

THE UNIVERSITY OF CALGARY

A Clausal Approach to Digital Logic Circuit Design

by

Ronan O'Byrne

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

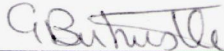
CALGARY, ALBERTA
JUNE, 1987

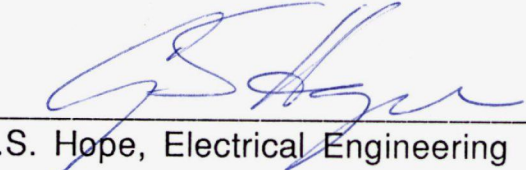
© Ronan O'Byrne, 1987

The University of Calgary
Faculty of Graduate Studies

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "A Clausal Approach to Circuit Design" submitted by Ronan O'Byrne in partial fulfillment of the requirements for the degree of Master of Science.


John Kendall, Computer Science


G. Birtwistle, Computer Science


G.S. Hope, Electrical Engineering

1987-06-01

Abstract

This thesis explores the use of logic programming as a technique to aid in the design of complex logic systems. The design of digital systems requires many purely clerical tasks which must be performed accurately and within the confines of many interacting rules. The design result which is sought is usually the result of a search over a solution space rather than a unique, procedurally generated answer. In the development of newer CAD tools the emphasis is on automatic logic generation from functional specifications and an ability to deal effectively with design complexity. Logic programming is proposed as a useful technique necessary to develop these flexible yet automatic design tools. Some research findings are presented, and a simple logic synthesis and design system based on Prolog is illustrated.

Acknowledgement

I thank my advisor John Kendall for sparking my interest in logic circuit design and for supplying direction and advice, even while giving me free rein to explore on my own. He has been a pleasure to work with, generous with time and research support, and a most constructive critic and remaining flaws are, of course, entirely of my own devising.

I thank the faculty and students of the Department of Computer Science, and other members of the JADE project for their support and interest. I thank the Natural Sciences and Engineering Research Council of Canada (NSERC). Most of all I thank my wife Ann, for her patience, understanding and encouragement.

Table of Contents

1. Logical Design of Digital Systems	1
1.1 The Design Process	1
1.2 The Challenge to CAD	5
1.3 Logic Design & Computer Aided Engineering	8
1.3.1 DA4 - International Computers Ltd.	9
1.3.2 IEDS - Intergraph Corporation	12
1.4 VLSI Design	14
1.4.1 Design Styles for VLSI	16
1.4.2 TANCELL - Tangent Systems Corp.	18
1.4.3 Electric	19
2. Synthesis of Combinational Logic	20
2.1 What is Logic Synthesis	20
2.2 Why Logic Synthesis	22
2.3 Automatic Logic Synthesis & Optimization	24
2.3.1 ALERT	24
2.3.2 LSS (Logic Synthesis System)	25
2.3.3 DDL/SX (Digital Design Language/Synthesis eXpert)	29
2.3.4 MACDAS Circuit Design System	31
2.3.5 Socrates	32

2.3.6 Logic Design using Tokio & C-Prolog	35
2.3.7 DAA (Design Automation Assistant)	38
3. Logic Programming	39
3.1 Logic Programming	39
3.1.1 Prolog Logic Programming	40
3.2 Prolog as an Expert System Shell	43
3.3 The Case for a Clausal based Expert System Approach	45
4. Circuit Representation in Prolog	48
4.1 Circuit Representation in Prolog	48
4.2 Prolog Horn Clause Circuit Representation	48
4.3 Prolog Structure Circuit Representation	52
4.3.1 Pre-defined Circuits	54
4.3.2 Circuit Input Signals	55
4.3.3 Handling Errors	56
5. Logic Minimization & Conversion	57
5.1 Boolean Logic	57
5.2 Logic Minimization	59
5.2.1 Absolute Logic Minimization	60
5.2.2 Heuristic Logic Minimization Techniques	61
5.2.3 Prolog Logic Rewrite Rules	62
5.2.4 Meta-level inferencing	63

5.2.4.1 Logic Minimization Example 1	65
5.2.5 Rewrite rules and true minimization	65
5.3 Logic Circuit Conversion	66
5.3.1 NAND/NOR Conversion	67
6. Counting Circuits	70
6.1 Counting Circuits	70
6.2 Designing Counter Circuits	72
7. Prolog Counter Design	78
7.1 User Interface	78
7.2 Limitations of a Prolog Interface	80
7.3 Selecting a Counter Type	81
7.4 Circuit Synthesis	85
7.4.1 Example 1 - Synchronous Ring Counter	85
7.4.2 Example 2 - Synchronous Count-by 23 Counter	87
7.5 Logic Minimization	91
7.6 NAND Logic Adoptation	93
7.7 Functional Simulation	93
7.7.1 Functional Simulation Example	94
7.8 PCD Session	96
8. Summary and Conclusions	102
8.1 Summary	102

8.2 Conclusions	103
8.3 Future Research	104
Bibliography	106
Index	116
Reference Index	118
Appendix A - PCD Listing	119
Appendix B - Prolog Simulation Trace	144

List of Tables

2-1 Logic Synthesis Systems	24
4-1 Known Circuits	54
5-1 Rewrite Rules	64
5-2 Conversion Rules for NAND logic representation	69
6-1 Counter Code Sequences	72
6-1 Flow table	75
7-1 PCD Commands	82

List of Figures

1-1 The Design Process	4
1-2 DA4 System Diagram	11
2-1 Levels of Description in LSS	26
2-2 Socrates System Diagram	33
4-1 Example circuit	52
6-1 Counter Tree Structure	71
6-2 Synchronous Ring Counter	73
6-3 Instability in a simple network	75

CHAPTER 1

Logical Design of Digital Systems

This chapter presents an overview of the evolution and use of CAD for logic design. It is an overview of a field which involves the works of many manufacturers, universities, and research foundations, and as such, the example systems are selected as representative of a particular type. Since this thesis is concerned with improved design automation techniques, this chapter answers the important question, “Why is there a need for better design automation ?”

Increasing design complexity of integrated systems is forcing a modification in the traditional approach to design. To illustrate current approaches to the problem of logic design several logic design and production systems are introduced. These systems are discussed with respect to their flexibility to adapt to changing fabrication technology, and their ability to adapt to increasing complexity.

1.1 The Design Process

Historically, the design task was carried out completely manually. One of the first design automation systems was presented at the 1956 Western Joint Computer Conference by Cray & Kisch [Cray 56]. They described a system which provided automatic checking of logic equations for logical, clerical, and timing errors, logic simulation and net-listing abilities. One interesting observation which also might indicate the pioneering nature of this article is that it cited no references. Several systems flourished in the sixties, with many computer manufacturers developing their own systems ([Kaskey 61],[Rosenthal 61], [Button 60], and others). Design automation was spurred on by tighter design constraints and complexity. These systems were generally not shared

among manufacturers, because they reflected each manufacturer's own design philosophy, circuit technology and design computer.

Meanwhile in an attempt to reduce the number of engineering changes caused by errors in design, logic simulation was introduced. The reasoning applied was that if the logic design was completely simulated, then automatic wiring machines, which used these "proven" designs, could produce an accurate reliable product. As system complexity increased, engineers quickly found out that they could not contain the entire design that they wanted to simulate on a small computer. Simulation had run into trouble. Simulation could not give full coverage. This problem was partially solved by the introduction of hierarchical design and the use of standardized components. Many current design systems are based on these principles. Now simulation could occur above the detailed level, and consequently could be more rigorous.

Hierarchical design became the first technique to deal with design complexity. In hierarchical design, initial design occurs at the highest level of abstraction and proceeds to lower detailed levels of design. As each level of the design is completed, it becomes a specification for the lower levels of the design. With hierarchical design techniques, optimization is hard to achieve because designs have downwardly imposing *design* constraints and upwardly imposing *physical* constraints. Failure in one or more levels of the design results in a redesign of that level and possibly of other levels in the design.

The physical constraints of design have become more complex as technology has evolved. In a first generation computer, one could point to a relay that performed some primitive Boolean operation, and hold it in one's hand. Fabrication technology has

advanced to where it is now impossible, without the aid of a microscope, to find the spot in an integrated circuit where that function is now performed. While the manufacturing design elements of the components in a microcomputer are considerably reduced compared to a first generation computer, the increase in the number of logic design elements, and increased the logic design task. Clock speeds are now much faster and so timing simulation is more important. Physical layout on silicon requires technology specific logic design.

The *Design Process* is the sequence of tasks required to create a design subject to constraints. In the design of digital systems, the design process takes a set of conceptual ideas which describe a proposed digital system, and transforms it into a set of detailed design data, such as part numbers, logic schematics etc., which provide suitable information required for manufacture. The conceptual design process is illustrated in Figure 1-1.

The functional design is the initial process of deriving a potential and realizable solution to the input design requirements. This is sometimes referred to as architectural analysis and design, and includes such activities as hardware/software tradeoffs and speed/power tradeoffs. With a firm functional design, analysis is then performed to determine the best way to implement the design, subject to the design constraints (technology, size, power, and cost). A schematic can then be drawn to show the proposed interconnections of available parts.

This proposed implementation is analyzed for proper functioning by applying a test sequence that emulates a subset of the conditions to be expected in real use. Once a designer is confident that the design will meet functional requirements, the physical lay-

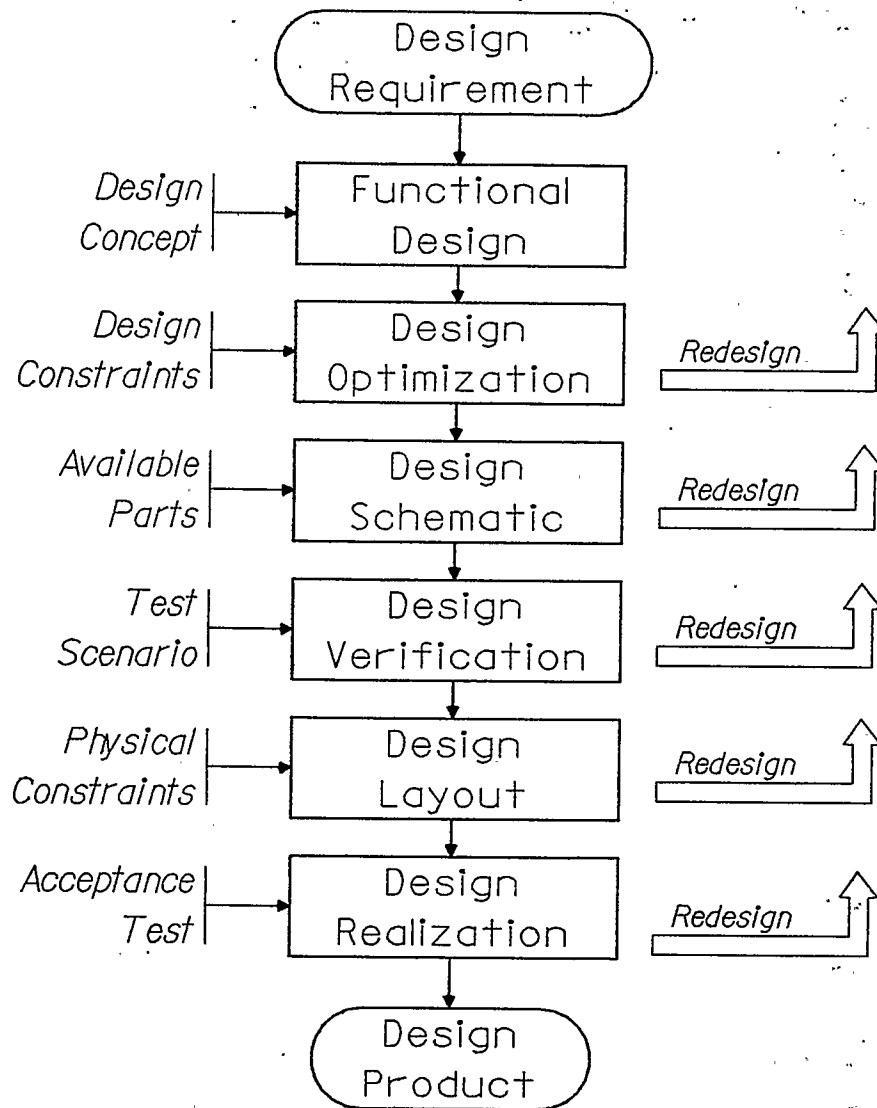


FIGURE 1-1 The Design Process

out of the actual interconnection of devices is formulated. Finally the design is realized, tested and ready for release as a product.

The design workflow is the sequence of tasks required to accomplish the design process. It is sequential only in an ideal conceptual case. It is conceivable that during

physical layout, flaws in the functional or schematic design may be uncovered, which would cause redesign at the conceptual level. During product acceptance tests, flaws in the functional design may be uncovered, making a major redesign necessary. The key to efficient design is to strive for a **sequential design workflow**, keeping any backtracking and recycling to one or two stages in the design flow. The purpose of Computer-Aided Design (CAD) is to assist the designer through each stage of the design workflow.

1.2 The Challenge to CAD

The expansion in the electronics field has occurred as a result of amazing progress over the past few years in the semiconductor fabrication technology. Gate densities and chip sizes both increased to make it possible to design larger, and more complex systems. This rapid change in the complexity of what could be made, changed the focus from "How to *make* it ? " to "How to *design* it ?" The challenge to CAD is to provide design automation which offers the opposing characteristics of flexibility and automation.

Provided with incredible potential for system integration, the electronic industry has been both innovative and reactionary in their response to the logic and system design problem. Innovative in their adoption of new fabrication technology, but reactive in advances in the use of CAD for design. The net result is the generally accepted belief that design is the major bottleneck to even greater system integration. This bottleneck occurs for a variety of reasons. Design systems are technology specific and have problems adapting to changes in the fabrication technology. But, the most important, and yet subtle problem which plagues design systems is their evolutionary nature. Most design automation systems are based on a level of Man Computer interaction which was

appropriate a few years ago.

Most designers readily admit that their designs do not occur in a rigorous scientific manner. They start a new design with only a sketchy knowledge of how the final design might turn out. This allows them to keep their options open until later in the design cycle. At the early stages, they are exploring possibilities of design. These specifications are solidified as design options are explored at various levels of the design hierarchy. When they look back at the process involved in creating a new design, they often believe that they stumbled upon the design rather than it being the result of a coherent design process. While it may be feasible for designs to occur in such a haphazard manner today, in future more complex designs will lay pressure for a more coherent approach. Inadequacies of current Computer-Aided Design (CAD), caused by a philosophy which considers CAD to *facilitate* design, makes design more of an art than a science, and places greater reliance upon the designer to direct the design process.

While most designers agree that CAD is the interaction of the designer with the computer to aid design decision making, they do not always agree as to what level of interaction is the most appropriate. Several implementation styles for CAD have evolved as a result, and are identified [McKinsey 84]. These design approaches are the result of the perceived role of CAD by corporate senior management. The roles can be classified as follows;

Electronic Pencil

An elementary role as an *Electronic Pencil* is suitable for very small designs, and for manual design workflow procedures. With this approach the CAD system provides only

the means for a designer to develop a new design. Yet, it is surprising how many design companies use CAD in this fashion.

Interactive Design

An *interactive* design system automates some common tedious step in the design process, but requires the designer to design the system. This is the approach adopted by design groups faced with a wide variety of design tasks, where a fully automated design system does not have the flexibility to address these design realms.

Automated Design Synthesis

An *automated design system* designs the system based on specifications entered by the designer. The CAD system is characterized by its complete involvement and active participation in the design process. Although this approach is new, it has been applied successfully in specialized design areas.([Darringer 80],[Fox 84],[Fujita 86] etc.)

1.3 Logic Design & Computer Aided Engineering

The commercial CAE (*Computer Aided Electronics*) marketplace is rapidly expanding in North America at an annual rate of approximately 35 % and, now represents approximately \$200 billion for both hardware and software. Despite these impressive figures the commercial use of design automation is in its infancy. Reviewing current CAD techniques, [Wayne 85] comments that it is the designers' reluctance to advance from low level design techniques which has restrained the advance of automation in commercial design environments. Design automation tools offer the logic designer greater productivity and reduced errors, but they require commitment to implement. It is this requirement to change methodologies, the comfort level in current low level design techniques, and the inflexibility of high level design techniques, which are responsible for the slow transition in the design community to higher level design techniques.

Vendors in the CAE/CAD marketplace have been classified [Bogert 87] according to the design systems offered. These classifications are;

(1) **High-end Electronic Generalist**

These companies attempt to provide an integrated set of design and management methodologies that address diverse types of design. These companies stress comprehensive management techniques, integrated hardware, and file and database computing. Although the design software of the electronic generalist is not as sophisticated as the semiconductor specialists', their ability to integrate and transfer information is superior.

(2) **Low-end Electronic Generalist**

These companies have built their electronic design automation products around the personal computer. The report's authors [Bogert 87] commented that these systems offer good design and management capabilities for their unusually low prices, but do not have the functionality of high-end systems. These systems are most suited to the "single-user" environment, but this is likely to change as low cost networking "engineer workstations" become available.

(3) **Integrated Circuit Specialist**

Included in this group are silicon compiler companies, and other sophisticated design techniques for VLSI, (*Very Large Scale Integration*). The "IC Specialist" is considered to offer "leading edge" IC design tools, but users are warned to make sure that implementations of these circuit designs are straightforward.

(4) **Semiconductor & Engineering Tools**

These specialists offer design tools as an adjunct to their semiconductor manufacturing businesses. These tools are usually finely tuned to their own semicustom products and processes, and have design centers where customers can use CAE tools to design circuits.

The following sub sections review example design systems from these defined categories.

1.3.1 DA4 - International Computers Ltd.

DA4 [Adshead 81] is the design automation system used by International Computers Limited in the United Kingdom. Following earlier experiments with design automa-

tion systems it was introduced in its basic form in 1974 as DA1. It is now a very comprehensive system, supporting designs in nearly 20 different classes of interconnection technology. Thus DA4 is typical of an *integrated* CAD system, which It represents over 400 man years of internal development. The goals of DA4 were;

- (1) To provide a common design system for the whole company.
- (2) To provide the basis for LSI (*Large Scale Integration*) design.
- (3) To support hierarchical logical and physical design.

A system diagram is shown in Figure 1-2. High level system design uses a language to represent a computer at the architectural level in terms of structure and behaviour. The design can be expanded in a “top down” fashion. Logic input is represented in the RMOD [Wager 81] language, which describes the circuit at the register level. Logic designs can be keyed directly into alphanumeric terminals or entered graphically at the design stations. RMOD achieves compression of input data by expressing a circuit at the “register level” , rather than at the conventional “circuit-element level.”

Multi-strings are used to represent parallel data-paths, and are given structured signal names. *Multi-symbols* represent logic functions that are available as a functional block.

DA4 provides logic simulation for complex logic elements reporting on worst case delays, timing race etc. DA4 can also be used for automatic circuit testing. DA4 facilitates many production outputs such as schematics and photographic artwork.

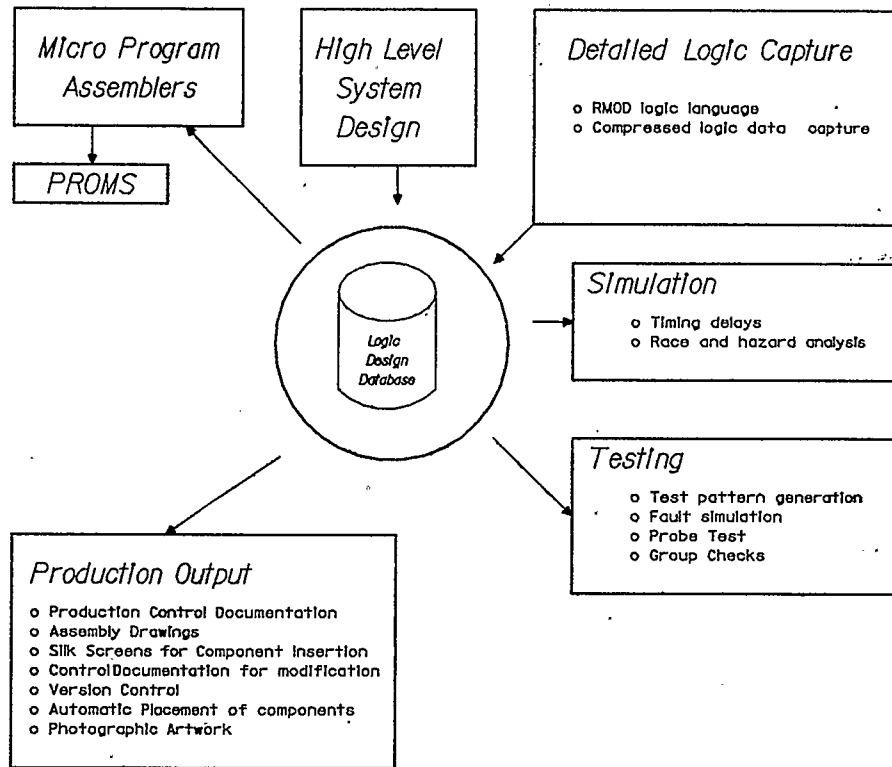


Figure 1-2 DA4 System Diagram

DA4 is typical of an in-house CAD system. These systems have evolved from a simple design automation tool to a complex workflow solution system which have been adapted to the evolving design automation needs of the company. With source code available within the company, future evolution of the design system can be controlled to conform to design practices and technologies. The result of this development guidance is a system which performs well in its intended environment, but fails to be adaptable in many other environments. This lack of flexibility means that DA4 is probably of no use

to any other company.

1.3.2 IEDS - Intergraph Corporation

IEDS (*Interactive Electronic Design System*) which runs on an Intergraph augmented DEC VAX and proprietary dual screen graphic workstations, is a typical graphical turnkey electronic design system for MSI and LSI circuit design. This system can be classified under "High end Electronic Generalist". The approach to the design is *interactive* where the designer can see instantly a graphical representation of the work he has performed. The design is stored in graphical and linked attribute databases. To speed the development of the schematic drawing, the designer can develop circuit cells and place these in the drawing with one operation. The schematic can be defined hierarchically, so that the design is partitioned into smaller functional blocks. Once the schematic of the circuit has been developed, the following automatic functions are available from IEDS.

- (1) **Automatic Net-list generation**

This is based upon schematic connectivity.

- (2) **Automatic component packaging for MSI**

IEDS packages gates into standard MSI logic ICs, and also assigns pin numbers.

- (3) **Best initial PC Component Placement**

When placing components on a printed circuit board, IEDS will show the outline of the component and "rubber banded" interconnects to other components already placed on the board.

(4) **Automatic Trace Routing**

IEDS is provided with a multi-layer circuit board router.

(5) **Manufacturing Interfaces**

Interfaces to manufacturing equipment such as drilling machines and photo masks are available.

Intergraph's IEDS electronic design software is typical of many turnkey CAD vendors such as Computervision, Calma or Applicon. Although most digital design groups use commercially available CAD systems, such as IEDS from Intergraph, they must put up with the lack of source code to make minor modifications and the lack of control in product evolution. These two drawbacks have not discouraged the use of commercial systems.

1.4 VLSI Design

VLSI (*Very Large Scale Integrated*) increases the size and complexity of logic design, and thus poses the following special design problems.

[a] **Verified Designs**

A traditional design approach which uses repeated low level iterations through tasks such as layout, detailed simulation, timing analysis, fault simulation, automatic test generation is prohibitively expensive for VLSI due to the complexity of the circuit. Circuits are more complex, where changes can have knock on effects which are difficult for the designer to comprehend and control. Design techniques must evolve to the stage where design is sequential to the greatest extent. This requires that testing and evaluation be performed as the design evolves and the use of design techniques which produce *verified* designs.

[b] **Design Input**

VLSI circuit design creates new problems for design input. Graphical input for logic design and layout is the traditional approach. It is based on using CAD as an “electronic pencil” allowing the designer to develop his design. For VLSI this becomes a cumbersome technique which limits the potential to improve the efficiency of the design. As the design is entered, it cannot be fully validated until completion and as a result, errors are not discovered until late in the design process.

[c] **Design Representation.**

Traditional design representation is in the form of a schematic which shows functional blocks, devices and the inter-relationships. These schematics describe only

the physical domain. There are two other domains of interest which, when combined with the physical domain, characterize any VLSI design. These are the structural and behavioural domains. These domains may be defined as follows. The *physical* domain is concerned with the specification of the physical layout of the integrated circuit via patterns on fabrication masks. The *structural* domain is concerned with describing the electrical characteristics of the design in terms of electrical components and their interconnections. The *behavioural* domain describes a design in terms of its function.

Any design language must interface directly with the design verification system, both at the structural and behavioural level. The necessity for having both structural and behavioural design verification is that, initially, it is likely that simple behavioural characteristics would be all that would be available. Hence design verification can begin at this level. As the design matures, structural design can be specified, and to some extent, automatically generated from the behavioural specification. The design verification process can then continue, until the behavioural and structural descriptions are verified to whatever degree of accuracy deemed necessary. This process can involve the elimination of numerous design errors and description errors. Furthermore, it can aid in the addition of testing hardware in the design to test those areas of logic that were found to be insufficiently tested by the design verification process.

[d] **Hierarchical Design.**

In a structured VLSI design environment there exists one hierarchy of description for all three domains. The VLSI design system should address functional and

physical problems at each level. SHIFT [Liblong 84] is an example of a hierarchic design language whose purpose is to capture the various descriptions of a circuit in a consistent manner.

1.4.1 Design Styles for VLSI

Integrated electronics has developed in a heatedly competitive and often secretive business environment. As a result there has been a proliferation of different device technologies, circuit design families, logic design techniques, mask making techniques, and wafer fabrication techniques, etc. Another obstacle is the high rate of change in the electronics industry, which is driven by improvements in fabrication technology.

Design constraints have caused the evolution of several *design styles*. These design styles have evolved to meet the requirements of particular design scenario.

The **fully custom** design method is an *ad hoc* implementation. To date, CAD techniques support custom design only to a limited extent, and as a consequence, custom design is profitable only for large production of complex systems, such as microprocessors or memories, or for circuits where special performance is required. Many industry analysts believe that fully custom IC design will be a growth market in the 1990's, and will be performed by designers whose current approach is MSI logic.

In **gate-array** design, a circuit is implemented in silicon by personalizing a master array of uncommitted gates using a set of interconnections. The design is constrained by the fixed structure of the master array, and is limited to routing the interconnections. CAD for gate-array design allows complex circuits to be implemented in a short time.

Gate-arrays are widely used, in particular for small volume production or for prototyping new designs.

The design of a VLSI circuit in a **standard-cell** (or poly-cell) design method requires partitioning the circuit into atomic units that are implemented by precommitted cells. Placement and routing of the cells is supported by computer-aided design tools. The standard-cell and gate-array design methods alone do not support highly optimized designs. Standard cell designs are more flexible than gate-array designs, but require longer development time. An approach which combines the speed of gate-arrays and the flexibility of standard-cells has been developed [Brown 74] called *CMOS Cell Arrays*. The CMOS Cell Array uses transistor isolation within pre-characterized standard cells to allow the cell row locations to be defined. This allows all wafers to be pre-processed with all necessary diffusions - final transistor size and placement are part of the customization and add to the flexibility. This technique is so similar to standard cell, that the same CAD software can be used. Even the CCA library has the function and performance as a SC library. Transistor isolation is the feature that allows pre-processing of the CCA wafer.

Designing using algorithmically generated macro-cells, bridges the gap between custom and standard cell design and is compatible with both methods. Macro-cells can implement functional units that are specified by design parameters and by their functionality. Macro-cells are usually highly regular and structured allowing computer programs, called module generators, to produce the layout of a macro-cell from its functional description.

