To the Graduate Council:

I am submitting herewith a master thesis written by Quinn Devine entitled "Comparison of Monolithic Versus Partitioned VLSI Designs.." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Donald W. Bouldin

Dr. Donald W. Bouldin, Major Professor

We have read this thesis
and recommend its acceptance:

Peyman Dehkordi

Roberto Balderrama

Accepted for the Council:

Lewinkel

Associate Vice Chancellor
and Dean of The Graduate School

COMPARISON OF MONOLITHIC VERSUS PARTITIONED VLSI DESIGNS

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Quinn Devine

May 1996

Acknowledgement

I would like to thank Dr. Don Bouldin for his guidance and support throughout the process of writing this thesis, and throughout my entire stay at the University of Tennessee.

A very special thanks to my parents, Nancy and Michael Devine, for their love, support and patience throughout my childhood and early adulthood. Thanks also to all my friends in Knoxville for their encouragement and for keeping me sane.

ABSTRACT

The design of application specific integrated circuits is becoming more complex as technology improves, allowing higher circuit density and faster speeds. As the system complexity increases, so does the complexity of the design and physical implementation processes. Consideration must be made of the higher probability of errors in the fabrication of a very large device. A system can be designed so as to allow one to detect manufacturing faults after it has been fabricated, therefore, faulty parts can be detected before they are shipped, which reduces per part cost significantly.

Consideration of other system configurations might be necessary to further reduce costs. A single, large chip design may be partitioned into a configuration of several die on a Multi-Chip Module, which reduces the cost of the system with respect to the monolithic version. Further cost reductions can be had if a different method of bonding the chip I/O to the MCM substrate is implemented. A tradeoff must occur between different partitions that considers the relative cost and performance, in order to determine the best solution.

A very large design (SPARC CPU) is synthesized and physically implemented as a single chip and as an MCM with wire bond die. The per part cost of the system is determined to be $529.50 and $303.13 for the single chip and MCM, respectively. However, the MCM is about 3.7 times larger than the mono-

lithic version. The cost of an MCM with area array bonded die is estimated to be $54.38 and is only about 10% larger than the monolithic version. These cost savings should convince a system designer to consider an MCM implementation of a very large design, as opposed to a die in a single chip package.

TABLE OF CONTENTS

CHAPTER PAGE

CHAPTER PAGE

CHAPTER PAGE

LIST OF TABLES

LIST OF FIGURES

FIGURE PAGE

FIGURE PAGE

CHAPTER 1

INTRODUCTION

The design of reliable LSI and VLSI circuits is of utmost importance, as circuits become increasingly complex, fast and compact. The large number of possible faults make it extremely difficult and expensive to test a circuit solely by mechanical means external to the unit under test [17]. The incorporation of testability into the circuit requires only a few more I/O pins and software to include the test circuitry and generate the test patterns used to verify the design once fabricated. The extra resources spent on design for testability pay off with more reliable circuits and less expensive testing processes[1].

As the design cycle is forced to contract in order to keep up with market forces, it is imperative that decisions affecting all aspects of the design be made quickly and with a strong measure of confidence. These decisions will be based on experience or familiarity with the design for test methodology.

Choice of design implementation is also a factor. A very large die area makes the design much more susceptible to manufacturing faults. Partitioning the design into several die configured on a Multi-Chip Module (MCM) may be

necessary to reduce the overall cost of the system. This requires some structural modifications to account for the new design environment. Further cost reductions for a multiple-chip design can be realized by using a different type of I/O bonding for each chip.

1.1 Overview

This thesis is organized as follows. In chapter 2, some of the issues concerning design testability are introduced. The topic of design for testability is rather broad and there are some issues and many approaches to the problem that are beyond the scope of this thesis and are not discussed. Chapter 3 serves as an introduction to the CAD environment and the implemention of the design for testability techniques. In chapter 4, a small example circuit, a microprogram address sequencer, is used to prove the test design methodology. A much larger design, a SPARC CPU, is put to the test in chapter 5. Several comparisons are made between a monolithic and a Multi-Chip Module version of the design. The cost for both configurations is quantified, taking into account yield, size, power dissipation, and testing costs. An estimation of the cost of an area array I/O bonded MCM is made and compared to the wire bonded MCM to demonstrate the benefits of restructuring the design for area array bonding. In chapter 6, proposals for further development of the ideas presented in this thesis are made.

CHAPTER 2

Background

2.1 Testing in a VLSI Environment

Testing is used to determine if an IC exhibits the properties it was designed to possess. The design must display correct timing and logical behavior and it must be free of physical defects, such as shorts or opens, that may occur during the manufacturing process. This chapter introduces some basic testing concepts and methods for assuring high expectation of device reliability.

2.2 Digital IC Testing

A design may be described at different levels of abstraction. At the highest level, the system is a black box, for which only the functionality is defined. At lower levels, the circuit model contains greater detail, consisting of submodules, gates, or transistors, as well as interconnections between the components. The objective of testing is to find faults, or the manifestations of manufacturing defects,

which lead to erroneous behavior. As a circuit is described in greater detail, there are more potential faults to consider. A test is based on a model of these faults, which is directly correlated with the level of abstraction of the circuit.

Once a fault model is established, a set of test input vectors can be generated which can be used to detect as many potential faults as possible, meaning that each vector will produce an opposite value at the output in the presence of a fault as compared to the fault-free output. A more complex fault model will likely require more vectors to be generated to cover the inherently greater number of faults, but since the model is more comprehensive, the vectors will detect more types of faults than can be specified by a simpler model. One must trade off the robustness of a test with the cost of generating and applying a greater number of vectors.

2.3 VLSI Fault Models

Not all defects that occur in the active area of an IC affect the behavior of the circuit. These defects may affect the reliability, but are ignored during the test generation process. A model must be established that represents the defects which cause logical errors, and therefore, system failure. The simplest and most widely used fault model is the *stuck-at* fault model, which is easy to implement, but inadequate at detecting some physical defects.

2.3.1 Stuck-at Faults

The stuck-at fault model is a structural model, meaning the faults are defined based on the structural description of the circuit [13]. It assumes that the components are fault-free, and only interconnections between components are defective. Typically, there are two defects associated with interconnect: shorts and opens. An unintentional short between a signal wire and power or ground will cause that signal to be stuck at a logical 1 or 0. An open on a unidirectional line with only one fanout causes a stuck line past the point of the fault. A single logical fault can represent a possible short, open, or an internal fault in the component driving the line that causes it to be stuck at a logic value.

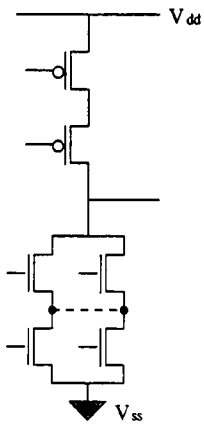
To test a stuck-at fault on a given line, the line should be activated, or sensitized to the logic value opposite to that of the assumed stuck-at value. For example, a 1 should be applied to a line to test for a stuck-at-0 fault. For internal line faults, the primary inputs must be set such that their implications can activate the fault. In an otherwise fault-free circuit, the fault effect must be observed at the primary outputs of the circuit. If the fault effect cannot be propagated to the primary outputs, or if the path cannot be sensitized by manipulating the primary inputs, a potential fault on that line is said to be redundant and cannot be detected. A redundant fault does not affect the logical function of the circuit, but it may affect the reliability [1]. The stuck-at fault model cannot detect other physical defects, such as those described in the next section.

2.3.2 Bridging Faults

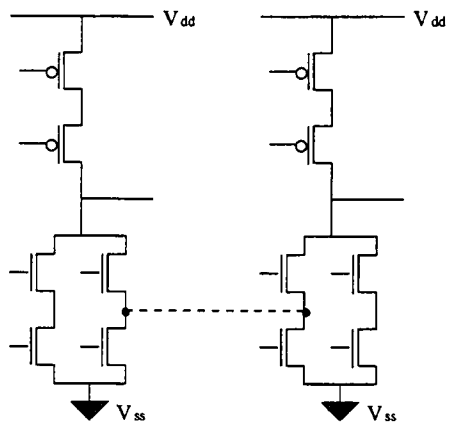
Bridging faults, like those shown in figure 2.1, are unintentional shorts between two signals [13]. They do not cause lines to be stuck at a value, but rather alter the functionality of the circuit. Bridging faults are a major failure mode in digital systems and can comprise up to 40% of all faults in a circuit [1]. There are three classes of bridging faults: bridging within a gate without feedback (figure 2.1(a)), bridging between two gates without feedback (figure 2.1(b)), and bridging between two gates with feedback (figure 2.1(c).) A fault within a gate, depending on the resistance of the bridge, might only cause a logical error. These faults have effects that are equivalent to stuck-at faults associated with the gate output lines. A fault between two gates without feedback must be analyzed at the transistor level to evaluate its effect, but may actually change the logical function of the circuit. Finally, if a fault occurs between two gates that depend on one another, the analysis of the effect can be complex, but the fault may cause unstable and oscillatory circuit behavior [17].

2.3.3 Open Faults

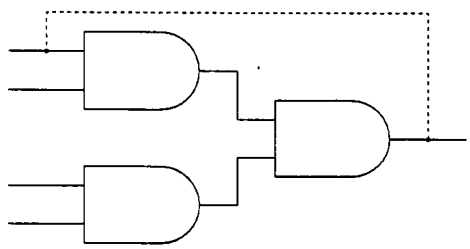
An open fault may be a break in a line connecting two gates and could be covered by the stuck-at fault model. An open may also occur within a CMOS combinational gate, causing it to exhibit sequential behavior, which cannot be modeled at the stuck-at level [13]. In figure 2.2, there are opens at locations 1 and 2 in



(a) Bridging within a gate, without feedback.



(b) Bridging between two gates, without feedback.



(c) Bridging between two gates, with feedback.

Figure 2.1: *Three types of bridging faults.*

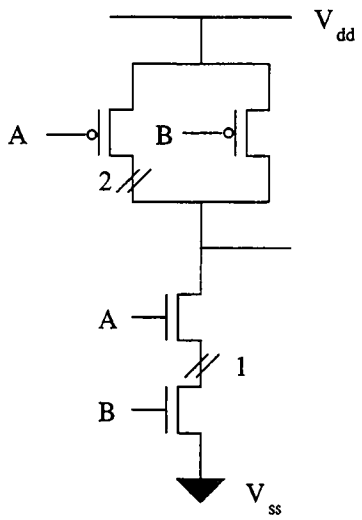


Figure 2.2: *A NAND gate with two possible open faults.*

a NAND gate. Open #1 prevents the output from ever being pulled down to V_{ss}, regardless of input. Therefore, the gate exhibits stuck-at-1 behavior. In the presence of open #2, the output is prevented from being charged when {AB} = 10_b. Because the load capacitance is not charged by the application of a 0 at B, the output retains its previous value. The gate is not stuck at a value, because if (A,B) makes a transition from 11_b to 01_b, the output changes from 0 to 1. The net effect is that the NAND gate is now a sequential device. Testing for open faults can be very difficult, requiring multi-pattern tests for single faults, and they are susceptible to timing skews, charge distribution, and glitches [13].

2.3.4 Parametric and Transient Faults

Parametric faults do not immediately affect the logical behavior of the circuit, but do degrade the performance and reliability of the device, and may eventually lead to logical errors [13]. Significant substrate or gate oxide leakage current, variation in threshold voltage (body effect), capacitive coupling, and crosstalk are examples of parametric faults [18]. A delay fault is also considered a parametric fault, caused by a physical defect or variation in process parameters, which manifests itself as a variation in delay through a gate. The cumulative effect of delay faults through a signal path may cause malfunction related to errors in timing, such as setup and hold violations, or static and dynamic hazards. Transient faults affect the circuit at random and are caused by external electromagnetic interference or ionization radiation, which can, among other problems, cause soft errors in dynamic memories [13].

2.3.5 Fault Equivalence, Location and Dominance

Two faults are said to be equivalent if a set of all test vectors that detect either fault yield the same circuit response in the presence of either fault [1]. A test set distinguishes two faults if it can provoke a different response for either fault. A fault f is said to dominate fault g if a test vector set that detects f also detects g. There may still be a test vector that distinguishes fault g from f [1].

The set of all possible faults is partitioned into functional equivalence classes, each containing a set of equivalent faults, and it is sufficient to consider only one representative fault from each class when generating a set of test vectors. A fault can be located within a fault equivalence class if it can be distinguished among detectable faults. Therefore, a complete location test vector set must include a complete detection test vector set in addition to vectors that will distinguish between each pair of detected faults. This extended vector set does not improve fault coverage, since any additional vectors detect faults that have already been covered.

A set of vectors can be collapsed by reducing the number of faults considered. For instance, if a fault is equivalent to another, then one can be eliminated from consideration. Extending this concept, a fault equivalence class need only contain a single fault. Also, if a fault is dominated by another, it can be eliminated, as long as fault location is not a concern.

2.3.6 Choosing a Fault Model

The cost of generating and applying more robust test vectors must be weighed against the benefits of higher total fault coverage. The most widely used model is the stuck-at fault model, because it is easy to implement, it generates few test vectors, but has reasonably high total fault coverage. This model can be independently formulated, regardless of the circuit target technology, since the

concept of a line being stuck at a logic value is applicable to any structural model. Often, a stuck-at fault dominates other types of faults that have the same logical effect, and can be collapsed. The total number of stuck-at faults in a circuit is less than the total possible faults, so even if a test vector set covers 100% of stuck-at faults, a fault-free circuit cannot be guaranteed. Generally, very high stuck-at fault coverage is a good indication of high total fault coverage. Other, more complicated models, such as the bridging fault model, can be used to generate a much more robust test vector set, but the time spent generating and applying the vectors may be prohibitive.

2.4 Fault Simulation

Fault simulation is used to determine the behavior of a design in the presence of faults to a given set of stimuli [1]. The simulator compares the response to a stimulus by the fault-free circuit and the circuit containing a fault. If the responses are different, then the fault has been detected. The primary advantage of fault simulation is that it allows for a complete evaluation of the design based on a modifiable model, avoiding the cost of physically fabricating a prototype and physically testing it. Simulating one fault after another is called serial simulation, which is impractical for large circuits since the number of simulation passes would be significant. There are methods which allow for simulating more that one fault per simulation pass.

Parallel fault simulation considers a set number, W, of faulty circuits and simulates them simultaneously along with the good circuit [1]. A set of F faults requires $\frac{F}{W}$ passes to simulate all possible faults. This method tends to waste computations, since the entire circuit is evaluated even if only one input changes from the previous iteration. Deductive simulation considers only the fault-free circuit, attempting to deduce which faults on gate inputs could cause faults at gate outputs [1]. Only faulty results are kept track of, requiring complex calculations to determine the set of outputs of each gate in a design that propagate on to the next gate. Concurrent simulation is similar to deductive simulation, except that each gate can be independently simulated to allow the circuit simulator to determine which fault should be propagated at each gate.

Critical path tracing obtains test vectors by simulating only the fault-free circuit [13]. It then traces a path backwards from a primary output to the primary input, identifying all the critical values along the way. A critical value on a line is the value on that line for which a test vector can test for a fault opposite to the value. Critical path tracing directly identifies faults detected by a test, without simulating the set of all possible faults.

2.5 Test Generation

A method to determine a set of test vectors that, when simulated, detect a reasonably high number of possible faults. An exhaustive test generation refers to the generating and testing of all possible input vectors to determine the existence of faults. This method ensures maximum fault coverage for the circuit, but as the number of inputs increases, it quickly becomes impractical. For example, if there are 40 circuit inputs, then there are 2^{40}, or about 1.1 trillion possible input vectors. If the simulator runs at 5MHz, it would take two and a half days to simulate all the vectors. A trade off must be made between generating fewer vectors to reduce simulation time and detecting fewer faults.

A random test generation algorithm uses pseudorandom methods to generate test vectors, and is easy to implement, but in order to achieve a high fault coverage, a very large vector set is needed, as well as a lot of simulation time [13]. A deterministic test generation algorithm utilizes heuristics to determine the test vectors. The heuristics are based on highlighting the faulty paths and their propagation through the circuit. The process by which a fault effect is propagated through a gate is based on line justification, which is the ability to set the circuit primary input (PI) in such a way that a desired value can appear at a given line in a circuit. The other inputs to a gate which lies on the path through which a fault effect is to be propagated must be set in such a way that the output of the

gate depends only on the input that contains the fault effect. The appropriate values at the other gate inputs must be justified by the PI, otherwise the fault cannot be propagated.

A sensitized path is the path from a fault to a primary output (PO), where the fault can be observed [13]. A path is sensitized by making sure that the fault effect can be justified at the output of every gate along the path to the PO. Figure 2.3 illustrates the concept of justification and path sensitization. Assuming a stuck-at-0 fault occurs on line E, the fault is activated by setting $\{AB\} = 11_b$. The path from E to G is sensitized if the other input to the NOR gate is 0 (justified by setting C = 0) and the other input to the second AND gate is set to 1 (justified by setting D = 1.) A 1 will appear at F and a 1 will appear at G. If the fault at E was not present, then the output at G would be a 0, given the same values at the PI as before. Therefore, the fault has been detected by the vector $\{ABCD\} = 1101_b$. These concepts are the basis for most deterministic test generation algorithms.

An early example of deterministic test generation is the D-algorithm [1], in which a fault or discrepancy is denoted by a D, and a forward propagation of the D to the PO is attempted. Once the fault has been observed, a backtracking step is implemented to determine the necessary PI values to justify the lines used in propagating the fault. If the backtracking is successful, then the PI define a test vector for fault D.

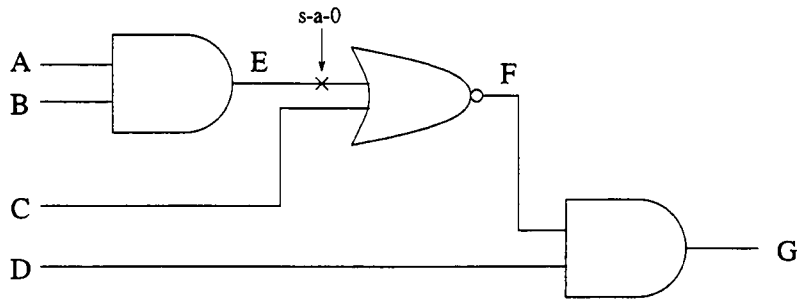


Figure 2.3: *A circuit containing a stuck-at-0 fault.*

A circuit which contains large fanout causes problems for the D-algorithm [1], since the decision of which branch to propagate the fault through must be made at the point of each fanout. During the backtracking step, it is possible that an assumed internal node value may not be justifiable by the PI or may have been set to two different values to satisfy fault propagation conditions at two different gates. If such a conflict arises, the proposed sensitized path is invalidated, and the desicions made along the path must be revised.

An algorithm that avoids conflicts altogether is called PODEM [1], or path-oriented decision making. PODEM defines the first objective to be the activation of the fault using line justification. The implications of the PI settings are simulated and stored so that future decisions can use that information. Subsequent objectives are to propagate the fault one step closer to the PO. Every decision that is made includes a line justification step, which requires setting more PI lines, and an implication step, in which the effect of the new PI settings is propagated

through the circuit. Once the fault effect is observed at the PO, the sensitized path is valid and the PI define a test vector that detects the associated fault. Execution time is saved by eliminating the final backtracking step and the possibility of conflicts.

2.6 Design For Testability

The testing of combinational circuits can be time consuming, but the testing of sequential circuits is even more complex. In order to detect a fault in a sequential circuit, multiple test vectors have to be generated, since the internal state of the circuit must be defined as well as the current primary inputs used to propagate the fault. Techniques falling under the title Design for Testability [17] have been developed which can reduce the problem of generating tests for a sequential circuit to generating a test for a combinational circuit by the designer adding test access points internal to the circuit.

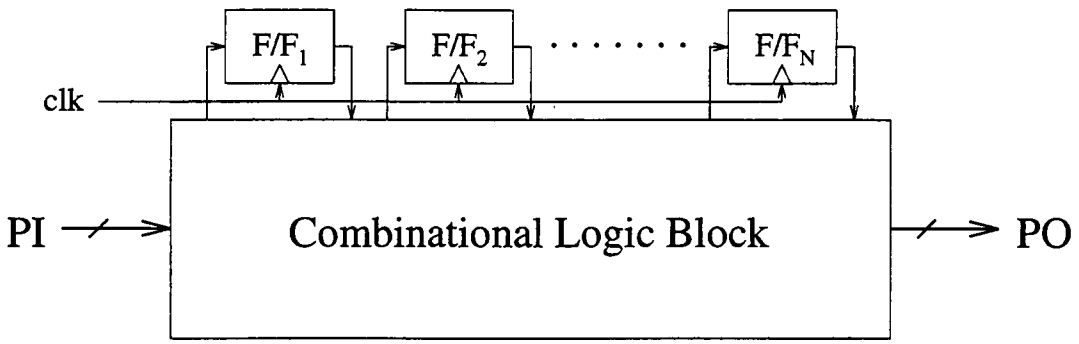
2.6.1 Full Serial Integrated Scan

Figure 2.4(a) shows a model of a sequential circuit, consisting of a combinational logic block (CLB) and N edge-triggered flip-flops. The inputs to the CLB are the primary inputs to the circuit and the outputs of each flip-flop. The outputs of the CLB are the primary outputs of the circuit and the inputs to each flip-flop.

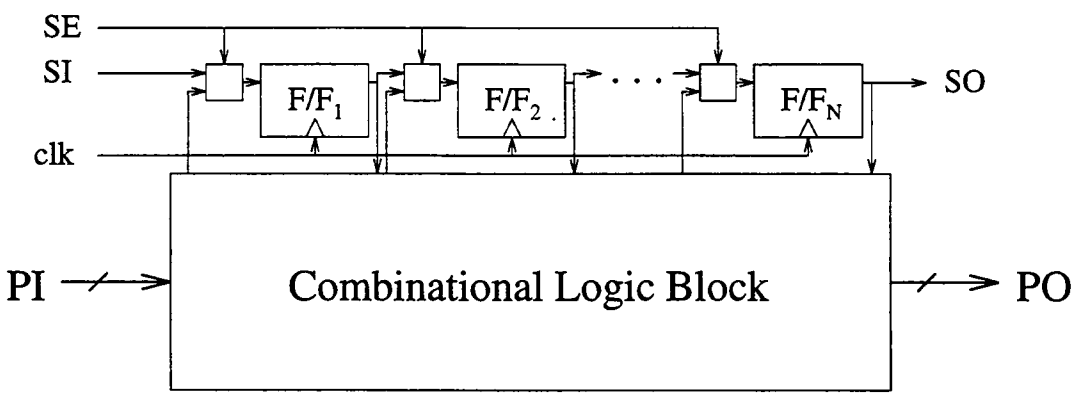
While observability and controllability of the PO and PI is possible, the present state lines to the flip-flops cannot be controlled and the next state lines from the flip-flops cannot be observed, so the test generation process will be profoundly impaired.

The circuit can be modified as shown in figure 2.4(b) to include an extra control input that is used as a multiplexor select line that selects between normal operation mode inputs and scan mode inputs. When the control input signal, or scan enable (SE), is activated, the flip-flops are connected together and act as a serial shift register. The first flip-flop takes a new signal, scan in (SI), as its input, and the last flip-flop sends its output to scan out (SO). One can now shift in an N-bit value into the register using the same system clock. This configuration allows for the use of combinational test generation techniques described earlier to generate test vectors without regard to sequential elements [17].

The input vector generated by the test generation (TG) algorithm consists of values to be applied to the primary inputs and values to be applied to the present state inputs by shifting them in through the scan chain. The output vector as specified by the TG algorithm is compared to the response of the circuit to the input vector observed at the primary outputs and the next state outputs, which are captured by the scan chain and shifted out. Test generation for a sequential circuit can be greatly simplified by replacing every sequential element with a scan-compliant equivalent that when enabled for scan, form a chain, but



(a) Simple model of a sequential circuit.



(b) A sequential circuit with scan circuitry inserted.

Figure 2.4: *Scan insertion into a sequential circuit.*

otherwise function normally.

The configuration in figure 2.4 assumes the sequential elements are edge-triggered. Figure 2.5 shows a generic full serial integrated latch-based scan design, or level-sensitive scan design (LSSD) [17]. This configuration is used to eliminate race conditions and static hazards that might occur in an edge-triggered implementation, by controlling when clocks change with respect to the inputs. In normal operation mode, the L_1 device latches in the next state using non-overlapping clocks A and C. While in scan mode, the L_1-L_2 pair latches the serial scan data shifted in from SI, using non-overlapping clocks A and B.

Although insertion of scan circuitry greatly simplifies testing of sequential circuits, there is an increased cost incurred. Adding extra logic increases the area of the entire circuit due to active circuitry and increased routing complexity. Extra dedicated scan pins will be necessary to accommodate the scan chains. The delay of the multiplexed flip-flop scan circuit is greater due to the additional two levels of delay added between the sequential elements. The LSSD circuit does not have this problem, but both implementations have double the fanout at each sequential element (one line back to the CLB, one to the next sequential element), thereby slowing the circuit down.

