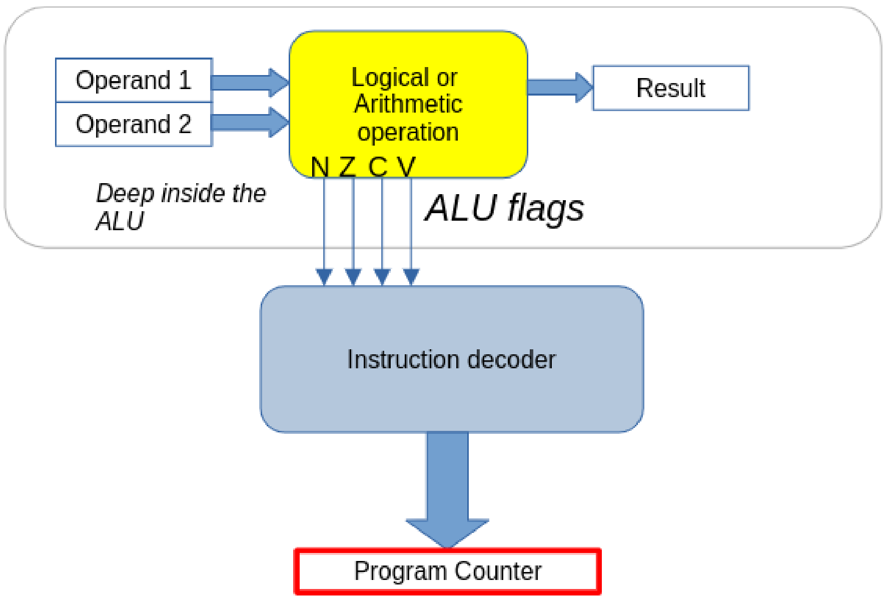
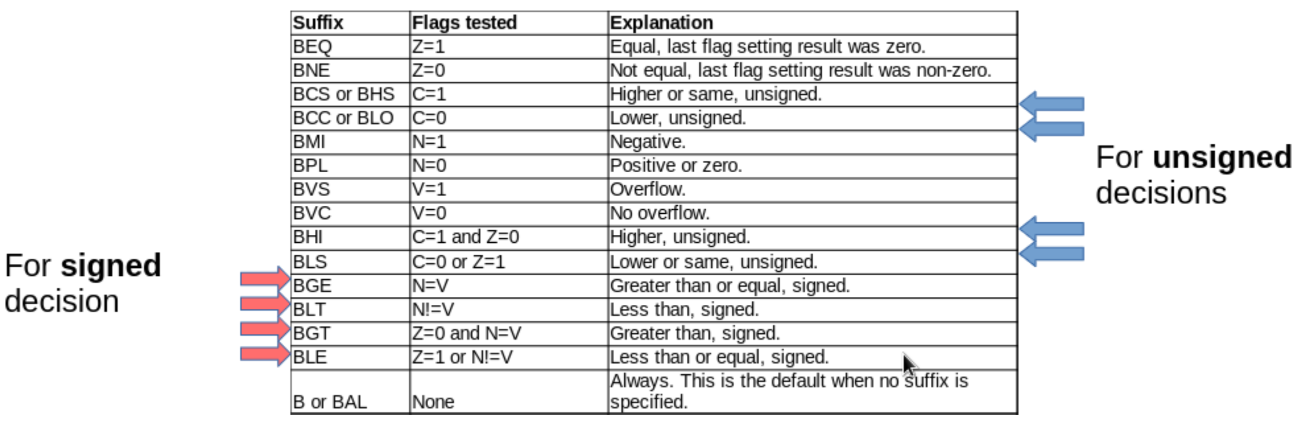
Assembler lab 2. Decision making.



The ALU in a microprocessor core works with the instruction decoder to support the decision making process. The ALU is used to compare values and based on this, programs will execute code in one branch or another. The link between the ALU and the instruction decoder is a set of 4 signals called “**flags**”. These flags can each have a value of 1 or 0. The four flags are:
Zero : The result of the last operation in the ALU was zero
Negative : The result of the last operation in the ALU was negative
Carry: The result of the last operation in the ALU was too big for its 32 bit result register
o**V**erflow: The last operation in the ALU resulted in a signed overflow.

