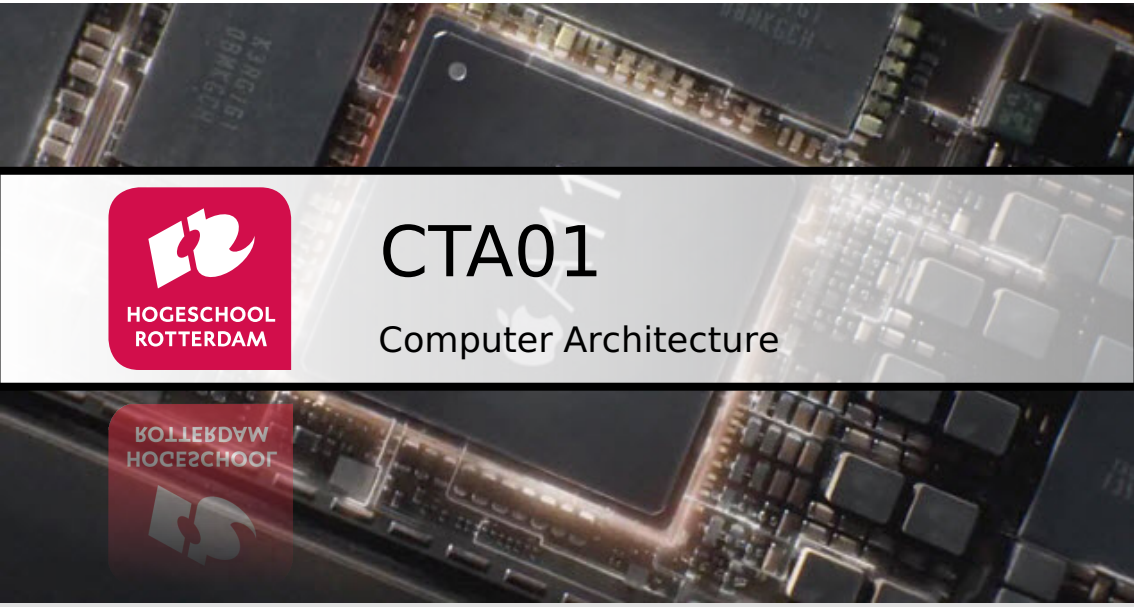

CTA01

Computer Architecture

Lab Work Handbook

Version 2.5a

J.W. Peltenburg
J.Z.M. Broeders

Version History

Date	Version	Description	Author
20-08-2021[1]	2.5a[2]	Updated for 2021–2022.	BroJZ
03-01-2020	2.4b	Updated for 2020–2021. Removed use of W-registers.	BroJZ
08-11-2019	2.3	Updated for 2019–2020. Added eight new assignments.	BroJZ
24-11-2018	2.2a	Updated for 2018–2019.	BroJZ
18-11-2017	2.1b	New assignments for 2017–2018.	BroJZ
01-12-2016	2.0	New version for LEGv8.	BroJZ
28-03-2011	1.0	Initial version for MIPS.	PelJH

[1] Dates are formated 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

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 engineering by looking at the LEGv8 processor [1] (which is especially suitable for this purpose) we will also write assembly programs for this processor. We will verify the programs by using a simulator, so we can easily single step instructions and look at the registers and memory in the meanwhile. In this way we will get a better understanding of and insights in:

- What type of instructions a computer will need to support to use to gain some desired functionality.
- How a compiler might translate high-level code into assembly.
- How to use memory (and especially the stack).

During the lab we will only use the LEGv8 Core Instruction Set. These instructions are defined in your book [1, Figure 2.1] and can also be found in the first column of the green LEGv8 reference data card which came with your book. **You are not**

allowed to use the Arithmetic Core Instruction Set but you may use the four instructions listed in the Pseudoinstruction 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 MS Teams 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 send to you by email.

Good luck!

2

Using ARM DS-5 and writing programs

In this lab you will use the free Community Edition of ARM's Integrated Development Environment called DS-5. Which can be downloaded from the CTA01 MS Team Documents[3]. The installation does not need any further explanation.

