PLUM

The University of Maryland PL/1 System

by

Marvin V. Zelkowitz
Department of Computer Science
University of Maryland
College Park, Maryland

Computer Science Center
University of Maryland
College Park, Maryland

Technical Report 318
July, 1974

Abstract

PLUM is a load and go PL/1 compiler for the Univac 1100-series computer. It generates absolute machine language and then immediately executes the program. PLUM is heavily instrumented and provides much useful information about program behavior both during and after program execution.

This report describes the organization of the PLUM compiler and describes some of the algorithms and data structures used. It also explains how some of the diagnostic information is produced.

NOTE: this document, except for this note, is an exact copy of the technical report which was issued in July, 1974. this document describes version 1:06 of PLUM, not version 2:02 which is the current version. The basic change in the compiler is that all characters are now 9 bit ASCII characters instead of 6 bit fieldata characters. Other features have been added, but they do not change the fundamental structure of the compiler.

Introduction

Compilers were typically thought to be simply computer programs that translated a program written in some language into efficient machine code. With the relatively low costs of modern hardware and high programmer development costs, this assumption is not necessarily valid. Compilers should be valuable tools that assist the programmer in developing software. It is for this reason that PLUM was written. PLUM is a compiler for the PL/1 language that is designed to run efficiently and produce output that will greatly lessen the task of writing PL/1 programs. PLUM is designed to execute on a Sperry Univac 1100-series computer under the EXEC8 operating system.

This document is divided into two parts. the next chapter is designed to give the casual PL/1 programmer some insight into the structure of PLUM and to give the compiler writer some ideas of how various problems were solved in the PLUM implementation. The remainder of this document contains more detailed information about the data structures, the internal structure and implementation techniques used in writing PLUM.

PLUM is written in a pseudo implementation language, called PIT, which is implemented as procedures for the 1108 assembler. PIT is character and string oriented, so it removes many of the considerations of word boundary problems inherent in the word oriented 1108. Chapter 5 is a detailed description of the PIT language.

It is assumed that the reader has a familiarity with the PL/1 language. If not, it is hoped that the

following sketchy description will be sufficient in order to read the remainder of this document.

PL/1 is a block structured language that was defined by the IBM corporation in conjunction with the SHARE organization. At the present time, an ANSI standard PL/1 is being defined [ANSI, 1973]. PL/1 has two primitive forms of data: strings and arithmetic variables (PLUM makes this distinct - ANSI PL/1 does not).

Strings consist of character strings and bit strings. they may have fixed lengths or may be varying length (up to a declared maximum value).

Arithmetic variables have 4 attributes. they may be real or complex, they may be fixed point or floating point, they may be decimal or binary, and they have a precision (a certain number of digits of significance).

Arrays and tree-structured data structures may also be declared. Leaves of the structure will be either string or arithmetic variables.

Most data is of type automatic which means that space for the variable is allocated on block entry. Because of this, array bounds and string lengths may be executable expressions. Static storage (allocated at entry to program execution), and controlled storage (allocated explicitly by the user) are also available. Pointer variables and based storage also exist, although are not in the PLUM subset.

All parameters to subroutines are call by reference.

The control statements include the iterative do (`do i=1 to n`), logical do (`do while(a>b)`), the if (`if b<a then x=1; else y=73;`) and the usual assortment of statements such as assignment, begin, end, call, return and I/O statements.

General Structure of PLUM

Compiler Organization

PLUM is a load and go PL/1 compiler for the Univac 1100-series computer. It is designed as a diagnostic aid in program development and maintenance. Because of this, it has the following attributes:

1. Extensive diagnostic messages provided during program compilation and program execution.

2. Extensive data collection facilities for monitoring program behavior.

3. Very short compilation times for rapid inexpensive turnaround of debugging runs. (PLUM compiles about 10,000 average statements per minute on an 1108.)

PLUM generates absolute machine code and then immediately executes the program. No relocatable output is produced which means that no link edit step is necessary.

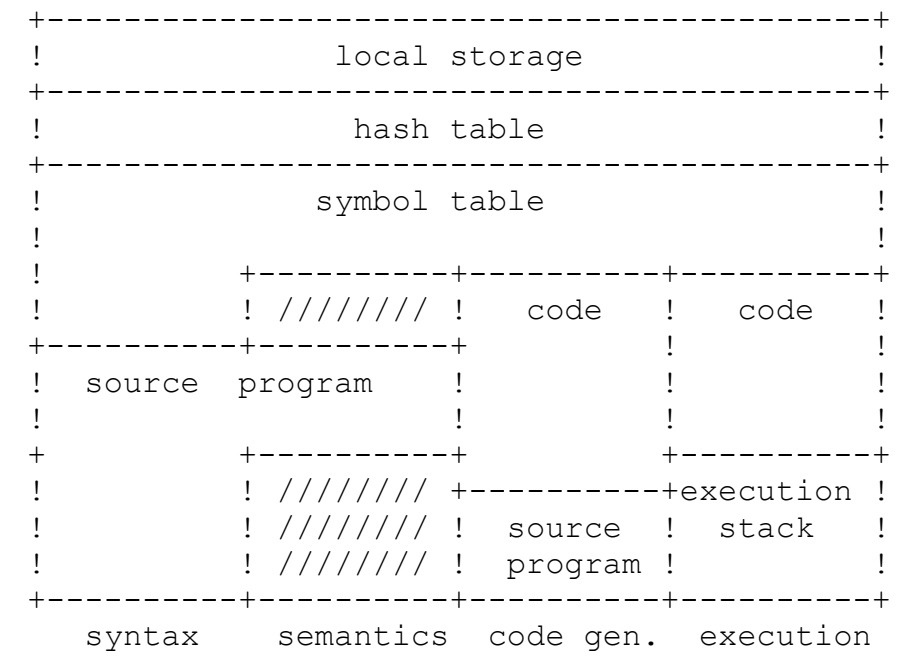
PLUM executes as a multisegmented program in the instruction bank (I bank) of the 1108. A root segment is always present while each compilation and execution phase overlays the previous phase in memory. Passes 1 and 2 constitute syntax analysis and are contained in one phase; the cross reference listing is a second phase; code generation is a third and execution is a fourth phase. The data area for all phases is a 10,000 word workspace (D bank) called PLUM common.

(This author spent several years working on Cornell University's PL/C project before coming to the University of Maryland. Because of that association, the general structure of PLUM, as well as the nomenclature used, is similar to Cornell's load and go PL/1 compiler [Conway, 1973].)

General Data Flow Through PLUM

In order to minimize I/O calls, all data is core resident. This means that the symbol table is always resident as well as the parsed program after syntax analysis and the object code after code generation. While this core residency does limit program size, it will shortly be shown that the theoretical limits are well beyond the practical range for program size.

As stated before, all data is kept in the 10,000 word PLUM common workspace. The organization of this workspace during each phase is as follows:



The first 800 words contain local storage that is used by each phase and the next 1024 words contain the hash table. Each variable name hashes to a unique location within this table. The table contains the address of the actual symbol table entry. (each entry in the hash table is actually a halfword pointer. the other halfword is used as data storage for the various phases.)

During pass 1, the source program is read and the symbol table is constructed for each identifier. the symbol table starts immediately after the hash table in memory. Pass 1 also constructs an internal form of the program. This internal form starts (arbitrarily) halfway down in the workspace. by the end of pass 1 both the symbol table and internal source table sizes are fixed.

Pass 2 does not change the size or location of either the symbol table or the source program table.

Just prior to code generation the internal source program is moved to the end of the workspace. This frees up the largest block of memory possible between the end of the symbol table and the start of the internal source program table. As the source program table is scanned, the object code is placed in this freed block. Also, as the source program is scanned, additional space is freed. Once code is generated for a statement, the space it took in the source program table is released.

Finally, after code generation, the remainder of the workspace serves as the run-time stack for the executing PL/1 program. Notice that in this case, both instructions and data reside in the workspace (D bank) of the 1108, thus execution is not as efficient as possible given the organization of the 1108 memory; however, it is in keeping with the PLUM philosophy of minimizing compilation times at a slight cost in execution time.

The size of the workspace has been mentioned to be 10,000 words. If the user specifies an option (M option), then before syntax analysis this size is increased to 16,000 words. This size increase can easily be altered. The actual limits within the compiler are:

1. Symbol table addresses must be less than 200,000 (octal) (or 65,536 decimal) which leaves a maximum symbol table of 48,000 words (assuming the first 16,000 addresses of the virtual memory to contain the instruction space). since each variable name hashes to a unique address, there are at most 1024 different names possible in a PLUM program. however, since PL/1 is a block structured language, more than 1024 different variables may be used since the same name may be used more than once.

2. Internal source program addresses must be less than 400,000 (octal) (or 131,000 decimal) which leaves 65,000 words available for program storage (assuming a maximum size symbol table).

3. The limit on generated code and run-time stack is 262,000 words, which is the maximum memory size in the 1108; however, the code to generate jump instructions must be altered if the addresses reach above 65,536. this will require a minimal change to one module of the code generator.

4. Without change PLUM can run in 65k words and process programs with 700 variables and 3000 statements. with minimal changes, these limits will enable PLUM to process programs that have approximately 7,000 statements with 2,000 variables - an order of magnitude greater than the largest programs for which it was designed.

Syntax Analysis

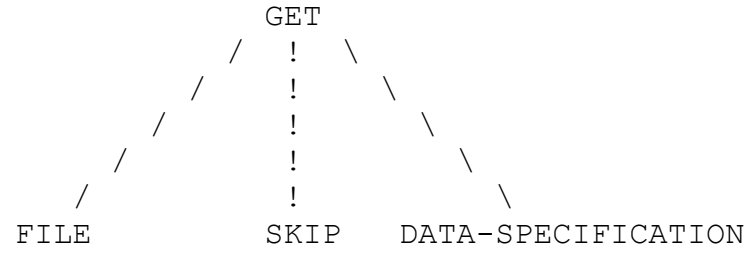
The first pass of PLUM is syntax analysis which reads in the source program and constructs both the symbol table and internal form of the source program.

