ARM Assembly for Beginners

There are many different ways of writing assembly code. You are free to code in the style and syntax you prefer for ARM assembly. However if you are struggling to get going, you can make use of methods shown in this tutorial.

It is difficult to cover a lot of material in a tutorial, however I have listed **10 Key Steps to ARM** that might help you.

You can also look for additional information here http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0552a/BABJCCDH.html or any other site you prefer.

1. Two sections in code

'.data' section (data)

You can treat this like your global space in a C program. You can define string/int variables here. All addresses aligned in this space can be accessed throughout your code.

One example is shown below.

C Code

```
#include <stdio.h>

char *alpha = "abcdef"

int *key;

....
```

ARM Code

```
.data

.balign 4
alpha: .asciz "abcdef"

.balign 4
key: .word 0

....
```

'.text' section (instructions)

This section is where you have your function declarations and definitions. Talking in assembly terms ***subroutines***

One example is shown below.

C Code

```
/*declarations */
int getZero(int );
void doNothing(void );

/*definitions*/
int main(void) {

    return 0;
}

int getZero(int sumNumber){

    return 0;
}

void doNothing(void){

    return;
}
```

ARM Code

Notice below that main function also has a declaration unlike above. In assembly it is a must to have a declaration otherwise the compiler cannot differentiate between **subroutines** and **labels**

Also, for now ignore the body of the subroutines. This is just for you to get a gist of what goes inside **.text**

```
.text

/*declarations */
.global getZero
.global doNothing
.global main

/*definitions */
main:
    str lr, [sp, #-4]!

    mov r0, #0

    ldr lr, [sp], #+4
    bx lr

/*End of main */

getZero:
    str lr, [sp, #-4]!

    mov r0, #0

    ldr lr, [sp], #+4
    bx lr

/*End of getZero */

doNothing:
    str lr, [sp, #-4]!

    ldr lr, [sp], #+4
    bx lr

/*End of doNothing */
```

2. Alignment

You must have noticed the use of ***.balign*** in Step 1. balign = byte alignment. Though not the same, this can be related to the use of pointers.

On a 32-bit machine, you need to use

```
.balign 4
```

Whereas on a 64-bit machine, you need

```
.balign 8
```

Note: This has nothing to do with the size of data being stored. Alignment stays the same whether the address is pointing to chars, ints or arrays.

Since Raspberry Pi is 32-bit, all examples in this tutorial use 4 byte alignment.

3. Variables

In assembly there is no such thing as global or local variables. However I am making use of the terms *global* and *local* to help your understanding.

Global Variables

In assembly, there is not much distinction between data types either, everything is dealt with in bytes and treated as an address before it is de-referenced. Below listed are few ways of creating variables that will come handy to you.

```
.balign 4
sayHello: .asciz "Hello World!"  /*constant */

.balign 4
someString: .word 0     /*variable */

.balign 4
someNumber: .word 0     /*variable */

.balign 4
myArray: .comm array, 168  /*array that can hold 168 bytes. For example 42
integers 4 bytes each */

.balign 4
someChar: .word 0x58   /*constant with ASCII value that represents 'X' */
```

As you can notice, there is no distinction between a string variable and int variable. This might make you wonder, how do you create a large enough variable to hold a string of any size ? You can make use of **.skip**

C Code

```
char alphabet[26];
```

ARM Code

```
.balign 4
alphabet: .skip 26  /*Can hold upto 26 bytes */
```

Local Variables

There are absolutely no local variables that will keep values preserved until the end of the subroutine. You will have to make use of **registers** for performing instructions on values and ultimately the **stack** to preserve values.

Registers can be divided into the following categories:

- Return value register
- Argument register(s)
- Callee saved register(s)
- Stack pointer
- Frame pointer
- Link register

- Program Counter

Different registers can be used for different purposes, refer to Step 4. Also one register can be a part of one or more of these categories and thus registers are not the best way to preserve values after a function call. You should make use of the **stack** whenever you want to save a value for long term.

