

# Writing Functions

Suresh Purini, IIIT-H

February 21, 2012

C-like programming languages provides the following abstractions for concise expression on algorithmic logic without worrying about the underlying architectural features.

1. Data abstractions
2. Data processing abstractions
3. Control abstractions
4. Functions

Assembly language programmers and compiler writers should know how to realize the language abstractions using the target machine's ISA. In this handout, we learn how to write functions.

## 0.1 Assemblers, Compilers, Linkers, and Loaders

When you write your ARM program using a *vi* editor and save it, it gets saved on the hard disk of your computer as a series of ASCII characters which you can read by using an appropriate tool. The program in its current form is human readable but the machine cannot understand it. When we compile the program using a compiler or an assembler, the source program gets translated into a object file. The processor do not understand the object file either. All the processor does is *fetch* the instruction whose main memory address is present in the program counter, *decode* it and carry out its it execution. The processor mindlessly keeps going through this *fetch*, *decode* and *execute* cycle. It is the job of the programmer and other system tools like compilers, linkers, loaders, operating system etc. to set up things in such a way that the mindless execution of instructions by the processor ultimately make sense.

Now coming back to the object files, the object file contains the program instructions and whole lot of other information which is necessary for the *linker* and *loader* programs to prepare an in-memory copy of the program suitable for its execution. Modern Linux systems use an ELF format for the object files and earlier UNIX systems use COFF format.

**Question 1.** *What is the difference between an object file and an executable file?*

**Question 2.** *What is the difference between a loader and a linker?*

**Question 3.** *Read the man page of the `objdump` program. Using `objdump`, disassemble an `a.out` file you have obtained by compiling a C-program.*

**Question 4.** *Suppose you have written a program which spans across two files `main.c` and `fun.c`. From the function `main()` in `main.c` you call the function `fun()` in `fun.c`. Now the problem is the compiler do not know the address of the function `fun()` while compiling `main.c`. Find out how this problem gets resolved.*

**Reading Exercise 1.** Read the Chapter on Linking from the book *Computer Systems: A Programmer's Perspective* by Bryant and O'Hallaron. This particular chapter is posted on the course website.

## 0.2 Program Layout

An assembly language is logically divided into sections. We can specify what goes into each section by using appropriate assembler directives. Following is a sample program which contains *text* and *data* sections.

```
/* '@' symbol is used for single line comments */

.arm                               @ For Thumb mode use .thumb directive

.data

num1:    .word 0xaaaa0000
num2:    .word 0x0000bbbb
sum:     .word

.text
.global main    @ 'main' function is mandatory.

main:
    ldr r1, =num1    @ r1 = &num1
    ldr r2, =num2    @ r2 = &num2
    ldr r3, [r1]      @ r3 = *r1
    ldr r4, [r2]      @ r4 = *r2
    add r5, r3, r4    @ r5 = r3 + r4
    ldr r6, =sum      @ r6 = &sum
    str r5, [r6]      @ *r6 = r5

@int sys_exit(int status)
    mov r0, #0        @ Return code
    mov r7, #1        @ sys_exit
    svc 0x00000000
```

Assembly language programmers mostly use text and data sections only. In the text section we write the program code and in the data section we allocate memory to variables.

**Question 5.** What is the purpose of `.lcomm` and `.comm` assembler directives?

**Question 6.** What is `bss` section? What is its purpose? Write a C program where you declare a global array which is uninitialized. Using `objdump` identify the section where the global array is allocated memory. Now modify the same C program and initialize the global array to some constants. Again using `objdump` identify the section in which the global array is allocated memory.

The assembler takes the input assembly program and generates an object file which contains the logical program layout along with other important details. The loader program takes the *on-disk* logical program layout specification from the object file and creates an in-memory layout for the

program. Figure 1 shows the in-memory layout of a process<sup>1</sup> running on a Linux machine over an IA-32 processor. The process layout is a convention which is decided by an operating system

