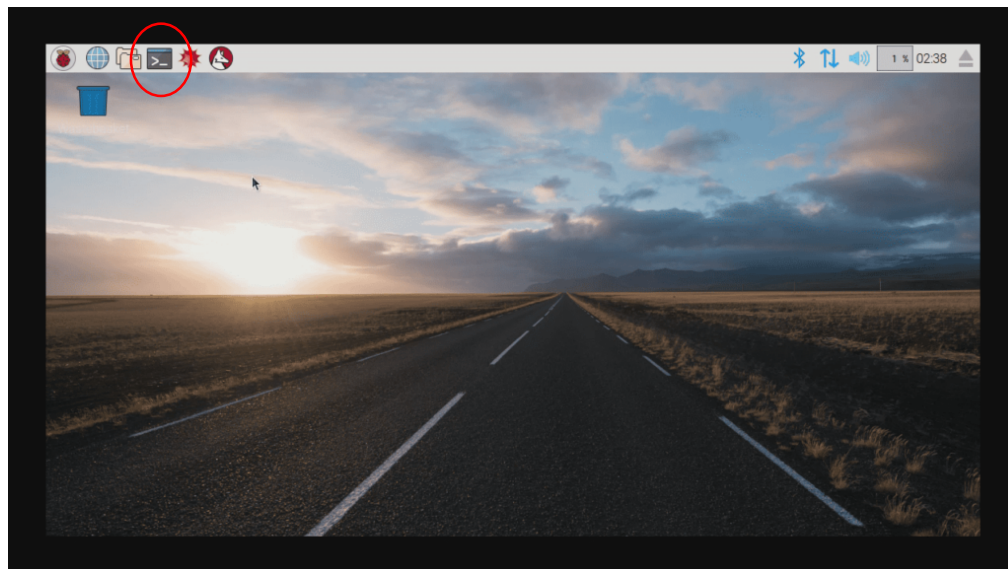


ARM assembler in Raspberry Pi

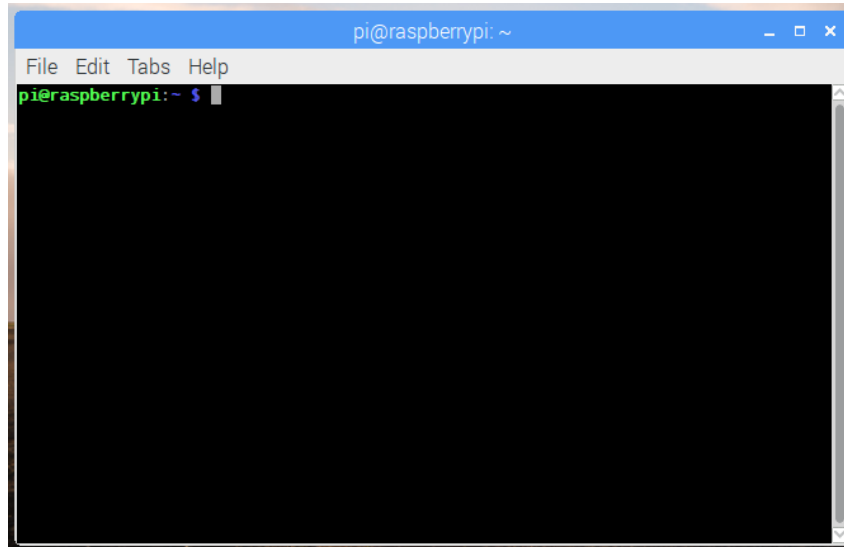
These instructions assume that you have a Raspberry Pi up and running. After starting up your Raspberry Pi, you should have a GUI interface to Raspbian, the Linux variant operating system for these little machines. For this tutorial, you will primarily use the terminal window and type commands into it. You will also use an editor (more on that below).

a) Part1: third program:

1. Open the **Terminal** (command line) in Raspberry Pi, click on the 4th icon to the left on the top bar.



You will see the terminal window as shown below:



2. Type the following in the terminal window, along with the file name:

nano third.s

Note: Control-w writes the file (save); control-x exits the editor.

3. Write the following program using nano and save it (control-w)
See appendix section to see how signed integer is represented

@ Third program

```
.section .data
a: .shalfword -2          @ 16-bit signed integer
```

```
.section .text
.globl _start
_start:
```

@ The following is a simple ARM code example that attempts to load a set of values into registers and it might have problems.

```
mov r0, #0x1           @ = 1
mov r1, #0xFFFFFFFF    @ = -1 (signed)
mov r2, #0xFF           @ = 255
mov r3, #0x101          @ = 257
mov r4, #0x400          @ = 1024
```

```
mov r7, #1             @ Program Termination: exit syscall
svc #0                 @ Program Termination: wake kernel
.end
```

4. To **assemble** the file, type the following command:

Debug your program by doing the following in the terminal window:

Note: A computer without output is not very interesting like the previous example program. For most of the programs in the manipulations of data between CPU registers and memory will be without any output (no use of IO). 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.

