

COMS12200

Introduction to

Computer Architecture

Dr. Cian O'Donnell

cian.odonnell@bristol.ac.uk

Topic 4: Processor control flow

Topics

1. Data, Control and Instructions
2. Memory
3. Execution cycle
4. Processor control flow
5. State machines and decoding
6. Machine types
7. Memory paradigms

Topics

1. Data, Control and Instructions
2. Memory
3. Execution cycle
4. Processor control flow
5. State machines and decoding
6. Machine types
7. Memory paradigms

Processor control flow

Overview

- Program counter under normal and conditional control flow.
- The mechanics of branching.
- The function of the flag register.

Q: So far, we have a processor, memory and a PC.
We could use this to write programs that execute several instructions in sequence.

What is missing to produce a useful system?


Anyone?

A: Some of the features that make modern architectures so versatile is their ability to:

1. React to changes in data values
2. Support modular code

Control vs data (recap)

- You can think of a processor's function as being dictated by two separate influences:



Control information
(tells it what to do)

Data information
(operated on to get result)

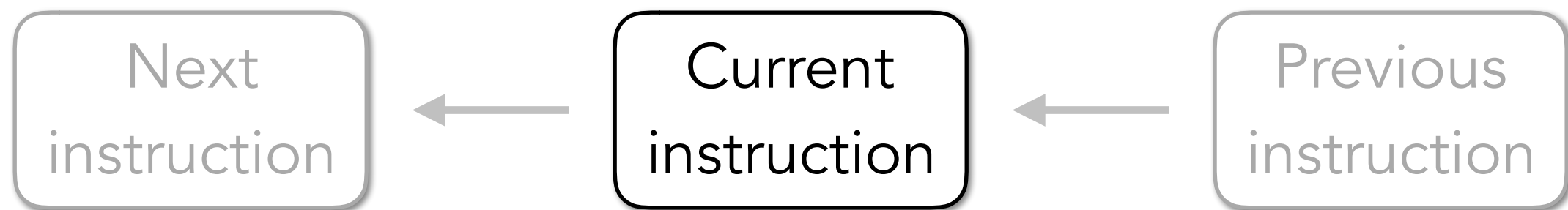
- These two influences form two paths into the processor logic

Processor control flow (1)

Controlling program execution

Execution

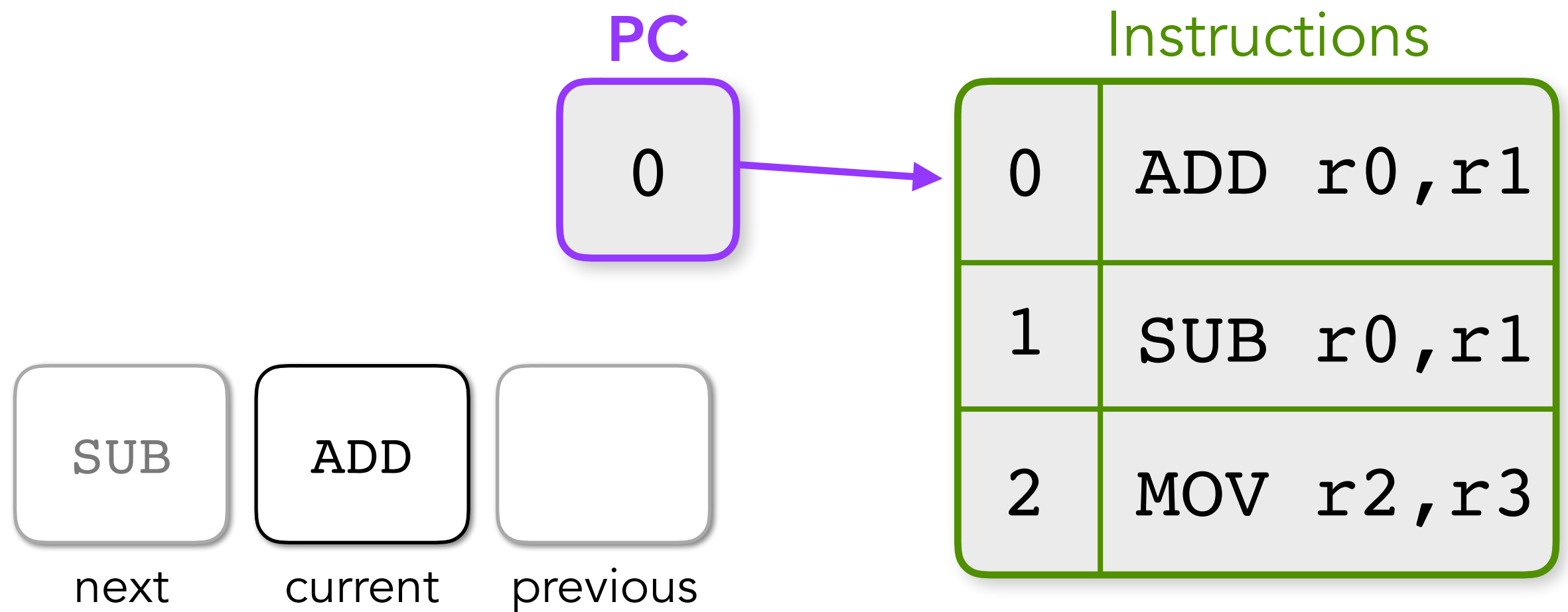
- We can model a series of steps of execution as a **sequence of instructions**.



- Modelling the sequence is useful for seeing instruction effects and data dependencies.

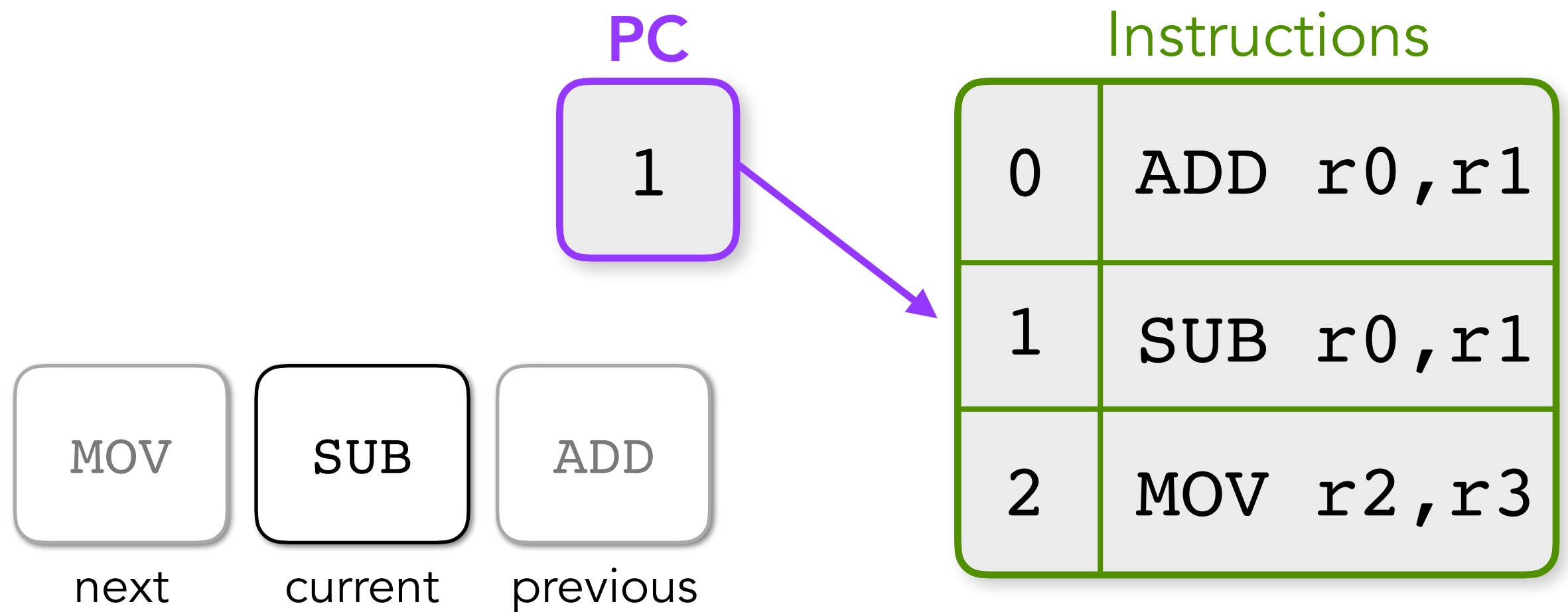
Normal PC Operation

Under normal program flow, the PC increments by one instruction address per instruction fetched.



