Copyright © 1984, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

RUBICC: A RULE-BASED EXPERT SYSTEM FOR VLSI INTEGRATED CIRCUIT CRITIQUE

by

C. Lob

Memorandum No. UCB/ERL M84/80

28 September 1984

(cover)

RUBICC: A RULE-BASED EXPERT SYSTEM FOR VLSI INTEGRATED CIRCUIT CRITIQUE

by

C. Lob

Memorandum No. UCB/ERL M84/80

28 September 1984

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Acknowledgements

I would like to thank Professor A.R. Newton for his encouragement and support during this project and for his help throughout the 1983-84 academic year. Discussions with Professors D.O. Pederson and A.L. Sangiovanni were highly beneficial in providing background ideas and energy to me throughout the year.

My financial support came from Hewlett-Packard Company's "Resident Fellowship Program." I would like to thank Dana Seccombe, Jack Anderson, Joe Beyers, and Norm Vlass for making this opportunity available to me and Lavonne Gardner and Linda Alvine for their help and support in the administration of this fellowship.

The Knowledge Base of the program came, in part, from a number of designers from the System Technology Operation of Hewlett-Packard in Ft. Collins, Colorado. Their contributions are greatly appreciated.

A number of people from the Application Technology Laboratory of Hewlett-Packard Laboratories provided assistance and support with the HPRL System. I would like to thank Jeff Eastman, Steven Rosenberg, Pierre Huyn, Mike Lemon, and Doug Lanam for their help.

The HP-9836 Systems, which supported the RUBICC development, were donated by The Design Automation Group at Hewlett-Packard in Cupertino. Thanks to Bill McCalla for his help in obtaining the equipment and to Martin Gates for his help with the system configuration.

Special thanks to Tom Quarles, Peter Moore, Mark Hoffman, Ron Gyurcsik, Karti Mayaram, Rick Spickelmier, Mike Klein and Grace Mah for making me feel welcome and teaching me "the system."

The figures in this report were prepared using an HP-9836C running Hewlett-Packard's Piglett Schematic and Layout Editor.

Finally, I'd like to thank JoAnn for her continuous support and *Love* during my "Master's Degree Year."

Abstract

RUBICC is an Expert System written to critique the design of VLSI circuits at the cell (transistor) level. The goal of this project is to explore the feasibility of using "Expert System Technology" to provide meaningful feedback to circuit designers on the quality of their designs in a manner similar to the critique provided by an experienced designer. The use of a Rule Base facilitates the encapsulation of the "Knowledge Base" associated with the design of circuits in a given technology and supports incremental additions to that knowledge base. RUBICC performs its critique without the use of simulation, by "reasoning" about the design using rules contained in its knowledge base. Early experiments with RUBICC indicate that the system performs a meaningful critique for a wide variety of circuit and transistor configurations and can find problems which are elusive to even worst case circuit simulation.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION 1

1.1 Problem Description 1

1.2 Brief System Overview 3

1.3 Personal Motivation 5

1.4 Report Organization 5

CHAPTER 2: EXPERT SYSTEMS CONSIDERATIONS 7

2.1 Expert System Characteristics 7

2.2 System Language Choice 7

CHAPTER 3: RELATED WORK / CRITIQUE ISSUES 9

3.1 Why Use Expert System Technology? 9

3.2 What Kind of Critique is Needed? 10

3.3 Analysis Approach 11

3.4 How And When to Use the Tool 12

3.5 Efficiency 13

CHAPTER 4: HPRL OVERVIEW 14

4.1 Frames 14

4.1.1 Generic Frames and Inheritance 14

4.1.2 Data Stored in Slots 18

4.1.3 Dynamic Storage Management 19

4.2 Pattern Matching 20

4.3 Backward and Forward Chaining Rules 23

4.3.1 Rule Overview 23

4.3.2 Backward Chain Rules 24

4.3.3 Forward Chain Rules 26

4.4 Demons 26

4.5 Escapes to Lisp 27

CHAPTER 5: RUBICC SYSTEM DESIGN 28

5.1 Program Structure and Control 28

5.2 Circuit Input Format 29

5.3 Frame Data Base 30

5.3.1 Schematic Frames 31

5.3.1.1 Transistor Hierarchy 31

5.3.1.2 Circuit Hierarchy 33

5.3.1.3 Struct Hierarchy 35

5.3.1.4 Nodes 37

5.3.1.5 One and Two Port Elements 37

5.3.2 Program Control Frames 38

5.3.2.1 Elements Frames 39

5.3.2.2 *G-Con 39

5.3.3 Error Frames 39

5.4 RUBICC Programming Paradigms 41

5.5 Program Output Format 45

5.6 Classification Strategies 46

5.6.1 When to Classify 47

5.6.2 How to Classify 47

5.7 Separation of Program Control and Knowledge Base 49

CHAPTER 6: RESULTS 50

CHAPTER 7: CONCLUSIONS AND FUTURE WORK 52

7.1 Conclusions 52

7.2 Future Work 52

7.2.1 User Interface 53

7.2.2 Checking Actual Circuits 53

7.2.3 CMOS Compatibility 53

7.2.4 Additional Structure Classifications 53

7.2.5 Program Tuning 54

CHAPTER 8: REFERENCES 55

APPENDIX A: FRAME HIERARCHIES 58

APPENDIX B: GENERIC FRAMES 59

APPENDIX C: TECHNOLOGY FRAME 65

APPENDIX D: RUBICC EXAMPLES 66

APPENDIX E: IMPLEMENTED CIRCUIT CHECKS 96

APPENDIX F: RUBICC SOURCE CODE 110

CHAPTER 1

Introduction

1.1. Problem Description and Goals

Many tools exist to aid in the design, simulation and layout of VLSI Integrated Circuits. A small subset of the tools at U.C. Berkeley include: Circuit and timing simulators such as Spice [1], Splice [2,3] and Relax [4], timing analyzers such as Crystal [5], behavioral level simulators such as the FTL system [6], hierarchical artwork systems such as Kic [7], Caesar [8] Hawk [9], and Magic [10], and module generators such as Panda [11]. A detailed explanation of many of these and other CAD Tools at U.C. Berkeley can be found in [12].

In addition, there is ongoing research and there exist operational systems which use Artificial Intelligence Technology in the form of Expert Systems to perform tasks which provide global synthesis functions to the designer. These systems include: The Design Automation Assistant [13] which starts from an algorithmic description and produces a VLSI design to the level of "technology independent registers, operators, data paths and control sequences"; The Talib System [14] which performs artwork layout; The Micon System [15] which designs single board computers from a series of hardware descriptions; and the SCHEMA System [16] which is intended to act in the capacity of a complete design assistant, not limited to design critique alone.

The thrust of all these tools is to aid the designer by reducing the time and effort required for analysis and synthesis during the design process.

Both RUBICC and the CRITTER System [17] represent research into another key area

of the circuit design process — that of design critique and review. At Hewlett Packard,[1] *Peer Group Review* is one of the major checkpoints of the VLSI design process. During this phase, a designer will have his work reviewed by other members of the project team. This review usually takes the form of one or more experienced people who study the schematics and layouts of a design, their mission being to "flush out bugs" that may have been overlooked by the original engineer and to provide feedback to him on his work.

The items turned up during this review cycle take many forms. In some cases they are very simple problems. Perhaps a new designer may not understand the implications of a design rule or needs some guidance as to the more practical ways of implementing a logic function. In other cases, extremely subtle problems are found which can elude even the most careful worst case simulations. Charge coupling, MOS capacitor inversion time constants, charge sharing on dynamic buses, voltage swings on bootstrapped nodes, and problems due to clock undershoot, overshoot, overlap and skew are some examples of the latter. Peer Group Review also provides a forum for indepth discussions which can trigger a designer's thought process to uncover errors in related designs.

Unfortunately, these reviews are time consuming and experienced people are hard to find within the organizational structure. Promotions, tight schedules and other job assignments make the "experts" a scarce commodity. With all of the other productivity tools being developed and with the great influx of new VLSI designs and designers, there are more new designs than there are experienced people to review them. Another problem is that the total knowledge base exists only collectively in the minds of numerous designers. Hence one expert's review may not catch something that another's might find.

The project described here, called RUBICC (Rule Based Integrated Circuit Critic), addresses these problems by performing a design critique of NMOS VLSI Cells. Specifically, RUBICC's goals are:

---

[1] author worked as a Project Manager at HP's Systems Technology Operation in Ft. Collins, Colorado and is currently a Section Manager at the Corvallis Components Operation of HP in Corvallis, Oregon.

1. To explore the feasibility of using "Expert System Technology" to aid in the design and feedback process of VLSI.

2. Determine the feasibility of encapsulating a "Knowledge Base" of Design Heuristics

3. Determine the productivity of using an expert system language for the implementation of the program.

As will be seen later on in this report, RUBICC achieves a high degree of success in addressing all of these goals.

The importance of the critique process should not be overlooked. Joe Beyers, R&D Lab Manager for the Cupertino Intergated Circuits Operation of Hewlett-Packard [18] told me that two of the major bottlenecks in the VLSI design process are designing new circuits at the cell level and training new VLSI designers. RUBICC could be used by both new and experienced designers to improve their productivity in these areas. It would also provide a uniform critique since its knowledge base would contain the collective knowledge of a large number of designers.

1.2. Brief System Overview

RUBICC is implemented as a Knowledge Based System using a language called HPRL (Heuristic Programming and Representation Language) [19] developed at the Computer Research Center of Hewlett-Packard Laboratories. It is modeled after Goldstein's and Roberts Frame Based FRL System [20]. HPRL is implemented on top of PSL (Portable Standard Lisp) [21] and currently runs on an HP's 9836 Desktop Computer. The HP-9836 is Motorola 68000 based. HPRL is licensed through HP-Labs.

RUBICC's data base is implemented as Hierarchical Frames in HPRL. Frames are similar to record structures in the "C" and "Pascal" programming languages with two major differences: More than one type of data can be stored in the slot of a frame as compared to a field of a record and frames can be arranged hierarchically. These two differences give the program very powerful search and pattern matching capabilities

which will be explained later. The actual knowledge base of RUBICC is implemented using HPRL rules. These are similar to productions in other languages such as OPS-5 [22]. HPRL allows both Forward and Backward Chaining Rules as well as demons and escapes to the underlying Lisp Language.[2]

To run RUBICC on a circuit, the designer provides a "spice-like" input in Lisp list notation. The program first reads in the circuit description. It then applies its knowledge base to the circuit. When finished, RUBICC prints out a summary of its critique. Example inputs and outputs are included in Appendix D. A complete description of RUBICC's current error checks is included in Appendix E. Process-dependent parameters which are needed by the rule system are stored as constants in a *Technology Frame*. In this way, RUBICC can be changed to handle various technologies of the same type, such as two NMOS processes having different transistor characteristics. For distinctly different processes technologies (i.e. CMOS vs NMOS) the rule system would also be different.

The present system contains 110 rules which are used to pick out various structures and to identify errors in these structures. Approximately 60 *generic* frames are required to hierarchically arrange the circuit data base. A specific frame is instantiated[3] for each circuit element. These elements are: *driver* (enhancement transistor), *load* (depletion transistor), *capacitor*, *power-supply* and *clock*. In addition, a specific frame is instantiated for each unique circuit node. The system is approximately 110 KBytes of source code.

HPRL seemed a natural choice for the implementation of RUBICC. It is a state-of-the-art system for the development of Expert Systems and the solving of general "reasoning type problems." The features of Lisp are available along with the HPRL functions. HPRL's rule system allowed separation of the program control from the knowledge base, thereby making incremental additions to the knowledge base possible without changes to the main control body of the program.

---

[2] productions are forward-chaining based

[3] instantiated means to make an instance of, or to create

1.3. Personal Motivation

The motivation for the RUBICC project came from my experience as a designer and manager of VLSI chips and systems at the System's Technology Operation (STO) of Hewlett-Packard Co., in Ft. Collins, Colorado. Examples of some of the work produced by STO's design team can be found in [23,24,25]. In early 1983, I managed a VLSI design team with limited experience in VLSI design. Being the only "expert" available, I spent many hours reviewing the group's designs.

The Knowledge (Rule) Base for the program itself resulted from my experience and the contributions of about 20 HP VLSI designers, mostly in Ft. Collins. RUBICC's current knowledge base is not intended to be complete or all encompassing. Rather, a substantial mix of rules was implemented to show the power of the program and to provide meaningful critique of interesting circuits. When completed, the current program surpassed my expectations. In one instance, RUBICC picked out a subtle error in a test case which I hadn't seen and which took on the order of a few minutes for me to understand.

1.4. Report Organization

The remainder of this report is organized as follows: In Chapter 2, considerations involved in building an expert system are presented and the choice of HPRL is explained. In Chapter 3, other related work applicable to this project is described and issues manifest in the design of this type of system are presented. A detailed overview of the HPRL programming environment is included in Chapter 4 and in Chapter 5, a description of the program, it's control structure, and frame hierarchy is given. Results, limitations and efficiency considerations are presented in Chapter 6. A summary and considerations for future work are given in Chapter 7. References are listed in Chapter 8. Five appendices are included. In Appendix A, the generic frame hierarchies of the system are illustrated. A complete listing of RUBICC's Generic-Frames is presented in Appendix B. The *Technology Frame*, *G-Con, appears in Appendix C. Examples of RUBICC's critiques are presented

in Appendix D, and a detailed summary of RUBICC's implemented circuit checks appears in Appendix E. A full source code listing is included in Appendix F.

CHAPTER 2

Expert System Considerations

Many Expert Systems have been created to perform a variety of tasks. A thorough treatment of the current state of the art of Expert Systems can be found in "Building Expert Systems" [26].

2.1. Expert System Characteristics

What are the characteristics of an Expert System? A common thread throughout the literature is that Expert Systems tend have the following five characteristics:

1. Performance as good as a human expert in some small domain.

2. Avoidance of blind searches through all possible combinations of data to produce an answer.

3. Reasonable efficiency -- This doesn't necessarily mean that it has to be faster than a human, only that its response time not be prohibitive.

4. Separation of Knowledge and Control Structures

5. Can answer "Why" -- This usually means it can show how it arrived at its conclusions.

These five items were used as overall guidelines in making design decisions throughout this project.

2.2. System Language Choice

A major design decision faced at the onset of this project was what language or language system to use. Generally, expert systems are written in symbolic languages such as Lisp. At the beginning of the project, I wondered why this work couldn't be done in "C" or "Pascal". Afterall, couldn't combinations of transistors be just as easily picked out from a "C" subroutine as an HPRL rule? What was unique about Lisp in this application?

The answers to these questions came later, after I gained experience with Lisp and HPRL. Lacking experience, I decided to take the AI Community's word and use a symbolic language.

In addition, what symbolic language would be best? I considered two languages: OPS-5 from CMU (Carnegie-Mellon University) [22] and HPRL from HP-Labs (Hewlett-Packard Laboratories) [19]. I decided to use HPRL which is based on PSL (Portable Standard Lisp) and is described in detail in Chapter 4. HPRL seemed to have all the characteristics that were said to be needed in the literature. HPRL differed from the OPS-5 in that backward chaining paradigms could be developed as well as the standard forward chaining productions of OPS-5. In addition, HPRL was developed by people I knew at HP Labs. I described my system to them and explained the design goals. They seemed to think that HPRL would be more than adequate for my needs. Another consideration was that as a loyal HP employee, I wanted to use an HP system and hopefully provide feedback on its strengths and weaknesses.

After writing RUBICC, I feel more capable of addressing these issues. HPRL is a powerful system. It was clearly and carefully thought out by people who recognized the issues inherent in building an expert system and in using machines for reasoning functions in general. A system such as RUBICC could be written in a language such as "C" though it would be harder, since the symbol manipulation features of Lisp are better attuned to handling the problems inherent in this application and all the HPRL functions would essentially have to be incorporated. Another key feature is that by its interpretive nature, each function can be "debugged" independent of the other functions. This greatly enhances coding productivity.

CHAPTER 3

Related Work

As previously mentioned, there exists substantial research in using Knowledge Based Expert Systems in the area of design synthesis. Surprisingly, there is very little research in progress for the analysis and critique areas of VLSI circuit design. In fact, Kelly's CRITTER SYSTEM at Rutgers [17] is the only other work I was able to find in this area. The CRITTER SYSTEM (Automated Critiquing of Digital Circuit Designs) was initially intended to be used for TTL SSI/MSI designs. It is currently being extended for use in the VLSI design environment.

Though both CRITTER and RUBICC have similar goals at the highest level (i.e., feedback to designers as to the quality and robustness of their design), their approaches to and emphasis on the problem are uniquely different. In this chapter, the fundamental issues of using machines as design critics are given. When appropriate, a comparison of CRITTER's and RUBICC's methods are included since they have much in common and at the same time have basic design differences.

3.1. Issue -- Why use Expert System Technology?

The idea of writing computer programs to check designs is not new. Electrical Rule Checking programs such as NCA's ERC Program [27], and ECAD's Dracula [28] exist to aid the designer in checking his design for electrical design rule violations. We used a program at HP called "Funny-Fet" which flagged transistors which had strange properties such as no connection to a terminal, or gate, drain and source shorted together. The Dialog Program [29] from the University of Leuven performs more in-depth checks of analog circuits.

These programs share a common characteristic - their knowledge tends to be *hardwired* in the sense that exact patterns and cases must be described in order for these programs to perform their checks. Due to this characteristic, it is very difficult to describe to these programs errors of a general nature which relate classes of transistors. Also, because of this *hardcoding*, detailed knowledge of the program is required to add or make changes to the knowledge-base. As a result, the checks performed, though highly useful, tend to be shallow. With the exception of Dialog, none of these programs can reason about complex timing or charge-sharing problems.

Expert System Technology provides an alternative to this situation and through its use, hardwiring can be avoided. Separation of the rule base from the program control allows incremental additions without affecting the overall program. The program can be "taught" to reason about general patterns of transistors and and rules can be written to check for classes of errors.

3.2. Issue -- What Kind of Critique does a Designer Need ?

For a m.chine to provide a useful critique, it should be accurate, provide an understandable summary of its analysis, allow the user to explore its conclusions in greater detail, and avoid showering the user with cryptic messages and long lists of items to be checked, some of which aren't real problems. With respect to the last point, the system should be conservative in that it is better to show a non-error than miss a real one.

In this respect, RUBICC and CRITTER are very similar. Both produce a critique in summary form. Both produce readable messages and both allow the user to go into more depth concerning conclusions reached. CRITTER provides suggestions as to how to fix the problem. RUBICC provides suggestions in some cases, though its major thrust as implemented is to flag and isolate the problems, leaving the fix to the designer. This is consistent with the types of errors RUBICC finds. Either the fix is obvious or the whole scheme needs to be rethought. CRITTER's user interface seems more sophisticated and

more interactive with the user.

3.3. Issue – Analysis Approach ?

This issue highlights a significant difference between RUBICC and CRITTER.

CRITTER's critique method is based on a heuristic use of simulation. As Kelly states on page 2 of his report [17], "It [CRITTER] collects comprehensive estimates of circuit performance by essentially emulating the operations of various circuit analysis techniques (e.g., subcircuit simulations, path delay analyses, proofs of key design specifications)." It then "summarizes this data for the engineer, spotlighting whatever seems most useful for diagnosing and fixing flaws ... ".

In contrast, RUBICC's critique method is based on heuristics of VLSI design obtained from the knowledge base of experienced designers. Pattern matching is performed between the sequences of transistors in the circuit and the programmed rule base. As various circuits are picked out, they are checked, and anomalies are reported. The patterns analyzed and checked by RUBICC are of a general nature in the sense that most rules check classes of objects rather than a specific "hardwired" pattern. For example, a single rule can pick out "nor gates" with arbitrary numbers of inputs. (an individual rule for a 2-input nor-gate, a 3-input nor-gate, ... is an example of hardwiring).

An advantage of CRITTER's simulation approach is that, in some sense, it's the bottom-line in terms of whether and how well a circuit functions. If, given a set of inputs, the outputs of the circuit perform as desired, then the circuit must work (given that proper simulation was performed). CRITTER also checks internal timing as well. From a set of timing constraints on the inputs and outputs, internal timing margins are flagged. This addresses a problem with the use of large simulation tools. Namely a designer might not notice a marginal or race condition on an internal node of a large circuit if the external outputs of the circuit seem to be correct. Simulation can also give

confidence to the designer concerning the logical correctness of his circuit.

Whereas the simulation of TTL SSI/MSI functions seems to lend itself to the CRITTER approach , it is not clear that this approach would do as well in the critique of VLSI circuits -- especially dynamic circuits sensitive to a variety of things such a charge sharing, clock overlaps and capacitive coupling. In these cases, it is either very difficult and time consuming or impossible to produce the "correct" worst case simulations to show the problem.

RUBICC tests the hypothesis that a useful and indepth critique can be performed on a VLSI circuit without simulation. Hence a conscious design decision was to NOT perform logic or circuit simulation. It is felt that other tools should be used to handle these aspects of the design.

The underlying idea behind RUBICC's critique method is that there are many circuit problems and anomalies that elude even the most careful worst case simulations, but can be found by careful, indepth thought. Many of these same problems would be caught by the proper simulation yet they are sometimes missed. The reasons for this range from inexperienced designers, to trying to keep track of too many details, to not being able to simulate the problem correctly. As an example of the latter, an error occurred on an HP VLSI chip because the circuit simulator didn't predict the long time constant associated with forming the inversion layer of an NMOS capacitor with a particularly long channel length. This caused the chip to have poor high frequency operation margins.

3.4. Issue -- How and When to Use the Tool ?

One danger of a design tool of this nature is that it can be used improperly. The key misuse of such a program would be to use it blindly as the Final-Say on whether a circuit had errors. The intended use RUBICC is to aid designers at the cell level. After a designer has simulated his circuit (logically and electrically), he would presumably submit it to RUBICC for a design review. He should think carefully about any errors flagged

by the program. If RUBICC flags no errors, the engineer's confidence in his design should be enhanced. However, it is the engineer, not RUBICC who is responsible for the proper operation of the circuit. Hence care and good judgement is needed to properly use a such a tool. In addition, any organization which supported this type of tool would need to provide a mechanism to enhance RUBICC's Rule Base with additional error checks.

3.5. Issue — How Efficient Does It Need To Be ?

Because of the interpretive nature of the Lisp Language, systems of this nature tend to be slower than simulation programs written in C, Pascal, or Fortran. RUBICC is no exception to this. As will be seen in the examples, a circuit with 50 transistors takes on the order of 40-50 minutes to analyze (note that no attempt has been made to optimize the code at this point). Is this too slow to be useful? I feel that even if it took overnight to perform a valid design review it would be useful. An experienced designer (if one could be found) would spend at least a few hours on a circuit of this complexity. However, efficiency is important where debugging a program is concerned and hence, it should not be completely ignored.

CHAPTER 4

HPRL Overview

HPRL (Heuristic Programming and Representation Language) was introduced in Chapter 1 of this report. In this chapter, HPRL is described in more detail, focusing on the key features of the system as used by RUBICC.

The HPRL system is comprised of approximately 2 Mbytes of Lisp source code. It provides a number of key features which led to a natural implementation of the RUBICC system. These features include hierarchical frame structures and inheritance among frames, pattern matching, backward and forward chaining rules, demons, and escapes to Lisp. These concepts are described in the remainder of this Chapter, drawing examples from the RUBICC System where appropriate.

4.1. Frames

Frames are the basis for the HPRL data base structure and are similar to records in "C" or "Pascal". Frames have slots which are the analog of a record's fields. Data is stored in the slots of frames in manner similar to storing data in the fields of records. The three basic differences between HPRL frames and C or Pascal records involve *generic-frames* and *inheritance*, the data stored in slots and dynamic storage management.

4.1.1. Generic Frames and Inheritance

Frames are arranged in a hierarchy starting with the distinguished frame THING. The THING frame sits at the top of this hierarchy and is hence the root of the hierarchy tree. Each frame has an *ako* (a kind of) slot which links it to a parent frame (likewise each parent has an *instance* slot which links it to its child). The frame hierarchy forms a tree in which frames must be either directly or indirectly *ako-thing* (a kind of thing). A frame is directly ako-thing if it has the frame THING stored as data in its ako slot. It is

indirectly ako-thing if there exists a path along *ako-links* from it to the THING Frame.

Generic frames store information which is typical to a whole class of entities. By design, these frames exist farther up in the hierarchy than their more specific children frames. Generic frames form templates where typical properties of a general class of entities are grouped. Values can be stored in generic frames as well as in more specific frames.

A process called *inheritance* is automatically supported by HPRL. Inheritance means that more specific frames inherit the properties of their more generic parents. Slots defined in parent frames by definition exist in the more specific children frames. Various functions are provided by HPRL to access values stored in the slots of frames. These functions determine how inheritance should be used in finding the data requested. One such function specifies that if there is no data in a particular slot of the specific frame named, return "nil."[1] Another function specifies that in this case, search through the frame hierarchy starting at the named frame, ending at the THING frame, and return the data from the requested slot of the first frame found containing data in that slot. If there is no data found, "nil" is returned.

In summary, inheritance allows information to be distributed throughout complex frame hierarchies and accessed appropriately as needed.

Generic frames are used in RUBICC to define classes of items which have common characteristics. Examples of these generic frames are *circuit-elements* (transistors, capacitors, supplies, etc.), *structures* (parallel-transistor-structure, etc.), and *circuits* (inverter, super-buffer, register cell, etc.). These frames are explained in detail in Chapter 5. Specific instances of these items are created when the circuit net-list is read in or as the program picks out various patterns of transistors matching the descriptions of these elements in the program rule base.

---

[1] Lisp symbol for the empty set

The frame hierarchy provides key features which are utilized throughout RUBICC. It is used to limit and prune searches through the data base. By appropriately formatting rules, a data base search can be limited to branches or sub-branches of the hierarchy tree, thereby facilitating more efficient searchs. In addition, it allows rules to be written more compactly and clearly. For example, writing a rule to reason about a specific class of transistor such as a precharger is easier, clearer, and more compact than writing its full description (an enhancement transistor with its gate tied to a clock, its drain tied to a supply and its source non-grounded) in the rule.

As an example to clarify this point, consider the Hierarchical Frame Structure of Figure 4.1 which shows a tree structure representing a small subset of the generic frames used in RUBICC. This tree is involved in the classification of transistors. The frame TRANSISTOR is a child of the frame THING. This is consistent with the hierarchy since a Transistor is more specific than a THING. *Loads* (depletion mode transistors) and *drivers* (enhancement mode transistors) are more specific instances of transistors and hence are children of *transistor* in the hierarchy. Below *driver*, two specific kinds of drivers are shown (*precharger* and *xfer-gate*). Likewise, below *load*, two specific kinds of loads are shown (*ckc* and *src*). Each of these specific transistors is differentiated by its connections in the circuit. For example, an *src-load* is one which has its gate and source connected together. Likewise an *xfer-gate driver* has either its source or drain tied to the gate of another driver and its other side not tied to ground. When rules are written, they can reason about all transistors or be restricted to only reason about *drivers* or be further restricted to reason only about *xfer-gates*.

The code in Figure 4.1 (alongside the tree) is the actual HPRL code required to implement the hierarchy . "DEFRAME" is one of the HPRL functions used to create a frame. Starting with the frame *transistor*, notice first that the value "thing" is found in its *ako* slot (the $value will be explained in Section 4.1.2). Notice that the general infor-

```
(deframe transistor          ;create transistor frame
   (ako ($value (thing)))    ;define slots
   (d-node)
   (g-node)
   (s-node)
   (width)
   (length)
   (l-div-w)
   (class)
   (status))

(deframe driver              ;create a child of transistor
   (ako ($value(transistor))))

   (deframe xfer-gate        ;create a child of driver
      (ako ($value (driver))))

   (deframe precharger       ;create a child of driver
      (ako ($value (driver))))

(deframe load                ;create another child of transistor
   (ako ($value (transistor))))

   (deframe ckc-load         ;create a child of load
      (ako ($value (load))))

   (deframe src-load         ;create a child of load
      (ako ($value (load))))
```

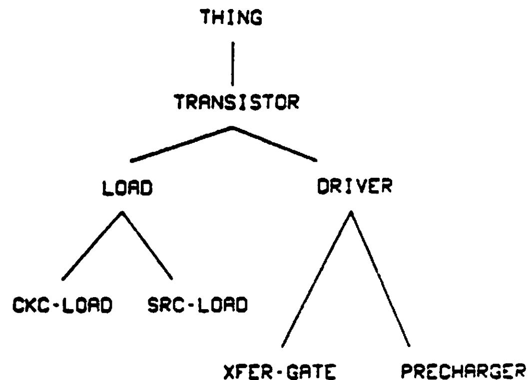

Figure 4.1 Hierarchical Frame Structure

mation common to all transistors is defined as the names of slots in the *transistor* frame. For example all transistors have a *width* (channel width), a *d-node* (drain node number) and an *l-div-w* (channel length divided by channel width), etc. If there was some property unique to a specific transistor, then a slot in that frame could be defined. Only frames existing below it would inherit that property.

Next the generic frame *driver* is created. Notice that it is *ako-transistor*; consistent with the frame hierarchy. Also notice, except for the ako slot, no other slots are named. Because a driver is ako-transistor, it implicitly inherits all the slots of transistor. Next, *xfer-gate* is defined to be a specific instance of *driver*. Since it is ako-driver, it inherits all the properties of a driver. In a similar manner the rest of the frames are defined to create the hierarchy. During RUBICC's execution, specific instances of the tree's leaf frames are

instantiated. For example precharger "m1" might be created. Specific data describing "m1" would be stored in "m1's" slots.

4.1.2. Data Stored in Slots

Unlike C or Pascal records, which can only store a single type of data in each field, slots of frames can have many different types of data stored in them and multiple instances of data of each type. The data stored in slots is actually stored in various "Facets" of the slot. The purpose of the "$VALUE" facet (all facet names begin with a "$") is to store data. This data can be any Lisp atom or s-expression. In addition, HPRL allows multiple values to be stored on a facet. HPRL treats multiple values on a Facet as a set in the sense that all HPRL rules which use this data understand that there may be more that one value, that the same value can't be entered twice on the facet, and that the order of data entry is indeterminate.

Each slot has associated with it a list of facets. Each facet has a name and a content (data). HPRL has a set of predefined facets, but in addition the user is free to create any addition facets. The significanc of the $VALUE facet is that there are built in HPRL access functions which assume this facet. The data stored in other facets is use to control interactions with the user ($ASK), restrict the type of data stored on the facet ($TYPE), and to specify procedures called DEMONS which are run whenever a slot's $Value data is modified ($IF-ADDED, $IF-REMOVED). The data stored in the $Value facet can also be a procedure name with parameters which is called to calculate the slot's value based on data elsewhere in the data base. (this procedure is referred to as a *procedural variable*).

Just as slots have facets with data, data can have tags with messages. Messages are used to comment the data as desired. The use of this in RUBICC is limited to an automatic HPRL function which places a message along with the data that is asserted into to a slot[2] as the result of a rule's conclusion. This message essentially explains "why" the data got

---

[2] Note that in this report, when it is said that data is asserted into a slot, the meaning is that data is asserted onto the $Value facet.

asserted.

A summary of the above points is shown schematically in Figure 4.2 (Frame Organization) [19].

The features provided by the flexible data storage facilities are used and relied upon throughout RUBICC. They lend to a natural, flexible implementation allowing complex associations and classes to be built.

4.1.3. Dynamic Storage Management

Since Lisp automatically handles the reclaiming of data structures which are no longer in use, the program doesn't have to worry about this problem. Memory is allocated as frames are created and data is added to these frames. When frames are no longer needed their memory is reclaimed by the PSL garbage collection facility. This garbage

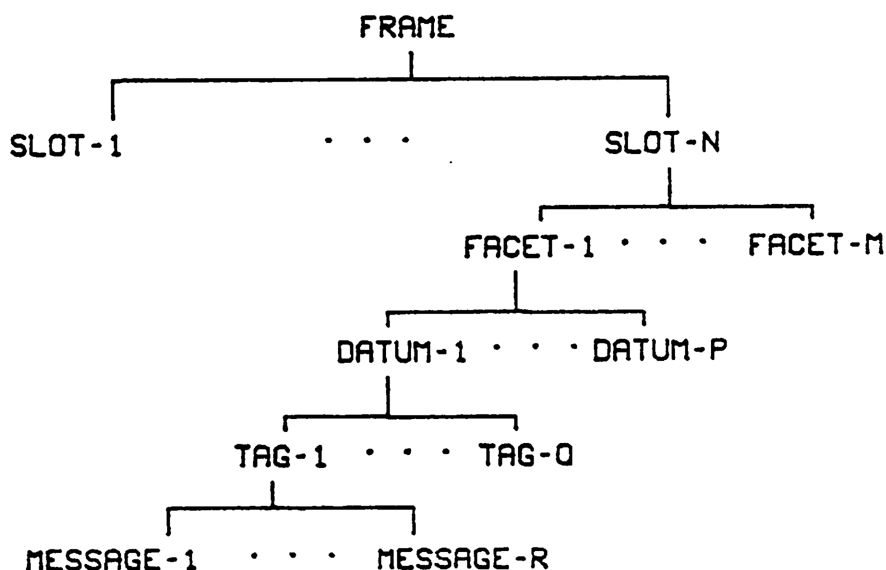

Figure 4.2 Frame Organization

collection imposes approximately a 10% overhead (i.e., the program spends about 10% of it's execution time in garbage collection) for normal size circuits.

4.2. Pattern Matching

Pattern matching is the search mechanism used in HPRL for querying the data base and "reasoning" about its contents. The pattern matching process is used to compare items for similarities, to reason about patterns of items and to make conclusions about them. Pattern matching variables in HPRL begin with a "?". Hence "?x", "?driver", "?foo" are pattern matching variables. Variables that do not begin with a "?" are called literals. Pattern matching variables can match anything. Literals are only allowed to match themselves. Pattern matching variables can be thought of as "wild cards", and a pattern matching search can be thought of as searching through a data base with wild cards (an excellent treatment of pattern matching is found in Chapters 21,22 of [30]).

To further illustrate pattern matching concepts, consider the following examples. Written in HPRL code, these examples concern dogs and whales, their breed, and the food they like. (this example is not representative of good frame hierarchies but it is illustrative of pattern matching concepts)

First a generic frame hierarchy is defined:

```
(deframe animal              ;define a frame called animal
   (ako($value(thing)))      ;make it ako-thing
   (breed)                   ;give it a breed slot
   (favorite-food)           ;favorite food slot
   (home))                   ;give it a home slot

(deframe dog                 ;define a frame called dog
   (ako($value(animal))))    ;make it ako-animal

(deframe whale               ;define a frame called whale
   (ako($value(animal))))    ;make it ako-animal
```

Next create some specific frames with data:

```
(deframe Fido                      ;define the frame Fido
   (ako($value(dog)))               ;Fido is a dog
   (breed($value(poodle)))          ;Fido is a poodle
   (favorite-food($value(caviar)))) ;Fido's like caviar

(deframe Morgan                        ;define Morgan
   (ako($value(dog)))                  ;Morgan is a dog
   (breed($value(golden-retriever)))  ;Golden retriever
   (favorite-food($value(chuck-wagon))));likes chuck-wagon

(deframe Dusty                     ;likewise for Dusty
  (ako($value(dog)))
  (breed($value(golden-retriever)))
  (favorite-food($value(purina-dog-chow))))

(deframe Moby-Dick                  ;Moby Dick is a whale
   (ako($value(whale)))
   (breed($value(white-whale)))     ;He's a white whale
   (favorite-food($value(Ahab))))   ;likes Ahab to eat
```

Now that a data base has been created, we can perform some queries. (".Solve" is the HPRL query function, and the "'" is needed to keep Lisp from treating the query pattern as a function with arguments). HPRL assumes that the ordered format for a solve clause is "frame, slot, value". (For readability, HPRL's response to a query is preceded by an arrow symbol ("->"):

```
(solve '(?x ako dog))    ;find a the dog in the data base
->(FIDO AKO DOG)
```

Here HPRL was asked to return any frame with the literal *dog* in its *ako* slot. HPRL returned the first one it found, namely the Fido frame. During the search, the variable "?x" was bound to the literal Fido.

```
(solve-all '(?x ako dog))  ;find all the dogs in the data base
->((DUSTY AKO DOG) (MORGAN AKO DOG) (FIDO AKO DOG))
```

"Solve-all" is the HPRL function used to find all frames with data matching the solve clause (?x ako dog). HPRL returned all the frames it found which were ako dog. In

this case the variable "?x" was bound successively to the three frames: Dusty, Morgan, and Fido.

```
(solve '(dusty favorite-food ?x))  ;find Dusty's favorite food
->(DUSTY FAVORITE-FOOD PURINA-DOG-CHOW)
```

Here Hprl returned Dusty's favorite food.

```
(solve '(and(?x breed golden-retriever)
           (?x favorite-food chuck-wagon)))
->(AND (MORGAN BREED GOLDEN-RETRIEVER)
         (MORGAN FAVORITE-FOOD CHUCK-WAGON))
```

HPRL supports conjunctions. First the variable "?x" is bound to any frame that has "golden-retriever" as its breed. This binding stays in effect throughout the rest of the conjunctive match. Then the frame's favorite-food slot is checked for the literal "chuck-wagon". If it exists, the conjunctive match succeeds and the result of the match is returned. If the match fails, HPRL looks for another frame whose breed is "golden- retriever". If no frames match, HPRL returns "nil", meaning that no data could be found in the data base matching the query.

```
(solve '(and(?x breed poodle)       ;find a dog
           (?x favorite-food Ahab))) ;who likes to eat Ahab
->NIL
```

Moby Dick's favorite food is Ahab. There is no poodle in the data base whose favorite food is Ahab.

```
(solve-all '(and(?x breed ?y)
                (?x favorite-food ?z)))

->((AND (MOBY-DICK BREED WHITE-WHALE)
        (MOBY-DICK FAVORITE-FOOD AHAB))
   (AND (DUSTY BREED GOLDEN-RETRIEVER)
        (DUSTY FAVORITE-FOOD PURINA-DOG-CHOW))
   (AND (MORGAN BREED GOLDEN-RETRIEVER)
        (MORGAN FAVORITE-FOOD CHUCK-WAGON))
   (AND (FIDO BREED POODLE)
        (FIDO FAVORITE-FOOD CAVIAR)))
```

This solve-all clause indirectly requested all items in the data base. The first time through, HPRL bound "?x" to "Moby-Dick", "?y" to "White-Whale", and "?z" to "Ahab". Since solve-all was used, it then removed those bindings and looked for additional bindings that would make the match succeed.

These working examples are intended to familiarize the reader with pattern matching and also introduce the syntax of the HPRL language. Other HPRL functions are available for pattern matching and accessing the values stored in the slots of frames. For further detail, the reader is referred to the HPRL Manual, Part 2 [19].

In summary, pattern matching is a flexible and powerful method of data base search.

4.3. Backward and Forward Chaining Rules

Rules provide the reasoning and inference mechanisms of the HPRL system. They form the "knowledge base" and are used to make conclusions about data patterns in the data base as well as create new patterns and associations of patterns.

4.3.1. Rule Overview

In HPRL, rules are frames that have a name and three slots: *type*, *premise* and *conclusion*.

The Conclusion Slot looks similar to a solve clause, in that it is of the form "frame, slot, value". The Conclusion Slot represents the action which is to take place if the conditions in the premise slot are found to be true. The Premise Slot represents conditions

which must be present in the data base for the rule conclusion to be asserted. The general form of the premise slot is also "frame, slot, value" though many forms (such as conjunction and disjunction) can be combined to form complex queries. The Type Slot of a rule is used to limit the data base search to certain branches of the frame hierarchy.

Whenever a rule concludes, some action occurs. The most common form of action is to assert the "value" specified in a rule's conclusion onto the "$VALUE" facet of the specific "frame" and "slot" also specified in the rule's conclusion. (this is always done for Backward Chaining Rules. Forward Chaining Rules allow other actions as well).

4.3.2. Backward Chaining Rules

Backward chaining rules are goal driven rules. They are invoked by the solve clause introduced in the previous section. In the context of Backward-Chaining Rules, the solve clause is called the user goal. To invoke backward-chaining, a goal is asserted using the solve clause as in the previous examples.

For illustration , consider the following backward chaining example, built upon the data base in the previous example:

First create two rules:

```
(rule whale-home-rule backward-chain-rule      ;define the rule
   (type (animal ?a))                         ;?a must be an animal
   (prèmise(?a ako whale))                     ;premise of rule
   (conclusion(?a home sea)))                  ;rule conclusion

(rule dog-home-rule backward-chain-rule
   (type (animal ?a))
   (premise(?a ako dog))
        (conclusion(?a home house)))
```

Each of these rules have their names following the atom "rule". Next the "backward-chain-rule" literal implies that this rule is to be use in goal directed searches. The type slot specifies that the pattern matching variable "?a" can only be bound to a frame which is either directl} or indirectly ako-animal. The first rule translates to the

following English description: "If any animal is a kind of whale, then its home is the sea." Likewise, in the second rule, a dog's home is a house.

Backward-chaining can be invoked using the solve clause as follows:

```
(solve '(Moby-Dick home ?y))

->Using Whale-Home-Rule:
   Since (moby-dick ako whale),
   Then (moby-dick home sea).

->(MOBY-DICK HOME SEA)
```

Here's a description of what happened. HPRL first looked at the "SVALUE" facet of Moby-Dick's home slot. If data existed there, it would have been returned. Since his home had not been asserted in the data base, HPRL collected all the rules whose conclusions match the user goal asserted. In this case it found both rules. HPRL tested the premises of both rules, found "whale-home-rules's" premise to be true and concluded that Moby-Dick's home was the sea. HPRL actually asserted the literal "sea" into the home slot as shown by the next HPRL command.

```
(fvalue-only 'Moby-Dick 'home)
->SEA
```

Fvalue-only is an HPRL frame access function which returns the data stored on the SValue facet of the frame and slot named.

Each premise clause of a backward-chaining rule is treated a user goal. HPRL will try to satisfy each user goal by data base lookup first. If that fails then it will try to solve it using additional rules. Hence a single solve command can trigger a very complex sequence of backward chaining rules. All clauses in the premise must be proven true for the conclusion to be asserted.

The conclusion of a rule can be made to do more than one thing using *procedural variables*. A function can be specified as the "value" part of a rule's conclusion. If the premise of a rule is proven to be true, this function is called. The function can generate

many "side-effects" before returning a value which is asserted into the $Value facet of the specified frame and slot. This technique is one of the main programming paradigms in RUBICC.

4.3.3. Forward Chaining Rules

Forward chaining rules are invoked when the frame data base is updated and hence are called *data-directed* or *data-driven* rules. Forward chaining rules are also known as *productions* in Production Languages such as "OPS-5" [22].

Forward chaining rules can be thought of as watching the data base for changes. As data is asserted, a certain rule's premise (or many rule's premises) may become true. When this occurs, the rule(s) conclusion(s) are asserted into the data base. This assertion may cause the premises of other forward chaining rules to become true. These rules assert their conclusions and the process continues until all rules are satisfied.

Forward chaining rules have a format similar to backward chaining rules. For more details see the HPRL Manual [19], part 2.

4.4. Demons

Demons are procedures which exist on the "$If-added" and "$If-removed" facets of slots. These procedures are called when data is added or removed from the "$Value" facet of a slot. Demons are similar to forward chaining rules and are usually used for "house keeping functions" which involve no pattern matching. For example, data asserted in the slot of some frame may also imply that data should be asserted somewhere else. A demon can be used for this purpose. They are also useful for debugging. Sometimes, it is hard to figure out how, why, or when data is being asserted into a specific slot. Temporarily adding an "If-added" demon to this slot allows functions to be called which aid in tracking down these problems.

