

ARM-7 Assembly: Functions

CSE 2312

Computer Organization and Assembly Language Programming

Vassilis Athitsos

University of Texas at Arlington

Making a Function

- Why are functions useful in assembly?

Making a Function

- Why are functions useful in assembly?
- For the same reasons they are useful in any programming language:
 - Modularity, making code easy to design, write, read, debug.
 - Reusability.
- What functionality from the previous programs would be a good candidate to make a function of?

Making a Function

- Why are functions useful in assembly?
- For the same reasons they are useful in any programming language:
 - Modularity, making code easy to design, write, read, debug.
 - Reusability.
- What functionality from the previous programs would be a good candidate to make a function of?
 - Printing a single hexadecimal digit.
 - Printing an entire 32-bit number in hexadecimal.

Making a Function

- Functions are easy to define and call in languages like C and Java.
- In assembly, calling a function requires several steps.
- This reflects that the CPU can do only a limited amount of work in a single step.
- Note that, to correctly do a function call, both the caller and the called function must do the right steps.

Caller Steps

- **Step 1: Put arguments in the right place.**
- Specific machines use specific conventions.
- Figure 5-4, on textbook page 355, specifies ARM-7 conventions:
 - "R0-R3 hold parameters to the procedure being called".
- So:
 - Argument 1 (if any) goes to r0.
 - Argument 2 (if any) goes to r1.
 - Argument 3 (if any) goes to r2.
 - Argument 4 (if any) goes to r3.
- If there are more arguments, they have to be placed in memory. We will worry about this case only if we encounter it.

Caller Steps

- **Step 2: branch to the first instruction of the function.**
 - Here, we typically use the bl instruction, not the b instruction.
 - The bl instruction, before branching, saves to register lr (the link register, aka r14) the return address.
 - The **return address** is the address of the instruction that should be executed when the function is done.
- **Step 3: after the function has returned, recover the return value, and use it.**
 - We will follow the convention that the return value goes to r0.
 - If there is a second return value, it goes to r1.

Called Function Steps

- **Step 1: Do the preamble:**
 - Allocate memory on the stack (more details in a bit).
 - Save to memory the return address. Why?
 - Save to memory all registers (except possibly for r0) that the function modifies. Why?
- **Step 2: Do the main body of the function.**
 - Assume arguments are in r0, r1, r2, r3.
 - This is where the actual work is done.
- **Step 3: Do the wrap-up:**
 - Store the return value (if any) on r0, and second return value (if any) on r1.
 - Retrieve from memory the return address. Why?
 - Retrieve from memory, and restore to registers, the original values of all registers that the function modified (except possibly for r0). Why?
 - Deallocate memory on the stack.
 - Branch to the return address.

Placing Register Values in Memory

- Why do we need to save register values in memory at the beginning of the function?
- Why do we need to restore the original register values from memory at the end of the function?

Placing Register Values in Memory

- Why do we need to save register values in memory at the beginning of the function?
- Why do we need to restore the original register values from memory at the end of the function?
- Suppose function A gets calls from functions B, C, D, E, ...
- Function A has no idea what function it got called from.
- Therefore, function A has no idea what registers the caller function was using.
- By saving register values at the beginning, and restoring them at the end, function A makes sure that, when it returns, the caller function finds all registers unchanged.
- This makes life more simple for the caller function, it doesn't need to worry about whether any registers got changed.

Placing Register Values in Memory

- In summary: the called function must:
 - Save register values at the beginning.
 - Restore register values at the end.
- In theory, we could have used a different convention (but we will not use it):
 - The called function does not worry about saving and restoring register values.
 - The caller:
 - Saves whatever register values it needs before making the function call.
 - Restores those register values after the function call has returned.
- Both conventions are okay, we just need to choose one and stick with it.

Placing Register Values in Memory

- What about r0?
- Why don't we restore the original value of r0 at the end of the function?

Placing Register Values in Memory

- What about r0?
- Why don't we restore the original value of r0 at the end of the function?
- Because r0 is supposed to hold the return value.
- This is the one register that the caller expects to find changed at the end of the function.
- We will follow the convention that, if the function does not return anything (returns void) then we will be restoring the original value of r0 as well.
- If the function returns a second value on r1, then obviously r1 should also not be restored.

Placing Register Values in Memory

- What about `lr` (the link register)?
- Why do we need to save it to memory at the beginning, and restore it from memory at the end of the function?

