



CPE 221: Computer Organization

07 ARM Stacks and Frames
rahul.bhadani@uah.edu

Rahul Bhadani

Generally speaking, the Stack is a memory region within the program/process. This part of the memory gets allocated when a process is created. We use Stack for storing temporary data such as local variables of some function, environment variables which helps us to transition between the functions, etc.

PUSH and POP

We interact with the stack using PUSH and POP instructions.

PUSH and POP are aliases to some other memory related instructions rather than real instructions, but we use PUSH and POP for simplicity reasons.

Growing a Stack

When we say that Stack grows, we mean that an item (32 bits of data) is put on to the Stack. The stack can grow UP (when the stack is implemented in a Descending fashion) or DOWN (when the stack is implemented in a Ascending fashion).

Stack Pointer (SP)

The actual location where the next (32 bit) piece of information will be put is defined by the **Stack Pointer**, or to be precise, the memory address stored in the **SP** register.

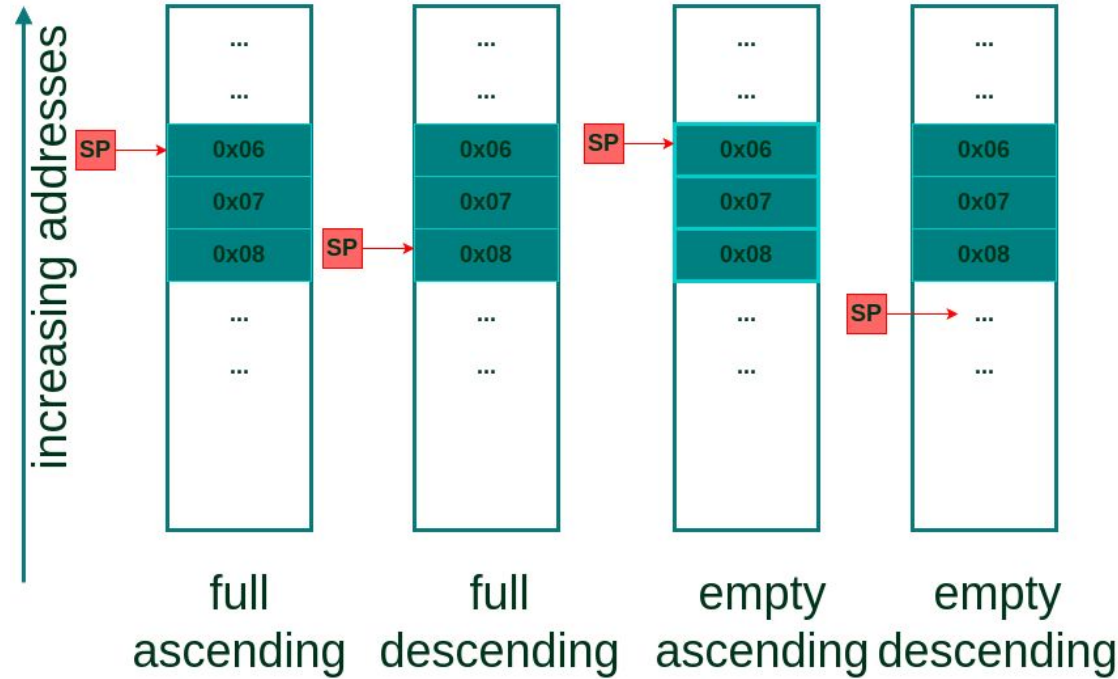
How does Stack Pointer (SP) work?

The address could be pointing to the current (last) item in the stack or the next available memory slot for the item.

If the SP is currently pointing to the last item in the stack (Full stack implementation) the SP will be decreased (in case of Descending Stack) or increased (in case of Ascending Stack) and only then the item will be placed in the Stack.

If the SP is currently pointing to the next empty slot in the Stack, the data will be first placed and only then the SP will be decreased (Descending Stack) or increased (Ascending Stack).

Stack Implementation Example



Different Stack Implementation

As a summary of different Stack implementations we can use the following table which describes which Store Multiple/Load Multiple instructions are used in different cases.

