ARM assembler in Raspberry Pi

Table of contents

Do you have a Raspberry Pi and you fancy to learn some assembler just for fun? These posts are for you!

24. Trampolines
25. Integer SIMD
26. A primer about linking

ARM assembler in Raspberry Pi – Chapter 1

January 9, 2013 Roger Ferrer Ibáñez, 69

In my opinion, it is much more beneficial learning a high level language than a specific architecture assembler. But I fancied learning some ARM assembler just for fun since I know some 386 assembler. The idea is not to become a master but understand some of the details of what happens underneath.

Introducing ARM

You will see that my explanations do not aim at being very thorough when describing the architecture. I will try to be pragmatic.

ARM is a 32-bit architecture that has a simple goal in mind: flexibility. While this is great for integrators (as they have a lot of freedom when designing their hardware) it is not so good for system developers which have to cope with the differences in the ARM hardware. So in this text I will assume that **everything is done on a Raspberry Pi Model B running Raspbian** (the one with 2 USB ports and 512 MB of RAM).
Some parts will be ARM-generic but others will be Raspberry Pi specific. I will not make a distinction. The ARM website has a lot of documentation. Use it!

Writing assembler

Assembler language is just a thin syntax layer on top of the binary code.

Binary code is what a computer can run. It is composed of instructions, that are encoded in a binary representation (such encodings are documented in the ARM manuals). You could write binary code encoding instructions but that would be painstaking (besides some other technicalities related to Linux itself that we can happily ignore now).

So we will write assembler, ARM assembler. Since the computer cannot run assembler we have to get binary code from it. We use a tool called, well, *assembler* to *assemble* the *assembler code* into a binary code that we can run.
The tool to do this is called as. In particular GNU Assembler, which is the assembler tool from the GNU project, sometimes it is also known as gas for this reason. This is the tool we will use to assemble our programs.

Just open an editor like vim, nano or emacs. Our assembler language files (called *source files*) will have a suffix .s. I have no idea why it is .s but this is the usual convention.

Our first program

We have to start with something, so we will start with a ridiculously simple program which does nothing but return an error code.

```
/* -- first.s */
/* This is a comment */
.global main /* 'main' is our entry point and must be global
*/

main:           /* This is main */
    mov r0, #2 /* Put a 2 inside the register r0 */
    bx lr      /* Return from main */
```

Create a file called first.s and write the contents shown above. Save it.
To *assemble* the file type the following command (write what comes after $).

```
$ as -o first.o first.s
```

This will create a first.o. Now link this file to get an executable.

```
$ gcc -o first first.o
```

If everything goes as expected you will get a first file. This is your program. Run it.

```
$ ./first
```

It should do nothing. Yes, it is a bit disappointing, but it actually does something. Get its error code this time.

```
$ ./first ; echo $?
2
```

Great! That error code of 2 is not by chance, it is due to that #2 in the assembler code.
Since running the assembler and the linker soon becomes boring, I'd recommend you using the following Makefile file instead or a similar one.

```
# Makefile
```

```
2
3  all: first
4
5  first: first.o
6          gcc -o $@ $+
7
8  first.o : first.s
9          as -o $@ $<
1
0  clean:
1          rm -vf first *.o
1
```

Well, what happened?

We cheated a bit just to make things a bit easier. We wrote a C main function in assembler which only does return 2;. This way our program is easier since the C runtime handled initialization and termination of the program for us. I will use this approach all the time.
Let's review every line of our minimal assembler file.

```
1 /* -- first.s */
2 /* This is a comment */
```

These are comments. Comments are enclosed in /* and */. Use them to document your assembler as they are ignored. As usually, do not nest /* and */ inside /* because it does not work.

```
3 .global main /* 'main' is our entry point and must be global
*/
```

This is a directive for GNU Assembler. A directive tells GNU Assembler to do something special. They start with a dot (.) followed by the name of the directive and some arguments. In this case we are saying that main is a global name. This is needed because the C runtime will call main. If it is not global, it will not be callable by the C runtime and the linking phase will fail.

```
5 main:          /* This is main */
```

Every line in GNU Assembler that is not a directive will always be like label: instruction. We can omit label: and instruction (empty and blank lines are ignored). A line with only label:, applies that label to the next line

(you can have more than one label referring to the same thing this way). The instruction part is the ARM assembler language itself. In this case we are just defining main as there is no instruction.

```
6     mov r0, #2 /* Put a 2 inside the register r0 */
```

Whitespace is ignored at the beginning of the line, but the indentation suggests visually that this instruction belongs to the main function.

This is the mov instruction which means *move*. We move a value 2 to the register r0. In the next chapter we will see more about registers, do not worry now. Yes, the syntax is awkward because the destination is actually at left. In ARM syntax it is always at left so we are saying something like *move to register r0 the immediate value 2*. We will see what *immediate value* means in ARM in the next chapter, do not worry again.

In summary, this instruction puts a 2 inside the register r0 (this effectively overwrites whatever register r0 may have at that point).

```
7     bx lr      /* Return from main */
```

This instruction bx means *branch and exchange*. We do not really care at this point about the *exchange* part. Branching means that we will change the flow of the instruction execution. An ARM processor runs instructions sequentially, one after the other, thus after the mov above, this bx will be run (this sequential execution is not specific to ARM, but what happens in almost all architectures). A branch instruction is used to change this implicit sequential execution. In this case we branch to whatever lr register says. We do not care now what lr contains. It is enough to understand that this instruction just leaves the main function, thus effectively ending our program.

And the error code? Well, the result of main is the error code of the program and when leaving the function such result must be stored in the register r0, so the mov instruction performed by our main is actually setting the error code to 2.

That’s all for today.

ARM assembler in Raspberry Pi – Chapter 2

January 10, 2013 Roger Ferrer Ibáñez, 4

Registers

At its core, a processor in a computer is nothing but a powerful calculator. Calculations can only be carried using values stored in very tiny memories called *registers*. The ARM processor in a Raspberry Pi has 16 integer registers and 32 floating point registers. A processor uses these registers to perform integer computations and floating point computations, respectively. We will put floating registers aside for now and eventually we will get back to them in a future installment. Let's focus on the integer registers.

Those 16 integer registers in ARM have names from r0 to r15. They can hold 32 bits. Of course these 32 bits can encode whatever you want. That said, it is convenient to represent integers in two's complement as there are instructions which perform computations assuming this encoding. So from now, except noted, we will assume our registers contain integer values encoded in two's complement.

Not all registers from r0 to r15 are used equally but we will not care about this for now. Just assume what we do is "OK".

Basic arithmetic

Almost every processor can do some basic arithmetic computations using the integer registers. So do ARM processors. You can ADD two registers. Let's retake our example from the first chapter.

```
1 /* -- sum01.s */
2 .global main
3
4 main:
5     mov r1, #3      /* r1 ← 3 */
6     mov r2, #4      /* r2 ← 4 */
7     add r0, r1, r2  /* r0 ← r1 + r2 */
8     bx lr
```

If we compile and run this program, the error code is, as expected, 7.

```
$ ./sum01 ; echo $?
```

7
Nothing prevents us to reuse r0 in a more clever way.

```
/* -- sum02.s */
.global main

main:
    mov r0, #3      /* r0 ← 3 */
    mov r1, #4      /* r1 ← 4 */
    add r0, r0, r1  /* r0 ← r0 + r1 */
    bx lr
```

Which behaves as expected.

```
$ ./sum02 ; echo $?
7
```

That’s all for today.

ARM assembler in Raspberry Pi – Chapter 3

January 11, 2013 Roger Ferrer Ibáñez, 31

We saw in chapter 1 and chapter 2 that we can move values to registers (using mov instruction) and add two registers (using add instruction). If our processor were only able to work on registers it would be rather limited.

Memory

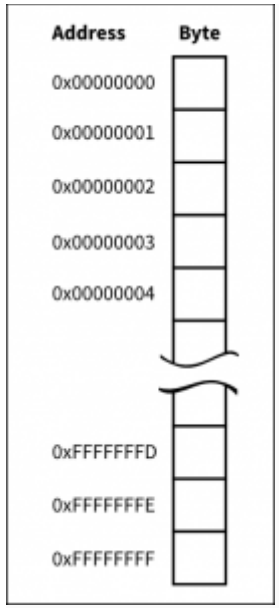
A computer has a memory where code (.text in the assembler) and data are stored so there must be some way to access it from the processor. A bit of digression here, in 386 and x86-64 architectures, instructions can access registers or memory, so we could add two numbers, one of which is in memory. You cannot do this in ARM where all operands must be registers. We can work around this problem (not really a problem but a deliberate decision design that goes beyond the scope of this text) by loading data to a register from memory and storing data from a register to a memory.

These two special operations, loading and store, are instructions on their own called usually *load* and *store*. There are several ways to load and store data from/to memory but today we will focus on the simplest ones: load to register ldr and store from register str.

Loading data from memory is a bit complicated because we need to talk about *addresses*.

Addresses

To access data we need to give it a name. Otherwise we could not refer what piece of data we want. But of course a computer does not have a different name for every piece of data it can keep in memory. Well, in fact it does have a name for every piece of data. It is the *address*. The address is a number, in ARM a 32-bit number that identifies every byte (this is 8 bits) of the memory.



Memory is like an array of bytes where each byte has its own address.

When loading or storing data from/to memory we need to compute an address. This address can be computed in many ways. Each of this modes is called an *addressing mode*. ARM has several of these addressing modes and it would took a while to explain them all here, so we will consider just one: addressing through a register.
It is not by chance that ARM has integer registers of 32 bits and the addresses of the memory are 32 bit numbers. This means that we can keep an address inside a register. Once we have an address inside a register, we can use that register to load or store some piece of data.

Data

We saw in the chapter 1 that the assembler contains both code (called *text*) and data. I was deliberately loose when describing labels of the assembler. Now we can unveil their deep meaning: labels in the assembler are just symbolic names to addresses in your program. These addresses may refer both to data or code. So far we have used only one

label main to designate the address of our main function. A label only denotes an address, never its contents. Bear this in mind.
I said that assembler is a thin layer on top of the binary code. Well, that thin layer may now look to you a bit thicker since the assembler tool (as) is left responsible of assigning values to the addresses of the labels. This way we can use these labels and the assembler will do some magic to make it work.
So, we can define data and attach some label to its address. It is up to us, as assembler programmers, to ensure the storage referenced by the label has the appropiate size and value.

Let's define a 4 byte variable and initialize it to 3. We will give it a label myvar1.

```
.balign 4
myvar1:
    .word 3
```

There are two new assembler directives in the example above: .balign and .word. When asencounters a .balign directive, it ensures the next address will start a 4-byte boundary. This is, the address of the next datum emitted (i.e. an instruction but it could be data as well) will be a multiple of 4 bytes. This is important because ARM imposes some restrictions about the addresses of the data you may work. This directive does nothing if the address was already aligned to 4. Otherwise the assembler tool will emit some *padding* bytes, which are not used at all by the program, so the alignment requested is fulfilled. It is possible that we could omit this directive if all the entities emitted by the assembler are 4 byte wide (4 bytes is 32 bits), but as soon as we want to use differently sized data this directive will become mandatory.
Now we define the address of myvar1. Thanks to the previous .balign directive, we know its address will be 4 byte aligned.
.word directive states that the assembler tool should emit the value of the argument of the directive as a 4 byte integer. In this case it will emit 4 bytes containing the value 3. Note that we rely on the fact that .word emits 4 bytes to define the size of our data.

Sections

Data lives in memory like code but due to some practical technicalities, that we do not care very much now, it is usually kept together in what is called a *data section*. .data directive tells the assembler to emit the entities in the *data section*.

That .text directive we saw in the first chapter, makes a similar thing for code. So we will put data after a .data directive and code after a .text.

Load

Ok, we will retrieve our example from the Chapter 2 and enhance it with some accesses to memory. We will define two 4 byte variables myvar1 and myvar2, initialized to 3 and 4 respectively. We will load their values using ldr, and perform an addition. The resulting error code should be 7, like that of chapter 2.

```
/* -- load01.s */

/* -- Data section */
.data

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar1 */
myvar1:
    /* Contents of myvar1 is just 4 bytes containing value '3'
*/
    .word 3

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is just 4 bytes containing value '4'
*/
    .word 4

/* -- Code section */
.text

/* Ensure code is 4 byte aligned */
.balign 4
.global main
main:
```

```
    ldr r1, addr_of_myvar1 /* r1 ← &myvar1 */
    ldr r1, [r1]           /* r1 ← *r1 */
    ldr r2, addr_of_myvar2 /* r2 ← &myvar2 */
    ldr r2, [r2]           /* r2 ← *r2 */
    add r0, r1, r2         /* r0 ← r1 + r2 */
    bx lr

/* Labels needed to access data */
addr_of_myvar1 : .word myvar1
addr_of_myvar2 : .word myvar2
```

I have cheated a bit in the example above because of limitations in the assembler. As you can see there are four ldr instructions. I will try to explain their meaning. First, though, we have to discuss the following two labels.

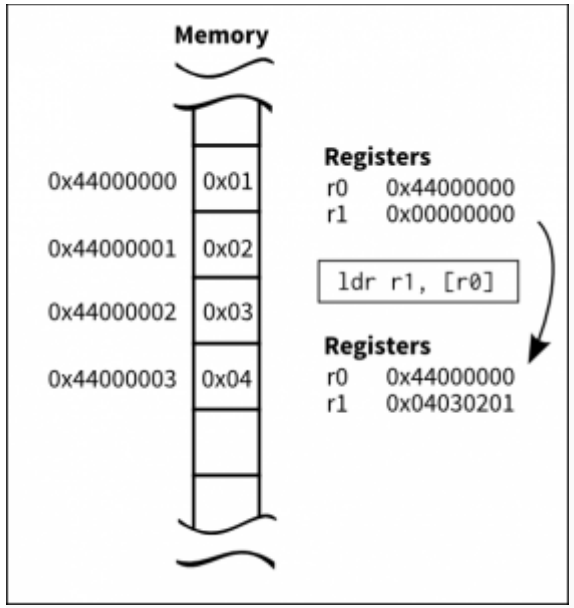
```
3
4
3   /* Labels needed to access data */
5   addr_of_myvar1 : .word myvar1
3   addr_of_myvar2 : .word myvar2
6
```

Well, these two labels, contain the address of myvar1 and myvar2. You may be wondering why we need them if we already have the address of our data in labels myvar1 and myvar2. Well a detailed explanation is a bit long, but what happens here is that myvar1 and myvar2 are in a different section: in the .data section. That section exists so the program can modify it, this is why variables are kept there. On the other hand, code is not usually modified by the program (for efficiency and for security reasons). So this is a reason to have two different sections with different properties attached to them. But, we cannot directly access a symbol from one section to another one. Thus, we need a special label in .code which refers to the address of an entity in .data section.

Well, when the assembler emits the binary code, .word myvar1 will not be address of myvar1but instead it will be a *relocation*. A relocation is the way the assembler uses to emit an address, the exact value of which is unknown but it will known be when the program is *linked* (i.e. when generating the final executable). It is like saying well, I have no idea where this variable will actually be, let's the linker patch this value for me later. So this addr_of_myvar1 will be used instead. The address of addr_of_myvar1 is in the same section .text. That value will be *patched* by the linker during the linking phase (when the final executable is created and it knows where all the entities of our program will definitely be laid out in memory). This is why the linker (invoked internally by gcc) is called ld. It stands for Link eDitor.

```
2
7       ldr r1, addr_of_myvar1 /* r1 ← &myvar1 */
2       ldr r1, [r1]           /* r1 ← *r1 */
8
```

Ok, so two loads. The first one in line 27 actually loads the relocation value of the address of myvar1. This is, there is some data in memory, the address of which is addr_of_myvar1, with a size of 4 bytes containing the real address of myvar1. So after the first ldr, in r1 we have the real address of myvar1. But we do not want the address at all, but the contents of the memory at that address, thus we do a second ldr.



Assuming the given memory contents, this is what happens to registers after a load instruction is executed.

Probably you are wondering why the two loads have different syntax. The first ldr uses the symbolic address of addr_of_myvar1 label. The second ldr uses the value of the register as the *addressing mode*. So, in the second case we are using the value inside r1 as the address. In the first case, we do not actually know what the assembler uses as the addressing mode, so we will ignore it for now.

The program loads two 32 bit values from myvar1 and myvar2, that had initial values 3 and 4, adds them and sets the result of the addition as the error code of the program in the r0 register just before leaving main.

```
$ ./load01 ; echo $?
7
```

Store

Now take the previous example but instead of setting the initial values of myvar1 and myvar2 to 3 and 4 respectively, set both to 0. We will reuse the existing code but we will prepend some assembler to store a 3 and a 4 in the variables.

```
/* -- store01.s */

/* -- Data section */
.data

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar1 */
myvar1:
    /* Contents of myvar1 is just '3' */
    .word 0

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is just '3' */
    .word 0

/* -- Code section */
.text

/* Ensure function section starts 4 byte aligned */
.balign 4
.global main
main:
    ldr r1, addr_of_myvar1 /* r1 ← &myvar1 */
    mov r3, #3             /* r3 ← 3 */
    str r3, [r1]           /* *r1 ← r3 */
    ldr r2, addr_of_myvar2 /* r2 ← &myvar2 */
    mov r3, #4             /* r3 ← 4 */
    str r3, [r2]           /* *r2 ← r3 */

    /* Same instructions as above */
    ldr r1, addr_of_myvar1 /* r1 ← &myvar1 */
    ldr r1, [r1]           /* r1 ← *r1 */
    ldr r2, addr_of_myvar2 /* r2 ← &myvar2 */
    ldr r2, [r2]           /* r2 ← *r2 */
```

```
4
2
5     add r0, r1, r2
2     bx lr
6
2  /* Labels needed to access data */
7  addr_of_myvar1 : .word myvar1
2  addr_of_myvar2 : .word myvar2
8
```

Note an oddity in the str instruction, the destination operand of the instruction is **not the first operand**. Instead the first operand is the source register and the second operand is the addressing mode.

$ ./store01; echo $?
7

That’s all for today.

ARM assembler in Raspberry Pi – Chapter 4

January 12, 2013 Roger Ferrer Ibáñez, 14

As we advance learning the foundations of ARM assembler, our examples will become longer. Since it is easy to make mistakes, I think it is worth learning how to use GNU Debugger gdb to debug assembler. If you develop C/C++ in Linux and never used gdb, shame on you. If you know gdb this small chapter will explain you how to debug assembler directly.

gdb

We will use the example store01 from chapter 3. Start gdb specifying the program you are going to debug.

```
$ gdb --args ./store01
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
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:
...
Reading symbols from /home/roger/asm/chapter03/store01...(no debugging symbols found)...done.
(gdb)
```

Ok, we are in the *interactive* mode of gdb. In this mode you communicate with gdb using commands. There is a builtin help command called help. Or you can check the GNU Debugger Documentation. A first command to learn is

```
(gdb) quit
```

Ok, now start gdb again. The program is not running yet. In fact gdb will not be able to tell you many things about it since it does not have debugging info. But this is fine, we are debugging assembler, so we do not need much debugging info. So as a first step let's start the program.

```
(gdb) start
Temporary breakpoint 1 at 0x8390
Starting program: /home/roger/asm/chapter03/store01

Temporary breakpoint 1, 0x00008390 in main ()
```

Ok, gdb ran our program up to main. This is great, we have skipped all the initialization steps of the C library and we are about to run the first instruction of our main function. Let's see whats there.

```
(gdb) disassemble
Dump of assembler code for function main:
=> 0x00008390 : ldr     r1, [pc, #40]   ; 0x83c0
   0x00008394 : mov     r3, #3
   0x00008398 : str     r3, [r1]
   0x0000839c : ldr     r2, [pc, #32]   ; 0x83c4
   0x000083a0 : mov     r3, #4
   0x000083a4 : str     r3, [r2]
   0x000083a8 : ldr     r1, [pc, #16]   ; 0x83c0
   0x000083ac : ldr     r1, [r1]
   0x000083b0 : ldr     r2, [pc, #12]   ; 0x83c4
   0x000083b4 : ldr     r2, [r2]
   0x000083b8 : add     r0, r1, r2
   0x000083bc : bx      lr
End of assembler dump.
```

Uh-oh! The instructions referring the label addr_of_myvarX are different. Ok. Ignore that for now, we will learn in a future chapter what has happened. There is an arrow => pointing the instruction we are going to run (it has not been run yet). Before running it, let's inspect some registers.

```
(gdb) info registers r0 r1 r2 r3
r0             0x1      1
r1             0xbefff744       3204446020
r2             0xbefff74c       3204446028
r3             0x8390   33680
```

We can modify registers using p which means print but also evaluates side effects. For instance,

```
(gdb) p $r0 = 2
$1 = 2
(gdb) info registers r0 r1 r2 r3
r0             0x2      2
r1             0xbefff744       3204446020
r2             0xbefff74c       3204446028
r3             0x8390   33680
```

gdb has printed $1, this is the identifier of the result and we can use it when needed, so we can skip some typing. Not very useful now but it will be when we print a complicated expression.

```
(gdb) p $1
$2 = 2
```

Now we could use $2, and so on. Ok, time to run the first instruction.

```
(gdb) stepi
0x00008394 in main ()
```

Well, not much happened, let's use disassemble, again.

```
(gdb) disassemble
Dump of assembler code for function main:
   0x00008390 : ldr     r1, [pc, #40]   ; 0x83c0
=> 0x00008394 : mov     r3, #3
   0x00008398 : str     r3, [r1]
   0x0000839c : ldr     r2, [pc, #32]   ; 0x83c4
   0x000083a0 : mov     r3, #4
   0x000083a4 : str     r3, [r2]
   0x000083a8 : ldr     r1, [pc, #16]   ; 0x83c0
   0x000083ac : ldr     r1, [r1]
   0x000083b0 : ldr     r2, [pc, #12]   ; 0x83c4
   0x000083b4 : ldr     r2, [r2]
   0x000083b8 : add     r0, r1, r2
   0x000083bc : bx      lr
End of assembler dump.
```

Ok, let's see what happened in r1.

```
(gdb) info register r1
r1             0x10564  66916
```

Great, it has changed. In fact this is the address of myvar1. Let's check this using its symbolic name and C syntax.

```
(gdb) p &myvar1
$3 = ( *) 0x10564
```

Great! Can we see what is in this variable?

```
(gdb) p myvar1
```

```
$4 = 0
```

Perfect. This was as expected since in this example we set zero as the initial value of myvar1 and myvar2. Ok, next step.

```
(gdb) stepi
0x00008398 in main ()
(gdb) disas
Dump of assembler code for function main:
   0x00008390 : ldr     r1, [pc, #40]   ; 0x83c0
   0x00008394 : mov     r3, #3
=> 0x00008398 : str     r3, [r1]
   0x0000839c : ldr     r2, [pc, #32]   ; 0x83c4
   0x000083a0 : mov     r3, #4
   0x000083a4 : str     r3, [r2]
   0x000083a8 : ldr     r1, [pc, #16]   ; 0x83c0
   0x000083ac : ldr     r1, [r1]
   0x000083b0 : ldr     r2, [pc, #12]   ; 0x83c4
   0x000083b4 : ldr     r2, [r2]
   0x000083b8 : add     r0, r1, r2
   0x000083bc : bx      lr
End of assembler dump.
```

You can use disas (but not disa!) as a short for disassemble. Let’s check what happened to r3

```
(gdb) info registers r3
r3             0x3      3
```

So far so good. Another more step.

```
(gdb) stepi
0x0000839c in main ()
(gdb) disas
Dump of assembler code for function main:
   0x00008390 : ldr     r1, [pc, #40]   ; 0x83c0
   0x00008394 : mov     r3, #3
   0x00008398 : str     r3, [r1]
=> 0x0000839c : ldr     r2, [pc, #32]   ; 0x83c4
   0x000083a0 : mov     r3, #4
   0x000083a4 : str     r3, [r2]
```

```
   0x000083a8 : ldr     r1, [pc, #16]   ; 0x83c0
   0x000083ac : ldr     r1, [r1]
   0x000083b0 : ldr     r2, [pc, #12]   ; 0x83c4
   0x000083b4 : ldr     r2, [r2]
   0x000083b8 : add     r0, r1, r2
   0x000083bc : bx      lr
End of assembler dump.
```

Ok, lets see what happened, we stored r3, which contained a 3 into myvar1, right? Let’s check this.

```
(gdb) p myvar1
$5 = 3
```

Amazing, isn’t it? Ok. Now run until the end.

```
(gdb) continue
Continuing.
[Inferior 1 (process 3080) exited with code 07]
```

That’s all for today.

ARM assembler in Raspberry Pi – Chapter 5

January 19, 2013 Roger Ferrer Ibáñez, 24

Branching

Until now our small assembler programs execute one instruction after the other. If our ARM processor were only able to run this way it would be of limited use. It could not react to existing conditions which may require different sequences of instructions. This is the purpose of the branch instructions.

A special register

In chapter 2 we learnt that our Raspberry Pi ARM processor has 16 integer general purpose registers and we also said that some of them play special roles in our program. I deliberately ignored which registers were special as it was not relevant at that time.

But now it is relevant, at least for register r15. This register is very special, so special it has also another name: pc. It is unlikely that you see it used as r15 since it is confusing (although correct from the point of view of the ARM architecture). From now we will only use pc to name it.
What does pc stand for? pc means program counter. This name, the origins of which are in the dawn of computing, means little to nothing nowadays. In general the pc register (also called ip, instruction pointer, in other architectures like 386 or x86_64) contains the address of the next instruction going to be executed.
When the ARM processor executes an instruction, two things may happen at the end of its execution. If the instruction does not modify pc (and most instructions do not), pc is just incremented by 4 (like if we did add pc, pc, #4). Why 4? Because in ARM, instructions are 32 bit wide, so there are 4 bytes between every instruction. If the instruction modifies pc then the new value for pc is used.
Once the processor has fully executed an instruction then it uses the value in the pc as the address for the next instruction to execute. This way, an instruction that does not modify the pcwill be followed by the next contiguous instruction in memory (since it has been automatically increased by 4). This is called implicit sequencing of instructions: after one has run, usually the next one in memory runs. But if an instruction does modify the pc, for instance to a value

other than pc + 4, then we can be running another instruction of the program. This process of changing the value of pc is called branching. In ARM this done using branch instructions.

Unconditional branches

You can tell the processor to branch unconditionally by using the instruction b (for branch) and a label. Consider the following program.

```
/* -- branch01.s */
.text
.global main
main:
    mov r0, #2 /* r0 ← 2 */
    b end      /* branch to 'end' */
    mov r0, #3 /* r0 ← 3 */
end:
    bx lr
```

If you execute this program you will see that it returns an error code of 2.

```
$ ./branch01 ; echo $?
2
```

What happened is that instruction b end branched (modifying the pc) to the instruction at the label end, which is bx lr, the instruction we run at the end of our program. This way the instruction mov r0, #3 has not actually been run at all (the processor never reached that instruction).

At this point the unconditional branch instruction b may look a bit useless. It is not the case. In fact this instruction is essential in some contexts, in particular when linked with conditional branching. But before we can talk about conditional branching we need to talk about conditions.

Conditional branches

If our processor were only able to branch just because, it would not be very useful. It is much more useful to branch when some condition is met. So a processor should be able to evaluate some sort of conditions.

Before continuing, we need to unveil another register called cpsr (for Current Program Status Register). This register is a bit special and directly modifying it is out of the scope of this chapter. That said, it keeps some values that can be read

and updated when executing an instruction. The values of that register include four condition code flags called N (**n**egative), Z (**z**ero), C (**c**arry) and V (o**v**erflow). These four condition code flags are usually read by branch instructions. Arithmetic instructions and special testing and comparison instruction can update these condition codes too if requested.

The semantics of these four condition codes in instructions updating the cpsr are roughly the following

- N will be enabled if the result of the instruction yields a negative number. Disabled otherwise.
- Z will be enabled if the result of the instruction yields a zero value. Disabled if nonzero.
- C will be enabled if the result of the instruction yields a value that requires a 33rd bit to be fully represented. For instance an addition that overflows the 32 bit range of integers. There is a special case for C and subtractions where a non-borrowing subtraction enables it, disabled otherwise: subtracting a larger number to a smaller one enables C, but it will be disabled if the subtraction is done the other way round.
- V will be enabled if the result of the instruction yields a value that cannot be represented in 32 bits two's complement.

So we have all the needed pieces to perform branches conditionally. But first, let's start comparing two values. We use the instruction cmp for this purpose.

```
cmp r1, r2 /* updates cpsr doing "r1 - r2", but r1 and r2 are not modified
*/
```

This instruction subtracts to the value in the first register the value in the second register. Examples of what could happen in the snippet above?

- If r2 had a value (strictly) greater than r1 then N would be enabled because r1-r2 would yield a negative result.
- If r1 and r2 had the same value, then Z would be enabled because r1-r2 would be zero.
- If r1 was 1 and r2 was 0 then r1-r2 would not borrow, so in this case C would be enabled. If the values were swapped (r1 was 0 and r2 was 1) then C would be disabled because the subtraction does borrow.
- If r1 was 2147483648 (the largest positive integer in 32 bit two's complement) and r1 was -1 then r1-r2 would be 2147483649 but such number cannot be represented in 32 bit two's complement, so V would be enabled to signal this.

How can we use these flags to represent useful conditions for our programs?

•EQ (equal) When Z is enabled (Z is 1)
•NE (not equal). When Z is disabled. (Z is 0)
•GE (greater or equal than, in two's complement). When both V and N are enabled or disabled (V is N)
•LT (lower than, in two's complement). This is the opposite of GE, so when V and N are not both enabled or disabled (V is not N)
•GT (greather than, in two's complement). When Z is disabled and N and V are both enabled or disabled (Z is 0, N is V)
•LE (lower or equal than, in two's complement). When Z is enabled or if not that, N and V are both enabled or disabled (Z is 1. If Z is not 1 then N is V)
•MI (minus/negative) When N is enabled (N is 1)
•PL (plus/positive or zero) When N is disabled (N is 0)
•VS (overflow set) When V is enabled (V is 1)
•VC (overflow clear) When V is disabled (V is 0)
•HI (higher) When C is enabled and Z is disabled (C is 1 and Z is 0)
•LS (lower or same) When C is disabled or Z is enabled (C is 0 or Z is 1)
•CS/HS (carry set/higher or same) When C is enabled (C is 1)
•CC/LO (carry clear/lower) When C is disabled (C is 0)

These conditions can be combined to our b instruction to generate new instructions. This way, beq will branch only if Z is 1. If the condition of a conditional branch is not met, then the branch is ignored and the next instruction will be run. It is the programmer task to make sure that the condition codes are properly set prior a conditional branch.

```
1  /* -- compare01.s */
2  .text
3  .global main
4  main:
5      mov r1, #2        /* r1 ← 2 */
6      mov r2, #2        /* r2 ← 2 */
7      cmp r1, r2        /* update cpsr condition codes with the value of r1-r2
8  */
9      beq case_equal    /* branch to case_equal only if Z = 1 */
1
```

```
0
1
1 case_different :
1     mov r0, #2        /* r0 ← 2 */
2     b end             /* branch to end */
1 case_equal:
3     mov r0, #1        /* r0 ← 1 */
1 end:
4     bx lr
1
5
```

If you run this program it will return an error code of 1 because both r1 and r2 have the same value. Now change mov r1, #2 in line 5 to be mov r1, #3 and the returned error code should be 2. Note that case_different we do not want to run the case_equal instructions, thus we have to branch to end (otherwise the error code would always be 1).

That’s all for today.

ARM assembler in Raspberry Pi – Chapter 6

January 20, 2013 Roger Ferrer Ibáñez, 10

Control structures

In the previous chapter we learnt branch instructions. They are really powerful tools because they allow us to express control structures. Structured programming is an important milestone in better computing engineering (a foundational one, but nonetheless an important one). So being able to map usual structured programming constructs in assembler, in our processor, is a Good Thing™.

If, then, else

Well, this one is a basic one, and in fact we already used this structure in the previous chapter. Consider the following structure, where E is an expression and S1 and S2 are statements (they may be compound statements like { SA; SB; SC; })

```
if (E) then
   S1
else
   S2
```

A possible way to express this in ARM assembler could be the following

```
if_eval:
    /* Assembler that evaluates E and updates the cpsr accordingly
*/
bXX else /* Here XX is the appropiate condition */
then_part:
   /* assembler for S1, the "then" part */
   b end_of_if
else:
   /* assembler for S2, the "else" part */
end_of_if:
```

If there is no else part, we can replace bXX else with bXX end_of_if.

Loops

This is another usual one in structured programming. While there are several types of loops, actually all reduce to the following structure.

```
while (E)
  S
```

Supposedly S makes something so E eventually becomes false and the loop is left. Otherwise we would stay in the loop forever (sometimes this is what you want but not in our examples). A way to implement these loops is as follows.

```
while_condition : /* assembler to evaluate E and update cpsr */
  bXX end_of_loop  /* If E is false, then leave the loop right now
*/
  /* assembler of S */
  b while_condition /* Unconditional branch to the beginning */
end_of_loop:
```

A common loop involves iterating from a single range of integers, like in

```
for (i = L; i < N; i += K)
  S
```

But this is nothing but

```
  i = L;
  while (i < N)
  {
     S;
     i += K;
  }
```

So we do not have to learn a new way to implement the loop itself.

1 + 2 + 3 + 4 + ... + 22

As a first example lets sum all the numbers from 1 to 22 (I'll tell you later why I chose 22). The result of the sum is 253 (check it with a calculator). I know it makes little sense to compute something the result of which we know already, but this is just an example.

```
1
2
3
4  /* -- loop01.s */
5  .text
6  .global main
7  main:
8      mov r1, #0        /* r1 ← 0 */
9      mov r2, #1        /* r2 ← 1 */
1  loop:
0      cmp r2, #22       /* compare r2 and 22 */
1      bgt end           /* branch if r2 > 22 to end */
1      add r1, r1, r2    /* r1 ← r1 + r2 */
1      add r2, r2, #1    /* r2 ← r2 + 1 */
2      b loop
1  end:
3      mov r0, r1        /* r0 ← r1 */
1      bx lr
4
1
5
```

Here we are counting from 1 to 22. We will use the register r2 as the counter. As you can see in line 6 we initialize it to 1. The sum will be accumulated in the register r1, at the end of the program we move the contents of r1 into r0 to return the result of the sum as the error code of the program (we could have used r0 in all the code and avoid this final mov but I think it is clearer this way).

In line 8 we compare r2 (remember, the counter that will go from 1 to 22) to 22. This will update the cpsr thus in line 9 we can check if the comparison was such that r2 was greater than 22. If this is the case, we end the loop by branching to end. Otherwise we add the current value of r2to the current value of r1 (remember, in r1 we accumulate the sum from 1 to 22).

Line 11 is an important one. We increase the value of r2, because we are counting from 1 to 22 and we already added the current counter value in r2 to the result of the sum in r1. Then at line 12 we branch back at the beginning of the loop. Note that if line 11 was not there we would hang as the comparison in line 8 would always be false and we would never leave the loop in line 9!

```
$ ./loop01; echo $?
253
```

Well, now you could change the line 8 and try with let's say, #100. The result should be 5050.

```
$ ./loop01; echo $?
186
```

What happened? Well, it happens that in Linux the error code of a program is a number from 0 to 255 (8 bits). If the result is 5050, only the lower 8 bits of the number are used. 5050 in binary is 1001110111010, its lower 8 bits are 10111010 which is exactly 186. How can we check the computed r1 is 5050 before ending the program? Let's use GDB.

```
$ gdb loop
...
(gdb) start
Temporary breakpoint 1 at 0x8390
Starting program: /home/roger/asm/chapter06/loop01

Temporary breakpoint 1, 0x00008390 in main ()
(gdb) disas main,+(9*4)
Dump of assembler code from 0x8390 to 0x83b4:
   0x00008390 <main+0>: mov     r1, #0
   0x00008394 <main+4>: mov     r2, #1
   0x00008398 <loop+0>: cmp     r2, #100        ; 0x64
   0x0000839c <loop+4>: bgt     0x83ac <end>
   0x000083a0 <loop+8>: add     r1, r1, r2
   0x000083a4 <loop+12>:        add     r2, r2, #1
   0x000083a8 <loop+16>:        b       0x8398 <loop>
   0x000083ac <end+0>:  mov     r0, r1
   0x000083b0 <end+4>:  bx      lr
End of assembler dump.
```

Let's tell gdb to stop at 0x000083ac, right before executing mov r0, r1.

```
(gdb) break *0x000083ac
(gdb) cont
Continuing.

Breakpoint 2, 0x000083ac in end ()
(gdb) disas
Dump of assembler code for function end:
=> 0x000083ac <+0>:     mov     r0, r1
   0x000083b0 <+4>:     bx      lr
End of assembler dump.
(gdb) info register r1
r1             0x13ba   5050
```

Great, this is what we expected but we could not see due to limits in the error code.

Maybe you have noticed that something odd happens with our labels being identified as functions. We will address this issue in a future chapter, this is mostly harmless though.

3n + 1

Let's make another example a bit more complicated. This is the famous 3n + 1 problem also known as the Collatz conjecture. Given a number n we will divide it by 2 if it is even and multiply it by 3 and add one if it is odd.

```
if (n % 2 == 0)
  n = n / 2;
else
  n = 3*n + 1;
```

Before continuing, our ARM processor is able to multiply two numbers but we should learn a new instruction mul which would detour us a bit. Instead we will use the following identity 3 * n = 2*n + n. We do not really know how to multiply or divide by two yet, we will study this in a future chapter, so for now just assume it works as shown in the assembler below.

Collatz conjecture states that, for any number n, repeatedly applying this procedure will eventually give us the number 1. Theoretically it could happen that this is not the case. So far, no such number has been found, but it has not been proved otherwise. If we want to repeatedly apply the previous procedure, our program is doing something like this.

```
n = ...;
while (n != 1)
{
  if (n % 2 == 0)
     n = n / 2;
  else
     n = 3*n + 1;
}
```

If the Collatz conjecture were false, there would exist some n for which the code above would hang, never reaching 1. But as I said, no such number has been found.

```
/* -- collatz.s */
.text
.global main
main:
    mov r1, #123            /* r1 ← 123 */
    mov r2, #0              /* r2 ← 0 */
loop:
    cmp r1, #1              /* compare r1 and 1 */
    beq end                 /* branch to end if r1 == 1 */

    and r3, r1, #1          /* r3 ← r1 & 1 */
    cmp r3, #0              /* compare r3 and 0 */
    bne odd                 /* branch to odd if r3 != 0 */
even:
    mov r1, r1, ASR #1      /* r1 ← (r1 >> 1) */
    b end_loop
odd:
    add r1, r1, r1, LSL #1 /* r1 ← r1 + (r1 << 1) */
    add r1, r1, #1          /* r1 ← r1 + 1 */

end_loop:
    add r2, r2, #1          /* r2 ← r2 + 1 */
    b loop                  /* branch to loop */

end:
    mov r0, r2
```

```
7       bx lr
```

In r1 we will keep the number n. In this case we will use the number 123. 123 reaches 1 in 46 steps: [123, 370, 185, 556, 278, 139, 418, 209, 628, 314, 157, 472, 236, 118, 59, 178, 89, 268, 134, 67, 202, 101, 304, 152, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]. We will count the number of steps in register r2. So we initialize r1 with 123 and r2 with 0 (no step has been performed yet).

At the beginning of the loop, in lines 8 and 9, we check if r1 is 1. So we compare it with 1 and if it is equal we leave the loop branching to end.

Now we know that r1 is not 1, so we proceed to check if it is even or odd. To do this we use a new instruction and which performs a bitwise and operation. An even number will have the least significant bit (LSB) to 0, while an odd number will have the LSB to 1. So a bitwise and using 1 will return 0 or 1 on even or odd numbers, respectively. In line 11 we keep the result of the bitwise and in r3 register and then, in line 12, we compare it against 0. If it is not zero then we branch to odd, otherwise we continue on the even case.

Now some magic happens in line 15. This is a combined operation that ARM allows us to do. This is a mov but we do not move the value of r1 directly to r1 (which would be doing nothing) but first we do an arithmetic shift right (ASR) to the value of r1 (to the value, no the register itself). Then this shifted value is moved to the register r1. An arithmetic shift right shifts all the bits of a register to the right: the rightmost bit is effectively discarded and the leftmost is set to the same value as the leftmost bit prior the shift. Shifting right one bit to a number is the same as dividing that number by 2. So this mov r1, r1, ASR #1 is actually doing r1 ← r1 / 2.

Some similar magic happens for the even case in line 18. In this case we are doing an add. The first and second operands must be registers (destination operand and the first source operand). The third is combined with a logical shift left (LSL). The value of the operand is shifted left 1 bit: the leftmost bit is discarded and the rightmost bit is set to 0. This is effectively multiplying the value by 2. So we are adding r1 (which keeps the value of n) to 2*r1. This is 3*r1, so 3*n. We keep this value in r1 again. In line 19 we add 1 to that value, so r1 ends having the value 3*n+1 that we wanted.

Do not worry very much now about these LSL and ASR. Just take them for granted now. In a future chapter we will see them in more detail.

Finally, at the end of the loop, in line 22 we update r2 (remember it keeps the counter of our steps) and then we branch back to the beginning of the loop. Before ending the program we move the counter to r0 so we return the number of steps we did to reach 1.

```
$ ./collatz; echo $?
46
```

Great.

That’s all for today.

Postscript

Kevin Millikin rightly pointed (in a comment below) that usually a loop is not implemented in the way shown above. In fact Kevin says that a better way to do the loop of loop01.s is as follows.

```
/* -- loop02.s */
.text
.global main
main:
    mov r1, #0        /* r1 ← 0 */
    mov r2, #1        /* r2 ← 1 */
    b check_loop      /* unconditionally jump at the end of the loop */
loop:
    add r1, r1, r2    /* r1 ← r1 + r2 */
    add r2, r2, #1    /* r2 ← r2 + 1 */
check_loop:
    cmp r2, #22       /* compare r2 and 22 */
    ble loop          /* branch if r2 <= 22 to the beginning of the loop
*/
end:
    mov r0, r1        /* r0 ← r1 */
    bx lr
```

If you count the number of instruction in the two codes, there are 9 instructions in both. But if you look carefully in Kevin’s proposal you will see that by unconditionally branching to the end of the loop, and reversing the condition check, we can skip one branch thus reducing the number of instructions of the loop itself from 5 to 4.

There is another advantage in this second version, though: there is only one branch in the loop itself as we resort to implicit sequencing to reach again the two instructions performing the check. For reasons beyond the scope of this post, the execution of a branch instruction may negatively affect the performance of our programs. Processors have mechanisms to mitigate the performance loss due to branches (and in fact the processor in the Raspberry Pi does have them). But avoiding a branch instruction entirely avoids the potential performance penalization of executing a branch instruction.
While we do not care very much now about the performance of our assembler. However, I thought it was worth developing a bit more Kevin’s comment.

ARM assembler in Raspberry Pi – Chapter 7

January 26, 2013 Roger Ferrer Ibáñez, 8

ARM architecture has been for long targeted at embedded systems. Embedded systems usually end being used in massively manufactured products (dishwashers, mobile phones, TV sets, etc). In this context margins are very tight so a designer will always try to spare as much components as possible (a cent saved in hundreds of thousands or even millions of appliances may pay off). One relatively expensive component is memory although every day memory is less and less expensive. Anyway, in constrained memory environments being able to save memory is good and ARM instruction set was designed with this goal in mind. It will take us several chapters to learn all of these techniques, today we will start with one feature usually named shifted operand.

Indexing modes

We have seen that, except for load (ldr), store (str) and branches (b and bXX), ARM instructions take as operands either registers or immediate values. We have also seen that the first operand is usually the destination register (being str a notable exception as there it plays the role of source because the destination is now the memory). Instruction mov has another operand, a register or an immediate value. Arithmetic instructions like add and and (and many others) have two more source registers, the first of which is always a register and the second can be a register or an immediate value.

These sets of allowed operands in instructions are collectively called indexing modes. Today this concept will look a bit off since we will not index anything. The name indexing makes sense in memory operands but ARM instructions, except load and store, do not have memory operands. This is the nomenclature you will find in ARM documentation so it seems sensible to use theirs.

We can summarize the syntax of most of the ARM instructions in the following pattern

instruction Rdest, Rsource1, source2

There are some exceptions, mainly move (mov), branches, load and stores. In fact move is not so different actually.

mov Rdest, source2

Both Rdest and Rsource1 must be registers. In the next section we will talk about source2.

We will discuss the indexing modes of load and store instructions in a future chapter. Branches, on the other hand, are surprisingly simple and their single operand is just a label of our program, so there is little to discuss on indexing modes for branches.

Shifted operand

What is this mysterious source2 in the instruction patterns above? If you recall the previous chapters we have used registers or immediate values. So at least that source2 is this: register or immediate value. You can use an immediate or a register where a source2 is expected. Some examples follow, but we have already used them in the examples of previous chapters.

```
mov r0, #1
mov r1, r0
add r2, r1, r0
add r2, r3, #4
```

But source2 can be much more than just a simple register or an immediate. In fact, when it is a register we can combine it with a shift operation. We already saw one of these shift operations in chapter 6. Not it is time to unveil all of them.

- LSL #n

  Logical Shift Left. Shifts bits n times left. The n leftmost bits are lost and the n rightmost are set to zero.
- LSL Rsource3

  Like the previous one but instead of an immediate the lower byte of a register specifies the amount of shifting.
- LSR #n

  Logical Shift Right. Shifts bits n times right. The n rightmost bits are lost and the n leftmost bits are set to zero,
- LSR Rsource3

  Like the previous one but instead of an immediate the lower byte of a register specifies the amount of shifting.
- ASR #n

  Arithmetic Shift Right. Like LSR but the leftmost bit before shifting is used instead of zero in the n leftmost ones.
- ASR Rsource3

  Like the previous one but using a the lower byte of a register instead of an immediate.

•ROR #n

Rotate Right. Like LSR but the n rightmost bits are not lost but pushed onto the n leftmost bits

•ROR Rsource3

Like the previous one but using a the lower byte of a register instead of an immediate.

In the listing above, n is an immediate from 1 to 31. These extra operations may be applied to the value in the second source register (to the value, not to the register itself) so we can perform some more operations in a single instruction. For instance, ARM does not have any shift right or left instruction. You just use the mov instruction.

```
mov r1, r2, LSL #1
```

You may be wondering why one would want to shift left or right the value of a register. If you recall chapter 6 we saw that shifting left (LSL) a value gives a value that the same as multiplying it by 2. Conversely, shifting it right (ASR if we use two's complement, LSR otherwise) is the same as dividing by 2. Since a shift of n is the same as doing n shifts of 1, shifts actually multiply or divide a value by 2n.

```
mov r1, r2, LSL #1      /* r1 ← (r2*2) */
mov r1, r2, LSL #2      /* r1 ← (r2*4) */
mov r1, r3, ASR #3      /* r1 ← (r3/8) */
mov r3, #4
mov r1, r2, LSL r3      /* r1 ← (r2*16) */
```

We can combine it with add to get some useful cases.

```
add r1, r2, r2, LSL #1    /* r1 ← r2 + (r2*2) equivalent to r1 ←
r2*3 */
add r1, r2, r2, LSL #2    /* r1 ← r2 + (r2*4) equivalent to r1 ←
r2*5 */
```

You can do something similar with sub.

```
sub r1, r2, r2, LSL #3   /* r1 ← r2 - (r2*8) equivalent to r1 ←
r2*(-7)
```

ARM comes with a handy rsb (Reverse Substract) instruction which computes Rdest ← source2 - Rsource1 (compare it to sub which computes Rdest ← Rsource1 - source2).

```
rsb r1, r2, r2, LSL #3       /* r1 ← (r2*8) - r2 equivalent to r1 ←
r2*7 */
```

Another example, a bit more contrived.

```
/* Complicated way to multiply the initial value of r1 by 42 = 7*3*2
*/
rsb r1, r1, r1, LSL #3  /* r1 ← (r1*8) - r1 equivalent to r1 ← 7*r1
*/
add r1, r1, r1, LSL #1  /* r1 ← r1 + (2*r1) equivalent to r1 ← 3*r1
*/
add r1, r1, r1          /* r1 ← r1 + r1     equivalent to r1 ← 2*r1
*/
```

You are probably wondering why would we want to use shifts to perform multiplications. Well, the generic multiplication instruction always work but it is usually much harder to compute by our ARM processor so it may take more time. There are times where there is no other option but for many small constant values a single instruction may be more efficient.

Rotations are less useful than shifts in everyday use. They are usually used in cryptography, to reorder bits and "scramble" them. ARM does not provide a way to rotate left but we can do a nrotate left doing a 32-n rotate right.

```
/* Assume r1 is 0x12345678 */
mov r1, r1, ROR #1   /* r1 ← r1 ror 1. This is r1 ← 0x91a2b3c
*/
mov r1, r1, ROR #31  /* r1 ← r1 ror 31. This is r1 ← 0x12345678
*/
```

That's all for today.

