COEN 311

Lab 1: Introduction to the software tool chain and STM32F334 Nucleo-64 Microcontroller
2023

T. Obuchowicz

OBJECTIVES

• To learn the rudiments of Linux (create a directory, change into a directory, edit a text file with the ‘nano’ text editor.

• To become acquainted with the ARM tool chain used in the COEN 311 labs for assembling, loading an ARM assembly language program and single stepping through the execution of the machine code on the STM32F334 Nucleo-64 microcontroller board

INTRODUCTION

This lab introduces the student to the Linux software tool chain used to assemble an ARM assembly language program and to program the Nucleo-64 microcontroller and single step through its execution. The tool chain consists of:

• arm-none-eabi-as: The GNU assembler (GNU Toolchain for the Arm Architecture)
• arm-none-eabi-ld: The GNU ld (loader) (GNU Toolchain for the Arm Architecture)
• arm-none-eabi-gdb: GNU gdb (GNU Toolchain for the Arm Architecture)
• openocd: The Open On-Chip Debugger, Licensed under GNU GPL v2

The assembler is used to assemble an program written in ARM assembly language into an object file. The loader creates an executable program from the object file. The openocd program establishes the serial communication between the gdb debugger program running on the local host Linux PC and the Nucleo-64 microcontroller board using a USB cable. Together, the openocd and gdb software allow to program the microcontroller board with the machine code corresponding to an executalbe program and to single-step through the execution of the program, one instruction at a time viewing the register and/or memory contents.

0. PREPARING THE LINUX ENVIRONMENT

The following is to be performed from the Linux prompt, onceone has logged in to an AITS Linux PC.

0. `ted@deadflowers Code 4:28pm >` **`module load COEN311`**

The `ted@deadflowers Code 4:28pm >` is an example of a Linux prompt. Your prompt may appear different. Commands are entered **after** the prompt and are terminated by pressing the **Enter** key.

The above command prepares one’s Linux environment to run the ‘openocd’ (open on-chip debugger), the ‘arm-none-eabi-as’ ARM assembler, the ‘arm-none-eabi-ld’ loader, and the ‘arm-none-eabi-gdb’ debugger.

After entering the above command, the Linux ‘which’ command may beused to verify that one’s search path is set properly:

```
ted@deadflowers Code 4:28pm >which openocd
/encs/pkg/openocd-0.11.0/root/bin/openocd

ted@deadflowers Code 4:33pm >which arm-none-eabi-as
/encs/pkg/gcc-arm-11.2.2022.02/root/bin/arm-none-eabi-as

ted@deadflowers Code 4:34pm >which  arm-none-eabi-ld
/encs/pkg/gcc-arm-11.2.2022.02/root/bin/arm-none-eabi-ld

ted@deadflowers Code 4:34pm >which arm-none-eabi-gdb
/encs/pkg/gcc-arm-11.2.2022.02/root/bin/arm-none-eabi-gdb
```

The which command shows the full path of built-in shell commands. The following is taken directly from the Linux man (manual) page entry:

```
ted@deadflowers 4:52pm >man which
WHICH(1)                  General Commands Manual
WHICH(1)

NAME
       which - shows the full path of (shell) commands.

SYNOPSIS
       which [options] [--] programname [...]

DESCRIPTION
      Which takes one or more arguments. For each of its arguments
it prints to stdout the full path of the executables that would
have been executed when this argument had been entered at the shell
prompt. It does this by searching for an executable or script in
the directories listed in the environment variable PATH using the
same algorithm as bash(1).
```

If for some reason, the specified command is not found in your Linux search path, the which command indicates this by reporting `"command not found"`.