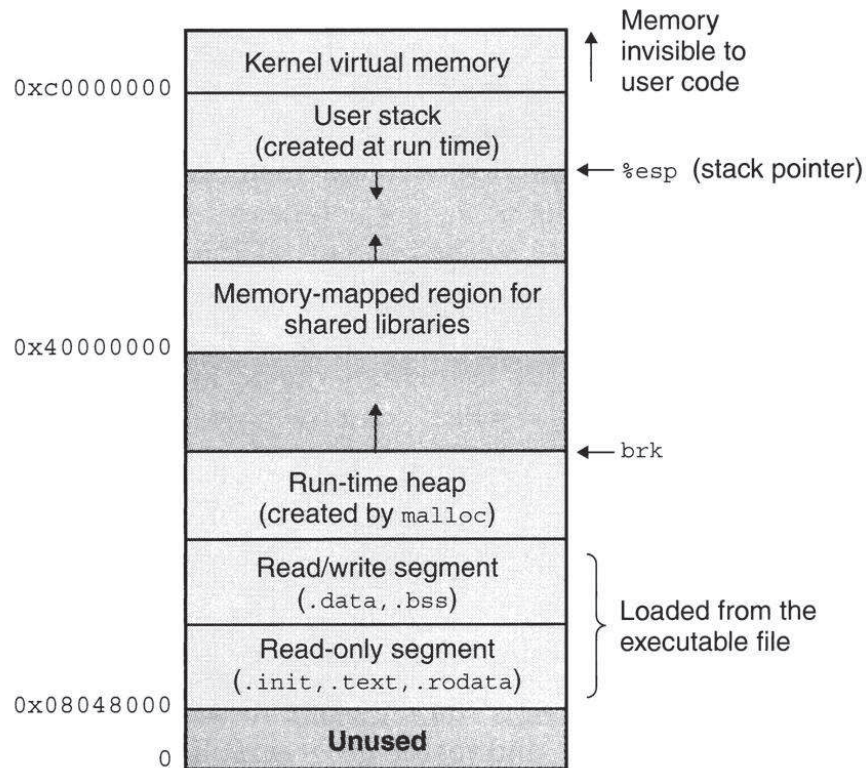


Figure 1: Process layout on Linux over IA-32 processors. A process is a program under execution.

running on a particular processor. The following are typical *segments* in a process layout.

1. A *read-only* segment where program code and other read-only data like constants will be kept.
2. A *read-write* segment where memory for the elements in the data and the bss section will be allocated.
3. A *stack* segment which is used to realize program functions.
4. A *heap* segment from where memory will be allocated to the malloc calls.
5. A *shared library* memory segment.
6. A *kernel* memory segment.

The stack and the heap grow in opposite directions. The stack pointer register points to the top of the stack. In many architectures we have a *descending* stack which means that the stack grows from higher memory address to lower memory address. The other convention to be decided is whether the stack pointer register points to the top full entry of the stack or it points to the empty entry immediately above the full entry. If the stack pointer points to the full entry then it is called as a *full stack*. In the process memory layout for Linux on an ARM processor, the stack follows the *full descending* convention. Figure 2 shows various possible stack conventions.

<sup>1</sup>A *process* is a program under execution.

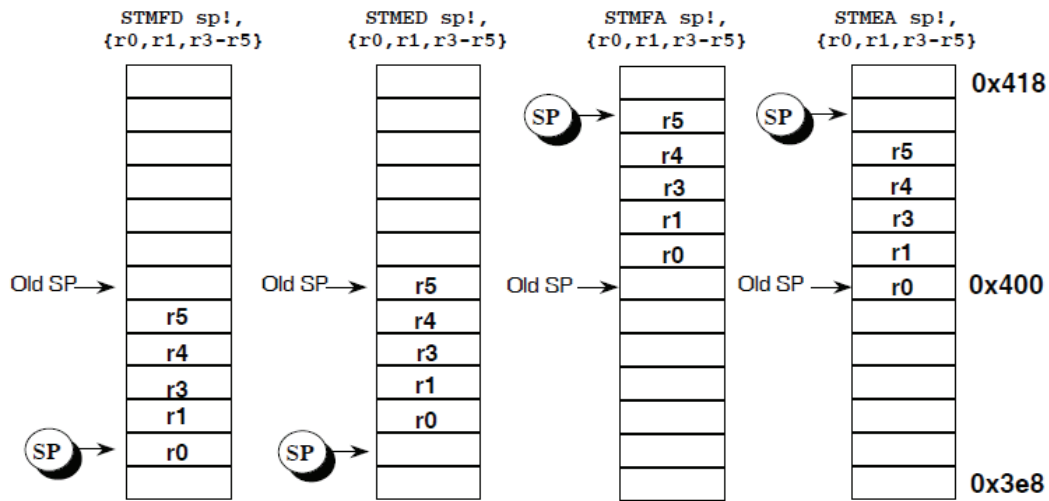


Figure 2: Various ways of organizing the stack

**Question 7.** In the class, I did not discuss STMFD, STMED, STMFA and STMEA instructions. Understand what these store instructions mean and the analogous `ldm` instruction variants. Check whether these are really ARM instructions or assembler pseudo-ops. You want to write a toy program using these instructions and check what assembler does to them by disassembling the object code.

### 0.3 Functions

We have to address the following questions to realize the *function abstraction* in assembly code.

1. How to transfer control to the *callee* function from the *caller* function and return back to the instruction after the *calling point*?
2. How to pass *parameters*?
3. How to pass *return values*?
4. Register preserving mechanisms.

We can handle control transfer from caller to callee and back using the BL (Branch and Link) instruction of the ARM ISA. Refer Figure 4 for the BL instruction format. When the processor



Figure 3: Branch and Link Instruction Format

executes the BL instruction it transfers control to a target instruction (which should be the first instruction of the callee) based on the offset specified in the instruction and it also stores the address

of the instruction immediately after the BL instruction in the Link Register (lr or r14 register). The callee function can then use the return address in the lr register to transfer control back to the caller. Check the following template code.

```
main:
    .
    .
    .
    BL sum          @ pc = AddressOf(sum).  lr = AddressOf(BL)+4
    ADD r0, r1, r2
    .
    .
    .
sum:          @ Make sure that you preserve the contents of lr
    .          @ Otherwise yo do not know the return address in the caller.
    .
    .
    MOV pc, lr      @ Transfer control back to the caller.
```

We can pass parameters and return values through registers. Usually if there are few parameters then they are passed through registers. If there are many we can pass them through stack also. Mostly return values are passed back to the caller using registers alone. When it comes to register preserving mechanisms, there are three approaches.

