



# CPE 221: Computer Organization

06 ARM Memory, Indexing, and Function Calls  
[rahul.bhadani@uah.edu](mailto:rahul.bhadani@uah.edu)

*Rahul Bhadani*

# Announcement

Exam 1: February 19, 2025, Wednesday: 4:20 PM - 5:40 PM

- 1 Page, One-sided , Handwritten Sheet Allowed
- Calculator Allowed
- Close Notes, Close Books
- May ask to write ARM Programming
- Convert C Code to ARM Code
- Will cover topics: Number Systems, 2s complements Floating Points, Digital Logic, Register-Transfer Language, Transistor/Gate Basics, K-map Minimization, Truth Tables, ARM Instructions, Questions on Disassembly and Memory Map of ARM Programs

# Pseudo Instructions

```
LDR R3, [R0, R2] @ R3 = scores[i]
```

LDR is a pseudo-instruction.

The ARM assembler supports a number of pseudo-instructions that are translated into the appropriate combination of ARM instructions at assembly time.

These are simple assembly language instructions that do not have a direct machine language equivalent.

During assembly, the assembler translates each pseudo-instruction into one or more machine language instructions.

# Memory

scores[200]

200 = length

Address	Data
1400031C	scores[199]
14000318	scores[198]
⋮	⋮
14000004	scores[1]
14000000	scores[0]

Main memory

index

# Scaling Index in ARM

ARM can scale (multiply) the index, add it to the base address, and load from memory in a single instruction.

```
LDR R3, [R0, R1, LSL #2]
```

**R1** is scaled (by using shift operation; shift left by two), then added to the base address (**R0**), and thus the memory address is  $R0 + (R1 \times 4)$ .

# Indexing in ARM: Three Types

ARM also provides **offset**, **pre-indexed**, and **post-indexed** addressing to enable dense and efficient code for array accesses and function calls.

# Indexing in ARM ...

Mode	ARM Assembly	Address	Base Register
Offset	LDR R0, [R1, R2]	$R1 + R2$	Unchanged
Pre-index	LDR R0, [R1, R2]!	$R1 + R2$	$R1 = R1 + R2$
Post-index	LDR R0, [R1], R2	$R1$	$R1 = R1 + R2$

$[R_n, \#offset]$

Offset addressing calculates the address as the base register  $\pm$  the offset; the base register is unchanged.

# Indexing in ARM ...

Mode	ARM Assembly	Address	Base Register
Offset	LDR R0, [R1, R2]	$R1 + R2$	Unchanged
Pre-index	LDR R0, [R1, R2]!	$R1 + R2$	$R1 = R1 + R2$
Post-index	LDR R0, [R1], R2	R1	$R1 = R1 + R2$

$[Rn, \#offset]!$

Pre-indexed addressing calculates the address as the base register  $\pm$  the offset and updates the base register to this new address



# Indexing in ARM ...

Mode	ARM Assembly	Address	Base Register
Offset	LDR R0, [R1, R2]	$R1 + R2$	Unchanged
Pre-index	LDR R0, [R1, R2]!	$R1 + R2$	$R1 = R1 + R2$
Post-index	LDR R0, [R1], R2	R1	$R1 = R1 + R2$

$[R_n], \#offset$

Post-indexed addressing calculates the address as the base register only and then, after accessing memory, the base register is updated to the base register  $\pm$  the offset.

# For loop using post-indexing


## High-level Code

```
int i;  
int scores[200];  
...  
for (i = 0; i < 200; i = i + 1)  
    scores[i] = scores[i] + 10;
```

Post-indexing

no need to use  
ADD for  
incrementing  
index

## ARM Assembly Code

```
@ R0 = array base address  
@ initialization code ...  
  
LDR R3, =baseaddr  
LDR R0, [R3] @ R0 contains base address  
ADD R1, R0, #800 @ R1 = base address + (200*4)  
  
LOOP  
CMP R0, R1 @ reached end of array?  
BGE L3 @ if yes, exit loop  
 LDR R2, [R0] @ R2 = scores[i]  
ADD R2, R2, #10 @ R2 = scores[i] + 10  
STR R2, [R0], #4 @ scores[i] = scores[i] + 10  
                  @ then R0 = R0 + 4  
B LOOP @ repeat loop  
  
L3  
  
.data  
baseaddr: .word 0x14000000
```

