Transformations and Resynthesis for Testability of RT-Level Control-Data Path Specifications

Subhrajit Bhattacharya, Franc Brglez, *Member, IEEE* and Sujit Dey, *Member, IEEE*

Abstract— **This paper introduces a technique to transform a given RT-Level design, consisting of control logic and data path, into a functionally equivalent, minimized design which is 100% testable under full-scan at the gate level. The proposed RT-Level optimization technique uses the RT-Level structure and exploits the interaction between the control and the data path. Our approach maintains the RT-Level design hierarchy while performing RT-Level transformations of initially specified data path, followed by resynthesis of control using *don't cares* extracted from the data path. Experiments with several RTL benchmarks demonstrate the effectiveness of the technique in generating fully testable designs. In addition, comparison with logic-level techniques show the advantages of the proposed technique as an optimizing tool to produce circuits with reduced area and delay.**

1 Introduction

The goal of behavioral synthesis is to produce a RTL hardware description that meets various design constraints. The major tasks involved are scheduling of operations into control steps and allocating resources (functional units and registers) to perform operations and storage. A summary of recent approaches on these topics is available in [1], [2], [3].

In most approaches, scheduling is performed in a way that meets the resource and timing constraints before the actual RTL realization. Since we want to consider general designs consisting of control as well as data operations, as opposed to data-intensive designs like DSP applications, we examined two scheduling strategies applicable to control-dominated designs in greater detail: the AFAP schedule as described in [4] and an alternative MINC schedule, introduced and described in [5]. Our experiments confirm experience of other researchers: regardless of which schedule one uses, none of the RTL designs that are produced are 100% testable at the gate level. Also, the initial RTL designs can be significantly minimized before submitting to final technology mapping.

A general RT-Level design consists of control and data paths. Since the structural description of the data path is not amenable to two-level and multi-level minimization, and since extracting its state transition graph is not feasible, the logic-level synthesis for testability techniques are applicable only to the control part of the RT-Level description [6], [7]. More recently, techniques have been proposed to synthesize testable data paths from high level descriptions [8], [9], [10], [11]. While the control and data path can be made fully testable, the disadvantage of considering two parts of the synthesis process separately is that the whole design may not be fully testable.

An RTL description consisting of control and data path can be optimized at the logic level using known heuristics which minimize multi-level logic: either by direct redundancy removal [12], by using *don't cares* and factorization techniques [13], [14] or by combining both techniques on selected partitions [15], [16]. However, lack of knowledge of the RTL structure at the logic-level may make the logic-level techniques, like extracting *useful* don't cares, prohibitively expensive. Also, application of the logic-level techniques to the complete design destroys the RT-Level hierarchy.

An alternate and more efficient approach to optimize RT-Level designs is to use RT-Level transformations. The CALLAS high-level synthesis system uses RT-Level transformations to minimize the number of multiplexors in an RTL design[17]. In this paper, we propose a general methodology to generate optimized RTL control-data path designs which are 100% testable under full-scan at the gate level. Also, all false paths are implicitly removed from the data path. Our approach uses the hierarchy of the RT-Level design, and the interaction between control and data path. Using the knowledge of the RTL structure allows application of simple RT-Level transformations on the data path. The RTL hierarchy also allows for easy calculations of *exact* observabilities of data signals, and *exact* observability don't cares for control signals. The resultant synthesis process consists of transformations applied to the data path, followed by extracting don't cares from the data path to optimize the control.

The attractive features of the proposed RT-Level transformations are the following.

1. The RTL hierarchy is maintained.
2. Data path undergoes RT-Level transformations instead of gate level transformations. For instance, RTL units like Functional Units, multiplexors and registers are optimized.
3. Using the RTL structure simplifies extracting *don't cares* from the data path to minimize the control logic. Lack of knowledge of the RTL structure at the logic level makes extracting the same *don't cares* at the logic level computationally very expensive, if not impossible.
4. All *combinational* false paths in the data path are im-

The research of Subhrajit Bhattacharya has been supported by a benchmark grant from ACM/SIGDA and a grant from C&C Research Laboratories, NEC USA.

plicitly removed. This too is simpler at the RT-Level than at the gate level. A circuit with false paths can be fully testable for single stuck-at-faults. However, removing false paths in the data path optimizes logic as well as reduces the cost of test pattern generation.