Placing Register Values in Memory

- What about `lr` (the link register)?
- Why do we need to save it to memory at the beginning, and restore it from memory at the end of the function?
- Every time our function calls other functions, `lr` changes.
- By restoring it at the end of the function, we make sure we get the right return address.
- In principle, if our function does not make any other function calls, we do not really need to save `lr` to memory.
- In practice, personally I will follow the convention to always save `lr`, so as to avoid possible bugs.
- You will probably get a bug at some point, where:
 - You forget to restore `lr` at the end of the function.
 - Your function branches to a weird place at the end, instead of returning to the caller.

Saving to Memory

- When does a function need to save information to memory?
 - At the beginning, to save the original values of the registers.
 - At any later time, if there are not enough registers to store useful intermediate values.
- A very important question:
 - How does the function know what memory to use?
 - How can the function avoid messing up memory already used by other functions?
- Answer: the stack, and the stack pointer.

The Stack Pointer

- The stack pointer points to the beginning of the memory space used by a specific function.
- When we write an assembly function, at the end, we look at all the memory that we needed.
- Suppose that we needed X bytes.
- Then, at the beginning (first line) of the function, we put this line:

```
sub sp, sp, #X
```

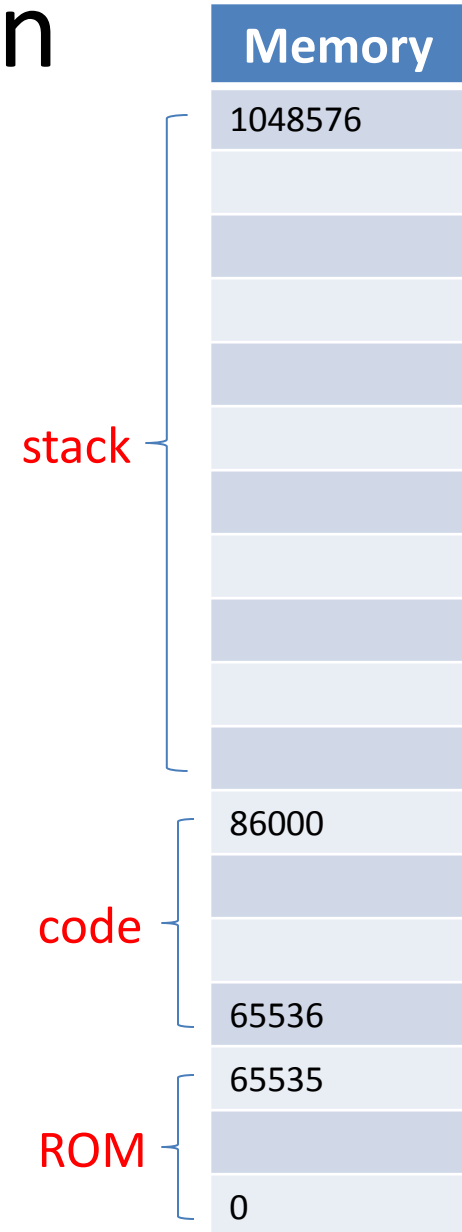
- At the end of the function (right before returning), we put this line:

```
add sp, sp, #X
```

- This way, we mark that the function uses memory addresses from [sp] to [sp+X-1].
- When the function is done, it restores the original value of sp.
- This way, when execution goes back to the caller, sp has the appropriate value for the caller.

Memory Organization

- In the simulated ARM machine we are using, memory addresses from 0 to 0xffff are read-only memory.
 - In decimal, these are addresses from 0 to 65535.
- Instructions will be saved at addresses 0x10000 and up.
 - In decimal, this is address 65536.
- Typically instructions will take no more than 20K.
 - Therefore, instructions go up to address 86000.
- At the beginning of the program (NOT the beginning of each function, just the beginning of the entire program) we will hardcode the stack pointer to hexadecimal address 0x100000.
- In decimal, this address is about 1.05 million.
- This leaves about (1.05 million - 86 thousand) bytes, i.e., roughly about 920 thousand bytes, for use by functions.
- By the term "stack" we simply mean these bytes, that are available for use by functions.



