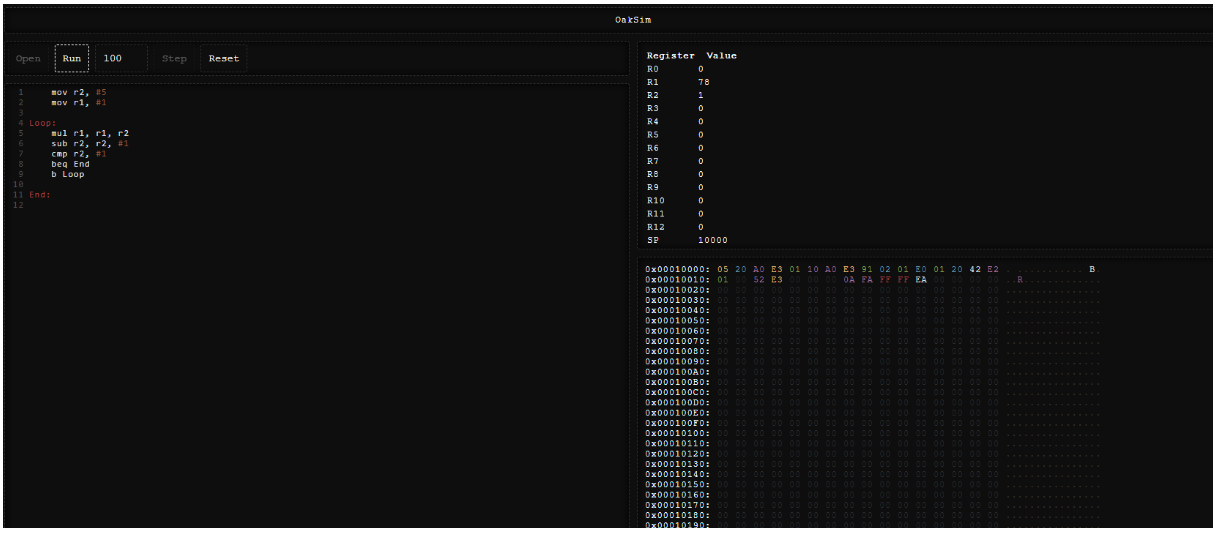
Template Week 4 – Software

Student number:

587889

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

```
wout@wout-VMware-Virtual-Platform: ~/Website2/site
wout@wout-VMware-Virtual-Platform:~/Website2/site$ javac --version
javac 21.0.9
wout@wout-VMware-Virtual-Platform:~/Website2/site$ 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)
wout@wout-VMware-Virtual-Platform:~/Website2/site$ 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.

wout@wout-VMware-Virtual-Platform:~/Website2/site$ python3 --version
Python 3.12.3
wout@wout-VMware-Virtual-Platform:~/Website2/site$ 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.
wout@wout-VMware-Virtual-Platform:~/Website2/site$
```

Assignment 4.3: Compile

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

Fib.c

Fibonacci.java

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?

Fib.c

How do I run a Java program?

java Fibonacci.java

How do I run a Python program?

python3 fib.py

How do I run a C program?

./fib

How do I run a Bash script?

./fib.sh

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

Fib.c -> fib

Fibonacci.java -> Fibonacci.class

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?**

```
wout@wout-VMware-Virtual-Platform:~/Downloads/code$ javac Fibonacci.java
wout@wout-VMware-Virtual-Platform:~/Downloads/code$ gcc fib.c -o fib
wout@wout-VMware-Virtual-Platform:~/Downloads/code$ python3 fib.py
Fibonacci(18) = 2584
Execution time: 0.36 milliseconds
wout@wout-VMware-Virtual-Platform:~/Downloads/code$ chmod a+x fib.sh
wout@wout-VMware-Virtual-Platform:~/Downloads/code$ sudo chmod a+x fib
wout@wout-VMware-Virtual-Platform:~/Downloads/code$ java Fibonacci
Fibonacci(18) = 2584
Execution time: 0.29 milliseconds
wout@wout-VMware-Virtual-Platform:~/Downloads/code$ python3 fib.py
Fibonacci(18) = 2584
Execution time: 0.41 milliseconds
wout@wout-VMware-Virtual-Platform:~/Downloads/code$ ./fib
Fibonacci(18) = 2584
Execution time: 0.03 milliseconds
wout@wout-VMware-Virtual-Platform:~/Downloads/code$ ./fib.sh

Fibonacci(18) = 2584
Excution time 5401 milliseconds
wout@wout-VMware-Virtual-Platform:~/Downloads/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.**

```
    -O1 Optimize.  Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

        With -O, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal  of  compilation
        time.