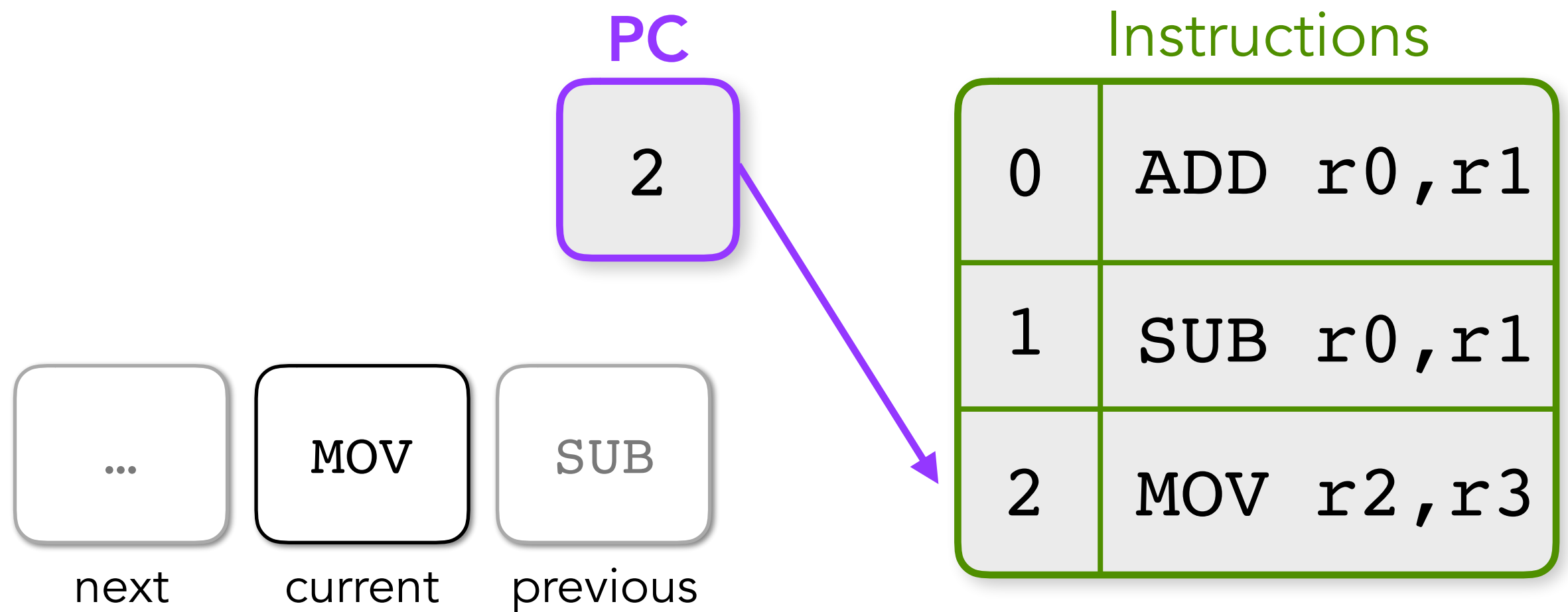
Normal PC Operation

Under normal program flow, the PC increments by one instruction address per instruction fetched.

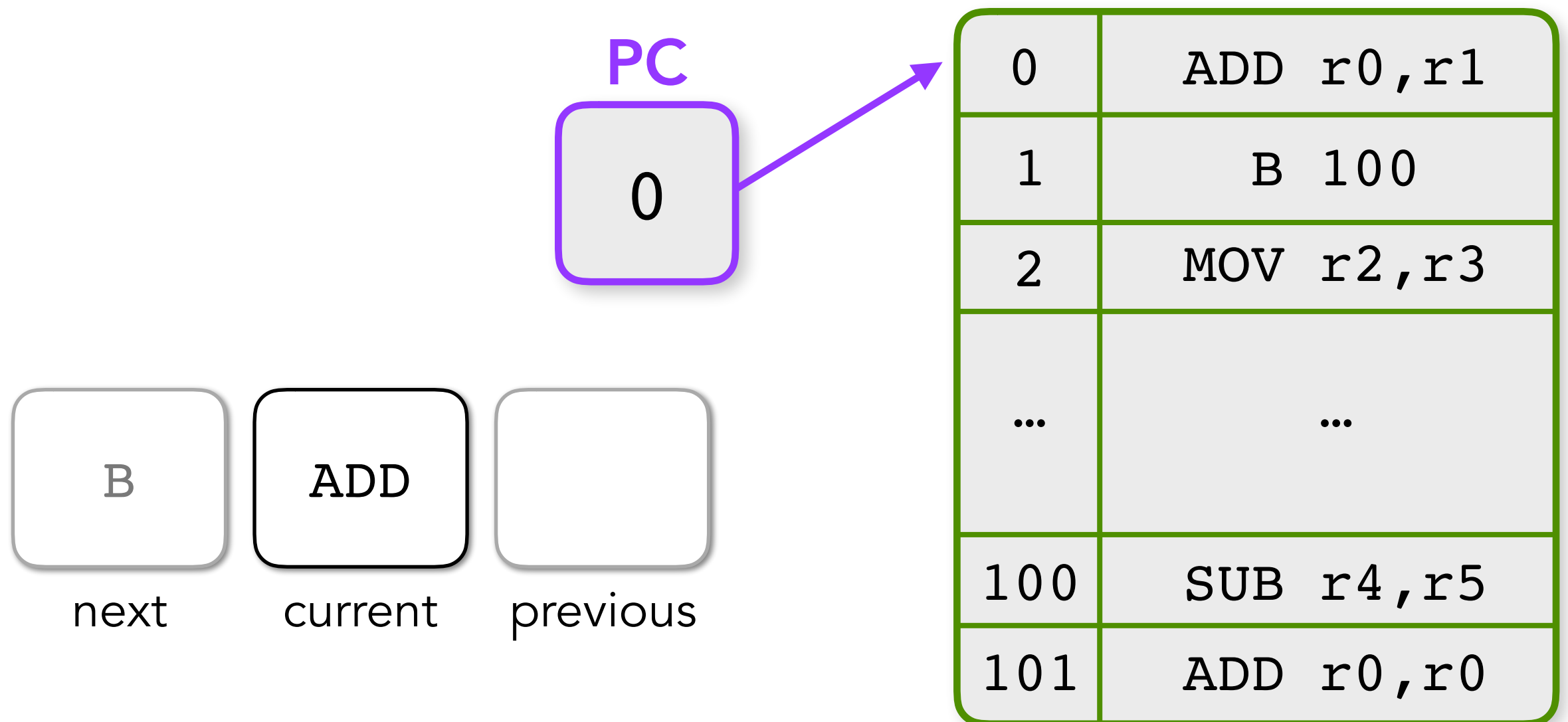


Normal PC Operation

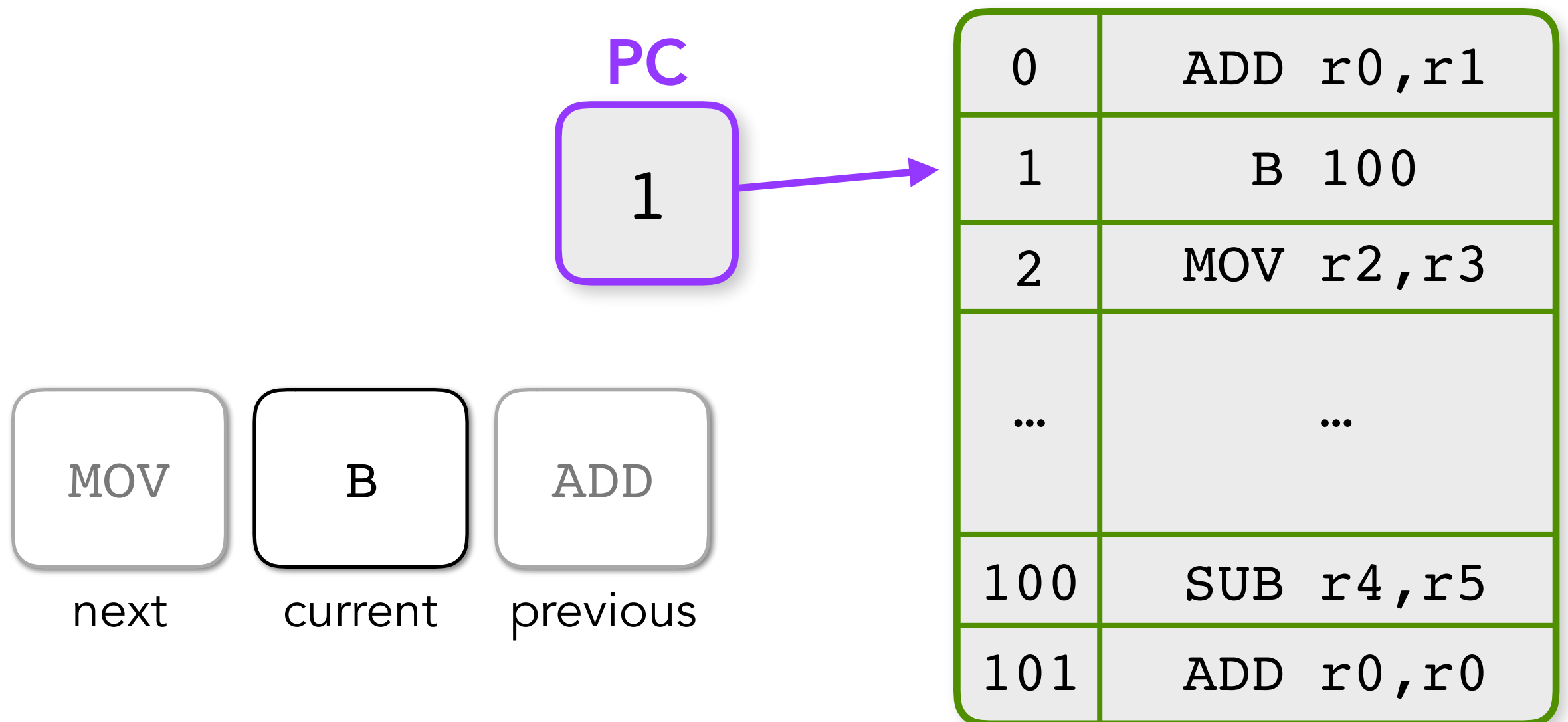
Under normal program flow, the PC increments by one instruction address per instruction fetched.