4. How to use Registers

Register Naming and uses

Name	Alternate name	Description
R0 - R3		Used to hold arguments for procedures and as scratch registers (for temporary storage). R0 is used to return the result of a function.
R4 - R9	V1 - V6	General purpose or storage of variables
R10	SL, V7	Stack limit pointer, used by assemblers for stack checking when this option is selected by the user.
R11	FP, V8	Frame pointer. Formal parameters are addressed as positive offsets from the frame pointer, and locals as negative offsets.[1]
R12	IP	Intra-Procedure-call scratch. Used with r0 - r3 for temporary storage and original contents do not need to be preserved.
R13	SP	Stack pointer
R14	LR	Link register, holding the return address from a function
R15	PC	Program counter, holding the address of the next instruction

[2]

Below is an example for performing basic instructions inside a subroutine using registers.

C Code

```
#include<stdio.h>

int *number;
char *string = "Welcome to CS250!"

int main(void) {
    int i = 0;
    i++;

    char *duplicate = string;

    int num = 100;

    *number = num;
}
```

ARM Code (Only showing the subroutine 'main')

```
mov R0, #0         /*int i = 0 */
add R0, R0, #1         /* i++; */

ldr R1, address_of_string   /* char *duplicate = string; */

mov R2, #100     /*int num = 100; */

ldr R3, address_of_number   /*load the address of number */
ldr R3, [R3]     /*de-reference address to get value -- *number; -- */
mov R3, R2     /*set the value to 100*/
```

5. Subroutines

Very Basic format of every subroutine

```
.text

.global doNothing

doNothing:
    str lr, [sp, #-4]!      /*push link register value on the stack */
    ....                    /*other instructions */
    ....
    ....

    ldr lr, [sp], #+4       /*assumes stack is at the same level */
    bx lr                   /*branch to address in link register */
```

Note: This needs to be done in every function/subroutine simply to return to the location from where the function was called, just like in any other program.

It doesn't matter how you preserve the value in link register till the end of the sub routine. You can also use a global variable to do so…

```
.data

.balign 4
returnValue: .word 0

.text

.global doNothing

doNothing:
    ldr r1, address_of_return
    str lr, [r1]
    ....
    ....
    ....

    ldr lr, address_of_return
    ldr lr, [lr]
    bx lr

address_of_return: .word returnValue
```

Passing Arguments to Subroutines

Subroutines do not have specific prototypes for setting parameters. By convention the following registers are treated as arguments.

Args	Register name
first argument	R0
second argument	R1
third argument	R2
fourth argument	R3

For passing more than four arguments you will have to use the **stack**. Otherwise values may not be stable

Calling Subroutines

Once you have moved the necessary values to corresponding argument registers, you need to **branch** to the subroutine. For example

```
b printf   /*branch to printf */
```

However, in doing so you may encounter a segmentation fault because you will not have an appropriate return address. Therefore you need to add the suffix **'l'** (lowercase L) to branching. Every time you make use or **bl** instead of **b**, you store the return address in link register, so the function knows where to return after execution.

```
bl printf
```

For more info on the suffix **'l'** refer to Step 7 and Step 8.

Return a value from Subroutine

By convention **R0** is the register for return value. Therefore, whichever value you wish to return from a function, move it to **R0**. For Example, the following function returns -10

```
.text

.global getValue
```

```
getValue:
    str lr, [sp, #-4]!

    mov r0, #-10

    ldr lr, [sp], #+4
    bx lr
```

Volatility of Register Values

Having learnt how value of registers are used across subroutines, it is intuitive that any subroutine may set some flags and vary value of registers as needed. Hence one should not rely too much on registers for preserving values for long term.

6. How to use the stack

Stack is a little tricky to use. Its only 1-2 lines of code but you need to remember what you added on what level of the stack. There are two ways of manipulating the stack.