You will also have to download a proper toolchain which includes an ARMv8 simulator. This toolchain can be downloaded from the Linaro website[4] or from the CTA01 MS Team Documents[5]. Download this file and extract it to `C:\`.

[3] https://hrnl.sharepoint.com/sites/EAS-ELE_CTA01_2021-2022/Lesmateriaal/Installatiebestanden/ds5-ce-windows64-29rel1.zip

[4] https://releases.linaro.org/components/toolchain/binaries/latest-7/aarch64-elf/gcc-linaro-7.5.0-2019.12-i686-mingw32_aarch64-elf.tar.xz

[5] https://hrnl.sharepoint.com/sites/EAS-ELE_CTA01_2021-2022/Lesmateriaal/Installatiebestanden/gcc-linaro-7.5.0-2019.12-i686-mingw32_aarch64-elf.tar.xz

2.1 Installing Linaro aarch64-elf GCC as a toolchain in DS-5

You can add the toolchain in DS-5 as follows:

- In DS-5, select Window → Preferences → DS-5 → Toolchains.
- In the Toolchains dialog, click Add.
- In the Select Toolchain Path dialog, enter the path to the toolchain binaries. This will normally be `C:\gcc-linaro-7.5.0-2019.12-i686-mingw32_aarch-64-elf\bin`. Then click Next.
- In the Discovered Toolchain Information dialog, check the toolchain information is correct (as shown in Figure 2.1), then click Finish.
- The toolchain appears in the Toolchains dialog, click Apply, then click Restart Eclipse.

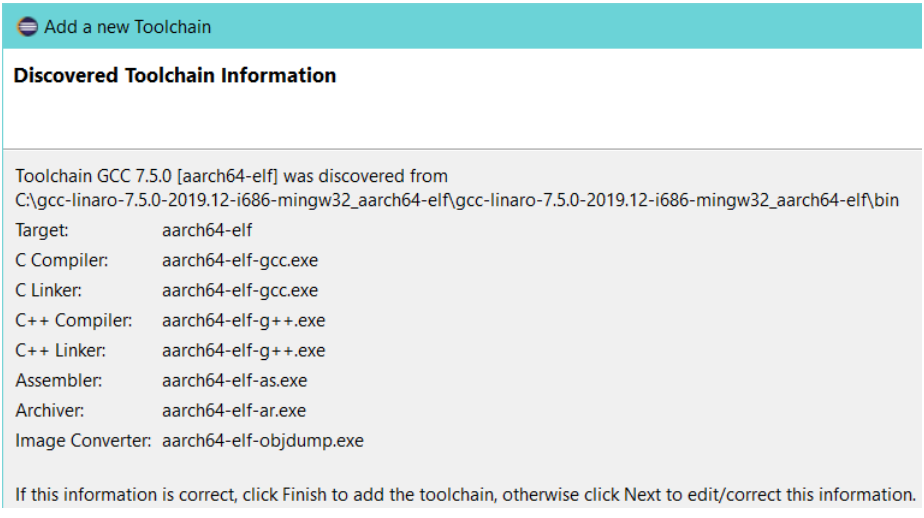

Figure 2.1: The Discovered Toolchain Information dialog.

2.2 Running a C program on the ARM-v8 simulator

Unzip the file `Bare-metal_examples_Armv8.zip` which can be found in `C:\Program Files\DS-5 CE v5.29.1\examples\`. For example into the directory `C:\Bare-metal_examples_Armv8\`.

You can run the `calendar_Armv8-A_GCC` example in DS-5 as follows:

- In DS-5, select File → Import... → General → Existing Projects into Workspace. Then click Next.
- Select the directory `C:\Bare-metal_examples_Armv8\`.
- Check the `calendar_Armv8-A_GCC` example and the Copy Projects into Workspace option, see Figure 2.2. Then click Finish.
- Right click on the project name `calendar_Armv8-A_GCC` in the Project Explorer and choose Properties.
- Select C/C++ Build → Tool Chain Editor
- Uncheck the Display compatible toolchains only option.
- Select `GCC 7.5.0 [aarch64-elf]` as Current toolchain.
- Click Yes and OK.
- Click on the Build button  or right click on the project name in the Project Explorer and choose Build Project.
- Right click on the project name in the Project Explorer and choose Debug As, Debug Configurations...
- Open the option DS-5 Debugger, select `calendar_Armv8-A_GCC-FM`, see Figure 2.3 and click Debug and Yes.
- You can now simulate the application. You can see the output and enter the input in the Target Console window, see Figure 2.4.

