Digital Signal Processing Language

Final Report

Jeff Cropsey

Dave Lariviere

Mike Lynch

Varun Maithel

Varun Mehta

Contents

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]

Brief Overview of Intel SIMD

Intel first introduced SIMD with MMX, which was available on some Pentiums as well as Pentium II's. MMX offered instructions for parallel operations on packed byte, word, or double word integers.

SSE (Streaming SIMD Execution) was first introduced in Pentium III, offering instructions for packed single-precision floating-point values.

SSE2 was introduced in the Pentium 4 and Intel Xeon processors, offering instructions for packed double-precision floating-point values, as well as new instructions for 128-bit integer operations.

SSE3 was most recently introduced in the Pentium 4 supporting Hyperthreading (P4+HT), which further added additional instructions, some of whose utility are demonstrated in Section XI.

DSPL will specifically target P4+HT and later architectures, in order to take full advantage of all SIMD instructions (MMX, SSE, SSE2, and SSE3).

Vector Operations

The main feature that DSPL offers is its ability to take advantage of SIMD (Singleinstruction Multiple-Data) instructions when compiled. These instructions allow for the parallel execution of multiple operations that would otherwise require separate instructions. The DSPL compiler uses these instructions when the language performs operations on arrays. For example, if `a`, `b`, and `c` are arrays of type `int` with an arbitrary number of elements, in DSPL it is possible to perform the following operation:

---

[1] ftp://download.intel.com/design/Pentium4/manuals/25366520.pdf

```
a = b + c; //where a[0] = b[0] + c[0], a[1] = b[1] + c[1], etc
```

The compiler uses SIMD instructions when creating assembly code for this statement, such that several additions can be performed simultaneously. In prior languages a `for` loop would be required, which would add each of the elements individually, resulting in many more instructions. Further, if the array sizes are known at compile time, it is possible to unroll the loop, further improving speed. Although this may seem like a rather simple improvement, in the applications noted above these operations are performed frequently and consequently optimization of this process has a dramatic overall effect.

Portability

A driving motivation behind DSPL is its use of all available SIMD instructions available on current-generation Intel processors. Therefore, DSPL has been designed to offer superior efficiency on modern systems (supporting SSE3), rather than worry about backwards compatibility with previous architectures. It is, however, conceivable to port the DSPL compiler to other architectures.

Data Types

As the focus of DSPL language is to increase ease and efficiency of vector calculations, DSPL uses a small set of data types. The data types implemented are `int`, `uint`, `byte`, `ubyte`, `float`, as well as the corresponding single-dimension array types `int_array[]`, `uint_array[]`, `byte_array[]`, `ubyte_array`, and `float_array[]`.

Operators

DSPL uses operators in very powerful ways. It implements the basic mathematical operators `+`, `-`, `*`, `/` (addition, subtraction, multiplication, and division), which are capable of operating on every data type (including single-dimension arrays. The use of array operators leads to cleaner, more elegant code that is less prone to mistakes by the programmer and more easily optimized by the compiler.

Control Keywords

The language supports a basic set of control characters needed for manipulating sets of data in calculations, consisting of `for`, `while` and a conditional structure `if` – `else`.

Calling functions

In order to provide a simple and clean way for a DSPL programmer to work with sets of data, the language includes the `ccall` funciton which allows the calling of C functions such as `printf`.

Syntax Example

C Base Program

```
/*
   Direct fourier transform
 * taken from http://local.wasp.uwa.edu.au/~pbourke/other/dft/
 * Removed sin/cos... so it doesn't actually compute DFT but something extremely close.
 * allows comparison with something DSPL is capable of (since we don't have sin/cos yet).
 */
#define FALSE 0
#define TRUE 1
int DFT(int dir,int m,float *x1,float *y1)
{
   long i,k;
   float arg;
   float cosarg,sinarg;
   float *x2=NULL,*y2=NULL;

   x2 = malloc(m*sizeof(float));
   y2 = malloc(m*sizeof(float));
   if (x2 == NULL || y2 == NULL)
      return(FALSE);

   for (i=0;i<m;i++) {
      x2[i] = 0;
      y2[i] = 0;
      arg = - dir * 2.0 * 3.141592654 * (float)i / (float)m;
      for (k=0;k<m;k++) {
         cosarg = k * arg; //cos(k * arg);
         sinarg = k * arg; //sin(k * arg);
         x2[i] += (x1[k] * cosarg - y1[k] * sinarg);
         y2[i] += (x1[k] * sinarg + y1[k] * cosarg);
      }
   }

   /* Copy the data back */
   if (dir == 1) {
      for (i=0;i<m;i++) {
         x1[i] = x2[i] / (float)m;
         y1[i] = y2[i] / (float)m;
      }
   } else {
      for (i=0;i<m;i++) {
         x1[i] = x2[i];
         y1[i] = y2[i];
      }
   }

   free(x2);
   free(y2);
   return(TRUE);
}
int main() {
         float x1[2048];
         float y1[2048];
         int i;

         for (i=0; i<2048; i++) {
                  x1[i] = (float) i;
                  y1[i] = (float) i * 1.0;
         }

         DFT(1, 2048, x1, y1);

         for (i=0; i<8; i++) {
                  printf("%d = %f, %f\n", i, x1[i], y1[i]);
```

```
        }
        for (i=2040; i<2048; i++) {
                printf("%d = %f, %f\n", i, x1[i], y1[i]);
        }
        return 0;
}
```