# What does the Following do?

```
LDR R2, [R3, R4, LSL #2]
```

# What does the Following do?

```
LDR R2, [R3, R4, LSL #2]
```

**Offset is R4 shifted left by 2 (so  $*4$ ), used to index an array of words.**

# What does the Following do?

STR R5, [R6, #-256]

# What does the Following do?

```
STR R5, [R6, #-256]
```

**Negative offset, storing to R6 - 256.**

# What does the Following do?

```
LDRB R7, [R8, R9]
```

# What does the Following do?

```
LDRB R7, [R8, R9]
```

**Byte load with register offset.**



# What does the Following do?

```
STR R2, [R3, R4, LSL #2]!
```

# What does the Following do?

```
STR R2, [R3, R4, LSL #2]!
```

**Pre-index:  $R3 = R3 + (R4 \ll 2)$ , then store R2**

# What does the Following do?

```
LDRB R5, [R6, #-64]!
```

# What does the Following do?

```
LDRB R5, [R6, #-64]!
```

**Pre-index with negative offset:  $R6 = R6 - 64$ , then load byte.**

# Bytes and Characters

$[-128, 127]$  only need single bytes

American Standard Code for Information Interchange (ASCII): assigns each text character a unique byte value.


ARM provides load byte (**LDRB**), load signed byte (**LDRSB**), and store byte (**STRB**) to access individual bytes in memory.

# Accessing individual bytes

Memory				
Byte Address	3	2	1	0
Data	F7	8C	42	03

**R4**

Registers				
R1	00	00	00	8C
R2	FF	FF	FF	8C
R3	xx	xx	xx	9B



9B	8C	42	03
----	----	----	----

**Leaves the rest of the bytes to be 0.**

LDRB R1, [R4, #2]

**Leaves the rest of the bytes to be 1 (signed).**

LDRSB R2, [R4, #2]

**Stores the the 0x9B into R4 at Byte 3**

STRB R3, [R4, #3]

# Operating on Strings

Array of characters – null terminated (0x00) in high-level languages.

“Hello!” = (0x 48 65 6C 6C 6F 21 00) stored in memory.



Null Character

# Strings in ARM Assembly

## High-level Code

```
// high-level code
// chararray[10] declared and
// initialized earlier
int i;
for (i = 0; i < 10; i = i + 1)
    chararray[i] = chararray[i] - 32;
```

## ARM Assembly Code

```
@ R0 = base address of chararray (initialized
earlier), R1 = i
    MOV R1, #0 @ i = 0
LOOP CMP R1, #10 @ i < 10 ?
    BGE DONE @ if (i >=10), exit loop
    LDRB R2, [R0, R1] @ R2 = mem[R0+R1] = chararray[i]
    SUB R2, R2, #32 @ R2 = chararray[i] - 32
    STRB R2, [R0, R1] @ chararray[i] = R2
    ADD R1, R1, #1 @ i = i + 1
    B LOOP @ repeat loop

DONE
```





# Function Calls

# Functions in ARM

- In ARM, the **caller (function that calls another function)** conventionally places up to four arguments in registers R0–R3 before making the function call, and the **callee (function that is called by another function)** places the return value in register R0 before finishing.

# Functions in ARM

- By following this convention, both functions know where to find the arguments and return value, even if the caller and callee were written by different people.

# Functions in ARM

- The caller stores the return address in the link register LR at the same time it jumps to the callee using the branch and link instruction (**BL**). The callee must not overwrite any architectural state or memory that the caller is depending on.

# Functions in ARM

- The callee must not interfere with the behavior of the caller, i.e. the callee must know where to return to after it completes and it must not trample on any registers or memory needed by the caller.