5. The final circuit is 100% testable under full scan.

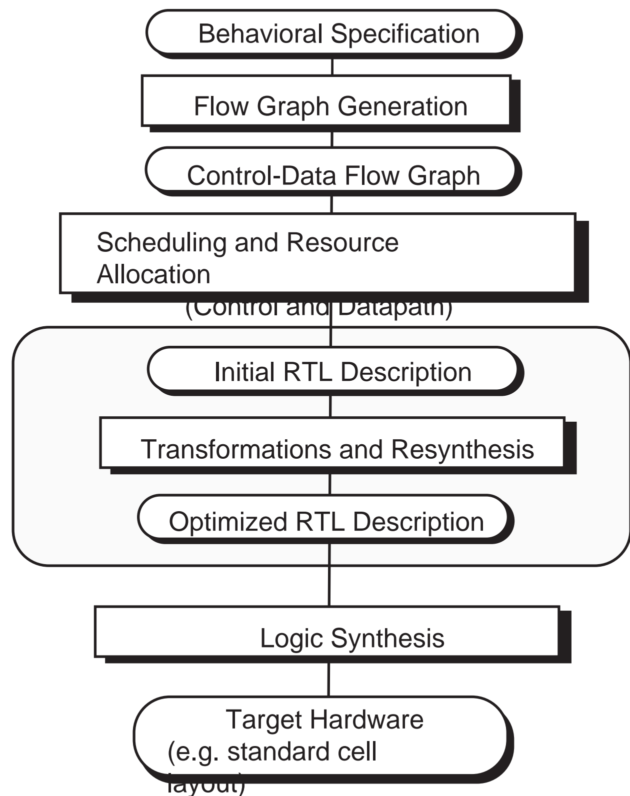

Figure 1: The Context of the Proposed RTL Transformations and Resynthesis.

We have applied our RT-Level optimization techniques on several RT-Level benchmarks [5]. We report on our results in the context of the overall synthesis process as depicted in Figure 1. Given a behavioral specification, we will have synthesized different RTL implementations, depending on the schedule we use. Experimental results with benchmarks demonstrate that while a scheduling strategy does affect the initial standard cell layout area, critical path delay and gate level testability, the proposed RTL transformations and resynthesis reduce this variance significantly while consistently achieving 100% fault coverage at gate level, regardless of the initial schedule.

The paper is organized as follows. In Section 2, we use a benchmark example to illustrate the RT-Level hierarchy and demonstrate the major steps of the proposed approach. In Sections 3 and 4, we formalize two important transformations that allow us to generate a canonical representation of the data path. We show in Section 5 how to efficiently extract and use *don't cares* associated with the canonical form of the data path to minimize the control logic such that the network combining both becomes fully testable, while also removing all false paths in the data path. The section on experimental results tabulates interesting effects of the transformations on area, delay and testability of several RT-Level benchmarks. The experiments show that while none of the initial designs are fully testable, the transformations almost always produce fully testable designs, while consistently reducing area and delay.

2 RT-Level Hierarchy and Transformations: An Illustration

A typical RT-Level design consists of a data path (DP) and control logic. We partition the control logic into three blocks, the encoding logic EL, the decoding logic DL and the control logic CL. The RT-Level hierarchy is illustrated by Figure 2, which shows a partial RT-Level specification of the benchmark Fancy.b [5]. The RT-Level description was obtained by scheduling the Control Data Flow Graph (CDFG) from a behavioral description of the benchmark, shown in Appendix B, using the MINC scheduling technique [5]. For simplicity, not all of the control signals in the block DL are specified. Besides the four major partitions, the data path registers P and state registers S are the other constituents of the RTL specification.

The data path (DP) block consists of a structural description of multiplexors (muxes) and functional units (FUs). The muxes are typically used to route the output of data path registers and FUs in the data path back to the data path registers. This corresponds to assignment statements in a high level description. The data path network follows two design rules. The data input signals to the FUs and muxes are either outputs of registers, muxes or FUs. The control signals to the muxes can only come from either the DL block or the data path registers. As will be seen later, the design rules allow simple algorithms for data path optimization.

