Lab #1: ARM Programming Environment and Emulation and Code Optimization

18–342 Fundamentals of Embedded Systems
Carnegie Mellon University

Out: September 12, 2015, 11:59 pm EDT
Part 1 Due: September 22, 2015, 11:59 pm EDT
Part 2 Due: October 2, 2015, 11:59 pm EDT

Contents

1 Objectives of this Lab

The 18–342 course uses the the ARM-based Gumstix board, specifically, the Gumstix Verdex Pro series [11] built on the ARMv5 architecture. In the embedded-systems industry, software is developed either offline (because the hardware poses constraints that do not permit a complete software-development toolchain to execute) or on-system (where the the code is developed, compiled and tested on the hardware or an emulator directly). In this lab exercise, you will be exposed to an on-system emulator-based emebdded software-development environment.

The overall objective of Lab #1 is to get you acquainted with the ARM programming environment (through an emulator called QEMU), and to give you hands-on experience in developing and optimizing your code for the ARM processor. To that end, there are two specific parts to Lab #1:

1. **Embedded software-development skills.** Setting up the development environment (including revision control and the QEMU emulator); building your projects using the `make` utility; writing a simple program for the ARM processor in `C` and in ARM assembly language.
2. **Code-optimization skills.**

2 Support Code

You have been provided with the following files as a part of the support code for this lab.

- `part1.tar.gz`: This contains files that will be necessary for the completion of Part 1.
- `part2asm.tar.gz`: This contains files that will be necessary for the completion of Part 2's ARM assembly-language optimization.
- `part2c.tar.gz`: This contains files that will be necessary for the completion of Part 2's `C`-language optimization.

3 Part 1: Basic Embedded Software-Development Skills

We will use the QEMU (Quick EMUlator) [15], a free and open-source hypervisor that can be used as a machine/hardware emulator. When used as a machine emulator, QEMU can run operating systems made for one machine (e.g., the Gumstix ARMv5 platform) on a different machine (e.g., your own PC running Linux). This kind of emulation environment can offer advantages: (i) it allows you to gain experience *immediately* in writing programs for the ARM processor without needing any access to a physical ARM processor, and (ii) it will be useful *later* in debugging your ARM programs, even when you have access to the physical ARM hardware (as you will for subsequent labs in 18–342).

List of tasks. You are asked to perform the following tasks to introduce you to best practices in embedded software development and to get you ready for more advanced software development in future labs:

1. Create and use a source repository for your team (Section 3.2).
2. Check out files from a Git repository (Section 3.3) *(15 points)*.
3. Set up QEMU to emulate the Gumstix ARMv5 hardware (Section 3.4.1).
4. Write a simple `C` program for the Gumstix (Section 3.6.1) *(20 points)* .
5. Write a simple ARM assembly-language program for the Gumstix (Section 3.6.2) *(20 points)* .
6. Disassemble and analyze binaries for the Gumstix (Section 3.6.3) *(20 points)*.
7. Write a `Makefile` to automate the build process (Section 3.7) *(20 points)*.

3.1 Revision Control

3.1.1 Introduction

A revision control or version control system is a toolset that assists programming teams in maintaining the history of and tracking changes made to a project that they are working on. It is an essential tool for even two person teams when projects require multiple iterations of design and testing. In this section, we shall describe the basic terminology related to a revision control system, how to set up a code repository, how to share work with your partners, how to track your partner's progress and how to work together, in parallel.

3.1.2 Basic Operations

All revision control systems have a concept of the 'set of things it is in charge of'. This set is usually a set of files or objects that the user has entrusted to the revision control system. This set of files is called a *repository* or a *depot*. Revision control systems track all changes made in a repository by different users. At regular intervals, the user commands the repository to take a snapshot of all the changes that have been made. The tool now considers the repository to be at a new *version* or *revision*. Changes such as what files have been added, what files have been removed, which lines have been changed and who changed them are all recorded. This operation is known as a *commit*. Suppose that a hardworking 18–342 group made a number of changes to their repository at 5 am. The next day, they notice that they had mistakenly overwritten an important file. Thankful that they had used a revision control system, they ask the tool to undo all of the changes they had made. This is known as a *revert*. Now consider two 18–342 students who live off campus and work independently. Instead of manually sending each other code, they each try to commit code to the repository that they have set up. They notice that their changes overlap and that they need to cleverly adjust their edits to not overlap. This operation of bringing together independent work on the same file is known as a *merge*. Any overlap in the code that is being merged causes a *merge conflict* that must be manually *resolved* by the merging user.

