

ARM Assembly Programming using Raspberry Pi

MicroDigitalEd
and
Gerardo Ospina

July 3rd, 2016

Contents

1	Introduction	2
2	GCC Tools	4
2.1	Sample Program	5
3	How to Assemble, Link and Run the Program	6
4	Program Termination	9
5	Using GDB	10
5.1	Preparation to Use GDB	10
5.2	Exit GDB	11
5.3	List source code	11
5.4	Set breakpoint	12
5.5	Run the program	12
5.6	Examine the CPU registers	12
5.7	Disassemble machine code	13
5.8	Step through the instructions	14
5.9	Continue program execution	14
5.10	Examine the memory	15
6	Floating Point	16
7	Program Output	17
7.1	Program Return Value	17

Chapter 1

Introduction

The Raspberry Pi is an inexpensive credit-card sized Linux computer. At its core is an ARMv6 CPU. The free download Raspbian package (from NOOBS <http://www.raspberrypi.org/help/noobs-setup/>) contains all the software necessary for ARM assembly language programming.

The downloaded package includes Raspbian operating system and several programming language supports. Among them is the GNU Compiler Collection (GCC) which supports programming in C, C++ and assembly languages.

In this document, we will use the commands `as` (assembler), `ld` (link loader), and `gdb` (GNU debugger) from GCC. These are command of command line interface that can be executed from the command prompt.

We will assume the reader is comfortable using the command line interface of the Raspberry Pi. The Raspbian software package comes with two command line text editors: nano editor and vi that may be used to enter and edit the assembly source code. For more text editor options , please visit:

<http://www.raspberrypi.org/documentation/linux/usage/text-editors.md>

Lastly, the screenshots in this document were captured remotely using PuTTY emulator (see figure 1.1). The content should look identical to the console display of the Raspberry Pi.

In order to use PuTTY your computer and the Raspberry Pi must be in the same network. You must know the IP address and an user/password pair for the Raspberry pi.

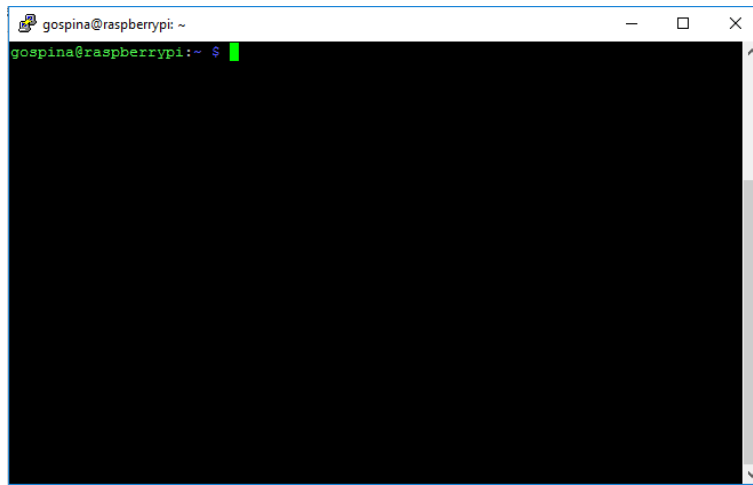


Figure 1.1: PuTTY

Below (figure 1.2) are the versions of the assembler and linker used in this document.

```
gospina@raspberrypi:~ $ as -v
GNU assembler version 2.25 (arm-linux-gnueabihf) using BFD version (GNU Binutils
for Raspbian) 2.25
^C
gospina@raspberrypi:~ $ ld -v
GNU ld (GNU Binutils for Raspbian) 2.25
gospina@raspberrypi:~ $
```

Figure 1.2: Display of version numbers for as and ld

Chapter 2

GCC Tools

Programs developed using GCC tools in Raspberry Pi are applications running under the Raspbian OS. Here are some of the major characteristics:

1. The code and data of the program are all loaded in the RAM allocated by the OS using virtual memory addressing. The virtual memory address is the address that the program sees. From a programmers perspective, it should make no difference unless you are attempting to read/write the hardware registers.
2. The program is running under Raspbian OS. The OS provides services such as console read/write or file I/O.
3. GCC was developed to support many different processors. With the recent version of GCC assembler (v2.25), ARM instructions are recognized.
 - Comments are preceded by @. The assembler also accepts C++ style comment enclosed with /* */ and //.
 - Labels must be followed by a colon (:).
 - Below (table 2.1) is a comparison table of the frequently used directives:

GCC Directive	Explanation
<code>.text</code>	Signifies the beginning of code or constant
<code>.data</code>	Signifies the beginning of read/write data
<code>.global <i>label</i></code>	Makes the label visible to linker
<code>.extern <i>label</i></code>	label is declared outside of this file
<code>.byte <i>byte</i> [<i>,byte, byte, ...</i>]</code>	Declares byte (8-bit) data
<code>.hword <i>hw</i> [<i>,hw, hw, ...</i>]</code>	Declares halfword (16-bit) data
<code>.word <i>word</i> [<i>,word, word, ...</i>]</code>	Declares word (32-bit) data
<code>.float <i>float</i> [<i>,float, float, ...</i>]</code>	Declares single precision floating point (32-bit) data
<code>.double <i>double</i> [<i>,double, double, ...</i>]</code>	Declares double precision floating point (64-bit) data
<code>.space <i>bytes</i> [<i>,fill</i>]</code>	Declares memory (in bytes) with optional fill
<code>.align <i>n</i></code>	Aligns address to <2 to the power of n byte
<code>.ascii "<i>ASCII string</i>"</code>	Declares an ASCII string
<code>.asciz "<i>ASCII string</i>"</code>	Declares an ASCII string with null termination
<code>.equ <i>symbol, value</i></code>	Sets the symbol with a constant value
<code>.set <i>variable, value</i></code>	Sets the variable with a new value
<code>.end</code>	Signifies the end of the program

Table 2.1: Frequently used GCC assembler directives

2.1 Sample Program

Below is a sample program that was written in GCC syntax for Raspberry Pi.

```
@P2_1.s ARM Assembly Language Program To Add Some Data and Store the SUM in R3.
.text
.global _start
_start: MOV R1, #0x25 @ R1 = 0x25
        MOV R2, #0x34 @ R2 = 0x34
        ADD R3, R2, R1 @ R3 = R2 + R1
HERE:   B   HERE      @ stay here forever
.end
```

GCC linker is expecting a label `_start` for the entry point of the program. This label also must be made global so that it is visible to the linker.

Chapter 3

How to Assemble, Link and Run the Program

In this example, we enter the program above into a file name p2_1.s in the \$HOME/asm directory, assemble, link, and execute the program.

First we make a directory with the name asm, change the current directory to asm and launch the editor vi for the file p2_1.s. We are showing (Figure 3.1) the use of editor vi here but you may use any text editor you prefer.

```
gospina@raspberrypi:~ $ mkdir asm
gospina@raspberrypi:~ $ cd asm
gospina@raspberrypi:~/asm $ vi p2_1.s
```

Figure 3.1: Make asm directory, change to asm directory and launch editor vi

After typing in the program in vi (Figure 3.2), the file is saved.

```
@P2_1.s ARM Assembly Language Program To Add Some Data and Store the SUM in R3.
.text
.global _start
_start: MOV R1, #0x25 @ R1 = 0x25
        MOV R2, #0x34 @ R2 = 0x34
        ADD R3, R2, R1 @ R3 = R2 + R1
HERE:   B   HERE      @ stay here forever
.end
```

Figure 3.2: The sample program viewed in editor vi

The program is assembled using command “as -o p2_1.o p2_1.s”. In this command, “as” is the name of the assembler, “-o p2_1.o” tells the assembler to create the output object file with the name p2_1.o, and lastly, “p2_1.s” is the assembly source file name (see below Figure 3.3).

```
gospina@raspberrypi:~/asm $ as -o p2_1.o p2_1.s
gospina@raspberrypi:~/asm $
```

Figure 3.3: The assemble command

Like many Unix¹ programs, it produces no output to the console when the program ran without errors.

Linker (Figure 3.4) takes one or more object files and creates an executable file. To run the linker, use command “ld -o p2_1 p2_1.o”. In this command, “ld” is the name of the linker program, “-o p2_1” tells the linker to produce the output executable file with the name p2_1, and lastly, “p2_1.o” is the input object file name.

```
gospina@raspberrypi:~/asm $ as -o p2_1.o p2_1.s
gospina@raspberrypi:~/asm $ ld -o p2_1 p2_1.o
gospina@raspberrypi:~/asm $
```

Figure 3.4: The linker command

Again, the linker produces no output to the console when there were no errors.