# Functions in ARM

- The callee must not overwrite any architectural state or memory that caller is depending on.
- The callee must leave the saved registers (R4–R11, and LR) and the stack, a portion of memory used for temporary variables, unmodified.

# Branch-and-Link instruction (**BL**)

**BL** is used for calling a function.

ARM moves the link register to PC to return from a function using **MOV PC, LR**

# A function call

## High-level Code

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

## ARM Assembly Code

```
0x00008000 MAIN ..  
... ..  
0x00008020 BL SIMPLE @ call the simple function  
... ..  
0x00009020 SIMPLE MOV PC, LR @ return
```

label of callee(address)

Branch and Link Instruction:  
stores the return address of  
the next instruction in the  
link register, and branches  
to the target

address to calling line



# Example

## High-level Code

```
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;
}
```

## ARM Assembly Code

```
@ 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
```

- diffofsums overwrote 3 registers:
- diffofsums can use the stack to temporarily store registers

We will revisit this code for improvement←later

# Function with Multiple Parameters

## ARM Assembly Code

```
.global _start
_start:
    MOV R0, #2          @ First parameter
    MOV R1, #3          @ Second parameter
    BL add_numbers      @ Call function
    B _exit

add_numbers:
    ADD R0, R0, R1      @ R0 = R0 + R1
    MOV PC, LR          @ Return

_exit:
    MOV R7, #1
```



# The Stack

# Stack

- The stack is memory that is used to save information within a function.
- Each function may allocate stack space to store local variables but must deallocate it before returning.
- Like stack of dishes, last-in-first-out (LIFO) queue
- Expands: uses more memory when more space needed
- Contracts: uses less memory when the space no longer needed
- The stack pointer, SP (R13), is an ordinary ARM register that, by convention, points to the top of the stack. A pointer is a fancy name for a memory address. SP points to (gives the address of) data.
- The stack pointer (SP) starts at a high memory address and decrements to expand as needed.



Address	Data	
BEFFFFAE8	AB000001	← SP
BEFFFFAE4		
BEFFFFAE0		
BEFFFFADC		
⋮	⋮	

(a)

Address	Data	
BEFFFFAE8	AB000001	← SP
BEFFFFAE4	12345678	
BEFFFFAE0	FFEEDDCC	
BEFFFFADC		
⋮	⋮	

(b)

Memory

# Stack for functions

- One of the important uses of the stack is to save and restore registers that are used by a function.

# Stack for functions

- A function should calculate a return value but have no other unintended side effects. In particular, it should not modify any registers besides R0, the one containing  $\leftarrow$  the return value.

# Stack for functions

- The **diffofsums** function in Code Example 6.21 violates this rule because it modifies R4, R8, and R9. If main had been using these registers before the call to **diffofsums**, their contents would have been corrupted by the function call.

# Stack for functions ...

To solve the problems of registers to avoid unintended consequences:

1. A function saves registers on the stack before it modifies them, then restores them from the stack before it returns
2. Makes space on the stack to store the values of one or more registers
3. Stores the values of the registers on the stack
4. Executes the function using the registers
5. Restores the original values of the registers from the stack
6. Deallocates space on the stack

