In-line test of synthesised systems exploiting latency analysis

A.C. Williams, A.D. Brown and M. Zwolinski

Department of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ
Hampshire, U.K.

Abstract

During normal operation, there are periods of time in which units in a digital system (adders, multipliers etc.) are inactive, i.e. are not processing any useful data. These "latent periods" may be exploited to continually perform sets of unit tests, thus providing a dynamic indication of the healthiness of the system with little or no effect on its performance.

This paper details an analysis technique for identifying and quantifying these latent periods by modelling the flow of control through the system as a Markov chain, which takes into account branching and feedback in the controller. The resulting data describes the distribution of latent periods in an entire design, and, given a testing requirement in the form of a minimum number of (latent) cycles required to perform a test, provides a figure for how often and to what extent a particular unit may be tested during normal operation. This analysis is utilised to investigate the impact particular optimisation strategies have on the distribution of latent periods, in a number of synthesised benchmark designs. These results are further developed to demonstrate how a knowledge of the latent period distribution can be used to direct the synthesis process and lead to a substantial improvement in the distribution of latent periods, whilst not over adversely affecting other design aspects, particularly the area.

1. Introduction

The use of synthesis in the design of complex digital systems is becoming ever more widespread. Furthermore, as the complexity of systems increases, so the abstraction of synthesis to higher levels will follow. Here, the designer tends to be concerned with the high-level specification of behaviour, and whether the implementation meets certain gross design criteria such as speed, area, power consumption and testability, while the underlying sub-circuits that make up the detailed *structural* implementation are largely hidden.

An important aspect of any hardware system is the inclusion of some form of testability. This is generally a post-synthesis task which adds structures appropriate to the structural implementation in order to provide the level of testability required by the designer[1]. These test structures can consist of some form of scan path (partial or otherwise) in conjunction with some automatic test pattern generation techniques[2,3] and perhaps built-in self-test (BIST)[4-6], thus allowing testing at manufacturing time. BIST additionally allows a test to take place at system initialisation time. Safety critical systems, which cannot easily be taken off-line, may also require a real-time indication of their “healthiness”, so that should a fault occur, redundant circuits (possibly at the chip or board level) can be immediately switched in, and the faulty device identified and replaced without any operational disruption. The exact nature of the testing technique used is not important within the context of this paper.

The use of on-line testing[7], that is testing the system whilst it is in operation, requires additional processing capabilities and may have a performance implication for the system. High level behavioural optimisation[8,9] allows the user to set strategic (i.e. overall system) goals. For the sake of brevity, let us restrict the current discussion to optimisation with respect to area and/or speed. The structural equivalent of the behavioural description supplied by the user then corresponds to a single point in the two-dimensional “design space”, the coordinates of which are (area, delay). The space is discrete, nonisotropic, inhomogeneous and degenerate. Alternative structural implementations (each generating the defined functional behaviour) will form a set of points within this space, constituting the “set of realisable designs”, as illustrated in figure 1. Points (A), (B), (C) and (D) represent designs on the edge of this region, the detection of which is easy; the decision as to which is “best” however, can only be made by the human designer. (A) is clearly superior to (B), but relaxing the speed requirement a little allows us to move to (C), which requires only half the area.

Shifting the perspective of the previous sentence, the design (C) may be made even faster at the cost of doubling the area.

In this paper, we introduce another dimension to the design space: that of *in-line testability*. The units of this extra dimension are a percentage, indicating the fraction of the design that can be tested *without impacting whatsoever on the data throughput*. (Formally, we define testability – a number between 0 and 1 – to be the minimum test completion probability of all the physical functional units in the structural design. The term "test completion probability" is defined in section 2.1). Thus a particular behavioural design may have structural implementations at (3 mm^2, 267 cycles, 78%), or at (6 mm^2, 500 cycles, 95%). As with the tradeoffs between (A)..(D) in figure 1, only the human designer can sensibly assert which is acceptable or "best". It could be argued that the testability has had an impact on the delay, but it could equally well be argued that the delay has had an impact on the area.

Terminology like this is misleading; the space exists, defined by the behavioural description (and the capabilities of the synthesis system). It is the responsibility of the human designer to make the ad hoc judgements about which is "best" – this is what humans are good at.

This paper details the methods and quantitative results of an investigation into the feasibility of adding in-line testability to synthesised synchronous systems. Such designs invariably have periods of time (both whole and partial clock cycles) during which some processing units are inactive, and it is these latent periods that are exploited to support in-line testing capabilities.

The study examines the in-line testability of a number of benchmark designs synthesised by the MOODS synthesis system[8,9], using a method for analysing the latent periods available during system operation, which takes into account the effects of branching and feedback in the control flow. Both area and speed optimised implementations are analysed to assess the main factors in the synthesis process influencing the development of latent periods. As a result, techniques to improve the overall in-line testability of a synthesised design are considered, together with their impact on the user's optimisation objectives.

At the broadest level of strategy, then, a synthesis system which supports optimisation with respect to testability rests on three planks:

1. The ability to deliver meaningful “what’s best ?” decisions about significantly different criteria (area, delay, testability, power dissipation…).

2. Proof/demonstration that testing is possible within the data throughput constraints of the overall design.

3. The techniques used to test the various physical functional units.

The first point has been discussed elsewhere[7-10]. The second is the subject of this paper, and the third is covered by the extensive literature already published on testing, for example[2,3,6,11-14]. Each approach has its advantages and disadvantages and each takes a different amount of time to perform its complete test.

The rest of this paper is organised as follows: section 1.1 describes the basic requirements for in-line testing along with some assumptions and simplifications made for this study. Section 2 then describes the *latency analysis* techniques developed, first with an introduction demonstrating the basic principles involved and defining a number of important terms. This is then followed by a detailed description of the practical methods used to analyse real designs synthesised by MOODS, which are then used, in section 3, to examine the in-line testability of a number of benchmark designs, with results described for the various initial and optimised implementations. The section concludes with an investigation into how the designs may be further optimised in order to improve their in-line testability, and the trade-offs necessary to achieve this.

1.1 In-line testing

In-line testing requires a system to be tested concurrently with externally supplied data throughput, i.e. “normal operation”. This may be done with some external monitoring system, however here we assume that part of the system may be tested using any latent periods created when units are not processing data required for normal operation[7]. These latent periods depend upon the scheduling and sharing of units within a design as well as the type of unit, and thus vary between different synthesised implementations of a single behavioural description. Within a distributed data and control path architecture, such as that generated by the MOODS system, the data path units typically constitute a major part of any implementation. For the purposes of this study only the functional units in the data path are

considered. (This notwithstanding, the analysis methods can equally be applied to other parts - registers and interconnect - of the design.)

The testability of a given unit in a larger design is dependent upon a number of parameters: the unit type, the length and fragmentation of the latent periods, the testing strategy and the test controller. A test controller is required to test each unit over one or more of its latent periods according to the selected testing strategy. If more than one period is used then the tests must be halted when the unit is required to process real data, thus complicating the controller architecture. There is clearly a trade-off between test controller complexity and the throughput of tests applied to units, but this particular aspect is treated on the same level as every other decision made by the optimisation algorithm.

In order to demonstrate that in-line test using latent periods is achievable, some assumptions on the approach to testing individual units must be made. For the purposes of this paper:

- It is assumed that an optimal set of test sets exists capable of testing any particular unit for the required single-stuck-at-fault coverage.

- To untangle the interaction of testability and other user targets, (optimisation dimensions), the paper centres around the analysis of two explicit examples. Each of these is optimised, separately, twice – once for area and once for delay. Thus there are *four* different structural implementations, two optimised with respect to area only, two with respect to delay only. In-line testing structures are then retrofitted to these, and the obtainable coverage analysed.

