Lab1: Hello World

1. Objective

To become familiar with the basics: text editor, assembler, linker, and debugger. After finishing this experiment, you should be able to do the following:

1. Use a text editor to create an assembly source code (.s).
2. Understand the general procedure to develop and debug an assembly program.

2. Background

In the following, we assume you have created aliases by adding these lines to end of ~/.bashrc (The last line adds the current directory to PATH. If you just added these lines without rebooting, then run 'source ~/.bashrc'):

```
alias as32='arm-linux-gnueabihf-as'
alias ld32='arm-linux-gnueabihf-ld'
alias gcc32='arm-linux-gnueabihf-gcc'
export PATH=".:$PATH"
```

To assemble a program (assuming the file name is lab1.s), one should type the command line:

```
$ as32 -o lab1.o lab1.s
```

where .s file is the source file and .o file is the output object file containing the machine code.

The linker creates an executable file (or a library) from one or more object files:

```
$ ld32 -o lab1 lab1.o
```

To run the program:

```
$ ./ lab1
```

The entry point of an assembly source program is usually referred to as32 "_start". If necessary, we can change the entry point to "main" (usually not needed):

```
$ ld32 -e main -o lab1 lab1 . o
```

The GNU debugger gdb allows you to execute, trace, inspect, and change variables during program execution. GNU ddd is a graphical front-end for the command-line gdb.

3. Lab Steps

Part 1: "Hello World" Program

In this section we run lab1p1.s, which calls the "printf" function from the C runtime library.

1. Create a file named lab1p1.s by copying its content from the Appendix. You can use vi or some other text editor.
2. Assemble and link the files with gcc32. (gcc links in the C runtime library libc, which contains the printf() function. If you run as32 and ld32, then libc is not linked. -g includes debug information.)

```
$ gcc32 -g -o lab1p1 lab1p1.s
```

3. Execute the program by typing lab1p1. You should see the output "Hello World!".

Next, we run lab1p2.s, which implements the "printf" function from the C library in Assembly.

1. Create another file named lab1p2.s by copying its content from the Appendix.
2. Assemble and link the files with as32 and ld32 (do not use gcc32, since we do not want to link in the C runtime library libc, which contains the printf function.)

```
$ as32 -g -o lab1p2.o lab1p2.s
$ ld32 -g -o lab1p2 lab1p2.o
```

3. Execute the program by typing lab1p2. You should see the same output “Hello World!”.

Part 2: Use the command-line tool gdb for debugging

In this section, we use gdb to debug the lab1p2.s program. In C programming, you can print out the value of each variable to make sure your program is functioning properly. In assembly, the **registers** take the position of “variables”, and you can examine their values with a debugger. (For some reason, gdb hangs when you debug a program directly within it. We need to start a gdbserver in one Raspberry PI terminal, and perform remote debugging in another terminal.)

1. Install gdb-multiarch and gdbserver.

```
$ sudo apt update
$ sudo apt install gdb-multiarch
$ sudo apt install gdbserver
```

2. In one Raspberry PI terminal, start the server.

```
$ gdbserver :1234 ./lab1p2
```

In another terminal, ssh to localhost to get another Raspberry PI terminal, and run the client.

```
$ ssh -p 2222 pi@localhost
$ gdb-multiarch ./lab1p2
```

3. In the client terminal (gdb) prompt, run:

```
(gdb) target remote localhost:1234
```

Then repeatedly run the following three commands to step through each line and examine the register values:

```
(gdb) stepi
(gdb) disassemble
(gdb) info registers
```

You should see output similar to the following screenshot:

```
(gdb) stepi
0x0001007c in main ()
(gdb) disassemble
Dump of assembler code for function main:
   0x00010074 <+0>:     mov     r0, #1
   0x00010078 <+4>:     ldr     r1, [pc, #20]   ; 0x10094 <main+32>
=> 0x0001007c <+8>:     mov     r2, #14
   0x00010080 <+12>:    mov     r7, #4
   0x00010084 <+16>:    mov     r2, #3
   0x00010088 <+20>:    svc     0x00000000
   0x0001008c <+24>:    mov     r7, #1
   0x00010090 <+28>:    svc     0x00000000
   0x00010094 <+32>:    muleq   r2, r8, r0
End of assembler dump.
(gdb) info registers
r0             0x1                 1
r1             0x20098             131224
r2             0x0                 0
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0xfffefc60          0xfffefc60
lr             0x0                 0
pc             0x1007c             0x1007c <main+8>
cpsr           0x10                16
fpscr          0x0                 0
```

3. After finishing running lab1p2, fill in the table below with register values (in hex) after each instruction has executed. (After the last step, the program has finished, so register values no longer exist.)

After Executing Instruction	r0	r1	r2	r7
mov r0, #1				
ldr r1, =message				
mov r2, =length				
mov r7, #4				
mov r2, #3				
swi 0				
mov r7, #1				
swi 0	--	--	--	--

Table 1: Instruction trace table.

Lab deliverable 1

Include Table 1 above in your lab report.

