

# Template Week 4 – Software

Student number: 585902

## Assignment 4.1: ARM assembly

Screenshot of working assembly code of factorial calculation:

The screenshot shows the OakSim interface. On the left, there's a sidebar with a character icon, a search icon, and a plus sign. In the center, there's a code editor window titled "OakSim" containing the following ARM assembly code:

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

To the right of the code editor is a register table titled "Register Value". It lists the values of registers R0 through R10. Below the register table is a memory dump section showing hex values from 0x000010000 to 0x000010090.

Register	Value
R0	0
R1	78
R2	0
R3	0
R4	0
R5	0
R6	0
R7	0
R8	0
R9	0
R10	0

Address	Value
0x000010000	05 20 A0 E3 05 10 A0 E3 01 20
0x000010010	01 00 00 0A 91 02 01 E0 FA FF
0x000010020	00 00 00 00 00 00 00 00 00 00 00
0x000010030	00 00 00 00 00 00 00 00 00 00 00
0x000010040	00 00 00 00 00 00 00 00 00 00 00
0x000010050	00 00 00 00 00 00 00 00 00 00 00
0x000010060	00 00 00 00 00 00 00 00 00 00 00
0x000010070	00 00 00 00 00 00 00 00 00 00 00
0x000010080	00 00 00 00 00 00 00 00 00 00 00
0x000010090	00 00 00 00 00 00 00 00 00 00 00
0x0000100A0	00 00 00 00 00 00 00 00 00 00 00

## Assignment 4.2: Programming languages

Take screenshots that the following commands work:

javac --version

java --version

gcc --version

python3 --version

bash --version

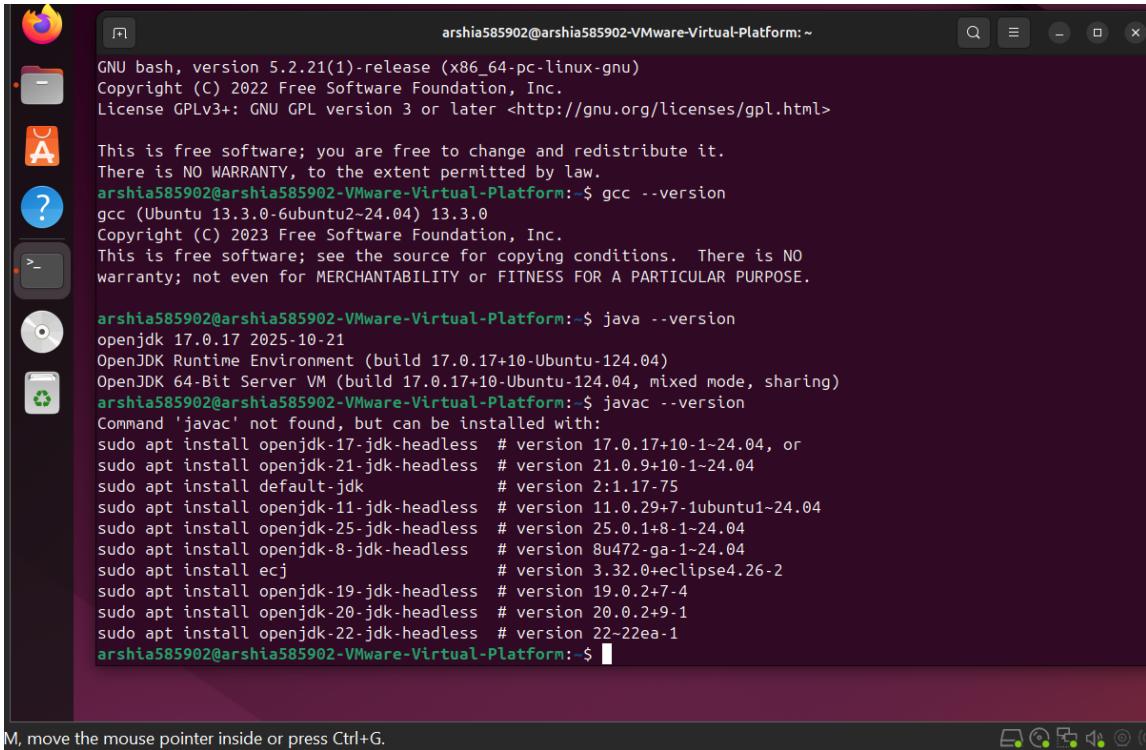
The screenshot shows a terminal window with the following command outputs:

```
Adding debian:USERTrust_ECC_Certification_Authority.pem
Adding debian:USERTrust_RSA_Certification_Authority.pem
Adding debian:vTrus_ECC_Root_CA.pem
Adding debian:vTrus_Root_CA.pem
Adding debian:XRamp_Global_CA_Root.pem
done.

arshia585902@arshia585902-VMware-Virtual-Platform:~$ python3 --version
Python 3.12.3
arshia585902@arshia585902-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.
arshia585902@arshia585902-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.

arshia585902@arshia585902-VMware-Virtual-Platform:~$
```

