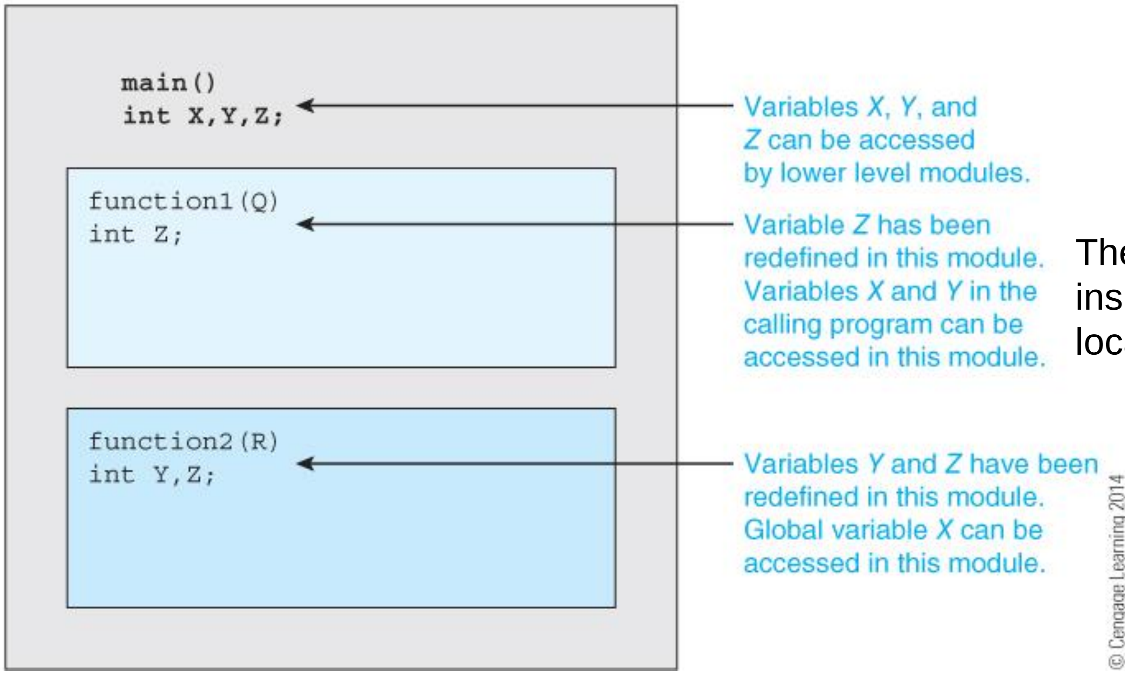
3.7 Stack, parameter passing and local variables

Stack frame

- In the subroutine section, we learned how to:
  - pass parameter in a subroutine
  - return a value in a register
  - preserve registers on a stack
- The mechanism would not be sufficient if:
  - we need more parameters than can be supported by built-in registers
  - we need to set aside local variables within a subroutine
- *Stack frame* is a more sophisticated mechanism for passing parameters, handling local variables and preserving registers.
- Some concepts before introducing the stack frame …..

Local and global variables

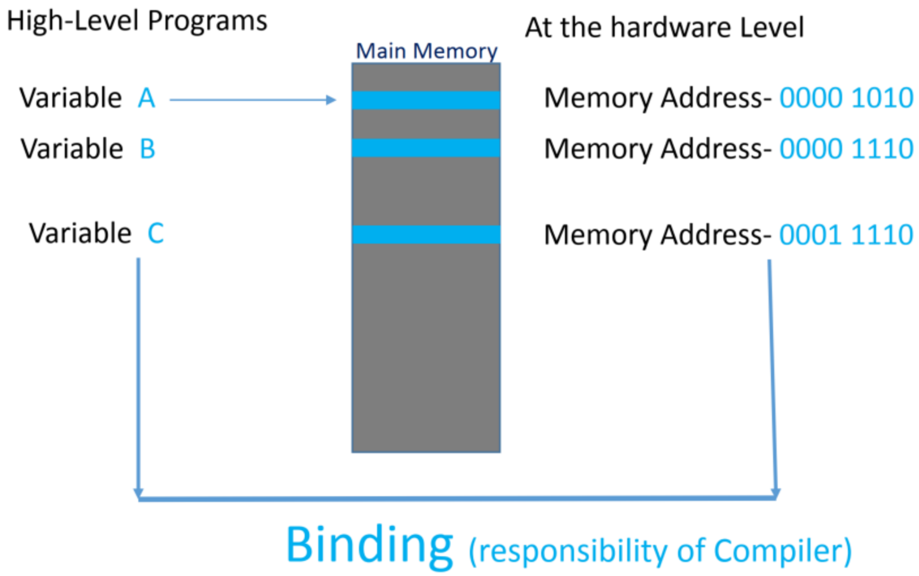
FIGURE 4.1 The concept of scope



The local variable Z is only accessible inside function1. The scope of the local variable Z is within function 1.

