# Introduction to Assembly using Raspberry Pi’s

### AQA Requirements

Understand and apply the basic machine-code operations of:

|  |  |  |
| --- | --- | --- |
| * load | * logical bitwise operators (AND, OR, NOT, XOR) | * store |
| * add | * branching (conditional and unconditional) | * compare |
| * subtract | * logical: (shift right, shift left) | * halt |

Use the basic machine-code operations above when machine-code instructions are expressed in mnemonic form- assembly language, using immediate and direct addressing

### Aim

The aim of this worksheet is for you to experience programming all of the elements of the AQA requirements through the process of creating assembly programs on the Raspberry Pi. Not everything will be fully explained. You will need to ensure you understand the terminology and ensure you understand the programs you are creating

### Setup

You will be using a Raspberry Pi. Using the terminal window. You will need to know how to navigate linux (the codecademy command line tutorial is ideal). The screenshots will show the use of the “nano “ texteditor ( although you could use “VIM” or even a graphical editor such as “leafpad”).

### Understanding the Assembly process

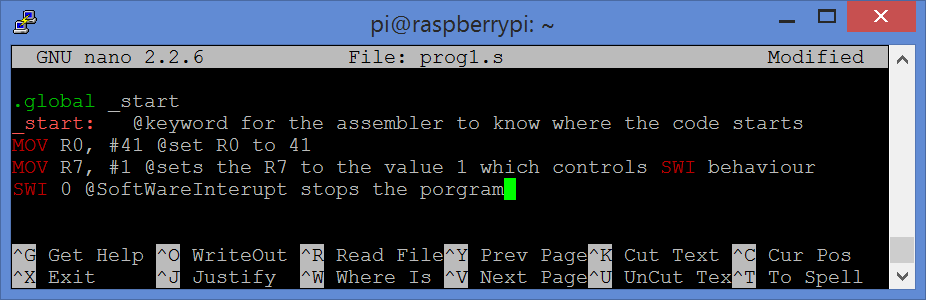
Rather than cut and pasting lots of information that is already in the public domain you should ensure you have read the following:

* <https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/introduction.html>
* AQA Computer Science, Bob Reeves (Hodder) Page281🡪 286

### Task1 Creating your first assembly program.

This will take you through the basics of creating your first program in assembly:

* At the linux command prompt <type> nano. **prog1.s** Enter the following



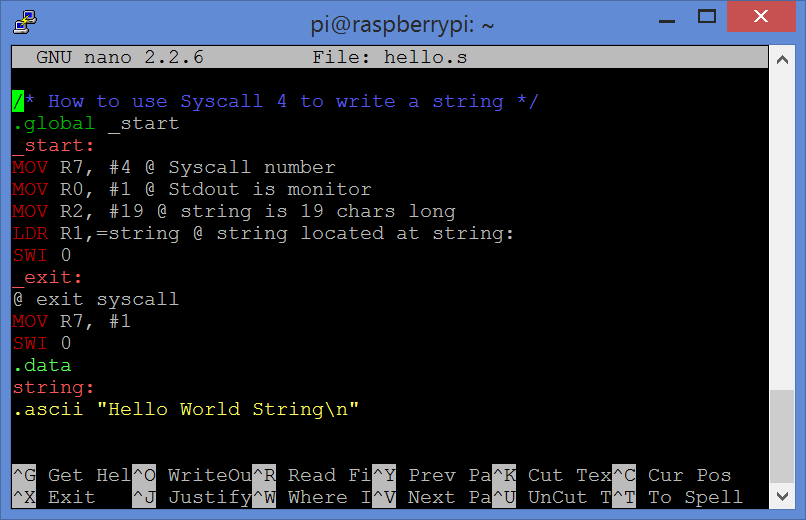
* Ctrl-X to leave nano.. saving the flle as **prog1.s***you are used to software that does all the donkey-work of compiling to machine code for you. as assembles to produce object code.. and ld “links” different parts of a program together for execution. If you find it easier you can use the file suffix .exe (like windows) If you want to know more then we are using the GCC GNU compiler/assembler*
* <type>: **as –o prog1.o prog1.s** (assembles the program)
* <type>: **ld –o prog1 prog1.o** ( this liking stage is only really useful if we have multiple files but is required)
* You have now created an executable file called prog1
* Run this by typing **./prog1; echo $?** (the echo $? Displays the value returned by the swi command)