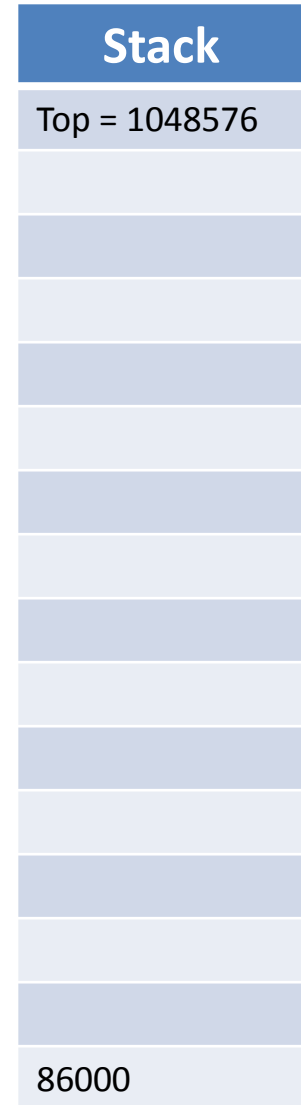
Stack Pointer Example

- At the beginning of the program, we do:

```
mov sp, #0x100000
```

- This points the stack pointer to the top of the stack.

start of program: sp →



Stack Pointer Example

- At the beginning of the program, we do:

```
mov sp, #0x100000
```

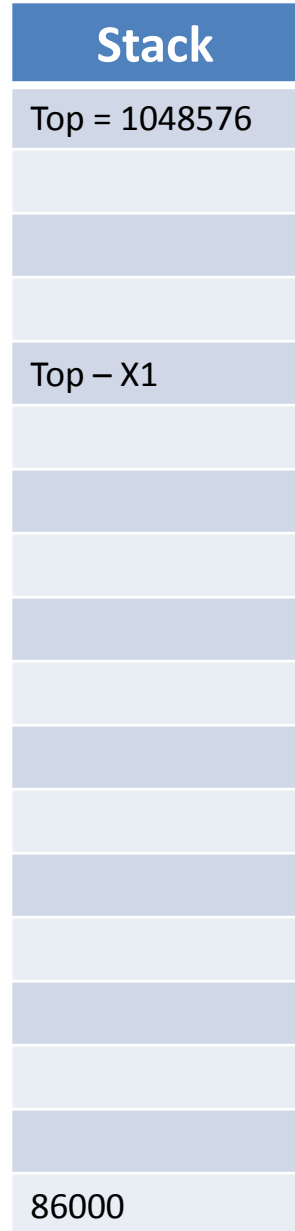
- This points the stack pointer to the top of the stack.
- Then, the initial function (called `_start` in our examples) immediately subtracts from the `sp` the space that it needs for its own use. Suppose it needs `X1` bytes.

```
sub sp, sp, #X1
```

start of program: `sp` →

memory for `_start`

`_start` function: `sp` →



Stack Pointer Example

- At the beginning of the program, we do:

```
mov sp, #0x100000
```

- This points the stack pointer to the top of the stack.
- Then, the initial function (called `_start` in our examples) immediately subtracts from the `sp` the space that it needs for its own use. Suppose it needs `X1` bytes.

```
sub sp, sp, #X1
```

- Then, suppose `_start` calls function `foo`. Function `foo` will set the `sp` to an even lower value. Suppose `foo` needs `X2` bytes.

```
sub sp, sp, #X2
```

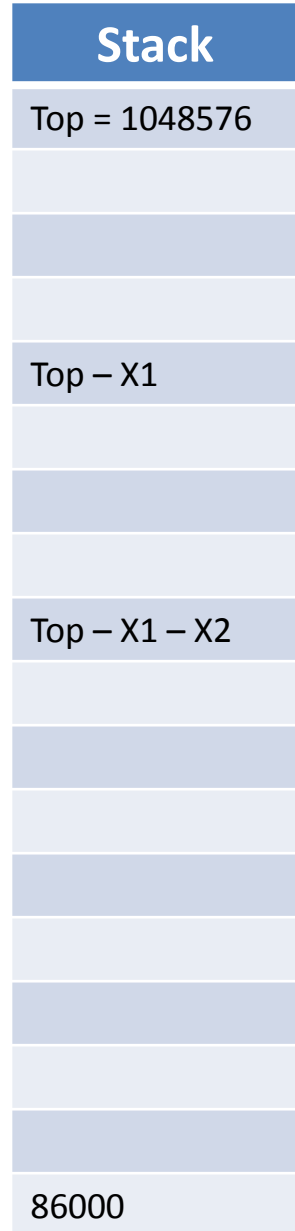
start of program: `sp` →

memory for `_start`

`_start` function: `sp` →

memory for `foo`

`foo`: `sp` →



Stack Pointer Example

- Then, suppose function foo calls function qq. Function qq will set the sp to an even lower value. Suppose qq needs X3 bytes.

```
sub sp, sp, #X3
```