3.1.3 Client-Server Model

A client-server model of revision control is the traditional revision control model. The model consists of a primary server that hosts the repository or depot. Users never directly edit the repository. They *check-out* a copy of the code out of the main repository. They then edit their *local* or *working* copy to their hearts content. When they are ready to show the world the work they have done, they commit their code. But before they commit their code, they need to make sure that some one else hasn't changed the repository. Hence, they need to *update* their working copy and resolve any merge conflicts. They then commit their code to the main repository. This model uses minimal space on client systems and has a canonical version history for the entire repository. This model is used by traditional version control systems such as CVS [6], SVN [4] and Perforce.

3.1.4 Distributed Model

The upcoming models of revision control are all based on distributed, non-central models. In these models, there is no one main repository. Each user's codebase is a bona fide repository in and of itself. Users can *clone* each others repositories to get each other's working copies. Fundamental to this concept is the concept of *branching* and independent histories. Since each user has their own independent repository, they each maintain their own version of the file histories. The independent arcs of commits are known as *branches*. A branch in the repository signifies alternate histories of what happened to a repository. Branches can be merged if they share a common version in the past known as an *ancestor*. Since multiple versions of a repository and its history are in flight at the same time, developers can easily choose which version they want to work on and who they want to merge with. The tip of a branch that a developer is working on is known as the *head* of that branch. Since branching and merging are such common operations in a distributed model, special tools are used to deal with these. The most common is the three way merge used by Mercurial [13] and Git [19]. A more advanced and formal system called patch calculus is used in Darcs [17].

3.2 Introduction to Git

In this section, we shall introduce you to a popular revision control system called **git**. Git is a widely used, distributed model, multi-platform revision control system. It is already installed on the ECE cluster machines and on all Andrew Unix machines. Clients for it are available online [19] including versions for Windows. We shall now go through a list of basic tasks and commands that you must get familiar with. While we summarize some of the commonly used operations below, you are expected to have read at least the first 2 chapters of the online Git book [3] for a complete description on how to use Git – how to import your teammate's code, how to merge changes, how to revert any mistakes that you may have made, travel through time in your code and have parallel universes. The exercise at the end of this section assumes that you either have good working knowledge of Git or have read the two Chapters of [3]. The tasks described can be performed on the ECE or Andrew Unix systems although they can be easily adapted to other environments as well.

- To create a new repository, use the `git init` command.

  ```
  % git init --bare /path/to/repository/here
  ```

  This directory will now act as the central storage area for all the code in your project. Do not ever manually add, remove or modify files in this directory unless you want to seriously damage your repository. Also note that the files that you work on in the project will not be stored in plain text in the main repo. Do not go looking for them.

- To actually start creating and editing code, you will first need to get a *working copy* of the (currently empty) repository. This is the aforementioned *clone* operation.

  ```
  % git clone /canonical/path/to/repository/here localcopy
  ```

  The above command will create a working copy of the repository in the `localcopy` directory. If you are cloning a repository from a different machine, then use:

  ```
  % git clone ssh://[user@]host.xz[:port]/path/to/repository localcopy
  ```

- You are now free to edit your code, add files and, in general, carry out your work in the `localcopy` directory.

- After you have done some work, it is now time to tell the repository to track the changes you have made. Add the files for commit by issuing the following command.

  ```
  % git add file1 file2 file3
  ```

- This will add your files to a staging area for the commit. If invoked interactively as shown in the following command, this area allows to add entire files, parts of files, just certain chunks of code and also provides various other alternatives.

  ```
  % git add -i
  ```

- To tell git to take a snapshot of your current progress as a *commit*, issue the following command.

  ```
  % git commit
  ```

  This command will bring up a text editor to allow you to properly and informatively label the changes you are going to commit. The editor that it brings up depends on your `EDITOR` environment variable. This variable is usually set in your `.bash_profile`, `.cshrc` or equivalent file depending on your shell. To find out what shell you are currently using, run:

```
% echo $SHELL
```

You can also temporarily change the `EDITOR` variable by using "`export EDITOR=path/to/editor`" for bash or "`setenv EDITOR path/to/editor`" for csh.

- You can now review your commits and changes using the `git log` command. You can also get elementary help by using the `git help` command.
- This saves the changes as a state / commit in your local repository. You can use the SHA-1 hash of each commit to come back to that state of your repository.
- To make your changes available in the central repository, issue the following command -

```
% git push origin master
```

- The steps described above are extremely elementary. Git allows you to perform time-travel through your repository state using `git checkout`. It also allows you to have different parallel universes of your code using branches. Github provides a very good interactive tutorial [**?**] where you actually create a repository and perform various operations on it. Gitflow [**?**] is an interesting article on how to use per feature git branches and how you can use git to manage your project. *Warning: Git is a very powerful system which includes the ability for other people to do many things that you, personally, should not do. Some commands can leave your repository in a mess if not used with extreme caution. These commands include:*

  git reset

  git revert

  git rebase

  It is also not a good idea to use –hard or –force parameters without knowing what you are trying to do. As a general rule, do not try to change commit history after the commit has been pushed to the central repository.