#### Alternations and questions (don’t change the filenames when altering)

* Change #41 for #65.. (reassemble, and run) what difference does it make?
* What does the # represent?
* Change R0 for R1 (reassemble, and run) what difference does it make?
* What’s the biggest number that can be set to R0?
* What is MOV an acronym for? (what does it do?)
* Change **MOV R0, #2**  to **LDR R0, =2 (**reassemble, and run) .. You should get the same result.
* For a fairly deep discussion on the difference between MOV and LDR see   
  <https://www.raspberrypi.org/forums/viewtopic.php?&t=16528>

### Task 2 Hello World

We need to be a little careful here as we are starting to use all sorts of “features” of the GCC assembler such as named data (basically variables) but… we really need to do that so you understand the use of SWI



* Create the above program, save it as **hello.s** assemble, link and run it.   
  (hopefully “hello world string” appears with the command prompt on the next line)
* Syscall uses specific registers to hold the different values it requires. List what you think the purpose of : R0,R1,R2,R7 are.

#### Alternations and questions

* Change #19 for #11; What Happens? What does the \n do?
* Change the string to one of a different length what happens?
* What do you think operation **LDR R1, =string** does?
* What does the section of code .data do? Make sure you understand every part.

### Task 3 Addition

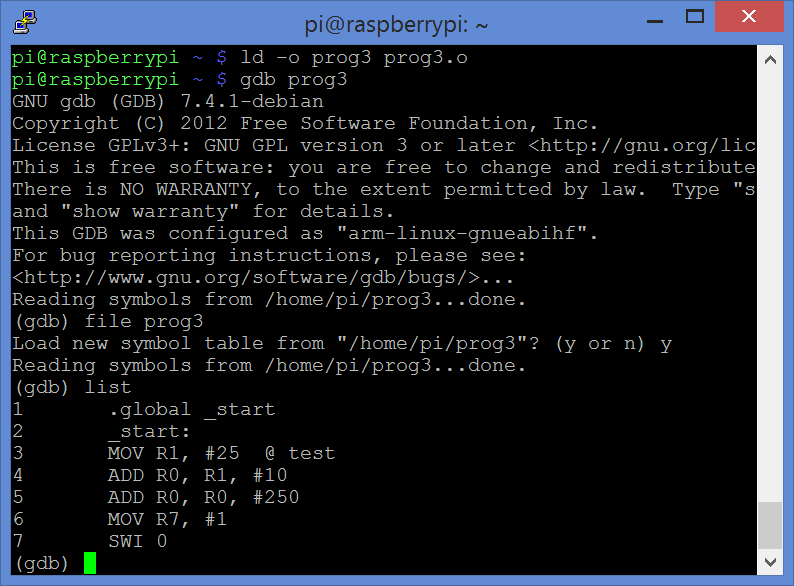
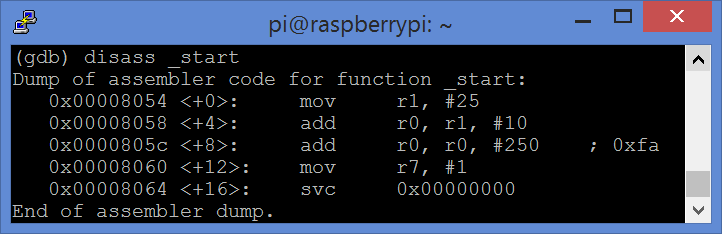
Create the above program and save the file as prog3.s

* Use the @ character and comment each line stating what it does.  
   (use pseudocode)
* Assemble, link and execute the program. Is the output as expected?

