MCSL Lab01

Environment Setup and Debugger Operations

0310004
Kuan-Yen Chou

1. Purpose

1. Test the hardware devices for labs in this course.
2. Help us get familiar with the development environment.

2. Steps

Prof. Tsao introduced the OpenSTM32 IDE to the class for developing programs on the STM32 NucleoL476RG board two weeks ago, but I prefer not to use it, rather than compiling, linking, and loading programs by myself.

2.1 Build the Development Environment and Compile the Program

Therefore, the first thing to do is to get cross-platform toolchains, like compiler, debugger, and a bunch of binary utilities. I am working on a x86-64 Arch Linux (GNU/Linux), so I issue the command to install cross toolchains for ARM target.

```
# pacman -S arm-none-eabi-{gcc,gdb,binutils}
```

With the aid of these tools, we could easily build and configure the environment by writing a simple makefile. For example, this is the makefile used in this lab.

When being able to build a binary ELF for ARM, it's time to load it into the board's memory and to get it running. I found a tool called OpenOCD (Open On-Chip Debugger) on the Internet, and it seems that the OpenSTM32 IDE is also using that tool underneath the interface. OpenOCD can be a GDB server that serves as an abstract layer between the user and the hardware, communicating with the board using the JTAG interface. More detailed usage and informations can be found in this repository's README or OpenOCD's official documentation. However, I cannot use the similar command to install the OpenOCD, since the latest release of the tool does not contain the configuration file of our board, so I use the AUR (Arch User Repository) to download the PKGBUILD, change its source URL to the latest git repository mirror, and then build the OpenOCD on my host.

After installing those necessary tools, I have to configure some udev and openocd settings, but I'm not going into details of configuration in this report. One can find them in the repository or other places on the Internet. When finishing all those installation and configuration, we could finally manage and write our codes, build them, and load them into the memory. (`lab01-1` is the generated ELF binary file.)

```
$ arm-none-eabi-gdb -q lab01-1
Reading symbols from lab01-1...done.
(gdb) target extended-remote :3333
Remote debugging using :3333
0x00000000 in ?? ()
(gdb) load
Loading section .isr_vector, size 0x188 lma 0x8000000
Loading section .text, size 0x210 lma 0x8000188
Loading section .rodata, size 0x8 lma 0x8000398
Loading section .init_array, size 0x8 lma 0x80003a0
Loading section .fini_array, size 0x8 lma 0x80003a8
Loading section .data, size 0x428 lma 0x80003b0
Start address 0x80001ec, load size 2008
Transfer rate: 8 KB/sec, 334 bytes/write.
(gdb) monitor reset halt
Unable to match requested speed 500 kHz, using 480 kHz
Unable to match requested speed 500 kHz, using 480 kHz
adapter speed: 480 kHz
stm32l4x.cpu: target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x080001ec msp: 0x20018000
(gdb) b main
Breakpoint 1 at 0x8000188: file src/main-1.s, line 10.
(gdb) s
Note: automatically using hardware breakpoints for read-only addresses.
Reset_Handler () at src/startup_stm32.s:46
46      b    LoopCopyDataInit
(gdb) c
Continuing.

Breakpoint 1, main () at src/main-1.s:10
10          movs   r0, #AA
(gdb) display $r2
1: $r2 = 536872004
(gdb) s
11          movs   r1, #20
1: $r2 = 536872004
(gdb) s
12          adds   r2, r0, r1
1: $r2 = 536872004
(gdb) s
L () at src/main.s:15
15          B     L
1: $r2 = 105
(gdb) q
```