Beta Code

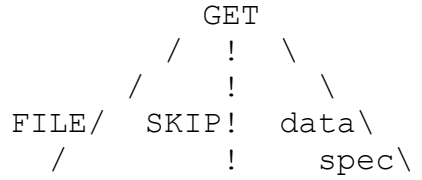
The internal source program (called beta code) is essentially a tree structured abstract syntax of the program. For example, the GET statement consists of the following four parts:

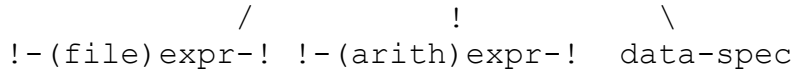
1. The keyword GET
2. The file phrase
3. The skip phrase
4. The data specification

While the keyword get must appear first, the other three parts are optional and can appear in any order. the only restriction is that one of the last two phrases must appear. The get statement can therefore be represented by the following tree:



Each of the three leaves represent a type of expression. The leftmost expression must be a file, the second can be any expression that results in a scalar arithmetic value and the third is actually a subtree consisting of the definition of the specific data specification that represents the statement. This depends upon whether the data specification is for a DATA, LIST or EDIT specification. Thus the GET statement can be represented by the following modification to the above tree:

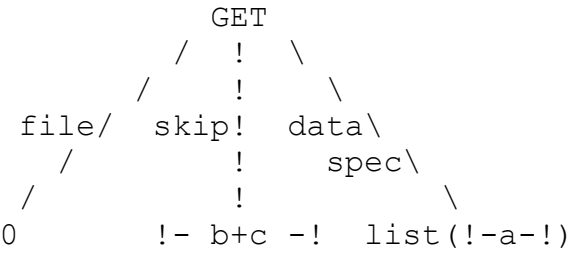




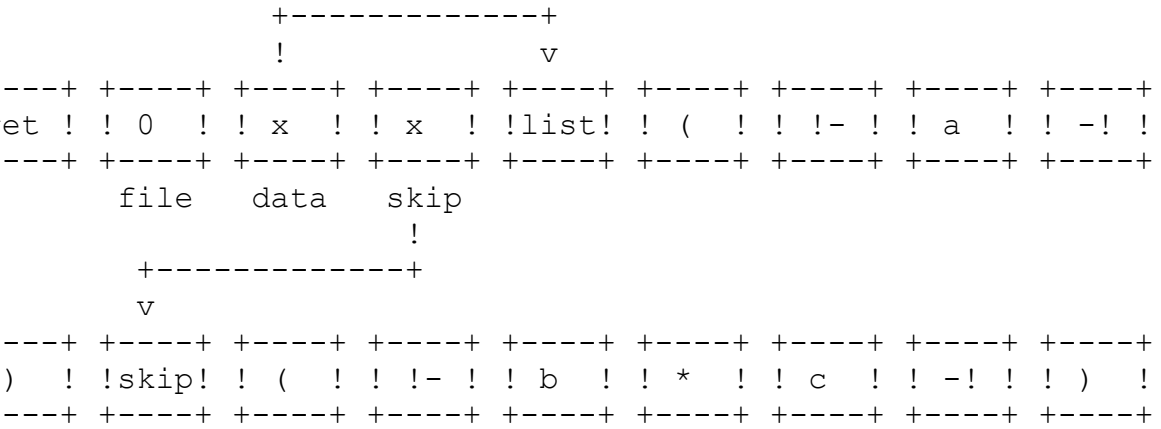
This is the form actually used in PLUM. for example, the statement

```
get list(a) skip ( b*c );
```

Would be represented by the tree



Or in the linear form



Where tokens are packed two per word. each special symbol (get, list, skip, etc.) has a unique token value while each identifier is a symbol table address.

Preceding each statement is a token stating that it is the start of the next statement, and gives the statement number. Statement prefixes also precede the above set of tokens for the get statement.

Structure of Syntax Analysis

Syntax analysis consists of a three level structure. At one level a lexical scanner reads in the source program (and prints the source listing) and converts the program into a stream of tokens called alpha

code. Each alpha code symbol is contained within a lexical class, also associated with the token. For example, for the lexical classes identifier, arithmetic or string constant, the alpha code would be the symbol table address. for the lexical classes of reserved word (get, put, do, etc.), unary operator or binary operator, The alpha code would be a unique identifier for that particular token. In all there are about a dozen lexical classes.

For each token whose alpha code is a symbol table entry, a symbol table entry is constructed if one does not already exist at block level 1. Thus variables which are used but never declared will always have valid symbol table entries. If the variable is later declared, then the created symbol table entry at block level 1 will never be used.

Because of the feature in PL/1 of allowing declare statements to appear after the use of the variables that they declare, it is not possible for the lexical scanner to return a symbol table address for identifiers. instead, the scanner returns the pointer to the name block in the symbol table. Pass 2 will later go over the program and convert all name pointers to valid symbol table addresses. If a declare statement exists for a variable, then a symbol table entry was constructed; if no declare statement exists, then a symbol table entry was constructed at block level 1 by default.

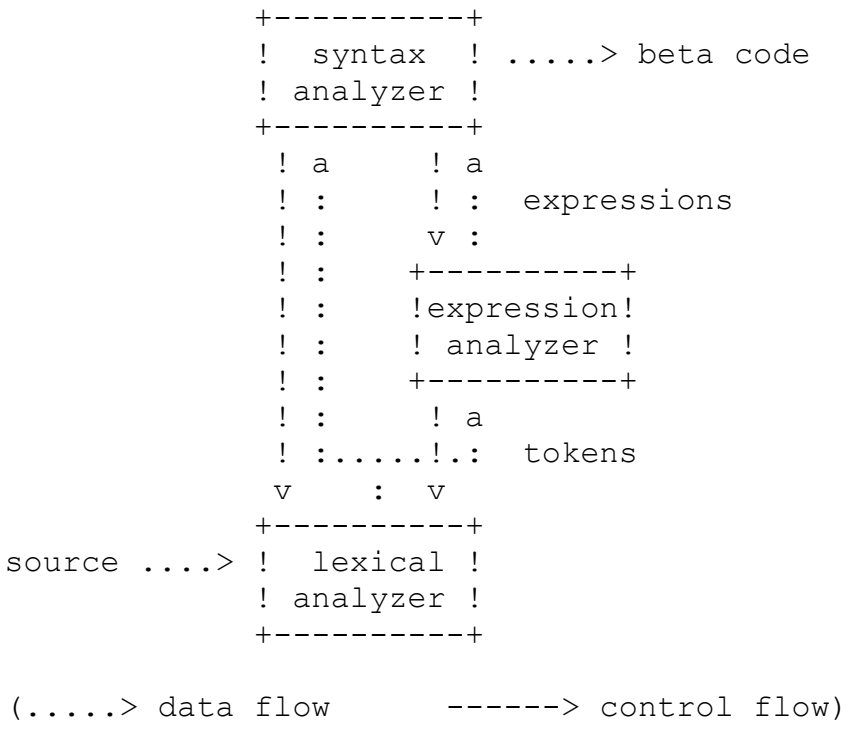
The lexical scanner is essentially a subroutine. Other sections of pass 1 call the scanner whenever a new token is needed. The parsing algorithm uses a 1 token lookahead where the parser is parsing token (n) while the scanner has already found token (n+1). this one token lookahead helps resolve some ambiguous decisions for the parser.

Syntax analysis is performed mostly by the routine called syntax. Each statement type contains a driver that sets up the appropriate tree. Since the leaves of the tree are usually expressions, an expression analyzer is called to parse expressions.

The expression analyzer is passed a code giving the context of the expression (file, arithmetic scalar, etc.). The entire expression (including the type of expression identification) is converted to beta code and attached to the tree. The expression analyzer then returns to the syntax analyzer for processing of the next phrase.

The expression analyzer is essentially a finite state automaton using a parenthesis stack. the states of the automaton are operand last (thus operator or left parenthesis is usually expected next), operator last, expression last, unary operator last and start. The transitions for the automaton are the lexical classes of the tokens. If legal, then the token is placed into the beta code, if illegal (for example two operators in a row), then an error message is generated and a valid token is inserted. When the parenthesis count goes to zero or a reserved word is encountered, the expression analyzer will terminate the expression and return to the syntax analyzer.

Briefly then, the structure of syntax analysis can be summarized by the following diagram:



Semantic Analysis

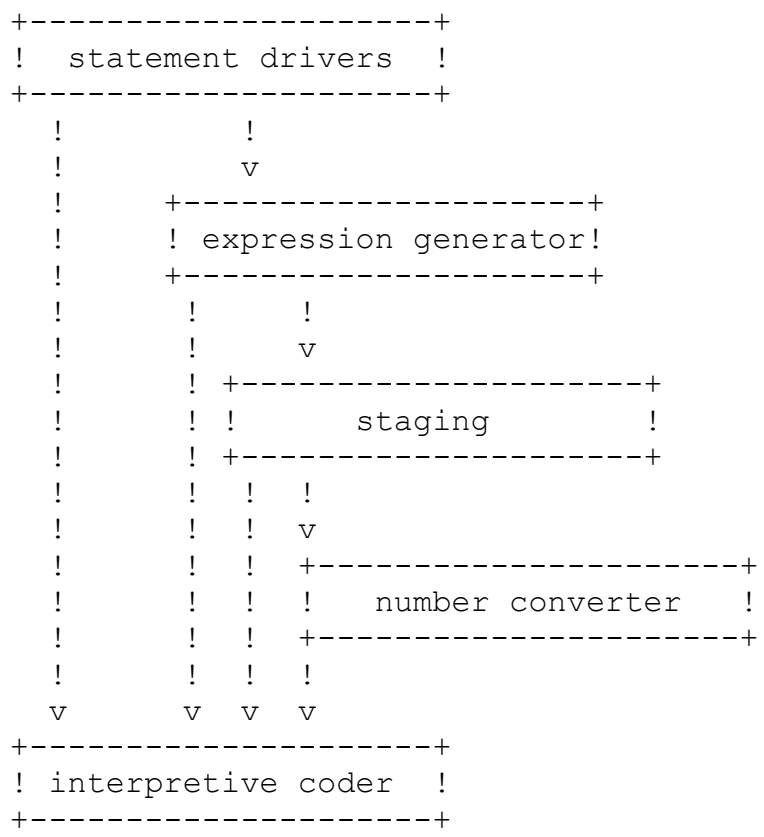
Semantic analysis resolves name pointers in the beta code into symbol table addresses. Semantic analysis looks for start expression indicators in the beta code. when an expression is scanned, semantic analysis knows which blocks are currently active, and can therefore resolve name pointers to the appropriate symbol table entry according to the proper scope rules of PL/1.

Semantic analysis also checks expressions for validity. Associated with each expression is the expression type giving the context in which it is used. Using this token, the context of the expression is checked, and if invalid, a default expression is used. The validity checking also includes checking arguments to functions with the defined attributes of the parameters of the function.

Finally, since the code generator is stack oriented, semantic analysis will convert the expression to polish postfix, for ease in generating code.

Code Generation

The code generator has the following general structure:



The code generation techniques used in PLUM are similar to tthe techniques used in the PL/C compiler. See the technical report by Wilcox [1971] for further information.

Statement Driver

Similar to pass 1, the code generator has different drivers for each statement type. The token at the start of each statement is used to determine which driver to call.

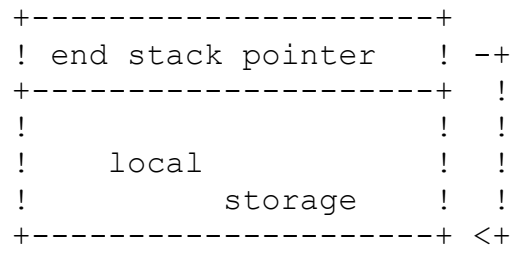
Code generation is stack oriented. whenever an expression is to be evaluated as the leaf for some phrase in the statement tree, the statement driver will call the expression generator to generate code to evaluate the expression. Upon return the top of the code generator stack will point to the location of the value of the expression during execution time.

