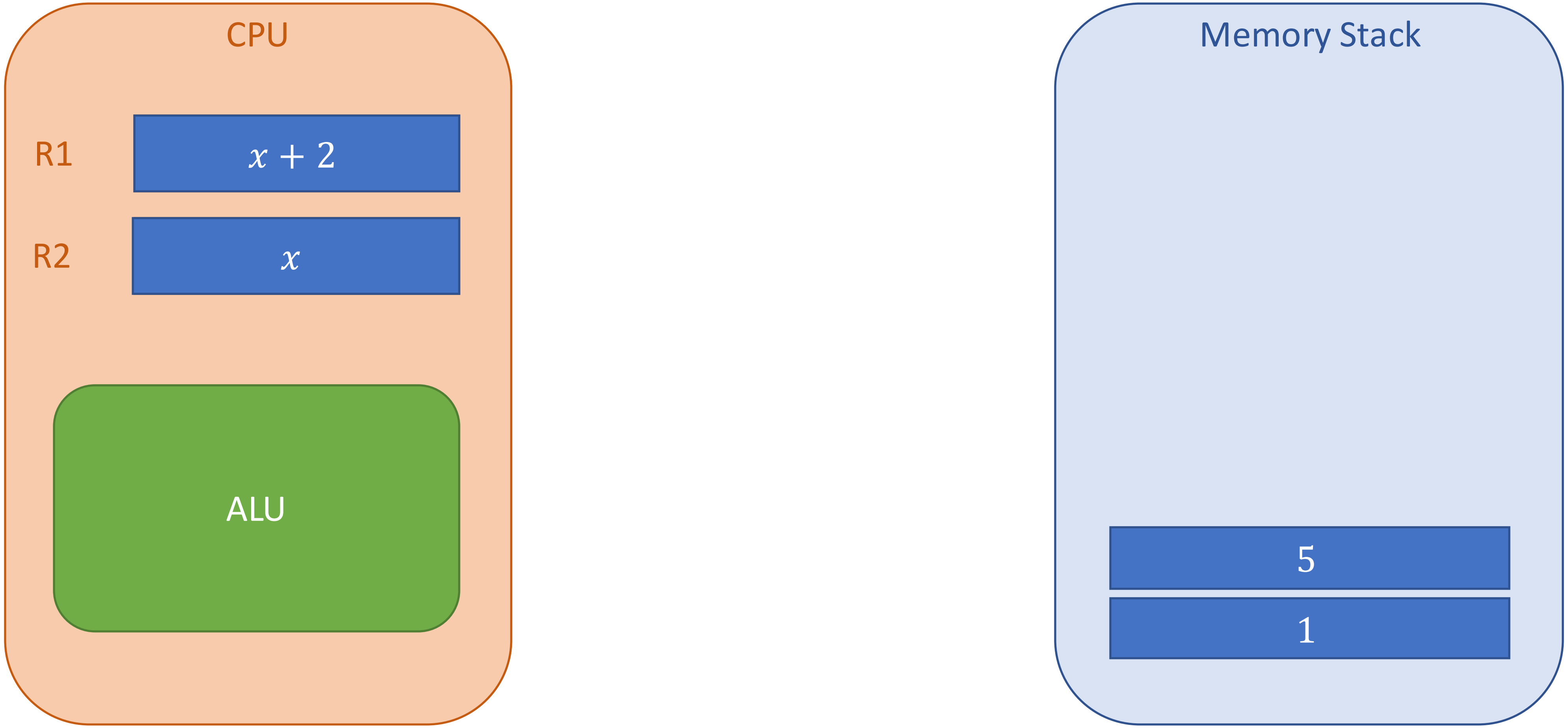
Example: Postfix evaluation with a stack

$1\ 5\ x\ 2\ x\ +\ \times\ \times\ -$



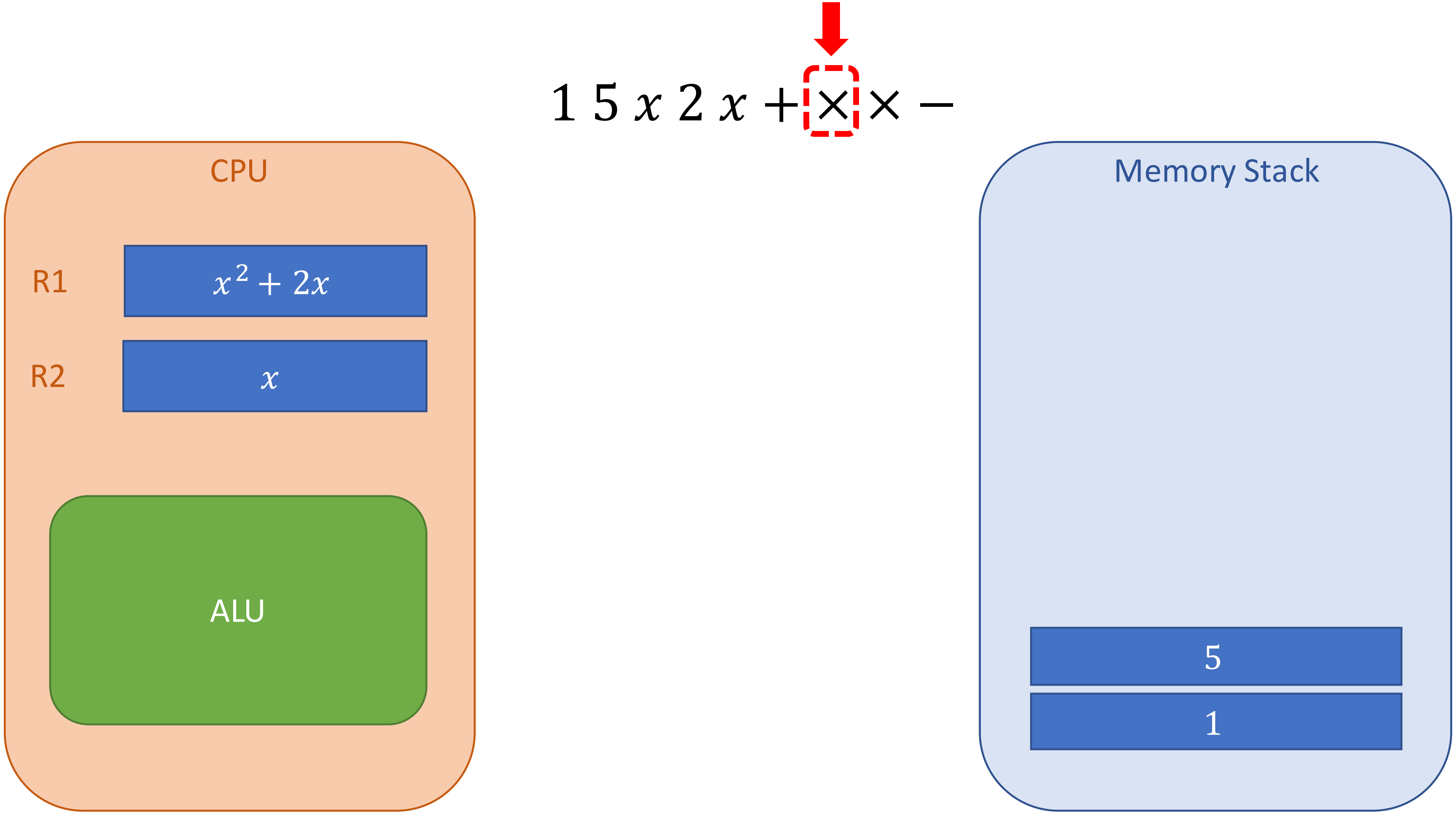
Example: Postfix evaluation with a stack