The test sets are generated externally using a commercial test vector generation tool[15], and are assumed to be applied at a rate of one vector per system clock cycle. Thus a latent period of n cycles is necessary to fully test a given unit, where n is the number of test vectors required. Exhaustive testing using pseudo-random sequence generators could equally be used, but this would entail the analysis of extremely long sequences, unsuited to this particular application. It is also assumed that the test controller is of a relatively simple design, which must completely test a unit within a *single* latent period. While it would be possible to employ an interruptible controller, capable of performing a single test over *several* latent periods, the overheads involved in the more complex architecture, and additional registers required to store partial results, are considered too expensive for general use.

The effect of these assumptions on the results and hypotheses thus formulated will be discussed in the results section.

2. Latency analysis

Section 2.1 describes the principles of latency analysis and the obstacles that must be overcome in order to obtain accurate results. The discussion describes latency analysis in the context of the architecture generated by the MOODS synthesis system. This is followed by a detailed description of the methods and algorithms implemented in an analysis tool used to determine latency and testability figures.

2.1 Design latency analysis

The MOODS synthesis system generates a distributed architecture containing a control unit which organises the flow of data through a data path. The control path consists of a single bit register for each state joined together by arcs through which tokens are conditionally passed, in a similar manner to that of Petri-nets. It comprises a combination of parallel and conditional branches, nested loops and sub-program calls. Each state may conditionally activate any number of functional units (adders, multipliers etc.) in the data path, which are interconnected by nets directing data through the system.

An initial behavioural description is translated from VHDL[16] source (the primary input language for the system) to an unoptimised register-level implementation whereby each instruction occupies a single control state, and each data path unit executes one instruction, the results of which are stored in registers at the end of each clock cycle. During optimisation both the control and data path graphs are transformed so that the user's objectives for constraints, such as area, speed and power dissipation are more closely met. This results in control states conditionally activating many data path units, each of which may be used to implement a number of instructions.

The purpose of latency analysis is to calculate the distribution of inactive clock cycles for a given data path unit. Armed with this information and the time required to perform a complete test on the unit, a figure may be obtained describing its in-line testability. This section explains the basis behind the various terms and figures used for latency analysis and in-line testability.

Figure 2a shows a simple control graph comprising a sequential section of four states linked by arcs. During execution control may exist in any one of the four states and follows

the sequence of states, repeating via the feedback arc from state 4 to 1. Each data path node implements a number of instructions executed from one or more control states. The set of control states that activate a particular data path unit is called the target set. In this example we define the target set to be only state 3, which activates the data path unit being analysed. From inspection of the graph it can be seen that this unit is inactive during states 1, 2 and 4. A latent vector is defined as a group of arcs (state transitions), none of which are incident on a member of the target set, except that the final state may terminate on a member of the target set. Thus in figure 2a, there exist latent vectors {(4,1)}, {(4,1),(1,2)}, {(4,1),(1,2),(2,3)}, {(1,2)}, {(1,2),(2,3)} and {(2,3)}. The length of a latent vector is the latent period.

If, for the sake of illustration, a full test (sufficient to provide the desired fault coverage) for the unit active in state 3 requires 3 cycles, then the test can only be completed when started from state 4, that is the test may only be applied during the latent vector {(4,1),(1,2),(2,3)}. At any moment in time, for this example, there is an *equal probability* of being in any one of the four states; thus the state probability for each state is 0.25. Since a full test can only be started from state 4, the overall probability of being able to fully test the unit at any given moment is 0.25. Alternatively, if the test only takes 2 cycles to complete, the probability increases to 0.5 as there are two possible latent vectors in which a full test may be applied: {(4,1),(1,2)} and {(1,2),(2,3)} (a sub-vector of {(4,1),(1,2),(2,3)}). This figure is termed the test completion probability and is defined as the probability that, at any arbitrary time, a test may be started and completed before the data path unit is activated.

Figure 2b shows a more complex example incorporating a conditional branch, which is typical of the structures obtained as a consequence of *if*, *for* and *while* constructs. Here, states 2 and 3 are mutually exclusive, their activation depending upon the condition of the preceding state 1. Assuming (for now) that there is an equal probability of choosing either alternative, the *arcs* can each be assigned an arc probability of 0.5. The presence of such conditional branches can have a profound effect on the latent vectors present. As an illustration, consider the latent vectors starting at state 1. Here, the shortest latent periods are obtained from vectors {(1,2)} and {(1,3)} each of which has a path probability of 0.5, there being an equal chance of branching one way or the other. Walking either of these paths will eventually return us to state 1, where there is again a 0.5 chance of taking either branch. (Thus the chances of ending up at the state in the target set (state 3) will be the asymptote of $\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\ldots$, i.e. 1, which is what we would intuitively expect.)

A latent vector's path probability describes the probability of obtaining a particular vector given that the vector's start state is currently active. However, to determine the probability of obtaining the vector at any point in time, the state probability of the start state must also be taken into consideration. Referring to figure 2b, the state probability for state 1 is 0.286[1], therefore the latent vector {(1,2),(2,5)} (path probability 0.5) is said to have a vector probability (vector probability × state probability) of 0.143. Whenever a vector includes the branch of state 1 it is divided into two paths, one down the left side that will continue, and one down the right side that will end at state 3. Each time this happens the vector path probabilities are halved, thus in this example, an infinite number of latent vectors with exponentially decreasing path probabilities are obtained.

The behaviour of individual data path units can be examined using the units' dead vectors, which are defined as latent vectors that start from a particular control state, and end on a member of the target set. Thus, a test requiring *n* test sets may be applied during any dead vector of length *n* or greater. The dead vector probability for a vector of length *n* is defined as the probability that at any given moment in time, a member of the target set is exactly *n* clock cycles away from a particular control state. These vectors are encapsulated in the complete dead vector probability histogram, which plots the sum of all dead vector probabilities for a particular functional unit against the dead vector length.

In order to investigate the in-line testability of real designs using data obtained from the methods above, three basic graphs are plotted. The first details the distribution of dead vectors for an individual data path unit by plotting the complete dead vector probability histogram. This shows how the individual vectors are affected by the control structure and the particular scheduling of unit activations within the control graph. A more general guide to the in-line testability of a particular unit is given by the second type of graph which plots the

---

[1] Let the probability of being in state *i* be P_i. Then from Fig. 1b, we see that $P_2 = P_1/2$, $P_3 = P_1/2$, $P_3 = P_4$, $P_5 = P_1$ and $P_1 + P_2 + P_3 + P_4 + P_5 = 1$. Inverting

$$\begin{bmatrix} \frac{1}{2} & -1 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

gives us $[P_1,P_2,P_3,P_4,P_5]^T = [0.286,0.143,0.143,0.143,0.286]^T$

unit's test completion probability against the test time required. Finally, to illustrate the overall in-line testability of an entire implementation, a set of sample test times is used to determine the test completion probability for each unit in an implementation, and the results plotted on a histogram. It should be emphasised that this probability does not refer to a unit's fault coverage which is defined by the test sets, it only represents how often a unit may be fully tested while the system is active.

2.2 Latency calculation methods

In order to obtain the above results for real synthesised VHDL designs, the output from the MOODS system is analysed using a dedicated post-processing tool. This includes a number of functions that operate on the control and data graph structures allowing the user to perform a detailed investigation of the optimised design. The data produced includes dead vectors and test completion probability versus test time for any given data path unit, together with combined testability results for all or part of the entire design.

2.2.1 Markov chains

For the purposes of this discussion, the controller is considered to be a deterministic state machine. The control unit of a synthesised implementation can be described by the complete set of control graph states, *S*. As the unit executes, the flow of control through the graph is a discrete time stochastic process[17] defined by the set of variables:

$$\left\{X_t \in S \;\middle|\; t = 0,1,2,...\right\}$$

where X_t represents the current state (not set of states) t cycles from the start of the process. The passage from state to state through the control graph is described by a set of instances of X_t where decisions made at forking states are independent of the past history of X_t. This property defines the process as a Markov chain[18].

