Lab 3: Programming with subroutines

3.1 Objectives

The main goal of this lab session is introducing the concept of subroutine and getting familiar with its use as a powerful programming technique. To do that, we have to study the following topics:

- The support for subroutine management that is provided by the ARM assembler.
- The concepts of *call stack* and *stack frame*. We must also understand how the stack frame is created (prolog) and deleted (epilog) during the execution of a subroutine.
- The protocol, or agreement, that is used for passing parameters from the calling program to the subroutine, and passing results back to the program.

3.2 Subroutines and Call Stack

A subroutine is a fragment of code that can be called from any point in the program in such a way that when the subroutine is finished the program execution is resumed at the instruction following the call. This definition includes the possibility that a subroutine is called from another subroutine (in which case we have nested subroutines), or even the same one (which is known as a recursive call).

Figure 3.1 shows an example with a main program that calls the subroutine `SubRut1`, where the dashed arrows indicate the change in the execution flow at invocation, as well as at return. Subroutines are used to simplify and reuse the code, and they are the mechanism used in high-level languages to implement procedures, functions and methods.

The code of a subroutine can be parameterized, that is, the code can be written using some parameters, which can be initialized at the invocation. For instance, the subroutine may have a loop from 1 to N, where N is a parameter. When the subroutine is called this internal symbol, N, is initialized with a value. It is said that the parameters are passed in the call (or invocation). Subroutines can also return a value to the calling program.

As Figure 3.1 shows, there is only one copy of the code of a subroutine, which must be placed outside the region of the main program. It is also recommended to identify the first instruction of the subroutine with a *label*, because in this case we can invoke it just performing these two tasks:

- Preserving somewhere the address of the instruction in which the execution of the program must continue upon return from the subroutine. This is referred as the *return address*.
- Executing an unconditional branch to the label that identifies the beginning of the subroutine.

