

Compile-time text/string manipulation

Source: <http://coding.derkeiler.com/Archive/Assembler/alt.lang.asm/2005-07/msg00651.html>

- *From:* randyhyde@xxxxxxxxxxxxxx
 - *Date:* 13 Jul 2005 12:14:26 -0700
-

Text and String Manipulation at Compile Time

The measure of an assembler's power rarely has anything to do with the translation of individual machine instructions into object code. ALL assemblers do this job, so the translation of source code instructions into their corresponding machine code is hardly a cause for celebration among assembler users. The place where an assembler rises above the crowd is in the areas where the assembler helps the user automate code generation. The principle tool for this task is the macro and the assembler's "compile-time language."

A "compile-time language" is one that processes statements during the compilation of some other program. Compile time languages for assemblers generally process these kinds of statements:

1. Compile-time assignment statements. These statements let an assembler assign values to a compile-time variable. Equates are a common example of compile-time assignment statements.
2. Compile-time declarations. This group includes macro declarations and constant declarations (also called equates, but the type of equate that doesn't allow a reassignment, which is fundamentally different from an equate that does allow one to reassign a value).
3. Compile-time control structures. This group of statements includes "conditional assembly/compilation statements", compile-time loops, macro invocations, and similar statements.
4. Compile-time functions that can be used to compute compile-time (constant) results based on other compile-time expressions.

Compile-time text/string manipulation

In general, compile-time language sequences are short scripts that automate the generation of more complex assembly language code fragments. Some low-level assemblers, for example, attempt to use macros to simulate (with varying degrees of success) HLL-like control structures found in high-level assemblers.

While many macros and compile-time code sequences work by mapping a single identifier (or an identifier with a set of parameters) to a fixed sequence of machine instructions, really sophisticated compile-time scripts often build up machine instructions and other assembler directives from their individual components. They treat portions of the source file as string data and manipulate those strings to produce new source code, which the assembler then converts to machine code. To utilize such sophistication, an assembler's user requires access to various string and text manipulation functions. And that is the purpose of this article, to describe text and string manipulation by an assembler's compile-time language.

Though the phrases "text data" and "string data" would appear to be synonymous, in assembly language programs these two types of data are handled quite differently. In HLA, for example, consider the following declaration sequence:

```
const
textConst:text := "i";
strConst:string := "i";
```

Internally, these two compile-time constants hold the same exact data, the single character 'i'. However, HLA treats these two identifiers quite differently. Whenever you use the identifier "strConst" in an expression, HLA substitutes the character string data "i" in place of the identifier. That is, the following will print the single character 'i' to the standard output device:

```
stdout.put( strConst );
```

That is, this is equivalent to the following:

```
stdout.put( "i" );
```

HLA treats text constants quite a bit different than string constants. HLA takes the string data associated with a text constant and expands it in-line as though it were part of the source file. E.g., given

Compile-time text/string manipulation

```
stdout.put( textConst );
```

HLA expands this to:

```
stdout.put( i );
```

There is a crucial difference here. For the string data, HLA simply prints the character string 'i'. For the text data, HLA expands the text constant to the identifier `i`, which causes `stdout.put` to print the current value of the `i` symbol. To demonstrate this, consider:

```
program textVsStr;
const
textConst:text := "i";
strConst:string := "i";

static
i :int32 := -1;

begin textVsStr;

stdout.put( strConst, nl ); // prints "i"
stdout.put( textConst, nl ); // prints "-1"

end textVsStr;
```

So what are TEXT objects useful for?

