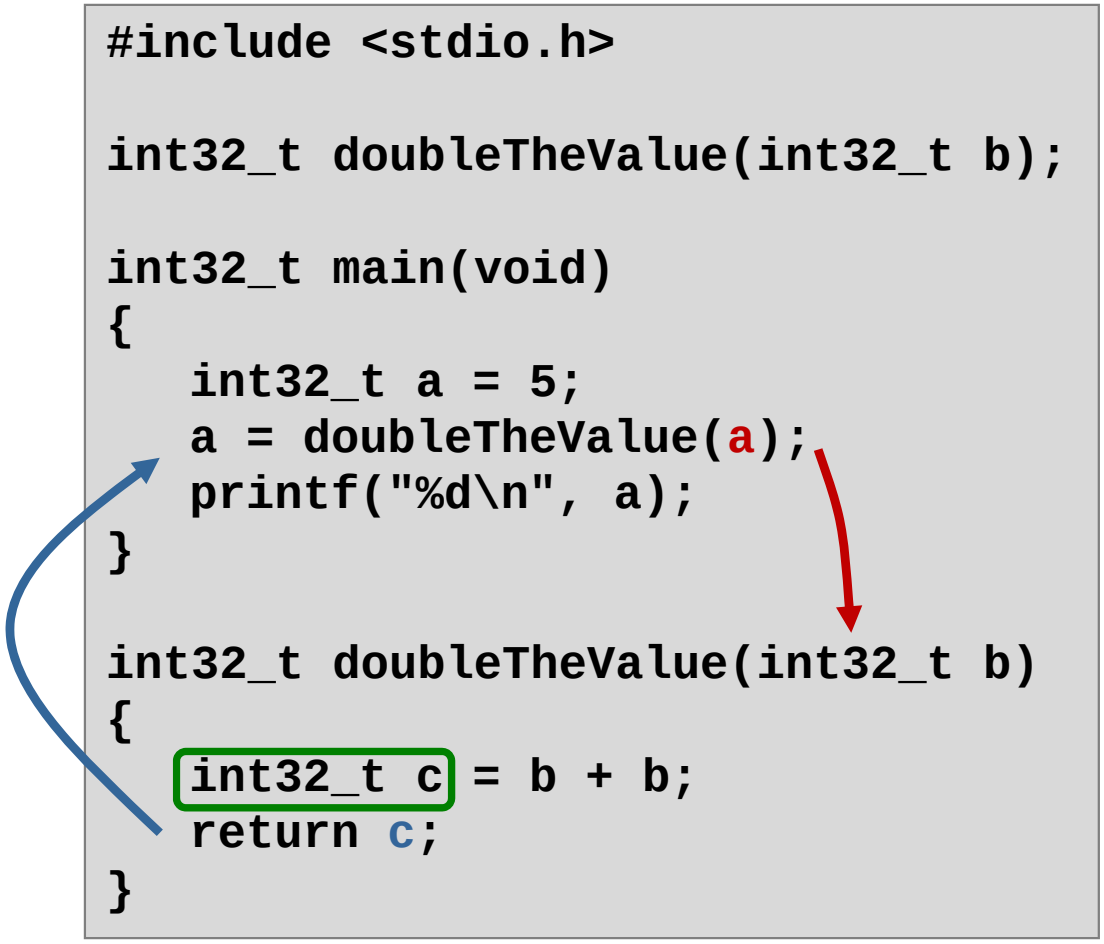
Parameter Passing

Computer Engineering 1

CT Team: A. Gieriet, J. Gruber, R. Gübeli, M. Meli, M. Rosenthal,
A. Rüst, J. Scheier, M. Thaler

Motivation



- 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()`**?