Equivalent DSPL

And now the equivalent for DSPL:

```
/**
 * @author: Big D
 * Fake DFT program. Add sin/cos --> =). Add complex/complex_array --> =) =) =) =) =)
 */
int i;
int k;
int m;
int dir;
float arg;
float cosarg;
float sinarg;
float_array[2048] x1;
float_array[2048] y1;
float_array[2048] x2;
float_array[2048] y2;

dir = 1;
m = 2048;

for (i=0; i<2048; i=i+1) {
        x1[i] = i;
        y1[i] = i * 1.0;
}

for (i=0;i<m;i=i+1) {
     x2[i] = 0;
     y2[i] = 0;
     arg = -1 * dir * 2.0 * 3.141592654 * i / m;
     for (k=0;k<m;k=k+1) {
        cosarg = k * arg; //cos(k * arg);
        sinarg = k * arg; //sin(k * arg);
        x2[i] = x2[i] + (x1[k] * cosarg - y1[k] * sinarg);
        y2[i] = y2[i] + (x1[k] * sinarg + y1[k] * cosarg);
     }
  }



  /* Copy the data back */
  if (dir == dir) {
     for (i=0;i<m;i=i+1) {
        x1[i] = x2[i] / m;
        y1[i] = y2[i] / m;
     }
  }

  else {
     for (i=0;i<m;i=i+1) {
        x1[i] = x2[i];
        y1[i] = y2[i];
     }
  }
```

```
for (i=0; i<8; i=i+1) {
          ccall printf("%d = %f, %f\n", i, x1[i], y1[i]);
}
for (i=2040; i<2048; i=i+1) {
          ccall printf("%d  = %f, %f\n", i,x1[i], y1[i]);
}
```

The above compiled sample programs output almost exactly the same results of.... For C:

```
0 = 0.000000, 0.000000
1 = 0.000000, -8572.362305
2 = 0.000000, -17144.724609
3 = 0.000000, -25717.082031
4 = 0.000000, -34289.449219
5 = 0.000000, -42861.796875
6 = 0.000000, -51434.164062
7 = 0.000000, -60006.472656
2040 = 0.000000, -17487624.000000
2041 = 0.000000, -17496186.000000
2042 = 0.000000, -17504760.000000
2043 = 0.000000, -17513336.000000
2044 = 0.000000, -17521916.000000
2045 = 0.000000, -17530470.000000
2046 = 0.000000, -17539056.000000
2047 = 0.000000, -17547618.000000
```

and for DSPL:

```
0 = 0.000000, 0.000000
1 = 0.000000, -8572.362305
2 = 0.000000, -17144.724609
3 = 0.000000, -25717.080078
4 = 0.000000, -34289.449219
5 = 0.000000, -42861.796875
6 = 0.000000, -51434.160156
7 = 0.000000, -60006.476562
2040  = 0.000000, -17487626.000000
2041  = 0.000000, -17496184.000000
2042  = 0.000000, -17504760.000000
2043  = 0.000000, -17513336.000000
2044  = 0.000000, -17521916.000000
2045  = 0.000000, -17530470.000000
2046  = 0.000000, -17539056.000000
2047  = 0.000000, -17547618.000000
```

The difference is most likely in GCC sometimes representing fp constants as integer expressions to simplify the assignment using `movl`, whereas DSPL follows ICL's habit of always creating double-precision FP constants.

Language Tutorial

The DSPL compiler is invoked by running the `dsplc` java compiled file. Put your dspl code into a file with .dspl extension, and run it through the compiler, as per the usage:

```
java dspl.dsplc [-c] [-t] inputFile outputFile
```

The options are as follows:

- -c: Invoke gcc and generate an executable program (named outputFile)
- -t: Display the parsed AST frame

If -c is not specified, assembly code will be output into the outputFile.

Language Manual

Lexical Conventions

A program consists of a series of tokens which are grouped together to create declarations and statements.

Tokens

Like C, there are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. White-space, which includes blanks, horizontal and vertical tabs, newlines, formfeeds, and comments, are used to separate tokens and are otherwise ignored.

Comments

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 character literals.

Identifiers

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

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

- `byte`
- `byte_array`
- `ccall`
- `else`
- `float`
- `float_array`
- `for`
- `if`
- `int`
- `int_array`
- `ubyte`
- `ubyte_array`
- `uint`
- `uint_array`
- `while`

Constants There are six kinds of constants, including signed and unsigned bytes and integers, strings, and floating point numbers.

Byte Constants Integer constants consist of a sequence of digits. The associated keywords are `byte` and `ubyte`. `byte` can be -128 to 127. `ubyte` can be 0 to 255.

Integer Constants Integer constants consist of a sequence of digits. The associated keywords are `int` and `uint`. `int` can be -(2^31) to (2^31)-1. `uint` can be from 0 to (2^32)-1.

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

String Literals

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. String literals are associated with no keyword, and are meant only to be used as arguments to ccall functions. A null byte `\0` is appended to the end so that programs can find the string's end.

Syntax Notation

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

Meaning of Identifiers

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

There are several fundamental types. Strings are any combination of characters except for the double-quote. 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.

Derived Types

Besides the basic types, there may also be arrays of any numeric basic type (`int_array, uint_array, byte_array, ubyte_array, float_array`). `byte_array` and `ubyte_array` must be declared with a length that is a multiple of 16. `int_array`, `uint_array`, and `float_array` must be declared with a length that is a multiple of 4.

