

Template Week 4 – Software

Student number: 582031

Assignment 4.1: ARM assembly

Screenshot of working assembly code of factorial calculation:

The screenshot shows the QEMU ARM simulator interface. On the left, there is a text editor window displaying the following ARM assembly code:

```
1 Main:
2     mov r1, #1      ; r1 = 1
3     mov r2, #5      ; r2 = 5
4 Loop:
5     mul r1, r1, r2 ; r1 = r1 * r2
6     sub r2, r2, #1 ; r2 = r2 - 1
7     cmp r2, #0      ; compare r2 with 0
8     beq End        ; if r2 == 0, jump to End
9     b Loop         ; otherwise, loop again
10 End:
```

On the right, there is a table titled "Register Value" showing the current state of the registers:

Register	Value
R0	51
R1	19a1
R2	0
R3	0
R4	0
R5	0
R6	0
R7	0
R8	0
R9	0
R10	0
R11	0
R12	0
SP	10000

Below the register table is a memory dump window showing the memory starting at address 0x00010000. The dump shows various assembly instructions and their addresses.

Assignment 4.2: Programming languages

Take screenshots that the following commands work:

```
javac --version
```

```
java --version
```

```
gcc --version
```

```
python3 --version
```

```
bash --version
```

```
javac 21.0.9
openjdk version "21.0.9" 2025-10-21
OpenJDK Runtime Environment (build 21.0.9+10-Ubuntu-124.04)
OpenJDK 64-Bit Server VM (build 21.0.9+10-Ubuntu-124.04, mixed mode, sharing)
gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
Python 3.12.3
GNU bash, version 5.2.21(1)-release (x86_64-pc-linux-gnu)
```


Assignment 4.3: Compile

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

Fibonacci.java en fib.c moeten eerst gecompileerd worden voordat je ze kunt uitvoeren.

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

fib.c C code wordt gecompileerd naar native machine code die direct door de processor uitgevoerd kan worden.

Which source code files are compiled to byte code?

Fibonacci.java Java wordt gecompileerd naar bytecode (.class bestand) die door de Java Virtual Machine (JVM) wordt uitgevoerd.

Which source code files are interpreted by an interpreter?

fib.py (Python interpreter) en fib.sh (Bash interpreter) worden regel voor regel geïnterpreteerd tijdens het uitvoeren.

How do I run a Java program?

Eerst compileren met javac Fibonacci.java, dan uitvoeren met java Fibonacci

How do I run a Python program?

python3 fib.py

How do I run a C program?

Eerst compileren met gcc fib.c -o fib, dan uitvoeren met ./fib

How do I run a Bash script?

Executable maken met chmod +x fib.sh, dan uitvoeren met ./fib.sh

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

Fibonacci.java maakt Fibonacci.class en fib.c maakt fib (ligt aan -o)

Take relevant screenshots of the following commands:

- Compile the source files where necessary
- Make them executable

```
server) in auto mode
lib/jvm/java_21_openjdk_amd64/bin/serialver to o
ver) in lib/jvm/
mode olivier@olivier:~/Desktop/code$ gcc fib.c -o fib
+exp1) olivier@olivier:~/Desktop/code$ javac Fibonacci.java
21.0.9+olivier@olivier:~/Desktop/code$ chmod +x fib.sh
ib/jvm/olivier@olivier:~/Desktop/code$ chmod +x fib
) in auolivier@olivier:~/Desktop/code$ 
2:1.21-7
+exp1)

21
d 21.0.9
t1.0.9+10
) 13.3.6

use (x86_

```

- Run them
- Which (compiled) source code file performs the calculation the fastest?

```
olivier@olivier:~/Desktop/code$ javac Fibonacci.java
olivier@olivier:~/Desktop/code$ chmod +x fib.sh
olivier@olivier:~/Desktop/code$ chmod +x fib
olivier@olivier:~/Desktop/code$ java Fibonacci
Fibonacci(18) = 2584
Execution time: 0.26 milliseconds
olivier@olivier:~/Desktop/code$ python3 fib.py
Fibonacci(18) = 2584
Execution time: 0.46 milliseconds
olivier@olivier:~/Desktop/code$ ./fib.sh

Fibonacci(18) = 2584
Excution time 7067 milliseconds
olivier@olivier:~/Desktop/code$ 
olivier@olivier:~/Desktop/code$ ./fib
Fibonacci(18) = 2584
Execution time: 0.02 milliseconds
olivier@olivier:~/Desktop/code$ 
```

De C code was het snelst.

Assignment 4.4: Optimize

Take relevant screenshots of the following commands:

- In de man page van gcc kun je zoeken met / en dan -O typen. De compiler heeft verschillende optimalisatie levels: -O0 (geen), -O1, -O2, -O3 en -Ofast. Hoe hoger het getal, hoe meer optimalisaties de compiler toepast. -O3 is de beste keuze voor maximale snelheid omdat deze alle beschikbare optimalisaties activeert zonder risico's met precisie.
- Compile **fib.c** again with the optimization parameters

```
olivier@olivier:~/Desktop/code$ gcc -O3 fib.c -o fib
olivier@olivier:~/Desktop/code$ ./fib
Fibonacci(18) = 2584
Execution time: 0.00 milliseconds
olivier@olivier:~/Desktop/code$
```

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

```
olivier@olivier:~/Desktop/code$ gcc -O3 fib.c -o fib
olivier@olivier:~/Desktop/code$ ./fib
Fibonacci(18) = 2584
Execution time: 0.00 milliseconds
olivier@olivier:~/Desktop/code$
```

Ja, het voert nu sneller uit dan 0,00ms

- 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 other.

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

Running Java program:
Fibonacci(19) = 4181
Execution time: 1.43 milliseconds

Running Python program:
Fibonacci(19) = 4181
Execution time: 0.72 milliseconds
```

```
olivier@olivier:~/Desktop/code$ nano runall.sh
olivier@olivier:~/Desktop/code$ sudo chmod +x ./runall.sh
olivier@olivier:~/Desktop/code$
```

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:

End:

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

Screenshot of the completed code here.

The screenshot shows the QEMU simulator interface. On the left, the assembly code is displayed:

```
1 Main:
2   mov r0, #1
3   mov r1, #2
4   mov r2, #4
5 Loop:
6   mul r0, r0, r1
7   sub r2, r2, #1
8   cmp r2, #0
9   beq End
10  b  Loop
11 End:
```

On the right, the register values are listed:

Register	Value
R0	10
R1	2
R2	0
R3	0
R4	0
R5	0
R6	0
R7	0
R8	0
R9	0
R10	0
R11	0
R12	0
SP	10000

The memory dump area at the bottom shows the memory starting from address 0x0000100000, containing the assembly code and some initial values.

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