UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

ESCUELA TECNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACION

PROYECTO FIN DE CARRERA

NAC- NLC & CADENCE : DESIGN FLOW MANAGEMENT IN CADECE DFW II

JUDITH C. E. RUSSELL

Las Palmas de Gran Canaria, 1995

NAC - *Nlc & Cadence*: Design Flow Management in Cadence DFW II

A Dissertation

submitted in partial fulfillment of

the requirements for the degree of

MASTER OF SCIENCE

in Computer Science and Applications

in the Faculty of Science,

the **Queen's University of Belfast**,

and in conjunction with

Escuela Técnica Superior de Ingenerios de Telecomunicación,

Universidad de Las Palmas de Gran Canaria

by

Judith C. E. Russell BSc Hons

September 1995

Tutors: **Pedro P. Carballo** and **Tomás Bautista**

Acknowledgments

Special thanks to my supervisors, Pedro P. Carballo and Tomás Bautista, for all their help and encouragement while working on this project. Thanks to Mike McKeag and Valentín de Armas Sosa for co-ordinating this ERASMUS exchange, and also to all the people at C.M.A. whom I have met in the past four months and helped make me feel very welcome at this university.

Contents

List of Figures

Abstract

Design Flow Management is a design framework service that helps designers to keep track of their design activities and to maintain an overview of the state of the design. The user interface component of the design flow manager informs designers about the state of the design and allows tools to be started.

Cadence Design Framework II (DFW II) has a mechanism to integrate design process in a design methodology. It is very useful when the design flow is clear and well understood. A group of tasks can be done in a single pass and the system takes the decision to let the designer continue with the next step, when the previous step is correct.

The main objectives of this project are to generate a general methodology in the designing of a design flow in Cadence, and to develop an nlc synthesis tool in Cadence by creating functions using the SKILL language and creating flowcharts using the graphical editor.

Chapter 1

Introduction

An overview of the evolution of Computer-Aided Design (CAD) frameworks in the area of Electronic Design Automation, the necessity for framework standards, and an outline of the characteristics and requirements of a CAD framework.

1.1 The evolution of CAD frameworks

Since the late 1960s, the rapid growth in the complexity of integrated circuits (IC's) and digital systems has led to an even more rapid growth in the complexity of the software tools and associated data needed to represent a design. Communication between tools was possible only if a translator for the respective formats was available. The emergence of de-facto standard formats, allowing the number of translators to be reduced, eased this problem a bit. It was realized that effective **Electronic Design Automation** (EDA) solutions not only had to provide the individual tools but also the integration facilities to support the communication between tools.

EDA brought about a great variety of loosely coupled tools to perform the many design tasks. But because the tools supported only individual design tasks, the problems of handling a multitude of tools and of successfully moving design data from one tool to another arose for the designer. What was lacking was *integration* and overall support for managing the design process. To achieve gains in productivity, attention needed to be paid to the overall efficiency of the design process. Enter **CAD Frameworks**. A CAD Framework, as defined by the CAD Framework Initiative (CFI), the international consortium developing framework standards, "is a software infrastructure that provides a common operating environment for CAD tools"[6]. It is generally agreed by the industry as a whole that advanced CAD frameworks could turn collections of individual tools into effective and user-friendly design environments.

Basically a CAD framework had to provide facilities for integrating multiple CAD tools into a coherent design environment, i.e. *tool integration*, and it has to support the end-user in conveniently operating the design environment, i.e. to become the *electronic assistant* of the designer for organizing the design information and managing the design process. There are two main categories of framework users: *developers*, i.e. CAD tool developers and CAD tool integrators, and *end-users*, i.e. design engineers, administrators and project managers.

Integration versus Encapsulation One aspect of tool integration is the actual *technique* used to perform the integration. In the field of EDA, it is normal to distinguish between *integration*(sometimes known as *tight integration*) and *encapsulation*. Upon integration, the source code of the design tool is modified to include code that handles the interaction with the CAD framework. Upon encapsulation, the source code of the design tool is not modified. Instead, a *wrapper* of additional code is written, to loosely interface the tool to the CAD framework. Figure 1.1 illustrates both types of tool integration.

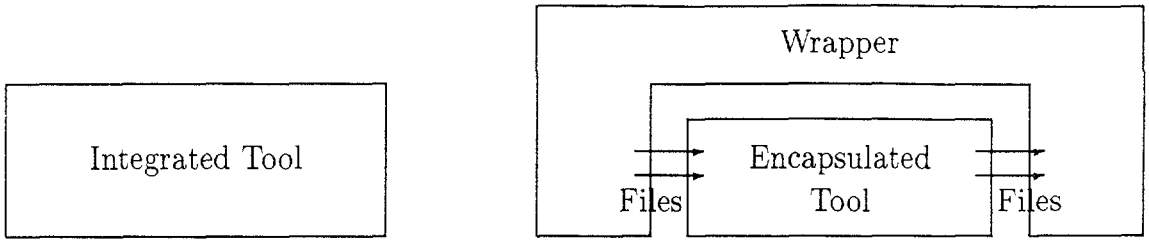


Figure 1.1: Integration versus Encapsulation

There are three main roles allotted to CAD frameworks:

Common Design Database - this is the key to tool integration and tool interoperability, since data which is common to a number of CAD tools is stored only once in the database from where it can be used for input for all tools. The design database may be a file-based system, implemented on top of the host operating system, such as in DFW II from Cadence Design Systems, Falcon from Mentor Graphics, and the Nelsis CAD Framework Release 4 [3]. Another approach is to employ a conventional Database Management Systems (DBMSs) to fulfill the role of the design database, or even object-oriented DBMSs which would offer more flexibility for handling interrelated data of different granularities on which different types of access are performed.

Design Data Manager - the management of the design database can aid the designer in the organization of his design information. It uses knowledge of the structure and status of this design information to provide support and enforce constraints on the design process. Management services may then enable a user interface to be added to allow the end-user to interact with the system to get informed about the structure and status of his design.

Design Process Manager - with the increasing number of tools found in today's CAD systems, there is a growing need to support the design engineer in correctly executing these tools to perform his design tasks. Framework services may help him in correctly invoking the individual tools, as well as provide support for executing the tools in the correct order, according to a pre-defined design procedure. The framework could automatically execute design tools , for example, when valid output data is required for a subsequent tool run. Two services that come under the role of the process manager are *design methodology management* and *design flow management*. Design methodology refers

to the definition of a design procedure in terms of abstract design stages, such as "circuit and layout design" or "circuit verification", and design flow refers to the definition of a design procedure in terms of individual tools and dependencies between tools.

A CAD framework will provide *environment stability* as it offers a standard operating environment to the ever evolving set of CAD tools. It also promotes *modularization* of CAD systems, as these will be constructed as cooperating tool components on top of a CAD framework, rather than being implemented as monolithic super-tools.

1.2 The need for CAD framework standards

There seems to be a world-wide industry consensus that there is an urgent need for CAD framework technology in order to improve design productivity by building effective integrated design environments. The EDA community appear to have agreed on the major functional requirements for a CAD framework, however, there is no common ideology on how to go about developing and integrating one.

The charter of the CFI, a non-profit consortium of CAD tool users, tool vendors and research institutions, is to define "interface standards that facilitate integration of design automation tools and design data for the benefit of end users and vendors worldwide" [4].

CAD users and vendors have encounter difficulties when integrating design tools into their production environments, so the primary aim of the CFI is to define standards and guidelines that will allow tools from different sources to cooperate in a single design environment. This ability to easily incorporate a CAD tool into a design environment, is referred to as "plug and play".

Another framework effort is the **Jessi-Common-Frame** (JCF) project [1]. This European project was started in 1990 and involves many companies and institutes, including Siemens Nixdorf Informationssysteme AG (SNI), Philips, ICL, and Delft University of Technology. The JCF is building a framework, the Jessi-Common-Framework, which is intended to be compliant with the standards defined by CFI.

1.3 The characteristics and requirements of CAD frameworks

Some of the key characteristics of the CAD framework architecture are:

- *openness*, the ability of the CAD framework to easily incorporate new tools, data and design methodologies,
- *efficiency*, which implies that overall efficiency in the design process is optimized,

- a *user-friendly* interface that is consistent and intuitive in appearance and behaviour,
- *flexibility and modularity*, the system must be able to evolve smoothly to satisfy new requirements

The primary functional requirements of a CAD framework are:

- to facilitate *tool interoperability*, the cooperation among tools from multiple sources,
- *tool interchangeability*, the ease of tool replacement within a design system,
- *multi-user, multi-tasking support*, multiple users must be allowed to operate concurrently on a design project in a controlled and coordinated manner, i.e., supporting cooperation between members of a design team,
- to guarantee *consistency and integrity of design information*, i.e., managing the process of change, inherent in evolutionary design, to ensure correctness of each portion, as well as the whole, of the design.
- supporting the *configuration management*, the management of a collection of related design objects,
- *tool management*, the assisting of the design engineers in conveniently executing tools,
- *design flow management*, the assisting of the design engineers in correctly and efficiently performing design activities.

Chapter 2

Cadence Design Framework II

An outline of the Design Framework II environment from Cadence Design Systems Inc, the graphical user interface and the design process.

2.1 Overview of the Cadence DFW II

Cadence provides EDA software and services that automate and enhance the design of integrated circuits (IC's) and electronic systems. The Cadence Design Framework (DFW II) architecture is the foundation of the Cadence design environment. The DFW II architecture solves many problems traditionally associated with EDA systems, such as having to master different user interfaces for different parts of the design process. The DFW II software provides a common interface for all integrated tools, a common functionality, and a common database.

The Cadence software consists of mainly application programs for tasks such as layout, compaction, and verification. These applications run to many hardware platforms. Cadence's open architecture also permits integration of your own tools from other vendors. In such a setting it is important that the tools and data be well integrated. DFW II provides that integration.

The main features of DFW II are:

- *software portability* which lets Cadence software run on a variety of hardware platforms,
- the *conformance to open system standards* which lets you integrate non-Cadence tools and design data, e.g., by using the industry-standard EDIF interchange format. By directly accessing the DFW II unified database, you can also integrate your own tools into the DFW II environment,
- a *consistent graphic interface*, based on the industry standard Motif window manager, makes all Cadence tools easy to learn
- a *unified database* eliminates data conversion
- having a *versatile, modular and configurable environment* means new tools that are integrated interact with the other DFW II tools, providing a more powerful design environment,
- the *powerful high-level SKILL interactive programming language*, which is the Cadence extension language and is based on the artificial intelligence language LISP, but which supports a more conventional C-like syntax,

- the *menu-driven* interface enables you to execute many SKILL commands from a menu instead of always through the command line,
- the *coordination and protection of design work* by checking them in and out.

2.2 The Cadence User Interface

Cadence runs under is the **UNIX operating system**. Cadence runs under the windowing system **X Windows**, which enables you to manage several tasks at one time from separate areas of the screen called windows. A window manager controls the size, placement and behaviour of windows. Cadence software follows the standards established by the Open Software Foundation in its **Motif Window Manager**, see figure 2.1:

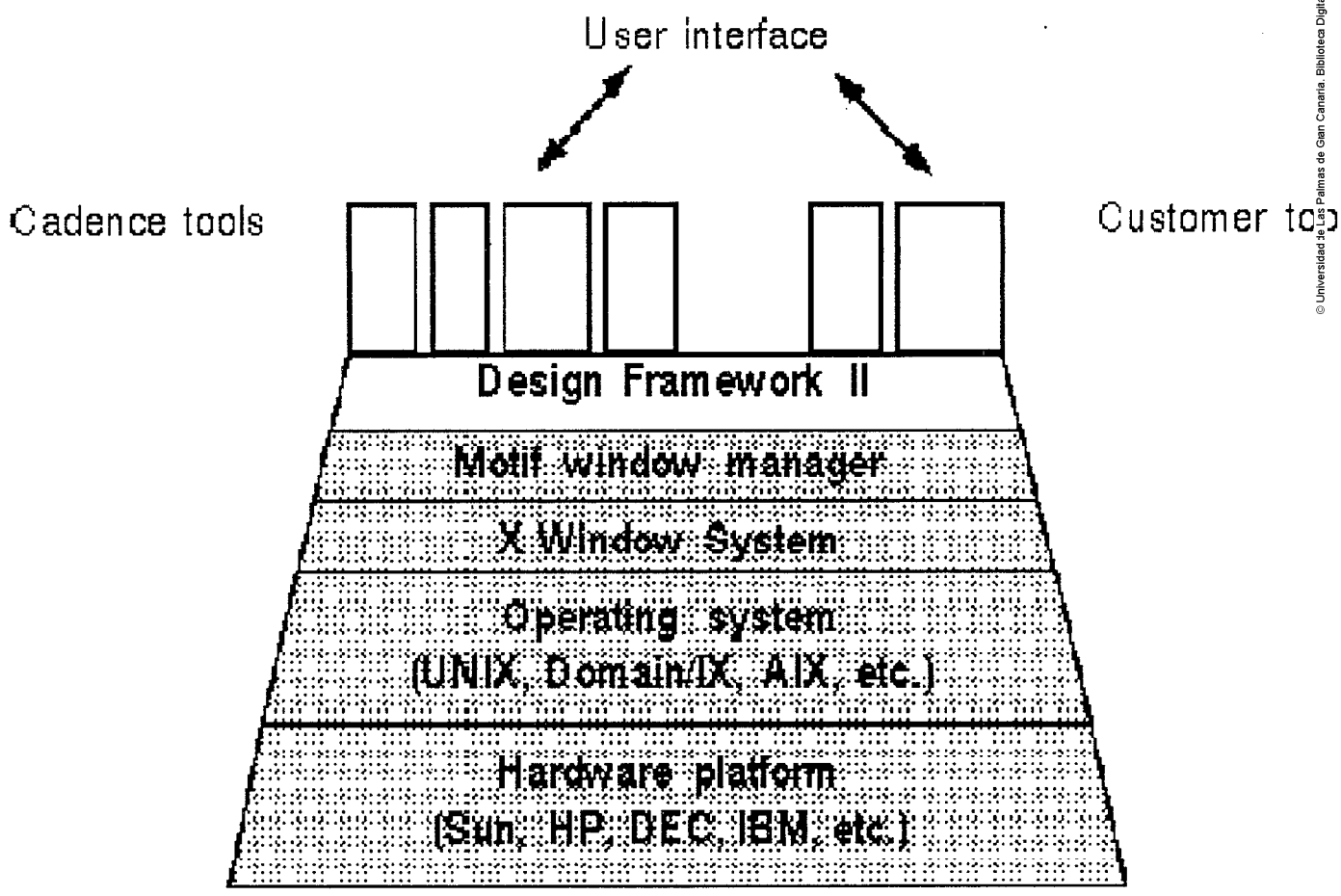


Figure 2.1: Layers of Software

Cadence software has a graphic user interface based on windows, forms and menus.

Windows - there are 5 major kinds of DFW II windows:

- Command Interpreter Window (CIW) - controls the design process
- Flowchart Browser Windows - display flowcharts that graphically represent steps in the design process
- Library Browser Windows - display a tree structure with the names of designs that you can choose for work
- Design Windows - display your designs; you can have several open at once, each running at the same time; the only design window associated with DFW II is the **Icon Editor Window**
- Text Windows - display text files; can have several open at once

Forms - provide a place to enter information that a command must have in order to execute. There are 3 types of DFW II forms:

- Standard Form - appears when you choose a menu command that is followed by three dots
- Options Form - appears during a task, giving you a chance to enter settings as you work
- Properties Form - appear when you choose *Properties* from a menu

N.B. Cadence database objects have *attributes* and *properties*. Attributes are inherently part of the object and cannot be deleted, or added to, though their values can be changed. Properties are characteristic traits that you can add, delete, or modify.

Other features

- Pop-up Menus - contain commands related to the task you are doing
- Dialog Boxes - give you information, warnings or cautions and require you to respond
- List Boxes - display a list of items for viewing and selecting

2.3 The Design Process

Cadence designs are organized into *libraries*, which are directories on your hard disk. There are two types of libraries: *reference libraries* which contain basic design objects, and *design libraries* which contain your ongoing design work. Each library contains cells, views and cellviews. A *cell* is a database object that forms a fundamental design unit. A *view* (e.g. schematic) is a way of representing that object on the screen. A *cellview* is the design you see of a certain view of a certain cell. Each cell is assigned a *version number* so it can be tracked. Cellviews must be *checked out* of the library before work, and *checked in* to the library after work is completed. The *Library Browser* shows you a tree structure of the libraries, cells, views, cellviews and versions in the library.

Using SKILL commands, you can create flowcharts that show the steps of the design process at your workplace. Once the flowcharts are created, you can use the *Flow Browser* to view information and run the design steps. You start your design by including *virtual copies* (representations) of *master cells* from the reference library or the design library. Then you connect, add to, or modify the cells. The virtual copy of the master cell that is included in your cells is an *instance*. As you build your design by including simple instances within more complex ones, you create a *design hierarchy* with multiple *levels*. You can move between levels and work at any level. When you open a design you see the *top level*, unless you specify otherwise. Information that affects all the designs in the library is stored in files known collectively as the *technology file*.

Chapter 3

Design Flow Management

A description of Design Flow Management with particular reference to the Cadence Design Flow.

3.1 Overview of Design Flow Management

Design Flow Management (DFM) is a framework service that helps designers to keep track of their design activities and maintain an overview of the state of the design. The underlying notion DFM, or *Design Methodology Management*, is that chip design is a process, involving a sequence of operations, performed on design data. DFM software attempts to capture and automate that process [5].

The DFM system is viewed as a meta-tool in the CAD environment, both in the sense that it deals with other tools as data, manipulating them to meet some design goal which goes beyond the scope of any individual tool, and also in the sense that it "packages" groups of tools into higher-level entities which may be manipulated by the user as a single tool. Through DFM it is possible to automate tedious sequences of tool invocations, such as an *edit netlist simulate cycle*. Under a DFM system the user leaves some of the decision-making up to the Design Flow Manager. A number of attempts have been made to provide DFM either as part of an integrated CAD system, or as part of a stand-alone framework.

The key DFM issues are (from an end-users perspective):

- *design tracking* - keeping track of the state of the design and the design history
- *constraint enforcement* - permitting constraints on the design process to be defined and enforced
- *guidance and automation* - supporting the design engineer in efficiently executing design activities

DFM is difficult to do without standards, as it needs built-in knowledge about the execution environments of individual tools. Without a standard interface to data management it is difficult to determine the state of the design. In the absence of a single standard database for electronic CAD, data interchange standards such as Electronic Design Interchange Format (EDIF) have begun to make it possible to link groups of tools into sequences. DFM will be more valuable with standards in user interfaces. The X-Window system not only supports multiple simultaneous application displays, but it also allows display on a single screen of the output from programs running on multiple hosts. A single data model, shared by the CAD tools and the DFM in a single environment, will greatly simplify tool management tasks, both at the tool integration level and the DFM level.

3.2 The Cadence Design Flow Management System

Cadence DFM System is an extension-language-based flow system. The design flow engine is driven by a set of data structures called *flowcharts* and *design steps* which describe tasks and task dependencies using hierarchical directed graphs. Branching and looping capabilities add to the richness of the model. Each step (node on the graph) is defined in terms of a set of procedures and data, defined in the SKILL extension language, which are activated by the design flow engine as required. The graphical mode is supported by the graphical user interface which illustrates the flow graph and supports user interaction through direct manipulation.

The Design Flow system provides a simple way to encapsulate internal and external tools. Once a tool is encapsulated, it can be shared among users and its behaviour can be further customized. The Design Flow is flexible enough to support either a strict methodology, for novices, or a loosely enforced methodology, for expert users. A strict methodology enables novice users to learn the design process without having to be concerned about how to start the tools involved. A lose methodology allows expert users to sue any of the steps in the design process in any order.

The Flowchart Browser The *Flow Browser Window* displays a flowchart created by the Cadence Design Flow system. It consists of a system pull-down menu and an area for displaying a single flowchart. The *Flowchart Browser* is the means by which you interact (invoke, query, reset) with a flowchart and the design steps it contains. A *Design Step* is a description of a tool, sequence of tools, or a decision point(switch box) encapsulated to work within the Design Flow system in DFW II. It provides a means of graphically viewing the design steps and their interrelationships.

A *Flowchart* links the design steps together conceptually and graphically to indicate the ways in which the design flow is controlled and the ways in which the data moves between design steps. A *node* represents a design step and the *arcs* between the nodes guide the designer by indicating which of many possible design steps might logically follow another step. *Flowchart and Design Step Instances* keep information regarding each design session. They are created each time a flowchart is displayed. They inherit all the properties of their masters, such as dependency information, data flow and node appearance. There are 2 types of flowchart steps: *design steps* - shown as rectangles on the flowchart, and *switch boxes* - shown as diamonds on the flowchart. See figure 3.1 for an example of a flowchart in a Flow Browser window. The following are design flow commands in the Flow Browser window:

1. The system menu on the Flow Browser window banner saves a flowchart to a new design, redraws a flowchart, and closes the Flow Browser window

2. Commands on a design step run the design step, show its properties, and display a subflowchart in the design step refers to one (shown as a downward arrow)

3. The default commands associated with a pop-up menu on a design step are the *run step*, *reset step* and *push to display the subflowchart*, although you can customize this and add other commands to the pop-up menu

The Design Flow Tools Within the Design Manager is the *Top Flow Browser*. From here you can access four design tools, namely

1. PIC Designer - a tool in the Cadence CAE design software series which is a complete set of software tools for designing complex systems, such as
   - Printed Circuit Boards (PCBs)
   - Field-Programmable Gate Arrays (FPGAs)
   - Complex Programmable Logic Devices (CPLDs)
   - Standard off-the-shelf components
2. ASIC Designer - the ASIC Workbench is an integrated system of design tools, data flows, and development procedures for designing, implementing, testing, and validating the physical layout of Application-Specific Integrated Circuits (ASICs). The ASIC designer starts by creating the logical or structural designs and verifying that they work correctly, this is the *front-end* of the design process, and finishes with the physical layout of the circuit
3. P&I Designer - the Placement & Interconnection design flow enables the placement and routing of packaged parts on a layout, and the back annotation of physical changes from the layout to the schematic; this is the *back-end* of the design process, via the Cadence tools Composer and Allegro
4. Verilog Simulator - this can be run as a stand-alone tool or in the Logic Workbench to simulate the results of PIC Designers. With the Verilog Simulator, the simulations can be interactive or batch and it is possible to view the waveforms interactively or postprocess using cWaves (the Cadence waveform tool)

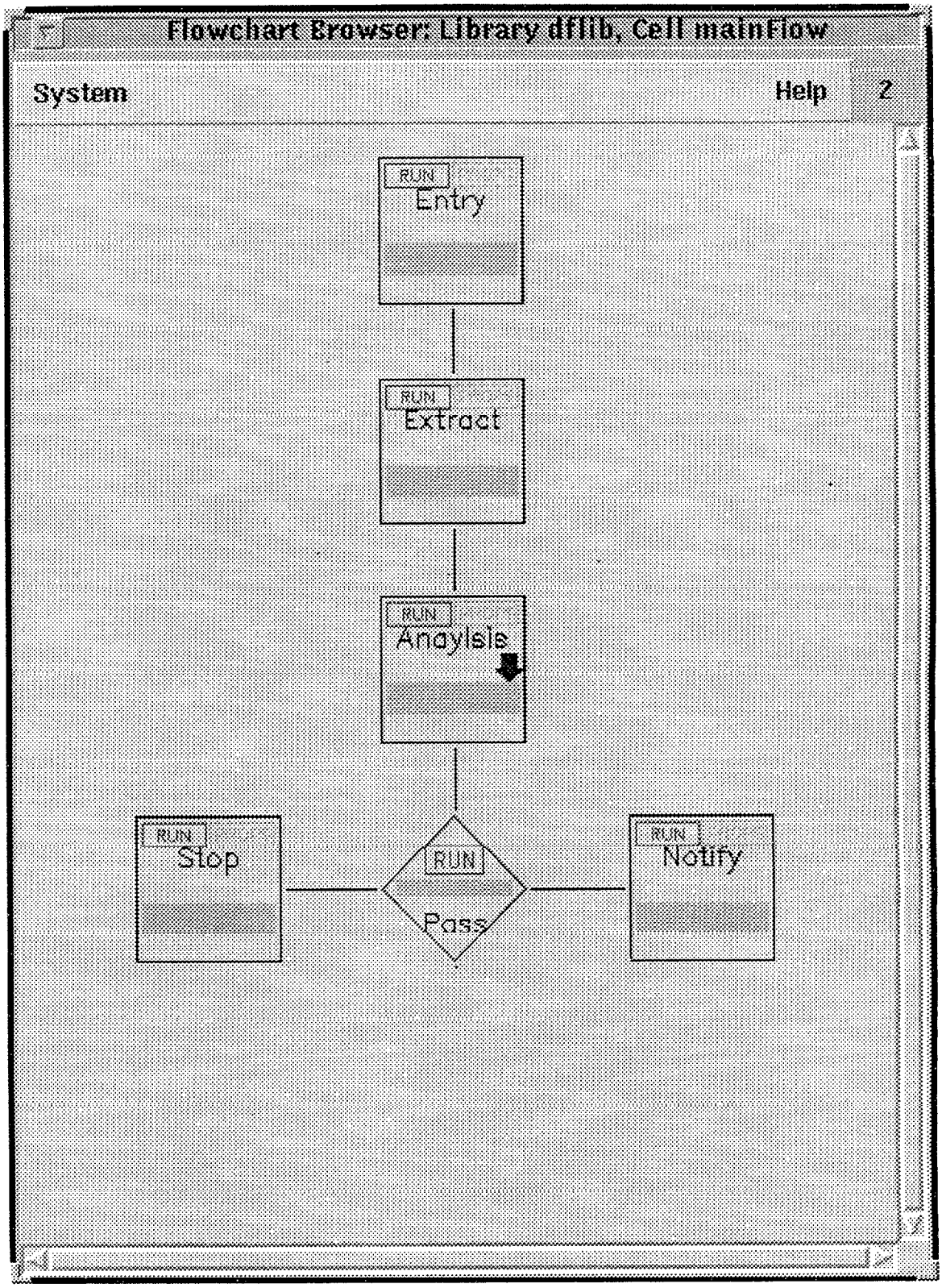


Figure 3.1: An example of a flowchart in a Flow Browser window

Chapter 4

SKILL Programming Language

An overview of the SKILL programming language, SKILL functions and how to develop them, and the SKILL development environment.

4.1 Introduction to the language

SKILL is the Cadence extension language. It is a high-level interactive programming language based on the artificial intelligence language LISP, however, it supports a more conventional C-like syntax. SKILL brings a functional interface to the underlying subsystems to the command line and lets you customize existing CAD applications and help you develop new applications. SKILL is ideal for rapid prototyping because you can incrementally validate the steps of your algorithm before incorporating them in a larger program. SKILL has automatic storage management which relieves you program of the burden of explicit storage management. It can also control notoriously error-prone system programming tasks like list management and complex exception handling.

Relationship to Lisp

- In SKILL, function calls can be written in either of the following notations:
  1. **Algebraic** notation - used by most programming languages, eg:

     ```
     func( arg1 arg2 ... )
     ```

  2. **Prefix** notation - used by Lisp programming languages, eg:

     ```
     ( func arg1 arg2 ... )
     ```

- Example of a SKILL program in the first notation:

```
procedure(fibonacci(n)
   if( (n == 1 || n == 2) then
     1
   else fibonacci(n - 1) + fibonacci(n - 2)
   )
)
```

- Same example of a SKILL program but in the second notation:

```
(defun fibonacci(n)
    (cond
        ((or (equal n 1) (equal n 2)) 1)
         (t (plus (fibonacci (difference n 1))
            (fibonacci (difference n 2))))
    )
)
```

4.2 What is a SKILL function?

A SKILL function is a "named, parameterized body of one or more SKILL expressions. You can invoke any SKILL function from the command input line available in the application by using its name and providing appropriate parameters" [2]. All SKILL functions compute a data value known as the return value of the function. A SKILL function is called by stating its name and arguments in a pair of parentheses.

Invoking a SKILL Function There are many ways to submit a SKILL function to the SKILL interpreter for evaluation. In many applications, whenever you use forms, menus, and bindkeys, the Cadence software triggers corresponding SKILL functions to complete your task. Normally you do not need to be aware of SKILL functions or any syntax issues (see figure 4.1).

- *bindkeys* - associate a SKILL function with a keyboard event. When you cause the keyboard event, the Cadence software sends the SKILL function to the SKILL interpreter for evaluation
- *forms* - some functions require you to provide data by filling out fields in a pop-up form
- *menus* - when you choose an item in a menu, the system sends an associated SKILL function to the SKILL interpreter for evaluation
- *CIW* - SKILL functions can be entered directly on a command line in the CIW to bypass the graphical interface
- *SKILL process* - you can launch a separate UNIX process that can submit functions directly to the SKILL interpreter

A collection of SKILL functions can be submitted for evaluation by loading a **SKILL source code file**. The compiler translates the source code into a target representation which might be machine instructions or an intermediate instruction set. The SKILL interpreter then executes SKILL programs within the Cadence environment. It translates the text source code into internal data structures, which actively consults the execution of the program.