Stack Type	Store	Load
Full descending	STMGD (STMDB, Decrement Before)	LDMGD (LDM, Increment after)
Full ascending	STMGA (STMIB, Increment Before)	LDMGFA (LDMDB, Decrement After)
Empty descending	STMED (STMDB, Decrement After)	LDMED (LDMIB, Increment Before)
Empty ascending	STMEA (STM, Increment after)	LDMEA (LDMDB, Decrement Before)

In our examples, we will use the Full descending Stack.

Example

ARM Assembly Code

```
.global _start
_start:

    MOV    R0, #2    /* set up R0 */
    PUSH  {R0}       /* save R0 onto the stack */
    MOV    R0, #3    /* overwrite R0 */
    POP    {R0}       /* restore R0 to it's initial state */
    BX     LR         /* finish the program */

done: B done
```

Example 1

Assume, at the beginning, the Stack Pointer points to address 0xbffff6f8, which represents the last item in the Stack. At this moment, we see that it stores some value (just an example):

0xbffff6f8: 0xb6fc7000

Example 1 Continues

After executing the first (MOV) instruction, nothing changes in terms of the Stack.

When we execute the PUSH instruction, the following happens:

- First, the value of SP is decreased by 4 (4 bytes = 32 bits).
- Then, the contents of R0 are stored to the new address specified by SP.
- When we now examine the updated memory location referenced by SP, we see that a 32 bit value of integer 2 is stored at that location:

0xbffff6f4: 0x00000002

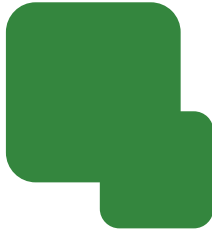
Example 1 Continues

The instruction (`MOV R0, #3`) in our example is used to simulate the corruption of the R0.

We then use `POP` to restore a previously saved value of R0.

When the `POP` gets executed, the following happens:

- First, 32 bits of data are read from the memory location (`0xbefff6f4`) currently pointed by the address in `SP`.
- Then, the `SP` register's value is increased by 4 (becomes `0xbefff6f8` again).
- The register `R0` contains integer value 2 as a result.



Stacks in Functions

Function Takes Advantage of Stacks

Functions take advantage of Stack for saving local variables, preserving register state, etc.

To keep everything organized, functions use Stack Frames, a localized memory portion within the stack which is dedicated for a specific function.

Stack Frame

A stack frame gets created in the prologue (more about this in the next section) of a function.

