UNIVERSITEIT
iYUNIVESITHI
STELLENBOSCH
UNIVERSITY

100

1918 · 2018

forward together · saam vorentoe · masiye phambili

Computer Systems / Rekenaarstelsels 245 - 2020

Lecture 4

Functions and the Stack
Funksies en die Stapel

Dr Rensu Theart & Dr Lourens Visagie

Lecture Overview

- Function calls
  - Branching
- The stack
  - Loading and storing multiple registers to memory at once

Function calls / Funksie roepe

- High-level languages support **functions** (also called procedures or subroutines) to reuse common code and to make a program more modular and readable.
- Functions have inputs, called **arguments/parameters**, and an output, called the **return value**.
- Functions should calculate the return value and cause no other unintended side effects to other parts of the program.

arguments

```c
int add(int x, int y)
{
    return x + y;
}
```

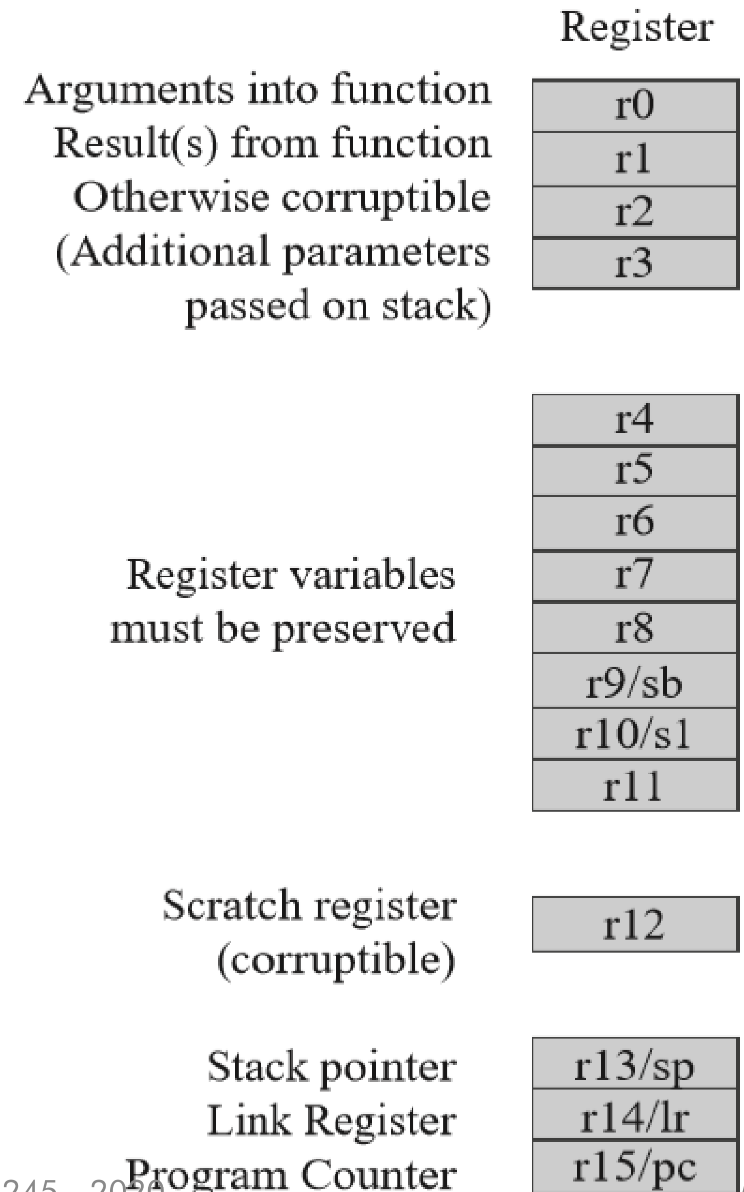
return value

What is the difference between an argument and a parameter in C?

- A **parameter** is the variable in the function definition.
- **Arguments** are the data you pass into the function's parameters.

Function calls / Funksie roepe

- When writing function in ARM you must follow the **AAPCS** (Arm Architecture Procedure Call Standard).
  - Describes contract between calling function and called function.
- R0 to R3 used as function arguments (parameters) and return value.
  - Often only use R0 for return value.
  - A double-word sized type is passed in two consecutive registers (e.g., R0 and R1 or R2 and R3).
- If there are more function arguments, use the stack to pass them.
- A function must preserve R4-R8, R10, R11, and SP – their content must be the same when the function is called and when the function returns.
  - Preserving SP: in general this means matched PUSH and POP instructions
- ARM Compiler uses a **full descending stack**



Branching / Vertakking

- For non-branching instructions (`ADD`, `ORR`, `LDR`, etc.) the **program counter (PC)** simply increments by 4 after each instruction. There is a sequential processing of instructions.
  - (Recall that instructions are 4 bytes long and ARM is a byteaddressed architecture.)
- 'Branching' means the sequential program execution is broken, and program flow will resume from somewhere else (the instruction to which the branch instruction is directed).
  - Specifically, <u>the Program Counter is changed</u>, to contain the address of the next instruction to process
- Branch instructions take a **label** (a text name) that it branches to, which is simply a name associated with a certain PC address.
  E.g. `B  TARGET`
- **NOTE**: labels cannot be reserved words, such as instruction mnemonics.

Branching and instruction cycle

- What happens when the processor is told to break execution with a branch instruction? (The CPU doesn't know beforehand if the branch conditions will true – i.e. will it branch or not)

Cycle				1	2	3	4	5
Address	Operation							
0x8000	B	Fetch	Decode	Execute				
0x8004	X		Fetch	Decode				
0x8008	XX			Fetch				
0x8FEC	ADD				Fetch	Decode	Execute	
0x8FF0	SUB					Fetch	Decode	Execute
0x8FF4	MOV						Fetch	Decode
								Fetch

⇒ Branching has an impact on performance, especially if there are many steps in the processing pipeline.