$$1\ 5\ x\ 2\ x + \times \times -$$



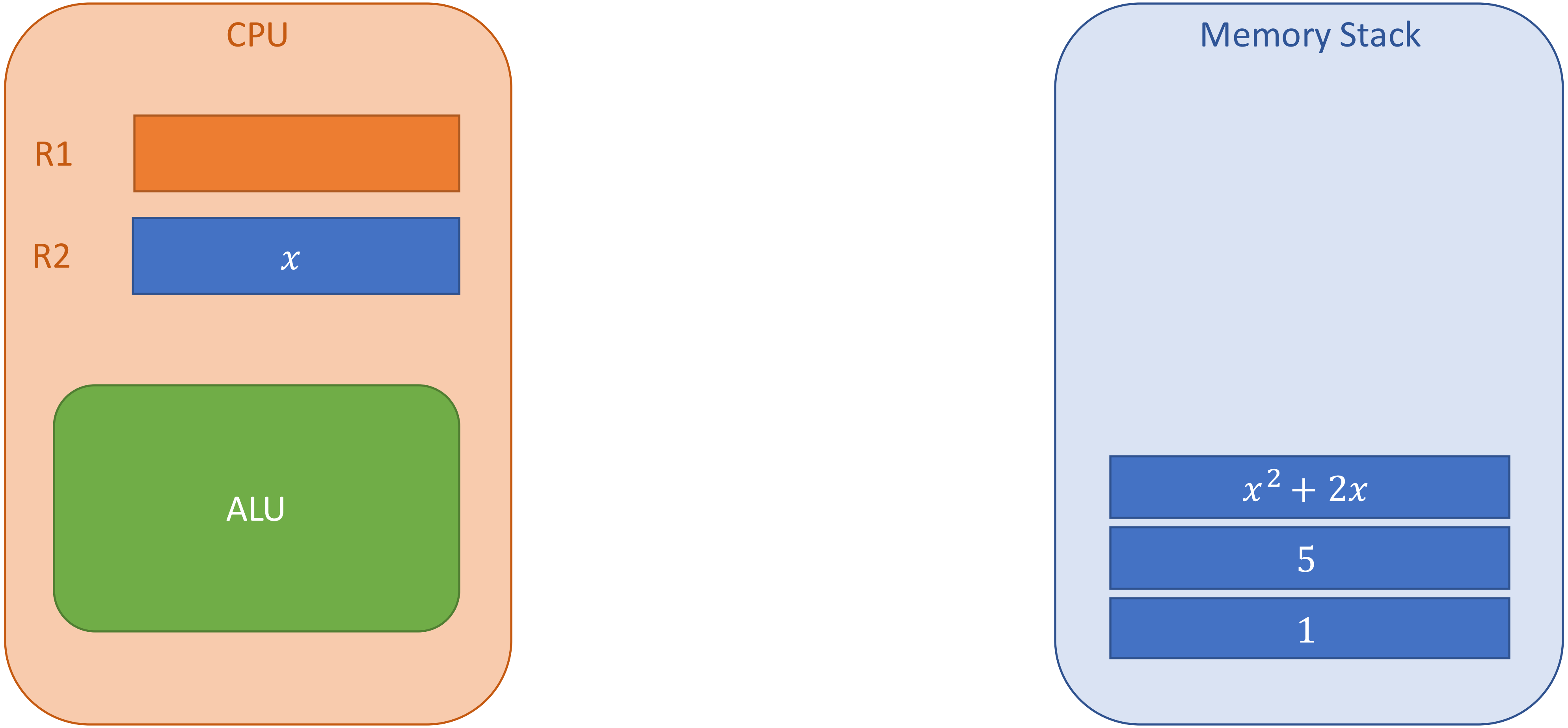
Example: Postfix evaluation with a stack

$1\ 5\ x\ 2\ x + \times \times -$



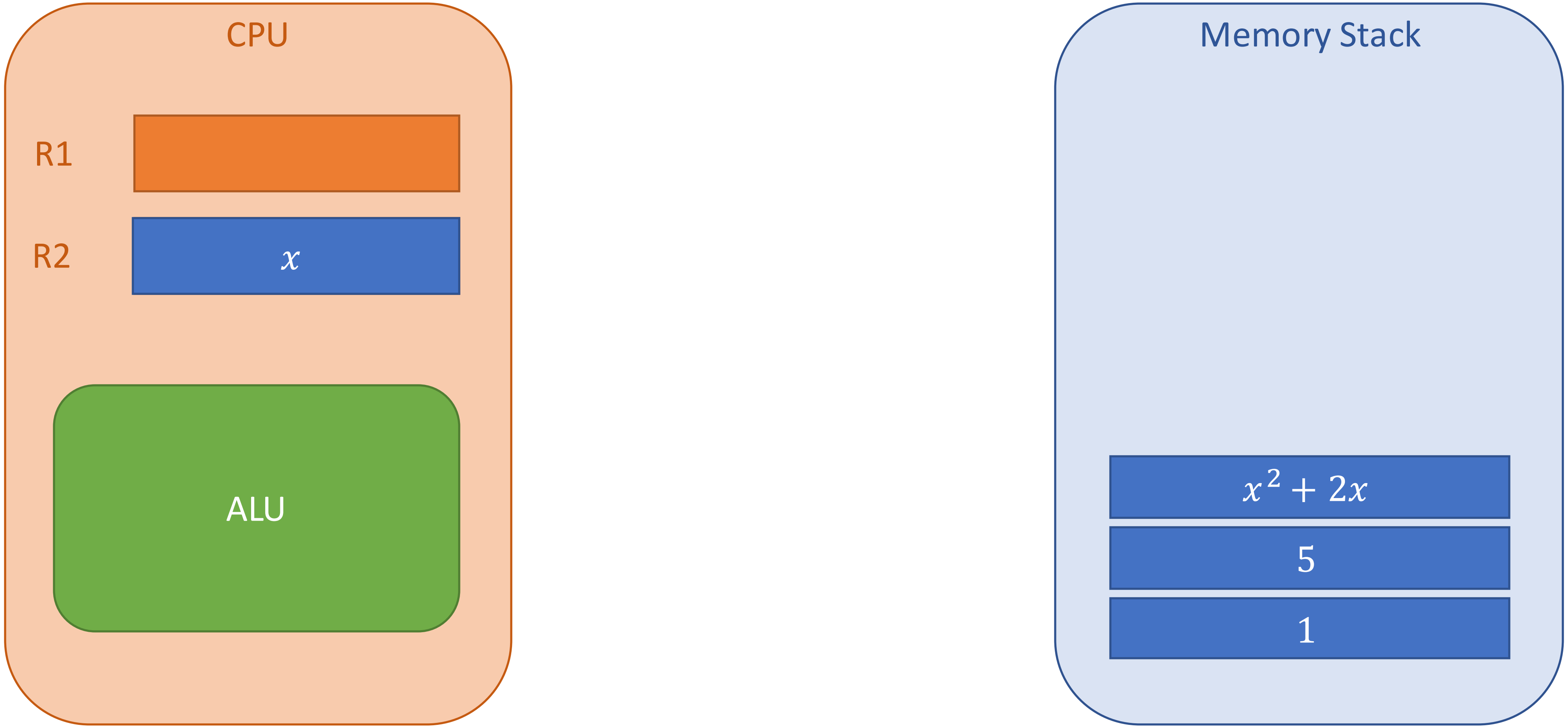
Example: Postfix evaluation with a stack