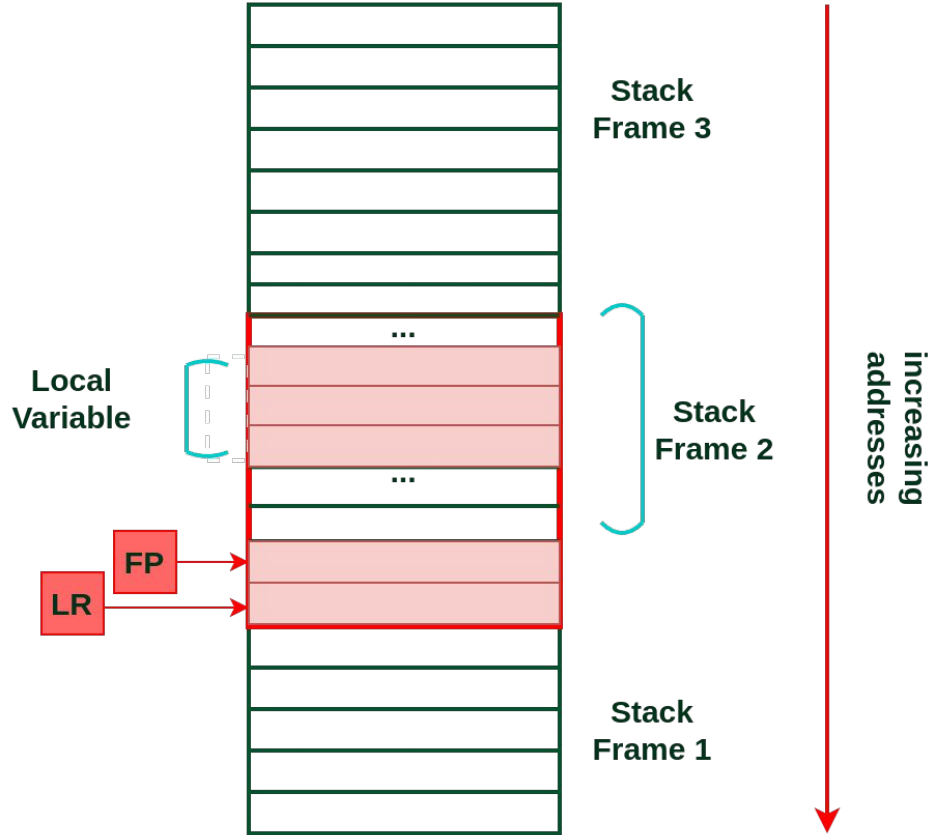
Stack Frame

The Frame Pointer (FP) is set to the bottom of the stack frame and then stack buffer for the Stack Frame is allocated.

The stack frame (starting from it's bottom) generally contains the return address (previous LR), previous Frame Pointer, any registers that need to be preserved, function parameters (in case the function accepts more than 4), local variables, etc.

While the actual contents of the Stack Frame may vary, the ones outlined before are the most common. Finally, the Stack Frame gets destroyed during the epilogue of a function.

Illustration of a Stack Frame within the Stack:



Example 2

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; }
```

Example 2 Continues ...

```
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 : 0xb6ff0000 -> 0x0002ff44
$r11 : 0xb6fff254 -> 0x00010418 -> <main+48> str r0, [r11, #-8]
$r12 : 0xb6ff1000 -> 0x0013cf20
$sp : 0xb6fff248 -> 0x00000002
$lr : 0x00010444 -> <main+24> ldr r2, [r11, #-8]
$pc : 0x00010464 -> <main+54> sub sp, r11, #4
$cpsr : [thumb fast interrupt overflow carry zero NEGATIVE]
-----[ stack ]-----
0xb6fff248|+0x00: 0x00000002 <--$sp
0xb6fff24c|+0x04: 0x00000001
0xb6fff250|+0x08: 0xb6fff26c -> 0xb6e8c294 -> <_libc_start_main+276> bl 0xb6e4db28 <__GI_exit>
0xb6fff254|+0x0c: 0x00010418 -> <main+48> str r0, [r11, #-8] <--$r11
0xb6fff258|+0x10: 0x00000000
0xb6fff25c|+0x14: 0x00000002
0xb6fff260|+0x18: 0x00000001
0xb6fff264|+0x1c: 0x00000000
gef> disassemble max
Dump of assembler code for function max:
0x0001042c <+0>: push {r11, lr}
0x00010430 <+4>: add r11, sp, #4
0x00010434 <+8>: sub sp, sp, #8
0x00010438 <+12>: str r0, [r11, #-8]
0x0001043c <+16>: str r1, [r11, #-12]
0x00010440 <+20>: bl 0x1046c <do nothing>
0x00010444 <+24>: ldr r2, [r11, #-8]
0x00010448 <+28>: ldr r3, [r11, #-12]
0x0001044c <+32>: cmp r2, r3
0x00010450 <+36>: bge 0x1045c <max+48>
0x00010454 <+40>: ldr r3, [r11, #-12]
0x00010458 <+44>: b 0x10460 <max+52>
0x0001045c <+48>: ldr r3, [r11, #-8]
0x00010460 <+52>: mov r0, r3
=> 0x00010464 <+56>: sub sp, r11, #4
0x00010468 <+60>: pop {r11, pc}
End of assembler dump.
```

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 0xb6fff254 which is the bottom of our Stack Frame.

This address on the Stack (green addresses) stores 0x00010418 which is the return address (previous LR).

4 bytes above this (at 0xb6fff250) we have a value 0xb6fff26c, which is the address of a previous Frame Pointer.

The 0x1 and 0x2 at addresses 0xb6fff24c and 0xb6fff248 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.

Prologue, Body and Epilogue of the Function

1. Prologue sets up the environment for the function;
2. Body implements the function's logic and stores result to R0;
3. Epilogue restores the state so that the program can resume from where it left off before calling the function.