When you run the program it ends with a message that the source for `svc.h` can not be found. This is caused by the fact that the program returns with a `return 0;` statement but there is nothing (no operating system) to return to. We are running the program bare-metal, remember. If you don't like this, you can set a breakpoint

on the `return 0;` statement in `main` or replace it with a `while (1);` instruction, so the program will not return. You have to stop the program yourself in this case.

Please note: this example deliberately contains an error so you can practice your debug skills, see the file `readme.html` which is included in the project directory.

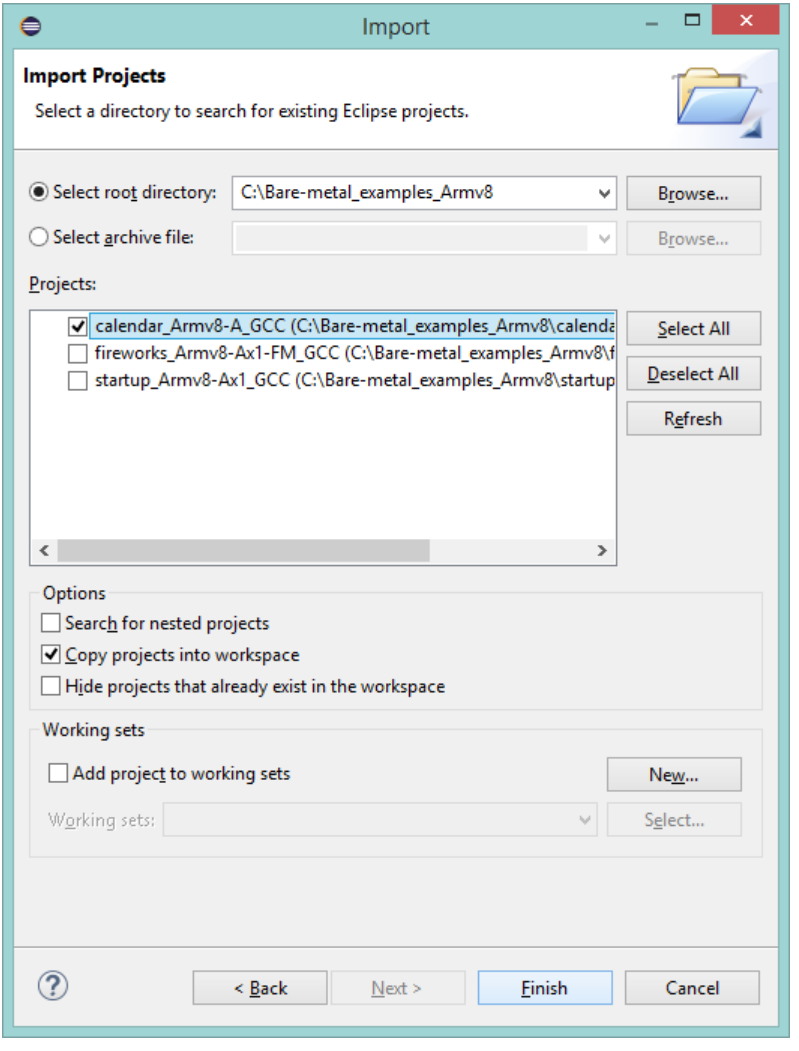

Figure 2.2: The Import Projects dialog.