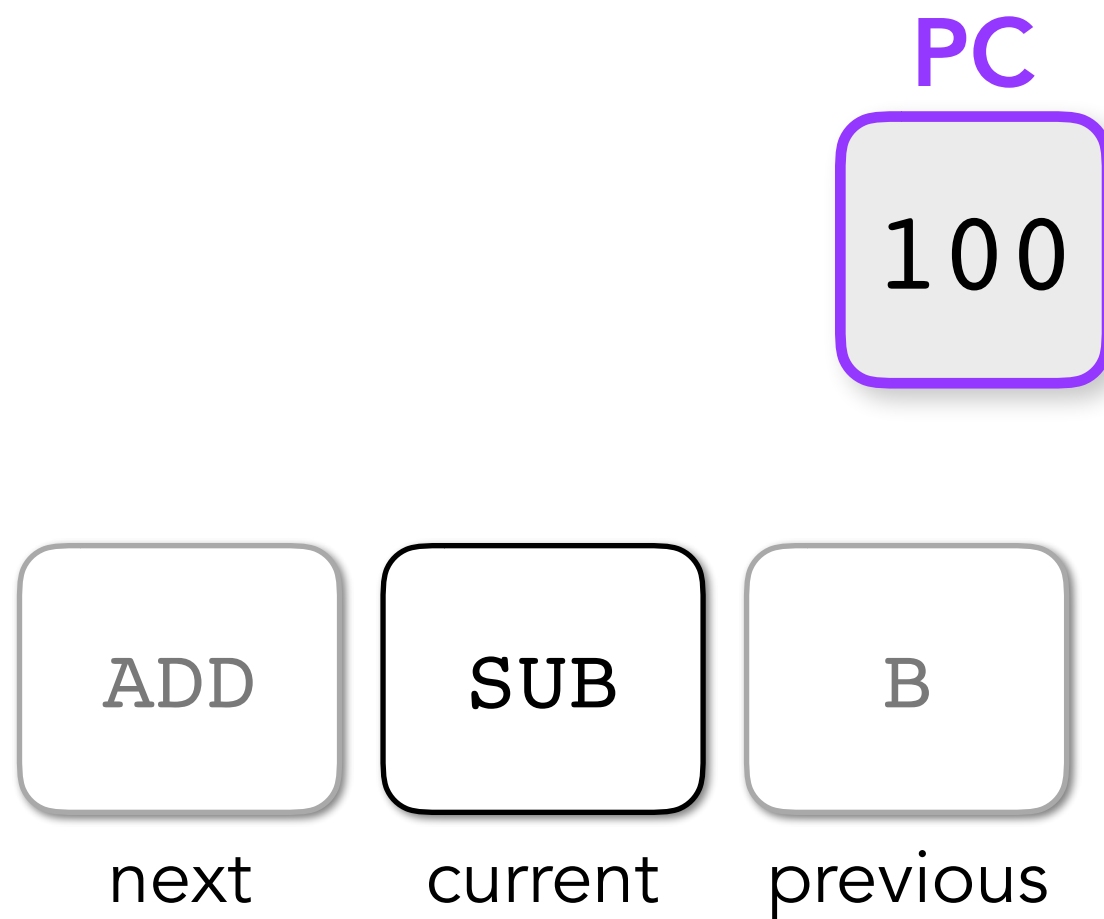
Taking a branch



Taking a branch

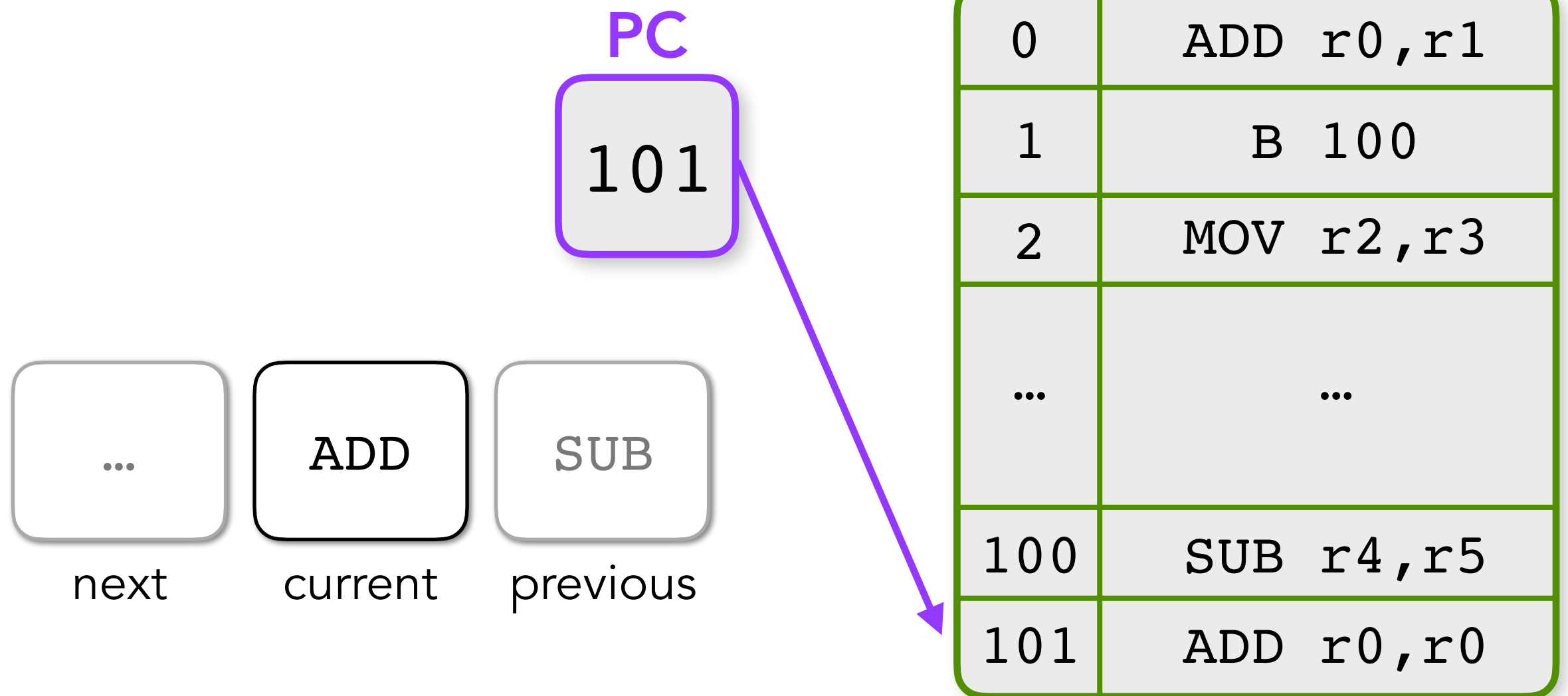


Taking a branch



| | |
|-----|-----------|
| 0 | ADD r0,r1 |
| 1 | B 100 |
| 2 | MOV r2,r3 |
| ... | ... |
| 100 | SUB r4,r5 |
| 101 | ADD r0,r0 |

Taking a branch



Conditional control flow

- We can make the control flow **conditional on data values**: “if *this* is true, then do *that*”
- These values are normally data previously seen by the processor.
- Can also be based on operands to the conditional instructions.

Conditional branching

- Most often, we make branches the target of conditional execution.
- This allows us to re-direct the flow of a program at run-time.
- It introduces **non-determinism** in the program.
 - PRO: Increases the programmer's power.
 - CON: Explodes the complexity of analysis.

Conditional example

- C code: `If (a == 0) then a = 1 else a = 2`
- Accumulator machine code

| If | then | else |
|------------|---------|---------|
| LOAD a | MOVE 1 | MOVE 2 |
| COMPARE 0 | STORE a | STORE a |
| BIFEQ then | B end | B end |
| BIFNE else | | |

Condition codes

How do the tests for the branches work?

Q: Where does the information come from?
(An argument was not supplied to the instruction.)

Answer: The information comes from the condition code or flag register.

The flag register

It's architecture specific, but normally contains most of the following:

| Abbreviation | Meaning |
|--------------|--------------------|
| EQ | Equal |
| Z | Zero |
| N | Negative (or sign) |
| C | Carry out |

Processor control flow (part 1)

Recap

- We have seen how basic control flow is handled in processors, in the form of **branches**.
- We have also seen how control flow behaviour may be made non-deterministic or data-controlled using **conditional branching**.

Week 10 Labs

This week you will:

- Explore the arithmetic unit (AU): addition, subtraction
- Chain AUs to handle data bigger than 4 bits.
- Build a looping counter.