SKILL programs are represented as lists and therefore can be manipulated like data. The ability to manipulate data, i.e. dynamically creating, modifying, or selectively evaluating function definitions and expressions, is one of the primary reasons why SKILL is based on Lisp. SKILL can be used to write flexible and powerful applications because it takes full advantage of the "program is data" concept of Lisp. SKILL supports a special notation for list construction from templates and so eliminates the long sequence of calls to *list* and *append*.

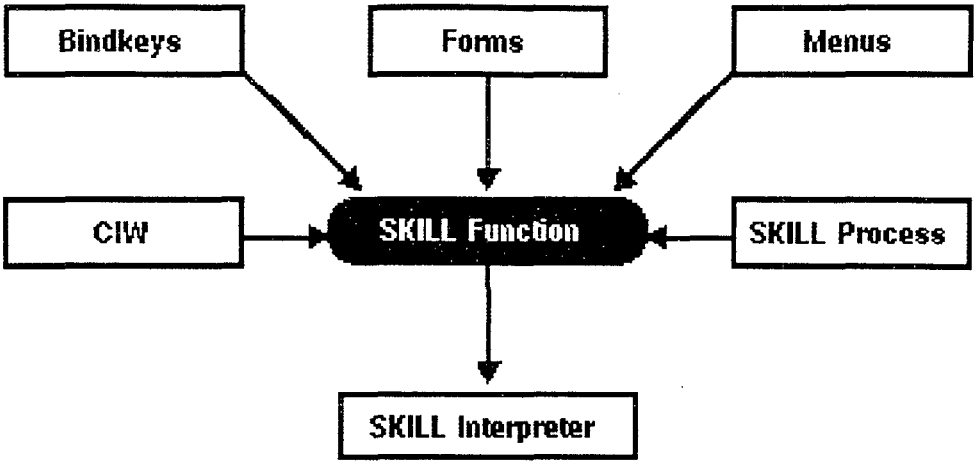


Figure 4.1: Invoking a SKILL Function

Developing a SKILL Function Developing a SKILL function includes the following tasks:

1. Grouping several SKILL statements into a single SKILL statement - using { } to group a collection of single SKILL statements into one. The return value is the return value of the last single SKILL statement (N.B. as normal, you can assign this return value to a variable)

2. Declaring a SKILL function with the *procedure* function - to refer to groups of SKILL statements by name, use the *procedure* declaration to associate a name with the group. The group of SKILL statements and the name make up a SKILL function. To execute the group of statements, mention the function name followed immediately by ().

3. Defining function parameters - to make your function more versatile, you can identify certain variables in the function body as formal parameters, therefore, when you invoke a function, you supply a parameter value for each formal parameter

4. Maintaining your source code - the Cadence environment makes it easy to invoke an editor of your own choice. Simply set the SKILL variable *editor* to a UNIX command line able to launch your editor, for example:

   editor = "x-term -e vi"

   The *ed* function invokes an editor of your choice or alternatively, you can use an editor independent of the Cadence environment

5. Loading your SKILL source code - the *load* function which evaluates each expression in the source code file, returns *t* if all expressions evaluate without error, and aborts

if there are any errors. Any expression following the offending expression is not evaluated

6. Redefining a SKILL function - the SKILL interpreter has an internal switch called *writeProtect* to prevent the virtual memory definition of a function from being attached during a session. By default it is set to *nil*. SKILL functions defined while *writeProtect* is *t* **cannot** be redefined during the same session

4.3 The SKILL Development Environment

The Cadence environment allows SKILL program development such as user interface customization. The SKILL development environment contains powerful tracing, debugging and profiling tools for more ambitious projects. SKILL allows you to access and control all the components of your tool environment: the User Interface Management System, the Design Database, and the commands of any integrated design tool. The SKILL Development Toolbox provides software tools that reduce the time it takes to develop SKILL code and that improve the efficiency and quality of the code. There are five main applications within the SKILL Development Environment which are discussed next.

SKILL Debugger The SKILL *Debugger* helps you debug SKILL code by examining the stack, single stepping, tracing, and setting breakpoints. It is accessed via the SKILL Development toolbox or by typing the SKILL command *ilDebugToolBox()* at the command line in the CIW. After the installation, when you run you code and and error occurs, you enter the debugger automatically. If you reach a breakpoint, regardless of whether the debugger is installed, you enter it. The *Dump Stacktrace,* and *Where* commands are used to display the SKILL stack and local variables when you run the code and errors occur. The *Set Breakpoints* is used to set breakpoints at different stages in you code to see where you went wrong. Each time you exit the debugger, SKILL exits the most recently entered (nested) debugger session.

SKILL Lint The SKILL *Lint* examines SKILL code for possible errors and inefficiencies. It is useful for detecting errors not found during normal testing, and helps you spot unused variables and global variables that are not declared as locals. When you choose the SKILL Lint from the SKILL Development toolbox, or type the SKILL command *skShowForm()*, enter the filename or context that you what analyzed. It is possible to check for:

- Errors - messages about a SKILL error that occurs if the code is executed
- Warnings - messages pointing out potential errors and areas where you should clean up your code

- Undefined functions - lists all the functions that cannot be executed in the executable form which you ran SKILL Lint
- Performance - messages that give hints or suggestions about potential performance problems in your SKILL code

SKILL Profiler The SKILL *Profiler* helps you analyze the performance of your functions and determine where the most time is being spent. The SKILL Profiler:

- measures the time spent in each function that executes longer that 1/60th of a second
- shows how much SKILL memory is allocated in each function
- measures performance without having to modify function definitions
- displays a function call tree graph of all functions executed and the time or memory spent in those functions
- allows you to filter functions so you can see only those functions in which you are interested

Code Browser The *Code Browser* displays the calling tree of a SKILL function. The calling tree shows the child functions called by the parent functions. Is is possible to expand the entire tree or one node at a time. When you choose the Code Browser from the SKILL Development toolbox or type *skCodeBrowser()* at the command line, you simply specify the function you wish to expand. The Functions pop up menu is available which lets you:

- view the source code of that function
- expand the function to display the children functions of the node selected
- expand "deep" to display all the functions recursively until the entire calling tree is expanded

Tracing *Tracing* lets you trace SKILL function calls and property and variable assignments. When you choose the Tracing option in the SKILL Development toolbox or type *ilTraceForm()* you select the type (functions, variables or properties) and the way in which you want to trace (by name, in context, matching regular expression, user functions, or all).

Chapter 5

Interprocess Communication

An outline of the concept of Interprocess Communication and the integration of tools to the Cadence DFW II.

5.1 Outline of Interprocess Communication

Interprocess Communication (IPC) facilities permit processes to exchange or share data. A *process* is when a program is actually running on a particular machine. Unix is an example of a multi-tasking operating system, as it can simultaneously execute multiple programs. The best IPC method for a given application depends on the structure of the communicating programs, the amount of data and kind of data that must be passed, the requirements for operation in a distributed computing environment, and the demanded performance.

Processes can communicate by writing to and reading information from *files*. The main advantages of file IPC are unlimited capacity and multiple delivery. The classic Unix mechanism for IPC on a single machine is the *pipe*, which employs the basic byte-stream model used for file I/O. A process may provide data for direct consumption by another concurrent process via a pipe. In a distributed computing environment however, processes living on different machines may need to exchange or share data. Unix offers facilities for IPC between processes on different machines according to the file I/O byte stream model. These transport facilities are known as *sockets*. They provide the basic service of reliable, end-to-end data transfer across the network.

The *Network File System* (NFS) is a facility for sharing files in a heterogeneous environment of machines connected to a network. In this kind of environment, different machines may be attached to the network which employ different representations of data. Therefore, upon transfer of data between different computer architectures, conversion between representations must be performed. NFS makes all the disk space available as needed, therefore, individual machines have access to all file-based information residing anywhere in the network.

A *Remote Procedural Call* (RPC) service can be implemented on top of a transport facility such as the IPC socket facility. The basic idea of RPC is to extend the use of procedure calls to a distributed environment. With RPC, a *client* process can have a procedure executed by a *server* process, which may be running on a different machine. In other words, a single thread of control logically winds through two processes. When a remote procedure is called, the client is suspended, a message containing the arguments is constructed and passed to the server process which then executes the procedure. When the procedure finishes, the results are passed in a message back to the client process, and the client resumes as if the procedure had run locally. RPC is a simple but efficient mechanism for communication for distributed programs. When an RPC server program implements one or more remote procedures, the procedures, their parameters and the

results are part of the specific program's *protocol specification.* The protocol specification is a set of procedures that are agreed upon by the client and server processes.

In a distributed computing environment, multiple tools can be operated concurrently at the same or different locations. The idea of mapping each *running tool* to a Unix *process*, fully matches the paradigms of Unix and the X Windows System in that multiple (graphical) applications can be run on one or more machines from multiple windows on a workstation. This concept of a multiple server processor serving the tool processes *in parallel* is most effective if the individual servers can operate independently from each other. A client, therefore, may be connected to multiple servers at the same time. This multiple server - multiple client process organization *scales well* with the size of the design environment. The multiple servers can effectively utilize the distributed computing power in the workstation environment.

5.2 IPC in Cadence

Tools can communicate in one of three ways:

- Point-to-point through sockets and pipes
- Common database through files and a distributed database management system
- Communications Manager through the Application Programming Interface

Point-to-Point Communication - Traditionally, a tool used point-to-point communication to exchange data directly with another tool, the data was private because only two tools shared it (see figure 5.1). One tool needed specific knowledge of the other tool. With point-to-point communication, you must create multiple IPC links when you have many tools. Maintaining and adding new links can be complicated and tedious.

Common Database - Sharing a common database is not practical when only a small amount of data is shared. It can be difficult and inefficient to synchronize processes and events through database transactions. A standard between vendors and customers would also have to be established.

Communications Manager - The CMan represents a major change from the traditional model of directed IPC requiring a connection between each pair of cooperating tools. With the CMan, each tool only needs to be connected to the server process to communicate with every other tool from the session. Adding and deleting tools from the session can be done with little impact on the other tools.

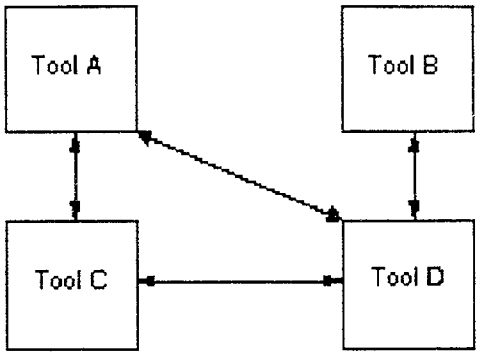


Figure 5.1: Point-to-Point Communication

Most existing tools are stand-alone and pass information through files. This method works fine until it is necessary to coordinate the activities of the stand-alone tool and the tool set as a whole. The Cadence Communications Manager offers a comprehensive solution for tools which must communicate interactively. It is a second-generation inter-tool communication system incorporating many essential concepts that allow developers to work at a higher level. The Communications Manager is one component enabling software engineers to utilize Cadence framework technology to integrate their tools into a complete design solution. Other integration products include:

- **User Interface Manager** - which lets interactive applications share a common user interface with DFW II-based tools
- **Open Simulation System** - which easily builds interfaces with tools that need connectivity data or are netlist driven
- **SKILL** - which gives complete access to the DFW II environment through a high-level extension language

5.2.1 The Communications Manager

The Cadence Communications Manager (CMan) enables any number of tools in the design process to interact and communicate together, exchanging data easily. Each tool used in the design process has its own view of the design data, but this data must be shared with other tools. Operations that occur in one tool can effect operations in others. The CMan allows a multitude of tools to exchange events and data by using its conceptual data bus. As new tools are added the existing tools do not need to be changed because it is not a point-to-point system. Programs developed with the Communications Manager benefit from:

- "Plug and Play" architecture, allowing new tools to connect to the design framework easily

- Multi-cast communications system allows new tools to be added easily without changes to existing tools
- "Message scenarios" which abstract tools from inter-tool dependencies making integration easier
- Transparent support for distributed processing in heterogeneous environments

Each Communications Manager client application (or tool) interacts with the Communications Manager using calls to the **Communications Manager Application Programming Interface (API)**. The CMan API is required to develop applications in C using the Communications Manager. The API allows a software developer to write new programs that work with applications that are integrated with DFW II. In figure 5.2 the *libcman.a* object library contains the API functions, and each tool must link to this library to access the Communications Manager functionality.

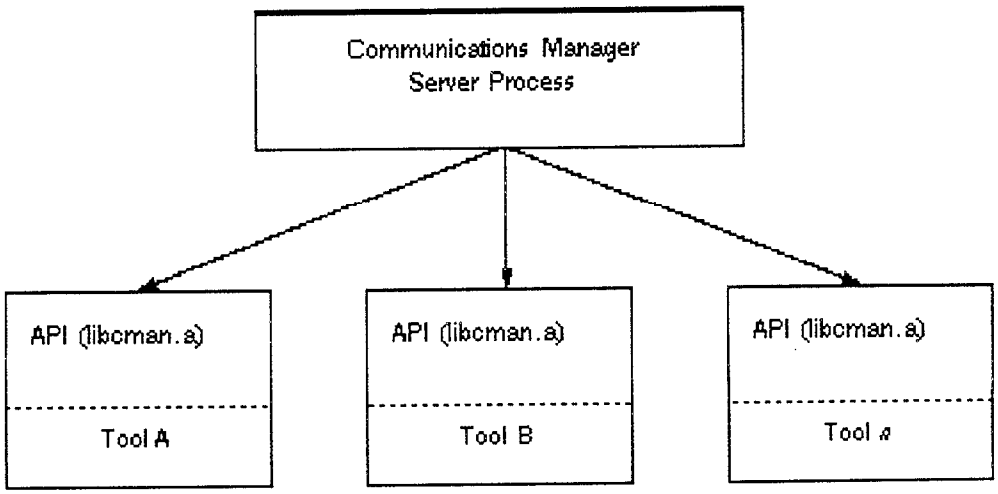


Figure 5.2: Interprocess Communication in Cadence

The **Communications Manager Server Process**, which is bundled with DFW II, coordinates all communication traffic between its tools. Each tool connects directly to the server process through a network connection.

5.2.2 Communications Manager Operations

CMan provides two fundamental services for tools which must interact: **notification** and **data transfer**. They are, however, separate and distinct operations. A CMan notification is simply a message, transferred to all tools in use, stating that some event has occurred. While multiple tools may be interested in events that occur during the design

process, not all need to process the data or to respond to the event. CMan prevents overloading tools with unnecessary data. By separating data transfer from the notification process, CMan offers a highly flexible and efficient communications system.

To reduce communication overhead, the Communications Manager API allows tools to express interest in notifications they wish to receive. By using the API, programmers register C functions as handlers for these notifications. The Communications Manager server forwards notifications only to the tools that have expressed interest in them. New tools can participate in existing message scenarios by expressing interest in documented notifications, or by creating their own message scenarios without affecting existing tools. Another advantage of notifications being separate from data transfer is that tool developers are not required to anticipate what data some future tool will want for a particular operation.

When an application sends a notification and expects the receivers to import a small data item in response, CMan provides a **Notify-with-Cache** option, where the data item is send with the notification. The tool receiving the notification imports the data item in the usual way. Therefore, there is no additional network traffic and the data item is available immediately.

Tools must be integrated with the Communications Manager according to a well-defined and agreed-upon **integration scenario.**This specifies exactly how the tools will communicate and also lists the features supported by the integration. The definition of the integration scenario is critical because it specifies the way the tools interact and helps to determine any Communications Manager notifications that need to be created.

Applications make data available to other tools through CMan with the export process. When a tool exports data it can then be imported by other tools. Exporting data does not immediately require any action by other tools, but is often used in conjunction with a notification to alert interested tools that new data is available. Data can be exported in conjunction with a notification, or data can be exported for later use by other tools at their own convenience. This choice gives developers flexibility on how and when data is passed between tools.

The Communications Manager automatically manages and performs **data tagging** and removal. If an operation, like an object selection or simulation run, is repeated several times, there is data associated with each occurrence of that operation. To ensure that tools responding to such an event get the associated data, the Communications Manager implements a data tagging scheme that associates imported data with its corresponding notification. Previous versions of exported data are saved until any "interested" tool has had an opportunity to import data. This prevents newer data from overwriting older data, while performing the necessary "garbage collection" to keep the process size growing.

The Communications Manager Message Dictionary defines the public notifications supported for intertool communications in an electronic design environment. It lists the notifications and data objects that Cadence tools use for inter-tool communication.

The CMan supports **distributed processing** by allowing programs to be run on any node in a network. The data and notifications are not affected by the location of the sending and receiving programs. The CMan makes it possible to transparently distribute computer-intensive operations to other machines on the network. This means that simulation or verification runs can be divided up so that many computers can work in parallel.

User-defined operations can be created so that external programs can interact with Cadence tools by using SKILL and the CMan API together. Therefore Cadence tools can be customized to work with their external programs.

5.2.3 Integrating a New Tool - using the Communications Manager

The general steps to integrate a tool are:

1. **Define your objectives**
2. **Determine the integration scenario** - Identify the following:
   - notifications in the Message Dictionary to broadcast to tools that have expressed an interest
   - notifications in the Message Dictionary that are of interest to your tool
   - new notifications to define or the new data structures to create
3. **Set up the development environment** - to set up the development directory, follow these general steps:
   (a) Create a new header and library links
   (b) Compile the tool with the C compiler
   (c) Link the tool with the Communications Manager library
   (d) If the tool must highlight, link the tool with the Communications Manager library
4. **Add the Communications Manager API calls** -
   - Add calls to the starting code of the tool to initialize the Communications Manager
   - Integrate communication events into the tool's event processing
   - Write handlers for the notifications in which the tool is interested
   - Add calls to import and export data

- Handle tool exit

5. **Modify the configuration file** - typically, a configuration file is an ASCII file containing startup information about the tools. When the Communications Manager starts, it reads ASCII configuration files and converts their contents into *group* structures. You can use one or more configuration files, all have the same syntax. They can have any name, but by convention, they have either a *.cfg* or *.grp* extension. The Communications Manager merges multiple entries in the same group into a single group. The configuration file stores specific tool and variable data.

6. **Start the Communications Manager** - this can be done in one of three ways:

   (a) From the command line

   (b) In the SKILL environment (the Communications Manager starts the first time you use the SKILL interface)

   (c) A Valid Frame script from the desktop starts the Communications Manager before starting the Process Manager

   When the Communications Manager starts, it starts up the initial tool(s) specified in the configuration file. The tools can include a toolbox, such as the Process Manager, from which the other tools start. The Communications Manager quits automatically when all the tools have exited.

7. **Troubleshoot as necessary**

After completing these steps, the new tool is integrated with the same mechanism Cadence tools use to coordinate operations.

5.3 The User Interface Manager and Inter-tool Communication

The User Interface (UI) Manager provides a suite of interfaces to the Cadence DFW II architecture. It gives you use of DFW II services, window management, user interface, and graphics, without the entire DFW II executable tool. The UI manager allows you to create tools that execute in a Unix process separate from DFW II, yet appear (to the user) as if the tool was part of the DFW II package. DFW II acts like a UI Management system for the tool process. With the UI SKILL functions, your tools UI code is programmed in SKILL and executes inside DFW II.

Inter-tool communication is established by handshaking between your tool and DFW II. The UI Manager provides your tool with a handshaking (initialization) function that

retrieves your tool's port number and connects with DFW II. Inter-tool communication continues each time your tool requests services from DFW II.

Job control can be included as part of your tool with the UI Manager. DFW II sends job control information on to the tool through Cadence's SKILL IPC. For example, the UI Manager provides you with a SKILL function, *rwBeginProcess*, to request a child process ID number to start your tool within the DFW II SKILL interpreter.

5.4 Open Simulation System

The OSS provides the foundation for Cadence's simulation strategy that lets you integrate simulators into the Cadence system. Simulators integrated using OSS present a consistent user interface for controlling the execution of simulation, the generation of netlists and input vectors, and the display of the waveform output.

The simulation process can be broken down into several steps. A simulation in the Cadence system can be run with a single command. The Simulation Environment (SE) is a non-graphics program that uses the SKILL language as its interface. The only difference between SKILL in the Cadence graphics program and SKILL in the SE is the extensions that have been made to them in each program. The Cadence tools make it possible for you to interface most simulators without knowing any of the database structure.

5.4.1 Integrating a Simulator - using the Open Simulation System

To integrate a simulator into the Cadence system, you must customize the tools required by your specific application. The following is a brief description of the simulation process stages and the tools provided in OSS to simplify the process:

1. **Create an STL code generator for the simulator** - the OSCG (Open Simulation Code Generator) offers the advantage that the high-level language STL (Simulation and Test Language) can be used to create the input vectors. The STL compiler parses the STL program, so that the developer need only be concerned with programming the vector output format.

2. **Create the appropriate netlist** - you first need to decide whether a flat or hierarchical netlist needs to be created, then, to customize a netlist, each cell in the library needs to have a view of itself that guides the netlister in terms of the properties that need to be extracted and their format in the netlist.

3. **Customize SE** - since SE controls the simulation extraction, including invoking the simulator and the netlister as well as loading the data, you need to modify SE so that it recognizes the new simulator and creates the control files for the specific simulator.

4. **Create the WSF** - in order to display the simulation output as waveforms, the output must be formatted into binary Cadence WSF (Waveform Storage Format). You can use the WAL (Waveform Access Library) to directly call procedures that can generate the binary WSF file.

5. **Create an error back-annotation file** - a mechanism called probing is used throughout the design analysis process. Probing is a way of graphically highlighting nets (nodes) and instances (devices) in your design. It is also a way of communicating which nets and devices are "of interest" between design analysis tools.

5.5 IPC & SKILL

The set of SKILL functions provided by Cadence for IPC allow you to create and communicate with child processes. An advantage of running a child process is that it can run concurrently with a parent process. A child process can be a program that executes normally under the given operating system; DFW II runs non-Cadence software as a child process. A child process can also be as simple as an execution of a mail tool, i.e. any process can be a child process, and run parallel with the parent process that created it, either synchronously or asynchronously. The ability to create and run a child process greatly expands the SKILL environment.

5.6 Process Manager

The Process Manager is a tool to help you:
- Visualize and document the design process
- Automate the sequencing and startup of tools
- Encapsulate customer and commercial tools into ValidFrame (The ValidFrame environment is automatically created when you install the Valid applications and libraries)

The Process Manager (PMan) displays a graphical description of a design process. It shows each design task in the process, and the sequence of these tasks. It allows you to set up tool sequences that run automatically by the Process Manager. It also lets you *encapsulate* tools into the framework. Therefore the tools are invoked in the ValidFrame environment. This allows you to use tools from other vendors and in-house customer tools in your Process Manager process.

The Process Manager encapsulates the PIC Designer tools into a process flow and starts automatically when you start a PIC designer. It also can be used to invoke the Logic Workbench (LWB), the Analog Workbench II (AWB), the Allegro Workbench (PCB), and the Integrated Circuit Workbench (ICWB). The Process Manager allows you to execute a design process through a set of commands. There are commands to start tools or processes, to send instructions to tools, and to check the status of running processes.

Chapter 6

Specification of the Design Flow

Using Cadence Design Flow Management, create a design flow encapsulating internal and external tools by invoking both user-defined and existing SKILL functions.

6.1 Objectives

1. Create a skeleton design flow

- create a flowchart and its design steps by incorporating the existing SKILL functions that are available
- add and link the design steps accordingly to the flowchart
- display the flowchart in a Flow Browser window

N.B. The advantage of creating the skeleton flow is that aspects, such as the direction of flow and design set dependencies, can be verified before proceeding with the details of the flow.

2. Add functions to the design steps

- each design step will describe a specific action or set of actions
- some of the actions to be taken will invoke non-Cadence tools, therefore the function must be encapsulated in SKILL
- the integration of non-Cadence tools must be supported in the source code using SKILL

3. Incorporate some additional features of the design flow

- to fulfill the requirements of a hierarchical flowchart, create a subflowchart, its design steps and the necessary functions
- add extra items to the pop-up menus and window/form banners, where necessary

4. Use forms to gather the information from the user

- use the SKILL functions available to create and display various types of forms to gather the information necessary to invoke the required events from the tools to be used

5. Integrate both Cadence and non-Cadence tools

- write the necessary SKILL source code to integrate a variety of tools
- incorporate SKILL functions that are already available with you own user-defined functions

6. Provide an online help facility and user manual

- use the html editor, "tkHTML 2.3", to create help pages for the individual design steps, where necessary
- create a user manual for the system

6.2 The proposed system

The proposed system (outlined in figure 6.1) is a design flow which guides the user through a series of steps which will enable him/her to generate XNF netlists from a C++ program, convert them to Verilog netlists, view the Verilog netlists and the schematic, and then call the Xilinx Design Manager to further implement the logical design.

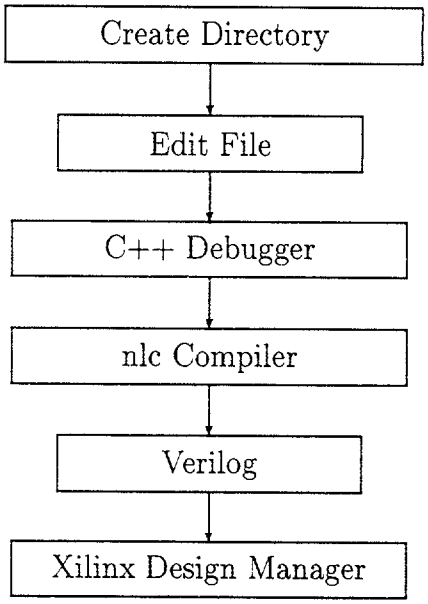


Figure 6.1: Flowchart of the proposed system

The design flow will be presented to the user in a Flow Browser window. Within this window, Design Flow provides certain menus and ready-made commands (see page 13). To increase the user-friendliness of the system, it is possible to add your own facilities and commands. An online user manual and a specific help facility for individual steps, would benefit the aesthetics of the system.

6.3 Specification of each design step

1. Create Directory

- ask the user to input the name of the new directory to use for saving files to, etc
- create this new directory in the user's account

2. Edit File

- ask the user to input the name of a file, either an existing C++ program to be edited, or a new one to be created
- open the file via a text editor

3. C++ Debugger

- call a debugging tool so the user has the opportunity to debug the C++ program

4. nlc Compiler

- ask the user to select the various compiler options he/she requires
- call the nlc compiler with the selected options and create the necessary files
- view the compiler output, symbol file and WIR or XNF netlist file that are created

5. Verilog

- convert the XNF netlists to Verilog netlists via calls to two procedures that will generate, first EDIF netlists from the XNF netlists, and then the EDIF netlists to Verilog netlists
- view the Verilog netlist file via a graphics window
- call "Verilog In" to enable the user to view the schematic in the Cadence graphical editor. Verilog In imports a design in Verilog Hardware Description Language (HDL) format into a DFW II database format library.

6. Xilinx Design Manager

- start the Xilinx Design Manager so the user can access the various Xilinx tools to implement the design

Chapter 7

Design and Implementation

From the outlined design flow in figure 6.1, the flowchart will contain six design steps. Each step will encapsulate a tool, with the "Verilog" step incorporating two further steps which will be dealt with via a hierarchical flowchart. All the examples of code are extracts from the source code file "desproj.il".

7.1 Encapsulating the Tools

Each step must first be encapsulated using the SKILL function *dfEncapsulate*.

```
createStep  = dfEncapsulate(?name                "Create Directory"
                            ?function            'funcCreate
              )
```

The process of encapsulating a tool is primarily that of creating a design step and assigning appropriate properties that fully describe it. The tool is defined in terms that the Design Flow system can use to control and invoke the tool. The Design Flow system requires that the core functions of the tool, be defined in SKILL. For foreign tools, this might mean creating a wrapper function in SKILL whose only operation is a system call that initiates an external process. More deeply encapsulated foreign tools might include SKILL code to interpret the information or exit codes returned by the tool. After defining the core functions in SKILL, the step can be created and customized.

7.2 Creating and Displaying the Flowchart

Create the Flowchart After the encapsulation of the steps, the main flowchart can be created. First, it is important to initialize a library in which the design flow will be stored by using *dfInitLib*, and then create the flowchart, using *dfCreateFlowchart*.

```
dfInitLib(?libName "projlib")
mainFlowchart = dfCreateFlowchart(?name        "mainFlowchart"
                                  ?libName     "projlib"
                                  ?lowerLeft   '(150 150)
                )
```

Add and Link Design Steps to the Flowchart To add the design steps to the flowchart, it is necessary to use *dfAddStepToFlowchart* function. This, however does not create any parent-child relationships between the design steps, so the steps must be linked accordingly using *dfLinkStepsToFlowchart*. This established the parent-child relationship between two steps on the flowchart.

```
dfAddStepToFlowchart(?flowchart   mainFlowchart
                     ?step        createStep
                     ?xy          '(0 3500)
)

dfLinkFlowchartSteps(?flowchart   mainFlowchart
                     ?parent      createStep
                     ?child       editStep
)
```

Create a Subflowchart for the "Verilog" Design Step In the case of the "Verilog" design step, the operation of a single step is too complicated. It makes sense to create a flowchart which represents the function of this design step. Therefore, a subflowchart is created and attached to the step. The subflowchart can be set in the encapsulation of the design step and then its steps are added and linked as for the main flowchart.

```
verilogStep = dfEncapsulate(?name                "Verilog"
                            ?subflowchartLibName "projlib"
                            ?subflowchartName    "verilogSubflowchart"
                            ?function             nil
              )
```

The subflowchart (see figure 7.2) is indistinguishable from an ordinary flowchart, except that it includes a pointer to the design step which it expands. This is in the form of a downward-pointing arrow inside the design step, which is automatically created to indicate that a sub-flow exists for the step (see "Verilog" Step on the flowchart in figure 7.1), and also a further item is created, called "Push to Subflowchart", in the pop-up menu of the design step.