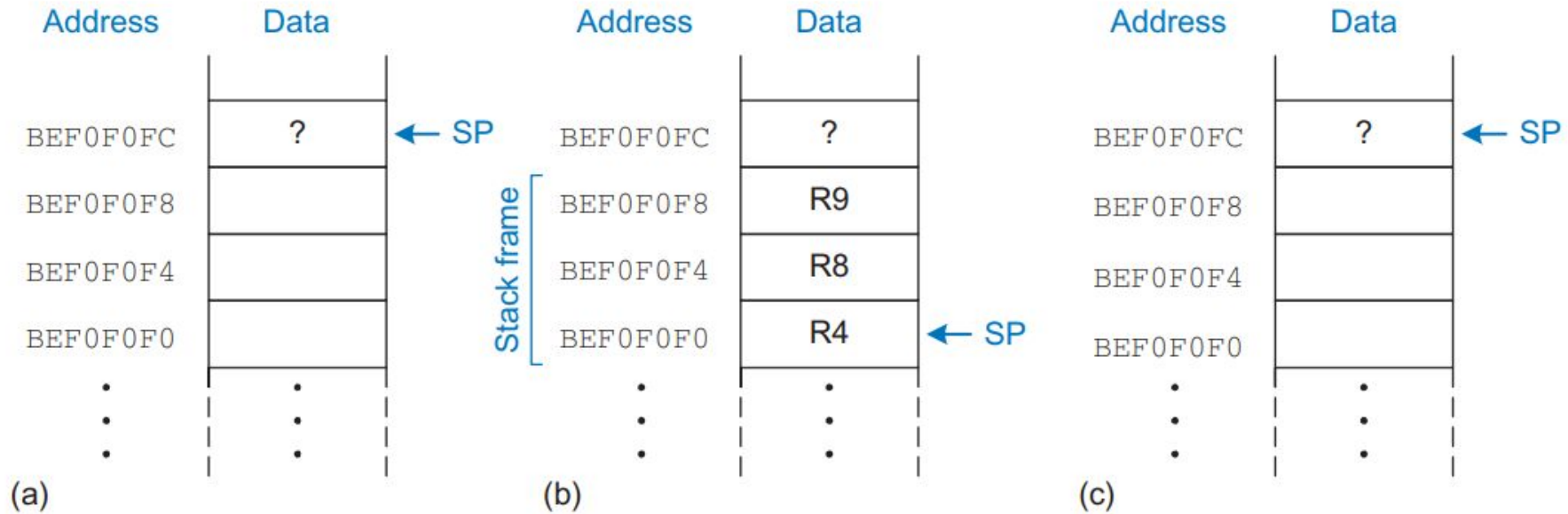


# Improved code for function in ARM

## ARM Assembly Code

```
@ 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
```

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



# Loading and Storing Multiple Registers

Saving and restoring registers on the stack is such a common operation that ARM provides Load Multiple and Store Multiple instructions (**LDM** and **STM**) that are optimized to this purpose.

STFD = Store Multiple Full Descending

Full means the fact that the memory address points to the location where the first register will be stored.

Descending refers to the stack grows downward in memory (i.e., the stack pointer decreases).

# Example

```
STMFD SP!, {R4, R5, R6, LR}
```

- The instruction stores the contents of registers R4, R5, R6, and the link register (LR) onto the stack.
- The stack pointer (SP) is decremented by 16 bytes (4 registers  $\times$  4 bytes each).
- After the operation, SP will point to the new top of the stack.

LDMFD = Load Multiple Full Descending

Load multiple registers from memory (typically the stack) in one operation.

# Example

```
LDMFD SP!, {R4, R5, R6, LR}
```

- This instruction loads the contents of the stack into registers R4, R5, R6, and the link register (LR).
- The stack pointer (SP) is incremented by 16 bytes (4 registers  $\times$  4 bytes each).
- After the operation, SP will point to the new top of the stack.

# Loading and Storing Multiple Registers

## ARM Assembly Code

@ R4 = result

DIFFOFSUMS

STMFD SP!, {R4, R8, R9} @ push R4/8/9 on full descending 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

LDMFD SP!, {R4, R8, R9} @ pop R4/8/9 off full descending stack

MOV PC, LR @ return to caller



Which series of signals will multiply a number Y by 145?

$R1 = Y$ ,

A

$R0 \leftarrow R1$   
 $R0 \leftarrow (R0 \ll 6)$   
 $R0 \leftarrow R0 + R1$   
 $R0 \leftarrow R0 + R1$   
 $R0 \leftarrow (R0 \ll 1)$   
 $R0 \leftarrow R0 + R1$   
 $R0 \leftarrow R0 + R1$

B

$R0 \leftarrow R1$   
 $R0 \leftarrow (R0 \ll 4)$   
 $R0 \leftarrow R0 + R1$   
 $R0 \leftarrow R0 + R1$   
 $R0 \leftarrow R0 + R1$   
 $R0 \leftarrow (R0 \ll 2)$