Function calls and returns

- ARM uses the **branch and link** instruction (BL) to call a function. To return from a function the **link register** is moved to the PC
  ⇒ `MOV PC, LR`
  - Recall that the program counter (PC) is the memory location of the current instruction being executed.
- In this code the main function calls the simple function.
- The simple function is called with no input arguments and generates no return value; it just returns to the caller.

High-Level Code

```c
int main()
{
    simple();
    ...
}
// void means the function returns no value
void simple()
{
    return;
}
```

Instruction addresses

ARM Assembly Code

```
0x00008000 MAIN     ...
...                 ...
0x00008020          BL   SIMPLE     ; call the simple function
...


0x0000902C SIMPLE   MOV PC, LR      ; return
```

Function calls and returns (2)

- `BL` (branch and link) and `MOV PC, LR` are the two essential instructions needed for a function call and return.
- `BL` performs two tasks: it stores the **return address** of the next instruction (the instruction after `BL`) in the link register (LR), and it branches to the target instruction.
- In the previous slide, the main function calls the simple function by executing the branch and link instruction (`BL`).
- `BL` branches to the `SIMPLE` label and stores 0x00008024 in LR.
- The simple function returns immediately by executing the instruction `MOV PC, LR`, copying the return address from the LR back to the PC.
- The main function then continues executing at this address (0x00008024).

Input arguments and return values

- According to ARM convention, the calling function, `main`, places the function arguments from left to right into the input registers, R0–R3.
- The called function, `diffofsums`, stores the return value in the return register, R0.
- When a function with more than four arguments is called, the additional input arguments are placed on the stack.

High-level code

```
int main()
{
    int y;
    y = diffofsums(5, 4, 3, 2);
    ...
}

int diffofsums(int f, int g, int h, int i)
{
    int result;
    result = (f + g) - (h + i);
    return result;
}
```

Assembly code

```
          ;          R4 = y
MAIN

          MOV        R0, #5 ; argument 0 = 5
          MOV        R1, #4 ; argument 1 = 4
          MOV        R2, #3 ; argument 2 = 3
          MOV        R3, #2 ; argument 3 = 2
          BL         DIFFOFSUMS ; call function
          MOV        R4, R0 ; y = returned value
          BL         DONE
          ;          R4 = result
DIFFOFSUMS
          ADD        R0, R0, R1 ; R8 = f + g
          ADD        R2, R2, R3 ; R9 = h + i
          SUB        R0, R0, R2 ; result = (f + g) - (h + i)
          MOV        PC, LR ; return to caller
DONE
```

Types of branching / Tipes vertakking

Assembly mnemonic	Branch type	Meaning
B	Branch	Simple branch to address, optionally making use of condition flags (i.e. BGT)
BX	Branch and Exchange	Branch to an address that is stored in a register
BL	Branch with Link	Store the current Program Counter in the Link Register, prior to branching to the new address (so that a sub-routine can return to the same point in the program)
BLX	Branch with Link and Exchange	Same as BL but the address to which to branch is held in a register
CBZ	Compare and branch if zero	Combines the CMP and BEQ instructions (i.e. CMP r0, #0 + BEQ label)
CBNZ	Compare and branch if non-zero	Combines the CMP and BNE instuctions
IT-blocks	If-Then blocks	Avoids branching for up to 4 instructions following the IF comparison

Branch and Exchange / Vertak en ruil om

- **Branch and Exchange** (BX), branches to an address contained in a register, i.e.

  `BX R5`

- Typically used with link register, to return from a function

  ⇒ `MOV PC, LR` then simply becomes `BX LR`

- (Note: this instruciton does not work in VisUAL)

Branching: Machine code / Vertakking: Masjien Kode

- For the Thumb instruction set there are various machine code implementations of Branch.
- The *imm* fields refers to the offset that is added to the Program Counter which changes the next instruction that will be fetched to be executed.
- You can also give a register offset by using BX.
- Branch accomplishes the same instructions such as:
  - `MOV PC, #0x20000000`, or `LDR PC,=0xBE000000`

B

Encoding T1 All versions of the Thumb instruction set.

`B<c> <label>` Not permitted in IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	cond				imm8							

Encoding T2 All versions of the Thumb instruction set.

`B<c> <label>` Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	imm11										

Encoding T3 ARMv7-M

`B<c>.W <label>` Not permitted in IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	cond				imm6						1	0	J1	0	J2	imm11										

Encoding T4 ARMv7-M

`B<c>.W <label>` Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	imm10										1	0	J1	1	J2	imm11										

BX

Encoding T1 All versions of the Thumb instruction set.

`BX<c> <Rm>` Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	0	Rm				(0)	(0)	(0)

Branching: Machine code / Vertakking: Masjien Kode

BL

Encoding T1 All versions of the Thumb instruction set.

BL<c> <label> Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	imm10										1	1	J1	1	J2	imm11										

- Permitted offsets are even numbers in the range -16777216 to 16777214. For other ranges use a register with BLX.

BLX (register)

