Language Reference Manual

From DSPLWiki

Jump to: navigation, search

Contents

[edit]

Introduction

Digital Signal Processing Language was created and designed for the purpose of providing an efficient language that can easily be compiled to make use of vector instructions that have been introduced in Intel's recent instruction sets. The language receives its name from the strong applicability these improvements have to many operations common in digital signal processing.

Since the introduction of Intel's MMX technology processor families, Intel has introduced four extensions into their architectures that support single-instruction multiple-data (SIMD). These extensions provide a group of instructions that perform SIMD operations on packed integer and/or packed floating-point data elements. Using these instructions enhances the performance of compatible processors for a variety of uses, including advanced 2-D and 3-D graphics, motion video, image processing, speech recognition, audio synthesis, telephony, and video conferencing.[1]

Lexical Conventions

[edit]

A program consists of a series of tokens which are grouped together to create declarations and statements. A program begins with a left brace "{" and ends with a right brace "}".

Tokens

[edit]

Like C, there are six classes of tokens: identifiers, keywords, constants, string literals, operators, and

other spearators. White-space, which includes blanks, horizontal and vertical tabs, newlines, formfeeds, and comments, are used to separate tokens and are otherwise ignored.

Comments [edit]

The characters "/*" introduce a comment, and "*/" terminate a comment. In addition, the characters "//" introduce a single-line comment, thus the next newline terminates the comment. Comments do not nest, and do not occur within string or characer literals.

Identifiers [edit]

Identifiers are sequences of letters, digits, and/or underscores. The first character must be a letter or underscore. Upper and lower case letters are different.

Keywords [edit]

The following identifiers are reserved as keywords and may not be used as identifiers:

- `byte`
- `ccall`
- `complex`
- `else`
- `float`
- `for`
- `if`
- `int`
- `string`
- `ubyte`
- `uint`
- `while`

Constants [edit]

There are several kinds of constants, including integers, characters, floating point numbers, double-precision floating point numbers, and complex numbers.

Integer Constants [edit]

Integer constants consist of a sequence of digits. The associated keyword is `int`. In order to specify an unsigned integer constant, create a signed int and then type cast to `uint`. To specify a signed byte, type caste to `byte`. To specify an unsigned byte, type cast to `ubyte`.

Floating Constants [edit]

Floating constants consist of a sequence of digits, a decimal point, and a sequence of digits. The associated keyword is `float`

Complex Constants [edit]

Complex constants consist of an opening brace character '{', a float (representing the real component), a comma, a float (representing the imaginary component), and a closing brace character '}'. The associated type is `complex`.

String Literals [edit]

String literals, also known as string constants, are a sequences of characters surrounded by double-quotes. Strings may consist of any character except for the double-quote, which must be escaped (`\"`). String literals are associated with the keyword `string`, and are implemented as arrays of type `char`. A null byte `/0` is appended to the end so that programs can find the string's end.

Syntax Notation [edit]

In the syntax notation used for this LRM, ANTLR-like code is used.

Meaning of Identifiers [edit]

Identifiers refer to objects. An object is a location in storage, and its interpretation depends on its storage type. The type determines the meaning of the values found in the identified object. The lifetime of all objects in DSPL is permanent and global; there is no scoping of variables.

Basic Types [edit]

There are several fundamental types. Strings are any combination of characters except for the double-quote, which must be escaped (`\"`). There are signed (`int`) and unsigned (`uint`) integers, which are 4 bytes long. There are signed (`byte`) and unsigned (`ubyte`) bytes, which are 1 byte long. Floating-point numbers (`float`) are 4 bytes long. Complex numbers (`complex float`) are pairs of 4-byte `float`s.

Derived Types [edit]

Besides the basic types, there may also be arrays any numeric basic type.

Objects and Lvalues [edit]

An object is a named region of storage, whereas an lvalue is an expression refrring to an object.

Conversions [edit]

No operators will automatically convert their operands to be compatible with each other; explicit typecasting must take place on the part of the user. When explicitly casting, only casts between `int`, `uint`, `byte`, and `ubyte` are allowed.

Expressions [edit]

The precedence of expression operators is the same as the order of the major subsections of this section, with higher precedence first. Left- or right- associativity is specified in each subsection for the operators discussed therein.

The handling of overflow, divide check, and other exceptions in expression evaluation will result in termination of a program, along with a message attempting to dictate where the exception ocurred.