From any given state i, the probability of moving to state j on the next clock cycle is defined as the transition probability:

$$p_{ij} = P\left(X_{m+1} = j \middle| X_m = i\right) \quad \text{where} \quad i, j \in S \tag{1}$$

This corresponds to the arc probabilities described in the previous section. Using these probabilities it is possible to fully determine the evolution of the control process as time progresses. In addition, the n-step transition probability of moving from state i to state j in n cycles can be defined by:

$$p_{ij}^{(n)} = P\left(X_{m+n} = j \middle| X_m = i\right) \tag{2}$$

The states of a Markov chain can be classified according to a number of different properties:

- An irreducible class is made up from states which form a set C where $\forall\ i, j \in C, \exists \text{ integers } m, n > 0 \ \middle|\ p_{ij}^{(m)} > 0, p_{ji}^{(n)} > 0$ (i.e. C is a connected graph). This can be viewed as a set of states which, once entered, will either never be exited or, having been exited will never be re-entered.
- A state is aperiodic if it regularly returns to itself, i.e. $p_{ii}^{(n)} \geq 0$ for $n > 0$, provided $p_{ii}^{(n)} = 0$ for only a finite number of n. If the state has a constant return period it is periodic.
- The first return time, T_{ii}, is defined as the number of steps between any two successive entries into a state i.
- A transient state is one for which there is a finite probability that it may never be re-entered.
- A positive-recurrent state recurs infinitely often, and has a finite return time.

2.2.2 Latent vector calculation

The Chapman-Kolmogrov equation[17, 18] describes the evolution of a Markov chain as a matrix multiplication based upon the transition matrix:

$$\mathbf{P}^{(n+m)} = \mathbf{P}^{(n)}\mathbf{P}^{(m)} \quad \text{for } m, n \geq 0 \tag{3}$$

By setting m to 1, an iterative method for evaluating each of the n-step transition probabilities from the start of the process is obtained:

$$p_{ij}^{(n+1)} = \sum_{k \in S} p_{kj}^{(n)} p_{ik}^{(1)} \quad \text{for } n = 0,1,2.... \tag{4}$$

This relationship forms the basis for the calculation of latent vector probabilities. A latent vector of length n is defined as an n-step transition starting from an arbitrary control state, which does not activate a particular target data path unit during any of its n steps, except perhaps the final one. Each of the target units may be activated by a number of states in the control graph which make up the target set, D. Thus, the probability of obtaining a latent vector of length n from a starting state i, to an end state j can be defined as:

$$V_{ij}^{(n)}(D) = P\left(X_{m+n} = j,\ \ X_m, X_{m+1}, ..., X_{m+n-1} \notin D \ \middle|\ X_m = i\right) \quad \text{where } i, j \in S \text{ and } D \subset S \tag{5}$$

Comparing this result to equation 1 it can be seen that the one-step latent vector $V_{ij}^{(1)}$ is directly related to the transition probability $p_{ij}^{(1)}$:

$$V_{ij}^{(1)} = \begin{cases} 0 & \text{if } i \in D \\ p_{ij}^{(1)} & \text{if } i \notin D \end{cases} \quad (6)$$

From this, equation 4 yields an iterative method for determining the latent vector probability:

$$V_{ij}^{(n+1)} = \sum_{k \in S} V_{kj}^{(n)} V_{kj}^{(1)} \quad \text{for } n = 0,1,2,... \quad (7)$$

Thus to calculate latent vector probabilities it is first necessary to modify the one-step transition matrix $\mathbf{P}^{(1)}$ so that all arc probabilities leaving the target states D are set to zero. This results in the one-step latent vector matrix $\mathbf{V}^{(1)}$. Calculating the multi-step latent vectors is then simply a matter of iteratively applying equation 7 via a matrix multiplication as in equation 3.

In order to be able to relate the latent vector probabilities for different start states to each other, it is necessary to weight the vectors according to the probability of being in each particular start state. This is achieved by taking into account the steady state probability described below.

2.2.3 Steady state probability calculation

π_j is the steady state probability of being in state j at any given moment. This can also be interpreted as the asymptotic transition probability for progressing from state i to j:

$$p_{ij}^{(n)} \to \pi_j \geq 0 \text{ as } n \to \infty \forall i, j \in S \quad (8)$$

which must be independent of the starting state i if π_j is to have a stationary value. For this to be true, the Markov chain must be both irreducible and aperiodic. These conditions are defined by Kolmogrov's theorem, which also states that in addition to the above properties, a positive-recurrent Markov chain will have a unique solution to the steady state equation:

$$\pi = \pi \mathbf{P}^{(1)} \quad (9)$$

The steady state probabilities for the entire Markov chain can therefore be calculated by determining the solution to the simultaneous linear equations:

$$\pi_j = \sum_{k \in S} \pi_k p_{kj}^{(1)} \quad \text{where } \sum_{j \in S} \pi_j = 1 \quad (10)$$

A Markov chain having the above properties is said to be ergodic. This describes a process in which it is possible to reach any state from every other state. Note that initialisation states (executed only during power-up or system reset) are transient, and have a steady state probability of 0.

2.3 Dead vector calculation

The final stage in the calculation of the complete dead vector probability $H^{(n)}(D)$ for vectors of length *n* is to combine the above methods into a single algorithm. The dead vector probability for a vector of length *n* starting from a particular state *i*, is defined as the sum of all possible latent vectors of length *n*, $V_{ij}^{(n)}$, which start at that state and end on any member of the target set *D*. To calculate the complete dead vector probability, the weighted sum of dead vectors for all possible start states is used:

$$H^{(n)}(D) = \sum_{i \in S} \pi_i \sum_{j \in D} V_{ij}^{(n)} \tag{11}$$

Note that from the definition of $V_{ij}^{(n)}$, if $i \in D$, $V_{ij}^{(n)}$ is zero.

$H^{(n)}$ is evaluated for a range of lengths to construct a dead vector histogram for any particular functional unit (see results section). In order to determine the probability of being able to test a unit given a minimum test time *t*, it is necessary to take into account all dead vectors of length *t* and above. This is obtained by integrating $H^{(n)}$ to determine the units test completion probability $T^{(t)}(D)$:

$$T^{(t)}(D) = \sum_{n=t}^{n \to \infty} H^{(n)}(D) \tag{12}$$

Theoretically, the sum of all the dead vector probabilities should be 1. However, since dead vectors starting from a target state will have a probability of 0%, due to the modifications made to the transition matrix to build $\mathbf{V}^{(1)}$,5 zero length dead vectors do not register in the results. Thus the actual sum of the dead vectors will be $\sum_{i \notin D} \pi_j$. Equation 12 then becomes:

$$T^{(t)}(D) = \sum_{i \notin D} \pi_i - \sum_{n=1}^{t} H^{(n)}(D) \tag{13}$$

It is now a relatively simple task to combine all the above information into a single dead vector calculation algorithm to construct a histogram of $H^{(n)}(D)$ for a range of *n*:

```
Calculate steady state probabilities, πj;       // equation 10
Build transition matrix from control arc probabilities, P(1);
Determine target set D for the test data path unit;
Process P to build V for target set D;          // equation 6
Iterate dead vector length n=1,2,3,...
{
  Evaluate next state transition matrix V(n);   // equation 7
  Calculate H(n)(D);                            // equation 11
}
```

These results can then be processed to obtain a test completion graph.

2.4 Constraints and Simplifications

In order to simplify the design of the analysis tool, a number of constraints and simplifications are imposed upon the implementations that can be analysed.

There are two main restrictions placed on the control structures: first, there may be no VHDL sub-programs and second, there can be no parallel executed control states, other than the basic set of top-level processes.

In practice this places little restriction on the analysis as sub-programs can be inline expanded prior to analysis, and VHDL only allows the gross definition of concurrency via processes. During synthesis, operations are automatically parallelised through their simultaneous execution in a single control state, rather than using concurrent states, and the processes themselves are analysed separately.