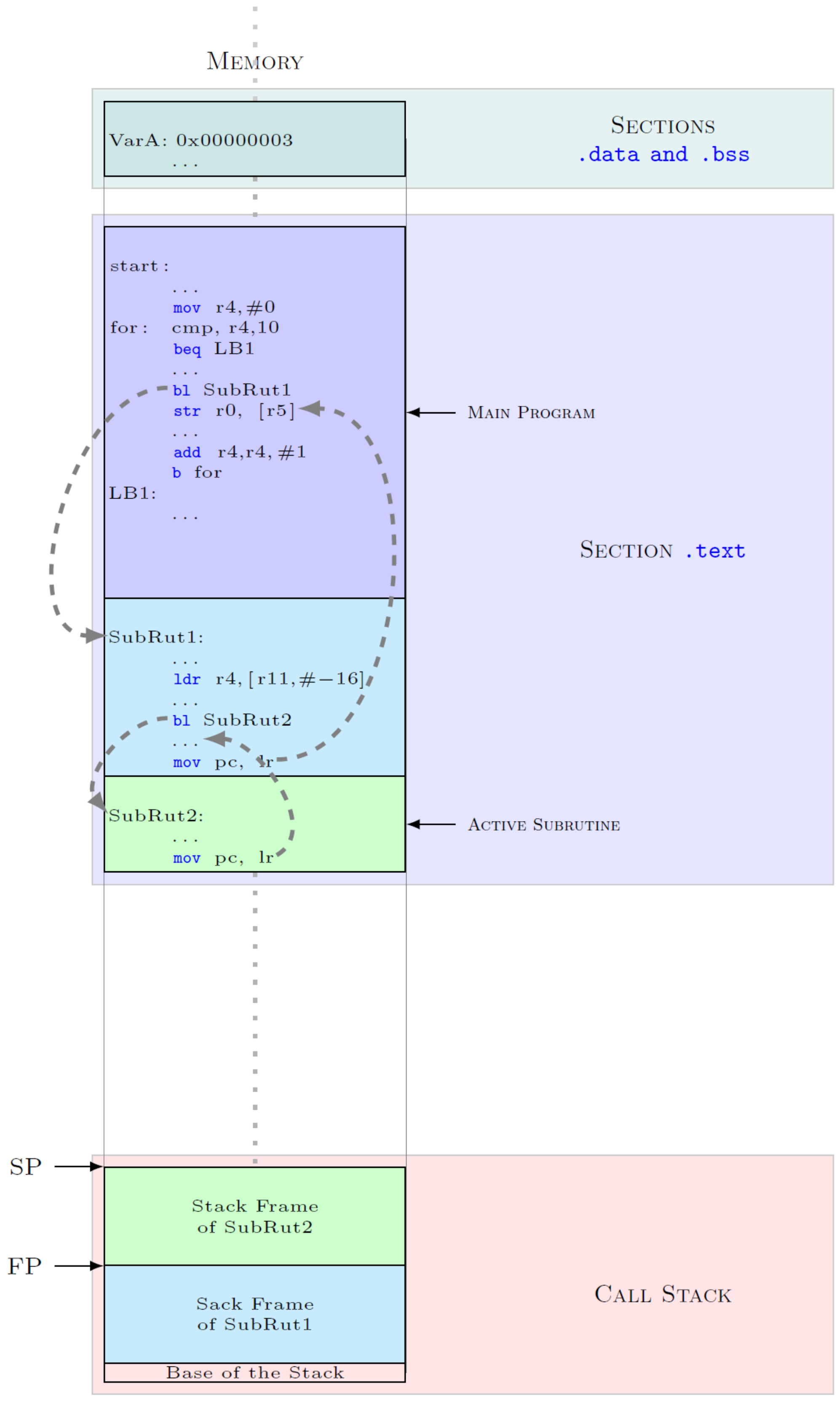


Figure 3.1 Subroutines and Call Stack. The main program invokes subroutine SubRut1, which in turn invokes subroutine SubRut2, which becomes the active subroutine.

This can be done in several ways, but generally all instruction sets provide a special instruction for subroutine invocation. In ARM this instruction is *Branch and Link*, whose assembly syntax is:

```
BL Label
```

This instruction saves to R14 the address of the next sequential instruction (i.e. the return address) and jumps to the instruction identified by Label. This is the reason why register R14 is also known as the *link register*, and therefore in assembly it can be also referenced as LR.

Return from subroutine can be implemented just by copying the value of R14 to the PC. This is usually done by means of a MOV instruction, as shown in Figure 3.1:

```
mov pc, lr
```

Alternatively, we can use the instruction BX (*branch and exchange*) for the same purpose:

```
bx lr
```

However, in order to safely invoke a subroutine from any location of the program, it is essential that the subroutine preserves the architectural state of the machine (i.e. the contents of those registers than can be seen by the programmer). Figure 3.1 illustrates this problem. The main program executes a for loop which uses r4 to control the number of iterations (i.e. r4 plays the role of the loop index). In the loop body, there is a call to subroutine SubRut1, which contains the instruction:

```
ldr r4, [r11, #-16]
```

As a consequence, register r4 is overwritten, with the undesirable side effect of modifying the loop index. Obviously, this wouldn't have happened if the programmer of the main program had selected another register to implement the loop index. However, it is ***not reasonable to force the programmer to know every internal detail about the implementation of a subroutine before calling it***. Consider, for instance, a subroutine to find the dot product of two vectors. ***The subroutine might have been written by somebody else years earlier***, and its internal implementation should not be relevant for the programmer, who is only interested in the result of the subroutine.

There is a simple solution to the above problem: every subroutine must create a temporary copy in memory of all those registers that will be modified, and restore their original values just before returning to the main program. Of course, some amount of extra memory space will be required to implement this idea. It should be observed that this space cannot be static, as we don't know "a priori" how many invocations of a given subroutine might be running. If we consider, for instance, the implementation of a subroutine for the recursive implementation of the function fact(n) to compute n!, we will have at some given moment n invocations of the subroutine on the fly, because the subroutine will call itself to implement the following computation: fact(n)= n*fact(n-1). So we have to reserve n memory slices to save the architectural state (also called the context) n times. The problem is that n is an input parameter of the subroutine, so we don't know exactly how much memory space has to be reserved. Consequently, the best solution is that the subroutine itself is in charge of reserving the memory space required to preserve the context at the beginning of its execution, and releasing such space at the end.

This solution is implemented by means of the *call stack*, which is a compact memory region that is accessed using a LIFO (Last-In-First-Out) policy. This region is used to store all the information related to the *live* subroutines of the program (i.e. those subroutines that have started, but not finished their execution). The stack normally begins at the end of the available memory, i.e. in the higher addresses, and grows towards lower ones.

The stack region that is used by a subroutine during its execution is called *Stack Frame* or *Activation Record*. The example in Figure 3.1 shows the state of the stack when the subroutine SubRut1 has invoked SubRut2, and the latter is being executed. As a consequence of the nested call, we have two live subroutines, and the stack stores the stack frames of both.

A subroutine is normally structured into three parts:

Entry code (prolog)
Subroutine body
Exit code (epilog)

So, it begins with a piece of code (the prolog) that is in charge of creating the stack frame of the subroutine. At the end, there is some code (the epilog) in charge of restoring the architectural state of the calling program, and release the memory space occupied by the stack frame, thus leaving the stack exactly as it was before executing the prolog.

