INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the
text directly from the original or copy submitted. Thus, some thesis and
dissertation copies are in typewriter face, while others may be from any type of
computer printer.

The quality of this reproduction is dependent upon the quality of the copy
submitted. Broken or indistinct print, colored or poor quality illustrations and
photographs, print bleedthrough, substandard margins, and improper alignment
can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and
there are missing pages, these will be noted. Also, if unauthorized copyright
material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning
the original, beginning at the upper left-hand corner and continuing from left to
right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced
xerographically in this copy. Higher quality 6" x 9" black and white photographic
prints are available for any photographs or illustrations appearing in this copy for
an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeib Road, Ann Arbor, MI 48106-1346 USA

UMI®
800-521-0600
RICE UNIVERSITY

EDIF Netlist Optimization of Pipelined Designs

by

Vasileios Balabanos

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

John K. Bennett, Associate Professor, Chair
Electrical and Computer Engineering

Joseph R. Cavallaro, Associate Professor
Electrical and Computer Engineering

J. Robert Kemp, Professor
Electrical and Computer Engineering

Keith D. Cooper Associate Professor
Computer Science

Houston, Texas
April, 2000
To my parents Sifis and Soula
and my brother Stelios
ABSTRACT

EDIF Netlist Optimization of Pipelined Designs

by

Vasileios Balabanos

This thesis describes the design, implementation, and evaluation of a software system for optimizing synthesized logic circuits. The particular implementation described is targeted to the Xilinx Virtex family of FPGAs, but the techniques developed are relevant to other families of array-based semi-custom programmable logic circuits. One of the unique aspects of my approach is that the optimization occurs after the circuit is mapped onto the logic array. Prior to this work it was commonly believed that optimization after mapping was infeasible. The advantages of this approach include the ability to optimize a design without having the VHDL source code, the opportunity to selectively optimize only parts of a circuit and the preservation of the original the state encoding. The optimizations are also transparent to the synthesis process. This is a powerful and versatile method, which gives the designer considerable freedom in optimizing parts of the design according to his or her preferences.

The optimization process proceeds as follows. The behavioral or structural description of the design is first written in VHDL. The design is then synthesized using the Workview Office synthesis tool and extracted to an EDIF (Electronic Design Interface Format) mapped netlist targeting Xilinx’s Virtex family of FPGAs. This netlist is then analyzed, and an internal representation of the given circuit is created. Any pipelines (blocks of combinational logic feeding one or more registers) that exist in the circuit are then identified and common blocks of logic that reside between the pipeline registers are extracted. Multilevel minimization algorithms in the SIS framework are applied in order to optimize the design. The optimized equations are then converted to an EDIF-compatible format and all the necessary modifications are computed in order to restructure the original netlist to produce the optimized one. The resultant remapped circuit is then placed and routed as before.
ACKNOWLEDGEMENTS

I would like to thank my advisor John K. Bennett for his indispensable help during the completion of this project. His remarks were always beneficial and our collaboration has always been enjoyable. Many thanks to the following professors: Joseph R. Cavallaro, J. Robert Jump and Keith D. Cooper for reviewing my report and for providing me with their useful feedback.

Special thanks also go to my colleagues Hazim Shafi, Demetrios Demopoulos, and Damian Dobric for reviewing my report and for their constructive comments.

I would also like to thank Amanda Gunn, Penny Wilson, Deanne Swetnam and Amanda Spurlin for their continuous inspiration. Their encouragement, patience and support were instrumental to the development of the project.


Finally, and most of all, I would like to thank my parents, for everything.
Table of Contents

1.0 Introduction

1.1 Computer Aided Design Synthesis

1.2 Logic Design Styles

1.3 SRAM-based Field Programmable Gate Arrays – Virtex Family

1.4 Hardware Description Languages

1.5 Circuit Models

1.6 EDIF Format

1.7 Methodology

2.0 Logic Optimization

2.1 Two-Level Optimization

2.2 Multilevel Optimization

3.0 Description of the Implementation

3.1 General Description

3.2 Parsing and Constructing the Network Logic Model

3.3 Pipeline Identification and Multilevel Equations Extraction

3.4 SIS Optimization

3.5 Netlist Modification

3.6 Creating the New EDIF Netlist

4.0 Results

4.1 State Machines Example

4.2 Huffman Encoding Example

5.0 Discussion

6.0 References
1.0 Introduction

Modern design of digital circuits is based on high-level descriptions using a hardware description language (HDL) such as VHDL. The designer usually describes his or her design in an HDL and after the source code is compiled, a set of logic gates and their interconnections is obtained, usually in the form of a netlist. The next step is to apply circuit optimization techniques. The now optimized logic is then mapped onto a specific device architecture before it is placed and routed.

Our approach to optimization differs significantly from this traditional design methodology in that the logic to be optimized is extracted directly from the mapped logic. Although it was believed previously that this approach was infeasible, we present in this work an efficient way of optimizing the mapped netlist for pipelined designs. We are careful to optimize only the logic that resides between pipeline registers so that we do not change the functionality of the original design. The actual optimization employs multilevel optimization techniques. After optimization, the resulting circuit is restructured and remapped, and then written to another EDIF file.

One obvious benefit of this approach is that the original source code of the design is not needed and we may therefore optimize a netlist without having knowledge of the circuit that is being represented. We may also selectively optimize parts of the design. This capability is useful when a design has been partitioned onto multiple devices or when we need to optimize different parts of the design in different ways. Since this approach
maintains the original state encoding, the optimization process is totally transparent to
the synthesis process.

We continue our discussion by presenting some background information that will
facilitate the understanding of our implementation.

1.1 Computer Aided Design Synthesis

Continuing advances in lithography have led to finer devices and to increasingly
integrated and complex circuits also known as Very Large Scale Integration (VLSI)
circuits. Although the requirements for finer precision and for reduction of the
manufacturing defects lead to increased manufacturing costs for VLSI circuits, they are
highly desirable for many different reasons.

First of all, VLSI circuits have definitely opened new markets (e.g. cellular phones,
notebooks, palmtops). Second, since the fabrication of VLSI circuits relies on the process
of replication, the cost is amortized over the volume of production. In addition, fewer
components per system are required, reducing the cost of packaging and assembly. Third,
higher integration results in higher speeds since the parasitic capacitance is reduced and
fewer interconnections are needed. Finally, the reliability of the system is highly
increased due to the fewer components.

The profitability of microelectronic circuits greatly depends on the time-to-market since
every delay will increase the design costs and reduce the revenue life of the product. The
revenue life of a product (that is the time during which the product generates most of its revenues) may be less than a year [9], which makes it critical to aim at reducing its design time. Computer Aided Design (CAD) tools have proved indispensable to modern circuit design since they can help automate most of the time-consuming design tasks and reduce the design turnaround time. They can help manage the increased complexity of VLSI circuits in verification tasks, the generation of the masks, automatic placing and routing and logic synthesis and optimization. CAD tools have also helped eliminate errors that are unavoidable for large-scale circuits during the design phase. Although CAD tools may impose certain restrictions on the design style, they can be of great help during the optimization of a circuit, since they enable the designer to explore more of the design space and use the optimal of many different implementation choices.

Synthesis is the process of creating more detailed circuit models such as a geometric layout, which has all the information needed for fabrication, from more abstract circuit models. Three different levels of abstraction may be considered: architectural, logic and geometrical and each level may be viewed in three different ways: behavioral, structural and physical. In the behavioral view, only the function of the circuit is described. The structural view describes how the different components of a circuit are interconnected. The physical view allows direct access to the physical components of the circuit. CAD tools may be used for either architectural synthesis or logic synthesis. Architectural synthesis transforms the circuit's view at the architectural level from behavioral to structural. Starting from a set of desired functions, we identify resources and their interconnections that are able to carry out those functions. In logic synthesis the
behavioral description of a circuit at the logic level, which may be a state diagram, is transformed into a structural description, which may be a set of logic gates and their interconnections (gate-level netlist). The process of associating the logic primitives resulting from the logic synthesis to the available library components is called library binding or technology mapping.

Immediately following the synthesis process, the optimization process may have different objectives such as the minimization of area or the maximization of performance. At the architectural level for example, an appropriate scheduling of the required operations will result in better performance, while resource sharing will reduce the circuit's required area. At the logical level two-level logic implementations (i.e. PLAs) are subject to two-level optimization techniques, while for multiple-level networks of logic gates heuristic methods have been used to reduce the area or the delay of the circuit.
1.2 Logic Design Styles

Depending on what kind of market a circuit is designed for, different design methodologies or styles may be used. For example for a general purpose circuit such as a microprocessor we are going to use a style that offers high performance and allows for flexible designs no matter what the final cost will be since it will be amortized over the production volume. For special application circuits that perform a limited set of tasks (ASICs) though, we may need shorter design times and lower cost products.

Two main approaches exist for designing VLSI circuits: custom and semi-custom. In the custom design style, the functional description and the physical design layout are all made by hand and the designer has complete freedom to tune the design to meet special design criteria. In the semi-custom design style, the circuits are designed by using a set of predefined sub-circuits. Therefore, with the semi-custom design methodologies, the designer has to choose from a set of well-designed sub-circuits and to define the interconnections among them. CAD tools may be successfully employed since the design space is now a set of discrete choices and the design time is significantly reduced. Today almost all the designs are of the semi-custom design style, and only specific parts of a circuit are designed using custom methodologies such as the floating-point unit of a microprocessor where specific performance criteria must be met.

Semi-custom design methodologies are further subdivided into cell-based and array-based designs. Cell-based designs do not offer any advantages over custom designs as far as the manufacturing time is concerned. However, the designer may choose from either a
predefined library of cells (standard-cell designs), or use cell-generator programs that will create the needed cells by their functional description (macro-cell designs) and the design time is greatly reduced. Standard-cell designs restrict the designer to the components of the library; however, these designs are usually more compact than the macro-cell designs because the placing and routing of the cells is more efficiently done. Both design styles may be combined with custom-designed sub-circuits to produce high performance designs. Array-based designs reduce both the manufacturing and the design time. The circuits are designed using an array of pre-diffused cells. Those cells are generic and may be programmed and interconnected either by metalization during the manufacturing phase (mask-programmable gate arrays or MPGAs) or by programming antifuses or memory configuration elements (field-programmable gate arrays or FPGAs). The designer has to use the cells that the array offers, however both the design and manufacturing times are reduced and the placing and routing processes are fully automated by CAD tools. In this project we are concerned with FPGA based designs and more specifically SRAM-based FPGAs which are described in the following section.

1.3 SRAM Programmable Field Programmable Gate Arrays – Virtex Family

FPGAs consist basically of an array of logic blocks, which are interconnected through programmable routing resources and are therefore useful in implementing general-purpose multilevel logic [7] (Figure 1.1).
Routing channels connect logic blocks to each other or to input/output blocks (IOBs) via programmable elements. More specifically, five-transistor SRAM cells are used to control the state of pass transistors. SRAM programmable FPGAs therefore contain a certain number of configuration memory cells, which are externally loaded with configuration data during power-up. SRAM-based FPGAs generally have larger logic blocks (than antifuse-based FPGAs) that are capable of implementing more complex functions with fewer routing wires and programmable interconnections [6,10]. This mainly happens to minimize the routing delay due to the high resistance and capacitance of the interconnection elements.

The main drawbacks of SRAM-based FPGAs are the requirement for external memory in order to store the configuration data, and the need to reprogram them each time the power is applied. However, this is also one of their advantages, that is, reprogrammability. A new design may be reprogrammed onto the same device as many times as it is necessary. Reprogrammability allows also for fully device testing before it is shipped which leads to high quality and reliable designs. Since CMOS SRAM technology is used, the stand-by
power consumption is minimal and there is only dynamic power consumption, which is proportional to the transition frequency of inputs, outputs and internal logic.

This project has been developed targeting Xilinx's Virtex family of FPGAs [4]. Virtex FPGAs are SRAM-based and comprise two major configurable elements: configurable logic blocks (CLBs) and input/output blocks (IOBs). The CLBs are the logic building blocks, and the IOBs are used as the interface between the CLBs and the pins of the device. The device's capacity ranges from 16x24 CLBs (corresponding to about 50K logic gates) to 64x96 CLBs (corresponding to about 1M logic gates). Synchronous clock rates may reach up to 200MHz including I/O. The design's speed is greatly affected by the routing delays. For direct connections between adjacent CLBs or for feedback connections to the same CLBs local routing resources may be used. Every CLB is adjacent to a General Routing Matrix (GRM), which is a switch matrix that allows the vertical routing resources to connect to the horizontal ones. GRMs also allow the CLBs to use the general purpose routing resources: 24 single-length lines that are used for direct GRM interconnections, 96 buffered lines that connect GRMs that are 6 blocks away and 12 buffered long lines that deliver signals across the device quickly and efficiently. At the periphery of a Virtex device there exists a routing ring (Versaring) that improves I/O routability and facilitates pin locking. Dedicated routing improves the performance for some classes of signals, for example four bus lines run across each row of CLBs and two dedicated vertical nets per CLB are used to propagate carry signals efficiently. Global routing consists of four primary nets that may distribute high fan-out clock signals to every clock pin in the device and of 24 secondary backbone lines that may route to any
pin in the device. Delay-Locked Loops (DLLs) are associated with each global clock buffer so that clock edges arrive at the internal flip-flops in synchronism with clock edges arriving at the input of the device. The IOBs support a wide variety of I/O signaling standards. The input path routes the input signal to the internal logic either directly or through a flip-flop. An optional delay element assures a zero pad-to-pad hold time. The output path consists of a 3-state output buffer that drives the device’s pads. The output signal arrives at the output buffer either directly or through an optional flip-flop. The Virtex CLBs contain two slices and each slice contains two basic logic blocks (LCs). Each LC contains a four input lookup table (LUT), a storage element and some additional logic (Figure 1.2).

![Figure 1.2 Virtex Configurable Logic Block (CLB)](image)

The LUT may compute any function of four inputs or it may be used as a 16x1-bit synchronous RAM, or as a 16-bit shift register. In addition the two LCs of a given slice may be combined to implement a 32x1-bit synchronous RAM or two 16x1-bit dual port synchronous RAM. Additional logic consists of additional multiplexers that may generate any function of up to 6 inputs or selected functions of up to 19 inputs. Dedicated carry chains support high-speed arithmetic functions and each LC has an extra XOR gate that
may be used to implement a one bit full adder. Each Virtex device supports dedicated block memories of 4096 bits each that are organized in columns. Two such columns run along the device’s vertical edges and there is one memory block (4096 bits) for every four CLBs along the edge. One such memory block is a fully synchronous dual ported RAM with independent control signals and independent bus width configurations per port. Dedicated routing resources inside each block RAM facilitate interfacing to other block RAMs or CLBs.
1.4 Hardware Description Languages

Hardware Description Languages (HDLs) are the designers' main tool to model a circuit today. They allow for a concise description of a given circuit, which can be unambiguously interpreted by either humans or machines. HDLs are used not only for describing a circuit but also for synthesizing, simulating, and verifying a design. Their main advantage is that they allow for device-independent descriptions of circuits and for portability, that is reuse of part of a design in many different projects. They facilitate a faster design process and early debugging in the development cycle.

Although HDLs share some of the advantages of the software programming languages (portability, platform independence, easy debugging) they are definitely very different from them mainly due to the very specific nature of hardware. Languages used to describe hardware must be able to express concurrency since signals change in parallel. Therefore programming in a HDL more closely resembles to parallel programming for multiprocessor systems. Another difference is their ability to include structural information for a circuit and to accept user guidance as far as the performance requirements of the synthesized circuit will be. Finally, HDLs may use timing information because sometimes it is necessary that a circuit obey specific timing requirements, which is not the case for software programming languages with some possible exceptions in real-time environments.

HDLs may be classified into structural and behavioral languages. Structural languages describe the circuit in a structural view where emphasis is given to the components
comprising a circuit and their interconnections. Sometimes it is possible to describe a circuit in a hierarchical and therefore modular way. Behavioral languages on the other hand, describe the behavioral view of a circuit. The designer doesn’t need to know the exact details of the circuit’s implementation and he or she only describes the behavior in terms of the operations that need to be carried out. The concurrency of the operations is mostly affected by data-flow dependencies and by using control-flow constructs such as branches and iterations.

```
Library IEEE;
Use IEEE.std_logic_1164.all;
Entity symmux is
  port (d0, d1, sel: IN STD_LOGIC;
  q: OUT STD_LOGIC);
end symmux;
Architecture struct_mux of symmux is
component and_comp
  port(a, b: in std_logic;
  c: out std_logic);
end component;
component or_comp
  port(a, b: in std_logic;
  c: out std_logic);
end component;
component inv_comp
  port(a: in std_logic;
  b: out std_logic);
end component;
signal i1, i2, sel, n: std_logic;
For U1: inv_comp USE Entity
work.inv_comp(inv_behe);
For U2, U3: and_comp USE Entity
work.and_comp(and_behe);
For U4: or_comp USE Entity
work.or comp(behi);
Begin
U1: inv_comp port map(sel, sel, i1);
U2: and_comp port map(d0, sel, i1);
U3: and_comp port map(d1, sel, i1, i2);
U4: or_comp port map(i1, i2, q);
End struct mux;
```

**Figure 1.3 Samples of structural and behavioral code for a 2to1 multiplexer**
For this project the circuits under consideration have been modeled with VHDL [8, 10], which stands for Very High Speed Integrated Circuit (VHSIC) Hardware Description Language.

VHDL has been standardized in 1987 by the IEEE for specifying digital circuits, but today it has become the standard for modeling and synthesizing digital circuits and systems. VHDL supports both structural and behavioral descriptions and any mix of the two. Components may be specified in a generic way (e.g. variable bus widths), stored in a library and reused in many designs. It accepts both concurrent and sequential statements and provides support for error management. The main building block in VHDL is the component whose contents are not necessarily visible to the user. Components may be stored in the library and used for a hierarchical top-down or bottom-up design. Their main parts are the entity and the architecture. The entity describes how to interface with the component, that is its input and output ports, and the architecture contains the structural or behavioral description of the component.

### 1.5 Circuit Models

In order for CAD tools to be able to "read" and understand a given circuit, formal abstract circuit models have been developed. Those models are generally the result of the compilation of a hardware description language design and they also specify a circuit such that no misunderstandings may take place. Models such as the structural, the logic network, the state diagram and the data-flow and sequencing graph models describe a
given circuit at different levels (architectural, logical) and at different views (behavioral, structural) and are basically using graphs to represent circuits [3].

