Introduction to ARM

Hello who is this?

Quick overview

ARM has multiple instructions mode:

- Thumb (all instruction are 16 bits)
- 32 bits (all instruction are 32 bits long)
- Jazelle (Java byte code)
-

Other instructions exists but these are the 3 more common.

As part of this introduction, we will focus exclusively on the 32 bits instructions.

Important difference with x86, ARM uses the Load / Store architecture.

Meaning: only load or store instructions can do memory access

Registers

GPRs available are r0 to r15. All registers are 32 bits wide.

r13 is also known as SP (stack pointer)
r14 is also known as LR (link register)
r15 is also known as PC (program counter)

When working with Gnu tools, you will often see r11 named as fp. Gnu tools uses r11 as a stack frame pointer.

The “flags” register is called CPSR (current program status register. It is also 32 bits wide.

R0
...
R12
R13 / SP
R14 / LR
R15 / PC

If ALL instructions and all registers are 32 bits, what is the consequence when loading values into registers?

Impact of fixed size instruction

Intel mov:

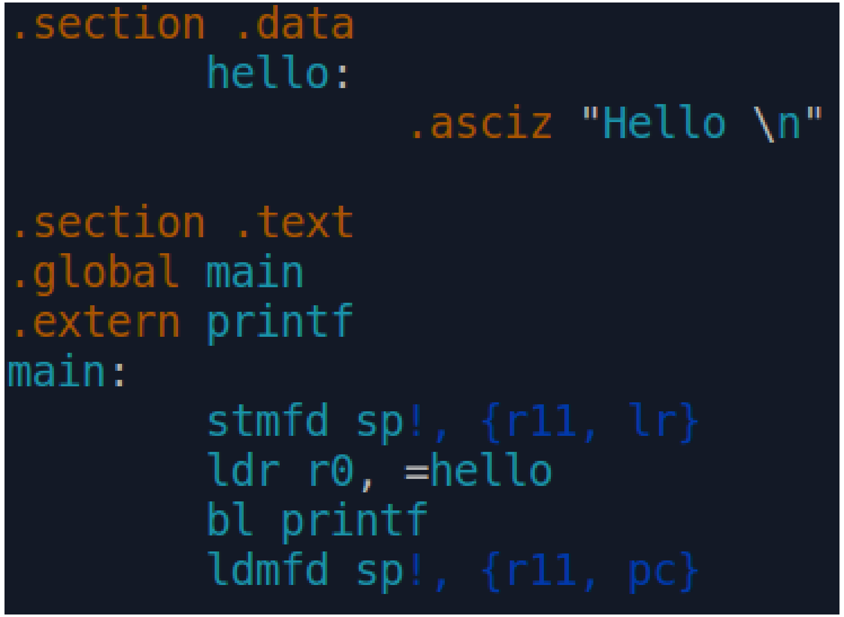
```
b8 e8 fd 00 00      mov    eax, 0xfde8   ; This is 65000
```

Arm ldr:

```
e51f0000    ldr  r0, [pc, #-0]       ; ?!?!?!
```

Unless you have prior ARM experience, you are probably wondering what is going on now right?

All instructions are encoded using 32 bits. Because of that, it is not possible to “hardcode” a 32 bit pointer as part of an instruction… Here’s an example.

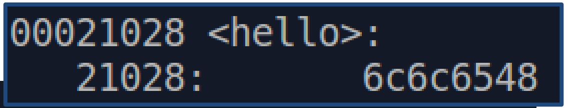


```
.section .data
        hello:
                .asciz "Hello \n"

.section .text
.global main
.extern printf
main:
        stmfd sp!, {r11, lr}
        ldr r0, =hello
        bl printf
        ldmfd sp!, {r11, pc}
```