The test time per input pattern is increased because part of the input vector has to be serially shifted in and part of the output vector has to be serially shifted out, so the test cycle is approximately equal to two times the number of

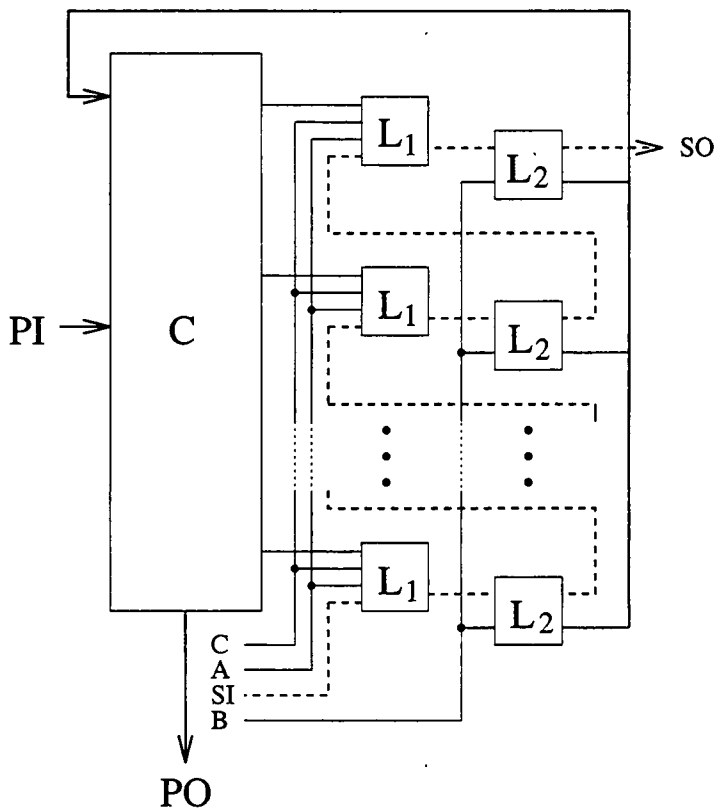


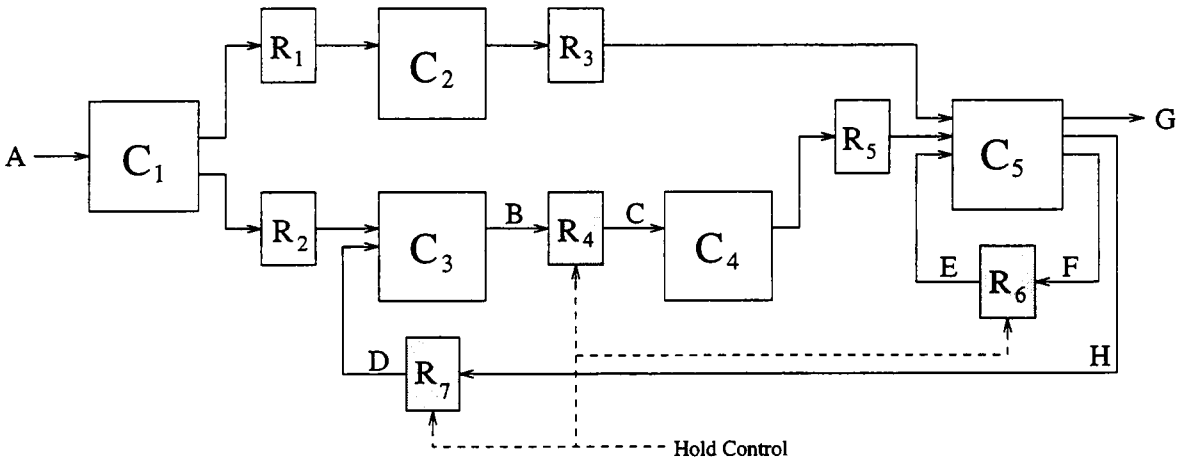
Figure 2.5: *Level-sensitive scan design (LSSD).*

scan cells in the scan chain divided by the scan clock frequency. This time can be reduced by using multiple scan chains, scanning in the internal state test vectors in parallel. Considering the fact that most combinational test vectors only require the setting of a few internal storage elements, the fault effect may also propagate to only a few storage elements. Therefore a partial scan implementation that converts only a few necessary sequential elements into scannable equivalents has been developed.

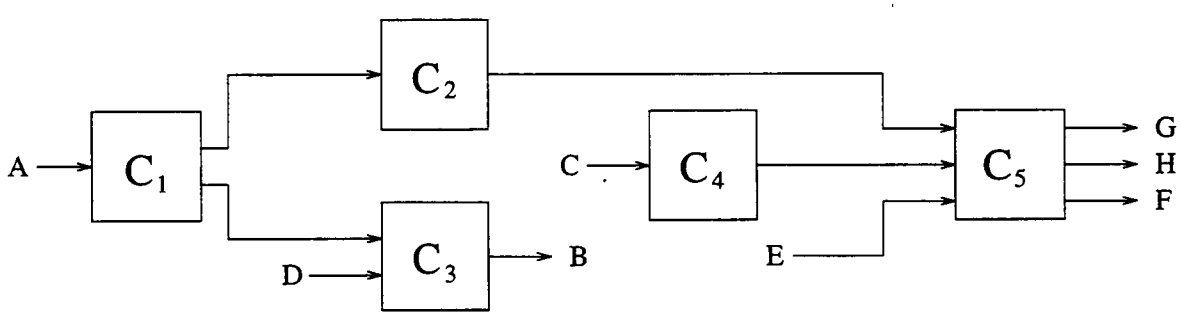
2.6.2 Partial Scan

The main consideration when implementing partial scan is the choice of which flip-flops to be included in the scan chain [1]. Factors that affect the choice is fault coverage, length of scan chain and therefore, scan cycle, and performance. Since adding scan circuitry slows down the circuit, sequential elements along the critical path should be excluded from the scan path.

Generally, a sequential circuit consists of blocks of combinational logic, or clouds, separated by registers, or collections of one or more sequential cells. An example of a sequential circuit grouped into registers and clouds is shown in figure 2.6(a), where C_i are clouds, and R_j are registers. The rule of thumb in selecting which registers to add to the scan chain, is to balance the circuit, which means that all paths between any two clouds should contain the same number of non-scan registers.



(a) Sequential circuit consisting of clouds C_i and registers R_j.



(b) Combinational equivalent circuit.

Figure 2.6: *Partial scan design.*

In figure 2.6 R_4 is made into a scan cell so that the two paths between C_1 and C_5 contain two non-scan registers. Choosing R_2 or R_5 instead of R_4 would have been appropriate as well. R_6 should also be converted to a scan register because C_5 feeds back to itself and is considered to be unbalanced with itself. Finally, R_7 should be converted to break the long feedback from C_5 to C_3.

The resulting circuit can now be transformed to a combinational equivalent of the original circuit. The remaining non-scan registers are replaced by wires connecting their input and output, and the scan registers are removed from consideration, their input becoming pseudo-primary outputs and their output becoming pseudo-primary input of the combinational equivalent circuit. The new circuit is shown in figure 2.6(b). The depth, d, of the combinational equivalent circuit is defined as the largest number of non-scan registers on any path between two clouds.

A set of test vectors can be generated using a combination test generation algorithm based on the combinational equivalent circuit and can be applied to the sequential circuit to detect faults. A test vector set $\mathrm{T} = \{t_1, t_2, \ldots, t_n\}$ detects all stuck at faults in the combinational equivalent circuit, where each t_i consists of the portion of the input test vector to be applied to the primary inputs and the portion to be scanned into the scan chain, essentially applied to the pseudo-primary inputs. The test vectors are applied as follows:

1. Shift t_i^b into the scan chain and apply t_i^a to primary inputs.

2. While holding t_i^a and t_i^b, clock the non-scan registers d times, thereby propagating all present values through the circuit.

3. Disable scan mode and clock once.

4. Observe the primary outputs.

5. Simultaneously shift out the scan chain and shift in the next t_i^b. The values shifted out are combined with the observed primary output to form the test response vector.

The advantages of implementing partial scan with respect to full scan are that the area overhead is not so high, the use of fewer scan cells means that there will be less delay between next state lines since many sequential elements will have one less fanout than they would have under a full scan configuration. Not all sequential circuits can be reduced to an equivalent combinational circuit, so a sequential test generation algorithm may be need to generated test vectors, but that task is made less difficult by implementing the partial scan methodology.

2.6.3 Built-In Self-Test

A scan-based design requires Automatic Test Equipment (ATE) to store test vectors and expected responses [17]. ATE is expensive and requires slow test

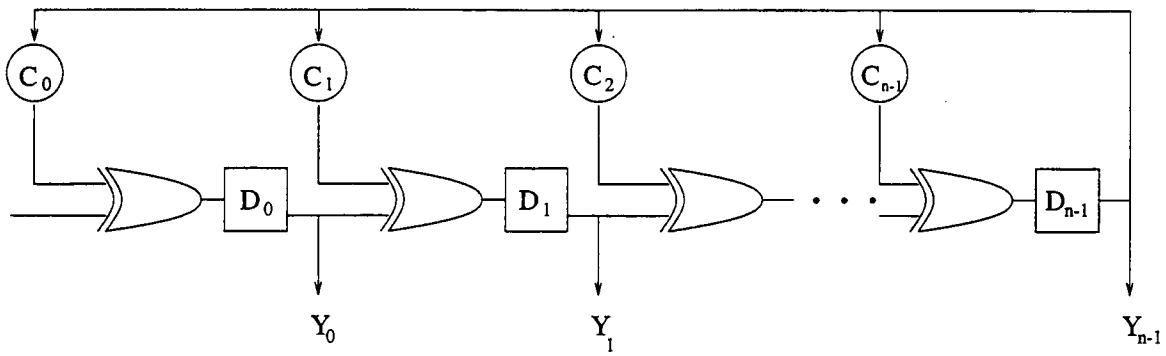


Figure 2.7: *Linear feedback shift register used to generate pseudorandom vectors.*

speed because of long interconnections between the ATE and the device under test (DUT). Most disadvantages of ATE are eliminated by implementing Built-In Self-Test, or BIST [17]. Extra circuitry is added around the original circuit and either comprise the test generation circuit or the test response compressor. The test generation circuit generates pseudorandom test vectors, and the response by the circuit to those vectors is compressed, since the volume of responses can be large and it would be impractical to store all the values on the chip.

A typical implementation of a pseudorandom vector generator is a linear feedback shift register (LFSR), shown in figure 2.7 [1]. An LFSR is comprised of an n-bit shift register with feedback available from the output of the last flip-flop to the input, via an XOR gate, of each previous flip-flop. The constants, C_i can be either a 0 (open circuit) or 1 (short). The initial state of the register is set

to all 0's, except for D_0, which is set to 1. At each clock cycle, the n-bit LFSR output vector, Y, can be used as a test vector. All of the $2^n - 1$ possible vectors will be generated by the LFSR, then the pattern will repeat. This implementation is useful because one can limit the input vector set to a subset of all possible n-bit vectors and be assured that every vector can be represented by the LFSR.

The amount of test response data is $N \times n \times m$, where N is the number of test vectors, n is the number of primary inputs, and m is the number of primary outputs. This data must be compared on-chip to the response of the fault-free circuit, therefore, it needs to be stored. The data must be compressed, and then compared to a compressed version of the expected response in order to fit all the data on-chip. If the compressed vectors, or signatures, match then either the circuit is fault free, the fault has not been sensitized by any pattern used during self test, or the information reflecting a fault detection has been lost during compression. The latter condition is known as aliasing [13].

The most widely used data compression technique is signature analysis [1]. A response signature is created by shifting an m-bit response vector into an LFSR similar to that shown if figure 2.7, except the length, p, of the register is chosen to be some number less than m. The signature is the p-bit output of the LFSR after the last bit of the m-bit response vector has been shifted in. The signature is then compared to the expected signature, which has been previously stored in on-chip memory. There are a possible $\frac{2^m}{2^p}$ response vectors that have the same signature,

therefore, a fault detected by a particular test vector and reflected in the output is aliased by $2^{m-p}-1$ other response vectors. The probability of aliasing is affected by the choice of the feedback constants, C_i, as well as the length of the LFSR.

2.7 IDDQ Testing

Test vectors generated to detect stuck-at faults in CMOS circuits do not cover most physical defects, so a test generation algorithm based on power supply current monitoring was developed. This technique, known as IDDQ testing, can be used to detect bridging faults, open faults, and some parametric faults such as gate oxide leakage current [13]. For a fault-free CMOS gate, either the nMOS or the pMOS transistors will be turned on, and a low impedance path between ground or power to the output is established. No steady state current flows, save a negligible junction leakage current. In the presence of a bridging fault, a steady state current may flow between power and ground if a low impedance path consisting of a bridge and active transistors turned on by the appropriate combination of gate inputs. In this case, a large current appears in the circuit, which can be detected by monitoring the power supply current.

This method does not require fault propagation, since the fault effect is monitored at the power pins, not the primary output. Slow current measurement speed is the limiting factor for efficient IDDQ testing. On-chip current sensors,

comprised of a circuit breaker and differential amplifier, are necessary to detect any abnormally high currents. However, the sensors increase the hardware overhead and slow down normal circuit operation. IDDQ testing has not been widely adopted in production testing.

2.8 Other Work

Test generation is a complex problem with many aspects to consider. The cost of the generation depends on the complexity of the calculations involved in generating the tests and the amount of data generated, which, of course, also depends on the complexity of the circuit. A larger test set means more memory will be required to store the data, and more time will be needed to simulate the faults. Since there are many ways to approach the problem, there are many solutions, a few of which are described in this section.

SEST[3] is a gate-level deterministic sequential circuit test pattern generation algorithm based on justification state equivalence techniques. SEST stores all line justification decisions it makes when activating faults and searches through them when justifying the next possible fault in case that fault falls into a previously established fault equivalence class. This saves time by avoiding the generation of test for equivalent faults. The user can specify how many passes are made through the existing fault list when searching for equivalent faults. Also, the

number of backtracking steps when trying to justify a fault before abandoning it can be specified. SEST uses a fault simulator to evaluate the quality of the test generated. The fault list that SEST generates can be quite large. For instance, generating a test for the ISCAS89 benchmark circuit s9234, which consists of over 200 sequential elements and 2000 gates, produced over 500M of fault data. A more efficient data structure would speed up the algorithm.

Nemesis[9] is a test pattern generator that generates test for stuck-at faults as well as bridging faults for combinational and sequential circuits. The algorithm used is based on a Boolean satisfiability method[11], which constructs a boolean equation for the circuit and its corresponding faulty circuit, combines them to form a Boolean difference equation. This equation is then satisfied by choosing the correct values so that the equation evaluates to 1. Accompanying Nemesis is a graphical user's interface, Voyeur, which can show the location of a particular stuck-at or bridging fault.

FATE[6] is a software tool that analyzes circuits described in a hardware description language, VHDL, at the structural level. It contains event-driven fault simulator that can evaluate static faults and dynamic faults. Stuck-at faults are considered static faults because they exist regardless of the stimuli. Delay faults are an example of dynamic faults, which require at least two vectors per fault, one to initialize the circuit to set an initial value at the fault site, and one to cause a transition at the point and propagate the effect to the primary output.

The circuit is simulated in between vector applications to propagate effects of the input vector. FATE generates patterns for the detection of both stuck-at and delay faults.

Cbase[7] is an object-oriented database system for VLSI circuits which supports programs that implement circuit testability design. One program, CRETE, partitions a circuit into clouds of combinational logic and groups the storage elements into registers. Another program, SIESTA, determines the order that the registers are connected into a scan path, depending on the type of scan design specified by the user. Full serial or partial scan chains are formed, and a deterministic test pattern generation algorithm is used to generate test vectors for the reorganized circuit. Cbase includes a helpful graphical user's interface that reflects all the modifications made to a circuit and feeds back statistics from fault simulations.

2.9 Board and System Level Test

Before the advent of LSI technology, board or system level test was performed using edge-connector function test. This set-up consisted of a power supply, a collection of digital drivers and receivers operating in parallel, controlled by a sequencer, and an interface to the device under test (DUT) [12]. Programming the tester was a significant problem, due to the increasing complexity of designs.

A logic simulator was used to generate stimulus-response vectors based on an abstract model of the DUT, and failure mechanisms could be added to the model to track differences in responses to a known stimulus. This set-up required the test engineer to have intimate knowledge of the circuit, and high fault coverage was difficult to attain. As the number of transistors increased, the simulators were not able to process the larger models.

In-circuit testing solved some of the problems associated with edge-connection testing. A bed of nails fixture was used to gain access to internal nodes that were previously inaccessible from the edge pins [12]. One could now test individual devices on the printed circuit board, and if a test were to fail, the source could be located precisely. In-circuit testing was successful when most integrated circuits were mounted in dual in-line packages with through-hole mounting, since every signal was available for probing at the bottom of the board. However, when surface mount technology gained wide acceptance, probing became less effective, since more pins were not accessible to in-circuit probes. Also, integrated circuits could be mounted on both sides of the board, increasing further the difficulty of signal access. Another technological innovation, the Multi-Chip Module (MCM) allows several dies to be mounted on a single substrate. The density of the circuit is very high and access to the system is limited to the input and output pins [2].

2.9.1 Boundary Scan

To reduce the difficulty of in-circuit testing and to enable the testing of MCM's, boundary-scan testing was developed and standardized (IEEE Standard 1149.1-1990.) The standard, referred to as JTAG (Joint Test Action Group), defines architectural requirements and I/O protocol that provide a chip or board test interface and are independent of technology [14]. The boundary-scan technique uses a serial scan chain to access chips on a board or dies on an MCM, instead of using the traditional bed-of-nails approach. The boundary-scan chain forms a register between primary I/O and the core circuitry, and allows direct application and sampling of data, as well as access to the internal test structures. Besides chip testing, boundary-scan allows board level testing such as interconnect and cluster test [12].

The basic boundary-scan architecture is shown in figure 2.8. The TAP, or test access port, consists of 4 or 5 dedicated I/O pins that provide serial access to the instruction and data registers. The TAP controller is a 16 state finite state machine that controls the operation of the boundary scan logic. The instruction register accepts an instruction and decodes it through the instruction decode logic. The instruction defines the configuration of the test logic. Specifically, it selects which of the data registers is to be connected between the test data input pin (TDI) and the test data output pin (TDO). The decoded instruction activates certain control lines that are connected to the data registers and governs their

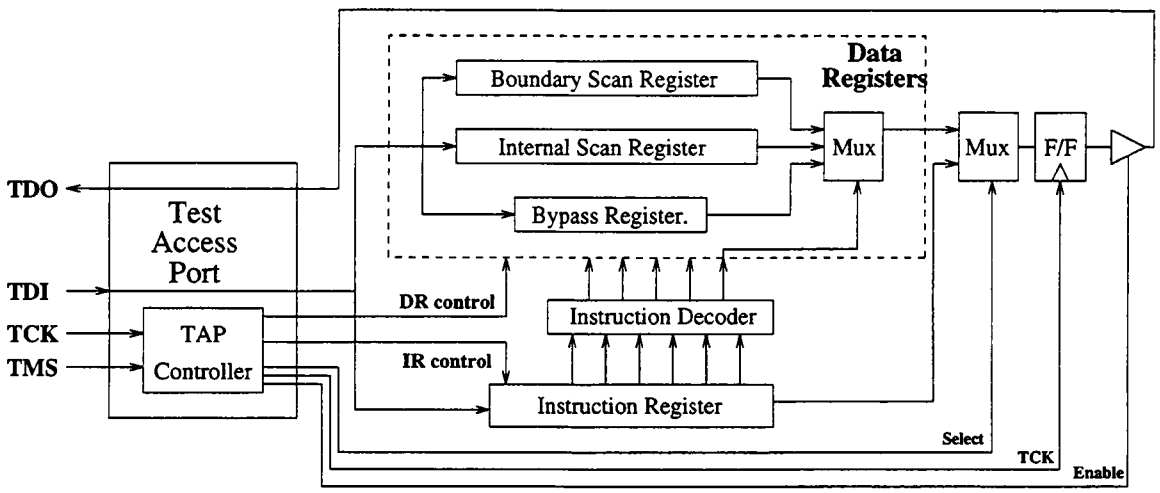


Figure 2.8: *Boundary-Scan Architecture.*

behavior. The data registers shown in figure 2.8 are the bypass register, boundary scan register and internal scan register.

The TMS signal is used to set the mode of the TAP controller, then the input data is serially shifted through the TDI port on the rising edge of TCK, into the instruction register. The instruction is decoded, and a data register is selected. The TDI port is then used to shift data into the data register. The test is then applied and the output of the core circuitry is parallel-loaded into the data register, and shifted out via the TDO port. The boundary-scan logic allows

one to perform a DC functional test of the core circuit by applying a functional test vector via the boundary scan chain. Also, fault test vectors can be applied by first selecting the internal scan register and shifting in that part of the test vector, then selecting the boundary scan register and shifting in that part of the test vector (the part corresponding to the primary inputs of the core circuit). The test is applied, then the test response vector is shifted out of the boundary scan register and the internal scan register through TDO.

Another primary function of the boundary scan architecture is to perform board-level test. Figure 2.9 shows just one of several possible configurations for testing two boundary-scan compliant chips. In this case, only one board pin each is needed for the TDI, TDO, TCK and TMS signals. One can test for shorts or opens on the interconnections between the two chips using the boundary scan chain, by serially shifting a pattern into chip 1, capturing the signal propagation at the input of chip 2, then serially shifting the result out of TDO. If a test is to be applied to only one chip, then an instruction that selects the bypass registers is shifted into the instruction registers of all other chips. The bypass register is a single flip-flop and provides a minimum path from TDI to TDO on a particular chip. All subsequent access to the TDI and TDO ports on the chip to be tested will not require shifting through the boundary scan chains of all other chips.

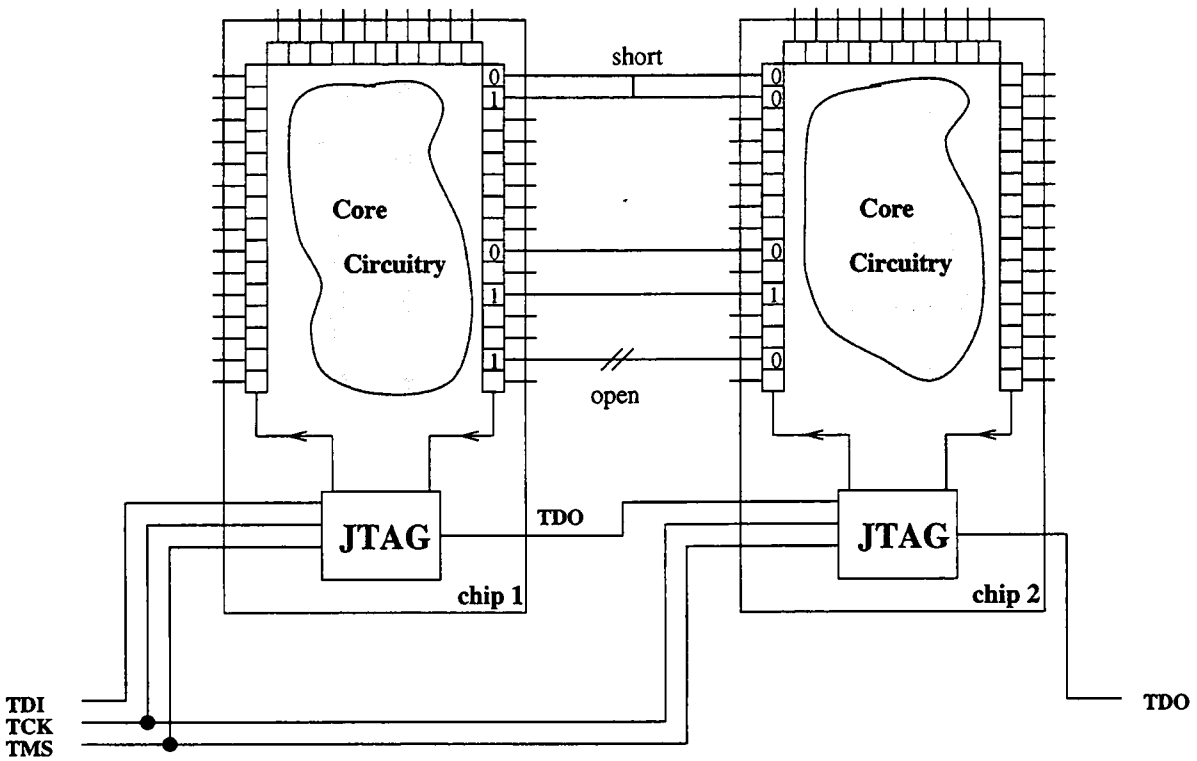


Figure 2.9: *Interconnect Test Using Boundary-Scan*

2.9.2 The Importance of Known Good Die

The classification of an unpackaged IC as "known good" implies a very high manufacturing yield, meaning the producer can guarantee a high percentage of the IC's are free of any defects and that they will remain fully functional and defect free [8]. This can be assured if the IC remains functional at full operating frequency throughout the full temperature range after next level assembly, environmental stress screening and field testing for a time approaching minimum lifetime. Mature products can achieve yields of 99% and higher, but the yield is much lower for new products [8].