In addition to the above, the other main simplification concerns the treatment of data dependent operation. As has been mentioned, conditional branches in the control graph have a substantial effect on the length of latent vectors obtained. These are, however, heavily data dependent and cannot therefore be accurately modelled without extensive analysis of the system during operation. At present, these factors are quantified through the manipulation the arc probabilities to favour worst case scenarios. For example, VHDL *wait* statements are implemented by way of a state that continually loops back to itself until a particular event occurs[10], usually a change on an input signal. In this situation, the arc probabilities are biased such that the probability of dropping through the statement is much greater than that of looping back to continue waiting. Future enhancements to the system, however, could determine the arc probabilities from user supplied data on typical input transitions, or simulation of a set of “typical” input vectors[19]. This technique is also suitable for estimating dynamic power consumption[20], a feature currently under development.

None of the problems avoided by the restrictions described above are insoluble, and they do not substantially affect the conclusions of the study. The data-dependent simplifications are much more difficult to quantify without performing detailed simulation based upon a range of typical input vectors, however, the use of worst case values for the transition probabilities ensures that “real-life” testability of an implementation will, if anything, be better than the figures calculated.

3. Experimental Results

The MCNC High Level Synthesis Design benchmarks feature a number of synthesisable VHDL designs[21,22]. Several of these have been synthesised and analysed using the methods described in the previous sections. Two of the designs, a Fast Fourier Transform processor (FFT)[21] and a RISC microprocessor (FRISC)[22] are discussed in detail in this section.

The designs are investigated in their initial and optimised implementations, the optimisation objectives being area and delay. The results show full test completion probability histograms for various implementations based upon a set of sample test sets for each of the main functional units. The figures used were generated by the TransTest package[15] and are listed in table 1. In addition, a number of specific units are examined in detail to illustrate the effects that the different optimisation strategies have on the latent vectors available for testing. These are displayed using the complete dead vector probability histograms and by plotting the test completion probability against test time curves for the individual units concerned. By associating the results with the implementations' control graphs, the main factors affecting in-line testability are identified. This information is then used to demonstrate how re-synthesis may be applied in order to improve the systems' in-line testability, and what effects any changes have on the user's initial optimisation criteria.

The results below are split into two sections. The first examines the FFT design in an attempt to determine the factors affecting the in-line testability of the individual units and pinpoint areas for possible improvement. These ideas are then extended in the second section which details the FRISC processor, presenting some methods that may be used to improve the design, and investigating the effect these changes have on the user's optimisation objectives.

3.1 Design in-line testability - FFT processor

The Fast Fourier Transform contains around 140 lines of behavioural VHDL code, which was processed by MOODS to synthesise both area and delay optimised implementations. The resulting control graphs are shown in figure 3. Both of these have the same general structure formed from two main *while* loops in the VHDL description. Within this framework, there are a number of significant differences, the most obvious of which is the smaller number of states in the delay optimised version. This results in a shorter control path and hence a quicker overall implementation. In addition the maximum state delay, that is the longest propagation delay for a single control state, is shorter allowing a higher maximum

clock speed. This improved performance is achieved by sacrificing area in order to increase the component count and enable a different scheduling scheme.

There are two main mechanisms involved in the optimisation process: unit sharing and graph compaction. Unit sharing takes two data path units, each activated by one control state, and combines these into a single unit with multiplexed inputs activated by a number of control states. This results in a reduction in area, but there may be a speed penalty to pay due to increased complexity resulting from the use of input multiplexors. In addition, combining, for example, a 16-bit and 32-bit adder into a shared 32-bit unit will significantly slow down the smaller instruction and may affect the maximum clock speed permitted depending on the schedule. Graph compaction utilises the slack time within a state, that is the difference between the state delay and the clock period to combine two adjacent control states into one, thereby shortening the control path.

Both of these mechanisms have a profound effect on the length and distribution of dead vectors. Figure 4 shows histograms of the test completion probability for each data path unit in the area and delay optimised implementations based upon the sample test sets. These results serve to illustrate a number of important factors influencing the in-line testability of an implementation. It should be noted that comparisons may only be drawn between the full set of units of the same type since each unit implements a different subset of instructions. For example, in the area optimised design (figures 3b and 4b) unit 29 implements all six of the system's subtract instructions, whereas the delay optimised version (figures 3a and 4a) spreads these across two subtractors, units 29 and 30.

The units in each histogram split into two general groups: arithmetic units and comparators. During optimisation none of the comparators will be shared since, for the technology library used, the additional hardware cost involved in sharing a unit is greater than the cost of a separate comparator. Comparing the results of the two optimisations it can be seen that the comparators in the area optimised implementation have a generally higher test completion probability than those in the delay optimised version. This is attributed to the greater number of control states (and hence longer execution time) as illustrated in figure 3b. In contrast, the arithmetic units in this implementation are much more heavily, thus the test completion probability for these units is lower than in the delay optimised implementation. The results for the two types of unit show that both increased sharing and decreased graph length conspire to limit the in-line testability of a synthesised design.

The factors influencing the in-line testability of the two implementations can be further understood through a detailed examination of a single operation, in this case the subtract instructions described above. The control states that activate subtract units are shaded in the control graphs of figure 3, and are analysed to determine the dead vector probability histograms, shown in figure 5. The graphs show a relatively smooth reduction in dead vector probability as the vector length increases. This is typical of designs containing several mutually exclusive branches, which result in a continual increase in the number and length of dead vectors obtained each time control passes around the main loop. From the dead vector histograms corresponding graphs of test completion probability versus test time can be obtained. These are shown in figure 6 which clearly illustrates the difference between the three units, in particular the more gradual curve for delay optimised unit 30. Considering the control graphs it can be seen that this difference is due to the localised nature of the states activating functional unit 30 (48, 58 in figure 3a), which maximises the length of the dead vectors obtained. In addition, the unit only appears in one of the two main mutually exclusive branches. Therefore any vector paths following the right hand branch will not activate the unit until control loops back to state 8, resulting in substantially longer dead vectors. In contrast, functional unit 29 in both the delay and area optimised implementations has executing states scattered fairly evenly throughout the control graph (shown shaded in figure 3). This leads to greater fragmentation of the dead vectors and hence a lower test completion probability. The application of these observations to improving the in-line testability of an implementation is discussed later.

3.2 Design Optimisation - FRISC Microprocessor

The FRISC microprocessor is a larger design of around 300 lines of behavioural VHDL. This is a relatively simple RISC processor based around a VHDL *case* construct which performs instruction decoding together with a small amount of reset and interrupt request code.

Unlike the FFT, the synthesised implementations have a much more linear control structure centred around the *case* control state from which emerges a large number of mutually exclusive branches. In the delay-optimised implementation, these branches result in the *majority* of the units being highly testable (having test completion probabilities ranging from around 30% to 98%), however in the area optimised version, complete unit sharing (ie. all functional units of *each* type are mapped onto one physical unit) results in an

implementation with only three arithmetic units, with test completion probabilities of 26%, 16% and 0.3%, as shown on the left-hand side of figure 7.

In order to improve in-line testability a certain amount of re-synthesis must be applied. This involves moving *away* from the user-specified goals on area and delay, to allow a greater testability to be realised. Clearly the dominating factor in FRISC is the unit sharing, especially within mutually exclusive branches (ie. the *case* statement). Considering the untestable adder (plus 14 in the figure 7), it is possible to split this unit up into its constituent instructions each implemented by a different data path unit. The effect of this unsharing on the test completion probability histogram is shown on the right-hand side of figure 7. Now most of the adders have a test completion probability of more than 90%, with the lowest figure at 19% - a considerable improvement. It is clear, however, that the number of units has increased out of all proportion to the rest of the design, resulting in a 75% increase in total area, and a 250% increase in the area of the functional units.