- Either by Frame Pointer (FP)
- Or by the Stack Pointer (SP)

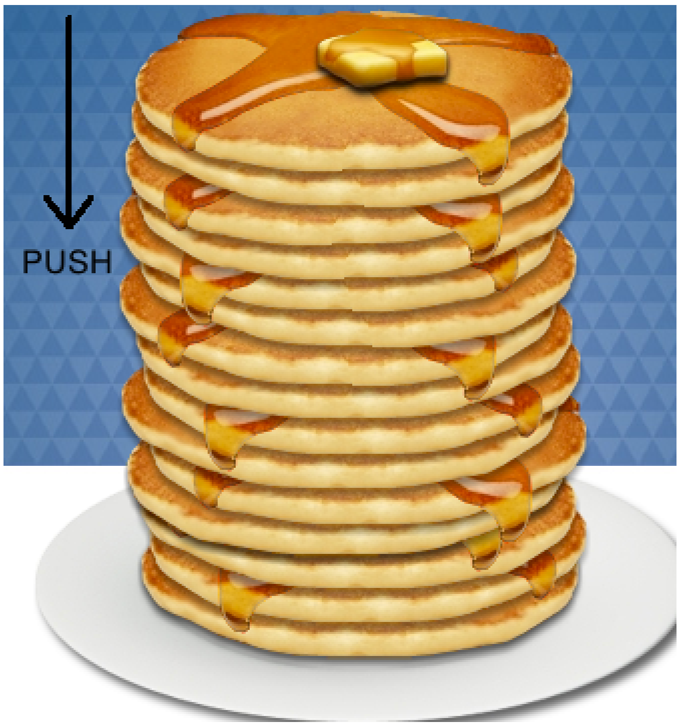
I prefer using the Stack Pointer, it really doesn't matter as far as you get it right. You already know there are two basic operations on any Stack:

- Push (Adding something on the stack)
- Pop (Removing something from the stack)

Note: All operations occur on Top level of the stack or in other words on **LIFO** basis.

Push

Whats interesting in assembly language is that, when trying to push/add something onto the stack you need to first make space on the stack, so you actually ***subtract*** an arbitrary number of bytes from the stack. For example on raspberry pi, you will subtract 4 bytes from the stack to make room for **one element**.



Vague Example: Think of Stack as a Stack of pancakes, of course there is a limit to how many elements you can stack in order to add more elements you will have push down a few by subtracting their level.

Pop

Pop works in the exact opposite manner. Every time you want to pop/remove something from the stack you need to ***add*** an arbitrary number of bytes to move the stack level upwards, making less room available on the stack.

An easy way to remember the relationship between Push and Pop is to think of it as **Malloc** and **Free** from CS240. Every push will need to have an equivalent pop or your program will throw a segmentation fault because you messed up some frames or the entire stack.

A good habit is always to divide a program into a good number of subroutines which deal with the Stack Pointer separately. This way it is easier to

1. Maintain the stack level
2. Test each subroutine seperately
3. Ultimately, know exactly where the segfault occurred and why

In very basic terms instead of pushing 10 items on the stack at once, try to divide the job such that you only push 3-4 items and then pop them in the appropriate order before you push more items.

Below are few examples that will help you get a better idea.

Example 1

```
someFunction:
    sub sp, sp, #4      /*Pushes the stack level down by 4 */
    ...
    ...
    ...
    add sp, sp, #4      /*Pops the stack level up by 4 */
```

Example 2

```
someFunction"
    str lr, [sp, #-4]!      /*Creates space at the same time stores address
in link register onto the stack ...
    ...                       Note: the exclamation mark and negative sign
are important */
    ...
    ...
    ldr lr, [sp], #+4       /*Loads the value from stack into the register
and also pops the stack level up ...
                              by 4 */
```

Example 3

```
someFunction:
    str r0, [sp, #-4]!
    str r1, [sp, #-4]!
    str r2, [sp, #-4]!
    ...
    ...
    ...
    ldr r5, [sp]            /*r5 has the same value as r2 */
    ldr r6, [sp, #+4]       /*r6 has the same value as r1 */
    ldr r7, [sp, #+8]       /*r7 has the same value as r0

/*Notice that without the exclamation (bottom lines), the value is only
loaded and not popped. */
```

For more on examples lookup Sample Stack.

