



RTS10

Real-Time Systems

Assembly Assignment

Version 1.2

J.Z.M. Broeders

Version History

Date	Version	Description	Author
20-07-2026 ¹	1.2 ²	Moved to GitHub.	BroJZ
03-10-2024	1.1	Clarified some assignments.	BroJZ
22-08-2023	1.0b	Corrected some typos. Replaced MS Teams with Brightspace.	BroJZ
25-06-2022	1.0	Initial version for LEGv7 Pinky.	BroJZ



Assembly Assignment Real-Time Systems from Rotterdam University of Applied Sciences is licensed by a [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Netherlands license](#).

¹ Dates are formatted in the Gregorian way (*dd-mm-yyyy*).

² Explanation version coding *A.Bc*: *A* = major change, *B* = minor change, *c* = linguistic or mathematical corrections.

Contents

1	Introduction	4
2	Assignments	6
2.1	Assignment 1. Testing a simple LEGv7 Pinky program	6
2.2	Assignment 2. Writing a simple multiplication program	9
2.3	Assignment 3. A smarter multiply algorithm	11
2.4	Assignment 4. A smarter, recursive multiply algorithm	12
2.5	Assignment 5. A smarter, tail recursive multiply algorithm	13
2.6	Assignment 6. Using your multiply function to calculate a power	14
2.7	Assignment 7. A smarter power algorithm	15
2.8	Assignment 8. A smarter, recursive power algorithm	16
2.9	Assignment 9. A smarter, tail recursive power algorithm	17
2.10	Assignment 10. Using your multiply function to calculate the dot product of two vectors	18
2.11	Assignment 11. Using your multiply function to calculate a square root	20
2.12	Assignment 12. Sorting an array of 32-bit integers by using insertion sort . .	23
2.13	Assignment 13. Sorting an array of 32-bit integers by using selection sort . .	25
2.14	Assignment 14. Sorting an array of 32-bit integers by using cocktail shaker sort	27
2.15	Assignment 15. Sorting an array of 32-bit integers by using gnome sort	29
2.16	Assignment 16. Sorting an array of 32-bit integers by using stooge sort	31
2.17	Assignment 17. Sorting an array of 32-bit integers by using slow sort	33
2.18	Assignment 18. Sorting an array of 32-bit integers by using optimized bubble sort	35
	Bibliography	37

1

Introduction

To fully understand how a computer program runs on computer hardware it is required to have knowledge, not only about the computer hardware, but also on how to write machine language instructions that will be executed on the hardware.

Because we explain the concepts of computer architecture by looking at the ARM Cortex M4 processor [4] (which is especially suitable for this purpose) we will also write assembly programs for this processor. The ARM Cortex M4 uses the ARMv7 Thumb instruction set [1] which consists of 16-bits and 32-bits instructions. This instruction set is quite big [3] and contains about 280 instructions. Therefore, we have defined a small subset which we call LEGv7 Pinky [2] which only consists of thirty-three 16-bit instructions. Because the LEGv7 architecture is a subset of the ARMv7 architecture and the Pinky instruction set is a subset of the Thumb instruction set, programs written in LEGv7 Pinky can run on any Cortex M4 processor. You will use the STM32F411E-DISCO Board³ to test your assembly programs.

Using the STM32CubeIDE⁴, you can easily single step instructions and look at the registers and memory in the meantime. In this way we will get a better understanding of and insights in:

- What type of instructions a computer will need to support to gain some desired functionality.
- How a compiler might translate high-level code into assembly.

³ <https://www.st.com/en/evaluation-tools/32f411ediscovery.html>

⁴ <https://www.st.com/en/development-tools/stm32cubeide.html>

- How to use memory (and especially the stack).

During this assembly assignment you will only use the LEGv7 Pinky instruction set. These instructions are defined in a separate document [2]. **You are not allowed to use the other instructions from the ARMv7 Thumb instruction set.**

You need to assemble (pun intended) a small report which contains all the assembler functions you have written, the C programs which you have used to test your assembler functions, and the outcomes of those tests. Also, the answers to the questions raised in the assignments should be included in the report. This report should be uploaded to the appropriate assignment in Brightspace before the deadline which is defined in the “[Cursushandleiding](#)”.

Please note: you do not have to do all the assignments, only the ones assigned to you by your instructor. The list of assignments you have to do will be sent to you by email.

Good luck!

2

Assignments

In this chapter you will use STM32CubeIDE to develop and test a few LEGv7 Pinky assembler programs.

2.1 Assignment 1. Testing a simple LEGv7 Pinky program

This project consists of a very simple assembler function and a C program which calls the assembler function.

The C program [ass01/main.c](#) is given in [Listing 2.1](#).

