[1] looks as follows –

```
class DFG : public remote_base {
private :
  local_streams cout;
  symbol_table_rec *ST_ptr;
  constant_table_rec *CT_ptr;
  module_table_rec *MT_ptr;
  ...

public :
  DFG();
  ~DFG();
  int add_str(symbol_table_rec);
  int add_ctr(constant_table_rec);
  int add_mtr(module_table_rec);
  ...
  ...
};
```

As is evident, the major components of the Intermediate Form are represented by pointers to lists. The methods shown above are used to build the **DFG** from the ascii file.

[1] Note : The class definition presented here is by no means complete. Additions and modifications will follow as the implementation starts

```
   struct inout *next_io_ptr;
};

struct input
{  /* Input Format : eg. (m11.i3) etc. */
   char inp_id[MAX_COMP_NAME_LEN];
               // Of the form <parent_id>.<module_id>:<VHDL_name>
               // eg. m3.i2:blah
   struct input *next_in_ptr;
};

struct op_erator
{  /* Operator Format */
   char op_id[MAX_ID_LEN];        // Operator id : Format x<num> eg. x7
   int op_code;
   struct input* input_list;      // List of inputs
   struct branch_rec *br_ptr;     // For SELECT/DIVERGE operators
   char module_id[MAX_ID_LEN];    // For ENTER/LEAVE operators
   struct inout* output_list;     // List of outputs
   struct op_erator *next_op_ptr;
};

struct module_table_rec
{  /* VIFmodule Format */
   struct symbol_table_rec* str_ptr;       // pointer to the Symbol Table Entry
   struct inout* input_list;               // List of Inputs
   struct op_erator* operator_list;        // List of Operators
   struct inout* output_list;              // List of Outputs
   struct module_table_rec *prev_mtr_ptr;
   struct module_table_rec *next_mtr_ptr;
};


struct branch_rec
{  /* used to store branch specific information */
   char branch_id[MAX_COMP_NAME_LEN];   // Format [b|d]<num>.x<num>
   char sel_val[MAX_ID_LEN];            // To be modified later
   struct op_erator *op_ptr;            // List of operators in branch
   struct branch_rec *next_br_ptr;
};
```

The **DFG** class has three components corresponding to the symbol table, constant table and the module list. These three components are implemented in terms of linklists of the structures 'symbol_table_rec', 'constant_table_rec' and 'module_table_rec'. The 'module_table_rec' in turn consists lists of structures 'op-erator', 'inout' etc. The methods of the class manipulate the lists to modify the DFG as needed. In C++, the **DFG** class

```
        unsigned int not_used      : 4;
    } flags;

    struct {
        int from;
        int to;
    } word_struct;                // word structure

    struct {
        int from;
        int to;
    } bit_struct;                 // bit structure

    char resn_fn_id[MAX_NAME_LEN]; // if it is a port or signal and has a
                                   // resolution fn
    char name[MAX_NAME_LEN];       // VHDL name if any
    int mode;                      // If it is a port
    struct {
        char alias_of[MAX_ID_LEN];
        int start_index;
    } alias_info;                  // Information about alias

    struct symbol_table_rec *next_str_ptr;
};

struct constant_table_rec
{ /* Constant Record Format */
    char constant_id[MAX_ID_LEN];   // Format c<num> eg. c10
    int value;                      // Decimal value of the constant
    int width;                      // bit_width of the constant
    struct constant_table_rec *next_ctr_ptr;
};

struct inout
{  /* InOut format */
    char inout_id[MAX_ID_LEN];
        /* format is <letter><num>
                <letter> :: 'i' # module input
                            'l' # local storage of module
                            'f' # formal parameter of module
                            'o' # module output
                            'p' # operator output
                <num> :: Sequential number
                example :: 'i1' , 'p10' etc.  */

    char destn_id[MAX_ID_LEN]; // id of the destn as stored in the symbol table
    int width;                 // Size of the carrier in context
    char name[MAX_NAME_LEN];   // name of the carrier if any
```

```
! Symbol Table Information :
! Format % Id Parent Flags Wd_struct Bit_struct Name Res_fn_id [:: Mode]
e1 # "0000000000000000" # # template # ;
a2 e1 "0000000000000000" # # template # ;
s3 a2 "0000010000000010" # <3:0> SomeSignal # ;
m4 a2 "0000000010000000" # # SomeProcedure # ;
s5 m4 "0001100000000010" # <3:0> SomeFormal # :: IN ;
m6 a2 "0000000000001000" # # P1 # ;
s7 m6 "0000100000000010" # <3:0> SomeVariable # ;

! Constant Info :
! Format % Id Value Width
c1 0 <2>;
c2 1 <4>;

! Module List
! Format % (1)Symbol Table entry (2)Input List (3)Operator List (4)Output List
a2 e1 "0000000000000000" # # template # ;
l1 s3 <4> SomeSignal ;
x1 DIVERGE
d1.x1 PARALLEL
x2 ENTER @m6
p1 s3 <4> SomeSignal ;
d1.x1 ENDPAR
x1 MERGE
-----
o1 s3 <4> SomeSignal ;

m4 a2 "0000000010000000" # # SomeProcedure # ;
f1 s5 <4> SomeFormal ;
-----
x1 RD (m4.f1) (#.c1)
p1 s3 <4> SomeSignal ;
-----
o1 s3 <4> SomeSignal ;

m6 a2 "0000000000001000" # # P1 # ;
l1 s7 <4> SomeVariable ;
x1 RD (#.c2) (#.c1)
p1 s7 <4> SomeVariable ;
x2 CALL @m4 (x1.p1)
p1 s3 <4> SomeSignal ;
-----
o1 s3 <4> SomeSignal ;
```

```
entity template is
end template;
architecture template of template is
  signal SomeSignal : Bit_Vector(3 downto 0);
  procedure SomeProcedure(SomeFormal : Bit_Vector(3 downto 0)) is
  begin
    SomeSignal <= SomeFormal;
  end SomeProcedure;
begin
  P1 : process
         variable SomeVariable : Bit_Vector(3 downto 0);
       begin
 SomeVariable := 1;
 SomeProcedure(SomeVariable);
      end process;
end template;
```