Accesses to the stack are managed by means of a pointer, the so called *stack pointer* (SP), that stores the address of the last occupied location of the stack. The role of the SP can be played by an architectural register, which should be initialized with the address of the base of the stack (i.e. the highest address of the stack region). When this address is stored in the SP, it means that the stack is empty. In order to facilitate the access to the information stored in the stack frame, a second pointer, the so called *frame pointer* (FP), is habitually used. The frame pointer is normally pointing to the base address (i.e. the highest address) of the stack frame.

The subroutine will use the stack frame as a private memory region, which can only be accessed from the subroutine itself. A part of the stack frame is devoted to save a copy of the architectural registers that are to be modified by the subroutine.

The stack frame can also be used as a way of passing parameters to the subroutine through memory locations. To implement this, the calling program must copy the parameters on top of the stack. Then the subroutine inserts its stack frame just on top of the parameters, so that the subroutine can access to them with a positive offset from the FP. Alternatively, parameter passing could be done by means of registers.

Although the programmer who writes the calling program should not be concerned about the internal details of the subroutine, it is obvious that there must be a clearly established agreement with subroutine programmers, whoever they might be, about the mechanism for parameter passing. It is important to observe that this will allow us even to call assembly subroutines that might have been generated, for example, by a compiler of some unknown high-level language. The terms of this agreement are defined by the processor maker, ARM in our case, in a document called subroutine calling standard, which is a part of the so called *Application Binary Interface* (ABI). Next section is devoted to the study of the ARM standard.

3.3 Subroutine Call Standard

The *ARM Architecture Procedure Call Standard* (AAPCS) is the current standard for subroutine calling in ARM architecture [aap]. It specifies a set of rules to allow the interaction among subroutines, even when they have been separately compiled or assembled. It means, indeed, a kind of contract between the calling program and the called subroutine. In the next paragraphs, the most relevant aspects of the standard are summarized. It should be observed, however, that this standard is under permanent evolution and revision, which means that it is important to check the ABI version that is going to be used and study the corresponding standard.

3.3.1 Architectural registers

The standard determines which registers must be preserved by the subroutine. The others can be overwritten. As mentioned above, if the subroutine needs to overwrite any of the registers that must be preserved, it must previously make a copy of such register in its stack frame and restore the value before returning. On the other hand, the calling program must be aware that there are some registers that the subroutine is not obliged to preserve, and thus the original value of these registers will most likely be lost after the call.

Table 3.1 describes the intended role that the AAPCS assigns to each register. As we can see, registers have alternative names (alias), which somehow remind their roles. It is relevant to observe that, although saving the value of LR is not mandatory, this is the register that contains the return address. Therefore, if the subroutine calls other subroutines (or recursively calls itself), it will have to store the new return address to R14, which of course will destroy its initial value. This is the case illustrated in Figure 3.1, where SubRut1 calls SubRut2. To ensure that SubRut1 is able to find its way back to the main program, it must save to its stack frame the value of LR before calling SubRut2. This will not be required if the subroutine doesn't call other subroutines, as for example it is the case in SubRut2, which is said to be a *leaf subroutine*.

3.3.2 Stack Model

Figure 3.2 illustrates the so called full-descending stack model, which is imposed by the AAPCS. This model has the following characteristics:

- The SP is initialized with a high memory address, and grows towards decreasing addresses.
- The SP will always be pointing to the last **occupied** position of the stack.

Additionally, it is imposed that the contents of the SP must always be a multiple of 4.

Table 3.1 Role of the architectural registers according to AAPCS

REGISTERS	ALIAS	MUST BE PRESERVED?	ROLE ACCORDING TO AAPCS
r0-r3	a1-a4	No	Passing parameters to the subroutine and getting the return value
r4-r10	v1-v7	Yes	Normally used to store variables because they are preserved through subroutine calls
r11	fp, v8	Yes	Frame Pointer
r12	ip	No	It can be used as an auxiliary registers in prologs
r13	sp	Yes	Stack Pointer
r14	lr	No	Link Register

