

# Template Week 4 – Software

Student number: 569091

## Assignment 4.1: ARM assembly

Screenshot of working assembly code of factorial calculation:

The screenshot shows the OakSim ARM simulator interface. The left pane displays the assembly code for a factorial calculation. The right pane shows the register values and a memory dump.

**Assembly Code:**

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

**Register Values:**

Register	Value
R0	0
R1	78
R2	1
R3	0
R4	0
R5	0
R6	0
R7	0
R8	0
R9	0
R10	0
R11	0
R12	0
SP	10000
LR	0
PC	10018
CPSR	60000013

**Memory Dump:**

Address	Value
0x00010000	05 20 A0 E3 01 10 A0 E3 91 02 01 E0 01 20 42 E2
0x00010010	01 00 52 E3 FB FF FF 1A FE FF FF EA
0x00010020	...
0x00010030	...
0x00010040	...
0x00010050	...
0x00010060	...
0x00010070	...
0x00010080	...
0x00010090	...
0x000100A0	...
0x000100B0	...
0x000100C0	...
0x000100D0	...
0x000100E0	...
0x000100F0	...
0x00010100	...
0x00010110	...
0x00010120	...
0x00010130	...
0x00010140	...
0x00010150	...
0x00010160	...
0x00010170	...
0x00010180	...
0x00010190	...
0x000101A0	...
0x000101B0	...
0x000101C0	...
0x000101D0	...
0x000101E0	...
0x000101F0	...
0x00010200	...
0x00010210	...
0x00010220	...
0x00010230	...
0x00010240	...
0x00010250	...
0x00010260	...
0x00010270	...
0x00010280	...
0x00010290	...
0x000102A0	...
0x000102B0	...
0x000102C0	...
0x000102D0	...
0x000102E0	...
0x000102F0	...
0x00010300	...
0x00010310	...
0x00010320	...
0x00010330	...
0x00010340	...
0x00010350	...

**Stack Trace:**

```
abort() at Error at jsStackTrace
(https://wunkolo.github.io/OakSim/lib/unicorn-
arm.min.js:5:18821) at stackTrace
(https://wunkolo.github.io/OakSim/lib/unicorn-
arm.min.js:5:18992) at Object.abort
(https://wunkolo.github.io/OakSim/lib/unicorn-
arm.min.js:29:7211) at abort
(https://wunkolo.github.io/OakSim/lib/unicorn-
arm.min.js:5:200323) at Array.Sb
(https://wunkolo.github.io/OakSim/lib/unicorn-
arm.min.js:16:17608) at Object.TMa [as dynCall_iii]
(https://wunkolo.github.io/OakSim/lib/unicorn-
arm.min.js:9:170658) at invoke_iii
(https://wunkolo.github.io/OakSim/lib/unicorn-
arm.min.js:9:170658) at invoke_iii
```

## Assignment 4.2: Programming languages

Take screenshots that the following commands work:

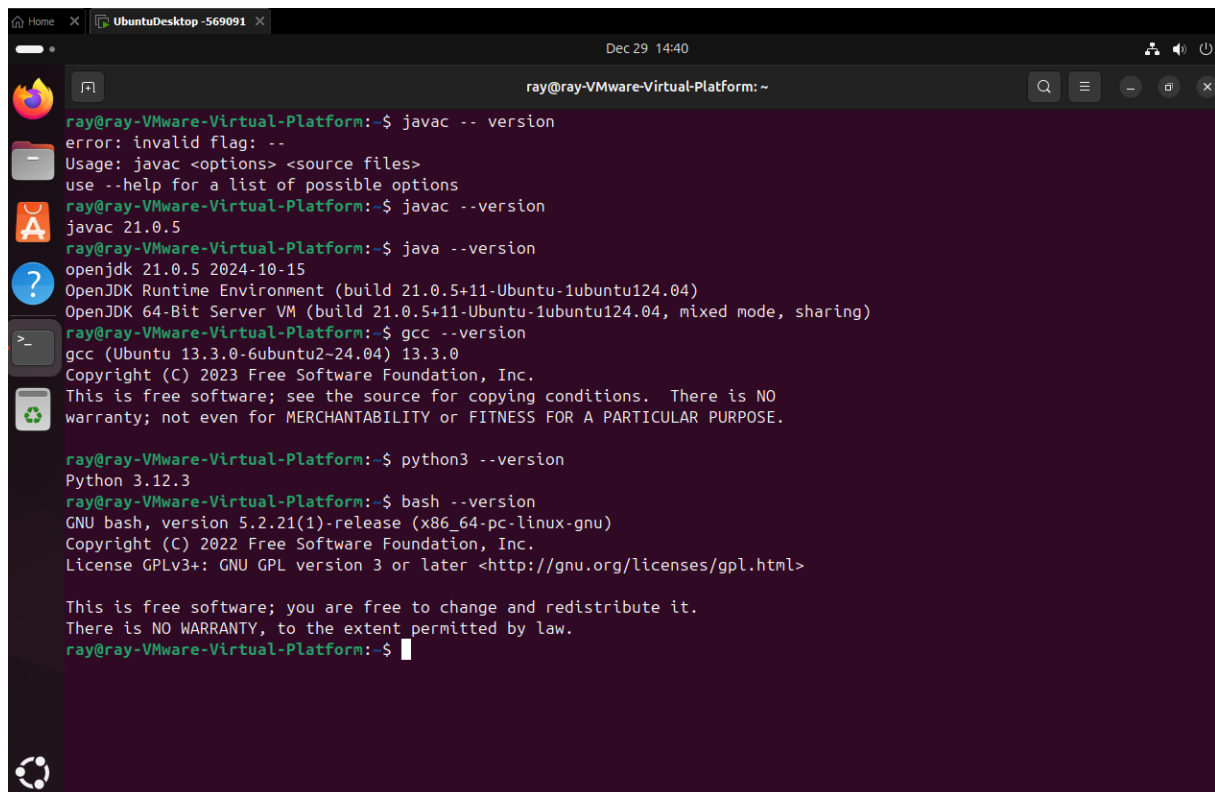
`javac --version`

`java --version`

`gcc --version`

`python3 --version`

`bash --version`



The screenshot shows a terminal window titled "ray@ray-VMware-Virtual-Platform: ~" with a dark background. The terminal displays the following commands and their outputs:

```
ray@ray-VMware-Virtual-Platform:~$ javac -- version
error: invalid flag: --
Usage: javac <options> <source files>
use --help for a list of possible options
ray@ray-VMware-Virtual-Platform:~$ javac --version
javac 21.0.5
ray@ray-VMware-Virtual-Platform:~$ java --version
openjdk 21.0.5 2024-10-15
OpenJDK Runtime Environment (build 21.0.5+11-Ubuntu-1ubuntu124.04)
OpenJDK 64-Bit Server VM (build 21.0.5+11-Ubuntu-1ubuntu124.04, mixed mode, sharing)
ray@ray-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.

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

### Assignment 4.3: Compile

