THE UNIVERSITY *of* EDINBURGH

The Automatic Synthesis of Fault Tolerant and Fault Secure VLSI Systems

Ian Michael Nixon

Ph D

University of Edinburgh

1987

Abstract

This thesis investigates the design of fault tolerant and fault secure (FTFS) systems within the framework of silicon compilation. Automatic design modification is used to introduce FTFS characteristics into a design. A taxonomy of FTFS techniques is introduced and is used to identify a number of features which an "automatic design for FTFS" system should exhibit.

A silicon compilation system, Chip Churn 2 (CC2), has been implemented and has been used to demonstrate the feasibility of automatic design of FTFS systems. The CC2 system provides a design language, simulation facilities and a back-end able to produce CMOS VLSI designs. A number of FTFS design methods have been implemented within the CC2 environment; these methods range from triple modular redundancy to concurrent parity code checking. The FTFS design methods can be applied automatically to general designs in order to realise them as FTFS systems.

A number of example designs are presented; these are used to illustrate the FTFS modification techniques which have been implemented. Area results for CMOS devices are presented; this allows the modification methods to be compared. A number of problems arising from the methods are highlighted and some solutions suggested.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Critical Systems

Integrated circuits have influenced 20th century western civilisation almost as much as the harnessing of steam power affected the same civilisation one and a half centuries earlier. In the 20 years since their development, ICs have found their way into innumerable everyday items and few aspects of modern life have escaped the influence of the ubiquitous "micro-chip". It is now quite common for highly complex electronic systems to be responsible for the lives or livelihoods of large numbers of people. It is clear that such *Critical Systems*, as they will be called here, must be reliable.

There are many examples of critical systems, for instance: control computers for power stations and transport systems, such as air traffic and railways; commercial computers controlling on-line services such as share transactions; avionics systems which help to control aircraft; and medical systems such as pacemakers, patient monitoring equipment and life support machines. In some of these applications the size of the system is of secondary importance to safety, but in others there must be some trade-off between size and reliability.

Though reliability is at a premium in critical systems, there are benefits to be gained from improving the reliability of many types of device. In fact there are commercial benefits to be accrued from improving the reliability of almost all electronic systems, from washing machines to mainframe computers.

As integrated circuits become smaller and the complexity of VLSI devices increases, the trend towards "systems on silicon" accelerates. This is the process whereby complete electronic systems, which may once have occupied several printed circuit boards (PCBs), are realised as single VLSI chips or small chip sets. In applications where size and weight are constrained, critical systems will also be translated to silicon and it is therefore important to study the design of reliable VLSI devices.

1.2 Trust

If a device is to be seen as trustworthy, that is if you are going to ship it in a product or rely on it for your life, it must be tested in some way. Such "confidence" testing can take many forms, from one-off batch inspection to prolonged *in situ* evaluation. All testing is aimed at deciding whether or not a device is working. This thesis seeks to divide testing into two basic types: *static* and *dynamic*. Static tests are those tests which occur once, or infrequently, and do not occur during the normal operation of a device. Dynamic tests can be continuous or sporadic but are concurrent with the operation of a device. The main interest of this thesis is in dynamic testing.

The term "reliability" has both quantitative and qualitative connotations and it is in the latter sense that it is used in this work. It should be noted that reliability is used in a more general sense here to encompass the notion of fault security as well as fault tolerance.

Fault tolerant systems are capable of continuing to operate correctly in the presence of certain faults or types of fault. Fault secure systems behave in such a way that an external agency can observe whether certain types of fault have occurred in the system. This means that either the fault secure system can identify and flag faults, or that it behaves in some recognisably illegal way in the presence of faults. Thus fault tolerant and fault secure (FTFS) systems are reliable in the sense that the former can be relied upon to operate in the

presence of some faults, and the latter can be relied upon to indicate that they are faulty.

Though the study of FTFS systems is not restricted to the area of VLSI device design, it is to this area that this work has been directed. It will be seen later that many of the results of the work can be applied to a wider range of implementation styles.

1.3 VLSI Design

Designing any type of complex electronic system is a difficult task and there are features of designing in silicon which further complicate matters. In particular the "manufacture-debug" cycle is slow and expensive. This severely limits the number of design iterations which are commercially feasible. The testing of VLSI devices can be expensive in time, effort and money. This is because of the difficulty of gaining access to the individual components which make up a VLSI device.

Approaches to the problem of managing the complexity of VLSI design are as old as the problem itself. The simple expedient of employing structured design is now well known and widely used. By using high level languages for behavioural description and abstraction, the VLSI design process can be moved away from the consideration of individual device elements such as gates. The use of CAD tools is now almost universal and it is generally accepted that the design of modern VLSI devices would be impossible without such tools.

The following list indicates a few of the areas in which computers and CAD tools can be of use in the design of VLSI devices.

- Managing the design task. Version and source control etc.
- Synthesis systems such as sticks and mask editors.
- Checking of design and electrical rules etc.

- Design Verification.
- Design Simulation.
- Testing e.g. test pattern generation, testability analysis etc.
- Computationally expensive tasks such as Pattern Generation for mask making.
- Automation, e.g. cell generation, routing, placement etc.
- Abstraction e.g. structured design and high level descriptions.
- Simplification. Reducing the total volume of knowledge required by a designer

As VLSI devices become yet more complex, the sophistication of the CAD tools used in their design must also increase. In some senses the "ultimate" CAD tool is the silicon compiler. The aim of silicon compilation is no less than the complete automation of the design process from specification to implementation.

Though a silicon compiler could be seen as a black box taking design specifications in at one end and producing chips at the other, this view is neither realistic nor desirable in the short term. Current silicon compilers cover a range of the design tasks but typically do not handle those levels of design below the generation of artwork. The question of where certain types of checking are carried out varies from system to system; it is not unusual for design and electrical rule checking to be carried out on mask data for completed designs outwith a silicon compiler.

Most currently available silicon compilers allow some form of manual intervention; the extent and level of this intervention varies. Designer intervention is a two edged sword; many systems could not operate realistically without it, but it is a source of potential error with which a system may not be able to cope. Manual intervention is used either because a designer can carry out a

task better than the automatic system or because a designer "knows more". A typical example of superior knowledge is the selective breaking of geometry rules in leaf cell design.

It was claimed earlier that CAD tools could simplify the design task by reducing the amount of knowledge required by a designer. If a silicon compiler requires no manual intervention and guarantees to produce correct silicon, it can remove the need for a designer to acquire knowledge about some aspects of the design process. Such completely automatic systems can also provide access to VLSI devices for designers with little or no experience of VLSI design.

It is not only the naïve user who can benefit from the inherent knowledge built into a silicon compiler. Experienced designers can be exposed to new ideas, for instance in design for testability (DFT).

1.4 Rationale

The primary object of this study has been the investigation of fault tolerant and fault secure system design within the framework of silicon compilation. It is only within such a framework that a realistic study of automatic methods can be carried out.

If a silicon compiler is to relieve the designer of some of the burden of knowledge, it must itself possess that knowledge which the user lacks. It must also be able to apply that knowledge in the design process. The problems of obtaining knowledge and encapsulating it within a system are aspects of knowledge engineering which are not addressed in this thesis.

To be able to use knowledge not possessed by a designer, a system must either influence the designer, in some sense *educate* him, or change the characteristics of the design autonomously. For a completely automatic system, it must be the design which is changed, rather than the designer.

1.5 Chapter Outline

1.5.1 Chapter1

This introductory chapter has sought to show that the design of reliable VLSI devices is desirable for both critical systems and more mundane applications. It has also been suggested that the silicon compiler represents the most sophisticated approach to the design of VLSI devices. Therefore it has been decided to study the design of FTFS devices within the framework of silicon compilation.

1.5.2 Chapter 2

The next chapter will look at the general area of device testing. The object of this chapter is to identify existing ideas which would be suitable for automatic application. Particular note will be made of some schemes suggested for designing easily testable or self-testing programmable logic arrays (PLAs).

1.5.3 Chapter 3

The work in this chapter is intended to establish a taxonomic framework into which existing and suggested FTFS methods can be placed. Such a taxonomy provides a structure within which to carry out further investigation. It also indicates areas where new methods might most usefully be sought.

1.5.4 Chapter 4

Having identified possible methods of producing FTFS systems, and having decided to implement those systems automatically, it becomes necessary to provide an automatic design environment. Chapter 4 describes Chip Churn 2 (CC2) which is such an environment. The history of CC2 is described, with

particular reference to Chip Churn, its predecessor. The main elements of CC2, the language, the simulator and the back-end will be covered.

1.5.5 Chapter 5

A number of automatic FTFS methods have been implemented within the CC2 environment. In this chapter these methods are described and their implementation is explained. A worked example of a simple controller design is also presented.

1.5.6 Chapter 6

This chapter presents a number of example CC2 designs and looks at how they can be modified automatically to become FTFS. The examples range in complexity from a simple 8 bit adder to a complete microprocessor. The results of applying automatic design modifications are presented.

1.5.7 Chapter 7

The final chapter highlights the important elements of this work. Many of the problems which have been encountered are discussed and ideas for further work are suggested.

Chapter 2

Testing and Reliability

In the introduction, it was stated that testing was an essential "confidence building" exercise in the design and production of any system. This chapter looks at some of the existing methods of carrying out testing. It was established in the introduction that this work is directed towards the implementation of automatic design of FTFS systems. To this end, this chapter is intended to identify those methods which might be applicable in this area. Trends towards autonomous test and automatic design for testability are also of interest and are therefore mentioned.

2.1 Test and Testability

2.1.1 What Causes Faults ?

Broadly speaking there are four types of fault which can occur in VLSI devices; some of these faults can also occur in other types of system. The first type of fault is the design fault. Design faults can arise from errors in the design specification or faulty implementation; this can make them very difficult to identify. There should be no design faults in production devices but sometimes there are. The second type of fault is the fabrication fault. Fabrication faults occur during manufacture and can result from such things as badly aligned

or damaged masks. Ideally all fabrication faults would be identified by the manufacturer but again not all of them are. The third class of fault is caused by fatigue failure; for example, PCB failure due to vibration or damage of a VLSI device by metal migration. A manufacturer cannot test for fatigue faults as they are not present when a system is "shipped". However, the manufacturer should have designed and tested the system so that fatigue faults are unlikely in the expected working environment. The final class of faults are transient faults. These are faults which cause no permanent damage to a system. Transient faults can be the most difficult to find and can be the result of faulty design, such as an unstable reset line, or short lived environmental effects such as radioactive decay or electro-magnetic pulse (EMP).

The testing of devices, which will be discussed next, is intended to identify the first three classes of fault. These permanent faults should always be identifiable in a system once they have occurred. There is no sure way of testing for transient faults other than by monitoring the actual operation of a device.

2.1.2 The Problem.

All testing is designed to answer the question, "Does the device under test (DUT) operate as expected ?". As the complexity of the DUT increases, it becomes more difficult to give a categorical response to this question. In simple devices, exhaustive testing can be employed. In an exhaustive test every possible input pattern is supplied to the DUT and its outputs are compared with the expected outputs. In sequential logic there is the added problem of having to stimulate the DUT in every possible state. With complex devices containing sequential logic, it rapidly becomes impractical to rely on exhaustive testing.

Exhaustive testing effectively answers the question, "Does it work ?". Beyond exhaustive testing, it becomes more common only to answer the question, "Can any faults be found ?", the assumption being that it is simpler, or more practical, to find faults than it is to prove correctness. In order to be able to say whether any faults can be found, it is first necessary to decide which faults

are likely to occur and therefore which faults will be sought. In other words it is necessary to adopt a fault model.

Strictly speaking a fault model is not essential and random testing could be carried out without one. Random testing [2] means applying randomly selected input patterns to the DUT and comparing the outputs with the expected responses. However, without a notion of what faults to expect, it is difficult to gauge how effective random testing will be for a given DUT.

To identify faults in a system it must be possible to stimulate a DUT in such a way that its response in the presence of a fault differs from its fault free response. If such a stimulation is impossible, the fault is undetectable. In effect the fault is not a fault at all, in that the DUT performs identically with or without it. This may seem paradoxical, but an example can be found in certain types of programmable logic array (PLA) where the presence or absence of a contact has no effect on the function realised by the PLA.

Once the simple approaches to testing become inadequate, two characteristics of a device become important. Those characteristics are *controllability* and *observability*. These are, respectively, the ability to control what is going on, and the ability to observe what is going on. The idea of controllability goes beyond a simple ability to control the inputs to a functional unit; in the case of sequential logic it is necessary to be able to control the internal state of a unit.

Controllability and observability are essential in fault identification testing because of the need to stimulate identified parts of a device. Take for example a DUT in which a fault model allows for a particular gate to have its output value stuck at 0. To test whether that fault exists, it is necessary to provide an input pattern to the gate which would, under fault free operation, result in a 1 being output. In other words, it is necessary to *control* the inputs to the gate and *observe* the outputs from it. If the gate is buried in the middle of a piece of complex logic, it will be necessary to guide the required input signal through other gates and to observe the output as an effect on other gates.

Controllability and observability are inherent characteristics of a device and

so it is essential that the *testability* of a device is considered during the design process. The need to consider the testability of systems during their design has given rise to the ideas which are embodied in the principles of *design for testability* (DFT).

2.1.3 Verification versus Confidence

It is possible to identify two separate reasons for testing. The first is *verification* testing which is used to check that a design has been implemented correctly. The second is *confidence* testing which is used to check that a particular instance of an implementation does not contain any faults. Confidence testing can be carried out once only, for example at the time of manufacture, or sporadically, for instance whenever a test program is run.

As the need for verification testing is transient, verification test measures might, in theory, be absent from a production device. In practice it is seldom the case that verification and confidence testing are separated. This is because of the cost of design effort and fabrication. If the testability features of a design are to be discarded, so is the design effort that went into them. Also the cost of fabricating the prototypes must be written off; this is because the production devices will not be exactly the same as the development devices. The need for verification testing of individual functional units should decline as the use of verified cell generators and the like becomes more widespread.