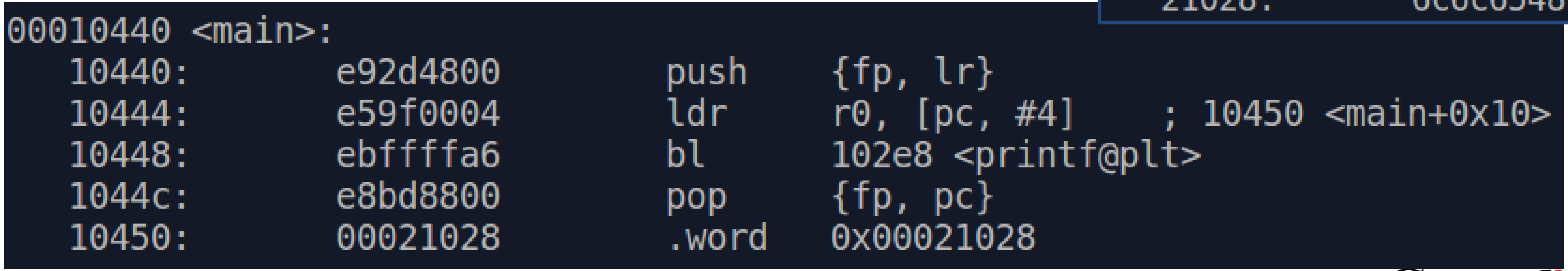
To the left, original code.

Below is the code after the assembler made its magic.

As you can see the assembler is changing the code a bit and is using a relative memory access to a zone of memory that is within reach. That memory then contain a pointer to the data we are looking for.



```
00021028 <hello>:
   21028:       6c6c6548
```



```
00010440 <main>:
   10440:       e92d4800        push    {fp, lr}
   10444:       e59f0004        ldr     r0, [pc, #4]    ; 10450 <main+0x10>
   10448:       ebfffa6         bl      102e8 <printf@plt>
   1044c:       e8bd8800        pop     {fp, pc}
   10450:       00021028        .word   0x00021028
```

General instructions

Mnemonic	Example	Description
mov	mov r0, r1 mov r0, #0xFFFF	Move data across registers. Can also load a constant into a register (maximum value is 0xFFFF).

ldr	`ldr r0, =0xFFFFFFFF    ; Pseudo instruction`
	`ldr r0, =test       ; Pseudo instruction!`
	`ldr r0, [r0]        ; Deref r0`
	`ldr r0, [r0, #1]    ; Offset 1 byte pass r0`
	`ldr r0, [r0, #1]!   ; Offset 1 byte pass r0 add 1 to r0 after`
	`ldrb r0, [r0]       ; Load r0 with one byte found at r0`
	
	The pseudo instructions are used to load values that are too big to be encoded in an instruction. See previous example!
		Load register with value. Use of square brackets means the address accessed is dereferenced. Size modificator can be added at the end of the mnemonic (b for byte zero extended, bs for byte zign extended, h for halfword zero extended, hs half word sign extended).

General instructions

Mnemonic	Example	Description
str	str r1, [r0] ; Deref r0 str r1, [r0, #1] ; Offset 1 byte pass r0 str r1, [r0, #1]! ; Offset 1 byte pass r0 add 1 to r0 after strb r1, [r0] ; Load r0 with one byte found at r0	Store register value at address. Size modificator can be added at the end of the mnemonic (b for byte zero extended, bs for byte zign extended, h for halfword zero extended, hs half word sign extended).

ldmfd	ldmfd sp!, {r11, pc}
	pop {r11, pc}
		Load multiple registers from the stack (SP).

stmfd	stmfd sp!, {r11, lr}
	push {r11, pc}
		Store multiple registers to the stack (SP)

General instructions

Mnemonic	Example	Description
add	add r0, r1, r2 ; r0 = r1 + r2 add r0, r0, r1 ; r0 = r0 + r1 add r0, r1, #255 ; r0 = r1 + 255	Addition. When using a constant as the last operant, the maximum value is 255.
sub	sub r0, r1, r2 ; r0 = r1 - r2 sub r0, r0, r1 ; r0 = r0 - r1 sub r0, r1, #255 ; r0 = r1 - 255	Subtraction. When using a constant as the last operant, the maximum value is 255.
and	and r0, r1, r2 ; r0 = r1 & r2	And operation.
orr	orr r0, r1, r2 ; r0 = r1 | r2	Or operation.
eor	eor r0, r1, r2 ; r0 = r1 ^ r2	Exclusive or operation

