Enhanced Methods for the Design and Test of CMOS Process Test Vehicles

By

Ugonna Echeruo

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science and Master of Engineering in Electrical Engineering

and Computer Science

at the Massachusetts Institute of Technology

May 22, 1998

Author_____
Department of Electrical Engineering and Computer Science
May 22, 1998

Certified by_____
Anantha Chandrakasan
Associate Professor
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Enhanced Methods for the Design and Test of CMOS Process Test Vehicles

By

Ugonna Echeruo

Submitted to the Department of Electrical Enginnering and Computer Science
on May 22, 1998, in partial fulfillment of the
requirements for the degree of
Masters of Science in Electrical Engineering.

Abstract

Test vehicles used in the verification of chip fabrication processes have been dominated by static RAM arrays. While SRAM's are a very good test vehicle to identify structural faults in the process, they do not truly represent the varied logic circuits that make up about 2/3 of the die area of present day microprocessors. And hence the continued reliance on SRAMS pose a serious problem for process engineers since this leads to a growing disparity in the fault mechanism of products and the test vehicles, which may lead to the masking of potentially serious process defects. A new approach to test vehicle design has been made. The approach replaces the simple RAM cell with a complex but testable unit that implements the various circuit styles and topologies found on modern microprocessors and ASICs. This solves the problem of divergence of product and test vehicle and allows for grater confidence in the suitability of new processes for mass production and hence faster turnaround for each new process generation.

Thesis Supervisor: Anantha Chandrakasan
Title: Associate Professor

Thesis Supervisors: Dilip Bhvsar and Bill Bowhill
Title: Engineering Staff/ Digital Equipment Corporation.

Acknowledgments

I am grateful to Digital Equipment Corporation and especially to Dilip Bhavsar, Bill Bowhill and Dave Akeson, whose time spend on giving advice teaching and lending a helping hand was very much appreciated. Thanks also to all the guys who helped layout the chip.

I would also like to thank my parents and siblings, whose encouragement has been the wind in my sail throughout my studies.

Contents

List of Figures

List of Tables

Chapter 1

1.1 Introduction

As the time between the introduction of new CMOS fabrication processes has precipitously shrunk [1], as can be seen in Figure 1, the challenge for semiconductor companies to maximize their yields has grown accordingly. To meet this challenge, manufacturers of applications specific integrated circuits (ASICS) and microprocessors have begun to develop new generations of test vehicles (structures) to better predict the behavior of the final product during the ramping up of the CMOS process.

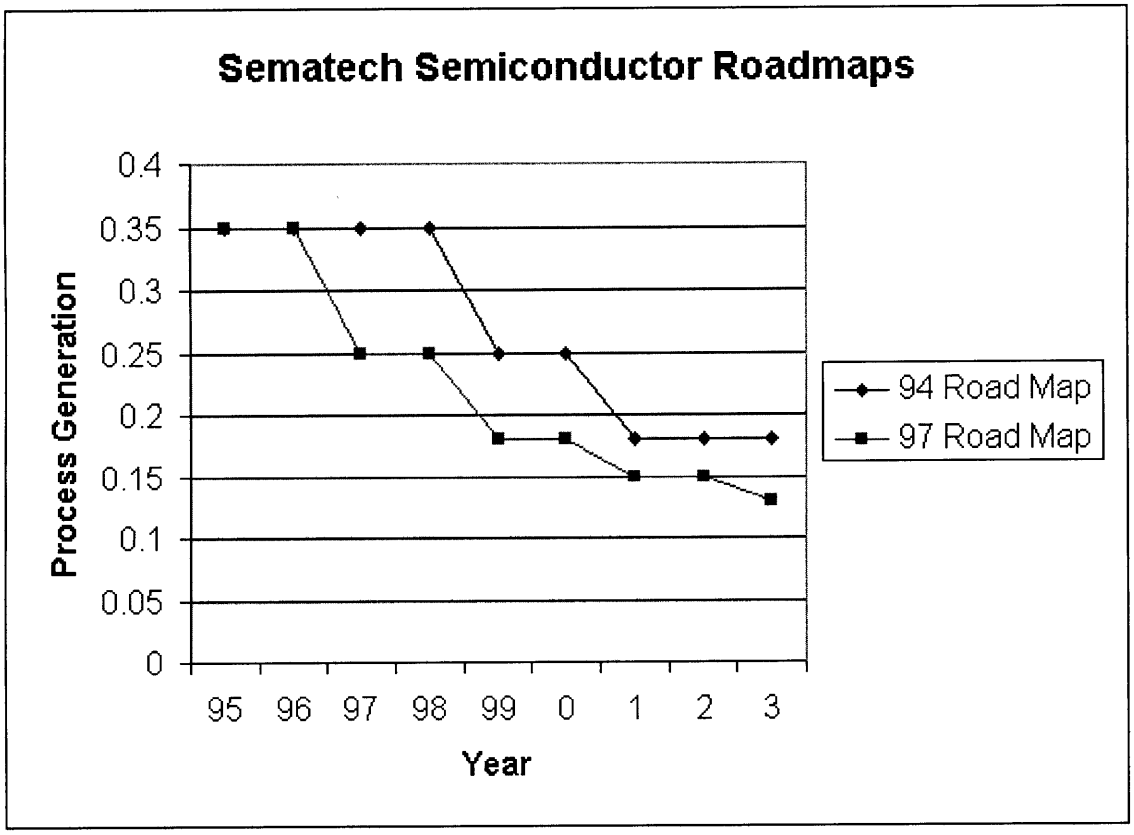

Figure 1 : CMOS generation timelines

Test vehicles used in the verification of chip fabrication processes have been dominated by static RAM arrays. RAM arrays posses several advantages over other techniques for identifying and characterizing fault mechanisms in IC fabrication processes. Firstly SRAM's have a very simple functional model, which consists entirely of the writing and reading of data. Secondly SRAM's also have a simple structural model based on one unique, but small, memory cell,Figure 2 a, replicated in a two dimensional array, Figure 2 b.

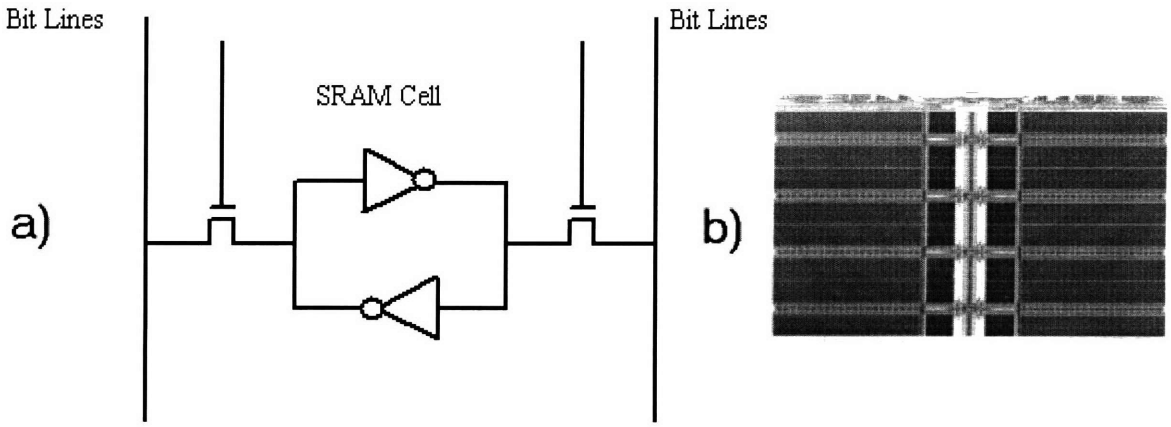

Figure 2 : a) Static Ram Cell b) Static Ram Array

Other vehicles that can be used to detect and characterize fault mechanisms in the CMOS process (process bring up), such as micro-controllers and ASICs, have been hampered by the fact that they do not fully share the strengths of RAM arrays. In particular these vehicles do not posses a simple structural model. For the process test engineer functional fault detection is necessary, but it is far from sufficient. To correctly identify and characterize a process fault the process test engineer has to localize the fault to the transistor level and even to a faulty contact. With the growing complexity of microprocessor circuitry and systems on a chip ASICs it is increasingly

not cost effective to do routine process verification and qualification on actual production chips. This is in part due to the fact that detailed structural models of these complex microprocessors and ASICs are not readily available. Also the structural models for complex logic chips are too complex for current test generation and analysis technology. Because SRAM's have such a simple structural model, it is very easy to produce tests for structural faults such as stuck faults or bridging faults. In addition to this, because SRAM's are built in uniform arrays it is quite easy to localize the faulty transistors or contacts. This has led to the adoption of SRAM's by industry as the standard test vehicle in qualifying semiconductor processes.

While SRAM's are a very good test vehicle to identify structural faults in the process, they do not truly represent the varied logic circuits that make up about 2/3 of the die area of present day microprocessors. And hence the continued reliance on SRAMS pose a serious problem for process engineers since this leads to a growing disparity in the fault mechanism of products and the test vehicles, which may lead to the masking of potentially serious process defects. To remedy this problem and breach the growing gap between test vehicles and product, a new generation of test vehicles that better represent the current complexity of microprocessors and ASICs need to be developed. Two approaches to solve this problem have been considered. The first approach is to replicate a section of a typical complex microprocessor and use it as the test vehicle. While this approach solves the problem of divergence of the test chip from real microprocessor design and potentially shorter lead times, it introduces problems associated with the functional testing of a complex device, as well as controllability and observability issues. Another approach is to design a totally new test vehicle that incorporates varied logic and RAM circuits in an easily controllable and observable manner. While this would seem to be the best approach it has its own disadvantages. Firstly it would need a lot of design resources to implement and verify a sizable design. If the design was simplified and then arrayed to reach the desired size, it would go against one of the design criterions, which was to produce a test vehicle that resembled actual microprocessor and ASIC chips. The solution to this dilemma was to

characterize the features of high performance Integrated Circuits (ICs) to determine which circuit and geometrical structures were not being represented in the SRAM arrays and then create a replicated structure that addressed these issues.

The two principal areas of concern is the representativeness of the test vehicle to product and the provision of Built in Self Test (BIST) and diagnosis functions to facilitate the full identification and diagnosis of fault mechanisms. Figure 3 show the method of identifying and replicating the structural and functional blocks present in product on the test vehicle.

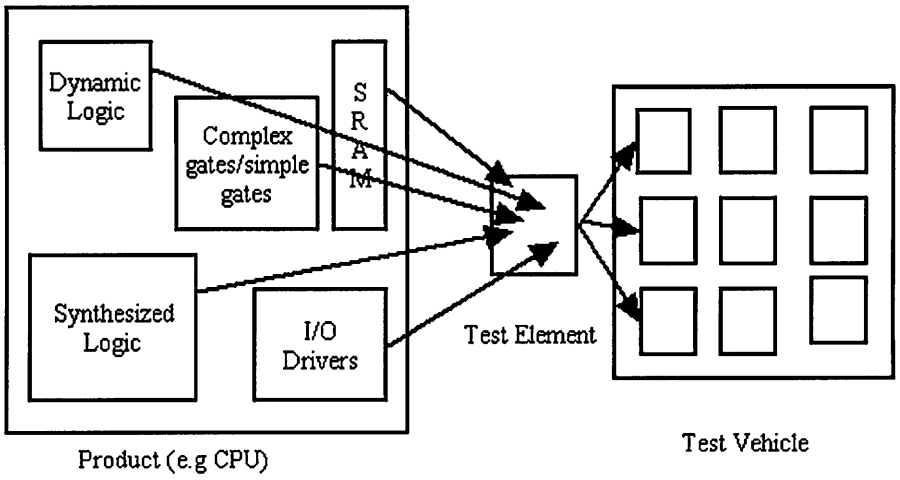

Figure 3 : Extraction of test vehicle from product

The need to find and diagnose fault mechanisms during the qualification of new CMOS processes is driven by the need to reduce the defect level (DL), or ratio of defective parts to defect-free parts, of products run through the FAB. The DL is a function of test coverage T and the manufacturing yield Y [7].

$$DL = 1 - Y^{(1-T)} \qquad (1.1)$$

While the test coverage has remained constant or decreased for very complex microprocessors and system on a chip ASICs, manufacturing yields have had to bear most of the brunt for

reductions in the defect level. This problem is compounded by the increasing use of new process generations to maintain the performance curves of leading microprocessors. The trend is highlighted in Figure 4, which shows the number of CMOS process generations a particular microprocessor design undergoes during its lifetime.

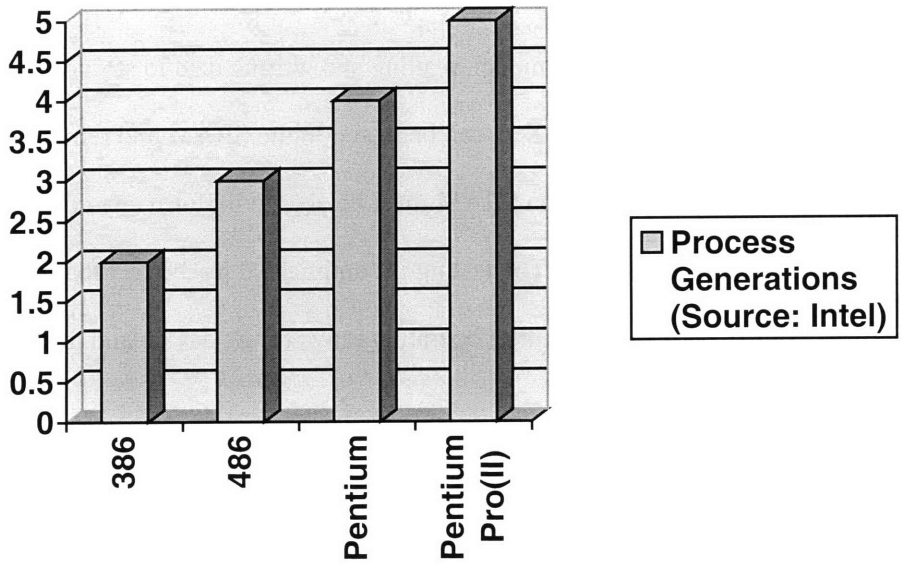

Figure 4 : Process generations in lifetime of microprocessor architectures

The need to further decrease the DL is also spurred by the extremely high cost of replacing defective parts that make it to the customer's field site.

This thesis reports on a method to better predict the defect level of final products during the development period of each new CMOS generations. The goal is accomplished by designing test vehicles that more accurately mimic that fault mechanisms exhibited by the product. By doing so, appropriate action can be taken to remedy any significant manufacturing problems before production begins and in time to inform design engineers of the yield affects of high performance circuit topologies. The target product used for this research is a microprocessor core.

1.2 SRAM Testing Methodology

As stated earlier, SRAMs have been the mainstay of test vehicle design and consequently a lot of effort has been put into developing testing methodologies for the SRAMs. Taking this wealth of experience in designing test procedures into consideration, designs of replacements for the SRAM must leverage past work by using the same basic array structure of the SRAM as well as its simple interface to external logic.

SRAM are tested using its functional model, reading and writing data to storage cells. The goal of the functional testing is to ensure that data can be stored in the SRAM and retrieved at the desired time. To meet this functional model, an SRAM must be able to perform any combination and permutation of data writes and reads. The testing is complicated by the fact that the functional model must be performed across the entire operating range of the device and meet timing requirements at every test point.

The tests conducted on the SRAM are broken down into two categories DC tests and AC tests [9].

The DC tests are as follows:

1. *Address non-uniqueness test*: An address non-uniqueness test will insure that every SRAM cell can be addressed separately and correctly from the input pins. This test is needed to ensure the validity of subsequent test since there is no way of verifying in the read and write instructions that the intended cell is actually performing test.

2. *Stuck at Cell test*: This test is used to verify if a cell in the SRAM array is permanently (or intermittently) stuck at a particular logic value. The causes of this fault are usually process imperfections such as mask alignment.

The AC tests are as follows:

1. *Access time test*: This test is used to find and verify the delay through each SRAM cell is bounded by the desired design and process specifications.

2. *Cycle time test*: This test is used to verify that the SRAM operates at the predicted clock frequency.

3. Set-up and hold time test: These tests are used to verify the proper operation of all latching structures inside the SRAM.

Enhanced SRAMs include Built-in Self-Test (BiST) capabilities which allow the sequential nature of an SRAM to be converted into a combinational path by including a scan path for data in each simultaneous latch stage to be read out sequentially. By converting the SRAM cell into a level sensitive latch, testing of the array can be enhanced to that of level testable. As a level testable circuit the response of the SRAM, its functional model, can be measured independent of the delay between memory elements in the design. i.e Level testable designs also makes the order at which input signal change irrelevant, a feature that greatly simplifies test generation.

1.3 Yield Estimation

Estimating the yield of devices in a given process is done by creating a model for determining the number and spatial distribution of faults in the process. The model chosen is dependent on measured characteristics of the process equipment, but a random distribution is a good priori estimate. Using this model a poisson process for defect distribution is given by equation 1.2 [9].

$$\text{Prob}_i\{X = k\} = e^{-\lambda i}\lambda_i^{\,k}/k! \qquad (1.2)$$

Equation 1.2 gives the probability of having exactly k faults per chip at the *i*th process step, where λ is the average number of faults per chip generated over all processing steps. From this we can determine the fraction of devices on a wafer that are not affected by faults at the end of processing, or in order words the yield as,

$$Y = \Pi Y_i = e^{-\lambda} \qquad (1.3)$$

1.4 Test Coverage

To determine the DL of a given product we finally need to determine the test coverage T, the fraction of all faults that can be tested and detected. Tests are sets of input patterns (vectors) that are input to a device under test and its output compared to a table of correct responses. To get 100% test coverage the set of responses, f_i , to input vectors, v_i, must be able to distinguish between two devices deferring by at least one malfunctioning element.

Thus the test coverage is determined by summing all nodes that exhibit the same response to a vector set when faulty and fault free.

$$T = 1 - (\#nodes, \text{where } \sum f_i(faulty) = \sum f_i \text{ (fault-free)} / \#nodes) \qquad (1.4)$$

Chapter 2

2.1 Characterizing and Reducing Product Divergence

To alleviate the fundamental problem with the current process test vehicles, the divergence between them and complex logic must be reduced. This divergence has been fueled by increased developments in process technology namely the addition of more metal lines and increased logic densities due to improved processes and more aggressive design rules and styles. Figure 5 shows typical sections of an SRAM array a), compared to a logic block b).

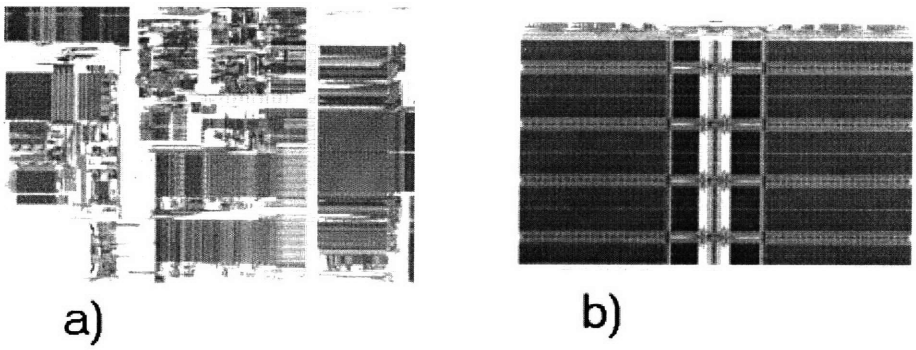

Figure 5 : a) Logic block b) SRAM array

Regular SRAM cells do not need or benefit from all these developments and thus are not designed to include them. To minimize the divergence for test purposes, SRAM cells can be modified to include extraneous metal lines and circuit styles. But this comes at a cost of reducing the test coverage since a paths for controlability and observablity must be added to the design for each new feature that is added to the basic cell. Since the SRAM cell must be modified significantly to bring it closer to the logic block, we might as well make a clean break from the

past and design a test vehicle dedicate to CMOS process bring-up instead of incremental changes that do not entirely address the problems at hand.

To arrive at a new cell to replace the SRAM an analysis of the features of current logic circuits and layout topologies must be done. The test case for this analysis was an advanced microprocessor, the Alpha 21264 chip, by Digital Equipment Corporation. Figure 6 shows a micrograph of the microprocessor.

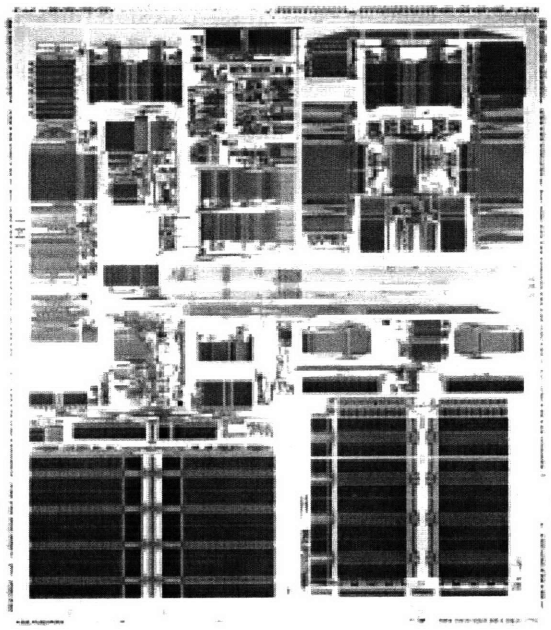

Figure 6: Digital Alpha 21264 Microprocessor

2.2 Representative Logic

The task of creating representative logic has been focused on microprocessors, which show the greatest divergence from current test vehicle technology. The deferent circuit and topological features of the microprocessor have been identified and divided into several functional groups to better asses their representativeness in test vehicles.

1. RAM structures
2. Synthesized Logic
3. Random Logic
4. Data paths
5. Control paths
6. Decoders
7. Interconnect
8. Topology and Layout
9. Drivers/Receivers
10. Special Circuits

Each of these functional groups consists of several circuit structures which are broken down into the various groups and list of structures that need to be recreated in the test vehicle to insure that the test vehicle is representative. The following is a brief description of each functional group and the circuit structures classified under them.

1. RAM structures:

   These consist of small memory cell arrayed into large structures and are used as memory storage blocks. Examples of which are:

   - Caches
   - Register Files

2. Synthesized Logic

   Synthesized logic are usually random control blocks that are generated by CAD programs and exhibit a varying degree of regularity depending on the size of the macro cell used to compile the structure. Examples of these structures can be found in the:

   - Memory Controller
   - Integer/Floating point Mappers

- Bus Interface Unit

3. Random Logic

   Random logic blocks are scattered around the processor core and are exemplified by the lack of any replication. Examples of random logic are located in:

   - Memory Controller
   - Integer and floating point execution units

4. Data paths

   These are topological structures, usually made up of wires and transistors that direct the flow of data through the functional blocks of the microprocessor. They can be found almost every where in a microprocessor but are usually identified in the:

   - Integer/Floating point execution units
   - Data and Control Busses

5. Control paths

   These are somewhat identical to the data paths except that they carry control instructions to the various sections of the microprocessors and are usually orthogonal to the data paths. They also contain significant amount of random logic. Control paths can be found in the:

   - Integer/Floating point execution units
   - Data and Control Busses
   - Memory Controllers

6. Decoders

   Decoders are structures that exhibit a significant amount of repetition, and are used often enough in the design of microprocessors to have a significant effect on yield. Though decoders are used throughout the design their densities are highest in these sections:

   - Integer/Floating point execution units

- Caches
- Instruction Data Path

7. Interconnect

   The transistors and larger blocks on the microprocessor are all connected through the use of various forms of interconnect. These include poly-silicon and metal layers. Various functional blocks of the microprocessor have differing degrees of utilization of any particular interconnect layer but the processor as a whole tries to maximize the use of all these layers to transport signals as well as power supplies.

8. Topology and Layout

   This issue is concerned with how the various transistors and interconnect layers are arranged to form the final product. This is an issue because fabrication processes are affected by the orientation of transistors and metal layers on the circuit to be fabricated. An example of this is seen in the reduced yield of beveled interconnect. Examining how topological differences in larger structures that are absent in SRAM arrays is the main concern.

9. Drivers/Receivers

   These are usually large devices that are used to communicate over long distances such as to other chips on the motherboard. Examples can be found in:

   - I/O Pads
   - Data and Control Busses

10. Special Circuits

    Special Circuits refers to the various circuit styles that are sometimes used in each of the structures listed above. The circuits listed below define the eventual layout decisions that are made in the design of the microprocessor and thus have a strong impact on the fault mechanisms of the processor.

Table 1 : Special Circuit Types

Logic gates (simple/complex)
Multiplexers (MUX)
PLA
De-coupling capacitors
ESD structures
Clock distribution
Bit lines
Cascade logic
Large drivers
Dynamic logic
Cascode logic
Latches
Carry chains
Strapped poly
Oscillators
Pass Logic

In this initial design, numerical enumeration of the percentage coverage of all the circuit styles listed above was not attempted. The goal of the project was to define a framework for analyzing the various components of the product individually and then determining the factors that are most likely to affect yield.

Chapter 3

3.1 Design

Discussion of the design strategies employed in the logic test vehicle shall now procced with brief descriptions of the various aspects of the design.

3.2 Motivation

Development of the test vehicle is motivated by several factors among which are:

1. Complement Current SRAM testing
2. Alleviate identified shortfalls of RAM testing
3. Greater representation of circuit styles and products
4. Better use of metal layers
5. Test-bed for designs to be used in the future
6. Built In Self Test (BIST)
7. Smart error logging capability
8. Operation at Speed (300Mhz)

3.3 Chip Architecture

The test vehicle maintains the basic SRAM array structure to facilitate easy diagnosis of detected faults as well as to reduce the design effort and create a scalable architecture for use in other generations of the test vehicle. The RAM cell that characterizes SRAM arrays is replaced by the Logic Test Element (LTE) as the basic building block. The LTE incorporates most of the circuit and some of the topologic styles that were identified as needed to create a test vehicle that is representative of product.

The LTE's are grouped into 2 symmetric blocks of 32 LTEs, which make up a cluster consisting of 64 LTEs. Within the cluster each LTE is uniquely addressable and its output can be

read separately. Multiple clusters are then arrayed to make up the test vehicle. The goal is to create a 4x4 array of clusters consisting of approximately 700,000 transistors.

Hedged between the clusters is the centralized control and support logic, which interfaces the test vehicle with external logic. It is also responsible for generating addresses and monitoring failures that occur in each cluster.

3.3.1 Logic Test Cluster

The Logic Test Cluster (LTC) consists of 64 LTEs, grouped into 2 symmetric blocks of 32 LTEs. Figure 7 shows a block diagram of the structure of the LTC.

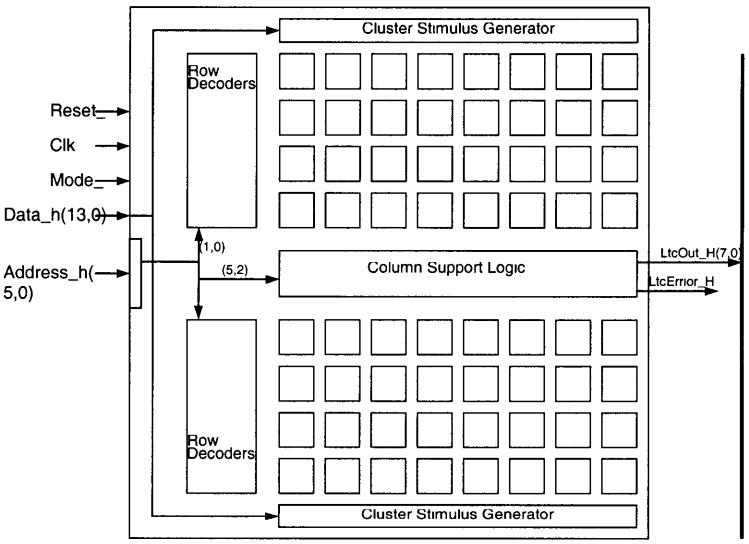

Figure 7 : Logic Test Cluster (LTC)

3.3.2 Logic Test Element (LTE)

The LTE incorporates most of the circuit and some of the topologic styles that were identified as needed to create a test vehicle that is representative of product. In This case the LTE consists of a 4 bit ALU similar to the 74X181. The choice of this particular device was due to its complexity as well as the reasonable number of vectors needed to fully detect faults as well as diagnose the failure. A more in-depth discussion of the reasons for choosing the 4-bit ALU, is contained in

chapter 4. The LTE incorporates approximately 700 transistors. Figure 8 shows a block diagram of the LTE

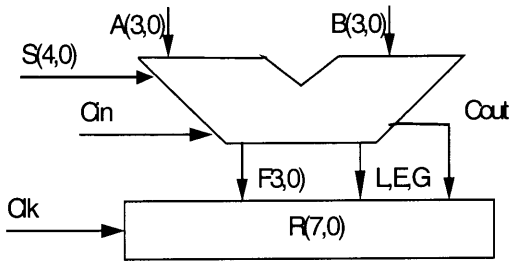

Figure 8 : LTE Block Diagram

3.3.3 Control Logic

The centralized control and support logic interfaces the test vehicle with external logic. It is also responsible for generating addresses and monitoring failures that occur in each cluster. The control logic is designed to out live The LTEs that make up the bulk of the test vehicle.

3.3.4 Logic Test Vehicle (LTV)

The Logic Test Vehicle (LTV) is then made from arraying 16 LTCs. Each LTC can be addressed by the control logic and the output from each LTE contained in the LTC directed to off-chip test equipment. Figure 9 shows a block diagram of the LTV.

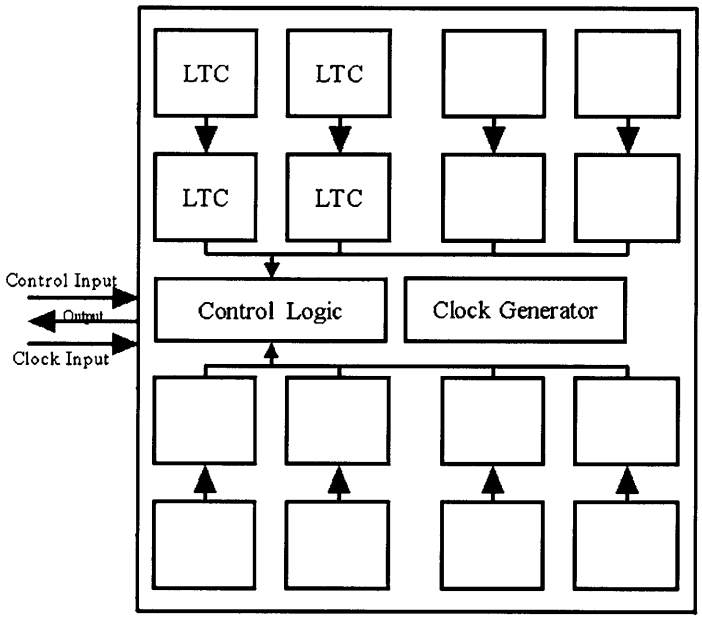

Figure 9 : Logic Test Vehicle (LTV)

The replication done for this generation of test vehicle can be expanded to fill any desired die size by arraying the LTV itself.