The main function calls the test function which is defined in the file [ass01/test.s](#) shown in [Listing 2.2](#). The first argument is passed in register `R0` and the second argument is passed in register `R1`. The return value must be placed in register `R0`⁵

This very simple assembly program will return the sum of the two arguments. You can test this program as follows:

- Download [ass01.zip](#).
- In STM32CubeIDE, select   and choose “Projects from Folder or Archive”. Click .
- Click  and select the .zip file you just downloaded.
- Click .

⁵ This conforms with the *Procedure Call Standard for the Arm Architecture* [5].

```

/* main.c simple program to test assembler program */

#include <stdio.h>

extern int test(int a, int b);

int main(void)
{
    extern void initialise_monitor_handles(void);
    initialise_monitor_handles();

    int a = test(3, 5);
    printf("Result of test(3, 5) = %d\n", a);
    return 0;
}

```

Listing 2.1: A simple C program which calls our assembly function `ass01/main.c`.

```

.cpu cortex-m4
.thumb
.syntax unified
.globl test
.text
.thumb_func
test:
    ADD.N    R0, R0, R1
    BX.N     LR

```

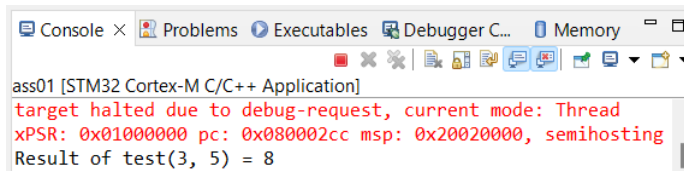
Listing 2.2: A very simple LEGv7 Pinky assembly function `ass01/test.s`.

- Open the directory “ass01.zip_expanded” and select the project “ass01”.
- Build this project and debug it as “STM32 Cortex-M C/C++ Application”, see [Figure 2.1](#).



Figure 2.1: Debug the ass01 project.

- When you run the program the output shown in [Figure 2.2](#) should appear in the Console window.



```
ass01 [STM32 Cortex-M C/C++ Application]
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x080002cc msp: 0x20020000, semihosting
Result of test(3, 5) = 8
```

Figure 2.2: The output of the ass01 project.

We use so called “semihosting” [6, paragraph 7.4.3]⁶ to make use of the `printf` and `scanf` functions. When you want to start a new project (to do the next assignments), the easiest way to do this is to copy the project “ass01” as follows:

- Select the project “ass01” and press `Ctrl` + `c`, `Ctrl` + `v`.
- Enter a project name (the default will be “ass2”) and click `Copy`.
- **Within the newly created project**, select the file “ass01.cfg”, press `F2`, and rename it to the new project name (e.g. ass2.cfg).
- **Within the newly created project**, select the file “ass01.launch”, press `F2` and rename it to the new project name (e.g. ass2.launch).
- Double-click on the new .launch file to open it, press `Ctrl` + `f` and change all occurrences of “ass01” into the new project name. Press `Ctrl` + `s` to save these changes.
- Build the project by pushing .
- Start the debugger by pushing .
- Run the program using the menu `Run` `Resume` or by pressing `F8`.
- You can now modify or remove the files in the new project’s directory “Core/Src” and/or add your own .c and .s files.

⁶ Direct link to [paragraph 7.4.3](#) of AN4989.

2.2 Assignment 2. Writing a simple multiplication program

Write an assembler subroutine in LEGv7 Pinky which can multiply two 32-bit unsigned integers based on the C code shown in [Listing 2.3](#). You may assume that the result fits into a 32-bit unsigned integer.

```
unsigned int multiply(unsigned int a, unsigned int b)
{
    unsigned int m = 0;
    for (unsigned int i = 0; i != a; i++)
    {
        m = m + b;
    }
    return m;
}
```

Listing 2.3: A naive multiply algorithm: `mulOne.c`.

Please note:

- You are not really using a LEGv7 Pinky assembler but you are actually using an ARMv7 Thumb assembler⁷. **For this assembly assignment you are only allowed to use Pinky instructions [2].**
- In the ARMv7 architecture registers `R0` to `R7` (low registers) and `R8` to `R12` (high registers) are available, in the LEGv7 architecture only the low registers (`R0` to `R7`) are available.
- To ensure that you only use 16-bit instructions add the suffix `.N` after each assembly instruction.
- ARMv7 software is required to have the stack pointer aligned to double word (8 byte) addresses at a public interface⁸, see [5, paragraph 6.2.1.2].

In [Listing 2.4](#) a C program is given which calls your assembly code to test it. The GNU C compiler⁹ which is used by STM32CubeIDE conforms to the *Procedure Call Standard for the Arm Architecture* [5]. So, you can assume that the value of parameter `a` is present in argument register `R0` and that parameter `b` is present in argument register `R1`. Also, the return value should be saved in result register `R0`.

