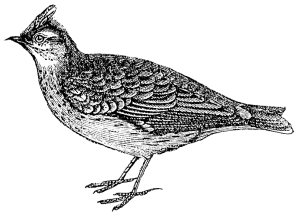


Larc – A Little Architecture for the Classroom

Lab Manual

Version 1.1

Marc Corliss

Hobart and William Smith Colleges

corliss@hws.edu

http://math.hws.edu/mcorliss

About the Author

Marc Corliss is an Assistant Professor in the Mathematics and Computer Science Department at Hobart and William Smith Colleges. In 2006, Professor Corliss received his PhD from the University of Pennsylvania in computer science. His research interests are in system design, and in particular the design of compilers and processors. He is also interested in computer science education and building new tools for teaching computer systems courses.

Acknowledgements

First and foremost, I must thank my summer research student, Rob Hendry, who helped design the Larc ISA and assembly language, and built or helped build many of the tools described in this manual. Rob contributed significantly to Larc and I was fortunate that he agreed to work with me on this project. In addition, I wish to thank the Provost's Office and the Department of Mathematics and Computer Science at Hobart and William Smith Colleges for their funding and support for this work.

Contents

1 Introduction

Computer architecture is the study of how computers are designed and implemented, in terms of hardware as well as low-level software. For instance, computer architecture explores the hardware structures that form a modern computer. It also explores the (low-level) software interface exported by the hardware and how computers can be programmed through this interface without the aid of a translator. Finally, computer architecture looks at the software components needed to make a computing system practical.

As one might imagine, computer architecture is a critical component of an undergraduate computer science student's education. Obviously, computer scientists who work with hardware must have a good understanding of computer architecture. Computer scientists who build system software such as an *operating system* (the software, such as Windows 7 or Mac Snow Leopard, that manages the machine) must also naturally know how computers are designed and implemented. Even computer scientists who only write non-system software must know some computer architecture. For example, computer scientists working in computer graphics must have a deep understanding of how the monitor works and is programmed if they want to build high-performance graphical applications. The same is true for many other areas of computer science (*e.g.*, databases, embedded systems).

There are many different hardware and software components that come together to form a modern computer system. Of course, the hardware *processor* or *central processing unit (CPU)* executes programs, using *memory* to store the program and program data. *Input/output devices*, also called peripherals, such as the keyboard and monitor, make the computer interactive. The *operating system* (OS) software manages the resources of the machine and provides some key abstractions, which make it much easier to develop applications and use the machine. Finally, translation tools such as the *compiler* convert *higher-level* programs, such as Java or C programs, into a language the machine can understand, *i.e.*, *machine language*. Computer scientists should understand (at least, functionally) how all of these components work, and how they interact with one another.

Just understanding how machines are programmed can be challenging. In modern computing systems, so much happens between the high-level program (*e.g.*, Java program) and the code that is actually executed. First and foremost, computers process only programs and program data that are expressed in binary, *i.e.*, sequences of 0's and 1's. Because working in binary is difficult for

most computer programmers, compilers are used, which enable programmers to work in high-level languages like Java. These high-level languages allow computer programmers to write in a mathematical, text-based format, which is much easier to comprehend.

More significantly, machine languages are more primitive than most high-level languages. For example, most machine languages have only a few simple data types such as unsigned integers, signed integers, and characters but not more complex data types such as arrays, linked lists, or objects. Machine languages also provide only a few simple operations for manipulating these data types (*e.g.*, addition, subtraction). Of course, any program that can be written in a high-level language can also be written in machine language, otherwise, the high-level program could not be run on the computer. But this leap from natural, high-level languages to binary, low-level machine languages is challenging for students who have not studied computer architecture.

The compiler significantly simplifies application development by allowing programmers to work in high-level languages. The OS also simplifies development by providing an abstract view of the underlying hardware. The OS exports an interface, via what are called *system calls*, which allows the compiled program to access the machine's input and output devices such as the hard drive, monitor, mouse, *etc.* (as well as performing other critical services). But it is important in many contexts to understand how input and output devices are programmed directly.

One of the primary goals of this work is to demystify to the student how computers are programmed in the absence of compilers, operating systems, and other software tools. At the same time, the assignments in this manual will help show how all of these tools come together to form a complete computing system. Because of time constraints, we will not explore each component in detail; that is best done in a separate course (*e.g.*, a course in operating systems or compilers). Instead, the assignments give the student a broad view of computer systems.

To learn computer architecture, the student must look at and implement parts of an some existing architecture. Ideally, this architecture would be a commercial architecture used in standard machines on the market. Unfortunately, many commercial computers have incredibly complicated architectures (*e.g.*, x86). This complexity is often a result of market pressure, changes in the industry over time, and the need for interoperability with existing hardware and software (among other things). Much of this complexity does not represent conceptual challenges. Furthermore, large parts of the architecture are not publicly accessible since they are a part of a commercial product. As such, commercial architectures are not ideal for the classroom. The ideal classroom computer architecture contains the conceptual challenges of commercial architectures without many of the

mundane details. Furthermore, each component of the architecture should be available to the student, perhaps as part of a course assignment, which the student helps build.

In this work, we have defined a simple architecture and created several related assignments that help students thoroughly understand computer architecture. Our architecture is called Larc, which stands for "Little ARChitecture". Larc is based on the MIPS commercial architecture [3], although it is simplified in several ways to make easy to work with in a single-semester introductory architecture course. First, unlike most commercial architectures (with the exception of some used in embedded systems), Larc is a 16-bit architecture, *i.e.*, the storage width of an individual value is 16 bits. The width defines the range of values that can be expressed, *i.e.*, 2^{16} unique values. The width also determines the size of the memory address space (*i.e.*, number of memory units): 2^{16} bytes or 64 KB. Most commercial architectures are 32-bit or 64-bit wide, which provide much larger address spaces. Larc also has only 16 types of instructions (*i.e.*, operations that can be directly executed on the processor), unlike other commercial architectures, which often have more than 100. Finally, Larc has fewer registers (*i.e.*, a small set of values, which the instructions operate on) than most architectures. Larc has 16 registers while many commercial architectures have 32 (although x86 actually has fewer registers than Larc).

As with any architecture, Larc is split into an interface and implementation. The *instruction set architecture (ISA)*, or architecture for short, is the interface to the machine and specifies the aspects of the hardware that are visible to software. For example, the ISA would include, among other things, a specification of each supported machine instruction, the number of registers, and the size of the address space. The *microarchitecture* is the underlying implementation of the ISA. The microarchitecture is hidden from software. Using this well-known design principle (*i.e.*, *abstraction*) allows hardware designers to modify the microarchitecture without impacting the software that runs on top of the hardware (so long as the ISA remains unchanged). Because this manual is intended for an introductory course, it will focus primarily on the ISA although aspects of the microarchitecture are covered in some places.

We have defined the ISA and machine language for Larc. We have also defined the Larc *assembly language*, a textual form of machine language. We have written several thorough sections describing the ISA and assembly language, along with sections describing how to program in both machine and assembly language. We have also built several tools for Larc including a *functional simulator* for running a program written in Larc machine code (called functional because it models the behavior of a Larc machine but not the timing, *i.e.*, how long it actually takes to run a

program); a *debugger* for identifying errors in a Larc machine or assembly program; an *assembler* for translating Larc assembly code into Larc machine code; a tiny operating system for handling input and output with a keyboard and (character-based) monitor; and a compiler for a simple C-like language. In addition, we have designed several course assignments where students will program in Larc machine and assembly language, and make use of some of these tools. In other labs they will actually build some of these tools (*e.g.*, a functional simulator and assembler).

1.1 Project Goals

We used several project goals to guide our design of the Larc architecture.

Simple architecture. First, we wanted to make the architecture as simple as possible. For this reason, we created a simplified version of the MIPS [3] ISA (which, itself, is much simpler than most commercial ISAs). For instance, all instructions have a fixed width (16 bits) and encoding with only a few simple addressing modes. Unlike many commercial machines (excluding embedded processors), Larc is 16-bit wide with 16 instructions and 16 registers. These simplifications make it easier to implement architectural tools such as the assembler and functional simulator.

Programming-intensive assignments. We also wanted students to build many of the architectural tools (*e.g.*, assembler) that they would be making use of. This gives them a deeper grasp of the ISA and assembly language. Of course, it also means students must have prior experience programming, and moreover, be familiar with a particular programming language. We chose Java for the high-level programming language since it is used in many introductory computer science programming classes. The assignments in this lab were developed with the expectation that students would have taken one semester in Java. We assume that the students will not be familiar with other languages that are covered in this manual such as machine and assembly language.

Broad coverage. A third goal was to create assignments that would not only teach students about computer architecture but also provide them with an introduction to other courses in the broader area of computer systems such as operating systems and compilers. Although the assignments cannot go deeply into these areas, they help give students a broad understanding of computer systems.

Well-engineered infrastructure. We also wanted to build an infrastructure that is easy for the student to extend and modify. In most of the assignments, students need to work on only a single

file or Java class. There are many auxiliary tools such as a debugger that make it easier for students to complete the assignments.

Summary. In summary, we have attempted to create a simple architecture along with a set of assignments where students will get a glimpse into a number of areas in computer systems. The assignments will broadly cover many areas in computer systems such as operating systems and compilers at the expense of some depth. Still, because we expect students to come in with some Java programming experience, a few of the assignments, particularly those involving the ISA and assembly language, are deep. For instance, students will build a functional simulator in one assignment, which will give them a good understanding of the ISA (after all, they are implementing it!).

1.2 Intended Use

This infrastructure and manual is intended for use as a set of course projects in a computer architecture course. It should be supplemented with a traditional computer architecture textbook [6, 7, 8]. The principal thing that this manual lacks is a discussion of how computer hardware works. It does discuss some of the design issues with respect to the instruction set architecture. It also discusses in detail some of the low-level software components needed in a computing system (since these are built in course projects) including an assembler and disassembler.

In addition, these architecture projects and this manual are intended for students that already know some Java. It does not teach programming or programming in Java. Students should be familiar with Java or at the very least familiar with a similar object-oriented language (*e.g.*, C++, C#). For those students who are less familiar with Java, they may need to consult a Java reference book [1], while reading this manual.

As in the textbook by Patt and Patel [6], we use a bottom-up ordering of lab assignments, starting with hardware and moving up towards high-level programming languages. However, the assignments could be re-ordered with some work for a different approach. The assignments cover many different topics such as machine simulation, machine/assembly languages, assemblers, operating systems, and compilers. To fit all of these assignments in a single semester, many of them only skim the surface of the topic they cover, especially the assignments in operating systems and compilers. Therefore, the assignments in this manual give a broad introduction to computer architecture, and more generally, computer systems.

As discussed below, the last two lab assignments can be done in C as opposed to Java. We do not expect students to know C, although this may be the case in particular courses. Instead, the inclusion of C allows students to see how a simple high-level language (*i.e.*, C) is translated into assembly code by a compiler. Our intention is that instructors would teach some C concurrently with these two labs. This would allow an instructor to use a textbook, which incorporates some material on C, such as the one by Patt and Patel [6]. However, these labs can also be done in Java as well, for courses that do not cover any C.

Finally, this infrastructure and manual cover only introductory material in computer architecture and more generally, computer systems. Advanced topics in computer architecture are not covered such as pipelining, superscalar execution, speculation, multithreading, or multiprocessors. Furthermore, although there is a lab assignment in both operating systems and compilers, these labs are only an introduction to these two areas. While all of these topics are important, they are best covered in additional courses.

1.3 Future Work

In future work, we will extend this work in two ways. First, we will add course assignments, which ask the student to build a processor specification in Verilog. These assignments could be used in a second course on computer architecture and would help the student better understand the microarchitecture.

In addition, we also hope to build a complete but simple operating system to run on top of Larc, along with a set of course assignments using this OS. (Note: as discussed below, students do build an OS component in one assignment, however, this is far from a complete OS.) This OS will help connect the computer architecture and operating system courses, and demonstrate to students the interaction between the machine and OS.

1.4 Outline

The remainder of this manual is organized as follows. Section 2 describes the Larc toolset and provides instructions for installing the toolset. Sections 3-7 are informational sections describing different aspects of the Larc architecture. Sections 3 and 4 discuss the Larc ISA and directly programming a Larc machine, respectively. Sections 5 and 6 discuss the Larc assembly language and programming in Larc assembly, respectively. Section 7 discusses the Larc system architecture

for supporting a simple operating system and communicating with input/output devices such as a keyboard and monitor.

Sections 8-17 contain the lab assignments. There are 10 labs in total. In the first 8 labs, students are asked to build various architectural tools in Java, or write or analyze Larc machine/assembly programs:

- Section 8: writing a decimal/hexadecimal/binary number converter in Java (no required reading).
- Section 9: writing a mock arithmetic and logical unit (ALU) in Java (no required reading).
- Section 10: writing a Larc functional simulator in Java (Section 3 required).
- Section 11: finding errors in a simple Larc machine program (Section 3-4 required).
- Section 12: writing a Larc assembler in Java (Section 3-5 required).
- Section 13: writing a program for playing Tic Tac Toe in Larc assembly language (Section 3-6 required).
- Section 14: writing a tiny operating system (*i.e.*, trap and input/output handler) in Larc assembly language (Section 3-7 required).
- Section 15: writing a recursive function (using a stack for storage) for computing fibonacci numbers in Larc assembly language. (Section 3-6 required).

In the last two lab assignments, students are asked to build two more architectural tools. These can be built in either Java or C. (If implemented in C, these assignments are intended to act as an introduction to the language.) These include:

- Section 16: writing a disassembler for converting Larc machine code into Larc assembly code in Java or C (Section 3-5 required).
- Section 17: writing a compiler (at least, part of one) for a tiny subset of the C programming language written in Java or C (Section 3-5 required).

Many of the lab assignments assume that the student has read some of the informational sections. These are listed parenthetically in the bulleted lists above. (If doing the last two labs in C then some documentation on C may also be needed.)

Note: although we use a bottom-up order in the arrangment of our labs some re-arranging could be used for a top-down course in computer architecture. Furthermore, some of these lab assignments can be compressed or removed in a time-pressed course.

2 Toolset

This section gives an overview of the Larc toolset as well as instructions for installing and running the various tools (simulator, assembler, debugger). Even if your toolset is already pre-installed you should still read (or at least skim) this section as it describes how the project infrastructure is layed out.

2.1 Overview and Installation Instructions

The Larc toolset will compile and should run on any linux/x86 machine, although it is primarily tested on an Intel 32-bit x86 machine running Kubuntu Linux. It also may run on other *nix platforms and other architectures. It is available as a compressed tarball (larc.tar.gz) at the website: *http://math.hws.edu/larc/*.

Once you have downloaded the tarball, move it to the appropriate location. To uncompress and untar the tarball, use the following:

```
bash$ tar xvfz larc.tar.gz
```

This command will create a new directory, **larc/** (the *root directory*), in the current location. The **larc/** directory contains the following files and directories (note: directories end with '/', files do not):

Makefile **api/** **bin/** **labs/** **lib/** **manual/**

Each of these files and directories are described below (although not in order).

Makefile. The **Makefile** builds the toolset. To run it, type "make" in the root directory. Currently, only the debugger has to be installed.

api/. The **api/** contains the application programming interface for the labs that are implemented in Java. It is generated from comments in the source code using **javadoc**. It is also online at *http://math.hws.edu/larc/api*.

bin/. The **bin/** contains the auxiliary tools. In particular, it contains a functional simulator called **sim**, an assembler called **asm**, an assembly merger called **merge**, and a debugger called **db**. See the subsections below for details on how to use all four of these tools.

labs/. The **labs/** directory contains a subdirectory for each particular lab. Within each lab subdirectory (*e.g.*, **lab3/**) are the files and directories needed to complete each lab. These files and

directories are described in the sections describing the individual labs (Section 8-17).

lib/. The **lib/** directory contains files needed by the auxiliary tools. Currently, it contains a single file needed by the debugger.

manual/. The **manual/** directory contains the PDF file for this manual.

2.2 Using the Functional Simulator

The functional simulator, which is called 'sim', is located in **bin/**. It takes as a command-line argument the name of the machine code file (the format of which is described in Section 3) to simulate. For example, the following would simulate the program contained in the file **hello-world.out**, which prints "Hello, world!" to the screen, assuming the command is executed in the root directory and the machine code file is located in the root directory:

```
bash$ bin/sim hello-world.out
Hello, world!
bash$
```

The output is printed in the shell window as shown above. The program may prompt the user for (keyboard) input in which case the simulator will block until the user enters some input.

If an error occurs when the machine code file is run, then the error is printed to the screen and the simulator halts.

The machine code file need not be in the same directory as the directory where the command is executed. For example, the following would simulate **hello-world.out** given that it is in a directory called **tests/** within the root directory:

```
bash$ bin/sim tests/hello-world.out
Hello, world!
bash$
```

The user can also simulate with an operating system (OS), as will be done in the lab in Section 14, via a "-t" flag. "-t" is used since at the, very least, the OS code must contain a trap handler although other features of modern operating systems might be excluded (see Section 7 for details on trap handling). The file containing the OS code must be specified after the "-t" flag. A test program, or user program, must also be specified as before, *i.e.*, an operating system will not run by itself. For example, the following simulates the program in **hello-world.out** using the operating system in **os.out**:

```
bash$ bin/sim -t os.out hello-world.out
Hello, world!
bash$
```

See Section 7 for details on writing and using a Larc operating system.

Note: the simulator has several other flags but we can ignore them in this manual.

2.3 Using the Assembler

The assembler, which is called **asm**, is located in **bin/**. It takes as a command-line argument the name of the assembly file to assemble. For example, the following would assemble the program in the assembly file **hello-world.s**, which prints "Hello, world!" to the screen, assuming the command is executed in the root directory and the assembly file is located in the root directory:

```
bash$ bin/asm hello-world.s
bash$
```

If no output appears then this means that no errors were encountered and the assembly file was successfully assembled. The assembler creates a new file with the same base name but with extension ".out", which contains the machine code. For example, the command above would create a file called **hello-world.out**. This file could then be simulated by executing the following command:

```
bash$ bin/sim hello-world.out
Hello, world!
bash$
```

The assembly file need not be in the same directory as the directory where the command is executed. For example, the following would compile **hello-world.s** given that it is in a directory called **tests/** within the root directory:

```
bash$ bin/asm tests/hello-world.s
bash$
```

If there are no errors then the assembled machine code is placed in the same directory as the assembly file (*e.g.*, in **tests/**).

If there are errors then these are printed to the screen and the assembly file is not assembled. For each error, an error message is printed to the screen with the line number where it occurred. Students can use this information to help identify any errors. The assembler sometimes prints warnings to the screen, but for warnings, the assembly file is still assembled.

A different location for the machine code file can be specified via the "-o" flag. The machine code file must be specified after "-o". For example, the following puts the machine code for **hello-**

world.s into **hello.out**:

```
bash$ bin/asm -o hello.out hello-world.s
bash$
```

The Larc assembler does not support assembling multiple files at once although they can be merged using the **merge** tool described below.

The flag "-d" turns on debugging, which will simply print out the assembly program along with annotations (in comments) describing the addressing and linkage (*e.g.*, the address a branch refers to) of all elements in the assembly program along with some other information.

The flag "-nowarn" turns off any warning messages. For example, the assembler will print warning messages when using kernel registers.

The flag "-noext" turns off support for extended assembly instructions (assembly instructions that are translated into several machine instructions). An error will occur if this flag is used and an extended instruction is found in the assembly program.

The flag "-k" indicates operating system code is being assembled ("-k" for kernel, another name for the operating system). This suppresses some warnings and changes how some extended instructions are translated.

Note: the assembler has several other flags but we can ignore them in this manual.

2.4 Using the Merger

The merger can be used to combine several assembly files into a single assembly file. This allows for some separate compilation, at least when programming a high-level language. For the most part, you should not need to make use of the merger for the labs in this manual. However, we describe its use below in case you do find a need for it.

For example, the following combines **test1.s**, **test2.s**, and **test3.s** into a single assembly file assuming that the command is executed from the root directory and all the test programs are also located in the root directory.

```
bash$ bin/merge test1.s test2.s test3.s
bash$
```

If nothing is printed to the screen then the merge was successful. Otherwise, errors are printed. In the example above, a file called **out.s** is created in the root directory with (translated) code from all three input assembly files.

Errors will arise when two or more files cannot be merged. This usually arises because two

globals are declared with the same name or a referenced used across two or more files was not made global (see Section 5 for information on globals).

If one of the input programs is meant to act as the main module in the program you can specify this with a "-m" flag:

```
bash$ bin/merge -o test.s -m test1.s test2.s test3.s
bash$
```

In this case, **test1.s** would be treated as the main module.

The "-o" flag can be used to name the output file:

```
bash$ bin/merge -o test.s test1.s test2.s test3.s
bash$
```

In this case, the resulting file will be called **test.s**.

As with the assembler, the "-k" flag should be used when merging operating system code.

2.5 Using the Debugger

The graphical debugger, called **db**, is located in **bin/**. Using this application once it is running is described in later sections. In particular, Section 4 describes how to use this application to debug machine programs and Section 6 describes how to use it to debug assembly programs. Below, we discuss how to open the debugger and the various command-line arguments that it takes.

The debugger always takes at least one command-line argument: the name of the machine code file or assembly file to debug. For example, the following debugs the machine program **hello-world.out**:

```
bash$ bin/db hello-world.out &
bash$
```

This will open up a graphical window allowing the user to debug the program **hello-world.out**. Note: the & allows the user to continue entering in commands while the debugger remains open.

The user can also debug an assembly program. For example, the following command debugs the assembly file **hello-world.s**:

```
bash$ bin/db hello-world.s &
bash$
```

In this case, the debugger automatically assembles the assembly program, and then debugs the assembled code. However, it allows the user to step through the assembly program rather than the machine program.

As with the simulator and assembler, the debugged program file need not be in the same directory as the directory where the command is executed. For example, the following would debug **hello-world.out** given that it is in a directory called **tests/** within the root directory:

```
bash$ bin/db tests/hello-world.out &
bash$
```

The debugger does not take any additional debugger-specific options. However, it does allow the user to specify the simulator options (since it uses the simulator to execute the program) and assembler options when debugging an assembly program. To specify the simulator options, the user can use the "-s" flag. The simulator options must be provided after "-s" in double quotes. For example, the following would allow the user to debug the machine code file "hello-world.out"using the operating system in the file **os.out**:

```
bash$ bin/db -s "-t os.out" hello-world.out
bash$
```

When debugging an assembly program, the user can also set the assembler options with the flag "-a". This works similarly to "-s".

Known bugs. Unfortunately, the debugger, itself, has some bugs, which you should be aware of (the debugger is in need of some debugging!). It works well in many cases, but in other cases, errors can occur. In particular, here are a couple of known bugs in the debugger:

- Null pointer exception can occur in library graphics classes. This is probably the result of performing some graphics operations outside of the Event Dispatch Thread. It seems to happen rarely and when it does the debugger still seems to work although scrolling inside the windows can result in odd behavior. You might restart the debugger if this exception occurs (usually titled **Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException**).

- When setting values while waiting for program input, the debugger window may no longer listen to input. This should be avoided if possible. Clicking on the pause button often will allow you to enter text again (in fact, whenever problems occur, it is a good idea to try the pause button).

