

Template Week 4 – Software

Student number: 586377

Assignment 4.1: ARM assembly

Screenshot of working assembly code of factorial calculation:

The screenshot shows the OakSim simulation interface. On the left, the assembly code for calculating factorial is displayed:

```
1 Main:
2     mov r1,
3     mov r2,
4
5
6 Loop:
7     mul r1, r1, r2 @ r1 = r1 * r2
8     add r2, r2,
9     cmp r2,
10    beq End @ if r2 == 0, branch to End
11    b Loop @ otherwise continue loop
12
13
14 End:
15 @ r1 now contains 5! = 120 (0x78)
```

On the right, the register values are listed:

Register	Value
R0	0
R1	0
R2	0
R3	0
R4	0
R5	0
R6	0
R7	0
R8	0
R9	0
R10	0
R11	0
R12	0

The memory dump area shows the value 0x78 at address 0x00010000, which corresponds to the value of R1.

Assignment 4.2: Programming languages

Take screenshots that the following commands work:

```
luuk@luuk-Virtual-Platform:~$ javac --version
javac 21.0.8
luuk@luuk-Virtual-Platform:~$ java --version
openjdk 21.0.8 2025-07-15
OpenJDK Runtime Environment (build 21.0.8+9-Ubuntu-0ubuntu124.04.1)
OpenJDK 64-Bit Server VM (build 21.0.8+9-Ubuntu-0ubuntu124.04.1, mixed mode, sharing)
luuk@luuk-Virtual-Platform:~$ gcc --version
gcc (Ubuntu 13.3.0-0ubuntu2-24.04) 13.3.0
Copyright (C) 2023 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

luuk@luuk-Virtual-Platform:~$ python3 --version
Python 3.12.3
luuk@luuk-Virtual-Platform:~$ bash --version
GNU bash, version 5.2.21(1)-release (x86_64-pc-linux-gnu)
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
luuk@luuk-Virtual-Platform:~$
```

javac --version

java --version

gcc --version

python3 --version

Assignment 4.3: Compile

Which of the above files need to be compiled before you can run them?

Fibonacci.java & fib.c

Which source code files are compiled into machine code and then directly executable by a processor?

fib.c needs to be compiled by gcc into a machine-code executable.

Which source code files are compiled to byte code?

Fibonacci.java needs to be compiled by javac into Java bytecode (.class file)

Which source code files are interpreted by an interpreter?

fib.py needs to be interpreted by the Python interpreter (python3)

fib.sh needs to be interpreted by the Bash shell (bash)

These source code files will perform the same calculation after compilation/interpretation. Which one is expected to do the calculation the fastest?

Generally fastest → C program

Because:

- C is compiled into native machine code
- Runs directly on the CPU
- No virtual machine or interpreter overhead

Next fastest → Java

Slower → Python, Bash

How do I run a Java program?

First you need to compile and then run the program commands: javac Fibonacci.java & java Fibonacci

How do I run a Python program?

This runs directly with this command: python3 fib.py

How do I run a C program?

First you need to compile and then run the program commands: gcc fib.c -o fib & ./fib

How do I run a Bash script?

This runs directly with this command: ./fib.sh

If I compile the above source code, will a new file be created? If so, which file?

It will produce Fibonacci.class in java.

In C it will produce: fib

And in python and bash it won't produce new files

Take relevant screenshots of the following commands:

- Compile the source files where necessary
- Make them executable
- Run them
- Which (compiled) source code file performs the calculation the fastest?

```
luuk@luuk-Virtual-Platform:~/Downloads/code$ javac Fibonacci.java
luuk@luuk-Virtual-Platform:~/Downloads/code$ java Fibonacci
Fibonacci(18) = 2584
Execution time: 0.25 milliseconds
```

```
luuk@luuk-Virtual-Platform:~/Downloads/code$ gcc fib.c -o fib
luuk@luuk-Virtual-Platform:~/Downloads/code$ ./fib
Fibonacci(18) = 2584
Execution time: 0.02 milliseconds
```

```
luuk@luuk-Virtual-Platform:~/Downloads/code$ python3 fib.py
Fibonacci(18) = 2584
Execution time: 0.41 milliseconds
```

```
luuk@luuk-Virtual-Platform:~/Downloads/code$ chmod +x fib.sh
luuk@luuk-Virtual-Platform:~/Downloads/code$ ./fib.sh
Fibonacci(18) = 2584
Excution time 7161 milliseconds
```

gcc is the fasttest

Assignment 4.4: Optimize

Take relevant screenshots of the following commands:

- a) Figure out which parameters you need to pass to **the gcc** compiler so that the compiler performs a number of optimizations that will ensure that the compiled source code will run faster. **Tip!** The parameters are usually a letter followed by a number. Also read **page 191** of your book, but find a better optimization in the man pages. Please note that Linux is case sensitive.

-O3 was de snelste want dit geeft de maximale optimalisatie.

- b) Compile **fib.c** again with the optimization parameters

```
luuk@luuk-Virtual-Platform:~/Downloads/code$ gcc -O3 -march=native fib.c
-o fib_fast
luuk@luuk-Virtual-Platform:~/Downloads/code$ ./fib
Fibonacci(18) = 2584
Execution time: 0.02 milliseconds
```

- c) Run the newly compiled program. Is it true that it now performs the calculation faster?

```
luuk@luuk-Virtual-Platform:~/Downloads/code$ ./fib
Fibonacci(18) = 2584
Execution time: 0.02 milliseconds
luuk@luuk-Virtual-Platform:~/Downloads/code$ ./fib_fast
Fibonacci(18) = 2584
Execution time: 0.00 milliseconds
```

- d) Edit the file **runall.sh**, so you can perform all four calculations in a row using this Bash script. So the (compiled/interpreted) C, Java, Python and Bash versions of Fibonacci one after the

```
Running C program:
Fibonacci(19) = 4181
Execution time: 0.01 milliseconds

Running Java program:
Fibonacci(19) = 4181
Execution time: 0.24 milliseconds

Running Python program:
Fibonacci(19) = 4181
Execution time: 0.54 milliseconds

Running BASH Script
S

other.
```

Assignment 4.5: More ARM Assembly

Like the factorial example, you can also implement the calculation of a power of 2 in assembly. For example you want to calculate $2^4 = 16$. Use iteration to calculate the result. Store the result in r0.

Main:

```
mov r1, #2  
mov r2, #4
```

Loop:

```
mul r0, r0, r1    @ r0 = r0 * 2  
sub r2, r2, #1    @ r2 = r2 - 1  
cmp r2, #0        @ is exponent finished?  
bne Loop          @ if not zero, repeat
```

End:

Complete the code. See the PowerPoint slides of week 4.

Screenshot of the completed code here.

The screenshot shows the OakSim debugger interface. The assembly code window displays the following code:

```
1 Main:  
2     mov r0, #1      @ result = 1  
3     mov r1, #2      @ base = 2  
4     mov r2, #4      @ exponent = 4  
5  
6 Loop:  
7     mul r0, r0, r1  @ r0 = r0 * 2  
8     sub r2, r2, #1  @ r2 = r2 - 1  
9     cmp r2, #0      @ is exponent finished?  
10    bne Loop       @ if not zero, repeat  
11 End:
```

The register values window shows the following initial values:

Register	Value
R0	2
R1	2
R2	4
R3	0
R4	0
R5	0
R6	0
R7	0
R8	0
R9	0
R10	0
R11	0
R12	0

The memory dump window shows the memory starting at address 0x000010000, with the first few bytes being 01 00 A0 E3 02 10 A0 E3 04 20 A0 E3 90 01 00 E0 ... B .. R.

Ready? Save this file and export it as a pdf file with the name: [week4.pdf](#)