C

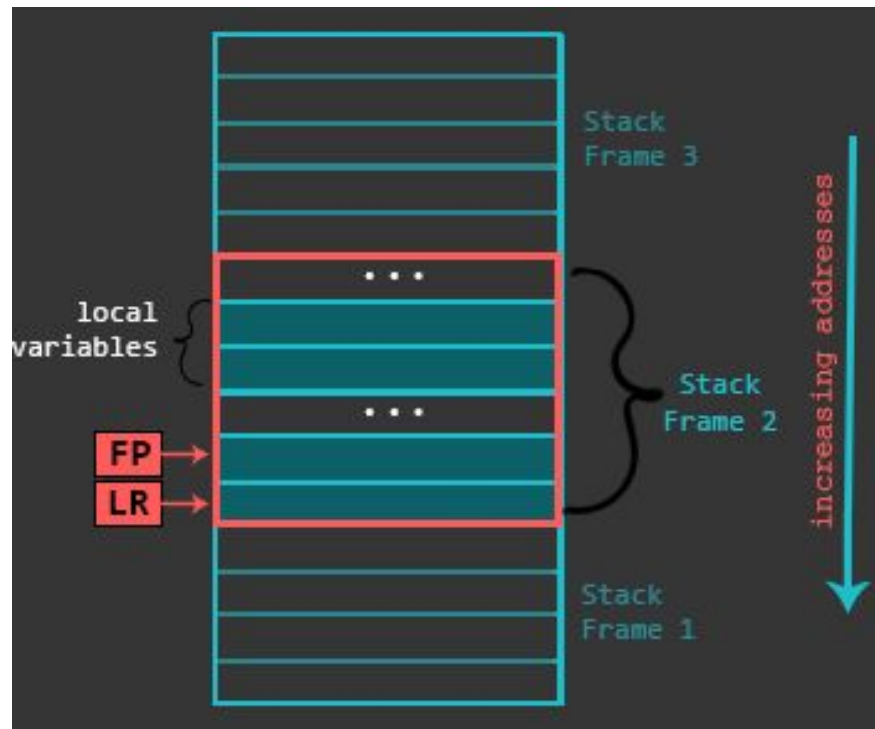
$R0 \leftarrow R1$   
 $R0 \leftarrow (R1 \ll 4)$   
 $R0 \leftarrow R0 + R1$   
 $R0 \leftarrow (R0 \ll 1)$   
 $R0 \leftarrow R0 + R1$   
 $R0 \leftarrow R0 + R1$   
 $R0 \leftarrow R0 + R1$   
 $R0 \leftarrow (R0 \ll 2)$   
 $R0 \leftarrow R0 - R1$   
 $R0 \leftarrow R0 - R1$   
 $R0 \leftarrow R0 - R1$

# Stack Pointers

1. The location where the next (32 bit) piece of information will be stored is defined by the Stack Pointer (SP), or, the memory address stored in the SP register.
2. Functions take advantage of Stack for saving local variables, preserving register state, etc.
3. To keep everything organized, functions use Stack Frames, a localized memory portion within the stack which is dedicated for a specific function.
4. A stack frame gets created during the function creation. The Frame Pointer (FP) is set to the bottom of the stack frame and then stack buffer for the Stack Frame is allocated.

# Frame Pointers

1. The stack frame (starting from its bottom) generally contains the return address (previous Link Register or LR), previous Frame Pointer, any registers that need to be preserved, function parameters (in case the function accepts more than 4), local variables, etc.
2. The actual contents of the Stack Frame may vary, the give example is one possible scenario. The Stack Frame gets destroyed when function execution finishes.