Chapter 4

4.1 Implementation

The implementation of the Logic Test Vehicle was broken up into three stages. First a behavioral model of the LTV was written from the architectural specifications. The behavioral model was used to simulate LTV and iron out implementation details such as communications protocols between the LTV and external test logic. A listing of the behavioral model is included in appendix B. The second stage involved generating schematics from the behavioral model. The schematics, available in appendix C. The schematics were then verified using a Boolean verification tool against the behavioral model to insure there were no inconsistencies. Finally Layout of LTV was done from the schematics, as well as backend verification of the layout to insure that it met the design rules of the .28u process that the LTV was to be manufactured in.

4.2 Logic Test Element (LTE)

The logic test element (LTE) is the basic building block for the test vehicle. The LTE is designed replace the SRAM memory cells. To accomplish this the LTE must be fully testable and diagnosable. In digital logic circuits this criterion is met through solving controllability and observability issues. Controllability is the ability to establish a specific signal value at each node in a circuit by setting values on the circuits inputs [2]. Observability is the ability to determine the signal value at any node in a circuit by controlling the circuits' inputs and observing its output. For an SRAM cell this task is not very complicated since the ram cell's ratio of logic to inputs and outputs is high. Figure 10 shows how the different paths in an SRAM cell are sensitized for controllability and observability.

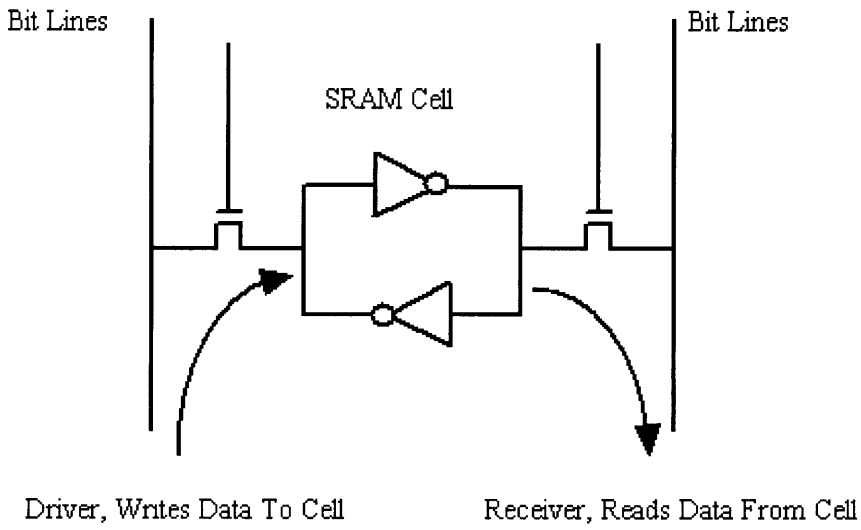

Figure 10 : Sensitization of SRAM cell for controllability and observability.

To effectively migrate to the LTE as the basic building block, its observability and conrollability must be as close as possible to that of the SRAM for the test engineers to effectively use the test vehicle to develop the process. This fact runs counter to the needs of making the test vehicle representative of product, which in this case is a very complex microprocessor. To solve this problem the choice of functions that comprise the LTE is severely limited. This restriction led to the adoption of a simple 4-bit Arithmetic Logic Unit (ALU) as the function to be performed by the LTE.

The 4-bit ALU was chosen because it provided a relatively simple functional model that was easily modeled to enhance automatic test vector creation and testing. It also allowed the easy detection of structural faults through the application of a limited set of test vectors, which can then be thoroughly examined to determine the particular process parameter that was responsible for a failure. Another benefit of the 4-bit ALU is that it is several orders of magnitude more complex than the RAM cell. This added complexity allows for the replication of different circuit structures and styles within each LTE. A palette of LTE's incorporating many different variants

of the circuit styles determined relevant for the particular design can be compiled and assembled for the test vehicle. Figure 11 shows the gate level logic schematic of the LTE.

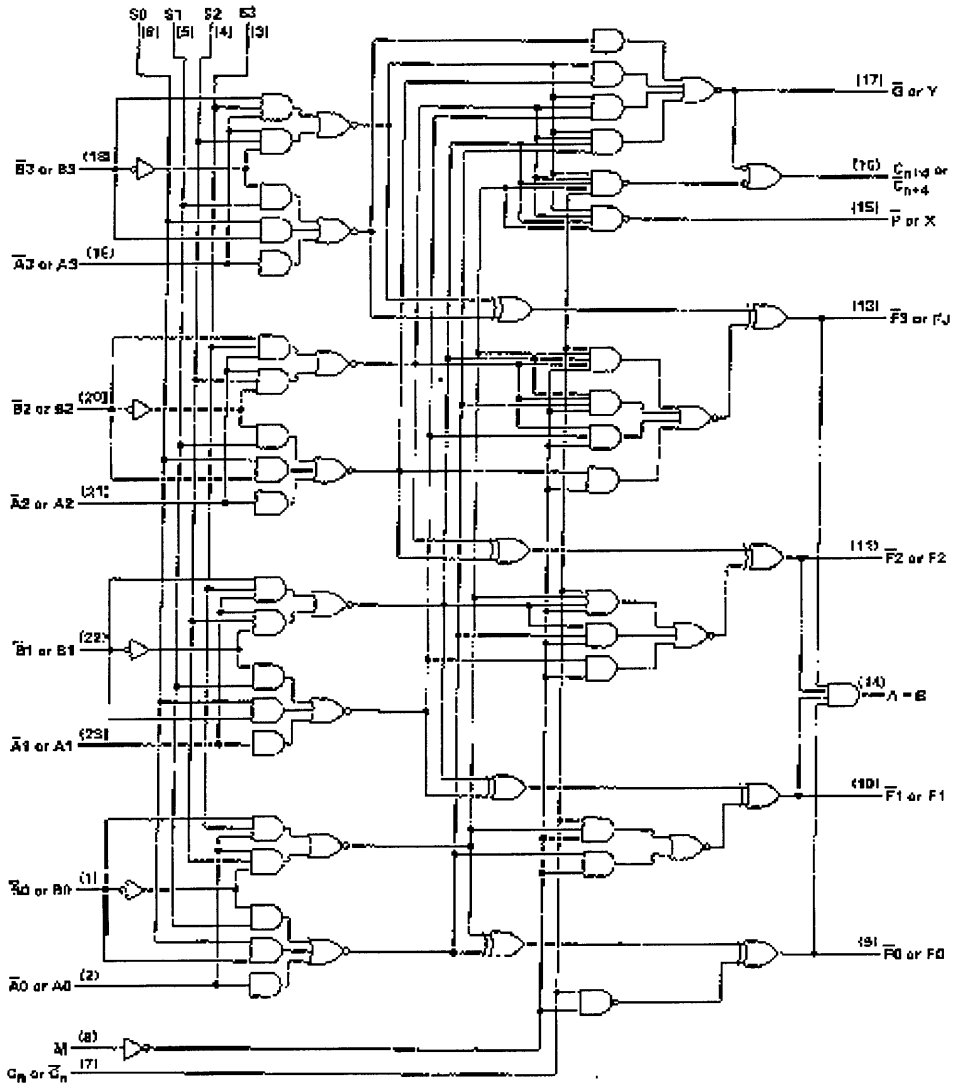

Figure 11 : LTE Logic Diagram

The actual circuit styles employed in the final implementation of the LTE can be seen in circuit diagram of the LTE in appendix C.

4.3 Analysis of the testability and diagnosability of the LTE

Once the structure of the LTE has been arrived at, an analysis of its testability and diagnosability can proceed. This is done by analysis of the structural fault models, the predicted failure mechanisms, associated with such a structure, namely [2]:

1. Short - a short is formed by the connection of two nodes not intended to be connected to each other. Extra conducting material placed across wires during fabrication usually cause this fault. Figure 12 shows an example of a short.

Figure 12 : Short between metal lines [5]

2. Open - an open is formed as a result of a break in the connection of two nodes in a circuit. Opens are usually cased by the absence of conduction material during the fabrication of devices or the wearing away of material due to chemical action or electromigration. An example of an open can be seen in Figure 13.

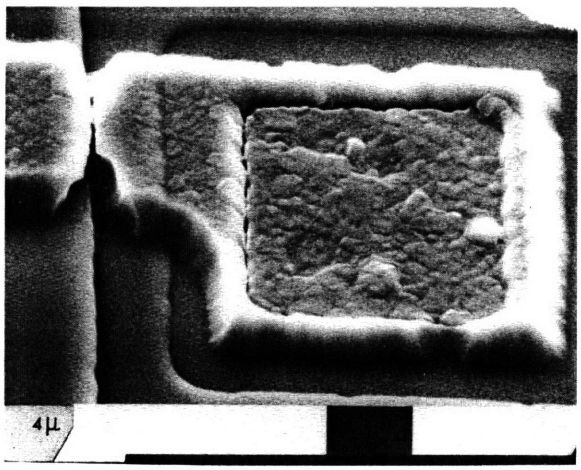

Figure 13 : Break in metal line [5]

These two structural fault models combine to create the two logical fault models we shall consider:

1. Stuck at fault.

   The stuck at fault is when a node in a circuit appears to be held at a constant value independent of the inputs to the circuit. This fault is usually cause by a short between the node and a power or ground line, or an open that leaves a node at a low value indefinitely.

2. Bridging fault

   The bridging fault is caused by shorts between two signal wires, which result in a new logic function from the combination of signals.

The approach employed to detect these structural faults is based on analyzing the functional model of the ALU. To do this several assumptions are made about the fault mechanisms of the CMOS process. First there is an assumption that there is at most one logical fault in the system. This simplifying single-fault assumption, is justified by the frequent testing strategy, which states that we should test a system often enough so that the probability of more than one fault developing between tests is close to zero. Secondly structural fault models assume that

components are fault-free and only the interconnect is affected, the problem of discriminating between interconnect and devices is an area of added research.

In this technique the output of the ALU to any specific combination of inputs is a function of the inputs as well as any structural fault present in the device. To simplify the analysis an assumption that only a single fault can occur between each test and diagnosis phase.

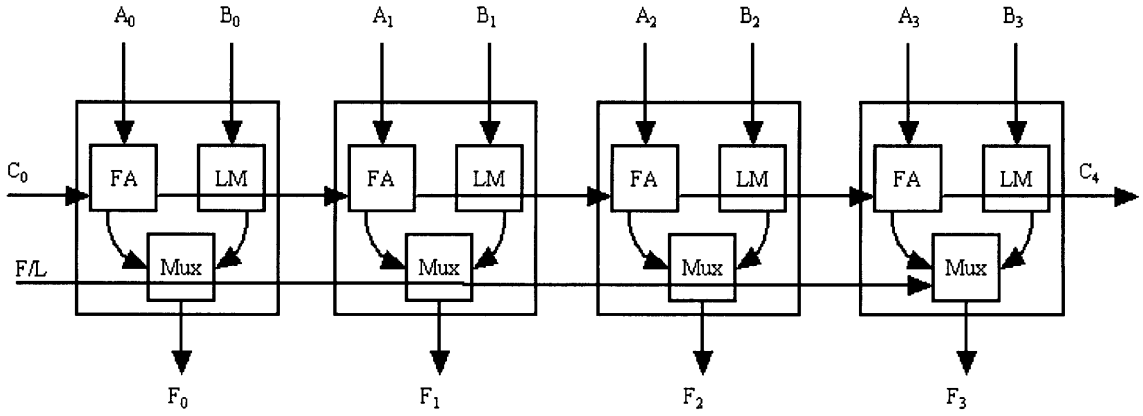

Figure 14 : Array implementation of a 4-bit ALU slice

All the building blocks of the ALU shown in Figure 14 are combinational and it is assumed that any fault in the ALU will also leave it as a combinational circuit. Therefore a functional fault $F(s)_j$ will modify the output of the ALU to any given input ip_j from its fault free output o_j to o'_j. This translation is denoted by the term $f_{j,(o,o)}$. The functional fault $F(s)$ is therefore the sum of all input patterns that produces an erroneous set of outputs.

$$F(s) = \Sigma f_{j,(o,o)} \qquad (4.1)$$

The number of possible errors for an n-input, m-output module is [3]

$$2^n(2^m-1) \qquad (4.2)$$

Since performance is not a major goal in the design, the simple array ALU with independent modules for each bit, makes testing easy. This is possible since each input bit of the ALU can be addressed and its corresponding output analyzed, thereby the problem of testing the ALU can be

decomposed into one of testing each of the ALU modules, the FA, LM and MUX shown in Figure 14. These individual module tests are then addressed to each ALU contained in the test vehicle to achieve the desired coverage for the entire test vehicle. By doing this the vector length to test the entire test vehicle is significantly reduced making test generation easy and efficient.

With minimal effort put into the analysis of testing schemes that can be applied to the test vechicle, its simple functional model enables the test engineer to simply use an exhaustive testing scheme. Exhaustive testing is were all the possible stimuli are applied to the test vehicle and its output are compared to a table of correct responses generated by a software model of the test vehicle. Several researchers have published valuable information on fault analysis of ALU structures, including Hayes and Sridhar [4], the testability analysis concluded above was based on work done by Blanton and Hayes [3].

4.4 At Speed Test

In addition to the problems of circuit divergence, test vehicles also face a problem of operating at the same frequency range as products. With the astonishingly high frequency ranges of modern microprocessors, testing at speed is becoming a critical aspect of qualification, because of reliability concerns due to power fluctuations and transistor characteristics at these speeds.

Test vehicles have been slow to increase their clock speeds because of limited design resources and the unavailability of inexpensive logic testers that can operate at high frequencies. A solution to this problem that is implemented in the logic test vehicle is to add a high frequency on-chip clock generator to the test vehicle as well as a low frequency external clock input from the logic tester. The internal clock source can be used with the onboard BIST capability to fully detect faults in the test vehicle as well as during diagnosis to measure timing characteristics at speed. On detection of a fault in the logic test vehicle, the test vehicle transitions its clock from the internal clock source to the external clock, for the tester to read out the failure information. For the external logic testers to communicate with the logic test vehicle during diagnosis or fault

detection, the external clock source is used. The external lock source can also be used to operate the logic test vehicle at any frequency. To reduce the design resources needed to implement the internal clock source a multi-tap ring oscillator was chosen as the clock source. The delay of each intermediate ring of the oscillator was chosen to produce a range of frequencies between 400Mhz and 50Mhz. The particular tap that will be used is programmed into the logic test vehicle at power up from the input pins.

To guarantee proper operation of the logic test vehicle during transitions from the internal clock to the external clock source, a clock arbiter is added to de-glitch the global clock of the test chip. A description of the clock arbiter is provided in chapter 5.

4.5 BIST/Monitoring Architecture

As well as satisfying its representative value, the test vehicle must be easily tested. This is accomplished by the inclusion of self-test hardware on the test vehicle to perform Built in Self-Test (BIST), as well as facilitate failure monitoring and diagnosis. The implementation strategy is to surround the representative logic with BIST logic for life cycle tests. Life cycle tests are used to simulate failures in field deployment of product, by creating an accelerated failure environment in a burn in oven through increased temperatures and voltage stresses. The test vehicle is also designed to easily facilitate off-line fault detection and analysis. The BIST circuitry will emphasize the functional model of the design and will be accomplished through redundancy failure detection. Diagnosis of failure data and fault model detection and analysis will rely on a direct scan architecture, which will utilize automatic test program generation (ATPG) to generate test vectors for fault detection and identification.

Development of ATPG tools and interfaces for use with transistor level analysis is an area of research that is needed to fully develop the capabilities of this architecture. Since the

observerbility of many of the nodes in the test vehicle is very low, very efficient ATPG tools will be necessary to fully maximize fault detection and diagnosis the of the test vehicle.

4.6 Chip Operation

The test vehicle has two operating Modes:

1. BiST Mode
2. Off-line Test Mode

In BiST mode the test vehicle generates pseudo random stimulus to all LTEs and compares the results of a pair of LTEs to determine if there has been a failure. Failures are then communicated to external test logic.

In Off-line test mode stimulus can be loaded into each LTE and response data of an addressed LTE read out by external logic, or a combination of internal generation of stimulus and external generation of addresses can be employed, with each LTE exhaustively stimulated from primary inputs. The control specifications of the test vehicle are included in appendix B.

Chapter 5

5.1 Design of Logic Test Vehicle (LTV)

The following chapter discuses the design of the Logic Test Vehicle (LTV). The discussion will include the blocks of the LTV already discussed above as well as several other aspects of the LTV including its clocking scheme and power analysis. The design of the LTV started with the creation of a behavioral model of the LTV then followed by the creation of the schematics and then physical layout of the device.

The LTV was targeted at the CMOS 7 (enhanced .28 micron) process of Digital Equipment Corporation. Figure 15 shows a layout plot of the LTV.

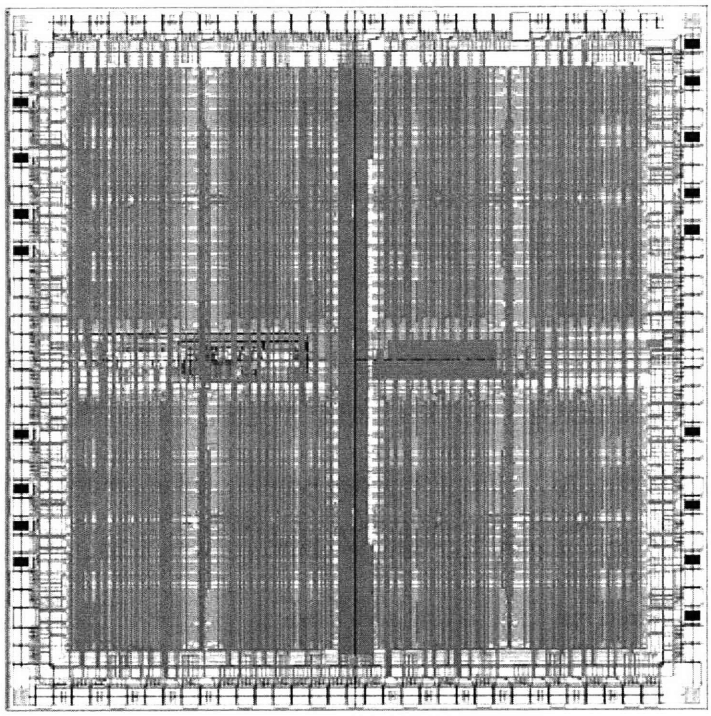

Figure 15 : Logic Test Vehicle

There were several requirements on the design that were encountered. They included:

1. Completion of the architecture, specification and design of Logic test vehicle in 6 months.
2. Area limitation of 34,000 CDU (4.76mm) on a side
3. High frequency operation, (target frequency of ~300Mhz)
4. Minimal external test equipment (low pin count)
5. External and internal clock generation

These requirements were met in the design and are discussed in some depth in the following sections. The tools used in the design were provided by Digital Equipment Corporation, and included:

1. Behavioral modeling tools used to simulate the entire LTV structure and generate test patterns.
2. Schematic entry and simulations tools.
3. Boolean verification tools used to verify the consistency of the schematics with the behavioral models.

The behavioral models, as well as schematics for the LTV are included in appendix B and C respectively.

5.1.1 Logic Test Element (LTE)

The design of the LTE was based on the 7X181 ALU. While not a simple array ALU it exhibits many of the traits of the array ALU. It is also widely studied and understood. The LTE is the basis of the representativeness that the test vehicle exhibits, and as such the design of the LTE takes into account the varied logic and layout styles that are present in the target product (Alpha 21264 microprocessor). Figure 16 shows a block diagram of the logic design styles implemented in the LTE.

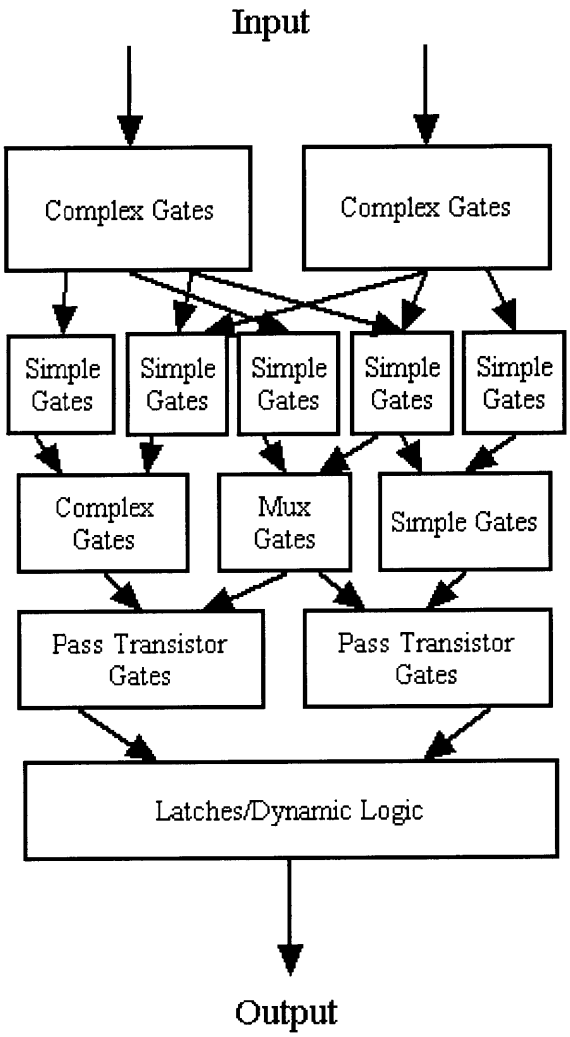

Figure 16 : Logic design styles in LTE

The first stage of the LTE consists of complex gates these in turn feed the second stage which is made up of a broad range of multi input simple gates. The third stage of the LTE consists of mixture of static MUX implementations as well as some complex logic. The fourth stage is implemented with PASS transistor logic that feeds the final stage, which is a set of latches implemented with sense amps, a form of dynamic logic. Figure 17 and Figure 18 show the contrast between the layout of a RAM cell with the LTE cell.

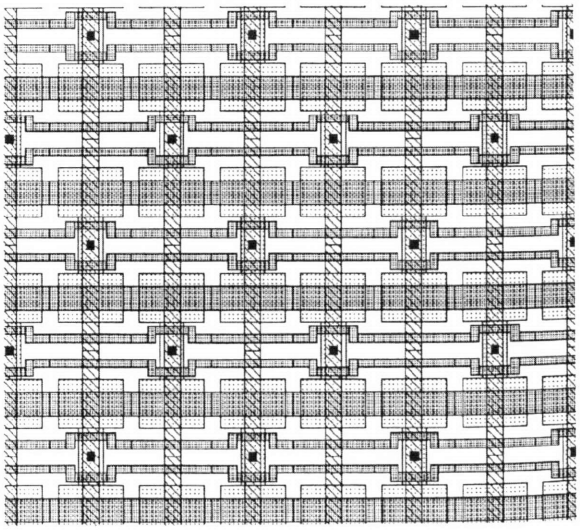

Figure 17: Layout of RAM cells

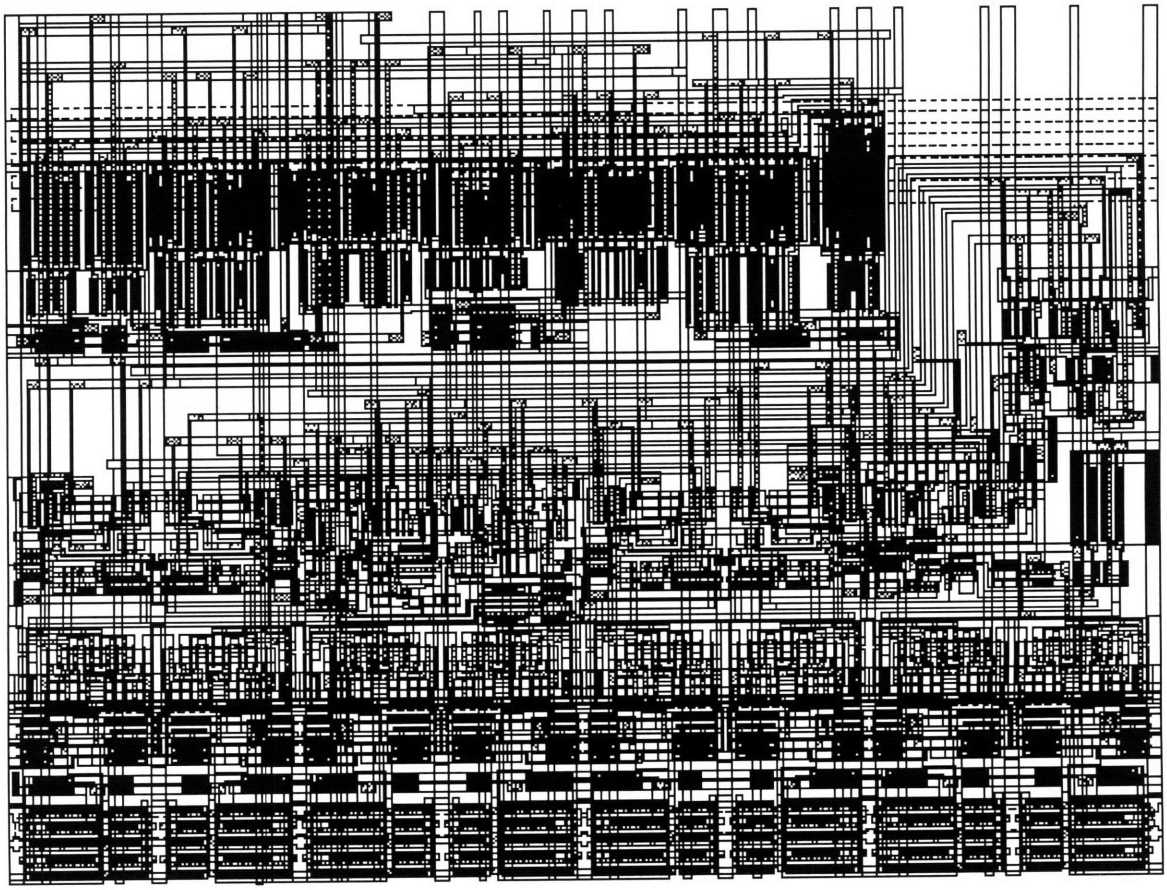

Figure 18 : Layout of LTE cell

To allow the LTE cells to be arrayed and addressed, tristate drivers are added to the outputs of each LTE.

5.1.2 Logic Test Cluster (LTC)

To facilitate the aggregation of several hundred LTE's onto the test vehicle. The LTE are grouped into another structure called the LTC. The LTC is also used as the basis for the BIST and monitoring support of the test vehicle.

An important issue in the design of the LTC was determining whether the outputs of individual LTE cells are to be accessed sequentially or in parallel during self-test and offline testing. The various pros and cons of each strategy are analyzed in Table 2.

Table 2 :Tradeoff in Parrallel vs. Sequential LTE access

Issue	Parallel	Sequential
Array size	Larger, more signal wires.	Smaller.
Address Generation	Smaller infrastructure.	Larger, unique address needed.
Row Decoding	May not be needed if all outputs are available for comparison.	Needed.
Utilization of Address decoders	If present then will be run at the same rate as in sequential case.	Used every cycle (may hamper off-line testing if a fault occurs).
Column Decoding	Not needed.	May be needed to reduce multiplexing of outputs.
Column Multiplexer	Large multiplexer needed, May need to break it up into multiple stages. (Looks like the sequential case).	Needed if no column decoding is used.
Comparator	Row comparison or array comparison may be done depending on the scale of parallelism.	Parallel comparison may still be used, but serial comparison uses more of the same path as off-line testing.
Testablity	Decoders may still have to be made testable.	Decoders need to be testable.
Control	Less complex	More complex.
Similarity to SRAM	Not as similar	Can be made to be identical.
Error Generation	Extra work needed to extract error from individual comparison (if needed)	Not needed.

From these tradeoffs it seems that either strategy, parallel or sequential, will eventually resemble the other because of implementation constraints. The only difference between the two is solely a matter of ease of implementation. The benefit of the parallel approach is that less address signal lines must be generated in the central control logic and distributed throughout the test chip. Another benefit of the parallel approach is that the interval between when each LTE is tested is reduced and hence test-throughput is increased, a need that has been identified by industry experts [10]. This advantage is somewhat reduced due to the added work needed to extract the identity of a particular failing LTE cell.

The LTC consists of two 8x4 arrays of LTE's, each with independent linear feedback shift registers (LFSR) that serve as pseudo random stimulus generators for LTE data and control inputs. LFSR's as shown in Figure 19, are cyclic elements that go through a fixed sequence of states when clocked. In this implementation the ability of loading the LFSR with arbitrary data from external test equipment is provided. This enables the stimulation of the LTE's with specific inputs during offline testing. The choice of LFSR's for the cluster stimulus generators, as opposed to counters, was based on the simplicity of their design, low area overhead, and possible reduced failure analysis to determine if they are functioning properly.

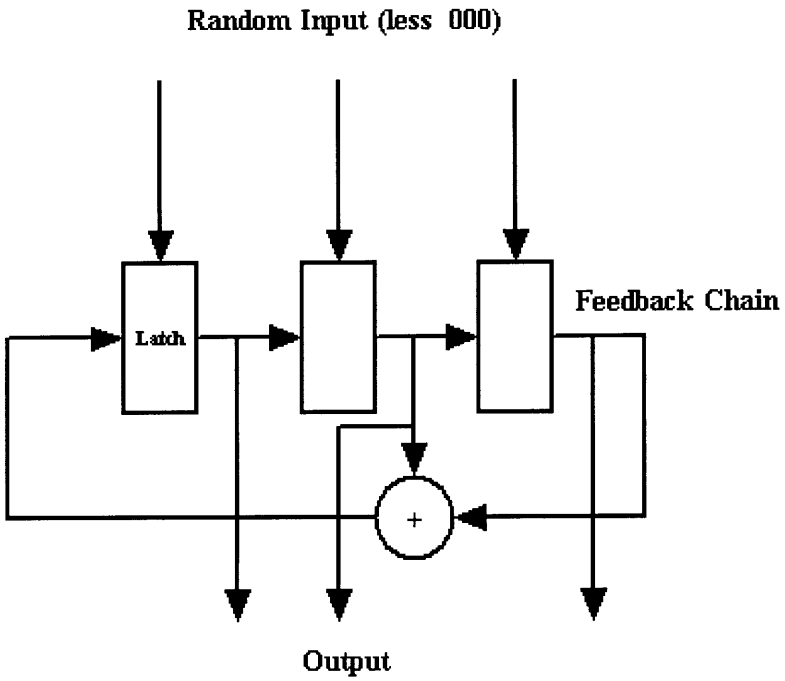

Figure 19 : Example Linear Feedback Shift Register (LFSR)

Each column and row of the array is fed a permuted version of the LFSR to help detect errors in the decoders for each array as well as to vary activity within the array. The locations of the sections discussed above can be seen in the plot of the LTC shown in Figure 20.