7. Instructions

ARM assembly has many instructions, a few can be found

- here - http://ozark.hendrix.edu/~burch/cs/230/arm-ref.pdf
- and here - http://www.peter-cockerell.net/aalp/html/frames.html

Below are examples that show how to use some of the very common instructions, namely move, load, store, compare, add, subtract, multiply, left shift, right shift, and branch

Move

Used for moving immediate values between registers or for moving numbers to registers.

Note: Cannot be used to load something from 'data' memory.

```
/*mov destination, source */

/*Correct Uses */
mov r0, #100
mov r1, #200
mov r2, r0

/*Incorrect Uses */
mov r0, address_of_someThing
```

Load

Used for loading values either from 'data' memory or from the stack. Also used while de-refrencing addresses.

```
/*ldr destination, source */

/*Correct Uses */
ldr r5, address_of_someThing
ldr r5, [r5]

ldr r0, [sp]

/*Incorrect Uses */
ldr r2, #100
```

Store

Compare

Used for comparing values in two registers. There are flags that are set based on the comparison values that you may not need to know. Compare is always followed by branching to some or the other label based on the flags that get set. Combination of compare and branch is used as for, while, if, else and other conditionals.Refer to Step 8 for branching conditionals.

It is always the first register being compared to the second.

```
/*cmp register, register/value */

/*Correct uses */
mov r0, #0
mov r1, #1

cmp r0, r1
blt someLabel       /*branch if r0 is less that r1 */

/*OR */ cmp r0, #10

/*Incorrect uses */
mov r0, #0

cmp #1, r0         /*Will throw compile error */
blt someLabel
```

Add, Subtract, Multiply

These work as described. Can be used to add, subtract or multiple values in two registers and or numbers.

```
/*add/sub/mul destination, register, register/value */

/*Correct uses */

mov r0, #0
add r1, r0, #1    /*r1 now has value = 1 */
add r2, r1, #5    /*r2 now has value = 6 */

mov r3, #100
mov r4, #150
add r5, r3, r4    /*r5 now has value = 250 */

sub r5, r5, #200  /*r5 now has value = 50 */

mul r5, r5, #2    /*r5 now has value = 100 */

/*Incorrect Uses */

add/sub/mul #100, r1, r2

add/sub/mul r1, #100, r2
```

Left Shift, Right Shift

Binary operations are preferred over arithmetic on so many occasions. One important use of Right Shift in ARM assembly is to divide a number by power of 2. Yes there is no method for division in ARM, and division with an odd number like 7 can be very complicated.

```
/*LSL/LSR destination, register, register/value */
```

```
/*Correct uses */

mov r5, #2

/*Left Shift */
lsl r1, r5, #1        /*r1 now has value = 4 */
lsl r1, r1, #2        /*r1 now has value = 16 */

/*Right Shift */
lsr r1, r1, #1        /*r1 now has value = 8 */
lsr r1, r1, #2        /*r1 now has value  = 2 */

/*Incorrect uses */

There are many ways of incorrectly using this.
```

Branch

Is somewhat similar to **goto** statements in C. Can be used to branch to any label/subroutine.

```
/*b/bl label/subroutine */

mov r0, #0
mov r1, #1
mov r2, #3

b someLabel     /*code execution jumps/branches to someLabel */

mov r1, r0      /*dead code */
mov r2, r1      /*dead code */
mov r0, #100    /*dead code */

someLabel:
....
....
....
```

Well that was a bad example. But you got the idea of branching. Branching is often done after some sort of comparison or in other words, based on some sort of conditionals, these are explained below in Step 8.

8. Conditional branching

Branch Instructions: The branch instructions cause the processor to execute instructions from a different address. Two branch instructions are available - **b** and **bl**. The bl instruction in addition to branching, also stores the return address in the **lr** register, and hence can be used for sub-routine invocation. The instruction syntax is given below[3)]

Conditional Execution: Most other instruction sets allow conditional execution of branch instructions, based on the state of the condition flags. In ARM, almost all instructions can be conditionally executed. If corresponding condition is true, the instruction is executed. If the condition is false, the instruction is turned into a nop.

