

Parameter Passing

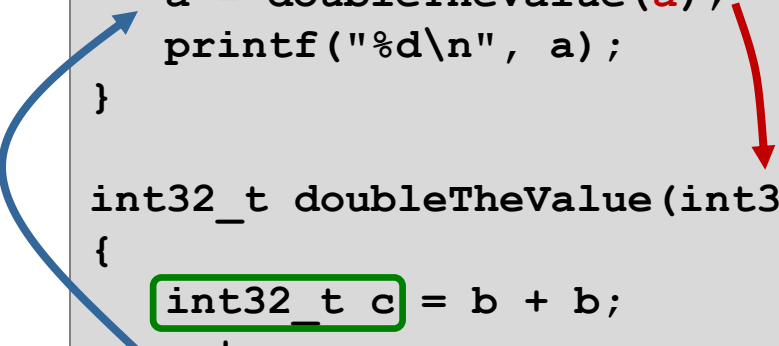
Computer Engineering 1

```
#include <stdio.h>

int32_t doubleTheValue(int32_t b);

int32_t main(void)
{
    int32_t a = 5;
    a = doubleTheValue(a);
    printf("%d\n", a);
}

int32_t doubleTheValue(int32_t b)
{
    int32_t c = b + b;
    return c;
}
```



- How does `main()` pass the value of **a** to the function?
- Where is the local variable **c** stored?
- How is the value of **c** returned to `main()`?

- **Parameter Passing**
- **Passing through Registers**
- **Passing through Global Variables**
- **Reentrancy**
- **ARM Procedure Call Standard**
- **Functions - Stack Frame**
- **Calling Assembly Subroutines from C**

At the end of this lesson you will be able

- to explain and classify the different possibilities to pass data between different parts of the program
- to outline what an Application Binary Interface is
- to name the roles of the different registers in the ARM Procedure Call Standard
- to enumerate and describe the operations of the caller of a subroutine
- to summarize the structure of a subroutine and describe what happens in the prolog and epilog respectively
- to explain, interpret and discuss stack frames
- to access elements of a stack frame in assembly
- to understand the build-up and tear-down of stack-frames
- to call an assembly subroutine from a C program

■ Where?

- **Register**
 - Caller and Callee¹⁾ use the same register
- **Global variables**
 - Shared variables in data area (section)
- **Stack**
 - Caller → PUSH parameter on stack
 - Callee → access parameter through `LDR <Rt>, [SP, #<imm>]`

■ How?

- **pass by value**
 - Handover the value
- **pass by reference**
 - Handover the address to a value

¹⁾ Caller: the routine that calls the subroutine; Callee: the subroutine being called

Passing through Registers

Register / "pass by value"

```
AREA exData, DATA, ...
```

```
...
```

```
AREA exCode, CODE, ...
```

```
...
```

```
MOVS R1, #0x03
```

```
BL double
```

```
MOVS ..., R0
```

```
...
```

```
double
```

```
LSLs R0, R1, #1
```

```
BX LR
```

caller

callee
function
double

■ Values in agreed registers, e.g.

- **R1** Parameter:
Caller → function
- **R0** Return value of function

■ Efficient and simple

■ Limited number of registers

- How do we pass tables and structs?

Register / "pass by reference"

```
TLENGTH      EQU      16

AREA exData,DATA,...
p1Table SPACE TLENGTH

AREA exCode,CODE,...
... 1)
LDR    R0,=p1Table
MOVS   R1,#TLENGTH
BL     doubleTableValues
...

doubleTableValues
    MOVS   R2,#0
loop  LDRB  R4,[R0,R2]
      LSL  R4,R4,#1
      STRB R4,[R0,R2]
      ADDS R2,#1
      CMP  R2,R1
      BLO  loop
      BX   LR
```

caller

callee

function

doubleTableValues

- Pass reference (= address) of data structure in register
- Allows passing of larger structures
- Example
 - Function `doubleTableValues`
 - doubles each value in the table
 - **R0** Caller passes address of `p1Table`
 - **R1** Caller passes length of table (pass by value)