The macro-cell approach is attractive because its flexibility allows the designer to exploit the advantages of both custom and standard cell methods. Highly optimized and area-efficient modules can be designed in a short time. In particular *Programmable Logic Arrays (PLA)* macros have been shown to be efficient for designing both combinatorial and sequential functions.

1.4.2 TANCELL - Tangent Systems Corp.

TANCELL is a Cell-based IC design system, developed by Tangent Systems Corp. TANCELL offers *timing driven* layout [Teig 86]. Timing-driven layout of semicustom ICs incorporates circuit timing requirements as basic criteria for layout optimization. The timing-driven layout process consists of circuit timing analysis, automatic layout using timing analysis results, and report generation documenting circuit performance. Properly applied, timing-driven layout can produce, in a single pass ICs that satisfy difficult timing specifications. Performed repeatedly during the layout process, timing analysis uses the latest layout information to calculate the propagation delay for every circuit path in the design. The timing analysis also measures the timing margin or criticality, which is used to drive the automatic layout tools. Each automatic layout tool uses this *criticality* in making placement and routing decisions. The frequent feedback from the analysis of how the layout is progressing, keeps the designer in control over the performance of the design.

This cell-based approach to IC design migrates timing analysis earlier in the design to reduce circuit design changes.

CHAPTER 2

Synthesis of Combinational Logic

This chapter presents Logic Synthesis and a number of systems which use that approach to design. It is a relatively new design technique which will extend the current role of CAD in logic design. The technique is characterised, its strengths and weaknesses highlighted, and various research systems are discussed. Later chapter 7 will show how logic programming can be used to implement a logic synthesis system.

2.1 What is Logic Synthesis

Logic Synthesis is a technique which generates a logic implementation in the desired technology from a designers functional specifications.

Logic synthesis programs are designed to improve engineering productivity by designing combinational circuits automatically. The effectiveness of such programs depend on their ease of use and the quality of the circuits they produce in the light of constraints applied to the design. Circuits are constrained by the types and characteristics of components available, and by area, delay and power, and testing requirements. Synthesis programs should therefore be capable of generating circuits with competitive area, speed, power and testability characteristics. Different constraints are not always compatible, ie. the smallest implementation is not always the fastest. So, an automatic synthesis program should also be able to make tradeoffs between competing constraint goals.

The logic synthesis problem is defined as follows:

- (1) Given a circuit family of components, including all constraints and circuit limitations associated with each circuit element type.
- (2) Given a logical description of a digital system in some language, such as the language of register transfers, Boolean equations or functions, or even gate equations.
- (3) Realize the system described in item (2) using components given in item (1) and in addition minimize the total implementation cost. This cost consists of both the circuit costs and the per-unit design costs. Usually these two costs are inversely proportional to each other.

The logic synthesis problem is analogous to the problem of machine-language translation. Logic synthesis implements a given digital system in terms of elements from a given circuit family. The compilation of a Fortran program, which consists of a set of high level language statements, results in a set of object level code which the computer understands.

The goal of generating an acceptable, technology-specific hardware implementation from a functional specification is not new. Three strategies have been developed.

One approach concentrated on translating Boolean functions into minimum two-level networks of Boolean primitives ([Breuer 72],[Dietmeyer 78]) and were later extended to limited circuit fan-in and alternative cost functions. Unfortunately, since these systems had algorithms which searched for true circuit minimums, they require time exponential in the number of circuits and cannot be used on most actual designs.

A second approach viewed the problem as one of assembling large macros. In these design systems, the data flow of the machine was generated in terms of predesigned or generated macros, such as multiplexors and ALUs. The control logic was usually implemented by PLAs, Weinberger arrays, or ROMs with microcode. Most of the current silicon compiler work falls into this category ([Johannsen 79],[Southard 83]).

Other research attempted to raise the level of specification. The **DDL** at Wisconsin [Duley 68], **APDL** at Carnegie-Mellon University [Darringer 69], and **ALERT** at IBM [Friedman 70], all began with behavioral specifications and produced technology-independent implementations at the level of Boolean equations. Designs produced were less efficient than a manually produced design, and they did not take advantage of the target technology. These systems pointed out the need for an appropriate level of efficiency and control for the designer over the circuit.

2.2 Why Logic Synthesis

Logic synthesis and optimization has recently gained significant credibility and practical use. Earlier systems only optimized cell counts, while current systems attempt to synthesis and optimize digital systems based on many technology criteria. With timing constraints, testability, wirability, and efficient use of available primitives in addition to cell counts, the system is able to produce quality designs which do not need further re-design. Such systems are favoured because of their ability to produce efficient technology specific logic.

Hand crafted logic designs are normally checked by modeling and simulation. But,

simulation alone is an inadequate check. The growing complexity of circuits, together with the increasing number of parameters, do not allow thorough simulation with a complete set of test patterns. Circuits which have been subjected to some incomplete functional testing are not guaranteed to be safe and reliable in operation. More thorough testing increases design costs, and introduces additional cost if flaws in the design are found.

A system which produces a *verified* design avoids costs associated with design iteration. The major difference to conventional design, is the requirement to completely determine behavior and interface description *before* starting any concrete design steps. Designers often decline to take this step because a very early detailed definition is not possible real circumstances. The alternative is for design to proceed unguided, where parts are added as necessary. The introduction of a formal task specification primarily shifts activities from a late stage in design to an early one, with subsequent time savings brought about through a reduction in design process cycling. A designer who adopts this approach has to change his interests at the early stage of design ; he must try to validate synthesis procedures used to produce a design rather than rely on simulation to later verify that his design will work.

This approach presents a number of challenges which are problems with current systems.

(1) **Modifiable & Extendible**

Synthesis algorithms should be easy to understand, and should be modifiable by the designer. This is a common problem with design automation tools and the most common reason why design automation is not so widespread. With pro-

cedural languages like Fortran, or C, this requirement is very difficult to satisfy.

(2) **Sub-optimal Results**

The design system should be able to develop suboptimal results where optimal results are not practical. This is important when designs are complex and sub-optimal results are of more value.

(3) **Portability**

The CAD system should be portable to different hardware.

2.3 Automatic Logic Synthesis & Optimization

A number of logic synthesis systems have been developed, and are listed in Table

2-1. Several of these systems are described in detail.

SYSTEM	DESCRIPTION	INSTITUTION
ALERT	Logic design generator	IBM Watson Research Center
DAA	Knowledge based synthesis	Carnegie-Mellon University
DDL/SX	Rule based logic synthesis	Fujitsu Laboratories Ltd
DFT	Synthesis with testability	Syracuse University
EL/SYN	Expert analog circuit synthesis	MIT
LSS	Logic synthesis system	IBM Watson Research Center
MACDAS	AND/OR circuit synthesis	Osaka University
MACPITTS	VLSI synthesis	MIT
Socrates	Synthesis & Optimization	University of Colorado
Tokio	Automatic CMOS gate array synthesis	University of Tokyo

Table 2-1 Logic Synthesis Systems

2.3.1 ALERT

The ALERT system [Friedman 69] converts preliminary high-level descriptions of computers into logic. ALERT is unique in its use of Iverson notation [Falkoff 64] to

describe the architecture of the computer. This input is processed by eight routines in series. First the “translator” checks the input and translates it into a less compact internal representation. Then the “selection decoding” routine scans for variable subscripts and if found replaces it with an appropriate block of logic. The “macro generator” replaces higher order logic elements with the complete combinational logic required to accomplish that operation, and the “sequence analyzer” determines the sequence and control requirements. The “consolidation” process eliminates duplicate logic blocks, and inefficiently connected arrays of elements.

2.3.2 LSS (Logic Synthesis System)

Introduction

LSS (*Logic Synthesis System*) ([Darringer 84],[Joyner 86]) is an experimental logic synthesis production system used to produce bipolar gate array chips. It has slowly developed from initial logic synthesis experiments at IBM’s Poughkeepsie laboratory in New York. The development team background of automatic theorem proving (D Brand,W Joyner), program verification (J Darringer) and logic design tools (J Gerbi) is reflected in the approach used in LSS. What follows is a brief description of LSS.

Design Approach

In LSS, logic is transformed from the high-level specification into production-quality implementation through a sequence of local transformations. The system takes (as one of several input forms) a description language at the register transfer level, and attempts to transform it into an interconnection of gates specific to a target technology.

LSS uses levels of description, to which local transformations are applied. These local transformations have the effect of simplifying the design and moving it towards the next level. Figure 2-1 illustrates this transformation process.

At the initial level, advantage is taken of “high level” constructs such as adders, decoders and parity generators. For example, decoders may be present in the logic

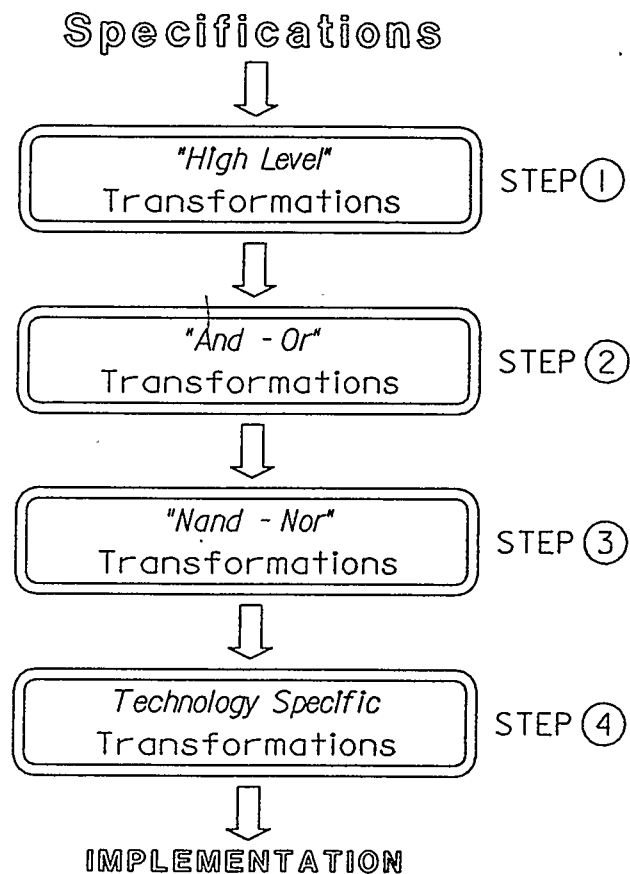


FIGURE 2-1 Levels of Description in LSS

because of their presence in the initial description, or because they are discovered by a transformation. Transformations are applied at this level before the information inherent in these operators is obscured by their expansion to more primitive gates.

At the AND/OR level of description, transformations simplify the logic by bunching together nested AND/ORs and by doing straightforward simplification. In addition transformations convert the design into an equivalent NAND or NOR representation.

At the final technology specific level of description, technology constraints are enforced and advantage is taken of technology opportunities. Complex primitives present in the target technology, such as exclusive-ORs and parity functions, AND-OR and OR-AND combinations and multiplexors are utilized. Timing requirements are enforced at this stage of synthesis.

LSS In Use

LSS is used in a production environment as an automatic tool even though it was originally conceived as an interactive design system. With standard scenarios of transformation occurring repeatedly, users become familiar with the resulting output logic and performed less manual examination. LSS is particularly suited to completing first pass designs rapidly. With LSS sweeping design changes can be considered to solve timing problems. This ability to correct timing problems through *high-level* changes rather than low-level path tuning has contributed to the success of the project.

LSS did not fair well in refining logic designs. To make a design better seems to require information which is not available to the synthesis system such as global planning and “don’t care” conditions. LSS would benefit from a knowledge engineering

approach. With this approach, the designer would have the flexibility to incorporate additional rules as required to improve LSS design refinement performance. LSS is implemented in PL/1, and suffers from the limitations associated with that programming style.

Technology Adaptation

LSS has been designed to produce efficient, technology-specific implementations. Rather than using technology information throughout the scenario of transformations, LSS uses table driven transformations to give LSS adaptability to different technologies while maintaining the ability to produce designs which take the most advantage of the technology. With this technique, new technologies can be incorporated quickly. Technology specific information is used in the technology independent parts of the synthesis scenario as well as in the technology-specific section to help decide whether a particular transformation should be applied. This information is typically used to calculate the potential savings in replacing a grouping of gates. For example, size information about a generic primitive (such as **OR**), which gives the number of cells it would take to implement in the target technology, can be used to evaluate the potential savings in replacing a group of **NAND** or **NOR** gates by such a primitive. In fact LSS will evaluate all the potential replacement candidates, and perform the conversions in the ranked savings order.

Timing

Timing is an important design criteria which must be considered for optimal circuit performance. The design goal of “speed” is usually to shorten certain critical paths on a

chip to meet design constraints, rather than to minimize all paths or total path length. LSS uses a technology specific delay calculator which computes the difference between the required and actual arrival times. This information is then used for timing correction transformations within LSS. These timing correction transformations attempt to meet the user-specified timing constraints, sometimes at the cost of area and power. The timing analysis procedure computes the worst case arrival time of a signal at any logic block input pin by tracing forward through the logic starting at the primary inputs, and it computes the worst case required time by tracing backwards through the logic ending at the primary outputs. The difference between the required and arrival times is the *slack*, which when negative indicates that a signal does not meet the required timing. The slack information is used by the timing correction transformations to determine their course of action.

2.3.3 DDL/SX (Digital Design Language/Synthesis eXpert)

DDL/SX [Saito 86] is a CMOS gate-array rule-based system for logic circuit synthesis. The system inputs technology-independent functional diagrams, and automatically generates conventional technology-dependent logic diagrams. A rule-based approach was adopted because the synthesis steps were not clear and were likely to change. This approach made it easy to incrementally improve the system's capabilities by adding, deleting, or modifying design knowledge represented as rules. Experimental results reported at Fujitsu in Japan, reveals that logic designs generated automatically are almost as good as the manual design, and that the design time is reduced by a factor of four.

The DDL/SX synthesis system's development was motivated by a desire to avoid errors introduced by manual gate-level design, and to automate a task which occurs regularly in electronic telephone system design.

The production system was implemented using *ESHELL* which is a general purpose tool for building expert systems. It provides the kernel of a production system based on a "backboard model" [Craig 86], and an environment which facilitates knowledge base construction. Knowledge rules are classified as follows;

(1) **Macro expansion**

These rules are knowledge about how to organize cells in order to implement a function of a macro.

(2) **Optimization Rules**

These rules are for removing redundant cells, and for replacing a group of cells with a single cell.

(3) **Constraint Check Rules**

These rules are for detecting and eliminating violations of design constraints such as fanout.

(4) **Miscellaneous Rules**

To interface the LSI with external circuits, I/O buffer cells and clock buffers must be inserted. Unused pins of the components should be connected to dummy cells which represent connections to ground or pull-up circuits. Scan path design rules are also included.

(5) **Scheduling Synthesis Tasks**

Rules for scheduling the synthesis tasks and checking whether the problem has been solved are in this category.

Experimental results shows that designs generated automatically have approximately 20 % more unit cells, but are created in approximately one quarter the time (allowing for input and slight modification) compared to manual design. Actual CPU times for a circuit with 2,000 basic cells is approximately 10 secs on a FACOM M-380 15 MIPS machine.

2.3.4 MACDAS Circuit Design System

MACDAS (Multi-level AND-OR Circuit Design Automation System) [Sasao 86] is a system developed at Osaka University which designs a multi-level circuit with fan-in limited AND-OR gates. To use MACDAS, the user presents the specification of the circuit in the form of a truth table, or a net-list of the circuit diagram, or an arithmetic expression. This input is processed as follows;

(1) **AND-OR Conversion**

The given specification is converted into an AND-OR two level circuit.

(2) **Two variable function generators**

Input variables are paired to produce an AND-OR two level circuit with two-variable function generators (TVFG). A TVFG generates all the functions of one and two-variables, and when inputs are paired each TVFG represents a “super variable”

(3) **Complement**

Some of the outputs are complemented to obtain a circuit with fewer AND gates.

(4) **Factorization**

The circuit is transformed into a multi-level fan-in limited AND-OR circuit. This algorithm is based on finding common factors to resolve the fan-in limitations. The algorithm, which maximally reduces the number of gates is drawn from [Dietmeyer 78].

(5) **Local transformation**

Finally the circuit is optimized by local transformations.

MACDAS uses two PLA optimization techniques. The first one is the optimal assignment of the input variables to PLA's with two bit decoders. The second is the optimal selection of the output phases. These optimization techniques produce designs which are better designed than would have been produced with a manual approach.

MACDAS is a useful tool for designing multi-level arithmetic circuits.

2.3.5 Socrates

The *Socrates* System [Gregory 86] diagram is presented in Figure 2-2. To have the best representational form at each level, Socrates uses three different design representations.

- (1) A logic Level representation is required for operations which operate on the logic of the circuit. This *logic level* representation uses an extended version of *Expresso's* PLA format.[Brayton 84]

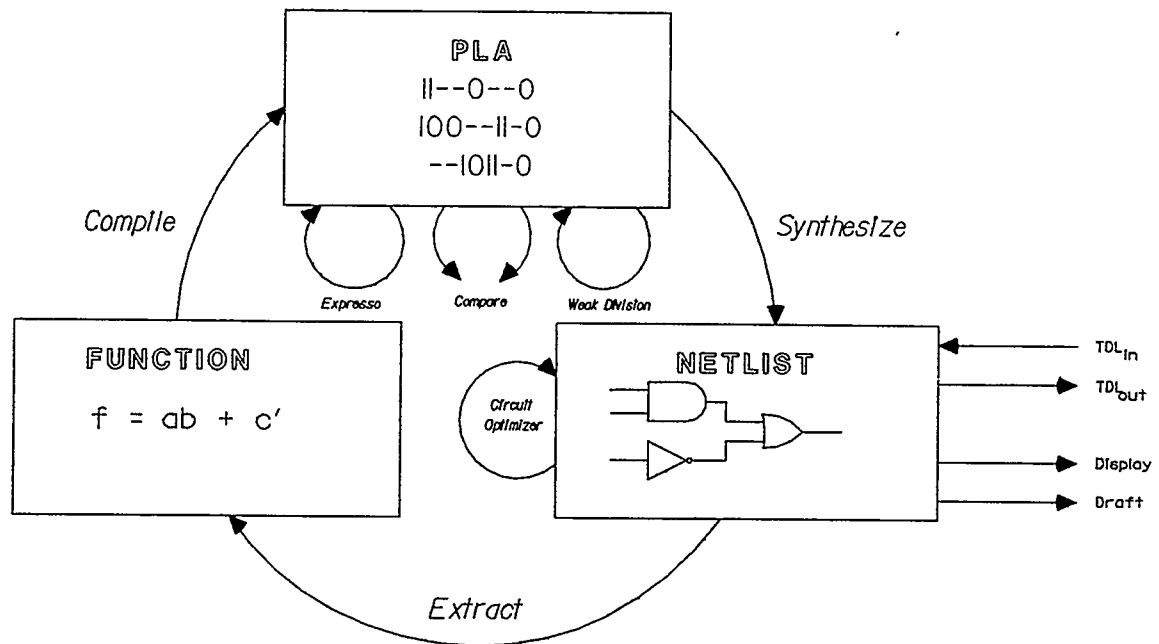


Figure 2-2 Socrates System Diagram

- (2) A circuit level representation is required for operations on circuits. They are represented using a *net-list format*.
- (3) A Boolean equation format for entering designs by hand.

Translators *Compile*, *Synthesize* and *Extract*, are provided to convert designs from one format to another. *Compile* converts Boolean equations to a two level *Expresso* format. Multilevel equations are flattened to two-level equations in this step. *Synthesize*

converts designs from the PLA format to the netlist format. Generic AND, OR and NOT gates are used to implement corresponding logic in this step. Extract converts a netlist to Boolean equations. *Extract uses a Boolean variable to represent each signal in the netlist, and writes an equation for each gate.*

A constraint specification allows designers to describe the desired characteristics of their circuit. Designers can specify when signals arrive at inputs, and the drive factor associated with them. Designers can specify the maximum propagation delays to individual outputs and the loads that must be driven at those outputs. Two programs *Expresso* and *Weak Division* perform logic level manipulations on designs. *Expresso* finds a minimum sum of products for each two level function. *Weak Division* decomposes two level functions into multiple levels by iteratively dividing out common subexpressions algebraically. The *circuit optimizer* program manipulates designs at the circuit level. The program improves circuit characteristics by iteratively replacing and rearranging groups of components in the circuit. It uses a library of alternate circuit implementations. The alternatives are given in the form of a rule, where conditions that are required to be true for the circuit are listed, if that alternative circuit is to be considered. Competing alternatives can thus be implemented in turn, and the performance of the circuit measured. After each rule application, an incremental timing and area analysis is performed. These analysis are based on user supplied values for timing and area models of each gate.

Before a rule is selected, transformations on the circuit resulting from its application are attempted. The program evaluates the effects a transformation will have on other transformations in the future by performing a state search. The depth and breadth of the

search tree determine how far, and how exhaustively the program looks into the future before selecting a new rule. This look-ahead mechanism enables the *circuit optimizer* to choose transformations which do not immediately improve a circuit, but which lead to other transformations which do. This look-ahead feature is an example of the application of *meta-level inferencing* in SOCRATES. Meta rules control how area and speed are traded off against each other, and when and where CPU time is used.

2.3.6 Logic Design using Tokio & C-Prolog

A program which automatically synthesizes logical circuits for CMOS gate array from state diagram has been developed at the University of Tokyo ([Fujita 86]). This system is written in Prolog and Tokio [Aoyagi 85], where Tokio is a logic programming language which is based on temporal logic. Tokio can be considered as a version of Prolog that has been extended to describe concurrent processing.

Synthesis Program Description

The synthesis process is divided into six steps:

(1) **Convert DDL description to Prolog.**

DDL [Duley 68] is a well accepted hardware description language, which is used as a neutral file format to input the design. The DDL description includes “Terminal Transfer Tables” , “Register Transfer Tables” and “State Transition Tables”

(2) **Expand & Fetch common parts from similar logic expressions.**

This is the first stage of simplification, which is illustrated by the following example:

$$T1 = A.B.C.D \quad T2 = A.B.C.E$$

is converted to:

$$COM = A.B.C$$

$$T1 = COM.D$$

$$T2 = COM.E$$

(3) **Primary Simplification: Eliminate duplicate units**

Four simplification rules are applied as appropriate.

1. Unification of units that have the same functions, same inputs, but different nets.
2. Simplification of AND or OR gates that have several identical inputs.
3. Simplification of multiplexer gates containing sets that have the same source under different conditions.
4. Simplification according to the replacement rules. Prolog is used to find circuit patterns where the replacement rules are applicable. Then the replacement rule is applied.

(4) **Analysis & modify the design according to the results.**

Perform an analysis the number of gates, the delay time, the fan-in and fan-out and modify the result if there are any design constraint violations.

Processing Results

Performance of this synthesis system is quoted for a Unify Processor. This processor consists of approximately 500 TTL IC circuits and 17 internal registers.

The DDL description consisted of 1000 lines.

The processing time on a VAX 11/730 is as follows;

	Hours:mins
Phase 1	5:30
Phase 2	
Expansion	:40
Cross-Reference	11:30
Phase 3	74:30
Phase 4	
Fan-out	3:30
Gate count	:10
Delay Time	2:00
Total	97:50

The initial design consisted of over 26,000 gates which was reduced, through simplification, by 10,000 gates or 40%.

Conclusions

While this system shows that logic synthesis is practical, the processing times shows that 20,000 gates is approximately the limit of capability for an interpretive system. The greatest processing time occurred for primary simplification. It occurs as a result of the repeated backtracking as the various simplification rules are tested. With more clever program design it should be possible to reduce this processing time.

2.3.7 DAA (Design Automation Assistant)

The Design Automation Assistant (DAA) [Kowalski 83] expert system was developed at Carnegie-Mellon University to investigate the application of Knowledge Based Expert Systems (KBES) for cost effective design of low-volume special purpose VLSI systems. DAA's area of expertise is allocating an architecture for a VLSI system. Its input is an algorithmic data-flow description of a VLSI system, and its output is a list of technology independent registers, operators, data paths and control signals. DA4 is implemented as a production system using the OPS5 [Forgy 81] Knowledge Based Expert System writing system. OPS5 facilitated the separation of expert knowledge from reasoning in DAA. Incremental addition of new rules and the refinement of old ones is easy because the rules have minimal interaction with one another.

DAA uses ordered subtasks to design the VLSI architecture. These subtasks are implemented using about 130 rules. Particular rules are applied if the specified conditions of the rule are met.

Experimental results from DAA indicate that a KBES approach to logic synthesis improves the performance of logic synthesis. Such an approach requires that the expert knowledge of design be explicitly defined. The definition of this knowledge aids our understanding of the design process and can also be used in the teaching process.

CHAPTER 3

Logic Programming

Logic programming is emerging as one possible technique for Computer-Aided Design (CAD) system development to cope with the recent increase in complexity of VLSI designs. Logic programming contrasts with current algorithmic solutions which are based on languages such as Fortran or C. This chapter discusses what logic programming is, and some specific advantages in its application to logic design.

3.1 Logic Programming

Logic Programming is a technique which combines logic clauses (or hypotheses) and a form of automatic logic deduction. Logic programming has gained acceptance as a suitable technique for implementing expert systems, and as a suitable programming technique for Japan's Fifth-Generation computer research and development project [Feigenbaum 83]. The theoretical basis of logic programming is Predicate Logic and the Resolution theorem [Chang 73].

The logic programming language **Prolog** ([Clocksin 81], [Cambell 84],[Pereira 84]) has gained wide acceptance throughout the AI community, and has been selected as one of the basic languages for Japan's fifth generation computer project. Prolog is a relatively new language, developed in the early seventies, which already has been used to develop specific expert systems for logic synthesis. Specific expert systems such as DFT (Design for Testability) [Hortmann 84] and DEMO, (meta Prolog experimentation sys-

tem), have been implemented in Prolog. Another system is under development at the University of Tokyo [Fujita 86]. Supporting such system development are numerous research papers which propose the use of the language for logic synthesis, simulation and testing ([Suzuki 85],[Gullichen 85], and [Noda 85]).

3.1.1 Prolog Logic Programming

A Prolog program is a set of “Horn” clauses, but the notation differs slightly from the traditional notation. In classical logic, a Horn clause may be written as,

$$P1 \& P2 \dots \& Pn \rightarrow Q$$

In Prolog syntax, the same clause would be expressed as,

$$Q \text{ :- } P1, P2, \dots, Pn.$$

where the antecedent is written to the right of the implication arrow, the consequent to the left of the arrow, the arrow itself is reversed and is written as “:-” and the “&” signs are replaced by commas, with a full stop at the end. “Q” is put at the left of the antecedent to put emphasis that the antecedent constitutes the body of a procedure for calculating “Q”. These clauses are both declarative, describing objects and their relationships and procedural, in that they are executed as functions. The symbol “:-” means “implied by” in the clause context. Each consequent and antecedent can be thought of as a function call of the form:

$$p(t1, t2, \dots, tn)$$

where “p” is an arbitrary predicate symbol, and “t1” through “tn” are terms. Clauses without antecedents are the facts of the system, while those without consequent clauses are used as goals.

A Prolog logic program usually consists of a set of rule and fact clauses which is used for the resolution of the goal clause. The goal clause is supplied from an external source. The resolution process involves matching the antecedents in the goal with consequents in the fact and rule set, and then using those antecedents as subgoals. This resolution process continues sometimes recursively until either the empty goal is reached (thereby proving the goal to be true) or a match is unavailable. When a match is unavailable backtracking occurs. Backtracking is the process by which the Prolog interpreter selects alternative choices for subgoals if they are defined. If backtracking exhausts all possible alternative definitions of the subgoal, then the subgoal is unprovable, and it fails for the given set of facts and clauses. During this resolution process the variable terms encountered are unified (“instantiated”) across antecedents and consequents. It is these *instantiations* of variables that are used as answers when the goal is proved. Logic programs prove or disprove goals only in relation to the set of clauses (facts, rules and goals) provided.