⁷ See <https://sourceware.org/binutils/docs/as/>. To generate Thumb instructions you have to use the `.thumb` assembler directive as shown in [Listing 2.2](#).

⁸ For example, when calling a C function from assembly.

⁹ See <https://gcc.gnu.org/onlinedocs/gcc/>.

```

int main()
{
    extern void initialise_monitor_handles(void);
    initialise_monitor_handles();

    unsigned int a[] = {0, 1, 0, 1, 2000,      2,      10000,      ↵
    ↵ 1,      65535 };
    unsigned int b[] = {0, 0, 1, 1, 2, 65535,      65535, ↵
    ↵ 4294967295u,      65535 };
    unsigned int r[] = {0, 0, 0, 1, 4000, 131070, 655350000, ↵
    ↵ 4294967295u, 4294836225u};

    for (size_t i = 0; i != sizeof(a)/sizeof(a[0]); i++)
    {
        printf("%u x %u: ", a[i], b[i]);
        unsigned int result = multiply(a[i], b[i]);
        unsigned int correct = r[i];
        if (result != correct)
        {
            printf("Failed, function returned %u but the correct ↵
    ↵ answer is %u\n", result, correct);
        }
        else
        {
            printf("Passed, %u\n", result);
        }
    }
    return 0;
}

```

Listing 2.4: A program to test your assembly code: [mulOne.c](#).

How many instructions does it take your procedure to run the code:
 multiply(65535, 65535)?

2.3 Assignment 3. A smarter multiply algorithm

As we have seen in [Section 2.2](#), the multiply routine takes a lot of time in the worst case. There is a smarter method to multiply two values. It is given in C for your convenience in [Listing 2.5](#).

```
unsigned int multiply(unsigned int a, unsigned int b)
{
    unsigned int m = 0;

    while (b != 0)
    {
        if ((b & 1) == 1) /* b is odd */
        {
            m = m + a;
        }
        a = a << 1;
        b = b >> 1;
    }
    return m;
}
```

Listing 2.5: A smarter multiply algorithm [mulTwo.c](#).

Adjust the multiply function you wrote for [Section 2.2](#) to work like the code given in [Listing 2.5](#) and test all the cases listed in [Listing 2.4](#) again.

How many instructions does it take your procedure to run the code:
multiply(65535, 65535)?

2.4 Assignment 4. A smarter, recursive multiply algorithm

As we have seen in [Section 2.2](#), the multiply routine takes a lot of time in the worst case. There is a smarter method to multiply two values. It is given in C for your convenience in [Listing 2.6](#).

```
unsigned int multiply(unsigned int a, unsigned int b)
{
    if (b == 0) return 0;
    if (b == 1) return a;
    if ((b & 1) == 0) /* b is even */ return multiply(a << 1, b >> 1);
    ↩ else /* b is odd */ return a + multiply(a << 1, b >> 1);
}
```

Listing 2.6: A smarter, recursive multiply algorithm [mulThree.c](#).

Adjust the multiply function you wrote for [Section 2.2](#) to work like the code given in [Listing 2.6](#) and test all the cases listed in [Listing 2.4](#) again.

How many instructions does it take your procedure to run the code:
multiply(65535, 65535)?

2.5 Assignment 5. A smarter, tail recursive multiply algorithm

As we have seen in [Section 2.2](#), the multiply routine takes a lot of time in the worst case. There is a smarter method to multiply two values. It is given in C for your convenience in [Listing 2.7](#).

```
unsigned int multiply2(unsigned int m, unsigned int a, unsigned int b)
↪ int b)
{
    if (b == 0) return 0;
    if (b == 1) return m + a;
    if ((b & 1) == 0) /* b is even */ return multiply2(m, a << 1, ↪
↪ b >> 1);
    else /* b is odd */ return multiply2(m + a, a << 1, b >> 1);
}

unsigned int multiply(unsigned int a, unsigned int b)
{
    return multiply2(0, a, b);
}
```

Listing 2.7: A smarter, tail recursive multiply algorithm [mulFour.c](#).

Adjust the multiply function you wrote for [Section 2.2](#) to work like the code given in [Listing 2.7](#) and test all the cases listed in [Listing 2.4](#) again.

Please note that the function `multiply2` uses tail recursion¹⁰. You can use this fact to simplify your assembler code.

How many instructions does it take your procedure to run the code:
`multiply(65535, 65535)?`

¹⁰ See: https://en.wikipedia.org/wiki/Tail_call.