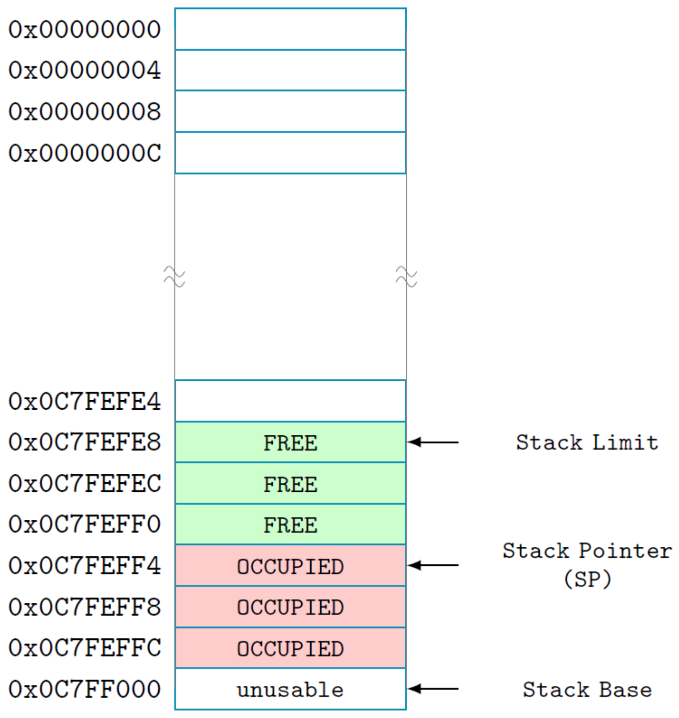


Figure 3.2 Illustration of a *Full Descending* stack

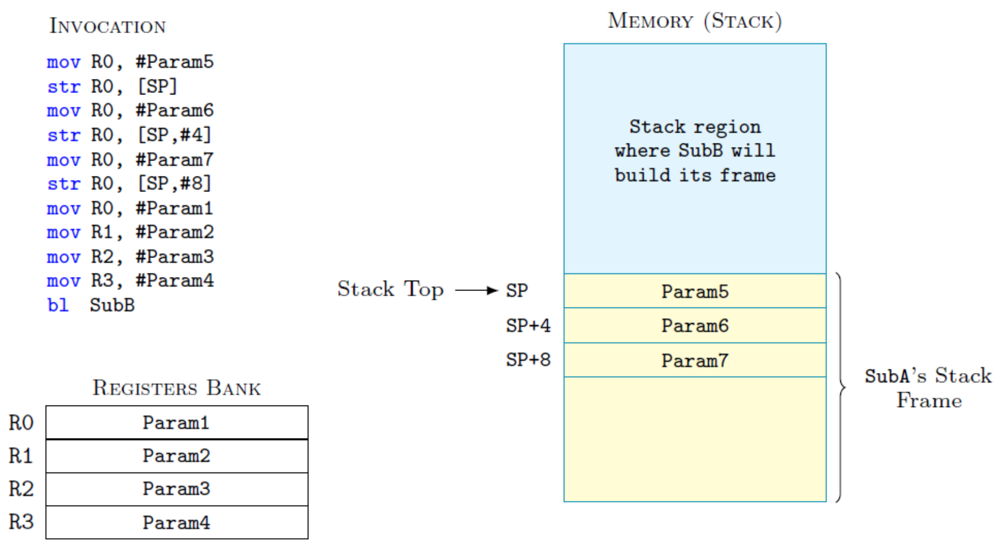
3.3.3 Parameter passing

According to the standard, the first four parameters of a subroutine must be passed through registers, using r0-r3 in this order. Any parameters beyond the fourth position must be passed through memory, by writing them in the top of the stack as follows:

- The fifth parameter in the address [SP]
- The sixth in [SP+4]
- The seventh in [SP+8], etc.

This way the subroutine can always access these parameters using the frame pointer with positive offsets.

We should observe that this procedure means writing these parameters on the top of the stack frame of the calling subroutine. This **extra space** for parameters should be taken into account in the prolog of any subroutine that executes a call with more than four parameters, in order to compute the total amount of memory space that is required to build the stack frame. Figure 3.3 shows an example in which a subroutine SubA has to call the subroutine SubB, passing seven parameters that are represented as Param1-7. As we can see, the first four parameters are passed through registers (r0-r3), the last three are passed through the top positions of the stack frame of SubA.



INVOCATION

```
mov R0, #Param5
str R0, [SP]
mov R0, #Param6
str R0, [SP,#4]
mov R0, #Param7
str R0, [SP,#8]
mov R0, #Param1
mov R1, #Param2
mov R2, #Param3
mov R3, #Param4
bl  SubB
```

REGISTERS BANK

R0	Param1
R1	Param2
R2	Param3
R3	Param4

Figure 3.3 Passing seven parameters from routine SubA to SubB. It is assumed that these parameters are immediate operands. If they were variables, ldr instructions should have been used instead of mov.

3.3.4 Return Value

For those subroutines that return a value (functions in high-level language), the AAPCS specifies that such value must be retuned through r0[1].