Prolog has been criticized as falling short of the ideal logic programming language in two areas [Naish 83]

(1) **Poor Implementation of Negation**

Horn clauses can only be used to deduce positive information. The best way of dealing with negation using Horn clauses is to use the *closed world assumption*,

that is anything which cannot be proved true is assumed to be false. This cannot be easily implemented so Prolog uses a weaker rule, *negation as failure*. A goal is assumed to be false if the interpreter finds a finite proof that the goal is unprovable. Most Prolog systems implement the clause $\text{not}(p(X))$ with a meaning of,

$$\text{not}(p(X)) \iff \text{for all } X, \sim p(X)$$

rather than,

$$\text{not}(p(X)) \iff \text{there exists } X \text{ such that } \sim p(X)$$

(2) Inadequate Control Facilities.

The basic control facilities of Prolog are just the ordering of clauses and atoms within clauses. Once a program has been written in a particular way, the clauses and sub-goals are always tried in the same order. While facilities such as *cut* and *var* partly overcome these problems, correctness and clarity suffers. Prolog's poor control facilities leads to poor program reliability, infinite loops and inefficient algorithms.

Two improved Prolog systems have been developed which overcome these problems of the basic system.

- (1) **MU Prolog** [Naish 83] comes closer to the goals of logic programming addressing the negation and control facility problems of Prolog. MU-Prolog uses a system of delaying and resuming calls to clauses to provide more flexibility,

efficiency and termination. Negation is implemented soundly by delaying the computation of the clause to be negated. The clause is *woken up* when the variables in the clause are bound.

- (2) **IC-Prolog** is probably the best known Prolog system with improved control [Clark 81]. In IC-Prolog, control information is specified by adding annotation to the program clauses. There are a wide range of annotations, and for certain applications IC-Prolog can achieve more efficient algorithms than MU-Prolog.

3.2 Prolog as an Expert System Shell

Rule-based languages are generally considered to be the most suitable for representing knowledge in expert systems. Rules are relatively easy to understand, and their modularity makes modification easy during knowledge base development and use.

Horn clause logic can be viewed as a rule-based language, which with appropriate extensions, it is a candidate for representing knowledge in an expert system. Any collection of Horn clauses can be run directly as a Prolog program. It has been commented [Hammond 83], that for some applications, running the expert system rules as a Prolog program is adequate, and the implementation of the expert system becomes trivial. However, two *important* expert system shell features are not automatically available in Prolog.

- (1) Prolog does not provide *automatically* an ability to explain and justify reasoning.
- (2) Prolog does not provide *automatically* a request for data based on inference.

However, such features can be added to a Prolog program.

Prolog has the advantages of uniformity and extensibility. Uniformity is provided in the form of rules that can perform both program control and data manipulation. To further refine the expert system, rules can be modified or added. One preliminary conclusion on the use of Prolog for DFT [Horstmann 84] [Horstmann 84] - CAD Using Logic Programming suggest that rules can be added or changed easily, even while using the system, and this feature was especially useful in developing and debugging the system. Careful system design which separates rule function gives a Prolog program “modularity” to adopt to design changes.

The efficiency of a Prolog program or the lack of it is a key concern for the acceptability of Prolog as the development language. If the task is numerically intensive, or if it can be procedurally defined then the task might be better suited to a procedural language which would execute more efficiently than Prolog. In later chapters it can be shown that there are many aspects of logic design which are best implemented in Prolog. Even if the task may be suitable to Prolog, if a subtask of the design process does not lend itself to Prolog programming, it can make sense to implement that task using a procedural language. A combination of Prolog and an efficient procedural language can provide considerably better overall performance when compared to a system programmed only in Prolog, without having to compromise any of the benefits offered by Prolog.

The performance and efficiency of Prolog depends on the system programmer understanding of the problem and how solutions can be obtained. It has been illustrated in many texts on logic programming [Bundy 83] that the existence of a solution does not guarantee that a solution will be found by the Prolog interpreter. It is thus necessary for

searching to be guided to obtain the solution in the fastest possible way. A Prolog system designer should always place “likely” rules first in a Prolog program, so that those rules are tested first. Backtracking, as the Prolog interpreter “tests” alternate rules is the major source of inefficiency in Prolog programs.

3.3 The Case for a Clausal based Expert System Approach

The advent of VLSI technology has put considerable strains on current design techniques in dealing with the growth in design complexity. Even hierarchical design techniques, which were introduced to deal with the complexity issue, are often inadequate to match downwardly imposed design criteria and upwardly imposed physical constraints. This thesis is in support of a clausal based expert system approach to design as the most effective long term strategy for inexpensive exploitation of VLSI technology. Such a design technique will make low-volume special purpose chips economically feasible.

Others support this design approach [Brewer 86], and propose a new model of design which is based on communicating expert systems which operate at different levels of design abstraction. The purpose of the expert on a given level is to create a structure out of the design components predefined for that level. With this approach design is not forced in a top to bottom fashion with little consideration for factors which arise at lower levels. Design still proceeds top to bottom as each level is completed, with the provision that any level may fail in its attempt to achieve its goal. When this occurs, control passes back to the parent in the form of a failure report. The higher level task may decide to re-allocate constraints, or change styles, or indeed fail itself. This procedure allows backtracking of earlier design decisions between levels of the design hierarchy, forcing

iterative refinement of design. It also effectively manages both upward and downward propagation of design styles and parameters. Constraint propagation and failure reporting augment the completed design specification, and can aid the “expert designer” to complete his design in much the same way as the human designer.

Although [Brewer 86] does not propose any specific language for implementing this model of design, the author suggests Prolog as suitable for the task. Prolog has many characteristics which would facilitate such an expert system model for design;

(1) **Backtracking**

Prolog’s backtracking feature is directly useful for implementing the failure reporting feature between levels of design abstraction.

(2) **Expert System Language**

Prolog has been used to implement Knowledge Based Expert Systems. There are KBES logic design systems ie DEMO, LSS.

(3) **Rule based Language**

Prolog is a rule based language with a built in simple inference mechanism.

(4) **Timing Representation**

Concurrent Prolog is available to represent the timing element of a design.

(5) **Unifying Language**

Prolog could be used for all aspects of system development, thereby unifying or linking these expert systems together. Prolog clauses are both declarative, in that they describe objects and relationships, and procedural in that they are executed as functions.

(6) **Circuit Transformation**

Prolog re-write rules facilitate circuit transformations. These transformations are required frequently in logic synthesis, logic minimization and for technology conversion.

CHAPTER 4

Circuit Representation in Prolog

This chapter presents two techniques for representing and manipulating circuits which are available in Prolog - data structures and Horn clauses. Circuit representational techniques are introduced first because they determine which manipulations can be performed easily. The PCD program (described in chapter 7) is based on a Prolog data structure circuit representation.

4.1 Circuit Representation in Prolog

In choosing a representational technique two questions are normally posed. Does the representational technique make efficient use of memory ? Does the representational technique allow for efficient manipulation ? Unfortunately it is very hard to find a technique which optimizes both these requirements simultaneously. A circuit which is represented using sets of Horn clausal statements presents circuit information suitable for logic manipulations, but this representation does not make the most efficient use of memory. A circuit represented as a Prolog data structure is stored more efficiently, but cannot be accessed in the same manner as in a Horn clause representation. These techniques are fully described in the following sections.

4.2 Prolog Horn Clause Circuit Representation

Digital logic circuits can be viewed as a network of primitive gates whose interconnection imposes constraints. Satisfying the constraints with some lines bound to some constant values serves to simulate the operation of the circuit. Many features of Prolog

make it suitable to direct representation and simple simulation of logic circuits.

The following are characteristics of a Prolog Horn clause circuit representation;

(1) **Functional & Physical Characteristics**

The Prolog database mechanism can record both functional and physical characteristics of logic elements.

(2) **Hierarchical Circuit Representation**

Prolog representation facilitates an abstraction of complexity using hierarchical descriptions. In digital circuitry, the subsystems tend to be homogeneous. Smaller components are replicated and interconnected to produce a larger piece of hardware. Hence, arbitrarily complex circuits, within implementation limits of the Prolog interpreter, may be constructed in a hierarchical manner.

(3) **Parallel Circuit Representation**

Parallelism of physical computer components are closely modeled using Concurrent Prolog [Suzuki 85]. Concurrent Prolog is very similar to Prolog, but it has multiprocessing features which make it suitable for describing and simulating highly concurrent systems.

(4) **Forward and Reverse Simulation**

As inputs and outputs of a Prolog predicate need not be specified, but may be left unbound at the time a predicate is invoked. These inputs are instantiated through the action of the Prolog interpreter, to make the predicate true. With these features, Prolog is very amenable to functional simulation of many circuit types.

Functional simulation is an alternative to transistor-level logic simulation.

It is often a better alternative because the circuit model, which can be modeled at any desired level of abstraction, can be generated quickly and the functional simulation, which is written at an abstract level is more efficient. A Prolog simulator provides an effective methodology to create a functional specification in a high level language and to debug these specifications against test data. Simulation can occur in the forward as well as the reverse direction, and even bidirectionally. The ability to efficiently perform backwards simulation is useful in both fault detection test generation and deductive methods for fault isolation.

(5) **Don't Care Values**

As Prolog can deal with unbound variables, the problem of “don't care” and “don't know” values is simplified.

Any logic gate, such as an *AND* gate, can be directly represented in Prolog as a collection of axioms which describe its behavior. For example, a 2-input *AND* gate is functionally specified by the following 4 Prolog axioms;

and(in(0,0),out(0)).

and(in(0,1),out(0)).

and(in(1,0),out(0)).

and(in(1,1),out(1)).

Queries may be posed to Prolog to simulate operation of the *AND* gate:

!?- and(in(0,1),out(X)).

X=0,

yes.

!?- and(in(X,Y),out(1)).

X=1,

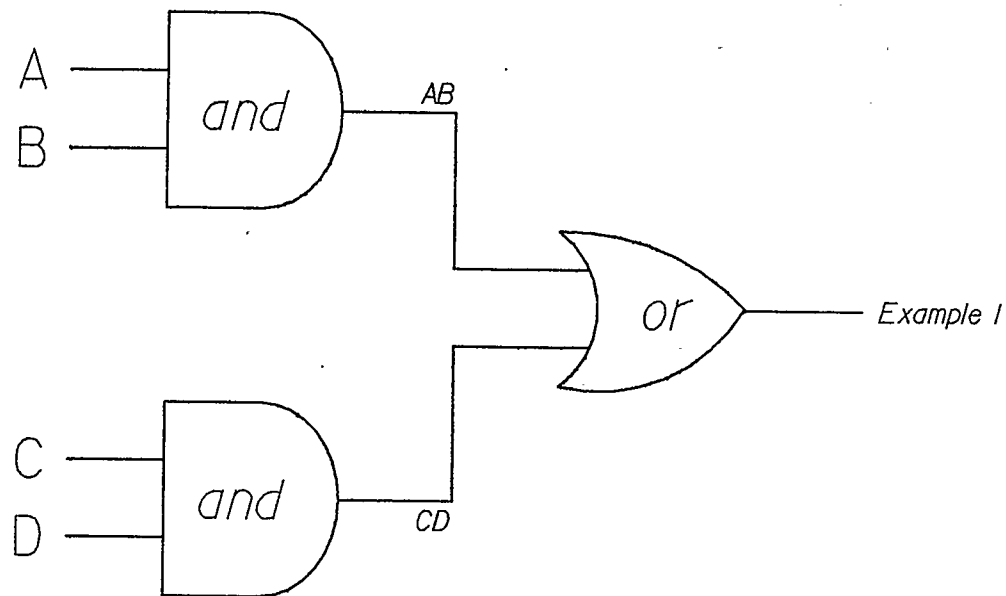
Y=1,

yes

In the first query, the operation of the *AND* gate has been simulated in the forward direction, with gate inputs being propagated forward to the output. The second query is an example of a backward simulation, and operates in a manner in which the hardware cannot. Gate output is propagated backward to the inputs. The query asks what inputs X and Y to the *AND* produce an output of 1. By matching the axioms in the database, Prolog indicates that both inputs must be 1.

So far we have considered only the representation of a two value primitive logic *AND* gate. This technique may be employed for representing gates which implement multivalued logic, and for representing more complex combinational circuits. Consider the representation of the logic function represented by Figure 4-1. Figure 4-1 represents a simple combinational logic circuit. Circuits are fashioned from an interconnected network of logic gates, and may be represented by Prolog implications in a manner amenable to simulation. This circuit could be represented as Prolog predicate circuit as;

example(InA,InB,InC,InD,Out) :- or(AB,CD,Out),and(A,B,AB),and(C,D,CD).



$$\text{Example 1} = (A.B) + (C.D)$$

Figure 4-1 Example circuit

where circuits $or(_,_,_)$ and $and(_,_,_)$ are pre-defined circuits.

4.3 Prolog Structure Circuit Representation

The use of a Prolog data structure for circuit representation is believed to be original to this thesis. This is the representational technique which is used as the basis of the PCD Prolog logic design system which is described in chapter 7. It was chosen for the

following reasons.

(1) **Circuit Manipulation**

A data structure circuit can be manipulated easier than a collection of predicate clauses. Circuits can be written and read from file in one operation. An arbitrary large circuit can be passed as parameters in a Prolog predicate clause. Several circuit definitions can co-exist and be individually manipulated.

(2) **Circuit Transformations**

Logic synthesis can be considered as a guided incremental transformation of a behavioral description into a logic description. The ability to perform transformations and combinations, which is an essential component of logic synthesis, makes data structures the best choice for a logic synthesis design system.

(3) **Compact Representation**

A Prolog data structure representation is more efficient circuit representation than a predicate representation. When several alternate large designs are being evaluated, this efficiency becomes important.

(4) **Flexibility**

A data structure representation can cope with multi-value logic, with any fan-in, and with any fan-out. The flexibility is such that circuits can be represented which are invalid.

To illustrate how a circuit can be represented as a Prolog data structure, let us consider the circuit in Figure 4-1. This circuit would be represented in Prolog as follows;

example(or(and(A,B),and(C,D))).

The circuit type is given by the functor, in this case “example.” The definition itself is not standalone. It uses the definition of common circuit elements which, for convenience, are considered “pre-defined” circuits. Also, in this case, circuit input signals are represented as variables. These can be circuits or signals in their own right.

4.3.1 Pre-defined Circuits

A distinction is made between known circuits such as *and(A,B)* or *or(A,B)* and user-defined circuits such as *a_circuit(and(Clock1,Sig1))*

Known circuits are primitive gates and circuit devices which the user can use to define his circuit. These known circuits form the lowest level representation possible, and their logic behaviour is pre-defined. Since these gates are frequently used, pre-definition of these logic elements reduces the representation of the user circuit. Known circuits are listed in Table 4-1.

CIRCUIT	DESCRIPTION
and(A,B)	Boolean logic operator "AND" between A and B
or(A,B)	Boolean logic operator "OR" between A and B
not(A)	Boolean logic operator "NOT" of A
jk_ff(J,K)	J K Flip-flop

Table 4-1 Known Circuits

A user can extend the range of pre-defined circuits to include additional circuit elements. In PCD, J K flip-flops are included in the list of pre-defined circuits because they occur frequently in the circuits created. To add an additional circuit as “pre-defined” additional rules, which describe its behaviour, need to be added to the database.

4.3.2 Circuit Input Signals

Inputs to a circuit can be either variable or fixed signals. Fixed signals have values such as “1” or “0” while variable signals do not. When the signals are variables, they represent circuits of arbitrary complexity. Ultimately these circuits can be evaluated to have values of “1” or “0” or “undefined” (or as appropriate in a multi-value logic system). For example 4-1 the circuit with variable inputs is represented as;

example(or(and(A,B),and(C,D)))

The use of uppercase for variable signals and lowercase for fixed is consistent with Prolog’s syntax. This circuit can be correctly interpreted as having signals represented by “A” , “B” , “C” , and “D” which can have a value of either “1” , “0” or “undefined.” It can be seen from this that a circuit cannot be evaluated until its inputs have been evaluated.

If the circuit has only fixed inputs the circuit itself is defined. In Prolog this can be represented using lower case variables and integers “1” and “0.” The example in Figure 4-1 with fixed inputs would be;

example(or(and(1,0),and(1,1)))

or with predefined signals;

example(or(and(a,b),and(c,d)))

4.3.3 Handling Errors

These rules which govern data structure representation allow the definition of the following valid circuits.

and(a,b)

not(a)

1

and(a,and(b,and(c,d)))

or(and(1,0),and(1,1))

The user can define circuits which are invalid. These circuits cannot be evaluated or minimized correctly. A few simple Horn clauses can be written to check the syntax of a circuit definition for correct signal usage and nesting of circuits.

CHAPTER 5

Logic Minimization & Conversion

Many attempts have been made to increase the size of logic minimization problem which can be addressed by sacrificing absolute minimality. This chapter introduces a technique which uses Prolog to implement a heuristic logic minimization tool. In addition the same transformation technique is applied to the problem of technology adaptation or logic conversion. The logic minimization and conversion techniques are presented as original work.

5.1 Boolean Logic

Boolean logic (more precisely *binary logic*) is the foundation for applications of logic circuits used in digital logic design. Boolean expressions are created by combining Boolean Operators such as *AND* , *OR* and *NOT*. These combinations are chosen to meet desired behaviour by the logic function under all possible variable inputs. Boolean expressions can be expressed in Truth Table form, in Canonical form, and in Circuit form. They can also be expressed as equivalent Boolean expressions using the rules of Boolean Logic. The rules of Boolean Logic are;

Basic Definition

- (1) $0 = 1'$
- (2) $1 = 0'$
- (3) $A + 0 = A$
- (4) $A \& 0 = 0$

$$(5) \quad A + 1 = A$$

$$(6) \quad A \& 1 = 1$$

Complements

$$(1) \quad A \& A' = 0$$

$$(2) \quad A + A' = 1$$

Commutative Laws

$$(1) \quad A + B = B + A$$

$$(2) \quad A \& B = B \& A$$

Identity Laws

$$(1) \quad A + A = A$$

$$(2) \quad A \& A = A$$

Distributive Laws

$$(1) \quad (A + B) + C = A + (B + C)$$

$$(2) \quad (A \& B) \& C = A \& (B \& C)$$

$$(3) \quad A + (B \& C) = (A + B) \& (A + C)$$

$$(4) \quad A \& (B + C) = (A \& B) + (A \& C)$$

DeMorgan's Law

$$(1) \quad (A + B)' = A' \& B' \quad (A \& B)' = A' + B'$$

5.2 Logic Minimization

Logic Minimization is the search for an equivalent circuit implementation of a Boolean expression which is “minimal” in both design and production costs. Minimal is generally taken to mean “minimal cost” but, the tradeoff between absolute minimal production cost and design time, and cost factors in the implementation technology have made minimization a more general circuit design problem. To illustrate this point, consider the minimization of a 3 variable Boolean expression with 4 terms expressed in its canonical form. It is practical to obtain the Boolean expression which contains the least number of Boolean operators by applying the rules of Boolean Algebra. This is possible because both the number of variables and the number of terms are small.

The minimization of;

$$F = A \& B \& C' + A \& B' \& C + A' \& B \& C + A \& B \& C$$

applying the Identity law,

$$= A \& B \& C' + A \& B' \& C + A' \& B \& C + (A \& B \& C + A \& B \& C + A \& B \& C)$$

applying the Distributive Law,

$$= (A \& B \& C' + A \& B \& C) + (A \& B' \& C + A \& B \& C) + (A' \& B \& C + A \& B \& C)$$

$$= (A \& B \& (C' + C)) + (A \& (B' + B) \& C) + ((A' + A) \& B \& C)$$

applying Complements,

$$= (A \& B \& (1)) + (A \& (1) \& C) + ((1) \& B \& C)$$

by basic definition,

$$= (A \& B) + (A \& C) + (B \& C)$$

$$= A \& B + A \& C + B \& C$$

In this minimization “proof” a directed search is occurring by selecting the correct rule at each stage. The rules are selected because they fit into the overall minimization strategy. These human proofs can be automated by computers using the Resolution Theorem and uniform proof procedures ([Bundy 83] Chapter 7). In effect, these techniques exhaustively apply all rules at each step. All equivalent Boolean expressions are generated, and the appropriate minimal expression is used, and the remaining expressions are discarded. In [Bundy 83] Chapter 7, this technique is criticized. At each rule application, a branching rate equal to the number of applicable rules in the database (often greater than 15) and recursive application of rules cause unreliable termination.

5.2.1 Absolute Logic Minimization

In the 1950s, when logic gates were expensive, it was very important to develop techniques that produced, for a given function, an implementation with the smallest number of devices. Such simplification of logic functions became an active area of research, and produced the map methods such as Karnaugh [Karnaugh 53] and Veitch maps, and later other more sophisticated tabular methods. The map methods were only practical for functions of up to 5 variables, while the tabular techniques were restricted by the computational intensity of the problem.

The tabular method ([McCluskey 56], [Quine 55]) consists of three basic stages;

(1) **Identification of prime implicants**

Although the generation of all prime implicants has become more efficient, it can be shown [Miller 65] that the number of prime implicants of a logic function with n inputs can be as large as:

$$3^n / n$$

(2) **Identification of essential prime implicants**

The problem of selecting a minimal cost set of prime implicants which covers the function “ f ”, is referred to as the *prime implicant covering problem*.

(3) **Prime Implicant Covering Table.**

Since the number of elements in the covering problem may be proportional to the exponential of the number of input variables of the logic function, processing makes this technique impractical for even medium sized problems (10 - 15 variables).

5.2.2 Heuristic Logic Minimization Techniques

Lower cost for logic gates in the early seventies reduced the requirement for an exact minimum. Large complex PLA implementations with over 30 inputs and 100 product terms made exact minimization impractical. Many heuristic techniques were developed to obtain a near minimum.

Some approaches start by generating all the prime implicants, and then instead of generating a minimum cover, a near minimum cover is selected heuristically ([Arevalo 78],[Hong 74] & [Rhyne 77]). With this approach there is the potential to generate a

very large number of prime implicants.

In two methods ([Rhyne77],[Arevalo 78]) a base minterm of the care-set of the logic function to be minimized is selected. It is expanded until it is prime, and all minterms that are covered by this prime are removed. The procedure is repeated until all the minterms of the care-set are removed. In [Rhyne 77], where all prime implicants containing the selected base minterm are generated, this method can be inefficient. In [Arevalo 78] only a subset of all prime-implicants covering the base minterm is generated. This gives a faster method with results which are not as good as in [Rhyne 77].

More recently heuristic minimization has found practical application in the design of PLAs. The first was MINI developed at IBM in the middle 70s [Hong 74]. Later a heuristic minimization program called PRESTO was introduced by D Brown [Brown 81]. During the summer of 1981 the authors created a program ESPRESSO - I [Brayton 82] to compare the various strategies employed by MINI and PRESTO.

5.2.3 Prolog Logic Rewrite Rules

Prologs rewrite rules express all valid manipulations to convert one form into another. The use of rewrite rules for manipulation is not new, and is based on ideas originally expounded by Bundy [Bundy 81].

The exhaustive application of Prolog rewrite rules has been criticized. Some problems will not terminate, while some result in an inefficient search. The repeated exhaustive application of rewrite rules is not guaranteed to result in a solution, and so a technique for controlling inference is required here.

5.2.4 Meta-level Inferencing

The term “meta-level” inference has been described [Bundy 81], where inference is conducted at two levels simultaneously: the “object-level” and the “meta-level”. The object-level is where knowledge about facts of the domain are encoded, while the meta-level encodes control or strategic knowledge. This style of inference results in a “guided” search for a solution.

While meta-level inferencing is not original to this thesis, the application of meta-level inferencing to logic minimization is. Reasoning at the meta level can range from the simple to the complex.

Let us start by considering a simple but effective meta level technique which is used in PCD. This technique could be called the *most effective rule first* technique and it relies on ordering of rules to guide the search for the true minimum. Rewrite rules are grouped into sets which address the main operators in the circuit. For typical circuits these are broken into rule sets for *AND* , *OR* etc. In each of these rule sets, the rewrite rules are ordered so that rules which have the most minimizing effect are placed first in the search. This simple technique provides rudimentary guidance to improve the efficiency of the search for a logically minimal representation. In PCD only those logic conversion rules which do not expand the logic expression are considered. Table 5-1 lists a complete set of minimization rewrite rules applicable for two input *AND* , *OR* logic. These rules are tested for a match in the order they are listed in the table. The Prolog notation and Boolean notation are listed together for comparison. The reader is referred to the Appendix listing of PCD for a complete list of Prolog rewrite minimization rules. A Boolean

Number	Rewrite Rule in Prolog	Boolean logic
1.	<code>min_str(and(1,1),1).</code>	$1.1 = 1$
2.	<code>min_str(and(1,X),Y) :- min_str(X,Y).</code>	$1.X = X$
3.	<code>min_str(and(X,1),Y) :- min_str(X,Y).</code>	$X.1 = X$
4.	<code>min_str(and(0,X),0).</code>	$0.X = 0$
5.	<code>min_str(and(X,0),0).</code>	$X.0 = 0$
6.	<code>min_str(and(X,Y),Z) :- min_str(X,1),min_str(Y,Z).</code>	$(X = 1).Y = Y$
7.	<code>min_str(and(X,Y),Z) :- min_str(Y,1),min_str(X,Z).</code>	$(Y = 1).X = X$
8.	<code>min_str(and(X,Y),0) :- min_str(X,0).</code>	$(X = 0).Y = 0$
9.	<code>min_str(and(X,Y),0) :- min_str(Y,0).</code>	$(Y = 0).X = 0$
10.	<code>min_str(and(X,Y),Z) :- min_str(X,A),min_str(Y,B),min1(and(A,B),Z).</code>	$X.Y = A.B$
11.	<code>min_str(or(1,X),1).</code>	$1+X = 1$
12.	<code>min_str(or(X,1),1).</code>	$X+1 = 1$
13.	<code>min_str(or(0,0),0).</code>	$0+0 = 0$
14.	<code>min_str(or(0,X),Y) :- min_str(X,Y).</code>	$0+X = X$
15.	<code>min_str(or(X,0),Y) :- min_str(X,Y).</code>	$X+0 = X$
16.	<code>min_str(or(X,Y),1) :- min_str(X,1).</code>	$(X = 1)+Y = 1$
17.	<code>min_str(or(X,Y),1) :- min_str(Y,1).</code>	$(Y = 1)+X = 1$
18.	<code>min_str(or(X,Y),Z) :- min_str(X,0),min_str(Y,Z).</code>	$(X = 0)+Y = Y$
19.	<code>min_str(or(X,Y),Z) :- min_str(Y,0),min_str(X,Z).</code>	$(Y = 0)+X = X$
20.	<code>min_str(or(X,Y),Z) :- min_str(X,A),min_str(Y,B),min1(and(A,B),Z).</code>	$X+Y = A+B$
21.	<code>min_str(not(1),0).</code>	$\sim 1 = 0$
22.	<code>min_str(not(0),1).</code>	$\sim 0 = 1$
23.	<code>min_str(not(not(X)),Y) :- min_str(X,Y).</code>	$\sim \sim X = X$
24.	<code>min_str(not(X),1) :- min_str(X,0).</code>	$\sim 0 = 1$
25.	<code>min_str(not(X),0) :- min_str(X,1).</code>	$\sim 1 = 0$
26.	<code>min_str(not(X),not(Y)) :- min_str(X,Y).</code>	$\sim X = \sim Y$
27.	<code>min_str(X,X).</code>	$A = A$

Table 5-1 Rewrite Rules

logic equation such as;

$$A.1 = A$$

is used to represent a common logic simplification. “A” can be anything from a simple logic variable to complex Boolean formula. Prolog can directly express these rules of Boolean logic simplification. Each Prolog rule has two parameters, for the input, and output structures. The input structure represents the circuit to be minimized, and the output structure is the equivalent minimized structure. As an example, if $(1 + b + c).\sim(1 + d)$

was being minimized the call to “min_str” would look like;

```
min_str(and(or(1,or('B','C')),not(or(1,'D'))),Out_str)
```

When minimization of this structure is complete, the variable “Out_str” will be instantiated to the minimized circuit data structure.

