

## 5 Larc 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 5.3 and 5.4 describe Larc assembly instructions and datum, respectively. Finally, Section 5.5 describes how to write Larc assembly programs.

### 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` (assembly instructions are described in more detail in Section 5.3). The same is true for data used within an assembly program. For example, the integer 5 can be put in an assembly program by inserting the data directive `.word 5` as opposed to inserting `0000000000000101`. Therefore, 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, Larc assembly language includes one form of commenting, a single-line comment that start with `#`. The `#` is ignored as well as all subsequent characters until a newline is encountered. Therefore, programmers can use `#` to document their programs, which is extremely helpful, especially when programming at such a low level.

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 the assembler will resolve when it translates the assembly program into a machine program. Another advantage is

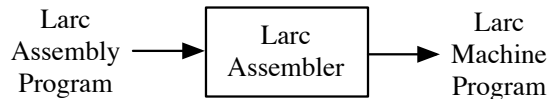


Figure 11: A Larc assembler.

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 assembler converts the label into the appropriate PC-relative, 8-bit immediate.

Larc assembly language also has built-in support for handling 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 byte. On the other hand, an assembly language programmer can simply use the Larc `ascii` data directive to insert a string. For example, the string “abc” can be put in an assembly program by inserting the directive `.ascii “abc”` as opposed to inserting the following (data within an assembly program is described in more detail in Section 5.4):

```
0110000101100010  
0110001100000000
```

The assembly programmer does not have to know the mapping of each character to its ASCII encoding nor do they have to worry about terminating the string with a null byte. The assembly programmer can also think of the `ascii` string as a single datum rather having to expand it into several words.

A final difference between Larc assembly language and Larc machine language is that the assembly language includes some additional 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 several additional types of branches that are not built-in to the machine language (*e.g.*, branch greater than zero). These assembly instructions map to several machine instructions.

## 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 11 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 immediate. 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. The ascii directives may have to 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 5.3) then the assembler will need to do one more pass to convert each extended instruction into several base instructions. Probably the best approach for dealing with extended instructions is to expand them into base instructions before doing the other passes described above.

**Error handling.** The assembler must also find any errors in the Larc assembly program. These include syntax errors, such as misspelling an operator, as well as semantic errors, such as referencing a non-existent label. Here is a list of possible errors that must be handled by the assembler:

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.*, an immediate in an ALU instruction). Note: this error will also occur if the programmer tries to place a data directive in the text section.

- Unknown data directive.

If the programmer uses an unrecognized data directive (*e.g.*, `‘.extern’`).

- Improper data directive format.

If the programmer incorrectly formats a data 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.

- Multiply-defined text section.

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

- Undefined data section.

If the programmer forgets to define the data section of the assembly program. This section is required in an assembly program although it could be empty.

- 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.

- Usage of non-existent label.

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).

- Unknown or restricted register identifier.

If an instruction uses a non-existent register identifier (*e.g.*, ‘\$17’) or a restricted register (*e.g.*, ‘\$15’ – see below for a discussion of restricted registers).

- Immediate too large.

If an instruction uses an immediate that will not fit into the allotted 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.

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 Larc Assembly Instructions

A Larc assembly instruction is written textually rather than in binary. The components of an 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 mnemonic name. For example, the stack pointer register, register 10, can be written as \$sp or \$10. Table 6 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 mnemonic name.

There is one important difference between the use of registers in assembly language versus machine language. Registers 11 and 12 (\$at1 and \$at2) 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 preceded by a “\$” (“\$” differentiates

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	\$s0	saved temporary register
\$7	\$s1	saved temporary register
\$8	\$s2	saved temporary register
\$9	\$ra	return address register
\$10	\$sp	stack pointer register
\$11	\$at1	assembler register
\$12	\$at2	assembler register
\$13	\$k0	OS register
\$14	\$k1	OS register
\$15	\$psr	OS register (processor status register)

Table 6: Larc assembly registers.

between a register and an immediate). They must be able to fit into 16 bits using twos complement encoding and in some cases an even smaller number of bits (*e.g.*, when used in an instruction).

**Instructions.** Figure 7 shows the base assembly instructions. Each instruction corresponds directly to a 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 can 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. In the case of branches, this changes the semantics somewhat. The branch now refers to what looks like an absolute target rather than a relative target. However, the assembler will convert the label into an instruction with an immediate, which uses relative addressing. Immediate values are still supported in both branch and load (lower or upper) immediate instructions. In this case, the immediate works as it does in machine language (*i.e.*, branch immediate uses relative addressing).

Figure 8 shows the extended assembly instructions. These can be implemented by the student for extra credit when building the assembler. They must be translated by the assembler into several base assembly instructions.