Well, most HLA programmers use text constants as a sort of "tiny macro" that will expand an identifier to a fixed piece of text. For example, it's common to see text constants like the following in an HLA program:

```
type
x:record
<<various fields>>
endrecord;

const
xEAX :text := "(type x [eax])";
```

This text constant implies that `EAX` contains a pointer to a record object of type `x`. With this text equate, you can write code like the following:

```
mov( xEAX.someField, cl );
```

Rather than having to write out

```
mov( (type x [eax]).someField, cl );
```

Text objects pop up all over the place in HLA. For example, formal macro parameters are good examples of text objects. When you invoke a macro, the HLA compiler assigns the text associated with the actual macro parameter to macro parameter text object. That way, when you reference the macro parameter within the macro, it expands to the text of the actual parameter. Local symbols in HLA are also text objects. The HLA compiler generates a string of characters that form a unique symbol and then assigns this string to the local label (which is a text object). Whenever you reference the local label within the macro, it expands to the text assigned to it, which is a unique symbol that HLA has generated.

Though you can do some interesting things with text objects, their real power lies in the fact that HLA provides a *very* rich set of string manipulation functions that help you build complex strings for use by text variables.

Perhaps the six most important compile-time string functions HLA provides are:

1. `@length(s)` -- Computes the current length of `s`.
2. `s1 + s2` -- Concatenates string `s1` and `s2`.
3. `@substr(s, start, len)` -- Extracts a substring from `s`.
4. `@index(s1, start, s2)` -- Searches for the first occurrence of `s2` within `s1`, starting at character position "start" in `s1`.
5. `@string(v)` converts `v` to string form. If `v` is an integer value, this produces a string representation of that numeric value. If `v` is a text object, `@string` returns its string value without first expanding the text in-place in the source file.
6. `@text(s)` expands the string data in `s` as a text value at that point in the source file.

HLA, literally, provides several dozen (if not over a hundred) different compile-time string functions that you can use to manipulate text objects. For more information on the wide variety of functions, please

Compile-time text/string manipulation

consult the HLA manual. This article will simply use the above functions, which are almost sufficient (with a bit of work, mind you) to simulate most of the other functions you might want to use.

Let's assume that `r1` and `r2` are both string constants, holding 32-bit register names. Now consider the following text generation:

```
const
aNewInstr :text := "mov( " + r1 + ", " + r2 + " );
```

If `r1` happens to be "EAX" and `r2` is "EBX", then placing "aNewInstr;" in your source file (within the scope of the above declaration) yields:

```
mov( EAX, EBX );
```

The interesting part is that this statement can generate different instructions, depending upon the values of `r1` and `r2` when HLA encounters the declaration for "aNewInstr".

Of course, the above example could also be handled by a macro, but that's not the point -- the point is that we can generate new instructions on the fly using string manipulation.

Another reason for having string manipulation functions in an assembler's compile-time language is that macro parameters and local symbols are text and string objects. Built-in string functions let you manipulate the parameters you pass to a macro.

For example, Rene ("Betov") has suggested using the operator "<s" to imply a signed less than comparison in a boolean expression. Though this is handled in a completely different way in HLA (using type coercion), if you're really dead-set on using this syntax, it's easy enough to use HLA's built-in text manipulation functions to do this for you. Consider the following macro:

```
program main;

#macro _if( _expr_ ):
  _x_,
  _leftOperand_,
  _rightOperand_,
  _pos_,
  _type_,
```

Compile-time text/string manipulation

```
_size_ ;

?_x_ := @string( _expr_ );
?_pos_ := @index( _x_, 0, "<s" );
#if( _pos_ <> -1 )

?_leftOperand_ := @substr( _x_, 0, _pos_ - 1 );
?_rightOperand_ := @substr( _x_, _pos_ + 2, 10000 );
?_size_ := @size( @text( _leftOperand_ ) );
#if( _size_ = 1 )

?_type_ := "int8";

#elseif( _size_ = 2 )

?_type_ := "int16";

#elseif( _size_ = 4 )

?_type_ := "int32";

#else

#error( "<s only allows 8, 16, or 32-bit operands" )
?_type_ := "int8";

#endif
if( @text( "( type "+_type_" "+_leftOperand_+" ) < " +
_rightOperand_ ) )

// If you like, insert code to handle other other three signed
// comparisons here.

#else

if( _expr_ )

#endif

#endifmacro

begin main;

_if( eax <s ebx ) then

neg( ecx );

endif;

_if( eax < ebx ) then
```