start of program: sp →

memory for `_start`

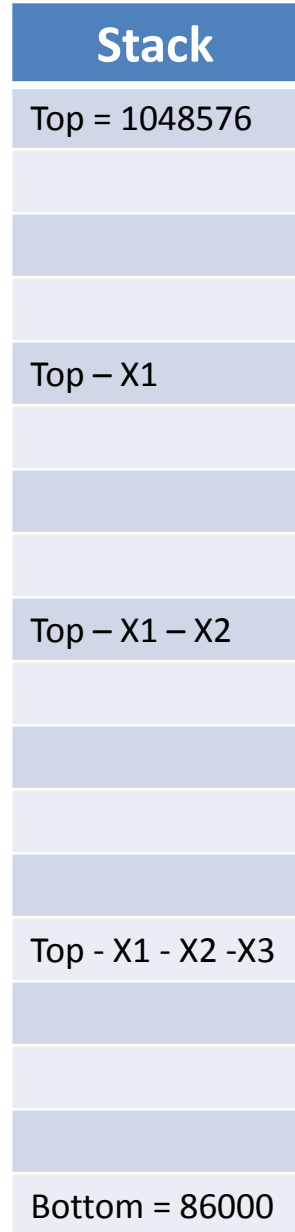
`_start` function: sp →

memory for `foo`

`foo`: sp →

memory for `qq`

`qq`: sp →



Stack Pointer Example

- Then, suppose function foo calls function qq. Function qq will set the sp to an even lower value. Suppose qq needs X3 bytes.

```
sub sp, sp, #X3
```

- Then, function qq is getting ready to return, and restores the sp to the value it was set to by the caller function (function foo).

```
add sp, sp, #X3
```

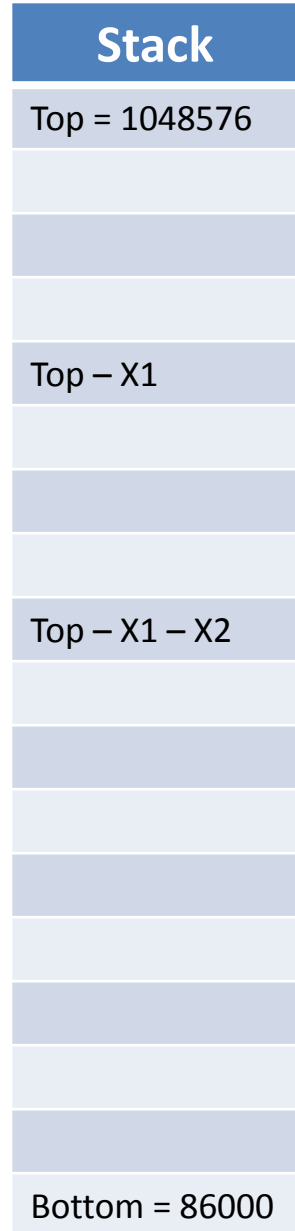
start of program: sp →

memory for `_start`

`_start` function: sp →

memory for `foo`

`foo`: sp →



Stack Pointer Example

- Then, suppose function foo calls function qq. Function qq will set the sp to an even lower value. Suppose qq needs X3 bytes.

```
sub sp, sp, #X3
```

- Then, function qq is getting ready to return, and restores the sp to the value it was set to by the caller function (function foo).

```
add sp, sp, #X3
```