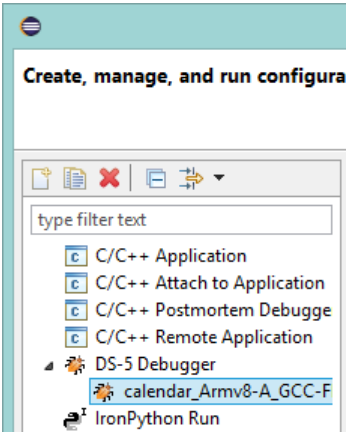

Figure 2.3: The Debug Configurations dialog.

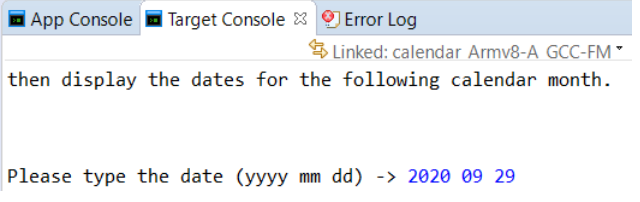

Figure 2.4: The Target Console window.

3

Assignments

You have installed ARM DS-5 and the Linaro toolchain and simulator now. In this chapter you will use these tools to develop and test a few LEGv8 assembler programs.

3.1 Assignment 1. Testing a simple LEGv8 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 3.1.

The `main` function calls the `test` function which is defined in the file `ass01/test.S` shown in Listing 3.2. The first argument is passed in register `X0` and the second argument is passed in register `X1`. The return value must be placed in register `X0`[6].

[6] This conforms with the Procedure Call Standard for the ARM 64-bit Architecture (AArch64), see https://developer.arm.com/docs/ihi0055/latest/procedure-call-standard-for-the-arm-64-bit-architecture.

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

#include <stdio.h>

extern long long int test(long long int a, long long int b);

int main(void)
{
    long long int a = test(3, 5);
    printf("Result of test(3, 5) = %lld\n", a);
    return 0;
}
```

Listing 3.1: A simple C program which calls our assembly function ass01/main.c.

```
    .globl test
test:
    add X0, X0, X1
    br  X30
```

Listing 3.2: A very simple LEGv8 assembly function ass01/test.S.

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

- Visit https://bitbucket.org/HarryBroeders/legv8 and fork this repository.
- In DS-5, select File → Import... → Git → Projects from Git. Click Next.
- Select Clone URI and click Next.
- Fill in the URI to your own fork of the repository and click Next (twice).
- Fill in the Destination Directory and click Next.
- Select ass01 and click Finish.
- Right click on the project name ass01 in the Project Explorer and choose Build Project.

- Right click on the project name in the Project Explorer and choose Debug As, Debug Configurations...
- Open the option DS-5 Debugger, select `ass01` and click Debug and Yes.

When you run the program the output shown in Figure 3.1 should appear in the Target Console window.

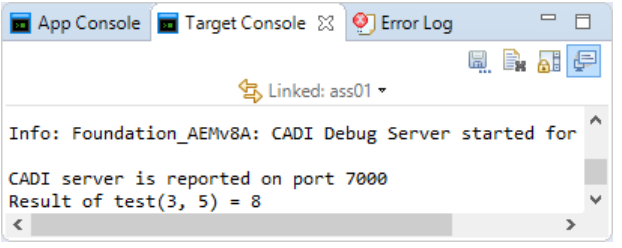

Figure 3.1: The output of the `ass01` project.

3.2 Assignment 2. Writing a simple multiplication program

Write an assembler subroutine for LEGv8 which can multiply two 64-bit unsigned integers based on the C code shown in Listing 3.3. You may assume that the result fits into a 64-bit unsigned integer.

```
unsigned long long int multiply(unsigned long long int a, ↩
 ↪ unsigned long long int b)
{
    unsigned long long int i;
    unsigned long long int m = 0;