Expression Generator

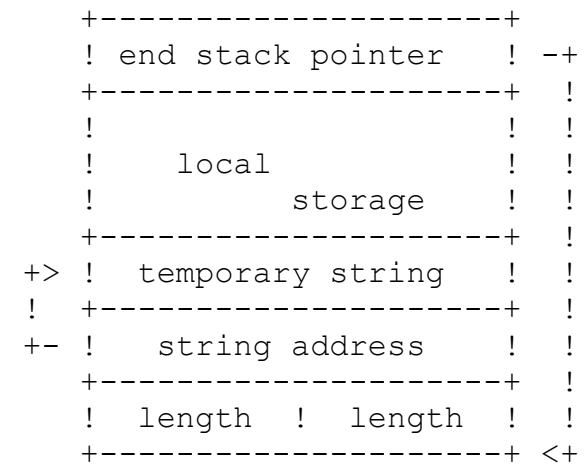
The expression generator is essentially a polish string evaluator. Whenever an operand is detected, a

stack element is placed on the top of the stack giving the data attributes of the operand and giving information about its location during program execution (an address word , usually stating that a variable is n words within a certain activation record pointed to by register y). Whenever an operator is detected, the appropriate number of operands are used to construct the result. The top of stack will reflect the location of the resulting operation (usually an address word defining one of the a registers (accumulators) in the 1108). For function calls, the arguments are stacked and then the function name is interpreted as an n-ary operator.

Arithmetic temporaries are allocated in the activation record. Once a procedure or begin statement is scanned, all storage for variables in the block are allocated. thus the compiler knows where to allocate arithmetic temporaries on demand. String temporaries are allocated at the top of the runtime stack, since the length of the string is not known during program compilation. Each activation record has a pointer to the end of itself. this pointer is used to create and access the temporary as in the following:



Which results in the following temporary structure



It is up to the string functions to recognize a string temporary and pop it off the runtime stack after being used.

Staging

Whenever an operator is found by the expression generator, staging is called. Staging contains 3 parts: targeting, staging and code generation.

Targeting determines the resulting data type of the operation. Many different operators have the same resulting data type, and hence the same targeting routines.

Staging converts the arguments to the resulting data type. The actual convert is done by the number converter.

Each operator has a set of operations to be generated. The interpretive coder is called to generate the code.

Number Converter

In order to generate code for an operator, the expression generator will call the number converter to convert arithmetic arguments from one data type to another. The number converter is passed two stack elements, and generates code to convert the datum represented by one of the stack elements into the data type represented by the other stack element.

For example, in order to generate code for the '+' operator, both arguments must be converted to the same data type. a dummy stack element is constructed which is the 'union' of the two arguments (according to PL/1's conversion rules) by the targeting routines. Converter is called twice. The first time it is passed the dummy argument and operand 1 of the '+' operator, and the second time it is passed operand 2 and the dummy argument. The number converter passes back a code giving the resulting data type of the converted arguments (fixed or floating data, real or complex). Using this code, staging can generate the appropriate sequence of 1108 instructions to perform the '+' operation.

Interpretive Coder

Code is generated using an interpretive code. a call is made to the interpretive coder, and the following words of memory are interpretive executed. these interpretive executed commands direct that specific instructions be generated.

For example, the interpretive command

```
GINST opcode,register,stackelement
```

Will generate the 1108 instruction

```
opcode    register,address-of-stackelement
```

If a temporary register had to be loaded first, then the interpretive coder would perform all of the necessary operations. Arguments for the interpretive coder commands are generally address words. see section 3.5 for an explanation of code generation data structures.

The interpretive coder commands provide a powerful command language for generating tailored

instruction sequences for a variety of conditions. Some commands that are implemented allow for setting and testing flags, jumping to other locations if the specified stack element defines a register, allocating a register, allocating temporary space in the runtime stack and generating all forms of 1108 instructions.

For example, the code for abs(x) (absolute of x) could be coded as follows:

```
        GENM                    call interpretive coder
        GIFR    x,stack1        goto x if stack1 is in register
        GGETGR  temp            allocate a register and place
                                its name in temp
        GINST   lma,temp,stack1 generate load magnitude
        GSET    stack1,temp     set top of stack to be temp
                                register
        GGOTO   y               branch to location y to finish
x       GLBL
        GINST   lma,stack1,stack1 use same register
y       GLBL
        GFIN                    terminate interpretive coder
```

Thus the sequence will generate

```
        LMA   register,x
```

or

```
        LMA   register,register
```

Depending upon where the operand is located, and the top of the stack will now reflect the fact that abs(x) is now in a register. If it was desired to store the result into a temporary, then the following commands could have been added at the end:

```
        GGETT temp,1             get 1 word temp in activation
                                 record
        GINST sa,stack1,temp     store abs(x) into temp
        GSET  stack1,temp        set top of stack to be temp
```

which would have added the instruction

```
        SA    register,temporary
```

to the generated code.

Execution

PLUM generates reentrant code with the data stored in a run time stack called the activation record.

subroutine calls, goto statements, and data directed I/O are executed interpretively by calling a run time routine, passing it the symbol table address of the variable. In this way the diagnostic capabilities of the executing PL/1 program can be enhanced.

Data

PL/1 arithmetic data can be broken down into 3 different classes

1. Float
2. Fixed binary
3. Fixed decimal

Floating point numbers are stored as double precision floating point numbers in the 1108. The precision is effectively ignored except during I/O where it defines the number of significant digits to print.

Fixed binary numbers are stored as 1108 integers in the form n*(2**q) where q is the number of fractional bits. For example the number 1.101 (1 5/8) is stored as 1101 with the compiler remembering that q=3.

Similarly to fixed binary, fixed decimal numbers are stored as integers, only in double precision floating point format. Thus 12.345 is stored as 12345 with the compiler remembering that q=3 (10**3=1000).

Fixed decimal operations require that the number be scaled by the proper power of 10. For example, 12.34 is stored as 1234 and 12.345 is stored as 12345. The code to evaluate 12.34+12.345 generates

```
1234*(10**(3-2))+12345= 1234*10+12345
```

Roundoff of fixed decimal numbers does not present a problem. Fixed decimal numbers are truncated by adding an unnormalized zero to it. the unnormalized zero has a zero mantissa, but an exponent that assumes the binary point to be at the extreme right of the number, rather than at the left. by adding this to a floating point number the number is shifted right, truncating any fractional digits and then renormalized without the fractional digits.

PL/1 string data is represented by dope vectors which are two words long. The first word is a pointer to the string while the second word is the length. For varying strings, there are two length fields- the current length and the maximum length. Bit strings are simply characters strings restricted to the elements 0 and 1.

I/O

PL/1 I/O (get and put statements) is stream oriented. this means that the input and output I/O files

consist of a continuous stream of characters where the line boundaries are somewhat arbitrary. On input PLUM will read characters from one record and then read the next record when out of characters on output, the buffer will be printed when filled, and the next buffer started.

Besides file I/O there is also string I/O. On input it is possible to consider a string expression to be a record and to read from it (get string). Similarly on output it is possible to write into a string variable instead of an output buffer (put string).

In order to implement these operations, I/O is handled via the following three step process:

1. Initialize I/O operation

2. Fill (or empty) buffer of data

3. Terminate I/O operation

The initialize buffer space subroutine (getopn or putopn) sets up a three word workspace that controls the I/O operation. For file I/O this three word area is part of an 8 word file control block; for string I/O it is a three word area in the activation record. After setting up this workspace, all subsequent I/O operations are the same for both file and string I/O.

Seven different I/O routines exist for actually transferring the data (2 for get and put list, 2 for get and put edit, 1 for get data and 2 for put data). For each item in the data list, list directed I/O passes the address of the datum and a descriptor word (giving attributes and precision) to 1 of two routines (getlst or putlst). For edit I/O the list directed information also with the format information is passed to a routine (getedt or putedt). Get data (getdta) is passed a list of sd pointers and put data (putdat, putdar) is passed either a list of sd's or the address of an array element with its sd. This special call is necessary since the subscripts have to be printed along with the value and name.

The get data routines also have a version of the lexical
scanner used in pass 1 except that

1. The number of special symbols is less than at syntax time (e.g. * and / are illegal).

2. No new symbol table entries are created. an identifier which is not already in the symbol table is an illegal identifier.

Edit I/O is handled by a generated co-routine. the next datum in the data list is obtained and control passes to the format list. The next format item is obtained and the runtime edit routines is called. for data format items (e, f, c, etc.) control returns to the data list for the next item. for control items (skip, page, etc.) control passes back to the format list for the next item, looking for a data format item.

Finally the terminate I/O operation is usually a null operation. For put data, a ';' is printed. for get string, any string temporaries are popped off the activation record.

Interactive Debugging

The interactive debugger is called whenever an error message is printed during execution of a demand program. It prints a message and reads and acts on the response. The first letter of the response determines the action to be performed.

For the most part, the interactive debugger is a set of independent subroutines; however, it does interact with the rest of the runtime support package of PLUM. For example, the get data lexical analyzer is used to read parameters from the teletype and to convert them to symbol table addresses.

The alter command is simulated as a get string(x) data statement while a display command simulates a put data(x) statement. The goto command executes a PLUM goto label statement while a call command executes a call x statement.

The breakpoint command changes the `lr`,u r11,statement-number into an illegal operation code. the operation code contingency has been modified to detect this special form of illegal operation, and to respond accordingly.

Mathematical Subroutines

The mathematical subroutines are mostly adapted from the fortran v library. The entry sequence was altered so that the function name is saved in a common location. this information is used by the error routine.

An error routine was written to trap all errors in the mathematical function routines. This was done by defining the labels nerra$, nerrb$ and nerrc$ as external labels. all mathematical functions trap to one of these locations on an error condition.

Diagnostic Features

Illegal Branches

Two forms of illegal goto statements can be executed. These are jumps into inactive blocks and jumps into inactive do loops. These are detected as follows:

Jumps Into Inactive Blocks

Associated with each block is an invocation count. This count is incremented by one each time a

new activation record is created, and is stored in the activation record. Whenever a label variable is assigned a value, the invocation count, along with the code address, is saved. When executing a goto statement, all activation records are scanned looking for the proper activation record. thus if a label variable is assigned a location and the block becomes inactive, then active again, the goto routine will catch this error.

Jumps Into Inactive Do Loops

Associated with each do loop is a location in the activation record containing the return jump at the bottom of the loop. This location as well as the return address is associated with all labels defined within the loop. Whenever a jump is executed to any label, the contents of this location is compared with the return address at the top of the loop. If the same, then this do loop is currently active, and thus the jump is valid. if this location is 0 (no do loop is active) or contains some other value, then the do loop is not active, and the jump is invalid.

Subscripts

Normally all array references are handled interpretively by passing a subscript list and array dope vector to a run time routine. This option can be disabled by the user to get more efficient code.

Trace and Histograms

The statement drivers during code generation generates code to load r11 with the current statement number. This helps the user when error messages are generated. At this point in the compiler, code was inserted to produce a statement trace (if requested by the user) and to produce an execution profile (also if requested by the user).

Use of Uninitialized Variables

For all string variables, a bit is set in the first word of the dope vector when the dope vector is created. upon assigning any value to the string, this bit is turned off. Thus if a string is accessed without first assigning a value to it, an appropriate error message can be generated.

At the present time there is no such test for arithmetic variables; however, the following algorithm is under consideration for eventual inclusion in the compiler. Each arithmetic variable will actually have 1 additional word associated with it. When storage is assigned, the address of the actual data will be stored in this word, and all references to the data will be indirect. Initially the address will be illegal so any use of the variable will cause a guard mode contingency. The contingency routine will be modified so that a guard mode caused by storing into this location will simply change the illegal address to point to the data (the following word of memory), while a load from that location means that an uninitialized variable was referenced. due to the 50% increase is data storage, and the fact that every variable will

cause a contingency the first time it is referenced, this algorithm has not been installed yet; but it may be installed in the near future.