  The files in the repository in the exercise below have HOWTOs to create a repository on andrew machines and adjust the directory permissions so that only you and your teammates can access and edit the files in the directory.

3.3 Revision Control System Exercise (15 points)

We have created an git repository at `/afs/andrew.cmu.edu/usr6/rgandhi/public/18342_lab1_repository` which you have read-access to. Clone this repository and then answer the following questions. Record your answers in a plain text file called `repository.txt`.

1. How many branches are there in the remote repository ?
2. Discounting hidden files, what are the files that are in the master branch ? What are their names? What is/are the content of the file(s) last committed on the master branch of the repository?
3. Who committed the files on the master branch and when (date and time) and what was the commit message?
4. Who made the last commit on all the other (other than the master) branches in the repository? When did (s)he commit them (date and time) and what was the commit message?
5. If you are on the master branch and modify a file (original file version v0, modified version v1) and then do a "git commit file", which version of the file gets committed?
6. If you are on the master branch and modify a file (original file version v0, modified version v1) and then add the file into the staging area by doing "git add file" and then commit the changes using "git commit file", which version of the file gets committed?

7. If you are on the master branch and modify a file (let's call the modified version v1) and then add the file into the staging area by doing "git add file". After adding the file into the staging area, let's assume you make more modifications to the same file (let's call the newly modified version v2) and then commit the changes. Which version of the file gets committed?

Warning: Github, Bitbucket and various other sites provide free hosting for git repositories. However, the default settings on these services will make your repositories (and your code) public. This is against the course policy. If you choose to use these services for hosting your code, it is your responsibility to make sure that the repositories are private and visible only to your team (you could apply for student accounts.)

3.4 QEMU

QEMU [15] is a generic machine-emulator that can run operating systems and programs built for one machine architecture on a different machine. In our case, we use QEMU to emulate the ARM-based Gumstix platform on your x86/64–based laptop/desktop. QEMU also supports running a GDB server, an invaluable feature which allows us to debug kernel-mode code. *We will refer to the emulated Gumstix and the operating system running on it as the guest machine and the guest OS, respectively, and your x86 laptop/desktop and the operating system running on it as the host machine and the host OS, respectively.*

This section explains how to set up the QEMU and how to use it to emulate the Gumstix hardware board. Although you will run your programs on the real Gumstix hardware in later labs, the process of learning how to emulate the Gumstix platform now will enable your team to be more productive later because your team can split up the work of future lab exercises, and team members will be able to test their individual code without needing access to the single Gumstix hardware-kit that will be assigned to your team.

Neither QEMU nor the ARM-version of GDB are installed on the CMU machines (`unix.andrew.cmu.edu`, `ece*.ece.cmu.edu`), so you must install them on your personal machine in order to use them. Although QEMU and GDB are available for Windows, Mac OS, and Linux, maintaining cross-compiling toolchains and cross-debugging environments is somewhat of a difficult, dark art. For this reason, we are going to describe how to set up the environment in Linux (specifically Ubuntu 12.04 LTS), using pre-packaged binaries whenever possible. If you don't already have a Linux development environment available, you can use VirtualBox to install and use Ubuntu 12.04 LTS, a popular Linux distribution.

3.4.1 Installing QEMU

The instructions here support installing QEMU version 1.0.50 on Ubuntu 12.04 LTS. If you would like to install and use QEMU + GDB on a Mac OS X system (or even a Windows system), the usage steps should be similar, but finding the exact packages for installation may be difficult. For Mac OS X, we recommend using the Homebrew package manager (though the packages may also be available through Fink or MacPorts).

As mentioned before, the easiest way to install QEMU is to use your favorite package manager to obtain the pre-packaged binaries. If you're feeling more adventurous, you can try compiling the latest version of QEMU from source. The packages needed for QEMU are:

1. `qemu` (base system which includes emulation support for x86/x86-64)
2. `qemu-system` (includes emulation support for other architectures, including ARM)

which can be installed with:

```
$ sudo apt-get update
$ sudo apt-get install qemu qemu-system
```

After successfully installing QEMU, you should be able to run:

```
$ qemu-system-arm --version
QEMU emulator version 1.0.50 (Debian 1.0.50-2012.03-0ubuntu2)
```

3.4.2 Obtaining system files

In order to emulate the Gumstix platform in QEMU, we will need to obtain:

1. a Flash image (holds Gumstix bootloader, operating system)
2. an SD card image (can hold Gumstix operating system files, and our kernel and tasks)

The flash image is the original contents of memory when the Gumstix boots. Typically flash memory is a ROM. However, with QEMU this is not necessarily enforced, so you may have to either re-create the image periodically or re-download it if it gets corrupted (symptoms: your code may run wild). **When you download the files locally, make a copy before using QEMU (the images are large).**

You can download the Flash image `flash.img` and the microSD card image `sdcard.img` from the course Blackboard site.

