

## **Template Week 4 – Software**

Student number: 588963

## Assignment 4.1: ARM assembly

Screenshot of working assembly code of factorial calculation:

The screenshot shows the OakSim debugger interface. The assembly code window displays the following code:

```
1 Main: //588963 Tony Jarwa
2     mov r1, #1
3     mov r2, #5
4
5     Loop:
6         mul r1, r1, r2
7         sub r2, r2, #1
8         cmp r2, #1
9         b Loop
10
11
12
```

The register window shows the following 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

The memory dump window shows memory starting at address 0x0000100001. The first few bytes are:

Address	Value
0x0000100001	01 10 A0 E3 05 20 A0 E3 91 02 01 E0 01 20 42 82
0x0000100010	01 00 S2 B3 00 00 0A FA FF FF EA 00 00 00 00 00 00 00 00
0x0000100020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000100030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000100040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000100050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000100060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000100070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000100080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000100090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00001000A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00001000B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00001000C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00001000D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00001000E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00001000F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000100100	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000100120	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000100140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000100150	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000100170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000100180	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000100190	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00001001A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00001001B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00001001C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00001001D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00001001F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000100210	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

The status bar at the bottom indicates "© 2017".

78 in Hex is 120 in Decimal.

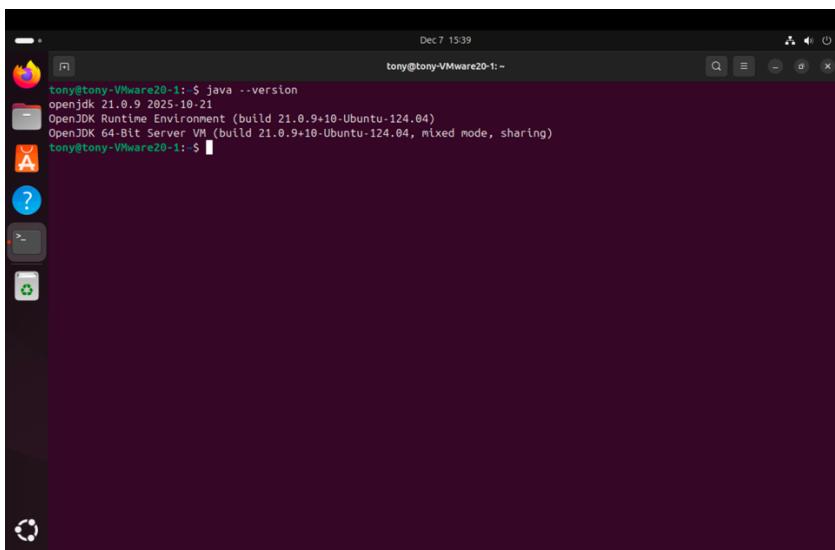
## Assignment 4.2: Programming languages

Take screenshots that the following commands work:

javac –version

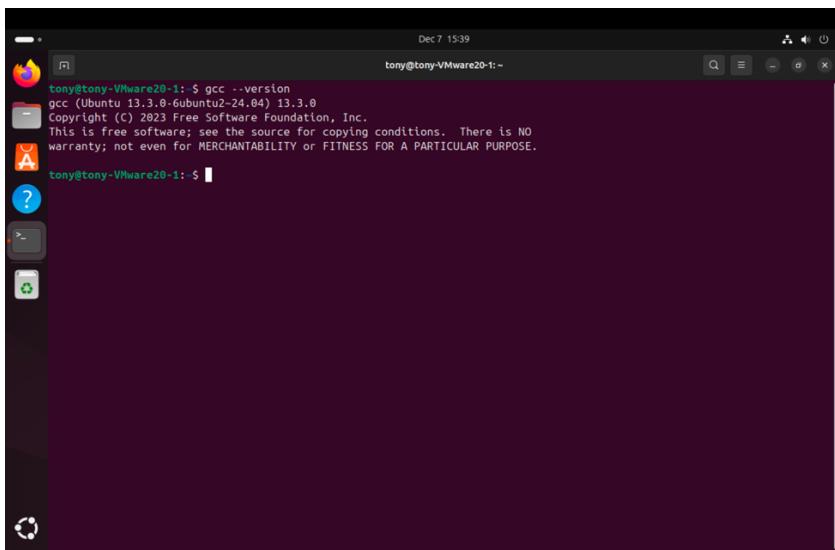
```
Dec 7 15:37
tony@tony-VirtualBox:~$ javac --version
javac 21.0.9
tony@tony-VirtualBox:~$
```