ARM assembler in Raspberry Pi – Chapter 8

January 27, 2013 Roger Ferrer Ibáñez, 17

In the previous chapter we saw that the second operand of most arithmetic instructions can use a shift operator which allows us to shift and rotate bits. In this chapter we will continue learning the available indexing modes of ARM instructions. This time we will focus on load and store instructions.

Arrays and structures

So far we have been able to move 32 bits from memory to registers (load) and back to memory (store). But working on single items of 32 bits (usually called scalars) is a bit limiting. Soon we would find ourselves working on arrays and structures, even if we did not know.

An array is a sequence of items of the same kind in memory. Arrays are a foundational data structure in almost every low level language. Every array has a base address, usually denoted by the name of the array, and contains N items. Each of these items has associated a growing index, ranging from 0 to N-1 or 1 to N. Using the base address and the index we can access an item of the array. We mentioned in chapter 3 that memory could be viewed as an array of bytes. An array in memory is the same, but an item may take more than one single byte.

A structure (or record or tuple) is a sequence of items of possibly diferent kind. Each item of a structure is usually called a field. Fields do not have an associated index but an offset respect to the beginning of the structure. Structures are laid out in memory to ensure that the proper alignment is used in every field. The base address of a structure is the address of its first field. If the base address is aligned, the structure should be laid out in a way that all the field are properly aligned as well.

What do arrays and structure have to do with indexing modes of load and store? Well, these indexing modes are designed to make easier accessing arrays and structs.

Defining arrays and structs

To illustrate how to work with arrays and references we will use the following C declarations and implement them in assembler.

```
int a[100];
struct my_struct
{
  char f0;
  int f1;
} b;
```

Let's first define in our assembler the array 'a'. It is just 100 integers. An integer in ARM is 32-bit wide so in our assembler code we have to make room for 400 bytes (4 * 100).

```
1 /* -- array01.s */
2 .data
3
4 .balign 4
5 a: .skip 400
```

In line 5 we define the symbol a and then we make room for 400 bytes. The directive .skip tells the assembler to advance a given number of bytes before emitting the next datum. Here we are skipping 400 bytes because our array of integers takes 400 bytes (4 bytes per each of the 100 integers). Declaring a structure is not much different.

```
7 .balign 4
8 b: .skip 8
```

Right now you should wonder why we skipped 8 bytes when the structure itself takes just 5 bytes. Well, it does need 5 bytes to store useful information. The first field f0 is a char. A char takes 1 byte of storage. The next field f1 is an integer. An integer takes 4 bytes and it must be aligned at 4 bytes as well, so we have to leave 3 unused bytes between the field f0 and the field f1. This unused storage put just to fulfill alignment is called padding. Padding should never be used by your program.

Naive approach without indexing modes

Ok, let's write some code to initialize every item of the array a[i]. We will do something equivalent to the following C code.

```
for (i = 0; i < 100; i++)
  a[i] = i;
```

```
1 .text
```

```
0
1
1
1
2
1
3
1
4
1
5
1
6
1
7
1
8
1
9
2
0
2
1
2
2
2
2
3
2
4
2
5
```

```
.global main
main:
    ldr r1, addr_of_a       /* r1 ← &a */
    mov r2, #0              /* r2 ← 0 */
loop:
    cmp r2, #100            /* Have we reached 100 yet? */
    beq end                 /* If so, leave the loop, otherwise continue
*/
    add r3, r1, r2, LSL #2  /* r3 ← r1 + (r2*4) */
    str r2, [r3]            /* *r3 ← r2 */
    add r2, r2, #1          /* r2 ← r2 + 1 */
    b loop                  /* Go to the beginning of the loop */
end:
    bx lr
addr_of_a: .word a
```

Whew! We are using lots of things we have learnt from earlier chapters. In line 14 we load the base address of the array into r1. The address of the array will not change so we load it once. In register r2 we will keep the index that will range from 0 to 99. In line 17 we compare it to 100 to see if we have reached the end of the loop.

Line 19 is an important one. Here we compute the address of the item. We have in r1 the base address and we know each item is 4 bytes wide. We know also that r2 keeps the index of the loop which we will use to access the array

element. Given an item with index i its address must be &a + 4*i, since there are 4 bytes between every element of this array. So r3 has the address of the current element in this step of the loop. In line 20 we store r2, this is i, into the memory pointed by r3, the i-th array item, this is a[i].
Then we proceed to increase r2 and jump back for the next step of the loop.
As you can see, accessing an array involves calculating the address of the accessed item. Does the ARM instruction set provide a more compact way to do this? The answer is yes. In fact it provides several indexing modes.

Indexing modes

In the previous chapter the concept indexing mode was a bit off because we were not indexing anything. Now it makes much more sense since we are indexing an array item. ARM provides nineof these indexing modes. I will distinguish two kinds of indexing modes: non updating and updating depending on whether they feature a side-effect that we will discuss later, when dealing with updating indexing modes.

Non updating indexing modes

1.[Rsource1, +#immediate] or [Rsource1, -#immediate]

It justs adds (or subtracts) the immediate value to form the address. This is very useful to array items the index of which is a constant in the code or fields of a structure, since their offset is always constant. In Rsource1 we put the base address and in immediate the offset we want in bytes. The immediate cannot be larger than 12 bits (0..4096). When the immediate is #0 it is like the usual we have been using [Rsource1].

For example, we can set a[3] to 3 this way (we assume that r1 already contans the base address of a). Note that the offset is in bytes thus we need an offset of 12 (4 bytes * 3 items skipped).

```
mov r2, #3          /* r2 ← 3 */
str r2, [r1, #+12]  /* *(r1 + 12) ← r2 */
```

2.[Rsource1, +Rsource2] or [Rsource1, -Rsource2]

This is like the previous one, but the added (or subtracted) offset is the value in a register. This is useful when the offset is too big for the immediate. Note that for the +Rsource2case, the two registers can be swapped (as this would not affect the address computed).

Example. The same as above but using a register this time.

```
mov r2, #3          /* r2 ← 3 */
mov r3, #12         /* r3 ← 12 */
str r2, [r1,+r3]    /* *(r1 + r3) ← r2 */
```

3.[Rsource1, +Rsource2, shift_operation #immediate] or [Rsource1, -Rsource2, shift_operation #immediate].
This one is similar to the usual shift operation we can do with other instructions. A shift operation (remember: LSL, LSR, ASR or ROR) is applied to Rsource2, Rsource1 is then added (or subtracted) to the result of the shift operation applied to Rsource2. This is useful when we need to multiply the address by some fixed amount. When accessing the items of the integer array a we had to multiply the result by 4 to get a meaningful address. For this example, let's first recall how we computed above the address in the array of the item in position r2.

```
1
9  add r3, r1, r2, LSL #2  /* r3 ← r1 + r2*4 */
2  str r2, [r3]            /* *r3 ← r2 */
0
```

We can express this in a much more compact way (without the need of the register r3).

```
str r2, [r1, +r2, LSL #2]  /* *(r1 + r2*4) ← r2 */
```

Updating indexing modes

In these indexing modes the Rsource1 register is updated with the address synthesized by the load or store instruction. You may be wondering why one would want to do this. A bit of detour first. Recheck the code of the array load. Why do we have to keep around the base address of the array if we are always effectively moving 4 bytes away from it? Would not it make much more sense to keep the address of the current entity? So instead of

```
1
9  add r3, r1, r2, LSL #2  /* r3 ← r1 + r2*4 */
2  str r2, [r3]            /* *r3 ← r2 */
0
```

we might want to do something like

```
str r2, [r1]        /* *r1 ← r2 */
add r1, r1, #4      /* r1 ← r1 + 4 */
```

because there is no need to compute everytime from the beginning the address of the next item (as we are accessing them sequentially). Even if this looks slightly better, it still can be improved a bit more. What if our instruction were able to update r1 for us? Something like this (obviously the exact syntax is not as shown)

```
/* Wrong syntax */
str r2, [r1] "and then" add r1, r1, #4
```

Such indexing modes exist. There are two kinds of updating indexing modes depending on at which time Rsource1 is updated. If Rsource1 is updated after the load or store itself (meaning that the address to load or store is the initial Rsource1 value) this is a post-indexing accessing mode. If Rsource1 is updated before the actual load or store (meaning that the address to load or store is the final value of Rsource1) this is a pre-indexing accessing mode. In all cases, at the end of the instruction Rsource1 will have the value of the computation of the indexing mode. Now this sounds a bit convoluted, just look in the example above: we first load using r1 and then we do r1 ← r1 + 4. This is post-indexing: we first use the value of r1 as the address where we store the value of r2. Then r1 is updated with r1 + 4. Now consider another hypothetic syntax.

```
/* Wrong syntax */
str r2, [add r1, r1, #4]
```

This is pre-indexing: we first compute r1 + 4 and use it as the address where we store the value of r2. At the end of the instruction r1 has effectively been updated too, but the updated value has already been used as the address of the load or store.

Post-indexing modes

4. [Rsource1], #+immediate or [Rsource1], #-immediate

   The value of Rsource1 is used as the address for the load or store. Then Rsource1 is updated with the value of immediate after adding (or subtracting) it to Rsource1. Using this indexing mode we can rewrite the loop of our first example as follows:

```
loop:
    cmp r2, #100              /* Have we reached 100 yet? */
    beq end                    /* If so, leave the loop, otherwise continue
*/
    str r2, [r1], #4          /* *r1 ← r2 then r1 ← r1 + 4 */
    add r2, r2, #1            /* r2 ← r2 + 1 */
    b loop                    /* Go to the beginning of the loop */
```

```
92
02
21
22
end:
```

5.[Rsource1], +Rsource2 or [Rsource1], -Rsource2

Like the previous one but instead of an immediate, the value of Rsource2 is used. As usual this can be used as a workaround when the offset is too big for the immediate value.

6.[Rsource1], +Rsource2, shift_operation #immediate or [Rsource1], -Rsource2, shift_operation #immediate

The value of Rsource1 is used as the address for the load or store. Then Rsource2 is applied a shift operation (LSL, LSR, ASR or ROL). The resulting value of that shift is added (or subtracted) to Rsource1. Rsource1 is finally updated with this last value.

Pre-indexing modes

Pre-indexing modes may look a bit weird at first but they are useful when the computed address is going to be reused soon. Instead of recomputing it we can reuse the updated Rsource1. Mind the ! symbol in these indexing modes which distinguishes them from the non updating indexing modes.

7.[Rsource1, #+immediate]! or [Rsource1, #-immediate]!

It behaves like the similar non-updating indexing mode but Rsource1 gets updated with the computed address. Imagine we want to compute a[3] = a[3] + a[3]. We could do this (we assume that r1 already has the base address of the array).

```
ldr r2, [r1, #+12]!  /* r1 ← r1 + 12 then r2 ← *r1 */
add r2, r2, r2       /* r2 ← r2 + r2 */
str r2, [r1]         /* *r1 ← r2 */
```

8.[Rsource1, +Rsource2]! or [Rsource1, +Rsource2]!

Similar to the previous one but using a register Rsource2 instead of an immediate.

9.[Rsource1, +Rsource2, shift_operation #immediate]! or [Rsource1, -Rsource2, shift_operation #immediate]!

Like to the non-indexing equivalent but Rsource1 will be updated with the address used for the load or store instruction.

Back to structures

All the examples in this chapter have used an array. Structures are a bit simpler: the offset to the fields is always constant: once we have the base address of the structure (the address of the first field) accessing a field is just an indexing mode with an offset (usually an immediate). Our current structure features, on purpose, a char as its first field f0. Currently we cannot work on scalars in memory of different size than 4 bytes. So we will postpone working on that first field for a future chapter.
For instance imagine we wanted to increment the field f1 like this.

```
b.f1 = b.f1 +
7;
```

If r1 contains the base address of our structure, accessing the field f1 is pretty easy now that we know all the available indexing modes.

```
1 ldr r2, [r1, #+4]!  /* r1 ← r1 + 4 then r2 ← *r1 */
2 add r2, r2, #7      /* r2 ← r2 + 7 */
3 str r2, [r1]        /* *r1 ← r2 */
```

Note that we use a pre-indexing mode to keep in r1 the address of the field f1. This way the second store does not need to compute that address again.

That’s all for today.

ARM assembler in Raspberry Pi – Chapter 9

February 2, 2013 Roger Ferrer Ibáñez, 18

In previous chapters we learnt the foundations of ARM assembler: registers, some arithmetic operations, loads and stores and branches. Now it is time to put everything together and add another level of abstraction to our assembler skills: functions.

Why functions?

Functions are a way to reuse code. If we have some code that will be needed more than once, being able to reuse it is a Good Thing™. This way, we only have to ensure that the code being reused is correct. If we repeated the code whe should verify it is correct at every point. This clearly does not scale. Functions can also get parameters. This way not only we reuse code but we can use it in several ways, by passing different parameters. All this magic, though, comes at some price. A function must be a a well-behaved citizen.

Do’s and don’ts of a function

Assembler gives us a lot of power. But with a lot of power also comes a lot of responsibility. We can break lots of things in assembler, because we are at a very low level. An error and nasty things may happen. In order to make all functions behave in the same way, there are conventions in every environment that dictate how a function must behave. Since we are in a Raspberry Pi running Linux we will use the (chances are that other ARM operating systems like RISCOS or Windows RT follow it). You may find this document in the ARM documentation website but I will try to summarize it in this chapter.

New special named registers

When discussing branches we learnt that r15 was also called pc but we never called it r15anymore. Well, let’s rename from now r14 as lr and r13 as sp. lr stands for link register and it is the address of the instruction following the instruction that called us (we will see later what is this). sp stands for stack pointer. The stack is an area of memory owned only by the current function, the sp register stores the top address of that stack. For now, let’s put the stack aside. We will get it back in the next chapter.

Passing parameters

Functions can receive parameters. The first 4 parameters must be stored, sequentially, in the registers r0, r1, r2 and r3. You may be wondering how to pass more than 4 parameters. We can, of course, but we need to use the stack, but we will discuss it in the next chapter. Until then, we will only pass up to 4 parameters.

Well behaved functions

A function must adhere, at least, to the following rules if we want it to be AAPCS compliant.

- A function should not make any assumption on the contents of the cpsr. So, at the entry of a function condition codes N, Z, C and V are unknown.
- A function can freely modify registers r0, r1, r2 and r3.
- A function cannot assume anything on the contents of r0, r1, r2 and r3 unless they are playing the role of a parameter.
- A function can freely modify lr but the value upon entering the function will be needed when leaving the function (so such value must be kept somewhere).
- A function can modify all the remaining registers as long as their values are restored upon leaving the function. This includes sp and registers r4 to r11.

  This means that, after calling a function, we have to assume that (only) registers r0, r1, r2, r3 and lr have been overwritten.

Calling a function

There are two ways to call a function. If the function is statically known (meaning we know exactly which function must be called) we will use bl label. That label must be a label defined in the .text section. This is called a direct (or immediate) call. We may do indirect calls by first storing the address of the function into a register and then using blx Rsource1.

In both cases the behaviour is as follows: the address of the function (immediately encoded in the bl or using the value of the register in blx) is stored in pc. The address of the instruction following the bl or blx instruction is kept in lr.

Leaving a function

A well behaved function, as stated above, will have to keep the initial value of lr somewhere. When leaving the function, we will retrieve that value and put it in some register (it can be lragain but this is not mandatory). Then we will bx Rsource1 (we could use blx as well but the latter would update lr which is useless here).

Returning data from functions

Functions must use r0 for data that fits in 32 bit (or less). This is, C types char, short, int, long (and float though we have not seen floating point yet) will be returned in r0. For basic types of 64 bit, like C types long long and double, they will be returned in r1 and r0. Any other data is returned through the stack unless it is 32 bit or less, where it will be returned in r0.

In the examples in previous chapters we returned the error code of the program in r0. This now makes sense. C's main returns an int, which is used as the value of the error code of our program.

Hello world

Usually this is the first program you write in any high level programming language. In our case we had to learn lots of things first. Anyway, here it is. A "Hello world" in ARM assembler.

(Note to experts: since we will not discuss the stack until the next chapter, this code may look very dumb to you)

```
1  /* -- hello01.s */
2  .data
3
4  greeting:
5   .asciz "Hello world"
6
7  .balign 4
8  return: .word 0
9
1  .text
0
1  .global main
1  main:
1      ldr r1, address_of_return     /*   r1 ← &address_of_return */
2      str lr, [r1]                  /*   *r1 ← lr */
1
3      ldr r0, address_of_greeting   /* r0 ← &address_of_greeting */
1                                  /* First parameter of puts */
4
1      bl puts                       /* Call to puts */
```

```
                                /* lr ← address of next instruction */

    ldr r1, address_of_return     /* r1 ← &address_of_return */
    ldr lr, [r1]                  /* lr ← *r1 */
    bx lr                         /* return from main */
address_of_greeting: .word greeting
address_of_return: .word return

/* External */
.global puts
```

We are going to call puts function. This function is defined in the C library and has the following prototype int puts(const char*). It receives, as a first parameter, the address of a C-string (this is, a sequence of bytes where no byte but the last is zero). When executed it outputs that string to stdout (so it should appear by default to our terminal). Finally it returns the number of bytes written.

We start by defining in the .data the label greeting in lines 4 and 5. This label will contain the address of our greeting message. GNU as provides a convenient .asciz directive for that purpose. This directive emits as bytes as needed to represent the string plus the final zero byte. We could have used another directive .ascii as long as we explicitly added the final zero byte.
After the bytes of the greeting message, we make sure the next label will be 4 bytes aligned and we define a return label in line 8. In that label we will keep the value of lr that we have in main. As stated above, this is a requirement for a well behaved function: be able to get the original value of lr upon entering. So we make some room for it.
The first two instructions, lines 14 an 15, of our main function keep the value of lr in that return variable defined above. Then in line 17 we prepare the arguments for the call to puts. We load the address of the greeting message into r0 register. This register will hold the first (the only one actually) parameter of puts. Then in line 20 we call the function. Recall that bl will set in lr the address of the instruction following it (this is the instruction in line 23). This is the reason why we copied the value of lr in a variable in the beginning of the main function, because it was going to be overwritten by bl.
Ok, puts runs and the message is printed on the stdout. Time to get the initial value of lr so we can return successfully from main. Then we return.
Is our main function well behaved? Yes, it keeps and gets back lr to leave. It only modifies r0and r1. We can assume that puts is well behaved as well, so everything should work fine. Plus the bonus of seeing how many bytes have been written to the output.

```
$ ./hello01
Hello world
$ echo $?
12
```

Note that “Hello world” is just 11 bytes (the final zero is not counted as it just plays the role of a finishing byte) but the program returns 12. This is because puts always adds a newline byte, which accounts for that extra byte.

Real interaction!

Now we have the power of calling functions we can glue them together. Let’s call printf and scanf to read a number and then print it back to the standard output.

```
/* -- printf01.s */
.data

/* First message */
.balign 4
message1: .asciz "Hey, type a number: "

/* Second message */
.balign 4
message2: .asciz "I read the number %d\n"

/* Format pattern for scanf */
.balign 4
scan_pattern : .asciz "%d"

/* Where scanf will store the number read */
.balign 4
number_read: .word 0

.balign 4
return: .word 0

.text

.global main
main:
    ldr r1, address_of_return        /* r1 ← &address_of_return */
    str lr, [r1]                     /* *r1 ← lr */

    ldr r0, address_of_message1      /* r0 ← &message1 */
    bl printf                        /* call to printf */

    ldr r0, address_of_scan_pattern  /* r0 ← &scan_pattern */
    ldr r1, address_of_number_read   /* r1 ← &number_read */
    bl scanf                         /* call to scanf */

    ldr r0, address_of_message2      /* r0 ← &message2 */
    ldr r1, address_of_number_read   /* r1 ← &number_read */
```

```
    ldr r1, [r1]                    /* r1 ← *r1 */
    bl printf                       /* call to printf */

    ldr r0, address_of_number_read  /* r0 ← &number_read */
    ldr r0, [r0]                    /* r0 ← *r0 */

    ldr lr, address_of_return       /* lr ← &address_of_return */
    ldr lr, [lr]                    /* lr ← *lr */
    bx lr                           /* return from main using lr */
address_of_message1 : .word message1
address_of_message2 : .word message2
address_of_scan_pattern : .word scan_pattern
address_of_number_read : .word number_read
address_of_return : .word return

/* External */
.global printf
.global scanf
```

In this example we will ask the user to type a number and then we will print it back. We also return the number in the error code, so we can check twice if everything goes as expected. For the error code check, make sure your number is lower than 255 (otherwise the error code will show only its lower 8 bits).

```
$ ./printf01
Hey, type a number: 123↴
I read the number 123
$ ./printf01 ; echo $?
Hey, type a number: 124↴
I read the number 124
124
```

Our first function

Let’s define our first function. Lets extend the previous example but multiply the number by 5.

```
.balign 4
return2: .word 0

.text

/*
mult_by_5 function
*/
mult_by_5:
    ldr r1, address_of_return2       /* r1 ← &address_of_return */
    str lr, [r1]                     /* *r1 ← lr */

    add r0, r0, r0, LSL #2           /* r0 ← r0 + 4*r0 */

    ldr lr, address_of_return2       /* lr ← &address_of_return */
    ldr lr, [lr]                     /* lr ← *lr */
    bx lr                            /* return from main using lr */
address_of_return2 : .word return2
```

```
40
```

This function will need another “return” variable like the one main uses. But this is for the sake of the example. Actually this function does not call another function. When this happens it does not need to keep lr as no bl or blx instruction is going to modify it. If the function wanted to use lr as the the r14 general purpose register, the process of keeping the value would still be mandatory.
As you can see, once the function has computed the value, it is enough keeping it in r0. In this case it was pretty easy and a single instruction was enough.
The whole example follows.

```
/* -- printf02.s */
.data

/* First message */
.balign 4
message1: .asciz "Hey, type a number: "

/* Second message */
.balign 4
message2: .asciz "%d times 5 is %d\n"

/* Format pattern for scanf */
.balign 4
scan_pattern : .asciz "%d"

/* Where scanf will store the number read */
.balign 4
number_read: .word 0

.balign 4
return: .word 0

.balign 4
return2: .word 0

.text
```

```
/*
mult_by_5 function
*/
mult_by_5:
    ldr r1, address_of_return2       /* r1 ← &address_of_return */
    str lr, [r1]                     /* *r1 ← lr */

    add r0, r0, r0, LSL #2           /* r0 ← r0 + 4*r0 */

    ldr lr, address_of_return2       /* lr ← &address_of_return */
    ldr lr, [lr]                     /* lr ← *lr */
    bx lr                            /* return from main using lr */
address_of_return2 : .word return2

.global main
main:
    ldr r1, address_of_return        /* r1 ← &address_of_return */
    str lr, [r1]                     /* *r1 ← lr */

    ldr r0, address_of_message1      /* r0 ← &message1 */
    bl printf                        /* call to printf */

    ldr r0, address_of_scan_pattern  /* r0 ← &scan_pattern */
    ldr r1, address_of_number_read   /* r1 ← &number_read */
    bl scanf                         /* call to scanf */

    ldr r0, address_of_number_read   /* r0 ← &number_read */
    ldr r0, [r0]                     /* r0 ← *r0 */
    bl mult_by_5

    mov r2, r0                       /* r2 ← r0 */
    ldr r1, address_of_number_read   /* r1 ← &number_read */
    ldr r1, [r1]                     /* r1 ← *r1 */
    ldr r0, address_of_message2      /* r0 ← &message2 */
    bl printf                        /* call to printf */

    ldr lr, address_of_return        /* lr ← &address_of_return */
```

```
    ldr lr, [lr]                    /* lr ← *lr */
    bx lr                           /* return from main using lr */
address_of_message1 : .word message1
address_of_message2 : .word message2
address_of_scan_pattern : .word scan_pattern
address_of_number_read : .word number_read
address_of_return : .word return

/* External */
.global printf
.global scanf
```

I want you to notice lines 58 to 62. There we prepare the call to printf which receives three parameters: the format and the two integers referenced in the format. We want the first integer be the number entered by the user. The second one will be that same number multiplied by 5. After the call to mult_by_5, r0 contains the number entered by the user multiplied by 5. We want it to be the third parameter so we move it to r2. Then we load the value of the number entered by the user into r1. Finally we load in r0 the address to the format message of printf. Note that here the order of preparing the arguments of a call is nonrelevant as long as the values are correct at the point of the call. We use the fact that we will have to overwrite r0, so for convenience we first copy r0 to r2.

```
$ ./printf02
Hey, type a number: 1234↴
1234 times 5 is 6170
```

That’s all for today.

ARM assembler in Raspberry Pi – Chapter 10

February 7, 2013 Roger Ferrer Ibáñez, 17

In chapter 9 we were introduced to functions and we saw that they have to follow a number of conventions in order to play nice with other functions. We also briefly mentioned the stack, as an area of memory owned solely by the function. In this chapter we will go in depth with the stack and why it is important for functions.

Dynamic activation

One of the benefits of functions is being able to call them more than once. But that more than once hides a small trap. We are not restricting who will be able to call the function, so it might happen that it is the same function who calls itself. This happens when we use recursion.

A typical example of recursion is the factorial of a number n, usually written as n!. A factorial in C can be written as follows.

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

Note that there is only one function factorial, but it may be called several times. For instance: factorial(3) → factorial(2) → factorial(1) → factorial(0), where → means a «it calls». A function, thus, is dynamically activated each time is called. The span of a dynamic activation goes from the point where the function is called until it returns. At a given time, more than one function is dynamically activated. The whole dynamic activation set of functions includes the current function and the dynamic activation set of the function that called it (the current function).

Ok. We have a function that calls itself. No big deal, right? Well, this would not be a problem if it weren't for the rules that a function must observe. Let's quickly recall them.

- Only r0, r1, r2 and r3 can be freely modified.

- lr value at the entry of the function must be kept somewhere because we will need it to leave the function (to return to the caller).
- All other registers r4 to r11 and sp can be modified but they must be restored to their original values upon leaving the function.

In chapter 9 we used a global variable to keep lr. But if we attempted to use a global variable in our factorial(3) example, it would be overwritten at the next dynamic activation of factorial. We would only be able to return from factorial(0) to factorial(1). After that we would be stuck in factorial(1), as lr would always have the same value.

So it looks like we need some way to keep at least the value of lr per each dynamic activation. And not only lr, if we wanted to use registers from r4 to r11 we also need to keep somehow per each dynamic activation, a global variable would not be enough either. This is where the stack comes into play.

The stack

In computing, a stack is a data structure (a way to organize data that provides some interesting properties). A stack typically has three operations: access the top of the stack, push onto the top, pop from the top. Dependening on the context you can only access the top of the stack, in our case we will be able to access more elements than just the top.

But, what is the stack? I already said in chaper 9 that the stack is a region of memory owned solely by the function. We can now reword this a bit better: the stack is a region of memory owned solely by the current dynamic activation. And how we control the stack? Well, in chapter 9 we said that the register sp stands for stack pointer. This register will contain the top of the stack. The region of memory owned by the dynamic activation is the extent of bytes contained between the current value of sp and the initial value that sp had at the beginning of the function. We will call that region the local memory of a function (more precisely, of a dynamic activation of it). We will put there whatever has to be saved at the beginning of a function and restored before leaving. We will also keep there the local variables of a function (dynamic activation).

Our function also has to adhere to some rules when handling the stack.

- The stack pointer (sp) is always 4 byte aligned. This is absolutely mandatory. However, due to the Procedure Call Standard for the ARM architecture (AAPCS), the stack pointer will have to be 8 byte aligned, otherwise funny things may happen when we call what the AAPCS calls as public interfaces (this is, code written by other people).
- The value of sp when leaving the function should be the same value it had upon entering the function.

The first rule is consistent with the alignment constraints of ARM, where most of times addresses must be 4 byte aligned. Due to AAPCS we will stick to the extra 8 byte alignment constraint. The second rule states that, no matter how large is our local memory, it will always disappear at the end of the function. This is important, because local variables of a dynamic activation need not have any storage after that dynamic activation ends.

It is a convention how the stack, and thus the local memory, has its size defined. The stack can grow upwards or downwards. If it grows upwards it means that we have to increase the value of the sp register in order to enlarge the local memory. If it grows downwards we have to do the opposite, the value of the sp register must be subtracted as many bytes as the size of the local storage. In Linux ARM, the stack grows downwards, towards zero (although it never should reach zero). Addresses of local variables have very large values in the 32 bit range. They are usually close to 232.

Another convention when using the stack concerns whether the sp register contains the address of the top of the stack or some bytes above. In Linux ARM the sp register directly points to the top of the stack: in the memory addressed by sp there is useful information.

Ok, we know the stack grows downwards and the top of the stack must always be in sp. So to enlarge the local memory it should be enough by decreasing sp. The local memory is then defined by the range of memory from the current sp value to the original value that sp had at the beginning of the function. One register we almost always have to keep is lr. Let's see how can we keep in the stack.

```
sub sp, sp, #8  /* sp ← sp - 8. This enlarges the stack by 8 bytes */
str lr, [sp]    /* *sp ← lr */
... // Code of the function
ldr lr, [sp]    /* lr ← *sp */
add sp, sp, #8  /* sp ← sp + 8. /* This reduces the stack by 8 bytes
                            effectively restoring the stack
                            pointer to its original value */
bx lr
```

A well behaved function may modify sp but must ensure that at the end it has the same value it had when we entered the function. This is what we do here. We first subtract 8 bytes to sp and at the end we add back 8 bytes.

This sequence of instructions would do indeed. But maybe you remember chapter 8 and the indexing modes that you could use in load and store. Note that the first two instructions behave exactly like a preindexing. We first update sp and then we use sp as the address where we store lr. This is exactly a preindex! Likewise for the last two instructions. We first load lr using the current address of sp and then we decrease sp. This is exactly a postindex!

```
str lr, [sp, #-8]!  /* preindex: sp ← sp - 8; *sp ← lr */
... // Code of the function
ldr lr, [sp], #+8   /* postindex; lr ← *sp; sp ← sp + 8 */
bx lr
```

Yes, these addressing modes were invented to support this sort of things. Using a single instruction is better in terms of code size. This may not seem relevant, but it is when we realize that the stack bookkeeping is required in almost every function we write!

First approach

Let's implement the factorial function above.

First we have to learn a new instruction to multiply two numbers: mul Rdest, Rsource1, Rsource2. Note that multiplying two 32 bit values may require up to 64 bits for the result. This instruction only computes the lower 32 bits. Because we are not going to use 64 bit values in this example, the maximum factorial we will be able to compute is 12! (13! is bigger than 232). We will not check that the entered number is lower than 13 to keep the example simple (I encourage you to add this check to the example, though). In versions of the ARM architecture prior to ARMv6 this instruction could not have Rdest the same as Rsource1. GNU assembler may print a warning if you don't pass -march=armv6.

```
1  /* -- factorial01.s */
2  .data
3
4  message1: .asciz "Type a number: "
5  format:   .asciz "%d"
6  message2: .asciz "The factorial of %d is %d\n"
7
```

```
.text

factorial:
    str lr, [sp,#-4]!  /* Push lr onto the top of the stack */
    str r0, [sp,#-4]!  /* Push r0 onto the top of the stack */
                       /* Note that after that, sp is 8 byte aligned */
    cmp r0, #0         /* compare r0 and 0 */
    bne is_nonzero     /* if r0 != 0 then branch */
    mov r0, #1         /* r0 ← 1. This is the return */
    b end
is_nonzero:
                       /* Prepare the call to factorial(n-1) */
    sub r0, r0, #1     /* r0 ← r0 - 1 */
    bl factorial
                       /* After the call r0 contains factorial(n-1) */
                       /* Load r0 (that we kept in th stack) into r1 */
    ldr r1, [sp]       /* r1 ← *sp */
    mul r0, r0, r1     /* r0 ← r0 * r1 */

end:
    add sp, sp, #+4    /* Discard the r0 we kept in the stack */
    ldr lr, [sp], #+4  /* Pop the top of the stack and put it in lr */
    bx lr              /* Leave factorial */

.global main
main:
    str lr, [sp,#-4]!           /* Push lr onto the top of the stack */
    sub sp, sp, #4              /* Make room for one 4 byte integer in the stack */
                               /* In these 4 bytes we will keep the number */
                               /* entered by the user */
                               /* Note that after that the stack is 8-byte aligned */
    ldr r0, address_of_message1  /* Set &message1 as the first parameter of printf */
    bl printf                   /* Call printf */

    ldr r0, address_of_format   /* Set &format as the first parameter of scanf */
    mov r1, sp                  /* Set the top of the stack as the second parameter */
                               /* of scanf */
```

```
    bl scanf                    /* Call scanf */

    ldr r0, [sp]                /* Load the integer read by scanf into r0 */
                               /* So we set it as the first parameter of factorial */
    bl factorial                /* Call factorial */

    mov r2, r0                  /* Get the result of factorial and move it to r2 */
                               /* So we set it as the third parameter of printf */
    ldr r1, [sp]                /* Load the integer read by scanf into r1 */
                               /* So we set it as the second parameter of printf */
    ldr r0, address_of_message2  /* Set &message2 as the first parameter of printf */
    bl printf                   /* Call printf */


    add sp, sp, #+4             /* Discard the integer read by scanf */
    ldr lr, [sp], #+4           /* Pop the top of the stack and put it in lr */
    bx lr                       /* Leave main */

address_of_message1: .word message1
address_of_message2: .word message2
address_of_format: .word format
```

Most of the code is pretty straightforward. In both functions, main and factorial, we allocate 4 extra bytes on the top of the stack. In factorial, to keep the value of r0, because it will be overwritten during the recursive call (twice, as a first parameter and as the result of the recursive function call). In main, to keep the value entered by the user (if you recall chapter 9 we used a global variable here).

It is important to bear in mind that the stack, like a real stack, the last element stacked (pushed onto the top) will be the first one to be taken out the stack (popped from the top). We store lrand make room for a 4 bytes integer. Since this is a stack, the opposite order must be used to return the stack to its original state. We first discard the integer and then we restore the lr. Note that this happens as well when we reserve the stack storage for the integer using a sub and then we discard such storage doing the opposite operation add.

Can we do it better?

Note that the number of instructions that we need to push and pop data to and from the stack grows linearly with respect to the number of data items. Since ARM was designed for embedded systems, ARM designers devised a way to reduce the number of instructions we need for the «bookkeeping» of the stack. These instructions are load multiple, ldm, and store multiple, stm.

These two instructions are rather powerful and allow in a single instruction perform a lot of things. Their syntax is shown as follows. Elements enclosed in curly braces { and } may be omitted from the syntax (the effect of the instruction will vary, though).

ldm addressing-mode Rbase{!}, register-set
stm addressing-mode Rbase{!}, register-set

We will consider addressing-mode later. Rbase is the base address used to load to or store from the register-set. All 16 ARM registers may be specified in register-set (except pcin stm). A set of addresses is generated when executing these instructions. One address per register in the register-set. Then, each register, in ascending order, is paired with each of these addresses, also in ascending order. This way the lowest-numbered register gets the lowest memory address, and the highest-numbered register gets the highest memory address. Each pair register-address is then used to perform the memory operation: load or store. Specifying ! means that Rbase will be updated. The updated value depends on addressing-mode.

Note that, if the registers are paired with addresses depending on their register number, it seems that they will always be loaded and stored in the same way. For instance a register-setcontaining r4, r5 and r6 will always store r4 in the lowest address generated by the instruction and r6 in the highest one. We can, though, specify what is considered the lowest address or the highest address. So, is Rbase actually the highest address or the lowest address of the multiple load/store? This is one of the two aspects that is controlled by addressing-mode. The second aspect relates to when the address of the memory operation changes between each memory operation.

If the value in Rbase is to be considered the the highest address it means that we should first decrease Rbase as many bytes as required by the number of registers in the register-set(this is 4 times the number of registers) to form the lowest address. Then we can load or store each register consecutively starting from that lowest address, always in ascending order of the register number. This addressing mode is called decreasing and is specified using a d. Conversely, if Rbase is to be considered the lowest address, then this is a bit easier as we can use its value as the

lowest address already. We proceed as usual, loading or storing each register in ascending order of their register number. This addressing mode is called increasing and is specified using an i.
At each load or store, the address generated for the memory operation may be updated after or before the memory operation itself. We can specify this using a or b, respectively.
If we specify !, after the instruction, Rbase will have the highest address generated in the increasing mode and the lowest address generated in the decreasing mode. The final value of Rbase will include the final addition or subtraction if we use a mode that updates after (an amode).
So we have four addressing modes, namely: ia, ib, da and db. These addressing modes are specified as suffixes of the stm and ldm instructions. So the full set of names is stmia, stmib, stmda, stmdb, ldmia, ldmib, ldmda, ldmdb. Now you may think that this is overly complicated, but we need not use all the eight modes. Only two of them are of interest to us now.
When we push something onto the stack we actually decrease the stack pointer (because in Linux the stack grows downwards). More precisely, we first decrease the stack pointer as many bytes as needed before doing the actual store on that just computed stack pointer. So the appropiate addressing-mode when pushing onto the stack is stmdb. Conversely when popping from the stack we will use ldmia: we increment the stack pointer after we have performed the load.

Factorial again

Before illustrating these two instructions, we will first slightly rewrite our factorial.

If you go back to the code of our factorial, there is a moment, when computing n * factorial(n-1), where the initial value of r0 is required. The value of n was in r0 at the beginning of the function, but r0 can be freely modified by called functions. We chose, in the example above, to keep a copy of r0 in the stack in line 12. Later, in line 24, we loaded it from the stack in r1, just before computing the multiplication.
In our second version of factorial, we will keep a copy of the initial value of r0 into r4. But r4 is a register the value of which must be restored upon leaving a function. So we will keep the value of r4 at the entry of the function in the stack. At the end we will restore it back from the stack. This way we can use r4 without breaking the rules of well-behaved functions.

```
factorial:
    str lr, [sp,#-4]!  /* Push lr onto the top of the stack */
    str r4, [sp,#-4]!  /* Push r4 onto the top of the stack */
                      /* The stack is now 8 byte aligned */
    mov r4, r0         /* Keep a copy of the initial value of r0 in r4 */


    cmp r0, #0         /* compare r0 and 0 */
    bne is_nonzero     /* if r0 != 0 then branch */
    mov r0, #1         /* r0 ← 1. This is the return */
    b end
is_nonzero:
                      /* Prepare the call to factorial(n-1) */
    sub r0, r0, #1     /* r0 ← r0 - 1 */
    bl factorial
                      /* After the call r0 contains factorial(n-1) */
                      /* Load initial value of r0 (that we kept in r4) into r1 */
    mov r1, r4         /* r1 ← r4 */
    mul r0, r0, r1     /* r0 ← r0 * r1 */

end:
    ldr r4, [sp], #+4  /* Pop the top of the stack and put it in r4 */
    ldr lr, [sp], #+4  /* Pop the top of the stack and put it in lr */
    bx lr              /* Leave factorial */
```

Note that the remainder of the program does not have to change. This is the cool thing of functions 🙂

Ok, now pay attention to these two sequences in our new factorial version above.

```
    str lr, [sp,#-4]!  /* Push lr onto the top of the stack */
```

```
    str r4, [sp,#-4]!  /* Push r4 onto the top of the stack */
```

```
    ldr r4, [sp], #+4  /* Pop the top of the stack and put it in r4 */
    ldr lr, [sp], #+4  /* Pop the top of the stack and put it in lr */
```

Now, let’s replace them with stmdb and ldmia as explained a few paragraphs ago.

```
    stmdb sp!, {r4, lr}    /* Push r4 and lr onto the stack */
```

```
    ldmia sp!, {r4, lr}    /* Pop lr and r4 from the stack */
```

Note that the order of the registers in the set of registers is not relevant, but the processor will handle them in ascending order, so we should write them in ascending order. GNU assembler will emit a warning otherwise. Since lr is actually r14 it must go after r4. This means that our code is 100% equivalent to the previous one since r4 will end in a lower address than lr: remember our stack grows toward lower addresses, thus r4 which is in the top of the stack in factorialhas the lowest address.

Remembering stmdb sp! and ldmia sp! may be a bit hard. Also, given that these two instructions will be relatively common when entering and leaving functions, GNU assembler provides two mnemonics push and pop for stmdb sp! and ldmia sp!, respectively. Note that these are not ARM instructions actually, just convenience names that are easier to remember.

```
    push {r4, lr}
```

```
    pop {r4, lr}
```

That’s all for today.

ARM assembler in Raspberry Pi – Chapter 11

March 16, 2013 Roger Ferrer Ibáñez, 8

Several times, in earlier chapters, I stated that the ARM architecture was designed with the embedded world in mind. Although the cost of the memory is everyday lower, it still may account as an important part of the budget of an embedded system. The ARM instruction set has several features meant to reduce the impact of code size. One of the features which helps in such approach is predication.

Predication

We saw in chapters 6 and 7 how to use branches in our program in order to modify the execution flow of instructions and implement useful control structures. Branches can be unconditional, for instance when calling a function as we did in chapters 9 and 10, or conditional when we want to jump to some part of the code only when a previously tested condition is met.

Predication is related to conditional branches. What if, instead of branching to some part of code meant to be executed only when a condition C holds, we were able to turn some instructions offwhen that C condition does not hold?. Consider some case like this.

```
if (C)
  T();
else
  E();
```

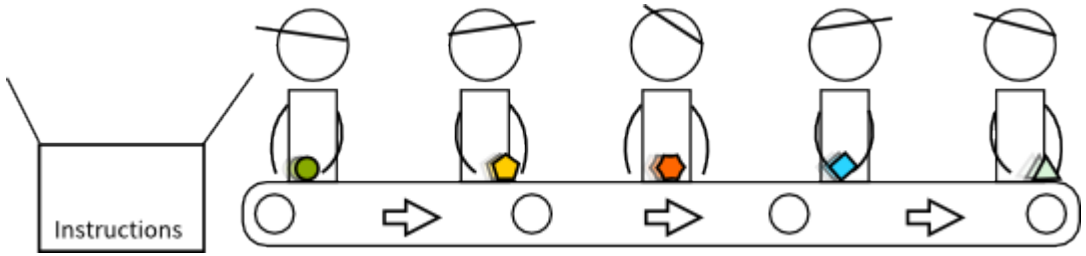
Using predication (and with some invented syntax to express it) we could write the above if as follows.

```
P = C;
[P]  T();
[!P] E();
```

This way we avoid branches. But, why would be want to avoid branches? Well, executing a conditional branch involves a bit of uncertainty. But this deserves a bit of explanation.

The assembly line of instructions

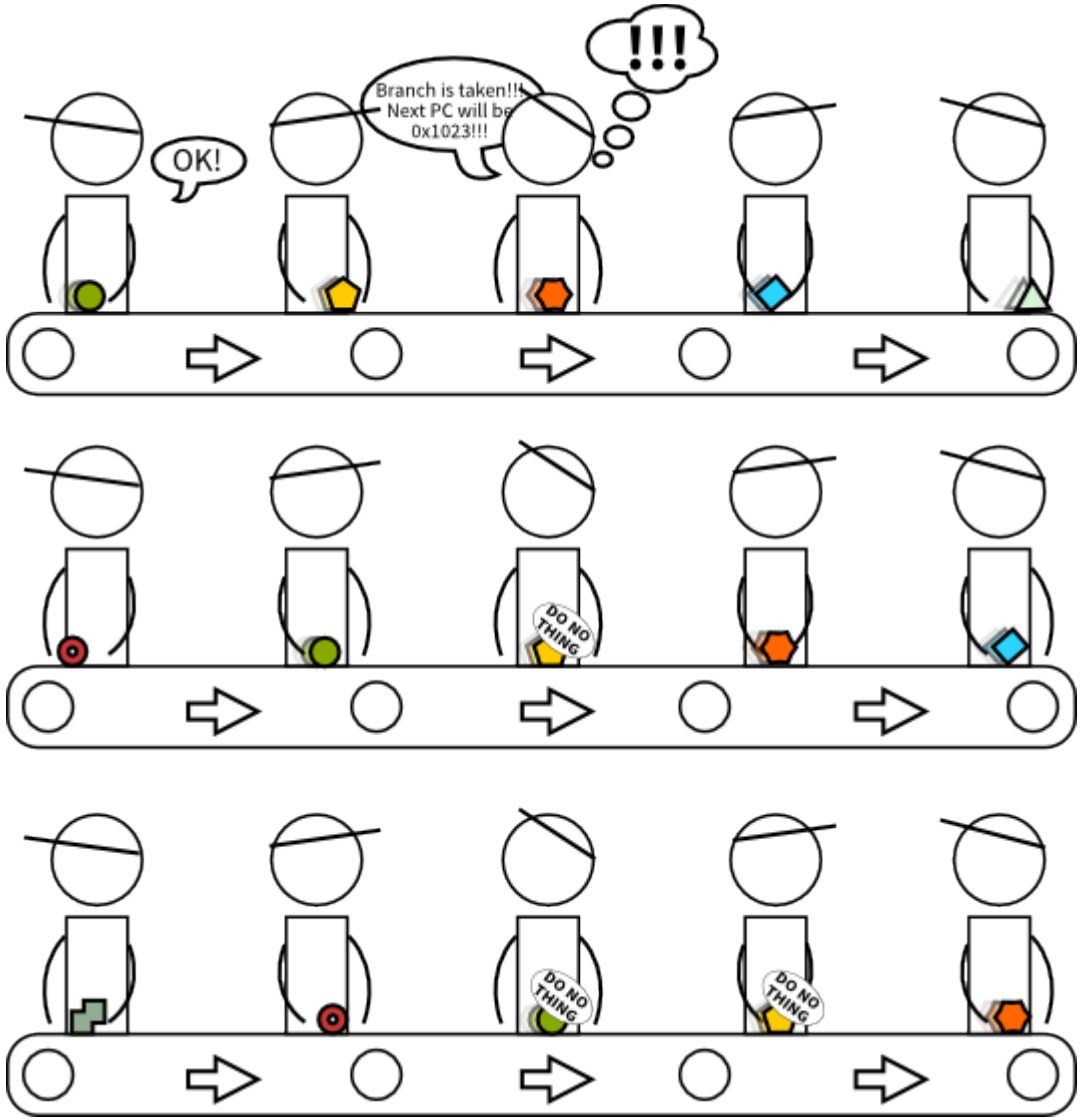
Imagine an assembly line. In that assembly line there are 5 workers, each one fully specialized in a single task. That assembly line executes instructions. Every instruction enters the assembly line from the left and leaves it at the right. Each worker does some task on the instruction and passes to the next worker to the right. Also, imagine all workers are more or less synchronized, each one ends the task in as much 6 seconds. This means that at every 6 seconds there is an instruction leaving the assembly line, an instruction fully executed. It also means that at any given time there may be up to 5 instructions being processed (although not fully executed, we only have one fully executed instruction at every 6 seconds).



The first worker fetches instructions and puts them in the assembly line. It fetches the instruction at the address specified by the register pc. By default, unless told, this worker fetches the instruction physically following the one he previously fetched (this is implicit sequencing).

In this assembly line, the worker that checks the condition of a conditional branch is not the first one but the third one. Now consider what happens when the first worker fetches a conditional branch and puts it in the assembly line. The second worker will process it and pass it to the third one. The third one will process it by checking the condition of the conditional branch. If it does not hold, nothing happens, the branch has no effect. But if the condition holds, the third worker must notify the first one that the next instruction fetched should be the instruction at the address of the branch.

But now there are two instructions in the assembly line that should not be fully executed (the ones that were physically after the conditional branch). There are several options here. The third worker may pick two stickers labeled as DO NOTHING, and stick them to the two next instructions. Another approach would be the third worker to tell the first and second workers «hey guys, stick a DO NOTHING to your current instruction». Later workers, when they see these DO NOTHING stickers will do, huh, nothing. This way each DO NOTHING instruction will never be fully executed.



But by doing this, that nice property of our assembly line is gone: now we do not have a fully executed instruction every 6 seconds. In fact, after the conditional branch there are two DO NOTHINGinstructions. A program that is constantly doing

branches may well reduce the performance of our assembly line from one (useful) instruction each 6 seconds to one instruction each 18 seconds. This is three times slower!
Truth is that modern processors, including the one in the Raspberry Pi, have branch predictorswhich are able to mitigate these problems: they try to predict whether the condition will hold, so the branch is taken or not. Branch predictors, though, predict the future like stock brokers, using the past and, when there is no past information, using some sensible assumptions. So branch predictors may work very well with relatively predictable codes but may work not so well if the code has unpredictable behaviour. Such behaviour, for instance, is observed when running decompressors. A compressor reduces the size of your files removing the redundancy. Redundant stuff is predictable and can be omitted (for instance in “he is wearing his coat” you could ommit “he” or replace “his” by “its”, regardless of whether doing this is rude, because you know you are talking about a male). So a decompressor will have to decompress a file which has very little redundancy, driving nuts the predictor.
Back to the assembly line example, it would be the first worker who attempts to predict where the branch will be taken or not. It is the third worker who verifies if the first worker did the right prediction. If the first worker mispredicted the branch, then we have to apply two stickers again and notify the first worker which is the right address of the next instruction. If the first worker predicted the branch right, nothing special has to be done, which is great.

