1. Modify function `SIMD_Find4Mins` (Listing 11-2) so that it can be used to find the minimum among an array of 16-bit signed integers. Write a C program to test your function.

```
// int32_t SIMD_Find4Mins(int16_t a[], int32_t n) ;

SIMD_Find4Mins:
        LDR     R2,[R0],4   // Load initial minimums
loop:   SUBS    R1,R1,2     // Decrement count, bump adrs
        BEQ     done        // Test for completion
        LDR     R3,[R0],4   // Get next pair of 16-bit halfwords
        SSUB16  R12,R3,R2   // Compare to minimums
        SEL     R2,R2,R3    // Select the minimums
        B       loop        // Repeat
done:   MOV     R0,R2       // Copy minimums to R0
        BX      LR          // Return
```

3. Write a function in assembly language called `Slow_USatAdd` like function SIMD_`USatAdd` (Listing 11-5) that adds a constant to each byte of a one-dimensional array of bytes with saturation. However, do not use any of the saturating instructions of the processor in your solution.

```
// void Slow_USatAdd
        (uint8_t bytes[], uint32_t count, uint8_t amount) ;

Slow_USatAdd:
        BFI      R2,R2,8,8      // Two 8-bit copies of amount
        BFI      R2,R2,16,16    // Four 8-bit copies of amount
loop:   CBZ      R1,done        // Test for completion (count = 0)
        LDRB     R3,[R0]        // Get next four bytes
        UQADD8   R3,R3,R2       // Add amount to all four
        ADDS     R3,R3,R2       // Add amount to the byte
        CMP      R3,255         // Unsigned overflow?
        BLS      L1             // No overflow – don’t replace sum
        LDR      R3,=255        // Yes: Replace sum by max value
L1:     STRB     R3,[R0],1      // Store the results, bump adrs
        SUB      R1,R1,1        // decrement the count by 1
        B        loop           // repeat until done
done:   BX       LR             // return
```

Problem 4 asks you to create functions in assembly to perform addition, subtraction, multiplication and division of complex numbers. Since C doesn't natively support complex numbers, it suggests using a 64-bit integer as a container to hold two 32-bit floats -one for the real part and one for the imaginary part of a complex number, as in:

    typedef uint64_t COMPLEX ;

However, this means that when you pass a complex number to a function, the compiler really thinks it is passing a 64-bit integer and thus does so using a pair of integer registers like R0 and R1. It also means that functions that return a complex number must leave the result in R0 and R1. That wouldn't be so bad except that it would require using VMOV instructions to copy the values into floating-point registers in order to do any floating-point arithmetic, and then using VMOV instructions again at the end to copy the floating-point result back into R0 and R1.

A better way is to define COMPLEX as:

    typedef double COMPLEX ;

That way, the compiler will use a pair of floating-point registers to pass a complex parameter and the function result can be left in S0 and S1. I.e., there is no need for any VMOV instructions!

So, your assignment is to do problem 4, but use double instead of uint64_t as the container for COMPLEX.

```
// typedef double COMPLEX ;
// COMPLEX AddComplex(COMPLEX a, COMPLEX b) ;

// Parameters of all functions:
// S0 = real part of a, S1 = imaginary part of a
// S2 = real part of b, S3 = imaginary part of b
```

// $a + b = (x_a + x_b) + (y_a + y_b)i$, x=real part, y = imag part

```
AddComplex:
    VADD.F32    S0,S0,S2      // Add the real parts
    VADD.F32    S1,S1,S3      // Add the imaginary parts
    BX          LR
```

// $a - b = (x_a - x_b) + (y_a - y_b)i$, x=real part, y = imag part

```
SubComplex:
    VSUB.F32    S0,S0,S2      // Add the real parts
    VSUB.F32    S1,S1,S3      // Add the imaginary parts
    BX          LR
```

```
// a × b = (x_a x_b − y_a y_b) + (x_a y_b + x_b y_a)i,   x=real part, y = imag part
MulComplex:
    VMUL.F32     S4,S0,S2       // Multiply the real parts
    VMLS.F32     S4,S1,S3       // Multiply the imaginary parts
    VMUL.F32     S1,S0,S3       // Cross-multiply #1
    VMLA.F32     S1,S1,S2       // Cross-multiply #2
    VMOV         S0,S4          // copy real part to S0
    BX           LR
```

$// \ a \times b = (x_a x_b - y_a y_b) + (x_a y_b + x_b y_a)i$, x=real part, y = imag part

$// \ a \div b = \frac{(x_a x_b + y_a y_b) + (x_b y_a - x_a y_b)i}{x_b^2 + y_b^2}$, x=real part, y = imag part

```
DivComplex:
    VMUL.F32     S4,S0,S2       // real part of numerator (top)
    VMLA.F32     S4,S1,S3
    VMUL.F32     S5,S2,S1       // imag part of numerator
    VMLS.F32     S5,S0,S3
    VMUL.F32     S6,S2,S2       // denominator (btm)
    VMLA.F32     S6,S3,S3
    VDIV.F32     S0,S4,S6       // real part of result
    VDIV.F32     S1,S5,S6       // imag part of result
    BX           LR
```