The problem is now how to improve the area optimised implementation's in-line testability, without having too severe an effect on its total size. As mentioned in the FFT results, the best testability figures are obtained by localising the use of a particular physical unit, and taking advantage of mutually exclusive branches to increase the length of the dead vectors. Figure 8 shows the result of attempting to share units only within the *same conditional* section, thus extending the dead vectors resulting from mutually exclusive control branches. This new graph describes an implementation with a considerably improved in-line testability when compared to the original, however the increase in total area is now a more acceptable 20%, with the increase in functional unit area being around 50%. It should be remembered that the test completion probability refers to *how often* a unit may be fully tested, as opposed to the fault coverage obtained. Thus figures of around 30 to 50% should be perfectly acceptable in many situations.

In some cases it may not be possible to obtain the desired figure even with all units fully unshared. Here, the only solutions may be either to use two units for alternate executions of the operation (resulting in a doubling of the unit area), or more likely, to modify the control graph and include some extra states to lengthen the dead vectors. Although ostensibly dummy states, these may also be used to re-schedule other operations in order to decrease the maximum state delay (and thus *increase* the maximum clock speed).

4. Final Remarks

This analysis has concentrated exclusively on testing the functional units that make up a design, however, one should not lose sight of the fact that registers and interconnects can also develop faults. Furthermore, the difference between conditional and unconditional execution of units needs extremely careful analysis; where the probability of execution is data-dependent *and* highly skewed the strategy employed here can deliver misleading results if the testability (defined in section 1) is not close to 100%. We also note that the results are dependent on the test strategy used - further interaction between units must exist if a single test controller is employed.

The work described here has taken as a starting point a synthesised design that has been optimised for area or delay, and has quantified the potential in-line testability of the design. The study confirms that in-line testing using design latency is feasible, although near 100% coverage has a disproportionate effect on the other design figures of merit. There are unquestionably, however, applications (nuclear installations, oil exploration) where reliability and pre-emptive fault indication substantially outweigh considerations of processing speed or silicon area.

Acknowledgements

The authors gratefully acknowledge the assistance of TransEDA Ltd., for the use of the TransTest[15] test coverage system used in the preparation of this work. The work was funded by EPSRC grant GR/K00752

References

[1] R.C. Aitken, “An Overview of Test Synthesis Tools”, IEEE Design and Test of Computers, Summer 1995, pp. 8-15.

[2] V. Chickername, J. Lee and J.K. Patel, “Addressing design for testability at the architectural level”, IEEE Transactions on computer-aided design, 1994, CAD-13, no 7, pp 920-934.

[3] S. Bhattacharya and S. Dey, “H-SCAN: a high level alternative to full scan testing with reduced area and test application overheads”, 1996, 14th VLSI test symposium, pp 74-80.

[4] M. Abramovici, M.A. Breuer and A.D. Friedman, “Digital Systems Testing and Testable Design”, IEEE Press, 1990, ISBN 0-7803-1062-4.

[5] V.D. Agrawal, C.R. Kime and K.K. Saluja, “A tutorial on built-in self test (I): Principles”, IEEE Design and test of computers, 1993, DTC-10, no 1, pp 73-92.

[6] D. Gizopoulos, A. Paschalis and Y. Zorian, “An effective BIST scheme for datapaths”, 1996, IEEE International test conference, pp 76-85.

[7] A.D. Brown, K.R. Baker and A.C. Williams, “On-Line Testing of Statically and Dynamically Scheduled Synthesized Systems”, IEEE Transactions on Computer-Aided Design, CAD-16, No. 1, January 1997, pp. 47-57.

[8] K.R. Baker, A.J. Currie and K.G. Nichols, “Multiple objective optimisation in a behavioural synthesis system”, IEE Proceedings-G, 140, no. 4, August 1993, pp 253-260.

[9] K.R. Baker, A.D. Brown and A.J. Currie, “Optimisation Efficiency in Behavioural Synthesis”, IEE Proceedings-G, 141, no. 5, October 1994, pp 399-406.

[10] A.C. Williams, “A Behavioural VHDL Synthesis System using Data Path Optimisation”, PhD. Thesis, University of Southampton, October 1997.

[11] S. Dey and M. Potkonjak, “Nonscan design fro testability techniques using RT-level design information”, IEEE Transactions on computer-aided design, CAD-16, no 12, pp 1488-1506.

[12] I. Ghosh, A. Raghunathan and N.K. Jha, “Design for hierarchical testability of RTL circuits obtained by behavioural synthesis”, IEEE Transactions on computer-aided design, 1997, CAD-16. no 9, pp 1001-1014.

[13] I. Ghosh, A. Raghunathan and N.K. Jha, “A design for testability technique for RTL circuits using control/data flow extraction”, IEEE Transactions on computer-aided design, 1998, CAD-17. no 8, pp 706-723.

[14] G.L. Craig, C.R. Kime and K.K. Saluja, “Test scheduling and control for VLSI built-in self-test”, IEEE Transactions on computers, 1988, C-37, no 9, pp 1099-1109.

[15] TransTest User Guide, Version 1.0, TransEDA Ltd., 1992.

[16] IEEE Standard VHDL Reference Manual, IEEE Std 1076-1987, IEEE Catalogue No. SH11957, 1987.

[17] P.G. Harrison and N.M. Patel, “Performance Modelling of Communication Networks and Computer Architectures”, Addison-Wesley, 1994, pp 81-163, ISBN 0-201-54419-9.

[18] D. Freedman, “Markov Chains”, Springer-Verlag, 1983, ISBN 0-387-90808-0.

[19] F.N. Najim, “Transition Density: A New Measure of Activity in Digital Circuits”, IEEE Transactions on Computer-Aided Design, CAD-12, No. 2, February 1993, pp. 310-323.

[20] D. Singh, J.A. Rabaey, M. Pedram, P. Catthoor, S. Rajgopal, N. Sehgal and T.J. Mozdzen, “Power Conscious CAD Tools and Methodologies: A Perspective”, Proceedings of the IEEE, 83, No. 4, April 1995, pp. 570-594.

[21] Microelectronics Centre of North California, “High-Level Synthesis Workshop Benchmarks”, 1989, 1991.

[22] P.R. Panda and N. Dutt, “1995 High Level Synthesis Design Repository”, University of California, Irvine, Technical Report #95-04, 7 February 1995.

Table and figure captions

Table 1 Functional unit test sets

Figure 1 Design space

Figure 2 Example control graphs

Figure 3 Area and delay optimised FFT control graphs

Figure 4a Delay optimised FFT test completion probability per unit

Figure 4b Area optimised FFT test completion probability per unit

Figure 5 FFT subtractor dead vector probabilities

Figure 6 FFT subtractor test completion probability vs. test time

Figure 7 Area optimised FRISC test completion probability per unit, with and without fully unshared adder

Figure 8 Area optimised FRISC test completion probability per unit optimised for in-line testing

DP unit type (number)	Bit width	Number of test sets	Fault coverage (%)
Add (14)	32	22	98.4
Minus (15)	32	22	99.6
	64	30	99.6
Rshift (32)	16	16	N/A
Comparators (20-25)	1	3	100
	16	18	100
	32	35	100