Figure 1: A VHDL Example

```
struct symbol_table_rec
{  /* Symbol Table Record Format  */
   char symbol_id[MAX_ID_LEN];
        /* format is <letter><num>
                <letter> :: 'e' # symbol is an entity
                            'a' # symbol is an architecture
                            's' # symbol is a storage
                            'm' # symbol is a VIFmodule
                <num> :: Sequential number
                example :: 'e1' , 's10' etc.  */

   char parent_id[MAX_ID_LEN];
        /* Same format as above - identifies the parent */

   struct {  // a collection of 16 bits that identifies various attributes
        unsigned int is_bit_vector : 1;
        unsigned int is_array      : 1;
        unsigned int is_alias      : 1;
        unsigned int is_process    : 1;
        unsigned int is_block      : 1;
        unsigned int is_conc_sa    : 1;
        unsigned int is_subprogram : 1;
        unsigned int is_loop       : 1;
        unsigned int is_port       : 1;
        unsigned int is_signal     : 1;
        unsigned int is_variable   : 1;
        unsigned int is_formal     : 1;
```

- DIVERGE/MERGE : This pair of op-codes are used to list all parallel branches. The organization of this composite operator is similar to that of the SELECT/ENDSEL pair. It has no inputs or outputs. The begining of each parallel branch is marked by a line having "dj.xm" as it's Id that represents the j'th branch of operator 'xm' (a DIVERGE operator), followed by the keyword PARALLEL. The operators of the parallel branch follow this line and the branch is terminated by a line having "dj.xm" as the first field and the keyword ENDPAR as the next field. After all the branches have been listed, the DIVERGE operator terminates with a line that has the operator Id as the first field followed by the keyword MERGE.
- CALL/ENTER : These two operators have all the sub-components of an normal operator (viz. inputs and outputs). In addition, it has the optional field that identify the **vif** module referenced. This field appears after the Op-code.
- LEAVE : This operator does not have any inputs or outputs, But it has the optional field that identify the **vif** module referenced.
- **stable/quiet :** These control operations are VHDL specific. These operations are used for evaluation of the two predefined attributes **STABLE** and **QUIET** of VHDL signals. These nodes have single input and output edges. The input edge identifies the parent operation while the output edge carries the attribute evaluation result.

Module Output List : The last sub-component of the **vif** modules is a list of outputs. These identify the storages that are modified inside the module body. The outputs are represented in the **InOut** format.

Figure (**??**) is an illustrative VHDL example. The equivalent intermediate is shown in Figure (**??**).