1. RUNNING THE 'openocd' MONITOR PROGRAM

1a. Plug in the STM Nucleo-64 board into the provided USB cable and plug in the other end of the USB cable into the PC. Have your lab TA verify the connection is correct. To verify the USB device has been recognized by the operatingsystem, issue the 'lsusb' command:

```
 ted@deadflowers Code 4:37pm >lsusb
Bus 002 Device 004: ID 0483:374b STMicroelectronics ST-LINK/V2.1
```

```
Bus 002 Device 002: ID 8087:0020 Intel Corp. Integrated Rate Match-
ing Hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 004: ID 046d:c06a Logitech, Inc. USB Optical Mouse
Bus 001 Device 003: ID 03f0:0024 HP, Inc KU-0316 Keyboard
Bus 001 Device 002: ID 8087:0020 Intel Corp. Integrated Rate Match-
ing Hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

The device of interest is the 'STMicroelectronics ST-LINK/V2.1' (Your listing may show other devices)

1b. In a terminal window in which you have entered the 'module load COEN311', enter the following command:

```
ted@deadflowers Code 4:39pm > openocd -f board/st_nucleo_f3.cfg
```

The following will be displayed in the terminal window:

```
Open On-Chip Debugger 0.11.0
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control.
The results might differ compared to plain JTAG/SWD
srst_only separate srst_nogate srst_open_drain
connect_deassert_srst

Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 1000 kHz
Info : STLINK V2J30M19 (API v2) VID:PID 0483:374B
Info : Target voltage: 3.267107
Info : stm32f3x.cpu: hardware has 6 breakpoints, 4 watchpoints
Info : starting gdb server for stm32f3x.cpu on 3333
Info : Listening on port 3333 for gdb connections
```

LED LD3 on the Nucleo-64 board will be illuminated in red, andLED LD1 will be alternating between green and red.

It is necessary for the 'openocd' monitor program to be running continuosly while one is using the gdb debugger to single step through the machine code insructions. Leave 'openocd' running in the terminal window and open a new terminal window. Once the single stepping with gdb is completed, the openocd monitor can be terminated with CTRL-C in its terminal window,or by simply closing its terminal window.

2. ASSEMBLING and LOADING an ARM ASSEMBLY LANGUAGE PROGRAM

These steps may be performed without the ‘openocd’ monitor running. They are best performed in a new terminal window. The ‘openocd’ monitor program is only required when using the ‘gdb’ debugger program.

2a. Use a Linux text editor to edit an ASCII text file with the following contents:

```
.syntax unified
.cpu cortex-m4
.thumb

.word 0x20000400
.word 0x800000ed
.space 0xe4

start:
        mov r0, #4
        mov r1, #5
        add r2, r1, r0

stop:   b stop
```

It is strongly recommended that you create a directory called COEN311 to hold all your COEN 311 related files, and a directory called CODE within your COEN 311 directory to hold all your assembly language programs. The followingLinux commands are used to do this:

```
mkdir COEN311
cd  COEN311
mkdir CODE
cd CODE
```

Save the file with a name of your choice with a filename extension of .s . For example: `add.s` or `lab1.s`.

The above assumed that the user is familiar with using a Linux text editor. For those unfamiliar with Linux and the available text editors, here is a very short guide to using the ’nano’ Linux text editor. Any Linux text command line text editor can be used such as ’vi’, ’emacs’, ’nano’, ’pico’. This example gives the basics of using the ’nano’ text editor as it is easy anduser-friendly. To start the ‘nano’ text editor, simply type the command ‘nano junk.txt’:

```
ted@willpower CODE  6:04pm > nano junk.txt
```

After you press Enter, you will see on your terminal:

```
 GNU nano 2.3.1          File: junk.txt


Start to type your code here.
blah blah blah..
you can use the  up, down, left, right
arrows to move around in the fkile and use
the Backspace and Delete keys.

                                    [ New File ]
^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C
Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T
To Spell
```

To save the file, press the Ctrl and O keys simultaneosly. The name of the file (junk.txt) will be the default file name, you can specify another name if your wish and the contents of the file will be saved a file residing in the directory from where you invoked the nano text editor from.
It is wise to save often, or else you may accidentally lose you typed in.

To exit from the text editor (after you have saved your file), press the Ctrl and X keys simulataneosly (this keystroke combination is shortened to 'Ctrl-X').

You may use the nano text editor to create the assembly language program and save it with filename of add.s or lab1.s .

2b. Assemble the source code using the assembler from the directory which contains your .s program :

