THE UNIVERSITY OF
ALABAMA IN HUNTSVILLE

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.

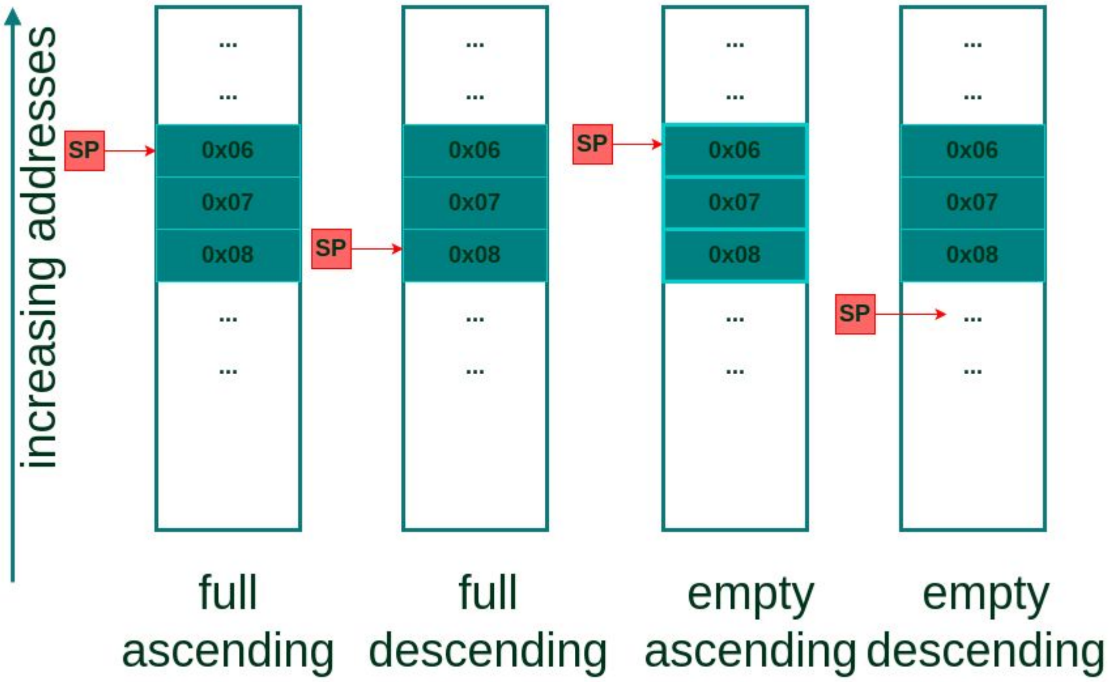
How does Stack Pointer (SP) work?

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 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	STMFD (STMDB, Decrement Before)	LDMFD (LDM, Increment after)
Full ascending	STMFA (STMIB, Increment Before)	LDMFA (LDMDA, Decrement After)
Empty descending	STMED (STMDA, 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 0xbefff6f8, which represents the last item in the Stack. At this moment, we see that it stores some value (just an example):

```
0xbefff6f8: 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:

```
0xbefff6f4: 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.

