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

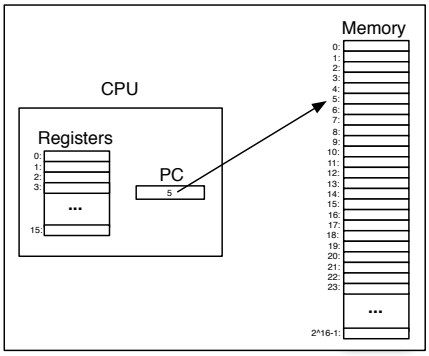

Figure 1: The basic components of a Larc machine.

(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.

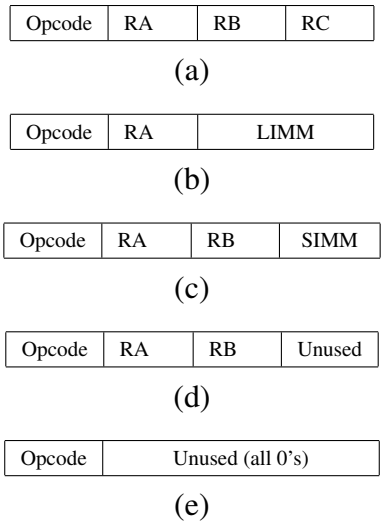

Figure 3: The encoding of Larc instructions: (a) ALU, (b) long immediate, (c) short immediate, (d) jumps, and (e) system calls.

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

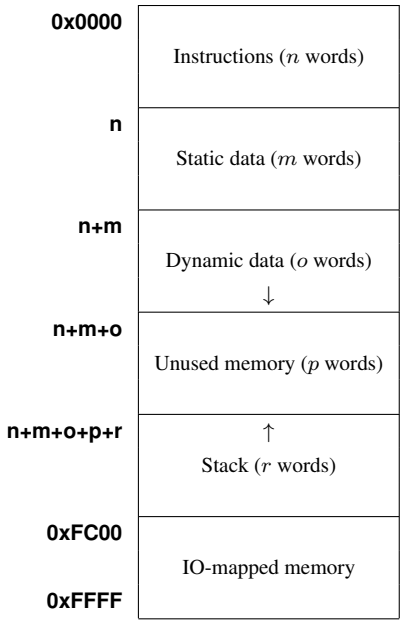

Figure 4: The general memory layout of a Larc program.

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.

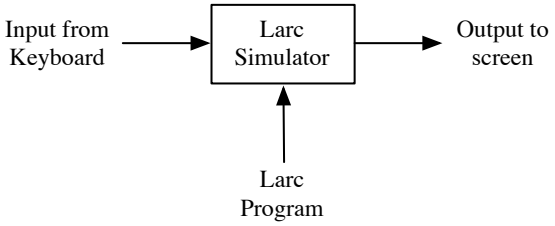

Figure 6: A Larc simulator.

```
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").