## C Code

```
int main() {
    int res = 0;
    int a = 1;
    int b = 2;
    res = max(a, b);
    return res;
}

int max(int a,int b) {
    do_nothing();
    if(a<b) {
        return b; }
    else {
        return a; }
}

int do_nothing() {
    return 0;
}
```

simple illustration  
of a Stack Frame  
through the  
perspective of  
GDB debugger.

```
gef> gef config context.layout "regs stack"
gef> nexti 13
0x00010464 in max ()
```

```
-----[ registers ]-----
$r0 : 0x00000002
$r1 : 0x00000002
$r2 : 0x00000001
$r3 : 0x00000002
$r4 : 0x00000000
$r5 : 0x00000000
$r6 : 0x000102c0 -> <_start+0> mov r11, #0
$r7 : 0x00000000
$r8 : 0x00000000
$r9 : 0x00000000
$r10 : 0xb6ffc000 -> 0x0002ff44
$r11 : 0xbffff254 -> 0x00010418 -> <main+48> str r0, [r11, #-8]
$r12 : 0xb6fb1000 -> 0x0013cf20
$sp : 0xbffff248 -> 0x00000002
$lr : 0x00010444 -> <max+24> ldr r2, [r11, #-8]
$pc : 0x00010464 -> <max+56> sub sp, r11, #4
$cpsr : [thumb fast interrupt overflow carry zero NEGATIVE]
-----[ stack ]-----
```

```
0xbffff248|+0x00: 0x00000002 <-$sp
0xbffff24c|+0x04: 0x00000001
0xbffff250|+0x08: 0xbffff26c -> 0xb6e8c294 -> <_libc_start_main+276> bl 0xb6ea4h28 <__GI_exit>
0xbffff254|+0x0c: 0x00010418 -> <main+48> str r0, [r11, #-8] <-$r11
0xbffff258|+0x10: 0x00000000
0xbffff25c|+0x14: 0x00000002
0xbffff260|+0x18: 0x00000001
0xbffff264|+0x1c: 0x00000000
```

```
gef> disassemble max
```

```
Dump of assembler code for function max:
```

```
0x0001042c <+0>: push {r4, r5, r6, r7, r8, r9, r10, r11}
0x00010430 <+4>: add r11, sp, #0
0x00010434 <+8>: sub r11, sp, #0
0x00010438 <+12>: str r11, [r11, #-4]
0x0001043c <+16>: str r11, [r11, #-8]
0x00010440 <+20>: bl 0xb6e8c294
0x00010444 <+24>: ldr r2, [r11, #-8]
0x00010448 <+28>: ldr r2, [r11, #-8]
0x0001044c <+32>: cmn r2, r2, #0
0x00010450 <+36>: bgt r2, r2, #0
0x00010454 <+40>: ldr r2, [r11, #-8]
0x00010458 <+44>: b r2, r2, #0
0x0001045c <+48>: ldr r2, [r11, #-8]
0x00010460 <+52>: mov r0, r2
=> 0x00010464 <+56>: sub sp, r11, #4
0x00010468 <+60>: pop {r4, r5, r6, r7, r8, r9, r10, r11}
End of assembler dump.
```


In this situation, we are about to leave the function **max** (see the arrow in the disassembly at the bottom). At this state, the FP (R11) points to 0xbffff254 which is the bottom of our Stack Frame. This address on the Stack (green addresses) stores 0x00010418 which is the return address (previous Link Register). 4 bytes above this (at 0xbffff250) we have a value 0xbffff26c, which is the address of a previous Frame Pointer. The 0x1 and 0x2 at addresses 0xbffff24c and 0xbffff248 are local variables which were used during the execution of the function max. So the Stack Frame which we just analyzed had only LR, FP and two local variables.

# Preserved vs Non-preserved Registers

- If the calling function does not use registers that are saved and restored upon returning from the callee, the effort to save and restore them is wasted.
- To avoid this waste, ARM divides registers into preserved (R0-R3, R12, SP, LR) and non-preserved categories.
- A function must save and restore any of the preserved registers that it wishes to use, but it can change the nonpreserved registers freely.

# Less number of preserved registers

## ARM Assembly Code

```
@ R4 = result
DIFFOFSUMS

PUSH {R4} @ save R4 on stack
ADD R1, R0, R1 @ R1 = f + g
ADD R3, R2, R3 @ R3 = h + i
SUB R4, R1, R3 @ result = (f + g) - (h + i)
MOV R0, R4 @ put return value in R0
POP {R4} @ pop R4 off stack
MOV PC, LR @ return to caller
```

Saves only R4 on the stack.

