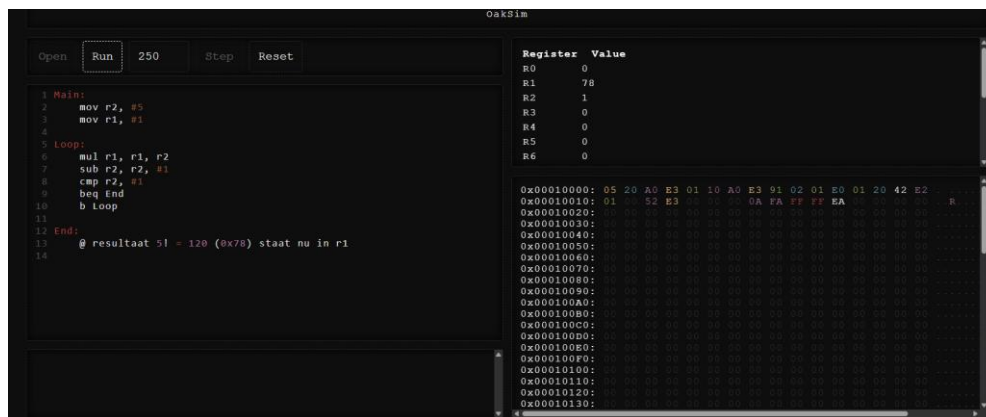


Template Week 4 – Software

Student number: 589948

Assignment 4.1: ARM assembly

Screenshot of working assembly code of factorial calculation:



Assignment 4.2: Programming languages

Take screenshots that the following commands work:

javac --version

java --version

gcc --version

python3 --version

bash --version

```
morris@morris-VMware-Virtual-Platform: ~  
morris@morris-VMware-Virtual-Platform:~$ javac --version  
javac 21.0.8  
morris@morris-VMware-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, sha  
ring)  
morris@morris-VMware-Virtual-Platform:~$ gcc --version  
gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 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.  
morris@morris-VMware-Virtual-Platform:~$ python3 --version  
Python 3.12.3  
morris@morris-VMware-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.  
morris@morris-VMware-Virtual-Platform:~$
```

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 → na compileren krijg je een native executable (bijv. a.out of fib_c)

Which source code files are compiled to byte code?

- Fibonacci.java → Fibonacci.class (Java bytecode, draait in de JVM)

Which source code files are interpreted by an interpreter?

- fib.py → door de Python-interpreter
- fib.sh → door de bash-shell

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

- Het C-programma (fib.c) is het snelst, omdat C wordt direct gecompileerd naar machinecode en CPU voert dit meteen uit. Java draait via de JVM echt snel, maar met extra lagen. Python is geïnterpreteerd, dus langzamer. Bash is veel langzamer, dus slecht geschikt voor zware berekeningen.

How do I run a Java program?

-Een Java-programma moet eerst gecompileerd worden. Je voert hiervoor het commando `javac` uit op het `.java`-bestand. Hierdoor ontstaat er een nieuw `.class`-bestand. Daarna kun je het programma uitvoeren door `java` te gebruiken gevolgd door de bestandsnaam zonder extensie.

How do I run a Python program?

-Een Python-programma hoeft niet gecompileerd te worden. Het wordt rechtstreeks uitgevoerd door de Python-interpreter. Je voert het uit door `python3` te gebruiken gevolgd door de naam van het `.py`-bestand.

How do I run a C program?

-Een C-programma moet eerst gecompileerd worden met een C-compiler, bijvoorbeeld GCC. Het compileren zet de C-broncode om naar een uitvoerbaar bestand. Na het compileren kun je het programma uitvoeren door het gegenereerde uitvoerbestand te starten.

How do I run a Bash script?