To execute the program (Figure 3.5), type the command `./p2_1` at the prompt. It tells the shell to execute the program named p2_1 at the current directory.

```
gospina@raspberrypi:~/asm $ as -o p2_1.o p2_1.s
gospina@raspberrypi:~/asm $ ld -o p2_1 p2_1.o
gospina@raspberrypi:~/asm $ ./p2_1
```

Recall the last instruction in the program is an infinite loop. After executing the first three instructions, the program is stuck at the infinite loop that consumes 100% of the CPU time. An infinite loop is typical for a program in a simple embedded system without an operating system. For now, type Ctrl-C (Figure 3.6) to terminate the program and get the prompt back.

¹Raspbian is ported from Debian which derived from Linux and Linux is derived from Unix. All of them are very similar. In this document we will use Unix as a generic term for these operating system.


```
gospina@raspberrypi:~/asm $ as -o p2_1.o p2_1.s
gospina@raspberrypi:~/asm $ ld -o p2_1 p2_1.o
gospina@raspberrypi:~/asm $ ./p2_1
^C
gospina@raspberrypi:~/asm $ █
```

Figure 3.6: Ctrl-C to terminate the program

Chapter 4

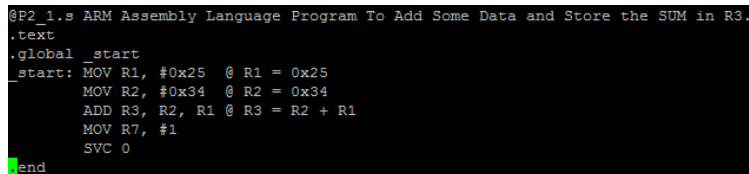
Program Termination

As an application running under a multitasking operating system, the program should be terminated when done. Otherwise, the program running a dummy infinite loop will consume the CPU time and slow down all the other programs.

To terminate a program, replace the dummy infinite loop at the end of the program HERE: B HERE by:

```
MOV R7, #1
SVC 0
```

The number 1 placed in Register 7 tells the operating system to terminate this program. The instruction “SVC 0” is the system call, that transfers the program execution to the operating system. If you place a different number in R7, the operating system will perform a different service.



```
@P2_1.s ARM Assembly Language Program To Add Some Data and Store the SUM in R3.
.text
.global _start
_start: MOV R1, #0x25 @ R1 = 0x25
        MOV R2, #0x34 @ R2 = 0x34
        ADD R3, R2, R1 @ R3 = R2 + R1
        MOV R7, #1
        SVC 0
end
```

Figure 4.1: Modified program

After replacing the dummy infinite loop with the system call to end the program (See Figure 4.1), run the assembler and linker again. This time after you execute the program, the program will terminate by itself and the prompt reappears immediately without user intervention.

Chapter 5

Using GDB

A computer without output is not very interesting like the previous example program. We will see how to generate output from a program in the next section, but for most of the programs in this book, they demonstrate the manipulations of data between CPU registers and memory without any output. GDB (GNU Debugger) is a great tool to use to study the assembly programs. You can use GDB to step through the program and examine the contents of the registers and memory.

In the following example, we will demonstrate how to control the execution of the program using GDB and examine the register and memory content.

5.1 Preparation to Use GDB

When a program is assembled, the executable machine code is generated. To ease the task of debugging, you add a flag `-g` to the assembler command line then the symbols and line numbers of the source code will be preserved in the output executable file and the debugger will be able to link the machine code to the source code line by line. To do this, assemble the program with the command (`$` is the prompt):

```
$ as -g -o p2_1.o p2_1.s
```

The linker command stays the same:

```
$ ld -o p2_1 p2_1.o
```

To launch the GNU Debugger, type the command `gdb` followed by the executable file name at the prompt:

```
$ gdb p2_1
```

After displaying the license and warranty statements, the prompt is shown as (`gdb`) (Figure 5.1).

```

gospina@raspberrypi:~/asm $ as -g -o p2_1.o p2_1.s
gospina@raspberrypi:~/asm $ ld -o p2_1 p2_1.o
gospina@raspberrypi:~/asm $ gdb p2_1
GNU gdb (Raspbian 7.7.1+dfsg-5) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from p2_1...done.
(gdb) █

```

Figure 5.1: Assemble with debug option and launch of gdb

5.2 Exit GDB

GDB has a wealth of commands. We will only touch a few here. At the gdb prompt, you only need to type the minimal number of characters to be distinctive for the command. We will show the full command but underscore the minimal required characters. To exit GDB, the command is “quit” (Figure 5.2).

```

(gdb) q
gospina@raspberrypi:~/asm $ █