$$1\ 5\ x\ 2\ x + \times \times -$$



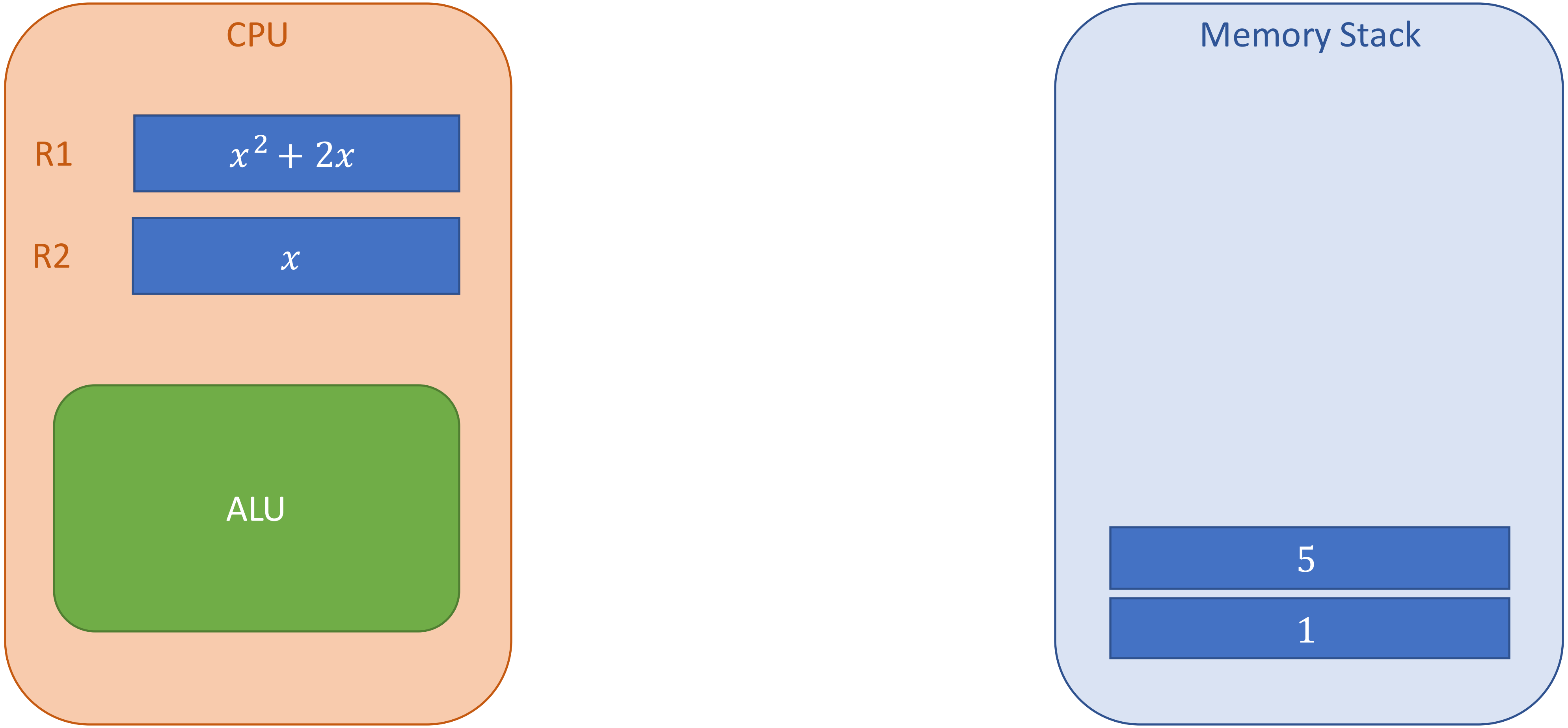
Example: Postfix evaluation with a stack

$$1\ 5\ x\ 2\ x + \times \times -$$



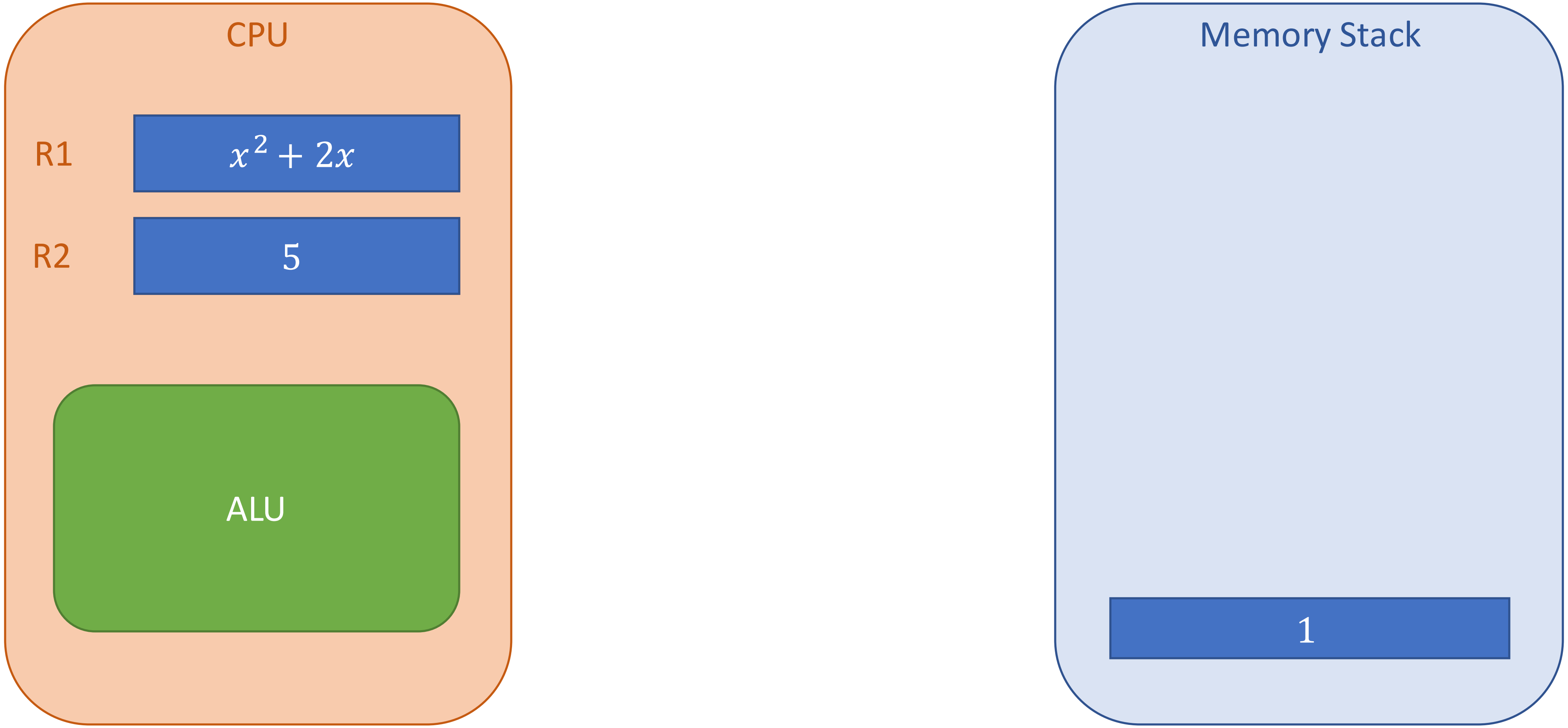
Example: Postfix evaluation with a stack

$$1\ 5\ x\ 2\ x + \times \times -$$



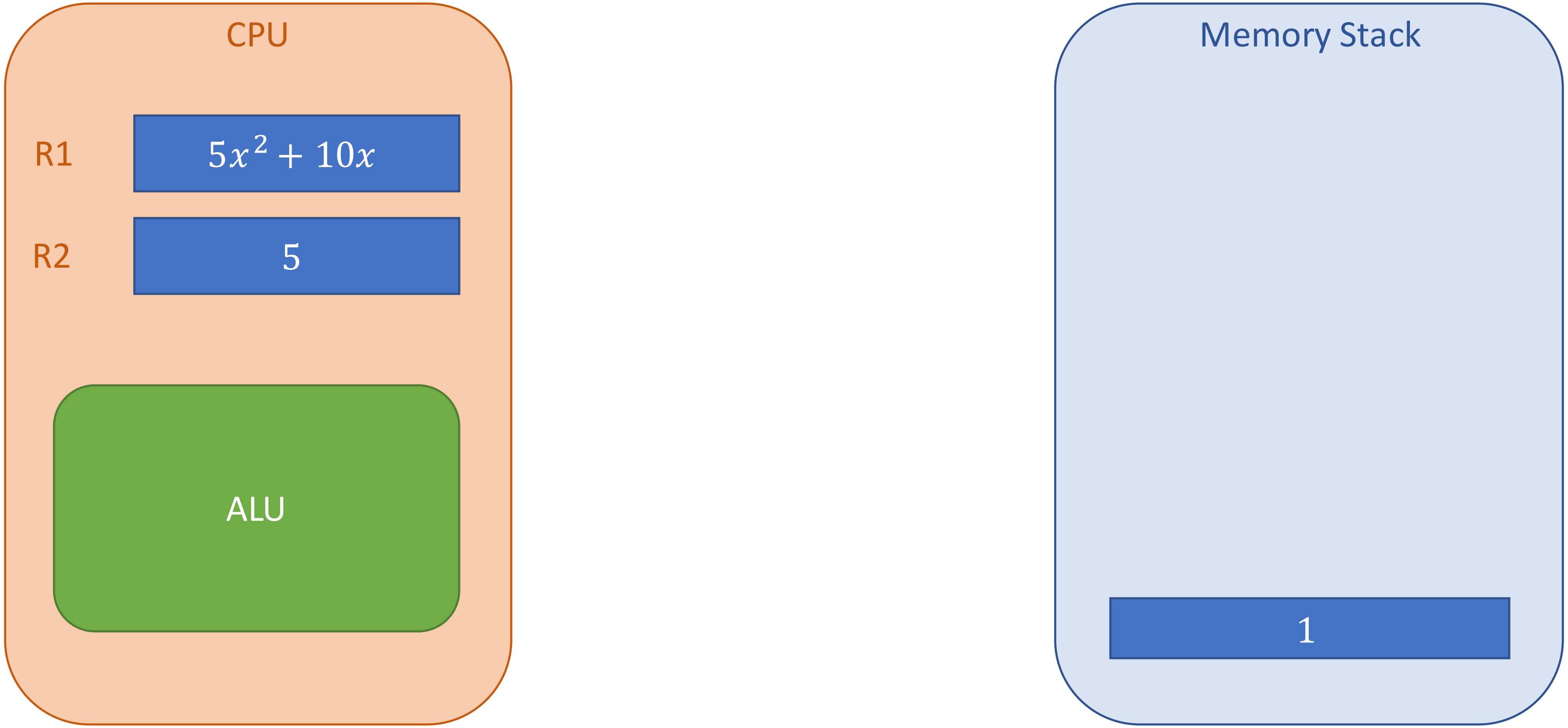
Example: Postfix evaluation with a stack

