Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Talen en Compilers

2009/2010, periode 2

Andres Löh

Department of Information and Computing Sciences
Utrecht University

December 15, 2009

9. Simple stack machine

Recap: Semantic functions

In the previous lectures, we have seen how to evaluate (interpret) expressions.

Recap: Semantic functions

In the previous lectures, we have seen how to evaluate (interpret) expressions.

- We have added variables and talked about environments.
- We have added local definitions and talked about nesting and blocks.
- We have added (mutually) recursive definitions and talked about scoping.

Recap: Semantic functions

In the previous lectures, we have seen how to evaluate (interpret) expressions.

- We have added variables and talked about environments.
- We have added local definitions and talked about nesting and blocks.
- We have added (mutually) recursive definitions and talked about scoping.

Now we are going to generate code in a low-level language instead of interpreting the expression directly.

This lecture

Simple stack machine

- Architecture of the simple stack machine
- Instructions
- Translating programs
- Functions / methods

9.1 Architecture of the simple stack machine

Simple stack machine

A virtual machine that executes programs consisting of assembly language instructions.

Simple stack machine

A virtual machine that executes programs consisting of assembly language instructions.

- The program is a list of instructions with arguments, stored in a continuous block of memory.
- A **stack** is used to store the current state of execution.
- There are eight **registers**, four with a special name:
  - the **program counter** (PC)
  - the **stack pointer** (SP)
  - the **mark pointer** (MP)
  - the **return register** (RR)

Simple stack machine

A virtual machine that executes programs consisting of assembly language instructions.

- The program is a list of instructions with arguments, stored in a continuous block of memory.
- A **stack** is used to store the current state of execution.
- There are eight **registers**, four with a special name:
  - the **program counter** (PC)
  - the **stack pointer** (SP)
  - the **mark pointer** (MP)
  - the **return register** (RR)

Question

Why a stack?

Execution

- A step in the execution interprets the instruction pointed to by the program counter.
- Depending on the instruction, the contents of the stack and registers are modified.

Execution

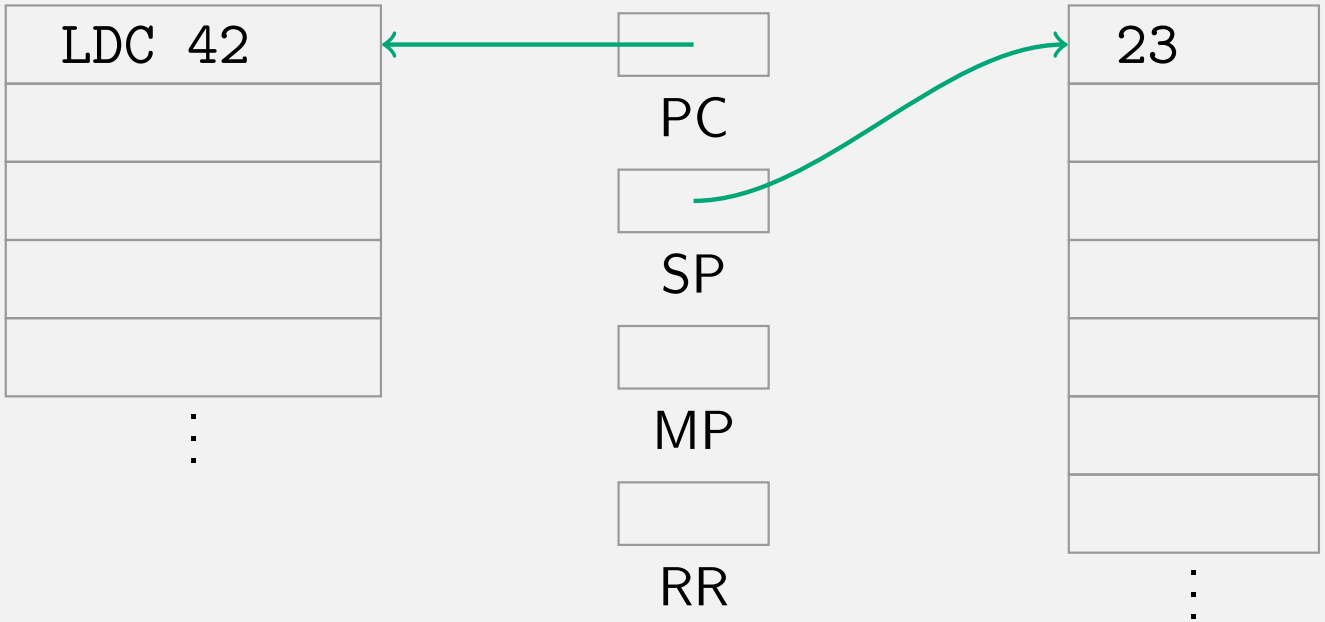
- A step in the execution interprets the instruction pointed to by the program counter.
- Depending on the instruction, the contents of the stack and registers are modified.

Example: `LDC` (load constant)

$$
\begin{array}{lll}
SP_{post} & = SP_{pre} + 1 & \text{(increment stack pointer)} \\
M_{post}\,[SP_{post}] & = M_{pre}\,[PC_{pre} + 1] & \text{(place argument on stack)} \\
PC_{post} & = PC_{pre} + 2 & \text{(adjust program counter)}
\end{array}
$$

Visualizing the execution

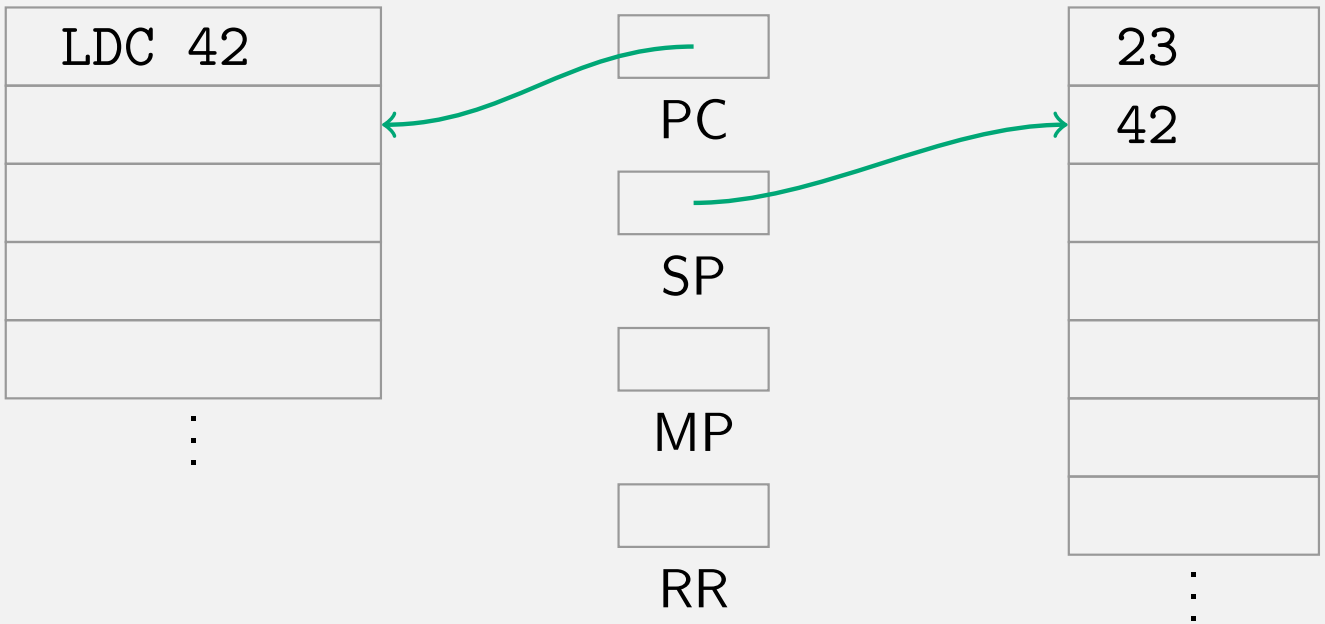
$$
\begin{aligned}
SP_{post} &= SP_{pre} + 1 && \text{(increment stack pointer)} \\
M_{post}\,[SP_{post}] &= M_{pre}\,[PC_{pre} + 1] && \text{(place argument on stack)} \\
PC_{post} &= PC_{pre} + 2 && \text{(adjust program counter)}
\end{aligned}
$$



The instruction `LDC 42` takes up two words in memory, but we write it in one cell.

Visualizing the execution

$$
\begin{aligned}
SP_{post} &= SP_{pre} + 1 && \text{(increment stack pointer)} \\
M_{post}\,[SP_{post}] &= M_{pre}\,[PC_{pre} + 1] && \text{(place argument on stack)} \\
PC_{post} &= PC_{pre} + 2 && \text{(adjust program counter)}
\end{aligned}
$$



The instruction `LDC 42` takes up two words in memory, but we write it in one cell.

9.2 Instructions

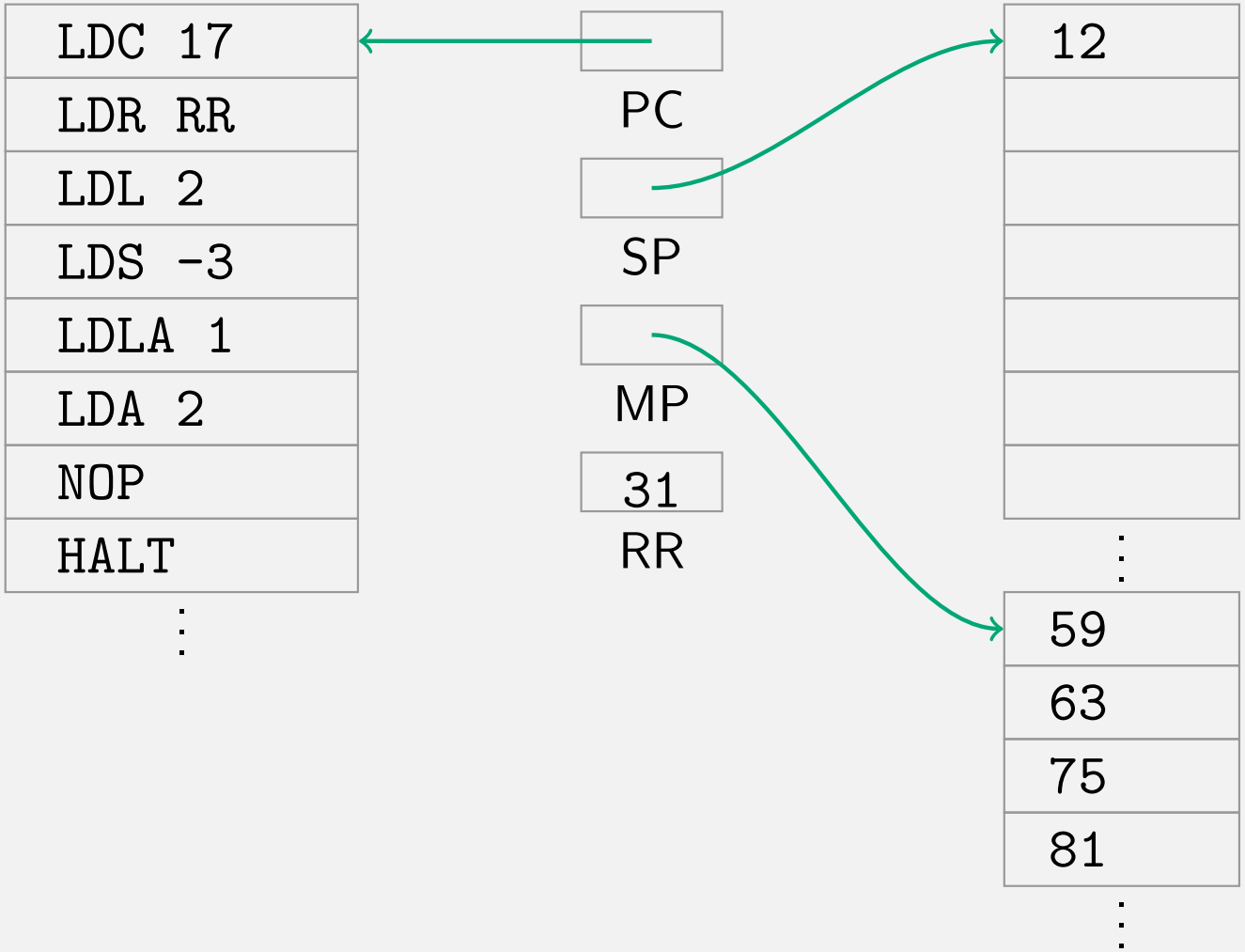
Instructions

Most instructions can be classified into the following groups:

- load instructions
- store instructions
- jump instructions
- arithmetic and logical operations