Processor control flow (part 2)

Allowing modular programming

Sub-routines

- Sub-routines (or procedures, methods, functions) are a key component of modern modular programming.
- Supporting them efficiently in hardware is therefore very important.
- Most architectures, therefore provide some primitive support for sub-routines.

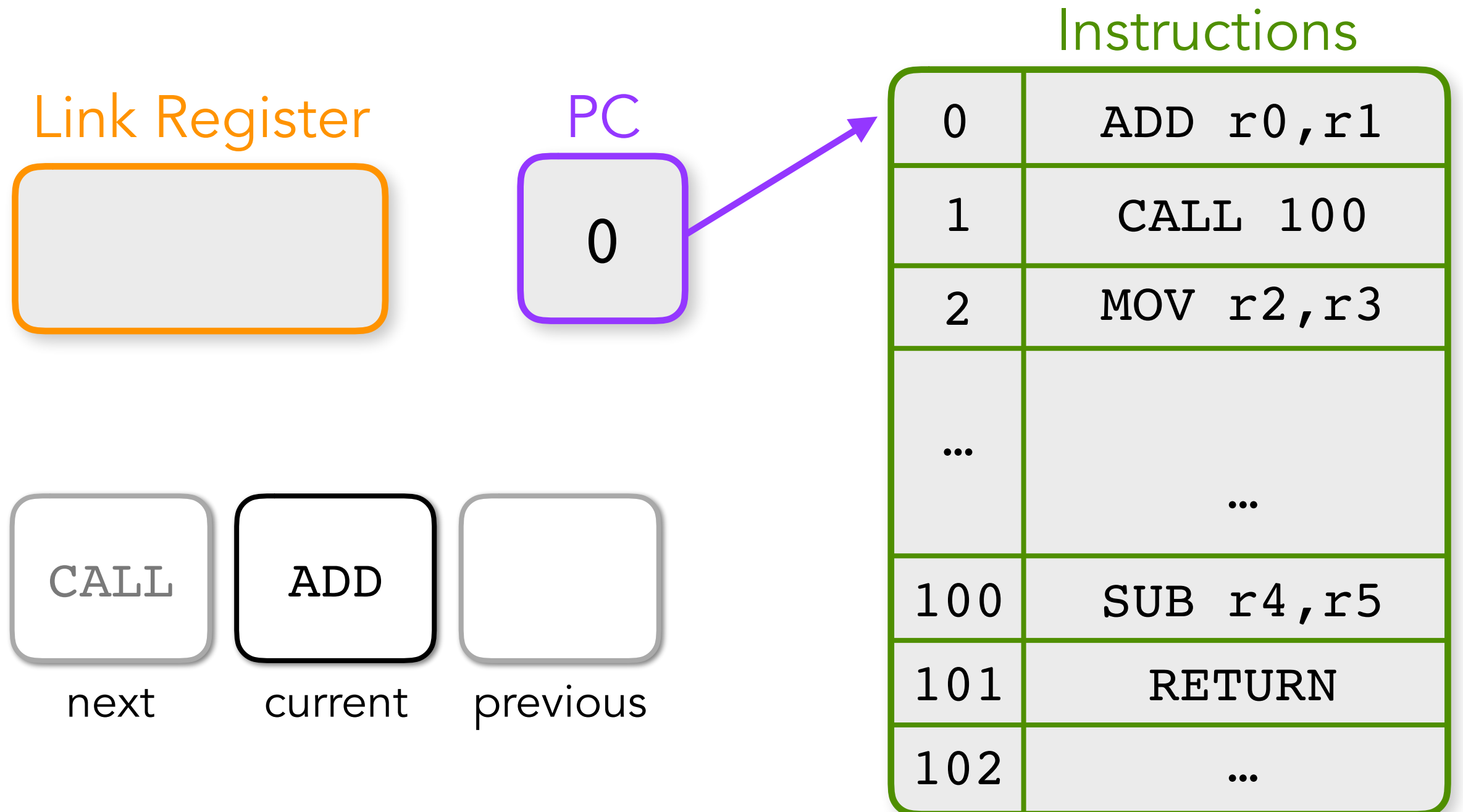
The Link-Register (LR)

- Most processor implementations make use of a component called a **link register** (LR).
- It's a register, like any other.
- It is used on sub-routine calls and exits, where it stores addresses.
- It may or may not be available for general purpose usage as well.

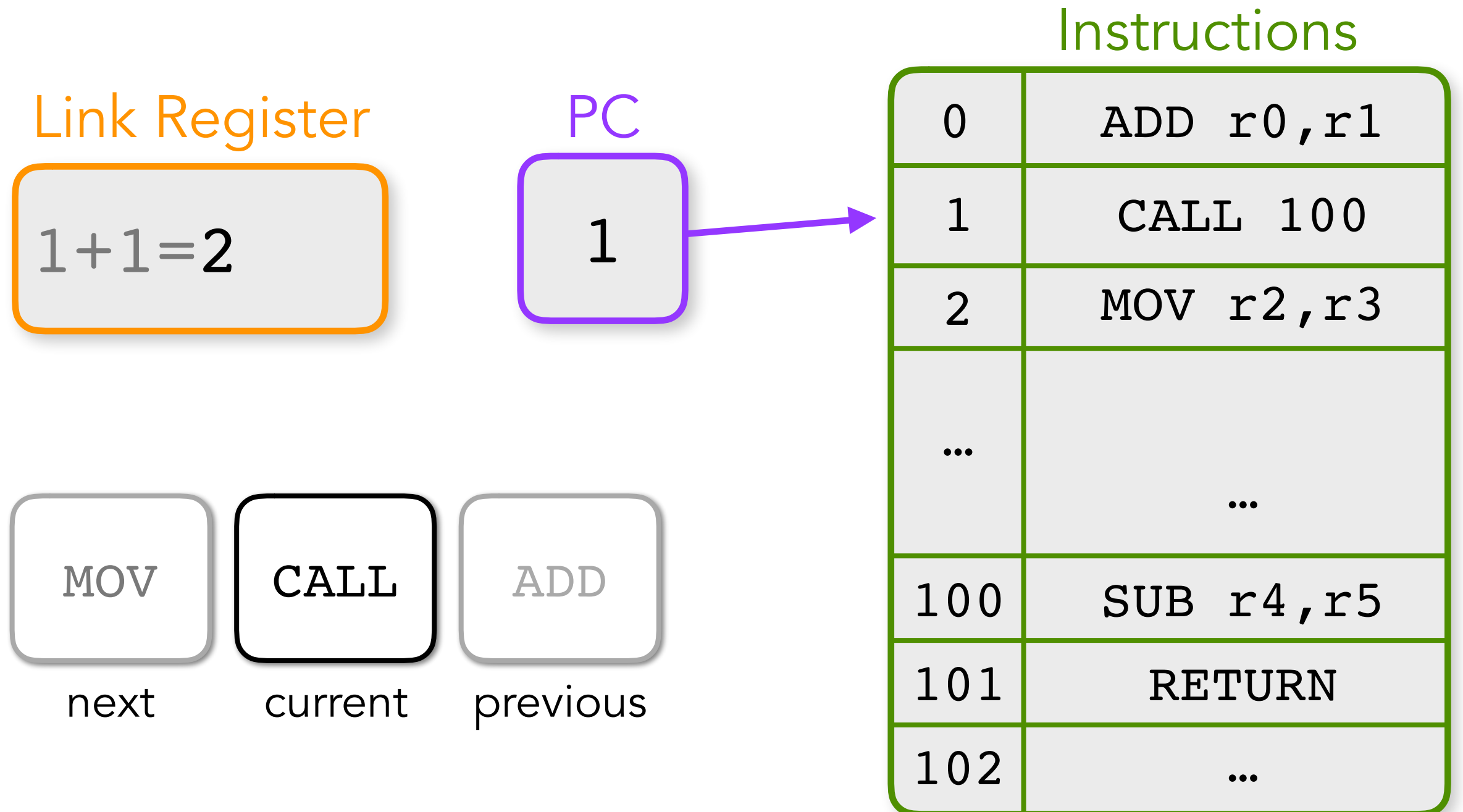
Register-based calling

- Most processors support instructions for calling and returning from sub-routines. For example:
- `CALL foo` = copy return address to link register, then branch to `foo`.
- `RETURN` = branch to link register address.