4.5. Escapes to Lisp

To deal with situations which cannot be handled by the normal control flow of the pattern matching / rule system, it may be necessary to call a Lisp function. Procedural variables, and demons are examples of escapes to Lisp. An example of where it is used in RUBICC is as follows: RUBICC has rules which pick out patterns of transistors to make specific circuits out of them. When the premise of such a rule becomes true, the normal flow in HPRL would be to put a single piece of data into some slot. However, RUBICC needs to create a complete new frame structure for this circuit. Hence a function is called which performs these needed tasks as side effects.

CHAPTER 5

RUBICC System Design

The key design issues and decisions involved in the development of the RUBICC system were:

- Program Structure and Control
- Circuit Input Format
- Frame Data Base
- Program Control Paradigms
- Program Output Format
- Classification Strategies
- Separation of Program Control and Knowledge

In this Chapter, these issues are explained and a description how they are addressed in the RUBICC system is presented.

5.1. Program Structure and Control

Defining the program structure and control technique involved the tradeoffs between writing Lisp routines and using functions provided by HPRL. It was decided to write the overall program control routines in Lisp, utilizing the HPRL routines and the rule base as needed. This proved to be an effective method, taking advantage of the strengths of both Lisp and HPRL, without forcing either to handle situations to which it was not well suited. HPRL does not lend itself to easily handle some of the initial program setup and initialization requirements. In addition, the use of Lisp routines which perform explicit calls to HPRL's solve functions adds more control over the order of rule firings than is currently implemented in HPRL[1]. HPRL's frame data base, pattern matching, goal directed and data driven reasoning functions are used extensively throughout the program.

---

[1] HPRL's next release will provide agenda control and rule-firing ordering functions directly to the user

Data driven programming techniques (similar to those described in [31,32]) are used to keep the program control functions separate from the knowledge base. In this way incremental rules, new circuit patterns, and additional error checks can be added with minimum perturbation to the program control structures.

5.2. Circuit Input Format

At this stage, the circuit input format is similar to a Spice-Deck but written in Lisp list notation. It is planned to add additional information, such as which mask layer a net exists on, in a future version of RUBICC. An example of a circuit and is corresponding input format is shown in Figure 5.1. Each subexpression of the list begins with a type of circuit element. The types supported currently are *driver* (enhancement transistors), *load* (depletion transistors), *supply* (static power supplies), *clock* (clock driver), *capacitor* (non-mos capacitors) and *pad* (input pads of the chip). The next atom in the list is the name of the circuit element. The only restriction on names is that they must be unique throughout a given circuit. The atoms after the name represent appropriate values for elements. For example, the five numbers fter driver "m2" represent its drain node, gate node, source node, channel width, and channel length respectively. Similarly, the three numbers after the supply "v1" represent its positive node, negative node, and voltage. One other input format is allowed in RUBICC: If a sublist's first atom starts with a "*", RUBICC assumes that this atom is the circuit name.

RUBICC has a few built-in conventions. Ground is always considered to be node0. The negative node of a clock or supply is assumed to be connected to node0. Transistors can be entered with sources and drains used symmetrically. If transistors are entered with their drains connected to ground, an HPRL rule reverses the source and drain nodes. Likewise for transistors entered with their sources connected either to a supply or clock. Note that the last convention is not applicable to a CMOS Technology since, for example, a supply is usually considered to be connected to the source of a p-channel transistor.

```
((*sample-circuit)
(supply v1 1 0 5)
(clock ck1 8 0 5)
(pad in 3)
(load m1 1 2 2 4 8)
(driver m2 3 8 4 6 2)
(driver m3 2 4 0 6 2)
(driver m4 2 6 5 12 2)
(driver m5 5 7 0 12 2))
```

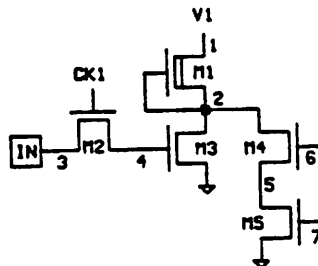

Figure 5.1 Circuit Input Format

The conventions that are adopted in the examples are that transistors begin with an "m" (i.e. m1 m2 ...), supplies begin with a "v" (i.e., v1 v2 ...), clocks begin with "ck" (ck1 ck2 ...) and capacitors begin with "c" (c1 c2 ...).

It is envisioned that in the future, the output of a layout extract program would be used as the input to RUBICC. In this case, either the extract output data could be "massaged" to the input format described or RUBICC's input format parser could be changed to recognize the new format.

5.3. Frame Data Base

The design of the frame data base and frame hierarchy evolved along with the program. The organization of this data base along with the assignment of slots to each frame is the paramount issue in the overall effectiveness of the program. It affects the reasoning capabilities of the program, the classification of various circuit structures, and the directness and simplicity of writing the rules to perform these tasks.

The frame data base is organized along three lines. Schematic Frames are those frames which hold the information about the circuit elements as they are initially entered and further classified by the program. Program Control Frames are used for program "housekeeping" functions and as storage for technology dependent constants used

throughout the system. Error Frames hold the names of transistors and circuit elements which violate rules and checks performed by the program.

In the remaining sections, these three frame hierarchies are described. The complete hierarchy is included in Appendix A. For a listing of the slots of these frames, see Appendix B.

5.3.1. Schematic Frames

A tree structure representation of the Schematic Frame Hierarchy is shown in Figure 5.2. Note that these are generic frames. Circuit specific frames of the appropriate types are instantiated as RUBICC runs. Just below the root ("Thing") there are six frames: *Transistor*, *Circuit*, *Struct*, *Node*, *Two-Port-Element*, and *One-Port-Element*. An explanation of each of these hierarchies is given in the following sections.

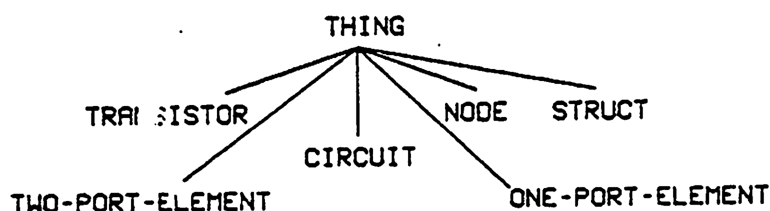

Figure 5.2 Schematic Frame Hierarchy

5.3.1.1. Transistor Frame Hierarchy

A representation of the Transistor Hierarchy is illustrated in Figure 5.3. The transistor hierarchy is further broken down into *drivers* (enhancement fets[2]), *loads* (depletion fets), and *mos-caps* (a transistor with its drain and source connected together).

[2] fet = field effect transistor

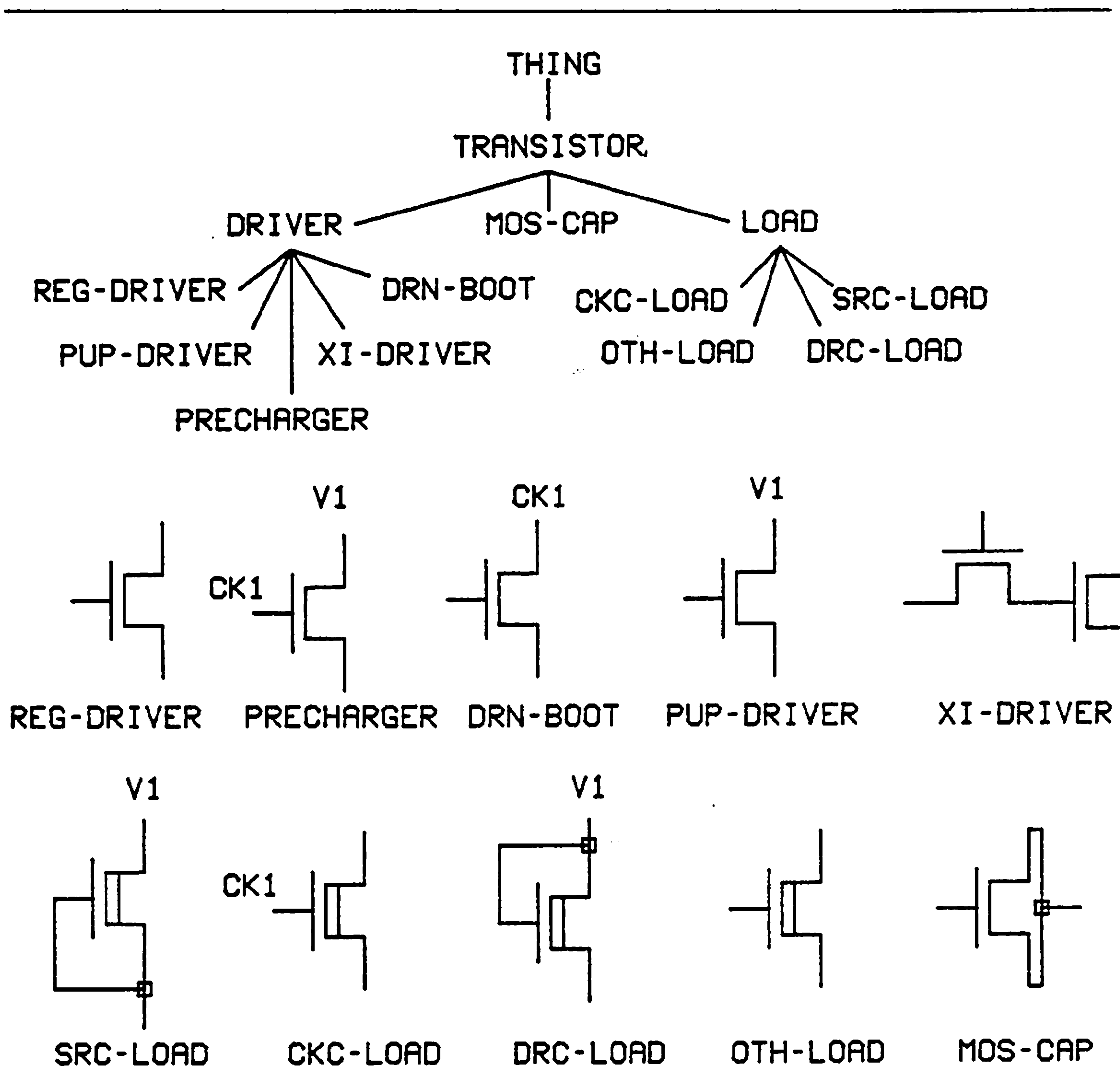

Figure 5.3 Transistor Hierarchy

Figure 5.3 Transistor Hierarchy

RUBICC further classifies *drivers* and *loads* by how they are configured in the circuit. These configurations, which are shown schematically in Figure 5.3, are:

Driver classifications:

precharger — driver with a clock tied to its gate and a supply to its drain

drn-boot — (drain bootstrapper) - driver with a dynamic gate node and a clock tied to its drain

xi-driver — driver with a transfer gate attached to its gate

pup-driver — driver with its drain tied to a supply and its gate not tied to a clock

reg-driver — (regular driver) - cannot be classified as any of the preceding classes

Load classifications:

src-load — load with its gate tied to its source

drc-load — load with its gate tied to its drain

ckc-load — load with its gate tied to a clock

oth-load — (other load) - cannot be classified as any of the other preceding classes

5.3.1.2. Circuit Hierarchy

The circuit hierarchy is shown in Figure 5.4. Circuits are combinations of transistors used to perform some basic function. Circuits are further divided into *Gates, Buffers Drain-Bootstraps* and *Registers.* This set forms the circuits which RUBICC can recognize and critique. (RUBICC also has rules which critique related sequences of transistors which are not necessarily classified into the above circuits).

Gates are combinational circuits having one or more inputs and (in RUBICC's domain) one output. Gates are further defined in RUBICC as having a single pull-up transistor with it's drain connected to a supply and a single pull down structure (combination of series and/or parallel fets) connected between the pull-up's source and ground. Gates are further subdivided into static and dynamic classes. This classification is based on whether the gate's pull-up transistor is a load (thereby making it static) or a driver. Figure 5.4 also shows the specific schematics which RUBICC recognizes for each or these types

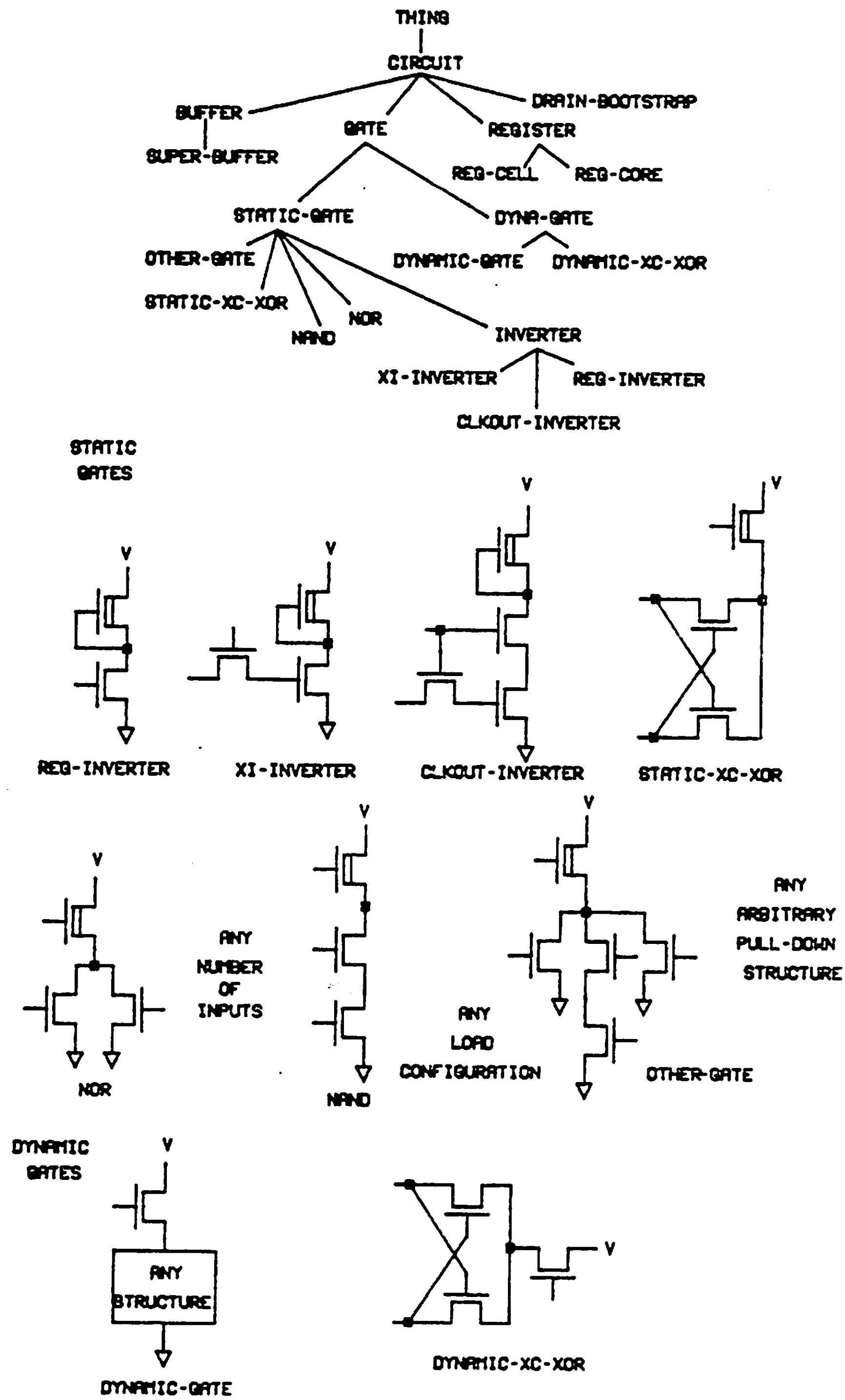

Figure 5.4 Circuit Hierarchy

of gates.

Buffers are circuits which amplify signals. They may or may not perform other logic functions on the signal. RUBICC recognizes a class of buffers called *Super-Buffers*. Two types are recognized: *inverting* and *non-inverting* buffers. These are shown in Figure 5.5.

Another type of circuit is a *Drain-Bootstrapper*. It is used to selectively switch a clock signal to a given node. Drain-bootstrappers always have a driver classified as a *Drn-Boot* in their output stage. In addition there is a *predriver* stage connected to the gate of the Drn-Boot. RUBICC recognizes three typical types of predriver circuits for drain bootstraps. These are shown schematically in Figure 5.5.

Registers are also recognized by RUBICC. A *Register-Cell* is composed of a *Register-Core* and various transistors forming the inputs and outputs. Schematics are shown in Figure 5.5.

In general, the lower level circuits such as inverters and gates are composed of related transistors. The higher level circuits such as buffers, drain bootstraps and registers are composed of combinations of transistors, gates and other structures.

5.3.1.3. Struct(ures) Hierarchy

Structures are combinations of series and parallel enhancement fets or other structures. No internal node of a structure may be connected to the gate of a transistor or to a depletion transistor. RUBICC recognizes most combinations of transistors composed in this manner and classifies them as follows:

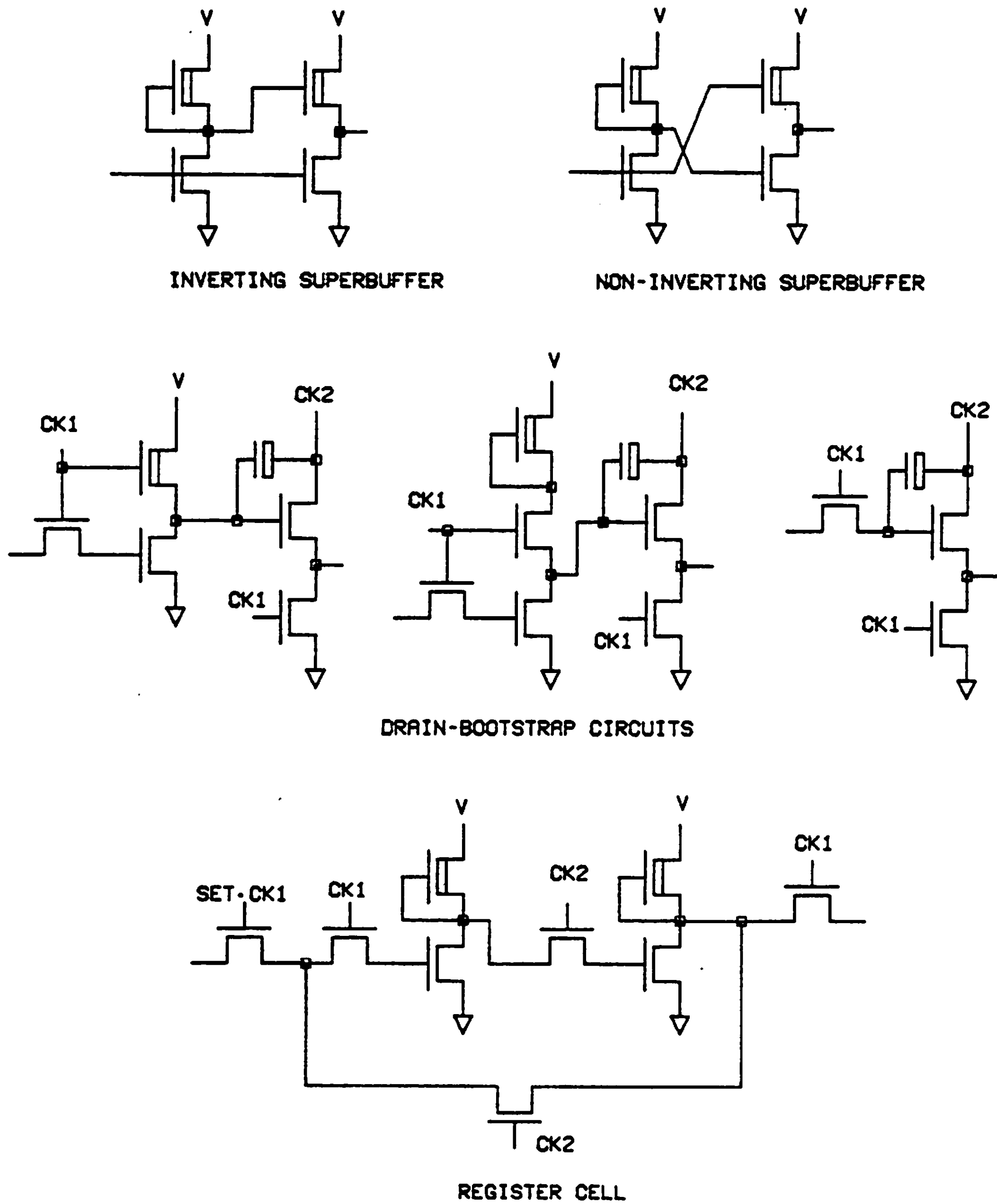

Figure 5.5 Other Circuit Schematics

series-struct — only composed of series driver fets

parallel-struct — only composed of parallel driver fets

super-struct — composed of combinations of series and parallel structs, super-structs, and single transistors.,

xc-xor-struct — a specialized structure corresponding to a potential piece of a cross-coupled xor gate.

These structures are shown schematically in Figure 5.6. They are a key pattern recognition and classification method used in RUBICC. A present weakness in RUBICC is that it cannot recognize certain combinations of transistors which a human would classify as a structure. An example of such a structure is shown in Figure 5.6. At this time, RUBICC would include these transistors in its list of circuit elements which were not checked. This problem can be solved by the addition of new rules.

5.3.1.4. Nodes

Nodes are frames which correspond to where and how circuit elements are connected. An individual node frame is created for each unique circuit node. Node frames store a large amount of data. They keep track of the gates, drain and sources, supplies and clocks connected to it. In addition, nodal capacitances are kept as values in appropriate slots. Nodes are not classified further in the hierarchy. However, they have *class* and *aspect* slots which are used to further identify their properties and which are important for reasoning about dynamic clocking situations. See Section 5.6 for more details.

5.3.1.5. Two and One Port Elements

Two port elements are classified as Active-Two-Port-Elements (ATP's) and Passive-Two-Port-Elements (PTP's). See Figure 5.2. ATP's are either supplies or clocks. The only PTP's which RUBICC recognizes at present are fixed capacitors. It is envisioned that in the future, RUBICC would also recognize resistors and inductors.

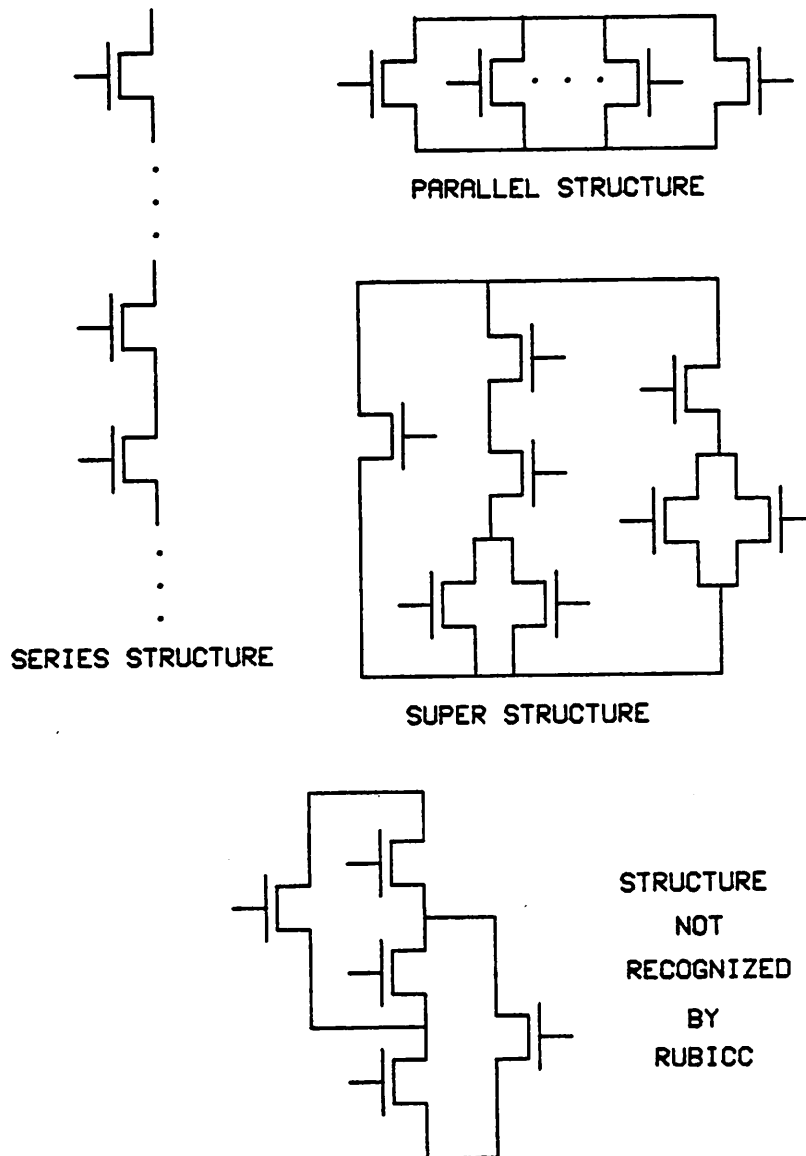

Figure 5.6 Structures

5.3.2. Program Control Frames

There are two frames used for program control. These frames are the Elements Frames and a frame called *G-Con. This hierarchy is represented in Figure 5.7.

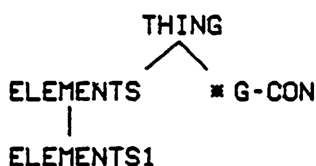

Figure 5.7 Program Control Hierarchy

5.3.2.1. Elements Frames

Elements1 is a frame used to hold the results of various solve commands given to HPRL. Whenever backward chaining is invoked, a concluding rule must assert data. *Elements1* provides slots named to correspond to invocations of the HPRL "solve" command and a corresponding place for rules to assert their data after their "side effects" have been completed. *Elements1* also proves useful for debugging and monitoring the progress of the program since it contains a record of all deductions completed.

5.3.2.2. *G-Con – The Technology Frame

A Technology Frame called **G-Con* (Global Constants) and is used to store all technology dependent parameters used for reasoning about the circuit by RUBICC. Examples of these constants are the beta-ratio's required for different load configurations, and gate oxide capacitance. The complete *G-Con frame is included in Appendix C.

5.3.3. Error Frames

Error Frames, whose hierarchy is represented in Figure 5.8, are used to store the names of circuit elements which have errors of the kind implied by the particular frame and slot. For example, consider the frame *inv-errors* (shown in Figure 5.9) which has the following slots: beta-ratio, coupling, and input-clocking. RUBICC checks all of the invert-

ers it finds in the circuit for these three types of errors. Suppose "reg-inverter-1" had a beta-ratio error. RUBICC would store the lisp atom "reg-inverter-1" on the "$Value" facet of the beta-ratio slot of the *inv-errors* frame.

The use of error frames facilitates a data-driven programming technique which is described in Section 5.4.

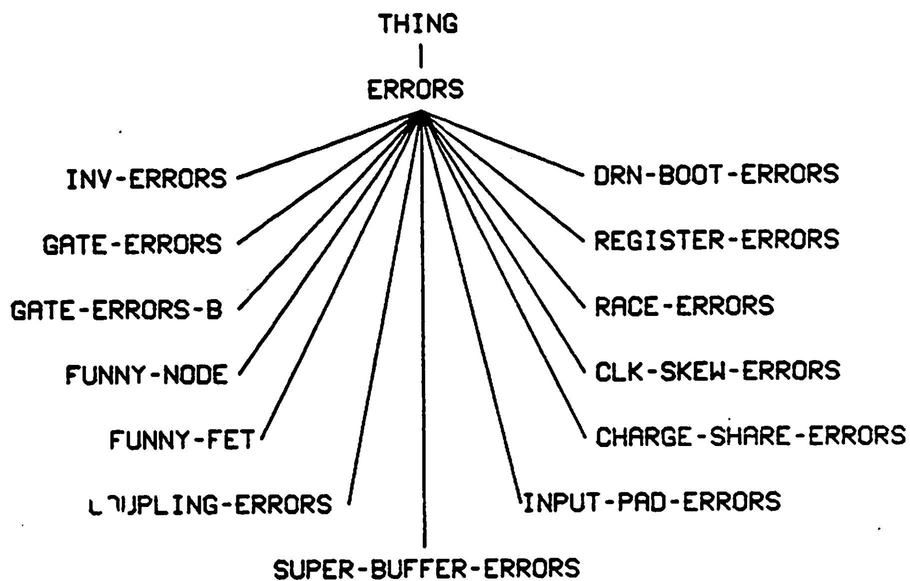

Figure 5.8 Error Frame Hierarchy

```
(deframe inv-errors
    (ako ($value (thing)))
    (beta-ratio)
    (coupling)
    (input-clocking))
```

Figure 5.9 Inverter-Errors Frame

5.4. RUBICC Programming Paradigms

RUBICC's main programming technique used for finding errors is as follows: First, backward-chaining rules are used to pick out patterns of transistors and circuits, after which either backward or forward chaining rules are used to check these patterns. Backward-Chaining is invoked explicitly with the HPRL "solve clause". Forward-Chaining is invoked implicitly by the existence of rules which trigger when data that is asserted into the data base makes their premise true. As higher level structures of transistors and circuits are built up, their constituents are marked "in-use". Rules which search to identify new circuit patterns are only allowed to construct them from "free" lower level elements. This method eliminates overlapping usages of transistors, structures and circuits.

The complete upper-level control function of RUBICC is listed in Appendix F, and is repeated in Figure 5.10 with numbers alongside each line. The following explanation of these 48 lines of source code is intended to clarify the previous discussion.

Line 2 defines the main circuit function "check-circuit". This function is called from the top level Lisp "read-eval-print loop" and is passed the name of a variable which is bound to the net-list of the circuit which is to be checked. Lines 3-7 perform global variable initialization.

Lines 8-11 perform the tasks of inputing and initializing the predefined Lisp functions, frames and rules. Line 12 is a call to the Lisp function "patom" which prints out the string following it. "Terpri" is the Lisp "line-feed" command.

The real work begins on Line 13. The function "create-i-frames" (create-initial-frames) is called with the net-list of the circuit to be checked as its actual parameter. This function successively takes each sublist from the net-list and instantiates the initial specific *transistor*, *mos-capacitor*, *capacitor*, *supply*, *clock*, *pad* and *node* frames. At this point, all transistors are either ako-driver, ako-load or ako-mos-cap.

```
;
1  (setq *input-functions nil)

;;;;;;;;;;;;;;;;;;;; main program control function ;;;;;;;;;;;;;;;;;;;;

2  (defun check-circuit (net-list)
3  (setq *circuit-name 'unnamed)
4  (setq gctime!* 0)
5  (setq *clka nil)(setq *clkb nil)(setq *clkc nil)(setq *clkd nil)
6   (setq *longrc 0)
7  (let( (s-time (time)) )
8     (input-functions)
9     (clear-frame-syms)
10    (input-frames-rules)
11    (make pop frame)               ; push frame marker onto *frames*
; create initial circuit frames
12    (patom "BUILDING CIRCUIT FRAMES"(tab 30))(terpri)
13    (create-i-frames net-list)
14    (patom "CIRCUIT-NAME: ")(patom *circuit-name)(terpri)
15    (patom "88 88 88")(terpri)
; transistor classification
16    (solve-all '(?transistor s-d-reversed ?x))
17    (solve-all '(?elements src-load ?x))
18    (solve-all '(?elements drc-load ?x))
19    (solve-all '(?elements ckc-load ?x))
20    (solve-all '(?elements oth-load ?x))     ; must be last load solve
21    (solve-all '(?elements xi-driver ?x))
22    (solve-all '(?elements precharger ?x))
23    (solve-all '(?elements pup-driver ?x))
24    (solve-all-drn-boots)
25    (solve all '(?elements reg-driver ?x))   ; must be last driver solve
; series-parallel combinations
26    (find-parallel-fets)
27    (find series-fets)
28    (cond((or(fchildren 'series-struct)(fchildren 'parallel-struct))
29          (combine-structs)))
30   (find-other-structs)
; check circuit for errors
31    (solve-error-frame 'funny-fet)
32    (solve-error-frame 'funny-node)
33    (solve-all '(?elements inverter ?x))
34    (solve-all '(?elements static-gate ?x))
35    (solve-all '(?elements dynamic-gate ?x))
36    (solve-all '(?elements super-buffer ?x))
37    (solve-all-drain-bootstraps)
38    (solve all-reg-cells)
39    (solve-error-frame 'race-errors)
40    (solve error frame 'gate-errors b)
41    (solve error-frame 'charge-share-errors)
42    (solve error-frame 'drn-boot-errors)
43    (solve error frame 'input pad errors)
44    (solve all clk skew-errors)
; print results
45    (show-circuit-errors)(terpri)
46    (show-all-gates-and-circuits)(terpri)
47    (show-not-checked)(terpri)
48    (print-stats)))
```

Figure 5.10 RUBICC Main Control Function

Lines 16-25 call the HPRL function "solve-all" to classify transistors in the frame hierarchy. The code in line 16 is used to find any transistors with sources and drains reversed. A trace of how this rule works will help clarify RUBICC programming techniques. When Line 16 is executed, the following backward chain rule whose conclusion matches (from a pattern matching point of view) the goal (?transistor s-d-reversed ?x) will fire if any transistors have their sources and drains reversed.

```
(rule reverse-src-drn-rule backward-chain-rule
    (type (transistor ?tr)(active-two-port-element ?ae))
    (premise (or (and (?tr s-node ?sn)
                      (?ae pos-node ?sn))
                 (?tr d-node 0)))
    (conclusion(?tr s-d-reversed ^(reverse-s-d ?tr))))
```

The type slot of this rule specifies that "?tr" is a pattern matching variable which can be bound only to frames that are part of the transistor hierarchy. Likewise, "?ae" can only be bound to a supply or clock.

The rule's premise contains a disjunctive clause consisting of two subclauses either of which, if proven true, will cause the conclusion to be asserted. The first subclause is a conjunction which, translated in English says: "Find a transistor and bind the pattern matching variable "?sn" to its source node number. If there is an active-two-port-element connected whose positive node is also connected to this node then the premise is true." The second clause says: "Find a transistor with it's drain connected to node0 (ground)."

If the premise is proven true, the conclusion is asserted. The conclusion has a procedural variable in its "value" position (signified by the "^"). The action performed by the conclusion is to assert the data returned by the Lisp function "reverse-s-d" into the "s-d-reversed" slot of the transistor frame which is bound to the pattern matching variable "?tr". This function always returns "t"[3]. In addition, it also causes two side effects to

[3]Lisp symbol for "true"

"?tr", namely two swap the source and drain nodes of the transistor which is currently bound to "?tr". Note also that the variable "?tr" was passed as a parameter to the function "reverse-s-d".

Since the "solve-all" clause was used, HPRL will search through the entire data base and find all transistors which satisfy the premise of this rule.

This represents the key programming method in RUBICC. Rules which classify different patterns of transistors are invoked by "solve-all" commands. If the rules which pick out these patterns conclude, a function is called. Depending on the situation, this function will perform some action, such as switching the drain and source node of a transistor, or instantiating a new specific frame corresponding to the new element picked out and putting appropriate values in the slots of the new frame. When the data in the new frame is complete, Forward-Chaining Rules which trigger off this data preform error checks, putting the name of the offending circuit element in an appropriate slot. (In the above case, there are no forward chaining rules which trigger on transistor classifications).

Lines 17-25 are additional examples of "solve-all" clauses which classify transistors

The functions called in lines 26-30 find all series and parallel transistor structures. These are Lisp routines which invoke HPRL solve functions. After their execution, specific instances of series, parallel and super structures corresponding to these patterns of transistors will have been created.

Lines 31-43 perform the bulk of the circuit checks and critiques. The "solve-error-frame" function on line 31 is utilized for a data-driven programming technique. This function is passed the name of a specific error frame. Using HPRL access functions, a list of all the slots contained in the error frame is built up. "Solve-Error-Frame" then iteratively forms explicit "solve-all" commands using the names of these slots in conjunction with the name of the specific error frame. In this way additional error checks can be added to the program by simply adding new rules and additional slots to an appropriate

error frame. If a new error frame needs to be created, then only one single line need to be added in the main control function -- a call to "solve-error-frame" with the name of the new error frame.

The "solve-all" command in line 33, instructs HPRL to find all the inverters that exist in the circuit using rules that have the form (?x inverter ?y) in their conclusions. These rules are found in Appendix F (in module inv-rules). As inverters are identified, functions are called which instantiate them and fill their slots with appropriate data. Forward chaining rules then fire which perform various checks on the new inverter. Forward chaining is another form of data driven programming in that if a new inverter check is desired, all that is required is a new forward chain rule. (If it doesn't make sense for the new rule to conclude its answer in a currently existing error frame slot, then a new error slot would also be added to the system.)

Line 45 is a call to the function "Show-Circuit-Errors", which prints out RUBICC's error summary. Show-Circuit-Errors knows how to find all error frames in the hierarchy. This is another example of data driven programming. If additional error frames are added, no change to the function is required to print out new errors found in these frames. Lines 46-48 call additional summary generating functions.

5.5. Program Output Format

The output of the RUBICC is a summary of errors as shown in Figure 5.11. Each error frame containing error data is printed out along with the corresponding slot name and data. Error frames and slots are named to give the user some idea of the nature of the error. Next, RUBICC prints out the transistor constituents of the various gates and circuits it picked out. The last part of the summary is a list of all the transistors which RUBICC didn't know what to do with and hence, probably weren't criticized. This is useful information because it tells a designer what hasn't been looked at. It also may indicate a problem (either with the Circuit or RUBICC) or it may indicate an additional case for RUBICC

to check. Finally, the summary is concluded with statistics such as the total run time of the program and the time spent during the run in garbage collection.

RUBICC's user interface could be improved upon. At present, a user would have to read the documentation about the error slots to find out more details about the error. In addition, the user would have to know something about the HPRL system to be able to ask "why" a certain error occurred or which rule fired to cause a certain error. The "hooks" exist within HPRL and the RUBICC system to improve the user interface.

5.6. Classification Strategies

Throughout the program design, decisions were made as to how and when to classify various patterns of transistors, circuits and nodes. In this Section, the strategies involved in these decisions are explained, using examples from RUBICC as appropriate.

```
ERRORS FOUND FOR CIRCUIT: *NET-LIST-14-DYNAMIC-CLOCKING
COUPLING-ERRORS
   XI-DRIVER-COUPLING      (M22)
   XI-DRIVER-COUPLING-1     (M10 M18 M34)
FUNNY-NODE
   CLOCKING-FLAG         (NODE4)
GATE-ERRORS
   DYNAMIC-CLOCKING-1      (DYNAMIC-GATE-7 DYNAMIC-GATE-4 DYNAMIC-GATE-1)
   DYNAMIC-CLOCKING-2      (DYNAMIC-GATE-7)
   DYNAMIC-CLOCKING-4      (DYNAMIC-GATE-6)

CIRCUITS and GATES IDENTIFIED:
  DYNAMIC-GATE-7  (M1 M22 M21)
  DYNAMIC-GATE-6  (M33 M34 M35)
  DYNAMIC-GATE-5  (M28 M29 M30)
  DYNAMIC-GATE-4  (M13 M14 M15)
  DYNAMIC-GATE-3  (M11 M10 M12)
  DYNAMIC-GATE-2  (M6 M7 M8)
  DYNAMIC-GATE-1  (M17 M18 M20 M19)
  XI-INVERTER-2  (M27 M26 M25)
  XI-INVERTER-1  (M2 M4 M3)

FREE TRANSISTORS:
   (M24)
garbage collection time = 4.116 min
total run time= 20.88633 min
```

Figure 5.11 Error Listing

5.6.1. When to Classify

The tradeoffs inherent in when to cause classifications to occur involve program efficiency and simplicity. If all items are classified in all ways, the program becomes prohibitively slow. Also, much of the work performed in creating data structures and classifications may not explicitly be needed by the program. For example, in many circuits it is not relevant whether a node is *dynamic*, hence work that was performed by the program to classify the node as such is wasted effort. On the other hand, classifications help to simplify the writing of rules and tend to "prune" the search space. For example, it is easier and more efficient (from a search point of view) to write a rule specifying a *precharger* than a "driver with its gate tied to a clock, its drain tied to a supply and it's source not tied to ground."

The strategy followed in RUBICC was to completely classify items only when it was clear that program efficiency and/or simplicity would be positively affected. In these cases, solve-all clauses are invoked which search through the entire data base. An example of this strategy is the fet classification statements in lines 16-25 of the main program control function (Figure 5.10). Unless this criteria was met, classifications were not done until specifically required by the program. The node classification strategy mentioned in the previous paragraph is an example of this approach.

5.6.2. How to Classify

In general, a different *classification-class* is required for each unrelated item that is to be checked for. Setting these classifications up in an efficient, consistent manner is a major design issue. Too many classifications make rules clumsy, hard to read and hard to understand. Too few actually limit the functionality of the program. Careful thought in this area can really pay off.

An example of such a problem is the node classification scheme in RUBICC, which is tricky because there are many seemingly unrelated ways in which nodes function in

dynamic circuits. To reason about complex clocking schemes, is necessary to have this information available. For example a node can be *static* (always driven) or *dynamic* (capable of being high impedance). It can be connected to a clock, connected to a driver whose gate is connected to a clock; the driver being either a precharger or pull down, or in the middle of some series/parallel fet string. A node can always be low on a given clock, or will sometimes be low, depending upon some other logic function such as the firing of a drain-bootstrap circuit. Another piece of information required is when are these nodes active or driven. If driven by a clock, which clock phase? If a node is held low by a clock and then released, how should it be classified?

The scheme adopted in RUBICC is to give nodes two attributes: *Class* and *Aspect*. Class refers to how it functions in the circuit, Aspect refers to what causes it to perform the function. For example, if clk1 (clock-driver-phase-1) is connected to a node3, then node3's class is *always-clocked* and it's aspect is *clk1*. Suppose the output of a drain-bootstrap is connected to node4 and the drain bootstrap can fire on clk2. Node4's class is *clocked-conditional* and its aspect is *clk2*. A summary of the Class and Aspect for nodes is given in Figure 5.12. This straightforward scheme makes all the necessary information about a node's function available for reasoning about complex clocking structures.

Node Class	Node Aspect
dynamic (node can become high-impedance)	nil
always-clocked	clock phase
conditional-clocked	clock phase
clocked-low (always low on a given clock)	clock phase
always-high	supply

Figure 5.12 Node Classifications

Backward chain rules exist to determine a node's classification and aspect. They are invoked as needed in other rules by using the user goal "(?node class ?x)". Hence, only those nodes which are specifically needed by the rule system are classified.

5.7. Separation of Program Control and Knowledge Base

The separation of program control from the knowledge base is achieved in RUBICC by the Rule Domains inherent in HPRL and the data-driven techniques described previously. This is a very powerful concept in that it allows allows additional rules and error checks to be incrementally added to the system. A new rule which checks an existing structure can be added by including the rule in the rule system and perhaps creating a new error frame. If a new class of circuits is to be picked out, a new generic frame must be included as well as the rules for picking this circuit and an additional solve-clause would be added to the main Check-Circuit function. (This seems to be a change in the program control – and it is. However, it is very simple, and straightforward. One could imagine a data driven technique by which this new "solve-clause" is added to a list which is used by the main program control loop.)

This separation is not achieved entirely throughout RUBI... The lower level transistor classifications are interrelated with Lisp function calls. If additions or changes to the transistor classes are made, certain functions must be modified. There is no fundamental reason for these dependencies; rewriting these routines could remedy this situation.