$$1\ 5\ x\ 2\ x + \times \times -$$

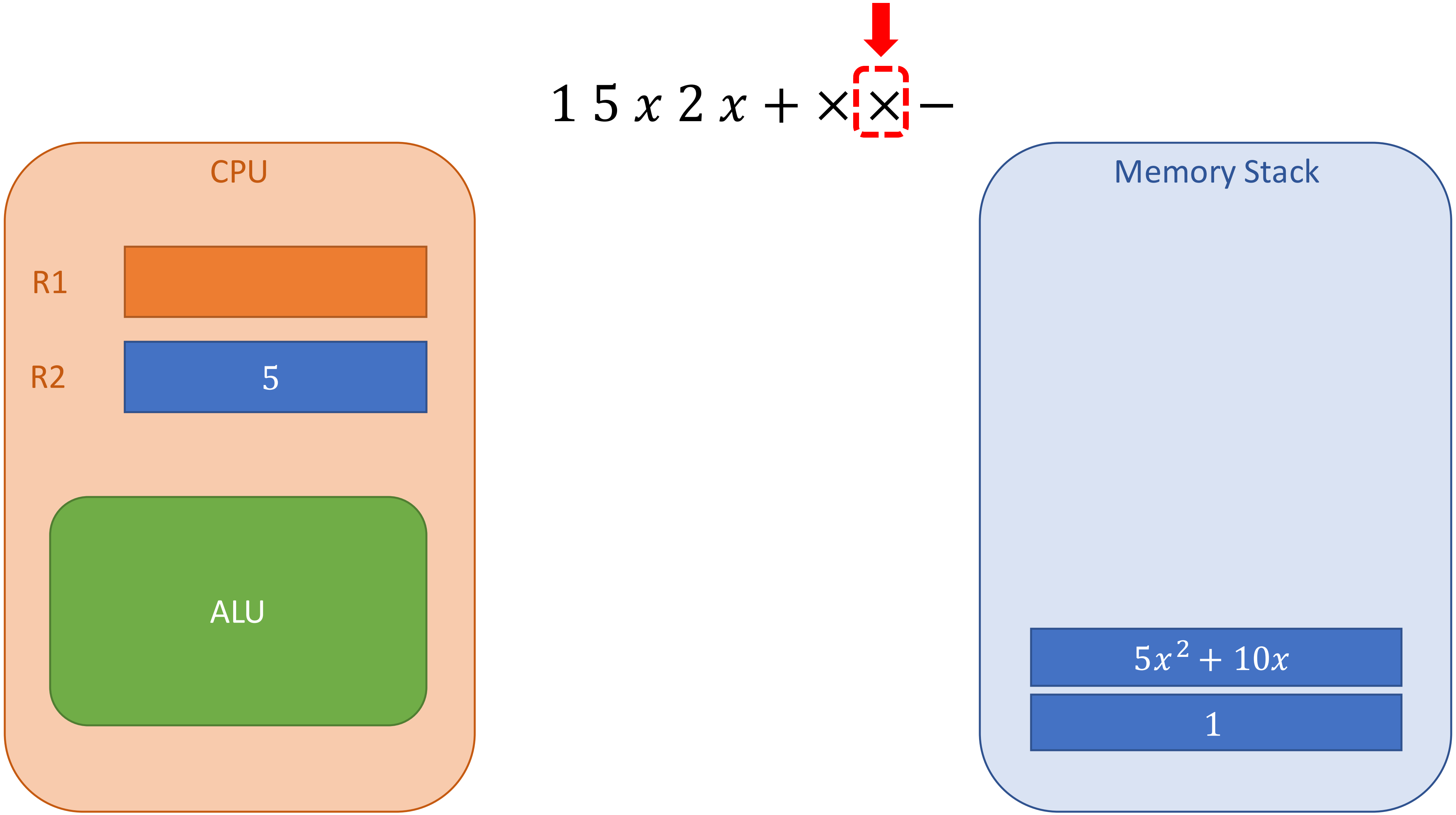


Example: Postfix evaluation with a stack

$$1\ 5\ x\ 2\ x + \times \times -$$

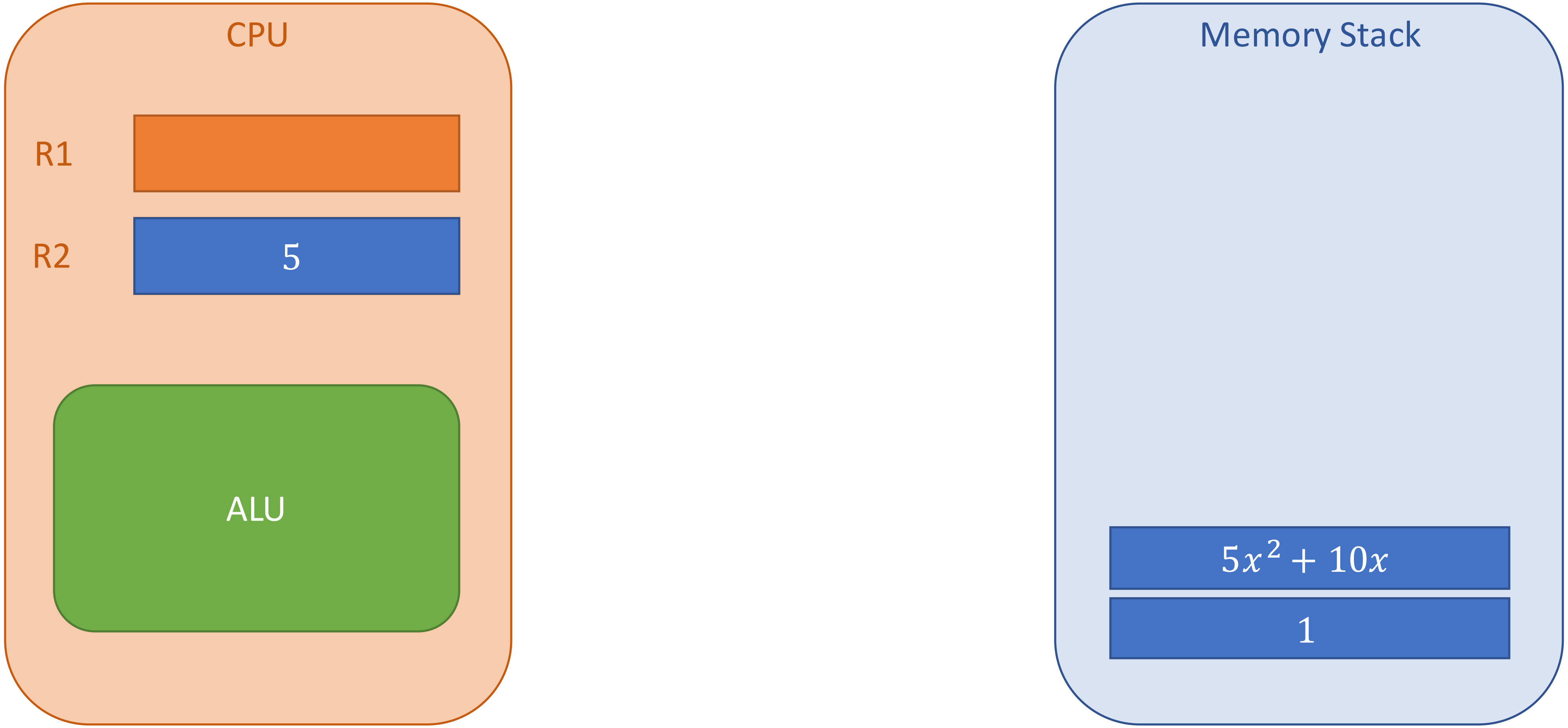


Example: Postfix evaluation with a stack