Die yield is a primary contributor to MCM yield. For a system comprised of $\mathcal{N}$ die with corresponding probability of being known good die, P_k, an approximate relationship for MCM yield is as follows:

$$Y_M = 100\% \times [(P_1)^{L_1} \times (P_2)^{L_2} \times \cdots \times (P_{\mathcal{N}})^{L_{\mathcal{N}}}] \tag{2.1}$$

where L_k is the number of each type die P_k. The multiplicative relationship between die yield and MCM yield indicates how important it is to achieve very high die yield. For instance, if each of $\mathcal{N} = 9$ dies has a yield of 95%, the MCM yield is 63%. The above equation does not take into consideration other contributions to MCM yield, such as known good probability for substrate, assembly, and interconnect [8]. A more complete yield equation would be:

$$Y_{MCM} = Y_M \times (P_{sub}) \times (P_{assem}) \times (P_I)^J \tag{2.2}$$

where P_{sub}, P_{assem} and P_I are the known good probabilities for the substrate, assembly and interconnect, and J is the number of interconnect wires.

CHAPTER 3

Design Methodology

An implementation of an integrated circuit requires a design methodology chosen by the designer as a fast means of achieving a die that conforms to speed, power, size, and function specifications, and has a low defect level. The IC designer can describe the design at different levels of abstraction, subject to different constraints, which allows one to break down a large digital system into manageable parts and utilize automated procedures to generate acceptable solutions to problems faced during the design synthesis process.

IC design involves trading off between cost, performance, and time, which influences the development of a design methodology. Some general design issues common to many design methodologies are:

- Design Capture - the functionality of a digital system is described at one or more levels of abstraction.
- Compilation/Optimization - design can be logically optimized, mapped into a specified technology, then subjected to user-defined constraints such as timing, area, loading, temperature, and operating voltage.

- Design for Testability - implement test methodology, then re-optimize new circuit to meet constraints.
- Physical Design - translate structural netlist to a physical layout by placing and routing gates or macros while meeting design constraints.
- Packaging - select a configuration for packaging the IC, and route the core circuitry to the package I/O pads.
- Simulation - verify that the circuit behaves correctly.

In an ideal world, one would have access to a Computer-Aided Design (CAD) package that could perform all of these tasks, thus avoiding the problem of interfacing with other CAD tools. For the purposes of this research, the Synopsys toolset is used for prelayout synthesis and optimization and postlayout simulation, and Cascade's EPOCH tool is used for physical design and preparation of post-layout simulation data. Magic, a VLSI layout tool is used to add boundary-scan logic to I/O pads, which are then used by EPOCH to package the IC. The specific functions of each tool is described in the following sections.

3.1 Design Methodology Overview

The design path followed in this research is outlined in figure 3.1.

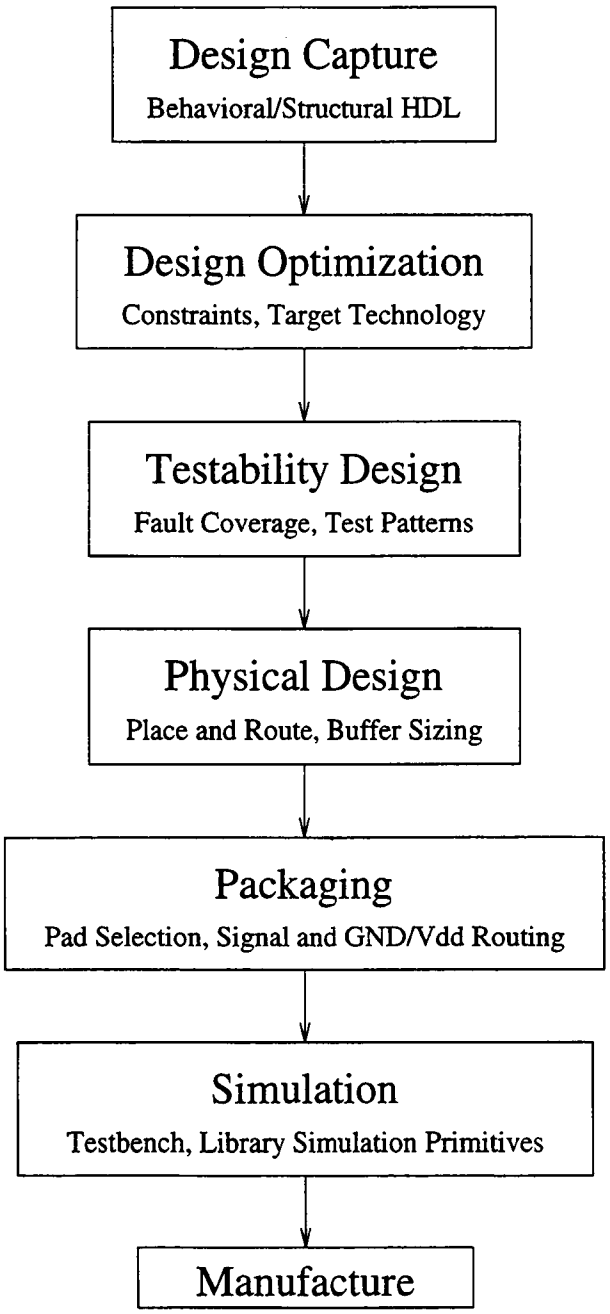


Figure 3.1: *Design Methodology.*

3.2 Synthesis

Synopsys' Design Analyzer is a menu-based graphical interface to all of the synthesis tools in the Synopsys CAD toolset, namely, HDL Compiler, Design Compiler and Test Compiler. HDL Compiler is used for design entry when the design is described in a high-level hardware description language like VHDL or Verilog. Design Compiler allows exploration of the design trade-offs by controlling user-defined constraints. Test Compiler inserts test structures and routes test signals. The interaction of these tools is managed by Design Analyzer providing an efficient means of managing a large, hierarchical design.

3.2.1 Design Capture

Synopsys allows several means of entering a digital design: gate-level netlist, FSM state table, PLA table, RTL-level HDL, or a schematic drawn using the Synopsys Graphical Environment tool. For this research, all designs are described using mixed-level HDL, either in VHDL or Verilog. Hardware description languages describe the architecture and behavior of a digital system. An HDL-based system allows the design to be broken into parts which can be synthesized concurrently by several designers, and provides for structured development.

An advantage of using HDL is that one can verify design functionality very early in the design process, by simulating the HDL coded design. This aids in early

architectural and design decision that have to be made. Using Synopsys' HDL compiler, an HDL description can be converted to a gate-level implementation in a given technology, which reduces circuit design time. Synopsys feeds back design information which can guide the designer to make better architectural decisions. HDL descriptions are not linked to any particular technology, so it can be used to generate the design in a different technology without having to translate from the original design technology.

Two kinds of HDL are used in this research, VHDL and Verilog. VHDL stands for VHSIC (Very High Speed Integrated Circuit) Hardware Description Language and is recognized by the IEEE and DoD (where it was developed) as a standard HDL. Verilog was developed by Phil Moorby at Gateway Design Automation in 1983 and was a proprietary HDL until 1990, when Cadence Design Systems opened it to the public. Both HDLs support mixed-level descriptions where structural netlists are mixed with behavioral and dataflow descriptions. A behavioral description specifies a function of a portion of a design as a sequential process. A dataflow description specifies the concurrent transformations acting on data as it flows from input to output of a portion of the design. A structural netlist connects all the components of a design, which can be gates or instantiations of modules that may be described at a higher level of abstraction.

The HDL compiler converts HDL source code to Synopsys' internal database format and performs architectural optimizations on the design. The design is rep-

resented at the logic level if the HDL contains a high-level description of the circuit behavior. A hierarchical design may include gate-level or structural level representations as well. The entire design can then be translated to a gate-level description in the target technology library using Design Compiler. The HDL can be simulated before it is converted to verify correct functionality using Synopsys' VHDL System Simulator (described later). This assures the designer that the design is on course to successful completion before any physical implementation is obtained. Future design simulation runs, such as a prelayout gate-level simulation and a post-layout simulation are compared to the HDL simulation, since it most accurately defines the intended behavior of the system.

3.2.2 Target Technology - HP26G

The target technology used for this research is the HP26G 0.8μ CMOS process provided by Cascade Design Automation. Technology libraries exist in two different formats, one for the physical design tool, EPOCH, and another for Synopsys. The Synopsys version contains information that applies to an entire design or to a single cell. Design attributes include operating conditions, timing variations, wire load models, and process parameter variations. These attributes exist because Synopsys doesn't synthesize a physical design, only a gate-level netlist which satisfies design constraints based on approximations of parameters that can't be accurately determined until a physical layout is obtained.

Some of the attributes that are attached to each library cell are: input and output pin capacitances, fanout loads, gate function, rise and fall times, pin timing relationships and gate area. Several different library cells may perform the same function, but may better satisfy the requirements of the design in other ways over a similar cell. The design compiler is responsible for searching the technology library and selecting the most appropriate cell for that particular iteration of design optimization.

3.2.3 Constrained Optimization

Once a design has been entered into the Synopsys internal database via HDL Compiler, Design Compiler is used to modify the structure or instances of logic in a netlist by performing logic-level and gate-level optimizations. The compilation can be driven by various constraints, either design rule constraints or optimization constraints. Design attributes can also be attached to ports, nets, or the entire design and guide the optimization to meeting the requirements.

Logic-level optimizations operate on a technology-independent logic representation of the design. If the design is already mapped into a technology, it is unmapped and translated to a set of Boolean logic equations. One type of logic-level optimization is flattening, in which all intermediate variables are removed from the design, yielding two levels of combinational logic. A flattened design can be very fast, but can have a larger area.

Logic minimization can be implemented to reduce the number of size of the product and sum terms in a boolean logic circuit representation. Karnaugh maps and the Quine-McCluskey algorithm are two manual methods for logic minimization. Synopsys also supports single and multiple output minimization. Phase assignment is then used to reduce circuit area based on the results of flattening and minimization. Implementations of each logic equation are compared to implementations of its complement, and the form that can be implemented with the smallest component area is chosen, taking into consideration that the complemented form requires an extra inverter at the output.

Another type of logic-level optimization is structuring. Structuring does the exact opposite of flattening in that it adds intermediate variables and logic structure to a design. This results in a smaller, but slower circuit. Structuring is incremental, which means that it operates on the existing structure of the circuit and only changes it when improvements can be made. After flattening and structuring, the design can be mapped to a gate-level netlist, given user-defined and technology-specific constraints, as well as design attributes.

Since the design may be part of a larger system, information about the subdesign must be provided to the higher level, so that optimization can be performed at that level. This information can be extracted after the subdesign has been optimized, or, attributes can be attached to the subdesign and passed to the higher level. These attributes include input and output delays and with respect to

a specified clock, input signal arrival time, drive strength for input ports, external loads for input and output ports and fanout loads for output ports. This information provides a characterization of a subdesign viewed from the outside and is used to drive optimization while preserving the state of the subdesign itself.

Design constraints define the goals of the synthesis process for a given design, such as performance and area. Performance-driven optimization usually takes precedence over area considerations. Design-rule constraints reflect restrictions that must be met for a design and are specific to the target technology and not set by the user. Optimization constraints are the design goals that are user-defined, and may not be crucial to circuit operation. Meeting design rule constraints is essential, and Design Compiler gives precedence to these specifications. For each type of constraint, a cost function is calculated every time a change is made to the circuit.

The design mapping phase, in which a design is compiled into a gate-level netlist, is divided into two phases. During the first phase, Design Compiler tries to reduce the optimization cost function. In the second phase, an attempt is made to fix all design-rule violations without affecting the optimization constraints. If this cannot be done, then Design Compiler fixes all design-rule violations at the expense of optimization constraints. Maximum transition time at a cell input pin, maximum fanout load and maximum net capacitance are the primary design-rule constraints.

The optimization constraint cost function consists of maximum delay cost, minimum delay cost and maximum area cost, in that order of importance. Every optimization step that improves maximum delay cost is taken. Each step that improves minimum delay cost, but does not adversely affect maximum delay cost is taken. Optimization continues until either each cost function is zero, or no further improvement to the overall cost function can be made.

The maximum delay targets for each path are determined by considering the clock waveform, external delays, maximum delay attributes on output ports, setup times on library cell pins, wire load model, and load and drive conditions for the cells in the design. Design Compiler has a built-in static timing analyzer that evaluates timing constraints by calculating all path delays between cells, including interconnect delay estimations. The timing analyzer performs critical path tracing for every path in the design. The longest path delay in violation of the maximum delay target alone determines the maximum delay cost. The minimum delay cost is affected by all paths that have delays less than their target minimum delay constraints. These violations manifest themselves as hold time violations and must be fixed by the compiler. Maximum area cost is simply the difference between the current area and the maximum area constraint. Setting the maximum area constraint to zero yields the smallest circuit area.

There is a lot of flexibility available to the designer through the use of design constraints and attributes. Moving through a hierarchical design is made

simple with Design Analyzer's graphical user's interface, and Design Compiler reports back information to the designer such as constraint violations, maximum and minimum paths and area. If the design needs to be altered, attributes can be changed and the design can be fully compiled, starting from the logical level. A design can be incrementally compiled as well, meaning the existing gate-level netlist is considered, and is changed only when improvements in the optimization cost function are possible.

Once a satisfactory design is attained, the design can either be simulated at the gate level to verify that the functionality has been retained through the compilation process, or the design can be enhanced with testability features using Synopsys' Test Compiler. As mentioned in the previous chapter, a testable design is very important, especially for a very large scale digital circuit.

3.2.4 Testability Design

Synopsys' Test Compiler helps a designer synthesize a highly testable design in very little time by performing test circuitry synthesis and insertion, and Automatic Test Pattern Generation (ATPG), which generates high fault coverage test patterns for the scan-based design. Test Compiler also supports the addition of JTAG Boundary-Scan Standard circuitry. Combinational ATPG is used to produce scan patterns for full scan designs, yielding high stuck-at fault coverage. Test Compiler also allows partial scan synthesis to save resources.

The first step when adding test structures to a design is to check the test design rules. This feature of Synopsys provides early feedback to the designer on the testability of the circuit, so that decisions can be made that will ensure a highly testable design. The test design rule checker also preprocesses a design by flagging all valid sequential cells that can be replaced by a scannable equivalent. These equivalent scan cells must be defined in the target technology library.

Test Compiler uses symbolic simulation when checking design rules, which means that the concept of a scan element is generalized to include any sequential element that can be controlled and observed. It is up to the designer to choose which scan path methodology is best suited for the design, such as multiplexed flip-flop, LSSD or clocked scan. During the pattern generation phase of testability design, the design rule checker effectively removes all scan circuitry, partitioning the circuit into blocks of combinational logic. Then, the pseudoprimary inputs and outputs are defined so that the combinational ATPG algorithm can generate appropriate test patterns.

The next step is to actually insert the scan circuitry. Test Compiler removes all sequential cells that are flagged for replacement and replaces them with appropriate scannable equivalents. The existing inputs and outputs of the cells are preserved, and the dedicated test pins are left unconnected. Test Compiler then routes the common signals of the scan cells, such as the test clocks and scan enable. The scan cell inputs and outputs are then connected together to form a

scan chain. The routing order can either be automatically or manually assigned. Test Compiler allows for more than one scan chain, which may be necessary if multiple phase clocking is present, or if the scan chains are longer than a specified constraint.

After scan circuitry is added, the design should be re-optimized using Design Compiler to ensure that the design still meets area and performance specifications. The circuit area increases because scan cells are larger than non-scan cells, and there is an increase in routing complexity due to the routing of the scan chain and the scan enable and clock signals. Performance also suffers due to the electrical characteristics of the scan cells which lead to longer delays than for the non-scan cells. Design Compiler evaluates the new design with the old design constraints intact, and if the design fails to meet the constraints, the designer must either relax the contraints or restructure the design.

If desired, JTAG boundary-scan circuitry can be added to a design. Standard logic is added automatically, such as the Test Access Port controller, instruction register, bypass register and boundary-scan register. Boundary-scan cells are placed between the primary inputs and the core circuitry, and linked together to form the boundary-scan chain. Dedicated JTAG pins are automatically added to allow access to JTAG test structures. The designer can define instructions and their encoding patterns, and the instruction register is synthesized automatically to reflect the desired configuration. A JTAG instruction determines which data

register is to be connected between the TDI and TDO pins to form a serial shift scan chain. This data register can be the boundary-scan register, bypass register, or a user-defined register, such as an internal scan register.

Once the test structures are in place, Test Compiler implements a combinational ATPG algorithm to generate test patterns for full-scan designs and a sequential ATPG algorithm for partial-scan designs. The designer has some control over the ATPG process. One can limit the amount of time spent covering each fault, amount of effort backtracking from a detected fault, target fault sample percentage and pattern compaction effort. Test Compiler also allows tri-state nets and bidirectional drivers by avoiding bus contention and bus float problems. The test vectors are generated and stored in the Synopsys internal database format, and can be formatted to vendor-specific formats, or HDL simulation files. These are used later to simulate the physical layout of the design.

Automation of the testability phase of circuit design gives the designer the freedom to quickly determine the effects of adding circuitry and choosing a particular scan methodology with respect to circuit performance and area and quality of test. After a reasonable configuration is established, the enhanced circuit is re-optimized to meet optimization and design-rule constraints. The circuit is ready for the physical design phase.

3.2.5 Synopsys-EPOCH Interface

Translation of a design from a source CAD tool to a destination tool requires either direct conversion between internal design databases, or conversion to a standard format, which can be read in by the destination tool and converted to its internal database. The latter is the case for translating a design from Synopsys' Design Analyzer to the EPOCH IC design system. One intermediate format is a Verilog structural netlist consisting of combinational gates and sequential cells from the target technology library. This library, HP26G, is consistent between EPOCH and Synopsys, so the output from Synopsys is a fully implemented design compatible with EPOCH.

EPOCH reads the netlist in and links it to its version of the target technology library, which contains information about ports, functionality, timing, capacitance and resistance, and physical layout structure. EPOCH flattens the hierarchy of the design to a single level, except where a fixed-block status attribute has been specified. The result of netlist input is a collection of composite cells, each with its own internal netlist consisting of leaf cells corresponding to the named parts in the EPOCH library. The design is considered uncompiled at this point, for even though the leaf cells have explicit layout and simulation specifications, the design does not.

3.3 Physical Design

Physical design refers to the process of realizing the design in physical terms rather than relative terms. This involves placing and connecting the components of an optimal manner, then modifying component parameters or enhancing the circuit to meet design specifications. These specifications, or design parameters, are established before the physical design process is implemented and actually drive the process to meet the goals of the designer.

3.3.1 Design Parameters

Power parameters affect power estimation, and therefore, affect the width of the power rails in the circuit. The main power parameters are default clock frequency, ambient operating temperature, default switching frequency, maximum allowable voltage drop, and current limitations. The clock frequency is used to determine dynamic power dissipation. The ambient operating temperature, or the temperature of the air surrounding the device, is used along with power estimations to determine the temperature of the device itself. This affects, among other things, the allowable current density of power rails.

Switching frequency is the percent of clock pulses that components switch logical value. CMOS gates dissipate a significant amount of power only when it switches, so the higher the switching frequency, the higher the dynamic power dis-

sipation. Maximum voltage drop specifies the minimum allowable supply voltage available to a leafcell which ensures correct operation. All of these parameters affect the way EPOCH estimates power dissipation, which, in turn, affects the size of the power and ground rails.

There are three types of timing parameters that are used by EPOCH's timing-driven placement and buffer-sizing algorithms: clock, input and output. Clocks can be specified by a frequency, pulse width, rise and fall time, jitter and relative start time. Clocks are the reference point for other timing parameters. Primary input timing parameters include: minimum and maximum arrival time rising and falling, and rise and fall time. Primary output timing parameters are determined by output load capacitance and maximum delay constraints. These constraints can be transferred from Synopsys to help drive the placement of leafcells so as to minimize signal delay and meet design timing specifications. Buffer-sizing of gates along a signal path is performed wherever it is necessary to speed a signal up to satisfy path timing constraints.

3.3.2 Design Compilation

Design Compilation is the process of placing and routing the components of a circuit in an optimal manner given certain specifications, and the sizing of the transistors of components to fine tune timing relationships and guarantee that components are able to drive their capacitive loads. Placement can be performed

either automatically or manually using EPOCH's FloorPlanner tool. Routing is performed automatically and is affected by timing constraints, net attributes or signal type. Buffer sizing is done automatically or can be driven by user specifications.

Cell placement is defined as the arrangement of a cell in a particular orientation with respect to other cells. A cell can be rotated by increments of 90° if the circuit is laid out using only Manhattan geometry. A cell can be translated in the x or y direction, and can be replaced by a mirrored version of itself.

EPOCH gives the designer the option of automatic placement and manual placement. Automatic placement optimizes for density and minimizes routing using a simulated annealing algorithm, which makes a random series of placement moves, then rejects or accepts them based on their effect on a cost function, which depends on area and route length. In general, only moves that result in an improvement in the cost function are made, although simulated annealing allows for some regression in cost to escape from possible local minima of the function. The process stops after a given number of moves have been made.

Placement can be timing-driven, given certain timing constraints and clock definition. EPOCH's TACTIC identifies critical nets and attaches appropriate attributes to them, then the automatic placement algorithm emphasizes the minimization of critical net length to avoid any potential timing errors.

Placement can also be performed automatically using FloorPlanner, where the designer can make all the placement moves available to the automatic placement. Additionally, the aspect ratio of standard cell groups and RAM and ROM arrays can be changed to better fit the floorplan. One can also change port locations and modify routing channel definitions between the cells.

Routing is performed automatically in two stages: global routing and detail routing. The global router assigns nets to routing channels defined during the placement phase. Essentially, it breaks down a large routing problem into several smaller and easier problems by planning the routing globally to reduce the size of the chip, shorten connection length and delay, and balance routing congestion. Once the global connections are defined, a detail router distributes the nets within channels and assigns routing layers and widths to each net. The result is a design that is completely realized physically, but not necessarily optimal with regard to timing and other constraints.

Special attention must be paid to nets that carry clock signals or serve as power or ground nets. A net with an attribute attached to it, specifying that it carries a clock signal, is treated differently by the router than other signal nets. The clock signal is routed to the center of capacitive load, then balanced branch routes radiate to the clock inputs of each cell. This is called a balanced clock tree and eliminates the possibility of a very long clock net delivering a pulse with too much delay.

Power and ground nets are automatically sized connected to all Vdd and GND ports of cells and sized according to the cells power requirements, which are based on power consumption estimates. A signal net can also have a weight attached to it to give it a higher routing priority than other nets. The net weight is the factor by which the route length, as seen by the router, is multiplied. The router will work harder to route that net at the expense of other routes.

Buffer-sizing is the process by which sizes are assigned to the transistors of leafcells based on load capacitance and timing requirements of the circuit. After changes are made to the buffer sizes, EPOCH regenerates the leaf cells with the new transistor sizes, performs power dissipation estimation on the entire design, and the design is rerouted, since power and ground rail widths may need to be adjusted to account for new power requirements. Buffer sizing may be performed automatically, or may be timing-driven. A buffer file can also be provided by the user to specify the desired buffer sizes. When a design is originally compiled in Synopsys, the netlist contains library cells that are classified by their buffer size as well as their function. When the design is read into EPOCH, the buffer sizes are stripped out of the netlist and directed into a buffer file, which is used during the buffer sizing process.

3.3.3 Packaging

Packaging refers to the selection of a package for the die, specification of the types of pads used, and the port mapping of pad terminals to pins located in the core design. The designer selects a package based on the number of pads required, the area required by the design and the routing from the core to the pads, as well as other factors. There are five classes of pads: input, output, bi-directional, power and ground. Input, output and bi-directional pads have buffers, electrostatic discharge (ESD) protection transistors and guard rings to prevent latchup.

Power and ground pads can be directed to supply the core circuit, the pad ring itself, or both. The pad ring is the configuration of the pads around the core circuit. The designer assigns all of the circuit I/O to specific pad locations, and EPOCH assembles the pad ring, attaches the bonding fingers and connects them to the external pins on the package. EPOCH can also automatically select from a database the available package that best suits the needs of the design. FloorPlanner can be used after the initial packaging to reassign pad locations or choose a different package.

Pads are considered to be library cells by the compiler, and can be specifically configured for the needs of a particular design. For output and bidirectional pads, the output drive is specified, which determines the output current and the dimensions of the pad. Other pads are automatically sized to conform to the

output pad dimensions, for optimal packaging. The width, or pitch, of all pads can also be specified.

3.4 Special Pad Synthesis

EPOCH does not provide pads that are compliant with the JTAG Boundary-Scan standard. Extra circuitry must be added to input, output, tri-state output and bi-directional pads that enables switching between normal circuit operation mode and boundary-scan mode. Boundary-scan cells are placed between the pads and the core circuit, and must then be connected together to form a chain. For this particular design methodology, the designer is responsible for adding the scan cells to the pads, thus defining a new set of pads. The pad itself must be altered since any logic added to the circuit otherwise will be considered part of the core, as opposed to the pad ring. EPOCH can then assemble the pad ring using the new pads and route the signals from the pads to the core and from one scan cell to the next.