```

Figure 5.2: Quit GDB

5.3 List source code

To list the source code, the command is “list”. The list command displays 10 lines of the source code with the line number in front of each line (Figure 5.3). To see the next 10 lines, just hit the Enter key.

```

(gdb) l
1      @P2_1.s ARM Assembly Language Program To Add Some Data and Store the SUM
      in R3.
2      .text
3      .global _start
4      _start: MOV R1, #0x25 @ R1 = 0x25
5              MOV R2, #0x34 @ R2 = 0x34
6              ADD R3, R2, R1 @ R3 = R2 + R1
7              MOV R7, #1
8              SVC 0
9      .end
(gdb) █

```

Figure 5.3: Use gdb command list to display source code with line number

5.4 Set breakpoint

To be able to examine the registers or memory, we need to stop the program in its execution. Setting a breakpoint will stop the program execution before the breakpoint. The command to set breakpoint is “`break`” followed by line number. The following command sets a breakpoint at line 6 of the program. When we run the program, the program execution will stop right before line 6 is executed.

```
(gdb) b 6
```

GDB will provide a confirmation of the breakpoint (Figure 5.4).

```
(gdb) b 6
Breakpoint 1 at 0x1005c: file p2_1.s, line 6.
(gdb)
```

Figure 5.4: Set a breakpoint at line 6 of the source code

5.5 Run the program

To start the program, use command “`run`”. Program execution will start from the beginning until it hits the breakpoint. The line just following where the breakpoint was set will be displayed (Figure 5.5). Remember, this instruction has not been executed yet.

```
(gdb) r
Starting program: /home/gospina/asm/p2_1

Breakpoint 1, _start () at p2_1.s:6
6          ADD R3, R2, R1 @ R3 = R2 + R1
(gdb)
```

Figure 5.5: Run the program and it stops at the breakpoint

5.6 Examine the CPU registers

With the program stopped before line 6, the last instruction on line 5 moved a literal value 0x25 into Register 1. We can verify that by using command “`info registers`”. The display consists of three columns: the register name, the contents in hexadecimal, and the contents in decimal (Figure 5.6). The registers holding addresses such as SP or PC will not display decimal values.

```

(gdb) i r
r0          0x0      0
r1          0x25     37
r2          0x34     52
r3          0x0      0
r4          0x0      0
r5          0x0      0
r6          0x0      0
r7          0x0      0
r8          0x0      0
r9          0x0      0
r10         0x0      0
r11         0x0      0
r12         0x0      0
sp          0x7efff770 0x7efff770
lr          0x0      0
pc          0x1005c 0x1005c <_start+8>
cpsr       0x10     16
(gdb)

```

Figure 5.6: Display register contents using info register command

5.7 Disassemble machine code

GDB has the ability of disassembling the machine code back to assembly instructions. The command is “disassemble” (Figure 5.7). Because the assembler does more than translating the source code to machine instructions, the disassembled result may differ from the original source code. For example, a pseudo-instruction entered as “ldr R5, =0x1234” in the source code may have the disassembled output as “ldr r5, [PC, 32]”.

```

(gdb) disas
Dump of assembler code for function _start:
0x00010054 <+0>:    mov     r1, #37 ; 0x25
0x00010058 <+4>:    mov     r2, #52 ; 0x34
=> 0x0001005c <+8>:    add     r3, r2, r1
0x00010060 <+12>:   mov     r7, #1
0x00010064 <+16>:   svc     0x00000000
End of assembler dump.
(gdb)

```

Figure 5.7: Disassemble the machine code

In the disassembled display, the breakpoint instruction is marked by an arrow at the left margin. The disassemble command also takes a pair of starting address and ending address separated by a comma such as:

```
$ disas 0x10054, 0x1006c
```

Note in the example in Figure 5.8 that although the last instruction of the program is at address 0x00010060, the disassembled output continued until the specified address was met.

```

(gdb) disas 0x10054, 0x1006c
Dump of assembler code from 0x10054 to 0x1006c:
0x00010054 <_start+0>:      mov     r1, #37 ; 0x25
0x00010058 <_start+4>:      mov     r2, #52 ; 0x34
=> 0x0001005c <_start+8>:      add     r3, r2, r1
0x00010060 <_start+12>:     mov     r7, #1
0x00010064 <_start+16>:     svc     0x00000000
0x00010068:      andeq    r1, r0, r1, asr #6
End of assembler dump.
(gdb)

```

Figure 5.8: Disassemble the machine code between address 0x10054 and 0x1006C

5.8 Step through the instructions

When the program execution is halted by the breakpoint, we may continue by stepping one instruction at a time by using command “stepi”. The step instruction command may also take a numeric argument to step more than one instruction at a time. For example, “stepi 5” will execute the five instructions or until another breakpoint is hit.