Compile-time text/string manipulation

```
neg( edx );  
  
endif;  
  
end main;
```

HLA compiles this to the following MASM code:

```
cmp eax, ebx  
jnl L7_false__hla_  
neg ecx  
L7_false__hla_:  
cmp eax, ebx  
jnb L8_false__hla_  
neg edx  
L8_false__hla_:
```

So this example provides a good demonstration of why you would want to have a good set of text/string manipulation functions in an assembler -- so you can adjust the syntax of macro calls to a form you find more convenient. Whether you think using "<s" is a good idea or not (personally, I think it stinks, how do you differentiate "if(eax < s)..." from "if(eax <s ebx)...", for example), having the ability to process statements like this is very important.

HLA, without question, provides the richest compile-time language available for x86 assemblers (and probably all assemblers, for that matter). So how do other assemblers stack up? Well, MASM provides the basic length, substring, concatenation, index, text expansion and string conversion operations (though not much more than this), so it is capable of most major text/string operations you'd want to do at compile-time (though you may have to write some sophisticated macros to synthesize more complex string operations out of the ones that MASM provides).

As Rene brought up the "<s" operator in a previous post, how well does RosAsm stack up in this department? Well, AFAIKT, there is no string length operator. Substring doesn't really exist, though you can peel one character at a time off of a string (though you only keep the remainder of the string, not the character you peeled off, which would have been useful). RosAsm seems to support concatenation, though I cannot tell how

Compile-time text/string manipulation

general it is. And I don't see any sort of index operation present in the RosAsm compile-time language.

With RosAsm, your ability to write sophisticated macros depends upon your ability to leverage the existing RosAsm syntax, it is quite difficult to "roll your own" syntax using string manipulation at assembly time in RosAsm. Clearly, MASM and HLA are far more powerful than RosAsm in this regard.

As a closing example, consider the following "u32" macro that converts an arithmetic expression involving unsigned 32-bit operands into a sequence of machine instructions that compute the result. The interesting part of this example isn't so much the compiler, but rather than "lexical analyzer" function that does the string processing on the parameters you pass to the U32 function. This lexer macro demonstrates some fairly sophisticated compile-time string and text manipulation.

Cheers,
Randy Hyde

```
// u32expr.hla
//
// This program demonstrates how to write an "expression compiler"
// using the HLA compile-time language. This code defines a macro
// (u32expr) that accepts an arithmetic expression as a parameter.
// This macro compiles that expression into a sequence of HLA
// machine language instructions that will compute the result of
// that expression at run-time.
//
// The u32expr macro does have some severe limitations.
// First of all, it only support uns32 operands.
// Second, it only supports the following arithmetic
// operations:
//
// +, -, *, /, <, <=, >, >=, =, <>.
//
// The comparison operators produce zero (false) or
// one (true) depending upon the result of the (unsigned)
// comparison.
//
// The syntax for a call to u32expr is
//
// u32expr( register, expression )
//
// The macro computes the result of the expression and
// leaves this result sitting in the register specified
```

Compile-time text/string manipulation

```
// as the first operand. This register overwrites the
// values in the EAX and EDX registers (though these
// two registers are fine as the destination for the
// result).
//
// This macro also returns the first (register) parameter
// as its "returns" value, so you may use u32expr anywhere
// a 32-bit register is legal, e.g.,
//
// if( u32expr( eax, (i*3-2) < j ) ) then
//
// << do something if (i*3-2) < j >>
//
// endif;
//
// The statement above computes true or false in EAX and the
// "if" statement processes this result accordingly.
```

```
program TestExpr;
#include( "stdlib.hhf" )
```

```
// Some special character classifications the lexical analyzer uses.
```

```
const
```

```
// tok1stIDChar is the set of legal characters that
// can begin an identifier. tokIDChars is the set
// of characters that may follow the first character
// of an identifier.
```

```
tok1stIDChar := { 'a'..'z', 'A'..'Z', '_' };
tokIDChars := { 'a'..'z', 'A'..'Z', '0'..'9', '_' };
```

```
// digits, hexDigits, and binDigits are the sets
// of characters that are legal in integer constants.
// Note that these definitions don't allow underscores
// in numbers, although it would be a simple fix to
// allow this.
```

```
digits := { '0'..'9' };
hexDigits := { '0'..'9', 'a'..'f', 'A'..'F' };
binDigits := { '0'..'1' };
```

```
// CmpOps, PlusOps, and MulOps are the sets of
// operator characters legal at three levels
// of precedence that this parser supports.
```

```
CmpOps := { '>', '<', '=', '!' };
```