Table 1 Functional unit test sets

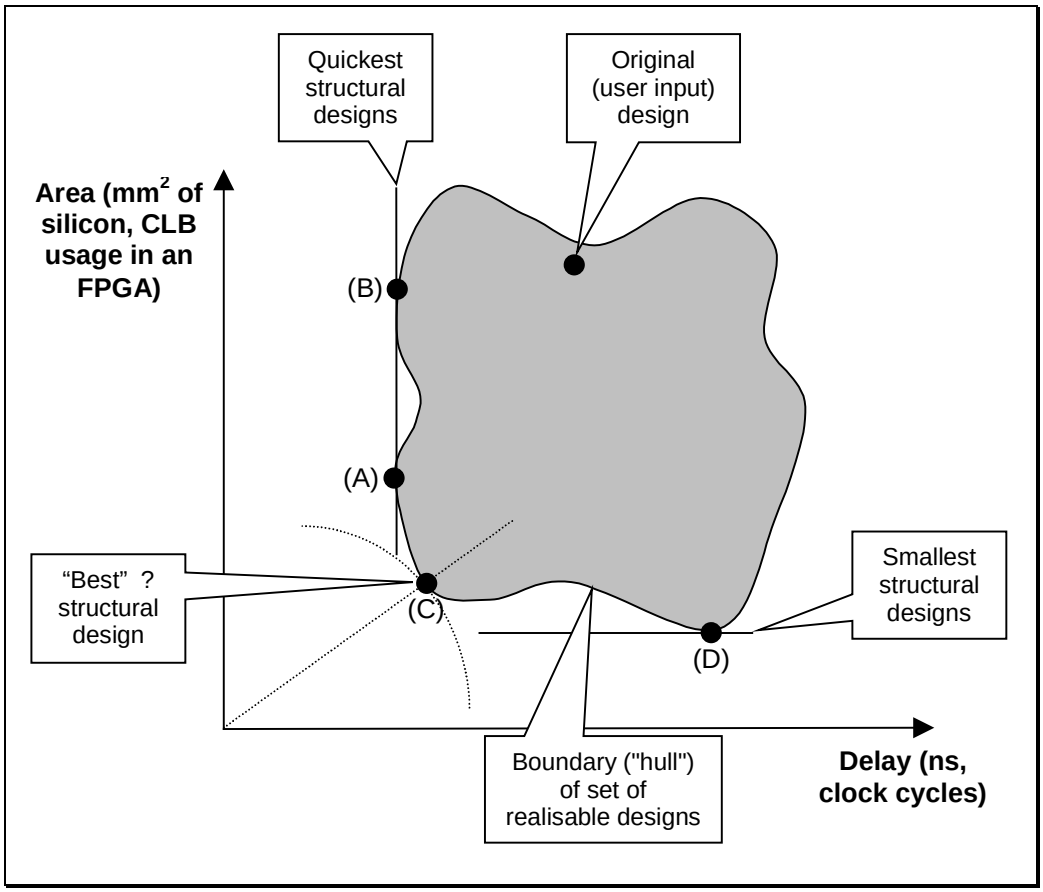

Figure 1 Design space

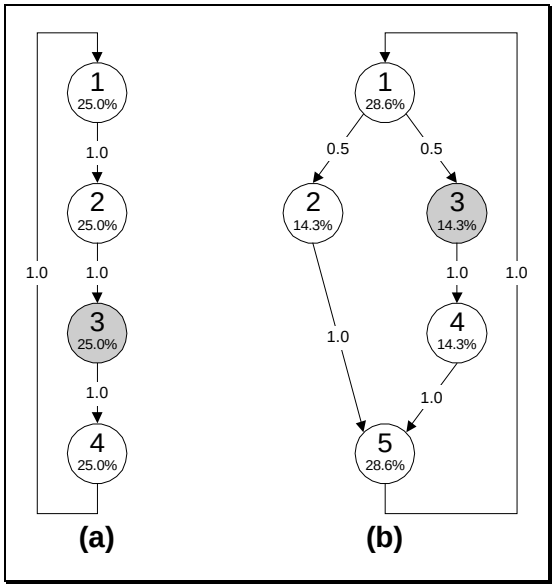

Figure 2 Example control graphs

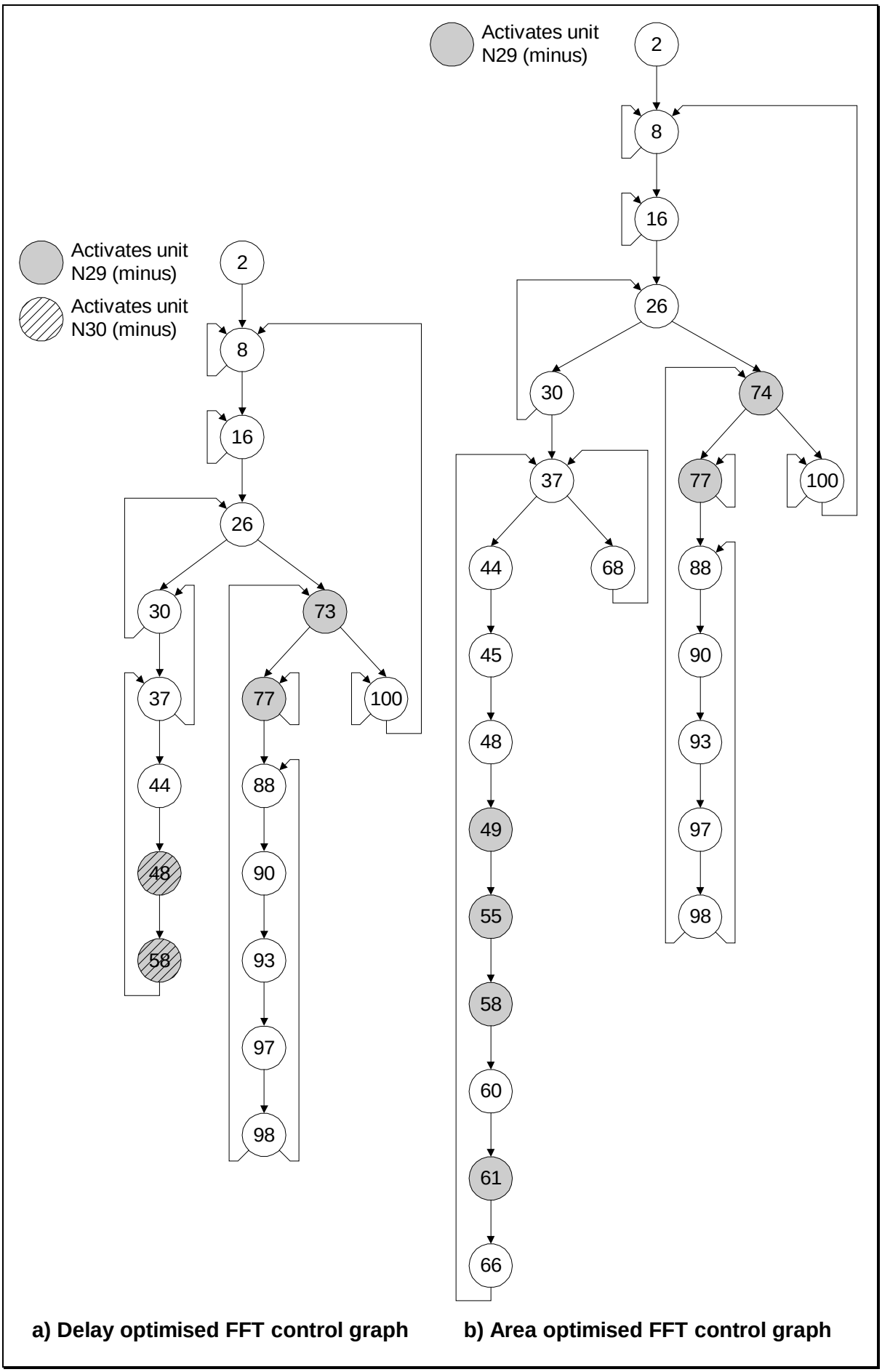

Figure 3 Area and delay optimised FFT control graphs

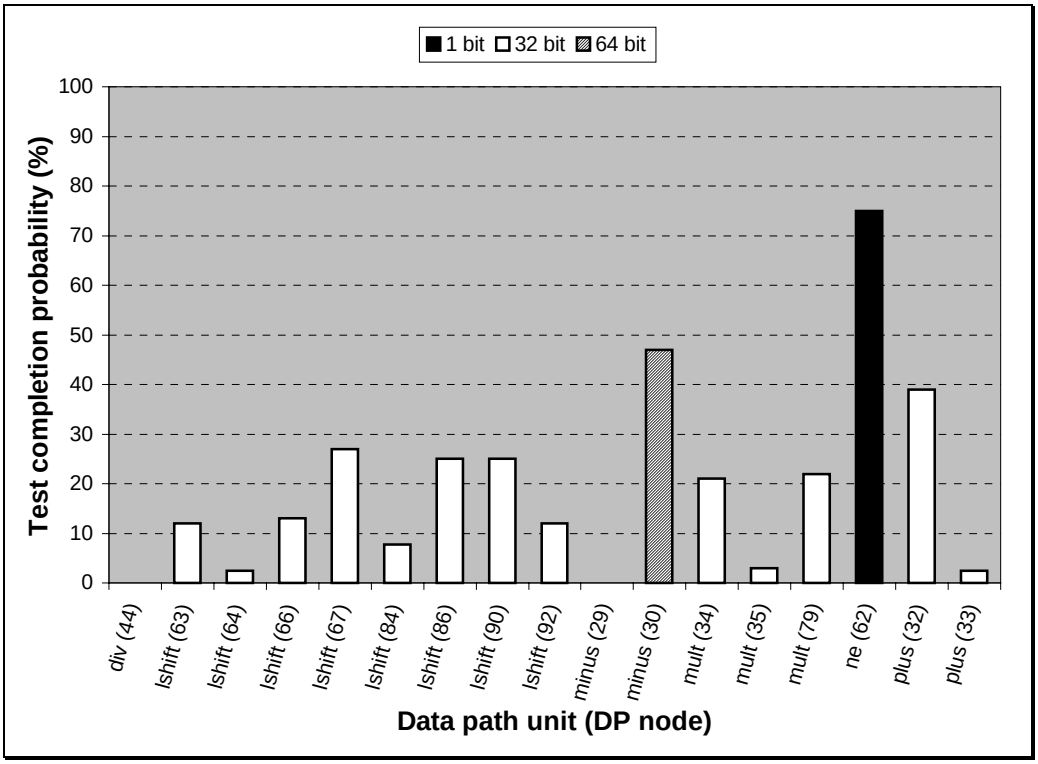