Objects and Lvalues

An object is a named region of memory, whereas an lvalue is a name referring to an object.

Programs

A program is a list of declarations followed by statements:

```
program:
        (declaration)* (statement)*
```

Sub-programs are block of code nested in braces, used in loops and branching statements:

```
sub-program:
        '{' (statement)* '}'
```

Conversions

All operators will automatically convert their operands to be compatible with each other; explicit typecasting is not possible. For any assignment statement, all operands will be converted so that they are of the same type as the destination **before** any operations are performed. This excludes array operations as arrays cannot be converted.

Expressions

There are two types of expressions in DSPL, boolean expressions and numerical expressions. Numerical expressions are typically used in assignment statements and boolean expressions pertain to control flow and looping.

Boolean expressions in DSPL are comparisons between atomic values:

```
boolean-expression:
         unary-operator ("==" | "!=" | '<' | "<=" | '>' | ">=") unary-operator)?;
```

Both sides of the operand must be of the same numerical type.

Numerical expressions in DSPL are combinations of numerical operations:

```
numerical-expression:
         additive-operator
```

Because these expressions are only used in assignment statements, automatic type-casting is possible (all values are converted to the type of the value being assigned to before any operations). This excludes array operations as arrays cannot be converted. The handling of overflow, divide check, and other exceptions in expression evaluation will result in unpredictable results.

Additive Operators

The additive operators + and - group left-to-right.

```
additive-operator:
         multiplicative-operator ( ('+' | '-') multiplicative-expression)*
```

The result of the + operator is the sum of the operands. The result of the - operator is the difference of the operands. If performed on arrays, the two arrays must be of same length and type (no automatic type-casting), and the result is an array for which each element is the result of the operation on the corresponding elements of the original arrays' elements.

Multiplicative Operators

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

```
multiplicative-operator:
         parentheses ( ('*' | '/') parentheses)*
```

The operands of * and / must have arithmetic type or array of arithmetic type. The binary * operator denotes multiplication. The binary / operator yields the quotient; if the second operand is 0, the result is undefined. 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' elements.

Unary Operators

Expressions with unary operators group right-to-left.

```
unary-operator:
         '-' (int-constant | float-constant)
         | '+' unary-operator cast-expression
         | '<code>byte_array</code> and <code>
         | value
parentheses:
     '(' numerical-expression ')'
     | unary-operator
```

The unary negation `-` operator only works on constants.

Atomic Values

The types of lvalues and values are:

```
value:
         lvalue
         | int-constant
         | float-constant
         | string-literal

lvalue:
         identifier ( '(' numerical-expression ')' )?
```

lvalues can thus either be single numbers, strings, whole-arrays, or elements of an array.

Declarations

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:
         ("int" | "uint" | "byte" | "ubyte" | "float") identifier ';'
         | ("int_array" | "uint_array" | "byte_array" | "ubyte_array" | "float_array")
'[' int-constant ']' identifier ';'
```

Note that values can not be initialized in the declaration and that all arrays must have their sizes declared.

Statements

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:
         assignment-statement
         selection-statement
         iteration-statement
         ccall-statement
         | ';'
```

Assignment Statement

The assignment assigns values to the object an lvalue refers to.

```
assignment-statement:
         lvalue '=' numerical-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 do not necessarily have to have the same numerical type unless an array is being assigned. Arrays must be of the same length.

Selection Statement

The selection statement choose one of several flows of control.

```
selection-statement:
         "if" '(' bool-expression ')' subprogram ('else' subprogram)?
```

In both forms of the `if` statement, the boolean expression is evaluated and if it compares true, the first substatement is executed. Optionally, if there is an else, the second subprogram is executed if the expression is false.

Iteration Statement

The iteration statement specifies looping.

```
iteration-statement:
         "while" '(' bool-expression ')' subprogram
         | "for" '(' (assignment-statement (',' assignment-statement)*)? ';' bool-
expression ';' (assignment-statement (',' assignment-statement)*)? ')' subprogram
```

In the `while` statement, the substatement is executed repeatedly so long as the value of the expresion remains true. The test, including all side effects from the expression, occurs before each execution of the statement. The `for` executes its subprogram while the bool-expression evaluates true. The first set of assign-statements are executed prior to its first evaluation, and the second set are evaluated afer every execution of its subprogram.

C Call Statement

The C call statement calls libraries from C:

```
ccall-statement:
         "ccall" identifier '(' (unary-operator (',' unary-operator)*)? ')' ';'
```

It should be noted that only atomic values can be passed to a ccall function.

Project Plan

In this section we will explain the development of DSPL as a team project.

Processes

The process of creating DSPL was a long and detailed process. The core of the language revolved around automating and optimizing arithmetic operations on sequential elements of an array. First, a detailed analysis and research of Intel's current instruction set architecture was performed. Then, it was necessary to learn x86 assembly syntax, and generate simple programs utilizing the optimized x86 instructions for array operations. With the types of array operations and the necessary assembly output required in mind, the actual features of the language were then created.

Planning

We decided early on that we wanted to create a compiler that made use of Intel's advanced SSE instructions. We decided to start off with a limited version of C that would provide some additional functionality, such as a new type, `complex`, as well as the ability to have mathematical operations work with entire arrays as operands. We would also add the convolution operator ~, which would take arrays as operands. However, as we learned more about compiler design during the course, we soon realized that our initial plans were a bit too ambitious. In order to maintain SSE functionality, we decided to further scale back some features of C that were not absolutely necessary, such as AND/OR operators.

Specification

We captured the initially-planned specifications in the first round of our Language Reference Manual. However, it evolved as work on the project continued. The general rule in choosing language features was that if it was possible to maintain functionality, it was acceptable to eliminate certain features. Examples of this are using `&&`/`||` in if statements, instead of just listing multiple if statements and repeating code. Another example was `i++`. The major factor deciding language features was intended/hoped use for the language. Images are typically unsigned byte arrays, thus we absolutely needed unsigned and byte sized support. FFTs typically operate either on signed integer or float arrays, thus the need for both integers and floats. At one point, it was decided to drop FP-element (non-array) operations, but after a refresher on DFT/FFT code snippets, it was quickly realized that array FP support alone was not enough. Also, originally conversions were specified, but later removed due to complexity. Eventually it was deemed absolutely necessary, however, again due to implementing a DFT. The final specification actually added a lot more functionality than was removed.

Development

Development of DSPL was separated into three architectural components: the front-end, intermediate stage, and back-end. The front-end was created using ANTLR, which generated an abstract syntax tree. The intermediate stage, consisting of a TreeWalker also created with ANTLR, converted the output of the front-end into DSPL's intermediate format. The back-end finally took this representation and converted it to assembly code. The compiler would generate a GCC-friendly AT&T-style assembly (*.s) file, which could finally be compiled with GCC into an executable program.

The intermediate representation, consisting of Instruction types and DSPL Variable types was created first. The intention was to create a solid and complete IR, before commencing on the backend or front-end. The IR represented a contract by which the front and back-ends had to adhere to. This drastically reduced integration efforts between the front and back ends, as the output and input, respectively, of each was already set. The most vigorous argument and type checking was done in the IR.

Once work started on the backend, it became clear that certain operations, mainly the formatting of variables for use in asm instructions, would be required extremely often, and sometimes required very complex formatting, depending on the situation. In order to solve this, a paradigm similar to the one used in the IR was applied, but only done for the Variables. Four main X86 Operand types were created: registers, immediates, memory, and indexed memory elements.