        -O turns on the following optimization flags:

        -fauto-inc-dec   -fbranch-count-reg  -fcombine-stack-adjustments  -fcompare-elim  -fcprop-registers  -fdce  -fdefer-pop  -fdelayed-branch  -fdse
        -fforward-propagate  -fguess-branch-probability  -fif-conversion  -fif-conversion2  -finline-functions-called-once  -fipa-modref   -fipa-profile
        -fipa-pure-const  -fipa-reference  -fipa-reference-addressable  -fmerge-constants -fmove-loop-invariants -fmove-loop-stores -fomit-frame-pointer
        -freorder-blocks  -fshrink-wrap  -fshrink-wrap-separate  -fsplit-wide-types  -fssa-backprop  -fssa-phiopt  -ftree-bit-ccp  -ftree-ccp  -ftree-ch
        -ftree-coalesce-vars   -ftree-copy-prop   -ftree-dce  -ftree-dominator-opts  -ftree-dse  -ftree-forwprop  -ftree-fre  -ftree-phiprop  -ftree-pta
        -ftree-scev-cprop -ftree-sink -ftree-slsr -ftree-sra -ftree-ter -funit-at-a-time

    -O2 Optimize even more.  GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff.  As compared to -O, this option
        increases both compilation time and the performance of the generated code.

        -O2 turns on all optimization flags specified by -O1.  It also turns on the following optimization flags:

        -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  -findirect-inlining  -fipa-bit-cp  -fipa-cp  -fipa-icf  -fipa-ra
        -fipa-sra  -fipa-vrp -fisolate-erroneous-paths-dereference -flra-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-builtin-call-dce  -ftree-loop-vectorize  -ftree-pre
        -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, hppa, 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
```

```
  -Os Optimize for size.  -Os enables all -O2 optimizations except those that often increase code size:

      -falign-functions  -falign-jumps -falign-labels  -falign-loops -fprefetch-loop-arrays  -freorder-blocks-algorithm=stc

      It also enables -finline-functions, causes the compiler to tune for code size rather than execution speed, and  performs  further  optimizations
      designed to reduce code size.

  -Ofast
      Disregard  strict  standards  compliance.   -Ofast  enables  all  -O3  optimizations.   It also enables optimizations that are not valid for all
      standard-compliant  programs.   It  turns  on  -ffast-math,   -fallow-store-data-races   and   the   Fortran-specific   -fstack-arrays,   unless
      -fmax-stack-var-size is specified, and -fno-protect-parens.  It turns off -fsemantic-interposition.
```

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

```
wout@wout-VMware-Virtual-Platform:~/Downloads/code$ man gcc
wout@wout-VMware-Virtual-Platform:~/Downloads/code$ gcc -O3 fib.c -o fib_opt
wout@wout-VMware-Virtual-Platform:~/Downloads/code$ ./fib_opt
Fibonacci(18) = 2584
Execution time: 0.01 milliseconds
wout@wout-VMware-Virtual-Platform:~/Downloads/code$
```

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

Ja, het is nu sneller dan eerst.

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.**

```
wout@wout-VMware-Virtual-Platform: ~/Downloads/code
GNU nano 7.2                                   runall.sh
#!/bin/bash

echo "Running C version:"
./fib_opt

echo "Running Java version:"
java Fibonacci

echo "Running Python version:"
python3 fib.py

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

```
wout@wout-VMware-Virtual-Platform:~/Downloads/code$ nano runall.sh
wout@wout-VMware-Virtual-Platform:~/Downloads/code$ chmod a+x runall.sh
wout@wout-VMware-Virtual-Platform:~/Downloads/code$ ./runall.sh
Running C version:
Fibonacci(18) = 2584
Execution time: 0.01 milliseconds
Running Java version:
Fibonacci(18) = 2584
Execution time: 0.24 milliseconds
Running Python version:
Fibonacci(18) = 2584
Execution time: 0.29 milliseconds
Running Bash version:
Fibonacci(18) = 2584
Excution time 5474 milliseconds
```

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.

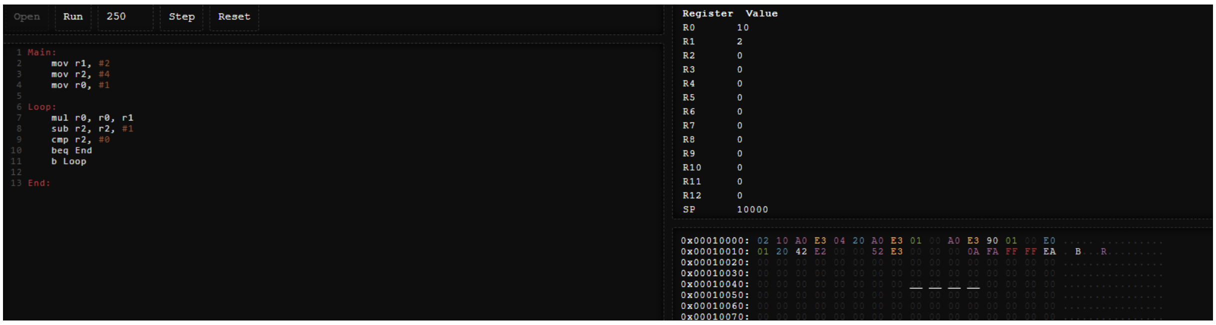
```
Main:
mov r1, #2
mov r2, #4

Loop:

End:
```

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

Screenshot of the completed code here.



Ready? Save this file and export it as a pdf file with the name: **week4.pdf**