Figure 4a Delay optimised FFT test completion probability per unit

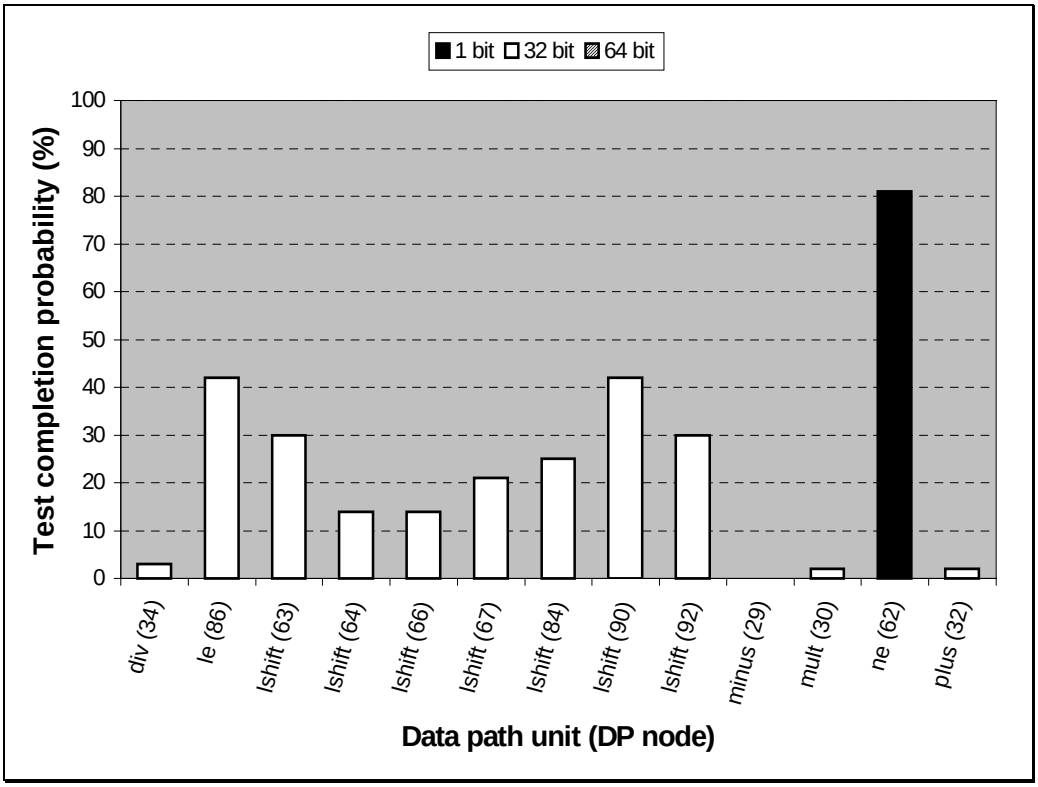

Figure 4b Area optimised FFT test completion probability per unit

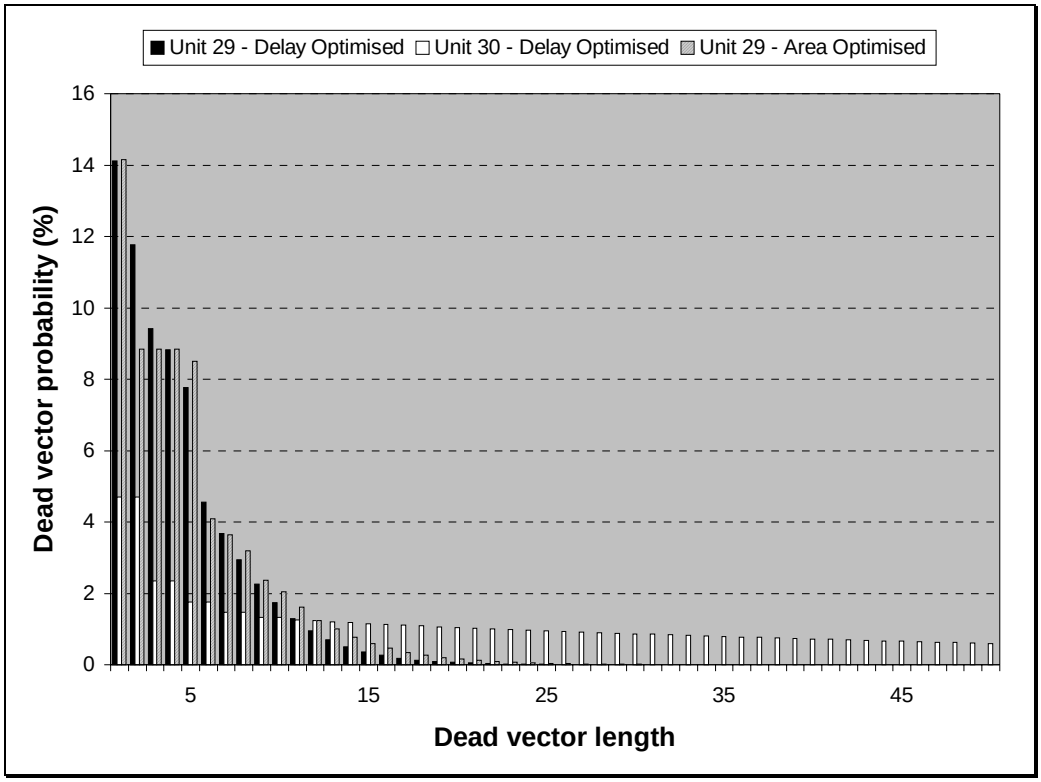

Figure 5 FFT subtractor dead vector probabilities

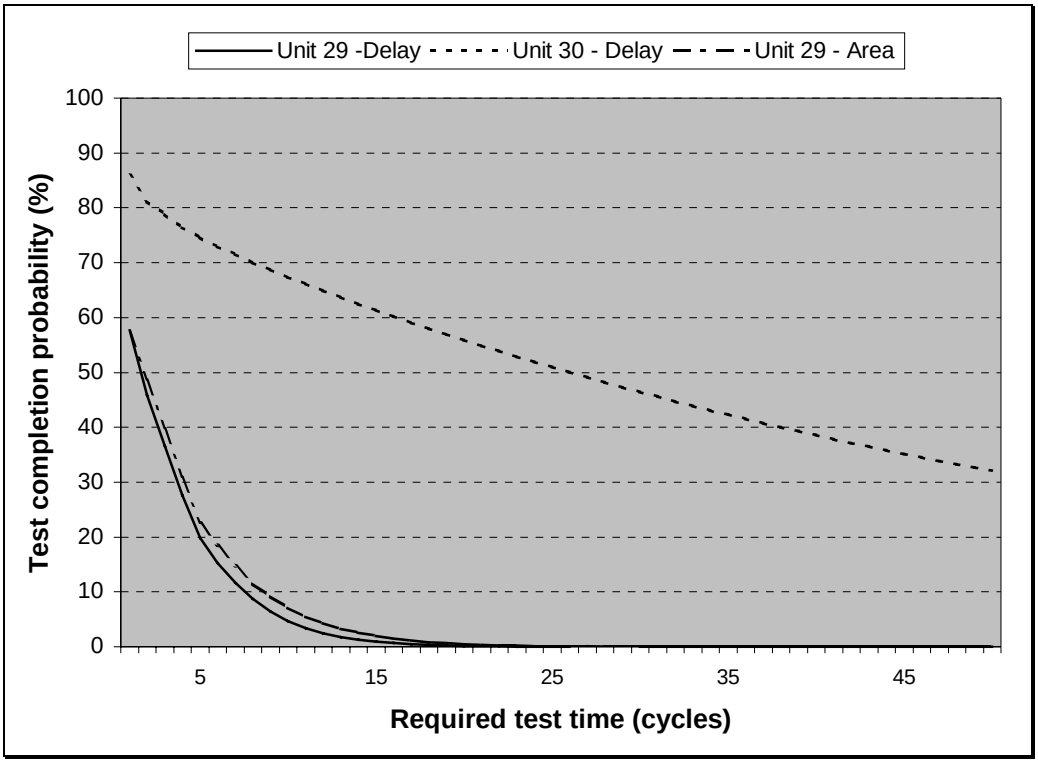