ARM operations do not change the CPSR. Conditional branching requires this. For an instruction to change CPSR, you need to add the suffix “s” to a mnemonic.

example: ands r0, r0, r1

Have a quick look at ARM documentation for the “and” instruction. You should find something nice… What is it ?

Most ARM instruction have the format:

mnemonic{s}{cond} dReg, sReg, Operand2

Branches

The “b” instruction allows for a branch to happen.
this:

- b test

Is equivalent to this on x86:

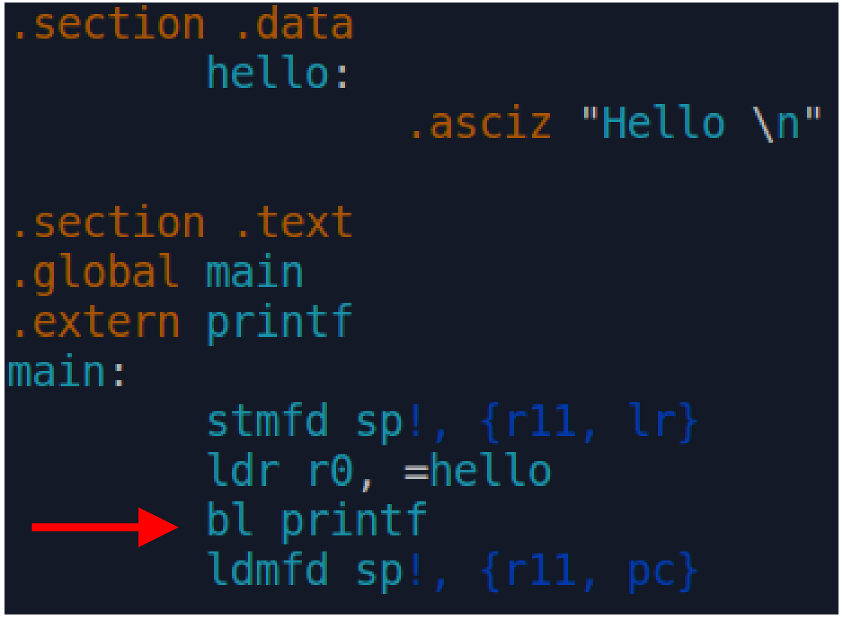
- jmp test

Conditional branches

The “b” instruction can be complemented by a condition. This allows for conditional branches. These can also be used in doing conditional execution.

Condition tag	**Description**
eq	If equal (Z is set)
ne	If not equal (if Z is not set)
hs	If higher or same (unsigned, use “ge” for signed version)
hi	If higher (unsigned, use “gt” for signed version)
ls	If lower or same (unsigned, use “ls” for signed version)
lo	If lower (unsigned, use “lt” for signed version)

Branch with link (bl)



```
.section .data
        hello:
                .asciz "Hello \n"

.section .text
.global main
.extern printf
main:
        stmfd sp!, {r11, lr}
        ldr r0, =hello
        bl printf
        ldmfd sp!, {r11, pc}
```

The “bl” instruction is somewhat similar to the “call” instruction on x86.