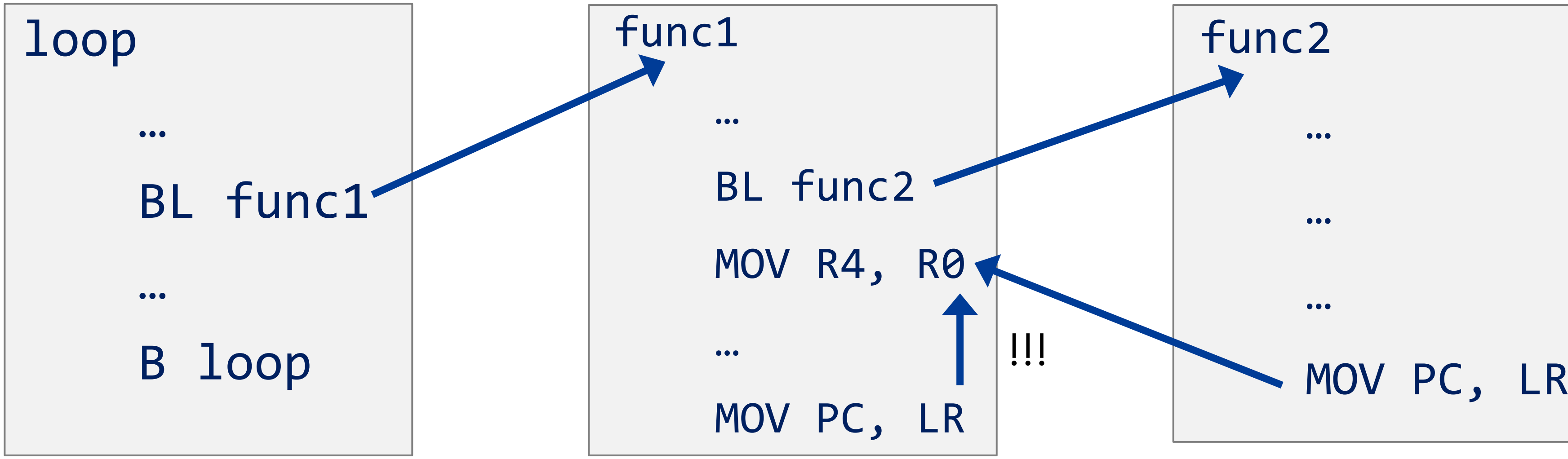
Encoding T1 ARMv5T*, ARMv6-M, ARMv7-M

BLX<c> <Rm> Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	1	Rm				(0)	(0)	(0)

Branching and C functions / Vertakking en C funksies

- Calling a function from another function (called **nonleaf functions**):



- Main loop calls function `func1`, which calls another function, `func2`
- Function execution is implemented using `BL` (branch and save PC of next instruction to Link Register), and return from function by restoring PC to contents of saved LR
- The second `BL func2` will overwrite the Link Register (and probably R0-R3). `func1` now does not know where to return to anymore…

  ⇒ Functions must preserve some register states using the Stack.

Branching and C functions / Vertakking en C funksies

- For recursive functions the LR and arguments must repeatedly be stored to the stack.

```
int factorial(int n)
{
    if (n <= 1)
        return 1;
    else
        return (n * factorial(n - 1));
}
```

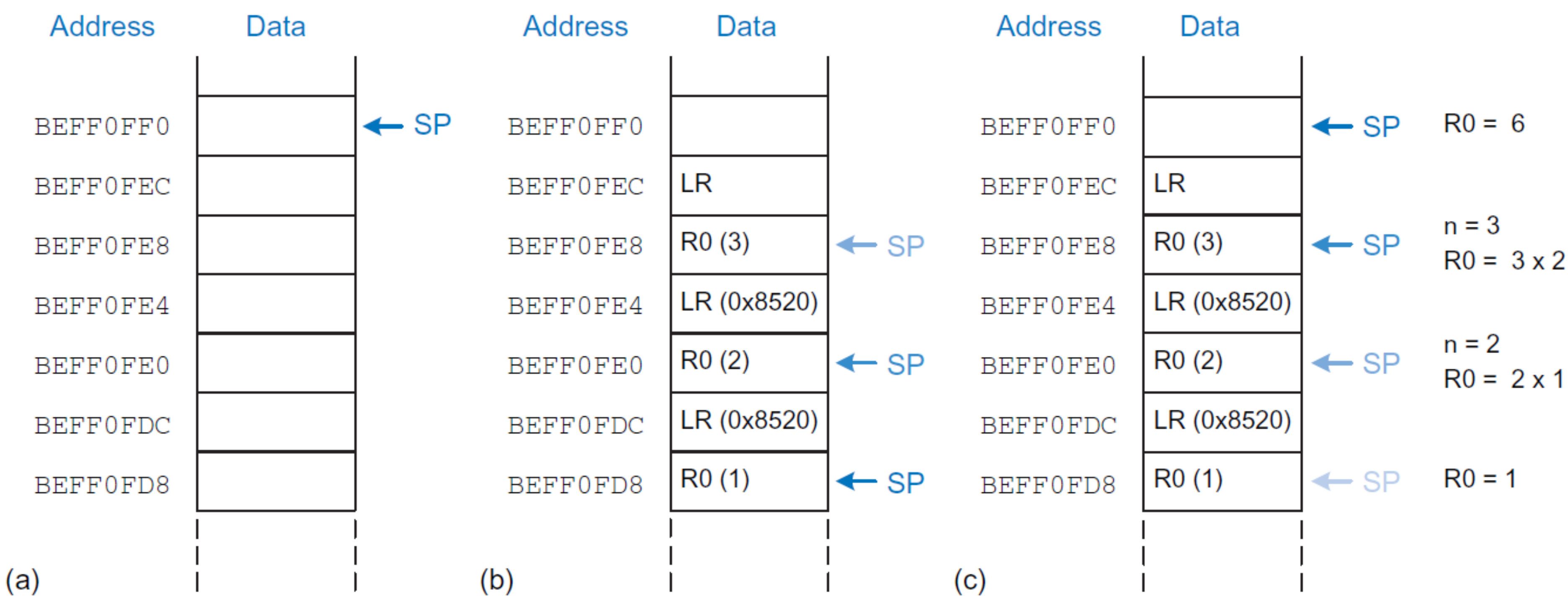


Figure 6.14 Stack: (a) before, (b) during, and (c) after factorial function call with $n = 3$

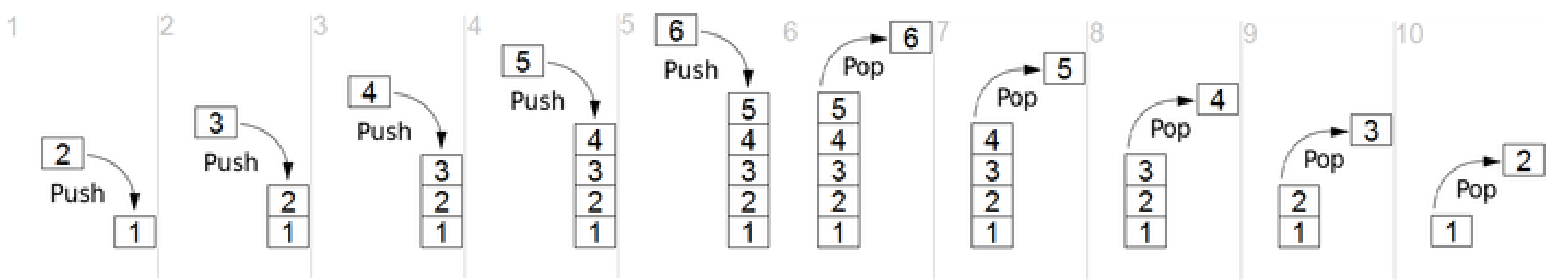
The stack / Die stapel

We need the stack to store:

- Function arguments that does not fit into R0-R3
- R4-R8, R10, R11, and SP within the called function if we wish to use any of them for scratch space and should be restored before the function returns.
- Local function variables that does not fit into the available registers.
- The processor state (register values) before calling a function, or interrupt handler execution.
  - e.g. LR before a function call, to ensure we do not overwrite the LR that is necessary to return the PC to the correct address.

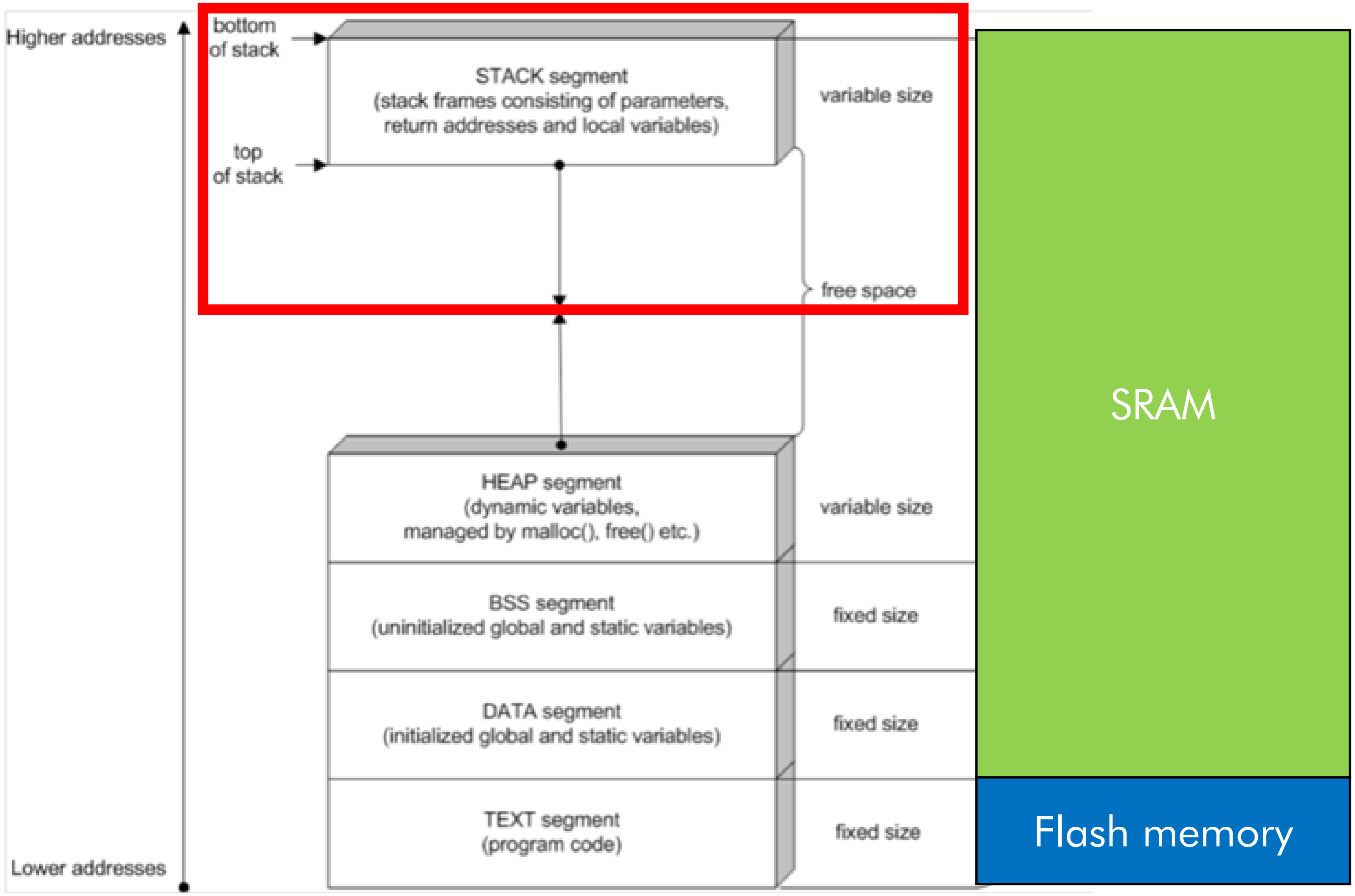
The stack / Die stapel

- **The stack** is a section in volatile memory (RAM) that is used for passing parameters/saving state between function calls and interrupts.
- Function may also allocate stack space to store local variables but must deallocate it before returning.
- The stack <u>expands</u> (uses more memory) as the processor needs more "scratch space" and <u>contracts</u> (uses less memory) when the processor no longer needs the variables stored there.
- The stack is a last-in-first-out (LIFO) queue. Like a stack of dishes, the last item **pushed** onto the stack (the top dish) is the first one that can be **popped** off.
- The top of the stack is the most recently allocated space.



The stack / Die stapel

Memory layout of our microcontroller



The stack pointer / Die staplewyser

- The **stack pointer, SP** (R13), is an ordinary ARM register that, by convention, points to the top of the stack from where elements are removed or added.
  - SP simply stores the memory address of the top of the stack.
  - It starts at a high memory address and decrements to expand as needed.
- In example (a) below, the stack pointer, SP, holds the address value 0XBEFFFAE8 and points to the data value 0xAB000001.
- Example (b) shows the stack expanding (by decrementing by 8) to allow two more data words of temporary storage.

