

Strings in the SPARC

By Tom S. Lee

Introduction

The aim of this tutorial is to explore the use of strings in SPARC. We will look at how to declare a string, how to look at individual characters within a string, and how to print a string to the console output in tkisem.

The lab manual illustrates the use and printing strings in lab 10. However, the point of the lab was not to only discuss the use of strings, so the explanation of what to do with strings is not as detailed as could be. This tutorial will pick up where lab 10 left off concerning strings.

Strings

Strings are defined in SPARC like this:

```
string_name:      .asciz  "This is a string!"
```

Strings are essentially arrays of ASCII characters. They are indexed in the same way strings are in C and C++. Therefore, accessing each character in a string is easy. A pointer can be used to...well...point to each character in the string. Traversing the string is just like traversing an array or vector. The value of the pointer corresponds to the position in the string.

SPARC has a neat little feature with its .asciz strings: they are NULL terminated. This means that there is a NULL character at the end of the string, which can be easily identified. Having the NULL terminating the string makes it simple to sequentially read the characters of the string, and then easily stop at the end every time without pointing to a location off the string. More on this subject matter later.

Initializing and Setup

Now that we have defined the ASCII string, let's initialize it and start having fun.

First, we need to set the string to a register, so we can have fun with it:

```
set    string_name, %r2
```

In this example, we place the string into Register 2. From here on out, %r2 will serve as the pointer to each character in the string. %r2 holds the address of the current character of the string. In this first step, it holds the address to the first character of the string.

To load individual characters into a register, we do the following:

```
ldub  [%r2], %rx
```

where %rx is any register. The operation ldub stands for “load unsigned bit.” This will load the ASCII value of the character pointed to by %r2. Remember, we are loading the contents of the address pointed to by %r2.

The destination register %rx now holds the ASCII value that corresponds to the character of the string. In our example, if we were to ldub the first character in our string, we would have the following:

```
“This is a string!”    (our string)
%rx = 0x54             (%rx holds a value of 0x54, which is ASCII ‘T’)
```

To move to the next character in the string, we need to move the pointer. This is accomplished by simply incrementing the pointer:

```
inc    %r2
```

This moves %r2 to the address of the next character in the string. If we do another ldub into %rx, %rx will now hold the ASCII value of that character.

Let’s look at an example of this:

```
Original string:    “This is a string!”

ldub  [%r2], %rx    %rx = 0x54          (%rx holds ASCII value for ‘T’)
inc   %r2
ldub  [%r2], %rx    %rx = 0x68          (%rx now holds ASCII value for ‘h’)
inc   %r2
ldub  [%r2], %rx    %rx = 0x69          (%rx holds ASCII value for ‘i’)
inc   %r2
ldub  [%r2], %rx    %rx = 0x73          (%rx now holds ASCII value for ‘s’)
inc   %r2
ldub  [%r2], %rx    %rx = 0x20          (%rx holds ASCII value for SPACE)
inc   %r2
.....
```

And so on. By incrementing the pointer each time, we are simply moving down the array, picking up each character as we go. Then, the corresponding ASCII value of the character in the address pointed to by %r2 gets loaded into %rx. With the ASCII value of the character in %rx, we are free to do as we wish with that character.

Notice a neat thing about this code? It is simply two lines of code repeated over and over again. Do I smell a loop coming on? The ease and simplicity of this code means that it is very loop-friendly. All you have to do is write a few lines of code, and then have a branching command that will perform these actions a certain number of times to achieve a desired result.

Eventually, in our example, we will come to the end of the string. But how do we know that we reached the end of the string? The computer does not know that the “!” at the end of our string is a type of punctuation mark, and must therefore stop reading characters. Keep in mind that these are null-terminated strings. There is a NULL character at the end of the string, which signifies that the string has no more characters past that point.

If we continued to loading characters from the string into %rx, we would end up with the following:

```
..... (continued from above example)
ldub [%r2], %rx    %rx = 0x21      (%rx holds ASCII value for '!')
inc    %r2
ldub [%r2], %rx    %rx = 0x0       (%rx now holds a NULL)
inc    %r2
```

When we have reached the NULL character, it means that we have reached the end of the string. Since a NULL character cannot appear anywhere else in the string, we can safely check for a NULL character within the string, and use the presence of a NULL character as a terminating condition for our character reading and loading procedures.

Applied Strings: What To Do Now?

We now know how to load the characters of a string into a register. Now what could we possibly do with this knowledge?