The control of a general design consists of three parts, a functional description of an FSM implementing the schedule (CL), a structural description of comparators implementing the conditionals in the high level description (EL) and a decoding logic (DL). The DL uses the state information from the FSM state registers S and the evaluation of the conditionals in EL to control the data path by switching the muxes. While the CL and DL blocks are amenable to two-level and multi-level minimization, the structural description of the comparators in the EL block is not. Hence, the EL block is maintained separately while minimizing CL and DL using *don't cares* derived from the data path. However, any comparator module with constant inputs is made irredundant separately before using it in the circuit.

While each block in an RT-Level description may be in itself irredundant, the overall circuit may not be fully testable, in spite of all the scan registers shown. The initial RT-Level design of benchmark `Fancy.b` is shown in Figure 2. The complete design is not 100% testable, as shown in Table 5. Using RT-Level transformations introduced in this paper, the initial RT-Level design (Figure 2)

can be transformed into the optimized RT-Level design

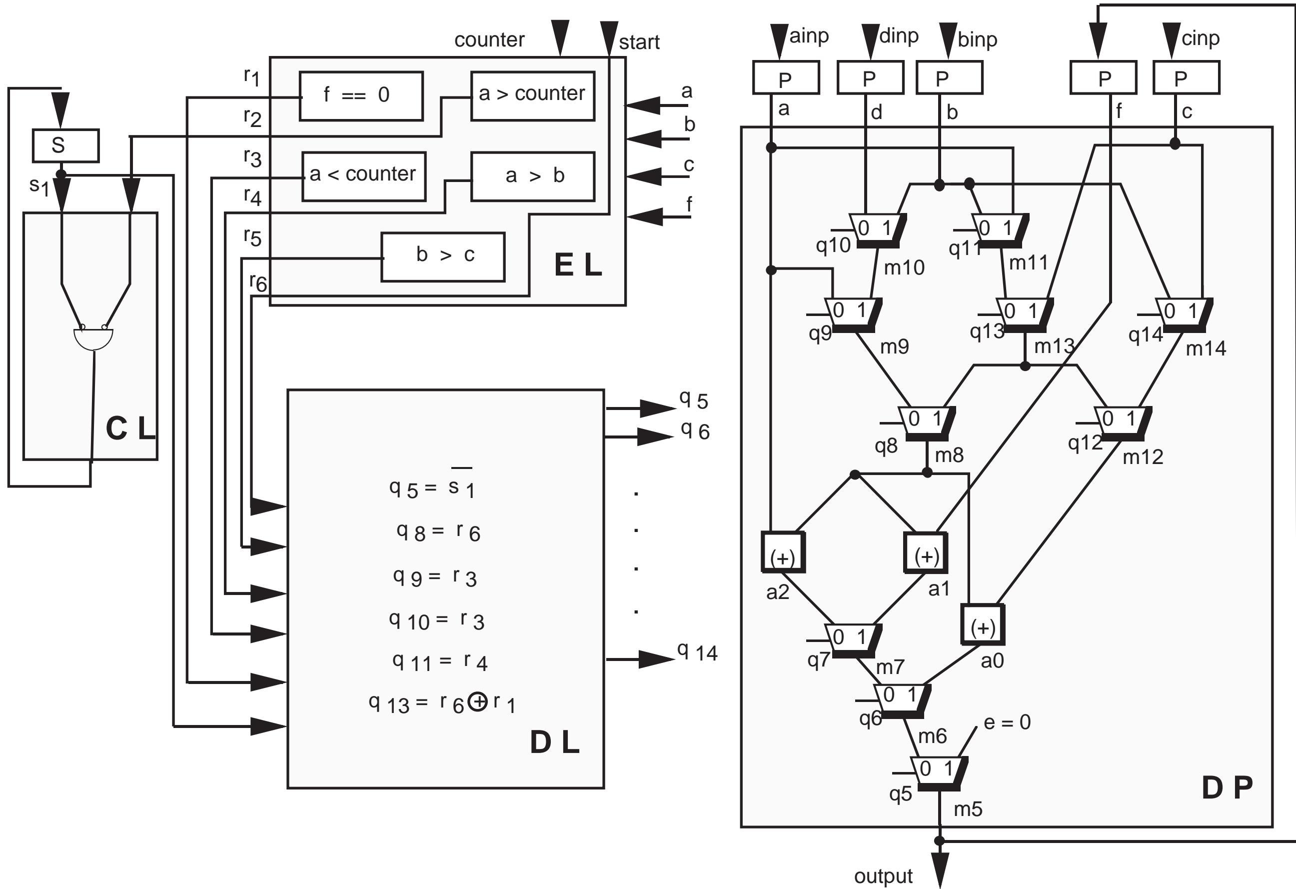

Figure 2: The initial RT-Level design of the benchmark `Fancy.b`.