The structural model represents the circuit as a set of defined components, a set of nets and a relation among components and nets. It is usually a hypergraph where the nodes of the graph represent the components of the circuit and the nets among the components (or equivalently the arcs among the nodes) are represented by an incidence matrix. Sometimes only the ports of the components are given and the incidence matrix specifies net connections among the ports. In the special case where not the whole matrix is given, but instead we have a list of existing nets among nodes, then the model is called a netlist.

The logic network model is a hierarchical structure, which is usually represented by a graph. The pins or ports of the circuit are either inputs or outputs and the pins that do not belong to subcomponents and help to interface to external circuits are called primary inputs and primary outputs correspondingly. Each component represents a multiple-input single-output logic function. Each net connects a primary input or the output of a component to a number of input pins of components or to primary outputs of the circuit. The model may be represented by a graph where the nodes are either the primary ports of the circuit, or a component. The edges represent the nets among the node pins. No combinational feedback exists among the components and therefore the nets induce a partial order on the components. The logic network model may be considered a hybrid model since a given logic function may not be mapped as given and may need to be
further decomposed to simpler functions. The model may also be extended to include sequential elements in which case it is called a synchronous network model.

The state diagram model is a finite-state-machine transition diagram, which describes the behavior of a sequential circuit at the logic level. It is usually given as a graph, where the nodes of the graph represent the different states of the circuit and the edges of the graph represent the state transitions. Hierarchical representations may be obtained by splitting the state diagram into subdiagrams, which are connected through entry and exit states to calling nodes of a diagram at a higher level in the hierarchy. The model may either represent a Moore state-machine where the outputs depend only on the state of the circuit or a Mealy state-machine where the outputs depend on both the state and the current inputs of the circuit.

The data flow model is used to describe the behavior of circuits at the architectural level. It is basically a graph where its nodes represent operations and the edges represent data dependencies among the operations. Data dependencies may exist because an operation may need a previous result to start its computation, because not enough resources exist (an operation may wait for a resource that is occupied by another operation) or for event synchronization. Control-flow information may be added by using branching nodes that evaluate conditional clauses. The sequencing graph model is a hierarchical data-flow model where control-flow operations such as branching and iterations have been added as link nodes that link to another sequencing graph entity at a lower level. The link and operation nodes of the upper level constitute an acyclic graph that shows only data-flow
and serialization dependencies. It is also polar, since there is a first or source task and a
last or sink task. Nodes in such graphs may carry information about area or delay
estimates. Data dependent delays may be bounded or unbounded. The total delay is called
latency. In case we have only bounded delays the model is called a bounded-latency
graph.

1.6 EDIF format

EDIF stands for Electronic Design Interchange Format and is a standard format that
allows for easy data interchange between CAD systems. It has gained widespread
industry acceptance for easy exchange of netlist and schematic data. It has a similar to
Lisp syntax, which makes it easy for machine parsing and attractive for future extensions.
Every construct in the format starts with an opening parenthesis and a keyword, which is
followed by data items or by other constructs nested recursively. A data item may be an
integer or a string token or an EDIF identifier. Relationships among the various objects of
an EDIF description are possible through either containment or name referencing by
proper use of identifiers. Containment expresses simple relationships. For example a
component that exists in a library’s scope belongs to that library. Identifiers allow for
more flexible relationships such as connectivity between components. There are no
predefined identifiers in EDIF, however a named object must be already defined before it
is being referenced. Therefore no self-reference definitions are allowed in the format. The
scope of a name extends from the immediately after the closing parenthesis of the named
object’s description to immediately before the end of its smallest enclosing name scope.
The names of objects belonging to the same scope but to different classes (types) may not
interfere with each other. The definition of an object may be extended in order to add or change data. However an attribute or property value may not be overridden more than once in the same context. The value types supported are the integer, the range of integers, the point (which is useful for geometric figures descriptions), Booleans and strings whose size is unlimited. Real numbers are only supported as scaled integers.

1.7 Methodology

The goal of this project is to optimize pipelined designs that have been synthesized for FPGAs. Pipelined designs are used where high performance is required at the expense of some additional resources for registering. Long data-path operations may be split to smaller tasks that may be overlapped and executed in less time. Pipeline registers are used to hold the intermediate results between successive tasks. The operation may take more clock cycles to finish (the latency is increased), but each cycle is only a fraction of the time needed for non-pipelined operation (increased throughput). FPGAs suit pipelined designs well, because of their abundance of storage elements that may be used for registering.

A behavioral or structural description of the design is first given in VHDL. The design is then compiled using Workview Office and it is synthesized targeting the Virtex family of FPGAs. The output is an EDIF netlist, which describes the interconnections among the Look-Up Tables (LUTs) the Flip-Flops and the input and output buffers used for the design. The netlist is then analyzed and a logic network model of the circuit is created. The logic network model is used since it is more efficient to represent multiple-level
networks in a compact form. The sets of pipeline registers are then identified and the logic blocks between the sets are extracted. These blocks of logic are then fed separately to the SIS logic optimization tool [11], which uses scripts of well-known logic optimization algorithms. The transformed blocks of logic, which are now just a set of output equations (as given by SIS) need to be represented by the appropriate Look-Up Tables and inserted into the new netlist file. The final interconnections are then computed and inserted as well. The steps of the procedure may be seen in the next figure (Figure 1.4) and will be discussed in section 3.

Figure 1.4 Methodology
The importance of the method relies to its simplicity, which only requires that the netlist file be available instead of the original source code. This is extremely useful when the source code for a design is not available. Another application is when a netlist needs to be split in order for the design to be partitioned in multiple FPGAs. The new netlist files may then be separately optimized. The more efficient space utilization which results from the optimization, allows for more complicated designs to fit in the same space, or even for multiple ones in case there is no restriction on the available device pins.
2.0 Optimization

2.1 Two-Level Optimization

A completely specified single-output Boolean function $f$ is a mapping from $B^n$ to $B$ where $B = \{0,1\}$. Any Boolean function may be specified as a sum of products (or equivalently as a product of sums). This is called the two-level representation of the function. Each product in the sum of products representation is called a "cube". For example the Boolean function $f = ab + bc + a'c'$ ($f : B^3 \to B$) consists of three products or cubes and of three variables ($a.b.c$) or five literals ($a.b.c,a',c'$). We may visualize $f$ in the three-dimensional Boolean space where each dimension corresponds to a variable and the function takes its values at the corners of a cube:

![Figure 2.1 Representing f in the 3D Boolean space](image)

In Figure 2.1 small circles on the corners of the cube show where $f$ takes the value of 1 (True) and the 3 ellipses correspond to the 3 products of $f$. The products or cubes of maximum number of literals (the maximum number in this case is three since we only
have three variables) are called minterms. A cube of fewer literals is “bigger”, since it contains, or “covers” more minterms. A minterm is “in” \( f \) if \( f \) takes the value TRUE in that minterm. As we can clearly see from the figure, five out of the eight possible minterms are included in \( f : a'b'c',a'bc',abc',abc,a'bc \). It must be obvious that there is not a unique definition of \( f \). For example an equivalent definition for \( f \) could be: \( f = a'c' + bc + bc \) or even \( f = a'c' + bc + bc + ab \). The purpose of two-level logic minimization is to express \( f \) in the minimum number of products and literals. Two-level minimization for \( f \) would therefore produce \( f = a'c' + b \) which consists of only two products and three literals.

An implicant of \( f \) is any cube (product) of \( f \) and a cover of \( f \) is the set of implicants that contains all the minterms in \( f \). A prime implicant (PI) of \( f \) is a cube that cannot be made any bigger by removing any of its literals, and a prime cover is a cover that consists only of prime implicants. Exact logic minimization algorithms try to produce a cover whose cardinality is minimum. By Quine’s theorem [9] there is a minimum cover that is prime, since if it were not prime, some of the implicants could be expanded to become prime which would not affect the cover’s cardinality. Quine and McCluskey’s [12] exact logic minimization algorithm tries therefore to identify first all the Prime Implicants of the function, and it then chooses those prime implicants that cover the function’s minterms. The covering problem is exponential in complexity [13] and the input size (number of prime implicants) is also exponential \( \left( \frac{3^n}{n} \right) \) for an \( n \)-input function) which makes exact minimization impractical for functions of more than 20 to 25 inputs.
Because of that, heuristic algorithms for logic minimization are used in practice. Most of them use a set of operators iteratively until a minimal cover is achieved. Such operators include “essentials”, “reduce”, “expand”, and “irredundant” [1,3]. We explain these terms below:

A prime implicant is called essential if it covers minterms that are not covered by other prime implicants. In the function $f$ given in the example above, $a'c'$ is an essential prime implicant. The “essentials” operator identifies those essential primes and stores them, since they must participate in the final minimal cover.

The “reduce” operator replaces an implicant with a smaller one that is contained in it, in a way that the new set of implicants is not prime but it still covers the function. The cardinality of the cover is not affected, however redundant implicants may be discarded and subsequent “expansions” may give a better cover.

The “expand” operator tries to expand implicants such that they become prime, and removes the implicants that are covered by the expanded ones. The result is a prime, irredundant cover.

The “irredundant” operator transforms the cover to an irredundant or minimal cover. It achieves that by finding the maximum number of implicants that may be discarded while the function is still covered.
2.2 Multilevel Optimization

Although two-level logic is sufficient to describe an arbitrary Boolean function, multilevel designs often offer a compact way to express a complex circuit, whose implementation would be prohibitive as a sum of products (or product of sums). Multilevel logic is often more flexible when circuit changes must be made or certain specifications need to be met. In addition the designer may only have a certain number of components which only allow for multilevel representations. This is exactly the case when designing with FPGA's where the design is expressed as a set of interconnected Look-Up Tables. Exact optimization algorithms have been proved inefficient for multilevel designs and heuristic methods are used instead.

It is often convenient to use the logic network model when representing multilevel logic:

![Figure 2.2 Logic network example](image-url)
The model in Figure 2.2 has five primary inputs, three primary outputs and four internal nodes. Optimization in such multilevel logic networks is based on logic transformations, which may alter the structure of the network without affecting its functionality. Such logic transformations include Elimination, Decomposition, Extraction, Simplification and Substitution [2,3]. These terms are explained below. Heuristic approaches use these transformations iteratively in order to improve the original circuit. The criterion of termination is based on the count of literals and gates used.

Elimination removes an internal node. The representation of the function that the node was implementing must be substituted in every reference of the removed node. In the example above, node \( h \) may be removed but the function it implements \((d + a)\) must be inserted in node \( i \). The resulting design would be:

![Figure 2.3 Elimination example](image-url)
Decomposition splits a node into two or more nodes, which are logically equivalent to the original node. For example node $g$ above may be decomposed to two nodes:

$k = a + b$ and $g = k(c + d) + e$, producing:

![Diagram](image)

Figure 2.4 Decomposition example

Extraction identifies common factors that exist in two or more nodes and extracts them to new nodes. In the example above, the factor $c + d$ may be extracted from nodes $g$ and $i$ and represented as a new node $m$:

![Diagram](image)

Figure 2.5 Extraction example
Simplification tries to optimize the representation of the function of a specific node. It may eliminate variables and therefore affect the structure of the circuit. In our example simplification of node \( f = ae'd + d' + be' \) results in \( f = ae' + d' + be' = (a + b)e' + d' \):

![Figure 2.6 Simplification example](image)

Substitution inserts an existing node into the representation of the function of another in order to simplify it. For example node \( k \) may be substituted in node \( f \):

![Figure 2.7 Substitution example](image)
3.0 Implementation

3.1 General Description

The goal of the project is to identify and isolate common blocks of logic between the pipelines that exist in the design, but in order to be able to scan the logic of the given design we must first have a way to represent the circuit's model in the computer's memory. A logic network model is therefore created that represents the interconnections among the Look-Up Tables described in the EDIF netlist. The procedure followed by the code is highly automated and consists of the following tasks, which are carried out in the order they are described.

First the input netlist is parsed in order to identify the major circuit blocks and the blocks are then connected to form the circuit's model. The design's storage elements (flip-flops) are identified and the blocks of logic feeding each flip-flop are defined. The flip-flops are then chosen to participate in a number of classes (a class comprises of flip-flops that exist in the same pipeline) and the logic in each class is extracted in the form of multilevel equations. The equations need to be transformed in a form readable by SIS, which is the multilevel optimization tool that is used and are in turn written to a number of class files. Each class file is then optimized using SIS optimization scripts. The scripts may vary according to the class being optimized and this gives the designer a very high flexibility to his or her optimization approach. The output, optimized files are then read by the program and are compared to the original class files. If the files don't show any reduction in logic, no action is taken. If on the other hand an optimized output file is found a
number of algorithms are executed to remove the old blocks from the network model, to insert the new blocks and to modify the existing nets or to insert new nets to the netlist. The algorithms construct a number of delete-insert commands that are read by a separate function that transforms the input netlist to the optimized one. The above steps are described in more detail in the following sections.

3.2 Parsing and Constructing the Network Logic Model

The parser has to identify the major building blocks of the circuit, which are called instances and their interconnections, which are called nets in the EDIF language. Each instance has a type associated with it and may be one of the following: an input buffer (IBUF), an output buffer (OBUF), a storage element (FD), a Look-Up Table of one up to four inputs (LUT1, LUT2, LUT3, LUT4), a ground (GND) or supply (VCC) voltage or a clock driver (BUFGP). Every block has an output net, which describes to which other blocks its output is connected. As each instance is identified, it is added to the design, which is a linked list of blocks. Every block is a structure of type blockstr (Figure 3.1) and its basic fields are its name (name), the type associated with it (type), the netlist's line number where it is described (lineinfile), the line number where its net is described (outnetinfile), the name of its net (netName), a pointer to a table describing the net of the block (ports_ptr), a number of the blocks it is connected to (outputsNum), pointers to the blocks that are inputs to it (input[]), pointers to the blocks its output goes to(output[]) and a pointer to the next block in the design's list (next). In case the block is a Look-Up Table (LUT1-LUT4) there is a pointer to the equation it implements (EQN).
Since we are only interested in nets connecting LUTS and Flip-Flops we mark as invalid the nets that drive the preset (PRE), clear (CLR), chip enable (CE) and clock (C) inputs and the power and ground lines. The valid nets are kept in the block’s ports_ptr table, which is a list of port names and types (input or output). Each net that is detected is fed to the join_net function, which is responsible for connecting the input and output pointers of every block according to the block’s ports_ptr table. The logic network model is then completed after all instances and all nets have been read and the following figure (Figure 3.2) shows part of the circuit’s internal representation.
The auxiliary function $showconnections$ may be used to navigate through the whole circuit once the model is complete. Global variables are used to keep the maximum instance and net label and the position of the last net in the file, in case we need to add new instances or nets.

### 3.3 Pipeline Identification and Multilevel Equations Extraction

After the model of the circuit is constructed, $identifyblocks$ is called, which will find the pipelines in the circuit and the logic between them. First the design's Flip-Flops are found and an array of $FF\_classes\_str$ is constructed, with pointers to the Flip-Flops. Then the recursive function $visit$ is called, which starts at every Flip-Flop and visits its input. If a LUT is visited, the visit continues at its inputs (one to four inputs depending on the LUT type) recursively and it only stops when a Flip-Flop is encountered. Every time a LUT is found, it is added to the Flip-Flop's list of logic, which is pointed by $class\_list$ in the $FF\_classes\_str$ structure and the number of blocks feeding the Flip-Flop is recorded in the $logicblocksNo$ field. The Flip-Flops are then divided into classes, where each class
contains Flip-Flops that have common blocks of logic and are therefore part of the same pipeline. After the classes have been identified, the \texttt{FF\_classes\_str}, which point to Flip-Flops that belong to the same class, are interconnected through the \texttt{next\_inclass} pointers. This is an easy way to travel through all the logic blocks (LUTs) that are feeding Flip-Flops of the same class (Figure 3.3).

![Figure 3.3 FF\_classes\_str structure](image)

Finally the class files are written, which contain all the LUT equations that exist in the same class. The equations must be transformed so that the indefinite inputs are given specific names that correspond to the blocks that are feeding the LUTs. There is a
difference between inputs that come directly from LUTs in the same class, which are
named after the feeding block’s name, and the external inputs, which are written in the
file in a descriptive way. For example input i0Cxxx is the external 0th input to the Cxxx
logic block. In the following figure such an equation transformation is shown:

\[
C358: \text{"(I1' I3 + I3 I0 I2)"}
\]

\[
\downarrow
\]

\[C358 = i1C358' C359 + C359 i0C358 C363;\]

*Figure 3.4 Equation Transformation Example*

### 3.4 SIS Optimization

SIS is a logic synthesis and optimization tool [11] developed at the University of
California, Berkeley, which is capable of running scripts to optimize a set of multilevel
logic equations. Each equation is associated with a circuit’s node and the variables of the
equation are the node’s inputs. The scripts contain commands that employ multilevel
optimization algorithms to the given equations. The commands used in the scripts for this
project are: sweep, eliminate, full_simplify, xl_partition, xl_imp, xl_merge and xl_cover.

Sweep “cleans” the input equations by eliminating constant expressions and removing
single-input nodes. It must be executed before the network has been mapped since it may
easily “unmap” it.
Eliminate takes an argument as input and collapses the nodes with value less than the
threshold provided into their fanouts. The value of a node is computed by counting the
literals that are saved by leaving the node in the network. A node that is used only once
has a value of -1. It cannot eliminate primary inputs or outputs.

Full_simplify runs the 2-level ESPRESSO minimization algorithm for each node. It is
useful for removing network redundancies.

Xl_partition may reduce the number of nodes by collapsing them into their fanouts. Each
node is associated with a cost depending on the new nets that are created after the
collapse and the nodes with the lower cost are chosen for collapsing. The size of the LUT
is given so that the number of fanins is not enlarged. The -t option tries first the nodes
that may be collapsed into all their fanouts.

Xl_imp applies various decomposition techniques to transform an infeasible network to a
network ready for mapping for a given support.

Xl.merge tries to minimize the number of nodes by merging pairs of nodes. Limits on the
common fanins of the mergeable nodes and on the union of their fanins may be specified.
Finally xl_cover, is used to map the nodes onto Xilinx’s CLB’s. The CLB’s size is given as input and a number of different heuristic algorithms may be used to solve the covering problem.

3.5 Netlist Modification

For every class, the optimized equations are extracted from the corresponding output optimized file, and if they are found fewer than the original equations in the input class file, they are used to modify the final netlist. The extract_eqn_fromfile function fills an array of outeqnstr structures for every class, where every structure contains a pointer to the optimized equations, the type and the name of the block owing the equation and a mapping-renaming table. The mapping-renaming table maps the equation variables to specific LUT inputs and assigns an instance name for new nodes in the optimized equations. The renaming is consistently performed for every equation in the set and the equations for every LUT are then transformed into an EDIF compatible form. The replaceinstances and replacenets functions are called next to create a list of the necessary netlist modifications.