Compile-time text/string manipulation

```
PlusOps := { '+', '-' };  
MulOps := { '*', '/' };
```

type

```
// tokEnum-  
//  
// Data values the lexical analyzer returns to quickly  
// determine the classification of a lexeme. By  
// classifying the token with one of these values, the  
// parser can more quickly process the current token.  
// I.e., rather than having to compare a scanned item  
// against the two strings "+" and "-", the parser can  
// simply check to see if the current item is a "plusOp"  
// (which indicates that the lexeme is "+" or "-").  
// This speeds up the compilation of the expression since  
// only half the comparisons are needed and they are  
// simple integer comparisons rather than string comparisons.
```

```
tokEnum: enum  
{  
  identifier,  
  intconst,  
  lparen,  
  rparen,  
  plusOp,  
  mulOp,  
  cmpOp  
};
```

```
// tokType-  
//  
// This is the "token" type returned by the lexical analyzer.  
// The "lexeme" field contains the string that matches the  
// current item scanned by the lexer. The "tokClass" field  
// contains a generic classification for the symbol (see the  
// "tokEnum" type above).
```

```
tokType:  
record  
  
  lexeme:string;  
  tokClass:tokEnum;  
  
endrecord;
```

Compile-time text/string manipulation

```
// lexer-  
//  
// This is the lexical analyzer. On each call it extracts a  
// lexical item from the front of the string passed to it as a  
// parameter (it also removes this item from the front of the  
// string). If it successfully matches a token, this macro  
// returns a tokType constant as its return value.  
  
#macro lexer( input ):theLexeme,boolResult;  
  
?theLexeme:string; // Holds the string we scan.  
?boolResult:boolean; // Used only as a dummy value.  
  
// Check for an identifier.  
  
#if( @peekCset( input, tok1stIDChar ))  
  
// If it began with a legal ID character, extract all  
// ID characters that follow. The extracted string  
// goes into "theLexeme" and this call also removes  
// those characters from the input string.  
  
?boolResult := @oneOrMoreCset( input, tokIDChars, input,  
theLexeme );  
  
// Return a tokType constant with the identifier string and  
// the "identifier" token value:  
  
tokType:[ theLexeme, identifier ]  
  
  
// Check for a decimal numeric constant.  
  
#elseif( @peekCset( input, digits ))  
  
// If the current item began with a decimal digit, extract  
// all the following digits and put them into "theLexeme".  
// Also remove these characters from the input string.  
  
?boolResult := @oneOrMoreCset( input, digits, input, theLexeme  
);  
  
// Return an integer constant as the current token.  
  
tokType:[ theLexeme, intconst ]  
  
  
// Check for a hexadecimal numeric constant.
```

