Lab work #1

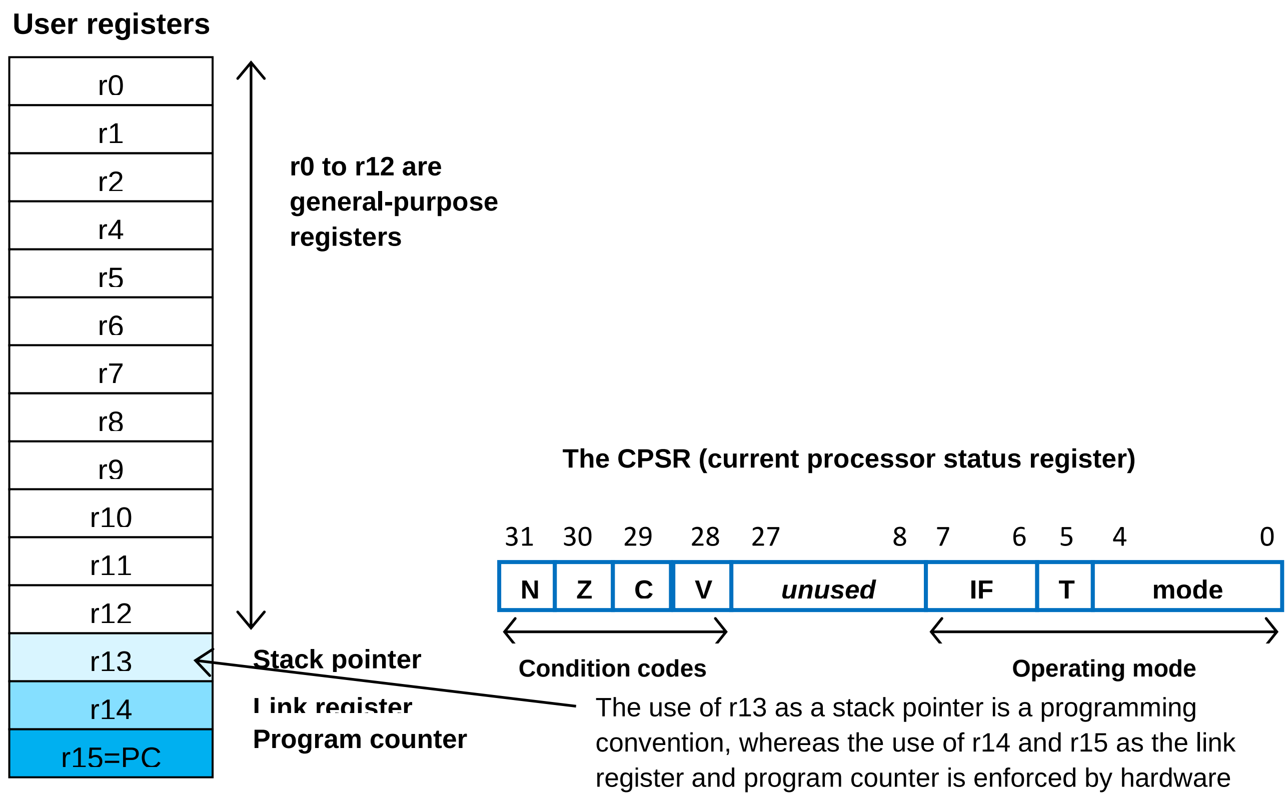
1. **Objective of this lab:** To investigate ***Arithmetic Operations*** and implement them in ARM uVision5

2. **Preparation**

2.1. ARM Data-processing Instructions

ARM Registers and the Conventions of Use

As mentioned in the previous assignment, ARM has 16 programmer-visible registers and a *Current Program Status Register*, CPSR.
Here is a picture to show the **ARM register set**.




```
R0 to R12 are the general-purpose registers.
R13 is reserved for the programmer to use it as the stack pointer.
R14 is the link register, which stores a subroutine return address.
R15 contains the program counter and is accessible by the programmer.

            CPSR: current program status register (32 bit)
            Stores the status of the previous ALU operation
            4 flags
            N = 1, if result was negative
            Z = 1, if result was zero
            C = 1, if result had a carry-out
```

Each ARM instruction is encoded into a 32-bit word. Access to memory is provided only by `Load` and `Store` instructions.
ARM data-processing instructions operate on data and produce new value. They are not like the branch instructions that control the operation of the processor and sequencing of instructions.
ARM instructions have the following general format:

```
Label Op-code operand1, operand2, operand3    ; comment
```

Arithmetic Instructions

Arithmetic instructions are very basic and frequently used in your ARM programming. Here is a table that demonstrates the usage of the ARM processor's arithmetic instructions with examples.