CHAPTER 6

Results

RUBICC was run on a number of test circuits included in Appendix D. These circuits show the various types of errors which RUBICC can find. The real utility of the program became apparent when RUBICC discovered errors which were not deliberately put in these test-cases. This was especially significant since the creator of these cases was a "highly experienced" , "seasoned" designer (the author). This example is described in detail in Appendix D, the example for Net-List-27.

As previously mentioned, RUBICC's knowledge-base contains 110 rules. These rules break down as follows: A little over 50% are used in the classification of objects, a little less than 50% are used to check for error cases and the remaining 10% or so are used for "house-keeping" functions such as reversing transistors sources and drains. RUBICC is approximately 110 KBytes of source code.

During the Spring Semester of 1984 at U.C. Berkeley, I learned Lisp, investigated Expert Systems, and became familiar with HPRL. Prior to this, I had no experience in any of the above subjects. Starting from scratch, I spent about 80 hours learning HPRL. The current RUBICC System was coded in about 450 hours (11 man-weeks) over a seven week period. This would not have been possible without the excellent introduction to Lisp and Artificial Intelligence Programming Techniques covered in U.C. Berkeley's CS283 Course [31] which I took in the Spring, and the productive coding environment provided by Lisp and HPRL.

The current implementation is very close to being usable for checking actual circuits. About two weeks would be required to update the Technology Frame with process specific parameters. In addition, another man month would be involved in running cells

through, checking the results, and extending the rule base.

CHAPTER 7

Conclusions and Future Work

7.1. Conclusions

As a system, RUBICC has met its goals which were:

1. Showing the feasibility of using Expert System Technology to build a system which provides meaningful critique of circuits.

2. Showing feasibility of encapsulating a Knowledge-Base of Design-Heuristics.

3. Determining the productivity of using Lisp and HPRL in writing such a system.

The design examples and errors checked show that Goal 1 has been met. One could imagine using this tool as part of the design cycle to check new designs and to aid in the training experience of new designers. The Rule System as written shows that Goal 2 can be met. It is envisioned that while the entire knowledge base would never be fully encapsulated, a major subset would evolve from the contributions of engineers and designers, working either specifically to add rules to RUBICC or as error cases arose which RUBICC didn't check. HPRL and PSL proved to be highly productive, as shown by the program statistics mentioned in Chapter 6. In addition, the system is very close to being usable for real applications.

7.2. Future Directions

Further work on RUBICC should be considered in the following areas:

7.2.1. User Interface

The user interface should be improved to better aid the designer in understanding the errors found by RUBICC. A number of items could be included: First a documentation file which contained an explanation of each error, and a proposed fix for that error, written by the author of the error check, would be helpful. The user could display this information to obtain more information about the error. In addition, HPRL provides information as to why an item of data was asserted into the slot of a frame. This information takes the form of the rule that asserted the data and is very cryptic. A user interface function could be written to take better advantage of this feature of HPRL.

7.2.2. Checking Actual Circuits

RUBICC should be extended to handle one or more technologies utilized by U.C. Berkeley and run on complete cells from a real chip design. To perform this task, the technology file must be updated for a particular process technology. Cells from a chip design could be hand coded into RUBICC input format or a translator could be written to convert chip "extract" outputs appropriately.

7.2.3. CMOS Compatibility

RUBICC's rule base and frame hierarchy must be reimplemented for CMOS critique. Though a significant task, RUBICC control structure and program paradigms would still be valid, hence this does not involve a complete rewrite.

7.2.4. Additional Structure Classification Algorithms

As previously mentioned, RUBICC doesn't handle certain structures such as the one shown in Figure 5.6. A more general algorithm could be devised for picking out these types of structures. The utility of this algorithm would have to be weighed against the costs involved versus the benefit it would provide.

7.2.5. Program Tuning

A 1.3-1.5x performance improvement could be achieved by compiling RUBICC's Lisp routines. This speedup is estimated as follows: HPRL routines are already compiled. The program probably spends no more that 1/4 - 1/3 of its time in interpretive Lisp code. If the time spent in this code went to zero (by compiling), the predicted performance improvement would occur.

Another improvement in efficiency could come by program tuning. A histogram of the times spent in each routine would show where to start. Suspected slow algorithms are the series / parallel algorithms and the clock skew sensitivity checks.

CHAPTER 8

References

[1] Nagel, L.N., "Spice 2: A Computer Program to Simulate Semiconductor Circuits," *University of California, Berkeley*, Memo ERL-M520, May, 1975.

[2] Saleh, R., Kleckner, J., Newton, A.R., "Iterated Timing Analysis in Splice 1," *Proceedings of the IEEE International Conference on Computer- Aided Design*, Santa Clara, Ca., 1983, pp. 139-141.

[3] Newton, A.R., "The Simulation of Large-Scale Integrated Circuits," *University of California, Berkeley*, Memo # UCB/ERL M78/52, July, 1978.

[4] Lelarasmee, E., "The Waveform Relaxation Method For Time Domain Analysis of Large Scale Integrated Circuits: Theory and Applications," *University of California, Berkeley*, Memo# UCB/ERL M82/40, May, 1982.

[5] Ousterhout, J. K., "Crystal: A Timing Analyzer for NMOS VLSI Circuits," *University of California, Berkeley*, EECS Technical Report # UCB/CAD 83/115A, 1983.

[6] Deutsch, J.T., "Behavioral-Level Simulation and Synthesis of Digital Systems," *University of California, Berkeley*, Memo # UCB/ERL M83/47, August, 1983.

[7] Keller, K., Newton, A.R., "KIC2: A Low Cost, Interactive Editor for I.C. Design," *Digest of Papers, Compcon 82*, IEEE Computer Society, 1982, pp. 305-306.

[8] Ousterhout, J.K., "Caesar: An Interactive Editor for VLSI Layouts," *VLSI Design*, Q4, 1981, pp. 34-38.

[9] Keller, K., "A Electronic Circuit CAD Framework," *University of California, Berkeley*, Dept. of EECS, Prof. A.R. Newton, June, 1984, Memo # UCB/ERL M84/54.

[10] Ousterhout,J.K., et.al., "Magic: A VLSI Layout System," *Proceedings of the 21st Design Automation Conference*,, June, 1984, pp. 152-159.

[11] Mah, G., Newton, A.R., "Panda: A PLA Generator for Multiply-Folded PLA's," *Proc. I.E.E.E. Int. Conf. on Cad*, Santa Clara, Ca., Nov., 1984, to appear.

[12] "CAD Tool Box User's Manual," *University of California, Berkeley*.

[13] Kowalski, T.J., "The VLSI Design Automation Assistant: A Knowledge Based Expert System," *Carnegie-Mellon University*, Report # CMUCAD-84-29, April, 1984.

[14] Kim, J., McDermott, J., "TALIB: An IC Layout Design Assistant," *Proceedings of the National Conference on A.I.*, American Association for Artificial Intelligence, William Kaufman, 1983.

[15] Birmingham, W.P., "MICON: A Knowledge Based Single Board Computer Designer, " *Carnegie-Mellon University*, Report # CMUCAD-83-21, December, 1983.

[16] Zipple, R., "An Expert System for VLSI Design," *1983 IEEE Symposium on Circuits and Systems*, 1983, pp. 191-193.

[17] Kelly, V.E., "The CRITTER System – Automated Critiquing of Digital Circuit Designs," *Rutgers University*, Rutgers AL/VLSI Project memo # 13, May, 1984.

[18] Beyers, J.P., Hewlett-Packard Co., Cupertino I.C. Operation, July, 1984.

[19] Lanam, D., et.al. "Guide to the Heuristic Programming and Representation Language,", Parts 1-3, *Application Technology Laboratory*, CRC, Hewlett-Packard Laboratories, 1984, available under license only.

[20] Goldstein, I.P., Roberts, R.B., "The FRL Manual" *Cambridge, Mass.*, Memo # 409, 1977.

[21] Griss, M.L., et.al, "HP 9836 PSL User's Guide," *Application Technology Laboratory*, CRC, Hewlett-Packard Laboratories, 1984.

[22] Forgy, C.L., "OPS-5 User's Manual," *Carnegie-Mellon University*, Memo # CMU-CS-81-13ᶜ 1981.

[23] "Hewle.t-Packard Journal", Volume 34, No. 8, August, 1983.

[24] Beyers, J.W., et. al., "A 32b VLSI Chip," *Digest of Technical Papers*, 1981 IEEE International Solid State Circuits Conference, Tham 9.1, 1981.

[25] Mikkelson, J. et. al., "An NMOS VLSI Process for Fabrication of a 32b VLSI Chip," *Digest of Technical Papers*, 1981 IEEE International Solid State Circuits Conference, Tham 9.1, 1981.

[26] Hayes-Roth, et. al. "Building Expert Systems,", *Addison-Wesley*, 1983.

[27] NCA Corp., "Electrical Rules Check User's Guide, Version 3.0," Santa Clara, Ca., April 1984.

[28] ECAD Inc., "ERC-Dracula," Santa Clara, Ca., January 1984.

[29] DeMan, H., "Dialog," *Catholic University of Leuven*, Leuven, Belgium.

[30] Wilensky, R., "LispCraft," *W.W. Norton Co.*, New York, N.Y., 1984.

[31] Wilensky, R., "CS-283: Artificial Intelligence Programming Techniques," *University of California, Berkeley,* (course), Spring, 1984.

[32] Charniak, E. et.al., "Artificial Intelligence Programming Techniques, "*Lawrence Ealbaum Co.,* 1980.

Appendix A - Frame Hierarchies

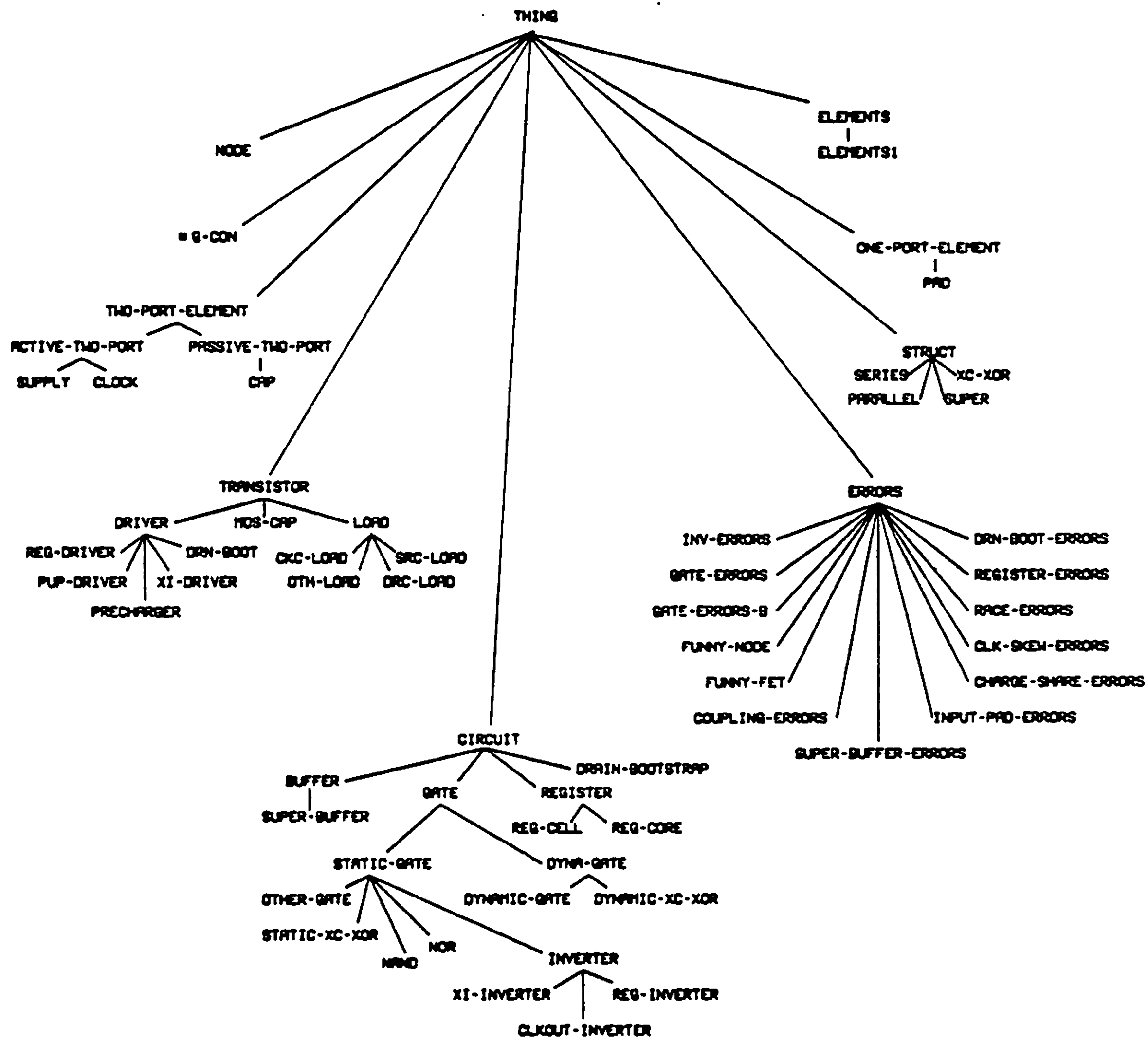

Figure A-1 Complete Generic Frame Hierarchy

Appendix B - Generic Frames

```
;*********************FILE FRAMES-1SL *********

(deframe transistor
      (ako($value(thing)))
      (d-node)
      (g-node)
      (s-node)
      (width)
      (length)
      (l-div-w)
      (string-l)
      (clk-input($ask(dont)))
      (clk-class)
      (class)
      (s-d-reversed)
      (trigger)
      (fb-tran-flag)      ; t if transistor is used as a feedback transistor
      (o-ins($ask(dont))) ; used for drain bootstrap rules
      (check)
      (status))   ; ($if-added((my-print :frame :value))))

(deframe load
      (ako($value(transistor))))

(deframe src-load
      (ako($value(load)))
      (instance($if-added((fput :value 'status '$value 'free))))) (deframe ckc-load
      (ako($value(load)))
      (instance($if-added((fput :value 'status '$value 'free))))) (deframe drc-load
      (ako($value(load)))
      (instance($if-added((fput :value 'status '$value 'free))))) (deframe oth-load
      (ako($value(load)))
      (instance($if-added((fput :value 'status '$value 'free)))))

(deframe driver
      (ako($value(transistor))))

(deframe reg-driver
      (ako($value(driver)))
      (instance($if-added((fput :value 'status '$value 'free))))) (deframe xi-driver
      (ako($value(driver)))
      (instance($if-added((fput :value 'status '$value 'free))))
      (xfer-gate) ;($if-added((fput :value 'status '$value 'in-use))))
      (fb-tran)   ;($if-added((fput :value 'status '$value 'in-use))))
      (in-node)
      (ga-xfw-ratio))   ; gate area to xfer-gate width ratio (deframe mos-cap
      (ako($value(transistor)))
      (instance($if-added((fput :value 'status '$value 'free))))) (deframe precharger
      (ako($value(driver)))
      (instance($if-added((fput :value 'status '$value 'free))))
      (pre phase)
      (supply)) (deframe pup-driver
      (ako($value(driver)))
      (instance($if-added((fput :value 'status '$value 'free))))
      (supply)) (deframe drn-boot
      (ako($value(driver)))
      (instance($if-added((fput :value 'status '$value 'free))))
      (s-node($ask(dont)))
      (other-clk-hold-down
                  ($if-added((freplace :value 'status '$value 'in-use)))
```

```
                    ($ask(dont)))
      (boot-phase($ask(dont)))
      (xfer-gate)
      (fb-tran))

(deframe node
      (ako($value(thing)))
      (instance($if-added((fput :value 'status '$value 'free))))
      (number)
      (status)            ;class/aspect doc: (class(allowable aspects))
      (class($ask(dont)))  ;((static(load push-pull)) (dynamic-hiZ(clk-phase))
      (class-1($ask(dont))) ;(clocked-always(clock-phase)) (clocked-conditional
      (aspect($ask(dont))   ; (clock-phase)) (precharge (clock-phase))
            ($if-added((aspect-print :frame :value)))
                   ; (always-high(supply)) (gnd (gnd)))
      (trans-struct)   ; all transistors with what part connected
      (trans)          ; all transistors connected
      (gate)           ; all transistors with gate connections
      (src-drn)        ; all transistors with source or drain connections
      (load)           ; all load transistors connected
      (driver)         ; all drivers connected to  node
      (struct)         ; all structures connected to node
      (cap)            ; capacitors connected
      (mos-cap)        ; mos capacitors connected
      (supply)         ; power supplies connected
      (clock)          ; clocks connected
      (gate-cap)       ; capacitance of all gates connected to node (upf)
      (static-cap)     ; capacitance of all capacitors tied to gnd or supply
      (clka-cap)       ; capacitance of node to clka
      (clkb-cap)       ; capacitance of node to clkb
      (clkc-cap)       ; capacitance of node to clkc
      (clkd-cap)       ; capacitance of node to clkd
      (src-drn-cap)    ; capacitance of node to all sources and drains
      (other-cap)      ; capacitance of node to other places
      (total-cap)      ; sum of all capacitances
      (pad))           ; i/o pads connected

(deframe one-port-element
    (ako($value(thing)))
    (node-num)
    (prot-device($if-added((freplace :value 'status '$value 'in-use)))
              ($ask(dont)))) (deframe pad
    (ako($value(one-port-element))))

(deframe two-port-element
     (ako($value(thing)))
     (pos-node)
     (neg-node)
     (e-value))

(deframe active-two-port-element
      (ako($value(two-port-element)))) (deframe passive-two-port-element
      (ako($value(two-port-element)))) (deframe cap
      (ako($value(passive-two-port-element)))) (deframe supply
      (ako($value(active-two-port-element)))) (deframe clock
      (ako($value(active-two-port-element))))

(deframe circuit
      (ako($value(thing))))

(deframe gate
     (ako($value(thing)))
     (beta-ratio)
     (trigger)
```

```
     (status)
 (pull-up ($if-added((freplace :value 'status '$value 'in-use))))
 (pull-down ($if-added((make-status-in-use :value))))
 (in-node)
 (out-node)
 (supply-node)
 (supply)
 (struct($if-added((freplace :value 'status '$value 'in-use))))
 (dummy)) (deframe static-gate
     (ako($value(gate)))) (deframe dyna-gate
     (ako($value(gate))))

(deframe dynamic-gate
     (ako($value(dyna-gate)))
     (instance($if-added((fput :value 'status '$value 'free))))
     (pre-phase)
     (true-phase))

(deframe dynamic-xc-xor
     (ako($value(dyna-gate)))
     (instance($if-added((fput :value 'status '$value 'free))))
     (pre-phase)
     (true phase))

(deframe inverter
     (ako($value(static-gate)))
     (xfer-gate)) (deframe reg-inverter
     (ako($value(inverter)))
     (instance($if-added((fput :value 'status '$value 'free)))))

(deframe xi-inverter
     (ako($value(inverter)))
     (instance($if-added((fput :value 'status '$value 'free))))
     (xfer-gate($if-added((freplace :value 'status '$value 'in-use))))
     (clock-node)
     (clock))

(deframe clko[illegible]nverter
     (ako($value(in[illegible]rter)))
     (instance($if-added((fput :value 'status '$value 'free))))
     (xfer-gate($if-added((freplace :value 'status '$value 'in-use))))
     (clock-node)
     (clkgate($if-added((freplace :value 'status '$value 'in-use))))
     (clock)) (deframe nor-gate
     (ako($value(static-gate)))
     (instance($if-added((fput :value 'status '$value 'free)))))

(deframe nand-gate
     (ako($value(static-gate)))
     (instance($if-added((fput :value 'status '$value 'free)))))

(deframe other-gate
     (ako($value(static-gate)))
     (instance($if-added((fput :value 'status '$value 'free)))))

(deframe static-xc-xor
     (ako($value(static-gate)))
     (instance($if-added((fput :value 'status '$value 'free)))))

(deframe struct
     (ako($value(thing)))
     (status)
     (l-div-w)
     (string-l)              ; length of fet string
```

```
    (node-1($ask(dont)))
 (node-2($ask(dont)))
 (clk-input($ask(dont)))
 (clk-class($ask(dont)))
 (class)
 (clking-check)
 (substruct($if-added((make-status-in-use :value))))
 (trans($if-added((make-status-in-use :value)))))

(deframe parallel-struct
    (ako($value(struct)))
    (instance($if-added((fput :value 'status '$value 'free)))))

(deframe series-struct
    (ako($value(struct)))
    (instance($if-added((fput :value 'status '$value 'free)))))

(deframe super-struct
    (ako($value(struct)))
    (instance($if-added((fput :value 'status '$value 'free)))))

(deframe xc-xor-struct
    (ako($value(struct)))
    (instance($if-added((fput :value 'status '$value 'free)))
    (in-node)
    (out-node))
```

```
;********************FILE FRAMES-2SL ********

(deframe elements
     (ako($value(thing)))
     (dummy)
     (nodes)
     (driver) (reg-driver)(xi-driver)(precharger)(pup-driver)(db1)
             (drn-boot($ask(dont)))
     (load) (src-load)(ckc-load)(drc-load)(oth-load)
     (mos-cap)
     (dynamic-gate)
     (supply)
     (clock)
     (cap)
     (inverter) (simple-inverter)(d-i-inverter)
     (static-gate)
     (super-buffer)
     (reg-core)(reg-cell)
     (drain-bootstrap)
     (struct)
     (series-struct($if-added((fput :frame 'struct '$value :value))))
     (parallel-struct($if-added((fput :frame 'struct '$value :value))))
     (super-struct($if added((fput :frame 'struct '$value :value))))
     (xc-xor-struct($if-added((fput :frame 'struct '$value :value)))))


(deframe elements1
     (ako($value(elements)))
     (dummy($value(1))))

(deframe errors
     (ako($value(thing))))

(deframe inv-errors
     (ako($value(errors)))
     (dummy)
     (beta-ratio($ask(dont)))
     (coupling($ask(dont)))
     (input-clocking($ask(dont))))

(deframe gate-errors        ; filled by forward chain rules
     (ako($value(errors)))
     (nand-length($ask(dont)))
     (beta-ratio($ask(dont)))
     (dynamic-clocking($ask(dont)))
     (dynamic-clocking-1($ask(dont)))(dynamic-clocking-2($ask(dont)))
     (dynamic-clocking-3($ask(dont)))(dynamic-clocking-4($ask(dont)))
     (race-condition($ask(dont))))

(deframe gate-errors-b      ; filled by backward chain rules
     (ako($value(errors)))
     (feedback-desirable($ask(dont)))
     (input-clocking-error($ask(dont))))

(deframe funny fet
     (ako($value(errors)))
     (max-driver-length($ask(dont)))
     (min-driver-width($ask(dont)))
     (min-load-length($ask(dont)))
     (min-load-width($ask(dont)))
     (max-cap-length($ask(dont)))
     (single-connection($ask(dont))))

(deframe funny-node
```

```
     (ako($value(errors)))
  (gate-only($ask(dont)))
  (supply-gate-only($ask(dont)))
  (clock-supply-short($ask(dont)))
  (single-connection($ask(dont)))
  (clocking-flag($ask(dont)))
  (long-rc-flag($ask(dont))))

(deframe coupling-errors
     (ako($value(errors)))
     (xi-driver-coupling($ask(dont)))
     (xi-driver-coupling-1($ask(dont))))

(deframe net-errors
     (ako($value(errors)))
     (clocks($ask(dont))))

(deframe super-buffer-errors
     (ako($value(errors)))
     (power-waste-flag($ask(dont)))
     (aggressive-br-flag($ask(dont)))
     (poor-input-drive($ask(dont))))

(deframe drn-boot-errors
     (ako($value(errors)))
     (phase-hold-down($ask(dont)))
     (clocking-error($ask(dont)))
     (mos-cap-backwards($ask(dont)))
     (boot-node-not-active-low($ask(dont)))
     (longer-driver-needed($ask(dont))))

(deframe register-errors
   (ako($value(errors)))
   (critical-node-flag ($ask(dont)))
   (clocking-error ($ask(dont)))
   (internal-connection ($ask(dont))))

(deframe race-errors
   (ako($value(errors)))
   (precharge-loss($ask(dont)))
   (input-skew-flag($ask(dont))))

(deframe clk-skew-errors
   (ako($value(errors)))
   (clock-skew-flag-1($ask(dont)))
   (clock-skew-flag-2($ask(dont))))

(deframe charge-share-errors
   (ako($value(errors)))
   (feedback glitch-flag($ask(dont)))
   (feedback-glitch-error($ask(dont))))

(deframe input-pad-errors
   (ako($value(errors)))
   (missing protection-device($ask(dont)))
   (undershoot-flag($ask(dont))))
```

Appendix C - Technology Frame

```
;   ********************FILE: TECH-FILE.SL *****

  ; Technology Frame for 5v only NMOS, 2 phase non-overlapping clocks

(deframe *g-con
  (ako($value(thing))) ;;;; transistor constants ;;;;;;;;;;
 (mx-dr-l($value(2.5)))           ; maximum driver length
 (mn-dr-w($value(3)))             ; minimum driver width
 (mn-ld-l($value(3)))             ; minimum load length
 (mn-ld-w($value(3.5)))           ; minimum load width
 (mx-cap-l($value(15)))           ; maximum mos-cap length
 (std-ld-current($value(0.050)))  ; current for load with w/l = 1 (ma)
 (dr-eq-ratio                     ; convert driver l-div-w to load l-div-w
    ($value(3))) ;;;;; gate constants ;;;;;;;;;;;;;;
 (st-nand-sl($value(3)))          ; maximum static nand string length
 (br-src-load($value(4.0)))       ; beta ratio for source connected load
 (br-drc load($value(6.0)))       ; beta ratio for drain  connected load
 (br-ckc-load($value(6.0)))       ; beta ratio for clock connected load
 (br oth-load($value(4.0)))
 (xi-dr-wrf($value(0.9)))         ; xi-driver width reduction factor
 (mn-dr-ga-xf-w($value(3.2)))     ; minimum driver gate-area to xfer-gate
                                  ;       width ;;;;; gate and overlap capacitances ;;;;;;;
 (gox-cap ($value(910e-6)))       ; gate area capacitance (pf/u^ 2)
 (gox-overlap-cap($value(200e-6))) ;gate overlap capacitance (pf/u) ;;;;; dynamic circuit constants
 (noise-tau($value(20)))          ; time constant for noise (ns) ;;;;; super-buffer constants
 (mn-sup-buf-pwr-ratio            ; min ratio of predriver w/l to driver w/l
       ($value(0.25)))
 (mx-sup-buf-pwr-ratio            ; max ratio of predriver w/l to driver w/l
       ($value(0.75)))
 (mx-sup-buf-agg-br               ; max predriver aggressive beta-ratio
     ($value(^(* 0.9 (fvo '*g-con 'br-src load)))))
 (mn-sup-buf agg-br               ; min predriver aggressive beta-ratio
     ($value(^(* 0.8 (fvo '*g-con 'br-src-load))))) ;;;;; charge-share-ratios
 (dr-cshare ratio                 ; (/ l/w of pull down l/w of xfer-gate)
   ($value(4.0)))                 ; couple back at most 1/5 of voltage
 (cap-cshare-ratio                ; (/ cap of driven node cap of couple-node)
   ($value(4.0)))                 ; couple back at most 1/5 of voltage ;;;;; drain-bootstrap constants
 (db-ahd-l($value(2.5)))          ; drain bootstrap active holddown width ;;;;; clock skew sensitivity
 (clk-skew-flag($value(t)))  ; true if clock skew is a problem )
```

Appendix D - RUBICC Examples

Examples of RUBICC critiques are given in this appendix. Each example contains a circuit schematic, the circuit net-list input given to RUBICC, and critique summary.

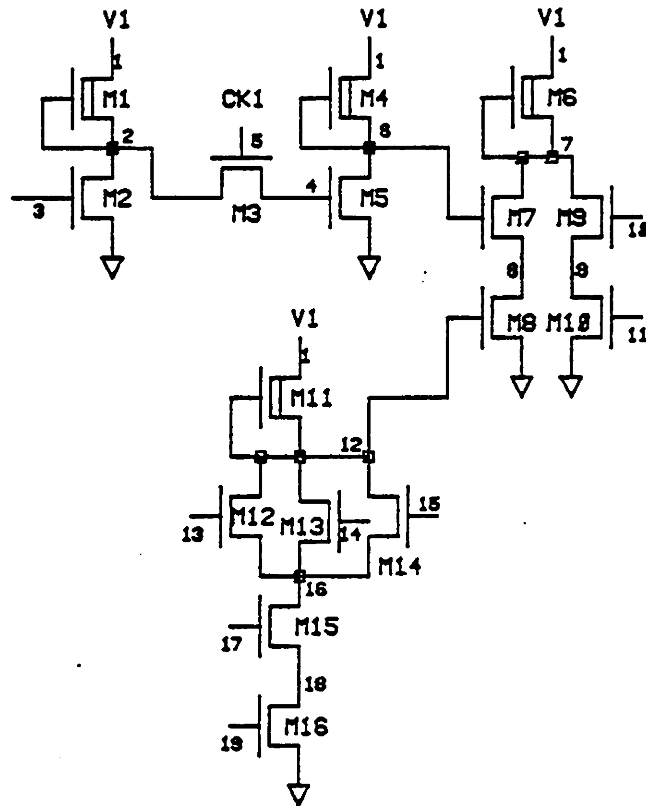

Figure D-1 -- *Net-List-2

Critique:

```
ERRORS FOUND FOR CIRCUIT: *NET-LIST-2
COUPLING-ERRORS
   XI-DRIVER-COUPLING     (M5)
FUNNY-NODE
   GATE-ONLY              (NODE3 NODE5 NODE10 NODE11 NODE13 NODE14 NODE15
NODE17 NODE19)
FUNNY-FET
   MIN-DRIVER-WIDTH       (M2 M5)
   SINGLE-CONNECTION      (M3 M16 M15 M14 M13 M12 M10 M9 M2)
GATE-ERRORS
   BETA-RATIO             (OTHER-GATE-2 OTHER-GATE-1)
INV-ERRORS
   BETA-RATIO             (XI-INVERTER-1 REG-INVERTER-1)

CIRCUITS and GATES IDENTIFIED:
  OTHER-GATE-2  (M11 M14 M12 M13 M15 M16)
  OTHER-GATE-1  (M6 M9 M10 M8 M7)
  XI-INVERTER-1  (M4 M5 M3)
  REG-INVERTER-1  (M1 M2)

FREE TRANSISTORS:
   NIL
garbage collection time = 0.95583 min
total run time= 5.91583 min
```

Circuit Input List:

```
(setq *net-list-2 '((*net-list-2)
   (load m1 1 2 2 4 8)
   (driver m2 2 3 0 2 2)
   (driver m3 2 5 4 4 2)
   (load m4 1 6 6 4 8)
   (driver m5 6 4 0 2 2)
   (load m6 1 7 7 4 8)
   (driver m7 7 6 8 6 2)
   (driver m8 8 12 0 6 2)
   (driver m9 7 10 9 6 2)
   (driver m10 9 11 0 6 2)
   (load m11 1 12 12 4 8)
   (driver m12 12 13 16 6 2)
   (driver m13 12 14 16 6 2)
   (driver m14 12 15 16 6 2)
   (driver m15 16 17 18 6 2)
   (driver m16 18 19 0 6 2)
   (supply v1 1 0 5)
))
```

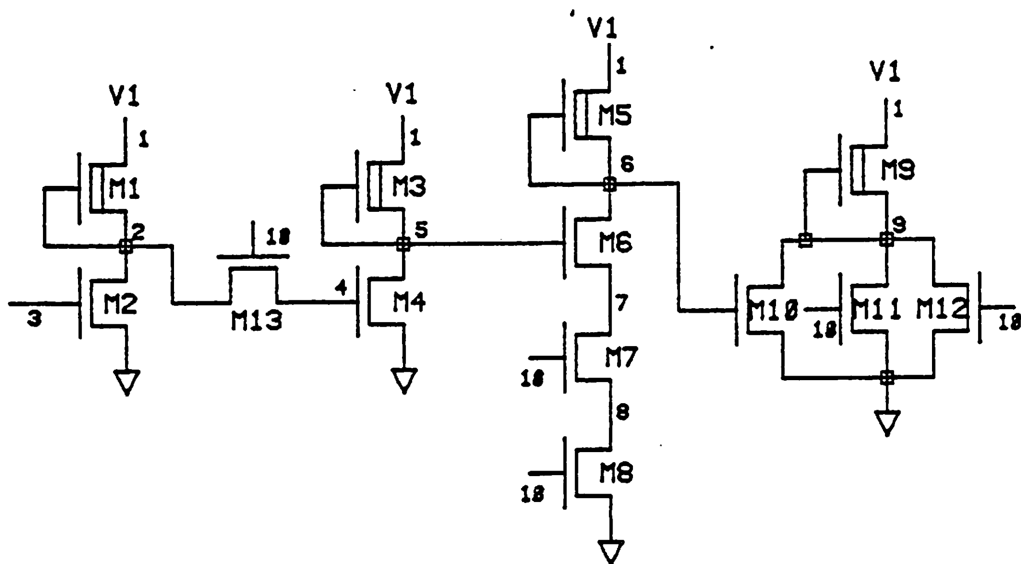

Figure D-2 – *Net-List-5

Critique:

```
ERRORS FOUND FOR CIRCUIT: *NET-LIST-5
COUPLING-ERRORS
    XI-DRIVER-COUPLING      (M4)
FUNNY-NODE
    GATE-ONLY               (NODE3 NODE10)
FUNNY-FET
    MIN-DRIVER-WIDTH        (M2 M4)
    SINGLE-CONNECTION       (M2)
GATE-ERRORS
    BETA-RATIO              (NAND-GATE-1)
INV-ERRORS
    BETA RATIO              (XI-INVERTER-1 REG-INVERTER-1)

CIRCUITS and GATES IDENTIFIED:
  NAND-GATE-1  (M5 M6 M7 M8)
  NOR-GATE-1  (M9 M12 M10 M11)
  XI-INVERTER-1  (M3 M4 M13)
  REG-INVERTER-1  (M1 M2)

FREE TRANSISTORS:
    NIL
garbage collection time = 0.44933 min
total run time= 3.37683 min
```

Circuit Input List:

```
(setq *net-list-5 '((*net-list-5)
    (load m1 1 2 2 4 8)
    (driver m2 2 3 0 2 2)
    (supply v1 1 0 5)
    (driver m13 2 10 4 4 2)
    (load m3 1 5 5 4 8)
    (driver m4 5 4 0 2 2)
    (load m5 1 6 6 4 8)
    (driver m6 6 5 7 6 2)
    (driver m7 7 10 8 6 2)
    (driver m8 8 10 0 6 2)
    (load m9 1 9 9 4 8)
    (driver m10 9 6 0 6 2)
    (driver m11 9 10 0 6 2)
    (driver m12 9 10 0 6 2)
))
```

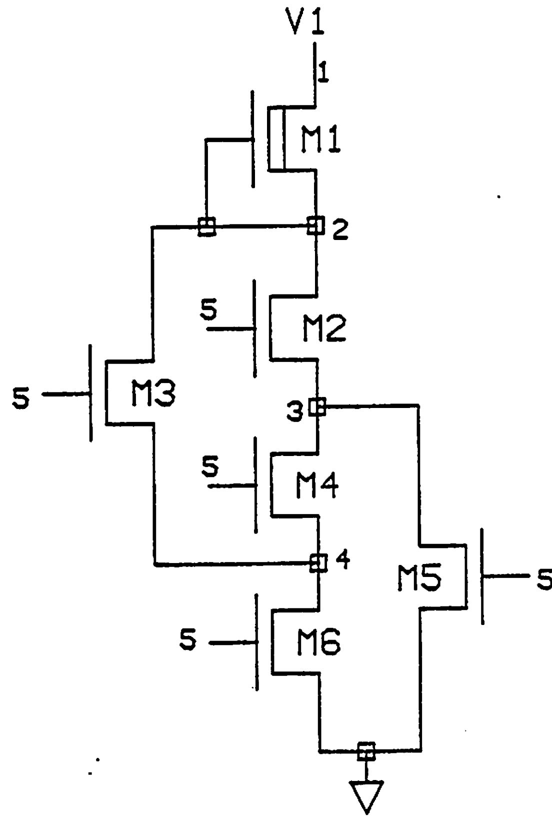

Figure D-3 — Net-. [illegible] -8

Critique:

ERRORS FOUND FOR CIRCUIT: *NET-LIST-8
FUNNY-NODE
GATE-ONLY (NODE5)

CIRCUITS and GATES IDENTIFIED:

FREE TRANSISTORS:
(M1 M6 M5 M4 M3 M2)
garbage collection time = 0.22767 min
total run time= 1.09267 min

Circuit Input List:

```
(setq *net-list-8 '((*net-list-8)
    (load m1 1 2 2 4 6)
    (supply v1 1 0 5)
    (driver m2 2 5 3 6 2)
    (driver m3 2 5 4 6 2)
    (driver m4 3 5 4 6 2)
    (driver m5 3 5 0 6 2)
    (driver m6 4 5 0 6 2)))
```

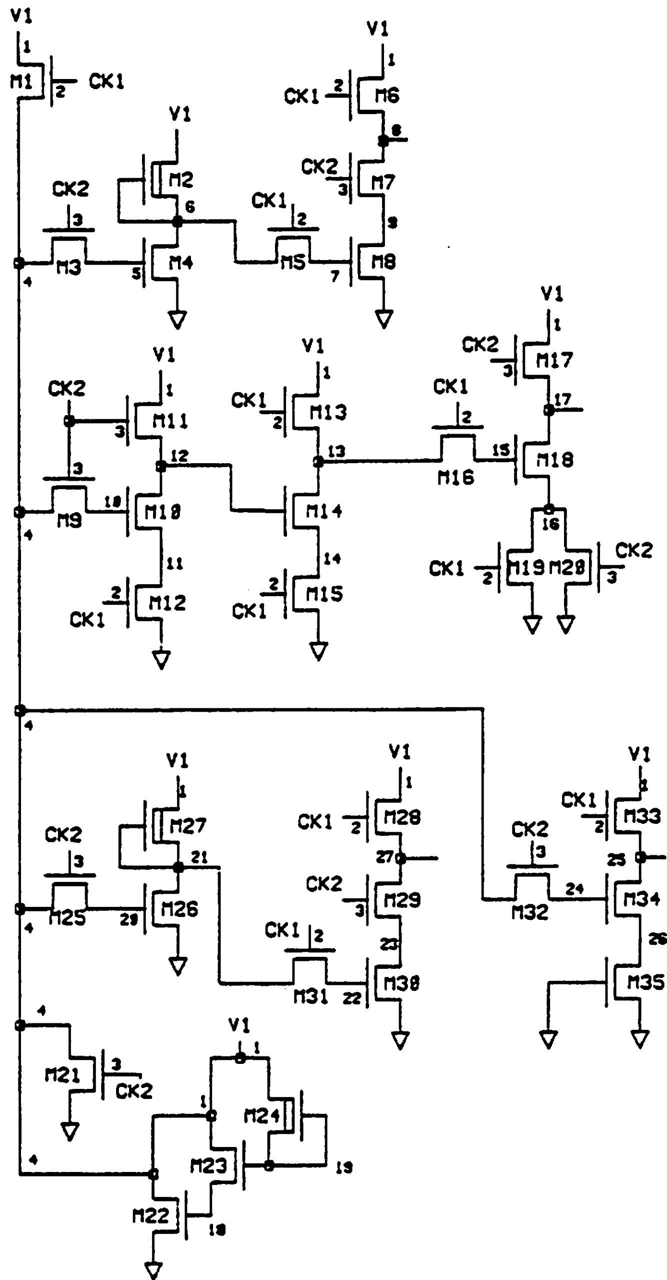

Figure D-4 -- Net-List-14

Critique:

```
ERRORS FOUND FOR CIRCUIT: *NET-LIST-14-DYNAMIC-CLOCKING
COUPLING-ERRORS
   XI-DRIVER-COUPLING      (M22)
   XI-DRIVER-COUPLING-1    (M10 M18 M34)
FUNNY-NODE
   CLOCKING-FLAG        (NODE4)
GATE-ERRORS
   DYNAMIC-CLOCKING-1     (DYNAMIC-GATE-7 DYNAMIC-GATE-4 DYNAMIC-GATE-1)
   DYNAMIC-CLOCKING-2     (DYNAMIC-GATE-7)
   DYNAMIC-CLOCKING-4     (DYNAMIC-GATE-6)

CIRCUITS and GATES IDENTIFIED:
  DYNAMIC-GATE-7  (M1 M22 M21)
  DYNAMIC-GATE-6  (M33 M34 M35)
  DYNAMIC-GATE-5  (M28 M29 M30)
  DYNAMIC-GATE-4  (M13 M14 M15)
  DYNAMIC-GATE-3  (M11 M10 M12)
  DYNAMIC-GATE-2  (M6 M7 M8)
  DYNAMIC-GATE-1  (M17 M18 M20 M19)
  XI-INVERTER-2  (M27 M26 M25)
  XI-INVERTER-1  (M2 M4 M3)

FREE TRANSISTORS:
   (M24)
garbage collection time = 4.116 min
total run time= 20.88633 min
```

Circuit Input List:

```
(setq *net-list-14
'((*net-list-14-dynamic-clocking)
(supply v1 1 0 5)              (clock ck1 2 0 5)              (clock ck2 3 0 5)
(driver m1 1 2 4 10 2)         (load m2 1 6 6 4 8)            (driver m3 4 3 5 4 2)
(driver m4 6 5 0 10 2)         (driver m5 6 2 7 4 2)          (driver m6 1 2 8 10 2)
(driver m7 8 3 9 6 2)          (driver m8 9 7 0 10 2)         (driver m9 4 3 10 4 2)
(driver m10 12 10 11 10 2)     (driver m11 1 3 12 6 2)        (driver m12 11 2 0 6 2)
(driver m13 1 2 13 6 2)        (driver m14 13 12 14 6 2)      driver m15 14 2 0 6 2)
(driver m16 13 2 15 4 2)       (driver m17 1 3 17 6 2)        (driver m18 17 15 16 12 2)
(driver m19 16 2 0 6 2)        (driver m20 16 3 0 6 2)        (driver m21 4 3 0 6 2)
(driver m22 4 18 0 6 2)        (driver m23 1 19 18 6 2)       (load m24 1 19 19 4 8)
(driver m25 4 3 20 4 2)        (driver m26 21 20 0 16 2)      (load m27 1 21 21 4 8)
(driver m28 1 2 27 6 2)        (driver m29 27 3 23 6 2)       (driver m30 23 22 0 16 2)
(driver m31 21 2 22 4 2)       (driver m32 4 3 24 4 2)        (driver m33 1 2 25 6 2)
(driver m34 25 24 26 16 2)     (driver m35 26 0 0 6 2)))
```

