# Stack and subroutines

R. Ferrero, P. Bernardi

Politecnico di Torino

Dipartimento di Automatica e Informatica (DAUIN)

Torino - Italy

- LDM {xx} / STM {xx} <Rm> {!}, <reglist> è il comando per trasferire una o più parole contenute nella reglist dall'indirizzo contenuto in Rm. Con il punto esclamativo Rm viene aggiornato puntando così all'ultimo valore inserito

- PUSH <reglist>
  POP <reglist>
  } facilitano l'uso di un full descending stack

- Una subroutine è chiamata con BL e con BLX (indirettamente)
  BL <label> oppure BXL <Rm> (con Rm = <label>)

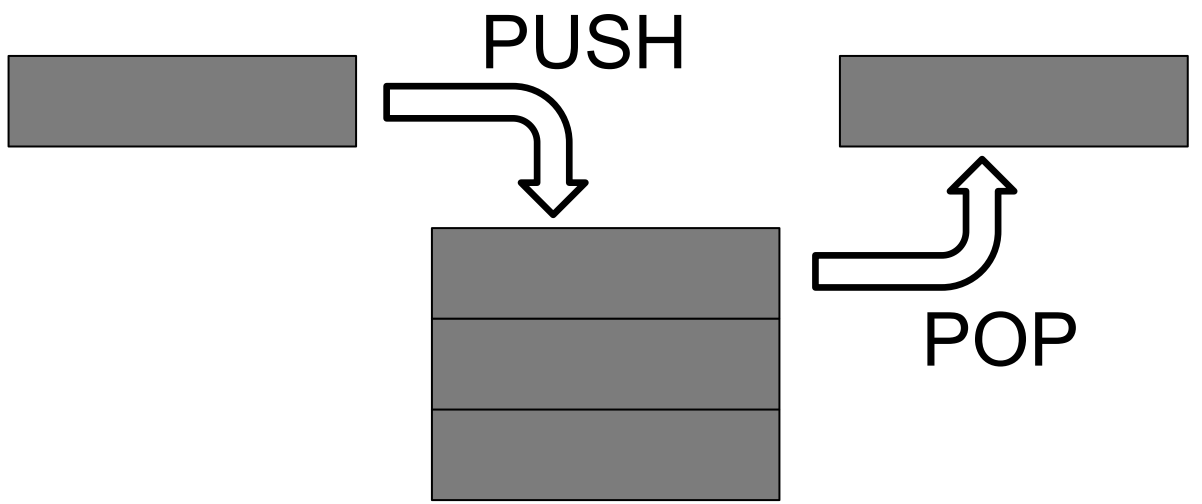
- L'inizio di una subroutine può essere indicato con le direttive PROC e ENDP oppure FUNCTION e ENDFUNC

- Una subroutine permette di tornare al programma principale con
  MOV PC, LR

- Ogni subroutine appena viene chiamata deve prima salvare i valori dei registri che userà nello stack + il LR → PUSH {reglist, LR}
  Alla fine la subroutine ripristina il PC e il valore iniziale dei registri usati → POP {reglist, PC}

# Stack

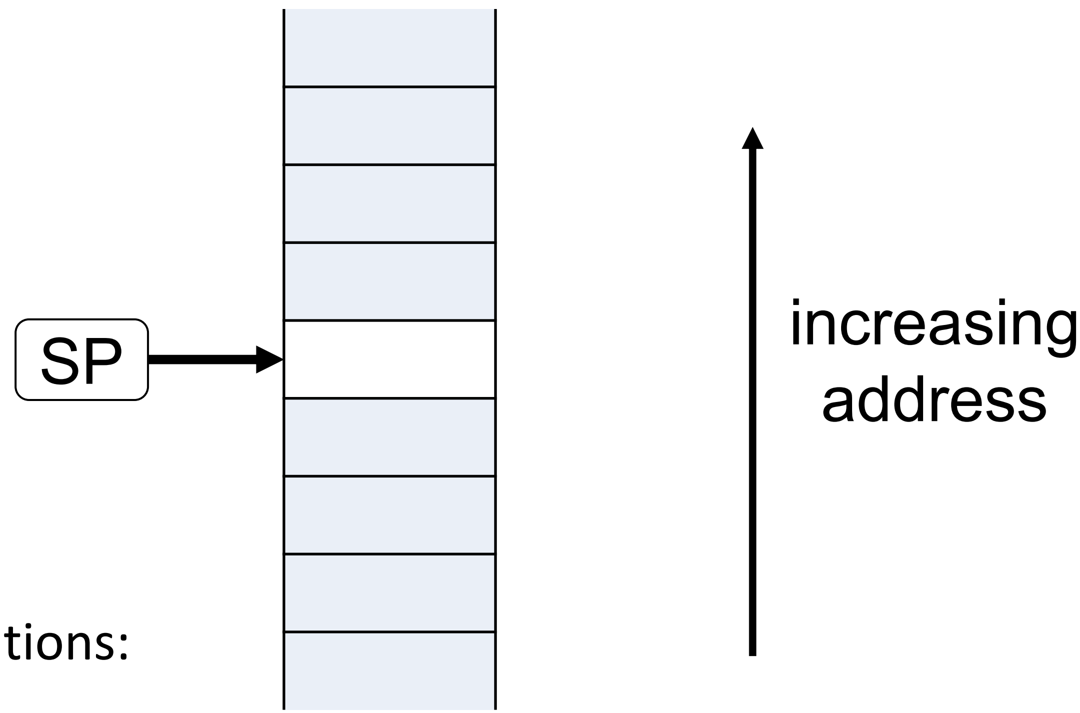
- A stack is a Last In-First Out (LIFO) queue.
- Data is pushed (written) to and popped (read) from the top of the stack.
- The stack pointer contains the address of the top of the stack.



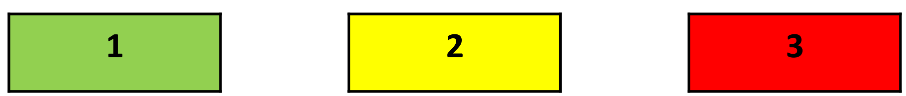
# Types of stack

- Stack pointer is updated after push:
  - *descending stack*: the address of the top of the stack decreases after a push
  - *ascending stack*: the address of the top of the stack increases after a push
- Content of the entry at the top of the stack:
  - *empty stack*: the stack pointer points to the entry where new data will be pushed
  - *full stack*: the stack pointer points to the last pushed entry.