Replaceinstances performs all the necessary tasks to obtain a netlist with the new instances. It first finds the instances in the old netlist that do not exist any more in the new optimized equations, and issues delete commands for them and their nets. It then creates new instances for the remaining blocks. If a block was already in the original class file, its instance is first deleted before its new instance description is inserted.
The commands are inserted in a linked list of commandstr structures. Each commandstr contains a command field that shows whether the command is a deletion or an insertion, a lineno field which indicates the line number in the original netlist file that the command has to take place, a newitem field which contains the new instance or net for the insertion commands and a pointer to the next node in the list. Every time a new command is inserted, it is placed such that the list is always ordered by the lineno field. If an insert and a delete command are having equal line numbers, they are also ordered such that the deletion happens before the insertion. This approach was chosen to make the output netlist creation faster.

Replacenets is the function that makes all the net modifications. In order to accomplish that, it maintains a list of modified nets (modifiednets), which holds pointers to the blocks whose nets have been modified. It first scans the mapping-renaming tables of every equation node in the class and the action it takes depends on whether it detects a primary input described as iyCxxx or an internal input (from a block internal to the class). In the case of primary inputs the block that was feeding the y-th input of the Cxxx block must be found and its net must be changed to reflect the new situation. More specifically after the feeding block it is found, the entry in its ports_ptr table referring to the Cxxx’s block y-th input is deleted and replaced by the name and port of the node that is associated with the equation being scanned. The feeding block is then inserted in the modified nets list. For the internal inputs case, the block’s ports_ptr table is scanned and every reference to blocks in the same class is deleted. Then one entry is added reflecting the connection from the block’s output to the node, which is associated with the equation, and the block
is inserted in the modified nets list. The ports_ptr tables are dynamically allocated and
their size varies according to the space requirements. For every new node in the
optimized equations a new net is created. After the mapping-renaming tables for every
equation are scanned, the new net is filled with the block names the output of the new
node connects to. It is then copied in a commandstr buffer and an insertion command is
added to the commands list.

Finally the modified nets list is traversed and for every node in the list a new net is
created, deletion commands are issued for the old nets and insertion commands are added
to the commands list for the corresponding new nets.

3.6 Creating the New EDIF Netlist

The desired result, which is the new transformed netlist is the task of the createnewedif
function. This function reads the original netlist line by line and checks whether the
command at the head of the commands list matches the current line number. If it doesn’t,
the line is copied to the output netlist file as is, otherwise the command is executed first.
Delete commands specify the line where the instance or the net to be deleted begins and
they are executed by just skipping the appropriate number of lines from the input file.
The last line to be skipped is easily found since the removed expression must have
balanced parentheses. For the insert commands, the buffer describing the new net or node
is just copied to the output netlist file. The commands list must be searched until no more
commands exist for the current line number since multiple commands may have been
issued at the same line.
4.0 Results

4.1 Test Designs

In order to verify the feasibility of our approach, we tested two different designs. The first design is a number of state machines feeding each other, so the set of flip-flops determining their states are the actual pipeline registers. The second design is the implementation of huffman encoding. It is part of the benchmarking tools for configurable logic developed by Honeywell, Inc. Although the huffman encoding design is much bigger than the state machines example, it is not a pipelined design since it only has one set of input latches and one set of output flip-flops.
4.2 State Machines Example

The state machines example consists of three state machines connected one after the other. The VHDL source code for this design may be seen in Appendix A. In the following figure (Figure 4.1) we may see the logic implemented by each one of them:

![State Machines Example Diagram]

*Figure 4.1 State Machines Example*

The next figure (Figure 4.2) shows the schematic that was obtained after compiling the design under Workview Office and translating the resulting EDIF netlist to a schematic read by Viewdraw:

![Schematic Diagram]

*Figure 4.2 Original translated schematic (Example-1)*
It is clear that we have three pipeline registers, one for every state machine’s output state. The five identified classes of logic are also shown as found by the algorithm. The class files containing the equations for each one of them may be found at Appendix C.

These class files are then fed to SIS and five output files are obtained which may also be seen at Appendix C. It is obvious from those files that only Class-4 (the class containing 18 Look-Up Tables originally) could be optimized with only 16 equations. It is important to note that the optimization depends on the specific script used under SIS. The script that we have finally decided to use for our design may be found at the end of Appendix C as well. We have therefore achieved a 2 LUTs reduction from the original design. The complete design after the program modifies the netlist is shown in the following figure (Figure 4.3)

![Diagram](image)

**Figure 4.3 Optimized design schematic (Example-1)**
4.3 Huffman Encoding Example

The second example we tried to optimize was the implementation of static Huffman coding trees in VHDL (see Appendix A). These trees are used for variable bit length encoding of input characters and are able to compress an input stream of characters by assigning shorter codes to the most frequently occurring characters.

In the following figure (Figure 4.4) we may see the schematic obtained by the original EDIF netlist:

![Figure 4.4 Original translated schematic (Example-2)]

Most of the LUTs participate in the same class and in the figure they are shown inside the drawn loop. This is the only class that was optimized and we had a reduction of one LUT for this design. The optimized schematic is shown in the next figure (Figure 4.5).
Figure 4.3 Optimized design schematic (Example-1)
5.0 Discussion

The examples above show that the technique works best for pipelined designs. The reason for that is mainly because it concentrates in blocks of logic inside the pipeline stages, and doesn't look at the whole design at once. If we have a non-pipelined design as is the case in the second example presented it has already been optimized globally by the synthesis software and therefore the method cannot optimize it much more.

The circuit designs tested were optimized in terms of area (since the optimized designs use fewer LUTs) without increasing the levels of logic. The technique we have presented is therefore useful when a big pipelined design needs to fit in a device, when multiple designs are targeted to the same device or even when a design has been partitioned to fit in multiple devices.

The versatility of the method however, helps the designer to do speed optimization as well. It is possible for example to manually edit the class files to selectively optimize a certain propagation paths. The designer may even change the SIS scripts to sacrifice more area in order to reduce the levels of logic for a particular design or even for just one pipeline stage of the design.

The fact that the method manipulates the netlist directly makes it powerful, not only because a variety of tools may use its results, but also because it eliminates the need for the original source code especially when it is not available.
A number of techniques for sequential logic optimization have been successfully tested that use state minimization and state encoding algorithms. The idea is that if the number of states is reduced, fewer storage elements will be needed, and since fewer state transitions will occur, the driving logic will be simpler. However these techniques have to alter the state encoding which is sometimes unacceptable, for example when interfacing to another circuit.

Retiming is another popular technique for sequential logic minimization [5]. The main idea behind retiming is to move the registers so that the area or the cycle-time is minimized and the overall circuit functionality remains the same. This method is not applicable when we are interested in preserving the intermediate pipeline states (for example when they are used to control some extra logic). If on the other hand we have the freedom to move the registers, retiming can also be combined with our approach.

The approach we have presented is a flexible and powerful technique for optimizing the multilevel logic combinational components in pipelined designs targeted for Field Programmable Gate Arrays. It provides the designer with many degrees of freedom in order to optimize parts of the design focusing on either area or speed. It is applied on the circuit’s netlist directly and is therefore transparent to the synthesis process.
6.0 References


Appendix A

The VHDL source code for the designs we tested may be seen below:

State Machines Example:

```vhdl
Library IEEE;
Use IEEE.std_logic_1164.ALL;

Entity m3fsm is port (  
    reset, clk,  
in_l_a, in_l_b,  
in_l_c : in std_logic;  
out_l_a,  
out_l_b,  
out_l_c : out std_logic  
);
end m3fsm;

Architecture fsm of m3fsm is  
type State3Type is ( state_a, state_b, state_c);  
type State4Type is ( state_a, state_b, state_c, state_d);

signal a_p_state, a_n_state : State4Type;  
signal b_p_state, b_n_state : State3Type;  
signal c_p_state, c_n_state : State3Type;

signal out_l_a, out_l_b, out_l_c, out_l_d : std_logic;  
signal out_2_a, out_2_b, out_2_c : std_logic;
begin

state_comb_a: process (reset, a_p_state, in_l_a, in_l_b, in_l_c)
begin
  if (reset = '1') then
    out_l_a <= '-' ; out_l_b <= '-' ; out_l_c <= '-' ; out_l_d <= '-' ;  
a_n_state <= state_a;  
  else  
    case a_p_state is  
      when state_a => out_l_a <= '1'; out_l_b <= '0'; out_l_c <= '0'; out_l_d <= '0';  
      if ( in_l_a = '1' and in_l_b = '1') then
        a_n_state <= state_c;
      elsif (in_l_a = '1' and in_l_c = '1') then
        a_n_state <= state_b;
      elsif (in_l_a = '1' and in_l_c = '1') then
        a_n_state <= state_d;
      else
        a_n_state <= state_a;  
      end if;

      when state_b => out_l_a <= '0'; out_l_b <= '1'; out_l_c <= '1'; out_l_d <= '1';
      if (in_l_b = '1') then
        a_n_state <= state_a;
      elsif (in_l_a = '1' and in_l_c = '1') then
        a_n_state <= state_c;
      elsif (in_l_a = '1') then
        a_n_state <= state_b;
      else
        a_n_state <= state_b;
      end if;

      when state_c => out_l_a <= '1'; out_l_b <= '0'; out_l_c <= '0';
      if (in_l_a = '1' and in_l_c = '1') then
        a_n_state <= state_c;
      elsif (in_l_a = '1') then
        a_n_state <= state_b;
      else
        a_n_state <= state_b;
      end if;

      when state_d => out_l_a <= '1'; out_l_b <= '0'; out_l_c <= '0';
      if (in_l_a = '1' and in_l_c = '1') then
        a_n_state <= state_d;
      elsif (in_l_a = '1') then
        a_n_state <= state_c;
      elsif (in_l_a = '1' and in_l_c = '1') then
        a_n_state <= state_b;
      else
        a_n_state <= state_b;
      end if;

end if;
```
a_n_state <= state_b;
elseif (in_l_b = '1' and in_l_a = '1') then
  a_n_state <= state_g;
elseif (in_l_c = '1' and in_l_b = '1') then
  a_n_state <= state_a;
else
  a_n_state <= state_c;
end if;
when state_d => out_l_a <= '1'; out_l_b <= '1'; out_l_c <= '0'; out_l_d <= '1';
if (in_l_a = '1' and in_l_c = '0') then
  a_n_state <= state_b;
elseif (in_l_b = '1' and in_l_a = '0') then
  a_n_state <= state_g;
elseif (in_l_b = '0' and in_l_a = '0') then
  a_n_state <= state_c;
else
  a_n_state <= state_c;
end if;
end case;
end if;
end process state_comb_a;

state_comb_b: process (reset, b_p_state, out_l_a, out_l_b, out_l_c, out_l_d)
begin
if (reset = '1') then
  out_2_a <= '-' when state_a => out_2_a <= '1'; out_2_b <= '1'; out_2_c <= '0';
b_n_state <= state_a;
else
  case b_p_state is
    when state_a => out_2_a <= '1'; out_2_b <= '1'; out_2_c <= '0';
    when state_b => out_2_a <= '0'; out_2_b <= '1'; out_2_c <= '1';
    when state_c => out_2_a <= '1'; out_2_b <= '1'; out_2_c <= '1';
  end case;
end if;
end process state_comb_b;

state_comb_c: process (reset, c_p_state, out_2_a, out_2_b, out_2_c)
begin

if (reset = '1') then
  out_3_a <= '>'; out_3_b <= '>'; out_3_c <= '>'; c_n_state <= state_a;
else
  case c_p_state is
    when state_a => out_3_a <= '1'; out_3_b <= '1'; out_3_c <= '1';
    when state_b => out_3_a <= '0'; out_3_b <= '1'; out_3_c <= '0';
    when state_c => out_3_a <= '0'; out_3_b <= '0'; out_3_c <= '0';
  end case;
end if;
end process state_comb_c;

state_clocked: process (clk)
begin
  if rising_edge (clk) then
    a_p_state <= a_n_state;
    b_p_state <= b_n_state;
    c_p_state <= c_n_state;
  end if;
end process state_clocked;
Huffman Example:

Interface.vhd:

```
-- Copyright (November 1997) Honeywell Inc. Unpublished - All rights reserved. This material may be reproduced by, or for, the U.S. Government pursuant to the copyright license under the clause at DFARS 252.227-7013 (Oct. 1988).
--
-- Filename: interface.vhd
-- Description: Provides the host and memory interface
--
LIBRARY IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_misc.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

ENTITY interface IS
PORT (
    clk : IN std_logic;
    resetn : IN std_logic;
    interrupt_ackn : IN std_logic;
    bus_grantn : IN std_logic;
    done : IN std_logic;
    bus_reqn : OUT std_logic;
    interrupt_reqn : OUT std_logic
);
END interface;

ARCHITECTURE interface_arch OF interface IS

TYPE states IS (init, request_intl1, process_int1, request_bus, process_bus, request_int2, process_int2);
signal current_state, next_state : states;
signal init_count : std_logic_vector(1 downto 0);

begin

statetrans: PROCESS(clk, resetn)
begin
if (resetn = '0') then
    current_state <= init;
elsif (clk'event and clk='1') then
    current_state <= next_state;
end if;
end process;

process(clk, resetn)
begin
if (resetn = '0') then
    init_count <= "00";
elsif (clk'event and clk='1') then
    init_count <= init_count + '1';
end if;
end process;

PROCESS(current_state, init_count, interrupt_ackn, bus_grantn, done)
```
begin
  interrupt_reqn <= '1';
  bus_reqn <= '1';
  case (current_state) is
  when init =>
    if (init_count = "11") then
      next_state <= request_int1;
    else
      next_state <= init;
    end if;
  when request_int1 =>
    interrupt_reqn <= '0';
    if (interrupt_ackn = '0') then
      next_state <= process_int1;
    else
      next_state <= request_int1;
    end if;
  when process_int1 =>
    if (interrupt_ackn = '1') then
      next_state <= request_bus;
    else
      next_state <= process_int1;
    end if;
  when request_bus =>
    bus_reqn <= '0';
    if (bus_grantn = '0') then
      next_state <= process_bus;
    else
      next_state <= request_bus;
    end if;
  when process_bus =>
    bus_reqn <= '0';
    if (done = '1') then
      next_state <= request_int2;
    else
      next_state <= process_bus;
    end if;
  when request_int2 =>
    interrupt_reqn <= '0';
    if (interrupt_ackn = '0') then
      next_state <= process_int2;
    else
      next_state <= request_int2;
    end if;
  when process_int2 =>
    next_state <= process_int2;
  when others =>
    next_state <= process_int2;
  end case;
end process;
end interface_arch;

CONFIGURATION config_interface OF interface IS
  FOR interface_arch
END FOR;
END config_interface;
State.vhd:

---
-- Copyright (November 1997) Honeywell Inc. Unpublished - All rights reserved. This material may be reproduced by, or for, the U.S. Government pursuant to the copyright license under the clause at DFARS 252.227-7013 (Oct. 1988).
--
-- Filename: state.vhd
-- Description: Provides the appropriate latency and control generation for interfacing to the memory controller
--
---

LIBRARY IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_misc.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

ENTITY state IS
  PORT (  
    Clk : IN  std_logic;
    Reset : IN  std_logic;
    Go : IN  std_logic;
    EncodeLength : IN  std_logic_vector(2 downto 0);
    Length : IN  std_logic_vector(2 downto 0);
    Count : IN  std_logic_vector(2 downto 0);
    Term : IN  std_logic;
    LoadState : OUT  std_logic;
    ShiftState : OUT  std_logic;
    WriteState2 : OUT  std_logic;
    WriteState3 : OUT  std_logic
  );
END state;

ARCHITECTURE state_arch OF state IS

TYPE states IS (load, shift, write1, write2, write3, done);
signal current_state, next_state : states;
signal WriteState1, DoneState : std_logic;
begin
  staterans : PROCESS(Clk,Reset)
  begin
    if (Reset = '0') then
      current_state <= load;
    elsif (Clk'event and Clk='1') then
      if (Go = '1') then
        current_state <= next_state;
      end if;
    end if;
  end process;

  state : PROCESS(current_state, EncodeLength, count, length, term)
  begin
    LoadState <= '0';
    ShiftState <= '0';
    WriteState1 <= '0';
    WriteState2 <= '0';
    WriteState3 <= '0';
    DoneState <= '0';
    case (current_state) is
      when load =>
        if (Count = "111" or (EncodeLength = "000" and Term = '1')) then
          next_state <= write1;
      when shift | write1 | write2 | write3 | done => next_state <= current_state;
    end case;
  end process;

end state_arch;

elsif (EncodeLength = "000") then
  next_state <= load;
else
  next_state <= shift;
end if;
LoadState <= '1';
when shift =>
  if (Count = "111" or (Length = "001" and Term = '1')) then
    next_state <= writel;
  elsif (Length = "001") then
    next_state <= load;
  else
    next_state <= shift;
  end if;
ShiftState <= '1';
when writel =>
  next_state <= write2;
  WriteState1 <= '1';
when write2 =>
  next_state <= write3;
  WriteState2 <= '1';
when write3 =>
  if (Term = '1' and Length = "000") then
    next_state <= done;
  elsif (Length = "000") then
    next_state <= load;
  else
    next_state <= shift;
  end if;
  WriteState3 <= '1';
when done =>
  next_state <= done;
  DoneState <= '1';
when others =>
  next_state <= done;
end case;
end process;

end state_arch;

CONFIGURATION config_state OF state IS
  FOR state_arch
    END FOR;
END config_state;
Huffman.vhd:

LIBRARY IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_misc.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

ENTITY huffman IS
  PORT(
    Clk    : IN std_logic;
    Resetn : IN std_logic;
    Go     : IN std_logic;
    Done   : OUT std_logic;
    WEn    : OUT std_logic;
    OE     : OUT std_logic;
    Address : OUT std_logic_vector(17 downto 0);
    DataIn : IN std_logic_vector(7 downto 0);
    DataOut : OUT std_logic_vector(7 downto 0)
  );
END huffman;

ARCHITECTURE huffman_arch OF huffman IS

COMPONENT control PORT(Clk : IN std_logic;
  Resetn : IN std_logic;
  Go : IN std_logic;
  EncodeLength : IN std_logic_vector(2 downto 0);
  DoneOut : OUT std_logic;
  LoadStateOut : OUT std_logic;
  ShiftStateOut : OUT std_logic;
  ReadAddressOut : OUT std_logic_vector(5 downto 0);
  WriteAddressOut : OUT std_logic_vector(5 downto 0);
  WEn : OUT std_logic
); END COMPONENT;

