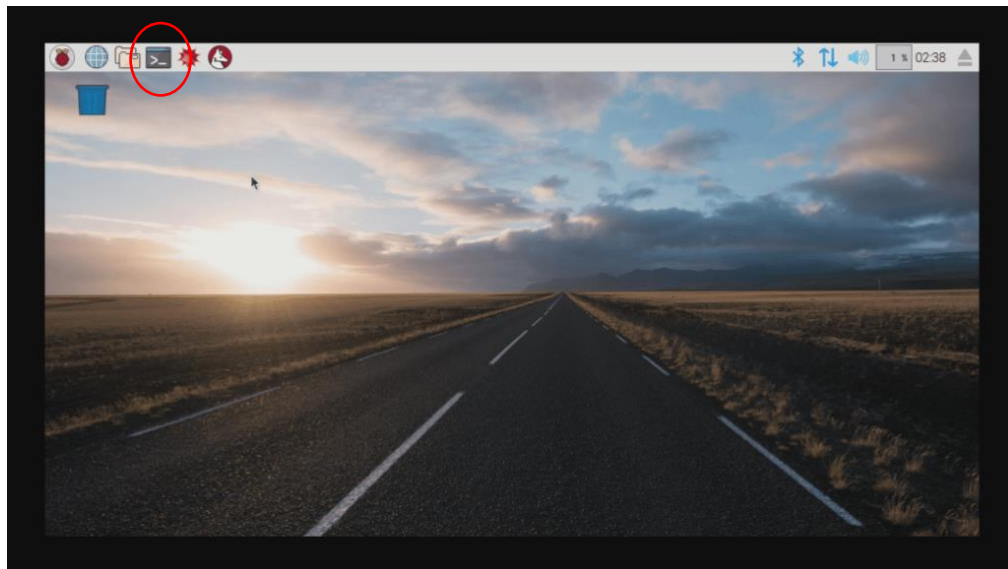


# ARM assembler in Raspberry Pi

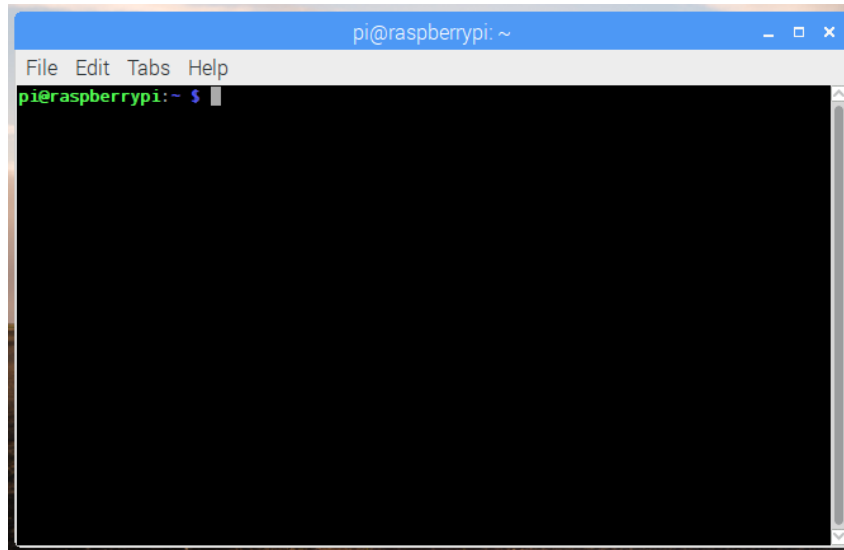
These instructions assume that you have a Raspberry Pi up and running. After starting up your Raspberry Pi, you should have a GUI interface to Raspbian, the Linux variant operating system for these little machines. For this tutorial, you will primarily use the terminal window and type commands into it.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

## a) Part1: Second 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 second.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)