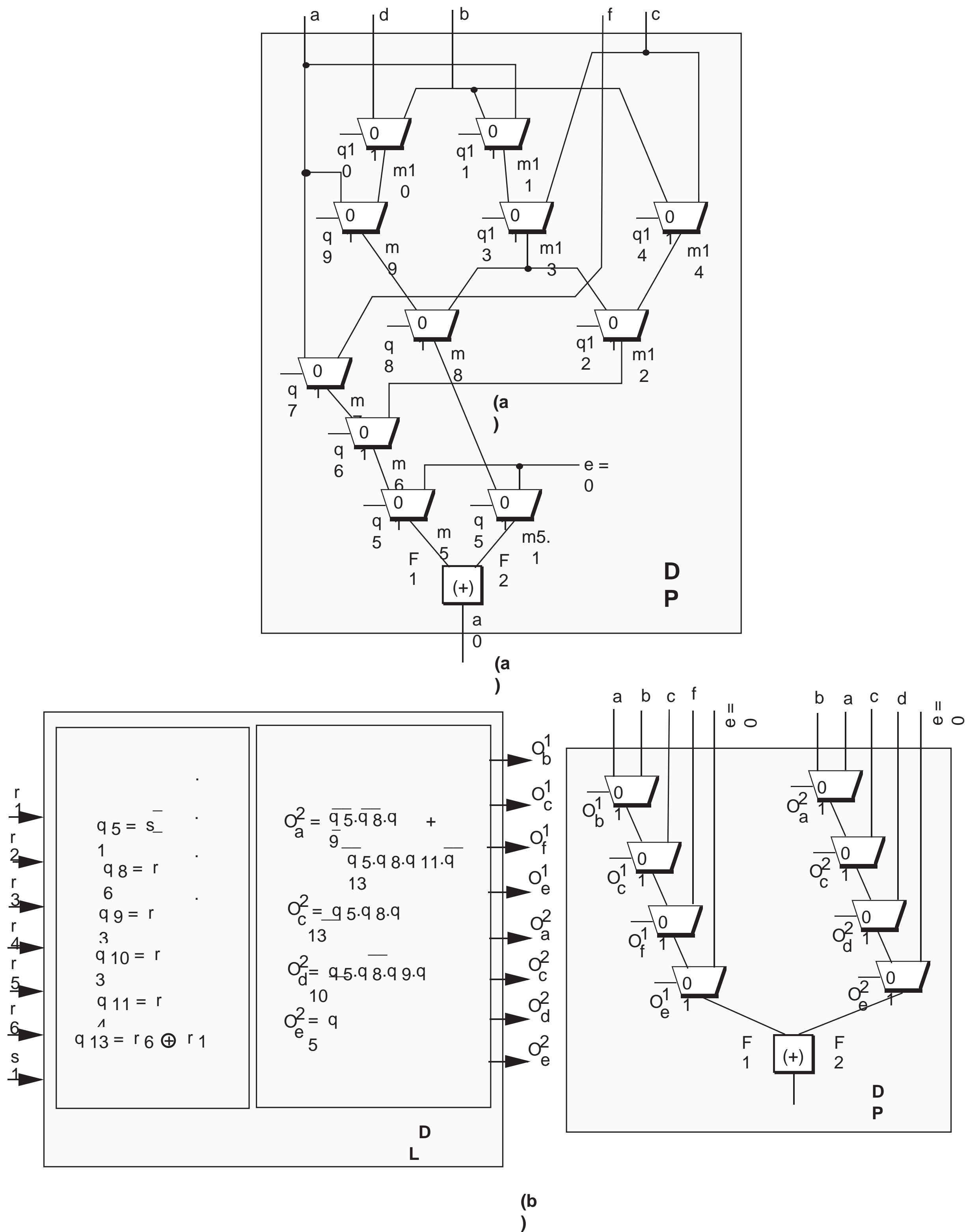

Figure 3: Fancy.b Example - (a) The data path transformed into a Mux-FU Cascade. (b) The canonical form of the data path and the change in the control logic (DL).