- Then, function foo calls function rrr. Suppose function rrr needs X4 bytes:

```
sub sp, sp, #X4
```

start of program: sp →

memory for _start

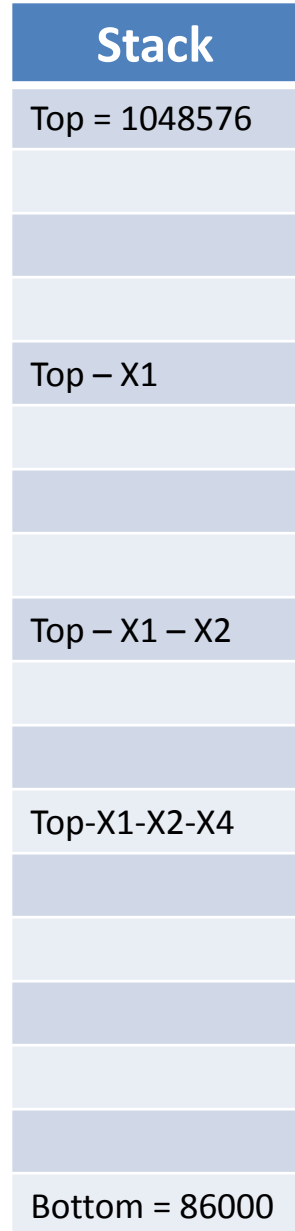
_start function: sp →

memory for foo

foo: sp →

memory for rrr

rrr: sp →



Stack Pointer Example

- Then, function `foo` is getting ready to return, and restores the `sp` to the value it was set to by the caller function (function `foo`).

```
add sp, sp, #X4
```

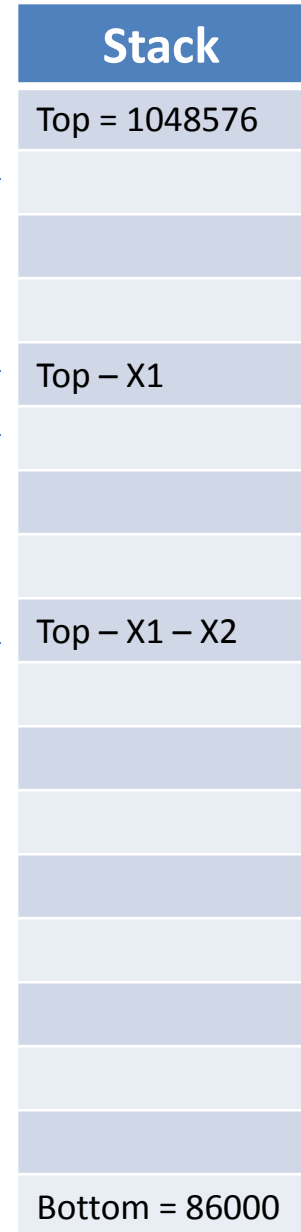
start of program: `sp` →

memory for `_start`

`_start` function: `sp` →

memory for `foo`

`foo`: `sp` →



Stack Pointer Example

start of program: sp →

- Then, function rrr is getting ready to return, and restores the sp to the value it was set to by the caller function (function foo).

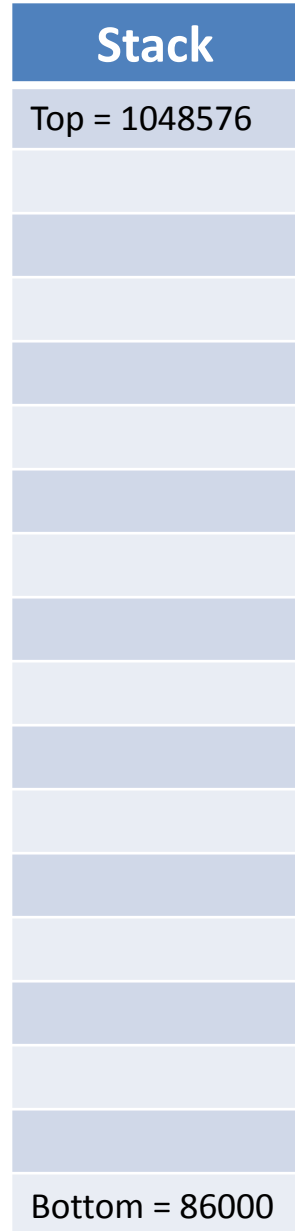
```
add sp, sp, #X4
```

- Function foo is now also getting ready to return, and restores the sp to the value it was said to by the caller function (function _start).

```
add sp, sp, #X2
```

- Finally, function _start is wrapping up, and restores the sp to point to the top of the stack:

```
add sp, sp, #X1.
```



Summary of Caller and Callee Steps

- Caller steps:
 - Step 1: Put arguments in the registers r0, r1, r2, r3.
 - Step 2: Branch to the function, using the bl instruction.
 - Step 3: After the function has returned, recover the return value (if any), and use it.
- Callee (called function) steps:
 - Step 1 (preamble): Allocate memory on the stack, and save register lr, and other registers that the function modifies, to the stack.
 - Step 2: Do the main body of the function.
 - Step 3 (wrap-up):
 - Store the return value (if any) on r0, second return value (if any) on r1.
 - Restore, from the stack, the original values of all registers that the function modified, as well as the value of register lr.
 - Deallocate memory on the stack (increment sp).
 - Branch to the return address using instruction bx.