Instruction	Mnemonic	Meaning
Addition	`ADD   R0, R1, R2`	R0 = R1 + R2
Addition	`ADDS  R1, R2, R3`	R1 = R2 + R3, and FLAGs are updated
Subtraction	`SUB   R0, R1, R2`	R0 = R1 - R2
Subtraction	`SUBS  R1, R2, R3`	R1 = R2 - R3, and FLAGs are updated
	`SUBS  R7, R6, #20`	R7 = R6 – 20 Sets the flags on the result
Reverse Subtraction	`RSB R4, R4, #120`	R4 = 120 - R4
Multiply	`MUL   R0, R1, R2`	R0 = R1 * R2
	`UMULL R0, R4, R5, R6`	Unsigned (R4,R0) = R5 * R6
	`SMLAL R4, R5, R3, R8`	Signed (R5,R4) = (R5,R4) + R3 * R8
Division	`SDIV  R0, R2, R4`	Signed divide, R0 = R2/R4
	`UDIV  R8, R8, R1`	Unsigned divide, R8 = R8/R1.

Examples of Move Instructions

Mnemonic	Meaning
`MOVS R11, #0x000B`	Write value of 0x000B to R11, flags get updated
`MOV   R1, #0xFA05`	Write value of 0xFA05 to R1, flags are not updated
`MOVS R10, R12`	Write value in R12 to R10, flags get updated
`MOV R3, #23`	Write value of 23 to R3
`MOV R8, SP`	Write value of stack pointer to R8
`MVNS R2, #0xF`	Write value of 0xFFFFFFF0 (bitwise inverse of 0xF) to the R2 and update flags.

Logical Operation Instructions

`AND R9, R2, R1`	`; R9 = R2 AND R1`
`AND R9, R2, #0xFF00`	`; R9 = R2 AND #0xFF00`
`ORR R9, R2, R1`	`; R9 = R2 OR R1`
`ORR R9, R2, #0xFF00`	
`ORREQ R2, R0, R5`	
`ANDS R9, R8, #0x19`	
`EOR  R7, R11, R10`	`; R7 = R11 XOR R10`
`BIC R0, R1, #0xab`	`; R0 = R1 AND (NOT(#0xab))`
`ORN R7, R11, R14, ROR #4`	`; R7 = R11 OR (NOT(R14 ROR #4))`
`ORNS R7, R11, R14, ROR #2`	`; update the flags`

Conditional Execution of Instructions

Each ARM instruction is encoded into a 32-bit word.
The basic encoding format for the instructions such as
Load, Store, Move, Arithmetic, and Logic instructions, is as follows:

31 – 28	27 – 20	19 – 16	15 – 12	11 – 4	3 – 0
Condition	**OP code**	**Rn**	**Rd**	**Other info**	**Rm**

An instruction specifies a conditional execution code (Condition), the OP code, two or three registers (Rn, Rd, and Rm), and some other information.
Here is a more detailed description.

31-28	27	26	25	24-21	20	19-16	15-12	11-0
Condition	**0**	**0**	**I**	**OP code**	**S**	**Rn**	**Rd**	**Operand 2**

- **Rn** = source register operand 1
- **Rd** = destination register
- **31-28: condition code**
  - ALL ARM instructions can be conditionally executed
  - eg: `ADDEQ`
    - add, but only if the previous operation produced a result of zero
    - checks CPSR stored from previous operation.

All the ARM instructions are conditionally executed depending on a condition specified in the instruction (bits 31-28).

CONDITION		**FLAGS**	**Note**
0000	EQ	Z==1	Equal
0001	NE	Z==0	Not Equal
0010	HS/CS	C==1	>= (U) / C=1
0011	LO/CC	C==0	< (U) / C=0
0100	MI	N==1	minus (neg)
0101	PL	N==0	plus (pos)
0110	VS	V==1	V set (ovfl)
0111	VC	V==0	V clr
1000	HI	C==1 && Z==0	> (U)
1001	LS	C==0 || Z==1	<= (U)
1010	GE	N==V	>=
1011	LT	N!=V	<
1100	GT	Z==0 && N==V	>
1101	LE	Z==1 || N!=V	<=
1110	AL	always	
1111	NE	never	
(U) = unsigned			

The instruction is executed only if the current state of the processor condition code flag satisfies the condition specified in bits `b31-b28` of the instruction.

```
For example:
            CMP R0, #25              ; flags are updated according to (R0 - #25)
            ADDGT R1, R2, #12
```

The instructions whose condition does not meet the processor condition code flag are not executed.
One of the conditions is used to indicate that the instruction is always executed.