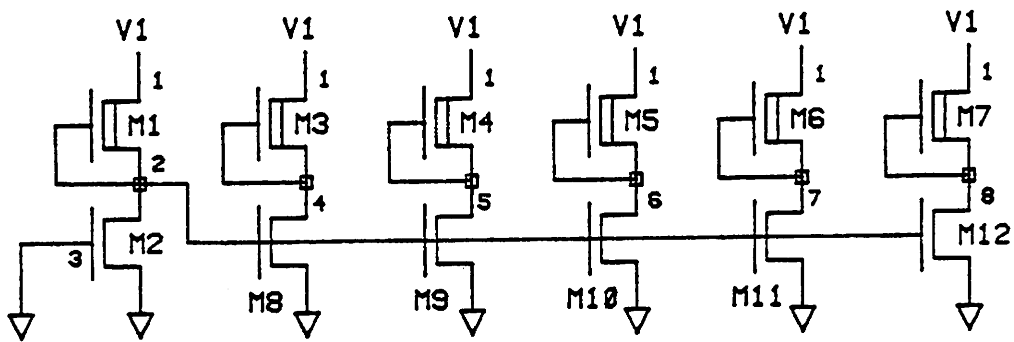

Figure D-5 -- Net-List-17

Critique:

```
ERRORS FOUND FOR CIRCUIT: *NET-LIST-17
FUNNY-NODE
    GATE-ONLY           (NODE3)
    LONG-RC-FLAG         (NODE2)
FUNNY-FET
    SINGLE-CONNECTION      (M2)

CIRCUITS and GATES IDENTIFIED:
  REG-INVERTER-6  (M7 M12)
  REG-INVERTER-5  (M6 M11)
  REG-INVERTER-4  (M5 M10)
  REG-INVERTER-3  (M4 M9)
  REG-INVERTER-2  (M3 M8)
  REG-INVERTER-1  (M1 M2)

FREE TRANSISTORS:
    NIL
garbage collection time = 0.25783 min
total run time= 5.37133 min
NIL
```

Circuit Input List:

```
(setq *net-list-17 '((*net-list-17)
   (supply v1 1 0 5)
   (load m1 1 2 2 4 30)
   (driver m2 2 3 0 6 2)
   (load m3 1 5 5 4 6)
   (load m4 1 6 6 4 6)
   (load m5 1 7 7 4 6)
   (load m6 1 8 8 4 6)
   (load m7 1 9 9 4 6)
   (driver m8 5 2 0 15 2)
   (driver m9 6 2 0 15 2)
   (driver m10 7 2 0 15 2)
   (driver m11 8 2 0 15 2)
   (driver m12 9 2 0 15 2)
))
```

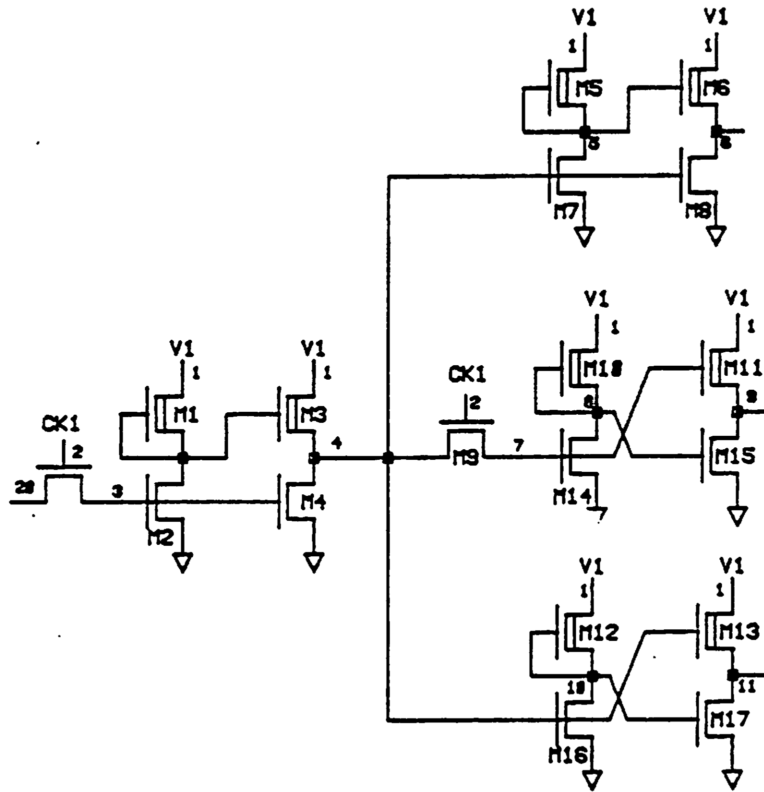

Figure D-6 – *Net-List-18

Critique:

```
ERRORS FOUND FOR CIRCUIT: *NET-LIST-18--SUPER-BUFFERS
SUPER-BUFFER-ERRORS
   POWER-WASTE-FLAG        (SUPER-BUFFER-4)
   AGGRESSIVE-BR-FLAG      (SUPER-BUFFER-4)
   POOR-INPUT-DRIVE        (SUPER-BUFFER-1)
COUPLING-ERRORS
   XI-DRIVER-COUPLING      (M14)
FUNNY-FET
   SINGLE-CONNECTION       (M18)
INV-ERRORS
   BETA-RATIO              (XI-INVERTER-2 XI-INVERTER-1)

CIRCUITS and GATES IDENTIFIED:
 SUPER-BUFFER-4  (REG-INVERTER-1 REG-INVERTER-2)
 SUPER-BUFFER-3  (REG-INVERTER-4 REG-INVERTER-5)
 SUPER-BUFFER-2  (XI-INVERTER-1 XI-INVERTER-2)
 SUPER-BUFFER-1  (XI-INVERTER-3 REG-INVERTER-3)
 XI-INVERTER-3  (M10 M14 M9)
 XI-INVERTER-2  (M2 M4 M18)
 XI-INVERTER-1  (M1 M3 M18)
 REG-INVERTER-5  (M13 M17)
 REG-INVERTER-4  (M12 M16)
 REG-INVERTER-3  (M11 M15)
 REG-INVERTER-2  (M6 M8)
 REG-INVERTER-1  (M5 M7)

FREE TRANSISTORS:
   NIL
garbage collection time = 0.68933 min
total run time= 3.71067 min
```

Circuit Input List:

```
(setq *net-list-18 '((*net-list-18--super-buffers)
  (supply v1 1 0 5)
  (clock ck1 2 0 5 )
  (load m1 1 12 12 6 4)
  (load m2 1 12 4 10 4)
  (driver m3 12 3 0 10 2)
  (driver m4 4 3 0 20 2)
  (load m5 1 5 5 4 8)
  (load m6 1 5 6 4 8)
  (driver m7 5 4 0 6 2)
  (driver m8 6 4 0 6 2)
  (driver m9 4 2 7 10 2)
  (load m10 1 8 8 4 8)
  (load m11 1 7 9 4 8)
  (load m12 1 10 10 4 8)
  (load m13 1 4 11 4 8)
  (driver m14 8 7 0 6 2)
  (driver m15 9 8 0 6 2)
  (driver m16 10 4 0 6 2)
  (driver m17 11 10 0 6 2)
  (driver m18 3 2 20 4 2)
))
```

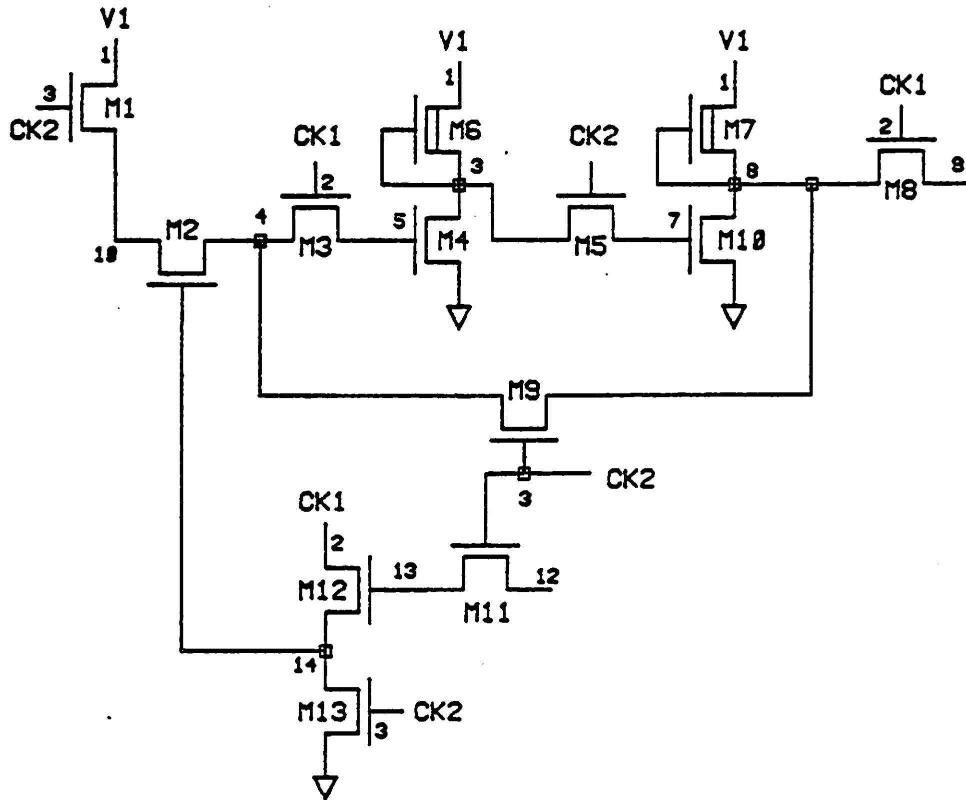

Figure D-7 -- Net-List-22

Critique:

```
ERRORS FOUND FOR CIRCUIT: *NET-LIST-22--REGISTER-CELL
CLK-SKEW-ERRORS
   CLOCK-SKEW-FLAG-2        ((M2 M3))
REGISTER-ERRORS
   CRITICAL-NODE-FLAG       (REG-CELL-1)
DRN-BOOT-ERRORS
   BOOT-NODE-NOT-ACTIVE-LOW (DRAIN-BOOTSTRAP-1)
FUNNY-FET
   SINGLE-CONNECTION        (M11 M8)

CIRCUITS and GATES IDENTIFIED:
 DRAIN-BOOTSTRAP-1  (M11 M12 M13)
 REG-CELL-1  (REG-CORE-1 M2 M8)
 REG-CORE-1  (XI INVERTER-1 XI INVERTER-2 M9)
 XI-INVERTER 2  (M7 M10 M5)
 XI-INVERTER-1  (M6 M4 M3)

FREE TRANSISTORS:
   (M1)
garbage collection time = 0.70667 min
total run time= 3.9335 min
```

Circuit Input List:

```
(setq *net-list 22 '((*net-list-22--register-cell)
  (supply v1 1 0 5)
  (clock ck1 2 0 5)
  (clock ck2 3 0 5)
  (driver m1 1 3 10 10 2)
  (driver m2 10 14 4 6 2)
  (driver m3 4 2 5 6 2)
  (driver m4 6 5 0 10 2)
  (driver m5 7 3 6 4 2)
  (load m6 1 6 6 4 8)
  (load m7 1 8 8 4 8)
  (driver m8 8 14 9 6 2)
  (driver m9 8 3 4 6 2)
  (driver m10 8 7 0 10 2)
  (driver m11 12 3 13 6 2)
  (driver m12 2 13 14 20 2)
  (driver m13 14 3 0 10 2)
))
```

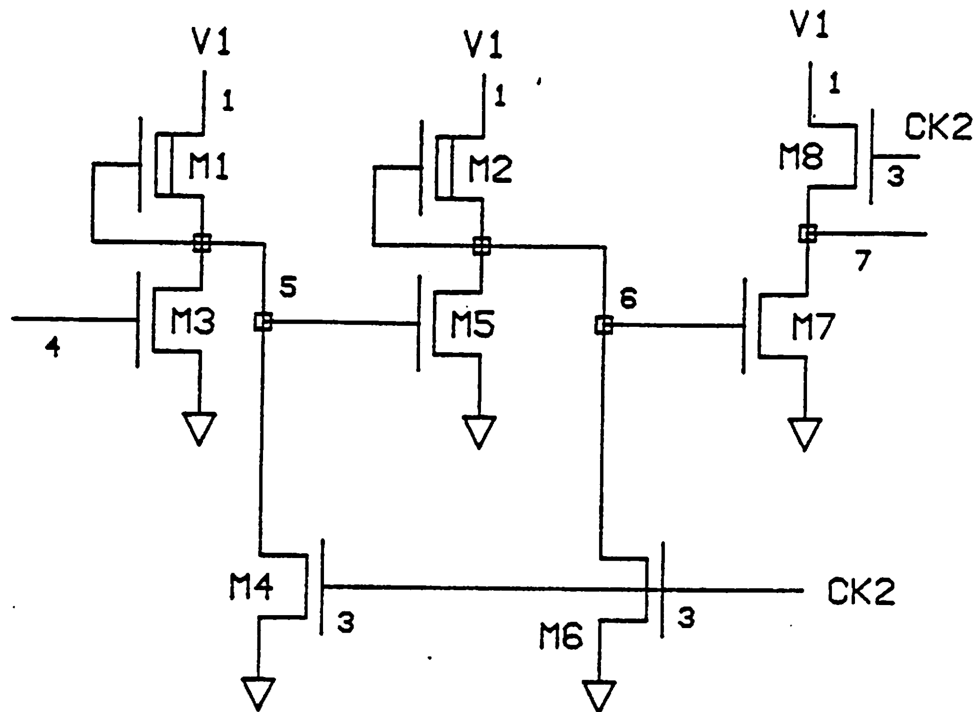

Figure D-8 -- Net-List-23

Critique:

```
ERRORS FOUND FOR CIRCUIT: *NET-LIST-23--RACE-CONDITION
RACE-ERRORS
   PRECHARGE-LOSS        (DYNAMIC-GATE-1)
FUNNY-NODE
   GATE-ONLY             (NODE4)
   SINGLE-CONNECTION     (NODE2)
FUNNY-FET
   SINGLE-CONNECTION     (M3)

CIRCUITS and GATES IDENTIFIED:
 DYNAMIC-GATE-1  (M8 M7)
 NOR-GATE-2  (M2 M6 M5)
 NOR-GATE-1  (M1 M4 M3)

FREE TRANSISTORS:
   NIL
garbage collection time = 0.4755 min
total run time= 2.53867 min
```

Circuit Input List:

```
(setq *net-list-23 '((*net-list-23--race-condition)
  (supply v1 1 0 5)
  (clock ck1 2 0 5)
  (clock ck2 3 0 5)
  (driver m3 5 4 0 10 2)
  (driver m4 5 3 0 6 2)
  (driver m5 6 5 0 6 2)
  (driver m6 6 3 0 6 2)
  (driver m7 7 6 0 6 2)
  (driver m8 1 3 7 10 2)
  (load m1 1 5 5 4 8)
  (load m2 1 6 6 4 8)
))
```

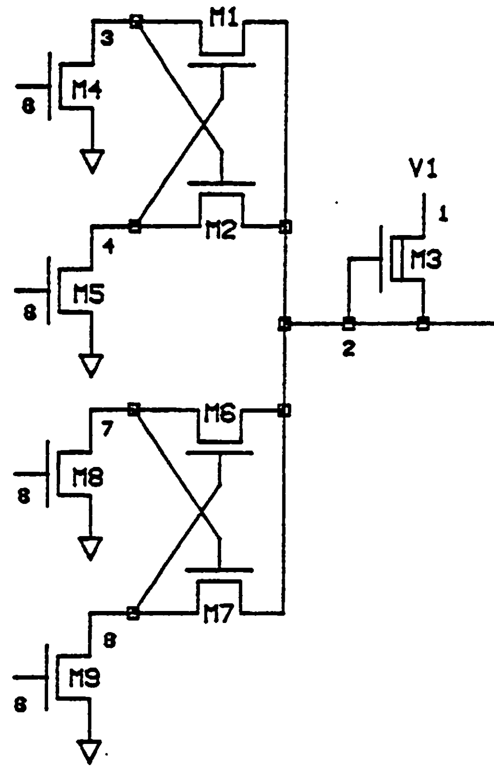

Figure D-9 -- Net-List-24

Critique:

```
ERRORS FOUND FOR CIRCUIT: *NET-LIST-24--XC-XOR-GATES
FUNNY-NODE
   GATE-ONLY            (NODE6)
   SINGLE-CONNECTION       (NODE10)
GATE-ERRORS
   BETA-RATIO           (STATIC-XC-XOR-1)

CIRCUITS and GATES IDENTIFIED:
 STATIC-XC-XOR-1  (M3 M6 M7 M1 M2)

FREE TRANSISTORS:
   (M9 M8 M5 M4)
garbage collection time = 0.46267 min
total run time= 2.65583 min
```

Circuit Input List:

```
(setq *net-list-24 '((*net-list-24--xc-xor-gates)
  (supply v1 1 0 5)
  (clock ck1 10 0 5)
  (driver m1 3 4 2 3 2)
  (driver m2 2 3 4 6 2)
  (load m3 1 2 2 4 8)
  (driver m4 3 6 0 6 2)
  (driver m5 4 6 0 6 2)
  (driver m6 7 8 2 6 2)
  (driver m7 8 7 2 6 2)
  (driver m8 7 6 0 6 2)
  (driver m9 8 6 0 6 2)
))
```

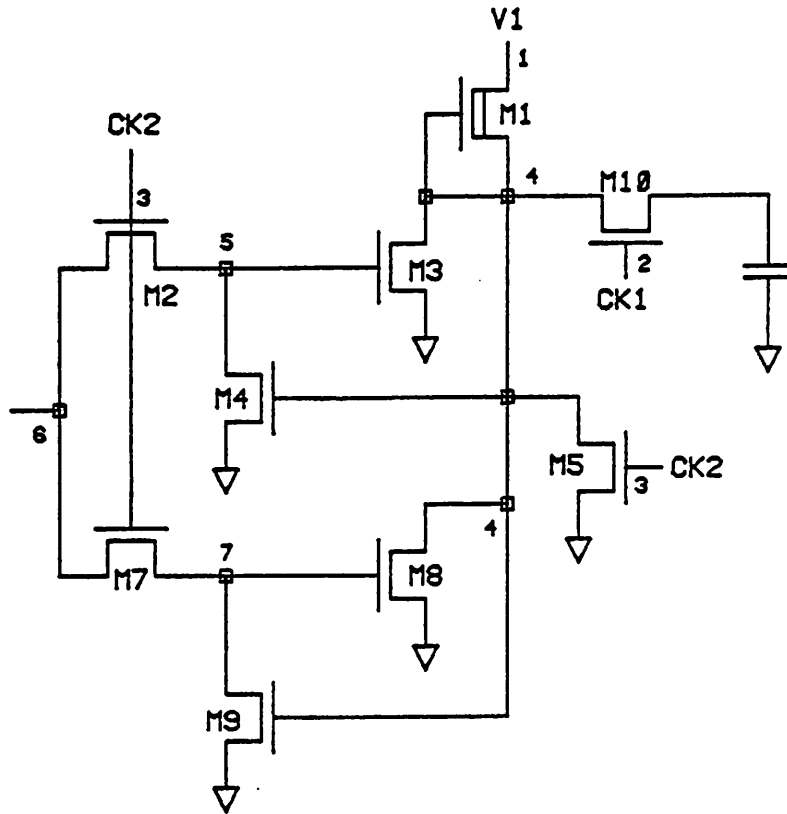

Figure D-10 – Net-List-25

Critique:

```
ERRORS FOUND FOR CIRCUIT: *NET-LIST-25--FEEDBACK
CHARGE-SHARE-ERRORS
    FEEDBACK-GLITCH-FLAG     ((NODE4 NODE8))
    FEEDBACK-GLITCH-ERROR    ((NODE4 NODE8))
FUNNY-FET
    SINGLE-CONNECTION        (M10)

CIRCUITS and GATES IDENTIFIED:
  NOR-GATE-1  (M1 M8 M3 M5)

FREE TRANSISTORS:
    (M10)
garbage collection time = 0.4635 min
total run time= 2.37583 min
```

Circuit Input List:

```
(setq *net-list-25 '((*net-list-25--feedback)
  (supply v1 1 0 5)
  (clock ck1 2 0 5)
  (clock ck2 3 0 5)
  (load m1 1 4 4 4 8)
  (driver m2 6 3 5 4 2)
  (driver m3 4 5 0 10 2)
  (driver m4 5 4 0 3 2)
  (driver m5 4 3 0 6 2)
  (driver m7 6 3 7 4 2)
  (driver m8 4 7 0 10 2)
  (driver m9 7 4 0 4 2)
  (driver m10 4 2 8 10 2)
  (cap c1 8 0 1)
))
```

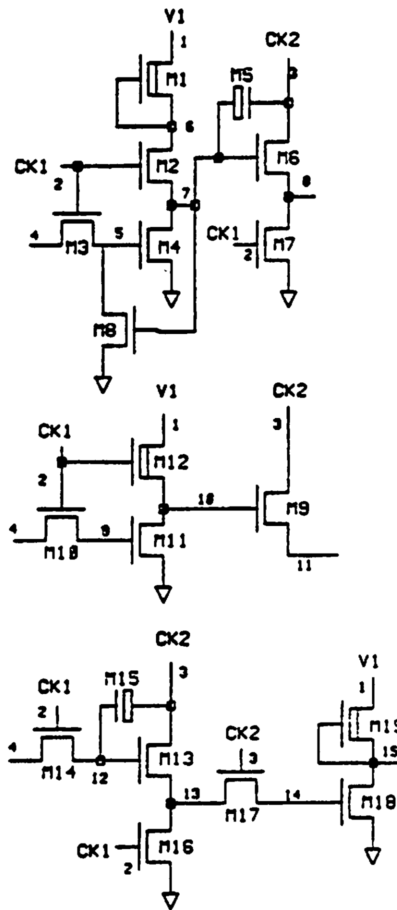

Figure D-11 – Net-List-27

Explanation: RUBICC mentions a glitch error and flag involving node6 and node7. After thinking about this for a few minutes it became apparent that what's happening here is that node7 could still be high when transfer gate "m3" turns on during clk1. Then the voltage at node5 is determined by the ratio of drivers "m3" and "m7" rather than just "m3". The proper connection to the gate of "m8" would be to node8.

Critique:

```
ERRORS FOUND FOR CIRCUIT: *NET-LIST-27--BOOTSTRAPS-WITH-CLOCK-SKEW-ERRORS
CHARGE-SHARE-ERRORS
   FEEDBACK-GLITCH-FLAG     ((NODE7 NODE6) (NODE5 NODE4))
   FEEDBACK-GLITCH-ERROR    ((NODE7 NODE6))
CLK-SKEW-ERRORS
   CLOCK-SKEW-FLAG-1        ((M13 M17))
DRN-BOOT-ERRORS
   PHASE-HOLD-DOWN          (M9)
   CLOCKING-ERROR           (DRAIN-BOOTSTRAP-1)
   MOS-CAP-BACKWARDS        (DRAIN-BOOTSTRAP-2)
   BOOT-NODE-NOT-ACTIVE LOW (DRAIN-BOOTSTRAP-1)
   LONGER-DRIVER-NEEDED     (M4 M11)
COUPLING-ERRORS
   XI-DRIVER-COUPLING       (M8)
FUNNY-FET
   MIN-DRIVER-WIDTH         (M8)
   SINGLE-CONNECTION        (M9)

CIRCUITS and GATES IDENTIFIED:
  DRAIN-BOOTSTRAP-3  (XI-INVERTER 1 M9)
  DRAIN-BOOTSTRAP-2  (CLKOUT-INVERTER-1 M6 M5 M7)
  DRAIN-BOOTSTRAP-1  (M14 M13 M15 M16)
  CLKOUT-INVERTER-1  (M1 M4 M3)
  XI-INVERTER-2  (M19 M18 M17)
  XI-INVERTER-1  (M12 M11 M10)

FREE TRANSISTORS:
   (M8)
garbage collection time = 0.96133 min
total run time= 5.02017 min
```

Circuit Input List:

```
(setq *net-list-27 '((*net-list-27--bootstraps-with-clock s[illegible]w-errors)
  (supply v1 1 0 5)
  (clock ck1 2 0 5)
  (clock ck2 3 0 5)
  (load m1 1 6 6 6 4)
  (driver m2 6 2 7 6 2)
  (driver m3 4 2 5 6 2)
  (driver m4 7 5 0 12 2)
  (driver m5 7 3 7 4 6)
  (driver m6 3 7 8 20 2)
  (driver m7 8 2 0 4 2)
  (driver m8 5 7 0 3 2)
  (driver m9 3 10 11 20 2)
  (driver m10 4 2 9 6 2)
  (driver m11 10 9 0 10 2)
  (load m12 1 2 10 4 8)
  (driver m13 3 12 13 20 2)
  (driver m14 4 3 12 6 2)
  (driver m15 3 12 3 4 8)
  (driver m16 13 2 0 10 2)
  (driver m17 13 3 14 6 2)
  (driver m18 15 14 0 10 2)
  (load m19 1 15 15 4 6)
))
```

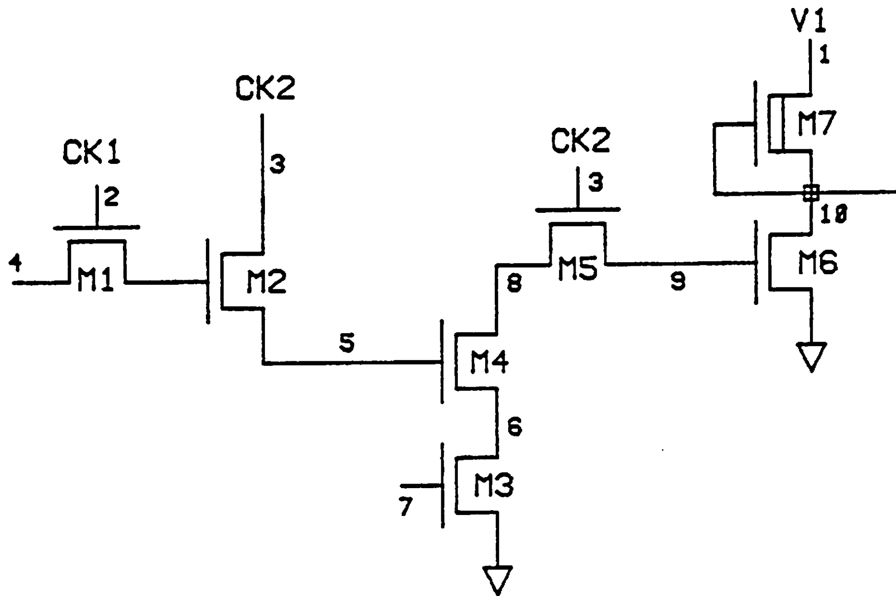

Figure D-12 -- Net-List-28

Critique:

```
ERRORS FOUND FOR CIRCUIT: *NET-LIST-28--CLOCK-SKEW-ERROR
CLK-SKEW-ERRORS
   CLOCK-SKEW-FLAG-2       ((M4 M5))
DRN-BOOT-ERRORS
   PHASE-HOLD-DOWN         (M2)
FUNNY-NODE
   GATE-ONLY           (NODE7)
FUNNY-FET
   SINGLE-CONNECTION       (M1 M3)

CIRCUITS and GATES IDENTIFIED:
 XI-INVERTER-1   (M7 M6 M5)

FREE TRANSISTORS:
   (M3 M4 M1)
garbage collection time  =  0.48367 min
total run time=  2.688 min
```

Circuit Input List:

```
(setq *net-list-28 '((*net-list-28--clock-skew-error)
  (supply v1 1 0 5)
  (clock ck1 2 0 5)
  (clock ck2 3 0 5)
  (driver m1 4 2 11 4 2)
  (driver m2 3 11 5 20 2)
  (driver m3 6 7 0 6 2)
  (driver m4 8 5 6 4 2)
  (driver m5 9 3 8 6 2)
  (driver m6 10 9 0 10 2)
  (load m7 1 10 10 4 6)
))
```

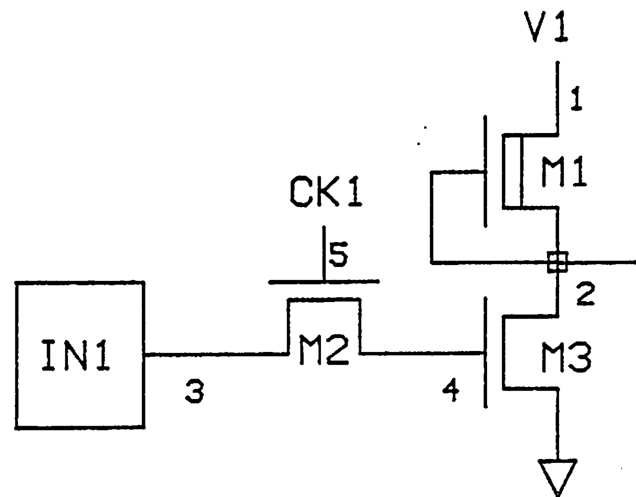

Figure D-13 – Net-List-29

Critique:

```
ERRORS FOUND FOR CIRCUIT: *NET-LIST-29--INPUT-PAD
INPUT-PAD-ERRORS
    MISSING-PROTECTION-DEVICE(IN1-PAD)
    UNDERSHOOT-FLAG        (IN1-PAD)
FUNNY-FET
    SINGLE-CONNECTION      (M2)

CIRCUITS and GATES IDENTIFIED:
  XI-INVERTER-1  (M1 M3 M2)

FREE TRANSISTORS:
    NIL
garbage collection time = 0.00000E+000 min
total run time= 0.63617 min
```

Circuit Input List:

```
(setq *net-list-29 '((*net-list-29--input-pad)
   (supply v1 1 0 5)
   (pad in1 3)
   (load m1 1 2 2 4 8)
   (driver m3 2 4 0 10 2)
   (driver m2 3 5 4 4 2)
   (clock ck1 5 0 5)
))
```

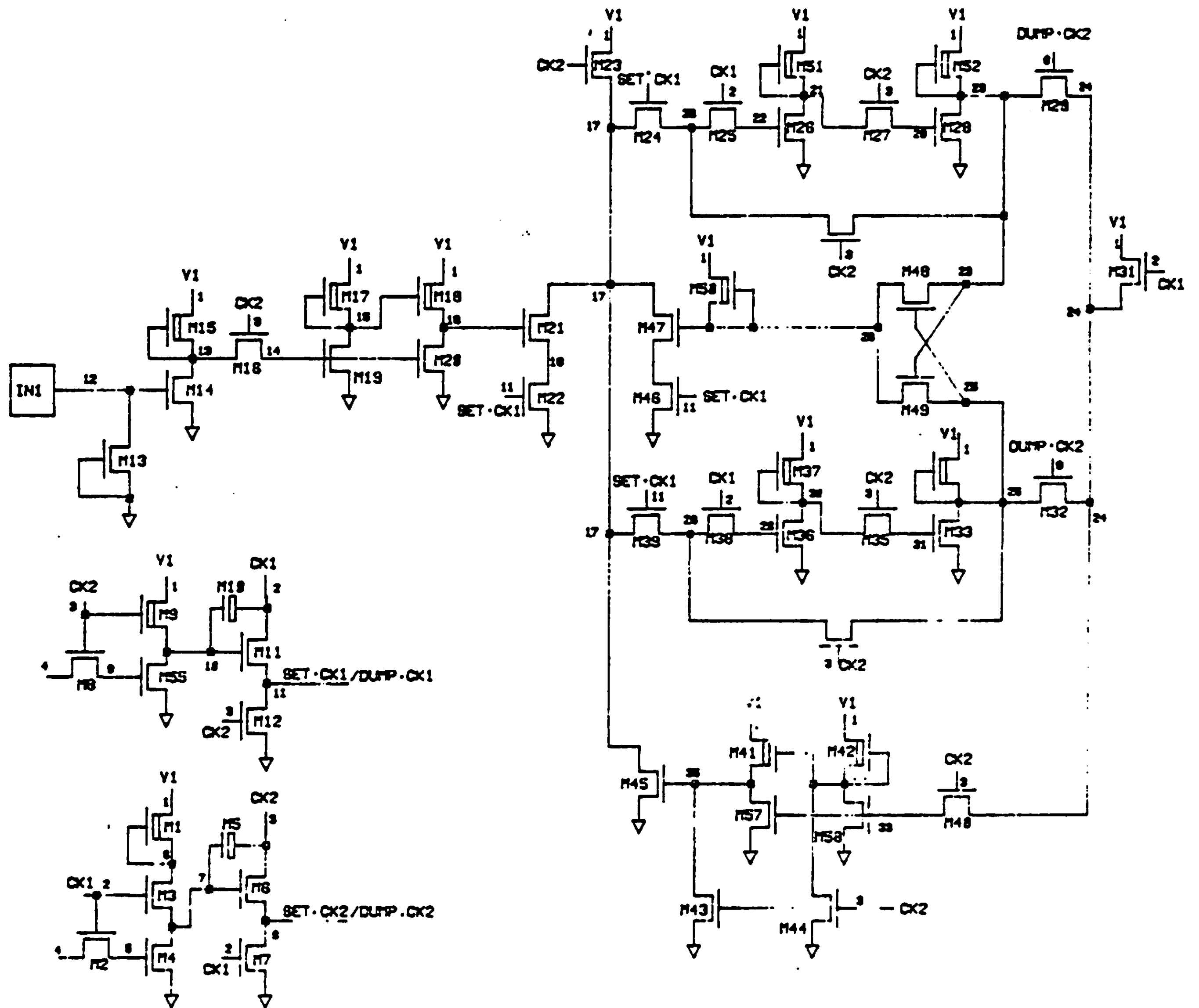

Figure D-14 -- Net-List-30

Critique:

```
ERRORS FOUND FOR CIRCUIT: *NET-LIST-30--EXAMPLE-1
CLK-SKEW-ERRORS
   CLOCK-SKEW-FLAG-2        ((M24 M25) (M39 M38))
REGISTER-ERRORS
   CRITICAL-NODE-FLAG       (REG-CELL-1 REG-CELL-2)
   CLOCKING-ERROR           (REG-CELL-1 REG-CELL-2)
DRN-BOOT-ERRORS
   LONGER-DRIVER-NEEDED     (M4 M55)
SUPER-BUFFER-ERRORS
   AGGRESSIVE-BR-FLAG       (SUPER-BUFFER-2 SUPER-BUFFER-1)
COUPLING-ERRORS
   XI-DRIVER-COUPLING       (M36 M57 M58)
GATE-ERRORS-B
   FEEDBACK-DESIRABLE       (NOR-GATE-2)
GATE-ERRORS
   DYNAMIC-CLOCKING-1       (DYNAMIC-GATE-1)

CIRCUITS and GATES IDENTIFIED:
  DRAIN-BOOTSTRAP-2  (XI-INVERTER-1 M11 M10 M12)
  DRAIN-BOOTSTRAP-1  (CLKOUT-INVERTER-1 M6 M5 M7)
  REG-CELL-2  (REG-CORE-1 M39 M32)
  REG-CELL-1  (REG-CORE-2 M24 M29)
  REG-CORE-2  (XI-INVERTER-4 XI-INVERTER-5 M30)
  REG-CORE-1  (XI-INVERTER 7 XI-INVERTER-6 M56)
Interrupt
  SUPER-BUFFER-1  (NOR-GATE-2 NOR-GATE-1)
  DYNAMIC-GATE-1  (M23 M45 M22 M21 M47 M46)
  STATIC-XC-XOR-1  (M50 M48 M49)
  NOR-GATE-2  (M42 M58 M44)
  NOR-GATE-1  (M41 M57 M43)
  CLKOUT-INVERTER-1  (M1 M4 M2)
  XI-INVERTER-7  (M37 M36 M38)
  XI-INVERTER-6  (M34 M33 M35)
  XI-INVERTER-5  (M52 M28 M27)
  XI-INVERTER-4  (M51 M26 M25)
  XI-INVERTER-3  (M18 M20 M16)
  XI-INVERTER-2  (M17 M19 M16)
  XI-INVERTER-1  (M9 M55 M8)
  REG-INVERTER-1 -- (M15 M14)

FREE TRANSISTORS:
   (M31)
garbage collection time = 12.3875 min
total run time= 51.65183 min
```

Circuit Input List:

```
(setq *net-list-30
'((*net-list-30--example-1)
(supply v1 1 0 5)            (clock ck1 2 0 5)              (clock ck2 3 0 5)
(pad in1 12)                 (load m1 1 6 6 6 4)            (driver m2 4 2 5 6 2)
(driver m3 6 2 7 8 2)        (driver m4 7 5 0 10 2)         (driver m5 3 7 3 4 10)
(driver m6 3 7 8 20 2)       (driver m7 8 2 0 6 2)          (driver m8 4 3 9 4 2)
(load m9 1 3 10 6 4)         (driver m10 2 10 2 4 10)       (driver m11 2 10 11 20 2)
(driver m12 11 3 0 6 2)      (driver m55 10 9 0 20 2)       (driver m13 12 0 0 10 2)
(driver m14 13 12 0 10 2)    (load m15 1 13 13 4 8)         (driver m16 13 3 14 6 2)
(load m17 1 15 15 6 6)       (load m18 1 15 16 8 6)         (driver m19 15 14 0 12 2)
(driver m20 16 14 0 16 2)    (driver m21 17 16 18 10 2)     (driver m22 18 11 0 10 2)
(driver m23 1 3 17 20 2)     (driver m24 17 11 36 6 2)      (driver m25 36 2 22 6 2)
(driver m26 21 22 0 10 2)    (load m51 1 21 21 4 8)         (load m52 1 23 23 4 8)
(driver m27 21 3 20 6 2)     (driver m28 23 20 0 12 2)      (driver m29 23 8 24 12 2)
(driver m30 23 3 36 6 2)     (driver m31 1 2 24 20 2)       (driver m32 24 8 25 10 2)
(driver m33 25 31 0 10 2)    (load m34 1 25 25 4 6)         (driver m35 31 3 30 6 2)
(driver m36 30 29 0 6 2)     (load m37 1 30 30 4 6)         (driver m56 25 3 28 6 2)
(driver m38 29 2 28 6 2)     (driver m39 17 11 28 6 2)      (driver m40 24 3 33 10 2)
(load m41 1 34 35 6 6)       (load m42 1 34 34 4 6)         (driver m43 35 3 0 12 2)
(driver m44 34 3 0 10 2)     (driver m57 35 33 0 12 2)      (driver m58 34 33 0 10 2)
(driver m45 17 35 0 20 2)    (driver m46 27 11 0 10 2)      (driver m47 17 26 27 10 2)
(driver m48 26 25 23 8 2)    (driver m49 26 23 25 8 2)      (load m50 1 26 26 4 12)))
```

Appendix E - Implemented Circuit Checks

In this Appendix, the errors checked for by the current implementation of RUBICC are presented. The format is as follows: Each error frame is listed with its slots, the rules which, if proven true, will fill these slots, the type of rule (forward or backward chaining) and the program module which contains the rule. Explanations of each of these errors are included after each table.

E.1. Error Frame: INV-ERRORS (Inverter Errors)

Error Slot	Rule Name	Type	Program Module
Beta-Ratio	Check-Beta-Ratio-1	F	Inv-Rules
Input-Clocking	Check-Clkout-Inverter-1	F	Inv-Rules
	Check-Clkout-Inverter-2	F	Inv-Rules

Explanations:

BETA-RATIO is defined as the width to length ratio of the driver divided by the width to length ratio of the load. Static gates and inverters, must have a minimum beta ratio to guarantee proper output zero levels. This number is dependent on numerous parameters such as process technology characteristics, circuit operation voltages and load configuration. RUBICC uses constants stored in *G-CON for the various load configurations to determine the required beta-ratio of the gate.

INPUT-CLOCKING rules check *clkout inverters* (Figure E-1) to be sure that the same clock drives the input transfer gate (m1) and the pull-up driver (m2).

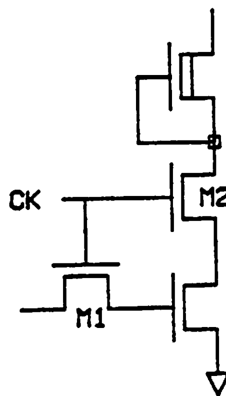

Figure E-1 Clkout-Inverter

E.2. Error Frame: GATE-ERRORS

Error Slot	Rule Name	Type	Program Module
Nand-Length	Static-Gate-Check-1	F	Gate-Rules p.
Beta-Ratio	Static-Gate-Check-2	F	Gate-Rules
Dynamic-Clocking-1	Dynamic-Gate-Check-1	F	Gate-Rules
Dynamic-Clocking-2	Dynamic-Gate-Check-2	F	Gate-Rules
Dynamic-Clocking-4	Dynamic-Gate-Check-4	F	Gate-Rules
Race-Condition	Dynamic-Gate-Check-3	F	Gate-Rules

Explanations:

NAND-LENGTH refers to the maximum number of series fets in the gate's current path from the gate-output to ground. Most technologies have a maximum limit to this number because each series transistor degrades the gate's output zero level. The constant RUBICC uses for this check is 3. Any static gates with greater that 3 transistors in this path are flagged.

BETA-RATIO refers to the same type of check as listed in inverter errors, Section E.1.

DYNAMIC-CLOCKING-1 performs checks on dynamic gates to detect any of the following error conditions:

Check if precharge and true phases are equal

Check if pull-down structure is clocked on more than one clock phase

Check if pull-down structure has a gate which is always held high

DYNAMIC-CLOCKING-2 checks for a dynamic gate which is always pulled low on a particular clock phase. This isn't a particularly useful circuit.

DYNAMIC-CLOCKING-4 checks for a dynamic gate that is never clocked. There are cases where this wouldn't be an error, though its a strange use for this type of gate.

RACE-CONDITION checks if there exists an xi-driver (driver with transfer gate) in the pull-down structure of a dynamic gate whose transfer gate is clocked on the phase that the gate is supposed to be true. This can cause precharge loss problems at node4 (Figure E-2) due to a race condition that occurs when the *xi-driver* ("m1") gate node comes true during ck2.

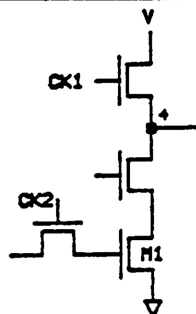

Figure E-2

E.3. Error Frame: GATE-ERRORS-B

Error Slot	Rule Name	Type	Program Module
Feedback-Desirable	Feedback-Check	B	Gate-Rules
Input-Clocking-Error	Static-Gate-Input-Clock-Check	B	Gate-Rules

Explanations:

FEEDBACK-DESIRABLE means that a feedback transistor is recommended to be placed in the position of "m1" in Figure E-3 to solve the following problem: the output (node 4) is held low by the same clock phase that loads "m3's" gate (node3) through transfer-gate "m2". If a zero level was stored on node3, node4 will rise when the clock phase goes low. Drain to gate coupling from node4 to node3 will cause node3 to also rise. This can cause the gate to be slow. The feedback transistor is usually a minimum driver since it's only job is to keep node 3 low in the event of this coupling.

INPUT-CLOCKING-ERROR checks any gate whose output is clocked low on a phase (see Figure E-3) to be sure that any transfer gate ("m2") which loads any input to the gate is clocked on the same phase. If this is not the case, an error occurs, since it doesn't make sense for the input to such a gate to change during the time when the output is to become true.

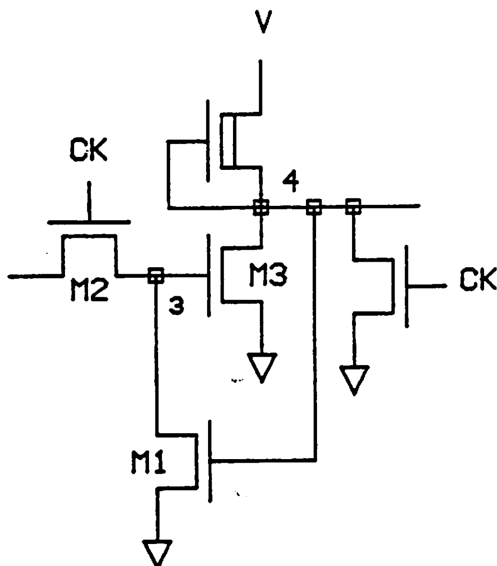

Figure E-3

E.4. Error Frame: FUNNY-FET

Error Slot	Rule Name	Type	Program Module
Max-Driver-Length	Funny-Fet-1	B	F-F-Rules
Min-Driver-Width	Funny-Fet-2	B	F-F-Rules
Min-Load-Length	Funny-Fet-4	B	F-F-Rules
Min-Load-Width	Funny-Fet-3	B	F-F-Rules
Max-Cap-Length	Funny-Fet-5	B	F-F-Rules
Single-Connection	Funny-Fet-6	B	F-F-Rules

Explanations:

These are transistors which either violate circuit design rules or make no sense. Note that all the constants are accessed from the *G-Con Frame.