¹⁾ Filling the table with values is not shown in the code

Global Variables

```
AREA exData, DATA, ...
param1 SPACE 1
result SPACE 1

AREA exCode, CODE, ...

...
LDR      R4, =param1
MOVS     R5, #0x03
STRB     R5, [R4]
BL       double_g
LDR      R4, =result
LDRB     ..., [R4]
...

double_g
LDR      R4, =param1
LDRB     R1, [R4]
LSLS     R0, R1, #1
LDR      R4, =result
STRB     R0, [R4]
BX       LR
```

caller

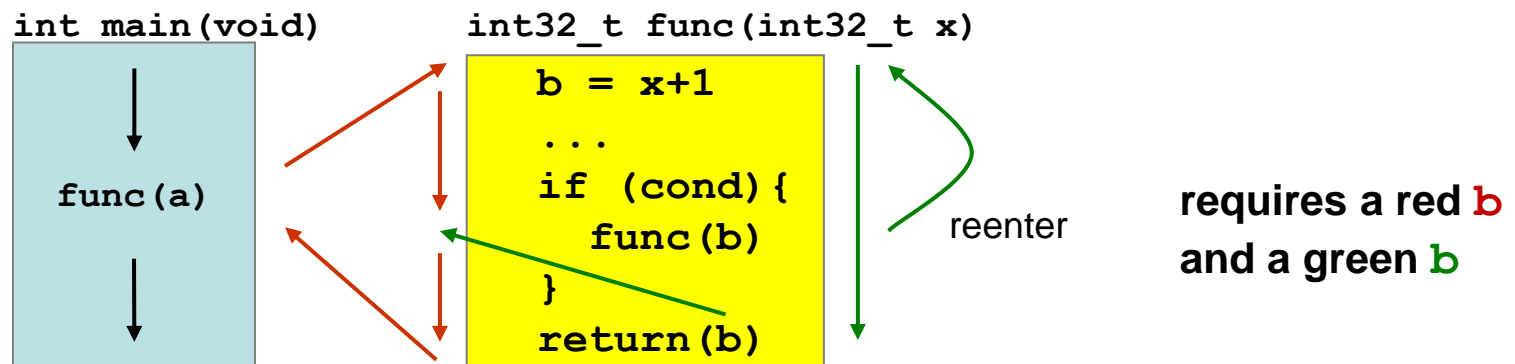
callee
function
double_g

- **Shared variables in data area**
 - **param1** Caller → procedure
 - **result** Return value
- **Overhead to access variable**
 - In Caller and Callee
- **Error-prone, unmaintainable**
 - No encapsulation,
 - Many dependencies
 - Multiple use of the same variable
 - Where is the variable written?
 - Who is allowed to read variable?
 - Requires unique variable names
 - Challenge if there is a large number of modules

Reentrancy

■ Recursive Function Calls?

- Registers and global variables are overwritten
- Requires an own set of data for each call
 - Parameters / Local variables



■ Solution

- Combined use of registers and stack for parameter passing
- See ARM Procedure Call Standard

- **Procedure Call Standard for the ARM architecture (AAPCS)**
 - http://infocenter.arm.com/help/topic/com.arm.doc.ihl0042e/IHL0042E_aapcs.pdf
 - Part of the ABI (Application Binary Interface) for the ARM Architecture
 - ABI → Specification to which independently produced relocatable object files must conform to be statically linkable¹⁾ and executable
 - Function calls
 - Parameter passing
 - Binary formats of information



source: colourbox

“The AAPCS defines how subroutines can be separately written, separately compiled, and separately assembled to work together. It describes a contract between a calling routine (caller) and a called routine (callee).”

→ Enables interaction of code produced by different compilers.

¹⁾ Linking will be covered later

■ AAPCS specifies

- Most of the items have already been covered in this course

Layout of data

- Size, alignment, layout of fundamental data types

Register Usage

- What are the registers used for

Memory Sections and Stack

- Code, read-only data, read-write data, stack, heap

Stack

- Full-descending, word-aligned, ...

Subroutine Calls

- Mechanism using LR and PC

Result Return

- Returning arguments through r0 (and r1 – r3)

Parameter Passing

- Passing arguments in r0-r3 and on stack

■ Register Usage

Register	Synonym	Role
r0	a1	Argument / result / scratch register 1
r1	a2	Argument / result / scratch register 2
r2	a3	Argument / scratch register 3
r3	a4	Argument / scratch register 4
r4	v1	Variable register 1
r5	v2	Variable register 2
r6	v3	Variable register 3
r7	v4	Variable register 4
r8	v5	Variable register 5
r9	v6	Variable register 6
r10	v7	Variable register 7
r11	v8	Variable register 8
r12	IP	Intra-Procedure-call scratch register ¹⁾
r13	SP	
r14	LR	
r15	PC	

Register contents might be modified by callee

Callee must preserve contents of these registers (Callee saved)

Cortex-M0: Registers r8 – r11 have limited set of instructions. Therefore, they are often not used by compilers.