Instead of designing a set of pad drivers from scratch, the layout of a pad driver compiled in EPOCH can be extracted and modified using another CAD tool, Magic, which is a VLSI layout editing tool. The pad drivers that are compiled by EPOCH should be appropriate for a boundary-scan enhanced pad as well. The drivers and boundary-scan logic are compiled in EPOCH and translated

to Caltech Intermediate Format (CIF) so that it can be read in by Magic.

In Magic, the boundary-scan cells are attached to the input or output of the pad depending on the pad type. Tri-state output pads require two boundary-scan cells, one for the output signal and a control cell for the output enable signal. Bi-directional pads require three boundary-scan cells, one for the input signal, one for the output signal, and a control cell for the signal from the core that enables the output direction.

The new pads are then translated into the GDSII (Calma stream) format, which can be read in by EPOCH. The cell is considered an import cell, and is no longer directly related to the EPOCH library database. Port location and timing attributes must be attached to the pads, as well as the functionality, input pin capacitances and output impedances of the boundary-scan cell, so that EPOCH can perform a static timing analysis on the entire circuit. Figures 3.2 through 3.5 show the physical layout of the input, output, tri-state output, and bidirectional pads, respectively.

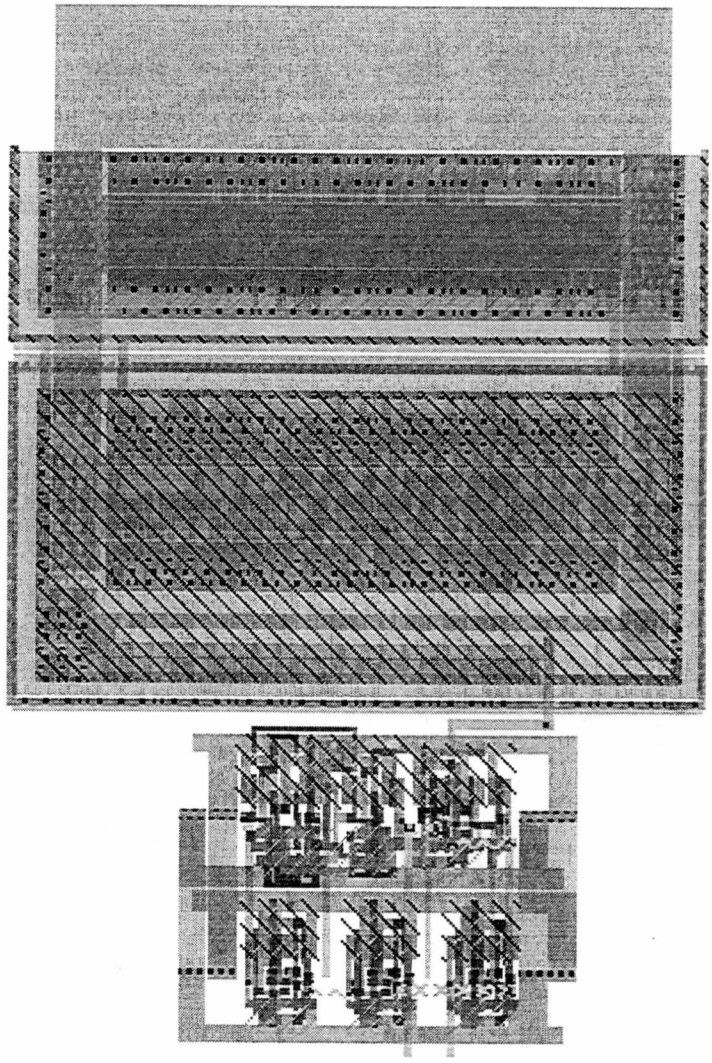


Figure 3.2: *Input pad with boundary scan cell.*

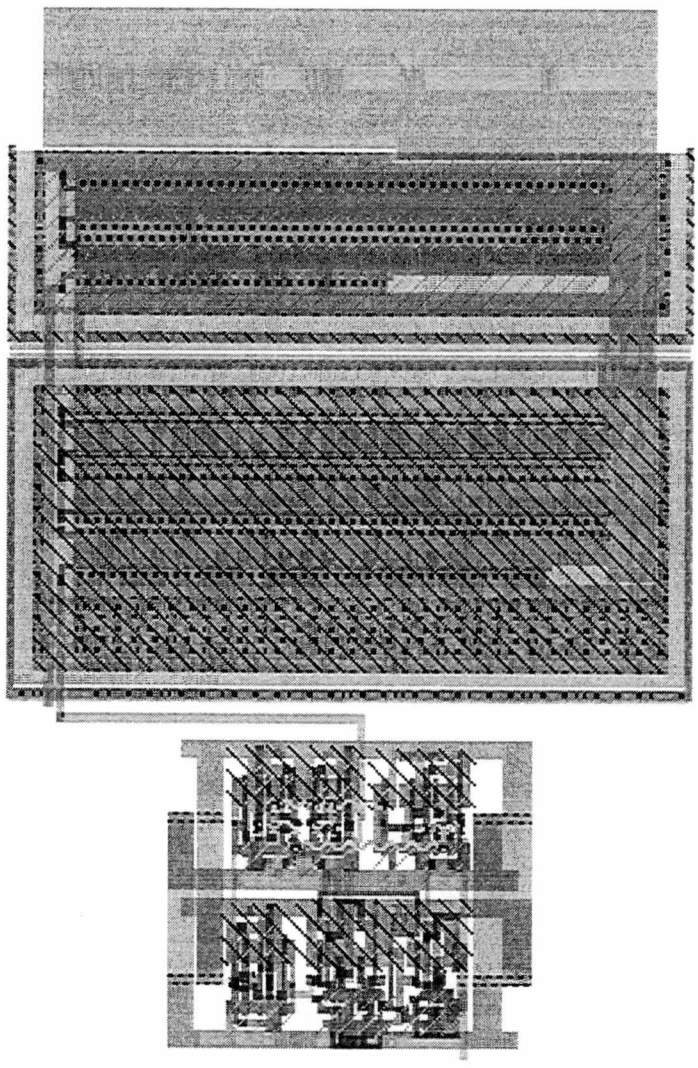


Figure 3.3: *output pad with boundary scan cell.*

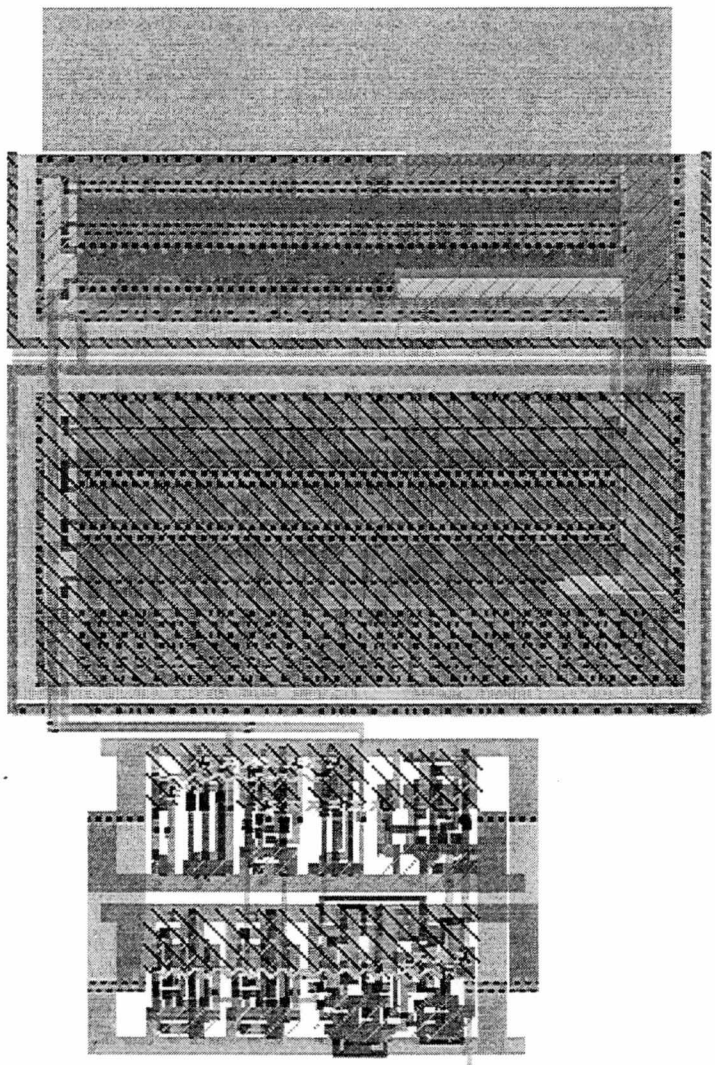


Figure 3.4: *Tri-state output pad with boundary scan cells.*

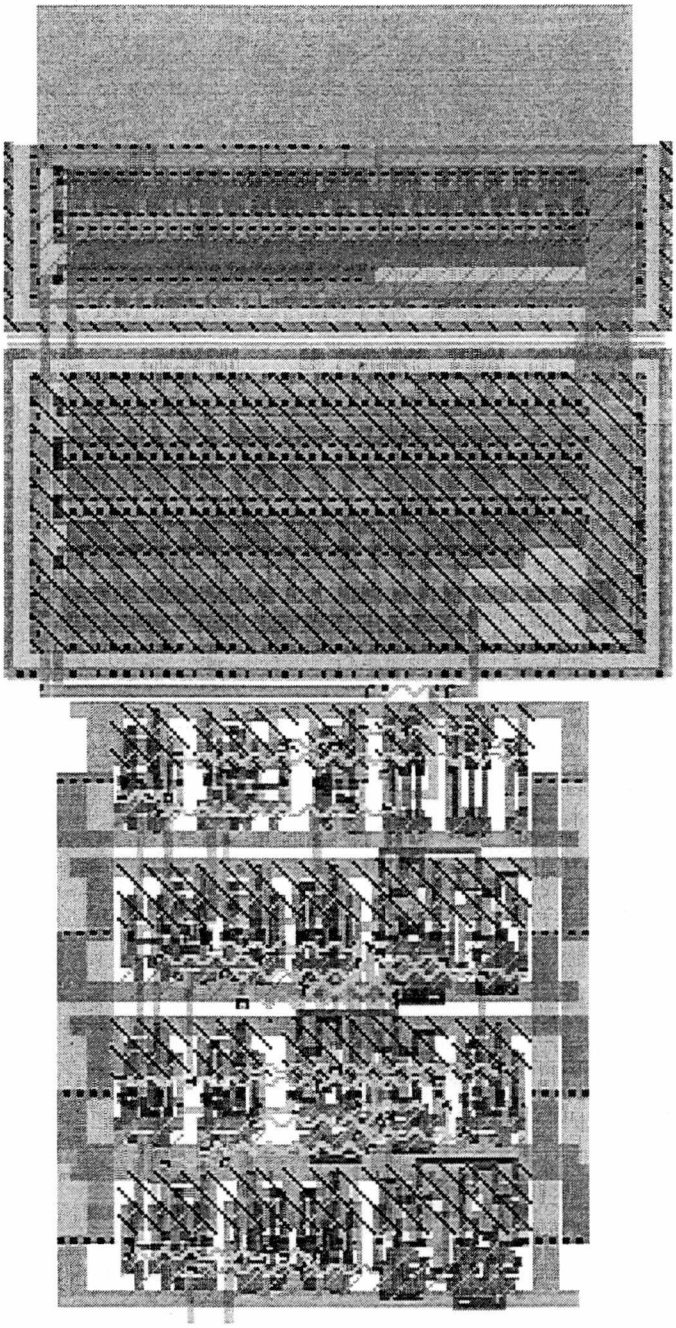


Figure 3.5: *Bidirectional pad with boundary scan cells.*

3.5 Static Timing Analysis

EPOCH has a static timing analysis tool called TACTIC, previously described when timing-driven compilation was discussed. Static timing analysis does not involve the simulation of the circuit, but rather is concerned with the relationship between the clock signal arrival time at every sequential element and the reference clock signal at the clock input port. TACTIC detects hold and setup errors for rising and falling signals and also finds the most critical path, which determines the maximum speed at which the clock can run without causing system failure.

Once critical paths are identified, they can be weighted heavily with respect to other nets, so that when the design is re-routed, it will have a higher routing priority and may be reduced in length, thus possibly increasing the performance of the design. TACTIC can be used as a pre-processor for timing-driven placement and buffer-sizing to eliminate errors paths by identifying critical nets, which are appropriately weighted by TACTIC, and placing cells or sizing buffers in such a way as to reduce path delay on critical nets.

3.6 Simulation

EPOCH does not have an event-driven simulator, but can provide information necessary for simulation to another tool, in this, case Synopsys' VHDL System Simulator. A design in EPOCH consists of primitive cells, each containing information important for simulation such as pin-to-pin delays, input and output capacitance and output impedance, all based on the target technology library (HP26G). Also, interconnect delays are determined based on the topology of the physical layout. EPOCH can gather all the information necessary for a simulation and translate it to a useful form, such as a structural VHDL description. Two separate VHDL files are provided, one containing a lookup table of delay data, the other containing simulation primitive instantiations and interconnection that reflect the structure of the circuit. Each primitive instantiation is parameterized by a range of delay data referenced from the delay lookup table.

Synopsys' VHDL Analyzer is used to convert the VHDL into interpreted simulation files that drive the simulation. A simulation file is created for every instance in the circuit, and high level simulation files link them together. Synopsys' VHDL Simulator, invoked inside the graphical VHDL Debugger, performs the simulation, which is driven by a user-defined command file. Synopsys also has a graphical waveform viewer (WAVES) that allows one to monitor the behavior of the circuit during simulation.

The simulator is also used early in the design cycle to verify that the design exhibits the intended behavior. The simulation is independent of technology, gates are assigned unit delays, and interconnect delay is not considered. Although the timing will not be realistic, the functionality can be verified before any further design steps are implemented. Simulation can also be performed after the design has been compiled and mapped to the target technology, but not implemented as a physical layout. The delays will be more realistic, but interconnect delays are only estimated. This simulation verifies that the intended behavior of the circuit has been retained through the compilation process. The designer can be more confident that the physical design process will yield a circuit that meets all the specifications.

In a nutshell, the design flow is as follows: design capture, simulation, compilation, simulation, physical design, simulation. Each simulation provides feedback to the designer that is useful at that point in the design flow. This allows for quick turn-around if errors are detected, as it is probable that only the step prior to the simulation that caught the error will have to be repeated. In the next chapter, the design methodology is exercised using a small example.

CHAPTER 4

Tutorial

4.1 Objective outline

In this chapter, a small example design, the AM2910 microprogram address sequencer, will be taken through the design methodology discussed in the previous chapter. The focus of this chapter is on the synthesis and physical design of the circuit. The design was captured in VHDL at the structural and behavioral level. Synopsys' VHDL analyzer and debugger was used to verify the correct functionality of the design, using an appropriate simulation file containing known input vectors corresponding to resulting known output vectors.

The tutorial begins with the compilation and mapping of the design into the HP26G technology. The design will then be placed and routed, and packaged in a standard 64 pin package. After compilation, the design will be verified, and various figures of merit determined, such as critical path and area.

The effects of designing the circuit for testability will then be determined. The design will be compiled, internal scan circuitry added, then boundary scan

circuitry. Figures of merit will again be determined at this point. After physical implementation and packaging the figures of merit will be re-evaluated and the overall merit of design for test will be determined. Of course, correct circuit functionality during normal mode and scan mode will be verified as well.

4.2 AM2910 design specification

The AM2910 is a microprogram address sequencer based on the device of the same name manufactured by Advanced Micro Devices, Inc. A block diagram of the AM2910 is shown in figure 4.1. The chip consists of 5 functional units: multiplexor, register/counter, microprogram counter, stack, and control. The AM2910 provides the address of the next microinstruction to the processor. The specific function of each of the five modules is described as follows:

- Multiplexer - The multiplexer has four 12-bit inputs, and selects from the output of the register, stack, uPC, or from direct data input (D). The output of the multiplexer (Y) is the next microprogram instruction address.

- uPC - The microprogram counter contains a 12-bit register, preceded by an incrementer. A carry-in bit (CI) tells the incrementer to either add one to the uPC input, which is the current output of the multiplexer (Y), or to just pass the value unchanged. The register will then hold either the next sequential instruction address, or the current address.

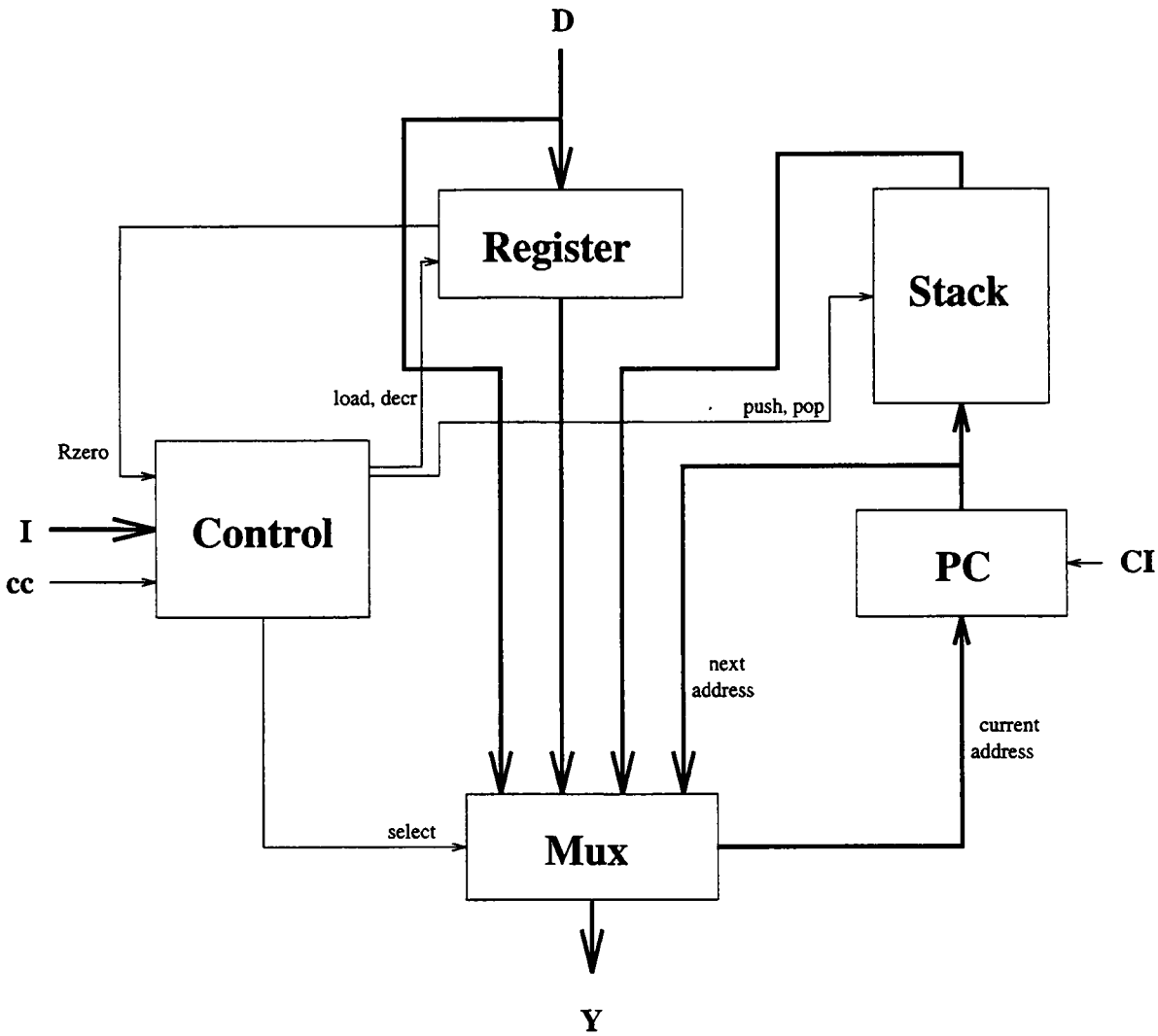


Figure 4.1: *Block diagram of the AM2910 Microprogram Address Sequencer.*

- Register - The register module contains a 12-bit register, and when its load control signal (RLD_BAR) is active new data is loaded via D on the next positive clock edge. This module also contains a down counter, which decrements the value held in the register by one. This feature is used for loops, or the execution of a sequence of instructions a known number of times.

- Stack - The stack is a 5-word by 12-bit register file referenced by a 3-bit stack pointer. When a loop or subroutine is executed, the return address is pushed onto the stack. Nested loops or subroutines require pushing addition return addresses on to the stack. When the program exits the loop or returns from a subroutine, the return address is popped from the stack and the stack pointer is updated.

- Control A four-bit instruction (I) is decoded into one of sixteen instructions that determine the function of the circuit for the next clock cycle. A condition code bit affects conditional instructions like conditional jumps. For each instruction, either PL_BAR, MAP_BAR or VECT_BAR will be active. When PL_BAR is active, the value contained in a pipeline regsiter is made available at D. When MAP_BAR is active, the address provided by a mapping PROM is made available at D. Finally, when VECT_BAR is active, the value provided by direct addressing is available at D.

4.3 Baseline synthesis

The design synthesis begins with the conversion of the VHDL description of the design into Synopsys' internal database format. The AM2910 design consists of the five modules described earlier, which are combined into a single design using a top-level VHDL structural netlist. The five modules have to be read into Synopsys before the top-level VHDL, since it depends on all lower levels in the design hierarchy.

Figure 4.2 shows the hierarchical view of the design. The icon for the top-level design (am2910mod) includes a gate symbol, meaning that it is a strucutral netlist. The other icons corresponding to the five modules include an equation, which means that they are described by boolean equations. Pushing into the top-level design, a symbol view is created, showing the top-level pinout (see figure 4.3). A schematic view is also created (figure 4.4), showing block symbols for the five modules, their interconnections, and the top-level ports.

The five modules each have a symbol and schematic view as well. The schematics for the modules are a generic interpretation of the logic structure based on signal flow and Boolean representation. Specifying a target technology is necessary before compiling the design into a gate-level netlist. Synopsys chooses gates that are available from a library of cells provided by the vendor of the target technology.