Instruction	Condition
BEQ or BLEQ	Equal
BNE or BLNE	Not Equal
BCS or BLCS	Carry Set
BCC or BLCC	Carry Clear
BVC or BLVC	Overflow Clear
BVS or BLVS	Overflow Set
BPL or BLPL	Positive
BMI or BLMI	Minus
BHI or BLHI	Higher Than
BHS or BLHS	Higher Than or Same
BLO or BLLO	Lower Than
BLS or BLLS	Lower Than or Same
BGT or BLGT	Greater Than
BGE or BLGE	Greater Than or Equal
BLT or BLLT	Less Than
BLE or BLLE	Less Than or Equal

The example below shows conditional branching

```
 /*Some branching */
```

9. Compiling

All your code needs to be saved in a file with extension ***.s*** . Let us name our file **hello.s**

Copy code from the example in Step 10 into the file and compile it.

```
pi@raspberrypi:~$ gcc -std=c99 -o hello hello.s
```

Now run the program.

```
pi@raspberrypi:~$ ./hello
Hello World!
```

Congrats you just compiled your first program in ARM assembly.

But lets not stop here, the goal is to be able to make more complex programs by the end of the course. And as you progress you are very likely to encounter the all time favorites - **segmentation faults** To tackle these, compile your program with a **-g** flag and use gdb and try to backtrace the problem.

```
pi@raspberrypi:~$ gcc -g -std=c99 -o hello hello.s
pi@raspberrypi:~$ gdb ./hello
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/pi/hello...(no debugging symbols found)...done.
(gdb) run
 ...
 ...
(gdb) backtrace
 ...
 ...
(gdb) frame 0 or 1 or 2 or 3 or 4 ...
```

10. Code Example

Let us begin with the traditional “Hello World!” program.

Example 1: This code uses '*address_of_someThing*' to reference variables.

```
.data /* The end of .data section is marked by the beginning of .text
section */

.balign 4 /* Make room of 4 bytes (on 32-bit machine) to accommodate an
address */
message: .asciz "Hello World!" /*address is now pointing to the string
constant "Hello World!"

.balign 4
format: .asciz "%s\n" /* asciz means ascii + zero termination (null
termination) at the end */
                      /* an alternate way of writing the above is, .ascii
"%s\n\0" */

.balign 4
return: .word 0

.text

.global main

main:
        ldr r1, address_of_return
        str lr, [r1]                  /*Saving the address where function
needs to return eventually */

        ldr r0, address_of_format     /*moving first argument for printf to
r0 */
        ldr r1, address_of_message    /*moving second argument for printf to
r1 */
        bl printf                     /* call printf */


        ldr lr, address_of_return     /*load the address of return to the
link register */
        ldr lr, [lr]                  /*de-reference the address to obtain a
value */
        bx lr                         /*branch to the address pointed by
link register */


address_of_message: .word message     /*method of referencing elements in
global space */
address_of_format: .word format       /*alternative to writing
'address_of_something', is to use '=' as prefix */
address_of_return: .word return       /*for example ldr r0, =message (loads
the address_of_message into r0 */
```

Example 2: This code uses '=' to reference variables.

```
.data

.balign 4
message: .asciz "Hello World!"

.balign 4
format: .asciz "%s\n"


.balign 4
return: .word 0

.text

.global main

main:

        ldr r1, =return      /*Notice the difference here */
        str lr, [r1]

        ldr r0, =format
        ldr r1, =message
        bl printf


        ldr lr, =return
        ldr lr, [lr]
        bx lr

/* ... and here */
```

Fun Coding (Not Graded)

[1] Adapted from http://www.pp4s.co.uk/main/tu-trans-comp-jc-13.html#co-tu-trans-comp-jc-13__local
[2] Adapted from http://www.pp4s.co.uk/main/tu-trans-asm-arm.html
[3] Adapted from http://www.bravegnu.org/gnu-eprog/arm-iset.html