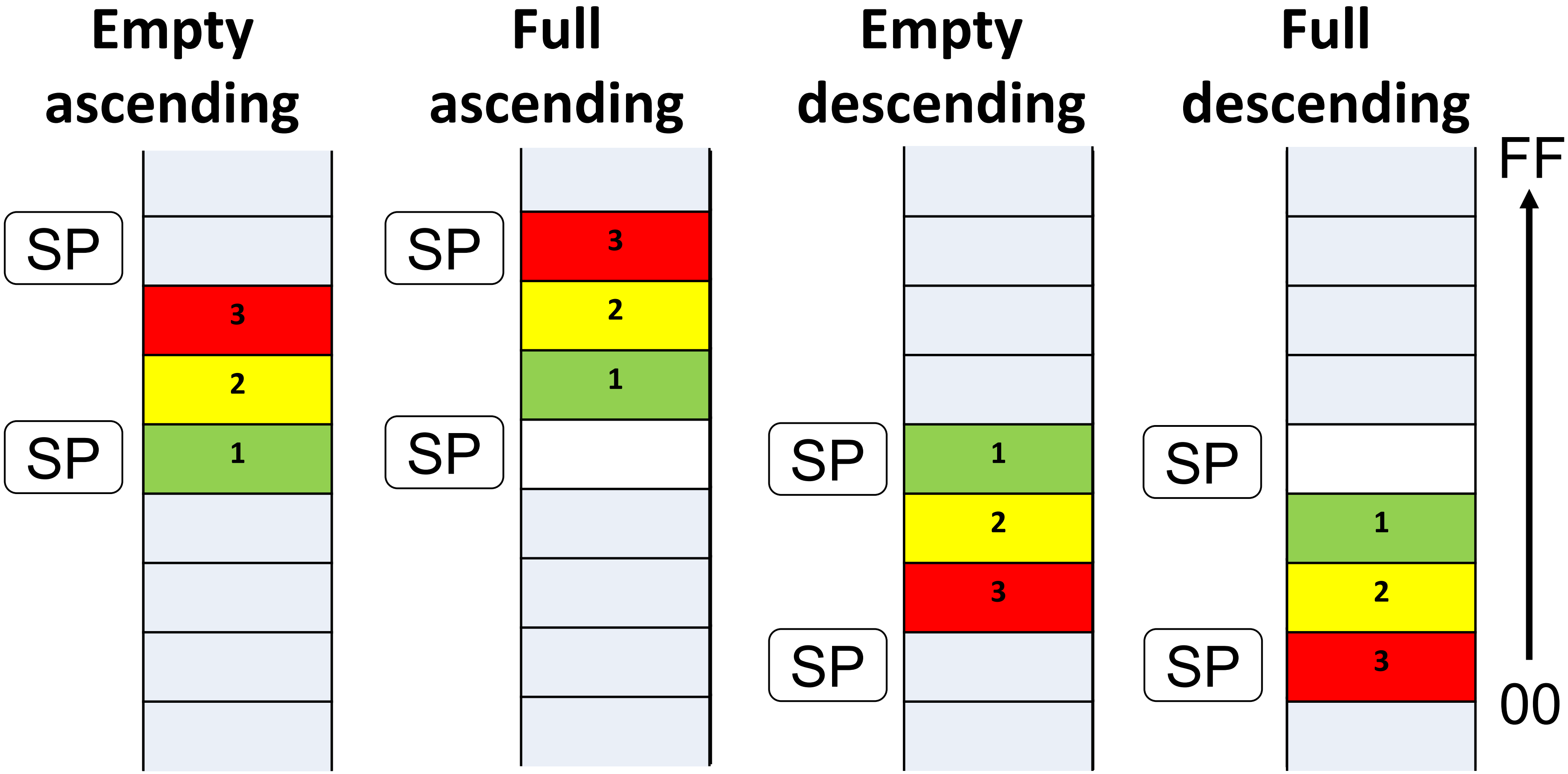
# Example with PUSH: initial state



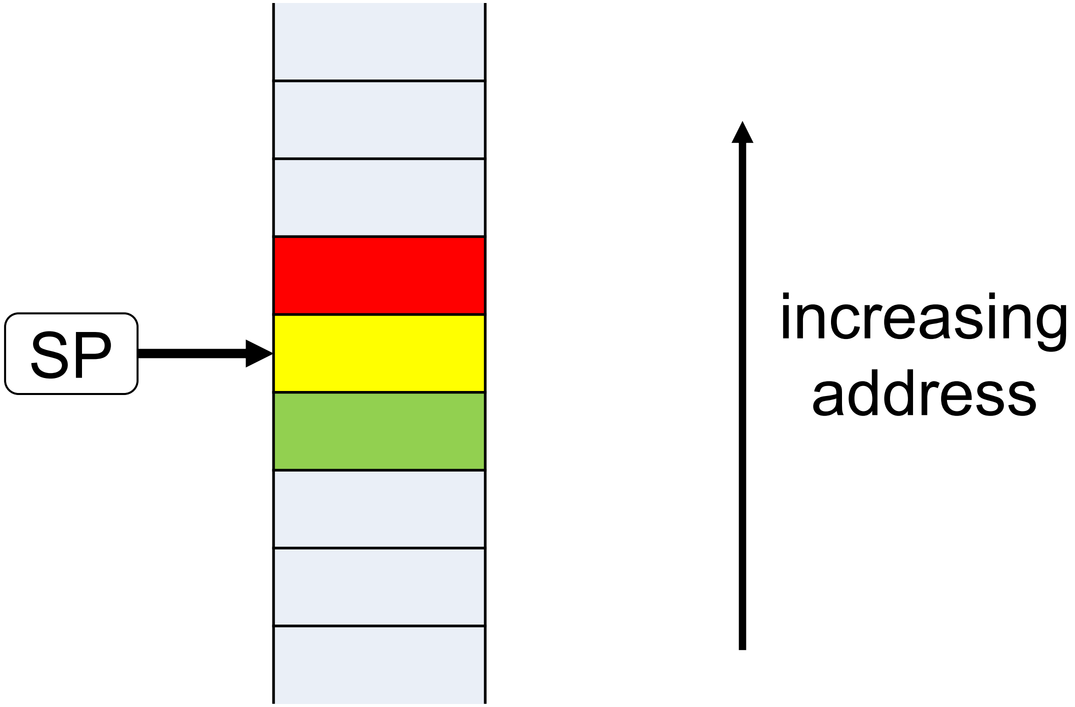

- 3 PUSH operations:



# Example after 3 PUSH operations

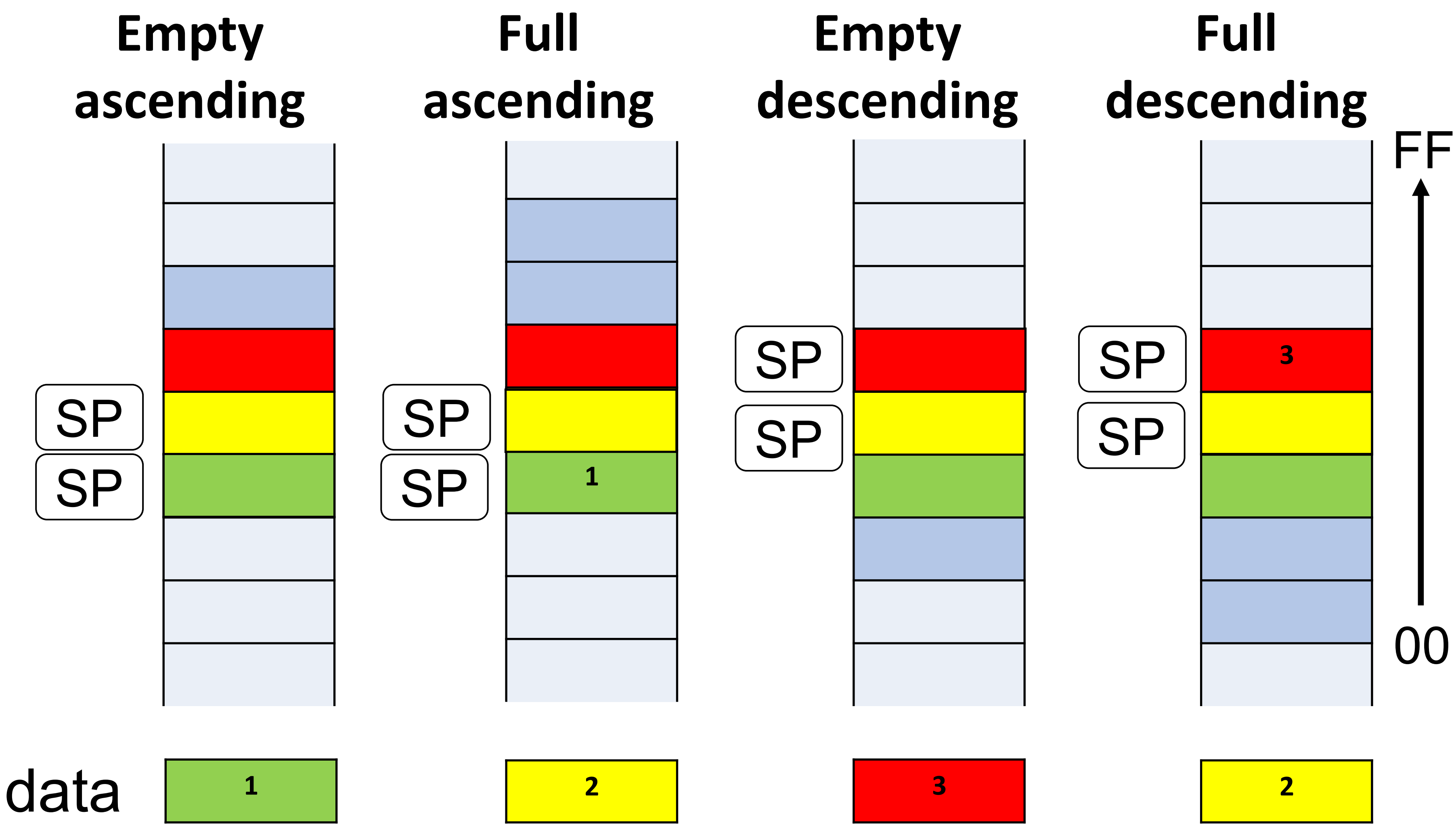


# Example with POP: initial state




- 1 POP

# Example after 1 POP



# LDM and STM

- They transfer one or more words:

`LDM{xx}/STM{xx} <Rn>{!}, <regList>`

- `Rn` is the base register
- `xx` specifies the addressing mode, i.e., how and when `Rn` is updated during the instruction (Increment After, decrement Before)
- at the end of the instruction:
  - with `!`, `Rn` is set to the updated value
  - without `!`, `Rn` is set to the initial value
- `regList` is a list of registers.