Binding

- Binding: Associating a variable with its storage location
- Static binding:
  – this association is made at compile time.



- Dynamic binding:
  – this association is made at run-time.

Recursion

- Recursive function: a function calling itself.

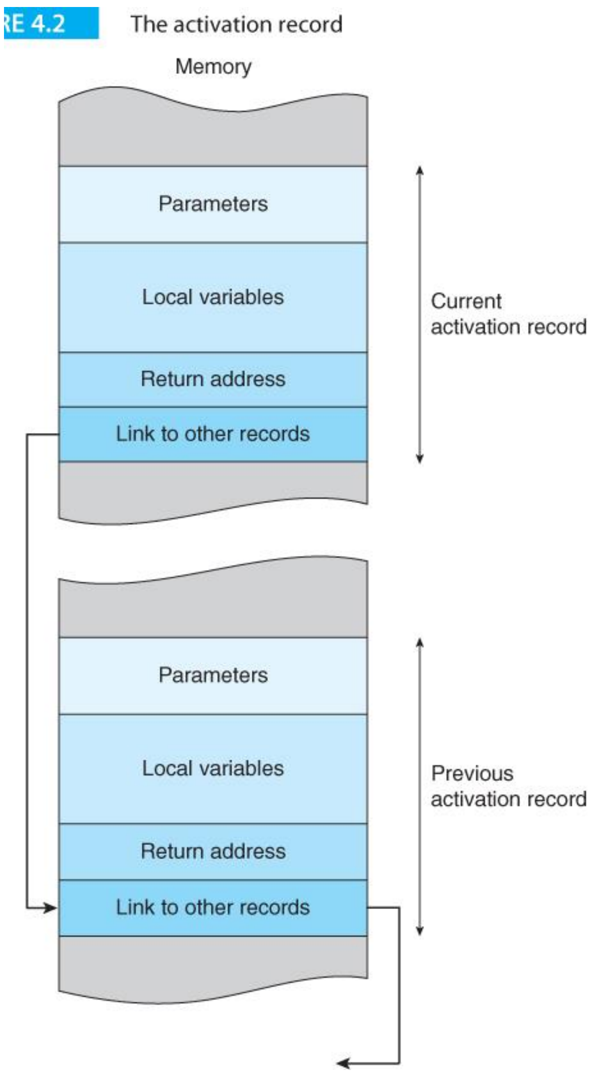
```
int Factorial(int n)
{
  if(n == 1)
    return 1;
  else
    return n * Factorial(n-1);
}
```

$$N! = N \times (N-1)!$$

- The local variable `n` exists at different levels of the call.
- Static binding does not permit recursion.
- We need dynamic data storage discussed below.

The stack and dynamic data storage

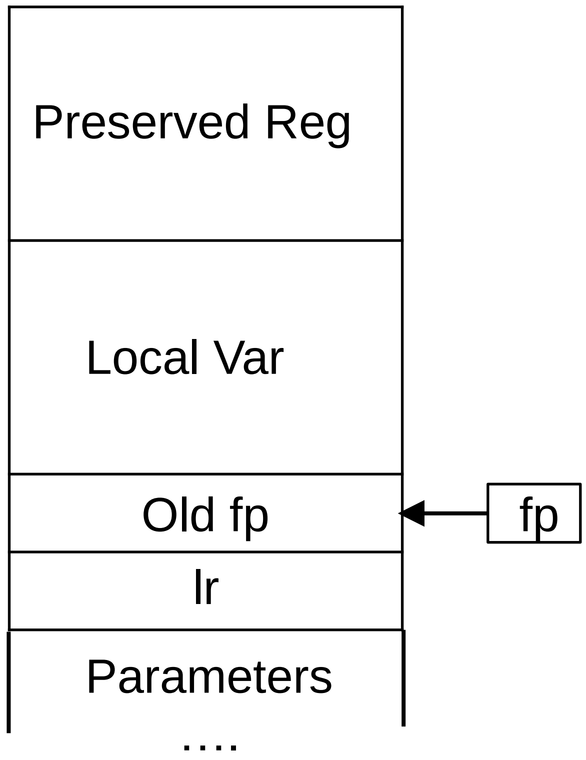
- When a subroutine is called, it is said to be *activated*.
- Each call is associated with an *activation record* (also called as *stack frame*)
- The activation record is the subroutine's view of the world.
- Recursion requires dynamic data storage because storage must be allocated at *runtime.*
- Executing a return from a subroutine *deallocates* or frees the storage taken up by the record.