MAX-DRIVER-LENGTH flags drivers longer that 2.5 microns

MIN-DRIVER-WIDTH flags drivers narrower than 3.5 microns.

MIN-LOAD-LENGTH flags loads narrower than 3.5 microns.

MAX-CAP-LENGTH flags mos-capacitors which are longer than 15u. Mos-caps longer than this may cause high-frequency problems due to time constants associated in forming the inversion layer.

SINGLE-CONNECTION flags transistors that have no other circuit elements connected to them. These transistors are probably not very useful.

E.5. Error Frame: FUNNY-NODE

Error Slot	Rule Name	Type	Program Module
Gate-Only	Funny-Node-1	B	F-F-Rules
Supply-Gate-Only	Funny-Node-2	B	F-F-Rules
Clock-Supply-Short	Funny-Node-3	B	F-F-Rules
Single-Connection	Funny-Node-4	B	F-F-Rules
	Funny-Node-5	B	F-F-Rules
Clocking-Flag	Dynamic-Clocking-Rule-1	B	Gate-Rules
Long-Rc-Flag	Long-Rc-Flag	B	Rc-Rules

Explanations:

GATE-ONLY means that a node has only gates connected to it. This isn't a very useful part of a circuit.

SUPPLY-GATE-ONLY flags nodes with only gates and supplies connected to it.

CLOCK-SUPPLY-SHORT flags a node which is connected to both the positive end of a supply and clock.

SINGLE-CONNECTION means that a given node has only one circuit element connected to it.

CLOCKING-FLAG means that this node has a probable error due to the following conditions: The node is precharged, and there is a driver connected to this node in the configuration of "m2" in Figure E-4. Either driver "m2's" gate is always held high, or connected to a clock.

LONG-RC-FLAG checks for noise sensitive cases where a static signal may drive many gates over a long distance. Since the signal is static, there are no timing constraints on it and hence the tendency is to use long, high impedance load fets for this application. If the fets being driven control highly dynamic signals, coupling can dangerously reduce the "high level" on the static signal. RUBICC checks for this by calculating the total node capacitance and approximating a resistance for the load transistor by using constants stored in *G-CON. An error is flagged if the "RC Time Constant" for the node is greater than a *G-CON constant called "noise-tau".

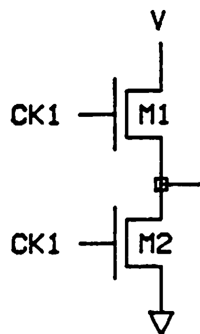

Figure E-4

E.6. Error Frame: Coupling Errors

Error Slot	Rule Name	Type	Program Module
Xi-Driver-Coupling	Xi-Driver-Coupling-Rule-1	F	Couple-Rules
Xi-Driver-Coupling-1	Xi-Driver-Coupling-Rule-2	F	Couple-Rules

Explanations:

XI-DRIVER-COUPLING flags an xi-driver (driver with transfer gate) whose ratio of driver gate area to transfer gate width is below a constant stored in *G-CON (mn-dr-ga-xf-w). See Figure E-5a. The problem here is that if "m1" is too wide, coupling from node3 to node4 becomes significant and a serious loss of charge can occur at node4. If a logical one-level is stored on node4 during a clock phase, "m1" comes out of inversion when the clock falls. The coupling is just due to the gate-drain overlap capacitance inherent in any mos-transistor.

XI-DRIVER-COUPLING-1 flags xi-drivers whose sources are not grounded. Consider the circuit in Figure E-5b. Suppose a high level is stored on node3 during clk1. During clk2, nodes 2 and 4 will start going low. Capacitive coupling from node2 to node3 and node4 to node3 due to overlap capacitances and "m2's" inversion layer will also pull node3 low, limiting the performance of the circuit. The solution to this problem is to put the xi-driver on the bottom of the fet string with its source connected to ground.

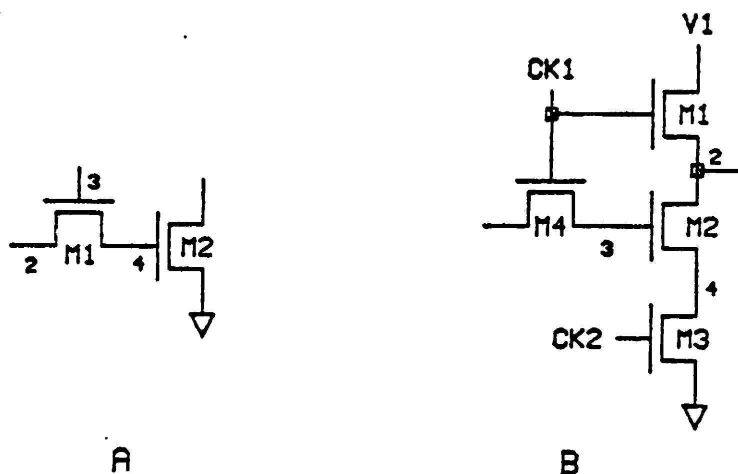

Figure E-5

E.7. Error Frame: SUPER-BUFFER-ERRORS

Error Slot	Rule Name	Type	Program Module
Power-Waste-Flag	Super-Buffer-Flag-1	F	Supbuf-Rules
Aggressive-Br-Flag	Super-Buffer-Flag-2	F	Supbuf-Rules
Poor-Input-Drive	Super-Buffer-Flag-4	F	Supbuf-Rules

Explanations:

POWER-WASTE-FLAG checks for the following conditions in a super-buffer (Figure E-6a) which may mean that its speed-power-product could be improved:

(W/L of "m1" > 0.75 * W/L of "m2) or (W/L of "m1" <0.25 W/L of "m2")

AGGRESSIVE-BR-FLAG suggests that the beta-ratio of the inverter composed of "m1" and "m4" (Figure E-6a) could be made smaller (more aggressive) than the minimum required inverter beta-ratio because a slightly higher zero level on node1 will make the buffer faster and not be detrimental. If the beta-ratio is too small, it will be flagged as having a gate beta-ratio error.

POOR-INPUT-DRIVE flags the case where a transfer gate drives the input to a non-inverting super-buffer. See Figure E-6b. This is a problem because transfer-gate "m5" limits the gate of "m1" to the voltage V1-Vt, where Vt is "m5's" threshold.

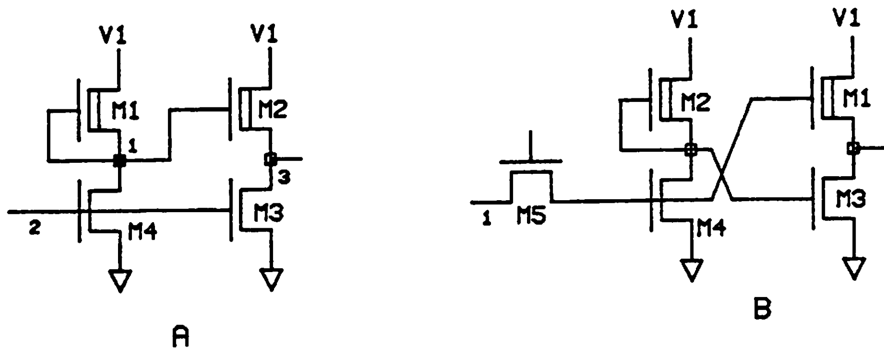

Figure E-6

E.8. Error Frame: DRN-BOOT-ERRORS

Error Slot	Rule Name	Type	Program Module
Phase-Hold-Down	Drn-Boot-Check-1	B	D-Boot-Rules
Clocking-Error	Check-Db-Clockin	B	D-Boot-Rules
Mos-Cap-Backwards	Check-Db-Mos-Cap	B	D-Boot-Rules
Boot-Node-Not-Active-Low	Check-Db-Boot-Node	B	D-Boot-Rules
Longer-Driver-Needed	Check-Db-Hd-Length	B	D-Boot-Rules

Explanations:

PHASE-HOLD-DOWN flags a drain-bootstrap circuit without a pull-down transistor ("m5" in Figure E-7a) on its output which is clocked by a phase other than the boot-phase.

CLOCKING-ERROR flags a bootstrapper whose predriver is clocked on the same phase as the boot-phase (clock connected to the drain of the output transistor). This is not a meaningful circuit.

MOS-CAP-BACKWARDS flags a bootstrapper whose mos-capacitor is connected backwards. Drain bootstrapper mos-caps must be connected with their gates connected to the boot node (node4 of Figure E-7a). If connected backwards the circuit doesn't work.

BOOT-NODE-NOT-ACTIVE-LOW means that the boot-node is not actively held low when the bootstrapper is not supposed to fire. This is illustrated in Figure E-7b. In some technologies, capacitive ratios are such that the bootstrapper may fire under this condition even if node2 was initially precharged low during clk1.

LONGER-DRIVER-NEEDED means that the active hold down to the drain-bootstrapper ("m3" in Figure E-7a) needs to be made longer. This is because the boot-node (node4 in Figure E-7a) voltage goes above the supply voltage and a longer fet is needed to avoid the occurrence of punch-through.

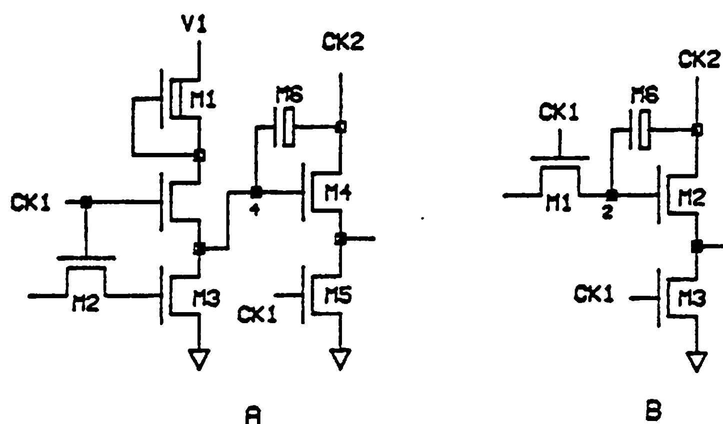

Figure E-7

E.9. Error Frame: Register Errors

Error Slot	Rule Name	Type	Program Module
Critical-Node-Flag	Reg-Critical-Node-Flag	B	Reg-Rules
Clocking-Error	Reg-Clocking-Check	B	Reg-Rules
	Reg-Clocking-Check-2	B	Reg-Rules
Internal-Connection	Reg-Internal-Con-Check	B	Reg-Rules

Explanations: (see Figure E-8)

CRITICAL-NODE-FLAG flags node2 in Figure E-8. This node is critical because it is sensitive to coupling due to the fact that this register's refresh transistor ("m9") is connected to a clock phase, rather than an an inverted set signal. Care must be taken in the layout to keep extraneous signals from coupling to this node.

CLOCKING-ERROR flags registers that are not clocked correctly. It means that the register violates one of the following rules:

The set transistor ("m2") must be clocked by the same phase as the core input transistor ("m3").

The recirculate transistor ("m9") must be clocked on the same phase as the second stage input ("m7").

The recirculate transistor ("m9) and the input transistor ("m2") must be clocked on different phases.

The input transistor ("m2") and the output transistor ("m8") must be clocked on the same clock phases.

INTERNAL-CONNECTION refers to errors caused by extraneous transistors connecting the internal nodes of the register cell together, and thereby not allowing the circuit to function as a register cell.

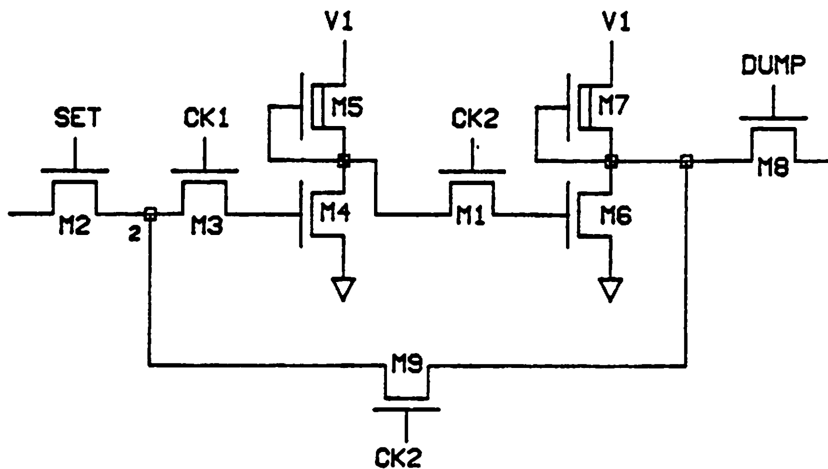

Figure E-8

E.10. Error Frame: Race Errors

Error Slot	Rule Name	Type	Program Module
Precharge-Loss	Race-Condition-1	B	Race-Cond
Input-Skew-Flag	Dynamic-Xor-Race-Condition	B	Race-Cond

Explanations:

PRECHARGE-LOSS can occur on the output of a dynamic gate if driven by a static gate under the following condition: Static gate "A" drives the input of static gate "B". Static Gate "B" drives the input to dynamic gate "C". Gates "A" and "B" are held low during the precharge phase of dynamic gate "C". A race condition occurs when clk2 goes low which could turn on dynamic gate "C" for a short time and thereby erroneously discharge gate "C's" output. This situation is illustrated in Figure E-9a.

INPUT-SKEW-FLAG flags gates whose correct operation is sensitive to the timing and / or rise and fall times of their inputs. This is illustrated by the dynamic xor-gate in Figure E-9b. If nodes 2 and 3 don't fall within a certain amount of time relative to each other (determined by process parameters and layout geometries), node4 which was intended to stay high in this case can go low.

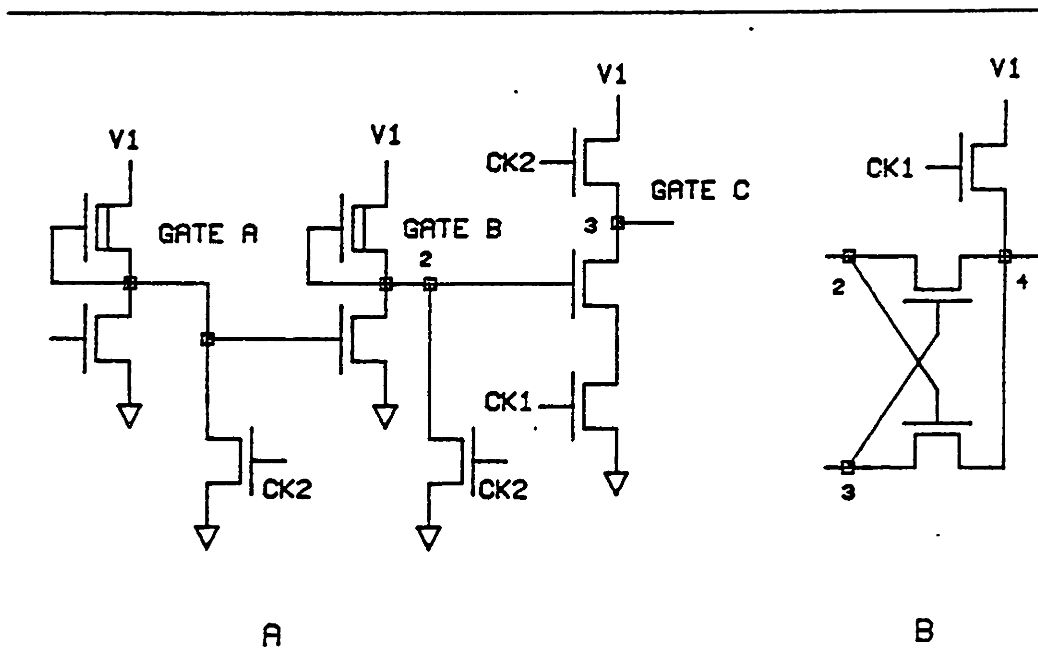

Figure E-9

E.11. Error Frame: Clock-skew-errors

Error Slot	Rule Name	Type	Program Module
Clock-Skew-Flag-1	Clock-Skew-Rule-1	B	Race-Cond
	Clock-Skew-Rule-2	B	Race-Cond
Clock-Skew-Flag-2	Clock-Skew-Rule-3	B	Race-Cond

Explanations:

CLOCK-SKEW-FLAG-1 flags any transfer gate whose drain or source is driven by a bootstrapper as sensitive to clock skew. This skew might come from distribution delays on the chip or from timing delays inherent in the bootstrapper itself. This condition is illustrated in Figure E-10a. If there exists significant clock skew, node4 could be erroneously charged or discharged before the transfer gate ("m2") is turned off.

CLOCK-SKEW-FLAG-2 flags two drivers who are connected in series, one whose gate is connected to a clock, and the other whose gate is connected to a drain bootstrapper. This is sensitive to the same problem as described above. This case is illustrated in Figure E-10b, where "m1" is the drain-bootstrapper.

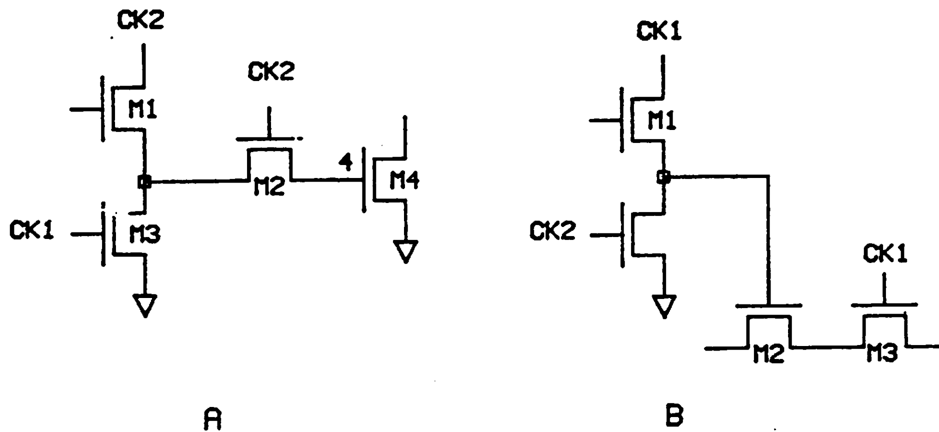

Figure E-10

E.12. Error Frame: Charge-share Errors

Error Slot	Rule Name	Type	Program Module
Feedback-Glitch-Flag	Charge-Share-2	B	Cshare-Rules
Feedback-Glitch-Error	Charge-Share-1	B	Cshare-Rules

Explanations:

FEEDBACK-GLITCH-FLAG implies that the nodes involved are sensitive to the feed-back-glitch error described next. However, RUBICC performed some calculations and the circuit seems to be OK. But be careful, this is a really nasty problem to find on a fabricated chip because it's processing and voltage dependent.

FEEDBACK-GLITCH-ERROR means RUBICC has calculated that the nodes involved will probably have this problem under some process and voltage conditions. The schematic in Figure E-11 is sensitive to the "glitch". Here's what happens. Assume that node3 is high at the end of clk1 and that somehow capacitor "c1" got charged and hence node5 is also high at the end of clk1. The designed intention of the circuit is for node4 to stay low during clk2 under this circumstance. Depending upon the ratios of the W/L's of "m3" and "m6" and the relative values of capacitor "c1" and the parasitic capacitance "c2", node4 can glitch significantly above the threshold voltage when clk2 goes high. If node4 gets above "m4's "Vt", "m4" turns on, discharging node3, which allows node4 to go high, causing the circuit function improperly (die).

RUBICC performs the following calculations to determine the severity of this problem: If the the W/L of "m3" is 4 times bigger that the W/L of "m6" then the maximum glitch would be 0.2 times the supply voltage. For a 5v supply and a 1 volt threshold this should be OK. Likewise if the value of "c2" is 4 times bigger that the value of "c1" then the maximum glitch would be 1 volt also. Under either of these circumstances, RUBICC gives the glitch flag. If neither of these condition are met, RUBICC gives the error flag.

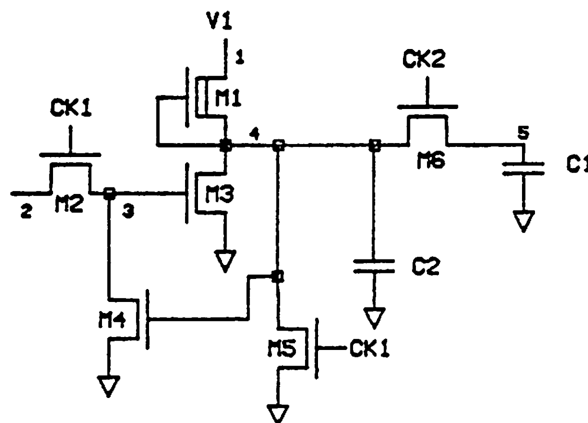

Figure E-11

E.13. Error Frame: Input-pad-errors

Error Slot	Rule Name	Type	Program Module
Missing-Protection-Device	Input-Protection-Check	B	Pad-Rules
Undershoot-Flag	Input-Undershoot-Check	B	Pad-Rules

Explanations:

MISSING-PROTECTION-DEVICE flags any chip input pad without a protection device configured as "m1" in Figure E-12a. Without this device, the gate of "m3" will not be protected against electrostatic zap and will probably cause the whole chip to die in assembly or be subject to infant mortality.

UNDERSHOOT-FLAG points out that the voltages on input pads of chips sometimes will undershoot below ground. Under these circumstances, transistors whose sources or drains are connected to the pad can inadvertently turn on. In Figure E-12b, if during clk2, the input pad went negative, it could turn on transistor "m3" and wrongly discharge node3.

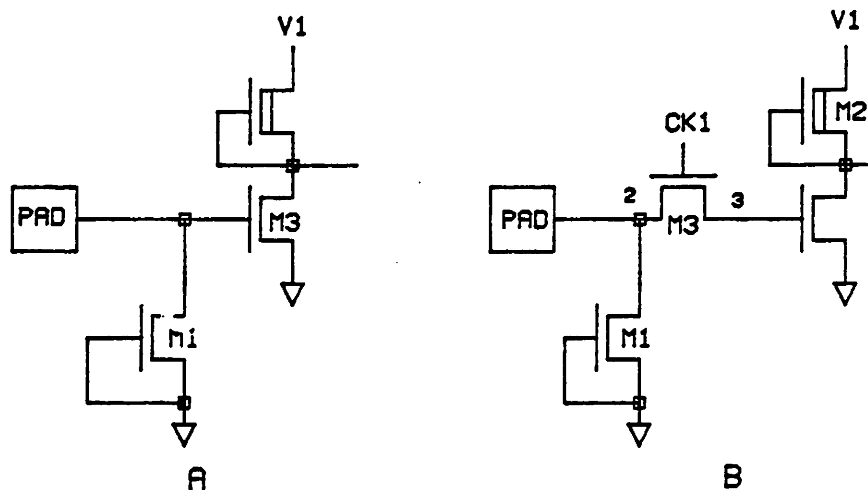

Figure E-12

Appendix F - RUBICC Source Code

RUBICC Source Code Modules

Main-1.sl 111
Tech-File.sl 114
Frames-1.sl 115
Frames-2.sl 120
Init-Funcs.sl 123
Fclass-Rules.sl 128
Fclass-Funcs.sl 130
S-P-Fet.sl 132
Comb-Structs.sl 134
Inv-Rules.sl 137
Inv-Funcs.sl 139
F-F-Rules.sl 141
Gate-Rules.sl 143
Gate-Funcs.sl 147
Couple-Rules.sl 148
Clking-Rules.sl 149
Clking-Funcs.sl 153
RC-Rules.sl 155
RC-Funcs.sl 155
Supbuf-Rules.sl 157
Supbuf-Funcs.sl 159
D-Boot-Rules.sl 160
D-Boot-Funcs.sl 162
Reg-Rules.sl 164
Reg-Funcs.sl 166
Struct-Rules.sl 167
Struct-Funcs.sl 167
Cshare-Rules.sl 169
Cshare-Funcs.sl 169
Race-Cond.sl 170
Race-Funcs.sl 172
Pad-Rules.sl 173
Net-Lists.sl 174

```
;********************FILE: MAIN-1SL**********
;
(setq *input-functions nil)

;;;;;;;;;;;;;;;;;;;;; main program control function ;;;;;;;;;;;;;;;;;;;;;
```

check-circuit

```
(defun check-circuit (net-list)
  (setq *circuit-name 'unnamed)
  (setq gctime!* 0)
  (setq *clka nil)(setq *clkb nil)(setq *clkc nil)(setq *clkd nil)
  (setq *longrc 0)
  (let( (s-time (time)) )
    (input-functions)
    (clear-frame-syms)
    (input-frames-rules)
    (make-pop-frame)              ; push frame marker onto *frames*
; create initial circuit frames
    (patom "BUILDING CIRCUIT FRAMES"(tab 30))(terpri)
    (create-i-frames net-list)
    (patom "CIRCUIT-NAME: ")(patom *circuit-name)(terpri)
    (patom "88 88 88")(terpri)
; transistor classification
    (solve-all '(?elements src-load ?x))
    (solve-all '(?elements drc-load ?x))
    (solve-all '(?elements ckc-load ?x))
    (solve-all '(?elements oth-load ?x))     ; must be last load solve
    (solve-all '(?transistor s-d-reversed ?x))
    (solve-all '(?elements xi-driver ?x))
    (solve-all '(?elements precharger ?x))
    (solve-all '(?elements pup-driver ?x))
    (solve-all-drn-boots)
    (solve-all '(?elements reg-driver ?x))   ; must be last driver solve
; series-parallel combinations
    (find-parallel-fets)
    (find-series-fets)
    (cond((or(fchildren 'series-struct)(fchildren 'parallel-struct))
          (combine-structs)))
   (find-other-structs)
; check circuit for errors
    (solve-error-frame 'funny-fet)
    (solve-error-frame 'funny-node)
    (solve-all '(?elements inverter ?x))
    (solve-all '(?elements static-gate ?x))
    (solve-all '(?elements dynamic-gate ?x))
    (solve-all '(?elements super-buffer ?x))
    (solve-all-drain-bootstraps)
    (solve-all-reg-cells)
    (solve-error-frame 'race-errors)
    (solve-error-frame 'gate-errors-b)
    (solve-error-frame 'charge-share-errors)
    (solve-error-frame 'drn-boot-errors)
    (solve-error-frame 'input-pad-errors)
    (solve-all-clk-skew-errors)
; print results
    (show-circuit-errors)(terpri)
    (show-all-gates-and-circuits)(terpri)
    (show-not-checked)(terpri)
    (print-stats)))
;
;**********************************************************************
; main-1 utility functions
```

input-functions

```
(defun input-functions ()
   (cond((null *input-functions)
         (setq *input-functions t)
         (dskin "init-funcs.sl")
         (dskin "inv-funcs.sl")
         (dskin "gate-funcs.sl")
         (dskin "s-p-fet.sl")
         (dskin "comb-structs.sl")
         (dskin "fclass-funcs.sl")
         (dskin "clking-funcs.sl")
         (dskin "rc-funcs.sl")
         (dskin "supbuf-funcs.sl")
         (dskin "d-boot-funcs.sl")
         (dskin "reg-funcs.sl")
         (dskin "struct-funcs.sl")
         (dskin "cshare-funcs.sl")
         (dskin "race-funcs.sl")
   )))
```

input-frames-rules

```
(defun input-frames-rules ()
  (cond((null *frames*)              ; read in generic frames
        (freset)
        (patom "LOADING GENERIC FRAMES"(tab 30))(terpri)
        (accept-forward-references)
        (dskin "tech-file.sl")
        (dskin "frames-2.sl")
        (dskin "frames-1.sl")
        (dskin "frames-3.sl")
        (patom "LOADING RULES"(tab 30))(terpri)
        (dskin  "inv-rules.sl")
        (dskin  "f-f-rules.sl")
        (dskin  "gate-rules.sl")
        (dskin "fclass-rules.sl")
        (dskin "couple-rules.sl")
        (dskin "clking-rules.sl")
        (dskin "rc-rules.sl")
        (dskin "supbuf-rules.sl")
        (dskin "d-boot-rules.sl")
        (dskin "reg-rules.sl")
        (dskin "race-cond.sl")
        (dskin "struct-rules.sl")
        (dskin "cshare-rules.sl")       ; charge-sharing-rules
        (dskin "pad-rules.sl")
        (end-forward-references)
        (patom "BUILDING RULE SYSTEM"(tab 30))(terpri)
        (terpri)
        (build-system))
     (t (clear-frame-syms)
       (pop frames)
       (clear-errors)
       (clear elements1)
       (build-system))))
```

pop-frames

```
(defun pop-frames ()          ; remove all frames instantiated after *pop*
   (let((frame-list *frames*))
   (while (and (car frame-list)(neq (setq frame (car frame-list)) '*pop*))
       (setq frame-list (cdr frame-list))
       (fremove frame))))
```

make-pop-frame

```
(defun make-pop-frame ()        ; add *pop* onto *frames*
```

```
(cond((null (framep '*pop*))
      (finstantiate 'elements '*pop*))))
```

clear-elements1

```
(defun clear-elements1 ()        ; clear the elements1 frame
  (let ((slot-list (fslots 'elements1))(e-slot nil))
    (while (setq e-slot (car slot-list))
      (setq slot-list(cdr slot-list))
      (cond((and(neq e-slot 'ako)(neq e-slot 'dummy))
            (fremove 'elements1 e-slot))))))
```

clear-errors

```
(defun clear-errors ()          ;clear all error frames
  (let ((frame-list (fchildren 'errors))(e-frame nil)(slot-list nil)
        (e-slot nil))
    (while (setq e-frame (car frame-list))
       (setq frame-list (cdr frame-list))
       (fremove 'thing 'instance '$value e-frame) ;demon removes e-frame ako
       (setq slot-list (delete 'ako (fslots-with-values e-frame)))
       (cond ((null slot-list)(next)))
       (while (setq e-slot (car slot-list))
          (setq slot-list (cdr slot-list))
          (fremove e-frame e-slot '$value)))))
```

solve-error-frame

```
(defun solve-error-frame (s-frame)
  (let ((slot-list (delete 'ako (fslots s-frame)))(slot nil))
    (while (setq slot (car slot-list))
      (setq slot-list (cdr slot-list))
      (solve-all '(,s-frame ,slot ?x)))))
```

my-freset

```
(defun my-freset ()
  (clear-frame-syms)
  (freset))
```

```
; causes finstantiate naming to start over from "1"
```

clear-frame-syms

```
(defun clear-frame-syms ()
  (let ((fr-list *frames*)(fr nil))
    (while (setq fr (car fr-list))
      (setq fr-list (cdr fr-list))
      (put fr 'next-id-number nil))))
```

toggle-rule-msgs

```
(defun toggle-rule-msgs ()
  (cond(*suppress-rule-messages* (setq *suppress-rule-messages* nil))
       (t (setq *suppress-rule-messages* t))))
```

toggle-skew-flag

```
(defun toggle-skew-flag ()
  (cond((fvo '*g con 'clk skew flag)
            (freplace '*g con 'clk-skew-flag '$value nil))
       (t(freplace '*g-con 'clk-skew-flag '$value t))))
```

```
; **********************FILE: TECH-FILE.SL *****

 ; Technology Frame for 5v only NMOS, 2 phase non-overlapping clocks

(deframe *g-con
  (ako($value(thing)))
;;;; transistor constants ;;;;;;;;;
 (mx-dr-l($value(2.5)))         ; maximum driver length
 (mn-dr-w($value(3)))            ; minimum driver width
 (mn-ld-l($value(3)))           ; minimum load length
 (mn-ld-w($value(3.5)))          ; minimum load width
 (mx-cap-l($value(15)))         ; maximum mos-cap length
 (std-ld-current($value(0.050)))  ; current for load with w/l = 1 (ma)
 (dr-eq-ratio                 ; convert driver l-div-w to load l-div-w
    ($value(3)))
;;;; gate constants ;;;;;;;;;;;;
 (st-nand-sl($value(3)))         ; maximum static nand string length
 (br-src-load($value(4.0)))       ; beta ratio for source connected load
 (br-drc-load($value(6.0)))       ; beta ratio for drain  connected load
 (br-ckc-load($value(6.0)))       ; beta ratio for clock connected load
 (br-oth-load($value(4.0)))
 (xi-dr-wrf($value(0.9)))         ; xi-driver width reduction factor
 (mn-dr-ga-xf-w($value(3.2)))      ; minimun driver gate-area to xfer-gate
                                  ;        width
;;;; gate and overlap capacitances ;;;;;;
 (gox-cap ($value(910e-6)))       ; gate area capacitance (pf/u^2)
 (gox-overlap-cap($value(200e-6))) ;gate overlap capacitance (pf/u)
;;;; dynamic circuit constants
 (noise-tau($value(20)))      . ; time constant for noise (ns)
;;;; super-buffer constants
 (mn-sup-buf-pwr-ratio          ; min ratio of predriver w/l to driver w/l
        ($value(0.25)))
 (mx-sup-buf-pwr-ratio          ; max ratio of predriver w/l to driver w/l
        ($value(0.75)))
 (mx-sup-buf-agg-br         ; max predriver aggressive beta-ratio
     ($value(^ (* 0.9 (fvo '*g-con 'br-src-load)))))
 (mn-sup-buf-agg-br         ; min predriver aggressive beta-ratio
     ($value(^ (* 0.8 (fvo '*g-con 'br-src-load)))))
;;;; charge-share-ratios
 (dr-cshare-ratio          ; (/  l/w of pull-down l/w of xfer-gate)
    ($value(4.0)))          ; couple back at most 1/5 of voltage
 (cap-cshare-ratio          ; (/ cap of driven node cap of couple-node)
    ($value(4.0)))          ; couple back at most 1/5 of voltage
;;;; drain-bootstrap constants
 (db-ahd-l($value(2.5)))   ; drain bootstrap active holddown width
;;;; clock skew sensitivity
 (clk-skew-flag($value(t))) ; true if clock skew is a problem
 )
```

```
;********************FILE FRAMES-1SL ********

(deframe transistor
      (ako($value(thing)))
      (d-node)
      (g-node)
      (s-node)
      (width)
      (length)
      (l-div-w)
      (string-l)
      (clk-input($ask(dont)))
      (clk-class)
      (class)
      (s-d-reversed)
      (trigger)
      (fb-tran-flag)      ; t if transistor is used as a feedback transistor
      (o-ins($ask(dont))) ; used for drain bootstrap rules
      (check)
      (status))   ; ($if-added((my-print :frame :value)))))

(deframe load
      (ako($value(transistor))))

(deframe src-load
      (ako($value(load)))
      (instance($if-added((fput :value 'status '$value 'free)))))

(deframe ckc-load
      (ako($value(load)))
      (instance($if-added((fput :value 'status '$value 'free)))))

(deframe drc-load
      (ako($value(load)))
      (instance($if-added((fput :value 'status '$value 'free)))))

(deframe oth-load
      (ako($value(load)))
      (instance($if-added((fput :value 'status '$value 'free)))))

(deframe driver
      (ako($value(transistor))))

(deframe reg-driver
      (ako($value(driver)))
      (instance($if-added((fput :value 'status '$value 'free)))))

(deframe xi-driver
      (ako($value(driver)))
      (instance($if-added((fput :value 'status '$value 'free))))
      (xfer-gate)  ;($if-added((fput :value 'status '$value 'in-use))))
      (fb-tran)    ;($if-added((fput :value 'status '$value 'in-use))))
      (in-node)
      (ga-xfw-ratio))   ; gate area to xfer-gate width ratio

(deframe mos-cap
      (ako($value(transistor)))
```

transistor

load

src-load

ckc-load

drc-load

oth-load

driver

reg-driver

xi-driver

mos-cap

```
    (instance($if-added((fput :value 'status '$value 'free)))))
```

precharger

```
(deframe precharger
    (ako($value(driver)))
    (instance($if-added((fput :value 'status '$value 'free))))
    (pre-phase)
    (supply))
```

pup-driver

```
(deframe pup-driver
    (ako($value(driver)))
    (instance($if-added((fput :value 'status '$value 'free))))
    (supply))
```

drn-boot

```
(deframe drn-boot
    (ako($value(driver)))
    (instance($if-added((fput :value 'status '$value 'free))))
    (s-node($ask(dont)))
    (other-clk-hold-down
                ($if-added((freplace :value 'status '$value 'in-use)))
                ($ask(dont)))
    (boot-phase($ask(dont)))
    (xfer-gate)
    (fb-tran))
```

one-port-element

```
(deframe one-port-element
   (ako($value(thing)))
   (node-num)
   (prot-device($if-added((freplace :value 'status '$value 'in-use)))
            ($ask(dont))))
```

pad

```
(deframe pad
   (ako($value(one-port-element))))
```

two-port-element

```
(deframe two-port-element
    (ako($value(thing)))
    (pos-node)
    (neg-node)
    (e-value))
```

active-two-port-element

```
(deframe active-two-port-element
    (ako($value(two-port-element))))
```

passive-two-port-element

```
(deframe passive-two-port-element
    (ako($value(two-port-element))))
```

cap

```
(deframe cap
    (ako($value(passive-two-port-element))))
```

supply

```
(deframe supply
    (ako($value(active-two-port-element))))
```

clock

```
(deframe clock
    (ako($value(active-two-port-element))))
```

node

```
(deframe node
    (ako($value(thing)))
    (instance($if-added((fput :value 'status '$value 'free))))
    (number)
```

```
    (status)           ;class/aspect doc: (class(allowable aspects))
    (class($ask(dont)))  ;((static(load push-pull)) (dynamic-hiZ(clk-phase))
    (class-1($ask(dont))) ;(clocked-always(clock-phase)) (clocked-conditional
    (aspect($ask(dont))   ; (clock-phase)) (precharge (clock-phase))
         ($if-added((aspect-print :frame :value))))
                       ; (always-high(supply)) (gnd (gnd)))
    (trans-struct)   ; all transistors with what part connected
    (trans)          ; all transistors connected
    (gate)           ; all transistors with gate connections
    (src-drn)        ; all transisitors with source or drain connections
    (load)           ; all load transistors connected
    (driver)         ; all drivers connected to node
    (struct)         ; all structures connected to node
    (cap)            ; capacitors connected
    (mos-cap)        ; mos capacitors connected
    (supply)         ; power supplyies connected
    (clock)          ; clocks connected
    (gate-cap)       ; capacitace of all gates connected to node (upf)
    (static-cap)     ; capacitance of all capacitors tied to gnd or supply
    (clka-cap)       ; capacitance of node to clka
    (clkb-cap)       ; capacitance of node to clkb
    (clkc-cap)       ; capacitance of node to clkc
    (clkd-cap)       ; capacitance of node to clkd
    (src-drn-cap)    ; capacitance of node to all sources and drains
    (other-cap)      ; capacitance of node to other places
    (total-cap)      ; sum of all capacitances
    (pad))           ; i/o pads connected
```

circuit

```
(deframe circuit
    (ako($value(thing))))
```

gate

```
(deframe gate
    (ako($value(thing)))
    (beta-ratio)
    (trigger)
    (status)
    (pull-up ($if-added((freplace :value 'status '$value 'in-use))))
    (pull-down ($if-added((make-status-in-use :value))))
    (in-node)
    (out-node)
    (supply-node)
    (supply)
    (struct($if-added((freplace :value 'status '$value 'in-use))))
    (dummy))
```

static-gate

```
(deframe static-gate
    (ako($value(gate))))
```

dyna-gate

```
(deframe dyna gate
    (ako($value(gate))))
```

dynamic-gate

```
(deframe dynamic-gate
    (ako($value(dyna-gate)))
    (instance($if-added((fput :value 'status '$value 'free))))
    (pre-phase)
    (true-phase))
```

dynamic-xc-xor

```
(deframe dynamic-xc-xor
    (ako($value(dyna-gate)))
    (instance($if-added((fput :value 'status '$value 'free))))
```

```
      (pre-phase)
      (true-phase))
```

inverter

```
(deframe inverter
      (ako($value(static-gate)))
      (xfer-gate))
```

reg-inverter

```
(deframe reg-inverter
      (ako($value(inverter)))
      (instance($if-added((fput :value 'status '$value 'free)))))
```

xi-inverter

```
(deframe xi-inverter
      (ako($value(inverter)))
      (instance($if-added((fput :value 'status '$value 'free))))
      (xfer-gate($if-added((freplace :value 'status '$value 'in-use))))
      (clock-node)
      (clock))
```

clkout-inverter

```
(deframe clkout-inverter
      (ako($value(inverter)))
      (instance($if-added((fput :value 'status '$value 'free))))
      (xfer-gate($if-added((freplace :value 'status '$value 'in-use))))
      (clock-node)
      (clkgate($if-added((freplace :value 'status '$value 'in-use))))
      (clock))
```

nor-gate

```
(deframe nor-gate
      (ako($value(static-gate)))
      (instance($if-added((fput :value 'status '$value 'free)))))
```

nand-gate

```
(deframe nand-gate
      (ako($value(static-gate)))
      (instance($if-added((fput :value 'status '$value 'free)))))
```

other-gate

```
(deframe other-gate
      (ako($value(static-gate)))
      (instance($if-added((fput :value 'status '$value 'free)))))
```

static-xc-xor

```
(deframe static-xc-xor
      (ako($value(static-gate)))
      (instance($if-added((fput :value 'status '$value 'free)))))
```

struct

```
(deframe struct
      (ako($value(thing)))
      (status)
      (l-div-w)
      (string l)              ; length of fet string
      (node-1($ask(dont)))
      (node-2($ask(dont)))
      (clk-input($ask(dont)))
      (clk-class($ask(dont)))
      (class)
      (clking-check)
      (substruct($if-added((make-status-in-use :value))))
      (trans($if-added((make-status-in-use :value)))))
```

parallel-struct

```
(deframe parallel-struct
    (ako($value(struct)))
    (instance($if-added((fput :value 'status '$value 'free)))))
```

series-struct

```
(deframe series-struct
    (ako($value(struct)))
    (instance($if-added((fput :value 'status '$value 'free)))))
```

super-struct

```
(deframe super-struct
    (ako($value(struct)))
    (instance($if-added((fput :value 'status '$value 'free)))))
```

xc-xor-struct

```
(deframe xc-xor-struct
    (ako($value(struct)))
    (instance($if-added((fput :value 'status '$value 'free))))
    (in-node)
    (out-node))
```

```
;********************FILE FRAMES-2SL ********
```

elements

```
(deframe elements
     (ako($value(thing)))
     (dummy)
     (nodes)
     (driver) (reg-driver)(xi-driver)(precharger)(pup-driver)(dbl)
          (drn-boot($ask(dont)))
     (load) (src-load)(ckc-load)(drc-load)(oth-load)
     (mos-cap)
     (dynamic-gate)
     (supply)
     (clock)
     (cap)
     (inverter) (simple-inverter)(d-i-inverter)
     (static-gate)
     (super-buffer)
     (reg-core)(reg-cell)
     (drain-bootstrap)
     (struct)
     (series-struct($if-added((fput :frame 'struct '$value :value))))
     (parallel-struct($if-added((fput :frame 'struct '$value :value))))
     (super-struct($if-added((fput :frame 'struct '$value :value))))
     (xc-xor-struct($if-added((fput :frame 'struct '$value :value)))))
```

elements1

```
(deframe elements1
     (ako($value(elements)))
     (dummy($value(1))))
```

errors

```
(deframe errors
     (ako($value(thing))))
```

inv-errors

```
(deframe inv-errors
     (ako($value(errors)))
     (dummy)
     (beta-ratio($ask(dont)))
     (coupling($ask(dont)))
     (input-clocking($ask(dont))))
```

gate-errors

```
(deframe gate-errors        ; filled by forward chain rules
     (ako($value(errors)))
     (nand length($ask(dont)))
     (beta ratio($ask(dont)))
     (dynamic clocking($ask(dont)))
     (dynamic-clocking 1($ask(dont)))(dynamic-clocking-2($ask(dont)))
     (dynamic-clocking 3($ask(dont)))(dynamic-clocking-4($ask(dont)))
     (race condition($ask(dont))))
```

gate-errors-b

```
(deframe gate-errors-b      ; filled by backward chain rules
     (ako($value(errors)))
     (feedback-desirable($ask(dont)))
     (input-clocking-error($ask(dont))))
```

funny-fet

```
(deframe funny-fet
     (ako($value(errors)))
```

```
    (max-driver-length($ask(dont)))
    (min-driver-width($ask(dont)))
    (min-load-length($ask(dont)))
    (min-load-width($ask(dont)))
    (max-cap-length($ask(dont)))
    (single-connection($ask(dont))))
```

funny-node

```
(deframe funny-node
    (ako($value(errors)))
    (gate-only($ask(dont)))
    (supply-gate-only($ask(dont)))
    (clock-supply-short($ask(dont)))
    (single-connection($ask(dont)))
    (clocking-flag($ask(dont)))
    (long-rc-flag($ask(dont))))
```

coupling-errors

```
(deframe coupling-errors
    (ako($value(errors)))
    (xi-driver-coupling($ask(dont)))
    (xi-driver-coupling-1($ask(dont))))
```

net-errors

```
(deframe net-errors
    (ako($value(errors)))
    (clocks($ask(dont))))
```

super-buffer-errors

```
(deframe super-buffer-errors
    (ako($value(errors)))
    (power-waste-flag($ask(dont)))
    (aggressive-br-flag($ask(dont)))
    (poor-input-drive($ask(dont))))
```

drn-boot-errors

```
(deframe drn-boot-errors
    (ako($value(errors)))
    (phase-hold-down($ask(dont)))
    (clocking-error($ask(dont)))
    (mos-cap-backwards($ask(dont)))
    (boot-node-not-active-low($ask(dont)))
    (longer-driver-needed($ask(dont))))
```

register-errors

```
(deframe register-errors
  (ako($value(errors)))
  (critical-node-flag ($ask(dont)))
  (clocking-error ($ask(dont)))
  (internal-connection ($ask(dont))))
```

race-errors

```
(deframe race-errors
  (ako($value(errors)))
  (precharge-loss($ask(dont)))
  (input-skew-flag($ask(dont))))
```

clk-skew-errors

```
(deframe clk-skew-errors
  (ako($value(errors)))
  (clock-skew-flag-1($ask(dont)))
  (clock-skew-flag-2($ask(dont))))
```

charge-share-errors

```
(deframe charge-share-errors
   (ako($value(errors)))
   (feedback-glitch-flag($ask(dont)))
   (feedback-glitch-error($ask(dont))))
```

input-pad-errors

```
(deframe input-pad-errors
   (ako($value(errors)))
   (missing-protection-device($ask(dont)))
   (undershoot-flag($ask(dont))))
```