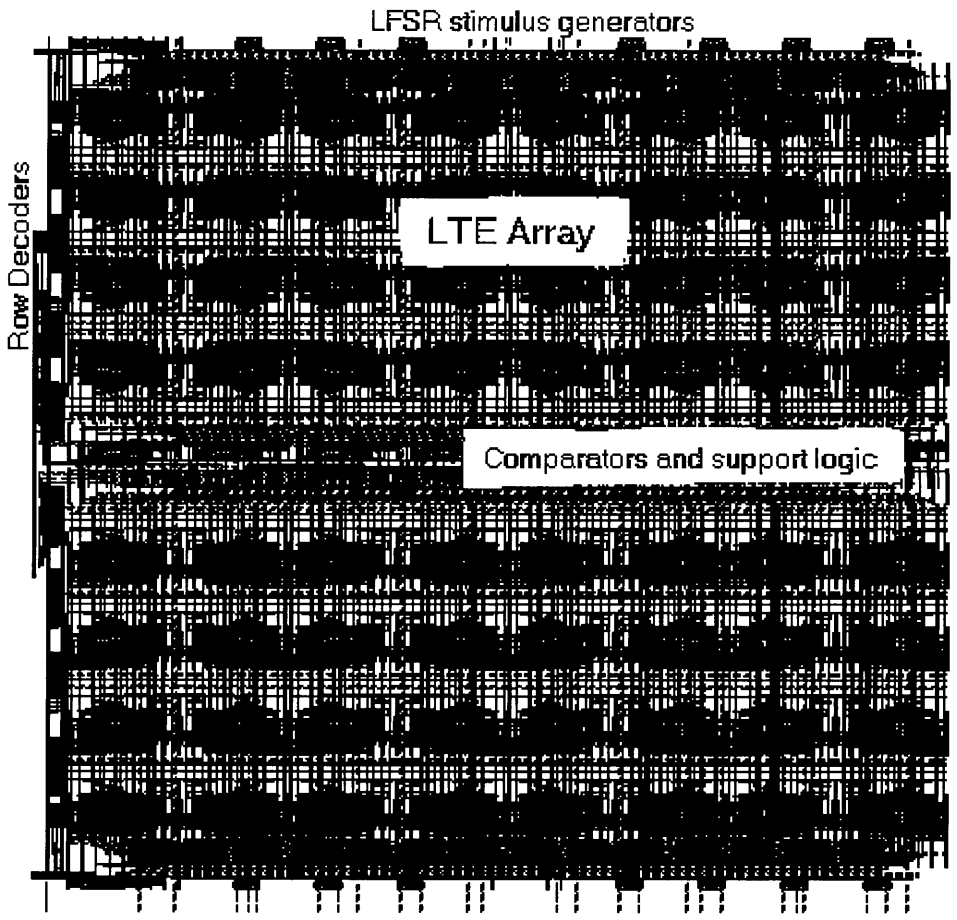

Figure 20 : Plot of Logic Test Cluster (LTC)

Embedded between the LTE arrays is the column support logic, its duty is to compare the outputs of each row of LTE's from the upper and lower arrays to check for mis-compares and hence failures in any of the LTE's. The comparison is made every cycle and once a failure is detected a signal to the central control logic is generated. The column support logic also multiplexes the output of a selected LTE to the central control logic for extraction by external test logic.

5.1.3 Central Control logic

The central control logic monitors the activity of all the LTC's arrayed onto the test vehicle. It also interfaces the test vehicle to external test equipment. The central control logic posses two modes:

1. Command Mode

In the Command mode the control logic waits for, or executes commands from the external test equipment. The control logic also defaults to this mode when a BIST self cycle as finished.

2. Run Mode

In the Run mode the control logic generates successive address for each LTC to perform a BIST or user controlled test cycle.

A block diagram of the control logic can be seen in Figure 21. A detailed explanation of the workings of the control logic can be found in appendix C.

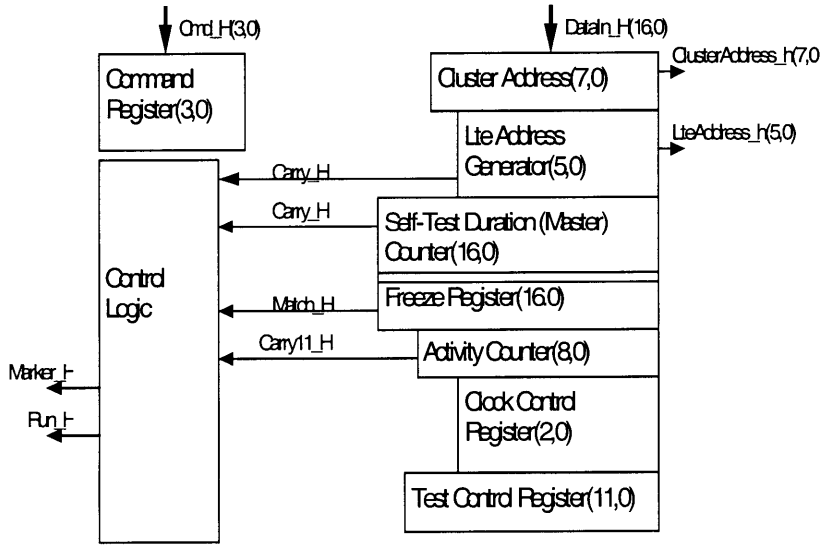

Figure 21 : Control Logic

The control logic consists of several registers and decoders. These registers control the address of the LTE to be analyzed, the vector to stimulate the LTE with, and markers to determine the end of the test cycle.

5.1.4 Clock generation and Distribution

As with all synchronous systems the test vehicle needs a clock source to synchronize all the activity within it. This clock source must be free of excessive skew induced by asymmetric wire delays from the clock source to clock sinks. To solve this problem designers have implemented several clock distribution networks optimized for different chip topologies. An example of one topology is the hierarchical clock-buffering scheme. In this method clock skew is kept at a manageable level by creating a symmetric clock-buffering tree that keeps the clock skew along each branch at a manageable level [4]. A block diagram of such a scheme is shown in Figure 22.

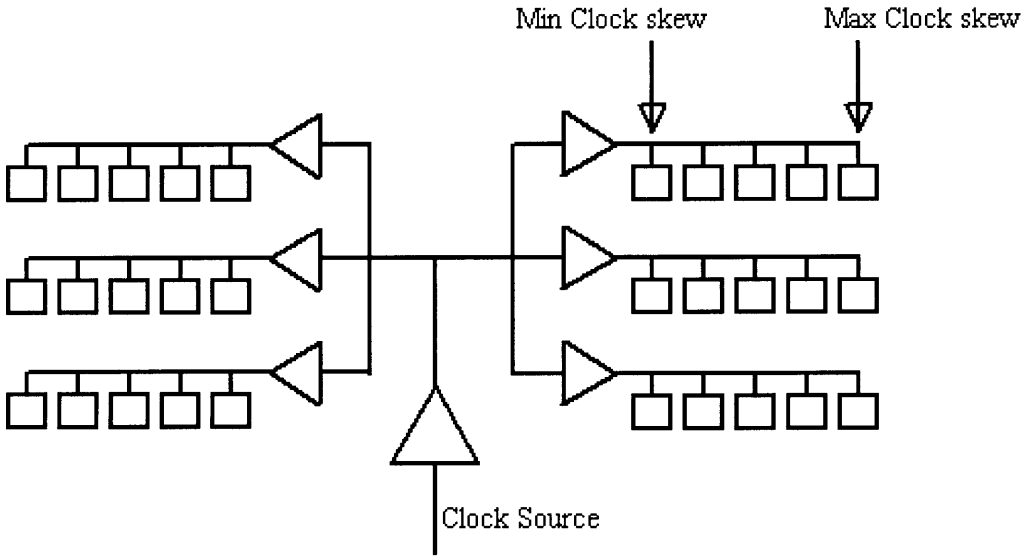

Figure 22 : Hierarchical clock-buffering scheme

Where clock skew is to be kept as close to zero as possible another approach, the H-tree is used. The H-Tree approximates zero skew by using a single clock source that feeds clock sinks that are connected with the same length of interconnect wire. This approach shown in Figure 23, has the disadvantage of extra interconnect wiring, capacitive load and hence longer RC delays.

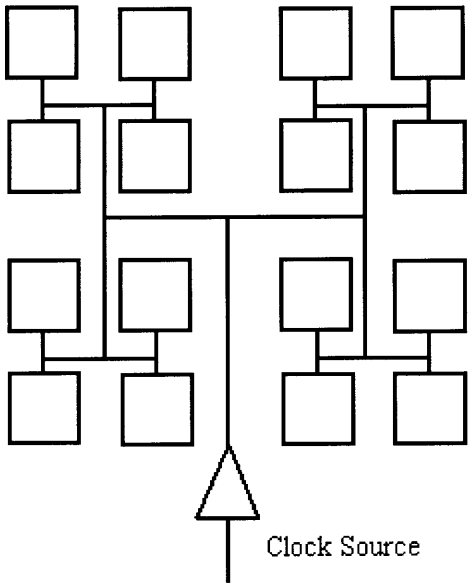

Figure 23 : H-Tree Clock Network

To minimize the clock skew on the test vehicle and reduce simulation time to qualify the clock distribution network, a modified H-Tree network was used. The H-Tree was used to supply the clock to each Logic Test Cluster (LTC), but within the LTC a hierarchical clock network was used without additional buffers. The H-Tree guarantees zero clock skew to the LTV's and within them the RC delays is small enough to meet the clock skew budget of 30ps max. Figure 24 shows a model of the clock distribution network employed in the test vehicle.

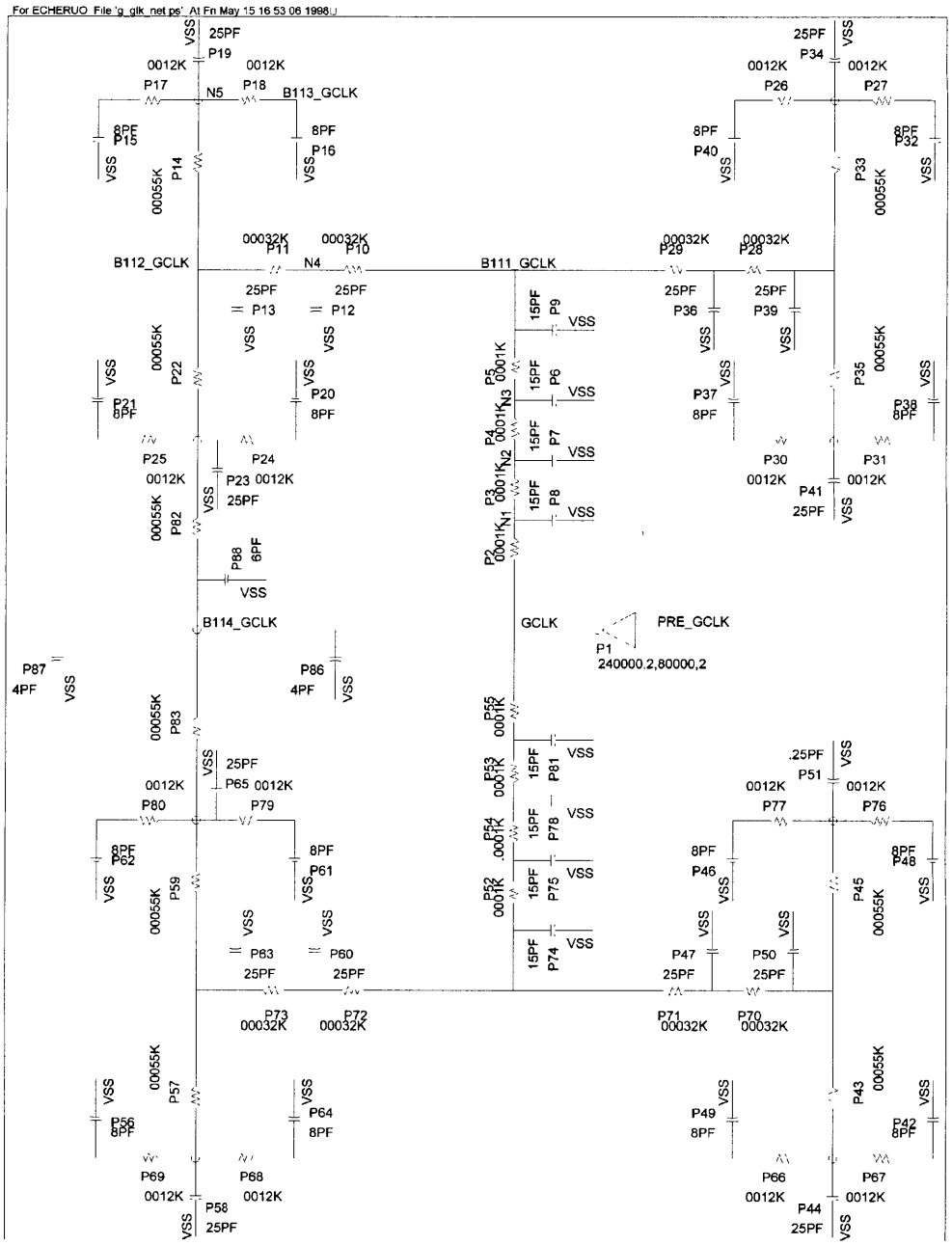

Figure 24 : Global Clock distribution network model

Simulation of this model using spice was used to show that the network met the limit of 30ps clock skew. Simulation data also showed that the rise and fall times of the clock network were good enough for the test vehicle. Figure 25 shows several timing traces from spice simulations of the clock network.

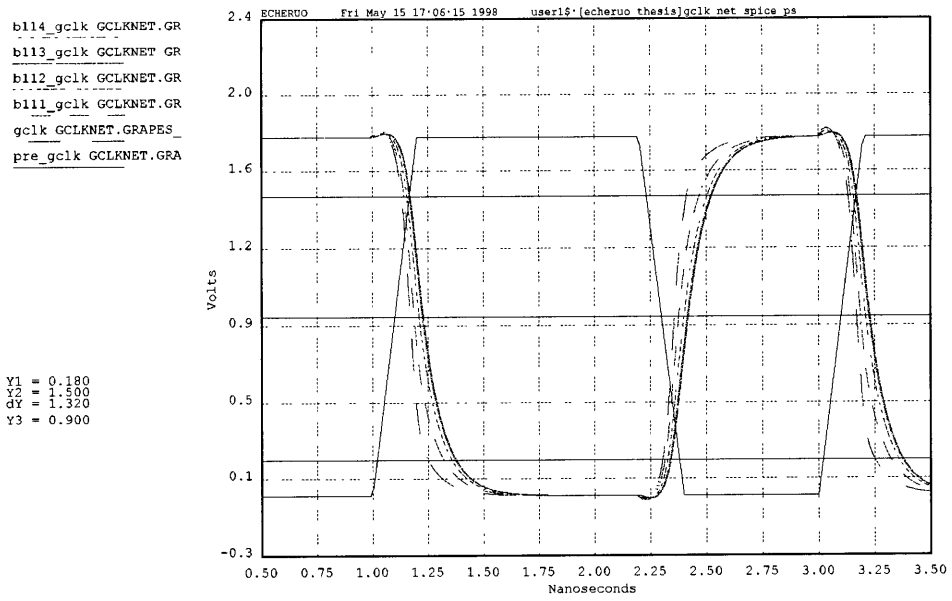

Figure 25 : Traces from Spice Simulation of Clock Network.

For the Cluster itself another model was created, shown in Figure 26, it is a basic hierarchical clock model without clock buffers. The global clock distribution network connects to the center of the cluster clock distribution network and is distributed in a grid like fashion to all the LTE blocks contained in the cluster. To supply the central control logic with a clock source, taps are made from the second and third cluster quadrants as can be seen in Figure 24. These taps also transverse the same interconnect length from the clock generators as do the clusters to guarantee zero clock skew between the central control logic and the logic test clusters.

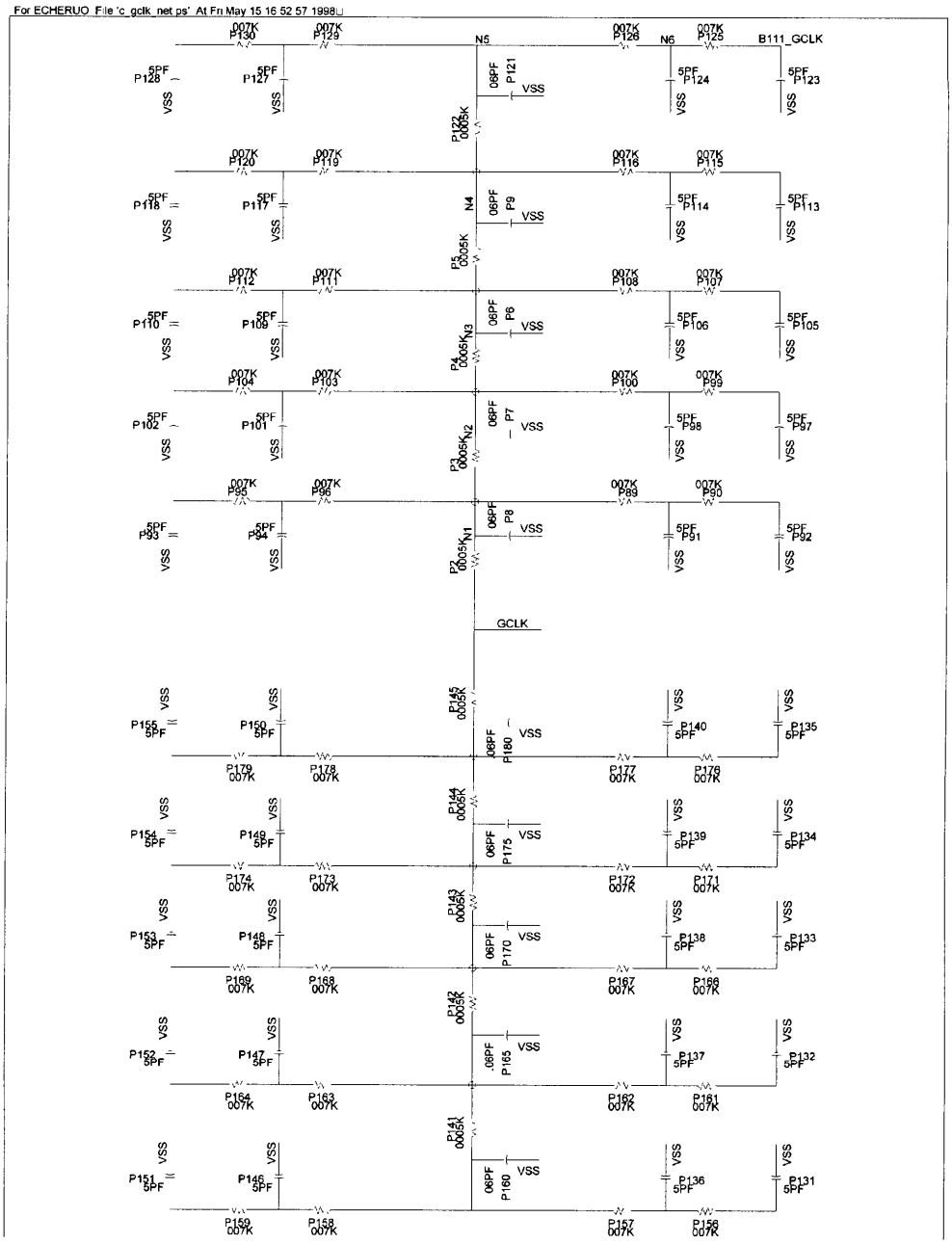

Figure 26 :Cluster Clock Distribution Network

5.1.5 Clock Domain Arbiter and Distribution Network

The LTV uses several different clock sources, an external clock source (tck) as well as several internal taps from a ring oscillator. In order to allow selection of any of the clock sources while guaranteeing a glitchless global clock it is necessary to incorporate a deglitching circuit into the

clock selection mux (clock domain arbiter). The deglitching circuit acts as the clock arbiter during transitions between clock domains. An implementation of this clock arbiter circuit is shown in Figure 27. It succeeds in deglitching the global clock by holding it low (and preventing state change) until all glitches that may be present in the clock generation circuits are resolved.

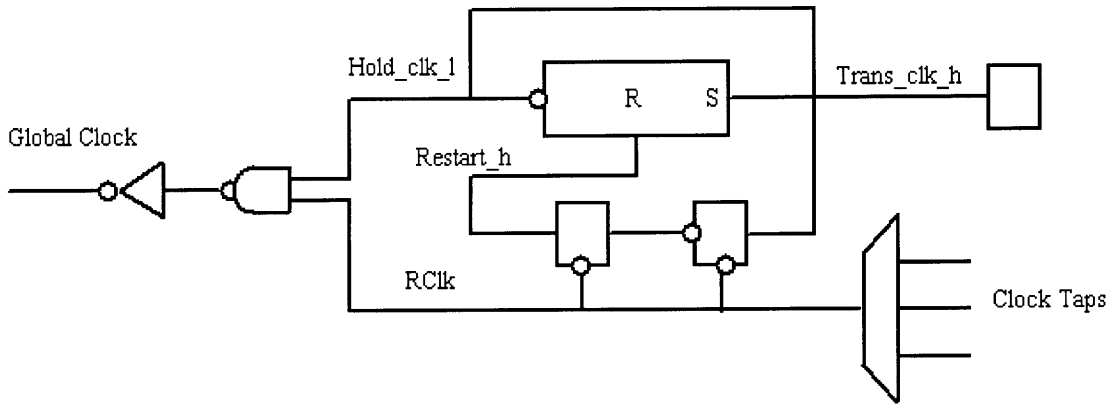

Figure 27:Clock Arbiter and Deglitching Circuit

The signal Trans_clk_h is generated to initiate transitions between clock domains during the A phase of a transition. Trans_clk_h sets the asynchronous RSR latch causing hold_clk_l to be asserted. Hold_clk_l is and'ed with rclk to produce the global clock gclk. By doing this, the global clock is held low during the transition period and the LTV state is preserved and glitches removed from the global clock.

Hold_clk_l is also used to feed a set of synchronization latches (L1 and L2) clocked by rclk. When Trans_clk_l is asserted, a different clock tap was selected through the clock source mux. The clock source mux feeds rclk making it inherently glitch prone during transitions between different clock taps. rclk is used as the clock to the synchronizing latches L1 and L2 which feed the reset input of the RSR latch. (more latches can be used to reduce the probability of metastablity, resulting from latching asynchronous signals, from reaching the RSR latch). The

purpose of the latches is to synchronize the falling edge of the new rclk with the de-assertion of Hold_clk_l. By doing so we guarantee that gclk is held low throughout the phase uncertainty and glitches that may occur during clock transition. Instead of holding the clock low the circuit can be modified the hold the clock high instead depending on the timing requirements of the driving circuits. A timing diagram to illustrate the operation of the clock arbiter circuit is shown in Figure 28. The phase misalignment of the clock domains is compensated for and glitches during clock transitions are eliminated from the global clock.

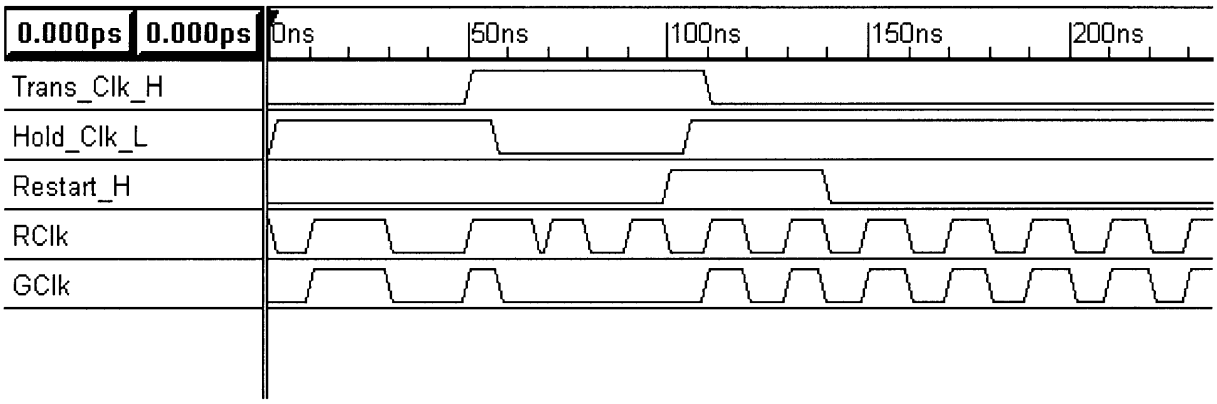

Figure 28:Timing Diagram

The operation of the clock arbiter circuit is transparent to the rest of the LTV. The only additional signals needed are those for signaling a transition between clock domains, Trans_clk_h, and the selecting the clock tap to be used. Generation of Trans_clk_H, is done in the control logic by decoding the current instruction or at the initiation of a RUN or CMD mode. The CMD mode is entered into when a command from the external pins is needed such as loading registers and reading data from the LTV clusters. The RUN mode is entered into when internal generation of data stimulus and address is desired. Depending on which run mode is entered into a failing vector may return the LTV to CMD mode. Figure 29 shows the state machine for the generation of Trans_clk_h . For timing reasons this signal is latched until the next cycle before it is passed

to the clock arbiter. All data latches on the LTV are operated on gclk. This simplifies clock routing and lowers clock skew.

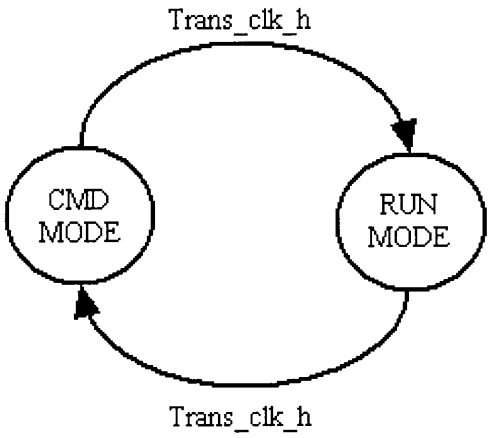

Figure 29:Trans_clk_h generation

Design concerns in implementing this scheme of clock synchronization are as follows:

1) Clock to Q + RSR prop delay must be shorter than a phase of gclk

2) Clock to Trans_clk_ah + RSR prop delay must be shorter than phase of clk

3) Probability of Metastability must be lowered by addition of enough synchronizing latches.

5.1.6 Decoupling capacitance

Due to the potenitally high operating frequency of the LTV and high current demands of clock drivers and LTC's it is necessary to add a considerable amount of decoupling capacitance to the test vehicle. The decoupling capacitor is implemented as an NMOS device with its source and drain connections tied to VSS and its gate tied to VDD, this is shown in Figure 30.

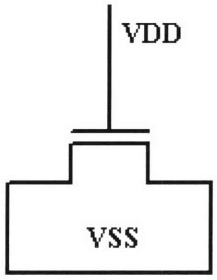

Figure 30: Decoupling Capacitor

Decoupling capacitors are placed around high current devices and each LTC to supply current during pathological switching scenarios. Figure 31 highlights the area taken up by decoupling capacitors on the LTV die.

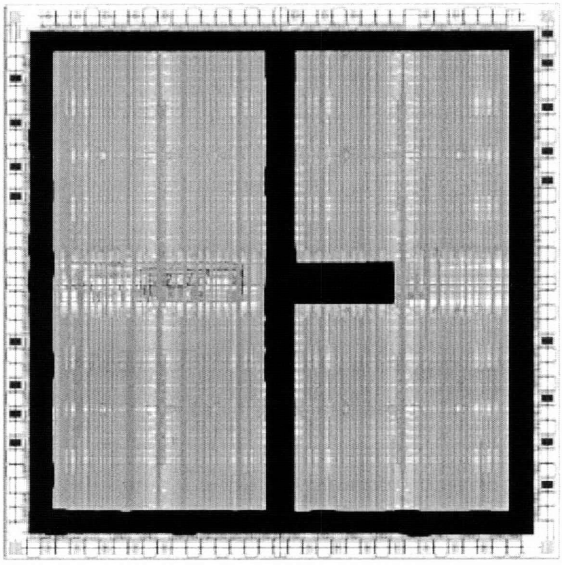

Figure 31 : Decoupling Capacitors on the LTV

5.1.7 Pad Ring and IDDQ

The pad ring is located at the perimeter of the LTV die and contains the I/O drivers and power and ground pins to control the LTV. To facilitate better IDDQ testing, a method of testing VLSI chips by monitoring the current consumption of the device, the power grid is segmented into several partitions. A separate power grid feeds each quadrant of the LTV, comprising of four LTC blocks. Also an additional power grid feeds the central control logic and clock drivers.

IDDQ testing is accomplished by measuring the quiescent leakage current of the LTV caused by gate-oxide shorts and other sources [5]. The usefulness of IDDQ testing has been decreasing due to the increasing number of devices being built into each VLSI device, because it makes discrimination of the individual faulty gate harder. The increasing leakage current produced by transistors in each new process generation also compounds this problem by masking faulty gates due to the high background leakage current. By segregating the power sources to each cluster of the LTV, IDDQ testing of the LTV has been enhanced by a factor of more than 4.

5.2 Power Analysis

Power consumption of the LTV can be determined by analysis of the capacitive load of the various components of the design and from the equation:

$$P = \tfrac{1}{2}(CV^2)*f*U \qquad (5.1)$$

Where C=Capacitance, V=voltage, f=Frequency, U=Utilization

Assuming each node in the LTV is an XOR function, to compensate for glitching and other spurious transitions, the utilization of the LTV can be approximated by an analysis of the switching probability of an XOR gate [11].

Table 3 : 2 Input XOR Gate

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

Assuming that the initial probabilities:

$$P(A=1) = ½, P(B=1) = ½$$

Therefore:

$$P(Out = 1) = ½, P(Out = 0) = ½$$

$$P(Out, 0->1) = P(Out=0).P(Out=1) = ½ * ½ = ¼$$

$$P(Out, 1->0) = P(Out=1).P(Out=0) = ½ * ½ = ¼$$

The probability of a transition, or the utility, in the output is:

$$U = P(Out, 0->1) + P(Out, 1->0) = ¼ + ¼ = ½$$

Next the components of the LTV must be broken down and converted into a capacitive load. Due to the unavailability of capacitance data on features in the target .25u process, the power estimation is done using capacitance data for a .35u process. Scaling the result by a factor of 0.5 should approximate the actual power usage in the fabricated test chip.

The total area for MOSFETs in each logic test cluster is listed in Table 4.

Table 4 :Cluster Diffusion Area

Name	Area (CDU)
N Diffusion	1432236
P Diffusion	2112080
Total	3544316

Using a conversion factor of 0.4ff of capacitance for each CDU (CMOS design unit) of area, we arrive at a capacitive estimate of the TLC of 1.42e6 ff. In addition to the diffusion capacitance local interconnect capacitance must also be added. A general rule of thumb for calculating

interconnect capacitance is to assume a 50% split between diffusion/gate capacitance and interconnect. This means the diffusion capacitance has to be multiplied by 2 to determine the total capacitance for each LTC.

In addition to the local interconnect in each LTC there is additional capacitance from global routing interconnect feeding each LTC. Using approximations for the capacitance of interconnect in the process, the total global interconnect attributed to each LTC was found to be 2e4ff.

Another approximation used in approximating power usage is to assume that the control logic's contribution to power dissipation is insignificant, since it is hardly used and comprises a very small percentage of the total test chip. Since there are 16 TLC on the test chip we multiply the capacitive load attributed to each LTC by 16 and arrive at a total load of 1.44e6 ff.