    for (i = 0; i != a; i++)
    {
        m = m + b;
    }

    return m;
}
```

Listing 3.3: A naive multiply algorithm `mulOne.c`.

Please note: you are not really using a LEGv8 assembler but you are actually using an ARMv8 assembler. So please review the remarks made by Patterson and Hennessy [1, Paragraph 2.23], which are summarized below:

- Immediate versions are not separate assembly language instructions in ARMv8. While LEGv8 has `ADDI`, `SUBI`, etc, ARMv8 does not. You simply use the non-immediate version and use an immediate operand. For example, to use the LEGv8 instruction: `ADDI X0, X1, #4` you can just use `ADD X0, X1, #4` (without the `I`).
- In the ARMv8 instruction set, register `X31` is `XZR` in most instructions but the stack pointer `SP` in others. In LEGv8 `X28` is used as stack pointer. So you best use the symbolic names `XZR` and `SP` in your assembly code, to avoid confusion.

- In the LEGv8 assembler language the symbolic register name `LR` can be used. This name is not recognized by the ARMv8 assembler so you must use `X30`.
- The immediate field for the logic instructions are not simple 12-bit constants in ARMv8. ARMv8 uses an algorithm to encode these immediate values. This means that some values are valid and others are not (e.g., 0 and 5).

There is another **important difference** between LEGv8 and ARMv8 to be considered, see [1, page 114]:

- ARMv8 software is required to keep the stack pointer aligned to quadword (16 byte) addressses. **This means that the `SP` can only be in- or decreased by a multiple of 16!**

Also write a C program which calls your assembler code to test it. You can assume that the value of parameter `a` is present in argument register `X0` and that parameter `b` is present in argument register `X1`. Also, the return value should be saved in result register `X0`.

Test your function by at least running:

- `multiply(0, 0)`
- `multiply(1, 0)`
- `multiply(0, 1)`
- `multiply(1, 1)`
- `multiply(2000, 2)`
- `multiply(2, 4294967295)`
- `multiply(1000000, 4294967295)`
- `multiply(1, 18446744073709551615u)`
- `multiply(4294967295, 4294967296)`

How many instructions does it take your procedure to run the code: `multiply(4294967295, 4294967295)`?

3.3 Assignment 3. A smarter multiply algorithm

As we have seen in Section 3.2, the multiply routine takes a lot of time in the worst case. There is a smarter method to multiply two values. The algorithm is shown in Figure 3.4 of the book [1, page 194]. It is actually explained to be for hardware, but it works in software as well. It is given in C for your convenience in Listing 3.4.

```
unsigned long long int multiply(unsigned long long int a, ↩
 ↪ unsigned long long int b)
{
    unsigned long long 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 3.4: A smarter multiply algorithm `mulTwo.c`.

Adjust the multiply function you wrote for Section 3.2 to work like the code given in Listing 3.4 and test all the cases listed in Section 3.2 again.

How many instructions does it take your procedure to run the code: `multiply(4294967295, 4294967295)`?

3.4 Assignment 4. A smarter, recursive multiply algorithm

As we have seen in Section 3.2, 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 3.5.

```
unsigned long long int multiply(unsigned long long int a, ↩
 ↪ unsigned long long 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 3.5: A smarter, recursive multiply algorithm `mulThree.c`.

Adjust the multiply function you wrote for Section 3.2 to work like the code given in Listing 3.5 and test all the cases listed in Section 3.2 again.

How many instructions does it take your procedure to run the code: `multiply(4294967295, 4294967295)`?

3.5 Assignment 5. A smarter, tail recursive multiply algorithm

As we have seen in Section 3.2, 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 3.6.

```
unsigned long long int multiply2(unsigned long long int m, ↩
 ↪ unsigned long long int a, unsigned long long 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 long long int multiply(unsigned long long int a, ↩
 ↪ unsigned long long int b)
{
    return multiply2(0, a, b);
}
```

Listing 3.6: A smarter, tail recursive multiply algorithm `mulFour.c`.

Adjust the multiply function you wrote for Section 3.2 to work like the code given in Listing 3.6 and test all the cases listed in Section 3.2 again.

Please note that the function `multiply2` uses tail recursion[7]. You can use this fact to simplify your assembler code.

How many instructions does it take your procedure to run the code: `multiply(4294967295, 4294967295)`?

[7] See: https://en.wikipedia.org/wiki/Tail_call.

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

Now write an assembly function called `power` to calculate n^m were n and m are 64-bit unsigned integers. You may assume that the result fits into a 64-bit unsigned integer. You *have to* call the `multiply` function you wrote for Section 3.2 from within your `power` function. The algorithm is given in C for your convenience in Listing 3.7. Make sure your code is properly tested.