Primary Expressions [edit]

Primary expressions are identifiers, constants, strings, or expressions in parentheses:

```
primary-expression:
        identifier
        | constant
        | string
        | '(' expression ')'
```

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration. An identifer is an lvalue if it refers to an object and if its type is arithmetic.

A constant is a primary expression. Its type depends on its form as discussed earlier.

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression.

C-library Call Expressions [edit]

C-library call expressions call functions from the C libray using objects existing in the current program.

```
ccall-expression:
        primary-expression
        | "ccall" string '(' (argument-expression-list)? ')'

argument-expression-list:
        assignment-expression
        | argument-expression-list ',' assignment-expression
```

Postfix Expressions [edit]

The operators in postfix expressions group left to right.

```
postfix-expression:
        ccall-expression
        | postfix-expression '[' expression ']'
```

The only postfix expression defined in DSPL is for array references. A postfix expression followed by an expression in square brackets is a postfix expression denoting a subscripted array reference. The post-fix expression must resolve to the lvalue of an array object, and the expression in the brackets must resolve to a non-negative integer.

Unary Operators [edit]

Expressions with unary operators group right-to-left.

```
post-fix expression:
        post-fix expression
        | unary-operator cast-expression

unary-operator:
        '!'
```

Logical Negation Operator [edit]

The operands of the `!` operator must have arithmetic type (can not be an array of arithmetic type), and the result is 1 if the value of its operand compares equal to 0, and 0 otherwise. The type returned is `int`.

Casts [edit]

A unary expression preceded by the parenthesized name of a type causes conversion of the value of the expression to the names type.

```
cast-expression:
        unary-expression
        | '(' type-name ')' cast-expression
```

Convolutional Operator [edit]

The convolutional operator ~ groups left to right.

```
convolutional-expression:
        unary-expression
        convolutional-expression '~' unary-expression
```

The operands of ~ must have array types. The two arrays needn't be of the same length. The convolutional operator performs the mathematical operation of calculating the convolution of the arrays.

Multiplicative Operators [edit]

The multiplicative operators `*`, `/`, and % group left-to-right.

```
multiplicative-expressions:
        unary-expression
        | multiplicative-expression '*' unary-expression
        | multiplicative-expression '/' unary-expression
        | multiplicative-expression '%' unary-expression
```

The operands of `*` and `/` must have arithmetic type or array of arithmetic type; the operands of '%' must have type of `int` or arrays of `int`. If performed on arrays, the two arrays must be of same length and type, and the result is an array for which each element is the result of the operation on the corresponding elements of the original arrays' elemennts.

The binary `*` operator denotes multiplication.

The binary `/` operator yields the quotient, and the % operator the remainder, of division of the first operand by the second; if the second operand is 0, the result is undefined.

Additive Operators [edit]

The additive operators + and `-` group left-to-right. The operands have arithmetic type or be arrays of

arthimetic types.

```
additive-operator:
        multiplicative-expression
        | additive-expression '+' multiplicative-expression
        | additive-expression '-' multiplicative-expression
```

The result of the + operator is the sum of the operands. The result of the - operator is the difference of the operands. The same rules apply when arrays are used as operands as described above for multiplicative operators.

Relational Operators [edit]

The relational operators group left-to-right, but this fact is not useful; a<b<c is parsed as (a<b)<c, and a<b evaluates to either 0 or 1.

```
relational-expression:
        additive-expression
        | relational-expression '<' additive-expression
        | relational-expression '>' additive-expression
        | relational-expression '<=' additive-expression
        | relational-expression '>=' additive-expression
```

The operators <<code> (less), <code>> (greater), <= (less or equal), >= (greather or equal) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is int. If arrays are used as operands, the relation must hold true for all elements in order to evaluate true.

Equality Operators [edit]

```
equality-expression:
        relational-expression
        | equality-expression '==' relational-expression
        | equality-expression '!=' relational-expression
```

The == (equal to) and the != (not equal to) operators are analogous to the relational operators except for their lower precedence.

Logical AND Operator [edit]

The && operator groups left-to-right.

```
logical-AND-expression:
        equality-expression
        | logical-AND-expression '&&' equality-expression
```

The && operator returns 1 if both its operands compare unequal to zero, 0 otherwise. It guarantees left-to-right evaluation: the first operand is evaluated, including all side effects; if it is equal to 0, the value of the expression is 0. Otherwise, the right operand is evaluated, and if it is equal to 0, the expression's value is 0, otherwise 1. The operands must evaluate to and integer value.