- Debugging with an operating system does not work properly. In particular, the highlighted instruction in the code section is sometimes wrong (*i.e.*, it is not the currently executing instruction). In addition, debugging in assembly mode is not supported while using a trap handler.

There are probably some other bugs as well. We hope to fix these errors soon and release a new version of the debugger. In practice, students have found the debugger quite useful so long as they know about the errors above.

3 ISA

This section describes the Larc instruction set architecture (ISA). Section 3.1 discusses the general characteristics of Larc such as processor width and addressing modes. Sections 3.2, 3.3, and 3.4 describe the Larc built-in data types, registers, and instructions. Section 3.5 describes trapping in Larc as well as some supported system calls. Although system calls are not technically part of the ISA (they are a part of the operating system's interface), we include a few of them as students will make use of them in many of the assignments. Finally, Section 3.6 describes Larc machine programs.

This section excludes a detailed discussion of the system architecture such as trapping and input/output (*e.g.*, reading from a keyboard, writing to a monitor). Section 7 covers this material.

This section should be read before doing the labs described in Sections 10, 11, and 12. It will also be helpful when doing the most of the other labs with the exception of the first two labs described in Sections 8 and 9.

Note that this manual and this section are not intended to teach computer architecture, but rather to discuss and describe the Larc architecture. Some background is provided as review in several places, but this manual should be supplemented with a computer architecture textbook [6, 7, 8].

3.1 General

Larc was modeled after the MIPS [3] architecture (it also has some similarities to the Alpha [2]), although it has far fewer features since it is intended for the classroom. As discussed in more detail in Section 3.4, the Larc instruction set is a subset of the MIPS instruction set. Although Larc is small, it is still a fully-functional and practical architecture. Below we discuss the basic components of a Larc machine as well as some of the general characteristics.

Basic components. Figure 1 shows the basic components of a Larc machine (this should be review for the student, but if not, then consider supplementing this material with an introductory architecture textbook [6, 7, 8]). As with almost any other computer architecture, the central processing unit (CPU), or more simply the processor, executes the program. A program consists of a set of basic instructions, which perform simple arithmetic or logic functions, move data from one location to another, or change program control. For example, one instruction might add two numbers and save the result. All complex tasks that are computable can be formulated using these basic instructions

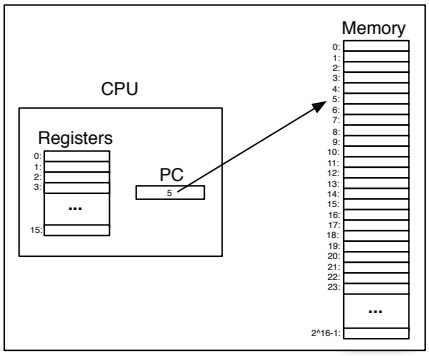


Figure 1: The basic components of a Larc machine.

(so long as the architecture is Turing complete as is Larc).

The program instructions and the program data are stored in memory, which is a sequence of addressable storage units. Most architectures also have a hard disk for storing even more data and for persistent storage (storage even when there is no power). Although, Larc does have support for a hard disk, you will not make use of it in this manual. Therefore, for us, the entire state of the program will be stored in memory.

The CPU contains two hardware structures that are a part of the instruction set architecture, *i.e.*, they are not hidden in the microarchitecture. The first, is the program counter (more aptly called an instruction pointer) or PC, which holds the address of the next instruction to execute. The second is a small set of registers (*e.g.*, 16 of them) for storing instructions and (more likely) data values. Registers have a much faster access time than memory, but because of their limited size, most instructions and data must be stored in memory.

Fetch-and-execute cycle. To execute a program, a Larc machine repeatedly fetches the next instruction from memory using the address in the PC; executes this instruction updating the registers, memory, and PC in the appropriate way; and increments the address in the PC. This process is known as the *fetch-and-execute cycle*. For example, in Figure 1 the PC currently refers to an instruction at memory address 5. This instruction will be fetched from memory, executed within the PC, and the PC will then be set to 6. On the next cycle, the instruction at memory address 6 will be executed. It is important to point out though, that programs are not necessarily executed sequentially. There are certain instructions that can modify the PC and set it to an arbitrary value. For example, the instruction at memory address 6 might set the PC to 0.

At the start of the program, the PC is set to 0, *i.e.*, the first instruction in a Larc machine resides at memory address 0. The fetch-and-execute cycle is then repeated continuously. The program stops only when it performs a halt instruction (which is actually a system call as described in Section 3.4).

RISC. Like MIPS, Larc is a *Reduced Instruction Set Computer (RISC)*. A RISC machine has a set of simple instructions. Alternatively, *Complex Instruction Set Computer (CISC)* computers have a set of more complex instructions. For example, a CISC machine might have a built-in instruction for manipulating a complex data structure such as linked list or stack.

Like other RISC machines, Larc instructions are fairly simple (*e.g.*, adding two numbers). Instructions are encoded using a fixed number of bits and there are only five types of encoding (as shown in Section 3.4). Unlike both CISC and RISC machines, Larc has a very small set of instructions; 16 in total.

Width and addressability. Larc is a 16-bit wide architecture; each register and each word in memory (*i.e.*, a single memory unit) contains 16 bits of data. For this reason, instructions are also 16 bits. Likewise, a memory address is 16 bits long, which means that the address space is 2^{16}. Because a word in memory contains two bytes (*i.e.*, 16 bits), the total size of memory in bytes is 2^{17} bytes or 128 KB.

Larc is word addressable only. Data that is shorter or longer than 16 bits (bytes, double words, *etc.*) cannot be retrieved from memory in a single operation.

Note that the width and addressability of Larc are different than in MIPS, which is 32-bit wide and byte addressable.

Endianness. Larc is a big-endian architecture. The most significant byte appears first in the word (*i.e.*, the leftmost byte). This issue generally does not arise as almost everything works at a word-size granularity. Note: this is slightly different than MIPS, which has support for both big- and little-endian encoding.

Register-register. Like most RISC machines including MIPS, Larc is a register-register architecture meaning that computations are performed on values in registers. The general format is that an operation is performed on two source registers and the result is stored in a third destination register. The source and/or destination registers can be the same.

Special load and store instructions are used to move values from memory to registers or from registers to memory, respectively. Larc does not allow computation to be performed directly on an

operand in memory. The value must first be moved to a register.

There are two instructions (load immediate and load upper immediate), which allow programmers to encode a value, called an *immediate*, directly in an instruction. The instruction allows the programmer to move the immediate into a register. The Larc ISA does not support the use of immediates in most other instructions (there are a few exceptions discussed below). Instead, the value must first be moved to a register.

Addressing modes. Larc supports only a single addressing mode for accessing data in memory: *base plus offset*. The programmer must specify a base register and an offset immediate (*i.e.*, a value encoded within the instruction). These are specified in either a load instruction, to move a value from memory to a register, or in a store instruction, to move a value from a register to memory. The value within the base register is added to the offset immediate and the result is used as the memory address.

Larc supports two addressing modes for changing program control, *i.e.*, transferring control to an instruction besides the next instruction in memory. The first addressing mode, called *register indirect*, allows the programmer to transfer control to the address stored within some register. These control transfers, called a jump, are unconditional, *i.e.*, the jump is not predicated on some condition. Jumps are often used for calling and returning from *subroutines* (analogous to methods in Java or functions in C). The second addressing mode, called *PC relative*, allows the programmer to transfer control to an offset plus the current PC. The offset is a signed immediate encoded within the instruction. These control transfers, called a branch, are conditional. Control is transferred if and only if some condition is true (*e.g.*, a value in a register is zero).

3.2 Data Types

Larc has built-in support for three data types: bitmaps, signed integers, and characters. Other data types can be simulated using these three built-in types.

Bitmaps. Larc has a built-in operation for performing the **NOR** bitwise logic function (*i.e.*, ORing two bit sequences and negating the result). This can be used to modify a 16-bit section of a bitmap, *i.e.*, a sequence of 0's and 1's. As **NOR** is logically complete, this instruction, applied several different times, can be used to perform any logical function (*e.g.*, **NOT**, **OR**, **AND**, **XOR**). In addition, Larc has built-in operations for shifting a 16-bit value left or right, logically (shift right arithmetic is not directly supported). These operations can be used to access a particular bit or

00000000	01001000	‘H’,
00000000	01100101	‘e’,
00000000	01101100	‘l’,
00000000	01101100	‘l’,
00000000	01101111	‘o’,
00000000	00101100	‘,’,
00000000	00100000	‘ ’,
00000000	01110111	‘w’,
00000000	01101111	’o’,
00000000	01110010	‘r’,
00000000	01101100	‘l’,
00000000	01100100	‘d’,
00000000	00100001	‘!’,
00000000	00001010	‘\n’
00000000	00000000	null word (string terminator)

Figure 2: “Hello, world!\n” as a Larc ASCII character string in memory.

sequence of bits within a 16-bit section of the bitmap.

Signed integers. Larc has built-in operations for adding, subtracting, multiplying, and dividing signed, 16-bit integers represented using 2’s complement. Other arithmetic operations such as modulus can be simulated using the built-in operations. Larc also has an instruction for comparing two 16-bit integers, checking if one integer is less than the other. This instruction, applied in various ways, can be used to perform any relational operation (*e.g.*, <, <=, >, >=). In addition, the shift left logical operation can be used to multiply a value by 2. The shift right logical operation can used to divide a value by 2. But since it is not an arithmetic shift right (*i.e.*, the sign bit is not shifted onto the left of the value), the value must be non-negative, or a more complex set of instructions must be used. Finally, there are two operations (*i.e.*, system calls) that allow a programmer to print an integer to the screen and read an integer typed by the user via the keyboard (technically speaking, system calls are part of the OS interface, not the ISA).

Characters. Larc has some support for working with 16-bit characters. Although 16 bits allows Larc to support Unicode encoding currently it is assumed that characters are encoded using ASCII (American Standard Code for Information Interchange). Thus, only 8 of the first 16 bits are used (the low order 8 bits). For example, a string of characters in memory can be printed to the screen. The format of the string can be seen in Figure 2, which shows “Hello, world!\n” in memory. The character string must be terminated with a null word (16 0’s).

Larc has support for printing a string in memory to the screen and reading a string from the

Register	General-Purpose/Specialized?	Function
0	specialized	always holds zero
1	general-purpose	result register
2	general-purpose	argument register
3	general-purpose	argument register
4	general-purpose	temporary register
5	general-purpose	temporary register
6	general-purpose	temporary register
7	general-purpose	saved temporary register
8	general-purpose	saved temporary register
9	general-purpose	saved temporary register
10	general-purpose	stack pointer register
11	general-purpose	return address register
12	general-purpose	temporary register (reserved for assembler)
13	general-purpose	temporary register (reserved for assembler)
14	specialized	OS (or kernel) register
15	specialized	OS (or kernel) register

Table 1: Larc registers.

user via the keyboard (technically, these are system calls, which are part of the OS interface, not the ISA). When printing the string, the programmer supplies a pointer to the string in memory and the length of the string (Section 3.5 discusses the details of this operation). The characters in the string are printed until either a null word is encountered or the supplied length is reached, whichever occurs first. Reading a string from the keyboard works similarly. The programmer supplies a pointer to memory where they want the string placed, and the maximum length of the read string. The characters read from the keyboard are placed in memory until either a newline is encountered or the supplied length is reached, whichever occurs first. A null word is placed at the end of the read string. Note: if the newline is reached, it is not placed in memory.

A common question is why null words are needed at all. For example, whenever a character string is printed or read, a length must be supplied by the programmer. The null word allows programmers to easily work with user-supplied variable-length strings. For example, the programmer could read in the name of the user (as a character string) and use 25 for the length. The user might enter fewer characters, for example, “Marc”. Because the null byte is inserted at the end of the string, when the name is printed, only the first 4 characters will be printed. Without the null byte, 25 characters would be printed (*e.g.*, “Marc” followed by 21 other characters).

3.3 Registers

Larc has sixteen registers, thirteen of which are general purpose and three of which are specialized. These registers are shown in Table 1. Many of the general purpose registers also have a dedicated use, although these are by convention only (*i.e.*, they could potentially be used in any way the programmer wishes).

Larc includes a zero register (register 0), which always contains the value 0. This register can be written, however, the value cannot be changed from 0 (in some contexts, this use of the zero register is convenient). More commonly, the zero register is used as a source input value.

Registers 1-13 are general-purpose registers. Each of these has pre-defined conventions, which are also shown in Figure 1. Programmers could choose to ignore these conventions, especially if writing simple programs that will not be used in larger programs. But if writing larger programs (like those in the assembly language assignments), these conventions should be followed.

Register 1 is used to hold the result of a subroutine, *i.e.*, the subroutine's return value. Registers 2 and 3 are used to hold the first two arguments. If a subroutine requires additional arguments, then they should be passed via memory.

Registers 4-9 are temporary registers, which have no pre-defined use. But by convention, registers 7-9 are preserved across subroutine calls. In other words, a called subroutine must save these registers to memory before using them and restore them to their original values before returning to the caller (*i.e.*, the subroutine that performed the call). Registers 4-6 are not preserved across a subroutine call. If a caller needs to preserve these values, then it should save them to memory before performing the subroutine call.

Registers 12 and 13 are also temporary registers (not saved temporaries), although these are not for general use at the assembly level. (The assembler makes use of them when translating assembly code to machine code.) But at the machine level, the programmer can use these as they see fit.

Register 10 is used to hold the stack pointer, which points to the stack in memory. The stack is a region of memory, which houses the state of each currently-executing subroutine. Because calls and returns of subroutines follow a stack pattern, a stack data structure is used to save the state of each executing subroutine (hence the name 'stack'). Larc does not provide hardware support managing the stack or the stack pointer. It must be managed explicitly by the program.

Register 11 is used to hold the return address on a call to a subroutine. This address allows

Type	Operation	Opcode	Semantics
ALU	addition	**0000**	**Reg[RA] = Reg[RB] + Reg[RC];**
	subtraction	**0001**	**Reg[RA] = Reg[RB] - Reg[RC];**
	multiplication	**0010**	**Reg[RA] = Reg[RB] * Reg[RC];**
	division	**0011**	**Reg[RA] = Reg[RB] / Reg[RC];**
	shift left logical	**0100**	**Reg[RA] = Reg[RB]** $<<$ **Reg[RC];**
	shift right logical	**0101**	**Reg[RA] = Reg[RB]** $>>>$ **Reg[RC];**
	bitwise NOR	**0110**	**Reg[RA] =** $\sim$ **(Reg[RB]** $\mid$ **Reg[RC]);**
	set on less than	**0111**	**if (Reg[RB]**$<$**Reg[RC]) Reg[RA]=1; else Reg[RA]=0;**
Long immediate	load immediate	**1000**	**Reg[RA] = sext(LIMM);**
	load upper immediate	**1001**	**Reg[RA] = LIMM** $<<$ **8;**
	branch equal to 0	**1010**	**if (Reg[RA] == 0) PC=PC+sext(LIMM);**
	branch not equal to 0	**1011**	**if (Reg[RA] != 0) PC=PC+sext(LIMM);**
Short immediate	memory load	**1100**	**Reg[RA] = Mem[Reg[RB]+sext(SIMM)];**
	memory store	**1101**	**Mem[Reg[RB]+sext(SIMM)] = Reg[RA];**
Jump	jump and link register	**1110**	**retn_addr=PC; PC=Reg[RB]; Reg[RA]=retn_addr;**
System call	System call	**1111**	perform system call (type in **Reg[1]**)

Table 2: Larc instructions. **RA**, **RB**, and **RC** are 4-bit register identifiers; **limm** refers to the long immediate; **simm** refers to the short immediate; **Reg[]** refers to the registers; **Mem[]** refers to the memory; **PC** refers to the program counter; and **sext()** is a function that sign extends an immediate to 16 bits. The operators in the semantics column are expressed in Java. Note: the CPU automatically increments the PC after fetching each instruction from memory, before executing it.

the called subroutine to link back to the call site. The caller places the address of the instruction following the call in register 11 when performing a call. When finished, the called subroutine can then transfer control to the address in register 11. As discussed in Section 3.4, Larc provides some hardware support for setting the return address in register 11.

Registers 14 and 15 are specialized registers (along with register 0, these make up all the specialized registers). These two registers can be manipulated only by the operating system (sometimes called the kernel). It is illegal for a user-level program to access them (an attempt to do so results in a trap to the operating system).

3.4 Instructions

Larc supports 16 different types of instructions, each of which are encoded in one of five ways. Because the type of instruction is encoded using the first 4 bits within the instruction, which is called the opcode, 16 is also the maximum number of distinct instructions possible (there is no room for extensions). Table 2 lists the 16 operations that are supported in Larc along with their

respective opcodes in binary and their semantics using Java syntax.

Types of operations. Larc supports eight arithmetic and/or logic operations (ALU). It supports adding, subtracting, multiplying, and dividing. It also supports logic shifts to the left and to the right as well as a bitwise **NOR**. Finally, Larc includes a relational operation, called set on less than, for computing whether one value is less than another value (1 for a true result, 0 otherwise). All eight of these operations take two registers as input (called *RB* and *RC*), and put the result in a third register (called *RA*). Note: input/output registers need not be unique. Other unsupported ALU operations (*e.g.*, modulus, bitwise AND) can be formulated using these built-in operations.

Larc does not support the use of an immediate as a source operand in an ALU operation. However, immediates can be loaded into a register in one of two ways and then used in an ALU computation. With a load immediate instruction, the register (called *RA*) is set to the result of sign extending the 8-bit immediate (called *LIMM*, short for long immediate). With a load upper immediate instruction, the register (also called *RA*) is set to the result of shifting the unsigned long immediate (also called *LIMM*) left by 8 bits.

To move values to and from memory, Larc provides two operations: load and store. The load operation allows the programmer to move a word from memory to a register (called *RA*). The store operation allows the programmer to move the value in a register (called *RA*) to a memory location. For both loads and stores, the memory location is computed by adding the value in a specified base register (called *RB*) to a signed-extended immediate (called *SIMM*, short for short immediate).

To transfer control based on some condition, programmers can use one of two conditional branches, which branches to a PC-relative target based on the value in an input register (called *RA*). The first branch, branch equal to zero, compares the value in the input register to zero. If the value in the register is zero, then the PC is incremented by a sign-extended immediate value (called *LIMM*, short for long immediate). In either case, the PC is incremented by one. The second branch, branch not equal to zero, branches if and only if the input register is not zero. Other branches are not supported (*e.g.*, branch greater than zero), but can easily be formulated in terms of the two supported branches with the aid of ALU instructions such as the set on less than instruction.

To jump to an absolute target (*i.e.*, not PC-relative), programmers can use an indirect jump-and-link-register. The jump-and-link-register operation transfers control to the value (an address) within an input register (called *RB*). Before the transfer occurs, the address of the instruction following the jump-and-link-register (current PC plus one) is stored in an output register (called *RA*).

This is useful for implementing calls and returns of subroutines (it is where the term 'link' in jump-and-link-register comes from). The called subroutine can use the value in the output register as the target, when it is finished.

The final operation is a system call. To perform system-related functions such as printing to the screen, getting input from the keyboard, *etc.*, user-level programs must perform a system call. The system call triggers a hardware trap to operating system code (somewhere within the address space) that performs the necessary operation. In many assignments, although unrealistic, we will assume system calls are built-in to the CPU and not implemented in the OS. This simplification allows us to delay looking at the OS.

The system call has no explicit operands, however, the type of the system call (*e.g.*, printing a string to the screen) must be passed in register 1. Depending on the particular type of system call, other arguments may be passed via registers 2-3.

While the system call instruction is part of the Larc ISA, the supported system calls are not. Instead, they are part of the interface to the operating system. However, in this manual, we assume several supported system calls. Table 3 lists these system calls, along with their arguments and return values. In Section 3.5, we discuss these system calls in more detail.

Encoding. As shown in Table 2, there are five classes of instructions. Each class has its own type of encoding, which is shown in Figure 3. Each type of encoding consists of several fields, which contain either an opcode, register identifier (RA, RB, or RC), or immediate (LIMM or SIMM). Table 2 lists the operations with their respective binary opcodes. Register identifiers are encoded into an instruction as a 4-bit, unsigned binary number. For example, register 12 would be encoded as 1100. Immediates are encoded as either 8-bit (LIMM – long immediate) or 4-bit (SIMM – short immediate) values using 2's complement. All immediates are signed except in the load upper immediate. For example, -24 in a branch's 8-bit, long immediate would be encoded as 1111111111101000, while 7 in memory load's 4-bit, short immediate would be encoded as 0111.

The first type of encoding is for arithmetic and logic instructions (ALU) as shown in Figure 3(a). These are encoded as follows:

- Opcode. The first 4 bits (4 leftmost bits) are dedicated to the opcode.
- RA. The next 4 bits are dedicated to the RA register, which is the target register.
- RB. The next 4 bits are dedicated to the RB register, which is the first source register.
- RC. The final 4 bits are dedicated to the RC register, which is the second source register.

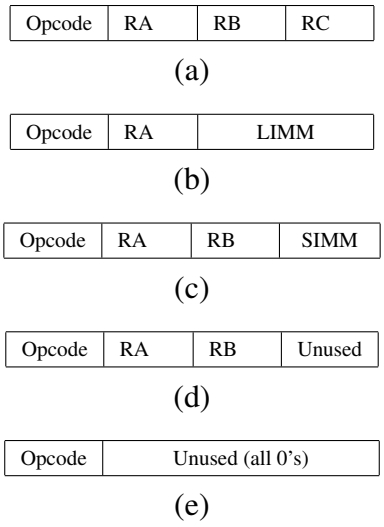


Figure 3: The encoding of Larc instructions: (a) ALU, (b) long immediate, (c) short immediate, (d) jumps, and (e) system calls.

The second type of encoding (Figure 3(b)) is for long immediate instructions, which include conditional branches and load immediates. These are encoded as follows:

- Opcode. The first 4 bits (4 leftmost bits) are dedicated to the opcode.
- RA. The next 4 bits are dedicated to the RA register. For conditional branches this register is the source, comparison register. For load immediates this register is the target.
- Long immediate. The next 8 bits are dedicated to the long immediate.

The third type of encoding (Figure 3(c)) is for short immediate instructions, which include loads and stores of memory. These are encoded as follows:

- Opcode. The first 4 bits (4 leftmost bits) are dedicated to the opcode.
- RA. The next 4 bits are dedicated to the RA register. For a load, this register is the destination, and for a store, it is the source.
- RB. The next 4 bits are dedicated to the RB register, which contains the base address.
- Short immediate. The next 4 bits are dedicated to the short immediate, offset.

The fourth type of encoding (Figure 3(d)) is for jumps. These are encoded as follows (note: the rightmost 4 bits are unused):

- Opcode. The first 4 bits (4 leftmost bits) are dedicated to the opcode.
- RA. The next 4 bits are dedicated to the RA register. The return address (for linking) is written into this register.
- RB. The next 4 bits are dedicated to the RB register, which contains the target address.

The final type of encoding (Figure 3(e)) is for system calls. These are encoded as follows (note: only the opcode is used in a system call):

- Opcode. The first 4 bits (4 leftmost bits) are dedicated to the opcode. The remaining bits must all be 0.