2.2 Variable Declaration and Memory Observation

```
(gdb) load
Loading section .isr_vector, size 0x188 lma 0x8000000
Loading section .text, size 0x220 lma 0x8000188
Loading section .rodata, size 0x8 lma 0x80003a8
Loading section .init_array, size 0x8 lma 0x80003b0
Loading section .fini_array, size 0x8 lma 0x80003b8
Loading section .data, size 0x440 lma 0x80003c0
```

```
Start address 0x80001f8, load size 2048
Transfer rate: 9 KB/sec, 341 bytes/write.
(gdb) b main
Breakpoint 1 at 0x8000188: file src/main-2.s, line 14.
(gdb) c
Continuing.
Note: automatically using hardware breakpoints for read-only addresses.

Breakpoint 1, main () at src/main-2.s:14
14          ldr    r1, =X
(gdb) display X
1: X = 100
(gdb) display (char *)&str
2: (char *)&str = 0x20000004 "Hello World!"
(gdb) display $r0
3: $r0 = 536872000
(gdb) display $r1
4: $r1 = 1088
(gdb) display *$r1
5: *$r1 = 0
(gdb) display $r2
6: $r2 = 536872028
(gdb) s
15          ldr    r0, [r1]
1: X = 100
2: (char *)&str = 0x20000004 "Hello World!"
3: $r0 = 536872000
4: $r1 = 536870912
5: *$r1 = 100
6: $r2 = 536872028
(gdb) s
16          movs   r2, #AA
1: X = 100
2: (char *)&str = 0x20000004 "Hello World!"
3: $r0 = 100
4: $r1 = 536870912
5: *$r1 = 100
6: $r2 = 536872028
(gdb) s
17          adds   r2, r2, r0
1: X = 100
2: (char *)&str = 0x20000004 "Hello World!"
3: $r0 = 100
4: $r1 = 536870912
5: *$r1 = 100
6: $r2 = 85
(gdb) s
18          str    r2, [r1]
1: X = 100
2: (char *)&str = 0x20000004 "Hello World!"
3: $r0 = 100
4: $r1 = 536870912
5: *$r1 = 100
6: $r2 = 185
(gdb) s
20          ldr    r1, =str
1: X = 185
2: (char *)&str = 0x20000004 "Hello World!"
```

```
3: $r0 = 100
4: $r1 = 536870912
5: *$r1 = 185
6: $r2 = 185
(gdb) s
21          ldr    r2, [r1]
1: X = 185
2: (char *)&str = 0x20000004 "Hello World!"
3: $r0 = 100
4: $r1 = 536870916
5: *$r1 = 1819043144
6: $r2 = 185
(gdb) s
L () at src/main.s:24
24          B      L
1: X = 185
2: (char *)&str = 0x20000004 "Hello World!"
3: $r0 = 100
4: $r1 = 536870916
5: *$r1 = 1819043144
6: $r2 = 1819043144
(gdb) q
```

2.3 Basic Arithmetic Computation and Memory Manipulation

Here is the code:

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

.data
        X:      .word   5
        Y:      .word   10
        Z:      .word   0

.text
        .global main

main:
        ldr     r0, =X
        ldr     r1, =Y
        ldr     r2, [r0]
        ldr     r1, [r1]
        muls    r2, r2, r1
        adds    r2, r2, r1
        str     r2, [r0]
        subs    r2, r1, r2
        ldr     r1, =Z
        str     r2, [r1]

L:
        B       L
```

3. Results and Analysis

3.1 Build the Development Environment and Compile the Program

Results have been shown above in the second section. It is obvious that the r2 register contains 105 in decimal when the program jumps to the `L()` function. Since 0x55 equals to 85 in decimal, so r2 will contain the sum of 85 and 20, after this instruction “adds r2, r0, r1”.

When debugging with GDB, I usually use two commands for observing the value of variables and registers. One is the ‘display’ command, it will then show the variable or register each time the program executes the next instruction. The other command is the ‘print’ command, usually issued as ‘p’ for short, it functions like the ‘display’ command, but it only shows once when you issue the command, like the “one-shot” ‘display’.

3.2 Variable Declaration and Memory Observation

Use the arm-none-eabi-objdump command, we can see the contents of “.data” section of the statically linked ELF file, and also the section headers.