Stack frame in subroutine calling

The stack frame involves three major registers:

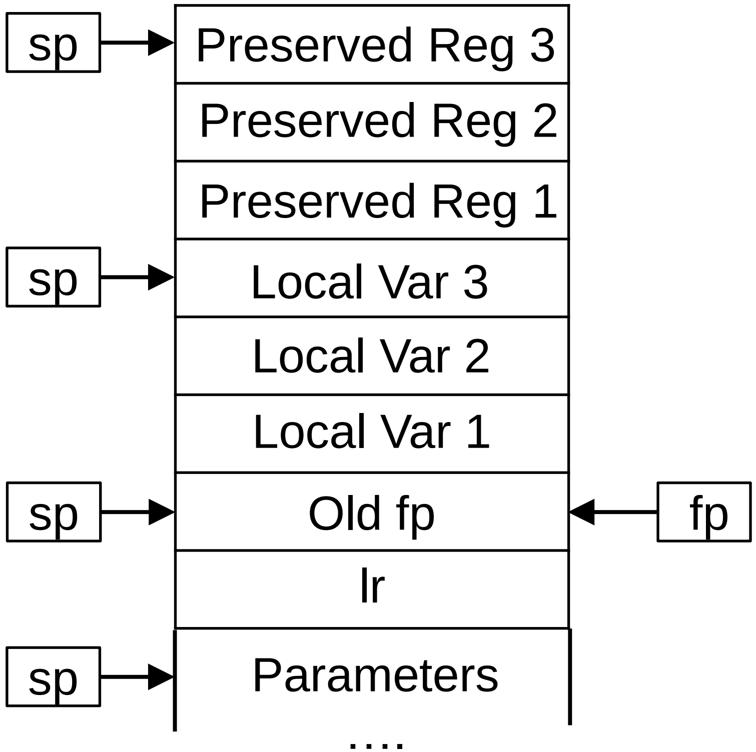
1. Stack pointer (sp) contains the address (points to) the top of stack.
2. Link register (lr) keeps track of the program line to return to after the subroutine finishes.
3. Frame pointer (fp) provides a reference point in the current activation record for users to locate parameters, local variables, saved link registers and preserved registers.



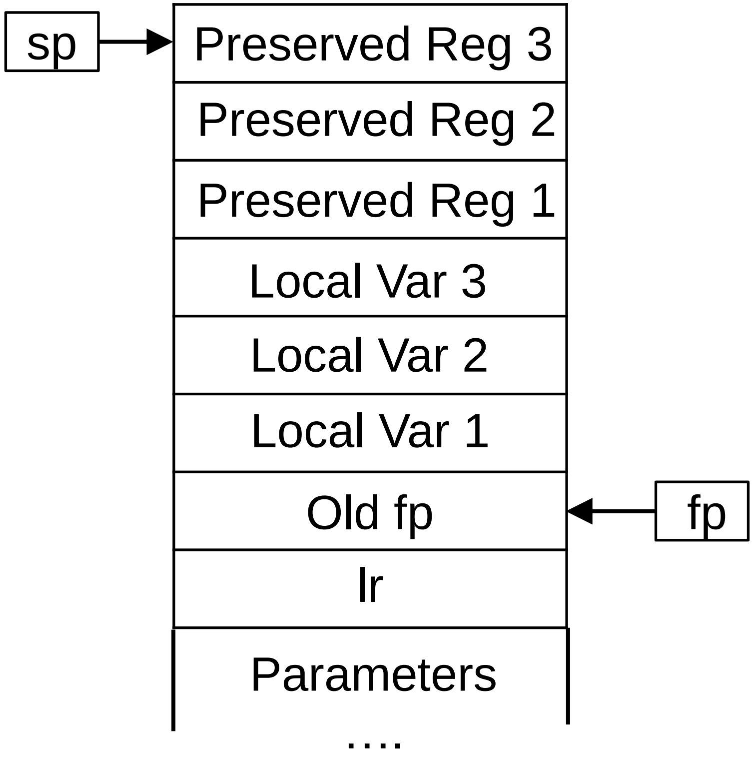
Stack frame prologue

The prologue of a subroutine consists of the following steps:

1. Push the fp and lr onto the stack
   - fp is being preserved
   - The subroutine is called by the Branch Link (bl) instruction, which stores the instruction immediately following the subroutine call into lr.
2. Save the current sp to the fp.
3. Allocate local variable storage space on the stack.
4. Push register to preserve onto the stack