3.4.3 Using QEMU to emulate the Gumstix

Once you have all the necessary files, you can use this command to start emulating the Gumstix on your machine:

```
$ qemu-system-arm -nographic -M verdex -pflash flash.img -sd sdcard.img
```

Once you are done with emulating the Gumstix, you can quit QEMU and return back to the terminal by pressing `Ctrl-a x`.

3.5 Emulating Gumstix ARM-Based Hardware

3.5.1 Introducing the Qemu-Gumstix

Once Qemu-Gumstix is running, the first text that appears on the terminal reads:

```
U-Boot 1.2.0 (May  10 2008 - 21:17:19) - PXA270\@400 MHz - 1604

*** Welcome to Gumstix ***


DRAM :  64 MB
Flash: 16 MB
Using default environment
```

If you see the above text on the terminal running QEMU, this should indicate to you that you have successfully emulated the Gumstix hardware on your x86 system. Congratulations! *For the remainder of this handout, the terms "Gumstix hardware" or "Qemu-Gumstix" will refer to the QEMU-emulated Gumstix rather than the actual Gumstix hardware*. Similarly, any reference to the microSD card would be assumed to be a reference to the `sdcard.img` image that was used in order to emulate the Gumstix hardware.

U-Boot [5] is the Gumstix bootloader, and is a popular bootloader used in many 32-bit embedded systems. It is responsible for the initial setup of the hardware before passing control to the kernel. Incidentally, U-Boot will also be one of the target environments for programming in later labs.

As U-Boot executes, it detects a file system and Linux kernel on the microSD (`sdcard.img`) card. U-Boot loads the kernel into memory, and then transfers control to the Linux kernel itself. The Linux kernel initializes the remaining emulated peripheral hardware and launches the userspace `init` process. Eventually when userspace initialization is complete, you will be greeted with the login prompt on the serial console:

```
Welcome to the Gumstix Linux Distribution!

gumstix login:
```

Let's login:

1. Enter "root" as the login name.
2. If you are asked for a password, enter "gumstix" as the password.

This will bring you to a bash shell prompt:

```
Welcome to the Gumstix Linux Distribution!

gumstix login: root
Password:
Welcome to Gumstix!
[root@gumstix ~]#
```

In the following sections, the ("#") prompt indicates commands that should be typed, and the non-"#" statements that follow indicate program output (or logical equivalent) that will be displayed as result.

3.5.2 Shell & Environment

When you login to Qemu-Gumstix as root, the shell starts with a working directory of /root.[1] We recommend that all work performed on the Gumstix be located in /root or a subdirectory thereof.

Installed on the microSD rootfs is the full set of GNU Coreutils (cat, cp, echo, ln, mkdir, mv, pwd, rm, rmdir, etc.) [8] and other GNU utilities (find, grep, etc.). Although we do not expect you to be an expert, you should have some familiarity with the these tools as they will be essential for working in the Gumstix environment. If you are unfamiliar with them, please consider trying a GNU/Linux tutorial.

3.5.3 Tar & Compressed Archives

Tar [10] is the traditional Unix utility for creating file archives. In the context of the Gumstix platform, compressed file archives are useful for transferring multiple files over a slow serial connection.

Tar is typically used to compress a directory including all files and subdirectories within. Since the archives produced by tar are uncompressed, various file compression utilities are often used in tandem with tar. These compression utilities typically can only compress and decompress a single file, hence the need to pair them with an archiver. The combined output of the archiver and compressor is a compressed archive somewhat analogous to the popular ZIP archive format.