Add Pop-Up Menu Items The pop-up menu on each of the design steps has two commands as its default, namely *Run Step* and *Reset Step*. Additional items can be added to this menu. An extra item, *Bypass Step*, has been added to give the user the opportunity of "skipping" the step, if it is not needed. This gives the user a certain control over which steps he/she can run.

```
bypassCreateAction = list("Bypass Step" "bypassCreateStep" "")

createStepNode = dfGetStepNode(?step       createStep
                               ?flowchart  mainFlowchart
                 )

dagAddActionToObject(bypassCreateAction createStepNode)
```

```
procedure(bypassCreateStep()
  prog((stepInst)
    stepInst =  dfGetStepInst(?step          createStep
                              ?flowchartInst mainFlowchartInst
                )
    dfRunPostFunc(?stepInst    stepInst
    )
  ) ; ** prog  **
) ; ** procedure - bypassCreateStep **
```

Set Dependencies for the Design Steps However dependencies are set, using *dfSetDependency*, so the Design Flow system can validate each step, by ensuring that the previous step has finished correctly, before allowing the next step to be run.

```
dfSetDependency(?step        editStep
                ?flowchart   mainFlowchart
                ?dependency  'allParents
)
```

Display the Flowchart Finally, the flowchart can be displayed in a Flow Browser window using the command *dfDisplayFlowchart* (see figure 7.1).

```
mainFlowchartInst = dfDisplayFlowchart(?libName     "projlib"
                                       ?cellName    "topLevelFlow"
                                       ?flowchart   mainFlowchart
                    )
```

Each step in the design will appear with a colour-coded bar. The bar is green to begin with, and changes to grey while the step is running, and then to black when the step has finished. The steps also have two smaller rectangles at the top, called *Run* and *Props*. The *Run* box can be used to invoke the step instead of the using the *Run Step* command on the pop-up menu, and the *Props* box brings up a separate window displaying the information that has been stored on the design step to indicate the status of the step, such as *running, finished, lastRunTime*, etc.

7.3 The Functions of the Design Steps

Each design step has a function which is called when the step is activated. Within the functions, forms are used to gather information, dialog boxes are used to display messages and warnings to aid the user, and the various tools are integrated and called to produce the desired output.

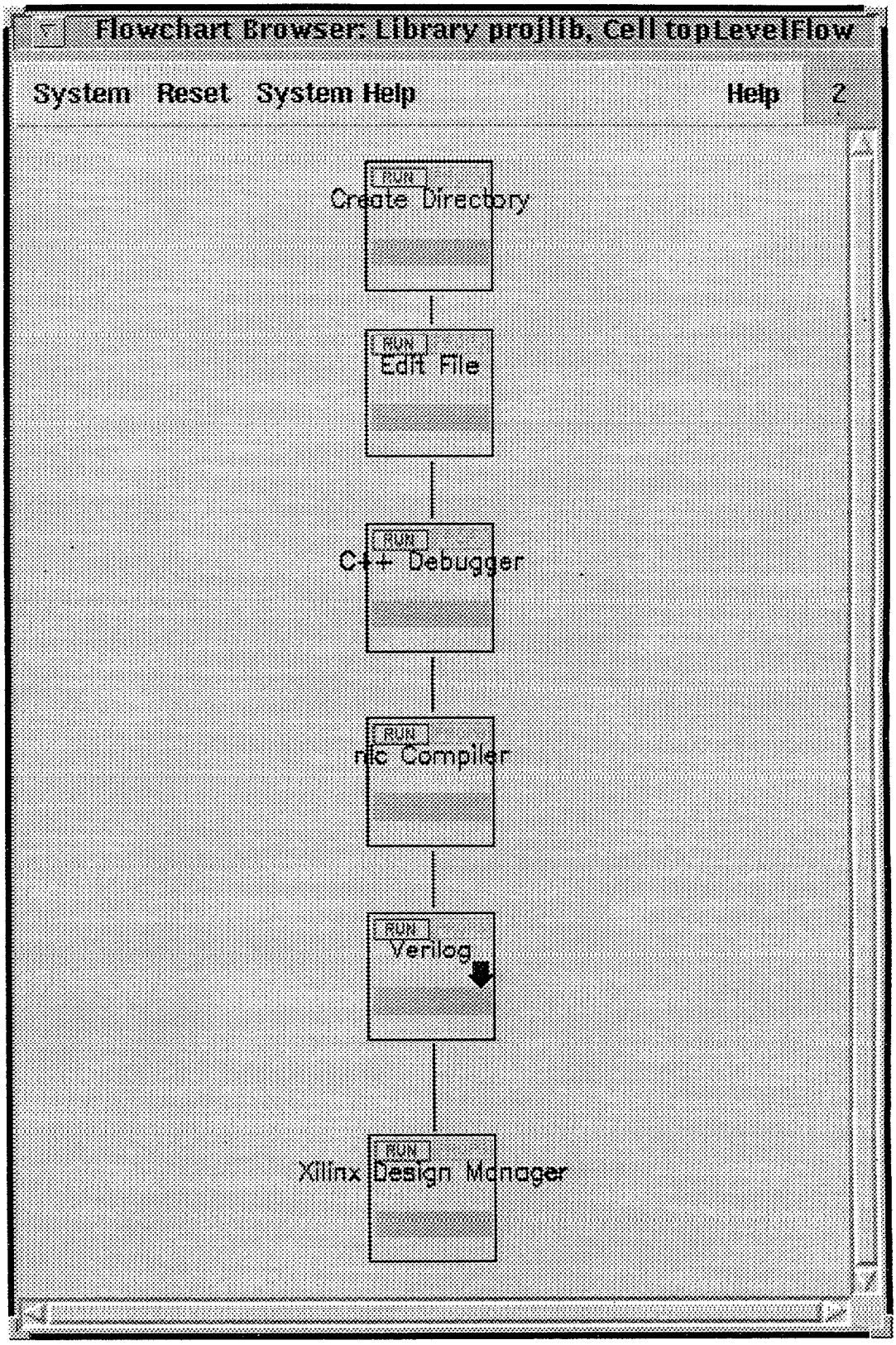


Figure 7.1: The NAC flowchart in a Flow Browser window

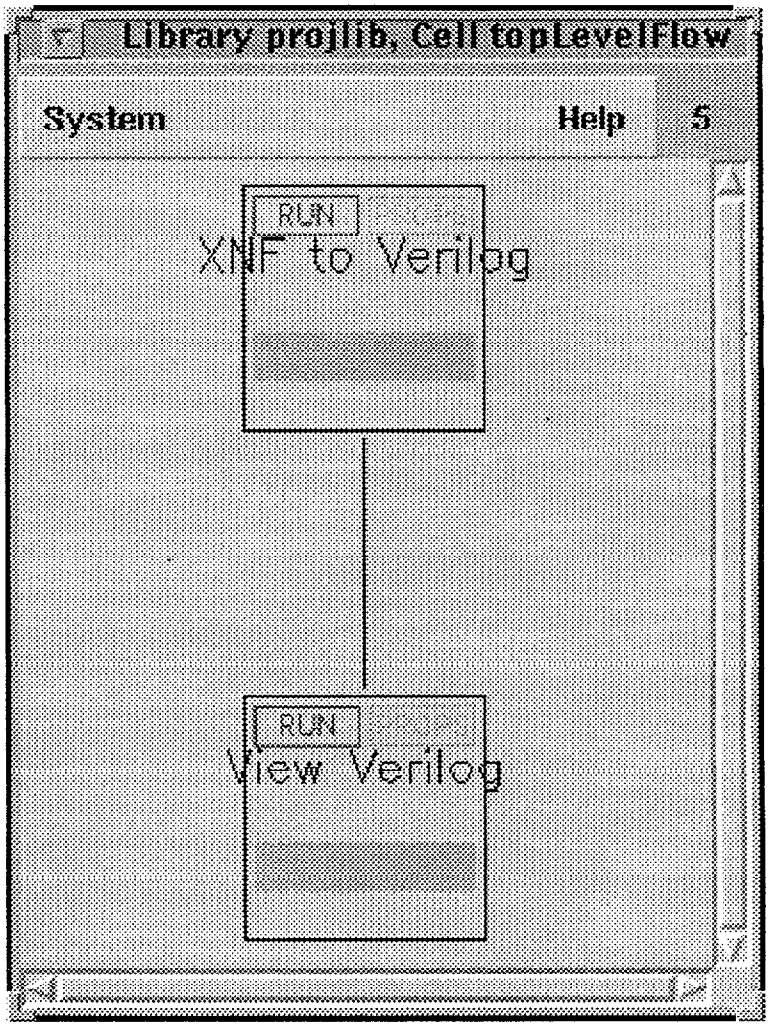


Figure 7.2: The Verilog subflowchart in a Flow Browser window

7.3.1 Step 1 Create Directory - *funcCreate*

funcCreate first displays a dialog box to inquire of the user if he/she wishes to create a new directory (see figure 7.3). By clicking on *Yes*, it calls another function which displays a form for the user to input the name of the directory to be created (see figure 7.4). When the user clicks on *OK*, this invokes a procedure which creates the new directory in the users account.

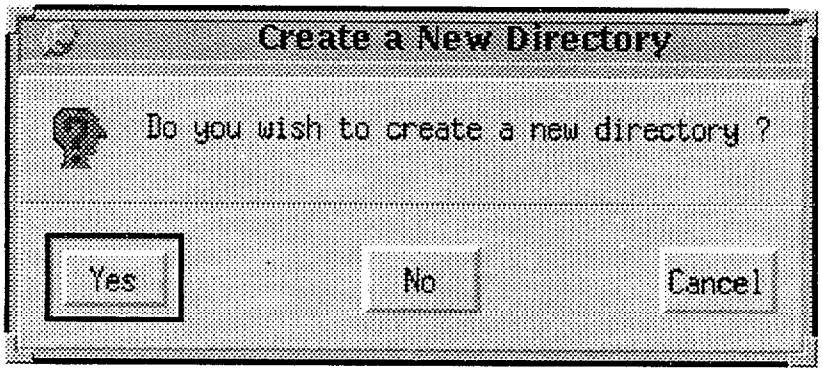


Figure 7.3: Dialog Box - "Create a New Directory"

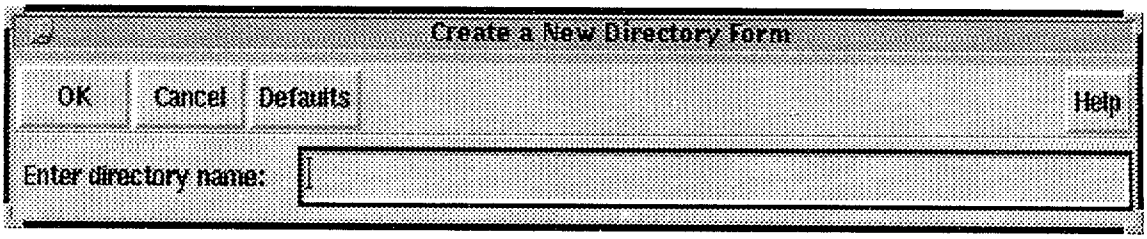


Figure 7.4: Form - "Create a New Directory - File Name Form"

• The procedure below, which is called form *funcCreate*, creates the form to gather the new directory's name.

```
procedure(gotoCreateDirectory()
  strfield = hiCreateStringField(?name     'dirName
                                 ?prompt   "Enter directory name:"
             )

  hiCreateAppForm(?name          'createForm
                  ?formTitle     "Create a New Directory Form"
                  ?fields        '(strfield)
                  ?callback      '(funcOK)
                  ?buttonLayout  'OKCancelDef
  ) ; ** hiCreateAppForm **
```

• This function is called from the *Create a New Directory Form* to create the new directory.

```
procedure(funcOK(createForm "r")
  createDir(createForm->dirName->value)
) ; ** procedure - funcOK **
```

• This command displays the *Create a New Directory Form.*

```
  hiDisplayForm(createForm)
) ; ** procedure - gotoCreateDirectory **
```

• The dialog box is created here and calls the function *gotoCreateDirectory* if the user clicks on *OK*, otherwise the function finishes.

```
procedure(funcCreate( @key (stepInst nil) )
  hiDisplayAppDBox(?name        'createDirDBox
                   ?dboxBanner  "Create a New Directory"
                   ?dboxText    "Do you wish to create a new
                                 directory ?"
                   ?callback    "gotoCreateDirectory()"
                   ?dialogType  hicQuestionDialog
                   ?buttonLayout 'YesNoCancel
  ) ; ** hiDisplayAppDBox **
) ; ** procedure - funcCreate **
```

7.3.2 Step 2 Edit File - *funcEdit*

funcEdit first creates a dialog box to inquire of the user if he/she wishes to edit or create a C++ file (see figure 7.5). By clicking on *Yes*, it calls another function which displays a form for the user to input the name of the file to be edited or created (see figure 7.6). When the user clicks on *OK*, this invokes a procedure which calls the text editor ready for editing or creating a file.

• This procedure, which is called from *funcEdit*, sets the test editor in Cadence to "emacs" and then the SKILL command *hiEditFile()* brings up a form for the user to input the file name. When the user clicks on *OK*, the file is presented in a text editor window.

```
procedure(gotoTextEditor()
  editor = "$EDITOR"
  hiEditfile()
) ; ** gotoTextEditor **
```

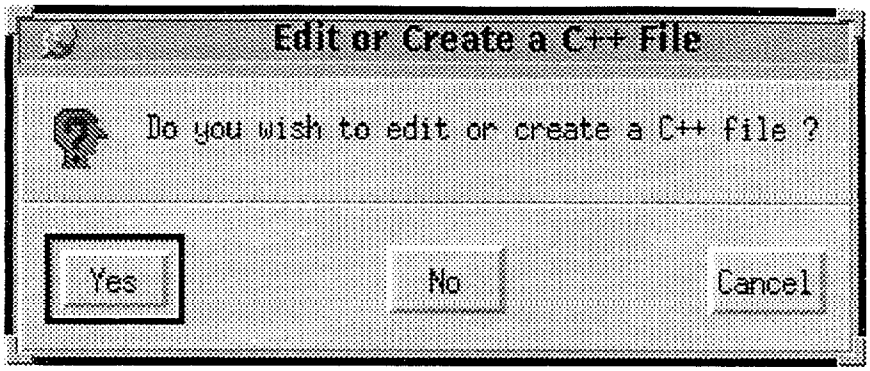


Figure 7.5: Dialog Box - "Edit or Create a C++ File"

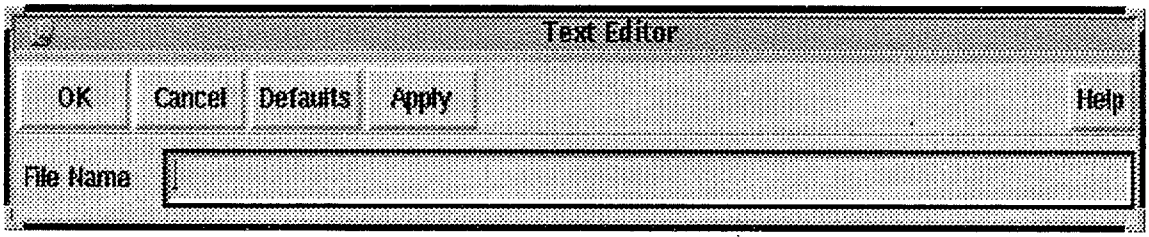


Figure 7.6: Form - "Text Editor"

• The dialog box is created here and calls the function *gotoTextEditor* if the user clicks on *OK*, otherwise the function finishes.

```
procedure(funcEdit( @key (stepInst nil) )
  hiDisplayAppDBox(?name         'editFileDBox
                   ?dboxBanner   "Edit or Create a C++ File"
                   ?dboxText     "Do you wish to edit or create a
                                  C++ file ?"
                   ?callback     "gotoTextEditor()"
                   ?dialogType   hicQuestionDialog
                   ?buttonLayout 'YesNoCancel
  ) ; ** hiDisplayAppDBox **
) ; ** procedure - funcEdit **
```

7.3.3 Step 3 C++ Debugger - *funcDebug*

funcDebug first creates a dialog box to inquire of the user if he/she wishes to debug a C++ file (see figure 7.7). By clicking on *Yes*, it calls another function which displays a form for the user to input the name of the file to be edited or created (see figure 7.8). When the user clicks on *OK*, a message dialog box appears to check that the file name has been entered in the correct format (see figure 7.9). This message box returns you to

the form and you can either change the format, if necessary, or else click on *OK* and the procedure to call the debugger is invoked.

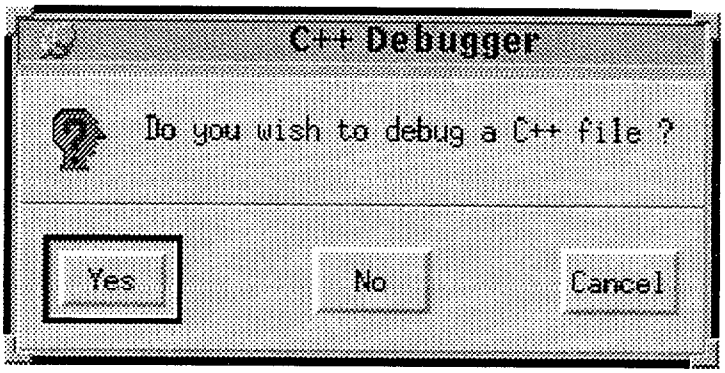


Figure 7.7: Dialog Box - "C++ Debugger"

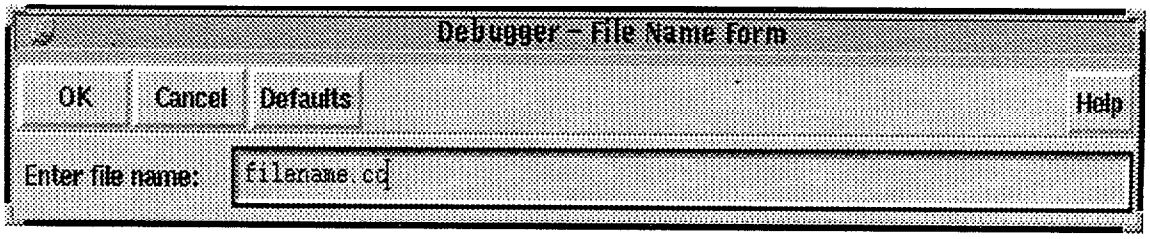


Figure 7.8: Form - "C++ Debugger - File Name Form"

• This function, which is called from *gotoDebugForm*, calls the debugger with file name inputted by the user in the *Debugger - File Name Form*. To invoke the debugger with the necessary information (ie the file name), the file name is passed to a previously created "C Shell Script", which in turn makes the call to the tool.

```
procedure(funcCallToDebugger(option)
   setq(debugFileName debuggerForm -> filename -> value)
   p = outfile("callDebugger")
   fprintf(p "#! /bin/csh -f \n")
   fprintf(p "# Call C++ Debugger\n\n")
   fprintf(p "set nameOfFile = %s \n\n" debugFileName)
   fprintf(p "($DEBUGGER $nameOfFile &) \n\n")
   fprintf(p "exit")
   close( p )
   system("callDebugger")
) ; ** procedure - funcCallToDebugger  **
```

• This dialog box checks with the user that the file name is in the correct format.

Figure 7.9: Dialog Box - "File Name Check"

```
procedure(fileNameMessage()
  hiDisplayAppDBox(?name          'fileNameQtnBox
                   ?dboxBanner    "File Name Check"
                   ?dboxText      "Is the file name specified in the
                                   style of the format given ?"
                   ?dialogType    hicQuestionDialog
                   ?buttonLayout  'YesNoCancel
  ) ; ** hiDisplayAppDBox **
) ; ** procedure - fileNameMessage **
```

• The procedure below creates the form to get the name of the C++ file to be debugged.

```
procedure(gotoDebugForm()
  strfield = hiCreateStringField(?name      'filename
                                 ?prompt    "Enter file name:"
                                 ?defValue  "filename.cc"
                                 ?callback  "fileNameMessage()"
             ) ; ** string **

  hiCreateAppForm(?name          'debuggerForm
                  ?formTitle     "Debugger - File Name Form"
                  ?fields        '(strfield)
                  ?callback      '(funcCallToDebugger)
                  ?buttonLayout  'OKCancelDef
                  ?initialSize   t
  ) ; ** hiCreateAppForm **

  hiDisplayForm(debuggerForm)
) ; ** procedure - gotoDebugForm **
```

• The dialog box is created here and calls the function *gotoDebugForm* if the user clicks on *OK*, otherwise the function finishes.

```
procedure(funcDebug( @key (stepInst nil) )
  hiDisplayAppDBox(?name          'debugDBox
                   ?dboxBanner    "C++ Debugger"
                   ?dboxText      "Do you wish to debug a C++ file ?"
                   ?callback      "gotoDebugForm()"
                   ?dialogType    hicQuestionDialog
                   ?buttonLayout 'YesNoCancel
    ) ; ** hiDisplayAppDBox **
) ; ** procedure - funcDebug **
```

7.3.4 Step 4 nlc Compiler - *funcCompile*

funcCompile is significantly longer than any of the other design step functions as it contains then nlc Compiler options. From the main menu the user makes various choices of what he/she requires for the compilation. This menu calls several forms, each containing more compiler options. Once all the necessary information is gathered (i.e. filenames and options) the nlc Compiler is called via a "C Shell script" which has all the information passed to it.

• This function is called from the *nlc Compiler Output Options Form* when the user clicks on the *nlc Help* button. It displays a message box asking the user if they require help (see figure 7.10), if they do, *funcHelp* is called which invokes a system call to the html browser with the relevant "Help pages" (see the Appendix).

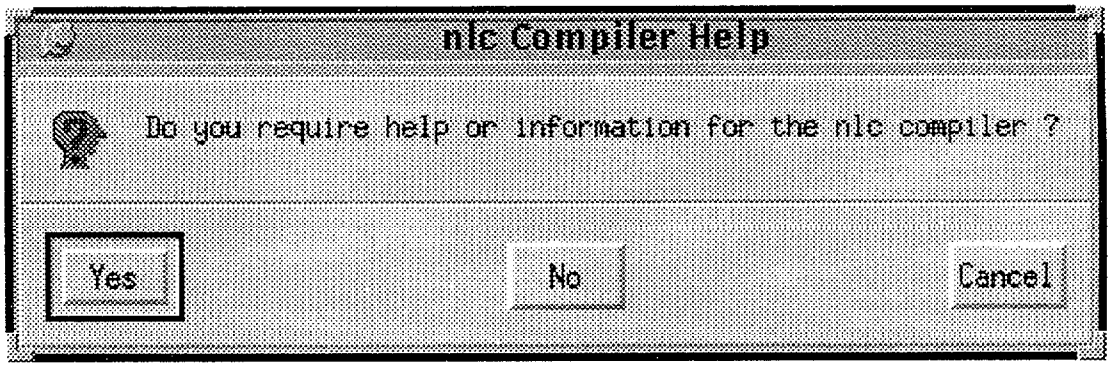


Figure 7.10: Dialog Box - "nlc Compiler Help"

```
procedure(gotoNlcHelpOption()

  procedure(funcHelp()
    system("$HTMLBROWSER html/nlcHelp.html &")
  ) ; ** procedure - funcHelp **
```

```
  hiDisplayAppDBox(?name          'helpDBox
                   ?dboxBanner    "nlc Compiler Help"
                   ?dboxText      "Do you require help or information
                                   for the nlc compiler ?"
                   ?callback      "funcHelp()"
                   ?dialogType    hicQuestionDialog
                   ?buttonLayout 'YesNoCancel
  ) ; ** hiDisplayAppDBox **
) ; ** procedure - gotoNlcHelpOption  **
```

• This is the body of procedure *funcInternalInfoOptions* where the various choices that the user makes in the *Internal Information Options Form* are evaluated and available for use in the *systemCallToCompiler* function.

```
procedure(funcInternalInfoOptions( @optional choice1 choice2 choice3
                                              choice4 choice5 choice6 )

  setq(choice1 internalInfoForm -> internalInfoOptions -> toggle1 ->
       value)
  setq(choice2 internalInfoForm -> internalInfoOptions -> toggle2 ->
       value)
  setq(choice3 internalInfoForm -> internalInfoOptions -> toggle3 ->
       value)
  setq(choice4 internalInfoForm -> internalInfoOptions -> toggle4 ->
       value)
  setq(choice5 internalInfoForm -> internalInfoOptions -> toggle5 ->
       value)
  setq(choice6 internalInfoForm -> internalInfoOptions -> toggle6 ->
       value)

  cond(
       (choice1 == t && choice2 != t && choice3 != t && choice4 != t
        && choice5 != t && choice6 != t opt4 = "-d{t}")
       (choice1 != t && choice2 == t && choice3 != t && choice4 != t
        && choice5 != t && choice6 != t opt4 = "-d{s}")
       (choice1 != t && choice2 != t && choice3 == t && choice4 != t
        && choice5 != t && choice6 != t opt4 = "-d{o}")
       (choice1 != t && choice2 != t && choice3 != t && choice4 == t
        && choice5 != t && choice6 != t opt4 = "-d{c}")
       (choice1 != t && choice2 != t && choice3 != t && choice4 != t
        && choice5 == t && choice6 != t opt4 = "-d{m}")
       (choice1 != t && choice2 != t && choice3 != t && choice4 != t
        && choice5 != t && choice6 == t opt4 = "-d{z}")
       (choice1 == t && choice2 == t && choice3 != t && choice4 != t
        && choice5 != t && choice6 != t opt4 = "-d{t}{s}")
```

```
        .
        .
        .
        .
        .
       (choice1 == t && choice2 == t && choice3 == t && choice4 == t
        && choice5 == t && choice6 == t opt4 = "-d{a}")
      ) ; ** cond **
) ; ** procedure - funcInternalInfoOptions **
```

• This is the body of procedure *gotoInternalInfoOptions* which creates and displays the *Internal Information Options Form* (see figure 7.11).

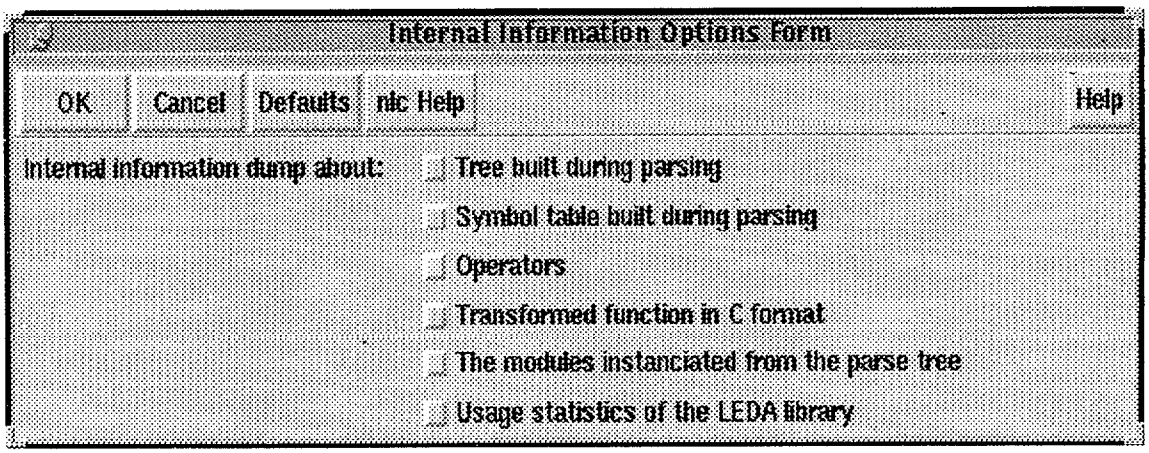


Figure 7.11: Form - "Internal Information Options Form

```
procedure(gotoInternalInfoOptions()
  opt2Toggle = hiCreateToggleField(?name        'internalInfoOptions
                                   ?choices     list(
                                   '(toggle1 "Tree built during parsing")
                                   '(toggle2 "Symbol table built during
                                     parsing")
                                   '(toggle3 "Operators")
                                   '(toggle4 "Transformed function in C
                                     format")
                                   '(toggle5 "The modules instanciated
                                     from the parse tree")
                                   '(toggle6 "Usage statistics of the
                                      LEDA library")
                                             ) ; ** list **
                                    ?numSelect   6
                                    ?prompt      "Internal information
```

```
                                                    dump about:"
                                    ?itemsPerRow 1
                                    ?value '(nil nil nil nil nil nil)
                ) ; ** opt2Toggle **

  hiCreateAppForm(?name     'internalInfoForm
                  ?formTitle    "Internal Information Options Form"
                  ?fields       list( opt2Toggle )
                  ?callback     '(funcInternalInfoOptions)
                  ?buttonLayout '(OKCancelDef
                                  (nlc\ Help "gotoNlcHelpOption()")
                                 )
                  ?initialSize  t
  ) ; ** hiCreateAppForm **

  hiDisplayForm(internalInfoForm)
) ; ** procedure - gotoInternalInfoOptions **
```

• This is the body of procedure *gotoWirOption* which creates and displays the *WIR Netlists Directory Option Form* (see figure 7.12).