COMPONENT huff PORT(DataIn : IN std_logic_vector(7 downto 0);
  Length : OUT std_logic_vector(2 downto 0);
  DataOut : OUT std_logic_vector(7 downto 0)
); END COMPONENT;

signal EncodeLength : std_logic_vector(2 downto 0);
signal dgo : std_logic;
signal Count : std_logic_vector(2 downto 0);
signal LoadState, ShiftState : std_logic;
signal WriteState : std_logic;
signal UncodeData, EncodeData : std_logic_vector(7 downto 0);
signal ShiftData : std_logic_vector(6 downto 0);
signal WriteAddress, ReadAddress : std_logic_vector(5 downto 0);
signal TempDataOut : std_logic_vector(7 downto 0);

begin
ctrl : control PORT MAP(Clk => Clk, Resetn => Resetn, Go => Go,
   EncodeLength => EncodeLength,
   DoneOut => Done,
   LoadStateOut => LoadState,
   ShiftStateOut => ShiftState,
   ReadAddressOut => ReadAddress,
   WriteAddressOut => WriteAddress,
   WEn => WEn);

enc : huff PORT MAP(DataIn => UncodeData, Length => EncodeLength,
    DataOut => EncodeData);

DataOut <= TempDataOut;

PROCESS(Clk)
begin
  if (Clk'event and Clk='1') then
    if (LoadState = '1') then
      ShiftData <= EncodeData(7 downto 1);
    elsif (ShiftState = '1') then
      ShiftData <= '-' & ShiftData(6 downto 1);
    end if;
    if (LoadState = '1') then
      TempDataOut(7) <= EncodeData(0);
    elsif (ShiftState = '1') then
      TempDataOut(7) <= ShiftData(0);
    end if;
    if (((LoadState = '1' and Go = '1') or ShiftState = '1') then
      TempDataOut(6 downto 0) <= TempDataOut(7 downto 1);
    end if;
    if (LoadState = '1') then
      UncodeData <= DataIn;
    end if;
  end if;
END PROCESS;

Address(17 downto 6) <= "0000000000000000";

PROCESS(LoadState, ShiftState, WriteAddress, ReadAddress)
begin
  if (LoadState = '1' or ShiftState = '1') then
    Address(5 downto 0) <= ReadAddress;
  else
    Address(5 downto 0) <= WriteAddress;
  end if;
END PROCESS;

end huffman_arch;

CONFIGURATION config_huffman OF huffman IS
  FOR huffman_arch
END FOR;
END config_huffman;
LIBRARY IEEE;
use IEEE.std_logic_1164.all;

ENTITY huff IS
  PORT (   
    DataIn : IN std_logic_vector(7 downto 0);
    Length : OUT std_logic_vector(2 downto 0);
    DataOut : OUT std_logic_vector(7 downto 0)
  );
END huff;

ARCHITECTURE huff_arch OF huff IS
BEGIN

PROCESS(DataIn)
BEGIN
  case DataIn is
    when "00011111" =>
      Length <= "101";
      DataOut <= "00011110";
    when "00011110" =>
      Length <= "101";
      DataOut <= "00011111";
    when "00011101" =>
      Length <= "101";
      DataOut <= "00100110";
    when "00011011" =>
      Length <= "101";
      DataOut <= "00100111";
    when "00010111" =>
      Length <= "101";
      DataOut <= "00100100";
    when "00010101" =>
      Length <= "101";
      DataOut <= "00100101";
    when "00001101" =>
      Length <= "101";
      DataOut <= "00101000";
    when "00001100" =>
      Length <= "101";
      DataOut <= "00101001";
    when "00001011" =>
      Length <= "101";
      DataOut <= "00110000";
    when "00001010" =>
      Length <= "101";
      DataOut <= "00110001";
    when "00001000" =>
      Length <= "101";
      DataOut <= "00110100";
    when "00001001" =>
      Length <= "101";
      DataOut <= "00110101";
  end case;
END PROCESS;

END huff_arch;
when "C0010110" =>
  Length <= "101";
  DataOut <= "00110111";
when "00010010" =>
  Length <= "100";
  DataOut <= "00000000";
when "00010001" =>
  Length <= "100";
  DataOut <= "00000111";
when "00011110" =>
  Length <= "100";
  DataOut <= "00010001";
when "00011101" =>
  Length <= "100";
  DataOut <= "00001101";
when "00010101" =>
  Length <= "100";
  DataOut <= "00001111";
when "00011000" =>
  Length <= "100";
  DataOut <= "00010000";
when "00011100" =>
  Length <= "100";
  DataOut <= "00011000";
when "00001010" =>
  Length <= "100";
  DataOut <= "00011111";
when "00001001" =>
  Length <= "100";
  DataOut <= "00010101";
when "00010111" =>
  Length <= "100";
  DataOut <= "00010110";
when "00010001" =>
  Length <= "100";
  DataOut <= "00011011";
when "00011101" =>
  Length <= "100";
  DataOut <= "00010011";
when "00001000" =>
  Length <= "100";
  DataOut <= "00001000";
when "00001100" =>
  Length <= "100";
  DataOut <= "00011001";
when "00001001" =>
  Length <= "100";
  DataOut <= "00010001";
when "00001010" =>
  Length <= "100";
  DataOut <= "00001010";
when "00001011" =>
  Length <= "100";
  DataOut <= "00001101";
when "00001100" =>
  Length <= "100";
  DataOut <= "00000001";
when "00001101" =>
  Length <= "100";
  DataOut <= "00000011";
when "00000010" =>
  Length <= "100";
  DataOut <= "00000100";
when "00000011" =>
  Length <= "100";
  DataOut <= "00000010";
when "00000001" =>
  Length <= "100";
  DataOut <= "00000011";
when "00000000" =>
  Length <= "010";
  DataOut <= "00001111";
when others =>
  Length <= "---";
  DataOut <= "--------";
end case;
END PROCESS;
END huff_arch;

CONFIGURATION config_huff OF huff IS
  FOR huff_arch
    END FOR;
END config_huff;
LIBRARY IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_misc.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

ENTITY control IS
PORT ( 
  Cclk : IN  std_logic;
  Resetn : IN  std_logic;
  Go : IN  std_logic;
  EncodeLength : IN  std_logic_vector(2 downto 0);
  DoneOut : OUT std_logic;
  LoadStateOut : OUT std_logic;
  ShiftStateOut : OUT std_logic;
  ReadAddressOut : OUT std_logic_vector(5 downto 0);
  WriteAddressOut : OUT std_logic_vector(5 downto 0);
  WEn : OUT std_logic 
);
END control;

ARCHITECTURE control_arch OF control IS

COMPONENT state 
PORT ( 
  Cclk : IN  std_logic;
  Resetn : IN  std_logic;
  Go : IN  std_logic;
  EncodeLength : IN  std_logic_vector(2 downto 0);
  Length : IN  std_logic_vector(2 downto 0);
  Count : IN  std_logic_vector(2 downto 0);
  Term : IN  std_logic;
  LoadState : OUT std_logic;
  ShiftState : OUT std_logic;
  WriteState2 : OUT std_logic;
  WriteState3 : OUT std_logic 
);
END COMPONENT;

signal Length : std_logic_vector(2 downto 0);
signal Count : std_logic_vector(2 downto 0);
signal LoadState : std_logic;
signal ShiftState : std_logic;
signal WriteState2 : std_logic;
signal WriteState3 : std_logic;
signal WriteAddress, ReadAddress : std_logic_vector(5 downto 0);
signal Done : std_logic;

begin

LoadStateOut <= LoadState;
ShiftStateOut <= ShiftState;
WriteAddressOut <= WriteAddress;
ReadAddressOut <= ReadAddress;
DoneOut <= Done;

state_mach : state PORT MAP (Clk => Clk, Resetn => Resetn, Go => Go,
  EncodeLength => EncodeLength, Length => Length,
  Count => Count, Term => Done,
  LoadState => LoadState, ShiftState => ShiftState,
  WriteState2 => WriteState2,
  WriteState3 => WriteState3);

PROCESS (WriteState2)
begin
  if (WriteState2 = '1') then
    WEn <= '0';
  else
    WEn <= '1';
  end if;
END PROCESS;

PROCESS (Clk, Resetn)
begin
  if (Resetn = '0') then
    ReadAddress <= (others => '0');
    WriteAddress <= (others => '0');
    Count <= (others => '0');
    Length <= (others => '0');
    Done <= '0';
  elsif (Clk'event and Clk='1') then
    if (Go = '1') then
      if (LoadState = '1') then
        ReadAddress <= ReadAddress + '1';
      end if;
      if (WriteState1 = '1') then
        WriteAddress <= WriteAddress + '1';
      end if;
      if (ShiftState = '1' or LoadState = '1') then
        Count <= Count + '1';
      end if;
      if (LoadState = '1') then
        Length <= EncodeLength;
      elseif (ShiftState = '1') then
        Length <= Length - '1';
      end if;
    end if;
    if (ReadAddress = "111111") then
      Done <= '1';
    end if;
  end if;
end process;

end control_arch;

CONFIGURATION config_control OF control IS
  FOR control_arch
  END FOR;
END config_control;
Appendix B

This is my application's source code:

Process.h:

```c
#define FILE_NAME "huffman.edf"
#define OUTFILE "outhuff.edf"

#define MAX_STR_LEN 500
#define MAX_PORTSTABLE_SZ 60
#define MAX_OUTPUTS_NO 60
#define MAX_NAME_LEN 50
#define MAX_PORT_NAME_SZ 15

char type_names[10][10] = {
  "FD",
  "IBUF",
  "OBUF_P_12",
  "LUT1",
  "LUT2",
  "LUT3",
  "LUT4",
  "BUFQ",
  "VCC",
  "GND",
};
enum block_type { FD, IBUF, OBUF, LUT1, LUT2, LUT3, LUT4, BUFQ, VCC, GND };

struct blockstr{
  char name[MAX_NAME_LEN];
  enum block_type type;
  char* EQN;

  char* instbuf;
  char* netbuf;

  int lineinfile;
  int outnetinfile;
  char* netName;
  short int netType;
  struct ports_table* ports_ptr;

  short int eqn_processed;
  short int flag; /* to mark an event */
  short int entries_removed; /* to mark an event */
  int class_of_block;
  int outputsNum;
  struct blockstr* input[4];
  struct blockstr* output[MAX_OUTPUTS_NO];
  struct blockstr* next;
};

struct ports_table{
  char port[MAX_PORT_NAME_SZ];
  char name[MAX_NAME_LEN];
  char status; /*'f'->entry written 'e'->entry available*/
};

struct net_entry_node{
  char port[MAX_PORT_NAME_SZ];
  char name[MAX_NAME_LEN];
};

struct classstr{
```
struct blockstr* blockptr;
struct classstr* next;
);

struct FF_classes_str{
 struct blockstr* FF;
 short int marked;
 short int class_processed;
 int classType;
 int logicblocksNo;
 struct classstr* class_list;
 struct FF_classes_str* next_inclass;
};

struct commandstr{
 char* command; /* command to execute d->delete i->insert */
 int lineno; /* line in edif file where to execute the command */
 char* newitem; /*buffer holding new instance or net */
 struct commandstr* next;
};

struct outeqnstr{
 char blockname[MAX_NAME_LEN];
 enum block_type type;
 char* eqn;
 char mappingTable[4][2][MAX_NAME_LEN]; /* 4 entries, 1 entry per LUT input */
 1st row=original token,
 2nd row=replaced new-node token*/
 char* outeqn1;
 char* outeqn2;
 char nodeType; /* 'o'=old Node 'n' = new Node */
};

/*modified nets list */
struct mnetslist{
 struct blockstr* blockptr;
 struct mnetslist* next;
};
Process.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "process.h"

struct FF_classes_str* FF_classes;
int FFcount;
int numClasses;
struct FF_classes_str** class_ptrArray;
int* orig_num_eqns;
int max_net_pos=0;

/***********************************************************/

getblockptr() Returns a pointer to a specific block in the design.
/***********************************************************/
void getblockptr(struct blockstr* head, struct blockstr** block_ptr, char* name)
{
    while (head!=NULL)
    {
        if (!strcmp(head->name, name))
        {
            block_ptr = head;
            break;
        }
        head = head->next;
    }
}

/***********************************************************/

join_net() Joins the design's blocks according to the nets read.
/***********************************************************/
void join_net(struct blockstr* head, struct ports_table* ports_ptr, int numports, int last_net_line, char* netname, short int rename_net)
{
    struct blockstr* out_block_ptr;
    struct blockstr* in_block_ptr;
    int i, outblock_index, outputs_count;

    for (i=0; i<numports; i++)
    {
        if ((ports_ptr[i].port[0]=='O') || (ports_ptr[i].port[0]=='Q'))
        {
            getblockptr(head, &out_block_ptr, ports_ptr[i].name);
            ifdef DEBUG2
            if (out_block_ptr==NULL)
            {
                printf("Output pointer found Null\n");
                exit(1);
            }
            endif
            outblock_index=i;
            break;
        }
    }

    out_block_ptr->outputsNum = numports-1;
    out_block_ptr->outnetinfile = last_net_line;
    if (rename_net==0)
    out_block_ptr->netType = 0;
    else
    out_block_ptr->netType = 1;

    out_block_ptr->netName = (char*)malloc(strlen(netname));
    strcpy(out_block_ptr->netName, netname);
```
```c
#define DEBUG2

output.count = 0;
for (i = 0; i < numports; i++)
    if (i == outblock_index)
        getblockptr(head, &in_block_ptr, ports_ptr[i].name);

#define DEBUG

if (in_block_ptr == NULL)
    printf("Input pointer found Null, i=%d\n", i);
    exit(1);
#endif

out_block_ptr->output[outblocks_count] = in_block_ptr;

if (!strcmp(ports_ptr[i].port, "I0") ||
    (!strcmp(ports_ptr[i].port, "D") ||
    (!strcmp(ports_ptr[i].port, "I"))
    in_block_ptr->input[0] = out_block_ptr;
else if (!strcmp(ports_ptr[i].port, "I1") ||
    (!strcmp(ports_ptr[i].port, "CE") ||
    (!strcmp(ports_ptr[i].port, "I2")
    in_block_ptr->input[1] = out_block_ptr;
else if (!strcmp(ports_ptr[i].port, "I3")
    in_block_ptr->input[2] = out_block_ptr;
else
    printf("Unknown port detected: %s Aborting Join\n", ports_ptr[i].port);
    exit(1);
    outputs_count++;
}

out_block_ptr->ports_ptr =
    (struct ports_table*)malloc((numports)*sizeof(struct ports_table));
    memcpy(out_block_ptr->ports_ptr, ports_ptr, numports*sizeof(struct ports_table));

#define DEBUG2

printf("Netname=%s Type=%d Outblockname=%s\n", 
    out_block_ptr->netName, out_block_ptr->netType, out_block_ptr->name);

for (i = 0; i < numports; i++)
    printf("i=%d port=%s name=%s status=%c\n", 
        i, out_block_ptr->ports_ptr[i].port, 
        out_block_ptr->ports_ptr[i].name, 
        out_block_ptr->ports_ptr[i].status);
#endif