*;********************FILE: INIT-FUNCS.SL ********

create-i-frames

```
(defun create-i-frames(net-list)
   (mapc net-list '(lambda(x)
                         (cond((circuit-name x)
                                (setq *circuit-name (car x))
                                (patom (car x))(terpri))
                               ((transistor x)(make-tran x))
                               ((equal (e-type x) 'pad)(make-pad x))
                               (t(make-element x))))))
```

transistor

```
(defun transistor (x)
   (or(equal(e-type x) 'load)
      (equal(e-type x) 'driver)))
```

circuit-name

```
(defun circuit-name (x)
       (equal (cadr (explode x)) '*))
```

make-tran

```
(defun make-tran (x)
   (let ((t-name (name x))
         (e-type (cond((equal (drain x)(source x)) 'mos-cap)
                      (t(e-type x)))))
      (finstantiate e-type t-name)
      (patom e-type)(patom " ")(patom t-name)(terpri)
      (fput 'elements1 e-type '$value t-name)
      (fput t-name 'd-node '$value (drain x))
      (fput t-name 'g-node '$value (gate x))
      (fput t-name 's-node '$value (source x))
      (fput t-name 'width  '$value (chan-width x))
      (fput t-name 'length '$value (chan-length x))
      (fput t-name 'class '$value 'unclassified)
      (fput t-name 'l-div-w '$value (/ (* 1.0 (chan-length x))
                                       (* 1.0 (chan-width x))))
      (fput t-name 'string-l '$value 1)
      (fput t-name 'check '$value 'unchecked)
      (make-t-node t-name (drain x)(gate x)(source x))))
```

make-element

```
(defun make-element (x)
  (let((e-name (name x)))
    (finstantiate (e-type x) e-name)
    (patom (e-type x))(patom " ")(patom e-name)(terpri)
    (fput 'elements1 (e-type x) '$value e-name)
    (fput e-name 'pos-node '$value (pos-node x))
    (fput e-name 'neg-node '$value (neg-node x))
    (fput e-name 'e-value  '$value (e-value x))
    (cond((equal (e-type x) 'clock)
          (cond((or(equal e-name '*clka)(equal e-name '*clkb)
                           (equal e-name '*clkc)(equal e-name '*clkd))
                 (fput 'net-errors 'clocks '$value e-name))
                ((null *clka)(setq *clka e-name))
                ((null *clkb)(setq *clkb e-name))
                ((null *clkc)(setq *clkc e-name))
                ((null *clkd)(setq *clkd e-name))
                (t(fput 'net-errors 'clocks '$value e-name)))))
    (make-en (e-type x) e-name (pos-node x) 'pos)
    (make-en (e-type x) e-name (neg-node x) 'neg)))
```

make-t-node

```
(defun make-t-node (t-name d-node g-node s-node)
      (make-tn t-name d-node 'drain)
      (make-tn t-name g-node 'gate)
      (make-tn t-name s-node 'source))
```

make-tn

```
(defun make-tn (t-name port-node port-type)
   (let ((node-name (make-node-name port-node))
        (tmp1 nil))
     (cond ((not (framep node-name))
            (finstantiate 'node node-name)
            (fput node-name 'number '$value port-node)
            (fput 'elements1 'nodes '$value port-node)))
     (setq tmp1 (assoc t-name (fvalues-only node-name 'trans)))
     (fremove node-name 'trans-struct '$value tmp1)
     (fput node-name 'trans '$value t-name)
     (fput node-name e-type '$value t-name)
     (fput node-name (cond((equal port-type 'gate) 'gate)
                          (t 'src-drn)) '$value t-name)
     (fput node-name 'trans-struct '$value
                         (cond ((null tmp1) `(,t-name ,port-type))
                               (t `(,(car tmp1) ,port-type ,(cadr tmp1)))))))
```

make-en

```
(defun make-en (e-type e-name port-node port-type)
   (let ((node-name (make-node-name  port-node))
        (tmp1 nil))
     (cond ((not (framep node-name))
            (finstantiate 'node node-name)
            (fput node-name 'number '$value port-node)
            (fput 'elements1 'nodes '$value port-node)))
     (setq tmp1 (assoc e-name (fvalues-only node-name e-type)))
     (fremove node-name e-type '$value tmp1)
     (fput node-name e-type '$value
                         (cond ((null tmp1) `(,e-name ,port-type))
                               (t `(,(car tmp1) ,port-type ,(cadr tmp1)))))))
```

single-connection

```
(defun single-connection (node-number)
      (let((node (make-node-name node-number)))
           (< (+ (length(fvalues-only node 'trans))
                 (length(fvalues-only node 'supply))
                 (length(fvalues-only node 'clock)))
               2)))
```

make-pad

```
(defun make-pad (x)
    (let ((p-name (implode `(,@(explode(name x)) - p a d )))
          (node-num (caddr x))(node (make-node-name (caddr x))))
       (finstantiate 'pad  p-name)
       (patom  p-name)(terpri)
       (fput p-name 'node-num '$value node-num)
       (cond ((not (framep node))
              (finstantiate 'node node)
              (fput node 'number '$value node-num)
              (fput 'elements1 'nodes '$value node-num)))
      (fput node 'pad '$value p-name)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; the rest of these are Utility functions that are used throughout the program

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

my-solve-all

```
(defun my-solve-all (clause)
   (let((s-list nil)(r-frame nil)
        (frame (car clause))(slot (cadr clause))(val (caddr clause)))
     (cond((equal frame '?elements)(setq r-frame 'elements1))
          (t (setq r-frame frame)))
     (while (solve '(,frame ,slot ,val))
         (cond ((null s-list)(setq s-list (fvso r-frame slot)))
               (t (setq s-list (cons (fvo r-frame slot) s-list))))
         (fremove r-frame slot '$value))
     (cond (s-list (fput-values r-frame slot s-list)))
     s-list))
```

make-node-name

```
(defun make-node-name (x)
     (implode '(n o d e ,@(explode x))))

; simple macro's for parsing the net-list

(dm e-type (l)
    '(car ,(cadr l)))
(dm name (l)
    '(cadr ,(cadr l)))
(dm drain (l)
    '(caddr ,(cadr l)))
(dm gate (l)
    '(cadddr ,(cadr l)))
(dm source (l)
    '(caddddr ,(cadr l)))
(dm chan-width (l)
    '(cadddddr ,(cadr l)))
(dm chan-length (l)
    '(caddddddr ,(cadr l)))
(dm pos-node (l)
    '(caddr ,(cadr l)))
(dm neg-node (l)
    '(cadddr ,(cadr l)))
(dm e-value (l)
    '(caddddr ,(cadr l)))

;  needed macros not in PSL
(dm caadadar (x)
    '(caar(cdadar ,(cadr x))))
(dm caadadr (x)
    '(caar(cdadr ,(cadr x))))

; macros for abbreviations of hprl functions:
;;;;;;;;;;;;

; fvalue-only

;;;;;;;;;;;;
(dm fvo (l)
  '(fvalue-only ,(cadr l) ,(caddr l)))

;;;;;;;;;;;;
; fvalues-only
;;;;;;;;;;;;
(dm fvso (l)
    '(fvalues-only ,(cadr l) ,(caddr l)))

; symbol manipulation functions
```

newsym

```
(defun newsym (symb)
```

```
   (let ((numb (get symb 'num)))
     (cond ((null numb)(setq numb 0)
                          (put '*syms* 'sym (cons symb (get '*syms* 'sym)))))
     (implode (append (explode symb)
              (cons '- (explode (put symb 'num (+ numb 1))))))))
```

clear-new-syms

```
(defun clear-new-syms (x)
    (let ((sym-list (get x 'sym))(symbol nil))
    (while (setq symbol (car sym-list))
        (setq sym-list (cdr sym-list))
        (remprop symbol 'num)
        (remprop x 'sym))))
```

```
; print statistics on the program
```

print-stats

```
(defun print-stats ()
   (patom "garbage collection time = ")
           (patom (/ (/ gctime!* 1000.0) 60.0))(patom " min") (terpri)
   (patom "total run time= ")
           (patom (/ (/ (- (time) s-time) 1000.0) 60.0))
           (patom " min")(terpri))
```

```
; recursively finds all transistors in a gate or structure
```

get-trans

```
(defun get-trans (x)
 (cond ((null x) nil)
       ((atom x) (cond ((akop x 'transistor)(list x))
                       ((or(akop x 'series-struct)(akop x 'parallel-struct))
                        (fvso x 'trans))
                       ((and (akop x 'xc-xor-struct)(null(fvso x 'substruct)))
                        (fvso x 'trans))
                       ((akop x 'gate)(append(fvso x 'pull-up)
                         (append(fvso x 'pull-down)(fvso x 'xfer-gate))))
                       ((akop x 'super-buffer)
                         (append (fvso x 'predriver)(fvso x 'driver)))
                       ((akop x  'drain-bootstrap)
                        (append
                        (append (fvso x 'predriver)(fvso x 'drn-boot))
                        (append (fvso x 'mos-cap)
                                (fvso x 'other-clk-hold-down))))
                       ((akop x 'reg-cell)
                         (append (fvso x 'reg-core)
                             (append (fvso x 'in-stage)(fvso x 'out-stage))))
                       ((akop x 'reg-core)
                        (append (fvso x 'in-stage)
                            (append(fvso x 'out-stage)(fvso x 'rec-stage))))
                       (t(get-trans (fvso x 'substruct)))))
       (t(append (get trans (car x))(get-trans (cdr x))) )))
```

```
; will eventually be part of the user interface
;shows all transistors in a given circuit
```

show-me

```
(defun show-me (x)
   (cond((framep x)(get trans x))
        (t(patom x)(patom " is not a frame")(terpri))))
```

show-all-gates

```
(defun show-all-gates()
   (let ((g-list (fringe 'gate))(g nil)(e-list nil))
    (while (setq g (car g-list))
         (setq g-list (cdr g-list))
```

```
        (setq e-list (show-me g))
        (cond (e-list
              (patom g(tab 2))(patom "   ")
              (patom e-list)
              (terpri))) )) )
```

show-all-circuits

```
(defun show-all-circuits()
   (let ((g-list (fringe 'circuit))(g nil)(e-list nil))
     (while (setq g (car g-list))
        (setq g-list (cdr g-list))
        (setq e-list (show-me g))
        (cond (e-list
              (patom g(tab 2))(patom "   ")
              (patom e-list)
              (terpri))) )) )
```

show-all-gates-and-circuits

```
(defun show-all-gates-and-circuits ()
(patom "CIRCUITS and GATES IDENTIFIED: ")(terpri)
   (show-all-circuits)
   (show-all-gates))
```

show-not-checked

```
(defun show-not-checked ()
  (let ((c-list (append(solve-all '(?transistor status free))
                       (solve-all '(?struct status free))))
       (new-list nil) (x nil))
  (while (setq x (car c-list))
       (setq c-list (cdr c-list))
       (setq new-list (append (get-trans (car x)) new-list)))
  (patom "FREE TRANSISTORS: ")(terpri)
  (patom new-list(tab 5))))
```

show-circuit-errors

```
(defun show-circuit-errors ()
   (let ((frame-list (fchildren 'errors))(e-frame nil)(slot-list nil)
        (e-slot nil))
     (patom "ERRORS FOUND FOR CIRCUIT:  ")(patom 'circuit-name)(terpri)
     (while (setq e-frame (car frame-list))
        (setq frame-list (cdr frame-list))
        (setq slot-list (delete 'ako (fslots-with-values e-frame)))
        (cond ((null slot-list)(next))
              (t(patom e-frame)(terpri)))
        (while (setq e-slot (car slot-list))
           (setq slot-list (cdr slot-list))
           (patom  e-slot(tab 5))
           (patom (fvalues-only e-frame e-slot)(tab 30))(terpri)))))
```

aspect-print

```
(defun aspect-print (frame val)
  (patom "aspect added: ")(patom frame)(patom " ")(patom val)(terpri))
```

make-status-in-use

```
(defun make-status-in-use (x)
   (let((fb (fvo x 'fb-tran))(xf (fvo x 'xfer-gate)))
;   (patom "make-status-in-use ")(terpri)
;   (patom x)(patom " ")(patom fb)(patom " ")(patom xf)(terpri)
   (cond (fb(freplace fb 'status '$value 'in-use)))
   (cond (xf(freplace xf 'status '$value 'in-use)))
   (freplace x 'status '$value 'in-use)))
```

```
;*******************FILE: FCLASS-RULES.SL******

; find an xi-driver without feedback
(rule find-xi-driver backward-chain-rule
   (type(elements ?elements)(driver ?driver ?driver1))
   (premise (test (and (?elements dummy 1)
                       (?driver g-node ?dgn)
                       (or (?driver1 s-node ?dgn (neq ?driver1 ?driver))
                           (?driver1 d-node ?dgn (neq ?driver1 ?driver))))
              (xi-driver-condition ?driver1 ?dgn)))
   (conclusion (?elements xi-driver ^(make-xi-driver ?driver ?driver1 nil))))

; find an xi-driver with feedback
(rule find-xi-driver-1 backward-chain-rule
   (type(elements ?elements)(driver ?dr ?dr1 ?dr2))
   (premise (test (and (?elements dummy 1)
                       (?dr g-node ?dgn)
                       (?dr s-node ?dsn)
                       (?dr d-node ?ddn)
                       (or (?dr1 s-node ?dgn (neq ?dr1 ?dr))
                           (?dr1 d-node ?dgn (neq ?dr1 ?dr)))
                       (?dr2 s-node 0(neq ?dr2 ?dr1))
                       (?dr2 d-node ?dgn(neq ?dr2 ?dr))
                       (?dr2 g-node ?gn2(or (equal ?gn2 ?dsn)
                                            (equal ?gn2 ?ddn))))
              (xi-driver-condition-1 ?dr1 ?dgn)))
   (conclusion (?elements xi-driver ^(make-xi-driver ?dr ?dr1 ?dr2))))

(rule find-precharger backward-chain-rule
     (type(driver ?dr)(clock ?clk)(supply ?sup)(elements ?el))
     (premise (and (?el dummy 1)
                   (?dr d-node ?dn)
                   (?dr g-node ?gn)
                   (?clk pos-node ?gn)
                   (?sup pos-node ?dn)))
     (conclusion(?el precharger ^(make-precharger ?dr ?clk ?sup))))

(rule find-pup-driver backward-chain-rule
     (type(driver ?dr)(clock ?clk)(supply ?sup)(elements ?el))
     (premise (and (?el dummy 1)
                   (?dr d-node ?dn)
                   (?dr g-node ?gn)
                   (?sup pos-node ?dn)
                   (unknown ?clk pos-node ?gn)))
     (conclusion(?el pup-driver ^(make-pup-driver ?dr ?sup))))

(rule find-regular-drivers backward-chain-rule
   (premise (?driver class unclassified))
   (conclusion (elements1 reg-driver ^(make-reg-driver ?driver))))

(rule find-src-loads backward-chain-rule
    (type (elements))
    (premise (and (?elements dummy 1)
                  (?load class unclassified)
                  (?load g-node ?lgn)
                  (?load s-node ?lgn)
                  (?load d-node ?ldn (neq ?ldn ?lgn))
                  (?supply pos-node ?ldn)))
    (conclusion (?elements src-load ^(classify-load ?load 'src-load))))

(rule find-drc-loads backward-chain-rule
     (type (elements))
    (premise (and (?elements dummy 1)
                  (?load class unclassified)
```

```
                    (?load g-node ?lgn)
                    (?load d-node ?lgn)
                    (?load s-node ?lsn (neq ?lsn ?lgn))
                    (?supply pos-node ?lgn)))
   (conclusion (?elements drc-load ^(classify-load ?load 'drc-load))))

(rule find-ckc-loads backward-chain-rule
    (type (elements))
   (premise (and (?elements dummy 1)
                    (?load class unclassified)
                    (?load g-node ?lgn)
                    (?load d-node ?ldn (neq ?lgn ?ldn))
                    (?load s-node ?lsn (neq ?ldn ?lsn))
                    (?clock pos-node ?lgn)
                    (?supply pos-node ?ldn)))
   (conclusion (?elements ckc-load ^(classify-load ?load 'ckc-load))))

(rule find-oth-loads backward-chain-rule
   (type (elements))
   (premise (and(?elements dummy 1)
                  (?load class unclassified)))
   (conclusion (?elements oth-load ^(classify-load ?load 'oth-load))))

(rule reverse-src-drn-rule backward-chain-rule
   (type (transistor ?tr)(active-two-port-element ?ae))
   (premise (or (and (?tr s-node ?sn (not(akop ?tr 'mos-cap)))
                     (?ae pos-node ?sn))
                (?tr d-node 0 (not (akop ?tr 'mos-cap)))))
   (conclusion(?tr s-d-reversed ^(reverse-s-d ?tr))))
```

;*********************FILE: FCLASS-FUNCS.L *****

make-reg-driver

```
(defun make-reg-driver (dr)
   (freplace dr 'ako '$value 'reg-driver)
   (freplace dr 'class '$value 'reg)
    dr)
```

make-pup-driver

```
(defun make-pup-driver (dr sup)
   (freplace dr 'ako '$value 'pup-driver)
   (freplace dr 'class '$value 'pup)
   (freplace dr 'supply '$value sup)
   dr)
```

make-precharger

```
(defun make-precharger (dr clk sup)
    (freplace dr 'ako '$value 'precharger)
    (freplace dr 'class '$value 'precharger)
    (freplace dr 'pre-phase '$value clk)
    (freplace dr 'supply '$value sup)
     dr)
```

xi-driver-condition

```
(defun xi-driver-condition (driver node-num)
   (let ((n1 (make-node-name node-num))(n2 (find-next-node driver node-num)))
    (and (or(equal 1 (length (fvso n1 'src-drn)))
            (and(equal 2 (length (fvso n1 'src-drn)))
                (equal 1 (length (fvso n1 'mos-cap)))) )
         (null (fvso n1 'supply))
         (null (fvso n1 'clock))
         (neq n1 0)
         (neq n2 0))))
```

xi-driver-condition-1

```
(defun xi-driver-condition-1 (driver node-num)
   (let ((n1 (make-node-name node-num))(n2 (find-next-node driver node-num)))
    (and (or(equal 2 (length (fvso n1 'src-drn)))
            (and(equal 3 (length (fvso n1 'src-drn)))
                (equal 1 (length (fvso n1 'mos-cap)))) )
         (null (fvso n1 'supply))
         (null (fvso n1 'clock))
         (neq n1 0)
         (neq n2 0))))
```

make-xi-driver

```
(defun make-xi-driver (dr xf fb-tran)
   (freplace dr 'ako '$value 'xi-driver)
   (freplace xf 'ako '$value 'reg-driver) ; make status free
   (fput dr 'xfer-gate '$value xf)
   (freplace dr 'class '$value 'xi)
   (freplace xf 'class '$value 'xfer)
   (freplace dr 'ga-xi w-ratio '$value
                        (/ (* 1.0 (fvo dr 'width)(fvo dr 'length))
                           (* 1.0 (fvo xf 'width))))
   (freplace dr 'width '$value (* (fvo '*g-con 'xi-dr-wrf)
                                  (fvo dr 'width)))
   (freplace dr 'l-div-w '$value (* 1.0 (/ (fvo dr 'length)(fvo dr 'width))))
   (fput dr 'in-node '$value (cond ((equal (fvo dr 'g-node)(fvo xf 'd-node))
                                    (fvo xf 's-node))
                                   (t (fvo xf 'd-node))))
   (cond(fb-tran (fput dr 'fb-tran '$value fb-tran)
                 (freplace fb-tran 'fb-tran-flag '$value t)))
   (freplace dr 'trigger '$value t)
```

```
    dr)
```

classify-load

```
(defun classify-load (load class)
    (freplace load 'ako '$value class)
    (freplace load 'class '$value class)
    load)
```

reverse-s-d

```
(defun reverse-s-d (tran)
    (let ((new-src (fvo tran 'd-node))(new-drn (fvo tran 's-node)))
      (freplace tran 'd-node '$value new-drn)
      (freplace tran 's-node '$value new-src)
     t))
```

;*******************FILE: S-P-FET.SL *******

find-parallel-fets

```
(defun find-parallel-fets ()
  (let ((tran-list (fvalues-only 'elements1 'driver))
        (trans nil)(trans1 nil)(struct nil)
        (n1 nil)(n2 nil))

     (patom "find-parallel-fets")(terpri)
     (while (setq trans (car tran-list))
         (setq tran-list (cdr tran-list))
         (setq n1 (fvalue-only trans 's-node))
         (setq n2 (fvalue-only trans 'd-node))
         (cond((setq trans1
                   (caadadr(solve
                    `(test(and(?driver status free(member ?driver tran-list))
                            (?driver s-node ?x)
                            (?driver d-node ?y))
                        (or (and(equal ?x ,n1)(equal ?y ,n2))
                            (and(equal ?x ,n2)(equal ?y ,n1))) ) )))
               (setq tran-list (delete trans1 tran-list))
               (setq struct (finstantiate 'parallel-struct))
               (patom struct)(terpri)
               (fput struct 'class '$value 'parallel)
               (fput struct 'trans '$value trans)
               (fput struct 'trans '$value trans1)
               (fput struct 'node-1 '$value n1)
               (fput struct 'node-2 '$value n2)
               (fput struct 'string-1 '$value 1)
               (freplace struct 'l-div-w '$value
                          (max(fvalue-only trans 'l-div-w)
                              (fvalue-only trans1 'l-div-w)))
               (while (setq trans1 (caadadr(solve
                    `(test(and(?driver status free(member ?driver tran-list))
                            (?driver s-node ?x)
                            (?driver d-node ?y))
                        (or (and(equal ?x ,n1)(equal ?y ,n2))
                            (and(equal ?x ,n2)(equal ?y ,n1))) ) )))
                   (setq tran-list (delete trans1 tran-list))
                   (freplace struct 'l-div-w '$value
                              (max(fvalue-only struct 'l-div-w)
                                  (fvalue-only trans1 'l-div-w)))
                   (fput struct 'trans '$value trans1)) ))) ))
```

find-series-fets

```
(defun find-series-fets ()
   (let ((nodes (fvalues-only 'elements1 'nodes))
         (tmp nil)(start-node nil)(current-node nil)
         (next-node nil)(tran-list nil)(tran nil)(struct nil))

     (patom "find-series-fets")(terpri)
     (setq tmp nodes)

     (while (setq start-node (car tmp))
         (setq tmp (cdr tmp))
         (cond((series-fet-condition start-node)(next)))
         (cond ((null (setq tran-list (find-trans start-node)))(next)))

         (setq current-node start-node)
         (while (setq tran (car tran-list))
             (setq tran-list (cdr tran-list))
             (setq next-node (find-next-node tran current-node))

             (cond ((null (series-fet-condition next-node))(next)))
```

```
            (setq struct (finstantiate 'series-struct))
            (patom struct)(terpri)
            (fput struct 'node-1 '$value start-node)
            (fput struct 'class '$value 'series)
            (fput struct 'trans '$value tran)
            (freplace struct 'l-div-w '$value (fvalue-only tran 'l-div-w))
            (freplace struct 'string-l '$value 1)
            (while (series-fet-condition next-node)
                (setq current-node next-node)
                (setq tmp (delete current-node tmp))
                (setq tran (caadadr(solve
                        '(test(and(?driver status free)
                                (?driver s-node ?sn)
                                (?driver d-node ?dn))
                            (or(equal ?sn ,current-node)
                              (equal ?dn ,current-node))))))
                (fput struct 'trans '$value tran)
                (freplace struct 'l-div-w '$value (+(fvalue-only tran 'l-div-w)
                                                (fvalue-only struct 'l-div-w)))
                (freplace struct 'string-l '$value (+ 1
                                            (fvalue-only struct 'string-l)))
                (setq next-node (find-next-node tran current-node))
                (freplace struct 'node-2 '$value next-node)) ))))
```

series-fet-condition

```
(defun series-fet-condition (node)
   (null(or (>(length (fvalues-only (make-node-name node) 'trans)) 2)
          (solve '(?transistor g-node ,node))
          (solve '(?load s-node ,node))
          (solve '(?load d-node ,node))
          (solve '(?precharger s-node ,node))
          (solve '(?pup-driver s-node ,node))
          (solve '(?drn-boot s-node ,node))
          (equal node 0)
          (solve '(?xi-driver in node ,node))
          (single-connection node)
          (solve '(?active-two-port-elemen* ,s-node ,node))
          (solve '(?active-two-port-element neg-node ,node)))))
```

find-next-node

```
(defun find-next-node (tran node)
   (fvalue-only tran (cond((equal node (fvalue-only tran 'd-node)) 's-node)
                       (t 'd-node))))
```

find-trans

```
(defun find-trans (node)
   (let ((tran-list (caadadr (solve
                  '(test(and(?driver status free)
                          (?driver s-node ?sn)
                          (?driver d node ?dn))
                      (or(equal ?sn ,node)(equal ?dn ,node))))))
        (tran nil))
     (cond((null tran-list) nil)
        (t(setq tran-list (list tran list))
          (while (setq tran (caadadr (solve
                    '(test(and(?driver status free(not
                                    (member ?driver tran-list)))
                            (?driver s-node ?sn)
                            (?driver d-node ?dn))
                        (or(equal ?sn ,node)(equal ?dn ,node))))))
              (setq tran-list (cons tran tran-list))) ))
      tran-list ))
```

;********************FILE: COMB-STRUCTS.SL ****

combine-structs

```
(defun combine-structs()
  (let ((keep-looking t))
  (while keep-looking
       (setq keep-looking nil)
       (combine-parallel-structs)
       (combine-series-structs))))
```

combine-parallel-structs

```
(defun combine-parallel-structs ()
  (let ((struct-list (nconc(get-free-drivers)(get-free-structs)))
       (super-struct nil)(struct nil)(struct1 nil)(n1 nil)(n2 nil))

  (patom "combine-parallel-structs")(terpri)
  (while (setq struct (car struct-list))
      (setq struct-list (cdr struct-list))
      (cond ((flinkp 'ako struct 'driver)
             (setq n1 (fvalue-only struct 'd-node))
             (setq n2 (fvalue-only struct 's-node)))
            (t(setq n1 (fvalue-only struct 'node-1))
              (setq n2 (fvalue-only struct 'node-2))))
      (cond((setq struct1 (caadadr(solve
                `(test(and(?struct node-1 ?x (member ?struct struct-list))
                         (?struct node-2 ?y))
                    (or(and(equal ,n1 ?x)(equal ,n2 ?y))
                       (and(equal ,n1 ?y)(equal ,n2 ?x)))))))
            (setq struct-list (delete struct1 struct-list))
            (setq super-struct (finstantiate 'super-struct))
            (patom super-struct)(terpri)
            (fput super-struct 'class '$value 'parallel)
            (fput super-struct 'substruct '$value struct)
            (fput super-struct 'substruct '$value struct1)
            (fput super-struct 'node-1 '$value n1)
            (fput super-struct 'node-2 '$value n2)
            (fput super-struct 'string-1 '$value
                       (max(fvalue-only struct 'string-1)
                           (fvalue-only struct1 'string-1)))
            (freplace super-struct 'l-div-w '$value
                       (max(fvalue-only struct 'l-div-w)
                           (fvalue-only struct1 'l-div-w)))
            (while (setq struct1 (caadadr(solve
                 `(test(and(?struct node-1 ?x (member ?struct struct-list))
                           (?struct node-2 ?y))
                     (or(and(equal ,n1 ?x)(equal ,n2 ?y))
                        (and(equal ,n1 ?y)(equal ,n2 ?x)))))))
                (setq struct-list (delete struct1 struct-list))
                (freplace super-struct 'l-div-w '$value
                         (max(fvalue-only super-struct 'l-div-w)
                             (fvalue-only struct1 'l-div-w)))
                (freplace super-struct 'string-1 '$value
                         (max(fvalue-only super struct 'string-1)
                             (fvalue-only struct1 'string-1)))
                (fput super-struct 'substruct '$value struct1)) ))) ))
```

get-free-drivers

```
(defun get-free-drivers ()
  (let ((d-list nil)(dr nil))
     (setq d-list (car (solve '(?driver status free))))
     (cond ((null d-list) nil)
           (t(setq d-list (list d-list))
            (while (setq dr (car (solve
                   '(?driver status free(not(member ?driver d-list))))))
```

```
                (setq d-list (cons dr d-list)))
            d-list))))
```

get-free-structs

```
(defun get-free-structs ()
   (let ((s-list nil)(st nil))
      (setq s-list (car (solve '(?struct status free))))
      (cond ((null s-list) nil)
            (t(setq s-list (list s-list))
              (while (setq st (car (solve
                          '(?struct status free(not(member ?struct s-list))))))
                     (setq s-list (cons st s-list)))
              s-list))))
```

combine-series-structs

```
(defun combine-series-structs ()
   (let ((nodes (fvalues-only 'elements1 'nodes))
         (tmp nil)(start-node nil)(current-node nil)
         (next-node nil)(struct-list nil)(super-struct nil)(struct nil))

      (patom "combine-series-structs")(terpri)
      (setq tmp nodes)
      (while (setq start-node (car tmp))
           (setq tmp (cdr tmp))
           (cond((series-struct-condition start-node nil)(next)))
           (cond ((null (setq struct-list (nconc (find-trans start-node)
                                                  (find-structs start-node nil))))
                  (next)))
           (setq current-node start-node)
           (while (setq struct (car struct-list))
                (setq struct-list (cdr struct-list))
                (setq next-node (find-next-struct-node struct current-node))

                (cond ((null (series-struct-condition next-node nil))(next)))

                (setq super-struct (finstantiate 'super-struct))
                (patom super-struct)(terpri)
                (setq keep-looking t)
                (fput super-struct 'class '$value 'series)
                (fput super-struct 'node-1 '$value start-node)
                (fput super-struct 'substruct '$value struct)
                (freplace super-struct 'l-div-w '$value
                          (fvalue-only struct 'l-div-w))
                (fput super-struct 'string-1 '$value
                          (fvalue-only struct 'string-1))
                (while (series-struct-condition next-node super-struct)
                     (setq current-node next-node)
                     (setq tmp (delete current-node tmp))
                     (setq struct (car(find-structs current-node super-struct)))
                     (fput super-struct 'substruct '$value struct)
                     (freplace super-struct 'l-div-w '$value
                               (+ (fvalue-only struct 'l-div-w)
                                  (fvalue-only super-struct 'l-div-w)))
                     (freplace super-struct 'string-1 '$value
                               (+ (fvalue-only struct 'string-1)
                                  (fvalue-only super-struct 'string-1)))
                     (setq next-node (find-next-struct-node struct current-node))
                     (freplace super-struct 'node-2 '$value next-node)) ))))
```

series-struct-condition

```
(defun series-struct-condition (node ss)
   (let((s1 nil)(d1 nil))
     (null(or (equal node 0)
              (single-connection node)
```

```
        (solve `(?transistor g-node ,node))
        (solve `(?load s-node ,node))
        (solve `(?load d-node ,node))
        (solve `(?precharger s-node ,node))
        (solve `(?pup-driver s-node ,node))
        (solve `(?drn-boot s-node ,node))
        (solve `(?active-two-port-element pos-node ,node))
        (solve `(?xi-driver in-node ,node))
        (solve `(?active-two-port-element neg-node ,node))
       (solve `(?driver s-node ,node (equal (fvo ?driver 'class) 'xfer)))
       (solve `(?driver d-node ,node (equal (fvo ?driver 'class) 'xfer)))
        (>(setq dl (length (solve-all
                  `(test(and(?driver status free)
                          (?driver s-node ?sn)
                          (?driver d-node ?dn))
                   (or (equal ?sn ,node)(equal ?dn ,node)))))) 2)
        (>(setq sl (length (solve-all
                `(test(and(?struct status free (neq ?struct ',ss))
                        (?struct node-1 ?n1)
                        (?struct node-2 ?n2))
                  (or (equal ?n1 ,node)(equal ?n2 ,node)))))) 2)
        (>(+ dl sl) 2) ))))
```

find-next-struct-node

```
(defun find-next-struct-node (struct node)
  (cond((flinkp 'ako struct 'struct)
        (fvalue-only struct
              (cond((equal node (fvalue-only struct 'node-1)) 'node-2)
                   (t 'node-1))))
       (t(fvalue-only struct
              (cond((equal node (fvalue-only struct 's-node)) 'd-node)
                   (t 's-node))))))
```

find-structs

```
(defun find-structs (node ss)
  (let ((struct-list (caadadr (solve
              `(test(and(?struct status free (neq ?struct ',ss))
                      (?struct node-1 ?x)
                      (?struct node-2 ?y))
                 (or (equal ?x ,node)(equal ?y ,node))))))
     (struct nil))

    (cond((null struct-list) nil)
         (t(setq struct-list (list struct-list))
          (while (setq struct (caadadr (solve
                  `(test(and(?struct status free
                             (and(neq ?struct ',ss)
                                 (not(member ?struct struct-list))))
                          (?struct node-1 ?x)
                          (?struct node-2 ?y))
                   (or (equal ?x ,node)(equal ?y ,node))))))
             (setq struct-list (cons struct struct-list))) ))
    struct-list ))
```

```
;*********************FILE: INV-RULES.SL *******

; regular inverter
(rule find-simple-inverter backward-chain-rule
    (type(elements ?elements))
    (premise (and(?reg-driver status free)
                  (?load status free)
                  (?reg-driver s-node 0)
                  (?reg-driver d-node ?ddn)
                  (?reg-driver g-node ?dgn)
                  (?load s-node ?ddn)
                  (?load d-node ?ldn)
                  (?supply pos-node ?ldn)
                  (?elements dummy 1)))
     (conclusion (?elements inverter ^(make-inverter
                                              ?reg-driver ?load ?supply))))
;inverter with dynamic input
(rule find-dynamic-input-inverter backward-chain-rule
    (type(elements ?elements))
    (premise (and(?xi-driver status free)
                  (?load status free)
                  (?xi-driver s-node 0)
                  (?xi-driver d-node ?ddn)
                  (?xi-driver g-node ?dgn)
                  (?load s-node ?ddn)
                  (?load d-node ?ldn)
                  (?supply pos-node ?ldn)
                  (?elements dummy 1)))
     (conclusion (?elements inverter ^(make-xi-inverter
                                          ?xi-driver ?load ?supply ))))

; inverter with driver below load -- used as drain-boot predriver
(rule find-clkout-inverter-1 backward-chain-rule
   (type(load ?ld)(reg-driver ?dr2)(xi-driver ?dr1)(node ?no)
        (elements ?elements)(supply ?sup)(load ?ld))
   (premise(and(?elements dummy 1)
                (?dr1 s-node 0)
                (?dr1 d-node ?d1)
                (?dr1 status free)
                (?dr2 s-node ?d1)
                (?dr2 d-node ?d2)
                (?dr2 status free)
                (?dr2 g-node ?g2)
                (?no number ?g2)
                (?no class ?cl (equal ?cl 'always-clocked))
                (?ld d-node ?ldn)
                (?sup pos-node ?ldn)
                (?ld s-node ?d2)
                (?ld status free)))
   (conclusion(?elements inverter
                           ^(make-clkout inverter ?dr1 ?ld ?sup ?dr2))))

(rule find-clkout-inverter-2 backward-chain rule
      (type(load ?ld)(reg driver ?dr2)(xi-driver ?dr1)(node ?no)
           (elements ?elements)(supply ?sup)(load ?ld))
   (premise(and(?elements dummy 1)
                (?dr1 s-node 0)
                (?dr1 d-node ?d1)
                (?dr1 status free)
                (?dr2 d-node ?d1)
                (?dr2 s-node ?d2)
                (?dr2 status free)
                (?dr2 g-node ?g2)
                (?no number ?g2)
```

```
            (?no class ?cl (equal ?cl 'always-clocked))
            (?ld d-node ?ldn)
            (?sup pos-node ?ldn)
            (?ld s-node ?d2)
            (?ld status free)))
   (conclusion(?elements inverter
         ^(make-clkout-inverter ?dr1 ?ld ?sup ?dr2))))

;;;; inverter checks ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(rule check-beta-ratio-1 forward-chain-rule
    (type (inverter ?inverter))
    (premise (?inverter beta-ratio ?br0
                      (< ?br0 (req-br (fvalue-only ?inverter 'pull-up)))))
    (conclusion (inv-errors beta-ratio ?inverter)))

; clocks to all inputs must be the same or error
(rule check-clkout-inverter-1 forward-chain-rule
    (type(clkout-inverter ?inv)(driver ?xf ?cg)(clock ?clk1 ?clk2))
    (premise(and(?inv trigger t)
            (?inv xfer-gate ?xf)
            (?xf g-node ?xfg)
            (?clk1 pos-node ?xfg)
            (?inv clkgate ?cg)
            (?cg g-node ?cgg)
            (?clk2 pos-node ?cgg(neq ?clk2 ?clk1))))
    (conclusion(inv-errors input-clocking ?inv)))

(rule check-clkout-inverter-2 forward-chain-rule
 (type(clkout-inverter ?inv)(load ?ld)(driver ?xf ?cg)
     (clock ?clk1 ?clk2 ?clk3))
 (premise(and(?inv trigger t)
          (?inv xfer-gate ?xf)
          (?xf g node ?xfg)
          (?clk1 pos node ?xfg)
          (?inv clkgate ?cg)
          (?cg g-node ?cgg)
          (?clk2 pos-node ?cgg)
          (?inv pull-up ?ld)
          (?ld g-node ?lgn)
          (?clk3 pos-node ?lgn (or(neq ?clk3 ?clk2)(neq ?clk3 ?clk1)))))
 (conclusion(inv-errors input-clocking ?inv)))
```

;********************FILE: INV-FUNCS.SL *******

std-inv-put

```
(defun std-inv-put (driver load supply)
   (fput inv-name 'pull-down '$value driver)
   (fput inv-name 'pull-up   '$value load)
   (freplace driver 'status '$value 'in-use)
   (freplace load 'status '$value 'in-use)
   (fput inv-name 'supply '$value supply)
   (fput inv-name 'out-node '$value (fvo driver 'd-node))
   (fput inv-name 'supply-node '$value (fvo load 'd-node))
   (fput inv-name 'struct '$value driver))
```

make-inverter

```
(defun  make-inverter (driver load supply)
   (let ((inv-name (finstantiate 'reg-inverter)))
        (patom inv-name)(terpri)
        (std-inv-put driver load supply)
        (fput inv-name 'in-node '$value (fvo driver 'g-node))
        (fput inv-name 'beta-ratio '$value (calc-br driver load))
        inv-name))
```

make-xi-inverter

```
(defun  make-xi-inverter (driver load supply )
   (let ((inv-name (finstantiate 'xi-inverter))
         (xfer-gate (fvo driver 'xfer-gate)))
        (patom inv-name)(terpri)
        (std-inv-put driver load supply)
        (fput inv-name 'xfer-gate '$value xfer-gate)
        (fput inv-name 'in-node '$value
                  (cond ((equal (fvo driver 'g-node)(fvo xfer-gate 's-node))
                         (fvo xfer-gate 'd-node))
                        (t(fvo xfer-gate 's-node))))
        (fput inv-name 'beta-ratio '$value (calc-br driver load))
        inv name))
```

make-clkout inverter

```
(defun  make-clkout-inverter (driver load supply clkgate)
   (let ((inv name (finstantiate 'clkout-inverter))
         (xfer gate (fvo driver 'xfer-gate)))
        (patom inv-name)(terpri)
        (std inv put driver load supply)
        (fput inv-name 'xfer-gate '$value xfer-gate)
        (fput inv name 'in-node '$value
              (cond ((equal (fvo driver 'g-node)(fvo xfer-gate 's-node))
                     (fvo xfer-gate 'd-node))
                    (t(fvo xfer-gate 's-node))))
        (fput inv-name 'clkgate '$value clkgate)
        (fput inv-name 'beta-ratio '$value (calc-br-1 driver load clkgate))
        (fput inv-name 'trigger '$value t)
        inv name))
```

calc-br

```
(defun calc br (p-down p-up)
     (* (/ 1 (fvo p-down 'l-div-w))(fvo p-up 'l-div-w)))
```

calc-br-1

```
(defun calc-br-1 (p-down p-up clkgate)
  (* (/ 1 (fvo p-down 'l-div-w))
     (+ (fvo p-up 'l-div-w)(* (fvo clkgate 'l-div-w)
                              (fvo '*g-con 'dr-eq-ratio))) ))
```

req-br

