

Project CHARS

Out: 16 Oct 2009

Due: 23 Oct 2009, 11:59pm

0 Description

For this assignment, you will implement two short programs. The first program, `i2s`, will convert a MIPS integer to a null-terminated string representing the same integer. The second program, `s2i`, will convert a null-terminated string representing an integer to the MIPS integer that it represents. For both programs you only need to handle positive numbers and can assume an input string with a `'-'` in it is an error.

Examples

`i2s`

- `8675309` \rightarrow `"8675309"`
- `0` \rightarrow `"0"`

`s2i`

- `"8675309"` \rightarrow `8675309`
- `"0"` \rightarrow `0`

After converting, your programs should print the result using the appropriate syscall (`i2s` should use `print_string`, `s2i` should use `print_int`).

In `s2i`, the ASCII string, and its length, will be coded in the data section of your program (your program does not have to determine the length of the string). `s2i` does not have to handle negative numbers; only positive integers are valid inputs. However, it should error-check for invalid characters (anything other than digits) and terminate with an error message if any are found.

In i2s, the integer will be stored as a word in the data section of your program. You should also declare a buffer for the output string in the data section. Think about how big the buffer needs to be to hold a positive 32-bit 2's complement integer in radix-10.

1 Implementation

You need to implement this program in MIPS assembly language (MAL) using mipscope. You'll write two short programs and hand in a separate file for each one.

String to Integer (s2i)

As mentioned above, the string to convert will be encoded in the data section, looking something like this:

```
thestring:  .asciiz  "54381"      #the string to convert
```

Then you will have a variable to store the integer value of the string once you've converted it:

```
theint:    .word    0            #int val of the string
```

You can assume that the number will fit into a single 32 bit word.

Pseudocode for the conversion:

```
print string with print_str syscall
chars:=address of char array
theint:=0 (this will hold our converted integer)
curr:=0 (this holds index of current character)
WHILE NOT (chars[curr] = NULL)
    get curr byte of string (start with most sig. digit)
    error check byte (does it represent a number?)
    byte_val := integer val for that byte (use subtraction!)
    theint := (theint*10) + byte_val
    curr := curr+1 (incr curr variable for next digit)
END WHILE
print the int using print_int syscall
```

Of course, there are other ways to do the conversion, but this is the easiest to understand.

Integer to String (i2s)

This is just the reverse of the ascii to integer problem. The integer will be in the data section of your program, as will a buffer that will store the final string:

```
theint:      .word      54321      #the integer to convert
final_buf:   .byte      0:12      #buffer for up to 11-digits
```

A little pseudocode:

```
load integer into register
print integer using print_int syscall
temp := integer
numdigits := 1                      #count the number of digits
WHILE (temp >= 10)
    numdigits := numdigits + 1
    temp := temp DIV 10
END WHILE
bufferindex := numdigits
buffer[bufferindex] := NULL (put terminating null char on string)
bufferindex := bufferindex - 1 (move left to the next index)
FOR x := (bufferindex) DOWNTO 0 DO (from less sig. digit)
    digit := integer MOD 10 (find the less sig. digit)
    convert digit to ascii (use addition!)
    integer := integer DIV 10
    buffer[x] := ascii_digit (store in buffer)
    x := x - 1
END FOR
print buffer using print_str syscall
```

Again, you don't have to use this algorithm, and it may not be the most efficient one possible, but it should give you an idea of what to do. Take a look at the MIPS division instructions, there's one for quotient (DIV in the above pseudocode) and one for remainder (MOD in the above pseudocode.)

We don't expect you to do subroutines for these programs; all of your code should be in one procedure in each program.

There is an ascii character code chart available for reference. Type `man ascii` in a terminal window to view it.

2 Some MIPS

Arrays

A string is just an array of characters that ends with a null. In `i2s` you're writing to a string, and in `s2i` you're reading from a string. When writing to the string you need to declare a buffer that's big enough to hold the maximum number of characters your number can have, PLUS a null. Note that all strings **MUST** end with a null, which is just the number 0 (not the character '0').

```
final_buf:  .byte    0:12      #buffer for up to 11-digits
```

Here, `final_buf` is a 12 byte array with each member initialized to zero.