5.2.4.1 Logic Minimization Example 1

To illustrate how the rewrite rules work, let us consider the minimization of the following Boolean function

$$f2 = (1.a.c.d) + (1.a.e.f)$$

which can be represented as the following Prolog *data structure*

```
f2(or(and(1,and(a,and(c,d))),and(1,and(a,and(e,f)))))
```

This example has been specially chosen because its true minimization results in logic “false” and is independent of variables a, c, d, e and f.

Each “min_str” clause has two parameters, one for the input, the other for the output structure. There are also some corresponding ‘min1’ rules which are equivalent to “min_str”, but are not recursive. These clauses are used to prevent infinite loops.

5.2.5 Rewrite Rules and True Minimization

The approach taken to logic minimization in PCD can be summarized as;

(1) **Avoid term expansion**

Any given logic expression is not expanded. Since only those conversions which either reduce or convert to equal size are considered.

(2) **Apply maximum reduction first**

By ordering the rules in order of maximal reduction, the first minimal which satisfies this transformation procedure is likely to be the best.

This strategy avoids most of the computation required to obtain a true minimum. While it is possible for this strategy to provide a true minimal, it cannot be proven that a true minimum has been obtained.

A true minimum can be obtained using Prolog's rewrite technique, but the method is not practical for any real size problem due to the computational explosion, and the possibility that recursive expansion will prevent successful termination. The extra computation arises due to term expansion, and the requirement to search for *all* minimal solutions to determine which is the true minimal. The reader is referred to [Bundy 83] chapter 7 for a full explanation of the difficulties associated with this approach.

5.3 Logic Circuit Conversion

There are a number of different approaches to the problem of converting a logic network from one family of gate types to another ([Merwin 67], [Asija 68]). In digital logic circuit design, conversion is required in two areas;

(1) **NAND or NOR Logic**

Often one of the final stages in logic design calls for the conversion of an expres-

sion in *AND* , *OR* and *NOT* into an equivalent expression in *NAND* or *NOR*. It can be shown that any Boolean expression can be expressed in terms of *NAND* gates only, and also that these *NAND* gates can be expressed economically in MOS (Metal Oxide Semiconductor) Transistors.

(2) Canonical Form

A multi-level Boolean expression can be converted either into the sum of products or the products of sums. The *standard* or canonic form of these two forms is one in which each input variable appears in each of the minterms or maxterms. The standard form is useful because delay is limited to two gates, and if two different Boolean expressions have the same standard form, then they must be equivalent.

5.3.1 NAND/NOR Conversion

The problem is to convert a circuit description based on Boolean operators (*AND*, *OR* and *NOT*) into *NAND*(or *NOR*) based representation for the reasons previously outlined. Rewrite Rules can be applied here for the same reasons as with Logic Minimization. The rule set is clearer, because there are fewer rules to apply at any step. Let us consider logic conversion to *NAND* based technology. The conversion rules are;

Conversion Rules

- (1) $\text{and}(A,B) \rightarrow \text{not}(\text{nand}(A,B))$

- (2) $\text{or}(A,B) \rightarrow \text{nand}(\text{not}(A),\text{not}(B))$
- (3) $\text{not}(\text{not}(A)) \rightarrow A$
- (4) $\text{nand}(A) \rightarrow \text{nand}(A)$

For a circuit consisting of basic gate elements such as *AND* or *OR* gates etc., the rewrite rules in Table 5-2 on the following page describe the transformation to *NAND/NOT* technology, which obey these rules.

Usually, logic conversion to universal gate logic is considered in conjunction with logic minimization, but logic minimization can be required without logic conversion as is the case with a PLA implemented combinational logic function. The rules of conversion to universal gate logic have the tendency to expand the expression, and introduce more operators. Intuitively the sequence of logic minimization followed by universal logic gate conversion is not the most effective approach to the problem.

If required, these two tasks can be combined together. If a single rule set is developed then only one computationally intensive search needs to be performed, and the result will have less redundancy. Additional rules can be introduced to simplify the resulting expression.

```

conv_ug(and(A,and(B,C)),not(nand(D,nand(E,F)))) :- conv_ug(A,D),
    conv_ug(B,E),conv_and(C,F).
conv_ug(and(A,B),not(nand(C,D))) :- conv_ug(A,C),conv_ug(B,D).
conv_ug(or(and(A,B),and(C,D)),nand(nand(E,F),nand(G,H))) :-conv_ug(A,E),
    conv_ug(B,F),conv_ug(C,G),conv_ug(D,H).
conv_ug(or(A,and(B,C)),nand(not(D),nand(E,F))) :-conv_ug(A,D),conv_ug(B,E),
    conv_ug(C,E).
conv_ug(or(and(A,B),or(C,D)),nand(nand(E,F),nand(G,H))) :-conv_ug(A,E),
    conv_ug(B,F),conv_ug(C,G),conv_and(D,H).
conv_ug(or(not(A),or(not(B),not(C))),nand(D,nand(E,F))) :- conv_ug(A,D),
    conv_ug(B,E),conv_nand(C,F).
conv_ug(or(A,or(B,C)),nand(not(D),nand(not(E),not(F)))) :- conv_ug(A,D),
    conv_ug(B,E),conv_or(C,F).
conv_ug(or(not(A),not(B)),nand(C,D)) :- conv_ug(A,C),conv_ug(B,D).
conv_ug(or(A,B),nand(not(C),not(D))) :- conv_ug(A,C),conv_ug(B,D).
conv_ug(not(and(A,and(B,C))),nand(D,nand(E,F))) :- conv_ug(A,D),
    conv_ug(B,E),conv_and(C,F).
conv_ug(not(and(A,B)),nand(C,D)) :- conv_ug(A,C),conv_ug(B,C).
conv_ug(not(not(A)),B) :- conv_ug(A,B).
conv_ug(not(A),not(B)) :- conv_ug(A,B).
conv_ug(A,A) :- integer(A).
conv_ug(A,A) :- atom(A).
conv_ug(A,A) :- var(A),display('PCD error: structure not instantiated'),nl,abort.
conv_ug(S,Sug) :- S =.. [Class|List],cvt_ug_list(List,[],Uglist),
    Sug =.. [Class|Uglist].
cvt_ug_list([],List,Uglist) :- rev(List,[],Uglist).
cvt_ug_list([First|Rest],Tmp,Uglist) :- conv_ug(First,Ug),
    cvt_ug_list(Rest,[Ug|Tmp],Uglist).
conv_and(and(A,B),nand(C,D)) :- conv_ug(A,C),conv_and(B,D).
conv_and(A,B) :- conv_ug(A,B).
conv_or(or(A,B),not(C),not(D))) :- conv_ug(A,C),conv_or(B,D).
conv_or(A,B) :- conv_ug(A,B).
conv_nand(or(not(A),not(B)),C,D) :- conv_ug(A,C),conv_ug(B,D).
conv_nand(or(not(A),B),C,not(D)) :- conv_ug(A,C),conv_ug(B,D).
conv_nand(A,B) :- conv_or(A,B).

```

Table 5-2 Conversion Rules for NAND logic representation

CHAPTER 6

Counting Circuits

As a prelude to chapter 7, this chapter discusses the design constraints required for counters. This is provided first as background information and secondly to illustrate how constrained the design process is.

6.1 Counting Circuits

Counters are devices that count the number of times an event occurs. What we are concerned with here is recognizing the different types, how they differ, and how they are designed. Different types of counter circuits can be considered as branches of a *counter tree* as per Figure 6-1.

The first branch at the top of the counter tree is the most significant - Asynchronous or Synchronous. They differ not in the sequence counted, but in how they are clocked. The synchronous type is clocked directly, while the asynchronous has clocking only applied to the first flip-flop, and thus changes of state ripple from one flip-flop to the next.

The most general counters are the binary sequence *count-by n* or *count-to n*. The “count-by n” counts to n and resets, while the “count-to n” counts to n and must be reset before counting the sequence for a second time. Other common binary counters can be derived from these. For example a *BCD (Binary Coded Decimal)* counter counts from 0 to 9 and recycles, is a special case of a count-by n counter. Another example is a full modulo binary counter which counts from 0 to 15, and recycles. This counter also is a

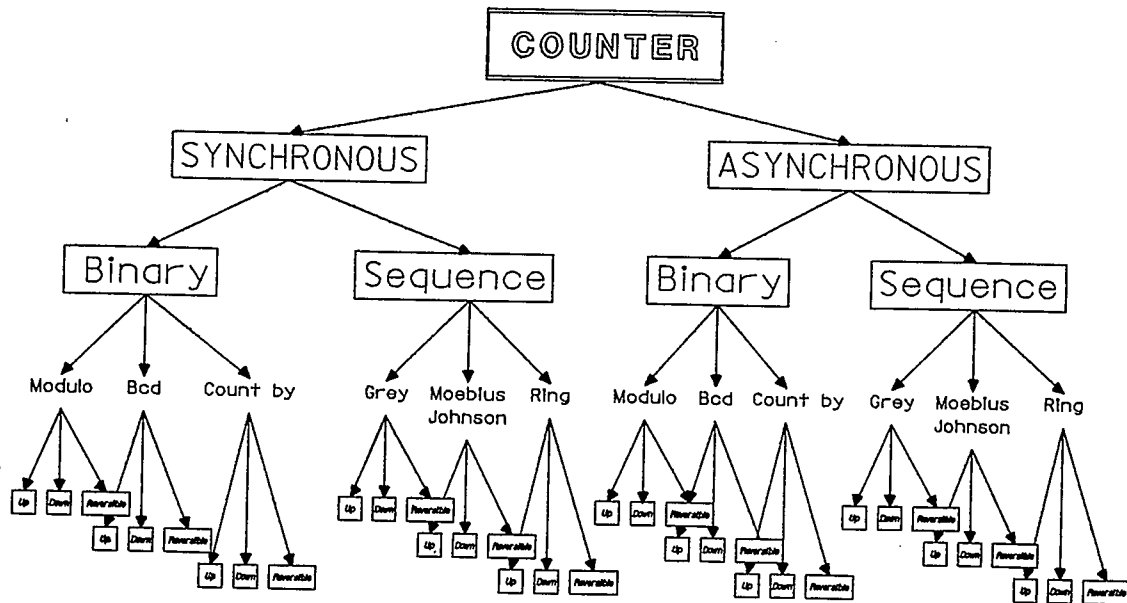


Figure 6-1 Counter Tree Structure

special case of a count-by n counter. All of these counter types can count in the forward direction (also known as *Up* counters) or reverse (also known as *Down* counters) or, reversible (ie both forward, and reverse depending in a reverse logic signal).

There are other special sequence counters worthy of note. A grey code is a sequence where only one bit changes with each count. Then there are the shift counters. These are the *Moebius* or *Johnson* sequence, and the *Ring* and *Switchtail* counters. This sequence is listed in Table 6-1.

GreyCode	Moebius	Ring
000	000	000
001	100	001
011	110	010
010	111	100
110	011	
111	001	
101		
100		

Table 6-1 Counter Code Sequences

6.2 Designing Counter Circuits

A counter circuit is a finite state machine. It has inputs, outputs and memory. Counter circuits should be designed considering the following design criteria.

(1) **REGULARITY**

A traditional minimization function, which strives for absolute absolute switching components reduction, is not suitable for VLSI design. For VLSI, circuit regularity is important to reduce silicon area required for wires. A large complex circuit which is regular can be a manageable design problem. This design criterion produces circuits with serial *pipeline* communications, and counter circuits which are implemented with PLA's.

(2) **SPEED**

Fast counter circuits are often required to meet critical system timing

requirements.

Asynchronous counter circuits are usually slower, and are more regular than a synchronous equivalent. This causes the designer to often have to trade-off circuit speed and area through reduced regularity.

Synchronous Ring Counter, and *Synchronous Switchtail Counter* are fast counter designs. To be fast, the counter must respond in the shortest possible time before the occurrence of the next clock edge. This interval will determine the effective maximum clock speed for the counter design. The Synchronous

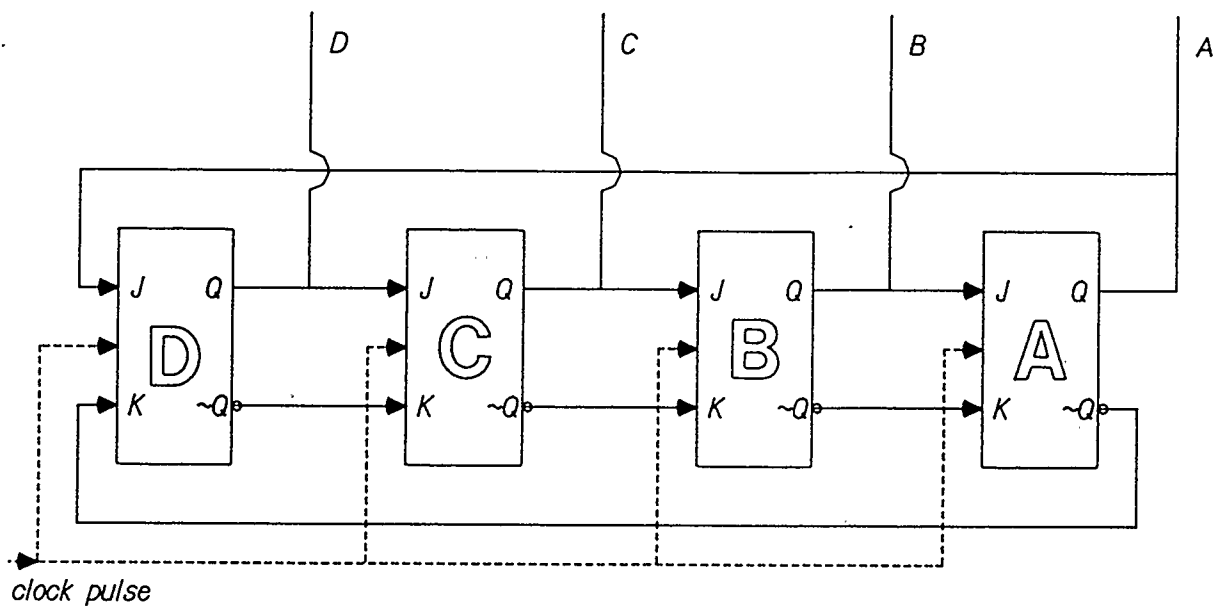


Figure 6-2 Synchronous Ring Counter

Ring counter is the fastest because it has no combinational logic to add to the delay caused by the flip-flop. Thus the maximum clock frequency is given by the following formula

$$f = \frac{1}{\text{propagation} + \text{setup time} + \text{strobe time}}$$

(3) **HARDWARE ECONOMY**

The desire for hardware economy leads to logic minimization. For VLSI design, silicon area minimization is a stronger requirement which leads to logic minimization. There can also be a trade-off between hardware economy and speed. A ripple counter can be implemented with flip-flop memory elements only, and is regular and economical, while a synchronous counter requires combinational logic gates to implement but can be clocked faster.

(4) **STABILITY**

Unstable counters are undesirable and result from poor design. They occur due to clocking too fast, or through unexpected reactions in the circuit design. If we consider the circuit in Figure 6-3 we can see how a simple circuit like this has instability. The stable condition is when $X = 0$, and $Y = 0$. When X changes to 1 then $\text{not}(Y)$ is 1, causing the output of the AND gate to go to 1. This will change the value of $\text{not}(Y)$ to 0, which will cause the oscillation to continue. In a simple case this problem can be avoided through intuitive reasoning, but in a more com-

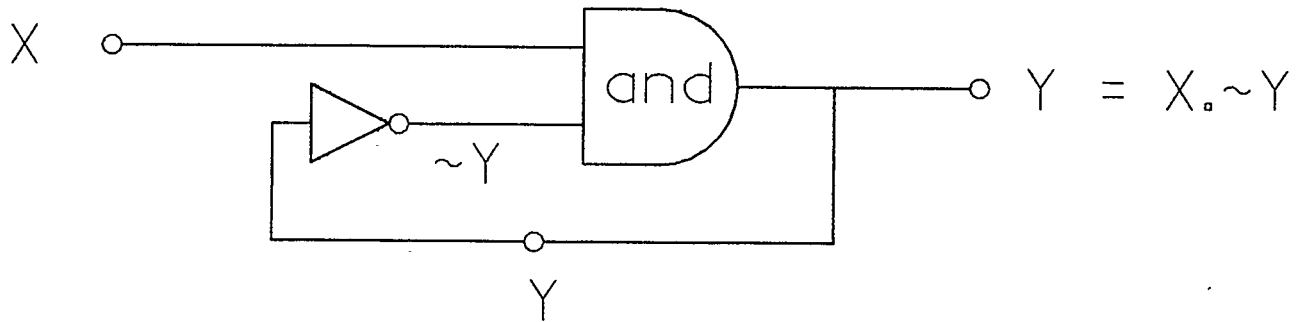


Figure 6-3 Instability in a simple network

plex circuit a Y-map or flow table must be used. For the example of $Y = X \cdot \text{not}(Y)$ the flow table is given in Figure 6-3.

Horizontal movement in the map corresponds to changes in the input variable X , and vertical for Y . The *energisation states* (entries outside the boxes) define the operational state that the circuit must assume. Stable operation is achieved when the energisation and operation states are identical. Only one such state occurs in the above table; when X and Y are zero. The operation states are deduced from

Y	X	0	1
	0	0	1
	1	1	0

\uparrow
0

\uparrow
 $\sim Y$

Table 6-1 Flow Table

the logical equations of the system. The equation is $Y = X \cdot \text{not}(Y)$. When $X = 0$ then $Y = 0 \cdot \text{not}(Y) = 0$. When $X = 1$ then $Y = \text{not}(Y)$, ie the operation state is always $\text{not}(Y)$. The Y values of the function are 0 for $X = 0$, and are plotted in the upper and lower left-hand cells. When $X = 1$ the operation state is always $\text{not}(Y)$, giving the values in the right-hand cells which are opposite the Y values outside the cell.

If initially $Y = '1'$ and $X = '0'$ the operation state is $'0'$. This becomes the new energisation state. Corresponding to this energisation state the operation state is 0, and the system moves to the cell $X, Y = 0, 0$. This is a stable operating state since the energisation and operation states are equal. The output will lock in this state. If X is now changed to $'1'$, the operation state becomes $'1'$ and the state of the system moves to the cell defined by the energisation state $X, Y = (1, 1)$. This results in a new operation state of $Y = 0$, with the output oscillating between $'0'$ and $'1'$ as shown by the arrows on the table. When X is changed to $'0'$, the network always returns to its stable operating state with $X, Y = (0, 0)$.

(5) RACE

When flip-flops are changing, all kinds of false outputs can be produced due to the variation in speeds of the devices in the circuit. A ripple counter gets its name from the effect of its flip-flops as they change state. With clocked logic, these momentary glitches have no effect but with unclocked logic these glitches may or may not have set the flip-flop. This can cause unreliable behavior. Most circuit designs should not use unclocked logic especially for PRESET and

CLEAR.

Race problems are often associated with asynchronous circuits, and waveform diagrams are useful to check for race problems. By exaggerating possible timing differences, possible race conditions can be examined.

A simple rule developed to help avoid race problems is

Never change more than one device in response to an asynchronous input signal

(6) **CLOCK SKEW**

Clock skew is caused by the delay in the propagation of clock signals throughout the circuit. For proper circuit operation, clock skew *MUST* be less than the minimum propagation delay minus the hold time.

CHAPTER 7

Prolog Counter Design

I have developed a Prolog based logic circuit design system PCD (Prolog Counter Design) , which is not intended as a production logic design system, but rather as a vehicle to illustrate how logic programming techniques, implemented in Prolog can be used for logic circuit design. PCD is also an experiment in the application of logic programming to logic synthesis, where the domain of interest is MSI implemented counter circuits. Circuits are represented as Prolog data structures, and individual designs are created using sets of Horn clauses which guide the creation of the circuit. PCD can minimize and convert these circuits to universal NAND gate logic, and can perform rudimentary simulation of the circuit to confirm compliance to design criteria. A listing of PCD appears in Appendix A at the end of this thesis.

7.1 User Interface

A simple user-interface for *PCD* was developed to provide an environment for the user to interact with *PCD*

(1) On-line Help

The user can request help for a listing of available commands, and a detailed explanation of what each does. Help information is stored in a separate file and is not read in until requested by the user. Each help information is represented as a “help rule” which matches if help is called concerning that item. This approach makes it easy to add *additional* help items.

(2) Command Interpretation

A command interpreter is provided to check command syntax and report program errors. For each valid command there is an occurrence of the clause “interpret(X)” , where “X” is the command. For a command that is entered, it’s syntax is checked by the clause **interpret(X)** to see if the command falls into any known structure. Each command option is tried. If no match occurs then the command always matches with the final rule definition, which echoes a message telling the user that that command is not understood. This approach allows the user to add additional commands to the program by adding in a new definition of *interpret(X)* while at the same time provides a *simple* command interpreter structure.

(3) **Design Storage**

A design can be created, modified and stored to a file, and recovered at a later stage. A design can be given a unique atomic name and subsequently manipulated by that name.

(4) **File Management**

As a measure to improve the speed of *PCD* not all clausal definitions are read from disk when *PCD* is started. Thus *PCD* is able to start faster because only core clausal definitions are read in. When particular functions are called up for the first time, the file containing their definitions are read in, and those functions are invoked in the normal manner. This process does not require the user’s initiation, and as far as he is concerned, *PCD* operates with only one file.

(5) **Error Messages**

PCD uses a standard error reporting technique which tells where in the program the error is coming from and why. In addition compound error message reporting is provided through failure backtracking of the goal. The combination of error messages provided, if programmed correctly, can provide valuable additional insight into the cause of the problem to the user.

7.2 Limitations of a Prolog Interface

A user interface which is written solely in Prolog suffers from some limitations of the environment provided by C Prolog programs. In the development of the user interface for *PCD* the following limitations were encountered.

(1) EOT marker

After every response by the user, a dot must be entered to signify EOT. So, a command at the *PCD* prompt would look like;

```
==> save(count23).
```

(2) Starting *PCD*

To initiate a *PCD* design session requires two operations. First the Prolog interpreter must be started, and second the Prolog program itself must be read into memory. Prolog's *saved states*, which allow a Prolog interpreter to load a Prolog saved state, does not fully solve this problem. First, "saved states" in Prolog are not guaranteed compatible between revision modifications to the Prolog interpreter. In addition there is no simple technique which can maintain an initial

“saved state”

(3) **Reading ‘?’ and ‘ ’**

Prolog’s built in function ‘read(X)’ is unable to read either a blank input or a question mark. Thus a novice user who enters a question mark or a carriage return, would not get any worthwhile response.

7.3 Selecting a Counter Type

The process of matching a circuit to requirements occurs at the early stages of functional design. This “matching” process can be considered as a “searching” process from available circuit types. This searching for the right circuit type can be implemented directly using Prolog’s depth first search techniques. Searching can be either *implicit* (depth first) or, *explicit* (breadth first) where the search is guided by circuit specifications entered by the user. In both search strategies, the search is also guided by the order in which the circuit options are placed in the database. At each node of the search tree, Prolog checks the first definition (usually represented as the left leg of the search tree) before checking alternative definitions (branches of the search tree). At each node of the search tree, the circuit specifications make only one leg of the search space valid. “Preferred circuits” come into effect when there are more than one branch which is valid using the circuit specifications. “Preferred circuits” is the bias which causes the best circuit to be returned when more than one circuit meets the circuit specifications entered by the user.

PCD requires the user to give a list of counter circuit specifications. Table 7-1 lists

and describes *PCD*

Specification	Description
check	Check the syntax of a circuit for errors
clear	Removes user counter circuit specifications
commands	Lists on terminal all valid commands
convert(design)	Convert circuit 'design' to universal gate logic
counter	Design a counter
designs	Lists all designs stored in memory and file
exit	Exit from the PCD environment
fan	Determine maximum fan in of circuit
get(design)	Retrieves 'design' from file
help(item)	Displays help information on 'item'
min(design)	Evaluate and display minimum for circuit 'design'
print(design)	Print to terminal the circuit 'design'
save(design)	Writes the circuit 'design' to file
shell	Create a Unix shell
simulate(design)	Simulate a design

Table 7-1 PCD Commands

To design a counter the user enters “counter.” at the command prompt as follows;

==> counter.

PCD then responds asking for a list of specifications. At this prompt the user can enter either a special command such as “help” , or “exit” or a valid circuit specification. If “help” is entered, then a list of valid specifications with brief descriptions is printed, or if “exit” is entered, then the design process is terminated. When the list of specifications has been entered, it is checked for syntax before any specifications are added to the database. This syntax checking compares the specifications entered by the user with a list of known specification formats. This verifies that all specifications are known to the system and points out early if there are any typographical errors. If all the

specifications are valid, then they are added to the database. Then they are checked against known conflicts rules. *PCD* contains a rule set of known conflicts caused by improper use of specification combinations. If any of these rules match then an error is printed, and the selection process stops. To illustrate how these rules are implemented in *PCD*, let us consider one known conflict in the use of the “binary” and “moebius” specifications. These specifications cannot be used together because they refer to different sequences that one counter could not count. To catch this if it should occur, the following rule exists. When the conflict condition occurs the specifications “binary” and “moebius” would be in the Prolog database, and so the error flag “invalid_spec” would be added to the database, and an error message indicating the problem is printed at the terminal.

```
check :- binary,moebius,
        asserta(invalid_spec),
        write('Error: binary and moebius spec conflict'),!,fail.
```

Note the use of cut and fail to force the use of other definitions of 'check'. A final definition of 'check' always succeeds. The spec *invalid_spec* is used to prevent any searching with that specification combination.

The user may not have supplied sufficient specifications to allow *PCD* to find a counter circuit type. In this case additional specifications are required to narrow down the choice. I have chosen to add rule sets which checks for specifications which do not allow *PCD* to find at least one path from root to leaf in the search tree. For example, if “modulus” is specified, but not “binary”, then the following clauses will catch this,

and in this instance make the necessary changes to fix the problem.

```
check :- binary,modulus,counter(T).

check :- modulus,asserta(binary),

        display('Warning: asserting spec binary').
```

PCD is ready now to perform the search. The following clauses illustrate how part of the search tree is represented in *PCD*.

```
select(T) :- sync(T).
select(T) :- async(T).
async(T) :- binary,async_binary(T).
async(T) :- ring,async_ring(T).
async(T) :- shift,async_shift(T).

sync(T) :- binary,sync_binary(T).
sync(T) :- ring,sync_ring(T).
sync(T) :- shift,sync_shift(T).
```

Here the search is controlled by the ordering of the clauses and instantiation of specifications. The depth first search here is directed to search the family of synchronous counters first. Only if no synchronous counter is matched will Prolog backtrack and try asynchronous counters. The first branch of synchronous counters that are checked are binary counters. This process of narrowing the definition continues until a unique type of counter circuit is defined. When a unique counter is defined then the Prolog variable “T” is instantiated to the name of that circuit type. If no unique counter is found, then the “T” variable is instantiated to “no_match”

7.4 Circuit Synthesis

PCD has clauses for logic synthesis. When the counter type is instantiated to a known counter type, *PCD* will build a circuit which conforms to the specifications. An important feature of the approach is that *PCD generates* a counter circuit meeting specifications rather than merely retrieving a previously stored circuit ! *PCD* performs the logic synthesis using groups of Prolog clausal statements, one for each circuit type. These clauses rely on recursive calling to create the circuit as a Prolog data structure. The following two examples illustrate how logic synthesis is performed.