■ Scratch Register

- Used to hold an intermediate value during a calculation
- Usually, such values are not named in the program source and have a limited lifetime

■ Variable Register

- A register used to hold the value of a variable, usually one local to a routine, and often named in the source code.
- Cortex-M0 registers R8 – R11 (v5 – v8) are often unused
 - as they are accessible only by few instructions

■ Argument, Parameter

- Used interchangeably
- Formal parameter of a subroutine

■ Parameters

- Caller copies arguments to R0 to R3
- Caller copies additional parameters to stack

■ Returning fundamental data types

- Smaller than word zero or sign extend to word; return in R0
- Word return in R0
- Double-word return in R0 / R1 ¹⁾
- 128-bit return in R0 – R3 ¹⁾

■ Returning composite data types (*structs, arrays, ...*)

- Up to 4 bytes return in R0
- Larger than 4 bytes stored in data area; address passed as extra argument at function call

¹⁾ Least significant word stored in lowest register

ARM Procedure Call Standard

■ Example

```
void caller(void)
{
    uint32_t p = 4;
    uint32_t q = 5;
    uint32_t r = 6;
    uint32_t sum;

    sum = callee(p,q,r);
}
```

```

MOVS    r4,#4
MOVS    r5,#5
MOVS    r6,#6    } local variables
                    R4 – R6

MOV     r2,r6
MOV     r1,r5
MOV     r0,r4    } copy parameters
                    to R0 – R2

BL      callee

MOV     r7,r0    } copy return value
                    to local variable
```

```
uint32_t callee(uint32_t a,
                uint32_t b,
                uint32_t c)
{
    return a + b + c;
}
```

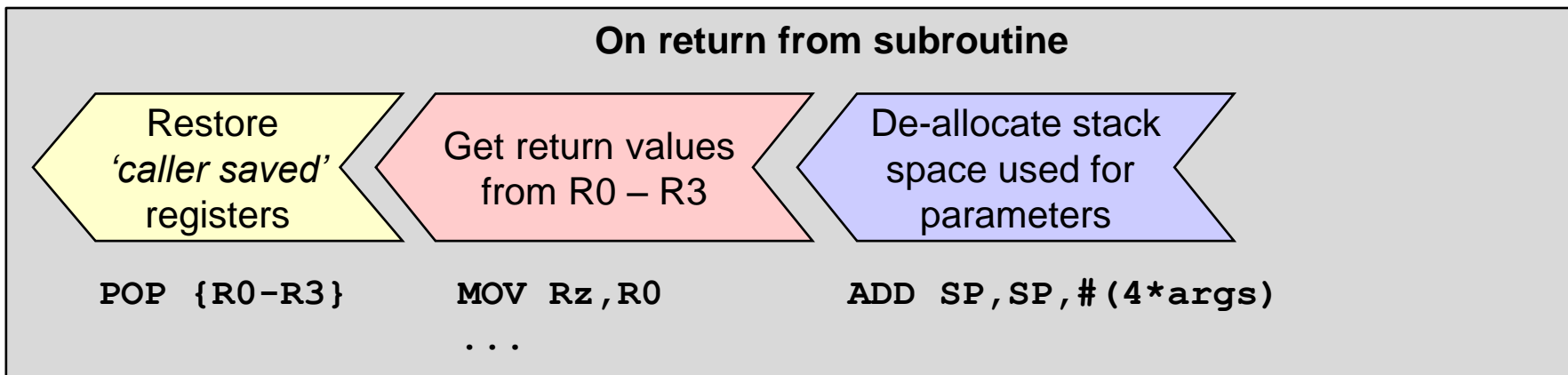
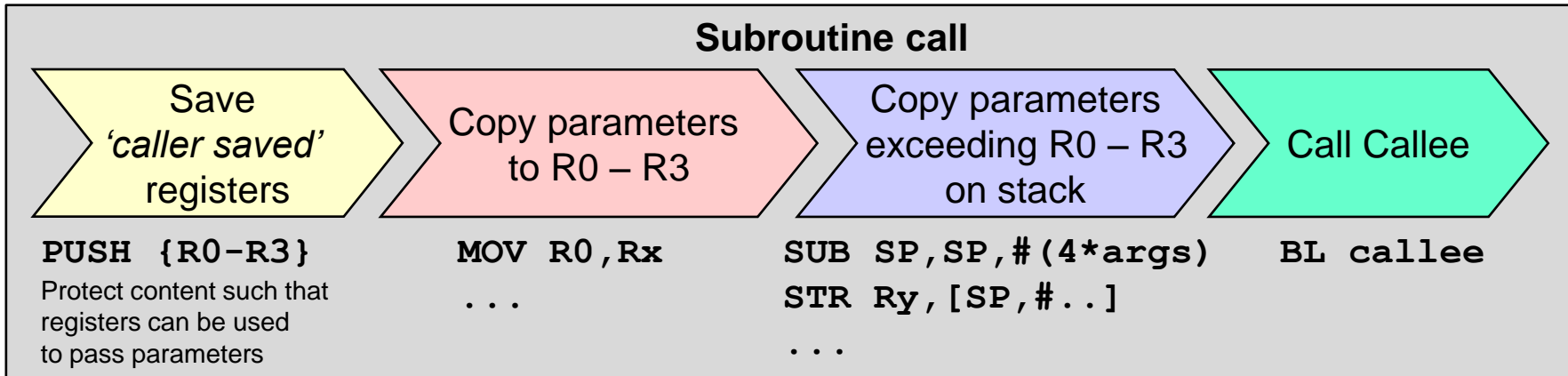
```

callee PROC
    ADDS    r0,r0,r1
    ADDS    r0,r0,r2    } callee uses
    BX      lr           } own copies
                        } of parameters
calleee ENDP
```