```
ray@ray-VMware-Virtual-Platform:~$ ls ~/Downloads/code
fib.c  Fibonacci.java  fib.py  fib.sh  runall.sh
ray@ray-VMware-Virtual-Platform:~$
```



fib.c



fib.py



fib.sh



Fibonacci.java



runall.sh

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

#### Files that require compiling:

- Fibonacci.java (Java source code).
- fib.c (C source code).

#### Files that do not need compilation:

- fib.py (Python source code is interpreted by an interpreter).
- fib.sh (Bash script executed by the shell interpreter).

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

- fib.c (compiled to a binary file, executable by the processor).

Which source code files are compiled to byte code?

- Fibonacci.java (compiled to Java bytecode, typically in a .class file, executed by the Java Virtual Machine).

Which source code files are interpreted by an interpreter?

- fib.py (interpreted by the Python interpreter).
- fib.sh (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?

- fib.c (C source code) is expected to perform the fastest because it is compiled into machine code and executed directly by the processor without additional layers like an interpreter or virtual machine.

### How do I run a Java program?

Compile the Java file: `javac Fibonacci.java`

This creates a Fibonacci.class file.

Run the compiled Java program: `Java Fibonacci`

### How do I run a Python program?

Run the Python program using the Python interpreter: `python3 fib.py`

### How do I run a C program?

Compile the C file using a compiler like gcc: `gcc fib.c -o fib`

Run the compiled program: `./fib`

### How do I run a Bash script?

Make the script executable: `chmod a+x fib.sh`

Run the script: `./fib.sh`

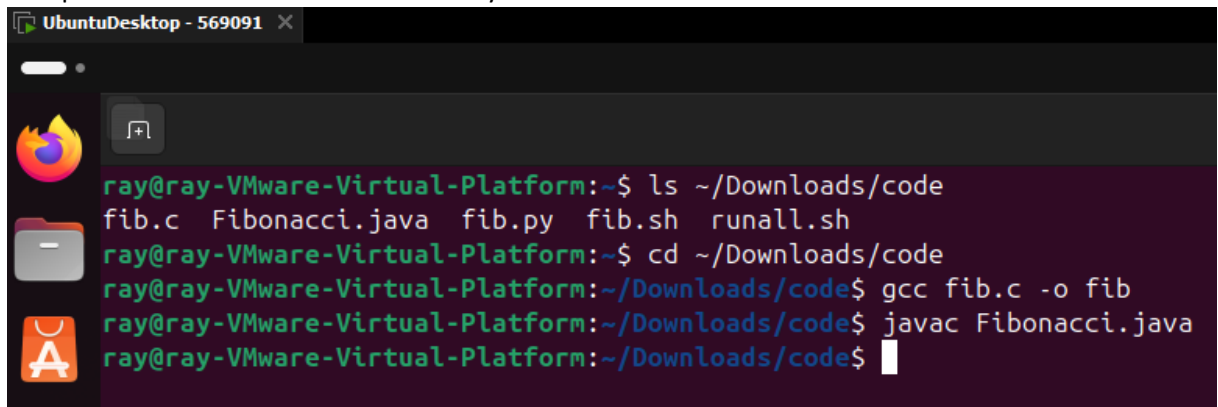
### If I compile the above source code, will a new file be created? If so, which file?

Yes, compiling the source code creates new files:

- **Fibonacci.java**: Creates Fibonacci.class.
- **fib.c**: Creates fib (or another name you specify with -o during compilation).
- **fib.py and fib.sh**: Do not create compiled files as they are interpreted.

### Take relevant screenshots of the following commands:

- Compile the source files where necessary



```
UbuntuDesktop - 569091 x
ray@ray-VMware-Virtual-Platform:~$ ls ~/Downloads/code
fib.c  Fibonacci.java  fib.py  fib.sh  runall.sh
ray@ray-VMware-Virtual-Platform:~$ cd ~/Downloads/code
ray@ray-VMware-Virtual-Platform:~/Downloads/code$ gcc fib.c -o fib
ray@ray-VMware-Virtual-Platform:~/Downloads/code$ javac Fibonacci.java
ray@ray-VMware-Virtual-Platform:~/Downloads/code$
```

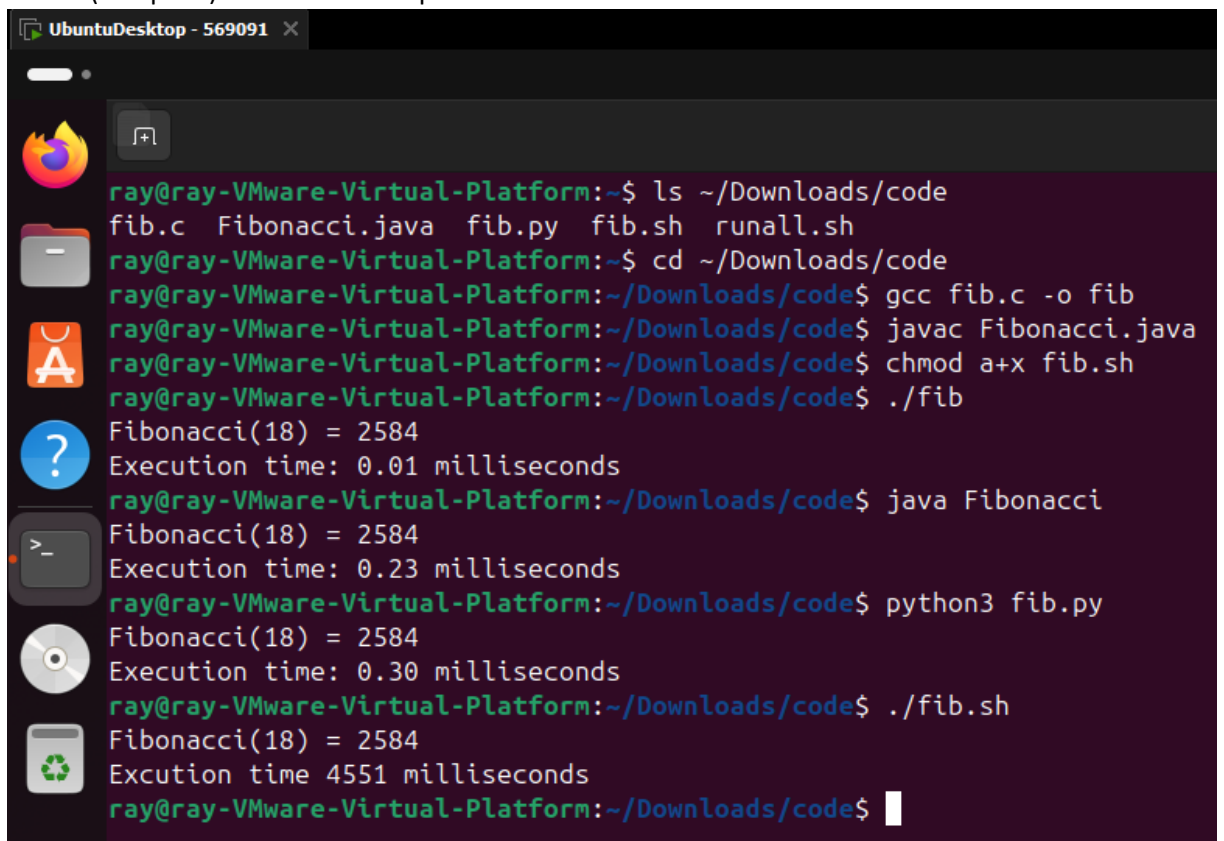
- Make them executable