7.4.1 Example 1 - Synchronous Ring Counter

Consider that a design for a *Synchronous Ring* counter is required. This circuit is to be implemented using two MSI J-K flip-flops. To design this counter we call up *PCD* as follows. Note that user input is shown in bold type.

% prolog

C-Prolog version 1.5

| ?- [**pcd**].

Prolog Circuit Design

(type "commands." for listing of available commands)

Version: April 14 1987

=>

“counter” to design a counter.

==> counter.

counter consulted 10764 bytes 3.1667 sec.

Please enter Counter Circuit Specifications

Enter “help” for available options >>

The user can access help information, or go straight to entering the specifications which will guide *PCD* in its search for a counter circuit type. An appropriate choice here is “ff(2)” to indicate that the circuit is implemented with 2 flip-flop or state variables, “ring” to indicate that a ring sequence is required, and finally “sync” to indicate that the counter is synchronous. The entry at the terminal would look like.

Enter “help” for available options >> **ff(2),ring,sync.**

PCD first checks the syntax of the specification entry. If valid then it prints out the message

Counter circuit Specification syntax check ok

Then *PCD* checks known specification conflict rules, and if no conflict is found, the circuit is searched for on the counter tree. In this case no match to any conflict rule is found and the search proceeds. With the given specifications, the search matches *synchronous ring counter* as the correct circuit type, and that circuit is created using recursive rules for that counter type. To synthesize this counter, *PCD* used just the following three clauses.

```
counter('synchronous ring',sync_ring(List)) :- ff(N),Next is (N-1),
```

```

ring(Next,(jk_ff('a',not('a'),clk)),List).
ring(0,L,L).
ring(N,Tmp,List) :- Next is (N-1),Last is (N+1),alpha(Last,Letter),
ring(Next,(jk_ff(Letter,not(Letter),clk),Tmp),List).

```

The following Prolog trace illustrates how these clauses and some additional utility clauses are used to build up the circuit;

```

(231) 11 Call: counter(synchronous ring,_33498) ?
(232) 12 Call: ff(_217) ?
(232) 12 Exit: ff(2)
(233) 12 Call: _33507 is 2-1 ?
(233) 12 Exit: 1 is 2-1
(234) 12 Call: ring(1,jk_ff(a,not a,clk),_216) ?
(235) 13 Call: _33533 is 1-1 ?
(235) 13 Exit: 0 is 1-1
(236) 13 Call: _33534 is 1+1 ?
(236) 13 Exit: 2 is 1+1
(237) 13 Call: alpha(2,_224) ? s
> (237) 13 Exit: alpha(2,b)
> (238) 13 Call: ring(0,(jk_ff(b,not b,clk),jk_ff(a,not a,clk)),_216) ?
(238) 13 Exit: ring(0,(jk_ff(b,not b,clk),jk_ff(a,not a,clk)),
(jk_ff(b,not b,clk),jk_ff(a,not a,clk)))
(234) 12 Exit: ring(1,jk_ff(a,not a,clk),(jk_ff(b,not b,clk),
jk_ff(a,not a,clk)))
(231) 11 Exit: counter(synchronous ring,sync_ring((jk_ff(b,not b,clk),
jk_ff(a,not a,clk)))

```

The call to the clause *counter* causes a call to *ff(N)* to determine the number of flip-flops required for the circuit. *ff(N)* returns “N” as equal to 2, which assigns variable “Next” as 1 in the call to *ring*. There are two definitions of *ring*, where the first is used as a terminating condition and is always checked first, and the second, which is recursively called for each flip-flop required in the ring counter. The actual circuit structure is built up as the clauses exit from the terminating clause.

7.4.2 Example 2 - Synchronous Count-by 23 Counter

As a more comprehensive example, consider the design of a synchronous counter which is to count a binary sequence to decimal equivalent “23” and then reset. It is possible to design such a *count-by n* counter using the same set of Horn clauses. For this example the user specifications entered would look like

Please enter Counter circuit specifications

Enter “help” for available options >> **count(23),clock(28).**

“count(23)” tells *PCD* that the counter must count 0..23 and “clock(28)” sets the clock cycles per micro-second. The concept here is that this clocking requirement will set the overall style of the counter ie with a fast clock speed forcing the use of a synchronous style.

To build a *Synchronous Count-by 23* counter, *PCD* uses the following clauses.

```

counter('synchronous binary count by',Str) :-
    count(D),binary(D,[Bit|Binary_list]),number_ff(D,N),
    process_true(N,B,Reset_term),process_false(N,B,Set_term),
    Next is (N - 1),form_and(N,Anded_term),
    do_eqtn_j(Set_term,Anded_term,Eqtnj),
    do_eqtn_k(Reset_term,Anded_term,Eqtnk),
    reset(Next,B,Binary_list,[jk_ff(Eqtnj,Eqtnk)],Str,Set_term,Reset_term).

reset(0,_,List,Str,_) :- Str =.. [sync_bin_ctl|List].
reset(P,B,[Bit|Binary_list],Tmp,Str,Set_term,Reset_term) :-
    Next is (P - 1),form_and(P,Anded_term),
    do_eqtn_j(Set_term,Anded_term,Eqtnj),
    do_eqtn_k(Reset_term,Anded_term,Eqtnk),
    reset(Next,B,Binary_list,[jk_ff(Eqtnj,Eqtnk)|Tmp],Str,Set_term,Reset_term).

do_eqtn_j(Set_term,Anded_term,and(Anded_term,Set_term)).
do_eqtn_k(Reset_term,Anded_term,or(Anded_term,Reset_term)).

process_false(0,[],'').
process_false(1,[1],not('a')).
process_false(1,[0],('a')).
process_false(P,[Bit|Binary_list],and(B1,B2)) :- Next is (P - 1),
    alpha_bit_false(Bit,P,B1),process_false(Next,Binary_list,B2).

```

```

process_true(0,[],'').
process_true(1,[1],('a')).
process_true(1,[0],not('a')).
process_true(P,[Bit|Binary_list],and(B1,B2)) :- Next is (P - 1),
    alpha_bit_true(Bit,P,B1),process_true(Next,Binary_list,B2).

alpha_bit_false(1,P,not(B)) :- alpha(P,B).
alpha_bit_false(0,P,B) :- alpha(P,B).
alpha_bit_true(1,P,B) :- alpha(P,B).
alpha_bit_true(0,P,not(B)) :- alpha(P,B).

```

The design approach is based on the recognition that at the end of the counting sequence the flip-flops must be reset, and that during the counting, a flip-flop is toggled when the flip-flops to the right are all 1's. A *false* and a *true* term are formed to ensure the reset. An anding equation is formed to ensure the toggling, and these equations are *ORed* together to form the input equation to the flip-flop. The following Prolog trace gives the calling sequence as the counter is being designed. Note, that to reduce the length of the trace, some calling sequences have been “jumped” through.

```

Call: design_counter(synchronous binary count by) ?
Call: counter(synchronous binary count by,_33462) ?
Call: count(_33472) ?
Exit: count(23)
Call: binary(23,[_131|_132]) ? leap
Exit: binary(23,[1,0,1,1,1])
Call: number_ff(23,_133) ? leap
Exit: number_ff(23,5)
Call: process_true(5,_33473,_33474) ?

[Sub-calls omitted]

Exit: process_true(5,[1,1,1,1,1],and(e,and(d,and(c,and(b,a))))))
Call: process_false(5,[1,1,1,1,1],_33475) ?

[Sub-calls omitted]

Exit: process_false(5,[1,1,1,1,1],and(not e,and(not d,and(not c,and(not b,not a))))))
Call: _33476 is 5-1 ?
Exit: 4 is 5-1
Call: form_and(5,_33477) ?
Exit: form_and(5,and(a,b,c,d))

```

Call: do_eqtn_j(and(not e,and(not d,and(not c,and(not b,not a))))and(a,b,c,d),_134) ?
 Exit: do_eqtn_j(and(not e,and(not d,and(not c,and(not b,not a))))and(a,b,c,d),
 and(and(a,b,c,d),and(not e,and(not d,and(not c,and(not b,not a))))))
 Call: do_eqtn_k(and(e,and(d,and(c,and(b,a))))and(a,b,c,d),_135) ?
 Exit: do_eqtn_k(and(e,and(d,and(c,and(b,a))))and(a,b,c,d),or(and(a,b,c,d),
 and(e,and(d,and(c,and(b,a))))))
 Call: reset(4,[1,1,1,1],[0,1,1,1],j_k_ff(and(and(a,b,c,d),and(not e,and(not d,
 and(not c,and(not b,not a))))),or(and(a,b,c,d),and(e,and(d,and(c,and(b,a)))))),_33462,
 and(not e,and(not d,and(not c,and(not b,not a))))),and(e,and(d,and(c,and(b,a)))))) ?

[Sub-calls omitted]

Exit: reset(4,[1,1,1,1],[0,1,1,1],
 j_k_ff(and(and(a,b,c,d),and(not e,and(not d,and(not c,and(not b,not a))))),or(and(a,b,c,d),and(e,and(d,and(c,and(b,a))))),
 sync_bin_ct(
 j_k_ff(and(1,and(not e,and(not d,and(not c,and(not b,not a))))),or(1,and(e,and(d,and(c,and(b,a))))),
 j_k_ff(and(a,and(not e,and(not d,and(not c,and(not b,not a))))),or(a,and(e,and(d,and(c,and(b,a))))),
 j_k_ff(and(and(a,b),and(not e,and(not d,and(not c,and(not b,not a))))),or(and(a,b),and(e,and(d,and(c,and(b,a))))),
 j_k_ff(and(and(a,b,c),and(not e,and(not d,and(not c,and(not b,not a))))),or(and(a,b,c),and(e,and(d,and(c,and(b,a))))),
 j_k_ff(and(and(a,b,c,d),and(not e,and(not d,and(not c,and(not b,not a))))),or(and(a,b,c,d),
 and(e,and(d,and(c,and(b,a))))),and(not e,and(not d,and(not c,and(not b,not a))))),
 and(e,and(d,and(c,and(b,a))))))
 Exit: counter(synchronous binary count by,sync_bin_ct(
 j_k_ff(and(1,and(not e,and(not d,and(not c,and(not b,not a))))),or(1,and(e,and(d,and(c,and(b,a))))),
 j_k_ff(and(a,and(not e,and(not d,and(not c,and(not b,not a))))),or(a,and(e,and(d,and(c,and(b,a))))),
 j_k_ff(and(and(a,b),and(not e,and(not d,and(not c,and(not b,not a))))),or(and(a,b),and(e,and(d,and(c,and(b,a))))),
 j_k_ff(and(and(a,b,c),and(not e,and(not d,and(not c,and(not b,not a))))),or(and(a,b,c),and(e,and(d,and(c,and(b,a))))),
 j_k_ff(and(and(a,b,c,d),and(not e,and(not d,and(not c,and(not b,not a))))),or(and(a,b,c,d),and(e,and(d,and(c,and(b,a))))).

7.5 Logic Minimization

Logic minimization is provided as a general purpose combinational logic reduction tool. Its design has been guided by the desire to provide logic minimization with reasonable processing time, and is implemented using Prolog *re-write rules* as described in section 5.2.3. To illustrate how these rules are used in *PCD*, let us consider the minimization of $(1+b+c).(1+d)$ using *PCD*. The following is a minimization session with *PCD*.

```
% prolog
C-Prolog version 1.5
!?- [Pcd].

Prolog Counter Design
(type "commands." for listing of available commands)
Version: April 14 1987

==> get(ex1).
Circuit structure retrieved

==> print(ex1).
print consulted 3092 bytes 0.866666 sec.
ex1 with input No 1;
  ((1+b+c).(1+d)))

==> min(ex1).
min consulted 4176 bytes 1.1 sec.
ex1 with input No 1;
  1
Do you wish to store this NEW design y/n >
y.
Enter design name (atom) min_ex1.
==> designs.
Designs currently in memory
min_ex1
ex1

Designs saved to file;
other_design
==> exit.

[ Prolog execution halted ]
```

The following C Prolog trace shows the rules which are being called to perform the minimization.

```
Call: min_str(ex1(and(or(1,b,c),not or(1,d))),_33376) ?
Call: integer(ex1(and(or(1,b,c),not or(1,d)))) ?
Fail: integer(ex1(and(or(1,b,c),not or(1,d))))
Call: atom(ex1(and(or(1,b,c),not or(1,d)))) ?
Fail: atom(ex1(and(or(1,b,c),not or(1,d))))
Call: var(ex1(and(or(1,b,c),not or(1,d)))) ?
Fail: var(ex1(and(or(1,b,c),not or(1,d))))
Back to: min_str(ex1(and(or(1,b,c),not or(1,d))),_33376) ?
```

Call: ex1(and(or(1,b,c),not or(1,d)))=..[_216|_217] ?
 Exit: ex1(and(or(1,b,c),not or(1,d)))=..[ex1,and(or(1,b,c),not or(1,d))]
 Call: min_list_str([and(or(1,b,c),not or(1,d))],[],_218) ?
 Call: min_str(and(or(1,b,c),not or(1,d)),_231) ?
 Call: min_str(or(1,b,c),_234) ?
 Call: integer(or(1,b,c)) ?
 Fail: integer(or(1,b,c))
 Call: atom(or(1,b,c)) ?
 Fail: atom(or(1,b,c))
 Call: var(or(1,b,c)) ?
 Fail: var(or(1,b,c))
 Back to: min_str(or(1,b,c),_234) ?
 Call: or(1,b,c)=..[_240|_241] ?
 Exit: or(1,b,c)=..[or,1,b,c]
 Call: min_list_str([1,b,c],[],_242) ?
 Call: min_str(1,_255) ?
 Call: integer(1) ?
 Exit: integer(1)
 Exit: min_str(1,1)
 Call: min_list_str([b,c],[1],_242) ?
 Call: min_str(b,_261) ?
 Call: integer(b) ?
 Fail: integer(b)
 Call: atom(b) ?
 Exit: atom(b)
 Exit: min_str(b,b)
 Call: min_list_str([c],[b,1],_242) ?
 Call: min_str(c,_269) ?
 Call: integer(c) ?
 Fail: integer(c)
 Call: atom(c) ?
 Exit: atom(c)
 Exit: min_str(c,c)
 Call: min_list_str([], [c,b,1],_242) ?
 Call: rev([c,b,1],[],_242) ? leap
 Exit: rev([c,b,1],[],[1,b,c])
 Exit: min_list_str([], [c,b,1],[1,b,c])
 Exit: min_list_str([c],[b,1],[1,b,c])
 Exit: min_list_str([b,c],[1],[1,b,c])
 Exit: min_list_str([1,b,c],[],[1,b,c])
 Call: _234=..[or,1,b,c] ?
 Exit: or(1,b,c)=..[or,1,b,c]
 Exit: min_str(or(1,b,c),or(1,b,c))
 Call: min_str(not or(1,d),_235) ?
 Call: min_str(or(1,d),_297) ?
 Exit: min_str(or(1,d),1)
 Exit: min_str(not or(1,d),not 1)
 Call: min1(and(or(1,b,c),not 1),_231) ?
 Exit: min1(and(or(1,b,c),not 1),and(or(1,b,c),not 1))
 Exit: min_str(and(or(1,b,c),not or(1,d)),and(or(1,b,c),not 1))
 Call: min_list_str([], [and(or(1,b,c),not 1)],_218) ?
 Call: rev([and(or(1,b,c),not 1)],[],_218) ?
 Call: rev([], [and(or(1,b,c),not 1)],_218) ?
 Exit: rev([], [and(or(1,b,c),not 1)], [and(or(1,b,c),not 1)])

```

Exit: rev([and(or(1,b,c),not 1)],[],[and(or(1,b,c),not 1)])
Exit: min_list_str([],[and(or(1,b,c),not 1)],[and(or(1,b,c),not 1)])
Exit: min_list_str([and(or(1,b,c),not or(1,d))],[],[and(or(1,b,c),not 1)])
Call: _33376=..[ex1,and(or(1,b,c),not 1)] ?
Exit: ex1(and(or(1,b,c),not 1))=..[ex1,and(or(1,b,c),not 1)]
Exit: min_str(ex1(and(or(1,b,c),not or(1,d))),ex1(and(or(1,b,c),not 1)))

```

7.6 NAND Logic Adoption

PCD has a set of logic clauses which convert a circuit structure defined using *AND* or *OR* gates into a logically equivalent circuit which is built using only *NAND* and *NOT* gates. The same Prolog re-write rules that were used for logic minimization are applied to the logic conversion problem. The use of logic conversion is illustrated in the *PCD* session in 7.8.

7.7 Functional Simulation

PCD is provided with a set of logic clauses to perform logic simulation at the gate level. If *PCD* had a clausal circuit definition, then logic simulation would have been almost directly obtainable from the definition of the circuit. These clauses illustrate that it is possible to obtain logic simulation from a data structure represented circuit using Prolog, although not as elegantly.

The approach taken is to first determine all the signals of the circuit and the values of all variables. Then using the axiom values for the “known” gates, the circuit can be evaluated.

This approach is used to simulate synchronous and asynchronous counters by using the convention of letters “a” , “b” etc., to designate the output from the circuit on the previous pulse. This works fine provided input signals to a one output circuit are not

named “a” or, “a” or “b” for a two output circuit.

7.7.1 Functional Simulation Example

Let us consider the functional simulation of a synchronous modulus binary counter, which counts from 0 to 7. The circuit definition for this counter is as follows

```
sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
             jk_ff(and(a,b,c),and(a,b,c),clk))
```

A simulation of that circuit would look as follows;

```
Prolog Counter Design
(type "commands." for listing of available commands)
Version: May 15 1987
```

```
==> get(sim).
Circuit structure retrieved

==> print(sim).
print consulted 3400 bytes 0.966668 sec.
sync_bin_mod input 1;
J-K flipflop with J input;
1
K input;
1
Clock input; clk

sync_bin_mod input 2 ;
J-K flipflop with J input;
a
K input;
a
Clock input; clk

sync_bin_mod input 3 ;
J-K flipflop with J input;
(a.b)
K input;
(a.b)
Clock input; clk

sync_bin_modInput 4;
J-K flipflop with J input;

(a.b.c)
```

K input;

(a.b.c)

Clock input; clk

==> **simulate(sim).**

simulate consulted 7644 bytes 2.36667 sec.

Warning: Circuit has fixed "1" input value

Warning: Circuit has fixed "1" input value

What value should clk have ? 0.

What value should a have ? 0.

What value should b have ? 0.

What value should c have ? 0.

Circuit state is

[d,c,b,a]

[0,0,0,1]

Do you wish to continue the simulation ? <y/n>. y.

Circuit state is

[d,c,b,a]

[0,0,1,0]

Do you wish to continue the simulation ? <y/n>. y.

Circuit state is

[d,c,b,a]

[0,0,1,1]

Do you wish to continue the simulation ? <y/n>. y.

Circuit state is

[d,c,b,a]

[0,1,0,0]

Do you wish to continue the simulation ? <y/n>. y.

Circuit state is

[d,c,b,a]

[0,1,0,1]

Do you wish to continue the simulation ? <y/n>. y.

Circuit state is

[d,c,b,a]

[0,1,1,0]

Do you wish to continue the simulation ? <y/n>. y.

Circuit state is

[d,c,b,a]

[0,1,1,1]

Do you wish to continue the simulation ? <y/n>. y.

Circuit state is

[d,c,b,a]

[1,0,0,0]

Do you wish to continue the simulation ? <y/n>. y.

Circuit state is

[d,c,b,a]

[1,0,0,1]

Do you wish to continue the simulation ? <y/n>. y.

Circuit state is

[d,c,b,a]


```

[1,0,1,0]
Do you wish to continue the simulation ? <y/n>. y.
Circuit state is
[d,c,b,a]
[1,0,1,1]
Do you wish to continue the simulation ? <y/n>. y.
Circuit state is
[d,c,b,a]
[1,1,0,0]
Do you wish to continue the simulation ? <y/n>. y.
Circuit state is
[d,c,b,a]
[1,1,0,1]
Do you wish to continue the simulation ? <y/n>. y.
Circuit state is
[d,c,b,a]
[1,1,1,0]
Do you wish to continue the simulation ? <y/n>. y.
Circuit state is
[d,c,b,a]
[1,1,1,1]
Do you wish to continue the simulation ? <y/n>. y.
Circuit state is
[d,c,b,a]
[0,0,0,0]
Do you wish to continue the simulation ? <y/n>. n.
Simulation Concluded.

```

Appendix B contains a listing of the Prolog trace through this simulation, and shows the calling sequence to perform the simulation.

7.8 PCD Session

The following design session illustrates how a variety of design activities would be performed using PCD.

```

% prolog
C-Prolog version 1.5
! ?- [pcd].

```

Prolog Counter Design
 (type "commands." for listing of available commands)
 Version: May 18 1987
=> commands.

The following are a list with a brief description of commands available within PCD.

check(design)	Performs a syntax check on the definition of "design".
clear	Remove specs & circuits
commands	Lists all available commands (this help).
convert(design)	Generates NAND circuit representation.
counter	Design a counter.
designs	Lists all designs in memory & file.
exit	Terminates the current PCD session.
fan(design)	Calculate maximum fan in in circuit "design".
get(design)	Retrieves the named design from file.
help	Gives general help information.
min(design)	Generates a logic minimal representation of the named design.
print(design)	Displays the design on the terminal.
save (design)	Saves the design to file.
shell	Create a C- shell session with PCD.
store(design)	Stores a particular design in memory.
simulate(design)	Simulates a circuit represented by the named design.

=> help(clear).
 help_lib consulted 6464 bytes 0.800002 sec.

This command removes any design stored in memory, and also clears all counter circuit specifications.

=> help(switchtail).

A Switchtail counter is a modification of a ring counter. In the Switchtail, the interconnections between FF's is the same, but the connections at the end of the cas- cade of flip-flops are reversed.

=> counter.
 counter consulted 10764 bytes 3.16667 sec.
 Please enter Counter Circuit Specifications
 Enter "help." for available options >> **help.**

count(n)	:Count from zero to n
grey	:Count greycode sequence
shift	:Count a shift sequence
bcd	:Count 0 .. 9 and reset
modulo	:Count 0 .. 15 and reset

ff(n) :Number of flip flops
 clock(n) :Circuit clock speed nSec
 delay(n) :Max Circuit Timing delay = n

The following are example entries:

"count(23),clock(28)."

"shift,ff(3),delay(27)."

Please enter Counter Circuit Specifications

Enter "help." for available options >> count(23),clock(28).

Counter Circuit Specification syntax check OK

print consulted 3092 bytes 0.933333 sec.

sync_bin_ct input 1;

J-K flipflop with J input;

(1.~(e).~(d).~(c).~(b).~(a))

K input;

(1+(e.d.c.b.a))

sync_bin_ct input 2 ;

J-K flipflop with J input;

(a.~(e).~(d).~(c).~(b).~(a))

K input;

(a+(e.d.c.b.a))

sync_bin_ct input 3 ;

J-K flipflop with J input;

((a.b).~(e).~(d).~(c).~(b).~(a))

K input;

((a.b)+(e.d.c.b.a))

sync_bin_ct input 4 ;

J-K flipflop with J input;

((a.b.c).~(e).~(d).~(c).~(b).~(a))

K input;

((a.b.c)+(e.d.b.c.b.a))

sync_bin_ct input 5;

J-K flipflop with J input;

((a.b.c.d).~(e).~(d).~(c).~(b).~(a))

K input;

((a.b.c.d)+(e.d.c.b.a))

Do you wish to store this NEW design y/n > y.

Enter design name (atom) ct23.

=> min(ct23).

min consulted 4136 bytes 1.05001 sec.

sync_bin_ct input 1;

J-K flipflop with J input;

(~(e).~(d).~(c).~(b).~(a))))

K input;

1

```

sync_bin_ct input 2 ;
J-K flipflop with J input;
(a.(~(e).(~(d).(~(c).(~(b).~(a))))))
K input;
(a+(e.(d.(c.(b.a))))))

sync_bin_ct input 3 ;
J-K flipflop with J input;
((a.b).(~(e).(~(d).(~(c).(~(b).~(a))))))
K input;
((a.b)+(e.(d.(c.(b.a))))))

sync_bin_ct input 4 ;
J-K flipflop with J input;
((a.b.c).(~(e).(~(d).(~(c).(~(b).~(a))))))
K input;
((a.b.c)+(e.(d.(c.(b.a))))))

sync_bin_ct input 5;
J-K flipflop with J input;
((a.b.c.d).(~(e).(~(d).(~(c).(~(b).~(a))))))
K input;
((a.b.c.d)+(e.(d.(c.(b.a))))))
Do you wish to store this NEW design y/n > y.

```

Enter design name (atom) **min_ct23.**

```

==> designs.
Designs currently in memory
min_ct23
ct23

```

```

Designs saved to file;
ct23
==> get(sim).
Circuit structure retrieved

```

```

==> print(sim).
print consulted 3400 bytes 1.11667 sec.
sync_bin_mod input 1;
J-K flipflop with J input;
1
K input;
1
Clock input; clk

```

```

sync_bin_mod input 2 ;
J-K flipflop with J input;
a
K input;
a
Clock input; clk

```

```

sync_bin_mod input 3 ;

```

J-K flipflop with J input;
 (a.b)
 K input;
 (a.b)
 Clock input; clk

sync_bin_mod input 4;
 J-K flipflop with J input;
 (a.b.c)
 K input;
 (a.b.c)
 Clock input; clk
 ==> shell.