```
...
Contents of section .fini_array:
 80003b8 a1010008 00000000                    ........
Contents of section .data:
 20000000 64000000 48656c6c 6f20576f 726c6421  d...Hello World!
 20000010 00000000 00000000 00000000 04030020  ...............
 20000020 6c030020 d4030020 00000000 00000000  l.. ... ........
 20000030 00000000 00000000 00000000 00000000  ................
 20000040 00000000 00000000 00000000 ac030008  ................
 20000050 00000000 00000000 00000000 00000000  ..............…
...
...
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .isr_vector   00000188  08000000  08000000  00010000  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .text         00000220  08000188  08000188  00010188  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .rodata       00000008  080003a8  080003a8  000103a8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
...
  8 .data         00000440  20000000  080003c0  00020000  2**3
                  CONTENTS, ALLOC, LOAD, DATA
  9 .bss          0000001c  20000440  08000800  00020440  2**2
                  ALLOC
 10 ._user_heap_stack 00000404  2000045c  08000800  0002045c  2**0
                  ALLOC
 11 .ARM.attributes 00000030  00000000  00000000  00020440  2**0
...
```

Hence, it can be seen that the LMA (load memory address) and the VMA (virtual memory address) are different for the ".data", ".bss", and "_user_heap_stack" sections. The reason is that those data sections are first loaded into ROM and then copied into RAM for later operation when the program starts. Here is the description that I found in the GNU ld's documentation:

> In most cases the two addresses will be the same. An example of when they might be different is when a data section is loaded into ROM, and then copied into RAM when the program starts up (this technique is often used to initialize global variables in a ROM based system). In this case the ROM address would be the LMA, and the RAM address would be the VMA.

So we can understand that the variable 'X' that we are going to access will be stored at its VMA 0x20000000, and also the variable 'str' will be stored at 0x20000004. It is natural then for us to guess that the initialization code is written in the startup code. Here is the code of `startup_stm32.s` and `'Reset_Handler'` is the entry point of this program.

```
Reset_Handler:

/* Copy the data segment initializers from flash to SRAM */
  movs  r1, #0
  b     LoopCopyDataInit

CopyDataInit:
        ldr     r3, =_sidata
        ldr     r3, [r3, r1]
        str     r3, [r0, r1]
        adds    r1, r1, #4

LoopCopyDataInit:
        ldr     r0, =_sdata
        ldr     r3, =_edata
        adds    r2, r0, r1
        cmp     r2, r3
        bcc     CopyDataInit
        ldr     r2, =_sbss
        b       LoopFillZerobss
/* Zero fill the bss segment. */
FillZerobss:
        movs    r3, #0
        str     r3, [r2]
        adds    r2, r2, #4

LoopFillZerobss:
        ldr     r3, = _ebss
        cmp     r2, r3
        bcc     FillZerobss

/* Call the clock system intitialization function.*/
        bl      SystemInit
/* Call static constructors */
        bl      __libc_init_array
/* Call the application's entry point.*/
```

```
        bl      main

LoopForever:
        b       LoopForever
```

By observing the linker script, we know that '_sdata' denotes the VMA where ".data" section begins, '_edata' denotes the VMA where ".data" section ends, and that '_sidata' denotes the LMA where the ".data" section is loaded into the flash memory, namely, ROM.

After setting the r1 register to zero, the CPU jumps to the 'LoopCopyDataInit' procedure, setting r0 to the address where the ".data" section begins and r3 to the address where the ".data" section ends.

Each time whenever the value of _sdata+r1 (r0+r1) is smaller than the value of _edata (r3), the procedure calls the 'CopyDataInit' to load four bytes data from the _sidata+r1 (the address of data in ROM) to the _sdata+r1 (the address of data in RAM). The loop continues until r0+r1 >= r3, which means that the whole ".data" section has been moved from ROM to RAM, so the next thing to do is to fill the ".bss" section with zeros, which is implemented by 'LoopFillZerobss' procedure.

The procedure are much simpler than the 'LoopCopyDataInit'. It basically set r2 to the address where ".bss" section starts, r3 to the address where ".bss" section ends, and compare the two registers. If r2 is smaller than r3, then it fill four-bytes zero in the place to which r2 points, and increase r2 by four. Continue the loop until the whole ".bss" section is filled with zeros.

Therefore, the answer to the first question is that since the initialized global variables 'X' and 'str' are placed in the ".data" section, so they are copied from the LMA to the VMA by the 'LoopCopyDataInit' procedure. In other words, their SRAM space is initialized by the values stored in the flash memory. Here is the execution process that have been explained above:

```
Breakpoint 1, Reset_Handler () at src/startup_stm32.s:30
30      movs r1, #0
(gdb) display X
1: X = -2147419952
(gdb) display (char *)&str
2: (char *)&str = 0x20000004 "\270", <incomplete sequence \361>
(gdb) s
31      b   LoopCopyDataInit
1: X = -2147419952
2: (char *)&str = 0x20000004 "\270", <incomplete sequence \361>
(gdb)
40          ldr   r0, =_sdata
1: X = -2147419952
2: (char *)&str = 0x20000004 "\270", <incomplete sequence \361>
(gdb)
41          ldr   r3, =_edata
1: X = -2147419952
2: (char *)&str = 0x20000004 "\270", <incomplete sequence \361>
(gdb)
42          adds  r2, r0, r1
1: X = -2147419952
2: (char *)&str = 0x20000004 "\270", <incomplete sequence \361>
(gdb)
43          cmp   r2, r3
```