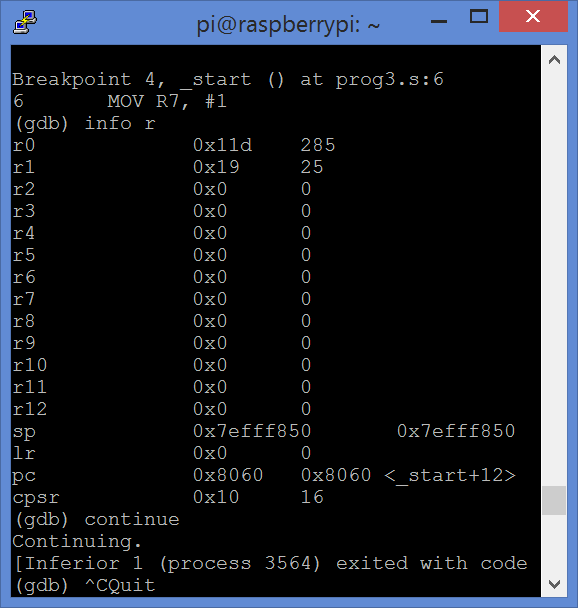
#### Alternations and questions (keep all the alterations under Prog3.s

* Change the program so it adds 15 to 25 (test this)
* Write a second ADD instruction to add a further 30 to R0 (should only take 1 line)
* Change your second ADD so that it now add 250 to R0. (test it)
* Was the output as expected?

### Task 4 Using the debugger

* We are now going to look at prog3 and try different debugging options using GDB.
* We need to store the debugging data:   
  <type>**: as –g –o prog3.o prog3.s**
* Link as usual then <type> **gdb** <enter> [welcome msg appears]  
  the prompt is now (gdb)
* <type> **file prog3…** <enter> [Blah Symbols blah : Y or N] <type> **Y**<enter>
* <type> **list** <enter>
* Now lets Disassemble our program <type> **disass \_start** <enter>   
  *the program is already the focus so we just need to specify \_start*

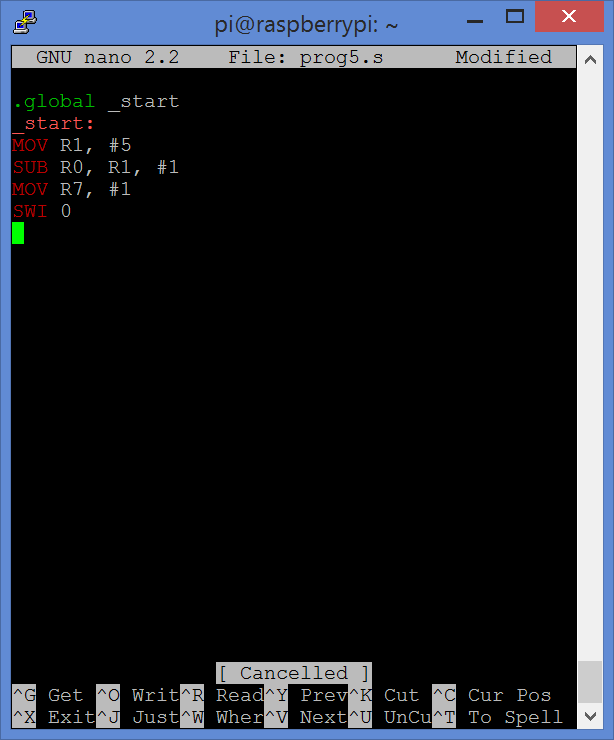
This is taken from memory, the left column is the memory address storing the assembled code.  
*Yours may well be different numbers!*

