Circuit Compilers Don't Have To Be Slow

William C. Diss

Mentor Graphics Corporation*

Abstract – This paper describes a method for efficient compilation of circuits which is used in a commercially available fault grader. Data structures and algorithms are presented which can be used in processing circuits in a textural or schematic format. Performance is documented showing the results of various circuits.

1. Introduction

Much work has been done over the years to improve the performance of various design phases such as simulation, layout, fault grading, etc. However, very little work has been done on improving the performance of the compilation phase. As a result of this, many design systems have circuit compilers which are painfully slow. Since this phase is necessary for all analysis areas of circuit design and because of its repetitive nature, it is critical that this step be as efficient as possible.

The techniques presented here are used in Caedent Corporation's circuit compiler, (CDLCON), which is used to process Caedent's hierarchical design language, (CDL). CDL is the entry point for circuit information into Caedent's fault grading system [2].

2. Definitions

The following definitions are presented to show the terminology used in this paper:

- **Module:** A hierarchical body of circuit information. It contains instances of other modules which are connected by nets. This body is usually a functional block designated by the circuit or library designer. However in a schematic capture system it can be a portion of a functional block, which is commonly known as a sheet.
- **Primitive**: A module with no internal connections. It is the lowest level module in the hierarchy.
- **Instance:** A unique occurrence of a module.
- **Net:** A connection between instances.
- **Pin**: The point of contact between a net and an instance.

3. Compilation Phases

The compilation process can be viewed as three separate phases. The first phase compiles the circuit by checking for logic and syntax errors, and stores each module in a library with the original hierarchy. The second phase loads in all of the necessary modules referenced in the design. The third phase processes the hierarchical modules by flattening them to the primitive level. These phases are known as the *compiler*, *loader*, and *flattener*.

The data from a schematic can be stored directly into a library without a separate compile step. The compile phase is assumed to be an integral part of the schematic editor. The end result of the schematic entry session is the storing of each sheet in the circuit library.

It should be noted that the flattened data does not have to be stored in a data file. The disk access time for reading flattened data is greater than the time to read hierarchical data because the flattened data file occupies more space. The additional disk access time to load flattened data is greater than the additional CPU time involved in flattening the hierarchical data. However, a flattened file is warranted if the platform is limited in memory or for accelerators.

*Note: This paper describes work performed while the author was with Caedent Corporation. Caedent's technology was acquired by Mentor Graphics on July 31, 1987.

CH2540-3/88/0000/0622$01.00 © 1988 IEEE

4. Data Structures for Circuit Objects

The data structures for the hierarchy involve three interconnected structures: modules, instances and nets. The data on these structures is used to produce the flattened instances and nets. The data is organized so that the structures containing the most information are the least used entities in the circuit, and the entities that occur most frequently contain small amounts of information. This technique allows modules to be stored efficiently in memory so that paging is minimized.

4.1 Hierarchical Module Details

A module object (see Figure 1) is created for each unique module used in the design. Each module object contains:

- **Module Name:** The object containing the module name is kept separate from the contents. This separation allows modules with various names to access the same body of information.

- **Module Statistics:** These statistics contain the number of nets and instances used in the module. The number of dependents refers to the number of unique modules called by this module. The number of primary inputs, outputs and bidirects refers to the boundary nets of the module.

- **Flags**: The primitive flag designates a module that is at the primitive level. The loaded flag is used in the loader to indicate that a particular module has been loaded. The in-use flag is used in the loader to check that no module is recursively calling itself in the same hierarchical branch.

- **Instances:** Reference to list of instances used in module (see Figure 2).

- **Nets:** Reference to list of nets used in module (see Figure 3).

- **Boundary Pins:** The external contacts of the module. Each pin contains a reference to the net connected to that pin (see Figure 3).

- **Dependent Modules:** This list contains a reference to each of the unique modules used by this module.

4.2 Hierarchical Instance Details

An instance object (see Figure 2) is created for each instance used in the module. Each instance object contains:

- **Number of Inputs, Outputs, Bidirects:** The number of respective nets which are connected to the input, output, and bidirect pins of the instance. These numbers can differ from the numbers on the module being called due to variable width pins.

- **Index in Module Dependent List:** The index refers to a position in the list of dependent modules (see Figure 1). The reference at the position refers to the module called by this instance

- **Pins:** The contacts to this instance. Each pin contains a reference to the net connected to it (see Figure 3).