```
1: X = -2147419952
2: (char *)&str = 0x20000004 "\270", <incomplete sequence \361>
(gdb)
44          bcc    CopyDataInit
1: X = -2147419952
2: (char *)&str = 0x20000004 "\270", <incomplete sequence \361>
(gdb)
34          ldr    r3, =_sidata
1: X = -2147419952
2: (char *)&str = 0x20000004 "\270", <incomplete sequence \361>
(gdb)
35          ldr    r3, [r3, r1]
1: X = -2147419952
2: (char *)&str = 0x20000004 "\270", <incomplete sequence \361>
(gdb)
36          str    r3, [r0, r1]
1: X = -2147419952
2: (char *)&str = 0x20000004 "\270", <incomplete sequence \361>
(gdb)
37          adds   r1, r1, #4
1: X = 100
2: (char *)&str = 0x20000004 "\270", <incomplete sequence \361>
(gdb)
40          ldr    r0, =_sdata
1: X = 100
2: (char *)&str = 0x20000004 "\270", <incomplete sequence \361>
(gdb)
41          ldr    r3, =_edata
1: X = 100
2: (char *)&str = 0x20000004 "\270", <incomplete sequence \361>
(gdb)
42          adds   r2, r0, r1
1: X = 100
2: (char *)&str = 0x20000004 "\270", <incomplete sequence \361>
(gdb)
43          cmp    r2, r3
1: X = 100
2: (char *)&str = 0x20000004 "\270", <incomplete sequence \361>
(gdb)
44          bcc    CopyDataInit
1: X = 100
2: (char *)&str = 0x20000004 "\270", <incomplete sequence \361>
(gdb)
34          ldr    r3, =_sidata
1: X = 100
2: (char *)&str = 0x20000004 "\270", <incomplete sequence \361>
(gdb)
35          ldr    r3, [r3, r1]
1: X = 100
2: (char *)&str = 0x20000004 "\270", <incomplete sequence \361>
(gdb)
36          str    r3, [r0, r1]
1: X = 100
2: (char *)&str = 0x20000004 "\270", <incomplete sequence \361>
(gdb)
37          adds   r1, r1, #4
1: X = 100
```

```
2: (char *)&str = 0x20000004
"Hell!\320Eh\270\353\005\006D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\
350\002g\277\363O\217&i\026\364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
40          ldr    r0, =_sdata
1: X = 100
2: (char *)&str = 0x20000004
"Hell!\320Eh\270\353\005\006D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\
350\002g\277\363O\217&i\026\364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
41          ldr    r3, =_edata
1: X = 100
2: (char *)&str = 0x20000004
"Hell!\320Eh\270\353\005\006D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\
350\002g\277\363O\217&i\026\364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
42          adds   r2, r0, r1
1: X = 100
2: (char *)&str = 0x20000004
"Hell!\320Eh\270\353\005\006D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\
350\002g\277\363O\217&i\026\364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
43          cmp    r2, r3
1: X = 100
2: (char *)&str = 0x20000004
"Hell!\320Eh\270\353\005\006D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\
350\002g\277\363O\217&i\026\364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
44          bcc    CopyDataInit
1: X = 100
2: (char *)&str = 0x20000004
"Hell!\320Eh\270\353\005\006D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\
350\002g\277\363O\217&i\026\364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
34          ldr    r3, =_sidata
1: X = 100
2: (char *)&str = 0x20000004
"Hell!\320Eh\270\353\005\006D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\
350\002g\277\363O\217&i\026\364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
35          ldr    r3, [r3, r1]
1: X = 100
2: (char *)&str = 0x20000004
"Hell!\320Eh\270\353\005\006D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\
350\002g\277\363O\217&i\026\364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
36          str    r3, [r0, r1]
1: X = 100
2: (char *)&str = 0x20000004
"Hell!\320Eh\270\353\005\006D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\
350\002g\277\363O\217&i\026\364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
37          adds   r1, r1, #4
1: X = 100
2: (char *)&str = 0x20000004 "Hello
Wo\270\353\005\006D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\350\002g\2
77\363O\217&i\026\364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
```