```
unsigned long long int power(unsigned long long int n, ↩
 ↪ unsigned long long int m)
{
    unsigned long long int i;
    unsigned long long int p = 1;

    for (i = 0; i != m; i++)
    {
        p = p * n;
    }

    return p;
}
```

Listing 3.7: A simple power algorithm `power0ne.c`.

3.7 Assignment 7. A smarter power algorithm

A simple implementation of the power function, see Section 3.6 performs m multiplications to calculate n^m. There is a smarter method to calculate a power. This method is called exponentiation by squaring[8]. Now write an assembly function called `power` to calculate n^m were n and m are 64-bit unsigned integers. You may assume that the result fits into a 64-bit unsigned integer. You *have to* call the `multiply` function you wrote for Section 3.2 or Section 3.5 from within your `power` function. The algorithm is given in C for your convenience in Listing 3.8. Make sure your code is properly tested.

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

    unsigned long long int p = 1;

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

    return p * n;
}
```

Listing 3.8: A powerTwo power algorithm `powerTwo.c`.

[8] See: https://en.wikipedia.org/wiki/Exponentiation_by_squaring.

The simple calculation of 7^{21}, using the algorithm given in Section 3.6, performs 21 multiplication. How many multiplications are needed in your implementation of the `power` function.

3.8 Assignment 8. A smarter, recursive power algorithm

A simple implementation of the power function, see Section 3.6 performs m multiplications to calculate n^m. There is a smarter method to calculate a power. This method is called exponentiation by squaring[9]. Now write an assembly function called `power` to calculate n^m were n and m are 64-bit unsigned integers. You may assume that the result fits into a 64-bit unsigned integer. You *have to* call the `multiply` function you wrote for Section 3.2 or Section 3.5 from within your `power` function. The algorithm is given in C for your convenience in Listing 3.9. Make sure your code is properly tested.

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

Listing 3.9: A powerThree power algorithm `powerThree.c`.

The simple calculation of 7^{21}, using the algorithm given in Section 3.6, performs 21 multiplication. How many multiplications are needed in your implementation of the power function.

[9] See: https://en.wikipedia.org/wiki/Exponentiation_by_squaring.

3.9 Assignment 9. A smarter, tail recursive power algorithm

A simple implementation of the power function, see Section 3.6 performs m multiplications to calculate n^m. There is a smarter method to calculate a power. This method is called exponentiation by squaring[10]. Now write an assembly function called `power` to calculate n^m were n and m are 64-bit unsigned integers. You may assume that the result fits into a 64-bit unsigned integer. You *have to* call the `multiply` function you wrote for Section 3.2 or Section 3.5 from within your `power` function. The algorithm is given in C for your convenience in Listing 3.10. Make sure your code is properly tested.

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

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

Listing 3.10: A powerFour power algorithm `powerFour.c`.

Please note that the function `power2` uses tail recursion[11]. You can use this fact to simplify your assembler code.

[10] See: https://en.wikipedia.org/wiki/Exponentiation_by_squaring.

[11] See: https://en.wikipedia.org/wiki/Tail_call.

The simple calculation of 7^{21}, using the algorithm given in Section 3.6, performs 21 multiplication. How many multiplications are needed in your implementation of the `power` function.

3.10 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 $\mathbf{a} \cdot \mathbf{b}$ were $\mathbf{a}$ and $\mathbf{b}$ are both vectors which contain n 64-bit unsigned integers. The dot product of two vectors of size n is defined in Equation (3.1).

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n-1} a_i b_i = a_0 b_0 + a_1 b_1 + \cdots + a_{n-1} b_{n-1} \tag{3.1}$$

You may assume that the result fits into a 64-bit unsigned number. You *have to* call the `multiply` function you wrote for Section 3.3 or Section 3.5 from within your `dotProduct` function. Make sure your code is properly tested. A basic test program is shown in Listing 3.11.