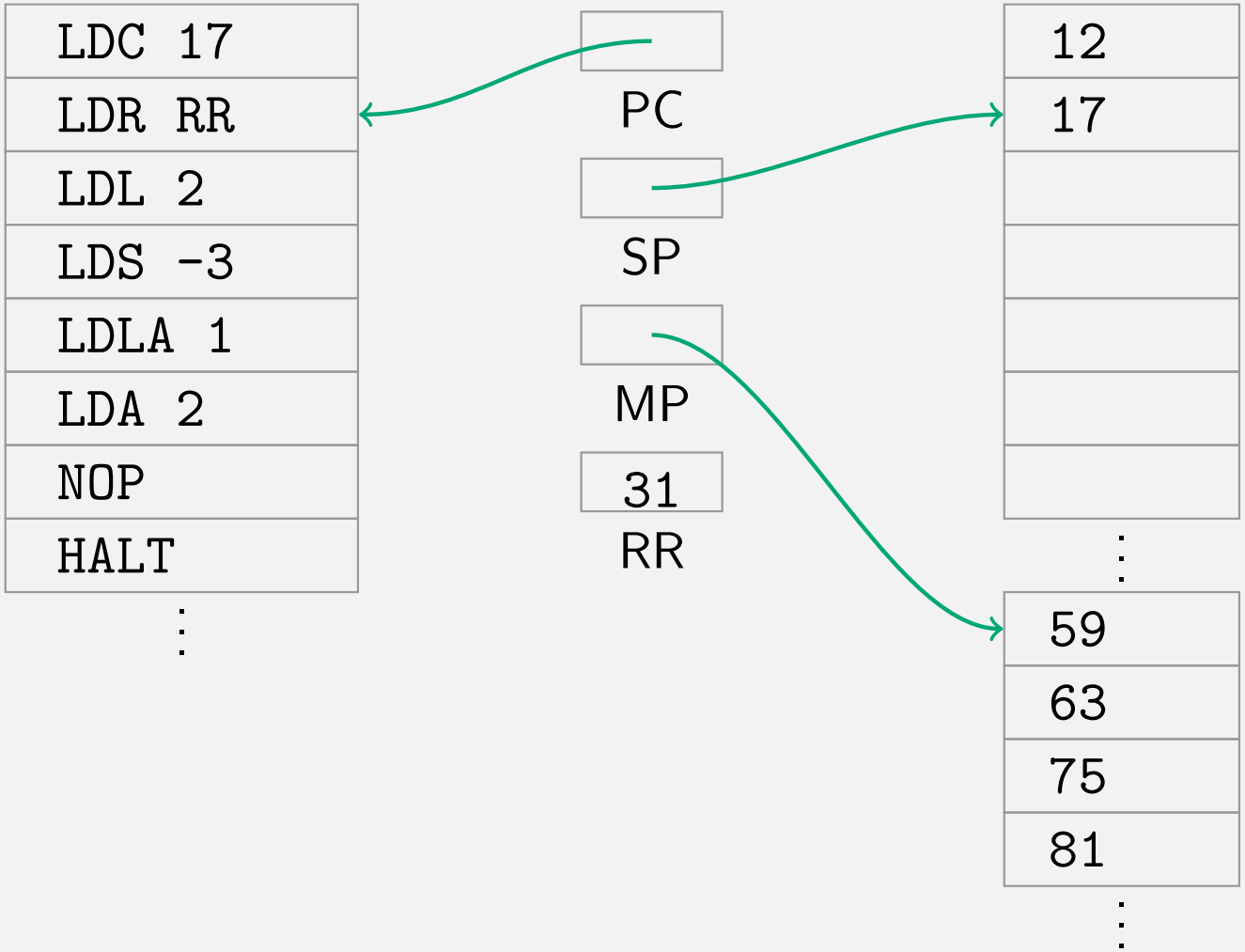
Load instructions

LDC – load constant



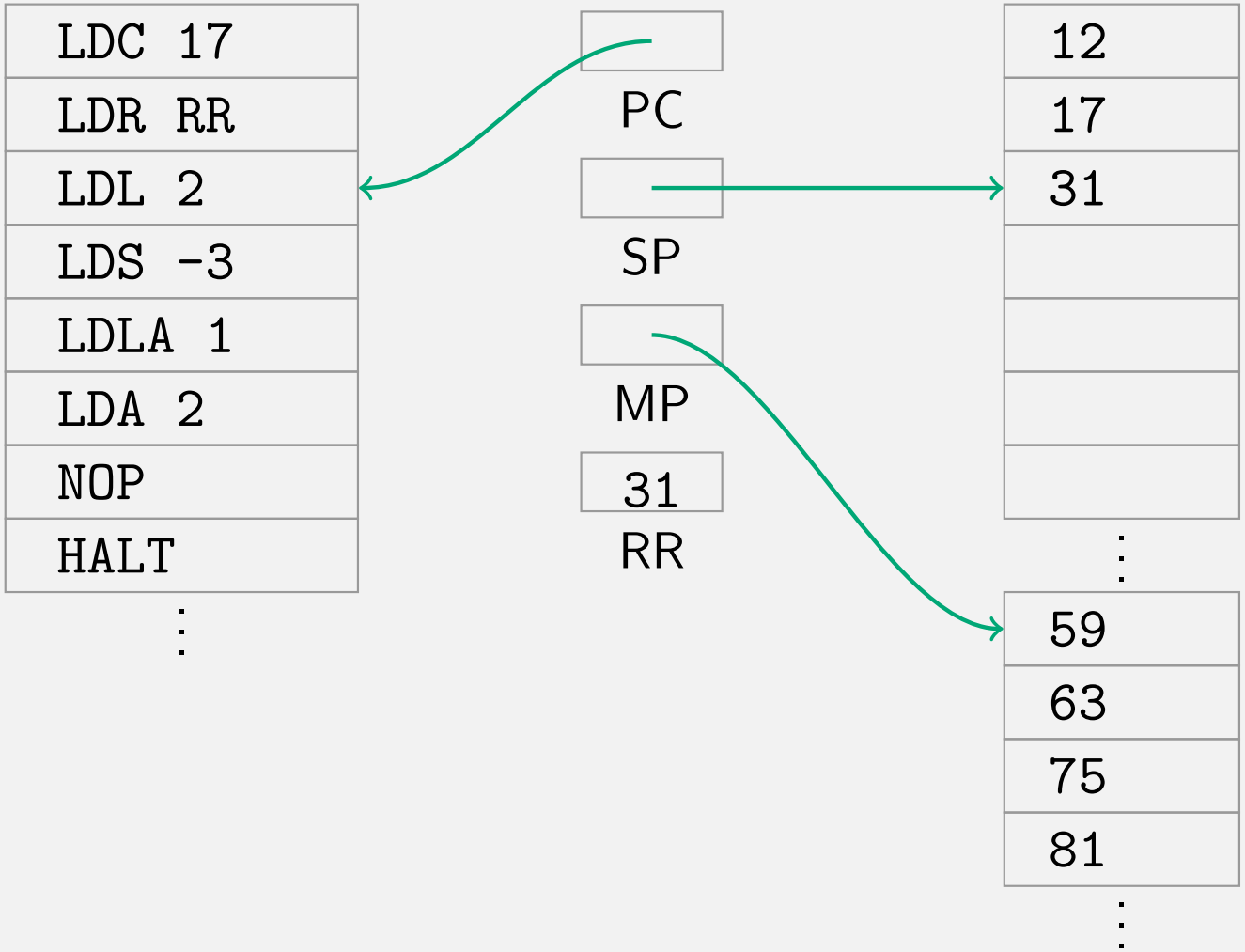
Load instructions

LDR – load from register



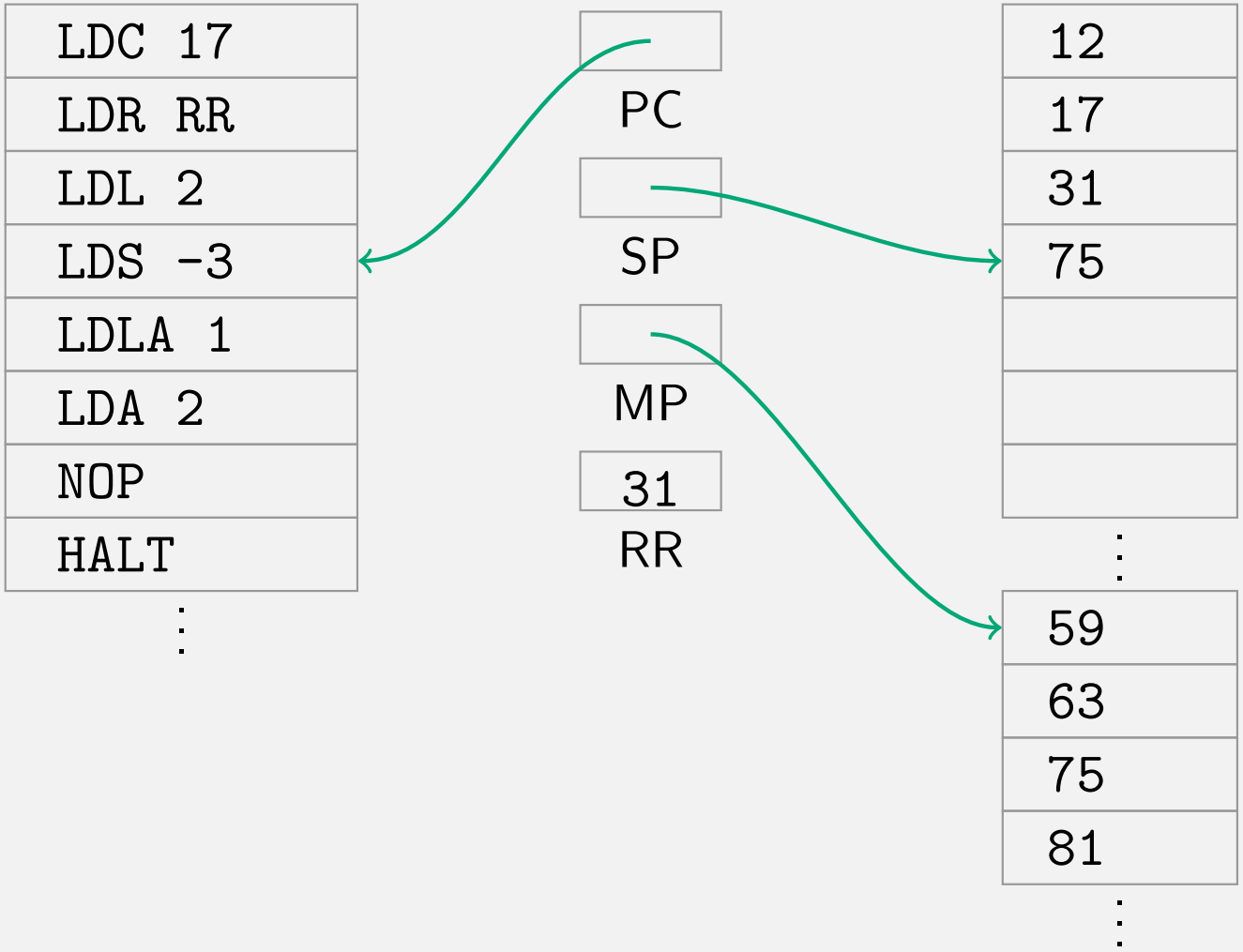
Load instructions

LDL – load local



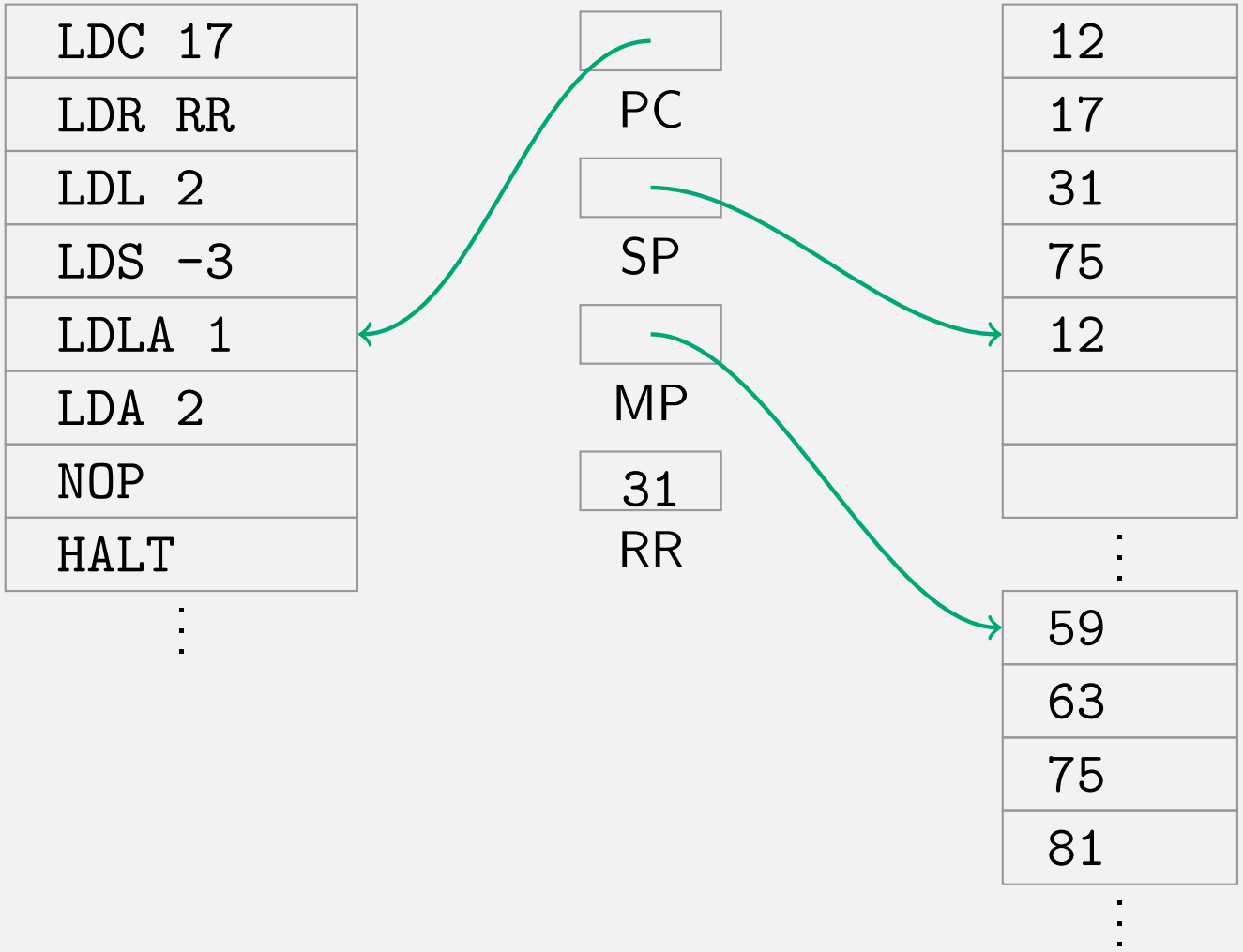
Load instructions

LDS – load from stack



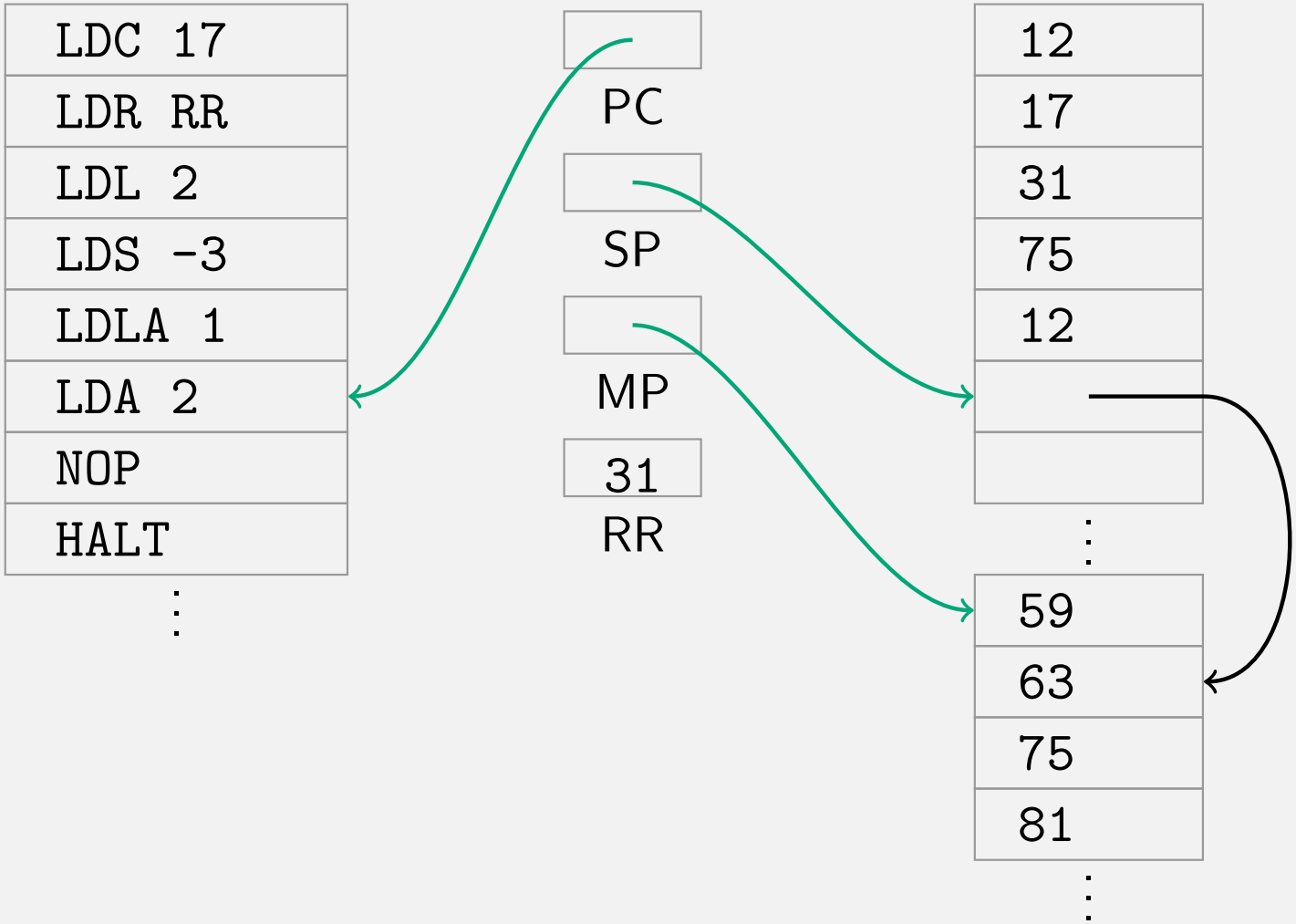
Load instructions

LDLA – load local address



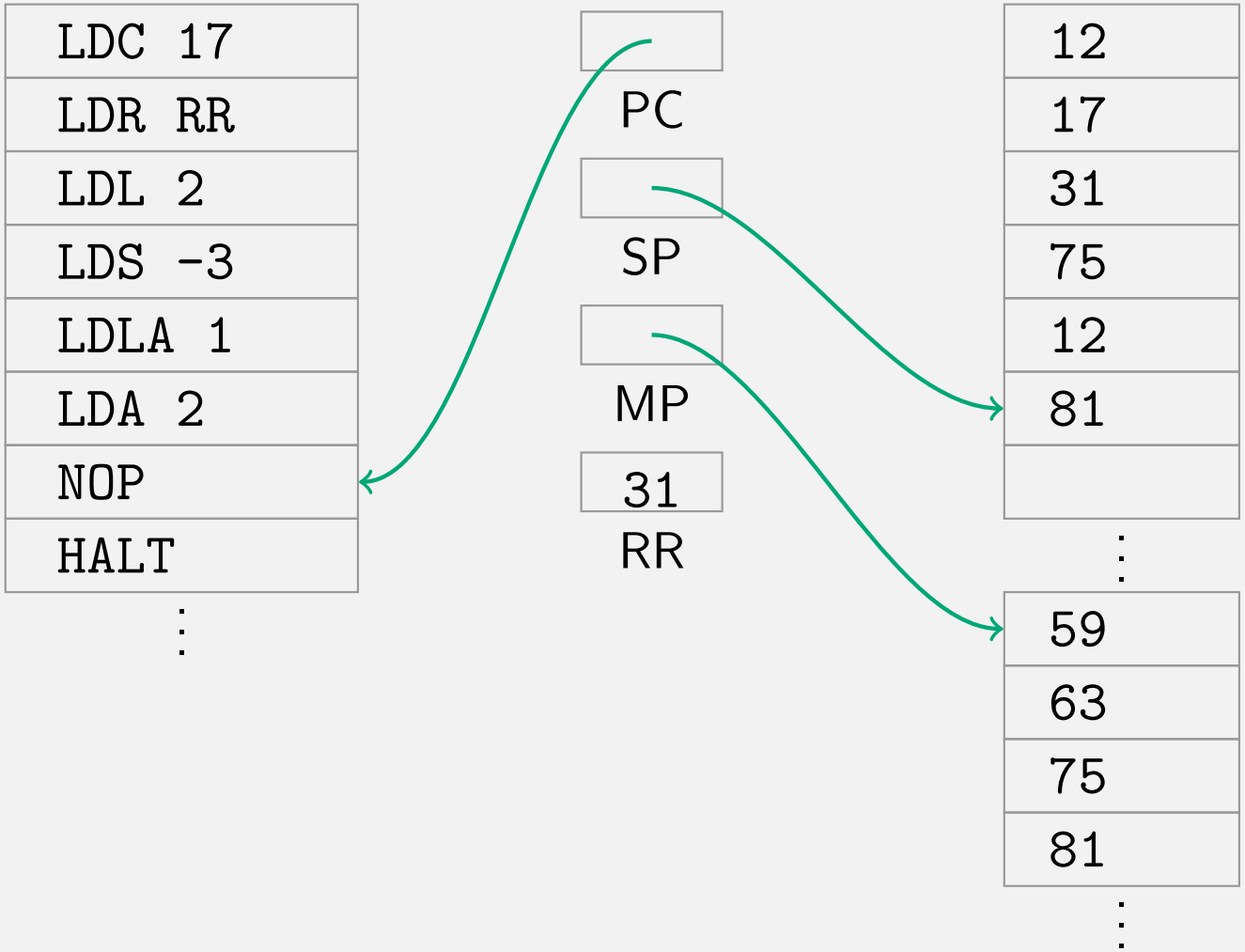
Load instructions

LDA – load via address



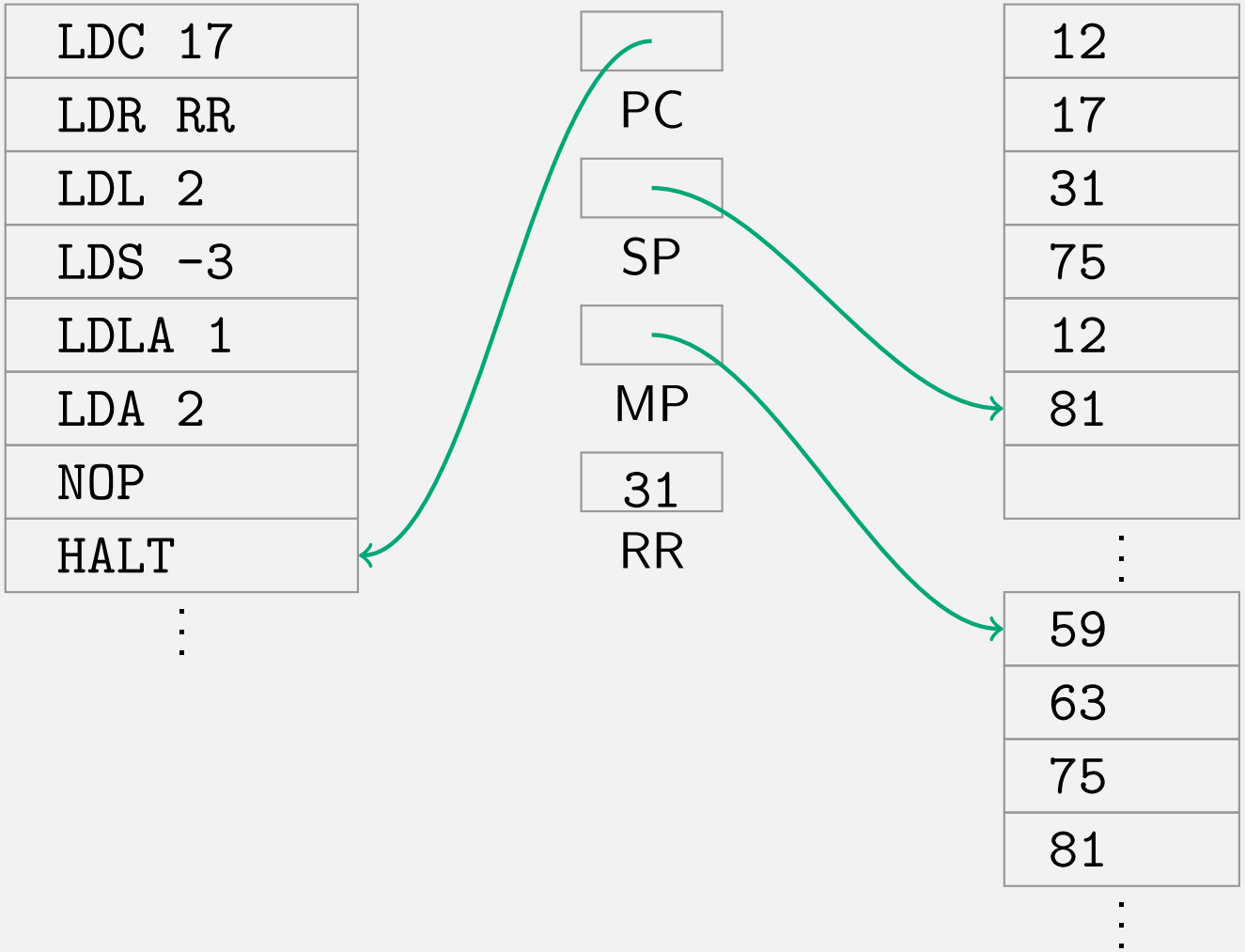
Load instructions

NOP – noop



Load instructions

HALT – halt program



Load and store instructions

`LDC`

load constant

`LDR`

load from register

`STR`

store to register

`LDL`

load local

`STL`

store local

`LDS`

load from stack

`STS`

store to stack

`LDLA`

load local address

`LDA`

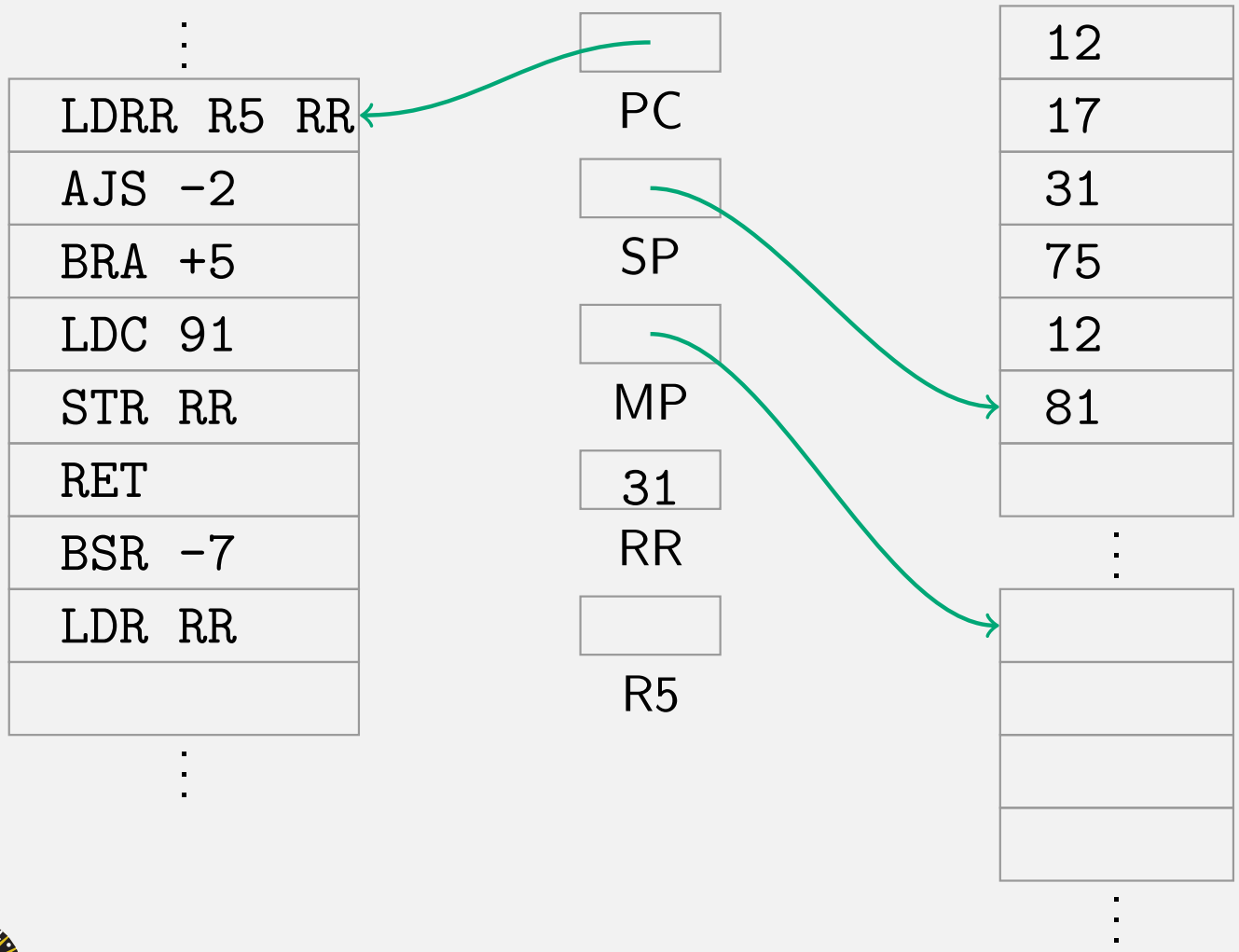
load via address

`SDA`

store via address

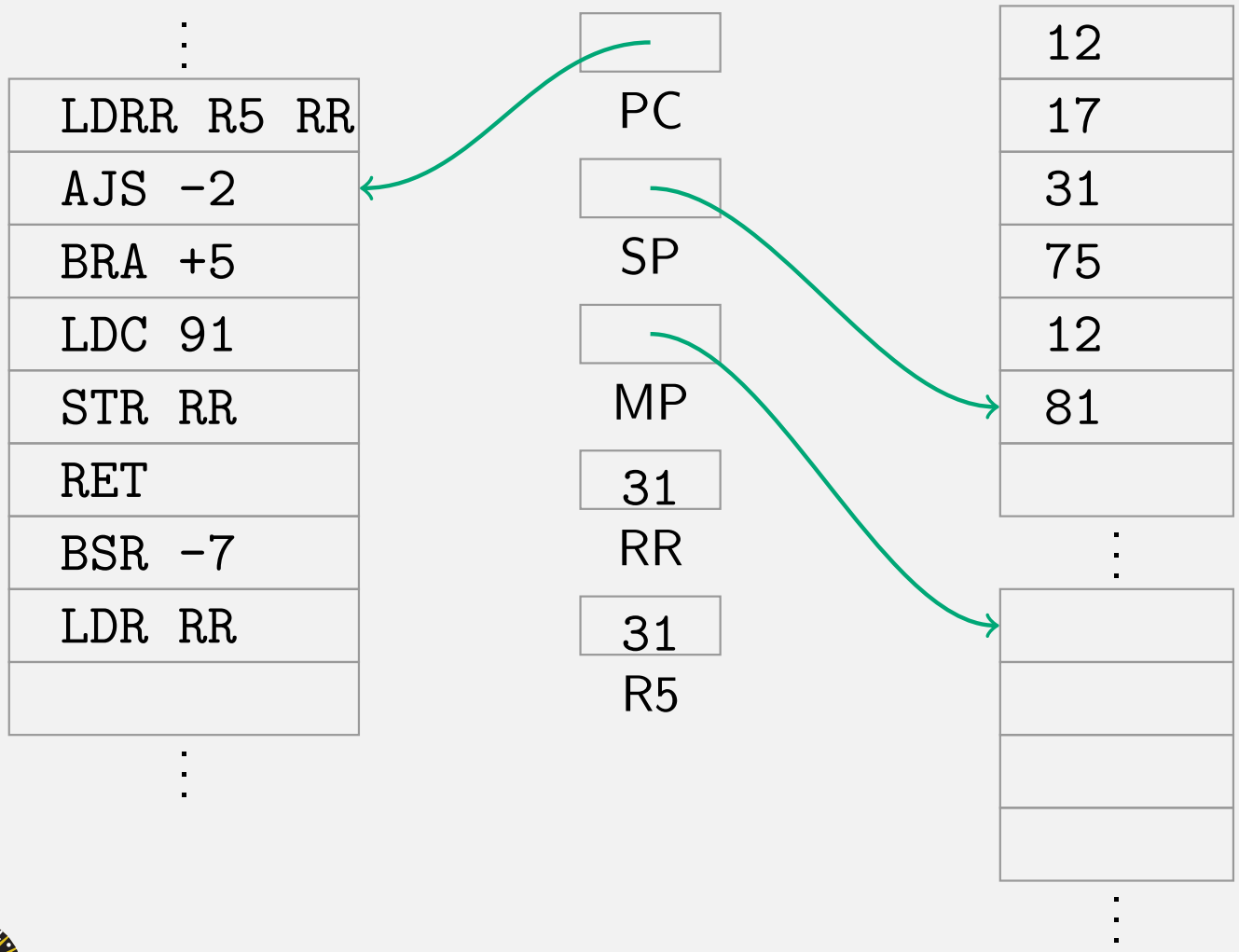
Register and jump instructions

LDRR – load register from register



