The Department of Electrical and Computer Engineering

The University of Alabama in Huntsville

CPE 435 Lab-12

Bare Metal Demo

Introduction

Bare metal programs run without an underlying operating system, and they are useful in understanding the deeper layers of the architecture. They give us valuable information about hardware architecture and lower levels of operating systems.

This lab assignment is derived from following sites: https://balau82.wordpress.com/2010/02/14/simplest-bare-metal-program-for-arm/ and https://balau82.wordpress.com/2010/02/28/hello-world-for-bare-metal-arm-using-qemu/

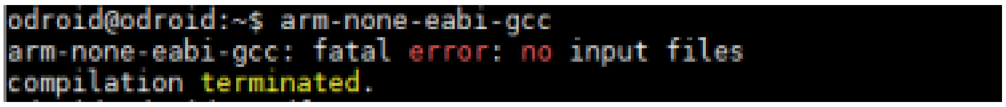
In this lab, we will write a program that we will compile for the platform that we are not using. The compiled platform will not have any operating system, which is why the name bare metal. The platform is ARM926EJ-S. QEMU will be used to emulate the platform and run the compiled program.

For doing the lab assignment, we will need basic knowledge on following terms. Please write at least two sentences about each of them highlighting how they differ from each other?

Complier
Assembler
Linker

Please make sure you have GCC toolchain installed. Type in the following in your odroid machine: `arm-none eabi-gcc`

If you get an error that says the following:



```
odroid@odroid:~$ arm-none-eabi-gcc
arm-none-eabi-gcc: fatal error: no input files
compilation terminated.
```

If you see something that suggests the package was not installed, install the package using the following command: `sudo apt-get install gcc-arm-none-eabi`

This will take some time to install.

Any source file has to be compiled into an object file by using Compiler. But there needs to be another object file that tells the hardware how to handle the object file that we created. If your program has a simple function as shown below:

```
/*FIlename: test.c*/
int c_entry()
{
        return 0;
}
```

You will compile it using `arm-none-eabi-gcc -c -mcpu=arm926ej-s -g test.c -o test.o`

Here `-mcpu` indicates the processor that we are using. When any processor is switched on, they go to a certain address to fetch the address of the next instruction to execute. Most of the time, this address will be the interrupt vector table address. The instructions in the vector table will indicate the processor to jump to the code that handles certain event which can be reset condition, unknown instruction etc

You will write your own assembler and a linker file. A sample assembler file that creates a interrupt vector table is given below:

```
/*Assembler, file name startup.s*/
 .section INTERRUPT_VECTOR, "x"
 .global _Reset
_Reset:
    B Reset_Handler /* Reset */
    B . /* Undefined */
    B . /* SWI */
    B . /* Prefetch Abort */
    B . /* Data Abort */
    B . /* reserved */
    B . /* IRQ */
    B . /* FIQ */

Reset_Handler:
    LDR sp, =stack_top
    BL c_entry
    B .
```

You will compile this using `arm-none-eabi-as -mcpu=arm926ej-s -g startup.s -o startup.o` . Now you need to link both the files. So you will need a linker.

A sample linker code is given below:

```
/*Linker test.ld*/
ENTRY(_Reset)
SECTIONS
{
   . = 0x0;
   .text : {
  startup.o (INTERRUPT_VECTOR)
  *(.text)
  }
```

```
    .data : { *(.data) }
    .bss : { *(.bss COMMON) }
    . = ALIGN(8);
    . = . + 0x1000; /* 4kB of stack memory */
   stack_top = .;
 }
```

The script tells the linker to place the INTERRUPT_VECTOR section at address 0, and then subsequently place the code (.text), initialized data (.data) and zero-initialized and uninitialized data (.bss). Line 11 and 12 tells the linker to move 4kByte from the end of the useful sections and then place the stack_top symbol there. Since the stack grows downwards the stack pointer should not exceed its own zone, otherwise it will corrupt lower sections. The script on line 1 tells the linker also that the entry point is at _Reset.

To link the program, use the following command:

```
arm-none-eabi-ld -T test.ld test.o startup.o -o test.elf
```

The final output is the elf file that you can execute.

Assignment

1. After you have studied all the materials above, please go to Hello world for bare metal ARM using QEMU | Freedom Embedded (wordpress.com). Please support your answers by screenshots.
2. How many serial terminals are supported in the platform that we plan to use? What is the address of the UART terminal used in the example?
    1. What register is used to transmit and receive the bytes?
3. Copy the code for `test.c`, and add comments to the code based on your answer for above question. Compile the code using `arm-none-eabi-gcc -c -mcpu=arm926ej-s -g test.c -o test.o`
4. Copy the assembler code. Generate the object file using `arm-none-eabi-as -mcpu=arm926ej-s -g startup.s -o startup.o`
5. Linker file is also provided. Copy the linker file code and link the object files generated. `arm-none-eabi-ld -T test.ld test.o startup.o -o test.elf`
6. Generate the binary file using `arm-none-eabi-objcopy -O binary test.elf test.bin`
7. Follow the command it says to run the binary in Qemu emulator.
8. Describe in detail all the steps that you just did in two paragraphs.

8. What are the other platforms that Qemu supports? (There is a command for this, please search it and run it. Grab the screenshot)
9. What are the benefits of Virtualization? What are benefits of Emulation?
10. Can a guest machine run a hypervisor. Please research on this topic and describe how the case varies in case of full virtualization, paravirtualization and hardware virtualization.
    1. What role does hypervisor have in all these?