Registers corresponded to actual X86 registers, including both the 8 general purpose registers, as well as the newer 128-bit SSE registers. Immediates were constants that could be supplied directly within the instruction, for example `movl $5, %eax`, which would set the value of the `%eax` register to `5`. Memory operands were where the memory location could be represented with a single variable, as in `movl intA, %eax`, where `intA` is the memory location of an integer value.

The most complicated operand format, indexed-memory locations, was the primary reason for the creation of the backend operand format. When accessing elements in an array, one the recommended x86 format is some variant of `movl 0(%eax,%ebx,4), %ecx`, where the contents of the memory location stored at `(0 + %eax + (%ebx * 4))` is moved into register `ecx`. This may seem unnecessary, but given the focus of our project on optimizing array operations, having a method of efficiently accessing their elements was absolutely necessary.

Testing

Testing, like the specification process, required consistent adaptations to the dynamic implementation of features. Test cases were created that tested both the functionality, or things that should work, and syntax errors, or things that shouldn't work. The testing script was written in Python and allowed the creation of suites of tests. Every time a new feature was added, test cases were created, and if a feature was changed, the corresponding test cases were as well.

Programming style guide

The two main restrictions everyone had to adhere to when programming were to use javadoc-style comments so a comprehensive javadoc could easily be generated, and to adhere to the class structure we had agreed on for each of the various parts.

Project Timeline

- 2006-10-15: Complete initial specifications
- 2006-10-31: Complete initial front-end
- 2006-11-15: Complete full compilation of basic Hello World Program
- 2006-11-31: Complete full compilation of 2nd Hello World Program
- 2006-12-05: Begin active testing phase
- 2006-12-15: Complete final specifications and almost all implementation aspects
- 2006-12-18: Complete project

Roles and Responsibilities

- Jeffrey Cropsey: Intermediate Stage (AST to IR and Semantic Analysis)
- David Lariviere: Back-end, Team Leader
- Michael Lynch: Testing
- Varun Maithel: Documentation
- Varun Mehta: Front-end

Software Development Environment

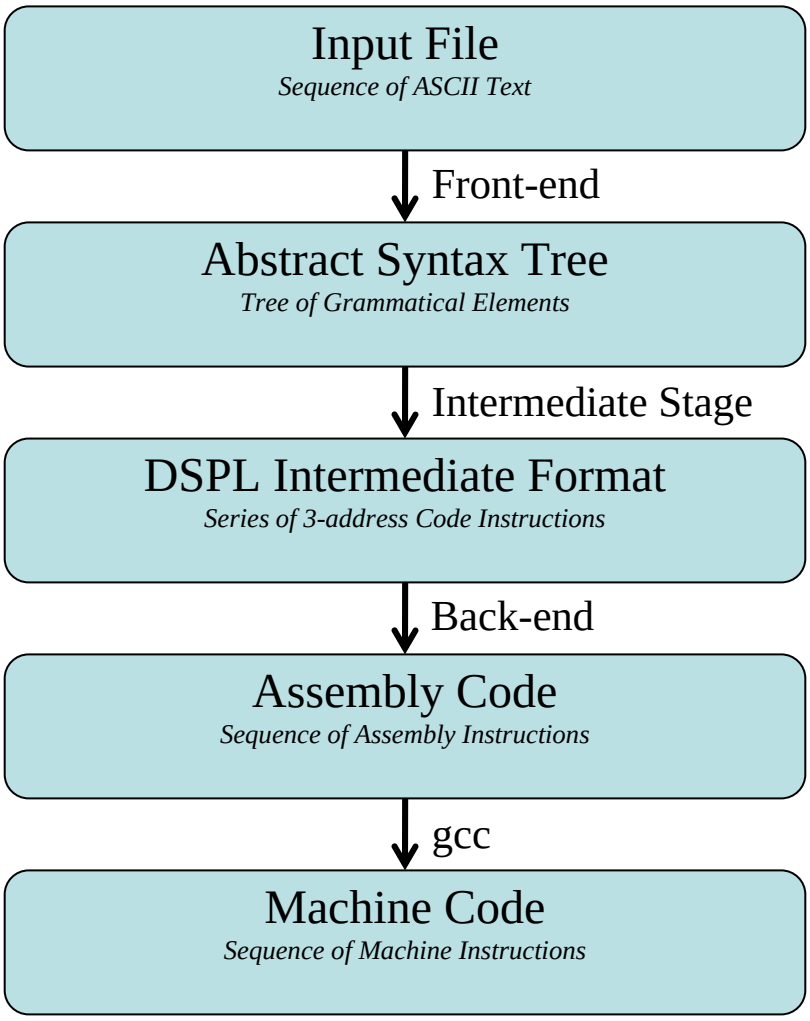
The team wrote most of the compiler for DSPL in Java, utilized ANTLR in the front-end to generate the parser and lexer Java code, and used Python in the testing scripts. The IDE used was Eclipse, and CVS was used to manage versioning.

Project Log

- 2006-09-13: Wrote first ASM program, queries CPUID string of the compiler, compiled on Fedora Core 2 Linux box
- 2006-09-20: Determined necessary changes to compile ASM programs in Windows using Cygwin
- 2006-10-05: Completed Regression tester script
- 2006-10-09: Finished writing complete ASM program to multiply two pairs of complex numbers utilizing SSE3 and print the result.
- 2006-10-19: Intial LRM Completed.
- 2006-10-23: Front-end creates ASTs
- 2006-11-01: 20 test cases created for regression suite
- 2006-11-12: Intermediate Format (Instructions/Variables) completed
- 2006-11-13: Backend Infrastructure completed. IR of HelloWorldv1 compiles sucessfully
- 2006-11-18: Initial tree-walker created.
- 2006-11-29: Completed work on compiling ASM modules and calling from c programs
- 2006-12-04: HelloWorldv2.dspl fully compiles and properly runs
- 2006-12-06: Regression Tester up and running utilizing functional compiler
- 2006-12-08: Indexing into arrays completed
- 2006-12-09: Regression Tester modified to recognize error codes, allows single source file tests
- 2006-12-16: Final specifications updated in LRM
- 2006-12-17: Over 100 test cases
- 2006-12-18: Project completed
- 2006-12-19: Final report submitted

Architectural Design

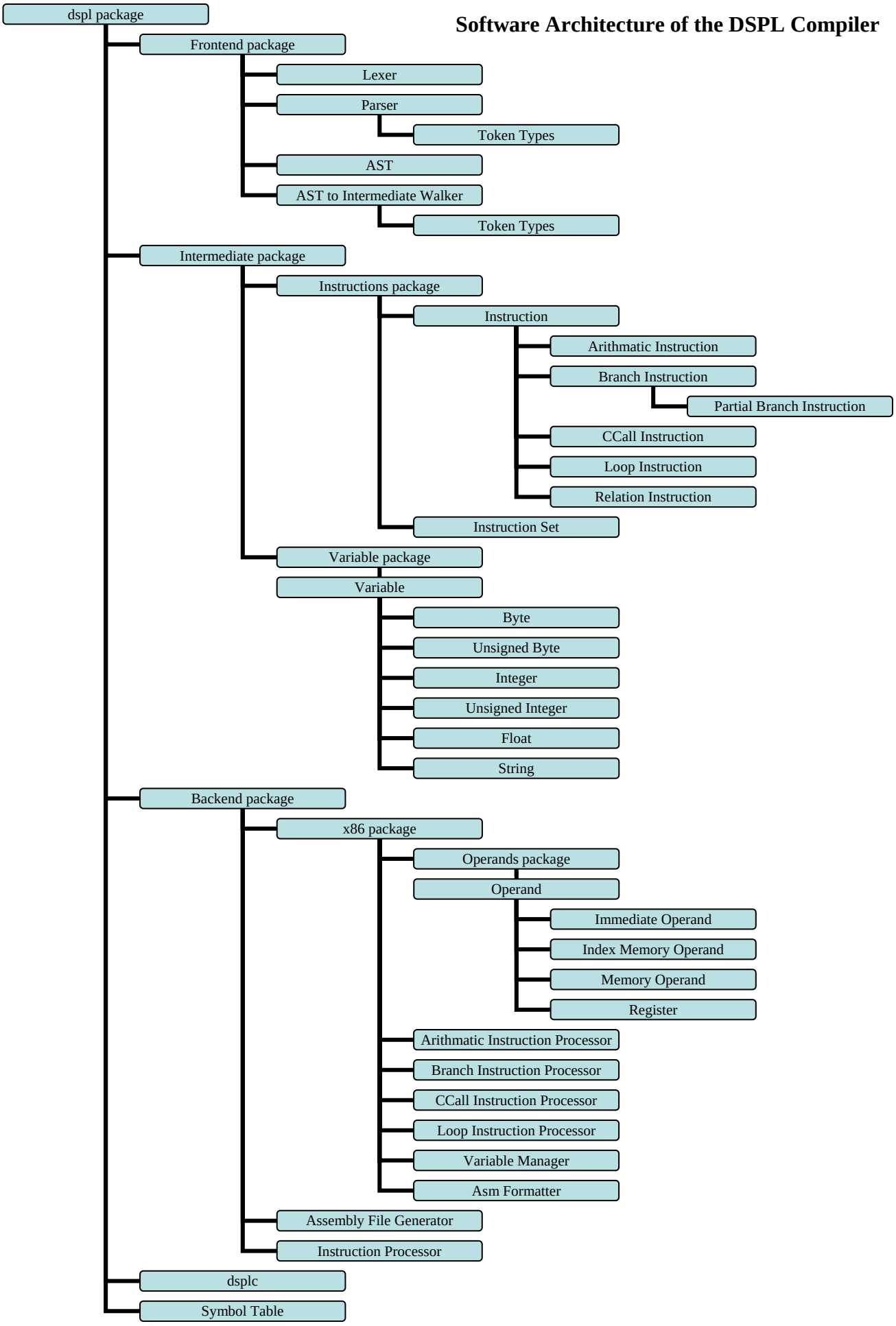
The process of compilation in DSPL can be broken down into three main stages: front-end, intermediate, and back-end.