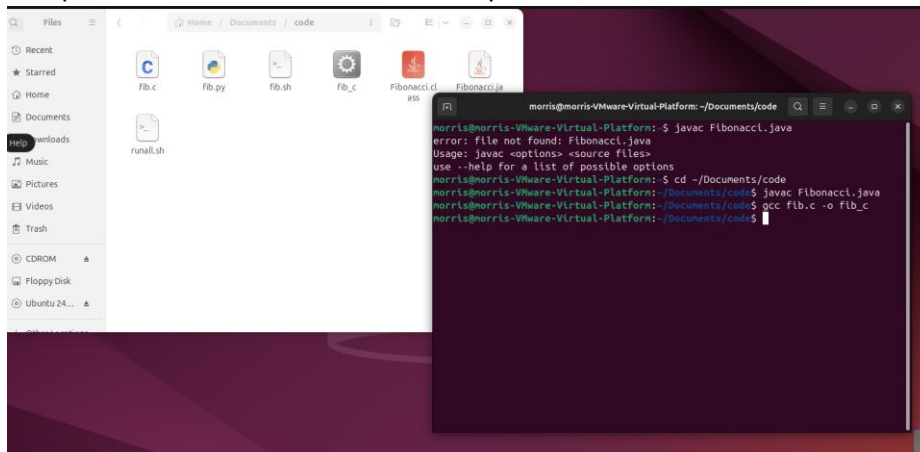
-Een Bash-script hoeft niet gecompileerd te worden, maar het moet wel uitvoerrechten krijgen. Je geeft het script uitvoerrechten met het commando `chmod`. Daarna kun je het uitvoeren door het scriptbestand direct te starten.

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

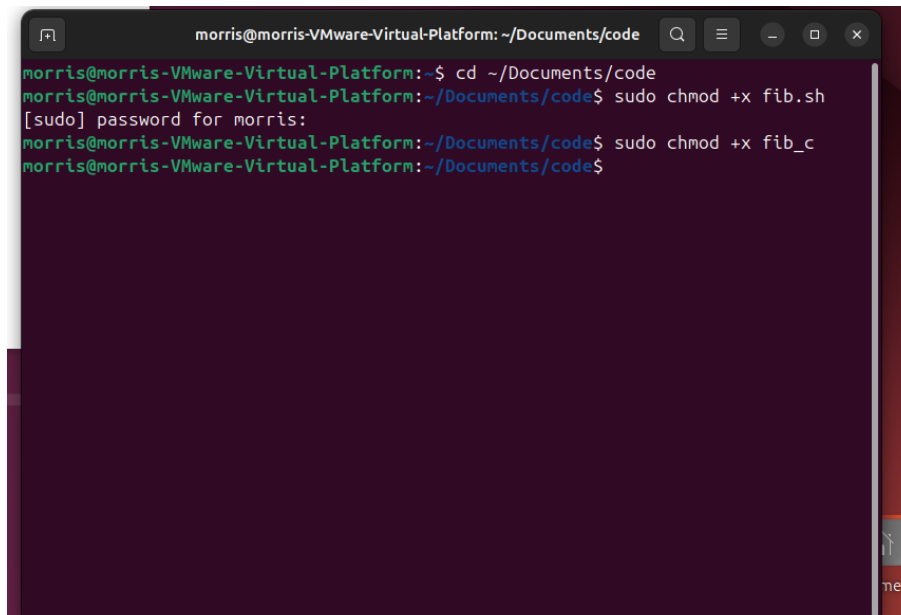
-Bij Java wordt er een nieuw .class-bestand aangemaakt wanneer je het .java-bestand compileert. Bij C wordt er een nieuw uitvoerbaar bestand aangemaakt wanneer je het .c-bestand compileert. Bij Python wordt er geen nieuw bestand aangemaakt omdat Python geïnterpreteerd wordt. Bij Bash wordt er geen nieuw bestand aangemaakt; alleen de permissies van het script veranderen.