3.5 Traps and System Calls

Nearly all machines, Larc included, support some form of trapping. A trap is a special control transfer into a part of the operating system, which can then respond to the event in some way. A trap can occur due to a program error such as a divide by zero or the use of an operating system register (registers 14 and 15) by a non-OS program. A trap also occurs when a program executes a system call instruction. A system call is essentially a call to a subroutine implemented within the operating system.

On a trap, the processor places the ID of the particular trap that occurred (*e.g.*, divide by zero error) in a well-known place in memory and transfers control to the trap handler component of the operating system, which resides at a well-known place in the address space. The trap handler, which is software, then responds to the particular trap that occurred. In the case of an error, it would probably terminate the program that triggered the trap. In the case of a system call, it would carry out the requested operation on behalf of the program.

We will ignore the details of trapping until Section 7, which focus on the Larc system architecture. Until that point, if an error occurs, then the machine (or simulator) will immediately halt with an appropriate error message. Moreover, system calls will execute atomically. In other words, when a system call executes, the machine (or simulator) will carry out the operation without the aid of operating system software and then proceed to the next instruction.

Although we will ignore the details of how traps and system calls work until Section 7, nearly all of the assignments will make use of system calls. So the student will need to know how functionally system calls work. Table 3 shows the (base) Larc system calls. These operations correspond to writing and reading data to and from the display and keyboard, respectively. In addition, one system call performs a halt to shutdown the computer. The student will implement these system calls when building a Larc simulator and I/O handler.

System call 0, *i.e.*, when register 1 contains a 0, halts the computer. There are no parameters and this system call does not return (since the computer shuts down).

Identifier	Operation	Arguments/Return Values
0	Halt the system	no arguments, doesn't return
1	Print string	argument 1: string pointer in **Reg[2]**, argument 2: string length in **Reg[3]**
2	Print int	argument 1: integer in **Reg[2]**
3	Read string	argument 1: string pointer in **Reg[2]**, argument 2: string length in **Reg[3]**
4	Read int	returns int in **Reg[1]**

Table 3: Larc system calls and their semantics.

System call 1, *i.e.*, when register 1 contains a 1, prints a string to the screen. The pointer of the string in memory is taken from register 2 and the length, in number of characters, is taken from register 3. A final newline character is not automatically printed. The programmer must specify within the string all newline characters that they wish to print.

System call 2 prints an integer to the screen. The integer is taken from register 2.

System call 3 reads a string from the user via the keyboard. The arguments are similar as when printing a string. The pointer of the string in memory is passed via register 2 and the length (in number of characters) is passed via register 3. Note: the read string may actually be shorter than the value in register 3. The length specifies the upper bound of the size of the string. For example, if the value in register 3 is 10, then the user could enter a string of length 8. But if the user attempts to enter a string of length 11, then the last character is ignored. Note also: the final newline character is not copied ot the string in memory.

System call 4 reads an integer from the user via the keyboard. The resulting integer is placed in register 1. If the user enters a non-decimal value, then 0 is placed in register 1, even if part of the value is numeric. For example, if the user enters "23a", then register 1 is set to 0 not 23.

It should be pointed out that these are only a few of the system calls for a Larc operating system. These are called the *base* system calls. There are others for doing things such as manipulating files and directories. But in this manual, we will not make use of these system calls so they can be ignored.

3.6 Larc Programs

This section describes the structure and memory layout of of a Larc machine program.

Program file. As with any architecture, a Larc (machine) program file is made up of each program instruction and data word encoded in a binary format. In a commercial architecture, the (machine)

program file itself is usually encoded as a binary, executable file. However, in Larc, a program file is encoded in ASCII, which makes it easier to write and manipulate by hand. For example, students can use a simple text editor (*e.g.*, emacs) to write a Larc program. Unlike commercial executable files, Larc machine files can be annotated with comments. We will defer discussion on annotations until the next section, which looks at programming in Larc machine language.

In the absence of annotations, each line in the ASCII file contains either an instruction or a data word. These lines must be made up of exactly 16 '0' and '1' ASCII characters or 4 hexadecimal (0-9, A-F) characters preceeded by "0x" (these two formats can even be mixed within a single machine file).

As an example, one line might contain an instruction for adding the value in register 2 to the value in register 3 and putting the result in register 1. In this case, the corresponding line in the file, if using the binary format, would look as follows:

0000000100100011

The first 4 bits encode the addition opcode (**0000**), the next 4 bits encode the target register identifier (**0001**), the next 4 bits encode the first source register identifier (**0010**), and the final 4 bits encode the second source register identifier (**0011**).

Another line might contain a data word, which encodes the value -10,000 in 2's complement. In this case, the corresponding line in the file would look as follows:

1101100011110000

The first line of the program file is assumed to be the initial instruction and is placed at address 0 in memory. Subsequent lines contain other instructions and/or data words. The next line in the program file (containing a word) is placed at address 1, the next at address 2, and so on.

Memory layout. In general, a Larc program in memory has the structure shown in Figure 4. In general, there are six sections in memory, though some of these may not be used by all programs. The first section in memory contains the instructions. This section starts at address 0. The second section in memory is the static data section (data encoded in the program file). If the program contains **n** instructions, then the static data section starts at address **n**. If there are **m** data words encoded in the program, then the static data section would end at address **n+m**. This is followed by the dynamically-allocated data section (data not encoded in the program file, but rather created during program allocation). Unlike the first two sections, this section can grow and shrink as the program executes. It grows towards the high addresses in memory (bottom part of the figure). This

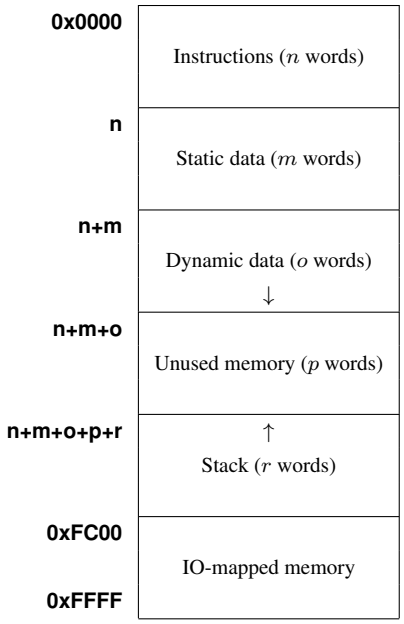


Figure 4: The general memory layout of a Larc program.

section is followed by an area of unused memory. After that, resides the stack (which houses the state of every called subroutine). Like the dynamically-allocated section, the stack section grows and shrinks as the program executes. However, it grows towards the low addresses in memory (top part of the figure). Note: the dynamically-allocated memory and stack could potentially grow into one another, which would most likely lead to a program crash. It is up to the programmer to catch this error. The final section in memory is the I/O memory-mapped section. This section cannot be used by the program except to communicate with I/O devices, which we will look at in Section 7.

This is just a general template for how Larc programs are laid out in memory, but other layouts are possible. For example, a just-in-time compiler might create instructions in the dynamically-allocated area that are later executed. Furthermore, some programs could mix instructions and static data, although this would not be recommended. But, in general, Larc programs will follow these conventions.

Example program. Figure 5 shows a trivial program that prints "Hello, world!" to the screen. It consists of 6 instructions for executing the program, 15 data words that hold the "Hello, world!\n" string (note: the string contains 14 characters plus a null terminating word). It does not require any dynamically-allocated data. Figure 5(a) shows the raw program, while Figure 5(b) describes each individual word within the program.

```
1000000100000001
1000001000000101
1000001100001110
1111000000000000
1000000100000000
1111000000000000
0000000001001000
0000000001100101
0000000001101100
0000000001101100
0000000001101111
0000000000101100
0000000000100000
0000000001110111
0000000001101111
0000000001110010
0000000001101100
0000000001100100
0000000000100001
0000000000001010
0000000000000000
```

(a)

Program instructions				
Addr.	Instruction			Description
0	1000 (ld. imm.)	0001 (reg. 1)	00000001 (1)	put system call # in reg. 1
1	1000 (ld. imm.)	0010 (reg. 2)	00000110 (6)	put string addr. (6) in reg. 2
2	1000 (ld. imm.)	0011 (reg. 3)	00001110 (14)	put length (14) in reg. 3
3	1111 (sys. call)	000000000000 (unused)		write string to screen
4	1000 (ld. imm.)	0001 (reg. 1)	00000000 (0)	put system call # in reg. 1
5	1111 (sys. call)	000000000000 (unused)		halt computer
Program data				
Addr.	Data			Description
6	00000000		01001000 (‘H’)	start of string: ‘H’ (72)
7	00000000		01100101 (‘e’)	string continued: ‘e’ (101)
8	00000000		01101100 (‘l’)	string continued: ‘l’ (108)
9	00000000		01101100 (‘l’)	string continued: ‘l’ (108)
10	00000000		01101111 (‘o’)	string continued: ‘o’ (111)
11	00000000		00101100 (‘,’)	string continued: ‘,’ (44)
12	00000000		00100000 (‘ ’)	string continued: ‘ ’ (32)
13	00000000		01110111 (‘w’)	string continued: ‘w’ (119)
14	00000000		01101111 (‘o’)	string continued: ‘o’ (111)
15	00000000		01110010 (‘r’)	string continued: ‘r’ (114)
16	00000000		01101100 (‘l’)	string continued: ‘l’ (108)
17	00000000		01100100 (‘d’)	string continued: ‘d’ (100)
18	00000000		00100001 (‘!’)	string continued: ‘!’ (33)
19	00000000		00001010 (newline)	string continued: ‘\n’ (10)
20	0000000000000000 (null word)			string end: null terminator

(b)

Figure 5: A simple program that prints “Hello, World!\n” to the screen. (a) shows the raw program and (b) shows the structure of the program along with a description of each line.

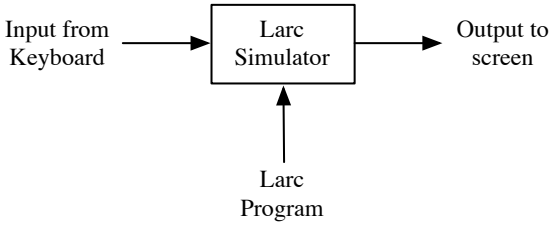


Figure 6: A Larc simulator.

```
bash$ ./sim hello-world.out
Hello, world!
bash$
```

Figure 7: Running the "Hello, world!" program using the Larc simulator.

Running programs. As Larc is classroom architecture, there are no existing Larc machines. To run a Larc machine program we use a simulator (which the student will implement in a lab assignment). As shown in Figure 6, the simulator takes as input a Larc machine program (as defined in the previous section) and behaviorally simulates a Larc processor. In other words, it simulates the input/output behavior (*i.e.*, keyboard events or monitor events) exhibited from running the program on a real Larc machine. Given some keyboard input, it will display the monitor output that a real Larc machine would display. This type of simulator is known as a functional simulator as it simulates only the functionality of a Larc machine and not the timing, *i.e.*, the time it would take a Larc machine to run a particular program given some particular input.

A Larc simulator precisely follows the specification of the Larc ISA described in this section. For example, it correctly executes the instructions defined Table 2 and Figure 3. It models the registers defined in Table 1. It lays memory out as described in Section 3.6. It supports the system calls defined in Table 3.

In Figure 7, the simulator is used to run the "Hello, world!" program from Figure 5. The simulator is run within the shell and the monitor output is also displayed within the shell. In a real Larc machine, the output below the shell prompt would be all that appears on the screen. Note: "sim" is a shell script for running the Java simulator, which resides in the current directory along with the program file. The shell script takes the simulator arguments as arguments (see Section 2 for details). In this case, only the program is specified, "hello-world.out" (which must end in ".out").

4 Machine Programming

This section describes how to write, run, and debug Larc machine programs. It should be read before doing the lab described in Section 11. It may also be helpful for later labs. It need not be read before doing earlier labs.

4.1 Writing Programs

As described in the previous section, a Larc program consists of several 16-bit words where each word corresponds to an instruction or data value. In general, each line in a Larc machine program file contains the next word, often written in binary format (16 0's and 1's). Unlike commercial systems, the programs files are encoded in ASCII rather than binary making them easier to work with in standard text editors (*e.g.*, emacs, vi). Section 3.6 described the structure and layout of Larc programs. Here we describe some strategies for Larc machine code programming.

Obviously, it is hard to read and write programs in machine language, which is why most programmers use higher-level languages (as we will in later sections). Extra care must be taken when programming at the machine code level. There are a couple of features of Larc machine program files, however, that make machine programming easier.

Hexadecimal format. One nice feature of Larc machine program files is that they can be written in a binary format or hexadecimal format. To write a word in hexadecimal, you would write 4 hexadecimal digits (0-9, A-F, a-f), which correponds to 16 bits (the word size), preceeded by "0x". In fact, one can even mix the hexadecimal and binary formats within the same Larc machine program.

It is often much easier to write instructions and data values in hexadecimal rather than binary. For instance, as was discussed in the last section, the width of a field in an instruction is always a multiple of 4 (*i.e.*, 4, 8, or 12). This makes encoding them into a hexadecimal very easy since each hexadecimal digit corresponds to 4 bits.

As an example, imagine we need to write an instruction for NORing register 4 and register 5 and putting the result in register 8. The instruction in binary would look as follows:

```
0110100001000101
```

The first four bits correspond to the NOR opcode (6), the second four bits correspond to the target register identifier (8), the third four bits correspond to the first source register identifier (4),

```
# instructions for printing string “Hello, world!\n”
0x8101
0x8206
0x830E
0xF000
# instructions for halting machine
0x8100
0xF000

# data string ”Hello, world!\n”
0x0048
0x0065
0x006C
0x006C
0x006F
0x002C
0x0020
0x0077
0x006F
0x0072
0x006C
0x0064
0x0021
0x000A
0x0000
```

Figure 8: Annotated version of the “Hello, world!” Larc machine program.

and the last four bits correspond to the second source register identifier (5). Now let’s look at the same instruction in hexadecimal:

```
0x6845
```

In hexadecimal, each digit corresponds to a different field in the instruction. Furthermore, each field is much easier to write and read when encoded as a hexadecimal digit.

Comments. Another nice feature of Larc machine program files, unlike commercial machine program files, is that they can be commented. Comments are made using the ‘#’ character. Any text following ‘#’, including the ‘#’ character itself, is ignored. Unlike in higher-level languages the comment must use up the entire line; the ‘#’ must be the first character on the line. In addition, empty lines can be included to improve readability.

Figure 8 shows a commented version of the “Hello, world!” Larc machine program using the hexadecimal format. Each line is clearly described and the program is now much easier to read. In general, comments should always be used when programming at the Larc machine code level.

Syntax summary. To summarize the syntax of a Larc machine program, each line in the program must contain one of the following:

- A comment that starts with '#' with nothing left of '#' in the line.
- A word in binary consisting of exactly 16 0's and 1's.
- A word in hexadecimal consisting of exactly 4 hexadecimal characters (0-9, A-F, a-f) preceeded by "0x".
- An empty line.

If a Larc machine program contains a line that does not correspond to one of these four formats then the program has a syntactic error and will not run.

4.2 Running Programs

As described in Section 3, a functional simulator is used to run Larc programs. The simulator will simulate the input/output behavior, such as keyboard and monitor events, that occur when running the program on a real machine. It will not simulate the timing of the machine, *i.e.*, how long it takes to execute a program.

Although you will write a simulator when doing the lab in Section 10, a reference simulator is provided for completing the other labs. Unlike your simulator, which is written in Java, the reference simulator is written in C and compiled to run natively on a Linux machine. Aggressive optimization is enabled during compilation of the simulator. As a result it executes most programs fairly quickly (like those that you will be simulating in this manual). Still, simulation takes orders of magnitude longer than running programs natively, so some long-running programs may take a while to complete.

The Larc simulator will also help in catching and finding some errors that may occur in a Larc machine program. It will catch and identify all syntactic bugs. For example, if a line of the program file is not properly formatted (*i.e.*, is not a blank line, comment line, word in binary consisting of 16 0's and 1's, or a word in hexadecimal consisting of 4 hexadecimal digits preceeded by "0x"), the simulator will print out an error message. The simulator also catches and identifies some semantic bugs. For example, if the Larc program attempts to divide by zero, this bug will be caught and reported by the simulator. (When the student implements their own simulator, they should try to catch and identify some of these bugs as well.) Still, for more subtle bugs, the simulator is less helpful. A debugger is provided for these cases.

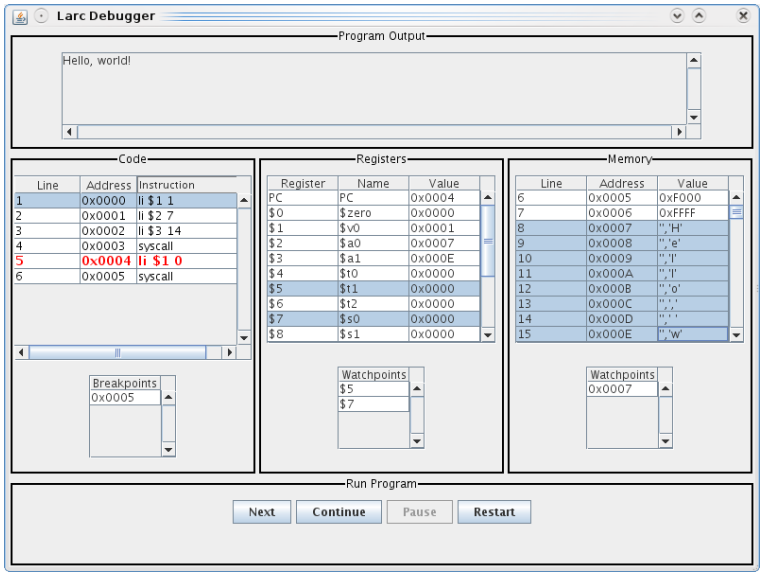


Figure 9: A screenshot of the Larc debugger running the "Hello, world!" program.

4.3 Debugging Programs

The Larc toolset includes a debugger for identifying and fixing errors within a Larc machine program (it can also be used for assembly programs as discussed in Section 6). The Larc debugger effectively allows the student to run a Larc machine program, an instruction at a time and watch how the contents of the PC, registers, and memory change. The debugger is a graphical application, written in Java. It is slower than the Java simulator, so it should probably only be used when testing. Note: the Larc debugger is modeled in part on the MipsPilot debugger [10].

To run the debugger on the "Hello, world!" program, the student would type "./db hello-world.out" into the shell assuming both the debugger and "Hello, world!" program are located in the current directory. This command opens a new window, which will allow the student to run the program and watch the machine state change as the program is run. The Larc debugger can only run a single program at a time. To run a different program, the student would need to rerun the debugger using a different program name.

Figure 9 shows a screenshot of the debugger. The debugger window consists of five panels. The top panel displays the program input and output as it would appear in the shell. The middle three panels display the code (instructions), registers, and memory. The bottom panel is a panel for running the program. It contains four buttons for running or stopping the program. Below the buttons is also an area where information and error messages are displayed.

Running the program. The program is not initially run when the debugger is first started. To run the program, the user can push either the "next" or "continue" buttons. The "next" button will execute the next instruction (the first one), stop, and wait for further input from the user. The "continue" button will execute the entire program. However, it can be stopped by pushing the the "pause" button. This will stop the program at the current instruction and allow the user to inspect the machine state. The program can also be paused without the button if the program requests keyboard input via a system call. In this case, the program is paused and a message is displayed on the button of the run panel to alert the user that they must provide keyboard input. If the user enters some input and does not click the "pause" button, then the program will continue running until the "pause" button is entered, keyboard input is requested again, or the program halts. A fourth button, "restart", allows the programmer to restart the program from the beginning.

Inspecting the state of the machine. Whenever the program has been paused, either with the "pause" button, after an instruction is executed using the "next" button, or while the debugger is waiting for keyboard input, the user can inspect the state of the machine. The state is shown in the upper table of the three middle panels (the one *not* labeled "Breakpoints" or "Watchpoints"). One panel shows the instructions that are being executed, one panel shows the current register values, and one panel shows the current memory values. For now, ignore the lower table in each of these panels.

Inspecting the instructions. On the far left, the code panel (labeled "Code") shows the instructions in the program. It highlights the current instruction. For each instruction, the debugger displays the line number of the instruction in the program file, the address of the instruction in memory, and a textual representation of the instruction.

To represent instructions in a textual form, we use assembly language, which will be discussed in more detail in the next section. We give only the pertinent details here. For each instruction, the assembly representation contains a list of the instruction fields separated by spaces. The opcode field is represented textually. Figure 4 shows the textual representation of each opcode field. Register identifier and immediate fields are represented in decimal. Register identifiers are preceeded by '$', immediates are not. Unused fields are not listed (*e.g.*, the operand fields in a system call instruction).

With only two exceptions the fields are ordered as they appear in the machine instruction. For example, a subtraction instruction that sets register 1 to the result of subtracting register 2 by

Operation	Opcode	Assembly Name
addition	**0000**	**add**
subtraction	**0001**	**sub**
multiplication	**0010**	**mul**
division	**0011**	**div**
shift left logical	**0100**	**sll**
shift right logical	**0101**	**srl**
bitwise NOR	**0110**	**nor**
set on less than	**0111**	**slt**
load immediate	**1000**	**li**
load upper immediate	**1001**	**lui**
branch equal to 0	**1010**	**beqz**
branch not equal to 0	**1011**	**bnez**
memory load	**1100**	**lw**
memory store	**1101**	**sw**
jump and link register	**1110**	**jalr**
system call	**1111**	**syscall**

Table 4: Larc assembly instruction names.

register 3 would look as follows in assembly:

sub $1 $2 $3

The two exceptions are the memory load and store. The format of these two instructions is:

lw $a simm($b)

sw $a simm($b)

where **a** and **b** are the RA and RB register identifiers, respectively, and **simm** is the short immediate. The following is an example load instruction:

lw $2 -4($5)

This instruction sets register 2 to the value in memory at the address resulting from adding register 5 to -4.

A careful reader might be wondering how the debugger knows where the instructions reside within the program file. The debugger knows the first word in the program is the first instruction. Moreover, it assumes that the instructions and data are not interleaved, *i.e.*, all of the instructions are at the beginning of the program. It looks for a special marker to determine the end of the instructions. The marker is all 1's, which is not a valid instruction. (This marker pushes the data string down one word, which is why it starts at address **0x0007** rather than **0x0006**.) If no marker is found then each word in the program file will be interpreted as an instruction, even if it is actually a data value. It is a good idea to always insert this marker in any Larc machine program you write

in case you need to debug the program later on.

Inspecting the memory. The far right panel (labeled "Memory"), shows the contents of memory. These include instructions as well as data values. For each word in memory, the debugger shows the line from the program file corresponding to that memory word ("N/A" if there is no such line), the address of the word, and the value of the word.

Each memory value in the panel is initially shown in hexadecimal although it can also be viewed as a signed decimal int, an unsigned decimal int, an ASCII character, or an instruction. To change the value format, highlight via the mouse the entries that you wish to change and then right click with the cursor still inside the memory panel. (The user can use the control and shift buttons to select more than one memory value at a time.) Click on"Format of values", and select the appropriate format.