Register and jump instructions

AJS – adjust stack pointer

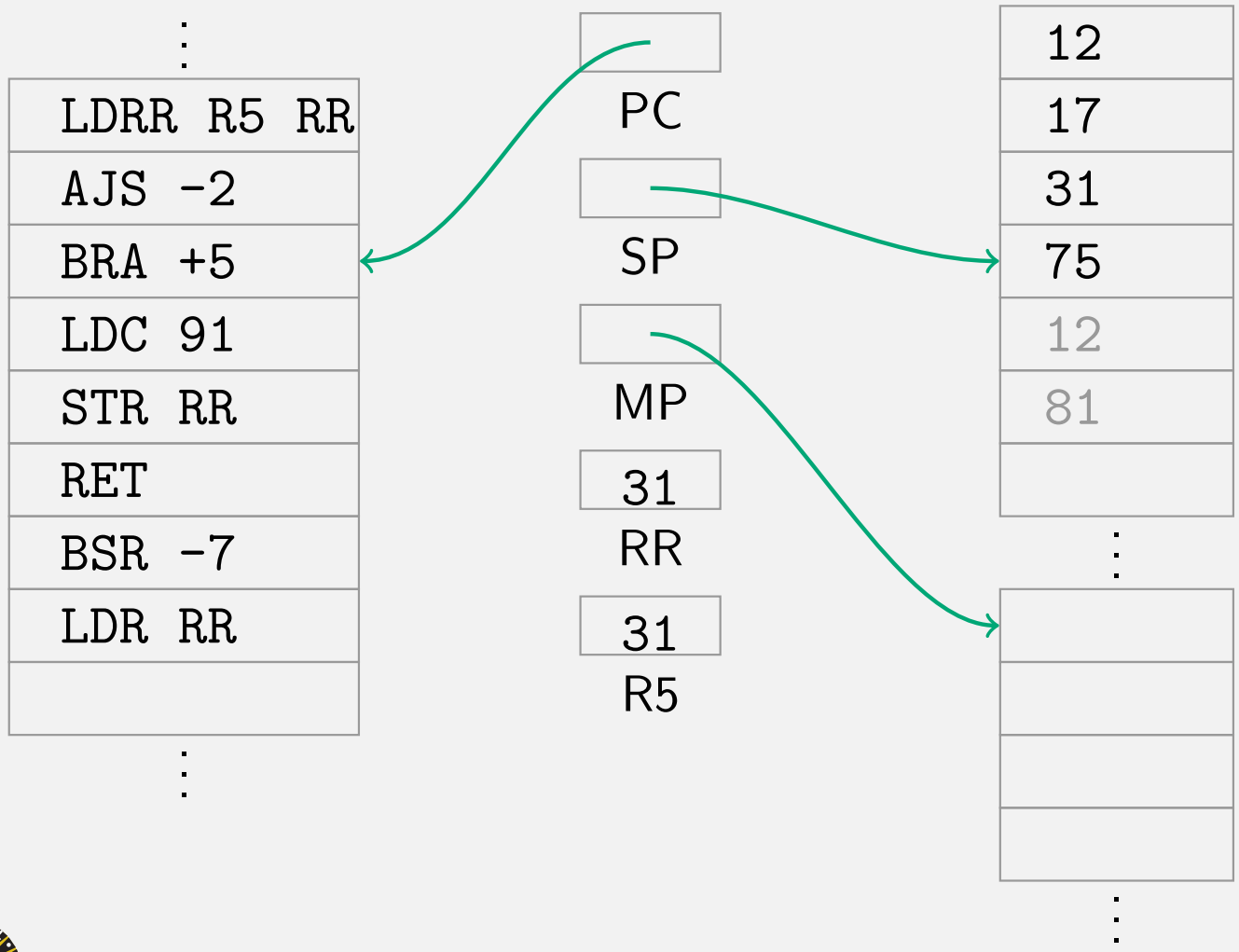


Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

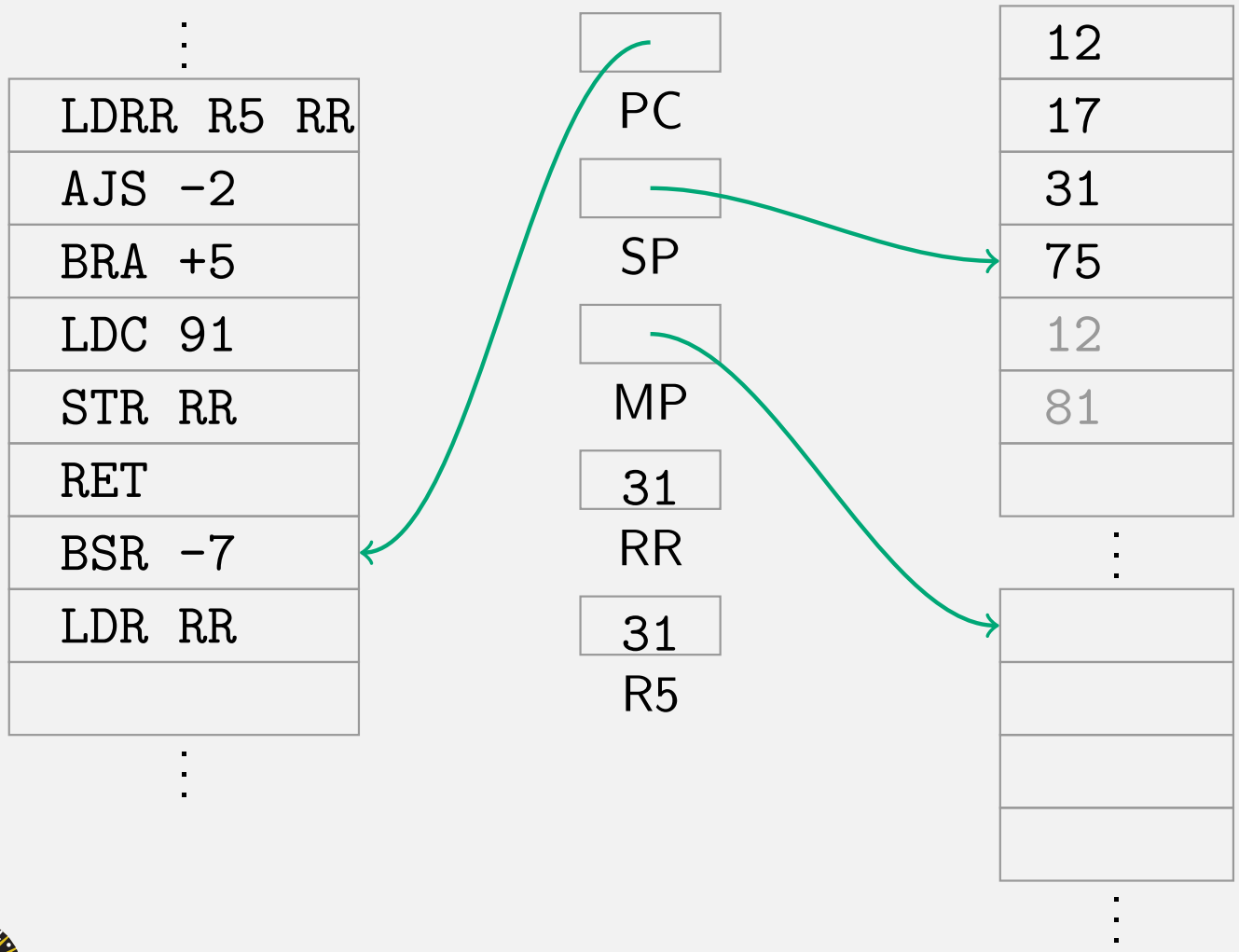
Register and jump instructions

BRA – unconditional branch



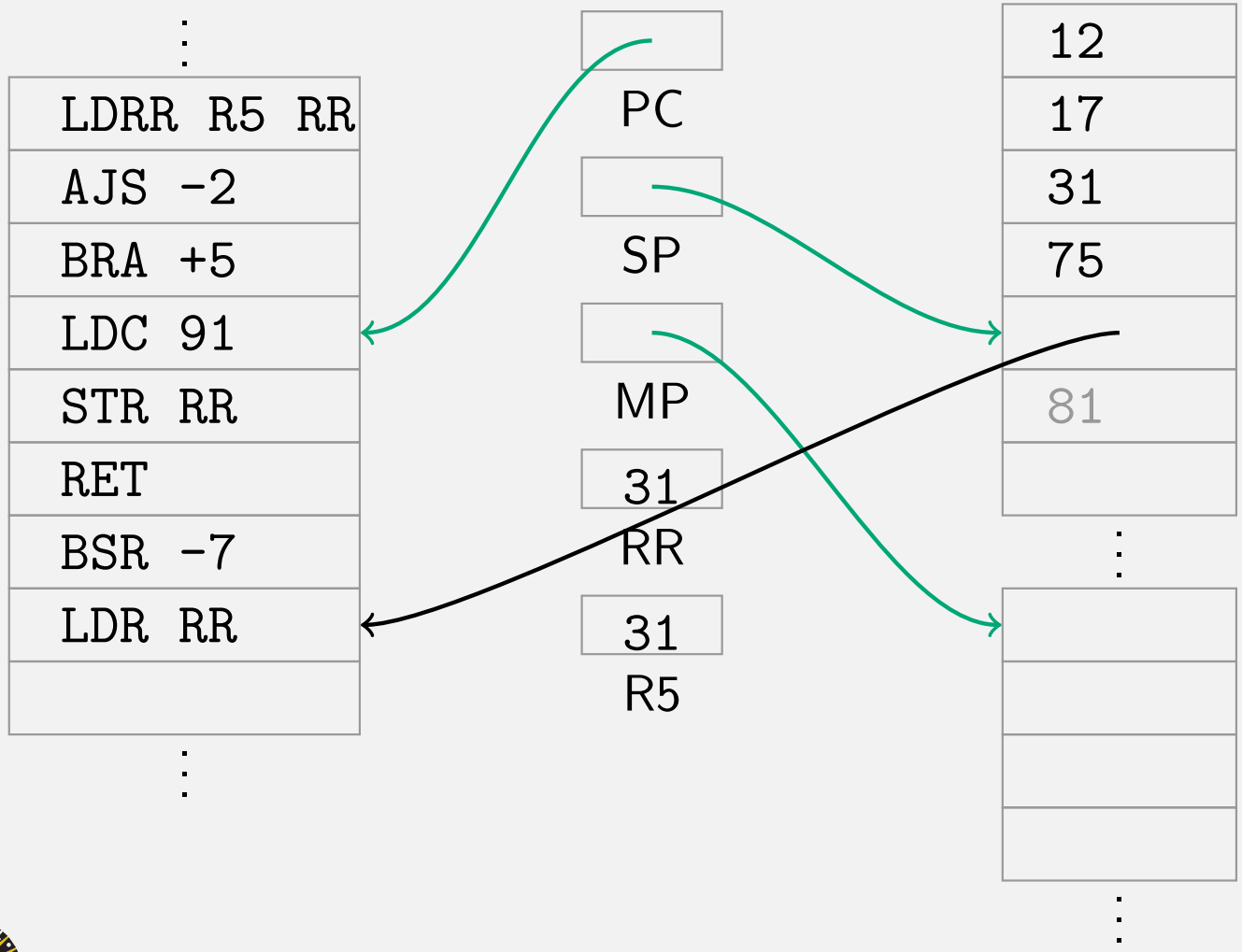
Register and jump instructions

BSR – branch to subroutine



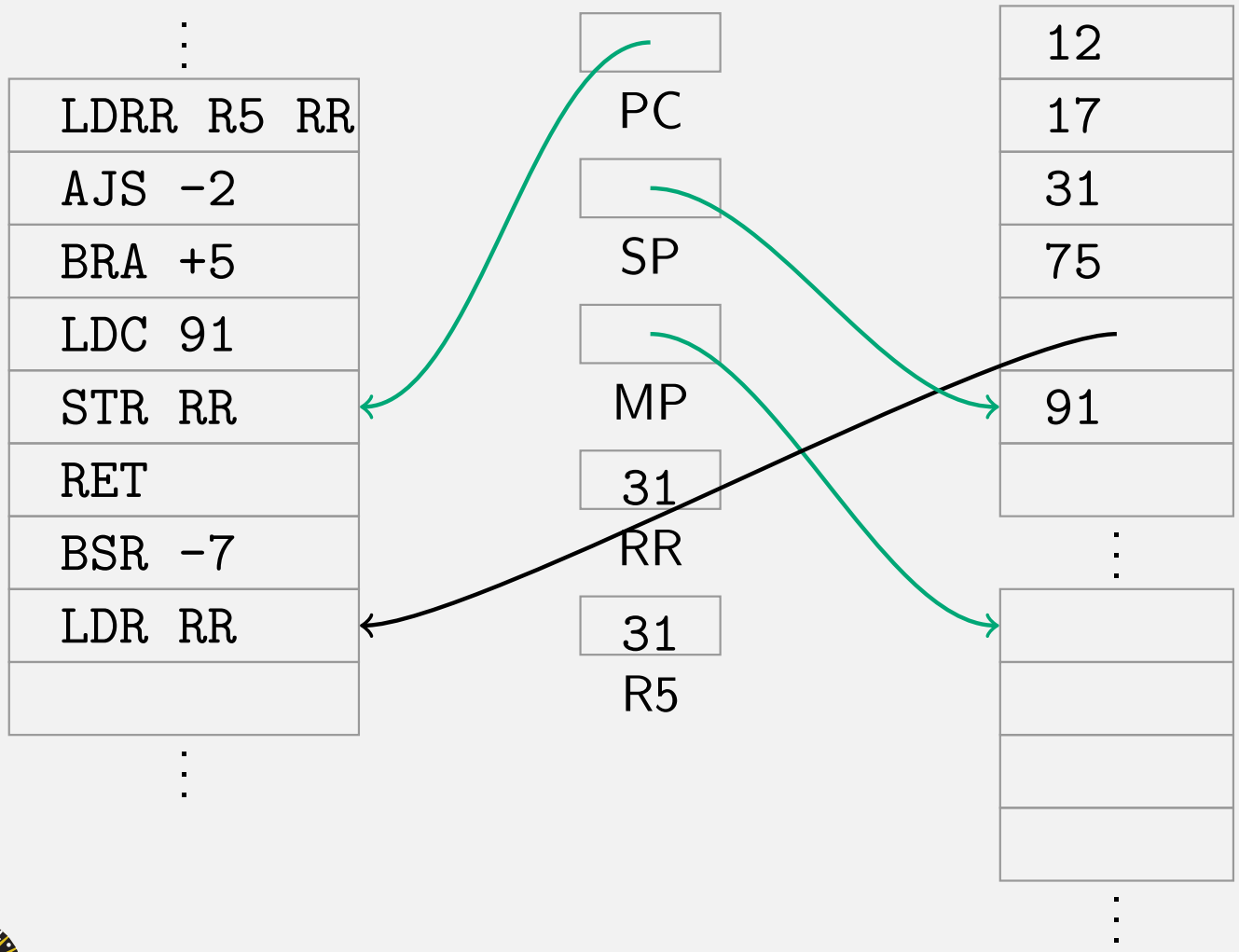
Register and jump instructions

LDC – load constant



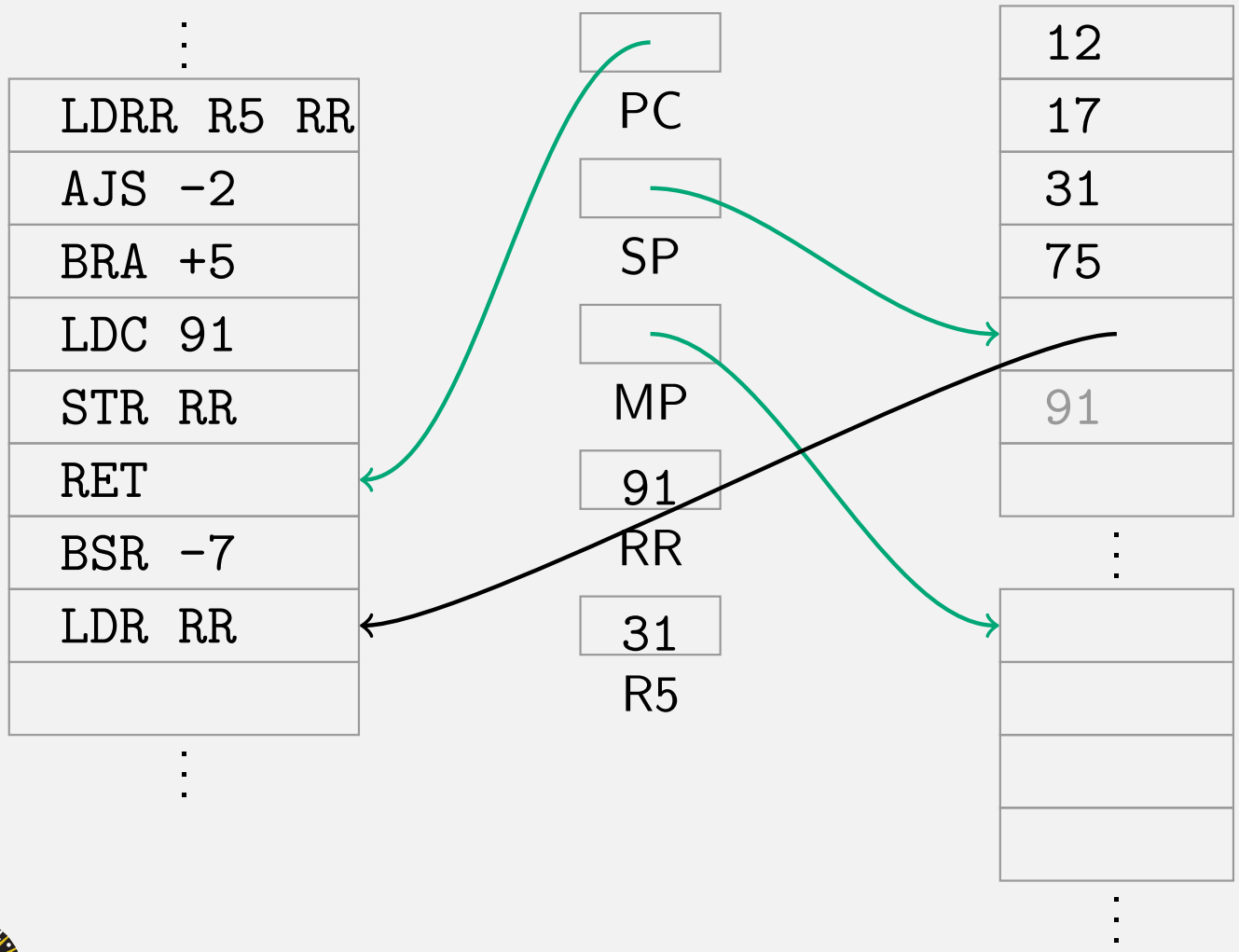
Register and jump instructions

STR – store to register



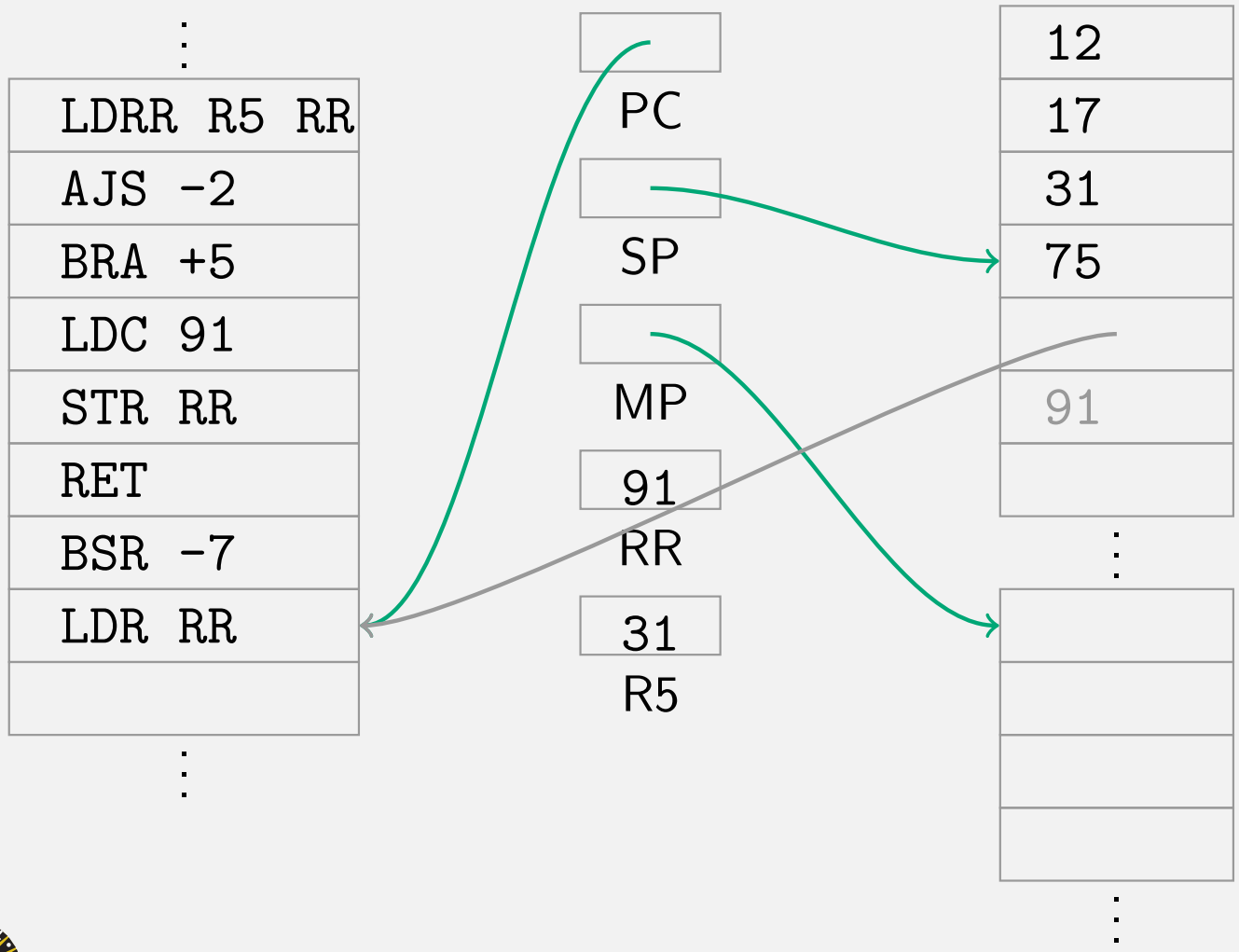
Register and jump instructions

RET – return

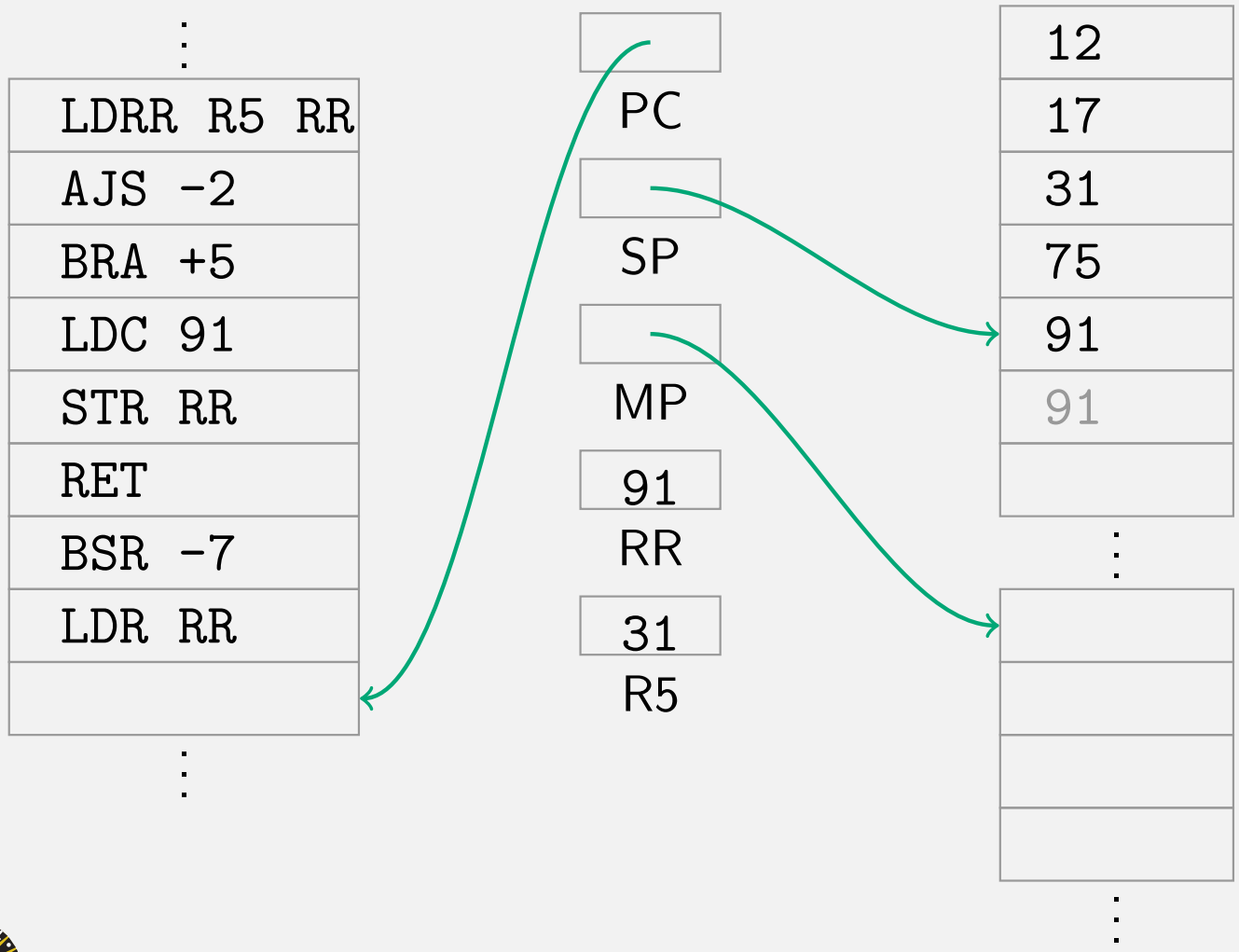


Register and jump instructions

LDR – load from register



Register and jump instructions



Branch instructions

`BRA`

unconditional branch (jump)

`BRT`

branch on true

`BRF`

branch on false

`BSR`

branch to subroutine (puts return address on stack)

Operators

Operators remove one (or two) stack arguments, and put the result back on the stack.