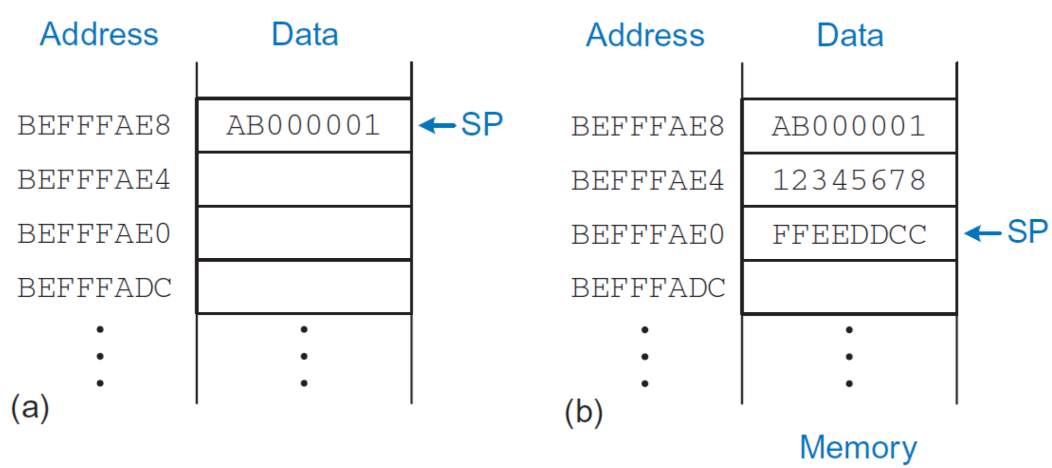


Figure 6.11 The stack (a) before expansion and (b) after two-word expansion

Stack / Stapel

- On the ARM Cortex M4 each element in the stack is a 32-bit word
- The ARMv7-M architecture uses a **full descending stack** which means:
  - When **pushing**, the hardware decrements the stack pointer to the end of the new stack frame before it stores data onto the stack.
  - When **popping**, the hardware reads the data from the stack frame and then increments the stack pointer.

```
void push(int dataValue)
{
    stackPointer--;
    *stackPointer = dataValue;
}

void pop(int *dataLocation)
{
    *dataLocation = *stackPointer;
    stackPointer++;
}
```

Descending	**Stack Pointer decreases as stack grows (stack grows downwards from high to low addresses)**
Ascending	Stack Pointer increases as stack grows (stack grows upwards from low to high addresses)
Full	**SP points to the last item on the stack**
Empty	SP points to the next free space on the stack

Stack / Stapel

Full descending stack

- Stack base address is 0x00080000
- SP is 0x0007FFDC (9 elements)
- After a push operation (add value 0x83596900 onto the stack):
  - SP is **decremented** first (new SP = 0x7FFD8)
  - **Store** new word to 0x7FFD8
  - Stack now has 10 elements

	Offset(h)	00	01	02	03
	0007FFC8	1A	1A	78	04
	0007FFCC	F4	0A	27	00
	0007FFD0	75	03	02	40
	0007FFD4	24	00	9E	FC
	0007FFD8	83	59	69	00
SP →	0007FFDC	11	00	49	FF
	0007FFE0	F9	FF	38	FF
	0007FFE4	C2	FC	38	FF
	0007FFE8	00	00	CF	F0
	0007FFEC	00	00	5F	00
	0007FFF0	14	00	21	00
	0007FFF4	00	00	5A	04
	0007FFF8	00	00	00	00
	0007FFFC	48	03	00	00
	00080000	00	00	5E	01
	00080004	00	00	14	9C

Stack / Stapel

Full descending stack

- Stack base address is 0x00080000
- SP is 0x0007FFDC (9 elements)
- After a pop operation (remove a previous value from the stack)
  - **Load** word at current SP address (0x7FFDC)
    - Get 0x110049FF
  - **Increment** SP (new SP is 0x7FFE0)
  - Stack now has 8 elements

	Offset(h)	00	01	02	03
	0007FFC8	1A	1A	78	04
	0007FFCC	F4	0A	27	00
	0007FFD0	75	03	02	40
	0007FFD4	24	00	9E	FC
	0007FFD8	DF	02	FE	00
SP →	0007FFDC	11	00	49	FF
	0007FFE0	F9	FF	38	FF
	0007FFE4	C2	FC	38	FF
	0007FFE8	00	00	CF	F0
	0007FFEC	00	00	5F	00
	0007FFF0	14	00	21	00
	0007FFF4	00	00	5A	04
	0007FFF8	00	00	00	00
	0007FFFC	48	03	00	00
	00080000	00	00	5E	01
	00080004	00	00	14	9C

- What is the value of SP when the stack is empty?
  - 0x00080000

Stack Example / Stapel Voorbeeld

- One of the important uses of the stack is to save and restore registers that are used by a function. A function should calculate a return value but have no other unintended side effects.
- The assembly function below violates this rule since it modifies R4, R8 and R9.

```c
int main()
{
   int y;
   . . .
   y = diffofsums(2, 3, 4, 5);
   . . .
}

int diffofsums(int f, int g, int h, int i)
{
   int result;
   result = (f + g) - (h + i);
   return result;
}
```

```
                ;       R4 = y
MAIN
                ...
                MOV     R0, #2 ; argument 0 = 2
                MOV     R1, #3 ; argument 1 = 3
                MOV     R2, #4 ; argument 2 = 4
                MOV     R3, #5 ; argument 3 = 5
                BL      DIFFOFSUMS ; call function
                MOV     R4, R0 ; y = returned value
                ///
                ;               R4 = result
DIFFOFSUMS
                ADD     R8, R0, R1 ; R8 = f + g
                ADD     R9, R2, R3 ; R9 = h + i
                SUB     R4, R8, R9 ; result = (f + g) - (h + i)
                MOV     R0, R4 ; put return value in R0
                MOV     PC, LR ; return to caller
```