In the example below in Figure 5.9, we stepped two instructions and examined the register contents.

```

(gdb) stepi
7      MOV R7, #1
(gdb) stepi
8      SVC 0
(gdb) i r
r0      0x0      0
r1      0x25     37
r2      0x34     52
r3      0x59     89
r4      0x0      0
r5      0x0      0
r6      0x0      0
r7      0x1      1
r8      0x0      0
r9      0x0      0
r10     0x0      0
r11     0x0      0
r12     0x0      0
sp      0x7efff770 0x7efff770
lr      0x0      0
pc      0x10064 0x10064 <_start+16>
cpsr    0x10     16
(gdb)

```

Figure 5.9: Step instruction twice and examine the register content

5.9 Continue program execution

When the program execution is stopped, you may continue from where execution was halted by command “continue”.

```
(gdb) c
Continuing.
[Inferior 1 (process 29648) exited normally]
(gdb)
```

Figure 5.10: Continue the program execution and terminated

In this example, as seen in Figure 5.10, if we continue program execution and there are no more breakpoints left, the program will run until termination.

5.10 Examine the memory

The command to examine the memory is “x” followed by options and the starting address. This command has options of length, format, and size (see table 5.10). With the options, the command looks like “x/nfs address”.

Options	Possible values
Number of items	any number
Format	<u>o</u> ctal, <u>h</u> ex, <u>d</u> ecimal, <u>u</u> nsigned decimal, <u>b</u> it, <u>f</u> loat, <u>a</u> ddress, <u>i</u> nstruction, <u>c</u> har, and <u>s</u> tring
Size	<u>b</u> yte, <u>h</u> alfword, <u>w</u> ord, <u>g</u> iant (8-byte)

Table 5.1: Options for examine memory command

For example (Figure 5.11), to display eight words in hexadecimal starting at location 0x10054, the command is “x/8xw 0x10054”.

```
(gdb) x/8xw 0x10054
0x10054 <_start>:    0xe3a01025    0xe3a02034    0xe0823001    0xe3a070
01
0x10064 <_start+16>:  0xef000000    0x00001341    0x61656100    0x010069
62
(gdb)
```

Figure 5.11: Example of examine memory command

Chapter 6

Floating Point

The Raspberry Pi has hardware floating point support (VFP). You may write 32-bit or 64-bit floating point instructions in the assembly program.

To assemble a program with floating point instructions, you need to let the assembler know that you are using the VFP instructions by adding the command line option “-mfpu=vfp”. A sample command will be:

```
$ as -mfpu=vfp -g -o p.o p.s
```

The linker command remains the same.

In GDB, you may examine the VFP registers using command “`info float`”. The registers are displayed in 64-bit mode and 32-bit mode with floating point output and hexadecimal output. The content of the VFP status register `fpscr` is also displayed.

Chapter 7

Program Output

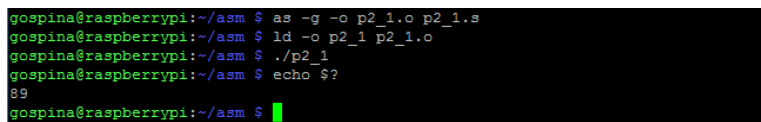
7.1 Program Return Value

Although most of the programs in the book manipulate data internal to the processor and memory, it is still convenient to generate output to be observed outside of the debugger. Recall in section 5 we discussed terminating a program using system call 1. Following the Unix convention, when a program terminates, it should return an exit code. In an assembly program the exit code should be left in Register 0 before the exit system call is made. After the program exit, the user may retrieve the exit code by reading the shell variable “\$?”. This is a simple method for a single integer output of a program.

Using the last sample program, we will send the result of the addition to output. We do so by placing the result in R0 before making the exit system call. The end of the program will look like this:

```
MOV R0, R3
MOV R7, #1
SVC 0
```

After the program exit, at the prompt type command “echo \$?” and the result 89 (decimal) will be displayed (Figure 7.1).



```
gospina@raspberrypi:~/asm $ as -g -o p2_1.o p2_1.s
gospina@raspberrypi:~/asm $ ld -o p2_1 p2_1.o
gospina@raspberrypi:~/asm $ ./p2_1
gospina@raspberrypi:~/asm $ echo $?
89
gospina@raspberrypi:~/asm $
```

Figure 7.1: An example of producing a program output number