A C implementation of the function you have to implement in LEGv8 is shown in Listing 3.12.

```c
#include <stdio.h>

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

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

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

Listing 3.11: A basic test program to test the function dotProduct, dotProduct.c.

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

Listing 3.12: A C implementation of the function dotProduct, dotProduct.c.

3.11 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 64-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 3.3 or Section 3.5 from within your `sqrtFloor` function. Make sure your code is properly tested. A basic test program is shown in Listing 3.13.

A C implementation of the function you have to implement in LEGv8 is shown in Listing 3.14.

```c
#include <stdio.h>

unsigned long long int sqrtFloor(unsigned long long int n);

int main()
{
    unsigned long long int a[] = {0, 1, 2, 3, 4, 152416036884721 00u, 152416036884720 99u, 184467440737095 51615u};
    unsigned long long int r[] = {0, 1, 1, 1, 2, 123456890, 123456889, 4294967295};

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

Listing 3.13: A basic test program to test the function sqrtFloor, sqrtFloor.c.

```c
unsigned long long int sqrtFloor(unsigned long long int n)
{
    unsigned long long int p = 1u << 31;
    unsigned long long int r = 0;

    do
    {
        r = p | r;
        if (r * r > n)
        {
            r = r & ~p;
        }
        p = p >> 1;
    }
    while (p != 0);

    return r;
}
```

Listing 3.14: A C implementation of the function `sqrtFloor`, `sqrtFloor.c`.

3.12 Assignment 12. Sorting an array of 64-bit integers by using insertion sort

In paragraph 2.13 of your book [1] a simple bubble sort function is given. Now write a `insertionSort` function in LEGv8 using the insertion sort algorithm[12].

A C implementation of the function you have to implement in LEGv8 is shown in Listing 3.15. A basic test program is shown in Listing 3.16. Your assembly function must call the C function `swap` given in Listing 3.17.

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

Listing 3.15: A C implementation of the function `insertionSort`, `insertionSort.c`.

[12] See: https://en.wikipedia.org/wiki/Insertion_sort

```c
#include <stdio.h>

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

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

    insertionSort(a, sizeof(a)/sizeof(a[0]));
    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 3.16: A basic test program to test the function `insertionSort`, `insertionSort.c`.

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

Listing 3.17: A C implementation of the function `swap`, `insertionSort.c`.

3.13 Assignment 13. Sorting an array of 64-bit integers by using selection sort

In paragraph 2.13 of your book [1] a simple bubble sort function is given. Now write a `selectionSort` function in LEGv8 using the selection sort algorithm[13].

A C implementation of the function you have to implement in LEGv8 is shown in Listing 3.18. A basic test program is shown in Listing 3.19. Your assembly function must call the C function `swap` given in Listing 3.20.

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

Listing 3.18: A C implementation of the function `selectionSort`, `selectionSort.c`.

[13] See: https://en.wikipedia.org/wiki/Selection_sort

```c
#include <stdio.h>

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

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

    selectionSort(a, sizeof(a)/sizeof(a[0]));
    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 3.19: A basic test program to test the function selectionSort, selectionSort.c.

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

Listing 3.20: A C implementation of the function swap, selectionSort.c.

3.14 Assignment 14. Sorting an array of 64-bit integers by using cocktail shaker sort

In paragraph 2.13 of your book [1] a simple bubble sort function is given. Now write a `cocktailShakerSort` function in LEGv8 using the cocktail shaker sort algorithm[14].

A C implementation of the function you have to implement in LEGv8 is shown in Listing 3.21. A basic test program is shown in Listing 3.22. Your assembly function must call the C function `swap` given in Listing 3.23.