Logical OR Operator [edit]

The || operator groups left-to-right.

```
logical-OR-expression:
        logical-AND-expression
        | logical-OR-expression '||' logical-AND-expression
```

The || operator returns 1 if either of its operands compare unequal to zero, and 0 otherwise. Like &&, || guarantees left-to-right: the first operand is evaluated, including all side effects; if it is unequal to 0, the value of the expression is 1. Otherwise, the right operand is evaluated, and if it is unequal to 0, the expression's value is 1, otherwise 0. The operands must be integers.

Assignment Expression [edit]

The assignment operator groups left-to-right.

```
assignment-expression:
        conditional-expression
        | unary-expression '=' assignment-expression
```

The = operator requires an lvalue as left operand, and the lvalue must be modifiable. In DSPL, this means it may be an array, as well as a primitive type. The value of the expression replaces that of the object referred to by the lvalue. Both operands must have the same type, whether they are both the same primitive type or arrays of the same primitive type.

Declarations [edit]

Declarations specify the interpretation given to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations that reserve storage are called definitions. Declarations have the form

```
declaration:
        declaration-specifiers (init-declarator)? ';'
```

The declarator in the init-declarator contains the identifier being declared; the declaration-specifiers consist of a sequence of type specifiers.

```
declaration-specifiers:
        type-specifier (declaration-specifiers)?

init-declarator:
        declarator
        | declarator '=' initializer
```

Declarators will be discussed later; they contain the names being declared. A declaration has one and only one declarator.

Type Specifiers [edit]

The type specifiers are

```
type-specifier:
        "char"
        | ("complex")? "int"
        | ("complex")? "float"
        | "string"
```

One type-specifier may be given in a declaration.

Declarators [edit]

Declarators have the syntax:

```
declarator:
        identifier
        | declarator '[' constant-expression ']'
```

Meaning of Declarators [edit]

A declarator appears after a type specifier. The declarator declares a unique main identifier. The type depends on the form of the declarator. A declarator is read as an assertion that when its identifier appears in an expression of the same form as the declarator, it yields an object of the specified type. Thus, a

declaration has the form "`T D`," where `T` is a type and `D` is a declarator.

Array Declarators [edit]

In a declaration `T D` where `D` has the form

```
D1[ constant-expression ]
```

and the type of the identifier in the declaration `T D1` is "type modifier `T`," the type of the identifier of `D` is "type-modifier array of `T`." The constant-expression must be a positive `int`.

An array may be constructed from an arithmetic type. Multi-dimensional arrays are not supported.

Initialization [edit]

When an object is declared, its init-declarator may specify an initial value for the identifier being declared. The initalizer is preceded by `=`, and is an expression:

```
initializer:
        assignment-expression
```

The expression for an object need not be a constant expression, but must merely have appropriate type for assignment to the object. Initializer lists for arrays are not supported.

Statements [edit]

Except as described, statements are executed in sequence. Statements are executed for their effect, and do not have values. They fall into several groups.

```
statement:
        expression-statement
        | compound-statement
        | selection-statement
        | iteration-statement
```

Expression Statement [edit]

Most statements are expression statements, which have the form

```
expression-statement:
```

```
        (expression)? ';'
```

Most expression statements are assignments. All side effects from the expression are completed before the next statement is executed. If the expression is missing, the construction is called a null statement; it is often used to supply an empty body to an iteration statement.

Compound Statement [edit]

So that several statements can be used where one is expected, the compout statement (also called "block") is provided.

```
compound-statement:
        '{' (declaration-list)? (statement-list)? '}'

declaration-list:
        declaration
        | declaration-list declaration

statement-list:
        statement
        statement-list statement
```

DSPL does not support scoping, so careful attention should be paid to the inclusion of declarations in block statements that may be repeated.

Selection Statement [edit]

The selection statement choose one of several flows of control.

```
selection-statement:
        "if" '(' expression ')' statement
        | "if" '(' expression ')' statement "else" statement
```

In both forms of the `if` statement, the expression, which must have an `int` type, is evaluated, including all side-effects, and if it compares unqual to 0, the first substatement is executed. In the second form, the second substatement is executed if the expression is 0. The `else` ambiguity is resolved by connecting an `else` with the last encountered `else`-less `if` at the same block nesting level.

Iteration Statement [edit]

The iteration statement specify looping.

```
iteration-statement:
        "while" '(' expression ')' statement
```

In the `while` statement, the substatement is executed repeatedly so long as the value of the expresion remains unequal to 0; the expression must be of type `int`. The test, including all side effects from the expression, occurs before each execution of the statement.

Grammar [edit]

metaV's Grammar