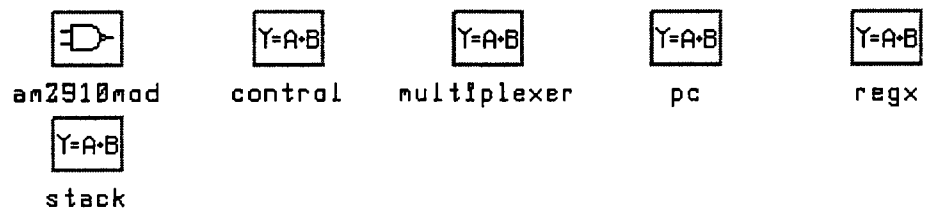


Figure 4.2: *Hierarchical view of the AM2910.*

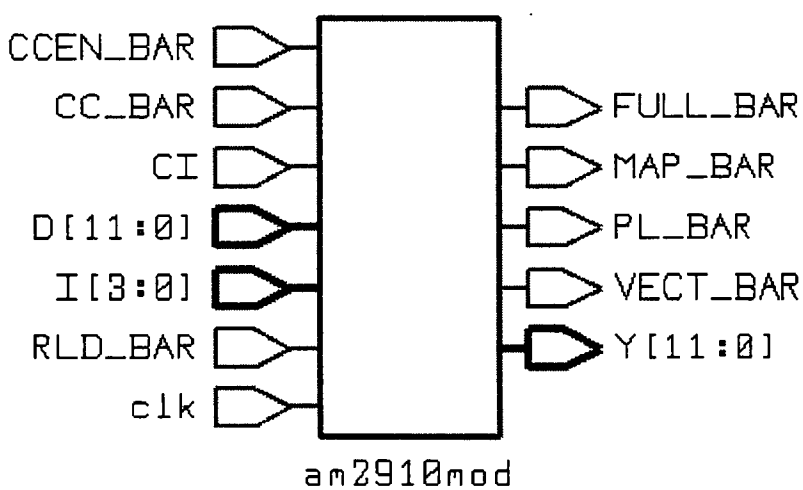


Figure 4.3: *Symbol view of the AM2910.*

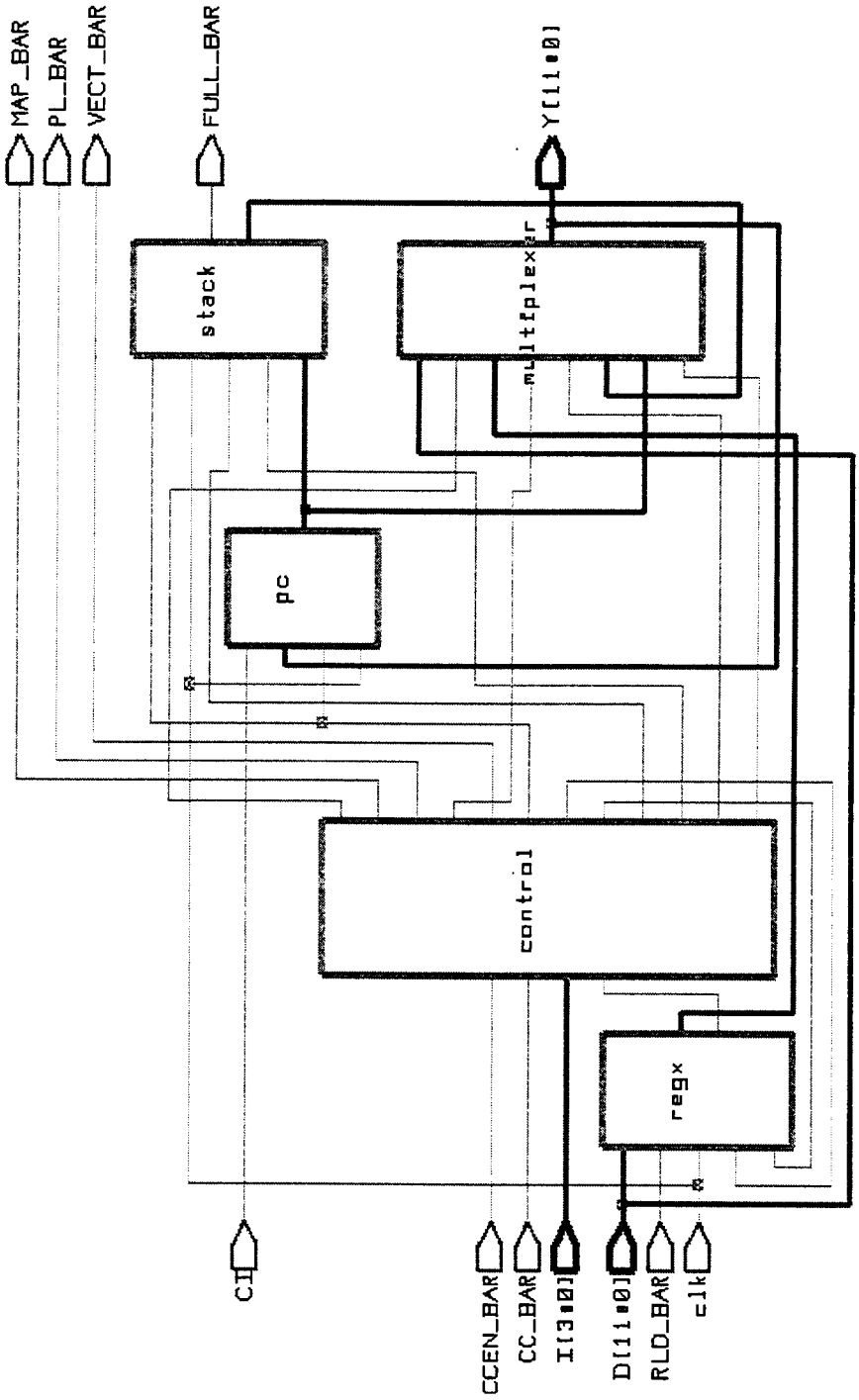


Figure 4.4: *Schematic view of the AM2910.*

Each module can be compiled separately, which allows one to apply different constraints to each if necessary, or the entire design can be compiled at once. The hierarchy of the design may be retained or the design can be flattened to a single level. Maintaining the hierarchy may be required if the layout of the design requires the separation of the logic into discrete parts, or it may just be a matter of convenience when managing a large hierarchical design that may be flattened later in the design cycle.

In this case, the AM2910 will be flattened into a single group of standard cells from the HP26G 0.8μ CMOS technology library, but only when it is placed and routed with the EPOCH tool. The hierarchy is maintained in Synopsys so as to distinguish the modules from one another. Timing-driven structuring is performed during compilation, which yields a smaller design, since product terms in the output Boolean equations are shared amongst outputs if possible. Savings in area sometimes lead to a slower circuit, so care is taken not to violate timing constraints placed on the design. Figure 4.5 shows the result of the compilation of the *regx* module into a gate level netlist.

Constraining the AM2910 design involves setting up an input clock signal, specifying loads on the output ports, placing delays on the input ports, and establishing operating conditions. In the HP26G technology library there are three sets of operating to choose from: best case, typical and worst case. Typical operating conditions were chosen for this design, where the compiler has to account

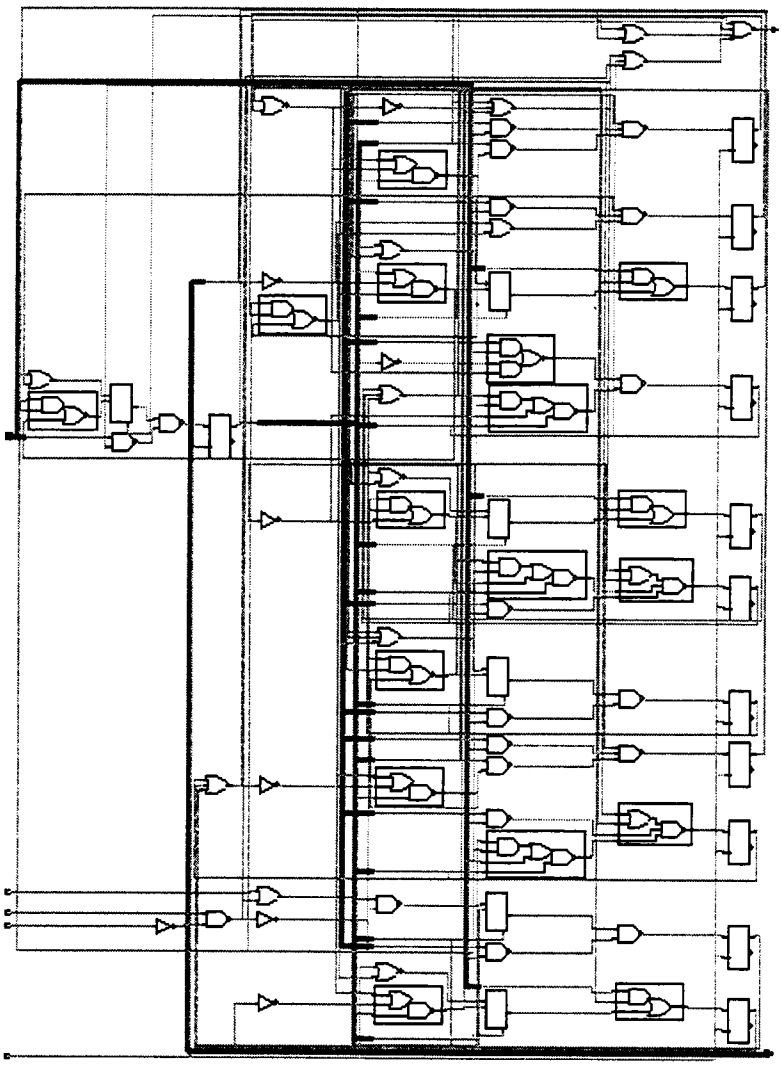


Figure 4.5: *Gate-level schematic of the regx module.*

for small variations in semiconductor manufacturing processes, moderate ambient operating temperature, and uses an interconnect model where the loads on the nets emanating from a node are assumed to be on separate, equal branches of wire. A typical timing range is also specified, where multiplicative constants affect the maximum and minimum path delays, modelling the effects of process variations.

A typical wire load model is selected to account for the fact that actual wire lengths cannot be determined until the design is physically implemented later in the design cycle. Resistance, capacitance and area per unit length are specified and multiplied by the estimated length of a net to yield total lumped resistance, capacitance and design area due to the net. Synopsys uses these estimations to evaluate the impact of different optimization possibilities on the area and performance of the design.

Each gate in the HP26G library comes in several different buffer sizes, which correspond to the switching speed of the gate - the larger the buffer, the faster the gate switches. A large buffer compensates for a large capacitive load on the output of the gate, which would otherwise cause significant delay in the propagaion of a signal through that gate[18]. The buffer size actually refers to the size of the pMOS and nMOS transistors in the gate, with the p-transistor size affecting the rise time through the gate, and the n-transistor size affecting the fall time. Synopsys takes into consideration the clock setup and timing path constraints when deciding how to size the transistors of the gates along each path[18].

The clock for the AM2910 is set at 40MHz. No input delays were applied and the output loads were left unspecified because all the output signals drive output pads, and later on in the design cycle, EPOCH will insert buffers between the output signals and the pads. The critical path for this design is a path between two flip-flops, and the path delay from the clock input of the first to the data input of the second is about 18.2ns. The maximum allowable time for this path is equal to the clock period less the skew and the setup time for the second flip-flop, and is about 23ns. So, there is no setup violation for the critical path and the design is sound enough to proceed to the physical implementation phase.

The optimized technology-specific gate-level netlist is written out in structural verilog by Synopsys, and is read in by EPOCH. EPOCH parses through the netlist and copies all of the unique parts that exist in the design into the local design database. These parts correspond to physical cells with geometry as well as electrical specifications. The netlist is then translated for the entire design, and EPOCH preserves any hierarchy that exists for the design as well as the buffer sizes calculated by Synopsys.

Constraints are now placed on the design in EPOCH much the same way they were in Synopsys. The clock is established, output loads and input delays are defined. Although the buffer sizes calculated by Synopsys do reflect what is necessary for satisfactory circuit performance, EPOCH will resize the buffers again to reflect the timing demands of the physical layout.

EPOCH also estimates power dissipation for the circuit, which is used to size the power rails that supply the circuit. Ambient temperature, signal switching factor percentage, maximum allowable voltage drop, DC current limitation and maximum simultaneous switching current are all factors set by the user which affect power dissipation calculations. Once these variables have been set, in this case to default values, then automatic compilation can be performed on the design with no manual intervention. This entails placement, routing, buffer sizing and rail sizing. Since the AM2910 is entirely comprised of standard cells, no manual placement of cells is required. Figure 4.6 shows the layout produced by EPOCH for the AM2910 design.

The AM2910 design is ready to be packaged. There are 21 input signals and 16 output signals for a total of 37 required signal pads. Also required are ground and power pads to supply the core and the pad ring. The number of ring-only supply pads is about one of each kind for every eight signal pads. This rule of thumb helps limit the effects of ground bounce, which occurs when several signals switch values at the same time, causing a surge in current through the ground lead supplying them. Having pads that supply only the pad ring protects the core circuit from the effects of these surges. In this design, 4 each of ring-only and core-only power and ground pads are used, for a total of 16 supply pads and a grand total of 53 pads.

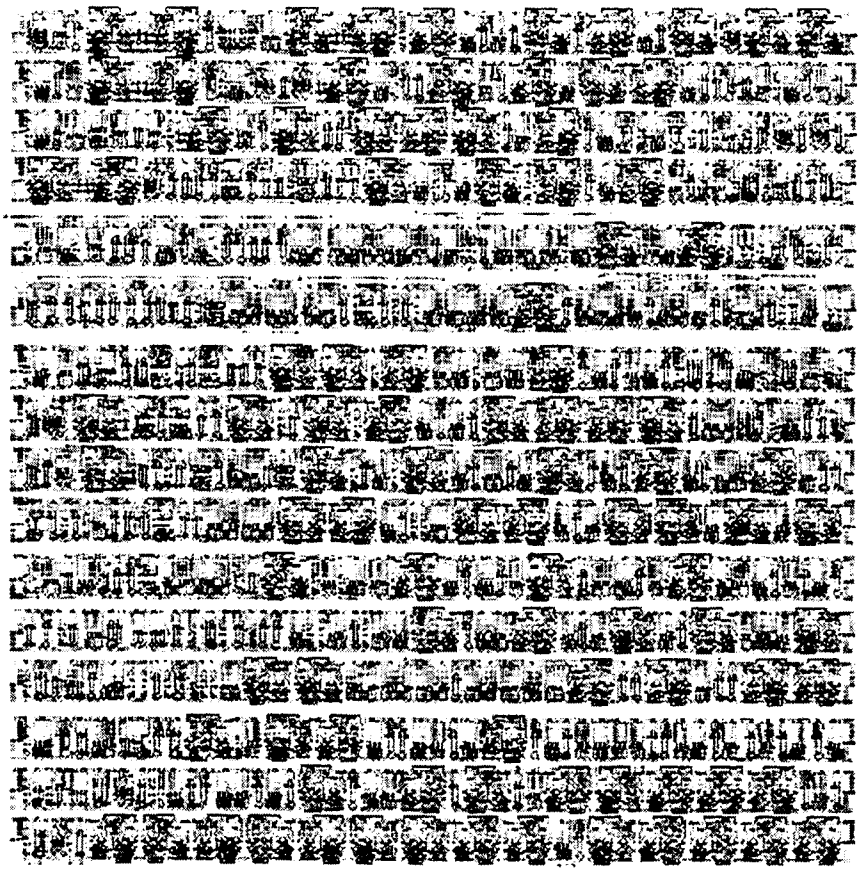


Figure 4.6: *Physical layout of the AM2910 design.*

EPOCH provides a small selection of standard packages to choose from. The most appropriate available package for this design is a 64 pin side-brazed ceramic package with alumina substrate and a cavity area of 1.032 cm^2, given that there must be at least 53 pads and the core circuit is about 0.394 cm^2 . The only information about the package that pertains to EPOCH's packaging process is the the pin number and cavity area. The packager can be directed to place the pads radially or in a staggered configuration. The packager must also adhere to set bonding rules, which affect how much freedom there is in manipulating the pad ring when trying to fit the core and core-to-pad routing into the package.

The AM2910 core must be instantiated in a top-level structural verilog file, along with all of the input, output and supply pads. The pads are part of the EPOCH library and several user-defined options exist for each type of pad. Input pad attributes include a Schmitt trigger for protection against noise and distortion immunity, TTL interfacing, and different pad pitches. Output pad attributes include different pad pitches based on drive strength, as well as a tristate output option. The pin number corresponding to the package specification is also attributed to each pad. An attribute is attached to the core logic that has already been compiled so that EPOCH does not recompile it when packaging the chip. EPOCH automatically includes buffers in between each core signal origin or destination and their corresonding signal pads, then sizes these buffers to account for the potentially long wire leading to the pad. Figure 4.7 shows the packaged AM2910 design.

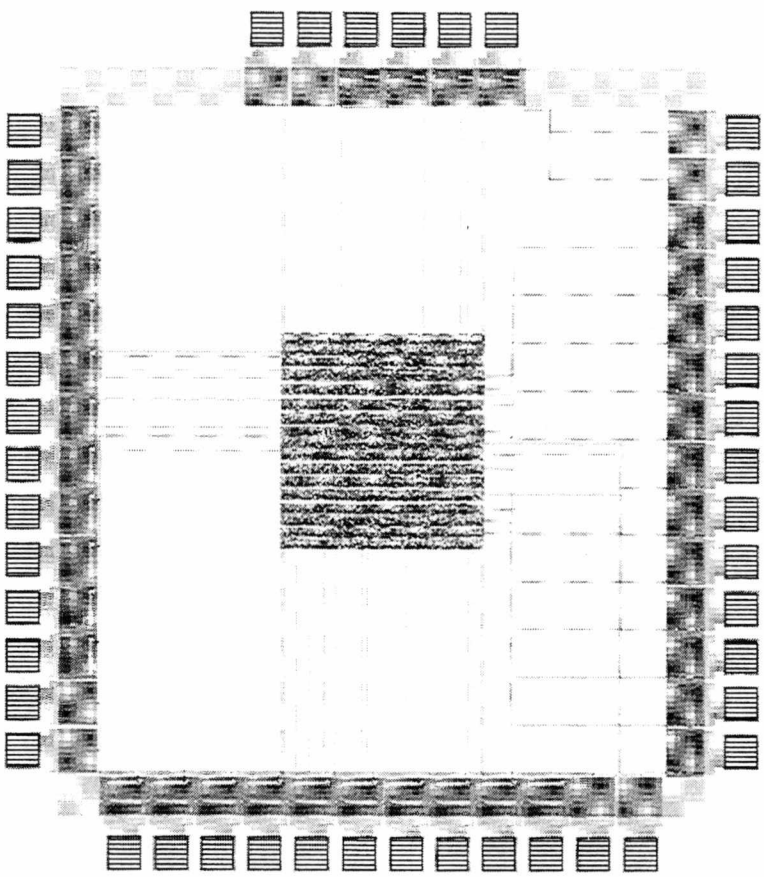


Figure 4.7: *The AM2910 design in a 64 pin package.*

4.4 Testable Design

As mentioned previously, this chip is part of a large system, and it will probably be integrated with the rest of the system on a board or an Multi-Chip Module. Testing the AM2910 chip independently will be impossible unless it is equipped with JTAG boundary-scan logic to provide access to all input and output signals. This allows access to internal test structures used to test for logic faults, and board-level interconnection testing can also be performed.

The first step in adding testability to the design is to insert an internal scan chain into the circuit to provide observability and controllability at the appropriate points, which allows for testing for manufacturing faults given known input patterns. For this design, a full internal scan methodology is implemented, meaning all sequential components are made scannable and linked together to form a scan chain. Synopsys' Test Compiler swaps the non-scan sequential cells for their scannable equivalent, then routes them in such a way as to minimize affect on the area of the design as well as performance. Figure 4.8 shows the new symbol for the AM2910 with scan enable, scan input, and scan output pins.

Once the scan chain is in place, Test Compiler runs an ATPG algorithm consisting of a random pattern generation phase, then a deterministic pattern generation phase to reduce execution time yet still obtain high fault coverage. ATPG resulted in 158 patterns that detect 4680 out of 4681 total possible faults,

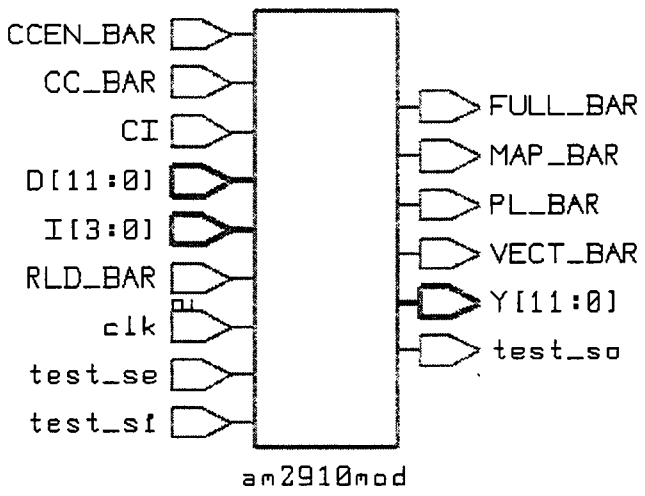


Figure 4.8: *Symbol for AM2910, with scan pins.*

or 99.98% fault coverage. Each pattern consists of four parts: one vector to apply to the primary inputs of the circuit, one vector to scan into the internal scan chain, one vector to match the pattern scanned out of the scan chain and one vector to match the pattern observed at the primary outputs. The longest vector in this case is the one shifted in or out of the scan chain. There are 99 scan cells in this design, so 99 bits must be shifted in, then out, for every pattern. Later, this will prove to be the primary contribution to total test time.

In order to utilize the internal scan structures, boundary-scan logic needs to be added to the chip so that signals are accessible at the board level[14]. Several components are added: the TAP controller, instruction register, bypass

register, and boundary-scan cells for each input and output signal. Synopsys' Test Compiler automatically adds these components and allows the designer to define their own instructions, which affects the type of instruction register to be selected for the design. An instruction is defined as a bit pattern and tells the instruction register which scan chain to select when that particular instruction is encountered. In this case an INTSCAN instruction is defined, telling the instruction register to select the internal scan chain for the purpose of shifting in bit patterns.

Synopsys synthesizes the logic and includes a control signal that must be manually attached to the scan enable pin of the AM2910. This control signal is also used to disable the system clock and enable the test clock. The test clock is controlled by the JTAG_TCK pin, which allows the tester to step through the 16 different states of the TAP controller. When the TAP controller is in the SHIFT-DR state, the scan chain is enabled and the test pattern is shifted in serially on each rising edge of JTAG_TCK. Figure 4.9 shows the schematic of the boundary-scan enhanced AM2910 circuit. The shaded symbol in the center is that of the AM2910 core logic. Figure 4.10 shows the new top-level symbol for the AM2910. There are a total of 5 new pins, all associated with the JTAG circuitry: JTAG_TCK, JTAG_TMS, JTAG_TRST, JTAG_TDI and JTAG_TDO.

The boundary-scan cells associated with each signal pad will be added to the pads as was discussed in the previous chapter. The rest of the JTAG circuitry and the AM2910 core logic are written out by Synopsys into a gate-level verilog

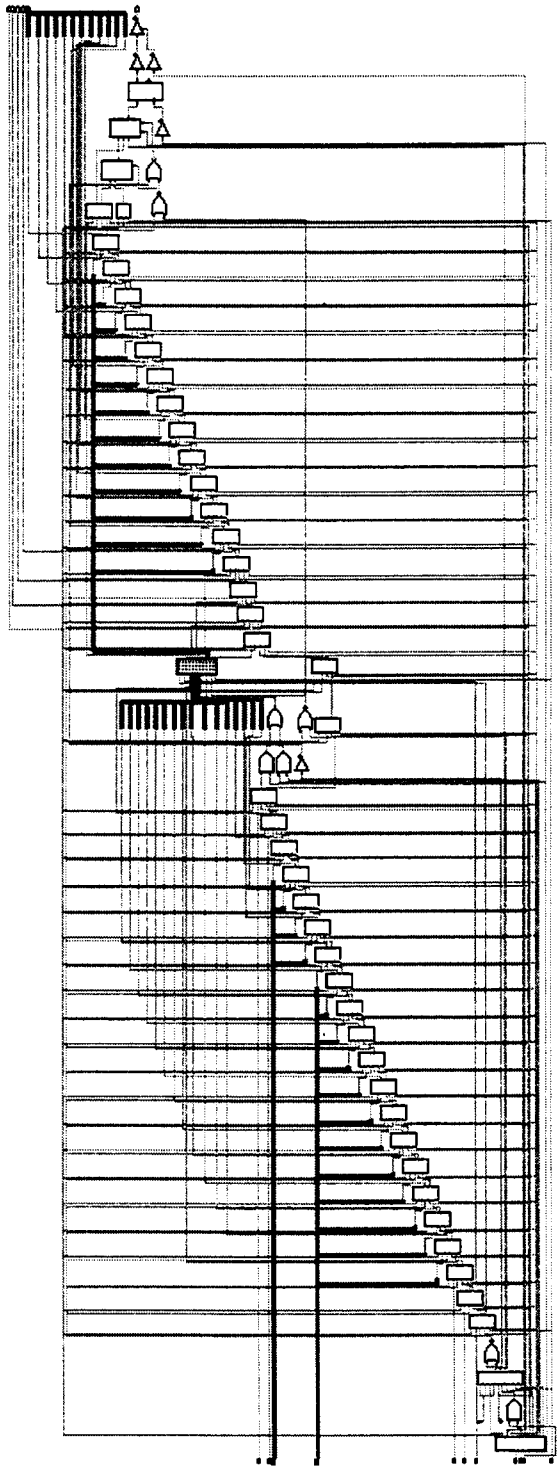


Figure 4.9: *Schematic of boundary-scan compliant AM2910 chip.*

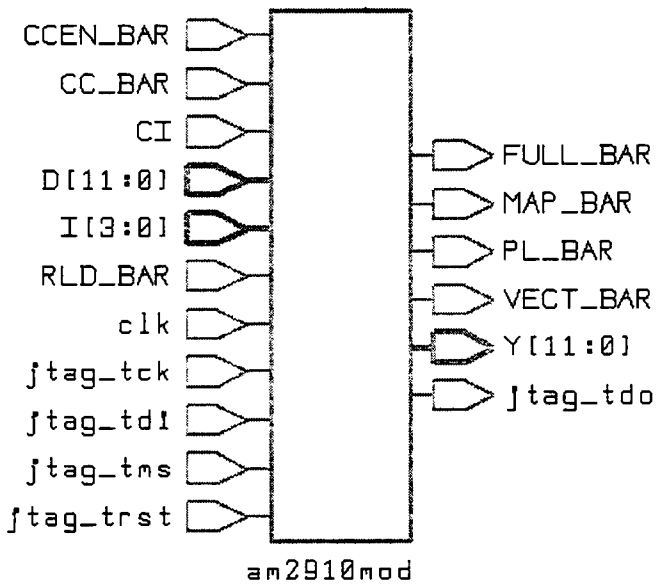


Figure 4.10: *Symbol for boundary-scan compliant AM2910 chip.*

netlist. The design is then translated into an EPOCH design much the same was it was previously. The boundary-scan compliant design requires five new pads, giving a total of 58 pads. After the core circuit is placed and routed, the area of the design is determined to be 0.567 cm^2, which is less than the maximum cavity area allowed by the 64 pin packaged used before, so it will again suffice for this design. Instead of using pads generated by EPOCH as before, the new set of pads equipped with boundary-scan cells are used in the pad ring. The pads conform to EPOCH's geometry specifications and have an attribute that tells the packager that they are import parts to be used as pads. Figure 4.11 shows the layout of the new packaged chip, with the new boundary-scan compliant pads linked together to form the boundary-scan chain.