Take relevant screenshots of the following commands:

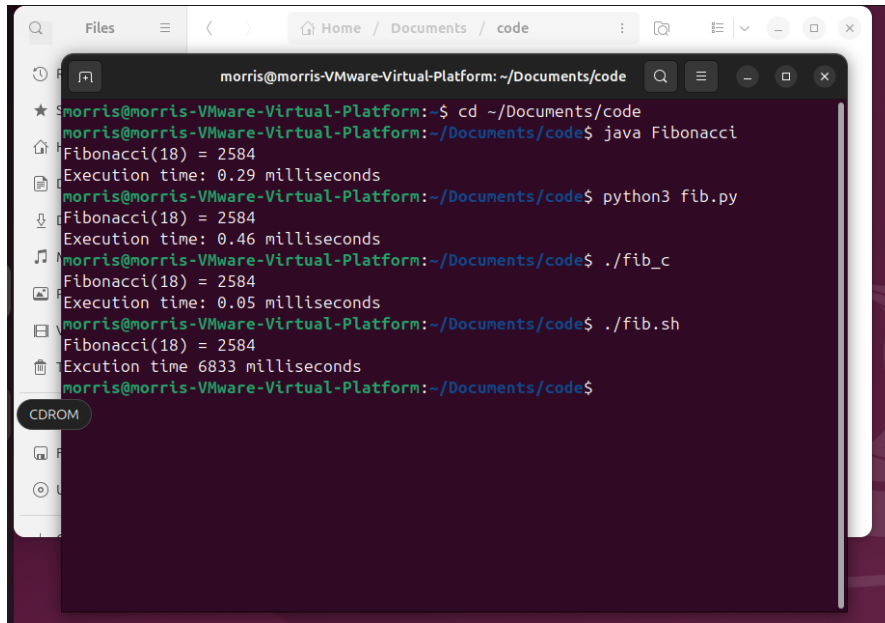
- Compile the source files where necessary



- Make them executable



- Run them



A terminal window titled 'morris@morris-VMware-Virtual-Platform: ~/Documents/code' displays the execution of four different Fibonacci programs. Each program calculates the 18th Fibonacci number, which is 2584. The execution times are as follows:

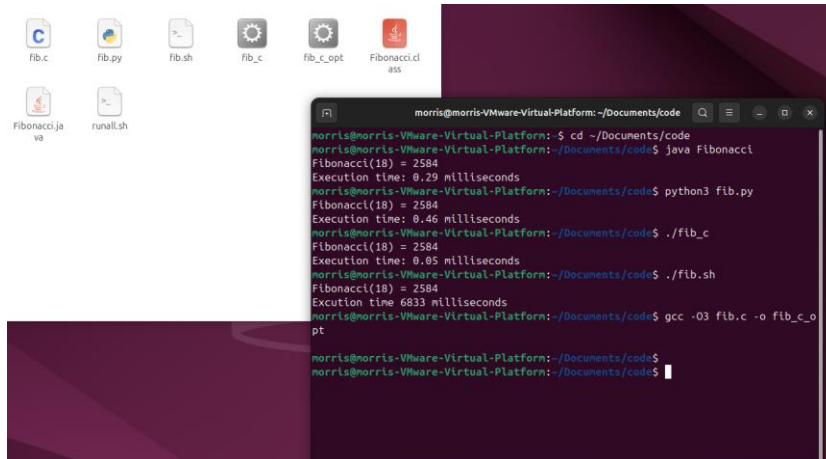
Program	Execution Time
java Fibonacci	0.29 milliseconds
python3 fib.py	0.46 milliseconds
./fib_c	0.05 milliseconds
./fib.sh	6833 milliseconds

- Which (compiled) source code file performs the calculation the fastest?
-C-programma is het snelst (0.05 ms)