The front-end parses the input file and identifies tokens, creating an abstract syntax tree. During the intermediate stage, a tree-walker converts the abstract syntax tree into DSPL's intermediate format, which is described in further detail below. Finally, during the back-end stage, assembly code is created from the intermediate representation.

As there are three principal components, there are two corresponding interfaces which are the output of one and the input of the next: the abstract syntax tree and the intermediate format. The abstract syntax tree represents the program as a hierarchy of grammatical elements. DSPL's intermediate format is a simplified version of the abstract syntax tree, in that the hierarchy represented is of 3-address code statements that are analogous to assembly instructions. Lastly, the back end consists of a series of interfaces and x86 implementations responsible for processing each IR instruction type and generating the appropriate assembly code.

Software Architecture of the DSPL Compiler



Implementation credits:

- front-end (Grammar/Treewalker): Varun Mehta
- intermediate (Treewalker): Jeff Cropsey
- back-end (IR/IR->x86): Dave Lariviere

Test Plan

We used Python to write several utilities that maintained a suite of test cases monitoring the stability of our language. The most important was a regression tester, which compiled and executed code from our suite of test source files and compared the results to expected output. In cases where the source code was expected to be rejected by the compiler following semantic analysis, the expected output was an error code corresponding to the error that the compiler should have returned upon exiting. In cases where the source was expected to compile and execute successfully, the expected output would be the output that program should generate (determined either by human inspection of the code or output of an equivalent C program).

For each source file, *filename*.dspl, processed by the regression tester, 2-4 output files would be created:

- *filename*.log - Compilation log containing the command used to execute the compiler as well as its output
- *filename*.s - Assembly code generated for the source file
- *filename*.exe - Compiled executable for the source file (if semantic analysis was successful)
- *filename*.out - Output of the compiled program upon successful compilation and execution of the source file

The regression tester had three different modes of operation:

- single file - Perform a test on a single specified source file
- file suite - Perform a test on each file from a list of source programs from a plaintext file
- global - Perform a test on each file in the test folder

Test Suites

As test cases were written, they were added either to the "syntax" suite, for files that are geared toward testing elements of the compiler's semantic analysis, or the "functionality" suite, for files that test the actual execution of the program once compiled. Once a file was working correctly in the current build, it was added to the "working" suite, which contained all files that compiled and executed successfully. The working suite was very important, as no changes could be committed to the repository unless it passed every test in the current working suite.

The test cases written were each designed to test an isolated feature of the language so that it could be clear to the programmer which compiler features were failing without having to pore through lines of a test case to find specifically where an error occurred. The entire collection of

test cases contains over 100 different source files, which can be found in the appendix. The collection of tests is intended to cover every significant feature of DSPL, though our most aggressive and exhaustive testing was on numerical operations including arithmetic on both atomic and array variables.

Testing Scenarios

In testing our source files, the results would fall into four different scenarios, each of which the regression tester would inform the user of.

Scenario 1: Program output matches correct output

When a source file was compiled and executed successfully, its output would match the correct expected output and the regression tester would report this. In the case of our test case comment2.dspl:

```
/**      Tests different types of comments **/
ccall printf("this line should be printed.\n");
//ccall printf("this one shouldn't be\n");
ccall /* some people think comments are unnecessary */ printf("but this one should
be\n");
/* ccall printf("this is a comment and should be ignored by the compiler\n"); */
ccall printf("the beginning of this line should be printed..."); // ccall printf(" but
not the end");
ccall printf("\n");
```

This program is syntactically correct and should compile with the expected output:

```
this line should be printed.
but this one should be
the beginning of this line should be printed...

```

Since the actual output matches the correct output, the regression tester would output

```
Running test for `testcases\comment2.dspl': OK!
```

Scenario 2: Program output does not match correct output

If the compiler had an error that caused the previous program's output to be the following:

```
this line should be printed.
an erroneous line!
the beginning of this line should be printed...

```

The regression tester would reflect this in the test and display the part of the output that differed (with the verbose flag):

```
Running test for `testcases\comment2.dspl': MISMATCH - WRONG OUTPUT
  `testcases\correct\comment2.out' != `test_output\_12-19-2006_14-33-15\comment2.out'
         2c2
         < but this one should be
         ---
         > an erroneous line!
```

Scenario 3: Invalid program caught by semantic analysis

In cases where we expected the source to fail semantic analysis, the expected output file would contain only an integer representing the expected error code on which the compiler is expected to exit. The following program should be rejected by the compiler since it incorrectly assigns a variable in its declaration.