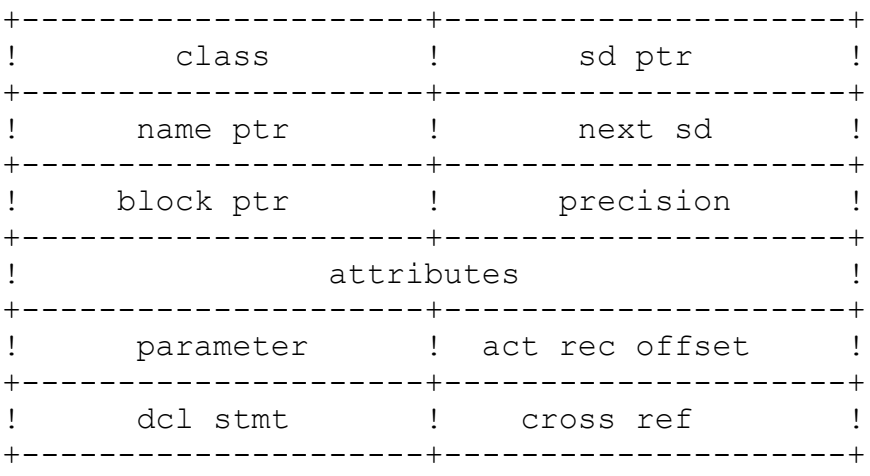
Data Structures

Symbol Table

There are two forms of symbol table entries. sd entries describe data structures in the program and bcd entries describe the name of user variables.

Symbol Descriptor (sd) Entries

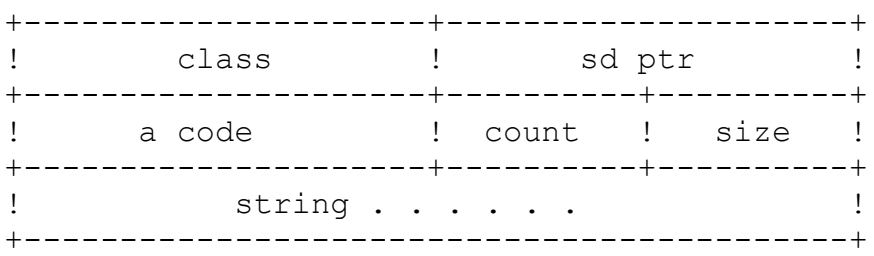
sd entries are usually 13 words long. while the format is variable, depending upon type of sd entry, the first six words of any type generally has the following format:



'Class' describes the type of sd. 'sd ptr' is a pointer to the next sd with the same name. 'Name ptr' is a pointer to the bcd block which describes the name of the sd. ' Next sd' is a pointer to the next sd in the same block as this sd. 'Block ptr' is a pointer to the block sd containing this sd. 'Precision' is the precision of arithmetic variables. 'Attributes' is the attributes of the variable. 'Parameter' is the location in the run time stack of the parameter pointer (if this is a parameter), and 'act rec offset' is the offset into the runtime stack of the variable. 'Dcl stmt' is the statement number where the variable was declared and 'cross ref' is used by the attribute listing routine.

Name Descriptor (bcd) Entries

Name entries have the following format

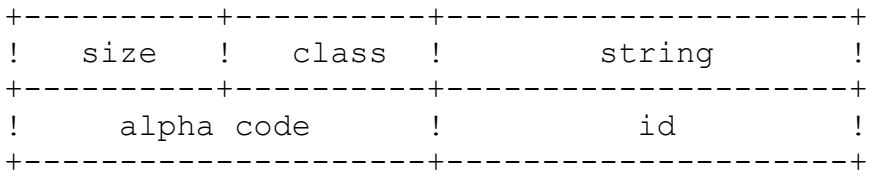


'Class' is the lexical class of the name (identifier, reserved word, decimal number, character string, etc.). 'sd ptr' is a pointer to the first sd with that name. 'a code' is a unique 18 bit token if this is the string for a PL/1 keyword. 'count' is the number of times that the variable is referenced and 'size' is the number of characters in the name. 'string' is the actual fieldata string.

Tokens (a code)

Tokens are two word entries generated by the routine lexi. Syntax analysis (syna) calls lexi for the next token. syna uses dlxac (current token) while lexi fills in dlxacn (next token). Thus syna has a one token look a head.

Each token has the following format:



'Size' is the number of source characters in the token.

'class' is a lexical class for this token. some of the classes are identifier, number, reserved word, keyword, etc. The complete list is in the procedure plac in the element dsects.

'string' is a pointer to the actual string for an identifier (or keyword).

'alpha code' is an 18 bit code designating the type of token represented. each keyword or reserved word has a unique alpha code associated with it. All alpha codes have values between 0200000 and 0377777. plac in the element dsects has the complete list of alpha codes.

'id' is a pointer to a bcd block within the symbol table for the identifier. All reserved words, keywords, numbers and identifiers have symbol table entries created for them by lexi. (reserved words

and keywords are predefined into the symbol table). pass 2 (sema) will convert bcd pointers into symbol table variable entries (sd's).

Internal Form (b code)

Types of Tokens

The internal form (b code) is produced by syna, and it is the table from which code is generated. the b code is a stream of halfword tokens which represent the program in a standard form. There are three forms of b code tokens.

1. 1. Operators. these are the 18 bit alpha codes generated by lexi. they are in the range 0200000 to 0377777.

2. 2. Operands. these are 18 bit pointers to symbol table entries. they are in the range 0 to 0177777; so they can be distinguished from alpha code operators.
   Note. PLUM will run in at most 65k core. 16 bit addressing is assumed for all symbol table references.

3. Pointers. pointers are 18 bit operands that point to other b code addresses. These pointers are distinguished from other b code operands by context.

A pointer contains two fields: 17 bits and 1 bit. the 17 bit field is the address and the 1 bit field signifies the h1 part of the word (0) or the h2 part (1). The routine sagbc converts the current location into a pointer, and the routine sagsbc converts a pointer to the actual location.

Statement Forms

Each statement is a stream of tokens which will be denoted by the following bnf-like notation. the following additional conventions are included:

[name] means that name is optional. (x ! y) means that x or y must appear.

*name means that the token name begins on a word boundary. If a token space has to be filled, acnil is inserted.

name[1] refers to a 9 bit token, the first 9 bits of name.

Expressions

```
expr = acstex[1] d exp acenex
     ( d refers to a 9 bit flag that signifies
     the expression type. it consists of 3
     arguments. argument 1 is type (scalar or
     structure), argument 2 is context (lhs
     is left hand side (assigned to), ntr
     is entry) and argument 3 is data type
     (string, arithmetic, label, anything))
     sema only looks for expressions and the d describes
     the context of the expression.
exprio = acstxi[1] d 0 exp acenxi
pexpr = aclpar expr acrpar
sexpr = expr ! sexpr accoma  expr
lexpr = aclpar sexpr acrpar
     (note. similar rules hold for lexprio, sexprio and
     pexprio, except that for I/O exprs, the following
     is also allowed:
exp = acldop sexpr acdo dospec acrpar)
exp = var ! exp biop var
biop = acbpls ! acbmin ! ...
var = aclpar exp acrpar ! uop var ! elem
uop = acupls ! acumin ! acnot
elem = name ! elem.name ! elem aclbrk slist acrbrk
slist = exp ! slist accoma exp
```

Statement Formats

```
statement = header prefix stmt

header = *acstmt[1] flags[1] stmt-number

prefix = [labels] [cond]
labels = labels label ! label
label = aclabc sd accoln ! aclabs sd expr d=(scl,lhs,lab)

cond = aclcpr cnd acrcpr
cnd = acchk lexpr ! acnchk lexpr d=(scl,any,any)

stmt = *asg-stmt ! *begin-stmt ! *call-stmt ! ...

then-stmt = flag set in acstmt flags

else-stmt = flag set in acstmt flags
```

```
deleted-stmt = acdlt   (because of error.)

end-of-b-code-stmt = aceac


asg-stmt-stmt = acasn[1] flags[1] exprio d=(any,als,any)

begin-stmt = acbgin[1] flags[1] sd

call-stmt = accall expr d=(any,ntr,any)

close-stmt = acclos[1] count close-list
close-list = close-file ! close-list accoma close-file
close-file = acfleb pexpr d=(scl,any,fle)

declare-stmt = token by token translation of source
     identifiers resolve to sd's not bcd's

do-stmt = simple-do-stmt ! do-while-stmt !
     do-spec-stmt ! do-lim-spec-stmt
simple-do-stmt = acdo
do-while-stmt = acdow acwhile pexpr d=(scl,any,sng)
do-spec-stmt = acdos do-ptr1 do-ptr2 expr d=(scl,lhs,ari)
     aceq dospecs
do-ptr1 = ptr to acdspc of first spec
do-ptr2 = ptr to right paren in get/put
do-lim-spec-stmt = (no 'to' or 'by') acdols expr
     d=(scl,lhs,any)
dospecs = dospec ! dospecs dospec
dospec = acdspc dosp1 dosp2 dosp3 dosp4 expr
     d=(scl,mto,any) acto expr d=(scl,any,ari)
     acby expr d=(scl,any,ari) acwhil expr
     d=(scl,any,sng) (any of the three phrases can
     be omitted, or can appear in any order.)
dosp1 = ptr to next do spec ! 0
dosp2 = ptr to acto ! 0
dosp3 = ptr to acby ! 0
dosp4 = ptr to acwhil ! 0

end-stmt = acend ! acend sd (for block or proc)

entry-stmt = acentr [parm-list]
     (label prefix must be present)
parm-list = aclpar id-list acrpar
id-list = name ! id-list accoma name

flow-stmt = acflow
```

```
format-stmt = acfmat edit-list
     (a label prefix must appear on this stmt)

get-stmt = acget I/O-spec
I/O-spec = get-1 get-2 get-3 I/O-spec-2
I/O-spec-2 = [file-spec] [data-spec] [control-spec]
     (all are optional except 1 of last 2)
file-spec = acfile pexpr
control-spec = acpage ! acskip [pexpr] ! acline [pexpr]
     d = (scl,any,ari)
data-spec = list-spec ! data-dir-spec ! edit-spec
list-spec = aclist lexprio
     d=(any,any,pdt) for put list
     d=(any,lhs,pdt) for get list
data-dir-spec = acdata ! acdata pnamelst
pnamelst = aclpar name-list acrpar
name-list = var ! var accoma name-list
edit-spec = acedit edit-pairs
edit-pairs = one-pair ! edit-pairs one-pair
one-pair = lexprio edit-list
edit-list = aclpar format-list acrpar
format-list = format-item ! format-item accoma format-list
format-item = ([pexpr] ! [name]) format-spec
format-spec = format-name [lexpr]
format-name = acfta ! acftb ! acftc ! acfte ! acftf!
     acskip ! acline ! acpage ! accol ! acftr ! acftx
get-1 = ptr to file ! 0
get-2 = ptr to data-dir-spec ! 0
get-3 = ptr to control-spec ! 0

goto-stmt = acgoto expr d=(scl,any,lab)

if-stmt = acif ifptr1 ifptr2 expr d=(scl,any,sng)
ifptr1 = acstmt of else ! 0
ifptr2 = acstmt of continue stmt

noflow-stmt = acnflw

open-stmt = acopen acnil open-spec
open-spec = open-p1 open-p2 open-p3 open-p4 open-p5
     [aclsiz pexpr]  d=(scl,any,ari)
     [acpsiz pexpr]  d=(scl,any,ari)
     [actitl pexpr]  d=(scl,any,sng)
     [acfile pexpr]  d=(scl,any,fle)
     (these can appear in any order)
open-p1 = ptr to next open-spec
```

```
open-p2 = ptr to linesize ! 0
open-p3 = ptr to pagesize ! 0
open-p4 = ptr to title ! 0
open-p5 = ptr to file

proc-stmt = acproc block-sd [parm-list]
     [acrcur] [acrtns aclpar attr-list acrpar]
     [acoptn aclpar acmain acrpar]
attr-list = attribute ! attr-list accoma attribute
attribute = acbin ! acdec ! acchar pexpr ! ....

put-stmt = acput I/O-spec

return-stmt = acrtrn ! acrtrn pexpr d=(scl,any,any)

signal-stmt = acsign acerrr

stop-stmt = acstop

null-stmt = acnull (;)
```

Internal Code Generation Form (g code)

G code is beta code with all expressions converted to polish postfix. In addition, all bcd entries are converted to actual variable sd entries corresponding to the scope rules of PL/1.