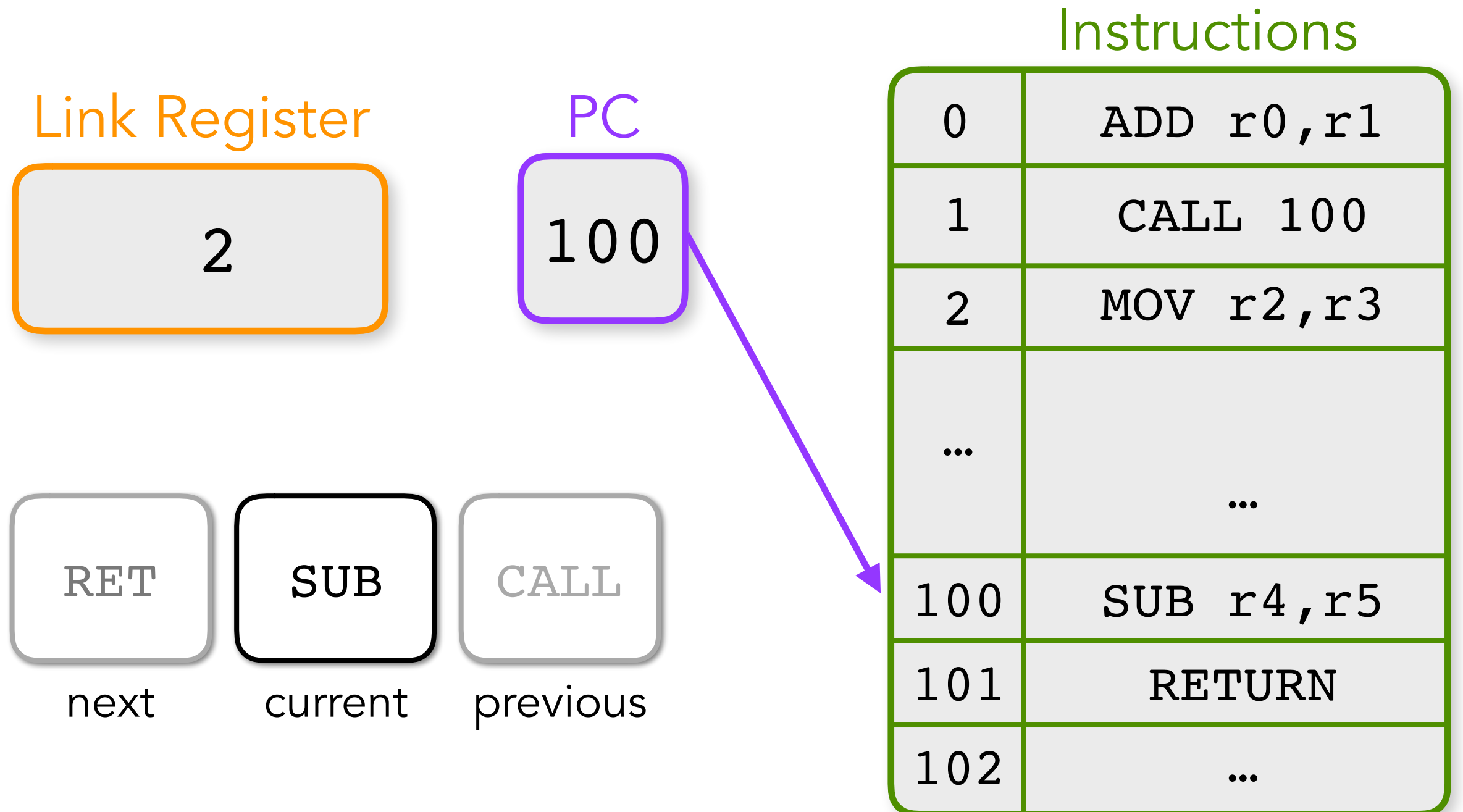
Register linking



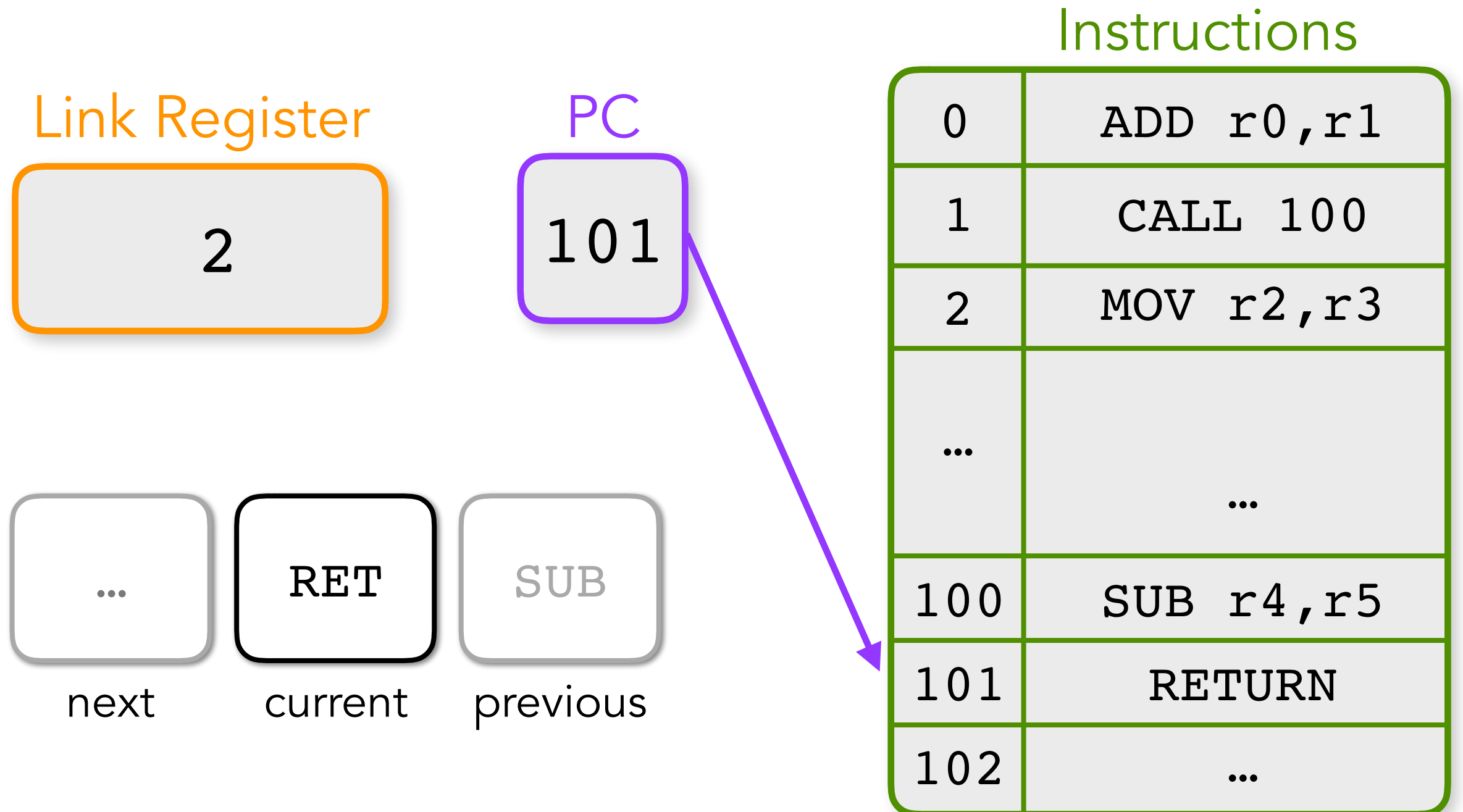
Register linking



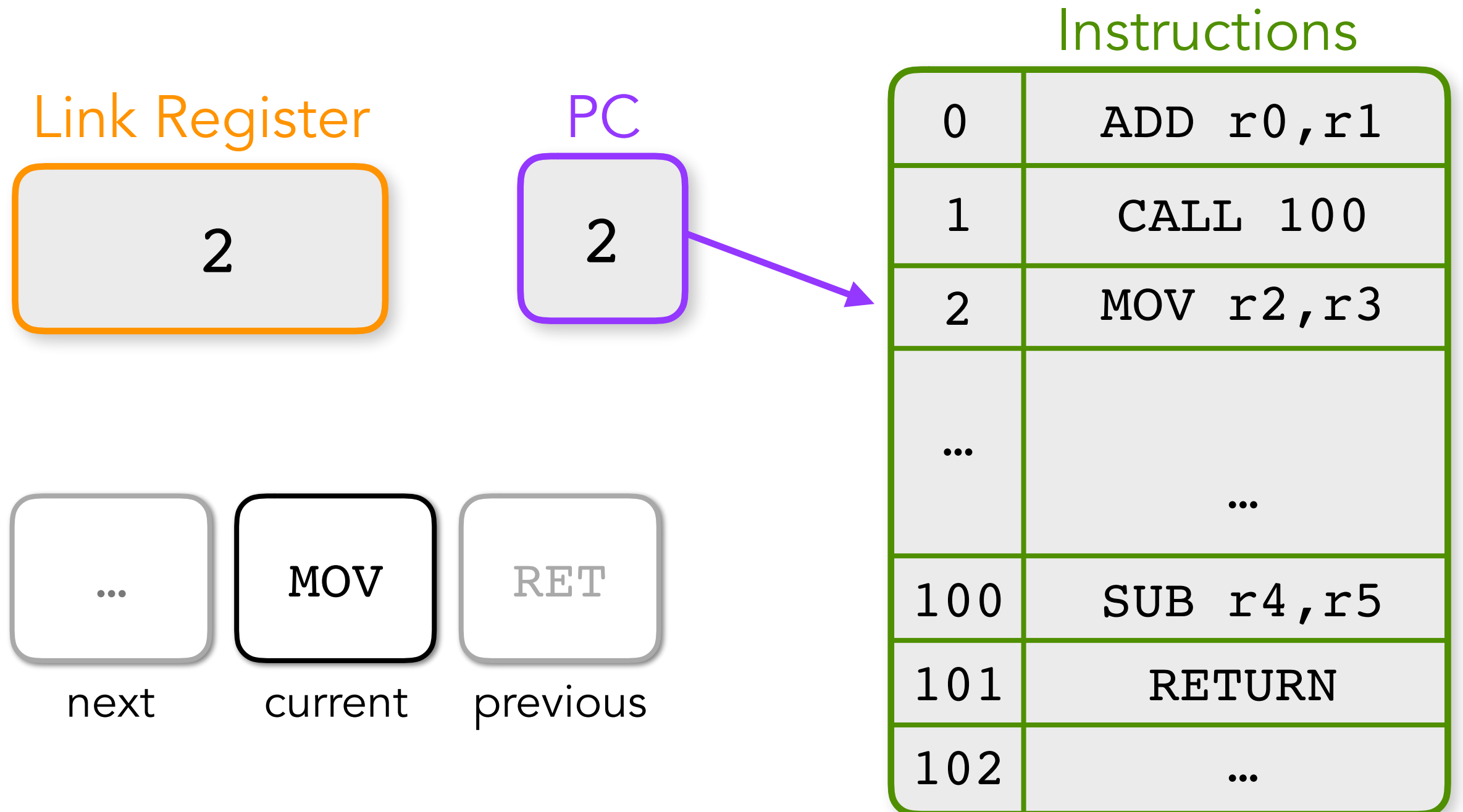
Register linking



Register linking



Register linking



Supporting multiple calls

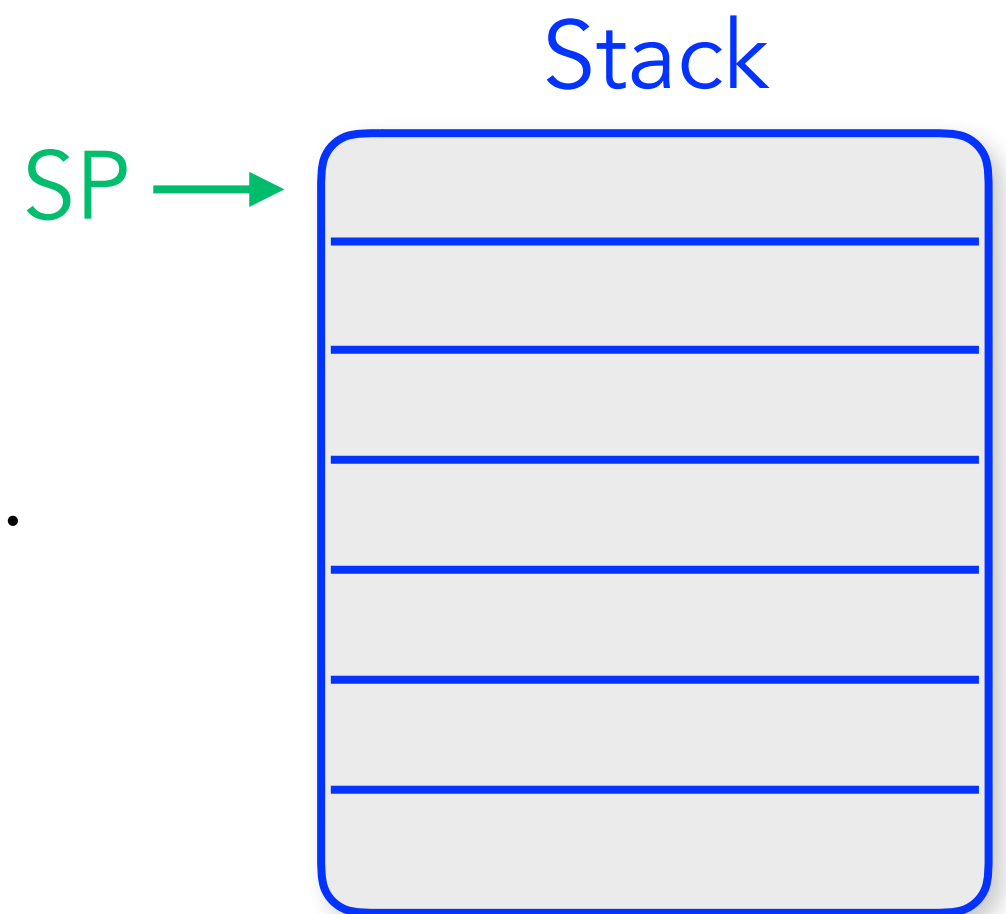
- How to allow for:
 - Multiple levels of a sub-routine call?
 - Recursive sub-routine calls?

Stack-based linking