While tar itself is nearly ubiquitous (an occasionally-used alternative is cpio), each of the different compression utilities[2] have various benefits and tradeoffs in terms of compression ratio, speed, complexity, and patent restrictions.[3] The most popular Unix compressor is gzip [12], which offers median performance in terms of both compression ratio and encoding/decoding speed.[4]

To create a tar + gzip (.tar.gz) compressed archive of the example directory foo and all its contents, execute the commands:

```
# tar cvf foo.tar foo
(list of files added to archive)
...
(creates foo.tar)
# gzip -9 foo.tar
(creates foo.tar.gz)
```

Due to gzip's popularity as a compressor for tar archives, many versions of tar include an option to call gzip internally, allowing the entire compressed archive to be created in one command:

```
# tar cvzf foo.tar.gz foo
```

[1] This may be verified by executing "pwd".
[2] Includes gzip, bzip2, lzop, lzma, compress, and others.
[3] See LZW algorithm & Unisys patent debacle.
[4] Uses the same DEFLATE compression algorithm as the ZIP format.

3.5.4 Coding & Editors

The 18–342 staff recommends that most of your group's coding be done on a host machine and not on the Gumstix or the Qemu-Gumstix itself. Due to the fragile nature of the microSD cards, any significant code changes made on the microSD cards could be lost before you have a chance to transfer and backup.

Given the nature of the typical "edit, compile, debug" cycle, it is practical to make minor code changes on the Gumstix itself while debugging. However, be sure not to leave any code on the microSD that can't be rewritten or remodified in case of loss.

For when you do need to edit code on the Gumstix, the ubiquitous `vi` editor is available on the Gumstix. Although quite popular with Unix programmers, it is often regarded as unintuitive and as having a steep learning curve. Unless you're already proficient in `vi` or want to commit to learning it, it's probably best for you to do most of your coding on the host machine and copy the files to the Gumstix/Qemu-Gumstix.

3.5.5 Transferring Files Between Host and Qemu

There are several ways of transferring files between your laptop (host) and the Linux operating system (guest) running on QEMU. In this lab project, we will use the microSD card to transfer files between the host and the guest operating systems.

To transfer files between the host and the guest systems, you will first need to mount the microSD card image file (`sdcard.img`) using the following commands on your host operating system
`sudo mkdir /media/bootfs` to create the directory `/media/bootfs` if it did not exist before.
`sudo losetup /dev/loop0 /path/to/sdcard.img`
`sudo kpartx -a /dev/loop0`
`sudo mount /dev/mapper/loop0p1 /media/bootfs`.

Once `sdcard.img` is mounted as a directory on `/media/bootfs`, you will be able to copy all the relevant files that you want to transfer to the Qemu-Gumstix using `sudo cp /path/to/files /media/bootfs/` followed by `sync`. The `sync` command will flush any buffered writes to the microSD card image.
The microSD card is mounted on the guest running on `QEMU` as `/media/card`. However, if you are running `QEMU` and you add files to `/media/bootfs` on the host, the files will not show up on the guest operating system until you remount the microSD card filesystem again. In order to accomplish this remounting, use the following commands on the guest running on QEMU[5]
`umount /media/card; sleep 1; mount /media/card`
The `sleep` command has been added to introduce one second delay between un-mounting and re-mounting the filesystem. Once the files are available on the guest OS in `QEMU` at `/media/card`, they can be copied over to the home directory.

To make sure that you understand the process, run `QEMU` as explained in Section 3.4.3. Create a text file `test.txt` on your host laptop and transfer this file to Qemu-Gumstix using the steps described above. In the following sections, you are free to create the files on your host laptop and transfer them to Qemu-Gumstix using the steps described above (however, be aware that all compile, assembling, etc. will need to be done on the Qemu-Gumstix).

Use the following commands on your host machine to unmount the `sdcard.img` once you are done with emulating Gumstix.
`sudo umount /media/bootfs`
`sudo kpartx -d /dev/loop0`
`sudo losetup -d /dev/loop0`

[5]You may want to create a script so that you don't need to type these commands all the time.

3.5.6 Compilers & Development

Traditionally, code for an embedded system is compiled on a desktop or workstation computer with a cross compiler.[6] To maximize lab consistency and to minimize the software requirements of the host PC, 18–342 labs will actually be compiled on Qemu-Gumstix itself.

The microSD rootfs features two compilers, a GCC ARM native compiler (`gcc`) [7], and a GCC AVR cross compiler (`avr-gcc`). In addition, the microSD rootfs contains both AVR Libc [16], and (ARM Linux) uClibc [1], as well as other traditional Unix development tools such as diff/patch and GNU make.

3.6 Programming Exercises

The following programming exercises are meant to serve as a tutorial-like introduction to writing, compiling, executing, and debugging code on the ARM platform. You will perform all of the following exercises on the Qemu-Gumstix. While the code is simple, the process of going through these steps will be invaluable in helping you to prepare for subsequent labs.