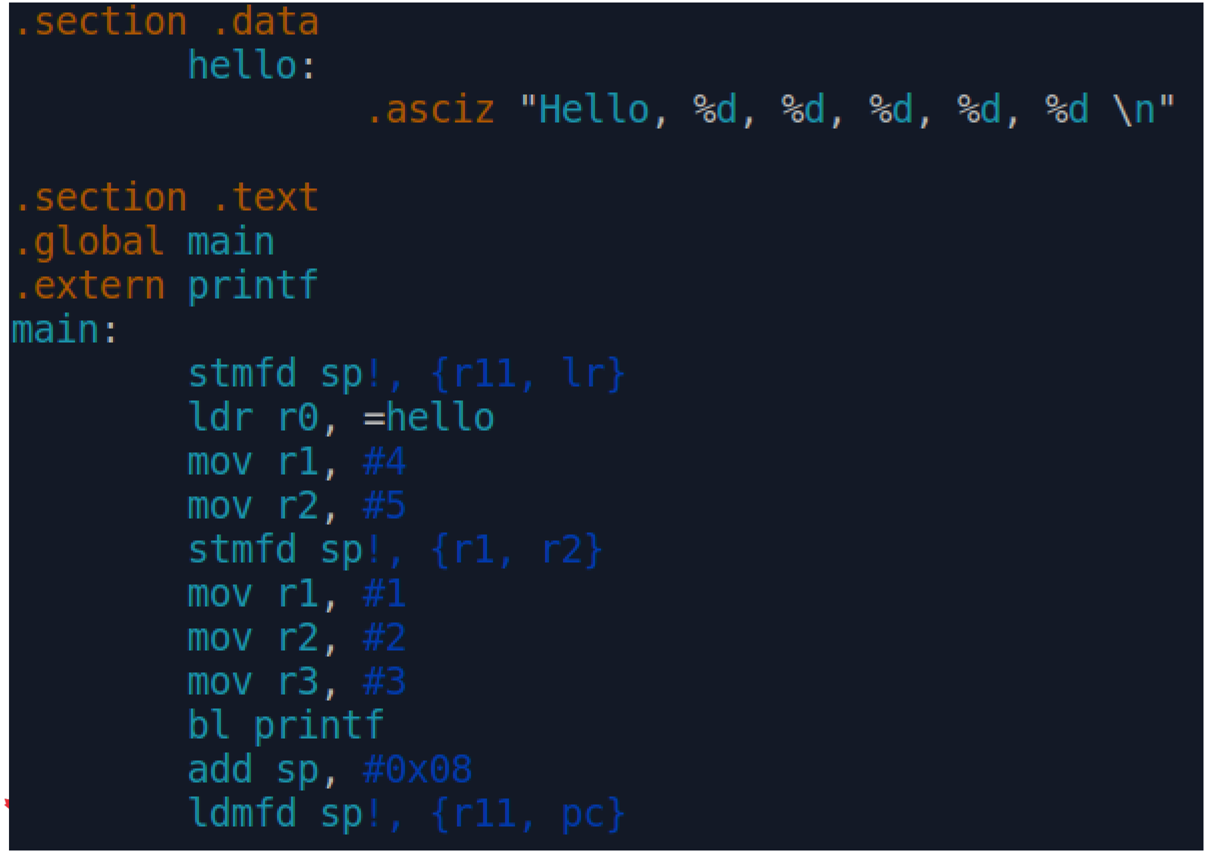
There is one major difference: the “return” address is not pushed on the stack. When using “bl”, the return address is stored in the “lr” (link return) register.

When returning from a function call, we copy the value of “lr” to “pc” and execution resume to the intended “return” address

Function call calling convention

- When using Linux on an arm 32 bits architecture, the function calling convention goes as:
  - r0, r1, r2, r3 are used for the first 4 parameters sent into a function
  - The other parameters are put in reverse order on the stack
    - The caller is responsible for cleaning the stack
  - The return value from a function is put in r0 and r1 (r1 for values larger than 32 bits, this is like edx:eax on x86)
  - The stack needs to be 8 bytes aligned
  - r0, r1, r2, r3, r12 are volatile (available for use to the programmer)
  - All other registers must be saved by the callee