After prologue



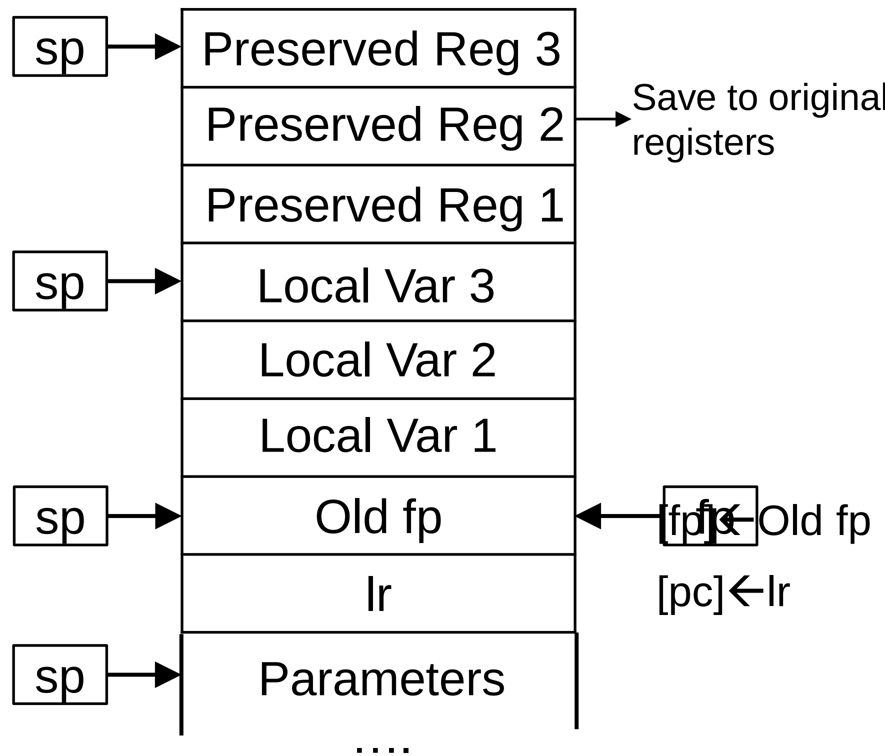
ARM code implementing prologue

```
Sub:
stmfd sp!, {fp, lr} //Step 1

mov fp, sp //Step 2

sub sp, sp, #4 //Step 3: assuming
only one local variable

stmfd sp!, {r1-r3} // Step 4:
assuming r1 to r3 are used in the
subroutine
```

Stack frame epilogue

The epilogue of a subroutine consists of the following steps:

1. Pop preserved registers from the stack.
2. Deallocate local variable storage from the stack
3. Pop fp and pc from the stack
   – the pc now contains the saved lr → Execution resumes at the line after the subroutine call

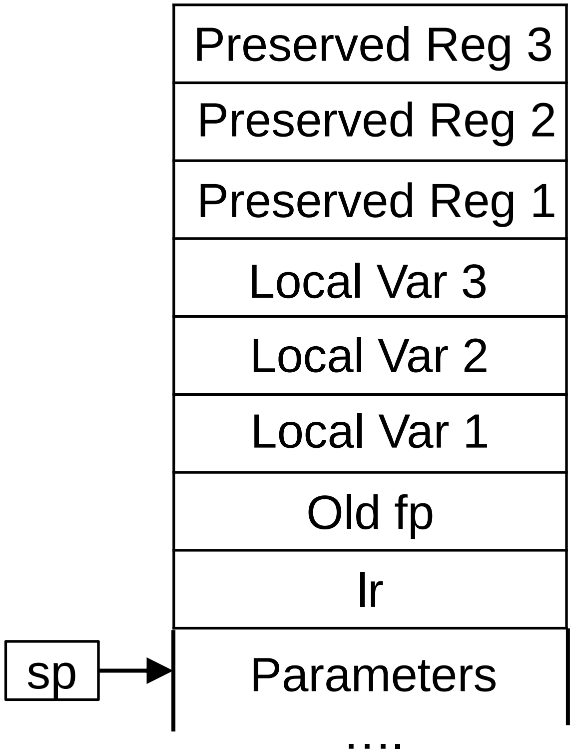


ARM code implementing prologue

```
ldmfd sp!, {r1-r3} // Step 1

add sp, sp, #4 // Step 2

ldmfd sp!, {fp, pc} // Step 3
```

State after epilogue

- The state of memory, except for the returned values in `r0` and updated parameter values, should be the same as before the subroutine was called.
- *Local variables* are called “local” because after the subroutine is done, references to all these local variables are removed (i.e., they cannot be accessed).
- Note that popped and deallocated values are still there, as the stack frame does not “clean up” memory used in the stack, but they are no longer kept track of by `sp` and will be overwritten by other uses of the stack.



Before and after calling a subroutine

Before calling the subroutine:

- Push parameter(s) to the stack

After returning from the subroutine

- Deallocate parameter(s) from the stack

Example 1: Putting it together

- Write a program that calculates the value of f(x) where
  - $f(x) = g(x) + h(x)$,

    where $g(x) = x^2$ and $h(x) = x+0x18$
- Use r1 to store g
- Use r2 to store h
- The output is stored in r0
- Need to preserve r1 and r2
- No local variable

```
.global _start
_start:

mov sp, #0
mov fp, #0
ldr r1, =0x08
stmfd sp!, {r1} // push parameter to stack (Point 1)
bl Sub
add sp, sp, #4 // Deallocate the parameter (Point 7)

_stop:
b _stop

Sub:
//Prologue
stmfd sp!, {fp, lr} // Step 1 (Point 2)
mov fp, sp //Step 2 (Point 3)
//Step 3: No local variable
stmfd sp!, {r1-r3} // Step 4: Preserve r1 and r2 (Point 4)

ldr r3, [fp, #8] // Retrieve parameter and store in r3
mul r1, r3, r3   // Use r1 to store g
add r2, r3, #0x18 // Use r2 to store h

add r0, r1, r2 // Compute sum of g and h and save in
               // r0 as the return value.

//Epilogue
ldmfd sp!, {r1-r3} // Step 1 (Point 5)
// Step 2: No local variable
ldmfd sp!, {fp, pc} // Step 3 (Point 6)

.end
```