Lastly the capacitance of the clock network, 6e5 ff, extracted using cad tools must be added to arrive at the total capacitance of the chip. Table 5 lists the different components of the test chip and their capacitive loads.

Table 5 :LTV capacitance

Name	Unit Capacitance (ff)	Quantity	Total Capacitance (ff)
TLC	1.44e6	16	2.3e7
Clock Network	6e5	1	6e5
Total			2.36e7

Plugging in the total capacitance of the LTV and using a target frequency of 300Mhz, a supply voltage of 2.5Vand a utilization of ½, the total power dissipated of the LTV is 11W in a .35u process. Using an optimistic scaling factor of a half for the .28u process, the actual dissipation of the LTV should be 5.5W.

In order to supply the needed power to the LTV, a grid network of power supply busses was implemented. The grid is fed by wide power and ground rings, which in turn are fed by multiple power and ground pads. Figure 32 shows an example of the power and ground grid.

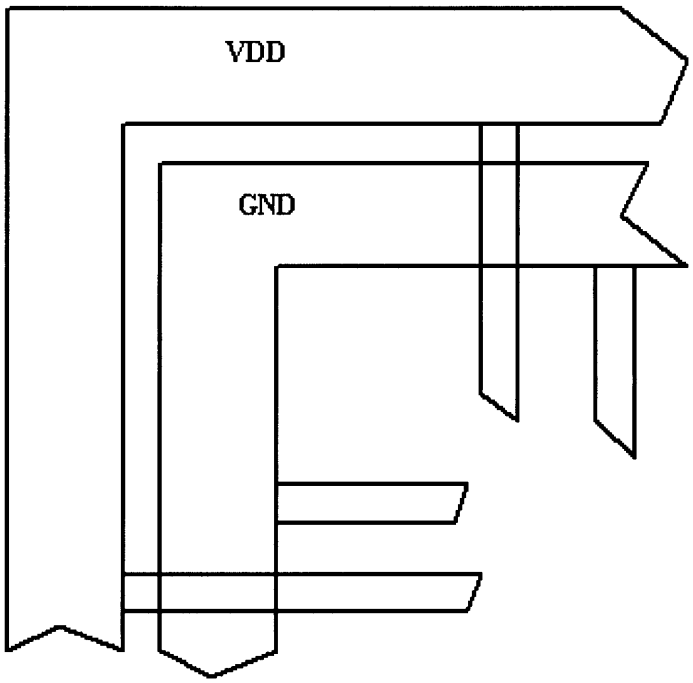

Figure 32: Power and Ground Grid

By doing this the power needs of the LTV was spread over a vast number of pins contained in the 132 Pin PGA, reducing inductive power drops across each of the power pins.

Chapter 6

Conclusion

Designing test vehicles for future CMOS process generations is a continuously evolving process, the work included here tries to pose a methodology for improving the value of the test vehicle, by making them more accurately model the actual products made with the process. The design of the test vehicle is a reasonably complex, but is still diagnosable using some of the same methodologies used in testing SRAMs. The architecture used in implementing the test vehicle allows the inclusion of multiple implementations of the basic LTE cell, without changes to any of the support logic. Because of this modular approach to designing the test logic, the test chip has a better chance of keeping up with advances in IC integration and circuit implementations employed in modern ICs. An important feature of the design is that the diagnosability of the test vehicle remains constant with transistor count, allowing the design to be replicated, as process dimensions shrink, without hindering effective testing and diagnosability. Finally by keeping the number of unique structures on the chip to the essentials, the design can be easily portable to the next process generation.

There are several challenges that were not addressed in the design of the LTV. These include full testability of the support circuitry and full non-uniqueness testing of the LTE array. While these shortcomings are needed to completely perform process testing, they are fairly straightforward to address in future generations of the LTV. The Design resources for creating new generations of the LTV are minimal and will mostly be relegated to layout modifications for each new CMOS process.

Due to the latency in getting silicon back from fabrication, testing of the LTV to verify its operation at the indicated speed and measure improvements in IDDQ testing could not be done. The behavioral model was thus used to verify the operation of the LTV.

Appendix A: Specifications

Logic Test Vehicle Chip

Generation 1

Specifications [8]

Contents

Introduction

The Logic Test Vehicle chip (henceforth referred to as the LTV chip) is a new process bring-up and process/product qualification tool. Similar in scope and character to the SRAM test chips presently used for process bring up, LTV is expected to overcome some of the short-comings of the SRAM test vehicle. The test circuitry on the LTV chip is more complex and therefore a bit more representative of the complexities found on the real products. The LTV is not intended to replace the use of SRAM test vehicles, but rather supplement it. Also at present in its current scope of definition, the LTV is not expected to represent all aspects of circuit complexities found on a product and therefore is not expected to replace or eliminate the product qualification effort. With the insight and experience gathered over time, the LTV is expected to open new avenues to process and product qualification and allow us to get higher quality products quicker to the market.

The LTV chip is designed to satisfy the following goals and constraints.

- Representative of a broad range of circuits and circuit topologies.
- Diagnosable to the smallest possible cluster of gates by non-destructive testing methods.
- Capable of thorough self-exercising and self-testing at a representative gate transition speeds and gate delays during life test.
- Stress all interconnect levels.
- Life testable in the existing Wakefield equipment.
- Testable on the same tester that is used for testing SRAM test vehicles.
- Portable to process generations with minimal redesign effort.

Some goals not addressed by this version of chip are:

- Smart monitoring. This goal is deferred to a later version of the chip. The LTV architecture, however, is such that it easily supports this goal.

Table 6 :LTV Chip Summary

Transistors	700,000 approx.	Package	132 Pin PGA
Die-Size	34,000x34,000 cdu	Speed	300Mhz
Signal Pins	37	Power	5.5W

Chip Architecture

The LTV chip is a two dimensional arrangement of the basic building block called Logic Test Element (LTE) and the test and diagnosis support logic. The former, supplies the bulk of the process test transistor and interconnect geometry implemented in the test vehicle, the latter provides an effective self-exercise and self-test during life testing as well as supports convenient off-line test and diagnosis from a simple tester.

Figure 33 shows the top-level block diagram of the LTV. It consists of an array of Logic Test Clusters, LTV Control Logic, Clock Generator, Error Monitoring Logic, and Pin Interface Logic.

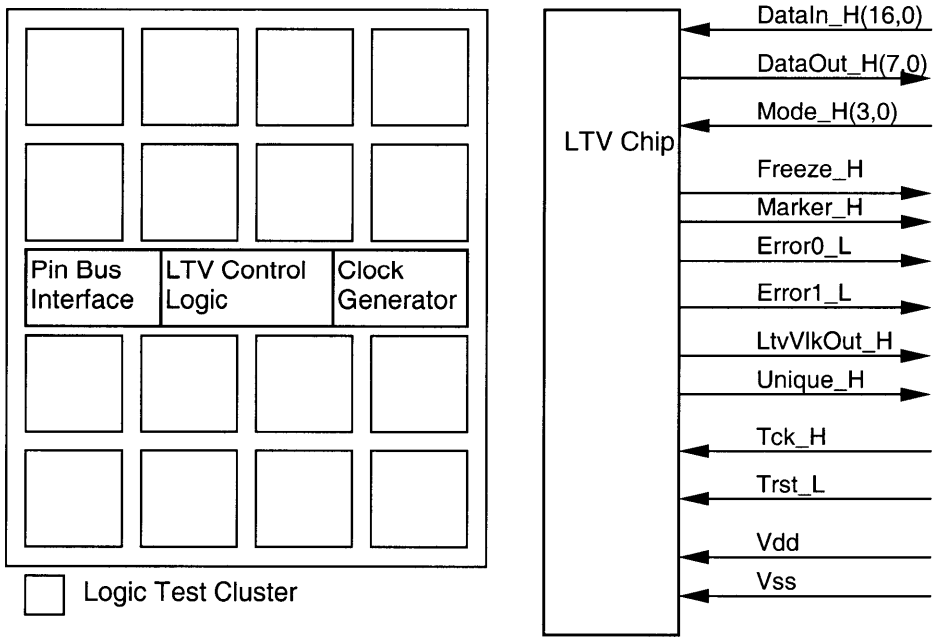

Figure 33 : Logic Test Vehicle Chip

A Logic Test Cluster (LT cluster) is a grouping 64 identical LTEs, the target test logic. Each LT cluster is capable of supporting self-exercise and self-test of its LTEs. Each LT Cluster and each Logic test Element can be accessed from the chip pins via an addressing scheme, similar to the one employed for accessing cells in a RAM array.

The LTV Control Logic supports the self-test operations. It generates and broadcasts address and control to the LT clusters. It houses a master counter which keeps track of the self-test and self-exercise of the clusters.

The Clock generator is a ring oscillator with outputs taken from several taps. It provides clock for the LTV circuits during the life test.

The Error Monitoring Logic performs simple on-chip error gathering and analysis task that allows LTV to detect and flag occurrence of intermittent and non-unique failures during the life test.

The Pin Interface Logic provides a convenient interface to the tester and the life test burn-in equipment to control modes and operations of the LTV chip.

A Logic Test cluster is designed with a specific design style. Using LTE clusters designed with representative product design styles easily increases product representation of this test vehicle. The LTE chosen for this LTV chip is a 4-bit slice of ALU with a set of output latches. The LTE has approximately 700 transistors. A Logic Test Cluster with 64 LTEs contains approximately 45K transistors. A 4x4 arrangement of LT Clusters shown in Figure 33 gives approximately 700K test transistors in 1024 LTEs. The actual number and the arrangement of the LT clusters can be chosen to suit the constraints of the host chip.

Figure 33 also shows the pin interface. Pin Interface is described in detail in Section 0. During life test modes, the only signals to be supported on the burn-in tray are the Tck_H, Trst_L, and Error0_L, Error1_l signals. The rest of the signals can be suitably tied off. Cmd_H(3,0)(3,0) pins are tied off to enable RunForLife mode. In this mode, upon deasserting Trst_L, the LTV enters an eternal self-exercise/self-test mode and remains in that mode until Trst_L is asserted again or

power is turned off. When the self-test circuitry detects an error, the Error Monitoring Logic asserts chips error outputs as explained later.

Logic Test Cluster

Logic Test Cluster houses the bulk of the test transistors and the local self-test support logic. Figure 34 show the block diagram of the LT cluster. It consists of an array arrangement of 64 Logic Test Elements, a pair of Stimulus Generators, a pair of row decoders and a common Column Mux and Cluster Compare Logic.

The LTE array is organized as two symmetrical halves of 4x8 array of LTEs. Each half is fed by a Cluster Generator to exercise the LTEs. During life test, the address dispatch from the Cluster Control Logic randomly selects an LTE in the upper half cluster and pairs it with an LTE in the lower half cluster to dynamically form the basic self-testing unit. The comparator in the Cluster Support Logic compares the outputs of the selected LTE pair and flags an error if a mismatch is detected.

Address_H(4,0) broadcast from the LTV Control Logic selects two LTEs (one from each half of cluster) for self-test, comparing outputs of each against the corresponding outputs of the other. Address_H(5) selects outputs from one two LTEs selected for self-test and makes them available at the cluster output. When the cluster is selected during off-line test, these outputs are made available to the chip pins via the global output lines.

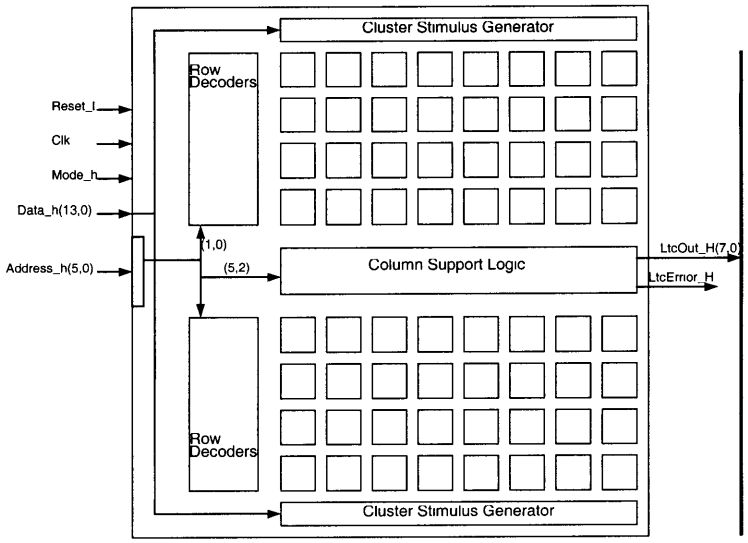

Figure 34 :Logic Test Cluster

Logic Test Element

The Logic Test Element chosen for this LTV chip is a 4-bit slice of ALU with a set of output latches. The 4-bit ALU slice is simple yet a few orders of magnitude more complex than an SRAM cell. Its testability is well understood, requiring only 13 vectors to detect all single stuck-at faults. Ability to test it with exhaustive set of inputs is expected to provide a rich fault dictionary for non-destructive fault isolation and failure analysis. Output latches provide the opportunity to bring in a variety of latch representations. The LTE has approximately 700 transistors.

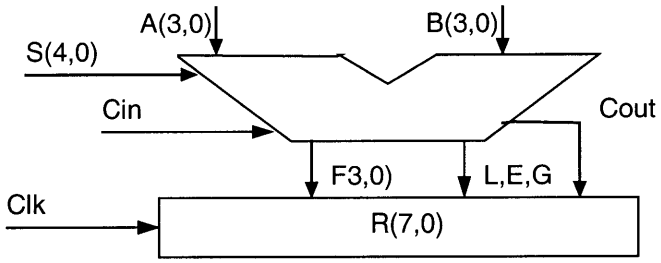

Figure 35 :The Logic Test Element

Cluster Stimulus Generator

Cluster Stimulus Generator is a 17-bit LFSR capable of generating a stream of 2^{17}-1 pseudorandom vectors. A set of 14 outputs (bits 13,0) from the generator are used as inputs to drive the LTEs (Figure 36). The stimulus generator thus not only supplies all input combinations to the LTE, but also provides eight opportunities for varying vector to vector transitions. Bits (13,9) are used to drive the function select lines of the LTEs. Bits (8,0) supply the A, B arguments and Cin. The connections of bits (8,0) to the argument inputs of LTEs are rotated by two bits from

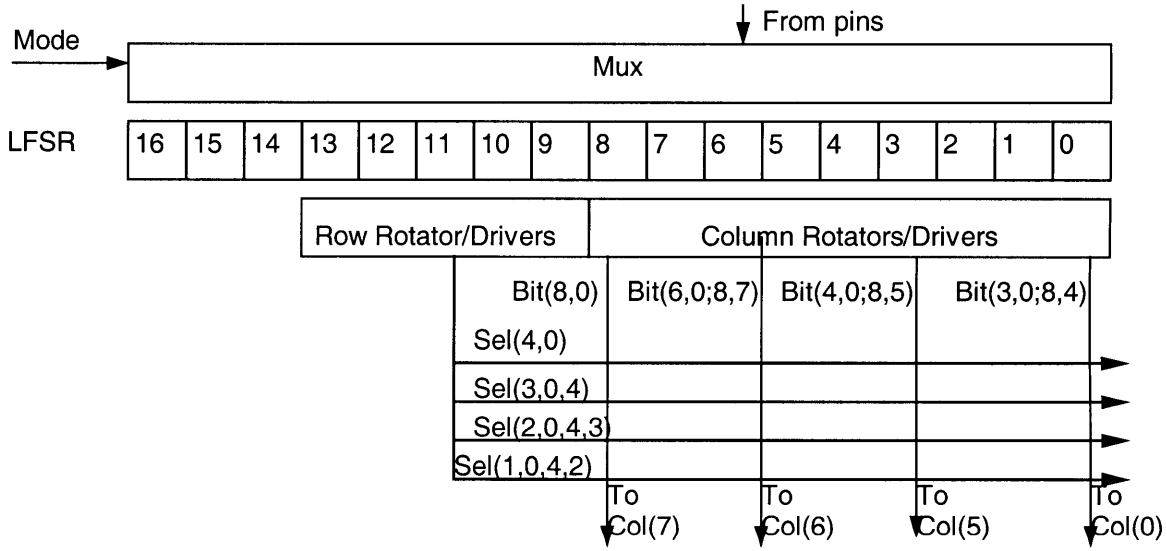

Figure 36 :Cluster Stimulus Generator

column to column. Likewise, connections of bits (13,9) are rotated by one bit from row to row. This staggers the nodal activity in the LTEs in a cluster as well as helps to establish a unique identity of an LTE within a cluster. Similarly, the nodal activity from cluster to cluster is staggered by offsetting cluster generator to a different initial value.

The generator is clocked by gclk. An assertion on Trst_L initializes the generator with 0x1. The Address_H(xxx) bits address the generator for writing from pins. When its address is selected,

LdCsg command loads the generator from Data_H pins. The generator counts every cycle for which run_H is asserted.

Cluster Support Logic

This logic selects a pair of LTEs, one form each half of the cluster and compares their outputs to generate the error signal during the life test modes. During the off-line test modes it transfers the outputs of the selected LTE to the global output bus. Figure 37 shows the block diagram of the Cluster Support Logic. It consists of a pair of column multiplexers, a comparator and the output drivers.

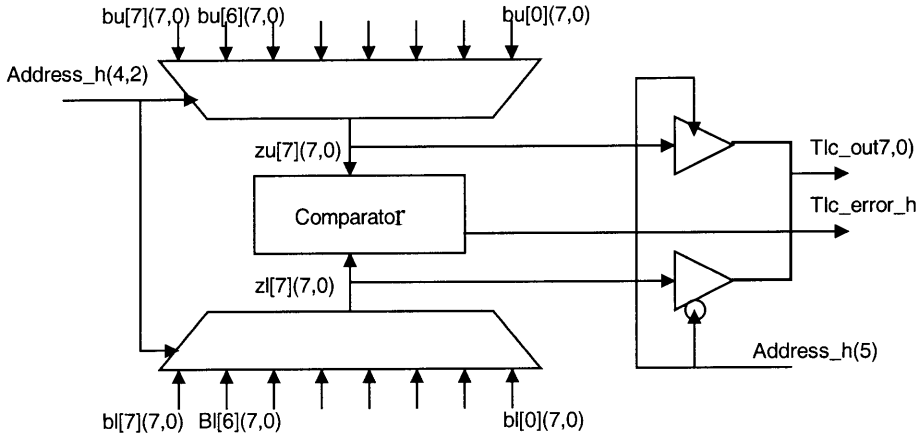

Figure 37: Cluster Support Logic

LTV Control Logic

LTV Control Logic broadcasts LTE addresses and control to the LT clusters. During the life test modes, it generates the addresses, while during the off-line test modes it passes the addresses received from the chip pins.

Figure 38 shows a block diagram of the LTV Control Logic. It consist of Cluster Address Register, LTE Address Generator, Self-Test Duration Counter, Freeze Register, Activity Counter, Clock Control Register, and Test Control Register and the control logic.

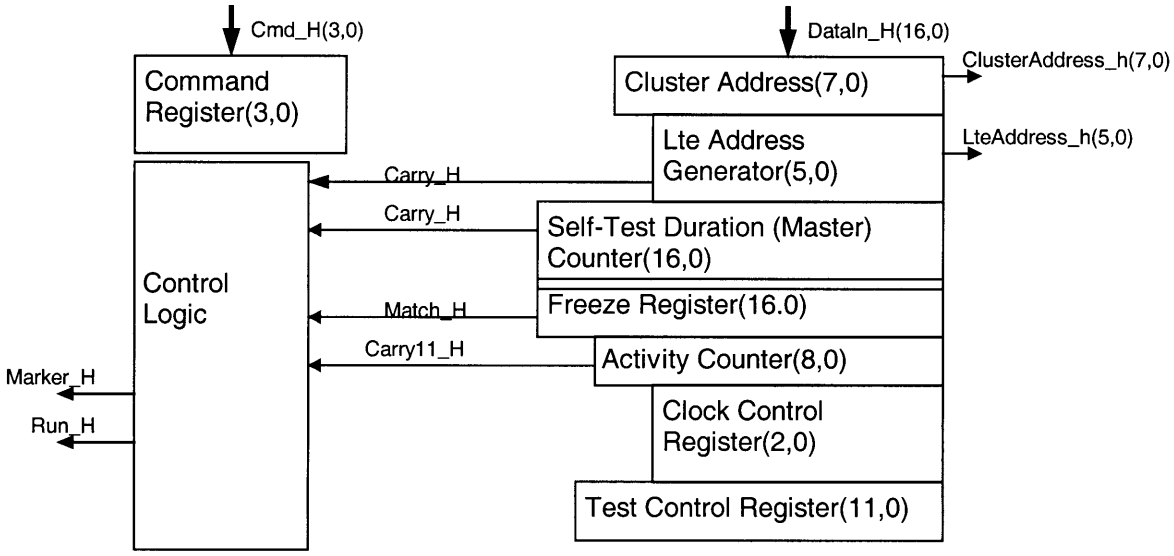

Figure 38: Cluster Address Generator

Cluster Address Register

Cluster Address Register is 8-bit wide. Its outputs are decoded and used for enabling one of the 16 clusters to connect to the DataOut_H pins. Trst_L clears the register. LoadLteAddress command loads it from the DataIn_H pins (see Table 9). The counter can be observed on DataOut_H(7,0) pins by suitably loading the Test Control Register (see Table 8). (Note: The present generation of LTV has only 16 clusters. Therefore, only bits (3,0) are used. The other bits are ignored.)

Lte Address Generator

Lte Address Generator is a 6-bit LFSR Counter, counting all 2^6 states. Its outputs uniquely select one of the 64 LTEs on each cluster. Its carry output indicates the end of major self-test cycle. That is, completion of minor self-test cycles (defined later) on each of the 64 LTEs in a cluster. The Trst_L resets the counter. It is clocked by the global clock gclk. It counts every time the Self-Test Duration Counter puts out a carry. The counter can be loaded directly from the DataIn_H

pins by a load command (see Table 9). The counter can be observed on DataOut_H(7,0) pins by suitably setting the Test Control Register (see Table 8).

Self-test Duration Counter

Self-test Duration Counter is 17-bit wide (same size as the stimulus generators in the clusters). The Trst_L initializes the counter to a starting value. The counter is clocked by the global clock gclk. It counts every cycle for which run_H is asserted, that is the LTV is not in a freeze state. When the count reaches 2^{17}- 1, the counter outputs a carry to indicate the end of the minor self-test cycle on an LTE. The counter can be loaded directly from the DataIn_H pins by a load command (see Table 9). The counter can be observed on DataOut_H(7,0) pins by suitably loading the Test Command Register (see Table 8).

Freeze Register

Freeze Register is a 17-bit wider register used for halting the self-test and freezing the state for debug. During RunFreezeOnCount mode the LTV state freezes when the Freeze Register contents match the contents of the Self-Test Duration Counter. Trst_L clears The Freeze Register. LoadFreezeRegister command loads the register from the DataIn_H pins.

Activity Counter Register

Activity Counter Register is a 9-bit LFSR counter. The counter is used to apply a burst of 2^9 - 1 cycles worth of self-exercise and then hold the LTV in a temporary freeze state until the restart cue (Tck_H edge) is received. An assertion on Trst_L or an issue of a new run command resets the counter to the starting count.

Clock Control Register

Clock Control Register is a 3-bit register. It selects one of 8 possible clocks to run the LTV logic. The register is loaded from DataIn_H pins up on deassertion of Trst_L. Table shows the clock selection.

Table 7: Clock Control Register and LTV Clock Selection

CCR(2,0)	ClockTap	CCR(2,0)	ClockTap	CCR(2,0)	ClockTap	CCR(2,0)	ClockTap
000	500Mhz	010	Tbd	100	Tbd	110	tbd
001	Tbd	011	Tbd	101	Tbd	111	Tck_H

Test Control Register

The Test Control Register is 12 bit register used for miscellaneous control of the operation. A portion of the register provides observability control by selecting and steering information from various sources in the LTV chip to the output pins. Table 8 shows the bit fields and their effect on the outputs. Trst_L clears this register to and establishes the default output control. LoadTcrReg mode loads the register from the pins (see Table 9).

Table 8 :Test Control Register

Field	Tcr(11.0)	Value	Output
ErrorControl(0)	0	0 1	Bist-Cycle-End Error Cycle-by-Cycle Error
ErrorControl(1)	1	0 1	Or of all cluster error outputs Selected cluster error output
ObsSel(2,0)	4:2	000 001 010 011 1xx	Selected Lte outputs Address Generator Master Counter Fuse ID For future use
ActivityFlag(0)	5	0 1	Enable activity equalization Disable activity equalization
Spares	11:6		TBD

Test Command Register

Test Command Register is a 4-bit register that provides control over the various test and diagnosis operations of the LTV chop. The register is directly loaded from the pins when the LTV is in Command Mode (described later). The register is cleared by Trst_L. Table 9 lists the opcodes for the various commands together with the function of the other input pins.

Table 9 :Command Register Opcodes and Input Control

Cmd_H(3,0)				Command	DataIn_H(16,0)															
					16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	RunForLife															OutFSel<2:0>		
1	1	1	0	RunFrzOnCount															OutFSel<2:0>		
1	1	0	1	RunFrzOnError															OutFSel<2:0>		
1	1	0	0	RunSingleBurst															OutFSel<2:0>		
0	1	1	1	LoadCltgen	ClusterGenerator<16:0>																
1	0	0	0	LoadMasterCntr	MasterCounter(16,0)																
0	1	1	0	LoadFreezeReg	FrrezeRegister(16,0)																
0	1	0	1	LoadLteAdress				ClusterAddressRegister(7,0)								LteAddressGenerator(5,0)					
0	1	0	0	LoadTcr						TestControlRegister(11,0)											
0	0	0	0	NOP															OutFSel<2:0>		

As seen from the table, LTV supports two groups of commands; various Load commands that establish the internal state of LTV control registers, and the various Run commands that put LTV into a Self-Test/self exercise mode. Self-test/self-exercise modes begin from the default state established by the power-on reset (assertion of Trst_L) or from the control state established by the various load commands.

RunForLife: This command puts LTV into the eternal Run state. It is use for life test. The Clusters undergo self-exercise/self-test automatically and continuously until interrupted by the reset or the loss of power.

RunFrzOnCount: This command is used to advance the LTV state to the desired self-test cycle during debug and failure analysis. The command puts LTV in the Run state. It initiates the self-test/self-exercise from the established LTV state and returns the LTV to Freeze (Command) state when the Master Counter reaches the value in Freeze Register. Due to the pipeline latencies, the LTV state advances by two cycles after the freeze is issued.

RunFrzOnError: This command is used to stop on an error during debug and failure analysis. The command puts LTV in the Run state from the established LTV state and returns the LTV to Freeze (Command) state when an error in Lte is detected. Due to the pipeline latencies, the LTV state advances by two cycles after the error is detected. During this command, the value in the Freeze Register establishes the self-test cycle (Master Counter value) up to which the errors must be masked, thus preventing them from causing a freeze.

RunSingleBurst: This command is used to apply a single burst consisting of two cycles to check out the dynamic performance of the selected LTE during debug and failure analysis. The command puts LTV in the Run state from the established LTV state and returns the LTV to Freeze (Command) state after executing exactly two cycles of self-test.

LoadCltGen: This command is used to test a selected LTE directly from the primary I/P pins. The command loads the cluster generator selected by the Cluster Address Register(7,0) and the LteAddress Register(6) directly from DataIn_H(16,0) pins.

LoadMasterCounter: This command loads the Master Counter directly from the DataIn_H(16,0) pins. The command is primarily useful for testing out the Master Counter itself during debug and failure analysis.

LoadLteAddress: This command loads the Cluster Address Register and the Lte Address Counter directly from the DataIn_H(13,0) pins. The command is used to select a specific LTE in the LTV for testing, debug and diagnosis.

LoadLteAddress: This command loads the Test Control Register directly from the DataIn_H(11,0) pins.

LTV State Machine

LTV chip has basically two states: Run state and the Command-Freeze state.

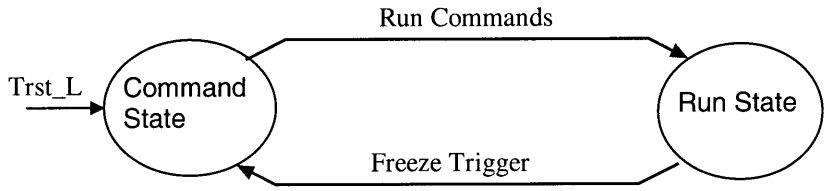

Figure 39 : LTV State Machine

Command State:

This is the state in which LTV can be issued various run and load commands from the Cmd_H(3,0) pins. The state is forced upon power-up or by assertion of Trst_L. The state is also entered automatically when a freeze is triggered during any of the run commands. The LTV operates with the external Tck_H clock during this state. The self-test activity is frozen. Only operations allowed are the loading of the various control registers by the load commands. The LTV remains in this state until a run command is issued.

Run State:

The LTV enters this state from the Command state when any of the Run commands is issued. The LTV remains in this state until a trigger action, or reset forces it to return to the Command state. In this state, LTV executes the self-test/self-exercise cycles with a clock selected by the Clock Control Register (see Table 7).

Activity Stress Control

Chip-to-chip variations in the clock frequency supplied by the LTV's on-chip ring oscillator can subject LTV samples in a life test run to different amounts of nodal activity stress. This can complicate interpretation of the life test results. The LTV chip architecture therefore has a provision to equalize the nodal activity stress on different LTVs in a given wall clock interval.

During Run state, the activity counter outs out a carry after every 512 cycles. This causes the LTV to enter a temporary freeze state and remains there until the next rising edge of Tck_H. At that time it exits the temporary freeze state and the self-test/self-exercise begins from where it was left off.

Thus, all LTVs started on Life Test at the same time will enter this temporary freeze state after 512 cycles. After the slowest known part has entered the freeze state, the burn-in chamber' s clock module issues the rising edge of Tck_H and the all LTV's resume the next burst of self-test/self-exercise.

The scheme requires that the range f the operating speed of all LTVs in a life test run be known. The period of the Tck_H that the clock module must supply is determined by the following relationship.

$$T_{tck} > 1.1 \times 10^9 \times T_{gclk}$$

where, T_{gclk} T_{gcl} is the period of the slowest LTV in the life test.

Clock Generator

Clock Generator Logic provides the clock for the LTV chip as well as provides for the synchronization between the external Tck_H and the clock internally generated by the ring oscillator.

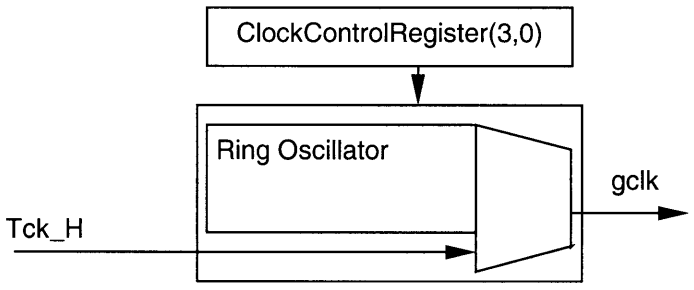

Figure 40: Clock Generator and Activity Control Logic

Figure 40 shows a simplified block diagram of the Clock Generator. Ring Oscillator produces a 50% duty cycle clock with a cycle times T_{clk1} and T_{clk2} ns. T_{clk1} is selected to be approximately **200%** of the worst case path delay on the LTV. T_{clk2} is $1.5xT_{clk2}$. This slack immunizes the proper functioning of LTV from the ring oscillator frequency variations.