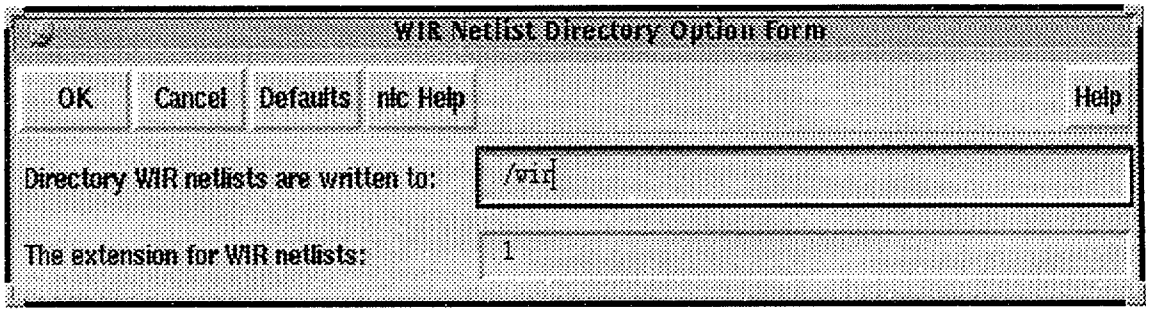


Figure 7.12: For - "WIR Netlist Directory Option Form"

```
procedure(gotoWirOption()
  wirString1 = hiCreateStringField(?name        'wirDir
                                   ?prompt      "Directory WIR
                                    netlists are written to: "
                                   ?defValue    "./wir"
               ) ; ** wirString1 **

  wirString2 = hiCreateStringField(?name        'wirExt
                                   ?prompt      "The extension for
                                    WIR netlists:"
                                   ?defValue    ".1"
```

```
            ) ; ** wirString2 **

   hiCreateAppForm(?name          'wirForm
                   ?fields        list(wirString1 wirString2)
                   ?formTitle     "WIR Netlist Directory Option Form"
                   ?buttonLayout  '(OKCancelDef
                                    (nlc\ Help "gotoNlcHelpOption()")
                                   )
                   ?initialSize   t
   ) ; hiCreateAppForm **

   hiDisplayForm(wirForm)
) ; ** procedure - gotoWirOption **
```

• This is the body of procedure *gotoXnfOption* which creates and displays the *XNF Netlists Directory Option Form* (see figure 7.13).

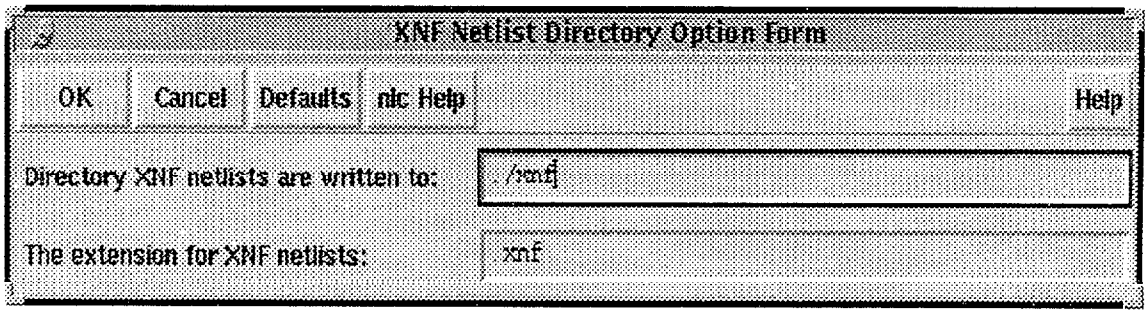


Figure 7.13: Form - "XNF Netlist Directory Option Form"

```
procedure(gotoXnfOption()
  xnfString1 = hiCreateStringField(?name        'xnfDir
                                   ?prompt      "Directory XNF
                                    netlists are written to: "
                                   ?defValue    "./xnf"
               ) ; ** wirString1 **

  xnfString2 = hiCreateStringField(?name        'xnfExt
                                   ?prompt      "The extension for
                                    XNF netlists:"
                                   ?defValue    ".xnf"
               ) ; ** wirString2 **

  hiCreateAppForm(?name          'xnfForm
                  ?fields        list(xnfString1 xnfString2)
```

```
                ?formTitle      "XNF Netlist Directory Option Form"
                ?buttonLayout   '(OKCancelDef
                                  (nlc\ Help "gotoNlcHelpOption()")
                                 )
                ?initialSize    t
  ) ; hiCreateAppForm **

  hiDisplayForm(xnfForm)
); procedure - gotoXnfOption **
```

• This is the body of procedure *funcNetlistAndSymsOptions* where the various choices that the user makes in the *Netlists & Symbols Options Form* are evaluated and available for use in the *systemCallToCompiler* function.

```
procedure(funcNetlistAndSymsOptions( @optional choice1 choice2 choice3
                                     choice4 choice5 choice6)
  setq(choice1 netlistAndSymsForm -> netlistAndSymsOpt1 -> toggle1 ->
       value)
  setq(choice2 netlistAndSymsForm -> netlistAndSymsOpt1 -> toggle2 ->
       value)
  setq(choice3 netlistAndSymsForm -> netDir -> value)
  setq(choice4 netlistAndSymsForm -> symDir -> value)
  setq(choice5 netlistAndSymsForm -> symExt -> value)
  setq(choice6 netlistAndSymsForm -> bitvectorSize -> value)

  if( choice3 == "wir" then
     gotoWirOption()
  else
     gotoXnfOption()
  ) ; ** if **

  cond(
       (choice1 == t && choice2 != t opt5a = " -k ")
       (choice1 != t && choice2 == t opt5a = " -O ")
       (choice1 == t && choice2 == t opt5a = " -k  -O ")
       (choice1 != t && choice2 != t opt5a = " ")
      ) ; ** cond **

  if( choice3 == "wir" then
     setq(wirOpt1 wirForm -> wirDir -> value)
     if( wirOpt1 !=  "" then
        setq(opt5b strcat(" -N " wirOpt1))
     else
        setq(opt5b " ")
     ) ; ** if **
```

```
  else
     setq(xnfOpt1 xnfForm -> xnfDir -> value)
     if( xnfOpt1 !=  "" then
        setq(opt5b strcat(" -A -X " xnfOpt1))
     else
        setq(opt5b " ")
     ) ; ** if **
  ) ; ** if **

  if( choice4 != "" then
     setq(opt5c strcat(" -S " choice4))
  else
     setq(opt5c " ")
  ) ; ** if **

  if( choice3 == "wir" then
     setq(wirOpt2 wirForm -> wirExt -> value)
     if( wirOpt2 != "" then
        setq(opt5d strcat(" -n " wirOpt2))
     else
       setq(opt5d " ")
           ) ; ** if **
  else
     setq(xnfOpt2 xnfForm -> xnfExt -> value)
     if( xnfOpt2 !=  "" then
        setq(opt5d strcat(" -x " xnfOpt2))
     else
        setq(opt5d " ")
     ) ; ** if **
  ) ; ** if **

  if( choice5 != "" then
     setq(opt5e strcat(" -s " choice5))
  else
     setq(opt5e " ")
  ) ; ** if **

  if( choice6 != "" then
     setq(opt5f strcat(" -w " choice6))
  else
     setq(opt5f " ")
  ) ; ** if **

  setq(opt5 strcat(opt5a opt5b opt5c opt5d opt5e opt5f)
) ; ** procedure - funcNetlistAndSymsOptions **
```

• This is the body of the *gotoNetlistAndSymsOptions* procedure which creates and displays the *Netlists & Symbols Options Form* where the user selects certain compile options (see figure 7.14).

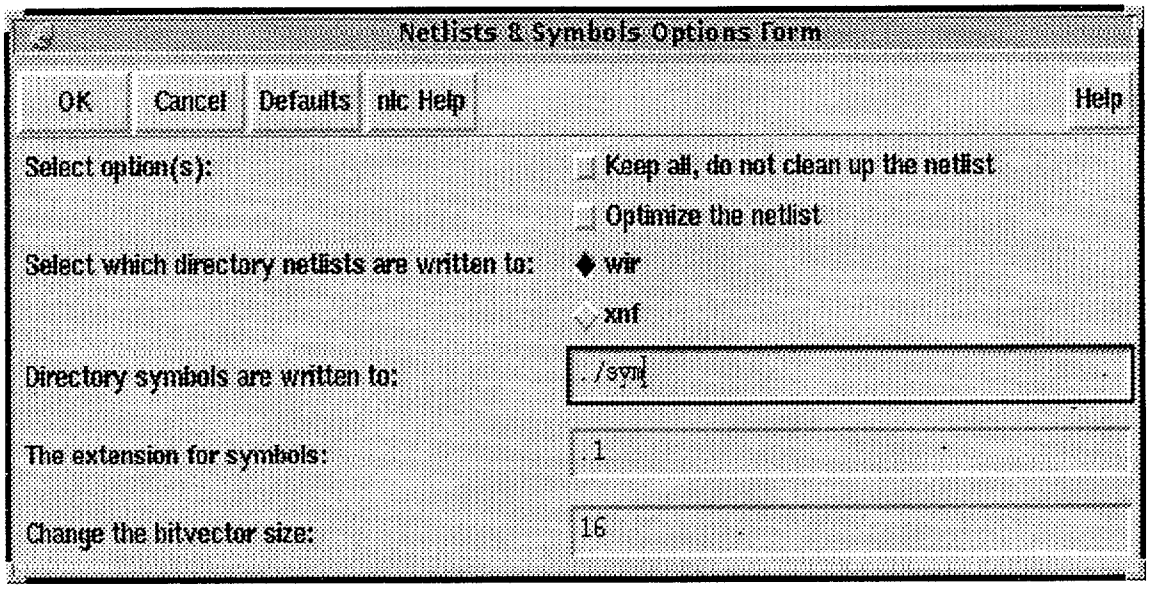


Figure 7.14: Form - "Netlists & Symbols Options Form"

```
procedure(gotoNetlistAndSymsOptions()
  opt3Toggle = hiCreateToggleField(?name        'netlistAndSymsUpt1
                                   ?choices     list(
                                   '(toggle1 "Keep all, do not clean up
                                     the netlist")
                                   '(toggle2 "Optimize the netlist")
                                                ) ; ** list **
                                   ?numSelect   2
                                   ?prompt      "Select option(s):"
                                   ?itemsPerRow 1
                                   ?value       '(nil nil)
               ) ; ** opt3Toggle **

  opt3Radio = hiCreateRadioField(?name        'netDir
                                 ?choices     '("wir" "xnf")
                                 ?prompt      "Select which directory
                                  netlists are written to:"
                                 ?itemsPerRow 1
              ) ; ** opt3Radio **

  opt3String1 = hiCreateStringField(?name        'symDir
```

```
                                ?prompt      "Directory symbols are
                                written to:"
                                ?defValue    "./sym"
              ) ; ** opt3String1 **

  opt3String2 = hiCreateStringField(?name        'symExt
                                ?prompt      "The extension for symbols:"
                                ?defValue    ".1"
              ) ; ** opt3String3 **

  opt3String3 = hiCreateStringField(?name        'bitvectorSize
                                ?prompt      "Change the bitvector size:"
                                ?defValue    "16"
              ) ; ** opt3String4 **

  hiCreateAppForm(?name         'netlistAndSymsForm
                  ?formTitle    "Netlists & Symbols Options Form"
                  ?fields       list( opt3Toggle opt3Radio opt3String1
                                      opt3String2 opt3String3 )
                  ?callback     '(funcNetlistAndSymsOptions)
                  ?buttonLayout '(OKCancelDef
                                  (nlc\ Help "gotoNlcHelpOption())")
                                 )
                 ?initialSize   t
  ) ; ** hiCreateAppForm **

  hiDisplayForm(netlistAndSymsForm)
) ; ** procedure - gotoNetlistAndSymsOptions **
```

• The procedure displays a dialog box informing the user that the "version information" option they choose will be written to the *nlc.log" file* (see figure 7.15), which can be viewed from the output form.

```
procedure(gotoPrintVersionInfoOption()
  hiDisplayAppDBox(?name         'printVersionDBox
                   ?dboxBanner   "Version Information Message"
                   ?dboxText     "Version information is now being
                                  written to the nlc.log file"
                   ?dialogType   hicMessageDialog
                   ?buttonLayout 'Close
  ) ; ** hiDisplayAppDBox **
) ; ** procedure - gotoPrintVersionInfoOption **
```

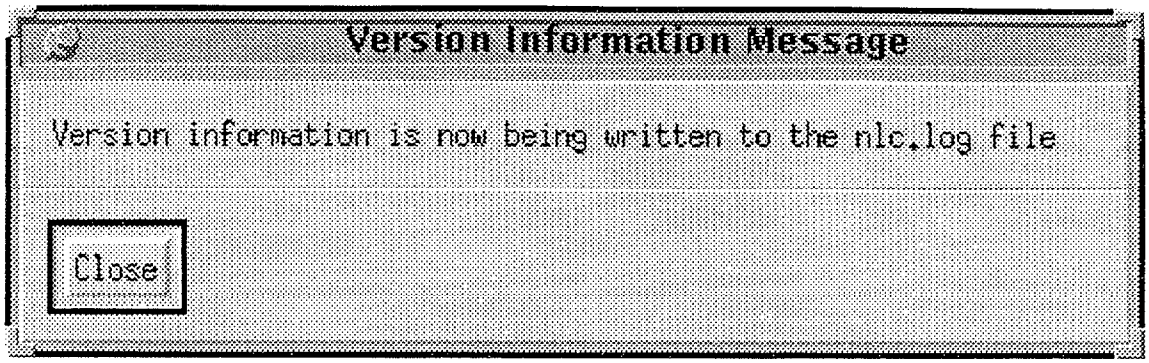


Figure 7.15: Dialog Box - "Version Information Message"

• the procedures *gotoDisplayWarningMessage1* and *gotoDisplayWarningMessage2* display message boxes to indicate to the user that the output they wish to see cannot be viewed as they did not choose that option in the previous forms (see figures 7.16 and 7.17).

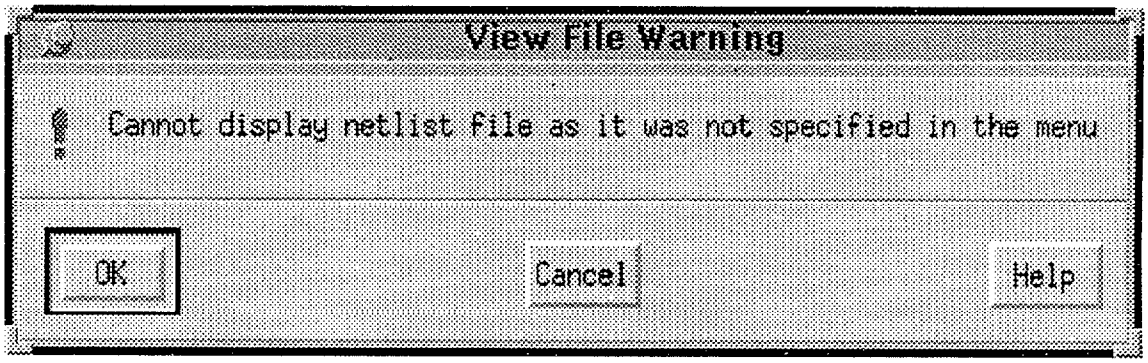


Figure 7.16: Dialog Box - "View File Warning - Netlists"

```
procedure(gotoDisplayWarningMessage1()
  hiDisplayAppDBox(?name          'errorMessageBox
                   ?dboxBanner    "View File Warning"
                   ?dboxText      "Cannot display netlist file as it
                                   was not specified in the menu"
                   ?callback      "gotoNlcOutputForm()"
                   ?dialogType    hicWarningDialog
                   ?buttonLayout 'OKCancel
  ) ; ** hiDisplayAppDBox **
) : ** procedure - gotoDisplayWarningMessage1 **

procedure(gotoDisplayWarningMessage2()
  hiDisplayAppDBox(?name          'errorMessageBox
                   ?dboxBanner    "View File Warning"
                   ?dboxText      "Cannot display symbol file as it was
```

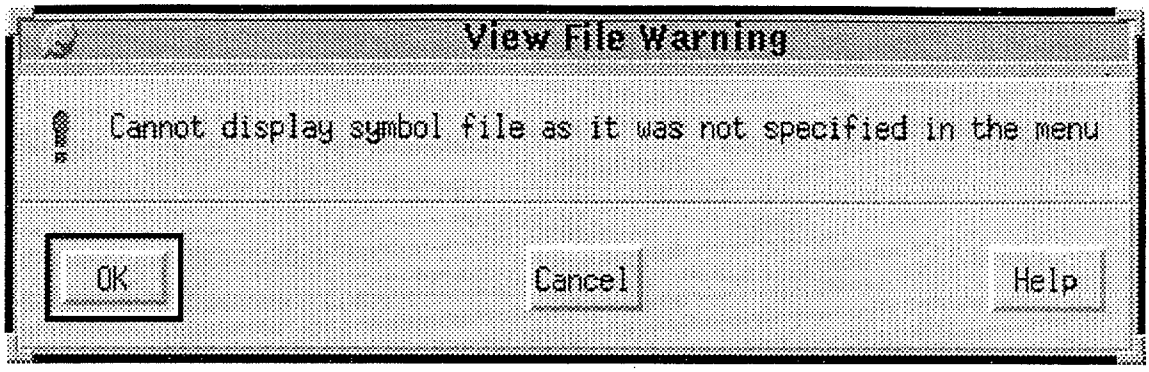


Figure 7.17: Dialog Box - "View File Warning - Symbols"

```
                                not specified in the menu"
                    ?callback       "gotoNlcOutputForm()"
                    ?dialogType     hicWarningDialog
                    ?buttonLayout   'OKCancel
  ) ; ** hiDisplayAppDBox **
) ; ** procedure - gotoDisplayWarningMessage2 **
```

• This part of the code of *funcNlcOutput* passes the necessary information to a"C Shell Script" to view the netlist file output (WIR or XNF) either, in a view window, or via the standard output (xterm window), depending on what options were chosen earlier.

```
procedure(funcNlcOutput(@optional outopt1 outopt2 outopt3)
  setq(outopt1 nlcOutputForm -> viewNlcOutput -> toggle1 -> value)
  setq(outopt2 nlcOutputForm -> viewNlcOutput -> toggle2 -> value)
  setq(outopt3 nlcOutputForm -> viewNlcOutput -> toggle3 -> value)
```

• If the user has chosen the nlc compiler output option, then "view" the log file.

```
  if( eq(outopt1 t) then
     view("nlc.log")
  ) ; ** if **
```

• If the user has chosen netlist output option, then depending on which type of netlist was opted for earlier, this procedure will produce the desired output.

```
  if( eq(outopt2 t) then
     setq(netDirOpt netlistAndSymsForm -> netDir -> value)
     setq(fileName mainMenuForm -> filename -> value)
     setq(fileNameLength strlen(fileName))
     setq(reqdLength fileNameLength -  3)
     setq(newFileLength fileNameLength * -1)
     setq(fileNameStem substring(fileName newFileLength reqdLength))
```

• If the user has chosen the WIR option, then check whether they opted to send the output to the standard output or to an appropriate file, if not then display an error message.

```
if( netDirOpt == "wir" then
   setq(fileExtA wirForm -> wirExt -> value)
   if( fileExtA == "-" then
      setq(newFileExtA " ")
   else
      setq(newFileExtA fileExtA)
   ) ; ** if **
   setq(wirDirName wirForm -> wirDir -> value)
   if( wirDirName == "" then
     gotoDisplayWarningMessage1()
   else
     if( wirDirName != "-" then
        setq(wirDirLength strlen(wirDirName))
        setq(newWirDirName substring(wirDirName 3 wirDirLength))
        setq(newFileName strcat(newWirDirName "/" fileNameStem
             newFileExtA))
        view(newFileName)
     else
        println("Netlist directory options (wir) stated the file be
                written to the stdout")
        p = outfile("nlcStdOutNet1")
        fprintf(p "#! /bin/csh -f \n")
        fprintf(p "# Write wir netlist directory to stdout\n\n")
        fprintf(p "set nameOfFileStem = %s\n" fileNameStem)
        fprintf(p "set netDirExt = %s\n\n" newFileExtA)
        fprintf(p "more wir/$nameOfFileStem$netDirExt \n")
        fprintf(p "exit")
        close(p)
        system("xterm -e nlcStdOutNet1")
     ) ; ** if wirDirName **
   ) ; ** if wirDirName **
```

• If the user has chosen the XNF option, then perform the same checks as for the WIR option above.

```
   else ; ** ( if netDirOpt == "xnf" ) **
      setq(fileExtB xnfForm -> xnfExt -> value)
      if( fileExtB == "-" then
         setq(newFileExtB " ")
      else
         setq(newFileExtB fileExtB)
```

```
        ) ; ** if **
        setq(xnfDirName xnfForm -> xnfDir -> value)
        if( xnfDirName == "" then
           gotoDisplayWarningMessage1()
        else
           if( xnfDirName != "-" then
              setq(xnfDirLength strlen(xnfDirName))
              setq(newXnfDirName substring(xnfDirName 3 xnfDirLength))
              setq(newFileName strcat(newXnfDirName "/" fileNameStem
                   newFileExtB))
              view(newFileName)
           else
              println("Netlist directory options (xnf) stated the file
                       be written to the stdout")
              p = outfile("nlcStdOutNet2")
              fprintf(p "#! /bin/csh -f \n")
              fprintf(p "# Write xnf netlist directory to stdout\n\n")
              fprintf(p "set nameOfFileStem = %s\n" fileNameStem)
              fprintf(p "set netDirExt = %s\n\n" newFileExtB)
              fprintf(p "more xnf/$nameOfFileStem$netDirExt \n")
              fprintf(p "exit")
              close(p)
              system("xterm -e nlcStdOutNet2")
           ) ; ** if xnfDirName **
        ) ; ** if xnfDirName **
     ) ; ** if netDirOpt **
  ) ; ** if eq **
```

• If the user has chosen the Symbol option, then check whether they opted to send the output to the standard output or to an appropriate file, if not then display an error message.

```
if( eq(outopt3 t) then
   setq(symDirName netlistAndSymsForm -> symDir -> value)
   if( symDirName == "" then
      gotoDisplayWarningMessage2()
   else
      setq(fileName mainMenuForm -> filename -> value)
      setq(fileNameLength strlen(fileName))
      setq(reqdLength fileNameLength - 3)
      setq(newFileLength fileNameLength * -1)
      setq(fileNameStem substring(fileName newFileLenght reqdLength))
      setq(fileExtC netlistAndSymsForm -> symExt -> value)
      if( fileExtC == "-" then
         setq(newFileExtC " ")
```

```
        else
           setq(newFileExtC fileExtC)
        ) ; ** if **
        if( symDirName != "-" then
           setq(symDirLength strlen(symDirName))
           setq(newSymDirName substring(symDirName 3 symDirLength))
           setq(newFileName strcat(newSymDirName "/" fileNameStem
               newFileExtC))
           view(newFileName)
        else
           println("Symbol directory options stated the file be
                    written to the stdout")
           p = outfile("nlcStdOutSym")
           fprintf(p "#! /bin/csh -f \n")
           fprintf(p "# Write symbol directory to stdout\n\n")
           fprintf(p "set nameOfFileStem = %s \n" fileNameStem)
           fprintf(p "set symDirExt = %s \n\n" newFileExtC)
           fprintf(p "more sym/$nameOfFileStem$symDirExt \n")
           fprintf(p "exit")
           close(p)
           system("xterm -e nlcStdOutSym")
        ) ; ** if symDirName **
      ) ; ** if symDirName **
    ) ; ** if eq **
  ) ; ** procedure - funcNlcOutput **
```

• This creates and displays a form asking the user to choose which compiler outputs he/she wishes to view (see figure 7.18).