A First Function Example

- In this program, we define and use a function `print_digit`.
- This function:
 - Takes a single argument.
 - Assumes that the argument is a number between 0 and 15.
 - Prints that number in hexadecimal.
- The program prints numbers 0 to 15 in hex.

```
.globl _start
```

```
_start:
```

```
    mov sp, #0x100000  
    @initialize sp at start
```

```
@ program main body
```

```
    mov r5, #0x0
```

```
my_loop:
```

```
    cmp r5, #0xf  
    bgt program_exit
```

```
    mov r0, r5  
    bl print_digit
```

```
    add r5, r5, #1  
    b my_loop
```

```
program_exit:
```

```
    ldr r4,=0x101f1000
```

```
    mov r1, #'r'
```

```
    str r1, [r4]
```

```
    mov r1, #'n'
```

```
    str r1, [r4]
```

```
    mov r1, #'E'
```

```
    str r1, [r4]
```

```
    mov r1, #'N'
```

```
    str r1, [r4]
```

```
    mov r1, #'D'
```

```
    str r1, [r4]
```

```
the_end:
```

```
    b the_end
```

print_digit:

@ print_digit preamble

sub sp, sp, #16

str lr, [sp, #0]

str r0, [sp, #4]

str r4, [sp, #8]

str r5, [sp, #12]

@ print_digit main body

ldr r4,=0x101f1000

@ ASCII codes stored

@ at [r4] get printed

cmp r0, #10

addlt r5, r0, #48

addge r5, r0, #55

str r5, [r4]

@ print_digit wrap-up

ldr lr, [sp, #0]

ldr r0, [sp, #4]

ldr r4, [sp, #8]

ldr r5, [sp, #12]

add sp, sp, #16

bx lr

Things to Note

- Structure of the source code file:
 - Part 1: definition of main.
 - Part 2: definition of all functions (the order doesn't matter).

Things to Note

- The main program uses `r5` as the loop variable.
 - The values of `r5` range from 0 to 15.
 - For each of those values, `print_digit` is called.
- Function `print_digit` also uses `r5`.
- Why does that not mess up the value of `r5` in the main program?

Things to Note

- The main program uses r5 as the loop variable.
 - The values of r5 range from 0 to 15.
 - For each of those values, print_digit is called.
- Function print_digit also uses r5.
- Why does that not mess up the value of r5 in the main program?
 - Because **print_digit leaves the values of all registers as it found them.**
 - **Every function should do that.**
 - It is the job of the function preamble and the function wrap-up to do that.

Things to Note

- One of the registers we save and restore is `lr`.
- Strictly speaking, it is not necessary.
 - Function `print_digit` does not modify `lr` at any point.
- If we wanted to make performance as fast as possible, we would not save and restore `lr`.
- In practice, it is a good habit, so as to avoid bugs.
- It is recommended that you guys always save and restore `lr` in any function you write.

How to Write a Function

- You can follow two approaches.
- Approach 1:
 - First, write a preamble and wrapup that save and restore all registers you may possibly need.
 - Second, write the main body, test, and debug the function.
 - Third, rewrite the preamble and wrapup, to avoid saving and restoring registers that you did not end up using.

How to Write a Function

- Approach 2:
 - First, write the function main body.
 - Second, see what registers you are using in the function main body.
 - Third, write the preamble and wrapup, to save and restore all registers you use.
- Disadvantage of second approach:
 - As you debug and make changes, you may use more or fewer registers.
 - You have to keep modifying the preamble and wrapup.
 - Value to subtract from sp.
 - Memory locations used for the registers.

A Second Function Example

- In this program, we define and use a function `print_number`.
- This function:
 - Takes a single argument, that is a 32-bit number.
 - Prints that number in hexadecimal.
- The program prints numbers `0xffffffff` to `0x1000010` in hex.