2 The VSS Intermediate Format Data Structures

The first phase of VSS constitutes reading the intermediate form ascii file and develop data structures that will be operated on by the subsequent phases of VSS. A C++ implementation of VSS on the ES-Kit has affectuated the organization of the Intermediate Form in terms of a C++ class description. The C++ class defined to represent the Intermediate format is called *DFG* (an acronym for Data Flow Graph). The private part of the class consists of link lists of structures that represents the Intermediate Format and the class has methods that update these data structures.

The private part of the class **DFG** is organized in terms of actual C++ structures that closely reflects the ascii counterpart of the Intermediate Format. Let's look at these structures first, the similarities with their ascii counterparts should be very evident –

that correspond to constants, are represented similarly, except that before the '.' there is a '#' eg. "(#.c4)".

5. **Operator Output List** : A list of all outputs of the operators, one per line, in the **InOut** format.

The Op-code decides the interpretation of the input list and rest of the operator fields. The Logical, Arithmetic and Relational operators (except for NOT) are binary, and have two inputs and one output. The interpretation for other operators are as follows –

- RD : It has two inputs and one output. The first input identifies the carrier to be read, and the second one is a constant that identifies the bit offset where the read starts from. The carrier can be a storage defined in the input VHDL or one generated in the **vif**. The number of bits to be read is specified in the outputs section.
- WR : It has two inputs and one output. The first one identifies the source. The second input is a constant that identifies the offset where the writing should start. The destination storage and the number of bits that are updated is specified in the outputs section.
- ARRRD : It has two inputs and one output. The first input identifies the array itself, and the second one identifies the index of the array that is to be read. The number of bits to be read is specified in the outputs section.
- ARRWR : It has three inputs and one output. The first one identifies the array itself, the second one identifies the index of the array and the third one identifies the source carrier. The number of bits to be modified (written into) is specified in the outputs section.
- SELECT/ENDSEL : It has one input and multiple outputs. The input identifies the carrier on which the branching is decided. SELECT is a composite operator, which lists the branches, the conditions under which the branch is taken and a list of operators in the branch.

  Each branch of the SELECT begins with a line having "bi.xn" in the Id field. This represents the i'th branch of operator 'xn' (a SELECT operator). This is followed by the keyword BRANCH. The final field in the branch line shows the selector value for the branch ([TRUE] or [FALSE] for now). All operators of the branch follow this line. The branch is terminated by a line that has "bi.xn" as the first field and the keyword ENDBR as the next field.

  After all branches have been listed, the SELECT operator terminates with a line that has the operator Id as the first field followed by the keyword ENDSEL. This line is followed by a list of all outputs of the SELECT operator in the **InOut** format.

Operator List : This constitutes the third subcomponent of the module. It is a list of operators that implement the behavior of the input VHDL description. The operators have the following fields –

1. **Operator Id** : An 'x' followed by a number.
2. **Op-code** : This identifies the type of the operator. Following are the possible op-codes –

Op-code	Meaning
AND	Logical AND operator
NAND	Logical NAND operator
OR	Logical OR operator
NOR	Logical NOR operator
XOR	Logical XOR operator
XNOR	Logical XNOR operator
NOT	Logical NOT operator
PLUS	Arithmetic PLUS operator
MINUS	Arithmetic MINUS operator
MULT	Arithmetic MULTIPLY operator
DIV	Arithmetic DIVIDE operator
EQ	Relational EQUAL operator
GT	Relational GREATER THAN operator
LT	Relational LESS THAN operator
CONCAT	VHDL CONCAT operator
WAIT	VHDL WAIT operator
RD	Bit-Vector READ
WR	Bit-Vector WRITE
ARRRD	Array READ
ARRWR	Array WRITE
SELECT	Begining of a IF/CASE
ENDSEL	End of IF/CASE
DIVERGE	Begining of a PARALLEL BRANCH
MERGE	End of PARALLEL BRANCH
ENTER	Call a single entry vif module
LEAVE	Leave a vif module
CALL	Call a subprogram
PAD	PAD left with 0's

3. **Optional Module Id** : For operators that refer another **vif** module (eg. ENTER, CALL or LEAVE).
4. **Operator Input List** : Each input to the operator is enclosed between '(' and ')'. The inputs reference values somewhere in the **vif**. As such, the operator input format is a '.' seperated composite of two fields. The first field identifies the parent (operator, **vif** module or constant). The second field identifies a particular child of the first field. Hence, "(x9.p2)" identifies the second output of operator 'x9', "(m3.l4)" identifies the fourth input of the module 'm3' (which happens to be *local* to the module). Inputs

1. **vif Id** : It is a constant ID (number prefaced by 'c').
2. **value** : The decimal value of the constant.
3. **width** : The width of the constant.