Compile-time text/string manipulation

```
#elseif( @peekChar( input, '$' ))

// If we had a "$" symbol, grab it and any following
// hexadecimal digits. Set boolResult true if there
// is at least one hexadecimal digit. As usual, extract
// the hex value to "theLexeme" and remove the value
// from the input string:

?boolResult := @oneChar( input, '$', input ) &
@oneOrMoreCset( input, hexDigits, input,
theLexeme );

// Returns the hex constant string as an intconst object:

tokType:[ '$' + theLexeme, intconst ]

// Check for a binary numeric constant.

#elseif( @peekChar( input, '%' ))

// See the comments for hexadecimal constants. This boolean
// constant scanner works the same way.

?boolResult := @oneChar( input, '%', input ) &
@oneOrMoreCset( input, binDigits, input,
theLexeme );
tokType:[ '%' + theLexeme, intconst ]

// Handle the "+" and "-" operators here.

#elseif( @peekCset( input, PlusOps ))

// If it was a "+" or "-" sign, extract it from the input
// and return it as a "plusOp" token.

?boolResult := @oneCset( input, PlusOps, input, theLexeme );
tokType:[ theLexeme, plusOp ]

// Handle the "*" and "/" operators here.

#elseif( @peekCset( input, MulOps ))

// If it was a "*" or "/" sign, extract it from the input
// and return it as a "mulOp" token.

?boolResult := @oneCset( input, MulOps, input, theLexeme );
```

Compile-time text/string manipulation

```
tokType:[ theLexeme, mulOp ]

// Handle the "=" ("=="), "<>" ("!="), "<", "<=", ">", and ">="
// operators here.

#elseif( @peekCset( input, CmpOps ))

// Note that we must check for two-character operators
// first so we don't confuse them with the single
// character operators:

#if
(
  @matchStr( input, ">=", input, theLexeme )
| @matchStr( input, "<=", input, theLexeme )
| @matchStr( input, "<>", input, theLexeme )
)

tokType:[ theLexeme, cmpOp ]

#elseif( @matchStr( input, "!=", input, theLexeme ))

tokType:[ "<>", cmpOp ]

#elseif( @matchStr( input, "==", input, theLexeme ))

tokType:[ "=", cmpOp ]

#elseif( @oneCset( input, {'>', '<', '='}, input, theLexeme ))

tokType:[ theLexeme, cmpOp ]

#else

#error( "Illegal comparison operator: " + input )

#endif

// Handle the parentheses down here.

#elseif( @oneChar( input, '(', input, theLexeme ))

tokType:[ "(", lparen ]

#elseif( @oneChar( input, ')', input, theLexeme ))

tokType:[ ")", rparen ]
```

Compile-time text/string manipulation

```
// Anything else is an illegal character.

#else

#error
(
  "Illegal character in expression: " +
  @substr( input, 0, 1 ) +
  " ($" +
  string( dword( @substr( input, 0, 1 ))) +
  ")"
)
?input := @substr( input, 1, @length(input) - 1 );

#endif

#endmacro;

// Handle identifiers, constants, and sub-expressions within
// parentheses within this macro.
//
// terms-> identifier | intconst | '(' CmpOps ')'
//
// This compile time function does the following:
//
// (1) If it encounters an identifier, it emits the
// following instruction to the code stream:
//
// push( identifier );
//
// (2) If it encounters an (unsigned) integer constant, it emits
// the following instruction to the code stream:
//
// pushd( constant_value );
//
// (3) If it encounters an expression surrounded by parentheses,
// then it emits whatever instruction sequence is necessary
// to leave the value of that (unsigned integer) expression
// sitting on the top of the stack.
//
// (4) If the current lexeme is none of the above, then this
// macro prints an appropriate error message.
//
// The end result of the execution of this macro is the emission
// of some code that leaves a single 32-bit unsigned value sitting
// on the top of the 80x86 stack (assuming no error).
```

Compile-time text/string manipulation

```
#macro doTerms( expr ):termToken;

?expr := @trim( expr, 0 );
?termToken:tokType := lexer( expr );
#if( termToken.tokClass = identifier )

// If we've got an identifier, emit the code to
// push that identifier onto the stack.

push( @text( termToken.lexeme ) );

#elseif( termToken.tokClass = intconst )

// If we've got a constant, emit the code to push
// that constant onto the stack.

pushd( @text( termToken.lexeme ) );

#elseif( termToken.tokClass = lparen )

// If we've got a parenthetical expression, emit
// the code to leave the parenthesized expression
// sitting on the stack.

doCmpOps( expr );
?expr := @trim( expr, 0 );
?termToken:tokType := lexer( expr );
#if( termToken.tokClass <> rparen )

#error( "Expected closing parenthesis: " + termToken.lexeme
)

#endif

#else

#error( "Unexpected term: '" + termToken.lexeme + "'" )

#endif

#endifmacro;

// Handle the multiplication, division, and modulo operations here.
//
// MulOps-> terms ( mulOp terms )*
//
// The above grammar production tells us that a "MulOps" consists
// of a "terms" expansion followed by zero or more instances of a
```