2.1.4 Design For Testability

There are a number of papers, books and tutorial guides such as [56] and [4], which give a general introduction to this area. What follows is a brief outline of the main methods suggested for use in DFT.

Scan/Set Logic

The simplest approach to improving the controllability and observability of a system is to add test points into the design. Ideally test points would be connected directly to the outside world via input/output pads. However, in most designs the large number of test points would make it impractical to connect directly to pads. In the Scan/Set logic [56] scheme, the test points are connected to a shift register on the device itself; this allows test data to be shifted in and out serially; thus reducing the number of test pads required.

Scan Path Logic and LSSD

The problem of controlling the internal states of sequential logic has already been mentioned. The problem can be reduced to one of controlling that part of the sequential logic which constitutes its memory. If this memory cannot be directly controlled, it is necessary to develop such measures as "homing sequences". A homing sequence is a procedure whereby a piece of sequential logic can be taken from any state to a known state. Clearly there are simpler approaches to the problem, such as incorporating a reset signal which takes the machine directly to a known state. However, the best possible control which can be obtained is the ability to set the memory elements directly; this allows control of the state of the sequential logic.

Scan Path Logic [15] and Level Sensitive Scan Design (LSSD) [12] approach the problem of controllability and observability in essentially the same way. All the latches used in a design are of a special type which can be configured to work in one of two modes. In the first mode, the latch behaves ordinarily, but in the second, it can be used as an element in a shift register. The configuration of the latch can be carried out dynamically by the use of control signals.

If a design is partitioned in such a way that there are latches at the inputs and outputs of all the main functional blocks, then the shift register composed of all the latches in the device allows the inputs to a block to be controlled,

and its outputs observed. In addition, if the latches are used as the memory elements of any sequential logic, the state of that logic can be controlled; this reduces the testability problem to one of testing combinatorial logic.

Signature Analysis

One problem with testing can be the volume of input and output data. To reduce the amount of output data that has to be processed, a number of data compression techniques have been developed.

Data compression methods can be seen as mapping between a set of possible outputs and a smaller set of check patterns. Because the set of check patterns is smaller than the output set, information is lost during the mapping process. In effect, each check pattern has more than one output pattern mapped into it; this makes it possible for a faulty output to map to the same check pattern as its fault free equivalent. Faulty outputs which are mapped to their fault free equivalents will not be detected.

Perhaps the simplest form of data compression is transition counting [23]. This involves keeping a running total of the number of times that the output stream switches between logic states.

Signature Analysis [24] was developed as a data compression technique with particular application in the testing of PCB systems. The compression principle is based on the use of a linear feedback shift register (LFSR). Such shift registers can be used as pseudo-random binary sequence (PRBS) generators. With a suitable selection of feedback taps, such generators can be made to cycle through all possible internal states before repeating themselves. If an output stream is EXORed into a LFSR, the sequence of generated states is altered. So, if during a test the output stream from a DUT is fed into a LFSR, it will produce, at the end of the test, a value in the LFSR which represents a "signature" for the output stream under examination.

Some of the mathematical properties of signature analysis are discussed in [14], where it is also compared with transition counting. There are two main

points noted in this paper, the first is that signature analysis can detect all single bit errors. That is to say, no two output streams which differ in only one bit can produce the same signature. The second result quoted in the paper is that as the length of the output sequence gets larger, the chance of failing to detect a multi-bit error tends towards :

$$\frac{1}{2^n}$$

where n is the number of registers in the LFSR.

Syndrome Testable Design

Another testability method which deserves note is that of syndrome testability [48]. The syndrome of a combinatorial logic function can be obtained by counting the number of 1s present in the output stream produced by an exhaustive test of the function. The advantages of this method are that the test patterns are easily generated, the output method is in itself a data compression technique, and the expected result of the test can be calculated mathematically from the function of the DUT.

2.1.5 Test Patterns and Expected Results

Test pattern generation (TPG) for exhaustive testing is straightforward; the test set being made up of all the possible inputs to a system. When the aim of testing is the identification of faults, TPG is more problematic. The task is to derive a set of inputs for a device which will identify a number of possible faults. We have already seen that a fault model is essential to establish which faults are expected. Once the set of possible faults has been decided upon, the TPG problem is one of deciding how to feed the appropriate input patterns to those areas of a device where faults might occur, and then to guide the results to a point where they can be observed. A number of algorithms such as the D-algorithm [45] and PODEM [19] have been used for TPG.

The expected results of a test can be derived from simulation or from existing versions of a device which are known to contain no faults. Such "known-good" devices are often called *gold units.*

2.1.6 Built In Self-Test (BIST)

The use of design for testability measures is intended to produce designs which are controllable and observable. Even when this aim is achieved, there are still the problems of TPG, the application of the test patterns and the analysis of the results. One drawback of scan based systems is that they work serially. This can slow down the performance of a test, a test being the application of the test patterns and the collection of the output data. It may be possible to execute a number of tests on different blocks in parallel and some work, such as [8], has attempted to speed up the test process by using such *parallel test scheduling.*

As we have seen, the primary steps in the testing process are test pattern generation, test pattern application and test result evaluation. The object of BIST [39],[32] is to move all three of these steps onto the DUT. Most of the current BIST techniques adopt a broadly similar approach to the problem and what follows is an outline of that approach.

It would be impractical to carry out automatic TPG (ATPG) "on-chip" if that ATPG were based upon some sophisticated algorithm. Consequently a return to the use of pseudo-random and exhaustive testing has taken place. As we saw earlier, exhaustive testing is impractical for complex devices so for BIST methods to employ this kind of testing, complex designs must be partitioned into units which can be exhaustively tested in a reasonable amount of time. Not all devices are so complex that they need to be partitioned. The work presented in [42] used special pads capable of test pattern generation and test pattern compaction, in order to carry out autonomous test of complete chips. The particular feature of the devices discussed in [42] which make them testable in this way is that they are serial.

Scan paths can be used to partition a design but they have other uses as well. LFSRs can be used for data compression (signature analysis) and as PRBS generators. So if a device is partitioned in such a way that its inputs come from a set of latches, and its outputs go into a set of latches, it is possible to configure the former to be a PRBS generator and the latter to be a signature analyser. The Built In Logic Block Observer (BILBO) is an example of this type of special register. The PRBS generators in these scan type applications differ from the original signature analysis LFSRs in that they compress multi-bit streams rather than single bit streams.

Once a test has been performed, the result needs to be compared with the expected result. This comparison can either be done off-chip, or can be hardwired into the device.

2.1.7 Automation

Currently, aids to DFT and BIST are more common than completely automatic approaches. TMEAS [21] and CAMELOT [4] are examples of testability measurement programs. These are intended to provide a designer with some idea as to how difficult it will be to test a device. To this end they provide metrics or scores for parts of a design; these measures reflect the controllability and observability of those parts. There are knowledge based approaches to both testability in [1] and BIST in [28] but again, neither are integrated into a complete design environment. However the work described in [16] and [3] does attempt to automate the process of DFT and BIST within a design environment.

2.1.8 Programmable Logic Arrays (PLAs)

The PLA is possibly the most widely used and widely studied circuit idiom currently available to the VLSI designer. Because PLAs have a very regular structure and are straightforward to generate automatically, they have become

popular for realising arbitrary logic such as datapath controllers. The regular structure of the PLA and its widespread use have led to the development of testability aids directed specifically at the PLA.

In [9] Cha presented a fault model for PLAs and a TPG method for the model[1]. The design of testable PLAs has been studied in [30] and elsewhere. In other work such as [17] and [18] styles of PLA design which have function independent tests have been suggested. These projects aim to produce PLAs which can be tested by universal test sets, thereby reducing the TPG problem. The main technique employed in these papers is the addition of logic which allows individual bit and product lines to be stimulated. The product lines are controlled by a shift register and simple select lines are used to control the bit lines; this allows step by step checking for cross point faults.

Self-testing of PLAs has also been studied. In [11] the AND and OR planes of a PLA are partitioned and each is tested by use of BILBO registers. The work in [20] uses the idea of function independent test sets to implement self-test. The PLAs are designed so that they can be tested with a very simple universal test set; this test set can be generated by extra logic in the PLA.

Possibly the most interesting work from the FTFS point of view is that presented in [31]. This paper presents a collection of methods which can be used to create PLAs which carry out concurrent error detection. The main limitation of the work presented is that it claims only to work on non-concurrent PLAs. Those are PLAs in which any input pattern selects only one product term. This is a severe limitation as a large percentage of minimised PLAs will be concurrent. However methods are presented in the paper which can be applied to concurrent PLAs, in particular the use of two-rail code checkers to check the input buffer lines. Two-rail codes will be mentioned later, but the point noted in the paper is that the input buffers of a PLA always produce the input signal

[1] A version of this TPG method was implemented for the Chip Churn system

and its inverse. Therefore if the buffer lines ever had the same value, an error would have occurred.

2.1.9 Limitations of Static Test Methods.

So far this chapter has shown how the growing complexity of devices has led to a need for more sophisticated testing methods. Yet even the most advanced BIST techniques are static in the sense that they perform testing outwith the normal operation of a device. This means that they can only be used to check a device at some point before normal operation begins. In systems which do not need to operate continuously, BIST methods could be used during idle cycles. Though this would provide a more consistent test coverage, it would still not indicate that a device was operating correctly.

2.2 Reliability

In the introduction to this thesis, we saw that there was a growing need to study the design of reliable VLSI devices. However, the need for reliable electronic systems is not new and a large body of work relating to reliability already exists. Not all of the methods popular with the implementors of reliable systems are applicable to VLSI devices. For instance on-line maintenance would be difficult, if not impossible, to implement at the chip level[2]. What follows is an outline of some of the main aspects of reliable system design as they relate to the development of reliable VLSI devices.

[2]On line maintenance of multi-chip systems is not impossible.

2.2.1 What is required ?

If a device is to be considered trustworthy, it must either be fault tolerant or fault secure. If a device is to be fault tolerant it must be able to exhibit fault-free behaviour in the presence of a fault. To do this it must either have enough duplicate hardware to replace that which is at fault, or it must be able to recreate a fault-free behaviour from a faulty one. A fault secure device must contain some method of differentiating between faulty and fault-free operation, or faulty outputs from it must be easily identified.

Of the four types of fault described in section 2.1.1, FTFS systems should be able to cope with all but design faults and even some of these might be detected. Having said that, reasonable care in the fabrication testing of devices should eliminate all faults except transient ones and those fatigue failures which occur during the normal operation of a device. Confidence or fabrication testing of FTFS devices can make use of the concurrent tests being carried out by the devices. So FTFS devices can be tested by use of random or selective system excitations.

It is fair to say that it would be impossible to make a completely FTFS VLSI device. This is because there are certain types of catastrophic failure against which it would be difficult to guard; for instance power failure or static discharge which resulted in device destruction. Though they may seem extreme, these types of disastrous faults are in some ways more difficult to avoid in VLSI devices than in larger systems. The size of VLSI systems makes duplication of such things as power supplies more expensive than for larger systems.

It should also be noted that certain types of technology are susceptible to specific failures, for instance CMOS and "latch-up". If a device is to be placed in an environment which is hostile to a particular technology, it makes no sense to use that technology in the implementation of the device. A change to a less vulnerable technology is likely to be more valuable than an investment in FTFS techniques for the susceptible technology.

A common feature of most FTFS systems is that they should be able to identify if a fault has occurred and, in the case of fault tolerant systems, take some corrective action. Not all fault tolerant systems are fault secure as certain fault tolerant techniques do not rely on fault identification. In the following sections a number of FTFS methods will be outlined.

2.2.2 Hardware Redundancy

Hardware redundancy is a major technique used in the design of reliable systems. The idea of on-line maintenance mentioned earlier is based on an ability to remove parts of a system without affecting its operation. This allows faulty units to be replaced without having to stop the whole system. To allow this kind of flexibility, it is necessary to have at least two working copies of any module which might have to be replaced. Still, the results of such efforts can be remarkable. The Bell ESS telephone exchange cited in [35] was designed for a maximum of 2 hours "down-time" in forty years, and achieved that performance.

The classic example of hardware duplication is Triple Modular Redundancy (TMR); a specific instance of the more general N Modular Redundancy (NMR). The basic principle of NMR systems is to have n copies of each function and to "vote" on the result. An NMR system can tolerate m faults where

$$m \leq \frac{n}{2} - 1$$

This is because there must always be a majority decision on the fault free output. Typically in NMR systems n is odd thus avoiding the possibility of a tied vote. An NMR system in which $n = 2$ would be fault secure to single faults.

It should be noted that modular redundancy carries a reliability overhead in that there are n times as many components that might fail. So, for instance, the 2MR fault secure system would have a mean time to failure (MTF) of about half that of the equivalent simplex system. There is also a question of faults in

the voting mechanism - the precautions of having duplicate hardware will come to naught if there is a single voter and it fails.

There are many flavours of hardware redundancy systems and not all of them keep duplicate hardware working in parallel. It is possible to keep redundant modules as "spares" to be switched in if the main unit fails. Systems which adopt this approach rely on error detection mechanisms to identify when a unit has failed and should be switched out. Voting is, conceptually at least, the simplest form of error detection but in the absence of operating duplicate hardware, other mechanisms have to be adopted. Some of these mechanisms will be discussed in the following section.

2.2.3 Information Redundancy

Probably the most widely used example of information redundancy is the error detecting/correcting code. These codes have been used extensively to cope with data transmission through a noisy communications channel. As their name suggests, these codes are designed to deal with errors rather than faults, i.e. they deal with the effect of a fault rather than the fault itself.

Error detecting/correcting codes are not new to the field of VLSI devices. Such codes have been widely used to improve the reliability of memory devices such as RAMs. The simplest error detection code is probably the single parity bit. Effectively the modulo 2 sum of the bits in a code word[3], single bit parity is able to detect all odd bit errors which might occur in a code word.