A screenshot of a Linux desktop environment, likely Ubuntu, showing a terminal window. The terminal window has a dark purple background and contains the following text:

```
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.
arshia585902@arshia585902-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.

arshia585902@arshia585902-Virtual-Platform:~$ java --version
openjdk 17.0.17 2025-10-21
OpenJDK Runtime Environment (build 17.0.17+10-Ubuntu-124.04)
OpenJDK 64-Bit Server VM (build 17.0.17+10-Ubuntu-124.04, mixed mode, sharing)
arshia585902@arshia585902-Virtual-Platform:~$ javac --version
Command 'javac' not found, but can be installed with:
sudo apt install openjdk-17-jdk-headless # version 17.0.17+10-1~24.04, or
sudo apt install openjdk-21-jdk-headless # version 21.0.9+10-1~24.04
sudo apt install default-jdk # version 2:1.17-75
sudo apt install openjdk-11-jdk-headless # version 11.0.29+7~ubuntu1~24.04
sudo apt install openjdk-25-jdk-headless # version 25.0.1+8-1~24.04
sudo apt install openjdk-8-jdk-headless # version 8u472 ga-1~24.04
sudo apt install ecj # version 3.32.0+eclipse4.26-2
sudo apt install openjdk-19-jdk-headless # version 19.0.2+7-4
sudo apt install openjdk-20-jdk-headless # version 20.0.2+9-1
sudo apt install openjdk-22-jdk-headless # version 22-22ea-1
arshia585902@arshia585902-Virtual-Platform:~$
```

### Assignment 4.3: Compile

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

The files in C and Java language need to be compiled first

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

The file in C language is compiled to native machine code

Which source code files are compiled to byte code?

The file in Java is first compiled to bytecode and then run by Java virtual machine

Which source code files are interpreted by an interpreter?

The files in python and bash are executed line by line by a runtime interpreter

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

C because after being compiled into machine code it directly communicates with the CPU.  
Then comes Java and after that are python and bash since they need to be interpreted line by line; with bash being even slower with the extra barrier of the shell in between

How do I run a Java program?

Using the javac compiler I first create a 'main' class through a bytecode file like this: javac Main.java and then I run that main as such: java Main(referring to the name of the newly created class file) since its not directly executed by the os afterwards

How do I run a Python program?

I call python3 interpreter on it. It does make a bytecode like in java, but it doesn't run manually because that byte code is handled in the cache rather than being visible by the user so although no "visible" additional file is created, but it still takes longer to run

How do I run a C program?

Using the gcc compiler I create a compiled version of the file which is basically a separate file which then I can store and run anywhere I want and it goes like:

```
gcc ./example.c -o customName
```

How do I run a Bash script?

I use bash runtime for it as such: bash name (this way its run by directly invoking bash interpreter and passing the script to it as an argument). In this approach the script doesn't need execute permissions because bash explicitly interprets it