3.4 The Stack Frame

Any subroutine which is in accordance with the previous rules is said to be standard compliant. Nevertheless, in this lab we will adopt the following rules, which are a bit more restrictive:

[1] This is a simplification that is valid for results having the size of one word or less. The rest of the cases lie beyond the scope of this document.

- If a subroutine A invokes another subroutine B, then we will have to store the value of LR in the stack. The reason is that we need the value of LR to return to the program that has called A, and at the same time we also need LR to return from B to A.
- A frame pointer will be always used to facilitate debugging, as well access to the stack frame. So, storing the previous value of FP in the stack will be required.
- Although SP must be saved, it is not necessary to include it in the stack frame, since we can derive the previous value of SP from the value of FP.
- As for registers r4-r10, we will save in the stack only those that are going to be modified in the subroutine body.

Figure 3.4 illustrates the stack frame resulting from applying the above rules to a non-leaf subroutine, which is the stack frame generated by the gcc-4.7 compiler, with debugging activated and without optimizations[2].

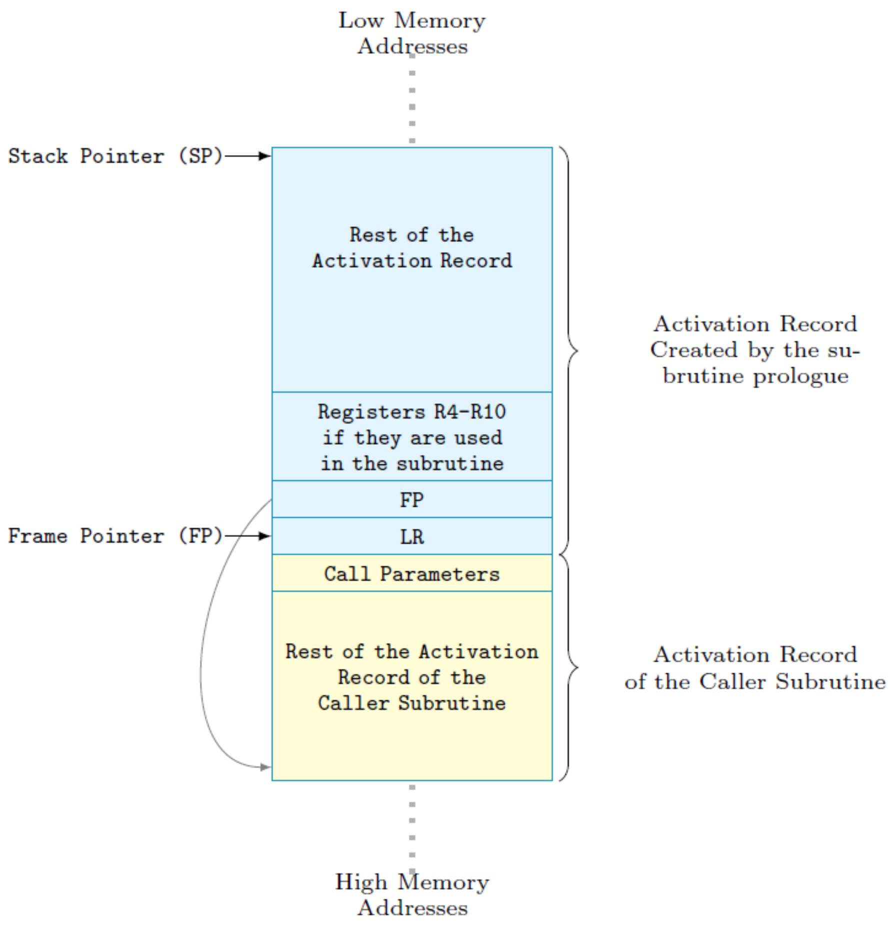

Low Memory Addresses
Stack Pointer (SP)
Rest of the Activation Record
Registers R4-R10 if they are used in the subrutine
FP
Frame Pointer (FP)
LR
Call Parameters
Rest of the Activation Record of the Caller Subrutine
Activation Record Created by the subrutine prologue
Activation Record of the Caller Subrutine
High Memory Addresses

Figure 3.4. Recommended structure of the stack frame or activation record. For leaf subroutines we can omit the copy of LR.

[2] With flags –g –O0

3.4.1 Load and Store Multiple

In addition to the conventional load and store instructions, the ARM instruction set offers instructions to load or store multiple words. These instructions are very useful to manage the stack frame, as we have normally to save and restore multiple words to/from the stack.