If we avoid branches, we avoid the uncertainty of whether the branch is taken or not. So it looks like that predication is the way to go. Not so fast. Processing a bunch of instructions that are actually turned off is not an efficient usage of a processor.

Back to our assembly line, the third worker will check the predicate. If it does not hold, the current instruction will get a DO NOTHING sticker but in contrast to a branch, it does not notify the first worker.
So it ends, as usually, that no approach is perfect on its own.

Predication in ARM

In ARM, predication is very simple to use: almost all instructions can be predicated. The predicate is specified as a suffix to the instruction name. The suffix is exactly the same as those used in branches in the chapter 5: eq, neq, le, lt, ge and gt. Instructions that are not predicated are assumed to have a suffix al standing for always.

That predicate always holds and we do not write it for economy (it is valid though). You can understand conditional branches as predicated branches if you feel like.

Collatz conjecture revisited

In chapter 6 we implementd an algorithm that computed the length of the sequence of Hailstone of a given number. Though not proved yet, no number has been found that has an infinite Hailstone sequence. Given our knowledge of functions we learnt in chapters 9 and 10, I encapsulated the code that computes the length of the sequence of Hailstone in a function.

```
/* -- collatz02.s */
.data

message: .asciz "Type a number: "
scan_format : .asciz "%d"
message2: .asciz "Length of the Hailstone sequence for %d is %d\n"

.text

collatz:
    /* r0 contains the first argument */
    /* Only r0, r1 and r2 are modified,
       so we do not need to keep anything
       in the stack */
    /* Since we do not do any call, we do
       not have to keep lr either */
    mov r1, r0                  /* r1 ← r0 */
    mov r0, #0                  /* r0 ← 0 */
  collatz_loop:
    cmp r1, #1                  /* compare r1 and 1 */
    beq collatz_end             /* if r1 == 1 branch to collatz_end */
    and r2, r1, #1              /* r2 ← r1 & 1 */
    cmp r2, #0                  /* compare r2 and 0 */
    bne collatz_odd             /* if r2 != 0 (this is r1 % 2 != 0) branch to collatz_odd
*/
  collatz_even:
```

```
    mov r1, r1, ASR #1          /* r1 ← r1 >> 1. This is r1 ← r1/2 */
    b collatz_end_loop          /* branch to collatz_end_loop */
  collatz_odd:
    add r1, r1, r1, LSL #1      /* r1 ← r1 + (r1 << 1). This is r1 ← 3*r1 */
    add r1, r1, #1              /* r1 ← r1 + 1. */
  collatz_end_loop:
    add r0, r0, #1              /* r0 ← r0 + 1 */
    b collatz_loop              /* branch back to collatz_loop */
  collatz_end:
    bx lr

.global main
main:
    push {lr}                       /* keep lr */
    sub sp, sp, #4                  /* make room for 4 bytes in the stack */
                                   /* The stack is already 8 byte aligned */

    ldr r0, address_of_message      /* first parameter of printf: &message */
    bl printf                       /* call printf */

    ldr r0, address_of_scan_format  /* first parameter of scanf: &scan_format */
    mov r1, sp                      /* second parameter of scanf:
                                     address of the top of the stack */
    bl scanf                        /* call scanf */

    ldr r0, [sp]                    /* first parameter of collatz:
                                     the value stored (by scanf) in the top of the stack
*/
    bl collatz                      /* call collatz */

    mov r2, r0                      /* third parameter of printf:
                                     the result of collatz */
    ldr r1, [sp]                    /* second parameter of printf:
                                     the value stored (by scanf) in the top of the stack
*/
     ldr r0, address_of_message2     /* first parameter of printf: &address_of_message2
*/
```

```
    bl printf

    add sp, sp, #4
    pop {lr}
    bx lr


address_of_message: .word message
address_of_scan_format: .word scan_format
address_of_message2: .word message2
```

Adding predication

Ok, let's add some predication. There is an if-then-else construct in lines 22 to 31. There we check if the number is even or odd. If even we divide it by 2, if even we multiply it by 3 and add 1.

```
    and r2, r1, #1              /* r2 ← r1 & 1 */
    cmp r2, #0                  /* compare r2 and 0 */
      bne collatz_odd                 /* if r2 != 0 (this is r1 % 2 != 0) branch to
collatz_odd */
  collatz_even:
    mov r1, r1, ASR #1          /* r1 ← r1 >> 1. This is r1 ← r1/2 */
    b collatz_end_loop          /* branch to collatz_end_loop */
  collatz_odd:
    add r1, r1, r1, LSL #1      /* r1 ← r1 + (r1 << 1). This is r1 ← 3*r1 */
    add r1, r1, #1              /* r1 ← r1 + 1. */
  collatz_end_loop:
```

```
3
1
```

Note in line 24 that there is a bne (branch if not equal). We can use this condition (and its opposite eq) to predicate this if-then-else construct. Instructions in the then part will be predicated using eq, instructions in the else part will be predicated using ne. The resulting code is shown below.

```
    cmp r2, #0                 /* compare r2 and 0 */
    moveq r1, r1, ASR #1       /* if r2 == 0, r1 ← r1 >> 1. This is r1 ← r1/2 */
     addne r1, r1, r1, LSL #1   /* if r2 != 0, r1 ← r1 + (r1 << 1). This is r1 ←
3*r1 */
    addne r1, r1, #1           /* if r2 != 0, r1 ← r1 + 1. */
```

As you can se there are no labels in the predicated version. We do not branch now so they are not needed anymore. Note also that we actually removed two branches: the one that branches from the condition test code to the else part and the one that branches from the end of the then part to the instruction after the whole if-then-else. This leads to a more compact code.

Does it make any difference in performance?

Taken as is, this program is very small to be accountable for time, so I modified it to run the same calculation inside the collatz function 4194304 (this is 222) times. I chose the number after some tests, so the execution did not take too much time to be a tedium.

Sadly, while the Raspberry Pi processor provides some hardware performance counters I have not been able to use any of them. perf tool (from the package linux-tools-3.2) complains that the counter cannot be opened.

```
$ perf_3.2 stat -e cpu-cycles ./collatz02
   Error:  open_counter  returned  with  19  (No  such  device).  /bin/dmesg  may  provide  additional
information.

  Fatal: Not all events could be opened
```

dmesg does not provide any additional information. We can see, though, that the performance counters was loaded by the kernel.

```
$ dmesg | grep perf
[      0.061722]  hw  perfevents:  enabled  with  v6  PMU  driver,  3  counters
```

```
available
```

Supposedly I should be able to measure up to 3 hardware events at the same time. I think the Raspberry Pi processor, packaged in the BCM2835 SoC does not provide a PMU (Performance Monitoring Unit) which is required for performance counters. Nevertheless we can use cpu-clock to measure the time.
Below are the versions I used for this comparison. First is the branches version, second the predication version.

```
collatz:
    /* r0 contains the first argument */
    push {r4}
    sub sp, sp, #4  /* Make sure the stack is 8 byte aligned */
    mov r4, r0
    mov r3, #4194304
  collatz_repeat:
    mov r1, r4                  /* r1 ← r0 */
    mov r0, #0                  /* r0 ← 0 */
  collatz_loop:
    cmp r1, #1                  /* compare r1 and 1 */
    beq collatz_end             /* if r1 == 1 branch to collatz_end */
    and r2, r1, #1              /* r2 ← r1 & 1 */
    cmp r2, #0                  /* compare r2 and 0 */
     bne collatz_odd                /* if r2 != 0 (this is r1 % 2 != 0) branch to
collatz_odd */
  collatz_even:
    mov r1, r1, ASR #1          /* r1 ← r1 >> 1. This is r1 ← r1/2 */
    b collatz_end_loop          /* branch to collatz_end_loop */
  collatz_odd:
    add r1, r1, r1, LSL #1      /* r1 ← r1 + (r1 << 1). This is r1 ← 3*r1 */
    add r1, r1, #1              /* r1 ← r1 + 1. */
  collatz_end_loop:
    add r0, r0, #1              /* r0 ← r0 + 1 */
    b collatz_loop              /* branch back to collatz_loop */
  collatz_end:
    sub r3, r3, #1
    cmp r3, #0
    bne collatz_repeat
```

```
    add sp, sp, #4  /* Make sure the stack is 8 byte aligned */
    pop {r4}
    bx lr
```

```
collatz2:
    /* r0 contains the first argument */
    push {r4}
    sub sp, sp, #4  /* Make sure the stack is 8 byte aligned */
    mov r4, r0
    mov r3, #4194304
  collatz_repeat:
    mov r1, r4                  /* r1 ← r0 */
    mov r0, #0                  /* r0 ← 0 */
  collatz2_loop:
    cmp r1, #1                  /* compare r1 and 1 */
    beq collatz2_end            /* if r1 == 1 branch to collatz2_end */
    and r2, r1, #1              /* r2 ← r1 & 1 */
    cmp r2, #0                  /* compare r2 and 0 */
    moveq r1, r1, ASR #1        /* if r2 == 0, r1 ← r1 >> 1. This is r1 ← r1/2 */
```

```
    addne r1, r1, r1, LSL #1    /* if r2 != 0, r1 ← r1 + (r1 << 1). This is r1 ← 3*r1 */
    addne r1, r1, #1            /* if r2 != 0, r1 ← r1 + 1. */
  collatz2_end_loop:
    add r0, r0, #1              /* r0 ← r0 + 1 */
    b collatz2_loop             /* branch back to collatz2_loop */
  collatz2_end:
    sub r3, r3, #1
    cmp r3, #0
    bne collatz_repeat
    add sp, sp, #4              /* Restore the stack */
    pop {r4}
    bx lr
```

The tool perf can be used to gather performance counters. We will run 5 times each version. We will use number 123. We redirect the output of yes 123 to the standard input of our tested program. This way we do not have to type it (which may affect the timing of the comparison).
The version with branches gives the following results:

```
$ yes 123 | perf_3.2 stat --log-fd=3 --repeat=5 -e cpu-clock ./collatz_branches 3>&1
Type a number: Length of the Hailstone sequence for 123 is 46
Type a number: Length of the Hailstone sequence for 123 is 46
```

```
Type a number: Length of the Hailstone sequence for 123 is 46
Type a number: Length of the Hailstone sequence for 123 is 46
Type a number: Length of the Hailstone sequence for 123 is 46

 Performance counter stats for './collatz_branches' (5 runs):

      3359,953200 cpu-clock                 ( +-  0,01% )

      3,365263737 seconds time elapsed                                          ( +-  0,01%
)
```

(When redirecting the input of perf one must specify the file descriptor for the output of perf stat itself. In this case we have used the file descriptor number 3 and then told the shell to redirect the file descriptor number 3 to the standard output, which is the file descriptor number 1).

```
$ yes 123 | perf_3.2 stat --log-fd=3 --repeat=5 -e cpu-clock ./collatz_predication 3>&1
Type a number: Length of the Hailstone sequence for 123 is 46
Type a number: Length of the Hailstone sequence for 123 is 46
Type a number: Length of the Hailstone sequence for 123 is 46
Type a number: Length of the Hailstone sequence for 123 is 46
Type a number: Length of the Hailstone sequence for 123 is 46

 Performance counter stats for './collatz_predication' (5 runs):

      2318,217200 cpu-clock                 ( +-  0,01% )

      2,322732232 seconds time elapsed                                          ( +-  0,01%
)
```

So the answer is, yes. In this case it does make a difference. The predicated version runs 1,44 times faster than the version using branches. It would be bold, though, to assume that in general predication outperforms branches. Always measure your time.

That’s all for today.

ARM assembler in Raspberry Pi – Chapter 12

March 28, 2013 Roger Ferrer Ibáñez, 21

We saw in chapter 6 some simple schemes to implement usual structured programming constructs like if-then-else and loops. In this chapter we will revisit these constructs and exploit a feature of the ARM instruction set that we have not learnt yet.

Playing with loops

The most generic form of loop is this one.

```
while (E)
  S;
```

There are also two special forms, which are actually particular incarnations of the one shown above but are interesting as well.

```
for (i = lower; i <= upper; i += step)
  S;
```

```
do
  S
while (E);
```

Some languages, like Pascal, have constructs like this one.

```
repeat
  S
until E;
```

but this is like a do S while (!E).

We can manipulate loops to get a form that may be more convenient. For instance.

```
   do
     S
   while (E);
```

```
/* Can be rewritten as */

   S;
   while (E)
      S;
```

```
   while (E)
     S;

/* Can be rewritten as */

   if (E)
   {
      do
        S
      while (E);
   }
```

The last manipulation is interesting, because we can avoid the if-then if we directly go to the while part.

```
/* This is not valid C */
goto check;
do
  S
check: while (E);
```

In valid C, the above transformation would be written as follows.

```
goto check;
loop:
  S;
check:
  if (E) goto loop;
```

Which looks much uglier than abusing a bit C syntax.

The -s suffix

So far, when checking the condition of an if or while, we have evaluated the condition and then used the cmp intruction to update cpsr. The update of the cpsr is mandatory for our conditional codes, no matter if we use branching or predication. But cmp is not the only way to update cpsr. In fact many instructions can update it.
By default an instruction does not update cpsr unless we append the suffix -s. So instead of the instruction add or sub we write adds or subs. The result of the instruction (what would be stored in the destination register) is used to update cpsr.
How can we use this? Well, consider this simple loop counting backwards.

```
/* for (int i = 100 ; i >= 0; i--) */
mov r1, #100
loop:
  /* do something */
  sub r1, r1, #1       /* r1 ← r1 - 1 */
  cmp r1, #0           /* update cpsr with r1 - 0 */
  bge loop             /* branch if r1 >= 100 */
```

If we replace sub by subs then cpsr will be updated with the result of the subtration. This means that the flags N, Z, C and V will be updated, so we can use a branch right after subs. In our case we want to jump back to loop only if i >= 0, this is when the result is non-negative. We can use bpl to achieve this.

```
/* for (int i = 100 ; i >= 0; i--) */
mov r1, #100
loop:
  /* do something */
  subs r1, r1, #1      /* r1 ← r1 - 1  and update cpsr with the final r1 */
  bpl loop              /* branch if the previous sub computed a positive number (N flag in cpsr is
0) */
```

It is a bit tricky to get these things right (this is why we use compilers). For instance this similar, but not identical, loop would use bne instead of bpl. Here the condition is ne (not equal). It would be nice to have an alias like nz (not zero) but, unfortunately, this does not exist in ARM.

```
/* for (int i = 100 ; i > 0; i--). Note here i > 0, not i >= 0 as in the example above */
mov r1, #100
```

```
loop:
  /* do something */
  subs r1, r1, #1      /* r1 ← r1 - 1  and update cpsr with the final r1 */
  bne loop              /* branch if the previous sub computed a number that is not zero (Z flag in cpsr is
0) */
```

A rule of thumb where we may want to apply the use of the -s suffix is in codes in the following form.

```
s = ...
if (s @ 0)
```

where @ means any comparison respect 0 (equals, different, lower, etc.).

Operating 64-bit numbers

As an example of using the suffix -s we will implement three 64-bit integer operations in ARM: addition, subtraction and multiplication. Remember that ARM is a 32-bit architecture, so everything is 32-bit minded. If we only use 32-bit numbers, this is not a problem, but if for some reason we need 64-bit numbers things get a bit more complicated. We will represent a 64-bit number as two 32-bit numbers, the lower and higher part. This way a 64-bit number n represented using two 32-bit parts, nlower and nhigher will have the value n = 232 × nhigher + nlower

We will, obviously, need to kep the 32-bit somewhere. When keeping them in registers, we will use two consecutive registers (e.g. r1 and r2, that we will write it as {r1,r2}) and we will keep the higher part in the higher numbered register. When keeping a 64-bit number in memory, we will store in two consecutive addresses the two parts, being the lower one in the lower address. The address will be 8-byte aligned.

Addition

Adding two 64-bit numbers using 32-bit operands means adding first the lower part and then adding the higher parts but taking into account a possible carry from the lower part. With our current knowledge we could write something like this (assume the first number is in {r2,r3}, the second in {r4,r5} and the result will be in {r0,r1}).

```
add r1, r3, r5      /* First we add the higher part */
                   /* r1 ← r3 + r5 */
adds r0, r2, r4     /* Now we add the lower part and we update cpsr */
                   /* r0 ← r2 + r4 */
addcs r1, r1, #1    /* If adding the lower part caused carry, add 1 to the higher part
*/
```

```
                    /* if C = 1 then r1 ← r1 + 1 */
                    /* Note that here the suffix -s is not applied, -cs means carry set
*/
```

This would work. Fortunately ARM provides an instructions adc which adds two numbers and the carry flag. So we could rewrite the above code with just two instructions.

```
adds r0, r2, r4      /* First add the lower part and update cpsr */
                    /* r0 ← r2 + r4 */
adc r1, r3, r5       /* Now add the higher part plus the carry from the lower one
*/
                    /* r1 ← r3 + r5 + C */
```

Subtraction

Subtracting two numbers is similar to adding them. In ARM when subtracting two numbers using subs, if we need to borrow (because the second operand is larger than the first) then C will be disabled (C will be 0). If we do not need to borrow, C will be enabled (C will be 1). This is a bit surprising but consistent with the remainder of the architecture (check in chapter 5 conditions CS/HS and CC/LO). Similar to adc there is a sbc which performs a normal subtraction if C is 1. Otherwise it subtracts one more element. Again, this is consistent on how C works in the subsinstruction.

```
subs r0, r2, r4      /* First subtract the lower part and update cpsr */
                    /* r0 ← r2 - r4 */
sbc r1, r3, r5        /* Now subtract the higher part plus the NOT of the carry from the lower
one */
                    /* r1 ← r3 - r5 - ~C */
```

Multiplication

Multiplying two 64-bit numbers is a tricky thing. When we multiply two N-bit numbers the result may need up to 2*N-bits. So when multiplying two 64-bit numbers we may need a 128-bit number. For the sake of simplicity we will assume that this does not happen and 64-bit will be enough. Our 64-bit numbers are two 32-bit integers, so a 64-bit x is actually $x = 232 \times x1 + x0$, where x1 and x0 are two 32-bit numbers. Similarly another 64-bit number y would be $y = 232 \times y1 + y0$. Multiplying x and y yields z where $z = 264 \times x1 \times y1 + 232 \times (x0 \times y1 + x1 \times y0) + x0 \times y0$. Well, now our problem is multiplying each xi by yi, but again we may need 64-bit to represent the value.

ARM provides a bunch of different instructions for multiplication. Today we will see just three of them. If we are multiplying 32-bits and we do not care about the result not fitting in a 32-bit number we can use mul Rd, Rsource1,

Rsource2. Unfortunately it does not set any flag in the cpsr useful for detecting an overflow of the multiplication (i.e. when the result does not fit in the 32-bit range). This instruction is the fastest one of the three. If we do want the 64-bit resulting from the multiplication, we have two other instructions smull and umull. The former is used when we multiply to numbers in two's complement, the latter when we represent unsigned values. Their syntax is {s,u}mull RdestLower, RdestHigher, Rsource1, Rsource2. The lower part of the 64-bit result is kept in the register RdestLower and the higher part in he register RdestHigher.

In this example we have to use umull otherwise the 32-bit lower parts might end being interpreted as negative numbers, giving negative intermediate values. That said, we can now multiply x0 and y0. Recall that we have the two 64-bit numbers in r2,r3 and r4,r5 pairs of registers. So first multiply r2 and r4. Note the usage of r0 since this will be its final value. In contrast, register r6 will be used later.

```
umull r0, r6, r2, r4
```

Now let's multiply x0 by y1 and x1 by y0. This is r3 by r4 and r2 by r5. Note how we overwrite r4and r5 in the second multiplication. This is fine since we will not need them anymore.

```
umull r7, r8, r3, r4
umull r4, r5, r2, r5
```

There is no need to make the multiplication of x1 by y1 because if it gives a nonzero value, it will always overflow a 64-bit number. This means that if both r3 and r5 were nonzero, the multiplication will never fit a 64-bit. This is a suficient condition, but not a necessary one. The number might overflow when adding the intermediate values that will result in r1.

```
adds r2, r7, r4
adc r1, r2, r6
```

Let's package this code in a nice function in a program to see if it works. We will multiply numbers 12345678901 (this is 2×232 + 3755744309) and 12345678 and print the result.

```
1  /* -- mult64.s */
2  .data
3
4  .align 4
5  message : .asciz "Multiplication of %lld by %lld is %lld\n"
6
7  .align 8
```

```
number_a_low: .word 3755744309
number_a_high: .word 2

.align 8
number_b_low: .word 12345678
number_b_high: .word 0

.text

/* Note: This is not the most efficient way to doa 64-bit multiplication.
   This is for illustration purposes */
mult64:
   /* The argument will be passed in r0, r1 and r2, r3 and returned in r0, r1 */
   /* Keep the registers that we are going to write */
   push {r4, r5, r6, r7, r8, lr}
   /* For covenience, move {r0,r1} into {r4,r5} */
   mov r4, r0   /* r0 ← r4 */
   mov r5, r1   /* r5 ← r1 */

   umull r0, r6, r2, r4    /* {r0,r6} ← r2 * r4 */
   umull r7, r8, r3, r4    /* {r7,r8} ← r3 * r4 */
   umull r4, r5, r2, r5    /* {r4,r5} ← r2 * r5 */
   adds r2, r7, r4         /* r2 ← r7 + r4 and update cpsr */
   adc r1, r2, r6          /* r1 ← r2 + r6 + C */

   /* Restore registers */
   pop {r4, r5, r6, r7, r8, lr}
   bx lr                   /* Leave mult64 */

.global main
main:
    push {r4, r5, r6, r7, r8, lr}       /* Keep the registers we are going to modify */
                                        /* r8 is not actually used here, but this way
                                           the stack is already 8-byte aligned */
    /* Load the numbers from memory */
    /* {r4,r5} ← a */
    ldr r4, addr_number_a_low       /* r4 ← &a_low */
```

```
    ldr r4, [r4]                    /* r4 ← *r4 */
    ldr r5, addr_number_a_high      /* r5 ← &a_high  */
    ldr r5, [r5]                    /* r5 ← *r5 */

    /* {r6,r7} ← b */
    ldr r6, addr_number_b_low       /* r6 ← &b_low  */
    ldr r6, [r6]                    /* r6 ← *r6 */
    ldr r7, addr_number_b_high      /* r7 ← &b_high  */
    ldr r7, [r7]                    /* r7 ← *r7 */

    /* Now prepare the call to mult64
    /*
       The first number is passed in
       registers {r0,r1} and the second one in {r2,r3}
    */
    mov r0, r4                   /* r0 ← r4 */
    mov r1, r5                   /* r1 ← r5 */

    mov r2, r6                   /* r2 ← r6 */
    mov r3, r7                   /* r3 ← r7 */

    bl mult64                   /* call mult64 function */
    /* The result of the multiplication is in r0,r1 */

    /* Now prepare the call to printf */
    /* We have to pass &message, {r4,r5}, {r6,r7} and {r0,r1} */
    push {r1}                    /* Push r1 onto the stack. 4th (higher) parameter */
    push {r0}                    /* Push r0 onto the stack. 4th (lower) parameter */
    push {r7}                    /* Push r7 onto the stack. 3rd (higher) parameter */
    push {r6}                    /* Push r6 onto the stack. 3rd (lower) parameter */
    mov r3, r5                   /* r3 ← r5.                2rd (higher) parameter */
    mov r2, r4                   /* r2 ← r4.                2nd (lower) parameter */
    ldr r0, addr_of_message      /* r0 ← &message           1st parameter */
    bl printf                    /* Call printf */
    add sp, sp, #16              /* sp ← sp + 16 */
                                /* Pop the two registers we pushed above */

```

```
47
48     mov r0, #0                    /* r0 ← 0 */
49     pop {r4, r5, r6, r7, r8, lr}        /* Restore the registers we kept */
50     bx lr                         /* Leave main */
51
52 addr_of_message : .word message
53 addr_number_a_low: .word number_a_low
54 addr_number_a_high: .word number_a_high
55 addr_number_b_low: .word number_b_low
56 addr_number_b_high: .word number_b_high
57
```

Observe first that we have the addresses of the lower and upper part of each number. Instead of this we could load them by just using an offset, as we saw in chapter 8. So, in lines 41 to 44 we could have done the following.

```
40
41     /* {r4,r5} ← a */
42     ldr r4, addr_number_a_low       /* r4 ← &a_low */
43     ldr r5, [r4, +#4]               /* r5 ← *(r4 + 4) */
44     ldr r4, [r4]                    /* r4 ← *r4  */
45
46
```

In the function mult64 we pass the first value (x) as r0,r1 and the second value (y) as r2,r3. The result is stored in r0,r1. We move the values to the appropiate registers for parameter passing in lines 57 to 61.
Printing the result is a bit complicated. 64-bits must be passed as pairs of consecutive registers where the lower part is in an even numbered register. Since we pass the address of the message in r0 we cannot pass the first 64-bit integer in r1. So we skip r1 and we use r2 and r3 for the first argument. But now we have run out of registers for parameter passing. When this happens, we have to use the stack for parameter passing. Two rules have to be taken into account when passing data in the stack.

1. You must ensure that the stack is aligned for the data you are going to pass (by adjusting the stack first). So, for 64-bit numbers, the stack must be 8-byte aligned. If you pass an 32-bit number and then a 64-bit number, you will

have to skip 4 bytes before passing the 64-bit number. Do not forget to keep the stack always 8-byte aligned per the Procedure Call Standard for ARM Architecture (AAPCS) requirement.

2.An argument with a lower position number in the call must have a lower address in the stack. So we have to pass the arguments in opposite order.

The second rule is what explains why we push first r1 and then r0, when they are the registers containing the last 64-bit number (the result of the multiplication) we want to pass to printf.
Note that in the example above, we cannot pass the parameters in the stack using push {r0,r1,r6,r7}, which is equivalent to push {r0}, push {r1}, push {r6} and push {r7}, but not equivalent to the required order when passing the arguments on the stack.
If we run the program we should see something like.

```
$ ./mult64_2
Multiplication of 12345678901 by 12345678 is 152415776403139878
```

That’s all for today.

Capybara, pop up windows and the new PayPal sandbox

April 27, 2013 brafales, 0

This past weeks we have been doing a massive refactoring of our testing suite at work to set up a nice CI server setup, proper factories, etc. Our tool-belt so far is basically a well known list of Rails gems:

- Factory Girl for factories.
- RSpec as a testing framework (although we'll switch back to Test::Unit soon).
- Capybara for integration testing.

For the CI server we decided to use a third party SaaS as our dev team is small and we don't have the manpower nor the time to set it up ourselves, and we went for CircleCI, which has given us good results so far (easy to set up, in fact almost works out of the box without having to do anything, it has a good integration with GitHub, it's reasonably fast, and the guys are continuously improving it and very receptive to client's feedback).

Back to the post topic, when refactoring the integration tests, we discovered that PayPal decided recently to change the way their development sandbox works, and the tests we had in place broke because of it.

The basic workflow when having to test with PayPal involves a series of steps:

- Visit their sandbox page and log in with your testing credentials. This saves a cookie in the browser.
- Go back to your test page and do the steps needed to perform a payment using PayPal.
- Authenticate again to PayPal with your test buyers account and pay.
- Catch the PayPal response and do whatever you need to finish your test.

With the old PayPal sandbox, the login was pretty straightforward as you only needed to find the username and password fields in the login form of the sandbox page, fill them in, click the login button, and that was all. But with the new version it's not that easy. The new sandbox has no login form at the main page. It has a login button which you have to click, then a popup window is shown with the login form. In there you have to input your credentials and click on

the login button. Then this popup window does some server side magic, closes itself and triggers a reload on the main page, which will finally show you as logged in.

There's probably a POST request that you can automatically do to simplify all this, but PayPal is not known as developer documentation friendly so I couldn't find it. As a result, we had to modify our Capybara tests to handle this new scenario. As we've never worked with pop up windows before I thought it'd be nice to share how we did it in case you need to do something similar.
The basic workflow is as follows:

- Open the main PayPal sandbox window.
- Click on the login button.
- Find the new popup window.
- Fill in the form in that new window.
- Go back to your main window.
- Continue with your usual testing.

This assumes you are using the Selenium driver for Capybara. Here's the code we used to get this done:

```
describe "a paypal express transaction", :js => true do
  it "should just work" do
    # Visit the PayPal sandbox url
    visit "https://developer.paypal.com/"

    # The link for the login button has no id...
                find(:xpath,    "//a[contains(@class,'ppLogin_internal    cleanslate    scTrack:ppAccess-login
ppAccessBtn')]").click

    # Here we have to use the driver to find the newly opened window using it's name
    # We also get the reference to the main window as later on we'll have to go back to it
    login_window = page.driver.find_window('PPA_identity_window')
    main_window = page.driver.find_window('')
```

```
    # We use this to execute the next instructions in the popup window
    page.within_window(login_window) do
      #Normally fill in the form and log in
      fill_in 'email', :with => "<your paypal sandbox username>"
      fill_in 'password', :with => "<your paypal sandbox password>"
      click_button 'Log In'
    end

    #More on this sleep later
    sleep(30)

    #Switch back to the main window and do the rest of the test in it
    page.within_window(main_window) do
      #Here goes the rest of your test
    end
  end
end
```

Now there is an important thing to note on the code above: the sleep(30) call. By now you may have read on hundreds of places that using sleep is not a good practice and that your tests should not rely on that. And that's true. However, PayPal does a weird thing and this is the only way I could use to make the tests pass. It turns out that after clicking the Log In button, the system does some behind the curtains magic, and after having done that, the popup window closes itself and then triggers a reload on the main page. This reload triggering makes things difficult. If you instruct Capybara to visit your page right after clicking the Log In button, you risk having that reload trigger fired in between, and then your test will fail because the next selector you use will not be found as the browser will be in the PayPal sandbox page.

There are probably better and more elegant ways to get around this. Maybe place a code to re-trigger your original visit if it detects you are still on the PayPal page, etc. Feel free to use the comments to suggest possible solutions to that particular problem.

ARM assembler in Raspberry Pi – Chapter 13

May 12, 2013 Roger Ferrer Ibáñez, 22

So far, all examples have dealt with integer values. But processors would be rather limited if they were only able to work with integer values. Fortunately they can work with floating point numbers. In this chapter we will see how we can use the floating point facilities of our Raspberry Pi.

Floating point numbers

Following is a quick recap of what is a floating point number.

A binary floating point number is an approximate representation of a real number with three parts: sign, mantissa and exponent. The sign may be just 0 or 1, meaning 1 a negative number, positive otherwise. The mantissa represents a fractional magnitude. Similarly to 1.2345 we can have a binary 1.01110 where every digit is just a bit. The dot means where the integer part ends and the fractional part starts. Note that there is nothing special in binary fractional numbers: 1.01110 is just 20 + 2-2 + 2-3 + 2-4 = 1.43750(10. Usually numbers are normalized, this means that the mantissa is adjusted so the integer part is always 1, so instead of 0.00110101 we would represent 1.101101 (in fact a floating point may be a denormal if this property does not hold, but such numbers lie in a very specific range so we can ignore them here). If the mantissa is adjusted so it always has a single 1 as the integer part two things happen. First, we do not represent the integer part (as it is always 1 in normalized numbers). Second, to make things sound we need an exponent which compensates the mantissa being normalized. This means that the number -101.110111 (remember that it is a binary real number) will be represented by a sign = 1, mantissa = 1.01110111 and exponent = 2 (because we moved the dot 2 digits to the left). Similarly, number 0.0010110111 is represented with a sign = 0, mantissa = 1.0110111 and exponent = -3 (we moved the dot 3 digits to the right).
In order for different computers to be able to share floating point numbers, IEEE 754 standardizes the format of a floating point number. VFPv2 supports two of the IEEE 754 numbers: Binary32 and Binary64, usually known by their C types, float and double, or by single- and double-precision, respectively. In a single-precision floating point the mantissa is 23 bits (+1 of the integer one for normalized numbers) and the exponent is 8 bits (so the exponent ranges from -126 to 127). In a double-precision floating point the mantissa is 52 bits (+1) and the exponent is 11 bits (so the exponent

ranges from -1022 to 1023). A single-precision floating point number occupies 32 bit and a double-precision floating point number occupies 64 bits. Operating double-precision numbers is in average one and a half to twice slower than single-precision.
Goldberg's famous paper is a classical reference that should be read by anyone serious when using floating point numbers.

Coprocessors

As I stated several times in earlier chapters, ARM was designed to be very flexible. We can see this in the fact that ARM architecture provides a generic coprocessor interface. Manufacturers of system-on-chips may bundle additional coprocessors. Each coprocessor is identified by a number and provides specific instructions. For instance the Raspberry Pi SoC is a BCM2835 which provides a multimedia coprocessor (which we will not discuss here).

That said, there are two standard coprocessors in the ARMv6 architecture: 10 and 11. These two coprocessors provide floating point support for single and double precision, respectively. Although the floating point instructions have their own specific names, they are actually mapped to generic coprocessor instructions targeting coprocessor 10 and 11.

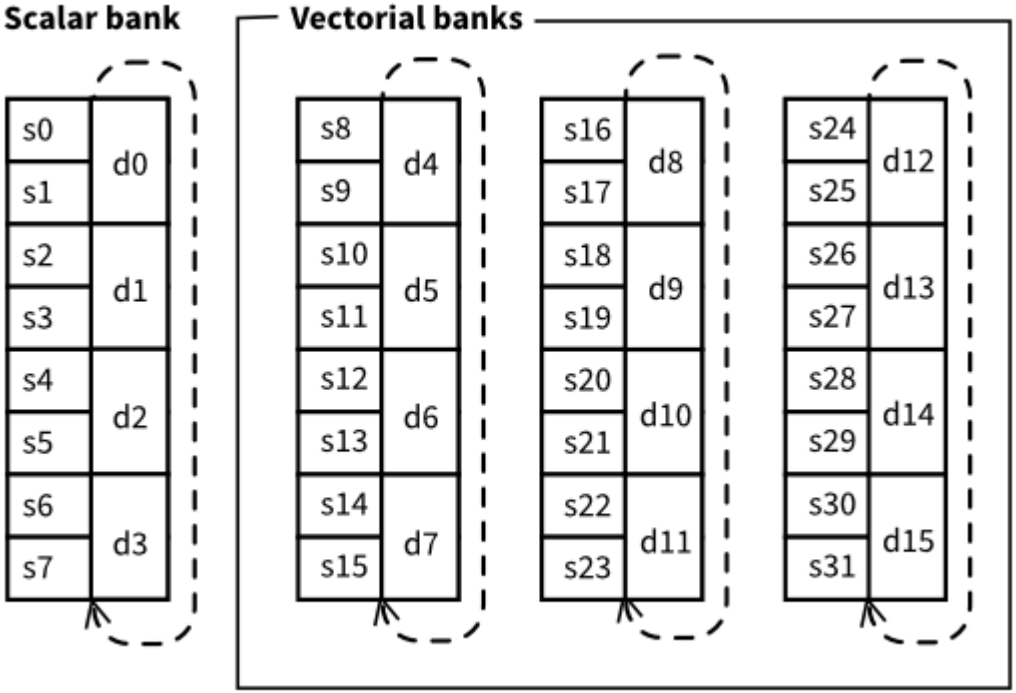
Vector Floating-point v2

ARMv6 defines a floating point subarchitecture called the Vector Floating-point v2 (VFPv2). Version 2 because earlier ARM architectures supported a simpler form called now v1. As stated above, the VFP is implemented on top of two standarized coprocessors 10 and 11. ARMv6 does not require VFPv2 be implemented in hardware (one can always resort to a slower software implementation). Fortunately, the Raspberry Pi does provide a hardware implementation of VFPv2.

VFPv2 Registers

We already know that the ARM architecture provides 16 general purpose registers r0 to r15, where some of them play special roles: r13, r14 and r15. Despite their name, these general purpose registers do not allow operating floating point numbers in them, so VFPv2 provides us with some specific registers. These registers are named s0 to s31, for single-precision, and d0 to d15 for double precision. These are not 48 different registers. Instead every dn is mapped to two (consecutive) registers s2n and s2n+1, where $0 \leq n \leq 15$.

These registers are structured in 4 banks: s0–s7 (d0–d3), s8–s15 (d4–d7), s16–s23 (d8–d11) and s24–s31 (d12–d15). We will call the first bank (bank 0, s0–s7, d0–d3) the scalar bank, while the remaining three are vectorial banks (below we will see why).



VFPv2 provides three control registers but we will only be interested in one called fpscr. This register is similar to the cpsr as it keeps the usual comparison flags N, Z, C and V. It also stores two fields that are very useful, len and stride. These two fields control how floating point instructions behave. We will not care very much of the remaining information in this register: status information of the floating point exceptions, the current rounding mode and whether denormal numbers are flushed to zero.

Arithmetic operations

Most VFPv2 instructions are of the form vname Rdest, Rsource1, Rsource2 or fnameRdest, Rsource1. They have three modes of operation.

- Scalar. This mode is used when the destination register is in bank 0 (s0–s7 or d0–d3). In this case, the instruction operates only with Rsource1 and Rsource2. No other registers are involved.
- Vectorial. This mode is used when the destination register and Rsource2 (or Rsource1 for instructions with only one source register) are not in the bank 0. In this case the instruction will operate as many registers (starting from the given register in the instruction and wrapping around the bank of the register) as defined in field len of the fpscr (at least 1). The next register operated is defined by the stride field of the fpscr (at least 1). If wrap-around happens, no register can be operated twice.
- Scalar expanded (also called mixed vector/scalar). This mode is used if Rsource2 (or Rsource1 if the instruction only has one source register) is in the bank0, but the destination is not. In this case Rsource2 (or Rsource1 for instructions with only one source) is left fixed as the source. The remaining registers are operated as in the vectorial case (this is, using len and stride from the fpscr).

Ok, this looks pretty complicated, so let's see some examples. Most instructions end in .f32 if they operate on single-precision and in .f64 if they operate in double-precision. We can add two single-precision numbers using vadd.f32 Rdest, Rsource1, Rsource2 and double-precision using vadd.f64 Rdest, Rsource1, Rsource2. Note also that we can use predication in these instructions (but be aware that, as usual, predication uses the flags in cpsrnot in fpscr). Predication would be specified before the suffix like in vaddne.f32.

```
// For this example assume that len = 4, stride = 2
vadd.f32 s1, s2, s3  /* s1 ← s2 + s3. Scalar operation because Rdest = s1 in the bank 0 */
vadd.f32 s1, s8, s15 /* s1 ← s8 + s15. ditto */
vadd.f32 s8, s16, s24 /* s8  ← s16 + s24
                     s10 ← s18 + s26
                     s12 ← s20 + s28
                     s14 ← s22 + s30
                               or  more  compactly  {s8,s10,s12,s14}  ←  {s16,s18,s20,s22}  +
{s24,s26,s28,s30}
                     Vectorial, since Rdest and Rsource2 are not in bank 0
                  */
```

```
vadd.f32 s10, s16, s24 /* {s10,s12,s14,s8} ← {s16,s18,s20,s22} + {s24,s26,s28,s30}.
                       Vectorial, but note the wraparound inside the bank after s14.
                     */
vadd.f32 s8, s16, s3 /* {s8,s10,s12,s14} ← {s16,s18,s20,s22} + {s3,s3,s3,s3}
                      Scalar expanded since Rsource2 is in the bank 0
                    */
```

Load and store

Once we have a rough idea of how we can operate floating points in VFPv2, a question remains: how do we load/store floating point values from/to memory? VFPv2 provides several specific load/store instructions.

We load/store one single-precision floating point using vldr/vstr. The address of the load/store must be already in a general purpose register, although we can apply an offset in bytes which must be a multiple of 4 (this applies to double-precision as well).

```
vldr s1, [r3]          /* s1 ← *r3 */
vldr s2, [r3, #4]      /* s2 ← *(r3 + 4) */
vldr s3, [r3, #8]      /* s3 ← *(r3 + 8) */
vldr s4, [r3, #12]      /* s3 ← *(r3 + 12) */

vstr s10, [r4]         /* *r4 ← s10 */
vstr s11, [r4, #4]      /* *(r4 + 4) ← s11 */
vstr s12, [r4, #8]      /* *(r4 + 8) ← s12 */
vstr s13, [r4, #12]       /* *(r4 + 12) ← s13
*/
```

We can load/store several registers with a single instruction. In contrast to general load/store, we cannot load an arbitrary set of registers but instead they must be a sequential set of registers.

// Here precision can be s or d for single-precision and double-precision
// floating-point-register-set is {sFirst-sLast} for single-precision
// and {dFirst-dLast} for double-precision
vldm indexing-mode precision Rbase{!}, floating-point-register-set
vstm indexing-mode precision Rbase{!}, floating-point-register-set

The behaviour is similar to the indexing modes we saw in chapter 10. There is a Rbase register used as the base address of several load/store to/from floating point registers. There are only two indexing modes: increment after and decrement before. When using increment after, the address used to load/store the floating point value register is increased by 4 after the load/store has happened. When using decrement before, the base address is first subtracted as many bytes as foating point values are going to be loaded/stored. Rbase is always updated in decrement before but it is optional to update it in increment after.

```
vldmias r4, {s3-s8} /* s3 ← *r4
                       s4 ← *(r4 + 4)
                       s5 ← *(r4 + 8)
                       s6 ← *(r4 + 12)
                       s7 ← *(r4 + 16)
                       s8 ← *(r4 + 20)
                     */
vldmias r4!, {s3-s8} /* Like the previous instruction
                        but at the end r4 ← r4 + 24
                      */
vstmdbs r5!, {s12-s13} /*  *(r5 - 4 * 2) ← s12
                          *(r5 - 4 * 1) ← s13
                          r5 ← r5 - 4*2
                       */
```

For the usual stack operations when we push onto the stack several floating point registers we will use vstmdb with sp! as the base register. To pop from the stack we will use vldmia again with sp! as the base register. Given that these instructions names are very hard to remember we can use the mnemonics vpush and vpop, respectively.

```
vpush {s0-s5} /* Equivalent to vstmdb sp!, {s0-s5} */
vpop {s0-s5}  /* Equivalent to vldmia sp!, {s0-s5} */
```

Movements between registers

Another operation that may be required sometimes is moving among registers. Similar to the movinstruction for general purpose registers there is the vmov instruction. Several movements are possible.
We can move floating point values between two floating point registers of the same precision

```
vmov s2, s3  /* s2 ← s3 */
```

Between one general purpose register and one single-precision register. But note that data is not converted. Only bits are copied around, so be aware of not mixing floating point values with integer instructions or the other way round.

```
vmov s2, r3  /* s2 ← r3 */
vmov r4, s5  /* r4 ← s5 */
```

Like the previous case but between two general purpose registers and two consecutive single-precision registers.

```
vmov s2, s3, r4, r10 /* s2 ← r4
                        s3 ← r10 */
```

Between two general purpose registers and one double-precision register. Again, note that data is not converted.

```
vmov d3, r4, r6  /* Lower32BitsOf(d3) ← r4
                    Higher32BitsOf(d3) ← r6
                 */
vmov r5, r7, d4 /* r5 ← Lower32BitsOf(d4)
                   r7 ← Higher32BitsOf(d4)
                 */
```

Conversions

Sometimes we need to convert from an integer to a floating-point and the opposite. Note that some conversions may potentially lose precision, in particular when a floating point is converted to an integer. There is a single instruction vcvt with a suffix .T.S where T (target) and S (source) can be u32, s32, f32 and f64 (S must be different to T). Both registers must be floating point registers, so in order to convert integers to floating point or floating point to an integer value an extra vmov instruction will be required from or to an integer register before or after the conversion. Because of this, for a moment (between the two instructions) a floating point register will contain a value which is not a IEEE 754 value, bear this in mind.

```
vcvt.f64.f32 d0, s0  /* Converts s0 single-precision value
                        to a double-precision value and stores it in d0 */

vcvt.f32.f64 s0, d0  /* Converts d0 double-precision value
                        to a single-precision value  and stores it in s0 */
```

```
vmov s0, r0           /* Bit copy from integer register r0 to s0 */
vcvt.f32.s32 s0, s0   /* Converts s0 signed integer value
                         to a single-precision value and stores it in s0 */

vmov s0, r0           /* Bit copy from integer register r0 to s0 */
vcvt.f32.u32 s0, s0   /* Converts s0 unsigned integer value
                         to a single-precision value and stores in s0 */

vmov s0, r0           /* Bit copy from integer register r0 to s0 */
vcvt.f64.s32 d0, s0   /* Converts r0 signed integer value
                         to a double-precision value and stores in d0 */

vmov s0, r0           /* Bit copy from integer register r0 to s0 */
vcvt.f64.u32 d0, s0   /* Converts s0 unsigned integer value
                         to a double-precision value and stores in d0 */
```

Modifying fpscr

The special register fpscr, where len and stride are set, cannot be modified directly. Instead we have to load fpscr into a general purpose register using vmrs instruction. Then we operate on the register and move it back to the fpscr, using the vmsr instruction.

The value of len is stored in bits 16 to 18 of fpscr. The value of len is not directly stored directly in these bits. Instead, we have to subtract 1 before setting the bits. This is because lencannot be 0 (it does not make sense to operate 0 floating points). This way the value 000 in these bits means len = 1, 001 means len = 2, ..., 111 means len = 8. The following is a code that sets len to 8.

```
/* Set the len field of fpscr to be 8 (bits: 111) */
mov r5, #7                          /* r5 ← 7. 7 is 111 in binary */
mov r5, r5, LSL #16                 /* r5 ← r5 << 16 */
vmrs r4, fpscr                      /* r4 ← fpscr */
orr r4, r4, r5                      /* r4 ← r4 | r5. Bitwise OR */
vmsr fpscr, r4                      /* fpscr ← r4 */
```

stride is stored in bits 20 to 21 of fpscr. Similar to len, a value of 00 in these bits means stride = 1, 01 means stride = 2, 10 means stride = 3 and 11 means stride = 4.

Function call convention and floating-point registers

Since we have introduced new registers we should state how to use them when calling functions. The following rules apply for VFPv2 registers.

- Fields len and stride of fpscr have all their bits as zero at the entry of a function and those bits must be zero when leaving it.
- We can pass floating point parameters using registers s0–s15 and d0–d7. Note that passing a double-precision after a single-precision may involve discarding an odd-numbered single-precision register (for instance we can use s0, and d1 but note that s1will be unused).
- All other floating point registers (s16–s31 and d8–d15) must have their values preserved upon leaving the function. Instructions vpush and vpop can be used for that.
- If a function returns a floating-point value, the return register will be s0 or d0.

Finally a note about variadic functions like printf: you cannot pass a single-precision floating point to one of such functions. Only doubles can be passed. So you will need to convert the single-precision values into double-precision values. Note also that usual integer registers are used (r0–r3), so you will only be able to pass up to 2 double-precision values, the remaining must be passed on the stack. In particular for printf, since r0 contains the address of the string format, you will only be able to pass a double-precision in {r2,r3}.

Assembler

Make sure you pass the flag -mfpu=vfpv2 to as, otherwise it will not recognize the VFPv2 instructions.

Colophon

You may want to check this official quick reference card of VFP. Note that it includes also VFPv3 not available in the Raspberry Pi processor. Most of what is there has already been presented here although some minor details may have been omitted.
In the next chapter we will use these instructions in a full example.

That’s all for today.