2.2.4 Codes and Coding

Before going on to talk about specific codes, it might be worth looking at codes in general.

[3]This is for even parity, odd parity is the inverse of this.

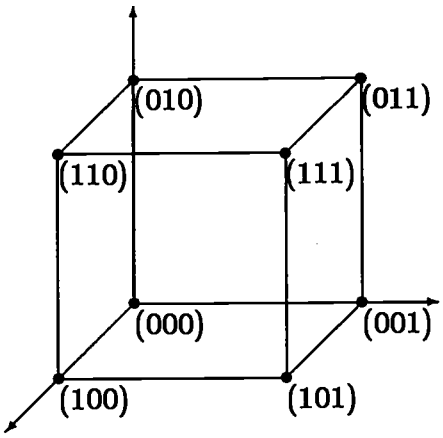


Figure 2–1: Codes words as Cube Vertices

Background

Any sequence of binary digits can be viewed as a code word. For a sequence of m bits there are 2^m possible bit patterns and so the same number of code words. An n bit code can be seen as a set of co-ordinates for a vertex of an n-dimensional unit cube. Figure 2–1 illustrates this idea for a 3 bit code. Any error in the transmission of a code word will change the vertex to which the code refers. An m bit error will move a vertex by m edges, so if two code words map to vertices m edges apart, such an error might transform one into the other. The minimum number of edges between two vertices represents the number of bits by which the two corresponding code words differ. This value is known as the *Hamming Distance.*

The basic principle of error detection coding is to choose a set of code words which are greater than a certain distance apart. For instance if a code is to be able to detect all single bit errors, none of the code words can represent adjacent vertices. Choosing a subset of the vertices of the n-cube creates a complimentary set of vertices which represent invalid code words.

Error correction codes work on the principle of chosing vertices (code words) so that no error could move a valid vertex closer to another valid vertex than it was to the original. In other words, if n bit error correction is required, vertices

should be m edges apart where

$$m \geq 2n + 1$$

In this way any error of n bits or less will leave the resulting code word closer to its original vertex than it is to any other valid code vertex.

Other forms of error detection, such as those designed to detect specific types of error, work by selecting vertices so that any error of the given type will transform a valid vertex to an invalid one. For instance parity codes can detect all odd bit errors because they represent only even (or only odd) vertices and it is impossible to find a path with an odd number of edges between two such vertices.

Separable and Non-Separable Codes

In general, information which has to be coded is already itself a binary code; in this situation there are two approaches to the encoding process. The first approach adds information to the existing code, for instance a parity bit. The added information increases the Hamming distance of the original code to give it error detection/correction properties. The alternative approach is to map the existing code words into a new set of code words which may not have any bit patterns in common with the original set. The first of these approaches yields *separable* codes, so called because the check bits are separable from the information bits. The second method produces *non-separable* codes. Once a separable code has been checked, it can be translated back to the source code simply by stripping off the check bits. Non-separable codes have to be mapped back into the source code. This means that the checker/translator for a non-separable code must contain knowledge of the information being transmitted whereas that for a separable code need only know about the coding method used.

To illustrate this point consider the transmission of the information {(00),(10), (01),(11)}. If coded using an even parity bit the transmission code would

be {(000),(101),(011),(110)}. All that the checker/translator would need to know was that the last bit of the message should be the modulo 2 sum of the other bits, and that the actual message was all the bits but the last one. On the other hand, if a 1-of-4 code was used, the transmission code would be {(1000),(0100),(0010),(0001)}. In this case the checker/translator would need to know that it should only expect one 1 in any message. It would also have to know how to map the received code word back into the information bits.

Because non-separable codes require information about what is being sent, they are seldom used in transmission applications. They are usually used in systems where code translation is unnecessary, for instance for internal bus communication on VLSI devices.

Parity

In its simplest form, parity coding uses vertices a minimum of 2 edges apart. This allows the detection of all odd bit errors. The appeal of parity coding is its extreme simplicity and low overhead. Any set of code words with a Hamming distance of one can have that distance extended to two by the addition of a single extra parity bit.

The main drawback of parity coding is the low error coverage it provides. If the number of erroneous bits in a code word were random, parity would detect only about half of the corrupted transmissions. It is possible to improve the error coverage of parity codes by adopting group parity. In this scheme collections of bits in a source code are each given their own check bit. This increases the overhead of the code but also its error detection ability.

Hamming Codes

In order to be able to correct single bit errors a code must have a Hamming distance of at least three. Though it is relatively straightforward to generate non-separable codes with this property, some algorithm must be sought to generate suitable separable codes. Hamming codes [22] use just such an algorithm.

In a Hamming code the check bits occupy those bit positions which are a power of two, i.e. 1,2,4,8,16, etc. Each of these check bits is a parity bit for a collection of other bits, including check bits, in the code word. Though this may seem unnecessarily complicated, the checking procedure for the code makes error location very simple. The check procedure works by building up a second binary number by examining the check bits in the transmitted code. If a check bit is correct, a 0 is placed in this secondary number whereas an error causes a 1 to be used, these bits are arranged from right to left and eventually yield an n bit binary number where n is the number of check bits in the code. When this number is finished, it represents the binary value of the bit position which is in error. If the value is zero, there is no error.

Residue Codes

Residue codes are a class of separable codes in which the check bits are composed of the modulo n remainder of the number represented by the source code word. Residue codes have attracted particular attention because if n is chosen such that

$$n = 2^l - 1 \quad \text{where } l \geq 2$$

then a checker for the code can be implemented by a tree of -bit adders with end-around carry. Residue codes with this property are called *low-cost residue codes* and the typical value of n is 3.

Berger Codes

Berger codes are separable codes in which the check bits are the inverse of the number of 1s in the source code word. For example, the check bits for the code word (1000101) would be the inverse of (011) that is (100). Berger codes have the property that they can detect all unidirectional errors in a code word, that is errors which only change 1s to 0s or 0s to 1s but not both. The calculation of Berger codes in straightforward.

N-of-M Codes

N-of-M codes are a class of non-separable codes in which the code word length is m and the number of 1s in a code word is always n. N-of-M codes are a particular instance of *unordered codes.* These are codes in which no code word contains a set of ones which are a subset of the ones in another code word. This gives them the property that they can detect all unidirectional errors.

Serial and Parallel Codes

Many codes, particularly those inherited from the field of data communications, are easily checked by serial mechanisms. Little, if anything, is ever said about the difficulty and expense of code checking in parallel. Checking with trees of checkers such as EXOR gates for parity, or adders for low-cost residue codes, is not truly parallel and incurs a heavy cost for long code words.

Omissions

Certain types of codes have been omitted from this discussion because they do not map easily into the sorts of applications envisaged here. Most notable among these omissions is the class of *block codes.* For this type of code check bits are calculated for blocks of code words, something not often required within VLSI devices.

Another class of codes which have not been explicitly discussed are those codes which preserve their check bit integrity under simple arithmetic operations. These types of code can be used in arithmetic chips but their use was considered too limited for application to more general systems. This does not mean that no codes of this type have been considered, rather that they were not considered for this property alone.

2.2.5 Totally Self-Checking Checkers

The aim of totally self-checking logic is to produce outputs which can easily be identified as correct or faulty. In [35] Lala devotes an entire section to the design of TSCCs for different types of codes. Totally self-checking checkers (TSCCs) produce outputs in the form of two-rail or 1-of-2 codes; thus their only valid outputs are 10 and 01. This makes them able to detect stuck-at faults. The simplest form of TSCC checks 1-of-2 codes; thus allowing TSCCs to be used on the product of TSCCs. The following logic equations describe just such a TSCC :

$$c_0 = (x_0.y_1) + (y_0.x_1)$$

$$c_1 = (x_0.x_1) + (y_0.y_1)$$

Here c_0 and c_1 are the two-rail code and (x_0,x_1) and (y_0,y_1) are the two input codes. This checker is totally self-checking for all unidirectional faults.

2.2.6 Systemic Approaches

Hardware redundancy and localised checking can be applied at the level of individual function blocks in a design. An alternative approach to designing FTFS devices is to work at the systems level where architectural choices can be made. The design of FTFS systems for particular types of architecture, such as systolic arrays, has been studied; as has the design of fault tolerant microprocessors, as in [13]. Unfortunately these methods represent an ad hoc approach to the problem which it would be difficult to incorporate in a general automatic system. For this reason these methods are not considered here.

2.2.7 General Approaches

Though there is no lack of material which relates to the systemic design of FTFS systems, and particular techniques also abound, little work has been carried out on the automatic implementation of FTFS techniques. This does not mean that

methods suitable for automatic application have not been suggested as such. In [49] and [51] the application of low-cost residue codes is suggested as just such an automatic method, and in [50] a manual implementation of the ideas is presented. We have already seen that low cost residue codes are attractive because of the straightforward checking logic they require; another advantage is the predetermined nature of the check bits.

2.3 Conclusions

We have seen that redundant hardware can be used to duplicate function in order to provide FTFS systems and that redundant information in the form of coding can be used to detect and correct errors. How then can these techniques be applied automatically to implement FTFS systems? The next chapter presents a taxonomy for automatic techniques and suggests some examples of FTFS techniques which might be applied automatically.

Chapter 3

Form versus Function

3.1 Introduction

It has already been established that the aim of this thesis is to study the automatic design of FTFS systems. Design modification has been identified as a mechanism by which FTFS features can be introduced into a design. This chapter is intended to establish a framework in which to study FTFS system design methods; to this end a classification of design modification techniques is presented.

Three general terms will now be defined; though some people might disagree with these definitions, they are made at this point to clarify their use in the following discussion. These definitions are :

Behaviour The behaviour of a system is the way in which it responds to stimulation. In simple IO terms, it is the pattern of outputs produced by inputs to the system. For instance the behaviour of a five input AND gate is that it produces a 1 at the output if and only if all of its inputs are 1.

Function The function of a system is the method by which it achieves its behaviour. So the behaviour of a five input AND gate might be achieved by a single complex gate or as a cascade of two input AND gates.

Structure The structure of a system is the way in which the function is implemented.

The term *design modification* is used here to describe a process of changing a design specification so as to modify the characteristics of the design without modifying its behaviour. It should be differentiated from the more widely used term *design transformation.* Design transformation takes a design specification between description levels, for instance in transforming a structural specification to an artwork layout.

This chapter suggests that there are essentially two types of design modification. These are *structural* modifications, which change the form or structure, and *functional* modifications which change the function of a design. Hence a functional design modification takes a functional design description and produces another, different functional description. Similarly structural modification effects changes in the structural description or at the structural level of a design. Though a functional modification may *imply* a change in the structure of a design, it is not at the structural level that the change originates.

For a simple example of modification, we move away from the field of FTFS systems for a moment. Consider a design consisting of a single two input AND gate which must be modified to AND together three signals. The modification could be achieved either by designing a three input AND gate or by using two of the original AND gates cascaded together. The first of these modifications represents a functional change because the design remains one consisting of a single functional unit, whereas the second change is structural because the new design consists of two functional units rather than one.

As another example consider a two-bit ripple carry adder made up of two full adders. These full adders will communicate by means of a ripple carry signal. If the adders were modified to use a two rail code for the carry signal, the behaviour of the design would not be affected but the function and structure would.

3.1.1 Function (or What It Does)

The most attractive feature of functional modification is that its results are independent of any implementation. That is to say that the FTFS characteristics of a modified design will be present in any successful implementation of the design. This feature makes it possible to carry out technology independent functional modification. However, if the modification process is to be independent in this way, it cannot make use of technology dependent information such as technology specific fault models. Nor can it use any special structures which might be particularly attractive in a specific technology. The necessity of a fault model will be discussed later.

3.1.2 Structure (or How It Does It)

Structural modification involves either adding structures which enhance the FTFS characteristics of a design, or changing existing structures to exhibit such features. Because structural modification may be tied to a particular implementation technology, it can take into account fault models and structures specific to that technology[1].

3.1.3 Purity

For simplicity the two preceding sections describe somewhat "pure" implementations of the relevant modification techniques. There is no need for functional modification to be implementation independent just as there is no need for structural modification to be implementation dependent. Having said that, implementation independence is a valuable goal and is worth pursuing at least as far as a study of its feasibility. Because the biggest single drawback of technology independence is that it denies access to specific fault models, the next section looks at the possible alternatives to using such fault models.

[1]For example the EXOR PLA buffers of section 6.10.2

3.2 Fault Models

The concept of fault tolerance and fault security is intimately connected with the basic notion of what constitutes a fault. Therefore it is impossible to consider FTFS design without recourse to some notion of fault or error. Placed in this situation, a general system can adopt a number of approaches. Firstly a fault model which is so general that it could be applied to almost all target technologies could be chosen. Probably the most widely adopted technology independent fault model is the Single Stuck At Fault (SSAF) model. This assumes that any faulty system contains at most one fault and that the fault manifests itself as a single signal being stuck at a given value.

The use of fault models is essentially an exercise in *a priori* reasoning, in that it seeks to predict effect (errors) from cause (faults). An alternative approach is that already taken by error detection/correction schemes. In this approach, the errors are considered rather than the faults. This does not preclude an attempt to characterise the errors but such a characterisation is not a prerequisite of the approach.

If it were possible to formulate a general implementation independent fault model, what advantage would be gained by using it? Primarily, fault models are used to predict what errors or kinds of errors might occur. This allows a system to use those methods best able to detect or correct the relevant types of errors. For instance Lala [35] cites work by Mak which indicates that faults in PLAs will only ever produce uni-directional errors. If this is known, then a code capable of detecting all uni-directional errors (e.g. Berger Codes) could be employed. Against the use of a general fault model is a doubt over whether it would be realistic to apply such a model to any specific implementation technology. If such a model predicted errors which were in fact unlikely, it would be valueless.

