PrairieLearn ARM32 Assembly Autograding with QEMU (Cortex-M3)

Senior Design May 2024 - 33

Mitch Hudson
Tyler Weberski
Chris Costa
Andrew Winters
Carter Murawski
Matt Graham

Table of Contents

Introduction

This document aims to supplement the previous one about QEMU autograding with a system for running bare-metal code on a simulated LM3S6965 board. QEMU by default emulates a Linux kernel to run code in, but this can be changed with the -kernel switch. Making this change takes a lot of work, as you will effectively be writing your own kernel. Thankfully, most of this work is already done by the creators of the LM3S6965 board, and we just need to fit it to our use case.

GitHub Repository and Installation

This autograder is built into the same GitHub repository as the Linux version, with the major changes stored in the bareMetalTemplateExample directory. For cloning and building instructions, refer back to the previous manual: here. The Docker container's code has been supplemented with special commands for using the bare-metal template, and these changes are documented in the Example Question section.

QEMU with Custom Kernel

By specifying the -kernel switch, we can use our own custom ELF binary that we compile using a Makefile. To do so, we added several files to the tests folder: startup.s, lms36965.ld, and main.c.

Startup.s and lm3s6965.ld are taken from this GitHub repository, and define the kernel structure as required by the Cortex-M3. The .ld file is a linker script designed to tell the compiler where each section of code needs to be for proper execution on the processor, and the startup file defines the different sections of the vector table, and their code. Finally, main.c is C code that defines the actual tests run by the auto-grader. This will be outlined in more detail later.

QEMU is set up in a way that, by default, writing to UART0 is hooked up to stdout, making it easy to display output. This makes it impossible to prevent students from manually printing outputs, but there are some workarounds to this, such as doing output testing by the python script instead of the emulated program.

QEMU Command Explanation

The current version of the QEMU command, found in the Makefile, is as follows:

```
$ qemu-system-arm -M lm3s6965evb -semihosting -nographic -kernel
system.bin
```

qemu-system-arm

Instead of qemu-arm as before, which runs QEMU in user mode, we are using the emulator in full system mode, where it will load a kernel and run a full emulated version of the defined board.

-M lm3s6965evb

This switch specifies the board for QEMU to emulate. In our case, we use the LM3S6965, which utilizes a Cortex-M3 processor. This processor is very similar in architecture to the TM123 used in class, which is why we use it.

-semihosting

Semihosting is a feature of ARM that allows the processor to communicate with an externally connected debugger. In the case of QEMU, it allows us to use a special syscall to cleanly exit the program on completion. Normally, when the execution is completed, the processor is sent to a hang function that is just an infinite loop, and never exits on its own. Instead, we use a function called _exit_qemu() to tell QEMU to stop execution, and exit, allowing the output to be read by PrairieLearn.

-nographic

Self-explanatory, this switch tells QEMU to not run a graphics output. This is required because we are running in a Docker container terminal with no graphical output.

-kernel system.bin

This switch tells QEMU which binary to load as the kernel. This binary holds the entire system's code, and needs to define everything the processor expects to run. This includes the vector table at the start of the file, and the reset point. These are defined in startup.s and linked using lm3s6965.ld.

-S -s

This is an optional flag to tell QEMU to start the internal GDB server and wait for a connection before starting execution. More info about this is available in the Debugging section.

Compiling System.bin

To get system.bin, there is a bit of a process, which has been made much simpler through the use of a Makefile. The general steps are as follows:

```
$ arm-none-eabi-as -mcpu=cortex-m3 student.s -o student.o
$ arm-none-eabi-as -mcpu=cortex-m3 startup.s -o startup.o
$ arm-none-eabi-gcc -c -mcpu=cortex-m3 main.c -o main.o
$ arm-none-eabi-ld -T lm3s6965.ld main.o student.o startup.o -o
system.elf
$ arm-none-eabi-objcopy -O binary system.elf system.bin
```

arm-none-eabi?