Memory values can also be modified by clicking inside the value field for a particular memory word and changing it. The current format of the value must be followed when modifying the value. For example, if the memory word at address 2 is "0x0100", *i.e.*, the format is hexadecimal, then it cannot be set to "5". It can be set to "0xA2B1", however. Modifying the value will impact the behavior of the program if that value is used in the future, so use care when modifying values. But this feature can be helpful, to see if modifying a particular value, does in fact eliminate the error. The line or address fields cannot be modified.

Notice that the instructions in the code panel can not be modified nor can their format be changed. However, the memory panel can be used to do either by accessing the instruction at its address.

Inspecting the registers. The middle panel (labeled "Registers") shows the contents of registers. The PC is also included in the list of registers. For each register, the identifier (*e.g.*, "$0"), name (*e.g.*, "$zero"), and value is shown (register names are discussed in Section 5, they can be ignored for now). Like a memory value, the value of a register can be displayed as a hexadecimal, signed decimal, unsigned decimal, an ASCII character, or an instruction. Also like a memory value, a register value can be modified by clicking within the value field and modifying the contents, following the current format.

Breakpoints and watchpoints. It is often helpful to be able to pause the program whenever a particular instruction is executed. For example, the user might want to pause the debugger every time the first instruction within some loop is executed. This feature is called a breakpoint and is

supported in the Larc debugger. The lower table within the code panel lists memory addresses of the current breakpoints. To add a breakpoint, the user can right click within the code panel, select "Add/remove breakpoints", and then select "Add selected breakpoints". A breakpoint is added for each instruction that has been selected by the mouse. (The user can use the control and shift buttons to select more than one instruction at a time.) To remove breakpoints, the user can right click within the code panel, select "Add/remove breakpoints", and then select either "Remove selected breakpoints" or "Clear breakpoints". "Remove selected breakpoints" removes each breakpoint that is selected within the breakpoint table. "Clear breakpoints" removes all breakpoints within the breakpoint table.

Another helpful feature, which the Larc debugger provides, is watchpoints. These also pause the program when a particular event occurs. However, watchpoints stop the program when a particular data value is written rather than when an instruction is executed. The value must be changed in order for the watchpoint to trigger. The user can watch values in either a register or in memory. For example, the user could set a watchpoint to stop the program whenever register 5 is changed. Often, in programming at the machine language level, programmers will find that a particular region of memory is unintentionally getting corrupted by the program. Watchpoints allow the programmer to find the instruction or instructions that are causing this corruption. Adding and removing watchpoints works similarly to adding and removing breakpoints.

5 Assembly Language

This section describes the Larc assembly language. Section 5.1 describes the general characteristics of the Larc assembly language, contrasting it with the Larc machine language. Section 5.2 describes the assembler, a tool for converting an assembly language program into a machine language program. Sections 5.4 and 5.5 describe Larc assembly instructions and data, respectively. Section 5.6 describes how to write Larc assembly programs. Finally, Section 5.7 enumerates the errors that can occur in Larc assembly programs.

This section should be read before doing the lab described in Section 12 as well as later labs. It need not be read before doing earlier labs.

5.1 General

Larc assembly language (like other assembly languages) is basically a text-based form of Larc machine language. The semantics of the Larc assembly language is very similar to the semantics of the Larc machine language (although not identical), but the syntax is much different. Instructions and data are written in a text format as opposed to a binary format. For example, if the programmer wishes to put add an instruction that loads immediate 3 into register 1, they can write it as

li $1 3

rather than

1000000100000011 or **0x8103**

The same is true for data used within an assembly program. For example, the integer -5 can be put in an assembly program by writing

.word -5

as opposed to inserting

1111111111111011 or **0xFFFB**.

Clearly, assembly programs are much easier to read and write.

In addition to the syntax of the language, there are other important differences between Larc assembly language and Larc machine language. First, unlike in machine language, in assembly language the programmer can also use *labels*, which are essentially names for memory addresses. For example, the programmer could define the label "loop_start" to refer to the address at start of some section of code implementing a loop. Branches and load immediates (a special form of a load

immediate called a load address) can then refer to these labels rather than to a numeric immediate. The advantage of this approach is that changes to the assembly code (*e.g.*, inserting instructions) do not impact the branch or label, unlike when a numeric value is used. Basically, labels represent a level of indirection, which is resolved (by the assembler) when the assembly program is translated into a machine program. Another advantage is that with branches, the programmer does not have to think in terms of a PC-relative addressing mode, but rather can specify an absolute target (*e.g.*, the label). The label is converted (by the assembler) into a PC-relative, 8-bit immediate in the translated machine code.

Larc assembly language also has built-in support for handling multi-word data types such as arrays and strings. For example with strings, at the machine code level, the programmer encodes a string in a program by inserting the ASCII encoding of each character into the program, combining adjacent characters into a single word. In addition, the machine code programmer must remember to terminate a string with a null word. For example, here is the string "foo" at the machine code level (in hexadecimal format):

```
0x0066
0x006F
0x006F
0x0000
```

On the other hand, an assembly language programmer can simply write:

```
.asciiz "foo"
```

Another difference between Larc assembly language and Larc machine language is that the assembly language includes some *extended instructions*. Although most assembly instructions map directly to a single machine instruction, this is not the case for all assembly instructions. For example, the Larc assembly language includes extended instructions for computing bitwise AND and bitwise OR (among other things), which are not machine instructions. These assembly instructions map to several machine instructions.

A final difference between assembly language and machine language, in most architectures, is that the programmer can use commenting and spacing to make their program more readable. Larc, however, does support machine code commenting and empty lines (since it is encoded in ASCII). But even in Larc machine programs, comments cannot be placed in arbitrary places (*i.e.*, they must use the entire line) nor can arbitrary spacing be used.

As in Larc machine programs, a (single-line) comment in Larc assembly language starts with '#' although it can occur in the middle of a line. The '#' is ignored as well as all subsequent

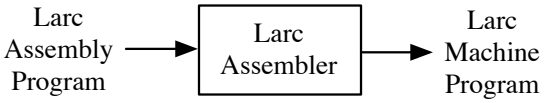


Figure 10: A Larc assembler.

characters until a newline is encountered. As with programming at the machine code level, it is critical to comment assembly programs in order to make them readable by others.

5.2 Assembler

A Larc machine understands Larc machine code only. It does not understand Larc assembly code. Therefore, in order to run a Larc assembly program, it must first be translated into a machine program using a translation program called an assembler (note: this is typical of almost all architectures). Figure 10 depicts the Larc assembler translation process. The assembler reads in a Larc assembly program and produces a Larc machine program. The generated Larc machine program can then be run on a Larc machine or simulator.

Assembly process. The Larc assembler, like assemblers on other architectures, must perform multiple passes over the Larc assembly program. In the first pass, the assembler must determine the addresses of all instructions, data words, and labels, and build a symbol table for labels so that the addresses of labels can be looked up in a subsequent pass. In the second pass, the assembler must patch all references to labels with the correct address. It uses the symbol table from the first pass to find the address of each label. Finally, in a third pass, which could be merged with the second pass, the assembler converts each instruction or data word into the corresponding 16-bit word. Multi-word data types must be translated into several 16-bit words.

Extended instructions. If the assembler supports extended assembly instructions (instructions that map to more than one Larc machine instruction – discussed in Section 5.4) than the assembler will also need to convert each extended instruction into several semantically equivalent base instructions. It might make it easier to expand these instructions into base instructions before doing the other passes described above.

Error handling. The assembler must also find any errors in the Larc assembly program. For example, one error the assembler should look for is an improperly formatted instruction (*e.g.*, a load immediate without any operands). Section 5.7 lists all the possible errors that can occur in a

Larc assembly program. When the assembler discovers an error, it should print a meaningful error message and exit without assembling. Whenever possible, the assembler should try to discover as many errors as it can before exiting. This allows the assembly programmer to fix several errors at a time.

5.3 Labels

Labels are an important feature of any assembly language. As mentioned above, they allow the programmer to name parts of memory. Instructions and data elements can make use of labels as opposed to a numeric address. This is clearly more readable. But more importantly, memory references used in instructions and data elements will remain correct when code is inserted or removed, making application development much easier.

A label name is any sequence of upper or lowercase alphabetical letters, digits, or underscore ("_"). The number sign ("#") or at sign ("@") can also be used in a label name, however, these should generally be avoided. The assembler often needs to create its own labels (*e.g.*, when translating extended instructions) and makes use of these characters to avoid duplicating an existing label.

To define a label, the programmer inserts a legal label name followed by ":" at the appropriate place in the program. For example, the programmer might write:

```
foo:
li $1 3
```

The label **foo** will refer to the address of the first instruction or data element following it. In this case, it will refer to the address of the load immediate instruction (*i.e.*, where the instruction will be placed in memory).

The label can then be used in certain instructions and data elements (as discussed below) by referring to the name without the colon. For example, we could write a branch that branches to the address of **foo** when its condition is true. It is the job of the assembler to determine what the address of **foo** will be and to determine the correct 8-bit PC-relative immediate to use in the branch machine instruction.

Labels can also be used for data elements. The following label refers to the address of the "foo" string. We can then write instructions (*e.g.*, for printing the string), which put the address of the string in a register.

Register	Name	Function
$0	$zero	always holds zero
$1	$v0	result register
$2	$a0	argument register
$3	$a1	argument register
$4	$t0	temporary register
$5	$t1	temporary register
$6	$t2	temporary register
$7	$s0	saved temporary register
$8	$s1	saved temporary register
$9	$s2	saved temporary register
$10	$sp	stack pointer register
$11	$ra	return address register
$12	$at0	assembler register
$13	$at1	assembler register
$14	$k0	OS (or kernel) register
$15	$k1	OS (or kernel) register

Table 5: Larc assembly registers.

```
foo:
.asciiz "foo"
```

Multiple labels can also be defined consecutively. For example, we could write:

```
foo:
goo:
li $1 3
```

In this case, labels **foo** and **goo** would refer to the same memory address, *i.e.*, the address of the load immediate instruction.

We can make use of the same label more than once (*e.g.*, in multiple branch instructions). Each label definition must be unique, however. For example, defining the label **foo** twice is an error.

5.4 Instructions

The components of a Larc assembly instruction (operator and operands) are split up using spaces or tabs as in high-level programming languages like Java. The operator (*e.g.*, "li") is the first component of the instruction. The operands follow the operator. For example, a load immediate of 3 into register 1 would be written as "li $1 3". The "$1" represents register 1 and "3" represents the immediate 3.

Registers. Registers in assembly language are written using the format: $<id> where <id> is

replaced with an identifier from 0 to 15. Registers can also be written using a pneumonic name. For example, the stack pointer register, register 10, can be written as $sp or $10. Table 5 lists the 16 registers, showing both their use in assembly language (which is generally the same as in machine language, with a few exceptions) and their pneumonic name.

There is one important difference between the use of registers in assembly language versus machine language. Registers 12 and 13 ($at0 and $at1) are reserved by the assembler. The assembler uses these registers to generate assembly code. In general, they should not be used by the programmer (the assembler will give a warning, if the programmer does).

Immediates. Immediates are written in decimal or hexadecimal in assembly language. They cannot be expressed in any other form (*e.g.*, binary). They are not proceeded by a "$" ("$" diffentiates between a register and an immediate). They must be able to fit into the number of bits in the corresponding machine instruction (although some assemblers may use several instructions to support larger immediates).

Instructions. Table 6 shows the base assembly instructions. Each instruction corresponds directly to a single machine instruction discussed in Section 3.4. The semantics of each of these instructions is nearly identical to the semantics of the corresponding machine instruction (the syntax is of course quite different). The only difference, in terms of semantics, is that the branches use labels in place of the immediate, and a new instruction, load address (*i.e.*, **la**), can be used to store the address of a label into a register. The load address instruction is converted into a load immediate by the assembler once the address of the label is known.

It should be pointed out that the semantics of branches at the assembly level differs somewhat from branches at the machine code level. An assembly branch with a label branches to the instruction following the label, *i.e.*, it branches to an absolute address. A machine code branch with an immediate branches relative to the branch's PC. Of course, the assembly branch is converted into a machine instruction by the assembler. The assembler converts the label into the appropriate relative immediate. (Many assembly languages support branching with a label and with an immediate, however, Larc supports only branching with a label.)

For the base assembly instructions, the immediate of each instruction should fit in the width of the corresponding machine instruction. Otherwise, the assembler will not be able to translate it into the proper machine instruction. So, the immediate in the load immediate and load upper immediate should fit in 8 bits; and the immediate in the memory instructions should fit in 4 bits.

Type	Operation	Format	Semantics
ALU	addition	**add $a $b $c**	**Reg[a] = Reg[b] + Reg[c];**
	subtraction	**sub $a $b $c**	**Reg[a] = Reg[b] - Reg[c];**
	multiplication	**mul $a $b $c**	**Reg[a] = Reg[b] * Reg[c];**
	division	**div $a $b $c**	**Reg[a] = Reg[b] / Reg[c];**
	shift left logical	**sll $a $b $c**	**Reg[a] = Reg[b] $<<$ Reg[c];**
	shift right logical	**srl $a $b $c**	**Reg[a] = Reg[b] $>>>$ Reg[c];**
	bitwise NOR	**nor $a $b $c**	**Reg[a] = $\sim$ (Reg[b] | Reg[c]);**
	set on less than	**slt $a $b $c**	**if (Reg[b]$<$Reg[c]) Reg[a]=1; else Reg[a]=0;**
Load constant	load immediate	**li $a imm**	**Reg[a] = imm;**
	load address	**la $a label**	**Reg[a] = addr(label);**
	load upper immediate	**lui $a imm**	**Reg[a] = imm $<<$ 8;**
Branch	branch equal to 0	**beqz $a label**	**if (Reg[a] == 0) PC=addr(label);**
	branch not equal to 0	**bnez $a label**	**if (Reg[a] != 0) PC=addr(label);**
Memory	memory load	**lw $a imm($b)**	**Reg[a] = Mem[Reg[b]+imm];**
	memory store	**sw $a imm($b)**	**Mem[Reg[b]+imm] = Reg[a];**
Jump	jump and link register	**jalr $a $b**	**retn_addr=PC; PC=Reg[b]; Reg[a]=retn_addr;**
System call	system call	**syscall**	perform system call (type in **Reg[1]**)

Table 6: Larc base assembly instructions. **$a**, **$b**, and **$c** are register identifiers (**$0**-**$15**); **imm** refers to an immediate with the same width as the corresponding machine instruction (although immediates in extended assembly instructions can be up to 16 bits wide); **label** refers to a label in the assembly program; **Reg[]** refers to the registers; **Mem[]** refers to the memory; **PC** refers to the program counter; and **addr()** is a function that converts a label into a numeric address. The operators in the semantics column are expressed in Java. Note: the CPU automatically increments the PC after fetching each instruction from memory, before executing the instruction.

Similarly, the memory address of a label used in a load address should fit within 8 bits since it is converted into a load immediate. Likewise, for branches, the difference between the address of the label and the address of the branch should fit in 8 bits as the assembler will need to convert the branch into a branch using a relative immediate.

For the base assembly instructions, all of the immediates are signed except for in the load upper immediate instruction.

Figure 7 shows the extended assembly instructions. These can be implemented by the student for extra credit when building the assembler. The reference assembler supports these instructions. The extended instructions must be translated by the assembler into one or more base assembly instructions. In particular, there is a **move** instruction for copying values from one register to another. There are several new ALU instructions including **neg** (unary negation), **rem** (remainder

Type	Operation	Format	Semantics
Move	move	**move $a $b**	**Reg[a] = Reg[b];**
ALU	negation	**neg $a $b**	**Reg[a] = - Reg[b];**
	modulus	**rem $a $b $c**	**Reg[a] = Reg[b] % Reg[c];**
	bitwise NOT	**not $a $b**	**Reg[a] =** $\sim$ **Reg[b];**
	bitwise OR	**or $a $b $c**	**Reg[a] = Reg[b]** | **Reg[c];**
	bitwise AND	**and $a $b $c**	**Reg[a] = Reg[b] & Reg[c];**
	bitwise XOR	**xor $a $b $c**	**Reg[a] = Reg[b]** ^ **Reg[c];**
	set on less than or equal to	**sle $a $b $c**	**if (Reg[b]<=Reg[c]) Reg[a]=1; else Reg[a]=0;**
	set on greater than	**sgt $a $b $c**	**if (Reg[b]>Reg[c]) Reg[a]=1; else Reg[a]=0;**
	set on greater than or equal to	**sge $a $b $c**	**if (Reg[b]>=Reg[c]) Reg[a]=1; else Reg[a]=0;**
	set on equal to	**seq $a $b $c**	**if (Reg[b]==Reg[c]) Reg[a]=1; else Reg[a]=0;**
	set on not equal to	**sne $a $b $c**	**if (Reg[b]!=Reg[c]) Reg[a]=1; else Reg[a]=0;**
	any binary ALU instruction (with an immediate)	**op $a $b imm** (where **op** is replaced with operator name like **rem**)	**Reg[a] = Reg[b] o imm;** (where **o** is replaced with operator like **%**)
Branch	branch less than 0	**bltz $a label**	**if (Reg[a]** $<$ **0) PC=addr(label);**
	branch less than or equal to 0	**blez $a label**	**if (Reg[a] <= 0) PC=addr(label);**
	branch greater than 0	**bgtz $a label**	**if (Reg[a]** $>$ **0) PC=addr(label);**
	branch greater than or equal to 0	**bgez $a label**	**if (Reg[a] >= 0) PC=addr(label);**
	branch equal to (binary)	**beq $a $b label**	**if (Reg[a] == Reg[b]) PC=addr(label);**
	branch not equal to (binary)	**bne $a $b label**	**if (Reg[a] != Reg[b]) PC=addr(label);**
	branch less than (binary)	**blt $a $b label**	**if (Reg[a]** $<$ **Reg[b]) PC=addr(label);**
	branch less than or equal to (binary)	**ble $a $b label**	**if (Reg[a] <= Reg[b]) PC=addr(label);**
	branch greater than (binary)	**bgt $a $b label**	**if (Reg[a]** $>$ **Reg[b]) PC=addr(label);**
	branch greater than or equal to (binary)	**bge $a $b label**	**if (Reg[a] >= Reg[b]) PC=addr(label);**
	branch unconditional	**b label**	**PC=addr(label);**
Jump	jump (no linking)	**jr $a**	**PC=Reg[a];**
	subroutine call	**jal label**	**Reg[11]=PC; PC=addr(label);**

Table 7: Larc extended assembly instructions. **$a**, **$b**, and **$c** are register identifiers (**$0**-**$15**); **imm** refers to a 16-bit, signed immediate; **label** refers to a label in the assembly program; **Reg[]** refers to the registers; **PC** refers to the program counter; and **addr()** is a function that converts a label into a numeric address. The operators in the semantics column are expressed in Java. Note: the CPU automatically increments the PC after fetching each instruction from memory, before executing the instruction.

or modulus), **not** (unary not), **or** (bitwise inclusive OR), and **xor** (bitwise exclusive OR). There are also instructions for doing all forms of relational comparison (in addition to less than): **sle** for <=, **sgt** for >, **sge** for >=, **seq** for ==, and **sne** for !=.

There are also more control flow instructions. For instance, there are several branches, which allow for comparing a register to zero beyond checking if it is equal or not equal to 0: **bltz** (branch if less than 0), **blez** (branch if less than or equal to 0), **bgtz** (branch if greater than 0), and **bgez** (branch if greater than or equal to 0). There are also branches for comparing two registers: **beq** (branch if two registers are equal), **bne** (branch if two registers are not equal), **blt** (branch if the first register is less than the second), **ble** (branch if the first register is less than or equal to the second), **bgt** (branch if the first register is greater than the second), and **bge** (branch if the first register is greater than or equal to the second). Finally, there is a jump instruction that does no linking (**jr**) and an instruction (**jal**) for calling a (labeled) subroutine, which automatically saves the return address in the return address register (**$ra** or **$11**).

As an example, assume the programmer uses the extended instruction

```
blt $1 $2 foo
```

which branches to label **foo** if the value in register 1 is less than the value in register 2. An extended assembler (an assembler that supports the extended instructions) might convert this instruction into the following base assembly instructions:

```
slt $12 $1 $2
bnez $12 foo
```

Immediates work differently in the Larc extended assembly language than in the base language. First, every ALU instruction can be computed using an immediate in the place of the second source operand rather than a register (although an immediate cannot be used as the destination or the first source operand). Second, there are no limitations on the size of an immediate so long as it can fit within 16 bits. This rule applies to instructions shown in Table 6 as well as Table 7. For example, the following instruction is legal with an extended assembler:

```
lw $1 100($2)
```

It would not be legal with a base assembler because the immediate (100) will not fit within 4 bits, which is the size of the field in the machine instruction. An extended assembler will convert this instruction into several base instructions. For example, it might convert the instruction into the following:

```
li $13 100
```

Name	Type	Example	Description
word	data	**.word 97**	inserts a single word into the data section
space	data	**.space 10**	inserts several null words into the data section
asciiz	data	**.asciiz "foo"**	inserts a string terminated by null into the data section
text	section	**.text**	indicates start of text section
data	section	**.data**	indicates start of data section
globl	label	**.globl main**	makes a label globally visible

Table 8: Larc assembly directives.

```
add $13 $13 $2
lw $1 0($13)
```

For extended instructions, all immediates can be written as signed or unsigned although they will be treated as signed in the translated machine code.

5.5 Assembly Data

Larc supports three types of data elements in an assembly program: a single word, several null words (which can be used for arrays), or an ASCII string. To insert a data element into the data section, the programmer must use a *directive*. A *directive* is a macro function interpreted by the assembler. Table 8 lists the directives that are supported in a Larc assembly program. For now, we will focus on the first three, which are *data directives*, a directive that tells the assembler how to insert data into the translated machine program.

Word directives. To place a word of data in the program, the programmer can use ".word". For example, the following would direct the assembler to place 97, as a 16-bit, 2's complement, binary number into the machine code program:

```
.word 97
```

The value of the word must be written in decimal or hexadecimal (not binary). In addition, the programmer can direct the assembler to place the address of some label into the machine code program using the same directive:

```
.word label1
```

Space directives. To place several words in the program with the value 0, the programmer can use ".space". For example, the following would direct the assembler to place 50 words of 0 into the machine code program:

.space 50

The length of the space must be written in decimal or hexadecimal (not binary) and it must be positive. All word values are set to 0. This directive is often used to create space for an array in the static data section of memory.

Ascii directives. To place a string of characters in the program, the programmer can use ".asciiz". For example, the following would direct the assembler to place "abc", as a sequence of 8-bit, ASCII characters, into the machine code program:

.asciiz "abc"

The assembler translates this directive into the following:

0x0061
0x0062
0x0063
0x0000

Note: as with the space directive, the assembler may need to insert multiple words (in this case four) into the machine program rather than just one. The number of inserted words is equal to one more than the length of the string (a null terminator is put at the end of the string).

5.6 Assembly Programs

A Larc assembly program file ends with ".s" as is the case for some other architectures. The file contains two sections. The first is a text section, which contains the instructions, and the second is a data section, which contains the data elements. Both the text and data sections may also contain label definitions. However, instructions cannot be put in the data section and data directives cannot be put in the data section. The data section is optional (it can be excluded), but the text section is required in any legal Larc assembly program.