You must access the individual bytes in this array in order to read or set each digit. There are a few different ways of going about this.

Assume that the register `$t0` contains the ASCII value of the character that you want to store into the array. You could just keep track of offsets into the array, in which case you would access the buffer as

```
lbu $t0, final_buf($s0)      # $s0 contains the offset into the array
```

Instead of storing offsets, you could walk the array one byte at a time. Doing this you don't maintain a pointer to the start of the array, but since you probably won't be doing random array access there's no real reason to keep that pointer anyways.

```
la $s0, final_buf           # put the address of final_buf into $s0
process_loop:
# Do some processing
lbu $t0, ($s0)
add $s0, $s0, 1
# if not done, jump back to process_loop
```

Syscalls

You will need two syscalls for this assignment. Syscall 1 is output integer, and syscall 4 is output string.

In order to make a syscall first put the ID of the syscall you want to make into \$v0, and then load other registers with the arguments to the syscall.

In order to print an integer put the actual integer that you want to output into \$a0, not the address of the integer. When printing a string put the address of the string you want to print into \$a0.

Once you have the registers loaded just use the syscall command.

The following code outputs the integer number 100:

```
li $v0, 1           # syscall 1 is print integer
li $a0, 100        # put the value to print into $a0
syscall            # the actual syscall
```

Style Requirements

For this project, and all future assembly assignments, we will be expecting you to write assembly code that conforms to Java/C flow control abstractions, and comment it with pseudocode. That is to say that your assembly code should map directly to Java/C, like so:

```

        .data
theint: .word 5
error: .asciiz "Error: negative number in factorial"
        .text
main:
# Factorial
# Computes the factorial of theint from the .data section
# Register Usage
# $s0 - n, the number we're computing the factorial of
# $s1 - accum, the factorial we're constructing
#####
        lw $s0, theint           #int n = 3;
        li $s1, 1                #int accum=1;
        bgez $s0, fact_beginloop #if(n<0){
        la $a0, error            #
        li $v0, 4                #
        syscall                  #     print error;
        j exit                   #     exit;
fact_beginloop:                  #}
        beqz $s0, fact_finish    #while(n != 0){
        mul $s1, $s1, $s0        #     accum *= n;
        sub $s0, $s0, 1          #     n--;
        j fact_beginloop        #}
fact_finish:                     #
        move $a0, $s1           #
        li $v0, 1                #
        syscall                  #print accum;
exit:
        done

```

You should write and comment all your code like this, with high level programming abstractions alongside your actual code. If your code contains jumps all over the place such that it can't be mapped to Java/C control structures, it is what we call spaghetti code, and we will be taking points off.

We have to mention one stylistic point here: you should have only one use of `done` in your program. This is because it's generally a good idea to have only one exit point in your program. You can do something similar to this sample code: have an `exit` label, or something similar, and jump to that

whenever you want your program to finish.

It's often tempting to write spaghetti code in order to make your programs faster or reduce the amount of code written; however, the point of this class is not to teach you write blazing fast assembly code (you can work on that in your own time). The point of the class is to allow you to understand what's going on underneath all the higher level code you write, and thus we want you to write assembly that reflects that.

You should also add regular, explanatory comments where appropriate. For this assignment psuedocode will probably suffice, unless you're doing something unusual. In general make sure you document your register usage for EACH SUBROUTINE, but here you just need to document your main procedure. This is crucial. Register tables are very helpful when debugging assembly as well.

3 Grading criteria

You will be graded on the following criteria:

- does the program work?
- are error cases handled properly?
- good structure of code (loops, if-else, branching)
- commenting (header, inlines, register usage)

This is a warm up assignment. You should use it to get used to mipscope and the CS31 coding conventions. Getting the programs to work should not be very difficult. Using good control structures and commenting will be as important as having a working program for this assignment.

4 Handing in:

To hand in your programs, type: `cs031.handin chars`.