

# Week 4 – Software

Student number: 576255

## Assignment 4.1: ARM assembly

Screenshot of working assembly code of factorial calculation:

The screenshot shows a debugger interface with the following details:

- Registers:** A table showing register values:

Register	Value
R0	1
R1	78
R2	0
R3	0
R4	0
R5	0
R6	0
R7	0
R8	0
R9	0
R10	0
- Memory Dump:** Hex dump of memory starting at address 0x000010000, showing the factorial result (78) in memory.
- Assembly Code:** The assembly code for calculating factorial is displayed in the main window:

```
1    MOV R0, #5
2    MOV R1, #1
3
4    loop:
5    MUL R1, R0, R1
6    SUB R0, R0, #1
7
8    CMP R0, #1
9    BNE loop
10
11 stop:
12 B stop
```

## Assignment 4.2: Programming languages

Take screenshots that the following commands work:

The terminal window displays the following command outputs:

- Java:** java --version  
openjdk 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)
- Javac:** javac --version  
javac 21.0.9
- Gcc:** 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.
- Python3:** python3 --version  
Python 3.12.3
- Bash:** 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>>

A message at the bottom states: This is free software; you are free to change and redistribute it. There is NO WARRANTY, to the extent permitted by law.

## Installed compilers and interpreters on my Ubuntu 24.04 VM

javac version: [21.0.9]

java runtime version: [21.0.9]

gcc version: [13.3.0]

python3 version: [3.12.3]

bash version: [5.2.21]

### Assignment 4.3: Compile

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

- **Fibonacci.java** and **fib.c** need to be compiled before they can be run. **fib.py** and **fib.sh** do not need compilation

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

- The C source file **fib.c** is compiled by **gcc** into native machine code that the processor can execute directly

Which source code files are compiled to byte code?

- The Java source file **Fibonacci.java** is compiled by **javac** into Java bytecode stored in a .class file that runs on the Java Virtual Machine

Which source code files are interpreted by an interpreter?

- **fib.py** is interpreted by the Python interpreter **python3** and **fib.sh** is interpreted by the Bash shell

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

- The C program from **fib.c** is expected to be the fastest because it is compiled to optimized native machine code with very little overhead during execution.

How do I run a Java program?

1. Compile the source file: **javac Fibonacci.java**
2. Run the compiled program: **java Fibonacci**

How do I run a Python program?

- Use the Python interpreter: **python3 fib.py**

How do I run a C program?

1. Compile the source file, for example: **gcc fib.c -o fib**
2. Run the compiled binary: **./fib**

How do I run a Bash script?

- First make the script executable, then run it:
  - **chmod a+x fib.sh**
  - **./fib.sh**

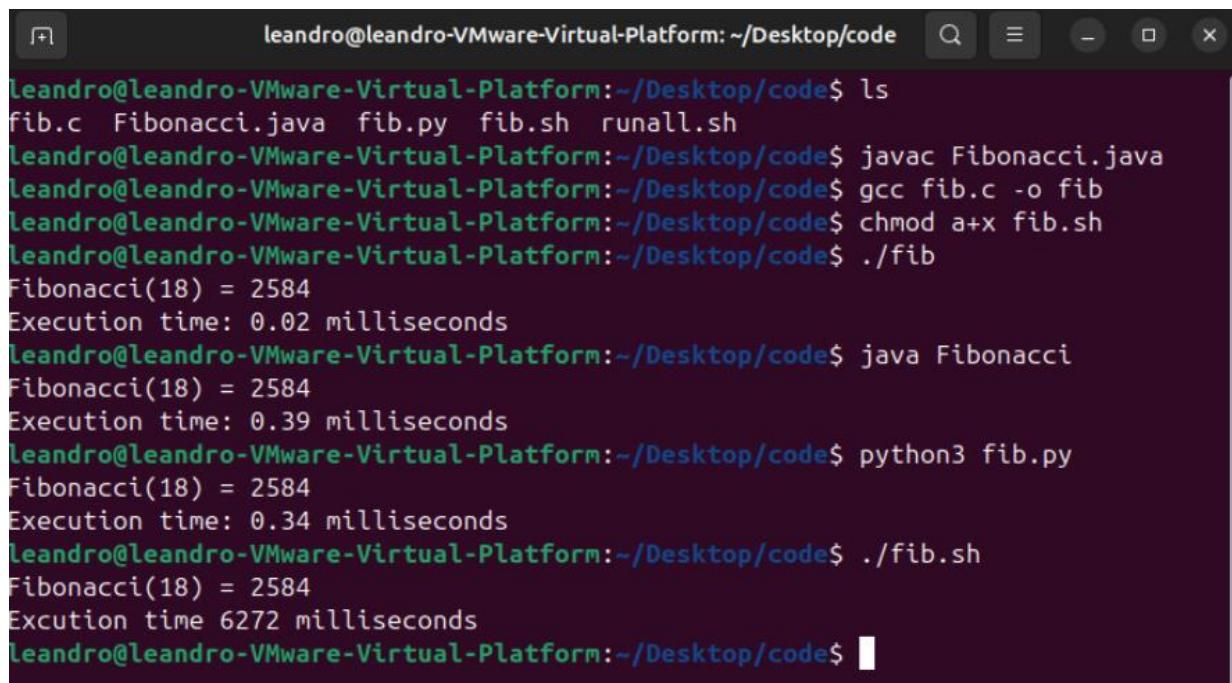
If I compile the above source code, will a new file be created? If so, which file?