Pt	Stack	sp	fp
1	ffffffe0 aaaaaaaa aaaaaaaa aaaaaaaa aaaaaaaa fffffff0 aaaaaaaa aaaaaaaa aaaaaaaa 00000008	FFFFFFFC	0
2	ffffffe0 aaaaaaaa aaaaaaaa aaaaaaaa aaaaaaaa fffffff0 aaaaaaaa 00000000 00000014 00000008	FFFFFFF4	0
3	ffffffe0 aaaaaaaa aaaaaaaa aaaaaaaa aaaaaaaa fffffff0 aaaaaaaa 00000000 00000014 00000008	FFFFFFF4	FFFFFFF4
4	ffffffe0 aaaaaaaa aaaaaaaa 00000008 00000000 fffffff0 00000000 00000000 00000014 00000008	FFFFFFE8	FFFFFFF4
5	ffffffe0 aaaaaaaa aaaaaaaa 00000008 00000000 fffffff0 00000000 00000000 00000014 00000008	FFFFFFF4	FFFFFFF4
6	ffffffe0 aaaaaaaa aaaaaaaa 00000008 00000000 fffffff0 00000000 00000000 00000014 00000008	FFFFFFFC	0
7	ffffffe0 aaaaaaaa aaaaaaaa 00000008 00000000 fffffff0 00000000 00000000 00000014 00000008 00000000 e3a0d000	0	0

Example 2: Nested Subroutine Revisited

- The following program calculates the value of f(0x08) where
  - f(x) = g(x+0x18) + 0x05
  - g(x) = x+0x12
- No local variables in both subroutines
- r0 is used to store the output

```
ldr sp, =0
ldr fp, =0
ldr r1, =0x08
stmfd sp!, {r1} //push parameter of f to
stack (Point 1)
bl f
add sp, sp, #4 //deallocate parameter for f
(Point 14)
Here:
b Here

f:
// Prologue
stmfd sp!, {fp, lr} // Step 1 (Point 2)
mov fp, sp // Step 2 (Point 3)
// Step 3: No local variable
stmfd sp!, {r1} // Step 4 (Point 4)

ldr r1, [fp, #8] // Retrieve parameter of f
add r1, r1, #0x18 // generate parameter for g
stmfd sp!, {r1} // push parameter of g to
stack (Point 5)

bl g

add sp, sp, #4 // deallocate parameter for g
(Point 11)
add r0, r0, #0x05 // Calculate output
parameter and store in r0

//Epilogue
ldmfd sp!, {r1} // Step 1 (Point 12)
//Step 2: No local variable
ldmfd sp!, {fp, pc} // Step 3 (Point 13)
```

```
g:
// Prologue
stmfd sp!, {fp, lr} // Step 1 (Point 6)
mov fp, sp // Step 2 (Point 7)
stmfd sp!, {r1} (Point 8)

ldr r1, [fp, #8]
add r0, r1, #0x12 // Calculate output
parameter and store in r0

//Epilogue
ldmfd sp!, {r1} // Step 1 (Point 9)
//Step 2: No local variable
ldmfd sp!, {fp, pc} // Step 3 (Point 10)

.end
```

Pt	Stack	sp	fp
1	ffffffe0 aaaaaaaa aaaaaaaa aaaaaaaa aaaaaaaa fffffff0 aaaaaaaa aaaaaaaa aaaaaaaa 00000008	FFFFFFFC	0
2	ffffffe0 aaaaaaaa aaaaaaaa aaaaaaaa aaaaaaaa fffffff0 aaaaaaaa 00000000 00000014 00000008	FFFFFFF4	0
3	ffffffe0 aaaaaaaa aaaaaaaa aaaaaaaa aaaaaaaa fffffff0 aaaaaaaa 00000000 00000014 00000008	FFFFFFF4	FFFFFFF4
4	ffffffe0 aaaaaaaa aaaaaaaa aaaaaaaa aaaaaaaa fffffff0 00000008 00000000 00000014 00000008	FFFFFFF0	FFFFFFF4
5	ffffffe0 aaaaaaaa aaaaaaaa aaaaaaaa 00000020 fffffff0 00000008 00000000 00000014 00000008	FFFFFFEC	FFFFFFF4
6	ffffffe0 aaaaaaaa fffffff4 00000038 00000020 fffffff0 00000008 00000000 00000014 00000008	FFFFFFE4	FFFFFFF4
7	ffffffe0 aaaaaaaa fffffff4 00000038 00000020 fffffff0 00000008 00000000 00000014 00000008	FFFFFFE4	FFFFFFE4

