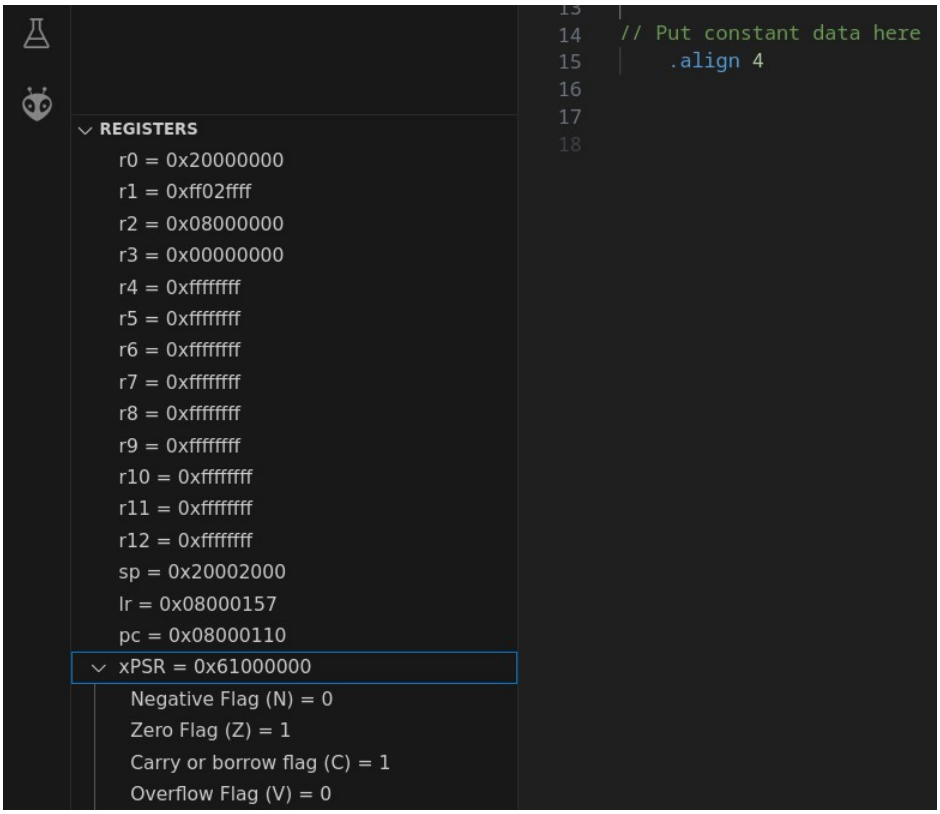
Assembler 1

Foreword

Assembly language is case sensitive in the sense that you must be consistentent about upper and lower case in variable or symbol names. Instructions can be entered in upper or lower case
Labels should go against the left hand margin of the source file and end with a colon
Comments are like C single line comments and start with '//'
Assembler directives control memory allocation and the operation of the assembler during when you program is being generated. They always begin with a full-stop '.'
Common directives:

.text // what folows is allocated in ROM
.data // what follows is allocated in RAM
.word // allocate space for a 32 bit integer
.byte // allocate space for a single byte
.asciiz // allocate space for the string that follows (enclosed in quotes) and insert a null at the
// end
.syntax unified // use the more modern version of assembler for this chip
.global // create an export record in the object file for the symbol that follows. This allows
// other program modules use this symbol
.cpu // specify the actual CPU type
.align // use this to force allocations to align on 2 or 4 byte boundaries. e.g. .align 4

You will run your code differently in this lab. Instead of just doing an upload (and implied 'run'), you will debug your code by single stepping through it line by line. This allows you see effect of your instructions on registers and memory. You can debug your code by choosing 'Start Debugging' from the 'Run' menu. You can examine at register contents and arithmetic flags in the "Registers" window on the left hand side of the VSC user interface as shown below:



Introduction
In this lab you will write and test assembly language code for the STM32 microcontroller. There is a quiz associated with the lab which you can fill in as you go or at the end. The quiz lets us know whether you have completed the lab or not.

There is a template file to help you get statred. Download the template, extract it, and open it in Visual Studio Code/PlatformIO. Modify **main.s** to suit.

Part 1: Arithmetic and bitwise logical operations

C-code

```
R0=1;
R1=2;
R0 = R0 + R1;
```

Assembler-code

```
main:
        movs R0,#1
        movs R1,#2
        adds R0,R0,R1
```

Implement the following:

C-code

```
R0=1;
R1=2;
R0 = R0 - R1;
```

C-code

```
R0=1;
R1=2;
R0 = R0 & R1;
```

C-code

```
R0=1;
R1=2;
R0 = R0 * R1;
```

C-code

```
R0=1;
R1=2;
R0 = R0 ^ R1;
```

Now try quiz questions 1 to 4.

When translating from C to assembler it can be useful to rename C variables in a way that makes them look like CPU register. For example:

int X; would become int R0;
int *Y; would become int *R3;
This can help you translate C to assembler.

Part 2: Addressing modes.

You will explore addressing modes in this section. An addressing mode describes how data is moved into or out of a register. We used immediate addressing in Part 1 (the # symbol indicates immediate addressing mode).

Register indirect addressing example

C-code

```
int ary[]={1,2,3,4,5};
R0 = ary;
R1 = *R0;
```

Assembler-code

```
        .data
ary .word 1,2,3,4,5
        .text
        :
main:   ldr R0,=ary
        ldr R1,[R0]
```

Now try to translate the following C code fragment into assembler:

C-code

```
const int src[]={1,2,3,4};
int dest[4];
        dest[0]=src[0];
        dest[1]=src[1];
        dest[2]=src[2];
        dest[3]=src[3];
```

Note the use of the 'const' keyword. This means that the array 'src' should be allocated in ROM. You can do this assembler by locating it in the .text section of memory. Variables go in the .data section of memory.