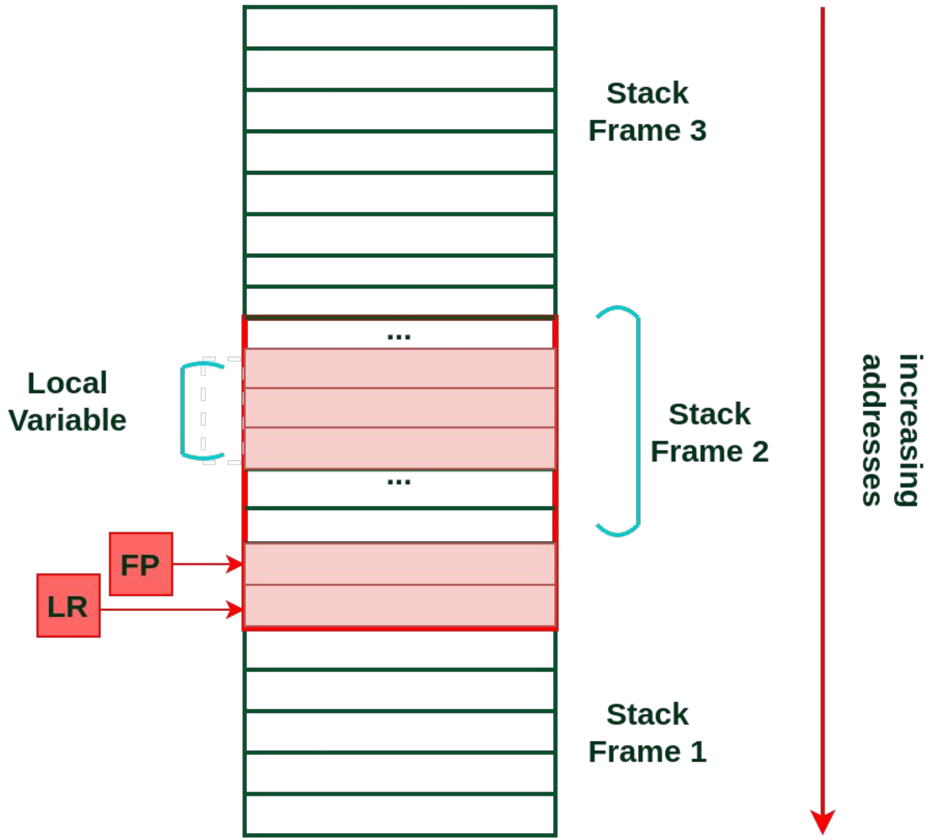
Stack Frame

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  : 0xb6ffc000 -> 0x0002ff44
$r11  : 0xbefff254 -> 0x00010418 -> <main+48> str r0,  [r11,  #-8]
$r12  : 0xb6fb1000 -> 0x0013cf20
$sp   : 0xbefff248 -> 0x00000002
$lr   : 0x00010444 -> <max+24> ldr r2,  [r11,  #-8]
$pc   : 0x00010464 -> <max+56> sub sp,  r11,  #4
$cpsr : [thumb fast interrupt overflow carry zero NEGATIVE]
-----------------------------------------------------------------------------[ stack ]----
0xbefff248|+0x00: 0x00000002     <-$sp
0xbefff24c|+0x04: 0x00000001
0xbefff250|+0x08: 0xbefff26c -> 0xb6e8c294 -> <__libc_start_main+276> bl 0xb6ea4b28 <__GI_exit>
0xbefff254|+0x0c: 0x00010418 -> <main+48> str r0,  [r11,  #-8]  <-$r11
0xbefff258|+0x10: 0x00000000
0xbefff25c|+0x14: 0x00000002
0xbefff260|+0x18: 0x00000001
0xbefff264|+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.
```

Frame Pointer

Stack Frame

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 0xbefff254 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 0xbefff250) we have a value 0xbefff26c, which is the address of a previous Frame Pointer.

The 0x1 and 0x2 at addresses 0xbefff24c and 0xbefff248 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.

Example: Swap – Pass by Reference

C Code

```
void swap (int*, int*);

 void main (void)
 {
   int x = 2, y = 3;
  swap (&x, &y);        /* swap x and y */
 }

 void swap (int *a, int *b)
 {
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
 }
```

Example: Swap – Pass by Reference

ARM Assembly Code