`java --version`



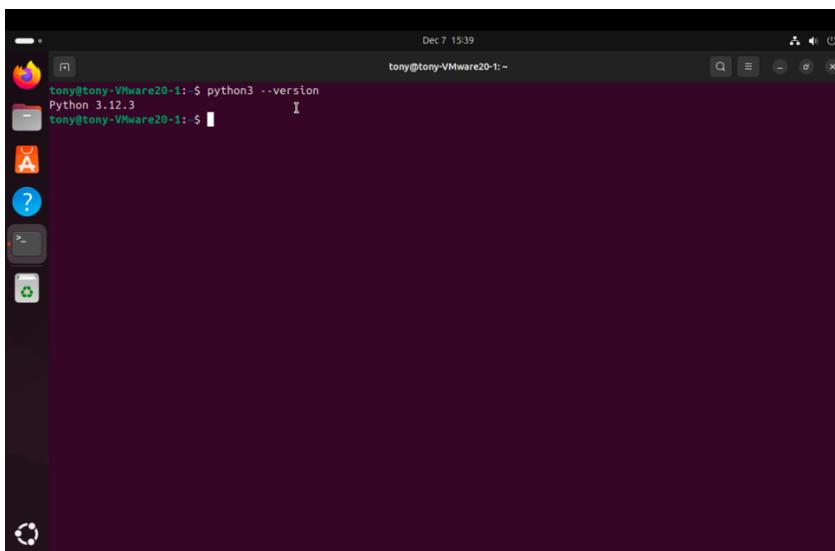
```
Dec 7 15:39 tony@tony-VMware20-1:~$ java --version
openjdk 21.0.9 2025-10-21
OpenJDK Runtime Environment (build 21.0.9+10-Ubuntu-124.04)
OpenJDK 64-Bit Server VM (build 21.0.9+10-Ubuntu-124.04, mixed mode, sharing)
tony@tony-VMware20-1:~$
```

`gcc --version`



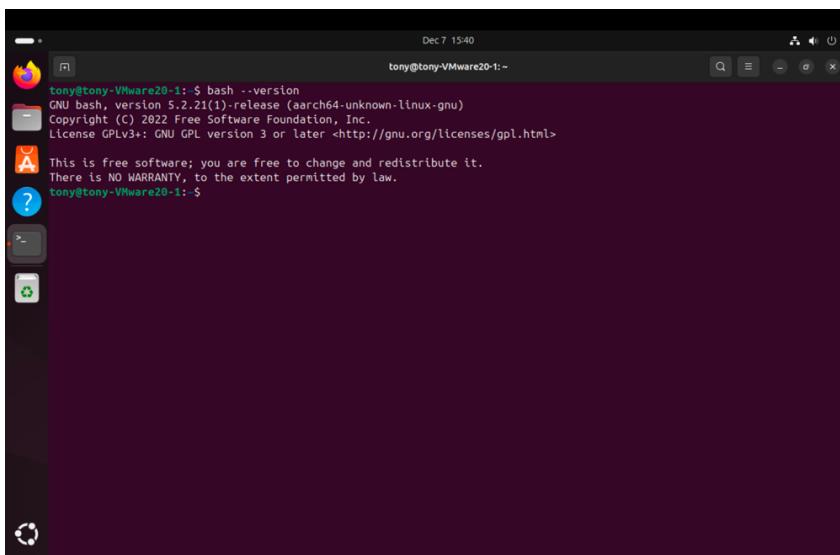
```
Dec 7 15:39 tony@tony-VMware20-1:~$ gcc --version
gcc (Ubuntu 13.3.0-6ubuntu2-24.04) 13.3.0
Copyright (C) 2023 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
tony@tony-VMware20-1:~$
```

`python3 --version`



```
Dec 7 15:39 tony@tony-VMware20-1:~$ python3 --version
Python 3.12.3
tony@tony-VMware20-1:~$
```

```
bash --version
```



The screenshot shows a terminal window titled 'tony@tony-VMware20-1: ~'. The window displays the output of the 'bash --version' command. The output includes the version number (GNU bash, version 5.2.21(1)-release (aarch64-unknown-linux-gnu)), copyright information (Copyright (C) 2022 Free Software Foundation, Inc.), license details (License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>), and a note about being free software with no warranty.

```
tony@tony-VMware20-1: ~$ bash --version
GNU bash, version 5.2.21(1)-release (aarch64-unknown-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.
tony@tony-VMware20-1: ~$
```

### 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](#)

Which source code files are compiled to byte code?