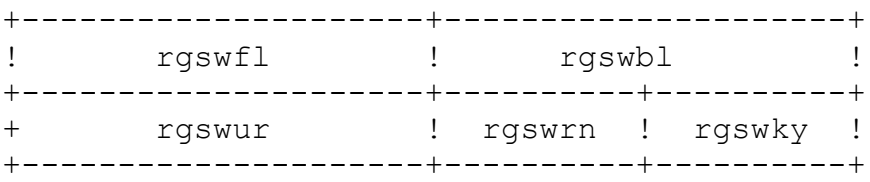
Changes in the gamma code from the beta code:

exp = name ! exp uop ! exp exp biop

Code Generation Structures

Register Status Words

A register status word describes a register in the 1108 which will be used in some generated instruction. Each register in the machine has a unique register status word describing it. Each register status word has the following format:



rgswfl and rgswbl are pointers to other register status words. All registers which can be allocated are maintained on one of two lists, a free list and a in use list. rgswfl points to the next register status word in the list, and rgswbl points to the previous register status word in the list. Registers are allocated from the head of the free list, and added to the back of the in use list. If the free list is empty, then a register at the head of the in use list will be allocated (by storing the contents of that register into a temporary).

Two separate sets of lists are maintained. One list is for the a registers (a7, a9, a11, a13, and a15) and one set is for the x registers (x4 and x6). All other registers which are used have special functions, thus rgswfl and rgswbl are set to -1 for these.

rgswur is a pointer to an address word (see next section) that points to this register.

rgswrn is the actual register number.
space
rgswky is a set of flags describing the use of that register.

flag 0 - register is in use (on in use list).

flag 1 - register contains one word operand (fixed point).

flag 2 - register contains 2 word operand (floating point).

flag 3 - x register contains an address pointer.

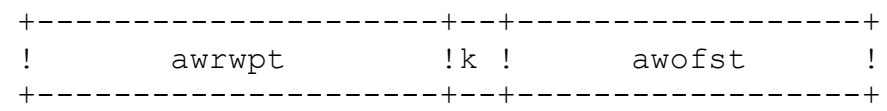
Address Words

Every variable that is acted upon by the code generator has an address word associated with it. These address words describe the location of a variable during execution, and is thus the main structure which is used to generate code. Address words come in two forms:

a) Register address words (raw) signify that the variable (or expression) described by the address word is currently in a register.

b) Core address words (caw) signify that the variable described by the address word is in core memory.

Address words have the following structure:



k is the keep bit. If code is generated which takes data from the location specified by this address word, then if the keep bit is on the register will not be returned to the pool of unallocated registers.

Register Address Words

awrgpt is zero.

awofst is a pointer to a register status word signifying the register containing the variable. the pointer rgswur in the register status word points to this address word.

Core Address Words

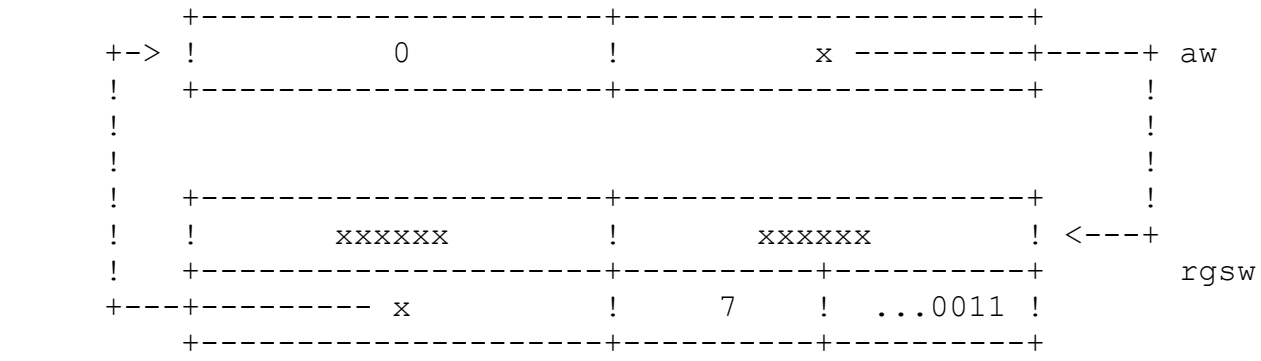
awrwpt points to the address word containing the register where the data area can be addressed.

awofst is the offset within the data area pointed to by awrwpt.

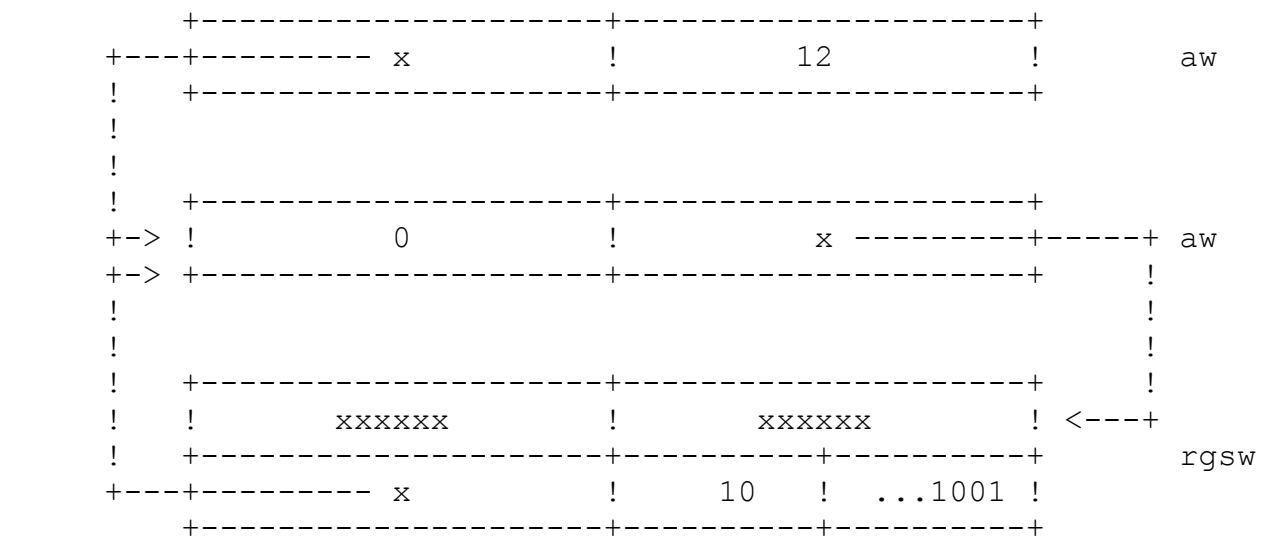
Examples of Address Words

An address word is created for each block in a program. This address word points to the location of the beginning of the run time activation record for that block. For each variable in the block an address word is created by setting awrwpt to be a pointer to this block address word and by setting awofst to be the local activation record offset for that variable. (compare examples 2 and 3.)

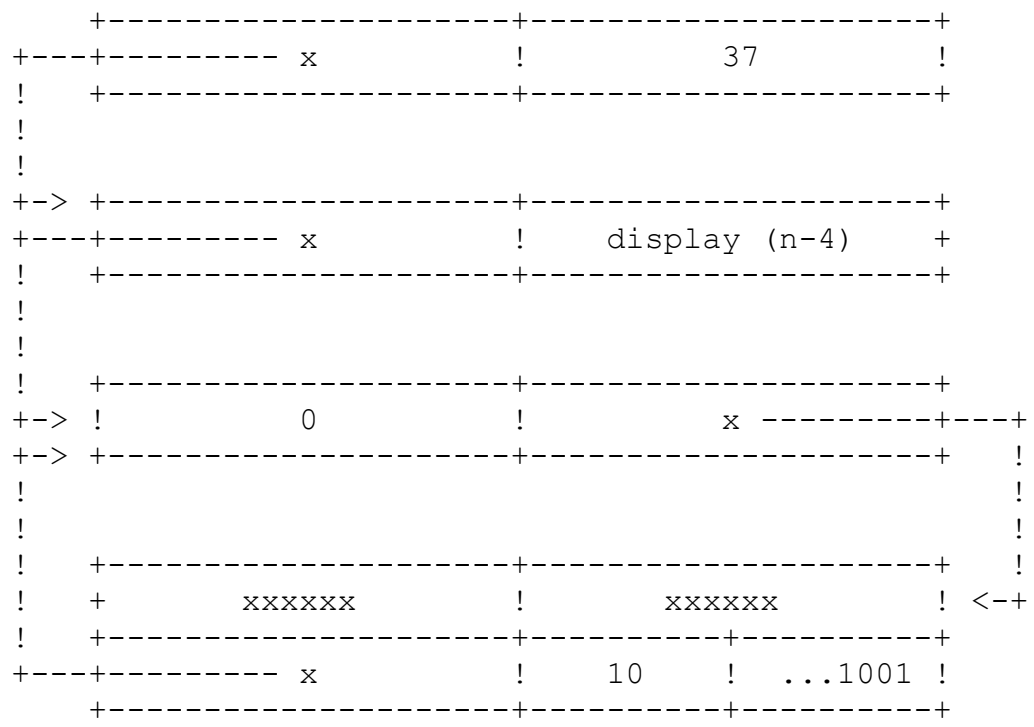
Example 1. a variable is in register a7. the address word will look as follows:



Example 2. if a variable is word 12 of the local activation record (pointed to by register x10), then the address word will be:

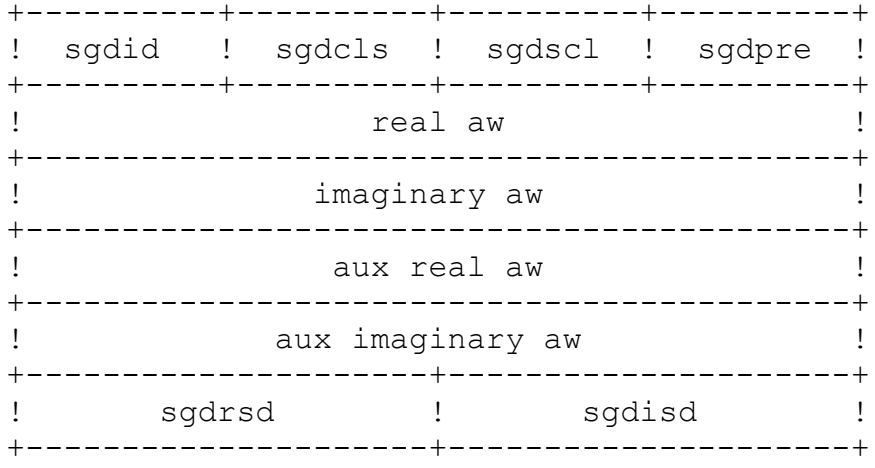


Example 3. if a variable is at word 37 of an activation record at level n-4 (where the current block is level n), then the address words created are as follows:



Code Generator Stack

The code generator uses a stack to generate code. for each variable, the following 6 word stack entry (sgdsct) is allocated.



sgdid gives the type of stack entry (expression, do block, if block, procedure-begin block).

sgdcls gives attributes for this entry.

sgdscl gives the scale for the variable.

sgdpre gives the precision for the variable.

real aw is an address word (see next two sections) describing the location of the real part of the variable.

imaginary aw is an address word describing the location of the imaginary part of the variable. (it points to the variable kkdo, which is permanently allocated in plumcm, for real variables.)

aux real aw and aux imaginary aw are auxilliary address words for structure accessing.

sgdrsd is a pointer to the symbol table entry (sd) for the real part of the variable.

sgdisd is a pointer to the sd corresponding to the imaginary part of the variable.

agrl1 is a name that refers to the real address word of the first stack entry. agrl2 is a name that refers to the real part aw of the second stack entry. Similarly agim1 points to the imaginary aw of the first stack entry and agim2 refers to the imaginary aw of the second stack entry.

Code Generation Macros

Code is generated via the interpretive coder. all code is a series of macro calls. the following notation will be used to describe these macros:

aw is an address word. raw is a register aw.

flag is a 9 bit flag.

lbl is a statement label.

j is an instruction j field (for the 1108).

reg is a register number.

Invoking the Interpretive Coder

There are 4 entries to the interpretive coder. they are invoked as follows:

```
gen                 generate the following instruction.
genm                generate the following list of
                    instructions (until gfin)
```

```
genr  lbl ! r,reg   generate instruction at lbl or at
                    address in register reg.
genmr lbl ! r,reg   generate list at lbl or reg.
```

Terminating the Interpretive Coder (from genm or genmr)

```
gfin  ['pop']       terminate genm or genmr. ('pop'
                    says to pop stack entry on returning.)
```

Instructions

```
gload aw [,2]       change aw to 1 [2] word raw
                    (generate load instruction)
ginst opcode,raw,aw generate opcode raw,aw
gdble opcode,raw,aw generate double inst (071)
gfloat opcode,raw,aw generate float inst
gshin opcode,raw,aw generate shift inst
op codes are written opxx where xx is the 1108 mneumonic.
```

Flag instructions

```
gni   flag          flag and dgflgz -> dgflgz
goi   flag          flag or dgflgz -> dgflgz
gmvi  flag          flag -> dgflgz
gxi   flag          flag xor dgflgz -> dgflgz
gbon  loc,flag      goto loc if flag is on in dgflgz
gboff loc,flag      goto loc if flag is off in dgflgz
gbr   loc,flag,[on ! off],aw,j if flag is (on ! off)
                    in aw,j then go to loc
```

Gen Control Macros

```
ggoto loc           transfer control to loc
gbal  loc           internal subroutine call to loc
                    (one level only)
grtrn               return from internal subroutine
gappn loc           go to loc. return via ljmp. this
                    is an escape from the gen macros.
                    use sparingly. also registers must
                    not be altered while in appendage.
```

```
gifr  loc,aw        go to loc if aw is really raw
gife  loc,aw        goto loc if aw is even register
gifo  loc,aw        goto loc if aw is odd register
```

Jump Instructions

```
glabel aw           define location (of generated code)
                    in aw. used in jump instructions
gjump aw            generate jump to location in aw
gcjmp opcode,raw,aw generate test-opcode,u raw,0
                               j     aw
gsub  name          generate lmj x11,name
glmj  raw           generate lmj raw,$+1
gjgd  raw,aw        generate jgd instruction
```

Other Gen Macros

```
glbl                define label
glit  n             insert the n following words
                    of literal data
glitrg reg          reg has next instruction.
gmove n,m,s         stack(m) -> stack(n) (s words)
gfree aw            free temp or register
gkeep aw            do not free aw
gshift aw           change register in odd-even pair
                    to point to other register of pair
ggetgr aw ['x']     allocate an 'a' ['x'] register in aw
gfsr  raw           free scratch register (x5 or x11 only)
gset  aw1,aw2       aw2 -> aw1
ggett aw,n          allocate n word temp in aw
gmark aw            save current generated code location
guse  aw            set next location to be generated
```

Example of Generation Macros

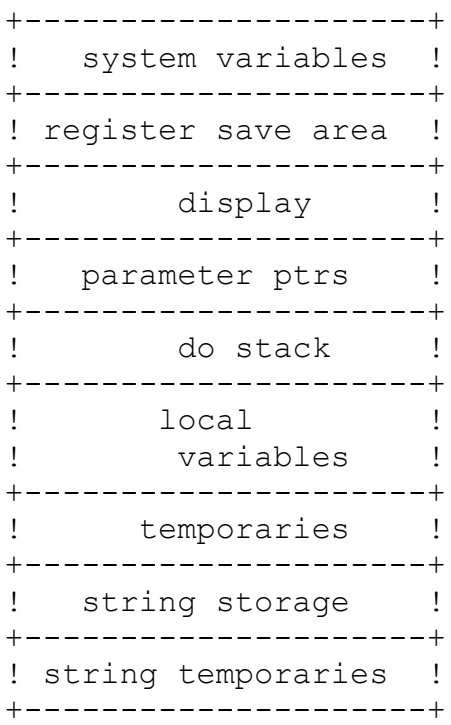
If the top two stack entries contain fixed binary numbers, then the following sequence of instructions will add the two numbers and store the result into a temporary. the top stack element will be deleted and the new top of stack entry will contain the address of the temporary.

```
        genm                        start generation
        gifr  inreg,agrl2           goto inreg if agrl2 is in reg.
        gload agrl2,1               put 2nd argument in reg.
inreg   glbl
        ginst opaa,agrl2,agrl1      add top of stack to 2nd
        ggett agrl1,1               allocate 1 word temp.
        ginst opsa,agrl2,agrl1      store sum into temp
       gset  agrl2,agrl1           set agrl2 to be temp
        gfin  'pop'                 terminate code and pop
                                    stack
```

Execution Structures

Activation Record Format

Data storage is managed by a runtime stack. Each element in the stack is called an activation record, and contains the data area for a procedure or a begin block. The format of the activation record is as follows:



'system variables' contain information about the activation record. this information includes sd of procedure, invocation count of activation record, activation record of calling procedure and type of call

(as a function or as a subroutine).

'register save area' is the save area for the registers from the calling procedure.

'disply' contains the activation record points for the static nesting of the current procedure. Normally 4 levels of activation records are kept in index registers x7 to x10 however, if a variable from a more deeply nested procedure is used, then an index register (x5 or x11) will first be loaded with the appropriate pointer from the display variables.

'parameter ptrs' contain pointers to the call by reference parameters.

'do stack' contains a stack of words used to contain the return jumps in do loops. The nesting of a do loop is calculated, and the appropriate element in the stack is used. This do stack is used in trapping on illegal goto statements to labels in an active do loop.

'local variables' are the declared variables of the procedure. For string variables only dope vectors are allocated.

'temporaries' are allocated temporaries (arithmetic) allocated during code generation time. 'string temporaries' are allocated during execution.

Data Formats

Fixed Binary

Fixed binary data is stored as binary integers. A number n with a given (p,q) is stored as the 36 bit integer n*2**q. The position of the binary point is assumed to be q bits from the left (low order bits) end of the number. This q is known at compile time, and the appropriate scaling is done (shifting left or right) whenever a calculation calls for it.

Float Binary

A float binary number is stored as a double precision floating point number .

Float Decimal

A float decimal number is stored as a double precision floating point number.

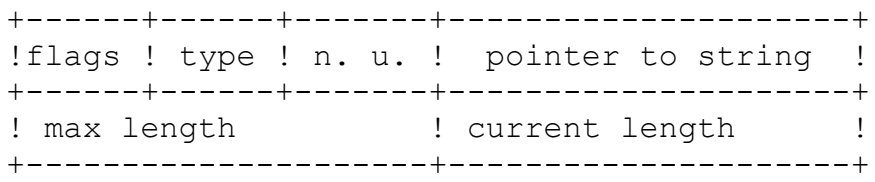
Fixed Decimal

A fixed decimal number of precision (p,q) is stored as a double precision floating point number in

the form n*10**q. Just as in fixed binary, the number is represented as an integer, with the appropriate scaling done by the compiler by generating code to multiply or divide by the proper power of 10.

Character Strings

Each character string is described by a dope vector as follows:



The flags which are defined are:

1 - string is uninitialized. data must be stored into string before it may be referenced.

Type flags are the following:

1 - dope vector is for a bit string
2 - string has varying attribute

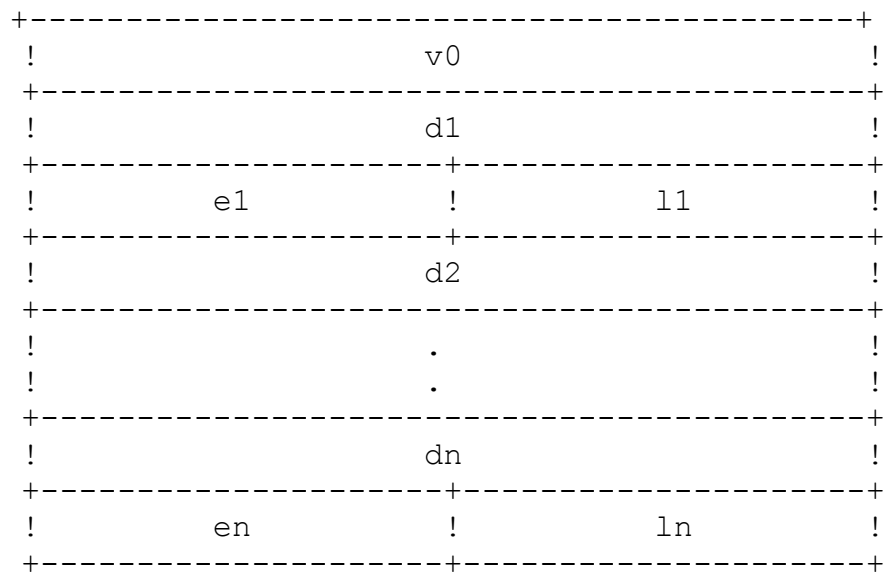
The field n. u. is currently not used.