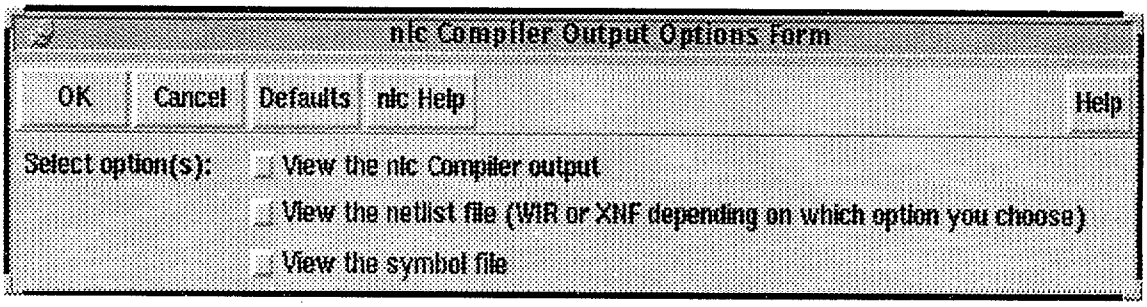


Figure 7.18: Form - "nlc Compiler Output Options Form"

```
procedure(gotoNlcOutputForm()
  outputToggle = hiCreateToggleField(?name        'viewNlcOutput
                                     ?choices     list(
                        '(toggle1 "View the nlc Compiler output")
```

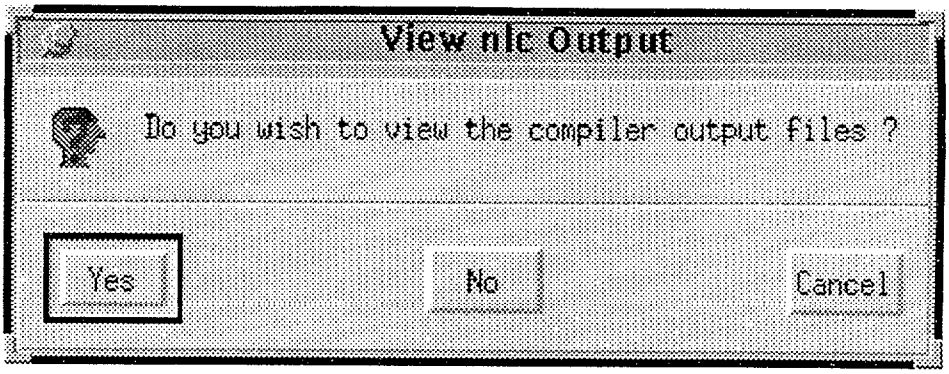


Figure 7.19: Dialog Box - "View nlc Output"

```
                    '(toggle2 "View the netlist file (WIR or XNF
                      depending on which option you choose)")
                    '(toggle3 "View the symbol file")
                                                    )
                              ?numSelect   3
                              ?prompt      "Select option(s):"
                              ?itemsPerRow 1
                ) ; outputToggle **

  hiCreateAppForm(?name         'nlcOutputForm
                  ?fields       '( outputToggle )
                  ?formTitle    "nlc Compiler Output Options Form"
                  ?callback     '(funcNlcOutput)
                  ?buttonLayout '(OKCancelDef
                                  (nlc\ Help "gotoNlcHelpOption()")
                                 )
                  ?initialSize  t
  ) ; ** hiCreateAppForm **

  hiDisplayForm(nlcOutputForm)
) ; ** procedure - gotoNlcOutputForm **
```

• This displays a dialog box asking the user if he/she wishes to view the output files created (see figure 7.19).

```
procedure(gotoNlcOutputBox()
  hiDisplayAppDBox(?name         'nlcOutputBox
                   ?dboxBanner   "View nlc Output"
                   ?dboxText     "Do you wish to view the compiler
                                  output files ?"
                   ?callback     "gotoNlcOutputForm()"
```

```
                    ?dialogType   hicQuestionDialog
                    ?buttonLayout 'YesNoCancel
  ) ; ** hiDisplayAppDBox **
) ; ** procedure - gotoNlcOutputBox **
```

• This is the body of the function *systemCallToNlcCompiler* which passes the necessary information (ie the file name and the options) via the "C Shell Script" to the nlc Compiler using a system call.

```
procedure(systemCallToNlcCompiler(opt1 opt2 opt3 opt4 opt5 opt6)
  setq(fileName mainMenuForm -> filename -> value)
  printf("File Name is: %s \n" fileName)
  printf("Options are: %s %s %s %s %s %s \n\n" opt1 opt2 opt3 opt opt5 opt6)
  p = outfile("nlcRunOpt")
  fprintf(p "#! /bin/csh -f \n")
  fprintf(p "# Run option for nlc \n\n")
  fprintf(p "set nameOfFile = %s \n" fileName)
  fprintf(p "set optionChoice = '%s %s %s %s %s %s' \n\n" opt1 opt2
                                opt3 opt4 opt5 opt6)
  fprintf(p "($NLC $optionChoice $nameOfFile >&nlc.log)\n")
  fprintf(p "exit")
  close( p )
  system("nlcRunOpt")
  gotoNlcOutputBox()
) ; ** procedure - systemCallToNlcCompiler **
```

• This is the body of *funcMenuOptions* which sets the various nlc options if they have been selected by the user and calls function *systemCallToNlcCompiler* to run nlc.

```
procedure(funcMenuOptions( @optional filename compOpt1 compOpt2 compOpt3
                           mainOpt1 mainOpt2 mainOpt3 )
  setq(compilerOption1 mainMenuForm -> compilerOpt1 -> value)
  setq(compilerOption2 mainMenuForm -> compilerOpt2 -> value)
  setq(compilerOption3 mainMenuForm -> compilerOpt3 -> value)
  setq(mainMenuOption1 mainMenuForm -> mainMenuOptions -> toggle1 -> value)
  setq(mainMenuOption2 mainMenuForm -> mainMenuOptions -> toggle2 -> value)
  setq(mainMenuOption3 mainMenuForm -> mainMenuOptions -> toggle3 -> value)

  if( compilerOption1 != "" then
     setq(opt1 strcat(" -P " compilerOption1))
  else
     setq(opt1 " ")
  ) ; ** if **
```

```
  if( compilerOption2 != "" then
     setq(opt2 strcat(" -D " compilerOption2))
  else
     setq(opt2 " ")
  ) ; ** if **

  if( compilerOption3 != "" then
     setq(opt3 strcat(" -I " compilerOption3))
  else
     setq(opt3 " ")
  ) ; ** if **

  if( neq(mainMenuOption1 t) then
     setq(opt4 " ")
  else
     gotoInternalInfoOptions()
  ) ; ** if **

  if( neq(mainMenuOption2 t) then
     setq(opt5 " ")
  else
     gotoNetlistAndSymsOptions()
  ) ; ** if **

  if( neq(mainMenuOption3 t) then
     setq(opt6 " ")
  else
     gotoPrintVersionInfoOption()
     setq(opt6 "-v")
  ) ; ** if **

  systemCallToNlcCompiler(opt1 opt2 opt3 opt4 opt5 opt6)
) ; ** procedure - funcMenuOptions **
```

• This dialog box checks with the user that the file name is in the correct format.

```
procedure(fileNameMessage()
  hiDisplayAppDBox(?name         'fileNameQtnBox
                   ?dboxBanner   "File Name Check"
                   ?dboxText     "Is the file name specified in the style
                                  of the format given ?"
                   ?dialogType   hicQuestionDialog
                   ?buttonLayout 'YesNoCancel
  ) ; ** hiDisplayAppDBox **
) ; ** procedure - fileNameMessage **
```

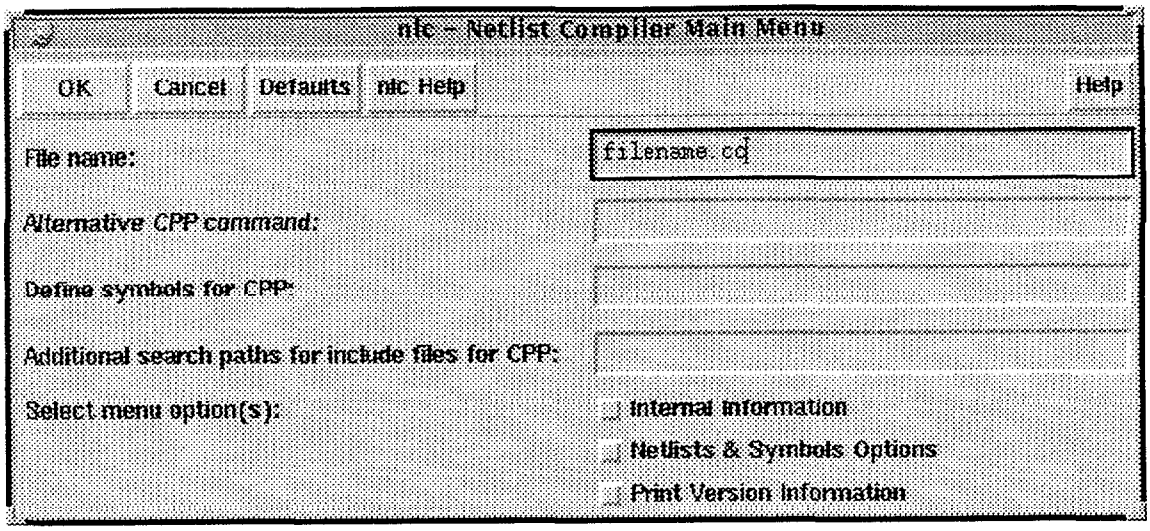


Figure 7.20: Form - "nlc - Netlist Compiler Main Menu"

• This is the body of *funcCompile* in which the *nlc - Netlist Compiler Main Menu* form is created and displayed (see figure 7.20).

```
procedure(funcCompile( @key (stepInst nil) )
  fileNameString = hiCreateStringField(?name        'filename
                                       ?prompt      "File name:"
                                       ?defValue    "filename.cc"
                                       ?callback    "fileNameMessage()"
                   ) ; ** fileNameString **

  compile1String = hiCreateStringField(?name        'compilerOpt1
                                       ?prompt      "Alternative CPP command:"
                   ) ; ** compile1String **

  compile2String = hiCreateStringField(?name        'compilerOpt2
                                       ?prompt      "Define symbols for CPP:"
                   ) ; ** compile2String **

  compile3String = hiCreateStringField(?name        'compilerOpt3
                                       ?prompt      "Additional search paths
                                       for include files for CPP:"
                   ) ; ** compile3String **

  mainToggle = hiCreateToggleField(?name        'mainMenuOptions
                                   ?choices     list(
                                   '(toggle1 "Internal Information")
                                   '(toggle2 "Netlists & Symbols Options")
```

Figure 7.21: Dialog Box - "XNF to Verilog"

```
                                '(toggle3 "Print Version Information")
                                          )
                                ?numSelect   3
                                ?prompt      "Select menu option(s):"
                                ?itemsPerRow 1
                                ?value       '(nil nil nil)
            ) ; ** mainToggle **

  hiCreateAppForm(?name          'mainMenuForm
                  ?formTitle     "nlc - Netlist Compiler Main Menu"
                  ?fields        list( fileNameString compile1String
                                 compile2String compile3String mainToggle )
                  ?callback      '(funcMenuOptions)
                  ?buttonLayout  '(OKCancelDef
                                     (nlc\ Help "gotoNlcHelpOption()")
                                  )
                  ?initialSize  t
  ) ; ** hiCreateAppForm **

  hiDisplayForm(mainMenuForm)
) ; ** procedure - funcCompile **
```

7.3.5 Step 5a XNF to Verilog - *funcXnfToVerilog*

funcXnfToVerilog is the first of the child design steps of the "Verilog" design step. It first asks the user if he/she wishes to translate XNF netlists to Verilog netlists (see figure 7.21) and then produces a form for the user to input the file name (see figure 7.22).It translates the XNF netlists to Verilog netlists via system calls to two procedures, namely "xnf2edif" and "edif2verilog". These procedures require file names which are provided by passing them to the Scripts and then calling the procedures.

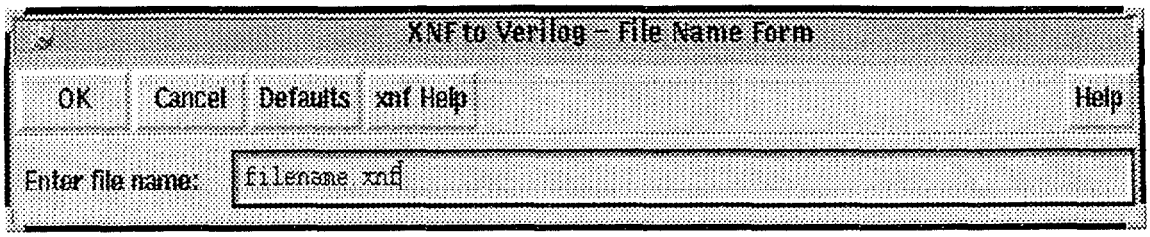


Figure 7.22: Form - "XNF to Verilog - File Name Form"

Figure 7.23: Dialog Box - "XNF to Verilog Help"

• This function is called from the *XNF to Verilog - File Name Form* when the user clicks on the *xnf Help* button. It displays a message box asking the user if they require help(see figure 7.23), if they do, *funcHelp* is called which invokes a system call to the html browser with the relevant "Help page" (see the Appendix).

```
procedure(gotoXnfHelpOption()

  procedure(funcHelp()
    system("netscape html/verilogHelp.html &")
  ) ; ** procedure - funcHelp **

  hiDisplayAppDBox(?name          'helpDBox
                   ?dboxBanner    "XNF to Verilog Help"
                   ?dboxText      "Do you require help ?"
                   ?callback      "funcHelp()"
                   ?dialogType    hicQuestionDialog
                   ?buttonLayout 'YesNoCancel
  ) ; ** hiDisplayAppDBox **
) ; ** procedure - gotoXnfHelpOption  **
```

• *funcGenerateVerilog* invokes two "C Shell Scripts" consecutively to call procedures to, first convert XNF netlists to EDIF netlists, and then EDIF netlists to Verilog netlists.

These two procedures must be invoked with file names which are passed to the "C Shell Scripts" which then invoke the procedures.

```
procedure(funcGenerateVerilog(option)
  setq(xnfFileName xnfToVerilogForm -> filename -> value)
  setq(xnfDirName "xnf/")
  p = outfile("xnfToEdif")
  fprintf(p "#! /bin/csh -f \n")
  fprintf(p "# Generate EDIF netlists from XNF netlists\n\n")
  fprintf(p "set nameOfFile = %s \n" xnfFileName)
  fprintf(p "set nameOfDir = %s \n\n" xnfDirName)
  fprintf(p "(xnf2edif -v $nameOfDir$nameOfFile) \n\n")
  fprintf(p "exit")
  close( p )
  system("xterm -e xnfToEdif")

  setq(fileNameLength strlen(xnfFileName))
  setq(reqdLength fileNameLength - 4)
  setq(newFileLength fileNameLength * -1)
  setq(fileNameStem substring(xnfFileName newFileLength reqdLength))
  setq(edifFileName strcat(fileNameStem ".net"))
  p = outfile("edifToVerilog")
  fprintf(p "#! /bin/csh -f \n")
  fprintf(p "# Generate Verilog netlists from EDIF netlists \n\n")
  fprintf(p "set nameOfEdifFile = %s \n\n" edifFileName)
  fprintf(p "(edif2verilog $nameOfEdifFile -f -nt -v) \n\n")
  fprintf(p "exit")
  close( p )
  system("xterm -e edifToVerilog")
) ; ** procedure - funcGenerateVerilog **
```

- This dialog box checks with the user that the file name is in the correct format.

```
procedure(fileNameMessage()
  hiDisplayAppDBox(?name         'fileNameQtnBox
                   ?dboxBanner   "File Name Check"
                   ?dboxText     "Is the file name specified in the style
                                  of the format given ?"
                   ?dialogType   hicQuestionDialog
                   ?buttonLayout 'YesNoCancel
   ) ; ** hiDisplayAppDBox **
) ; ** procedure - fileNameMessage **
```

- The procedure below creates the form to get the name of the XNF file to be translated.

```
procedure(gotoVerilogForm()
  xnfString = hiCreateStringField(?name        'filename
                                  ?prompt      "Enter file name:"
                                  ?defValue    "filename.xnf"
                                  ?callback    "fileNameMessage()"
              ) ; ** xnfString **

  hiCreateAppForm(?name          'xnfToVerilogForm
                  ?fields        list(xnfString)
                  ?formTitle     "XNF to Verilog - File Name Form"
                  ?callback      '(funcGenerateVerilog)
                  ?buttonLayout '(OKCancelDef
                                   (xnf\ Help "gotoXnfHelpOption()")
                                  )
                  ?initialSize    t
  ) ; ** hiCreateAppForm **

  hiDisplayForm(xnfToVerilogForm)
) ; procedure - gotoVerilogForm **
```

• The dialog box is created here and calls the function *gotoVerilogForm* if the user clicks on *OK*, otherwise the function finishes.

```
procedure(funcXnfToVerilog( @key ( stepInst nil ) )
  hiDisplayAppDBox(?name          'verilogDBox
                   ?dboxBanner    "XNF to Verilog"
                   ?dboxText      "Do you wish to generate Verilog netlists
                                   from XNF netlists ?"
                   ?callback      "gotoVerilogForm()"
                   ?dialogType    hicQuestionDialog
                   ?buttonLayout 'YesNoCancel
    ) ; ** hiDisplayAppDBox **
) ; ** procedure - funcXnfToVerilog **
```

7.3.6 Step 5 ViewVerilog - *funcViewVerilog*

funcViewVerilog is the second child step of the "Verilog" design step. It first asks the user if he/she wishes to view the Verilog netlists and schematic (see figure 7.24) and then produces a form for the user to input the file name (see figure 7.25). It allows the user to view the Verilog netlists and the schematic. This function invokes two of the built-in Cadence tools, namely "view" and "Verilog-In".

• This function is called from the *View Verilog - File Name Form* when the user clicks on the *Verilog Help* button. It displays a message box asking the user if they require help

Figure 7.24: Dialog Box - "View Verilog"

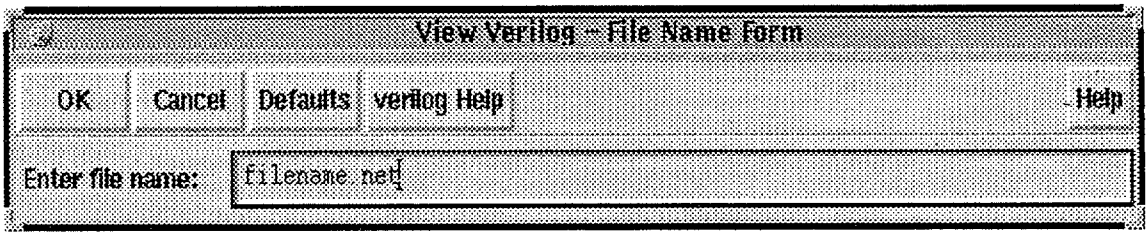


Figure 7.25: Form - "View Verilog - File Name Form"

(see figure 7.26), if they do, *funcHelp* is called which invokes a system call to the html browser with the relevant "Help page" (see the Appendix).

Figure 7.26: Dialog Box - "View Verilog Help"

```
procedure(gotoVerilogHelpOption()

  procedure(funcHelp()
    system("$HTMLBROWSER html/verilogHelp.html &")
  ) ; ** procedure - funcHelp **
```

```
  hiDisplayAppDBox(?name         'helpDBox
                   ?dboxBanner   "View Verilog Help"
                   ?dboxText     "Do you require help ?"
                   ?callback     "funcHelp()"
                   ?dialogType   hicQuestionDialog
                   ?buttonLayout 'YesNoCancel
  ) ; ** hiDisplayAppDBox **
) ; ** procedure - gotoVerilogHelpOption  **
```

• The procedure *funcView* passes the name of the Verilog file to a SKILL command *view* which brings up the file in a view window. A call is then made, via the SKILL command *impHdlDisplay*, which brings up the "Verilog In" form from Cadence. The user can then fill this form out and it will allow him/her to view the schematic.

```
procedure(funcView(option)
  setq(verilogFileName viewVerilogForm -> filename -> value )
  view(verilogFileName)
  impHdlDisplay(impHdlOptionsForm)
) ; ** procedure - funcView **
```

• The procedure below creates the form to get the name of the Verilog file to be translated.

```
procedure(gotoViewVerilogForm()
  verilogString = hiCreateStringField(?name        'filename
                                      ?prompt      "Enter file name:"
                                      ?defValue    "filename.net"
                                      ?callback    "fileNameMessage()"
                  ) ; ** verilogString **

  hiCreateAppForm(?name         'viewVerilogForm
                  ?fields       list(verilogString)
                  ?formTitle    "View Verilog - File Name Form"
                  ?callback     '(funcView)
                  ?buttonLayout '(OKCancelDef
                                  (verilog\ Help "gotoVerilogHelpOption()")
                                 )
                  ?initialSize   t
  ) ; ** hiCreateAppForm **

  hiDisplayForm(viewVerilogForm)
) ; procedure - gotoViewVerilogForm **
```

• The dialog box is created here and calls the function *gotoViewVerilogForm* if the user clicks on *OK*, otherwise the function finishes.

```
procedure(funcViewVerilog( @key (stepInst nil) )
  hiDisplayAppDBox(?name         'verilog2DBox
                   ?dboxBanner   " View Verilog"
                   ?dboxText     "Do you wish to view the Verilog netlists
                                 and the schematic?"
                   ?callback     "gotoViewVerilogForm()"
                   ?dialogType   hicQuestionDialog
                   ?buttonLayout 'YesNoCancel
    ) ; ** hiDisplayAppDBox **
) ; procedure - funcViewVerilog **
```

7.3.7 Step 6 Xilinx Design Manager - *funcXilinx*

funcXilinx first creates a dialog box to inquire of the user if he/she wishes to start the Xilinx Design Manager(see figure 7.27). By clicking on *Yes*, it makes a simple system call to the tool.

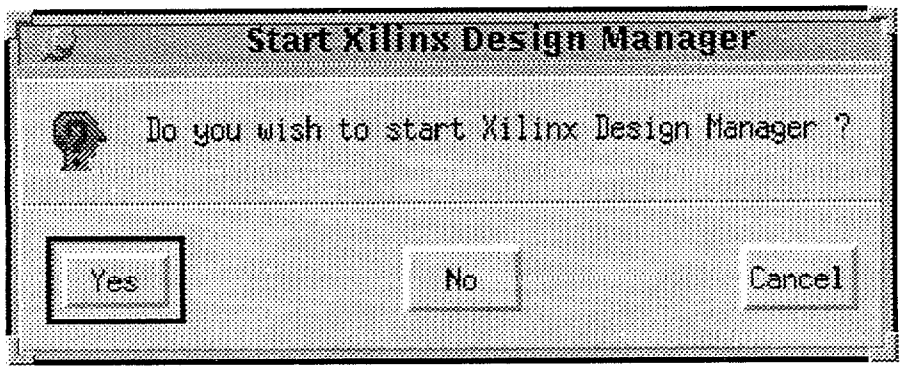


Figure 7.27: Dialog Box - "Start Xilinx Design Manager"

- This procedure calls the Xilinx Design Manager.

```
procedure(funcCallToXilinx()
  system("XDM &")
) ; ** procedure - funcCallToXilinx **
```

- The dialog box is created here and calls the function *funcCallToXilinx* if the user clicks on *OK*, otherwise the function finishes.

```
procedure(funcXilinx( @key ( stepInst nil ) )
  hiDisplayAppDBox(?name         'xilinxDBox
                   ?dboxBanner   "Start Xilinx Design Manager"
                   ?dboxText     "Do you wish to start Xilinx Design Manager ?"
                   ?callback     "funcCallToXilinx()"
```

```
                    ?dialogType   hicQuestionDialog
                    ?buttonLayout 'YesNoCancel
    ) ; ** hiDisplayAppDBox **
) ; ** procedure - funcXilinx **
```

7.4 Adding Buttons to the Flow Browser Window

Add a "System Help" button to the menu banner of the Flow Browser window
This is achieved by getting the identity (window number) of the window of flow browser containing the flowchart, creating a banner menu and assigning the function to call the "System Help" pages to a menu item on a pull-dwn menu (see figure 7.28).

• This function is called from the *System Help* menu item on the banner of the flow browser window. It displays a message box asking the user if they require help, if they do, *funcHelp* is called which invokes a system call to the html browser with the relevant "Help pages" (see the Appendix).

```
  procedure(funcHelp()
    system("$HTMLBROWSER html/manual.html &")
  ) ; ** funcHelp **
procedure(funcSystemHelp()
  hiDisplayAppDBox(?name         'helpDBox
                   ?dboxBanner   "System Help"
                   ?dboxText     "Do you require help on how to use this
                                  system ?"
                   ?callback     "funcHelp()"
                   ?dialogType   hicQuestionDialog
                   ?buttonLayout 'YesNoCancel
  ) ; ** hiDisplayAppDBox **
) ; ** procedure - funcSystemHelp **
```

• A button is created on the banner of the flow browser window in which the design flow is contained. To create a banner button it is necessary to first create a pull-down menu to attach to the button.

```
winId = hiGetCurrentWindow()

menuItem = hiCreateMenuItem(?name        'helpItem
                            ?itemText    "System Help"
                            ?callback    "funcSystemHelp"
       )

hiCreatePulldownMenu('sysHelpMenu "System Help" '(menuItem))

hiInsertBannerMenu(winId 'sysHelpMenu 2)
```

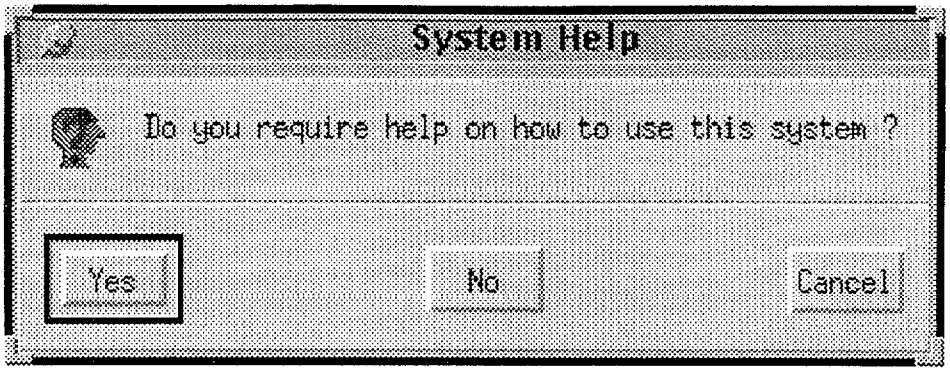


Figure 7.28: Dialog Box - "System Help"

Add a "Reset System" button to the menu banner of the Flow Browser window
This is achieved by getting the identity (window number) of the window of flow browser containing the flowchart, creating a banner menu and assigning the necessary commands to reset all the steps. A warning dialog box appears first to alert the user to the fact that the whole system can only be reset after the all the steps have been set to "Done"(see figure 7.29)..

- This function contains the SKILL commands to reset each step.

```
procedure(gotoResetSystem()
  dagDoNamedAction( dagNumToTool(1) dagGetObjectByPath( dagNumToTool(1)
                '( "dummyNode" "dfDummyClass" "Create Directory.0"
                   "dfDefaultClass" ) ) "Reset Step" )
  dagDoNamedAction( dagNumToTool(1) dagGetObjectByPath( dagNumToTool(1)
                '( "dummyNode" "dfDummyClass" "Edit File.0"
                   "dfDefaultClass" ) ) "Reset Step" )
  dagDoNamedAction( dagNumToTool(1) dagGetObjectByPath( dagNumToTool(1)
                '( "dummyNode" "dfDummyClass" "C++ Debugger.0"
                   "dfDefaultClass" ) ) "Reset Step" )
  dagDoNamedAction( dagNumToTool(1) dagGetObjectByPath( dagNumToTool(1)
                '( "dummyNode" "dfDummyClass" "nlc Compiler.0"
                   "dfDefaultClass" ) ) "Reset Step" )
  dagDoNamedAction( dagNumToTool(1) dagGetObjectByPath( dagNumToTool(1)
                '( "dummyNode" "dfDummyClass" "Verilog.0"
                   "dfDefaultClass" ) ) "Reset Step" )
  dagDoNamedAction( dagNumToTool(2) dagGetObjectByPath( dagNumToTool(2)
                '( "dummyNode" "dfDummyClass" "XNF to Verilog.0"
                   "dfDefaultClass" ) ) "Reset Step" )
  dagDoNamedAction( dagNumToTool(2) dagGetObjectByPath( dagNumToTool(2)
                '( "dummyNode" "dfDummyClass" "View Verilog.0"
                   "dfDefaultClass" ) ) "Reset Step" )
  dagDoNamedAction( dagNumToTool(1) dagGetObjectByPath( dagNumToTool(1)
```

Figure 7.29: Dialog Box - "Reset System Warning"

```
                '( "dummyNode" "dfDummyClass" "Xilinx Design Manager.0"
                   "dfDefaultClass" ) ) "Reset Step" )
  ) ; ** procedure - gotoResetSystem **
```

• The dialog box containing the warning which, if the user clikcs on *OK*, calls the function *gotoResetSystem.*

```
procedure(funcResetSystem()
  hiDisplayAppDBox(?name         'resetDBox
                   ?dboxBanner   "Reset System Warning"
                   ?dboxText     "You can only reset the system if ALL
                                  the steps have been set to DONE"
                   ?callback     "gotoResetSystem()"
                   ?dialogType   hicWarningDialog
                   ?buttonLayout 'OKCancel
  ) ; ** hiDisplayAppDBox **

) ; ** procedure - funcResetSystem **
```

• A button, *Reset System*, is created on the banner of the flow browser window in which the design flow is contained. To create a banner button it is necessary to first create a pull-down menu and the necessary menu item(s), ie *Reset All Steps*, to attach to the button. When the user clicks on this menu item, the function *funcResetSystem* is called.

```
winId2 = hiGetCurrentWindow()

menuItem2 = hiCreateMenuItem(?name         'resetItem
                             ?itemText     "Reset All Steps"
                             ?callback     "funcResetSystem"
            )
```

```
hiCreatePulldownMenu('resetSystem "Reset" '(menuItem2))

hiInsertBannerMenu(winId2 'resetSystem 1)
```

7.5 External Programs

From the *NAC - Nlc & Cadence* System, the following external programs have been used:

1. **EDITOR** - *emacs*, the GNU project Emacs
2. **HTMLBROWSER** - *Netscape*, the preferred html browser
3. **NLC** - adapted version of *nlc* which includes a facility to generate XNF netlist as well as WIR netlists
4. **DEBUGGER** - *xdbx*, the GNU debugging tool
5. **xnf2edif** - Xilinx program to translate XNF netlists to EDIF netlists
6. **edif2verilog** - Xilinx program to translate EDIF netlists to Verilog
7. **XDM** - Xilinx Design Manager

To access the EDITOR, HTMLBROWSER and XDM simple "system" calls are made from the source code, with the appropriate file where necessary.

To access NLC, a "C Shell Script" is created, called *nlcRunOpt*, which calls NLC with the appropriate file name and options gathered from the user information inputted during run-time. When using NLC, further C Shell Scripts are used to send information to the *stdout* (i.e. standard output), namely *nlcStdOutNet1*, *nlcStdOutNet2* and *nlcStdOutSym*, to output the WIR, XNF and Symbol files, respectively, to the standard output.

To activate the two Xilinx translation programs, xnf2edif and edif2verilog, two C Shell Scripts are used, called *xnfToEdif* and *edifToVerilog*. As with NLC, these scripts call the two programs with the appropriate file name and extensions gathered from the user information inputted during run-time.

A list of all the programs and files associated with NAC can be seen in the README file.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

The Cadence DFW II has proven to fulfill many of the qualities and characteristics necessary of a CAD framework, especially with its *open architecture* permitting efficient and effective tool integration.

Tool integration is achieved primarily via *encapsulation* of the application using the extension language, SKILL, and the Design Flow Management System. The graphical design flow system presents a clearly set out design flow which is easy to use, and aids the designer in keeping track of the state of the design. The powerful language enables the designer to create complex user-defined SKILL functions to utilize alongside the Cadence built-in SKILL functions. This combination provides an environment for the integration of tools which is straightforward to use and easy to understand.

The development of NAC - *Nlc & Cadence* has shown how it is possible to select the tools that the user requires and incorporate them into one system which provides a user-friendly environment and alleviates the user from having to access each tool on its own. A straightforward approach has been used, which highlights the benefits of the Design Flow Management System. The steps of the design flow were first created and encapsulated, and then the functions, which each step called, were developed and added to the steps to integrate the tools into the system.

One of the major benefits of creating NAC is that it can be used as a general methodology in creating more systems that integrate tools of a different kind into the Cadence environmment.

There are, however a few problems which have been incurred with certain tools, namely the "xnf2edif" and "edif2verilog" programs. They have been encapsulated within NAC, but problems arise during run-time. Both are called from NAC so the integration is correct, but the netlists that are generated from "xnf2edif" are not able to be translated into verilog netlists when "edif2verilog" is run. This is the 1st version of NAC, and these programs are still alpha code which need to be further refined. Aside from this problem, the actual task of encapsulation the tools into the system has been achieved.