# List of registers

- Consecutive registers are indicated by separating the initial and final registers with a dash TRATTINO
- Non consecutive registers are separated with a comma.
- Example: `{r0-r4, r10, LR}` indicates `r0`, `r1`, `r2`, `r3`, `r4`, `r10`, `r14`.
- `SP` can not appear in the list.
- `PC` can appear only with `LDM` and only if `LR` is missing in the list.

# Order of registers in the list

Do R1 a R14

- The written order of registers does not matter.
- Registers are automatically sorted in increasing order:
  - the lowest register is stored into / loaded from the lowest memory address
  - the highest register is stored into / loaded from the highest memory address
- Example: `{r8, r1, r3-r5, r14}` indicates `r1`, `r3`, `r4`, `r5`, `r8`, `r14`.

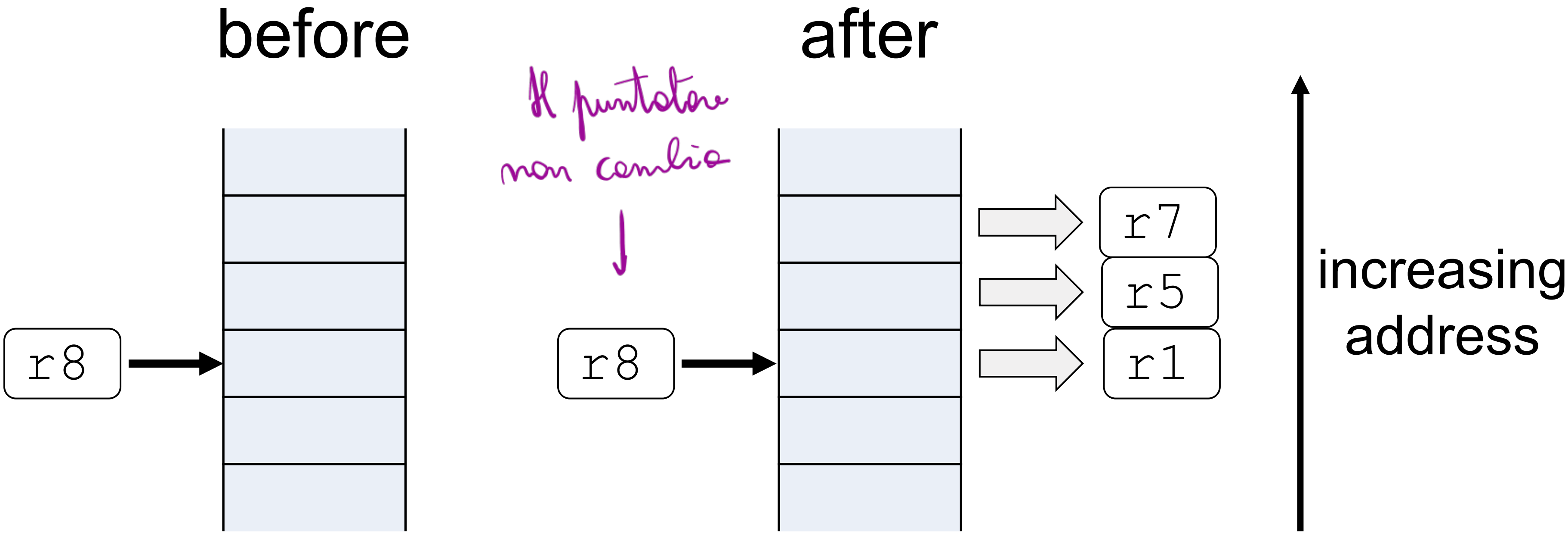
# Addressing modes

- `IA`: increment after (default)
  1. The memory is accessed at the address specified in the base register
  2. The base register is incremented by 1 word (4 bytes)
  3. if there are other registers in the list, go to 1.
- `DB`: decrement before
  1. The base register is decremented by 1 word (4 bytes)
  2. The memory is accessed at the address specified in the base register
  3. if there are other registers in the list, go to 1.