* In order for us to look at what happens when the code is being execute we need to add a few breakpoints
* <type> **b 4** <enter> (this adds a breakpoint at line 4 (see the list you created))
* <type> **b 6**<enter> <type> **info b** <enter> (you should see the brake positions listed)
* <type> **run** <enter> (runs to the first breakpoint).. Let’s look at the registers
* <type> **info r** <enter> (you should see the register list with hex and decimal values)
* <type> **continue** <enter> (runs to the next breakpoint).. <type> **info r** <enter>
* What’s changed? Does the value of R0 surprise you? (think back to the output)
* <type> **cont** <enter> [msg saying code as executed]
* Quit GDB <type> **quit** <enter>

Use GDB to examine the assembled prog1 examine R0 specifically look at the difference between **MOV R0, #260**  and **LDR R0, =260**

Prog3 and Prog1 are fairly straightforward. Go through a similar process with hello.s . Note the difference in the disassembly to the written program!

### Task 5 Subtracting

* I’m hoping you can guess the output of this program… Write and test it
* Change the program so it subtracts 10 from 5. Test it..
* Surprised? Use GDB to examine the registers just before and then just after the SUB operation has executed.
* What value is R0 holding? Why?
* Note the CPRS register value…
* Change the **SUB** instruction to **SUBS.**
* Test it and examine with GDB as above…
* How has the CPRS register changed? Make a note…
* Going back to Prog3 and trying ADDS instead of ADD should show similar results….. Whilst you don’t need to understand the CPRS in depth.. You ought to know it holds carry bits, flags etc…   
  (see Chpt5 of Bruce Smiths book)

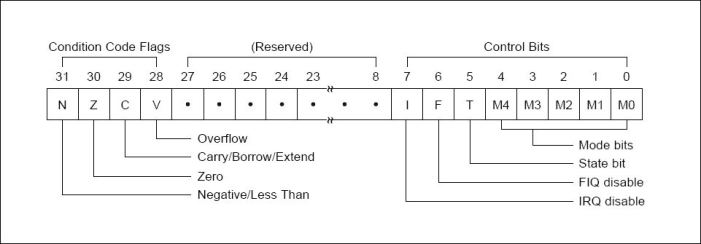
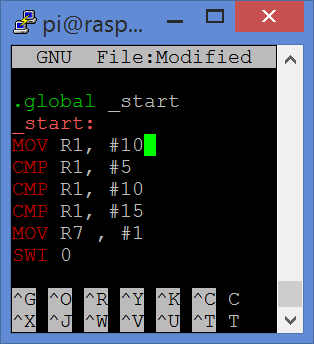