```
(defun req-br (load)
   (let((class (fvo load 'class)))
      (fvo '*g-con (implode `(b r - ,@(explode class))))))
```

```
;********************FILE: F-F-RULESSL *******

; Max Driver Length
(rule funny-fet-1 backward-chain-rule
     (type(errors))
     (premise (?driver length ?l
                    (> ?l (fvalue-only '*g-con 'mx-dr-l))))
     (conclusion(funny-fet max-driver-length ?driver)))

; Min Driver Width
(rule funny-fet-2 backward-chain-rule
     (type(errors))
     (premise (?driver width ?w
                    (< ?w (fvalue-only '*g-con 'mn-dr-w))))
     (conclusion(funny-fet min-driver-width ?driver)))

; Min Load Width
(rule funny-fet-3 backward-chain-rule
     (type(errors))
     (premise (?load width ?w
                    (< ?w (fvalue-only '*g-con 'mn-ld-w))))
     (conclusion(funny-fet min-load-width ?load)))

; Min Load Length
(rule funny-fet-4 backward-chain-rule
     (type(errors))
     (premise (?load length ?l
                    (< ?l (fvalue-only '*g-con 'mn-ld-l))))
     (conclusion(funny-fet min-load-length ?load)))

; Max Mos-Cap Length
(rule funny-fet-5 backward-chain rule
     (type(errors))
     (premise (?mos-cap length ?l
                    (> ?l (fvalue only '*g-con 'mx-cap-l))))
     (conclusion(funny fet max cap length ?mos-cap)))

; Single Connection
(rule funny-fet-6 backward-chain rule
     (type(errors))
     (premise( or (?transistor s node ?sn (single-connection ?sn))
                  (?transistor g node ?gn (single-connection ?gn))
                  (?transistor d node ?dn (single-connection ?dn))))
      (conclusion(funny-fet single connection ?transistor)))

; Gate-Only Node Connection
(rule funny-node-1 backward-chain rule
     (type (errors)(node ?node))
     (premise (known ?node gate ?g (and(null (fvso ?node 'src-drn))
                                       (null (fvso ?node 'supply))
                                       (null (fvso ?node 'clock)))))
     (conclusion(funny node gate only ?node)))

; Supply-gate-only Node connection
(rule funny-node 2 backward-chain-rule
     (type (errors)(node ?node))
     (premise (test(and (known ?node supply ?s)
                        (known ?node gate ?g))
                  (null(fvso ?node 'src-drn))))
     (conclusion (funny-node supply-gate-only ?node)))

; clock-supply-short Node connection
(rule funny-node-3 backward-chain-rule
     (type(errors))
```

```
    (premise (and(?supply pos-node ?spn)
                 (?supply neg-node ?snn)
                 (?clock pos-node ?spn)
                 (?clock neg-node ?snn)))
    (conclusion(funny-node clock-supply-short ^(make-node-name ?spn))))

; single-conection
(rule funny-node-4 backward-chain-rule
    (type(errors))
    (premise (?supply pos-node ?spn (single-connection ?spn)))
    (conclusion (funny-node single-connection ^(make-node-name ?spn))))
(rule funny-node-5 backward-chain-rule
    (type(errors))
    (premise (?clock pos-node ?spn (single-connection ?spn)))
    (conclusion (funny-node single-connection ^(make-node-name ?spn))))
```

```
;*********************FILE: GATE-RULES.SL *******

;;;;;; static gate classification

(rule find-nor-rule backward-chain-rule
     (type(elements ?elements)(parallel-struct ?p-s))
     (premise(test(and(?load status free)
                        (?p-s status free)
                        (?load d-node ?ldn)
                        (?load s-node ?lsn)
                        (?supply pos-node ?ldn)
                        (?p-s node-1 ?psn1)
                        (?p-s node-2 ?psn2)
                        (?elements dummy 1))
                   (or(and(equal ?psn1 0)(equal ?psn2 ?lsn))
                      (and(equal ?psn1 ?lsn)(equal ?psn2 0)))))
     (conclusion(?elements static-gate ^(make-gate ?load ?p-s 'nor-gate))))

(rule find-nand-rule backward-chain-rule
     (type(elements ?elements)(series-struct ?s-s))
     (premise(test(and(?load status free)
                        (?s-s status free)
                        (?load d-node ?ldn)
                        (?load s-node ?lsn)
                        (?supply pos-node ?ldn)
                        (?s-s node-1 ?ss1)
                        (?s-s node-2 ?ss2)
                        (?elements dummy 1))
                   (or(and(equal ?ss1 0)(equal ?ss2 ?lsn))
                      (and(equal ?ss2 0)(equal ?ss1 ?lsn)))))
     (conclusion(?elements static-gate ^(make-gate ?load ?s-s 'nand-gate))))

(rule find-other-gate-rule backward-chain-rule
     (type(elements ?elements)(super-struct ?s-s))
     (premise(test(and(?load status free)
                        (?s-s status free)
                        (?load d-node ?ldn)
                        (?load s-node ?lsn)
                        (?supply pos-node ?ldn)
                        (?s-s node-1 ?ss1)
                        (?s-s node-2 ?ss2)
                        (?elements dummy 1))
                   (or(and(equal ?ss1 0)(equal ?ss2 ?lsn))
                      (and(equal ?ss2 0)(equal ?ss1 ?lsn)))))
     (conclusion(?elements static-gate ^(make-gate ?load ?s-s 'other-gate))))

(rule find-static-xc-xor backward-chain-rule
   (type (elements ?elements)(xc-xor-struct ?xc)(load ?ld)(supply ?sup))
   (premise(and(?elements dummy 1)
               (?ld s node ?sn)
               (?ld d node ?dn)
               (?sup pos node ?dn)
               (?ld status free)
               (?xc out node ?sn)
               (?xc status free)))
   (conclusion (?elements static gate ^(make-gate ?ld ?xc 'static xc xor))))
;;;;;;;; dynamic gate classification

(rule find-d-gate-1 backward-chain-rule
     (type(elements ?el)(precharger ?pre)(struct ?st))
     (premise (test(and(?el dummy 1)
                        (?st status free)
                        (?pre status free)
                        (?pre s-node ?sn)
```

```
                    (?st node-1 ?n1)
                    (?st node-2 ?n2) )
                (or (and (equal ?n1 ?sn)(equal ?n2 0))
                    (and (equal ?n2 ?sn)(equal ?n1 0))) ))
    (conclusion (?el dynamic-gate ^(make-gate ?pre ?st 'dynamic-gate))))

(rule find-d-gate-2 backward-chain-rule
    (type(elements ?elements)(precharger ?precharger)(driver ?driver))
    (premise (and(?elements dummy 1)
                (?driver status free)
                (?precharger status free)
                (?precharger s-node ?sn)
                (?driver d-node ?sn)
                (?driver s-node 0)))
    (conclusion (?elements dynamic-gate
                    ^(make-gate ?precharger ?driver 'dynamic-gate))))

(rule find-dynamic-xc-xor backward-chain-rule
    (type(elements ?elements)(precharger ?precharger)(xc-xor-struct ?xc))
    (premise (and(?elements dummy 1)
                (?xc status free)
                (?precharger status free)
                (?precharger s-node ?sn)
                (?xc out-node ?sn)))
    (conclusion (?elements dynamic-gate
                    ^(make-gate ?precharger ?xc 'dynamic-xc-xor))))

;;;;;;; static gate checks ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; checks for string-length
(rule static-gate-check-1 forward-chain-rule
    (type (static-gate ?ga))
    (premise (?ga trigger t (>(fvalue-only
                                (fvalue-only ?ga 'struct) 'string-l)
                            (fvalue-only "*-con 'st-nand-sl) )))
    (conclusion (gate errors nand-length ?ga))

; performs beta-ratio checks
(rule static-gate-check-2 forward-chain-rule
    (type (static-gate ?ga))
    (premise (?ga trigger t (gate-br-error ?ga)))
    (conclusion (gate-errors beta-ratio ?ga)))

; checks for feedback on static gates whose outputs are clocked-low
(rule feed-back-check backward-chain rule
   (type (static-gate ?ga)(node ?no)(xi-driver ?dr)(super-buffer ?sup))
   (premise(and(?ga out-node ?num)
               (?no number ?num)
               (?no class clocked-low)
               (?ga pull-down ?dr (akop ?dr 'xi-driver))
               (unknown ?dr fb tran ?fbt)
               (unknown ?sup driver ?ga)))
   (conclusion(gate-errors b feedback-desirable ?ga)))

; checks for correct clocking on a clocked-low static-gate
(rule static-gate-input clock cneck backward-chain-rule
   (type (static gate ?ga)(xi-driver ?dr)(driver ?xg)(node ?no ?no1))
   (premise(test(and(?ga trigger t)
                   (?ga out-node ?n)
                   (?no number ?n)
                   (?no class clocked-low)
                   (?ga pull-down ?dr (akop ?dr 'xi-driver))
                   (?dr xfer-gate ?xg)
                   (?xg g-node ?gn)
```

```
                  (?no1 number ?gn)
                  (?no1 class ?x (or (equal ?x 'always-clocked)
                                     (equal ?x 'conditional-clocked))))
        (neq (fvo ?no1 'aspect)(implode (cddr (explode (fvo ?no 'aspect))))))
    (conclusion(gate-errors-b input-clocking-error ?ga)))

;;;;;;;;;;; dynamic gate and structure checks ;;;;;;;;;;;;;

;error if precharge and true phases are equal
;error if pull-down structure is clocked on both phases
;error if pull-down structure is always held high
(rule dynamic-gate-check-1 forward-chain-rule
     (type (dynamic-gate ?ga))
     (premise (test (and (?ga trigger t)
                         (?ga pre-phase ?pp)
                         (?ga true-phase ?tp))
               (or(equal ?tp ?pp)
                  (equal ?tp 'error)
                  (equal ?tp '*high) )))
     (conclusion(gate-errors dynamic-clocking-1 ?ga)))


; error if pull-down structure is always low on a given clock phase
;  (ie on a given phase a pull down structure always pulls the precharge low)

(rule dynamic-gate-check-2 forward-chain-rule
     (type(dynamic-gate ?ga)(struct ?st))
     (premise (and (?ga trigger t)
                   (?ga struct ?st)
                   (?st clk-class hard)))
     (conclusion (gate-errors dynamic-clocking-2 ?ga)))


; race condition if a dynamic gate pull-down-structure has an xi-driver
;        and the xi-driver transfer gate is clocked on the true-phase
;        of the gate

(rule dynamic-gate-check-3 forward-chain-rule
     (type (dynamic-gate ?ga)(struct ?st)(xi-driver ?dr)(driver ?xf))
     (premise(test(and (?ga trigger t)
                       (?ga struct ?st)
                       (?st trans ?dr(akop ?dr 'xi-driver))
                       (?dr xfer-gate ?xf)
                       (?ga true-phase ?tp)
                       (knowable ?xf clk-input ?clkin))
              (equal ?tp ?clkin)))
      (conclusion (gate-errors race-condition ?ga)))

;;;;;   flag a dynamic gate whose pulldown stage is never clocked
(rule dynamic gate-check-4 forward chain-rule
     (type(dynamic gate ?ga)(struct ?st))
     (premise (and (?ga trigger t)
                   (?ga true phase nil)))
     (conclusion (gate errors dynamic clocking 4 ?ga)))


; probable dynamic clocking problem if the is precharged node
;    and there is a driver connected to this node which is clocked
;    on the precharge phase of the precharger
; or driver's gate is always -high
(rule dynamic-clocking-rule-1 backward-chain-rule
     (type(precharger ?pre)(driver ?dr))
     (premise(test(and (?pre s-node ?sn)
```

```
                (?pre pre-phase ?pp)
                (?dr d-node ?sn)
                (?dr s-node 0)
                (?dr clk-input ?clkin))
        (or(equal ?pp ?clkin)(equal ?clkin '*high))))
(conclusion(funny-node clocking-flag ^(make-node-name ?sn))))
```

;********************FILE: GATE-FUNCS.SL *******

make-gate

```
(defun make-gate (load struct g-type)
   (let ((gate (finstantiate g-type))(tran nil)
        (tran-list (get-trans struct)))
     (patom gate)(terpri)
     (fput gate 'out-node '$value (fvalue-only load 's-node))
     (fput gate 'struct '$value struct)
     (fput gate 'supply-node '$value (fvalue-only load 'd-node))
     (fput gate 'pull-up '$value load)
     (fput gate 'beta-ratio '$value (calc-br struct load))
     (freplace load 'status '$value 'in-use)
     (freplace struct 'status '$value 'in-use)
     (while (setq tran (car tran-list))
         (setq tran-list (cdr tran-list))
         (fput gate 'pull-down '$value tran)
         (fput gate 'in-node '$value (fvalue-only tran 'g-node)))
     (cond ((or(equal g-type 'dynamic-gate)(equal g-type 'dynamic-xc-xor))
            (fput gate 'pre-phase '$value (fvo load 'pre-phase))
            (fput gate 'true-phase '$value (fvo struct 'clk-input))))
      (freplace gate 'trigger '$value t)
      gate ))
```

gate-br-error

```
(defun gate-br-error (gate)
      (< (fvalue-only gate 'beta-ratio)
         (get-req-br (fvalue-only gate 'pull-up))))
```

get-req-br

```
(defun get-req-br (load)
  (let((class (fvalue-only load 'class)))
   (fvalue-only '*g-con (implode '(b r - ,@(explode class))))))
```

```
;********************FILE: COUPLE-RULES.SL *****

(rule xi-driver-coupling-rule-1 forward-chain-rule
    (type (xi-driver ?xi-dr))
    (premise(and(?xi-dr trigger t)
                (?xi-dr ga-xfw-ratio ?x (< ?x (fvo '*g-con 'mn-dr-ga-xf-w)))))
    (conclusion (coupling-errors xi-driver-coupling ?xi-dr)))

(rule xi-driver-coupling-rule-2 forward-chain-rule
    (type (xi-driver ?xi-dr))
    (premise (and(?xi-dr trigger t)
                (?xi-dr s-node ?sn (neq ?sn 0))))
    (conclusion(coupling-errors xi-driver-coupling-1 ?xi-dr)))
```

```
;*********************FILE: CLKING-RULES.SL******

(rule determine-struct-clocking-1 forward-chain-rule
   (type (struct ?struct)(driver ?dr))
   (premise (and (?struct class ?class)
                 (?struct trans ?dr)
                 (knowable ?dr clk-input ?clkin)))
   (conclusion(and (replace ?struct clk-input
                       ^(det-clk-input ?struct ?class ?dr ?clkin))
                   (replace ?struct clk-class
                       ^(det-clk-class ?struct ?class ?dr ?clkin)))))

(rule determine-struct-clocking-2 forward-chain-rule
   (type (struct ?struct)(driver ?dr))
   (premise (and (?struct class ?class)
                 (?struct trans ?dr)
                 (unknowable ?dr clk-input ?clkin)))
   (conclusion(and (replace ?struct clk-input
                       ^(det-clk-input ?struct ?class ?dr nil))
                   (replace ?struct clk-class
                       ^(det-clk-class ?struct ?class ?dr nil)))))

(rule determine-struct-clocking-3 forward-chain-rule
   (type (struct ?struct)(driver ?dr))
   (premise (and (?struct class ?class)
                 (?struct substruct ?dr)
                 (knowable ?dr clk-input ?clkin)))
   (conclusion(and (replace ?struct clk-input
                       ^(det-clk-input ?struct ?class ?dr ?clkin))
                   (replace ?struct clk-class
                       ^(det-clk-class ?struct ?class ?dr ?clkin)))))

(rule determine-struct-clocking-4 forward-chain-rule
   (type (struct ?struct)(driver ?dr))
   (premise (and (?struct class ?class)
                 (?struct substruct ?dr)
                 (unknowable ?dr clk-input ?clkin)))
   (conclusion(and (replace ?struct clk-input
                       ^(det-clk-input ?struct ?class ?dr nil))
                   (replace ?struct clk-class
                       ^(det-clk-class ?struct ?class ?dr nil)))))


(rule determine-struct clocking-5 forward-chain-rule
   (type (struct ?struct ?st))
   (premise (and (?struct class ?class)
                 (?struct substruct ?st)
                 (?st clk input ?clkin)))
   (conclusion(and (replace ?struct clk-input
                       ^(det clk-input ?struct ?class ?st ?clkin))
                   (replace ?struct clk class
                       ^(det clk-class ?struct ?class ?st ?clkin)))))

(rule determine struct clocking 6 forward chain rule
   (type (struct ?struct ?st))
   (premise (and (?struct class ?class)
                 (?struct substruct ?st)
                 (unknown ?st clk-input ?clkin)))
   (conclusion(and (replace ?struct clk input
                       ^(det-clk-input ?struct ?class ?st nil))
                   (replace ?struct clk-class
                       ^(det-clk-class ?struct ?class ?st nil)))))

; driver clock input classification
```

```
(rule clk-input-rule-1  backward-chain-rule
    (type (driver ?dr)(clock ?ck))
    (premise (and (?dr g-node ?gn)
                  (?ck pos-node ?gn)))
    (conclusion (?dr clk-input ^(define-clking-class ?dr ?ck 'hard))))

(rule clk-input-rule-2 backward-chain-rule
     (type (driver ?dr)(node ?no))
     (premise (and(?dr g-node ?gn)
                  (?no number ?gn)
                  (?no class ?x (equal ?x 'always-high))))
     (conclusion(?dr clk-input ^(define-clking-class ?dr '"high 'hard))))

(rule clk-input-rule-3 backward-chain-rule
     (type (driver ?dr)(node ?no))
     (premise(and(?dr g-node ?gn)
                 (?no number ?gn)
                 (?no class ?x (equal ?x 'conditional-clocked))))
     (conclusion(?dr clk-input ^(define-clking-class ?dr (fvo ?no 'aspect)
                                                     'conditional))))
(rule clk-input-rule-4 backward-chain-rule
     (type(driver ?dr)(node ?no))
     (premise(and(?dr g-node ?gn)
                 (?no number ?gn)
                 (?no class ?x (equal ?x 'clocked-low))))
     (conclusion(?dr clk-input ^(define-clking-class ?dr (fvo ?no 'aspect)
                                                     'clocked-low))))

; node classifcation rules
(rule always-high-node-rule-1 backward-chain-rule
     (type(driver ?dr)(load ?ld)(supply ?sup)(node ?no0 ?no1))
     (premise (and(unknown ?no0 class always-high)
                  (?no0 number ?gn(neq ?gn 0))
                  (?no0 trans ?t (= (length (fvso ?no0 'trans)) 2 ))
                  (?dr d node ?dn)
                  (?dr s-node ?gn)
                  (?sup pos-node ?dn)
                  (?dr g-node ?gn1)
                  (?no1 number ?gn1)
                  (?no1 class-1 ?x (equal ?x 'always-high))))
    (conclusion(?no0 class ^(make-node-class ?no0 'always-high nil nil) )))
                                ;       node   class   aspect switch

(rule always-high-node-rule-2 backward-chain-rule
    (type(load ?ld)(supply ?sup)(node ?no))
    (premise (or (and(unknown ?no class always-high)
                     (?no number ?gn(neq ?gn 0))
                     (?sup pos-node ?gn))
                 (and (unknown ?no class always-high)
                     (?no number ?gn (= (length (fvso ?no 'src-drn)) 1))
                     (?ld s-node ?gn(neq ?gn 0))
                     (?ld d-node ?ldp)
                     (?sup pos node ?ldp)) ))
   (conclusion(?no class ^(make-node-class ?no 'always high nil 1) )))

(rule always-high 3 backward-chain-rule
     (type(node ?no))
     (premise(and (unknown ?no class-1 always-high)
                  (?no class ?x (equal ?x 'always-high))))
     (conclusion(?no class-1 ^(make-node-class ?no 'always-high nil 1) )))

(rule always-clocked-rule backward-chain-rule
   (type(node ?no)(clock ?clk))
   (premise(and (unknown ?no class always-clocked)
```

```
                (?no number ?n)
                (?clk pos-node ?n)))
   (conclusion(?no class ^(make-node-class ?no 'always-clocked ?clk nil))))

(rule conditional-clocked-rule backward-chain-rule
   (type(node ?no)(drn-boot ?db))
   (premise(and(unknown ?no class conditional-clocked)
                (?no number ?n)
                (?db s-node ?n)
                (?db d-node ?dn)
                (?db boot-phase ?bp))).
   (conclusion(?no class
                   ^(make-node-class ?no 'conditional-clocked ?bp nil))))

(rule precharged-node-rule backward-chain-rule
    (type(node ?no)(precharger ?pre))
    (premise(and(unknown ?no class precharge)
                (?no number ?n)
                (?pre s-node ?n)
                (?pre pre-phase ?clk)))
   (conclusion(?no class ^(make-node-class ?no 'precharge ?clk nil))))

(rule dynamic-node-rule-1 backward-chain-rule
   (type (node ?no)(precharger ?pdr)(load ?ld)(driver ?dr)(clock ?clk)
          (drn-boot ?db)(active-two-port-element ?atp))
   (premise(and(unknown ?no class dynamic)
                (?no number ?n(neq ?n 0))
                (unknown ?ld s-node ?n)
                (unknown ?atp pos-node ?n)
                (unknown ?pdr s-node ?n)
                (unknown ?db s-node ?n)))
   (conclusion(?no class ^(make-node-class ?no 'dynamic nil nil))))

; dynamic node if it has one clocked load
(rule  dynamic-node-rule-2 backward-chain-rule
    (type(node ?no1 ?no)(load ?ld)(supply ?sup))
    (premise(and(unknown ?no class dynamic)
                (?no number ?n(equal 1 (length(fvso ?no 'load))))
                (?ld s-node ?n(neq ?n 0))
                (known ?ld class ?c (equal ?c 'ckc-load))
                (?ld d-node ?dnum)
                (?sup pos-node ?dnum)))
    (conclusion(?no class ^(make-node-class ?no 'dynamic nil nil))))

; clocked-low if it has pull-down driver which is clocked
(rule clocked low-rule backward-chain-rule
   (type (node ?no)(driver ?dr)(clock ?clk)(precharger ?pre)(drn-boot ?db)
        (load ?ld))
   (premise(and(unknown ?no class clocked-low)
                (?no number ?n)
                (?dr d node ?n)
                (?dr s-node 0)
                (?dr g-node ?gn)
                (?clk pos node ?gn)
                (unknown ?pre s-node ?n)
                (unknown ?db s-node ?n)))
    (conclusion(?no class
  ^(make-node-class ?no 'clocked-low (implode `(n - ,@(explode ?clk))) nil))))

;(rule vice-versa-1  backward-chain-rule
;   (type(node ?no))
;   (premise(?no class ?x))
;   (conclusion(?no class-1 ?x)))
```

```
;(rule vice-versa-2  backward-chain-rule
;   (type(node ?no))
;   (premise(and(?no class-1 ?x)))
;   (conclusion(?no class ?x)))

; aspect determination rules
(rule determine-aspect-rule-1 backward-chain-rule
    (type(node ?no)(supply ?sup))
    (premise(and(?no number ?n)
                 (?sup pos-node ?n)))
    (conclusion(?no aspect ?sup)))

(rule determine-aspect-rule-2 backward-chain-rule
   (type(node ?no ?gno)(driver ?dr)(clock ?clock))
   (premise(and(?no number ?n(neq ?n 0))
                (?dr s-node ?sn)
                (?dr d-node ?dn(or (equal ?n ?dn)(equal ?n ?sn)))
                (?dr g-node ?gn)
                (?clk pos-node ?gn)))
   (conclusion(?no aspect ?clk)))

(rule determine-aspect rule-3 backward-chain-rule
     (type (node ?no)(clock ?clk))
    (premise (and(?no number ?n)
                  (?clk pos-node ?n)))
    (conclusion(?no aspect ?clk)))
```

;*******************FILE: CLKING-FUNCS.SL******

det-clk-input

```
(defun det-clk-input (struct class substruct substruct-clkin)
   (let ( (struct-clkin (fvo struct 'clk-input))
         (struct-clk-class (fvo struct 'clk-class))
         (substruct-clk-class (fvo substruct 'clk-class)) )
      (cond ((or (equal struct-clkin 'error)(equal substruct-clkin 'error))
               'error)
            ((null struct-clkin) substruct-clkin)
            ((null substruct-clkin) struct-clkin)
            ((neq struct-clkin substruct-clkin) 'error)
            (t struct-clkin))))
```

det-clk-class

```
(defun det-clk-class (struct class substruct substruct-clkin)
   (let ( (struct-clkin (fvo struct 'clk-input))
         (struct-clk-class (fvo struct 'clk-class))
         (substruct-clk-class (fvo substruct 'clk-class)) )
      (cond ((or (equal struct-clk-class 'error)
                (equal substruct-clk-class 'error)) 'error)
            ((equal 1 (cond ((akop struct 'super-struct)
                              (length(fvso struct 'substruct)))
                        (t(length(fvso struct 'trans)))) )
              substruct-clk-class)
            ((equal class 'parallel)
                 (cond ((or(equal struct-clk-class 'hard)
                            (equal substruct-clk-class 'hard)) 'hard)
                       ((or(equal struct-clk-class 'conditional)
                            (equal substruct-clk-class 'conditional))
                                                         'conditional)
                       (t nil)))
            ((equal class 'series)
              (cond ((equal struct clk class substruct clk class) ; hard-hard
                                 struct-clk-class)    ;nil-nil cond-cond
                    ((or (and(equal struct clk class 'hard)
                             (equal substruct-clk-class nil))
                         (and(equal struct-clk-class  nil)
                             (equal substruct-clk-class 'hard)))
                         'conditional)
                    ((or (equal struct-clk-class 'conditional)   ; cond-any
                         (equal substruct-clk class 'conditional))
                         'conditional))) ) ))
```

define-clking-class

```
(defun define-clking-class (driver clk class)
    (fput driver 'clk-class '$value class)
    clk)
```

make-node-class

```
(defun make node class (node class aspect switch)
     (cond (switch (freplace node 'class '$value class))
           (t (freplace node 'class 1 '$value class)))
     (cond(aspect (freplace node 'aspect '$value aspect))
          (t(solve '(,node aspect ?x))))
      class)

;(defun determine-aspect (node class)
;  (cond((equal class 'always-high)(caaddr(solve
;              '(and(,node number ?x)(?supply pos-node ?x)))))))
;
; documentation of combinations
; class of structure:  series, parallel
```

```
; clk-input to transistor or structure: ck1 ck2
; clk-class of transistor or structure:
;    hard -> source and drain or node1 node2 connected during clock
;    conditional --> connection is possible but conditional on other inputs
;    error  --> violation of clocking rules
```

```
;********************FILE: RC-RULES.SL   ******

(rule long-rc-flag backward-chain-rule
   (type(load ?ld)(node ?sn) (supply ?sup))
   (premise(test(and(?ld d-node ?dn)
                    (?sup pos-node ?dn)
                    (?ld s-node ?lsn)
                    (?sn number ?lsn(>(length (fvso ?sn 'gate)) 0))
                    (?sn total-cap ?y))
            (long-rc-condition ?ld ?sn ?sup)))
   (conclusion (funny-node long-rc-flag ?sn)))

(rule node-total-cap backward-chain-rule
   (type(node ?nde))
   (premise(?nde number ?x))
   (conclusion(?nde total-cap ^(calc-capacitance ?nde))))

;********************FILE: RC-FUNCS.SL******
```

long-rc-condition

```
(defun long-rc-condition (load node supply)
 (let ( (res (/ (* (fvo supply 'e-value) (fvo load 'l-div-w))
                (fvo '*g-con 'std-ld-current)))
      (cap (fvo node 'total-cap)) )   ; res=kohms , cap=pf, tau=ns
;   (patom "long-rc  ")(patom res)(patom "  ")(patom cap)(terpri)
   (>(* res cap)(fvo '*g-con 'noise-tau))))
```

calc-capacitance

```
(defun calc-capacitance (node-name)
      (let((n-class nil)(n-aspect nil)
           (cap-list (fvso node-name 'cap))(cap nil)
           (tran-list (fvso node-name 'gate))(tran nil)
           (src-drn-list (fvso node-name 'src-drn))(src drn nil)
           (gate-cap 0)(stat-cap 0)(clka-cap 0)(clkb-cap 0)
           (clkc-cap 0)(clkd-ca 0)(other-cap 0)(src-drn-cap 0))

       (while(setq cap (car cap-list))      ;add up static capacitances
          (setq cap-list (cdr cap-list))
          (setq other-node (make-node-name
                           (find-other-port cap (fvo node-name 'number))))
          (solve '(,other-node class ?x))  ; determines node class and aspect
          (cond ((equal other-node 0)
                   (setq stat-cap (+ stat-cap (fvo cap 'e-value))))
                ((member (fvo other-node 'aspect) (fvso 'elements1 'supply))
                   (setq stat-cap (+ stat-cap (fvo cap 'e-value))))
                ((equal (fvo other-node 'aspect) *clka)
                   (setq clka-cap (+ clka-cap (fvo cap 'e-value))))
                ((equal (fvo other-node 'aspect) *clkb)
                   (setq clkb-cap (+ clkb-cap (fvo cap 'e-value))))
                (t(setq other-cap (+ other-cap (fvo cap 'e-value)))))) ;end while
;add up gate capacitance
      (while (setq tran (car tran list))
         (setq tran list (cdr tran list))
         (setq gate cap (+ gate-cap (* (fvo tran 'width)(fvo tran 'length)
                                       (fvo '*g-con 'gox cap))))) ;end while
;add up src-drn capacitance
      (while (setq src-drn (car src-drn-list))
          (setq src-drn-list (cdr src-drn-list))
          (setq src-drn-cap (+ src-drn-cap (* (fvo src-drn 'width)
                                              (fvo '*g-con 'gox-overlap-cap)))))
; put them in the  node frame
     (freplace node-name 'static-cap '$value stat-cap)
     (freplace node-name 'gate-cap '$value gate-cap)
     (freplace node-name 'clka-cap '$value clka-cap)
```

```
    (freplace node-name 'clkb-cap '$value clkb-cap)
    (freplace node-name 'other-cap '$value other-cap)
    (freplace node-name 'src-drn-cap '$value src-drn-cap)
    (+ stat-cap gate-cap clka-cap clkb-cap other-cap src-drn-cap)))
```

find-other-port

```
(defun find-other-port (cap node-number)
  (cond((equal (fvo cap 'pos-node) node-number) (fvo cap 'neg-node))
       (t(fvo cap 'pos-node))))
```

```
;*******************FILE: SUPBUF-RULES.SL******

; find inverting super-buffer
(rule find-super-buffer-1 backward-chain-rule
   (type(static-gate ?g1 ?g2)(load ?lg1 ?lg2)(elements ?elements))
   (premise(test(and(?g1 pull-up ?lg1)
                     (?g2 pull-up ?lg2 (neq ?g1 ?g2))
                     (?lg1 class ?x (equal ?x 'src-load))
                     (?lg2 class ?y (equal ?y 'oth-load))
                     (?g1 out-node ?g1on)
                     (?lg2 g-node ?g1on)
                     (?elements dummy 1))
              (super-buffer-condition ?g1 ?g2)))
    (conclusion (?elements super-buffer ^(make-super-buffer ?g1 ?g2 'inv))))

; find non-inverting super-buffer
(rule find-super-buffer-2 backward-chain-rule
   (type(static-gate ?g1 ?g2)(driver ?dr1)(load ?lg1 ?lg2)(elements ?elements))
    (premise(and(?elements dummy 1)
                 (?g1 pull-up ?lg1)
                 (?g2 pull-up ?lg2 (neq ?g1 ?g2))
                 (?lg1 class ?x (equal ?x 'src-load))
                 (?lg2 class ?y (equal ?y 'oth-load))
                 (?g1 out-node ?on1)
                 (?g1 pull-down ?dr1)
                 (?dr1 g-node ?in1)
                 (?g2 in-node ?on1)
                 (?lg2 g-node ?in1)))
   (conclusion(?elements super-buffer ^(make-super-buffer ?g1 ?g2 'non-inv))))

; flag super-buffer that may be wasting power
(rule super-buffer-flag-1 forward-chain-rule
    (type(super-buffer ?sb))
    (premise(and(?sb class ?x (equal ?x 'inv))
                 (?sb trigger t (sb-power-waste condition ?sb))))
    (conclusion (super-buffer-errors power-wa  e flag ?sb)))

;  flag super buffer without aggerssive predriver beta-ratio
(rule super-buffer-flag-2 forward-chain-rule
  (type(super-buffer ?sb)(gate ?pd))
  (premise (and(?sb class ?x (equal ?x 'inv))
                (?sb predriver ?pd)
                (?sb trigger t (>(fvo ?pd 'beta-ratio)
                                 (fvo '*g-con 'mx-sup-buf-agg-br)) ) ))
  (conclusion(super-buffer-errors aggressive-br-flag ?sb)))

;; if predriver has aggressive beta-ratio then remove it from br errors
(rule super-buffer-flag-3 forward-chain-rule
   (type(super-buffer ?sb)(gate ?pdr))
   (premise (and(?sb class ?x (equal ?x 'inv))
                 (?sb predriver ?pdr)
                 (?sb trigger t ( >(fvo ?pdr 'beta-ratio)
                                   (fvo '*g-con 'mn sup-buf-agg-br)))))
   (conclusion (eval (fremove
                      (cond((akop ?pdr 'inverter) 'inv-errors)
                           (t 'gate-errors)) 'beta ratio '$value ?pdr))))

; error if non-inverting super-buffer input isn't connected  to either
;   a load, drain-bootstrapper, or clock
(rule super-buffer-flag-4 forward-chain-rule
   (type (super-buffer ?sup)(driver ?dr)(clock ?clk)
        (static-gate ?ga)(drn-boot ?drnb)(load ?ld))
   (premise(and(?sup class ?x (equal ?x 'non-inv))
                (?sup trigger t)
```

```
            (?sup predriver ?ga)
            (?ga pull-down ?dr)
            (?dr g-node ?in)
            (unknown ?clk pos-node ?in)
            (unknown ?drnb s-node ?in)
            (unknown ?ld s-node ?in)))
    (conclusion (super-buffer-errors poor-input-drive ?sup)))
```

;********************FILE: SUPBUF-FUNCS.SL******

sb-power-waste-condition

```
(defun sb-power-waste-condition (sb)
   (let((pre-load (fvo (fvo sb 'predriver) 'pull-up))
        (drv-load (fvo (fvo sb 'driver) 'pull-up))
        (ratio nil))
(patom "predriver ")(patom pre-load)(patom "  driver ")(patom drv-load)(terpri)
     (setq ratio (* (fvo drv-load 'l-div-w)(/ 1 (fvo pre-load 'l-div-w))))
     (or (> ratio (fvo '*g-con 'mx-sup-buf-pwr-ratio))
         (< ratio (fvo '*g-con 'mn-sup-buf-pwr-ratio)))))
```

super-buffer-condition

```
(defun super-buffer-condition (predriver driver)
   (let ((flag t)(in-node1 (fvso predriver 'in-node))
              (in-node2 (fvso driver 'in-node))
              (node nil))
     (cond((neq (length in-node1)(length in-node2))(setq flag nil)))
     (while (and flag (setq node (car in-node1)))
         (setq in-node1 (cdr in-node1))
         (cond((null(member node in-node2))(setq flag nil))))
    flag))
```

make-super-buffer

```
(defun make-super-buffer (predriver driver class)
  (let((sup-buf (finstantiate 'super-buffer)))
    (patom sup-buf)(terpri)
    (fput sup-buf 'class '$value class)
    (fput sup-buf 'predriver '$value predriver)
    (fput sup-buf 'driver '$value driver)
    (fput sup-buf 'out-node '$value (fvo driver 'out-node))
    (fput-values sup-buf 'in-node  (fvso predriver 'in-node))
    (fput sup-buf 'driver br '$value (fvo driver 'beta-ratio))
    (fput sup-buf 'predriver br '$value (fvo predriver 'beta ratio))
    (fput sup-buf 'supply '$value (fvo driver 'supply))
    (fput sup-buf 'supply-node '$value (fvo (fvo driver 'pull u[illegible]d-node))
    (fput sup-buf 'trigger '$value t)
    sup-buf))
```

```
;********************FILE: D-BOOT-RULES.SL******

;;;;;;;;;;;;;;;;;; find drn-boot drivers ;;;;;;;;;;;;;;;;;;

; find a drn-boot driver
(rule find-drn-boot-1 backward-chain-rule
   (type(driver ?dr)(node ?gn)(elements ?elements)(clock ?clk))
   (premise(and(?elements dummy 1)
               (known ?dr class ?c (or(equal ?c 'unclassified)
                                      (equal ?c 'xi)(equal ?c 'xfer)))
               (?dr d-node ?d-num)
               (?clk pos-node ?d-num)
               (?dr g-node ?g-num)
               (?gn number ?g-num)
               (?gn class ?y (equal ?y 'dynamic))))
   (conclusion(?elements drn-boot ^(make-drn-boot ?dr ?clk))))

; find a drn-boot driver
(rule find-drn-boot-2 backward-chain-rule
   (type(driver ?dr)(node ?dn ?gn)(elements ?elements))
   (premise(and(?elements dummy 1)
               (known ?dr class ?c (or(and(neq ?c 'drn-boot)
                                          (equal ?c 'unclassified))
                                      (and(neq ?c 'drn-boot)
                                          (equal ?c 'xi))))
               (?dr d-node ?d-num)
               (?dn number ?d-num)
               (?dn class ?x (equal ?x 'conditional-clocked))
               (?dr g-node ?g-num)
               (?gn number ?g-num)
               (?gn class ?y (equal ?y 'dynamic))))
   (conclusion(?elements db1 ^(make-drn-boot ?dr nil))))

(rule other-clk-hold-down-1 backward-chain-rule
   (type(drn-boot ?db)(driver ?dr)(clock ?clk)(node ?no))
   (premise(and(?db s-node ?sn)
               (?dr d-node ?sn)
               (?dr s-node ?x (equal ?x 0))
               (?dr g-node ?gn)
               (?clk pos-node ?gn(neq ?clk (fvo ?dr 'boot-phase)))))
   (conclusion(?db other-clk-hold-down ?dr)))


;;;;;;;;;;;;;;;;;;;;;; find drain-bootstrap cells;;;;;;;;;

(rule find-drain-bootstrap-1 backward-chain-rule
   (type(drn-boot ?db)(inverter ?inv)(driver ?cg)(elements ?elements)
        (node ?no))
   (premise(and(?db status free)
               (?inv status free)
               (?db g-node ?gn)
               (?inv out-node ?gn)
               (?inv in node ?in)
               (?inv xfer gate ?cg)
               (?cg g-node ?cggn)
               (?no number ?cggn)
               (?no aspect ?clk)
               (?elements dummy 1)))
   (conclusion(?elements drain-bootstrap
                  ^(make-drain-bootstrap ?db ?inv ?clk ?in))))

(rule find-drain-bootstrap-3 backward-chain-rule
   (type(drn-boot ?db)(driver ?xf)(elements ?elements)(node ?no))
   (premise(and(?db status free)
```

```
                (known ?db o-ins xi)
                (?db xfer-gate ?xf)
                (?xf g-node ?g)
                (?no number ?g)
                (?no aspect ?clk)
                (?elements dummy 1)))
   (conclusion(?elements drain-bootstrap
           ^(make-drain-bootstrap ?db ?xf ?clk nil))))

(rule find-mos-cap-rule backward-chain-rule
   (type(drain-bootstrap ?d)(mos-cap ?m)(drn-boot ?db))
   (premise(test(and(?d drn-boot ?db)
                   (?db g-node ?gn)
                   (?db d-node ?sn)
                   (?m status free)
                   (?m g-node ?mgn)
                   (?m s-node ?msn))
           (or(and(equal ?gn ?mgn)(equal ?sn ?msn))
              (and(equal ?gn ?msn)(equal ?sn ?mgn)))))
   (conclusion(?d mos-cap ?m)))

;;;;;;;;;;;;;;;;;;;;;;;;;;; drn-boot checks ;;;;;;;;;;;;;;;;;;;;;

; check for other phase hold-down
(rule drn-boot-check-1 backward-chain-rule
    (type (drn-boot ?db))
    (premise(and(elements1 drn-boot ?db)
                (unknown ?db other-clk-hold-down ?x)))
    (conclusion (drn-boot-errors phase-hold-down ?db)))

; check for mos-cap in correctly
(rule check-db-mos-cap backward-chain-rule
   (type(drain-bootstrap ?d)(mos-cap ?m))
   (premise(and(?d boot-node ?bn)
               (known ?d mos cap ?m)
               (?m g-node ?g (neq ?g ?bn))))
   (conclusion(drn-boot-errors mos cap-backwards ?d)))

; checkin for clocking errors in drain-bootstrap
(rule check-db-clockin backward-chain-rule
   (type(drain-bootstrap ?d))
   (premise(and(?d boot-phase ?bp)
               (?d pre-phase ?bp)))
   (conclusion (drn boot-errors clocking-error ?d)))

; check for proper hold-down length
(rule check-db-hd-length backward-chain-rule
  (type(drain bootstrap ?d)(driver ?dr)(inverter ?pd))
  (premise(and(?d predriver ?pd (null (akop ?pd 'transistor)))
              (?pd pull-down ?dr)
              (?dr length ?l (< ?l (fvo "g con 'db ahd 1)))))
  (conclusion(drn-boot-errors longer driver needed ?dr)))

; check for bootnode active low
(rule check-db boot node backward chain rule
  (type(drain bootstrap ?d))
  (premise(?d predriver ?pd (akop ?pd 'transistor)))
  (conclusion(drn-boot-errors boot-node-not-active-low ?d)))
```

;********************FILE: D-BOOT-FUNCS.SL******

make-drn-boot

```
(defun make-drn-boot (dr boot-phase)     ; trigger put is asserted later on
 (let ((d-node (make-node-name (fvo dr 'd-node)))(xid nil)
      (s-node (make-node-name (fvo dr 's-node))))
  (cond((akop dr 'xi-driver)
        (fput dr 'o-ins '$value 'xi)
        (freplace dr 'width '$value (/ (fvo dr 'width)
                                       (fvo '*g-con 'xi-dr-wrf)))
        (fremove 'elements1 'xi-driver '$value dr)
        (fremove 'coupling-errors 'xi-driver-coupling '$value dr)
        (fremove 'coupling-errors 'xi-driver-coupling-1 '$value dr)))
  (freplace dr 'ako '$value 'drn-boot)
  (freplace dr 'class '$value 'drn-boot)
  (freplace dr 'boot-phase '$value
                         (cond ((null boot-phase)(fvo d-node 'aspect))
                               (t boot-phase)))
  (make-node-class s-node 'conditional-clocked (fvo dr 'boot-phase) nil)
  (patom "drn-boot  ")(patom dr)(terpri)
  dr) )
```

solve-all-drn-boots

```
(defun solve-all-drn-boots ()
  (let((drn-boot-list nil)(db nil)(temp nil)(temp-list nil))
    (solve-all '(?elements drn-boot ?x))
    (setq temp-list (fvso 'elements1 'drn-boot))
    (while (setq temp (car temp-list))
          (setq temp-list (cdr temp-list))
          (fput 'elements1 'db1 '$value temp))
    (while (fvso 'elements1 'db1)
         (setq drn-boot-list (append (fvso 'elements1 'db1) drn-boot-list))
         (fremove 'elements1 'db1)
         (solve-all '(?elements db1 ?x)))
    (while (setq db (car drn-boot-list))
          (setq drn-boot-list (cdr drn-boot-list))
          (freplace db 'trigger '$value t)
          (fput 'elements1 'drn-boot '$value db))
    (fix-xi-driver-errors)
    (solve-all '(?drn-boot other-clk-hold-down ?x))))
```

make-other-clk-h-d

