Lab #1: ARM Programming Environment and Code Optimization

18–342 Fundamentals of Embedded Systems
Carnegie Mellon University

Out: September 23, 2013, 11:59 pm EDT
Due: October 7, 2013, 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 [10] 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) [14], 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 3.1.5).
2. Check out files from an SVN repository (Section 3.1.6) *(15 points)*.
3. Set up QEMU to emulate the Gumstix ARMv5 hardware (Section 3.2.1).
4. Write a simple `C` program for the Gumstix (Section 3.4.1) *(20 points)* .
5. Write a simple ARM assembly-language program for the Gumstix (Section 3.4.2) *(20 points)* .
6. Disassemble and analyze binaries for the Gumstix (Section 3.4.3) *(20 points)*.
7. Write a `Makefile` to automate the build process (Section 3.5) *(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 [5], SVN [3] 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 [12] and Git [18]. A more advanced and formal system called patch calculus is used in Darcs [16].

3.1.5 Introduction to SVN

In this section, we shall introduce you to a popular revision control system called Subversion. Subversion is a widely used, client-server 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 [3] including GUI versions for Windows XP and Vista. We shall now go through a list of basic tasks and commands that you must get familiar with. 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 `svnadmin` command.

  ```
  % svnadmin create /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.

  Note: Due to an issue involving the bad interaction of AFS with the back-end of Subversion, file locking is flaky at best if Subversion is used on Andrew. The course staff recommends that, should you choose to use subversion, that you host your repository on your own personal boxen or take regular snapshots of your Subversion repository should you choose to host it on Andrew.

- 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 *check-out* operation.

  ```
  % svn checkout file:///canonical/path/to/repository/here localcopy
  ```

  The above command will create a working copy of the repository in the `localcopy` directory. If you are checking out a file or project from a different machine than the repository, then use:

  ```
  % svn checkout svn+ssh://username@server.ip.addr/path/to/repository/here  localcopy
  ```

- You are now free to edit your code, add files and, in general, carry out your work in the `localcopy` directory. For hints on how to organize your directory and a more thorough tutorial, consult this page by Sandler at Rice University [17].

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

  ```
  % svn add file1 file2 file3
  ```

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

  ```
  % svn 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 `svn log` command. You can also get elementary help by using the `svn help` command.

- The steps described above are extremely elementary. Use this online SVN tutorial [2] for a more complete description on how to import your teammate's code, how to merge changes and how to revert any mistakes that you may have made.

3.1.6 Revision Control System Exercise (15 points)

We have created an SVN repository at /afs/andrew.cmu.edu/usr6/rgandhi/public/18342_lab0_repository which you have read-access to. Check out a working copy of this repository and then answer the following questions. Record your answers in a plain text file called repository.txt.

1. Discounting hidden files, what are the files that are in the repository? What are their names? What are their contents?

2. What is the userid of the person who committed those files?

3. When did (s)he commit them (date and time) and what was the commit message?

3.2 QEMU

QEMU [14] 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.

3.2.1 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)
```

3.2.2 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.

3.2.3 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`.

3.3 Emulating Gumstix ARM-Based Hardware

3.3.1 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 [4] 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.

3.3.2 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.) [7] 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.

3.3.3 Tar & Compressed Archives

Tar [9] 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 [11], 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.

3.3.4 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.

3.3.5 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 3.2.3. 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.

3.3.6 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`) [6], and a GCC AVR cross compiler (`avr-gcc`). In addition, the microSD rootfs contains both AVR Libc [15], and (ARM Linux) uClibc [1], as well as other traditional Unix development tools such as diff/patch and GNU make.

3.4 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.

3.4.1 "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 3.3.5
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!
```

3.4.2 "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!
```

3.4.3 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?

3.5 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` [8].

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.

3.5.1 Makefiles 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.

3.5.2 Example Makefiles

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.

3.5.3 Additional make Features

make provides a number of additional features that are utilized by Makefiles 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 "$(var)". Variable assignments may be overridden on the "make" command line. For example, most Makefiles assign the CC variable to gcc as the default compiler and write compile rules using "$(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 $@ $<
```

3.5.4 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.

3.5.5 `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
```

3.6 Completing Part 1

3.6.1 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-group-xx` (where xx is your group number) 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 3.1.6 and 3.4.3 in plain text format[9] to:

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

3.6.2 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 [13]. Documentation on the Gumstix platform is available on the Gumstix wiki [10]. *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.

3.6.3 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 [3], Mercurial [12], or Git [18] 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: Code Optimization

The 18–342 lectures should have provided you with insights about the ARM architecture, and also about possible optimization strategies that are architecture-independent as well as those that are ARM-specific. The purpose of Part 2 of Lab #1 is to give you hands-on experience with the content covered in the lectures. Specifically, you will be required to optimize a given piece of code using ARM assembly-language optimization strategies and, then, using high-level optimization strategies.

4.1 Assembly-Language Optimization (50 points)

4.1.1 Support Code

You have been provided with support code for this part of Lab #1 in the file `part2asm.tar.gz`. You should see three files in the extracted directory, as indicated from the output of "`ls -al`" in this directory.