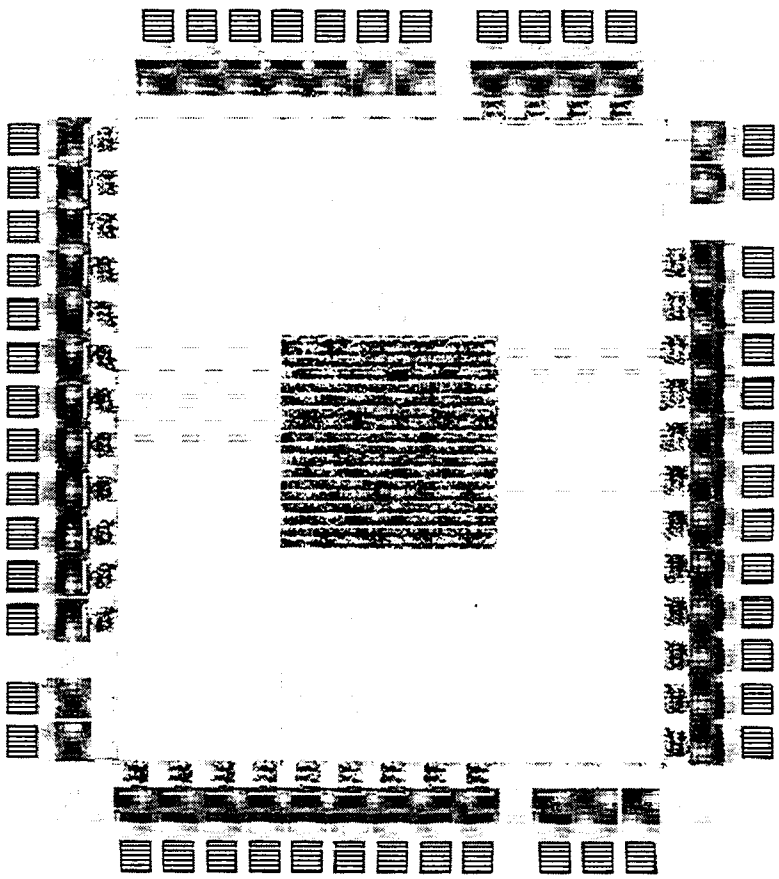


Figure 4.11: *Boundary-scan compliant AM2910 chip.*

The simulation is performed on the final version of the chip to make sure it functions in boundary-scan test mode, internal-scan test mode and normal operation mode. Selecting boundary-scan mode involves enabling the JTAG circuit and shifting in the instruction on the JTAG_TDI pin that corresponds to selection of the boundary-scan chain. The serial pattern is then shifted in on the JTAG_TDI pin and the parallel pattern can be applied to the primary inputs.

Applying ATPG test patterns involves enabling the JTAG circuit and shifting in the instruction that corresponds to selection of the internal-scan chain. The first internal pattern is then shifted in, then the input pattern is applied to the primary inputs of the chip. Not all of the inputs may be accessible at the board level, so the parallel input pattern must be applied using the test structures of the other chips that connect to the device under test. This can be achieved by setting other devices in boundary-scan mode and shifting in the parallel input patterns in the appropriate places so that they are applied to the device under test. After the first pattern is applied, the primary output pattern is captured by other devices that are in boundary-scan mode. This means that each chip's TAP controller must function independently from the other chips[2]. Obviously, a test protocol must be established for each type of test on each device, involving all test structures on the board or MCM.

EPOCH writes out the design as a VHDL files comprised of simulation primitives for every component in the design. Synopsys' VHDL Analyzer is used

to assemble the interpreted simulation for the design. The testbench used to test for faults contains the patterns generated by Synopsys' Test Compiler, formatted into VHDL, as well as the necessary setup using JTAG control signals. The simulation should show a correct primary output vector after the internal and primary input patterns are applied. The values shifted out of the scan chain through the JTAG_TDO pin, while the next internal pattern is shifted in, should also match the pattern generated by Test Compiler.

4.5 Results

The benefits of adding test circuitry to a design are straightforward. Not only can the die be tested before it is integrated with the rest of the system, but it can also be tested after it has been attached to the board or substrate because the test structures allow access to its inputs and output. Testing for manufacturing faults assumes that the circuit functions correctly. The defect level is decreased when test structures are present because most manufacturing faults that exist can be detected.

Adding test circuitry also increases the area and power consumption, and may also be a detriment to the performance of the system. The critical path in a circuit may have more delay due to added test logic along the path. Table 4.1 shows comparisons between the testable design and the original version.

Table 4.1: *Comparison between original AM2910 die and the testable version.*

	Original Die	Test Enhanced Die
core area (mm^2)	0.453	0.650
chip area (mm^2)	6.480	7.969
Total Power Dissipation (mW)	37.73	48.25
Critical Path (ns)	23.57	24.00
Pad count	53	58

The most significant effect of adding test circuitry is on die area and power dissipation. The die area increased 23% and the power dissipation increased 27.8%. The critical path increased slightly, however, the critical path delay can be improved through targeted optimization techniques. Since the designated clock period is 40Mhz, any path delay under 25ns less the setup time for a flip-flop clock pin, which for this technology is about a half of a nanosecond, will not cause a setup violation. Increased pad count had little effect in this case, since the same package was used, but in other situations, may force the designer to use a larger, less area-efficient package to accomodate the extra pads.

CHAPTER 5

Large Design Example

5.1 SPARC Design Specification

This CPU is an implementation of the SPARC (Scalable Processor Architecture) Version 8 RISC architecture[16], which was specified by the SPARC International Architecture Commitee. This committe is comprised of representatives from several corporations, who are all investing in the architecture and are trying to ensure that every implementation of the SPARC maintains a certain level of compatibility. Another goal is to dissuade exclusive ownership of a standard architecture, which allows a corporation to control an entire technology base and the market developed for their products. This impementation was provided by Sun Microsystems in the form of register-transfer and structural level verilog.

5.1.1 Integer Unit

The integer unit contains the general purpose registers and is responsible for overall processor operation by controlling the instruction pipeline. All integer

arithmetic instructions are executed, including integer multiply and divide, and addresses for control transfer instructions and memory loads and stores are computed. Program counters are saved for instructions in the pipeline, and operands are bypassed to the register file or ALU to avoid pipeline stalls or read-after-write hazards in the case where one instruction uses an operand that is the result of a recent instruction.

All floating point instructions are decoded by the IU. The FPU has its own interface with the instruction cache, but awaits a strobe from the IU that tells it that the next instruction is a floating point instruction. The IU initiates all data cache access for floating point loads and stores and handles cache misses as well. If the FPU detects a structural hazard, such as two floating-point multiply instructions occuring to close together, it will cause an interlock of the IU instruction pipeline until the new floating point instruction is cleared for execution.

A register file with 120 registers indexed by a 3 bit window pointer is used for general purpose storage. An addition 8 registers are used for global variable storage for a total of 128 registers. The IU handles traps according to their priority during the Write pipeline stage. When a trap is encountered, all instructions following it in the pipeline are flushed out, the processor status, program counter and next program counter are saved into registers, and the window pointer is decremented to point the next set of 24 registers in the register file. The trap routine starting address is addressed by a base register indexed by the trap type.

When the processor returns from the trap routine, the status of the processor is restored to the pre-trap state and execution continues.

The layout of the IU consists of 6 modules: *Mexec*, *Mir*, *Mpc*, *Mdecode*, *Mregfile* and *Mdcacheif*. *Mexec* is the datapath for execution stage of the pipeline, including the ALU, operand multiplexors, condition code generation logic, and ALU output register. *Mir* includes the instruction register pipeline, the datapath for the register file source and destination registers and the opcode. *Mpc* is the datapath of the program counters for each stage of the instruction pipeline, as well as an address adder for branch instructions. *Mdecode* contains all instruction decode logic. *Mregfile* is the register file with 2 read ports and 1 write port. *Mdcacheif* contains data cache address registers and incrementer, and selects between IU and FPU data.

5.1.2 Floating-Point Unit

The floating-point unit executes all floating-point operate instructions and use floating-point load and store instructions to retrieve or store operands. Operands and results can be either single or double precision. A 32×32bit register file is present for operand storage, so it can store 32 single precision values or 16 double precision values.

A floating-point operation takes one clock cycle in the IU, and then execution responsibilities are passed to the FPU. The instruction is decoded and is used to address a microinstruction control store ROM, which controls the floating-point execution pipeline. For load instructions, the address is sent to the IU, and the FPU waits for the IU to acknowledge the arrival of the data from main memory. For store instructions, the address and data are sent to the IU, which completes the store operation.

The FPU consists of seven modules: *fp_ctl*, *fp_fpc*, *fp_rom*, *fp_rf*, *fp_frac*, *fp_exp* and *fp_rw*. The *fp_ctl* module controls the three execution datapaths: the 58-bit fraction datapath, the 13-bit exponent datapath, and the one bit sign datapath. The *fp_fpc* module interfaces the FPU with the IU and handles top-level control signals. The *fp_rom* is the 256×64bit microinstruction control store unit that provides directives for each phase of floating point operation execution. The *fp_rf* module is the floating-point register file with 2 read ports and 1 write port. The *fp_frac* module is the datapath for the fractional part of the data and includes an adder and a multiplier with a 4-stage carry-save adder. The *fp_exp* module is the datapath for the exponent and includes an adder. Finally, the *fp_rw* module is the read/write interface between the FPU and the floating-point register file.

5.1.3 Memory Management Unit

The memory management unit performs virtual address translation, memory protection, virtual memory management, and arbitrates memory access between the data and instruction caches, translation lookaside buffer (TLB) and peripheral devices. Several status and control registers in the MMU affect the operation of the CPU.

All memory references within the processor have 32-bit virtual addresses and must be converted to 28-bit physical addresses before a page in main memory can be accessed properly. Virtual addressing allows a processor to distribute memory amongst several processes to avoid conflicts that may lead to costly main memory block replacement. The virtual address constists of three index fields and a page offset that correspond to the lower 12 bits of the physical address.

A virtual address translation is accomplished by walking through a context table and 1 to 3 levels of page tables, all of which are found in main memory. The context table address is stored in the Context Table Pointer Register (CTPR), which is written by software, and is indexed by the context register. The context register contains the current address space identifier, which tells whether the processor is user or supervisor mode, whether access are data or instruction accesses, or if an internal CPU register is being read or written to for diagnostic purposes.

The memory word indexed by the context register and CTPR is a physical address that points to the location of the first level page table in main memory. The page tables are indexed by the 3 index fields of the virtual address, and the memory words being pointed to either contain the physical address of the next level page table, or the physical page number (high order 19 bits) of the page being accessed in main memory. The fully translated physical address consists of the physical page number and the page offset. If a valid physical address is not found by the end of the page table walk, then an exception is generated and the referenced page must be loaded into main memory.

Each successful address translation is stored in the translation lookaside buffer, which is a fully associative cache for virtual address translations. The TLB is checked every time an address needs to be translated, in case it contains the full translation of that address. The index fields of the virtual address, along with the context and mode of the current process are used to match the tag field of the TLB, which is physically implemented by a content-addressable memory (CAM). If there is a TLB tag match, the associated data word is read from the TLB data field, implemented by RAM. If there is a TLB miss, then the address is translated as described previously. When a new translation is stored in the TLB, a pseudorandom replacement scheme is used to write the new data into the TLB CAM and RAM.

The MMU consists of three modules: *dp_mmu*, *m_mmu_cache*, and *mc_tlb*. The *dp_mmu* module is the virtual address datapath, in which several multiplexors are used to form the virtual address, the virtual tag for TLB matching, the conditions for address translation bypass, and loading of the Synchronous Fault Address Register, which holds a faulting virtual address. The *m_mmu_cache* contains MMU datapath control logic and control for the instruction and data caches. The *mc_tlb* contains the translation lookaside buffer and control logic.

5.1.4 Data Cache

The data cache is a 2KB, direct mapped cache and is virtually addressed, but physically tagged. On a load or store access to a cacheable page of main memory, a portion of the virtual address is used to address the data cache data array and tag array, which are both implemented by RAM. Meanwhile, the same virtual address is translated into a physical address in the MMU. The 16-bit tag retrieved from the tag array must match the upper 16 bits of the physical address for there to be a cache hit.

Upon cache read hit, the requested word is returned directly to the IU or FPU. Upon cache read miss, a 16-byte block is read from main memory, beginning with the requested word, followed by the other word of the first double-word, then the next double-word, wrapping around the 16-byte boundary if necessary. The block in the cache is filled, one word at a time, and the requested word is streamed

to the IU or FPU.

Upon cache write miss, the data is written only to main memory, following a "no-write allocate" policy[10]. There are two 32-bit write buffers, in case a double word is to be written. Upon cache write hit, the data is written to the cache and to main memory, following a "write through" policy[10]. These two write policies result in more bus traffic, as data is written to main memory for every write instruction. However, cache coherency is maintained, meaning that for a particular process and address, the cache and main memory contain identical data, if the data cache contains the block being addressed. This ensures that when a peripheral device requests data from main memory, it will retrieve the most current data. There is a valid bit associated with every 32-bit word in the data cache. It is set when data is filled into the cache and is cleared when there is a cache flush, memory parity error, or upon power-up.

5.1.5 Instruction Cache

The instruction cache is a 4KB, direct mapped cache and is virtually addressed, but physically tagged. The instruction cache data array and tag array are addressed by a portion of the virtual address, while the same address is being translated to a physical address in the MMU. The upper 15 bits of the physical address must match the tag for there to be a cache hit.

Upon cache hit, the 32-bit instruction word is read and transferred to the IU. Upon cache miss, a 32-byte block is read from main memory, beginning with the requested word, followed by the other word of the first double-word, then the next double-word, wrapping around the 32-byte boundary if necessary. The block in the cache is filled, one word at a time, and the requested instruction word is streamed to the IU. There is a valid bit associated with every word in the instruction cache. It is set when a block is filled into the cache and is cleared when there is a cache flush or upon power-up.

5.1.6 Memory Interface

The memory interface is divided into three sections: the Memory Control Block (MCB), the Data aligner and Parity Check logic (DPC), and the ReFresh RAM control (RFR). This module interfaces the CPU with the main memory. Main Memory is organized as a 128MB block of DRAM divided into 4 banks of 32MB. Each bank has a dedicated Row Address Strobe (RAS) line, and each uses the same two Column Address Strobe (CAS) lines, one for high word and one for low word. The DRAM is addressed by a multiplexed Row/Column address bus, and there is a 64-bit data bus to increase bandwidth, and 1 parity bit per word to reduce cost. The interface was designed for a Single Inline Memory Module (SIMM) with 16 4-bit wide DRAM devices to provide the 64-bit data bus, plus 2 1-bit wide DRAMs to store the parity bits.

The MCB controls the DRAM and keeps track of the priorities of the different types of memory operations: data read, data write, instruction fetch, read-modify-write sequences, TLB access, IO reads and writes, and DRAM refresh requests. A state machine is used to arbitrate the requests, generally giving highest priority to MMU requests and lowest priority to DRAM refresh requests.

The DPC transfers data between the 64-bit external memory bus and the 32-bit internal memory bus, as well as parity generation and checking. If the memory access is to a byte or halfword, the DPC will align the data properly. It also provides storage space during read-modify-write cycles when a byte or halfword is written to in memory. These cycles are needed because main memory is word addressed, so the smallest segment of data accessed is 32 bits. The 64-bit external bidirectional data bus is split into two 32-bit segments and registered for read data and write data (a total of 4 32-bit registers). The data is then multiplexed onto the 32-bit internal bidirectional data bus, depending on which word was requested for the operation.

The RFR generates refresh requests signals to be sent to the MCB. A synchronized down counter is used to generate requests in regular intervals, depending on what the system clock period is. The RFR initializes the DRAM on power-up as well, waiting for about 200μsec followed by 8 DRAM CAS-before-RAS refresh cycles. The three modules that comprise the memory interface are grouped together in the physical layout.

5.1.7 SBus Controller

The SBUS is Sun Microsystem's local bus specification, and it connects peripheral devices with the CPU and memory subsystem. Standard features include dynamic bus sizing, atomic (no interrupts) load/store, bus arbitration and watchdog timer are supported. The SBUS Controller arbitrates SBUS requests, translates virtual address for Direct Virtual Memory Access (DVMA), enables slave cycles where the slave device controls the SBUS, and controls programmed I/O (PIO) transactions between the CPU and peripherals.

There are three types of data transactions that can occur on the SBUS: PIO, local DVMA, and bypass DVMA. PIO transactions require the SBC to act as SBUS master and allow the CPU to execute transactions to peripheral devices. Local DVMA transactions allow a peripheral to act as SBUS master, allowing it to execute transactions to the CPU or other local resources. Bypass DVMA transactions allow a peripheral to act as SBUS master, allowing transactions to other peripherals. Both types of DVMA transactions require the MMU to receive and translate a virtual address from the SBUS master, then the type of DVMA, whether local or bypass is determined from the physical address. Then the actual transaction takes place during what is known as a slave cycle.

The SBC handles bus errors and timeout errors. A bus error that occurs during a bypass DVMA causes the SBC to intercept the slave cycle from the

slave target peripheral in order to terminate the transaction and cause an error that is recognized by the CPU. Watchdog timeout errors occur when an internal timer expires while waiting for a PIO slave cycle or bypass DVMA slave cycle to complete, in which case the cycle is terminated with an error. The SBC is appears as a single module at the top-level of the SPARC.

5.1.8 Clock Controller

The clock controller generates all clock signals used by the SPARC core circuit, as well as the clock used by external SBUS interface devices. The controller is a 4-bit state machine clocked by an external clock input running at 80MHz. The low-order 2 bits are a free-running down counter. The MSB indicates whether the clocks are stopped or running. The system clock, *ss_clock*, runs at 40MHz and is equal to the LSB logically ORed with the MSB. The external SBUS device clock runs at 20MHz and is equal to bit 1 logically ORed with the MSB. If the clocks are stopped, then the state machine is either in the first half of the 20MHz SBUS cycle or the second, and this is indicated by bit 2 of the state variable. The only way to set the MSB is to scan in an appropriate value into a clock control register, which is done for diagnostic purposes. Another clock signal called *di_val* is a half-period delayed version of *ss_clock*, and is used in only a few areas of the circuit that require a delayed clock. All clock signals are buffered because of the massive load on each line.

5.1.9 Miscellaneous

The miscellaneous module contains reset and JTAG boundary-scan control logic. Most registers in the SPARC are reset to zero when the reset signal is asserted. Since the reset line is so heavily loaded, the output of the clock controller is disabled, thus allowing the reset signal to propagate without causing any timing errors. The JTAG control logic includes all boundary-scan logic except for the boundary-scan registers. If the circuit is in internal-scan mode, a global scan enable signal, connected to all scan cells, is activated, and the output of the clock controller is disabled. The circuit is then clocked according to the state of the TAP controller, which is controlled directly by the JTAG control signal inputs.

5.1.10 Module Test Protocol

There are three types of modules in this design, each with their own testing considerations. Control and other random logic are implemented using elements from the HP26G standard cell library. The logic that makes up the datapath is highly structured and is implemented using elements from EPOCH's library of datapath macros. These macros are indexed by the width of the input data bus, and are comprised of an array of identical bit-slice cells operating on each bit of data. Memory elements, such as RAM, ROM and register files are also synthesis macros provided by EPOCH. The dimensions of the memory array are provided by the user and EPOCH assembles it, as well as row and column address decode

logic and sense amplifiers.

Random logic is tested using the aforementioned serial scan techniques. A module consisting of random logic may exist on a chip that has other types of modules that were not taken into consideration during test generation. Some signals to or from the module may not be controllable or observable at the chip level, since they may only connect to other modules on the chip. The manufacturing test vectors generated for the module are to be applied to the primary inputs or observed at the primary outputs of that module, so access points must be inserted into the circuit that are transparent during normal operation, but allow one to shift in the appropriate portion of the input pattern, or shift out the appropriate part of the output pattern.

Figure 5.1 shows a random logic module with extra registers that allow access to points that are not accessible at the chip level. When the circuit is in test mode, the internal scan pattern is shifted in through the internal scan chain. At the same time, the part of the input pattern to be applied to the previously inaccessible inputs are shifted in through a separate scan chain. Then, the rest of the input pattern is applied to the appropriate input ports. The system clock is then toggled, capturing the result of the current state of the circuit. Part of the primary output pattern is immediately available at the output ports, the rest is captured in the new registers placed at the previously inaccessible module outputs. This data, along with the current state captured in the internal scan chain are

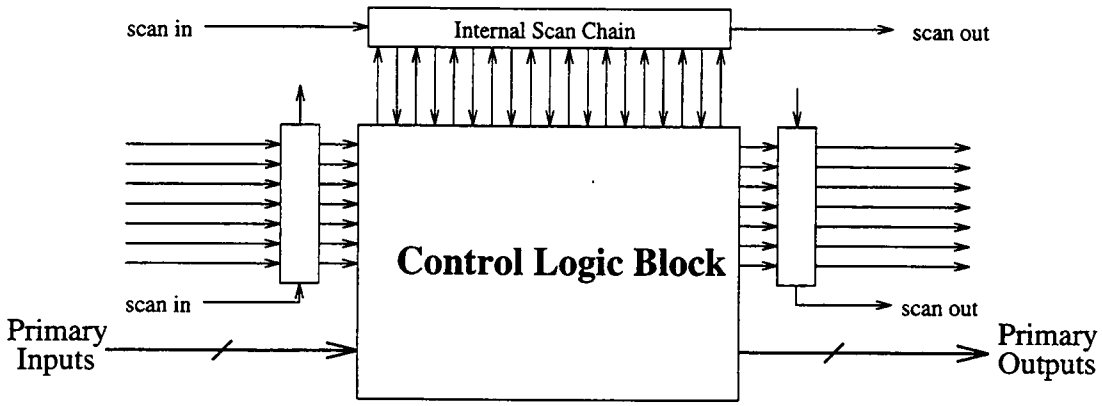


Figure 5.1: *Control block with controllability/observability access shift registers.*

shifted out serially.

Datapath logic also requires the insertion of access points so that when the circuit is in test mode, the datapath control signals can be set directly, and the input and output of the datapath can be controlled or observed. Vectors for the datapath are not generated automatically, but rather an attempt is made to write a one and a zero to each register element, as well as to determine if shift registers, adders, and other datapath elements produce the correct output given certain input and control values. Figure 5.2 shows a datapath block with the appropriate transparent shift registers inserted for test purposes.

Blocks of memory logic require control over address and control inputs, as well as data input and output. Figure 5.3 shows a memory block with the appropriate test logic inserted. The scan cells inserted to give access to the output

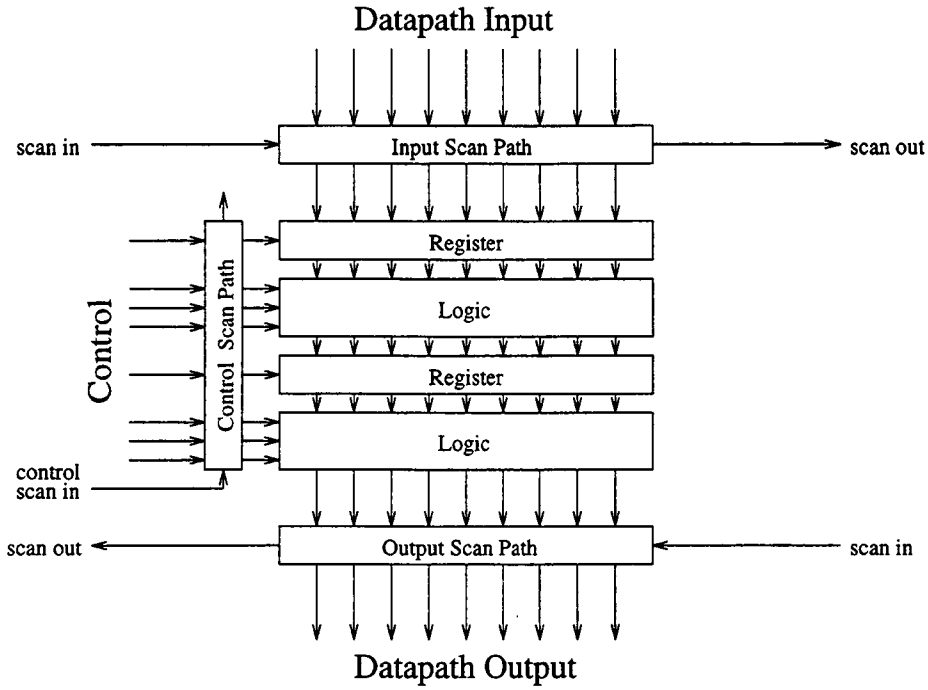


Figure 5.2: *Datapath block with controllability/observability access shift registers.*

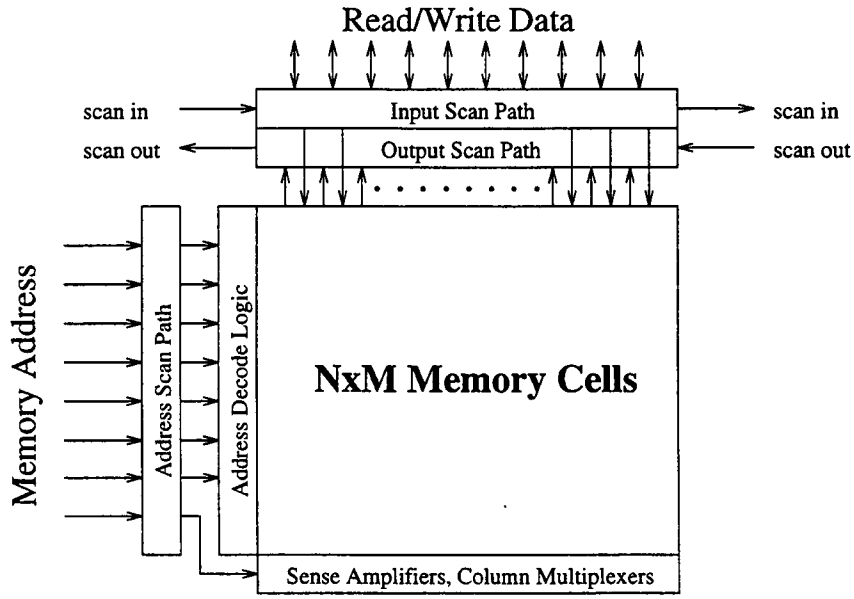


Figure 5.3: *Memory module with controllability/observability access shift registers.*

of the random control logic can also be used to control addresses, control signals and data input that are connected to datapath and memory modules. The scan cells inserted to give access to the input of control logic can be used to capture the output of memory and datapath modules as well. The manufacturing test vectors generated for the control logic will have to be applied independently from memory and datapath modules. There would be a conflict with the patterns to be applied to the memory and datapath, since the same elements would be used to capture the output of the control block and apply the input to the memory and datapath blocks.