Unlike the previous method where we used the ARM Linux cross-compiler, we need to use a more general one that doesn't insert the Linux overhead. To do so, we use the package gcc-arm-none-eabi, which contains compilation tools for processor specific (through the use of -mcpu switch) compilation and loading. This package comes pre-installed in the Docker container.

arm-none-eabi-as

The 'as' program assembles ARM assembly code into executable objects. This command needs to specify the -mcpu=cortex-m3 switch to tell the compiler what ARM commands the target system has access to. To do debugging, you need to add the -g flag. The two files we assemble are the startup script and the student's code.

arm-none-eabi-gcc

The 'gcc' program compiles C code into executable objects. Normally, this requires specialized methods for the entry point and linking, so to get around this we use the -c flag which prevents GCC from invoking the linker program (this step is done manually later). This step also needs to use the -mcpu=cortex-m3 switch for defining the available instruction set, and the -g switch if you want to do debugging. Here, we compile the main.c file into an object file for linking with the rest of the kernel.

arm-none-eabi-ld

The 'ld' program is the standalone linker program. This links all of the code together so that the program can function. We use the -T lm3s6965.ld switch to tell the linker to use the lm3s6965.ld linker script, which is specially designed for the Cortex-M3 processor. The output of this command is a compiled ELF program, which needs to be translated into a binary for use in QEMU. If you want to do debugging this step also needs the -g

flag, and the system.elf file is the file you will use when reading debugger symbols in GDB.

arm-none-eabi-objcopy

This last step translates the ELF file into a binary that can be read and executed by QEMU.

Makefile

The Makefile makes this process very simple. Aside from running each individual step, you can also run all of them at once using ‘make all’, which will compile both the regular and debuggable versions of system.bin. ‘make run’ can be used to compile and run the regular program through QEMU. ‘make dbg’ is the same as ‘make run’, except that it compiles a version of the program with debugger symbols intact and starts QEMU’s internal GDB server. This can be connected to in GDB through port 1234 and is gone into more detail in the Debugging section.

Example Question

The included example question for this version of the grader is stored in bareMetalTemplateExample. Inside this directory is a complete PrairieLearn question, and as the process for creating questions is much more involved, no simple template is provided.

server.py

Server.py has a new function called generateAddress() which generates random addresses that should be safe from being interfered with by the compiler. These addresses fall between 0x2000_0000 and 0x2000_0FFF (4 kB of memory). This should be more than enough for any question. The linker script for the system has been modified, so no variables should be placed in this address-space by the compiler, keeping it free for our uses.

The example question generates two addresses, one for ‘a’ and one for ‘b’, both integers.

main.c

Main.c stores the code wrapper to call the students’ code. There are some issues right now, specifically with reading large quantities of input data, so keep the input scope small.

Main.c several functions to make input and output simpler, specifically a basic implementation of scanf() and printf(). Both of these functions can handle strings, characters, and integers, as well as escaping the % sign for printf, and are found in io.h.

Most of your variables used by the students will be set up by tests.py upon compilation, so having errors on a = as[i] and b = bs[i] are normal here.

The general idea here is that inputs for the variables are passed in by tests.py when calling the function and outputs will be printed in a way that can be read by tests.py. This prevents students from printing a message like “Success!” that will cause them to pass the test without doing anything.

tests.py

Tests.py holds the code that actually compiles, runs, and tests the student's code. The options here have expanded since the previous version with Linux ARM. Specifically, we added self.generateHeaders(), self.make(), and self.test_make_run().

generateHeaders()

Generate headers takes a couple of arguments and allows the main.c code to be updated programmatically by the testing script.

main_file: path to the main.c file, for modifying it. Defaults to /grade/tests/main.c

generate_rand: tells the header to include an int array of this value number of random values from 0-65535. This can be used to introduce randomness to the C program itself, if you need it. Defaults to 256.