8.2 Future Work

Modularity is one of the key words that can be used to describe Cadence's Design Flow Management System. The modular environment enables the design system to be modified with out having to redesign the entire system. New requirements can be met, and new tools can be added easily. This feature demonstrates the versatility and flexibility of the Cadence DFW II and how it lends itself to facilitate *tool interchangeability*. With this in mind, NAC can be further developed by adding new tools without any undue difficulties.

More options could be provided at each step, for example, with the text editor that you use in the "Edit File" step, or with the debugger that is called form the "C++ Debugger" step, a choice could be made by the user, instead of the system calling the "defaults" of the operating system.

To improve NAC in the future, the EDIF interface could be improved and an nlc library (header file) could be generated from an edif description. Other facilities could be integrated, such as a tool to generate symbols in Cadence format. Further Cadence tools could be integrated which, once the verilog netlists have been built, simulation with verilog is carried out.

Appendix

Online Help Pages

The *Online Help Pages* include the following:

1. System Help (Online Manual)
2. Information and Help on nlc
3. nlc Information Page
4. Help on XNF to Verilog and Viewing Verilog

These "Help Pages" have been transferred from their *html* equivalent. They are the online help documentation that can be accessed from the NAC system design flow via the html browser.

All the figures that appear in the pages of the online help documentation, have been removed, to save duplication, and instead references to where they are to be found in the preceding chapters of this report appear.

User Manual and Online System Help

Before You Start

About This Manual This manual is for all users of the Design Flow. It can be accessed online from the design flow via the *System Help* button on the banner of the Flow Browser window that it appears in. A more indepth guide to the "nlc Compiler" and "Verilog" steps can be seen in the online help pages that follow, which are accessed from the design flow via the *nlc Help* and *verilog Help* buttons that appear on the banner of all the associated forms.

Introducing the Design Flow

Overview This system is presented in a flowchart which guides you through the series of steps. It enables you to, first create a new directory, then edit an existing C++ file or create a new one. It goes on to generate XNF netlists from a C++ program and convert them to Verilog netlists. You can then view the Verilog netlists and the schematic, and finally call the Xilinx Design Manager to further implement the logical design.

Getting Started / Working with the Flow Browser Window / Working with the Flowchart Steps

1. Getting Started

At the Command Interpreter Window (CIW) command line, type

```
load "desproj.il"
```

The Design Flow will appear in a Flow Browser Window (see figure 7.1).

2. Working with the Flow Browser Window

Banner Buttons The Flow Browser window has 4 buttons on the banner. Two are automatically created with a Flow Browser window and two have been created for this design flow.

1. **System** - the *System* pull-down menu is automatically created and contains 4 menu items:

(a) **Save Flowchart** - encrypts and saves the information associated with the displayed flowchart in the Flow Browser window as a cellview in the design library. The cell name is the name of the displayed flowchart, and the view name is *flowchartInst*.

(b) **Switch to New Design** - displays another flowchart instance you specify from the same design library or from a different design library in the same Flow Browser window. This command displays a form and the Library Browser to help you select a new design. If you do not enter any information into this form, the flowchart instance that is currently displayed is redisplayed.

(c) **Redraw** - redisplays the flowchart that is already displayed in the Flow Browser window.

(d) **Close Window** - saves the displayed flowchart and closes the Flow Browser window.

2. **Help** - the *Help* button provides by the Cadence environment informs the user to call "openbook", the Cadence online documentation.

3. **Reset** - contains one menu item to **Reset All Steps** of the design flow. When you click on this, it first displays a warning dialog box to inform the user the only if all the steps have been set to *Done* (ie, by each step being correctly finished, or **Bypass**ed), can then whole system be reset. Click on *Yes* to reset all the steps (including those in the subflowchart). Click on *No* or *Cancel* to stop the system from being reset.

4. **System Help** - contains one menu item, *System Help*. When you click on this, it first displays a question dialog box to ask you if you require help on this system. Click on *Yes* if you do and this manual in its online form will appear via the html browser.

Pop-up Menu Each design step contains a Pop-up Menu which is activated by clicking and holding the middle mouse button over the design step. By default this menu has two commands, *Run Step* and *Reset Step*. All design steps that are created have these two commands associated with them. Some of the steps in this system have other buttons added to the pop-up menu. Each of the six primary steps in the design flow have an extra command called *Bypass Step*, and the "Verilog" design step also has a *Push to Subflowchart* command.

1. **Run Step** - runs the design step. Click and hold the middle mouse button in the design step box and place the cursor over *Run Step* in the pop-up menu to run the design step.

2. **Reset Step** - resets the design step. Click and hold the middle mouse button in the design step box and place the cursor over *Reset Step* in the pop-up menu to reset the design step.

3. **Bypass Step** - bypasses the design step, ie sets the step to *Done* without having to first run the step. Click and hold the middle mouse button in the design step box and place the cursor over *Bypass Step* in the pop-up menu to bypass the design step.

4. **Push to Subflowchart** - displays a subflowchart referenced by the design step. Click and hold the middle mouse button in the design step box and place the cursor over *Push to Subflowchart* in the pop-up menu to display the subflowchart. This action can only be invoked if the design step references a subflowchart

Design Step Commands At the top of each design step there are two buttons, *Run* and *Props*. On a design step which has a subflowchart attached, there is a downward arrow on the design step.

1. **Run** - runs the design step. Place the cursor over *Run* in the design step and click right with the mouse to run the design step.

2. **Props** - displays the run-time properties of the design step. Place the cursor over *Props* in the design step and click right with the mouse to run to display the run-time properties. The properties are displayed in a text window.

3. **Down Arrow** - displays a subflowchart referenced by the design step. Place the cursor over down arrow in the design step and click right with the mouse to display the subflowchart.

3. Working with the Flowchart Steps

Step 1 - Create Directory When you run this step a question dialog box appears asking you if you wish to create a new directory (see figure 7.3). Click on *Yes* if you do, and a form will appear for you to input the name of the directory to be created (see figure 7.4). Click on *OK* and the directory will be created and the design step is set to *Done*.

Step 2 - Edit File When you run this step a question dialog box appears asking you if you wish to edit or create a C++ file (see figure 7.5). Click on *Yes* if you do, and a form will appear for you to enter the name of the file (see figure 7.6). Enter the name of the file and click on *OK* and the file will appear in a text window with the text editor. The design step sets itself to *Done*.

Step 3 - C++ Debugger When you run this step a question dialog box appears asking you if you wish to debug a C++ file (see figure 7.7). Click on *Yes* if you do, and a form will appear for you to enter the name of the file (see figure 7.8). Enter the name of the file and click on *OK* and the debugger will be invoked. The design step sets itself to *Done*.

Step 4 - nlc Compiler When you run this step, the "nlc Netlist Compiler Main Menu" appears (see figure 7.20). For online help on this step, click on the *nlc Help* button on the banner of the form. The information on this step can be seen later in the Appendix.

Step 5 - Verilog By clicking on the *Push to Subflowchart* button on the pop-up menu, or clicking on the down arrow, the "Verilog" subflowchart appears (see figure 7.2). The information on this step can be seen later in the Appendix.

Step 5a - XNF to Verilog When you run this step a question dialog box appears asking you if you wish to generate Verilog netlists from XNF netlists (see figure 7.21). Click on *Yes* if you do, and a form will appear for you to enter the name of the file (see figure 7.22). Click on *verilog Help* for online help, or enter the name of the file and click on *OK* and the XNF netlists are translated to Verilog netlists, via EDIF.

Step 5a - View Verilog When you run this step a question dialog box appears asking you if you wish to view the Verilog netlists and the schematic (see figure 7.24). Click on *Yes* if you do, and a form will appear for you to enter the name of the file (see figure 7.25). Click on *verilog Help* for online help, or enter the name of the file and click on *OK* and a view file with the netlists will appear and also the Cadence "Verilog In" form.

Step 5 - Xilinx Design Manager When you run this step a question dialog box appears asking you if you wish to start the Xilinx Design Manager (see figure 7.27). Click on *Yes* if you do, and the Xilinx Design Manager will be invoked.

Information and Help on nlc

nlc is a C to Netlist Compiler. It translates a source file written in a subset of the C (or C++) language into a netlist. This netlist is suitable as input for CAD tools that accept the WIR netlist format from ViewLogic.

Information on nlc

The information available on nlc includes information on the author, the language, how to build nlc, some examples of source input files and the limitations.

Help is available on the following compiler screens:

nlc - Netlist Compiler Main Menu / Internal Information / Netlist & Symbols Options / WIR Directory Option / XNF Directory Option / Print Version Information / Viewing the Output Files

nlc - NetList Compiler Main Menu

Please note that the following nlc compiler options may be used in any combination.

- **File name:**
  - You must enter the name of the C (or C++) source file you wish to compile (the format of the file name is given)
  - The following message box will then appear:
  - If you have specified the file name in the correct format, click on *Yes*, if not, then click on *No* and you can retype the file name.

  N.B. If you want to view the version information only, then a file name is not necessary, so simply delete the file name format and click *No* on the message box (and select the *Print Version Information option*).

- **Alternative CPP command:**(the default is "gcc -E -x c++ -D_NLC _")
  - If you select this option you can specify another C++ compiler

- **Define symbols for CPP:**

  - If you select this option, you can specify symbols

- **Additional search paths for include files for CPP:**

  - If you select this option, you can enter the search path of any include files necessary for your program (an example of one path has been given as the default)

- **Select menu option(s):**

  - All, one, two or none of these options can be selected

Internal Information Options Form

If you select this option from the main menu, you can then select as as many or as few of the options that appear on the form.

Netlist & Symbols Options Form

If you select this option from the main menu, you can then select as many or as few of the options that appear on the form.

- **Select option(s):**

  - You can select one, both, or neither of these options

- **Select which directory netlists are written to:**

  - You can select one of these. If you select *xnf* then you cannot select *wir*, and vice versa.

- **Directory symbols are written to:** (the default is already given)

  - If you want to write to stdout, replace the default with a dash (-)

  N.B. Ensure the name of the directory is three characters long, as in the default shown.

- **The extension for symbols:** (the default is already given)

  - If you do not want an extension, replace the default with a dash (-)

- **Change the bitvector size:** (the default is already given)

  - This may be changed to the size you require

WIR Directory Option Form

- **Directory WIR netlists are written to:** (the default is already given)
  - If you want to write to stdout, replace the default with a dash (-)

  N.B. Ensure the name of the directory is three characters long, as in the default shown.
- **The extension for netlists:** (the default is already given)
  - If you do not want an extension, replace the default with a dash (-)

XNF Directory Option Form

- **Directory XNF netlists are written to:** (the default is already given)
  - If you want to write to stdout, replace the default with a dash (-)

  N.B. Ensure the name of the directory is three characters long, as in the default shown.
- **The extension for netlists:** (the default is already given)
  - If you do not want an extension, replace the default with a dash (-)

Version Information Message

If you select this option only, a file name is not necessary as it simply prints the version of nlc.

nlc Compiler Output Options Form

If you wish to view the nlc compiler output files, then click on *Yes* and the following screen will appear:

You can choose to view one, two, all, or none of the output files. **N.B.** You can only view the netlist and symbol files if the directories have been specified in the *Netlist & Symbols Options Form*, otherwise the following warning messages will appear:

N.B. If you have specified that the netlist and/or symbol directories are to be written to the *stdout*, then then output can be viewed via the command tool window.

nlc Information Page

XNF to Verilog / View Verilog

The following information is available:

- **nlc Author Information**
- **nlc Version Information**
- **How to build nlc**
- **The nlc Language**
- **Examples of source files for nlc**
- **The nlc bitvector C++ library**
- **Limitations of nlc**

N.B. All this information is from the author of nlc.

nlc Author Information

Christian Iseli
Laboratoire de Systemes Logiques
Ecole Polytechnique Federale de Lausanne
Lausanne, Switzerland

Christian.Iseli@di.epfl.ch

nlc Version Information

NLC Version 0.8

Laboratoire de Systemes Logiques
Ecole Polytechnique Federale de Lausanne, Switzerland

This program is free software; you can redistribute it and/or modify it under the terms of either:

- a) the GNU General Public License as published by the Free Software Foundation; either version 2 (if necessary, can be found at */home/asic1/dcad/tmp/nlc-0.8/GPL*), or

- b) the "Artistic License" which comes with this Kit (if necessary, can be found at */home/asic1/dcad/tmp/nlc-0.8/Artistic*)

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See either the GNU General Public License or the Artistic License for more details.

How to build nlc

Introduction The Spyder project involves the development of a reconfigurable processor using FPGAs. Unlike most other reconfigurable processors, Spyder has a fixed underlying structure. It is based on the VLIW processor architecture, with a single control unit, shared multi-port registers and multiple reconfigurable execution units. The reconfigurability comes mostly from the execution units, which are implemented using Xilinx FPGA chips.

The interest to describe circuitry in C lies, for me, in the fact of being able to simulate it (or better said, to emulate it) as part of a larger C++ (or C) program. That's why this compiler came into being.

The C to netlist compiler, or nlc for short, is a part of the Spyder project. It is used to create the configuration of the execution units from a high-level description of their functionality in a subset of C++ (or C). This compiler knows about all the standard arithmetic and logic operators of C, except divide (/). It understands FOR loops, IF THEN ELSE constructs and functions. There is only one data type, which is called bitvector. The default length of a bitvector is 16 bits, as Spyder is a 16-bit architecture, and nlc can operate on single bits or on the whole bitvector.

Below you will find some hopefully useful information regarding this compiler. Try it out, hack away and have fun. Feel free to contact me with any comment, question, remark, suggestion, bug fix and improvement. The list *nlc@lslsun.epfl.ch* is also open for any public discussion about this compiler.

Building nlc In order to build nlc, you will need:

- a C++ compiler
- the LEDA 3.1 library, available for example from:
  - *ftp://ftp.mpi-sb.mpg.de/pub/LEDA*

– or some other archive site near you (ask archie...) the libg++ library

- the bison parser generator. I'm not sure if yacc will work...

Unpack the source code and try to type "make". If not, you might have to tweak the Makefile a bit...

I have built it here on a SPARCstation running Solaris 2.3, using gcc 2.6.3. It compiles cleanly, even with the -Wall option of gcc turned on.

You can try out the compiler by running "make check". This will check the current output with some reference output. Some minor differences might be normal...

The nlc language First a word about the intent of this language. The philosophy adopted is that everything happens in parallel. All loops are expanded. It is assumed that there is some external control device that calls the top level routine once every clock cycle. The static bitvectors (registers) are written once per clock cycle, at the end of the cycle.

The only data type is bitvector, so everything is a bitvector. They are similar to an array of bits and also to int. By default, a bitvector contains 16 bits, but bitvectors of specific length can be created by specifying the size explicitly to the constructor. You can create one-dimensional arrays of bitvectors (see example below).

Here is a small example:

```
1 foo(bitvector in(4), bitvector &out(4))
2 {
3    bitvector a, b(8), c(2)[10];
4    static bitvector r;
5    a = 8;
6    a[0] = b[3] ^ a[3];
7    a = a ^ b;
8 }
```

Line 1 declares a function (a module) with one 4-bit input vector in and one 4-bit output vector out. Line 3 declares a to be a bitvector of 16 bits, b a bitvector of 8 bits and c an array of bitvectors of 2 bits. Line 4 declares a static bitvector (a hardware register) that will keep its value across calls to the function. Line 5 assigns 8 to the whole a bitvector. Line 6 assigns the result of the exclusive or of bits b[3] and a[3] to the bit a[0] and line 7 performs the bitwise exclusive or of a and b.

As can be seen from the above example, there are some slight "extensions" from standard C++ code... in(4) is not allowed as a parameter specifier in C++, and c(2)[10] is not allowed either. That's why there are some *#ifdefs* in the example files...

There are two control flow constructs: FOR loops and IF THEN ELSE. FOR loop bounds must be known at compile time.

All functions have a void return type (i.e., they cannot return anything).

The library that the compiler is to use to generate the netlist must be described in the source file (or included by the source file). The general format of the description is:

```
asm OP {
  "OP NAME" [< {"SYMBOL ATTRIBUTE"}> ],
  ( "INPUT_1" [< {"PIN ATTRIBUTE"}> ], "INPUT_2", ...,
    "INPUT_N" ),
  ( "OUTPUT_1" [< {"PIN ATTRIBUTE"} >], "OUTPUT_2", ...,
    "OUTPUT_N" )
}
```

where OP is one of " \& | ^ ~ = == < ?static'': +:the addition. For the time being, thc subtraction is implemented with an addition where the second entry is complemented and the carry bit is set to 1.

```
asm + { "name", ("A inputs", "B inputs", "Carry in"),
      ("Carry out", "Sum")}
 &: logical and.
asm & { "name", ("A inputs", "B inputs"), ("Outputs")}
  |: logical or.
asm | { "name", ("A inputs", "B inputs"), ("Outputs")}
  ^: logical exclusive or (xor).
asm ^ { "name", ("A inputs", "B inputs"), ("Outputs")}
  ~: logical not.
asm ~ { "name", ("Inputs"), ("Outputs")}
  =: assignment (buffer).
asm = { "name", ("Inputs"), ("Outputs")}
  ==: equality comparison.
asm == { "name", ("A inputs", "B inputs"), ("Output")}
  <: a less than b comparison. This is always an unsigned
comparison. Optionally, the second output is the greater than output.
asm < { "name", ("A inputs", "B inputs"), ("LT output" [,
        "GT output"])}
  ?: multiplexor. If select == 0 then output = input_0,
     else output = input_1
asm ? { "name" , ("Select", "0 inputs", "1 inputs"), ("Outputs")}
  static: a register. Used to implement static bitvectors.
asm static { "name" , ("Data", "Clock", "Clk Enable"), ("Outputs")}
```

The \&, |, ^, ~, =, ?and static operators are mandatory (if they are used by the source code being compiled). The others will be generated by the compiler using basic

gates, as needed. You can specify more than one implementation for an operator, the compiler will use what it think is the best match.

The easiest thing is to take a look at the tests/x4000.h file, which is the description of the Xilinx XC4000 device.

The PINORDER and PINTYPE attributes are generated automatically by the compiler. Any other attribute must be specified in the operator description.

Examples of source files for nlc

Here are some examples of source files that will give you an idea of what nlc expects as its input file:

Example 1

testfor.cc,v 1.2 1994/11/04 13:00:51 chris Exp

This is an example of source code for nlc. Compile it by typing nlc testfor.in and then Autogen then you can go in the PV/tests directory and try to simulate the resulting netlist using ViewSim. A sample command file is provided for the simulator.

```
#ifdef __NLC__
#include "x4000.h"
#endif

testfor(bitvector in, bitvector &out)
{
  bitvector i;

  out[0] = in[15] ^ in[1];
  for (i=1;i<=14;i++) {
    out[i] = in[i-1] ^ in[i+1];
  }
  out[15] = in[14] ^ in[0];
}
```

Example 2

testinc.cc,v 1.1 1994/11/22 08:22:59 chris Exp

Test increments.

```
#ifdef __NLC__
#include "x4000.h"
#endif

testinc(bitvector cmd(1), bitvector
  static bitvector old;
```

```
  if (cmd[0])
    old = cur++;
  out = old;
)
```

Example 3
testop.cc,v 1.2 1994/11/04 13:00:55 chris Exp

```
#ifdef __NLC__
#include "x4000.h"
#endif

testop(bitvector in, bitvector

  for (i=1;i i++) {
    out[i] = in[i-1] ^ in[i+1];
  {
  j = in + i;
  k = i - j;
  out = i * j;
  out = i
  out = i | j;
  out = i ^ j;
  out = i &
  out = i || j;
  out = j
  out = j >> 3;
  out = -k;
  out = ~k;
  j = !k;
  out[0] = i
  out[0] = k
  if (i
  } else
    out = j;
  if (i
  out = i + j
}
```

Example 4
teststatic.cc,v 1.2 1994/10/25 15:40:33 chris Exp
This is an example of source code for nlc.

```
#ifdef __NLC__
#include "x4000.h"
#endif

void
#ifdef __NLC__
teststatic(bitvector ina, bitvector inb,
           bitvector incmd(21),
           bitvector &outa, bitvector &outb)
#else
teststatic(bitvector ina, bitvector inb,
           bitvector incmd,
           bitvector &outa, bitvector
  static bitvector temp(2);

  outa[0] = temp[0] ^ ina[1];
  for (i=1;i i++) {
    outa[i] = ina[i-1] ^ ina[i+1];
  }
  outa[15] = ina[14] ^ temp[1];
  temp[0] = ina[15];
  temp[1] = ina[0];
}
```

The Bitvector C++ Library

A partial implementation of the bitvector class for C++ is listed below, and also a test framework program:

```
bitvector.cc,v 1.1 1994/12/15 13:07:42 chris Exp
Bitvector class.

#include "bitvector.h"
#include

//
// onebit stuff
//

void
Add(onebit &x, onebit &y, onebit &r, onebit &c)
{
  if (x.value == X || y.value == X || c.value == X) {
    r.value = X;
    c.value = X;
```

```
    return;
  }
  int sum = (x.value - L) + (y.value - L) + (c.value - L);
  r.value = (sum & 1) ? H : L;
  c.value = (sum & 2) ? H : L;
}

onebit
onebit::operator ~ () return r
{
  switch (value) {
  case L:
    r.value = H;
    break;
  case H:
    r.value = L;
    break;
  default:
    r.value = X;
  }
}

//
// private methods
//

void
bitvector::adjust_size(size_t s)
{
  if (s > len) {
    if (fixed) {
      cerr << "ERROR: trying to grow a vector of fixed length
        (" << len << ")\n";
      return;
    }
#ifdef TRACE
    cerr << "Growing the vector from " << len <<" to "
         << s << "\n";
#endif
    size_t olen = len;
    onebit *obit = bit;
    len = s;
    bit = new onebit[len];
    for (size_t j = 0; j < olen; j++) {
      bit[j] = obit[j];
```

```
    }
    if (ref) {
      cerr << "WARNING: growing a reference vector.\n";
      ref = false;
    } else
      delete[] obit;
    if (def)
      while (j
  }
}

//
// element extraction
//

bitvector
bitvector::operator [] (size_t i) return r
{
  if (i >= len) {
    if (fixed) {
      cerr << "WARNING: accessing past the defined length
      (" << len << ") of the vector.\n";
      r.adjust_size(1);
      r.bit[0] = 0;
      r.def = true;
      r.fixed = true;
      return;
    } else
      adjust_size(i + 1);
  }
  r.len = 1;
  r.def = true;
  r.fixed = true;
  r.ref = true;
  r.bit = bit + i;
}

//
// assignments
//

bitvector&
bitvector::operator = (bitvector &v)
{
  if (v.len > len && !fixed) {
```

```
    size_t i = v.len;
    while (i > len)
      if (v.bit[--i] != 0) {
        adjust_size(i + 1);
        break;
      }
  }
  for (size_t i = 0; i < len && i < v.len; i++)
    bit[i] = v.bit[i];
  for ( ; i < len; i++)
    bit[i] = 0;
  def = true;
  return *this;
}

bitvector&
bitvector::operator = (int i)
{
  if (!fixed) {
    size_t max = 0;
    long temp = i;
    while (temp != 0) {
      temp = ((unsigned) temp) >> 1;
      max += 1;
    }
    if (max == 0)
      max = 1;
    adjust_size(max);
  }
  for (size_t j = 0; j < len; j++) {
    bit[j] = (i & 1);
    i = ((unsigned) i) >> 1;
  }
  def = true;
  return *this;
}

bitvector&
bitvector::operator = (char *s)
{
  onebit x, zero(0), one(1);
  size_t max = strlen(s);
  if (!fixed)
    adjust_size(max);
  for (size_t j = 0, i = max - 1; j < len; j++, i--) {
```

```
    if (j < max)
      switch(s[i]) {
      case '0':
        bit[j] = zero;
        break;
      case '1':
        bit[j] = one;
        break;
      default:
        bit[j] = x;
      }
    else
      bit[j] = zero;
  }
  def = true;
  return *this;
}

//
// arithmetic operations
//

bitvector
operator + (bitvector &x, bitvector &y) return r
{
  size_t max = (x.len > y.len) ? x.len : y.len;
  r.adjust_size(max + 1);
  onebit carry(0), zero(0);
  for (size_t i = 0; i < max; i++) {
    if (i >= x.len)
      Add(zero, y.bit[i], r.bit[i], carry);
    else if (i >= y.len)
      Add(x.bit[i], zero, r.bit[i], carry);
    else
      Add(x.bit[i], y.bit[i], r.bit[i], carry);
  }
  r.bit[max] = carry;
}

bitvector
operator + (bitvector &x, long y) return r
{
  size_t max = x.len > LONG_BIT ? x.len : LONG_BIT;
  r.adjust_size(max);
  onebit carry(0), zero(0), one(1);
```

```
  for (size_t i = 0; i < x.len; i++) {
    if (y & 1)
      Add(x.bit[i], one, r.bit[i], carry);
    else
      Add(x.bit[i], zero, r.bit[i], carry);
    y >>= 1;
  }
  for ( ; i <max; i++) {
    if (y & 1)
      Add(zero, one, r.bit[i], carry);
    else
      Add(zero, zero, r.bit[i], carry);
    y >>= 1;
  }
}

bitvector
operator + (long x, bitvector &y) return r(operator+(y, x))
{
}

bitvector
operator - (bitvector &x, bitvector &y) return r
{
  size_t max = (x.len > y.len) ? x.len : y.len;
  r.adjust_size(max + 1);
  onebit carry(1), zero(0), one(1);
  for (size_t i = 0; i < max; i++) {
    if (i >= x.len)
      Add(zero, ~y.bit[i], r.bit[i], carry);
    else if (i >= y.len)
      Add(x.bit[i], one, r.bit[i], carry);
    else {
      Add(x.bit[i], ~y.bit[i], r.bit[i], carry);
    }
  }
  r.bit[max] = carry;
}

bitvector
operator - (bitvector &x, long y) return r(operator+(x, -y))
{
}

bitvector
```

```
operator - (long x, bitvector &y) return r
{
  size_t max = y.len > LONG_BIT ? y.len : LONG_BIT;
  r.adjust_size(max);
  onebit carry(1), zero(0), one(1);
  for (size_t i = 0; i < y.len; i++) {
    if (x & 1)
      Add(one, ~y.bit[i], r.bit[i], carry);
    else
      Add(zero, ~y.bit[i], r.bit[i], carry);
    x >>= 1;
  }
  for ( ; i < max; i++) {
    if (x & 1)
      Add(one, one, r.bit[i], carry);
    else
      Add(zero, one, r.bit[i], carry);
    x >>= 1;
  }
}

bitvector
operator * (bitvector &x, bitvector &y) return r
{
  r = ((long) x) * ((long) y);
}

bitvector
operator * (bitvector &x, long y) return r
{
  r = ((long) x) * y;
}

bitvector
operator * (long x, bitvector &y) return r
{
  r = x * ((long) y);
}

//
// increments and decrements
//

bitvector&
bitvector::operator ++ ()
```

```
{
  onebit carry(1), zero(0);
  for (size_t i = 0; i < len; i++)
    Add(bit[i], zero, bit[i], carry);
  if (!fixed && carry) {
    adjust_size(len + 1);
    bit[i] = carry;
  }
  return *this;
}

bitvector
bitvector::operator ++ (int i) return r(*this)
{
  ++(*this);
}

bitvector&
bitvector::operator -- ()
{
  onebit carry(0), one(1);
  for (size_t i = 0; i < len; i++)
    Add(bit[i], one, bit[i], carry);
  return *this;
}

bitvector
bitvector::operator -- (int i) return r(*this)
{
  --(*this);
}

//
// comparisons
//

bool
operator == (bitvector &x, bitvector &&y)
{
  size_t max = (x.len > y.len) ? x.len : y.len;
  bool res = true;
  for (size_t i = 0; i < max; i++) {
    if (i >= x.len)
      res = res && y.bit[i] == 0;
    else if (i >= y.len)
```

```
      res = res && x.bit[i] == 0;
    else
      res = res && x.bit[i] == y.bit[i];
  }
  return res;
}

bool
operator == (bitvector &x, long y)
{
  bool res = true;
  for (size_t i = 0; i < x.len; i++) {
    res = res && x.bit[i] == (y & 1);
    y >>= 1;
  }
  if (y != 0 && !(y == -1 && x.len > 0 && x.bit
     [x.len - 1] -- 1))
    res = false;
  return res;
}

bool
operator < (bitvector &x, bitvector &y)
{
  size_t max = (x.len > y.len) ? x.len : y.len;
  for (size_t i = 0, j = max - 1; i < max; i++, j--) {
    if (j >= x.len) {
      if (y.bit[j] == 1)
        return true;
    } else if (j >= y.len) {
      if (x.bit[j] != 0)
        return false;
    } else {
      if (x.bit[j] == 1 && y.bit[j] == 0)
        return false;
      else if (x.bit[j] == 0 && y.bit[j] == 1)
        return true;
    }
  }
  return false;
}

bool
operator < (bitvector &x, long y)
{
```

```
  if ((((unsigned long) y) >> x.len) != 0)
    return true;
  for (size_t i = 0, j = x.len - 1; i < x.len; i++, j--)
  {
    bool b = (((unsigned long) y) >> j) & 1;
    if (x.bit[j] == 1 && !b)
      return false;
    else if (x.bit[j] == 0 && b)
      return true;
  }
  return false;
}

bool
operator < (long x, bitvector &y)
{
  if ((((unsigned long) x) >> y.len) != 0)
    return false;
  for (size_t i = 0, j = y.len - 1; i < y.len; i++, j--)
  {
    bool b = (((unsigned long) x) >> j) & 1;
    if (b && y.bit[j] == 0)
      return false;
    else if (!b && y.bit[j] == 1)
      return true;
  }
  return false;
}

//
// converters
//

bitvector::operator const long()
{
  long r = 0;

  for (size_t i = 0; i < len && i < LONG_BIT; i++)
    if (bit[i])
      r |= (1 << i);
  return r;
}

//
// IOs
```

```
//

ostream&
operator << (ostream& s, const bitvector& x)
{
  if (s.opfx())
    x.printon(s);
  return s;
}

void
bitvector::printon(ostream& s) const
// FIXME:  Does not respect s.width()!
{
  register streambuf* sb = s.rdbuf();
  int i;

  for (i = len -1; i > = 0; i--)
  {
    sb->sputc(bit[i].GetState());
  }
}
```