```
/* tests immediate assignment of a variable */
int i = 5;
ccall printf("This line should not print\n");
```

If the compiler rejects this program and exits with the proper code, the regression tester would output:

```
Running test for `testcases\assignment2.dspl': OK!
```

Scenario 4: Invalid program incorrectly passes semantic analysis

If the compiler contained an error that allowed the previous program to be compiled successfully, the regression tester would notify the user with the following output:

```
Running test for `testcases\assignment2.dspl': MISMATCH - UNEXPECTED COMPILE
```

Credits

The regression tester and the majority of test cases were created by Mike Lynch. The rest of the group contributed some test cases as well.

Lessons Learned

Individual Insights

Jeffrey Cropsey

I can confidently say that “starting earlier” would not have helped us in this endeavor, as we began right after our group was formed. However, there were some things that would have made our compiler design move much faster in the end if they were present in the beginning. The language was in flux for a while after the initial specification as we learned that certain features would be more complicated than others and that some features, while they might add finesse (like the ++ operator) didn’t add new functionality. The flux of the language caused a several changes in the grammar and tree walker, the components on which I worked.

The introduction of a strong IR helped to reduce the complications that grammar changes made to the walker. Nevertheless, when my focus changed from walking to semantic analysis, the problems caused by the changes returned. One specific example was in the implicit conversion of types in an expression. We chose to convert all values to the assign type before evaluation of the expression. I added functions to the walker to create the proper assignment and conversion instructions. However, after a few days, the analysis would break with the addition of a new type that started working, such as the addition of floating point support. On the other side of things we dropped support for complex numbers, but many other files relied on the definition of a complex data type to be present. Not having the time to go back and remove it at all stages, we had to catch the declaration of such variables in the walker.

There are three things that would have made the job of walking and semantic analysis easier for our group and me. First, taking more time in the beginning to more deeply evaluate exactly what types and operations and their combinations could be in the language would have fewer changes to the grammar necessary in the long run. In addition to this longer and more through definition phase, creating a well defined way to add and remove functionality that might fall “on the bubble” of being in or out of the language so its addition or removal would not create as much of a ripple in the code. Second would have been the addition of helper classes in semantic analysis. Because of the way functionality came online, each time there was a new check it was only “a few more lines of code.” Instead, having an assistant to the walker that encapsulated all of these little things would have made the walker cleaner to operate on in later stages. Finally, if time had allowed, it would have been helpful to build a java backend to the language early on while the assembly back end added functional support. In this way, we could have included support to visualize the IR in addition to test the walker’s ability to produce the IR for all types and features before assembly support was added. Such a backend would further separate dependency on the assembly backend for testing.

In all, it was quite exciting to build a compiler for the class. I am sure that some of my wishes (like the java backend) could take too much time to implement and cause problems of their own. Even when starting early, one is limited by other classes and the length of the semester. I know

however that even though this project was a toy compared to the “real world” it still represented the most complex system many of us have designed so far and the experience will be applicable as the scale of systems we develop increases. I would be excited to see, given another semester and an overhaul to the front end as described, how far our little language of DSPL could go.

David Lariviere

I learned so much while doing this project. My main hope was that I would finally learn assembly. I have made many brief attempts over the years, but could never quite get myself to spend the time to learn it sufficiently. After doing this project, however, I feel quite capable programming in x86 assembly, using a variety of addressing formats, data storage sizes, and instruction types.

Doing this project also helped get me much more familiar with the Intel's x86 architecture. For the last few months the Intel Manuals 1,2a,2b,and 4 have served as my bible and nightly reading of choice. I think the biggest surprise in doing this project and taking PLT was in gaining a new understanding of the impact that compilers and ISAs can have on a particular program's performance. I previously felt it was simply a matter of gaining a factor of 4 utilizing 4-element vector optimizations. Now I feel that entire orders of magnitude can be gained or lost, depending on precisely how intimate one is with the targeted machine architecture and the code that is generated on it. I never considered the importance of memory alignment of variables or the stack before, or even the possible importance of the placement of code within memory to improve the reliability of cache hits.

In doing this project, I saw first hand the importance of creating exact specifications in between the components that different people would be responsible for working on. I am so glad that I created the Intermediate instructions and data types before hand, rather than trying to do so in parallel with the other work being done. It was done early enough in the semester (about half way through) that corners didn't have to be cut, and there was sufficient time to completely think through and develop the entire package. Creating it made integration problems between the front and back ends non-existent, as both sides knew exactly what they had to adhere to and what was expected to be implemented. The IR was the result of long planning and discussion of exactly what features and data types our language would support. Doing the IR in an extensible format using base abstract classes and interfaces also has made it possible to include and remove certain features with vastly less overhead than if everything was hard-coded into the respective front and backends. It was designed with the intention that down the road, additional architectures could be supported, like X64 or the Cell.

Working on DSPL's IR has made me seriously think about the importance and impact of the choice of IR in modern day compilers. Historically, it seems that many compilers generated IR that was simply too close to a specific machine instruction set, and also removed too much of the high level functionality that was actually being represented. The variety and differences between modern CPUs, even in the same basic ISA, let alone across different types, makes instruction/register level IRs seem quite out of date. Especially as the multicore explosion begins to occur over the next few years, the importance of having IRs which are capable of capturing

high level desired functionality, rather than just low level manipulation of a machine state, will grow. Examples of this include loops and array operations. In taking a low-level IR approach, certain optimizations become vastly more difficult than if a higher level IR is utilized, which provides the backend with a greater understanding of the context in which certain instructions need to operate, and thus allows for a much greater variety of optimizations and hand-tuning for the high-level feature (like array arithmetic operations) than is otherwise possible if a backend compiler is forced to interrupt a low-end result and recognize patterns in the lowlevel IR which might indicate the high level feature. Another example of this is with synchronization and multithreading. A low-level IR completely obscures the fact that a particular cmp/branch instruction is indeed a synchronization lock waiting for another CPU or core to change the value of an integer. Depending on the ISA, certain instructions with vastly different performance impacts are available for implementing the same low-level IR, depending on the precise context.

Another area related to IR design is language design and methods of informing the compiler of the desired results. I had never really been exposed to compiler extrinsics before doing this project and researching implementations of modern compilers to try to optimize some of the situations we handle.

In working on DSPL, I also had the chance to become more familiar with the way other compilers were implemented. Using GCC and Intel (ICL) to compile example code snippets and view their resulting assembly code made the project much easier to implement. It also provided ideas for areas of improvement where modern compiler optimizations might fall short for particular instructions.

The project was extremely big, and while we didn't quite reach where I had hoped, I feel that the current state of the code is close. We had originally dropped many things that we later decided to go ahead and implement, the biggest of which being floating point and arrays for all data types. The project was intentionally designed to be quite more extensible than required for meeting the particular target for the end of the semester. In the coming months and years, I hope I will find more time to continue to play, work on, and improve DSPL. My goal is to eventually have it as a tool not for general purpose programming, but specifically just for generating absurdly optimized highly-platform specific code snippets to be linked in with programs written in general purpose languages. While it isn't there yet, one day DSPL shall p0wnz GCC/ICL, in its specific area of purpose >:}

Other quick lessons learned: the alignment issue mentioned in our final presentation turned out to be a bug in binutils (gas) for cygwin... :*(.

Also, we learned the "joy" of Floating point fun and all its consistencies....

Michael Lynch

The most important lesson I took from this project is that organization is paramount in program testing. In order to do effective, rigorous testing, one must carefully plan a procedure for creating

test cases and maintaining their correspondence with a correct output. A clear and organized system for maintaining our test files, their correct output, and files generated at compile-time and run-time made greatly facilitated the testing process and our ability to recognize quickly where bugs were occurring in our compiler.

I believe an incorrect strategy decision I made in testing was writing testcases by hand rather than writing a program to generate them automatically. I made the decision to manually write test cases based on the fact that so many of the boundary cases are very specific and would complicate the case writing if it was done through a script. Later in the testing, I found that this consideration was still valid, though a code generator would have the added benefit of being able to easily write C code equivalent to the DSPL code and automatically generate the output as well. A code generator would also have been beneficial in testing combinations of features, as the custom boundary cases are repeated over and over again and could easily be combined with one another by a code generator to ensure that features don't produce unwanted results when used in conjunction.

Varun Maithel

As far as documentation goes, don't take it lightly. Make sure that you keep up-to-speed with the latest changes that each group member is working on, add/change the appropriate documentation, and let everyone else know. The most difficult aspect of working in a large, complex project with interdependent parts is maintaining continuity between the different modules that each group member is working on. Fortunately, we quickly created the architecture for the entire compiler from the beginning, so our progress wasn't slowed by these difficulties. Other than that, documenting the project was a smooth process without any problems or unforeseen difficulties.

Varun Mehta

Overall my experiences in this group were very positive. Though we divided up tasks into blocks, we were quite flexible about moving to different modules as needed. The end result is a quality product that I am proud to have contributed to. Through my experiences I have gained additional valuable skills in teamwork and coordinating my own activities to meet some strict deadlines. We created an effective timetable and committed to “development contracts” which solidified the interfaces between different sections of our project. In this way each section could be independently developed. Communication was still extremely important, and to that end the creation of both a mailing list and a wiki allowed us to discuss and document our work so that everyone could be kept up-to-date. Without all of these support mechanisms in place, there would have been no hope of accomplishing everything that we were able to in merely a few months.

Advice for Future Teams

- Don’t make your grammar contrary to a CS major’s muscle memory (if you have to, make it *very* different)
- Test cases should be automatically generated, as well as correct output, based on several different compilers, and with different flags set
- Having weekly meetings is a good way to make sure everyone is staying on track and knows the status of the project and where they should be in their part for it.
- Start learning and writing assembly programs early. Implement from scratch example versions of programs directly in assembly in order to gain an understanding beforehand of all of the tasks required to generate the ASM.

Code Appendix

See code attached.