Two section directives from Table 8 are used to mark the start of text and data sections of the program. The text section is preceeded by the directive ".text" and the data section is preceeded by the directive ".data" (if a data section is defined). The programmer can put text and data sections in whatever order they wish (text then data or data then text), however, they can define only one of each type of section. Regardless of the ordering of the sections, the assembler will put the text section first in the Larc machine program followed by the data section. As in Larc machine language, the first instruction in the text section will reside at address **0x0000** and will be the first instruction executed.

```
    # Hello World program
.data
string: .asciiz "Hello, world!\n"

.text
    # print "Hello, world!\n"
    li $1 1
    la $2 string
    li $3 14
    syscall

    # halt
    li $1 0
    syscall
```

Figure 11: A simple assembly program that prints "Hello, world!\n" to the screen.

Like high-level programming languages (*e.g.*, C, Java), spaces and tabs are used to separate tokens in the program. Otherwise, spaces and tabs are ignored. So the programmer can use these to make their programs more readable. By convention, labels are usually not indented, but instructions and data items are indented. Newlines can be used to separate data items or instructions. However, no more than one instruction or one data item can be put on the same line (which is different than in high-level languages). The only exception is that a label can be combined with the instruction or data item following it (although it could also be put on its own line).

Figure 11 shows an example Larc assembly program for printing "Hello, world!" to the screen. This program would go in a file called "hello-world.s". Note: the text section and data section could have been switched (*i.e.*, text first then data). The assembler will translate this into the machine program shown in Figure 5 from Section 3 (Larc ISA section).

Separate compilation. The Larc assembler assembles a single program file, which must contain the entire program. It does not support separate compilation. However, separate assembly files can be merged prior to assembling, which does allow for separate compilation, at least for programs written in a high-level language. A merge program is contained within the **bin/** directory in the root directory (see Section 2) for merging several assembly files into a single assembly program. Each assembly file has its own name space, *i.e.*, a label in one file is distinct from a label in another file. As shown in Figure 8, there is a directive for making a label globally visible to the other files. For instance, if a data element is defined in one file, code in the other files could make use of that data element so long as the label marking the element is made global.

In this manual, you should not need to make use of separate compilation nor the global directive

(the directive is ignored by the assembler). This feature is included to support the development of larger applications (*e.g.*, an operating system) in a high-level language such as C.

5.7 Assembly Errors

There are several errors that can occur in a Larc assembly program, many of which have been mentioned in the text above. These include syntax errors, such as misspelling an operator, as well as semantic errors, such as referencing a non-existent label. The assembler, described above in Section 5.2, must handle all of these errors by stopping translation and reporting an appropriate message to the programmer.

Here is a list of both syntactic and semantic errors that can occur in a Larc assembly program:
Syntax errors:

- Unknown operator.
  If the programmer uses an operator (*e.g.*, 'mod') that doesn't exist in the Larc assembly language or they misspell an operator (*e.g.*, 'addd' instead of 'add').

- Improper instruction format.
  If the programmer incorrectly formats an instruction such as using an improper operand for a particular operation (*e.g.*, no immediate provided in a load word or store word). Note: this error will also occur if the programmer tries to place data in the text section.

- Unknown directive.
  If the programmer uses an recognized directive (*e.g.*, '.extern').

- Improper directive format.
  If the programmer incorrectly formats a directive (*e.g.*, 'asciiz abc' rather than '.asciiz "abc"'). Note: this error will also occur if the programmer tries to place an instruction in the data section.

- Improper label definition.
  If the programmer incorrectly writes a label definition (*e.g.*, 'foo;' rather than 'foo:').

Semantic errors:

- Undefined text section.
  If the programmer forgets to define the text section of the assembly program. This section is required in an assembly program (although the data section is not).

- Multiply-defined text section.
  If the programmer defines more than one text section. Larc assembly language does not support this feature.

- Multiply-defined data section.
  If the programmer defines more than one data section. Larc assembly language does not support this feature.

- Label redefinition.
  If the programmer writes 'foo:' in two places in the program.

- Non-existent label usage.
  If an instruction references a label that has not been defined (*e.g.*, 'beqz $4 foo' where 'foo:' does not appear anywhere in the program).

- Bad register identifier.
  If an instruction uses a non-existent register identifier (*e.g.*, '$17') or a restricted kernel register (*e.g.*, '$15').

- Oversized immediate.
  If an instruction uses an immediate that will not fit into the alloted bits for that particular field (*e.g.*, 'li $1 8192' – 8192 is too large to fit within a load immediate instruction's long immediate). Note: some assemblers might convert an instruction using a large immediate (*e.g.*, one that doesn't fit in 8 bits) into several instructions. However, immediates that require more than 16 bits are not supported.

6 Assembly Programming

This section describes how to write, run, and debug Larc assembly programs. It should be read before doing the labs described in Sections 13, 14, and 15. It need not be read before doing any of the other labs.

6.1 Writing Programs

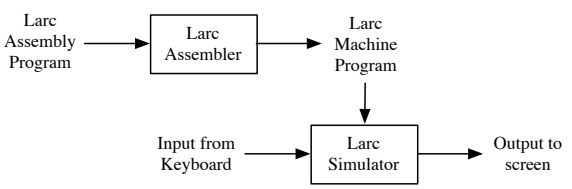
Assembly language programming is much easier than machine language programming, but it is still challenging, especially for non-trivial programs. It is important to remember the "good programming" tips from your introductory programming class. These tips will be even more critical when programming in assembly. Here are four of the more prevalent points:

- **Divide and conquer.**
  For larger programs, try splitting the program up into several components. Write each component separately rather than attempting to write the entire program all at once. Each component might be implemented in a separate subroutine.

- **Program incrementally.**
  Related to the point above, after writing each component make sure that you test it before moving on to the next component. This way, when you encounter an error, you will know that is likely related to the component that you just added. You may need to write some simple code to test each component, but this small amount of extra work will be worth it in the end (debugging a large assembly program all at once will take much more time).

- **Test extensively.**
  There are often many errors in the first draft of an assembly program. Do not assume that your program is the exception to the rule. Make sure that you thoroughly test your assembly program to find any and all errors. Devise various kinds of input to your program to make sure that it works in all cases. As with all programming, this is a crucial part of the development process.

- **Use good commenting.**
  Thoroughly document your program to make it easier to maintain your code. You will no



(a)

```
bash$ ./asm hello-world.s
bash$ ./sim hello-world.out
Hello, world!
bash$
```

(b)

Figure 12: Assembling and simulating the "Hello, world" Larc assembly program: (a) a picture of the process and (b) the shell commands.

doubt have errors when you first write an assembly program. If you have good commenting, it will be much easier to rework the parts of the program that are incorrect. For any subroutines you write, make sure that you specify all arguments and return values, and how they are passed from the caller to the called subroutine.

6.2 Running Programs

Figure 12(a) shows the process of running a Larc assembly program. This process is similar to other architectures. First, the assembly program is translated into a machine program by the assembler. Then, the machine program is run on the Larc simulator.

Figure 12(b) shows how the program "hello-world.s" is assembled and simulated via the shell assuming the assembler, simulator, and assembly program are all in the current directory. First, it is assembled using assembler called "asm". This produces a file called "hello-world.out", which contains the Larc machine program. This file is then passed to the simulator ("sim"), which prints "Hello, world!" to the screen. Note that if the programmer wanted to, they could edit and modify the machine program ("hello-world.out"). But in practice, programming is only done at the assembly level since it is easier and there is no loss of functionality. (Analogously, we do not often edit the generated assembly or machine code when programming in a high-level language.)

6.3 Debugging programs

The graphical debugger can be used to debug assembly programs as it was used to debug machine programs in Section 4.3. The debugger works similarly as with machine programs, however, there

```
    # Hello World program
.data
string: .asciiz "Hello, world!\n"

.text
    # print "Hello, world!\n"
    li $1 1
    la $2 string
    li $3 14
    syscall

    # halt
    or $1 $0 $0
    syscall
```

Figure 13: Assembly program for printing "Hello, world!" to the screen.

are some differences, which are described below. Note: the Larc debugger is modeled in part on the MipsPilot debugger [10].

Let's assume the programmer wants to debug the "Hello, world!" assembly program shown in Figure 13. This program is essentially the same as the one shown in the previous section, however, there is one difference. There is an **or** instruction in the middle of the program, which will be used to illustrate some features of the debugger. To debug the program, the programmer would type "./db hello-world.s" assuming the debugger and assembly program are in the current directory. Notice that the assembly file "hello-world.s" is passed to the debugger and not the machine file "hello-world.out". Passing the assembly file to the debugger automatically puts the debugger in assembly mode and allows the programmer to view assembly-specific details. However, if the programmer wished, they could first assemble the program and then debug the machine code as in Section 4.3.

Figure 14 shows a screen shot of the debugger. The window looks nearly identical to the window from Section 4.3, but there are some important differences. First, line numbers refer to the assembly file rather than the machine file. This makes it easier for an assembly programmer to map errors found in the debugger to instructions or data in the assembly file.

Another difference is that the instructions correspond to assembly file instructions. If the instruction uses a label, then that label is shown in the instruction with the address of the label in parentheses. The second instruction in the code panel is an illustration of this: **la $2 string (0x0008)**. In the machine program, this instruction loads the immediate 8 into register 2 (**li $2 8**). The label does not exist at the machine language level. But the debugger allows the programmer to view the instruction as an assembly instruction. It displays the address next to the label, so the programmer

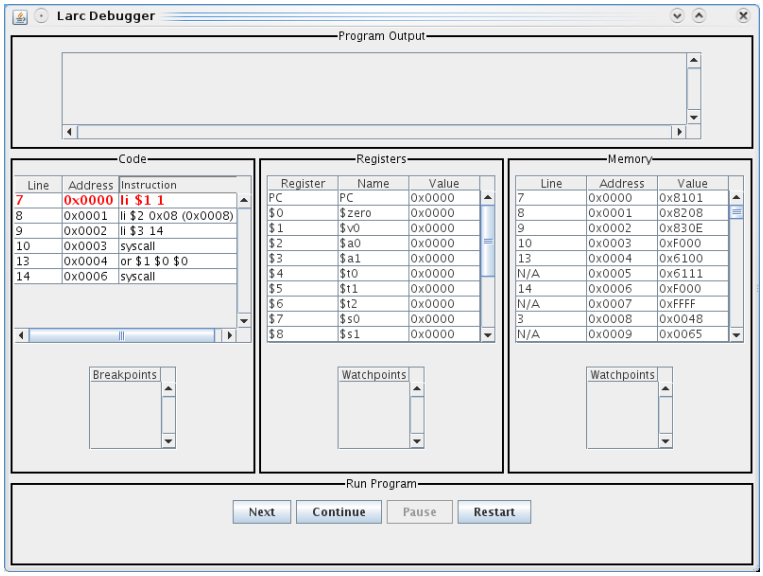


Figure 14: A screenshot of the Larc debugger running the "Hello, world!" assembly program.

can see the outcome of the instruction.

In addition, if the programmer uses an extended instruction, which might actually be composed of several base instructions, the debugger displays only the extended instruction. This can be seen in the **or** instruction (**or $1 $0 $0**). When assembled, this instruction is converted into two **nor** instructions. This is why the instruction following the **or** is at address **0x0006** rather than **0x0005**. If the programmer debugged the machine program rather than the assembly program, they would see two **nor** instructions rather than one **or** instruction.

All of these differences make it easier for programmers at the assembly level to identify and fix their errors. Therefore, if debugging an assembly file it is a good idea to pass the debugger the assembly file as opposed to the machine file.

Other than these differences, debugging works the same at the machine level as at the assembly level. The techniques discussed in Section 4.3 for stepping through the program, viewing values in different formats, or using breakpoints and watchpoints are the same.

7 System Architecture

This section describes the Larc system architectural support. It describes the machine interface when running an operating system (OS) in addition to user programs. It also describes how to program input/output (I/O) devices in Larc. In the lab described in Section 14, you will write a very simple OS that handles I/O for a single running program.

Section 7.1 gives some general details about the system software Larc supports. Section 7.2 describes how traps are handled by both Larc hardware and system software. Section 7.3 shows how memory is laid out when an operating system is used. Finally, Section 7.4 and Section 7.5 discuss how communication with I/O devices work.

Much of the material covered in this section is part of the complete instruction set architecture (ISA) but for organizational purposes we include it in its own section. This section should be read before doing the trap and I/O handler lab described in Section 14 although it need not be read before doing any of the other labs.

7.1 General

Before describing the system architecture, we should make clear what we mean by "operating system". An operating system is the software that manages the resources of the machine [9]. For instance, in a commercial machine, the operating system ensures that a running program does not write to a file for which it should not have access to. The operating system also provides an abstract interface to the hardware [9]. For instance, it will program input/output devices such as the keyboard and monitor, on behalf of a running program.

We also call this software the *kernel* to distinguish it from other software that is often thought of as part of the operating system. For example, it is customary to think of the shell as part of the Linux or Unix operating systems. But the shell is in fact a regular program. For example, we can remove the shell program we are using and install another. We cannot do this with the kernel unless we reinstall an entirely new operating system. In this manual, we focus on the kernel and ignore other software that often comes bundled with an operating system such as Windows 7 or Kubuntu Linux.

Non-kernel software is generally called a *user program* or *user application*. For example, the web browser, email client, and shell are user programs. They are named as such since these

Register	Full Name	Description
mem_base	memory base register	holds starting address of user program
mem_limit	memory limit register	holds limit of user program address space
sys_ra	system return address register	holds user program address to return to on a system return
pcr	processor control register	holds several fields (*e.g.*, kernel/user mode flag, trap type) for managing the system

Table 9: Larc system management registers.

programs are generally run on behalf of a particular user (although we will not support a multi-user system). A running user program is called *process*. Note: each program run is its own process even if it is run from the same program file.

The kernel essentially allows the processes to safely use the resources of the machine (*e.g.*, memory) while providing each process with an abstract interface to the machine. One could spend an entire semester looking at how developers build kernels to achieve these two goals but since this manual is for a course on introductory architecture we will just skim the surface in this manual. In particular, we will make several important simplifications. First, we will assume that only one process will run per boot of the system. In other words, the Larc system will run a single process after booting and when that program completes, the system will halt.

We will also focus on only two simple I/O devices: the keyboard and monitor. Furthermore, we will assume the monitor is character-based, *i.e.*, that it displays characters only and no other graphics. Communication with a keyboard and a character-based monitor are much simpler than communication with other I/O devices such as a hard disk or graphical monitor.

A set of special registers will be used for communicating with I/O devices as well as for managing the system. Table 9 shows the registers for managing the system and Table 10 shows the registers for managing I/O. These registers will work differently than the registers introduced in previous sections. We will discuss how they are written later in the section.

One important register in this list is the processor control register (**pcr**). (The other registers are discussed later.) It contains several fields for managing the system. Figure 15 shows the layout of the **pcr**. The high-order bit indicates whether the current program is running in user mode (1) or kernel mode (0). The next 7 bits indicate the particular trap type that occurred (*e.g.*, a system call trap). Both of these fields are automatically set by the processor as described below. The next 7 bits hold the identifier of the current running process. As we will not explore multiprocessing in this manual, this field can be ignored. The rightmost or low order bit is for halting the system. If a 1 is written into this position of the **pcr** then the processor halts. These last two fields are written

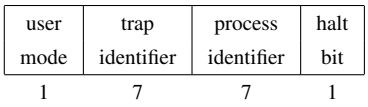


Figure 15: Contents of the processor control register (**pcr**).

by the kernel.

A new system instruction is also included in both the ISA and assembly language called a system return. This instruction is for use by the kernel when returning from a trap back into the user program. In assembly, the instruction is written as follows (with no operands):

sysretn

This assembly instruction is encoded into the following machine instruction:

1111100000000000 (binary format)
or
0xF800 (hexadecimal format)

Notice that the first four bits are the same as for a system call instruction. However, the fifth bit determines whether the instruction is a system call or system return. (Which is why for system calls this bit is required to be 0.)

Finally, we will also make use of the last two of the sixteen general Larc registers (shown in Figure 1 from Section 3). If you recall, these registers (**$14**-**$15** or, alternatively, **$k0**-**$k1**) were called kernel registers. The kernel registers are registers that can only be used by the kernel and not by a user program. The kernel can make use of them without worrying about saving any values needed by a suspended user program. In particular, these registers will be necessary when building a trap handler within the kernel.

7.2 Trap Handling

One key component of the kernel is the trap handler. The trap handler services all traps, which occur as a result of system calls, program errors, *etc.* It is always in memory even when running a user program. On a trap, the machine automatically transfers control to the trap handler.

There are many types of traps that can occur. For example, one class of traps occur when an error is made in a user program (*e.g.*, divide by zero). In this manual, we will focus on two types of traps: a boot trap during system initialization and on a system call. (Traps due to errors will cause the machine to simply halt.) To distinguish between the two types of traps, the trap handler can look at the trap identifier field in the **pcr**. For the system initialization trap, the trap handler should

set up registers and start the user program. For a system call trap, the trap handler should perform the corresponding task and then return to the program.

On a system call, the trap handler will need to return back to the user program at the correct address. It also needs to restore the state of the machine to what it was when the system call executed except for any register or memory entries where system call return values were placed. On any trap, the machine places the return address in the **sys_ra** register from Table 9. To return back to the user program, the kernel executes the **sysretn** instruction. This instruction will transfer control to the address in **sys_ra**.

On any trap, the processor automatically sets the kernel/user mode bit to 1 to transfer into kernel mode. While in kernel mode, no errors can occur, *i.e.*, the kernel can do whatever it wants. On a system return via **sysretn**, the processor automatically sets the kernel/user mode bit to 0 to transfer into user mode. While in user mode, an error will occur if an illegal memory address is accessed, a kernel register is accessed, a system return instruction is executed, and so forth. Normally, these types of errors would trigger an error trap, however, to simplify things in this lab, these errors will simply cause the simulator to trap.

Putting all of this discussion together, on a trap the following occurs, both in terms of hardware and kernel software:

1. **Trapping.** The processor will stop executing the current program (*i.e.*, a trap), save the return PC of the current program into the **sys_ra** register, enable kernel mode by setting the kernel/user mode bit in **pcr** to 0, put the trap identifier in the **pcr**, and transfer control to the kernel at address **0x0000**.

2. **Trap handling.** The trap handler code will then execute. It looks at the **pcr** to determine the particular trap that occurred (*e.g.*, a system call). It then services the trap.

3. **Restoring the user program.** After the trap handler has serviced the trap it returns back to the user program (assuming no error occurred) by restoring the program state and executing a **sysretn** instruction.

4. **System returning.** On a system return, the processor will disable kernel mode by setting the kernel/user mode bit in **pcr** to 1. It will transfer control to the address in **sys_ra**.

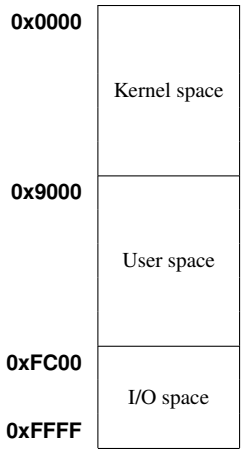


Figure 16: The memory layout when using an operating system.

7.3 Memory Layout

With an operating system, the layout of memory becomes somewhat more complicated. Both the kernel and the running program will reside simultaneously reside in memory. As shown in Figure 16, memory is split into three sections: kernel space, user space, and I/O space. The kernel space starts at address **0x0000** and ends at address **0x9000**. This section contains the operating system code and data for managing the machine and communicating with I/O devices. This space is large because the kernel can be potentially quite large. However, the kernel you will build in Section 14 will be very simple so much of this space will be unused.

The user space starts at address **0x9000** and ends at address **0xFC00**. This space contains one or more processes. In this manual, we will assume that only one process will be run so the user space will contain a single program. (In a course on operating systems this assumption would be lifted.) Note that this region will be further broken up into regions for holding instructions, data, the stack, and so forth (as will the kernel space).

Finally, the I/O space is reserved for communicating with I/O devices. By reading or writing to this part of memory, characters can be read from the keyboard, written to the monitor, *etc.*

Memory protection. With the kernel and device registers sharing memory with the user program, there must be ways to prevent the user program from corrupting the kernel and/or I/O devices. Larc has a very simple mechanism for doing this. (It also has support for a more complicated technique called *paging*, but we will not make use of paging in this manual.) It uses a base address and limit to confine the program to one area of memory. These values are held in the special

Register	Full Name	Description
kcr	keyboard control register	high-order bit indicates when keyboard is ready (other bits unused)
kdr	keyboard data register	holds character from keyboard
mcr	monitor control register	high-order bit indicates when monitor is ready (other bits unused)
mdr	monitor data register	holds character to send to monitor

Table 10: Larc I/O registers.

registers **mem_base** and **mem_limit**. The limit register holds the size alloted for the user program (**0xFC00-0x9000**). If the user program tries to access a memory address higher than the limit then the CPU will halt (in a real system this would cause a trap to the kernel). The base register holds the starting address of the user program (**0x9000**). Whenever the user program attempts to access memory, the base address is added to the program's address. This allows a user program to start at an address besides 0. While not explored in this manual, it would allow for running multiple programs simultaneously with all of the programs sharing memory.

Of course, the base/limit checking should be turned off when in the kernel. This is one of the functions of the kernel/user mode bit in the **pcr**. Base/limit checking only occurs when in user mode, *i.e.*, the kernel/user mode bit is 1.

7.4 I/O

One important task of the operating system, and a common reason for trapping via a system call, is communicating with input/output devices. This topic was skimmed over in Section 3, which described the Larc ISA, where system calls were assumed to be atomic, hardware-based, built-in operations. Instead, system calls trigger a trap and the kernel must handle the call. If I/O is required then the kernel communicates with the requested I/O device on behalf of the application.

Obviously, the kernel can not make use of a system call since the kernel implements system calls. To communicate with I/O devices, the kernel reads and writes special I/O device registers. We will have more to say about how to read and write these registers later in the section.

Table 10 lists the I/O registers we will make use of in this manual. There are four of them: two for the keyboard and two for the monitor. The **kcr** and **mcr** are control registers for the keyboard and monitor, respectively. They indicate when the keyboard or monitor is ready. If the leftmost bit (sign bit) is 1 then the device is ready, otherwise it is not. The other 15 bits are unused.

The **kdr** and **mdr** registers are data registers for the keyboard and monitor, respectively. They

are used to pass characters to or from the particular I/O device. As Larc currently supports ASCII and not Unicode, only the rightmost 8 bits (low-order 8 bits) are used for holding the character. The leftmost 8 bits (high-order 8 bits) are unused.

Polling. One problem in communicating with I/O devices is that the processor and I/O devices, such as the keyboard and monitor, work at very different speeds (*i.e.*, the processor is much faster than the I/O devices). Consequently, we need some way to synchronize the processor and I/O devices. There are two possibilities: polling or interrupt-driven I/O. In polling, the kernel repeatedly checks if the device is ready and, when it is, exchanges data with the device. In interrupt-driven I/O, the kernel executes another program while the device is busy and when the device is finished it interrupts the running program and a trap occurs back into the kernel, which can then exchange data with the device. Polling is simpler to implement but less efficient than interrupt-driven I/O. Because polling is simpler, Larc supports polling and not interrupt-driven I/O.