Agenda

School of Engineering
InES Institute of Embedded Systems

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

Learning Objectives

School of Engineering
InES Institute of Embedded Systems

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

Parameter Passing

School of Engineering
InES Institute of Embedded Systems

- **Where?**
  - **Register**
    - Caller and Callee[1)] 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

School of Engineering
InES Institute of Embedded Systems

Register / "pass by value"

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

AREA exCode,CODE,READONLY
   ...
   MOVS   R1,#0x03
   BL     double
   MOVS   ...,R0
   ...

double FUNCTION
   LSLS   R0,R1,#1
   BX     LR
   ENDFUNC
```

- **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?

Passing through Registers

Register / "pass by reference"

```
TABLE_LENGTH    EQU     16

AREA exData,DATA,READWRITE
p1Table SPACE TABLE_LENGTH

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

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

- **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

Passing through Global Variables

School of Engineering
InES Institute of Embedded Systems

Global Variables

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

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

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

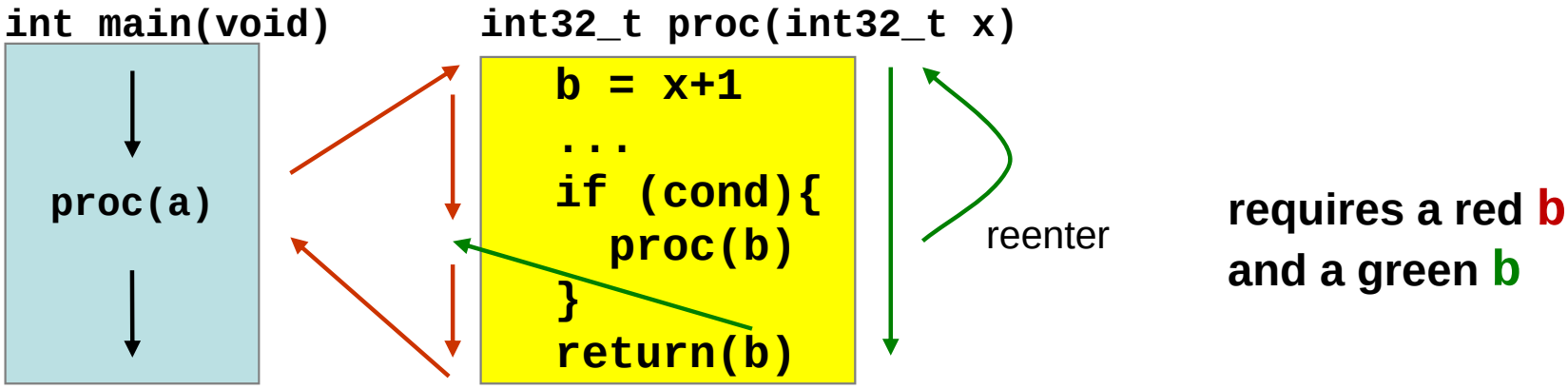
- **Shared variables in data area**
  - `param1` Caller → procedure
  - `result` Return value
- **High Overhead in Caller and Callee to access the variable**
- **Error-prone, unmaintainable**
  - No encapsulation,
  - Many dependencies
    - Multiple use of the same variable
    - Where is the variable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

Animation

Reentrancy

School of Engineering
InES Institute of Embedded Systems

- **Recursive Procedure 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

ARM Procedure Call Standard

School of Engineering
InES Institute of Embedded Systems

- **Procedure Call Standard for the ARM architecture**
  - Part of the ABI for the ARM Architecture
  - http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf [AAPCS]
- **ABI – Application Binary Interface**
  - Specification to which independently produced relocatable object files must conform to be statically linkable[1)] and executable
    - Function calls
    - Parameter passing
    - In which binary format should information be passed

source: colourbox

- **EABI**
  - An ABI suited to the needs of embedded applications

[1)] Linking will be covered in CT2

ARM Procedure Call Standard

School of Engineering
InES Institute of Embedded Systems

- **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

ARM Procedure Call Standard

School of Engineering
InES Institute of Embedded Systems

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)