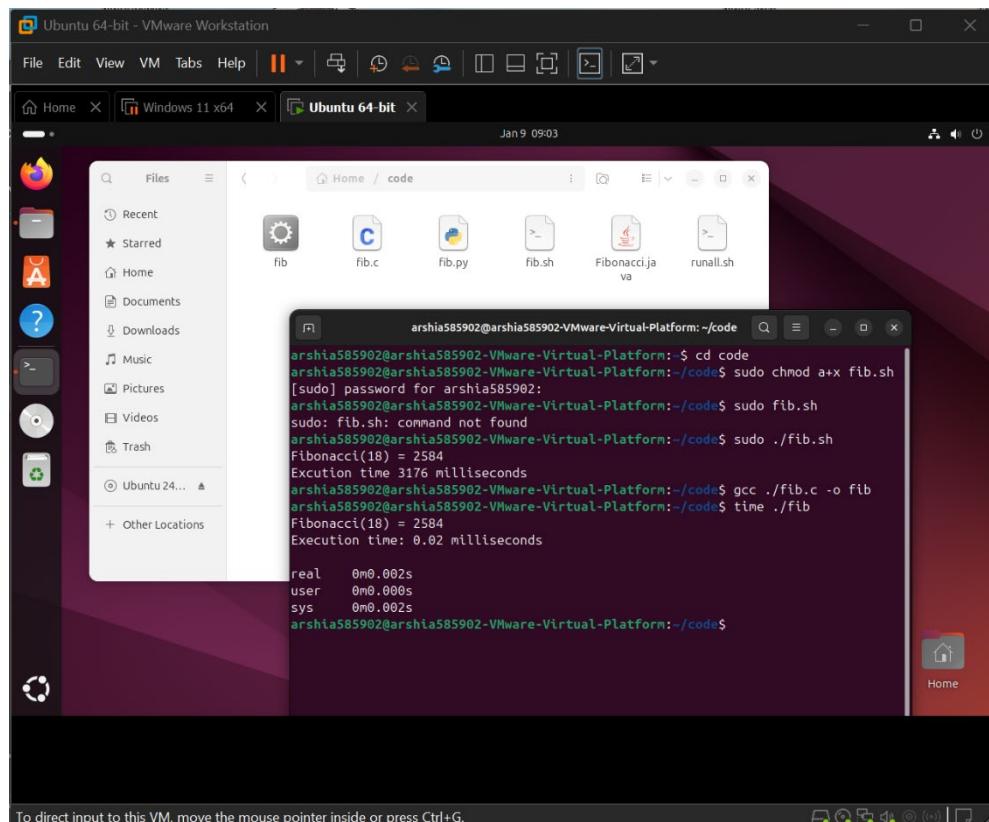
Alternatively, it can be run as an executable file. after making sure that the file has execute permissions using 'chmod +x', I can then use a shebang like so `#!/bin/bash` to specify the shell

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

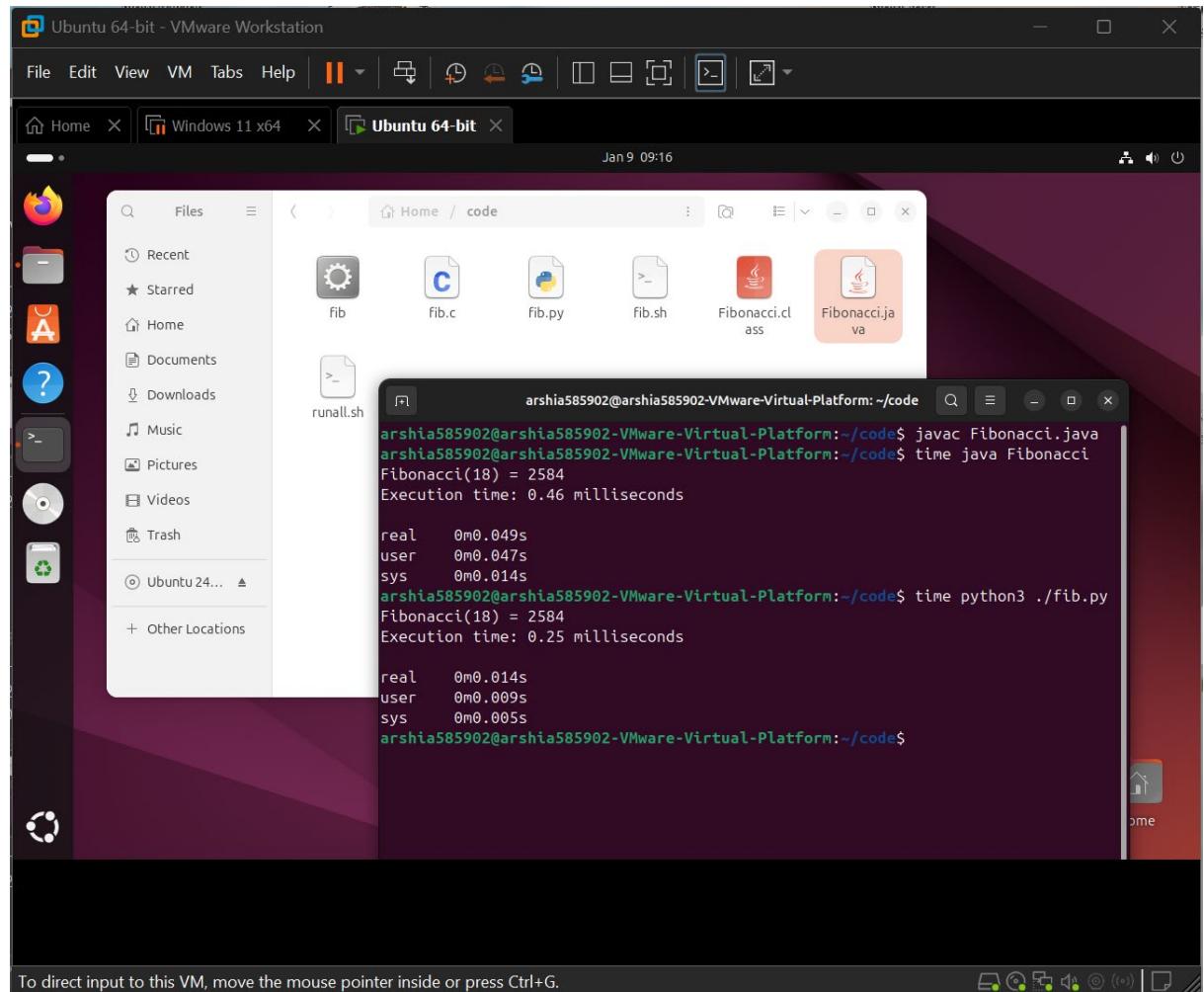
For compiled ones like java and c yes but for the interpreted ones such as py and bash no. for c that file is a direct machine code written in hex groups for java it's the main class which appears in the directory as soon as javac compiler compiles it