4.3 Hierarchical Net Details

A net object (see Figure 3) is created for each net in the module. Each net object contains:

- **Index into Module Boundary Pin List:** The index refers to a position in the module boundary pin list (see Figure 1). This index is used in flattening for passing the net references for net connections between modules.

- **Flattened Net Reference:** The reference points to the flattened net created in flattening process (see Section 4.5). This reference is used in flattening so that all connections within a particular module can access its unique flattened net.

- **Flags:** Flags are set if the net is a primary input, output, or bidirect of the module. Flags are also set if the net has a load and a source. The reserved flag is set if the net is connected to power or ground. The wired output flag is set if multiple instances drive the net. The variable flag is set if the net can vary in width depending on how the module is called.

4.4 Flattened Instance Details

A flattened instance is created each time a hierarchical path is traversed down to a primitive. The flattened instance is similar to the hierarchical instance (see Figure 2) with the number of inputs, outputs, and bidirects removed. During the flattening process the variable width pins are calculated so that there is a match between the number of instance pins and the number of pins on the primitive.

The other major difference with the flattened instance is that the pin list contains references to flattened net objects and not hierarchical net objects.

4.5 Flattened Net Details

A flattened net object is created for the boundary level nets of the top module, and for all internal nets of all referenced modules whenever a module is called.

The flattened net object contains a reference to the level alignment structure used when this particular flattened net is created (see Section 8). The flattened net object also contains fan-in and fan-out information. This information contains references to all instances and pins on those instances which are connected to each of the flattened nets.

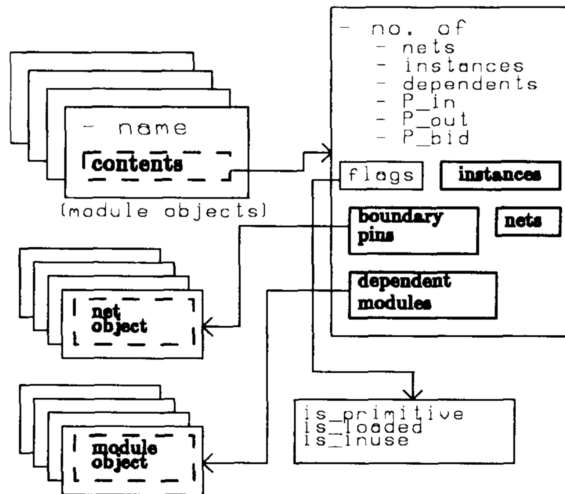

Figure 1 Hierarchical Module Structure

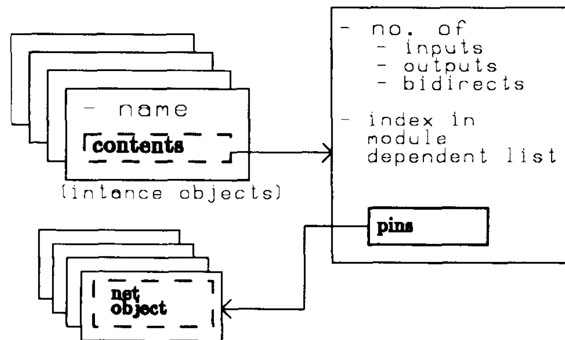

Figure 2 Hierarchical Instance Structure

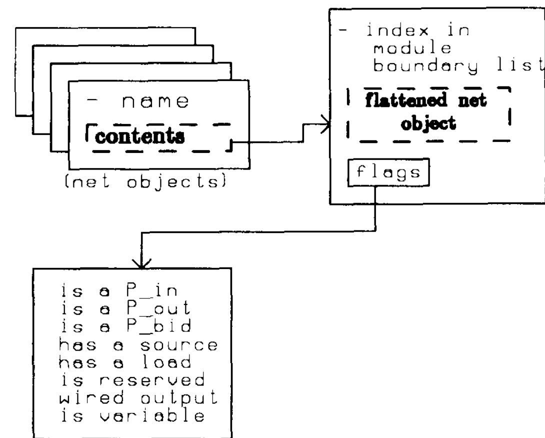

Figure 3 Hierarchical Net Structure

5. Language and Compiler Details

The CDL language is a keyword-driven language and has features to support various design languages. The key features are the following:

- Modules can be organized independently of their hierarchical order.
- Connections between modules can be designated using two different methods:
  - A pin-oriented list which contains all nets connected to the instance, or,
  - A net-oriented list which contains connections to the net by instance name and pin name.
- Multiple names (aliases) can be used to refer to one module, and to one net.
- Connections to a module can vary in width. Module connections are expanded to the appropriate number of pins based on how many connections are used by the calling instance.