5.2 Single Chip Case

Synopsys is used to synthesize all portions of the design that are implemented in standard cells. The single chip core netlist includes modules that do not exist as a Synopsys design, namely, the memory and datapath modules. These are added to the design in the physical design stage. All standard cell modules are individually compiled and optimized given the constraints of a 40MHz clock and appropriate capacitive loads. A new netlist is then defined containing only these modules, so that Test Compiler can be used to insert an internal scan chain and generate manufacturing test vectors.

Each module is then optimized again to meet timing constraints, given that

logic added for test purposes may affect the performance of the circuit. The buffer sizes that Synopsys calculates for each gate are not that important, since EPOCH will be used to size every buffer in the entire design to meet timing constraints. However, EPOCH cannot modify the structural netlist to meet performance goals. For instance, if a net has a high fanout and is being driven by an AND gate, EPOCH can't insert a tree of buffers at the fanout node to distribute the load. Only the buffer size of the AND gate can be changed to drive the load. The standard cell library has a limited number of buffer sizes for each gate, in the case of an AND gate, it has 6 sizes. One wouldn't even want an AND gate that could drive a very large load, because it would consume a lot of real estate, dissipate a significant amount of power, and present a large load to the gates driving it, since all of these factors increase as the size of the transistors in the gate increase.

During the optimization process in Synopsys, the need for structural changes is recognized, and the netlist is modified when necessary. This netlist allows EPOCH more flexiblity when sizing the buffers. The buffer sizes calculated by Synopsys are used as a starting point for EPOCH, then minor changes are made due to the difference in the nature of the technology libraries of the two CAD tools. EPOCH's database is more precise since it contains detailed characteristics of the physical layout of each gate, and can also calculate parasitic capacitances due to routing more accurately.

Figure 5.4 shows the structural netlist of the entire SPARC design. Some of the block symbols represent memory modules and datapath modules, but are actually black boxes. Other symbols represent standard cell designs and link to a gate-level netlist. The rest of the symbols represent structural netlists containing other block symbols for datapath, memory or control modules. The top level design is flattened, bringing all modules to the same level so that the control modules can be grouped together to form a single module. This allows Test Compiler to consider only these modules when inserting test circuitry and generating test vectors. Figure 5.5 shows the netlist flattened to the module level.

The new group comprising all standard cell modules is ready for Test Compiler to insert full serial scan circuitry, essentially placing a multiplexer in front of every flip-flop. A global scan mode signal is connected to each mux select line. Test Compiler orders and routes the scan chain, connecting the output of each flip-flop to the scan mode input of the multiplexer in front of the next flip-flop in the chain. Special consideration must be given to the clocks in this circuit. The SPARC design has two different kinds of clocks, one running at 40MHz, and another running at the same speed, only in anti-phase. However, in test mode, the clock signal that generates all system clocks is disabled, and the JTAG TCK pin is used to clock the entire chip.

Test Compiler handles multiple clocks in a different manner than it does a single system clock. A clock domain is defined for each clock edge, so if there is

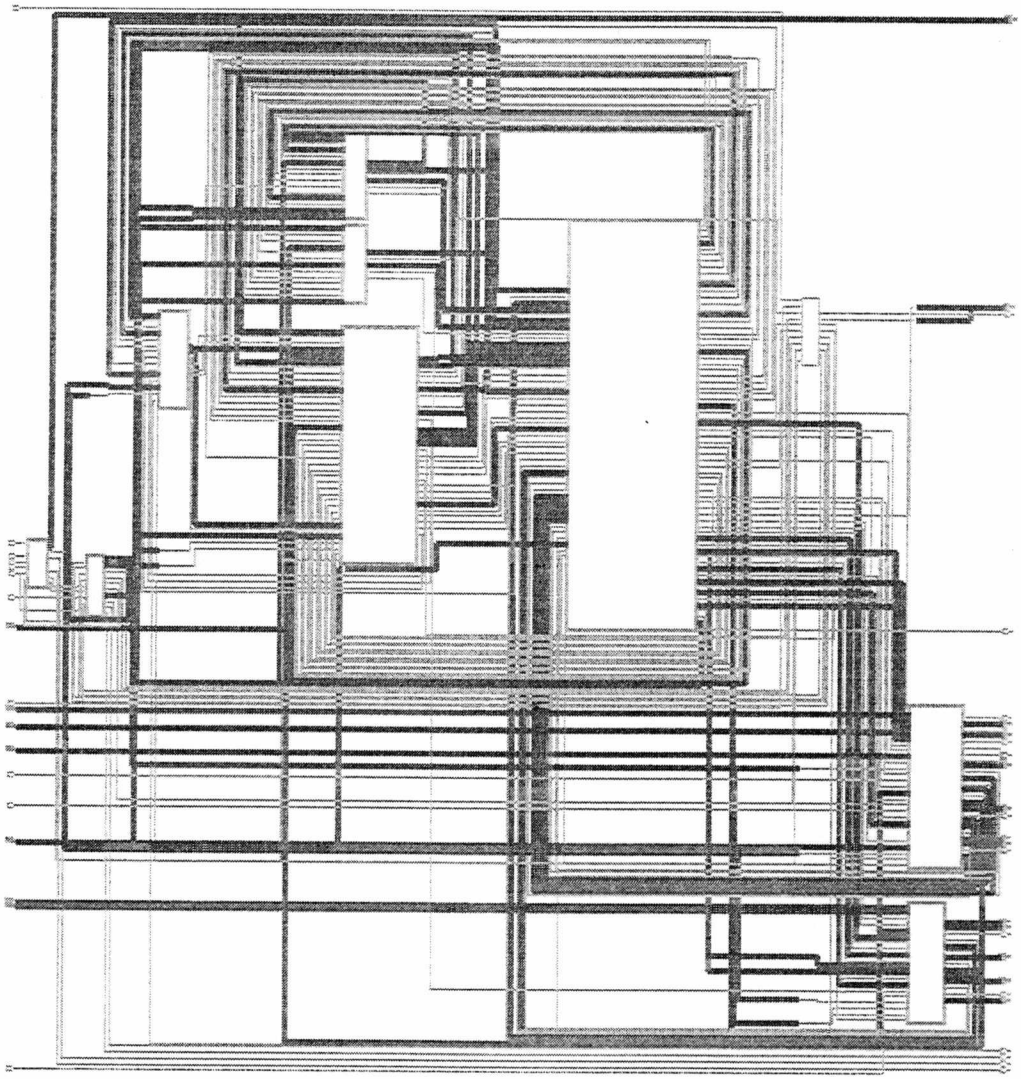


Figure 5.4: *Schematic of SPARC core.*

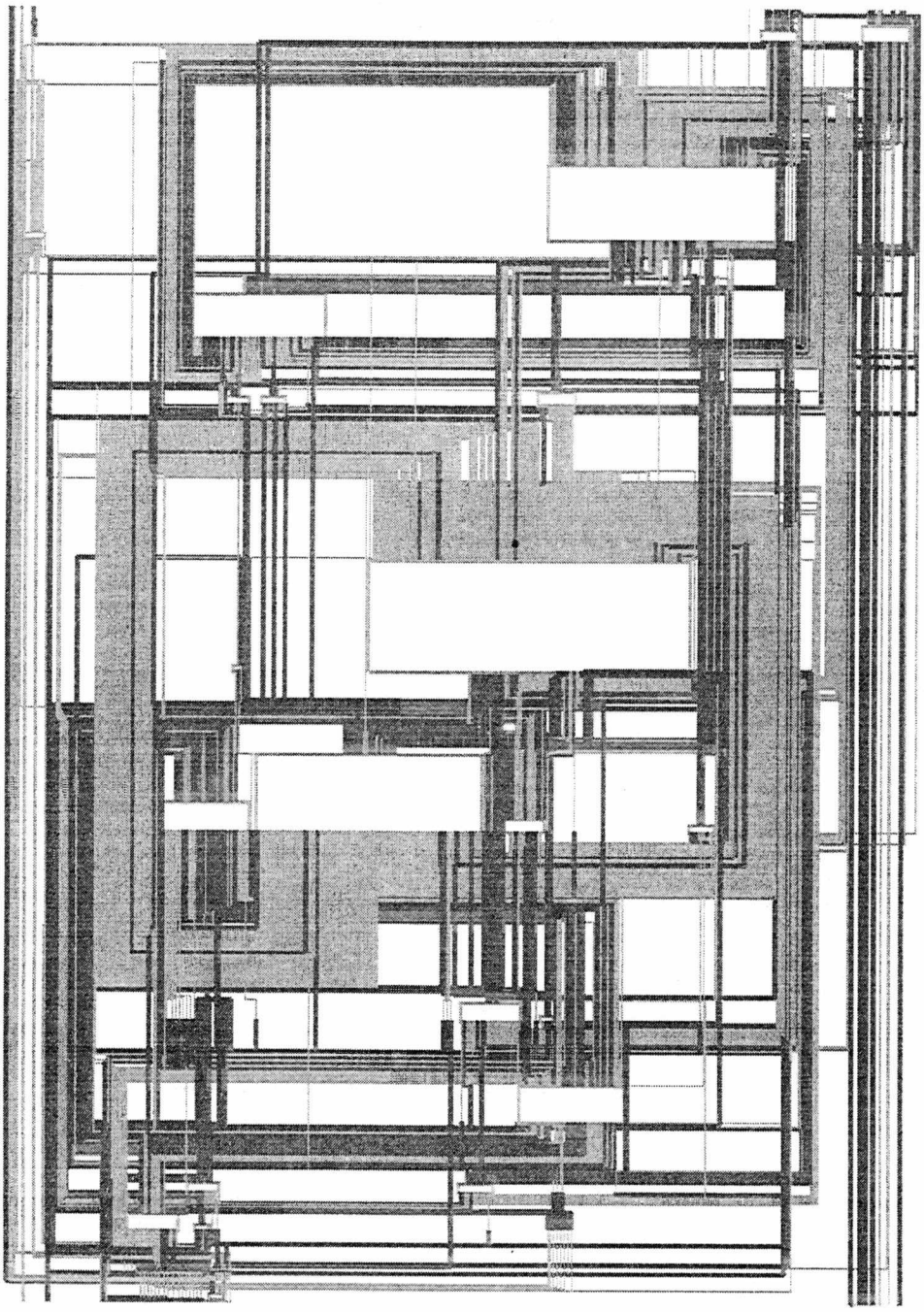


Figure 5.5: *Flattened schematic of SPARC core.*

a clock and an anti-phase clock, there will be two clock domains defined. A scan cell will be assigned to a chain with other cells in the same clock domain. In this case, there would be two separate scan chains. However, in test mode, all clocks are in phase, even though during test mode they are not. A single test clock must be defined on all clock inputs, so that Test Compiler will allocate all cells to a single clock domain, and therefore, a single scan chain. After the scan chain has been inserted, the normal mode clocks are set up again so that the circuit can be reoptimized to meet timing constraints. This reoptimization is an incremental optimization, which means that the circuit will not be structurally modified, only the buffers will be resized if necessary. Restructuring the design may invalidate the test vectors generated by the ATPG algorithm.

Each standard cell module is saved and written to a gate-level netlist. Each of these modules are separately placed, routed and buffer-sized in EPOCH. An EPOCH attribute called FIXEDBLOCK is added to the VHDL netlist that tells the compiler not to modify it when compiling a hierarchical design that uses it as one of its components. Otherwise, the compiler will consider that design to be just a gate level netlist and not a physical layout. Given the size of the SPARC design, one could imagine the complexity of the calculations involved in placing and routing the entire netlist at once. In fact, attempting to do this resulted in running out of swap space on a SPARC5 with 128M of RAM.

If each module in the design is locally optimized, then fixed, the task of

placing and routing at the top level of the design is significantly simplified. The only phase of the physical design process that must take into account the entire circuit at once is sizing the buffers. This is not as complex a process, since EPOCH already knows the position of every cell, what cells are driven by each cell, and the exact length of the nets connecting the cells. Once all the buffers have been sized, a power estimation algorithm is applied at the current level. This data will have changed, given that the buffer sizes have changed, which means that the size of the signal nets and especially the ground and power rails may have to be changed as well. The size of these nets is directly related to the loads they are driving, which, in turn, are proportional to the size of the buffers. The design is routed again after buffer-sizing and power estimation to reflect the new demands on the driving nets.

The other two types of modules, memory and datapath, are not compiled in Synopsys. EPOCH provides multiple-bit cells called *buscells*, which are comprised of a specified number of bit-slice cells which are placed and routed in a data bus oriented structure by a datapath router. Feedthrough paths are available in the direction of dataflow, and control routing is done between columns of buscells. Buscells used in this design include adders, an ALU, barrel shifters, registers, equality checkers, an incrementer, multiplexers and gates. EPOCH also provides several types of memory modules: RAM, ROM, FIFO, and register files (actually classified as buscells).

The verilog description of the SPARC datapath modules provided was not written with the EPOCH CMOS parts library in mind, but rather conforming to the parts available with the physical design tool used by the original designers. Fortunately, every slice of the datapath could be implemented by EPOCH buscells. Some manual placement was required to build the datapath. Some slices are comprised of several buscell segments, each with different inputs or control signals, so they cannot be merged into a single buscell. When compiling these slices, EPOCH automatically aligns each segment with other slices and segments according to their least significant bit cell. The segments are then manually placed into their appropriate postion in their slice. The entire datapath is the routed again, and a more compact configuration is achieved. As far as design for testability is concerned, all registers in the datapath are implemented as a special scan enabled register, and the compiler internally routes the scan chain within the register buscell. No scan cells are required at inputs or outputs of the datapath, since scan cells already exist at the origin or destination of those signals.

Constructing memory modules is required for the instruction and data caches, the floating-point control store, the floating-point and integer unit register files and the TLB. These are all fairly straighforward except for the TLB, which is a fully associative cache. The tag matching logic required for a fully associative memory is implemented with a content-addressable memory (CAM), however, EPOCH does not provide a memory module that is equivalent to a CAM. The workaround for this is to replace the CAM with a similarly sized RAM, essentially

turning the TLB into a direct-mapped cache for address translations. Adding test capabilities to memory modules requires no extra logic since all data, address, and control signals originate from or are destined for standard cell modules, which already have scan logic inserted, so full controllability and observability exists for each memory module.

The top-level netlist, consisting of references to datapath, memory and standard cell modules, is translated by EPOCH into its database. The design is then placed and routed given the geometry of the modules and their interconnections. When the modules are individually compiled, the external loading on the outputs and input delay are not known, but when the top-level design is under consideration, most loads can be calculated based on self-loading and external lumped capacitive load. The exceptions are the loads associated with the nets driving the pads and the boundary-scan logic, which are taken into consideration when the core circuitry is packaged.

EPOCH does not have a package defined for the SPARC pinout, but will place the pads and the core and route the design anyway. However, one cannot lock a signal to a particular pin number and location, and is at the mercy of EPOCH's packager. This supposedly can be overcome by defining a package and putting into a form that EPOCH can use. A separate verilog netlist is required for instantiating the core and the pads and how they interconnect. Attributes are attached to every pad that is imported into EPOCH, namely the boundary-

scan compliant pads. This chip netlist is translated by EPOCH, and the design is placed by a packaging algorithm and then routed. Figure 5.6 shows the final layout of the chip.

5.3 Multiple Die Case

The SPARC design is split up into three partitions. The grouping of the design considers only the functional unit level, which breaks the SPARC down to the nine units described at the beginning of the chapter. The problem of finding the best possible partition is discussed in [4]. The designer provides information related to the configuration of the SPARC such as bonding technology, maximum die size and substrate technology. A constraint generator uses this information to generate constraints for the partitioning algorithm, which is based on simulated annealing. The partitioner generates all possible partitions, which are then scored according to how well they meet the initial constraints. The partition candidates are further scored according to estimations of cost, size, power dissipation, and other factors. The candidate partition that best fits all constraints is shown in table 5.1. This partition does not take into account the fact that the Misc. module contains the JTAG control logic, which is actually required on every chip in the MCM.

For each individual die, the synthesis follows the same procedure as was

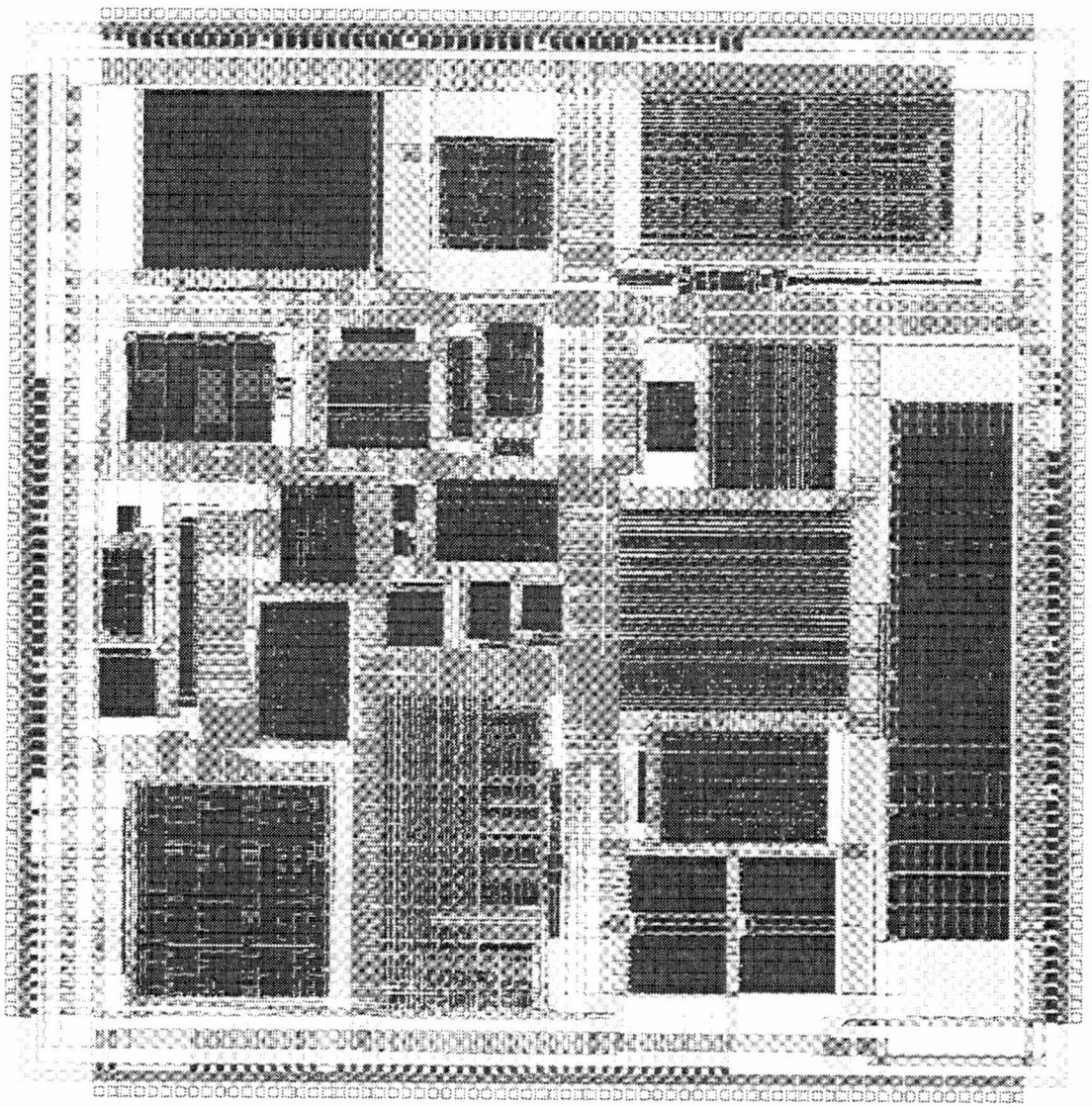


Figure 5.6: *Single Chip SPARC.*

Table 5.1: *Proposed best partition for SPARC*

Chip	Modules
Chip A	MMU, Instruction Cache, Data Cache
Chip B	Floating-Point Unit, Integer Unit
Chip C	Clock Controller, MEMIF, SBC, Misc.

used for the monolithic version. For instance, the standard cell modules for Chip B (*fp_fpc*, *fp_ctl*, *mdecode* and *mir*) are grouped together and Test Compiler inserts scan circuitry and generates test vectors. The modules are then reoptimized and translated into structural netlists. These netlists are read into EPOCH and each module is placed, routed and buffer-sized. The rest of Chip B is comprised of memory and datapath modules, and the JTAG control circuitry, which is implemented in standard cells. Scan circuitry is not inserted into the JTAG module since it must operate in normal mode at all times. All three chips require an instance of the JTAG controller, since it generates control signals to the boundary-scan chain as well as the global scan mode signal. The memory and datapath modules are identical to the corresponding modules from the monolithic case. EPOCH translates a package netlist, which instantiates the core and the pads, then places, routes and globally buffer-sizes the design. The physical layouts for each of the three chips are shown in figures 5.7 through 5.9