[14] See: https://en.wikipedia.org/wiki/Cocktail_shaker_sort

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

Listing 3.21: A C implementation of the function `cocktailShakerSort`, `cocktailShakerSort.c`.

```c
#include <stdio.h>

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

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

    cocktailShakerSort(a, sizeof(a)/sizeof(a[0]));
    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 3.22: A basic test program to test the function cocktailShakerSort, cocktailShakerSort.c.

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

Listing 3.23: A C implementation of the function swap, cocktailShakerSort.c.

3.15 Assignment 15. Sorting an array of 64-bit integers by using gnome sort

In paragraph 2.13 of your book [1] a simple bubble sort function is given. Now write a `gnomeSort` function in LEGv8 using the gnome sort algorithm[15].

A C implementation of the function you have to implement in LEGv8 is shown in Listing 3.24. A basic test program is shown in Listing 3.25. Your assembly function must call the C function `swap` given in Listing 3.26.

```
void gnomeSort(long long 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 3.24: A C implementation of the function `gnomeSort`, `gnomeSort.c`.

[15] See: https://en.wikipedia.org/wiki/Gnome_sort

```c
#include <stdio.h>

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

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

    gnomeSort(a, sizeof(a)/sizeof(a[0]));
    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 3.25: A basic test program to test the function gnomeSort, gnomeSort.c.

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

Listing 3.26: A C implementation of the function swap, gnomeSort.c.

3.16 Assignment 16. Sorting an array of 64-bit integers by using stooge sort

In paragraph 2.13 of your book [1] a simple bubble sort function is given. Now write a `stoogeSort` function in LEGv8 using the stooge sort algorithm[16].

A C implementation of the function you have to implement in LEGv8 is shown in Listing 3.27. A basic test program is shown in Listing 3.28. Your assembly function must call the C function `swap` given in Listing 3.29.

In this assignment you are allowed to use the **UDIV**-instruction from the Arithmetic Core Instruction Set.

```
void stoogeSort(long long 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 3.27: A C implementation of the function `stoogeSort`, `stoogeSort.c`.

[16] See: https://en.wikipedia.org/wiki/Stooge_sort

```c
#include <stdio.h>

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

int main()
{
    long long int a[] = {1, -2, 7, -4, 5};
    long long 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 3.28: A basic test program to test the function stoogeSort, stoogeSort.c.

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

Listing 3.29: A C implementation of the function swap, stoogeSort.c.

3.17 Assignment 17. Sorting an array of 64-bit integers by using slow sort

In paragraph 2.13 of your book [1] a simple bubble sort function is given. Now write a `slowSort` function in LEGv8 using the slow sort algorithm[17].

A C implementation of the function you have to implement in LEGv8 is shown in Listing 3.30. A basic test program is shown in Listing 3.31. Your assembly function must call the C function swap given in Listing 3.32.

```
void slowSort(long long 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 3.30: A C implementation of the function `slowSort`, `slowSort.c`.

[17] See: https://en.wikipedia.org/wiki/Slow_sort

```c
#include <stdio.h>

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

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

    slowSort(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 3.31: A basic test program to test the function slowSort, slowSort.c.

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

Listing 3.32: A C implementation of the function swap, slowSort.c.

3.18 Assignment 18. Sorting an array of 64-bit integers by using optimized bubble sort

In paragraph 2.13 of your book [1] a simple bubble sort function is given. Now write a `bubbleSortOpt` function in LEGv8 using the optimized bubble sort algorithm[18].

A C implementation of the function you have to implement in LEGv8 is shown in Listing 3.33. A basic test program is shown in Listing 3.34. Your assembly function must call the C function `swap` given in Listing 3.35.

```
void bubbleSortOpt(long long int a[], size_t n)
{
    size_t newn;
    do
    {
        newn = 0;
        size_t i;
        for (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 3.33: A C implementation of the function `bubbleSortOpt`, `bubbleSortOpt.c`.

[18] See: https://en.wikipedia.org/wiki/Bubble_sort

```c
#include <stdio.h>

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

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

    bubbleSortOpt(a, sizeof(a)/sizeof(a[0]));
    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 3.34: A basic test program to test the function bubbleSortOpt, bubbleSortOpt.c.

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

Listing 3.35: A C implementation of the function swap, bubbleSortOpt.c.

Bibliography

[1] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design: The Hardware Software Interface: ARM Edition*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2016. ISBN: 978-0-12-801835-4 (cit. on pp. 5, 16, 17, 18, 32, 34, 36, 39, 41, 43, 45).