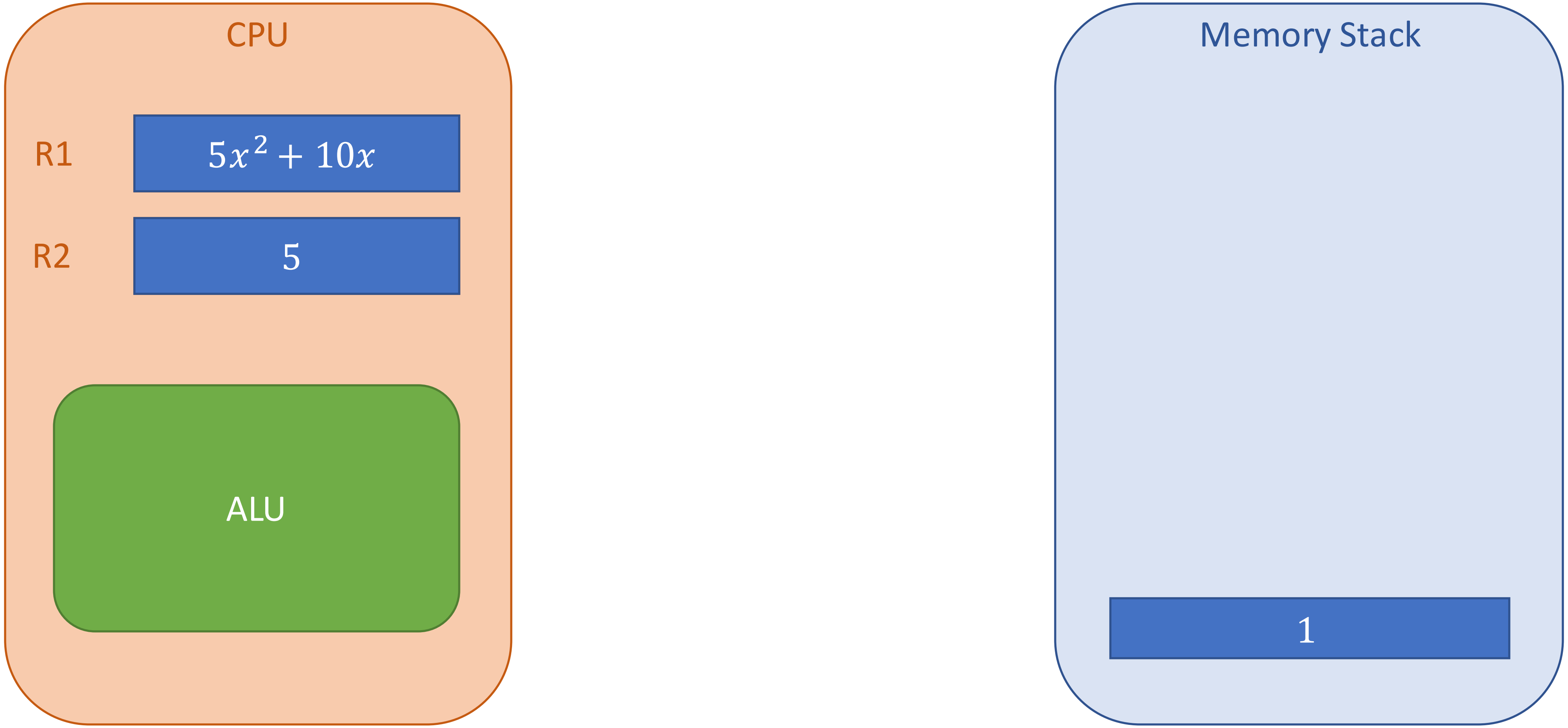
Example: Postfix evaluation with a stack

$$1\ 5\ x\ 2\ x + \times \times -$$



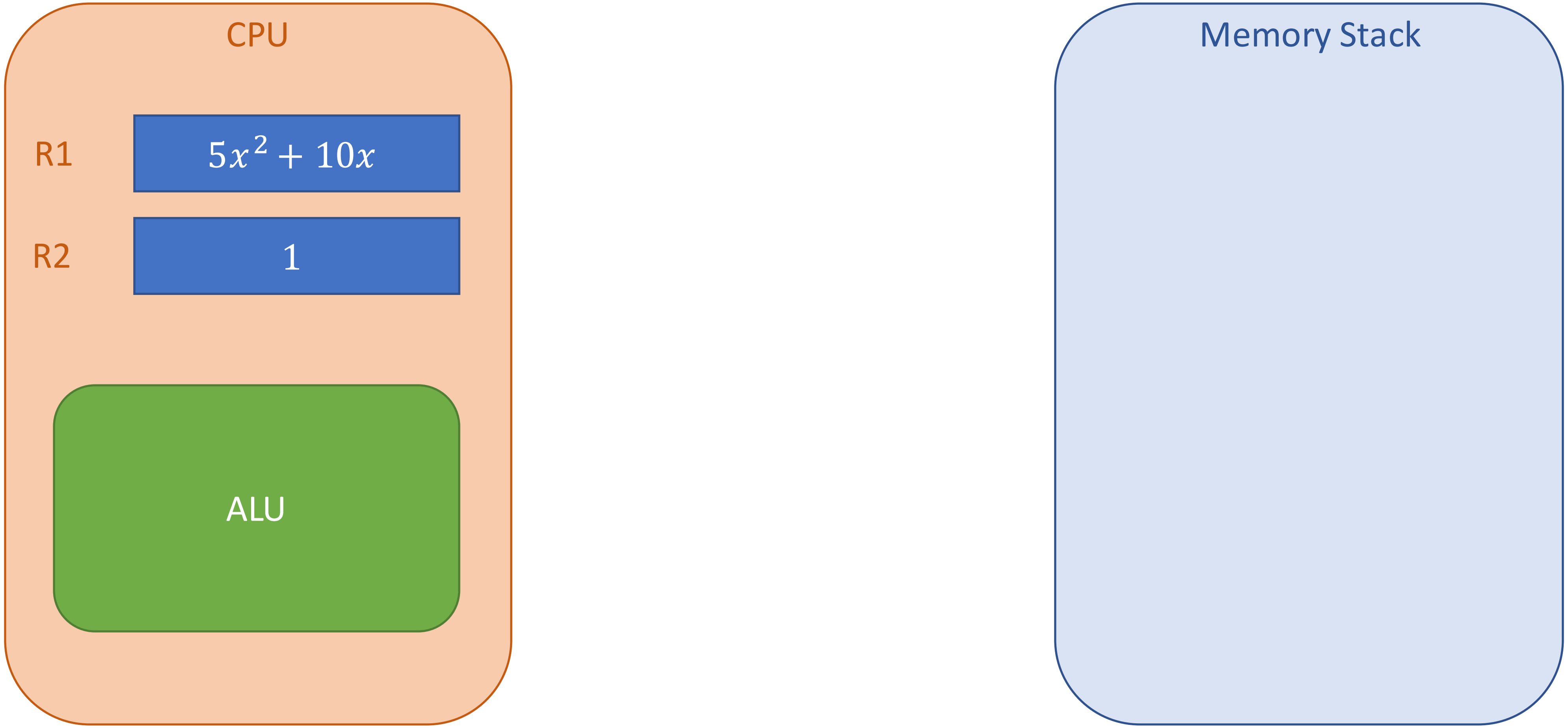
Example: Postfix evaluation with a stack