2.6 Assignment 6. Using your multiply function to calculate a power

Now write an assembly function called `power` to calculate n^m where n and m are 32-bit unsigned integers. You may assume that the result fits into a 32-bit unsigned integer. You *have to* call the `multiply` function you wrote for [Section 2.2](#) from within your `power` function. The algorithm is given in C for your convenience in [Listing 2.8](#). Make sure your code is properly tested.

```
unsigned int power(unsigned int n, unsigned int m)
{
    unsigned int p = 1;

    for (unsigned int i = 0; i != m; i++)
    {
        p = multiply(p, n);
    }
    return p;
}
```

Listing 2.8: A simple power algorithm [powerOne.c](#).

2.7 Assignment 7. A smarter power algorithm

A simple implementation of the power function, see [Section 2.6](#) performs m multiplications to calculate n^m . There is a smarter method to calculate a power. This method is called exponentiation by squaring¹¹. Now write an assembly function called `power` to calculate n^m where n and m are 32-bit unsigned integers. You may assume that the result fits into a 32-bit unsigned integer. You *have to* call the `multiply` function you wrote for [Section 2.2](#), [2.3](#) or [2.5](#) from within your power function. The algorithm is given in C for your convenience in [Listing 2.9](#). Make sure your code is properly tested.

```
unsigned int power(unsigned int n, unsigned int m)
{
    if (m == 0) return 1;

    unsigned int p = 1;

    while (m != 1)
    {
        if ((m & 1) == 1) /* m is odd */
        {
            p = multiply(p, n);
        }
        n = multiply(n, n);
        m = m >> 1;
    }
    return multiply(p, n);
}
```

Listing 2.9: A powerTwo power algorithm [powerTwo.c](#).

The simple calculation of 7^{11} , using the algorithm given in [Section 2.6](#), performs 11 multiplication. How many multiplications are needed in your implementation of the power function?

¹¹ See: https://en.wikipedia.org/wiki/Exponentiation_by_squaring.

2.8 Assignment 8. A smarter, recursive power algorithm

A simple implementation of the power function, see [Section 2.6](#) performs m multiplications to calculate n^m . There is a smarter method to calculate a power. This method is called exponentiation by squaring¹². Now write an assembly function called `power` to calculate n^m where n and m are 32-bit unsigned integers. You may assume that the result fits into a 32-bit unsigned integer. You *have to* call the `multiply` function you wrote for [Section 2.2](#), [2.3](#) or [2.5](#) from within your power function. The algorithm is given in C for your convenience in [Listing 2.10](#). Make sure your code is properly tested.

```
unsigned int power(unsigned int n, unsigned int m)
{
    if (m == 0) return 1;
    if (m == 1) return n;
    if ((m & 1) == 0) /* m is even */ return power(multiply(n, n), ←
    ↪ m >> 1);
    else /* m is odd */ return multiply(n, power(multiply(n, n), m ←
    ↪ >> 1));
}
```

Listing 2.10: A powerThree power algorithm [powerThree.c](#).

The simple calculation of 7^{11} , using the algorithm given in [Section 2.6](#), performs 11 multiplication. How many multiplications are needed in your implementation of the power function?

¹² See: https://en.wikipedia.org/wiki/Exponentiation_by_squaring.

2.9 Assignment 9. A smarter, tail recursive power algorithm

A simple implementation of the power function, see [Section 2.6](#) performs m multiplications to calculate n^m . There is a smarter method to calculate a power. This method is called exponentiation by squaring¹³. Now write an assembly function called `power` to calculate n^m where n and m are 32-bit unsigned integers. You may assume that the result fits into a 32-bit unsigned integer. You *have to* call the `multiply` function you wrote for [Section 2.2](#), [2.3](#) or [2.5](#) from within your power function. The algorithm is given in C for your convenience in [Listing 2.11](#). Make sure your code is properly tested.

```
unsigned int power2(unsigned int p, unsigned int n, unsigned int m)
{
    if (m == 0) return p;
    if (m == 1) return multiply(p, n);
    if ((m & 1) == 0) /* m is even */ return power2(p, multiply(n, ←
↵ n), m >> 1);
    else /* m is odd */ return power2(multiply(p, n), multiply(n, ←
↵ n), m >> 1);
}

unsigned int power(unsigned int n, unsigned int m)
{
    return power2(1, n, m);
}
```

Listing 2.11: A powerFour power algorithm [powerFour.c](#).

Please note that the function `power2` uses tail recursion¹⁴. You can use this fact to simplify your assembler code.

The simple calculation of 7^{11} , using the algorithm given in [Section 2.6](#), performs 11 multiplication. How many multiplications are needed in your implementation of the power function?

¹³ See: https://en.wikipedia.org/wiki/Exponentiation_by_squaring.

¹⁴ See: https://en.wikipedia.org/wiki/Tail_call.