The callee must save and restore any preserved registers that it wishes to use. The callee may change any of the non-preserved registers. Hence, if the caller is holding active data in a non-preserved register, the caller needs to save that non-preserved register before making the function call and then needs to restore it afterward. For these reasons, preserved registers are also called **callee-save**, and non-preserved registers are called **caller-save**.



## Non-leaf Function Calls

# Leaf and Non-leaf Functions

- A function that does not call others is called a **leaf function**.
- A function that does call others is called a **non-leaf function**.
- Non-leaf functions may need to save non-preserved registers on the stack before they call another function and then restore those registers afterward.
  - **Caller save rule:** Before a function call, the caller must save any non-preserved registers (R0–R3 and R12) that it needs after the call. After the call, it must restore these registers before using them.
  - **Callee save rule:** Before a callee disturbs any of the preserved registers (R4–R11 and LR), it must save the registers. Before it returns, it must restore these registers.
- A non-leaf function overwrites LR when it calls another function using BL. Thus, a non-leaf function must always save LR on its stack and restore it before returning.



# Non-leaf Function Calls

## High-level Code

```
int f1(int a, int b) {  
    int i, x;  
    x = (a + b)*(a - b);  
    for (i=0; i<a; i++)  
        x = x + f2(b+i);  
    return x;  
}
```

```
int f2(int p) {  
    int r;  
    r = p + 5;  
    return r + p;  
}
```

Non Leaf  
Function

## ARM Assembly Code

```
@ R0 = a, R1 = b, R4 = i, R5 = x  
F1  
PUSH {R4, R5, LR} @ save preserved registers used by f1  
    ADD R5, R0, R1 @ x = (a + b)  
    SUB R12, R0, R1 @ temp = (a - b)  
    MUL R5, R5, R12 @ x = x * temp = (a + b) * (a - b)  
    MOV R4, #0 @ i = 0  
FOR  
    CMP R4, R0 @ i < a?  
    BGE RETURN @ no: exit loop  
    PUSH {R0, R1} @ save non-preserved registers  
    ADD R0, R1, R4 @ argument is b + i  
    BL F2 @ call f2(b+i)  
    ADD R5, R5, R0 @ x = x + f2(b+i)  
    POP {R0, R1} @ restore non-preserved registers  
    ADD R4, R4, #1 @ i++  
    B FOR @ continue for loop  
RETURN  
    MOV R0, R5 @ return value is x  
    POP {R4, R5, LR} @ restore preserved registers  
    MOV PC, LR @ return from f1  
@ R0 = p, R4 = i  
F2  
    PUSH {R4} @ save preserved registers used by f2  
    ADD R4, R0, #5 @ r = p + 5  
    ADD R0, R4, R0 @ return value is r + p  
    POP {R4} @ restore preserved registers  
    MOV PC, LR @ return from f2
```

# Non-leaf vs Leaf Function

## ARM Assembly Code

```
/*@ A prologue of a non-leaf function */
```

```
PUSH    {R11, LR}    @/* Start of the prologue. Saving Frame Pointer and LR onto the  
stack */
```

```
ADD     R11, SP, #0   @/* Setting up the bottom of the stack frame */
```

```
SUB     SP, SP, #16   @/* End of the prologue. Allocating some buffer on the stack */
```

```
/*@ A prologue of a leaf function */
```

```
PUSH    {R11}         @/* Start of the prologue. Saving Frame Pointer onto the stack */
```

```
ADD     R11, SP, #0   @/* Setting up the bottom of the stack frame */
```

```
SUB     SP, SP, #12   @/* End of the prologue. Allocating some buffer on the stack */
```

# Recursive Function Calls

- Calls itself
- Non-leaf
- Behave as both: calle and caller
- Must save both: preserved and non-preserved registers

