ARM assembler in Raspberry Pi – Chapter 4

January 12, 2013 rferrer, 0

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)...d
(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.
```

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

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.

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

```
(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, debugger, gdb, pi, raspberry

ARM assembler in Raspberry Pi – Chapter 3 **ARM assembler in Raspberry Pi – Chapter 5**

Leave a Reply

Your email address will not be published. Required fields are marked *

Name *