It should be noted that a fault model is not, in itself, a FTFS method, rather it can be used as a guide to the types of errors to expect. So if a fault model,

general or otherwise, is not available, what can be used to direct the choice of FTFS techniques? Firstly there are methods which use a notion of fault so general as to require almost no reasoning about errors. NMR techniques fall into this category; the notion encapsulated by them is that a fault has occurred when the outputs from a block do not agree with those of the majority of its functionally equivalent modules. A second approach is to reason generally about errors; for instance it might be decided to treat all errors as being equally likely. Once general decisions about errors have been made, an FTFS method can be chosen which provides the greatest error coverage for an acceptable cost.

This section has illustrated that though a fault model can be a useful tool in deciding on a FTFS method, it is not an indispensable tool. There are methods so general that they require little or no reasoning about errors, and assumptions about the likelihood of errors can be made in the absence of a fault model. Consequently it is not unreasonable to adopt an implementation independent approach to design modification as long as enough care is exercised in the choice of an FTFS scheme.

3.3 Classification

To allow a structured evaluation of FTFS methods, this section presents a classification of such methods. To allow the comparison of FTFS devices with each other and with non-FTFS equivalents, two overhead measures are also suggested.

The first measure is that of *area overhead.* This is the difference in device size (or active area) between a FTFS device and its non-FTFS equivalent. The second measure is *time overhead* which is the difference in speed between equivalent FTFS and non-FTFS devices. In applications where the operating speed of a device is fixed, the time overhead can be used to measure the increase in speed needed for a FTFS device to operate at the same "global" speed as a non-FTFS equivalent.

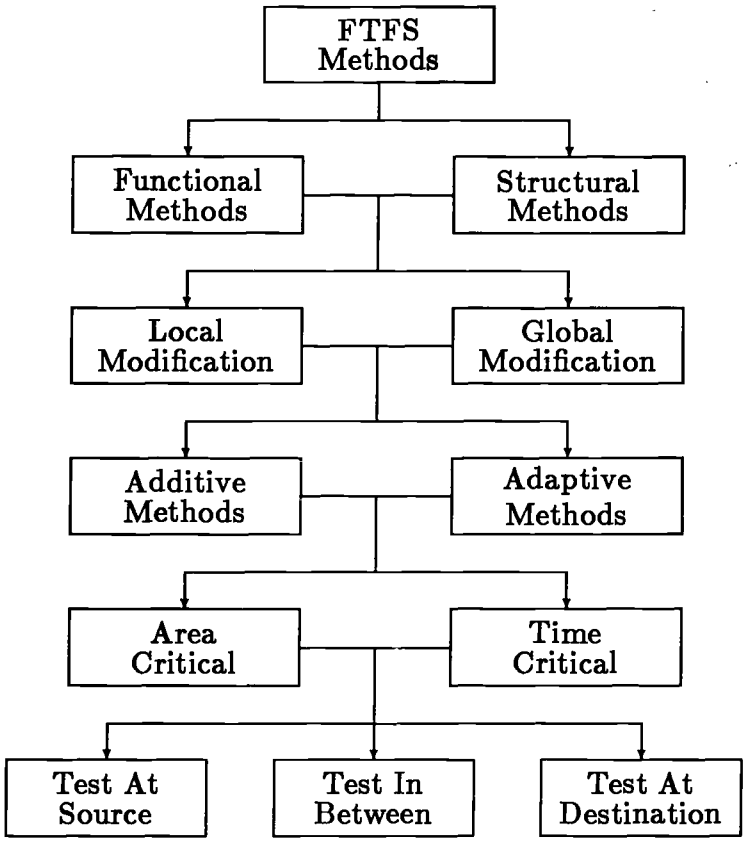


Figure 3–1: A Taxonomy of FTFS Design Methods

The following classification is not rigorous in the sense that a method might belong to two classes in the same level of the taxonomy. Though structured as a number of levels or layers, the classification at one level does not necessarily constrain the classification at a lower level. However, there are certain mutually exclusive classes.

3.3.1 Structure and Function

We have already seen that a design can be modified at either the structural or functional level. This essential dichotomy in modification techniques gives rise to the two primary classes of the taxonomy presented here and outlined in figure 3–1.

3.3.2 Local and Global Modification

The scope of modification changes can be viewed a little like the scope of changes which might be made to a software program. In this analogy, the procedures of the language are replaced by the functional units of a design. *Local* modifications affect only the procedure or block on which they act whereas *global* modifications affect the whole program or design. Local modifications should be *transparent* to all the units which use signals from the modified block.

The analogy goes further in that the advantages of local modification of a program are like those of local modification of a design. Notably, the modification is self-contained and can be used again if the procedure or block is re-used. Global modifications are generally less "portable" and may have to be carried out again even if some of the same blocks or procedures are involved.

The concurrent test structures for PLAs discussed in the last chapter are an example of local modification. Global changes to a design might include re-coding the communication codes on an internal bus, in which case all the blocks using the bus would have to be modified.

3.3.3 Additive and Adaptive Modifications

All the work in this thesis assumes that the starting point of any automatic design approach is some form of specification supplied by a user. This means that when the modification process begins, some form of structural or functional information has already been supplied. Typically this information takes the form of relationships defined between units in the design. For instance unit A connects to unit B by a signal called RESET. Automatic design modification can either add to the existing design units or adapt them. This gives rise to the classes of *additive* and *adaptive* modification.

As an example consider a scheme to use a parity code to check the outputs of a PLA. The generation of the parity bit could be carried out by the PLA, in effect the generation function would be *added* to the PLA. A parity code

checker could then be *added* to the design. Alternatively, it might be decided to *adapt* the function the PLA to use an n-of-m code for its outputs. The essential difference between additive and adaptive methods is that the first leaves the original function or structure essentially intact, whereas the second may completely change either.

3.3.4 Time and Area Critical Methods

At this point redundancy rears its head again, here in relation to time and area. In general, FTFS methods can make use of redundant time or redundant area to achieve their effect, that is they introduce area or time which might not be required by a non-FTFS device. The two classes of *time critical* and *area critical* modifications cover these methods and are the least rigid of those in this taxonomy. The main reason for laxity of these classes is that the same modification method might fall into different classes if applied to different designs. The main object of these classes is to identify the modified feature which is incurring the *critical* overhead. For instance if a chip must perform at high speed, the area overhead of a modification might be less important than the time overhead. Alternatively, in an application for which size was more important than speed, time might be sacrificed to achieve area reductions.

3.3.5 Where to Test

The majority of FTFS methods involve some form of concurrent testing or checking. The last three classes presented are intended to identify the site of that testing or checking. If we consider a system which consists of only three components, a source, a drain and a communication channel, we can see the three possible test sites. In *test at source* (TAS) methods, signal checking is carried out at the signal source. In *test at destination* or *test at drain* (TAD) methods, checking is carried out at the signal drain. In *test in between* (TIB) methods signals are checked "in transit" between source and drain; this means they will be checked either by or on the communications channel.

TAS

The main advantages of TAS methods are: firstly, that signals used by a number of drains need only be checked once; secondly, any check bits generated for separable codes do not need to be communicated to the drain, as their function has been served at the source. Finally, groups of signals which do not necessarily share the same drain can be checked together. The main disadvantage of TAS methods is that they do not allow a system to detect transmission errors; transmission errors being errors which arise during the transmission of a signal.

TAD

The main attraction of TAD methods is that they can detect transmission errors as well as errors originating at the source of the signal. However, TAD methods incur extra communication costs because of the need to transmit check bits. TAD methods can also incur extra check costs if a signal is used by a number of drains; in which case the signal is checked at each drain. There is also an overhead incurred by the fact that check bits can only be calculated on those groups of signals which have a common drain or drains[2].

One advantage of TAD methods is the possibility that checking might be carried out more cheaply at the destination. As an example of this consider the transmission of a 1-of-3 code between two PLAs. The receiving PLA could be coded as in table 3–1 to generate an OK signal for each valid 1-of-3 code. This check would require only an extra output column for the drain PLA whereas a TAS method would require an extra functional block for checking.

[2]It would be possible to calculate check bits for each group of signals which share a drain. This might give multiple error coverage because signals might be in more than one group. There would be an additional overhead for the extra check bits

S1	S2	S3	OK
1	0	0	1
0	1	0	1
0	0	1	1

Table 3–1: "Cheap" code checking

TIB

TIB methods allow for the detection of some transmission errors and can be used to check groups of signals which might not share either source or drain. An example of a TIB method might be a bus "watch-dog" which enforced a particular coding scheme on an internal bus. In general, coding can only be carried out on signals which share a source; as it is only in these cases that the check bits can be calculated in advance.

Having suggested this taxonomic approach to the investigation, the following section looks at a number of FTFS methods; these methods can be used to illustrate the classification which has been presented.

3.4 Examples

3.4.1 NMR

Of the methods introduced in the previous chapter, none is more clearly a form of structural modification than NMR. Where then does it fit into the other classes? First of all it is essentially a local modification. It might be argued that NMR at the board level, such as that required for on-line maintenance, was a global modification, however the size of the duplicated module is irrelevant because the changes made affect only the duplicated module and should be transparent outside it. NMR is an additive method as it need not change the basic function of the duplicated unit. In most applications NMR would be an

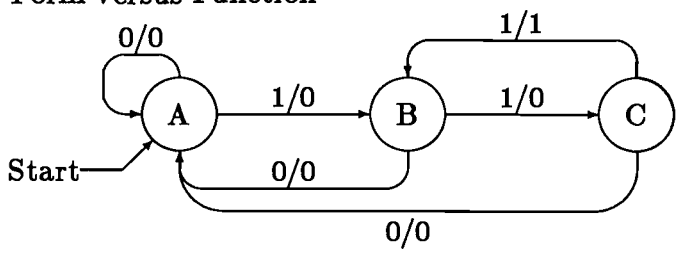


Figure 3–2: A 111 detector

area critical method as the only time overhead incurred is that introduced by the voting unit. NMR systems are primarily TAS systems though duplication of the voting mechanism at a number of drains could make a particular implementation TAD.

3.4.2 State Coding

There is no outstanding example of functional modification suggested in the previous chapter. This is because coding techniques are not in themselves complete FTFS methods, rather they are building blocks for those methods. This being the case, it is necessary to suggest a method in order to demonstrate functional modification. An example from the literature can be found in [47]. In this paper a method of implementing fault tolerant finite state machines (FSMs) is suggested. This method is based on the principle of using state variables coded to have a Hamming distance of 3. This allows the resulting FSM to be made tolerant to any single bit error in the state variable. In effect the correct state transition for each state is coded along with the correct transition for all those invalid code words at a distance one from the correct state. This method is illustrated by table 3–2 and table 3–3 which are alternative codings for the FSM of figure 3–2. In table 3–2 the state variable coding is A = 00, B = 01 and C = 10; in table 3–3 the coding is A = 00000, B = 11100 and C = 00111. It will be noted that this method is only tolerant to single errors in the state signals and if there were a large number of additional outputs, the total fault coverage would be low.

In	State In	State Out	Out
0	xx	00	0
1	00	01	0
1	01	10	0
1	10	10	1

Table 3–2: A simple coding for the FSM of figure 3–2

In	State In	State Out	Out
0	xxxxx	00000	0
1	00000	11100	0
1	10000	11100	0
1	01000	11100	0
1	00100	11100	0
1	00010	11100	0
1	00001	11100	0
1	11100	00111	0
1	01100	00111	0
1	10100	00111	0
1	11000	00111	0
1	11110	00111	0
1	11101	00111	0
1	00111	11100	1
1	10111	11100	1
1	01111	11100	1
1	00011	11100	1
1	00101	11100	1
1	00110	11100	1

Table 3–3: Fault Tolerant coding of FSM in figure 3–2

To implement this technique in an existing FSM would require a functional modification of the machine. This would give rise to a different structure for the FSM but the fault tolerance would be the result of a functional rather than purely structural change. Once again this type of modification would have only local effects and would be transparent to any blocks using signals from the FSM. This method is adaptive but what is not clear is whether it is time or area critical. If the FSM were realised as a PLA, then the change in coding would be likely to increase the size of the PLA, which would in turn affect the speed. This might have design level ramifications if the FSM were already the limiting factor on the speed of the device. Another implementation dependent classification would be that of the site of the checking. If a PLA is viewed as a single unit, then the method is TAS, but if the AND and OR planes are considered separately, then the method is TAD as the actual fault tolerance arises from the new AND plane coding.

3.4.3 Mixing Functional and Structural Modification

So far in this discussion no mention has been made of modification methods which have aspects of both functional and structural modification. This is not because such methods do not exist, but rather to simplify the demonstration of the taxonomy presented. However, it is now appropriate to look at an example of such a *mixed* method. No new example will be introduced, instead the simple parity coding example introduced in section 3.3.3 will be examined again. In this example an "expected" parity bit is generated and compared with the calculated parity bit from an added parity generator. It would be possible to realise this scheme in a structural, additive manner by employing three extra function blocks. Figure 3–3 shows such a modified system where X is the original block. To this has been added a parity generator, a comparator and a parity predictor. The predictor takes the same inputs as the block X but outputs only the parity bit which is to be expected.