- Produces a **stack** containing a list of return addresses.
- When calling, **PUSH** a return address **on** to the stack.
- When returning, **POP** a return address **off** the stack.

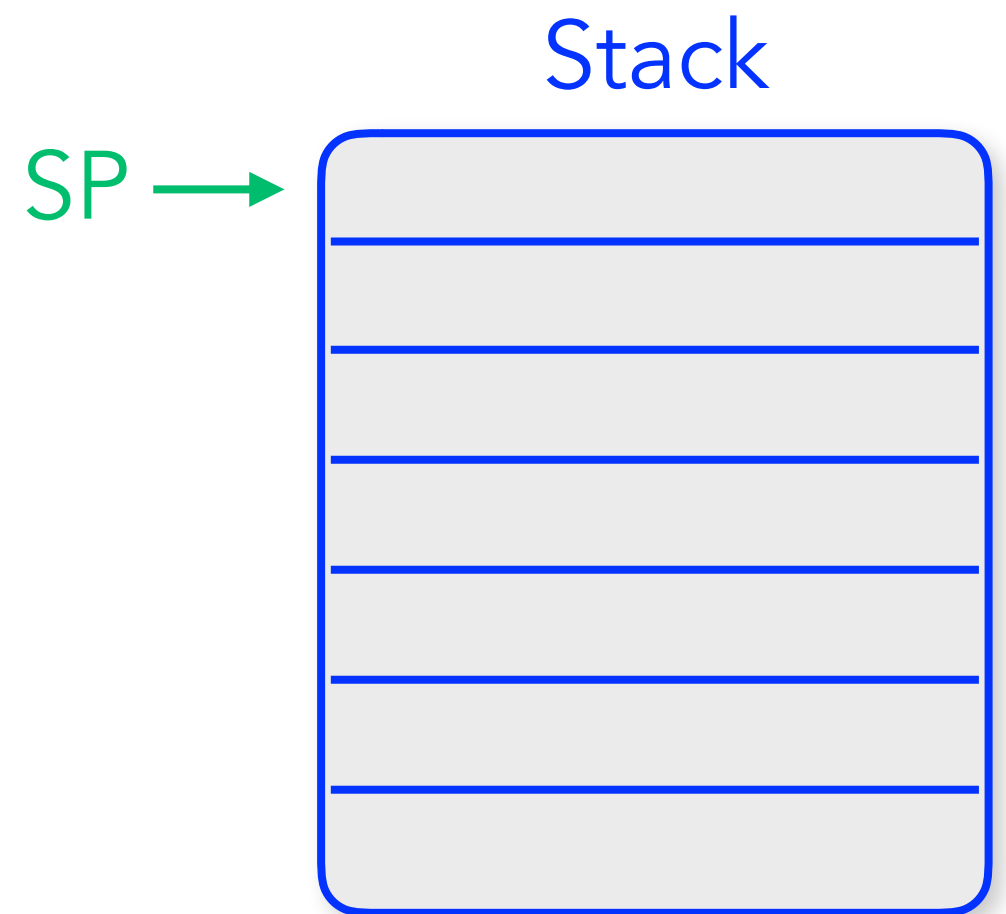
Recall the stack

- A stack is like a stack of plates.
- New items go on the top, and push down the previous ones.
- You can access the items at the top, but not those at the bottom.
- The number of items tracked by a "stack pointer".