ARM assembler in Raspberry Pi – Chapter 14

May 12, 2013 Roger Ferrer Ibáñez, 18

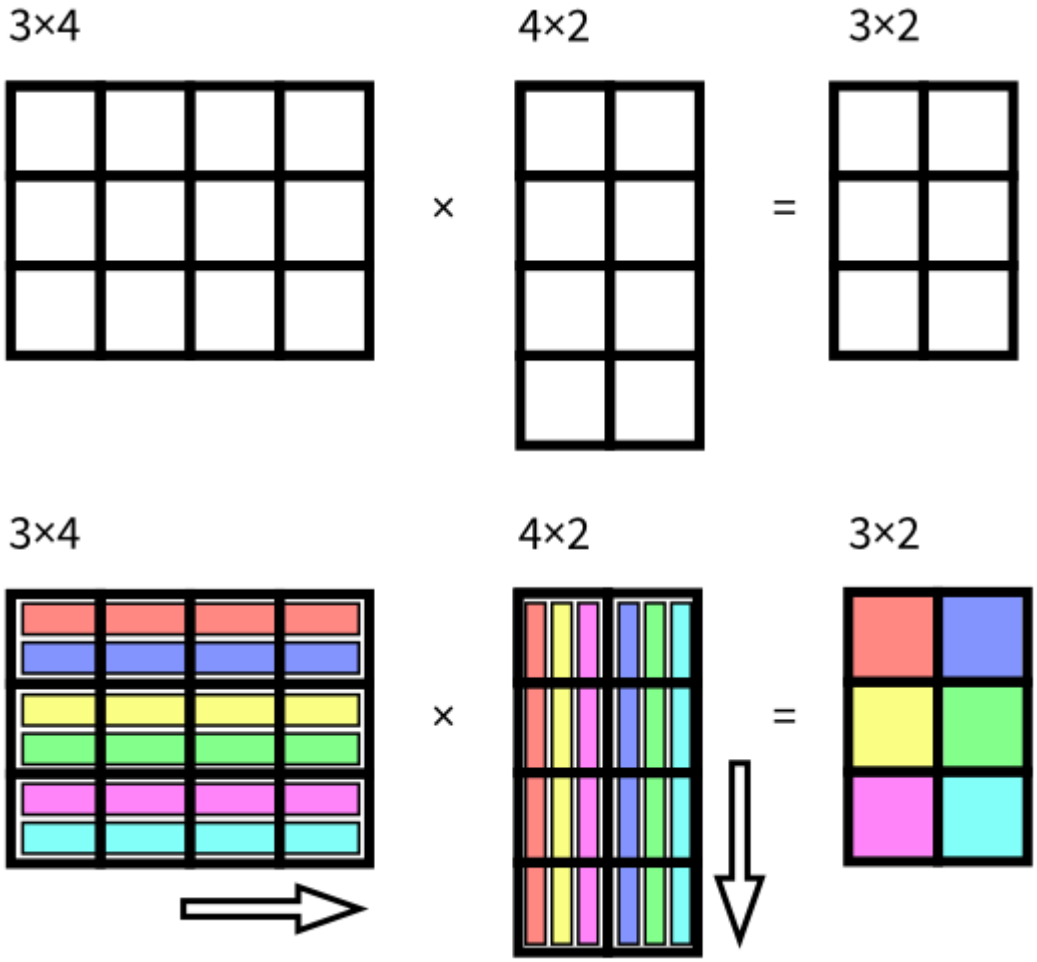
In chapter 13 we saw the basic elements of VFPv2, the floating point subarchitecture of ARMv6. In this chapter we will implement a floating point matrix multiply using VFPv2.

Disclaimer: I advise you against using the code in this chapter in commercial-grade projects unless you fully review it for both correctness and precision.

Matrix multiply

Given two vectors v and w of rank r where v = <v0, v1, ... vr-1> and w = <w0, w1, ..., wr-1>, we define the dot product of v by w as the scalar $v \cdot w = v0 \times w0 + v1 \times w1 + ... + vr-1 \times wr-1$.

We can multiply a matrix A of n rows and m columns (n x m) by a matrix B of m rows and p columns (m x p). The result is a matrix of n rows and p columns. Matrix multiplication may seem complicated but actually it is not. Every element in the result matrix it is just the dot product (defined in the paragraph above) of the corresponding row of the matrix A by the corresponding column of the matrix B (this is why there must be as many columns in A as there are rows in B).



A straightforward implementation of the matrix multiplication in C is as follows.

```c
float A[N][M]; // N rows of M columns each row
float B[M][P]; // M rows of P columns each row
// Result
float C[N][P];

```

```
for (int i = 0; i < N; i++) // for each row of the result
{
  for (int j = 0; j < P; j++) // and for each column
  {
    C[i][j] = 0; // Initialize to zero
    // Now make the dot matrix of the row by the column
    for (int k = 0; k < M; k++)
       C[i][j] += A[i][k] * B[k][j];
  }
}
```

In order to simplify the example, we will asume that both matrices A and B are square matrices of size n x n. This simplifies just a bit the algorithm.

```
float A[N][N];
float B[N][N];
// Result
float C[N][N];

for (int i = 0; i < N; i++)
{
  for (int j = 0; j < N; j++)
  {
    C[i][j] = 0;
    for (int k = 0; k < N; k++)
       C[i][j] += A[i][k] * B[k][j];
  }
}
```

Matrix multiplication is an important operation used in many areas. For instance, in computer graphics is usually performed on 3×3 and 4×4 matrices representing 3D geometry. So we will try to make a reasonably fast version of it (we do not aim at making the best one, though).

A first improvement we want to do in this algorithm is making the loops perfectly nested. There are some technical reasons beyond the scope of this code for that. So we will get rid of the initialization of C[i][j] to 0, outside of the loop.

```
1
2
3  float A[N][N];
4  float B[M][N];
5  // Result
6  float C[N][N];
7
8  for (int i = 0; i < N; i++)
9    for (int j = 0; j < N; j++)
1      C[i][j] = 0;
0
1  for (int i = 0; i < N; i++)
1    for (int j = 0; j < N; j++)
1      for (int k = 0; k < N; k++)
2        C[i][j] += A[i][k] * B[k][j];
1
3
```

After this change, the interesting part of our algorithm, line 13, is inside a perfect nest of loops of depth 3.

Accessing a matrix

It is relatively straightforward to access an array of just one dimension, like in a[i]. Just get i, multiply it by the size in bytes of each element of the array and then add the address of a (the base address of the array). So, the address of a[i] is just a + ELEMENTSIZE*i.

Things get a bit more complicated when our array has more than one dimension, like a matrix or a cube. Given an access like a[i][j][k] we have to compute which element is denoted by [i][j][k]. This depends on whether the language is row-major order or column-major order. We assume row-major order here (like in C language). So [i][j][k] must denote k + j * NK + i * NK * NJ, where NK and NJ are the number of elements in every dimension. For instance, a three dimensional array of 3 x 4 x 5 elements, the element [1][2][3] is 3 + 2 * 5 + 1 * 5 * 4 = 23 (here NK = 5 and NJ = 4.

Note that NI = 3 but we do not need it at all). We assume that our language indexes arrays starting from 0 (like C). If the language allows a lower bound other than 0, we first have to subtract the lower bound to get the position.
We can compute the position in a slightly better way if we reorder it. Instead of calculating k + j * NK + i * NK * NJ we will do k + NK * (j + NJ * i). This way all the calculus is just a repeated set of steps calculating x + Ni * y like in the example below.

```
/* Calculating the address of C[i][j][k] declared as int C[3][4][5] */
/* &C[i][j][k] is, thus, C + ELEMENTSIZE * ( k + NK * (j + NJ * i) ) */
// Assume i is in r4, j in r5 and k in r6 and the base address of C in r3 */
mov r8, #4         /* r8 ← NJ (Recall that NJ = 4) */
mul r7, r8, r4     /* r7 ← NJ * i */
add r7, r5, r7     /* r7 ← j + NJ * i */
mov r8, #5         /* r8 ← NK (Recall that NK = 5) */
mul r7, r8, r7     /* r7 ← NK * (j + NJ * i) */
add r7, r6, r7     /* r7 ← k + NK * (j + NJ + i) */
mov r8, #4         /* r8 ← ELEMENTSIZE (Recall that size of an int is 4 bytes) */
mul r7, r8, r7     /* r7 ← ELEMENTSIZE * ( k + NK * (j + NJ * i) ) */
add r7, r3, r7     /* r7 ← C + ELEMENTSIZE * ( k + NK * (j + NJ * i) ) */
```

Naive matrix multiply of 4×4 single-precision

As a first step, let’s implement a naive matrix multiply that follows the C algorithm above according to the letter.

```
1  /* -- matmul.s */
2  .data
3  mat_A: .float 0.1, 0.2, 0.0, 0.1
4         .float 0.2, 0.1, 0.3, 0.0
5         .float 0.0, 0.3, 0.1, 0.5
6         .float 0.0, 0.6, 0.4, 0.1
7  mat_B: .float  4.92,  2.54, -0.63, -1.75
8         .float  3.02, -1.51, -0.87,  1.35
9         .float -4.29,  2.14,  0.71,  0.71
1         .float -0.95,  0.48,  2.38, -0.95
0  mat_C: .float 0.0, 0.0, 0.0, 0.0
1         .float 0.0, 0.0, 0.0, 0.0
1         .float 0.0, 0.0, 0.0, 0.0
1
```

```
       .float 0.0, 0.0, 0.0, 0.0
       .float 0.0, 0.0, 0.0, 0.0

format_result : .asciz "Matrix result is:\n%5.2f %5.2f %5.2f %5.2f\n%5.2f %5.2f %5.2f %5.
2f\n%5.2f %5.2f %5.2f %5.2f\n%5.2f %5.2f %5.2f %5.2f\n"

.text

naive_matmul_4x4:
    /* r0 address of A
       r1 address of B
       r2 address of C
    */
    push {r4, r5, r6, r7, r8, lr} /* Keep integer registers */
    /* First zero 16 single floating point */
    /* In IEEE 754, all bits cleared means 0 */
    mov r4, r2
    mov r5, #16
    mov r6, #0
    b .Lloop_init_test
    .Lloop_init :
      str r6, [r4], +#4   /* *r4 ← r6 then r4 ← r4 + 4 */
    .Lloop_init_test:
      subs r5, r5, #1
      bge .Lloop_init

    /* We will use
           r4 as i
           r5 as j
           r6 as k
    */
    mov r4, #0 /* r4 ← 0 */
    .Lloop_i:  /* loop header of i */
      cmp r4, #4  /* if r4 == 4 goto end of the loop i */
      beq .Lend_loop_i
      mov r5, #0  /* r5 ← 0 */
      .Lloop_j: /* loop header of j */
```

```
        cmp r5, #4 /* if r5 == 4 goto end of the loop j */
         beq .Lend_loop_j
         /* Compute the address of C[i][j] and load it into s0 */
         /* Address of C[i][j] is C + 4*(4 * i + j) */
         mov r7, r5                /* r7 ← r5. This is r7 ← j */
         adds r7, r7, r4, LSL #2   /* r7 ← r7 + (r4 << 2).
                                      This is r7 ← j + i * 4.
                                      We multiply i by the row size (4 elements) */
         adds r7, r2, r7, LSL #2   /* r7 ← r2 + (r7 << 2).
                                      This is r7 ← C + 4*(j + i * 4)
                                      We multiply (j + i * 4) by the size of the element.
                                      A single-precision floating point takes 4 bytes.
                                      */
         vldr s0, [r7] /* s0 ← *r7 */

         mov r6, #0 /* r6 ← 0 */
         .Lloop_k :  /* loop header of k */
           cmp r6, #4 /* if r6 == 4 goto end of the loop k */
           beq .Lend_loop_k

           /* Compute the address of a[i][k] and load it into s1 */
           /* Address of a[i][k] is a + 4*(4 * i + k) */
           mov r8, r6                /* r8 ← r6. This is r8 ← k */
           adds r8, r8, r4, LSL #2   /* r8 ← r8 + (r4 << 2). This is r8 ← k + i * 4 */
           adds r8, r0, r8, LSL #2   /* r8 ← r0 + (r8 << 2). This is r8 ← a + 4*(k + i * 4) */
           vldr s1, [r8]             /* s1 ← *r8 */

           /* Compute the address of b[k][j] and load it into s2 */
           /* Address of b[k][j] is b + 4*(4 * k + j) */
           mov r8, r5                /* r8 ← r5. This is r8 ← j */
           adds r8, r8, r6, LSL #2   /* r8 ← r8 + (r6 << 2). This is r8 ← j + k * 4 */
           adds r8, r1, r8, LSL #2   /* r8 ← r1 + (r8 << 2). This is r8 ← b + 4*(j + k * 4) */
           vldr s2, [r8]             /* s1 ← *r8 */

           vmul.f32 s3, s1, s2       /* s3 ← s1 * s2 */
           vadd.f32 s0, s0, s3       /* s0 ← s0 + s3 */

```

```
          add r6, r6, #1           /* r6 ← r6 + 1 */
          b .Lloop_k               /* next iteration of loop k */
        .Lend_loop_k: /* Here ends loop k */
        vstr s0, [r7]            /* Store s0 back to C[i][j] */
        add r5, r5, #1  /* r5 ← r5 + 1 */
        b .Lloop_j /* next iteration of loop j */
       .Lend_loop_j: /* Here ends loop j */
       add r4, r4, #1 /* r4 ← r4 + 1 */
       b .Lloop_i     /* next iteration of loop i */
    .Lend_loop_i: /* Here ends loop i */

    pop {r4, r5, r6, r7, r8, lr}  /* Restore integer registers */
    bx lr /* Leave function */


.globl main
main:
    push {r4, r5, r6, lr}  /* Keep integer registers */
    vpush {d0-d1}          /* Keep floating point registers */

    /* Prepare call to naive_matmul_4x4 */
    ldr r0, addr_mat_A  /* r0 ← a */
    ldr r1, addr_mat_B  /* r1 ← b */
    ldr r2, addr_mat_C  /* r2 ← c */
    bl naive_matmul_4x4

    /* Now print the result matrix */
    ldr r4, addr_mat_C  /* r4 ← c */

    vldr s0, [r4] /* s0 ← *r4. This is s0 ← c[0][0] */
    vcvt.f64.f32 d1, s0 /* Convert it into a double-precision
                           d1 ← s0
                        */
    vmov r2, r3, d1      /* {r2,r3} ← d1 */

    mov r6, sp     /* Remember the stack pointer, we need it to restore it back later */
                   /* r6 ← sp */
```

```
99
77
00
77
11
77
22
77
33
77
44
77
55
77
66
77
77
77
88
77
99
88
00
88
11
88
22
88
33
88
44
88
55
```

```
    mov r5, #1  /* We will iterate from 1 to 15 (because the 0th item has already been handled */
    add r4, r4, #60 /* Go to the last item of the matrix c, this is c[3][3] */
    .Lloop:
        vldr s0, [r4] /* s0 ← *r4. Load the current item */
        vcvt.f64.f32 d1, s0 /* Convert it into a double-precision
                               d1 ← s0
                            */
        sub sp, sp, #8      /* Make room in the stack for the double-precision */
        vstr d1, [sp]       /* Store the double precision in the top of the stack */
        sub r4, r4, #4      /* Move to the previous element in the matrix */
        add r5, r5, #1      /* One more item has been handled */
        cmp r5, #16         /* if r5 != 16 go to next iteration of the loop */
        bne .Lloop

    ldr r0, addr_format_result /* r0 ← &format_result */
    bl printf /* call printf */
    mov sp, r6  /* Restore the stack after the call  */

    mov r0, #0
    vpop {d0-d1}
    pop {r4, r5, r6, lr}
    bx lr

addr_mat_A : .word mat_A
addr_mat_B : .word mat_B
addr_mat_C : .word mat_C
addr_format_result : .word format_result
```

That’s a lot of code but it is not complicated. Unfortunately computing the address of the array takes an important number of instructions. In our naive_matmul_4x4 we have the three loops i, j and k of the C algorithm. We compute the address of C[i][j] in the loop j (there is no need to compute it every time in the loop k) in lines 52 to 63. The value

contained in C[i][j] is then loaded into s0. In each iteration of loop k we load A[i][k] and B[k][j] in s1 and s2respectively (lines 70 to 82). After the loop k ends, we can store s0 back to the array position (kept in r7, line 90)
In order to print the result matrix we have to pass 16 floating points to printf. Unfortunately, as stated in chapter 13, we have to first convert them into double-precision before passing them. Note also that the first single-precision can be passed using registers r2 and r3. All the remaining must be passed on the stack and do not forget that the stack parameters must be passed in opposite order. This is why once the first element of the C matrix has been loaded in {r2,r3}(lines 117 to 120) we advance 60 bytes r4. This is C[3][3], the last element of the matrix C. We load the single-precision, convert it into double-precision, push it in the stack and then move backwards register r4, to the previous element in the matrix (lines 128 to 137). Observe that we use r6 as a marker of the stack, since we need to restore the stack once printf returns (line 122 and line 141). Of course we could avoid using r6 and instead do add sp, sp, #120 since this is exactly the amount of bytes we push to the stack (15 values of double-precision, each taking 8 bytes).
I have not chosen the values of the two matrices randomly. The second one is (approximately) the inverse of the first. This way we will get the identity matrix (a matrix with all zeros but a diagonal of ones). Due to rounding issues the result matrix will not be the identity, but it will be pretty close. Let's run the program.

```
$ ./matmul
Matrix result is:
  1.00 -0.00  0.00  0.00
-0.00  1.00  0.00 -0.00
  0.00  0.00  1.00  0.00
  0.00 -0.00  0.00  1.00
```

Vectorial approach

The algorithm we are trying to implement is fine but it is not the most optimizable. The problem lies in the way the loop k accesses the elements. Access A[i][k] is eligible for a multiple load as A[i][k] and A[i][k+1] are contiguous elements in memory. This way we could entirely avoid all the loop k and perform a 4 element load from A[i][0] to A[i][3]. The access B[k][j] does not allow that since elements B[k][j] and B[k+1][j] have a full row inbetween. This is a strided access (the stride here is a full row of 4 elements, this is 16 bytes), VFPv2 does not allow a strided multiple load, so we

will have to load one by one.. Once we have all the elements of the loop k loaded, we can do a vector multiplication and a sum.

```
naive_vectorial_matmul_4x4:
    /* r0 address of A
       r1 address of B
       r2 address of C
    */
    push {r4, r5, r6, r7, r8, lr} /* Keep integer registers */
    vpush {s16-s19}               /* Floating point registers starting from s16 must be preserved
*/
    vpush {s24-s27}
    /* First zero 16 single floating point */
    /* In IEEE 754, all bits cleared means 0 */
    mov r4, r2
    mov r5, #16
    mov r6, #0
    b .L1_loop_init_test
    .L1_loop_init :
      str r6, [r4], +#4   /* *r4 ← r6 then r4 ← r4 + 4 */
    .L1_loop_init_test:
      subs r5, r5, #1
      bge .L1_loop_init

    /* Set the LEN field of FPSCR to be 4 (value 3) */
    mov r5, #0b011                        /* r5 ← 3 */
    mov r5, r5, LSL #16                   /* r5 ← r5 << 16 */
    fmrx r4, fpscr                        /* r4 ← fpscr */
    orr r4, r4, r5                        /* r4 ← r4 | r5 */
    fmxr fpscr, r4                        /* fpscr ← r4 */

    /* We will use
           r4 as i
           r5 as j
           r6 as k
    */
    mov r4, #0 /* r4 ← 0 */
```

```
    .L1_loop_i:  /* loop header of i */
      cmp r4, #4  /* if r4 == 4 goto end of the loop i */
      beq .L1_end_loop_i
      mov r5, #0  /* r5 ← 0 */
      .L1_loop_j: /* loop header of j */
       cmp r5, #4 /* if r5 == 4 goto end of the loop j */
        beq .L1_end_loop_j
        /* Compute the address of C[i][j] and load it into s0 */
        /* Address of C[i][j] is C + 4*(4 * i + j) */
        mov r7, r5                /* r7 ← r5. This is r7 ← j */
        adds r7, r7, r4, LSL #2  /* r7 ← r7 + (r4 << 2).
                                    This is r7 ← j + i * 4.
                                    We multiply i by the row size (4 elements) */
        adds r7, r2, r7, LSL #2  /* r7 ← r2 + (r7 << 2).
                                    This is r7 ← C + 4*(j + i * 4)
                                    We multiply (j + i * 4) by the size of the element.
                                    A single-precision floating point takes 4 bytes.
                                    */
        /* Compute the address of a[i][0] */
        mov r8, r4, LSL #2
        adds r8, r0, r8, LSL #2
        vldmia r8, {s8-s11}  /* Load {s8,s9,s10,s11} ← {a[i][0], a[i][1], a[i][2], a[i][3]} */

        /* Compute the address of b[0][j] */
        mov r8, r5                /* r8 ← r5. This is r8 ← j */
        adds r8, r1, r8, LSL #2  /* r8 ← r1 + (r8 << 2). This is r8 ← b + 4*(j) */
        vldr s16, [r8]             /* s16 ← *r8. This is s16 ← b[0][j] */
        vldr s17, [r8, #16]        /* s17 ← *(r8 + 16). This is s17 ← b[1][j] */
        vldr s18, [r8, #32]        /* s18 ← *(r8 + 32). This is s17 ← b[2][j] */
        vldr s19, [r8, #48]        /* s19 ← *(r8 + 48). This is s17 ← b[3][j] */

        vmul.f32 s24, s8, s16      /* {s24,s25,s26,s27} ← {s8,s9,s10,s11} * {s16,s17,s18,s19} */
        vmov.f32 s0, s24           /* s0 ← s24 */
        vadd.f32 s0, s0, s25       /* s0 ← s0 + s25 */
        vadd.f32 s0, s0, s26       /* s0 ← s0 + s26 */
        vadd.f32 s0, s0, s27       /* s0 ← s0 + s27 */

```

```
        vstr s0, [r7]            /* Store s0 back to C[i][j] */
        add r5, r5, #1  /* r5 ← r5 + 1 */
        b .L1_loop_j /* next iteration of loop j */
       .L1_end_loop_j: /* Here ends loop j */
       add r4, r4, #1 /* r4 ← r4 + 1 */
       b .L1_loop_i     /* next iteration of loop i */
    .L1_end_loop_i: /* Here ends loop i */

    /* Set the LEN field of FPSCR back to 1 (value 0) */
    mov r5, #0b011                      /* r5 ← 3 */
    mvn r5, r5, LSL #16                 /* r5 ← ~(r5 << 16) */
    fmrx r4, fpscr                      /* r4 ← fpscr */
    and r4, r4, r5                      /* r4 ← r4 & r5 */
    fmxr fpscr, r4                      /* fpscr ← r4 */

    vpop {s24-s27}                /* Restore preserved floating registers */
    vpop {s16-s19}
    pop {r4, r5, r6, r7, r8, lr}  /* Restore integer registers */
    bx lr /* Leave function */
```

With this approach we can entirely remove the loop k, as we do 4 operations at once. Note that we have to modify fpscr so the field len is set to 4 (and restore it back to 1 when leaving the function).

Fill the registers

In the previous version we are not exploiting all the registers of VFPv2. Each rows takes 4 registers and so does each column, so we end using only 8 registers plus 4 for the result and one in the bank 0 for the summation. We got rid the loop k to process C[i][j] at once. What if we processed C[i][j] and C[i][j+1] at the same time? This way we can fill all the 8 registers in each bank.

```
naive_vectorial_matmul_2_4x4:
    /* r0 address of A
       r1 address of B
```

```
      r2 address of C
   */
   push {r4, r5, r6, r7, r8, lr} /* Keep integer registers */
   vpush {s16-s31}               /* Floating point registers starting from s16 must be preserved
*/
   /* First zero 16 single floating point */
   /* In IEEE 754, all bits cleared means 0 */
   mov r4, r2
   mov r5, #16
   mov r6, #0
   b .L2_loop_init_test
   .L2_loop_init :
     str r6, [r4], +#4   /* *r4 ← r6 then r4 ← r4 + 4 */
   .L2_loop_init_test:
     subs r5, r5, #1
     bge .L2_loop_init

   /* Set the LEN field of FPSCR to be 4 (value 3) */
   mov r5, #0b011                        /* r5 ← 3 */
   mov r5, r5, LSL #16                   /* r5 ← r5 << 16 */
   fmrx r4, fpscr                        /* r4 ← fpscr */
   orr r4, r4, r5                        /* r4 ← r4 | r5 */
   fmxr fpscr, r4                        /* fpscr ← r4 */

   /* We will use
         r4 as i
         r5 as j
   */
   mov r4, #0 /* r4 ← 0 */
   .L2_loop_i:  /* loop header of i */
     cmp r4, #4  /* if r4 == 4 goto end of the loop i */
     beq .L2_end_loop_i
     mov r5, #0  /* r5 ← 0 */
     .L2_loop_j: /* loop header of j */
      cmp r5, #4 /* if r5 == 4 goto end of the loop j */
       beq .L2_end_loop_j
       /* Compute the address of C[i][j] and load it into s0 */
```

```
        /* Address of C[i][j] is C + 4*(4 * i + j) */
        mov r7, r5                /* r7 ← r5. This is r7 ← j */
        adds r7, r7, r4, LSL #2   /* r7 ← r7 + (r4 << 2).
                                     This is r7 ← j + i * 4.
                                     We multiply i by the row size (4 elements) */
        adds r7, r2, r7, LSL #2   /* r7 ← r2 + (r7 << 2).
                                     This is r7 ← C + 4*(j + i * 4)
                                     We multiply (j + i * 4) by the size of the element.
                                     A single-precision floating point takes 4 bytes.
                                     */
        /* Compute the address of a[i][0] */
        mov r8, r4, LSL #2
        adds r8, r0, r8, LSL #2
        vldmia r8, {s8-s11}  /* Load {s8,s9,s10,s11} ← {a[i][0], a[i][1], a[i][2], a[i][3]} */

        /* Compute the address of b[0][j] */
        mov r8, r5                /* r8 ← r5. This is r8 ← j */
        adds r8, r1, r8, LSL #2   /* r8 ← r1 + (r8 << 2). This is r8 ← b + 4*(j) */
        vldr s16, [r8]             /* s16 ← *r8. This is s16 ← b[0][j] */
        vldr s17, [r8, #16]        /* s17 ← *(r8 + 16). This is s17 ← b[1][j] */
        vldr s18, [r8, #32]        /* s18 ← *(r8 + 32). This is s17 ← b[2][j] */
        vldr s19, [r8, #48]        /* s19 ← *(r8 + 48). This is s17 ← b[3][j] */

        /* Compute the address of b[0][j+1] */
        add r8, r5, #1             /* r8 ← r5 + 1. This is r8 ← j + 1*/
        adds r8, r1, r8, LSL #2    /* r8 ← r1 + (r8 << 2). This is r8 ← b + 4*(j + 1) */
        vldr s20, [r8]             /* s20 ← *r8. This is s20 ← b[0][j + 1] */
        vldr s21, [r8, #16]        /* s21 ← *(r8 + 16). This is s21 ← b[1][j + 1] */
        vldr s22, [r8, #32]        /* s22 ← *(r8 + 32). This is s22 ← b[2][j + 1] */
        vldr s23, [r8, #48]        /* s23 ← *(r8 + 48). This is s23 ← b[3][j + 1] */

        vmul.f32 s24, s8, s16      /* {s24,s25,s26,s27} ← {s8,s9,s10,s11} * {s16,s17,s18,s19} */
        vmov.f32 s0, s24           /* s0 ← s24 */
        vadd.f32 s0, s0, s25       /* s0 ← s0 + s25 */
        vadd.f32 s0, s0, s26       /* s0 ← s0 + s26 */
        vadd.f32 s0, s0, s27       /* s0 ← s0 + s27 */
```

```
    vmul.f32 s28, s8, s20      /* {s28,s29,s30,s31} ← {s8,s9,s10,s11} * {s20,s21,s22,s23} */

    vmov.f32 s1, s28           /* s1 ← s28 */
    vadd.f32 s1, s1, s29       /* s1 ← s1 + s29 */
    vadd.f32 s1, s1, s30       /* s1 ← s1 + s30 */
    vadd.f32 s1, s1, s31       /* s1 ← s1 + s31 */

    vstmia r7, {s0-s1}         /* {C[i][j], C[i][j+1]} ← {s0, s1} */

    add r5, r5, #2  /* r5 ← r5 + 2 */
    b .L2_loop_j /* next iteration of loop j */
   .L2_end_loop_j: /* Here ends loop j */
   add r4, r4, #1 /* r4 ← r4 + 1 */
   b .L2_loop_i     /* next iteration of loop i */
.L2_end_loop_i: /* Here ends loop i */

/* Set the LEN field of FPSCR back to 1 (value 0) */
mov r5, #0b011                      /* r5 ← 3 */
mvn r5, r5, LSL #16                 /* r5 ← ~(r5 << 16) */
fmrx r4, fpscr                      /* r4 ← fpscr */
and r4, r4, r5                      /* r4 ← r4 & r5 */
fmxr fpscr, r4                      /* fpscr ← r4 */

vpop {s16-s31}                /* Restore preserved floating registers */
pop {r4, r5, r6, r7, r8, lr}  /* Restore integer registers */
bx lr /* Leave function */
```

Note that because we now process j and j + 1, r5 (j) is now increased by 2 at the end of the loop. This is usually known as loop unrolling and it is always legal to do. We do more than one iteration of the original loop in the unrolled loop. The amount of iterations of the original loop we do in the unrolled loop is the unroll factor. In this case since the number of

iterations (4) perfectly divides the unrolling factor (2) we do not need an extra loop for the remainder iterations (the remainder loop has one less iteration than the value of the unrolling factor).
As you can see, the accesses to b[k][j] and b[k][j+1] are starting to become tedious. Maybe we should change a bit more the matrix multiply algorithm.

Reorder the accesses

Is there a way we can mitigate the strided accesses to the matrix B? Yes, there is one, we only have to permute the loop nest i, j, k into the loop nest k, i, j. Now you may be wondering if this is legal. Well, checking for the legality of these things is beyond the scope of this post so you will have to trust me here. Such permutation is fine. What does this mean? Well, it means that our algorithm will now look like this.

```
1
2
3  float A[N][N];
4  float B[M][N];
5  // Result
6  float C[N][N];
7
8  for (int i = 0; i < N; i++)
9    for (int j = 0; j < N; j++)
1      C[i][j] = 0;
0
1  for (int k = 0; k < N; k++)
1    for (int i = 0; i < N; i++)
1      for (int j = 0; j < N; j++)
2        C[i][j] += A[i][k] * B[k][j];
1
3
```

This may seem not very useful, but note that, since now k is in the outermost loop, now it is easier to use vectorial instructions.

```
for (int k = 0; k < N; k++)
  for (int i = 0; i < N; i++)
  {
```

```
    C[i][0] += A[i][k] * B[k][0];
    C[i][1] += A[i][k] * B[k][1];
    C[i][2] += A[i][k] * B[k][2];
    C[i][3] += A[i][k] * B[k][3];
  }
```

If you remember the chapter 13, VFPv2 instructions have a mixed mode when the Rsource2register is in bank 0. This case makes a perfect match: we can load C[i][0..3] and B[k][0..3] with a load multiple and then load A[i][k] in a register of the bank 0. Then we can make multiply A[i][k]*B[k][0..3] and add the result to C[i][0..3]. As a bonus, the number of instructions is much lower.

```
better_vectorial_matmul_4x4:
    /* r0 address of A
       r1 address of B
       r2 address of C
    */
    push {r4, r5, r6, r7, r8, lr} /* Keep integer registers */
    vpush {s16-s19}               /* Floating point registers starting from s16 must be preserved */
    vpush {s24-s27}
    /* First zero 16 single floating point */
    /* In IEEE 754, all bits cleared means 0 */
    mov r4, r2
    mov r5, #16
    mov r6, #0
    b .L3_loop_init_test
    .L3_loop_init :
      str r6, [r4], +#4   /* *r4 ← r6 then r4 ← r4 + 4 */
    .L3_loop_init_test:
      subs r5, r5, #1
      bge .L3_loop_init

    /* Set the LEN field of FPSCR to be 4 (value 3) */
    mov r5, #0b011                        /* r5 ← 3 */
    mov r5, r5, LSL #16                   /* r5 ← r5 << 16 */
    fmrx r4, fpscr                        /* r4 ← fpscr */
    orr r4, r4, r5                        /* r4 ← r4 | r5 */
    fmxr fpscr, r4                        /* fpscr ← r4 */
```

```
1
9     /* We will use
2             r4 as k
0             r5 as i
2     */
1     mov r4, #0 /* r4 ← 0 */
2     .L3_loop_k:  /* loop header of k */
2       cmp r4, #4  /* if r4 == 4 goto end of the loop k */
2       beq .L3_end_loop_k
3       mov r5, #0  /* r5 ← 0 */
2       .L3_loop_i: /* loop header of i */
4        cmp r5, #4 /* if r5 == 4 goto end of the loop i */
2         beq .L3_end_loop_i
5         /* Compute the address of C[i][0] */
2         /* Address of C[i][0] is C + 4*(4 * i) */
6         add r7, r2, r5, LSL #4          /* r7 ← r2 + (r5 << 4). This is r7 ← c + 4*4*i */
2         vldmia r7, {s8-s11}             /* Load {s8,s9,s10,s11} ← {c[i][0], c[i][1], c[i][2], c[i][3]}
7 */
2         /* Compute the address of A[i][k] */
8         /* Address of A[i][k] is A + 4*(4*i + k) */
2         add r8, r4, r5, LSL #2          /* r8 ← r4 + r5 << 2. This is r8 ← k + 4*i */
9         add r8, r0, r8, LSL #2          /* r8 ← r0 + r8 << 2. This is r8 ← a + 4*(k + 4*i) */
3         vldr s0, [r8]                   /* Load s0 ← a[i][k] */
0
3
1         /* Compute the address of B[k][0] */
3         /* Address of B[k][0] is B + 4*(4*k) */
2         add r8, r1, r4, LSL #4          /* r8 ← r1 + r4 << 4. This is r8 ← b + 4*(4*k) */
3          vldmia r8, {s16-s19}             /* Load {s16,s17,s18,s19} ← {b[k][0], b[k][1], b[k][2], b[k]
3 [3]} */
3
4         vmul.f32 s24, s16, s0           /* {s24,s25,s26,s27} ← {s16,s17,s18,s19} * {s0,s0,s0,s0} */
3         vadd.f32 s8, s8, s24            /* {s8,s9,s10,s11} ← {s8,s9,s10,s11} + {s24,s25,s26,s7} */
5
3             vstmia  r7,  {s8-s11}                   /*  Store  {c[i][0],  c[i][1],  c[i][2],  c[i][3]} ←
6 {s8,s9,s10,s11} */
3
7         add r5, r5, #1  /* r5 ← r5 + 1. This is i = i + 1 */
```

```
        b .L3_loop_i /* next iteration of loop i */
       .L3_end_loop_i: /* Here ends loop i */
       add r4, r4, #1 /* r4 ← r4 + 1. This is k = k + 1 */
       b .L3_loop_k     /* next iteration of loop k */
    .L3_end_loop_k: /* Here ends loop k */

    /* Set the LEN field of FPSCR back to 1 (value 0) */
    mov r5, #0b011                          /* r5 ← 3 */
    mvn r5, r5, LSL #16                     /* r5 ← ~(r5 << 16) */
    fmrx r4, fpscr                          /* r4 ← fpscr */
    and r4, r4, r5                          /* r4 ← r4 & r5 */
    fmxr fpscr, r4                          /* fpscr ← r4 */

    vpop {s24-s27}                /* Restore preserved floating registers */
    vpop {s16-s19}
    pop {r4, r5, r6, r7, r8, lr}  /* Restore integer registers */
    bx lr /* Leave function */
```

As adding after a multiplication is a relatively usual sequence, we can replace the sequence

```
vmul.f32 s24, s16, s0          /* {s24,s25,s26,s27} ← {s16,s17,s18,s19} * {s0,s0,s0,s0} */
vadd.f32 s8, s8, s24           /* {s8,s9,s10,s11} ← {s8,s9,s10,s11} + {s24,s25,s26,s7} */
```

with a single instruction vmla (multiply and add).

```
vmla.f32 s8, s16, s0          /* {s8,s9,s10,s11} ← {s8,s9,s10,s11} + ({s16,s17,s18,s19} * {s0,s0,s0,s0}) */
```

Now we can also unroll the loop i, again with an unrolling factor of 2. This would give us the bestversion.

```
best_vectorial_matmul_4x4:
    /* r0 address of A
       r1 address of B
```

```
      r2 address of C
   */
   push {r4, r5, r6, r7, r8, lr} /* Keep integer registers */
   vpush {s16-s19}               /* Floating point registers starting from s16 must be preserved */

   /* First zero 16 single floating point */
   /* In IEEE 754, all bits cleared means 0 */
   mov r4, r2
   mov r5, #16
   mov r6, #0
   b .L4_loop_init_test
   .L4_loop_init :
     str r6, [r4], +#4   /* *r4 ← r6 then r4 ← r4 + 4 */
   .L4_loop_init_test:
     subs r5, r5, #1
     bge .L4_loop_init

   /* Set the LEN field of FPSCR to be 4 (value 3) */
   mov r5, #0b011                        /* r5 ← 3 */
   mov r5, r5, LSL #16                   /* r5 ← r5 << 16 */
   fmrx r4, fpscr                        /* r4 ← fpscr */
   orr r4, r4, r5                        /* r4 ← r4 | r5 */
   fmxr fpscr, r4                        /* fpscr ← r4 */

   /* We will use
          r4 as k
          r5 as i
   */
   mov r4, #0 /* r4 ← 0 */
   .L4_loop_k:  /* loop header of k */
     cmp r4, #4  /* if r4 == 4 goto end of the loop k */
     beq .L4_end_loop_k
     mov r5, #0  /* r5 ← 0 */
     .L4_loop_i: /* loop header of i */
      cmp r5, #4 /* if r5 == 4 goto end of the loop i */
       beq .L4_end_loop_i
       /* Compute the address of C[i][0] */
```

```
        /* Address of C[i][0] is C + 4*(4 * i) */
        add r7, r2, r5, LSL #4          /* r7 ← r2 + (r5 << 4). This is r7 ← c + 4*4*i */
        vldmia r7, {s8-s15}             /* Load {s8,s9,s10,s11,s12,s13,s14,s15}
                                           ← {c[i][0],   c[i][1],   c[i][2],   c[i][3]
                                              c[i+1][0], c[i+1][1], c[i+1][2], c[i+1][3]} */
        /* Compute the address of A[i][k] */
        /* Address of A[i][k] is A + 4*(4*i + k) */
        add r8, r4, r5, LSL #2          /* r8 ← r4 + r5 << 2. This is r8 ← k + 4*i */
        add r8, r0, r8, LSL #2          /* r8 ← r0 + r8 << 2. This is r8 ← a + 4*(k + 4*i) */
        vldr s0, [r8]                   /* Load s0 ← a[i][k] */
        vldr s1, [r8, #16]              /* Load s1 ← a[i+1][k] */

        /* Compute the address of B[k][0] */
        /* Address of B[k][0] is B + 4*(4*k) */
        add r8, r1, r4, LSL #4          /* r8 ← r1 + r4 << 4. This is r8 ← b + 4*(4*k) */
        vldmia r8, {s16-s19}            /* Load {s16,s17,s18,s19} ← {b[k][0], b[k][1], b[k][2], b[k][3]} */

         vmla.f32 s8, s16, s0            /* {s8,s9,s10,s11} ← {s8,s9,s10,s11} + ({s16,s17,s18,s19} * {s0,s0,s0,s0})
*/
            vmla.f32  s12,  s16,  s1              /*  {s12,s13,s14,s15} ← {s12,s13,s14,s15}  +  ({s16,s17,s18,s19}  *
{s1,s1,s1,s1}) */

        vstmia r7, {s8-s15}             /* Store {c[i][0],   c[i][1],   c[i][2],    c[i][3],
                                                  c[i+1][0], c[i+1][1], c[i+1][2]}, c[i+1][3] }
                                               ← {s8,s9,s10,s11,s12,s13,s14,s15} */

        add r5, r5, #2  /* r5 ← r5 + 2. This is i = i + 2 */
        b .L4_loop_i /* next iteration of loop i */
       .L4_end_loop_i: /* Here ends loop i */
       add r4, r4, #1 /* r4 ← r4 + 1. This is k = k + 1 */
       b .L4_loop_k     /* next iteration of loop k */
    .L4_end_loop_k: /* Here ends loop k */

    /* Set the LEN field of FPSCR back to 1 (value 0) */
    mov r5, #0b011                      /* r5 ← 3 */
    mvn r5, r5, LSL #16                 /* r5 ← ~(r5 << 16) */
    fmrx r4, fpscr                      /* r4 ← fpscr */
```

```
    and r4, r4, r5                      /* r4 ← r4 & r5 */
    fmxr fpscr, r4                      /* fpscr ← r4 */

    vpop {s16-s19}                  /* Restore preserved floating registers */
    pop {r4, r5, r6, r7, r8, lr}  /* Restore integer registers */
    bx lr /* Leave function */
```

Comparing versions

Out of curiosity I tested the versions, to see which one was faster.

The benchmark consists on repeatedly calling the multiplication matrix function 221 times (actually 221-1 because of a typo, see the code) in order to magnify differences. While the input should be randomized as well for a better benchmark, the benchmark more or less models contexts where a matrix multiplication is performed many times (for instance in graphics).
This is the skeleton of the benchmark.

```
main:
    push {r4, lr}

    ldr r0, addr_mat_A  /* r0 ← a */
    ldr r1, addr_mat_B  /* r1 ← b */
    ldr r2, addr_mat_C  /* r2 ← c */
    mov r4, #1
    mov r4, r4, LSL #21
```

```
    .Lmain_loop_test:
      bl <<tested-matmul-routine>> /* Change here with the matmul you want to test */
      subs r4, r4, #1
      bne .Lmain_loop_test         /* I should have written 'bge' here,
                                      but I cannot change it without
                                      having to run the benchmarks again :) */

    mov r0, #0
    pop {r4, lr}
    bx lr
```

Here are the results. The one we named the best turned to actually deserve that name.

Version	Time (seconds)
naive_matmul_4x4	6.41
naive_vectorial_matmul_4x4	3.51
naive_vectorial_matmul_2_4x4	2.87
better_vectorial_matmul_4x4	2.59
best_vectorial_matmul_4x4	1.51

That’s all for today.

Check_MK, software updates and mount options alarms

June 8, 2013 brafales, 0

We have been using nagios (more specifically Check_MK) recently at work to get some monitoring information on our CentOS instances. Recently we decided to reprovision all of our EC2 instances to apply several security upgrades. Among the packages upgraded, there was the kernel (which I guess was the cause of our subsequent problems).
After updating all our instances, nagios began to complain about the mount options no being the right ones for the root file system, and started sending critical alarms. The file system was ok, it was mounted without problems and everything was woking fine, but for some reason the mount options had changed after the reprovisioning.

Turns out that Check_MK checks the options in place when it does the initial inventory, and if the options change over time, it issues an alarm. If you face this problem, just do a reinventory of your machines and reload the configuration and restart the service, and it should be fine:

cmk -II && cmk -R

acts_as_list gem and single table inheritance in Rails

July 10, 2013 brafales, 0

If you ever need to use the acts_as_list gem in Rails on a model that uses single table inheritance, here's the snippet you need to use for the list methods to work if you want the setup done on the base model:

```
acts_as_list    :scope    =>
[:type]
```

You'll need to use the array syntax as neither the string nor the symbol versions will work. The symbol one assumes the column ending in _id, while the string one will simply not allow you to evaluate the current object's type.

It'd be nice to have a lambda syntax in future versions of the gem so you can inject code into the conditions.

ARM assembler in Raspberry Pi – Chapter 15

August 11, 2013 Roger Ferrer Ibáñez, 10

It may be suprising, but the ARMv6 architecture does not provide an integer division instruction while it does have a floating point instruction in VFPv2. In this chapter we will see usual ways to workaround this limitation with different techniques that can be used in specific scenarios involving divisions.

What does integer division mean?

First we should clearly define what we understand by integer division. Given two integer numbers N (for numerator) and D (for denominator, different than zero) we define the integer division of N and D as the pair of integer numbers Q (for quotient) and R (for remainder) satisfying the following equality.

$N = D \times Q + R$ where $0 \leq |R| < |D|$

The equality implies that there are two solutions $0 < R < |D|$ and $0 < |-R| < |D|$. For example, N=7 and D=3 has two solutions (Q=2, R=1) and (Q=3, R=-2). While both solutions may be useful, the former one is the preferred as it is closer to our natural notion of the remainder. But what if D is negative? For example N=7 and D=-3 has two solutions as well (Q=-2, R=1) and (Q=-3, R=-2). When negative numbers are involved the choice of the remainder is not intuitive but conventional. Many conventions can be used to choose one solution. We can always pick the solution with the positive remainder (this is called euclidean division), or the negative, or the solution where the sign of the remainder matches the numerator (or the denominator).

Most computers perform an integer division where the remainder has the same sign as the numerator. So for N=7 and D=3 the computed solution is (Q=2, R=1) and for N=7 and D=-3the computed solution is (Q=-2, R=1). We will assume such integer division convention in the remainder (no pun intended) of this post.

Unsigned division

An unsigned integer division is an integer division involving two unsigned integers N and D. This has the consequence that Q and R will always be positive. A very naive (and slow) approach for unsigned division is as follows.

```
1 unsigned_naive_div:
```

```
    /* r0 contains N */
    /* r1 contains D */
    mov r2, r1              /* r2 ← r0. We keep D in r2 */
    mov r1, r0              /* r1 ← r0. We keep N in r1 */

    mov r0, #0              /* r0 ← 0. Set Q = 0 initially */

    b .Lloop_check
    .Lloop:
       add r0, r0, #1       /* r0 ← r0 + 1. Q = Q + 1 */
       sub r1, r1, r2       /* r1 ← r1 - r2 */
    .Lloop_check:
       cmp r1, r2           /* compute r1 - r2 */
       bhs .Lloop             /* branch if r1 >= r2 (C=0 or Z=1) */

    /* r0 already contains Q */
    /* r1 already contains R */
    bx lr
```

This algorithm, while correct and easy to understand, is not very efficient (think on dividing a big N with a small D). Is there a way that we can compute the division in a fixed amount of time? The answer is yes, just adapt how you divide by hand but to binary numbers. We will compute a temporary remainder picking bits, from left to right, of the dividend. When the remainder is larger than the divisor, we will subtract the divisor from that remainder and set the appropiate bit in the quotient.

```
unsigned_longdiv:
    /* r0 contains N */
```

```
    /* r1 contains D */
    /* r2 contains Q */
    /* r3 contains R */
    push {r4, lr}
    mov r2, #0                  /* r2 ← 0 */
    mov r3, #0                  /* r3 ← 0 */

    mov r4, #32                 /* r4 ← 32 */
    b .Lloop_check1
    .Lloop1:
        movs r0, r0, LSL #1     /* r0 ← r0 << 1 updating cpsr (sets C if 31st bit of r0 was 1) */
        adc r3, r3, r3          /* r3 ← r3 + r3 + C. This is equivalent to r3 ← (r3 << 1) + C */

        cmp r3, r1              /* compute r3 - r1 and update cpsr */
        subhs r3, r3, r1        /* if r3 >= r1 (C=1) then r3 ← r3 - r1 */
        adc r2, r2, r2          /* r2 ← r2 + r2 + C. This is equivalent to r2 ← (r2 << 1) + C */
    .Lloop_check1:
        subs r4, r4, #1         /* r4 ← r4 - 1 */
        bpl .Lloop1             /* if r4 >= 0 (N=0) then branch to .Lloop1 */

    pop {r4, lr}
    bx lr
```

This approach is a bit more efficient since it repeats the loop a fixed number of times (always 32). For each bit of N starting from the most significant one (line 13), we push it to the right of the current value of R (line 14). We do this by adding R to itself, this is 2*R which is actually shifting to the right R 1 bit. Then we add the carry, that will be 1 if the most significant bit of N before the shift (line 13) was 1. Then we check if the current R is already bigger than D (line 16) If so we subtract N from R, R ← R - N (line 17) and then we push a 1 to the right of Q (line 18), again by adding Q to itself plus the carry set by the comparison (if R >= N then there is no borrow so C became 1 in cmp of line 16).
The shown code is fine but it can be improved in a number of ways. First, there is no need to check all the bits of a number (although this gives as an upper bound of the cost in the worst of the cases). Second, we should try hard to reduce the number of used registers. Here we are using 5 registers, is there a way we can use less registers? For this we will have to use a slightly different approach.