3.6.1 "Hello World" on the ARM (20 points)

Complete these steps to write a "Hello world!" application on ARM:

1. Boot the Qemu-Gumstix and login as `root`.
2. Create a `/root/lab0/hello` project directory:

   ```
   # mkdir -p ~/lab0/hello
   # cd ~/lab0/hello
   ```

3. Using vi, edit the file `hello.c`. Or, alternatively, create the file `hello.c` on the host laptop using an editor of your choice and once you have written the file (as described below) transfer it to the Qemu-Gumstix using the steps described in Section 3.5.5
4. In `hello.c`, write a version of the canonical "Hello world!" program that:
   - Writes the string "`Hello world!`" followed by a new line to *stdout*.
   - Terminates with exit status 0.
   - Style and comment your code properly.
5. On Qemu-Gumstix, compile `hello.c` by executing the commands below, and verify that your output is proper:

   ```
   # gcc -Wall -Werror -o hello hello.c
   # ./hello
   Hello world!
   ```

3.6.2 "Goodbye World" on the ARM (20 points)

Complete these steps to write a "Goodbye world!" application on ARM:

1. Generate the assembly code for the `hello.c` program:

   ```
   # gcc -S -Wall -Werror hello.c
   (creates hello.s)
   ```

2. Create a new `/root/lab0/goodbye` project directory.

[6] A cross compiler generates code for a target platform that is different from the host platform on which the compiler executes.

3. Copy the `hello.s` assembly code to the new project directory and rename it `goodbye.s`.
4. Modify `goodbye.s` assembly code so that the new behavior of the program is:
   - Writes the string "`Hello world!`" followed by a new line to *stdout*.
   - Writes the string "`Goodbye world!`" followed by a new line to *stdout*.
   - Terminates with exit status 42.
   - Comment your code properly.
5. On Qemu-Gumstix, assemble `goodbye.s` by executing the commands below, and verify that your output matches:

```
# gcc -o goodbye goodbye.s
# ./goodbye
Hello world!
Goodbye world!
```

3.6.3 Disassembly on the ARM (20 points)

Complete the following steps to disassemble the "Hello world!" application:

1. Change back to the `/root/lab0/hello` project directory.
2. Disassemble the `hello` executable with the following commands:

```
# objdump -d hello > hello-d.txt
# objdump -D hello > hello-bigd.txt
```

3. Transfer `hello-d.txt` & `hello-bigd.txt` to the host machine and analyze them.

Each question is worth five points. Record your answers to the following questions in a plain text file called `disassembly.txt`:

1. What is the entry point address of the program? (Hint: The `readelf` program may provide a clue.)
2. What is the name of the first function branched to in the program? (Hint: One of `readelf -s`, `readelf -r`, `objdump -t`, or `objdump -T` may provide a clue.)
3. What is the *key* difference between the output of `objdump -d` (`hello-d.txt`) and `objdump -D` (`hello-bigd.txt`)?
4. Is the interpretation of the instructions under the `.rodata` section of `hello-bigd.txt` correct? What does this interpretation mean?

3.7 Automating the Build Process with `make` (20 points)

Manually rebuilding executables as part of the "edit, compile, debug" cycle quickly becomes a time consuming and tedious task, especially for large projects with many source files. The traditional Unix development utility used to automate and manage the build process is `make` [9].

Projects using `make` ship with a `Makefile` which describes how to build various "targets" of the project from source files. When `make` is invoked, it builds the first target listed in the `Makefile`, which is typically the project executable in its default configuration. Most projects contain other useful targets such as "`make clean`" which typically removes all of the temporary object files used in the building process.

3.7.1 `Makefile`s at a Glance

A `Makefile` is a plain text file that contains a set of rules. Each rule describes how to generate a target file from a list of prerequisite files and a list of shell commands. These rules have the form:[7]

```
target: prerequisites ...
        commands to build target from prerequisites
        ...
```

If one of the prerequisite files specified by a rule doesn't exist, `make` attempts to build that prerequisite from another rule that specifies the prerequisite as its target. Once all prerequisites are satisfied, `make` builds the target by executing the build commands.

Once a target is built, it will not be rebuilt by subsequent invocations of `make` unless a prerequisite is modified (and, thus, making the target out of date). This feature enables `make` to automatically rebuild the minimum number of files to generate an up-to-date target, speeding up the compile portion of the "edit, compile, debug" cycle.

3.7.2 Example `Makefile`s

A simple example `Makefile` that is sufficient for building the "Hello world!" application:

```
hello: hello.c
        gcc -Wall -Werror -o hello hello.c
```

To build the hello executable, execute "`make hello`" or even "`make`".