[Fibonacci.java](#)

Which source code files are interpreted by an interpreter?

[fib.py](#)

[fib.sh](#)

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

The fib.c (C program) is expected to be the fastest one because it is compiled directly into machine code.

How do I run a Java program?

[javac Fibonacci.java](#)

[java Fibonacci](#)

How do I run a Python program?

```
python3 fib.py
```

How do I run a C program?

```
gcc fib.c -o fib
```

```
./fib
```

How do I run a Bash script?

```
chmod a+x fib.sh
```

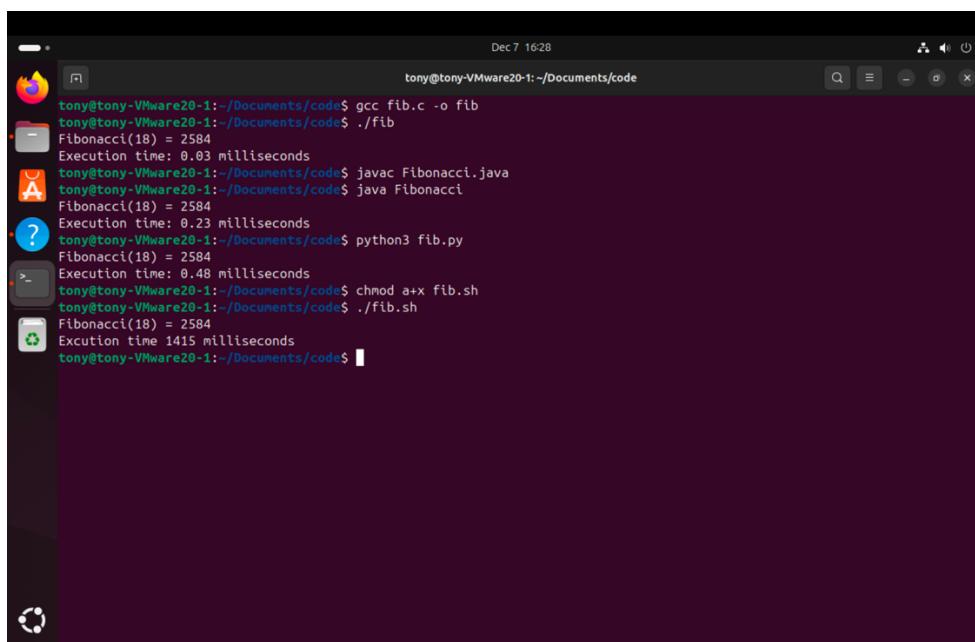
```
./fib.sh
```

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

Yes, by Fibonacci.java a new file (Fibonacci.class), and fib.c (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?



```
tony@tony-VMware20-1:~/Documents/code$ gcc fib.c -o fib
tony@tony-VMware20-1:~/Documents/code$ ./fib
Fibonacci(18) = 2584
Execution time: 0.03 milliseconds
tony@tony-VMware20-1:~/Documents/code$ javac Fibonacci.java
tony@tony-VMware20-1:~/Documents/code$ java Fibonacci
Fibonacci(18) = 2584
Execution time: 0.23 milliseconds
tony@tony-VMware20-1:~/Documents/code$ python3 fib.py
Fibonacci(18) = 2584
Execution time: 0.48 milliseconds
tony@tony-VMware20-1:~/Documents/code$ chmod a+x fib.sh
tony@tony-VMware20-1:~/Documents/code$ ./fib.sh
Fibonacci(18) = 2584
Execution time 1415 milliseconds
tony@tony-VMware20-1:~/Documents/code$
```

The C program (fib.c) is the fastest (0,03 milliseconds).