```
-rw-r--r-- 1 root root 399 Sep 24 00:31 Makefile
-rw-r--r-- 1 root root 695 Sep 24 00:31 main.c
-rw-r--r-- 1 root root 665 Sep 24 0o:31 strTable.s
```

Of the two source-files, the `.c` file is written in `C` and the `.s` file is written in the ARM assembly-language. If you examine these files, you will notice that `main()` in the `main.c` file invokes a function called

[9] ASCII or UTF-8 encoding.

`strTable()`. The `strTable()` function is defined in a source-file called `strTable.c` that is *not* provided to you. Instead, the ARM assembly-language equivalent of `strTable.c` is contained in the `strTable.s` file. In case you are curious, this is how we generated `strTable.s` from `strTable.c`.

```
[root@gumstix lab1]# gcc -S -O -fomit-frame-pointer -mcpu=xscale strTable.c
```

Use the provided `Makefile` to link the two files, `main.c` and `strTable.s`, together to form an executable called `part2asm`.

4.1.2 Focus of Exercise

As we have indicated in the previous section, we used the compile-time `-O` optimization flag with `gcc`, in order to generate the ARM assembly-language code contained in the `strTable.s` file. However, there is still plenty of room for further manual optimization, based on the principles that were taught in our 18–342 lectures.

You need to optimize only the ARM assembly-language code in the `strTable.s` file, specifically, to improve the performance of the `strTable()` function. You do *not* need to worry about performing optimizations in any other part of the program, outside of the `strTable()` function. In fact, we expressly disallow any optimizations in `main.c` (this is an important point because we intend to use a different `main.c` file to test your optimized `strTable()` function).

We have intentionally not provided you with a high-level specification of what the function `strTable()` accomplishes. You can infer what the `strTable()` function does by examining its assembly-language equivalent (that we have provided).

4.1.3 Test Cases

Here is what you should see when you run the executable, `part2asm`, on the Gumstix.

```
[root@gumstix lab1]# ./part2asm
PRE: src:  theinitialstring (16 bytes), dst:  PENGUINS (8 bytes)
POST: src:  EhGiIitiaNsPriSgU (16 bytes), dst:  PENGUINS (8 bytes)
```

We recommend that you generate additional test cases of your own, to verify for yourself that the (given) unoptimized and (your) optimized versions of the `strTable()` function produce the same output. To generate additional test cases, you will need to edit the `main.c` file to modify the `src[]` and `dst[]` strings.

After your perform your optimizations, your code should still work correctly under all of the test cases. Thus, the optimized `strTable()` should still be *logically* equivalent to its unoptimized version that we have provided.

4.1.4 Grading Criteria

Keep in mind that you are aiming for both time and space optimization, i.e., you are trying to reduce both the run-time as well as the code size. Therefore, you will be graded on the following cost-based criteria. The overall cost of your optimized version will be a combination of two sources of cost for each instruction: instruction memory-cost, and instruction runtime-cost. Here is a breakdown of the incurred costs.

Instruction memory-cost

- ALL instructions = 5pts each

Instruction runtime-cost

- Each conditional branch instruction costs 3pts
  Rationale: Conditional branch instructions, in most cases, will stall the pipeline of a processor. In terms of runtime, it could cost up to 3 cycles (assuming a 3-stage pipeline).

- Each load-store instruction costs 3pts
  Rationale: Memory-loads and memory-stores are expensive operations, and in most ARM processors, the pipeline is not designed to skip stages.

- Each load-store-multiple instruction costs 3pts * number of registers involved
  Rationale: If 3 registers are loaded together, the memory bandwidth is the same as 3 separate loads. In comparison, the instruction memory-cost will be reduced.

- ALL other instructions cost 1pt each

Course staff will assess your cost based on
`PRE: src: theinitialstring (16 bytes), dst: PENGUINS (8 bytes)` input.

To determine your overall cost, you should trace through your program, figure out what is being executed and what is dead code (so you can eliminate it), look at each instruction, and compute its associated cost. To make your life easier, you can use `gdb`. While the debugger might not work on the Gumstix out-of-the-box, we have provided instructions in the following section to help you with this process.

While doing optimizations, keep in mind that `gdb` will fail if you change the value of `lr` at any point. It is in your best interests to refrain from modifying `lr` as well as the last line in `strTable`, (i.e., `mov pc, lr`). As far as the grading criteria, we will count this instruction, `mov pc, lr`, as a branch instruction rather than as a conditional-branch instruction.
To give you an idea of the cost of a program, here is the cost for the baseline/provided implementation:

Type	Count
Instructions	26
Load-Store	37
Conditional-branch	41
Normal	85
TOTAL	449

And, for comparison, here is the cost of an optimized sample version of this program:

Type	Count
Instructions	21
Load-Store	28
Conditional-branch	25
Normal	83
TOTAL	347

With all that said, students who turn in correct, valid implementations achieving a cost below 380 will be guaranteed at least 90% of the points on this section, given that they didn't lose any style or lateness points. Course staff may send updates on this value as the assignment progresses. Extra credit points will be given to top 5-10% of teams or for other creative endeavors.

4.1.5 Debugging with GDB

To run `gdb`, you would do something like:

```
[root@gumstix lab1]# gdb part2asm
```

Now, at the `gdb` prompt, BEFORE attempting to run your program, enter the command:

```
(gdb) set osabi default
```

To avoid having to do this every time you start `gdb`, you can make this automatic by simply placing the `"set osabi default"` command in the `~/.gdbinit` file (please note that your gumstix may already have this line in this file).
To summarize, something like this should get you started on debugging your program:

```
[root@gumstix lab1]# gdb part2asm
(gdb) set osabi default
(gdb) break strTable
(gdb) run
```

To step through your program you can then go ahead and use the step command. There are many resources online on gdb usage.

4.1.6 Hints

As you know from the lectures, loop unrolling has the potential to reduce the number of branches taken and the number of instructions executed; but at the same time, you need to keep in mind that an excessive unrolling of loops increases the size of the code. Remember that this lab is an interesting trade-off between two perspectives–runtime performance and the amount of memory needed. If you get stuck while optimizing from one perspective, you can always switch to the other perspective.

Because the nature of the solution might vary and there is, by no means, a ”perfect” solution, the course staff may give up to 10 points of extra credit for outstanding performance. Outstanding performance involves turning in code such that all versions are well-documented and that the overall performance index is within that of the top few groups in the class.

4.2 C-Language Optimization (50 points)

4.2.1 Focus of Exercise

Oddball Optimization. Given an array of $2n-1$ integers containing integers between 1 and n inclusive, every integer, except one, will appear twice in the array. Write a function that finds the integer that occurs only once. A base implementation is given to you in part2c.c, in the function oddball. No credit will be given to you if you turn it in as is. Assume that the input array is always properly formatted and that it follows the guidelines mentioned above. Also, assume $n \geq 2$.

Show *at least two ways* of optimizing this algorithm and place them both in part2c.c. Indicate using comments about the optimizations you have made and why you made them. Label the best version of your algorithm as function oddball. Keep all other versions of the oddball function in part2c.c either commented out (use #ifdefs) or label the function differently. We will grade you based on the runtime complexity and memory usage. We will deduct points for any inefficient or unnecessary code in your function. Please understand the function oddball before starting to optimize it.

With the Makefile and part2c.c provided, doing make N=100, to compile the code will cause the value of n to be set to 100. We list the code below for your convenience:

```
int oddball(int *arr, int len)
{
    int i, j;
    int foundInner;
    int result = 0;
    for (i = 0; i < len; i++)
    {
        foundInner = 0;
        for (j = 0; j < len; j++)
        {
            if (i == j)
            {
                continue;
            }
            if (arr[i] == arr[j])
            {
                foundInner = 1;
            }
        }
        if (foundInner != 1)
        {
            result = arr[i];
```

```
        }
    }
    return result;
}
```

4.2.2 Support Code

The archive `part2c.tar.gz` on Blackboard contains the following files

1. Makefile: `makefile` will compile your code into multiple binary files
   - `part2_def`: default binary using base implementation of `oddball`
   - `part2_o1`: oddball optimization version 1
   - `part2_o2`: oddball optimization version 2
2. `main.c`: driver for the program (you do not need to optimize this function).
3. `part2c.c`: you should add your optimized code here, make sure the function signature does not change
   - Under `#ifdef OPTIMIZE1`, place your first version of `oddball` here
   - Under `#ifdef OPTIMIZE2`, place your second version of `oddball` here

4.3 Completing Part 2

4.3.1 What to Turn In

For this part of Lab #1, your group needs to create one archive: `lab1-part2-group-xx.tar.gz` (where `xx` is your group number) and upload it on Blackboard. This archive should contain

- `strTable.s` which should contain your optimized code for `part2asm` of this lab
- `part2c.c` which should contain your optimized code for `oddball` function

Before submitting your optimized code, you should verify that your `strTable.s` file actually compiles with the provided `main.c` in `part2asm` to produce a working executable, `part1`. We may test your optimized code by compiling your submitted `strTable.s` file with a new `main.c` file where we have changed the `src[]` and `dst[]` strings to be other than the test cases we have provided you.

Although `main.c` is not mandatory for submission for oddball optimization, you can submit it if you wrote any validation code for us to see.

4.3.2 Documentation Required with Solution

Add a comment at the top of each file with the Andrew IDs of all group members. For each version of your function, document the improvements that you have made to the function, and how it is better than the previous version (in terms of memory and runtime complexity). Include a separate text file, `part2.txt`, with your submitted code to document all of your attempted optimizations.

Example: `SUBS r1, r1, #1 @ replaced 2 instructions SUB and CMP with SUBS`

If you fail to adequately document all versions of your `strTable()` function and the optimization mechanisms that you have chosen to use, you will lose points. While grading your assignment, the course staff will run a series of validation tests on your submitted code. If your code fails our validation tests, points will be taken off based on the severity of the error(s).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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