2.10 Assignment 10. Using your multiply function to calculate the dot product of two vectors

Now write an assembly function called `dotProduct` to calculate the dot product of two vectors $\mathbf{a} \cdot \mathbf{b}$ where $\mathbf{a}$ and $\mathbf{b}$ are both vectors which contain n 32-bit unsigned integers. The dot product of two vectors of size n is defined in [Equation \(2.1\)](#).

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n-1} a_i b_i = a_0 b_0 + a_1 b_1 + \cdots + a_{n-1} b_{n-1} \quad (2.1)$$

You may assume that the result fits into a 32-bit unsigned number. You *have to* call the multiply function you wrote for [Section 2.2](#), [2.3](#) or [2.5](#) from within your `dotProduct` function. A basic test program is shown in [Listing 2.12](#). Expand this program to make sure your code is properly tested.

A C implementation of the function you have to implement in LEGv7 Pinky is shown in [Listing 2.13](#).

```
#include <stdio.h>

unsigned int dotProduct(unsigned int a[], unsigned int b[], size_t n) {
    ↪ n);

int main()
{
    extern void initialise_monitor_handles(void);
    initialise_monitor_handles();

    unsigned int a[] = { 1, 2, 3, 4, 5};
    unsigned int b[] = {10, 11, 12, 13, 14};

    if (dotProduct(a, b, 5) == 190)
    {
        printf("OK\n");
    }
    else
    {
        printf("Error\n");
    }
    return 0;
}
```

Listing 2.12: A basic test program to test the function dotProduct, [dotProduct.c](#).

```
unsigned int dotProduct(unsigned int a[], unsigned int b[], size_t n)
{
    unsigned int p = 0;
    for (size_t i = 0; i != n; i++)
    {
        p = p + multiply(a[i], b[i]);
    }
    return p;
}
```

Listing 2.13: A C implementation of the function dotProduct, [dotProduct.c](#).

2.11 Assignment 11. Using your multiply function to calculate a square root

Now write an assembly function called `sqrtFloor` to calculate the floor of the square root of a 32-bit unsigned integer. The floor of the square root means that the sqrt is rounded *down* to the nearest integer. You may note that the result fits into a 32-bit unsigned number. You *have to* call the multiply function you wrote for [Section 2.2](#), [2.3](#) or [2.5](#) from within your `sqrtFloor` function. Make sure your code is properly tested. A basic test program is shown in [Listing 2.14](#).

A C implementation of the function you have to implement in LEGv7 Pinky is shown in [Listing 2.15](#).

```

#include <stdio.h>

unsigned int sqrtFloor(unsigned int n);

int main()
{
    extern void initialise_monitor_handles(void);
    initialise_monitor_handles();

    unsigned int a[] = {0, 1, 2, 3, 4, 152399025, 152423715, ↵
↵ 4294836225, 4294967295};
    unsigned int r[] = {0, 1, 1, 1, 2,      12345,      12345,      ↵
↵ 65535,      65535};

    for (size_t i = 0; i < sizeof(a)/sizeof(a[0]); i++)
    {
        printf("sqrtFloor(%u): ", a[i]);
        unsigned int result = sqrtFloor(a[i]);
        unsigned int correct = r[i];
        if (result != correct)
        {
            printf("Failed, function returned %u but the correct ↵
↵ answer is %u\n", result, correct);
        }
        else
        {
            printf("Passed, %u\n", result);
        }
    }
    return 0;
}

```

Listing 2.14: A basic test program to test the function `sqrtFloor`, `sqrtFloor.c`.

```
unsigned int sqrtFloor(unsigned int n)
{
    unsigned int p = 1u << 15;
    unsigned int r = 0;

    do
    {
        r = p | r;
        if (multiply(r, r) > n)
        {
            r = r & ~p;
        }
        p = p >> 1;
    }
    while (p != 0);
    return r;
}
```

Listing 2.15: A C implementation of the function `sqrtFloor`, [sqrtFloor.c](#).

2.12 Assignment 12. Sorting an array of 32-bit integers by using insertion sort

Write a `insertionSort` function in LEV7 Pinky using the insertion sort algorithm¹⁵.

A C implementation of the function you have to implement in LEV7 Pinky is shown in [Listing 2.16](#). A basic test program is shown in [Listing 2.17](#). Your assembly function must call the C function `swap` given in [Listing 2.18](#).

```
void insertionSort(int a[], size_t n)
{
    for (size_t i = 0; i != n; i++)
    {
        for (size_t j = i; j != 0 && a[j-1] > a[j]; j--)
        {
            swap(&a[j], &a[j-1]);
        }
    }
}
```

Listing 2.16: A C implementation of the function `insertionSort`, [insertionSort.c](#).