The individual and combined states of these flags are used to make decisions using conditional branch instructions. These instructions are:

List of conditional branch instructions



Suffix	Flags tested	Explanation
BEQ	Z=1	Equal, last flag setting result was zero.
BNE	Z=0	Not equal, last flag setting result was non-zero.
BCS or BHS	C=1	Higher or same, unsigned.
BCC or BLO	C=0	Lower, unsigned.
BMI	N=1	Negative.
BPL	N=0	Positive or zero.
BVS	V=1	Overflow.
BVC	V=0	No overflow.
BHI	C=1 and Z=0	Higher, unsigned.
BLS	C=0 or Z=1	Lower or same, unsigned.
BGE	N=V	Greater than or equal, signed.
BLT	N!=V	Less than, signed.
BGT	Z=0 and N=V	Greater than, signed.
BLE	Z=1 or N!=V	Less than or equal, signed.
B or BAL	None	Always. This is the default when no suffix is specified.

The ALU is only connected to the register set in the CPU. You can not compare values that are in memory. With one exception, these values must be loaded into registers and then compared. The exception case is when you are comparing a register with an immediately addressed 8 bit value.

Example translation from C to assembler:

C code
```
if (R0 == 1)
{
// R0 is 1
        R1 = 2;
}
else
{
// R0 is not 1
        R1 = 4;
}
```

Assembler code
```
        CMP R0,#1
        BEQ R0_is_1
R0_is_not_1:
        MOVS R1,#4
        B exit
R0_is_1:
        MOVS R1,#2

exit:
```

A pair of instructions is used to implement the if statement:
CMP R0,#1 : This sends the contents of R0 and the value 1 to the ALU. The ALU performs a subtraction operation but discards the result. The ALU flags are changed appropriately and sent to the instruction decoder. If R0 did indeed contain a 1 then the Zero flag will be set to 1 to indicate that there was a zero result from the comparison.
BEQ R0_is_1: This instruction checks the Z flag. If it is 1 then the program branches to the address labelled R0_is_1. If Z is not 1 then the branch is not taken and the instruction that follows BEQ in memory is executed instead. The "else" is implied rather than explicitly coded.

Consider the following C code:

```
count=10;
total=0;
while(count != 0)
{
        total = total + count;
        count = count - 1;
}
```

After this code is done, the value in 'total' will be the sum of all numbers between 10 and 0. To implement this in assembler, we could begin by renaming all the variables as if they were registers:

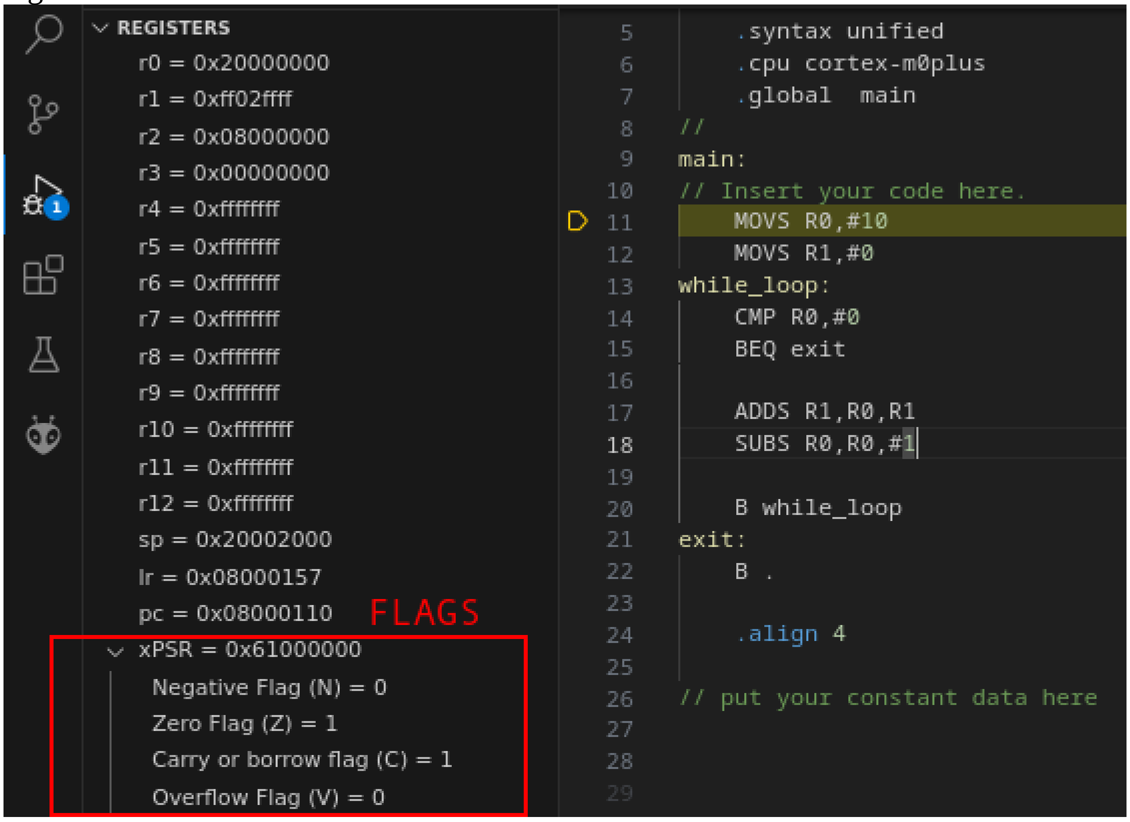
```
R0=10;
R1=0;
while(R0 != 0)
{
        R1 = R1 + R0;
        R0 = R0 - 1;
}
```