Compile-time text/string manipulation

```
// "mulop" followed by a "terms" expansion (like wildcard filename
// expansions, the "*" indicates zero or more copies of the things
// inside the parentheses).
//
// This code assumes that "terms" leaves whatever operands/expressions
// it processes sitting on the 80x86 stack at run time. If there is
// a single term (no optional mulOp/term following), then this code
// does nothing (it leaves the result on the stack that was pushed
// by the "terms" expansion). If one or more mulOp/terms pairs are
// present, then for each pair this code assumes that the two "terms"
// expansions left some value on the stack. This code will pop
// those two values off the stack and multiply or divide them and
// push the result back onto the stack (sort of like the way the
// FPU multiplies or divides values on the FPU stack).
//
// If there are three or more operands in a row, separated by
// mulops ("*" or "/") then this macro will process them in
// a left-to-right fashion, popping each pair of values off the
// stack, operating on them, pushing the result, and then processing
// the next pair. E.g.,
//
// i * j * k
//
// yields:
//
// push( i ); // From the "terms" macro.
// push( j ); // From the "terms" macro.
//
// pop( eax ); // Compute the product of i*j
// mul( (type dword [esp]));
// mov( eax, [esp]);
//
// push( k ); // From the "terms" macro.
//
// pop( eax ); // Pop K
// mul( (type dword [esp])); // Compute K* (i*j) [i*j is value
// on TOS].
// mov( eax, [esp]); // Save product on TOS.

#macro doMulOps( sexpr ):opToken;

// Process the leading term (not optional). Note that
// this expansion leaves an item sitting on the stack.

doTerms( sexpr );

// Process all the MULOPs at the current precedence level.
// (these are optional, there may be zero or more of them.)

?sexpr := @trim( sexpr, 0 );
```

Compile-time text/string manipulation

```
#while( @peekCset( sexpr, MulOps ))

// Save the operator so we know what code we should
// generate later.

?opToken := lexer( sexpr );

// Get the term following the operator.

doTerms( sexpr );

// Okay, the code for the two terms is sitting on
// the top of the stack (left operand at [esp+4] and
// the right operand at [esp]). Emit the code to
// perform the specified operation.

#if( opToken.lexeme = "*" )

// For multiplication, compute
// [esp+4] = [esp] * [esp+4] and
// then pop the junk off the top of stack.

pop( eax );
mul( (type dword [esp]) );
mov( eax, [esp] );

#elseif( opToken.lexeme = "/" )

// For division, compute
// [esp+4] = [esp+4] / [esp] and
// then pop the junk off the top of stack.

mov( [esp+4], eax );
xor( edx, edx );
div( [esp], edx:eax );
pop( edx );
mov( eax, [esp] );

#endif
?sexpr := @trim( sexpr, 0 );

#endwhile

#endmacro;

// Handle the addition, and subtraction operations here.
//
// AddOps-> MulOps ( addOp MulOps )*
```