PUSH and POP are omitted in the example

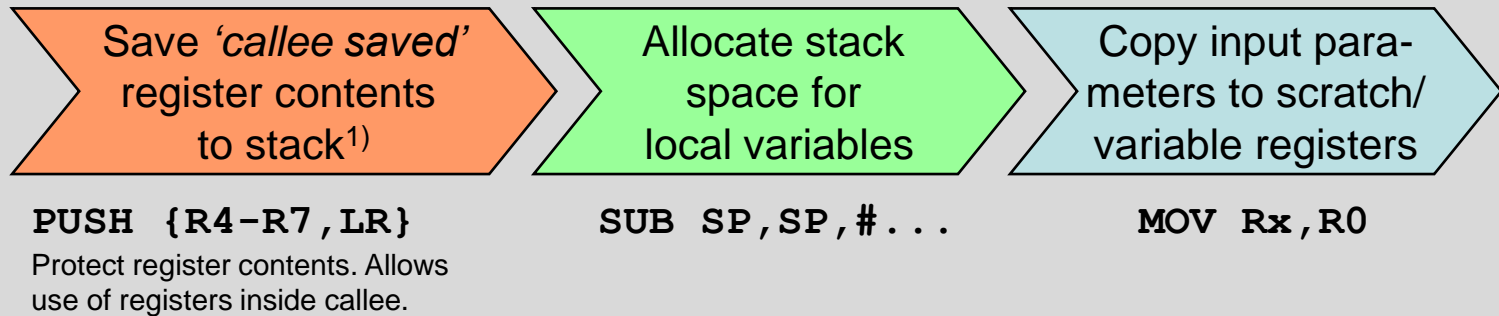
■ Subroutine Call – **Caller Side**

Pattern as used by the compiler. Manually written assembly code may be slightly different.

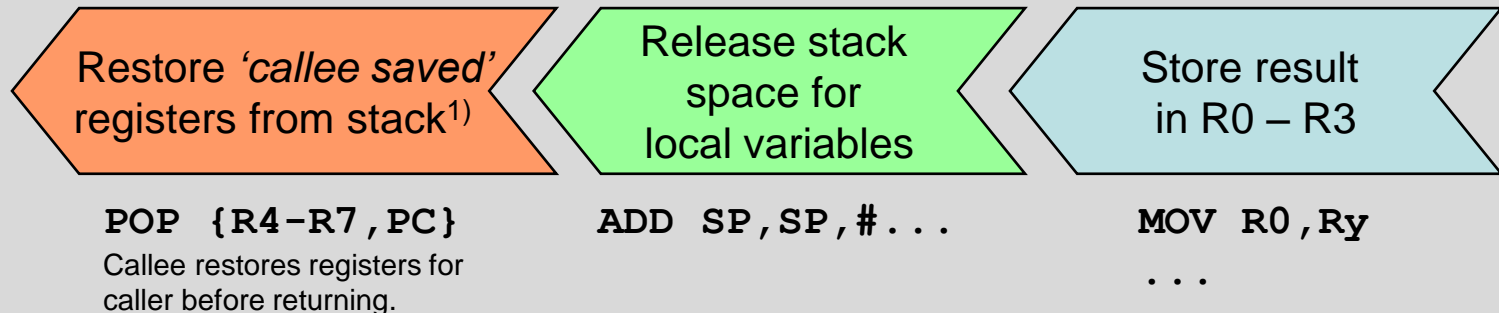


■ Subroutine Structure – Callee Side Pattern as used by the compiler. Manually written assembly code may be slightly different.