Monitoring Logic

For the first generation of LTV chip, the monitoring logic consists of a simple OR output of cluster error signals that sets a set-reset flop. The flop is cleared at the start of the self-test cycle. The error flop in turn updates the Fail Flag, which drives the Error0_L, Error1_l pin as specified earlier.

Pin Bus

DataIn_H(16,0) are the 17 data input pins. They are useful during production test and during debug and diagnosis to load various control registers and to test each LTE directly from pins.

DataOut_H(7,0) are the output pins. They are used during production test and debug and diagnosis. They allow to external logic to examine outputs of a selected LTE or allow observation of the selected internal registers, including fuse die-ID.

Cmd_H(3,0) are Command input pins. They provide the operational control of the LTV. See Section 0 and Table 9 for the listing and description of various commands.

Marker_H outputs a pulse to mark the end of a self-test cycle and the beginning of a new self-test cycle on the LT Clusters. The pulse is minimum fast tap* 512 cycle wide...

tbd ns wide - wide enough for the tester or Burn-in clock module to sample it for performing failure monitoring functions.

Error0_L, Error1_L are the two open drain output pins that indicate the Pass/Fail status. Error1_L is the copy of the Error0_L. This redundancy is provided to support a scheme to easily locate a failing LTV on a burn-in tray.

The error output is provided either, on the fly cycle-by-cycle, or at the end of the self-test major-cycle. The former is used during production and test and debug modes. The latter is used during life test. The self-test cycle end error-reporting works as follows. If an error is detected during a self-test cycle, the signals assert low with the rising edge of the Marker_H pulse and remain high until updated again at the end of the next self-test cycle. Thus if the fault on the LTV chip is permanent, the signals remain asserted until cleared by chip reset. On the other hand, if the fault is intermittent, the signals are marked by periods of assertion and deassertion levels.

Freeze_H output indicates that the LTV is in Freeze state. The signal is asserted a cycles after LTV state machine enters the Freeze state and remains asserted until it reenters the Run state.

LtvClkOut_H bring out the output of the on-chip Clock generator. This output is used for measuring and characterizing the clock used during the life test mode.

Trst_L is the chip reset. It clears all control register and puts LTV in the Command state.

Tck_H is the external clock input to the LTV chip. It drives LTV during Command Modes and also provides the reference edges for the activity equalization scheme explained in Section 0.

Operation

Life Test Operation

The only pins to be supported for life test operation are: Trst_L, Tck_H, DataIn_H(3,0), Cmd_H(3,0), and the three output signals Marker_H, Error0_H and Error1_H. To start life test do the following:

1. Tie off Cmd_H(3,0) to RunForLife command. Tie off DataIn_H(3,0) to select the the desired ring oscillator clock frequency tap. Assert Trst_L.
2. Feed Tck_H with a suitable frequency clock. As explained in Section 0, the period for Tck_H is determined by the following relationship:

$$T_{tck} > 1.1 \times 10^{9} \times T_{gclk}$$

where, T_{gclk} T_{gcl} is the period of the slowest LTV in the life test.

3. Turn on power.
4. Deassert Trst_L. With this the LTV enters the life test mode with the control register defaults established by the reset.
5. Monitor the Marker_H, and the Error0_H, and Error1_H signals as explained in Section 0.

Testing the Selected LTE from Tester

Although the Logic Test Elements in LTV are self-testing during production test and debug and diagnosis it may be desired to test the LTEs directly from a tester. This can be done in a variety of ways. The following procedure shows one way to test a selected LTE from pins. The testing is done with the external clock Tck_H.

1. Set Cmd_H(3,0) to NOP. Set DataIn_H(3,0) to select the external clock. Assert Trst_L.
2. Feed Tck_H with a clock of desired frequency.
3. Turn on power.
4. Deassert Trst_L. This will establish Tck_H as the operating clock for the LTV.
5. Using LoadTcr command, set up the Test Control Register (ObsSelect(2,0) field. see Table 8) to select Lte outputs for observing at the DataOut_H(7,0) pins. The other fields may be ignored.
6. Using the LoadLteAddress command, load the Cluster Address register and the Lte Address Generator with the address of the LTE to be tested.
7. Set the command to LoadClstGen. Supply the test vectors to be applied to LTE at the DataIn_H(16,0) pins. Observe outputs on DataOut_H pins. Figure 41 shows the timing information.
8. Repeat step 6 and 7 to select a new LTV and test it.

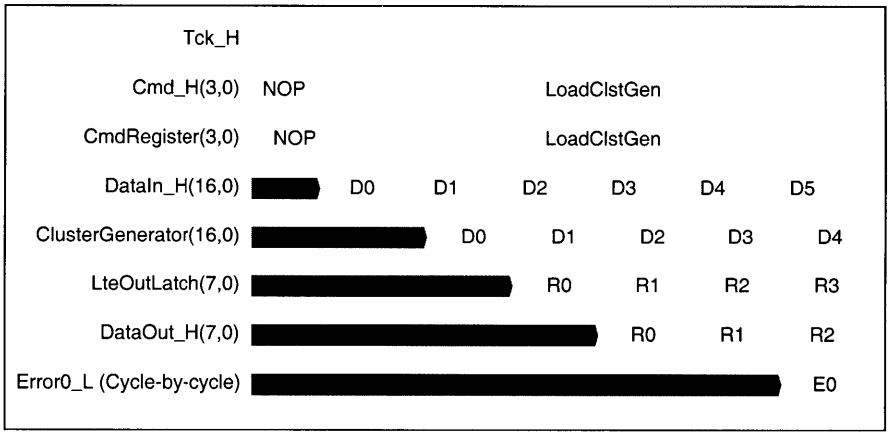

Figure 41 :Timing Diagram for Testing LTV from Pins

Debug and Diagnosis of Speed problems from a Simple Tester

LTV architecture supports debug and diagnosis of speed failures using a simple tester. Suppose that an LTV develops a fault that shows up only during at speed test with one of the clock taps and the tester available for debug cannot support the frequency corresponmding that tap. Some of the features in the LTV architecture allow it to let the LTV self-test at the internal clock rate and isolate/identify failing vectors to aid failure analysis. The following procedure shows how this may be accomplished.

1. Set Cmd_H(3,0) to NOP. Set DataIn_H(3,0) to select the frequency tap for which LTV is failing. Assert Trst_L.
2. Feed Tck_H with a clock of desired frequency.
3. Turn on power.
4. Deassert Trst_L. This will establish the selected clock tap as the operating clock for the LTV.
5. Using LoadLteAddress command, load the Cluster Address register to select one of the 16 clusters. Lte Address Generator may be loaded arbitrarily.
6. Issue RunFrzOnError command. This will begin the self-test with the selected clock tap. If any of the LTEs in the cluster is bad, the LTV enter a freeze state. Freeze_H wire will be

asserted when the freeze occurs. If no errors are found, the LTV will reach the end of the major BIST Cycle and enter the Freeze state. At which point new commands may be loaded.

7. Repeat steps 5 and 6 until the failing cluster is located.
8. Once the failing cluster is located, using the LoadTcr command, set up Test Control Register (ObsSelect(2,0) to select Lte Address Generator to appear on DataOut_H(8,0) pins.
9. Issue NOP command with DataIn_H(3,0) se to 0x0. This will output the bits 7,0 of the Lte Address Register. Repeat this step with DtataIn_H(3,0) set to 0x1 and 0x2 to extract the other fields. When done, the failing LTE's complete address is identified.
10. LoadTcr command, set up Test Control Register (ObsSelect(2,0) to Master Counter to appear on DataOut_H(7,0) pins.
11. Using procedure similar to step 10, off load the contents of the Master Counter. The Master Counter contents are two cycles ahead of the cycle in which the error was detected. This identifies the first failing vector.
12. To identify the subsequent failing vectors, load the Freeze Register with the value corresponding to the count where the last error occurred. Reissue the RunFrzOnError command. This time LTV will enter the Freeze state on the second error. By repeating the procedure, the entire failure syndrome of the fault can be obtained.

Once the failing vectors are identified, it may be desired to examine the actual output responses to the failing vectors. This is accomplished by using the RunFrzOnCount command and following the procedure similar to the one outlined above.

Appendix B: Behavioral Model

Behavioral Model for Logic Test Vechicle (LTV)

Container File (Signals)

Ltv.cnt

```
User {*
#define ACLK  CLK(k->clk)
#define BCLK  CLK(~k->clk)

#define A_CLK k->clk
#define B_CLK ~k->clk

#define And (~((c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](3)&c_lte[cnum]-
>sel_2a_h[arrn][arrx][arry](0))|(c_lte[cnum]->sel_2a_h[arrn][arrx][arry](1)&(~c_lte[cnum]-
>bdata_2a_h[arrn][arrx][arry](3)))|c_lte[cnum]->adata_2a_h[arrn][arrx][arry](3)))
#define Cnd (~((c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](2)&c_lte[cnum]-
>sel_2a_h[arrn][arrx][arry](0))|(c_lte[cnum]->sel_2a_h[arrn][arrx][arry](1)&(~c_lte[cnum]-
>bdata_2a_h[arrn][arrx][arry](2)))|c_lte[cnum]->adata_2a_h[arrn][arrx][arry](2)))
#define End (~((c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](1)&c_lte[cnum]-
>sel_2a_h[arrn][arrx][arry](0))|(c_lte[cnum]->sel_2a_h[arrn][arrx][arry](1)&(~c_lte[cnum]-
>bdata_2a_h[arrn][arrx][arry](1)))|c_lte[cnum]->adata_2a_h[arrn][arrx][arry](1)))
#define Gnd (~((c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](0)&c_lte[cnum]-
>sel_2a_h[arrn][arrx][arry](0))|(c_lte[cnum]->sel_2a_h[arrn][arrx][arry](1)&(~c_lte[cnum]-
>bdata_2a_h[arrn][arrx][arry](0)))|c_lte[cnum]->adata_2a_h[arrn][arrx][arry](0)))

#define Bnd (~((~c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](3))&c_lte[cnum]-
>sel_2a_h[arrn][arrx][arry](2)&c_lte[cnum]->adata_2a_h[arrn][arrx][arry](3)|c_lte[cnum]-
>adata_2a_h[arrn][arrx][arry](3)&c_lte[cnum]->sel_2a_h[arrn][arrx][arry](3)&c_lte[cnum]-
>bdata_2a_h[arrn][arrx][arry](3)))
#define Dnd (~((~c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](2))&c_lte[cnum]-
>sel_2a_h[arrn][arrx][arry](2)&c_lte[cnum]->adata_2a_h[arrn][arrx][arry](2)|c_lte[cnum]-
>adata_2a_h[arrn][arrx][arry](2)&c_lte[cnum]->sel_2a_h[arrn][arrx][arry](3)&c_lte[cnum]-
>bdata_2a_h[arrn][arrx][arry](2)))
#define Fnd (~((~c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](1))&c_lte[cnum]-
>sel_2a_h[arrn][arrx][arry](2)&c_lte[cnum]->adata_2a_h[arrn][arrx][arry](1)|c_lte[cnum]-
>adata_2a_h[arrn][arrx][arry](1)&c_lte[cnum]->sel_2a_h[arrn][arrx][arry](3)&c_lte[cnum]-
>bdata_2a_h[arrn][arrx][arry](1)))
#define Hnd (~((~c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](0))&c_lte[cnum]-
>sel_2a_h[arrn][arrx][arry](2)&c_lte[cnum]->adata_2a_h[arrn][arrx][arry](0)|c_lte[cnum]-
>adata_2a_h[arrn][arrx][arry](0)&c_lte[cnum]->sel_2a_h[arrn][arrx][arry](3)&c_lte[cnum]-
>bdata_2a_h[arrn][arrx][arry](0)))


// LTV STATE MACHINE
//=================================================================

//STATES
#define CMDMODE          0x0              // INITIALIZE LTV mode
```

```
#define RUN             0x1                     // NORMAL mode

//COMMANDS
#define NOP             0x0                     // NO OPERATION
#define LOAD_TCR        0x4                     // load test control register
#define LOAD_GAG        0x5                     // load global address generator
#define LOAD_FZR        0x6                     // load freeze register
#define LOAD_CSG        0x7                     // load cluster stimulus generator
#define LOAD_GSC        0x8                     // load global stimulus generator
#define RUN_SS          0xC                     // RunSingleStep
#define RUN_FOE             0xD                         // RunFreezeOnError
#define RUN_FOC             0xE                         // RunFreezeOnCount
#define RUN_FL          0xF                     // RunForLife

// ALU FUNCTION SELECTION TABLE
//=============================================================================
//
// Merged control wire
//
// LAN CNO S3 S2 S1 S0    X = XOR
// ^              ^    N = XNOR
// |              |    * = LEFT SHIFT
// MSB           LSB
//
//          LOGIC          ARITHMETIC
//
// S3 S2 S1 S0  LAN=H          LAN=L,CNO=H                 LAN=L,CNO=L
// L  L  L  L  F=/A        F=A                     F=A+1
// L  L  L  H  F=/(A+B)    F=A+B                   F=(A+B)+1
// L  L  H  L  F=(/A)B         F=A+/B                  F=(A+/B)+1
// L  L  H  H  F=0             F=-1(2's comp)          F=0
// L  H  L  L  F=/(AB)         F=A+A(/B)               F=A+A(/B)+1
// L  H  L  H  F=/B            F=(A+B)+A(/B)           F=(A+B)+A(/B)+1
// L  H  H  L  F=AXB           F=A-B-1                     F=A-B
// L  H  H  H  F=A(/B)         F=A(/B)-1               F=(/A)B
// H  L  L  L  F=/A+B          F=A+AB                      F=A+AB+1
// H  L  L  H  F=/AN/B         F=A+B                   F=A+B+1
// H  L  H  L  F=B             F=(A+/B)+AB             F=(A+/B)+AB+1
// H  L  H  H      F=AB        F=AB-1                  F=AB
// H  H  L  L      F=1         F=A+A*                      F=A+A+1
// H  H  L  H      F=A+/B      F=(A+B)+A               F=(A+B)+A+1
// H  H  H  L      F=A+B       F=(A+/B)+A              F=(A+/B)+A+1
// H  H  H  H      F=A         F=A-1                   F=A
*};

//=============================================================================
//
//                      Global Clock
//
//=============================================================================

Container k
{
   Signal clk               {clock, timing="|_/~|~\_|"};        // global clock
   Signal tck               {clock, timing="|_/~|~\_|"};        // tck clock
```

```
};

Container c_elu_gen[16]{
 Signal newcsg_1a_h(16,0);                    // new csg
 Signal csg_2a_h(16,0);                              // cluster stimulus generator bits
 Signal xext_1a_h;                            // lfsr xor term
 Signal ldcsg_1a_h;                           // load generator
 Signal recir_1a_h;                                  // recirculate contents of lfsr
 Signal rdcsg_2a_h[8](8,0);                          // rotated csg data inputs to lte array
   Signal rccsg_2a_h[4](4,0);                        // rotated csg control inputs to lte array
 Signal svector(16,0);                        // reset value
};

Container c_ala_gen[16]{
 Signal newcsg_1a_h(16,0);                    // new csg
 Signal csg_2a_h(16,0);                     // cluster stimulus generator bits
 Signal xext_1a_h;                                   // lfsr xor term
 Signal ldcsg_1a_h;                                  // load generator
 Signal recir_1a_h;                                  // recirculate contents of lfsr
 Signal rdcsg_2a_h[8](8,0);                          // rotated csg inputs to lte array
 Signal rccsg_2a_h[4](4,0);                          // rotated csg control inputs to lte array
 Signal svector(16,0);                               // reset value
};

Container c_elu_dec[16]{
 Signal wdline_3a_h(3,0);                    // Word line
 Signal radd_3a_h(1,0);                      // piped radd
};

Container c_ala_dec[16]{
 Signal wdline_3a_h(3,0);                    // Word line
 Signal radd_3a_h(1,0);                      // piped radd
};

Container c_elu[16]{
 Signal zout_3a_h[8](7,0);                           // Output of upper lte column
};

Container c_ala[16]{
 Signal zout_3a_h[8](7,0);                           // Output of lower lte column
};

Container c_cmp[16]{
 Signal elu_dataout_4a_h(7,0);               // shared output data
 Signal ala_dataout_4a_h(7,0);               // shared output data
 Signal s_dataout_4a_h(7,0);                         // output of cluster
 Signal elu_dataout_3a_h(7,0);               // upper array
 Signal ala_dataout_3a_h(7,0);               // lower array
 Signal s_fail_5a_h;                                 // failure signal
 Signal asel_4a_h;                                   // delayed array selector
 Signal csel_4a_h;                                   // cluster select
 Signal csel_5a_h;                                   // cluster select
 Signal cadd_3a_h(2,0);                      // piped cadd
};//c_cmp
```

```
Container c_lte[16]{
  Signal adata_2a_h[2][8][4](4,0);          // a input
  Signal bdata_2a_h[2][8][4](4,0);          // b input
  Signal sel_2a_h[2][8][4](3,0);            // select input
  Signal cn0_2a_h[2][8][4];                       // inverted carry input
  Signal lan_2a_h[2][8][4];                 // mode control input
  Signal fn_2a_h[2][8][4](4,0);             // output no carry in
  Signal f_2a_h[2][8][4](4,0);              // output
  Signal xn_2a_h[2][8][4];                  // carry propagate output
  Signal yn_2a_h[2][8][4];                  // carry generate output
  Signal cnp4_2a_h[2][8][4];                      // inverted carry output
  Signal aeb_2a_h[2][8][4];                       // comparator output
  Signal zdata_3a_h[2][8][4](7,0);          // output data

}; //lte


Container s{
  Signal c_gen_ldcsg_1a_h;                        // Load generator
  Signal run_1a_h;                                // run generators
  Signal c_dec_radd_2a_h(1,0);              // upper row address
  Signal c_cmp_cadd_2a_h(2,0);              // Column address
  Signal c_asel_2a_h;                             // array selector
  Signal c_csel_2a_h(15,0);                       // cluster selector
  Signal ldgag_1a_h;                              // load address generator
  Signal marker_2a_h;                             // marker

  Signal cstate_a_h;                              // current state
  Signal nstate_a_h;                              // new state
  Signal reset_h;                                 // global reset
  Signal p_stfail_a_h;                            // self test fail
  Signal m_asloc_3a_h(23,0);                            // location of stimulus and address

  Signal ldtcr_1a_h;                              // load tcr
  Signal ldfzr_1a_h;                              // load fzr
  Signal ldgsc_1a_h;                              // load gsc
  Signal tcr_a_h(11,0);                           // test control register
  Signal fzr_a_h(16,0);                           // freeze register
  Signal ctapr_a_h(2,0);                    // clock tap register
  Signal ctapsel_a_h(7,0);                  // decoded clock tap

  Signal instr_1a_h(3,0);                   // instruction latch
  Signal outc_a_h(7,0);                           // decoded output select control
  Signal errm_a_h;                                // error mode cc or bc
  Signal err_selm_a_h;                            // error mode LTV or selected cluster
  Signal p_output_a_h(7,0);                       // output signals to pins
  Signal ofs_a_h(7,0);                            // decoded output select
  Signal c_cmp_oce_3a_h;                    // or cluster error

  Signal k_transclk_h;                            // transition clock

    Signal s_ext_1a_h;                                  // counter first stage xor term
  Signal a_ext_1a_h;                              // address xor term
  Signal newgsc_1a_h(16,0);                       // new msc
  Signal gsc_2a_h(16,0);                    // global stimulus counter
  Signal newgadd_1a_h(13,0);                      // new gadd
```

```
 Signal gadd_2a_h(13,0);          // global address outputs
 Signal gadd_run_a_h;                    // increment gadd
 Signal gadd_zerod_h;                    // gadd zero detect
 Signal gadd_reset_h;                    // gadd reset
 Signal sdone_2a_h;                      // stimulus done
 Signal gsc_recir_a_h;                           // recirculate first stage gsc
 Signal gadd_recir_a_h;          // recirculate gadd
 Signal gsc_cout_a_h;                            // carry out of first stage gsc

 Signal c_iddq_smode;                            // enable selected cluster for Iddq testing

 Signal srun_1a_h;                       // start run
 Signal pcrun_a_h;                       //
 Signal ncrun_a_h;                       //
 Signal crun_a_h;                        // activity averaging signal
 Signal plfreeze_a_h;                    // latched freeze signal
 Signal pfreeze_a_h;                     // latched freeze signal
 Signal noop_1a_h;                       // no op instruction
 Signal ent_runm_a_h;                    // enter run mode
 Signal datain_1a_h(16,0);               // Data from pins
 Signal pfoc_2a_h;                       // init frz on gsc and frzr
 Signal foc_3a_h;                        // init frz on gsc and frzr
 Signal frzc_1a_h;                       // freeze LTV on count
 Signal frze_1a_h;                       // freeze LTV on error
 Signal frzss_1a_h;                      // freeze LTV on single step
 Signal match_2a_h;                      // match of gsc and fzr
 Signal p_fail_a_h;                      // Fail signal to external logic

 Signal ccfail_6a_h;                     // cycle per cycle fail
 Signal plfail_6a_h;                     // latch fail from any cycle
 Signal lfail_7a_h;                      // latch fail from any cycle
 Signal bcfail_a_h;                      // bist cycle fail
 Signal marker_6a_h;                     // marker
 Signal marker_7a_h;                     // marker
 Signal freeze_a_h;                      // freeze signal

 Signal en_tckavg_a_l;                   // enable tck averaging
 Signal ppulsed_marker_6a_h;                     // Pulse delayed marker signal to external
                                                 pins
 Signal p_pulsed_marker_7a_h;            // Pulse delayed marker signal to external pins
 Signal sync_tck_a_h;                    // Syncronized tck signal
 Signal scond_a_h(8,0);                  // Averaging delay
 Signal new_scond_a_h(8,0);              // new Avr delay
 Signal scond_ext_a_h;                   // xor feedback input
 Signal scond_reset_h;                   // reset
 Signal scond_recir_a_h;                 // recirculate
 Signal scond_cout_a_h;                  // Carry out
 Signal reset_l;                         // reset
 Signal or_ccfail_5a_h;                  // or'd fail from clusters
 Signal mux_ccfail_5a_h;                 // muxed fial from clusters
 Signal int_dataout_a_h[2][4](8,0);              // intermediate mux output values from
                                                 clusters
 Signal dataout_4a_h(7,0);               // output from selected cluster
}; //s

Container p{
```

```
  Signal s_datain_a_h(16,0);                    // data from input pins
  Signal s_control_a_h(3,0);                    // control input pins
  Signal tck_h;                                 // clock
  Signal s_reset_l;                     // reset
  Signal fail_h;                                // fail output
  Signal dataout_h(7,0);                // Output data
  Signal Marker_h;                              // marker signal
  Signal ltvclk_out_h;                          // clock output
  Signal unique_fail_h;                         // unique failure
}; // p

Container f{
  Signal fuse_h(47,0);                          // Fuse ID
}; //f
```