test.cc,v 1.1 1994/12/15 13:07:47 chris Exp
Test framework.

```
#include "bitvector.h"

int
main(int argc, char *argv[])
{
  bitvector a(10), b(5);
  bitvector c, d;
  long l;

  a = 0; b = 9;
  cout << "a = 0 (" << a << "); b = 9 (" << b <<")\n";
  a = 9; b = 0;
  cout << "a = 9 (" << a << "); b = 0 (" << b <<")\n";
  c = a;
  l = (long) c;
  cout << "c = a [= 9] (" << c << "); l = (long) c = " << l << "\n";
  l += 8;
  c = l; d = l;
  cout << "c = l [=17] (" << c << "); d = l [=17] (" << d << ")\n";
```

```
b = d;
c = a + b;
cout << "a = " << a << " (9); b = " << b << " (17);
a + b = " << c<< " \n";
cout << "a + b = " << a + b << " (9 + 17)\n";
cout << "a + b = " << a + b << " (17 + 9)\n";
cout << "a + 5 = " << a + 5 <<" (9 + 5)\n";
cout << "a + 9 = " << a + 9 <<" (9 + 9)\n";
cout << "a + 20 = " << a + 20 << " (9 + 20)\n";
cout << "5 + a = " << 5 + a << " (5 + 9)\n";
cout << "9 + a = " << 9 + a << " (9 + 9)\n";
cout << "20 + a = " << 20 + a << " (20 + 9)\n";
cout << "a - b = " << a - b << " (9 - 17)\n";
cout << "b - a = " << b - a << " (17 - 9)\n";
cout << "a - 5 = " << a - 5 << " (9 - 5)\n";
cout << "a - 9 = " << a - 9 << " (9 - 9)\n";
cout << "a - 20 = " << a - 20 << " (9 - 20)\n";
cout << "5 - a = " <<5 - a << " (5 - 9)\n";
cout << "9 - a = " <<9 - a << " (9 - 9)\n";
cout << "20 - a = " <<20 - a << " (20 - 9)\n";
cout << "a * b = " << a * b << " (9 * 17)\n";
cout << "b * a = " << b * a << " (17 * 19)\n";
cout << "a * 5 = " << a * 5 << " (9 * 5)\n";
cout << "9 * a = " << 9 * a << " (9 * 9)\n";
if (a < c)
  cout << "a (" << a << ") < c (" << c << ")\n";
if (a < c)
  cout << "a (" << a << ") > c (" << c << ")\n";
if (a <= c)
  cout << "a (" << a << ") <= c (" << c << ")\n";
if (a >= c)
  cout << "a (" << a << ") >= c (" << c << ")\n";
if (a == c)
  cout << "a (" << a << ") == c (" << c << ")\n";
if (a != c)
  cout << "a (" << a << ") != c (" << c << ")\n";
if (a < 10)
  cout << "a (" << a << ") < 10\n";
if (a > 8)
  cout << "a (" << a << ") > 8\n";
if (a <= 9)
  cout << "a (" << a << ") <= 9\n";
if (a >= 9)
  cout << "a (" << a << ") >= 9\n";
if (a == 9)
```

```
    cout << "a (" << a << ") == 9\n";
  if (a != 9)
    cout << "a (" << a << ") != 9\n";
  if (10 < a)
    cout << "10 < a (" << a << ")\n";
  if (9 < a)
    cout << "9 < a (" << a << ")\n";
  if (8 < a)
  " cout << "8 < a (" << a << ")\n";
}
```

Limitations of nlc

Some things like multiply and divide are unimplemented yet. The mechanism for the library description is not general enough yet.

The compiler should handle more cases of undefined operators and more cases of addition and comparison operators. For example, it can only handle add operators with carry in and carry out, however, most hard macros don't have carry in and/or carry out.

The generated netlist could be better optimized.

The bitvector C++ class isn't complete yet...

The documentation is being written...

Help on XNF to Verilog and Viewing Verilog

XNF to Verilog / View Verilog

XNF to Verilog

If you want to generate Verilog netlists from XNF netlists, then at the following screen:

...you must replace the example of the format with the name of your file. **N.B.** You must have the file extension *.xnf*.

To check you have the correct format of file name, the following message box appears:

If you have used the correct format, click on *Yes*, if not, then click on *No* and you can change the file name as required.

Two xterm windows will appear which contain calls to two procedures (*xnf2edif* and *edif2verilog*), to generate the Verilog netlists.

View Verilog

If you want to view the Verilog netlists and the schematic, then at the following screen:

...you must replace the example of the format with the name of your file. **N.B.** You must have the file extension *.net*.

To check you have the correct format of file name, a message box appears, similar to the one above.

If you have used the correct format, click on *Yes*, if not, then click on *No* and you can change the file name as required.

A window will appear listing the Verilog netlists, and a built-in Cadence form entitled *Verilog In*. You must fill in the appropriate information and it will bring up the schematic view in a graphics window.

Bibliography

[1] Adidev Jain Daniel C. Liebisch. Jessi common framework design management - the means to configuration and execution of the design process. *Jessi Common Framework*, pages 553 – 557, 1992.

[2] Cadence Online Documentation. *SKILL Language User Guide.* Cadence Online Library, 9401 edition.

[3] Pieter van der Wolf Olav ten Bosch, Peter Bingley. Design flow management in the nelsis ic design system. November 1992.

[4] Todd J. Scallan. Cad framework initiative - a user perspective. *CAD Framework Initiative*, (40.2):672 – 675, 1992. 29th ACM/IEEE Design Automation Conference.

[5] A. Richard Newton Rick L. Spickelmeir Timothy J. Barnes, David Harrison. *Electronic CAD Frameworks.* Kluwer Academic Publishers, P.O. Box 17, 3300 AA Dordrecht, The Netherlands, 1992.

[6] Pieter van der Wolf. *CAD Frameworks - Principles and Architecture.* Kluwer Academic Publishers, P.O. Box 322, 3300 AH Dordrecht, The Netherlands, 1994.

NAC - *Nlc & Cadence*

User Manual

Contents

Before You Start

About This Manual This manual is for all users of the Design Flow. It can be accessed online from NAC via the *System Help* button on the banner of the Flow Browser window that it appears in.

Introducing the Design Flow

Overview The NAC - *Nlc & Cadence* system is presented in a flowchart which guides you through the series of steps. It enables you to, first create a new directory, then edit an existing C++ file or create a new one. It goes on to generate XNF netlists from a C++ program and convert them to Verilog netlists. You can then view the Verilog netlists and the schematic, and finally call the Xilinx Design Editor to further implement the logical design.

The Steps in the Manual See figure 0.1 for a flowchart of the steps in the manual.

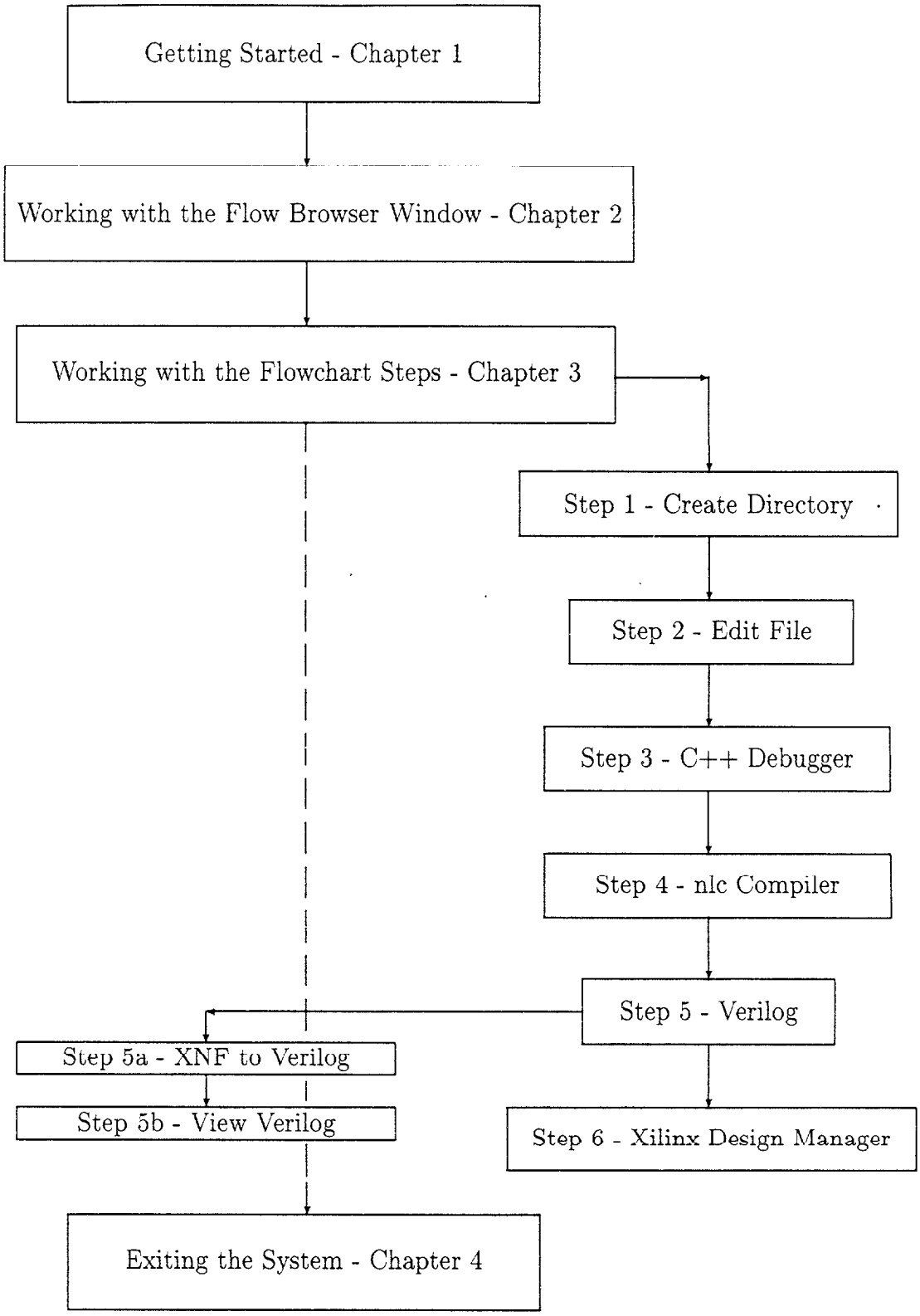


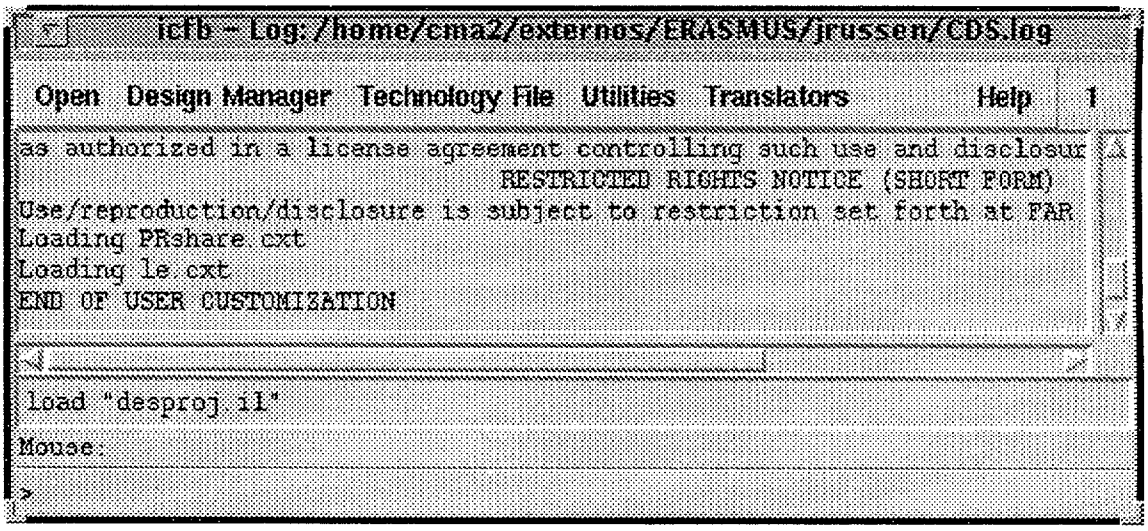
Figure 0.1: Flowchart of the Steps in this Manual

Chapter 1

Getting Started

At the Command Interpreter Window (CIW) command line, type

```
load "desproj.il"
```



The NAC system will appear in a Flow Browser Window as in figure 1.1.

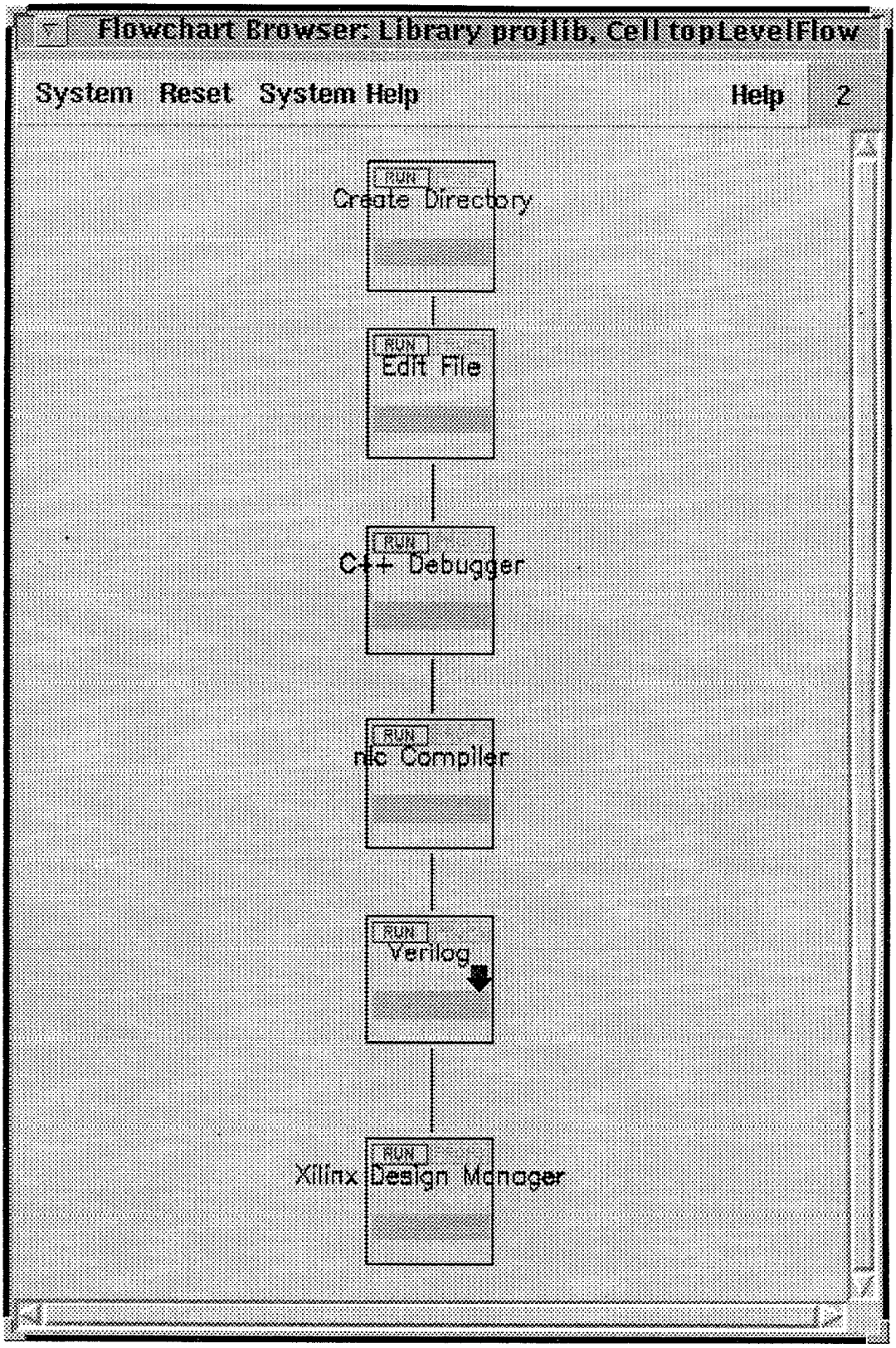


Figure 1.1: The NAC system flowchart in a Flow Browser window

Chapter 2

Working with the Flow Browser Window

2.1 Banner Buttons

The Flow Browser window has 4 buttons on the banner. Two are automatically created with a Flow Browser window and two have been created for this design flow (see figure 1.1).

(1) System - the *System* pull-down menu is automatically created and contains 4 menu items:

(i) Save Flowchart - encrypts and saves the information associated with the displayed flowchart in the Flow Browser window as a cellview in the design library. The cell name is the name of the displayed flowchart, and the view name is *flowchartInst.*

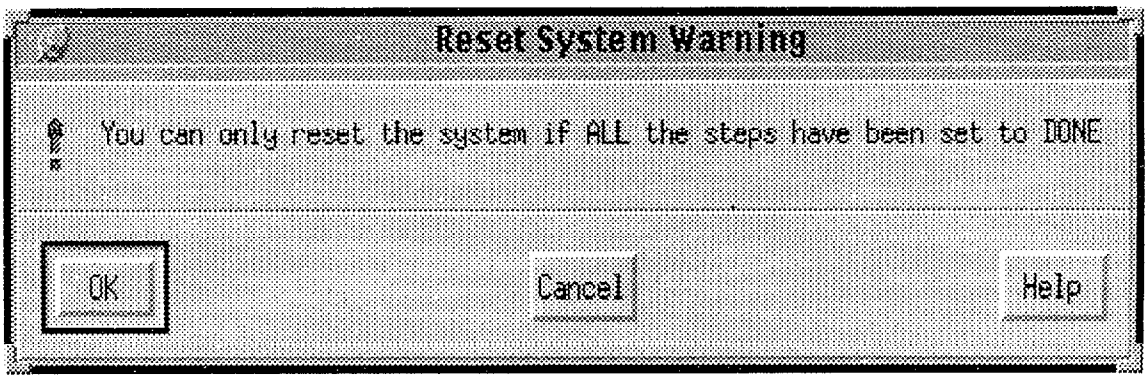
(ii) Switch to New Design - displays another flowchart instance you specify from the same design library or from a different design library in the same Flow Browser window. This command displays a form and the Library Browser to help you select a new design. If you do not enter any information into this form, the flowchart instance that is curently displayed is redisplayed.

(iii) Redraw - redisplays the flowchart that is already displayed in the Flow Browser window.

(iv) Close Window - saves the displayed flowchart and closes the Flow Browser window.

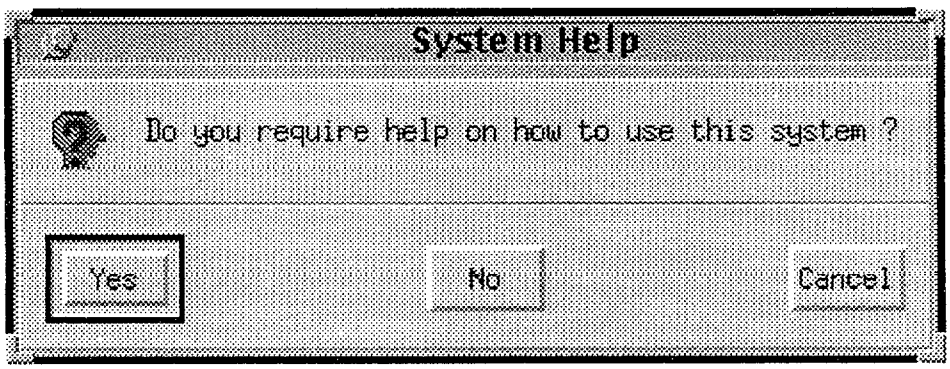
(2) Help - the *Help* button provides by the Cadence environment informs the user to call "openbook", the Cadence online documentation.

(3) Reset - contains one menu item to **Reset All Steps** of the design flow. When you click on this, it first displays a warning dialog box to inform the user the only if all the steps have been set to *Done* (ie, by each step being correctly finished, or **Bypassed**), can then whole system be reset.



Click on *Yes* to reset all the steps (including those in the subflowchart). Click on *No* or *Cancel* to stop the system from being reset.

(4) System Help - contains one menu item, *System Help*. When you click on this, it first displays a question dialog box to ask you if you require help on this system.



Click on *Yes* if you do and this manual in its online form will appear via the html browser.

2.2 Pop-Up Menu

Each design step contains a Pop-Up Menu which is activated by clicking and holding the middle mouse button over the design step. By default this menu has two commands, *Run Step* and *Reset Step*. All design steps that are created have these two commands associated with them. Some of the steps in this system have other buttons added to the pop-up menu. Each of the six primary steps in the design flow have an extra command called *Bypass Step*, and the "Verilog" design step also has a *Push to Subflowchart* command.

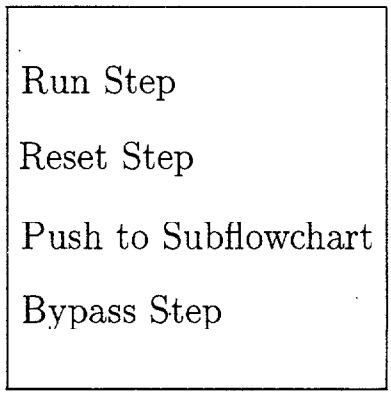


Figure 2.1: Pop-Up Menu on a Design Step

(1) Run Step - runs the design step. Click and hold the middle mouse button in the design step box and place the cursor over *Run Step* in the pop-up menu to run the design step.

(2) Reset Step - resets the design step. Click and hold the middle mouse button in the design step box and place the cursor over *Reset Step* in the pop-up menu to reset the design step.

(3) Push to Subflowchart - displays a subflowchart referenced by the design step. Click and hold the middle mouse button in the design step box and place the cursor over *Push to Subflowchart* in the pop-up menu to display the subflowchart. This action can only be invoked if the design step references a subflowchart.

(4) Bypass Step - bypasses the design step, ie sets the step to *Done* without having to first run the step. Click and hold the middle mouse button in the design step box and place the cursor over *Bypass Step* in the pop-up menu to bypass the design step.

2.3 Design Step Commands

At the top of each design step there are two buttons, *Run* and *Props* . On a design step which has a subflowchart attached, there is a downward arrow on the design step.

(1) Run - runs the design step. Place the cursor over *Run* in the design step and click right with the mouse to run the design step (see figure 1.1).

(2) Props - displays the run-time properties of the design step. Place the cursor over *Props* in the design step and click right with the mouse to run to display the run-time properties. The properties are displayed in a text window (see figure 1.1).

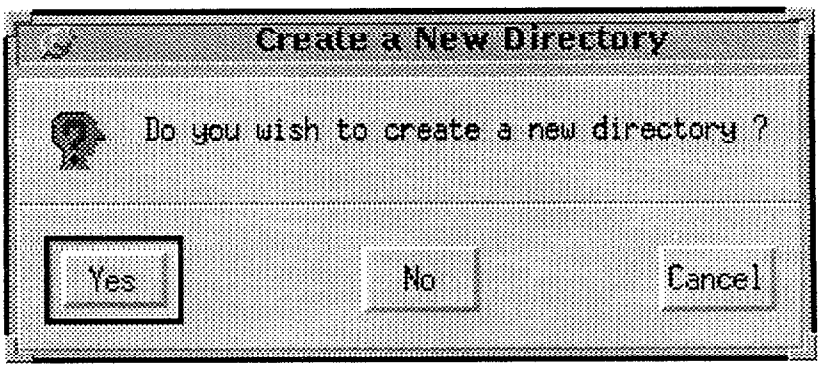
(3) Down Arrow - displays a subflowchart referenced by the design step. Place the cursor over down arrow in the design step and click right with the mouse to display the subflowchart (see figure 1.1).

Chapter 3

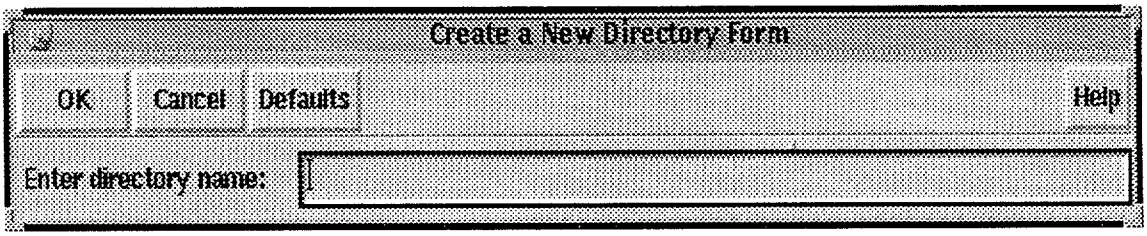
Working with the Flowchart Steps

3.1 Step 1 - Create Directory

When you run this step a question dialog box appears asking you if you wish to create a new directory .



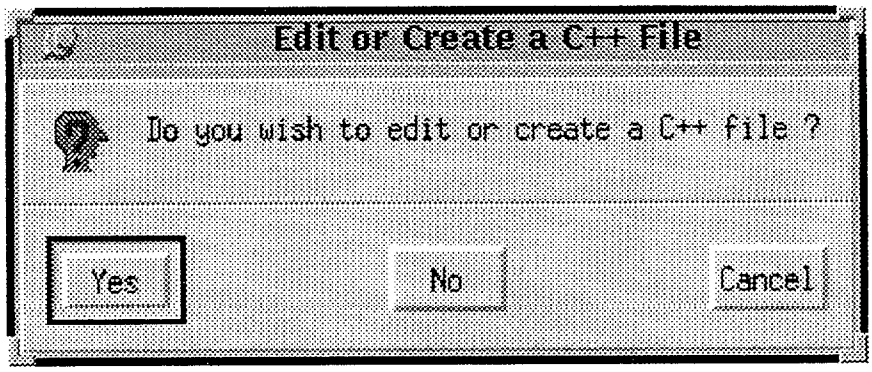
Click on *Yes* if you do, and a form will appear for you to input the name of the directory to be created.



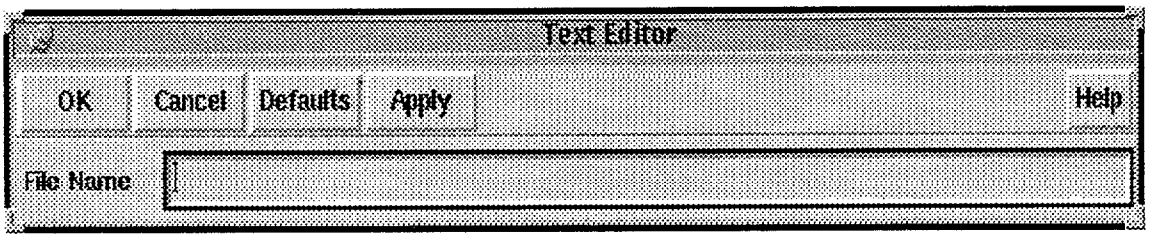
Enter the name of the new directory and click on *OK*. The directory will be created and the design step is set to *Done*.

3.2 Step 2 - Edit File

When you run this step a question dialog box appears asking you if you wish to edit or create a C++ file.

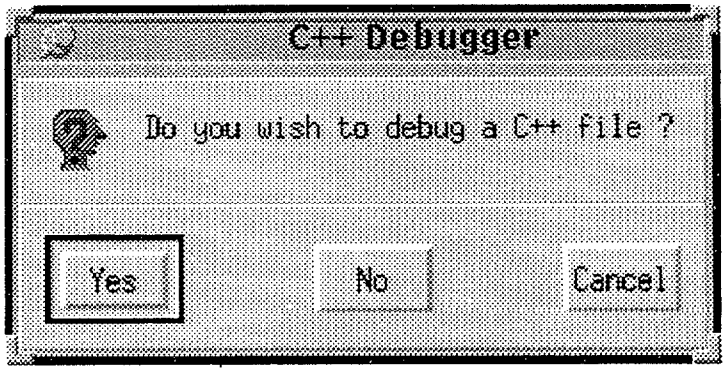


Click on *Yes* if you do, and a form will appear for you to enter the name of the file. Enter the name of the file and click on *OK*. The file will appear in a text window with the text editor. The design step sets itself to *Done*.