Prolog – Entry of subroutine

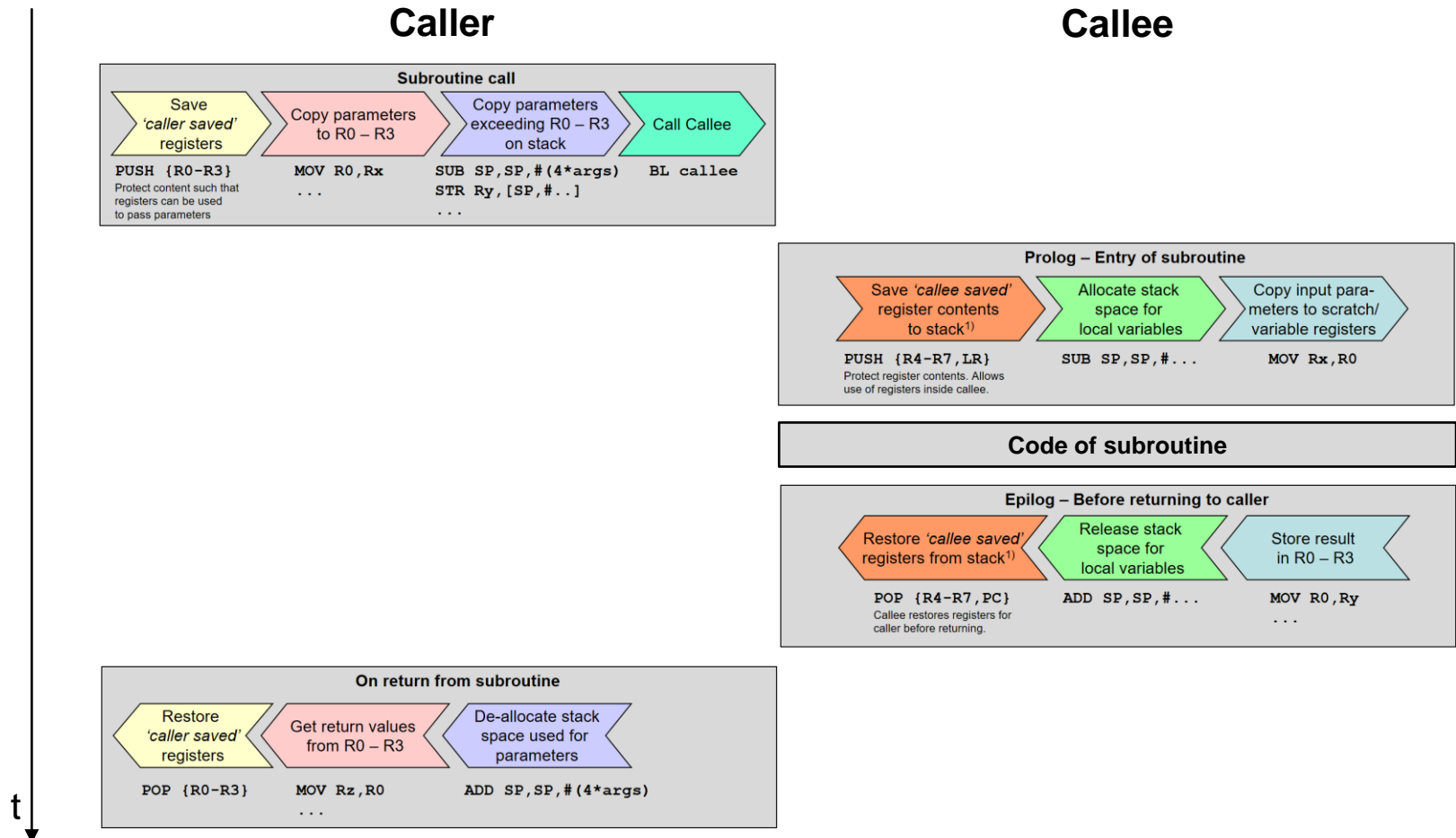


Epilog – Before returning to caller



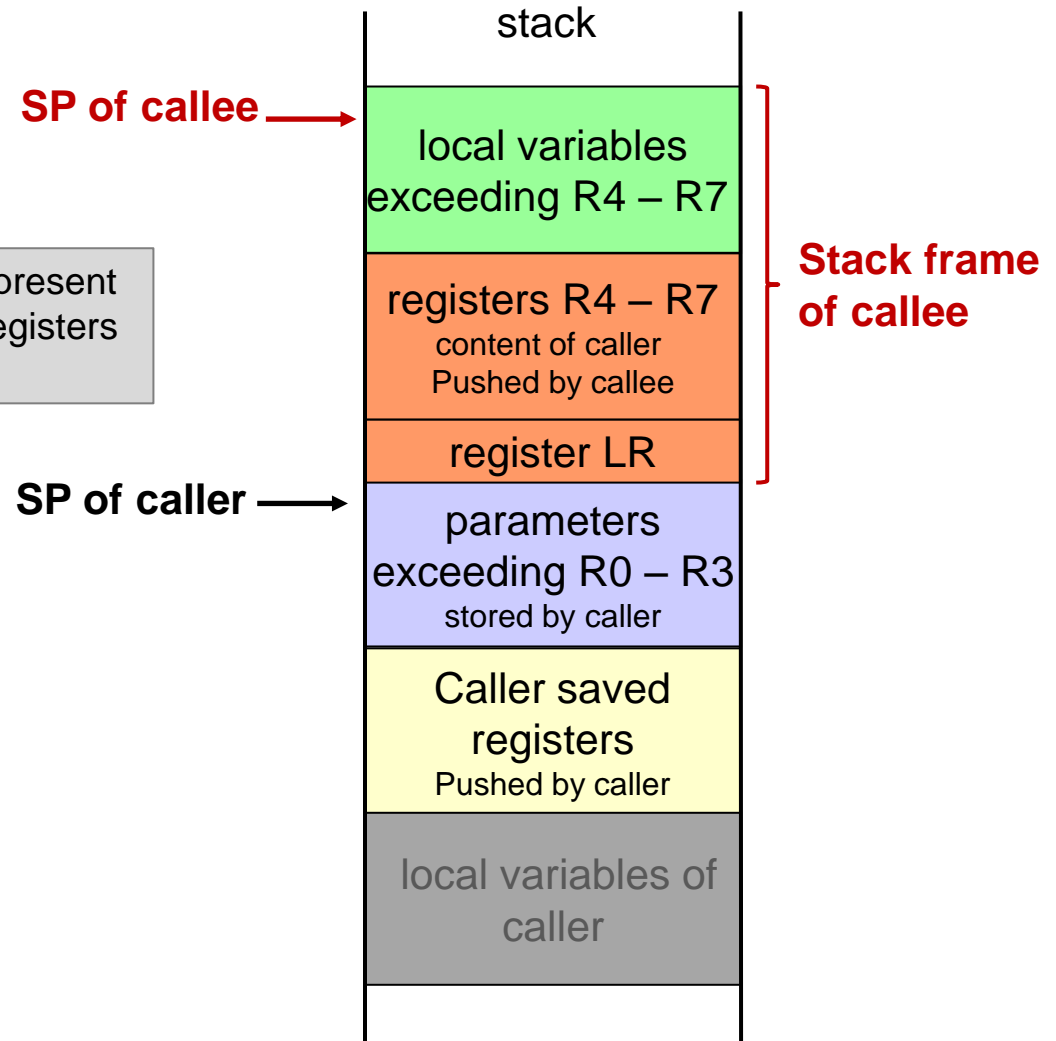
¹⁾ only registers modified by the callee are saved and restored. r8 – r11 are often unused by the compiler.