@ second program: **c = a + b**

.section .data

a: .word 2 @ 32-bit variable a in memory

b: .word 5 @ 32-bit variable b in memory

c: .word 0 @ 32-bit variable c in memory

.section .text

.globl \_start

\_start:

ldr r1, =a @ load the memory address of a into r1

ldr r1, [r1] @ load the value a into r1

ldr r2, =b @ load the memory address of b into r2

ldr r2, [r2] @ load the value b into r2

add r1, r1, r2 @ add r1 to r2 and store into r1

ldr r2, =c @ load the memory address of c into r2

str r1, [r2] @ store r1 into memory c

mov r7, #1 @ Program Termination: exit syscall

svc #0 @ Program Termination: wake kernel

.end

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

**as -o second.o second.s**

This will create a **second.o** (objective file)

5. Now **link** this file to get an executable.

**ld -o second second.o**

If everything goes as expected, you will get an executable file (exe).

6. **Run** your program using:

**./ second**

- Did you see any output, why?

7. 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 second.o second.s**

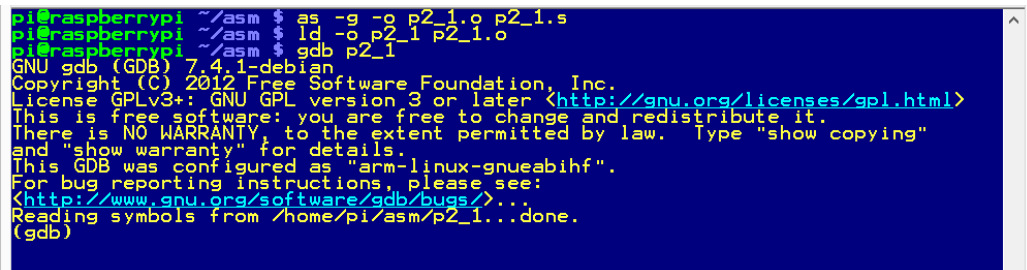
The linker command stays the same:

**ld -o second second.o**

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

**gdb second**

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 15 (str r1, [r2] @ store r1 into c) of the program. When we run the program, the program execution will stop right before line 15 is executed:

**(gdb) b 15**

(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 three words in hexadecimal starting at location 0x8054 (replace this memory address with the one shown in your gdb), the command is:

**(gdb) x/3xw 0x8054**

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

b) **Part2:** Using the second.s program as a reference, and use the same steps to edit, assemble, link, run, and debug the new program.

- Write a program that calculates the following expression:

$$\text{Register} = \text{val2} + 9 + \text{val3} - \text{val1}$$

Assume that val1, val2, and val3 are 32-bit integer memory variables.

- Besides, Val1, val2 and val3 are integers.
- Assign val2=11, val3=16, val1=6.
- 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)

name your program a *arithmetic2.s*

- Assemble, Link, run, and debug the program

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

# Appendix

- as: is a Unix assembler for ARM.
- .global \_start
  - assembler directive
  - \_start is an entry point
- The “pound” or “hash” sign (#) is used to indicate that the following actually a number.
- The @ sign is used to indicate that the following is a comment
- The following table is some of the ARM instructions:

Instruction	Description
mov	Move data
add	Addition
sub	Subtraction
mul	Multiplication
ldr	Load
str	store
svc	System Call

ex. **ldr r0, adr\_var1** @ load the memory address of var1 via label adr\_var1 into R0

A supervisor call (**SVC**) is a processor instruction that directs the processor to pass control of the computer to the operating system's supervisor program.

- The following table is just a quick glimpse into how the ARM registers could relate to those in Intel processors.

ARM	Description	x86
R0	General Purpose	EAX
R1-R5	General Purpose	EBX, ECX, EDX, ESI, EDI
R6-R10	General Purpose	–
R11 (FP)	Frame Pointer	EBP
R12	Intra Procedural Call	–
R13 (SP)	Stack Pointer	ESP
R14 (LR)	Link Register	–
R15 (PC)	<- Program Counter / Instruction Pointer ->	EIP
CPSR	Current Program State Register/Flags	EFLAGS