[1)] used by the linker

ARM Procedure Call Standard

School of Engineering
InES Institute of Embedded Systems

- **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

ARM Procedure Call Standard

School of Engineering
InES Institute of Embedded Systems

- **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

Animation

ARM Procedure Call Standard

School of Engineering
InES Institute of Embedded Systems

Example

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

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

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

```
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

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

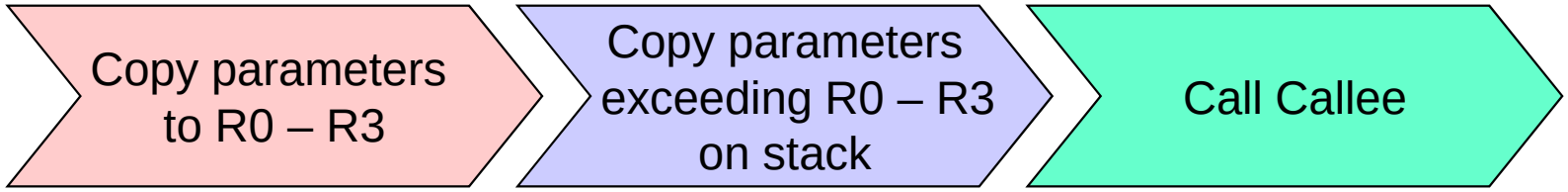
PUSH and POP are omitted in the example

ARM Procedure Call Standard

School of Engineering
InES Institute of Embedded Systems

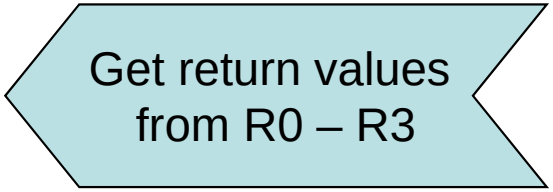
Subroutine Call – Caller Side

- Subroutine call




```
MOV R0,R4          SUB SP,SP,#(4*args)     BL callee
...                STR R4,[SP,#..]
                   ...
```

- On return from subroutine

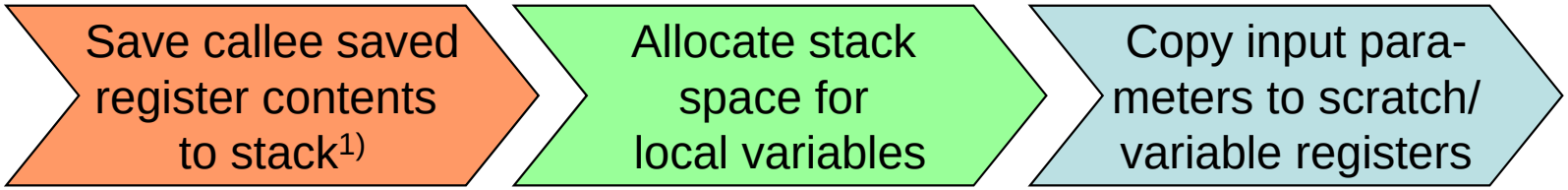