Stack Example / Stapel Voorbeeld

- To solve this problem, a function saves registers on the stack before it modifies them, then restores them from the stack before it returns.
- Specifically, it performs the following steps:
  1. Makes space on the stack to store the values of one or more registers
  2. Stores the values of the registers on the stack
  3. Executes the function using the registers
  4. Restores the original values of the registers from the stack
  5. Deallocates space on the stack

```
        ;R4       = result
DIFFOFSUMS
        SUB   SP, SP, #12 ; make space on stack for 3 registers
        STR   R9, [SP, #8] ; save R9 on stack
        STR   R8, [SP, #4] ; save R8 on stack
        STR   R4, [SP] ; save R4 on stack
        ADD   R8, R0, R1 ; R8 = f + g
        ADD   R9, R2, R3 ; R9 = h + i
        SUB   R4, R8, R9 ; result = (f + g) - (h + i)
        MOV   R0, R4 ; put return value in R0
        LDR   R4, [SP] ; restore R4 from stack
        LDR   R8, [SP, #4] ; restore R8 from stack
        LDR   R9, [SP, #8] ; restore R9 from stack
        ADD   SP, SP, #12 ; deallocate stack space
        MOV   PC, LR ; return to caller
```

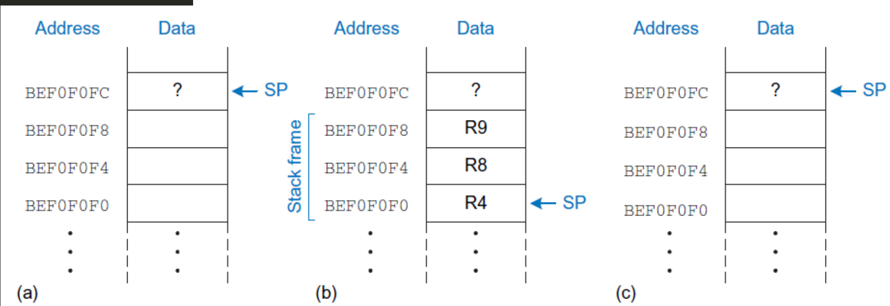


Figure 6.12 The stack: (a) before, (b) during, and (c) after the `diffofsums` function call

Store multiple / Stoor veelvoudige registers

- When working with the stack it is tedious to load and store a single 32-bit word at a time with LDR or STR.
- ARM also provides the LDM and STM instructions to load and store *multiple* registers to multiple consecutive memory locations.

```
LDM r10, {r0, r2, r5-r7} ;load/store all registers in {} brackets
```

- Load 5 consecutive 32-bit (word) values from memory, starting at the address stored in r10. Place the loaded values in r0, r2, r5, r6, and r7

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00080A00	79	00	00	00	6C	51	0B	0E	F4	03	F4	0A	27	00	75	03
00080A10	02	40	24	00	83	01	4D	FE	EC	FF	F9	FF	73	00	F5	FF
00080A20	28	00	37	FE	28	00	00	00	21	EF	00	00	62	00	17	00
00080A30	22	00	00	00	F3	04	00	00	00	00	64	00	00	00	00	00
00080A40	22	01	00	00	DC	A6	83	59	6D	00	CC	D1	03	00	0C	06
00080A50	40	17	18	00	00	60	0C	00	31	00	01	00	02	00	01	00

R10	0x00080A08
R0	0x0AF403F4
R2	0x03750027
R5	0x00245002
R6	0xFE4D0183
R7	0xFFF9FFEC

Store multiple / Stoor veelvoudige registers

- There is ambiguity regarding which direction in memory (ascending or descending) the registers will be loaded and stored.
- There are therefore two main variations of `LDM` and `STM` that is indicated with suffixes:

  -`IA` : increment after

  -`DB` : decrement before

- The ARMv7-M therefore allows for pushing and popping data to and from both Full Descending (-FD) and Empty Ascending (-EA) stacks, each with their own suffix as well.

Stack Type	Store	Load
Full descending	**`STMFD`** (or **`STMDB`**, Decrement Before)	**`LDMFD`** (or **`LDM`** or **`LDMIA`**, Increment after)
Empty ascending	**`STMEA`** (or **`STM`** or **`STMIA`**, Increment after)	**`LDMEA`** (or **`LDMDB`**, Decrement Before)

Store multiple / Stoor veelvoudige registers

```
LDMIA r10, {r0, r2, r5-r7}
```

Is equivalent to:

```
    LDR r0, [r10]
    LDR r2, [r10, #4]
    LDR r5, [r10, #8]
    LDR r6, [r10, #12]
    LDR r7, [r10, #16]
```

$\Rightarrow$ But uses fewer instructions and less execution time

```
STMIA r5, {r0-r2}
```

Is equivalent to:

```
    STR r0, [r5]
    STR r1, [r5, #4]
    STR r2, [r5, #8]
```

Store multiple / Stoor veelvoudige registers

- You can also optionally enable base register writeback by adding a !

```
LDMIA r10!, {r0, r2, r5-r7}
```

Is equivalent to:

```
LDR r0, [r10], #4
LDR r2, [r10], #4
LDR r5, [r10], #4
LDR r6, [r10], #4
LDR r7, [r10], #4
```

or

```
LDR r0, [r10]
LDR r2, [r10, #4]!
LDR r5, [r10, #4]!
LDR r6, [r10, #4]!
LDR r7, [r10, #4]!
```

```
STMIA r5!, {r0-r2}
```

Is equivalent to:

```
STR r0, [r5], #4
STR r1, [r5], #4
STR r2, [r5], #4
```

or

```
STR r0, [r5]
STR r1, [r5, #4]!
STR r2, [r5, #4]!
```

Store multiple / Stoor veelvoudige registers

- Access modes variations
  - -IA : increment after
  - -DB : decrement before

	LDMIA r10, {r0, r2, r5-r7} / STMIA r10, {r0, r2, r5-r7}	LDMDB r10, {r0, r2, r5-r7} / STMDB r10, {r0, r2, r5-r7}
R10	0x00080A38	0x0080A38
R0	0x00640000	0x00000028
R2	0x00000000	0x0000EF21
R5	0x00000122	0x00170062
R6	0x5983A6DC	0x00000022
R7	0xD1CC006D	0x000004F3