Type	Operation	Format	Semantics
<b>ALU</b>	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 AND	and \$a \$b \$c	Reg[a] = Reg[b] & Reg[c]
	bitwise NOR	nor \$a \$b \$c	Reg[a] = ~(Reg[b]   Reg[c])
<b>Long immediate</b>	load immediate	li \$a limm	Reg[a]=limm
	load address	la \$a label	Reg[a] = addr(target)
	load upper immediate	lui \$a limm	Reg[a] = limm << 8
	branch equal to 0	beqz \$a target	if (Reg[a] == 0) PC=addr(target)-1
		beqz \$a limm	if (Reg[a] == 0) PC=PC+limm
branch less than 0	bltz \$a target	if (Reg[a] < 0) PC=addr(target)-1	
	bltz \$a limm	if (Reg[a] < 0) PC=PC+limm	
<b>Short immediate</b>	memory load	lw \$a simm(\$b)	Reg[a] = Mem[Reg[b]+simm]
	memory store	sw \$a simm(\$b)	Mem[Reg[b]+simm] = Reg[a]
<b>Jump</b>	jump and link register	jalr \$a \$b	old_pc=PC, PC=Reg[b], Reg[a]=old_pc+1
<b>System call</b>	system call	syscall	perform system call (type in Reg[1])

Table 7: Larc base assembly instructions. \$a, \$b, and \$c are register identifiers (\$0-\$15). limm and simm refer to a long immediate and a short immediate, respectively. Reg[] refers to the registers (which must be indexed using a register identifier) and Mem[] refers to memory (which must be indexed with an address). PC refers to the program counter. addr() is a function that converts a target label into a numeric address. The operators in the semantics column are expressed using C syntax. Note: the PC is incremented after each instruction executes except the jump and link instruction.

## 5.4 Larc Assembly Data

Larc supports three types of data in an assembly program: a single word, several null words, or an ASCII string. These are specified using data directives.

**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

Type	Operation	Format	Semantics
<b>Move</b>	move	move \$a \$b	Reg[a] = Reg[b]
<b>ALU</b>	modulus	rem \$a \$b \$c	Reg[a] = Reg[b] % Reg[c]
	bitwise OR	or \$a \$b \$c	Reg[a] = Reg[b]   Reg[c]
	bitwise XOR	xor \$a \$b \$c	Reg[a] = Reg[b] ^ Reg[c]
<b>Branch</b>	branch not equal to 0	bnez \$a target bnez \$a limm	if (Reg[a] != 0) PC=addr(target)-1 if (Reg[a] != 0) PC = PC+limm
	branch greater than 0	bgtz \$a target bgtz \$a limm	if (Reg[a] > 0) PC=addr(target)-1 if (Reg[a] > 0) PC=PC+limm
	branch less than or equal to 0	blez \$a target blez \$a limm	if (Reg[a] <= 0) PC=addr(target)-1 if (Reg[a] <= 0) PC=PC+limm
	branch greater than or equal to 0	bgez \$a target bgez \$a limm	if (Reg[a] >= 0) PC=addr(target)-1 if (Reg[a] >= 0) PC=PC+limm
	branch equal to (binary)	beq \$a \$b target beq \$a \$b limm	if (Reg[a] == Reg[b]) PC=addr(target)-1 if (Reg[a] == Reg[b]) PC=PC+limm
	branch not equal to (binary)	bne \$a \$b target bne \$a \$b limm	if (Reg[a] != Reg[b]) PC=addr(target)-1 if (Reg[a] != Reg[b]) PC=PC+limm
	branch less than (binary)	blt \$a \$b target blt \$a \$b limm	if (Reg[a] < Reg[b]) PC=addr(target)-1 if (Reg[a] < Reg[b]) PC=PC+limm
	branch less than or equal to (binary)	ble \$a \$b target ble \$a \$b limm	if (Reg[a] <= Reg[b]) PC=addr(target)-1 if (Reg[a] <= Reg[b]) PC=PC+limm
	branch greater than (binary)	bgt \$a \$b target bgt \$a \$b limm	if (Reg[a] > Reg[b]) PC=addr(target)-1 if (Reg[a] > Reg[b]) PC=PC+limm
	branch greater than or equal to (binary)	bge \$a \$b target bge \$a \$b limm	if (Reg[a] >= Reg[b]) PC=addr(target)-1 if (Reg[a] >= Reg[b]) PC=PC+limm
<b>Jump</b>	branch unconditional	b target b limm	PC=addr(target)-1 PC=PC+limm
	jump (no linking)	jr \$a	PC=Reg[a]
	subroutine call	jal target	reg[9]=PC+1, PC=addr(target)

Table 8: Larc extended assembly instructions. \$a, \$b, and \$c are register identifiers (\$0-\$15). limm refers to a long immediate. Reg[] refers to the registers (which must be indexed using a register identifier). PC refers to the program counter. addr() is a function that converts a target label into a numeric address. The operators in the semantics column are expressed using C syntax. Note: the PC is incremented after each instruction executes except the jump instructions.

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.

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

```
.ascii "abc"
```

The assembler translates this directive into the following:

```
0110000101100010
```

```
0110001100000000
```

Note: as with the space directive, the assembler may need to insert multiple words (in this case two) into the machine program rather than just one. The number of inserted words is equal to one more than the length of the string divided by two. A null byte is put at the end of the string. If the string length is even then two null bytes are inserted to preserve word alignment.

## 5.5 Larc 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 directives. The text section is preceded by the directive “.text” and the data section is preceded by the directive “.data”. 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 0 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

# exit program with status 0
li $1 0
li $2 0
syscall

```

Figure 12: 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 12 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).