# LDMIA: an example

```
LDMIA r8, {r1, r5, r7}
LDM r8, {r1, r5, r7}
```

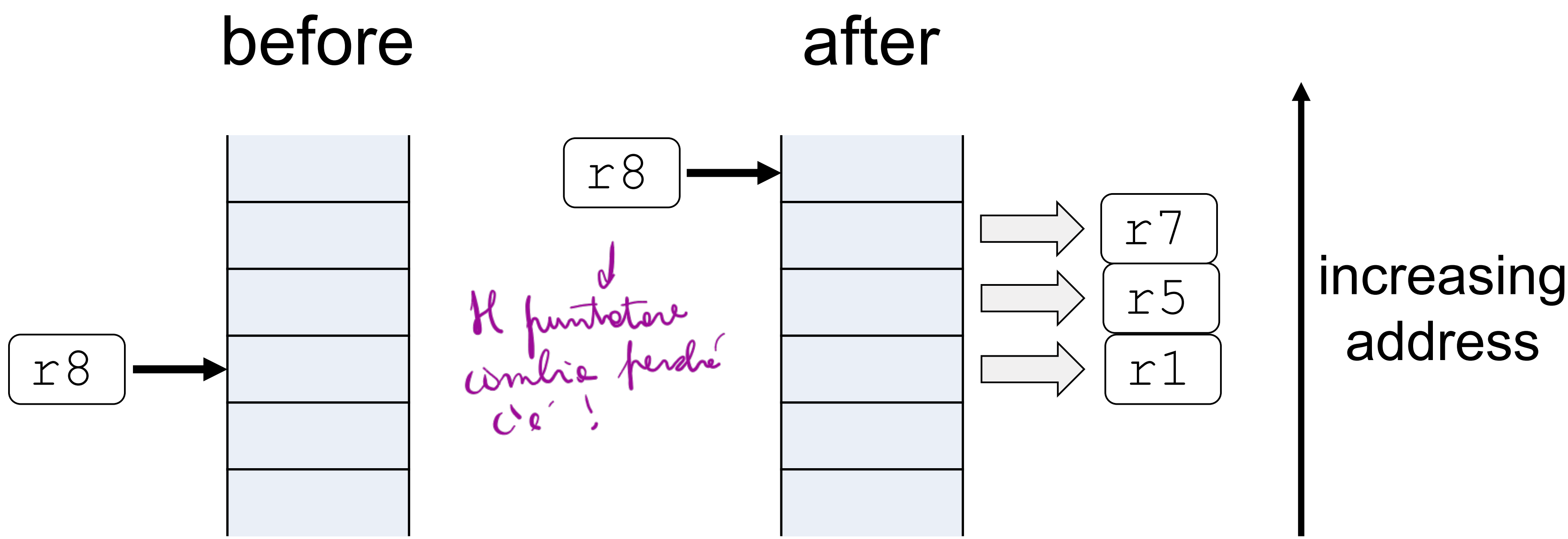
uguali tra loro



# LDMIA with '!': an example

```
LDMIA r8!, {r1, r5, r7}
LDM r8!, {r1, r5, r7}
```

uguali

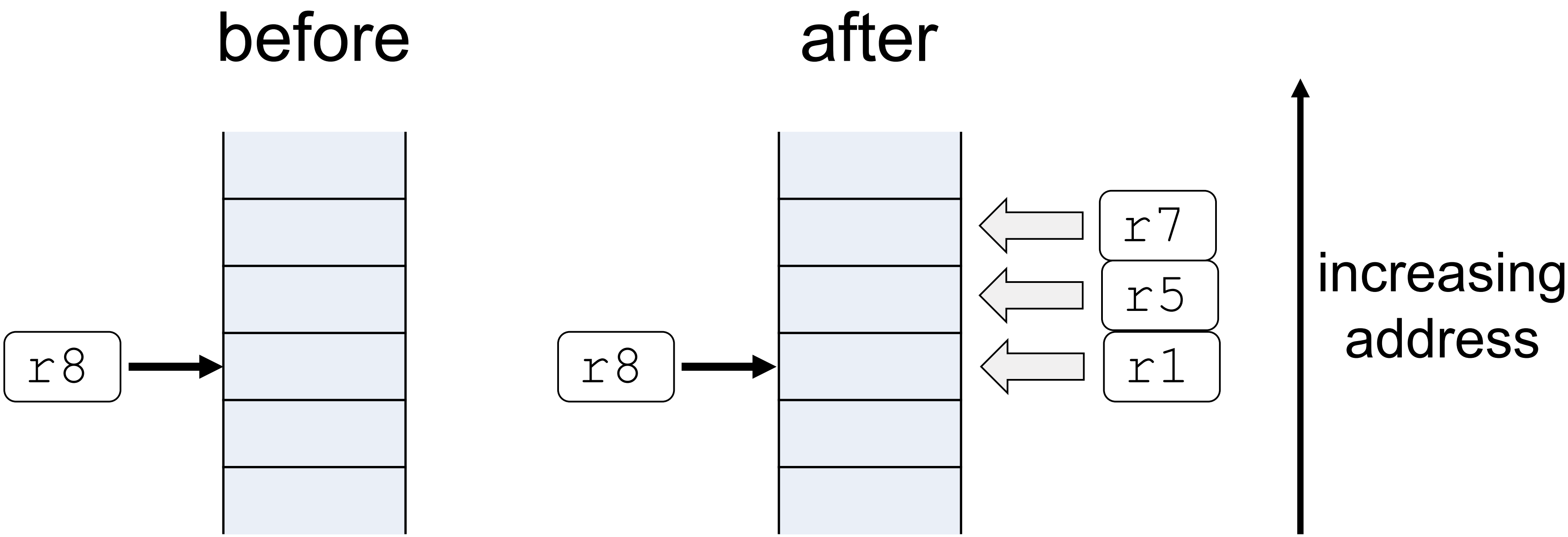



Il puntatore cambia perché c'è !

# STMIA: an example

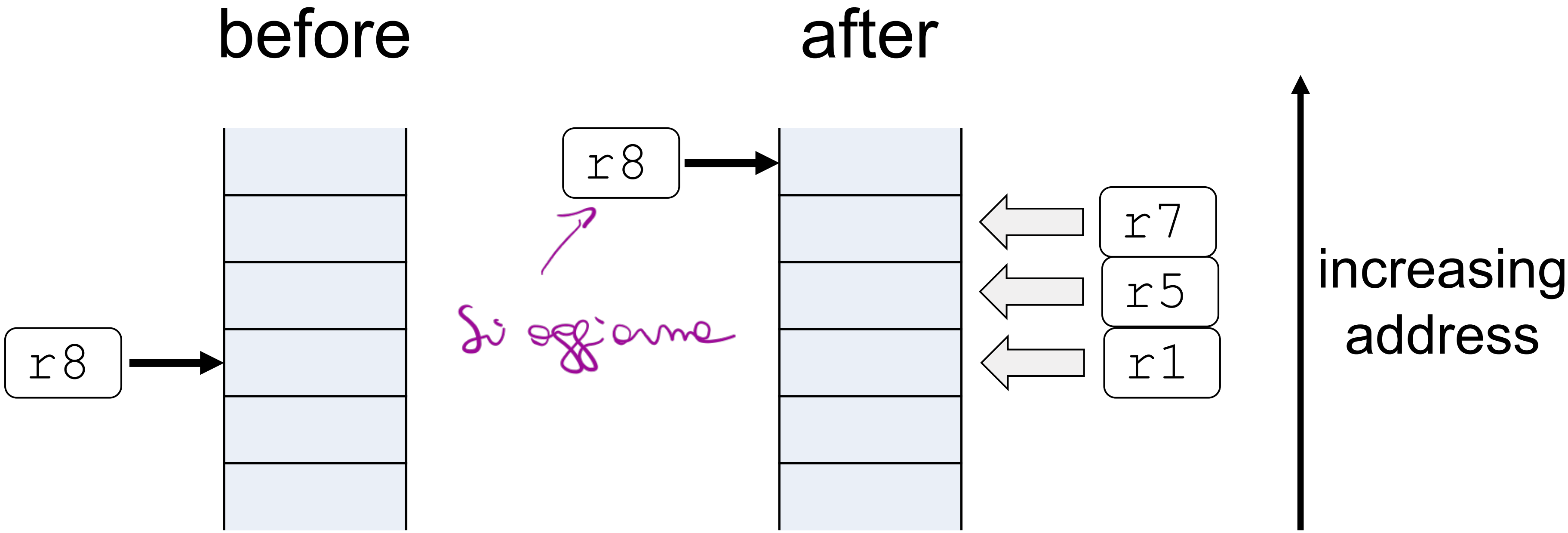
```
STMIA r8, {r1, r5, r7}
STM r8, {r1, r5, r7}
```