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:

as -g -o third.o third.s

- What error messages did you get and why (report that)?
- Correct the program and assemble it using (report the correction)

as -g -o third.o third.s

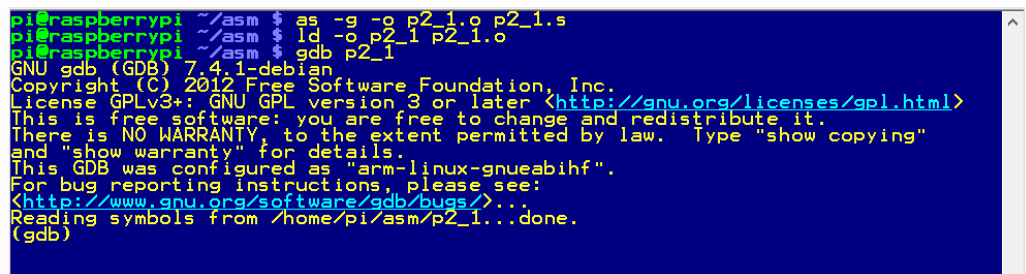
The linker command stays the same:

ld -o third third.o

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

gdb third

After displaying the license and warranty statements, the prompt is shown as (gdb). See the following figure:



```
pi@raspberrypi ~/asm $ as -g -o p2_1.o p2_1.s
pi@raspberrypi ~/asm $ ld -o p2_1 p2_1.o
pi@raspberrypi ~/asm $ gdb p2_1
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 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".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/pi/asm/p2_1...done.
(gdb)
```

- Type “**quit**” if you need to exit GNU Debugger
- 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. To see the next 10 lines, just hit the Enter key.

(gdb) list

- 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 7 of the program. When we run the program, the program execution will stop right before line 7 is executed:

(gdb) b 7

(remember to press enter)

GDB will provide a confirmation of the breakpoint.

- To start the program, use command “**run**”. Program execution will start from the beginning until it hits the breakpoint.

(gdb) run

The line just following where the breakpoint was set will be displayed. Remember, this instruction has not been executed yet.

- 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:

(gdb) stepi

- 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 the following Table: options for examine memory command). 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)

For the example, to display the one halfword in hexadecimal starting at location 0x8054 (replace this memory address with the one shown in your gdb), the command is:

(gdb) x/1xh 0x8054

remember to try sh instead of h, but first try h and see what happens and report that.

Report what you see or observe (include screenshots and snippets)

- b) **Part2:** Using the third.s program as a reference to edit, assemble, link, run, and debug the new program.

Write a program that calculates the following expression:

$$\text{Register} = \text{val2} + 3 + \text{val3} - \text{val1}$$

Assume that val1, val2, and val3 are 8-bit integer memory variables.

- Besides, Val1, val2 are unsigned integers and val3 is signed integer.
- Assign val2=11, val3=16, val1=-60.
- Register could be any of the Arm general purpose registers
- Use the debugger to verify the result in the memories and the Register.
- Report the Register value in hex (as shown in the debugger) and negative flag as shown in gdb

Note: See the following Appendix to see how signed and unsigned integers are defined in ARM and how to report flags

name your program a *arithmetic3.s*

Report what you see or observe (include screenshots and snippets)

Appendix

DATA TYPES

This is part two of the ARM Assembly Basics tutorial series, covering data types and registers. Similar to high level languages, ARM supports operations on different datatypes.

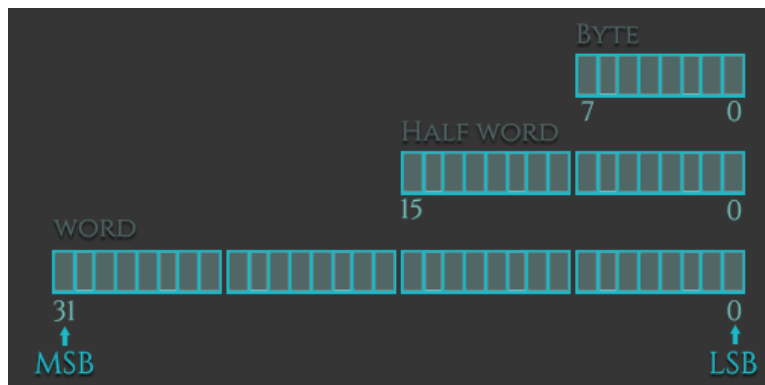
The data types we can load (or store) can be signed and unsigned words, halfwords, or bytes. The extensions for these data types are:

- h or sh for halfwords,
- b or sb for bytes,
- and no extension for words.
- The difference between signed and unsigned data types is:

- Signed data types can hold both positive and negative values and are therefore lower in range.
- Unsigned data types can hold large positive values (including 'Zero') but cannot hold negative values and are therefore wider in range.

Here are some examples of how these data types can be used with the instructions Load and Store:

```
ldr = Load Word
ldrh = Load unsigned Half Word
ldrsh = Load signed Half Word
ldrb = Load unsigned Byte
ldrsh = Load signed Bytes
str = Store Word
strh = Store unsigned Half Word
strsh = Store signed Half Word
strb = Store unsigned Byte
strsb = Store signed Byte
```



CURRENT PROGRAM STATUS REGISTER

When you debug an ARM binary with gdb, you see something called Flags:

```
-----[registers]-----
$r0      0x00000000 $r1      0x00000000 $r2      0x00000000 $r3      0x00000000
$r4      0x00000000 $r5      0x00000000 $r6      0x00000000 $r7      0x00000000
$r8      0x00000000 $r9      0x00000000 $r10     0x00000000 $r11     0x00000000
$r12     0x00000000 $sp     0xbefff7e0 $lr      0x00000000 $pc      0x00008074
$cpsr    0x00000010
Flags: [ thumb fast interrupt overflow carry zero negative ]
```

The register Scpsr shows the value of the Current Program Status Register (CPSR) and under that you can see the Flags thumb, fast, interrupt, overflow, carry, zero, and negative. These flags represent certain bits in the CPSR register and are set according to the value of the CPSR and turn **bold** when activated.

ARM	Description	x86
CPSR	Current Program State Register/Flags	EFLAGS