```
ray@ray-VMware-Virtual-Platform:~/Downloads/code$ chmod a+x fib.sh
```

- Run them

```
ray@ray-VMware-Virtual-Platform:~/Downloads/code$ ./fib
Fibonacci(18) = 2584
Execution time: 0.01 milliseconds
ray@ray-VMware-Virtual-Platform:~/Downloads/code$ java Fibonacci
Fibonacci(18) = 2584
Execution time: 0.23 milliseconds
ray@ray-VMware-Virtual-Platform:~/Downloads/code$ python3 fib.py
Fibonacci(18) = 2584
Execution time: 0.30 milliseconds
ray@ray-VMware-Virtual-Platform:~/Downloads/code$ ./fib.sh
Fibonacci(18) = 2584
Execution time 4551 milliseconds
ray@ray-VMware-Virtual-Platform:~/Downloads/code$
```

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



```

ray@ray-VMware-Virtual-Platform:~$ ls ~/Downloads/code
fib.c  Fibonacci.java  fib.py  fib.sh  runall.sh
ray@ray-VMware-Virtual-Platform:~$ cd ~/Downloads/code
ray@ray-VMware-Virtual-Platform:~/Downloads/code$ gcc fib.c -o fib
ray@ray-VMware-Virtual-Platform:~/Downloads/code$ javac Fibonacci.java
ray@ray-VMware-Virtual-Platform:~/Downloads/code$ chmod a+x fib.sh
ray@ray-VMware-Virtual-Platform:~/Downloads/code$ ./fib
Fibonacci(18) = 2584
Execution time: 0.01 milliseconds
ray@ray-VMware-Virtual-Platform:~/Downloads/code$ java Fibonacci
Fibonacci(18) = 2584
Execution time: 0.23 milliseconds
ray@ray-VMware-Virtual-Platform:~/Downloads/code$ python3 fib.py
Fibonacci(18) = 2584
Execution time: 0.30 milliseconds
ray@ray-VMware-Virtual-Platform:~/Downloads/code$ ./fib.sh
Fibonacci(18) = 2584
Execution time 4551 milliseconds
ray@ray-VMware-Virtual-Platform:~/Downloads/code$

```

Fib.c performs the calculation the fastest

## 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**: Enables aggressive optimizations to maximize performance, but may increase compile time and executable size.

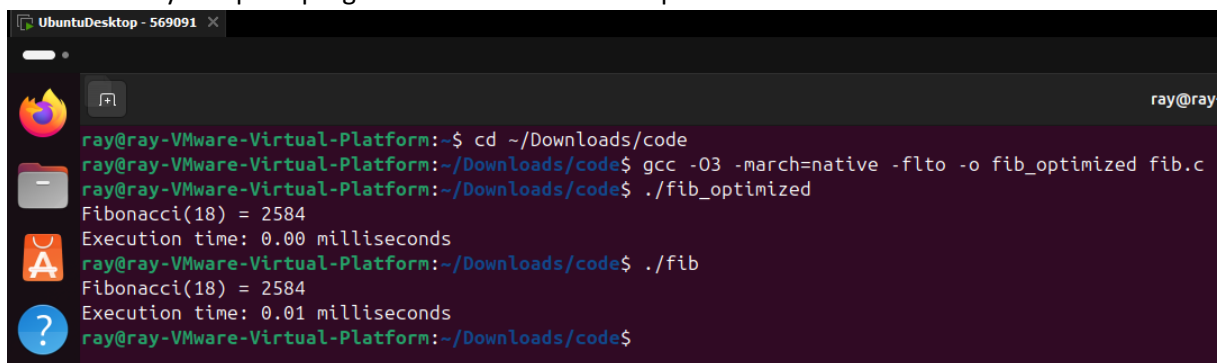
**march=native**: Optimizes the program for the architecture of the machine it's compiling on

**flto**: Enables Link Time Optimization, which can provide even better performance.

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

**gcc -O3 -march=native -flto -o fib\_optimized fib.c**

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



```
ray@ray-VMware-Virtual-Platform:~$ cd ~/Downloads/code
ray@ray-VMware-Virtual-Platform:~/Downloads/code$ gcc -O3 -march=native -flto -o fib_optimized fib.c
ray@ray-VMware-Virtual-Platform:~/Downloads/code$ ./fib_optimized
Fibonacci(18) = 2584
Execution time: 0.00 milliseconds
ray@ray-VMware-Virtual-Platform:~/Downloads/code$ ./fib
Fibonacci(18) = 2584
Execution time: 0.01 milliseconds
ray@ray-VMware-Virtual-Platform:~/Downloads/code$
```

yes

- 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.

Swapped **./fib \$n** with **./fib\_optimized \$n** within **runall.sh**

Provide **runall.sh** with permissions: **chmod +x runall.sh**

Run: **./runall.sh**

The screenshot shows an Ubuntu desktop environment with a terminal window titled "UbuntuDesktop - 569091". The terminal has a dark purple background and displays the results of running Fibonacci programs in different languages. On the left side of the terminal, there is a vertical dock with icons for Firefox, a file manager, the Ubuntu Software Center, an unknown application (blue circle with a question mark), a terminal, a CD/DVD drive, and a trash can. The terminal output is as follows:

```
Running C program:
Fibonacci(19) = 4181
Execution time: 0.02 milliseconds

Running Java program:
Fibonacci(19) = 4181
Execution time: 0.27 milliseconds

Running Python program:
Fibonacci(19) = 4181
Execution time: 0.48 milliseconds

Running BASH Script
Fibonacci(19) = 4181
Excution time 7399 milliseconds

ray@ray-VMware-Virtual-Platform:~/Downloads/code$
```

### Bonus point assignment – week 4

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:

Completed code:

Main:

```
mov r0, #1
```

```
mov r1, #2
```

```
mov r2, #4
```

Loop:

```
cmp r2, #0
```

```
beq End
```

```
mul r0, r0, r1
```

```
subs r2, r2, #1
```

```
b Loop
```

End:

```
b End
```

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

Screenshot of the completed code here.



Open

Run

250

Step

Reset

```

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

```

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
LR	0
PC	10020
CPSR	60000013

0x00010000: 01 00 A0 E3 02 10 A0 E3 04 20 A0 E3 00 00 52 E3 ...
0x00010010: 02 00 00 0A 90 01 00 E0 01 20 52 E2 FA FF FF EA ...
0x00010020: FF FF FF EA 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](#)