uguali



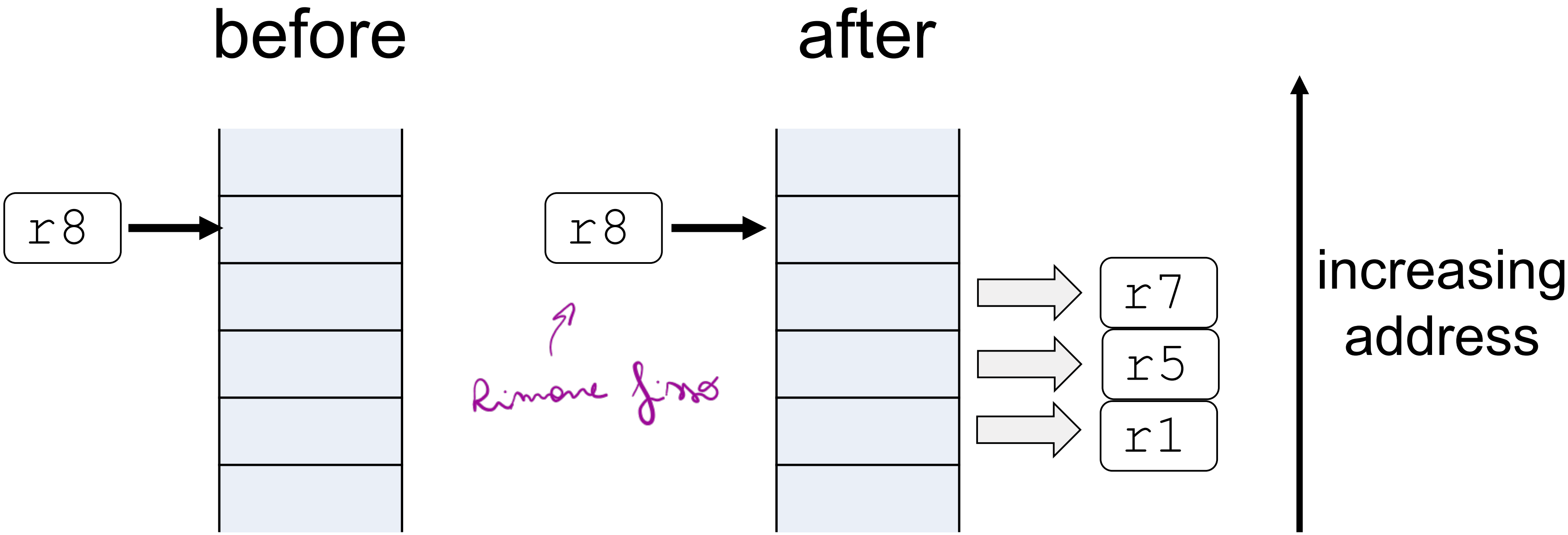
# STMIA with '!': an example

`STMIA r8!, {r1, r5, r7}` → uguali
`STM r8!, {r1, r5, r7}` →



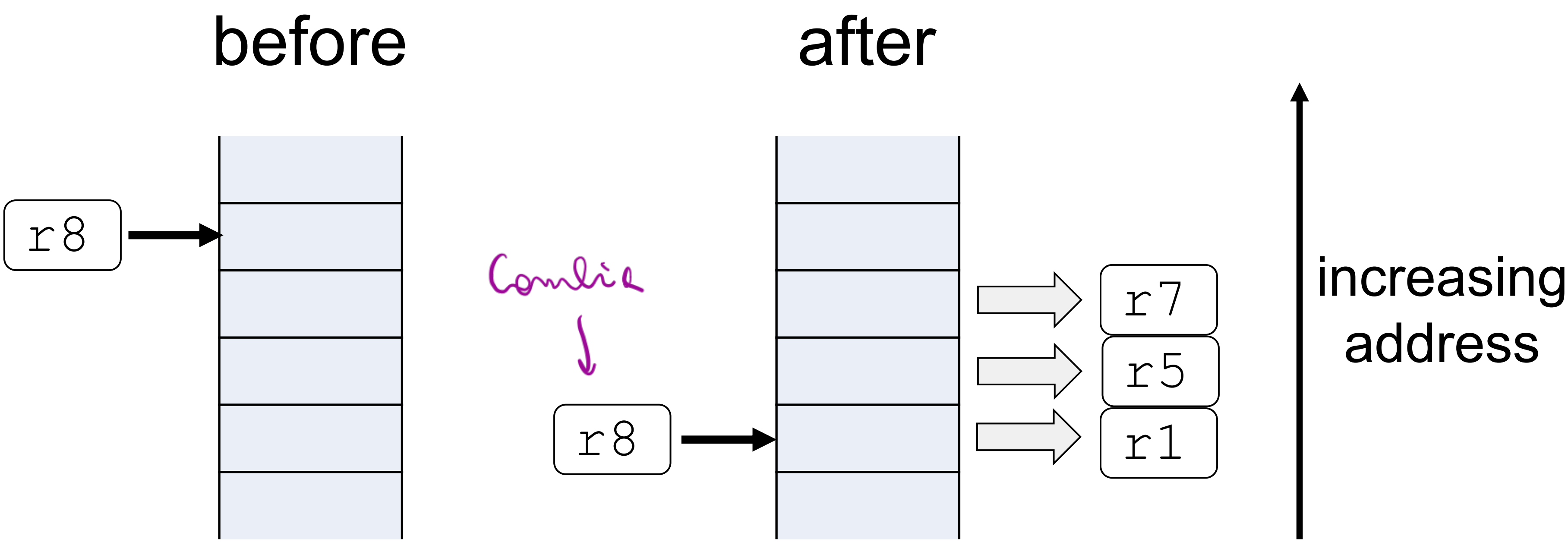
# LDMDB: an example

```
LDMDB r8, {r1, r5, r7}
```



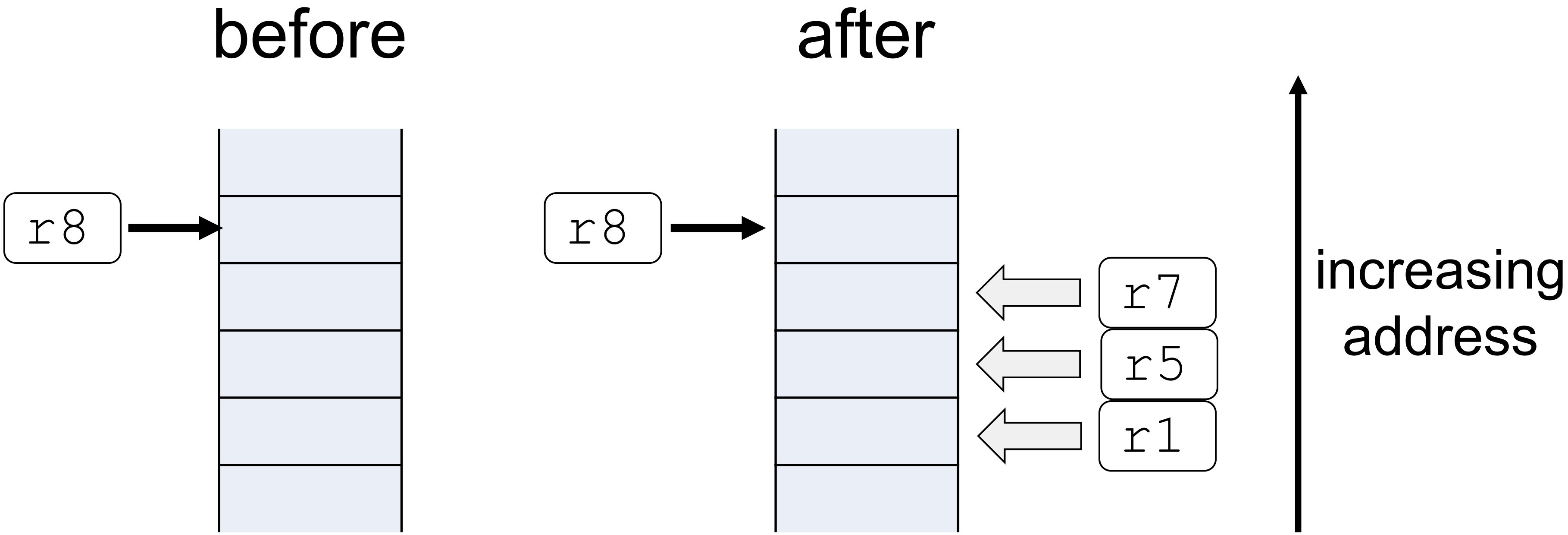
# LDMDB with '!': an example

```
LDMDB r8!, {r1, r5, r7}
```



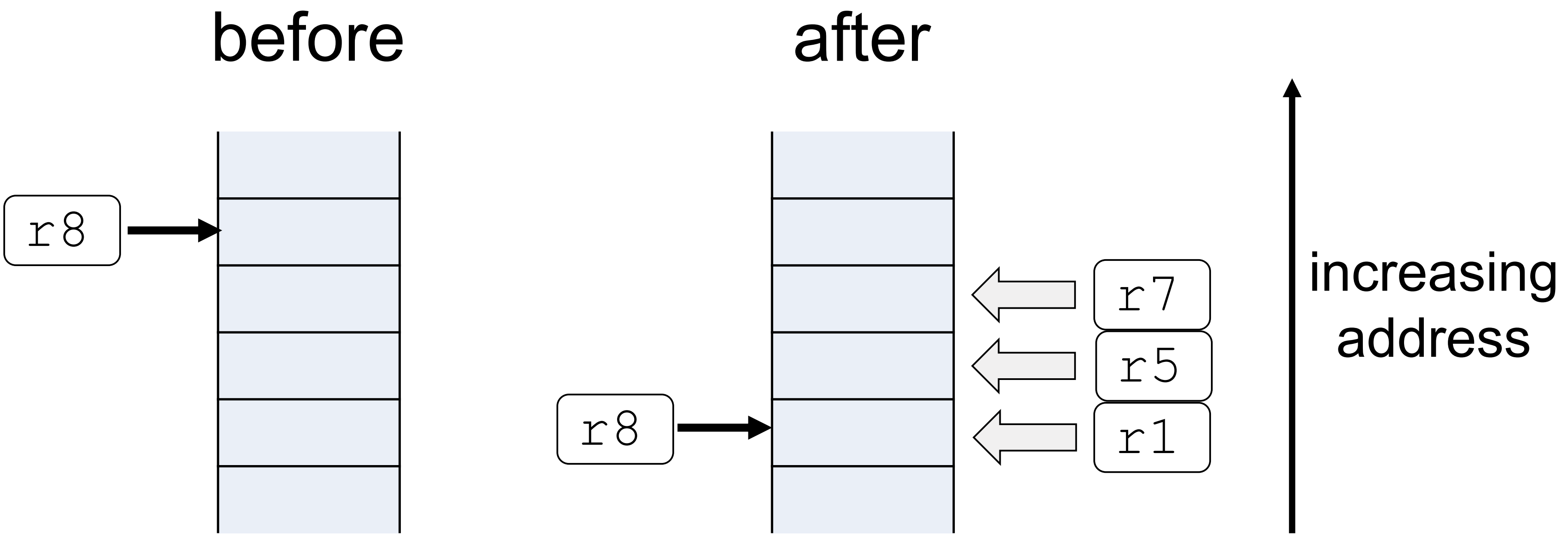
# STMDB: an example

```
STMDB r8, {r1, r5, r7}
```



