CSc 453: Operational Characteristics of Clike

The language we have been using in class has many of the characteristics of a block-structured programming language. Mentioned below are key issues regarding clike.

In general, clike is a 32-bit language: 32-bit pointers, 32-bit ints. The only exception is the double which is 64-bits.

Data

Scalars:

An object of type int occupies 32-bits (4 bytes); an object of type char occupies 8 bits (1 byte); and an object of type double occupies 64-bits (8 bytes). Values of type char are considered to be signed entities, and widening a char to an int requires sign extension.

Arrays:

Arrays are zero-based and stored contiguously in memory. An array of type T of n elements should consume exactly n*sizeof(T) bytes of memory.

Alignment:

The char has no particular alignment. The int should be 4-byte aligned when in memory and the double should be 8-byte aligned.

Expressions

Order of Evaluation:

Obviously, the operands of an expression have to be evaluated before the expression can be evaluated. When there is a question of order, (for example: In f()+g(), should f or g be evaluated first?) the order of evaluation is left unspecified. Note that we must still respect associativity and precedence.

The && and || operators are the exception: they should be evaluated using short-circuit evaluation. They should be evaluated left-to-right and the result should be returned as soon as the truthiness or falsiness of the statement is known (much like C: for example: if (ap != NULL && *ap==a) will NOT evaluate *ap==a if ap==NULL).

Type Conversion:

If an object of type char is part of an expression, its value is converted (sign extended) to a value of type int before the expression is evaluated.

There is no implicit type conversion between int/char and double.

Array Indexing:

Arrays are zero-based, i.e., the elements of an array of n elements are indexed from 0 to n-1. The result of indexing into an array with an out-of-range index is unspecified.

Assignment Statements

Order of evaluation:

The order in which the left and right hand sides of an assignment are evaluated is left unspecified.

Type Conversion:

A value of type char is converted (sign-extended) to a 32-bit quantity before it is assigned to a value of type int.

A value of type int is converted (truncated) to an 8-bit quantity, by discarding the top 24 bits, before it is assigned to an object of type char.

An object of type double can only be assigned from an object of type double.
Functions

Evaluation of Actuals:
The order in which the actual parameters in a function call are evaluated is unspecified.

Parameter Passing:
Scalar values are passed by value, except for `double`, which is passed by reference.
When a `double` is passed, it is passed by reference.
Since a function that has a formal parameter of type `char` will, in any case, be passed a 32-bit quantity as an actual, it must convert (truncate) the actual to an 8-bit quantity before using it.

Return from a Function:
Execution returns from a function if either (a) an explicit `return` statement is executed; or (b) if execution "falls off" the end of the function body. In the latter case, no value is returned.

Program Execution

Execution begins at a procedure named `main` (however, nothing special needs to be done for this because the appropriate initialization and cleanup code will be inserted by the C Compiler during assembly).

Standard Library

Your compiler will have a very small standard library (remember, libraries are part of a language specification as well).

A group of special functions allows you to print and convert:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void printint(int x)</code></td>
<td>prints a single int</td>
</tr>
<tr>
<td><code>void printchar(char c)</code></td>
<td>prints a single char</td>
</tr>
<tr>
<td><code>void printdouble(double d)</code></td>
<td>print a single double</td>
</tr>
<tr>
<td><code>int toint(double d)</code></td>
<td>floor of number, then convert to int</td>
</tr>
<tr>
<td><code>double todouble(int i)</code></td>
<td>convert int to a double</td>
</tr>
</tbody>
</table>

These functions can be implemented simply in assembly using the syscall facility of MIPS. For purposes of the `clike` language, you may assume the functions are defined, but you still need to declare prototypes before you can call them.

Separate Linking

The assembly generated by your compiler should be able to be linked against other assembly files generated by your compiler by "concatenating" them together. This means labels and branches in your input files should be completely distinct from labels and branches in other files.

This will help us test that you are adhering to the MIPS linkage conventions, as we will have legal assembly already generated that you will have to link against.

Misc

A handout describing some MIPS specifics (function call protocol, helper, etc.) will follow.