ECGR 6181/8181 - Lab 1

Objective: Programming bare-metal ARM with QEMU.

QEMU is a generic and open source machine emulator and virtualizer. When used as a machine emulator, QEMU can run OSes and programs made for one machine (e.g. an ARM board) on a different machine (e.g. your own PC). By using dynamic translation, it achieves very good performance.

Outcomes: After this lab, you will be able to

- Write a bare-metal “Hello World”
- Write simple GNU linker scripts
- Use gdb

Install QEMU

$ sudo apt-get update
$ sudo apt-get install qemu-system-arm
$ qemu-system-arm --version

Mine prints “QEMU emulator version 4.2.1 (Debian 1:4.2-3ubuntu6.17)”. Your might be a later version.

Install GNU GCC cross compiler toolchain

A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. In our case, we are compiling ARM instruction set on an x86 platform (depending on your machine).

1. Binutils package: This consists of the ld (linker), as (assembler), and so on.
$ sudo apt-get install binutils-arm-none-eabi

2. gcc package: to compile to a bare metal arm architecture
sudo apt-get install gcc-arm-none-eabi

3. gdb debugger for command line debugging.
sudo apt-get install gdb-multiarch

Hello World on Bare-metal ARM on QEMU

We will be using the ARM VersatilePB board that contains an ARM926EJ-S core and, among other peripherals, four UART serial ports. The first serial port in particular (UART0) works as a terminal. From the memory map of the board, UART0 is memory mapped to 0x101f100. A register (UARTDR) at offset 0x0 is used to transmit (when writing in the register) and receive (when reading) bytes. So to write "Hello World" to the terminal, we need to write to the beginning of the memory allocated for the UART0.

Make a new Lab1 directory (mkdir Lab1). Copy the code below to a file hello_world.c

```
// Bare metal hello world
volatile unsigned int * const UART0DR = (unsigned int *)0x101F1000;

void print_uart0(const char *s) {
     while(*s != '\0') { /* Loop until end of string */
          *UART0DR = (unsigned int)(*s); /* Transmit char */
          s++; /* Next char */
     }
}

void c_entry() {
     print_uart0("Hello world!\n");
}
```

The volatile keyword is necessary to instruct the compiler that the memory pointed by ART0DR can change or has effects independently of the program.

Next we need a startup code that sets up the stack required by C programs. We define a handler for the reset vector by branching to the text section containing our startup code that setups up the stack.

```
.global _Reset
_Reset:
     b Reset_Handler
     b . /* Undefined */
     b . /* SWI */
     b . /* Prefetch Abort */
     b . /* Data Abort */
     b . /* reserved */
     b . /* IRQ */
     b . /* FIQ */
.section .text
```

```
Reset_Handler:
      ldr sp, =stack_top
      bl c_entry
      b .
```

In the startup code, we load the stack pointer (sp) with the memory location for the top of the stack (supplied by the linker), and then calls the (bl instruction) entry function of the C program.

We must now tell the linker how to layout the different sections in memory via the linker script. We'll load the binary file at 64 KB, and then specify the startup, text, data, and a stack of 4 kB. The stack_top is defined at top of the stack with the stack growing downwards

```
ENTRY(_Reset)
SECTIONS
{
 . = 0x10000;
 .startup . : { startup.o(.text) }
 .text : { *(.text) }
 .data : { *(.data) }
 .bss : { *(.bss) }
 . = ALIEBN(8);
 . = . + 0x1000; /* 4kB of stack memory */
 stack_top = .;
}
```

Let's now compile, and link the code to produce a binary dump that we can load into the emulator.

Execute the following commands

$ arm-none-eabi-as -mcpu=arm926ej-s -g startup.s -o startup.o
$ arm-none-eabi-gcc -c -mcpu=arm926ej-s -g hello_world.c -o hello_world.o
$ arm-none-eabi-ld -T hello_world.ld hello_world.o startup.o -o hello_world.elf
$ arm-none-eabi-objcopy -O binary hello_world.elf hello_world.bin

The QEMU emulator is written especially to emulate Linux guest systems; for this reason we have to specify our binary after the kernel flag. The -M flag selects the specific machine to be emulated, the -m flag specifies the memory, and the -nographic flag specific the QEMU to be run as a command-line application, outputting everything to a terminal console. The serial port is also redirected to the terminal.

$ qemu-system-arm -M versatilepb -m 128M -nographic -kernel hello_world.bin

This should print “Hello World!” To exit QEMU press ctrl-a followed by x

Examine the disassembled contents of the HelloWorld.elf
$ arm-none-eabi-objdump -d hello_world.elf

Debug with gdb
GDB provides remote debugging capabilities, with a server called gdbserver running on the machine to be debugged, and then the main gdb client communicating with the server. QEMU is able to start an instance of gdbserver along with the program it’s emulating. QEMU implements a gdb connector using a TCP connection

$ qemu-system-arm -M versatilepb -m 128M -nographic -s -S -kernel hello_world.bin
This command freezes the system before executing any guest code, and waits for a connection on the TCP port 1234

In another terminal, from the Lab 1 directory, launch gdb
$ gdb-multiarch
(gdb) target remote localhost:1234
(gdb) file hello_world.elf // Type y to change the file if prompted
(gdb) load
(gdb) break c_entry // Sets a breakpoint at c_entry
(gdb) continue // Runs the program. Stop at c_entry
(gdb) step // step into the function
(gdb) next // next instruction
(gdb) info registers // prints content of registers
(gdb) print UART0DR // prints the contents of UART0DR

To do -
Use gdb and the disassembly to answer the following questions-
On how to use gdb, see http://www.gnu.org/software/gdb/documentation/
On the ARM instruction set, see https://iitd-plos.github.io/col718/ref/arm-instructionset.pdf

1. What is the memory address of stack_top label specified in startup.s? How much (in bytes) does the stack grow during the execution of the program?

2. At which memory location is the string “Hello world!\n” stored? Print out the contents of the 13 bytes of memory at this location (x /13c <mem address>)

3. How is this string passed to the function print_uart0() from c_entry()?

4. Identify the assembly instructions that implement the while loop of the function

print_uart0().

5. What is the endianness of this processor? Demonstrate by showing the memory layout of a multibyte word in memory.

The material in this lab is partly based on Balau blog by Francesco Balducci