% ls

check	help_lib	prpcd
check.BAK	help_lib.BAK	read_in
check.CKP	help_lib.CKP	ring.trace
convert	min	ring.trace.BAK
convert.BAK	min.BAK	s.trace
convert.CKP	min.session	sim.circuit
counter	min.session.BAK	sim.circuit.BAK
counter.BAK	min.session.CKP	sim.session
counter.CKP	min.session.trace	
ct23.select.trace	min.session.trace.BAK	
ct23.syn.trace	sim.session.BAK	
ct23.syn.trace.BAK	sim.session.CKP	
ct23.syn.trace.CKP		
ct23.trace.BAK	pcd	sim.trace
ct23.trace.CKP	pcd.BAK	sim.trace.BAK
designs	pcd.CKP	sim.trace.CKP
doc_files	pcd.log	simulate
doc_files.BAK	pcd.log.BAK	simulate.BAK
fan	pcd.log.CKP	simulate.CKP
fan.BAK	pcd.session	t.CKP
fan.CKP	pcd.session.BAK	tmp
help	pcd.session.CKP	typescript
	print	typescript.BAK
	print.BAK	
	print.CKP	

% date

Mon May 18 18:22:02 MDT 1987

% ruptime

cs-apollo-a	up 20+08:28,	1 user,	load 0.00, 0.00, 0.00
cs-sun-fsa	up 7+10:56,	0 users,	load 0.00, 0.00, 0.00
cs-sun-fsb	up 7+10:56,	0 users,	load 0.00, 0.00, 0.00
enel-fusion	up 20+04:46,	0 users,	load 0.00, 0.00, 0.00
enel-sun2-c	up 9+21:45,	0 users,	load 0.04, 0.00, 0.00
enel-sun3-a	up 29+18:24,	0 users,	load 0.03, 0.00, 0.00
enel-sun3-d	up 4+10:01,	0 users,	load 0.00, 0.00, 0.00
enel750	up 6+07:03,	1 user,	load 1.02, 1.09, 1.07
engg-sun3-a	up 3+05:34,	1 user,	load 0.15, 0.20, 0.00
engg-sun3-b	up 3+05:31,	0 users,	load 0.04, 0.00, 0.00
evdsvax	up 9+21:45,	1 user,	load 0.08, 0.08, 0.09
pepr	up 14+02:01,	1 user,	load 0.95, 0.53, 0.16

```

ssgvax      up 9+21:45,    0 users,   load 0.01, 0.03, 0.04
vaxa        up 6+07:15,    0 users,   load 1.85, 1.71, 1.92
vaxb        up 7+10:47,    7 users,   load 1.31, 1.30, 1.42
vaxc        up 3+04:57,    0 users,   load 0.02, 0.03, 0.02
vaxd        up 3+10:22,    0 users,   load 0.00, 0.02, 0.03

```

```
% ^D
```

```
==> min(sim).
```

```
min consulted 4216 bytes 1.13333 sec.
```

```
sync_bin_mod input 1;
```

```
J-K flipflop with J input;
```

```
1
```

```
K input;
```

```
1
```

```
Clock input; clk
```

```
sync_bin_mod input 2 ;
```

```
J-K flipflop with J input;
```

```
a
```

```
K input;
```

```
a
```

```
Clock input; clk
```

```
sync_bin_mod input 3 ;
```

```
J-K flipflop with J input;
```

```
(a.b)
```

```
K input;
```

```
(a.b)
```

```
Clock input; clk
```

```
sync_bin_mod input 4;
```

```
J-K flipflop with J input;
```

```
(a.b.c)
```

```
K input;
```

```
(a.b.c)
```

```
Clock input; clk
```

```
Do you wish to store this NEW design y/n > n.
```

```
Warning: circuit not stored
```

```
==> clear.
```

```
Counter specs cleared
```

```
==> exit.
```

```
[ Prolog execution halted ]
```

CHAPTER 8

Summary & Conclusions

In this chapter, important points that have been brought out during the thesis are summarized. Concluding remarks present a point form summary of the major arguments in favour of a logic programming approach to logic design.

8.1 Summary

In this thesis, I have described various logic programming techniques for digital system design. Chapter one summarized the current state of CAD design system. In describing these systems, the reader is introduced to the *need* for more intelligent CAD design systems which actively encourage the formulation of better designs.

In chapter two, the circuit design technique called “logic synthesis” was introduced as a circuit design approach which is most appropriate for logic programming.

Chapter three introduced and justified the use of Prolog as a clausal logic programming language. Then chapter four explored circuit storage and manipulation in Prolog. Chapter four clearly indicated the profound impact that representation has on the types of manipulations that can be performed. Chapter five described logic minimization and conversion techniques using Prolog re-write rules. In chapter five it was shown that when a Prolog rewrite rule based logic minimization system is employed, it is important that the application of the rules is controlled so that the search for the logic equivalent *minimal* expression is obtained, without risk of recursion or computational explosion.

Chapter six served to introduce the design criteria for counters, to form the basis for the experimental system described in chapter seven. Chapters seven concluded the thesis with a demonstration of a counter logic design system, which is used to illustrate the techniques discussed in the thesis. This system, written in Prolog, is called *Prolog Counter Design* (PCD), and is listed in Appendix A.

While the use of logic programming for circuit design is new, promising results from preliminary experimental systems suggest that the approach has potential for commercial logic circuit design.

8.2 Conclusions

The conclusion of the research in this thesis suggests that Prolog, or more specifically logic programming, is viable for circuit design. The practicality of this is demonstrated when systems, consisting of over 25000 gates, have been designed using this approach [Fujita 86]. Using a clausal approach to circuit design has several benefits over other circuit design programming techniques, which are summarized as follows;

(1) Flexible Automation

It is usually easy to *add* rules, or otherwise modify the system, since the expert knowledge is a separate entity from the reasoning mechanisms of the system. It is also possible to add or change rules while using the system. An approach which relies completely on executable code is not so readily modifiable. This feature is useful for the designer to adopt the system to the application, and to the system programmer during initial system development.

(2) **Maintaining System Performance**

A logic design system written in Prolog is typically criticized for lack of efficiency when compared to procedural languages like Fortran. Efficiency should not be a problem for Prolog provided two criteria are met. First any portion of the system which lends itself to procedural execution, should be programmed with a procedural language such as Fortran or C. Secondly the Prolog program should be programmed to avoid backtracking, unless it is worthwhile or is controlled. If these criteria can be met, a Prolog system can be comparable to Fortran in most aspects of logic design. In addition, the emergence of more powerful engineering design workstations will blur the efficiency distinction by allowing practical problems to be programmed in Prolog.

(3) **Rule-based Transformations**

For such conversions as logic minimization, or conversion to technology specific logic, Prolog can express clearly the transformation rules. The clearer style adds comprehension, reliability and, is intuitively a more correct approach.

These advantages must be considered against a requirement to carefully consider program control in Prolog. Without adequate control, a Prolog program becomes ineffective at solving the problem it was developed for.

8.3 Future Research

This thesis has broken new ground in the use of logic programming for circuit design in the areas of circuit synthesis, logic minimization and logic simulation at the functional level. Several research topics can be pursued in this area.

(1) **Logic Minimization & Prolog**

This thesis has shown that Prolog can directly express minimization rules. Further research can be applied to compare efficiency and performance of alternative guided search strategies. Traditional logic minimization only considers a cost function based on gate count. Prolog should be able to help minimize circuits based on more complex minimization functions.

(2) **Logic Simulation & Prolog**

Prolog has many features which facilitate logic simulation, and current research in this area is promising. Look for the development of micro processor simulation programs in Prolog. Topics in the use of Concurrent Prolog for solving logic timing problems are also wide open.

BIBLIOGRAPHY

In the following reference list the asterisk following a reference indicates that a personal copy of that reference has been obtained.

[Adshead 81]*

H. G. Adshead, *DA4 - An Integrated Design System*, European Conference on Electronic Design Automation Sept. 1981 page 1-4.

[Arevalo 78]*

Z. Arevalo and J G Bredeson, *A Method to simplify a Boolean function into a near minimum sum of products for PLAs*, IEEE Trans. on Comp., Vol C-27, No 11 p1078-1039 Nov 1978.

[Asija 68]

S. P. Asija, *Instant Logic Conversion* , in IEEE Spectrum, Vol. 5 ,December 1968, p77-80.

[Besslich 81]*

Ph. W. Besslich and P. Pichlbauer, *Fast transform procedure for the generation of near-minimal covers of Boolean functions*, in IEE Proc., Vol 128, Pt E No 6, November 1981.

[Bogert 87]

Bogert & Thomas Research, *CAD/CAE and VLSI Design* , published by Bogert/Thomas Research, Palo Alto, Calif, 1987. Tel. (617)232-8080

[Bowman 68]*

Robert Bowman and E S McVey, *A method for the fast approximate solution of large prime implicant charts*, IEEE Trans on Comp. Vol C-21 page 169-173, 1972

[Brayton 84]*

R.K. Brayton, G.D. Hachtel, C.T. McMullen, A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Netherlands, 1984.

[Breuer 72]

M. A. Breuer, (Ed.), *Design Automation of Digital Systems*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972.

[Brewer 86]*

F.D. Brewer, D.D. Gajski, *An Expert-System Paradigm for Design*, Proc 23rd ACM/IEEE Design Automation Conf 1986 p62-68.

[Bubenik 72]

V. Bubenik, *Weighting method for the determination of the irredundant set of prime implicants*, IEEE Trans. on Comp. Vol C-21 p1445-1451 1972.

[Bundy 81]*

A. Bundy, and B. Welham, *Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulations*, in Artificial Intelligence 16(2) 1981.

[Bundy 83]*

A. Bundy, *The Computer Modelling of Mathematical Reasoning*, published by Academic Press, London, 1983.

[Button 60]

C. H. Button, H. J. Grosskamp, J. L. Kenney, M. R. Murphy, and R. L. Simek, *Parts Usage Maintenance Program PUMP* IBM Technical Report 00.746 (Poughkeepsie, New York, October 5, 1960).

[Campbell 84]

J. A. Campbell, *Implementations of PROLOG*, a collection of papers in the Ellis Horwood Series in

Artificial Intelligence published by Halsted Press, 1984.

[Chang 73]

C. L. Chang, R Lee, *Symbolic logic and methematical theorem proving* , Academic Press, 1973.

[Clark 82/1]

K. L. Clark & F. G. McCabe, *Prolog: A language for implementing Expert Systems* , Department of computing, Imperial College, Technical Report no 80/21.

[Clark 82/2]

K. L. Clark, *IC Prolog: Aspects of its implementation* , in Proc of the Logic Programming Workshop, Debrecen, 1980.

[Clark 82]

K. L. Clark, S. A. Tarnlund (eds), *Logic Programming* , 1982, p153-172, Acedemic Press, New York, NY.

[Clocksin 81]*

W. F. Clocksin, and C. S. Mellish, *Programming in Prolog* , published by Springer-Verlag, 1981.

[Clocksin 85]

W. F. Clocksin, *Logic programming and the specification of circuits* , Computer Laboratory, University of Cambridge Technical Report No 72, 1985.

[Covington 85]

M. A. Covington, *Eliminating unwanted loops in Prolog* , in ACM SIGPLAN notices Vol 20, 1, pp20-26 Jan 1985.

[Craig 86]

I. D. Craig, *The Adriadne - 1 Blackboard System* , in The Computer Journal, Vol 29, No 3 p235 to

p240.

[Cray 56]

S. R. Cray and R. N. Kisch, *A Progress Report on Computer Applications in Computer Design* , in Proceedings of the Western Joint Computer Conference (1956), p 82-85.

[Darringer 69]

J Darringer, *The Description, Simulation, and Automatic Implementation of Digital Computer Processors* ,Ph. D. Thesis, Carnegie- Mellon University, Pittsburg, PA 1969.

[Darringer 80]

J. A. Darringer, W. H. Joyner, L. Berman, and L. Trevillyan, *Experiments in Logic Synthesis* , in Proceedings of the IEEE International Conference on Circuits and Computers, Port Chester, NY, 1980 page 234-237.

[Darringer 81]*

J. Darringer, W. Joyner, L. Berner, L. Trevillyan, *Logic Synthesis through local transformations*, IBM Journal of Research & Development, Vol. 25, July 1981, page 272-280.

[Darringer 84]*

J. A. Darringer, D. Brand, J. V. Gerbi, W. H. Joyner, L. Trevillyan, *LSS: A System for production Logic Synthesis* , in IBM Journal of Research & Development Vol 28 No 5 Sept. 1984.

[Das 71]

S.R. Das, *Comments on 'A new algorithm for generating prime implicants'*, IEEE Trans. on Comp. Vol C-20 p1614-1615 Dec 1971.

[deGeus 85]*

A.J. de Geus, W.W. Cohen, *A Rule Based System for Optimizing Combinational Logic*, IEEE Design and Testing of Computers, August 1985, page 22-32.

[Dietmeyer 78]

D.L. Dietmeyer, *Logic Design of Digital Systems*, Second Edition, Allyn and Bacon Inc. 1978.

[Duley 68]

J.R. Duley, *DDL A Digital Design Language*, Ph. D. Thesis, University of Wisconsin, Madison, WI, 1968.

[Falkoff 64]

A. D. Falkoff, K. E. Iverson, and E. H. Sussenguth, *Formal description of system/360* in the IBM System Journal Vol 3 pages 198-262, 1964.

[Feigenbaum 83]

E. A. Feigenbaum, and P. Mc Corduck, *The Fifth Generation*, published by Addison-Wesley, Reading, MA, 1983.

[Forgy 81]

C. L. Forgy, *OPS5 User's Manual*, Department of Computer Science, Carnegie-Mellon University, July 1981.

[Fox 84]*

J. R. Fox, *Performance Prediction with the MacPitts Silicon Compiler*, in IEEE proc on Computer Hardware, page 351-355, 1984.

[Friedman 69]*

T.D. Friedman, and S. C. Yang, *Methods Used in an Automatic Logic Design Generator (ALERT)*, in IEEE Transactions on Computers, Vol C-18, No 7 July 1969 page 593-614.

[Friedman 75]

A.D. Friedman, *Logical Design of Digital Systems*, Computer Science Press, 1975 page 72.

[Fujita 86]*

M. Fujita, M. Ishisona, H. Nakamura, H. Tanaka, & T. Moto-oka, *Using the temporal logic programming language Tokio for algorithm description and automatic CMOS gate array synthesis*, in Proc of the 4th Logic Programming Conference, held in Tokyo, July 1 - 3 1985, published by Springer-Verlag Berlin 1986, edited by Eiita Wada.

[Gregory 86]*

D. Gregory, K. Bartlett, A. deGeus, G. Hachtel, *SOCRATES: A System for Automatically Synthesizing & Optimizing Combinational Logic*, Proceedings from the 23rd ACM/IEEE Design Automation Conference June 29 - July 2, 1986.

[Gullichen 85]*

E. Gullichen, *Heuristic Circuit Simulation using Prolog Integration*, in the VLSI Journal, Vol 3, p282-318, 1985.

[Hammond 83]

P. Hammond & M. Sergot, *A Prolog Shell for Logic Based Expert Systems*, Special report, Dept. of Computing, Imperial College of Science and Technology, London.

[Hong 74]*

S.J. Hong, R.G. Cain, D.L. Ostapko, *MINI: A Heuristic approach for logic minimization*, IBM J Res & Develop. p443-458, September 1974.

[Horstmann 83]*

P.W. Horstmann, *Expert System & Logic Programming for CAD in VLSI Design* (USA periodical) Nov 1983, p34-40.

[Horstmann 84]*

P.W. Horstmann, *Computer Aided design(CAD) using logic programming (for VLSI)*, IEEE 21st

design automation conf proceedings p144-151, June 25-27 1984.

[Hu 84]*

Y.H. Hu, and D.Y. Yun, *Application of Artificial Intelligence to VLSI CAD systems*, in Proc of IEEE Conf on Comp Design 1984, P737-841, 8-11 Oct 1984.

[Hulme 75]

B.L. Hulme and R.B. Worrell, *A prime implicant algorithm with factoring*, IEEE Trans on Comp. Vol C-24 p1129-1131 Nov 1973.

[Johannsen 79]

D. Johannsen, *Bristle Blocks: A Silicon Compiler*, Proceedings of the 16th Design Automation Conference, June 1979, pp 310-313.

[Joyner 86]*

W.H Joyner, L.H Trevillyan, D. Brand, T.A Nix, S.C Gunderson, *Technology Adoption in Logic Synthesis*, Proceedings from the 23rd ACM/IEEE Design Automation Conference, June 29 July 2, 1986.

[Karnaugh 53]

M. Karnaugh, *The map method for synthesis of combinational logic circuits*, Trans AIEE Vol 72 Part 1 p 593-598 1953.

[Kaskey 61]

Y. Kaskey, H. Lukoff, and N. S. Prywes, *Application of Computers to Circuit Design for UNIVAC LARC*, in Proceedings of the Western Joint Computer Conference Vol 19 (1961) p 185-205.

[Kowalski 83]*

T. J. Kowalski and D. E. Thomas, *The VLSI Design Automation Assistant: Prototype System*, in Proc. of the 20th Design Automation Conference, Miami, Florida 1983.

[Liblong 84/1]*

B. Liblong, *SHIFT a Structured Hierarchic Intermediate Form for VLSI Design*, Masters Thesis, University of Calgary, 1984.

[Liblong 84/2]*

B. Liblong, T. Melham, G. Birtwistle, J. Kendall, *Towards a VLSI Design Tool System*, Research Report 84/175/33, Nov 84, Dept of Computer Science, University of Calgary.

[Lipp 79]*

H.M. Lipp, *Current Trends in the Design of Digital Circuits*, in Computer-Aided Design of Digital circuits and systems, G Musgrave editor, page 91-102, published by North-Holland, 1979.

[Merwin 67]

R. W. Merwin, & J. L. SanBorn, *Digital Computers for logical Designs*, in M Klerer & G Korn, eds *Digital Computer User's Handbook*, New York, Mc Graw-Hill Book Company, 1967, part 4 p167-192.

[McCluskey 56]*

E.J. McCluskey, *Minimization of Boolean Functions*, in The Bell System Technical Journal, page 1417-1444, November 1956.

[McDermott 85]*

R.M. McDermott, *Computer-Aided Logic Design*, published by H W Sams & Co Ltd, Indianapolis, Indiana, 1985.

[McKinsey 84]

McKinsey and Associates, *Report to Participants*, Toronto, 1984.

[Mead 80]*

C.A. Mead and L Conway, *Introduction to VLSI Systems*, Addison Wesley, Reading Mass 1980.

[Miller 65]

R. E. Miller, *Switching Theory, Vol. 1: Combinatorial Circuits*, John Wiley & Sons, Inc., New York, 1965.

[Morris 76]

Noel M. Morris, *Logic Circuits*, McGraw Hill 1976.

[Naish 83/1]*

Lee Naish, *An Introduction to Mu Prolog*, Technical Report, Department of Computer Science, University of Melbourne, February 1982, revised July 1983.

[Naish 83/2]*

Lee Naish, *MU-Prolog 3.0 Reference Manual*, Reference manual included with delivery of MU-Prolog interpreter, Melbourne University, July 1983.

[Noda 86]*

Y. Noda, T. Kinoshita, A. Okumura, T. Hirano, T. Hiruta, *A Parallel Logic Simulator based on Concurrent Prolog*, in Proc of the 4th Logic Programming Conference, held Tokyo July 1 - 3, 1985, published by Springer Verlag Berlin 1986, edited by Eiiti Wada.

[Oberman 70]*

R.M. Oberman, *Disciplines in Combinational and Sequential Circuit Logic*, in Electrical and Electronic Engineering series, McGraw Hill 1970.

[Obyrne 86]*

R.P. O'Byrne and J. Kendall, *Automatic Circuit Design for Digital Counters*, Research Report No 86/232/6, June 1986, Department of Computer Science, University of Calgary.

[Ostapko 74]*

D.L. Ostapko, S.J. Hong, *Generating Test Examples for Heuristic Boolean Minimization*, in IBM

Journal of Research and Development, Sept 1974.

[Pereira 84]*

F. Pereira from material by D Warren, D Bowen, L Byrd and L Pereira, *C-Prolog User's Manual* , supplied with C-Prolog interpreter ver 1.5, Sept 1984.

[Quine 55]

W.V. Quine, *A way to simplify truth functions*, American Math Mon. Vol 62 p627-631, 1955.

[Rhyne 77]*

V. T. Rhyne, P. S. Noe, M. N. McKinny, and U. W. Pooch, *A New Technique for the fast minimization of switching function* IEEE Trans. on Comp. Vol C-26, No 8, p757-764, August 1977.

[Rosenthal 61]

C. W. Rosenthal, *Computing Machine Aids to a Development Project* , in IRE Transactions on Electronic Computers, Vol EC-10 (September 1961). p 400-406.

[Rubin 82]*

S.M. Rubin, *An Integrated Aid for Top-Down Electrical Design*, Special Report, Fairchild Laboratory for Artificial Intelligence research, 4001 Miranda Avenue, Palo Alto, Calif 94304.

[Saito 86]*

T. Saito, S. Hiroyuki, M. Yamazaki, and N. Kawato, *A Rule-based Logic Circuit Synthesis System for CMOS Gate Arrays* in Proc from the 23rd ACM/IEEE Design Automation Conference, June 29 - July 2, 1986, page 594 to 600.

[Sasao 86]*

T. Sasao, *MACDAS: Multi-level AND-OR Circuit Synthesis using Two-Variable Function Generators* , Proceedings from the 23rd ACM/IEEE Design Automation Conference, June 29 - July 2, 1986.

[Southard 83]

J. R. Southard, *MacPitts: An Approach to Silicon Compilation*, Computer 16, No 12 p74-82 (December 1983).

[Suzuki 85]*

N. Suzuki, *Concurrent Prolog as an efficient VLSI design Language*, published in Computer, Feb 1985, page 33-40.

[Taub 80]*

Herbert Taub, *Digital Circuit and Microprocessors*, Mc Graw Hill 1980.

[Teig 86]*

S. Teig, R. Smith & J. Seaton, *Timing Driven Layout of Cell-based ICs*, in VLSI Systems Design, May 1986.

[Wager 81]*

S.J. Wager & S.J. Poulton, *Interactive logic diagrams at the register level*, Proc. of European Conference on Electronic Design Automation Sept 1-4 1981, page 149-153.

[Wayne 85]*

M. R. Wayne, and S. M. Braun, *Looking for Mr Turnkey*, in Proc. of the 22nd ACM/IEEE Design Automation Conference, 1985.

[Xin Da Lu 81]*

Xin Da Lu, *A special purpose VLSI chip dynamic up-down counter*, Computing Lab, Newcastle upon Tyne University 1981.

backtracking - definition of	41
behavioural domain - definition of	15
CAE vendor classification	8
circuit signals	55
circuit transformation	53
clock skew - designing counters	77
CMOS cell arrays - description of	17
commands - list of	82
counter tree	70
counters - definition of	70
data structure representation - advantages	52
design implementation styles	6
design process - definition of	3
design process - history	1
design process - use of CAD	6
design workflow - definition	4
error - compound error reporting	80
gate-array design	16
hardware economy - counter design	74
hierarchical design	2
horn clause representation - characteristics	49
known circuits - definition of	54
logic programming - definition of	39
logic synthesis - definition of	20
logic synthesis - definition of	21
macro-cell design	17
PCD - development goal	79
PCD - example design session	96
PCD - logic synthesis description	85
PCD command interpreter	79
PCD example minimization session	91
physical domain - definition of	15
PLA macro-cells	18
Prolog - processing efficiency	104
race - designing counters	77
regularity - designing counters	72
shift - hierarchic design language	16
Socrates - description of	32
speed - designing counters	72
stability - counter design	74
standard cell design - description of	17
structural domain - definition of	15
system performance - overall	104
VLSI design - custom	16
VLSI design domains	15

[Adshead 81] - DA4: An Integrated Design System	9
[Aoyagi 85] - Tokio	35
[Arevalo 78] - A Method to simplify a Boolean function into ...	61
[Asija 68] - Instant Logic Conversion	66
[Bogert 87] - CAE/CAD and VLSI Design	8
[Brayton 82] - A Comparison of logic minimization ...	62
[Brayton 84] - Espresso IIc: Logic Minization Algorithms for VLSI ...	33
[Breuer 72] - Design Automation of Digital System	21
[Brewer 86] - An Expert-System Paradigm for Design	45
[Brown 74] CMOS Cell Arrays - An Alternative to Gate Arrays	17
[Brown 81] - A State-Machine Synthesizer - SMS	62
[Bundy 81] - Using meta-level inference for selective....	62
[Bundy 83] - The Computer Modelling of Mathematical Reasoning	45,61,67
[Button 60] - Parts Usage Maintenance Program	1
[Campbell 84] - Implementations of Prolog	39
[Chang 73] - Symbolic logic and mathematical theorem proving	39
[Clark 81] - The Control Facilities of IC-Prolog	43
[Clocksin 81] - Programming in Prolog	39
[Craig 86] - The Ariadne - 1 Blackboard System	30
[Cray 56] - A Progress Report on Computer Applications in ...	1
[Darringer 69] - The Description, Simulation,....	22
[Darringer 80]	7
[Darringer 84] - LSS: A System for Production Logic Synthesis	25
[Dietmeyer 78] - Logic Design of Digital Systems	21,32
[Duley -68] - DDL - A Digital Design Language	22,36
[Falkoff 64] - Formal description of System /360	24
[Feigenbaum 83] - The fifth Generation	39
[Forgey 81] - OPS5 User's Manual	38
[Fox 84]	7
[Friedman 69] - Methods used in an Automatic Logic Design ...	24
[Friedman 70] - Quality of designs from an automatic logic ...	22
[Fujita 86] - Using the temporal logic programming ...	7,35,41,110
[Gregory 86] - SOCRATES: A System for Automatically ...	32
[Gullichen 85] - Heuristic Circuit Simulation using Prolog ...	40
[Hammond 83] - A PROLOG Shell for Logic Based Expert Systems	43
[Hong 74] - MINI: A Heuristic Approach for Logic Minimization	62,63
[Johannsen 79] - Bristle Blocks: A Silicon Compiler	22
[Joyner 86] - Technology Adoption in Logic Synthesis	25
[Karnaugh 53] - The map method for synthesis of combinational ...	0
[Kaskey 61] - Applications of Computers to Circuit Design for ...	1
[Kowalski 83] - The VLSI Design Automation Assistant: Prototype ...	38
[Liblong 84] - SHIFT - A Structured Hierarchical Intermediate form ...	16
[Lipp 79] - Current Trends in the Design of Digital Circuits	23
[McCluskey 56] - Minimization of Boolean Functions	60
[McDermott 85] - Computer Aided Logic Design	57

[McKinsey 84] - Report to Participants	6
[Merwin 67] - Digital Computers for Logic Design	66
[Naish 83] - An Introduction to Mu Prolog	41
[Naish 83] - MU-Prolog 3.0 Reference Manual	41
[Noda 86] - A Parallel Logic Simulator based on Concurrent Prolog	40
[Pereira 84] - C Prolog User's Manual	39
[Quine 55] - A Way to Simplify Truth Functions	60
[Rhyne 77] - A new technique for the fast minimization of ...	61
[Rosenthal 61] - Computing Machine Aids to ...	1
[Rubin 82] - An Integrated Aid for Top-Down Electrical Design	19
[Saito 86] - A Rule-based Circuit Synthesis System for CMOS ...	29
[Sasao 86] - MACDAS: Multi-level AND-OR Circuit Synthesis using ...	31
[Southard 83] - MacPitts: An Approach to Silicon Compilation	22
[Suzuki 85] - Concurrent Prolog as an Efficient VLSI Design Language	41,50
[Teig 86] - Timing Driven Layout of Cell-based ICs	18
[Wager 81] - Interactive Logic Diagrams at the Register Level	10
[Wayne 85] - Looking for Mr Turnkey	8

Appendix A - PCD Listing

PCD (*Prolog Counter Design*) consists of a number of files. These files are;

check	clauses to check the syntax of a circuit definition.
convert	clauses to convert AND/OR/NOT logic to NAND.
counter	clauses to select and synthesis a counter circuit.
fan	clauses to check the maximum fan in of the circuit.
help_lib	clauses to present help information.
min	clauses to perform logic minimization.
pcd	clauses for command interpretation & Utilities.
print	clauses to display a circuit at the terminal.
simulate	clauses to perform circuit simulation.