Binary operators			Unary operators
`ADD`	`AND`	`EQ`	`NOT`
`SUB`	`OR`	`NE`	`NEG`
`MUL`	`XOR`	`LT`	
`DIV`		`GT`	
`MOD`		`LE`	
		`GE`	

9.3 Translating programs

Arithmetic expressions

Expression

```
3+4*7+2
```

Arithmetic expressions

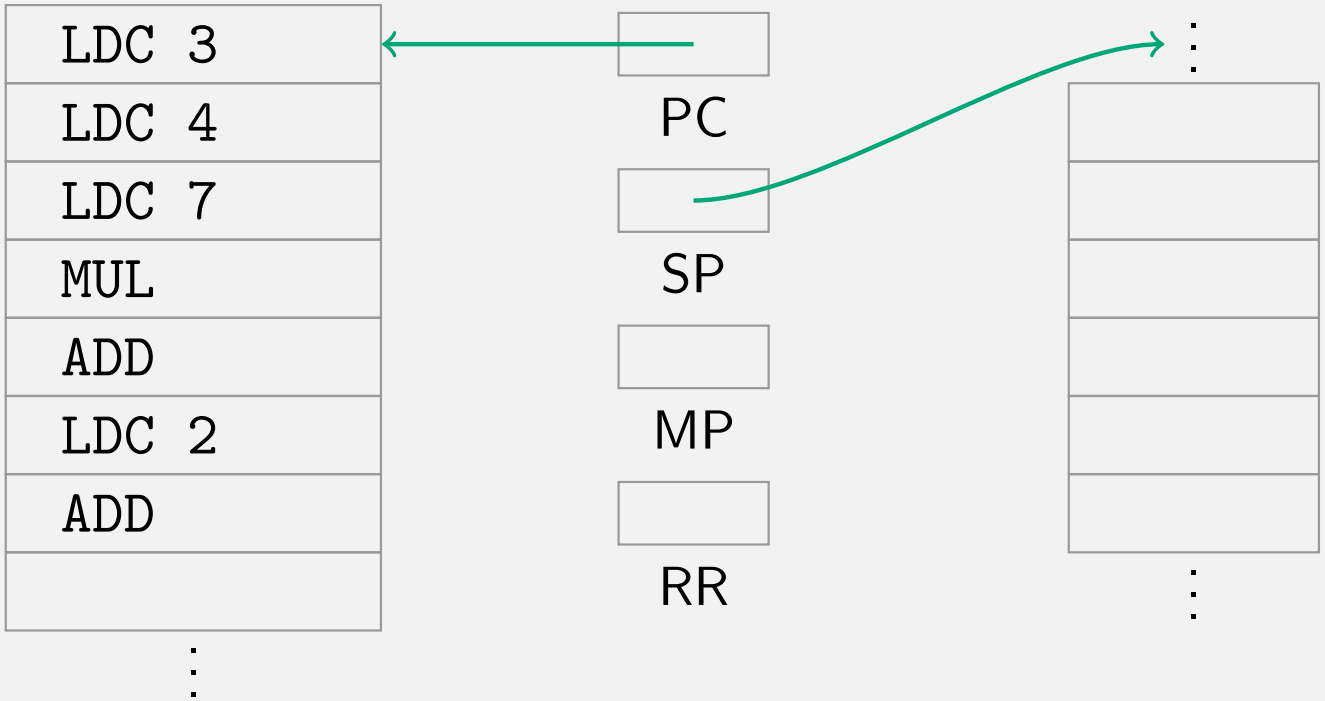
Expression

```
3+4*7+2
```

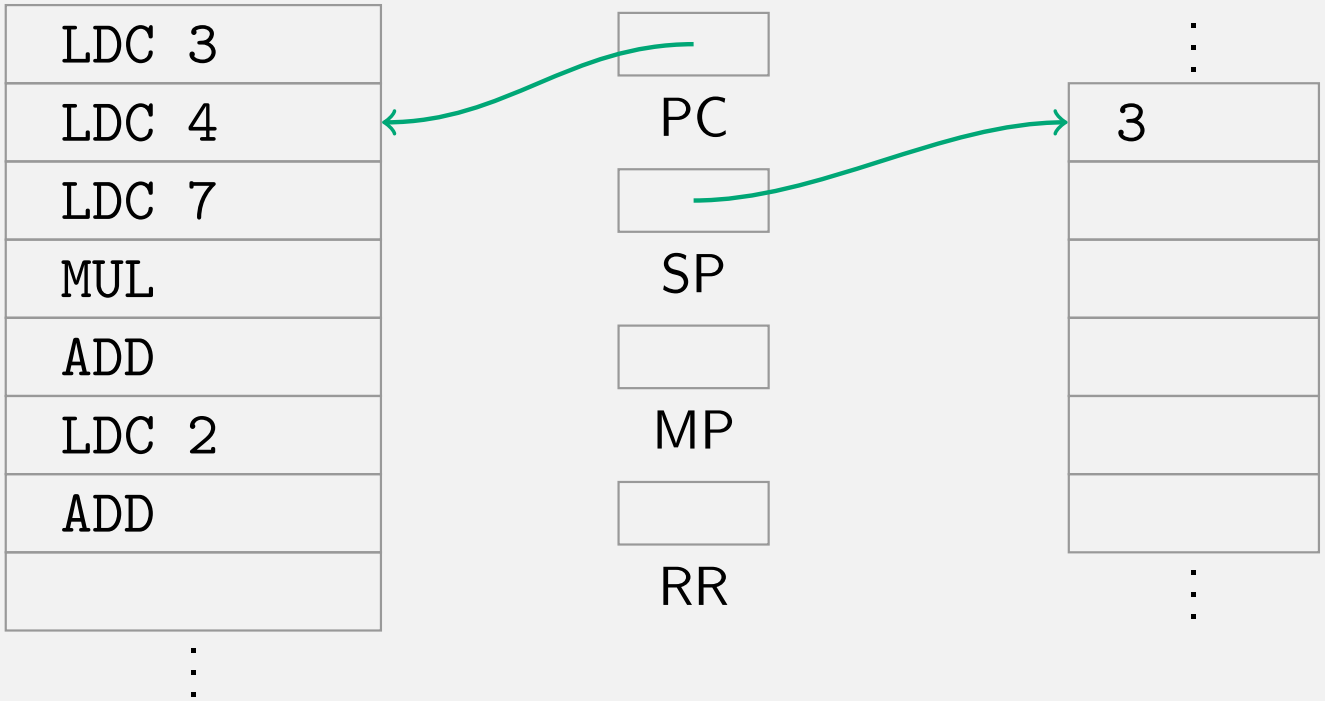
Code

```
LDC 3
LDC 4
LDC 7
MUL
ADD
LDC 2
ADD
```

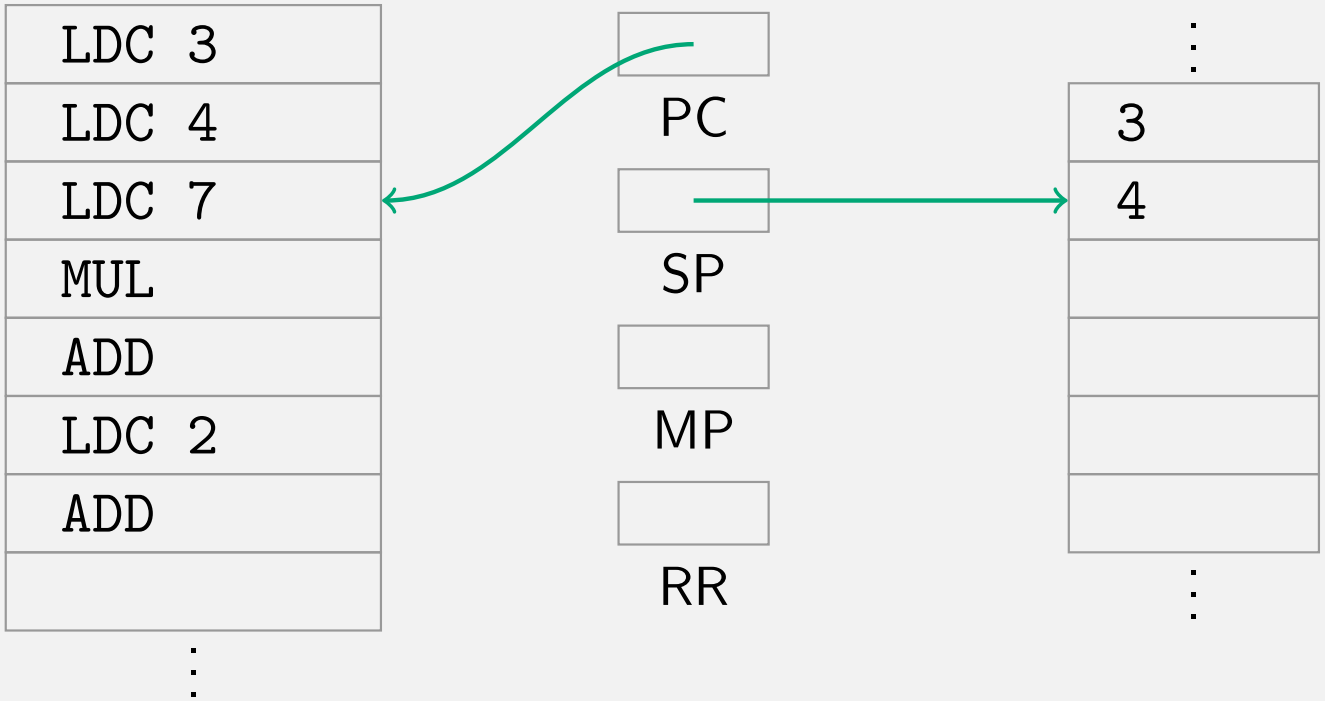
Arithmetic expression example



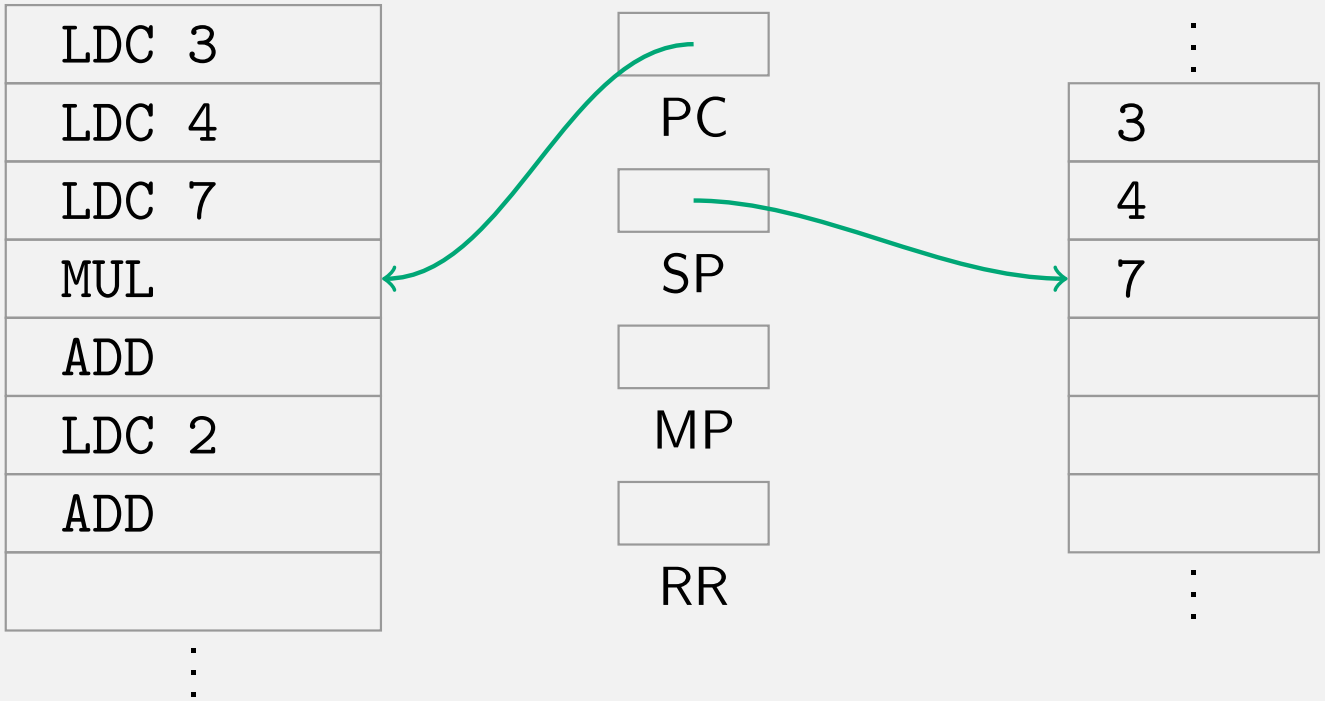
Arithmetic expression example



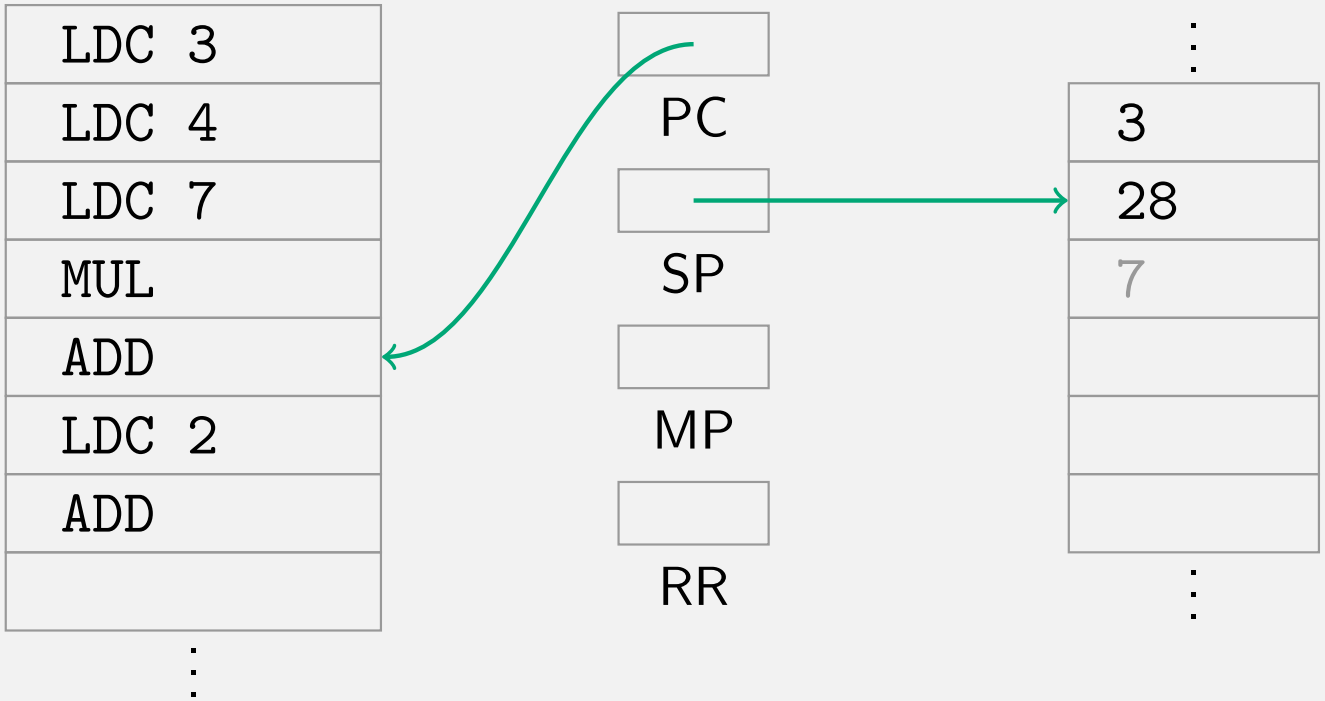
Arithmetic expression example



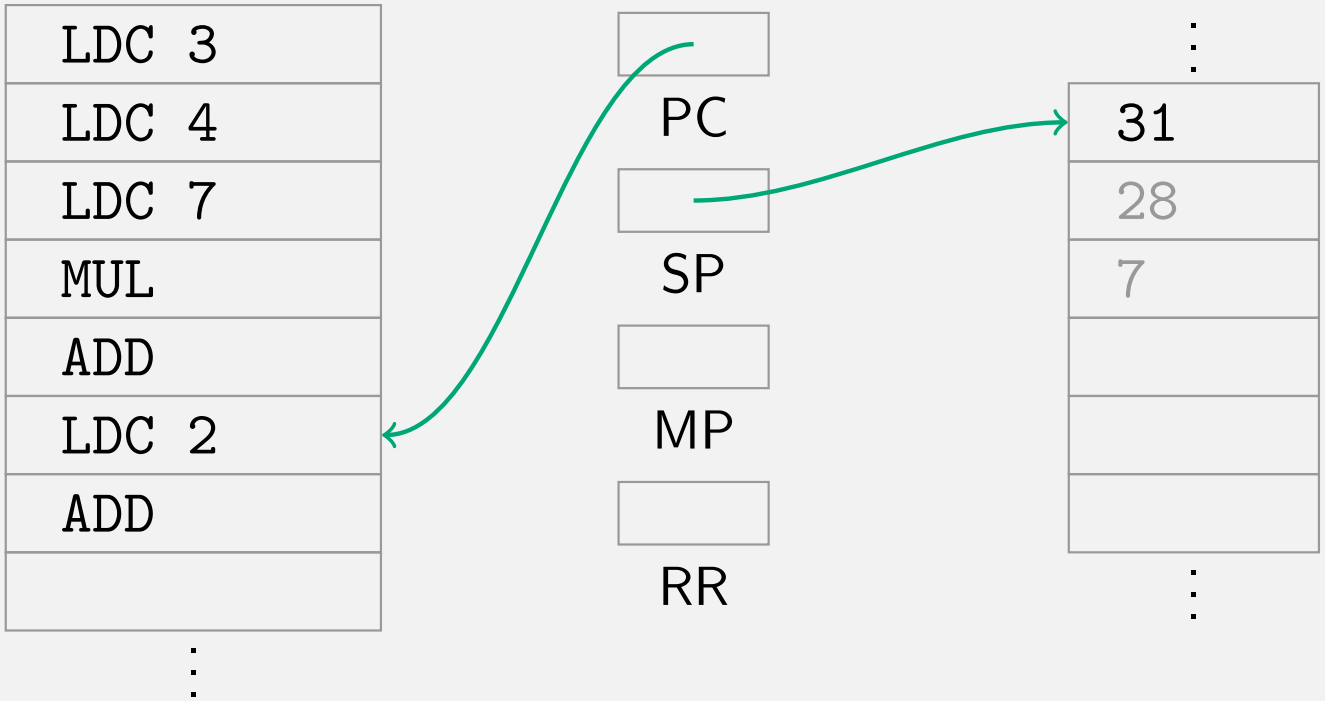
Arithmetic expression example



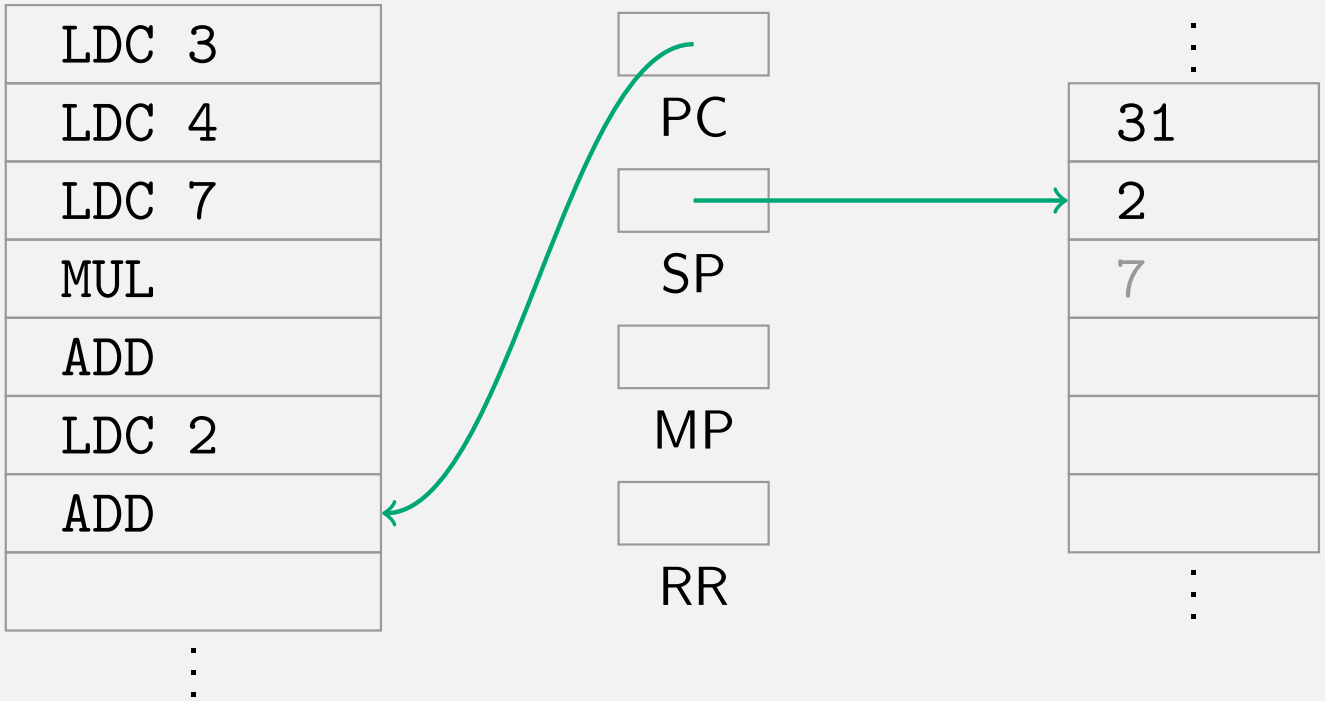
Arithmetic expression example



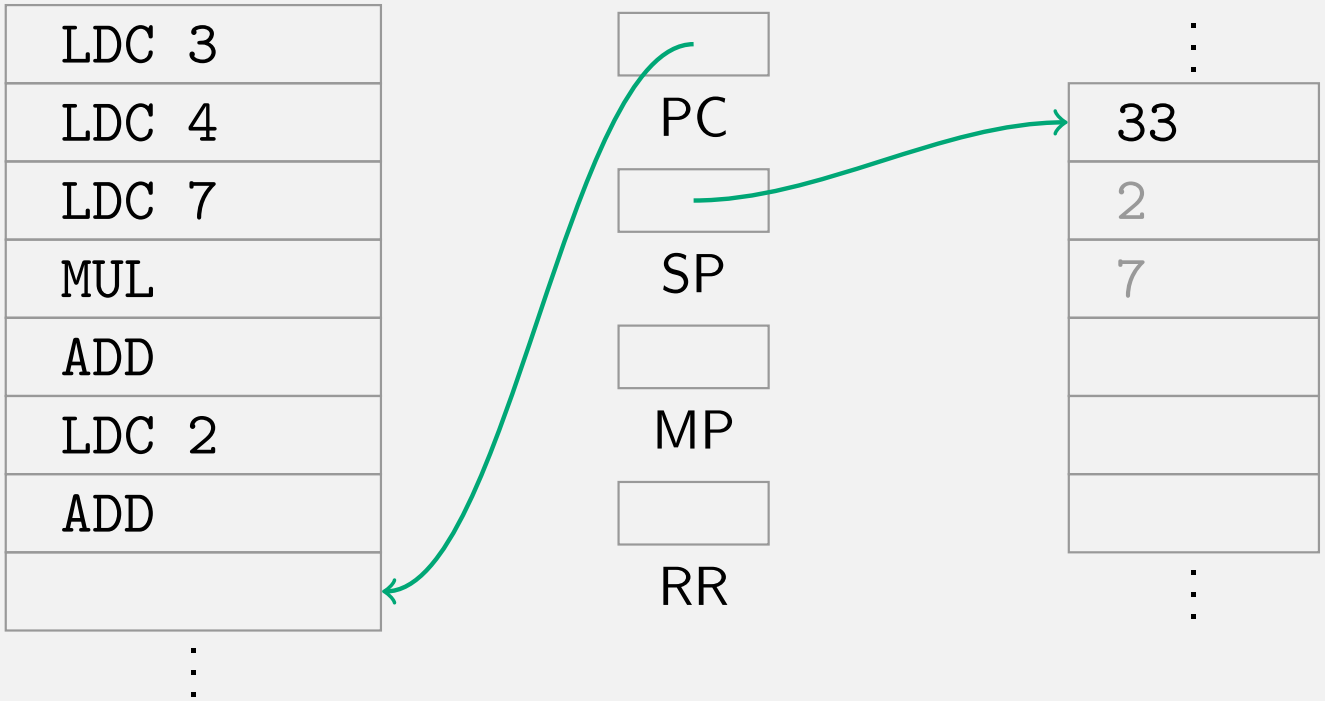
Arithmetic expression example



Arithmetic expression example



Arithmetic expression example



Representing code in Haskell

```
type Code = [Instr]

data Instr = LDC Int
           | LDL Int
           | ADD
           | NEG
           | EQ
           | ...

codeSize :: Code → Int
codeSize = sum . map instrSize

instrSize :: Instr → Int
instrSize (LDC n) = 2
instrSize ADD     = 1
...
```