When communicating with an I/O device (*e.g.*, sending a character to the monitor, reading a character from the keyboard), the kernel will need to repeatedly check if the device is ready before sending or retrieving any data. In particular, the kernel will need to execute a loop such as the following (written in pseudocode):

```
repeat infinitely
    check if device is ready
        if it is ready then break
```

To check if the I/O device is ready the kernel will look at the ready bit in the device's control register. After breaking from the loop above the kernel can safely send or receive data via the device's data register.

7.5 Memory-Mapped I/O

As described above, there are several I/O and system management registers. These registers are read and written in a different way than the processor registers. They are read and written by loading or storing from or to specific memory locations. This technique is known as *memory-mapped I/O*. With memory-mapped I/O, we do not have to use up valuable space in the ISA for device registers (by adding bits to the register identifier or supporting special I/O instructions), but instead can make use of existing instructions (**lw** and **sw**). It does require alloting a part of memory to I/O devices but since memory is reasonably large, this is generally not a big deal.

Table 11 shows the memory-mapped registers along with their particular memory address

Name	Address	Description
kcr	**0xFC00**	keyboard control register
kdr	**0xFC01**	keyboard data register
mcr	**0xFC02**	display control register
mdr	**0xFC03**	display data register
mem_base	**0xFFFC**	memory base register
mem_limit	**0xFFFD**	memory limit register
sys_ra	**0xFFFE**	system return address register
pcr	**0xFFFF**	processor control register

Table 11: Larc memory-mapped registers.

where they can be read and written. All of the addresses are in the I/O space (**0xFC00**-**0xFFFF**). Notice that the I/O registers are at the beginning of the I/O space (**0xFC00**) while the system management registers are at the end (**0xFFFF**). Much of the I/O space is unused, which allows us to add new I/O devices or system management registers in the future.

The following is Larc assembly code for reading the **kcr**:

```
li $2 0xFC00
lw $1 0($2)
```

We need one load immediate instruction to put the address of **kcr** in a register and then one load memory instruction for reading the value. The code above would require an extended assembler (see Section 5) since the immediate in the load immediate requires more than 8 bits. Fortunately, the reference assembler is an extended assembler and can handle instructions with larger immediates (so long as they fit within 16 bits). Reading other memory-mapped registers would work similarly. To write a memory-mapped register we would use a memory store rather than a memory load. (Note: some registers, such as **kcr**, should only be read and not written, and vice versa.)

8 Lab 1: Decimal/Binary/Hexadecimal Conversion

In this lab you will write a Java program called **Convert.java** for converting between the decimal, binary, and hexadecimal number systems. Users will be able to choose whether they want to perform one of three types of conversions: (1) from decimal to binary and hexadecimal, (2) from binary to decimal and hexadecimal, or (3) from hexadecimal to decimal and binary. In each case, the user will enter a value in the appropriate format and your program will convert the value to the other two number systems and print both converted values to the screen.

For example, assume a user chose to perform conversion (2), *i.e.*, from binary to decimal and hexadecimal. The user would then be prompted to enter a binary value. Assume the user entered **1111111111111111**. Your program would then convert that binary value to the decimal value **-1** and the hexadecimal value **0xFFFF**, and print each value to the screen.

This lab can be done before reading any of the other informational sections (*i.e.*, Sections 3-7).

8.1 Files and Directories

To get started move to the directory **labs/lab01** within the toolset's root directory. This directory contains one file, **IO.java**, that will help you complete this assignment. **IO.java** contains code for reading input from the command-line. The API for **IO.java** is available online at *http://math.hws.edu/larc/api/lab01*. It is also available from the root directory in the subdirectory **api/lab01**.

8.2 Lab Details

Here are some important details pertaining to this lab. First, your program will need to support signed decimal values, *i.e.*, values that can be positive or negative. Your program will encode binary values using exactly 16 bits and a 2's complement representation. Note: that this also restricts the range of decimal and hexadecimal values.

Your program should work as follows. It should repeatedly prompt the user for a type of conversion:

1. Convert from decimal to binary and hexadecimal.

2. Convert from binary to decimal and hexadecimal.

3. Convert from hexadecimal to decimal and binary.

Once the user makes their selection, then your program should perform the conversion and print out the results. If the user enters 0 at the prompt, then the program should exit.

Here is a sample run of the program (user input is underlined):

Program for converting 16-bit representations
Options:
0: to exit
1: to convert decimal to binary and hexadecimal
2: to convert binary to decimal and hexadecimal
3: to convert hexadecimal to decimal and binary
Select an option: 1
Enter a decimal integer: 10120
Binary representation: 0010011110001000
Hexadecimal representation: 0x2788

Program for converting 16-bit representations
Options:
0: to exit
1: to convert decimal to binary and hexadecimal
2: to convert binary to decimal and hexadecimal
3: to convert hexadecimal to decimal and binary
Select an option: 2
Enter a binary value: 0010011110001000
Decimal representation: 10120
Hexadecimal representation: 0x2788

Program for converting 16-bit representations
Options:
0: to exit
1: to convert decimal to binary and hexadecimal
2: to convert binary to decimal and hexadecimal
3: to convert hexadecimal to decimal and binary
Select an option: 3
Enter a hexadecimal value: 0x2788
Decimal representation: 10120
Binary representation: 0010011110001000

Program for converting 16-bit representations
Options:
0: to exit
1: to convert decimal to binary and hexadecimal
2: to convert binary to decimal and hexadecimal
3: to convert hexadecimal to decimal and binary
Select an option: 0

Note that you must perform the conversions without the aid of external classes or packages. Java has many features in its extensive library (e.g., within the **Integer** class or **math** package) for converting to binary values or hexadecimal values. You may not use any of these. Furthermore, when reading in a binary or hexadecimal value, it must be read in as a **String**. You cannot use

an external class or package to read in the value and automatically convert it to a decimal integer, although you can do this when the user enters a decimal value.

A class **IO** is provided for reading values from the command-line. The (static) method **IO.readLine()** will block and wait for a line of input from the user. It returns a **String** containing the entered text. The (static) method **IO.readInt()** will block and wait for the user to enter a decimal integer. Note: with **IO.readInt()**, after typing the integer, the user will enter a newline, but the newline is discarded. If the user does not enter an integer before the newline, then 0 is returned.

8.3 Error Handling

Your program will also need to do some error checking to make sure that entered value is in the correct format. Decimal numbers should be straight forward. Binary values must consist of exactly 16 0's and 1's. Hexadecimal values must start with "0x" and be followed by 4 hexadecimal digits (0-9, A-F or a-f). For decimal values, you will also need to check that the value is in range. When converting to binary, the value must fit within 16 bits, encoded using 2's complement. If that is not the case then the entered value is out of range. If an error occurs, your program should print an error message and continue attempting to read in a "good" value.

8.4 Extra Credit

For extra credit, allow for conversions to and from floating point, in addition to the other formats. For example, if the user enters conversion type 1 through 3, then it should also print out the value as a floating point number. You will also need to add a new conversion type 4, which allows the user to enter a floating point value and converts this value into decimal, binary, and hexadecimal.

The floating point value will be represented using 16 bits. The leftmost bit is the sign bit, followed by 5 exponent bits, and 10 significand bits. This format works like other floating point formats except that the actual exponent is equal to the encoded exponent minus 15 (rather than 127 for 32-bit floating point values).

For example, if the user enters conversion type 4, and then enters **-19.75**, your program should convert this value into the decimal value **-13072**, the binary value **1100110011110000**, and the hexadecimal value **0xCCF0**. Note: the decimal value is computed from the 16-bit binary value.

8.5 Last Words

This lab will take a significant amount of time, so make sure that you start it early. Also, make sure you follow good programming practices as well as comment your code. This will be part of your grade.

Have fun!

9 Lab 2: Mock ALU

In this lab, you will model some important hardware in software. This will help give you some idea for how the hardware works. In particular, you are going to model a very simple arithmetic logic unit (ALU) in Java. As detailed in the exercise below, you will not be able to use the full functionality of Java, but instead will be restricted to mainly boolean operations, which will force you to write an ALU in software in a similar way to how it is designed in hardware.

You will find discussion and diagrams of ALUs found in most computer architecture textbooks [6, 7, 8] will be an important resource in this lab. These diagrams will help you write the Java code for your ALU.

This lab can be done before reading any of the other informational sections (*i.e.*, Sections 3-7).

9.1 Files and Directories

To get started move to the directory **labs/lab02** within the toolset's root directory. This directory contains several files and directories that you will need to complete this assignment: **ALU.java**, **Calculator.java**, and **Convert.class**. **ALU.java** contains skeleton code for your mock arithmetic and logical unit. You will complete this code in this assignment. **Calculator.java** contains a main class that repeatedly prompts the user for an operation (*e.g.*, addition, bitwise AND) some operands, and then returns the result of that operation. To perform the operation it uses **ALU.java**, which is mainly incomplete and will need to be completed for this lab. But both classes will compile, and you can run **Calculator**, although it will always return 0 as the result since **ALU** is largely unimplemented. **Convert.class** contains some methods for converting between decimal and binary, which is made use of in **Calcular.java**. You will not need to make use of it in **ALU.java**.

The API for all the provided Java classes is available online at *http://math.hws.edu/larc/api/lab02*. It is also available from the root directory in the subdirectory **api/lab02**.

9.2 Lab Details

You will complete the **ALU** class in **ALU.java**, which should perform various arithmetic and logical operations such as add, subtract, bitwise NOT (which is already partially implemented), bitwise OR, bitwise AND, and bitwise XOR. The **Calculator** class will make use of these operations by

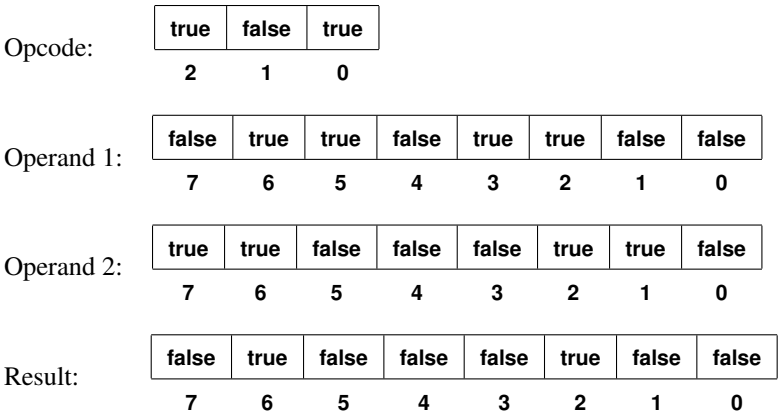


Figure 17: **ALU.calc** binary input and output encoded as boolean arrays.

repeatedly prompting the user to perform some operation and calling a **calc** method within the **ALU** class. (You will not have to worry about input/output in this lab.) The **calc** method is already contained in **ALU.java** although it is largely unimplemented.

Your **ALU** class will work with only two's complement binary values stored as boolean arrays where each element of the array represents a different bit in the binary value. Naturally, **true** means one and **false** means 0. The index of the element is the position of that bit in the binary value. For example, element 0 is the rightmost or low-order bit.

The **Calculator** class passes the **calc** method in **ALU** three binary numbers represented as boolean arrays. First, it passes it two 8-bit (*i.e.*, 8-element) operands to be used in the operation (note: the second operand is ignored for bitwise NOT). It also passes it one 3-bit (*i.e.*, 3-element) operation type, which we will call an *opcode*. For instance, an opcode value of 0 means perform an addition.

Figure 17 shows some example input to **calc**. The arrays are shown in reverse order so that the low-order bit is displayed on the far right. In this example, the opcode represents the unsigned, binary value **101** or **5** in decimal, which refers to the bitwise AND operation (see **ALU.java** for details on the particular opcodes). Operand 1 represents **01101100** or **108** in decimal, and operand 2 represents **11000110** or **-58** in decimal. Your **calc** method will need to compute the result of performing this operation. The result, which is also shown in Figure 17, is **01000100** or **68** in decimal.

Your task is to complete **ALU.java**. However, there are some restrictions on what you can use in Java to ensure that your **ALU** class works similarly to a hardware implementation. For example, you cannot use any arithmetic operator such as **+**, **-**, *****, **/**, **%**, or bitwise operations such as **&**, **|**, $\sim$,

<<, >>, >>> (there is one exception discussed below). You may only use boolean operands, *i.e.*, **&&**, ||, and **!**. These will act similarly to an AND gate, OR gate, and NOT gate, respectively. Also, with one exception, you may not use any statement besides an assignment statement and a method call. Therefore, you may not use if statements, switch statements, while loops, *etc.*. Finally, you may only call methods defined within the **ALU** class. You may not use any outside classes of any kind.

The one exception to all of these restrictions is that you are allowed to use a for loop to iterate over a boolean array. For example, if you need to take the complement of a binary value stored as a boolean array in the variable **binaryValue**, you could use the following:

```
for (int i = 0; i < 8; i++)
        binaryValue[i] = !binaryValue[i];
```

Note: the expression on the right hand side of the assignment can be as complicated as you like so long as it only involves boolean operators (*e.g.*, "**(!b1 && b2) || b3**"). Note also that you cannot perform any additional arithmetic in the for loop or use the iterator variable (*e.g.*, **i**) to set a binary value. The for loop can only be used to repeat a particular set of boolean operations.

To complete this exercise you find it useful look at diagrams of ALUs in your textbook or class notes. For example, a NOT gate translates into the boolean **!** operation, an AND gate translates into the boolean **&&** operation, and an OR gate translates into the boolean || operation. There is one method already defined for you in **ALU.java** that performs the bitwise NOT operation. It should be straightforward to use this same approach to implement the bitwise AND, OR, and XOR operations. You will then need to build a full adder, which you can use to build a ripple carry adder. For subtraction, you can extend your addition implementation to support both addition and subtraction.

One important thing that you will need to build is a multiplexor. This is necessary for selecting the result from the various operations (which are all done in parallel, as in hardware) based on the particular opcode. It can also be helpful when implementing subtraction. As a reminder, you cannot use if statements or switch statements, so you will definitely need to build a multiplexor out of boolean operators (look at a multiplexor diagram to see how this can be done).

Feel free to define as many methods as you would like, and to insert a method call where ever you would like. For example, you may want to have a method for each supported operation (add, subtract, bitwise AND, *etc.*). You may also want a method for doing a full 1-bit adder. Your add

operation can use the full adder method (*i.e.*, it can call it) to implement a ripple carry adder. A similar approach will help you in building the subtraction operation.

9.3 Error Handling

There is no error handling needed in this lab assignment although your ALU class should not crash on any input and return the correct result.

9.4 Extra Credit

For extra credit, add some other operations such as multiplication or bit shifting. As a 3-bit opcode supports 8 total operations and we are using 6 operations, you can add up to two more (using opcodes 6 and 7). Make sure you document any extra credit operations you implement at the comment at the top of the **ALU** class. Obviously, some operations are more difficult to implement than others, so the credit you will receive will depend on the particular operation(s) you implements.

9.5 Last Words

This lab will take a significant amount of time, so make sure that you start it early. Also, make sure you follow good programming practices as well as comment your code. This will be part of your grade.

Have fun!

10 Lab 3: Functional Simulator

In this lab you will write a Larc functional simulator in Java. As discussed in Section 3, a functional simulator takes a Larc machine program as input and behaviorally simulates a Larc processor. In other words, it simulates the input/output behavior (*i.e.*, keyboard events or monitor events) exhibited from running the program on a real Larc machine. Given some keyboard input, it will display the monitor output that a real Larc machine would display.

A functional simulator *does not* simulate the timing of a Larc machine, *i.e.*, the time it would take a Larc machine to run a particular program given some particular input. Therefore, your simulator does not have to model the microarchitecture and/or low-level hardware of the Larc machine. Instead, it only needs to keep track of the architectural state of the machine such as the memory, registers, and PC. Your simulator will repeatedly simulate the fetch-and-execute cycle, each cycle executing the next instruction and updating the architectural state. If the instruction performs some I/O, then your simulator will need to simulate this I/O. Your simulator will stop when the program performs a halt system call.

Important: before starting the assignment you should carefully read Section 3 that describes the Larc ISA. You need not read the other informational sections (*i.e.*, Sections 4-7).

10.1 Files and Directories

To get started move to the directory **labs/lab03** within the toolset's root directory. This directory contains several files and directories that you will need to complete this assignment: **Simulator.java**, **ISA.java**, **Convert.class**, **IO.java**, **sim**, and **tests/**.

Simulator.java contains skeleton code for the simulator program. You will complete this code in this assignment. **ISA.java** contains several variables and methods that you may find useful when implementing your simulator. The file **Convert.class** is a pre-compiled Java class containing methods for converting between decimal (represented as ints) and binary (represented as Strings). The file **IO.java** contains code for reading input from the command-line. The file **sim** is the reference simulator, which you can use to compare your simulator with. Finally, **tests/** is a directory containing several Larc machine programs, which you can use to test your simulator. Note: you may also need to write additional Larc machine programs to test your simulator (your simulator must be able to correctly run any Larc machine program).

The API for all the provided Java classes is available online at *http://math.hws.edu/larc/api/lab03*. It is also available from the root directory in the subdirectory **api/lab03**.

10.2 Lab Details

Here are some more details to help you complete this assignment. First, it should be pointed out that you are not implementing or modeling any hardware units when you write a functional simulator. Instead, you are implementing the ISA entirely in software. You are essentially writing a program that simulates the behavior (*i.e.*, monitor output) of a real Larc machine, given the same input program and keyboard strokes. Your program can accomplish this in any way you want (*i.e.*, you are not restricted to boolean operators).

You will, however, need to model some state stored in hardware, which is visible to software (*e.g.*, memory). But you do not have to worry about how the hardware is implemented that holds this state. You can simply use some Java data structure to house this state. In your simulator, you will need to model at least three parts of the Larc ISA: the program counter (PC), memory, and registers. You can use whatever data structures you find convenient for implementing these state elements.

In addition, you will of course need to be able to process Larc machine instructions, *i.e.*, perform the fetch-and-execute cycle. This includes fetching the next instruction to execute from memory, decoding that instruction from a 16-bit binary word, and correctly executing the instruction and updating the appropriate state elements. Refer to Section 3 for details on the semantics of each type of instruction. Your simulator will continuously perform the fetch-and-execute cycle until a halt system call is executed by the machine program. At that point, your simulator should terminate. (Your simulator should run infinitely if the program fails to perform a halt.)

In this assignment, your simulator will execute system calls atomically. In other words, the operation will be performed by the machine, in one fetch-and-execute cycle, rather than by software in the operating system. In a later lab (in Section 14), we will see how system calls are actually implemented.

The simulator should be implemented in the class **Simulator** within **Simulator.java**. This class already contains some skeleton code to help you get started. You may not use any outside classes, besides **String** and those that are provided in **labs/lab03** directory. You should make use of the

provided **Convert** class to convert between decimal, binary, and hexadecimal numbers (with binary and hexadecimal represented as strings). You should make use of the provided **IO** class for reading in input from the command-line on either a read string system call or read int system call (see Section 3 for details on these system calls). As mentioned above, the API is available online for both of these classes.

The directory **tests/** contains several Larc machine programs (ending in ".out") for testing the simulator. For example, the Larc machine program **hello-world.out** prints "Hello, world!" to the screen when run in the Larc simulator:

```
bash$ java Simulator hello-world.out
Hello, world!
bash$
```

When complete, your simulator should work similarly. The file **sim** is a reference simulator, which you can use to see what the correct output should look like. The following command runs the "Hello, world!" program above using the reference simulator:

```
bash$ ./sim hello-world.out
Hello, world!
bash$
```

10.3 Error Handling

Your simulator should catch syntactic errors as well as some runtime errors. Syntactic error checking (*i.e.*, when a line in the machine program is not correctly formatted correctly) is already implemented in the simulator. Your simulator will need to catch the following runtime errors:

- Dividing by zero.
  Performing a division instruction where the second operand is 0.

- Using uninitialized memory.
  Reading or executing a memory word that has not been initialized yet. A word in memory is initialized if it was part of the input program or it was stored to. An uninitialized memory access could happen in a load instruction, system call, or control transfer.

- Performing an unknown system call.
  Executing a system call instruction with an unknown system call identifier. Section 3 describes the system calls that need to be supported in your simulator.

You do not have to catch an arithmetic overflow error (operation such as multiply where the value is $< -2^{15}$ or $> 2^{15} - 1$). It is up to the programmer to avoid overflows.

In addition, your simulator should never crash as a result of executing a Larc program, even if that program is buggy (or has overflows). If the program is buggy, then your simulator should print an error message and exit (gracefully). Some care will need to be taken to ensure your simulator does not crash. For example, you will need to be careful with memory addresses. Memory addresses are unsigned integers, *i.e.*, they range from 0 to $2^{16} - 1$. If you mistakenly treat a memory address as a signed value then your simulator might crash (*e.g.*, terminate via an exception, which you did not explicitly throw).

10.4 Extra Credit

For extra credit, add support for memory-mapped I/O. You will need to read ahead to Section 7, which describes memory-mapped I/O. You can still treat system calls as atomic operations, however, give the programmer the option of working with input/output directly via memory-mapped device registers as opposed to solely through system calls. Note: you do not have to add the other system support, which is described in Section 7.

10.5 Last Words

This lab will take a significant amount of time, so make sure that you start it early. Also, make sure you follow good programming practices as well as comment your code. This will be part of your grade.

Have fun!

11 Lab 4: Machine Programming

This lab is broken into two parts. In part one, you will debug and annotate an existing, but errant, Larc machine program. In part two, you will write a new Larc machine program for computing whether a number is prime.

Important: Section 4 describes Larc machine language programming. Make sure you read this section carefully before starting this lab. Before reading Section 4 you should also read Section 3, which describes the Larc ISA. You need not read the other informational sections (*i.e.*, Sections 5-7).

11.1 Files and Directories

To get started move to the directory **labs/lab04** within the toolset's root directory. In this directory, you will find a Larc simulator, **sim**; a Larc debugger, **db**; and a Larc machine code program file that you must debug and annotate called **simple-check.out**. You will also add a file called **is-prime.out** for computing whether a number is prime.

11.2 Lab Details

Here are some more details to help you complete this assignment broken up into two parts.

Part 1: debugging and annotating an existing program. The program you will debug checks if a number is less than 10 and prints the result. A working version of the program would run as follows in the simulator:

```
bash$ ./sim simple-check.out
Enter a number: 5
Your number is less than 10
bash$ ./sim simple-check.out
Enter a number: 11
Your number is greater than or equal to 10
bash$
```

Your first task in this part of the lab is to comment the machine code file. Recall from Section 4 that comments follow '#' and must use up an entire line. Empty lines are allowed and are helpful for improving readability. You can use the debugger to help in determining what each line in the program represents. However, putting the assembly representation of an instruction or data element is not adequate. Instead, you should describe in english what each line represents and why it is

required in the program.

Your second task in this part of the lab is to identify and fix 5 bugs within the program. You can use your comments from the first task as a guide in helping identify bugs. Some of the bugs may not be apparent when just looking at the program. Use the debugger described in Section 4 to help in identifying errors that you cannot find by inspection. Once you have found each bug, fix it in the program, leaving the errant line in a comment, and using another comment to describe the bug and how you fixed it.