¹⁵ See: https://en.wikipedia.org/wiki/Insertion_sort

```
#include <stdio.h>

void insertionSort(int a[], size_t n);

int main(void)
{
    extern void initialise_monitor_handles(void);
    initialise_monitor_handles();

    int a[] = {1, -2, 7, -4, 5};
    int b[] = {-4, -2, 1, 5, 7};

    insertionSort(a, sizeof(a)/sizeof(a[0]));
    for (size_t i = 0; i != sizeof(a)/sizeof(a[0]); i++)
    {
        if (a[i] != b[i])
        {
            printf("Error\n");
            return 1;
        }
    }
    printf("OK\n");
    return 0;
}
```

Listing 2.17: A basic test program to test the function `insertionSort`, `insertionSort.c`.

```
void swap(int *p1, int *p2)
{
    int t = *p1;
    *p1 = *p2;
    *p2 = t;
}
```

Listing 2.18: A C implementation of the function `swap`, `insertionSort.c`.

2.13 Assignment 13. Sorting an array of 32-bit integers by using selection sort

Write a `selectionSort` function in LEGv7 Pinky using the selection sort algorithm¹⁶.

A C implementation of the function you have to implement in LEGv7 Pinky is shown in [Listing 2.19](#). A basic test program is shown in [Listing 2.20](#). Your assembly function must call the C function `swap` given in [Listing 2.21](#).

```
void selectionSort(int a[], size_t n)
{
    for (size_t j = 0; j != n - 1; j++)
    {
        size_t iMin = j;
        for (size_t i = j + 1; i != n; i++)
        {
            if (a[i] < a[iMin])
            {
                iMin = i;
            }
        }
        if (iMin != j)
        {
            swap(&a[j], &a[iMin]);
        }
    }
}
```

Listing 2.19: A C implementation of the function `selectionSort`, [selectionSort.c](#).

¹⁶ See: https://en.wikipedia.org/wiki/Selection_sort

```
#include <stdio.h>

void selectionSort(int a[], size_t n);

int main()
{
    extern void initialise_monitor_handles(void);
    initialise_monitor_handles();

    int a[] = {1, -2, 7, -4, 5};
    int b[] = {-4, -2, 1, 5, 7};

    selectionSort(a, sizeof(a)/sizeof(a[0]));
    for (size_t i = 0; i != sizeof(a)/sizeof(a[0]); i++)
    {
        if (a[i] != b[i])
        {
            printf("Error\n");
            return 0;
        }
    }
    printf("OK\n");
    return 0;
}
```

Listing 2.20: A basic test program to test the function `selectionSort`, [selectionSort.c](#).

```
void swap(int *p1, int *p2)
{
    int t = *p1;
    *p1 = *p2;
    *p2 = t;
}
```

Listing 2.21: A C implementation of the function `swap`, [selectionSort.c](#).

2.14 Assignment 14. Sorting an array of 32-bit integers by using cocktail shaker sort

Write a `cocktailShakerSort` function in LEGv7 Pinky using the cocktail shaker sort algorithm¹⁷.

A C implementation of the function you have to implement in LEGv7 Pinky is shown in [Listing 2.22](#). A basic test program is shown in [Listing 2.23](#). Your assembly function must call the C function `swap` given in [Listing 2.24](#).

```
void cocktailShakerSort(int a[], size_t n)
{
    for (size_t j = 0; j != n/2; j++)
    {
        for (size_t i = j; i != n - 1 - j; i++)
        {
            if (a[i] > a[i+1])
            {
                swap(&a[i], &a[i+1]);
            }
        }
        for (size_t i = n - 2 - j; i != j; i--)
        {
            if (a[i-1] > a[i])
            {
                swap(&a[i-1], &a[i]);
            }
        }
    }
}
```

Listing 2.22: A C implementation of the function `cocktailShakerSort`, [cocktailShakerSort.c](#).

¹⁷ See: https://en.wikipedia.org/wiki/Cocktail_shaker_sort

```
#include <stdio.h>

void cocktailShakerSort(int a[], size_t n);

int main()
{
    extern void initialise_monitor_handles(void);
    initialise_monitor_handles();

    int a[] = {1, -2, 7, -4, 5};
    int b[] = {-4, -2, 1, 5, 7};

    cocktailShakerSort(a, sizeof(a)/sizeof(a[0]));
    for (size_t i = 0; i != sizeof(a)/sizeof(a[0]); i++)
    {
        if (a[i] != b[i])
        {
            printf("Error\n");
            return 0;
        }
    }
    printf("OK\n");
    return 0;
}
```

Listing 2.23: A basic test program to test the function `cocktailShakerSort`, [cocktailShakerSort.c](#).

```
void swap(int *p1, int *p2)
{
    int t = *p1;
    *p1 = *p2;
    *p2 = t;
}
```

Listing 2.24: A C implementation of the function `swap`, [cocktailShakerSort.c](#).