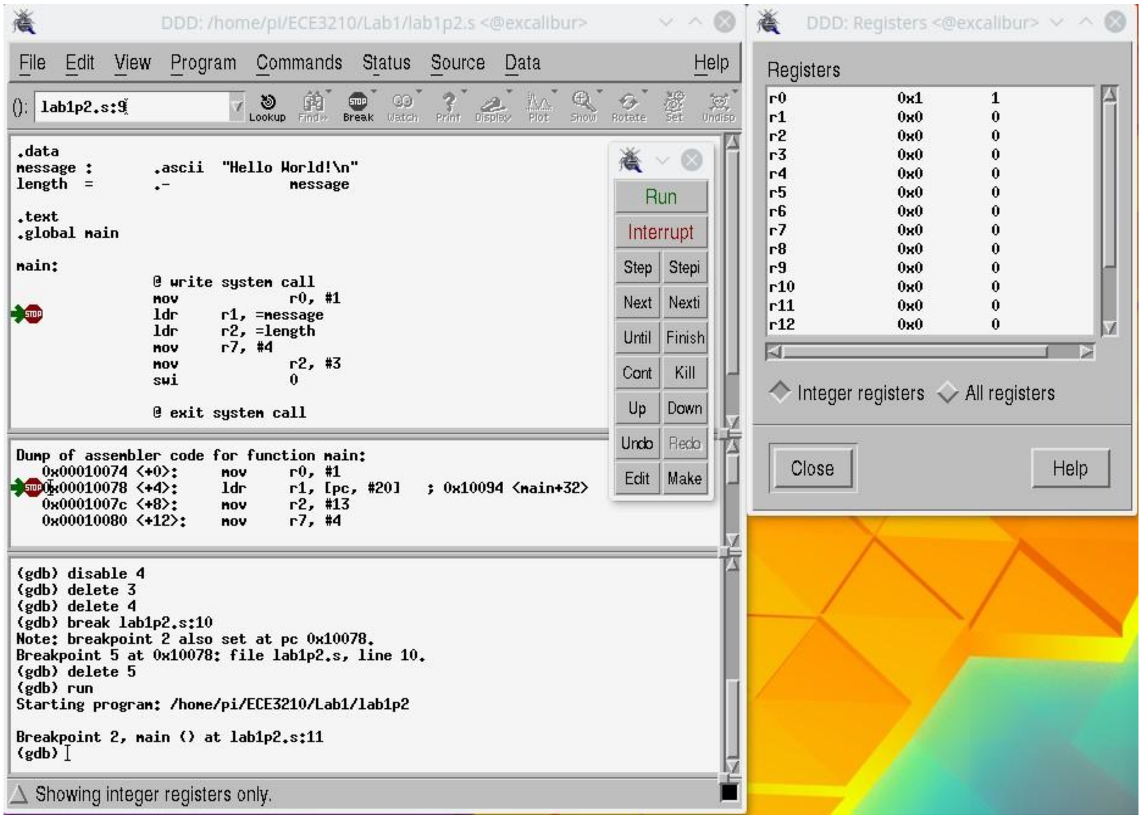

Figure 1: The DDD debugger.

Part 3: Use the graphical interface ddd for debugging (Optional)

In this section, we will use ddd to debug the lab1p2.s program. Enter the commands:

```
$ ssh -X pi@localhost -p 2222
$ ddd lab1p2
```

Under the “View” tab, open the “Machine Code Table”. Under the “Status” tab, click “Registers”. Now your interface should look similar to Figure 1. You can set a breakpoint by clicking in the blank area (left side as32 shown in Figure 1) next to each instruction. Once the breakpoint has been set, click the “Run” button to start debugging and the “stepi” button to trace each instruction. The value of each register should be displayed in the Registers status window on the right side. (As ddd is quite slow, and I personally find it more convenient to use the command-line gdb.)

Lab deliverable 2

lab1p2.s contains a small bug. After fixing the bug, change the program to print your name before Hello World, e.g., “John Doe Hello World!”. Include the modified program in your lab report, and also upload it as a separate .s file on Canvas for execution and grading.

5 Report

Please use the project report template and submit the report in PDF format. Describe your experiences in completing the project, and make sure to include Lab deliverables 1 and 2. Submit a separate source file for the modified lab1p2.s.

6 Appendix

lab1p1.s

```
.data
message: .asciz    "Hello World!\n"

.text
.global main
main:
push      {ip, lr}
ldr       r0, =message       @ Load the starting address of the message
bl        printf     @ Call the printf function
mov       r0, #0     @ Return 0.
pop       {ip, pc}
```

lab1p2.s

```
.data
message: .asciz   "Hello World!\n"
length = . - message  @ Returns string length of message

.text
.global main
main:
@ write syscall
mov r0, #1          @ For stdout
ldr r1, =message   @ buffer is loaded with message
ldr r2, =length     @ count is the length of message
mov r7, #4          @ write is syscall 4
mov r2, #3
swi 0              @ interrupt

mov r7, #1        @ exit syscall
swi 0
```