Ltv.mdl

```
//
// ltv.mdl
//
// $Log: ltv.mdl,v $
// Revision 1.2  1997/09/11  23:06:01  echeruo
// Basic frame work for LTV done!
//
// Revision 1.1  1997/07/24  18:09:10  echeruo
// Initial revision
//
//

#include "ltv__cnt.hxx"

#ifdef _MXX_RTL

MXX_VERSION("@(#) $Id: ltv.mdl,v 1.2 1997/09/11 23:06:01 echeruo Exp $");

void ewl_main();
void ltv_k_build();
void ltv_c_build();
void ltv_s_build();
void ltv_p_build();
void ltv_f_build();

void ewl_main()
{
  // build all boxes in LTV
  ltv_k_build();
  ltv_c_build();
  ltv_s_build();
  ltv_p_build();
  ltv_f_build();
} // ewl_main()

void ltv_k_build()
{
}//ltv_k_build()
```

```
void ltv_c_build()
{
  // initialize reset values of clusters
  c_elu_gen[0]->svector(16,0) = 0x1;
  c_ala_gen[0]->svector(16,0) = 0x1;
  c_elu_gen[1]->svector(16,0) = 0x2490;
  c_ala_gen[1]->svector(16,0) = 0x2490;
  c_elu_gen[2]->svector(16,0) = 0xc100;
  c_ala_gen[2]->svector(16,0) = 0xc100;
  c_elu_gen[3]->svector(16,0) = 0xb9a4;
  c_ala_gen[3]->svector(16,0) = 0xb9a4;
  c_elu_gen[4]->svector(16,0) = 0x12000;
  c_ala_gen[4]->svector(16,0) = 0x12000;
  c_elu_gen[5]->svector(16,0) = 0x9001;
  c_ala_gen[5]->svector(16,0) = 0x9001;
  c_elu_gen[6]->svector(16,0) = 0x16c90;
  c_ala_gen[6]->svector(16,0) = 0x16c90;
  c_elu_gen[7]->svector(16,0) = 0x7748;
  c_ala_gen[7]->svector(16,0) = 0x7748;
  c_elu_gen[8]->svector(16,0) = 0x8200;
  c_ala_gen[8]->svector(16,0) = 0x8200;
  c_elu_gen[9]->svector(16,0) = 0x16100;
  c_ala_gen[9]->svector(16,0) = 0x16100;
  c_elu_gen[10]->svector(16,0) = 0x2081;
  c_ala_gen[10]->svector(16,0) = 0x2081;
  c_elu_gen[11]->svector(16,0) = 0x7cd0;
  c_ala_gen[11]->svector(16,0) = 0x7cd0;
  c_elu_gen[12]->svector(16,0) = 0x4920;
  c_ala_gen[12]->svector(16,0) = 0x4920;
  c_elu_gen[13]->svector(16,0) = 0x1a690;
  c_ala_gen[13]->svector(16,0) = 0x1a690;
  c_elu_gen[14]->svector(16,0) = 0x1b248;
  c_ala_gen[14]->svector(16,0) = 0x1b248;
  c_elu_gen[15]->svector(16,0) = 0xf9a5;
  c_ala_gen[15]->svector(16,0) = 0xf9a5;
}//ltv_c_build()


void ltv_s_build()
{
  // control and address generation
  // ================================================================
  // LTV State Machine
  // Two modes CMDMODE and RUN
  // assertion of reset, returns state machine to CMDMODE

  s->ent_runm_a_h = s->srun_1a_h & (~s->freeze_a_h);

  s->nstate_a_h = SWITCH(SEL(s->reset_h),
          CASE(1),CMDMODE,
          CASE(0),SWITCH( SEL(s->cstate_a_h),
                  CASE(CMDMODE),SWITCH(SEL(s->ent_runm_a_h),
                          CASE(0),CMDMODE,
                          CASE(1),RUN),
                  CASE(RUN),SWITCH(SEL(s->freeze_a_h),
```

```
                        CASE(0),RUN,
                        CASE(1),CMDMODE)));

s->cstate_a_h = DFLOP(CLK(A_CLK),s->nstate_a_h);

// INSTRUCTION DECODE
//Decode all instructions in each state
//Bits of instruction decode are as follows
// 0 - noop_1a_h
// 1 - ldtcr_1a_h
// 2 - ldgag_1a_h
// 3 - ldfzr_1a_h
// 4 - ldcsg_1a_h
// 5 - ldgsc_1a_h
// 6 - srun_1a_h
// 7 - frzc_1a_h
// 8 - frze_1a_h
// 9 - frzss_1a_h
//
// STATE      Instruction      decode (6-0)
// CMDMODE NOP                 0000000001
// RUN        LOAD_TCR         0000000010
//            LOAD_GAG         0000000100
//            LOAD_FZR         0000001000
//            LOAD_CSG         0000010000
//            LOAD_GSC         0000100000
//            RUN_SS                  1001000000
//            RUN_FOE                 0101000000
//            RUN_FOC                 0011000000
//            RUN_FL                  0001000000

CONCAT(s->frzss_1a_h,s->frze_1a_h,s->frzc_1a_h,s->srun_1a_h,s->ldgsc_1a_h,
            s->c_gen_ldcsg_1a_h,s->ldfzr_1a_h,s->ldgag_1a_h,s->ldtcr_1a_h,s->noop_1a_h)
      = MSB_EXTEND(10,~s->reset_h)&SWITCH( SEL(s->instr_1a_h(3,0)),
            CASE(NOP),0x1,
            CASE(LOAD_TCR),0x2,
            CASE(LOAD_GAG),0x4,
            CASE(LOAD_FZR),0x8,
            CASE(LOAD_CSG),0x10,
            CASE(LOAD_GSC),0x20,
            CASE(RUN_SS),0x240,
            CASE(RUN_FOE),0x140,
            CASE(RUN_FOC),0xC0,
            CASE(RUN_FL),0x40,
            DEFAULT,0x0);

// LTV run signal
s->run_1a_h = s->srun_1a_h & (~s->freeze_a_h) & (s->crun_a_h);

//initiates transition between clock domains when entering or leaving run mode
s->k_transclk_h = (s->run_1a_h & (s->cstate_a_h==CMDMODE)) |
                        ((s->freeze_a_h | s->reset_h) & (s->cstate_a_h==RUN));
// Instruction register
// Load instructions when in CMDMODE, else recycle instruction
s->instr_1a_h(3,0) = DFLOP(CLK(A_CLK),
      MUX(EN(s->reset_h),0x0,
```

```
        EN((s->cstate_a_h==CMDMODE) & (~s->reset_h)),p->s_control_a_h(3,0),
        EN((s->cstate_a_h==RUN) & (~s->reset_h)),s->instr_1a_h(3,0)));
// Load Registers
//load tcr
s->tcr_a_h(11,0) = DFLOP(CLK(A_CLK),
        MUX(EN(s->reset_h),0x0,
        EN(s->ldtcr_1a_h),s->datain_1a_h(11,0),
        EN((~s->ldtcr_1a_h) & (~s->reset_h)),s->tcr_a_h(11,0)));

//load fzr
s->fzr_a_h(16,0) = DFLOP(CLK(A_CLK),
        MUX(EN(s->reset_h),0x0,
        EN(s->ldfzr_1a_h),s->datain_1a_h(16,0),
        EN((~s->ldfzr_1a_h) & (~s->reset_h)),s->fzr_a_h(16,0)));

//load ctapr
s->ctapr_a_h(2,0) = DFLOP(CLK(~s->reset_h),p->s_datain_a_h(2,0));

// Decode TCR Register fields

//extract data from fields in tcr
//decode output select control
s->outc_a_h(7,0) = DECODER(IN(s->tcr_a_h(4,2)));
s->ofs_a_h(7,0) = DECODER(IN(p->s_datain_a_h(2,0)));

// Enables tck averaging of activity
s->en_tckavg_a_l = s->tcr_a_h(5);

//extract error mode
s->errm_a_h = s->tcr_a_h(0);
s->err_selm_a_h = s->tcr_a_h(1);

//Decode clock tap
s->ctapsel_a_h(7,0) = DECODER(IN(s->ctapr_a_h(2,0)));

//Cluster output Mux
//=================================================================
s->dataout_4a_h(7,0) = SWITCH(SEL(s->gadd_2a_h(9)),
                CASE(0),SWITCH(SEL(s->gadd_2a_h(8,7)),
                        CASE(0),s->int_dataout_a_h[0][0](7,0),
                        CASE(1),s->int_dataout_a_h[0][1](7,0),
                        CASE(2),s->int_dataout_a_h[0][2](7,0),
                        CASE(3),s->int_dataout_a_h[0][3](7,0)),
                CASE(1),SWITCH(SEL(s->gadd_2a_h(8,7)),
                        CASE(0),s->int_dataout_a_h[1][0](7,0),
                        CASE(1),s->int_dataout_a_h[1][1](7,0),
                        CASE(2),s->int_dataout_a_h[1][2](7,0),
                        CASE(3),s->int_dataout_a_h[1][3](7,0)));


//LTV Output Mux
//=================================================================

s->p_output_a_h(7,0) = MUX(
        EN(s->outc_a_h(0)),s->dataout_4a_h(7,0),
        EN(s->outc_a_h(1)),MUX(
```

```
                EN(s->ofs_a_h(0)),s->gadd_2a_h(7,0),
                EN(s->ofs_a_h(1)),WIDTH_EXTEND(8,s->gadd_2a_h(13,8)),
                NO_ENS_DEF,0x0),
        EN(s->outc_a_h(2)),MUX(
                EN(s->ofs_a_h(0)),s->gsc_2a_h(7,0),
                EN(s->ofs_a_h(1)),s->gsc_2a_h(15,8),
                EN(s->ofs_a_h(2)),WIDTH_EXTEND(8,s->gsc_2a_h(16)),
                NO_ENS_DEF,0x0),
        EN(s->outc_a_h(3)),MUX(
                EN(s->ofs_a_h(0)),f->fuse_h(7,0),
                EN(s->ofs_a_h(1)),f->fuse_h(15,8),
                EN(s->ofs_a_h(2)),f->fuse_h(23,16),
                EN(s->ofs_a_h(3)),f->fuse_h(31,24),
                NO_ENS_DEF,0x0),
        NO_ENS_DEF,0x0);
//Misc bits

// Latch data from pins
s->datain_1a_h(16,0) = DFLOP(CLK(A_CLK),p->s_datain_a_h(16,0));

//reset
//asyncronous asserstion, syncronous deasserstion
s->reset_h = ~(p->s_reset_l & DFLOP(CLK(A_CLK),p->s_reset_l));
s->reset_l = ~s->reset_h;

// freeze on count signal (match of gsc and fzr)
s->match_2a_h = (s->gsc_2a_h(16,0) == s->fzr_a_h(16,0));
s->pfoc_2a_h = (~s->noop_1a_h) & (s->match_2a_h | s->foc_3a_h);
s->foc_3a_h = DFLOP(CLK(A_CLK),s->pfoc_2a_h);

// Activity Averaging
// bit 8 of TCR controls if tck averaging is used
s->sync_tck_a_h = DFLOP(CLK(A_CLK),p->tck_h);

s->pcrun_a_h = s->reset_h | s->sync_tck_a_h | ((~s->scond_cout_a_h) & s->ncrun_a_h);
s->ncrun_a_h = DFLOP(CLK(A_CLK),s->pcrun_a_h);
s->crun_a_h = s->en_tckavg_a_l | s->ncrun_a_h;


// Averaging delay (Int clock divider)
//
//   PRIMITIVE SEARCH PROGRAM VERSION 1.3  MARCH 9, 1986

// DATE = 10-13-97   TIME = 10:29 HRS
// No. of Stages in LFSR =  9
// No. of XOR Gates used for feedback =  1

// Primitive Table:

//     |=============================================================|
//     |     POLYNOMIAL *        | FEEDBACK TAP POSITIONS             ||
//     |=============================================================|
//     | 1000010001 B1           |  9  4                              ||
//     |=============================================================|
//     |    1  primitives found in     1 trials                      ||
//     |=============================================================|
```

```
// LFSR DESIGN ACCEPTED --
//                  .-.-.-.-.-.-.-.-.-.
//                <--|||||||||||<--
//                  '+'-'-'-'-'+'-'-'-'
//                   |       |

s->scond_recir_a_h = (~s->scond_reset_h) & (~s->run_1a_h);
s->scond_ext_a_h = s->scond_a_h(8)^s->scond_a_h(3);
s->scond_reset_h = s->reset_h | (~s->srun_1a_h);

s->new_scond_a_h(8,0) = MUX(
                EN(s->scond_reset_h),0x1,
                EN(s->scond_recir_a_h),s->scond_a_h(8,0),
                EN(s->run_1a_h),CONCAT(s->scond_a_h(7,0),s->scond_ext_a_h));

s->scond_a_h(8,0) = DFLOP(CLK(A_CLK),s->new_scond_a_h(8,0));

//generate carry out of first stage of gsc
s->scond_cout_a_h = (s->scond_a_h(8,0) == 0x100);


// Duration and Address counters
// ================================================================
// The duration and Address counters are implemented as LFSR's
// The duration counter is a duplicate of the stimulus generator that feeds the LTE
// The carry out of the duration counter is used to increment the Address counter
//
//    Address 7 bits              Duration 17 bits
//   .-.-.-.-.-.-.-.       .-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
// --|||||||<--  <--|||||||||||||||||||<--
// |  '+'-'-'-'+'-'-'   |     '+'-'-'-'-'-'-'-'-'-'-'-'+'-'-'-'-'
// |   |     |      |     |                  |
// |                |
// |                |
// |                |
// Marker            Cout
//
//
//


// Global stimulus counter
// ================================================================

// 17 bit LFSR with tap on bit 5
// No. of Stages in LFSR  =  17
// No. of XOR Gates used for feedback =   1

// Primitive Table:

//     |=======================================================|
//     || # |     POLYNOMIAL *    | FEEDBACK TAP POSITIONS          ||
//     |=======================================================|
//     ||  1 | 0400101 B3           |  17  6          ||
//     ||  2 | 0400041 B3           |  17  5          ||
```

```
//    |=============================================================|
//    ||   2  primitives found in     4 trials            ||
//    |=============================================================|
//
// LFSR DESIGN ACCEPTED --
//
//                  .-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
//                <--|||||||||||||||||||||||||||||||||||||||||||||<--
//                  '+'-'-'-'-'-'-'-'-'-'-'-'-'-'-'+'-'-'-'-'-'-'-'
//                   |                                           |
//

s->gsc_recir_a_h = (~s->run_1a_h) & (~s->reset_h) & (~s->ldgsc_1a_h);
s->s_ext_1a_h = s->gsc_2a_h(16)^s->gsc_2a_h(4);

s->newgsc_1a_h(16,0) = MUX(
                    EN(s->reset_h),0x1,
                    EN(s->ldgsc_1a_h),s->datain_1a_h(16,0),
                    EN(s->gsc_recir_a_h),s->gsc_2a_h(16,0),
                    EN(s->run_1a_h),CONCAT(s->gsc_2a_h(15,0),s->s_ext_1a_h));

s->gsc_2a_h(16,0) = DFLOP(CLK(A_CLK),s->newgsc_1a_h(16,0));

//generate carry out of first stage of gsc
s->gsc_cout_a_h = (s->gsc_2a_h(16,0) == 0x10000);

// address generator
// implemented as an LFSR with zero completion
// PRIMITIVE SEARCH PROGRAM VERSION 1.3  MARCH 9, 1986

// DATE = 10-30-97   TIME = 13:05 HRS
// No. of Stages in LFSR  =  6
// No. of XOR Gates used for feedback =  1

// Primitive Table:
//    |=============================================================|
// || # |      POLYNOMIAL * | FEEDBACK TAP POSITIONS       ||
//    |=============================================================|
//    ||  1 | 1000011 B1              |   6  1           ||
//    |=============================================================|
//    ||    1  primitives found in     3 trials            ||
//    |=============================================================|

// LFSR DESIGN ACCEPTED --
//                       .-.-.-.-.-.-.
//                     <--|||||||<--
//                       '+'-'-'-'-'+'
//                        |       |

//global address generator
s->gadd_zerod_h = (s->gadd_2a_h(5,0) == 0);
s->a_ext_1a_h = s->gadd_2a_h(5)^s->gadd_2a_h(0)^s->gadd_zerod_h;
s->gadd_run_a_h = s->run_1a_h & s->gsc_cout_a_h & (~s->marker_2a_h);
s->gadd_recir_a_h = (~s->gadd_run_a_h)&(~s->gadd_reset_h)&(~s->ldgag_1a_h);
s->gadd_reset_h =  (s->reset_h | s->marker_2a_h);
```

```
  s->newgadd_1a_h(13,0) = MUX(
                    EN(s->gadd_reset_h),0x0,
                    EN(s->ldgag_1a_h),s->datain_1a_h(13,0),
                    EN(s->gadd_recir_a_h),s->gadd_2a_h(13,0),
                    EN(s->gadd_run_a_h),CONCAT(s->gadd_2a_h(13,6),s->gadd_2a_h(4,0),s-
>a_ext_1a_h));

  s->gadd_2a_h(13,0) = DFLOP(CLK(A_CLK),s->newgadd_1a_h(13,0));

  //carry out of address counter gives marker signal
  //marker is used to check self test signature and drive off chip
  s->marker_2a_h = (s->gadd_2a_h(5,0)==0x20);

  //generate pulse delayed marker to drive offchip
  s->ppulsed_marker_6a_h = (~s->scond_cout_a_h) & (s->marker_6a_h | s->p_pulsed_marker_7a_h);
  s->p_pulsed_marker_7a_h = DFLOP(CLK(A_CLK),s->ppulsed_marker_6a_h);


  //drive address and stimulus to monitoring logic
  //s->m_asloc_3a_h(23,0) = DFLOP(CLK(A_CLK),DFLOP(CLK(A_CLK),
  //                          CONCAT(s->gadd_2a_h(6,0),s->gsc_2a_h(16,0))));

  //drive address bus
  //=================================================================
  //
  // Address Bus
  //    | 4 3 2 1 0 |
  //      \  / \ /
  //       |   |
  //       |   Row
  //       |   Address
  //       |
  //       Column
  //       Address
  //
  //
  // In life test mode
  //
  // global address is mapped directly in case of offline testing
  //
  //
  //         Address Bus
  //   13   6 5 4 3 2 1 0|
  //    \    / \  / \ /
  //      |     |   |
  //     Cluster |   Row
  //     Select  |   Address
  //             |
  //            Column
  //            Address

  //lower row address uses bit 1 and 0
  s->c_dec_radd_2a_h(1,0) = s->gadd_2a_h(1,0);

  //column address bits 4-2
  s->c_cmp_cadd_2a_h(2,0) = s->gadd_2a_h(4,2);
```

```
//upper or lower array select
s->c_asel_2a_h = s->gadd_2a_h(5);

//select desired cluster
s->c_csel_2a_h(0) =
        s->c_csel_2a_h(2) =
        s->c_csel_2a_h(4) =
        s->c_csel_2a_h(6) =
        s->c_csel_2a_h(8) =
        s->c_csel_2a_h(10) =
        s->c_csel_2a_h(12) =
        s->c_csel_2a_h(14) = ~s->gadd_2a_h(6);

s->c_csel_2a_h(1) =
        s->c_csel_2a_h(3) =
        s->c_csel_2a_h(5) =
        s->c_csel_2a_h(7) =
        s->c_csel_2a_h(9) =
        s->c_csel_2a_h(11) =
        s->c_csel_2a_h(13) =
        s->c_csel_2a_h(15) = s->gadd_2a_h(6);

// Freeze assertion
//=====================================================================
s->plfreeze_a_h = (s->frzc_1a_h & s->foc_3a_h) |                    //freeze on count
                  (s->frze_1a_h & s->foc_3a_h & s->ccfail_6a_h) |       //freeze on error
                  (DFLOP(CLK(A_CLK),s->frzss_1a_h))|                    //run single cycle
                  ((s->frzc_1a_h | s->frze_1a_h) & s->marker_7a_h);  //freeze if past marker
s->pfreeze_a_h = (~s->noop_1a_h) & (s->plfreeze_a_h | s->freeze_a_h);

s->freeze_a_h = DFLOP(CLK(A_CLK),s->pfreeze_a_h);

// error monitor
// ====================================================================

// error mode controls if the cluster signals are wire or'd or not. monitor unit selects cycle
// by cycle of bist cycle error
// marker signal indicates end of Major Cycle (bist cycle)
s->marker_6a_h = DFLOP(CLK(A_CLK),DFLOP(CLK(A_CLK),
                  DFLOP(CLK(A_CLK),DFLOP(CLK(A_CLK),s->marker_2a_h))));
s->marker_7a_h = DFLOP(CLK(A_CLK),s->marker_6a_h);

s->mux_ccfail_5a_h = SWITCH(SEL(s->gadd_2a_h(9)),
                CASE(0),SWITCH(SEL(s->gadd_2a_h(8,7)),
                        CASE(0),s->int_dataout_a_h[0][0](8),
                        CASE(1),s->int_dataout_a_h[0][1](8),
                        CASE(2),s->int_dataout_a_h[0][2](8),
                        CASE(3),s->int_dataout_a_h[0][3](8)),
                CASE(1),SWITCH(SEL(s->gadd_2a_h(8,7)),
                        CASE(0),s->int_dataout_a_h[1][0](8),
                        CASE(1),s->int_dataout_a_h[1][1](8),
                        CASE(2),s->int_dataout_a_h[1][2](8),
                        CASE(3),s->int_dataout_a_h[1][3](8)));

s->or_ccfail_5a_h = (c_cmp[0]->s_fail_5a_h) | (c_cmp[1]->s_fail_5a_h) |
```

```
                        (c_cmp[2]->s_fail_5a_h) | (c_cmp[3]->s_fail_5a_h) |
                        (c_cmp[4]->s_fail_5a_h) | (c_cmp[5]->s_fail_5a_h) |
                        (c_cmp[6]->s_fail_5a_h) | (c_cmp[7]->s_fail_5a_h) |
                        (c_cmp[8]->s_fail_5a_h) | (c_cmp[9]->s_fail_5a_h) |
                        (c_cmp[10]->s_fail_5a_h) | (c_cmp[11]->s_fail_5a_h) |
                        (c_cmp[12]->s_fail_5a_h) | (c_cmp[13]->s_fail_5a_h) |
                        (c_cmp[14]->s_fail_5a_h) | (c_cmp[15]->s_fail_5a_h);

  //determine cycle per cycle fail error
  s->ccfail_6a_h = DFLOP(CLK(A_CLK),SWITCH(SEL(s->err_selm_a_h),
                        CASE(0),s->or_ccfail_5a_h,
                        CASE(1),s->mux_ccfail_5a_h));

  //set SR latch when fail occurs in bist cycle
  s->plfail_6a_h = ~(s->marker_7a_h | s->reset_h) & (s->ccfail_6a_h | s->lfail_7a_h);
  s->lfail_7a_h = DFLOP( CLK(A_CLK),s->plfail_6a_h);

  //latch SR at end of bist cycle
  s->bcfail_a_h = DFLOP(CLK(A_CLK),
                MUX(EN(s->marker_7a_h),s->lfail_7a_h,
                        EN(~s->marker_7a_h),s->bcfail_a_h));

  //select between cycle by cycle or bist mode error
  //or and between LTV error or cluster error
  s->p_fail_a_h = MUX(
        EN(~s->errm_a_h),s->bcfail_a_h,
        EN(s->errm_a_h),s->ccfail_6a_h);


}//ltv_s_build()

void ltv_p_build()
{
  // pins
  // ================================================================

  p->tck_h = k->tck;
  //fail alert
  p->fail_h = s->p_fail_a_h;
  //output data
  p->dataout_h(7,0) = s->p_output_a_h(7,0);

}//ltv_p_build()

void ltv_f_build()
{
}//ltv_f_build()

void ltv_e_build()
{
}//ltv_e_build()

#else // _MXX_RTL
#endif
```

Cluster.mdl

```
int cnum;
int arrn,arrx,arry;

for (cnum=INDEX_FROM;cnum<INDEX_TO;cnum++) {
        // stimulus generators (upper and lower)
        // =======================================================

        // 17 bit LFSR with tap back to bit 5
        // No. of Stages in LFSR  =  17
        // No. of XOR Gates used for feedback =   1

         // Primitive Table:
         //     |=================================================|
         // ||  #  |      POLYNOMIAL *        | FEEDBACK TAP POSITIONS  ||
         //     |=================================================|
         //     ||   1 |  0400101 B3          |  17  6               ||
         //     ||   2 |  0400041 B3          |  17  5               ||
         //   |=================================================|
         //     ||    2  primitives found in      4 trials                  ||
         //     |=============================================================|
         // LFSR DESIGN ACCEPTED --
         //                  .-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
         //               <--|||||||||||||||||||<--
         //                  '+'-'-'-'-'-'-'-'-'-'-'-'-'+'-'-'-'-'
         //                   |                  |

        //check if required to load generator
        c_elu_gen[cnum]->ldcsg_1a_h = (~s->c_asel_2a_h) & s->c_gen_ldcsg_1a_h & s-
>c_csel_2a_h(cnum);
        c_ala_gen[cnum]->ldcsg_1a_h = (s->c_asel_2a_h) & s->c_gen_ldcsg_1a_h & s-
>c_csel_2a_h(cnum);
        c_elu_gen[cnum]->recir_1a_h = ~(c_elu_gen[cnum]->ldcsg_1a_h|s->run_1a_h|s->reset_h);
        c_ala_gen[cnum]->recir_1a_h = ~(c_ala_gen[cnum]->ldcsg_1a_h|s->run_1a_h|s->reset_h);

        //upper csg
        c_elu_gen[cnum]->xext_1a_h = c_elu_gen[cnum]->csg_2a_h(16)^c_elu_gen[cnum]-
>csg_2a_h(4);
        c_elu_gen[cnum]->newcsg_1a_h(16,0) = MUX(
                EN(s->reset_h),c_elu_gen[cnum]->svector(16,0),
                EN(c_elu_gen[cnum]->ldcsg_1a_h),s->datain_1a_h(16,0),
                EN(c_elu_gen[cnum]->recir_1a_h),c_elu_gen[cnum]->csg_2a_h(16,0),
                EN(s->run_1a_h),CONCAT(c_elu_gen[cnum]->csg_2a_h(15,0),c_elu_gen[cnum]-
>xext_1a_h));
        c_elu_gen[cnum]->csg_2a_h(16,0) = DFLOP(CLK(A_CLK),c_elu_gen[cnum]-
        >newcsg_1a_h(16,0));

        //lower csg
        c_ala_gen[cnum]->xext_1a_h = c_ala_gen[cnum]->csg_2a_h(16)^c_ala_gen[cnum]-
>csg_2a_h(4);
        c_ala_gen[cnum]->newcsg_1a_h(16,0) = MUX(
                EN(s->reset_h),c_elu_gen[cnum]->svector(16,0),
                EN(c_ala_gen[cnum]->ldcsg_1a_h),s->datain_1a_h(16,0),
                EN(c_ala_gen[cnum]->recir_1a_h),c_ala_gen[cnum]->csg_2a_h(16,0),
```

```
            EN(s->run_1a_h),CONCAT(c_ala_gen[cnum]->csg_2a_h(15,0),c_ala_gen[cnum]-
>xext_1a_h));

        c_ala_gen[cnum]->csg_2a_h(16,0) = DFLOP(CLK(A_CLK),c_ala_gen[cnum]-
>newcsg_1a_h(16,0));

        //rotate csg outputs
        c_elu_gen[cnum]->rdcsg_2a_h[0](8,0) = c_elu_gen[cnum]->csg_2a_h(8,0);
        c_elu_gen[cnum]->rdcsg_2a_h[1](8,0) =
                CONCAT(c_elu_gen[cnum]->csg_2a_h(1,0),c_elu_gen[cnum]->csg_2a_h(8,2));
        c_elu_gen[cnum]->rdcsg_2a_h[2](8,0) =
                CONCAT(c_elu_gen[cnum]->csg_2a_h(3,0),c_elu_gen[cnum]->csg_2a_h(8,4));
        c_elu_gen[cnum]->rdcsg_2a_h[3](8,0) =
                CONCAT(c_elu_gen[cnum]->csg_2a_h(5,0),c_elu_gen[cnum]->csg_2a_h(8,6));
        c_elu_gen[cnum]->rdcsg_2a_h[4](8,0) =
                CONCAT(c_elu_gen[cnum]->csg_2a_h(7,0),c_elu_gen[cnum]->csg_2a_h(8));
        c_elu_gen[cnum]->rdcsg_2a_h[5](8,0) =
                CONCAT(c_elu_gen[cnum]->csg_2a_h(0),c_elu_gen[cnum]->csg_2a_h(8,1));
        c_elu_gen[cnum]->rdcsg_2a_h[6](8,0) =
                CONCAT(c_elu_gen[cnum]->csg_2a_h(2,0),c_elu_gen[cnum]->csg_2a_h(8,3));
        c_elu_gen[cnum]->rdcsg_2a_h[7](8,0) =
                CONCAT(c_elu_gen[cnum]->csg_2a_h(4,0),c_elu_gen[cnum]->csg_2a_h(8,5));
        c_ala_gen[cnum]->rdcsg_2a_h[0](8,0) = c_ala_gen[cnum]->csg_2a_h(8,0);
        c_ala_gen[cnum]->rdcsg_2a_h[1](8,0) =
                CONCAT(c_ala_gen[cnum]->csg_2a_h(1,0),c_ala_gen[cnum]->csg_2a_h(8,2));
        c_ala_gen[cnum]->rdcsg_2a_h[2](8,0) =
                CONCAT(c_ala_gen[cnum]->csg_2a_h(3,0),c_ala_gen[cnum]->csg_2a_h(8,4));
        c_ala_gen[cnum]->rdcsg_2a_h[3](8,0) =
                CONCAT(c_ala_gen[cnum]->csg_2a_h(5,0),c_ala_gen[cnum]->csg_2a_h(8,6));
        c_ala_gen[cnum]->rdcsg_2a_h[4](8,0) =
                CONCAT(c_ala_gen[cnum]->csg_2a_h(7,0),c_ala_gen[cnum]->csg_2a_h(8));
        c_ala_gen[cnum]->rdcsg_2a_h[5](8,0) =
                CONCAT(c_ala_gen[cnum]->csg_2a_h(0),c_ala_gen[cnum]->csg_2a_h(8,1));
        c_ala_gen[cnum]->rdcsg_2a_h[6](8,0) =
                CONCAT(c_ala_gen[cnum]->csg_2a_h(2,0),c_ala_gen[cnum]->csg_2a_h(8,3));
        c_ala_gen[cnum]->rdcsg_2a_h[7](8,0) =
                CONCAT(c_ala_gen[cnum]->csg_2a_h(4,0),c_ala_gen[cnum]->csg_2a_h(8,5));
        c_elu_gen[cnum]->rccsg_2a_h[0](4,0) = c_elu_gen[cnum]->csg_2a_h(13,9);
        c_elu_gen[cnum]->rccsg_2a_h[1](4,0) =
                CONCAT(c_elu_gen[cnum]->csg_2a_h(10,9),c_elu_gen[cnum]->csg_2a_h(13,11));
        c_elu_gen[cnum]->rccsg_2a_h[2](4,0) =
                CONCAT(c_elu_gen[cnum]->csg_2a_h(12,9),c_elu_gen[cnum]->csg_2a_h(13));
        c_elu_gen[cnum]->rccsg_2a_h[3](4,0) =
                CONCAT(c_elu_gen[cnum]->csg_2a_h(9),c_elu_gen[cnum]->csg_2a_h(13,10));
        c_ala_gen[cnum]->rccsg_2a_h[0](4,0) = c_ala_gen[cnum]->csg_2a_h(13,9);
        c_ala_gen[cnum]->rccsg_2a_h[1](4,0) =
                CONCAT(c_ala_gen[cnum]->csg_2a_h(10,9),c_ala_gen[cnum]->csg_2a_h(13,11));
        c_ala_gen[cnum]->rccsg_2a_h[2](4,0) =
                CONCAT(c_ala_gen[cnum]->csg_2a_h(12,9),c_ala_gen[cnum]->csg_2a_h(13));
        c_ala_gen[cnum]->rccsg_2a_h[3](4,0) =
                CONCAT(c_ala_gen[cnum]->csg_2a_h(9),c_ala_gen[cnum]->csg_2a_h(13,10));

// row decoders
// ================================================================

        c_elu_dec[cnum]->radd_3a_h(1,0) = DFLOP(CLK(A_CLK),s->c_dec_radd_2a_h(1,0));
```

```
        c_ala_dec[cnum]->radd_3a_h(1,0) = DFLOP(CLK(A_CLK),s->c_dec_radd_2a_h(1,0));
        c_elu_dec[cnum]->wdline_3a_h(3,0) = DECODER(IN(c_elu_dec[cnum]->radd_3a_h(1,0)));
        c_ala_dec[cnum]->wdline_3a_h(3,0) = DECODER(IN(c_ala_dec[cnum]->radd_3a_h(1,0)));

// mux and compare logic
// ===================================================================

        //get upper array data
        c_cmp[cnum]->cadd_3a_h(2,0) = DFLOP(CLK(A_CLK),s->c_cmp_cadd_2a_h(2,0));
        c_cmp[cnum]->elu_dataout_3a_h(7,0) = SWITCH(SEL(c_cmp[cnum]->cadd_3a_h(2,0)),
                CASE(0),c_elu[cnum]->zout_3a_h[0](7,0),
                CASE(1),c_elu[cnum]->zout_3a_h[1](7,0),
                CASE(2),c_elu[cnum]->zout_3a_h[2](7,0),
                CASE(3),c_elu[cnum]->zout_3a_h[3](7,0),
                CASE(4),c_elu[cnum]->zout_3a_h[4](7,0),
                CASE(5),c_elu[cnum]->zout_3a_h[5](7,0),
                CASE(6),c_elu[cnum]->zout_3a_h[6](7,0),
                CASE(7),c_elu[cnum]->zout_3a_h[7](7,0));

        //get lower array data
        c_cmp[cnum]->ala_dataout_3a_h(7,0) = SWITCH(SEL(c_cmp[cnum]->cadd_3a_h(2,0)),
                CASE(0),c_ala[cnum]->zout_3a_h[0](7,0),
                CASE(1),c_ala[cnum]->zout_3a_h[1](7,0),
                CASE(2),c_ala[cnum]->zout_3a_h[2](7,0),
                CASE(3),c_ala[cnum]->zout_3a_h[3](7,0),
                CASE(4),c_ala[cnum]->zout_3a_h[4](7,0),
                CASE(5),c_ala[cnum]->zout_3a_h[5](7,0),
                CASE(6),c_ala[cnum]->zout_3a_h[6](7,0),
                CASE(7),c_ala[cnum]->zout_3a_h[7](7,0));

        //get selected array data
        c_cmp[cnum]->asel_4a_h = DFLOP(CLK(A_CLK),DFLOP(CLK(A_CLK),s->c_asel_2a_h));
        c_cmp[cnum]->elu_dataout_4a_h(7,0) = DFLOP(CLK(A_CLK),c_cmp[cnum]-
>elu_dataout_3a_h(7,0));
        c_cmp[cnum]->ala_dataout_4a_h(7,0) = DFLOP(CLK(A_CLK),c_cmp[cnum]-
>ala_dataout_3a_h(7,0));

        //compare upper and lower array output
        c_cmp[cnum]->s_fail_5a_h = DFLOP(CLK(A_CLK),
        (c_cmp[cnum]->elu_dataout_4a_h(7,0) != c_cmp[cnum]->ala_dataout_4a_h(7,0)));
        c_cmp[cnum]->csel_4a_h = DFLOP(CLK(A_CLK),DFLOP(CLK(A_CLK),s-
>c_csel_2a_h(cnum)));
         c_cmp[cnum]->csel_5a_h = DFLOP(CLK(A_CLK),c_cmp[cnum]->csel_4a_h);

         //drive bus if cluster selected
         if (cnum == 0 | cnum == 1)  {
                s->int_dataout_a_h[0][0](7,0) = BUS(EN(c_cmp[cnum]->csel_4a_h),
                        SWITCH(SEL(c_cmp[cnum]->asel_4a_h),
                        CASE(0),c_cmp[cnum]->elu_dataout_4a_h(7,0),
                        CASE(1),c_cmp[cnum]->ala_dataout_4a_h(7,0)));
                s->int_dataout_a_h[0][0](8) = BUS(EN(c_cmp[cnum]->csel_5a_h),
                        c_cmp[cnum]->s_fail_5a_h);
        }

        if (cnum == 2 | cnum == 3)  {
                s->int_dataout_a_h[0][1](7,0) = BUS(EN(c_cmp[cnum]->csel_4a_h),
```

```
                SWITCH(SEL(c_cmp[cnum]->asel_4a_h),
                CASE(0),c_cmp[cnum]->elu_dataout_4a_h(7,0),
                CASE(1),c_cmp[cnum]->ala_dataout_4a_h(7,0)));
        s->int_dataout_a_h[0][1](8) = BUS(EN(c_cmp[cnum]->csel_5a_h),
                c_cmp[cnum]->s_fail_5a_h);
}

if (cnum == 4 | cnum == 5) {
        s->int_dataout_a_h[0][2](7,0) = BUS(EN(c_cmp[cnum]->csel_4a_h),
                SWITCH(SEL(c_cmp[cnum]->asel_4a_h),
                CASE(0),c_cmp[cnum]->elu_dataout_4a_h(7,0),
                CASE(1),c_cmp[cnum]->ala_dataout_4a_h(7,0)));
        s->int_dataout_a_h[0][2](8) = BUS(EN(c_cmp[cnum]->csel_5a_h),
                c_cmp[cnum]->s_fail_5a_h);
}

if (cnum == 6 | cnum == 7) {
        s->int_dataout_a_h[0][3](7,0) = BUS(EN(c_cmp[cnum]->csel_4a_h),
                SWITCH(SEL(c_cmp[cnum]->asel_4a_h),
                CASE(0),c_cmp[cnum]->elu_dataout_4a_h(7,0),
                CASE(1),c_cmp[cnum]->ala_dataout_4a_h(7,0)));
        s->int_dataout_a_h[0][3](8) = BUS(EN(c_cmp[cnum]->csel_5a_h),
                c_cmp[cnum]->s_fail_5a_h);
}

if (cnum == 8 | cnum == 9) {
        s->int_dataout_a_h[1][0](7,0) = BUS(EN(c_cmp[cnum]->csel_4a_h),
                SWITCH(SEL(c_cmp[cnum]->asel_4a_h),
                CASE(0),c_cmp[cnum]->elu_dataout_4a_h(7,0),
                CASE(1),c_cmp[cnum]->ala_dataout_4a_h(7,0)));
        s->int_dataout_a_h[1][0](8) = BUS(EN(c_cmp[cnum]->csel_5a_h),
                c_cmp[cnum]->s_fail_5a_h);
}

if (cnum == 10 | cnum == 11) {
        s->int_dataout_a_h[1][1](7,0) = BUS(EN(c_cmp[cnum]->csel_4a_h),
                SWITCH(SEL(c_cmp[cnum]->asel_4a_h),
                CASE(0),c_cmp[cnum]->elu_dataout_4a_h(7,0),
                CASE(1),c_cmp[cnum]->ala_dataout_4a_h(7,0)));
        s->int_dataout_a_h[1][1](8) = BUS(EN(c_cmp[cnum]->csel_5a_h),
                c_cmp[cnum]->s_fail_5a_h);
}

if (cnum == 12 | cnum == 13) {
        s->int_dataout_a_h[1][2](7,0) = BUS(EN(c_cmp[cnum]->csel_4a_h),
                SWITCH(SEL(c_cmp[cnum]->asel_4a_h),
                CASE(0),c_cmp[cnum]->elu_dataout_4a_h(7,0),
                CASE(1),c_cmp[cnum]->ala_dataout_4a_h(7,0)));
        s->int_dataout_a_h[1][2](8) = BUS(EN(c_cmp[cnum]->csel_5a_h),
                c_cmp[cnum]->s_fail_5a_h);
}

if (cnum == 14 | cnum == 15) {
        s->int_dataout_a_h[1][3](7,0) = BUS(EN(c_cmp[cnum]->csel_4a_h),
                SWITCH(SEL(c_cmp[cnum]->asel_4a_h),
                CASE(0),c_cmp[cnum]->elu_dataout_4a_h(7,0),
```

```
                    CASE(1),c_cmp[cnum]->ala_dataout_4a_h(7,0)));
                s->int_dataout_a_h[1][3](8) = BUS(EN(c_cmp[cnum]->csel_5a_h),
                    c_cmp[cnum]->s_fail_5a_h);
        }
// LTE unit
// ================================================================
 //
 // LTE Logic Test Element is the basic building block of the test vehicle
 // It consists of a 4bit ALU slice similar to the 74181
 // Data is fed to each LTE by a data bus that runs down 8 columns of 4 LTE's
 // The output of each LTE is enabled by row decoders which span the columns
 // driving a common output bus runing down each column
 // upper TLE array
 // Same as lower TLE array
 //================================================================

        for (arrn=0;arrn<2;arrn++)
                for (arry=0;arry<4;arry++)
                        for (arrx=0;arrx<8;arrx++){

                        //assign inputs to each lte in array rotating inputs bits as needed
                        //also divide array into upper (ELU) and lower (ALA) and
                if (arrn == 0) {

                        c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0) =
                                WIDTH_EXTEND(5,c_elu_gen[cnum]->rdcsg_2a_h[arrx](3,0));
                        c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0) =
                                WIDTH_EXTEND(5,c_elu_gen[cnum]->rdcsg_2a_h[arrx](7,4));
                        c_lte[cnum]->cn0_2a_h[arrn][arrx][arry] =
                                c_elu_gen[cnum]->rdcsg_2a_h[arrx](8);
                        c_lte[cnum]->sel_2a_h[arrn][arrx][arry](3,0) =
                                c_elu_gen[cnum]->rccsg_2a_h[arry](3,0);
                        c_lte[cnum]->lan_2a_h[arrn][arrx][arry] =
                                c_elu_gen[cnum]->rccsg_2a_h[arry](4);
                }
                else {
                        c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0) =
                                WIDTH_EXTEND(5,c_ala_gen[cnum]->rdcsg_2a_h[arrx](3,0));
                        c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0) =
                                WIDTH_EXTEND(5,c_ala_gen[cnum]->rdcsg_2a_h[arrx](7,4));
                        c_lte[cnum]->cn0_2a_h[arrn][arrx][arry] =
                                c_ala_gen[cnum]->rdcsg_2a_h[arrx](8);
                        c_lte[cnum]->sel_2a_h[arrn][arrx][arry](3,0) =
                                c_ala_gen[cnum]->rccsg_2a_h[arry](3,0);
                        c_lte[cnum]->lan_2a_h[arrn][arrx][arry] =
                                c_ala_gen[cnum]->rccsg_2a_h[arry](4);
                }
                //determine and perform function
                c_lte[cnum]->fn_2a_h[arrn][arrx][arry](4,0) =
                        SWITCH(SEL(c_lte[cnum]->lan_2a_h[arrn][arrx][arry]),
                        CASE(0),SWITCH(SEL(c_lte[cnum]->sel_2a_h[arrn][arrx][arry](3,0)),
                                                                        // LAN = L
                        CASE(0x00),c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0), // F=A
                        CASE(0x01),c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)| // F=(A|B)
                                c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0),
                        CASE(0x02),c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)| // F=(A|/B)
```

```
        (~c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0)),
CASE(0x03),0xF,                                          // F=-1(2's comp)
CASE(0x04),c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)+// F=A+A(/B)
        (c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)&
        (~c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0))),
CASE(0x05),(c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)| // F=(A|B)+A/B
        c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0))+
        (c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)&
        (~c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0))),
CASE(0x06),c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)- // F=A-B-1
        c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0)+0xF,
CASE(0x07),(c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)&// F=A(/B)-1
        (~c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0)))+0xF,
CASE(0x08),c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)+// F=A+AB
        (c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)&
        c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0)),
CASE(0x09),c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)+// F=A+B
        c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0),
CASE(0x0A),(c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)| // F=(A|/B)+AB
        (~c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0)))+
        (c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)&
        c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0)),
CASE(0x0B),(c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)&// F=AB-1
        c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0))+0xF,
CASE(0x0C),c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)+// F=A+A
        c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0),
CASE(0x0D),(c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)|// F=(A|B)+A
        c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0))+
        c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0),
CASE(0x0E),(c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)|// F=(A|/B)+A
        (~c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0)))+
        c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0),
CASE(0x0F),c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)+0xF),     // F=A-1
CASE(1),SWITCH(SEL(c_lte[cnum]->sel_2a_h[arrn][arrx][arry](3,0)),
                                                         // LAN = H
CASE(0x00),~c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0),          // F=/A
CASE(0x01),~(c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)|// F=/(A|B)
        c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0)),
CASE(0x02),(~c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0))&// F=(/A)B
        c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0),
CASE(0x03),0,                                            // F=0
CASE(0x04),~(c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)&// F=/(AB)
        c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0)),
CASE(0x05),~c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0),          // F=/B
CASE(0x06),c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)^ // F=AXB
        c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0),
CASE(0x07),c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)&// F=A(/B)
        (~c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0)),
CASE(0x08),(~c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0))|// F=(/A)|B
        c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0),
CASE(0x09),~((c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0))^// F=~(AXB)
        (c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0))),
CASE(0x0A),c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0),           // F=B
CASE(0x0B),c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)&           // F=AB
        c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0),
CASE(0x0C),0x1F,                                         // F=1
```

```
                    CASE(0x0D),c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)| // F=A|(/B)
                        (~c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0)),
                    CASE(0x0E),(c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)|// F=A|B
                        c_lte[cnum]->bdata_2a_h[arrn][arrx][arry](4,0)),
                    CASE(0x0F),c_lte[cnum]->adata_2a_h[arrn][arrx][arry](4,0)));          // F=A

        //add carry if needed
        c_lte[cnum]->f_2a_h[arrn][arrx][arry](4,0) =
            SWITCH(SEL(c_lte[cnum]->lan_2a_h[arrn][arrx][arry]),
                    CASE(0),c_lte[cnum]->fn_2a_h[arrn][arrx][arry](4,0)+
                        WIDTH_EXTEND(5,~c_lte[cnum]->cn0_2a_h[arrn][arrx][arry]),
                    CASE(1),c_lte[cnum]->fn_2a_h[arrn][arrx][arry](4,0));

        // calculate carry outs
        c_lte[cnum]->xn_2a_h[arrn][arrx][arry] = ~( Hnd & Fnd & Dnd & Bnd );
        c_lte[cnum]->cnp4_2a_h[arrn][arrx][arry] =
            ~(c_lte[cnum]->yn_2a_h[arrn][arrx][arry] &
            ((~(Fnd&Bnd&Dnd)) | (~(c_lte[cnum]->cn0_2a_h[arrn][arrx][arry]&Hnd))));
        c_lte[cnum]->yn_2a_h[arrn][arrx][arry] =
             ~((And)|(Bnd&Cnd)|(Bnd&Dnd&End)|(Bnd&Dnd&Fnd&Gnd));

        // other output bits
        c_lte[cnum]->aeb_2a_h[arrn][arrx][arry] =
            (c_lte[cnum]->f_2a_h[arrn][arrx][arry](3,0) == 0xF);
        c_lte[cnum]->zdata_3a_h[arrn][arrx][arry](7,0) = DFLOP(CLK(A_CLK),
            CONCAT(c_lte[cnum]->aeb_2a_h[arrn][arrx][arry],
            c_lte[cnum]->cnp4_2a_h[arrn][arrx][arry],
            c_lte[cnum]->yn_2a_h[arrn][arrx][arry],
            c_lte[cnum]->xn_2a_h[arrn][arrx][arry],c_lte[cnum]->f_2a_h[arrn][arrx][arry](3,0)));

        //drive output lines if selected
        if (arrn == 0) {
            c_elu[cnum]->zout_3a_h[arrx](7,0) = BUS(EN(c_elu_dec[cnum]->wdline_3a_h(arry)),
            c_lte[cnum]->zdata_3a_h[arrn][arrx][arry](7,0));
        }
        else {
            c_ala[cnum]->zout_3a_h[arrx](7,0) = BUS(EN(c_ala_dec[cnum]->wdline_3a_h(arry)),
            c_lte[cnum]->zdata_3a_h[arrn][arrx][arry](7,0));
        }

}

}
```