$$1\ 5\ x\ 2\ x\ +\ \times\ \times\ -$$



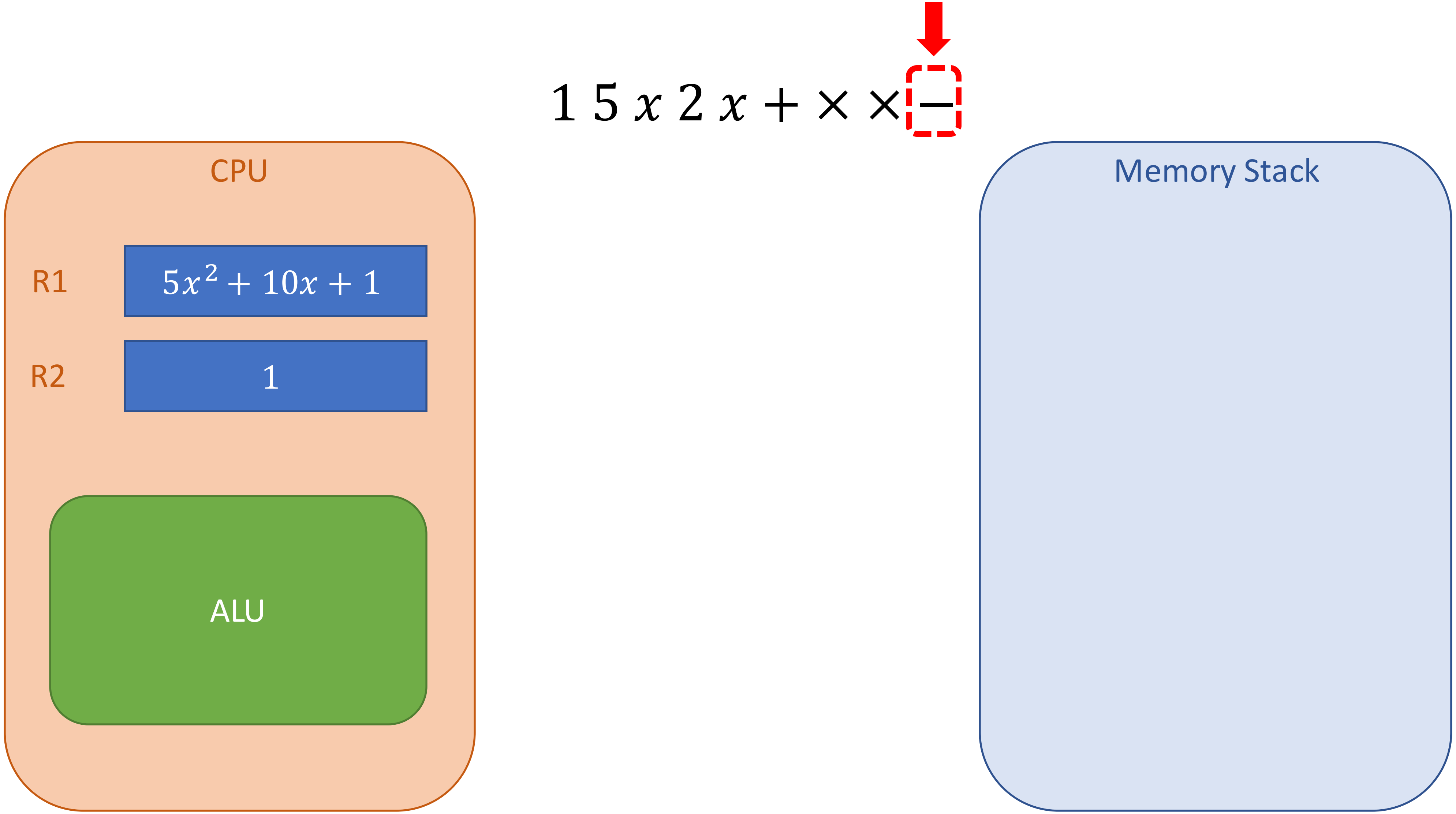
Example: Postfix evaluation with a stack

$$1\ 5\ x\ 2\ x + \times \times -$$



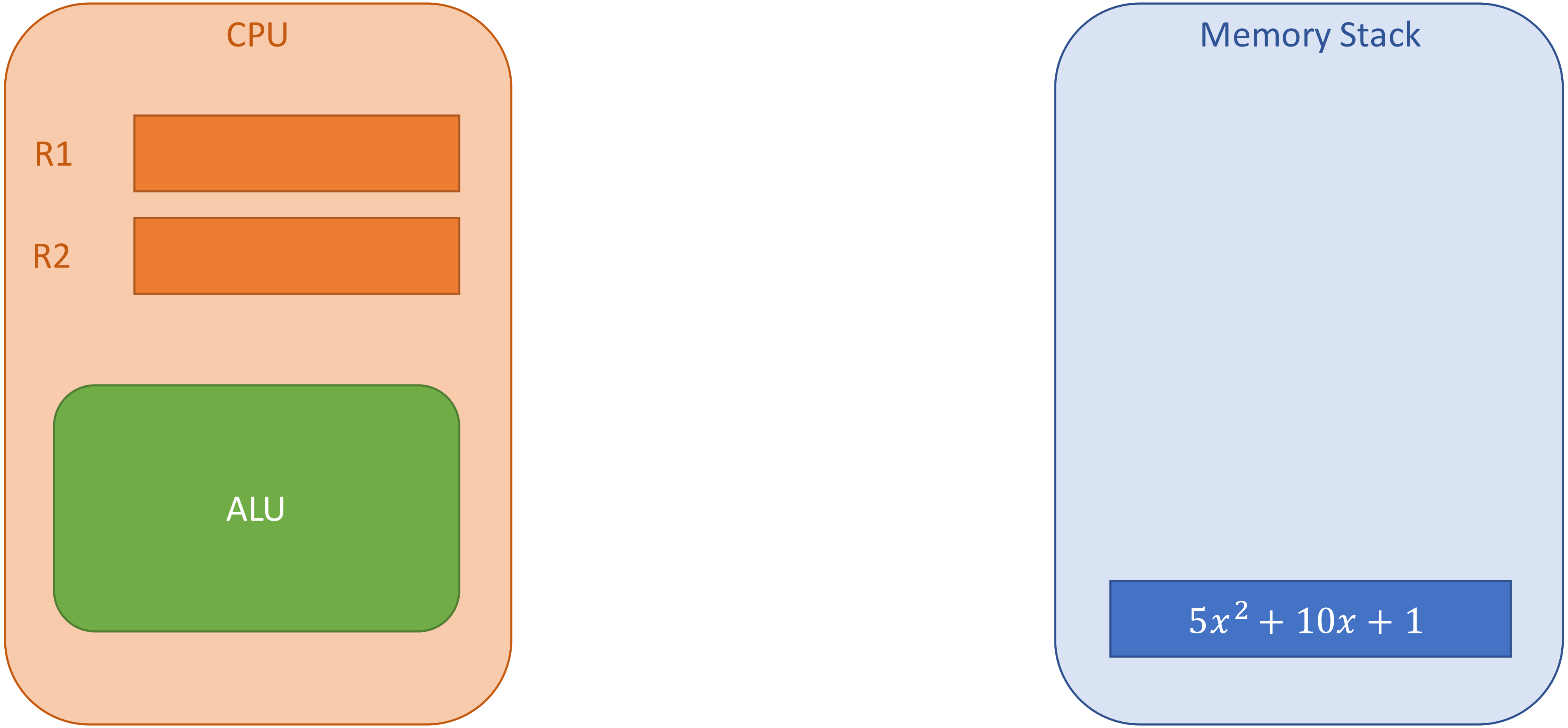
Example: Postfix evaluation with a stack