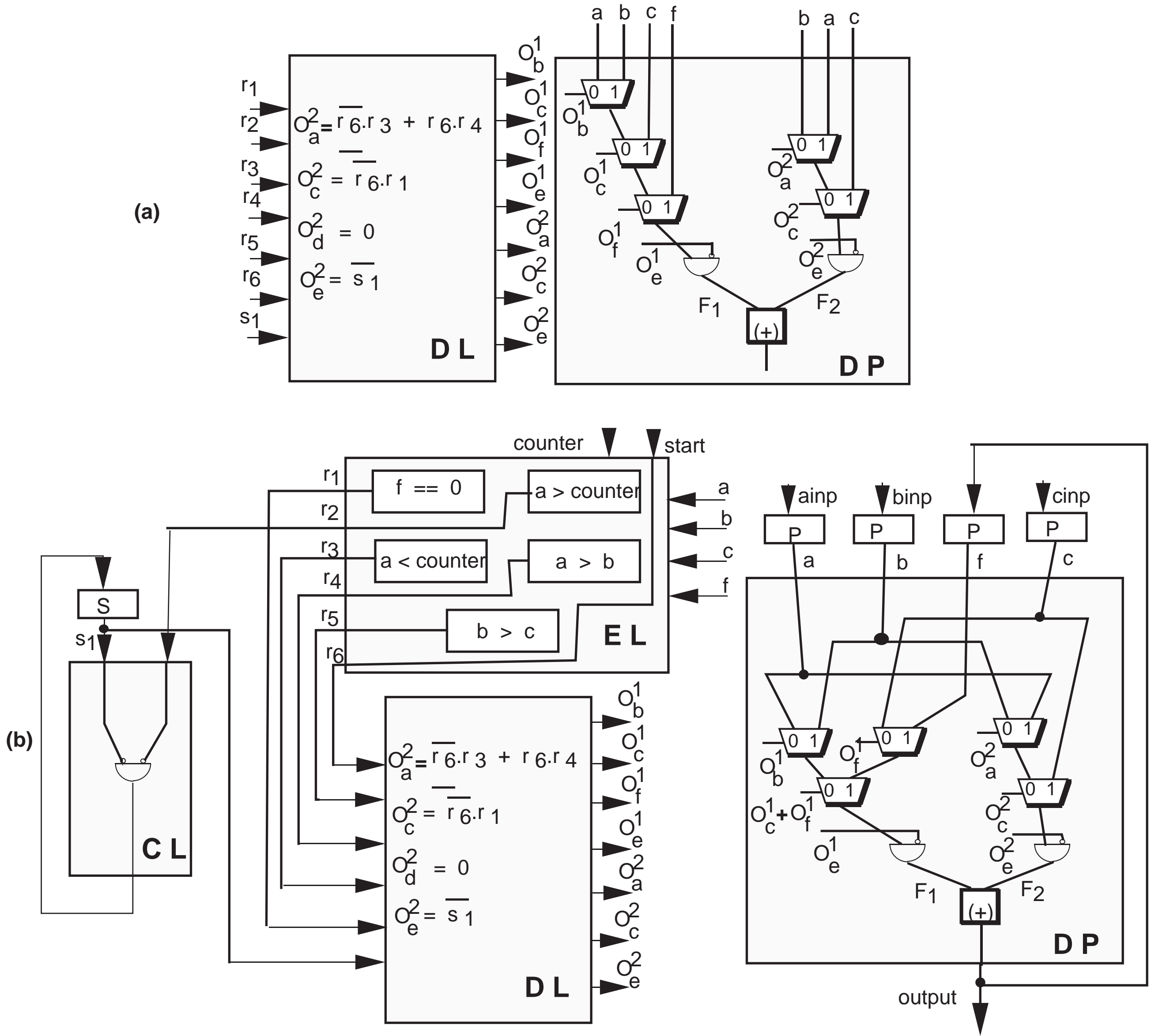

Figure 4: `Fancy.b` Example - (a) Effect of minimizing control using data path don't cares and constant propagation. (b) The final RTL design after mux balancing.