# STMDB with '!': an example

```
STMDB r8!, {r1, r5, r7}
```



# Supported types of stack

- Stack-oriented suffixes can be used instead of increment/decrement and before/after.

| Stack type | PUSH | POP |
|---|---|---|
| Full descending | STMDB<br>STMFD | LDM<br>LDMIA<br>LDMFD |
| Empty ascending | STM<br>STMIA<br>STMEA | LDMDB<br>LDMEA |

Il nostro

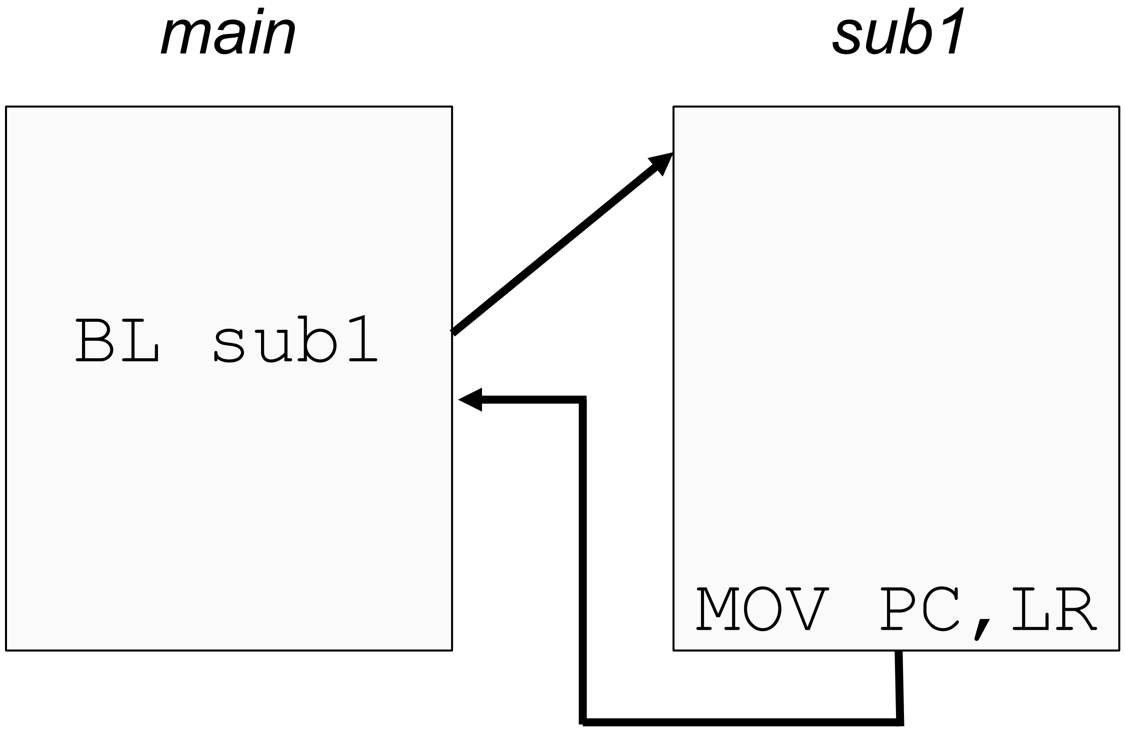
# PUSH and POP

- `PUSH` and `POP` instructions facilitate the use of a **full descending stack**.
- `PUSH <regList>` is the same as `STMDB SP!, <regList>`
- `POP <regList>` is the same as `LDMIA SP!, <regList>`

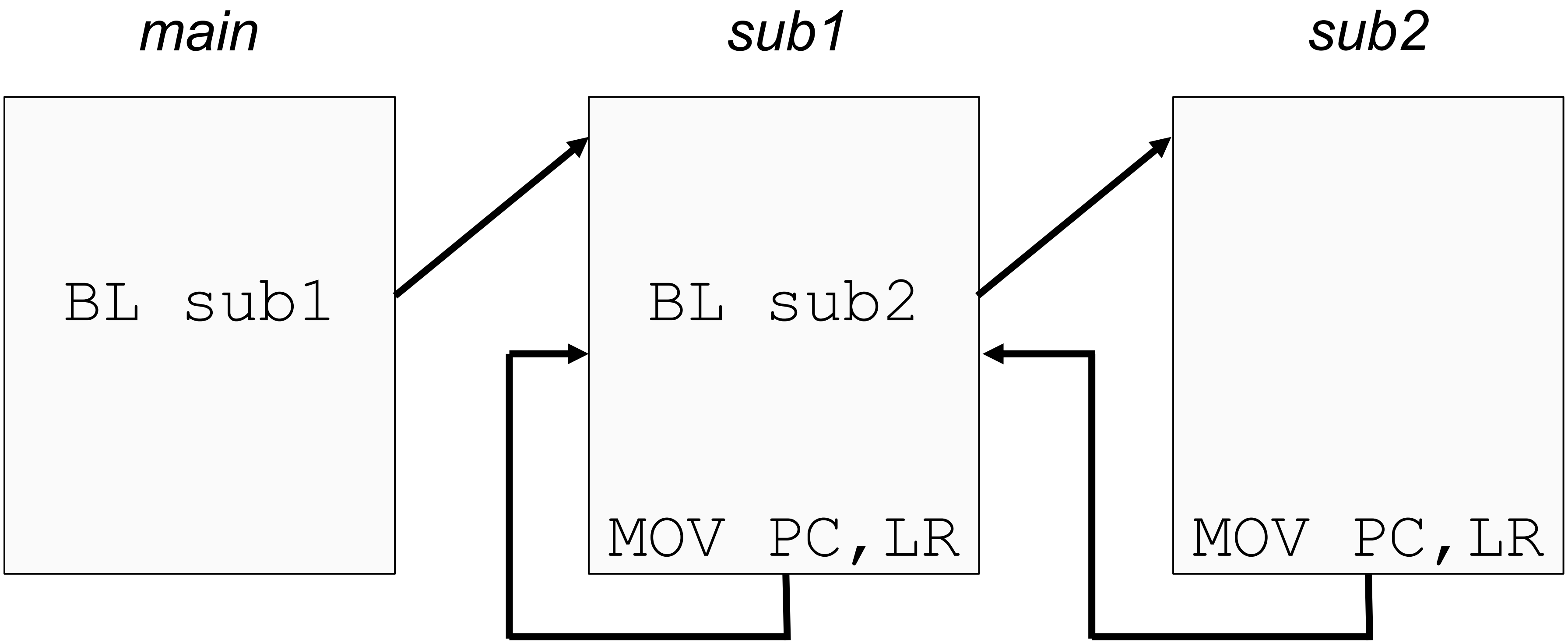
# Subroutine

- A subroutine is called with `BL` and `BLX`.
- `BL <label>` and `BLX <Rn>`:
  - write the address of the next instruction to `LR`
  - write the value of `label` or `Rn` to `PC`
- A reentrant procedure ends with a branch to the address stored in `LR`.
- Optionally, the begin and end of a subroutine can be indicated with the directives `PROC`/`FUNCTION` and `ENDP`/`ENDFUNC`.

# Call to subroutine



# Nested calls to subroutines




- When *sub1* calls *sub2*, `LR` is overwritten.
- *sub1* is not able to return to *main*.