Translating expressions

```
data Expr = Num Int
          | Add Expr Expr
          | Mul Expr Expr
          | Neg Expr
          | ...

code :: Expr → Code
code (Num n)     = [LDC n]
code (Add e1 e2) = code e1 ++ code e2 ++ [ADD]
code (Mul e1 e2) = code e1 ++ code e2 ++ [MUL]
code (Neg e)     = code e ++ [NEG]
```

Translating expressions

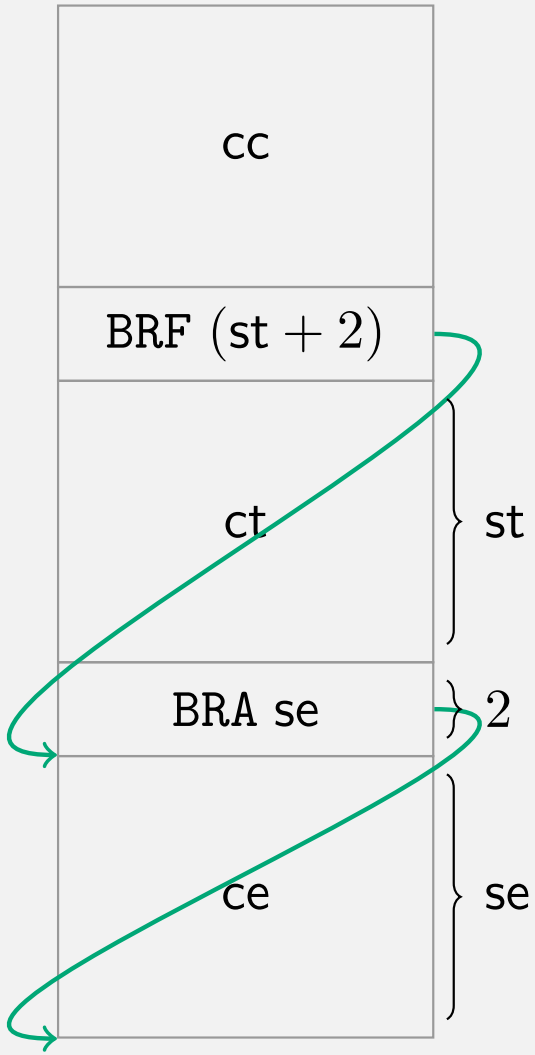
```
data Expr = Num Int
          | Add Expr Expr
          | Mul Expr Expr
          | Neg Expr
          | ...

code :: Expr → Code
code (Num n)     = [LDC n]
code (Add e1 e2) = code e1 ++ code e2 ++ [ADD]
code (Mul e1 e2) = code e1 ++ code e2 ++ [MUL]
code (Neg e)     = code e ++ [NEG]
```

We can also write this as a fold.

Translating conditional expressions

```
data Expr = ...
          | If Expr Expr Expr

code :: Expr → Code

...

code (If c t e) = cc          ++
                  [BRF (st + 2)] ++
                  ct          ++
                  [BRA se] ++
                  ce

  where cc = code c
        ct = code t
        ce = code e
        st = codeSize ct
        se = codeSize ce
```

Translating conditional expressions – contd.



Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Algebra for code generation

```
data Expr =
    Num Int
  | Add  Expr Expr
  | Neg  Expr
  | Eq   Expr Expr
  | If   Expr Expr Expr
```

```
code x = foldExpr codeAlg x
  where
  codeAlg :: ExprAlg          Code
  codeAlg =
    (λn      →
    ,λl r    →
    ,λl      →
    ,λl r    →
    ,λc l r  →

    )
```

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Algebra for code generation

```
data Expr =
    Num Int
  | Add  Expr Expr
  | Neg  Expr
  | Eq   Expr Expr
  | If   Expr Expr Expr
```

```
code x = foldExpr codeAlg x
  where
  codeAlg :: ExprAlg                Code
  codeAlg =
    (λn      →          [LDC n]
    ,λl r    →          l  ++ r  ++ [ADD]
    ,λl      →          l  ++ [NEG]
    ,λl r    →          l  ++ r  ++ [EQ]
    ,λc l r  →
                        )
```

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Algebra for code generation

```
data Expr =
    Num Int
  | Add  Expr Expr
  | Neg  Expr
  | Eq   Expr Expr
  | If   Expr Expr Expr
```

```
code x = foldExpr codeAlg x
  where
  codeAlg :: ExprAlg                Code
  codeAlg =
    (λn      →          [LDC n]
    ,λl r    →          l  ++ r  ++ [ADD]
    ,λl      →          l  ++ [NEG]
    ,λl r    →          l  ++ r  ++ [EQ]
    ,λc l r  →
```

```
          c  ++ [BRF (sl + 2)] ++
          l  ++ [BRA sr] ++ r

    )
```

Algebra for code generation

```
data Expr =
    Num Int
  | Add  Expr Expr
  | Neg  Expr
  | Eq   Expr Expr
  | If   Expr Expr Expr
```

```
code x = foldExpr codeAlg x
  where
  codeAlg :: ExprAlg                Code
  codeAlg =
    (λn      →          [LDC n]
    ,λl r    →          l  ++ r  ++ [ADD]
    ,λl      →          l  ++ [NEG]
    ,λl r    →          l  ++ r  ++ [EQ]
    ,λc l r  →
      let  sl = codeSize (l  )
           sr = codeSize (r  )
      in   c  ++ [BRF (sl + 2)] ++
           l  ++ [BRA sr] ++ r


    )
```

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Algebra for code generation

```
data Expr =
    Num Int
  | Add  Expr Expr
  | Neg  Expr
  | Eq   Expr Expr
  | If   Expr Expr Expr

  | Var  String
  | Let  String Expr Expr
```

```
code x = foldExpr codeAlg x
  where
  codeAlg :: ExprAlg                Code
  codeAlg =
    (λn      →          [LDC n]
    ,λl r    →          l  ++ r  ++ [ADD]
    ,λl      →          l  ++ [NEG]
    ,λl r    →          l  ++ r  ++ [EQ]
    ,λc l r  →
      let  sl = codeSize (l  )
           sr = codeSize (r  )
      in   c  ++ [BRF (sl + 2)] ++
           l  ++ [BRA sr] ++ r
    ,λs      →
    ,λs d b →

    )
```

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Algebra for code generation

```
data Expr =
    Num Int
  | Add  Expr Expr
  | Neg  Expr
  | Eq   Expr Expr
  | If   Expr Expr Expr

  | Var  String
  | Let  String Expr Expr
```

```
code x = foldExpr codeAlg x
  where
  codeAlg :: ExprAlg                Code
  codeAlg =
    (λn      →          [LDC n]
    ,λl r    →          l  ++ r  ++ [ADD]
    ,λl      →          l  ++ [NEG]
    ,λl r    →          l  ++ r  ++ [EQ]
    ,λc l r  →
      let  sl = codeSize (l  )
           sr = codeSize (r  )
      in   c  ++ [BRF (sl + 2)] ++
           l  ++ [BRA sr] ++ r
    ,λs      →          [LDL ??]
    ,λs d b →

    )
```

Algebra for code generation

```
data Expr =
    Num Int
  | Add  Expr Expr
  | Neg  Expr
  | Eq   Expr Expr
  | If   Expr Expr Expr

  | Var  String
  | Let  String Expr Expr
```

```
code x = foldExpr codeAlg x
  where
  codeAlg :: ExprAlg (Env → Code)
  codeAlg =
    (λn      →          [LDC n]
    ,λl r    →          l  ++ r  ++ [ADD]
    ,λl      →          l  ++ [NEG]
    ,λl r    →          l  ++ r  ++ [EQ]
    ,λc l r  →
      let  sl = codeSize (l  )
           sr = codeSize (r  )
      in   c  ++ [BRF (sl + 2)] ++
           l  ++ [BRA sr] ++ r
    ,λs      → λe  →  [LDL (e ! s)]
    ,λs d b → λe  →  d e ++ [STL (size e)]
                          ++ b (insert s (size e) e)
    )
```

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Algebra for code generation

```
data Expr =
    Num Int
  | Add  Expr Expr
  | Neg  Expr
  | Eq   Expr Expr
  | If   Expr Expr Expr

  | Var  String
  | Let  String Expr Expr
```

```
code x = foldExpr codeAlg x empty
  where
  codeAlg :: ExprAlg (Env → Code)
  codeAlg =
    (λn     → λe → [LDC n]
    ,λl r   → λe → l e ++ r e ++ [ADD]
    ,λl     → λe → l e ++ [NEG]
    ,λl r   → λe → l e ++ r e ++ [EQ]
    ,λc l r → λe →
      let sl = codeSize (l e)
          sr = codeSize (r e)
      in  c e ++ [BRF (sl + 2)] ++
          l e ++ [BRA sr] ++ r e
    ,λs     → λe → [LDL (e ! s)]
    ,λs d b → λe → d e ++ [STL (size e)]
                   ++ b (insert s (size e) e)
    )
```

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Expressions vs. statements

We extend our language with statements:

```
data Stmt =
    Assign String Expr
  | If     Expr Stmt Stmt
  | While  Expr Stmt
  | Call   String [Expr]
```

Expressions vs. statements

We extend our language with statements:

```
data Stmt =
    Assign String Expr
  | If     Expr Stmt Stmt
  | While  Expr Stmt
  | Call   String [Expr]
```

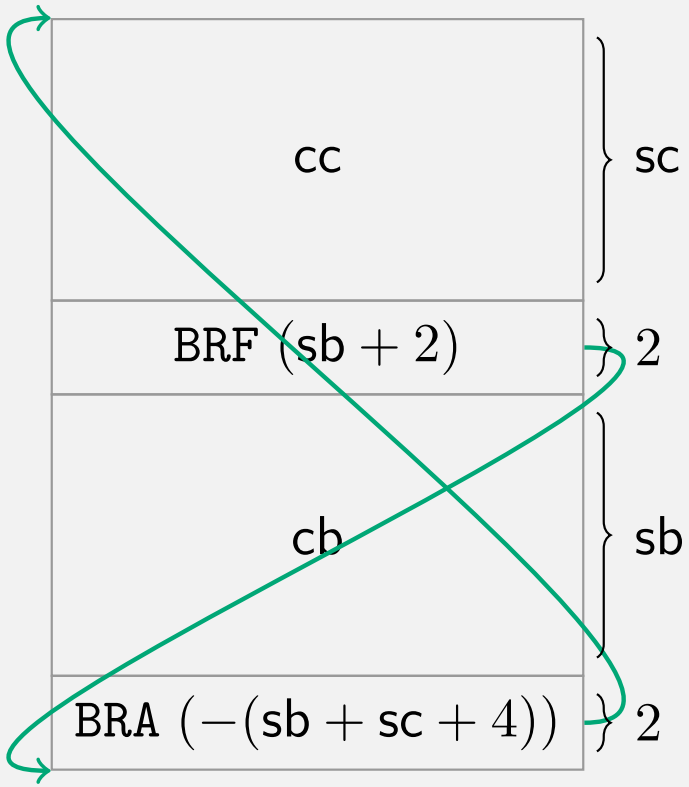
For many languages, the following invariants hold:

- Expressions always leave a single result on the stack after evaluation.
- Statements do not leave a result on the stack after evaluation.

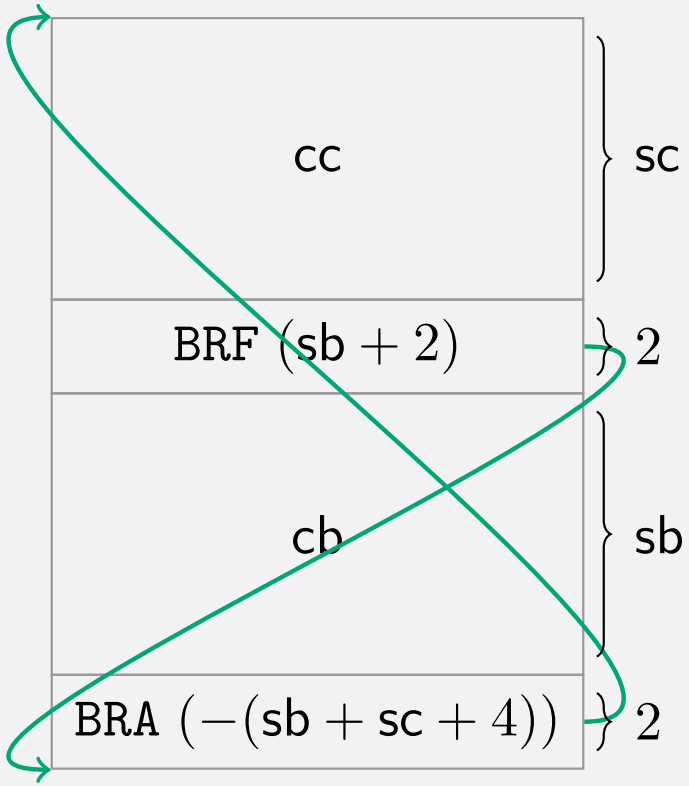
Translating while loops

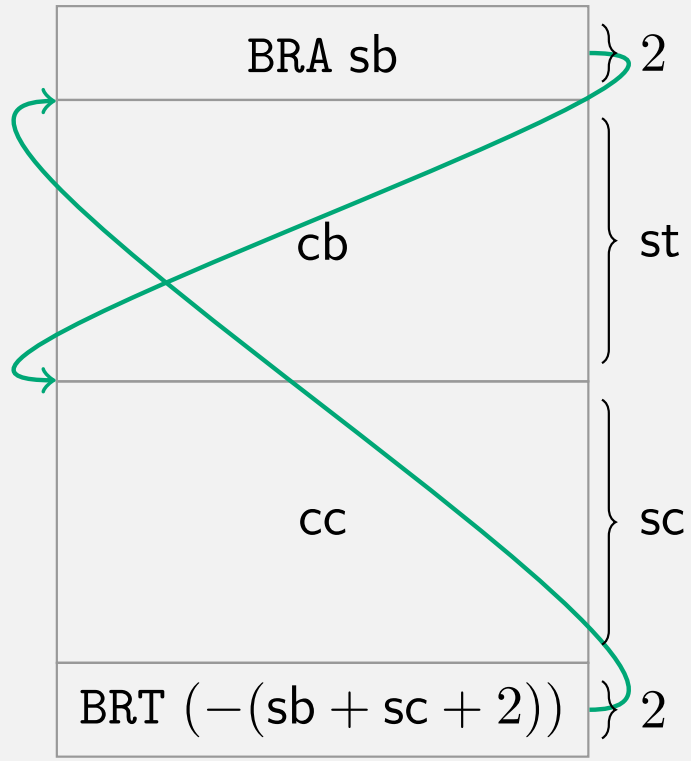
```
data Stmt = ...
          | While Expr Stmt

code :: Stmt → Code
...
code (While c b) = cc                    ++
                   [BRF (sb + 2)]        ++
                   cb                    ++
                   [BRA (−(sb + sc + 4))]
  where cc = code c
        cb = code b
        sc = codeSize cc
        sb = codeSize cb
```

Translating while loops – contd.



Translating while loops – contd.





Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Translating while loops – contd.

```
data Stmt = ...
          | While Expr Stmt

code :: Stmt → Code
...
code (While c b) = [BRA sb] ++
                   cb       ++
                   cc       ++
                   [BRT (-(sb + sc + 2))]
  where cc = code c
        cb = code b
        sc = codeSize cc
        sb = codeSize cb
```

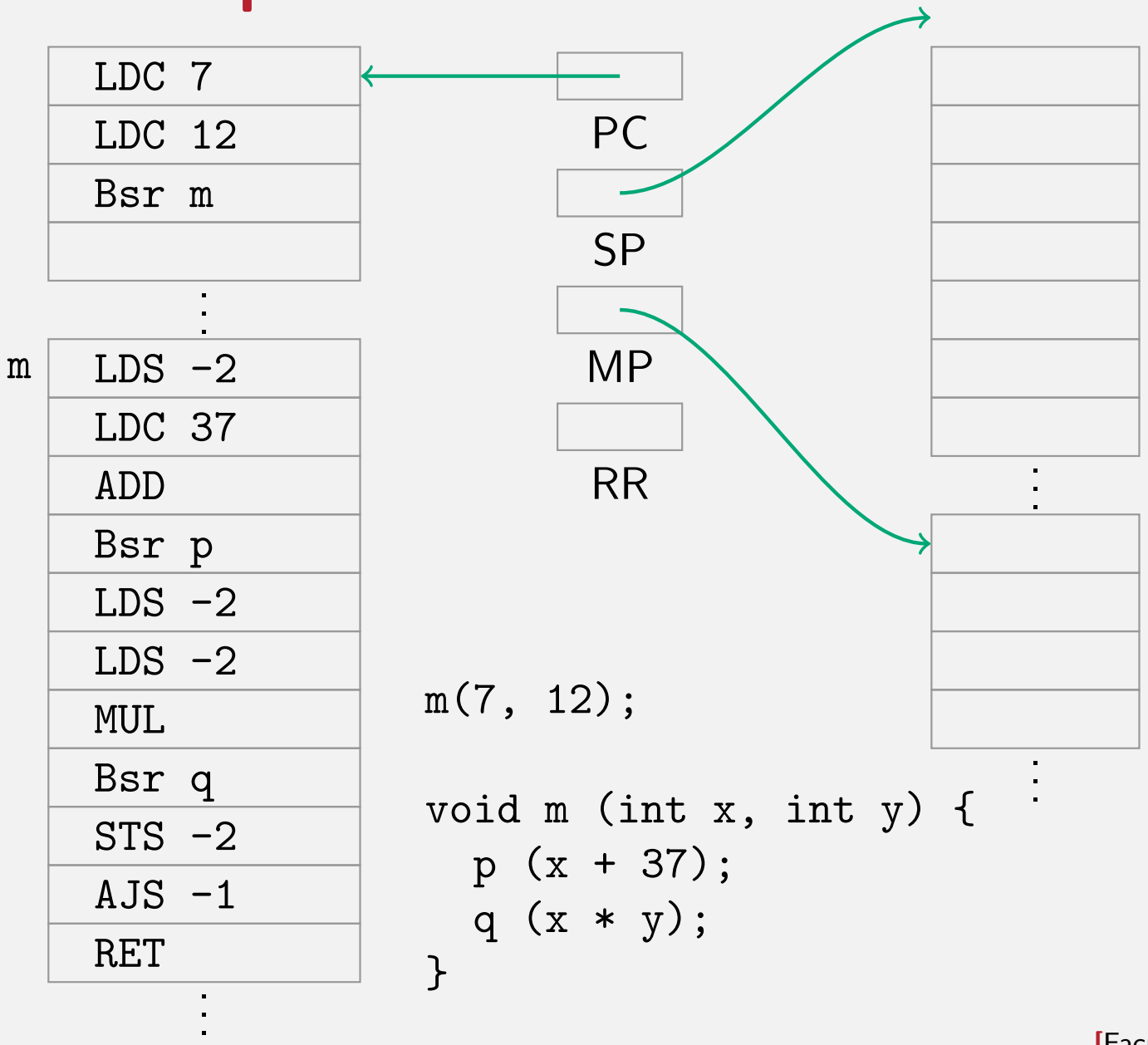
Algebra for code generation

```
data Stmt =
     Assign String Expr
   | If     Expr Stmt Stmt
   | While  Expr Stmt
   | Call   String [Expr]
```

```
code x = foldSE codeAlg x empty
  where
  codeAlg :: SEAlg (Env → Code) (Env → Code)
  codeAlg =
    (λs d   e → d e ++ [STL (e ! s)]
    , λc l r  e →
        let sl = codeSize (l e)
            sr = codeSize (r e)
        in  c e ++ [BRF (sl + 2)] ++
            l e ++ [BRA sr] ++ r e
    , λc b   e →
        let sc = codeSize (c e)
            sb = codeSize (b e)
        in  [BRA sb] ++ b e ++ c e ++
            [BRT (−(sb + sc + 2))]
    , λm ps e → concat [p e | p ← ps] ++ [Bsr m]
    , ...   -- components for Expr
    )
```