```
MOV R4,R0
ADD SP,SP,#(4*args)
...
```

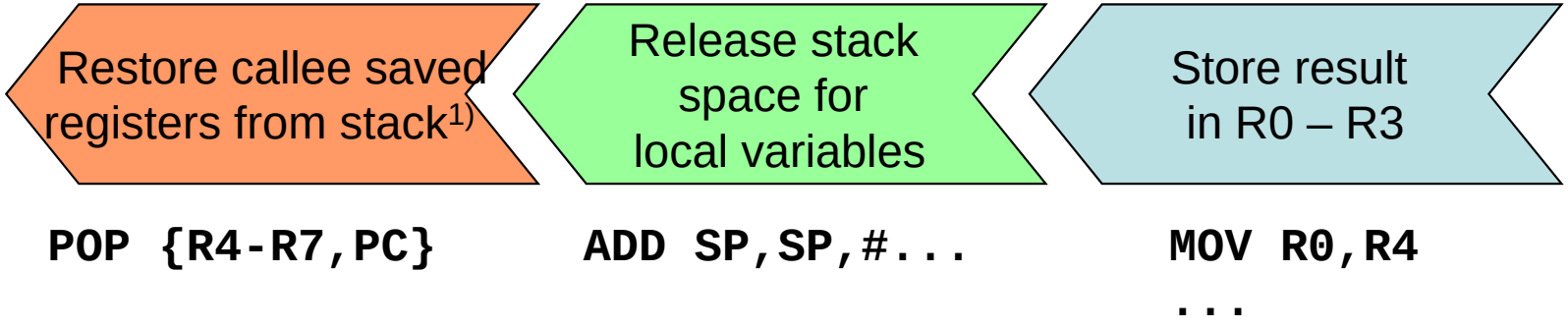
ARM Procedure Call Standard

Subroutine Structure – Callee Side

- Prolog – Entry of subroutine



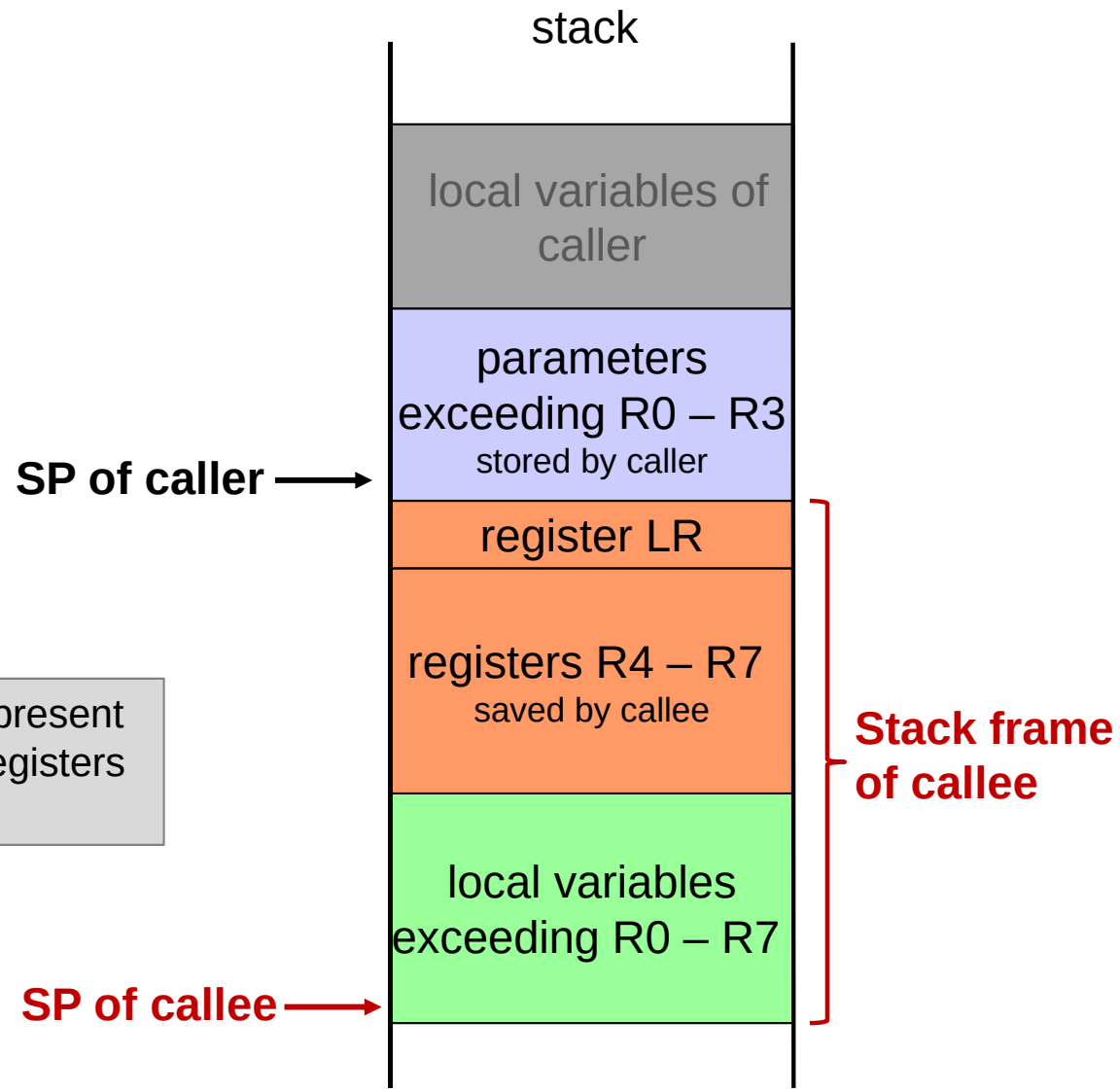
- Epilog – Before returning to caller



[1] only registers modified by the callee are saved and restored

Functions - Stack Frame

- **Stack Frame**



Functions – Stack Frame

Access to Local Variables on Stack

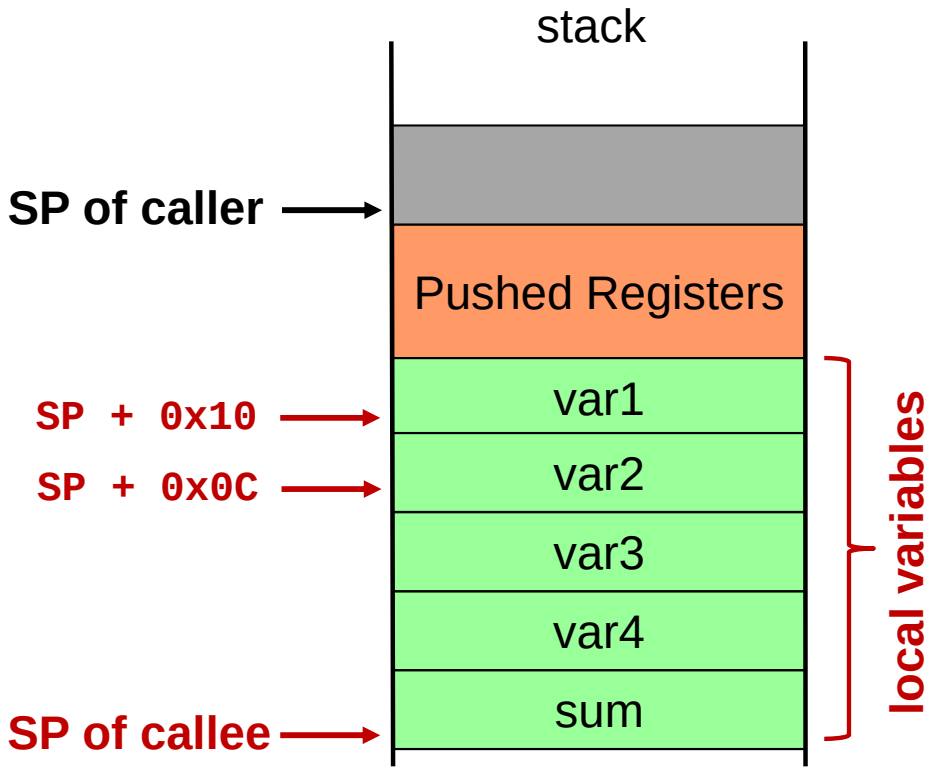
- Example using word sized variables