# Recursion in ARM

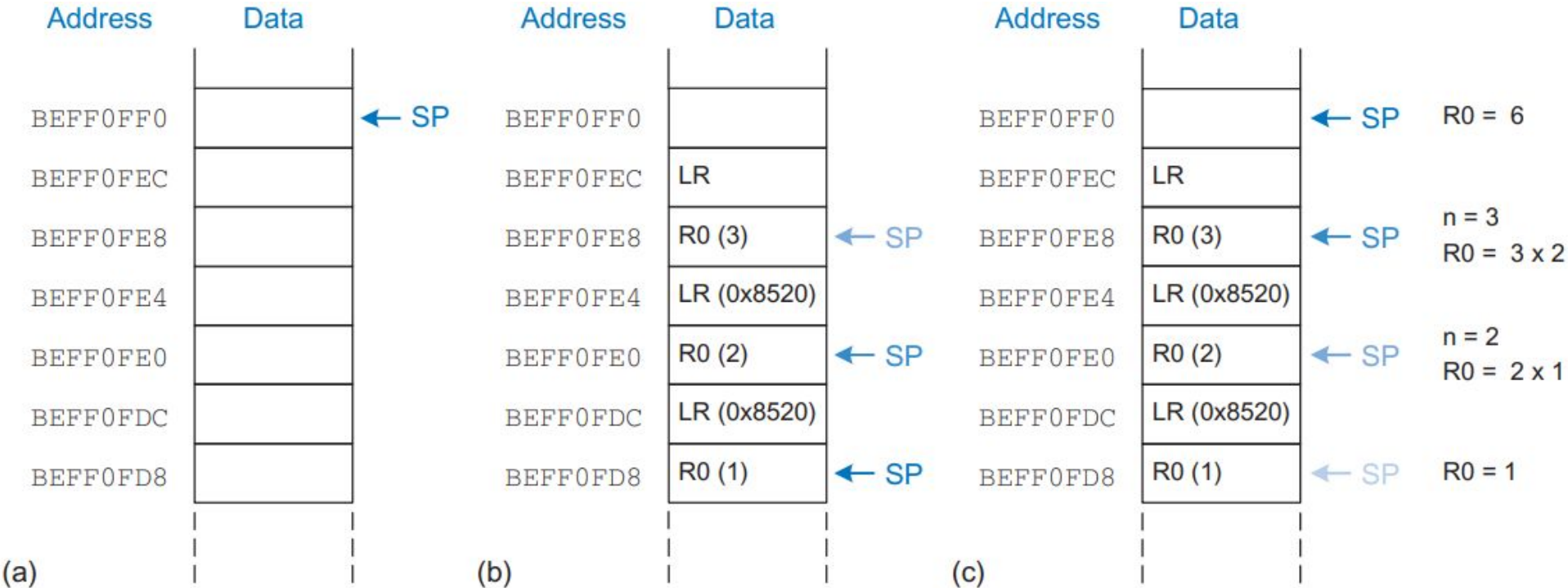
## High-level Code

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

## ARM Assembly Code

```
@ push n and LR on stack  
0x8500 FACTORIAL PUSH {R0, LR}  
0x8504 CMP R0, #1 @ R0 ≤ 1?  
0x8508 BGT ELSE @ no: branch to else  
0x850C MOV R0, #1 @ otherwise, return 1  
0x8510 ADD SP, SP, #8 @ restore SP  
0x8514 MOV PC, LR @ return  
0x8518 ELSE SUB R0, R0, #1 @ n = n - 1  
0x851C BL FACTORIAL @ recursive call  
0x8520 POP {R1, LR} @ pop n (into R1) and LR  
0x8524 MUL R0, R1, R0 @ R0 = n*factorial(n - 1)  
0x8528 MOV PC, LR @ return
```

# Stack during factorial(3) execution



# Functions with more than 4 arguments, too many local variables

- If a function has more than four arguments, the first four are passed in the argument registers as usual. Additional arguments are passed on the stack, just above SP.
- The caller must expand its stack to make room for the additional arguments.
- Local variables are stored in R4-R11; if there are too many local variables, they can also be stored in the function's stack frame.