The Load Multiple (LDM) instruction loads a group of registers with data from several consecutive memory locations, while the Store Multiple (STM) moves information in the opposite direction. The syntax of these instructions is:

```
LDM<mode> Rb[!], {list of registers}
STM<mode> Rb[!], {list of registers}
```

where Rb is the base register from which the starting address of the operation is derived. The “!” mark is optional; if included, then the base register will be properly updated in such a way that several accesses of the same kind could be chained.

There are four addressing modes for these instructions, which gives four versions for each one. In this lab we will only study the two modes that are useful to program the prolog and epilog of subroutines.

- *Increment After* (IA): The starting address is directly given by the base register Rb:

  starting address = Rb

- *Decrement Before* (DB): The starting address is obtained by subtracting from Rb the number of registers in the list multiplied by 4:

  starting address = Rb – 4*Number_of_registers

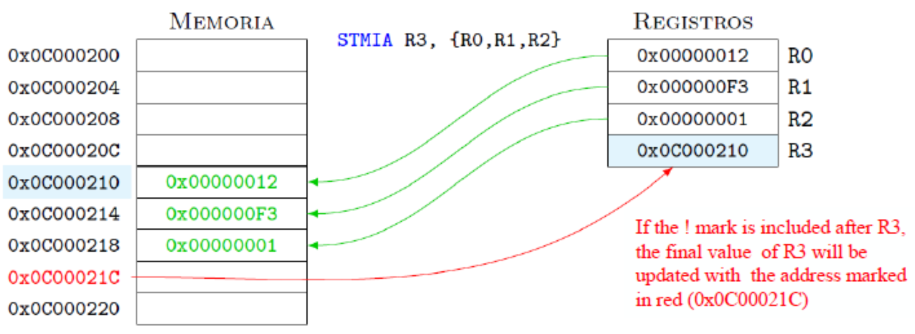
Figure 3.5 explains the behavior of the four corresponding instructions: STMIA, LDMIA, STMDB, and LDMDB. Beginning at the starting address, all the architectural registers are traversed in order (first R0, then R1…). If the register is in the list of registers, then the load or store operation for that register is executed, and the address is incremented by four bytes. Otherwise, the register is skipped.

The two instructions that allow us to implement the *push* and *pop* operations in a *full descending* stack are:

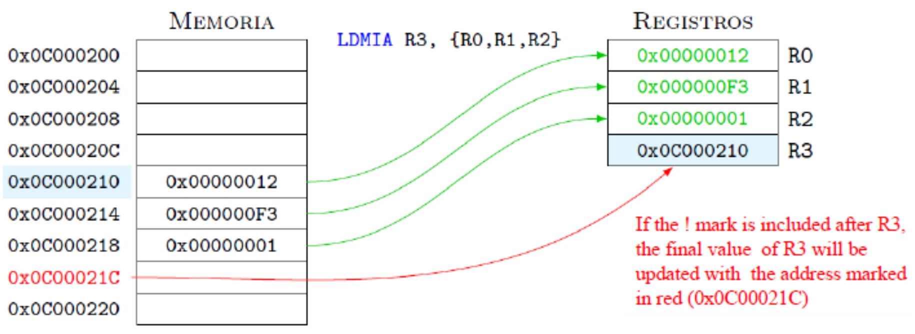
Push: STMDB Rb!, {list of registers}. If we use the SP as the base register, this instruction is totally equivalent to pushing the list of registers to the stack. In fact, we can refer to this instruction with the two following alias:

- STMFD Rb!, {list of registers}, where FD derives from full descending.

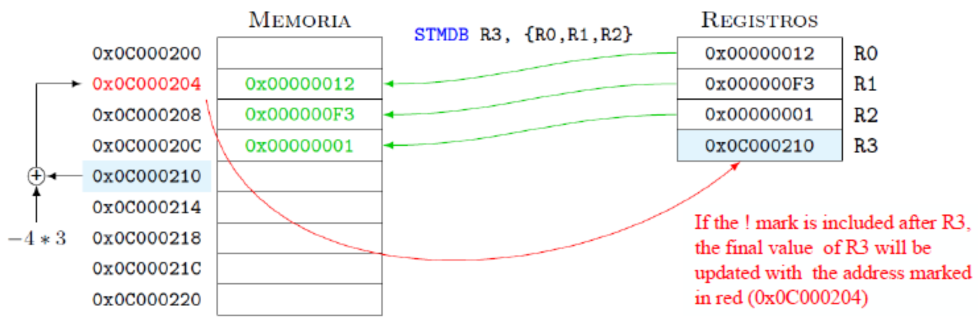
- PUSH {list of registers}. In this case the base register is implicitly SP. This is the format that we will use to program the prolog of subroutines.



