

Template Week 4 – Software

Student number: 572750

Assignment 4.1: ARM assembly

Screenshot of working assembly code of factorial calculation:

The screenshot shows an ARM assembly simulator interface. At the top, there are buttons for 'Open', 'Run', '250', 'Step', and 'Reset'. Below these buttons is a text area containing assembly code:

```
1 Main:
2   add r0, r0, #2
3   mul r1, r0, r0
4   sub r2, r1, r0
5   mov r3, #15
```

To the right of the code area is a table titled 'Register Value' showing the current values of the registers:

Register	Value
R0	2
R1	4
R2	2
R3	f
R4	0
R5	0
R6	0
R7	0
R8	0
R9	0
R10	0
R11	0
R12	0
SP	10000
PC	0

The screenshot shows the same ARM assembly simulator interface. The assembly code in the text area is:

```
1 Loop:
2   add r0, r0, #1
3   mul r1, r0, r0
4   cmp r1, #144
5   beq Exit
6   b Loop
7
8 Exit:
```

The 'Register Value' table on the right shows the state after execution:

Register	Value
R0	c
R1	90
R2	0
R3	0
R4	0
R5	0
R6	0
R7	0
R8	0
R9	0
R10	0
R11	0
R12	0
SP	10000
PC	0

The screenshot shows the same ARM assembly simulator interface. The assembly code in the text area is:

```
1 Main:
2   mov r2, #5
3   mov r3, #1
4
5 Loop:
6   cmp r2, #1
7   beq End
8   mul r3, r3, r2
9   sub r2, r2, #1
10  b Loop
11
12 End:
13  mov r1, r3
```

The 'Register Value' table on the right shows the state after execution:

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

Assignment 4.2: Programming languages

Take screenshots that the following commands work:

javac --version

```
davin@davin-ubuntu-vm:~$ javac --version
javac 21.0.5
```

java --version

```
davin@davin-ubuntu-vm:~$ 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)
```

gcc --version

```
davin@davin-ubuntu-vm:~$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/libexec/gcc/x86_64-linux-gnu/13/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none:amdgc-nvptx-none:amdgc-nvptx-none:amdgc-nvptx-none
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 13.2.0-23ubuntu4' --with-bug-url=file:///usr/share/doc/gcc-13/README.Bugs --enable-languages=c,ada,c++,go,d,fortran,objc,obj-c++,m2 --prefix=/usr --with-gcc-major-version-only --program-suffix=-13 --program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/libexec --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --enable-cloCALE=gnu --enable-libstdc++-debug --enable-libstdc++-time=yes --with-default-libstdc++-abi=new --enable-libstdc++-backtrace --enable-gnu-unique-object --disable-vtable-verify --enable-plugin --enable-default-pie --with-system-zlib --enable-libphobos-checking=release --with-target-system-zlib=auto --enable-objc-gc=auto --enable-multiarch --disable-werror --enable-cet --with-arch=32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-offload-targets=nvptx-none=/build/gcc-13-uJ7kn6/gcc-13-13.2.0/debian/tmp-nvptx/usr,amdgc-nvptx-none=/build/gcc-13-uJ7kn6/gcc-13-13.2.0/debian/tmp-gcn/usr --enable-offload-defaulted --without-cuda-driver --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
Supported LTO compression algorithms: zlib zstd
gcc version 13.2.0 (Ubuntu 13.2.0-23ubuntu4)
davin@davin-ubuntu-vm:~$
```

python3 --version

```
davin@davin-ubuntu-vm:~$ python3 --version
Python 3.12.3
```

bash --version

```
davin@davin-ubuntu-vm:~$ 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.
davin@davin-ubuntu-vm:~$
```


Assignment 4.3: Compile

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

Java

C

Python

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

Bash

Which source code files are compiled to byte code?

Java

Python

Which source code files are interpreted by an interpreter?

C is interpreter

Java is interpreted

Python is interpreted

Bash is interpreter

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

Bash

How do I run a Java program?

<https://www.geeksforgeeks.org/how-to-run-java-program/>

How do I run a Python program?

<https://realpython.com/run-python-scripts/>

How do I run a C program?

<https://unstop.com/blog/how-to-run-c-program#:~:text=After%20downloading%20a%20C%20compiler,file%20to%20get%20the%20output.>

How do I run a Bash script?

<https://phoenixnap.com/kb/run-bash-script>

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

Java: Will create a .class file

Python: Will create a .pyc file

C: Will create a .out file

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 runs the fastest

```
davin@davin-ubuntu-vm:~/Downloads/code$ sudo ./fib.sh
[sudo] password for davin:
Fibonacci(18) = 2584
Execution time 18350 milliseconds
davin@davin-ubuntu-vm:~/Downloads/code$
```

```
davin@davin-ubuntu-vm:~/Downloads/code$ javac Fibonacci.java
davin@davin-ubuntu-vm:~/Downloads/code$ ls
fib fib.c Fibonacci.class Fibonacci.java fib.py fib.sh runall.sh
davin@davin-ubuntu-vm:~/Downloads/code$ java Fibonacci
Fibonacci(18) = 2584
Execution time: 0.58 milliseconds
davin@davin-ubuntu-vm:~/Downloads/code$
```

```
davin@davin-ubuntu-vm:~/Downloads/code$ gcc fib.c -o fibc
davin@davin-ubuntu-vm:~/Downloads/code$ ls
fib fib.c fibc Fibonacci.class Fibonacci.java fib.py fib.sh runall.sh
davin@davin-ubuntu-vm:~/Downloads/code$ ./fibc
Fibonacci(18) = 2584
Execution time: 0.04 milliseconds
davin@davin-ubuntu-vm:~/Downloads/code$
```

```
davin@davin-ubuntu-vm:~/Downloads/code$ python3 fib.py
Fibonacci(18) = 2584
Execution time: 0.98 milliseconds
davin@davin-ubuntu-vm:~/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.
- 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?
- 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.

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:

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

Screenshot of the completed code here.

The screenshot shows an ARM assembly simulator interface. On the left, the assembly code is displayed with line numbers 1 through 15. The code includes a 'Main' section and a 'Loop' section. The 'Main' section moves values into registers r1, r2, r0, and r12. The 'Loop' section multiplies r0 by r1, adds the result to r12, compares r12 with r2, and branches back to the loop if not equal. The 'End' label is at line 15. On the right, a 'Register Value' table shows the current state of registers R0 through R15, SP, and PC. R0 is 10, R1 is 2, R2 is 4, and R12 is 4. SP is 10000. The PC is 0. At the bottom, a memory dump shows the first few bytes of memory, including the instruction at address 0x00010020: FB FF 1A -- -- -- --.

```
1 Main:
2   mov r1, #2
3   mov r2, #4
4   mov r0, r1
5   mov r12, #1
6   b loop
7
8
9 loop:
10  mul r0, r0, r1
11  add r12, r12, #1
12  cmp r12, r2
13  bne loop
14
15 End:
```

Register	Value
R0	10
R1	2
R2	4
R3	0
R4	0
R5	0
R6	0
R7	0
R8	0
R9	0
R10	0
R11	0
R12	4
SP	10000
PC	0

0x00010000: 02 10 A0 E3 04 20 A0 E3 01 A0 E1 01 C0 A0 E3
0x00010010: FF FF FF EA 90 01 E9 01 C0 EC E2 02 9C E1
0x00010020: FB FF 1A -- -- -- --

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