The problem with this purely structural approach is the overhead incurred

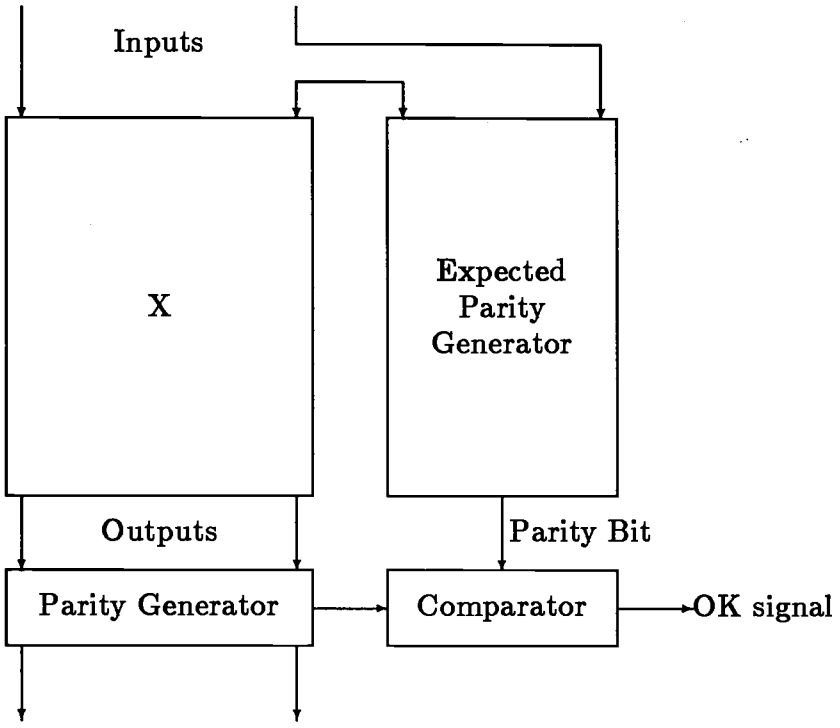


Figure 3–3: A structural modification for parity checking.

by having the parity predictor. If an automatic system had no way of changing the function of the block X, this would be the only way of achieving this particular modification. However, if the system could modify the function of X, it would be able to add the parity prediction function to it, resulting in a system such as that shown in figure 3–4. This modification technique contains a functional element, in the modification of X, and a structural element, in the addition of the parity generator and checker. Whether or not the mixed approach used less area than the purely structural approach would depend on the implementation technology, but if PLAs were being used, the second approach might be expected to be smaller.

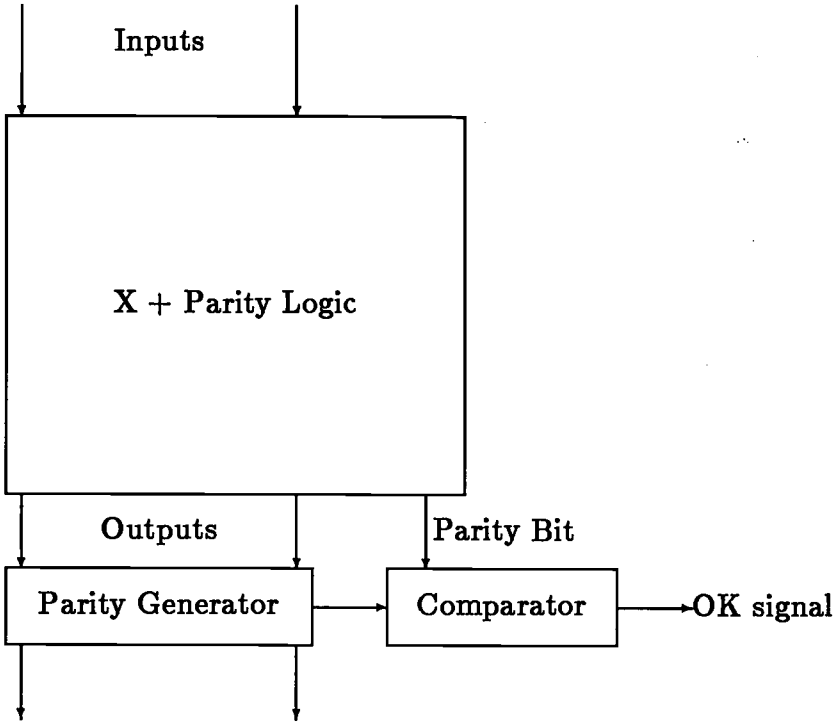


Figure 3–4: A mixed modification for parity checking.

3.5 Conclusions

The classification presented in this chapter is intended to provide a framework in which to study automatic FTFS systems. In the next chapter a design environment in which this work can be carried out is described.

Chapter 4

The Chip Churn Design Tools

4.1 Introduction

So far this thesis has only discussed existing and possible FTFS techniques; if the automatic use of these methods is to be evaluated, an environment in which that evaluation can take place must be established. In the introduction silicon compilation was identified as the "ultimate" automatic VLSI CAD tool and so silicon compilation has been chosen as the framework for this investigation. The work presented in this chapter forms the practical foundation for the evaluation of automatically applied FTFS techniques.

Once silicon compilation has been identified as the required design environment, a suitable system has to be found. At the time this work began there was no complete silicon compiler available to the author which fulfilled the requirements of automatic design modification. Therefore it was necessary to create such a system. In view of the effort available, it was clear that a highly sophisticated silicon compiler was not feasible. For this reason the main system discussed in this chapter was developed to provide the minimum support necessary to carry out the automatic modification of designs. The system also provides the facilities necessary to compare alternative modification methods.

4.2 Basics

In the introductory chapter some of the areas in which CAD tools have been used to help the VLSI designer were pointed out. Before going on to study the Chip Churn design tools, we must establish which design tasks the CAD system is going to tackle; to this end we begin by looking at the basic steps in the design process as outlined in figure 4–1.

Not all of the eight steps suggested in figure 4–1 can currently be automated, and few systems attempt a unified approach to more than three or four of the steps. It has already been said that the design effort for the proposed system was limited, so what is the minimum set of tools that a silicon compiler should provide?

First of all, any automatic design system must provide some mechanism for representing a design. This *design representation* can be anything from a high level language description to a leaf cell net list. Once a system has "captured" a design it should provide the user with a method of checking that the design formalisation conforms to the design specification. Finally a system should provide tools to help implement the specified design. Thus the three components which represent the minimum requirements for an automatic silicon compilation system are :

- A Method of Design Representation.
- Design validation tools.
- Tools for creating implementations, e.g. artwork generation tools.

It might be argued that design validation tools are not essential for the investigation being carried out here. However, some method of comparing designs is required in order to check that a modified design has the same behaviour as its unmodified equivalent. Simulation provides the simplest method for carrying

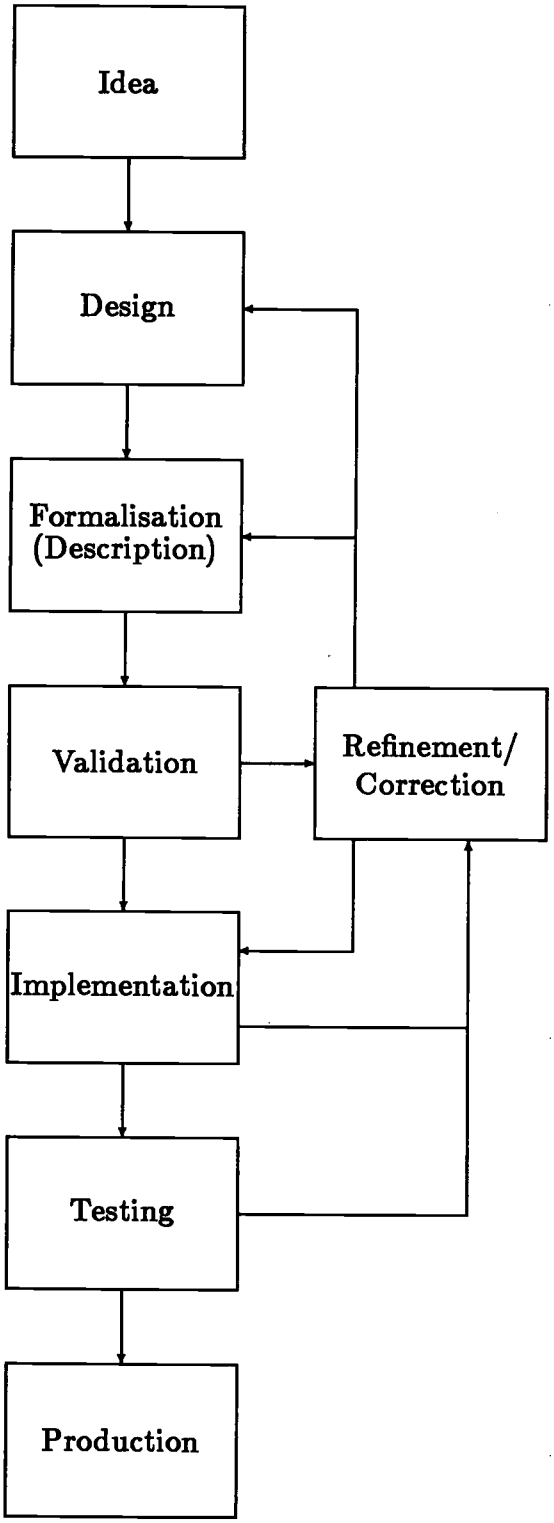


Figure 4–1: Stages in the Design Process

out such a comparison. Having established these three basic components, the following sections will discuss how they are provided by the Chip Churn tools.

4.3 Chip Churn

Before looking in detail at Chip Churn 2 (CC2), it is necessary to say a little about Chip Churn, the system from which CC2 was developed. None of the original Chip Churn code survives in CC2, but many of the ideas used in it were important in the development of the later system. A paper giving more details of Chip Churn can be found in appendix F. What follows are some of the pertinent points covered in the paper.

4.3.1 Design Representation

The Chip Churn design language takes the form of a text description of a number of interconnected blocks. The interconnection between blocks is indicated by signal names and the functions of the blocks are represented by truth tables. The truth tables are kept in files separate from the design description. This simplifies the interface to the existing design tools used for artwork generation in Chip Churn. Figure 4–2 gives an example of a Chip Churn design specification, in this case the design is for an eight bit ripple carry adder. Figure 4–3 shows the truth table which describes the function of the full adder block called `adder`.

4.3.2 Validation Tools

The word *validation* has been used rather than *verification* because the latter has become associated with a number of formal methods aimed at *proving* a design correct. Chip Churn is provided with a simulator and so makes no claims to provide formal proof.

```
Chip_Churn (Adder); {Create adder.cif}
  {All text between curly braces is ignored}
  {These two lines define primary IO signals}
  inputs : reset,a_in [0..7],b_in [0..7],c_in;
  outputs: res [0..7],c_out;

  {This line defines the blocks to be used in the design.}
  blocks : adder (addmin:file;4:in;2:out);

  {This section indicates where each block is used and }
  {how it connects to other blocks.                    }
  connections :
    adder (reset,a_in [0],b_in [0],c_in,res [0],rip [0]),
    adder (reset,a_in [1],b_in [1],rip [0],res [1],rip [1]),
    adder (reset,a_in [2],b_in [2],rip [1],res [2],rip [2]),
    adder (reset,a_in [3],b_in [3],rip [2],res [3],rip [3]),
    adder (reset,a_in [4],b_in [4],rip [3],res [4],rip [4]),
    adder (reset,a_in [5],b_in [5],rip [4],res [5],rip [5]),
    adder (reset,a_in [6],b_in [6],rip [5],res [6],rip [6]),
    adder (reset,a_in [7],b_in [7],rip [6],res [7],c_out);

End_Churn
```

Figure 4–2: A Chip Churn Description

```
IN reset,a,b,cin
     1   x x  x  0  0
     0   0 0  0  0  0
     0   0 1  0  1  0
     0   1 0  0  1  0
     0   1 1  0  0  1
     0   0 0  1  1  0
     0   0 1  1  0  1
     0   1 0  1  0  1
     0   1 1  1  1  1
OUT              out,cout
```

Figure 4–3: A Truth Table File

4.3.3 Artwork Generation

Chip Churn was never intended to support any implementation technology other than NMOS VLSI devices. For this reason no comment will be made at this point about other approaches to creating an implementation.

Though collected under a single heading in the earlier discussion, artwork generation encompasses a number of distinct tasks. The most important of these tasks present some of the fundamentally difficult problems associated with silicon compilation. Module generation, floorplanning, utility routing (power, ground, clocks etc) and signal routing all pose problems which are difficult to solve optimally. However, by adopting a few simple techniques a number of these problems can be alleviated. All of the function blocks in a Chip Churn device are realised as PLAs. This is because the truth tables which make up the functional information are easily translated into PLAs.

The simplest method of automatic floorplanning is to use a *target architecture*, that is to use the same floorplan style for all designs. The target architecture chosen for Chip Churn is based on a central wiring-channel with the PLAs arranged on either side of it. This type of floorplan was used in the FIRST [5] silicon compiler.

Once a channel based architecture has been adopted, floorplanning becomes a matter of deciding which PLAs go in which row, utility routing is straightforward and all the signal routing can be carried out by a channel router. The routing of primary IO signals is simplified by the use of PLAs which have their IO ports on either the top or the bottom.

In the FIRST silicon compiler, function blocks of a similar size were collected together on the same side of the channel in an attempt to reduce chip area. In Chip Churn the placement of PLAs is dependent on the ordering within the language description. This feature allows the designer some control over the final floorplan, and is sometimes necessary because of limitations in the channel router used by Chip Churn.

Chip Churn uses the ILAP [26] design tools to produce a geometric description in the form of CIF [40] for NMOS devices. A number of devices have been designed with Chip Churn and two of these have been fabricated. Both of the fabricated devices performed according to their functional specification. Unfortunately both of the designs had to be submitted for fabrication before the Chip Churn simulator was available. Consequently neither design was validated and both contained functional design errors. In one device the connectivity had been incorrectly specified, and in the other there was an error in the truth table of one function. The first of these errors was catastrophic, in that little could be salvaged from the device. The second error was minor and did not affect the performance of most of the device.

4.3.4 Problems

During the development and use of Chip Churn, a number of problems became apparent. Some of these problems were :

- Language Hierarchy. The Chip Churn design language is not hierarchical.
- Architectural Limitations. Large, complex chips tend to be long and thin.
- Incompatible Design Tools. Too many different kinds of interface between ILAP utilities.
- Data Structure problems, lack of a single central data structure and an inflexibility in the existing data structures.
- Technology Dependence (NMOS).
- Dependence on a particular circuit idiom, i.e. the PLA.

These problems had a major effect on the development of CC2, in fact they prompted that development. The following section describes CC2 in more detail.

4.4 Chip Churn 2

Chip Churn demonstrated that viable silicon compilation tools could be developed using relatively simple techniques. CC2 provides the same three basic components, a design formalism, a validation system and implementation tools, but at a rather more sophisticated level. The following sections describe the CC2 system in more detail. To help identify how the numerous components interact, a schematic overview of the CC2 system is provided in appendix E.