## 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.

```
Dec 7 22:19
tony@tony-VirtualBox: ~/Documents/code

-falign-functions -falign-jumps -falign-labels -falign-loops -fcaller-saves -fcode-hoisting
-fcrossjumping -fcse-follow-jumps -fcse-skip-blocks -fdelete-null-pointer-checks -fdevirtualize
-fdevirtualize-speculatively -fexpensive-optimizations -ffinite-loops -fgcse -fgcse-lm
-fhoist-adjacent-loads -finline-functions -finline-small-functions -finredirect-inlining -fipa-bit-cp
-fipa-cp -fipa-icf -fipa-ra -fipa-sra -fipa-vrp -fisolate-errorneous-paths-dereference -fira-remat
-foptimize-sibling-calls -foptimize-strlen -fpartial-inlining -fpeephole2 -freorder-blocks-algorithm=stc
-freorder-blocks-and-partition -freorder-functions -frerun-cse-after-loop -fschedule-insns
-fschedule-insns2 -fsched-interblock -fsched-spec -fstore-merging -fstrict-aliasing -fthread-jumps
-ftree-built-in-call-dce -ftree-loop-vectorize -ftree-pr -ftree-slp-vectorize -ftree-switch-conversion
-ftree-tail-merge -ftree-vrp -fvect-cost-model=very-cheap

Please note the warning under -fgcse about invoking -O2 on programs that use computed gotos.

NOTE: In Ubuntu 8.10 and later versions, -D_FORTIFY_SOURCE=2, in Ubuntu 24.04 and later versions, -D_FORTIFY_SOURCE=3, is set by default, and is activated when -O is set to 2 or higher. This enables additional compile-time and run-time checks for several libc functions. To disable, specify either -U_FORTIFY_SOURCE or -D_FORTIFY_SOURCE=0.

NOTE: In Debian 13 and Ubuntu 24.04 and later versions, -D_TIME_BITS=64 together with -D_FILE_OFFSET_BITS=64 is set by default on the 32bit architectures armel, armhf, m68k, mips, mipsel, powerpc and sh4.

-O3 Optimize yet more. -O3 turns on all optimizations specified by -O2 and also turns on the following optimization flags:

-fgcse-after-reload -fipa-cp-clone -floop-interchange -floop-unroll-and-jam -fpeel-loops
-fpredictive-commoning -fsplit-loops -fsplit-paths -ftree-loop-distribution -ftree-partial-pre
-funswitch-loops -fvect-cost-model=dynamic -fversion-loops-for-strides

-OO Reduce compilation time and make debugging produce the expected results. This is the default.

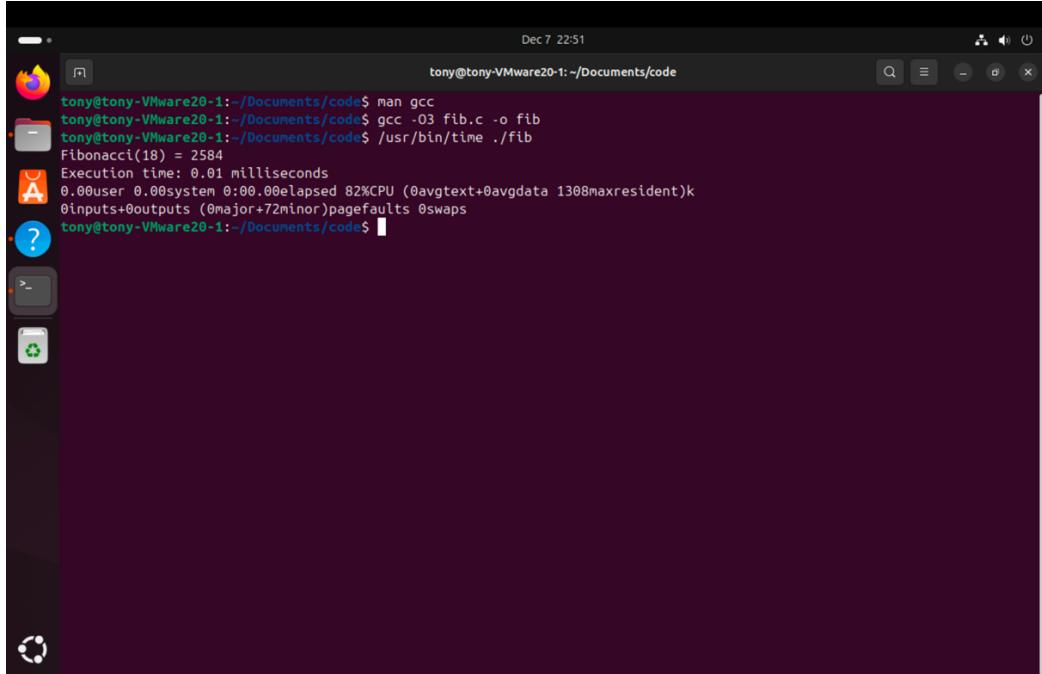
Manual page gcc(1) line 7613 (press h for help or q to quit)
```