These files are listed in alphabetical order on the following pages.

```

/*
    CIRCUIT SYNTAX CHECKER
    Created      March 5 1987
    Last Revision April 13 1987
*/
go_check(S,E) :- check_str(S,E).
go_check(S,'ok').

check_str(S,'Error: Circuit not instantiated') :- var(S).
check_str(S,'Error: Signal value not 1 or 0') :- integer(S),(S < 0;S > 1).
check_str(S,E) :- functor(S,Name,N),check_str(S,Name,1,N,E).

check_str(____,E) :- novar(E).

check_str(S,and,P,M,'Error: And gate too few params.') :- (M < 2).
check_str(S,and,M,M,E) :- arg(M,S,A),check_str(A,E).
check_str(S,and,P,M,E) :- Next is P + 1,arg(P,S,A),check_str(A,E),
    check_str(S,and,Next,M,E).

check_str(S,or,P,M,'Error: Or gate too few params.') :- (M < 2).
check_str(S,or,M,M,E) :- arg(M,S,A),check_str(A,E).
check_str(S,or,P,M,E) :- Next is P + 1,arg(P,S,A),check_str(A,E),
    check_str(S,or,Next,M,E).

check_str(S,not,P,M,'Error: Not gate too many params.') :- (M > 1).
check_str(S,not,1,1,E) :- arg(1,S,A),check_str(A,E).
check_str(S,not,____,'Error: Not gate definition').

check_str(S,nand,P,M,'Error: Nand gate too few params.') :- (M < 2).
check_str(S,nand,M,M,E).
check_str(S,nand,P,M,E) :- Next is P + 1,arg(P,S,A),check_str(A,E),
    check_str(S,nand,Next,M,E).

/*
The following clauses allow user defined circuits to be checked
*/
check_str(S,Name,M,M,E) :- arg(M,S,A),check_str(A,E).
check_str(S,Name,P,M,E) :- arg(P,S,A),check_str(A,E),Next is P + 1,
    check_str(S,Name,Next,M,E).

```

```

/*
    NAND/NOT CIRCUIT CONVERSION
    File Name      convert
    File Created   January 1987
    Last Revision  April 14 1987

    Rules Applied in Logic Conversion:

    This file contains clauses necessary to convert a circuit
    represented using general AND, OR and user defined gates
    into an equivalent circuit represented only in NAND.
    The following is a summary of the conversion rules
    applied. Uppercase letters "A", "B" etc. can represent
    circuits in their own right.

    LOGIC CONVERSION RULES

    nand(A,B) = ~(A.B)      nor(A,B) = ~(A+B)
    A.B = ~nand(A,B)        A+B = nand(~A,~B)
    ~A = nand(A,A)          ~A = A.

*/
go_convert(S,Sn) :- var(S),display('Fatal Error: Circuit not instantiated'),abort.
go_convert(S,Sn) :- functor(S,Name,N),conv_nd(S,Name,N,Sn).

conv_nd(S,_,0,S).
conv_nd(S,and,N,not(List)) :- conv_and(S,1,N,[],List).
conv_nd(S,or,N,nand(List)) :- conv_or(S,1,N,[],List).
conv_nd(S,not,1,Sn) :- conv_not(S,Sn).
conv_nd(S,not,_,_) :- display('Circuit Definition error in "not" function'),abort.
conv_nd(S,nand,N,Sn) :- conv_nand(S,1,N,[],Sn).
conv_nd(S,Circuit,N,Sn) :- conv_ct(S,1,N,[],List),conv_list(Sn,Circuit,List).

conv_and(S,N,N,C,Sn) :- arg(N,S,A),go_convert(A,An),conv_list(Sn,nand,[An|C]).
conv_and(S,P,N,C,Sn) :- arg(P,S,A),go_convert(A,An),Next is P+1,
    conv_and(S,Next,N,[An|C],Sn).

conv_or(S,N,N,C,Sn) :- arg(N,S,A),go_convert(A,An),
    conv_list(Sn,nand,[not(An)|C]).
conv_or(S,P,N,C,Sn) :- arg(P,S,A),go_convert(A,An),Next is P+1,
    conv_or(S,Next,N,[not(An)|C],Sn).

conv_not(S,Sn) :- arg(1,S,A),functor(A,not,1),arg(1,A,N),go_convert(N,Sn).
/*      ^ This clause satisfies ~A = A ^
*/
conv_not(S,nand(An,An)) :- arg(1,S,A),go_convert(A,An).

conv_nand(S,N,N,C,Sn) :- arg(N,S,A),go_convert(A,An),conv_list(Sn,nand,[An|C]).
conv_nand(S,P,N,C,Sn) :- arg(P,S,A),go_convert(A,An),Next is P+1,
    conv_nand(S,Next,N,[An|C],Sn).
/*      ^ These clauses allow conversion of a circuit which contains NAND gates
*/
conv_ct(S,N,N,C,[An|C]) :- arg(N,S,A),go_convert(A,An).
conv_ct(S,P,N,C,Sn) :- arg(P,S,A),go_convert(A,An),Next is P+1,

```

```
    conv_ct(S,Next,N,[AnlC],Sn).  
/*      ^ These two clauses allow user defined circuits to be converted.  
*/  
conv_list(S,F,L) :- S =..[F|L].
```

```

/*
    COUNTER SELECTION & DESIGN
    File Name      counter
    File Created   July 1986
    Last Revision  April 14 1987

    This file contains all the clauses required to select and
    synthesize a counter circuit. Counter selection is spec
    driven, with specifications entered by the user. These
    specifications are used to guide the selection process to
    select the most appropriate counter first. A detailed
    explanation of the selection procedure and synthesis is
    included in chapter 7 sections 3.
*/
counter :- type(T),design_counter(T),pcd.
design_counter(no_match) :-
    display('Error selecting a counter type for specifications. Re-specify'),nl.
design_counter(T) :- counter(T,S1),print_str(S1),!,store(S1).
design_counter(T) :- display('Error designing counter'),nl.
type(T) :- display('Please enter Counter Circuit Specifications'),nl,
    display('Enter "help." for available options >> '),
    read(Ans),nl,work_input(Ans,T).
type('no_match').
work_input(help,T) :- nl,select_help,!,type(T).
work_input(exit,no_match).

select_help :- system("more help/specs").
work_input(Ans,T) :- valid_list(Ans),assert_list(Ans),!,select(T).
work_input(Ans,T) :- display('Your list is invalid. Check and reenter'),nl,
    select_help.
valid_list((First_spec,Rest)) :- spec(First_spec),valid_list(Rest).
valid_list(Last_spec) :- spec(Last_spec),
    display('Counter Circuit Specification syntax check OK'),nl.
/*
    V      This is a list of valid user specs which is used to check that
           the user spec is syntactically correct */
spec(bcd).
spec(binary).
spec(clock(_)).
spec(count(_)).
spec(delay(_)).
spec(ff(_)).
spec(grey).
spec(modulo).
spec(ring).
spec(shift).
spec(switchtail).
spec(sync).

assert_list((First,Rest)) :- asserta(First),assert_list(Rest).
assert_list(Last) :- asserta(Last).

select(no_match) :- min_spec(no_match).

```

```

min_spec(no_match) :- bcd,count(N),(N > 9;N < 9),
    display('Error: spec bcd & count conflict'),nl.
/*      ^ These clauses are checking for known conflicts and return "no_match"
        to indicate the conflict.
*/

```

```

select(T) :- sync, sync(T).
select(T) :- delay(N), N <= 25, sync(T).
select(T) :- clock(N), N >= 25, sync(T).
select(T) :- clock(N), N <= 25, async(T).
select(T) :- display('Synchronous type counter'), nl, !, sync(T).
/*      ^ These clauses guide the search to either an asynchronous or
        synchronous counters.
*/
sync(T) :- (count(N); binary), sync_bin(T).
sync('synchronous greyscale serial') :- grey.
sync('synchronous ring') :- ring.
sync('synchronous shift parallel') :- shift.
sync('synchronous switchtail') :- switchtail.
sync('synchronous moebius') :- moebius.
sync('no_match') :- display('Error: no Synchronous counter found'), nl.
/*      ^ These clauses select synchronous counters
*/

```

```

async('asynchronous binary serial ripple').

```

```

sync_bin('synchronous bcd') :- bcd.
sync_bin('synchronous binary full modulo reverse') :- reverse, modulo.
sync_bin('synchronous binary full modulo bi-directional') :-
    up_down, modulo.
sync_bin('synchronous binary full modulo') :- modulo.
sync_bin('synchronous binary count by').

```

```

/*

```

BUILD LOGIC EQUATIONS

Data Structures (S) for circuit with n outputs;

```

circuit_type(S1,S2,... Sn)

```

where S is any valid data structure.

pre-defined valid data structures are;

```

and(S1,S2)      or(S1,S2)      not(S1)

```

```

and(S1,...Sn)   or(S1,...Sn)

```

```

nand(S1,S2)     nor(S1,S2)

```

```

nand(S1,...Sn)  nor(S1,...Sn)

```

new designs are constructed
from valid existing designs.

all valid circuit structures can be evaluated

and return values such as true (1) or false (0) or undefined

(uninstantiated).

```

*/

```

```

binary(Decimal,Result) :-
    binary(Decimal,[],Result).

```

```

binary(1,Binary,Result) :-
    binary(0,[1|Binary],Result).

```

```

binary(0,Binary,Binary).

```

```

binary(Decimal,Binary,Result) :-
    divide(Decimal,Quotient),
    remainder(Decimal, Remainder),
    binary(Quotient, [Remainder|Binary], Result).

/*
divide forms an integer division and returns the
result in Quotient. This is the number that will be divided again.
*/
divide(Decimal,Quotient) :-
    Quotient is Decimal // 2.

/*
The list is formed from the remainder of the division
Remainder uses the modulus function to find the remainder
after division by 2
*/
remainder(Decimal,Remainder) :-
    Remainder is (Decimal mod 2).

/*
"number_ff" is a clause used to calculate the number of flip-flops required
for a counter. If the highest states can be represented then the rest
of the counter can be represented. The number of flip-flops required
to represent a counter with top state 12 is the same as a state 13 counter,
and so "number_ff" returns the same number in both cases
*/

number_ff(Highest_state,Number) :- size_finder(Highest_state,2,1,Number).
/*
size_finder increments the number of ff until the highest state can
be represented. Inter_number is this incrementing variable
*/
size_finder(H,H,N,N).
size_finder(H,I,C,N) :- I > H,!, equal(N,C).
size_finder(Highest_state,Intermediate_state,Inter_number,Number) :-
    New_highest_state is (Intermediate_state * 2),
    New_number is (Inter_number + 1),
    size_finder(Highest_state,New_highest_state,New_number,Number).

counter('asynchronous binary serial ripple',async_bin_ser_rip(List)) :-
    count(Highest_state),
    number_ff(Highest_state,N),
    Next is (N - 1),alpha(Next,Letter),
    ripple(Next,(jk_ff('1','1',not(Letter))),List).

ripple(1,T,(jk_ff('1','1',cp(1)),T)).
ripple(N,Tmp,List) :- Next is (N - 1),alpha(Next,Letter),
    ripple(Next,(jk_ff('1','1',cp(not(Letter))),Tmp),List).

/*
SYNCHRONOUS BINARY FULL MODULO COUNTER
*/

```

```

counter('synchronous binary full modulo',sync_bin_mod(List)) :-
    form_and(4,Eqtn),
    modulo(3,(jk_ff(Eqtn,Eqtn,clk)),List).
modulo(0,List,List).
modulo(N,Tmp,List) :- Next is (N - 1),form_and(N,Eqtn),
    modulo(Next,(jk_ff(Eqtn,Eqtn,clk),Tmp),List).

/*
the form_and clause actually forms the logic equation for the modulo
counter. It is also called for other counters.
*/
form_and(1,1).
form_and(2,'a').
form_and(3,and(a,b)).
form_and(4,and(a,b,c)).
form_and(5,and(a,b,c,d)).
form_and(6,and(a,b,c,d,e)).
form_and(7,and(a,b,c,d,e,f)).

/*
SYNCHRONOUS BINARY FULL MODULO REVERSE
*/
counter('synchronous binary full modulo reverse sequence',sync_bin_mod_rev(List)) :-
    form_and(4,Eqtn),
    reverse_modulo(3,(jk_ff(not(Eqtn),not(Eqtn),clk)),List).

/*
the following clause is the terminating condition for the counter.
We know that for a full modulo counter the least sig. bit always
toggles on a clock input, so we can specify the equation directly.
*/
reverse_modulo(1,List,(jk_ff(1,1,clk),List)).
reverse_modulo(N,Tmp,List) :- Next is (N - 1),form_and(N,Eqtn),
    reverse_modulo(Next,(jk_ff(not(Eqtn),not(Eqtn),clk),Tmp),List).

/*
SYNCHRONOUS BINARY FULL MODULO REVERSIBLE

eqtn_modulo_reversible calculates for a reversible synchronous
binary modulo counter. The control signal U can be either 1 or 0,
where 1 = Forward, and 0 = reverse. When U = 1 E = Eqtn_up, when
U = 0, E = Eqtn_down is the basic rule. This is implemented

with: E = (U.Eqtn_up)+(not(U).Eqtn_down)
*/
counter('synchronous binary full modulo bi-directional',sync_bin_mod_bi(List)) :-
    form_and(4,Eqtn_up),
    reversible_modulo(3,(jk_ff(or(and('u',Eqtn_up),and(not('u'),
not(Eqtn_up)))),or(and('u',Eqtn_up),and(not('u'),not(Eqtn_up))))),clk),List).

/*
following is the terminating clause. It is interesting to note that
the least sig. bit always toggles wether up or down counting.
*/

```



```

reversible_modulo(1,L,(jk_ff(1,1,clk),L)).
reversible_modulo(N,Tmp,List) :- Next is (N - 1),form_and(N,Eqtn_up),
    reversible_modulo(Next,(jk_ff(or(and('u',Eqtn_up),and(not('u'),
        not(Eqtn_up))),or(and('u',Eqtn_up),and(not('u'),not(Eqtn_up)))),clk),Tmp),List).

```

```

/*
SYNCHRONOUS BCD COUNTER
eqtn_bcd calculates the logic equations for
a BCD counter. A Synchronous BCD counter which counts 0 - 9
is the same as a synchronous count by 9 counter.
The value of D will have been checked to be 9.
*/

```

```

counter('synchronous bcd',sync_bcd(L)) :-
    counter('synchronous count by',sync_bin_ct(L)).

```

```

/*
SYNCHRONOUS BINARY COUNT BY COUNTER
eqtn_reset is used to calculate the logic equations for
a synchronous binary count by parallel counter which is not a
synchronous binary full modulo counter.
*/

```

```

counter('synchronous binary count by',Str) :-
    count(D),binary(D,[Bit|Binary_list]),number_ff(D,N),
    process_true(N,B,Reset_term),
    process_false(N,B,Set_term),
    Next is (N - 1),
    form_and(N,Anded_term),
    do_eqtn_j(Set_term,Anded_term,Eqtnj),
    do_eqtn_k(Reset_term,Anded_term,Eqtnk),
    reset(Next,B,Binary_list,[jk_ff(Eqtnj,Eqtnk)],Str,Set_term,Reset_term).

```

```

reset(0,_,List,Str,_) :- Str =.. [sync_bin_ct|List].
reset(P,B,[Bit|Binary_list],Tmp,Str,Set_term,Reset_term) :-
    Next is (P - 1),
    form_and(P,Anded_term),
    do_eqtn_j(Set_term,Anded_term,Eqtnj),
    do_eqtn_k(Reset_term,Anded_term,Eqtnk),
    reset(Next,B,Binary_list,[jk_ff(Eqtnj,Eqtnk)|Tmp],Str,Set_term,Reset_term).

```

```

/*
RULE:

```

Counting sequence is a Toggling operation when the FF's to the right are all 1. With a J-K flip-flop, to toggle just have J=K=1. At the end of the counting sequence the FF must reset back to zero, no matter what the toggle would produce. To do this you 'OR' the toggle term with a term that is true (at highest state only) for the set input J, and 'AND' the toggle term with a term that is false only when you want to reset. All this ensures that the FF will reset when you want it to. The resulting logic equations are not necessarily 'minimized' so there could be redundant terms.

STEPS:

- 1 form anding eqtn
- 2 is full modulo ? yes stop
- 3 form false term (using clause 'process_false').
- 4 form true term (using clause 'process_true').
- 5 OR these terms into list as eqtns for this ff.

the following two clauses build up the J and K inputs to ensure that they both count and reset at the end of the sequence.

```
-----*/
do_eqtn_j(Set_term,Anded_term,and(Anded_term,Set_term)).
do_eqtn_k(Reset_term,Anded_term,or(Anded_term,Reset_term)).
```

```
process_false(0,[],'').
process_false(1,[1],not('a')).
process_false(1,[0],('a')).
process_false(P,[Bit|Binary_list],and(B1,B2)) :-
    Next is (P - 1),alpha_bit_false(Bit,P,B1),
    process_false(Next,Binary_list,B2).
```

```
process_true(0,[],'').
process_true(1,[1],('a')).
process_true(1,[0],not('a')).
process_true(P,[Bit|Binary_list],and(B1,B2)) :-
    Next is (P - 1),alpha_bit_true(Bit,P,B1),
    process_true(Next,Binary_list,B2).
```

```
/*-----
these following clauses set the logic term to be either
true or not true based on whether the highest state has a 1 or a
zero at that position.
```

```
-----*/
alpha_bit_false(1,P,not(B)) :- alpha(P,B).
alpha_bit_false(0,P,B) :- alpha(P,B).
alpha_bit_true(1,P,B) :- alpha(P,B).
alpha_bit_true(0,P,not(B)) :- alpha(P,B).
```

```
/*
```

SYNCHRONOUS GREYCODE COUNTER

eqtn_grey calculates the logic equations for a reflected grey code counter
A reflected grey code counter starts at state with all FF's at 0.

A 3 FF counter counts as :

```
000
001
011
010
110
111
101
100
```

```
*/
```

```
counter('synchronous greyscale serial',sync_grey(List)) :- ff(N),
    Next is (N - 1),
```

```

    grey(Next,(jk_ff(1,1,clk)),List).
grey(0,L,L).
grey(N,Tmp,List) :- Next is (N - 1),
    grey(Next,(jk_ff(1,1,clk),Tmp),List).
/*
logic_shift calculates the logic equation for a synchronous shift register counter.
This type of counter has an initial state of all FF's = 1. At each clock pulse
the data or bits shift towards 'a' or the least significant bit. This has the effect
of counting down. Normally the counter is set at its initial state of all ones.
The highest FF is held with a zero on the J input to set it to zero for
each clock pulse. The sequence is none resetting.
*/
counter('synchronous shift parallel',sync_shift(List)) :-
    ff(N),
    Next is (N - 1),
    ring(Next,(jk_ff('1','0',clk)),List).
/*
SYNCHRONOUS RING COUNTER
logic_ring calculates the logic equations for a counter
where at each count, the state of the flip-flop to the left
is assumed. In a 4 FF counter FF D would always assume the
state of FF A. The count sequence must start from a non-zero
starting state.
*/
counter('synchronous ring',sync_ring(List)) :- ff(N),
    Next is (N - 1),
    ring(Next,(jk_ff('a',not('a'),clk)),List).
ring(0,L,L).
ring(N,Tmp,List) :- Next is (N - 1),Last is (N + 1),alpha(Last,Letter),
    ring(Next,(jk_ff(Letter,not(Letter),clk),Tmp),List).
/*
SYNCHRONOUS SWITCHTAIL COUNTER
eqtn_switch calculates the equations for a switchtail counter
where FF D assumes the inverse of FF A A switchtail is different
from a shift counter, in that the swithtail counter gets its next
input from the least sig. FF output.
*/
counter('synchronous switchtail',sync_switch(List)) :- ff(N),
    Next is (N - 1),
    ring(Next,(jk_ff(not('a'),'a',clk)),List).
/*
SYNCHRONOUS MOEBIUS SEQUENCE COUNTER
eqtn_moebius calculates the logic equations for a
Moebius or Johnson sequence counter. This sequence starts
at all FF's at zero. The first clock puts 1 in the left most
FF, the next shifts that right, and moves another in its place.
When all the FF's are 1, the next clock introduces a 0 and the
action repeats itself.
*/
counter('synchronous moebius', sync_moebius(List)) :- ff(N),
    Next is (N - 1),
    alpha(N,Top_ff),
    alpha(Next,Letter),moebius(Top_ff,Next,(jk_ff(Letter,not(Letter),clk)),List).

moebius(Top_ff,1,L,(jk_ff(not(Top_ff),Top_ff,clk),L)).
moebius(Top_ff,N,Tmp,List) :- Next is (N - 1), alpha(Next,Letter),
    moebius(Top_ff,Next,(jk_ff(Letter,not(Letter),clk),Tmp),List).

```

```

/*
    MAX FAN IN
    File Name      fan
    File Created   February 1987
    Last Revision  April 14 1987

    This file contains all clauses necessary to determine
    the maximum fan in of a circuit. The maximum fan in
    is displayed at the terminal. The maximum fan is
    determined by comparing all the fan values and swaping
    a maximum value if required.
*/
fan_levels(S,F) :- var(S),display('Fatal Error: Circuit not instantiated'),abort.
/*      ^ The above clause protects against illegal call
*/
fan_levels(S,F) :- functor(S,_,N),asserta(fi(N)),fan_in(S,1,N),fi(F).
fan_in(S,_,0).
/*      ^ This terminates the fan search at a signal
*/
fan_in(S,1,1) :- arg(1,S,A),functor(A,_,N),fan_comp(N),fan_in(A,1,N).
/*      ^ The above clause allows for sub-circuits with a not or a user
        defined one parameter circuit.
*/
fan_in(S,1,N) :- arg(1,S,A),functor(A,_,M),fan_comp(M),fan_in(A,1,M),fan_in(S,2,N).
fan_in(S,N,M) :- arg(N,S,A),functor(A,_,Na),fan_comp(Na),fan_in(A,1,Na),Next is N+1,
        fan_in(S,Next,M).

fan_comp(N) :- fi(Old),Old<N,retract(fi(Old)),asserta(fi(N)).
fan_comp(N).

```

```

/*
    Prolog Circuit Design

    HELP LIBRARY
        - Counter Terminology
        - PCD commands
        - PCD description

    File Created    Jan 1987
    Last Modified   March 15 1987
*/
help(counter) :- system("more help/counter.intro").
help(switchtail) :- system("more help/switchtail").
help(ring) :- system("more help/ring").
help(ripple) :- system("more help/ripple").
help(countby) :- system("more help/countby").
/*
    Prolog Circuit Design
    COMMANDS
*/
help(check) :- system("more help/check").
help(check(design)) :- system("more help/check").
help(clear) :- system("more help/clear").
help(commands) :- system("more help/commands").
help(convert) :- system("more help/convert").
help(convert(design)) :- system("more help/convert").
help(counter) :- system("more help/counter").
help(designs) :- system("more help/designs").
help(fan) :- system("more help/fan").
help(fan(design)) :- system("more help/fan").
help(get) :- system("more help/get").
help(get(design)) :- system("more help/get").
help(min) :- system("more help/min").
help(min(design)) :- system("more help/min").
help(save) :- system("more help/save").
help(save(design)) :- system("more help/save").
help(store) :- system("more help/store").
help(store(circuit)) :- system("more help/store").
help(print) :- system("more help/print").
help(print(design)) :- system("more help/print").
/*
    PROLOG CIRCUIT DESIGN
    DESCRIPTION
*/
help(specs) :- system("more help/specs").
help(pcd) :- system("more help/pcd.intro").
help(Item) :- display('HELP error: '),display(Item),display(' ? '),nl,
               display('Valid options for help are '),nl,nl,showHelp.

showHelp :- clause(help(Item),Y),nonvar(Item),display(Item),nl,fail.
showHelp.

```

```

/*
EQUATION MINIMIZATION
Created Jan 1987
Last Revision March 6 1987
Boolean logic simplification rules:
AND  A.A  =  A
      1.A  =  A
      0.A  =  0
      A.1  =  A
      A.0  =  0
      A.~A =  0
OR   A+A  =  A
      1+A  =  1
      0+A  =  A
      A+1  =  1
      A+0  =  A
      A+~A =  1
NOT  ~A    =  A

*/
min_str(jk_ff(A,B),jk_ff(C,D)) :- min_str(A,C),min_str(B,D).
min_str(and(1,A),B) :- min_str(A,B).
min_str(and(A,1),B) :- min_str(A,B).
min_str(and(0,A),0).
min_str(and(A,0),0).
min_str(and(A,A),B) :- min_str(A,B).
min_str(and(A,not(A)),0).
min_str(and(not(A),A),0).
min_str(and(A,and(B,C)),D) :- mult_and([A],and(B,C),D).
min_str(and(A,B),Z) :- min_str(A,C),min_str(B,D),min1(and(C,D),Z).

min_str(or(1,A),1).
min_str(or(A,1),1).
min_str(or(A,A),B) :- min_str(A,B).
min_str(or(A,not(A)),1).
min_str(or(A,or(B,C)),D) :- mult_or([A],or(B,C),D).
min_str(or(A,B),Z) :- prod([A,B],Z).

min_str(not(not(A)),B) :- min_str(A,B).
min_str(not(A),not(B)) :- min_str(A,B).

min_str(A,A) :- integer(A).
min_str(A,A) :- atom(A).
min_str(A,A) :- var(A),display('PCD min error: structure not instantiated'),
nl,abort.

min_str(S,Smin) :- S =.. [Class|List],
min_list_str(List,[],MinList),
Smin =.. [Class|MinList].

min_list_str([],List,MinList) :- rev(List,[],MinList).
min_list_str([First|Rest],Str,MinList) :-
min_str(First,MinStr),

```

```

min_list_str(Rest,[MinStr|Str],MinList).

rev([],A,A).
rev([H|T],Tmp,A) :- rev(T,[H|Tmp],A).

min1(and(1,1),1).
min1(and(1,0),0).
min1(and(0,1),0).
min1(and(0,0),0).
min1(and(A,0),0).
min1(and(0,A),0).
min1(and(A,1),A).
min1(and(1,A),A).

min1(or(1,1),1).
min1(or(1,0),1).
min1(or(0,1),1).
min1(or(0,0),0).
min1(or(0,A),A).
min1(or(A,0),A).
min1(or(1,A),1).
min1(or(A,1),1).

min1(not(not(A)),A).
min1(A,A).

mult_and([List],and(A,and(B,C)),Z) :- mult_and([A|List],and(B,C),Z).
mult_and(List,and(A,B),Z) :- equal([A|List],Tmp),equal([B|Tmp],Sum),
    remov_dupl(Sum,Nodupl),
    remove_zeros(Nodupl,NoZeros),
    process_sum(NoZeros,Z).

process_sum([],0).
process_sum(Sums,0) :- any_zeros(Sums).
process_sum(Sums,Z) :- factor_sums(Sums,Z).

mult_or([List],or(A,or(B,C)),Z) :- mult_or([A|List],or(B,C),Z).
mult_or([List],or(A,B),Z) :- equal([A|List],Tmp),equal([B|Tmp],Prod),
    remov_dulp(Prod,Nodupl),
    remov_ones(Nodulp,NoOnes),
    process_prod(NoOnes,Z).

process_prod([],1).
process_prod(Prod,1) :- any_ones(Prod).
process_prod(Prod,Z) :- factor_prod(Prod,Z).