```
ted@deadflowers > arm-none-eabi-as -g add.s -o add.o -al=add.lst
```

Provided you have correctly typed in the code, there will be no errors/warnings listed and you will be returned to the Linux prompt. Two files will be created:

```
-rw------- 1 ted ted   563 Oct 11 16:59 add.lst
-rw------- 1 ted ted  1644 Oct 11 16:59 add.o
```

Use the Linux `'more'` command to view the contents of the add.lst file:

```
ted@deadflowers Code 4:24pm >more add.lst
ARM GAS  add.s                  page 1


   1                    .syntax unified
   2                    .cpu cortex-m4
```

```
 3                      .thumb
 4
 5 0000 00040020        .word 0x20000400
 6 0004 ED000080        .word 0x800000ed
 7 0008 00000000        .space 0xe4
 7      00000000
 7      00000000
 7      00000000
 7      00000000
 8
 9                      start:
10 00ec 4FF00400               mov r0, #4
11 00f0 4FF00501               mov r1, #5
12 00f4 01EB0002               add r2, r1, r0
13
14 00f8 FEE7           stop:   b stop
15
```

The listing file gives the program line numbers in the left-most column, the addresses (actually offset values from some starting address) in the next column, and the machine code in the third column and the assembly language statements in the right-most columns. Addresses and machine code are given in hexadecimal notation.

2c. Create an executable program (.elf) with the linker/loader (from the directory which contains your .o file created with the assembler in step 2b):

```
ted@deadflowers >  arm-none-eabi-ld add.o -o add.elf -Ttext=0x8000000
arm-none-eabi-ld: warning: cannot find entry symbol _start; defaulting
to 0000000008000000
```

This will create the file:

```
-rwx------ 1 ted ted 67188 Oct 11 17:00 add.elf
```

You can use the Linux ‘ls -al’ command to list the contents of a directory.

The add.elf file is a complete executable program, note that ‘x’ included in the file permissions - ‘x’ means the file has execute permissions. The input the the ‘ld’ command was the ‘add.o’ object file created by the assembler. The `-o` option is used to specify the desired name of the output executable file. The `-Ttext` option is used to specify the starting address in the Nucleo-64 ARM processor’s main memory.

The warning:

```
arm-none-eabi-ld: warning: cannot find entry symbol _start; de-
faulting to 0000000008000000
```

is of no concern and can be ignored as it is not relevant for running a program without an operating system. [1]

3. SINGLE STEPPING THROUGH AN EXECUTABLE PROGRAM WITH ‘gdb’

3a. Ensure that you have the ‘openocd’ monitor program running in a Linux terminal window before you perform the following steps.

In a terminal window in which you have entered the ‘module load COEN311’ command, invoke the debugger with your ‘add.elf’ file (the add.elf file should be located in thesame directory from where you invoke the debugger):

```
ted@deadflowers  > arm-none-eabi-gdb add.elf
```

You will see displayed in the terminal window:

```
GNU gdb (GNU Toolchain for the Arm Architecture 11.2-2022.02 (arm-
11.14)) 11.2.90.20220202-git
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licens-
es/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type “show copying” and “show warranty” for details.
This GDB was configured as “--host=x86_64-pc-linux-gnu --tar-
get=arm-none-eabi”.
Type “show configuration” for configuration details.
For bug reporting instructions, please see:
<https://bugs.linaro.org/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type “help”.
Type “apropos word” to search for commands related to “word”...
Reading symbols from add.elf...
(gdb)
```

3b. From the (gdb) prompt, issue the following:

```
(gdb) target extended-remote localhost:3333
```

The following will be listed:

```
Remote debugging using localhost:3333
0x00000000 in ?? ()
(gdb)
```

3c. From the gdb prompt:

```
(gdb) monitor reset halt
```

This command is used to stop the processor after a hard reset.You will see:

```
Unable to match requested speed 1000 kHz, using 950 kHz
Unable to match requested speed 1000 kHz, using 950 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x800000ec msp: 0x20000400
```

These are messages which can be safely ignored.

3d. Load the program machine code into the processor's flash memory:

```
(gdb) load
```

You will see:

```
Loading section .text, size 0xfa lma 0x8000000
Start address 0x08000000, load size 250
Transfer rate: 718 bytes/sec, 250 bytes/write.
```

3e. Set a program breakpoint at some label defined in the source code. Our sample `add.s` program contained the label `'start'` at the first line of the code. In general, any line can have a label and any label can be used to set a breakpoint. Since we want to stop executing at the first program instruction, we set a breakpoint at label `'start'`:

```
(gdb) break start
```

```
Breakpoint 1 at 0x80000ec: file add.s, line 10.
Note: automatically using hardware breakpoints for read-only addresses.
```

3e. The program must first be run with the `'continue'` command:

```
(gdb) continue
```
Continuing.

Breakpoint 1, start () at add.s:10
10 mov r0, #4

3f. Once the program has been run with a breakpoint set, we can single-step through the program instruction-by-instruction and examine the contents of registers (using the info registers gdb 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             0x20000400          0x20000400
lr             0xffffffff          -1
pc             0x80000ec           0x80000ec <start>
xPSR           0x41000003          1090519043
fpscr          0x0                 0
msp            0x20000400          0x20000400
psp            0x0                 0x0
primask        0x0                 0
basepri        0x0                 0
faultmask      0x0                 0
--Type <RET> for more, q to quit, c to continue without paging--
control        0x0                 0
(gdb)
```

Note that the processor is waiting to execute the:

```
mov r0, #4
```

instruction , so the contents of register r0 is not yet equal to 4.

3g. Single step though the program with the `'stepi'` command and examine the contents of the various CPU registers as each instruction is executed:

```
(gdb) stepi
halted: PC: 0x080000f0
11              mov r1, #5

(gdb) info registers
```

```
r0             0x4                 4
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             0x20000400          0x20000400
lr             0xffffffff          -1
pc             0x80000f0           0x80000f0 <start+4>
xPSR           0x41000003          1090519043
fpscr          0x0                 0
msp            0x20000400          0x20000400
psp            0x0                 0x0
primask        0x0                 0
basepri        0x0                 0
faultmask      0x0                 0
--Type <RET> for more, q to quit, c to continue without paging--
control        0x0                 0
```

Note that register r0 has been loaded with the immediate data 4. Continue single-stepping and observing register contents:

```
(gdb) stepi
halted: PC: 0x080000f4
12              add r2, r1, r0
(gdb) info registers
r0             0x4                 4
r1             0x5                 5
r2             0x0                 0
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
```

(Note all register contents shown here for reasons of brevity)

```
(gdb)  stepi
halted: PC: 0x080000f8
stop () at add.s:14
14      stop:   b stop
(gdb) info registers
r0             0x4                 4
r1             0x5                 5
r2             0x9                 9
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
```

The program is now executing an intentionally placed infinite loop:

```
 stop:   b stop
```

The b instruction is a branch statement which transfers control to the statement labelled stop.

3h. One can terminate the program using the “CTRL-C” sequence of characters entered from the ‘gdb’ terminal window. Enter the ‘quit’ command from the ‘gdb’ prompt to exit from ‘gdb’ and return to the Linux prompt:

```
(gdb) quit

A debugging session is active.

        Inferior 1 [Remote target] will be detached.

Quit anyway? (y or n)
```

Enter ‘y’ to quit:

```
Detaching from program: /nfs/home/t/ted/COEN311/ARM_LABS/Code/
Oct_4_Test/add.elf, Remote target
[Inferior 1 (Remote target) detached]
```

ted@deadflowers Oct_4_Test 5:31pm >

4. EXIT FROM THE ‘openocd’ MONITOR:

From the terminal window in which the ‘openocd’ command was run from,
press “CTRL-C” . You will be returned to the Linux prompt in thisterminal:

```
Info : halted: PC: 0x080000f8
Info : dropped ‘gdb’ connection
```

```
^Cshutdown command invoked

ted@deadflowers Code 5:32pm >
```

REFERENCES

1. ARM-ASM Tutorial, Niklas Gurtlet,
https://www.mikrocontroller.net/articles/ARM-ASM-Tutorial.

T. Obuchowicz
Dec. 6, 2022