Although the machine program uses a binary format, you are welcome to convert any or all of into the hexadecimal format, to make the program more readable.

Part 2: writing a new program. In the second part of this lab, write a Larc machine program that computes whether a number is prime (divisible by only 1 and itself) or not. Your program should prompt the user for a number >= 2, read in the number, and print whether or not it is prime. Here is some sample output from a working version of this program:

```
bash$ ../../bin/sim is-prime.out
Enter a number >= 2: 5
5 is prime.
bash$ ../../bin/sim is-prime.out
Enter a number >= 2: 10
10 is not prime.
bash$
```

You should submit the program in a file called **is-prime.out**. You should include comments in the program, describing each line, as well as how the program works. You may want to encode your machine program using the hexadecimal format (see 4) as it is easier to read and work with.

11.3 Error Handling

For part one of this lab, there is no required error handling. For part two, your primes program should make sure that the number entered is greater than or equal to 2. If it is not, it should reprompt the user repeatedly, until a number is received that is greater than or equal to 2.

11.4 Extra Credit

For extra credit, write an additional program in Larc machine code, something more sophisticated than the primes program. Obviously, the amount of extra credit will vary based on the program that you write.

11.5 Last Words

This lab will take a significant amount of time, so make sure that you start it early. Also, make sure you follow good programming practices as well as comment your code. This will be part of your grade.

Have fun!

12 Lab 5: Assembler

In this lab you will write an assembler for translating Larc assembly code into Larc machine code. For details on Larc assembly code and Larc machine code, see Sections 5 and 3, respectively. The assembler takes as input the name of a Larc assembly program and creates a file containing the Larc machine program. Larc assembly programs must end in ".s" and Larc machine programs must end in ".out". The assembler leaves the base name from the assembly file intact in the name of the machine code file. For example, if the assembly program is named **foo.s** then the assembler writes the machine code to **foo.out**. Likewise, if the assembly program is named **tests/foo.s** then the assembler writes the machine code to **tests/foo.out**.

Important: Section 5 describes the Larc assembly language as well as the Larc assembler. Make sure you read this section carefully before starting this lab. You should also read Section 3 prior to reading Section 5. You need not read the other informational sections (*i.e.*, Section 4, Section 6, and Section 7).

12.1 Files and Directories

To get started move to the directory **labs/lab05** within the toolset's root directory. This directory contains several files and directories that you will need to complete this assignment: **Assembler.java**, **Parser.java**, **ISA.java**, **Asm.java**, **Inst.java**, **Datum.java**, **Label.java**, **Convert.class**, **asm**, **sim**, **db**, and **tests/**.

Assembler.java contains skeleton code for the assembler program. You will complete this code in this project. **Parser.java** contains code for reading in and parsing the assembly program (*i.e.*, determining the syntactic structure of the program and creating Java objects to represent each element). **ISA.java** contains several variables and methods that you may find useful when implementing your assembler. **Inst.java**, **Datum.java**, and **Label.java** contain classes for representing instructions, data directives, and labels, respectively. They all inherit from the generic class **Asm** within **Asm.java**. The file **Convert.class** is a pre-compiled Java class containing methods for converting between decimal, binary, and hexadecimal. The files **asm**, **sim**, and **db** are a working, reference assembler, simulator, and debugger, respectively, which you can use when testing your assembler. Finally, **tests/** is a directory containing several Larc assembly programs, which you can use to test your assembler. Note: you may also need to write additional Larc assembly programs

to test your assembler (your assembler must work correctly on any Larc assembly program).

The API for all the provided Java classes is available online at *http://math.hws.edu/larc/api/lab05*. It is also available from the root directory in the subdirectory **api/lab05**.

12.2 Lab Details

Here are some more details to help you complete this assignment. Your assembler must follow the specifications laid out in Section 5.

Some parts of the assembler will be provided for you such as a class for parsing a Larc assembly program (**Parser.java**), classes for representing a generic assembly item (**Asm.java**), instruction (**Inst.java**), data element (**Datum.java**), and label (**Label.java**), as well as a class containing some useful variables and methods pertaining to the ISA (**ISA.java**). You will be responsible for completing **Assembler.java**, which converts the parsed assembly program into a machine program. It contains skeleton code to get you started. You will also be responsible for understanding the files (**Parser.java**, **Asm.java**, **Inst.java**, **Datum.java**, and **Label.java**) that you do not have to implement (see the API at *http://math.hws.edu/larc/api/lab05* for details).

The parser builds two vectors provided from the assembler. The first contains elements from the text section (instructions and labels) and the second contains elements from the data section (data elements and labels). Your assembler will need to perform multiple passes (*i.e.*, loops) over these vectors of assembly items. In the last pass, these items will be translated to machine code and the words (either in binary or hexadecimal format) will be written to the machine code file.

Currently, **Assembler.java** simply prints the assembly program back to the screen in a method called **printProgram** (you will eventually comment the call to **printProgram** out). This method is provided to you to show you how to work with the two vectors. You should look at it carefully before writing new code. Each pass that you perform over the assembly program will look structurally similar to the code in **printProgram**.

You will probably need to make two to three passes over the assembly program. In the first pass, you will need to compute an address for each instruction, data element, and label, which will be used later. In the second pass, you will patch each instruction or data element that uses a label. The reference to the label will be replaced with the appropriate numeric value. See the **Inst** and **Datum** API for details on how to do this. Finally, in a third pass, which could be merged with the

second pass, each assembly item is converted into binary words and stored into a vector, which is eventually written out to the machine code file. Most elements will correspond to one 16-bit word although not all will (*e.g.*, ".asciiz" and ".space" data elements).

When you compute addresses for each assembly item in the first pass, you will also need to set up a map from label name to its address. This will be needed in the second pass to convert labels in the text or data sections into the appropriate immediate. The **Label** class has a built-in map, which you can use to keep track of label addresses. To add to the map, you would call the method **Label.addToMap**. For example, the following adds the label "label1" to the map, giving it address 2000:

```
Label.addToMap("label1", 2000)
```

To find the address of a label, you would call the method **Label.getFromMap**. For example, the following sets variable **addr** to the address of the label named "label1":

```
int addr = Label.getFromMap("label1")
```

If "label1" has not been added to the map then **Label.getFromMap** returns -1. See the API for more details on how to use these methods.

In the third pass, when you convert the assembly program elements into machine code words, you should fill the vector **binaryProgram**. Each entry in the vector holds the next word, represented as a string, of the machine code program. This vector is already defined in **Assembler.java**. The method **writeMachineCode**, which has also already been defined, will automatically write the contents of this vector to the appropriate out file.

The directory **tests/** contains several Larc assembly programs for testing the assembler. For example, the Larc assembly program **hello-world.s** contains a Larc assembly program for printing "Hello, world!" to the screen.

Below is an example use of a working version of your assembler along with the reference simulator. The commands below convert **tests/hello-world.s** to a machine code file in **tests/hello-world.out**:

```
bash$ java Assembler tests/hello-world.s
bash$
```

If there are no errors then no output should be printed to the screen. The code below then runs the machine code file:

```
bash$ ./sim tests/hello-world.out
Hello, world!
bash$
```

You could also debug the program by swapping "./sim" with "./db".

To compare with the reference assembler you could execute the following command:

```
bash$ ./asm tests/hello-world.s
bash$
```

12.3 Error Handling

As illustrated by the code above, your assembler will need to handle any errors in the assembly file. The parser (in **Parser.java**) handles syntactic errors such as a bad register name or unknown operator. But some errors must be caught in **Assembler.java**. In particular, your assembler must catch the following:

- Labels that are defined more than once.
- References to non-existent labels.
- Use of extended operations such as **rem** (unless you are doing the extra credit).
- Immediates in instructions that are too large or too small (unless you are doing the extra credit).
- Labels that map to immediate or word values that are too large (unless you are doing the extra credit).

Finally, your assembler should never crash as a result of assembling a Larc program, even if that program is buggy. If the assembly program is buggy then your assembler should print an error message and exit (gracefully).

A method **assemblyError** is provided for reporting errors. You should call it with the appropriate message if your assembler discovers an error.

12.4 Extra Credit

For extra credit, you can add support for the extended instructions as described in Section 5. Your assembler should convert each extended instruction into the appropriate base assembly instruction(s). Some extended instructions are easy to translate but others are more challenging. For example, translating an unconditional branch (*e.g.*, **b foo**) is straightforward. On the other hand,

translating a load immediate instruction whose immediate does not fit in 8 bits is much more difficult. (Note: if not doing the extra credit either of these instructions would be an error).

You do not have to necessarily support all of the extended instructions. But obviously, the credit will vary based on the particular extended instructions that you add support for.

Warning: adding support for all of the extended instructions is difficult and time consuming. Make sure you complete the base instructions before moving on to the extended instructions.

12.5 Last Words

This lab will take a significant amount of time, so make sure that you start it early. Also, make sure you follow good programming practices as well as comment your code. This will be part of your grade.

Have fun!

13 Lab 6: Assembly Programming – Nim

In this lab, you will write a Larc assembly program for playing the game Nim. In Nim, there are some number of beans laid out and the players alternate turns in which a player can pickup one, two, or three beans. The winner is the player who picks up the last bean. (There are many variations on this game; perhaps you may have heard of another version.)

Important: Section 6 describes Larc assembly programming. Make sure you read this section carefully before starting this lab. Before reading Section 6 you should also read Sections 3-5, which describe the Larc ISA, machine programming, and assembly language. You need not read informational section on the system architecture (*i.e.*, Section 7).

13.1 Files and Directories

To get started move to the directory **labs/lab06** within the toolset's root directory. There are three existing files in this lab: **asm**, a Larc assembler for assembling your program; **sim**, a Larc simulator for running your assembled program; and **db**, a debugger for debugging your assembly program. You will also create and write a file called **nim.s**, which will contain your Nim program.

13.2 Lab Details

Here are some more details to help you complete this assignment. You will write a two-player version of the game. One user will start out by selecting the number of beans to start with. Then each user will enter their name. After that, the two users will alternate turns, selecting beans until there are none left (in which case, the current player wins). Each turn, your program will prompt the user for 1, 2, or 3 beans. When prompting a player, you should also print out their name. When their our no more remaining beans, your program should print out the winning player, and halt.

Here is some sample output from the program (user input is underlined):

How many beans to start? 10
Player 1, enter your name: Alice
Player 2, enter your name: Bob
Alice, how many beans to pick up? (10 left) 3
Bob, how many beans to pick up? (7 left) 2
Alice, how many beans to pick up? (5 left) 1
Bob, how many beans to pick up? (4 left) 2
Alice, how many beans to pick up? (2 left) 2
Alice, you win!

Your program should work similarly.

(As an aside, in Nim, depending on the number of starting beans, either the first player can always win or the second player can always win. Can you figure out why?)

13.3 Error Handling

Your assembly program must do some error handling. If the number of starting beans is less than 0 your program should print an error message and continue prompting the user for a value that is greater than 0. Moreover, if a user does not enter 1, 2, or 3 on their turn, or they pick more beans than are remaining, your program should print an error message and continue prompting the user for a legal value.

13.4 Extra Credit

For extra credit, write a 2-player version of Tic Tac Toe in Larc assembly. Note: this is much more challenging than Nim and should only be attempted after finishing the Nim program.

13.5 Last Words

This lab will take a significant amount of time, so make sure that you start it early. Also, make sure you follow good programming practices as well as comment your code. This will be part of your grade.

Have fun!

14 Lab 7: Tiny OS

In this lab, you will write a tiny operating system (OS). In particular, you will write a trap handler for Larc, which will handle system calls performed by a user program. Your trap handler will perform the system calls by communicating directly with the input/output (I/O) devices via the device registers (as opposed to in earlier labs).

Important: Section 7 describes the Larc system architecture for supporting an operating system. Make sure you read this section carefully before starting this lab. Before reading Section 7 you should also read the earlier informational sections (*i.e.*, Sections 3-6).

14.1 Files and Directories

To get started, move to the directory **labs/lab07** within the toolset's root directory. There are several existing files in this directory: **asm**, an assembler for assembling your trap handler; **sim**, a simulator for running a test program with your operating system; **db**, a debugger for debugging your operating system; and **tests/**, a directory containing several programs for testing your operating system. Unlike in previous assignments, the test programs for the base credit, which are in directory **tests/base**, and the test programs for extra credit, which are in directory **tests/extra-credit**. If you do not do the extra credit then the extra credit test programs will not work with your operating system.

14.2 Lab Details

Unlike in previous labs, when running a program using the simulator, two programs will be loaded into memory rather than just one. The first is the trap handler, which is loaded into memory start at address **0x0000**. The (user) program now resides at address **0x9000**. The simulator will start by performing the special system initialization trap, which allows the trap handler to initialize the state of the machine and start the user program. Then the user program will run. When the user performs a system call, the simulator will trap and transfer control to the trap handler. The trap handler will service the request and either halt the machine (on a halt system call) or return back to the user program. (Note: in a real system, a trap could also occur due to a program error but we will ignore these; on a program error the machine will automatically halt.)

You will write your trap handler in Larc assembly code. This code will look similar to other

Larc assembly code except you cannot use of any system calls (after all, you are implementing them). You will also need to use one new option when assembling the trap handler. You will need to specify that you are assembling kernel code with the "-k" flag. This flag tells the assembler to use the kernel registers as assembler registers. You will need this to ensure you do not accidentally modify any user register (registers 1-13) and corrupt the user program. The assembler will also not give a warning when using kernel registers or when using the system return instruction.

To use the operating system in the simulator, you must use the "-t" flag ('t' for trap handler), specifying the operating system assembly file after the flag. For example, the following command runs the program **tests/base/hello-world.out** using the operating system **os.out**.

```
bash$ ../../bin/sim -t os.out tests/base/hello-world.out
bash$
```

To debug assembly code when using an operating system, you can do the following:

```
bash$ ../../bin/db -s "-t os.out" tests/base/hello-world.out
bash$
```

Trap handling. The first instruction in your OS text section will be the start of your trap handler (since it goes at address **0x0000**). This code will need to determine the particular trap and service it. As described in Section 7, the type of the trap is placed in the **pcr** automatically by the CPU. The code in your OS can read the **pcr** to figure out whether it is a system initialization or system call trap.

You will need to be careful not to modify any user register without saving and restoring it in your trap handler. To achieve this, you can make use of the two kernel registers **$k0** and **$k1**. But be careful, these registers are also used as assembler reserved registers when assembling the OS. If you make use of extended instructions that write to the kernel registers, the kernel registers can become corrupted. Therefore, you should use the kernel registers sparingly, just to save a few user registers at the beginning of the trap handler. After that, they should not be used. For instance, the following pseudo-instructions would be safe:

```
# beginning of trap handler
li $k0 <OS address where registers can be saved>
sw $10 0($k0)
add $10 $k0 $0
# code below would then use $10 not $k0
```

Once the trap has been handled, your trap handler will need to return back to the user program. You should use the special instruction **sysretn** for doing this. This instruction will put the processor back in user mode and return to the correct point in the program (by returning to the address in

sys_ra where the CPU saved it at the start of the trap). You should only use this instruction when the trap handler has completed. For normal subroutine returns in your trap handler, you should use a standard **jalr** instruction.

Use care in structuring your assembly as this is a non-trivial section of code. In particular, you will probably want to divide the trap handler into several subroutines. For example, you could create a subroutine for handling each type of trap, and also a subroutine for handling each type of system call (halt, print string, read string). You might also make subroutines for reading a single character from the keyboard or writing a single character to the display.

System calls. Table 3 of Section 3 shows the system calls you will need to support although two of them are left for extra credit. In particular, your trap handler will need to support halt (0), print string (1), and read string (3). For extra credit, you can extend your trap handler to support print int (2) and read int (4).

Memory-mapped I/O. As discussed in Section 7, Larc has memory-mapped I/O. To read or write a device register, the programmer must load or store the appropriate word in memory. Table 11 on page 66 lists the device registers along with the memory address where they are mapped. For instance, to read data from the keyboard the programmer reads the keyboard data register at address **0xFC01**. To write data to the display monitor the programmer writes the data register at address **0xFC03**.

Both the keyboard and display devices are character based. When a key is pushed on the keyboard, the character representing that key is passed in the low-order 8 bits of the keyboard data register (the high-order 8 bits are ignored). Likewise, when the processor needs to write a character to the display, it writes the character to the low-order 8 bits of the display data register. (Note: unlike modern displays, the Larc display does not have support for setting arbitrary pixels on the monitor.)

Synchronization. Synchronization between the processor and I/O devices is handled via polling. Each device has a corresponding control register that contains a ready bit (the high-order bit) indicating whether the device is ready or not. When the trap handler needs to read or write a character to a device, it should repeatedly check this ready bit (*i.e.*, poll it). Once the ready bit is set, then it is alright for the trap handler to read or write the data register. Synchronization errors will result if the (buggy) trap handler fails to wait for the ready bit to be set.

Halting. In addition to the keyboard and display device registers, there is also a processor control

register. This device register can be used for halting the machine (among other things). If the low-order bit in the processor control register is set then the machine is automatically halted. Your trap handler will need to use the processor control register when the program performs a halt system call.

Memory protection. In addition, to handling system calls your trap handler should also protect the kernel and I/O memory from an errrant program. It can do this by setting two memory-mapped I/O registers: **mem_base** and **mem_limit**. The CPU automatically adds the address in **mem_base** to any memory address used by the program. The CPU also checks that each address is less than the value in **mem_limit**. Setting these two registers effectively sandboxes the user program into the user memory space.

These registers should be set during system initialization. The starting address of the user space, or the base address, is **0x9000**. The limit or length of the user memory space is **0xFC00-0x9000**.

14.3 Error Handling

Your trap handler will need to catch several errors that can occur due to a bad system call. In particular, your trap handler must check the operands of the system call to ensure they do not contain illegal values. These include:

- Bad system call identifier.
  The program performs a system call with an unknown system call identifier (*e.g.*, 100).

- Bad memory address.
  The program passes a bad memory address as an operand (*e.g.*, the address of the string in a print string system call). Any memory addresses must be less than **0x6C00**.

If the program makes one of the errors above then your trap handler should print an appropriate error message and halt. In addition, your trap handler should never crash as a result of a system call (illegal or otherwise).

14.4 Extra Credit

For extra credit, add support for two additional system calls to your trap handler: printing an int and reading an int. Hint: if you define subroutines for converting int to string and vice versa, then you could leverage your existing code for printing and reading strings.

14.5 Last Words

This lab will take a significant amount of time, so make sure that you start it early. Also, make sure you follow good programming practices as well as comment your code. This will be part of your grade.

Have fun!

15 Lab 8: Assembly Programming using the Stack

In this lab, you will work with assembly programs that use a stack for storing local subroutine data. In particular, you will write an assembly program with a recursive subroutine (*i.e.*, a subroutine that calls itself) for computing the greatest common divisor using Euclid's algorithm. You will also exploit an unchecked copy into an array stored on the stack to hijack an existing program (called a *buffer overflow* attack).

Important: Section 6 describes Larc assembly programming. Make sure you read this section carefully before starting this lab. Before reading Section 6 you should also read Sections 3-5, which describe the Larc ISA, machine programming, and assembly language. You need not read informational section on the system architecture (*i.e.*, Section 7).

15.1 Files and Directories

To get started move to the directory **labs/lab08** within the toolset's root directory. There are several existing files in this directory. These include: **asm**, a Larc assembler for assembling your programs; **sim**, a Larc simulator for running your assembled programs; and **db**, a debugger for debugging and inspecting your assembly programs. There is also a file called **print-reverse.s** for printing a set of inputted integers in reverse order. It contains a buffer overflow vulnerability, which you will exploit by giving it malicious input values and, in so doing, hijack the program.

You will also create several files. You will create a program called **gcd.s**, which will compute the greatest common divisor using Euclid's algorithm. You will also create a file called **input.txt** with the malicious values used to hijack the print reverse program. Finally, you will create a file called **input-annotated.txt**, which will contain the input values with annotations describing each input.

15.2 Lab Details

Here are some more details to help you complete this assignment broken up into two parts.

Part 1: writing a stack-based assembly program. In this part, you will write a complete assembly program called **gcd.s**, which will use the stack for storing local subroutine data. It will prompt the user for two positive numbers and print out the greatest common divisor (GCD) of those two numbers. It will use recursion to find the GCD.

In particular, your program will need four subroutines, which must follow the specifications below (if you do not follow the specifications you will not receive full credit):

- **main.** A main subroutine that follows ".text". This will call the other subroutines and halt when finished. In particular, it will call **get_number** (discussed below) twice to get two positive numbers from the user. It will then call **gcd** (discussed below) to compute the greatest common divisor of those two numbers. Finally, it will perform a halt.

- **get_number.** This subroutine will prompt the user for a positive number and read in the number. If the number is not positive, it should print an error message and continue reading a number until it receives a positive number. It will return this number to **main** in register 1 (note: it must use register 1).

- **gcd.** This subroutine will find the greatest common divisor (GCD) between the two numbers, taken as parameters from **main**. It will use recursion to find the GCD. In particular, it will use Euclid's algorithm:

  $$\text{gcd}(x,y) = \begin{cases} x, & \text{if } y \text{ is } 0 \\ \text{gcd}(y, x\%y), & \text{otherwise} \end{cases}$$

  Note: **x%y** is the remainder of **x** divided by **y**. **gcd** will call the subroutine **mod** (discussed below) to compute **x%y** (you cannot use the extended instruction **rem** although you can use other extended instructions). Note also: your subroutine must follow the definition above, *i.e.*, it must use recursion (it cannot use loops). Finally, **gcd** must take the first parameter (*i.e.*, the first number) in register 2, the second parameter (*i.e.*, the second number) in register 3, and return the result in register 1 (note: it must use these particular registers).

- **mod.** This subroutine computes the remainder when dividing one number by another (*i.e.*, modulus – %). It returns the result to the caller (*i.e.*, **gcd**). It should not use the extended instruction **rem** to compute the remainder although other extended instructions are allowed. **mod** must take the first parameter (*i.e.*, the first number) in register 2, the second parameter (*i.e.*, the second number) in register 3, and return the result in register 1 (note: it must use these particular registers).

In addition, each subroutine must preserve the values of all registers it modifies except register 1 (**$v0**). In other words, the values of each register besides register 1 should be the same when the

subroutine starts as when it finishes. Although some registers are technically not saved registers (see Section 3), you will treat all registers besides 1 as saved in this lab.

The stack will be used for saving registers modified by a subroutine. At the beginning of each subroutine, a new stack frame must be allocated and the modified registers besides 1 must be saved to that frame. At the end of each subroutine, just before returning, the stack frame must be deallocated and each modified register must be restored. This must be done even if the subroutine is not recursive, although you can omit allocating/deallocating a frame if the subroutine does not modify any registers besides 1.

The stack pointer (**$10** or **$sp**) register should be used for managing the stack. To allocate the stack frame, you can simply decrease the stack pointer by the size of the frame needed (decrease, since the stack grows in reverse). To deallocate the stack frame, you can increase the stack pointer by the size of the frame. Memory stores can be used to save register values to the frame. Likewise, memory loads can be used to restore registers to their original values.