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



```
Dec 7 22:51  
tony@tony-VMware20-1:~/Documents/code$ man gcc  
tony@tony-VMware20-1:~/Documents/code$ gcc -O3 fib.c -o fib  
tony@tony-VMware20-1:~/Documents/code$
```

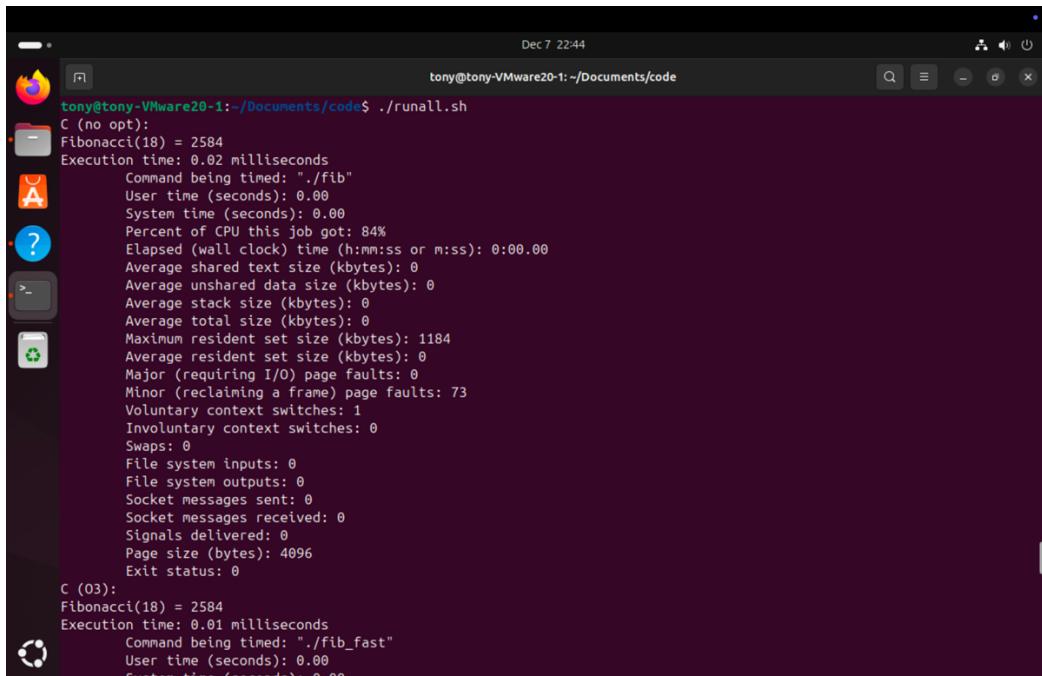
- c) Run the newly compiled program. Is it true that it now performs the calculation faster?



```
Dec 7 22:51
tony@tony-VMware20-1:~/Documents/code$ man gcc
tony@tony-VMware20-1:~/Documents/code$ gcc -O3 fib.c -o fib
tony@tony-VMware20-1:~/Documents/code$ /usr/bin/time ./fib
Fibonacci(18) = 2584
Execution time: 0.01 milliseconds
0.00user 0.00system 0:00.00elapsed 82%CPU (0avgtext+0avgdata 1308maxresident)k
0inputs+0outputs (0major+72minor)pagefaults 0swaps
tony@tony-VMware20-1:~/Documents/code$
```

It is sure faster (0,01 milliseconds).