For each keyword extracted from the file, the compiler looks up the appropriate processing function. The data following the keyword is passed to the function until a terminator is seen. All processing is done in one pass.

The compiler processes each module of the circuit and then adds or replaces it in the circuit library. Compression techniques are used to store the contents

of each module.

6. Library Organization

The circuit library contains three sections, a header section, a module section, and a index section (see Figure 4). Each section contains the following:

- **Header Section:** This section contains the file revision, number of modules, and the file offset into the index section.
- **Module Section:** This section contains all modules in the circuit or in a modeling library. Each module is stored with the names of the dependent modules, net objects, and instance objects. The dependent names are stored first so that they can be immediately accessed without reading any of the contents of the module.
- **Index Section** This section contains the name of each module in the file and its offset, installation date and other flags. The index section is stored at the end of the file so that new modules can be appended with minimal file access. The module offset permits direct access to each module in the file.

A module index structure is created for each module by using the module index section. These structures are used for all library operations, which include the addition of modules, replacement of modules, listing of modules, and loading of modules for flattening the circuit.

7. Loading the Circuit

The circuit is loaded as follows:

- **Load Index Sections** The index section of the circuit file and any libraries are loaded into memory. Checks are done for multiply-defined modules.
- **Input Name of Module for Analysis** The user inputs the name of any module in the circuit for analysis. This module is known as the "top" module.
- **Check for Existence of Necessary Modules** Starting with the top module, the dependent names are loaded, from the library file, and these names are checked on the module index structure. These dependent names have their dependent's names loaded and checked. This process continues until the only dependents of modules are primitives. By checking for all the necessary dependents before loading any of the contents, the user is immediately notified of any unresolved module entries.
- **Load the Contents of Each Module** The modules are loaded starting with the lowest level module. This "bottom up" loading is used so that modules that have connections by pin names can be in memory when they are called by a higher level module. All the contents of each module are stored on the appropriate hierarchical structures (see Section 4).

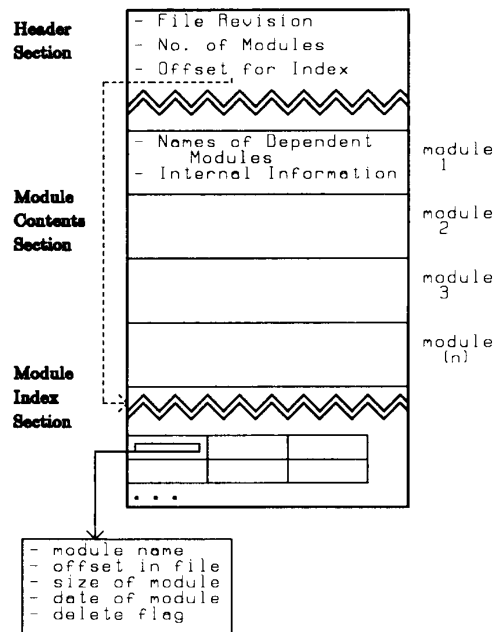

Figure 4 Library Organization

8. Flattening the Circuit

The flattener uses the level alignment structure (see Figure 5) to produce flattened nets and instances. The flattener uses a "depth-first search" algorithm to traverse through the entire hierarchical tree.

The following steps show the recursive algorithm for flattening:

- **BEGIN (Process Current Module)**
  - **FOR (All Boundary Pins)**
    - Align boundary pins of module to pins of level object.
    - Get the flattened net references from pin list of level object and store them on boundary level nets.
    - Resolve unconnected pins on the module.
  - **NEXT (Boundary Pin)**
  - **FOR (All Internal Nets in Module)**
    - Create a flattened net object and store reference on hierarchical net object.
  - **NEXT (Internal Net)**
  - **FOR (All Instances in Module)**
    - Get a level object.
    - Update hierarchical name by appending instance name.
    - Store module reference of instance on the level object.
    - Expand instance pins to correct width.
    - Get the flattened net references from the nets of the instance and store them on the pin list of the level object.
    - If module instance is not a primitive call **Process Current Module**.
    - If module instance is a primitive, do primitive checks.
  - **NEXT (Instance)**
- **END (Process Current Module)**

The flattening algorithm is not that intensive because of the organization of the data structures. The loader creates all the necessary hierarchical structures while the contents for each module are read from the disk.