/**************************
 showconnections() Navigates the user through the design.
***************************/
void showconnections (struct blockstr* head)
{
char name[MAX_NAME_LEN];
struct blockstr* block_ptr;
int i;
```
do{
    printf("please enter block name:");
    scanf("%s",name);
    if (!strcmp(name,"exit"))
        break;
    getblockptr(head, &block_ptr, name);
    printf("This block has %d inputs to the following blocks:
" ,block_ptr->outputNum);
    for (i = 0; i < block_ptr->outputNum; i++)
        printf("%s Type is: %s
",block_ptr->output[i]->name.type_names[(int)(block_ptr->output[i]->type)]);

    printf("This block accepts inputs from:
");
    for (i = 0; i < 4; i++)
        if (block_ptr->input[i] != NULL)
            printf("%d: %s Type is: %s
",i,block_ptr->input[i]->name.type_names[(int)(block_ptr->input[i]->type)]);
    printf("\n");
} while(1);

/* deleteclasses() Deletes a Flip-Flop's classes. */

void deleteclasses(struct classstr** head)
{
    struct classstr* temp_ptr;
    while ((*head) != NULL)
    {
        temp_ptr = *head;
        *head = (*head)->next;
        free((char*)temp_ptr);
    }
}

/* free_list() Frees the design list. */

void free_list(struct blockstr** head)
{
    struct blockstr* temp_ptr;
    while (((*head) != NULL))
    {
        #ifdef DEBUG2
        printf("Freeing list.\n");
        #endif
        temp_ptr = *head;
        (*head) = (*head)->next;
        if (((temp_ptr->type == LUT1) ||
            (temp_ptr->type == LUT2) ||
            (temp_ptr->type == LUT3) ||
            (temp_ptr->type == LUT4))
            free((char*)(temp_ptr->EQN));
            free((char*)(temp_ptr->Q0));
    }
}

/* getType_fromName() Returns the type of a block. */

enum block_type getType_fromName(struct blockstr* head, char* name)
{
    while (head != NULL)
    {
        if (!strcmp(head->name, name))
            return(head->type);
        head = head->next;
    }
}
findblocctype() Converts strings to block types.

enum block_type findblocctype(char* token)
{
    if (!strcmp(token, "FD", 2))
        return(FD);
    if (!strcmp(token, "IBUF"))
        return(IBUF);
    if (!strcmp(token, "OBUF_F_12"))
        return(OBUF);
    if (!strcmp(token, "LUT1"))
        return(LUT1);
    if (!strcmp(token, "LUT2"))
        return(LUT2);
    if (!strcmp(token, "LUT3"))
        return(LUT3);
    if (!strcmp(token, "LUT4"))
        return(LUT4);
    if (!strcmp(token, "BUFGP"))
        return(BUFGP);
    if (!strcmp(token, "VCC"))
        return(VCC);
    if (!strcmp(token, "GND"))
        return(GND);
}

insertclass() Adds a new node to a Flip-Flop's class_list

void insertclass(struct classstr **head, struct blockstr *newitem)
{
    struct classstr* temp;
    if (((*head) == NULL)
    {
        *head = (struct classstr*)malloc(sizeof(struct classstr));
        (*head)->next = NULL;
        (*head)->blockptr = newitem;
    }
    else
    {
        temp = (struct classstr*)malloc(sizeof(struct classstr));
        temp->blockptr = newitem;
        temp->next = *head;
        *head = temp;
    }
}

add_list() Adds a new block in the design.

void add_list(struct blockstr **head, struct blockstr *newitem)
{
    if (((*head) == NULL)
    {
        newitem->next = NULL;
        *head = newitem;
    }
    else
    {
        newitem->next = *head;
        *head = newitem;
    }
}

printlist() Prints the names of all the blocks in the design.

**********************************************************************************
void printlist(struct blockstr *head)
{
    int count=0;
    char c;
    printf("This is the list of blocks:\n");
    if (head != NULL)
    {
        do {
            printf(" %s\n",head->name);
    #ifdef DEBUG
        c = getchar();
        switch (head->type)
        {
            case LUT1:
            case LUT2:
            case LUT3:
            case LUT4:
                printf("EQN = %s\n",head->EQN);
                break;
            default:
                break;
        }
    #endif
        count++;
        head=head->next;
    } while(head != NULL);
    }
    printf("%d blocks in total.\n",count);
}

*****************************************************************************
get_token() strips any leading blank characters ("\t", " ", ...)
removes the end-of-line character (line was read using fgets)
and returns the first token in the line
*****************************************************************************
void get_token(char* line, char** token)
{
    char* templine;
    int i, j;

    line[strlen(line)-1]=0;
    templine=line;
    for (i=0; i<strlen(templine);i++)
        if (!isspace(templine[i]) || (templine[i]=='0'))
            break;

    if (templine[i]!='0')
    for (j=0; j<i; j++)
        templine++; /*token = strtok(templine, " ");*/
}

*****************************************************************************
nonresetblock() Checks whether we can move backward in the design.
*****************************************************************************
int nonresetblock(struct blockstr* block_ptr)
{
    int i;
    short int reset_input=0;
    short int nonff_input =0;
    for (i=0; i<4; i++)
    {
        if (block_ptr->input[i] != NULL)
        {
            if (!strcmp(block_ptr->input[i]->name,"C_reset"))
                reset_input=1;
            else if (!((block_ptr->input[i]->type == PD))
                nonff_input=1;
        }
    }
    return 0;
}
if ((reset_input==0) || (nonff_input))
    return(1);
return(0);
*
/** Starts visiting recursively the blocks behind a certain node.**/
void visit(struct blockstr* block_ptr, int level,
           int* logicblocksNo_ptr, struct classstr** chead)
{
    int i, j;

#define DEBUG3
    for (j=0; j<level; j++)
        printf(" ");
    printf("visit: Level-%d \%s\n", level, block_ptr->name);
#undef DEBUG3
    for (i=0; i<4; i++)
        if (block_ptr->input[i])
        {
#define DEBUG3
            for (j=0; j<level; j++)
                printf(" ");
            printf(" input#%d\n", i);
#undef DEBUG3
            if (block_ptr->input[i]->type==PD)
            {
#define DEBUG3
                for (j=0; j<level; j++)
                    printf(" ");
                printf(" reached FF: \%s\n", block_ptr->input[i]->name);
#undef DEBUG3
                else if (strcmp(block_ptr->input[i]->name, "C_reset") &&
                         nonresetblock(block_ptr->input[i]))
                    /* remove reset blocks*/
                    { /*logicblocksNo_ptr>++;
                       insertclass(chead, block_ptr->input[i]);
                       visit(block_ptr->input[i], level+1, logicblocksNo_ptr, chead);
                    }"*/
        }
}
*/

/***************************************************************************/
/* commonElements() Checks whether there are common blocks in two classes.*/
/***************************************************************************/
int commonElements(struct classstr* a, struct classstr* b)
{
    struct classstr* t1;
    struct classstr* t2;
    t1=a;
    t2=b;
    while(t1!=NULL)
    {
        while(t2!=NULL)
        {
            if (t1->blockptr == t2->blockptr)
                return(1);
            t2=t2->next;
        }
        t2=b;
        t1=t1->next;
    }
    return(0);
}
transform_eqn() Transforms LUT equations to a SIS compatible form.

void transform_eqn(struct blockstr* currblock, unsigned char helpmask)
{
    char eqnbuffer[500];
    char* readstr;
    char* writestr;
    int i, j;
    char nextchar;
    int namelen;
    char tempbuf[MAX_NAME_LEN];
    char c1;

    readstr = currblock->EQN;
    ifdef DEBUG2
    printf("EQN= %s\n", currblock->EQN);
    endif

    strcpy(eqnbuffer, currblock->name);
    writestr=&(eqnbuffer[strlen(eqnbuffer)]);
    *writestr++ = '=';
    *writestr=0;

    while (*readstr != 0)
    {
        nextchar = *readstr;
        readstr++;
        if (nextchar == 'I')
        {
            j = (*readstr) - '0';
            readstr++;
            if ((1<<j) & helpmask)
            {
                namelen = strlen(currblock->input[j]->name);
                if (currblock->input[j]->name[i]=='_')
                {
                    *writestr++ = 'c';
                    for (i=1; i<namelen; i++)
                        *writestr++ = currblock->input[j]->name[i];
                }
                else
                    for (i=0; i<namelen; i++)
                        *writestr++ = currblock->input[j]->name[i];
            }
        }
        else
        {
            sprintf(tempbuf,"i%d%s", j, currblock->name);
            namelen = strlen(tempbuf);
            for (i=0; i<namelen; i++)
                *writestr++ = tempbuf[i];
        }
        else
        {
            *writestr++ = nextchar;
        }
        *writestr=0;
    free((char*)currblock->EQN);
    currblock->EQN = malloc(strlen(eqnbuffer));
    strcpy(currblock->EQN, eqnbuffer);
}

identifyblocks() Identifies the Flip-Flops, assigns them to classes.
and creates the class files.

vecscount=0;
temphead = head;

while (temphead!=NULL)
{
    if (temphead->type == FD)
        vecscount++;
    temphead = temphead->next;
}

printf("The design has %d Flip Flops in total.\n", vecscount);

FF_classes = (struct FF_classes_str*)malloc(vecscount*sizeof(struct FF_classes_str));
temphead=head;
i=0;
while (temphead!=NULL)
{
    if (temphead->type == FD)
    {
        FF_classes[i].FF = temphead;
        i++;
    }
    temphead = temphead->next;
}

for (i=0; i<vecscount; i++)
{
    if (DEBUG) printf("%s\n", FF_classes[i].FF->name);
    FF_classes[i].marked=0; /*It hasn't been assigned any classType yet*/
    FF_classes[i].logicblocksNo = 0; /* no blocks inserted yet */
    FF_classes[i].class_list = NULL;
    FF_classes[i].class_processed = 0;
    FF_classes[i].classType = -1;
    FF_classes[i].next_inclass = NULL;
    if ( (FF_classes[i].FF->input[0] != NULL) &&
     (FF_classes[i].FF->input[0]->type != IBUF) &&
     (FF_classes[i].FF->input[0]->type != FD) )
    {
        FF_classes[i].logicblocksNo = 1;
     insertclass(&(FF_classes[i].class_list), FF_classes[i].FF->input[0]);
    }
    printf("Input 0: name= %s Type= %s\n", FF_classes[i].FF->input[0]->name,
     type_names[(int)FF_classes[i].FF->input[0]->type]);
}
/* visit FF's D-input */
visit(FF_classes[i].FF >input[0], 0,&(FF_classes[i].logicblocksNo),
&FF_classes[i].class_list);
#endif DEBUG3
printf("****************************************
\n");
#endif
}
#endif DEBUG3
for (i=0; i<FFScount; i++)
{
    printf("List of $d blocks connected to Flip Flop $s:\n", 
        FF_classes[i].logicblocksNo, FF_classes[i].FF->name);
    tempclassList = FF_classes[i].class_list;
    while (tempclassList!=NULL)
    {
        printf("$s\n", tempclassList->blockptr->name);
        tempclassList = tempclassList->next;
    }
    c = getchar();
}
#endif

currclassType=0;
for (i=0; i<FFScount; i++)
{
    #ifdef DEBUG3
    printf("Checking FF i=$d\n",i);
    #endif
    if (FF_classes[i].marked==0)
    {
        class_increment_already_cancelled = 0;
        first_back_join=1;
        FF_classes[i].classType=currclassType;
        tempclassList = FF_classes[i].class_list;
        while (tempclassList!=NULL)
        {
            tempclassList->blockptr->class_of_block = currclassType;
            tempclassList = tempclassList->next;
        }
        FF_classes[i].marked=1;
        tempFF_ptr = &(FF_classes[i]);
    }
    #ifdef DEBUG3
    printf("Not Marked! Assigning Type $d to FF $d\n",currclassType,i);
    #endif
    currclassType++;
    }
else
    continue;
for (j=0; j<FFScount; j++)
{
    if (i!=j)
    {
        if (commonElements(FF_classes[i].class_list,FF_classes[j].class_list))
        {
            #ifdef DEBUG3
            printf("Common els->$d with $d\n",i,j);
            #endif
            if (j<i)
            {
                #ifdef DEBUG3
                printf("j<i\n");
                #endif
                if (first_back_join==1)
                {
                    first_back_join=0;
                }
            }
        }
    }
}
if (class_increment_already_cancelled == 0)
{
    currclassType--; 
    ifndef DEBUG3
        printf("class type remains at %d\n",currclassType);
    endif
}
class_increment_already_cancelled = 1;
FF_classes[i].classType=FF_classes[j].classType;
tempclasslist = FF_classes[i].class_list;
while (tempclasslist!=NULL)
{
    tempclasslist->blockptr->class_of_block = FF_classes[j].classType;
    tempclasslist = tempclasslist->next;
}
    tempFF_ptr = &(FF_classes[i]);
    while (tempFF_ptr->next_inclass != NULL )
        tempFF_ptr = tempFF_ptr->next_inclass;
    tempFF_ptr->next_inclass = &(FF_classes[i]);
    tempFF_ptr = &(FF_classes[i]);
ifndef DEBUG3
        printf("Joining type->%d to %d (B)\n",tempFF_ptr->classType,i);
    endif
    else
{
    ifndef DEBUG3
        printf("join already made!!!\n");
    endif
}
else if (FF_classes[j].marked==0)
{
    ifndef DEBUG3
        printf("j>1\n");
    endif
    FF_classes[j].classType=FF_classes[i].classType;
    tempclasslist = FF_classes[j].class_list;
    while (tempclasslist!=NULL)
    {
        tempclasslist->blockptr->class_of_block = FF_classes[i].classType;
        tempclasslist = tempclasslist->next;
    }
    FF_classes[j].marked=1;
    /* connect all FFs in the same class */
    tempFF_ptr->next_inclass = &(FF_classes[j]);
ifndef DEBUG3
        printf("Joining %d to %d\n",i,j);
    endif
    tempFF_ptr = &(FF_classes[j]);
ifndef DEBUG3
        printf("Assigning last i temp_FF_ptr to %d\n",j);
    endif
}
    }
}
ifndef DEBUG3
    for (i=0; i<FFscount; i++)
    {
        printf("Type of Flip Flop %s: %d\n", FF_classes[i].FF->name, FF_classes[i].classType);
    }
endif
numClasses = currClassType; /* Total number of classes */
class_ptrArray = (struct FF_classes_str**)&malloc(numClasses*sizeof(struct
FF_classes_str*));
orig_num_eqns = (int*)malloc(numClasses*sizeof(int));

classes_processed=0;

#define DEBUG3
for (i=0; i<FFscount; i++)
{
    printf("List of %d blocks connected to Flip Flop %s:\n", FF_classes[i].logicblocksNo, FF_classes[i].FF->name);
    tempclassList = FF_classes[i].classList;
    while (tempclassList!=NULL)
    {
        printf("%s ClassType=%d\n", tempclassList->blockptr->name, tempclassList->
>blockptr->class_of_block);
        tempclassList = tempclassList->next;
    }
}
#endif

for (i=0; i<FFscount; i++)
{
#define DEBUG2
    printf("Checking i=%d\n",i);
#endif
    if (FF_classes[i].class_processed == 1)
        continue;
    else
    {
#define DEBUG2
        printf("Classes processed = %d\n",classes_processed);
#endif
        classes_processed++;
        sprintf(eqfilename,"class%d.eqn",FF_classes[i].classList);
        fp = fopen(eqfilename,"w");
        orig_num_eqns[FF_classes[i].classList]=0;
        class_ptrArray[FF_classes[i].classList]=tempFF_ptr = &(FF_classes[i]);
#endif
        printf("Starting at FF: %s\n",tempFF_ptr->FF->name);
#endif
        do {
#define DEBUG2
            printf("Processing FF: %s\n",tempFF_ptr->FF->name);
#endif
            tempclassList = tempFF_ptr->classList;
            while (tempclassList!=NULL)
            {
                currentblock=tempclassList->blockptr;
#endif
                printf("Currentblock: %s EQN:%d type:%s\n",currentblock->
>name,currentblock->class_processed.type_names((int)currentblock->type));
#endif
                if (((currentblock->eqn_processed == 0)
                &&
                ((currentblock->type == LUT1) ||
                (currentblock->type == LUT2) ||
                (currentblock->type == LUT3) ||
                (currentblock->type == LUT4))))
                {
                    helpmask = 0;
                    for (j=0; j<4; j++)
                        if ((currentblock->input[j]!=NULL) &&
                            (currentblock->class_of_block ==
                            currentblock->input[j]->class_of_block))
                            {helpmask|=1<<j;
```c
} transform_eqn(currentblock, helpmask);
fprintf(fp, "%s\n", currentblock->EQN);
#endif
#endif
#endif
#endif
currentblock->eqn_processed = 1;
orig_num_eqns[FF_classes[i].classType]++;

tempclasslist = tempclasslist->next;
} tempFF_ptr->class_processed = 1;
tempFF_ptr = tempFF_ptr->next_inclass;
while(tempFF_ptr != NULL);
fclose(fp);
}

for (i=0; i<FFscount; i++)
{
    if (FF_classes[i].class_processed == 0)
        printf("Error Detected at i=%d", i);
#endif
#endif
_declaration
insertcommand() Inserts a new command in the command list.

*****************************************************************************

void insertcommand(struct commandstr **commandlistptr, char command, int linen0, char* newitem)
{
    struct commandstr* temp;
    struct commandstr* t1;
    struct commandstr* t2;
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
```
t2=t2->next;
if (t2!=NULL)
{  
t1->next=temp;
   return;
}

while ((lineno > t2->lineno) & & (t2->next != NULL))
{  
t1 = t1->next;
   t2 = t2->next;
}

/* if inserting and deleting at the same line, lines are deleted first */
if (((lineno < t2->lineno) ||
    (lineno==t2->lineno & & command=="d"))
{
   temp->next = t2;
   t1->next = temp;
}
else
{
   temp->next = t2->next;
   t2->next = temp;
}
}

******************************************************************************
unmarkblocks() Resets the blocks' flag.
******************************************************************************
void unmarkblocks(struct blockstr *head)
{
   if (head != NULL)
   {
      do {
         head->flag=0;
         head=head->next;
      } while(head != NULL);
   }
}

******************************************************************************
extract_eqn_fromfile() Reads the equations from the out files,
creates the mapping and renaming tables for
every equation, and transforms them in an EDIF
compatible form.
******************************************************************************
void extract_eqn_fromfile(FILE* fp, struct outeqnstr** outeqnstr_dptr,
   int* eqn_num, int max_inst_label)
{
   int i,j,offset,curr_ptr_size;
   char linestr[MAX_STR_LEN];
   char c, nextchar;
   int tempsize,templen;
   char* tempstr;
   char* token;
   char* mappingTable[4];
   char* renameTable[4];
   int mptablesize,k;
   short int entryexists,neg_input;
   char* readstr;
   char* writesrl;
   char* writesr2;
   char* writesr3; /* to write the renamed-nodes equation */
   char* tempeq;
   int curr_inst_label;

```c
i=0;
do {
    fgets(linestr, MAX_STR_LEN, fp);
    if (linestr[strlen(linestr)-2]==':') /* each line ends with '::'/n' */
        i++;
}while(i<2); /* Skip first 2 lines */

curr_ptr_size=10; /* allocate space for 10 equations at first */
(*outeqnstr_dptr) = 
    (struct outeqnstr*)malloc(corr_ptr_size*sizeof(struct outeqnstr));
*eqn_num = 0;
offset=0;

while((c=fgetc(fp))! = EOF)
{
    if (isspace(c)) /* Ignore all spaces */
        continue;