Pt	Stack	sp	fp
8	ffffffe0 00000020 fffffff4 00000038 00000020 fffffff0 00000008 00000000 00000014 00000008	FFFFFFE0	FFFFFFE4
9	ffffffe0 00000020 fffffff4 00000038 00000020 fffffff0 00000008 00000000 00000014 00000008	FFFFFFE4	FFFFFFE4
10	ffffffe0 00000020 fffffff4 00000038 00000020 fffffff0 00000008 00000000 00000014 00000008	FFFFFFEC	FFFFFFF4
11	ffffffe0 00000020 fffffff4 00000038 00000020 fffffff0 00000008 00000000 00000014 00000008	FFFFFFF0	FFFFFFF4
12	ffffffe0 00000020 fffffff4 00000038 00000020 fffffff0 00000008 00000000 00000014 00000008	FFFFFFF4	FFFFFFF4
13	ffffffe0 00000020 fffffff4 00000038 00000020 fffffff0 00000008 00000000 00000014 00000008	FFFFFFFC	0
14	ffffffe0 00000020 fffffff4 00000038 00000020 fffffff0 00000008 00000000 00000014 00000008 00000000	0	0

Example 3: Pass by reference (Lab 6)

- Suppose you have the following data:

```
.data
.align
x:
.word 15
y:
.word 9
```

- You want to swap the two values.
- You could pass the values 15 and 9 to the stack and swap their positions in the stack, but that is not what you want to achieve. This “pass-by-value” approach does not work.
- You want to pass the *addresses* of 15 and 9 as parameters to the stack *so that the values in the two memory locations get swapped.* This is referred to as “**pass by reference**”.

Swap: Pass by reference

Main Program

```
.global _start
_start:

ldr     r1, =y          // get address of second parameter
stmfd   sp!, {r1}       // push second parameter onto stack
ldr     r1, =x          // get address of first parameter
stmfd   sp!, {r1}       // push first parameter onto stack
bl      swap            // call subroutine
add     sp, sp, #8      // release parameter memory

_stop:
b       _stop
```

Data Storage

```
.data
.align
x:
.word    15
y:
.word    9

.end
```

Swap: Pass by reference

----

swaps location of two values in memory.

Parameters:

  [fp, #8] – address of x

  [fp, #12] – address of y

Local variable

  temp (4 bytes on stack)

Uses:

  r0 - stores address of x

  r1 - stores address of y

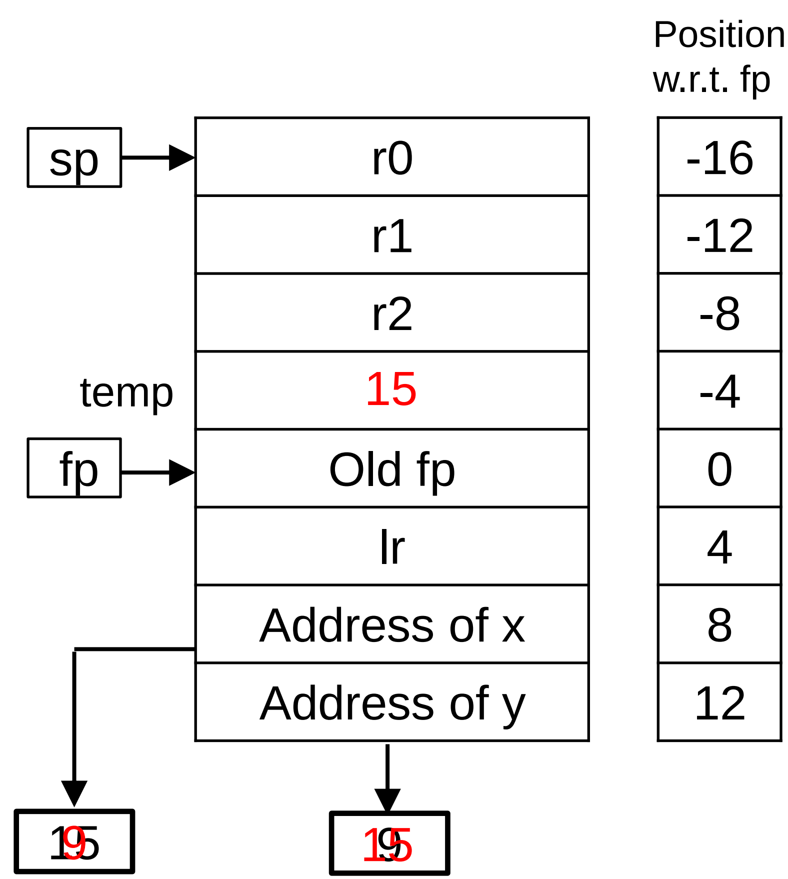
  r2 - registers for swapping operations

```
swap:
// Prologue
stmfd   sp!, {fp, lr}       // push frame pointer
mov     fp, sp          // save current stack top to frame pointer
sub     sp, sp, #4      // set aside space for local variable temp
stmfd   sp!, {r0-r2}    // preserve other registers

ldr     r0, [fp, #8]    // put address of x in r0
ldr     r1, [fp, #12]    // put address of y in r1

ldr     r2, [r0]        // put value of x in r2
str     r2, [fp, #-4]   // copy value of x to temp

ldr     r2, [r1]        // put value of y in r2
str     r2, [r0]        // store value of y in address of x

ldr     r2, [fp, #-4]   // get temp
str     r2, [r1]        // store temp in address of y

//Epilogue
ldmfd   sp!, {r0-r2}    // pop preserved registers
add     sp, sp, #4      // remove local storage
ldmfd   sp!, {fp, pc}       // pop frame pointer

.data
.align
x:
.word 15
y:
.word 9

.end
```