```
40          ldr    r0, =_sdata
1: X = 100
2: (char *)&str = 0x20000004 "Hello
Wo\270\353\005\006D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\350\002g\2
77\363O\217&i\026\364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
41          ldr    r3, =_edata
1: X = 100
2: (char *)&str = 0x20000004 "Hello
Wo\270\353\005\006D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\350\002g\2
77\363O\217&i\026\364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
42          adds   r2, r0, r1
1: X = 100
2: (char *)&str = 0x20000004 "Hello
Wo\270\353\005\006D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\350\002g\2
77\363O\217&i\026\364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
43          cmp    r2, r3
1: X = 100
2: (char *)&str = 0x20000004 "Hello
Wo\270\353\005\006D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\350\002g\2
77\363O\217&i\026\364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
44          bcc    CopyDataInit
1: X = 100
2: (char *)&str = 0x20000004 "Hello
Wo\270\353\005\006D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\350\002g\2
77\363O\217&i\026\364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
34          ldr    r3, =_sidata
1: X = 100
2: (char *)&str = 0x20000004 "Hello
Wo\270\353\005\006D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\350\002g\2
77\363O\217&i\026\364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
35          ldr    r3, [r3, r1]
1: X = 100
2: (char *)&str = 0x20000004 "Hello
Wo\270\353\005\006D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\350\002g\2
77\363O\217&i\026\364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
36          str    r3, [r0, r1]
1: X = 100
2: (char *)&str = 0x20000004 "Hello
Wo\270\353\005\006D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\350\002g\2
77\363O\217&i\026\364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
37          adds   r1, r1, #4
1: X = 100
2: (char *)&str = 0x20000004 "Hello World!
D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\350\002g\277\363O\217&i\026\
364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
40          ldr    r0, =_sdata
1: X = 100
```

```
2: (char *)&str = 0x20000004 "Hello World!
D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\350\002g\277\363O\217&i\026\
364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
41          ldr    r3, =_edata
1: X = 100
2: (char *)&str = 0x20000004 "Hello World!
D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\350\002g\277\363O\217&i\026\
364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
42          adds   r2, r0, r1
1: X = 100
2: (char *)&str = 0x20000004 "Hello World!
D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\350\002g\277\363O\217&i\026\
364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
43          cmp    r2, r3
1: X = 100
2: (char *)&str = 0x20000004 "Hello World!
D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\350\002g\277\363O\217&i\026\
364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
44          bcc    CopyDataInit
1: X = 100
2: (char *)&str = 0x20000004 "Hello World!
D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\350\002g\277\363O\217&i\026\
364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
34          ldr    r3, =_sidata
1: X = 100
2: (char *)&str = 0x20000004 "Hello World!
D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\350\002g\277\363O\217&i\026\
364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
35          ldr    r3, [r3, r1]
1: X = 100
2: (char *)&str = 0x20000004 "Hello World!
D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\350\002g\277\363O\217&i\026\
364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
36          str    r3, [r0, r1]
1: X = 100
2: (char *)&str = 0x20000004 "Hello World!
D\277v\030\066\032\b.\362\323\337\370\066`fa\365\350\002g\342\350\002g\277\363O\217&i\026\
364\200?\373\321\026\360\372\017\aəB(\277"
(gdb)
37          adds   r1, r1, #4
1: X = 100
2: (char *)&str = 0x20000004 "Hello World!"
...
```

If the ‘X’ declaration is placed in the “.text” section, the statically-linked ELF binary would look like this:

```
...
Contents of section .text:
```

```
 8000188 64000000 03490868 55221218 0a600249  d....I.hU"...`.I
 8000198 0a68fee7 88010008 00000020 10b5054c  .h......... ...L
 80001a8 227832b9 044b13b1 0448aff3 00800123  "x2..K...H.....#
 80001b8 237010bd 38040020 00000000 90030008  #p..8.. ........
...
Contents of section .data:
 20000000 48656c6c 6f20576f 726c6421 00000000  Hello World!....
...
```

It is clear that the variable ‘X’ is at 0x800188 with the value of 0x00000064, and the variable ‘str’ is now at 0x20000000. After debugging with arm-none-eabi-gdb, I noticed that the execution of instructions was almost the same, except that the instruction at the address 0x08000194 “str r2, [r1]” did not have any effect, which should store the sum of 85 and 100 back to the variable ‘X’. It is, of course, because X is stored in the flash memory (ROM), which is read-only.

When the “main” procedure finished execution, the first four bytes of the register r2 are exactly the same as the first four bytes of the variable ‘str’, which is obvious, since that is what the assembly code says. However, we can still observe that they are the same through GDB.