2.15 Assignment 15. Sorting an array of 32-bit integers by using gnome sort

Write a `gnomeSort` function in LEGv7 Pinky using the gnome sort algorithm¹⁸.

A C implementation of the function you have to implement in LEGv7 Pinky is shown in [Listing 2.25](#). A basic test program is shown in [Listing 2.26](#). Your assembly function must call the C function `swap` given in [Listing 2.27](#).

```
void gnomeSort(int a[], size_t n)
{
    size_t i = 0;
    while (i != n)
    {
        if (i == 0 || a[i] >= a[i-1])
        {
            i++;
        }
        else
        {
            swap(&a[i], &a[i-1]);
            i--;
        }
    }
}
```

Listing 2.25: A C implementation of the function `gnomeSort`, [gnomeSort.c](#).

¹⁸ See: https://en.wikipedia.org/wiki/Gnome_sort

```
#include <stdio.h>

void gnomeSort(int a[], size_t n);

int main()
{
    extern void initialise_monitor_handles(void);
    initialise_monitor_handles();

    int a[] = {1, -2, 7, -4, 5};
    int b[] = {-4, -2, 1, 5, 7};

    gnomeSort(a, sizeof(a)/sizeof(a[0]));
    for (size_t i = 0; i != sizeof(a)/sizeof(a[0]); i++)
    {
        if (a[i] != b[i])
        {
            printf("Error\n");
            return 0;
        }
    }
    printf("OK\n");
    return 0;
}
```

Listing 2.26: A basic test program to test the function `gnomeSort`, `gnomeSort.c`.

```
void swap(int *p1, int *p2)
{
    int t = *p1;
    *p1 = *p2;
    *p2 = t;
}
```

Listing 2.27: A C implementation of the function `swap`, `gnomeSort.c`.

2.16 Assignment 16. Sorting an array of 32-bit integers by using stooge sort

Write a `stoogeSort` function in LEGv7 Pinky using the stooge sort algorithm¹⁹.

A C implementation of the function you have to implement in LEGv7 Pinky is shown in [Listing 2.28](#). A basic test program is shown in [Listing 2.29](#). Your assembly function must call the C function `swap` given in [Listing 2.30](#).

In this assignment you are allowed to use the **UDIV** instruction from the Thumb instruction set, see [\[4, paragraph 3.6.12\]](#)²⁰.

```
void stoogeSort(int a[], size_t first, size_t last)
{
    if (a[first] > a[last])
    {
        swap(&a[first], &a[last]);
    }
    if ((last - first + 1) > 2)
    {
        size_t third = (last - first + 1) / 3;
        stoogeSort(a, first, last - third);
        stoogeSort(a, first + third, last);
        stoogeSort(a, first, last - third);
    }
}
```

Listing 2.28: A C implementation of the function `stoogeSort`, [stoogeSort.c](#).

¹⁹ See: https://en.wikipedia.org/wiki/Stooge_sort

²⁰ <https://documentation-service.arm.com/static/5f2ac76d60a93e65927bbdc5#G6.1094353>

```
#include <stdio.h>

void stoogeSort(int a[], size_t first, size_t last);

int main()
{
    extern void initialise_monitor_handles(void);
    initialise_monitor_handles();

    int a[] = {1, -2, 7, -4, 5};
    int b[] = {-4, -2, 1, 5, 7};

    stoogeSort(a, 0, sizeof(a)/sizeof(a[0]) - 1);
    size_t i;
    for (i = 0; i < sizeof(a)/sizeof(a[0]); i++)
    {
        if (a[i] != b[i])
        {
            printf("Error\n");
            return 0;
        }
    }
    printf("OK\n");
    return 0;
}
```

Listing 2.29: A basic test program to test the function `stoogeSort`, [stoogeSort.c](#).

```
void swap(int *p1, int *p2)
{
    int t = *p1;
    *p1 = *p2;
    *p2 = t;
}
```

Listing 2.30: A C implementation of the function `swap`, [stoogeSort.c](#).

2.17 Assignment 17. Sorting an array of 32-bit integers by using slow sort

Write a `slowSort` function in LEGv7 Pinky using the slow sort algorithm²¹.

A C implementation of the function you have to implement in LEGv7 Pinky is shown in [Listing 2.31](#). A basic test program is shown in [Listing 2.32](#). Your assembly function must call the C function `swap` given in [Listing 2.33](#).

```
void slowSort(int a[], size_t first, size_t last)
{
    if (first != last)
    {
        size_t middle = (first + last) >> 1;
        slowSort(a, first, middle);
        slowSort(a, middle + 1, last);
        if (a[last] < a[middle])
        {
            swap(&a[last], &a[middle]);
        }
        slowSort(a, first, last - 1);
    }
}
```

Listing 2.31: A C implementation of the function `slowSort`, [slowSort.c](#).