Bit Strings

Bit strings have the same format as character strings. Bits are represented as octal 60 ('0') and octal 61 ('1'). note that each bit of a bit string takes 6 bits (1 character) in the 1108.

Arrays

Arrays are described by the following dope vector



v0 = virtual origin (element a(0,0,0,...,0))

di is the multiplier for the ith subscript

li is the ith lower bound

ei is the extent-1 of the ith subscript.

The ith upper bound is li+ei.

v0 is the virtual origin of the array. vo is the address of the zeroth element (if it existed).
v0 = actual starting address of array -sigma(di*li) .

Address of a(s1,s2,...sn) = v0+sigma(si*di).

For example, if the declaration were

```
declare a (1:10, 1:20);
```

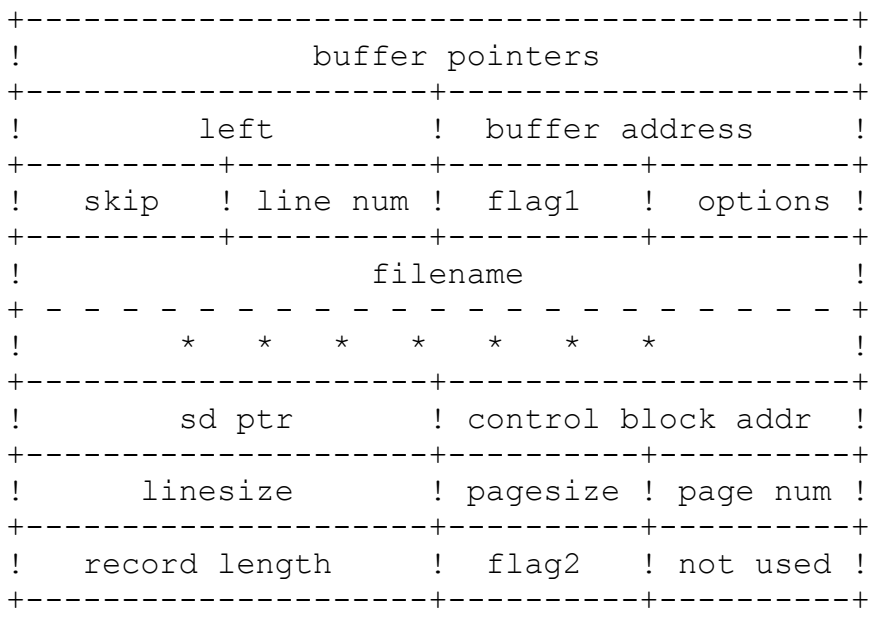
Then l2=2, e2=20 and d2=1 and l1=1, e1=10 and d1=20.

Structures

The dope vector of a structure is the sum of the dope vectors for each of its constituent leaf elements. Arithmetic variables have a one word dope vector pointing to its location in memory.

File I/O

I/O is managed during execution by associating an 8 word I/O control block (iocb) with each file. the format of the control block is as follows.



I/O in PLUM is stream oriented. When a buffer fills then it is printed and the next buffer is started. ' buffer pointers' contain the 2 register contents necessary to store the next character in the buffer. ' left' contains the number of characters left in the buffer and 'buffer address' is the start of the buffer. 'skip' contains the number of lines to skip before this buffer is printed and 'line num' is the current line number of this image on the printed page. 'flag1' contains flags used by the runtime I/O routines and 'options' are flags determining the type of I/O statement currently being executed using that iocb.

These first three words are also called the string control block.

'filename' is the EXEC8 filename for the file, 'sd ptr' is a pointer to the file sd currently accessing the iocb. 'control block addr' points to a read$/print$ packet, or points to an fct table entry if the file is an sdf data file. 'linesize' is the current linesize and 'pagesize' is the current page size and 'pagenum' is the current page number. 'record length' is the buffer size in words (ceil(linesize/6)) and 'flag2' contains additional flags used by the I/O system.

GET (and PUT) execute as follows. for each GET (or PUT) statement that specifies a file, an iocb is obtained. For each GET (or PUT) statement that specifies a string, a string control block is allocated in

the activation record. Each GET (or PUT) list, data or edit item is a separate call to the I/O system. only the first three words of the iocb (string control block) are used in filling the buffer with the data.

Register Allocation

Registers used in PLUM fall into three classes: index registers, accumulators and special registers. Each register which is used has an associated register status word and address word defined for it during code generation.

Index Registers

Index registers are allocated for string operations to contain the address of the dope vector. the register is always obtained from the set X4 or X6.

Accumulators

Accumulators are always allocated in odd-even pairs so that double word floating point can be accommodated easily. For fixed point operations, the odd register of the pair is used. Accumulators are registers A7 to A16.

Special Purpose

All other registers have a special purpose as defined in the list below.

X1-X3 temporary registers used by I/O code and initial code

X4 allocated index register

X5 temporary index register

X6 allocated index register

X7 pointer to level n-3 activation record

X8 pointer to level n-2 activation record

X9 pointer to level n-1 activation record

X10 pointer to local (level n) activation record

X11 link register and temporary index register

A0 PIT stack register

A1 parameter passing register

A2 parameter passing register

A3-A4 work register

A5-A6 work register

A7-A8 register pair 1

A9-A10 register pair 2

A11-A12 register pair 3

A13-A14 register pair 4

A15-A16 register pair 5

R1 repeat register (hardware used)

R2-R10 temporary scratch registers used by execution subroutines (not by compiled code)

R11 statement number of currently executing statement

R12 constant 1

R13 constant 2

R14 constant 3

R15 constant 4

Detailed Structure of PLUM

General Design

PLUM is a 3 pass compiler that produces absolute machine code for the Univac 1108. It is reentrant and resides in the I bank of the 1108. The compiler consists of 7 segments.

a. A resident nucleus which controls the execution of the compiler, and provides support functions such as input, output, error message generation and er interface with the EXEC8 operating system.

b1. Syntax analysis which converts the source program into a more structured internal form called beta code (or just b code).

b2. Semantic analysis completes the generation of the b code which syntax analysis was unable to complete. this operation includes converting all expressions into reverse polish.

c. An attribute listing and cross reference table is optionally printed at the user's discretion.

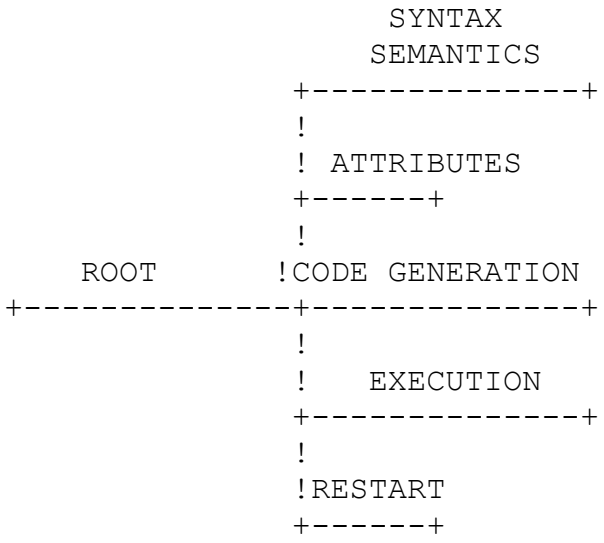
d. Code generation which converts the b code into absolute machine language.

e. Execution which monitors the executing program and provides support functions such as I/O, mathematical functions, and string operations.

f. PLUM common (called plumcm) is the compiler workspace which consists of a block of core allocated in the D bank of the 1108. Predefined variables are loaded into the D bank of the 1108 as a separate segment.

g. Restart. following execution, a user may edit the source program and recompile. Restart reinitializes the compiler without having to go through the EXEC8 command language interpreter.

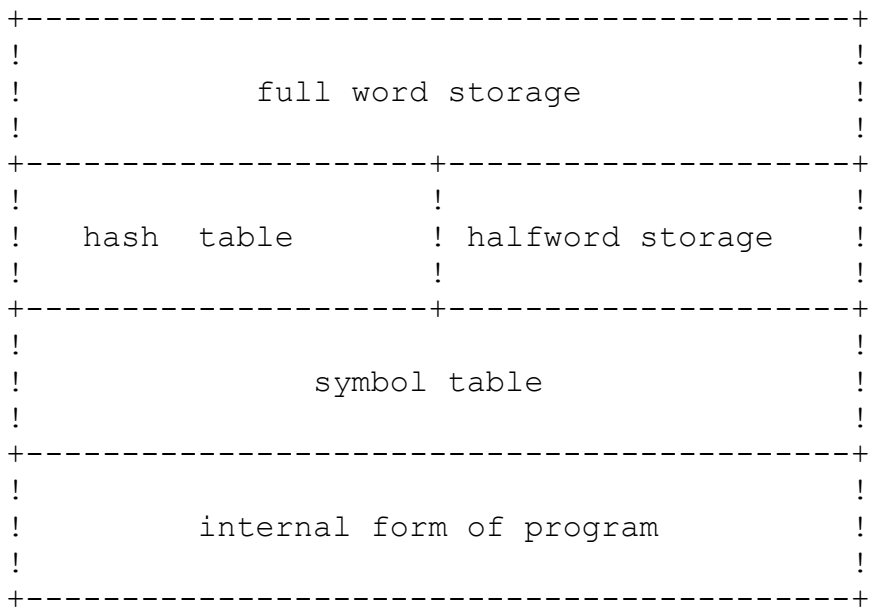
PLUM consists of an absolute element with the following tree structure.



The following sections describe this structure in greater detail. (chapter 5 contains a more detailed explanation of each of the modules in the compiler.)

PLUM Common

PLUM common is the workspace where the user program is compiled. It contains approximately 10000 words. it has the following general structure.



Full word storage is a block of 800 words of storage. This is divided into two segments. approximately the first 200 words of this block and the last 100 are used by the root segment, while the remaining 500 words are used as temporary storage by the various phases.

Hash table is a 1024 entry hash table. each entry points to an address in the symbol table.

Halfword storage is a block of 1024 halfwords. this is divided into two segments. approximately the first 100 are used by the root segment, while the remainder are used by each phase.

The symbol table (or sd table) is a table of symbol descriptors. There are two basic types of symbol table entries. sd entries describe data structures in the program (variables, labels, blocks, etc.) and bcd entries describe names used in programs. Each name will hash to a unique slot in the hash table and thus a program is limited to 1024 different names. Since all keywords are initially defined to be at specified slots in the table, the actual number of different permissible variables is on the order of 800 names.

The internal form (or bcode or beta code) is a compressed tree structured form of the source program. As code is generated during the code generation phase, b code is replaced by the object code. The remainder of the b code space is used for the runtime storage of the program.