clusterB0.mdl

```
#include "ltv__cnt.hxx"
#define INDEX_FROM 0
#define INDEX_TO 2

void ewl_main() {
#include "cluster.mdl"
}
```

clusterB1.mdl

```
#include "ltv__cnt.hxx"
#define INDEX_FROM 2
#define INDEX_TO 4

void ewl_main() {
#include "cluster.mdl"
}
```

clusterB2.mdl

```
#include "ltv__cnt.hxx"
#define INDEX_FROM 4
#define INDEX_TO 6

void ewl_main() {
#include "cluster.mdl"
}
```

clusterB3.mdl

```
#include "ltv__cnt.hxx"
#define INDEX_FROM 6
#define INDEX_TO 8

void ewl_main() {
#include "cluster.mdl"
}
```

clusterB4.mdl

```
#include "ltv__cnt.hxx"
#define INDEX_FROM 10
#define INDEX_TO 12

void ewl_main() {
#include "cluster.mdl"
}
```

clusterB5.mdl

```
#include "ltv__cnt.hxx"
#define INDEX_FROM 12
#define INDEX_TO 14

void ewl_main() {
#include "cluster.mdl"
}
```

clusterB6.mdl

```
#include "ltv__cnt.hxx"
#define INDEX_FROM 12
#define INDEX_TO 14

void ewl_main() {
#include "cluster.mdl"
}
```

clusterB7.mdl

```
#include "ltv__cnt.hxx"
#define INDEX_FROM 14
#define INDEX_TO 16

void ewl_main() {
#include "cluster.mdl"
}
```