Swap: Pass by reference

```
swap:
// Prologue
stmfd   sp!, {fp, lr}       // push frame pointer
mov     fp, sp          // save current stack top to frame pointer
sub     sp, sp, #4      // set aside space for local variable temp
stmfd   sp!, {r0-r2}    // preserve other registers

ldr     r0, [fp, #8]    // put address of x in r0
ldr     r1, [fp, #12]    // put address of y in r1

ldr     r2, [r0]        // put value of x in r2
str     r2, [fp, #-4]   // copy value of x to temp

ldr     r2, [r1]        // put value of y in r2
str     r2, [r0]        // store value of y in address of x

ldr     r2, [fp, #-4]   // get temp
str     r2, [r1]        // store temp in address of y

//Epilogue
ldmfd   sp!, {r0-r2}    // pop preserved registers
add     sp, sp, #4      // remove local storage
ldmfd   sp!, {fp, pc}       // pop frame pointer

.data
.align
x:
.word 15
y:
.word 9

.end
```

Stack after prologue of swap



MinMax subroutine in Lab 6

Finds the minimum and maximum values in a list.

Equivalent of: void MinMax (*start, *end, *min, *max)

Passes addresses of list, end of list, max, and min as parameters.

Parameters:

- start - start address of list (fp + 4)
- end - end address of list (fp + 8)
- min - address of minimum result (fp + 12)
- max - address of maximum result (fp + 16)

Uses:

- r0 - address of start of list
- r1 - address of end of list
- r2 - minimum value so far
- r3 - maximum value so far
- r4 - address of value to process

```
MinMax:
stmfd   sp!, {fp}         // preserve frame pointer
mov     fp, sp            // Save current stack top to frame pointer
// allocate local storage (none)
stmfd   sp!, {r0-r4}      // preserve other registers

ldr     r0, [fp, #4]      // Get address of start of list
ldr     r2, [r0]          // store first value as minimum
ldr     r3, [r0], #4      // store first value as maximum
ldr     r1, [fp, #8]      // get address of end of list

MinMaxLoop:
cmp     r0, r1            // Compare addresses
beq     _MinMax
ldr     r4, [r0], #4
cmp     r4, r2
movlt   r2, r4
cmp     r4, r3
movgt   r3, r4
b       MinMaxLoop

_MinMax:
// Store results to address parameters
ldr     r0, [fp, #12]
str     r2, [r0]
ldr     r0, [fp, #16]
str     r3, [r0]

ldmfd   sp!, {r0-r4}      // pop preserved registers
// deallocate local storage (none was allocated)
ldmfd   sp!, {fp}         // pop frame pointer
bx      lr                // return from subroutine
```

```
.data   // Data section
.align
Data:
.word    4,5,-9,0,3,0,8,-7,12    // The list of data
_Data:    // End of list address
Min:
.space 4
Max:
.space 4

.end
```

Example 3: Recursion (Lab 7)

- The following C code implements the Eucildean algorithm in computing the Greatest Common Denominator (gcd) of two integers.

```c
int gcd(a, b) {
    if(a == b) {
        return a;
    } else if(a > b) {
        return gcd(b, a - b);
    } else {
        return gcd(b, a);
    }
```

Example 3: Recursion (Lab 7)

```
int gcd(a, b) {
    if(a == b) {
        return a;
    } else if(a > b) {
        return gcd(b, a - b);
    } else {
        return gcd(b, a);
    }
```

Principle:

- Condition 1:
  - If a>b, the problem can be reduced using the following property:
    - gcd (a, b) = gcd (b, a-b)
- Condition 2:
  - If b>a, reverse the order by calling gcd (b,a)
  - Inside this gcd call, condition 1 would be satisfied, thereby reducing the problem.
- Eventually, the two input arguments are equal and gcd (a,a) = a

Numerical Examples

Large-Small	20	15
20-15	5	15
15-5	5	10
10-5	5	5

Large-Small	60	36
60-36	24	36
36-24	24	12
24-12	12	12

- gcd (20, 15)
- gcd (15, 5)
- gcd (5, 10)
- gcd (10, 5)
- gcd (5, 5)

Two arguments are the same.
gcd = 5

- gcd (60, 36)
- gcd (36, 24)
- gcd (24, 12)
- gcd (12, 12)

Two arguments are the same.
gcd = 12

Example 3: Recursion (Lab 7)

