ARM assembler in Raspberry Pi – Chapter 1

January 9, 2013 rferrer, 3

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.

```
1 /* -- first.s */
2 /* This is a comment */
3 .global main /* 'main' is our entry point and must be global */
4 .func main   /* 'main' is a function */
5
6 main:          /* This is main */
7     mov r0, #2 /* Put a 2 inside the register r0 */
```

Calendar

January 2013

M	T	W	T	F	S	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			
« Dec				Feb »		

Recent Posts

- Capybara, pop up windows and the new PayPal sandbox
- ARM assembler in Raspberry Pi – Chapter 12
- ARM assembler in Raspberry Pi – Chapter 11
- ARM assembler in Raspberry Pi – Chapter 10
- ARM assembler in Raspberry Pi – Chapter 9

Recent Comments

rferrer on ARM assembler in Raspberry Pi – Chapter 11

Einstieg in Pi-Assembler | ultramachine on ARM assembler in Raspberry Pi – Chapter 1

Fernando on ARM assembler in Raspberry Pi – Chapter 7

Loren Blaney on ARM assembler in Raspberry Pi – Chapter 11

ห.ร.ม. ด้วยภาษา Assembly บน Raspberry Pi | Raspberry Pi Thailand on ARM assembler in Raspberry Pi – Chapter 9

Tags

.net activerecord ajax apple archlinux arm assembler bind branches C# dhcp firebug firefox function function call functions gadgets html indexing modes ipod Java javascript jquery linux mac os mac os x MVC networking parallels pi programming tips rails raspberry ruby ruby on rails security software sports sql server subversion tips and tricks tools ubuntu visual studio xmonad

```
8     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 `$ `).

```
1 $ as -o first.o first.s
```

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

```
1 $ gcc -o first first.o
```

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

```
1 $ ./first
```

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

```
1 $ ./first ; echo $?
2 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.

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

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.

```
4 .func main   /* 'main' is a function */
```

Another GNU assembler directive. Here we state that `main` is a function. This is important

Archives

April 2013
March 2013
February 2013
January 2013
December 2012
November 2012
August 2012
July 2012
June 2012
February 2012
January 2012
December 2011
November 2011
October 2011
July 2011
June 2011
May 2011
April 2011
March 2011
February 2011
December 2010
November 2010
October 2009
July 2009
June 2009
March 2009
November 2008
July 2008
September 2007
July 2007
June 2007

because an assembler program usually contains instructions (i.e. code) but may also contain data. We need to explicitly state that `main` actually refers to a function, because it is code.

```
6 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.

```
7     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).

```
8     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, pi, raspberry

Fast and easy way to block bots from your website using Apache
ARM assembler in Raspberry Pi – Chapter 2

3 thoughts on “ARM assembler in Raspberry Pi – Chapter 1”

Jonathan Hinchliffe says:
March 6, 2013 at 10:22 pm

Really excellent tutorial.

Reply

Fernando says:
March 21, 2013 at 12:15 am