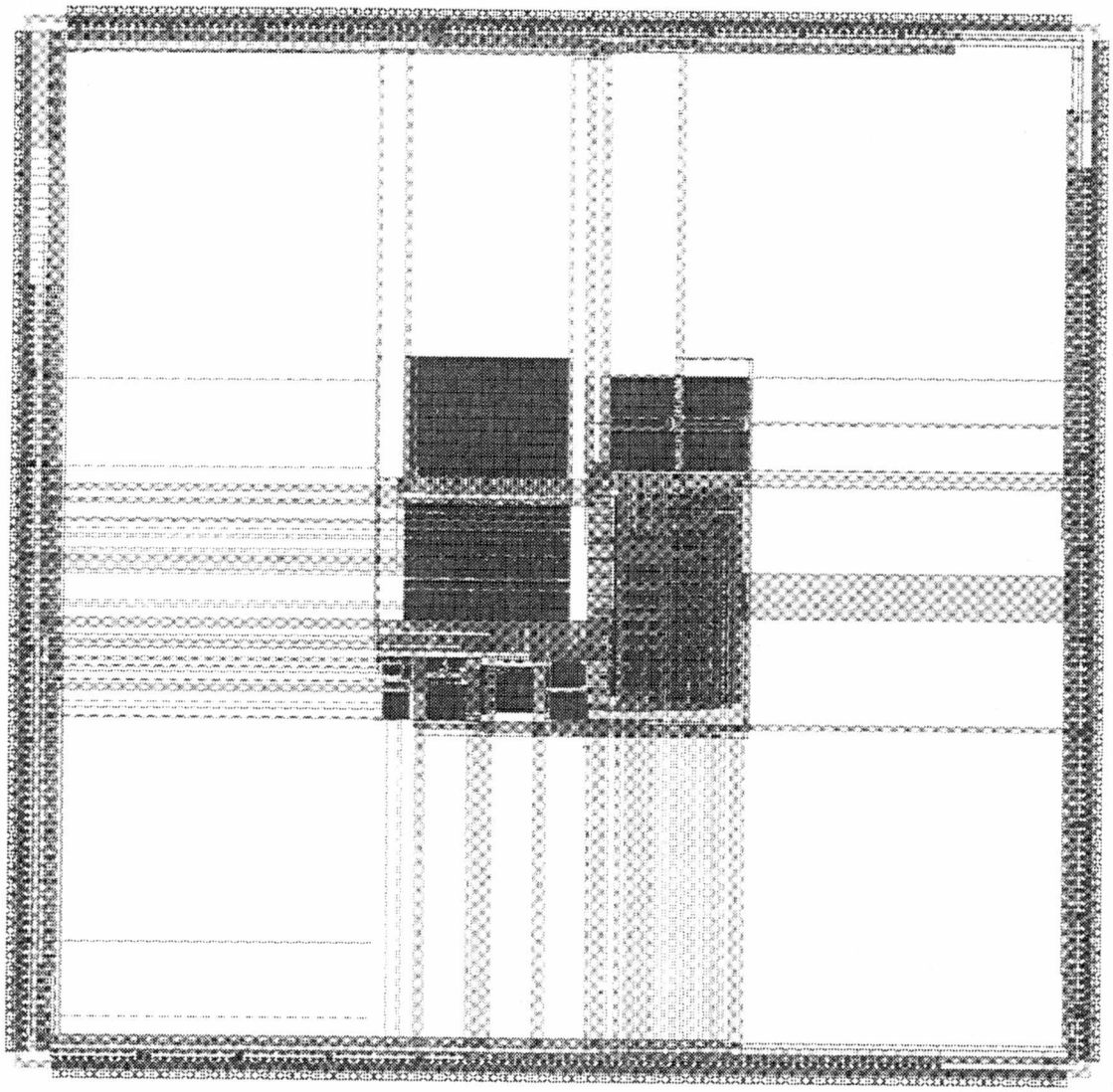


Figure 5.7: *Chip A.*

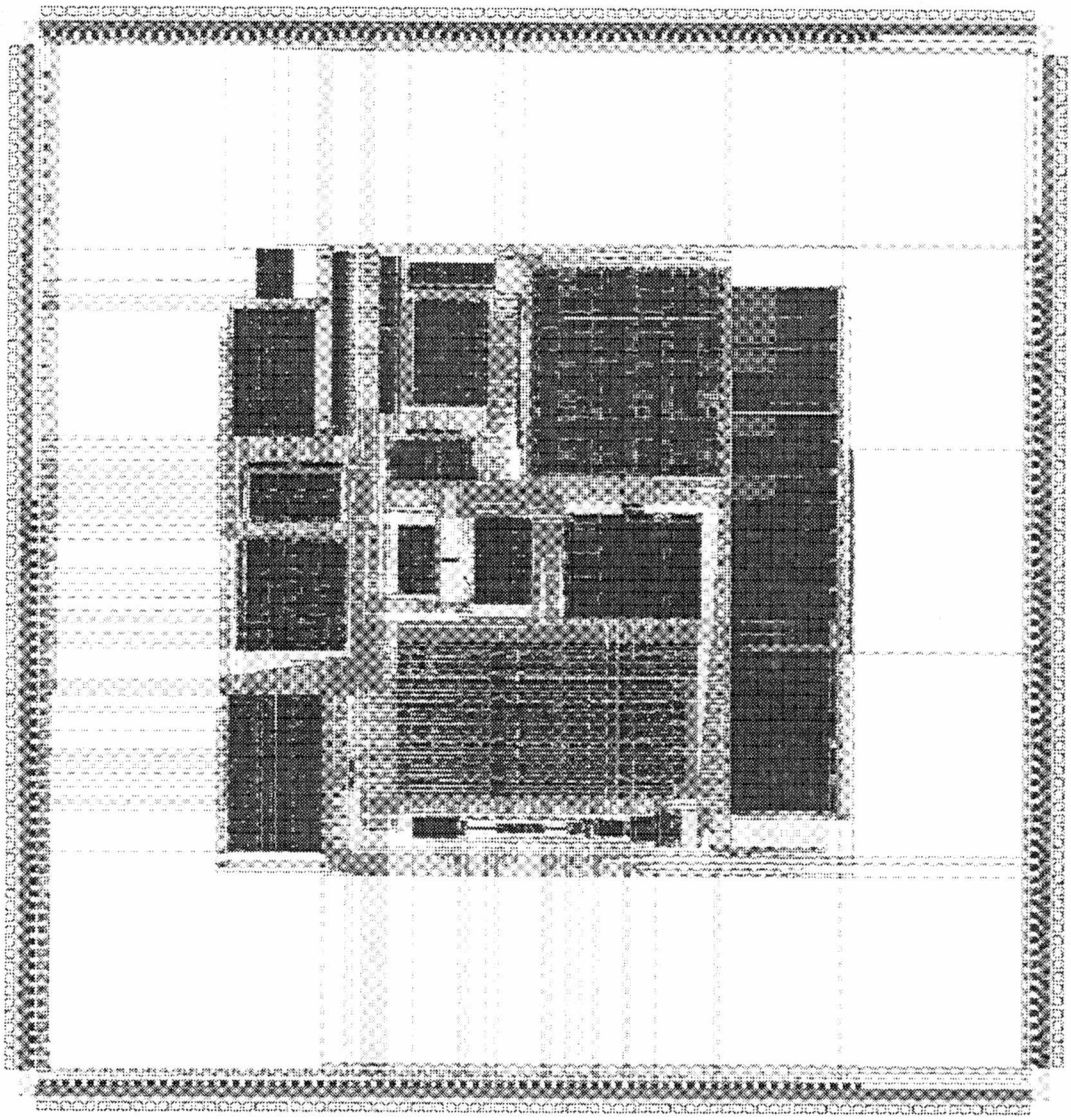


Figure 5.8: *Chip B.*

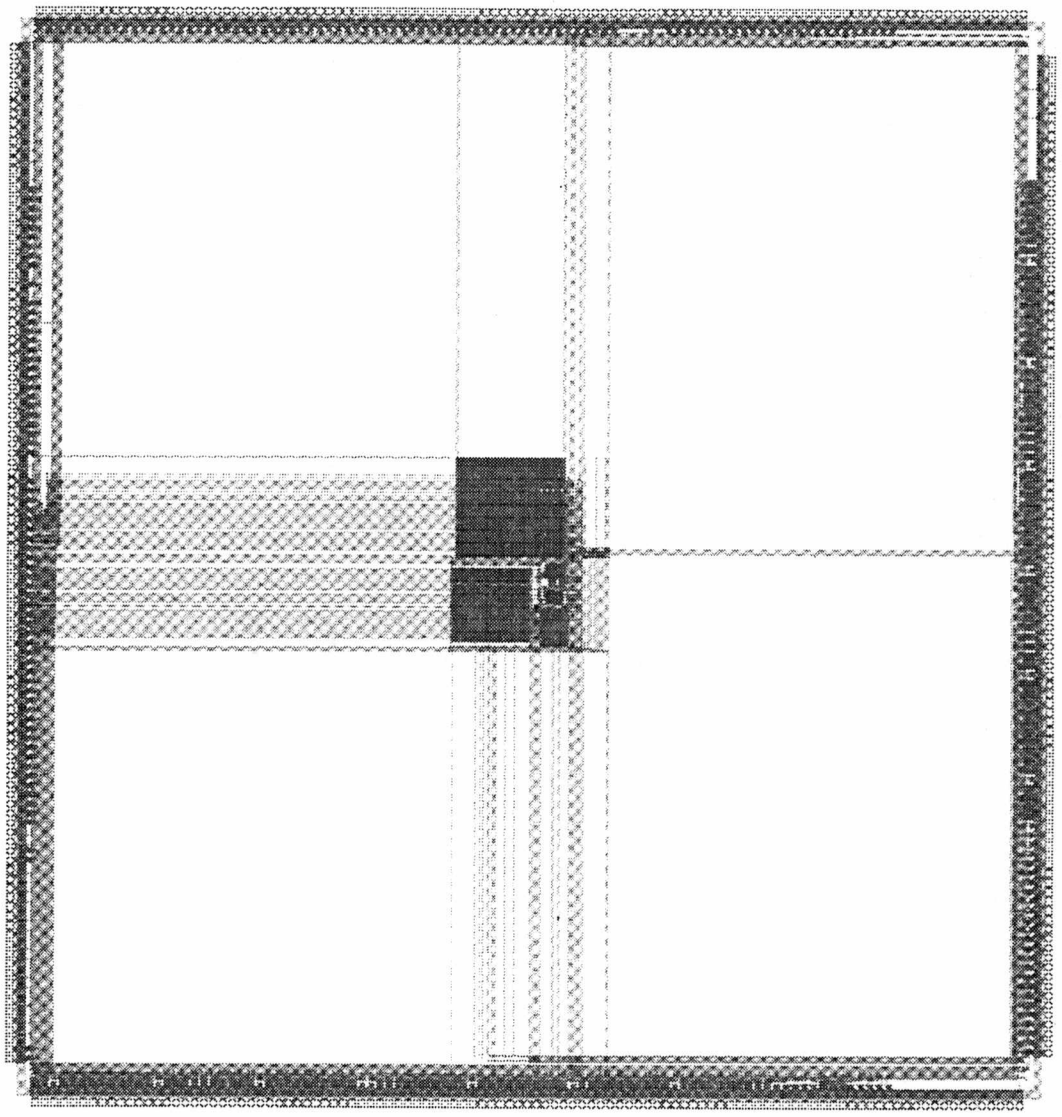


Figure 5.9: *Chip C.*

5.4 Cost Analysis

The analysis of the cost of the single-chip version of the SPARC and the MCM version is rather complex, taking into account many different factors. Figure 5.10 shows a simple flowchart for the process of assembling the multiple die or single die design into the target package. The overall cost consists of manufacturing costs and manufacturability costs. Manufacturing costs include the cost of materials and the processes involved in the production of the chips and the substrate, and the assembly of the MCM.

The chip manufacturing costs include process tooling, equipment and labor costs. The substrate processing and assembly steps actually consist of many processes, such as dispensing wire bond adhesive material (silver-filled epoxy) and die adhesive material, placing the wire bond dice, and attaching the wire bonds between the die and the substrate[5]. Each process has an associated cost that includes process time, labor rate, material, tooling and equipment costs. Each process also affects the yield, which in turn affects the overall cost of the device.

The manufacturability costs include the testing of bare die, bare substrate and substrate interconnect. Repairs, or rework, may be necessary to account for the probability of manufacturing faults in the production of chips and substrate interconnect. The final cost per unit of the device is calculated[5] as follows:

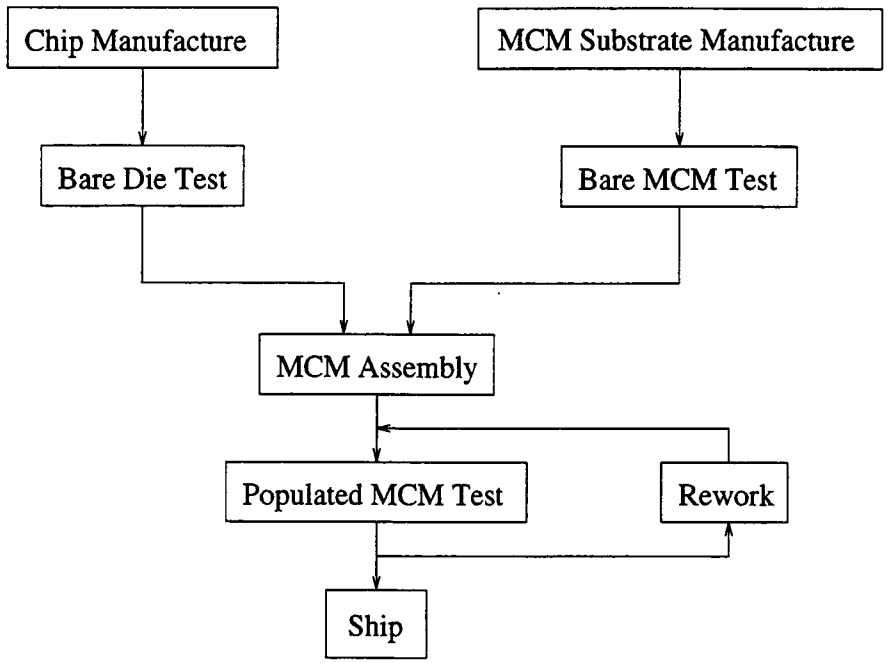


Figure 5.10: *The MCM manufacturing process flow.*

$$\text{FinalCost} = \frac{\text{MC} + \text{TC} + (\text{P(TE)} + \text{P(AF)}) \times (\text{RC} + \text{ATC})}{\text{FinalYield}}.$$

Manufacturing cost (MC) accounts for the cost of chip production, packages, connectors, heat removal mechanisms, power distribution features and the final casing of the MCM. This cost takes into account the yield of the die. Test costs (TC) refers to the cost of testing the IC, bare MCM and populated MCM one time each. Naturally, an additional cost must account for the probability of manufacturing faults. Some ICs pass the initial testing phase, but are later found to be defective. These ICs are referred to as test escapes, and P(TE) is the probability of an IC becoming a test escape. P(AF) is the probability of the occurrence

of an assembly fault during the die-attach process. If an assembly fault occurs, or a test escape IC finds its way onto an MCM, then rework is necessary to correct the situation.

Rework refers to the removal of a faulty chip, and the placement of a new one. The rework costs (RC) include demounting an IC by applying heat to the connection points, preparing the connection points, mounting the new IC and restoring the MCM casing. ATC is the cost of testing the substrate assembly again. Of course, there is still a probability of an assembly fault or test escape occuring during the rework process, so the rework and retest costs should be multiplied by the total percentage of faulty assemblies for each time rework is necessary. There is a maximum number (M) of reworks allowed before the MCM is scrapped, due to the desire to keep rework costs below the actual cost of the module. A new equation that accounts for multiple rework steps is as follows:

$$\text{FinalCost} = \frac{\text{MC} + \text{TC} + \sum_{j=1}^{M}(\text{P}_j(\text{TE}) + \text{P}_j(\text{AF})) \times (\text{RC} + \text{ATC})}{\text{FinalYield}}.$$

Manufacturing cost and the final cost are ususally dominated by the cost of manufacturing the die[5]. Die cost depends on the area of the die, number of die on the wafer, yield of the die, wafer cost, quantity, and non-recurring costs such as chip design and verification, and prototyping. An approximation for the number of useful die on a wafer is:

$$N_{die} = \frac{\pi r_{wafer}^2}{Area_{die}} - \frac{\pi(2r_{wafer})}{\sqrt{2Area_{die}}},$$

where r_{wafer} is the radius of the wafer. The first term divides the area of the wafer by the area of the die, and the second term corrects for the fact that the die is rectangular, but the wafer is round.

The die yield is the number of good chips on the wafer divided by the total number of chips. A model is defined for determining yield that is intended for large dice:

$$Yield = \left\{ \frac{(1 - e^{-(Area_{die}D)})}{Area_{die}D} \right\}^2,$$

where D is the defect density, which has units of defects per cm^2. The final cost for a die is given by:

$$DieCost = \frac{WaferCost}{(N_{die} \times Yield)} + \frac{NRE}{LifetimeProductionQuantity}.$$

The first term is an average recurring cost that is incurred for each die produced from a wafer. The second term accounts for non-recurring cost, which is divided by the total run of that design. Table 5.2 shows die cost calculations for the single chip SPARC design and each of the three die of the multiple-chip SPARC design. Wafer cost is taken to be \$800, defect density is 3 defects/cm^2, and wafer radius is

Table 5.2: *Cost data for the single-chip SPARC and multiple-chip SPARC dice.*

	Single-Chip	Chip A	Chip B	Chip C
die area (mm^2)	208.8	378.1	179.1	294.2
N_{die}	63	30	76	42
Yield (%)	2.5	23.56	17.64	75.71
Die Cost ($)	500	113.19	59.67	25.16
AA die area (mm^2)	-	55.8	69.5	9.5
AA N_{die}	-	281	236	1802
AA Yield (%)	-	58.7	51.9	90.8
AA die cost ($)	-	4.85	6.52	0.49

7.62 cm. The lifetime quantity is assumed to be large enough to make NRE costs negligible, at least for these calculations. The die area information is available after each die has been placed, routed and fixed in a wire bond pad frame.

The total die cost for the single chip SPARC is $500, compared to $198 for the multi-chip case. As figures 5.7 through 5.9 show, the size of each die in the multi-chip SPARC is determined by the number of pads, rather than the size of the core circuitry, whereas the single chip SPARC's die area is dominated by the core logic. Even though the area of the single chip SPARC is comparable to the area of each of the MCM die, its cost is much higher. Since it is much more

densely populated, it is more susceptible to defects, and therefore, the yield is much lower. The die cost for the MCM is rather high because the large area of each dice limits the number that can be cut from the wafer.

If area array bonding with flip-chip mounting is applied, each of the three die in the multi-chip design would be much smaller, since the signals, ground and supply would be routed to I/O locations near the origin or destination of the signal, rather than to the perimeter of the die. For flip-chip mounting, the die is flipped over so that the bonding pads rest on solder bumps, which are placed in a regular array pattern across the substrate[5].

The last four rows of table 5.2 shows die cost data for designs using area array (AA) I/O. The die area was calculated by increasing the original core circuitry area by a percentage relative to the number of I/O signals. This is to account for the pad driver logic area and the increase in routing complexity. The results show a significant decrease in total die cost of the multi-chip design, from $198 to $11.86.

The rest of the cost calculations are performed by a system tradeoff analysis tool called MSDA, or the Multichip Systems Design Advisor. MSDA calculates the manufacturing costs associated with assembling the MCM, testing costs and rework costs. The die cost is one important user input, since it dominates the cost of the MCM, but there are many other factors that determine cost, some of which are shown in table 5.3. Synopsys' Test Compiler provided the fault coverage,

Table 5.3: *Comparison between Monolithic SPARC and partitioned design.*

	Single-Chip	Chip A	Chip B	Chip C
core area (mm^2)	160.5	47.6	62.3	7.91
chip area (mm^2)	208.8	378.1	179.1	294.2
Total Power Dissipation (W)	2.88	2.06	0.85	0.19
Pad count	283	487	331	528
Signal I/O	194	427	271	468
Ground I/O	49	30	30	30
Fault coverage (%)	99.01	99.98	99.98	99.26
Pattern length	1907	634	621	652
Number of Patterns	1131	436	575	436

pattern length and number of test patterns. EPOCH provided the rest, based on the physical implementation of the SPARC.

MSDA considers NRE and development, learning curve delay, equipment and tooling, recurring fab and assembly, testing, rework and even maintenance costs when calculating the total cost of the system[15]. One major contributor to system cost besides die area is rework costs. If a test escape makes its way onto an MCM, or if the assembly of the MCM is faulty, then rework must be done to correct the situation. The offending die is removed, and another is placed onto the

assembly. There is still a probability that the new die is a test escape or that the assembly is faulty, so rework might be performed again. Each time the MCM is reworked, a cost is incurred, so one has to put a limit on how many rework cycles are executed before giving up on, or scrapping, that MCM. MSDA allows one to specify the number of rework cycles, the probability that the rework process was finished successfully, the yield of the rework process, and all the costs associated with the rework process.

Each chip is specified in MSDA, then a partition is defined, consisting of the set of all chips or possibly a subset. In this case, all three chips are grouped into one partition. Further specifications are then defined for the MCM partition such as interconnect technology (MCM-D is used), wiring complexity, cooling path, labor costs, lot size, lifetime quantity, and packaging. MSDA considers all user input to calculate total cost, size, and electrical and thermal properties. Table 5.4 shows the comparison in cost and module size for single and multiple chip wire bond designs, and the single and multiple chip area array bond designs. The cost of the wire bond MCM is about 43% less than the single chip version. The area-array bond MCM is about 90% less than the single chip version, and 82% less than the wire bond MCM. The area-array system will also be about 3.3 times smaller than the wire bond MCM.

Table 5.4: *Cost and area comparisons between the 4 design versions.*

	cost ($)	module area (mm^2)
Single chip, wire bond	529.50	486.8
Multiple chip, wire bond	303.13	1787
Multiple chip, area array	54.38	541.9

CHAPTER 6

SUMMARY AND FUTURE WORK

6.1 Summary

As ASIC designs become increasingly complex, design and implementation considerations become more important. Designing a chip for testability decreases the defect level, or the number of parts that exhibit incorrect behavior or system failure due to manufacturing defects. Test circuitry is inserted into the design to give access to internal points of the circuit, so that possible faults can be detected with a series of applied inputs and observed outputs. After fabrication, the chips are tested for faults before they are packaged, thus reducing the number of defective devices. Since cost per device depends on the final yield of the dice, it is important to have a high fault coverage for each. One must tradeoff the cost of extra design time and extra test circuitry for the savings resulting from higher final yield.

Partitioning a very large monolithic design may be necessary, since yield may be extremely low, thus driving up the die cost significantly. A multiple-chip

design requires more area and processing steps, but may end up costing less than the monolithic case, for the most part because of lower total die cost due to high yield. The bonding technology used for the Multi-chip module design is a key consideration.

It was shown that if the dies use peripheral wire bonding pads, their size is excessive. Although the number of top-level I/O stays relatively the same, the number of I/O of each die in the multiple-chip version is larger than that of the top-level. So, smaller core circuitry coupled with more I/O yields an I/O limited design, where it is the peripheral pad frame that dictates the size of the die, rather than the size of the core. If area-array bonding is used, the restrictions of the placement of the I/O pads is not limited to the periphery, but rather to a regular array of possible locations throughout the chip. Therefore, the size of each die is reduced significantly, along with the die yield and therefore, cost.

6.2 Future Work

There are several ways, described below, that the work done in this thesis can be extended or refined.

- Functional test vectors should be developed for AC testing of the SPARC.

- Test vectors for the datapath and memory modules should be developed, conforming to the test protocol.

- The multiple-chip version can be synthesized. This requires design of new pad drivers for the area array bonding environment. Also, a way to automatically insert the drivers and the bonding pads into the netlist, and place and route the design in such a way that the bonding pads align to the grid of possible bonding pad locations.

- Once the area array bonded, multiple-chip version of the SPARC is synthesized, a more accurate determination of cost and performance can be had, and a stronger case can be made for partitioning a very large monolithic design into an MCM design. This may persuade system designers to consider a MCM instead of a single chip design early in the design process.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design.* W. H. Freeman and Company, New York, 1990.

[2] Harry Bleeker, Peter van den Eijnden, and Frans de Jong. *Boundary-Scan Test - A Practical Approach.* Kluwer Academic Publishers, Norwell, MA, 1993.

[3] Xinghao Chen and Michael L. Bushnell. *SEST - A Sequential Circuit Automatic Test Pattern Generator at Rutgers - User's Guide.* Rutgers University, Piscataway, NJ, March 1995.

[4] Peyman Dehkordi, Karthi Ramamurthi, Don Bouldin, Howard Davidson, and Peter Sandborn. Impact of packaging technology on system partitioning: A case study. *Proceedings of the IEEE Multi-Chip Module Conference*, pages 144–149, 1995.

[5] Daryl Ann Doane and Paul D. Franzon, editors. *Multiple Module Technologies and Alternatives: The Basics.* Van Nostrand Reinhold, New York, 1993.

[6] Saverio Fazzari. Fate: A fault simulator and automatic test pattern generator for vhdl. Master's thesis, University of Pittsburgh, 1990.

[7] USC Test Group. *The USC Test System Manual.* University of Southern California, Los Angeles, CA, May 1993.

[8] John K. Hagge and Russell J. Wagner. High yield assembly of mcm through known good ic's and effective test strategies. *Proceedings of the IEEE*, 80(12), Dec 1992.

[9] Craig Hall, Brian Chess, and Tracy Larrabee. *The Nemesis User's Manual.* University of California, Santa Cruz, California, 1994.

[10] John L. Hennessey and David A. Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufman Publishers, Inc., San Mateo, CA, 1990.

[11] Tracy Larrabee. Test pattern generation using boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):4–15, Jan 1992.

[12] Kenneth P. Parker. *The Boundary-Scan Handbook.* Kluwer Academic Publishers, Boston, 1992.

[13] Rochit Rajsuman. *Digital Hardware Testing: Transistor-Level Fault Modeling and Testing.* Artech House, Boston, 1992.

[14] Markus F. Robinson, Frederic Mailhot, and Jim Konsevich. Technology-independent boundary scan synthesis. *IEEE International Test Conference*, 1993.

[15] Savantage, Inc., Austin, TX. *The Multichip Systems Design Advisor Tradeoff Analysis Tool User's Guide*, 4.0 edition, 1995.

[16] SPARC International, Inc, Menlo Parc, CA. *The SPARC Architecture Manual*, version 8 edition, 1992.

[17] Frank F. Tsui. *LSI/VLSI Testability Design.* McGraw-Hill, New York, 1987.

[18] Neil H. E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design.* Addison-Wesley, 4th edition, 1993.

VITA

Quinn Devine was born on September 16, 1969 in Salina, Kansas. He graduated from Lake Braddock Secondary School in 1987 and was enrolled at Cornell University in Ithaca, NY. After four of the most grueling years of his life, he received a Bachelor of Science degree in Electrical Engineering. After some time off, he enrolled in the graduate program of at the University of Tennessee in Knoxville. He received his Master of Science degree in Electrical Engineering in 1996, and has accepted a position at Intel in Chandler, Arizona.