Offset(h)	00	01	02	03
00080A20	28	00	37	FE
00080A24	28	00	00	00
00080A28	21	EF	00	00
00080A2C	62	00	17	00
00080A30	22	00	00	00
00080A34	F3	04	00	00
00080A38	00	00	64	00
00080A3C	00	00	00	00
00080A40	22	01	00	00
00080A44	DC	A6	83	59
00080A48	6D	00	CC	D1
00080A4C	03	00	0C	06
00080A50	40	17	18	00

Store multiple / Stoor veelvoudige registers

- Access modes variations
  - -IA : increment after
  - -DB : decrement before

	LDMIA r10, {r0, r2, r5-r7} / STMIA r10, {r0, r2, r5-r7}	LDMDB r10, {r0, r2, r5-r7} / STMDB r10, {r0, r2, r5-r7}
R10	0x00080A38	0x0080A38
R0	0x00640000	0x00000028
R2	0x00000000	0x0000EF21
R5	0x00000122	0x00170062
R6	0x5983A6DC	0x00000022
R7	0xD1CC006D	0x000004F3

Offset(h)	00	01	02	03
00080A20	28	00	37	FE
00080A24	28	00	00	00
00080A28	21	EF	00	00
00080A2C	62	00	17	00
00080A30	22	00	00	00
00080A34	F3	04	00	00
00080A38	00	00	64	00
00080A3C	00	00	00	00
00080A40	22	01	00	00
00080A44	DC	A6	83	59
00080A48	6D	00	CC	D1
00080A4C	03	00	0C	06
00080A50	40	17	18	00

Store multiple / Stoor veelvoudige registers

- Access modes variations
  - -IA : increment after
  - -DB : decrement before

	LDMIA r10, {r0, r2, r5-r7} / STMIA r10, {r0, r2, r5-r7}	LDMDB r10, {r0, r2, r5-r7} / STMDB r10, {r0, r2, r5-r7}
R10	0x00080A38	0x0080A38
R0	0x00640000	0x00000028
R2	0x00000000	0x0000EF21
R5	0x00000122	0x00170062
R6	0x5983A6DC	0x00000022
R7	0xD1CC006D	0x000004F3

Offset(h)	00	01	02	03
00080A20	28	00	37	FE
00080A24	28	00	00	00
00080A28	21	EF	00	00
00080A2C	62	00	17	00
00080A30	22	00	00	00
00080A34	F3	04	00	00
00080A38	00	00	64	00
00080A3C	00	00	00	00
00080A40	22	01	00	00
00080A44	DC	A6	83	59
00080A48	6D	00	CC	D1
00080A4C	03	00	0C	06
00080A50	40	17	18	00

Store multiple / Stoor veelvoudige registers

- ARM further provides the convenient PUSH and POP instructions.
  - They are not real instructions but instead simply aliases for `STMDB` and `LDMIA`.

```
PUSH {r0, r2} ↔ STMDB sp!, {r0, r2}
POP {r0, r2}  ↔ LDMIA sp!, {r0, r2}
```

- PUSH and POP uses the stack pointer by default updates it after the instruction.

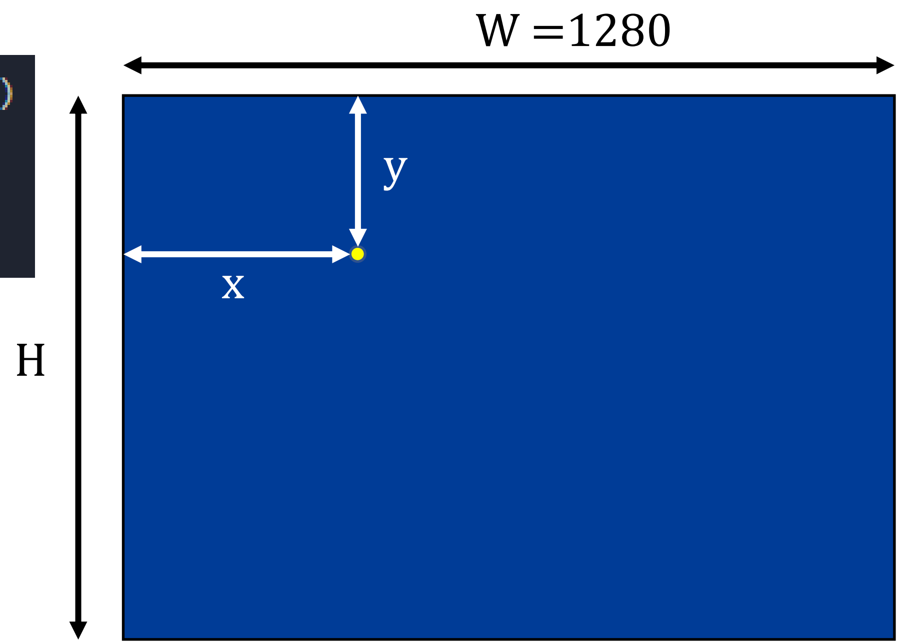
Offset(h)	00	01	02	03
0007FFC8	1A	1A	78	04
0007FFCC	F4	0A	27	00
0007FFD0	75	03	02	40
0007FFD4	24	00	9E	FC
0007FFD8	DF	02	FE	00
0007FFDC	11	00	49	FF
0007FFE0	F9	FF	38	FF
0007FFE4	C2	FC	38	FF
0007FFE8	00	00	CF	F0
0007FFEC	00	00	5F	00
0007FFF0	14	00	21	00
0007FFF4	00	00	5A	04
0007FFF8	00	00	00	00
0007FFFC	48	03	00	00
00080000	00	00	5E	01
00080004	00	00	14	9C

C functions – parameters and return values / C funksies – parameters en terugvoer waardes

- Example – function to calculate the address of a pixel at coordinates (x,y) on an image.

```c
uint32_t get_screen_pos( uint32_t x, uint32_t y)
{
    return y * 1280 + x;
}
```

```
MOV   r0, #100      ;x
MOV   r1, #50       ;y
LSL   r6, r1, #10  ;r6 = y * 2^10
LSL   r7, r1, #8   ;r7 = y * 2^8
ADD   r2, r6, r7   ;r2 = y*1280
ADD   r0, r2, r0   ;r0 = y*1280+x
```