Assignment 4.4: Optimize

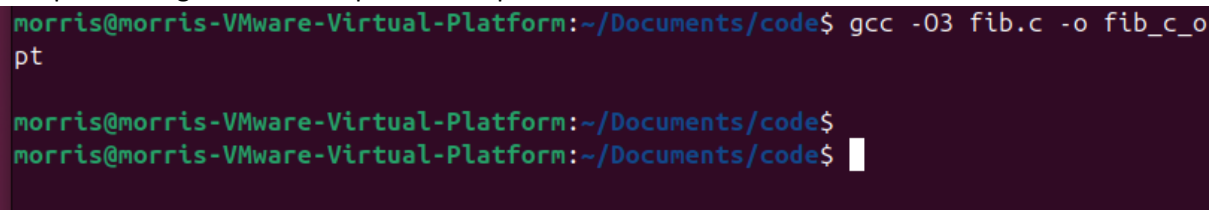
Take relevant screenshots of the following commands:

- 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.



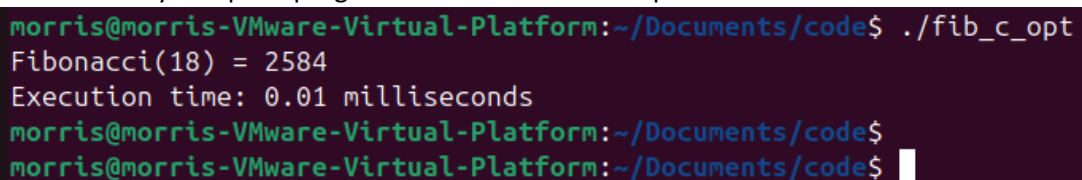
```
morris@morris-VMware-Virtual-Platform: ~/Documents/code
morris@morris-VMware-Virtual-Platform: ~/Documents/code$ java Fibonacci
Fibonacci(18) = 2584
Execution time: 0.29 milliseconds
morris@morris-VMware-Virtual-Platform: ~/Documents/code$ python3 fib.py
Fibonacci(18) = 2584
Execution time: 0.46 milliseconds
morris@morris-VMware-Virtual-Platform: ~/Documents/code$ ./fib_c
Fibonacci(18) = 2584
Execution time: 0.05 milliseconds
morris@morris-VMware-Virtual-Platform: ~/Documents/code$ ./fib.sh
Fibonacci(18) = 2584
Execution time: 6833 milliseconds
morris@morris-VMware-Virtual-Platform: ~/Documents/code$ gcc -O3 fib.c -o fib_c_opt
morris@morris-VMware-Virtual-Platform: ~/Documents/code$
```

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



```
morris@morris-VMware-Virtual-Platform: ~/Documents/code$ gcc -O3 fib.c -o fib_c_opt
morris@morris-VMware-Virtual-Platform: ~/Documents/code$
morris@morris-VMware-Virtual-Platform: ~/Documents/code$
```