```
sum = var1 + var2;
```

```
LDR   R0, [SP, #0x10]

LDR   R1, [SP, #0x0c]
ADDS  R0, R1, R0

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



R0 and R1 used as scratch register

Functions - Stack Frame

School of Engineering
InES Institute of Embedded Systems

- **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

Animation

Functions - Stack Frame

School of Engineering
InES Institute of Embedded Systems

- **Example**

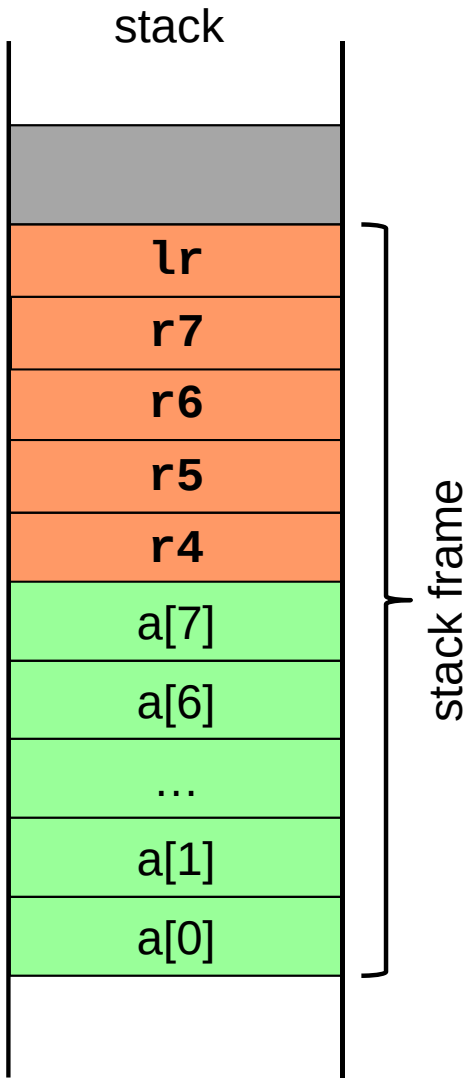
```
int main(void)
{
  uint32_t start = 0x1234;
  uint32_t result = 0;

  result = calc(start);
}

uint32_t calc(uint32_t val)
{
  uint32_t a[8];
  uint32_t res = 0;
  ...
  a[0] = val;
  ...
  return res;
}
```

```
MOV r0,... ;start
BL  calc

PUSH {r4-r7,lr}
;allocate a[] on stack
SUB sp,sp,#0x20
MOVS  r6,#0  ;res
;access a[0]
STR r0,[sp,#0x00]
MOV  r0,r6  ;res
;stack teardown
ADD sp,sp,#0x20
POP {r4-r7,pc}
```