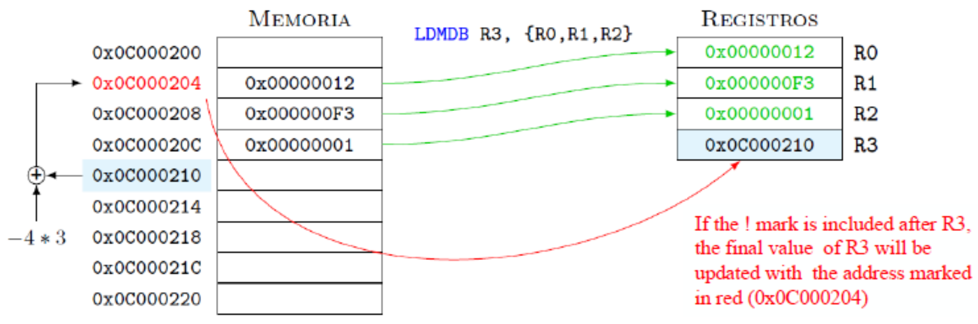
(a) STMIA



(b) LDMIA



(c) STMDB



(d) LDMDB

Figure 3.5. Load and store multiple instructions

Pop: LDMIA Rb!, {list of registers}. If we use the SP as the base register, this instruction is equivalent to popping the list of registers from the stack. We can also refer to this instruction with the two following alias:

- LDMFD Rb!, {list of registers}. As in the store instruction, FD derives from full descending.

- POP {list of registers}. This is the format that will be used to program the epilog of subroutines.

Examples:

```
STMDB SP!, {R4-R10,FP,LR}    @ Starting at address SP-4*9 (and moving towards increasing
                             @ addresses), the contents of registers R4-R10, FP and LR, are
                             @ copied to memory.
                             @ SP is decreased by 4*9
                             @ In other words…registers are inserted in the stack.
STMFD SP!, {R4-R10,FP,LR}    @ The same as above
PUSH {R4-R10,FP,LR}          @ The same as above

LDMIA SP!, {R4-R10,FP,LR}    @ Starting at address SP (and moving towards increasing
                             @ addresses), the memory contents is copied to registers R4-
                             @ R10, FP and LR.
                             @ SP is increased by 4*9.
                             @ In other words…registers are extracted from the stack.
LDMFD SP!, {R4-R10,FP,LR}    @ The same as above
POP {R4-R10,FP,LR}           @ The same as above
```

3.4.2 Prolog and Epilog

In Text Box 1 we can see a valid prolog to build the stack frame represented in Figure 3.4. The first instruction saves to the stack all those registers from the calling program that must be preserved, thus leaving SP pointing to the new top of the stack. It should be mentioned again that saving all the registers R4-R10 to the stack is, in general, not necessary. We have to save only those registers that are to be modified by the subroutine. The next instruction is used to store in FP the base address of the stack frame of the called subroutine. Finally, the last instruction reserves the extra space to store in the stack all the local variables, as well as the parameters for calling other subroutines. In some cases, this prolog can be simplified:

- If the code of the subroutine does not include any calls with more than four parameters, and there are not any local variables, then we don't need any extra space, and thus the third instruction can be deleted.
- If we insert only one register in the stack (which would obviously be FP), then the second instruction would become simply ADD FP, SP, #0, which could be also written as MOV FP, SP. This situation will happen in a leaf subroutine that does not modify r4-r10.

Text Box 1. Prolog to build the Stack Frame of Figure 3.4

```
PUSH {R4-R10,FP,LR}                      @ Save registers to the stack
ADD FP, SP, #(4*NumInsertedRegisters-4)  @ FP ← address of the base of the Stack Frame
SUB SP, SP, #4*NumExtraWords             @ Decrease SP to allow extra space for local
                                         @ variables and parameter passing
```

The Text Box 2 shows the corresponding epilog, which is in charge of restoring the architectural state of the calling program, thus leaving the stack exactly as before the call. The first instruction of the epilog restores SP to the same value it had after the execution of the first instruction of the prolog, which leaves SP pointing to the first stacked register. The second instruction extracts from the stack all the registers that had been inserted in the prolog. Finally, the last instruction implements the return to the calling program. At this moment, the values of R4-R10, FP, and LR will be exactly the same as before the call.

Text Box 2. Epilog to restore the architectural state from the Stack Frame of Figure 3.4

```
SUB SP, FP, #(4*NumInsertedRegisters-4)  @ Release the extra space
                                         @ SP ← address of the 1st inserted register
POP {R4-R10,FP,LR}                       @ Restore registers from the stack
BX LR                                    @ Return from subroutine
```

- If we have inserted only one register in the stack, then the first instruction of the epilog can be reduced to SUB SP, FP, #0, which could be also written as MOV SP, FP.
- If we have not reserved any extra space (NumExtraWords=0), and we have not changed the value of SP in the body of the subroutine, then the first instruction could be even deleted, because SP will still preserve the value that was written to it by the PUSH instruction of the prolog.