    if (offset==0) /* We have to begin with a new equation */
        {
            (*outeqnstr_dptr)[*eqn_num].nodeType='n';
            else
                (*outeqnstr_dptr)[*eqn_num].nodeType='o';
            (*outeqnstr_dptr)[*eqn_num].eqn = malloc(20); /* allocate 20 chars/eqn */
            strcpy(*outeqnstr_dptr)[*eqn_num].eqn,"$$$$$$$$$$$$$$$$$$$$$";
            }
            tempsize=strlen(*outeqnstr_dptr)[*eqn_num].eqn;
            if (offset >= tempsize) /* Offset exceeds allocated size */
            {
                tempsize+=2;
                (*outeqnstr_dptr)[*eqn_num].eqn = realloc(*outeqnstr_dptr)[*eqn_num].eqn,tempsize);
                for (i=offset;i<tempsize;i++)
                    (*outeqnstr_dptr)[*eqn_num].eqn[i]='$';
                    (*outeqnstr_dptr)[*eqn_num].eqn[offset]=c;
                    offset++;
                    }
                    if (c==':') /* We have reached the end of the equation */
                    {
                        if (offset<tempsize) /* If eqn is smaller than what was allocated, */
                            /* truncate it to its size. */
                            (*outeqnstr_dptr)[*eqn_num].eqn = realloc(*outeqnstr_dptr)[*eqn_num].eqn,offset);
                            (*outeqnstr_dptr)[*eqn_num].eqn[offset]=0;
                            }
                            offset=0;
                            (*eqn_num)++; /* A new eqn begins */
                            if (*eqn_num >= curr_ptr_size)/* EQNs exceed num of allocated entries */
                            {
                                (*outeqnstr_dptr) = (struct outeqnstr*)realloc((char*)(*outeqnstr_dptr),
                                    2*curr_ptr_size*sizeof(struct outeqnstr));
                                    curr_ptr_size+=2; /*Readjusting ptr size to 2*curr_ptr_size*/
                                    }
                                    }
                                    if (*eqn_num < curr_ptr_size) /*Reallocated to space needed*/
                                    (*outeqnstr_dptr) = (struct outeqnstr*)realloc((char*)(*outeqnstr_dptr),
                                    (*eqn_num)*sizeof(struct outeqnstr));
                                    for (i=0; i<(*eqn_num); i++)
        {
            tempsize=strlen(*outeqnstr_dptr)[i].eqn;
```
for (j=0; j<tempsize; j++)
{
    c = (*outeqnstr_dptr)[i].eqn[j];
    if (c == '=
    (*outeqnstr_dptr)[i].blockname[j]=c;
    else
    { (*outeqnstr_dptr)[i].blockname[j]=0;
      memmove((*outeqnstr_dptr)[i].eqn,(*outeqnstr_dptr)[i].eqn+j+1,tempsize-j);
      break;
    }
}

} /* Construct mapping tables */
for (i=0; i<(*eqn_num); i++)
{
    maptablesiz=0;
    for (j=0; j<4; j++)
    {
        mappingTable[j]=(char*)(*outeqnstr_dptr)[i].mappingTable[j][0];
        renameTable[j]=(char*)(*outeqnstr_dptr)[i].mappingTable[j][1];
        strcpy(mappingTable[j],"*");
        strcpy(renameTable[j],"*");
    }
    tempstr=malloc(strlen((*outeqnstr_dptr)[i].eqn));
    strcpy(tempstr,(*outeqnstr_dptr)[i].eqn);
    token=strtok(tempstr,"*");*
    while (token!=NULL)
    {
        if (token[0]=='/') /*ignore '//'*/
            token++;
        entryexists=0;
        for (j=0; j<maptablesiz; j++)
        {
            if (!strcmp(token, mappingTable[j]))
                entryexists=1;
        }
        if (!entryexists) /* if this token is not in the table */
        {
            strcpy(mappingTable[maptablesiz], token);
            maptablesiz++;
        }
        token=strtok(NULL, "*"aise);
    } /*while token!=NULL*/

} /*I do not include '!' in the delimiters string for strtok() because
"...*!
" sequences in the equation would return NULL pointers!
I only want a NULL pointer to be returned at the end of the eqn.*/

sprintf( linesr,"LUT%d",maptablesiz);
(*outeqnstr_dptr)[i].type = findblocktype(linesr);

#ifdef DEBUG3
    printf("string type = %s Type returned = %d\n", linesr, (*outeqnstr_dptr)[i].type);
    for (j=0; j<maptablesiz; j++)
    {
        printf(" \
        t = \\
        c=getchar();
        free(tempstr);
    } /* Rename [xxx] (new) nodes */

curr_ins labels = max_ins labels + 1; /* avoid VCC node label conflicts*/
for (i=0; i<(*eqn_num); i++)
{
    if ((*outeqnstr_dptr)[i].nodeType == 'n')
```c
{ 
    printf(linestr,"C\%d*,curr_inst_label);
    for (j=0; j<(*eqn_num); j++)
    {
        for (k=0; k<4;k++)
        {
            mappingTable[k]=(char*)(*outeqns_dptr)[j].mappingTable[k][0];
            renameTable[k]=(char*)(*outeqns_dptr)[j].mappingTable[k][1];
        }
        maptablesizet = (int)(*outeqns_dptr)[j].type - (int)LUT1 +1;
        for (k=0; k<maptablesizet;k++)
        {
            if ((strncpy((*outeqns_dptr)[i].blockname,mappingTable[k]))
                strncpy(renameTable[k],linestr); /* rename node references */
        } /* j-loop */
        curr_inst_label++; /* a new node will be processed next */
        strncpy((*outeqns_dptr)[i].blockname,linestr); /* rename node */
    } /* i-loop */
}

#define DEBUG3

/* Print the rename and mapping tables */
/ * **************************** */
for (i=0; i<(*eqn_num); i++)
{
    for (j=0; j<4;j++)
    {
        mappingTable[j]=(char*)(*outeqns_dptr)[i].mappingTable[j][0];
        renameTable[j]=(char*)(*outeqns_dptr)[i].mappingTable[j][1];
    }
    printf("EQN \%d Mapping: ",i);
    for (j=0; j<4;j++)
    {
        printf("%s ",mappingTable[j]);
    }
    printf("\n Renaming: ");
    for (j=0; j<4;j++)
    {
        printf("%s ",renameTable[j]);
    }
    printf("\n");
} /* **************************** */

/* Construct instance equations */
for (i=0; i<(*eqn_num); i++)
{
    (*outeqns_dptr)[i].outeq1=malloc(2*strlen((*outeqns_dptr)[i].eqn));
    (*outeqns_dptr)[i].outeq2=malloc(2*strlen((*outeqns_dptr)[i].eqn));
    tempeqn = malloc(2*strlen((*outeqns_dptr)[i].eqn));
    for (j=0; j<4;j++)
    {
        mappingTable[j]=(char*)(*outeqns_dptr)[i].mappingTable[j][0];
        renameTable[j]=(char*)(*outeqns_dptr)[i].mappingTable[j][1];
    }
    maptablesizet = (int)(*outeqns_dptr)[i].type - (int)LUT1 +1;
    readstr=(*outeqns_dptr)[i].eqn;
    writestr1=(*outeqns_dptr)[i].outeq1;
    writestr2=(*outeqns_dptr)[i].outeq2;
    writestr3=tempeqn;
    *writestr1++=‘\’;
    *writestr2++=‘\’;
    *writestr3++=‘\’;
    neg_input=0;
    while ( (*readstr != ‘\’) )
    {
        nextchar = *readstr;
        if (nextchar == ‘\’)
        {
        
```
neg_input=1;
    *writestr2+='-';
    *writestr3+='!';
}
else if (nextchar == 'v')
{
    *writestr1+='v';
    *writestr2+='v';
    *writestr3+='v';
}
else if (nextchar == '+')
{
    *writestr1+='+';
    *writestr2+='+';
    *writestr3+='+';
}
else
{
    for (j=0; j<maptablesize; j++)
    {
        if (!strcmp(readstr,maptable[j].strlen(mapstr)))
        {
            *writestr1+='I';
            *writestr1+='0'+'j';
            if (neg_input==1)
            {
                *writestr1='\',
                neg_input=0; /* reset neg_input flag */
            }
            *writestr2+='I';
            *writestr2+='0'+'j';
            if (mapstr[0]=='j')
            {
                for (k=0; k<strlen(renameTable[j]); k++)
                    *writestr3+= renameTable[j][k];
            }
            else
            {
                for (k=0; k<strlen(mappingTable[j]); k++)
                    *writestr3+= mappingTable[j][k];
            }
            readstr+=strlen(mappingTable[j]) - 1;
            break;
        }
    } /* for every entry in the maptable */
/* else */
readstr++;
} /* while = ':' */

*writestr1='t';
*writestr2='t';
*writestr3='t';
*writestr2='\';
*writestr3='\';
*writestr3='0';
*writestr3='0';
*writestr3='0';

(*outeqnstr_dptr)[i].eqn=realloc((*outeqnstr_dptr)[i].eqn, strlen(tempeqn));
strcpy((*outeqnstr_dptr)[i].eqn, tempeqn);

#ifndef DEBUG
    printf("node=4d name=\%s\n eqn1= \%s \n eqn2= \%s \n eqn = \%s\n\n",i,*outeqnstr_dptr)[i].block,(*outeqnstr_dptr)[i].outeqn,(*outeqnstr_dptr)[i].ou
eqn2, (*outeqnstr_dptr)[i].eqn);
c=getchar();
#endif
free(tempseqn);
} /* for every equation */

.isTrueNotInNewFile() Checks whether an old node exists in the new equations.

*******************************************************************************/
int isTrueNotInNewFile(char* oldName, 
struct outeqnstr* outeqnstr_ptr,int eqn_num)
{
    int i;
    for (i=0; i<eqn_num; i++)
        if (!eqnstr_ptr[i].nameType=='G') 
          (strcmp(oldName,eqnstr_ptr[i].blockname))
            return(0);
    return(1); /* the node is Not in the New File */
}

*******************************************************************************/
replaceinstances() Finds out which instances need to be deleted and creates new instances to be inserted.

*******************************************************************************/
void replaceinstances(struct outeqnstr* outeqnstr_ptr,int eqn_num, 
struct blockstr* design, struct commandstr** commandlistptr, int classnum)
{
    struct classstr* tempclasslist;
    struct FF_classes_str* tempFF_ptr;
    struct blockstr* currentblock;
    int i;
    int PosToInsertNewNodes;
    char* instbuf;
    char c;

    unmarkblocks(design);
    tempFF_ptr = class_ptrArray[classnum];
do {
    tempclasslist = tempFF_ptr->class_list;
    while (tempclasslist!=NULL)
    {
        currentblock=tempclasslist->blockptr;
        if (currentblock->flag==0)
        {
            currentblock->flag=1; /* signal that node has been checked */
            if (!(!currentblock->type==LUT1) ||
                (currentblock->type==LUT2) ||
                (currentblock->type==LUT3) ||
                (currentblock->type==LUT4)) &&
            isTrueNotInNewFile(currentblock->name,eqnstr_ptr, eqn_num)
            { /* Delete instances & nets of nodes that do not exist any more */
                ifdef DEBUG2
                    printf("Deleting Old Node %s\n",currentblock->name);
                endif
                insertcommand(commandlistptr,'d',currentblock->lineinfile,""");
                insertcommand(commandlistptr,'d',currentblock->outnetinfile,""");
            }
        }
        tempclasslist = tempclasslist->next;
    }
    tempFF_ptr = tempFF_ptr->next_inclass;
}while(tempFF_ptr != NULL);
/* delete instances of old nodes */
/* create and insert new instances */
PosToInsertNewNodes = 0;
for (i=0; i<eqn_num; i++)
    if (outeqnstr_ptr[i].nodeType=='o')
    {
        getblockptr (design, &currentblock, outeqnstr_ptr[i].blockname);
        if (currentblock->lineinfile > PosToInsertNewNodes)
            PosToInsertNewNodes = currentblock->lineinfile;
    }
#endif
    printf("Replace instances: Max Pos Found =\d\n", PosToInsertNewNodes);
#endif
for (i=0; i<eqn_num; i++)
    if (outeqnstr_ptr[i].nodeType=='o')
    {
        getblockptr (design, &currentblock, outeqnstr_ptr[i].blockname);
        insertcommand (commandlistptr, 'd', currentblock->lineinfile, "*");
    }
    instbuf=malloc(250+4*strlen(outeqnstr_ptr[i].outeqn2));
    strcpy (instbuf, "*");
printf (instbuf, "\t (instance %s\n\t (viewRef Netlist_representation\n\t (cellRef LUT\d\n\t (LibraryRef VIRTEX\n\t )\n\t (property lut_function\n\t (string \"*s\")\n\t )\n\t (property INIT'\n\t (string \"004\")\n\t )\n\t (property EQN'\n\t (string \"*s\")\n\t )\n\t )\n\t , outeqnstr_ptr[i].blockname,
        ((int)outeqnstr_ptr[i].type - (int)LUT1 + 1),
        outeqnstr_ptr[i].outeqn1,
        outeqnstr_ptr[i].outeqn2);
    /* if inserting and deleting at the same line, lines are deleted first */
    if (outeqnstr_ptr[i].nodeType=='o')
        insertcommand (commandlistptr, 'i', currentblock->lineinfile, instbuf);
    else
        insertcommand (commandlistptr, 'i', PosToInsertNewNodes, instbuf);
free(instbuf);
}

/***************************************************************************/

insert_modified_net() Inserts a new node in the modified-nets list.
/***************************************************************************/
void insert_modified_net(struct mnetslist** listptr, 
                    struct blockstr* mnetblock) 
{
    struct mnetslist* newitem;
    struct mnetslist* temp_ptr;
    char c;

    temp_ptr = (*listptr);
    while(temp_ptr != NULL)
    {
        if (temp_ptr->blockptr == mnetblock)
#ifndef DEBUG
  printf("Block %s already in the modified list\n", temp_ptr->blockptr->name);
#endif
  return;
}

#endif

/* Did not find block in the list. We must insert it now. */
temp_ptr = (struct mnetslist*)malloc(sizeof(struct mnetslist));
temp_ptr->next = (*listptr);
temp_ptr->blockptr = mnetblock;
(*listptr) = temp_ptr;
#endif DEBUG2
printf("Inserted block %s in the modified list\n", (*listptr)->blockptr->name);
#endif

/******************************************************************************
 free_mnet_list() Frees the modified-nets list.
 ******************************************************************************/
void free_mnet_list(struct mnetslist** listptr)
{
  struct mnetslist* temp_ptr;
  char c;
  #ifdef DEBUG2
  printf("Deleting Modified Nets List:\n");
  #endif
  while( (*listptr) != NULL )
  {
    temp_ptr = (*listptr);
    (*listptr) = (*listptr)->next;
    #ifdef DEBUG2
    printf("%s ", temp_ptr->blockptr->name);
    #endif
    free((char*)temp_ptr);
  }
  #ifdef DEBUG2
  printf("\n");
  c=getchar();
  #endif
}

/******************************************************************************
 replacenets() Finds the nets that need to be deleted, replaced and
 creates those that need to be inserted.
 ******************************************************************************/
void replacenets(struct outeqnstr* outeqnstr_ptr, int eqn_num,
  struct blockstr* design, struct commandstr** commandlistptr,
  int classnum, int max_net_label)
{
  struct classstr* tempclasslist;
  struct FF_classes_str* tempFF_ptr;
  struct blockstr* currentblock;
  struct blockstr* oldclassblock;
  struct blockstr* feedingblock;
  int i,j,k,t,s,tablesize;
  char* netbuf;
  char c;
  char tempchar;
  char tempname[MAX_NAME_LEN];
  struct mnetslist* modifiednets;
  struct mnetslist* temp_modified;
  short int old_block_matched=0;
  short int found_empty_entry=0;
  struct net_entry_node* netentrylist;
  struct net_entry_node* tnetentryptr;
  int curr_net_label;
  int count_net_entries;
  struct ports_table* t_ports;

int t_ports_num;
netentrylist=NULL;
modifiednets=NULL;

currentblock=design;
while (currentblock!=NULL)
{
    if (!strcmp(currentblock->name,"C_in",4))
    {
        ifdef DEBUG2
        printf("%s Block found\n",currentblock->name);
        =endif

        for (k=0; k<currentblock->outputsNum+1;k++)
        {
            if (currentblock->ports_ptr[k].status=="f")
            {
                unmarkblocks(design);
                old_block_matched=0;
                tempFF_ptr = class_ptrArray[classnum];
                do {
                    tempclasslist = tempFF_ptr->class_list;
                    while (tempclasslist!=NULL)
                    {
                        oldclassblock=tempclasslist->blockptr;
                        if (oldclassblock->flag==0)
                        {
                            /* signal that node has been checked */
                            oldclassblock->flag=1;
                            if (!strcmp(oldclassblock->name,
                                           currentblock->ports_ptr[k].name))
                            {
                                ifdef DEBUG2

                                printf("k%d Port= %s Name= %s status: %c\n",k,
                                           currentblock->ports_ptr[k].port,
                                           currentblock->ports_ptr[k].name,
                                           currentblock->ports_ptr[k].status);
                                printf("Matched! Type= LUT%d\n",(int)oldclassblock->type-"LUT",(int)LUT1+1);
                                =endif
                                old_block_matched=1;
                                break;
                            }
                            }/* if old block has not been checked yet */
                            tempclasslist = tempclasslist->next;
                    }
                    tempFF_ptr = tempFF_ptr->next_inclass;
                }while((tempFF_ptr != NULL) && (old_block_matched==0));

                if ((old_block_matched == 1) && (oldclassblock!=currentblock))
                {
                    ifdef DEBUG2
                    printf("Deleting Entry %d matched to old block %s!\n",k,oldclassblock->
                                           name);
                    =endif
                    insert_modified_net(&modifiednets, currentblock);
                    currentblock->ports_ptr[k].status='e';
                }
            }/* if net entry was full */
        }/* for every net entry */
    }/* if currentblock is a C_in block */
currentblock=currentblock->next;
    }/* while currentblock is not NULL */

for (i=0; i<eqn_num; i++)
{

```c
#define DEBUG2

printf("size =\d EQN \$d \&\$s =
%s", tablesiz e, i, outeqnstr_ptr[i].blockname, outeqnstr_ptr[i].eqn);
#endif