Functions - Stack Frame

School of Engineering
InES Institute of Embedded Systems

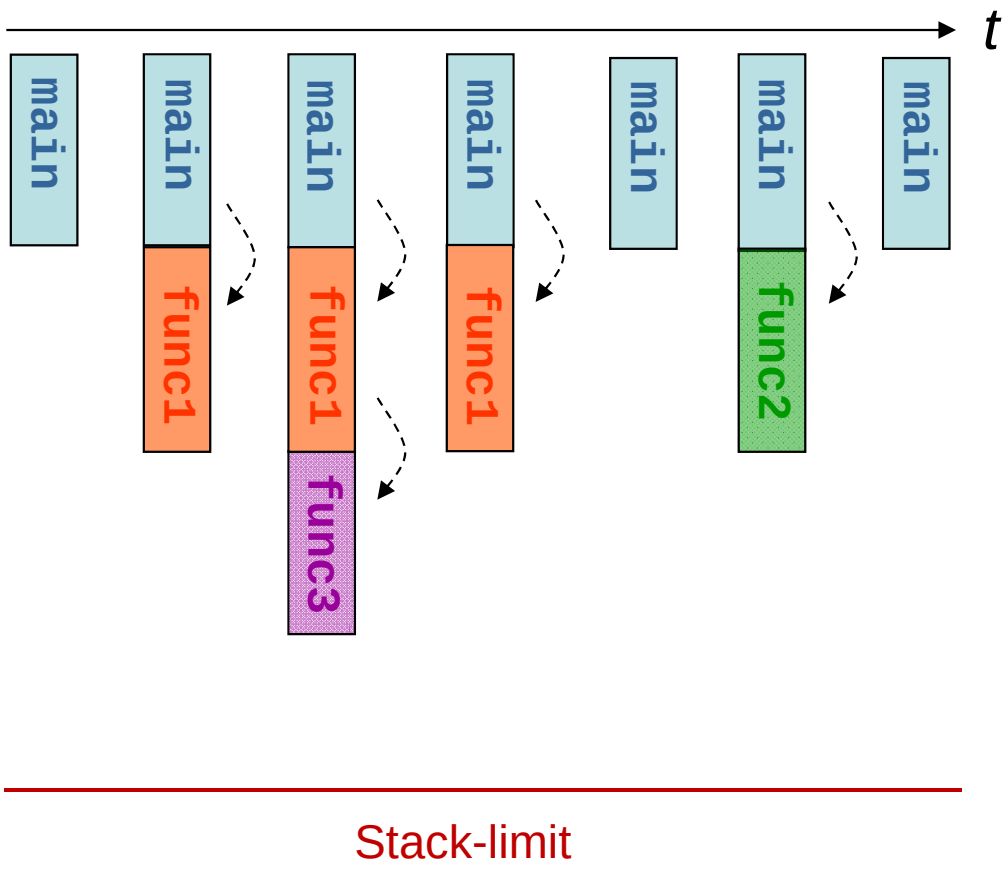
- **Build-up and tear-down of stack frames**

```
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

School of Engineering
InES Institute of Embedded Systems

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

```
            PRESERVE8
            AREA    SCopy, CODE, READONLY
            EXPORT strcopy

            ; R0 points to destination string
            ; R1 points to source string
strcopy     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  strcopy      ; Keep going if not
            BX   LR           ; Return
 END
```

Example : from ARM Ltd.

Conclusions

- **Parameter Passing**
  - Through registers
  - Through global variables
  - On the stack (stack frame)
- **Procedure Call Standard**
  - R0 – R3 parameters / scratch → callee might 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 removed when function terminates (returns)