Note : Constants with the same value but different sizes are considered to be distinct.

1.4 VIF Modules

This constitues the third and the last section of the **vif**. This section is a list of **vif modules** seperated by a dashed line. The modules are the prime objects that encapsulate the algorithm specified by the input VHDL description. But before we go into the modules, let us specify a generic format that is used extensively in the modules.

InOut Format

Each value produced or used anywhere in the **vif** is represented in the **InOut** format. It has four fields –

1. **Id** : It has a letter prefix, followed by a number. While the prefix identifies the object type, the number represents the count in the context used. Following are the prefixes and their meanings –

   f : for a *formal* parameter of a **vif** module.

   i : for a **vif** module input.

   l : for a *local* declaration in a **vif** module.

   o : for a **vif** module output.

   p : for an operator output.

2. **vif Id** : Id of the storage where the value is stored/referenced.
3. **Width** : The bit-width of the storage.
4. **Name** : VHDL name of the storage (if any).

Each module consists of the following components –

Symbol Table Entry : This is the first line of the module and is a repetation of the information stored in the symbol table.

Module Input List : This list identifies all the storages (one per line) that the module operates on, in the **InOut** format. The entries in this list can be of three types –

1. A *local* storage (eg. a process variable)
2. A *formal* storage (eg. a subprogram parameter)
3. A storage imported from a parent module (eg. a signal in a process statement)

1.2 The Symbol Table Section

The Symbol Table is the first section of the **vif**. It is organized one entry per line, terminated by a ';'. The first three lines of this section are comments that identify the section to be a symbol table and specify the format of the entries. Following are the fields of each symbol table entry –

1. **vif Id** : The Id letter identifies the entry type.

   e identifies an *entity*.

   s identifies a *storage*.

   a identifies an *architecture*.

   m identifies a **vif** *module*.

2. **parent Id**

3. **flags** : A collection of 16 flags, numbered 0 through 15 with 0 being the rightmost flag. These flags represent the following –

```
Position        = 1, if
--------        -------
 0-2 for internal use
  3 is a formal
  4 is a variable
  5 is a signal
  6 is a port
  7 is a loop
  8 is a subprogram
  9 is a wait
  10 is a conc_sa
  11 is a block
  12 is a process
  13 is a array
  14 is a bit_vector
  15 is a temp
```

4. **word structure**

5. **bit structure**

6. **VHDL name** (if any)

7. **resolution fn Id** (if any)

8. **Additional Information** : [port] : the mode (IN, OUT, INOUT)

1.3 The Constant Table

The constant table is the second section of the **vif**. It describes the constants used in the **vif** arranged one per line, each terminated by a ';'. The first two lines are comment lines identifying the section to be a constant table and describing the format of each entry. Each entry has three fields –

The VSS Intermediate Format

Jayanta Roy

December 19, 1991

This document describes the VSS Intermediate Format (**vif**) – it's structure and representation. The VHDL-to-Intermediate-Format translator (**v2if**) converts the input VHDL behavioral description and stores it in an ascii file. This ascii representation is then read in and operated on by VSS. Section **??** describes the ascii representation of the intermediate format, and Section **??** describes the way it is stored, after being read in by VSS – in terms of data structures.

1 The ascii representation

The **vif** (and it's ascii representation) is organized in three sections. The first section is like a symbol table, with an entry for each object in the intermediate format. The second section is a list of all constants that are used. The third section is a list of modules that encapsulates the input behavioral description.

1.1 Observed conventions

Let's get introduced to the conventions that we follow in the **vif** represention.

- A '!' denotes a comment. Anything following a '!' till the end of line is treated as a comment.
- A '#' denotes an empty field.
- A number contained in '<' and '>' indicates bit-width.
- A number contained in '[' and ']' indicates array-width.
- A '%' indicates "contained in" or "declared in", depending on the context.
- Inputs to operators are enclosed in '(' and ')'.
- A '::' indicates the begining of optional information.
- All objects of the intermediate format have an identifier associated with them. The identifier is a number, prefaced by an alphabet that indicates the type of that object.