variables: dictionary of variables and assigned addresses as hex strings. The template uses a and b, both placed in random addresses generated by server.py
dictionary of variables and assigned addresses as hex strings. For example, the template uses a, b, ans_a, and ans_b, putting the addresses generated by server.py in a and b, and randomly generating more addresses for ans_a and ans_b. You can tell it to use a random address by passing None as the address, and these addresses are generated between 0x20008000 and 0x20008FFF.

functions: list of strings that are just appended to the top of the C code after the include directives. This can be used to add anything you might need from the tests.py or server.py generation, like an answer function.

make()

This function invokes make to compile and prepare the program for execution. The function takes one parameter, 'student_file', and uses it to copy the student's code into the /grade/tests directory alongside the rest of the code. This defaults to 'student.s', and any output from make is put as the message for the test.

test_make_run()

This function uses the same parameters as the other testing functions in the C grader, with minor differences. Most notably, it takes no command argument, as the command is already known from the Makefile. Also, input here is helpful for passing input through to the emulator on UART0.

Writing Tests

Writing tests with this system should be fairly simple. All you should have to do is initialize any variables you need in the server.py to display to the student in question.html, write your code to read input, execute student code, and report outputs in main.c, then in tests.py, take the variables from server.py into account using the new generateHeaders() function, generate your input and expected output, make, and test the code using the new functions.

You can perform tests in any way you want, but this method was used in the template and was what the ARMGrader class was designed around.

Debugging Tests

Since adding make, the process for debugging has simplified considerably:

```
$ make dbg &                // starts the program in background
[1] <pid>
$ gdb-multiarch <executable>          // Loads the symbol table
> target remote localhost:<port>      // Connects to QEMU GDB server
> layout src                          // Sets the layout for GDB
```

To make this process even simpler, a one liner can be used instead (after compiling and linking):

```
$ make dbg & gdb-multiarch <exe> -ex 'target remote localhost:1000'
-ex 'layout src'
```

This starts QEMU in the background, starts GDB, and automatically connects to the remote server and sets the layout.

One issue that comes up with this method, however, is adding inputs. To add inputs, you will need to call the command that ‘make dbg’ does directly while piping in inputs using either echo or printf:

```
$ echo "inputs here" | qemu-system-arm ...<rest of the command>
```

Resources

Git Repositories

ARMGrader:
https://github.com/myriath/PrairieLearnARMGrader
New Question:
https://git.ece.iastate.edu/sd/sdmay24-33/-/tree/20-hw12-mitch/questions/H12_Q2a
Test Question (old):
https://git.ece.iastate.edu/sd/sdmay24-33/-/tree/mitch-asm-test/questions/bareMetalTemplateQuestion?ref_type=heads

QEMU and Assembly Help

Tutorial for QEMU and cross-compilation:
https://azeria-labs.com/arm-on-x86-qemu-user/
ARMv7 Cheat Sheet:
https://courses.cs.washington.edu/courses/cse469/20wi/armv7.pdf

ARM Bare-Metal Help

LM3S6965 Datasheet:
https://www.ti.com/lit/ds/symlink/lm3s6965.pdf
General Bare-Metal Tutorial 1:
https://github.com/umanovskis/baremetal-arm/blob/master/doc/06_uart.md
Bare-Metal Tutorial 2:
https://balau82.wordpress.com/2010/02/14/simplest-bare-metal-program-for-arm/
https://balau82.wordpress.com/2010/02/28/hello-world-for-bare-metal-arm-using-qemu/
Startup Script base:
https://github.com/varun-venkatesh/bare-metal-arm/blob/master/src/chapter1/startup_asm/startup_lm3s6965.s
Linker Script base:
https://github.com/varun-venkatesh/bare-metal-arm/blob/master/src/chapter1/startup_asm/lm3s6965_layout.ld
Linker Script resource:
https://mcyoung.xyz/2021/06/01/linker-script/
Header Files for LM3S6965:
https://github.com/speters/CMSIS/blob/master/Device/TI/LM3S/Include/LM3S6965.h