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? C



### Comparison between Bash and C regarding time and run method (slowest vs Fastest)



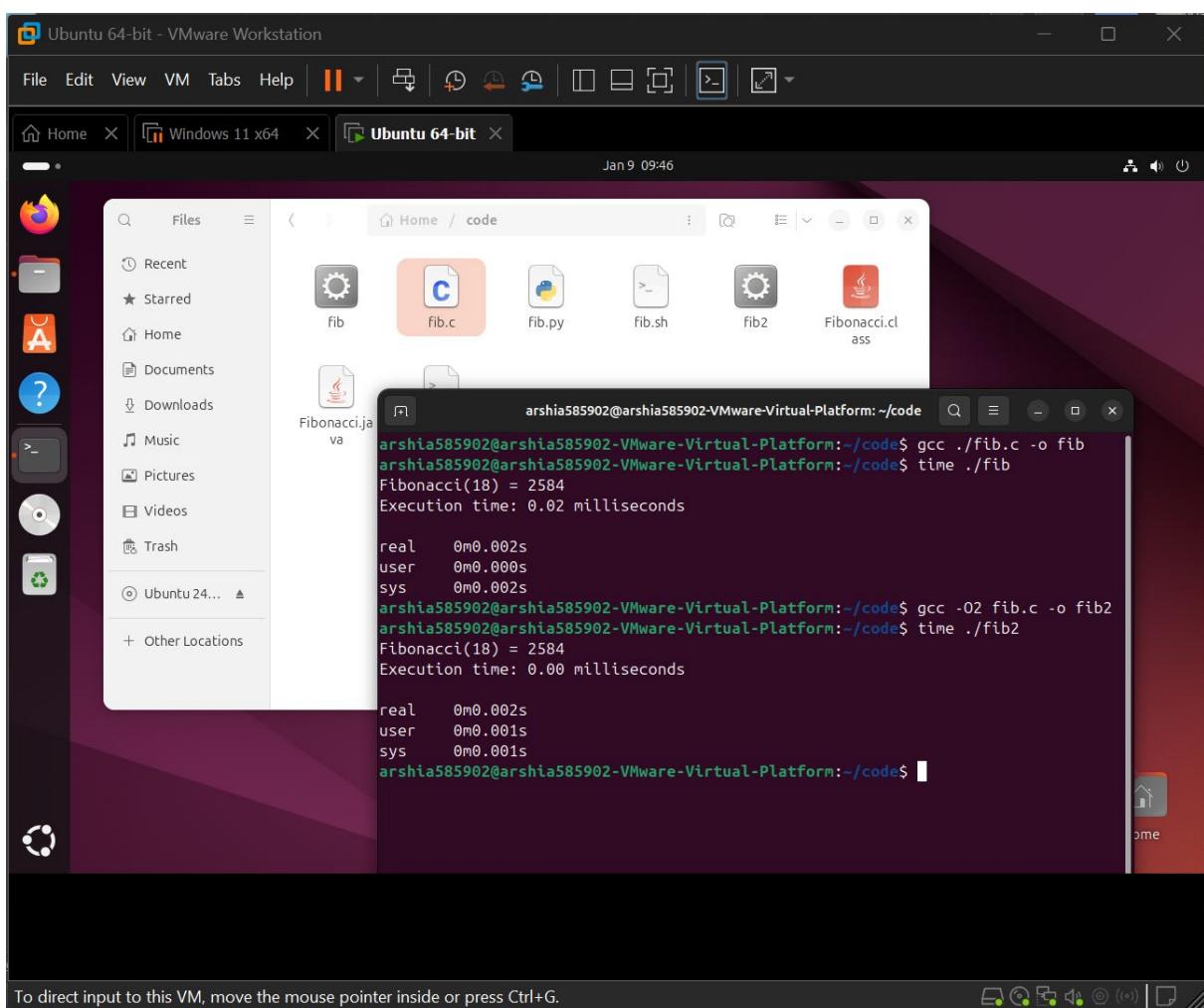
### Compariton between Java and Python regarding the method to run and the time it takes