#define DEBUG2

printf("Mapping Table:\n");
for (j=0; j<tablesiz e; j++)
    printf("%s ", outeqnstr_ptr[i].mappingTable[j][0]);
printf("\nRenaming Table:\n");
for (j=0; j<tablesiz e; j++)
    printf("%s ", outeqnstr_ptr[i].mappingTable[j][1]);
printf("\n");
#endif

for (j=0; j<tablesiz e; j++)
    {switch(outeqnstr_ptr[i].mappingTable[j][0][0])
        case 'i':
        {#ifdef DEBUG2
                printf("First Letter -> i\n");
            #endif
                getblockptr (design, &currentblock, (outeq nstr_ptr[i].mappingTable[j][0]+2));
                feedingblock = currentblock->input [(outeq nstr_ptr[i].mappingTable[j][0][1] - '0')];
        #ifdef DEBUG2
                printf("Block's %s input %d connected to Feeding Block B: %s\n", currentblock->name, (outeq nstr_ptr[i].mappingTable[j][0][1]-'0'), feedingblock->name);
                printf("B's net name = %s at line: %d\n", feedingblock->netName, feedingblock->outnetinline);
            #endif
        }
        for (k=0; k<feedingblock->outputsNum+1;k++)
            {if (!strcmp(feedingblock->ports_ptr[k].name, currentblock->name) &&           (feedingblock->ports_ptr[k].port[1] == outeqnstr_ptr[i].mappingTable[j][0][1]) &&            (feedingblock->ports_ptr[k].status == 'f') )
                    {feedingblock->ports_ptr[k].status = 'e';
        #ifdef DEBUG2
                    printf("Referenced Block (%s) at net entry k=%d of %s is deleted:\n", currentblock->name, k, feedingblock->name);
            #endif
            }
        }
    }

found_empty_entry = 0;
for (k=0; k<feedingblock->outputsNum+1;k++)
    {if (( feedingblock->ports_ptr[k].status == 'e')
            {#ifdef DEBUG2
                printf("Inserting %s to empty entry%#d (overwriting %s)\n", outeqnstr_ptr[i].blockname, k, feedingblock->ports_ptr[k].name);
            #endif
                strcpy(f eedingblock->ports_ptr[k].name, outeqnstr_ptr[i].blockname);
                sprintf(temppname, "%ld", j);
                strcpy( feedingblock->ports_ptr[k].port, temppname);
                feedingblock->ports_ptr[k].status = 'f';
        #ifdef DEBUG2
                printf("Filling Entry %#d -> name=%s port=%s status=%c\n", k, feedingblock->ports_ptr[k].name,
```
feedingblock->ports_ptr[k].port,
feedingblock->ports_ptr[k].status;
#endif

    found_empty_entry=1;
    break;
}
}

if (found_empty_entry==0)
{
    feedingblock->ports_ptr = (struct
ports_table*)realloc((char*)feedingblock->ports_ptr,(2*feedingblock->outputsNum+1)*sizeof(struct ports_table));
    for (k=feedingblock->outputsNum+1; k<2*feedingblock->outputsNum+1;k++)
    {
        strcpy(feedingblock->ports_ptr[k].port,"");
        strcpy(feedingblock->ports_ptr[k].name,"*****");
        feedingblock->ports_ptr[k].status='e';
    }
    strcpy(feedingblock->ports_ptr[feedingblock->outputsNum+1].name,
outeqnstr_ptr[i].blockname);
    strcpy(feedingblock->ports_ptr[feedingblock->outputsNum+1].port,tempname);
    strcpy(feedingblock->ports_ptr[feedingblock->outputsNum+1].status,'f';
    feedingblock->outputsNum+=2;
#endif DEBUG2
    printf("%s's net is full. Expanding now to %d
entries...\n",feedingblock->name,feedingblock->outputsNum+1);
    printf("Filling Entry %d-> name=%s port=%s status=%c\n",feedingblock->outputsNum/2+1,
    feedingblock->ports_ptr[feedingblock->outputsNum/2+1].name,
    feedingblock->ports_ptr[feedingblock->outputsNum/2+1].port,
    feedingblock->ports_ptr[feedingblock->outputsNum/2+1].status);
#endif DEBUG2
}

for (k=0; k<feedingblock->outputsNum+1;k++)
{
#endif DEBUG2
    printf("%d Port= %s Name= %s status: %c\n",k,
    feedingblock->ports_ptr[k].port,
    feedingblock->ports_ptr[k].name,
    feedingblock->ports_ptr[k].status);
#endif DEBUG2
    if ((k) && (k%15==0))
    c=getchar();
#endif DEBUG3

    insert_modified_net(&modifiednets,feedingblock);

    break;
#endif DEBUG2
    printf("First Letter -> c\n");
#endif DEBUG2
    strcpy(tempname,(outeqnstr_ptr[i].mappingTable[j][0]));
    tempname[0]='C'; /* convert c_in_1_x name to C_in_1_x */
    getblockptr(blockx,blockptr,tempname);
#endif DEBUG2
    printf("Input Block %s \n",currentblock->name);
#endif DEBUG2
    found_empty_entry=0;
    for (k=0; k<currentblock->outputsNum+1;k++)
    {
        if (currentblock->ports_ptr[k].status=='e')
        {
            printf("Empty entry%#d Port= %s Name= %s status: %c\n",k,
currentblock->ports_ptr[k].port,
currentblock->ports_ptr[k].name,
currentblock->ports_ptr[k].status);
#endif
strcpy(currentblock->ports_ptr[k].name, outeqnstr_ptr[i].blockname);

sprintf(currentblock->ports_ptr[k].port, "%d", j);
currentblock->ports_ptr[k].status = 'f';
#endif DEBUG2
printf("Filled entry $d Port $s Name $s status: $c\n", k,
currentblock->ports_ptr[k].port,
currentblock->ports_ptr[k].name,
currentblock->ports_ptr[k].status);
#endif DEBUG2

found_empty_entry=1;
break;
}
}

if (!found_empty_entry)
{
currentblock->ports_ptr = (struct ports_table*)realloc((char*)currentblock->ports_ptr, (2*currentblock->outputsNum+1)*sizeof(struct ports_table));
for (k=0; k<currentblock->outputsNum+1; k++)
{
    strcpy(currentblock->ports_ptr[k].port, "***");
    strcpy(currentblock->ports_ptr[k].name, "*****");
    currentblock->ports_ptr[k].status = 'e';
}
strcpy(currentblock->ports_ptr[currentblock->outputsNum+1].name,
      outeqnstr_ptr[i].blockname);
printf(tempname, "%d", j);
strcpy(currentblock->ports_ptr[currentblock->outputsNum+1].port,
      tempname);
currentblock->ports_ptr[currentblock->outputsNum+1].status = 'f';
currentblock->outputsNum++;
#endif DEBUG2

for (k=0; k<currentblock->outputsNum+1; k++)
{
    if ((k) & (k*15==0))
        c=getchar();
#endif DEBUG3
insert_modified_net(&modifiednets, currentblock);
break;
 case 'C':
#endif DEBUG2
    printf("First Letter -> C\n");
#endif DEBUG2
getblockptr(design, &currentblock, (outeqnstr_ptr[i].mappingTable[j][0]));
if (currentblock->entries_removed == 0)
{
    for (k=0; k<currentblock->outputsNum+1; k++)
    {
        if (currentblock->ports_ptr[k].status == 'f')
        {
            unmarkblocks(design);
            old_block_matched=0;
            tempFF_ptr = class_ptrArray[classnum];
            do {
                tempclassList = tempFF_ptr->class_list;
                while (tempclassList != NULL)
                {
                    oldclassblock=tempclassList->blockptr;
                    if (oldclassblock->flag == 0)
                    {
                        /* signal that node has been checked */
                        oldclassblock->flag = 1;
                        if (strcmp(oldclassblock->name,
                                    currentblock->ports_ptr[k].name))
                        {
                            #ifdef DEBUG2
                            printf("k=%d Port= %s Name= %s status: %c\n", k,
                                    currentblock->ports_ptr[k].port,
                                    currentblock->ports_ptr[k].name,
                                    currentblock->ports_ptr[k].status);
                            printf("Matched! Type= LUT%d\n", (int)oldclassblock->
                                    >type-(int)LUT1+1));
                            #endif
                            old_block_matched = 1;
                            break;
                        }
                    }/* if old block has not been checked yet */
                    tempclassList = tempclassList->next;
                }while((tempFF_ptr != NULL) && (old_block_matched == 0));
                #ifdef DEBUG2
                printf("Deleting Entry %d matched to old block
                        %s\n", k, oldclassblock->name);
                #endif
                insert_modified_net(&modifiednets, currentblock);
                currentblock->ports_ptr[k].status = 'e';
            }
        }/* if net entry was full */
    }/* for every net entry k-loop*/
for (k=0; k<currentblock->outputsNum+1; k++)
{
    if (currentblock->ports_ptr[k].status == 'e')
    {
        strcpy(currentblock->ports_ptr[k].name, outeqnstr_ptr[i].blockname);
        sprintf(currentblock->ports_ptr[k].port, "%d", j);
        currentblock->ports_ptr[k].status = 'f';
        #ifdef DEBUG2
        printf("Filled entry%td Port= %s Name= %s status: %c\n", k,
                currentblock->ports_ptr[k].port,
                currentblock->ports_ptr[k].name,
                currentblock->ports_ptr[k].status);
        #endif
        found_empty_entry = 1;
    }
break;
}
} /* for every net entry k-loop */
if (*found_empty_entry==0)
{
  currentblock->ports_ptr = (struct ports_table*)realloc((char*)currentblock->ports_ptr, (*2*currentblock->outputsNum+1)*sizeof(struct ports_table));
  for (k=currentblock->outputsNum+1; k<2*currentblock->outputsNum+1; k++)
  {
    strcpy(currentblock->ports_ptr[k].port, "***");
    strcpy(currentblock->ports_ptr[k].name, "****");
    currentblock->ports_ptr[k].status='e';
  }
  strcpy(currentblock->ports_ptr[currentblock->outputsNum+1].name, outeqnstr_ptr[i].blockname);
  printf(tempname, "1\d", j);
  strcpy(currentblock->ports_ptr[currentblock->outputsNum+1].port, tempname);
  currentblock->ports_ptr[currentblock->outputsNum+1].status='f';
  currentblock->outputsNum+=2;
}

#define DEBUG2
printf("%s's net is full. Expanding now to \d entries...\n", currentblock->name, currentblock->outputsNum+1);
printf("Filled entry=%d Port= %s Name= %s status: %c\n", currentblock->outputsNum/2+1, currentblock->ports_ptr[currentblock->outputsNum/2+1].port, currentblock->ports_ptr[currentblock->outputsNum/2+1].name, currentblock->ports_ptr[currentblock->outputsNum/2+1].status);
#endif
} /* if there was no empty entry */

insert_modified_net(&modifiednets, currentblock);
break;

  case ',':
  #ifdef DEBUG2
    printf("First Letter -> [\n];
    printf("New node: %s", outeqnstr_ptr[i].mappingTable[j][1]);
  #endif
    break;
  default:
    printf("Unknown Letter \n");
    break;
  } /* switch statement */
} /* for i-loop */
} /* for j-loop */

curr_net_label = max_net_label + 1;
for (i=0; i<eqn_num; i++)
{
  if (outeqnstr_ptr[i].nodeType=='n')
  {
    #ifdef DEBUG2
      printf("Examining Equation \d instance: %s\n", i, outeqnstr_ptr[i].blockname);
    #endif
    netentrylist = NULL;
    netentryptr = NULL;
    for (t=0; t<eqn_num; t++)
    {
      tablesiz = (int)outeqnstr_ptr[t].type - (int)LUT1 -1;
      for (s=0; s<tablesiz; s++)
      {
        if (!strcmp(outeqnstr_ptr[i].blockname,  
            outeqnstr_ptr[t].mappingTable[s][1]))
        {
          #ifdef DEBUG2
            printf("Node is feeding input \d of node %s\n", s,  
                outeqnstr_ptr[t].blockname);
          #endif
```
tnetentryptr =
(struct net_entry_node*)malloc(sizeof(struct net_entry_node));

sprintf(tnetentryptr->port, "%d", s);
strcpy(tnetentryptr->name, outeqnstr_ptr[t].blockname);
tnetentryptr->next=netentrylist;
netentrylist=tnetentryptr;
}
} /* t-loop checking renaming tables of all equations */
count_net_entries=0;
tnetentryptr=netentrylist;
while(tnetentryptr)
{
    count_net_entries++;
}
#endif DEBUG2
    printf("port= %s Name= %s \n", tnetentryptr->port, tnetentryptr->name);
#endif
    tnetentryptr=tnetentryptr->next;
}
#endif DEBUG2
    printf("%d Net entries found in %s's
net\n", count_net_entries, outeqnstr_ptr[i].blockname);
#endif

netbuf = malloc(40 + (count_net_entries+1)*85);
strcpy(netbuf,"\n\t (net sym$\\n\t (joined\"\n\t curr_net_label);

tnetentryptr=netentrylist;
while(tnetentryptr)
{
    printf(netbuf,"%s\n\t (portRef %s\n\t (instanceRef %s)\n\t )\n", netbuf, tnetentryptr->port, tnetentryptr->name);
    tnetentryptr=tnetentryptr->next;
}
    printf(netbuf,"%s\n\t (portRef O\n\t (instanceRef %s)\n\t )\n\t )\n", netbuf, outeqnstr_ptr[i].blockname);
#endif DEBUG2
    printf("Net created: %s\n", netbuf);
    printf("Using buffer of length: %d\n", strlen(netbuf));
#endif
insertcommand(commandlistptr, 'i', max_net_pos, netbuf);
free(netbuf);
while(netentrylist)
{
    tnetentryptr=netentrylist;
    netentrylist=netentrylist->next;
    free((char*)tnetentryptr);
}
curr_net_label++;
} /* if we have a new node [NN] */
} /* for i-loop */
#endif DEBUG2
printf("Modified Nets List:\n");
#endif
temp_modified = modifiednets;
while (temp_modified)
{
#endif DEBUG2
printf("Block: \$s Type: \$s Net type: \$d Net Name: \$s line \n", temp_modified->blockptr->name, type_names[(int)temp_modified->blockptr->type], temp_modified->blockptr->netType, temp_modified->blockptr->netName, temp_modified->blockptr->outnetinfile);
#endif

t_ports = temp_modified->blockptr->ports_ptr;
t_ports_num = temp_modified->blockptr->outputsNum + 1;
count_net_entries = 0;
for (i=0; i<t_ports_num; i++)
{
  ifdef DEBUG2
    printf("\$d \$s - \$s status: \$c\n", i, t_ports[i].port, t_ports[i].name, t_ports[i].status);
  endif
  if (t_ports[i].status == 'f')
    count_net_entries++;
}

ifdef DEBUG2
  printf("There are \$d non-empty entries.\n",count_net_entries);
endif

netbuf = malloc((60 + (count_net_entries+1)*85));
strcpy(netbuf,"\n");
switch(temp_modified->blockptr->netType)
{
  case 1:
  case 2:
  case 5:
    printf(netbuf, "\t (net:\n"
    sprintf(netbuf, "%s\t (rename \$s)\n", netbuf, temp_modified->blockptr->netName);
    break;
    
    default:
    sprintf(netbuf, "\t (net: \$s)\n", temp_modified->blockptr->netName); /*synXXX*/
    break;
} /* switch net Type statement */
    printf(netbuf, "%s\t (joined)\n", netbuf);
#endif DEBUG2
  printf("Net so far: \n\n\n", netbuf);
#endif

for (i=0; i<t_ports_num; i++)
{
  ifdef DEBUG2
    printf("\$s\n\t (portRef \$s)\n\t (instanceRef \$s)\n\t )\n", netbuf, t_ports[i].port, t_ports[i].name);
  endif
    sprintf(netbuf, "%s\n\t )\n\t )\n", netbuf);
#endif DEBUG2
  printf("Net created: \n\n\n", netbuf);
#endif DEBUG2
  printf("Using buffer of length: \$d\n", strlen(netbuf));
#endif

insertcommand(commandlistptr,'d', temp_modified->blockptr->outnetinfile, ");
insertcommand(commandlistptr, 'i', temp_modified->blockptr->outnetinfile, netbuf);

    free(netbuf);
    temp_modified = temp_modified->next;
}
free_mnet_list(&modifiednets);
}

/******************************************************************************/
/*computechanges() Opens out files and computes all the necessary changes */
/*to the netlist.*/
/******************************************************************************/
void computechanges(struct commandstr**, commandlistptr, struct blockstr* design,
        int max_inst_label, int max_net_label)
{
    FILE* fp;
    char outeqnfilename[20];
    struct outeqnstr** outeqnstr_ptr;
    int i, classnum;
    int* eqn_num;
    char tempchar;
    struct classstr* tempclasslist;
    struct FF_classes_str* tempFF_ptr;
    struct blockstr* currentblock;
    char c;

    #ifdef DEBUG
    struct commandstr* t1;
    #endif

    max_inst_label = max_inst_label + 5; /* avoid VCC node label conflicts*/
    outeqnstr_ptr = (struct outeqnstr**) malloc(numclasses*sizeof(struct outeqnstr*));
    eqn_num = (int*)malloc(numclasses*sizeof(int));

    for (classnum=0; classnum < numClasses; classnum++)
    {
        sprintf(outeqnfilename,"out%d.eqn",classnum);
        fp = fopen(outeqnfilename,"w");
        extract_eqn_fromfile(fp, &eqnstr_ptr[classnum], &eqn_num[classnum],
        max_inst_label);
        if (eqn_num[classnum]<orig_num_eqns[classnum])
        {
            /* delete old or non-existent instances, create & insert new ones */
            replaceinstances(outeqnstr_ptr[classnum], eqn_num[classnum], design, commandlistptr,
            classnum);
            #ifdef DEBUG
            printf("Replacing Nets:\n\n");
            #endif
            replacenets(outeqnstr_ptr[classnum], eqn_num[classnum], design, commandlistptr,
            classnum, max_net_label);
        }
        else
        {
            #ifdef DEBUG
            printf("Sorry, cannot optimize Class %d\n",classnum);
            #endif
        }
        fclose(fp);
    }

    #ifdef DEBUG2
    printf("---\n");
    for (classnum=0; classnum < numClasses; classnum++)
    {
        printf("%d EQNS recognized:\n",eqn_num[classnum]);
        for(i=0; i<eqn_num[classnum]; i++)
        {
print("i=%d Block= %s", i, outeqnstr_ptr[classnum][i].blockname);
print(" %s\n", outeqnstr_ptr[classnum][i].eqn);
}
tempchar=getchar();
}

/* Access All Blocks of a Class */
for (classnum=0; classnum < numClasses; classnum++)
{
    printf("\nChecking Class "%d\n",i);
tempFF_ptr = class_ptrArray[classnum];
do {
    tempclasslist = tempFF_ptr->class_list;
    while (tempclasslist!=NULL)
    {
        currentblock=tempclasslist->blockptr;
        printf("%s ",currentblock->name);
        tempclasslist = tempclasslist->next;
    }
    tempFF_ptr = tempFF_ptr->next_inclass;
}while(tempFF_ptr != NULL);
tempchar=getchar();
}
printf("\n");
#endif

createneeddef() Creates the new EDIF netlist by applying the commands
that have been recorded in the commandlist.
*******************************************************************************/
void createneeddef(struct commandstr* commandlist)
{
    FILE* in_fp;
    FILE* out_fp;
    char linesstr[MAX_STR_LEN];
    char temp;
    int lineno,i.skippedlines,prev_lineno;
    short int del_at_lineno;
    struct commandstr* t1;
    int parenth_bal;
    char c;

    in_fp = fopen(FILE_NAME, "r");
    if (!in_fp){ printf("Error opening %s\n", FILE_NAME);
        return; }
    out_fp = fopen(OUTFILE,"w");
    if (!out_fp){ printf("Error opening %s\n", OUTFILE);
        return; }
#endif

t1 = commandlist;
printf("\ncreateneeddef Command LIST:\n");
while(t1!=NULL)
{
    printf("command: %c lineno:%d\n",t1->command, t1->lineno);
    t1=t1->next;
    c = getchar();
}
printf("Inserting:n$s\n", t1->newitem);
t1=t1->next;
c = getchar();
}
#endif

t1 = commandlist;
prev_lineno=-1;
del_at_lineno=0;
lineno=0;
while(fgets(linestr, MAX_STR_LEN, in_fp))
{
    lineno++;
    if ((t1!=NULL) && (lineno==t1->lineno))
    {
        skippedlines=0;
        do {
            switch (t1->command){
                case 'd':
                    printf("Deleting at line:%d\n",lineno);
                    del_at_lineno=1;
parenth_bal=0;
do {
                        for (i=0; i<strlen(linestr); i++)
                        {
                            if (linestr[i]=='(')
                                parenth_bal++;
                            else if (linestr[i]==')')
                                parenth_bal--;
                        }
                        if (parenth_bal==0)
                        {
                            fgets(linestr, MAX_STR_LEN, in_fp);
skippedlines++;
                        }
} while (parenth_bal==0);
                    printf("%d lines were skipped\n",skippedlines);
                    break;
                case 'i':
                    /*strtok will destroy the char buffer t1->newitem.
                    Work on a copy if you need its contents later*/
                    printf("Inserting at line:%d\n",lineno);
                    fputs(out_fp,"$s",t1->newitem);
                    /* If we had no deletion at the same line (skippedlines>0) then
                    insert the first line of the next item located at lineno
                    in the original file */
                    /*There was no deletion or insertion at lineno */
                    if (((del_at_lineno==0) && (t1->next==NULL)) ||
                        ((del_at_lineno==0) && (t1->next->lineno != lineno))
                    )
fputs(linestr, out_fp);
                    break;
                } /* end of switch statement */
            prev_lineno = lineno;
t1 = t1->next;
        } while ((t1!=NULL) && (t1->lineno == lineno));
    lineno += skippedlines;
del_at_lineno=0;

    else
        fputs(linestr, out_fp);
}
fclose(in_fp);
close(out_fp);
}

="/*********************************************************************************/
main() Parses the EDIF netlist and converts it to an optimized one.
*********************************************************************************/
void main()
{
FILE* fp;
char linestr[MAX_STR_LEN];
char* token;
char* temp;
char name[MAX_NAME_LEN];
struct blockstr* design=NULL;
struct blockstr* tempblock=NULL;
int eqn_found = 0;
char prev_char='*';
struct ports_table* ports_ptr=NULL;
int tablesize;
short int valid_net=1;
char port_name[MAX_PORT_NAME_SZ];
int i;
char c;
int count_nets=0;
int lineno, last_instance_line, last_net_line;
struct commandstr* commandlist=NULL;
int temp_label;
int max_inst_label=0;
int max_net_label=0;
char netname[100];
short int rename_net=0; /*0-->net has no renaming
 1-->net has a rename line */

max_net_pos = 0;

fp = fopen(FILE_NAME, "r");
if (!fp)
{
 printf("Can not open file %s\n", FILE_NAME);
 exit(8);
}

ports_ptr = (struct ports_table*)malloc(MAX_PORTTABLE_SZ*sizeof(struct ports_table));

lineno=0;
while (fgets(linestr, MAX_STR_LEN, fp))
{
  lineno++;
  get_token(linestr,&token);

  if (!strcmp(token, "((instance")))
  {
    last_instance_line = lineno;
    temp = strtok(NULL, " ");