Compile-time text/string manipulation

```
//
// The above grammar production tells us that an "AddOps" consists
// of a "MulOps" expansion followed by zero or more instances of an
// "addOp" followed by a "MulOps" expansion.
//
// This code assumes that "MulOps" leaves whatever operands/expressions
// it processes sitting on the 80x86 stack at run time. If there is
// a single MulOps item then this code does nothing. If one or more
// addOp/MulOps pairs are present, then for each pair this code
// assumes that the two "MulOps" expansions left some value on the
// stack. This code will pop those two values off the stack and
// add or subtract them and push the result back onto the stack.

#macro doAddOps( sexpr ):opToken;

// Process the first operand (or subexpression):

doMulOps( sexpr );

// Process all the ADDOPs at the current precedence level.

?sexpr := @trim( sexpr, 0 );
#while( @peekCset( sexpr, PlusOps ))

// Save the operator so we know what code we should
// generate later.

?opToken := lexer( sexpr );

// Get the MulOp following the operator.

doMulOps( sexpr );

// Okay, emit the code associated with the operator.

#if( opToken.lexeme = "+" )

pop( eax );
add( eax, [esp] );

#elseif( opToken.lexeme = "-" )

pop( eax );
sub( eax, [esp] );

#endif

#endwhile

#endmacro;
```

Compile-time text/string manipulation

```
// Handle the comparison operations here.
//
// CmpOps-> addOps ( cmpOp AddOps )*
//
// The above grammar production tells us that a "CmpOps" consists
// of an "AddOps" expansion followed by zero or more instances of an
// "cmpOp" followed by an "AddOps" expansion.
//
// This code assumes that "MulOps" leaves whatever operands/expressions
// it processes sitting on the 80x86 stack at run time. If there is
// a single MulOps item then this code does nothing. If one or more
// addOp/MulOps pairs are present, then for each pair this code
// assumes that the two "MulOps" expansions left some value on the
// stack. This code will pop those two values off the stack and
// add or subtract them and push the result back onto the stack.

#macro doCmpOps( sexpr ):opToken;

// Process the first operand:

doAddOps( sexpr );

// Process all the CMPOPs at the current precedence level.

?sexpr := @trim( sexpr, 0 );
#while( @peekCset( sexpr, CmpOps ))

// Save the operator for the code generation task later.

?opToken := lexer( sexpr );

// Process the item after the comparison operator.

doAddOps( sexpr );

// Generate the code to compare [esp+4] against [esp]
// and leave true/false sitting on the stack in place
// of these two operands.

#if( opToken.lexeme = "<" )

pop( eax );
cmp( [esp], eax );
setb( al );
movzx( al, eax );
mov( eax, [esp] );
```

Compile-time text/string manipulation

```
#elseif( opToken.lexeme = "<=" )
```

```
pop( eax );  
cmp( [esp], eax );  
setbe( al );  
movzx( al, eax );  
mov( eax, [esp] );
```

```
#elseif( opToken.lexeme = ">" )
```

```
pop( eax );  
cmp( [esp], eax );  
seta( al );  
movzx( al, eax );  
mov( eax, [esp] );
```

```
#elseif( opToken.lexeme = ">=" )
```

```
pop( eax );  
cmp( [esp], eax );  
setae( al );  
movzx( al, eax );  
mov( eax, [esp] );
```

```
#elseif( opToken.lexeme = "=" )
```

```
pop( eax );  
cmp( [esp], eax );  
sete( al );  
movzx( al, eax );  
mov( eax, [esp] );
```

```
#elseif( opToken.lexeme = "<>" )
```

```
pop( eax );  
cmp( [esp], eax );  
setne( al );  
movzx( al, eax );  
mov( eax, [esp] );
```

```
#endif
```

```
#endwhile
```

```
#endmacro;
```

```
// General macro that does the expression compilation.  
// The first parameter must be a 32-bit register where
```

Compile-time text/string manipulation

```
// this macro will leave the result. The second parameter
// is the expression to compile. The expression compiler
// will destroy the value in EAX and may destroy the value
// in EDX (though EDX and EAX make fine destination registers
// for this macro).
//
// This macro generates poor machine code. It is more a
// "proof of concept" rather than something you should use
// all the time. Nevertheless, if you don't have serious
// size or time constraints on your code, this macro can be
// quite handy. Writing an optimizer is left as an exercise
// to the interested reader.

#macro u32expr( reg, expr):sexpr;

// The "returns" statement processes the first operand
// as a normal sequence of statements and then returns
// the second operand as the "returns" value for this
// macro.

returns
(
{

?sexpr:string := @string:expr;
#if( !@IsReg32( reg ) )

#error( "Expected a 32-bit register" )

#else

// Process the expression and leave the
// result sitting in the specified register.

doCmpOps( sexpr );
pop( reg );

#endif
},

// Return the specified register as the "returns"
// value for this compilation:

@string:reg
)

#endmacro;
```