```

```

/*
    Prolog Counter Design
    File Name      pcd
    File Created   Jan 1986
    Last Revision  May 15 1987

    This file contains all Prolog clauses required for the user
    interface, utilities, and file management.

*/
version('May 15 1987').
start :- seen,nofileerrors,pcd.
pcd :- prompt,read(X),interpret(X),!,pcd.
pcd :- pcd.
prompt :- nl,nl,display(' ==> ').

interpret(check(X)) :- design(X,Str),!,check(Str).
interpret(check(X)).
check(Str) :- check_read_in,!,go_check(Str,E),display(E),nl.
check(Str) :- [check],asserta(check_read_in),!,go_check(Str,E),display(E),nl.
check(Str) :- display('Error: Unable to locate file "check"'),nl.
interpret(clear) :- retract_specs.

retract_specs :- clock(_),retract(clock(_)),fail.
retract_specs :- count(_),retract(count(_)),fail.
retract_specs :- delay(_),retract(delay(_)),fail.
retract_specs :- grey,retract(grey),fail.
retract_specs :- moebius,retract(moebius),fail.
retract_specs :- modulo,retract(modulo),fail.
retract_specs :- ring,retract(ring),fail.
retract_specs :- switchtail,retract(switchtail),fail.
retract_specs :- display('Counter specs cleared'),nl.

interpret(commands) :- system("more help/commands").
interpret(counter) :- counter_read_in,!,counter.
interpret(counter) :- [counter],asserta(counter_read_in),!,counter.
interpret(counter) :- display('Error: unable to find file "counter"').
interpret(convert(Name)) :- design(Name,Str),!,convert(Str).
interpret(convert(Name)).
convert(Str) :- convert_read_in,!,go_convert(Str,NandStr),store(NandStr).
convert(Str) :- [convert],asserta(convert_read_in),!,go_convert(Str,NandStr),
    store(NandStr).
convert(Str) :- display('Error unable to locate file "convert"').
interpret(designs) :- display('Designs currently in memory'),nl,
    design(S,Str),display(S),nl,fail.
interpret(designs) :- nl,display('Designs saved to file;'),nl,
    system("ls designs").
interpret(end_of_file) :- halt.
interpret(exit) :- halt.
interpret(fan(X)) :- design(X,S),fan(S).
interpret(fan(X)).
fan(S) :- read_in_fan,!,fan_levels(S,N),display('Max fan in is '),display(N),nl.
fan(S) :- [fan],asserta(fan_read_in),fan_levels(S,N),display('Max fan in is '),

```



```

display(N),nl.
fan(_) :- display('Error: Unable to find file "fan"'),nl.
interpret(get(Name)) :- name(Name,List),name('designs/',Dir),
    append(Dir,List,FileList),name(File,FileList),
    see(File),read(S),asserta(S),seen,
    display('Circuit structure retrieved').
interpret(get(Name)) :- display('Error: Unable to get design').
interpret(help) :- interpret(help(pcd)).
interpret(help(X)) :- read_in_help,help(X).
interpret(help(X)) :- [help_lib],asserta(read_in_help),help(X).
interpret(help(X)) :- display('Error: cant get help').
interpret(min(Name)) :- design(Name,S),min(S).
interpret(min(Name)).
min(S) :- min_read_in,!,get_min(S).
min(S) :- [min],asserta(min_read_in),get_min(S).
min(S) :- display('Error locating file "min"'),nl.
get_min(S) :- min_str(S,MinS),print_str(MinS),store(MinS).
interpret(print(Name)) :- design(Name,Str),print_str(Str).
interpret(print(Name)).
print_str(S) :- print_read_in,pr_str(S).
print_str(S) :- [print],asserta(print_read_in),pr_str(S).
print_str(S) :- display('Error locating file "print"').
interpret(save(Name)) :- design(Name,Str),
    name(Name,List),name('designs/',Dir),append(Dir,List,FileList),
    name(File,FileList),display('File name '),display(File),nl,
    tell(File),write(design(Name,Str)),write(' '),told,tell(user).

interpret(save(Name)).
interpret(shell) :- system("csh").
interpret(shell(Command)) :- display('Unable to execute command').
interpret(simulate(Name)) :- design(Name,Str),!,sim(Str).
interpret(simulate(Name)).
sim(Str) :- sim_read_in,!,simulate(Str).
sim(Str) :- [simulate],asserta(sim_read_in),!,simulate(Str).
sim(Str) :- display('Error locating file "simulate"').
intrepret(store(Str)) :- store(Str).
interpret(store(Str)) :- display('Error: unable to store circuit definition'),
    nl.
store(Circuit) :- nl,display('Do you wish to store this NEW design y/n > '),
    read(y),nl,display('Enter design name (atom) '),read(Name),nl,
    atom(Name),asserta(design(Name,Circuit)),nl.
store(Circuit) :- nl,display('Warning: circuit not stored'),nl.

interpret(X) :- call(X).
interpret(Nonsense) :- display('Command Error'),nl.
/*
    UTILITY CLAUSES
*/
alpha(0,1).
alpha(1,'a').
alpha(2,'b').
alpha(3,'c').
alpha(4,'d').
alpha(5,'e').
alpha(6,'f').
alpha(7,'g').
alpha(8,'h').

```

```

alpha(9,'i').
alpha(10,'j').

append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).

equal(T,T).

inverse(1,0).
inverse(0,1).
inverse(X,Y) :- display('Fatal Error: incorrect inverse call with '),
                 display(X),nl,abort.

rev([],A,A).
rev([H|T],Tmp,A) :- rev(T,[H|Tmp],A).
/*
                                A U T O   S T A R T U P
*/
design(Name,Str) :- nonvar(Name),var(Str),
                  display('Error: Unable to find design ""'),display(Name),
                  display(''),nl,!,fail.
:- nl,nl,
   display('P r o l o g   C o u n t e r   D e s i g n'),nl,
   display('(type "commands." for listing of available commands)'),nl,
   display('Version: '),version(D),display(D),start.

```

```

/*
    DISPLAYING THE CIRCUIT
    File Name      print
    File Created   Jan 1987
    File Modified  April 14 1987

    This file contains all the clauses required to display a
    circuit at the terminal. The logic operator AND is repre-
    sented by a ".", the OR operator by "+" and the NOT oper-
    ator by a "~". Other logic operators and circuit names are
    represented by the data structure functor.

*/
pr_str(A) :- var(A),display('Fatal Error: Structure not instantiated'),abort.
pr_str(S) :- (atom(S);integer(S)),display(S).
pr_str(S) :- functor(S,Name,N),pr_str(Name,S,1,N).
pr_str(_) :- display('Print Error: Unable to print circuit'),nl.

pr_str(jk_ff,_,_,1) :- display('PrintError: JK flip-flop definition').
pr_str(jk_ff,S,1,3) :- nl,display('J-K flipflop with J input; '),nl,
    arg(1,S,J),pr_str(J),nl,display('K input; '),nl,
    arg(2,S,K),pr_str(K),nl,display('Clock input; '),
    arg(3,S,Ck),pr_str(Ck),nl.
pr_str(jk_ff,S,1,2) :- nl,display('J-K flipflop with J input; '),nl,
    arg(1,S,J),pr_str(J),nl,display('K input; '),nl,
    arg(2,S,K),pr_str(K),nl.
pr_str(jk_ff,_,_,_) :- display('Print Error: unable to print jk flipflop'),nl.

pr_str(not,_,_,N) :- N > 1,display('Print Error: Not circuit definition').
pr_str(not,S,1,1) :- display('~('),arg(1,S,A),pr_str(A),display(')').
pr_str(not,_,_,_) :- display('Print Error: unable to print not circuit'),nl.

pr_str(and,_,_,1) :- display('Print Error: And circuit definition').
pr_str(and,S,1,M) :- display('('),arg(1,S,A),pr_str(A),display(' '),
    pr_str(and,S,2,M).
pr_str(and,S,M,M) :- arg(M,S,A),pr_str(A),display(')').
pr_str(and,S,N,M) :- arg(N,S,A),pr_str(A),display(' '),Next is (N + 1),
    pr_str(and,S,Next,M).

pr_str(or,_,_,1) :- display('Print Error: Or Circuit definition').
pr_str(or,S,1,M) :- display('('),arg(1,S,A),pr_str(A),display(' + '),
    pr_str(or,S,2,M).
pr_str(or,S,M,M) :- arg(M,S,A),pr_str(A),display(')').
pr_str(or,S,N,M) :- arg(N,S,A),pr_str(A),display(' + '),
    Next is (N + 1),pr_str(or,S,Next,M).

pr_str(nand,S,N,N) :- arg(N,S,A),pr_str(A),display(')').
pr_str(nand,S,1,M) :- arg(1,S,A),display('nand('),pr_str(A),
    pr_str(nand,S,2,M).
pr_str(nand,S,P,M) :- arg(P,S,A),pr_str(A),Next is P+1,display(' '),
    pr_str(nand,S,Next,M).

pr_str(Circuit,S,1,1) :- arg(1,S,A),display(Circuit),
    display(' with input No 1;'),nl,display(' '),pr_str(A).

```

```

pr_str(Circuit,S,1,M) :- arg(1,S,A),display(Circuit),
    display('input 1; '),
    pr_str(A),pr_str(Circuit,S,2,M).
pr_str(Circuit,S,M,M) :- arg(M,S,A),nl,display(Circuit),
    display('Input '),display(M),display('; '),pr_str(A).
pr_str(Circuit,S,N,M) :- arg(N,S,A),nl,display(Circuit),
    display('nput '),display(N),display(' ; '),
    pr_str(A),Next is N + 1,pr_str(Circuit,S,Next,M).
pr_str(Name,_,_,_) :- display('Print Error : Unable to print circuit '),
    display(Name),nl,!.

```

```

/*
    CIRCUIT SIMULATION
    File Name      simulate
    File Created   February 21 1987
    Last Revision  May 18 1987

    This file contains the clauses required to perform
    gate level functional simulation. The following
    conventions are applied for simulation.

        1. A circuit with multiple outputs are labeled
           "a" for the first output, "b" for the second
           etc. Thus a circuit which has input "a" is
           actually connected to its output.

        2. Values for all signals (inputs to circuits)
           are determined before the circuit is simulated.
           These signals values are stored and can be revised
           as required.
*/
simulate(S) :- var(S),
    display('Error: Circuit not instantiated for simulation'),nl.
simulate(S) :- clear_values,eval_signals(S),functor(S,Name,N),assert(sim_circuit(Name)),
    assertz(value(X,0)),!,asserta(current_bit(1)),!,
    eval_str(S,1,N,Name,[],Result),!,simulate(S,N,Name,Result).
/*
    The second definition of "simulate" is the normal calling sequence
    for a simulation. First the circuit's signals are evaluated,
    and circuit signals are assigned using "value(Signal,value)" rules.
    Then the circuit is simulated using the "eval_str" call.
*/

simulate(S,N,Name,Next_st) :- update_var(Next_st),show_st(Next_st),!,
    go_on,retract(current_bit(X)),asserta(current_bit(1)),!,
    eval_str(S,1,N,Name,[],Next_next_st),
    simulate(S,N,Name,Next_next_st).
simulate(S,N,Name,Result) :- display('Simulation concluded. '),nl.
clear_values :- retract(value(X,Y)),fail.
clear_values.
/*
    ^ "clear_values" is used to clear all variable value assignments
    before a logic simulation is started. This avoids any errors
    arising from previous simulations that have been run.
*/

update_var(State_list) :- rev(State_list,[],Rev_state),update_var(1,Rev_state).
update_var(N,[First|Rest]) :- alpha(N,Letter),retract(value(Letter,X)),
    asserta(value(Letter,First)),Next is (N+1),update_var(Next,Rest).
update_var(N,[First|Rest]) :- alpha(N,Letter),asserta(value(Letter,First)),
    Next is (N+1),update_var(Next,Rest).
update_var(N,One_bit).
/*
    ^ These "update_var" clauses are used to update the "value(signal,signal_value)"
    clauses, which maintain the value of variables used in the simulation.
    The first, one parameter definition, reverses the list making the
    least significant bit "a", occur at the top of the state list.
    Note the second and third definitions differ in that a "retract"

```

```

call may not succeed in the case where the variable had not been
    previously assigned.
*/
show_st(State_list) :- var(State_list),display('Fatal Error: current state not defined'),abort.
show_st(State_list) :- display('Circuit state is'),nl,display(' '),
    state_lenght(0,N,State_list),display_state(N),nl,display(State_list),nl.
state_lenght(Ct,N,[First|Rest]) :- Next is (Ct+1),state_lenght(Next,N,Rest).
state_lenght(Ct,N,Rest) :- equal(N,Ct).

display_state(1) :- display('a]').
display_state(N) :- alpha(N,Letter),display(Letter),display(' '),Next is (N-1),
    display_state(Next).
/*      ^ The above clauses are used to display the current state of the
simulation, with the state variables names displayed above the list.
*/

go_on :- display('Do you wish to continue the simulation ? <y/n>.'),!,
    read('y').
go_on :- pcd.

eval_signals(S) :- (atom(S);integer(S)),what_value(S).
eval_signals(S) :- functor(S,_,N),eval_signals(S,1,N).
eval_signals(S,M,M) :- arg(M,S,A),eval_signals(A).
eval_signals(S,N,M) :- arg(N,S,A),eval_signals(A),Next is N+1,eval_signals(S,Next,M).
what_value(S) :- signal(S,Str),functor(Str,Name,N),eval_str(S,1,N,Name,[],Result).
what_value(S) :- signal(S,Str),display('Error in evaluating input signal'),nl,
    abort.
what_value(1) :- nl,display('Warning: Circuit has fixed "1" input value').
what_value(0) :- nl,display('Warning: Circuit has fixed "0" input value').
what_value(Atom) :- value(Atom,_).
what_value(Atom) :- nl,display('What value should '),display(Atom),
    display(' have ? '),read(X),asserta(value(Atom,X)).

eval_jk(1,1,1,0).
eval_jk(1,1,0,1).
eval_jk(1,0,_,1).
eval_jk(0,1,_,0).
eval_jk(0,0,P,P).
/*      V "eval_str" is the main evaluation clause set. Each clause has
        6 inputs which have the following meaning;

        1 = the circuit or signal.
        2 = the position within the circuit.
        3 = the number of parameters of the circuit.
        4 = the circuit name (functor of the structure).
        5 = intermediate result if list output
        6 = final output (can be list).
*/
eval_str(1,1,0,_,[],1).
eval_str(0,1,0,_,[],0).
eval_str(A,1,0,_,[],Result) :- value(A,Result).
eval_str(S,1,N,and,[],Result) :- arg(1,S,A),functor(A,NameA,Na),
    eval_str(A,1,Na,NameA,[],Rt),
    eval_str(S,2,N,and,[Rt],Result).
eval_str(S,N,N,and,List,Result) :- arg(N,S,A),functor(A,NameA,Na),
    eval_str(A,1,Na,NameA,[],Res),eval_and([Res|List],Result).
eval_str(S,P,N,and,C,Result) :- arg(P,S,A),functor(A,NameA,Na),

```

```

eval_str(A,1,Na,NameA,[],Rt),Next is P + 1,
    eval_str(S,Next,N,and,[Rt|C],Result).

eval_str(S,1,N,or,[],Result) :- arg(1,S,A),functor(A,NameA,Na),
    eval_str(A,1,Na,NameA,[],Rt),
    eval_str(S,2,N,or,Rt,Result).
eval_str(S,N,N,or,List,Result) :- arg(1,S,A),functor(A,NameA,Na),
    eval_str(A,1,Na,NameA,[],Rt),
    eval_or([Rt|List],Result).
eval_str(S,P,N,or,C,Result) :- arg(P,S,A),functor(A,NameA,Na),
    eval_str(A,1,Na,NameA,[],Rt),Next is P + 1,
    eval_str(S,Next,N,or,[Rt|C],Result).

eval_str(S,1,N,nand,[],Result) :- arg(1,S,A),functor(A,NameA,Na),
    eval_str(A,1,Na,NameA,[],Rt),
    eval_str(S,2,N,nand,Rt,Result).
eval_str(S,N,N,nand,List,Result) :- eval_nand(List,Result).
eval_str(S,P,N,nand,C,Result) :- arg(P,S,A),functor(A,NameA,Na),
    eval_str(A,1,Na,NameA,[],Rt),Next is P + 1,
    eval_str(S,Next,N,nand,[Rt|C],Result).

eval_str(S,1,N,not,[],Result) :- arg(1,S,A),functor(A,NameA,Na),
    eval_str(A,1,N,NameA,[],Res),equal(inverse(Res),Result).

eval_str(S,_,N,jk_ff,_,Result) :- arg(1,S,J),arg(2,S,K),
    functor(J,Namej,Nj),functor(K,Namek,Nk),
    eval_str(J,1,Nj,Namej,[],Rj),
    eval_str(K,1,Nk,Namek,[],Rk),current_bit(Any),alpha(Any,C),value(C,Cb),
    eval_jk(Rj,Rk,Cb,Result).

/*      The following clauses are used to evaluate user defined
circuits.
*/
eval_str(S,1,1,Name,[],Result) :- arg(1,S,A),functor(A,NameA,Na),
    eval_str(A,1,Na,NameA,[],Result).
eval_str(S,N,N,Name,C,Result) :- arg(N,S,A),functor(A,NameA,Na),
    eval_str(A,1,Na,NameA,[],ResA),equal(Result,[ResA|C]).
eval_str(S,P,N,Name,C,Result) :- arg(P,S,A),functor(A,NameA,Na),
    eval_str(A,1,Na,NameA,[],ResA),Next is P+1,pos(Name,P,N),
    eval_str(S,Next,N,Name,[ResA|C],Result).
eval_str(A,B,C,D,E,F) :- display('Fatal Simulation Error: Cannot evaluate circuit'),
    display(A),nl,display(B),nl,display(C),nl,display(D),nl,
    display(E),nl,display(F),nl,abort.

pos(Circuit,P,N) :- sim_circuit(Circuit),P < N,retract(current_bit(P)),
    Next is P + 1,asserta(current_bit(Next)).
pos(Circuit,P,N).
/*      ^ These two clauses maintain the "current_bit" clause. This is
required for simulating counter circuits where previous values
are required.
*/

eval_and([H|Rest],Result) :- equal(H,1),eval_and(Rest,Result).
eval_and([H|Rest],0) :- equal(H,0).
eval_and(0,0).
eval_and(1,1).

```

```

eval_and([],1).
eval_and([1],1).
eval_and([0],0).
/*      ^ These four clauses are used to return the Boolean operation
        AND which is performed on a list of 1's and 0's. The list
        is the first parameter and the result is the second parameter.
*/

eval_or([H|Rest],Result) :- equal(H,0),eval_or([Rest],Result).
eval_or([H|Rest],1) :- equal(H,1).
eval_or([0],0).
eval_or([1],1).
/*      ^ These four clauses are used to return the Boolean operation
        OR which is performed on a list of 1's and 0's.
        If a 1 is found then the result must be 1, and no further
        searching is required. If 0's only are found then keep searching
        and return 0 if ALL were zeros.
*/

```


Appendix B - Prolog Simulation Trace

The following is a trace of a synchronous modulus binary counter which is discussed in section 7.7.1. The simulation runs through one clock pulse.

```

Call: simulate(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk))) ?
Call: var(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk))) ?
Fail: var(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk))) ?
Back to: simulate(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk))) ?
Call: clear_values ? skip
>Exit: clear_values
>Call: eval_signals(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk))) ?
Call: atom(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk))) ?
Fail: atom(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk))) ?
Call: integer(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk))) ?
Fail: integer(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk))) ?
Back to: eval_signals(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk))) ?
Call: functor(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),_33371,_33372) ?
Exit: functor(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),sync_bin_mod,4)
Call: eval_signals(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),1,4) ?
Call: arg(1,sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),_33391) ?
Exit: arg(1,sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),jk_ff(1,1,clk))
Call: eval_signals(jk_ff(1,1,clk)) ? skip

Warning: Circuit has fixed "1" input value
Warning: Circuit has fixed "1" input value
What value should clk have ? 0.
Exit: eval_signals(jk_ff(1,1,clk))
Call: _33392 is 1+1 ?
Exit: 2 is 1+1
Call: eval_signals(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),2,4) ?
Call: arg(2,sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),_33692) ?
Exit: arg(2,sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),jk_ff(a,a,clk))
Call: eval_signals(jk_ff(a,a,clk)) ? skip
What value should a have ? 0.
Exit: eval_signals(jk_ff(a,a,clk))

```

Call: _33693 is 2+1 ?
 Exit: 3 is 2+1
 Call: eval_signals(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
 jk_ff(and(a,b,c),and(a,b,c),clk)),3,4) ?
 Call: arg(3,sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
 jk_ff(and(a,b,c),and(a,b,c),clk)),_33981) ?
 Exit: arg(3,sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
 jk_ff(and(a,b,c),and(a,b,c),clk)),jk_ff(and(a,b),and(a,b),clk))
 Call: eval_signals(jk_ff(and(a,b),and(a,b),clk)) ? skip

What value should b have ? 0.

Exit: eval_signals(jk_ff(and(a,b),and(a,b),clk))
 Call: _33982 is 3+1 ?
 Exit: 4 is 3+1
 Call: eval_signals(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
 jk_ff(and(a,b,c),and(a,b,c),clk)),4,4) ?
 Call: arg(4,sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
 jk_ff(and(a,b,c),and(a,b,c),clk)),_34488) ?
 Exit: arg(4,sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
 jk_ff(and(a,b,c),and(a,b,c),clk)),jk_ff(and(a,b,c),and(a,b,c),clk))
 Call: eval_signals(jk_ff(and(a,b,c),and(a,b,c),clk)) ? skip

What value should c have ? 0.

Exit: eval_signals(jk_ff(and(a,b,c),and(a,b,c),clk))
 Exit: eval_signals(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
 jk_ff(and(a,b,c),and(a,b,c),clk)),4,4)
 Exit: eval_signals(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
 jk_ff(and(a,b,c),and(a,b,c),clk)),3,4)
 Exit: eval_signals(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
 jk_ff(and(a,b,c),and(a,b,c),clk)),2,4)
 Exit: eval_signals(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
 jk_ff(and(a,b,c),and(a,b,c),clk)),1,4)
 Exit: eval_signals(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
 jk_ff(and(a,b,c),and(a,b,c),clk)))
 Call: functor(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
 jk_ff(and(a,b,c),and(a,b,c),clk)),_241,_33352) ?
 Exit: functor(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
 jk_ff(and(a,b,c),and(a,b,c),clk)),sync_bin_mod,4)
 Call: assert(sim_circuit(sync_bin_mod)) ?
 Exit: assert(sim_circuit(sync_bin_mod))
 Call: assertz(value(_242,0)) ?
 Exit: assertz(value(_242,0))
 Call: asserta(current_bit(1)) ?
 Exit: asserta(current_bit(1))
 Call: eval_str(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
 jk_ff(and(a,b,c),and(a,b,c),clk)),1,4,sync_bin_mod,[],_33353) ?
 Call: arg(1,sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
 jk_ff(and(a,b,c),and(a,b,c),clk)),_33366) ?
 Exit: arg(1,sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
 jk_ff(and(a,b,c),and(a,b,c),clk)),jk_ff(1,1,clk))
 Call: functor(jk_ff(1,1,clk),_33367,_33368) ?
 Exit: functor(jk_ff(1,1,clk),jk_ff,3)
 Call: eval_str(jk_ff(1,1,clk),1,3,jk_ff,[],_424) ? skip
 Exit: eval_str(jk_ff(1,1,clk),1,3,jk_ff,[],1)
 Call: _33369 is 1+1 ?
 Exit: 2 is 1+1
 Call: pos(sync_bin_mod,1,4) ? skip

```

Exit: pos(sync_bin_mod,1,4)
Call: eval_str(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),2,4,sync_bin_mod,[1],_33353) ?
Call: arg(2,sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),_33605) ?
Exit: arg(2,sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),jk_ff(a,a,clk))
Call: functor(jk_ff(a,a,clk),_33606,_33607) ?
Exit: functor(jk_ff(a,a,clk),jk_ff,3)
Call: eval_str(jk_ff(a,a,clk),1,3,jk_ff,[],_447) ? skip
Exit: eval_str(jk_ff(a,a,clk),1,3,jk_ff,[],0)
Call: _33608 is 2+1 ?
Exit: 3 is 2+1
Call: pos(sync_bin_mod,2,4) ? skip
Exit: pos(sync_bin_mod,2,4)
Call: eval_str(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),3,4,sync_bin_mod,[0,1],_33353) ?
Call: arg(3,sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),_33864) ?
Exit: arg(3,sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),jk_ff(and(a,b),and(a,b),clk))
Call: functor(jk_ff(and(a,b),and(a,b),clk),_33865,_33866) ?
Exit: functor(jk_ff(and(a,b),and(a,b),clk),jk_ff,3)
Call: eval_str(jk_ff(and(a,b),and(a,b),clk),1,3,jk_ff,[],_472) ? skip
Exit: eval_str(jk_ff(and(a,b),and(a,b),clk),1,3,jk_ff,[],0)
Call: _33867 is 3+1 ?
Exit: 4 is 3+1
Call: pos(sync_bin_mod,3,4) ? skip
Exit: pos(sync_bin_mod,3,4)
Call: eval_str(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),4,4,sync_bin_mod,[0,0,1],_33353) ?
Call: arg(4,sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),_34310) ?
Exit: arg(4,sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),jk_ff(and(a,b,c),and(a,b,c),clk))
Call: functor(jk_ff(and(a,b,c),and(a,b,c),clk),_34311,_34312) ?
Exit: functor(jk_ff(and(a,b,c),and(a,b,c),clk),jk_ff,3)
Call: eval_str(jk_ff(and(a,b,c),and(a,b,c),clk),1,3,jk_ff,[],_514) ? skip
Exit: eval_str(jk_ff(and(a,b,c),and(a,b,c),clk),1,3,jk_ff,[],0)
Call: equal(_33353,[0,0,0,1]) ?
Exit: equal([0,0,0,1],[0,0,0,1])
Exit: eval_str(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),4,4,sync_bin_mod,[0,0,1],[0,0,0,1])
Exit: eval_str(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),3,4,sync_bin_mod,[0,1],[0,0,0,1])
Exit: eval_str(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),2,4,sync_bin_mod,[1],[0,0,0,1])
Exit: eval_str(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),1,4,sync_bin_mod,[],[0,0,0,1])
Call: simulate(sync_bin_mod(jk_ff(1,1,clk),jk_ff(a,a,clk),jk_ff(and(a,b),and(a,b),clk),
jk_ff(and(a,b,c),and(a,b,c),clk)),4,sync_bin_mod,[0,0,0,1]) ?
Call: update_var([0,0,0,1]) ? skip
Exit: update_var([0,0,0,1])
Call: show_st([0,0,0,1]) ? skip
Circuit state is
[d,c,b,a]
[0,0,0,1]

```

Exit: show_st([0,0,0,1])
Call: go_on ? skip
Do you wish to continue the simulation ? <y/n>.n.



National Library
of Canada

Bibliothèque nationale
du Canada

Ottawa, Canada
K1A 0N4

TC -

ISBN

CANADIAN THESES ON MICROFICHE SERVICE - SERVICE DES THÈSES CANADIENNES SUR MICROFICHE

PERMISION TO MICROFILM - AUTORISATION DE MICROFILMER

• Please print or type - Écrire en lettres moulées ou dactylographier

AUTHOR - AUTEUR

Full Name of Author - Nom complet de l'auteur

Ronan P O'BYRNE

Date of Birth - Date de naissance

JUNE 7 1957

Canadian Citizen - Citoyen canadien

☒ Yes / Oui

☐ No / Non

Country of Birth - Lieu de naissance

IRELAND

Permanent Address - Résidence fixe

213 DEER BAY SE
CALGARY, ALTA T2J 6N9

THESIS - THÈSE

Title of Thesis - Titre de la thèse

DIGITAL LOGIC
A Causal APPROACH TO CIRCUIT DESIGN.

Degree for which thesis was presented
Grade pour lequel cette thèse fut présentée

M. Sc.

Year this degree conferred
Année d'obtention de ce grade

1987

University - Université

UNIVERSITY OF CALGARY.

Name of Supervisor - Nom du directeur de thèse

DR J KENDALL

AUTHORIZATION - AUTORISATION

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to
microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive
extracts from it may be printed or otherwise reproduced without the
author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE
DU CANADA de microfilmer cette thèse et de prêter ou de vendre des ex-
emplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs ex-
traits de celle-ci ne doivent être imprimés ou autrement reproduits sans
l'autorisation écrite de l'auteur.

ATTACH FORM TO THESIS - VEUILLEZ JOINDRE CE FORMULAIRE À LA THÈSE

Signature

Ronan O'Byrne

Date

SEPT 8 1987

W 18-03-88

Reject
p. 19 missing