- Yes. Compiling **Fibonacci.java** creates one or more **.class** files such as **Fibonacci.class**. Compiling **fib.c** with **gcc fib.c -o fib** creates an executable file named **fib**

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?

On my system the **C program (fib) compiled with gcc** performed the Fibonacci calculation the fastest.



The screenshot shows a terminal window with a dark background and light-colored text. The title bar reads "leandro@leandro-Virtual-Platform: ~/Desktop/code". The terminal output is as follows:

```
leandro@leandro-Virtual-Platform:~/Desktop/code$ ls
fib.c Fibonacci.java fib.py fib.sh runall.sh
Leandro@leandro-Virtual-Platform:~/Desktop/code$ javac Fibonacci.java
Leandro@leandro-Virtual-Platform:~/Desktop/code$ gcc fib.c -o fib
Leandro@leandro-Virtual-Platform:~/Desktop/code$ chmod a+x fib.sh
Leandro@leandro-Virtual-Platform:~/Desktop/code$ ./fib
Fibonacci(18) = 2584
Execution time: 0.02 milliseconds
Leandro@leandro-Virtual-Platform:~/Desktop/code$ java Fibonacci
Fibonacci(18) = 2584
Execution time: 0.39 milliseconds
Leandro@leandro-Virtual-Platform:~/Desktop/code$ python3 fib.py
Fibonacci(18) = 2584
Execution time: 0.34 milliseconds
Leandro@leandro-Virtual-Platform:~/Desktop/code$ ./fib.sh
Fibonacci(18) = 2584
Execution time: 6272 milliseconds
Leandro@leandro-Virtual-Platform:~/Desktop/code$
```

#### 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.

The GCC manual shows that the **-O** options enable different optimization levels. I chose to use **-O3**, which turns on a large set of aggressive optimizations for speed. The **-O3** flag makes the compiler unroll loops, inline more functions and perform additional code transformations to make the generated machine code faster

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

In the terminal I used:

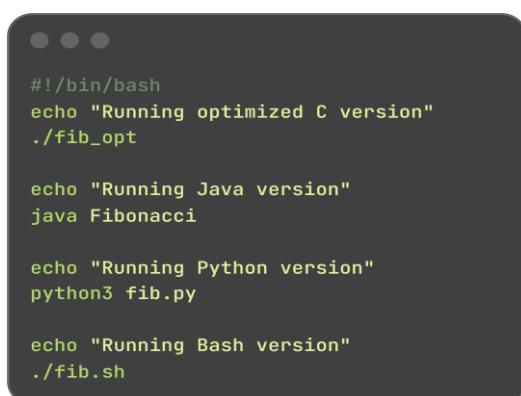
```
gcc -O3 fib.c -o fib_opt
```

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

After running **./fib\_opt** and comparing the execution time with the original **./fib** program, the optimized version ran faster and finished the Fibonacci calculation sooner, which confirms that the optimizations had a positive effect

- 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.

I edited **runall.sh** so that it runs the optimized C program, the Java program, the Python program and the Bash script one after another:



```
#!/bin/bash
echo "Running optimized C version"
./fib_opt

echo "Running Java version"
java Fibonacci

echo "Running Python version"
python3 fib.py

echo "Running Bash version"
./fib.sh
```

After saving this file I made it executable with **chmod a+x runall.sh** and then ran it with **./runall.sh**

```
leandro@leandro-Virtual-Platform:~
```