Programs that consist of multiple `C` source-files are typically built in multiple stages. First, each source file is compiled into a separate object file, and second, each object file is linked to form the final executable. Another simple example `Makefile` that demonstrates this approach is:

```
baz: foo.o bar.o
        gcc -o baz foo.o bar.o

foo.o: foo.c common.h
        gcc -c -Wall -Werror -o foo.o foo.c

bar.o: bar.c common.h
        gcc -c -Wall -Werror -o bar.o bar.c
```

One of the major advantages to separating the compile and link stages is that it minimizes the amount of rebuilding necessary to incorporate changes from a single source file. For example, if a change is made in `foo.c`, only `foo.o` and `baz` is rebuilt since `bar.o` remains unaffected by the changes. However, if a change in `common.h` is made, then all targets need to be rebuilt.

3.7.3 Additional `make` Features

`make` provides a number of additional features that are utilized by `Makefile`s for most software packages to increase flexibility[8], and reduce redundancy compared to the simple examples presented earlier. Typical features include:

Variable Assignment & Substitution. `make` allows a `Makefile` to assign variables with the syntax "`var = value`", and substitution of variables into rules with the syntax "`$(var)`". Variable assignments may be overridden on the "`make`" command line. For example, most `Makefile`s assign the `CC` variable to `gcc` as the default compiler and write compile rules using "`$(CC)`" to invoke it. If you wanted to substitute the default compiler with the `avr-gcc` cross compiler, you would execute "`make CC=avr-gcc`" instead of "`make`".

[7] Commands specified in a rule *must* be indented by a tab, and not spaces.

[8] For example, by allowing the user to substitute for a different compiler.

Automatic Variables. `make` automatically assigns the variables `$@`, `$<`, and `$^` when evaluating commands for a rule: `$@` is assigned the file name of the target, `$<` is assigned the filename of the first prerequisite, and `$^` is assigned a string consisting of the filenames of all the prerequisites with spaces between them. This feature allows the rule:

```
foo.o: foo.c common.h
        gcc -c -Wall -Werror -o foo.o foo.c
```

to be simplified to:

```
foo.o: foo.c common.h
        gcc -c -Wall -Werror -o $@ $<
```

Implicit & Pattern Rules. `make` handles rules that only specify a target and prerequisites (i.e., rules that *don't* specify commands for building the target) by selecting an appropriate implicit rule to use to build the target. Many implicit rules are built-in, but they may be specified manually with a pattern rule.

A pattern rule is an ordinary rule that specifies a target, prerequisite, and commands for building the target, except that the file names for the target and prerequisite contain a wildcard ("%") that matches at the beginning of the file name. Pattern rules define how to build files of a certain type. For example, the following pattern rule specifies how to build an object file from any `C` source-file:

```
%.o: %.c
        gcc -c -Wall -Werror -o $@ $<
```

3.7.4 A Real-World `Makefile`

Combining the above `make` features, an example of a typical real-world `Makefile` (or, at least, one that you are likely to encounter in later labs) is:

```
CC     = gcc
CFLAGS = -O2 -Wall -Werror

objects = foo.o bar.o

default: baz

.PHONY: default clean clobber

baz: $(objects)
        $(CC) -o $@ $^

foo.o: foo.c common.h
bar.o: bar.c common.h

%.o: %.c
        $(CC) -c $(CFLAGS) -o $@ $<

clean:
        rm -f $(objects)

clobber: clean
        rm -f baz
```

In general, you will not be expected to write this kind of `Makefile` from scratch. However, it is expected that you understand what this `Makefile` does, and be able to modify it to include additional source files in your own projects.

3.7.5 `Makefile` Exercise

Complete these steps to write a simple calculator program:

1. Create a `/root/lab0/calc` project directory:
2. Create a `math.c` source file containing five functions; `add`, `sub`, `mul`, `cdiv`, and `mod`; that implement integer addition, subtraction, multiplication, division, and modulo (remainder of division) respectively. Each function should take two integer arguments and return an integer result.
3. Create a `math.h` header file that contains function prototypes for each of the five functions implemented in `math.c`.
4. Create a `calc.c` source file with a single `main` function that implements a simple calculator program with the following behavior:
   - Accepts a line of input on *stdin* of the form "`number operator number`" where `number` is a signed decimal integer, and `operator` is one of the characters "`+`", "`-`", "`*`", "`/`", or "`%`" that corresponds to the integer addition, subtraction, multiplication, division, and modulo operation respectively.
   - Performs the corresponding operation on the two input numbers using the five `math.c` functions.
   - Displays the signed decimal integer result on a separate line and loops to accept another line of input.
   - Or, for any invalid input, immediately terminates the program with exit status 0.
5. Create a `Makefile` using the above "Real-World Makefile" example (using the `example.mk` file provided within the support-code file, `part1.tar.gz`, as a template). Modify this `Makefile` so that it builds the executable `calc` as the default target. The `Makefile` should also properly represent the dependencies of `calc.o` and `math.o` (hint: try "`gcc -MM`").
6. Compile the `calc` program by executing "`make`" and verify that the output is proper. For example:

```
# ./calc
3 + 5
8

6 * 7
42
```