As in previous labs, make sure you comment your code so that it is easy to read and understand.

Part 2: overflowing a stack buffer. In this part, you will play the role of a hacker trying to subvert a program. (Note: we are not encouraging you to become a hacker, but to prevent computer attacks you must be able to think like an attacker). The program, in the file **print-reverse.s**, reads in a sequence of numbers, terminated when the user enters 0, and prints them out in reverse order. To do this, the program reads the numbers in and stores them into an array in reverse order. It then prints them out, which prints them in reverse since they were stored in reverse order.

For example, here is an example run of the program (user input is underlined):

```
bash$ ./sim print-reverse.out
Enter the next number (0 to stop): 1
Enter the next number (0 to stop): 2
Enter the next number (0 to stop): 3
Enter the next number (0 to stop): 0
Numbers in reverse order:
3
2
1
bash$
```

Unfortunately, the program fails to check whether the length of the array in memory is exceeded (which is 15 in this case). Therefore, a clever hacker can overflow the array, corrupt a return address, execute their own attacker code, and subvert the program. Your task is to do this and force the program to print out ”zoinks!\n” (without the quotes) to the screen and exit. You have been

given the assembly program to look at, but you cannot modify it since the attacker has no way of doing this. Instead, you must supply the original program with appropriate inputs, which hijack the program.

Here are some details about the program you must subvert. It is made up of three subroutines. A main subroutine stores an array of 15 elements on the stack and calls **read_array_backwards**. It gives **read_array_backwards** the address of the last element of the array. **read_array_backwards** reads in the numbers storing them in reverse order. It fails to check whether it has exceeded the length of the array, and consequently, this subroutine has a buffer overflow vulnerability. It returns (assuming it wasn't subverted) the address to the entry it wrote last, which will be the first entry printed. **main** then calls **print_array** giving it the address returned by **read_array_backwards**. **print_array** prints the elements of the array (which will be in reverse since they were stored in reverse), starting from where **read_array_backwards** left off.

Your job is to overflow the buffer when entering numbers in **read_array_backwards** (by entering more than 15 numbers) and corrupt some return address. The return address should point back into the buffer where you've entered a short program, which will print "zoinks!\n" to the screen and exit. Therefore, when you run the program with these inputs, the program should be interrupted in the middle of its execution, "zoinks!\n" should be printed to the screen (without the quotes), and the program should exit without printing the entered values in reverse.

You should store your attacker input in a file called **input.txt**. You can encode your values in either decimal or hexadecimal (it is often easier to work with hexadecimal) with one value per line. To send these values to the running program, you could either copy them by hand or execute the following command:

```
bash$ cat input.txt | ./sim print-reverse.out
...
```

In this case, the input values are not printed to the screen (although they are sent to the program). However, the rest of the output is printed to the screen.

You should also make a file called **input-annotated.txt**, which contains your input values along with annotations describing what each input value is needed for (if anything) and what it represents (if anything).

15.3 Error Handling

There is not much error handling you will need to worry about in this lab. As mentioned above, your greatest common divisor program should make sure the entered values are positive, repeatedly reading a value from the user until one is entered. It should also not crash on any input. There is no error handling for the second part of the lab.

15.4 Extra Credit

For extra credit, write another recursive program in Larc assembly, using a stack to save local subroutine data. For instance, you might write a recursive program for computing factorial numbers or fibonacci numbers.

15.5 Last Words

This lab will take a significant amount of time, so make sure that you start it early. Also, make sure you follow good programming practices as well as comment your code. This will be part of your grade.

Have fun!

16 Lab 9: Disassembler

In this lab, you will build a disassembler, which converts a Larc machine program into a Larc assembly program. Disassemblers have many uses such as finding the source of an error that occurs in a machine program (especially if the source code is unavailable) or inspecting some properties of the machine program such as the ASCII strings it uses.

You will either program your disassembler in Java or C. For those programming it in C, you may need to consult a C manual or book (we assume you are proficient in Java). There are many good C resources available [4, 5].

Important: Section 3 describes the Larc ISA and machine language and Section 5 describes the Larc assembly language, which you will need to make use of when building your disassembler. Make sure you read both of these sections carefully before starting this lab. You need not read the other informational sections.

16.1 Files and Directories

If programming in Java, read the text underneath the "Java" bold heading. If programming in C, read the text underneath the "C" bold heading.

Java. To get started, move to the directory **labs/lab09/java/** within the toolset's root directory. There are several existing files in this directory. These include: **asm**, a Larc assembler for assembling your programs; **sim**, a Larc simulator for running your assembled programs; and **db**, a debugger for debugging and inspecting your assembly programs. There are also two Java files: **DisAsm.java** and **Convert.class**. **DisAsm.java** contains some skeleton code for the disassembler program. You will complete this code in this assignment. **Convert.class** is a pre-compiled Java class containing methods for converting between decimal, binary, and hexadecimal number systems. The API for these classes is available online at *http://math.hws.edu/larc/api/lab09*. It is also available from the root directory in the subdirectory **api/lab09**. Finally, there is a directory **tests/**, which contains some machine programs along with the original assembly programs they were assembled from for testing your disassembler.

To compile the Java disassembler, you can use the following command on the shell:

```
bash$ javac DisAsm.java
```

To run the disassembler on the "Hello, world!" program, enter:

bash$ java DisAsm tests/hello-world.out

The disassembler is incomplete and initially it will simply print the machine program back to the screen. Once you have completed it, it should print the assembly program back to the screen. You can compare this with the assembly files in **tests/**, which contain the original assembly programs (*e.g.*, **tests/hello-world.s** for "Hello, world!"). Although they probably won't exactly match, the two assembly programs should be semantically equivalent.

C. To get started, move to the directory **labs/lab09/c/** within th toolset's root directory. There are several existing files in this directory. These include: **asm**, a Larc assembler for assembling your programs; **sim**, a Larc simulator for running your assembled programs; and **db**, a debugger for debugging and inspecting your assembly programs. There are also 3 C files in this directory: **disasm.c**, **convert.o**, and **convert.h**. **disasm.c** contains some skeleton code for the disassembler program. You will complete this code in this assignment. **convert.o** is a pre-compiled object file containing functions for converting between decimal (represented as ints) and binary (represented as strings). The header file **convert.h** provides the API for these functions. Finally, there is a directory **tests/**, which contains some machine programs along with the original assembly programs they were assembled from for testing your disassembler.

To compile the disassembler in C, you will use the free **gcc** compiler. You can use the following command:

bash$ gcc -o disasm disasm.c convert.o

This creates an executable file called **disasm**, which can then be run. The command below runs the disassembler on the "Hello, world!" test program:

bash$ ./disasm tests/hello-world.out

The disassembler is incomplete and initially it will simply print the machine program back to the screen. Once you have completed it, it should print the assembly program back to the screen. You can compare this with the assembly files in **tests/**, which contain the original assembly programs (*e.g.*, **tests/hello-world.s** for "Hello, world!"). Although they probably won't exactly match, the two assembly programs should be semantically equivalent.

16.2 Lab Details

A disassembler converts a machine program into an assembly program. In general, it is not possible to perfectly perform this conversion. Assume a machine program contains the following word:

0000000100100011

This word could be interpreted in a number of ways. For example, it could refer to the following assembly instruction:

add $1 $2 $3

It could also refer to the signed decimal value 291. It might also be a character in a string although in that case we would expect the first 8 bits to be 0. Unfortunately, it is not always possible to tell how a word in a program will be used. Worse, some programs could use a word in multiple ways (*e.g.*, as an instruction and a data value). Without running the program, which the disassembler cannot do, it can be impossible to know. Therefore, there is no such thing as the perfect disassembler.

Fortunately, most programs conform to a set of rules that make it possible to disassemble them. For instance, as discussed in Section 3, it is recommended that all Larc instructions preceed the Larc data elements. In addition, as mentioned in Section 4, it is recommended that a marker of all 1's be used to specify the end of the instructions. This marker lets the debugger know where the instructions end. We will assume these conventions when building our disassembler.

The data elements are more difficult to translate. For example, how do you know if a data word should be its own word, or if it is part of an ASCII string or space directive? In general, this is impossible to determine but by making some careful assumptions, you can correctly disassemble most data elements. Of course, you will need to look closely at the instructions and how they make use of particular words in the data section, in order to do this translation.

Your task is to write a (Java or C) program that correctly disassembles conventional Larc machine programs (*e.g.*, those that have been assembled from the reference assembler). The resulting assembly program should always work. In other words, if you re-assemble it, then the translated machine code should exactly match the original machine code. In addition, given the assumptions above, your disassembler should correctly translate all instructions in the program into the appropriate assembly instruction. You can assume that no extended instructions were used. For base credit, each word in the data section should be translated into the corresponding word directive.

To simplify your disassembler, you can assume that Larc machine programs are ASCII files

made up of binary words. You do not have to support Larc programs, which use hexadecimal words. Moreover, you can assume that the Larc input machine program will not include comments.

16.3 Error Handling

Your disassembler will need to catch any syntax errors in the input Larc machine program. You can assume that the machine program is not commented and that is uses only the binary (and not hexadecimal) format. Therefore, each line should contain exactly 16 0's and 1's. If a syntax error is found, your disassembler should print an appropriate error message and exit.

Furthermore, your disassembler should not crash on any input file, whether it is syntactically correct or not. For syntactically correct programs, your disassembler should always produce an assembly file that is semantically equivalent to the machine program. In other words, if the generated assembly file is re-assembled and that machine code is run, it should behaviorally function the same as the original machine code. The generated assembly code may look different then the original assembly code (*e.g.*, they may use different data directives, one may use a set of built-in instructions while the other uses the corresponding extended instruction), but they should still be semantically equivalent.

16.4 Extra Credit

For extra credit, you can improve upon your disassembly technique. For example, you could attempt to discover ASCII strings or space directives in the machine program given a conventional program. You could also try to find extended assembly instructions in the text section. The closer the generated assembly code looks to the original assembly code, the more extra credit you will receive. Of course, the generated assembly code must still be semantically equivalent to the original machine program.

16.5 Last Words

This lab will take a significant amount of time, so make sure that you start it early. Also, make sure you follow good programming practices as well as comment your code. This will be part of your grade.

Have fun!

17 Lab 10: Simple C Compiler

In this lab you will complete a compiler for a simple C-like language (which we will call "simple C") . Your compiler will convert simple C programs into Larc assembly code, which can then be assembled and run, using the assembler and simulator you wrote in earlier labs. You will either program your simple C compiler in Java or (ANSI) C. For those programming it in C, you may need to consult a C manual book (we assume you are proficient in Java). There are many good C resources available [4, 5].

Simple C is a tiny subset of C (we will use "ANSI C" in this section to refer to full C as this is the name of the standard of C used by nearly all C compilers). Simple C does not include many features such as functions, loops, *etc.* Furthermore, parts of the compiler are given to you such as the parser, which reads in the program file, discovers its syntactic structure, and produces an internal representation of the source program. Still, this will give you a good glimpse into how compilers work. For more details, however, you will need to take a full course on compilers.

Important: Section 5 describes the Larc assembly language, the target language of your compiler. Make sure you read this section carefully before starting this lab. Before reading Section 5 you should also read Section 3, which describes the Larc ISA. You need not read the other informational sections.

17.1 Files and Directories

If programming in Java, read the text underneath the "Java" bold heading. If programming in C, read the text underneath the "C" bold heading.

Java. To get started, move to the directory **labs/lab10/java/** within the toolset's root directory. There are several files in this directory. These include: **asm**, a Larc assembler for assembling your programs; **sim**, a Larc simulator for running your assembled programs; and **db**, a debugger for debugging and inspecting your assembly programs. There are also 6 Java files in this directory: **SCC.java**, **Parser.java**, **Lexer.java**, **sym.java**, **Inst.java**, and **InsnList.java**. **SCC.java** contains some skeleton code for the simple C compiler. You will complete this class in this assignment. (This is the only file you should need to edit in this lab.) **Parser.java**, **Lexer.java**, and **sym.java** are classes for scanning and parsing (discovering and checking the syntactic structure of a program) the source simple C program. These are working classes and should not be edited. The last two files, **Inst.java**

and **InsnList.java**, you will need to use but you should not need to modify. **Inst.java** contains an intermediate representation of an instruction. Each statement in the C program is converted into an **Inst** by the compiler. **InsnList** represents a list of instructions. The API for these classes is available online at *http://math.hws.edu/larc/api/lab10*. It is also available from the root directory in the subdirectory **api/lab10**. The directory **tests/** contains some simple C programs, which you can use to test your compiler.

To compile your simple C compiler (which, interestingly enough, is also a compiler), you can use the following command on the shell:

```
bash$ javac SCC.java
```

This will automatically compile the other classes as well. To run the simple C compiler use:

```
bash$ java SCC tests/hello-world.c
...
```

where **tests/hello-world.c** is a test program that prints "Hello, world!" to the screen. When your compiler is working this will print a Larc assembly file to the screen. (It currently just prints the simple C program to the screen.) You can then use the reference assembler and simulator to assemble and run the program, respectively.

C. To get started, move to the directory **labs/lab10/c/** within the toolset's root directory. There are several files in this directory. These include: **asm**, a Larc assembler for assembling your programs; **sim**, a Larc simulator for running your assembled programs; and **db**, a debugger for debugging and inspecting your assembly programs. There are also 4 C files in this directory: **scc.c**, **util.c**, **parser.c**, and **lexer.c**. **scc.c** contains some skeleton code for the simple C compiler. You will complete this file in this assignment. (This is the only file you should need to edit in this lab.) **util.c** contains some utility functions that you will find useful when completing **scc.c**. **lexer.c** and **parser.c** are files for scanning and parsing (discovering and checking the syntactic structure of a program) the source simple C program. These are working modules and should not be edited.

In addition, there are some header files you should make use of. **scc.h** contains several definitions needed in your implementation **scc.c**. **util.h** provides the API for using the **util** module. Finally, **parser.h** is a header file for the parser although you should not need to use this.

The last item in the lab directory is a subdirectory called **tests/**. **tests/** contains several simple C programs for testing the implementation of your compiler.

To compile your simple C compiler (which, interestingly enough, is also a compiler), you will use the C compiler **gcc**. You can use the following command:

```
bash$ gcc -o scc scc.c parser.c lexer.c util.c
```

This command creates an executable, **scc**, which we can run as follows:

```
bash$ ./scc tests/hello-world.out
...
```

where **tests/hello-world.c** is a test program that prints "Hello, world!" to the screen. When your compiler is working this will print a Larc assembly file to the screen. (It currently just prints the simple C program to the screen.) You can then use the reference assembler and simulator to assemble and run the program, respectively.

17.2 Lab Details

Your simple C compiler will translate simple C programs into Larc assembly programs, which can then be assembled with a Larc assembler, and run with a Larc simulator. It will work similarly to the Larc assembler you wrote in Section 12. You might even find it useful to relook at the assembler lab when doing this lab.

Simple C language. Simple C is a small subset of the ANSI C language. Many of the C features have been removed including functions, loops, and arrays. It includes statements for performing simple integer operations as well as a primitive form of control flow.

Table 12(a) shows the supported simple C statements. A simple C programmer can define labels (as in assembly language) and write simple if statements to branch to them, via a **goto**, if a source variable or integer is not 0. (Note: both **goto** and labels should generally be avoided when programming in ANSI C.) A programmer can print an int variable or constant to the screen, read an int from the user and store in a variable, and print a string constant to the screen. A programmer can move a variable or integer into another variable. A programmer can perform a unary operation on a single variable or integer constant, and put the result in a variable. Finally, a programmer can perform a binary operation on two variable or integer constant operands, and store the result in a variable. Table 12(b) shows the unary and binary operations that are supported.

As in ANSI C, simple C programmers can comment their programs, as well as use white space to make their programs more readable.

Figure 18 shows an example simple C program called **test.c**, which reads in a number and prints whether it is odd or even.

Here is a sample run after compiling, assembling, and running this program with user input

Type	Format	Example
Label	<**label**>**:**	**foo:**
Conditional	**if (**<**src**>**) goto** <**label**>**;**	**if (x) goto foo;**
Print int	**print_int(**<**src**>**);**	**print_int(x);**
Read int	<**var**>**= read_int();**	**n = read_int();**
Print string	**print_string(**<**string**>**);**	**print_str("foo");**
Move	<**var**>**=** <**src**>**;**	**x = y;**
Unary	<**var**>**=** <**op**> <**src**>**;**	**x = -3;**
Binary	<**var**>**=** <**src1**> <**op**> <**src2**>**;**	**x = 3 + y;**

(a)

Type	Operation	Example
Unary	Negation	**x = - 3;**
Unary	Bitwise NOT	**x =** $\sim$ **y;**
Binary	Addition	**x = 3 + 4;**
Binary	Subtraction	**x = y - 5;**
Binary	Multiplication	**x = 2 * y;**
Binary	Division	**x = y / z;**
Binary	Modulus	**x = x % 2;**
Binary	Bitwise AND	**x = x & y;**
Binary	Bitwise OR	**x = x** $\mid$ **y;**
Binary	Bitwise XOR	**x = x** $\hat{}$ **y;**
Binary	Equals	**x = y == z;**
Binary	Not equals	**x = y != z;**
Binary	Less than	**x = y** $<$ **z;**
Binary	Less than or equal to	**x = y <= z;**
Binary	Greater than	**x = y** $>$ **z;**
Binary	Greater than or equal to	**x = y >= z;**

(b)

Table 12: Simple C statements (a) and unary/binary operations (b).

```
/* Program for testing whether
   a number is odd or even */
print_str("Enter a number: ");
n = read_int();
is_odd = n % 2;
if (is_odd)
    goto odd;
// even case:
print_str("Your number is even.\n");
if (1)
    goto end;
// odd case:
odd:
print_str("Your number is odd.\n");
end:
```

Figure 18: Simple C program for testing whether a number is odd or even.

underlined:

```
Enter a number: 5
Your number is odd.
```

Those already familiar with C will notice that a simple C program is not a compilable subset of ANSI C. That is to say, the program from Figure 18 will not compile with an ANSI C compiler such as **gcc**. But simple C code is legal ANSI C code and if other ANSI C code is added around it,

```
// added ANSI C code
#include <stdio.h>
void print_str(char * str) { printf("%s", str); }
void print_int(int n) { printf("%d", n); }
int read_int() { int n; scanf("%d", &n); return n; }
int main() {
    int n, is_odd;
    // start of simple C code
    print_str("Enter a number: ");
    n = read_int();
    is_odd = n % 2;
    if (is_odd) goto odd;
    print_str("Your number is even.\n");
    if (1) goto end;
    odd:
    print_str("Your number is odd.\n");
    end:
    // end of simple C code
    return 0;
}
```

Figure 19: ANSI C version of the simple C odd or even program.

it can be made to compile with an ANSI C compiler. For instance, in ANSI C, which has support for functions, all statements must go in a function and at the very least a main function must be defined. Therefore, to convert a simple C program to an ANSI C program, we would need to put the original simple C code in a main function, which returns 0 at the end. At the beginning of the main function, we would need to declare all the used variables, as integers, since this is the only supported variable type in simple C. Finally, we would need to define the **print_str**, **print_int**, and **read_int** functions.

Figure 19 shows the simple C program from Figure 18 as an ANSI C program. This program can be compiled with **gcc** and run natively on your machine.

Simple C compiler. The simple C compiler takes as input a simple C program and generates Larc assembly code, which it prints to the screen (normally this would be sent to a file). The Larc assembly code can use any features of the Larc assembly language including extended features so long as the resulting program assembles properly. On a buggy simple C program, the compiler should print an error message and exit.

As building an entire compiler is a semester-long task, you have been given some code to get you started. In particular, you have been given a working parser, which will discover the syntactic structure of the input program and check for any syntactic errors. You should not need to modify any of the parser code.

In addition, the internal representation of each statement has been defined for you. To see this representation, look in **Stmt.java** if you are writing your compiler in Java or **scc.h** if you are writing your compiler in C. The parser will produce a list of statements. You will need to loop over this list, probably a few times, in order to translate each statement into Larc assembly code. Note: each statement might translate into several assembly instructions.

Initially, your compiler will loop over the list of statements and print each one out as a simple C statement (see **SCC.java** if you are using Java or **scc.c** if you are using C). You will need to modify this code so that it translates it to assembly. The skeleton code should help show how to use statements and statement lists.

Although you are welcome to decide how you want the translation to work, we describe one approach. Recall that a Larc assembly program consists of a data and text section. In the data section, you might put all the C variables. You could put a word for each one of them (initially 0) with a label for accessing each. In addition, you could put an ASCII data directive for each string constant used in a print string statement along with a label for accessing that string.

In the text section, you would put the translated assembly instructions for each simple C statement. To get the value for a source variable in a statement you can do a memory load word. The label marking the variable word in the data section can be used for the memory address. Setting a variable would be implemented as a memory store word. At the end of the text section, you will add some Larc assembly instructions for halting and terminating the program.

17.3 Error Handling

In this assignment (and this assignment only) error checking is left as extra credit (below). However, your compiler should translate any correct simple C program into the equivalent Larc assembly code. It should also never crash on a working or broken simple C program.

17.4 Extra Credit

The parser takes care of most of the error handling but there are a couple of potential semantic errors that you should catch:

- Variable used before it is assigned to.
- Duplicate label definitions.

- Control transfer to an undefined label.

17.5 Last Words

This lab will take a significant amount of time, so make sure that you start it early. Also, make sure you follow good programming practices as well as comment your code. This will be part of your grade.

Have fun!

18 Closing Remarks

Hopefully if you have reached this point then you have successfully completed all of the labs. If so, congratulations! You now are ready to go on and explore in more depth some of the areas that were touched on in this manual such as microarchitecture, compilers, and operating systems.

We hope that you have enjoyed the Larc labs and found this manual helpful and informative. Please let us know if you found any bugs or errors along the way, in either the Larc toolset or in this manual. In addition, please contact us if you have questions about any part of this work or if you have requests for extensions. Finally, we welcome contributions from others, so please let us know if you would like to contribute in some way. To contact us you can send email to *corliss@hws.edu*.

Thanks,
Marc Corliss

References

[1] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, 4th edition, 2005.

[2] HP Technologies. *Alpha Architecture Reference Manual*. Digital Press, 3rd edition, 1998.

[3] Gerry Kane and Joseph Heinrich. *MIPS RISC Architecture*. Prentice Hall, 2nd edition, 1991.

[4] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice-Hall, Inc., 2nd edition, 1988.

[5] K. N. King. *C programming: a modern approach*. W. W. Norton and Company, 2008.

[6] Yale N. Patt and Sanjay J. Patel. *Introduction to Computing Systems: From Bits and Gates to C and Beyond*. McGraw Hill, 2nd edition, 2004.

[7] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Elsevier, 3rd edition, 2005.

[8] William Stallings. *Computer Organization and Architecture: Designing for Performance*. Pearson, 7th edition, 2006.

[9] Andrew S. Tanenbaum. *Modern Operating Systems*. Pearson Prentice Hall, 3rd edition, 2008.

[10] Steven R. Vegdahl. MipsPilot: A compiler-oriented MIPS simulator. *Journal of Computing Sciences in Colleges Northwest Conference*, 24(2):32–39, 2008.