Given N and D, we will first shift D as many bits to the left as possible but always having N > D. So, for instance if we divide N=1010(2 by D=10(2 we would adjust D until it was D0=1000(2 (this is shifting twice to the left). Now we start a similar process to the one above: if Ni ≥ Di, we set 1 to the lowest bit of Q and then we compute a new Ni+1 ← Ni - Di and a new Di+1 ← Di/2. If Ni < Di then we simply compute a new Di+1 ← Di/2. We stop when the current Di is smaller than the initial D (not D0). Note that this condition is what makes dividing N=1010(2 by D=10(2 different that dividing N=1010(2 by D=1(2 although the D0 of both cases is the same.

```
1  better_unsigned_division :
2      /* r0 contains N and Ni */
3      /* r1 contains D */
4      /* r2 contains Q */
5      /* r3 will contain Di */
6
7      mov r3, r1                    /* r3 ← r1 */
8      cmp r3, r0, LSR #1            /* update cpsr with r3 - r0/2 */
9      .Lloop2:
1        movls r3, r3, LSL #1        /* if r3 <= 2*r0 (C=0 or Z=1) then r3 ← r3*2 */
0        cmp r3, r0, LSR #1          /* update cpsr with r3 - (r0/2) */
1        bls .Lloop2                 /* branch to .Lloop2 if r3 <= 2*r0 (C=0 or Z=1) */
1
1      mov r2, #0                    /* r2 ← 0 */
2
1
```

```
    .Lloop3:
      cmp r0, r3                /* update cpsr with r0 - r3 */
      subhs r0, r0, r3          /* if r0 >= r3 (C=1) then r0 ← r0 - r3 */
      adc r2, r2, r2            /* r2 ← r2 + r2 + C.
                                   Note that if r0 >= r3 then C=1, C=0 otherwise */

      mov r3, r3, LSR #1        /* r3 ← r3/2 */
      cmp r3, r1                /* update cpsr with r3 - r1 */
      bhs .Lloop3               /* if r3 >= r1 branch to .Lloop3 */

    bx lr
```

We can avoid the first loop where we shift until we exceed by counting the leading zeroes. By counting the leading zeroes of the dividend and the divisor we can straightforwardly compute how many bits we need to shift the divisor.

```
clz_unsigned_division:
    clz  r3, r0               /* r3 ← CLZ(r0) Count leading zeroes of N */
    clz  r2, r1               /* r2 ← CLZ(r1) Count leading zeroes of D */
    sub  r3, r2, r3           /* r3 ← r2 - r3.
                                 This is the difference of zeroes
                                 between D and N.
                                 Note that N >= D implies CLZ(N) <= CLZ(D)*/
    add r3, r3, #1            /* Loop below needs an extra iteration count */
```

```
    mov r2, #0                  /* r2 ← 0 */
    b .Lloop_check4
    .Lloop4:
      cmp r0, r1, lsl r3        /* Compute r0 - (r1 << r3) and update cpsr */
      adc r2, r2, r2            /* r2 ← r2 + r2 + C.
                                   Note that if r0 >= (r1 << r3) then C=1, C=0 otherwise */
      subcs r0, r0, r1, lsl r3 /* r0 ← r0 - (r1 << r3) if C = 1 (this is, only if r0 >= (r1 << r3) ) */
    .Lloop_check4:
      subs r3, r3, #1           /* r3 ← r3 - 1 */
      bpl .Lloop4               /* if r3 >= 0 (N=0) then branch to .Lloop1 */

    mov r0, r2
    bx lr
```

Signed division

Signed division can be computed with an unsigned division but taking care of the signs. We can first compute |N|/|D| (this is, ignoring the signs of N and D), this will yield a quotient Q+ and remainder R+. If signs of N and D are different then Q = -Q+. If N < 0, then R = -R+, as we said at the beginning of the post.

Powers of two

An unsigned division by a power of two 2N is as simple as doing a logical shift right of N bits. Conversely, a signed division by a power of two 2N is as simple as doing an arithmetic shift right of N bits. We can use mov and the addressing modes LSR and ASR for this. This case is ideal because it is extremely fast.

Division by a constant integer

When we divide a number by a constant, we can use a multiplication by a magic number to compute the division. All the details and the theory of this technique is too long to write it here but you can find it in chapter 10 of Hacker's Delight. We can summarize it, though, into three values: a magic constant M, a shift S and an additional flag. The author set up a magic number calculator that computes these numbers.
It is not obvious how to properly use these magic numbers, so I crafted a small Python scriptwhich emits code for the signed and the unsigned case. Imagine you want to divide an unsigned number by 14. So let's ask our script.

```
$ ./magic.py 14 code_for_unsigned
u_divide_by_14:
   /* r0 contains the argument to be divided by 14 */
   ldr r1, .Lu_magic_number_14 /* r1 ← magic_number */
   umull r1, r2, r1, r0   /* r1 ← Lower32Bits(r1*r0). r2 ← Upper32Bits(r1*r0) */
   adds r2, r2, r0        /* r2 ← r2 + r0 updating cpsr */
   mov r2, r2, ROR #0     /* r2 ← (carry_flag << 31) | (r2 >> 1) */
   mov r0, r2, LSR #4     /* r0 ← r2 >> 4 */
   bx lr                  /* leave function */
   .align 4
   .Lu_magic_number_14: .word 0x24924925
```

Similarly we can ask for the signed version:

```
$ ./magic.py 14 code_for_signed
s_divide_by_14:
   /* r0 contains the argument to be divided by 14 */
   ldr r1, .Ls_magic_number_14 /* r1 ← magic_number */
   smull r1, r2, r1, r0   /* r1 ← Lower32Bits(r1*r0). r2 ← Upper32Bits(r1*r0) */
   add r2, r2, r0         /* r2 ← r2 + r0 */
   mov r2, r2, ASR #3     /* r2 ← r2 >> 3 */
```

```
   mov r1, r0, LSR #31    /* r1 ← r0 >> 31 */
   add r0, r2, r1         /* r0 ← r2 + r1 */
   bx lr                  /* leave function */
   .align 4
   .Ls_magic_number_14: .word 0x92492493
```

As an example I have used it to implement a small program that just divides the user input by 14.

```
/* -- divideby14.s */

.data

.align 4
read_number: .word 0

.align 4
message1 : .asciz "Enter an integer to divide it by 14: "

.align 4
message2 : .asciz "Number %d (signed-)divided by 14 is %d\n"

.align 4
scan_format : .asciz "%d"

.text

/* This function has been generated using "magic.py 14 code_for_signed" */
s_divide_by_14:
   /* r0 contains the argument to be divided by 14 */
   ldr r1, .Ls_magic_number_14 /* r1 ← magic_number */
   smull r1, r2, r1, r0   /* r1 ← Lower32Bits(r1*r0). r2 ← Upper32Bits(r1*r0) */
   add r2, r2, r0         /* r2 ← r2 + r0 */
   mov r2, r2, ASR #3     /* r2 ← r2 >> 3 */
   mov r1, r0, LSR #31    /* r1 ← r0 >> 31 */
   add r0, r2, r1         /* r0 ← r2 + r1 */
   bx lr                  /* leave function */
   .align 4
   .Ls_magic_number_14: .word 0x92492493
```

```
.globl main

main:
    /* Call printf */
    push {r4, lr}
    ldr r0, addr_of_message1       /* r0 ← &message */
    bl printf

    /* Call scanf */
    ldr r0, addr_of_scan_format   /* r0 ← &scan_format */
    ldr r1, addr_of_read_number   /* r1 ← &read_number */
    bl scanf

    ldr r0, addr_of_read_number   /* r1 ← &read_number */
    ldr r0, [r0]                  /* r1 ← *r1 */

    bl s_divide_by_14
    mov r2, r0

    ldr r1, addr_of_read_number   /* r1 ← &read_number */
    ldr r1, [r1]                  /* r1 ← *r1 */

    ldr r0, addr_of_message2      /* r0 ← &message2 */
    bl printf                     /* Call printf, r1 and r2 already
                                     contain the desired values */
    pop {r4, lr}
    mov r0, #0
    bx lr

addr_of_message1: .word message1
addr_of_scan_format: .word scan_format
addr_of_message2: .word message2
addr_of_read_number: .word read_number
```

Using VFPv2

I would not recommend using this technique. I present it here for the sake of completeness. We simply convert our integers to floating point numbers, divide them as floating point numbers and convert the result back to an integer.

```
1
2  vfpv2_division:
3      /* r0 contains N */
4      /* r1 contains D */
5      vmov s0, r0              /* s0 ← r0 (bit copy) */
6      vmov s1, r1              /* s1 ← r1 (bit copy) */
7      vcvt.f32.s32 s0, s0      /* s0 ← (float)s0 */
8      vcvt.f32.s32 s1, s1      /* s1 ← (float)s1 */
9      vdiv.f32 s0, s0, s1      /* s0 ← s0 / s1 */
1      vcvt.s32.f32 s0, s0      /* s0 ← (int)s0 */
0      vmov r0, s0              /* r0 ← s0 (bit copy). Now r0 is Q */
1      bx lr
1
```

Comparing versions

After a comment below, I thought it would be interesting to benchmark the general division algorithm. The benchmark I used is the following:

```
.set MAX, 16384
main:
    push {r4, r5, r6, lr}

    mov r4, #1                          /* r4 ← 1 */

    b .Lcheck_loop_i                    /* branch to .Lcheck_loop_i */
    .Lloop_i:
       mov r5, r4                       /* r5 ← r4 */
       b .Lcheck_loop_j                 /* branch to .Lcheck_loop_j */
       .Lloop_j:
```

```
        mov r0, r5                    /* r0 ← r5. This is N */
        mov r1, r4                    /* r1 ← r4. This is D */

        bl  <your unsigned division routine here>

        add r5, r5, #1
      .Lcheck_loop_j:
        cmp r5, #MAX                   /* compare r5 and MAX */
        bne .Lloop_j                  /* if r5 != 10 branch to .Lloop_j */
      add r4, r4, #1
    .Lcheck_loop_i:
      cmp r4, #MAX                    /* compare r4 and MAX */
      bne .Lloop_i                    /* if r4 != 10 branch to .Lloop_i */

    mov r0, #0

    pop {r4, r5, r6, lr}
    bx lr
```

Basically it does something like this

```
for (i = 1; i < MAX; i++)
 for (j = i; j < MAX; j++)
   division_function(j, i);
```

Timings have been obtained using perf_3.2 stat --repeat=5 -e cpu-clock. In the table below, __aeabi_uidiv is the function in libgcc that gcc uses to implement an unsigned integer division.

Version	Time (seconds)
unsigned_longdiv	45,43
vfpv2_division	9,70
clz_unsigned_longdiv	8,48
__aeabi_uidiv	7,37
better_unsigned_long	6,67

div	

As you can see the performance of our unsigned long division is dismal. The reason it is that it always checks all the bits. The libgcc version is like our clz version but the loop has been fully unrolled and there is a computed branch, similar to a Duff's device. Unfortunately, I do not have a convincing explanation why better_unsigned_longdiv runs faster than the other versions, because the code, a priori, looks worse to me.

That's all for today.

ARM assembler in Raspberry Pi – Chapter 16

August 23, 2013 Roger Ferrer Ibáñez, 4

We saw in chapters 6 and 12 several control structures but we left out a usual one: the switch also known as select/case. In this chapter we will see how we can implement it in ARM assembler.

Switch control structure

A switch in C has the following structure.

```
switch (E)
{
   case V1: S1;
   case V2: S2;
   default: Sdefault;
}
```

In the example above, expression E is evaluated and its value is used to determine the next statement executed. So if E evaluates to V2, S2 will be the next statement executed. If no casematches, the whole switch construct is ignored unless there is a default case the statement of which is executed instead.

Note that, once the flow jumps to a statement, the execution continues from that point unless a break statement is found. The break statement switch construct. Most of the time the programmer adds a break to end each case. Otherwise fall-through cases happens. In the example above, if E evaluates to V1 and there is no break in S1, the program would continue running S2 and Sdefault unless the program encounters a break statement inside S2 or Sdefault. Fall-through may look a bit weird and confusing but there are some cases where is useful.

That said, C is a particularly bad example to showcase this structure. The reason is that the exact language definition of a switch in C is as follows.

```
switch (E)
   S;
```

S can be anything but the flow will always jump to a case or a default inside S, so if S does not contain any of these statements, nothing happens.

```
switch (E)
```

```
  printf("This will never be printed\n");
```

So for a switch to be useful we will need at least one case or default statement. If more than one is needed, then we can use a compound statement (a list of statements enclosed in side {and } as shown in the first example above.
Note also, that case and default statements are only valid inside the S of a switch but this does not mean that they have to be immediately nested inside them.

```
switch (E)
{
  if (X) // Note that the check of the truth value of X will be never run!
  {
     default: printf ("Hi!\n");
  }
  else
  {
     case 10: printf ("Howdy stranger!\n");
  }
}
```

As you can see, the switch statement in C is pretty liberal. Other languages, like Pascal or Fortran, have stricter syntaxes that do not allow fall-through nor loose positioning of case/default.

```
{ Case statement in Pascal }
Case Number of
 1 : WriteLn ('One');
 2 : WriteLn ('Two');
Else
 WriteLn ('Other than one or two');
End;
```

In this post, we will not care about these strange cases of switch although we will allow fall-through.

Implementing switch

Probably you already have figured that a switch not involving fall-through in any of its cases is equivalent to a sequence of if-else blocks. The following switch,

```
switch (x)
{
```

```
  case 5: code_for_case5; break;
  case 10: code_for_case10; break;
  default: code_for_default; break;
  // break would not be required here as this is the last case
}
```

can be implemented as

```
if (x == 5)
  code_for_case5;
else if (x == 10)
  code_for_case10;
else /* default */
  code_for_default;

code_after;
```

In contrast to the usual if-else statement, there need not be a branch that goes after the if-statement once the if branch has been executed. This is, in the example above, it is optional to have a branch after code_for_case5 that goes to code_after. If such branch is omitted, then a fall-through to code_for_case10 happens naturally. So the break statement inside a switch is simply that unconditional branch.

```
  /* Here we evaluate x and keep it in r0 */
  case_5:             /* case 5 */
    cmp r0, #5        /* Compute r0 - 5 and update cpsr */
    bne case_10       /* if r0 != 5 branch to case_10 */
    code_for_case5
    b after_switch    /* break */

  case_10:            /* case 10 */
    cmp r0, #10       /* Compute r0 - 10 and update cpsr */
    bne case_default  /* If r0 != 10 branch to case_default */
    code_for_case10
    b after_switch    /* break */

  case_default:
    code_for_default
    /* Note that if default is not the last case
```

```
       we need a branch to after_switch here */

  after_switch:
```

We can put all the checks at the beginning, as long as we preserve the order of the cases (so fall-through works if break is omitted).

```
  /* Here we evaluate x and keep it in r0 */
  cmp r0, #5         /* Compute r0 - 5 and update cpsr */
  beq case_5         /* if r0 == 5 branch to case_5 */
  cmp r0, #10        /* Compute r0 - 10 and update cpsr */
  beq case_10        /* if r0 == 10 branch to case_10 */
  b case_default     /* branch to default case
                        Note that there is no default case
                        we would branch to after_switch */

  case_5:            /* case 5 */
    code_for_case5
    b after_switch   /* break */

  case_10:           /* case 10 */
    code_for_case10
    b after_switch   /* break */

  case_default:
    code_for_default
    /* Note that if default is not the last case
       we need a branch to after_switch here */

  after_switch:
```

This approach is sensible if the number of cases is low. Here “low” is not very well defined, let’s say 10 or less. What if we have lots of cases? A sequence of if-else checks will make as many comparisons as cases. If the values of the N cases are uniformly spread during the execution of the program, this means that in average we will have to do N/2 checks. If the values are not uniformly spread, then it is obvious that we should check the common values first and the rare last (sadly, most of the times we have no idea of their frequency).

There are a number of ways of reducing the cost of checking the cases: tables and binary search.

Jump tables

Imagine we have a switch like this one

```
switch (x)
{
  case 1: do_something_1;
  case 2: do_something_2;
  ...
  case 100: do_something_100;
}
```

If we implement it the way shown above we will make in average (for an uniformly distributed set of values of x) 50 comparisons. We can improve this if we simply use the value c to index a table of addresses to the instructions of the case instructions.

Consider the program below where we use the value of argc of a C program. In C, the mainfunction receives two parameters, argc and argv: argc is just an integer, in the register r0 as usual; argv is an address, in the register r1 as usual, to an array of the arguments passed in the command-line. There are as many elements in argv as the value of argc, at least one. We will not use argv today, only argc.

```
int main(int argc, char *argv[])
{
   int x;
   switch (argc)
   {
     case 1: x = 1; break;
     case 2: x = 2; break;
     case 3: x = 3; break;
     default: x = 42; break;
   }
   return x;
}
```

We are using just 3 cases plus the default one, but it would not be complex (yet cumbersome) to extend it to 100 cases.

```
1 /* jumptable.s */
2 .data
```

```
.text

.globl main

main:
  cmp r0, #1                  /* r0 - 1 and update cpsr */
  blt case_default            /* branch to case_default if r0 < 1 */
  cmp r0, #3                  /* r0 - 3 and update cpsr */
  bgt case_default            /* branch to case_default if r0 > 3 */

  sub r0, r0, #1              /* r0 ← r0 - 1. Required to index the table */
  ldr r1, addr_of_jump_table  /* r1 ← &jump_table */
  ldr r1, [r1, +r0, LSL #2]   /* r1 ← *(r1 + r0*4).
                                 This is r1 ← jump_table[r0] */

  mov pc, r1                  /* pc ← r1
                                 This will cause a branch to the
                                 computed address */

  case_1:
   mov r0, #1                 /* r0 ← 1 */
   b after_switch             /* break */

  case_2:
   mov r0, #2                 /* r0 ← 2 */
   b after_switch             /* break */

  case_3:
   mov r0, #3                 /* r0 ← 3 */
   b after_switch             /* break */

  case_default:
   mov r0, #42                /* r0 ← 42 */
   b after_switch             /* break (unnecessary) */

  after_switch:
```

```
5
2
6
2
7
2
8
2
9
3
0
3
1
3
2
3
3
3
```

```
  bx lr                         /* Return from main */

.align 4
jump_table:
   .word case_1
   .word case_2
   .word case_3

.align 4
addr_of_jump_table: .word jump_table
```

As you can see in line 43 we define a jump table, whose elements are the addresses of the labels of each case (in order). In lines 14 to 16 we load the appropiate value from that table after we are sure that the value of argc is between 1 and 3, checked in lines 9 to 12. Finally, we load the address to pc. This will effectively do a branch to the proper case.
If you run the program you will see diferent exit codes returned (remember that they are retuned through r0 in main). The program only counts the arguments, if instead of “a b” you use “one two” it will return 3 as well. More than two arguments and it will return 42.

```
$ ./jumptable ; echo $?
1
$ ./jumptable a ; echo $?
2
$ ./jumptable a b  ; echo $?
3
$ ./jumptable a b c ; echo $?
42
```

To safely use the jump table we have to make sure the value of the case lies in the bounds of the table. If m is the minimum case value and M is the maximum case value, our table will have M – m + 1 entries. In the example above m

= 1 and M = 3 so we have 3 entries in the table. We have to make sure that the value used to index is $m \leq x \leq M$, otherwise we would be accessing incorrect memory locations. Also remember that to properly index the jump table we will have to subtract m to the case value.

Jump tables are great, once we have checked that the case value is in the proper range (these are two comparisons) then we do not have to compare anything else. So basically the cost of comparisons in this approach is constant (i.e. it does not grow if the number of cases grow).

There are two big downsides to this approach which prevent us from always using it. The first one happens when the difference between M and m is large, our jump table will be large. This enlarges the code size. We have, basically, traded time by space. Now our code size will add 4 bytes per case handled in a jump table. A table of 256 entries will take up to 1 Kbyte (1024 bytes) of memory in our executable program. To be fair, this is the amount of space taken by 256 instructions. So if code size is a concern for you (and usually in the embedded world is), this approach may not be suitable. The second big downside happens when there are “holes” in the cases. Imagine our cases are just 1, 3 and 100. The table will be 100 items long but only 1, 3 and 100 will have useful entries: all the remaining entries will have the address of the default case (or the address after the switch if the default case is omitted). In this case we are not just taking 400 bytes, we are wasting 388 bytes (97% of entries would be useless!). So if the number of cases is low and the values are scattered in a large range, jump tables are not a good choice.

Compute the case address

This strategy is a bit complicated and has more constraints than a jump table, so it is less general. If all the cases are ordered and they take the same amount of instructions, we can compute the address of the case without using a jump table. This is risky because we have to be careful when computing the address of the branch using the current value (otherwise we will jump to a wrong address and bad things will happen for sure).

If not all the cases take the same amount of instructions, we can compensate them to take as many instructions as the case with the biggest number of instructions. We can do that using the nop instruction that does nothing but occupy space. If the variance of the number of instructions among cases is small we will just end adding some nops to a few cases. If the variance is large, we may end with code bloat, something we wanted to avoid when using this technique.

If there are holes, we can just branch them to the default case and fill the remaining instructions with nops. Again if the number of holes is large this is prone to code bloat as well.
In our example of the jump table, each case takes just two instructions. So we can get the address of the first case and use it as a base address to compute the branch.

```
/* calcjump.s */
.data

.text

.globl main

main:
  cmp r0, #1                  /* r0 - 1 and update cpsr */
  blt case_default            /* branch to case_default if r0 < 1 */
  cmp r0, #3                  /* r0 - 3 and update cpsr */
  bgt case_default            /* branch to case_default if r0 > 3 */

  sub r0, r0, #1              /* r0 ← r0 - 1. Required to index the table */
  ldr r1, addr_of_case_1      /* r1 ← &case_1 */
  add r1, r1, r0, LSL #3      /* r1 ← r1 + r0 * 8
                               Each instruction is 4 bytes
                               Each case takes 2 instructions
                               Thus, each case is 8 bytes (4 * 2)
                               */

  mov pc, r1                  /* pc ← r1
                               This will cause a branch to the
                               computed address */

  case_1:
   mov r0, #1                 /* r0 ← 1 */
   b after_switch             /* break */

  case_2:
   mov r0, #2                 /* r0 ← 2 */
```

```
    b after_switch              /* break */

  case_3:
    mov r0, #3                  /* r0 ← 3 */
    b after_switch              /* break */

  case_default:
    mov r0, #42                 /* r0 ← 42 */
    b after_switch              /* break (unnecessary) */

  after_switch:

  bx lr                         /* Return from main */

.align 4
addr_of_case_1: .word case_1
```

Binary search

Consider again our example with 100 cases. A string of if-else will require in average 50 comparisons. Can we reduce the number of comparisons? Well, the answer is yes. Perform a binary search of the case.

A binary search will discard half of the case set each time. This will allow us to dramatically reduce the amount of comparisons. The following example implements the same code in the jump table but with cases 1 to 10.

```
/* binsearch.s */
.data

.text

.globl main

main:
```

```
9
1   cmp r0, #1              /* r0 - 1 and update cpsr */
0   blt case_default        /* if r0 < 1 then branch to case_default */
1   cmp r0, #10             /* r0 - 10 and update cpsr */
1   bgt case_default        /* if r0 > 10 then branch to case default */
1
2   case_1_to_10:
1     cmp r0, #5            /* r0 - 5 and update cpsr */
3     beq case_5            /* if r0 == 5 branch to case_5 */
1     blt case_1_to_4       /* if r0 < 5 branch to case_1_to_4 */
4     bgt case_6_to_10      /* if r0 > 5 branch to case_6_to_4 */
1
5   case_1_to_4:
1     cmp r0, #2            /* r0 - 2 and update cpsr */
6     beq case_2            /* if r0 == 2 branch to case_2 */
1     blt case_1            /* if r0 < 2 branch to case_1
7                              (case_1_to_1 does not make sense) */
1     bgt case_3_to_4       /* if r0 > 2 branch to case_3_to_4 */
8
1   case_3_to_4:
9     cmp r0, #3            /* r0 - 3 and update cpsr */
2     beq case_3            /* if r0 == 3 branch to case_3 */
0     b case_4              /* otherwise it must be r0 == 4,
2                              branch to case_4 */
1
2   case_6_to_10:
2     cmp r0, #8            /* r0 - 8 and update cpsr */
2     beq case_8            /* if r0 == 8 branch to case_8 */
3     blt case_6_to_7       /* if r0 < 8 then branch to case_6_to_7 */
2     bgt case_9_to_10      /* if r0 > 8 then branch to case_9_to_10 */
4
2
5   case_6_to_7:
2     cmp r0, #6            /* r0 - 6 and update cpsr */
6     beq case_6            /* if r0 == 6 branch to case_6 */
2     b case_7              /* otherwise it must be r0 == 7,
7                              branch to case 7 */
2
```

```
case_9_to_10:
  cmp r0, #9            /* r0 - 9 and update cpsr */
  beq case_9            /* if r0 == 9 branch to case_9 */
  b case_10             /* otherwise it must be r0 == 10,
                           branch to case 10 */

case_1:
   mov r0, #1
   b after_switch
case_2:
   mov r0, #2
   b after_switch
.
.  /* Cases from 3 to 9 omitted */
.
case_10:
   mov r0, #10
   b after_switch

case_default:
 mov r0, #42               /* r0 ← 42 */
 b after_switch            /* break (unnecessary) */

after_switch:

bx lr                      /* Return from main */
```

This strategy is able to determine the case value in just only 3 comparisons (if we ignore the mandated two comparisons for range checking). What we do is we check the compare the case value with the middle one in the current range. This way we can discard half of the sets at every comparison step.

This strategy works well also for scattered case sets like [1, 2, 3, 24, 25, 26, 97, 98, 99, 300]. In this case the comparisons would be

```
case_1_to_300:
  cmp r0, #25
  beq case_25
  blt case_1_to_24
  bgt case_26_to_300
case_1_to_24:
  cmp r0, #2
  beq case_2
  blt case_1
  bgt case_3_to_24
case_3_to_24:
  cmp r0, #3
  beq case_3
  b case_24
case_26_to_300:
  cmp r0, #98
  beq case_98
  blt case_26_to_97
  bgt case_99_to_300
case_26_to_97:
  cmp r0, #26
  beq case_26
  b case_97
case_99_to_300:
  cmp r0, #99
  beq case_99
  b case_300
```

which is 3 comparisons at most also.

Using this strategy the number of comparisons is log(N), where N is the number of elements in the case set. So for 10 elements, in the worst of the cases, we will have to do 3 comparisons, for 20 at most 4, for 40 at most 5, etc.

Let’s retake the code bloat issue that arised with jump tables. If you check, every comparison requires 3 or 4 instructions, this is about 12 to 16 bytes per comparison. If we have a case set of 256 elements, the generated code will require 128 comparisons blocks in total. While the number of comparisons performed at runtime, 8 in the worst of the cases, we still need 128 case_x_to_y comparison blocks to perform the binary search. If we pessimistically assume that all comparison blocks take 4 instructions this will be 4*128*4 = 2048 bytes in instructions. Compare that to a jump table of 256 positions, each position takes 4 bytes: 256 * 4 = 1024 bytes. So, binary search is not so competitive in terms of code size.
Binary search, thus, is useful for large scattered sets. Recall that if-else strings are not efficient for large sets of cases and jump tables waste space if the case range lacks lots of cases.

Hybrid approach

Is it possible to combine the two strategies? The answer is yes. We will use two tables: a case values table (sorted, usually in ascending order) and addresses for each case in another table, in the same order as the case values table.

We will make a binary search inside the case value set. When the value is found we will use the index of the match to calculate a jump. For the example below we will use the case set [1, 2, 3, 24, 25, 26, 97, 98, 99, 300].

```
1   /* hybrid.s */
2   .data
3
4   .text
5
6   .globl main
7
8   main:
9     push {r4, r5, r6, lr}
1
0     cmp r0, #1                /* r0 - 1 and update cpsr */
1     blt case_default          /* if r0 < 1 then branch to case_default */
1     cmp r0, #300              /* r0 - 300 and update cpsr */
1     bgt case_default          /* if r0 > 300 then branch to case default */
```

```
2
1   /* prepare the binary search.
3      r1 will hold the lower index
1      r2 will hold the upper index
4      r3 the base address of the case_value_table
1   */
5   mov r1, #0
1   mov r2, #9
6   ldr r3, addr_case_value_table /* r3 ← &case_value_table */
1
7   b check_binary_search
1   binary_search:
8     add r4, r1, r2          /* r4 ← r1 + r2 */
1     mov r4, r4, ASR #1      /* r4 ← r4 / 2 */
9     ldr r5, [r3, +r4, LSL #2]   /* r5 ← *(r3 + r4 * 4).
2                                    This is r5 ← case_value_table[r4] */
0     cmp r0, r5              /* r0 - r5 and update cpsr */
2     sublt r2, r4, #1        /* if r0 < r5 then r2 ← r4 - 1 */
1     addgt r1, r4, #1        /* if r0 > r5 then r1 ← r4 + 1 */
2     bne check_binary_search /* if r0 != r5 branch to binary_search */
2
2     /* if we reach here it means that r0 == r5 */
3     ldr r5, addr_case_addresses_table /* r5 ← &addr_case_value_table */
2     ldr r5, [r5, +r4, LSL #2]   /* r5 ← *(r5 + r4*4)
4                                    This is r5 ← case_addresses_table[r4] */
2     mov pc, r5              /* branch to the proper case */
5
2   check_binary_search:
6     cmp r1, r2              /* r1 - r2 and update cpsr */
2     ble binary_search       /* if r1 <= r2 branch to binary_search */
7
2   /* if we reach here it means the case value
8      was not found. branch to default case */
2   b case_default
9
3
0     case_1:
3      mov r0, #1
```

```
    b after_switch
case_2:
    mov r0, #2
    b after_switch
case_3:
    mov r0, #3
    b after_switch
case_24:
    mov r0, #24
    b after_switch
case_25:
    mov r0, #95
    b after_switch
case_26:
    mov r0, #96
    b after_switch
case_97:
    mov r0, #97
    b after_switch
case_98:
    mov r0, #98
    b after_switch
case_99:
    mov r0, #99
    b after_switch
case_300:
    mov r0, #300    /* The error code will be 44 */
    b after_switch

case_default:
 mov r0, #42        /* r0 ← 42 */
 b after_switch     /* break (unnecessary) */

after_switch:

pop {r4,r5,r6,lr}
bx lr               /* Return from main */
```

```
0
5
1
5
2
5
3
5
4
5
5
6
5
7
5
8
5
9
6
0
6
1
6
2
0
4
```

```
case_value_table: .word 1, 2, 3, 24, 25, 26, 97, 98, 99, 300
addr_case_value_table: .word case_value_table

case_addresses_table:
    .word case_1
    .word case_2
    .word case_3
    .word case_24
    .word case_25
    .word case_26
    .word case_97
    .word case_98
    .word case_99
    .word case_300
addr_case_addresses_table: .word case_addresses_table
```

In lines 21 to 44 we implement the binary search. This implementation is an iterative binary search where r1 and r2 keep the lower and upper indexes of the table that is currently searched. We will leave the search if the lower index becomes larger than the upper, lines 42 to 44. When searching the range given by r1 and r2, we will compute r4 as the middle index (r1+r2)/2, lines 27 to 28. We will compare it to the current case value being searched, in r0, line 31. If the value, r5, in the case value table (which must be in ascending order) is lower than the current case value being searched, then we shrink the range from r1 to r4-1, so we update r2 only if r0 < r5, line 32. Conversely if r0 > r5 then we shrink the range from r4+1 to r2, line 33. If the value of r5 matches, then we use the

index r4 to load the case address and branch to it, lines 37 to 40. Note that if r0 is different to r5, we have to omit this step so we branch to the check of the loop, line 34.
You can check that this works.

```
$ ./hybrid ; echo $?
1
$ ./hybrid 2 ; echo $?
2
$ ./hybrid 2 3 ; echo $?
3
$ ./hybrid 2 3 4 ; echo $?
42
$ ./hybrid 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 ; echo $?
24
```

This approach has several interesting properties. It reduces the number of comparisons (we will make the same number of comparisons as in the binary search) and avoids the code bloat due to big jump tables (avoiding useless entries) and comparison blocks (by using a loop). As a drawback, this approach, requires two tables.

That's all for today.

Create a temporary zip file to send as response in Rails

November 15, 2013 brafales, 10

We have been doing a painful migration from Rails 2 to Rails 3 for several months at work, and refactoring some code the other day I had to do something in a non straightforward way, so I thought I'd share that.

Basically we had an action that would group several files into a zip file and return those zipped files to the user as a response. In the old code, a randomly named file was created on the /tmpfolder of the hosting machine, being used as the zip file for the rubyzip gem, and then returned in the controller response as an attachment.
During the migration, we've replaced all those bespoken temp file generation for proper Tempfileobjects. This was just another one of those replacements to do. But it turned out not to be that simple.
My initial thought was that something like this would do the trick:

```
filename = 'attachment.zip'
temp_file = Tempfile.new(filename)

Zip::File.open(temp_file.path, Zip::File::CREATE) do |zip_file|
    #put files in here
end
zip_data = File.read(temp_file.path)
send_data(zip_data, :type => 'application/zip', :filename => filename)
```

But it did not. The reason for that is that the open method, when used with the Zip::File::CREATE flag, expects the file either not to exist or to be already a zip file (that is, have the correct zip structure data on it). None of those 2 cases is ours, so the method didn't work.
So as a solution, you have to open the temporary file using the Zip::OutputStream class and initialize it so it's converted to an empty zip file, and after that you can open it the usual way. Here's a full simple example on how to achieve this:

```
#Attachment name
filename = 'basket_images-'+params[:delivery_date].gsub(/[^0-9]/,'')+'.zip'
temp_file = Tempfile.new(filename)
```

```ruby
begin
  #This is the tricky part
  #Initialize the temp file as a zip file
  Zip::OutputStream.open(temp_file) { |zos| }

  #Add files to the zip file as usual
  Zip::File.open(temp_file.path, Zip::File::CREATE) do |zip|
    #Put files in here
  end

  #Read the binary data from the file
  zip_data = File.read(temp_file.path)

  #Send the data to the browser as an attachment
  #We do not send the file directly because it will
  #get deleted before rails actually starts sending it
  send_data(zip_data, :type => 'application/zip', :filename => filename)
ensure
  #Close and delete the temp file
  temp_file.close
  temp_file.unlink
end
```

ARM assembler in Raspberry Pi – Chapter 17

November 20, 2013 Roger Ferrer Ibáñez, 13

In chapter 10 we saw the basics to call a function. In this chapter we will cover more topics related to functions.

Passing data to functions

We already know how to call a function and pass them parameters. We also know how to return data from a function. Nevertheless, there are some issues which we have not fully solved yet.

- Passing large amounts of data
- Returning more than one piece of data

There are several ways to tackle this problem, but most of them involve pointers. Pointers are dreaded by many people, who do not fully understand them, but they are a crucial part in the way computers work. That said, most of the troubles with pointers are actually related to dynamic memory rather than the pointers themselves. We will not consider dynamic memory here.

So what is a pointer?

A pointer is some location in the memory the contents of which are simply an address of the memory.

This definition may be confusing, but we have already been using pointers in previous chapters. It is just that we did not name them this way. We usually talked about addresses and/or labels in the assembler. Consider this very simple program:

```
/* first_pointer.s */

.data

.align 4
number_1  : .word 3
```

```
.text
.globl main

main:
    ldr r0, pointer_to_number    /* r0 ← &number */
    ldr r0, [r0]                 /* r0 ← *r0. So r0 ← number_1 */

    bx lr

pointer_to_number: .word number_1
```

As you can see, I deliberatedly used the name pointer_to_number to express the fact that this location in memory is actually a pointer. It is a pointer to number_1 because it holds its address.
Imagine we add another number, let's call it number_2 and want pointer_to_number to be able to point to number_2, this is, contain the address of number_2 as well. Let's make a first attempt.

```
.data

.align 4
number_1  : .word 3
number_2  : .word 4

.text
.globl main

main:
    ldr r1, address_of_number_2  /* r1 ← &number_2 */
    str r1, pointer_to_number    /* pointer_to_number ← r1, this is pointer_to_number ← &number_2 */

    bx lr

pointer_to_number: .word number_1
address_of_number_2: .word number_2
```

But if you run this you will get a rude Segmentation fault. We cannot actually modify pointer_to_number because, even if it is a location of memory that contains an address (and it would contain another address after the store) it is not in the data section, but in the textsection. So this is a statically defined pointer, whose value (i.e. the address it contains)

cannot change. So, how can we have a pointer that can change? Well, we will have to put it in the datasection, where we usually put all the data of our program.

```
.data

.align 4
number_1  : .word 3
number_2  : .word 4
pointer_to_number: .word 0

.text
.globl main


main:
    ldr r0, addr_of_pointer_to_number
                            /* r0 ← &pointer_to_number */

    ldr r1, addr_of_number_2 /* r1 ← &number_2 */

    str r1, [r0]              /* *r0 ← r1.
                               This is actually
                                 pointer_to_number ← &number_2 */

    ldr r1, [r0]              /* r1 ← *r0.
                               This is actually
                                 r1 ← pointer_to_number
                               Since pointer_to_number has the value &number_2
                               then this is like
                                 r1 ← &number_2
                            */


    ldr r0, [r1]              /* r0 ← *r1
                               Since r1 had as value &number_2
                               then this is like
                                  r0 ← number_2
                            */
```

```
    bx lr

addr_of_number_1: .word number_1
addr_of_number_2: .word number_2
addr_of_pointer_to_number: .word pointer_to_number
```

From this last example several things should be clear. We have static pointers to number_1, number_2 and pointer_to_number (respectively called addr_of_number_1, addr_of_number_2 and addr_of_pointer_to_number). Note that addr_of_pointer_to_number is actually a pointer to a pointer! Why these pointers are statically defined? Well, we can name locations of memory (i.e. addresses) using labels (this way we do not have to really know the exact address and at the same time we can use a descriptive name). These locations of memory, named through labels, will never change during the execution of the program so they are somehow predefined before the program starts. This why the addresses of number_1, number_2 and addr_of_pointer_to_number are statically defined and stored in a part of the program that cannot change (the .text section cannot be modified when the program runs).

This means that accessing to pointer_to_number using addr_of_pointer_to_numberinvolves using a pointer to a pointer. Nothing fancy here, a pointer to a pointer is just a location of memory that contains the address of another location of memory that we know is a pointer too.

The program simply loads the value 4, stored in number_2 using pointer_to_number. We first load the address of the pointer (this is, the pointer to the pointer, but the address of the pointer may be clearer) into r0 in line 13. Then we do the same with the address of number_2, storing it in r1, line 16. Then in line 18 we update the value pointer_to_number (remember, the value of a pointer will always be an address) with the address of number_2. In line 22 we actually get the value of pointer_to_number loading it into r1. I insist again: the value of pointer_to_number is an address, so now r1 contains an address. This is the reason why in line 31 we load into r0 the value of the in r1.

Passing large amounts of data

When we pass data to functions we follow the conventions defined in the AAPCS. We try to fill the first 4 registers r0 to r3. If more data is expected we must use the stack. This means that if we were to pass a big chunk of data to a function we may end spending a lot of time just preparing the call (setting registers r0 to r3 and then pushing all the data on top of the stack, and remember, in reverse order!) than running the code of the function itself.

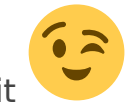
There are several cases when this situation arises. In a language like C, all parameters are passed by value. This means that the function receives a copy of the value. This way the function may freely modify this value and the caller will not see any changes in it. This may seem inefficient but from a productivity point of view, a function that does not cause any side effect to its inputs may be regarded as easier to understand than one that does.

```c
struct A
{
  // big structure
};

// This function computes a 'thing_t' using a 'struct A'
thing_t compute_something(struct A);

void my_code(void)
{
  struct A a;
  thing_t something;

  a = ...;
  something = compute_something(a)
  // a is unchanged here!
}
```

Note that in C, array types are not passed by value but this is by design: there are no array values in C although there are array types (you may need to repeat to yourself this last sentence several times before fully understanding it 

If our function is going to modify the parameter and we do not want to see the changes after the call, there is little that we can do. We have to invest some time in the parameter passing.

But what if our function does not actually modify the data? Or, what if we are interested in the changes the function did? Or even better, what if the parameter being modified is actually another output of the function?

Well, all these scenarios involve pointers.

Passing a big array by value

Consider an array of 32-bit integers and we want to sum all the elements. Our array will be in memory, it is just a contiguous sequence of 32-bit integers. We want to pass, somehow, the array

to the function (together with the length of the array if the length may not be constant), sum all the integers and return the sum. Note that in this case the function does not modify the array it just reads it.

Let's make a function sum_array_value that must have the array of integers passed by value. The first parameter, in r0 will be the number of items of the integer array. Registers r1 to r3 may (or may not) have value depending on the number of items in the array. So the first three elements must be handled differently. Then, if there are still items left, they must be loaded from the stack.

```
sum_array_value :
    push {r4, r5, r6, lr}

    /* We have passed all the data by value */

    /* r4 will hold the sum so far */
    mov r4, #0      /* r4 ← 0 */
    /* In r0 we have the number of items of the array */

    cmp r0, #1            /* r0 - #1 and update cpsr */
    blt .Lend_of_sum_array  /* if r0 < 1 branch to end_of_sum_array */
    add r4, r4, r1        /* add the first item */

    cmp r0, #2            /* r0 - #2 and update cpsr */
    blt .Lend_of_sum_array  /* if r0 < 2 branch to end_of_sum_array */
    add r4, r4, r2        /* add the second item */

    cmp r0, #3            /* r0 - #3 and update cpsr */
    blt .Lend_of_sum_array  /* if r0 < 3 branch to end_of_sum_array */
    add r4, r4, r3        /* add the third item */

    /*
     The stack at this point looks like this
       |                | (lower addresses)
       |                |
       | lr             |  <- sp points here
       | r6             |  <- this is sp + 4
       | r5             |  <- this is sp + 8
       | r4             |  <- this is sp + 12
```

```
      | big_array[3]   |  <- this is sp + 16 (we want r5 to point here)
      | big_array[4]   |
      |     ...        |
      | big_array[255] |
      |                |
      |                | (higher addresses)

   keep in r5 the address where the stack-passed portion of the array starts */
   add r5, sp, #16 /* r5 ← sp + 16 */

   /* in register r3 we will count how many items we have read
      from the stack. */
   mov r3, #0

   /* in the stack there will always be 3 less items because
      the first 3 were already passed in registers
      (recall that r0 had how many items were in the array) */
   sub r0, r0, #3

   b .Lcheck_loop_sum_array
   .Lloop_sum_array:
     ldr r6, [r5, r3, LSL #2]       /* r6 ← *(r5 + r3 * 4) load
                                       the array item r3 from the stack */
     add r4, r4, r6                 /* r4 ← r4 + r6
                                       accumulate in r4 */
     add r3, r3, #1                 /* r3 ← r3 + 1
                                       move to the next item */
   .Lcheck_loop_sum_array:
     cmp r3, r0           /* r0 - r3 and update cpsr */
     blt .Lloop_sum_array   /* if r3 < r0  branch to loop_sum_array */

 .Lend_of_sum_array:
   mov r0, r4  /* r0 ← r4, to return the value of the sum */
   pop {r4, r5, r6, lr}

   bx lr
```

```
65
```

The function is not particularly complex except for the special handling of the first 3 items (stored in r1 to r3) and that we have to be careful when locating inside the stack the array. Upon entering the function the items of the array passed through the stack are laid out consecutively starting from sp. The push instruction at the beginning pushes onto the stack four registers (r4, r5, r6 and lr) so our array is now in sp + 16 (see lines 30 and 38). Besides of these details, we just loop the items of the array and accumulate the sum in the register r4. Finally, we move r4into r0 for the return value of the function.

In order to call this function we have to put an array into the stack. Consider the following program.

```
1  .data
2
3  .align 4
4
5  big_array :
6  .word 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21
7  .word 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41
8  .word 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61
9  .word 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81
10 .word 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100
11 .word 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116
12 .word 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132
13 .word 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148
14 .word 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164
15 .word 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180
16 .word 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196
17 .word 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212
18 .word 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228
19 .word 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244
20 .word 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255
21
22 .align 4
23
24 message: .asciz "The sum of 0 to 255 is %d\n"
25
26 .text
```

```
.globl main

sum_array_value :
   /* code shown above */

main:
    push {r4, r5, r6, r7, r8, lr}
    /* we will not use r8 but we need to keep the function 8-byte aligned */

    ldr r4, address_of_big_array

    /* Prepare call */

    mov r0, #256  /* Load in the first parameter the number of items
                     r0 ← 256
                     */

    ldr r1, [r4]     /* load in the second parameter the first item of the array */
    ldr r2, [r4, #4] /* load in the third parameter the second item of the array */
    ldr r3, [r4, #8] /* load in the fourth parameter the third item of the array */

    /* before pushing anything in the stack keep its position */
    mov r7, sp

    /* We cannot use more registers, now we have to push them onto the stack
       (in reverse order) */
    mov r5, #255   /* r5 ← 255
                      This is the last item position
                      (note that the first would be in position 0) */

    b .Lcheck_pass_parameter_loop
    .Lpass_parameter_loop:

      ldr r6, [r4, r5, LSL #2]  /* r6 ← *(r4 + r5 * 4).
                                   loads the item in position r5 into r6. Note that
                                   we have to multiply by 4 because this is the size
                                   of each item in the array */
```

```
      push {r6}                 /* push the loaded value to the stack */
      sub r5, r5, #1            /* we are done with the current item,
                                   go to the previous index of the array */
    .Lcheck_pass_parameter_loop:
      cmp r5, #2                /* compute r5 - #2 and update cpsr */
      bne .Lpass_parameter_loop   /* if r5 != #2 branch to pass_parameter_loop */

    /* We are done, we have passed all the values of the array,
       now call the function */
    bl sum_array_value

    /* restore the stack position */
    mov sp, r7

    /* prepare the call to printf */
    mov r1, r0                  /* second parameter, the sum itself */
    ldr r0, address_of_message  /* first parameter, the message */
    bl printf

    pop {r4, r5, r6, r7, r8, lr}
    bx lr

address_of_big_array : .word big_array
address_of_message : .word message
```

In line 40 we start preparing the call to sum_array_value. The first parameter, passed in register r0, is the number of items of this array (in the example hardcoded to 256 items). Then we pass the first three items of the array in registers r1 to r3. Remaining items must be passed on the stack. Remember that in a stack the last item pushed will be the first popped, so if we want our array be laid in the same order we have to push it backwards. So we start from the last item, line 53, and then we load every item and push it onto the stack. Once all the elements have been pushed onto the stack we can call sum_array_value (line 73).
An important caveat when manipulating the stack in this way is that it is very important to restore it and leave it in the same state as it was before preparing the call. This is the reason we keep spin r7 in line 49 and we restore it right after the call in line 76. Forgetting to do this will make further operations on the stack push data onto the wrong place or pop from the stack wrong data. Keeping the stack synched is essential when calling functions.

Passing a big array by reference

Now you are probably thinking that passing a big array through the stack (along with all the boilerplate that this requires) to a function that does not modify it, is, to say the least, wasteful.

Note that, when the amount of data is small, registers r0 to r3 are usually enough, so pass by value is affordable. Passing some data in the stack is fine too, but passing big structures on the stack may harm the performance (especially if our function is being called lots of times).

Can we do better? Yes. Instead of passing copies of the values of the array, would it be possible to pass the address to the array? The answer is, again, yes. This is the concept of pass by reference. When we pass by value, the value of the data passed is somehow copied (either in a register or a stack). Here we will pass a reference (i.e. an address) to the data. So now we are done by just passing the number of items and then the address of the array, and let the function use this address to perform its computation.
Consider the following program, which also sums an array of integers but now passing the array by reference.

```
1  .data
2
3  .align 4
4
5  big_array :
```

```
  /* Same as above */

.align 4

message: .asciz "The sum of 0 to 255 is %d\n"

.text
.globl main

sum_array_ref :
    /* Parameters:
           r0  Number of items
           r1  Address of the array
    */
    push {r4, r5, r6, lr}

    /* We have passed all the data by reference */

    /* r4 will hold the sum so far */
    mov r4, #0      /* r4 ← 0 */
    mov r5, #0      /* r5 ← 0 */

    b .Lcheck_loop_array_sum
    .Lloop_array_sum:
      ldr r6, [r1, r5, LSL #2]   /* r6 ← *(r1 + r5 * 4) */
      add r4, r4, r6             /* r4 ← r4 + r6 */
      add r5, r5, #1             /* r5 ← r5 + 1 */
    .Lcheck_loop_array_sum:
      cmp r5, r0                 /* r5 - r0 and update cpsr */
      bne .Lloop_array_sum       /* if r5 != r0 go to .Lloop_array_sum */

    mov r0, r4  /* r0 ← r4, to return the value of the sum */
    pop {r4, r5, r6, lr}

    bx lr


main:
```

```
2
7
2
8
2
9
3
0
3
1
3
2
3
3
3
4
3
5
3
6
3
7
3
8
3
9
4
0
4
1
4
2
1
```

```
    push {r4, lr}
    /* we will not use r4 but we need to keep the function 8-byte aligned */

    mov r0, #256
    ldr r1, address_of_big_array

    bl sum_array_ref

    /* prepare the call to printf */
    mov r1, r0                  /* second parameter, the sum itself */
    ldr r0, address_of_message  /* first parameter, the message */
    bl printf

    pop {r4, lr}
    bx lr

address_of_big_array : .word big_array
address_of_message : .word message
```

Now the code is much simpler as we avoid copying the values of the array in the stack. We simply pass the address of the array as the second parameter of the function and then we use it to access the array and compute the sum. Much simpler, isn't it?