■ Summary – Flow of Execution for a Subroutine Call



■ Stack Frame

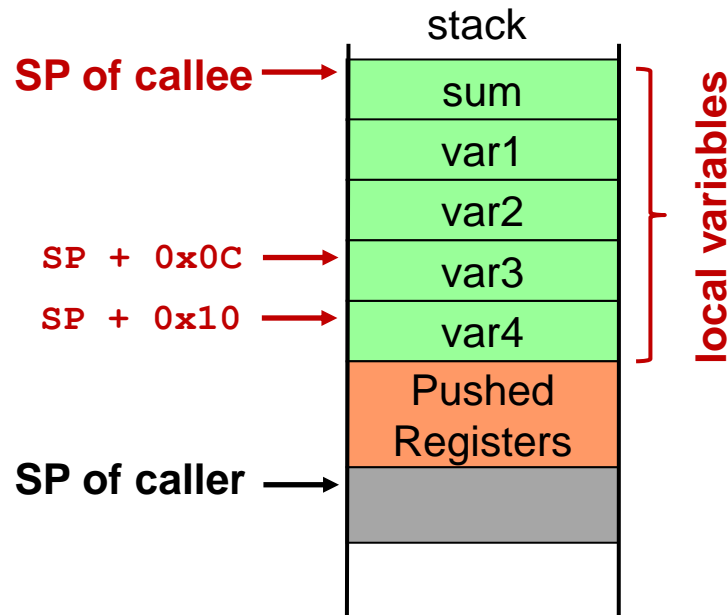
Each field may or may not be present depending on the number of registers required by the function