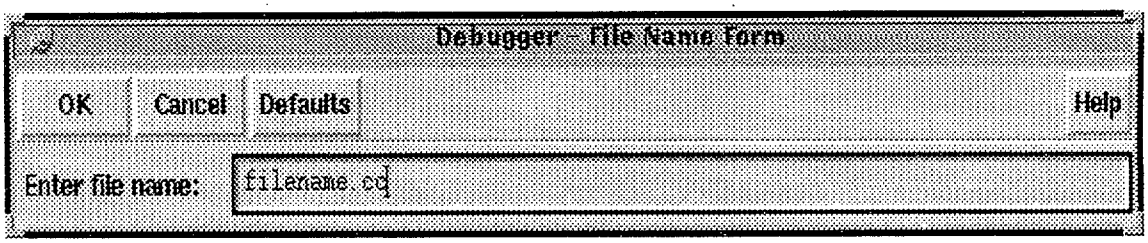


3.3 Step 3 - C++ Debugger

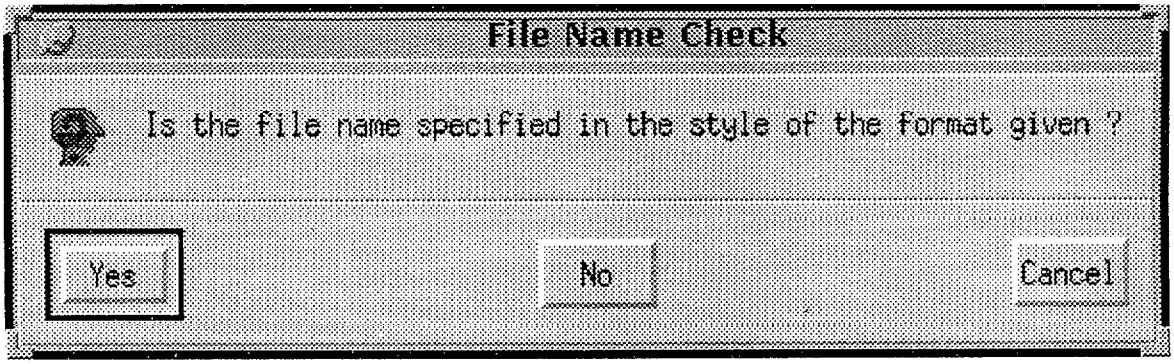
When you run this step a question dialog box appears asking you if you wish to debug a C++ file.



Click on *Yes* if you do, and a form will appear for you to enter the name of the file.



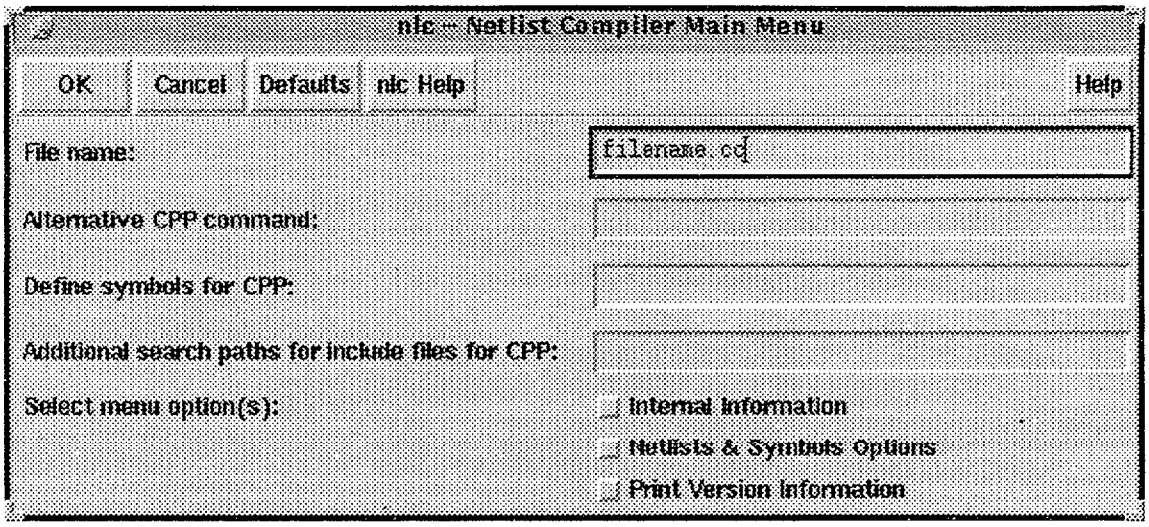
When you enter the name of the file a dialog box appears to check you have given a file name in the specified format.



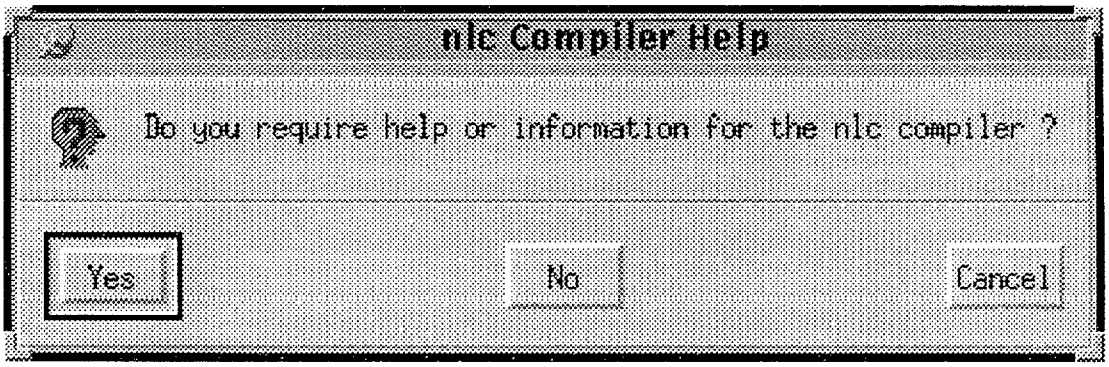
If you have, then click on *Yes*, if not then click on *No* and you will be returned to the file name input line. When you have entered a file name in the correct format, click on *OK* and the debugger will be invoked. The design step sets itself to *Done*.

3.4 Step 4 - nlc Compiler

When you run this step, the "nlc Netlist Compiler Main Menu" appears.



For online help on this step, click on the *nlc Help* button on the banner of the form and the following dialog box appears



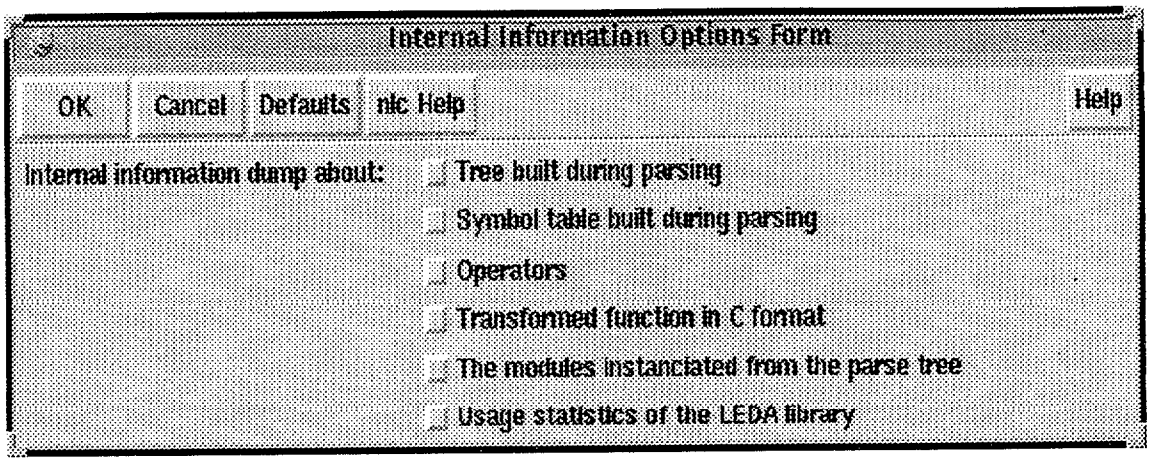
Please note that the following nlc compiler options may be used in any combination.

- **File name:**
  - You must enter the name of the C (or C++) source file you wish to compile (the format of the file name is given)
  - The following message box will then appear:
  - If you have specified the file name in the correct format, click on *Yes*, if not, then click on *No* and you can retype the file name.

N.B. If you want to view the version information only, then a file name is not necessary, so simply delete the file name format and click *No* on the message box (and select the *Print Version Information option*).

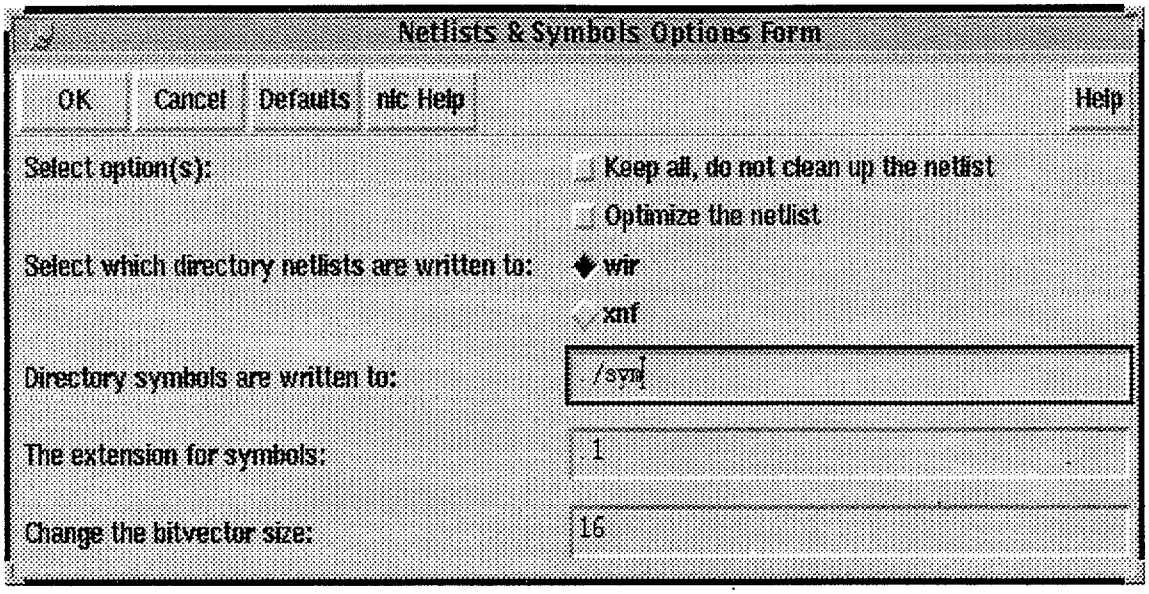
- **Alternative CPP command:**(the default is "gcc -E -x c++ -D_NLC _")
  - If you select this option you can specify another C++ compiler
- **Define symbols for CPP:**
  - If you select this option, you can specify symbols
- **Additional search paths for include files for CPP:**
  - If you select this option, you can enter the search path of any include files necessary for your program
- **Select menu option(s):**
  - All, one, two or none of these options can be selected

Internal Information Options Form



If you select this option from the main menu, you can then select as as many or as few of the options that appear on the form.

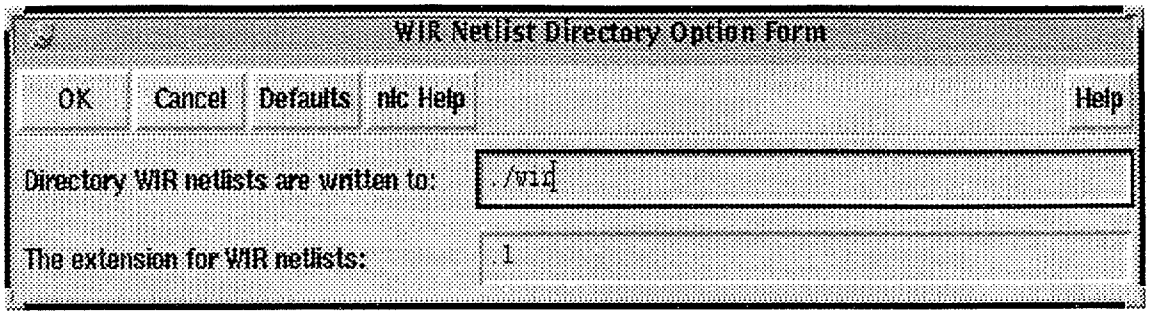
Netlist & Symbols Options Form



If you select this option from the main menu, you can then select as many or as few of the options that appear on the form.

- **Select option(s):**
  - You can select one, both, or neither of these options
- **Select which directory netlists are written to:**
  - You can select one of these. If you select *xnf* then you cannot select *wir*, and vice versa.
- **Directory symbols are written to:** (the default is already given)
  - If you want to write to stdout, replace the default with a dash (-)

  N.B. Ensure the name of the directory is three characters long, as in the default shown.
- **The extension for symbols:** (the default is already given)
  - If you do not want an extension, replace the default with a dash (-)
- **Change the bitvector size:** (the default is already given)
  - This may be changed to the size you require

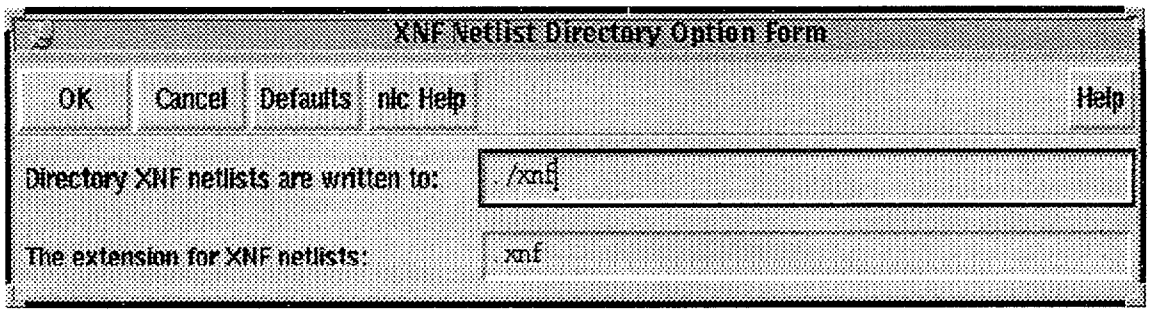


WIR Directory Option Form

- **Directory WIR netlists are written to:** (the default is already given)
  - If you want to write to stdout, replace the default with a dash (-)

  N.B. Ensure the name of the directory is three characters long, as in the default shown.
- **The extension for netlists:** (the default is already given)
  - If you do not want an extension, replace the default with a dash (-)

XNF Directory Option Form

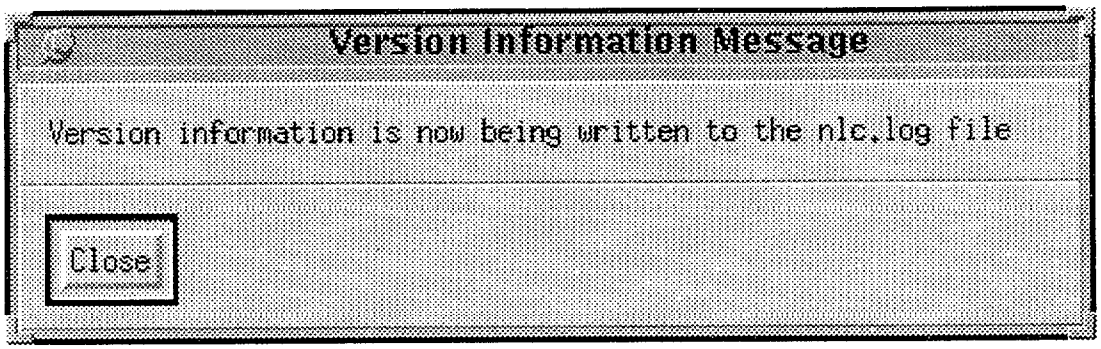


- **Directory XNF netlists are written to:** (the default is already given)
  - If you want to write to stdout, replace the default with a dash (-)

  N.B. Ensure the name of the directory is three characters long, as in the default shown.
- **The extension for netlists:** (the default is already given)

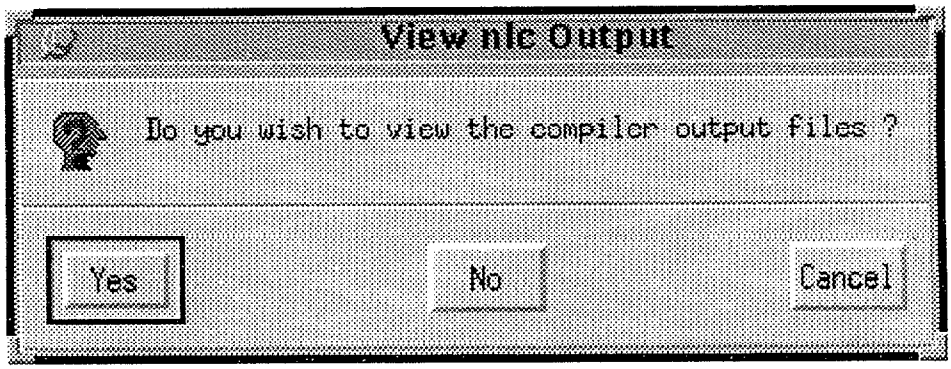
– If you do not want an extension, replace the default with a dash (-)

Version Information Message

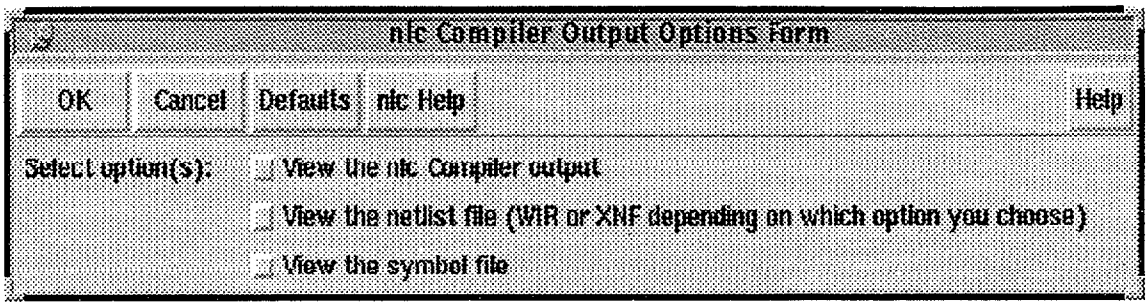


If you select this option only, a file name is not necessary as it simply prints the version of nlc.

nlc Compiler Output Options Form



If you wish to view the nlc compiler output files, then click on *Yes* and the following screen will appear:



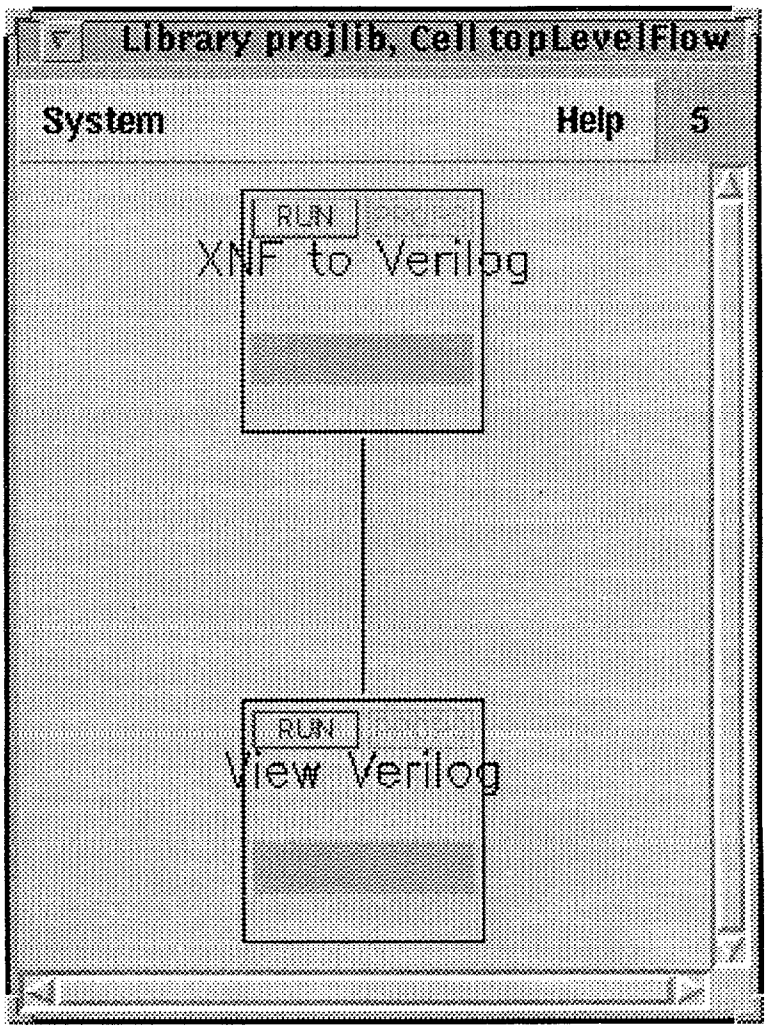
You can choose to view one, two, all, or none of the output files. **N.B.** You can only view the netlist and symbol files if the directories have been specified in the *Netlist & Symbols Options Form*, otherwise the following warning messages will appear:

N.B. If you have specified that the netlist and/or symbol directories are to be written to the *stdout*, then then output can be viewed via the command tool window.

Once this step is finished, it is set to *Done*.

3.5 Step 5 - Verilog

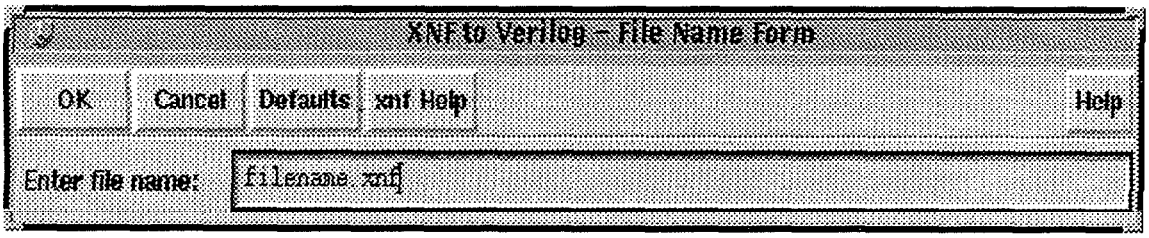
By clicking on the *Push to Subflowchart* button on the pop-up menu, or clicking on the down arrow, the "Verilog" subflowchart appears



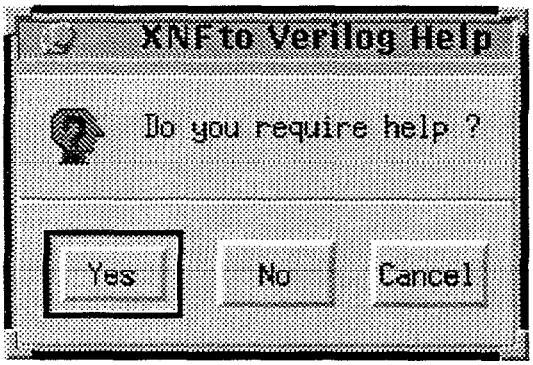
3.5.1 Step 5a - XNF to Verilog

When you run this step a question dialog box appears asking you if you wish to generate Verilog netlists from XNF netlists.

Click on *Yes* if you do, and a form will appear for you to enter the name of the file.



Click on *verilog Help* for online help and the following dialog box will appear

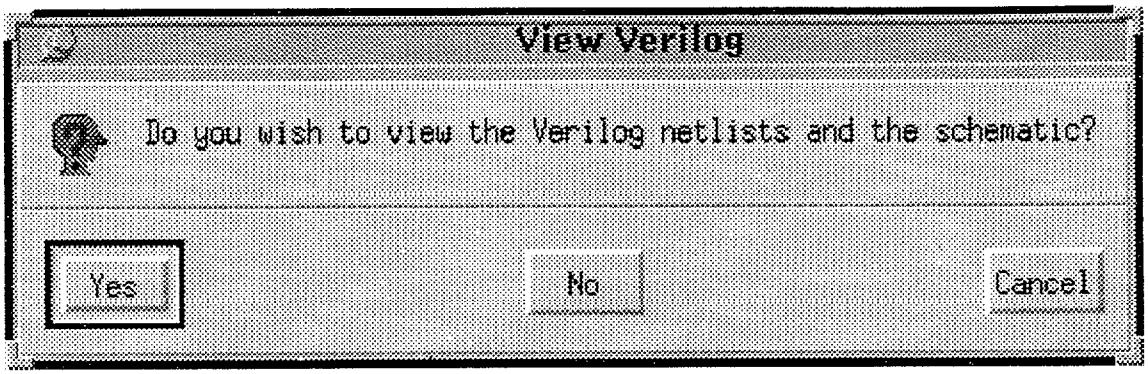


Enter the name of the file and click on *OK* and the XNF netlists are translated to Verilog netlists, via EDIF.

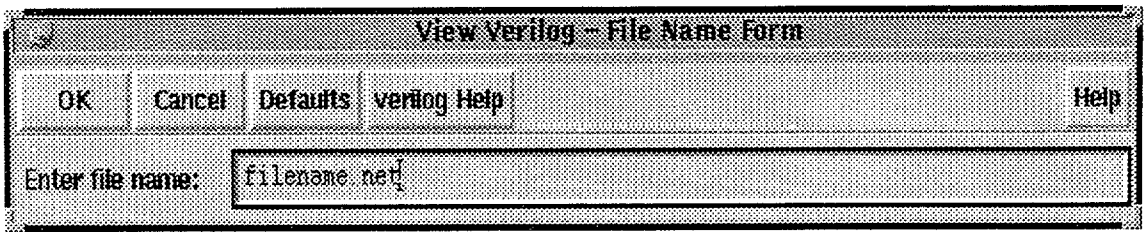
This step on the design flow is step to *Done* when completed.

3.5.2 Step 5b - View Verilog

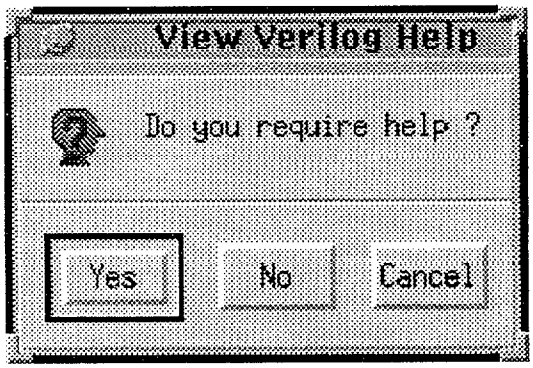
When you run this step a question dialog box appears asking you if you wish to view the Verilog netlists and the schematic.



Click on *Yes* if you do, and a form will appear for you to enter the name of the file.



Click on *verilog Help* for online help and the following dialog box will appear asking you if you require help

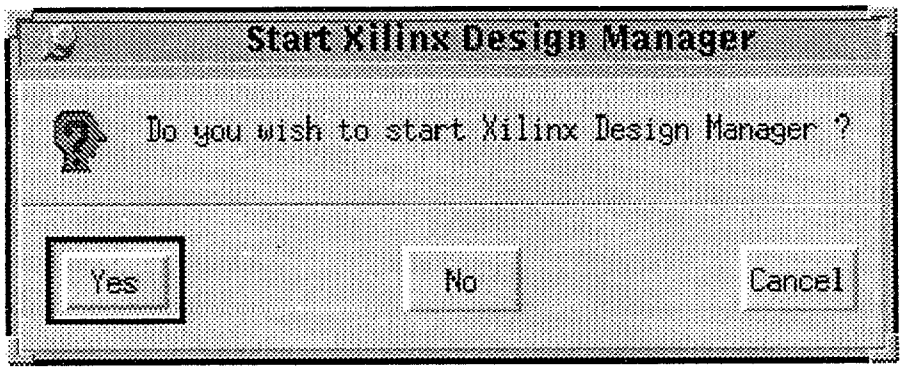


Enter the name of the file and click on *OK* and a view file with the netlists will appear and also the Cadence "Verilog In" form.

This step on the design flow is step to *Done* when completed.

3.6 Step 6 - Xilinx Design Manager

When you run this step a question dialog box appears asking you if you wish to start the Xilinx Design Manager.



Click on *Yes* if you do, and the Xilinx Design Manager will be invoked.

This step on the design flow is step to *Done* when completed.

Chapter 4

Exiting the System

Once you are finished using NAC, you can either *Reset All Steps* from the banner button to ready the system for the next time, or simply type

```
exit
```

at the CIW command line.