String Comparisons

There are those times that it is necessary to compare two strings, to determine which of the two strings is greater. C and C++ have string functions built-in to their STLs. However, SPARC does not. This means that manual coding is necessary in order to implement this kind of functionality.

Since we know how to extract each individual character from a string, we can use this principle as a basis of implementing string comparisons. When we extract each character in a string, we are placing the ASCII value of that character into a destination register. Keep this in mind. Think on this for a while. Each character, when loaded into a register, gets stored as some hexadecimal value, equal to the ASCII value of that character.

I will not go any further about how to implement string comparisons since this is an exercise in lab 10. So don't get too happy, I am not going to do your work for you. Darn. But, what I will mention is the fact that you can use the extraction capability I have described here to extract each character in the strings, and then compare each character of each string, based on its ASCII value. The comparison of ASCII values is the main principle to keep in mind when implementing this functionality. Extract, load, compare, go from there...

Printing Strings to Console Output

One of the most useful applications we can use for this is printing messages to the console output in tkisem.

Recall that we use %r8 to input and output characters in tkisem. The “ta” directives are used to accomplish these tasks. For our example here, “ta 1” will print the contents of %r8 to console output.

NOTE: The “ta 1” directive will print out the ASCII representation of the value held in %r8. This means that you can’t simply just throw something in %r8 and make it print out exactly how you want it to. You **MUST** break up your intended output into its constituent ASCII characters.

An example:

Let’s say I wanted to print out “100” to the console output.

I can’t just set %r8 to be equal to 100, and then do a “ta 1.” I would find that “@” would have been printed to the console. “@” is ASCII 100 (or 0x40).

If I wanted to *correctly* print “100” to the console output, I would have to break up “100” into its constituent characters, set %r8 to be equal to the ASCII value of each of those characters, and then do a “ta 1.”

“100”

```
set    0x31, %r8      ! '1' is ASCII 0x31
ta     1
set    0x30, %r8      ! '0' is ASCII 0x30
ta     1
set    0x30, %r8
ta     1
```

This is a HUGE hint to those wanting implement any of the number printing functions (especially for, say, a SPARC calculator). It is necessary to break up each and every single number into its constituent digits, convert those digits to ASCII, and then print them out via %r8.

This is the one of the main differences between strings in SPARC and strings in C and C++: you cannot print out the string in just one line of code, using one command. You have to systematically place each character of the string into %r8 as its appropriate ASCII value, and then print each individual character from there.

But do not fret! It is still easy to accomplish this task, if you are using ASCII strings. Recall from above how we loaded each character of the string into a register. The

destination register that holds each character holds the ASCII value that corresponds to that character. Since we need to pass ASCII values to %r8 for printing, we are all set!

Let's write a little snippet of code that prints out a null-terminated string. This code is very useful in that it is "cookie-cutter code;" you can use this code whenever and wherever you need to print a string. I have used this code many times in the course of my SPARC coding. It is pretty much the same code as what is presented in lab 10 of the SPARC lab manual. I have beefed up the comments to make it more clear as to what is going on in the code.

```
print_string:
    set    string_name, %r2    ! load our string into %r2, %r2 is our pointer
    ldub  [%r2], %r8          ! load character into %r8
    cmp   %r8, 0              ! check for null
    be    end                  ! if null, we are done
    nop                               ! delay slot
    ta    1                    ! print char
    ba    print_string        ! loop back up for next char
    inc   %r2                  ! increment the char pointer (delay slot)

end:      ta    0              ! terminate
```

This code demonstrates the usefulness of the null-terminated string. We need to have a way of knowing when we have read all the characters in the string. The comparison we perform on line 3 of the code checks for the end of the string. The NULL character is 0 (or ASCII 0, if you want to get more detailed). Once we have reached the end of the string, we are done printing. Sounds obvious enough, but it needs to be emphasized here to make it clear what exactly we are talking about.

This code is simple, small, and effective. If you want to have a quick debugging technique in your programs that does not involve keeping track of constantly changing registers ("Did %r2 get set to 1, or is it still 0? Crud I stepped through the program too quickly, got to start over again. Wait, %r3 just got reset, what happened here????"), you can print out appropriate error messages to your console output. Have a different ASCII string for different cases in your program, and with each case, you can print out the appropriate error message. Nifty huh?

Closing Comments

We have discussed how strings are handled in SPARC, how to manipulate and navigate strings, and how to make use of the string in SPARC. This tutorial is by no means a replacement for the information provided in the official SPARC lab manual, but serves as a supplement to that material. Hopefully the information presented here reinforces all of the ideas originally introduced in the SPARC lab manual. With that said, happy coding!