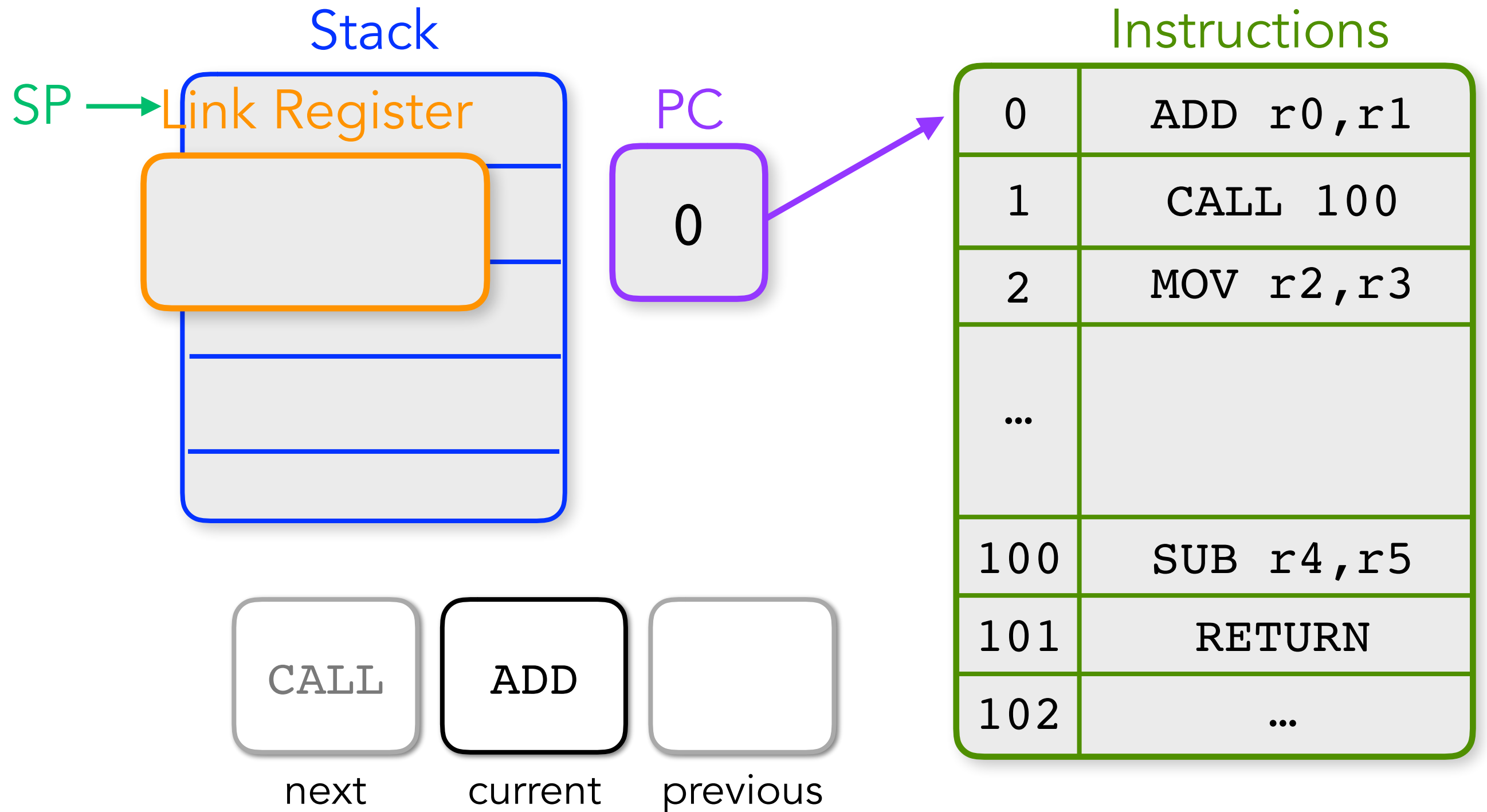


Stack example

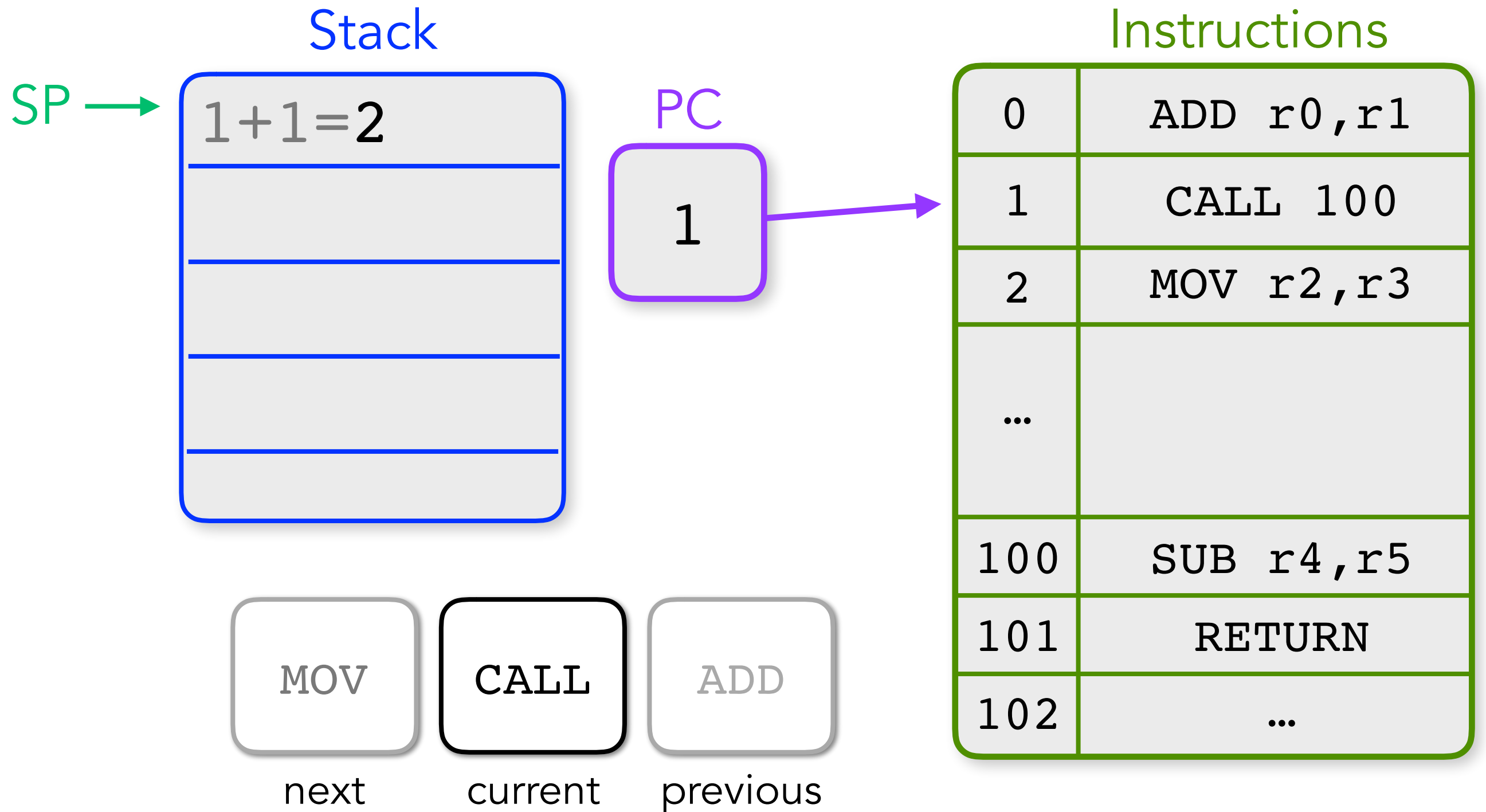
- Let's add the following values in order to the stack: 1, 2, 3.



