

# 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

### 2.1 Assembler

In the following, we assume you have created aliases, e.g., “as32” for arm-linux-gnueabi-hf-as, and “ld32” for arm-linux-gnueabi-hf-ld.

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

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

where .s file is the source file and .o file is the output object file containing the machine code. We include debugging information by using -g.

### 2.2 Linker

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 as “\_start”. If necessary, we can change the entry point to “main”:

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

As before, the linker will generate an executable file named “lab1”.

### 2.3 Debugger

The GNU debugger (**gdb**) allows you to execute, trace, inspect, and change variables during program execution. GNU DDD is a graphical front-end for command-line **gdb** debuggers.

## 3 Laboratory Workflow

### Part 1: “Hello World” Program

In this section you will create an assembly program that 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 vim or some other text editor.
2. Assemble and link the files.

```
$ as32 -o lab1p1.o lab1p1.s
$ ld32 -o lab1p1 lab1p1.o
```

3. Execute the program by typing ./lab1p1. You should see the output “Hello World!”.

Next, run lab1p2.s, which implements the “printf” function from the C library in Assembly. You should see similar output as lab1p1.s.

### Part 3: Use the command-line tool gdb for debugging

In this section, we will 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.

1. To start the debugger:

```
$ gdb lab1p2
```

You should see the following message on your screen.

```
Type "apropos word" to search for commands related to "word"...
Reading symbols from lab1_2...(no debugging symbols found)...done.
(gdb) █
```

2. Start the debug procedure by typing “start” within gdb:

```
(gdb) start
Temporary breakpoint 1 at 0x10074
Starting program: /home/pi/ECE3210/Lab1/lab1_2

Temporary breakpoint 1, 0x00010074 in main ()
(gdb) █
```

3. Next, we use the command “disassemble”. There is an arrow => pointing to the next instruction to be run.

```
(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, #13
    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) █
```

4. Before running the program, let's inspect some register values by using the “**info**” command.

```
(gdb) info registers
r0             0x0      0
r1             0x0      0
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             0x7efff510 0x7efff510
lr             0x0      0
pc             0x10074 0x10074 <main>
cpsr           0x10     16
(gdb) █
```

5. To step into each instruction one by one, we use the “**stepi**” command. Then we can follow with another “**disassemble**” to see what has happened.

```
(gdb) stepi
0x00010078 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, #13
    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) █
```

6. Repeat Steps 3~5 and fill the table with register values after each instruction has been run.

	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.

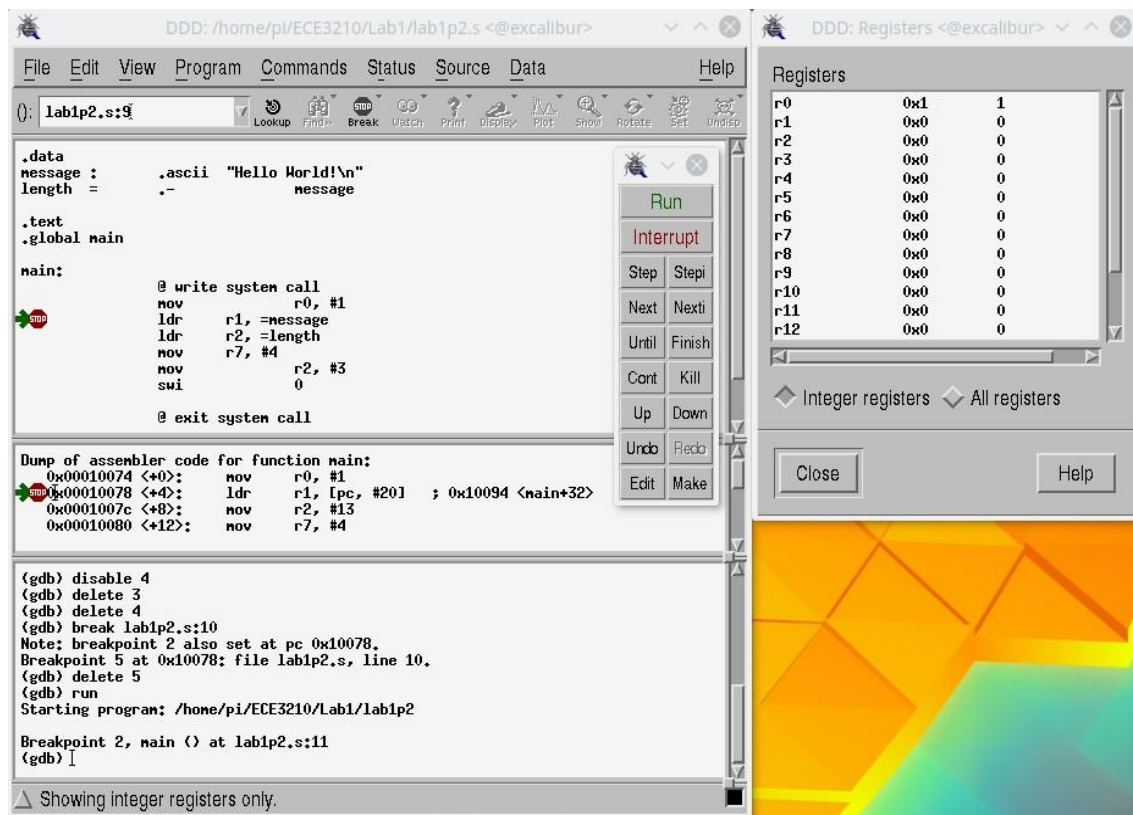


Figure 1: The DDD debugger.

### Part 3: Use the graphical interface DDD for debugging

In this section, we will use DDD to debug the **lab1p2.s** program. Enter the command:

```
$ 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 as shown in Figure 1) next to each instruction.

Once the breakpoint has been set, click the “**Run**” button to start debugging and the “**steppi**” button to trace each instruction. The value of each register should be displayed in the Registers status window on the right side.

#### 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.

## 5 Report

Please use the project report template. Describe your experiences in completing the project, and make sure to include Lab deliverables 1 and 2.

## 6 Appendix

### lab1p1.s

```
.data
message: .ascii "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: .ascii "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

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