4.4.1 The CC2 Language

There are essentially two types of design description available to a CAD system. The first of these is the schematic or graphical representation and the second is the textual description. It was decided to use a text based language for design capture in CC2 for a number of reasons. Firstly, the description of text languages is well understood and parser generator tools are available. Though general schematic capture systems also exist, none were available to the author at the start of this work. Other important reasons for choosing a textual description are that text can be stored directly on most computer systems, it does not require special graphics hardware, and it can be used as the static storage medium for designs. The last of these points is important because it means that a modified design can be stored as a CC2 language description.

The importance of functional and structural design modification has already been discussed and as the CC2 system is intended to support research in this area, it is essential that it should contain both structural and functional information about a design. However, this basic requirement can be extended. It has already been seen that some of the FTFS methods proposed have both structural and functional components, so it would be advantageous for the CC2 system to keep its structural and functional information in close association.

This is in preference to having functional information transformed into structural information during the implementation or *solidification* process.

The Chip Churn design language is essentially structural but with sufficient reference to function to allow for design simulation. CC2 advances this idea by incorporating the functional information within the language without discarding the essential structural information.

The concepts of structured programming and hierarchical design are now well known. The advantages of the styles were sufficiently desirable that it was thought essential that the CC2 language should provide facilities for the hierarchical description of designs. Though a formalised grammar for the CC2 language exists, it would be as well to explain the main features of the language in a less formal way here.

The CC2 language describes designs as collections of interconnected *blocks*. Two types of block are provided, *function* blocks and *composition* blocks. These are explained in the following sections.

Function Blocks

The function blocks in a CC2 description are intended to correspond to the leaf cells and low level function modules of a design. The CC2 language has no functional primitives such as AND, OR, NOT etc. Instead it uses a function description based on truth tables. This notation was chosen in preference to an equation based representation because it is easier to parse and store. Alternative notations such as those used in the PLAYER system [33] and in [41] are also possible. Both present more verbose descriptions, in the second case based on a programming language. Such linguistically inclined possibilities were rejected in favour of compactness and simplicity. Figure 4–4 shows an example of a simple function block specification. The function of figure 4–4 effectively says :

```
IF a = 0 THEN c := 0
IF b = 0 THEN c := 0
IF a = 1 AND b = 1 THEN c := 1
```

```
function and_gate (inputs: a,b; outputs: c);
  type = async;
  0x -> 0;
  x0 -> 0;
  11 -> 1;
end ;
```

Figure 4–4: A CC2 combinatorial function

```
function R_S_Flip_Flop (inputs: r,s; outputs: q,q_bar);
  type = sync;
  states : been_set,been_reset;
  been_set   : 01 -> been_set   : 10;
  been_set   : 10 -> been_reset : 01;
  been_reset : 10 -> been_reset : 01;
  been_reset : 01 -> been_set   : 10;
end ;
```

Figure 4–5: A CC2 sequential function

with the x's in the description standing for "don't care".

Both synchronous and asynchronous logic can be represented in a CC2 design, and the *type* statement is used to indicate which of these is being used. The timing model used by the CC2 simulator will be described later. As well as combinatorial logic of which figure 4–4 is an example, CC2 also has a notation for sequential logic or FSMs and figure 4–5 shows an example of this.

If design modification is to be carried out, information about the function of individual blocks is essential. However, CC2 can be used as a straightforward silicon compiler when the functional information is used only for simulation and module generation. Chip Churn has a facility which allows customised leaf cells to be included in a design; however, such designs cannot be simulated directly. CC2 is provided with a mechanism for using custom logic, or *library parts* as they are called here. The function of such logic can be expressed in the usual CC2 function notation, or an external program routine can be provided by the

```
function multi_nand <no_in> (inputs: in [1..no_in];
                             outputs: out);
  type = sync;
  [multinandroutine]
end ;
```

Figure 4–6: A parameterised CC2 function

user to be called by the simulator. Because the use of external program routines robs the CC2 system of information it would usually use for module generation, it is always necessary to use library cells for functions specified in this way[1]. Because the library parts of a design will be implementation dependent, they are implemented through use of the *prosaic* statement which will be discussed later.

CC2 function blocks can also be parameterised but when they are, the function specification must be in the form of an external program routine. This is a somewhat arbitrary restriction based on the difficulty of parameterising truth tables. Figure 4–6 gives an example which uses both function parameterisation and an external program routine. If the external routine name contains characters which are not allowed in CC2 names, then the name can be enclosed in quotes (" ") as in figure 4–8.

Composition Blocks

Composition blocks provide a way of collecting together groups of other blocks. Both function and composition blocks can be used in composition blocks. Connectivity within a composition block is indicated by the use of named signals

[1]If external program functions were specified in a language which could be interpreted (e.g. LISP) it might be possible to derive functional information from user supplied code.

```
composition exor_gate (inputs: a,b; outputs: c);
  and (a,inv (b)) -> int_one;
  and (inv (a),b) -> int_two;
  or (int_one,int_two) -> c;
end ;
```

Figure 4–7: A CC2 composition block

analogous to variables in a software programming language. Input and Output signals are defined in the block header but internal signals need not be pre-declared. Figure 4–7 shows a simple combination of function blocks. This example demonstrates that blocks can be instanced as arguments to instances of other blocks. In fact the functional part of the specification could have been expressed as :

```
or (and (a,inv (b)),and (inv (a),b)) -> c;
```

Though the ability to instance blocks as arguments is provided in the language, the feature is implemented at the parser level where flattening of the description takes place. At that stage, internal signals are created to provide the required connectivity. Blocks with multiple outputs can also be used as arguments, in which case the output signals are substituted in order into the instancing block. So the statement

```
block_x (sig [1],block_y (sig [3..4]),sig [2]) -> out;
```

is equivalent to

```
block_y (sig [3],sig [4]) -> sum,carry,check;
block_x (sig [1],sum,carry,check,sig [2]) -> out;
```

and more importantly, it is in this second form that the construct is stored by the CC2 system.

4.4.2 Other Features

There are a number of other features of the CC2 language, some of which are important and some of which are provided as a convenience for the user. Briefly these features are :

Block Ordering

The CC2 parser enforces simple scoping rules so a block can not be instanced until it has been declared. This simplifies some of the error checking at the parser stage and provides a simple mechanism for preventing recursive composition descriptions.

Include

Source text may be included in a CC2 description by using lines of the following type :

```
include "source file name";
```

Included files may not themselves contain include statements. The include statement is not permitted within a block definition.

End Of File

All CC2 source files must end with the statement :

```
End_Of_File.
```

Name Vectors

Signal names may take the form :
`sig [1..5], vals [0..6], bus [1], bus [2], bus [3],...`
Vector bounds must be non-negative integers. This is another feature provided at the parser level where expansion of names is carried out.

Bi-directional Ports

Any signal name which appears in both the input and output list of a block header is treated as a bi-directional signal. Bi-directional signals can be used for either input or output.

State Variable Coding

Figure 4–5 contained the definition of two states, `been_set` and `been_reset`. If the function were realised as a FSM then state variables would be required for each state. If no state variable is defined, CC2 will generate one automatically, but any number of state variables can be defined with missing ones generated by the CC2 system. For instance the statement

```
states : setup,one,found,cycle;
```

would result in the values (00),(01),(10) and (11) being assigned to the respective states, whereas the statement

```
states : setup (100),one,found,cycle (010);
```

would result in the values (100),(001),(011) and (010) being used. Statements of the form

```
states : setup (1),one,found,cycle (0);
states : setup (1000),one,found,cycle (000);
states : setup (100),one,found,cycle (100);
```

are faulted; in the first case because the state variable is not long enough to encode all the states and in the second case because state variables of different lengths are used. The third example is faulted because it uses the same coding for different states; in other words, multiple names for the same state are not allowed.

Many sophisticated techniques have been suggested for the optimal assignment of state variables, but the CC2 allocation is a straightforward number based approach. Integration of a more advanced assignment method would be trivial but the implementation of such a method was not considered sufficiently important for the work undertaken here.

The SET instruction

Because of the notation chosen for FSMs in the CC2 language, a mechanism is provided which allows specific events to send the machine from any state to a known state. This feature is provided by the *set* instruction. This allows the following function fragment

```
one    : 1xxx -> start : 0;
two    : 1xxx -> start : 0;
three  : 1xxx -> start : 0;
four   : 1xxx -> start : 0;
```

to replaced by the single statement[2]

```
set : 1xxx to start : 0;
```

Prosaic

CC2 makes use of a *prosaic* statement something akin to the *pragma* statement in ADA. The prosaic statement can be used to indicate certain types of implementation dependent information to the CC2 system. For instance the CC2 code in figure 4–8 illustrates three uses of the prosaic statement.

The first prosaic statement identifies a file which contains one or more libraries in which leaf cells have been defined. The second prosaic statement identifies a library called `cell lib` in which a leaf cell called `block_x` can be

[2]This assumes that there are only the four named states in the FSM.

```
prosaic inherit "lib.file";
function block_x (inputs: a,b; outputs: out);
  prosaic lib "cell lib";
  prosaic sim "long 5";
  type = async;
  ["block x sim"];
end;
```

Figure 4–8: Use of the Prosaic statement

found. The final statement indicates to the simulation program generator how many state bits the program routine `block x sim` should be given (i.e. 5).

Libraries

Access to libraries through use of the prosaic statement, as demonstrated in the previous section, is a feature implemented not by CC2 but by the current technology specific back-end. These libraries correspond to Elgar libraries which will be discussed in section 4.6.4.

Pad_Up

The pad_up statement identifies the composition block which will be the object of the design. It is for this block that geometry will be generated. It is also this block which will be the subject of design modification.

Comments

All text contained within curly braces, { and }, is treated as comment and is ignored by the parser.

4.4.3 Language Example

To demonstrate a complete CC2 description we shall return to the Chip Churn example of an eight bit ripple carry adder given earlier. Figure 4–9 shows the

source text for such an adder. For the purposes of this example, the adder has been built from a combination of single functional modules (`state_adder`) and composition blocks (`rand_adder`).

A number of features described above are demonstrated by the example. The two files "cmoslib.cc2" and "genlib.cc2" are assumed to contain the definitions of the function blocks used in the composition of `rand_adder`. The definition of `rand_adder` also demonstrates how parameterised functions are instanced, `mnand` being a multiple input nand gate. The number of parameters is specified in the call so that instances of multi-parameter blocks are not ambiguous. For example if a block was defined with the following header

```
function block_x <as,bs> (inputs: sig [1..as],check [bs..0];
                          outputs: out);
```

then in an instance of the form

```
block_x (in [1..10]) -> out;
```

it would not be clear how many of the input signals were `sig`'s and how many were `check`'s.

4.4.4 Technology Independent Intermediate Format

There are advantages to keeping a VLSI CAD system independent of any specific implementation technology; the main one being portability. With this in mind, the CC2 language is parsed into a Technology Independent Intermediate Format (TIIF). The parsing of the language is carried out by a parser automatically generated from a grammar description by APG [37]. APG is a parser generator tool similar to Yacc [27] and Lex [36]. Use of a parser generator greatly simplifies the development of languages by allowing rapid implementation of grammar definitions.

The TIIF data structure reflects the structural and functional information presented in the CC2 language; in fact it retains sufficient information about

```
include "cmoslib.cc2";
include "genlib.cc2";

function state_adder (inputs: a,b,c_in ; outputs: sum,c_out);
  { A simple full adder as a function block }
  type = sync ;
  100 -> 10 ; 010 -> 10 ; 110 -> 01 ; 001 -> 10 ;
  101 -> 01 ; 011 -> 01 ; 111 -> 11 ;
end {of state adder} ;

composition rand_adder (inputs: a,b,c_in ; outputs: sum,c_out);
  { A Random logic full adder }
  exor (exor (a,b),c_in) -> sum ;
  mnand <3> (nand (a,b),nand (a,c_in),nand (b,c_in)) -> c_out ;
end {of rand adder};

composition eight_bit_adder (inputs: a [0..7],b [0..7],c_in ;
                             outputs: out [0..7],c_out);
  { An 8 bit adder made up of half PLAs }
  { and half random logic adders }
  state_adder (a [0],b [0],c_in) -> out [0],rip [1];
  rand_adder  (a [1],b [1],rip [1]) -> out [1],rip [2];
  state_adder (a [2],b [2],rip [2]) -> out [2],rip [3];
  rand_adder  (a [3],b [3],rip [3]) -> out [3],rip [4];
  state_adder (a [4],b [4],rip [4]) -> out [4],rip [5];
  rand_adder  (a [5],b [5],rip [5]) -> out [5],rip [6];
  state_adder (a [6],b [6],rip [6]) -> out [6],rip [7];
  rand_adder  (a [7],b [7],rip [7]) -> out [7],c_out;
end {of eight bit adder};

pad_up eight_bit_adder;

end_of_file.
```

Figure 4–9: A CC2 example

a CC2 design to recreate a CC2 language description. The CC2 language description is the only static storage medium provided for TIIF data.

It has already been mentioned that the TIIF data structure keeps the function information about blocks in the form of truth tables. For composition blocks, instance and connectivity information is represented symbolically, that is blocks and signals are referenced by name rather than pointer. This simplifies design modification but can make some operations slow. As a compromise, instance information also has a pointer field which may or may not be instantiated at any given time.

4.5 Design Validation

The choice of the word *validation* was explained in an earlier section, and it is used again here because CC2 makes no attempt to formally prove the correctness of a design. As with Chip Churn, CC2 provides the user with a simulator to help with design validation. The CC2 simulator is radically different from that used by Chip Churn because the latter is an interactive program, whereas CC2 simulations are carried out by high level language programs generated from the design description. This approach was adopted because it offers a number of practical advantages over the alternative method.