<u>Main Program</u>

```
ldr     r1, =numbers    // get address of numbers
ldr     r2, [r1, #4]    // put second number into register (4 bytes past first number)
stmfd   sp!, {r2}       // push second number onto stack
ldr     r2, [r1]        // put first number into register
stmfd   sp!, {r2}       // push first number onto stack
bl      gcd             // call the subroutine
add     sp, sp, #8      // release the parameter memory from the stack
// Result in r0
ldr r1, =result
str     r0, [r1]    // write result back to memory
```

<u>Data Section</u>

```
.data
.align
numbers:
.word 30, 12
result:
.space 4
.end
```

Before calling the subroutine:

- Push second and then first parameter(s) to the stack

After returning from the subroutine

- Deallocate parameter(s) from the stack

gcd subroutine

Finds the Greatest Common
Denominator of two integers (recursive)

Uses simple recursive algorithm:

```
int gcd(a, b) {
    if(a == b) {
        return a;
    } else if(a > b) {
        return gcd(b, a - b);
    } else {
        return gcd(b, a);
    }
```

```
gcd:
stmfd   sp!, {fp, lr}        // preserve frame pointer
mov     fp, sp              // save current stack top to frame pointer
// No local variables
stmfd   sp!, {r1}

// Copy parameters into registers
ldr     r0, [fp, #8]      // get a
ldr     r1, [fp, #12]      // get b

cmp     r0, r1
beq     _gcd                // a is the gcd: return in r0
subgt   r0, r0, r1          // a > b: gcd(b, a - b)
                            // else:  gcd(b, a)
stmfd   sp!, {r0}           // 2nd parameter - push a onto the stack
stmfd   sp!, {r1}           // 1st parameter - push b onto the stack
bl      gcd                 // recursive call to subroutine
add     sp, sp, #8          // release parameter memory from stack

_gcd:
ldmfd   sp!, {r1}    // pop preserved registers
// No local variables
ldmfd   sp!, {fp, pc}        // pop frame pointer
```

Parameters:
- a - value of first number
- b - value of second number

Returns:
- r0 - GCD of a and b

Uses:
- r0 - value of a - return value
- r1 - value of b

Skeleton of the subroutine

```
gcd:
stmfd    sp!, {fp, lr}        // preserve frame pointer
mov      fp, sp            // save current stack top to frame pointer
// No local variables
stmfd    sp!, {r1}

// Copy parameters into registers
ldr      r0, [fp, #8]      // get a
ldr      r1, [fp, #12]      // get b

cmp      r0, r1
beq      _gcd              // a is the gcd: return in r0
subgt    r0, r0, r1        // a > b: gcd(b, a - b)
                           // else:  gcd(b, a)
stmfd    sp!, {r0}         // 2nd parameter - push a onto the stack
stmfd    sp!, {r1}         // 1st parameter - push b onto the stack
bl       gcd               // recursive call to subroutine
add      sp, sp, #8        // release parameter memory from stack

_gcd:
ldmfd    sp!, {r1}     // pop preserved registers
// No local variables
ldmfd    sp!, {fp, pc}        // pop frame pointer
```

Standard Prologue

Just like before and after calling a subroutine in the main program, we need to:

Push parameters to stack before calling subroutine

Deallocate parameters when the subroutine finishes

Standard Epilogue

Example: gcd (18,12)

Sequence of calls:

1. gcd (18, 12) from main program
2. gcd (12, 6)
3. gcd (6, 6) → return 6

```
int gcd(a, b) {
    if(a == b) {
        return a;
    } else if(a > b) {
        return gcd(b, a - b);
    } else {
        return gcd(b, a);
    }
```

Example: gcd (18,12)

1. Call gcd (18, 12) from main program

State of the stack after the prologue of gcd:

sp →	ffffffec	00001068	r1
fp →	fffffff0	00000000	Old fp
	fffffff4	0000101c	lr
	fffffff8	00000012	Parameters
	fffffffc	0000000c	

Example: gcd (18,12)

2. Call gcd (12, 6)

State of the stack after the prologue of gcd:

sp	ffffffd8	0000000c	r1
fp	ffffffdc	fffffff0	Old fp
	ffffffe0	00001054	lr
	ffffffe4	0000000c	Parameters
	ffffffe8	00000006	
	ffffffec	00001068	
	fffffff0	00000000	
	fffffff4	0000101c	
	fffffff8	00000012	
	fffffffc	0000000c	

Example: gcd (18,12)

3. Call gcd (6, 6)

State of the stack after the prologue of gcd:

sp	ffffffc4	00000006	r1
fp	ffffffc8	ffffffdc	Old fp
	ffffffcc	00001054	lr
	ffffffd0	00000006	
	ffffffd4	00000006	Parameters
	ffffffd8	0000000c	
	ffffffdc	fffffff0	
	ffffffe0	00001054	
	ffffffe4	0000000c	
	ffffffe8	00000006	
	ffffffec	00001068	
	fffffff0	00000000	
	fffffff4	0000101c	
	fffffff8	00000012	
	fffffffc	0000000c	