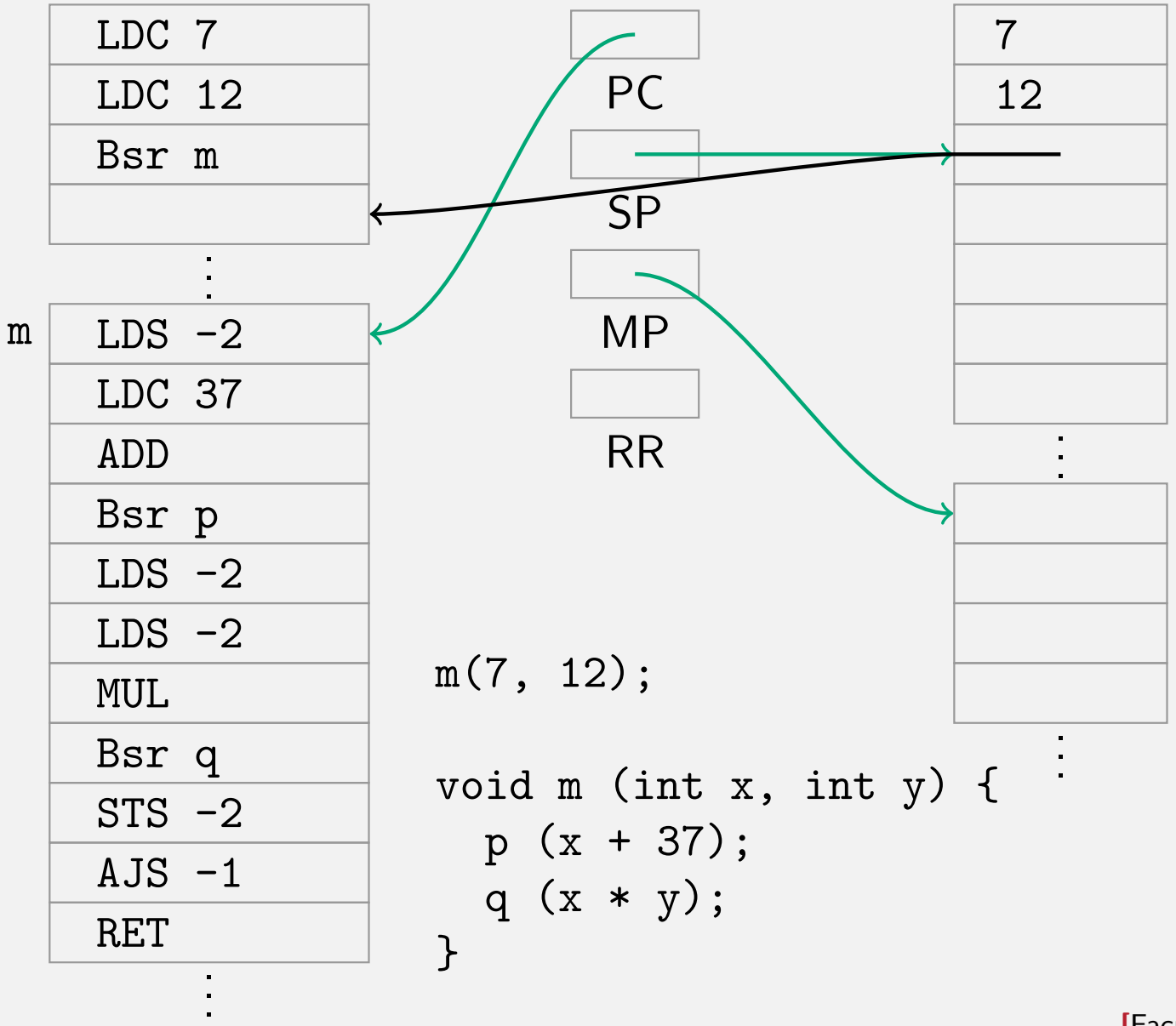
9.4 Functions / methods

Methods with parameters



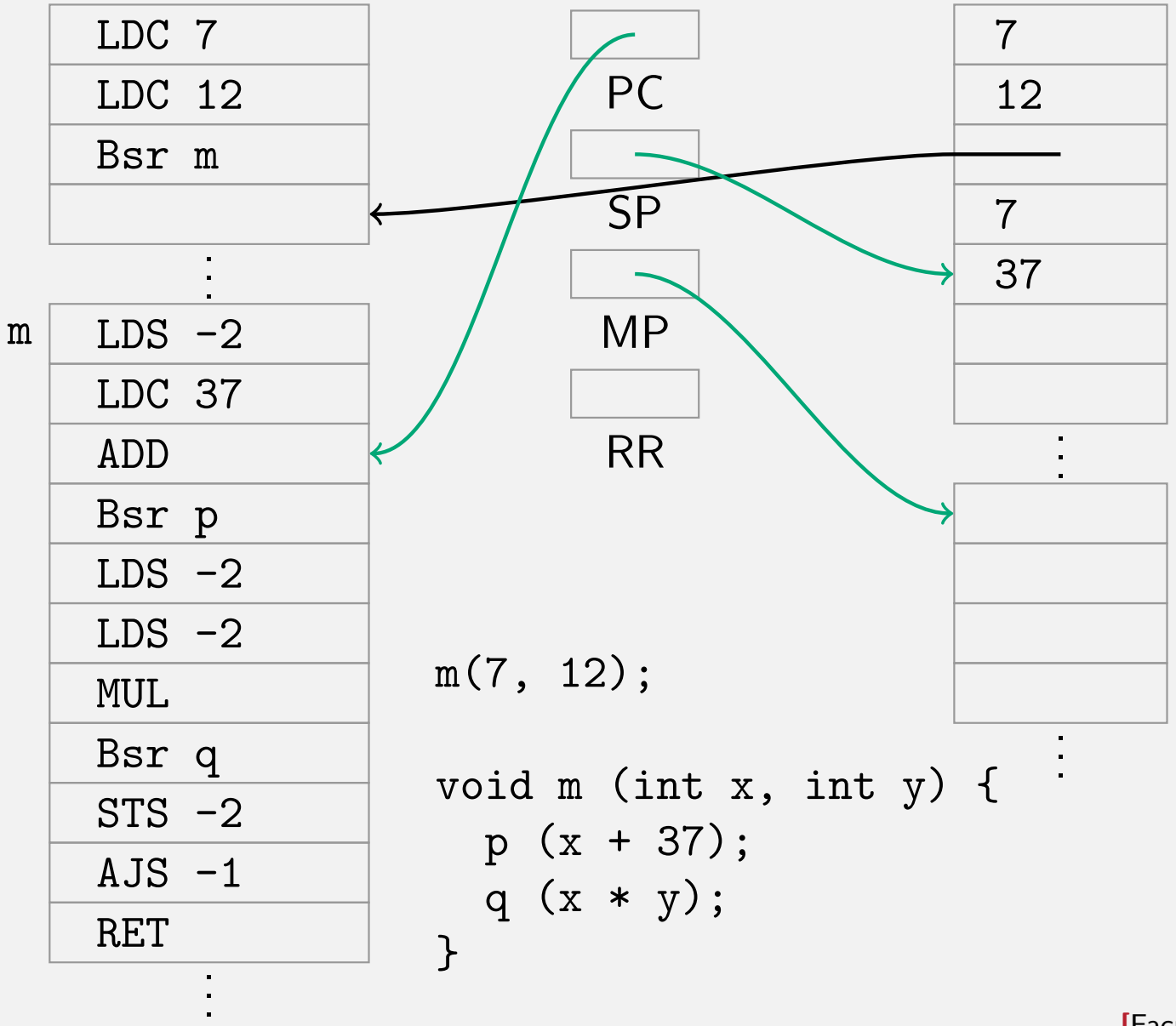

```
m(7, 12);

void m (int x, int y) {
  p (x + 37);
  q (x * y);
}
```

Methods with parameters




```
m(7, 12);

void m (int x, int y) {
  p (x + 37);
  q (x * y);
}
```

Methods with parameters

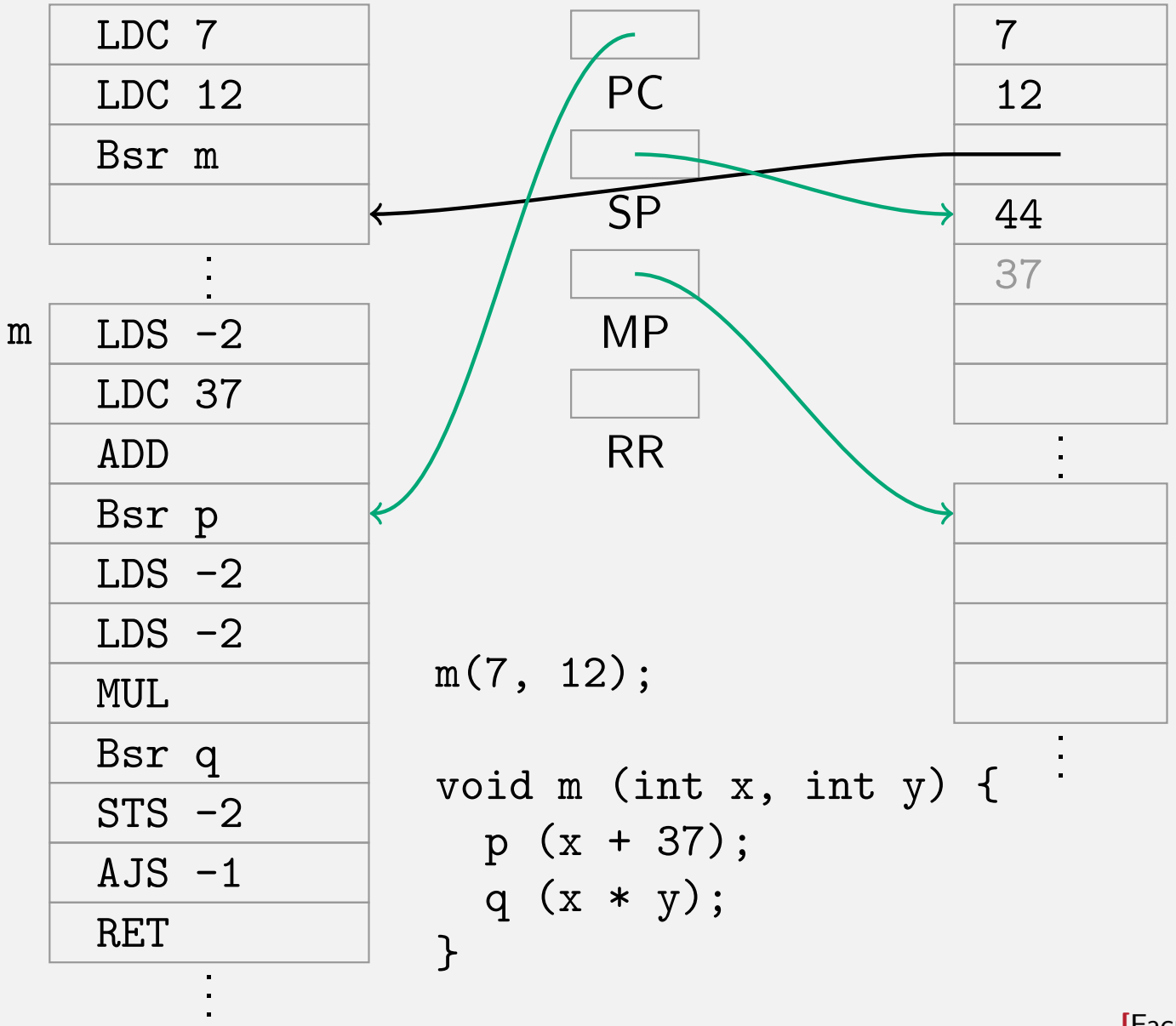



```
m(7, 12);

void m (int x, int y) {
  p (x + 37);
  q (x * y);
}
```

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Methods with parameters

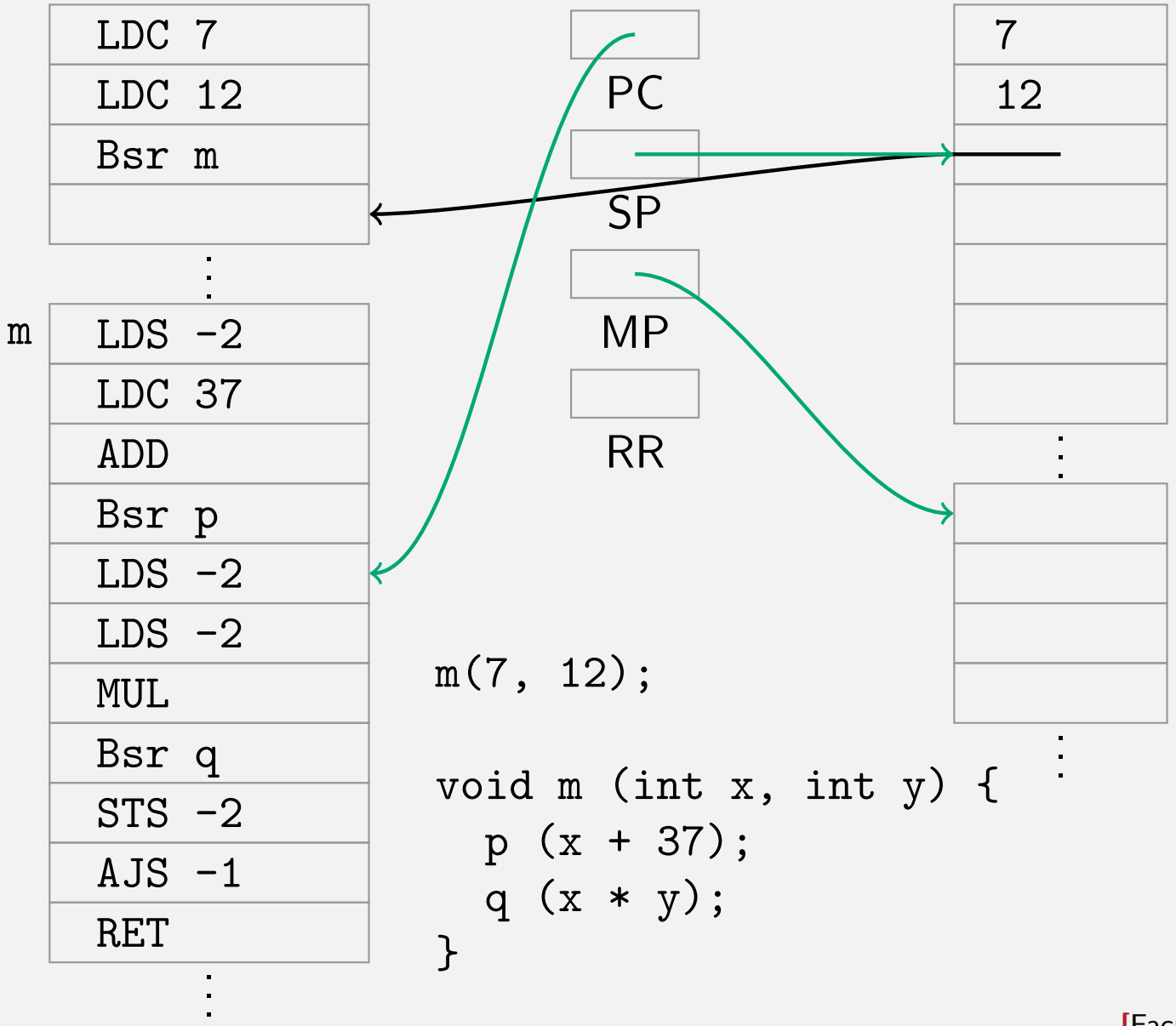



```
m(7, 12);

void m (int x, int y) {
  p (x + 37);
  q (x * y);
}
```

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Methods with parameters

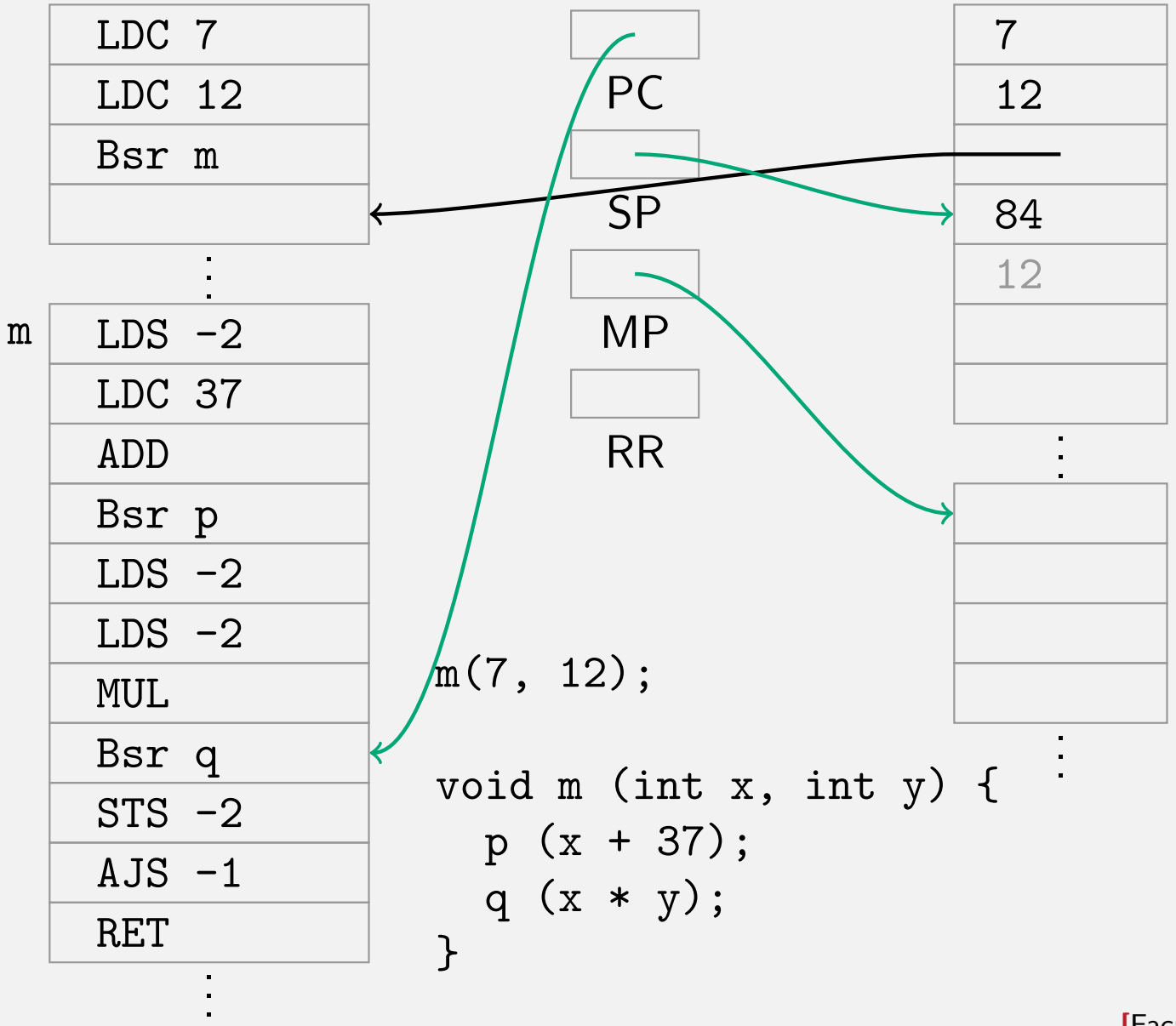



```
m(7, 12);

void m (int x, int y) {
  p (x + 37);
  q (x * y);
}
```

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

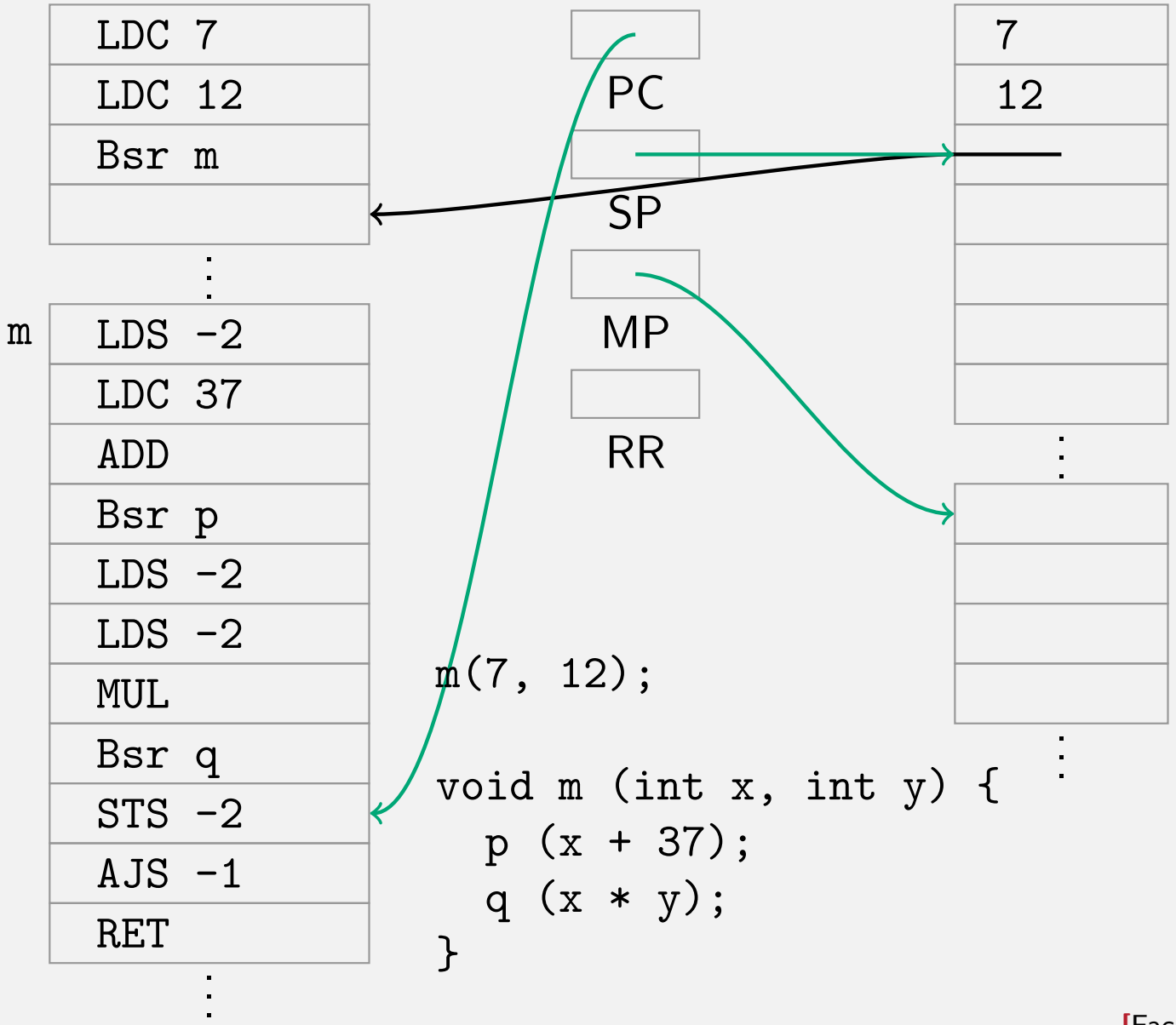
Methods with parameters



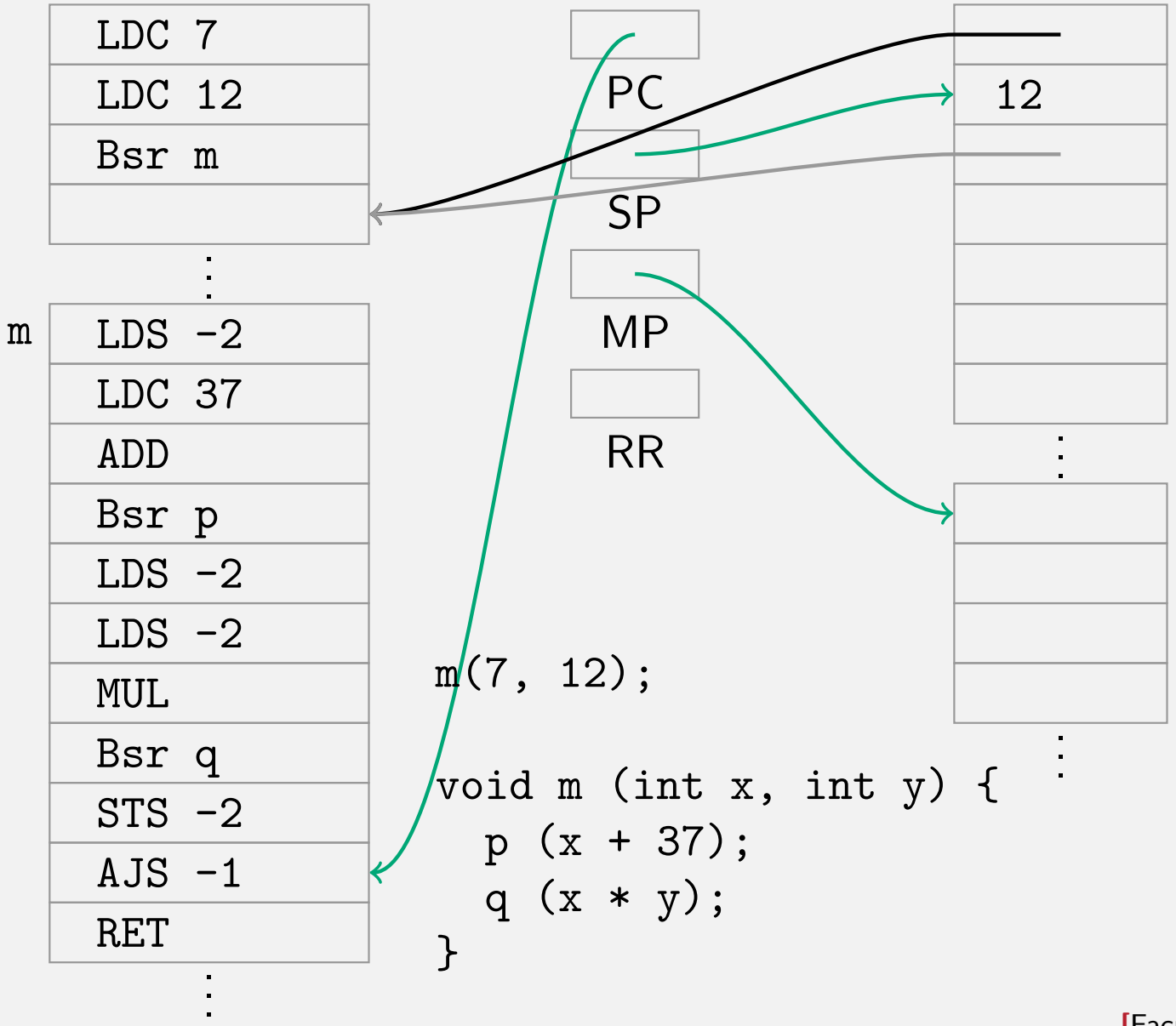
Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Methods with parameters