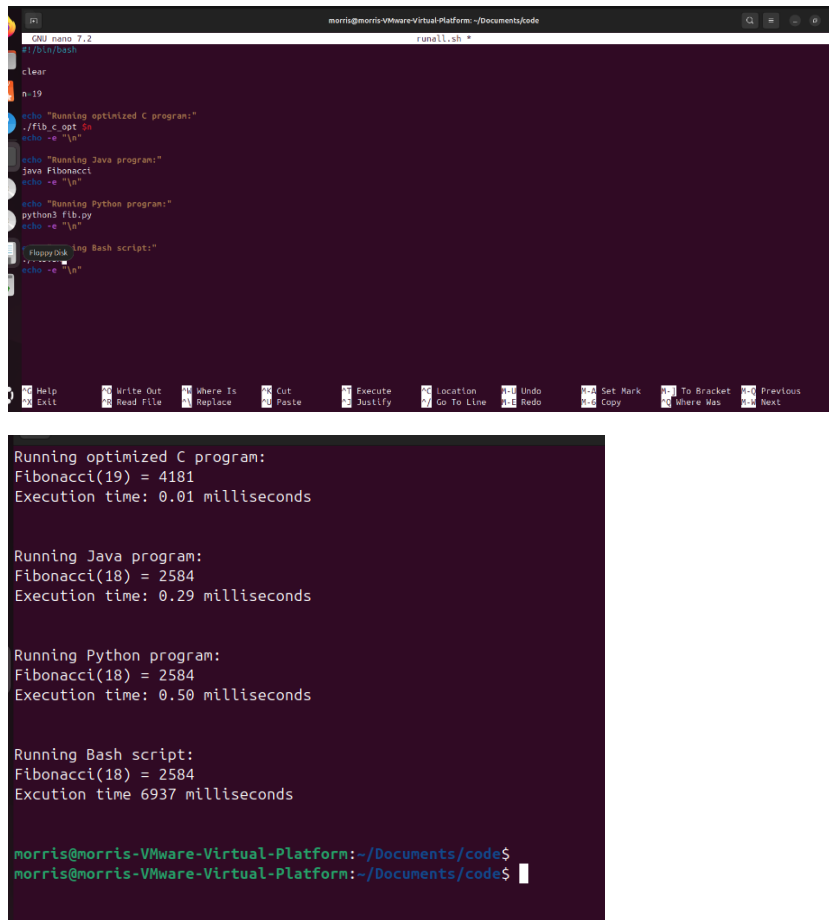
- c) Run the newly compiled program. Is it true that it now performs the calculation faster?



```
morris@morris-VMware-Virtual-Platform: ~/Documents/code$ ./fib_c_opt
Fibonacci(18) = 2584
Execution time: 0.01 milliseconds
morris@morris-VMware-Virtual-Platform: ~/Documents/code$
morris@morris-VMware-Virtual-Platform: ~/Documents/code$
```

Dus sneller dan de vorige: 0.04 milliseconden sneller

- d) 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.



```
GNU nano 7.2 runall.sh
#!/bin/bash

clear

n=19

echo "Running optimized C program:"
./fib_c_opt.sh
echo -e "\n"

echo "Running Java program:"
java Fibonacci
echo -e "\n"

echo "Running Python program:"
python3 fib.py
echo -e "\n"

echo "Running Bash script:"
./fib_bash.sh
echo -e "\n"

morris@morris-VMware-Virtual-Platform:~/Documents/code$
morris@morris-VMware-Virtual-Platform:~/Documents/code$
```

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

Running Java program:
Fibonacci(18) = 2584
Execution time: 0.29 milliseconds

Running Python program:
Fibonacci(18) = 2584
Execution time: 0.50 milliseconds

Running Bash script:
Fibonacci(18) = 2584
Execution time 6937 milliseconds

morris@morris-VMware-Virtual-Platform:~/Documents/code$
morris@morris-VMware-Virtual-Platform:~/Documents/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 OakSim emulator interface. On the left, the assembly code is displayed with line numbers 1 through 14. The code includes instructions for moving values into registers, a loop, and a comparison. On the right, the Register Value table shows the state of registers R0 through R6. Below the registers, a memory dump shows hexadecimal values for addresses from 0x00010000 to 0x00010130.

```
1 Main:
2   mov r1, #2
3   mov r2, #4
4   mov r0, #1
5
6 Loop:
7   mul r0, r0, r1
8   sub r2, r2, #1
9   cmp r2, #0
10  bne Loop
11
12 End:
13   @ Resultaat (16) staat in r0
14
```

Register	Value
R0	10
R1	2
R2	0
R3	0
R4	0
R5	0
R6	0

Memory dump (hexadecimal values):

```
0x00010000: 02 10 A0 E3 04 20 A0 E3 01 01 A0 E3 90 01 01 E0 ...
0x00010010: 01 20 42 E2 00 00 52 E3 FB FF 1A 00 00 00 00 ...
0x00010020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0x00010030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0x00010040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0x00010050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0x00010060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0x00010070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0x00010080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0x00010090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0x000100A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0x000100B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0x000100C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0x000100D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0x000100E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0x000100F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0x00010100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0x00010110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0x00010120: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0x00010130: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
```

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