```
(defun make-other-clk-h-d (driver)
    driver)
```

make-drain-bootstrap

```
(defun make-drain-bootstrap (drn-boot predriver inclk in-node)
   (let((db (finstantiate 'drain-bootstrap)))
     (patom db)(terpri)
     (fput db 'predriver '$value predriver)
     (fput db 'drn-boot '$value drn-boot)
     (fput db 'in-node '$value
       (cond((null in-node)
              (cond((equal (fvo predriver 'd-node)(fvo drn-boot 'g-node))
                     (fvo predriver 's-node))
                    (t (fvo predriver 'd-node)) ))
            (t in-node)) )
     (fput db 'out-node '$value (fvo drn-boot 's-node))
     (fput db 'boot-node '$value (fvo drn-boot 'g-node))
     (fput db 'pre-phase '$value inclk)
     (fput db 'boot-phase '$value (fvo drn-boot 'boot-phase))
     (cond((fvso drn-boot 'other-clk-hold-down)
           (fput-values db 'other-clk-hold-down
```

```
            (fvso drn-boot 'other-clk-hold-down))))
    db))
```

solve-all-drain-bootstraps

```
(defun solve-all-drain-bootstraps ()
  (find-drain-bootstrap-cells)
  (solve-all '(?drain-bootstrap mos-cap ?x)))
```

find-drain-bootstrap-cells

```
(defun find-drain-bootstrap-cells ()
  (let((db-list nil))
    (while (solve '(?elements drain-bootstrap ?x))
        (cond((null db-list)(setq db-list (fvso 'elements1 'drain-bootstrap)))
            (t(setq db-list (cons (fvo 'elements1 'drain-bootstrap) db-list))))
        (fremove 'elements1 'drain-bootstrap '$value))
    (cond (db-list (fput-values 'elements1 'drain-bootstrap db-list)))))
```

fix-xi-driver-errors

```
(defun fix-xi-driver-errors ()
  (let((db-list (fvso 'elements1 'drn-boot))(db nil)(xid nil))
  (while(setq db (car db-list))
      (setq db-list (cdr db-list))
      (cond((setq xid (cadr(solve '(known ?xi-driver xfer-gate ,db))))
            (fremove 'coupling-errors 'xi-driver-coupling '$value xid)
            (fremove 'coupling-errors 'xi-driver-coupling-1 '$value xid))))))
```

;********************FILE: REG-RULES.SL*****

```
(rule find-reg-core-1 backward-chain-rule
   (type(xi-inverter ?xi1 ?xi2)(reg-driver ?dr)(elements ?elements)
        (clock ?clk1 ?clk2 ?clk3))
   (premise(and(?elements dummy 1)
                (?xi1 in-node ?in1)
                (?xi1 out-node ?out1)
                (?xi1 status ?st1 (equal ?st1 'free))
                (?xi2 in-node ?in2(equal ?in2 ?out1))
                (?xi2 out-node ?out2)
                (?xi2 status ?st2 (equal ?st2 'free))
                (?dr status free)
                (?dr d-node ?dn(or(equal ?dn ?in1)
                                  (equal ?dn ?out2)))
                (?dr s-node ?sn(or(equal ?sn ?in1)
                                  (equal ?sn ?out2)))
                (?xi1 xfer-gate ?xf1)
                (?xf1 g-node ?xf1gn)
                (?clk1 pos-node ?xf1gn)
                (?xi2 xfer-gate ?xf2)
                (?xf2 g-node ?xf2gn)
                (?clk2 pos-node ?xf2gn)
                (?dr g-node ?drgn)
                (?clk3 pos-node ?drgn)))
    (conclusion(?elements reg-core
        ^(make-reg-core ?xi1 ?clk1 ?xi2 ?clk2 ?dr ?clk3 ?in1 ?out2 nil))))

(rule find-reg-cell-1 backward-chain-rule
   (type(reg-core ?rc)(reg driver ?dr1 ?dr2)(xi-inverter ?rcin)
          (elements ?elements)(node ?no1 ?no2))
  (premise(test(and(?elements dummy 1)
                (?rc in-node ?in)
                (?rc out-node ?out)
                (?rc status free)
                (?dr1 d node ?dn1)
                (?dr1 s-node ?sn1 (or(equal ?sn1 ?in)
                                     (equal ?dn1 ?in)))
                (?dr1 status free)
                (?dr2 d-node ?dn2(neq ?dr2 ?dr1))
                (?dr2 s-node ?sn2 (or(equal ?sn2 ?out)
                                     (equal ?dn2 ?out)))
                (?dr2 status free)
                (?rc in stage ?rcin)
                (?rcin out node ?rcinon))
       (and(neq ?rcinon ?dn1)(neq ?rcinon ?dn2)
           (neq ?rcinon ?sn1)(neq ?rcinon ?sn2))))

  (conclusion (?elements reg cell ^(make-reg-cell ?rc ?dr1  ?dr2  nil))))

(rule find-reg inclk backward chain rule
   (type(reg cell ?rc)(reg driver ?dr)(node ?no))
   (premise(and(?rc in stage ?dr)
               (?dr g node ?gn)
               (?no number ?gn)
               (?no class ?cl)))
   (conclusion(?rc in-clk ^(fvo ?no 'aspect))))

(rule find-reg-outclk backward-chain-rule
   (type(reg-cell ?rc)(reg-driver ?dr)(node ?no))
   (premise(and(?rc out-stage ?dr)
               (?dr g-node ?gn)
               (?no number ?gn)
               (?no class ?cl)))
```

```
(conclusion(?rc out-clk ^(fvo ?no 'aspect))))

; error if any other fets connected from internal core to outnode of reg
(rule reg-internal-con-check-1 backward-chain-rule
  (type(errors)(reg-cell ?rc)(struct ?st)(transistor ?tr)(reg-core ?r)
       (node ?no1)(struct ?st)(xi-inverter ?os))
  (premise(or(and(?rc reg-core ?r)
                 (?r in-node ?in )
                 (or (?tr s-node ?in)(?tr d-node ?in))
                 (?tr status free))
             (and(?rc reg-core ?r)
                 (?r out-stage ?os)
                 (?os in-node ?osin)
                 (?os out-node ?osout)
                 (or(and(?tr s-node ?osin)
                        (?tr d-node ?osout))
                    (and(?tr s-node ?osout)
                        (?tr d-node ?osin))
                    (and(?st node-1 ?osin)
                        (?st node-2 ?osout))
                    (and(?st node-1 ?osout)
                        (?st node-1 ?osin))))))
  (conclusion (register-errors internal-connection ?rc)))

; reg clocking must ber correct
(rule reg-clking-check backward-chain-rule
   (type(errors)(reg-cell ?rc)(reg-core ?r))
   (premise(test(and(?rc reg-core ?r)
                    (known ?rc in-clk ?inclk)
                    (known ?rc out-clk ?outclk)
                    (known ?r in-clk ?rinclk)
                    (known ?r out-clk ?routclk)
                    (known ?r rec-clk ?rrecclk))
            (or(neq ?inclk ?rinclk)(equal ?inclk ?rrecclk)
               (neq ?rrecclk ?routclk)(neq ?inclk ?outclk))))
   (conclusion (register-errors clocking error ?rc)))

;flag is register has only clocked gate for recirculate path
(rule reg-cirtical-node-flag backward-chain-rule
   (type (errors)(reg-cell ?rc)(reg-core ?r))
   (premise(and (?rc reg-core ?r)
                (?r rec-stage ?rs (akop ?rs 'driver))))
   (conclusion(register-errors critical node flag ?rc)))

; error if register input is clocked on the same phase as the precharger .
(rule reg-clking-check-2 backward-chain rule
   (type(precharger ?pr)(reg-cell ?rc))
   (premise(and(?pr s-node ?sn)
               (?rc in-node ?sn)
               (?pr pre phase ?pp)
               (?rc in-clk ?pp)))
   (conclusion(register errors clocking-error ?rc)))
```

;*******************FILE: REG-FUNCS.SL******

solve-all-reg-cells

```
(defun solve-all-reg-cells ()
      (solve-all '(?elements reg-core ?x))
      (solve-all '(?elements reg-cell ?x))
      (solve-all '(?reg-cell in-clk ?x))
      (solve-all '(?reg-cell out-clk ?x))
      (solve-error-frame 'register-errors))
```

make-reg-core

```
(defun make-reg-core (in-s inclk out-s outclk rec-p recclk in-n out-n rec-n)
  (let( (core (finstantiate 'reg-core)) )
    (patom core)(terpri)
    (fput core 'in-stage '$value in-s)
    (fput core 'out-stage '$value out-s)
    (fput core 'rec-stage '$value rec-p)
    (fput core 'in-clk '$value inclk)
    (fput core 'out-clk '$value outclk)
    (fput core 'rec-clk '$value recclk)
    (fput core 'in-node '$value in-n)
    (fput core 'out-node '$value out-n)
    (cond(rec-n (fput core rec-node '$value rec-n)))
    core))
```

make-reg-cell

```
(defun make-reg-cell(core in-stage  out-stage  rec-inv)
 (let((reg-cell (finstantiate 'reg-cell)))
   (patom reg-cell)(terpri)
   (fput reg-cell 'reg-core '$value core)
   (fput reg-cell 'in-stage '$value in-stage)
   (fput reg-cell 'out-stage '$value out-stage)
   (fput reg-cell 'in-node '$value (find-reg-cell-node core in stage 'in))
   (fput reg-cell 'out-node '$value (find-reg-cell-node core out-stage 'out))
   (cond(rec-inv (fput reg-cell 'rec-inv '$value rec-inv)))
   reg-cell))
```

find-reg-cell-node

```
(defun find-reg-cell-node(core stage key)
   (let((core-in (fvo core 'in-node))(core-out (fvo core 'out-node))
        (node1 (cond((akop stage 'driver)(fvo stage 'd-node))
                     (t(fvo stage 'node-1))))
        (node2 (cond((akop stage 'driver)(fvo stage 's-node))
                     (t(fvo stage 'node-2)))) )
        (cond ((equal key 'in)
                  (cond((equal node1 (fvo core 'in-node)) node2)
                       (t node1)))
              ((equal key 'out)
                  (cond ((equal node1 (fvo core 'out-node)) node2)
                        (t node1)) )) ))
```

```
;********************FILE: STRUCT-RULES.SL******

(rule find-xc-xor-struct backward-chain-rule
   (type(reg-driver ?dr1 ?dr2)(elements ?elements))
   (premise(test(and(?dr1 s-node ?s1)
                    (?dr1 g-node ?g1)
                    (?dr1 d-node ?d1)
                    (?dr1 status free)
                    (?dr2 s-node ?s2)
                    (?dr2 g-node ?g2)
                    (?dr2 d-node ?d2)
                    (?dr2 status free)
                    (?elements dummy 1))
         (or(and(equal ?d1 ?d2)(equal ?s1 ?g2)(equal ?s2 ?g1)(neq ?s1 ?s2))
            (and(equal ?d1 ?s2)(equal ?s1 ?g2)(equal ?d2 ?g1)(neq ?s1 ?d2))
            (and(equal ?s1 ?d2)(equal ?d1 ?g2)(equal ?s2 ?g1)(neq ?d1 ?s2))
            (and(equal ?s1 ?s2)(equal ?d1 ?g2)(equal ?d2 ?g1)(neq ?d1 ?d2)))))
  (conclusion(?elements xc-xor-struct ^(make-xc-xor-struct ?dr1 ?dr2))))

;********************FILE: STRUCT-FUNCS.SL******
```

find-other-structs

```
(defun find-other-structs()
   (solve-all '(?elements xc-xor-struct ?x))
   (combine-xc-xor-struct))
```

make-xc-xor-struct

```
(defun make-xc-xor-struct (dr1 dr2)
  (let((xor (finstantiate 'xc-xor-struct))
       (dn1 (fvo dr1 'd-node))(sn1 (fvo dr1 's-node))
       (dn2 (fvo dr2 'd-node))(sn2 (fvo dr2 's-node)))
   (patom xor)(terpri)
   (fput xor 'class '$value 'xc-xor)
   (fput xor 'l-div-w '$value (max (fvo dr1 'l-div-w)(fvo dr2 'l-div-w)))
   (fput xor 'trans '$value dr1)
   (fput xor 'trans '$value dr2)
   (fput xor 'string-l '$value 1)
   (fput xor 'out-node '$value
            (cond((equal dn1 dn2) dn1)
                 ((equal dn1 sn2) dn1)
                 ((equal sn1 sn2) sn1)
                 ((equal sn1 dn2) sn1)
                 (t (patom "*****make-xc-xor error******")(terpri))))
   (fput xor 'in node '$value
          (cond((equal dn1 (fvo xor 'out-node)) sn1)
               (t dn1)))
   (fput xor 'in-node '$value
          (cond((equal dn2 (fvo xor 'out-node)) sn2)
               (t dn2)))
   xor))
```

combine-xc-xor-struct

```
(defun combine xc xor struct ()
   (let((xc list (fvso 'elements1 'xc xor struct))(new-list nil)(xor nil)
        (xor out nil)(new-struct nil)(xor1 nil))
     (while (setq xor (car xc-list))
          (setq xc-list (cdr xc-list))
          (setq xor-out (fvo xor 'out-node))
          (cond ((setq xor1 (caadadr (solve
            '(test(and(?xc-xor-struct status free(neq ?xc-xor-struct ',xor))
                      (?xc-xor-struct out-node ,xor-out))
               (null (member ?xc-xor-struct new-list))))))
              (setq xc-list (delete xor1 xc-list))
              (setq new-struct (finstantiate 'xc-xor-struct))
```

```
            (patom new-struct)
            (cond((null new-list)(setq new-list (list new-struct)))
                 (t (setq new-list (cons new-struct new-list))))
            (fput new-struct 'substruct '$value xor)
            (fput new-struct 'class '$value 'xc-xor)
            (fput new-struct 'string-1 '$value 1)
            (fput-values new-struct 'in-node (fvso xor 'in-node))
            (fput new-struct 'l-div-w '$value (fvo xor 'l-div-w))
            (fput new-struct 'out-node '$value xor-out)
            (add-next-struct new-struct xor1)
          (while (setq xor1 (caadadr (solve
        '(test(and(?xc-xor-struct status free(neq ?xc-xor-struct ',xor))
                  (?xc-xor-struct out-node ,xor-out))
                     (null (member ?xc-xor-struct new-list))))))
             (add-next-struct new-struct xor1)) )) )
        (cond(new-list
                (fput-values 'elements1 'xc-xor-struct new-list))) ))
```

add-next-struct

```
(defun add-next-struct(new-struct xor1)
     (fput new-struct 'substruct '$value xor1)
     (fput-values new-struct 'in-node (fvso xor1 'in-node))
     (freplace new-struct 'l-div-w '$value (max (fvo new-struct 'l-div-w)
                                                (fvo xor1 'l-div-w))))
```

```
;********************FILE: CSHARE-RULESSL******

; flag capacitive-feedback into node with a feedback transistor
(rule charge-share-1 backward-chain-rule
   (type(node ?no1 ?no2 ?no3)(driver ?dr1 ?dr2 ?fb))
   (premise    (and(?fb fb-tran-flag t)
                 (?fb g-node ?fbgn)
                 (?fb d-node ?fbdn)
                 (?dr1 g-node ?fbdn)
                 (?no1 number ?fbgn)
                 (?no1 class ?c1)
                 (?no1 total-cap ?cap1)
                 (?dr2 d-node ?fbgn)
                 (?dr2 s-node ?n2(neq ?n2 0))
                 (?no2 number ?n2)
                 (?no2 total-cap ?cap2)
                 (?dr2 g-node ?gn2)
                 (?no3 number ?gn2)
                 (?no3 class ?x(or(equal ?x 'always-clocked)
                                  (equal ?x 'conditional-clocked)))))
    (conclusion(charge-share-errors feedback-glitch-flag
                 ^(check-feedback-glitch ?dr1 ?dr2 ?cap1 ?cap2 ?no1 ?no2))))

(rule charge-share-2 backward-chain-rule
   (type(node ?no1 ?no2 ?no3)(driver ?dr1 ?dr2 ?fb))
   (premise    (and(?fb fb-tran-flag t)
                 (?fb g-node ?fbgn)
                 (?fb d-node ?fbdn)
                 (?dr1 g-node ?fbdn)
                 (?no1 number ?fbgn)
                 (?no1 class ?c1)
                 (?no1 total-cap ?cap1)
                 (?dr2 s-node ?fbgn)
                 (?dr2 d-node ?n2(neq ?n2 0))
                 (?no2 number ?n2)
                 (?no2 total-cap ?cap2)
                 (?dr2 g-node ?gn2)
                 (?no3 number ?gn2)
                 (?no3 class ?x(or(equal ?x 'always-clocked)
                                  (equal ?x 'conditional-clocked)))))
    (conclusion(charge-share-errors feedback-glitch-flag
                 ^(check-feedback-glitch ?dr1 ?dr2 ?cap1 ?cap2 ?no1 ?no2))))

;********************FILE: CSHARE-FUNCSSL******
```

check-feedback-glitch

```
(defun check-feedback-glitch (dr1 dr2 c1 c2 n1 n2)
   (let((r1 (fvo dr1 'l-div-w))(r2 (fvo dr2 'l-div-w))(n-list (list n1 n2))
        (dr-ratio (fvo '*g-con 'dr-cshare-ratio))
        (cap-ratio (fvo '*g-con 'cap-cshare-ratio)))
    (cond((and(< c1 (* cap-ratio c2))(< r2 (* dr-ratio r1)))
          (fput 'charge-share-errors 'feedback-glitch-error '$value n-list)))
    n-list))
```

```
;********************FILE: RACE-COND.SL******

; precharge loss if two gates connected to each other are both clocked
;  low on the same phase and the output of the second drives a dynamic
;   gate input
(rule race-condition-1 backward-chain-rule
    (type (errors)(static-gate ?sg1 ?sg2)(dynamic-gate ?dg1)(node ?no1 ?no2))
    (premise(test(and (?sg1 out-node ?out1)
                      (?sg2 in-node ?out1 (neq ?sg1 ?sg2))
                      (?sg2 out-node ?out2)
                      (?dg1 in-node ?out2)
                      (?no1 number ?out1)
                      (?no2 number ?out2)
                      (?no1 class ?x)
                      (?no2 class ?y))
           (and(equal ?x 'clocked-low)(equal ?y 'clocked-low))))
     (conclusion (race-errors precharge-loss ?dg1)))

; dynamic xor has input timing sensitivities
(rule dynamic-xor-race-condition forward-chain-rule
   (type(dynamic-xc-xor ?xor))
   (premise(?xor trigger t))
   (conclusion (race-errors input-skew-flag ?xor)))

; flag a transfer gate driven by a bootstrapper if clockskew
;   sensitivity flag in *g-con is true
(rule clock-skew-rule-1 backward-chain-rule
   (type(driver ?dr)(drn-boot ?db)(node ?n1 ?n2))
   (premise (and(*g-con clk-skew-flag t)
                (?db check ?chk1(neq ?chk1 'skew-flag-1))
                (?dr check ?chk2(neq ?chk2 'skew-flag-1))
                (?db s-node ?sn)
                (?dr d-node ?sn)
                (?dr g-node ?dgn)
                (?dr s-node ?dsn)
                (?n2 number ?dgn)
                (?n2 class always-clocked)
                (?n2 aspect ?a (equal ?a (fvo ?db 'boot-phase)))
                (?n1 number ?dsn)
                (?n1 class dynamic)))
     (conclusion(clk-skew-errors clock-skew-flag-1 ^(put-clk-skew ?db ?dr 1))))

; same as above rule , just with fet s-d reversed
(rule clock-skew-rule-2 backward-chain-rule
   (type(driver ?dr)(drn-boot ?db)(node ?n1 ?n2))
   (premise (and(*g-con clk-skew-flag t)
                (?db check ?chk1(neq ?chk1 'skew-flag-1))
                (?dr check ?chk2(neq ?chk2 'skew-flag-1))
                (?db s-node ?sn)
                (?dr s-node ?sn)
                (?dr g-node ?dgn)
                (?dr d-node ?dsn)
                (?n2 number ?dgn)
                (?n2 class always-clocked)
                (?n2 aspect ?a (equal ?a (fvo ?db 'boot-phase)))
                (?n1 number ?dsn)
                (?n1 class dynamic)))
     (conclusion(clk-skew-errors clock-skew-flag-1 ^(put-clk-skew ?db ?dr 1))))

;; flag two drivers connected together with one gate conditionally
; clocked and one gate always clocked
(rule clock-skew-rule-3 backward-chain-rule
   (type(node ?dn1 ?sn1 ?gn1 ?gn2)(driver ?dr1 ?dr2))
   (premise (and(*g-con clk-skew-flag t)
```

```
                    (known ?gn1 class conditional-clocked)
                    (?gn1 number ?g1)
                    (?dr1 g-node ?g1)
                    (?dr1 check ?chk1 (neq ?chk1 'skew-flag-2))
                    (?dr2 check ?chk2 (neq ?chk2 'skew-flag-2))
                    (?dr1 s-node ?s1)
                    (?dr1 d-node ?d1)
                    (?dr2 s-node ?s2(neq ?dr2 ?dr1))
                    (?dr2 d-node ?d2(or(= ?d2 ?s1)(= ?d2 ?d1)
                                    (= ?s2 ?s1)(= ?s2 ?d1)))
                    (?sn1 number ?s1)
                    (?dn1 number ?d1)
                    (or(?sn1 class dynamic(or(= ?s1 ?s2)(= ?s1 ?d2)))
                      (?dn1 class dynamic(or(= ?d1 ?s2)(= ?d1 ?d2))))
                    (?dr2 g-node ?g2)
                    (?gn2 number ?g2)
                    (?gn2 class always-clocked)
                    (known ?gn1 aspect ?asp)
                    (known ?gn2 aspect ?asp)))
(conclusion(clk-skew-errors clock-skew-flag-2
                                ^(put-clk-skew ?dr1 ?dr2 2))))
```

;********************FILE: RACE-FUNCS.SL ****

solve-all-clk-skew-errors

```
(defun solve-all-clk-skew-errors ()
  (let((slot-list (delete 'ako (fslots 'clk-skew-errors)))(slot nil))
     (cond((fvo '*g-con 'clk-skew-flag)
           (while (setq slot (car slot-list))
              (setq slot-list (cdr slot-list))
              (patom "*suppress _")(patom *suppress-justifications*)(terpri)
              (my-solve-all '(clk-skew-errors ,slot ?x)))))))
```

put-clk-skew

```
(defun put-clk-skew (dr1 dr2 flag)
  (let ((skew-fg (implode '(s k e w - f l a g - ,@(explode flag))))
       (dr1-check (fvso dr1 'check))(dr2-check (fvso dr2 'check)))
   (fremove dr1 'check '$value)
   (fremove dr2 'check '$value)
   (fput-values dr1 'check (cons skew-fg (delete 'unchecked dr1-check)))
   (fput-values dr2 'check (cons skew-fg (delete 'unchecked dr2-check)))
   (list dr1 dr2)))
```

;*********************FILE: PAD-RULES.SL ******

```
(rule input-protection-check backward-chain-rule
  (type(pad ?pad)(driver ?dr))
  (premise(and(?pad node-num ?n)
              (unknowable ?pad prot-device ?dr)))
  (conclusion(input-pad-errors missing-protection-device ?pad)))

(rule input-undershoot-check backward-chain-rule
   (type(pad ?pad)(driver ?dr))
   (premise(test(and(?pad node-num ?n)
                    (?dr s-node ?sn)
                    (?dr d-node ?dn (or(equal ?dn ?n)(equal ?sn ?n))))
       (and(neq ?sn 0)(neq ?dn 0))))
   (conclusion(input-pad-errors undershoot-flag ?pad)))

(rule find-pad-protect-device backward-chain-rule
  (type(pad ?pad)(driver ?dr))
  (premise(and(?pad node-num ?n)
             (?dr d-node ?n)
             (?dr g-node 0)
             (?dr s-node 0)))
  (conclusion(?pad prot-device ?dr)))
```

```
;********************FILE: NET-LISTSSL ******

(setq *net-list-1 '((*net-list-1)
   (load m1 1 2 2 4 8)
   (driver m2 2 3 0 6 2)
   (driver m3 2 4 5 6 2)
   (driver m4 5 6 0 6 2)))

(setq *net-list-2 '((*net-list-2)
    (load m1 1 2 2 4 8)
    (driver m2 2 3 0 2 2)
    (driver m3 2 5 4 4 2)
    (load m4 1 6 6 4 8)
    (driver m5 6 4 0 2 2)
    (load m6 1 7 7 4 8)
    (driver m7 7 6 8 6 2)
    (driver m8 8 12 0 6 2)
    (driver m9 7 10 9 6 2)
    (driver m10 9 11 0 6 2)
    (load m11 1 12 12 4 8)
    (driver m12 12 13 16 6 2)
    (driver m13 12 14 16 6 2)
    (driver m14 12 15 16 6 2)
    (driver m15 16 17 18 6 2)
    (driver m16 18 19 0 6 2)
    (supply v1 1 0 5)
))

(setq *net-list-4 '((*net-list-4)
    (driver m17 20 23 25 6 2)
    (driver m18 25 23 21 6 2)
    (driver m19 20 23 24 6 2)
    (driver m20 24 23 21 6 2)
    (driver m21 20 23 21 6 2)
    (driver m22 21 23 22 6 2)
    (driver m23 22 23 0 6 2)
    (driver m24 20 23 26 6 2)
    (driver m25 26 23 21 6 2)
    (driver m26 21 23 27 6 2)
    (driver m27 27 23 0 6 2)
    (supply v1 20 0 5)
))

(setq *net-list-5 '((*net-list 5)
    (load m1 1 2 2 4 8)
    (driver m2 2 3 0 2 2)
    (supply v1 1 0 5)
    (driver m13 2 10 4 4 2)
    (load m3 1 5 5 4 8)
    (driver m4 5 4 0 2 2)
    (load m5 1 6 6 4 8)
    (driver m6 6 5 7 6 2)
    (driver m7 7 10 8 6 2)
    (driver m8 8 10 0 6 2)
    (load m9 1 9 9 4 8)
    (driver m10 9 6 0 6 2)
    (driver m11 9 10 0 6 2)
    (driver m12 9 10 0 6 2)
))

(setq *net-list-6 '((*net-list-6)
   (supply v1 1 0 5)
   (load m1 1 2 2 4 8)
   (load m2 1 4 4 4 8)
```

```
    (load m3 1 6 6 4 8)
    (load m4 1 8 8 4 8)
    (driver m5 2 3 0 6 2)
    (driver m6 4 2 0 6 2)
    (driver m7 6 5 0 6 2)
    (driver m8 8 7 0 6 2)
    (driver m9 4 9 5 4 2)
    (driver m10 6 9 7 4 2)))

(setq *net-list-7 '((*net-list-7)
     (supply v1 1 0 5)
     (clock ph1 2 0 5)
     (load m1 1 4 4 4 8)
     (load m2 1 1 5 4 8)
     (load m3 1 1 5 4 8)
     (load m4 1 2 7 4 8)
     (load m5 1 2 8 4 8)
     (load m6 8 2 9 4 8)
     (driver m7 4 3 0 6 2)
     (driver m8 5 4 0 6 2)
     (driver m9 6 5 0 6 2)
     (driver m10 7 6 0 6 2)
     (driver m11 8 7 0 6 2)))

(setq *net-list-8 '((*net-list-8)
      (load m1 1 2 2 4 6)
      (supply v1 1 0 5)
      (driver m2 2 5 3 6 2)
      (driver m3 2 5 4 6 2)
      (driver m4 3 5 4 6 2)
      (driver m5 3 5 0 6 2)
      (driver m6 4 5 0 6 2)))

(setq *net-list-9 '((*net-list-9)
     (supply v1 2 0 5)
     (driver m1 2 3 0 6 2)
     (driver m2 2 3 0 6 2)
     (driver m3 2 3 0 6 2)
     (driver m4 2 3 0 6 2)
     (driver m5 2 3 0 6 2)
     (driver m6 0 3 2 6 2)
     (driver m7 2 3 0 6 2)))

(setq *net-list-10 '((*net-list-10)
     (supply v1 7 0 5)
     (clock ck1 1 0 5)
     (clock ck2 2 0 5)
     (driver m1 3 10 5 6 2)
     (driver m2 3 10 5 6 2)
     (driver m3 7 10 3 6 2)
     (driver m4 7 10 3 6 2)
     (driver m5 5 1 0 6 2)
     (driver m6 7 1 0 6 2)
))

(setq *net-list-11 '((*net-list-11)
     (supply v1 1 0 5)
     (clock ck1 2 0 5)
     (clock ck2 3 0 5)
     (driver m1 1 2 5 6 2)
     (driver m2 1 10 4 6 2)
     (driver m3 4 2 5 6 2)
     (driver m4 1 10 6 6 2)
     (driver m5 6 10 7 6 2)
```

```
    (driver m6 7 2 5 6 2)
    (driver m7 5 10 8 6 2)
    (driver m8 8 10 0 6 2)
))

(setq *net-list-12 '((*net-list-12)
   (supply v1 1 0 5)
   (clock ph1 2 0 5)
   (driver m1 3 10 2 6 2)
   (driver m2 3 10 0 6 2)
   (driver m3 5 10 1 6 2)
   (driver m4 1 10 5 6 2)
   (driver m5 0 10 5 6 2)
))

(setq *net-list-13 '((*net-list-13—always-high)
   (driver m1 1 4 6 6 2)
   (driver m2 1 5 4 6 2)
   (load m3 1 5 5 4 8)
   (driver m4 2 6 0 6 2)
   (driver m5 1 7 2 6 2)
   (clock ck1 7 0 5)
   (supply v1 1 0 5)
   (clock ck2 8 0 5)
))

(setq *net-list-14 '((*net-list-14-dynamic-clocking)
   (supply v1 1 0 5)
   (clock ck1 2 0 5)
   (clock ck2 3 0 5)
   (driver m1 1 2 4 10 2)
   (load m2 1 6 6 4 8)
   (driver m3 4 3 5 4 2)
   (driver m4 6 5 0 10 2)
   (driver m5 6 2 7 4 2)
   (driver m6 1 2 8 10 2)
   (driver m7 8 3 9 6 2)
   (driver m8 9 7 0 10 2)
   (driver m9 4 3 10 4 2)
   (driver m10 12 10 11 10 2)
   (driver m11 1 3 12 6 2)
   (driver m12 11 2 0 6 2)
   (driver m13 1 2 13 6 2)
   (driver m14 13 12 14 6 2)
   (driver m15 14 2 0 6 2)
   (driver m16 13 2 15 4 2)
   (driver m17 1 3 17 6 2)
   (driver m18 17 15 16 12 2)
   (driver m19 16 2 0 6 2)
   (driver m20 16 3 0 6 2)
   (driver m21 4 3 0 6 2)
   (driver m22 4 18 0 6 2)
   (driver m23 1 19 18 6 2)
   (load m24 1 19 19 4 8)
   (driver m25 4 3 20 4 2)
   (driver m26 21 20 0 16 2)
   (load m27 1 21 21 4 8)
   (driver m28 1 2 27 6 2)
   (driver m29 27 3 23 6 2)
   (driver m30 23 22 0 16 2)
   (driver m31 21 2 22 4 2)
   (driver m32 4 3 24 4 2)
   (driver m33 1 2 25 6 2)
   (driver m34 25 24 26 16 2)
```

```
   (driver m35 26 0 0 6 2)
))

(setq *net-list-15 '((*net-list-15-small-dynamic-clocking)
   (supply v1 1 0 5)
   (clock ck1 2 0 5)
   (clock ck2 3 0 5)
   (driver m1 1 2 4 10 2)
   (driver m9 4 3 10 4 2)
   (driver m10 12 10 11 12 2)
   (driver m11 1 3 12 6 2)
   (driver m12 11 2 0 10 2)
   (driver m13 1 2 13 10 2)
   (driver m14 13 12 14 10 2)
   (driver m15 14 2 0 10 2)
))

(setq *net-list-16 '((*net-list-16)
   (supply v1 1 0 5)
   (clock ck1 2 0 5)
   (clock ck2 3 0 5)
   (driver m1 1 2 5 10 2)
   (driver m2 5 3 0 6 2)
   (load   m3 1 4 4 4 8)
   (driver m4 4 5 0 6 2)
   (driver m5 1 2 6 8 2)
   (driver m6 6 3 0 6 2)
   (cap c1 2 5 .01)
   (cap c2 5 4 .027)
   (cap c3 5 3 .14)
   (cap c4 5 6 .3)
))

(setq *net-list-17 '((*net-list-17)
    (supply v1 1 0 5)
    (load m   1 2 2 4 30)
    (driver m2 2 3 0 6 2)
    (load m3 1 5 5 4 6)
    (load m4 1 6 6 4 6)
    (load m5 1 7 7 4 6)
    (load m6 1 8 8 4 6)
    (load m7 1 9 9 4 6)
    (driver m8 5 2 0 15 2)
    (driver m9 6 2 0 15 2)
    (driver m10 7 2 0 15 2)
    (driver m11 8 2 0 15 2)
    (driver m12 9 2 0 15 2)
))

(setq *net-list-18 '((*net-list-18--super-buffers)
   (supply v1 1 0 5)
   (clock ck1 2 0 5 )
   (load m1 1 12 12 6 4)
   (load m2 1 12 4 10 4)
   (driver m3 12 3 0 10 2)
   (driver m4 4 3 0 20 2)
   (load m5 1 5 5 4 8)
   (load m6 1 5 6 4 8)
   (driver m7 5 4 0 6 2)
   (driver m8 6 4 0 6 2)
   (driver m9 4 2 7 10 2)
   (load m10 1 8 8 4 8)
   (load m11 1 7 9 4 8)
   (load m12 1 10 10 4 8)
```

```
   (load m13 1 4 11 4 8)
   (driver m14 8 7 0 6 2)
   (driver m15 9 8 0 6 2)
   (driver m16 10 4 0 6 2)
   (driver m17 11 10 0 6 2)
   (driver m18 3 2 20 4 2)
))

(setq *net-list-19 '((*net-list-19)
   (clock ck1 2 0 5)
   (clock ck2 3 0 5)
   (driver m1 4 2 5 6 2)
   (driver m2 3 5 6 10 2)
))

(setq *net-list-20 '((*net-list-20--drn-boots)
   (supply v1 1 0 5)
   (clock ck1 2 0 5)
   (clock ck2 3 0 5)
   (load m1 1 2 14 4 8)
   (driver m2 3 14 4 10 2)
   (driver m3 5 2 6 4 2)
   (driver m4 4 6 7 10 2)
   (load m5 1 15 15 4 8)
   (driver m6 15 2 8 6 2)
   (driver m7 8 5 0 6 2)
   (driver m8 4 8 9 10 2)
   (driver m9 9 10 11 10 2)
   (driver m10 5 2 10 5 2)
   (driver m11 9 12 13 10 2)
   (driver m12 5 2 12 6 2)
   (driver m13 14 5 0 10 2)
   (driver m14 7 2 0 6 2)
))

(setq *net-list-21 '((*net-list-21--drain-boot)
   (supply v1 1 0 5)
   (clock ck1 2 0 5)
   (clock ck2 3 0 5)
   (load m1 1 2 5 4 6)
   (driver m2 5 4 0 6 2)
   (driver m3 3 5 6 10 2)
))

(setq *net-list-22 '((*net-list-22--register-cell)
   (supply v1 1 0 5)
   (clock ck1 2 0 5)
   (clock ck2 3 0 5)
   (driver m1 1 3 10 10 2)
   (driver m2 10 14 4 6 2)
   (driver m3 4 2 5 6 2)
   (driver m4 6 5 0 10 2)
   (driver m5 7 3 6 4 2)
   (load m6 1 6 6 4 8)
   (load m7 1 8 8 4 8)
   (driver m8 8 14 9 6 2)
   (driver m9 8 3 4 6 2)
   (driver m10 8 7 0 10 2)
   (driver m11 12 3 13 6 2)
   (driver m12 2 13 14 20 2)
   (driver m13 14 3 0 10 2)
))

(setq *net-list-23 '((*net-list-23--race-conditon)
```

```
   (supply v1 1 0 5)
   (clock ck1 2 0 5)
   (clock ck2 3 0 5)
   (driver m3 5 4 0 10 2)
   (driver m4 5 3 0 6 2)
   (driver m5 6 5 0 6 2)
   (driver m6 6 3 0 6 2)
   (driver m7 7 6 0 6 2)
   (driver m8 1 3 7 10 2)
   (load m1 1 5 5 4 8)
   (load m2 1 6 6 4 8)
))

(setq *net-list-24 '((*net-list-24--xc-xor-gates)
   (supply v1 1 0 5)
   (clock ck1 10 0 5)
   (driver m1 3 4 2 3 2)
   (driver m2 2 3 4 6 2)
   (load m3 1 2 2 4 8)
   (driver m4 3 6 0 6 2)
   (driver m5 4 6 0 6 2)
   (driver m6 7 8 2 6 2)
   (driver m7 8 7 2 6 2)
   (driver m8 7 6 0 6 2)
   (driver m9 8 6 0 6 2)
))

(setq *net-list-25 '((*net-list-25--feedback)
   (supply v1 1 0 5)
   (clock ck1 2 0 5)
   (clock ck2 3 0 5)
   (load m1 1 4 4 4 8)
   (driver m2 6 3 5 4 2)
   (driver m3 4 5 0 10 2)
   (driver m4 5 4 0 3 2)
   (driver m5 4 3 0 6 2)
   (driver m7 6 3 7 4 2)
   (driver m8 4 7 0 10 2)
   (driver m9 7 4 0 4 2)
   (driver m10 4 2 8 10 2)
   (cap c1 8 0 1)
))

(setq *net-list-26 '((*net-list 26--clkout-inverters)
   (supply v1 1 0 5)
   (clock ck1 2 0 5)
   (clock ck2 3 0 5)
   (load m1 1 2 10 6 4)
   (load m6 1 9 9 6 4)
   (driver m2 10 3 5 6 2)
   (driver m3 6 3 11 4 2)
   (driver m4 5 11 0 10 2)
   (driver m5 5 3 7 4 2)
   (driver m7 8 3 9 6 2)
   (driver m8 8 7 0 10 2)
   (driver m9 12 8 0 4 2)
))

(setq *net-list-27 '((*net-list-27--bootstraps-with-clock-skew-errors)
   (supply v1 1 0 5)
   (clock ck1 2 0 5)
   (clock ck2 3 0 5)
   (load m1 1 6 6 6 4)
   (driver m2 6 2 7 6 2)
```

```
  (driver m3 4 2 5 6 2)
  (driver m4 7 5 0 12 2)
  (driver m5 7 3 7 4 6)
  (driver m6 3 7 8 20 2)
  (driver m7 8 2 0 4 2)
  (driver m8 5 7 0 3 2)
  (driver m9 3 10 11 20 2)
  (driver m10 4 2 9 6 2)
  (driver m11 10 9 0 10 2)
  (load m12 1 2 10 4 8)
  (driver m13 3 12 13 20 2)
  (driver m14 4 3 12 6 2)
  (driver m15 3 12 3 4 8)
  (driver m16 13 2 0 10 2)
  (driver m17 13 3 14 6 2)
  (driver m18 15 14 0 10 2)
  (load m19 1 15 15 4 6)
))

(setq *net-list-28 '((*net-list-28--clock-skew-error)
  (supply v1 1 0 5)
  (clock ck1 2 0 5)
  (clock ck2 3 0 5)
  (driver m1 4 2 11 4 2)
  (driver m2 3 11 5 20 2)
  (driver m3 6 7 0 6 2)
  (driver m4 8 5 6 4 2)
  (driver m5 9 3 8 6 2)
  (driver m6 10 9 0 10 2)
  (load m7 1 10 10 4 6)
))

(setq *net-list-29 '((*net-list-29--input-pad)
   (supply v1 1 0 5)
   (pad in1 3)
   (load m1 1 2 2 4 8)
   (driver m3 2 4 0 10 2)
   (driver m2 3 5 4 4 2)
   (clock ck1 5 0 5)
))

(setq *net-list-30 '((*net-list-30--example-1)
 (supply v1 1 0 5)
 (clock ck1 2 0 5)
 (clock ck2 3 0 5)
 (pad in1 12)
 (load m1 1 6 6 6 4)
 (driver m2 4 2 5 6 2)
 (driver m3 6 2 7 8 2)
 (driver m4 7 5 0 10 2)
 (driver m5 3 7 3 4 10)
 (driver m6 3 7 8 20 2)
 (driver m7 8 2 0 6 2)
 (driver m8 4 3 9 4 2)
 (load m9 1 3 10 6 4)
 (driver m10 2 10 2 4 10)
 (driver m11 2 10 11 20 2)
 (driver m12 11 3 0 6 2)
 (driver m55 10 9 0 20 2)
 (driver m13 12 0 0 10 2)
 (driver m14 13 12 0 10 2)
 (load m15 1 13 13 4 8)
 (driver m16 13 3 14 6 2)
 (load m17 1 15 15 6 6)
```

```
 (load m18 1 15 16 8 6)
 (driver m19 15 14 0 12 2)
 (driver m20 16 14 0 16 2)
 (driver m21 17 16 18 10 2)
 (driver m22 18 11 0 10 2)
 (driver m23 1 3 17 20 2)
 (driver m24 17 11 36 6 2)
 (driver m25 36 2 22 6 2)
 (driver m26 21 22 0 10 2)
 (load m51 1 21 21 4 8)
 (load m52 1 23 23 4 8)
 (driver m27 21 3 20 6 2)
 (driver m28 23 20 0 12 2)
 (driver m29 23 8 24 12 2)
 (driver m30 23 3 36 6 2)
 (driver m31 1 2 24 20 2)
 (driver m32 24 8 25 10 2)
 (driver m33 25 31 0 10 2)
 (load m34 1 25 25 4 6)
 (driver m35 31 3 30 6 2)
 (driver m36 30 29 0 6 2)
 (load m37 1 30 30 4 6)
 (driver m56 25 3 28 6 2)
 (driver m38 29 2 28 6 2)
 (driver m39 17 11 28 6 2)
 (driver m40 24 3 33 10 2)
 (load m41 1 34 35 6 6)
 (load m42 1 34 34 4 6)
 (driver m43 35 3 0 12 2)
 (driver m44 34 3 0 10 2)
 (driver m57 35 33 0 12 2)
 (driver m58 34 33 0 10 2)
 (driver m45 17 35 0 20 2)
 (driver m46 27 11 0 10 2)
 (driver m47 17 26 27 10 2)
 (driver m48 26 25 23 8 2)
 (driver m49 26 23 25 8 2)
 (load m50 1 26 26 4 12)
))

(setq *net-list-31 '((*net-list-31)
 (supply v1 1 0 5)
 (clock ck1 2 0 5)
 (clock ck2 3 0 5)
 (driver m1 9 2 8 4 2)
 (driver m2 9 3 5 10 2)
 (load m3 1 4 4 4 8)
 (load m4 1 6 6 4 4)
 (load m5 1 6 7 6 4)
 (driver m6 6 5 0 10 2)
 (driver m7 7 5 0 25 2)
 (driver m8 6 2 0 8 2)
 (driver m9 7 2 0 20 2)
 (driver m10 4 8 0 10 2)
))

(setq *net-list-32 '((*net-list-32)
  (driver m3 1 2 3 6 2)
  (driver m4 6 3 0 6 2)
  (driver m2 4 5 6 6 2)
  (driver m8 3 6 0 6 2)
))

(setq *net-list-33 '((*net-list-33)
```

```
  (driver m2 4 5 6 6 2)
  (driver m8 3 6 0 6 2)
  (driver m3 1 2 3 6 2)
  (driver m4 6 3 0 6 2)
))
```