1. Caller Savee: In this approach the caller function stores all the registers which it would like to be preserved across a function call on a stack and restores them back.

```
main:
    .
    .
    STMFD sp!, {r2, r5, r8} @ Push the contents of r2, r5 and r8 on the stack
    BL sum
    LDMFD sp!, {r2, r5, r8} @ Pop the original values back into registers.
    ADD r0, r1, r2
    .
    .
    .
sum:          @ Use all the register as you wish.
    .          @ The onus of preserving lr is still on you.
    .
    .
    MOV pc, lr      @ Transfer control back to the caller.
```

2. Callee Savee: In this approach the callee function stores all the registers which it would like to use at the beginning of the function and restores them back at the end.

```
main:
    .
    .
```

```

    .
    BL sum
    ADD r0, r1, r2
    .
    .
    .
sum:
    STMFD sp!, {r2, r5, r8, lr} @ Push the contents of the registers on to the stack.
    .                               @ Now use all the register as you wish.
    .
    .
    LDMFD sp!, {r2, r5, r8, pc}    @ Transfer control back to the caller.

```

3. Caller-Callee Savee: In this mechanism the register set is split into partitions. Caller takes the responsibility of preserving the registers in one partition and Callee takes the responsibility of preserving the registers in another partition.

**Question 8.** *Is there any advantage for the hybrid Caller-Callee register preserving mechanism when compared with either the pure Caller or Callee register preserving mechanisms?*

Consider the ARM assembly code for computing the factorial of a number in the next page. Answer the following questions.

**Question 9.** 1. *We know that the function calling sequence is*

$$\text{main}() \rightarrow \text{fact}(3) \rightarrow \text{fact}(2) \rightarrow \text{fact}(1).$$

*Clearly draw the contents of the stack at each of the points of the recursive function calls. For example, what are the contents of the stack when the function **fact** is activated or called with 2 parameter. Verify your answers using gdb by using commands like **backtrace** and **frame**.*

2. *How does the program code changes if we follow a caller savee mechanism?*

```

.arm
.data

N:      .byte 0x3
factN:  .word

.text
.global main      @ 'main' function is mandatory.

main:
    ldr r1, =N      @ r1 = &N
    ldrb r0, [r1]
    bl fact          @ pass parameter through r0. Return value comes through r0
    ldr r1, =factN   @ r1 = &factN
    str r0, [r1]

.Lexit:
@int sys_exit(int status)
    mov r0, #0      @ Return code
    mov r7, #1      @ sys_exit
    svc 0x00000000

fact:
    stmdb sp!, {r1, r2, lr} @ We can also use STMFD sp!, {r1, r2, lr}
    cmp r0, #1
    ldmeqia sp!, {r1, r2, pc} @We can also use LDMFD sp!, {r1, r2, pc}
    mov r1, r0      @ store n in r1
    sub r0, r0, #1
    bl fact
    mul r2, r0, r1   @ why not use mul r0, r0, r1
    mov r0, r2
    ldmia sp!, {r1, r2, pc} @ We can also use LDMFD sp!, {r1, r2, pc}

```

## 0.4 Function Calling Standards

When we are writing an application where in we write and use a collection of functions all by ourselves, then we can define our register preserving convention and as long as the function developers stick to the convention the functions can interoperate with each other perfectly subject to the correct overall program logic. However if we have to use libraries either standard ones or those provided by others, we have to follow certain standards. The same applies when either we call a function written in C or called by a function written in C. There has to be a common understanding between the assembly language programmer and the compiler.

**Question 10.** *Can you envisage any such convention problems when two functions have to interoperate with each other and both of them are written in C?*

In order solve all the inter-operability issues, all programmers, compilers have to follow what is called as a *function calling standard*. This is also called as *application binary interface (ABI)*. Here

we present some of ABI details that are relevant to the ideas who studied so far.

1. Parameters are passed through registers r0-r3. Any other extra parameters, they have to go through the stack. The callee function need not preserve the contents of these registers in any case (even if the callee takes only one parameter, it can still destroy the contents of register r3 for example). So if the caller functions cares about the contents of the registers r0-r3, it has to preserve them before it makes a function call. Similarly the register r12 contents will also not be preserved by the callee.
2. Return values are passed through the register r0 and r1.
3. Callee functions preserves the contents of the registers r4-r11. So if a callee function have to use these registers, it has to first save them on the stack and restore them back at the end.

Figure ?? shows a summary of the Application Binary Interface standard followed on Linux systems running on ARM processors.

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Figure 4: ABI for Linux Systems Running on ARM Processors.