Modifying data through pointers

We saw at the beginning of the post that we could modify data through pointers. If we pass a pointer to a function we can let the function modify it as well. Imagine a function that takes an integer and increments its. We could do this by returning the value, for instance.

```
increment:
    add r0, r0, #1  /* r0 ← r0 + 1 */
```

This takes the first parameter (in r0) increments it and returns it (recall that we return integers in r0).
An alternative approach, could be receiving a pointer to some data and let the function increment the data at the position defined by the pointer.

```
increment_ptr:
  ldr r1, [r0]      /* r1 ← *r0 */
  add r1, r1, #1    /* r1 ← r1 + 1 */
  str r1, [r0]      /* *r0 ← r1 */
```

For a more elaborated example, let's retake the array code but this time instead of computing the sum of all the values, we will multiply each item by two and keep it in the same array. To prove that we have modified it, we will also print each item.

```
1  /* double_array.s */
2
3  .data
4
5  .align 4
6  big_array :
7   /* Same as above */
8
9  .align 4
1  message: .asciz "Item at position %d has value %d\n"
0
1  .text
1  .globl main
1
2  double_array :
```

```
    /* Parameters:
           r0  Number of items
           r1  Address of the array
    */
    push {r4, r5, r6, lr}

    mov r4, #0      /* r4 ← 0 */

    b .Lcheck_loop_array_double
    .Lloop_array_double:
      ldr r5, [r1, r4, LSL #2]   /* r5 ← *(r1 + r4 * 4) */
      mov r5, r5, LSL #1         /* r5 ← r5 * 2 */
      str r5, [r1, r4, LSL #2]   /* *(r1 + r4 * 4) ← r5 */
      add r4, r4, #1             /* r4 ← r4 + 1 */
    .Lcheck_loop_array_double:
      cmp r4, r0                 /* r4 - r0 and update cpsr */
      bne .Lloop_array_double    /* if r4 != r0 go to .Lloop_array_double */

    pop {r4, r5, r6, lr}

    bx lr

print_each_item:
    push {r4, r5, r6, r7, r8, lr} /* r8 is unused */

    mov r4, #0      /* r4 ← 0 */
    mov r6, r0      /* r6 ← r0. Keep r0 because we will overwrite it */
    mov r7, r1      /* r7 ← r1. Keep r1 because we will overwrite it */


    b .Lcheck_loop_print_items
    .Lloop_print_items:
      ldr r5, [r7, r4, LSL #2]   /* r5 ← *(r7 + r4 * 4) */

      /* Prepare the call to printf */
      ldr r0, address_of_message /* first parameter of the call to printf below */
      mov r1, r4      /* second parameter: item position */
```

```
      mov r2, r5      /* third parameter: item value */
      bl printf       /* call printf */

      add r4, r4, #1              /* r4 ← r4 + 1 */
    .Lcheck_loop_print_items:
      cmp r4, r6                  /* r4 - r6 and update cpsr */
      bne .Lloop_print_items       /* if r4 != r6 goto .Lloop_print_items */

    pop {r4, r5, r6, r7, r8, lr}
    bx lr

main:
    push {r4, lr}
    /* we will not use r4 but we need to keep the function 8-byte aligned */

    /* first call print_each_item */
    mov r0, #256                    /* first_parameter: number of items */
    ldr r1, address_of_big_array    /* second parameter: address of the array */
    bl print_each_item              /* call to print_each_item */

    /* call to double_array */
    mov r0, #256                    /* first_parameter: number of items */
    ldr r1, address_of_big_array    /* second parameter: address of the array */
    bl double_array                /* call to double_array */

    /* second call print_each_item */
    mov r0, #256                    /* first_parameter: number of items */
    ldr r1, address_of_big_array    /* second parameter: address of the array */
    bl print_each_item              /* call to print_each_item */

    pop {r4, lr}
    bx lr

address_of_big_array : .word big_array
address_of_message : .word message
```

If you run this program you will see that the items of the array have been effectively doubled.

```
...
Item at position 248 has value 248
Item at position 249 has value 249
Item at position 250 has value 250
Item at position 251 has value 251
Item at position 252 has value 252
Item at position 253 has value 253
Item at position 254 has value 254
Item at position 255 has value 255
Item at position 0 has value 0
Item at position 1 has value 2
Item at position 2 has value 4
Item at position 3 has value 6
Item at position 4 has value 8
Item at position 5 has value 10
Item at position 6 has value 12
Item at position 7 has value 14
Item at position 8 has value 16
Item at position 9 has value 18
...
```

Returning more than one piece of data

Functions, per the AAPCS convention, return their values in register r0 (and r1 if the returned item is 8 bytes long). We can return more than one thing if we just pass a pointer to some storage (possibly in the stack) as a parameter to the function. More on this topic in a next chapter.

That’s all for today.

Check progress of a mysql database import

February 12, 2014 brafales, 0

If you've ever had to do a huge mysql import, you'll probably understand the pain of not being able to know how long it will take to complete.

At work we use the backup gem to store daily snapshots of our databases, the main one being several gigabytes in size. This gem basically does a mysqldump with configurable options and takes care of maintaining a number of old snapshots, compressing the data and sending notifications on completion and failure of backup jobs.
When the time comes to restore one of those backups, you are basically in the situation in which you simply have to run a mysql command with the exported sql file as input, which can take ages to complete depending on the size of the file and the speed of the system.
The command used to import the database snapshot from the backup gem may look like this:

```
tar -x -v -O -f database_snapshot.tar path_to_the_database_file_inside_the_tar_file.sql.gz | zcat | mysql -u mysql_user
-h mysql_host -ppassword database_name
```

What this command does is untar the gzipped file and sending it as an input to a mysqlcommand to the database you want to restore (passing it through zcat before to gunzip it).
And then the waiting game begins.

There is a way, though, to get an estimate of the amount of work already done, which may be a big help for the impatiens like myself. You only need to make use of the good proc filesystem on Linux.
The first thing you need to do is find out the tar process that you just started:

```
ps ax | grep "database_snapshot\.tar" | grep -v grep
```

This last command assumes that no other processes will have that string on their invocation command lines.

We are really interested in the pid of the process, which we can get with some unix commands and pipes, appending them to the last command:

```
ps ax | grep "database_snapshot\.tar" | grep -v grep | tail -n1 | cut -d" " -f 1
```

This will basically get the last line of the process list output (with tail), separate it in fields using the space as a delimiter and getting the first one (cut command). Note that depending on your OS and the ps command output you may have to tweak this.

After we have the pid of the tar process, we can see what it is doing on the proc filesystem. The information we are interested in is the file descriptors it has open, which will be in the folder /proc/pid/fd. If we list the files in that folder, we will get an output similar to this one:

```
[rails@ip-10-51-43-240 ~]$ sudo ls -l /proc/7719/fd
total 0
lrwx------ 1 rails rails 64 Jan 22 15:38 0 -> /dev/pts/1
l-wx------ 1 rails rails 64 Jan 22 15:38 1 -> pipe:[55359574]
lrwx------ 1 rails rails 64 Jan 22 15:36 2 -> /dev/pts/1
lr-x------ 1 rails rails 64 Jan 22 15:38 3 -> /path/to/database_snaphot.tar
```

The important one for our purposes is the number 3 in this case, which is the file descriptor for the file tar is unpacking. We can get this number using a similar strategy:

```
ls -la /proc/19577/fd/ | grep "database_snaphot\.tar" | cut -d" " -f 9
```

With that number, we can now check the file /proc/pid/fdinfo/fd_id, which will contain something like this:

```
[rails@ip-10-51-43-240 ~]$ cat /proc/7719/fdinfo/3
pos:    4692643840
flags:  0100000
```

The useful part of this list is the pos field. This field is telling us in which position of the file the process is now on. Since tar processes the files sequentially, having this position means we know how much percentage of the file tar has processed so far.

Now the only thing we need to do is check the original file size of the tar file and divide both numbers to get the percentage done.

To get the pos field we can use some more unix commands:

```
cat /proc/7719/fdinfo/3 | head -n1 | cut -f 2
```

To get the original file size, we can use the stat command:

```
stat -c %s /path/to/database_snaphot.tar
```

Finally we can use bc to get the percentage by just dividing both values:

```
echo "`cat /proc/7719/fdinfo/3 | head -n1 | cut -f 2`/`stat -c %s /path/to/database_snaphot.tar` * 100" |
bc -l
```

To put it all together in a nice script, you can use this one as a template:

```
file_path="<full path to your tar db snaphot>"
file_size=`stat -c %s $file_path`
file="<filename of yout db snapshot>"
pid=`ps ax | grep $file | grep -v grep | tail -n1 | cut -d" " -f 1`
fdid=`ls -la /proc/$pid/fd/ | grep $file | cut -d" " -f 9`
pos=`cat /proc/$pid/fdinfo/$fdid | head -n1 | cut -f 2`
echo `echo "$pos / $file_size * 100" | bc -l`
```

I developed this article and script following the tips in this stack overflow answer: http://stackoverflow.com/questions/5748565/how-to-see-progress-of-csv-upload-in-mysql/14851765#14851765

ARM assembler in Raspberry Pi – Chapter 18

May 11, 2014 Roger Ferrer Ibáñez, 7

In this chapter we will delve a bit more into the stack.

Local data

Most of our examples involving data stored in memory (in contrast to data stored in registers) have used global variables. Global variables are global names, i.e. addresses of the memory that we use through labels. These addresses, somehow, pre-exist before the program runs. This is because we define them when defining the program itself.

Sometimes, though, we may want data stored in memory the existence of which is not tied to the program existence but to the dynamic activation of a function. You may recall from previous chapters, that the stack allows us to store data the lifetime of which is the same as the dynamic activation of a function. This is where we will store local variables, which in contrast to global variables, only exist because the function they belong has been dynamically activated (i.e. called/invoked).

In chapter 17 we passed a very big array through the stack in order to pass the array by value. This will lead us to the conclusion that, somehow, parameters act as local data, in particular when they are passed through the stack.

The frame pointer

In ARM, we have plenty of general-purpose registers (up to 16, albeit some of them with very narrow semantics, so actually about 12 are actually useable as general-purpose) and the AAPCS forces us to use registers for the 4 first parameters (r0 to r3, note how this is consistent with the fact that these 4 registers are caller-saved while all other registers are callee-saved). Other architectures, like 386, have a lower number of general purpose registers (about 6) and the usual approach when passing data to functions always involves the stack. This is so because with such a small number of registers, passing parameters through registers, would force the caller to save them, usually in the stack or some other memory, which in turn will usually require at least another register for indexing! By using the stack a few more registers are easily available.

Up to this point one might wonder why we don't always pass everything through the stack and forget about registers r0 to r3. Well, passing through registers is going to be faster as we do not have to mess with loads and stores in the memory. In addition, most functions receive just a few parameters, or at least not much more than 4, so it makes sense to exploit this feature.
But then a problem arises, what if we are passing parameters through the stack and at the same time we have local variables. Both entities will be stored in the stack. How can we deal with the two sources of data which happen to be stored in the same memory area?

Here is where the concept of frame pointer appears. A frame pointer is a sort of marker in the stack that we will use to tell apart local variables from parameters. I want to emphasize the fact that a frame register is almost always unnecessary and one can always devise ways to avoid it. That said, a frame pointer gives us a consistent solution to access local data and parameters in the stack. Of course, most good things come with a price, and the frame pointer is not an exception: we need to use a register for it. Sometimes this restriction may be unacceptable so we can, almost always, get rid of the frame pointer.
Due to its optional nature, the frame pointer is not specified nor mandated by the AAPCS. That said, the usual approach is using register r11. As an extension (apparently undocumented, as far as I have been able to tell) we can use the name fp which is far more informative than just r11. Nothing enforces this choice, we can use any other register as frame pointer. Since we will use fp(i.e. r11) we will have to refrain ourselves from using r11 for any other purpose.

Dynamic link of the activation record

Activation record is a fancy name to specify the context of a called function. This is, the local data and parameters (if passed through the stack) of that function. When a function is written using a frame pointer some bookkeeping is required to correctly maintain the activation record.

First lets examine the typical structure of a function.

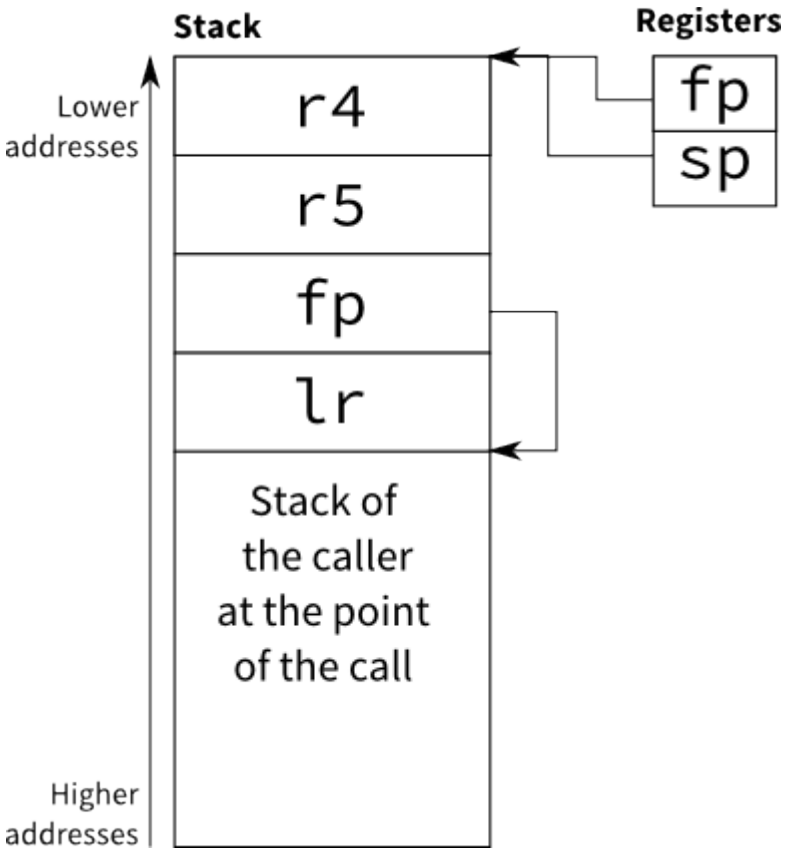
```
function:
  /* Keep callee-saved registers */
  push {r4, lr} /* Keep the callee saved registers */
  ... /* code of the function */
  pop {r4, lr}  /* Restore the callee saved registers */
  bx lr         /* Return from the function */
```

Now let’s modify the function to use a frame pointer (in the code snippet below do not mind the r5 register that only appears here to keep the stack 8-byte aligned).

```
1
2  function:
3    /* Keep callee-saved registers */
4    push {r4, r5, fp, lr} /* Keep the callee saved registers.
5                             We added r5 to keep the stack 8-byte aligned
6                             but the important thing here is fp */
7    mov fp, sp             /* fp ← sp. Keep dynamic link in fp */
8    ... /* code of the function */
9    mov sp, fp             /* sp ← fp. Restore dynamic link in fp */
1    pop {r4, r5, fp, lr}  /* Restore the callee saved registers.
0                             This will restore fp as well */
1    bx lr          /* Return from the function */
1
```

Focus on instructions at line 6 and 8. In line 6 we keep the address of the top of the stack in fp. In line 8 we restore the value of the stack using the value kept in fp. Now you should see why I said that the frame pointer is usually unnecessary: if the sp register does not change between lines 6 and 8, having a frame pointer will be pointless, why should we restore a register that didn’t change?

Let’s assume for now that the frame pointer is going to be useful. What we did in instruction line 6 is setting the dynamic link. The stack and registers will look like this after we have set it.

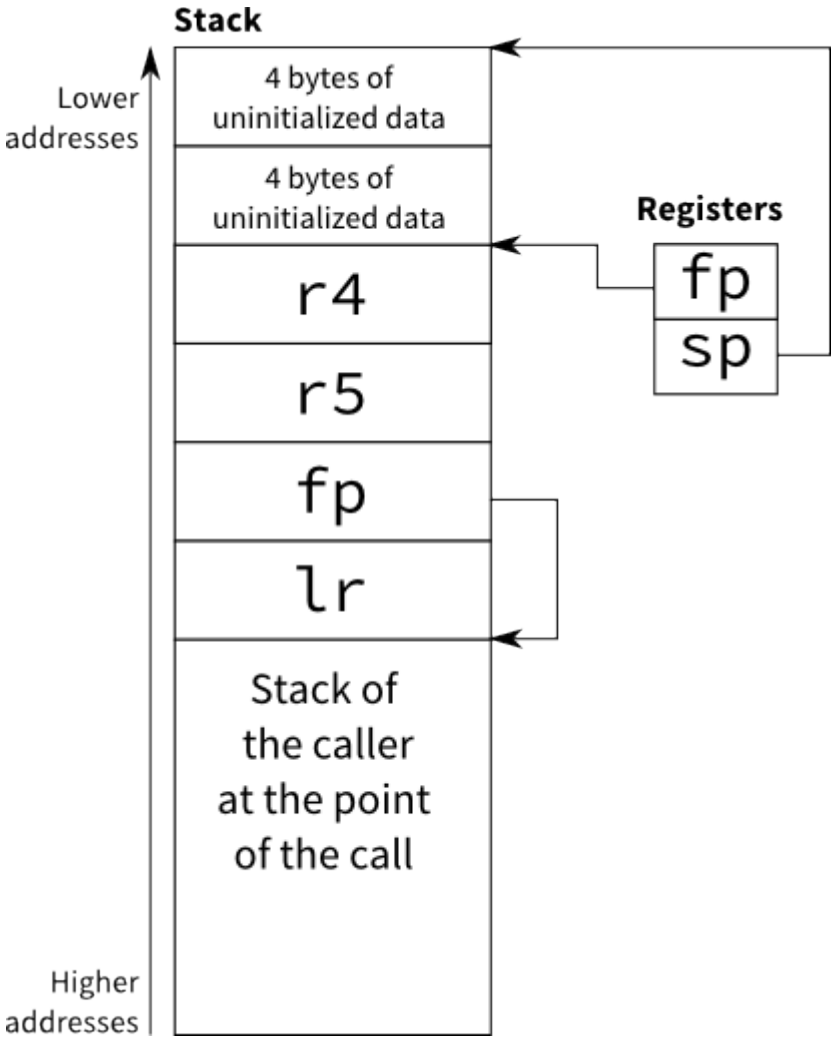


As you can see, the fp register will point to the top of the stack. But note that in the stack we have the value of the old fp (the value of the fp in the function that called us). If we assume that our caller also uses a frame pointer, then the fp we kept in the stack of the callee points to the top of the stack when our caller was called.
But still this looks useless because both registers fp and sp in the current function point to the same position in the stack.
Let's proceed with the example, make sure you check line 7.

```
1 function:
2   /* Keep callee-saved registers */
3   push {r4, r5, fp, lr} /* Keep the callee saved registers.
```

```
4
5                          We added r5 to keep the stack 8-byte aligned
6                          but the important thing here is fp */
7     mov fp, sp             /* fp ← sp. Keep dynamic link in fp */
8     sub sp, sp, #8         /* Enlarge the stack by 8 bytes */
9     ... /* code of the function */
1     mov sp, fp             /* sp ← fp. Restore dynamic link in fp */
0     pop {r4, r5, fp, lr}   /* Restore the callee saved registers.
1                          This will restore fp as well */
1     bx lr          /* Return from the function */
1
2
```

Now, after line 7, the stack and registers will look like this.



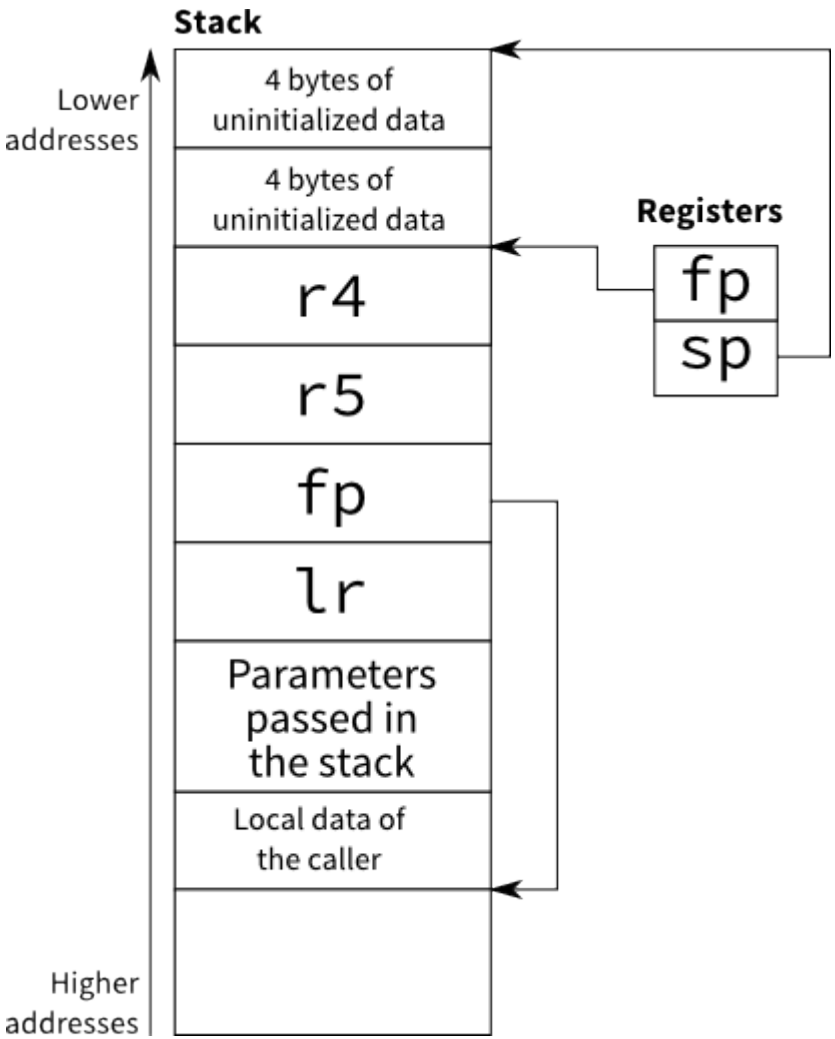
Can you see the range of data from sp to fp? This is the local data of our function. We will keep local variables of a function in this space when using a frame pointer. We simply have to allocate stack space by decreasing the value of sp (and ensuring it is 8-byte aligned per AAPCS requirements).

Now consider the instruction mov sp, fp near the end of the function. What it does is leaving the state of the registers just like before we enlarged the stack (before the sub sp, sp, #8). And voilà, we have freed all the stack our function was

using. A bonus of this approach is that it does not require keeping anywhere the amount of bytes we reserved in the stack. Neat, isn’t it?

What about parameters passed in the stack?

A player is still missing in our frame pointer approach: parameters passed through the stack. Let’s assume that our function may receive parameters in the stack and we have enlarged the stack by subtracting sp. The whole picture looks like this.



I want you to note that I just lied a bit in the two first figures. In them, the old fp pointer kept in the stack pointed to the top of the stack of the caller. Not exactly, it will point to the base of the local data of the caller, exactly like happens with the fp register in the current function.

Indexing through the frame pointer

When we are using a frame pointer a nice property (that maybe you have already deduced from the figures above) holds: local data is always at lower addresses than the address pointed by fpwhile parameters passed in the stack (if any) will always be at higher addresses than the one pointed by fp. It must be possible to access both kinds of local data through fp.
In the following example we will use a function that receives an integer by reference (i.e. an address to an integer) and then squares that integer.

```
void sq(int *c)
{
  (*c) = (*c) * (*c);
}
```

You may be wondering why the function sq has a parameter by reference (should not it be easier to return a value?), but bear with me for now. We can (should?) implement sq without using a frame pointer due to its simplicity.

```
1 sq:
2   ldr r1, [r0]   /* r1 ← (*r0) */
3   mul r1, r1, r1 /* r1 ← r1 * r1 */
4   str r1, [r0]   /* (*r0) ← r1 */
5   bx lr          /* Return from the function */
```

Now consider the following function that returns the sum of the squares of its five parameters. It uses the function sq defined above.

```
int sq_sum5(int a, int b, int c, int d, int e)
{
  sq(&a);
  sq(&b);
  sq(&c);
  sq(&d);
  sq(&e);
  return a + b + c + d + e;
}
```

Parameters a, b, c and d will be passed through registers r0, r1, r2, and r3 respectively. The parameter e will be passed through the stack. The function sq, though, expects a reference, i.e. an address, to an integer and registers do not have

an address. This means we will have to allocate temporary local storage for these registers. At least one integer will have to be allocated in the stack in order to be able to call sq but for simplicity we will allocate four of them.
This time we will use a frame pointer to access both the local storage and the parameter e.

```
sq_sum5:
  push {fp, lr}         /* Keep fp and all callee-saved registers. */
  mov fp, sp            /* Set the dynamic link */

  sub sp, sp, #16      /* sp ← sp - 16. Allocate space for 4 integers in the stack */
  /* Keep parameters in the stack */
  str r0, [fp, #-16]    /* *(fp - 16) ← r0 */
  str r1, [fp, #-12]    /* *(fp - 12) ← r1 */
  str r2, [fp, #-8]     /* *(fp - 8) ← r2 */
  str r3, [fp, #-4]     /* *(fp - 4) ← r3 */

  /* At this point the stack looks like this
     | Value  |  Address(es)
     +--------+----------------------
     |   r0   |  [fp, #-16], [sp]
     |   r1   |  [fp, #-12], [sp, #4]
     |   r2   |  [fp, #-8],  [sp, #8]
     |   r3   |  [fp, #-4],  [sp, #12]
     |   fp   |  [fp],       [sp, #16]
     |   lr   |  [fp, #4],   [sp, #20]
     |   e    |  [fp, #8],   [sp, #24]
     v
    Higher
    addresses
  */

  sub r0, fp, #16    /* r0 ← fp - 16 */
  bl sq              /* call sq(&a); */
  sub r0, fp, #12    /* r0 ← fp - 12 */
  bl sq              /* call sq(&b); */
  sub r0, fp, #8     /* r0 ← fp - 8 */
  bl sq              /* call sq(&c); */
  sub r0, fp, #4     /* r0 ← fp - 4 */
```

```
2
2
3
2
4
2
5
2
6
2
7
2
8
2
9
3
0
3
1
3
2
3
3
3
4
0
```

```
    bl sq               /* call sq(&d) */
    add r0, fp, #8      /* r0 ← fp + 8 */
    bl sq               /* call sq(&e) */

    ldr r0, [fp, #-16] /* r0 ← *(fp - 16). Loads a into r0 */
    ldr r1, [fp, #-12] /* r1 ← *(fp - 12). Loads b into r1 */
    add r0, r0, r1      /* r0 ← r0 + r1 */
    ldr r1, [fp, #-8]  /* r1 ← *(fp - 8). Loads c into r1 */
    add r0, r0, r1      /* r0 ← r0 + r1 */
    ldr r1, [fp, #-4]  /* r1 ← *(fp - 4). Loads d into r1 */
    add r0, r0, r1      /* r0 ← r0 + r1 */
    ldr r1, [fp, #8]   /* r1 ← *(fp + 8). Loads e into r1 */
    add r0, r0, r1      /* r0 ← r0 + r1 */

    mov sp, fp          /* Undo the dynamic link */
    pop {fp, lr}        /* Restore fp and callee-saved registers */
    bx lr               /* Return from the function */
```

As you can see, we first store all parameters (but e) in the local storage. This means that we need to enlarge the stack enough, as usual, by subtracting sp (line 5). Once we have the storage then we can do the actual store by using the fp register (lines 7 to 10). Note the usage of negative offsets, because local data will always be in lower addresses than the address in fp. As mentioned above, the parameter e does not have to be stored because it is already in the stack, in a positive offset from fp (i.e. at a higher address than the address in fp).

Note that, in this example, the frame pointer is not indispensable as we could have used sp to access all the required data (see the representation of the stack in lines 12 to 21).

In order to call sq we have to pass the addresses of the several integers, so we compute the address by subtracting fp the proper offset and storing it in r0, which will be used for passing the first (and only) parameter

of sq (lines 27 to 36). See how, to pass the address of e, we just compute an address with a positive offset (line 35). Finally we add the values by loading them again in r0 and r1 and using r0 to accumulate the additions (lines 38 to 46). An example program that calls sq_sum5(1, 2, 3, 4, 5) looks like this.

```
/* squares.s */
.data

.align 4
message: .asciz "Sum of 1^2 + 2^2 + 3^2 + 4^2 + 5^2 is %d\n"

.text

sq:
  <<defined above>>

sq_sum5:
  <<defined above>>

.globl main

main:
    push {r4, lr}          /* Keep callee-saved registers */

    /* Prepare the call to sq_sum5 */
    mov r0, #1             /* Parameter a ← 1 */
    mov r1, #2             /* Parameter b ← 2 */
    mov r2, #3             /* Parameter c ← 3 */
    mov r3, #4             /* Parameter d ← 4 */

    /* Parameter e goes through the stack,
       so it requires enlarging the stack */
    mov r4, #5             /* r4 ← 5 */
    sub sp, sp, #8         /* Enlarge the stack 8 bytes,
                              we will use only the
                              topmost 4 bytes */
    str r4, [sp]           /* Parameter e ← 5 */
    bl sq_sum5             /* call sq_sum5(1, 2, 3, 4, 5) */
    add sp, sp, #8         /* Shrink back the stack */
```

```
    /* Prepare the call to printf */
    mov r1, r0              /* The result of sq_sum5 */
    ldr r0, address_of_message
    bl printf               /* Call printf */

    pop {r4, lr}            /* Restore callee-saved registers */
    bx lr


address_of_message: .word message
```

```
1
4
2
4
3
4
4
4
5
```

```
$ ./square
Sum of 1^2 + 2^2 + 3^2 + 4^2 + 5^2 is 55
```

That’s all for today.

ARM assembler in Raspberry Pi – Chapter 19

May 24, 2014 Roger Ferrer Ibáñez, 10

So far our small assembler programs have output messages using printf and some of them have read input using scanf. These two functions are implemented in the C library, so they are more or less supported in any environment supporting the C language. But how does a program actually communicate with the world?

The operating system

Our Raspberry Pi runs Raspbian. Raspbian is an operating system based on Debian on top of the Linux kernel. The operating system is a piece of software (usually a collection of pieces that together form a useful system) that enables and manages the resources required by programs to run. Which sort of resources, you may be wondering? Well, many different kinds of them: processes, files, network devices, network communications, screens, printers, terminals, timers, etc.

From the point of view of the program, the operating system is just a big servant providing lots of services to the program. But the operating system is also a caretaker, taking action when something goes wrong or programs (sometimes caused by the users of the operating system) attempt to do something that they are not authorized to do. In our case, Linux is the kernel of the Raspbian operating system. The kernel provides the most basic functionality needed to provide these services (sometimes it provides them directly, sometimes it just provides the minimal essential functionality so they can be implemented). It can be viewed as a foundational program that it is always running (or at least, always ready) so it can serve the requests of the programs run by the users. Linux is a UNIX®-like kernel and as such shares lots of features with the long lineage of UNIX®-like operating systems.

Processes

In order to assign resources, the operating system needs an entity to which grant such resources. This entity is called a process. A process is a running program. The same program may be run several times, each time it is run it is a different process.

System calls

A process interacts with the operating system by performing system calls. A system call is conceptually like calling a function but more sophisticated. It is more sophisticated because now we need to satisfy some extra security requirements. An operating system is a critical part of a system and we cannot let processes dodge the operating system control. A usual function call offers no protection of any kind. Any strategy we could design on top of a plain function call would easily be possible to circumvent. As a consequence of this constraint, we need support from the architecture (in our case ARM) in order to safely and securely implement a system call mechanism.

In Linux ARM we can perform a system call by using the instruction swi. This instruction means software interruption and its sole purpose is to make a system call to the operating system. It receives a 24-bit operand that is not used at all by the processor but could be used by the the operating system to tell which service has been requested. In Linux such approach is not used and a 0 is set as the operand instead. So, in summary, in Linux we will always use swi #0 to perform a system call.

An operating system, and particularly Linux, provides lots of services through system calls so we need a way to select one of them. We will do this using the register r7. System calls are similar to function calls in that they receive parameters. No system call in Linux receives more than 7 arguments and the arguments are passed in registers r0 to r6. If the system call returns some value it will be returned in register r0.

Note that the system call convention is incompatible with the convention defined by the AAPCS, so programs will need specific code that deals with a system call. In particular, it makes sense to wrap these system calls into normal functions, that externally, i.e. from the point of the caller, follow the AAPCS. This is precisely the main purpose of the C library. In Linux, the C library is usually GNU Libc (but others can be used in Linux). These libraries hide the extra complexity of making system calls under the appearance of a normal function call.

Hello world, the system call way

As a simple illustration of calling the operating system we will write the archetypical “Hello world” program using system calls. In this case we will call the function write. Write receives three parameters: a file descriptor where we will write some data, a pointer to the data that will be written and the size of such data. Of these three, the most obscure may be now the file descriptor. Without entering into much details, it is just a number that identifies a file assigned to the process. Processes usually start with three preassigned files: the standard input, with the number 0, the standard

output, with the number 1, and the standard error, with the number 2. We will write our messages to the standard output, so we will use the file descriptor 1.

The “ea-C” way

Continuing with our example, first we will call write through the C library. The C library follows the AAPCS convention. The prototype of the write system call can be found in the Linux man pages and is as follows.

```
ssize_t write(int fd, const void *buf, size_t count);
```

Here both size_t and ssize_t are 32-bit integers, where the former is unsigned and the latter signed. Equipped with our knowledge of the AAPCS and ARM assembler it should not be hard for us to perform a call like the following

```
const char greeting[13] = "Hello world\n";
write(1, greeting, sizeof(greeting)); // Here sizeof(greeting) is 13
```

Here is the code

```
/* write_c.s */

.data

greeting: .asciz "Hello world\n"
after_greeting:

/* This is an assembler constant: the assembler will compute it. Needless to say
   that this must evaluate to a constant value. In this case we are computing the
   difference of addresses between the address after_greeting and greeting. In this
   case it will be 13 */
.set size_of_greeting, after_greeting - greeting

.text

.globl main

main:
    push {r4, lr}

    /* Prepare the call to write */
    mov r0, #1                /* First argument: 1 */
```

```
    ldr r1, addr_of_greeting  /* Second argument: &greeting */
    mov r2, #size_of_greeting /* Third argument: sizeof(greeting) */
    bl write                  /* write(1, greeting, sizeof(greeting));

    mov r0, #0
    pop {r4, lr}
    bx lr

addr_of_greeting : .word greeting
```

The system call way

Ok, calling the system call through the C library was not harder than calling a normal function. Let's try the same directly performing a Linux system call. First we have to identify the number of the system call and put it in r7. The call write has the number 4 (you can see the numbers in the file /usr/include/arm-linux-gnueabihf/asm/unistd.h). The parameters are usually the same as in the C function, so we will use registers r0, r1 and r2 likewise.

```
/* write_sys.s */

.data


greeting: .asciz "Hello world\n"
after_greeting:

.set size_of_greeting, after_greeting - greeting

.text

.globl main

main:
    push {r7, lr}

    /* Prepare the system call */
    mov r0, #1                  /* r0 ← 1 */
    ldr r1, addr_of_greeting    /* r1 ← &greeting */
    mov r2, #size_of_greeting   /* r2 ← sizeof(greeting) */
```

```
    mov r7, #4                  /* select system call 'write' */
    swi #0                      /* perform the system call */

    mov r0, #0
    pop {r7, lr}
    bx lr

addr_of_greeting : .word greeting
```

As you can see it is not that different to a function call but instead of branching to a specific address of code using bl we use swi #0. Truth be told, it is rather unusual to perform system calls directly. It is almost always preferable to call the C library instead.

That’s all for today.

TinyMCE checkbox toggler for jQuery

June 9, 2014 brafales, 0

Here's a small jQuery code snippet that you can use to have an easy to use checkbox toggler to enable or disable a TinyMCE editor with ease (tested on TinyMCE version 4 and jQuery version 2.1.1).
It's really easy to use. You just need to create a checkbox element with the class tiny_mce_toggler and a data attribute with the key editor and the text area id used as a TinyMCE editor as a value. The snippet can be easily extracted if you want to use it differently.

Here is the javascript snippet:

```
$(function() {
  var TinyMceToggler = function(_checkbox){
    var checkbox = $(_checkbox);
    var editor = checkbox.data('editor');

    checkbox.click(function(){
      if (this.checked) {
      console.log("Add");
        tinyMCE.execCommand( 'mceAddEditor', false, editor );
      }
      else {
      console.log("Remove");
        tinyMCE.execCommand( 'mceRemoveEditor', false, editor );
      }
    });
  };

  $("input.tiny_mce_toggler").each(function(){
    new TinyMceToggler(this);
  });
});
```

And here you can see how to integrate it on a page with a TinyMCE editor:

```
<!DOCTYPE html>
```

```
<html>
<head>
  <meta charset="utf-8" />
  <title>TinyMCE - Toggler</title>
  <link type="text/css" rel="stylesheet" href="http://moxiecode.cachefly.net/tinymce/v8/css/all.min.css?v=8" />
  <script src="//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js"></script>
  <script type="text/javascript" src="javascript/tinymce/tinymce.min.js"></script>
  <script type="text/javascript">
  $(function() {
    tinymce.init({
        selector: "textarea"
    });

    var TinyMceToggler = function(_checkbox){
      var checkbox = $(_checkbox);
      var editor = checkbox.data('editor');

      checkbox.click(function(){
        if (this.checked) {
        console.log("Add");
          tinyMCE.execCommand( 'mceAddEditor', false, editor );
        }
        else {
        console.log("Remove");
          tinyMCE.execCommand( 'mceRemoveEditor', false, editor );
        }
      });
    };

    $("input.tiny_mce_toggler").each(function(){
      new TinyMceToggler(this);
    });
  });
  </script>
</head>
<body>
<form method="post" action="#">
```

```
    <input type="checkbox" data-editor="tiny" checked="checked" class="tiny_mce_toggler" style="display: block;"
/>
    <textarea id="tiny"></textarea>
</form>
</body>
</html>
```

You can run the example on this fiddle: http://fiddle.tinymce.com/7jeaab

ARM assembler in Raspberry Pi – Chapter 20

August 20, 2014 Roger Ferrer Ibáñez, 5
Today we will see how to make indirect calls.

Labels

One of the distinguishing features of assemblers is the shortage of symbolic information. The only symbolic support available at this (low) level are labels. We already know that labels are just addresses to the memory of the program (both data and code).
When we define a function in assembler, we define a label for it.

```
fun: /* label 'fun' */
  push {r4, r5}
  ...
  pop {r4, r5}
  bx lr
```

Later (or before, assemblers usually do not care) we use the label. So a call like

```
  bl fun
```

Is saying to the assembler, I'm using fun here, but you have to put the appropiate address there when generating machine code, ok?.
In reality, calling a function is usually much more involved but at the end there is a label that brings us to the function.

Our first indirect call

What if rather than using the label of a function, we were able to keep the addres of a function (or several of them) somewhere and call a function indirectly? Let's try that. First, we will start with a basic Hello world that uses a label. We will call this a direct call.

```
1 .data     /* data section */
2 .align 4  /* ensure the next label is 4-byte aligned */
```

```
3
4
5
6  message: .asciz "Hello world\n"
7
8  .text     /* text section (= code) */
9
1  .align 4  /* ensure the next label is 4-byte aligned */
0  say_hello:
1      push {r4, lr}            /* keep lr because we call printf,
1                                  we keep r4 to keep the stack 8-byte
1                                  aligned, as per AAPCS requirements */
2      /* Prepare the call to printf */
1      ldr r0, addr_of_message  /* r0 ← &message */
3      bl printf                /* call printf */
1      pop {r4, lr}             /* restore r4 and lr */
4      bx lr                    /* return to the caller */
1
5  .align 4  /* ensure the next label is 4-byte aligned */
1  addr_of_message: .word message
6
1  .globl main /* state that 'main' label is global */
7  .align 4  /* ensure the next label is 4-byte aligned */
1  main:
8      push {r4, lr}            /* keep lr because we call say_hello,
1                                  we keep r4 to keep the stack 8-byte
9                                  aligned, as per AAPCS requirements */
2      bl say_hello             /* call say_hello, directly, using the label */
0
2      mov r0, #0               /* return from the program, set error code */
1      pop {r4, lr}             /* restore r4 and lr */
2      bx lr                    /* return to the caller (the system) */
2
2
3
```

Now let’s add some storage in the data section to keep the address of say_hello.

```
.data     /* data section */
```

```
...
.align 4  /* ensure the next label is 4-byte aligned */
ptr_of_fun: .word 0   /* we set its initial value zero */
```

Now we will add a new function make_indirect_call that does the indirect call using the value stored in ptr_of_fun.

```
.align 4
make_indirect_call:
    push {r4, lr}            /* keep lr because we call printf,
                                we keep r4 to keep the stack 8-byte
                                aligned, as per AAPCS requirements */
    ldr r0, addr_ptr_of_fun  /* r0 ← &ptr_of_fun */
    ldr r0, [r0]             /* r0 ← *r0 */
    blx r0                   /* indirect call to r0 */
    pop {r4, lr}             /* restore r4 and lr */
    bx lr                    /* return to the caller */

addr_ptr_of_fun: .word ptr_of_fun
```

Doing an indirect call is done using the instruction blx. It behaves like bl but expects a register rather than a label.

Yoy may be wondering whether we could have used bx rather than blx. We cannot. The instruction bx does not set the lr register to the next instruction, like bl and blx do. Thus, we would call the function but it would not be able to return: it would jump back to the wrong place! (try to think which one).

Now in the main we will keep the address of say_hello in ptr_of_fun and call make_indirect_call.

```
main:
    push {r4, lr}            /* keep lr because we call printf,
                                we keep r4 to keep the stack 8-byte
                                aligned, as per AAPCS requirements */

    ldr r1, addr_say_hello   /* r1 ← &say_hello */
    ldr r0, addr_ptr_of_fun  /* r0 ← &addr_ptr_of_fun */
    str r1, [r0]             /* *r0 ← r1
                                this is
                                ptr_of_fun ← &say_hello */

    bl make_indirect_call    /* call make_indirect_call */
```

```
    mov r0, #0                  /* return from the program, set error code */
    pop {r4, lr}                /* restore r4 and lr */
    bx lr                       /* return to the caller (the system) */

addr_ptr_of_fun: .word ptr_of_fun
addr_say_hello : .word say_hello
```

Note that, in the function make_indirect_call we did

```
    ldr r0, addr_ptr_of_fun  /* r0 ← &ptr_of_fun */
    ldr r0, [r0]             /* r0 ← *r0 */
```

while in the main we do

```
    ldr r1, addr_say_hello   /* r1 ← &say_hello */
```

This is a similar case like arrays: when we load an array address, we do not need to load again (as it happens when we load simple scalars). This is because if we did that, we would be loading the first element of the array. With functions a similar thing happens: the function itself, its label, is already an address. If we did another load we would be loading an instruction into the register!! Not quite what we want 🙂

In the function make_indirect_call we are not loading a function but a pointer to a function (addr_ptr_of_fun), so we have to do the typical double load we do for scalars (because at the end, a pointer is just an integer that happens to be an address of the memory of our program).

Feel the power

The last example does not look very interesting, but being able to call a function indirectly is a very powerful thing. It allows us to keep the address of a function somewhere and call it. It allows us to pass the address of a function to another function. Why would we want to do that? Well, it is a rudimentary, yet effective, way of passing code to another function.

As an example, let’s make a generic greeter function which receives a greeting function as a parameter. This way the exact greeting is actually deferred to another function.

```
.data     /* data section */
.align 4  /* ensure the next label is 4-byte aligned */
message_1: .asciz "Hello\n"
.align 4  /* ensure the next label is 4-byte aligned */
message_2: .asciz "Bonjour\n"

.text     /* text section (= code) */

.align 4  /* ensure the next label is 4-byte aligned */
say_hello:
    push {r4, lr}            /* keep lr because we call printf,
                                we keep r4 to keep the stack 8-byte
                                aligned, as per AAPCS requirements */
    /* Prepare the call to printf */
    ldr r0, addr_of_message_1 /* r0 ← &message */
    bl printf                 /* call printf */
    pop {r4, lr}              /* restore r4 and lr */
    bx lr                     /* return to the caller */

.align 4  /* ensure the next label is 4-byte aligned */
addr_of_message_1: .word message_1

.align 4  /* ensure the next label is 4-byte aligned */
say_bonjour:
    push {r4, lr}            /* keep lr because we call printf,
                                we keep r4 to keep the stack 8-byte
                                aligned, as per AAPCS requirements */
    /* Prepare the call to printf */
    ldr r0, addr_of_message_2 /* r0 ← &message */
    bl printf                 /* call printf */
    pop {r4, lr}              /* restore r4 and lr */
    bx lr                     /* return to the caller */

.align 4  /* ensure the next label is 4-byte aligned */
addr_of_message_2: .word message_2

.align 4
greeter:
```

```
    push {r4, lr}            /* keep lr because we call printf,
                                we keep r4 to keep the stack 8-byte
                                aligned, as per AAPCS requirements */
    blx r0                   /* indirect call to r0 */
    pop {r4, lr}             /* restore r4 and lr */
    bx lr                    /* return to the caller */

.globl main /* state that 'main' label is global */
.align 4  /* ensure the next label is 4-byte aligned */
main:
    push {r4, lr}            /* keep lr because we call printf,
                                we keep r4 to keep the stack 8-byte
                                aligned, as per AAPCS requirements */

    ldr r0, addr_say_hello   /* r0 ← &say_hello */
    bl greeter               /* call greeter */

    ldr r0, addr_say_bonjour /* r0 ← &say_bonjour */
    bl greeter               /* call greeter */

    mov r0, #0               /* return from the program, set error code */
    pop {r4, lr}             /* restore r4 and lr */
    bx lr                    /* return to the caller (the system) */

addr_say_hello : .word say_hello
addr_say_bonjour : .word say_bonjour
```

If we run it

```
$ ./greeter_01
Hello
Bonjour
```

You are probably not impressed by the output of this previous program. So let's try to make it more interesting: we will greet people generically, some people will be greeted in English and some other will be greeted in French.