Appendix C: Schematics

C-1: Logic Test Vehicle (LTV) Global Schematic

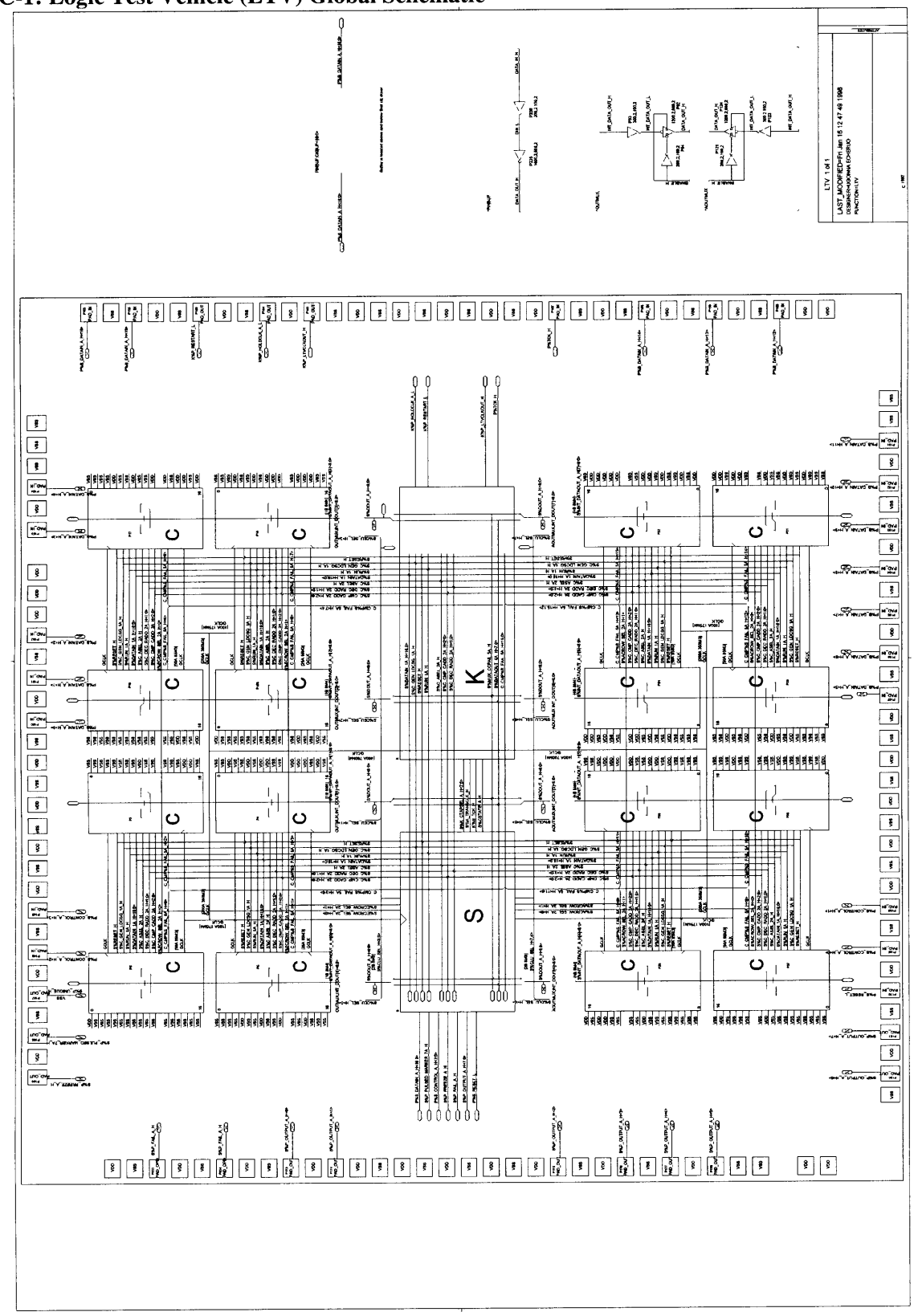

Figure 42 :LTV Global Schematic

C-2: Control Logic

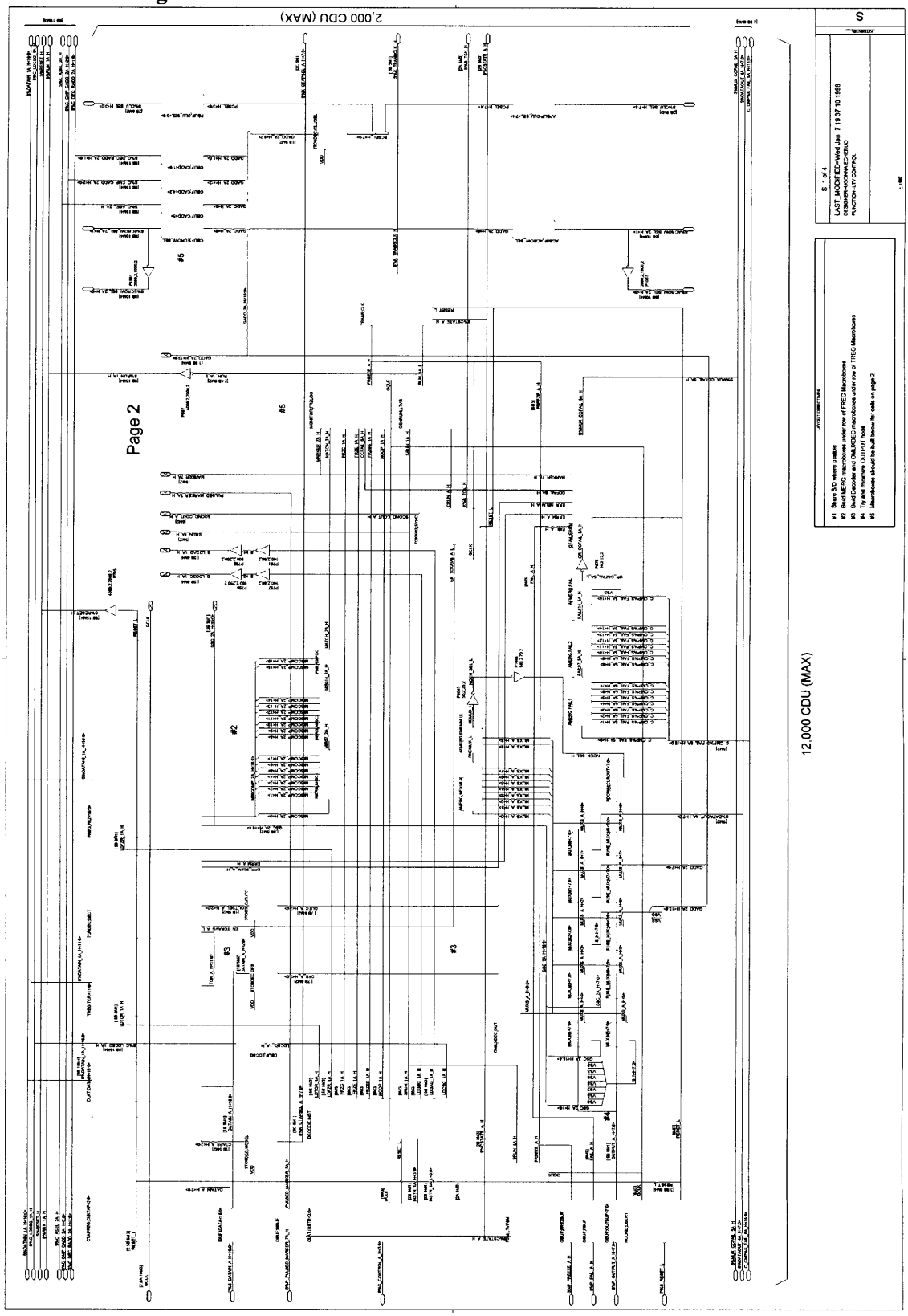

Figure 43 :Control Logic Schematic, Page 1

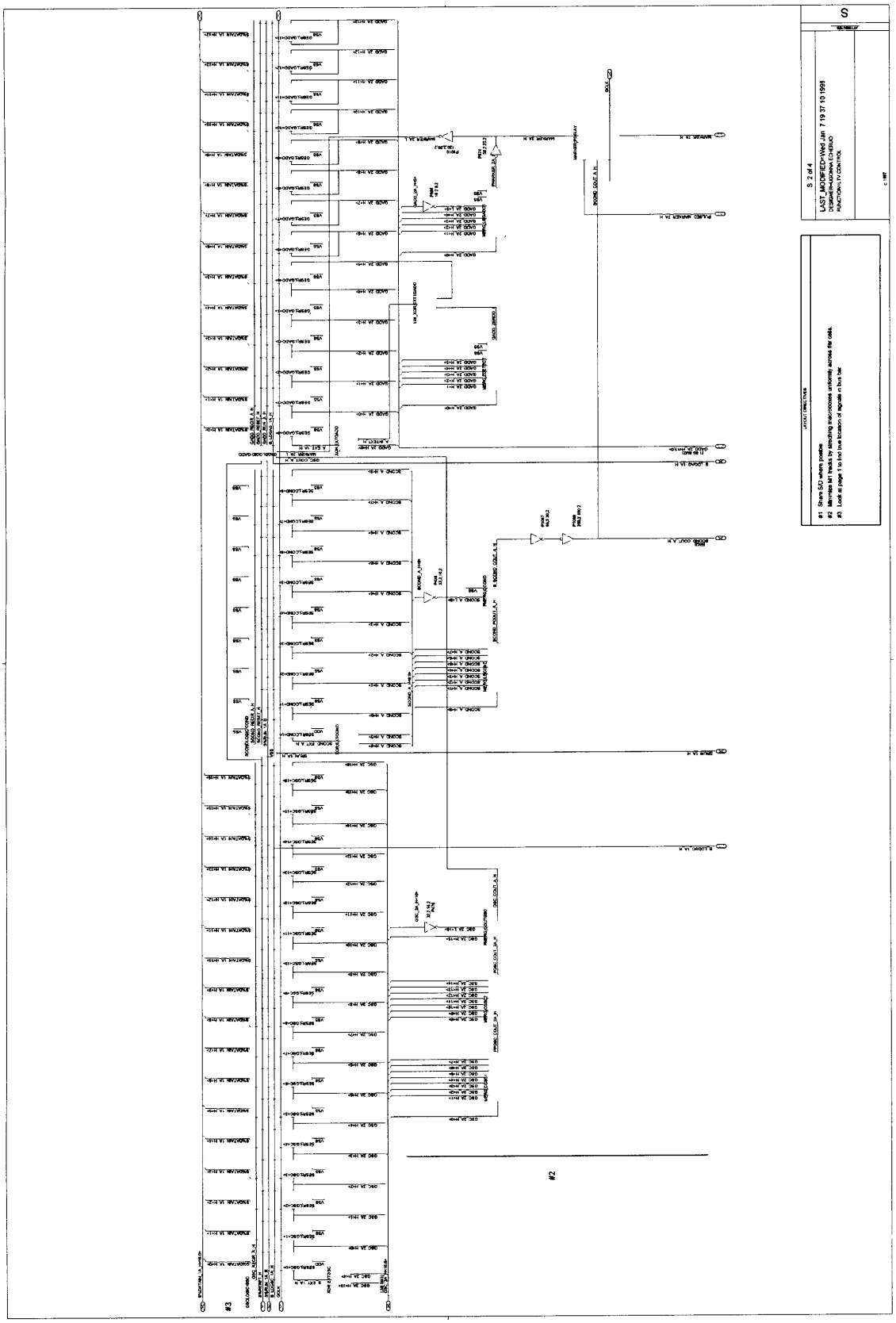

Figure 44 :Control Logic Schematic, Page 2

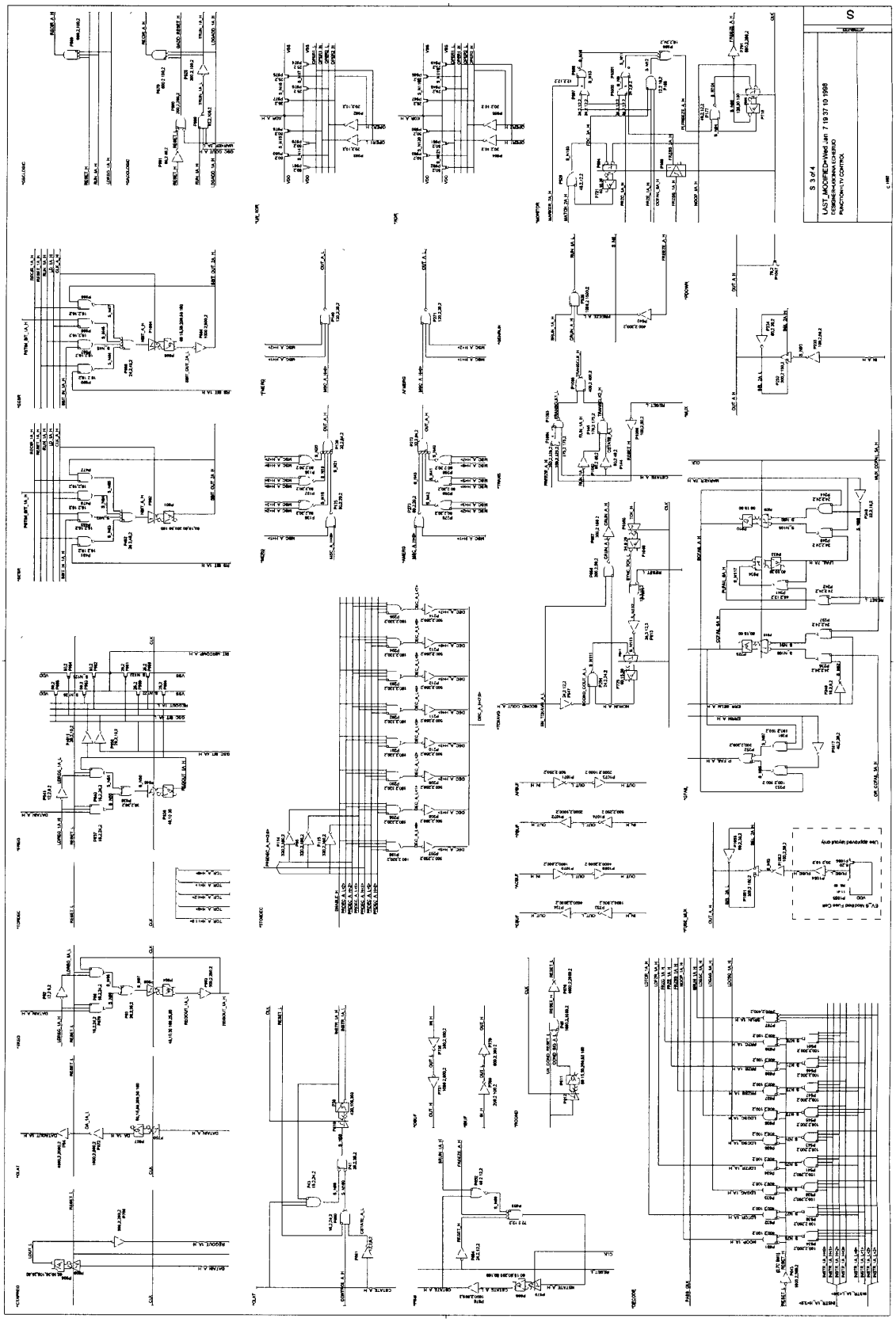

Figure 45 :Control Logic Schematic, Page 3

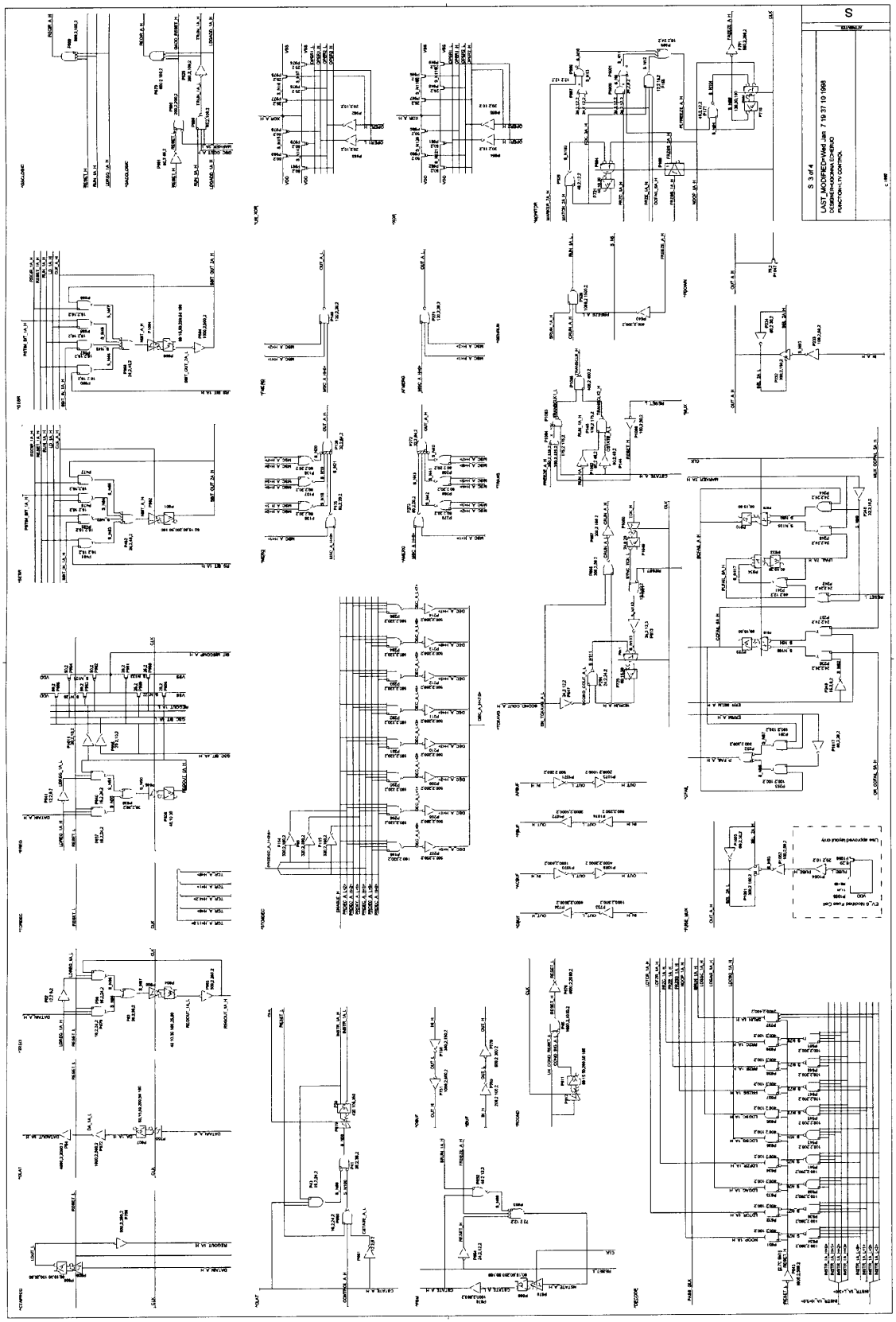

Figure 46 :Control Logic Schematic, Page 4

C-3: Clock Generator

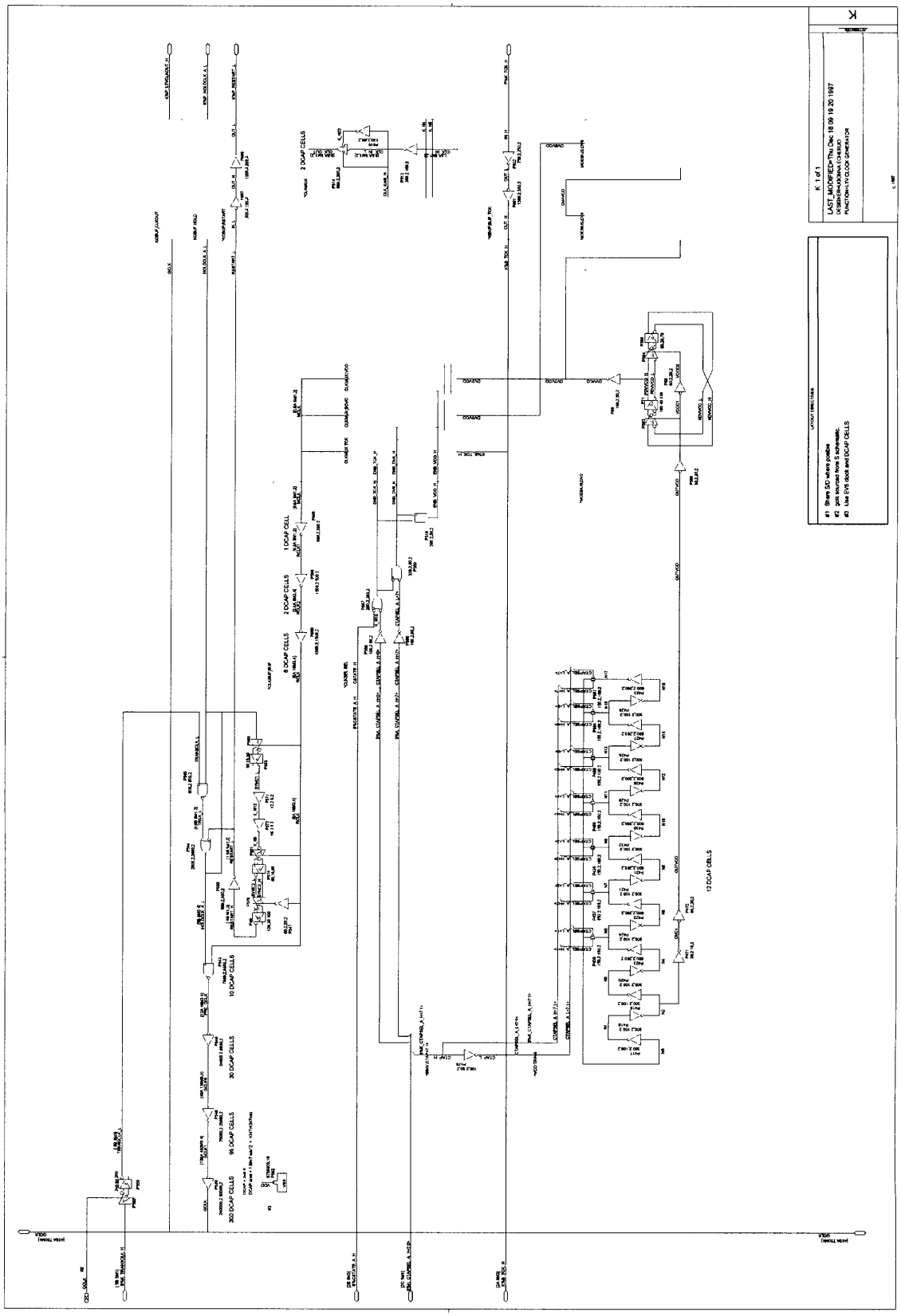

Figure 47 :Clock Generator Schematic

C-4: Fuse Farm

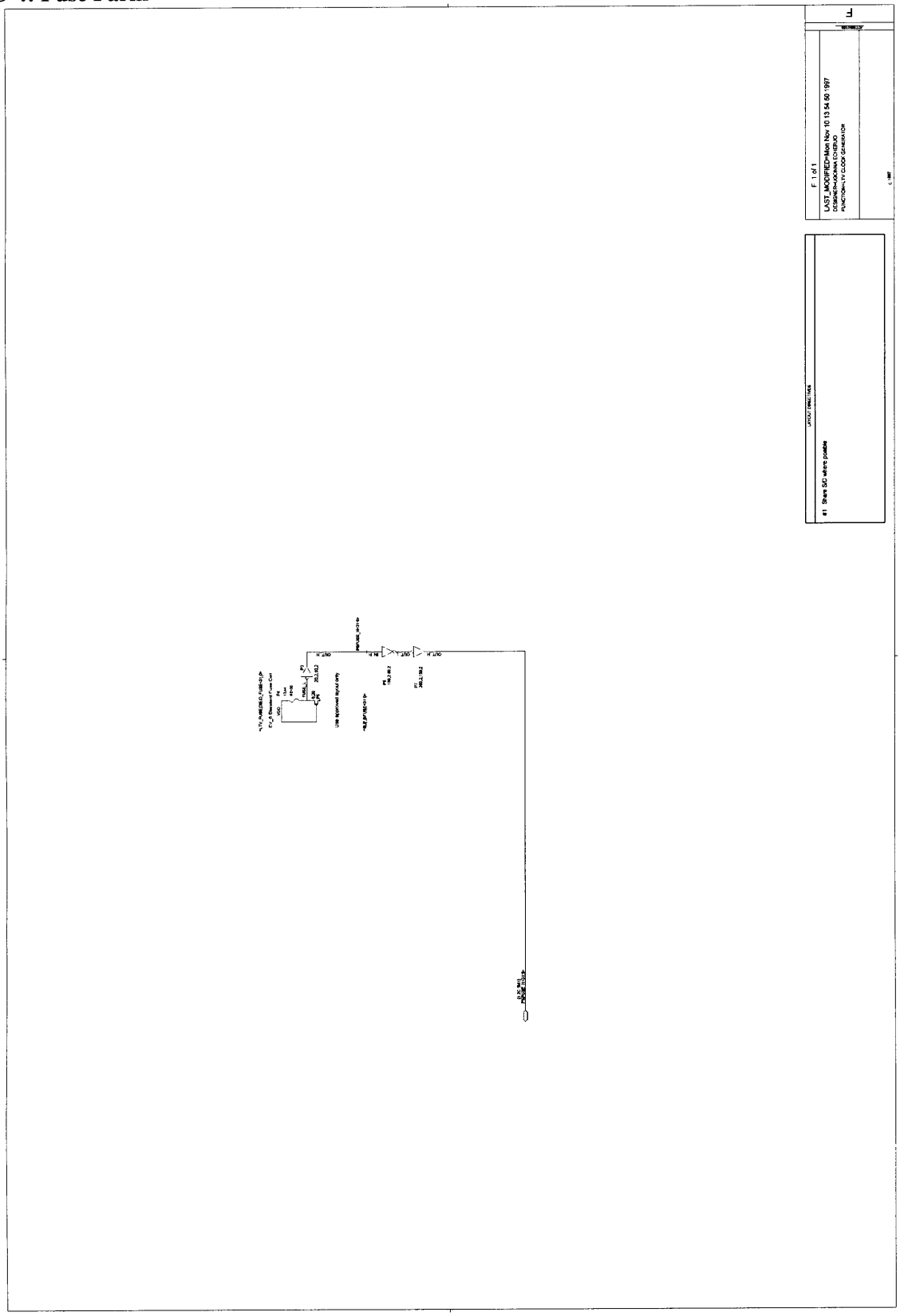

Figure 48 : Fuse Farm Schematic

C-5: Logic Test Cluster (LTC)

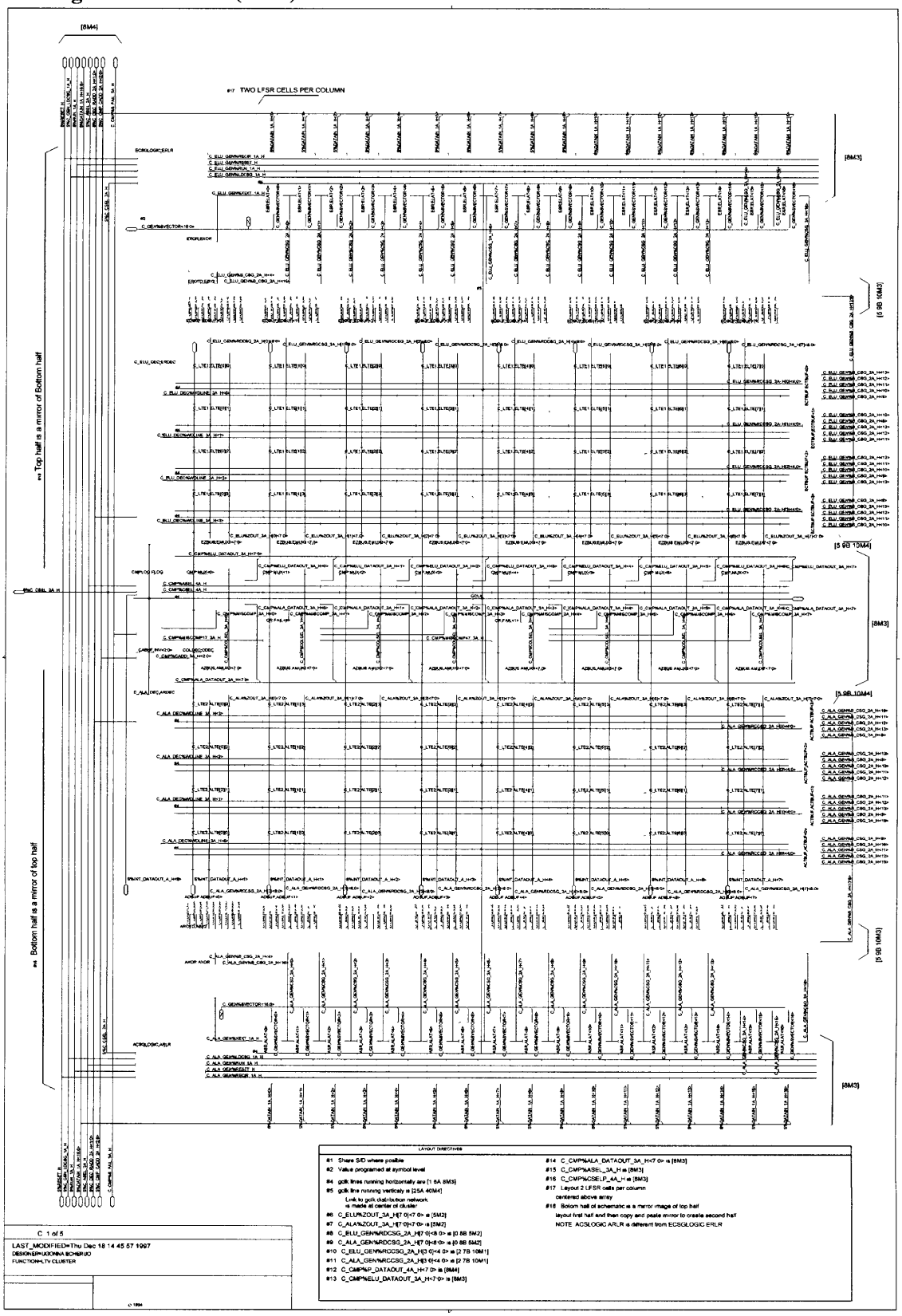

Figure 49 :LTC Global Schematic, Page 1

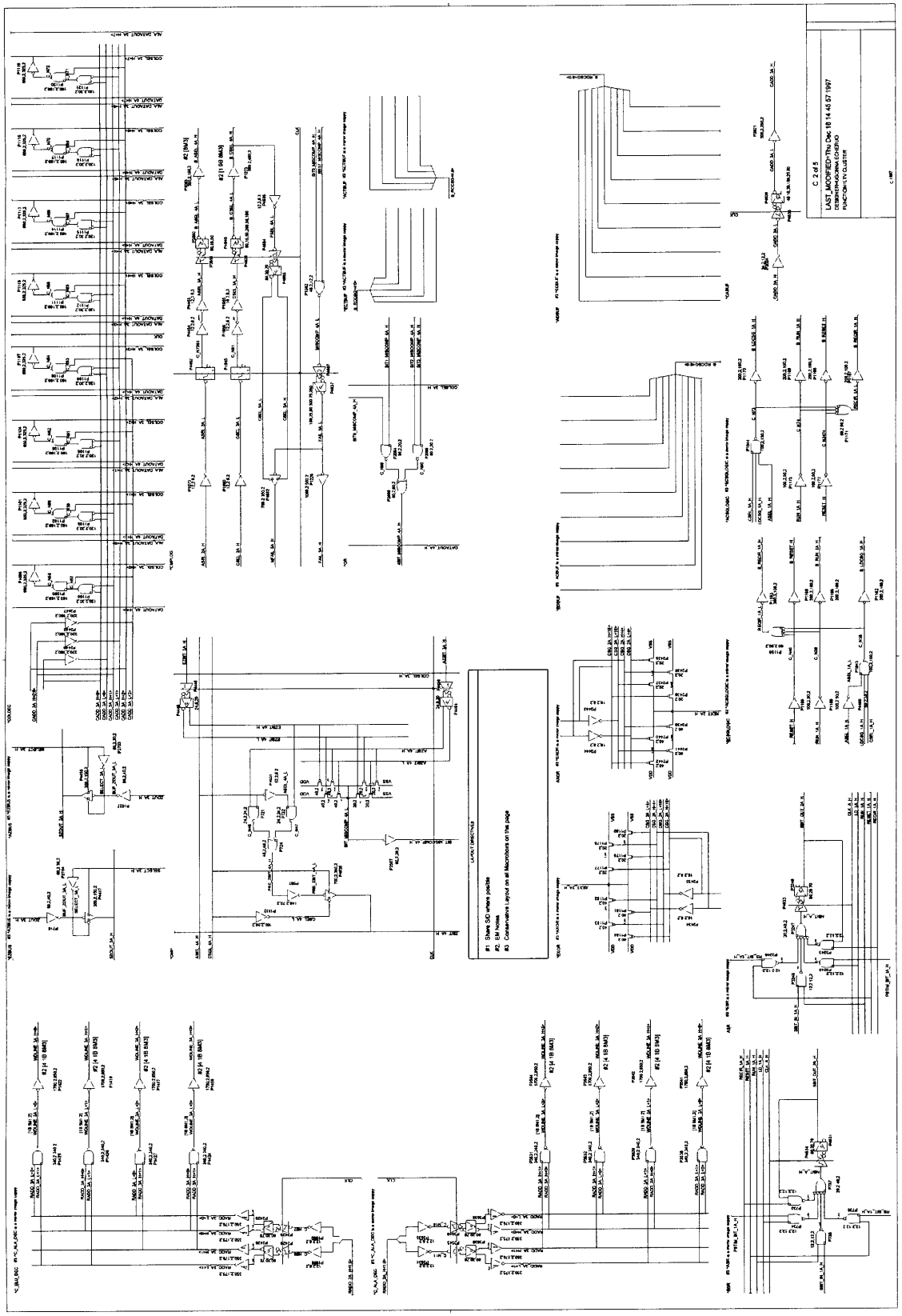

Figure 50 : LTC Schematic, Page 2

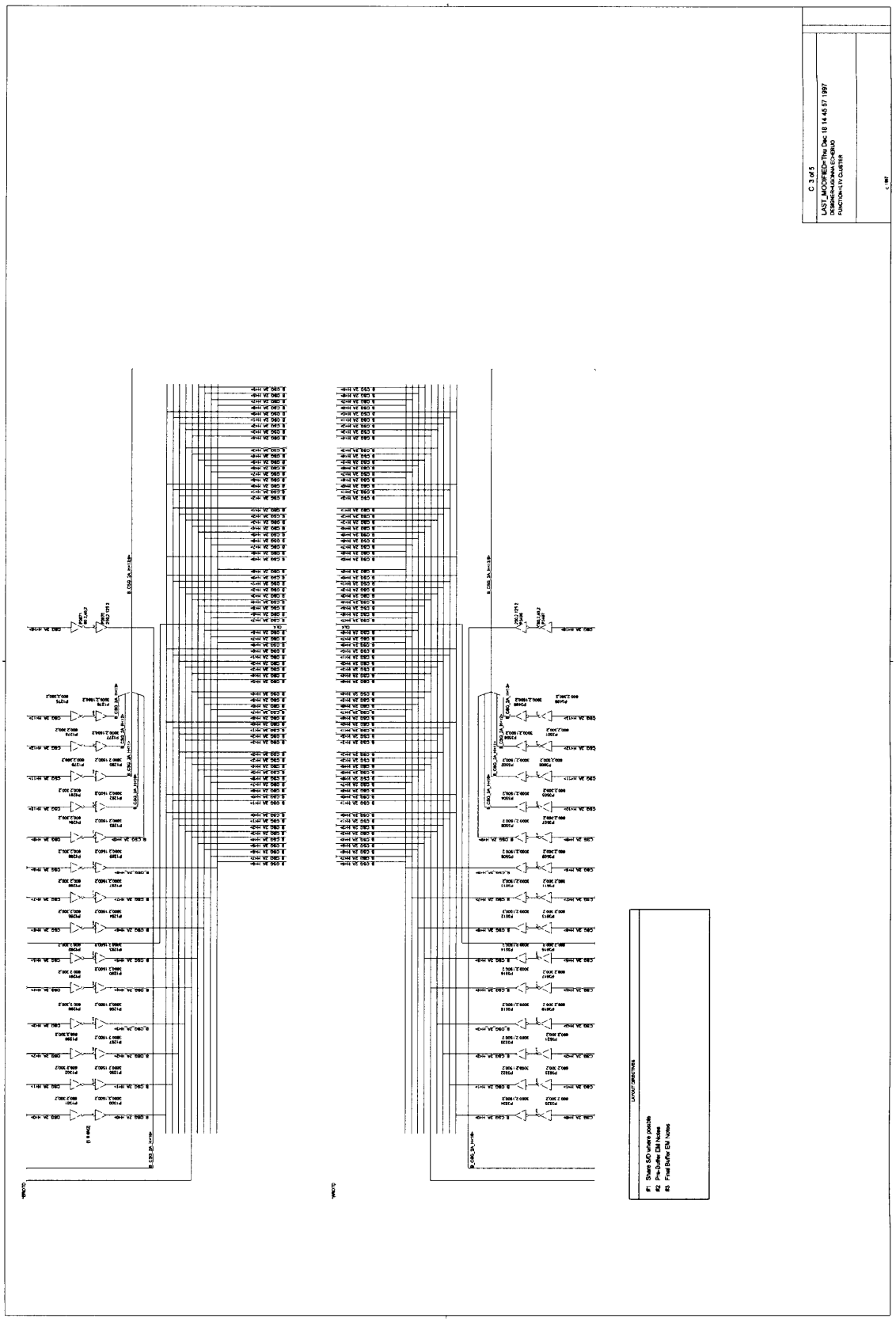

Figure 51 : LTC Schematic, Page 3

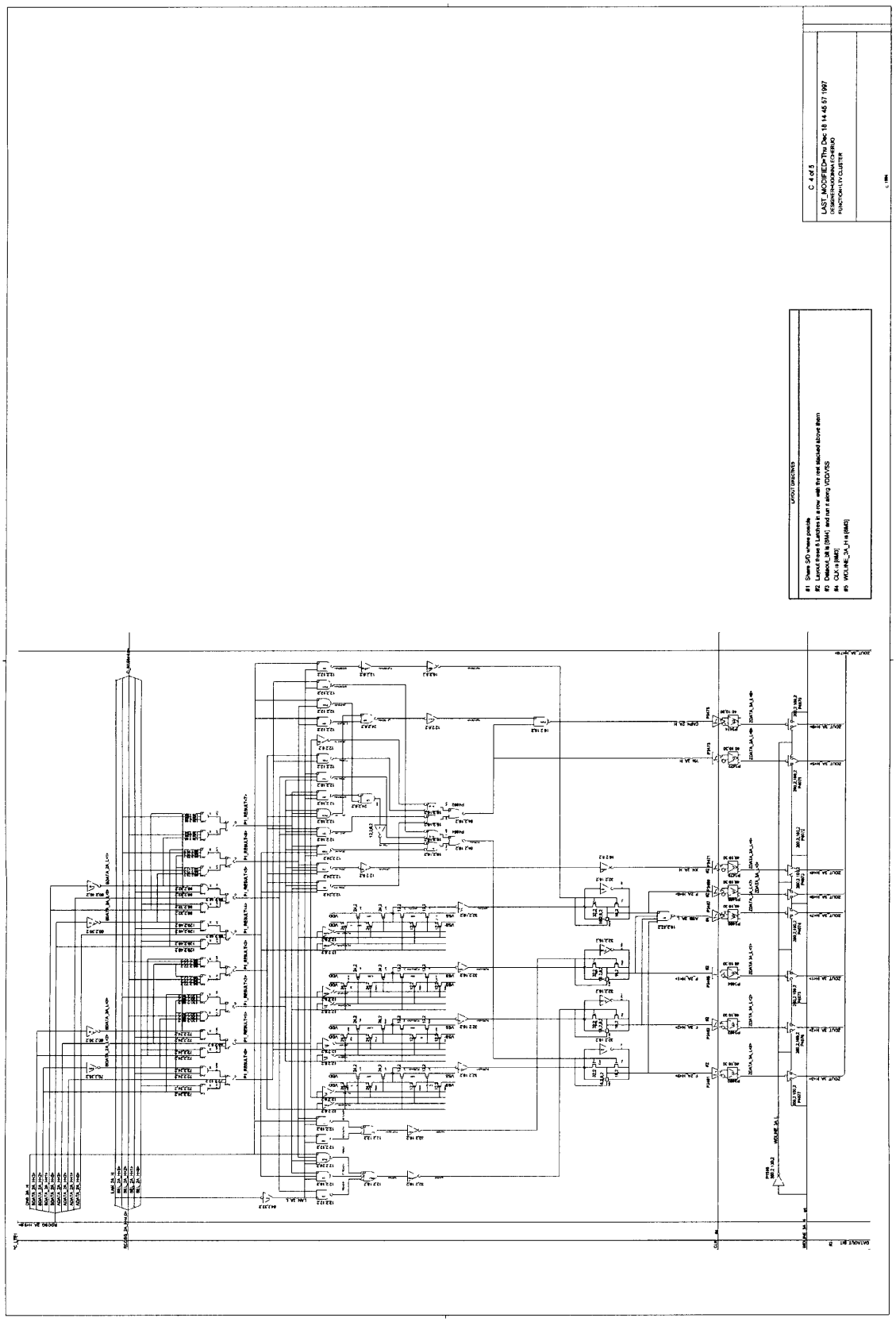

Figure 52 : LTC Schematic, Page 4

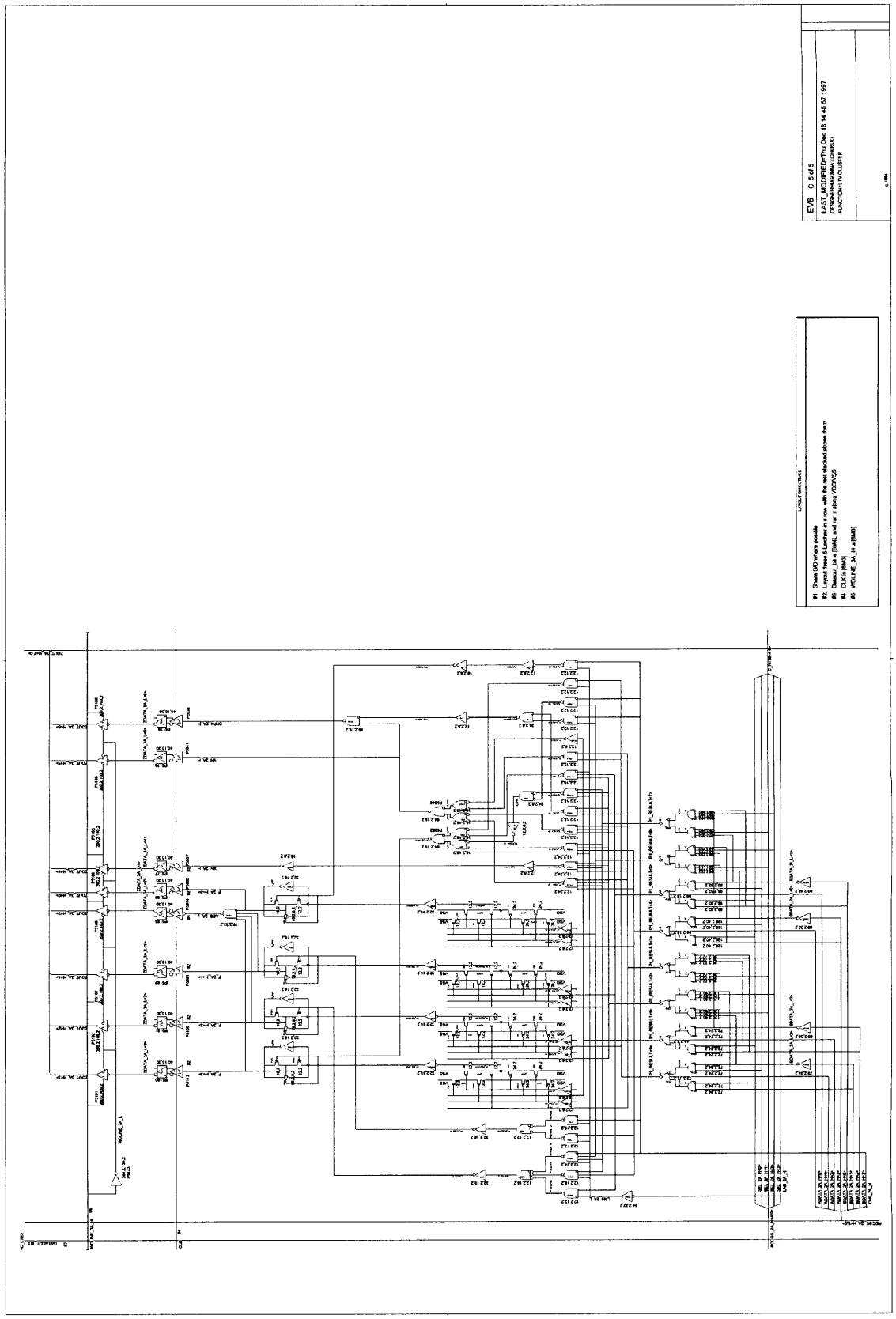

Figure 53 : LTC Schematic, Page 5

Appendix E: LTV PLOT

Logic Test Vehicle Plot

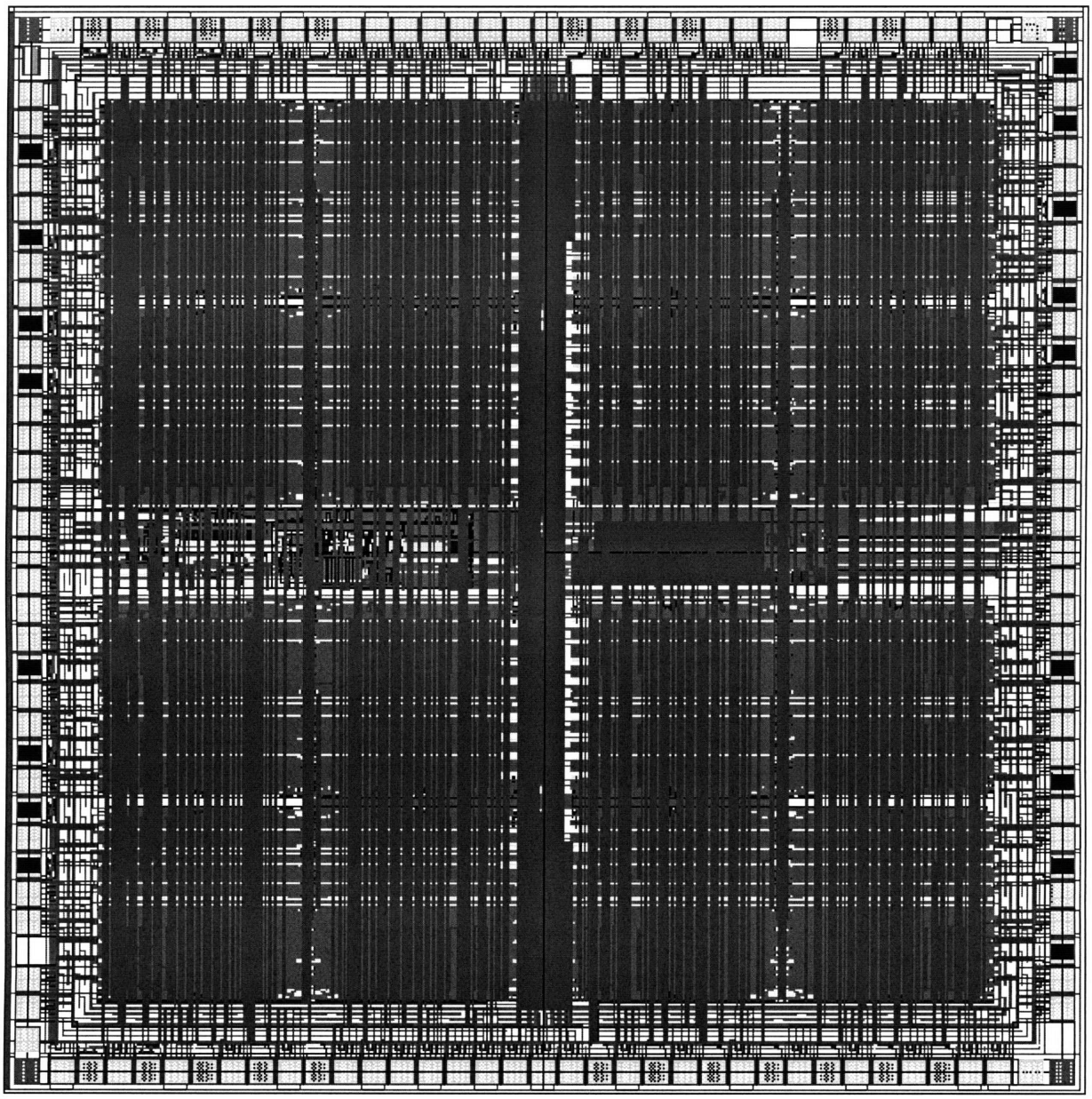

Figure 54 :LTV plot

Bibliography

[1] SEMATECH
The National Technology Roadmap for Semiconductors (NTRS), 1994-1997.

[2] Miron Abramovici, Melvin A. Breuer and Arthur D. Friedman, Digital Systems, Testing and Testable Design. Computer Science Press, New York, 1990.

[3] R.D. Blanton and John P. Hayes, Design of a Fast, Easily Testable ALU.
IEEE Test Symposium, 14th IEEE VLSI test symposium, pages 9-16. 1996.

[4] Jan M. Rabaey, Digital Integrated Circuits: A Design Perspective. Prentice Hall Electronics and VLSI Series, New Jersey, 1996

[5] Eugene R. Hnatek, Integrated Circuit Quality and Reliability, Second Edition. Marcel Dekker, Inc. New York, 1995.

[6] R. Jacob Baker, Harry W. Li and David E. Boyce, CMOS, Circuit Design, Layout, And Simulation. IEEE Press Series on Microelectronic Systems, New York, 1998.

[7] T.W Williams and N.C. Brown, Defect level as a function of fault coverage. IEEE Transactions on Computers, Vol. C-32, No. 12, pages 987-988. 1981.

[8] Dilip Bhavsar and Ugonna Echeruo,
Logic Test Chip Specifications v1.0, 1998.

[9] My Dieu CaoHuy, Design Methodology to Improve the Manufacturability of BiCMOS Embeded SRAM with Built-in Self-Test. MIT M.S. Thesis, 1991.

[10] Frank F. Tsui, LSI/VLSI Testability Design. McGraw-Hill Book Company, New York, 1987.

[11] Anantha Chandrakasan, Basic Analysis & Design of Digital IC Circuits, July 7-9, 1997 at Digital Equipment Corporation.