- Because a CC2 user can specify the function of a block by providing an external program routine, it is necessary for the simulator to be able to execute such routines. In the environment in which CC2 was developed, this was impossible without compiling the external routines with the simulation code[3].

- Being an interactive program, the Chip Churn simulator could only be driven by a human operator through a simple command language. As

[3]Use of an interpretable language such as LISP would remove this restriction.

the complexity of designs increased, it became apparent that this level of interaction was inadequate for many designs. The program generation approach allows the user to write code which can be used to drive the simulation.

- Program generation allows for the direct compilation of simulation code; thus speeding up program execution.

4.5.1 Timing Model

It was said earlier that both synchronous and asynchronous logic could be used in CC2 designs. In this section the way in which the simulator treats these two types of logic will be outlined.

The specification of timing and the manipulation of timing information is difficult; as such a general treatment of the problem was considered to be beyond the scope of this work. Therefore the CC2 simulator has a very simple timing model. Synchronous logic is considered to have its inputs latched for a period X and its outputs latched for a period Y, where X and Y are mutually exclusive. For the purposes of simulation, asynchronous logic is treated as if it were able to propagate its results in the period between the start of Y and the start of X. Feedback loops between asynchronous logic blocks are not permitted in current CC2 designs.

Though designs produced automatically by the CC2 system will conform to the timing restrictions outlined above, there is a difficulty associated with enforcing these restrictions on library parts supplied by a user. At present the CC2 system lacks the temporal reasoning capabilities to enforce such timing restrictions, a situation which could only be resolved by the addition of explicit timing information to the CC2 system. Without being able to enforce the restrictions, no guarantee can be made as to the legitimacy of simulation results for designs containing user supplied circuit modules.

4.5.2 Simulation Model

The CC2 simulator operates at the functional level and the functional primitives are the function blocks of the design. The function simulator is based on a PLA simulator and as such it makes certain assumptions about the outputs from a function. In particular it is assumed that the default value for outputs is 0. Only 3 logic levels are supported and those are 1, 0 and U, the last of these being for undefined signals.

The simulator has a notion of a *machine cycle* which is the complete operation of the system clock phases. This means that once in every machine cycle the existing inputs to a block are examined and its results are evaluated.

When simulating sequential logic, the question of where to keep the machine states arises. If the simulation program is a flattened representation of the design, then each block instance has a corresponding routine and the state information can be kept there. There are a number of problems with this flattened approach. First of all, it is difficult to get access to state information if it is being held locally in a routine. Such access may be required during the simulation process to help the user understand or influence what is going on. Secondly, the flattening of a large design can lead to large simulation programs.

One approach to the first of the above problems is to hold all the state data globally, rather than in the individual routines. There are two arguments against this approach. The first is that it does not solve the problem of potentially large simulation programs. The second objection is somewhat pragmatic and arises from a limitation present in the high level language compiler used for CC2 simulation programs; the compiler cannot cope with programs which use large amounts of static storage space.

To solve the program size problem, Genesis, the CC2 simulation generator program, generates a single routine for each block defined in the design description. When these routines are called, they are passed information about the inputs, outputs and state values with which they have to work. Space for signal and state information is allocated dynamically in the form of an array called

the *machine state vector.* Pointers to the machine state vector are kept in a dynamic data structure called a *state tree.* Each composition block is passed a branch of the state tree, with the top level block being passed the root. Each composition block routine contains a number of procedure calls, one for each block instanced in the composition. Each of these calls is passed a branch or leaf of the state tree. The function block routines are each passed a leaf of the state tree. These leaves take the form of *state vectors* which are arrays of pointers to slots in the machine state vector. Pointers are used in preference to array indices because they provide a faster access mechanism. Each state vector has slots allocated for the inputs and the outputs of the function. Figure 4–10 shows how this tree structure works for the simple composition block in figure 4–11. The state tree is in some senses a flattened representation of the design.

In the CC2 simulator there are in fact two machine state vectors, one for the last state of the machine, and one for the new state. The state tree also has two sets of state vectors for each function routine but these have been omitted from figure 4–10 for reasons of clarity.

The difference between synchronous and asynchronous logic is achieved by reading and writing function signals from and to different machine state vectors. In any machine cycle, synchronous logic blocks read their inputs from the last machine state vector and write their outputs into the new machine state vector. On the other hand asynchronous logic does both its reading and writing to the new machine state vector. There are two important consequences of this use of the new machine state vector. Firstly the new and old machine state vectors must contain the same state values at the start of each machine cycle. That is to say, that the results of the last machine cycle simulation must be written into both the new and old machine state vectors before the next machine cycle simulation begins. The second consequence is that the result of simulating asynchronous logic is dependent on the order in which the asynchronous logic blocks are evaluated. Code exists to order asynchronous logic and detect feedbacks in that logic.

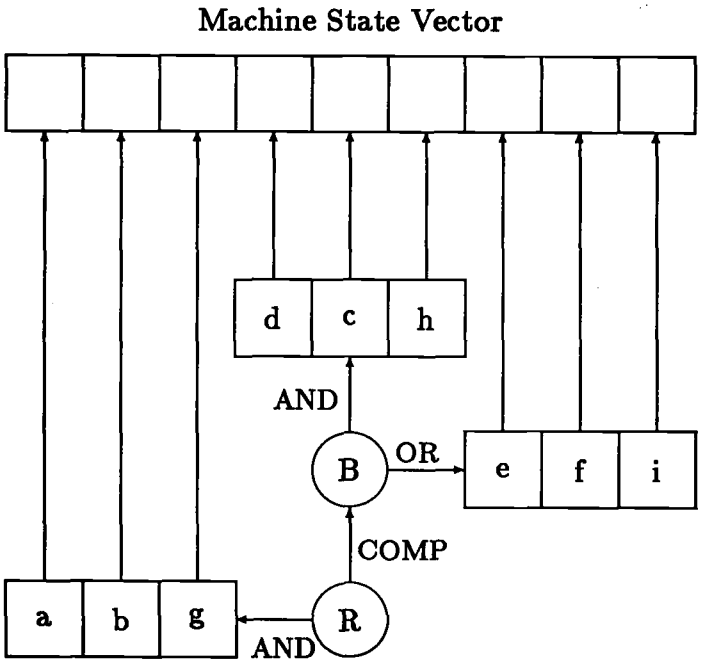


Figure 4–10: State Tree Mappings to Machine State Vector

```
composition comp (inputs: in [1..4]; outputs: out [1..2]);
  and (in [2..1]) -> out [1];
  or (in [3..4]) -> out [2];
end;
composition block_x (inputs: a,b,c,d,e,f;
                     outputs: g,h,i);
  and (a,b) -> g;
  comp (c,d,e,f) -> h,i;
end;
```

Figure 4–11: Simple Composition example

The function simulator uses logical operations on bit patterns, these operations being coded directly into routines. When the function of a block has been specified as an external routine, a call to that routine replaces the machine generated function code.

4.5.3 Simulator Interfaces

It has already been pointed out that the interactive interface to the Chip Churn simulator was not flexible enough to allow user specified programs to drive a simulation. Because the lowest level interface to the CC2 simulator is a set of routine calls, this is no longer a problem. That is not to say that an interactive interface is of no use, and one has been written. However, little effort has been expended on the provision of sophisticated user interfaces because of the ease with which the simulator can be incorporated into a user's own programs.

The basic interface routines provided by the CC2 simulation programs allow for the generation of state trees and for single machine cycle simulation. Code which allows the user to interpret the values in the machine state vector as signals is also provided. All the code necessary to simulate all the blocks in a CC2 design is present in the single simulation program. This removes the need for multiple simulation programs and allows the user to switch attention between blocks in a simulation within the same program.

4.6 Artwork Generation

4.6.1 CC2 Optimisations

There are three CC2 facilities which can be used to help reduce the area used in a design. These are truth table minimisation, function merging and function partitioning.

Table Minimisation

CC2 is provided with a truth table minimisation system based on the Quine-McCluskey method [38]. More details of the specific algorithm used are given in appendix B.

Function Merging

The CC2 merging facility is provided primarily to improve physical layout. Within the target architecture of the current CC2 back-end, a single row in a design is as tall as its tallest member; so if there is one tall block and many short blocks a great deal of space can be wasted. Thus it is desirable that all blocks should be of approximately the same height. This in turn means that the functions in the design should have approximately the the same number of rows in their truth tables.

There are essentially two types of function merging which can be carried out. The simplest merging method is to add two functions together so that the result is effectively two independent functions realised by the same block. This method is straightforward to implement and can be carried out on independent functions; some area saving might also result if the two merged functions share a number of inputs[4]. The second type of merging can only be carried out on two blocks which are logically connected. In the second type of merging, the functions are changed to eliminate the communication between the blocks. If we take the example of the design in figure 4–12, either of the merge methods could be used to merge the blocks a and b. If the first method were used, the block c would result. If the second method were used, the block d would result. In the second case there would be a net reduction in the number of signals being passed round the design.

[4]Outputs should never be directly shared between CC2 blocks as the wired OR construct is discouraged.

```
function a (inputs: i [1..2]; outputs: o);
  type = sync;
  10 -> 1; 01 -> 1;
end;
function b (inputs: i [1..2]; outputs: o);
  type = sync;
  11 -> 1; 00 -> 1;
end;
function c (inputs: i [1..4]; outputs: o [1..2]);
  type = sync;
  10xx -> 10; 01xx -> 10;
  xx11 -> 01; xx00 -> 01;
end;
function d (inputs: i [1..3]; outputs: o);
  type = sync;
  101 -> 1; 011 -> 1;
  110 -> 1; 000 -> 1;
end;
composition des (inputs: i [1..3];outputs: out);
  b (a (i [1..2]),i [3]) -> out;
end;
```

Figure 4–12: Function Merging Example

Only the first type of function merging has currently been implemented in the CC2 system. This is because there are temporal effects on a design when two synchronous logic blocks are merged. It is not possible to merge blocks of differing types (i.e. synchronous and asynchronous) and the merging of asynchronous blocks which communicate could result in feedbacks which should be avoided.

The current CC2 merging system is completely automatic but can be controlled manually in two ways. The system uses a number of parameters to decide whether two functions can be merged. These parameters are derived automatically from the design but can be set by the user. The second manual intervention method allows the user to specify which blocks in the design will be merged.

4.6.2 Function Partitioning

Most of the work which has been published on the subject of PLA partitioning [10], has been aimed at the structural minimisation of the area required to realise a function. The partitioning implemented, so far, in CC2 is intended to help alleviate the logic minimisation problem by breaking up large functions into smaller ones. To this end, a function is broken down into smaller functions, one for each of its outputs. Once a function has been broken up and minimised, it can be merged by the merging software. This system is not ideal but it has proved useful on a number of designs. In theory, the breaking up of a block and its subsequent re-merging might help to redistribute the function in a design. However, it seems unlikely that this automatic functional clustering will be as effective as that introduced by a competent designer.

All three types of optimisation are optional, in the case of minimisation because it can take a long time and in the case of the other two, because they do not necessarily improve design layout. Both partitioning and merging produce radical changes in a design and not all designs which have been minimised can be dumped back to a CC2 language description.

4.6.3 CC2 Back-Ends

Much has been made of the possibilities of technology and implementation independence, and little of what has been explained of CC2 up to now has been dependent on either technology or implementation. However, if modification methods are to be compared, the only realistic comparison which can be made is between implementations of FTFS and non-FTFS devices. For this reason it is necessary to provide the "implementation tools" mentioned in the introduction to this chapter.

The CC2 system has been structured in such a way as to allow any number of technology and implementation dependent back-ends. For the basic CC2 system a back-end must be able to realise arbitrary truth tables as synchronous or asynchronous logic; and be able to compose these elements into complete systems. If the CC2 system were restricted to using library parts, the need to realise arbitrary logic would not exist; however, the types of design modification which could be carried out would also be severely restricted.

Chip Churn demonstrated that by adopting simple approaches to artwork generation, worthwhile automatic VLSI design tools could be created. In the PLA a circuit idiom has been found which can realise the arbitrary functions required by the CC2 system. Unfortunately the PLA is essentially a synchronous device and so a complementary asynchronous idiom had to be found. There were a number of candidates for this role, the two most interesting being the Weinberger array [55] and the CMOS function arrays of [54] and [52]. CMOS was adopted as the technology of choice because of its dominance over NMOS in the VLSI field.

The original Weinberger arrays had been designed to use an NMOS technology so some effort was required to modify the style for CMOS. The final implementation of the CMOS Weinberger array was based on the Domino Logic principle of [34]. The Weinberger DomINO or Wino array architecture is described more fully in appendix C; here a totally self-checking (TSC) implementation

of truth tables is also mentioned. The Wino array was chosen over the CMOS function array because it presented a less complex implementation task.

Having established and implemented two mechanisms for realising arbitrary truth tables for CC2, the problems of floorplanning and routing still remained. Target architectures still present the simplest solution to the first of these problems but the single channel architecture of Chip Churn had proved limiting. A hierarchical use of single channel modules had been suggested for the FIRST system but this presented a more complex wiring problem than was thought approachable in the basic CC2 system. Consequently a multiple channel architecture was adopted for CC2. With this approach there are any number of rows of functions, with a channel between each.

The addition of extra rows and channels introduces the problem of how to arrange routing between blocks which do not share a common channel. This problem is mitigated by the use of blocks which have all their I/O ports on both sides of the cell but feed-throughs still have to be used in some instances. At this level of detail it is necessary to introduce the Elgar system.

4.6.4 Elgar

Many of the problems encountered during the development of Chip Churn arose as a result of the large number of incompatible data structures used by the ILAP tools. Each type of cell generator or router has its own particular format for the specification of port locations, widths, layers etc. To overcome these problems the Elgar system was developed to provide a consistent interface between low level design tools.

Elgar is a composition system, that is to say it is a set of design tools intended to allow the composition of low level leaf cells into functional modules and complete chips. This makes the system ideal for use in module generators and the like, as is demonstrated by the PLA generator used by CC2 which does all its compositions through the Elgar system.