    /* If it is a flip-flop (FF), there will be a renaming on the
    next line, otherwise temp will be NULL (this is the case
    for LookUp Tables (LUTS)). However the presence or not of
    the renaming line don't say anything about the existence of
    blocks other than FFs or LUTS */

    if (temp!=NULL)
      strcpy(name,temp);

    fgets(linestr, MAX_STR_LEN, fp);
"
lineno++;
get_token(linestr,&token);

if (!strcmp(token, "rename")
{
    token=strtok(NULL," ");
tempblock = (struct blockstr*)malloc(sizeof(struct blockstr));
strcpy(tempblock->name, token);
fgets(linestr, MAX_STR_LEN, fp); /* Bypass ViewRef line */
lineno++;
fgets(linestr, MAX_STR_LEN, fp);
lineno++;
get_token(linestr,&token);

if (!strcmp(token, "cellRef")
{
    token=strtok(NULL," ");
}
else
{
    printf("Error! cellRef expected instead of:%s line:%d\n.token.lineno");
exit(1);
}
}
else
{
    printf("Possible Flip-Flop Detected! Name=%s Type=%s\n",tempblock->name.type_names[1],tempblock->type_names[1]);
flush(stdout);
}

ifdef DEBUG2
printf("Block %s Detected! Name=%s\n",token,name);
flush(stdout);
endif

tempblock = (struct blockstr*)malloc(sizeof(struct blockstr));
strcpy(tempblock->name, name);
tempblock->type = findblocktype(token);
if (tempblock->type == LUT1) ||
    (tempblock->type == LUT2) ||
    (tempblock->type == LUT3) ||
    (tempblock->type == LUT4)
{
    /* hold the name of the last instance found*/
    scanf(name,"C%ld",&temp_label);
    if (temp_label>max_inst_label)
        max_inst_label=temp_label;

eqn_found = 0;
do {
    fgets(linestr, MAX_STR_LEN, fp);
    lineno++;
    get_token(linestr,&token);
    if (!strcmp(token, "(property")
{ 
    token=strtok(NULL, " "); /* read the type of property */
    /*Use EQN or lut_function*/
    if (!strcmp(token, "lut_function"))
    {
        fgets(linestr, MAX_STR_LEN, fp);
        lineno++;
        get_token(linestr,&token);
        token=strtok(NULL, "," ); /* read the EQN */
        token++;
        token[strlen(token)-1]=':'; /* get rid of parentheses*/
        eqn_found=1;
        tempblock-> EQN = malloc(strlen(token));
        strcpy(tempblock-> EQN, token);
    }
}
}

while (!eqn_found):
    if ((tempblock->type == GND) ||
        (tempblock->type == VCC))
    {
        /* hold the name of the last instance found*/
        sscanf(name, "C\d", &temp_label);
        if (temp_label>max_inst_label)
            max_inst_label=temp_label;
    }
    for (i=0; i<4; i++)
        tempblock->input[i] = NULL;
    tempblock->outputsNum=0;
    tempblock->lineinfile = last_instance_line;
    tempblock->eqn_processed=0;
    tempblock->entries_removed=0;
    tempblock->class_of_block=-1;
    tempblock->next=NULL;
    add_list(&design, tempblock);
}
else if (!strcmp(token, "(net") )
{
    last_net_line = lineno;
    max_net_pos = lineno;
    rename_net=0;
#endif DEBUG2
    printf("Net Detected!\n");
#endif DEBUG2
    temp = strtok(NULL, " ");
    /* if it has a name then temp will not be NULL
     * otherwise there will be a renaming on the next line */
    if (temp!="NULL")
    {
#if defined DEBUG2
        printf(" Net Name= ",temp);
#endif DEBUG2
        strcpy(netname,temp);
        if (!strcmp(temp, "syn",3))
        {
            /* hold the name of the last instance found*/
            sscanf(temp, "syn\d", &temp_label);
            if (temp_label>max_net_label)
                max_net_label=temp_label;
        }
        if (!strcmp(temp, "clk_BUFGPe") ||
            !strcmp(temp, "reset") ||
            !strcmp(temp, "clk") )
        {
            continue; /* ignore reset and clk nets */
        }
    } else
        rename_net=1; /* rename this net */
}
}
else
{
    rename_net=1; /* net has a rename line */
    fgets(linestr, MAX_STR_LEN, fp); /* bypass the rename statement */
    lineno++;
    get_token(linestr,&token);
    if (strncmp(token, "rename")
    {
        printf("Error Detected!!! Found $s instead of \\
            string\n",token);
        c=getchar();
    }
    temp = strtok(NULL,"*");
    strcpy(netname,temp);
    fgets(linestr, MAX_STR_LEN, fp); /* This is the (joined line */
    lineno++;
    valid_net=1; /* assume it is valid at first */
    /* A net is not valid if it connects an input 
     to an IBUF or an O_BUF_F_12 to an output */
    tablesz=0;
    prev_char='.';
    for (i=0; i<MAX_PORTSTABLE_SZ; i++)
        ports_ptr[i].status='e'; /*set all entries to 'empty' first*/
    do
    {
        fgets(linestr, MAX_STR_LEN, fp);
        lineno++;
        ifdef DEBUG3
        printf(" Net Line read is: $s\n",linestr);
        get_token(linestr,&token);
        ifdef DEBUG3
        printf(" Net Token read= $s\n",token);
        printf("token[0]= %c, prev= $c\n",token[0],prev_char);
        endif
        if ((token[0]==')') && (prev_char==')'))
            break;
        /* net ends with a sequence of ')' tokens */
        prev_char=token[0];
        if (!strcmp(token, "(portRef)"))
        {
            temp = strtok(NULL,"*");
            strcpy(ports_ptr[tablesiz].port, temp);
            strcpy(port_name, temp);
        }
        else if (!strcmp(token, "(instanceRef")
        {
            temp = strtok(NULL,"*");
            temp[strlen(temp)-1]=0;
            if (!((getType_fromName(design, temp) == IBUF) &&
                (!strcmp(port_name, "I"))) ||
                (getType_fromName(design, temp) == OBUF) &&
                (!strcmp(port_name, "O"))) ||
                (!strcmp(port_name, "CLR"))) ||
                (!strcmp(port_name, "PRE"))) ||
                (!strcmp(port_name, "C"))) ||
                (!strcmp(port_name, "P"))) ||
                (!strcmp(port_name, "U")))
            {
                getType_fromName(design, temp) == BUFGP)
                {
                    valid_net=0;
                    ifdef DEBUG3
                    printf("Net $s is not valid\n",netname);
                    endif
                    /* Net is not valid.
                     It connects either an input pad to an IBUF 
                     or an OBUF to an output pad, or is part of

```
the clear, preset, clock, ground or power nets.*/

} else
{
#endif DEBUG3
   fflush(stdout);
   printf("Port: %s \t Name: %s\n", port_name, temp);
   printf("Tablesize=%d\n", tablesize);
#endif
   strcpy(ports_ptr[tablesize].name, temp);
   /* set entry to 'full'*/
   ports_ptr[tablesize].status='F';
   tablesize++;
   }
} while(1);
/* we have just read a net and created the ports_table */
if (valid_net)
{
#ifndef DEBUG3
   printf("\nValid Net Table of size %d\n",tablesize);
   for (i=0; i<tablesize; i++)
   {
      printf("%s %s\n",ports_ptr[i].port, ports_ptr[i].name);
   }
#endif
   join_net(design, ports_ptr, tablesize, last_net_line,
            netname, rename_net);
   count_nets++;
}
else
{
#ifndef DEBUG3
   printf("Net is not Valid!\n");
   printf(" Size: %d\n",tablesize);
   for (i=0; i<tablesize; i++)
   {
      printf("%s %s\n",ports_ptr[i].port, ports_ptr[i].name);
   }
#endif
   }    /*... else if(:strcmp(token,"(net")...)*/
}    /* end of while-loop */
#else DEBUG2
printf("%d valid nets in total\n",count_nets);
printf("\n\n");
c = getchar();
printlist(design);
printf("\n\n");
c = getchar();
printf("max inst label=%d\n",max_inst_label);
c=getchar();
printf("max net label=%d\n",max_net_label);
c=getchar();
printf("max net pos=%d\n",max_net_pos);
c=getchar();
#endif DEBUG2
showconnections(design);
#endif
identifyblocks(design);
fclose(fp);
computechanges(&commandlist, design, max_inst_label, max_net_label);
createnewdef(commandlist);
for (i=0; i<FPeCount; i++)
    deleteclasses(&FP_classes[i].class_list);
free((char*)FP_classes);
#endif DEBUG3
printlist(design);
#endif
free_list(&design);
free((char*)ports_ptr);
}
Appendix C

In this appendix we may see the five class files obtained by running the application, the output optimized files that are obtained after running the SIS optimization tool and finally the SIS script used to optimize both of the designs.

The five class files are:

Class0.eqn:

\[ C517=i0C517' \ i1C517' \ i2C517 \ i3C517; \]
\[ C520=i0C520' \ i1C520 \ i2C520'; \]
\[ C515=C517 + i3C515 + i0C515 \ C520'; \]
\[ C518=i3C518 + i0C518' \ i1C518 \ C520; \]

Class1.eqn:

\[ C522=i0C522 + i1C522 + i2C522'; \]
\[ C521=i0C521' \ i1C521 \ C522; \]

Class2.eqn:

\[ C527=i2C527 + i0C527' \ i1C527; \]
\[ C523=i2C523 + i0C523 \ C527; \]
\[ C525=i3C525 + i0C525' \ i1C525' \ C527; \]

Class3.eqn:

\[ C528=i2C528 \ i3C528 + i3C528 \ i0C528' \ i1C528; \]

Class4.eqn:

\[ C531=c_{in\_1\_a} \ c_{in\_1\_b'} \ c_{in\_1\_c} \ i3C531; \]
\[ C546=c_{in\_1\_a'} \ c_{in\_1\_b'} \ i3C546 + c_{in\_1\_a} \ i3C546 \ c_{in\_1\_c}; \]
\[ C530=C531 + c_{in\_1\_a} \ C546 + C546 \ c_{in\_1\_b}; \]
\[ C549=c_{in\_1\_a} \ c_{in\_1\_c}; \]
\[ C550=c_{in\_1\_b'} \ i1C550 \ i2C550'; \]
\[ C533=i0C533 \ C549 + C549' \ C550; \]
\[ C534=c_{in\_1\_a'} \ c_{in\_1\_b} \ c_{in\_1\_c} \ i3C534; \]
\[ C536=c_{in\_1\_a} \ c_{in\_1\_c'}; \]
\[ C532=C534 + C533 + C536 \ i2C532; \]
\[ C538=i2C538 + c_{in\_1\_b} \ i1C538; \]
C539 = c_in_l_a' c_in_l_b c_in_l_c i3C539 + c_in_l_a c_in_l_b c_in_l_c' i3C539;
C540 = c_in_l_a' c_in_l_b' i2C540 + c_in_l_a' i2C540 c_in_l_c' + c_in_l_b' i2C540 c_in_l_c';
C541 = c_in_l_a' c_in_l_b' i2C541;
C537 = C541 + C540 + C539 + C538;
C544 = c_in_l_a' c_in_l_b' i3C544 + c_in_l_a' i3C544 c_in_l_c' + c_in_l_b' i3C544 c_in_l_c';
C543 = C544 + c_in_l_a' c_in_l_b' C546;
C548 = c_in_l_a c_in_l_b i2C548 i3C548' c_in_l_c';
C542 = C548 + C543 + C550 C549;

The corresponding output files are:

Out0.eqn:

INORDER = i0C517 i1C517 i2C517 i3C517 i0C520 i1C520 i2C520 i3C515 i0C515 i3C518 i0C518 i1C518;
OUTORDER = C515 C518;
C520 = !i0C520*i1C520*!i2C520;
C515 = !C520*i0C515 + [13] + i3C515;
C518 = C520*!i0C518*i1C518 + i3C518;
[13] = !i0C517*!i1C517*i2C517*i3C517;

Out1.eqn:

INORDER = i0C522 i1C522 i2C522 i0C521 i1C521;
OUTORDER = C521;
C521 = !i2C522*!i0C521*i1C521 + [8];
[8] = i1C522*!i0C521*i1C521 + i0C522*!i0C521*i1C521;

Out2.eqn:

INORDER = i2C527 i0C527 i1C527 i2C527 i0C523 i1C523 i2C523 i3C525 i0C525 i1C525;
OUTORDER = C523 C525;
C527 = !i0C527*i1C527 + i2C527;
C523 = C527*i0C523 + i2C523;
C525 = C527*!i0C525*!i1C525 + i3C525;
Out3.eqn:

\[
\text{INORDER} = i2C528 \ i3C528 \ i0C528 \ i1C528;\\
\text{OUTORDER} = C528;\\
C528 = i3C528*i0C528*i1C528 + i2C528*i3C528;
\]

Out4.eqn:

\[
\text{INORDER} = c\_in\_1\_a \ c\_in\_1\_b \ c\_in\_1\_c \ i3C531 \ i3C546 \ i1C550 \\
i2C550 \ i0C533 \ i3C534 \\
i2C532 \ i2C538 \ i1C538 \ i3C539 \ i2C540 \ i2C541 \ i3C544 \ i2C548 \\
i3C548;\\
\text{OUTORDER} = C530 \ C532 \ C537 \ C542;\\
C546 = c\_in\_1\_a*c\_in\_1\_c*i3C546 + \\
!c\_in\_1\_a!*c\_in\_1\_b*i3C546;\\
C530 = !c\_in\_1\_b*i3C531*C549 + C546*C549;\\
C549 = c\_in\_1\_a*c\_in\_1\_c;\\
C550 = !c\_in\_1\_b*i1C550*i2C550;\\
C532 = C549*i0C533 + !C549*C550 + [61];\\
C537 = [144] + i2C538;\\
C542 = C549*C550 + [209] + [207];\\
[60] = !c\_in\_1\_a*c\_in\_1\_b*c\_in\_1\_c*i3C534;\\
[61] = c\_in\_1\_a!*c\_in\_1\_c*i2C532 + [60];\\
[141] = c\_in\_1\_a!*c\_in\_1\_c*i3C539 + \\
!c\_in\_1\_a*c\_in\_1\_c*i3C539;\\
[142] = !c\_in\_1\_a*i2C541 + [141] + i1C538;\\
[143] = !c\_in\_1\_b!*c\_in\_1\_c + !c\_in\_1\_a!*c\_in\_1\_c + \\
!c\_in\_1\_a*c\_in\_1\_b;\\
[144] = i2C540*[143] + c\_in\_1\_b*[142];\\
[207] = !c\_in\_1\_b!*C549*i3C544 + C546*C549;\\
[208] = c\_in\_1\_a*c\_in\_1\_b*i2C548*i3C548;\\
[209] = !c\_in\_1\_a!*c\_in\_1\_c*i3C544 + [208];
\]
This is the SIS script that we used:

sweep
eliminate -l
full_simplify -m nocomp
xl_partition -n 4 -M 4 -t -m
sweep
full_simplify -m nocomp
xl_imp -n 4 -g 2 -b
xl_merge -n 4 -F -u 4 -l -o outfile
xl_cover -n 4 -h 0
C
CAD, 3, 5, 6, 13, 16
Cell-based designs, 5
circuit models, 3, 13
CLB, 8, 34
cube, 20, 21
custom design style, 5

D
data flow model, 15
Decomposition, 24, 25

E
EDIF, 16, 17, 27, 28, 34, 36, 39, 41, 46
Elimination, 24
essential prime implicant, 22
expand, 22
Extraction, 24, 25, 30

F
FPGA, 6, 23

G
General Routing Matrix, 8

I
implicant, 21, 22
IOB, 8
irredundant, 22

L
library binding, 4
logic network model, 14, 17, 23, 27, 29
LUT, 9, 30, 31, 33, 34, 41

M
macro-cell designs, 6
minterm, 21
MPGA, 6
multilevel logic, 6, 23, 24, 32, 44
multilevel representation, 23

P
prime implicant, 21, 22

Q
Quine, 21

R
reduce, 2, 4, 6, 22, 33, 43

S
semi-custom design style, 5
Simplification, 24, 26
SIS, 18, 27, 32, 40, 43, 46
standard-cell designs, 6
state diagram model, 15
structural model, 14
Substitution, 24, 26

T
technology mapping, 4
two-level representation, 20

V
VHDL, 13, 17, 39, 41, 46
Virtex, 6, 8, 10, 17, 45
VLSI, 2, 3, 5, 45