■ Access to Local Variables on Stack

- Example using word sized variables

Assume that the compiler has allocated the local variables as follows:



Compiler uses indirect addressing relative to stack pointer (SP)

```
sum = var3 + var4;
```

```
LDR    R0, [SP, #0x0c]    var3
```

```
LDR    R1, [SP, #0x10]    var4
```

```
ADDS   R0, R1, R0
```

```
STR    R0, [SP, #0x00]    sum
```

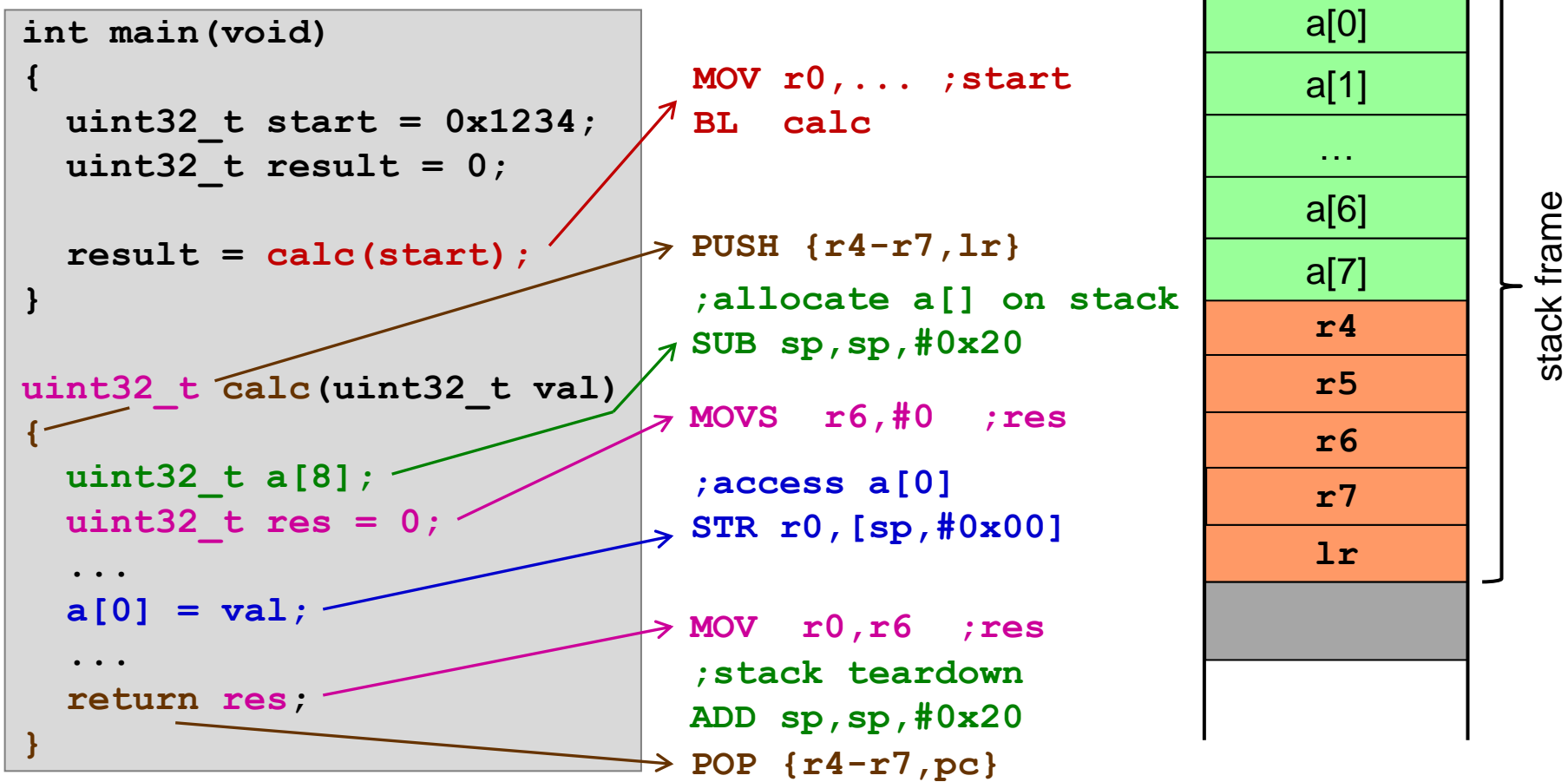
R0 and R1 used as scratch register

■ Functions in C

- Subroutine with return value (usually in register R0 / R1)
- Function Parameters
 - Always "pass by value"
 - Copy is being passed (not the original)
 - "pass by reference" only possible through use of pointers
→ Pointer itself passed by value
 - Registers R0 – R3
→ starting with the first argument in R0
 - Stack if more space is required
- Local Variables
 - In registers R4 – R7
 - On stack if more space is required or address operator (&) is used