Figure 1 CPSR Layout

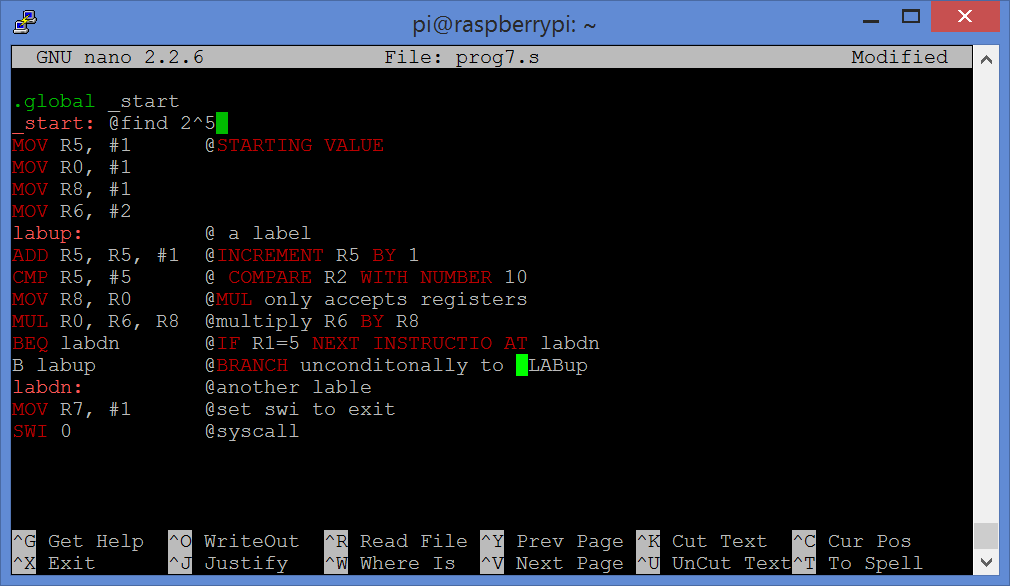
### Task 6 Compare

*  Create Prog6.s
* Assemble with –g option
* Use GDB breakpoints to view the CPSR after each of the compare statement.
* Do any of them hold the same value as the SUBS operation returning a negative number?

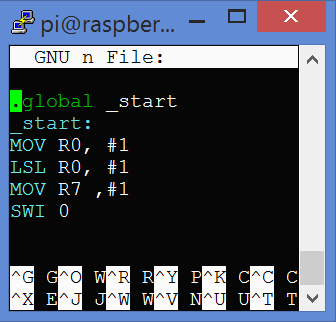
### Task 7 Branching (Specifying the location of the next instruction)

At a low level Iteration and selection look very similar.. both are just types of branching. Branching uses the CPSR (status register) to make the logical decisions. The CPSR is determined by the **LAST**  compare operation

The simplest branch is simply an unconditional branch B which always “jumps” to a label, BEQ jumps to a label IF compare shows two values are equal. This example is effectively a FOR 1to5 loop.

* The program doesn’t quite return a value as stated in the top comment!
* Write this program, test it, was the output as expected?
* Debug with GDB with a breakpoint at line 8.
* Step through the program examining the registers at each step
* Alter the program to find 2^5
* Alter the program to use BNE (no equal) instead of BEQ
* Use this program and Task2 to write a program that outputs “Hello” 10 times

### Task 8 Logical shifts

You have 2 choices… Read about logical shifts OR try the following task and see if you can work it out

* Create this program (prog8),
* Change the immediate value for LSL (Logical Shift Left) to #5; What value do you get?
* Try to create a program (prog8a) that loops through incrementing the   
  Left shift by 1 to 10 . (or cheat get the code here: <http://pastebin.com/JGtZ4HMi>)
* Use the GDB, add a breakpoint so you can see the effect of a LSL on R0
* Now change the initial value R0 holds to 9. Test it, watching R0 as the shift increases.
* What is the equivalent mathematical operation LSL is performing?
* Go back to prog8.s
* Change the initial value of R0 to 172 (128+32+16) now use LRS instead of LSL. Experiment with changing the number of bits to shift. What happens if you perform a shift that “loses” bits off to the right?
* What is the equivalent mathematical operation LSR is performing?

There are other ways to use LSL and LSR. See <http://thinkingeek.com/2013/01/26/arm-assembler-raspberry-pi-chapter-7/> for more shifty fun.

### Reading (treat as sources for the above!)

1. <http://thinkingeek.com/2013/01/09/arm-assembler-raspberry-pi-chapter-1/>
2. Raspberry Pi Assembly Language Raspbian Beginners By Bruce Smith  
   First 20% available here for free <https://www.smashwords.com/books/view/375449>
3. <https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/introduction.html>
4. <http://www.microdigitaled.com/ARM/ASM_ARM/Software/ARM_Assembly_Programming_Using_Raspberry_Pi_GUI.pdf>
5. <http://www.coranac.com/tonc/text/asm.htm>
6. <http://www.nasm.us/doc/> (want to keep going using your PC?)
7. Computer Science (AQA- A-level), Bob Reaves, Hodder Education(2015)
8. <https://letsembed.wordpress.com/category/arm/> (good source of diagrams!)
9. <http://www.davespace.co.uk/arm/>
10. <http://cas.ee.ic.ac.uk/people/gac1/Architecture/> (lecture 8 is particularly interesting)
11. <https://www.raspberrypi.org/forums/viewtopic.php?&t=16528> (The difference between MOV and LDR)