.globl _start

_start:

mov sp, #0x100000
@initialize sp at start

@ program main body
ldr r4,=0x101f1000

mov r5, #0x0f
lsl r5, r5, #8
add r5, r5, #0xff
lsl r5, r5, #8
add r5, r5, #0xff
lsl r5, r5, #8
add r5, r5, #0xfd

mov r6, #19

my_loop:

cmp r6, #0
blt my_exit

mov r0, r5
bl print_number
add r5, r5, #1
sub r6, r6, #1
b my_loop

my_exit:

@ program wrap-up
b my_exit

print_number:

@ print_number preamble

sub sp, sp, #24

str lr, [sp, #0]

str r0, [sp, #4]

str r4, [sp, #8]

str r5, [sp, #12]

str r6, [sp, #16]

str r7, [sp, #20]

@ print_number main body

ldr r4,=0x101f1000

@ ASCII codes stored

@ at [r4] get printed

mov r5, #28

mov r6, r0

print_number_loop:

cmp r5, #0

blt print_number_exit

lsr r7, r6, r5

and r7, r7, #0x0000000f

mov r0, r7

bl print_digit

sub r5, r5, #4

b print_number_loop

print_number_exit:

@ print newline

mov r5, #13

str r5, [r4]

mov r5, #10

str r5, [r4]

@ print_number wrap-up

ldr lr, [sp, #0]

ldr r0, [sp, #4]

ldr r4, [sp, #8]

ldr r5, [sp, #12]

ldr r6, [sp, #16]

ldr r7, [sp, #20]

add sp, sp, #24

bx lr

Things to Note

- Function `print_number` uses `r5`.
- Function `print_digit` also uses `r5`.
- Again, this is no problem because each function leaves the values of the registers as it found them.

Things to Note

- What would happen if `print_number` did not save and restore the value of `lr` in its preamble and wrapup?

Things to Note

- What would happen if `print_number` did not save and restore the value of `lr` in its preamble and wrapup?
- Register `lr` gets modified when, from `print_number`, we call `print_digit`.
- At that time, `lr` is set to point to what instruction?
 - The instruction `"sub r5, r5, #4"` that is in the `print_number` function, right after the call to `print_digit`.
 - If, at the end of `print_number` we do not restore `lr`, then instruction `"bx lr"` will go right back to the `"sub r5, r5, #4"` instruction, and the program gets into an infinite loop.

Recursive Function Example: Factorial

- How do we write function factorial in C, as a recursive function?
- How do we write function factorial in assembly?

Recursive Function Example: Factorial

- How do we write function factorial in C, as a recursive function?
- How do we write function factorial in assembly?

```
int factorial(int N)
{
    if (N== 0) return 0;
    return N* factorial(N -1);
}
```

Recursive Function Example: Factorial

- How do we write function factorial in C, as a recursive function?

```
int factorial(int N)
{
    if (N== 0) return 0;
    return N* factorial(N -1);
}
```

- How do we write function factorial in assembly?

```
@ factorial main body
mov r4, r0
cmp r4, #0
moveq r0, #1
beq factorial_exit

sub r0, r4, #1
bl factorial
mov r5, r0
mul r0, r5, r4
```

Recursive Function Example: Factorial

```
@ factorial preamble  
???
```

```
@ factorial main body
```

```
mov r4, r0
```

```
cmp r4, #0
```

```
moveq r0, #1
```

```
beq factorial_exit
```

```
sub r0, r4, #1
```

```
bl factorial
```

```
mov r5, r0
```

```
mul r0, r5, r4
```

```
@ factorial wrap-up  
???
```


Recursive Function Example: Factorial

```
@ factorial preamble
```

```
sub sp, sp, #12
```

```
str lr, [sp, #0]
```

```
str r4, [sp, #4]
```

```
str r5, [sp, #8]
```

```
@ factorial main body
```

```
mov r4, r0
```

```
cmp r4, #0
```

```
moveq r0, #1
```

```
beq factorial_exit
```

```
sub r0, r4, #1
```

```
bl factorial
```

```
mov r5, r0
```

```
mul r0, r5, r4
```

```
@ factorial wrap-up
```

```
ldr lr, [sp, #0]
```

```
ldr r4, [sp, #4]
```

```
ldr r5, [sp, #8]
```

```
add sp, sp, #12
```

```
bx lr
```