Functions - Stack Frame

■ Example

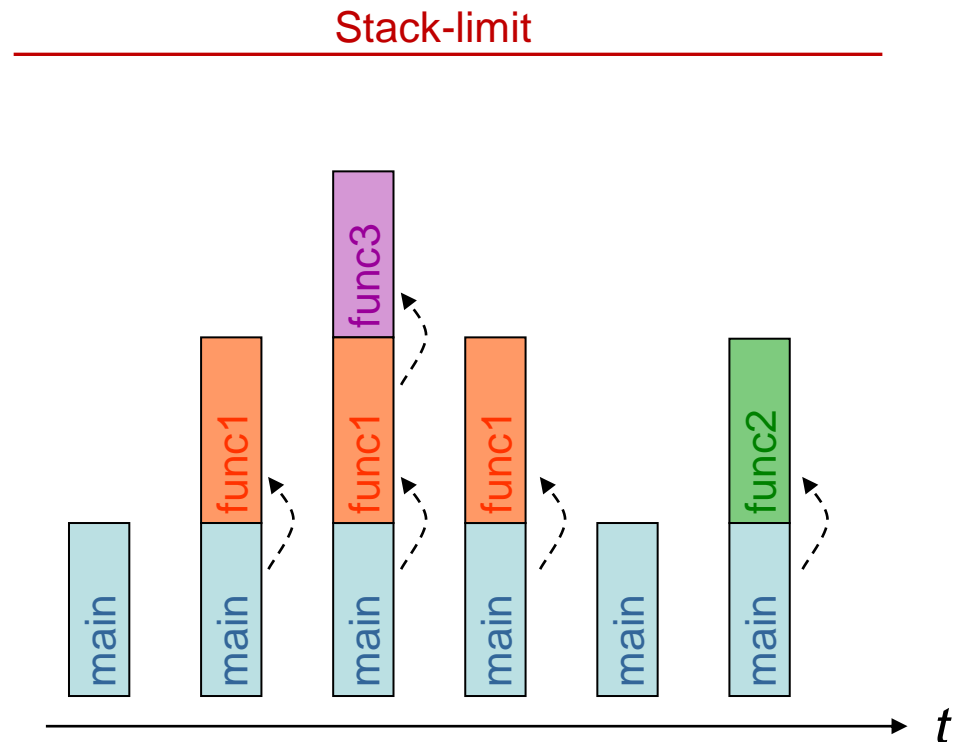


■ Build-up and tear-down of stack frames

```
int32_t main(void)
{
    int16_t x, y;
    x = func1(...);
    y = func2(...);
}

int16_t func1(...)
{
    z = func3(...);
    ...
}

int16_t func2(...) {...}
int16_t func3(...) {...}
```



Calling Assembly Subroutines from C

```
extern void strcpy(char *d, const char *s);  
int main(void)  
{  
    const char *srcstr = "First string ";  
    char dststr[] = "Second string";  
    strcpy(dststr,srcstr);  
    return (0);  
}
```

```
        PRESERVE8  
        AREA      SCopy, CODE, READONLY  
        EXPORT   strcpy  
  
        ; R0 points to destination string  
        ; R1 points to source string  
strcpy  LDRB R2, [R1]      ; Load byte and update address  
        ADDS R1, R1, #1  
        STRB R2, [R0]      ; Store byte and update address  
        ADDS R0, R0, #1  
        CMP  R2, #0        ; Check for null terminator  
        BNE  strcpy        ; Keep going if not  
        BX   LR            ; Return  
  
END
```

Example : from ARM Ltd.

■ Parameter Passing

- Through registers
- Through global variables
- On the stack (stack frame)

■ Procedure Call Standard

- R0 – R3 parameters / scratch → callee may modify these registers
- R4 – R7 local variables → callee restores potentially modified registers
- R0 – R3 Return

■ Stack Frame

- Nesting of subroutines
- Stack frame is constructed at function call
- Stack frame is deconstructed when function terminates (returns)