For the top module, a flattened net object is created for all nets. The flattened net references for the boundary pins are stored on the pin list of the top level object.

A level object is only created for each hierarchical level. Level objects are re-used as the flattener moves back up through the hierarchy.

The alignment step looks at the boundary pins of the current module and expands them to the necessary number by looking at the level object. If there are more module pins than pins in the level object, the output nets of the module are left as unconnected and the input nets of the modules are connected to default power or ground nets.

Primitives are checked for the proper number of pins. Since the hierarchical structure allows variable pin widths on modules, an incorrect usage of a primitive will result in the wrong number of pins.

During the flattening process, unique hierarchical names are not created and stored. One string contains a dynamic hierarchical name (see Figure 5). Whenever a new flattened net is created, a reference to the current level object is stored on the flattened net object (see Section 4.5). The hierarchical net name is extracted by referencing the hierarchical name index from the level object which is referenced on each flattened net. A unique hierarchical name is then made by using the characters from the beginning of the dynamic hierarchical name to the index.

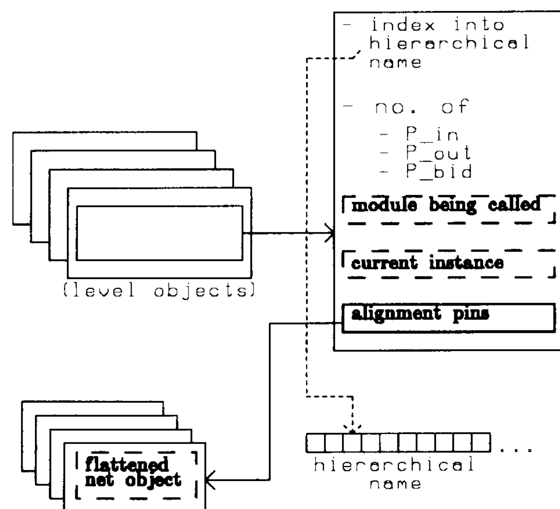

Figure 5 Level Alignment Structure

9. Performance Data

Table 1 shows the processing times and various statistics for different circuits and libraries. All circuits were actual designs used by Caedent's customers. The times, in CPU seconds, were computed on an Apollo Series 3000™ [1]. The flattening times do include the generation of a flattened file. Circuit B uses Library 1, and Circuit C uses Library 2. Circuit A has all of the necessary modules in the circuit file.

Times are given for each phase because they point out how much time can be saved in incremental changes. Incremental changes involve the compilation and replacement of certain modules. For example if a module of 200 lines in CDL format needs to be replaced in Circuit A, the time would be 15 seconds. Five seconds for compilation and replacing the module in the library, three seconds for loading the hierarchy and seven seconds for producing the flattened data.

10. Conclusion

On the average, the compiler can processes **2000** lines of CDL text a minute, and the loader and flattener process **10,300** flattened nets a minute. These times show that incremental circuit changes can be performed in analysis tools without using a separate compilation procedure.

This efficiency is possible by creating data structures which require minimal manipulation and by the organization of data in the circuit library. Memory utilization is kept to a minimum by optimizing the location of circuit information on the data structures.

11. Acknowledgment

The author would like to thank Paul Shupe for his contributions on the paper, Rob Walker, Gary Beihl and Tim Jennings for their help on CDLCON, and other individuals at Caedent Corporation for their support on the CDLCON project.

12. References

1. Apollo is a trademark of Apollo Corporation.

2. Berg, W.C., et al., "Performance of Probablistic Fault Grading", VLSI Design, January (1986) pp. 40-45.

Description	Circuit A	Library 1	Circuit B	Circuit C	Library 2
Lines of CDL	1150	500	880	5160	190
Size of Circuit Library in bytes	36,793	11,173	31,976	166,893	3271
No. of Modules used in Design	25	83	60	144	98
Maximum Hierarchical Depth	3	N/A	3	6	N/A
No. of Flattened Nets	1773	N/A	1072	30,829	N/A
No. of Flattened Instances	1751	N/A	998	27,550	N/A
Compile Time	34	13	28	161	5
Load Time	3	N/A	4	16	N/A
Flatten Time	7	N/A	5	124	N/A
Total Time	44	13	36	301	5

TABLE 1 Performance Data