We can then translate the C code to assembler (shown in the C comments below)

```
R0=10;  // MOVS R0,#10
R1=0;   // MOVS R1,#0
while_loop:
while(R0 != 0) // CMP R0,#0
               // BEQ exit
{
        R1 = R1 + R0; // ADDS R1,R0,R1
        R0 = R0 – 1; // SUBS R0,R0,#1

}       // B while_loop
exit:
 // B .
```

Task 1: Take the assembler from the comments above and implement the program in assembler in visual studio code. Test the code (using the debugger) on the development board. Be sure to note the state of the flags just after CMP is executed each pass through the loop. The flags can be found in the registers as shown here:



```
REGISTERS
  r0 = 0x20000000
  r1 = 0xff02ffff
  r2 = 0x08000000
  r3 = 0x00000000
  r4 = 0xffffffff
  r5 = 0xffffffff
  r6 = 0xffffffff
  r7 = 0xffffffff
  r8 = 0xffffffff
  r9 = 0xffffffff
  r10 = 0xffffffff
  r11 = 0xffffffff
  r12 = 0xffffffff
  sp = 0x20002000
  lr = 0x08000157
  pc = 0x08000110
  xPSR = 0x61000000
    Negative Flag (N) = 0
    Zero Flag (Z) = 1
    Carry or borrow flag (C) = 1
    Overflow Flag (V) = 0
```

```
        .syntax unified
        .cpu cortex-m0plus
        .global  main
//
main:
// Insert your code here.
        MOVS R0,#10
        MOVS R1,#0
while_loop:
        CMP R0,#0
        BEQ exit

        ADDS R1,R0,R1
        SUBS R0,R0,#1

        B while_loop
exit:
        B .

        .align 4

// put your constant data here
```

You should note that on the last pass through the loop, the Z flags will be 1 and so the conditional branch BEQ will indeed happen.

Task 2: Translate the following C code snippets into assembler. Run the code through the debugger and note the behaviour of the flags and conditional branch instructions.

(1)

```
total=0;
for (count=0; count< 11; count++)
{
        total = total + count;
}
```

(2)

```
uint32_t x=0xffffffff; // 8 f's
uint32_t y=1;
if (x > y)
{
        y = 0;
}
```

Note: This uses **unsigned** data

(3)

```
int32_t x=0xffffffff; // 8 f's
int32_t y=1;
if (x > y)
{
        y = 0;
}
```

Note: This uses **signed** data

Task 3:
The STM32L031/F031 is a 32 bit CPU. Can you use the flags in such a way that it can correctly calculate 64 bit additions and subtractions? (Hint: Look at the ADCS and SBCS instructions)