3.8 Completing Part 1

3.8.1 What to Turn In

When finished with this part of the lab, please submit *only* the following source code and project files (maintaining project directory paths) on Blackboard. Before submitting your files, create a folder `lab1-part1-xx` (where xx is your andrew id) that contains only the files listed below. Tar and gzip the folder and upload the .tar.gz version on Blackboard.

- `lab1/hello/hello.c`
- `lab1/goodbye/goodbye.s`
- `lab1/hello/hello-d.txt`
- `lab1/hello/hello-bigd.txt`
- `lab1/calc/math.c`
- `lab1/calc/math.h`

- `lab1/calc/calc.c`
- `lab1/calc/Makefile`

Please also submit your written answers to the questions asked in Section 3.3 and 3.6.3 in plain text format[9] to:

- `lab1/repository.txt`
- `lab1/disassembly.txt`

3.8.2 Where to Get Additional Help

Documentation for most Unix utilities are available as man pages (e.g., "`man objdump`"). Due to space constraints, man pages and other documentation are *not* included on the Gumstix. However, they should be readily available on any Linux machine including Andrew Linux servers and any machine in the ECE undergraduate cluster, or online [14]. Documentation on the Gumstix platform is available on the Gumstix wiki [11]. *Please use the 18–342 Piazza site, particularly the thread on Lab #1, to post your questions and to view the course staff's answers and tips for this lab.* Any and all updates to this lab handout or the lab instructions will be notified to the class via Piazza.

3.8.3 Cautionary Notes

You and your lab group are responsible for maintaining backups of source code while working on this and subsequent 18–342 labs. It is best for you and your team to store project source-code in Andrew or ECE AFS where it is backed up nightly.

The course staff also recommends the use of a revision control system such as Subversion [4], Mercurial [13], or Git [19] for managing source code among partners and to maintain version history. Although it is not necessary for this or future labs, the regular use of, and proficiency in, such a system will become valuable in later labs (and, in fact, in your career in the embedded industry) as code size and complexity increases.

4 Part 2:

Coming soon....

References

[1] E. Andersen. uclibc [online]. Available from: `http://www.uclibc.org/`.

[2] R. Bean. High speed tutorial: Subversion [online]. Available from: `http://svnbook.red-bean.com/en/1.4/svn.intro.quickstart.html`.

[3] S. Chacon. Pro git [online]. Available from: `http://git-scm.com/book`.

[4] CollabNet. Subversion [online]. Available from: `http://subversion.tigris.org/`.

[5] DENX Software Engineering. Das U-Boot: the universal boot loader [online]. Available from: `http://www.denx.de/wiki/UBoot`.

[6] Dick Grune. Concurrent versions system [online]. Available from: `http://www.nongnu.org/cvs/`.

[7] Free Software Foundation. Gcc, the GNU compiler collection [online]. Available from: `http://gcc.gnu.org/`.

[9] ASCII or UTF-8 encoding.

[8] Free Software Foundation. GNU Coreutils [online]. Available from: http://www.gnu.org/software/coreutils/.

[9] Free Software Foundation. GNU Make [online]. Available from: http://www.gnu.org/software/make/.

[10] Free Software Foundation. GNU Tar [online]. Available from: http://www.gnu.org/software/tar/.

[11] gumstix. Gumstix support wiki [online]. Available from: http://docwiki.gumstix.org/.

[12] J. loup Gailly and M. Adler. The gzip home page [online]. Available from: http://www.gzip.org/.

[13] M. Mackall. Mercurial [online]. Available from: http://www.selenic.com/mercurial/.

[14] Name.net. Linux man pages [online]. Available from: http://www.linuxmanpages.com/.

[15] qemu. Qemu support wiki [online]. Available from: http://wiki.qemu.org/Manual.

[16] T. A. Roth and J. Wunsch. AVR libc home page [online]. Available from: http://www.nongnu.org/avr-libc/.

[17] D. Roundy. Darcs [online]. Available from: http://darcs.net/.

[18] D. Sandler. Source control in ten minutes: a subversion tutorial [online]. Available from: http://www.owlnet.rice.edu/~comp314/svn.html.

[19] L. Torvalds and J. C. Hamano. Git: Fast version control system [online]. Available from: http://git-scm.com/.