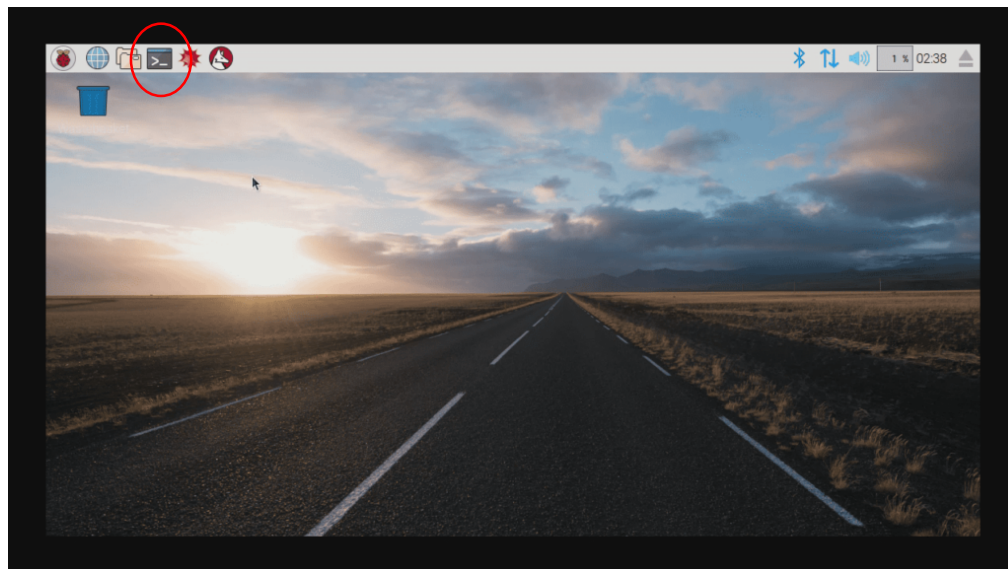


ARM assembler in Raspberry Pi

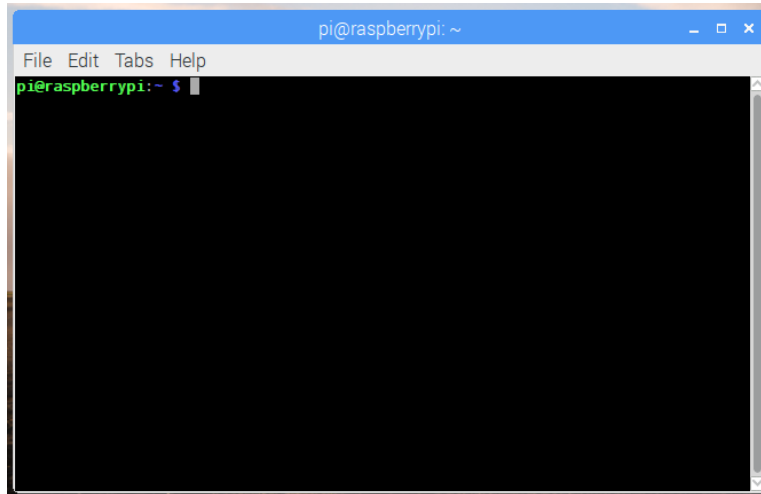
These instructions assume that you have a Raspberry Pi up and running. After starting up your Raspberry Pi, you should have a GUI interface to Raspbian, the Linux variant operating system for these little machines. For this tutorial, you will primarily use the terminal window and type commands into it.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

a) Part1: fourth 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 fourth.s

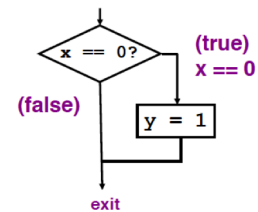
Note: Control-w writes the file (save); control-x exits the editor.