²¹ See: https://en.wikipedia.org/wiki/Slow_sort

```
#include <stdio.h>

void slowSort(int a[], size_t first, size_t last);

int main()
{
    extern void initialise_monitor_handles(void);
    initialise_monitor_handles();

    int a[] = {1, -2, 7, -4, 5};
    int b[] = {-4, -2, 1, 5, 7};

    slowSort(a, 0, sizeof(a)/sizeof(a[0]) - 1);
    for (size_t i = 0; i < sizeof(a)/sizeof(a[0]); i++)
    {
        if (a[i] != b[i])
        {
            printf("Error\n");
            return 0;
        }
    }
    printf("OK\n");
    return 0;
}
```

Listing 2.32: A basic test program to test the function `slowSort`, [slowSort.c](#).

```
void swap(int *p1, int *p2)
{
    int t = *p1;
    *p1 = *p2;
    *p2 = t;
}
```

Listing 2.33: A C implementation of the function `swap`, [slowSort.c](#).

2.18 Assignment 18. Sorting an array of 32-bit integers by using optimized bubble sort

Write a `bubbleSortOpt` function in LEGv7 Pinky using the optimized bubble sort algorithm²².

A C implementation of the function you have to implement in LEGv7 Pinky is shown in [Listing 2.34](#). A basic test program is shown in [Listing 2.35](#). Your assembly function must call the C function `swap` given in [Listing 2.36](#).

```
void bubbleSortOpt(int a[], size_t n)
{
    size_t newn;
    do
    {
        newn = 0;
        for (size_t i = 1; i != n; i++)
        {
            if (a[i-1] > a[i])
            {
                swap(&a[i-1], &a[i]);
                newn = i;
            }
        }
        n = newn;
    }
    while (n != 0);
}
```

Listing 2.34: A C implementation of the function `bubbleSortOpt`, [bubbleSortOpt.c](#).

²² See: https://en.wikipedia.org/wiki/Bubble_sort

```
#include <stdio.h>

void bubbleSortOpt(int a[], size_t n);

int main()
{
    extern void initialise_monitor_handles(void);
    initialise_monitor_handles();

    int a[] = {1, -2, 7, -4, 5};
    int b[] = {-4, -2, 1, 5, 7};

    bubbleSortOpt(a, sizeof(a)/sizeof(a[0]));
    for (size_t i = 0; i != sizeof(a)/sizeof(a[0]); i++)
    {
        if (a[i] != b[i])
        {
            printf("Error\n");
            return 0;
        }
    }
    printf("OK\n");
    return 0;
}
```

Listing 2.35: A basic test program to test the function `bubbleSortOpt`, `bubbleSortOpt.c`.

```
void swap(int *p1, int *p2)
{
    int t = *p1;
    *p1 = *p2;
    *p2 = t;
}
```

Listing 2.36: A C implementation of the function `swap`, `bubbleSortOpt.c`.

Bibliography

- [1] *Arm v7-M Architecture – Reference Manual*. ARM, 2021, DDI 0403E.e. URL: <https://documentation-service.arm.com/static/606dc36485368c4c2b1bf62f> (cit. on p. 4).
- [2] Harry Broeders. *De LEGv7 architectuur en de Pinky instructieset*. Hogeschool Rotterdam, 2022. URL: https://github.com/HR-ELEKTRO/rts1/wiki/LEGv7/LEGv7-Pinky_ebook.pdf (cit. on pp. 4, 5, 9).
- [3] *Cortex-M4 Datasheet*. ARM, 2020. URL: <https://www.arm.com/-/media/Arm%20Developer%20Community/PDF/Processor%20Datasheets/Arm%20Cortex-M4%20Processor%20Datasheet.pdf> (cit. on p. 4).
- [4] *Cortex-M4 Devices – Generic User Guide*. ARM, 2011, DUI 0553. URL: <https://documentation-service.arm.com/static/5f2ac76d60a93e65927bbdc5> (cit. on pp. 4, 31).
- [5] *Procedure Call Standard for the Arm Architecture*. ARM, 2022, AAPCS32-1. URL: <https://github.com/ARM-software/abi-aa/releases> (cit. on pp. 6, 9).
- [6] *STM32 microcontroller debug toolbox*. STMicroelectronics, 2021, AN4989. URL: https://www.st.com/resource/en/application_note/dm00354244-stm32-microcontroller-debug-toolbox-stmicroelectronics.pdf (cit. on p. 8).