Compile-time text/string manipulation

```
// The following main program provides some examples of the
// use of the above macro:
```

```
static
x:uns32;
v:uns32 := 5;

begin TestExpr;

mov( 10, x );
mov( 12, ecx );

// Compute:
//
// edi := (x*3/v + %1010 == 16) + ecx;
//
// This is equivalent to:
//
// edi := (10*3/5 + %1010 == 16) + 12
// := ( 30/5 + %1010 == 16) + 12
// := ( 6 + 10 == 16) + 12
// := ( 16 == 16) + 12
// := ( 1 ) + 12
// := 13
//
// This macro invocation emits the following code:
//
// push(x);
// pushd(3);
// pop(eax);
// mul( (type dword [esp]) );
// mov( eax, [esp] );
// push( v );
// mov( [esp+4], eax );
// xor edx, edx
// div( [esp], edx:eax );
// pop( edx );
// mov( eax, [esp] );
// pushd( 10 );
// pop( eax );
// add( eax, [esp] );
// pushd( 16 );
// pop( eax );
// cmp( [esp], eax );
// sete( al );
// movzx( al, eax );
// mov( eax, [esp+0] );
// push( ecx );
// pop( eax );
```

Compile-time text/string manipulation

```
// add( eax, [esp] );
// pop( edi );

u32expr( edi, (x*3/v+%1010 == 16) + ecx );
stdout.put( "Sum = ", (type uns32 edi), nl );

// Now compute:
//
// eax := x + ecx/2
// := 10 + 12/2
// := 10 + 6
// := 16
//
// This macro emits the following code:
//
// push( x );
// push( ecx );
// pushd( 2 );
// mov( [esp+4], eax );
// xor( edx, edx );
// div( [esp], edx:eax );
// pop( edx );
// mov( eax, [esp] );
// pop( eax );
// add( eax, [esp] );
// pop( eax );

u32expr( eax, x+ecx/2 );
stdout.put( "x=", x, " ecx=", (type uns32 ecx), " v=", v, nl );
stdout.put( "x+ecx/2 = ", (type uns32 eax ), nl );

// Now determine if (x+ecx/2) < v
// (it is not since (x+ecx/2)=16 and v = 5.)
//
// This macro invocation emits the following code:
//
// push( x );
// push( ecx );
// pushd( 2 );
// mov( [esp+4], eax );
// xor( edx, edx );
// div( [esp], edx:eax );
// pop( edx );
// mov( eax, [esp] );
// pop( eax );
// add( eax, [esp] );
```

Compile-time text/string manipulation

```
// push( v );  
// pop( eax );  
// cmp( eax, [esp+0] );  
// setb( al );  
// movzx( al, eax );  
// mov( eax, [esp+0] );  
// pop( eax );
```

```
if( u32expr( eax, x+ecx/2 < v ) ) then
```

```
    stdout.put( "x+ecx/2 < v" nl );
```

```
else
```

```
    stdout.put( "x+ecx/2 >= v" nl );
```

```
endif;
```

```
end TestExpr;
```

```
.
```

- ***Follow-Ups:***

- ◆ ***Re: Compile-time text/string manipulation***

- ◇ *From:* Betov

- Prev by Date: ***Re: Which assembler can handle the BIG stuff ?***

- Next by Date: ***Re: Local Symbols in Macros (#1)***

- Previous by thread: ***Betov Assembler Naming Competition***

- Next by thread: ***Re: Compile-time text/string manipulation***

- Index(es):

- ◆ ***Date***

- ◆ ***Thread***