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

@ Fourth program

@ This program compute the following if statement construct:

```
@ intx;
@ inty;
@ if(x == 0)
@   y = 1;
```



```
.section .data
```

```
x: .word 0 @ 32-bit signed integer, you can also use int directive instead of .word directive
```

```
y: .word 0 @ 32-bit signed integer,
```

```
.section .text
```

```
.globl _start
```

```
_start:
```

```
    ldr r1, =x @ load the memory address of x into r1
```

```
    ldr r1, [r1] @ load the value x into r1
```

```
    cmp r1, #0 @
```

```
    beq thenpart @ branch (jump) if true (Z==1) to the thenpart
```

```
    b endofif @ branch (jump) if false to the end of IF statement body (branch always)
```

```
thenpart: mov r2, 1
```

```
    ldr r3, =y @ load the memory address of y into r3
```

```
    ldr r2, [r3] @ load r2 register value into y memory address
```

```
endofif:
```

```
    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 fourth.o fourth.s

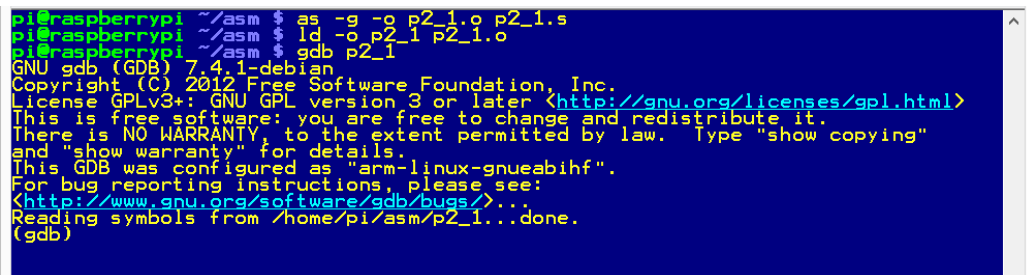
The linker command stays the same:

ld -o fourth fourth.o

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

gdb fourth

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 word in hexadecimal starting at location 0x8054 (replace this memory address with the one shown in your gdb), the command is: (examine y memory location and Z flag value and report their values)

(gdb) x/1xw 0x8054

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

Note: See the Appendix to learn about Control Structures

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

- Is the program in part one efficient?

The answer is no because the code contains back-to-back branches (**beq** followed by **b**). We would like to avoid this, since branches may cause a delay slot.

- What to do?
 - o Replace **beq** with **bnq** (branch on not equal ($Z==0$))
 - o Remove **b** instruction from the code

Note: What you did here is you jumped when the condition was false which is different from what we do in the high level languages. In Other words, we reversed the Boolean expression using DE Morgan Law ($<$ is $>=$, $>$ is $<=$, $==$ is $!=$, $<=$ is $>$ and $>=$ is $<$). I will explain that using X86 if statement in the class.

- Update fourth.s program, assemble , link, run, and debug the updated program
- Examine Z flag value and report its value

c) **Part3:** Using the fourth.s program as a reference to edit, assemble, link, run, and debug the new program.

Write a program that calculates the following expression:

```
if X <= 3
    X = X - 1
else
    X = X - 2
```

Assume that X is 32-bit integer memory variables and assign X 1.

- Use the debugger to verify the result in the memories and the Register.
- Report the X value in hex (as shown in the debugger) and Z flag as shown in gdb

Note: See the Appendix to learn about Control Structures

name your program a ControlStructure1.s

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

Appendix

CURRENT PROGRAM STATUS REGISTER

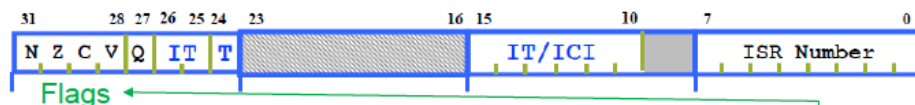
When you debug an ARM binary with gdb, you see something called Flags:

```
-----[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

Program status register (PSR)



- Program Status Register **xPSR** is a composite of 3 PSRs:
 - **APSR** - Application Program Status Register – **ALU condition flags**
 - N (negative), Z (zero), C (carry/borrow), V (2's complement overflow)
 - Flags set by ALU operations; tested by conditional jumps/execution
 - **IPSR** - Interrupt Program Status Register
 - Interrupt/Exception No.
 - **EPSR** - Execution Program Status Register
 - T bit = 1 if CPU in "Thumb mode" (**always for Cortex-M4**), 0 in "ARM mode"
 - IT field – If/Then block information
 - ICI field – Interruptible-Continuable Instruction information
- xPSR stored on the stack on exception entry

ARM Branch Instructions

Unconditional branch

B (or BAL) branch always

Conditional branches

testing result of compare or other operation (signed arithmetic):

beq – branch on equal	$(Z == 1)$
bne – branch on not equal	$(Z == 0)$
bls – branch on less than	$((N \text{ xor } V) == 1)$
ble – branch on less than or equal	$((Z \text{ or } (N \text{ xor } V)) == 1)$
bge – branch on greater than or equal	$((N \text{ xor } V) == 0)$
bgt – branch on greater than	$((Z \text{ or } (N \text{ xor } V)) == 0)$