-fsel-sched-pipeline -fshrink-wrap -fshrink-wrap-separate  
-fsemantic-interposition -fsingle-precision-constant  
-fsignal-nans -fsplit-loops -fsplit-paths  
-fsplit-wide-types -fsplit-wide-types-early -fssa-backprop  
-fssa-phipopt -fstdarg-opt -fstore-merging -fstrict-aliasing  
-fipa-strict-aliasing -fthread-jumps -ftracer -ftree-bit-ccp  
-ftree-built-in-call-dce -ftree-ccp -ftree-ch -ftree-coalesce-vars  
-ftree-copy-prop -ftree-dce -ftree-dominator-opts -ftree-dse  
-ftree-forwprop -ftree-fre -fcode-hoisting -ftree-loop-if-convert  
-ftree-loop-im -ftree-phiprop -ftree-loop-distribution  
-ftree-loop-distribute-patterns -ftree-loop-ivcanon  
-ftree-loop-linear -ftree-loop-optimize -ftree-loop-vectorize  
-ftree-paralleize-loops=- -ftree-pre -ftree-partial-pre  
-ftree-pta -ftree-reassoc -ftree-scev-cprop -ftree-sink  
-ftree-slur -ftree-sra -ftree-switch-conversion -ftree-tail-merge  
-ftree-ter -ftree-vectorize -ftree-vrp -ftrivial-auto-var-init  
-funconstrained-commons -funit-at-a-time -funroll-all-loops  
-funroll-loops -funsafe-math-optimizations -funswitch-loops  
-fipa-ra -fvariable-expansion-in-unroller -fvect-cost-model  
-fvpt -fweb -fwhole-program -fwpa -fuse-linker-plugin  
-fzero-call-used-reg -param name=value -O0 -O0 -O1 -O2 -O3  
-Os -Ofast -Og -Oz

Program Instrumentation Options

```
p -pg -fprofile-arcs --coverage -ftest-coverage  
-fprofile-abs-path -fprofile-dir=path -fprofile-generate  
-fprofile-generate=path -fprofile-info-section  
-fprofile-info-section=name -fprofile-note=path  
-fprofile-prefix-path=path -fprofile-update=method  
-fprofile-filter-files=regex -fprofile-exclude-files=regex
```

Manual page gcc(1) line 372 (press h for help or q to quit)

```
leandro@leandro-Virtual-Platform:~/Desktop/code$ gcc -O3 fib.c -o fib_opt  
leandro@leandro-Virtual-Platform:~/Desktop/code$ ./fib_opt  
Fibonacci(18) = 2584  
Execution time: 0.01 milliseconds  
leandro@leandro-Virtual-Platform:~/Desktop/code$ time ./fib  
Fibonacci(18) = 2584  
Execution time: 0.02 milliseconds  
  
real    0m0.004s  
user    0m0.002s  
sys     0m0.002s  
leandro@leandro-Virtual-Platform:~/Desktop/code$ time ./fib_opt  
Fibonacci(18) = 2584  
Execution time: 0.01 milliseconds  
  
real    0m0.003s  
user    0m0.002s  
sys     0m0.001s  
leandro@leandro-Virtual-Platform:~/Desktop/code$
```

```
leandro@leandro-VMware-Virtual-Platform:~/Desktop/code$ nano runall.sh
leandro@leandro-VMware-Virtual-Platform:~/Desktop/code$ chmod a+x runall.sh
leandro@leandro-VMware-Virtual-Platform:~/Desktop/code$ run
runcon      runqlat-bpfcc    runqlen.bt      run-with-aspell
runlevel     runqlat.bt      runqlen.bpfcc
run-parts    runqlen-bpfcc   runuser
leandro@leandro-VMware-Virtual-Platform:~/Desktop/code$ run
runcon      runqlat-bpfcc    runqlen.bt      run-with-aspell
runlevel     runqlat.bt      runqlen.bpfcc
run-parts    runqlen-bpfcc   runuser
leandro@leandro-VMware-Virtual-Platform:~/Desktop/code$ ./runall.sh
Running optimized C version
Fibonacci(18) = 2584
Execution time: 0.01 milliseconds
Running Java version
Fibonacci(18) = 2584
Execution time: 0.33 milliseconds
Running Python version
Fibonacci(18) = 2584
Execution time: 0.35 milliseconds
Running Bash version
Fibonacci(18) = 2584
Execution time 6669 milliseconds
leandro@leandro-VMware-Virtual-Platform:~/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 a debugger interface with the following components:

- Top Bar:** Open, Run, 250, Step, Reset.
- Assembly Code:**

```
1 Main:  
2     MOV r0, #1  
3     MOV r1, #2  
4     MOV r2, #4  
5       
6 Loop:  
7     MUL r0, r1  
8     SUBS r2, #1  
9     BNE Loop  
10      
11 End:  
12     B End
```
- Registers:** A table showing register values:

Register	Value
R0	10
R1	2
R2	0
R3	0
R4	0
R5	0
R6	0
R7	0
R8	0
R9	0
R10	0
- Memory Dump:** A hex dump of memory starting at address 0x00010000. The first few lines are:

Address	Value
0x00010000	01 20 E3 02 10 20 E3 04 20 E3 90 01 00 E0
0x00010010	01 20 52 E2 FC FF FF 1A FF FF EA
0x00010020	00 00 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 00 00
0x00010040	00 00 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 00 00
0x00010060	00 00 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 00 00
0x00010080	00 00 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 00 00
0x000100A0	00 00 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 00 00

My program uses a loop to multiply the running result (in R0) by the base value 2 (in R1). The exponent counter (R2) is decremented each iteration using SUBS, and BNE repeats the loop until R2 reaches zero. At the end of the loop, R0 contains 0x10, which is 16 in decimal, representing  $2^4$ .

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