shown in Figure 4(b), which is 100% testable under full-scan at the gate level.

The dependencies of the controlling inputs in this example give rise both to redundancies as well as to false paths. The initial data path is first transformed into a cascade of multiplexors and functional units, thereby reducing the functional units to its minimum, as shown in Figure 3(a). The second transformation converts the data path into a canonical form, minimizing the number of multiplexors. The two chains of multiplexors in Figure 3(b) represent the canonical form which is equivalent to the network of multiplexors in Figure 3(a). The new control signals are simply data input observabilities, e.g. signal O_a^2 represents the observability of data input a at output F_2 of the mux network. Consequently, the control logic block DL changes as shown in Figure 3(b).

The next step is identifying *don't cares* from the data path, and using the don't cares to minimize the control logic. For example, when control signal $O_c^2 = 1$, the value of the control signal O_a^2 does not affect the output of the data path. Hence, the ON-set minterms of O_c^2 are *don't cares* for signal O_a^2, and can be used to minimize O_a^2. Minimizing the control logic using data path *don't cares* identifies that $O_d^1 = 0$. This indicates that the data input d is redundant, allowing for removal of the last false path, and thereby further minimizing the data path. The multiplexor corresponding to the constant signal e is also pruned. The optimized control (DL) and data path of the benchmark `Fancy.b` are shown in Figure 4(a). Finally, the multiplexor chains are balanced, generating the final RT-Level design shown in Figure 4(b).

While the initial RTL design (Figure 2) has 10 multiplex-

ors and 3 adders, the final RTL design (Figure 4(b)) has only 5 multiplexors and 1 adder. Most importantly, while the initial design is not fully testable, the final design is 100% testable under full-scan at the gate level. Various parameters, reflecting the area, delay and gate-level testability, of the initial and final RT-Level design of the benchmark *Fancy.b* are tabulated in Table 5.

3 Transforming the Data Path into a Mux-FU Cascade

The first transformation applies to the data path of the RTL design. The transformation consists of converting a data path of multiplexors (Mux) and functional units (FU) into an ordered network of multiplexors cascaded with a network of functional units (Mux-FU network). It involves propagating the functional units across the multiplexors. The Mux-FU network has the property that the data input of the muxes can either be primary data inputs or the output of a mux. Similarly, the outputs of an FU can be either a primary output or the input to another FU. The motivation for this transformation is twofold:

- Preparing the data path for a subsequent step of converting the multiplexor network into a canonical form, which ensures removal of all *combinational* false paths from the data path.
- Reducing the number of FUs required, thereby increasing resource sharing.

A brief word on notation. The nodes of the data path network represent muxes and functional units. We designate the nodes connected to the "0" and "1" data input, control input and output of the multiplexor M_i as l_i, r_i, c_i, m_i respectively. The operator corresponding to a FU node "x" is referred to as "op(x)".

In this section, we give an algorithm to transform a data path of multiplexors and functional units into a Mux-FU network. It is assumed that functional units have two inputs (operands) and the operations they perform have an identity element. The data path transformation can be achieved in terms of two operations, *propagate* and *duplicate*, which we describe below.

The *propagate* operation is invoked when each input of a multiplexor M_i has a functional unit of the same type, that is, $(op(l_i) = op(r_i))$. It simply propagates the FUs through the mux M_i, effectively merging the functional units and introducing two multiplexors at each input of the propagated functional unit. The multiplexors now multiplex the inputs to the two functional units in the initial network and both muxes are controlled by c_i. If any of the FUs has multiple fanout, the FU is duplicated, one copy for the fanout feeding to the multiplexor M_i, and the other copy for the other fanouts. The propagation of the FUs from the inputs to the output of multiplexor M_i is done subsequently. The *propagate* operation is illustrated in Figure 5, which shows two adders propagated to the output of the multiplexor.