```
        MOV     SP, #0x00000000  @ (0)
        MOV     FP, #0xFFFFFF00  @ (4)
        B       main             @ (
swap:   PUSH    {FP}             @ (12)
        MOV     FP, SP           @ (16)
        SUB     SP, SP, #4       @ (20)
        LDR     R1, [FP, #4]     @ get address of parameter a (24)
        LDR     R2, [R1]         @ get value of parameter a (28)
        STR     R2, [FP, #-4]    @ store a in temp in stack frame (32)
        LDR     R0, [FP, #8]     @ get address of parameter b (36)
        LDR     R3, [R0]         @ get value of parameter b (40)
        STR     R3, [R1]         @ store parameter b in parameter a 44)
        LDR     R3, [FP, #-4]    @ get temp from stack frame (48)
        STR     R3, [R0]         @ store temp in b (52)
        MOV     SP, FP           @ Collapse stack frame; restore sp (56)
        POP     {FP}             @ (60)
        MOV     PC, LR           @ (64)
```

```
main:   PUSH    {FP}             @ (68)
        MOV     FP, SP           @ (72)
        SUB     SP, SP, #8       @ (76)
        ADR     R6, y            @ store y on the stack (80)
        STR     R6, [fp, #-4]    @ store &y on the stack (84)
        ADR     R6, x            @ (88)
        STR     R6, [fp, #-8]    @ store &x on the stack (92)
        BL      swap             @ (96)
        MOV     SP, FP           @ Collapse frame; restore sp (100)
        POP     {FP}             @ (104)
stop:   B       stop             @ (108)
x:     .word    2                @ (112)
y:     .word    3                @ (116)
```

Example: Multiply_by_Adding with Subroutines

C Code

```c
int mpy_ne(int, int)
int abs(int)
int main() {
int first = 8;
int second = -9;
int result;
result = mpy_ne(first, second);
}
```

```c
int  mpy_ne (int num1; int num2){
     int a, b, mult;
     a = abs(num1);
     b = abs(num2);
     mult = 0;
     for (i = 0; i < a; i++)
          mult = mult + b;
     if (num1 < 0) mult = -mult;
     if (num2 < 0) mult = -mult;
     return mult;
}
int abs(int x){
     if (x < 0) x = -x;
          return x;
}
```

Example: Multiply_by_Adding with Subroutines

ARM Assembly Code

```
             LDR   R1, num1         @ Put num1 in r1.
             LDR   R2, num2         @ Put num2 in r2.
             ADR   R9, result
             MOV   R3, #0           @ Set r3 to 0, it will hold
the result.
             TEQ   R1, #0           @ Compare first num to 0
             BEQ   done             @ If first num is 0 done,
result = 0.
             TEQ   R2, #0           @ Compare second numto 0
             BEQ   done             @ If second numis 0, done,
result = 0.
             CMP   R1, #0
             RSBMI R4, R1, #0
             CMP   R2, #0
             RSBMI R5, R2, #0
adding:      ADD   R3, R3, R5       @ Add num2.
             SUBS  R4, R4, #1       @ Decrement r4, the abs of
num1.
             BEQ   adjust           @ If r4 = 0, done adding, go
to adjust.
             B     adding           @ Otherwise, need to add
again.
```

```
adjust:      MOVS  R1, R1                @ Done adding, now
adjust result.
             RSBMI R3, R3, #0            @ If num2 negative,
negate result.
             MOVS  R2, R2
             RSBMI R3, R3, #0   @ If num1 negative, negate
result.
done:        STR   r3, [r9]
num1:        .word   -8            @ Give num1 a value
num2:        .word   -9            @ Give num2 a value
result:      .space 4
             .end
```

Difference between TEQ and CMP

1. `TEQ Rn, Operand2`: Tests the equivalence of the value in register Rn with Operand2. It performs a bitwise XOR operation between Rn and Operand2 (without storing the result) and updates the condition flags based on the result of the XOR. Only N and Z flags are affected.

2. `CMP Rn, Operand2`: Compares the value in register Rn with Operand2 (which can be a register, immediate value, or shifted register). It subtracts Operand2 from Rn (without storing the result) and updates the condition flags based on the result of the subtraction. All flags are affected.

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 of before calling the function.