3.5 An example

The code in Figure 3.6 illustrates a sample program that uses the subroutine Traverse in order to traverse an array A. At each position, j, it selects the larger of the pair of elements A[j] and A[j +1] by means of the subroutine Larger. The larger element is copied to position j of another array B. We can see the C implementation on the left, and the assembler implementation on the right. Note that at the beginning of the program SP is initialized with the _stack symbol, which is provided by the linker when executing the linker script (ld_script.ld). We also set FP to cero, so that when the first subroutine is called, this value is copied in the base of its stack frame. This serves to tell the debugger which subroutine is the first one in a collection of nested calls.

C code

```c
#define N 4
int A[N]={7,3,25,4};
int B[N];

void Traverse();
int Larger();

void main(){
	Traverse (A, B, N);
}

void Traverse (int A[], int B[], int M){
	for(int j=0; j<M-1; j++){
		B[j] = Larger(A[j],A[j+1]);
	}
}

int Larger (int X, int Y){
	if(X>Y)
		return X;
	else
		return Y;
}
```

Assembly code

```
		.extern _stack
		.global start
		.equ N, 4
.data
A:		.word 7,3,25,4
.bss
B:		.space N*4

.text
start:
		ldr	sp, =_stack
		mov	fp, #0
		ldr	r0, =A
		ldr	r1, =B
		mov	r2, #N
		bl	Traverse
		b .

Traverse:
		push	{r4-r8,fp,lr}
		add	fp, sp, #24	@ 24=(4*7)-4
		mov	r4, r0		@ R4, A
		mov	r5, r1		@ R5, B
		sub	r6, r2, #1	@ R6, M-1
		mov	r7, #0		@ R7, j
for:		cmp	r7, r6
		bge	Ret1
		ldr	r0, [r4, r7, lsl #2]
		add	r8, r7, #1
		ldr	r1, [r4, r8, lsl #2]
		bl	Larger
		str	r0, [r5, r7, lsl #2]
		add	r7, r7, #1
		b	for
Ret1:		pop	{r4-r8,fp,lr}
		mov	pc, lr

Larger:
		push	{fp}
		mov	fp, sp		@ SP - 0
		cmp	r0, r1
		bgt	Ret2
		mov	r0, r1
Ret2:		pop	{fp}
		mov	pc, lr
		.end
```

Figure 3.6 A sample program with subroutines

3.6 Development of the lab session

Before the lab session, the student must have developed and tested the codes corresponding to the programs a) and b) mentioned below. During the lab session some modification of one of these programs will be proposed to the student, who must code, test and debug the corresponding program.

In **EACH** program the student must define three sections:

- .data for variables with an initial value (declared with .word)
- .bss for variables without an initial value (declared with .space)
- .text for the code

a) Write an ARM assembly program that implements the following C function, which finds the maximum value of an array A of positive integers, of length lenA, and returns the position (i.e. the index) of such maximum:

```
int max(int A[], int lenA){
        int i, max_candidate, ind;
        max_candidate=0;
        ind=0;
        for(i=0; i<lenA; i++){
                if(A[i]>max_candidate){
                        max_candidate=A[i];
                        ind=i;
                }
        }
        return(ind);
}
```

In order to test the correctness of this function, the student must write a simple program that assigns some values to A, calls the function, and stores the returned value in memory. If you are using the predefined workspace, you can find such program already written in the file “pract2a.asm”.

Remark: The code of this function is the same (except for the subroutine management instructions) as the inner loop of the program in part b of the previous lab session. Note that **max** is a leaf subroutine, so you only need to save and restore FP and those registers r4-r10 that the subroutine modifies.

b) In this part the student must write a sorting algorithm in assembly code. The input will be an array A of N elements, each being an integer greater than 0. The output will be a similar array B, where the elements of A are stored in decreasing order. The assembly program will be based in the following high-level code, which calls the subroutine max:

```
#define N 8
int A[N]={7,3,25,4,75,2,1,1};
int B[N];
int j, ind;
void main(){
        for(j=0; j<N; j++){
                ind=max(A,N)
                B[j]=A[ind];
                A[ind]=0;
        }
}
```

Remark: The code of this program corresponds (except for the subroutine management instructions) to the outer loop of the program in part b of the previous lab session.

References

[aap] The arm architecture procedure call standard. It can be accessed at: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0042d/index.html