In describing the Elgar system, a bottom-up approach will be taken, that is, we shall look first at the lowest level of geometric design primitives, and work our way up to the highest level operations. We start therefore at the FLAP level.

FLAP

The lowest level of the Elgar system is required to generate artwork information in the form of CIF. The existing ILAP system provided a useful reference point for this level of tools. In fact the FLAP system is currently just a series of routine calls which have ultimate recourse to the ILAP system. The FLAP system provides a set of high level language routines which generate CIF. Routines are provided for the generation of wires, transistors, contacts etc. The two features provided by FLAP but not ILAP are the ability to nest symbol definitions and a feature which allows the generation of an ILAP program in addition to the CIF description. This last feature allows access to other design tools outwith the CC2 system.

Elgar and QV

The creation of leaf cells is a topic in its own right and was not considered within the scope of this work. However, QV [29], a mask level editor, has been interfaced to the Elgar system to allow the design of leaf cells.

Ports and Cells

The Elgar system views all objects as named *cells*. Cells are rectangular blocks with a number of *ports* associated with the sides. Each port identifies a wire which carries a signal into or out of the cell. Each port has a name, a location, a width, a layer, a type and a net. Nets are a mechanism for associating a number of ports which might not have the same name. Connectivity between

cells can be indicated by port net, port name or port location. The sides of a cell are identified by the directions North, South, East and West.

Libraries

Elgar cells can be collected into libraries and any number of libraries can be created. Cells within a library can be accessed by name. Libraries are used by many of the higher level functions as a simple way of passing groups of cells between applications. A typical example of this is where a module generator is passed a library of parts. Different modules can then be generated by the same routine, simply by passing it a different library of cells.

In section 4.4.2 the *prosaic* interface to the Elgar library system was introduced. This allows the user to specify a library cell which can be used to realise a function. To simplify the provision of such library parts by the user, the Elgar system has an interpreter for ILAP and FLAP code. This removes the need for the user to compile the library definitions in with the CC2 code.

Composition

The whole point of the Elgar system is to simplify the job of composing cells together. To this end the system provides a *compose* operator. This takes two cells and returns a new cell which is formed by abutting the first two cells together in either the horizontal or vertical direction.

A problem arises during compositions if the ports on adjacent sides of the two cells do not directly connect. There are a number of possible ways to tackle this problem. The simplest method would be to reject all such compositions as faulty. Though this is straightforward, it would place very strict limits on the types of cells which could be composed. A second, more flexible approach is to perform some form of automatic routing between cells if necessary. Some form of geometric modification such as stretching or shrinking could be carried out on cells; this idea was suggested for the Silver system [46]. This last alternative,

while being very powerful, was beyond the scope of the work presented here. Consequently, the Elgar system uses automatic routing to match the ports of abutting cells.

Three types of composition arise from the routing approach. The first is simple abutment which takes place when port locations, layers etc match. The next method provides river routing between cells where the port connectivity is planar. Finally channel routing can be carried out between sets of ports which have aplanar connectivity. Changes of layer between ports are carried out automatically. The performance of routing between cells can be suppressed by the user.

The channel router used by Chip Churn was grid based and was unable to cope with many types of cyclic constraint. For this reason a new, gridless channel router was developed for Elgar. Details of this channel router can be found in appendix A.

If there are ports on one side of a channel which do not connect to anything else in the channel, they are routed to appear at both ends of the channel. The user can suppress this feature or choose a single end on which the unused port will appear.

Multiple Compositions

One limitation of simple two-cell compositions is the proliferation of cell names that result. Though this may seem trivial at first, it has proved to be a very real problem in practical systems. As a result Elgar provides two methods of composing together many cells simultaneously. The first method allows compositions to be expressed in the form of text strings; the second method allows an applications program to build up a graph of cell compositions which can all be carried out at once.

```
start comp graph (elgar ver,part lib)
  start branch (elgar hor)
    define node ("a")
    define node ("b")
  end branch
  define node ("c")
  start branch (elgar hor)
    define node ("d")
    start branch (elgar ver)
      define node ("e")
      define node ("f")
      define node ("g")
    end branch
    define node ("h")
    define node ("i")
  end branch
  define node ("j")
end graph
```

Figure 4–13: Building a Liszt graph

Liszt

The Liszt system allows an Elgar user to construct a composition graph in which the nodes are cells and the arcs are composition operations. Composition graphs take the form of trees with branches of either horizontal or vertical compositions. Because Liszt graphs are acyclic (by construction) there is a clear order in which the composition operations can be applied. This means that the will be no cyclic constraints during placement and channel allocation. Composition of Liszt graphs can be done by a simple recursive descent of the graph structure. In the current version of the graph composition software, the Liszt graph is scanned to add routing cells before the final composition is carried out.

Figure 4–13 shows a program segment which builds a Liszt graph and figure 4–14 is a diagram of the resulting graph and its layout.

It can be seen from figure 4–13 that it is possible to nest branch definitions; there is no limit to the depth to which this nesting can be taken. Because the

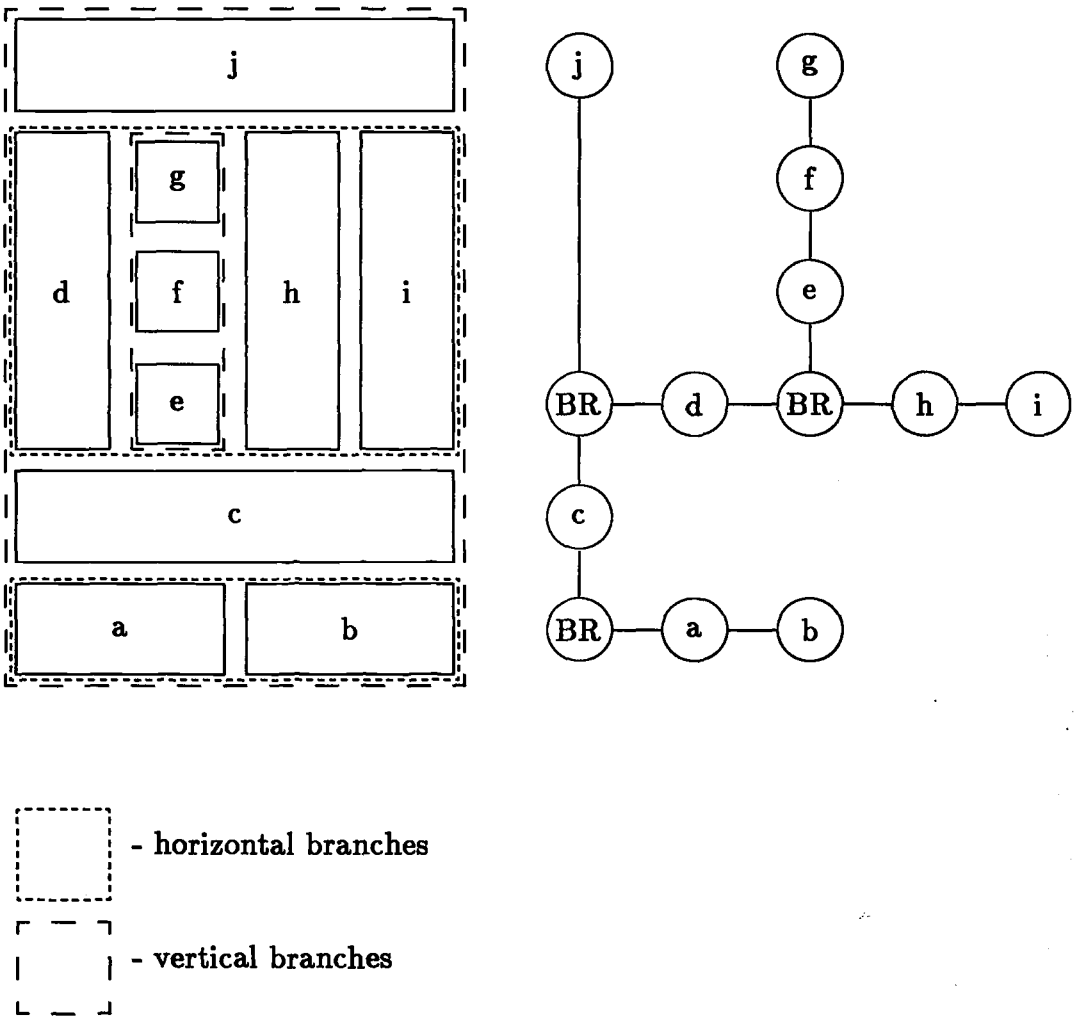
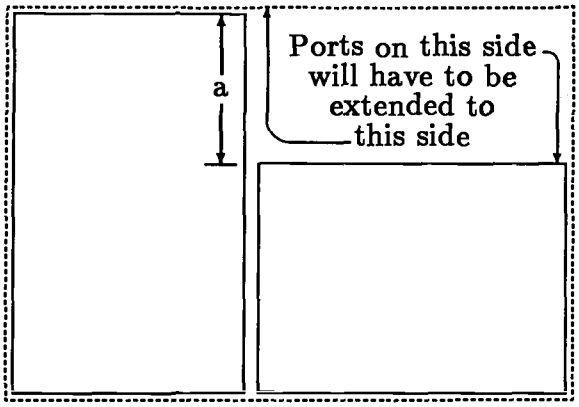


Figure 4–14: A Simple Liszt Composition - Layout and Graph



a : Ports in this area belong to the channel between the composed blocks.

Figure 4–15: A composition requiring port extension

Elgar system only deals with rectangular objects, it is sometimes necessary to extend the ports on those sides of blocks which do not correspond to the boundaries of the composition block. Figure 4–15 shows an example of this situation in a horizontal composition; the same thing can happen in vertical compositions. This port extension is done automatically within the Elgar system.

During compositions, cells are instanced by name. There are three special characters which can be prepended to a cell name to carry out transformations. These characters are :

'@' : Each of these symbols in front of a cell name implies a 90° anticlockwise rotation.

'!' : This symbol implies a reflection in the Y axis (i.e. of the x co-ordinates).

'_' : This symbol implies a reflection in the X axis (i.e. of the y co-ordinates).

4.6.5 Bend

Bend is the CMOS back-end for CC2 and as such it provides the interface between the TIIF data structure and the Elgar system. Once a composition

block has been identified for solidification, it is passed to the Bend system where the following steps are carried out.

1. Module Generation. This stage creates the geometry for the functional modules used in a design. During module generation, two libraries are constructed. The first of these is the archetype library in which there is a single example of each function used in the design. Generation of the archetype library involves the generation of PLAs, Wino arrays and the location of leaf cells in libraries.

   Once the archetype library is complete, a second "working" library is created. The working library contains a cell for each individual instance of a function in a design. It is at the working library generation stage that the connectivity of the CC2 design is embedded in the Elgar data structure. The working library will eventually contain all of the wiring and utility cells required in the final composition of the design.

   To simplify the routing problem, certain restrictions are placed on the form of functional modules used by Bend. In particular, the I/O ports on the modules must appear on both the north and south faces of the modules in the same order, although the location is not important. In addition, all the utility signals, power, ground, clocks etc, must be present on both the east and west sides of the modules, again in the same order.

2. Floorplanning. This stage creates the rows of function modules and carries out the routing between them. There are a number of steps involved in this process; these are :

   (a) Build Liszt : A Liszt graph is created in which a number of horizontal rows are composed vertically. The horizontal rows are made up of the cells in a design, divided up on the basis of the pattern of connectivity on the east and west sides of the cells. In this way all the cells with the same utility signals are collected in the same row. Ordering of

the blocks is done in an attempt to locate connected blocks in close proximity to each other.

(b) Aspect Ratioing : In an attempt to achieve an approximately square aspect ratio for the design, the (F)loorplan (OP)timiser (Fop) system takes the Liszt graph and breaks up the horizontal rows. Once Fop has broken up the original rows, alternative rows in the tree have their order reversed. This helps maintain the locality of connectivity achieved by the ordering of the original Liszt tree.

Manual intervention is provided at the Fop stage to allow user interaction with the floorplanning. This intervention can take two forms; the user may specify an ideal length which Fop will try and use as the x dimension in the layout, or the user may specify the complete floorplan. Complete floorplans are specified as lists of named cells, each list corresponding to one row in the finished design. These floorplan lists can be entered interactively or can be read from a file. The output from Fop is a Liszt composition graph.

(c) Minimise Port information : The Liszt graph produced by Fop can contain more port information than is required by the Elgar system. Because all of the cells have ports on both sides, not all of the ports may be needed to achieve minimum connectivity. There is also the problem of connecting modules which do not share a common channel. To solve these two problems unnecessary ports are hidden and feed-throughs are introduced where they are required. The final job done at this stage is to identify primary I/O signals (that is signals connected to pads) which will not have a port located on the boundary of the composition cell. These nets are assigned single ports in channels; this will result in their being routed to the ends of the channels and hence the edges of the composition cell.

(d) Place and Route : The Liszt graph containing feed-throughs and the minimised connectivity information is passed directly to the Elgar composition system where the placement and routing is done auto-

matically. Because the graph composition system is predictable, this automatic place and route results in the multi-channel architecture used by Bend.

3. Utility Routing : This process is straightforward as all the utility signals are present on the east and west sides of the main design composition cell.

4. Pad Placement : A simple pad placement system is called to put the pads on the design.

4.6.6 Bes

Earlier it was claimed that one advantage of implementation dependence was the availability of certain structures which might be of use in FTFS devices. If such structures are to be available to the CC2 system, it is desirable to standardise the interface between the modification routines and any particular back-end. The Back End Server (Bes) provides such an interface to the technology dependent information.

Bes is a set of routine calls which a technology back-end is expected to provide for the modification software. A typical application of Bes is to provide the name of library cells or cell generators which can be used to generate check logic gates such as parity checkers.

4.6.7 Example

As an example of the sort of layout produced for CC2 by Bend and Elgar, figure 4–16 shows a design for the eight bit ripple carry adder example of figure 4–9.