Let's start defining a bunch of data that we will require for this example. First greeting messages in English and French. Note that we will greet the person by name, so we will use a printf format string.

```
.data     /* data section */

.align 4  /* ensure the next label is 4-byte aligned */
message_hello: .asciz "Hello %s\n"
.align 4  /* ensure the next label is 4-byte aligned */
message_bonjour: .asciz "Bonjour %s\n"
```

Next we will define some tags that we will use to tag people as English or French. This tag will contain the address to the specific greeting function. The English tag will have the address of say_hello and the French tag will have the address of say_bonjour.

```
/* tags of kind of people */
.align 4  /* ensure the next label is 4-byte aligned */
person_english : .word say_hello /* tag for people
                                    that will be greeted
                                    in English */
.align 4  /* ensure the next label is 4-byte aligned */
person_french : .word say_bonjour /* tag for people
                                     that will be greeted
                                     in French */
```

Let’s define some names that we will use later, when defining people.

```
/* several names to be used in the people definition */
.align 4
name_pierre: .asciz "Pierre"
.align 4
name_john: .asciz "John"
.align 4
name_sally: .asciz "Sally"
.align 4
name_bernadette: .asciz "Bernadette"
```

```
2
3
2
4
2
5
2
6
```

And now define some people. Every person is actually a pair formed by an address to their name and an address to their tag.

```
2
8
2
9
3
0
3
1
3
2
3
3
3
4
3
5
```

```
.align 4
person_john: .word name_john, person_english
.align 4
person_pierre: .word name_pierre, person_french
.align 4
person_sally: .word name_sally, person_english
.align 4
person_bernadette: .word name_bernadette, person_french
```

Finally let’s group every person in an array. The array contains addresses to each people (not the people themselves).

```
3
8
3
9
```

```
/* array of people */
people : .word person_john, person_pierre, person_sally, person_bernadette
```

Now let’s define the code. These are the two specific functions for each language (English and French). Note that we already named their labels in the tags above.

```
4
1
4  .text     /* text section (= code) */
2
4  .align 4  /* ensure the next label is 4-byte aligned */
3  say_hello:
4      push {r4, lr}            /* keep lr because we call printf,
4                                  we keep r4 to keep the stack 8-byte
4                                  aligned, as per AAPCS requirements */
5      /* Prepare the call to printf */
4      mov r1, r0               /* r1 ← r0 */
6      ldr r0, addr_of_message_hello
4                              /* r0 ← &message_hello */
7      bl printf                /* call printf */
4      pop {r4, lr}             /* restore r4 and lr */
8      bx lr                    /* return to the caller */
4
9
5  .align 4  /* ensure the next label is 4-byte aligned */
0  addr_of_message_hello: .word message_hello
5
1  .align 4  /* ensure the next label is 4-byte aligned */
5  say_bonjour:
2      push {r4, lr}            /* keep lr because we call printf,
5                                  we keep r4 to keep the stack 8-byte
3                                  aligned, as per AAPCS requirements */
5      /* Prepare the call to printf */
4      mov r1, r0               /* r1 ← r0 */
5      ldr r0, addr_of_message_bonjour
5                              /* r0 ← &message_bonjour */
5      bl printf                /* call printf */
6      pop {r4, lr}             /* restore r4 and lr */
5      bx lr                    /* return to the caller */
7
5  .align 4  /* ensure the next label is 4-byte aligned */
8  addr_of_message_bonjour: .word message_bonjour
5
9
```

Before we go to the interesting function, let's define the main function.

```
.globl main /* state that 'main' label is global */
.align 4  /* ensure the next label is 4-byte aligned */
main:
    push {r4, r5, r6, lr}    /* keep callee saved registers that we will modify */

    ldr r4, addr_of_people   /* r4 ← &people */
    /* recall that people is an array of addresses (pointers) to people */

    /* now we loop from 0 to 4 */
    mov r5, #0               /* r5 ← 0 */
    b check_loop             /* branch to the loop check */

    loop:
      /* prepare the call to greet_person */
      ldr r0, [r4, r5, LSL #2]  /* r0 ← *(r4 + r5 << 2)   this is
                                   r0 ← *(r4 + r5 * 4)
                                   recall, people is an array of addresses,
                                   so this is
                                   r0 ← people[r5]
                                */
      bl greet_person           /* call greet_person */
      add r5, r5, #1            /* r5 ← r5 + 1 */
    check_loop:
      cmp r5, #4                /* compute r5 - 4 and update cpsr */
      bne loop                  /* if r5 != 4 branch to loop */

    mov r0, #0               /* return from the program, set error code */
    pop {r4, r5, r6, lr}     /* callee saved registers */
    bx lr                    /* return to the caller (the system) */

addr_of_people : .word people
```

```
1
1
1
2
1
```

As you can see, what we do here is to load elements 0 to 3 of the people array and call the function greet_person. Every element in people array is a pointer, so we can put them in a register, in this case r0 because it will be the first parameter of greet_person.
Let's see now the code for the function greet_person.

```
7
5
7
6
7
7
7
8
7
9
8
0
8
1
8
2
8
3
8
4
8
5
8
6
8
7
8
```

```
/* This function receives an address to a person */
.align 4
greet_person:
    push {r4, lr}           /* keep lr because we call printf,
                               we keep r4 to keep the stack 8-byte
                               aligned, as per AAPCS requirements */

    /* prepare indirect function call */
    mov r4, r0              /* r0 ← r4, keep the first parameter in r4 */
    ldr r0, [r4]            /* r0 ← *r4, this is the address to the name
                               of the person and the first parameter
                               of the indirect called function*/

    ldr r1, [r4, #4]        /* r1 ← *(r4 + 4) this is the address
                               to the person tag */
    ldr r1, [r1]            /* r1 ← *r1, the address of the
                               specific greeting function */

    blx r1                  /* indirect call to r1, this is
                               the specific greeting function */

    pop {r4, lr}            /* restore r4 and lr */
    bx lr                   /* return to the caller */
```

```
8
8
9
7
```

In register r0 we have the address of a person. We move it to r4 for convenience as r0 will be used for the indirectly called function. Then we load the name of the person, found in [r4], this is [r4, #0] (this is *(r4 + 0), so *r4) into r0. Then we load the person tag, found 4 bytes after the name (remember that the name of the person is an address, so it takes 4 bytes in ARM). The tag itself is not very useful except because it allows us to get the specific greeting function (either say_hello or say_bonjour). So we load [r4, #4], the address of the tag, in r1. Ok, now r1 contains the address of the tag and we know that the first 4 bytes of a tag contain the specific greeting function.
If we run this program the output is:

```
$ ./greeter_02
Hello John
Bonjour Pierre
Hello Sally
Bonjour Bernadette
```

Late binding and object orientation

In the last example we have implemented, in a very simple way, a feature of the object-oriented programming (OOP) called late binding, which means that one does not know which function is called for a given object.
In our example the objects are of kind Person. Every Person can be greeted, this is what greet_person does. We do not have objects of kind Person really, but EnglishPerson and FrenchPerson. When you greet an EnglishPerson you expect to greet him/her with Hello, when you greet a FrenchPerson you expect to greet him/her with Bonjour.
If you know C++ (or Java), you'll quickly realize that our last example actually implements something like this.

```
struct Person
{
  const char* name;
  virtual void greet() = 0;
};

struct EnglishPerson : Person
{
```

```
  virtual void greet()
  {
    printf("Hello %s\n", this->name);
  }
};

struct FrenchPerson : Person
{
  virtual void greet()
  {
    printf("Bonjour %s\n", this->name);
  }
};
```

In the snippet above, this is the Person we passed to our function greet_person. That parameter allowed us to retrieve the name of the person (this->name) and the specific version of greet we wanted.
I hope that this last example, albeit a bit long, actually shows you the power of indirect calls.

This is all for today.

ARM assembler in Raspberry Pi – Chapter 22

December 20, 2014 Roger Ferrer Ibáñez, 4

Several times in previous chapters we have talked about ARM as an architecture that has several features aimed at embedding systems. In embedded systems memory is scarce and expensive, so designs that help reduce the memory footprint are very welcome. Today we will see another of these features: the Thumb instruction set.

The Thumb instruction set

In previous installments we have been working with the ARMv6 instruction set (the one implemented in the Raspberry Pi). In this instruction set, all instructions are 32-bit wide, so every instruction takes 4 bytes. This is a common design since the arrival of RISC processors. That said, in some scenarios such codification is overkill in terms of memory consumption: many platforms are very simple and rarely need all the features provided by the instruction set. If only they could use a subset of the original instruction set that can be encoded in a smaller number of bits!

So, this is what the Thumb instruction set is all about. They are a reencoded subset of the ARM instructions that take only 16 bits per instructions. This means that we will have to waive away some instructions. As a benefit our code density is higher: most of the time we will be able to encode the code of our programs in half the space.

Support of Thumb in Raspbian

While the processor of the Raspberry Pi properly supports Thumb, there is still some software support that unfortunately is not provided by Raspbian. This means that we will be able to write some snippets in Thumb but in general this is not supported (if you try to use Thumb for a full C program you will end with a sorry, unimplemented message by the compiler).

Instructions

Thumb provides about 45 instructions (of about 115 in ARMv6). The narrower codification of 16 bit means that we will be more limited in what we can do in our code. Registers are split into two sets: low registers, r0 to r7, and high registers, r7 to r15. Most instructions can only fully work with low registers and some others have limited behaviour when working with high registers.

Also, Thumb instructions cannot be predicated. Recall that almost every ARM instruction can be made conditional depending on the flags in the cpsr register. This is not the case in Thumb where only the branch instruction is conditional.
Mixing ARM and Thumb is only possible at function level: a function must be wholly ARM or Thumb, it cannot be a mix of the two instruction sets. Recall that our Raspbian system does not support Thumb so at some point we will have to jump from ARM code to Thumb code. This is done using the instruction (available in both instruction sets) blx. This instruction behaves like the bl instruction we use for function calls but changes the state of the processor from ARM to Thumb (or Thumb to ARM).
We also have to tell the assembler that some portion of assembler is actually Thumb while the other is ARM. Since by default the assembler expects ARM, we will have to change to Thumb at some point.

From ARM to Thumb

Let's start with a very simple program returning an error code of 2 set in Thumb.

```
1  /* thumb-first.s */
2  .text
3
4  .code 16     /* Here we say we will use Thumb */
5  .align 2     /* Make sure instructions are aligned at 2-byte boundary */
6
7  thumb_function:
8      mov r0, #2   /* r0 ← 2 */
9      bx lr        /* return */
1
0  .code 32     /* Here we say we will use ARM */
1  .align 4     /* Make sure instructions are aligned at 4-byte boundary */
1
1  .globl main
2  main:
1      push {r4, lr}
3
1      blx thumb_function /* From ARM to Thumb we use blx */
4
```

```
    pop {r4, lr}
    bx lr
```

Thumb instructions in our thumb_function actually resemble ARM instructions. In fact most of the time there will not be much difference. As stated above, Thumb instructions are more limited in features than their ARM counterparts.

If we run the program, it does what we expect.

```
$ ./thumb-first; echo $?
2
```

How can we tell our program actually mixes ARM and Thumb? We can use objdump -d to dump the instructions of our thumb-first.o file.

```
$ objdump  -d thumb-first.o

thumb-first.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <thumb_function>:
   0:   2002            movs    r0, #2
   2:   4770            bx      lr
   4:   e1a00000        nop                     ; (mov r0, r0)
   8:   e1a00000        nop                     ; (mov r0, r0)
   c:   e1a00000        nop                     ; (mov r0, r0)
```

```
00000010 <main>:
  10:   e92d4010        push    {r4, lr}
  14:   fafffff9        blx     0 <thumb_function>
  18:   e8bd4010        pop     {r4, lr}
  1c:   e12fff1e        bx      lr
```

Check thumb_function: its two instructions are encoded in just two bytes (instruction bx lris at offset 2 of mov r0, #2. Compare this to the instructions in main: each one is at offset 4 of its predecessor instruction. Note that some padding was added by the assembler at the end of the thumb_function in form of nops (that should not be executed, anyway).

Calling functions in Thumb

In in Thumb we want to follow the AAPCS convention like we do when in ARM mode, but then some oddities happen. Consider the following snippet where thumb_function_1 calls thumb_function_2.

```
.code 16     /* Here we say we will use Thumb */
.align 2     /* Make sure instructions are aligned at 2-byte boundary */
thumb_function_2:
    /* Do something here */
    bx lr

thumb_function_1:
    push {r4, lr}
    bl thumb_function_2
    pop {r4, lr}    /* ERROR: cannot use lr in pop  in Thumb mode */
    bx lr
```

Unfortunately, this will be rejected by the assembler. If you recall from chapter 10, in ARM push and pop are mnemonics for stmdb sp! and ldmia sp!, respectively. But in Thumb mode push and pop are instructions on their own and so they are more limited: push can only use low registers and lr, pop can only use low registers and pc. The behaviour of these two instructions almost the same as the ARM mnemomics. So, you are now probably wondering why these two special cases for lr and pc. This is the trick: in Thumb mode pop {pc} is equivalent to pop the value val from the stack and then do bx val. So the two instruction sequence: pop {r4, lr} followed by bx lr becomes simply pop {r4, pc}.
So, our code will look like this.

```
/* thumb-call.s */
.text

.code 16     /* Here we say we will use Thumb */
.align 2     /* Make sure instructions are aligned at 2-byte boundary */

thumb_function_2:
    mov r0, #2
    bx lr   /* A leaf Thumb function (i.e. a function that does not call
               any other function so it did not have to keep lr in the stack)
               returns using "bx lr" */

thumb_function_1:
    push {r4, lr}
    bl thumb_function_2 /* From Thumb to Thumb we use bl */
    pop {r4, pc}  /* This is how we return from a non-leaf Thumb function */

.code 32     /* Here we say we will use ARM */
.align 4     /* Make sure instructions are aligned at 4-byte boundary */
.globl main
main:
    push {r4, lr}

    blx thumb_function_1 /* From ARM to Thumb we use blx */

    pop {r4, lr}
    bx lr
```

From Thumb to ARM

Finally we may want to call an ARM function from Thumb. As long as we stick to AAPCS everything should work correctly. The Thumb instruction to call an ARM function is again blx. Following is an example of a small program that says “Hello world” four times calling printf, a function in the C library that in Raspbian is of course implemented using ARM instructions.

```
/* thumb-first.s */
```

```
.text

.data
message: .asciz "Hello world %d\n"

.code 16     /* Here we say we will use Thumb */
.align 2     /* Make sure instructions are aligned at 2-byte boundary */
thumb_function:
    push {r4, lr}         /* keep r4 and lr in the stack */
    mov r4, #0            /* r4 ← 0 */
    b check_loop          /* unconditional branch to check_loop */
    loop:
       /* prepare the call to printf */
       ldr r0, addr_of_message  /* r0 ← &message */
       mov r1, r4               /* r1 ← r4 */
       blx printf               /* From Thumb to ARM we use blx.
                                   printf is a function
                                   in the C library that is implemented
                                   using ARM instructions */
       add r4, r4, #1           /* r4 ← r4 + 1 */
    check_loop:
       cmp r4, #4               /* compute r4 - 4 and update the cpsr */
       blt loop                 /* if the cpsr means that r4 is lower than 4
                                   then branch to loop */

    pop {r4, pc}          /* restore registers and return from Thumb function */
.align 4
addr_of_message: .word message

.code 32     /* Here we say we will use ARM */
.align 4     /* Make sure instructions are aligned at 4-byte boundary */
.globl main
main:
    push {r4, lr}      /* keep r4 and lr in the stack */
    blx thumb_function /* from ARM to Thumb we use blx  */
    pop {r4, lr}       /* restore registers */
    bx lr              /* return */
```

To know more

In next installments we will go back to ARM, so if you are interested in Thumb, you may want to check this Thumb 16-bit Instruction Set Quick Reference Card provided by ARM. When checking that card, be aware that the processor of the Raspberry Pi only implements ARMv6T, not ARMv6T2.

That’s all for today.

ARM assembler in Raspberry Pi – Chapter 23

January 2, 2015 Roger Ferrer Ibáñez, 4

Today we will see what happens when we nest a function inside another. It seems a harmless thing to do but it happens to come with its own dose of interesting details.

Nested functions

At the assembler level, functions cannot be nested.

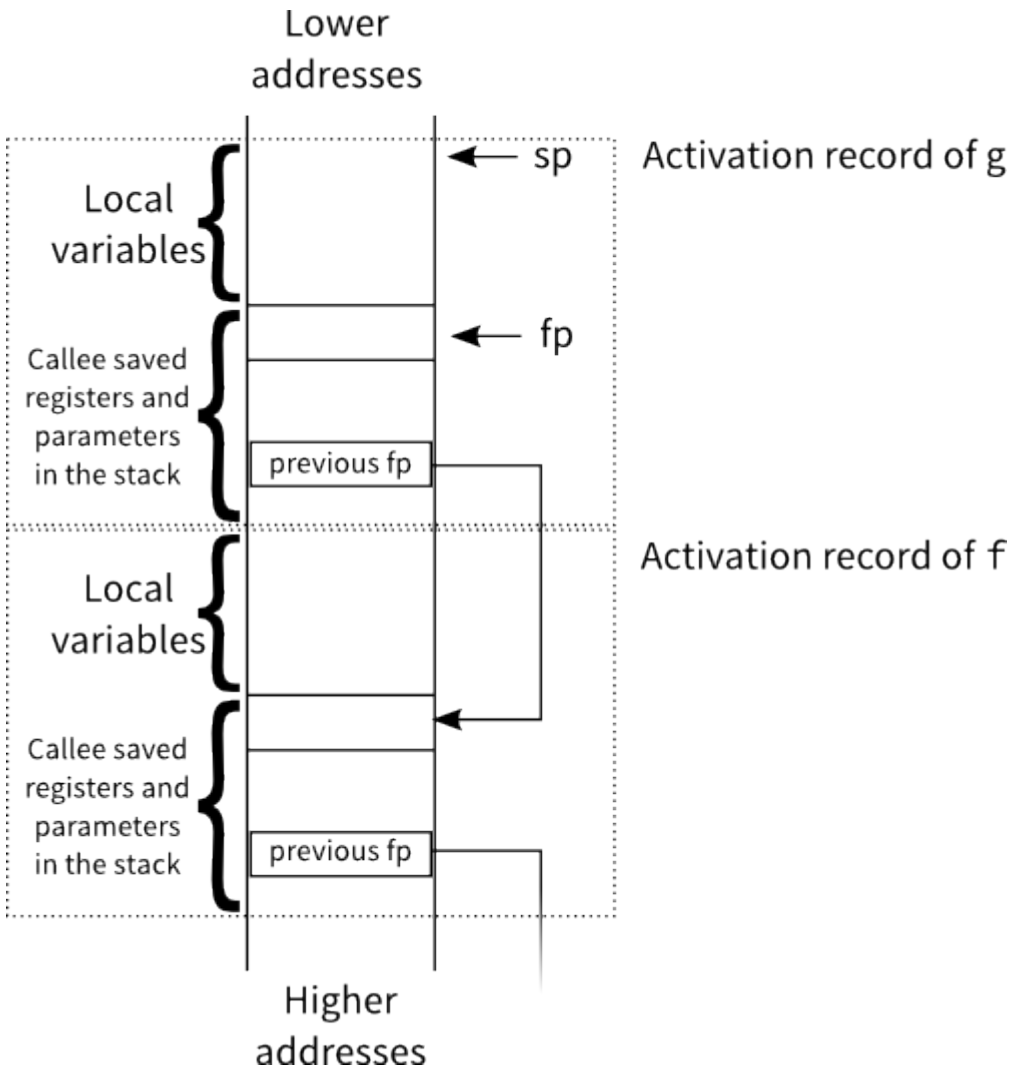
In fact functions do not even exist at the assembler level. They logically exist because we follow some conventions (in ARM Linux it is the AAPCS) and we call them functions. At the assembler level everything is either data, instructions or addresses. Anything else is built on top of those. This fact, though, has not prevented us from enjoying functions: we have called functions like printf and scanf to print and read strings and in chapter 20 we even called functions indirectly. So functions are a very useful logical convention.
So it may make sense to nest a function inside another. What does mean to nest a function inside another? Well, it means that this function will only have meaning as long as its enclosing function is dynamically active (i.e. has been called).

At the assembler level a nested function will look like much any other function but they have enough differences to be interesting.

Dynamic link

In chapter 18 we talked about the dynamic link. The dynamic link is set at the beginning of the function and we use the fp register (an alias for r11) to keep an address in the stack usually called the frame pointer (hence the fp name). It is dynamic because it related to the dynamic activation of the function. The frame pointer gives us a consistent way of accessing local data of the function (that will always be stored in the stack) and those parameters that have to be passed using the stack.

Recall that local data, due to the stack growing downwards, is found in negative offsets from the address in the fp. Conversely, parameters passed using the stack will be in positive offsets. Note that fp (aka r11) is a callee-saved register as specified by the AAPCS. This means that we will have to push it onto the stack upon the entry to the function. A non obvious fact from this last step is that the previous frame pointer is always accessible from the current one. In fact it is found among the other callee-saved registers in a positive offset from fp (but a lower offset than the parameters passed using the stack because callee-saved registers are pushed last). This last property may seem non interesting but allows us to chain up through the frame pointer of our callers. In general, this is only of interest for debuggers because they need to keep track of functions being called so far.
The following image shows how the stack layout, after the dynamic link has been set and the stack has been enlarged for local variables, looks like for a function g that has been called by f. The set of data that is addressed using the frame pointer is commonly called the activation record, since it is a bunch of information that is specific of the dynamic activation of the function (i.e. of the current call).



Static link

When a function calls a nested function (also called a local function), the nested function can use local variables of the enclosing function. This means that there must be a way for the nested function to access local variables from the

enclosing function. One might think that the dynamic link should be enough. In fact, if the programming language only allowed nested functions call other (immediately) nested functions, this would be true. But if this were so, that programming language would be rather limited. That said, for the moment, let's assume that this is the case: check again the image above. If g is a local function of f, then it should be possible for g to access local variables of f by getting to the previous fp.
Consider the following C code (note that Standard C does not allow nesting functions though GCC implements them as an extension that we will discuss in a later chapter).

```
1
2
3  void f() // non-nested (normal) function
4  {
5    int x;
6    void g() // nested function
7    {
8      x = x + 1; // x ← x + 1
9    }
1
0    x = 1;      // x ← 1
1    g();        // call g
1    x = x + 1; // x ← x + 1
1    // here x will be 3
2  }
1
3
```

The code above features this simple case where a function can call a nested one. At the end of the function f, x will have the value 2 because the nested function g modifies the variable x, also modified by f itself.
To access to x from g we need to get the previous fp. Since only f can call us, once we get this previous fp, it will be like the fp we had inside f. So it is now a matter of using the same offset as f uses.

```
1  /* nested01.s */
2
3  .text
4
5  f:
6      push {r4, r5, fp, lr} /* keep registers */
```

```
    mov fp, sp /* keep dynamic link */

    sub sp, sp, #8      /* make room for x (4 bytes)
                           plus 4 bytes to keep stack
                           aligned */
    /* x is in address "fp - 4" */

    mov r4, #1          /* r4 ← 0 */
    str r4, [fp, #-4]   /* x ← r4 */

    bl g                /* call (nested function) g
                           (the code of 'g' is given below, after 'f') */

    ldr r4, [fp, #-4]   /* r4 ← x */
    add r4, r4, #1      /* r4 ← r4 + 1 */
    str r4, [fp, #-4]   /* x ← r4 */

    mov sp, fp /* restore dynamic link */
    pop {r4, r5, fp, lr} /* restore registers */
    bx lr /* return */

    /* nested function g */
    g:
        push {r4, r5, fp, lr} /* keep registers */
        mov fp, sp /* keep dynamic link */

        /* At this point our stack looks like this

          Data | Address | Notes
         ------+---------+---------------------------
           r4  | fp      |
           r5  | fp + 4  |
           fp  | fp + 8  | This is the previous fp
           lr  | fp + 16 |
        */

        ldr r4, [fp, #+8] /* get the frame pointer
```

```
                          of my caller
                          (since only f can call me)
                        */

        /* now r4 acts like the fp we had inside 'f' */
        ldr r5, [r4, #-4] /* r5 ← x */
        add r5, r5, #1    /* r5 ← r5 + 1 */
        str r5, [r4, #-4] /* x ← r5 */

        mov sp, fp /* restore dynamic link */
        pop {r4, r5, fp, lr} /* restore registers */
        bx lr /* return */

.globl main

main :
    push {r4, lr} /* keep registers */

    bl f          /* call f */

    mov r0, #0
    pop {r4, lr}
    bx lr
```

Ok, the essential idea is set. When accessing a local variable, we always need to get the frame pointer of the function where the local variable belongs. In line 43 we get the frame pointer of our caller and then we use it to access the

variable x, lines 49 to 51. Of course, if the local variable belongs to the current function, nothing special has to be done since fp suffices, see lines 20 to 22.
That said, though the idea is fundamentally correct, using the dynamic link limits us a lot: only a single call from an enclosing function is possible. What if we allow nested functions to call other nested functions (sibling functions) or worse, what would have happened if g above called itself recursively? The dynamic link we will find in the stack will always refer to the previous dynamically activated function, and in the example above it was f, but if g recursively calls itself, g will be the previous dynamically activated function!
It is clear that something is amiss. Using the dynamic link is not right because, when accessing a local variable of an enclosing function, we need to get the last activation of that enclosing function at the point where the nested function was called. The way to keep the last activation of the enclosing function is called static link in contrast to the dynamic link.
The static link is conceptually simple, it is also a chain of frame pointers like the dynamic link. In contrast to the dynamic link, which is always set the same way by the callee), the static link may be set differently depending on which function is being called and it will be set by the caller. Below we will see the exact rules.

Consider the following more contrived example;

```
void f(void) // non nested (nesting depth = 0)
{
   int x;

   void g() // nested (nesting depth = 1)
   {
      x = x + 1; // x ← x + 1
   }
   void h() // nested (nesting depth = 1)
   {
      void m() // nested (nesting depth = 2)
      {
         x = x + 2; // x ← x + 2
         g(); // call g
      }
```

```
      g(); // call g
      m(); // call m
      x = x + 3; // x ← x + 3
   }

   x = 1; // x ← 1
   h();  // call h
   // here x will be 8
}
```

A function can, obviously, call an immediately nested function. So from the body of function f we can call g or h. Similarly from the body of function h we can call m. A function can be called by other (non-immediately nested) functions as long as the nesting depth of the caller is greater or equal than the callee. So from m we can call m (recursively), h, g and f. It would not be allowed that f or g called m.

Note that h and g are both enclosed by f. So when they are called, their dynamic link will be of course the caller but their static link must always point to the frame of f. On the other hand, m is enclosed by h, so its static link will point to the frame of h (and in the example, its dynamic link too because it is the only nested function inside h and it does not call itself recursively either). When m calls g, the static link must be again the frame of its enclosing function f.

Setting up a static link

Like it happens with the dynamic link, the AAPCS does not mandate any register to be used as the static link. In fact, any callee-saved register that does not have any specific purpose will do. We will use r10.

Setting up the static link is a bit more involved because it requires paying attention which function we are calling. There are two cases:

- The function is immediately nested (like when from f we call g or h, or when from h we call m). The static link is simply the frame pointer of the caller.

  For theses cases, thus, the following is all we have to do prior the call.

  ```
  mov r10, fp
  bl immediately-nested-function
  ```

- The function is not immediately nested (like when from m we call g) then the static frame must be that of the enclosing function of the callee. Since the static link forms a chain it is just a matter of advancing in the chain as many times as the difference of nesting depths.

For instance, when m calls g, the static link of m is the frame of h. At the same time the static link of h is the frame of f. Since g and h are siblings, their static link must be the same. So when m calls g, the static link should be the same of h.

For theses cases, we will have to do the following

```
ldr r10, [fp, #X0] /* Xi will be the appropiate offset
                      where the previous value of r10 is found
                      Note that Xi depends on the layout of
                      our stack after we have push-ed the
                      caller-saved registers
                    */
ldr r10, [r10, #X1]  \
ldr r10, [r10, #X2]   |
...                   | advance the static link as many times
...                   | the difference of the nesting depth
...                   | (it may be zero times when calling a sibling)
ldr r10, [r10, #Xn]  /
bl non-immediately-nested-function
```

This may seem very complicated but it is not. Since in the example above there are a few functions, we will do one function at a time. Let's start with f.

```
f:
    push {r4, r10, fp, lr} /* keep registers */
    mov fp, sp             /* setup dynamic link */

    sub sp, sp, #8      /* make room for x (4 + 4 bytes) */
    /* x will be in address "fp - 4" */

    /* At this point our stack looks like this

     Data | Address | Notes
    ------+---------+---------------------------
          | fp - 8  | alignment (per AAPCS)
      x   | fp - 4  |
      r4  | fp      |
      r10 | fp + 8  | previous value of r10
      fp  | fp + 12 | previous value of fp
```

```
3
9
4
0       lr  | fp + 16 |
4     */
1
4     mov r4, #1          /* r4 ← 1 */
2     str r4, [fp, #-4]   /* x ← r4 */
4
3     /* prepare the call to h */
4     mov r10, fp /* setup the static link,
4                    since we are calling an immediately nested function
4                    it is just the current frame */
5     bl h        /* call h */
4
6     mov sp, fp              /* restore stack */
4     pop {r4, r10, fp, lr}  /* restore registers */
7     bx lr /* return */
4
8
1
```

Since f is not nested in any other function, the previous value of r10 does not have any special meaning for us. We just keep it because r10, despite the special meaning we will give it, is still a callee-saved register as mandated by the AAPCS. At the beginning, we allocate space for the variable x by enlarging the stack (line 35). Variable x will be always in fp - 4. Then we set x to 1 (line 51). Nothing fancy here since this is a non-nested function.

Now f calls h (line 57). Since it is an immediately nested function, the static link is as in the case I: the current frame pointer. So we just set r10 to be fp (line 56).

Let’s see the code of h now.

```
6   /* ------ nested function ------------------ */
3   h :
6       push {r4, r5, r10, fp, lr} /* keep registers */
4       mov fp, sp /* setup dynamic link */
6
5       sub sp, sp, #4 /* align stack */
6
```

```
6
6
7     /* At this point our stack looks like this
6
8       Data | Address | Notes
6      ------+---------+---------------------------
9           | fp - 4  | alignment (per AAPCS)
7       r4  | fp      |
0       r5  | fp + 4  |
7       r10 | fp + 8  | frame pointer of 'f'
1       fp  | fp + 12 | frame pointer of caller
7       lr  | fp + 16 |
2     */
7
3     /* prepare call to g */
7     /* g is a sibling so the static link will be the same
4        as the current one */
7     ldr r10, [fp, #8]
5     bl g
7
6     /* prepare call to m */
7     /* m is an immediately nested function so the static
7        link is the current frame */
7     mov r10, fp
8     bl m
7
9     ldr r4, [fp, #8]  /* load frame pointer of 'f' */
8     ldr r5, [r4, #-4]  /* r5 ← x */
0     add r5, r5, #3     /* r5 ← r5 + 3 */
8     str r5, [r4, #-4]  /* x ← r5 */
1
8     mov sp, fp            /* restore stack */
2     pop {r4, r5, r10, fp, lr} /* restore registers */
8     bx lr
3
0
1
```

We start the function as usual, pushing registers onto the stack and setting up the dynamic link (lines 64 to 65). We adjust the stack so the stack pointer is 8-byte aligned because we have pushed an even number of registers (line 68). If you check the layout of the stack after this last adjustment (depicted in lines 72 to 79), you will see that in fp + 8 we have the value of r10 which the caller of h (in this example only f, but it could be another function) must ensure that is the frame pointer of f. This extra pointer in the stack is the static link.
Now the function calls g (line 86) but it must properly set the static link prior to the call. In this case the static link is the same as h because we call g which is a sibling of h, so they share the same static link. We get it from fp + 8 (line 85). This is in fact the case II described above: g is not an immediately nested function of h. So we have to get the static link of the caller (the static link of h, found in fp + 8) and then advance it as many times as the difference of their nesting depths. Being siblings means that their nesting depths are the same, so no advancement is actually required.
After the call to g, the function calls m (line 92) which happens to be an immediately nested function, so its static link is the current frame pointer (line 91) because this is again the case I.
Let's see now the code of m.

```
/* ------ nested function ------------------ */
m:
    push {r4, r5, r10, fp, lr} /* keep registers */
    mov fp, sp /* setup dynamic link */

    sub sp, sp, #4 /* align stack */
    /* At this point our stack looks like this

      Data | Address | Notes
     ------+---------+---------------------------
           | fp - 4  | alignment (per AAPCS)
       r4  | fp      |
       r5  | fp + 4  |
       r10 | fp + 8  | frame pointer of 'h'
       fp  | fp + 12 | frame pointer of caller
       lr  | fp + 16 |
    */

    ldr r4, [fp, #8]  /* r4 ← frame pointer of 'h' */
    ldr r4, [r4, #8]  /* r4 ← frame pointer of 'f' */
```

```
0
1
1
1
1
1
2
1
1
3
1
1
4
1
1
5
1
1
6
1
1
5
```

```
    ldr r5, [r4, #-4] /* r5 ← x */
    add r5, r5, #2    /* r5 ← r5 + 2 */
    str r5, [r4, #-4] /* x ← r5 */

    /* setup call to g */
    ldr r10, [fp, #8]   /* r10 ← frame pointer of 'h' */
    ldr r10, [r10, #8]  /* r10 ← frame pointer of 'f' */
    bl g

    mov sp, fp                /* restore stack */
    pop {r4, r5, r10, fp, lr} /* restore registers */
    bx lr
```

Function m starts pretty similar to h: we push the registers, setup the dynamic link and adjust the stack so it is 8-byte aligned (lines 106 to 109). After this, we again have the static link at fp + 8. If you are wondering if the static link will always be in fp + 8, the answer is no, it depends on how many registers are pushed before r10, it just happens that we always push r4 and r5, but if we, for instance, also pushed r6 it would be at a larger offset. Each function may have the static link at different offsets (this is why we are drawing the stack layout for every function, bear this in mind!).
The first thing m does is x ← x + 2. So we have to get the address of x. The address of x is relative to the frame pointer of f because x is a local variable of f. We do not have the frame pointer of f but the one of h (this is the static link of m). Since the frame pointers form a chain, we can load the frame pointer of h and then use it to get the static link of h which will be the frame pointer of f. You may have to reread this last statement twice 🙂 So we first get the frame pointer of h (line 122), recall that this is the static link of m that was set up when h called m (line 91). Now we

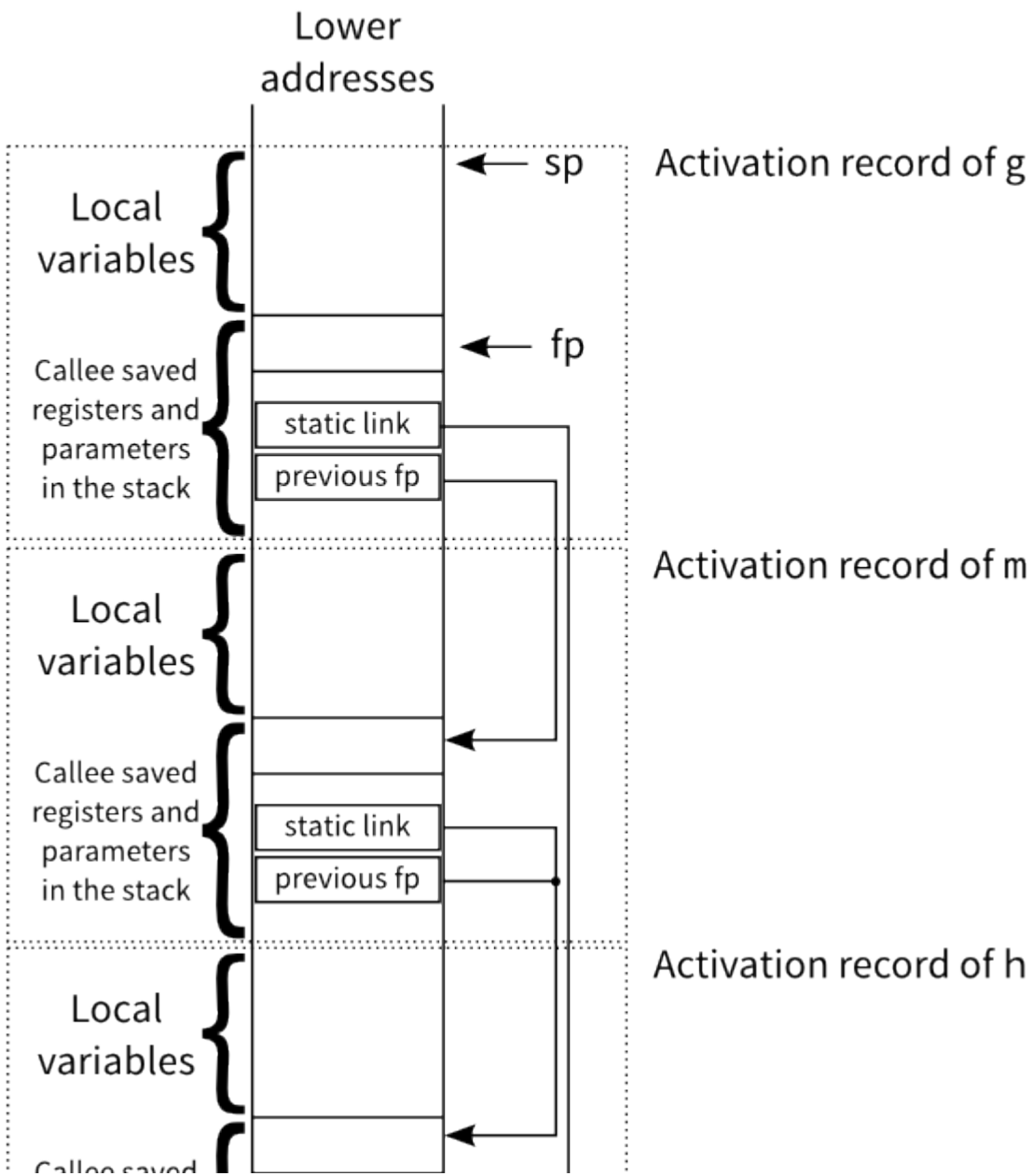
have the frame pointer of h, so we can get its static link (line 123) which again is at offset +8but this is by chance, it could be in a different offset! The static link of h is the frame pointer of f, so we now have the frame pointer f as we wanted and then we can proceed to get the address of x, which is at offset -4 of the frame pointer of f. With this address now we can perform x ← x + 2 (lines 124 to 126).
Then m calls g (line 131). This is again a case II. But this time g is not a sibling of m: their nesting depths differ by 1. So we first load the current static link (line 129), the frame pointer of h. And then we advance 1 link through the chain of static links (line 130). Let me insist again: it is by chance that the static link of h and f is found at fp+8, each function could have it at different offsets.
Let's see now the code of g, which is pretty similar to that of h except that it does not call anyone.

```
1   /* ------ nested function ------------------ */
2   g:
7       push {r4, r5, r10, fp, lr} /* keep registers */
1       mov fp, sp /* setup dynamic link */
2
8       sub sp, sp, #4 /* align stack */
1
2       /* At this point our stack looks like this
9
1         Data | Address | Notes
3        ------+---------+---------------------------
0             | fp - 4  | alignment (per AAPCS)
1          r4 | fp      |
3          r5 | fp + 4  |
1          r10 | fp + 8  | frame pointer of 'f'
1          fp | fp + 12 | frame pointer of caller
3          lr | fp + 16 |
2       */
1
3       ldr r4, [fp, #8]  /* r4 ← frame pointer of 'f' */
3       ldr r5, [r4, #-4] /* r5 ← x */
1       add r5, r5, #1    /* r5 ← r5 + 1 */
3       str r5, [r4, #-4] /* x ← r5 */
4
1       mov sp, fp /* restore dynamic link */
```

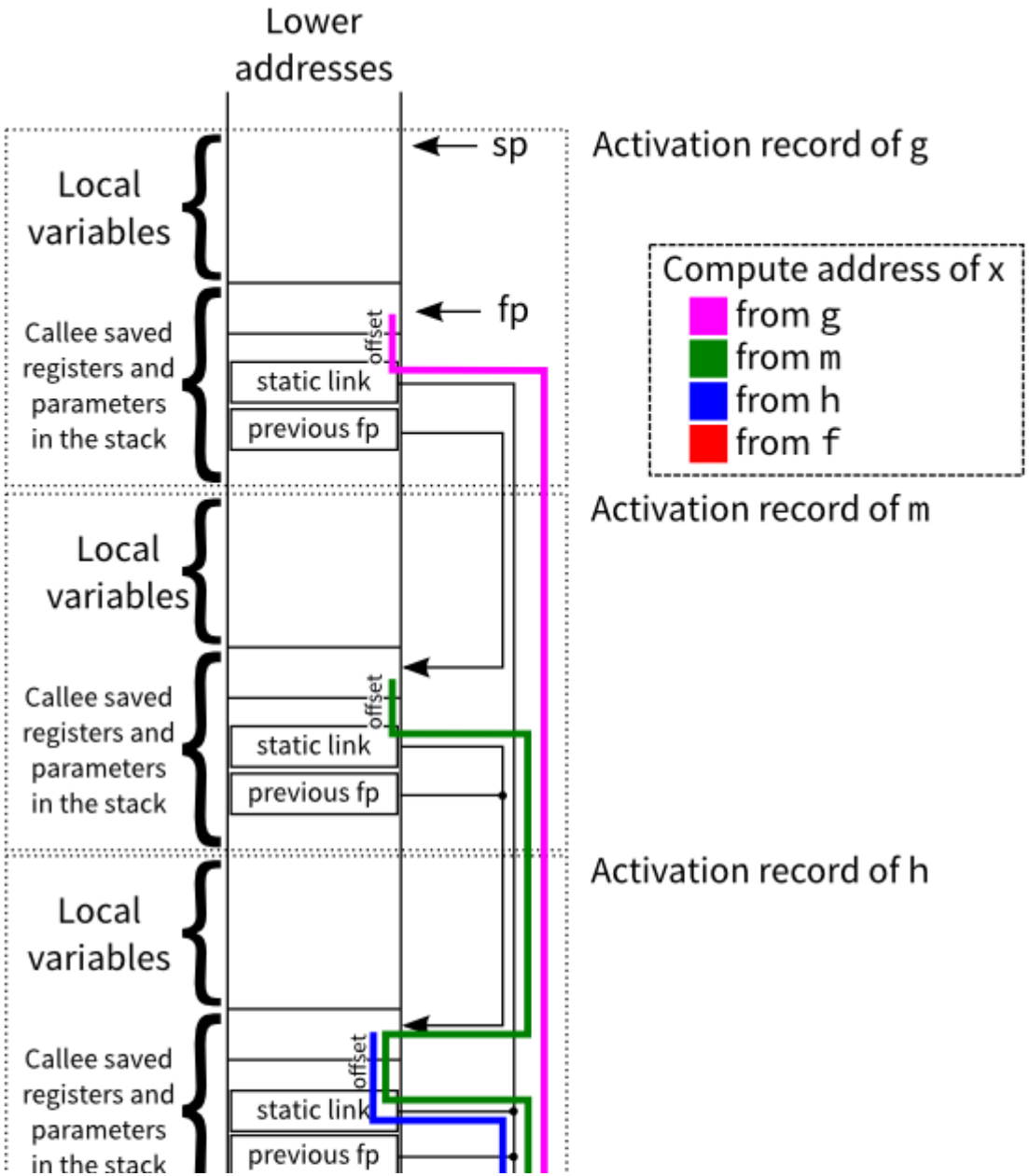
```
3
5
1
3
6
1
3
7
3
```

```
    pop {r4, r5, r10, fp, lr} /* restore registers */
    bx lr
```

Note that h and g compute the address of x exactly the same way since they are at the same nesting depth.
Below is a picture of how the layout looks once m has called g. Note that the static link of g and his the same, the frame pointer of f, because they are siblings.


Lower
addresses
sp
Activation record of g
Local
variables
fp
Callee saved
registers and
parameters
in the stack
static link
previous fp
Activation record of m
Local
variables
Callee saved
registers and
parameters
in the stack
static link
previous fp
Activation record of h
Local
variables

Below is the same image but this time using coloured lines to show how each function can compute the address of x.


Lower addresses
sp
Activation record of g
Local variables
fp
Callee saved registers and parameters in the stack
offset
static link
previous fp
Compute address of x
from g
from m
from h
from f
Activation record of m
Local variables
Callee saved registers and parameters in the stack
offset
static link
previous fp
Activation record of h
Local variables
Callee saved registers and parameters in the stack
offset
static link
previous fp

Finally here is the main. Note that when a non-nested function calls another non-nested function, there is no need to do anything to r10. This is the reason why r10 does not have any meaningful value upon the entry to f.

```
1
5
2
1
5
3
1
5
4
1
5
5
1
5
7
1
5
8
1
5
9
1
6
0
1
6
1
```

```
.globl main

main :
    push {r4, lr} /* keep registers */

    bl f          /* call f */

    mov r0, #0
    pop {r4, lr}
    bx lr
```

Discussion

If you stop and think about all this stuff of the static link you may soon realize that there is something murky with all this nested functions business: we are passing some sort of hidden parameter (through r10) to the nested functions. In fact,

we are somehow cheating, because we set r10 right before the call and then we push it at the entry of the nested functions even if they do not modify it in the called function. Why are we doing this seemingly useless step?
Well, by always pushing r10 in the stack, we are just covering up the naked truth: nested functions require a, somewhat hidden, extra parameter. This extra parameter is this static link thing. Sometimes it is also called the lexical scope. It is called the lexical scope because it gives us the context of the lexically (i.e. in the code) enclosing function (in contrast the dynamic scope would be that of our caller, which we do not care about unless we are a debugger). With that lexical context we can get the local variables of that enclosing function. Due to the chaining nature of the static link, we can move up the lexical scopes. This is the reason m can access a variable of f, it just climbs through the static links as shown in the last picture above.
Can we pass the lexical scope to a function using the stack, rather than a callee-saved register? Sure. For convenience it may have to be the first stack-passed parameter (so its offset from fp is easy to compute). Instead of setting r10 prior the call, we will enlarge sp as needed (at least 8 bytes, to keep the stack 8-byte aligned) and then store there the static link. In the stack layout, the static link now will be found after (i.e. larger offsets than) the pushed registers.
Can we pass the lexical scope using a caller-saved register (like r0, r1, r2 or r3)? Yes, but the first thing we should do is to keep it in the stack, as a local variable (i.e. negative offsets from fp). Why? Because if we do not keep it in the stack we will not be able to move upwards the static links.
As you can see, any approach requires us to keep the static link in the stack. While our approach of using r10 may not be completely orthodox ends doing the right thing.
But the discussion would not be complete if we did not talk about pointers. What about a pointer to a nested function? Is that even possible? When (directly) calling a nested function we can set the lexical scope appropiately because we know everything: we know where we are and we know which function we are going to call. But what about an indirect call using a pointer to a function? We do not know which (possibly nested) function we are going to call, how can we appropriately set its lexical scope. Well, the answer is, we cannot unless we keep the lexical scope somewhere. This means that just the address of the function will not do. We will need to keep, along with the address to the function, the lexical scope. So a pointer to a nested function happens to be different to a pointer to a non-nested function, given that the latter does not need the lexical scope to be set.

Having incompatible pointers for nested an non nested functions is not desirable. This may be a reason why C (and C++) do not directly support nested functions (albeit this limitation can be worked around using other approaches). In the

next chapter, we will see a clever approach to avoid, to some extent, having different pointers to nested functions that are different from pointers to non nested functions.

That’s all for today.

ARM assembler in Raspberry Pi – Chapter 24

January 9, 2015 Roger Ferrer Ibáñez, 7
Today we will continue with nested functions.

Sorting

We will first take a detour. The C function qsort can be used to sort any kind of array. Its C signature is the following.

```
void qsort(void *base,
     size_t nmemb,
     size_t size,
     int (*compar)(const void *, const void *));
```

qsort returns void, this is, it does not return anything because it performs the sort in place. This means that we will pass a (potentially unsorted) array called base of length nmemb to qsort. When qsort returns, the elements in this array will be sorted. If qsort were able to just sort a specific kind of arrays it would be rather limited. In order to be able to sort any array, qsort requires the size of each element in the array. Note that the array is passed by reference (otherwise the in place sorting would not be possible): void* is the C way to say «I accept an address to any kind of data».
We will come back later to the compar bit of qsort.

Print an array

Before we sort an array, we need a way to examine it. We will use for that a function print_array that prints an array of integers.

```
1  /* print-array.s */
2
3  .data
4
5  /* declare an array of 10 integers called my_array */
6  .align 4
7  my_array: .word 82, 70, 93, 77, 91, 30, 42, 6, 92, 64
8
9  /* format strings for printf */
1  /* format string that prints an integer plus a space */
0  .align 4
```

```
integer_printf: .asciz "%d "
/* format string that simply prints a newline */
.align 4
newline_printf: .asciz "\n"

.text

print_array:
    /* r0 will be the address of the integer array */
    /* r1 will be the number of items in the array */
    push {r4, r5, r6, lr}  /* keep r4, r5, r6 and lr in the stack */

    mov r4, r0             /* r4 ← r0. keep the address of the array */
    mov r5, r1             /* r5 ← r1. keep the number of items */
    mov r6, #0             /* r6 ← 0.  current item to print */

    b .Lprint_array_check_loop /* go to the condition check of the loop */

    .Lprint_array_loop:
      /* prepare the call to printf */
      ldr r0, addr_of_integer_printf  /* r0 ← &integer_printf */
      /* load the item r6 in the array in address r4.
         elements are of size 4 bytes so we need to multiply r6 by 4 */
      ldr r1, [r4, +r6, LSL #2]       /* r1 ← *(r4 + r6 << 2)
                                         this is the same as
                                         r1 ← *(r4 + r6 * 4) */
      bl printf                       /* call printf */

      add r6, r6, #1                  /* r6 ← r6 + 1 */
    .Lprint_array_check_loop:
      cmp r6, r5                /* perform r6 - r5 and update cpsr */
      bne .Lprint_array_loop    /* if cpsr states that r6 is not equal to r5
                                   branch to the body of the loop */

    /* prepare call to printf */
    ldr r0, addr_of_newline_printf /* r0 ← &newline_printf */
    bl printf
```

```
    pop {r4, r5, r6, lr}   /* restore r4, r5, r6 and lr from the stack */
    bx lr                  /* return */

addr_of_integer_printf: .word integer_printf
addr_of_newline_printf: .word newline_printf

.globl main
main:
    push {r4, lr}              /* keep r4 and lr in the stack */

    /* prepare call to print_array */
    ldr r0, addr_of_my_array  /* r0 ← &my_array */
    mov r1, #10               /* r1 ← 10
                                 our array is of length 10 */
    bl print_array            /* call print_array */

    mov r0, #0                /* r0 ← 0 set errorcode to 0 prior returning from main */
    pop {r4, lr}              /* restore r4 and lr in the stack */
    bx lr                     /* return */

addr_of_my_array: .word my_array
```

The code above is pretty straightforward and it does not feature anything that has not been seen in previous installments. Running it simply prints the current contents of the array.

```
$ ./print-array
82 70 93 77 91 30 42 6 92 64
```

Comparison

Above, when we talked about qsort we skipped the compar parameter. What is compar? It is a an address to a function. The funky syntax for C tells us that this function, if it is ever called, will be passed two addreses (again, it does not care what they are, so they are void*) and returns an integer. The manual of qsort explains that this function has to return lower than zero, zero or greater than zero. If the object in the address of the first parameter of compar is lower than the object in the address of the second parameter, then it has to return lower than zero. If they are equal, it should return zero. If the first object is greater than the second, then it should return greater than zero.

If you wonder why the parameters of compar are actually const void* rather than void*, it is the C way of telling us that the data of the referenced objects cannot change during the comparison. This may sound obvious given that changing things is not the job of a comparison function. Passing them by reference would let us to change them. So this is reminder that we should not.