Figure 6 FFT subtractor test completion probability vs. test time

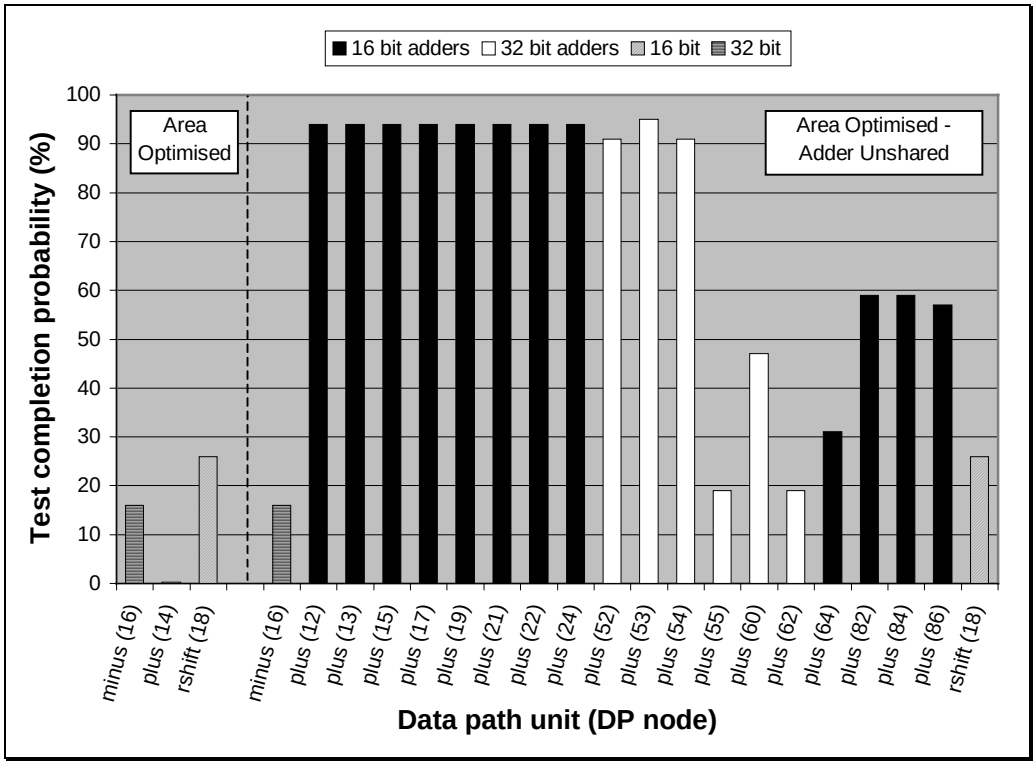

Figure 7 Area optimised FRISC test completion probability per unit, with and without fully unshared adder

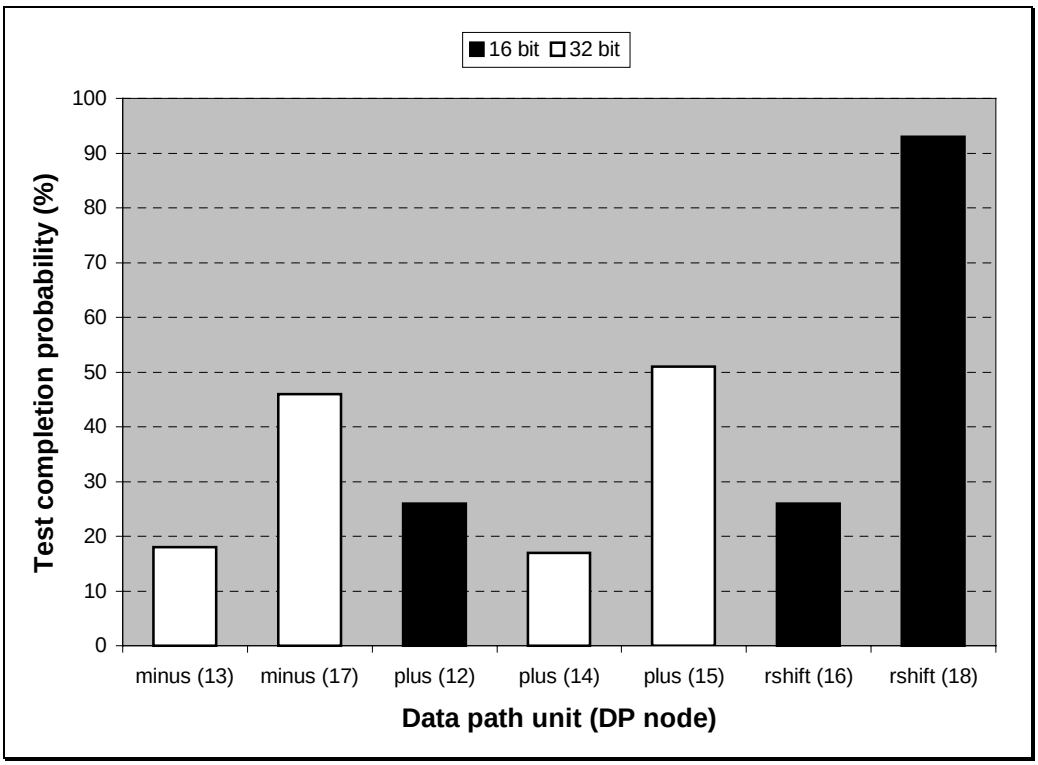

Figure 8 Area optimised FRISC test completion probability per unit optimised for in-line testing