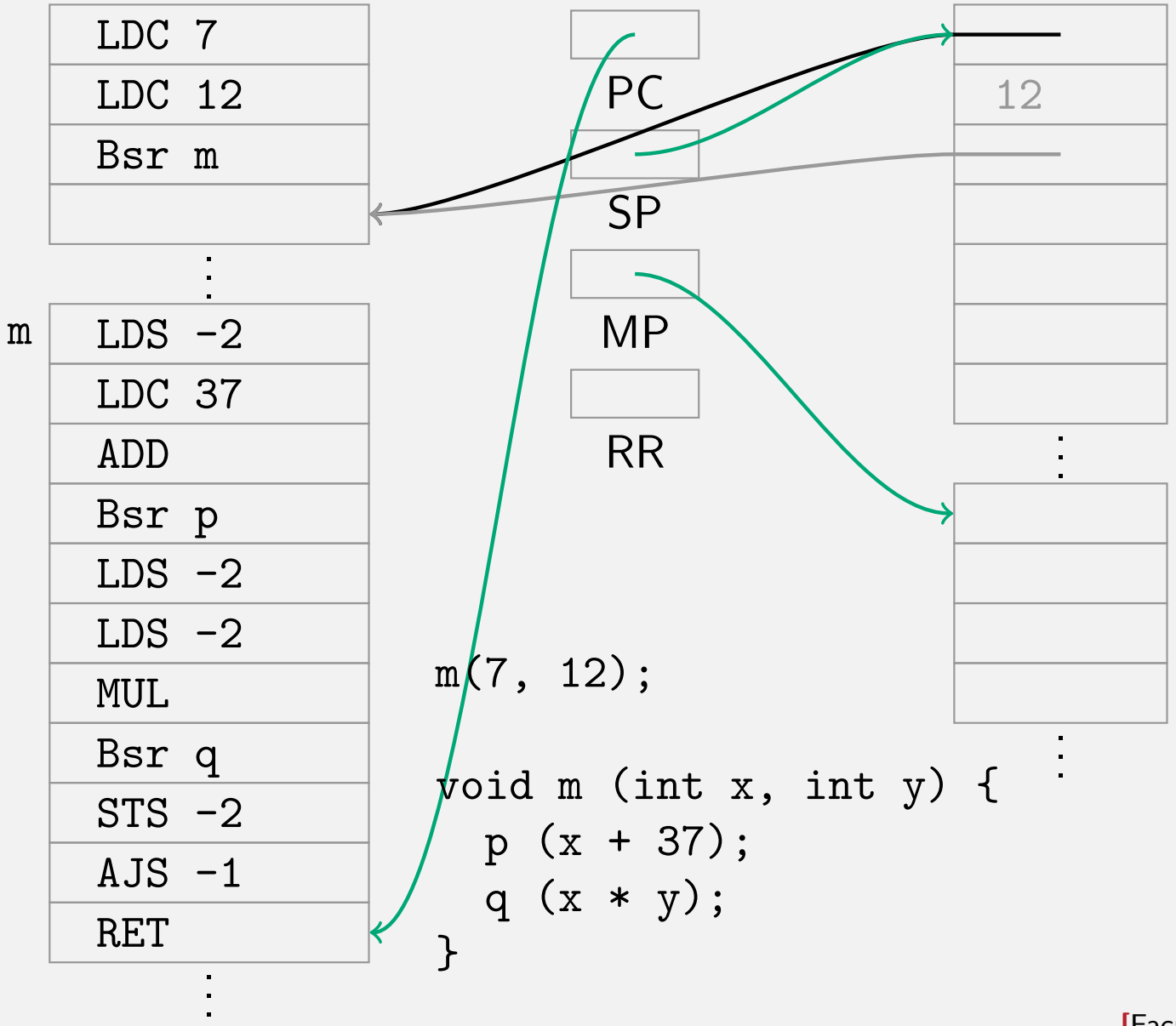
Methods with parameters



Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

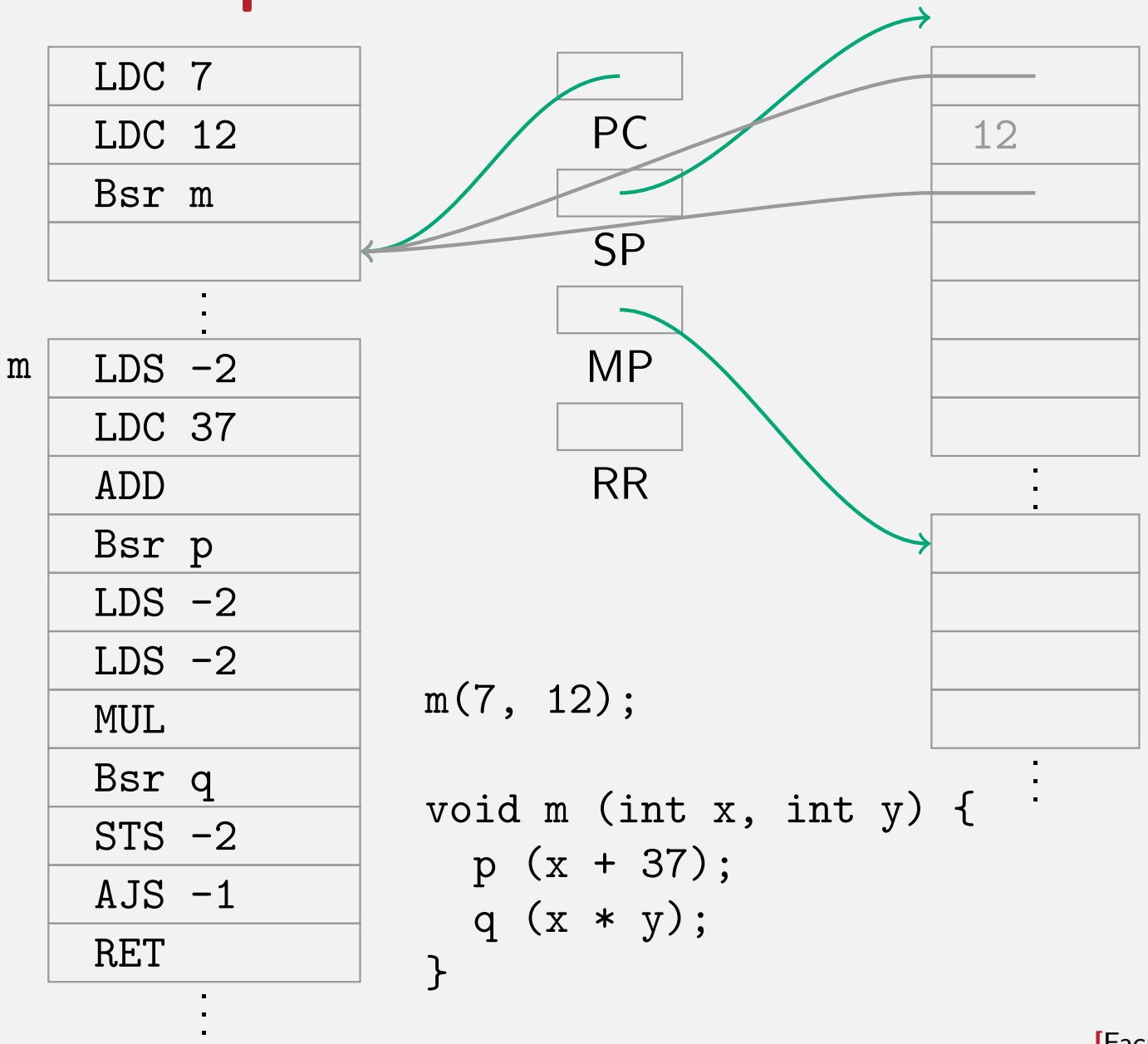
Methods with parameters



Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Methods with parameters




```
m(7, 12);

void m (int x, int y) {
  p (x + 37);
  q (x * y);
}
```

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Method translation

Method call

- Put parameters on the stack.
- Call `Bsr` with the method label.

Method translation

Method call

- Put parameters on the stack.
- Call `Bsr` with the method label.

Method definition

- Use parameters: from `LDS` $-(n+d)$ to `LDS` $-(1+d)$, where n is the number of parameters and d is your current offset (this becomes easier with the mark pointer).
- Clean up: `STS` $-n$ followed by `ADJ` $-(n-1)$.
- Return: `RET`

Method translation

Method call

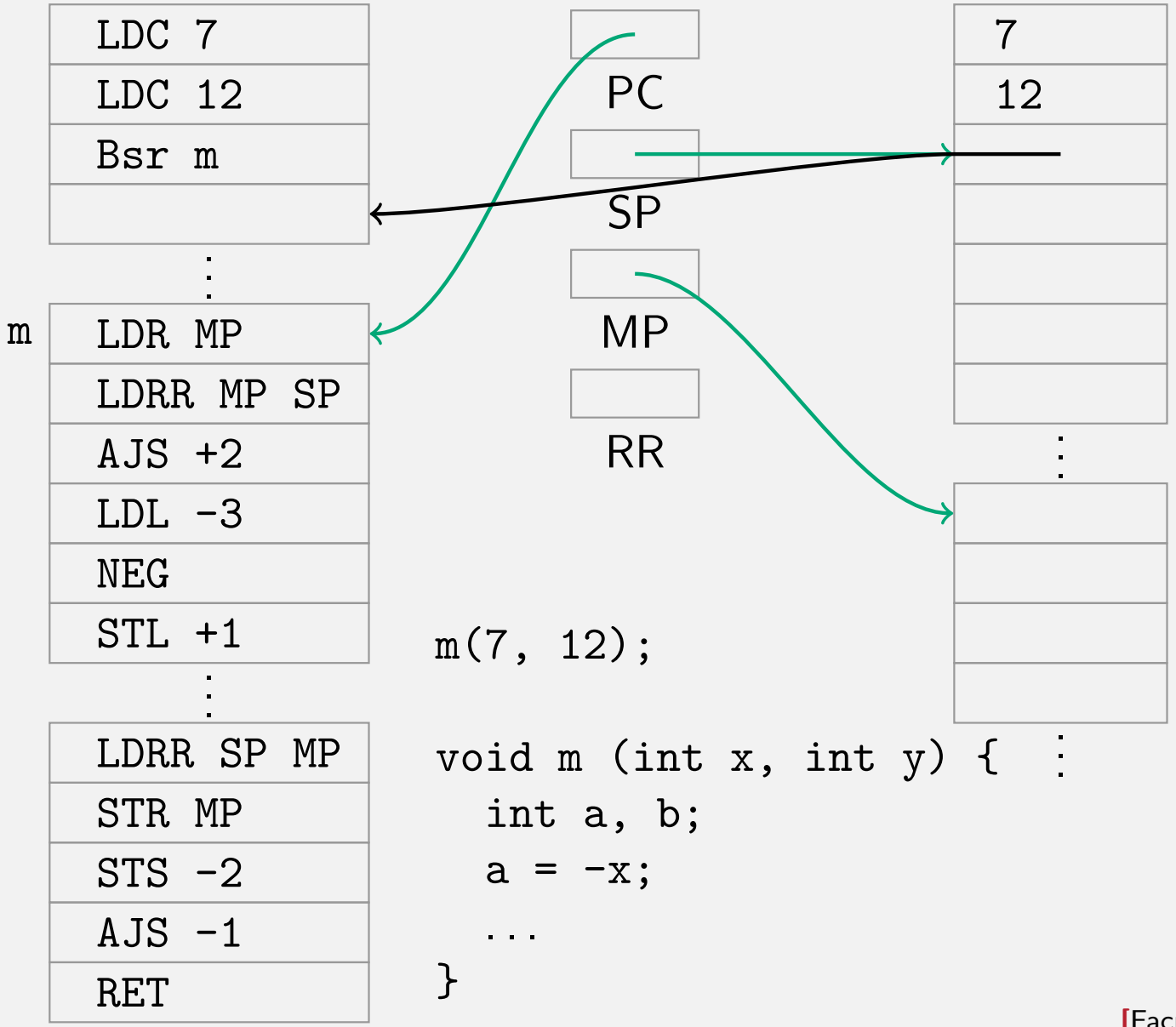
- Put parameters on the stack.
- Call `Bsr` with the method label.

Method definition

- Use parameters: from `LDS` $-(n+d)$ to `LDS` $-(1+d)$, where n is the number of parameters and d is your current offset (this becomes easier with the mark pointer).
- Clean up: `STS` $-n$ followed by `ADJ` $-(n-1)$.
- Return: `RET`

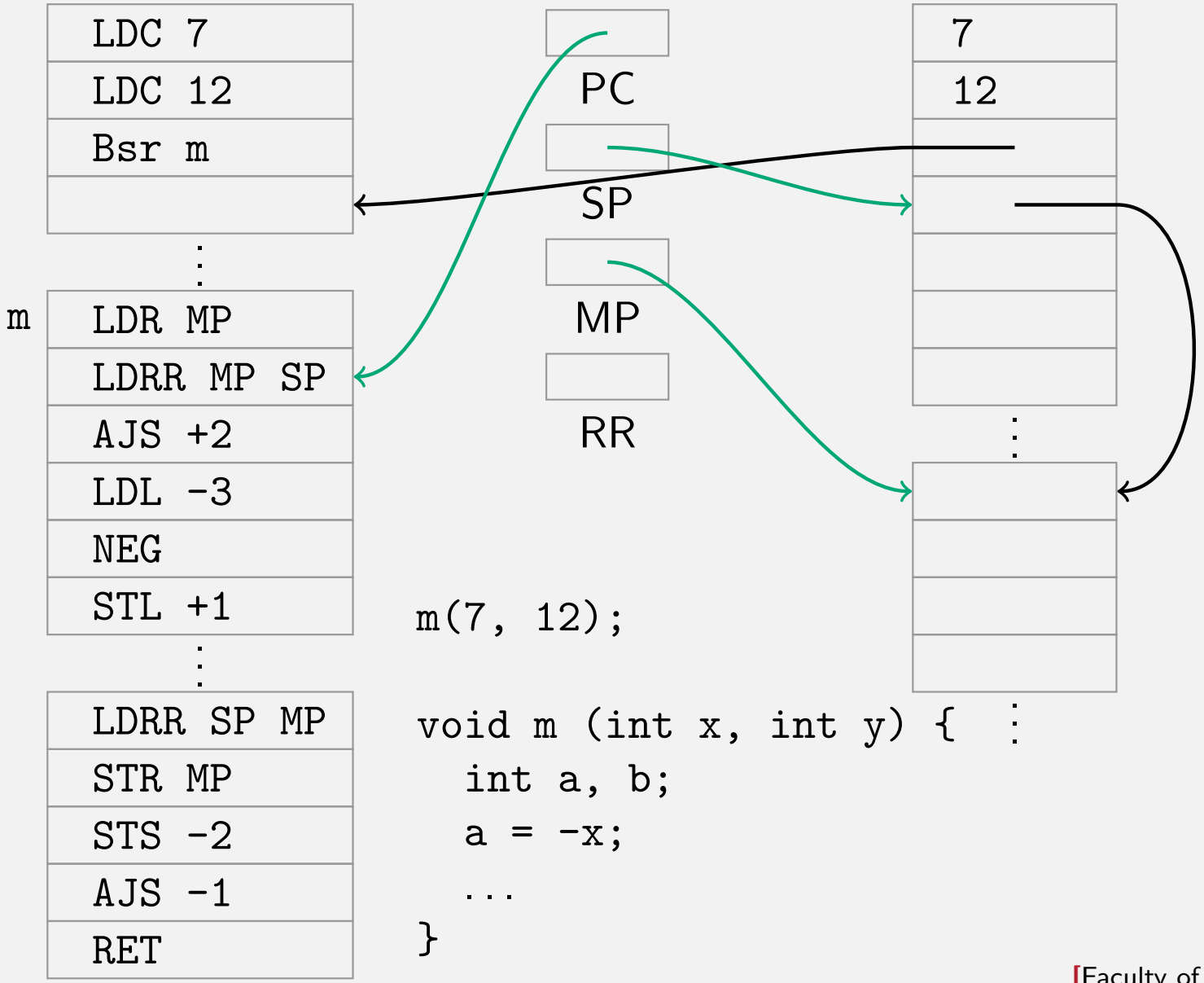
It is also possible, but less common, to let the caller clean up after a method call.

Methods with local variables




```
m(7, 12);

void m (int x, int y) {
  int a, b;
  a = -x;
  ...
}
```

Methods with local variables

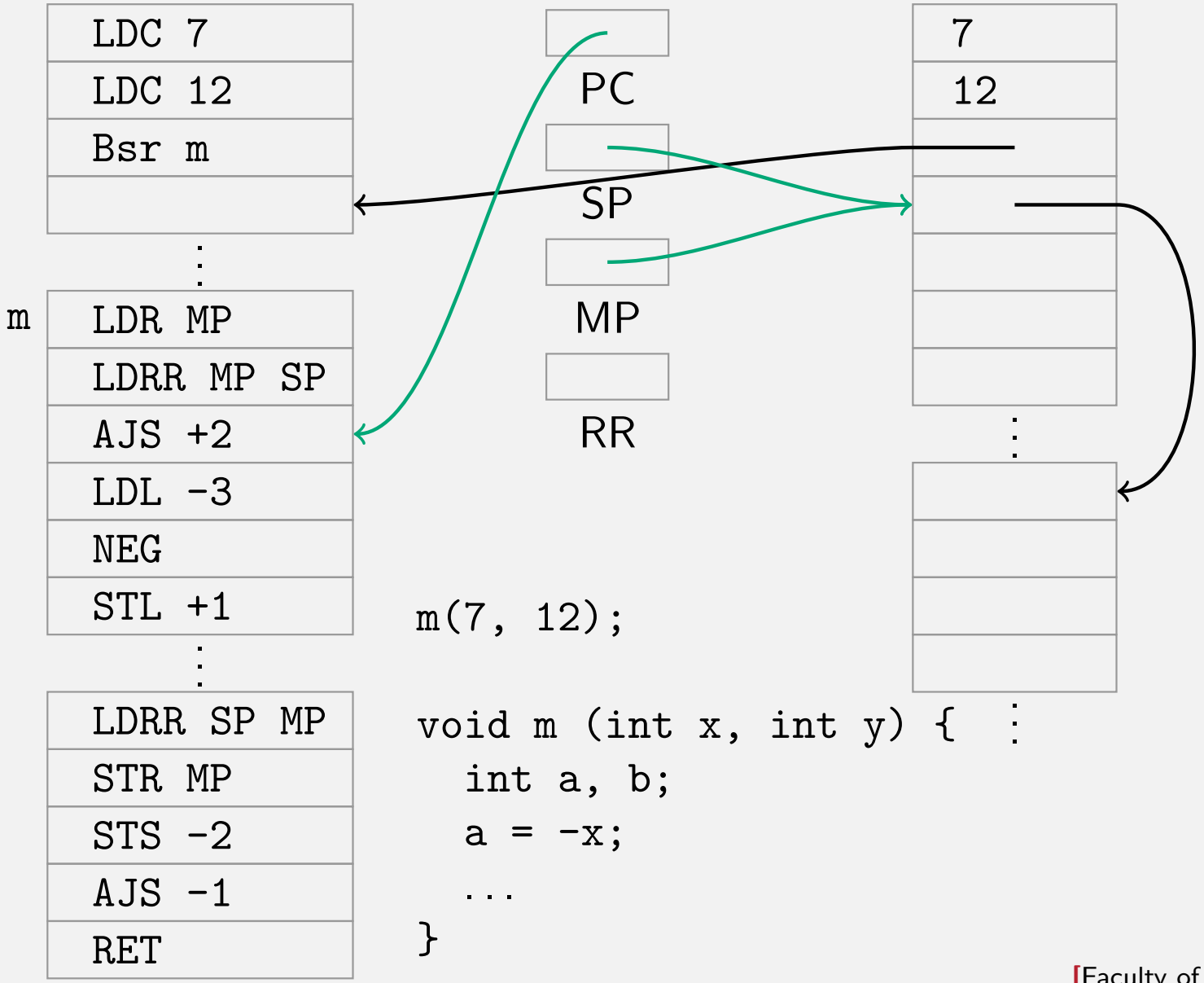



```
m(7, 12);

void m (int x, int y) {
  int a, b;
  a = -x;
  ...
}
```

Universiteit Utrecht

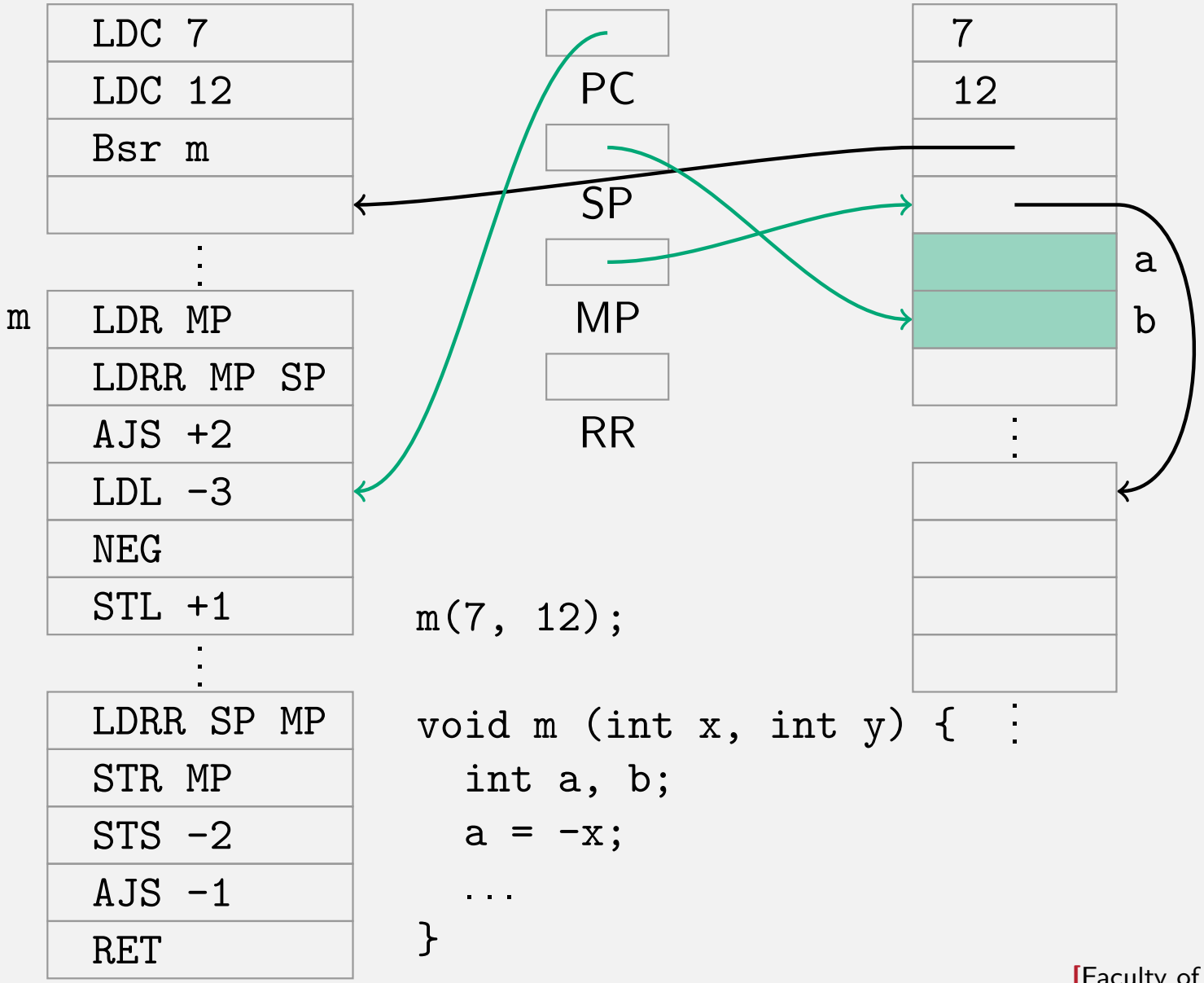
[Faculty of **Science**
Information and Computing Sciences]