Function call example



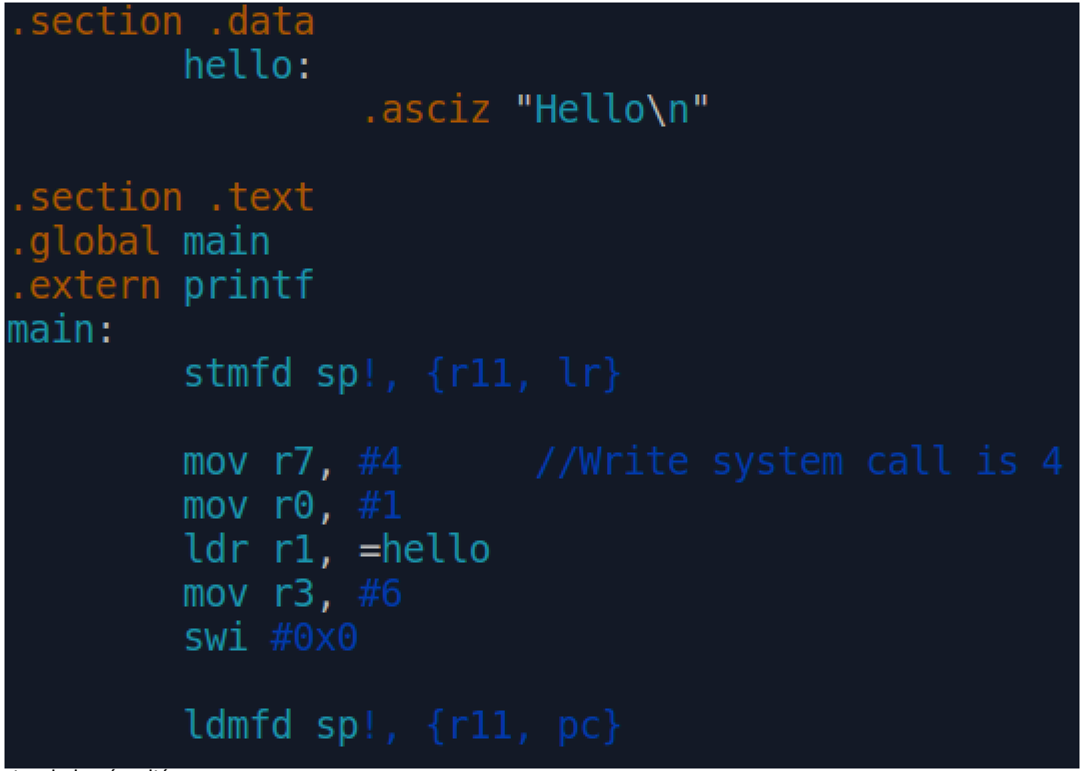
```
.section .data
        hello:
                .asciz "Hello, %d, %d, %d, %d, %d \n"

.section .text
.global main
.extern printf
main:
        stmfd sp!, {r11, lr}
        ldr r0, =hello
        mov r1, #4
        mov r2, #5
        stmfd sp!, {r1, r2}
        mov r1, #1
        mov r2, #2
        mov r3, #3
        bl printf
        add sp, #0x08
        ldmfd sp!, {r11, pc}
```

The cool thing is that “stmfd” will deal with the stack for you =)

System call calling convention

- When using Linux on an arm 32 bits architecture, the system call calling convention goes as:
  - Instruction to launch the system call is “swi #0x0”
  - System call number goes in r7
  - System call return value goes in r0
  - Parameters are put, in order, into registers:
    - r0, r1, r2, r3, r4, r5, r6
- Simple enough!
- The swi instruction stands for “software interrupt”
- System call numbers can be found in:
  - /usr/include/arm-linux-gnueabihf/asm/unistd.h

Code example



```
.section .data
        hello:
                .asciz "Hello\n"

.section .text
.global main
.extern printf
main:
        stmfd sp!, {r11, lr}

        mov r7, #4      //Write system call is 4
        mov r0, #1
        ldr r1, =hello
        mov r3, #6
        swi #0x0

        ldmfd sp!, {r11, pc}
```

Bottom line: While assembly languages have differences, in the end they usually are pretty close one to another.

Let’s write some code!