# Nested calls to subroutines

- Besides changing `LR` when called, *sub2* may also change the value of registers used in *sub1*.
- Every subroutine should save `LR` and the other used registers as first instruction:

```
PUSH {regList, LR}
```

- At the end, the subroutine restores `PC` and the initial value of the used registers:

```
POP {regList, PC}
```

# Passing parameters and result

- There are three approaches:
  - in registers
  - by reference, i.e., a register with an address in memory
  - on the stack
- Example: a main routine calls a subroutine for computing the absolute difference of two unsigned numbers.

# Passing parameters in registers

```
    MOV r0, #0x34
    MOV r1, #0xA3
    BL sub1
    ; r2 contains the result
...
stop B stop
...
```

# Passing parameters in registers

```
sub1    PROC
        PUSH {LR}
        CMP r0, r1
        SUBHS r2, r0, r1
        SUBLO r2, r1, r0
        POP {PC}
        ENDP
```

# Passing parameters by reference

```
    MOV r0, #0x34
    MOV r1, #0xA3
    LDR r3, =mySpace
    STMIA r3, {r0, r1}
    BL sub2
    LDR r2, [r3]
    ; r2 contains the result
...
stop B stop
...
```

By reference parameter

# Passing parameters by reference

```
sub2    PROC
        PUSH {r2, r4, r5, LR}
        LDMIA r3, {r4, r5}
        CMP r4, r5
        SUBHS r2, r4, r5
        SUBLO r2, r5, r4
        STR r2, [r3]
        POP {r2, r4, r5, PC}
        ENDP
```

Metto i valori del main da parte

Riprendo i valori del main

# Passing parameters on the stack

```
    MOV r0, #0x34
    MOV r1, #0xA3
1   PUSH {r0, r1, r2}
    BL sub3
    POP {r0, r1, r2}
    ; r2 contains the result
...
stop B stop
...
```

Prepare arguments and leave space for results

# Passing parameters on the stack

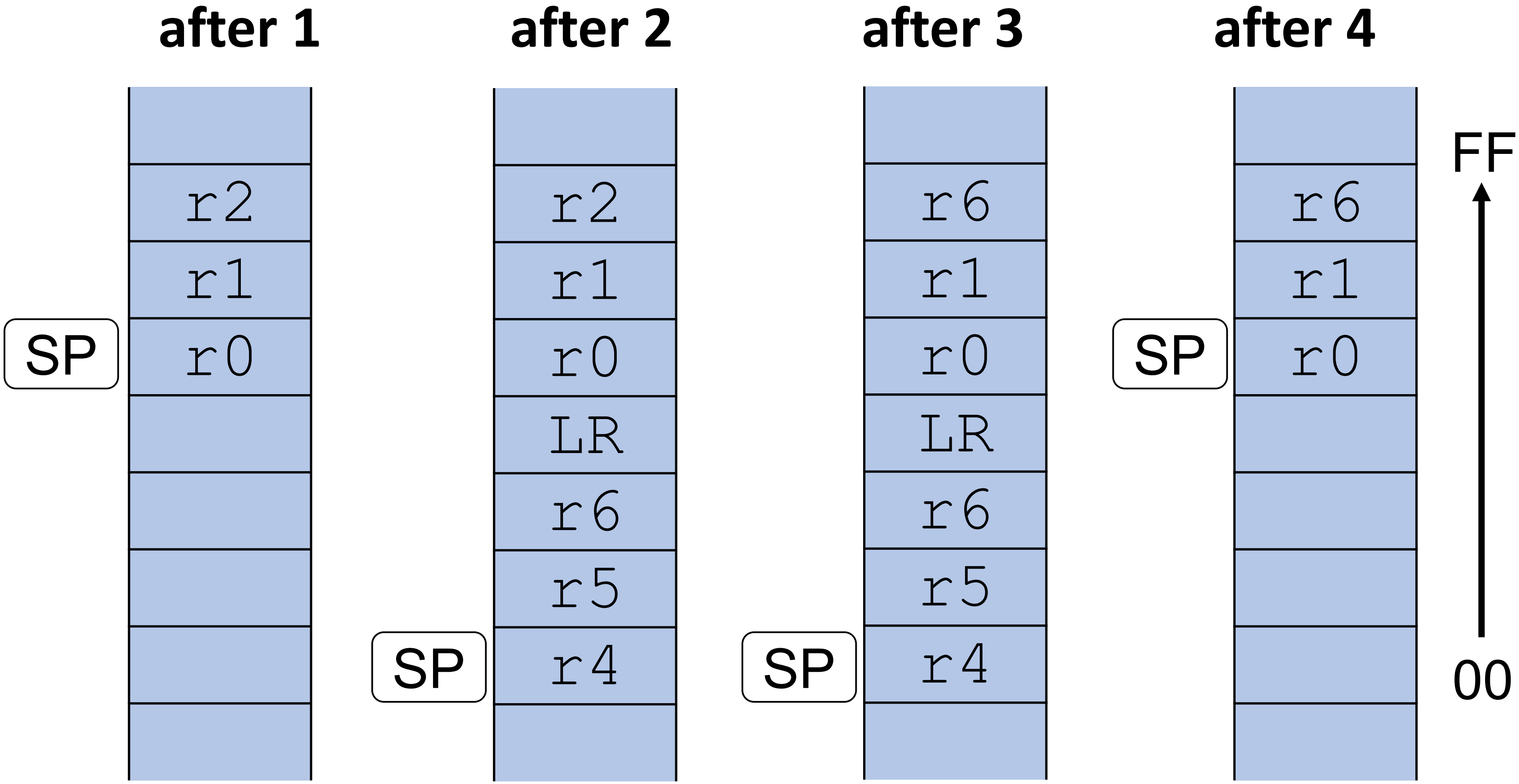
```
sub3      PROC
        2 PUSH {r6, r4, r5, LR}
          LDR r4, [sp, #16]  offset
          LDR r5, [sp, #20]  offset
          CMP r4, r5
          SUBHS r6, r4, r5
          SUBLO r6, r5, r4
        3 STR r6, [sp, #24]
        4 POP {r6, r4, r5, PC}
          ENDP
```

Save all registers that are used to hold the values of a routine's local variables

Saves results

Restore registers content, update PC to LR to return to the caller function

# Elements in the stack



# ABI – Application Binary Interface

- In computer software, an Application Binary Interface (ABI) is an interface between two binary program modules;
  - often, one of these modules is a library or operating system facility, and
  - the other is a program that is being run by a user
- A common aspect of an ABI is the calling convention, which determines how data is provided as input to or read as output from computational routines.

# ABI standard for ARM



## Application Binary Interface for the ARM® Architecture
### The Base Standard

Document number: ARM IHI 0036B, current through ABI release 2.09
Date of Issue: 10th October 2008, reissued 30th November 2012

**Abstract**

This document describes the structure of the Application Binary Interface (ABI) for the ARM architecture, and links to the documents that define the base standard for the ABI for the ARM Architecture. The base standard governs inter-operation between independently generated binary files and sets standards common to ARM-based execution environments.

**Keywords**

ABI for the ARM architecture, ABI base standard, embedded ABI

# ABI components

## 1.2 References

This document refers to the following documents.