$$1\ 5\ x\ 2\ x + \times \times -$$



Example: Postfix evaluation with a stack

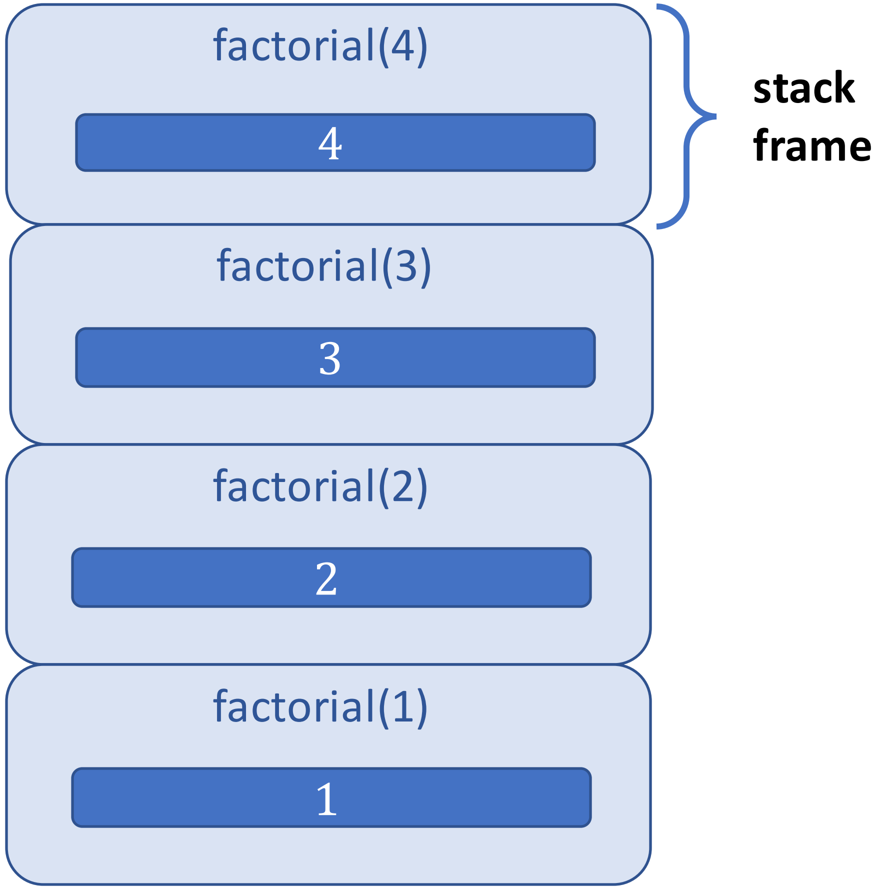
$$1\ 5\ x\ 2\ x + \times \times -$$



The Stack

$$\text{factorial}(n) = n \times \text{factorial}(n-1)$$
$$\text{factorial}(1) = 1$$

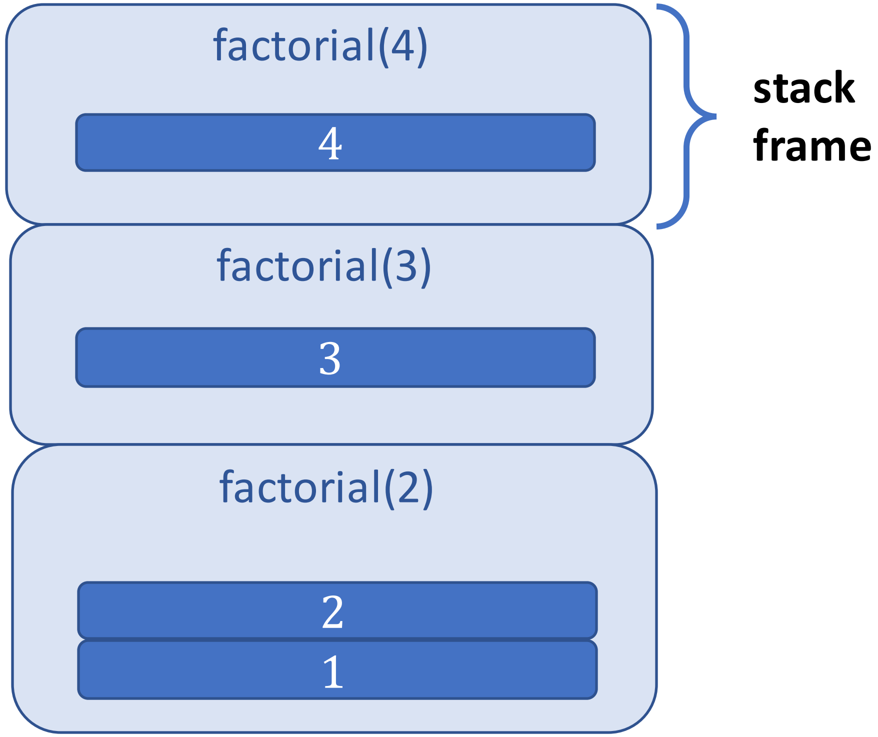
- Stack is used as local memory for a function call
- A **stack frame** is created for each function call



The Stack

$$\text{factorial}(n) = n \times \text{factorial}(n-1)$$
$$\text{factorial}(1) = 1$$

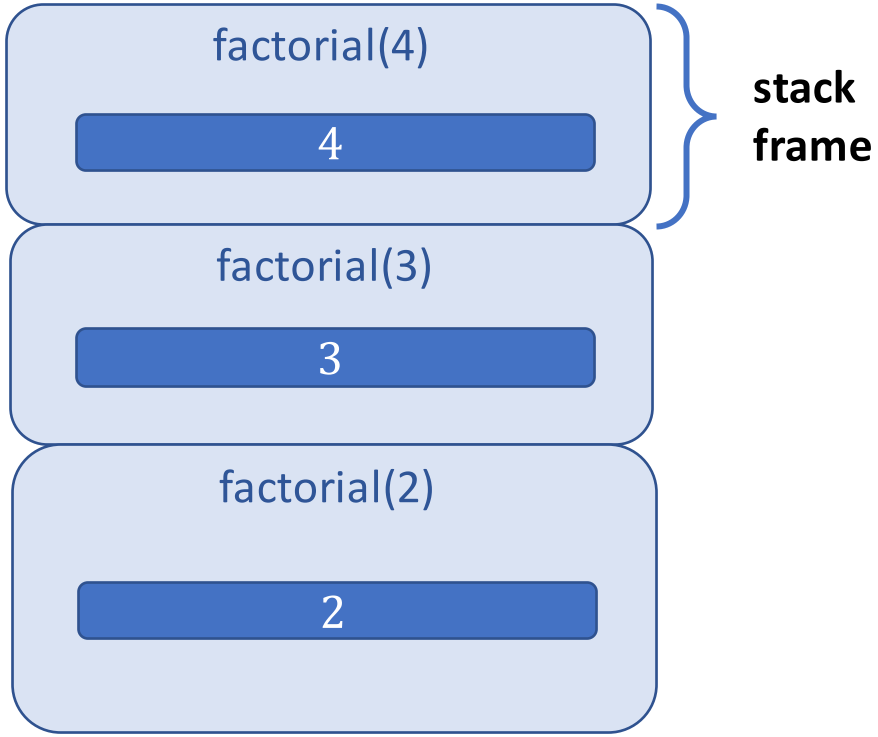
- Stack is used as local memory for a function call
- A **stack frame** is created for each function call



The Stack

$$\text{factorial}(n) = n \times \text{factorial}(n-1)$$
$$\text{factorial}(1) = 1$$

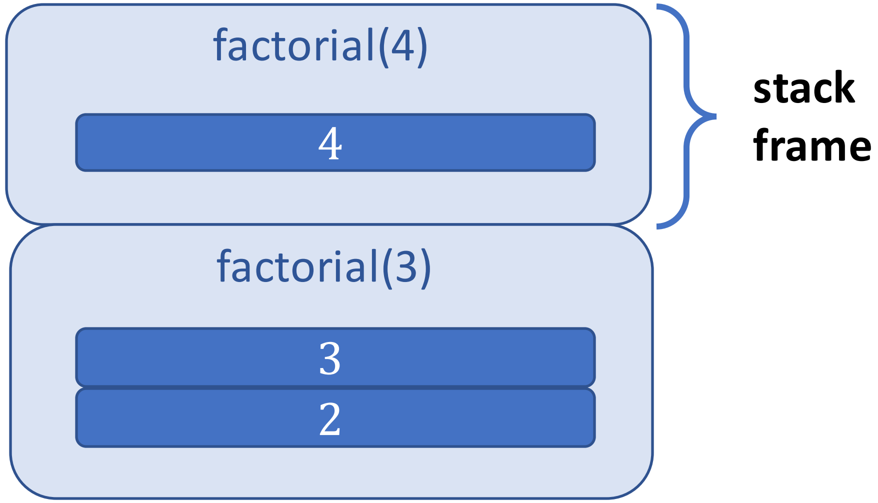
- Stack is used as local memory for a function call
- A **stack frame** is created for each function call