Examples of Shift Instructions

`LSLS R1,R2,#3`	; Logical shift left by 3 bits with flag update
`LSR R4,R5,#6`	; Logical shift right by 6 bits
`LSL R4,R5,#6`	; Logical shift left by 6 bits
`ROR R4,R5,R6`	; Rotate right by the value in bottom byte of R6
`RRX R4,R5`	; Rotate right with extend (one bit only)

An Example of Using Arithmetic Instructions

```
; The semicolon is used to lead an inline documentation
;
; When you write your program, you could have your info at the top document block
; For Example:  Your Name, Student Number, what the program is for, and what it does etc.
;
; This program will calculate the value of the following function:
;      f(x) = 5x^2 - 6x + 8  when x = 7
;
            AREA  MYCODE, CODE, READONLY
            ENTRY

            MOV R0, #7          ; x = 7
            MUL R1, R0, R0      ; R1 = x^2
            MOV R4, #5
            MUL R1, R1, R4

            MOV R5, #6
            MUL R2, R0, R5      ; R2 = 6x
            SUB R3, R1, R2      ; R3 = 5x^2 - 6x
            ADD R3, R3, #8      ; R3 = 5x^2 - 6x + 8

            ALIGN
STOP        B   STOP
            END                 ; End of the program
```

Load and Store Instructions

To access memory, we can only use Load and Store instructions.

```
        LDR dest, expression;
e.g.    LDR R0, NUM         ; load R0 with the value of NUM in memory

        STR STR{cond} srce,[base],offset

e.g.    STR R0,[R1]         ; store R0 in the byte address R1
e.g.    STR R0,[R1,#20]           ; store R0 in the byte address R1+20
e.g.    STR R0,[R1,R2,LSL#2]      ; store R0 in the address given by R1+R2*4

Example:
        LDR    R6, = NUM          ; Load the address of NUM to R0
        MOV    R0, #0x001C        ; Load the value to the R0
        STR    R0, [R6]           ; Store the value in R0 to NUM
```

3. Lab Assignment

3.1. Assignment #1:

You can convert temperatures from Celsius to Fahrenheit or from Fahrenheit to Celsius. Here are the two formulas for your reference.

$$C = 5 * (F - 32) / 9$$
$$F = (9 * C / 5) + 32$$

Write an ARM assembly language program **convertF2CandC2F.s**.
You will do the following:

1. You can put the Fahrenheit temperature, say 85, in the register R0;
2. and have the converted temperature in Celsius in the register R1.
3. You can put a Celsius temperature, say 32 in register R2;

4. and have the converted temperature in Fahrenheit in the register R3.
5. Build the program if there are any bugs, fix them.
6. Run the program step by step and see how values are changing in the registers.
7. Make a screenshot to capture the results in your designated registers.

You will hand in the following:

1. The source code in the file **convertF2CandC2F.s**
2. The print out of the screen shot (print screen) to show the program has been successfully built
3. The print out of the screen shot showing the converted temperatures in the registers

3.2. Assignment #2:

Write an ARM assembly language program to solve the following problem.

- For a set of numbers stored in an array, calculate the sum of the positive numbers and the sum of the negative numbers.
- The program should calculate both sums and store them to memory. Numbers in the array with have a zero value (0) being used to signal the end of data (the zero value is acting as a "`sentinel`" value).

For example, your .data section for the array values: 20_{10} -8_{10} -35_{10} 10_{10} 15_{10} -7_{10} 0_{10} will be:

```
        AREA SUM_POS_NEG, CODE, READONLY
        ENTRY

        ; add your code here

        STOP B STOP

        AREA SUM_POS_NEG, DATA, READWRITE
        ALIGN
NUMBERS DCD 20, -8, -35, 10, 15, -7, 0
POS_SUM DCD 0
NEG_SUM DCD 0
        END
```

You should turn in:

1. a printout of the ARM assembly language program file from inside the Keil uVision 5 IDE
2. a window capture of the simulator state after running your assembly language program with array values: 20_{10} -8_{10} -35_{10} 10_{10} 15_{10} -7_{10} 0_{10} and the POS_SUM result showing 45_{10} and NEG_SUM result showing -50_{10}. You can capture this window by (1) right-clicking anywhere in the window to make it the "currently active" window, (2) while holding down the <Alt> key, press the <PrtSc> key to capture the window into the Window's clipboard, and (3) open some word processor (Word, Open Office, etc.) and paste the image into the document. Add your name to this document.