As the times show, the C file was the fastest since it turned into machine code upon being compiled, and unlike bash it didn't have an extra shell overhead. Unlike java it didn't have to create an additional class to run

#### 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.
  
- b) Compile **fib.c** again with the optimization parameters
  
- c) Run the newly compiled program. Is it true that it now performs the calculation faster?



**Explanation below :**

In this test I compiled it twice with two different names since that's a feature of c compiler which lets the creation of individual compiled files. I measured their run time in the same window so that by comparison it would be apparent that the optimized version now runs even quicker. Here I used -O2 optimization option; however, There are other optimization options as well like O3 but its not always guaranteed to be quicker

- 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 other.

The screenshot shows a VMware Workstation interface with three windows:

- Ubuntu 64-bit - VMware Workstation**: The main window showing the desktop environment of the Ubuntu 64-bit VM.
- Windows 11 x64**: A minimized window showing the Windows taskbar.
- Ubuntu 64-bit**: An open terminal window on the Ubuntu desktop.

The terminal window content is as follows:

```
Fibonacci(19) = 4181
Execution time: 0.04 milliseconds

Running Java program:
Fibonacci(19) = 4181
Execution time: 0.36 milliseconds

Running Python program:
Fibonacci(19) = 4181
Execution time: 0.35 milliseconds

Running BASH Script
Fibonacci(19) = 4181
Execution time 6458 milliseconds

real    0m6.545s
user    0m4.447s
sys     0m2.705s
arshia585902@arshia585902-VMware-Virtual-Platform:~/code$
```

To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

### 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 debugger interface. The assembly code window contains the following code:

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

The register values window shows the following state:

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

The memory dump window shows the first 16 bytes of memory starting at address 0x00010000:

0x00010000:	02 10 A0 E3 04 20 A0 E3 01 00
0x00010010:	01 20 42 E2 00 00 52 E3 00 00
0x00010020:	00 00 00 00 00 00 00 00 00 00
0x00010030:	00 00 00 00 00 00 00 00 00 00
0x00010040:	00 00 00 00 00 00 00 00 00 00
0x00010050:	00 00 00 00 00 00 00 00 00 00
0x00010060:	00 00 00 00 00 00 00 00 00 00
0x00010070:	00 00 00 00 00 00 00 00 00 00
0x00010080:	00 00 00 00 00 00 00 00 00 00
0x00010090:	00 00 00 00 00 00 00 00 00 00
0x000100A0:	00 00 00 00 00 00 00 00 00 00
0x000100B0:	00 00 00 00 00 00 00 00 00 00
0x000100C0:	00 00 00 00 00 00 00 00 00 00
0x000100D0:	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](#)