C functions – parameters and return values / C funksies – parameters en terugvoer waardes

```
      MOV    r0, #5   ;x = 5
      MOV    r1, #10 ;y = 10
      BL     get_screen_pos
      B      the_end ;r0 has result

      ;function get_screen_pos
      ;calculates    y*1280+x
      ;r0          = x coordinate
      ;r1          = y coordinate
      ;r0          = return value (y*1280+x)
      ;r6, r7          = temporary result (scratchpad)
get_screen_pos
      STMDB r13!, {r6, r7, lr} ; preserve r6,r7 and lr
      MOV    r6, r1
      LSL    r6, r1, #10
      LSL    r7, r1, #8
      ADD    r2, r6, r7
      ADD    r0, r2, r0
      LDMIA r13!, {r6, r7, pc} ;restore r6, r7 and return

the_end
```

Passing Parameters by Reference/ Parameters aangestuur deur Verwysing

- Pass by value

```
void do_something(int param1)
{
    ...
}
```

- Pass by reference

```
void do_something(int* param1)
{
    ...
}
```

- Passing by reference, passes the **address** of the parameter to the function (not a copy of the value)

Passing Parameters by Reference/ Parameters aangestuur deur Verwysing

```
arguments   DCD 5, 10
      LDR   r0, =arguments ;r0 has address of 1st argument
      ADD   r1, r0, #4 ;r1 has address of 2nd argument
      BL    get_screen_pos
      B     the_end ;r0 has result

      ;function get_screen_pos
      ;calculates    y*1280+x
      ;r0        = address of x coordinate
      ;r1        = address of y coordinate
      ;r0        = return value (y*1280+x)
      ;r6,r7         = temporary result (scratchpad)
get_screen_pos
      STMDB r13!, {r6, r7, lr} ; preserve r6,r7 and lr
      LDR   r1, [r1] ;use address in r1 to load y value
      LSL   r6, r1, #10
      LSL   r7, r1, #8
      ADD   r2, r6, r7
      LDR   r0, [r0] ;use address in r0 to load x value
      ADD   r0, r2, r0
      LDMIA r13!, {r6, r7, pc} ;restore r6, r7 and return

the_end
```

Passing Parameters on the Stack/ Aanstuur van parameters deur die stapel

- Same as passing parameters by reference but use a dedicated register for the memory address – R13 – the Stack Pointer.

```
      MOV   SP, #0x1000
      MOV   r0, #5   ;x = 5
      MOV   r1, #10 ;y = 10
      STMDB sp!, {r0, r1} ;push params onto stack
      BL    get_address
      LDMIA sp!, {r0, r1} ;pop results off stack.
      B     the_end       ;now r0 contains result

      ;function get_address: calculates y*1280+x
get_address
      STMDB sp!, {r6, r7, lr} ; preserve r6, r7 and lr
      LDR   r0, [sp, #12] ;load x param from stack
      LDR   r1, [sp, #16] ;load y param from stack
      MOV   r6, r1
      LSL   r6, r1, #10
      LSL   r7, r1, #8
      ADD   r2, r6, r7
      ADD   r2, r2, r0
      STR   r2, [sp, #12] ;store result to bottom of stack
      LDMIA sp!, {r6, r7, pc} ;restore r6, r7 and return

the_end
```

Passing Parameters on the Stack/ Aanstuur van parameters deur die stapel

	0x0FE8	…
	0x0FEC	…
	0x0FF0	…
	0x0FF4	…
SP->	0x0FF8	Parameter *x*
	0x0FFC	Parameter *y*
	0x1000	…

r0

r1

Line 4

```
STMDB sp!, {r0, r1} ;push params onto stack
```

Passing Parameters on the Stack/ Aanstuur van parameters deur die stapel

	0x0FE8	…	…
SP->	0x0FEC	…	R6
	0x0FF0	…	R7
	0x0FF4	…	R14 (LR)
SP+12	0x0FF8	Parameter *x*	Parameter *x*
SP+16	0x0FFC	Parameter *y*	Parameter *y*
	0x1000	…	…

Line 11

```
STMDB sp!, {r6, r7, lr} ; preserve r6, r7 and lr
```

Passing Parameters on the Stack/ Aanstuur van parameters deur die stapel

	0x0FE8	…	…	…
SP->	0x0FEC	…	R6	R6
	0x0FF0	…	R7	R7
	0x0FF4	…	R14 (LR)	R14 (LR)
SP+12	0x0FF8	Parameter *x*	Parameter *x*	Result
	0x0FFC	Parameter *y*	Parameter *y*	Parameter *y*
	0x1000	…	…	…

Line 19

```
STR   r2, [sp, #12] ;store result to bottom of stack
```

Passing Parameters on the Stack/ Aanstuur van parameters deur die stapel

	0x0FE8	…	…	…	…
	0x0FEC	…	R6	R6	R6
	0x0FF0	…	R7	R7	R7
	0x0FF4	…	R14 (LR)	R14 (LR)	R14 (LR)
SP->	0x0FF8	Parameter *x*	Parameter *x*	Result	Result
	0x0FFC	Parameter *y*	Parameter *y*	Parameter *y*	Parameter *y*
	0x1000	…	…	…	…

Line 20

```
LDMIA sp!, {r6, r7, pc} ;restore r6, r7 and return
```

Passing Parameters on the Stack/
Aanstuur van parameters deur die stapel

	0x0FE8	…	…	…	…
	0x0FEC	…	R6	R6	R6
	0x0FF0	…	R7	R7	R7
	0x0FF4	…	R14 (LR)	R14 (LR)	R14 (LR)
	0x0FF8	Parameter *x*	Parameter *x*	Result	Result
	0x0FFC	Parameter *y*	Parameter *y*	Parameter *y*	Parameter *y*
SP->	0x1000	…	…	…	…

Line 6

```
LDMIA sp!, {r0, r1} ;pop results off stack.
```