Since our array is an array of integers we will have to compare integers: let's write a function that, given two pointers to integers (i.e. addresses) behaves as stated above.

```
1
9
2
0
2
1
2
2
2
3
2
4
2
5
2
6
2
7
2
8
```

```
integer_comparison:
    /* r0 will be the address to the first integer */
    /* r1 will be the address to the second integer */
    ldr r0, [r0]    /* r0 ← *r0
                       load the integer pointed by r0 in r0 */
    ldr r1, [r1]    /* r1 ← *r1
                       load the integer pointed by r1 in r1 */

    cmp r0, r1      /* compute r0 - r1 and update cpsr */
    moveq r0, #0    /* if cpsr means that r0 == r1 then r0 ←  0 */
    movlt r0, #-1   /* if cpsr means that r0 <  r1 then r0 ← -1 */
    movgt r0, #1    /* if cpsr means that r0 >  r1 then r0 ←  1 */
    bx lr           /* return */
```

```
2
9
3
0
3
1
```

Function integer_comparison does not feature anything new either: it simply avoids branches by using predication as we saw in chapter 11.

Now we have the last missing bit to be able to call qsort. Here is a program that prints (only main is shown) the array twice, before sorting it and after sorting it.

```
6  .globl main
6  main:
6      push {r4, lr}              /* keep r4 and lr in the stack */
7
6      /* prepare call to print_array */
8      ldr r0, addr_of_my_array   /* r0 ← &my_array */
6      mov r1, #10                /* r1 ← 10
9                                   our array is of length 10 */
7      bl print_array             /* call print_array */
0
7      /* prepare call to qsort */
1      /*
7      void qsort(void *base,
2           size_t nmemb,
7           size_t size,
3           int (*compar)(const void *, const void *));
7      */
4      ldr r0, addr_of_my_array   /* r0 ← &my_array
7                                   base */
5      mov r1, #10                /* r1 ← 10
7                                   nmemb = number of members
6                                   our array is 10 elements long */
7      mov r2, #4                 /* r2 ← 4
7                                   size of each member is 4 bytes */
8      ldr r3, addr_of_integer_comparison
```

```
                                /* r3 ← &integer_comparison
                                   compar */
    bl qsort                     /* call qsort */

    /* now print again to see if elements were sorted */
    /* prepare call to print_array */
    ldr r0, addr_of_my_array  /* r0 ← &my_array */
    mov r1, #10                  /* r1 ← 10
                                    our array is of length 10 */
    bl print_array               /* call print_array */

    mov r0, #0                   /* r0 ← 0 set errorcode to 0 prior returning from main */
    pop {r4, lr}                 /* restore r4 and lr in the stack */
    bx lr                        /* return */

addr_of_my_array: .word my_array
addr_of_integer_comparison : .word integer_comparison
```

If we put everything together, we can verify that our array is effectively sorted after the call to the qsort function.

```
$ ./sort-array
82 70 93 77 91 30 42 6 92 64
6 30 42 64 70 77 82 91 92 93
```

What is going on?

C function qsort implements a sorting algorithm (the C Standard does not specify which one must be but it is usually a fine-tuned version of quicksort) which at some point will require to compare two elements. To do this, qsort calls the function compar.

Count how many comparisons happen

Now, we want to count how many comparisons (i.e., how many times integer_comparison is called) when sorting the array. We could change integer_comparison so it increments a global counter.

```
.data
global_counter: .word 0

.text
integer_comparison_count_global:
    /* r0 will be the address to the first integer */
    /* r1 will be the address to the second integer */
    push {r4, r5}   /* keep callee-saved registers */
    ldr r0, [r0]    /* r0 ← *r0
                       load the integer pointed by r0 in r0 */
    ldr r1, [r1]    /* r1 ← *r1
                       load the integer pointed by r1 in r1 */

    cmp r0, r1      /* compute r0 - r1 and update cpsr */
    moveq r0, #0    /* if cpsr means that r0 == r1 then r0 ←  0 */
    movlt r0, #-1   /* if cpsr means that r0 <  r1 then r0 ← -1 */
    movgt r0, #1    /* if cpsr means that r0 >  r1 then r0 ←  1 */

    ldr r4, addr_of_global_counter /* r4 ← &global_counter */
    ldr r5, [r4]    /* r5 ← *r4 */
    add r5, r5, #1  /* r5 ← r5 + 1 */
    str r5, [r4]    /* *r4 ← r5 */

    pop {r4, r5}    /* restore callee-saved registers */
    bx lr           /* return */
addr_of_global_counter: .word global_counter
```

But this post is about nested functions so we will use nested functions. Recall that nested functions can access local variables of their enclosing functions. So we will use a local variable of main as the counter and a nested function (of main) that performs the comparison and updates the counter.

In the last chapter we ended with a short discussion about nested functions. A downside of nested functions it that a pointer to a nested function requires two things: the address of the function and the lexical scope. If you check again

the previous example where we call qsort, you will see that we do not mention anywhere the lexical scope. And there is a reason for that, it is not possible to pass it to qsort. In C, functions cannot be nested so a pointer to a function can just be the address of the function.

Trampoline

We will continue using the convention of the last chapter: r10 will have the lexical scope upon the entry of the function. But qsort, when calls integer_compare_count will not set it for us: we cannout count on r10 having a meaningful value when called from qsort. This means that qsort should actually call something that first sets r10 with the right value and then jumps to integer_compare_count. We will call this ancillary code (or pseudofunction) a trampoline. The technique used here is similar to the one used by GCC described in Lexical Closures for C++ (Thomas M. Breuel, USENIX C++ Conference Proceedings, October 17-21, 1988).

The trampoline is a small, always the same, sequence of instructions that behaves like a function and its only purpose is to set r10 and then make an indirect call to the nested function. Since the sequence of instructions is always the same, the instructions themselves look like a template.

```
1
7
4
1
7
5
1
7
6
1
7
7
1
7
8
1
7
9
1
```

```
.Laddr_trampoline_template : .word .Ltrampoline_template /* we will use this below */
.Ltrampoline_template:
    .Lfunction_called: .word 0x0
    .Llexical_scope: .word 0x0
    push {r4, r5, r10, lr}          /* keep callee-saved registers */
    ldr r4, .Lfunction_called       /* r4 ← function called */
    ldr r10, .Llexical_scope        /* r10 ← lexical scope */
    blx r4                          /* indirect call to r4 */
    pop {r4, r5, r10, lr}           /* restore callee-saved registers */
    bx lr                           /* return */
```

```
80
01
88
11
82
18
3
```

I used the word template because while the instructions are not going to change, there are two items in the trampoline, labeled function_called and lexical_scope, that will have to be appropriately set before using the trampoline.
It may be easier to understand if you consider the code above as if it were data: see it as an array of integers. The first two integers, function_called and lexical_scope, are still zero but will be set at some point. The remaining elements in the array are other integers (we do not care which ones) that happen to encode ARM instructions. The cool thing is that these instructions refer to the two first integers, so by changing them we are indirectly changing what the trampoline does. This trampoline takes 8 words, so 32 bytes.
Let’s start with this example.

```
/* trampoline-sort-arrays.s */

.data

/* declare an array of 10 integers called my_array */
.align 4
my_array: .word 82, 70, 93, 77, 91, 30, 42, 6, 92, 64

/* format strings for printf */
/* format string that prints an integer plus a space */
.align 4
integer_printf: .asciz "%d "
/* format string that simply prints a newline */
.align 4
newline_printf: .asciz "\n"
```

```
.align 4 /* format string for number of comparisons */
comparison_message: .asciz "Num comparisons: %d\n"

.text
```

The function print_array will be the same as above. Next is main.

```
.globl main
main:
    push {r4, r5, r6, fp, lr} /* keep callee saved registers */
    mov fp, sp                /* setup dynamic link */

    sub sp, sp, #4            /* counter will be in fp - 4 */
    /* note that now the stack is 8-byte aligned */

    /* set counter to zero */
    mov r4, #0        /* r4 ← 0 */
    str r4, [fp, #-4] /* counter ← r4 */
```

Nothing fancy here, we set the dynamic link, allocate space in the stack for the counter and set it to zero.

Now we make room for the trampoline in the stack. Recall that our trampoline takes 32 bytes.

```
    /* Make room for the trampoline */
    sub sp, sp, #32 /* sp ← sp - 32 */
    /* note that 32 is a multiple of 8, so the stack
       is still 8-byte aligned */
```

Now we will copy the trampoline template into the stack storage we just allocated. We do this with a loop that copies a word (4 bytes) at a time.

```
    /* copy the trampoline into the stack */
    mov r4, #32                          /* r4 ← 32 */
    ldr r5, .Laddr_trampoline_template /* r4 ← &trampoline_template */
    mov r6, sp                           /* r6 ← sp */
    b .Lcopy_trampoline_loop_check     /* branch to copy_trampoline_loop_check */

    .Lcopy_trampoline_loop:
        ldr r7, [r5]     /* r7 ← *r5 */
        str r7, [r6]     /* *r6 ← r7 */
        add r5, r5, #4   /* r5 ← r5 + 4 */
        add r6, r6, #4   /* r6 ← r6 + 4 */
        sub r4, r4, #4   /* r4 ← r4 - 4 */
    .Lcopy_trampoline_loop_check:
        cmp r4, #0                   /* compute r4 - 0 and update cpsr */
        bgt .Lcopy_trampoline_loop  /* if cpsr means that r4 > 0
                                       then branch to copy_trampoline_loop */
```

```
```

In the loop above, r4 counts how many bytes remain to copy. r5 and r6 are pointers inside the (source) trampoline and the (destination) stack, respectively. Since we copy 4 bytes at a time, all three registers are updated by 4.

Now we have the trampoline copied in the stack. Recall, it is just an array of words, the two first of which must be updated. The first 4 bytes must be the address of function to be called, i.e. integer_comparison_count and the second 4 bytes must be the static link, i.e. fp.

```
    /* setup the trampoline */
    ldr r4, addr_of_integer_comparison_count
                        /* r4 ← &integer_comparison_count */
    str r4, [fp, #-36] /* *(fp - 36) ← r4 */
                        /* set the function_called in the trampoline
                           to be &integer_comparison_count */
    str fp, [fp, #-32]  /* *(fp - 32) ← fp */
                         /* set the lexical_scope in the trampoline
                            to be fp */
```

Recall that our trampoline takes 32 bytes but in the stack we also have the counter. This is the reason why the trampoline starts in fp - 36 (this is also the address of the first word of the trampoline, of course). The second word is then at fp - 32.
Now we proceed like in the sort example above: we print the array before sorting it and after sorting it. Before printing the sorted array, we will also print the number of comparisons that were performed.

```
    /* prepare call to print_array */
    ldr r0, addr_of_my_array /* r0 ← &my_array */
    mov r1, #10              /* r1 ← 10
                                our array is of length 10 */
    bl print_array           /* call print_array */

    /* prepare call to qsort */
    /*
    void qsort(void *base,
         size_t nmemb,
         size_t size,
         int (*compar)(const void *, const void *));
    */
    ldr r0, addr_of_my_array /* r0 ← &my_array
                                base */
    mov r1, #10              /* r1 ← 10
                                nmemb = number of members
                                our array is 10 elements long */
    mov r2, #4               /* r2 ← 4
                                size of each member is 4 bytes */
    sub r3, fp, #28          /* r3 ← fp - 28 */
    bl qsort                 /* call qsort */

    /* prepare call to printf */
    ldr r1, [fp, #-4]                    /* r1 ← counter */
    ldr r0, addr_of_comparison_message   /* r0 ← &comparison_message */
    bl printf                            /* call printf */

    /* now print again the array to see if elements were sorted */
    /* prepare call to print_array */
```

```
    ldr r0, addr_of_my_array  /* r0 ← &my_array */
    mov r1, #10               /* r1 ← 10
                                 our array is of length 10 */
    bl print_array            /* call print_array */
```

Note that the argument compar passed to qsort (line 123) is not the address of the nested function but the trampoline. In fact, it is not the trampoline but its third word since, as we know, the two first words of the trampoline are the address of the nested function to call and the lexical scope (that we set earlier, lines 91 and 94).
Finally we return from main as usual.

```
    mov r0, #0                /* r0 ← 0 set errorcode to 0 prior returning from main */

    mov sp, fp
    pop {r4, r5, r6, fp, lr}      /* restore callee-saved registers */
    bx lr                     /* return */

addr_of_my_array: .word my_array
addr_of_comparison_message : .word comparison_message
```

```
4
1
4
5
1
4
6
```

The nested comparison function is next.

```
    /* nested function integer comparison */
    addr_of_integer_comparison_count : .word integer_comparison_count
    integer_comparison_count:
        /* r0 will be the address to the first integer */
        /* r1 will be the address to the second integer */
        push {r4, r5, r10, fp, lr} /* keep callee-saved registers */
        mov fp, sp                 /* setup dynamic link */

        ldr r0, [r0]    /* r0 ← *r0
                           load the integer pointed by r0 in r0 */
        ldr r1, [r1]    /* r1 ← *r1
                           load the integer pointed by r1 in r1 */

        cmp r0, r1      /* compute r0 - r1 and update cpsr */
        moveq r0, #0    /* if cpsr means that r0 == r1 then r0 ←  0 */
        movlt r0, #-1   /* if cpsr means that r0 <  r1 then r0 ← -1 */
        movgt r0, #1    /* if cpsr means that r0 >  r1 then r0 ←  1 */

        ldr r4, [fp, #8]  /* r4 ← *(fp + 8)
                             get static link in the stack */
        ldr r5, [r4, #-4] /* r5 ← counter
                             get value of counter */
        add r5, r5, #1    /* r5 ← r5 + 1 */
        str r5, [r4, #-4] /* counter ← r5
                             update counter */

        mov sp, fp        /* restore stack */
        pop {r4, r5, r10, fp, lr} /* restore callee-saved registers */
```

```
76        bx lr           /* return */
```

As you can see, the nested function expects r10 to be correctly set. This is what the trampoline does.

Harvard architecture

If you try to run the program as shown, it will probably work. But it will do it by chance. The reason is that we are featuring some simple form of self modifying code.

The Raspberry Pi processor features a modified Harvard architecture. This means that at some point there exists a distinction between instructions memory (.text) and the data memory (.data). Nowadays there are not many processors that feature a strict distinction between instruction and data memory (so at some point the program and the data are both in main memory, commonly called the RAM) but such differentiation is kept for caches.
A cache is a smaller and faster memory, that sits between the processor and the main memory. It is used to speed up memory accesses since most of the time such accesses happen close to other memory accesses (i.e. accessing elements of an array, different local variables in the stack or one instruction after the other in implicit sequencing) or close in time (i.e. accessing several times the same local variable or executing the same instruction when the code is in a loop).

Most modern processors feature distinguished caches for data (called the data cache) and instructions (called the instruction cache). The reason for such differentiation is that accessing to memory to execute instruction has a different pattern than accessing to memory to load/store data. It is beneficial to make such distinction but it comes at some price: when a program manipulates data that later will be executed as instructions (like we did with the trampoline, but also when the operating system loads a program in memory) the view of the two caches respect to the program state becomes incoherent: changes that we did in the data will have effect in the data cache but not in the instruction cache. Conversely, since the instruction cache will only get data from the main memory (and not from the data cache), we need to write back all the changes we did in the data cache to the main memory (this is called flushing the cache). We also have to make sure the instruction cache effectively gets the instructions from the memory, rather than reusing previously loaded instructions (which would be stale now), so we have to invalidate (or clear) the instruction cache.

In ARM the instructions that flush and invalidate caches are privileged operations (done through coprocessor instructions on the coprocessor 15 which manages the memory system of the CPU). This means that only the operating system can execute such instructions. As you see, user code may have to request a cache clear. Linux provides a cacheflush system call for this purpose.
Recall that in chapter 19 we saw ho to make system calls.
According to the Linux kernel, register r0 must contain the address of the beginning of the region to be flushed and invalidated. r1 must contain the address of the first byte that will not be invalidated. r2 must be zero. The cacheflush service number, that has to be set in r7 is 0xf0002.

```
    push {r7}           /* keep r7 because we are going to modify it */
    mov r7, #0xf0000    /* r7 ← 0xf0000 */
    add r7, r7, #2      /* r7 ← r7 + 2. So r7 ← 0xf0002
                           We do this in two steps because
                           we cannot encode 0xf0002 in
                           the instruction */
    mov r0, sp          /* r0 ← sp */
    add r1, sp, #32     /* r1 ← sp + 32 */
    mov r2, #0          /* r2 ← 0 */
    swi 0               /* system call */
    pop  {r7}           /* restore r7 */
```

As an alternative we can call an internal function implemented in libgcc (the GCC low-level runtime library) called __clear_cache. This function will internally call the Linux service.

```
9
8
9
9
1
0
0
1
0
1
```

```
    /* prepare call to __clear_cache */
    mov r0, sp        /* r0 ← sp */
    add r1, sp, #32   /* r1 ← sp + 32 */
    bl __clear_cache /* call __clear_cache */
```

We will invalidate and flush the caches right after setting up the trampoline (lines 89 to 94).

Now it only remains to run our program.

```
$ ./trampoline-sort-array
82 70 93 77 91 30 42 6 92 64
Num comparisons: 22
6 30 42 64 70 77 82 91 92 93
```

You can see the full listing here.

Discussion

Given that nested functions require a lexical scope, they cannot be trivially passed as plain addresses to other functions. Today we have seen that by using a trampoline it is possible to pass them to functions that do not allow passing a lexical scope. The price is having to copy a template, the trampoline, having to set it up with the proper values. We also have to flush caches in order to avoid executing wrong code. It is complicated but doable.

Having to flush the cache is undesirable (although not required in all architectures) and may cause a severe degradation of performance. Performance-critical pieces of code typically would not want to do this.

A serious concern, though, with the trampoline approach relates to the fact that we need an executable stack. A modern operating system, like Linux, can mark regions of memory to be readable, writable or executable. A region of memory that is not executable may contain instructions but if we branch to that region the processor will signal a fault, and the operating system will likely kill our process. Being able to disable execution of specific memory regions is done for security purposes. Most of the time one does not have to execute instructions that are found in stack or .data section. Only .text makes sense in these cases to be executable.
If you check what we did above, we actually copied some code (which was in .text) into the stack and then, qsort branched to the stack. This is because our programs allow an executable stack. Executable stacks are linked to common program vulnerability exploits like buffer overflows.
As we've seen in this chapter and in the previous one, nested functions come with several downsides, so it is not surprising that several programming languages do not provide support for them.

That's all for today.

ARM assembler in Raspberry Pi – Chapter 25

July 4, 2015 Roger Ferrer Ibáñez, 6

In chapter 13 we saw VFPv2 and the fact that it allows vectorial operations on floating-point numbers. You may be wondering if such a similar feature exists for integers. The answer is yes although in a more limited way.

SIMD

SIMD stands for single instruction multiple data and means that an instruction can be used to perform the same operation on several operands at the same time. In chapter 13 and 14 we saw that by changing the len field in the fpscr and using at least one operand in the vectorial banks, then an instruction operated on len registers in the vectorial bank(s), effectively doing len times a floating point operation. This way, a single instruction like vadd.f32 could then be used to perform up to 8 floating point additions. This strategy of speeding up computation is also called data parallelism.

SIMD with integers

SIMD support for integers exists also in ARMv6 but it is more limited: the multiple data are the subwords (see chapter 21) of a general purpose register. This means that we can do 2 operations on the 2 half words of a general purpose register. Similarly, we can do and up to 4 operations on the 4 bytes of a general purpose register.

Motivating example

At this point you may be wondering what is the purpose of this feature and why it does exist. Let’s assume we have two 16-bit PCM audio signals sampled at some frequency (i.e. 44.1kHz like in a CD Audio). This means that at the time of recording the “analog sound” of each channel is sampled many times per second and the sample, which represents the amplitude of the signal, is encoded using a 16-bit number.

An operation we may want to do is mixing the two signals in one signal (e.g. prior playing that final signal through the speakers). A (slightly incorrect) way to do this is by averaging the two signals. The code belows is a schema of what we want to do.

```
short int channel1[num_samples]; // in our environment a 'short int' is a half-word
short int channel2[num_samples];

short int channel_out[num_samples];
for (i = 0; i < num_samples; i++)
{
   channel_out[i] = (channel1[i] + channel2[i]) / 2;
}
```

Now imagine we want to implement this in ARMv6. With our current knowledge the code would look like this (I will omit in these examples the AAPCS function call convention).

```
naive_channel_mixing:
    /* r0 contains the base address of channel1 */
    /* r1 contains the base address of channel2 */
    /* r2 contains the base address of channel_out */
    /* r3 is the number of samples */
    /* r4 is the number of the current sample
          so it holds that 0 ≤ r4 < r3 */

    mov r4, #0              /* r4 ← 0 */
    b .Lcheck_loop          /* branch to check_loop */
    .Lloop:
      mov r5, r4, LSL #1    /* r5 ← r4 << 1 (this is r5 ← r4 * 2) */
                           /* a halfword takes two bytes, so multiply
                              the index by two. We do this here because
                              ldrsh does not allow an addressing mode
                              like [r0, r5, LSL #1] */
      ldrsh r6, [r0, r5]    /* r6 ← *{signed half}(r0 + r5) */
      ldrsh r7, [r1, r5]    /* r7 ← *{signed half}(r1 + r5) */
      add r8, r6, r7        /* r8 ← r6 + r7 */
      mov r8, r8, ASR #1    /* r8 ← r8 >> 1 (this is r8 ← r8 / 2)*/
      strh r8, [r2, r5]     /* *{half}(r2 + r5) ← r8 */
      add r4, r4, #1        /* r4 ← r4 + 1 */
    .Lcheck_loop:
      cmp r4, r3            /* compute r4 - r3 and update cpsr */
      blt .Lloop            /* if r4 < r3 jump to the
```

```
                                beginning of the loop */
```

We could probably be happy with this code but if you were in the business of designing processors for embedded devices you would probably be sensitive to your customer codes. And chances are that your portable MP3 player (or any gadget able to play music) is “ARM inside”. So this is a code that is eligible for improvement from an architecture point of view.

Parallel additions and subtractions

ARMv6 data parallel instructions allow us to add/subtract the corresponding half words or bytes. It provides them both for unsigned integers and signed integers.

- Halfwords
  - Signed: sadd16, ssub16
  - Unsigned: uadd16, usub16
- Bytes
  - Signed: sadd8, ssub8
  - Unsigned: uadd8, usub8

It should not be hard to find obvious uses for these instructions. For instance, the following loop can benefit from the uadd8 instruction.

```
// unsigned char is an unsigned byte in our environment
// a, b and c are arrays of N unsigned chars
unsigned char a[N], b[N], c[N];

int i;
for (i = 0; i < N; i++)
{
  c[i] = a[i] + b[i];
}
```

Let’s first write a naive approach to the above loop, which is similar to the one in the beginning of the post.

```
naive_byte_array_addition:
    /* r0 contains the base address of a */
    /* r1 contains the base address of b */
    /* r2 contains the base address of c */
    /* r3 is N */
    /* r4 is the number of the current item
          so it holds that 0 ≤ r4 < r3 */

    mov r4, #0              /* r4 ← 0 */
    b .Lcheck_loop0         /* branch to check_loop0 */

    .Lloop0:
      ldrb r5, [r0, r4]     /* r5 ← *{unsigned byte}(r0 + r4) */
      ldrb r6, [r1, r4]     /* r6 ← *{unsigned byte}(r1 + r4) */
      add r7, r5, r6        /* r7 ← r5 + r6 */
      strb r7, [r2, r4]     /* *{unsigned byte}(r2 + r4) ← r7 */
      add r4, r4, #1        /* r4 ← r4 + 1 */
    .Lcheck_loop0:
       cmp r4, r3           /* perform r4 - r3 and update cpsr */
       blt .Lloop0          /* if cpsr means that r4 < r3 jump to loop0 */
```

This loop again is fine but we can do better by using the instruction uadd8. Note that now we will be able to add 4 bytes at a time. This means that we will have to increment r4 by 4.

```
simd_byte_array_addition_0:
    /* r0 contains the base address of a */
    /* r1 contains the base address of b */
    /* r2 contains the base address of c */
```

```
    /* r3 is N */
    /* r4 is the number of the current item
          so it holds that 0 ≤ r4 < r3 */

    mov r4, #0              /* r4 ← 0 */
    b .Lcheck_loop1         /* branch to check_loop1 */

    .Lloop1:
      ldr r5, [r0, r4]      /* r5 ← *(r0 + r4) */
      ldr r6, [r1, r4]      /* r6 ← *(r1 + r4) */
      sadd8 r7, r5, r6      /* r7[7:0] ← r5[7:0] + r6[7:0] */
                           /* r7[15:8] ← r5[15:8] + r6[15:8] */
                           /* r7[23:16] ← r5[23:16] + r6[23:16] */
                           /* r7[31:24] ← r5[31:24] + r6[31:24] */
                           /* rA[x:y] means bits x to y of the register rA */
      str r7, [r2, r4]      /* *(r2 + r4) ← r7 */
      add r4, r4, #4        /* r4 ← r4 + 4 */
    .Lcheck_loop1:
       cmp r4, r3           /* perform r4 - r3 and update cpsr */
       blt .Lloop1          /* if cpsr means that r4 < r3 jump to loop1 */
```

A subtlety of the above code is that it only works if N (kept in r3) is a multiple of 4. If it is not the case (and this includes when 0 ≤ r3 < 4), then the loop will do fewer iterations than expected. If we know that N is a multiple of 4, then nothing

else must be done. But if it may be not a multiple of 4, we will need what is called an epilog loop, for the remaining cases. Note that in our case, the epilog loop will have to do 0 (if N was a multiple of 4), 1, 2 or 3 iterations. We can implement it as a switch with 4 cases plus fall-through (see chapter 16) or if we are concerned about code size, with a loop. We will use a loop.

We cannot, though, simply append an epilog loop to the above loop,because it is actually doing more work than we want. When N is not a multiple of four, the last iteration will add 1, 2 or 3 more bytes that do not belong to the original array. This is a recipe for a disaster so we have to avoid this. We need to make sure that when we are in the loop, r4 is such that r4, r4 + 1, r4 + 2and r4 + 3 are valid elements of the array. This means that we should check that r4 < N, r4 + 1 < N, r4 + 2 < N and r4 + 3 < N. Since the last of these four implies the first three, it is enough to check that r4 + 3 < N.

Note that checking r4 + 3 < N would force us to compute r4 + 3 at every iteration in the loop, but we do not have to. Checking r4 + 3 < N is equivalent to check r4 < N - 3. N - 3 does not depend on r4 so it can be computed before the loop.

```
simd_byte_array_addition_2:
    /* r0 contains the base address of a */
    /* r1 contains the base address of b */
    /* r2 contains the base address of c */
    /* r3 is N */
    /* r4 is the number of the current item
          so it holds that 0 ≤ r4 < r3 */

    mov r4, #0              /* r4 ← 0 */
    sub r8, r3, #3          /* r8 ← r3 - 3
                               this is r8 ← N - 3 */
    b .Lcheck_loop2         /* branch to check_loop2 */

    .Lloop2:
      ldr r5, [r0, r4]      /* r5 ← *(r0 + r4) */
      ldr r6, [r1, r4]      /* r6 ← *(r1 + r4) */
      sadd8 r7, r5, r6      /* r7[7:0] ← r5[7:0] + r6[7:0] */
                           /* r7[15:8] ← r5[15:8] + r6[15:8] */
                           /* r7[23:16] ← r5[23:16] + r6[23:16] */
                           /* r7[31:24] ← r5[31:24] + r6[31:24] */
```

```
5
1
6
1
7
1
8
1
9
2
0
2
1
2
2
2
3
2
4
2
5
2
6
```

```
      str r7, [r2, r4]     /* *(r2 + r4) ← r7 */
      add r4, r4, #4       /* r4 ← r4 + 4 */
    .Lcheck_loop2:
      cmp r4, r8           /* perform r4 - r8 and update cpsr */
      blt .Lloop2          /* if cpsr means that r4 < r8 jump to loop2 */
                          /* i.e. if r4 < N - 3 jump to loop2 */
```

In line 10 where we compute r8 which will keep N - 3, we use it in line 24 to check the loop iteration.
The epilog loop follows.

```
2
7
2
8
2
9
3
0
3
1
3
```

```
    /* epilog loop */
    b .Lcheck_loop3       /* branch to check_loop3 */

    .Lloop3:
      ldrb r5, [r0, r4]  /* r5 ← *{unsigned byte}(r0 + r4) */
      ldrb r6, [r1, r4]  /* r6 ← *{unsigned byte}(r1 + r4) */
      add r7, r5, r6     /* r7 ← r5 + r6 */
      strb r7, [r2, r4]  /* *{unsigned byte}(r2 + r4) ← r7 */

      add r4, r4, #1     /* r4 ← r4 + 1 */
    .Lcheck_loop3:
```

```
2
3
3
3
4
3
5
3
6
3
7
3
8
3
9
```

```
        cmp r4, r3          /* perform r4 - r3 and update cpsr */
        blt .Lloop3         /* if cpsr means that r4 < r3 jump to loop 3 */
```

The epilog loop is like the naive one, but it will only run 0, 1, 2 or 3 iterations. This means that for big enough values of N, in practice all iterations will use the data parallel instructions and only up to 3 will have to use the slower approach.

Halving instructions

The data parallel instructions also come in a form where the addition/subtraction is halved. This means that it is possible to compute averages of half words and bytes easily.

- Halfwords
  - Signed: shadd16, shsub16
  - Unsigned: uhadd16, uhsub16
- Bytes
  - Signed: shadd8, shsub8
  - Unsigned: uhadd8, uhsub8

Thus, the motivating example of the beginning of the post can be implemented using the shsub16 instruction. For simplicity, let's assume that num_samples is a multiple of 2 (now we are dealing with halfwords) so no epilog is necessary.

```
better_channel_mixing:
    /* r0 contains the base address of channel1 */
    /* r1 contains the base address of channel2 */
    /* r2 contains the base address of channel_out */
    /* r3 is the number of samples */
    /* r4 is the number of the current sample
          so it holds that 0 ≤ r4 < r3 */

    mov r4, #0              /* r4 ← 0 */
    b .Lcheck_loop          /* branch to check_loop */
    .Lloop:
      ldr r6, [r0, r4]      /* r6 ← *(r0 + r4) */
      ldr r7, [r1, r4]      /* r7 ← *(r1 + r4) */
      shadd16 r8, r6, r7    /* r8[15:0] ← (r6[15:0] + r7[15:0]) >> 1*/
                           /* r8[31:16] ← (r6[31:16] + r7[31:16]) >> 1*/
      str r8, [r2, r4]      /* *(r2 + r4) ← r8 */
      add r4, r4, #2        /* r4 ← r4 + 2 */
    .Lcheck_loop:
      cmp r4, r3            /* compute r4 - r3 and update cpsr */
      blt .Lloop            /* if r4 < r3 jump to the
                               beginning of the loop */
```

Saturating arithmetic

Let's go back to our motivating example. We averaged the two 16-bit channels to mix them but, in reality, mixing is achieved by just adding the two channels. In general this is OK because signals are not correlated and the amplitude of a mixed sample usually can be encoded in 16-bit. Sometimes, though, the mixed sample may have an amplitude that falls outside the 16-bit range. In this case we want to clip the sample within the representable range. A sample with a too positive amplitude will be clipped to 215-1, a sample with a too negative amplitude will be clipped to -215.

With lack of hardware support, clipping can be implemented by checking overflow after each addition. So, every addition should check that the resulting number is in the interval [-32768, 32767] Let's write a function that adds two 32-bit integers and clips them in the 16-bit range.

```
.data
max16bit: .word 32767

.text

clipped_add16bit:
    /* first operand is in r0 */
    /* second operand is in r0 */
    /* result is left in r0 */
    push {r4, lr}              /* keep registers */

    ldr r4, addr_of_max16bit   /* r4 ← &max16bit */
    ldr r4, [r4]               /* r4 ← *r4 */
                              /* now r4 == 32767 (i.e. 2^15 - 1) */

    add r0, r0, r1             /* r0 ← r0 + r1 */
    cmp r0, r4                 /* perform r0 - r4 and update cpsr */
    movgt r0, r4               /* if r0 > r4 then r0 ← r4 */
    bgt end                    /* if r0 > r4 then branch to end */

    mvn r4, r4                 /* r4 ← ~r4
                                  now r4 == -32768 (i.e. -2^15) */
    cmp r0, r4                 /* perform r0 - r4 and update cpsr */
    movlt r0, r4               /* if r0 < r4 then r0 ← r4 */

    end:

    pop {r4, lr}               /* restore registers */
    bx lr                      /* return */
addr_of_max16bit: .word max16bit
```

As you can see, a seemingly simple addition that clips the result requires a bunch of instructions. As before, the code is correct but we can do much better thanks to the saturated arithmeticsinstructions of ARMv6.

- Halfwords
  - Signed: qadd16, qsub16
  - Unsigned: uqadd16, uqsub16
- Bytes
  - Signed: qadd8, qsub8
  - Unsigned: uqadd8, uqsub8

Now we can write a more realistic mixing of two channels.

```
more_realistic_channel_mixing:
    /* r0 contains the base address of channel1 */
    /* r1 contains the base address of channel2 */
    /* r2 contains the base address of channel_out */
    /* r3 is the number of samples */
    /* r4 is the number of the current sample
          so it holds that 0 ≤ r4 < r3 */

    mov r4, #0              /* r4 ← 0 */
    b .Lcheck_loop          /* branch to check_loop */
    .Lloop:
      ldr r6, [r0, r4]      /* r6 ← *(r0 + r4) */
      ldr r7, [r1, r4]      /* r7 ← *(r1 + r4) */
      qadd16 r8, r6, r7     /* r8[15:0] ← saturated_sum_16(r6[15:0], r7[15:0]) */
                           /* r8[31:16] ← saturated_sum_16(r6[31:16], r7[31:16]) */
      str r8, [r2, r4]      /* *(r2 + r4) ← r8 */
      add r4, r4, #2        /* r4 ← r4 + 2 */
    .Lcheck_loop:
      cmp r4, r3            /* compute r4 - r3 and update cpsr */
      blt .Lloop            /* if r4 < r3 jump to the
                               beginning of the loop */
```

That's all for today.

ARM assembler in Raspberry Pi – Chapter 26

October 30, 2016 Roger Ferrer Ibáñez, 6

In this chapter we will talk about a fascinating step that is required to create a program, even when using assembler. Today we will talk about linking.

Linkers, the magic between symbols and addresses

Linkers are an essential yet often forgotten tool. Their main job is sticking all the pieces that form our program in a way that it can be executed. The fundamental work of a link is binding symbolic names with addresses (i.e. physical names). This process is conceptually simple but it is full of interesting details. Linking is a necessary step when separate compilation is used.

Separate compilation and modules

Modules are a mechanism in which programming languages let their users split programs in different logical parts. Modularization requires some amount of support from the tools that implement the programming language. Separate compilation is a mechanism to achieve this. In C, a program may be decomposed in several source files. Usually compiling a C source file generates an object file, thus several source files will lead to several object files. These object files are combined using a linker. The linker generates the final program.

ELF

Given that several tools manipulate object files (compilers, assemblers, linkers) a common format comes handy. There are a few formats available for this purpose like COFF, Mach-O or ELF. In the UNIX world (including Linux) the most popular format is ELF (Executable and Linking Format). This format is used for object files (called relocatable objects, we will see below why), shared objects (dynamic libraries) and executables (the program itself).

For a linker, an ELF relocatable file is a collection of sections. Sections represent a contiguous chunk of data (which can be anything: instructions, initial values of global variables, debug information, etc). Each section has a name and attributes like whether it has to be allocated in memory, loaded from the image (i.e. the file that contains the program), whether it can be executed, whether it is writable, its size and alignment, etc.

Labels as symbolic names

When we use global variables we have to use the following schema:

```
.data:
var: .word 42
.text
func:
    /* ... */
    ldr r0, addr_of_var  /* r0 ← &var */
    ldr r0, [r0]         /* r0 ← *r0 */
    /* ... */
addr_of_var : .word var
```

The reason is that in ARM instructions we cannot encode the full 32-bit address of a variable inside an instruction. So it makes sense to keep the address in a place, in this case in addr_of_var, which is amenable for finding it from the current instruction. In the case shown above, the assembler replaces the usage of addr_of_var into something like this:

```
   ldr r0, [pc, #offset]
```

Which means load the value found in the given offset of the current instruction. The assembler computes the right offset here so we do not have to. This is a valid approach because addr_of_var is found in the same section as the instruction. This means that it will for sure be located after the instructions. It also happens that it is close enough in memory. This addressing mode can encode any offset of 12-bit (plus a sign bit) so anything within 4096 bytes (i.e. within 1024 instructions) is addressable this way.

But the question that remains is, what does the assembler put in the that location designated by addr_of_var? We have written .word var but what does this mean? The assembler should emit the address of var, but at this point its address is unknown. So the assembler can only emit partial information at this point. This information will be completed later.

An example

Let’s consider a more complex example to see this process in action. Consider the following code that takes two global variables and adds them into a result variable. Then we call a function, that we will write in another file. This function will increment the result variable by one. The result variable has to be accessible from the other file, so we will have to mark it as global (similar to what we do with main).

```
/* main.s */
.data

one_var : .word 42
another_var : .word 66

.globl result_var             /* mark result_var as global */
result_var : .word 0

.text

.globl main
main:
    ldr r0, addr_one_var      /* r0 ← &one_var */
    ldr r0, [r0]              /* r0 ← *r0 */
    ldr r1, addr_another_var  /* r1 ← &another_var */
    ldr r1, [r1]              /* r1 ← *r1 */
    add r0, r0, r1            /* r0 ← r0 + r1 */
    ldr r1, addr_result       /* r1 ← &result */
    str r0, [r1]              /* *r1 ← r0 */
    bl inc_result             /* call to inc_result */
    mov r0, #0                /* r0 ← 0 */
    bx lr                     /* return */


addr_one_var  : .word one_var
addr_another_var  : .word another_var
addr_result  : .word result_var
```

Let's create an object file. Recall that an object file is an intermediate file that is used before we create the final program. Once created, we can use objdump -d to see the code contained in this object file. (The use of -march=armv6 avoids some legacy info be emitted that would be confusing for the sake of the exposition)

```
$ as -march=armv6 -o main.o main.s      # creates object file main.o
```

Relocations

We said above that the assembler does not know the final value and instead may put some partial information (e.g. the offsets from .data). It also annotates that some fix up is required here. This fix up is called a relocation. We can read the relocations using flags -dr of objdump.

```
$    objdump    -dr
main.o
```

```
main.o:     file format elf32-littlearm

Disassembly of section .text:

00000000 <main>:
   0:   e59f0020        ldr     r0, [pc, #32]   ; 28 <addr_one_var>
   4:   e5900000        ldr     r0, [r0]
   8:   e59f101c        ldr     r1, [pc, #28]   ; 2c <addr_another_var>
   c:   e5911000        ldr     r1, [r1]
  10:   e0800001        add     r0, r0, r1
  14:   e59f1014        ldr     r1, [pc, #20]   ; 30 <addr_result>
  18:   e5810000        str     r0, [r1]
  1c:   ebfffffe        bl      0 <inc_result>
                        1c: R_ARM_CALL  inc_result
  20:   e3a00000        mov     r0, #0
  24:   e12fff1e        bx      lr

00000028 <addr_one_var>:
  28:   00000000        .word   0x00000000
                        28: R_ARM_ABS32 .data

0000002c <addr_another_var>:
  2c:   00000004        .word   0x00000004
                        2c: R_ARM_ABS32 .data

00000030 <addr_result>:
  30:   00000000        .word   0x00000000
                        30: R_ARM_ABS32 result_var
```

Relocations are rendered the output above like

```
                        OFFSET: TYPE    VALUE
```

They are also printed right after the point they affect.

OFFSET is the offset inside the section for the bytes that will need fixing up (in this case all of them inside .text). TYPE is the kind of relocation. The kind of relocation determines which and how bytes are fixed up. VALUE is a symbolic entity for which we have to figure the physical address. It can be a real symbol, like inc_result and result_var, or a section name like .data.
In the current list, there is a relocation at .text+1c so we can call the actual inc_result. The other two relocations in .text+28, .text+2c are the relocations required to access .data. These relocations could have as VALUE the symbols one_var and another_var respectively but GNU as seems to prefer to represent them as offsets relative to .data section. Finally .text+30 refers to the global symbol result_var.
Every relocation kind is defined in terms of a few parameters: S is the address of the symbol referred by the relocation (the VALUE above), P is the address of the place (the OFFSET plus the address of the section itself), A (for addenda) is the value that the assembler has left in place. In our example, R_ARM_ABS32 it is the value of the .word, for R_ARM_CALL it is a set of bits in the bl instruction itself. Using these parameters, earch relocation has a related operation. Relocations of kind R_ARM_ABS32 do an operation S + A. Relocations of kind R_ARM_CALLdo an operation (S + A) – P.
Due to Thumb, ARM relocations have an extra parameter T that has the value 1 if the symbol S is a Thumb function, 0 otherwise. This is not the case for our examples, so I have omitted Tin the description of the relocations above

Before we can see the result computed by the linker, we will define inc_result otherwise linking will fail. This function will increment the value of addr_result (whose storage is defined in the first file main.s).

```
/* inc_result.s */
.text

.globl inc_result
inc_result:
    ldr r1, addr_result  /* r1 ← &result */
    ldr r0, [r1]         /* r0 ← *r1 */
    add r0, r0, #1       /* r0 ← r0 + 1 */
```

```
    str r0, [r1]          /* *r1 ← r0 */
    bx lr                 /* return */

addr_result  : .word result_var
```

Let’s check the relocations as well.

```
$ as -march=armv6 -o inc_result.o inc_result.s
$ objdump -dr inc_result.o
```

```
inc_result.o:     file format elf32-littlearm

Disassembly of section .text:

00000000 <inc_result>:
   0:   e59f100c        ldr     r1, [pc, #12]   ; 14 <addr_result>
   4:   e5910000        ldr     r0, [r1]
   8:   e2800001        add     r0, r0, #1
   c:   e5810000        str     r0, [r1]
  10:   e12fff1e        bx      lr

00000014 <addr_result>:
  14:   00000000        .word   0x00000000
                        14: R_ARM_ABS32 result_var
```

We can see that it has a relocation for result_var as expected.

Now we can combine the two object files to generate an executable binary.

```
$  gcc  -o  test.exe  print_float.o
reloc.o
```

And check the contents of the file. Our program will include a few functions from the C library that we can ignore.

```
$ objdump -d test.exe
```

```
...
00008390 <main>:
    8390:       e59f0020        ldr     r0, [pc, #32]   ; 83b8 <addr_one_var>
    8394:       e5900000        ldr     r0, [r0]
    8398:       e59f101c        ldr     r1, [pc, #28]   ; 83bc <addr_another_var>
```

```
    839c:       e5911000        ldr     r1, [r1]
    83a0:       e0800001        add     r0, r0, r1
    83a4:       e59f1014        ldr     r1, [pc, #20]   ; 83c0 <addr_result>
    83a8:       e5810000        str     r0, [r1]
    83ac:       eb000004        bl      83c4 <inc_result>
    83b0:       e3a00000        mov     r0, #0
    83b4:       e12fff1e        bx      lr

000083b8 <addr_one_var>:
    83b8:       00010578        .word   0x00010578

000083bc <addr_another_var>:
    83bc:       0001057c        .word   0x0001057c

000083c0 <addr_result>:
    83c0:       00010580        .word   0x00010580

000083c4 <inc_result>:
    83c4:       e59f100c        ldr     r1, [pc, #12]   ; 83d8 <addr_result>
    83c8:       e5910000        ldr     r0, [r1]
    83cc:       e2800001        add     r0, r0, #1
    83d0:       e5810000        str     r0, [r1]
    83d4:       e12fff1e        bx      lr

000083d8 <addr_result>:
    83d8:       00010580        .word   0x00010580

...
```

From the output above we can observe that addr_one_var is in address 0x00010578, addr_another_var is in address 0x0001057c and addr_result is in address 0x00010580. The last one appears repeated, but this is because both files main.s and inc_result.s refer to it so they need to keep the address somewhere. Note that in both cases it contains the same address.

Let’s start with the relocations of addr_one_var, addr_another_var and addr_result. These three relocations were R_ARM_ABS32 so their operation is S + A. S is the address of section .data whose address can be determined also

with objdump -h (plus flag -w to make it a bit more readable). A file may contain many sections so I will omit the uninteresting ones.

```
$ objdump -hw test.exe
```

```
test.exe:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn  Flags
...
 13 .text         0000015c  000082e4  000082e4  000002e4  2**2  CONTENTS, ALLOC, LOAD, READONLY, CODE
...
 23 .data         00000014  00010570  00010570  00000570  2**2  CONTENTS, ALLOC, LOAD, DATA
...
```

Column VMA defines the address of the section. In our case .data is located at 00010570. And our variables are found in 0x00010578, 0x0001057c and 0x00010580. These are offsets 8, 12 and 16 respectively from the beginning of .data. The linker has laid some other variables in this section before ours. We can see this asking the linker to print a map of the generated executable.

```
$ gcc -o test.exe main.o inc_result.o -Wl,--print-map > map.txt
$ cat map.txt
```

```
.data           0x00010570       0x14
                0x00010570                PROVIDE (__data_start, .)
 *(.data .data.* .gnu.linkonce.d.*)
  .data                 0x00010570             0x4  /usr/lib/gcc/arm-linux-gnueabihf/4.6/../../../arm-linux-gnueabihf/crt1.o
                0x00010570                data_start
                0x00010570                __data_start
  .data                 0x00010574             0x0  /usr/lib/gcc/arm-linux-gnueabihf/4.6/../../../arm-linux-gnueabihf/crti.o
 .data          0x00010574        0x4 /usr/lib/gcc/arm-linux-gnueabihf/4.6/crtbegin.o
                0x00010574                __dso_handle
 .data          0x00010578        0xc main.o
                0x00010580                result_var
 .data          0x00010584        0x0 inc_result.o
 .data          0x00010584        0x0 /usr/lib/arm-linux-gnueabihf/libc_nonshared.a(elf-init.oS)
 .data          0x00010584        0x0 /usr/lib/gcc/arm-linux-gnueabihf/4.6/crtend.o
```

```
193203
  .data          0x00010584        0x0 /usr/lib/gcc/arm-linux-gnueabihf/4.6/../../../arm-linux-gnueabihf/cr
```

If you check lines 317 to 322, you will see that that the final .data section (that effectively starts 0x00010570 as we checked above) of our program includes 4 bytes from crt1.o for the symbols data_start (and its alias __data_start). File crtbegin.o also has contributed a symbol __dso_handle. These global symbols come from the C library. Only symbol result_var appears here because is a global symbol, all other global variables are not global symbols. The storage, though, is accounted for all of them in line 323. They take 0xc bytes (i.e. 12 bytes because of 3 variables each one of 4 bytes).

So with this info we can infer what has happened: variable one_var is in address 0x00010570, variable another_var is in 0x00010574 and variable result_var is in 0x00010578. If you check the result of objdump -d test.exe above you will see that

```
000083b8 <addr_one_var>:
    83b8:       00010578        .word   0x00010578

000083bc <addr_another_var>:
    83bc:       0001057c        .word   0x0001057c

000083c0 <addr_result>:
    83c0:       00010580        .word   0x00010580
...
000083d8 <addr_result>:
    83d8:       00010580        .word   0x00010580
```

What about the call to inc_result?

```
    83ac:       eb000004        bl      83c4
```

This one is a bit more involved. Recall that the relocation operation is (S + A) - P. Here A is 0and P is 0x000083ac, S is 0x000083c4. So the relocation has to define an offset of 24 bytes (83c4 – 83ac is 24(10). Instruction bl encodes the offset by shifting it 2 bits to the right. So the current offset encoded in eb000004 is 16. Recall that the current pc points

to the current instruction plus 8 bytes, so this instruction is exactly telling us to jump to an offset + 24 bytes. Exactly what we wanted.

```
...
    83ac:       eb000004        bl      83c4 <inc_result>
    83b0:       e3a00000        mov     r0, #0
    83b4:       e12fff1e        bx      lr

000083b8 <addr_one_var>:
    83b8:       00010578        .word   0x00010578

000083bc <addr_another_var>:
    83bc:       0001057c        .word   0x0001057c

000083c0 <addr_result>:
    83c0:       00010580        .word   0x00010580

000083c4 <inc_result>:
    83c4:       e59f100c        ldr     r1, [pc, #12]   ; 83d8 <addr_result>

...
```

More information

Linkers are a bit of arcana because they must handle with the lowest level parts of code. So sometimes it is hard to find good resources on them.

Ian Lance Taylor, author of gold, made a very nice linker essay in 20 chapters. If you want a book, Linkers & Loaders is not a bad one. The ELF standard is actually defined in two parts, a genericone and a processor specific one, including one for ARM.

That’s all for today.

Rreferences

http://thinkingeek.com/arm-assembler-raspberry-pi/