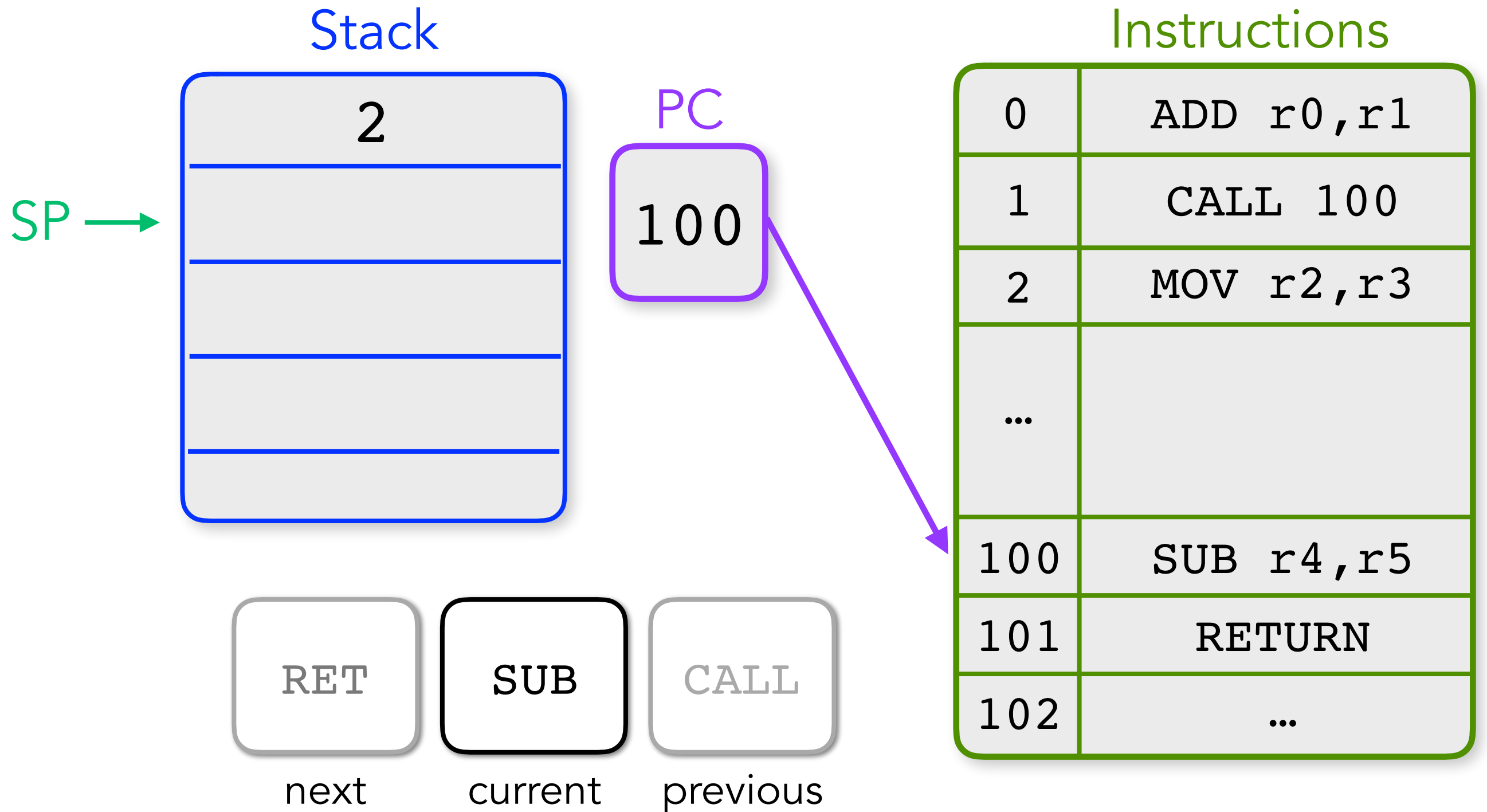
Stack-based linking



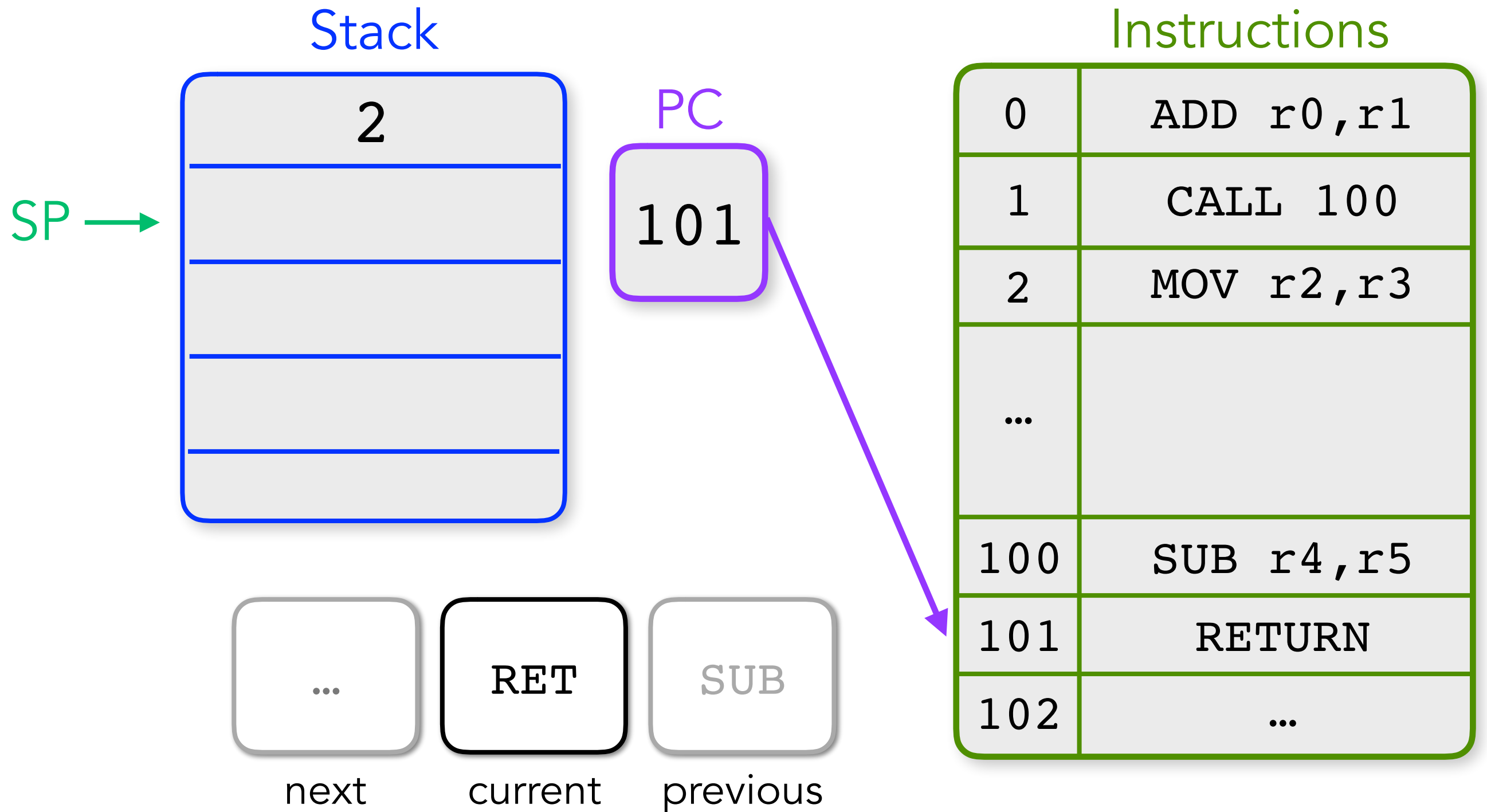
Stack-based linking



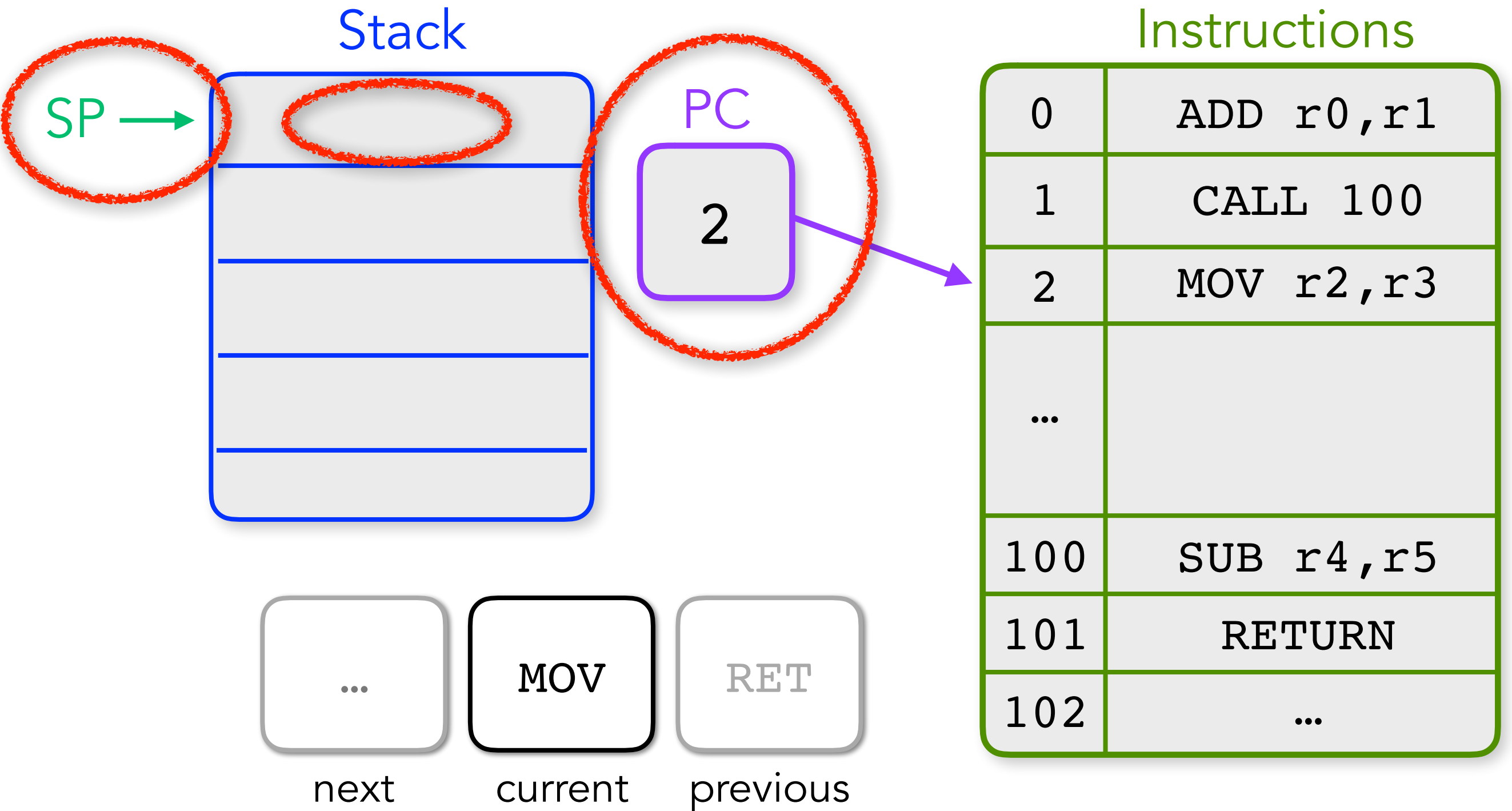
Stack-based linking



Stack-based linking



Stack-based linking



4. Processor control flow (2)

- We've seen that **control flow** is a vital part of modern programming flow.
- Therefore, we need **hardware support** to allow its use and provide efficient execution.
- We have seen:
 - Branches
 - Conditional flow
 - Sub-routine linking