Resident Nucleus

The resident nucleus consists of the following modules:

```
plex      EXEC 8 - PLUM interface. all er's done from
          this module.
cont      PLUM control - calls the various phases
```

```
          and has PLUM I/O functions
exer      error message generator
debg      demand debugging routine
list      disassembler (used by debg and code gen)
cvnm      number converter (fieldata to internal)
exfl      file control information (I/O)
updt      restart command initializer
plumcm    Dbank workspace
sir       EXEC 8 I/O routines
infor     ''
preprm    ''
```

Syntax Analysis

Syntax analysis consists of the following modules

```
syna      main syntax driver
decl      declare statement driver
expr      expression analyzer
lexi      lexical scanner and symbol table manager
pccd      reads options supplied on @PLUM image
main      initialized hash table with reserved words
          and key words.
bccm      bcode address computer
rvse      syntax error messages and reverse translate
ersy      pass 1 error messages
```

Semantic Analysis

Semantic analysis consists of the following modules:

```
sema      semantic analysis. resolve names and convert
          to polish
```

Attributes and Cross Reference

```
attr      cross reference and attribute listing
erat      attribute error messages
```

Code generation

Code generation consists of the following modules:

```
cgdr      main statement driver
cgas      assignment driver
cgio      I/O statement driver
cgbk      block storage allocator
cgxg      expression generator
cgoa      arithmetic operator code
```

```
cgos      string operator code
cgbf      built in function operators
cgtb      operator branch table and bcode address
          calculators
cgcv      arithmetic data conversion
cgig      instruction generator
ercg      code generation error messages
```

Execution

Execution consists of the following modules:

```
exec      main execution monitor
exst      string routines and string built in
          functions
rtdt      functions time and date
rter      mathematical function error routine
rtta      tangent routine
rtsq      square root routine
rtat      functions atan, atand
rtsc      functions sin, sind, cos, cosd
rtlg      function log
rtl1      functions log10 and log2
rtex      function e**x
rtff      function float ** float
rtfi      function float ** integer
rtii      function integer ** integer
exas      arithmetic overflow checks
excs      convert from arithmetic to string
excn      convert between data types
exop      open/close statements
exdt      get/put data routines
exgp      get / put list
exed      get / put edit
exai      remote format subroutine
exll      low level I/O routines
exlx      execution lexical scanner
exdb      execution debugger
exdi      functions dim, hbound and lbound
pmdp      post mortem dump
exht      execution profiles
exmt      static / dynamic program analysis
erex      execution error messages
```

Restart

Restart consists of the following modules:

```
rset      reset dbank to initialized state.
```

The PIT Macro Language

The Language

Notation

The following notation is used in this section of the report. [...] denotes an optional parameter, (x ! y) denotes a choice between x or y and <name> is a programmer defined label.

Basic Structure

Note. all procedures are defined in the element proclib.

The pseudo machine has three types of registers: Index, accumulators and a flag register. The index registers are labeled X1 to X10 (X11 is a link register), the flag register is F and the accumulators are A3-A4 and A6 to A15.

Registers A3, A4 and A6 are temporary. that is any PIT instruction which calls a subroutine (e. g. link, move cve etc.) will destroy them. Register A0 is the main stack and should never be referenced in a program directly. A1 and A2 are used by the getc and putc routines and should not be used. register A5 is the flag register.

All data areas (via dsects or csects) must be defined before the first executable instruction is declared.

Only the label, lloc, and local commands can have labels.

'field' will refer to one of the following:
- a storage location in a csect
- a storage location in a dsect
- a literal constant (written 'constant,i')

'*field' means indirect addressing.

Fields can be modified by index registers (as in 'field,Xi'. '*Xi' means auto incrementation of Xi.)

PIT Instructions

Register Instructions

All instructions in this class are of the form:

opcode (Xi ! Ai ! F),[*]field[(,[*]Xi ! ,i)]

```
load      xn an or f     load register
loadn     an             1108 lna
loadm     an             1108 lma
loadxi    xn             1108 lxi
loadxm    xn             1108 lxm
loada     xn an          1108 l,u (load address)
store     xn an or f     store register
storen    an             1108 sna
storem    an             1108 sma
add       an xn or f     1108 a
addm      an             1108 ama
sub       an xn or f     1108 an
subm      an             1108 anma
mpys      an             1108 msi
mpy       an             1108 mi (avoid using, use mpys)
div       an             1108 di
slb       an             1108 lssl (shift left bits)
srb       an             1108 ssl (shift right bits)
lor       an             1108 or (results in an)
land      an             1108 and (results in an)
lxor      an             1108 xor (results in an)
cve       xn an          convert to external
                         register is converted to a signed
                         fieldata string in 2 words at
                         field.
cvi       xn an          convert to internal
                         2 word field is converted to
                         interger and put in register.
cvo       xn an          convert to octal
                          low order 18 bits of register
                         converted to 6 fieldata chars
                         at field.
addd      an             1108 da
fmpyd     an             1108 dfm
fadd      an             1108 dfa
fdivd     an             1108 dfd
```

If Statement

a. arithmetic `if an,code,field then,<loc1> [else,<loc2>]`

code can be eq, ne, lt, gt, le, ge.

Field is the same as for register instructions, except index registers cannot be used. (a using

statement before the if can accomplish this in almost all cases.)

<loc1> and <loc2> can be programmer labels or the special labels jmp (return from subroutine), jmp1 (return from subroutine and skip one word), ljmp (return from local subroutine), and ljmp1 (return from local subroutine and skip one word).

The meaning of this statement is 'if an code field is true, then go to <loc1> else go to <loc2> (if any)'.

b. flag `if flag,code[,field] then <loc1> [else,<loc2>]`

Code can be on, off, same, n(ot)same.

On means some of the flags are on. off means that all of the flags are off. same tests for equality, and nsame tests for inequality.

If field is not present, then f is assumed.

c. compare `if field1,code,field2,length then,<loc1> [else,<loc2>]`

Code is eq or ne

Length is a constant (number of words) or a register containing the number of words (written R,register).

d. misc. `if field,code then,<loc1> [else,<loc2>]`

Code can be even or odd (field must be a register) or zero or n(ot)zero. (field can be a register or a core location.)

Set Statement

```
set flag,code[,field]
```

Code can be on or off.

If field is missing, then F is assumed.

This instruction sets the corresponding flags on or off in the desired field.

Move Statement

a. `move field,constant,i` - move constant to field.

b. `move field1,field2` - move field 2 to field 1.

c. `move field1,field2,length`

Length is a constant, or a register with the number of words (written R,register). If the length is less than 5, then an appropriate number of load-store (or loadd-stored) instructions are generated. If greater than 4 or R,register form is used, then a subroutine is called.

Control Statements

```
label     define statement label
link <n>  call subroutine <n>
local     entry point to a subroutine
lloc      entry point to a subroutine that doesn't
          call any others. (return address not saved).
          (including PIT subroutines e.g. move, cvi)
jmp  n    return from a local subroutine. skip n words.
ljmp n    return from a lloc subroutine. skip n words
setl an,field  set loop counter to loop 'field' times.
loop an,<loc>  loop to <loc> if not finished.
lpcnt an,field  get number of times in loop, if field is
                initial value.
goto <loc> transfer to <loc>
getlnk xi get return address into xi.
putlnk xi put return address (followed by ljmp)
```

Character Statements

```
          setgc n,field       set character reading routines (uses
                              x1 and a1 registers). n is character
                              size. (2,4 or 6 per word.)
          setpc n,field       setup put character registers (x2 and
                              a2).
          getc                read next character into a8.
                              a setgc must have been executed
                              previously to set up registers.
          putc                write next character from a8.
                              a setpc must have been executed
                              previously to set up registers.
          getgl xi            get current word getcing from in xi.
          getpl xi            get current location putcing to in xi.
```

Other Instructions

```
          push n              push stack pointer n words
          pop n               pop stack pointer n words
          stack reg           stack h2 part of xn or an register
```

```
unstk reg           unstack h2 part of reg xn or an.
strt                start program (from exec 8).
quit                return to exec 8 (er exit$).
equ                 same as assembler equ
origin <loc>        reset location counter to <loc>
                    (used in dsects to redefine same
                    storage area.)
get iocb            read line input
put iocb            write line output
iocb   type,device,data
                    input,rdr,buffer,eof-addr for input
                    out,lst,buffer,no.-of-words for ouput
snap id,lowaddr,upaddr  a snapshot dump is produced.
                    lowaddr and upaddr are the bounds.
                    id is a 3 character string identifying
                    the snapshot.
dclrg               define PIT equ's
ssws                move character and convert subroutines
```

Data Storage

All of these commands can have labels. If any variables, other than those defined by the following commands are used in an operand field of an instruction, the following must be appended to the label '+(in 0,j-field,1,0)'. this is an escape from PIT and should be used sparingly. The 'in' form is needed to set the bits used by the PIT procs to separate dsect from csect operands.

```
dsect n             template definition (n is unique <=31)
csect n             real storage definition (n is
                    location counter.)
dend                end of dsect
cend                end of csect
ss                  define sixth word constants
sq                  define quarter word constant
sh                  define half word constants
sac   x,y           define 2 halfwords (constants or
                    addresses).
sf                  define full word constants
                    ss, sq, sh or sf can have an
                    operand (initial value).
                    the initial value cannot be an
                    address.
sfw   n             reserve n fullwords.
```

Assembling Programs with PIT

All assemblies will begin as follows:

```
          axr$ .          define 1108 equu's
          dclrg .         define PIT equ's
          @add pltvds .   add common storage (PLUM common)
          @add other-workspace . add local storage for
                                                 that phase
          cend .          end PLUM common
          dsect-procs .   define dsects
    <n>   csect 3 .       use location counter 3
          program .       program to assemble
          end .
```

Acknowledgements

Partial support for the development of PLUM came from NASA grant NGL-21-002-008 to the Computer Science Center of the University of Maryland. Students who have contributed major sections to the project include john freeman, Robert Kirby, Paul McMullin and Joseph Pollizzi. Others associated with the project include Spec Bowers, Walter Christensen, Mary Lynn Fitzgerald, Kenichi Harada, J. Michael Kamrad and Richard Newton. Mr. John Menard, Director of the Computer Science Center, must also be acknowledged for the help he gave in supporting this project.

References

The following references may be of some use:

[ANSI, 1973]
ANSI - ECMA, basis/1-10, June, 1973
This is a preliminary standards document for the PL/1 Language.

[Conway, 1973]
Conway R. and Wilcox T., Design and Implementation of a Diagnostic Compiler for PL/1, Communications of the ACM 16, no. 3 (March, 1973) pp 169-179
This author spent 2 years working on the PL/C project before coming to the University of Maryland. because of that association, the structure of PLUM resembles the structure of PL/C, and this paper may shed some light on PLUM's organization.

[Wilcox, 1971]
Wilcox T, Generating Machine Code for High-Level Programming Languages, Cornell University Department of Computer Science Technical Report TR-71-103 (August, 1971)
The code generation techniques used in PLUM are similar to the techniques described by Dr. Wilcox in this report.

[Zelkowitz, 1972]
Zelkowitz M., PIT: A Macro-Implemented Implementation Language, Software Practice and Experience 2, no. 4 (October, 1972), pp. 337-346

This is a preliminary description of the PIT macro language used in the implementation of PLUM.

[Zelkowitz, 1974a]
Zelkowitz M., PLUM: A Diagnostic Compiler for PL/1, USE Technical Notes, New Orleans (March, 1974) pp. 8.27-8.38
This is a brief user's introduction to PLUM

[Zelkowitz, 1974b]
Zelkowitz M., PLUM Reference Guide, Computer Note CN-8,
Computer Science Center, University of Maryland, (July, 1974)
This is a user's guide to using the PLUM compiler.