```
Breakpoint 2, L () at src/main-2.s:24
24          B     L
1: (void *)$r2 = (void *) 0x6c6c6548
2: (void *)str = (void *) 0x6c6c6548
(gdb)
```

Since this is little-endian, so both of their first four bytes are `48 65 6c 6c`.

The variable ‘str’ is declared in the main.s as: ‘str: .asciz “Hello World!”’, where the ‘.asciz’ means that the string is ended with a zero. We can also declare a variable with the same effect using:

```
str:   .ascii “Hello World!”
       .byte  0
```

3.3 Basic Arithmetic Computation and Memory Manipulation

The code is shown above in the section 2.3. Here is the execution result:

```
Breakpoint 1, main () at src/main-3.s:14
14          ldr   r0, =X
1: X = 5
2: Y = 10
3: Z = 0
4: $r0 = 536871992
5: $r1 = 1080
6: $r2 = 536872020
(gdb) s
15          ldr   r1, =Y
```

```
1: X = 5
2: Y = 10
3: Z = 0
4: $r0 = 536870912
5: $r1 = 1080
6: $r2 = 536872020
(gdb) s
16          ldr    r2, [r0]
1: X = 5
2: Y = 10
3: Z = 0
4: $r0 = 536870912
5: $r1 = 536870916
6: $r2 = 536872020
(gdb) s
17          ldr    r1, [r1]
1: X = 5
2: Y = 10
3: Z = 0
4: $r0 = 536870912
5: $r1 = 536870916
6: $r2 = 5
(gdb) s
18          muls   r2, r2, r1
1: X = 5
2: Y = 10
3: Z = 0
4: $r0 = 536870912
5: $r1 = 10
6: $r2 = 5
(gdb) s
19          adds   r2, r2, r1
1: X = 5
2: Y = 10
3: Z = 0
4: $r0 = 536870912
5: $r1 = 10
6: $r2 = 50
(gdb) s
20          str    r2, [r0]
1: X = 5
2: Y = 10
3: Z = 0
4: $r0 = 536870912
5: $r1 = 10
6: $r2 = 60
(gdb) s
21          subs   r2, r1, r2
1: X = 60
2: Y = 10
3: Z = 0
4: $r0 = 536870912
5: $r1 = 10
6: $r2 = 60
(gdb) s
22          ldr    r1, =Z
1: X = 60
2: Y = 10
```

```
3: Z = 0
4: $r0 = 536870912
5: $r1 = 10
6: $r2 = -50
(gdb) s
23          str    r2, [r1]
1: X = 60
2: Y = 10
3: Z = 0
4: $r0 = 536870912
5: $r1 = 536870920
6: $r2 = -50
(gdb) s
L () at src/main.s:26
26          B      L
1: X = 60
2: Y = 10
3: Z = -50
4: $r0 = 536870912
5: $r1 = 536870920
6: $r2 = -50
(gdb) q
```

4. Reviews and Applications

This first lab is to help us get familiar with the tools and operations. I am glad that Mr. Tsao chose open source development tools for this course, although I am more used to the old ways of coding and of building a binary, but they are the same tools in fact, without only the editor and UI features.

Before this course, I've only worked on x86 assembly codes, with AT&T syntax, and I think that is much simpler than ARM assembly, maybe because there is an operating system that helps me to do lots of things, like virtual memory mapping, system resources control, etc.

Despite that this is only the first lab, I already have learned a lot from it, like what initialization tasks having to be done when the machine's on, some basic embedded systems concepts and architectures, and its difference to the common i686 or amd64 ISAs.

Nonetheless, it is hard for me to come up with useful applications related to the lab, since it does not even have any inputs or outputs. The only possible application that crosses my mind is a testing program which is used to test the limitation of the hardware ability, and encode or store the wanted information in the memory or registers.