Methods with local variables

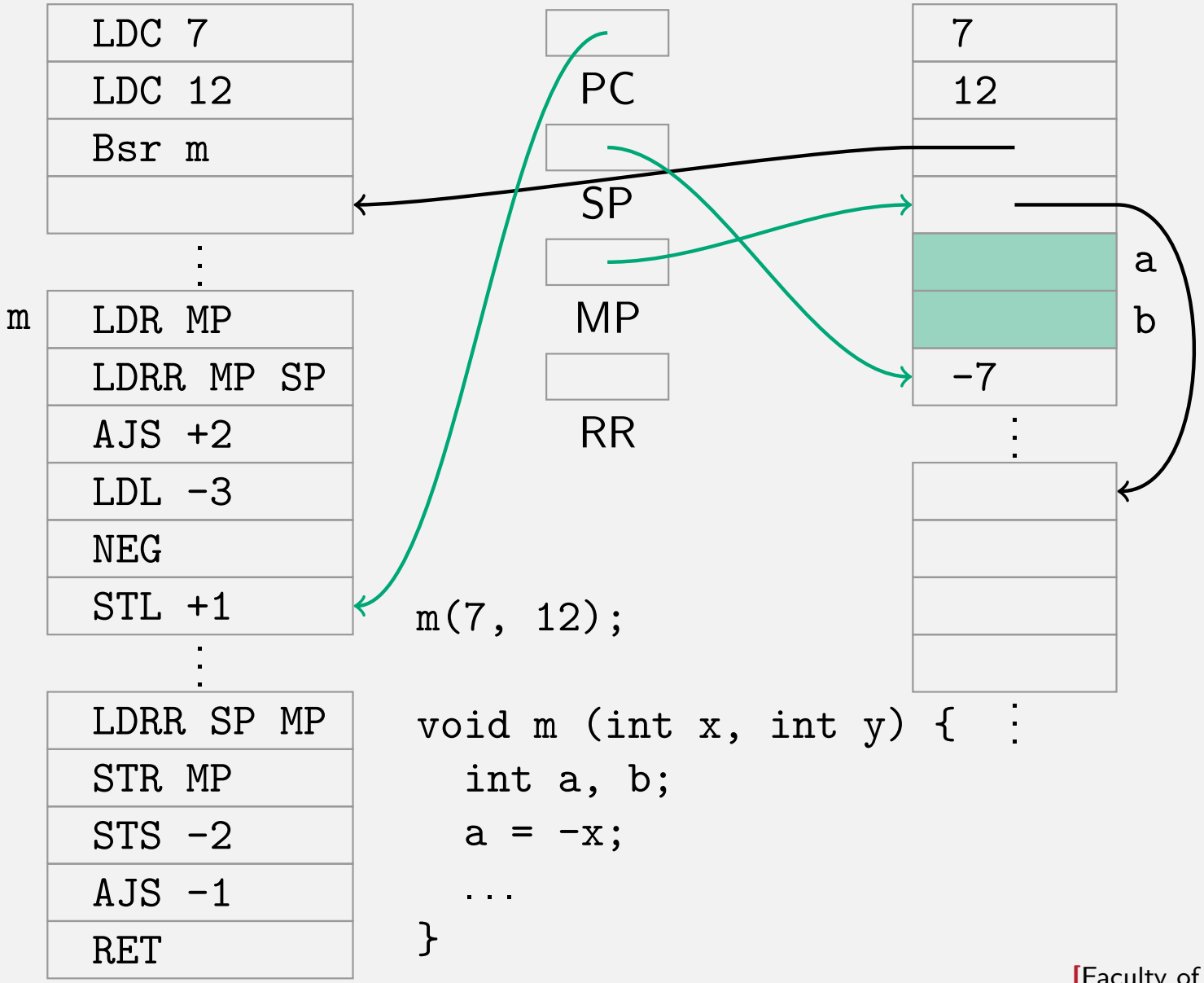



```
m(7, 12);

void m (int x, int y) {
  int a, b;
  a = -x;
  ...
}
```

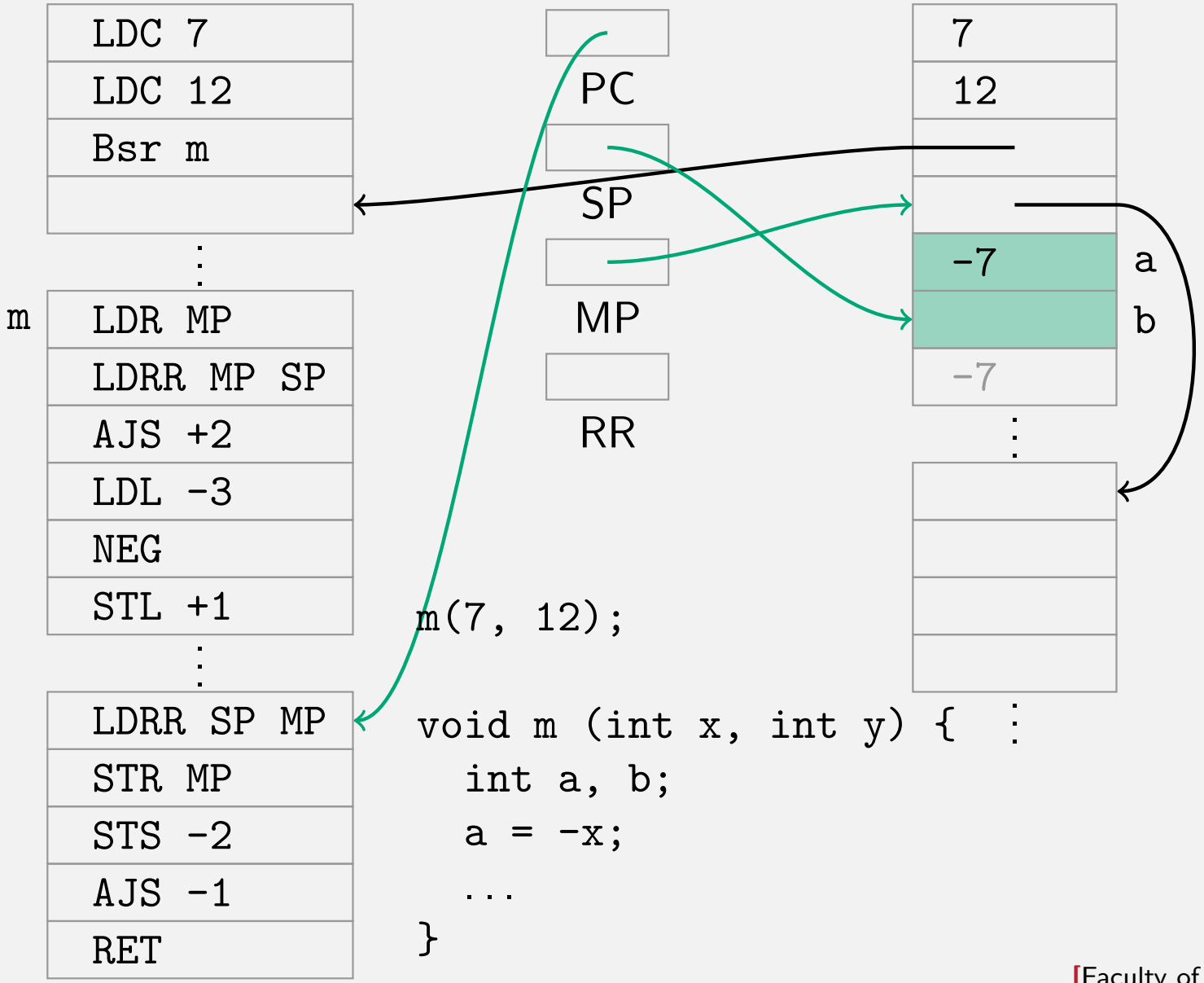
Methods with local variables



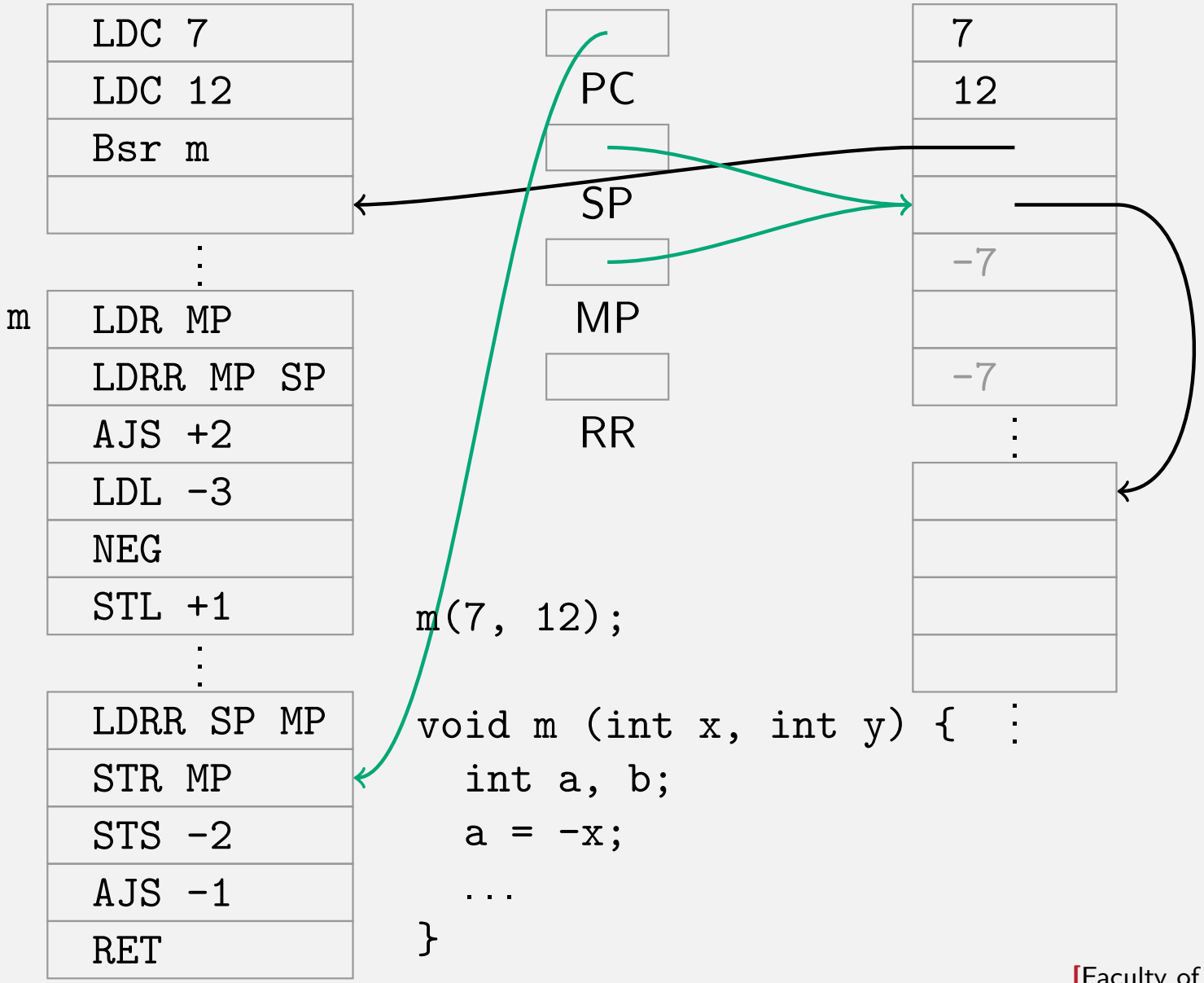
Methods with local variables



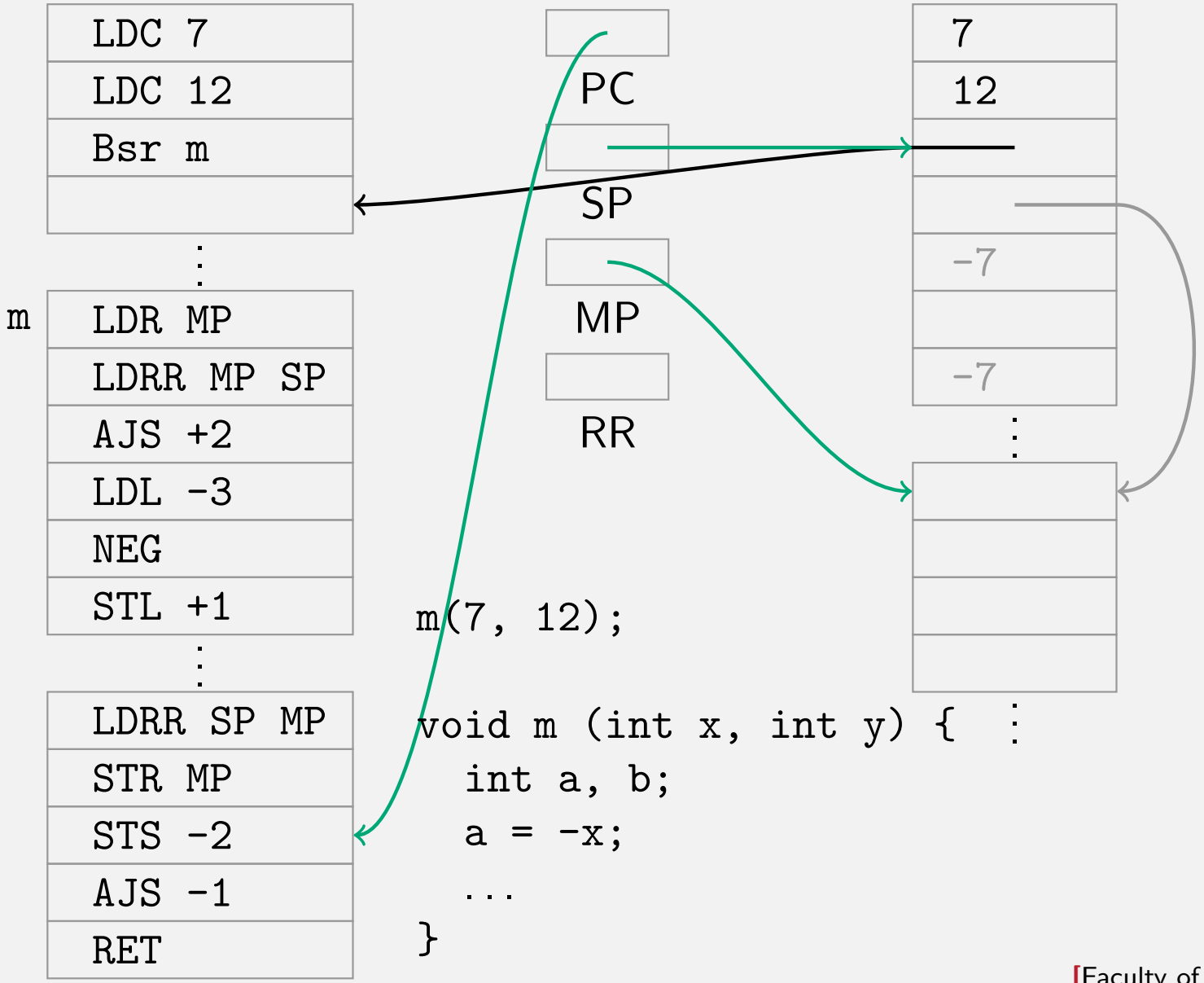
Methods with local variables



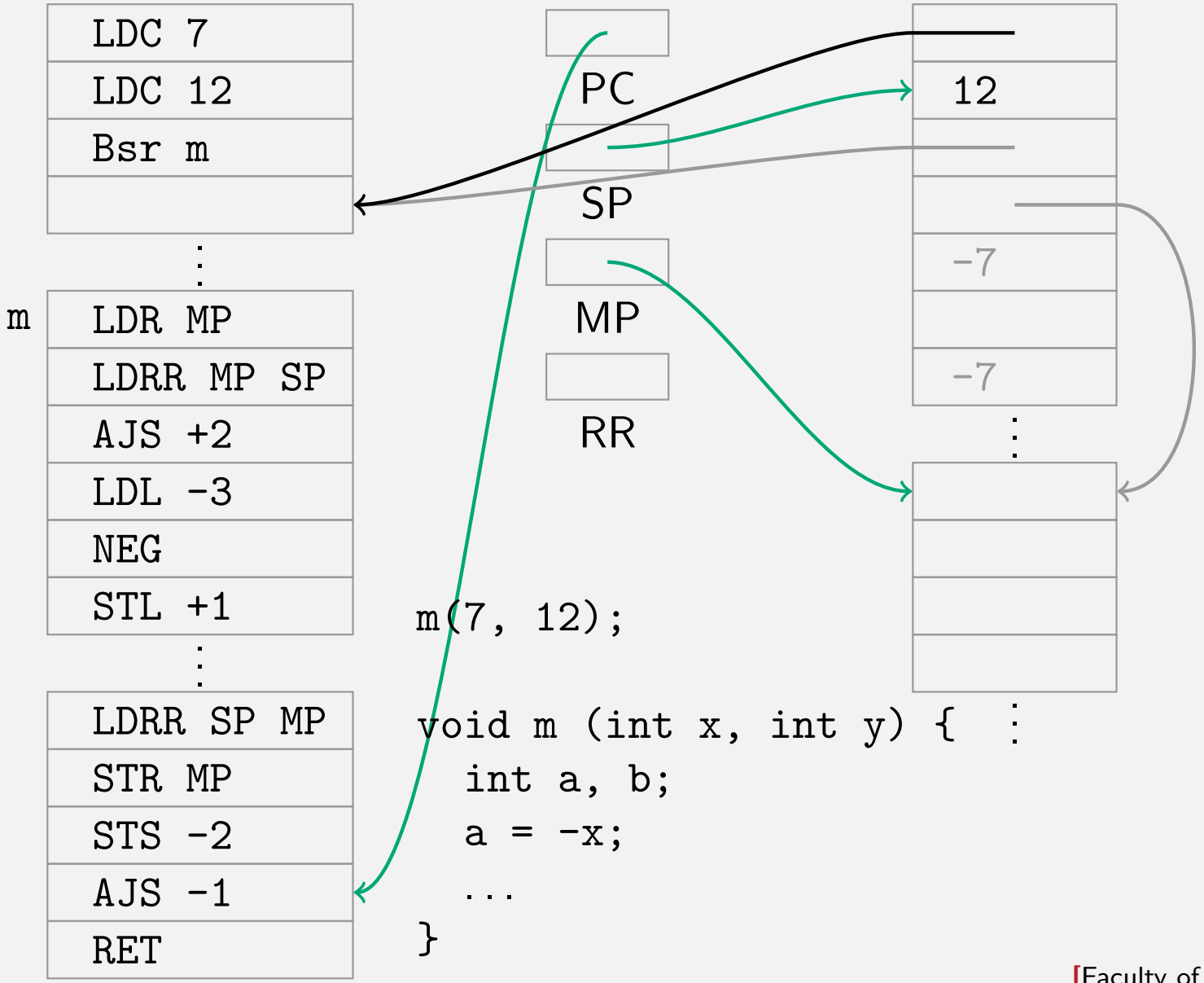
Methods with local variables



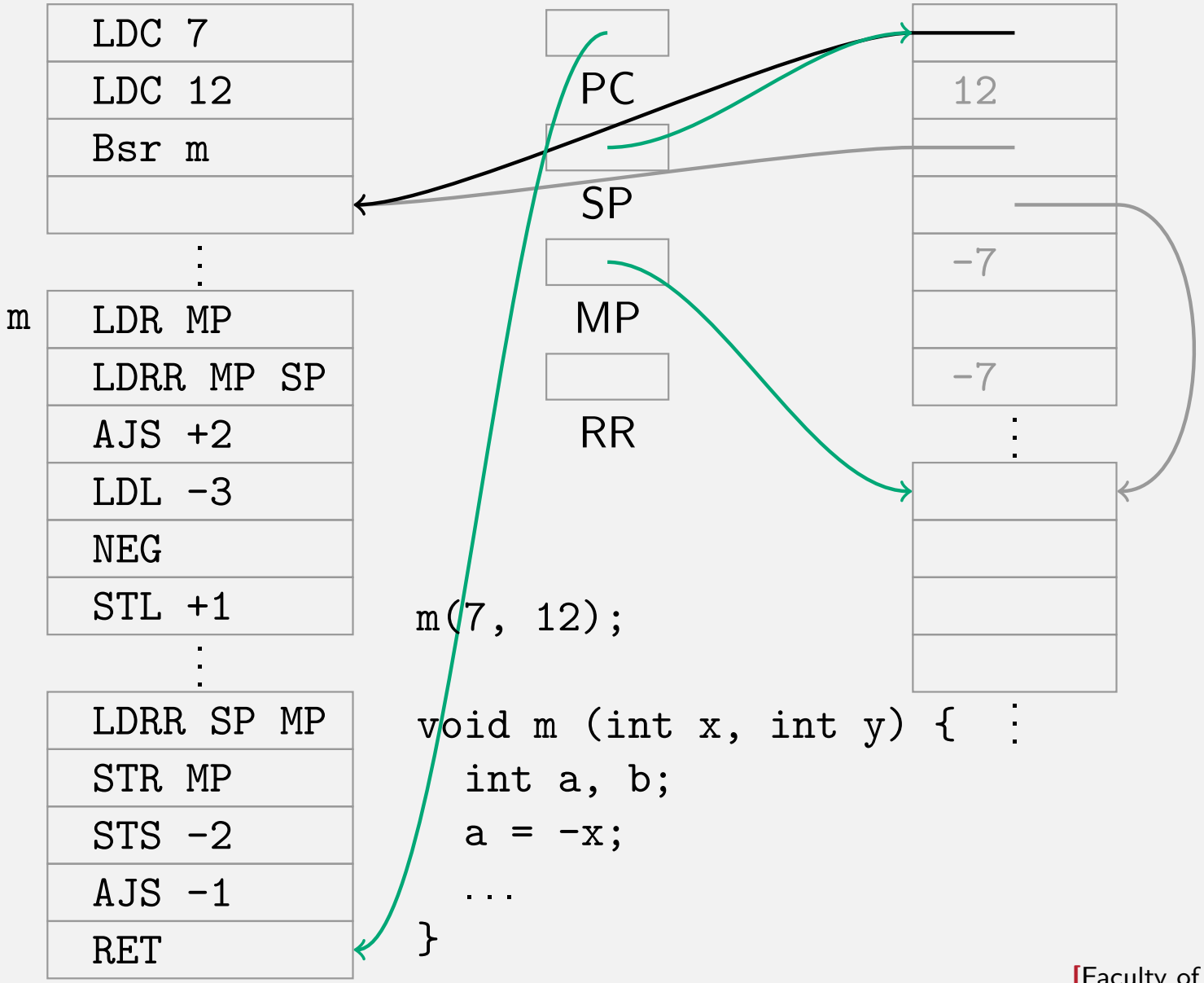
Methods with local variables



Methods with local variables



Methods with local variables



Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Method translation with local variables

Method call as before.

Method translation with local variables

Method call as before.

Method definition (n parameters, k local variables)

- Create room for local variables: `LDR MP` to save the mark pointer, `LDRR MP SP` to reset the mark pointer, `AJS` $+k$ to adjust the stack pointer. (Also available as a single instruction `LINK` k.)
- Use parameters: from `LDL` $-(n+1)$ to `LDL` -2.
- Use local variables: from `LDL` $+1$ to `LDL` $+k$.
- Clean up local variables: `LDRR SP MP` to reset the stack pointer, and `STR MP` to restore the mark pointer. (Also available as a single instruction `UNLINK`.)
- Clean up: `STS` $-n$ followed by `ADJ` $-(n-1)$.
- Return: `RET`

Methods with return values

Two options.

Methods with return values

Two options.

Result on stack

- Leave the result as the final value on the stack.
- Adapt the cleanup code so that this works.

Methods with return values

Two options.

Result on stack

- Leave the result as the final value on the stack.
- Adapt the cleanup code so that this works.

Result in register

- Place the result of a method call in a fixed free register (`RR` for example).
- Use the value from there at the call site.