The Stack

$$\text{factorial}(n) = n \times \text{factorial}(n-1)$$
$$\text{factorial}(1) = 1$$

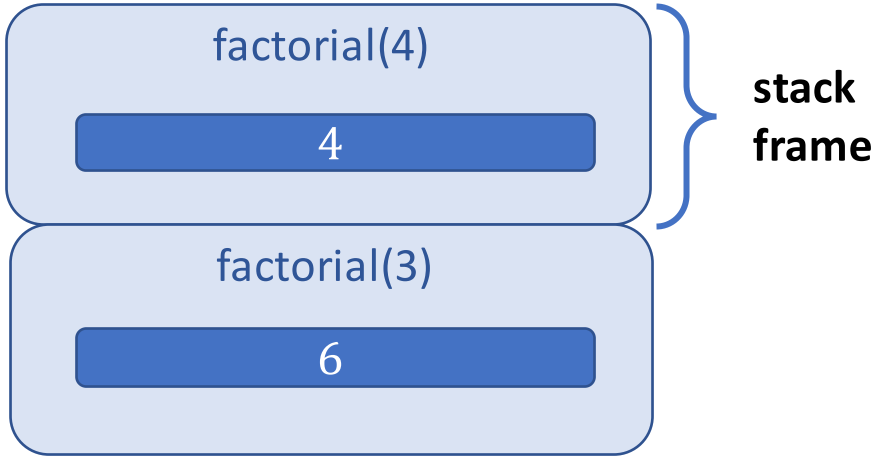
- Stack is used as local memory for a function call
- A **stack frame** is created for each function call



The Stack

$$\text{factorial}(n) = n \times \text{factorial}(n-1)$$
$$\text{factorial}(1) = 1$$

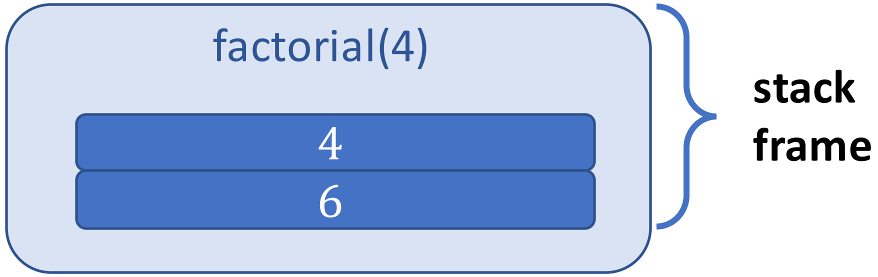
- Stack is used as local memory for a function call
- A **stack frame** is created for each function call



The Stack

$$\text{factorial}(n) = n \times \text{factorial}(n-1)$$
$$\text{factorial}(1) = 1$$

- Stack is used as local memory for a function call
- A **stack frame** is created for each function call



The Stack

$$\text{factorial}(n) = n \times \text{factorial}(n-1)$$
$$\text{factorial}(1) = 1$$

- Stack is used as local memory for a function call
- A **stack frame** is created for each function call

The Stack

$$\text{factorial}(n) = n \times \text{factorial}(n-1)$$
$$\text{factorial}(1) = 1$$

- Stack is used as local memory for a function call
- A **stack frame** is created for each function call
- Eventually, the result of the function will replace the function call

24

The Stack

Advantages:

- Stack memory that is no longer used can be safely **overwritten**

Limitations:

- Size of variables stored on the stack **must be known** at **compile time**
- Physical addresses and offsets are stored in the **read-only** instructions
- Size **cannot** depend on input arguments
- Stack has a **maximum** size

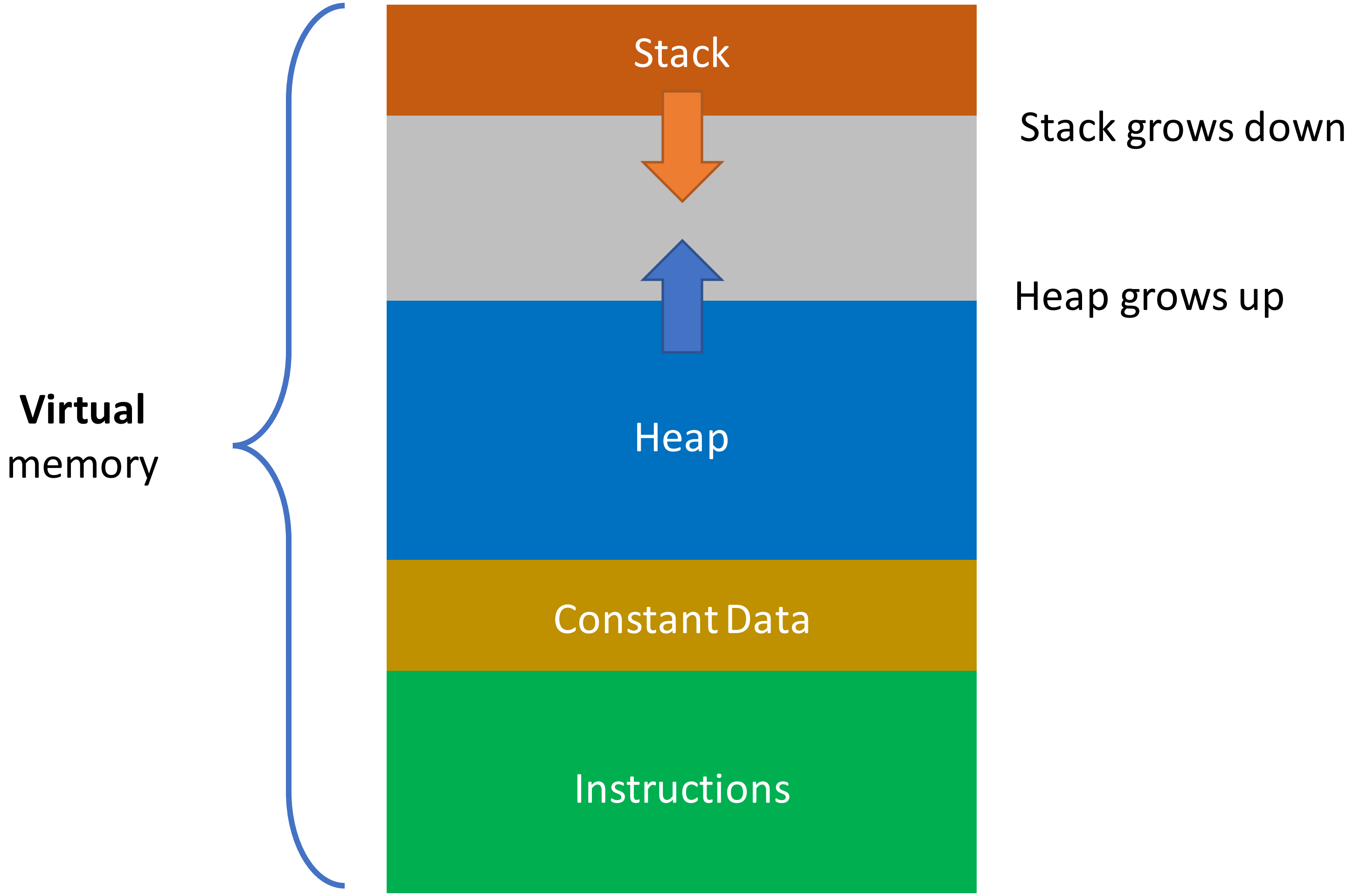
The Heap

- A large section of memory to store objects of an **arbitrary** size
- Objects whose sizes are not known at compile time are stored on the **heap** (e.g. **arrays**)
- Need some mechanism to manage the available memory:
  - Find memory with enough space that isn't being used currently
  - Keep track of all memory that is currently being used
  - Free up memory when it is no longer needed
- Many modern programming languages use a **Garbage Collector** (**GC**) which cleans up memory once it is no longer being used

The Ten Million Room Hotel - ***fasterthanlime***

https://youtu.be/553luW-0eZw?t=331

Process Memory



Workshop – Thursday 19/01/2023

Assignment

https://classroom.github.com/a/3bYk2x83

Tasks:

- Clone your repository
- Read through the README for assignment details
- Ask if you have any questions