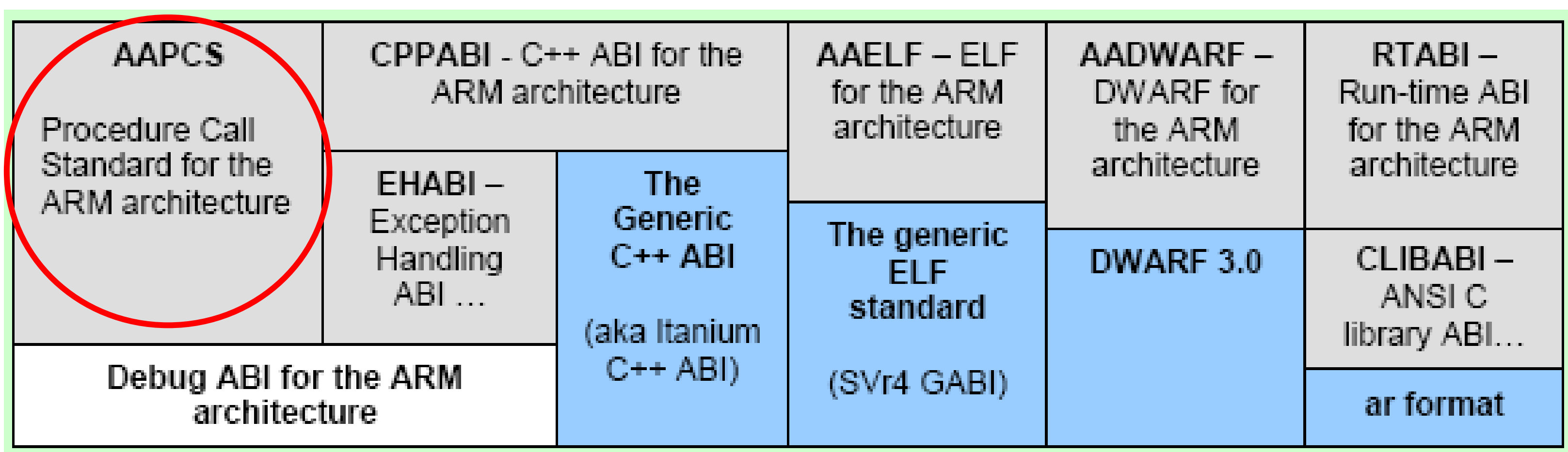
| Ref | External URL | Title |
|---|---|---|
| AADWARF | | DWARF for the ARM Architecture |
| AAELF | | ELF for the ARM Architecture |
| AAPCS | | Procedure Call Standard for the ARM Architecture |
| ADDENDA | | Addenda to, and errata in, the ABI for the ARM Architecture |
| BPABI | | Base Platform ABI for the ARM Architecture |
| **BSABI** | *This document* | ABI for the ARM Architecture (Base Standard) |
| CLIBABI | | C Library ABI for the ARM Architecture |
| CPPABI | | C++ ABI for the ARM Architecture |
| EHABI | | Exception Handling ABI for the ARM Architecture |
| EHEGI | | Exception handling components, example implementations |
| RTABI | | Run-time ABI for the ARM Architecture |

# Procedure Call Standard for the ARM architecture

- The *Procedure Call Standard for the ARM architecture* [AAPCS] specifies the use of the run-time stack, and the stack invariants that must be preserved.

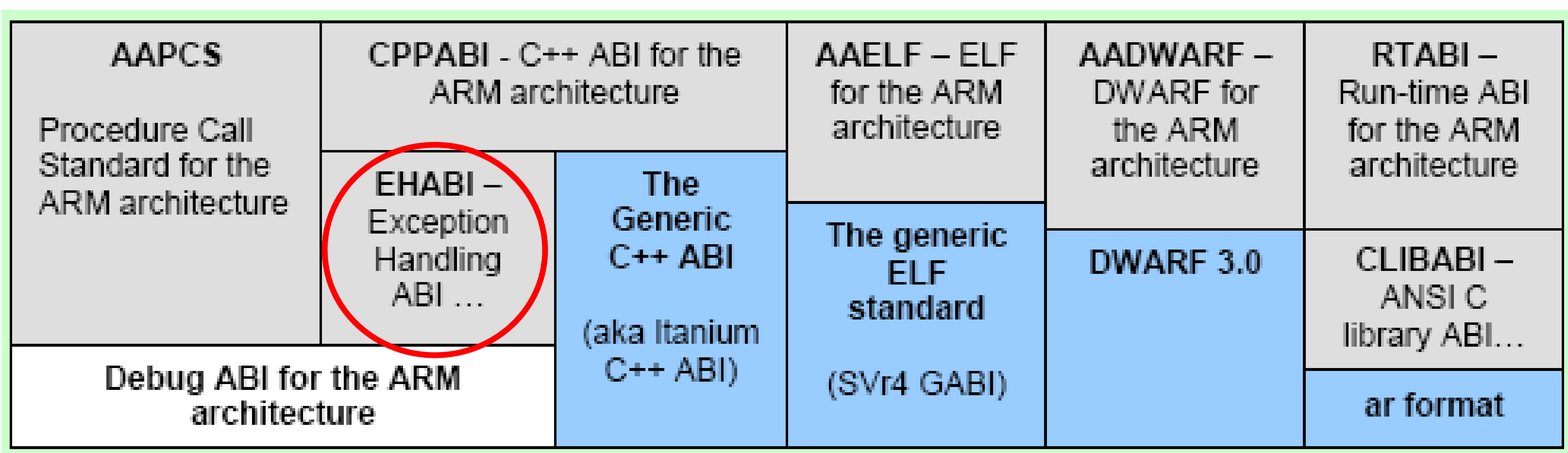
↳ Procedure non restituisce valore

Funzione lo restituisce



# Exception Handling ABI for the ARM architecture

- The *Exception Handling ABI for the ARM architecture* [EHABI] specifies table-based stack unwinding that separates language-independent unwinding from language specific concerns.



# ABI and other definitions in AAPCS

| Term | Meaning |
|---|---|
| ABI | Application Binary Interface:<br>1. The specifications to which an executable must conform in order to execute in a specific execution environment. For example, the *Linux ABI for the ARM Architecture*.<br>2. A particular aspect of the specifications to which independently produced relocatable files must conform in order to be statically linkable and executable. For example, the *C++ ABI for the ARM Architecture*, the *Run-time ABI for the ARM Architecture*, the *C Library ABI for the ARM Architecture.* |
| ARM-based | … based on the ARM architecture … |
| EABI | An ABI suited to the needs of embedded (sometimes called *free standing*) applications. |

| | |
|---|---|
| Routine, subroutine | A fragment of program to which control can be transferred that, on completing its task, returns control to its caller at an instruction following the call. *Routine* is used for clarity where there are nested calls: a routine is the *caller* and a subroutine is the *callee*. |
| Procedure | A routine that returns no result value. |
| Function | A routine that returns a result value. |
| Activation stack, call-frame stack | The stack of routine activation records (call frames). |
| Activation record, call frame | The memory used by a routine for saving registers and holding local variables (usually allocated on a stack, once per activation of the routine). |

non posso usarli

| Register | Synonym | Special | Role in the procedure call standard |
|---|---|---|---|
| r15 | | PC | The Program Counter. |
| r14 | | LR | The Link Register. |
| r13 | | SP | The Stack Pointer. |
| r12 | | IP | The Intra-Procedure-call scratch register. |
| r11 | v8 | | Variable-register 8. |
| r10 | v7 | | Variable-register 7. |
| r9 | | v6<br>SB<br>TR | Platform register.<br>The meaning of this register ned by the platform standard. |
| r8 | v5 | | Variable-register 5. |
| r7 | v4 | | Variable register 4. |
| r6 | v3 | | Variable register 3. |
| r5 | v2 | | Variable register 2. |
| r4 | v1 | | Variable register 1. |
| r3 | a4 | | Argument / scratch register 4. |
| r2 | a3 | | Argument / scratch register 3. |
| r1 | a2 | | Argument / result / scratch register 2. |
| r0 | a1 | | Argument / result / scratch register 1. |

*Table 2, Core registers and AAPCS usage*

Can be freely used to hold local variables

If there are more than 4 formal arguments, they have to be saved in the stack

# Passing arguments

- The first four registers r0-r3 (a1-a4) are used to pass argument values into a subroutine and to return a result value from a function.
  - A subroutine must preserve the contents of the registers r4-r8, r10, r11 and SP

Callee saving of caller status

- The base standard provides for passing arguments in core registers (r0-r3) and on the stack.
  - For subroutines that take a small number of parameters, only registers are used, greatly reducing the overhead of a call.

Caller saving to preserve status of non-volative register values

# STACK management

- The stack implementation is *full-descending*, with the current extent of the stack held in the register SP (r13).
- The stack will, in general, have both a *base* and a *limit*
  - though in practice an application may not be able to determine the value of either.

| | | |
|---|---|---|
| Full descending | STMDB<br>STMFD | LDM<br>LDMIA<br>LDMFD |

# Local variables in stack – stack frame

- It is possible to create local variables on the stack
  - in the same way we stored also the saved values there, by simply *subtracting the number of bytes required by each variable* from the stack pointer.
  - This does not store any data in the variables, it simply sets aside memory that we can use.