- 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.



```
Dec 7 22:44
tony@tony-VMware20-1:~/Documents/code$ ./runall.sh
C (no opt):
Fibonacci(18) = 2584
Execution time: 0.02 milliseconds
    Command being timed: "./fib"
    User time (seconds): 0.00
    System time (seconds): 0.00
    Percent of CPU this job got: 84%
    Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.00
    Average shared text size (kbytes): 0
    Average unshared data size (kbytes): 0
    Average stack size (kbytes): 0
    Average total size (kbytes): 0
    Maximum resident set size (kbytes): 1184
    Average resident set size (kbytes): 0
    Major (requiring I/O) page faults: 0
    Minor (reclaiming a frame) page faults: 73
    Voluntary context switches: 1
    Involuntary context switches: 0
    Swaps: 0
    File system inputs: 0
    File system outputs: 0
    Socket messages sent: 0
    Socket messages received: 0
    Signals delivered: 0
    Page size (bytes): 4096
    Exit status: 0
C (O3):
Fibonacci(18) = 2584
Execution time: 0.01 milliseconds
    Command being timed: "./fib_fast"
    User time (seconds): 0.00
    System time (seconds): 0.00
```

```
Fibonacci(18) = 2584
Execution time: 0.01 milliseconds
    Command being timed: "./fib_fast"
    User time (seconds): 0.00
    System time (seconds): 0.00
    Percent of CPU this job got: 85%
    Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.00
    Average shared text size (kbytes): 0
    Average unshared data size (kbytes): 0
    Average stack size (kbytes): 0
    Average total size (kbytes): 0
    Maximum resident set size (kbytes): 1304
    Average resident set size (kbytes): 0
    Major (requiring I/O) page faults: 0
    Minor (reclaiming a frame) page faults: 72
    Voluntary context switches: 1
    Involuntary context switches: 0
    Swaps: 0
    File system inputs: 0
    File system outputs: 0
    Socket messages sent: 0
    Socket messages received: 0
    Signals delivered: 0
    Page size (bytes): 4096
    Exit status: 0
Java:
Fibonacci(18) = 2584
Execution time: 0.27 milliseconds
    Command being timed: "java Fibonacci"
    User time (seconds): 0.05
    System time (seconds): 0.01
    Percent of CPU this job got: 108%
    Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.05
Exit status: 0

```

```
Java:
Fibonacci(18) = 2584
Execution time: 0.27 milliseconds
    Command being timed: "java Fibonacci"
    User time (seconds): 0.05
    System time (seconds): 0.01
    Percent of CPU this job got: 108%
    Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.05
    Average shared text size (kbytes): 0
    Average unshared data size (kbytes): 0
    Average stack size (kbytes): 0
    Average total size (kbytes): 0
    Maximum resident set size (kbytes): 42172
    Average resident set size (kbytes): 0
    Major (requiring I/O) page faults: 0
    Minor (reclaiming a frame) page faults: 5415
    Voluntary context switches: 224
    Involuntary context switches: 12
    Swaps: 0
    File system inputs: 0
    File system outputs: 64
    Socket messages sent: 0
    Socket messages received: 0
    Signals delivered: 0
    Page size (bytes): 4096
    Exit status: 0
Python:
Fibonacci(18) = 2584
Execution time: 0.21 milliseconds
    Command being timed: "python3 fib.py"
    User time (seconds): 0.00
    System time (seconds): 0.00
    
```

The image shows two terminal windows side-by-side. Both windows have a dark background and a light-colored text area. The top window is titled "tony@tony-VMware20-1: ~/Documents/code" and the bottom window is also titled "tony@tony-VMware20-1: ~/Documents/code". Both windows show the same command-line interface output.

```

Signals delivered: 0
Page size (bytes): 4096
Exit status: 0
Python:
Fibonacci(18) = 2584
Execution time: 0.21 milliseconds
Command being timed: "python3 fib.py"
User time (seconds): 0.00
System time (seconds): 0.00
Percent of CPU this job got: 100%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.00
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 8844
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 0
Minor (reclaiming a frame) page faults: 868
Voluntary context switches: 1
Involuntary context switches: 2
Swaps: 0
File system inputs: 0
File system outputs: 0
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0
Bash:
/usr/bin/time: cannot run ./fib_sh: No such file or directory
Command exited with non-zero status 127
Command being timed: "./fib_sh"

```

The bottom window shows the same output, but with a different timestamp: "Dec 7 22:45".

#### 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 assembly debugger interface. On the left, there is a code editor window containing the following assembly code:

```
1 Main: //588963 Tony Jarwa
2     mov r0, #1
3     mov r1, #2
4     mov r2, #4
5     Loop:
6     sub r0, r0, r1
7     sub r2, r2, #1
8     cmp r0, #0
9     bne End
10    b Loop
11 End:
12 }
```

On the right, there is a register dump window titled "OakSim" showing the state of all 16 registers (R0-R15, R16) at address 0x00010000. The values are as follows:

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

The memory dump area shows the first 256 bytes of memory starting at address 0x00010000. The data is mostly zeros, with some non-zero values at higher addresses. The first few bytes are:

Address	Value
0x00010000	01 00 A0 E3 02 10 A0 E3 04 20 A0 E3 90 01 E0
0x00010001	00 20 A2 B2 00 52 B3 00 DA FA FF FF EA B R
0x00010020	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
0x00010060	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
0x000100A0	00 00 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 00 00
0x000100E0	00 00 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 00 00
0x00010120	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010160	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010180	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000101A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000101C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000101E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00010210	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

10 in Hex is 16 in Decimal.

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