In a general data path, both the inputs of a multiplexor may not have FUs, or the FUs on the inputs may be of different types. This is illustrated by the data path shown in Figure 6(a). Multiplexor M_1 has an adder on one input and a multiplier on the other input. In that case, the *duplicate* operation is invoked so that both inputs of the multiplexor have an FU of the same type. The *duplicate* operation is invoked either when one of the inputs has a FU node and the other one does not or when the FU's on the left and right input of the mux are different.

In the latter case, either the left FU or the right FU could be duplicated before invoking the propagate operation. The FU chosen could affect the final delay or area but not the testability. Since our primary goal in this paper is testability, we arbitrarily choose the left FU to simplify the algorithm. In the former case, the operation introduces an FU at the mux input which does not have a FU node. This new FU node has one input corresponding to the input node to the mux (before introduction of the FU), the other input is the identity element of the operation the FU performs. This transformation enables propagation through the multiplexor, while preserving the functionality of the data path. The second case of assymetric FU's is dealt with similarly.

Figure 6 illustrates the process of propagating FUs to produce a Mux-FU network. A data path consisting of multiplexors and FUs is shown in Figure 6(a). Since there are FUs on the path from multiplexor N to multiplexor M, the data path needs to be transformed into a Mux-FU network. Figure 6(b) shows the effect of the duplicate operation on multiplexor M in Figure 6(a). Following the duplicate operation, the adders are propagated by a subsequent invocation of the *propagate* operation, as illustrated in Figure 6(c). Finally, a Mux-FU network is achieved by propagating the two multipliers as shown in Figure 6(d). We outline the algorithm which transforms a network of muxes and FU's into a Mux-FU network, using the propagate and duplicate operations described above.

Algorithm 1 (Deriving the Mux-FU Cascade)

```
1  for each mux M_i in a levelized order do {
2      if (l_i is a FU) || (r_i is a FU) {
3        if (l_i is a FU) && (r_i is a FU) {
4            if op(l_i) = op(r_i) {
5                propagate(M_i); }
6            else {
7                duplicateFU(M_i,l_i,right);
8                propagate(M_i); }}
9        else if (l_i is a FU) && (r_i is not(FU)) {
10           duplicateFU(M_i,l_i,right);
11           propagate(M_i); }
12       else if (l_i is not(FU)) && (r_i is FU) {
13           duplicateFU(M_i,r_i,left);
14           propagate(M_i); }}}
```

The transformation converting the data path into a Mux-FU network may affect two parameters of the RTL design: the number of functional units and the delay of the circuit. If there is no fanout from any FU in the original data

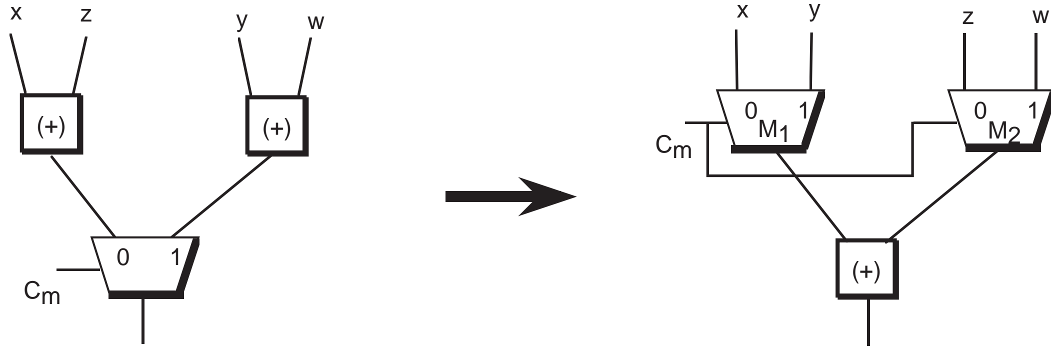

Figure 5: The Propagation Procedure for Functional Units.

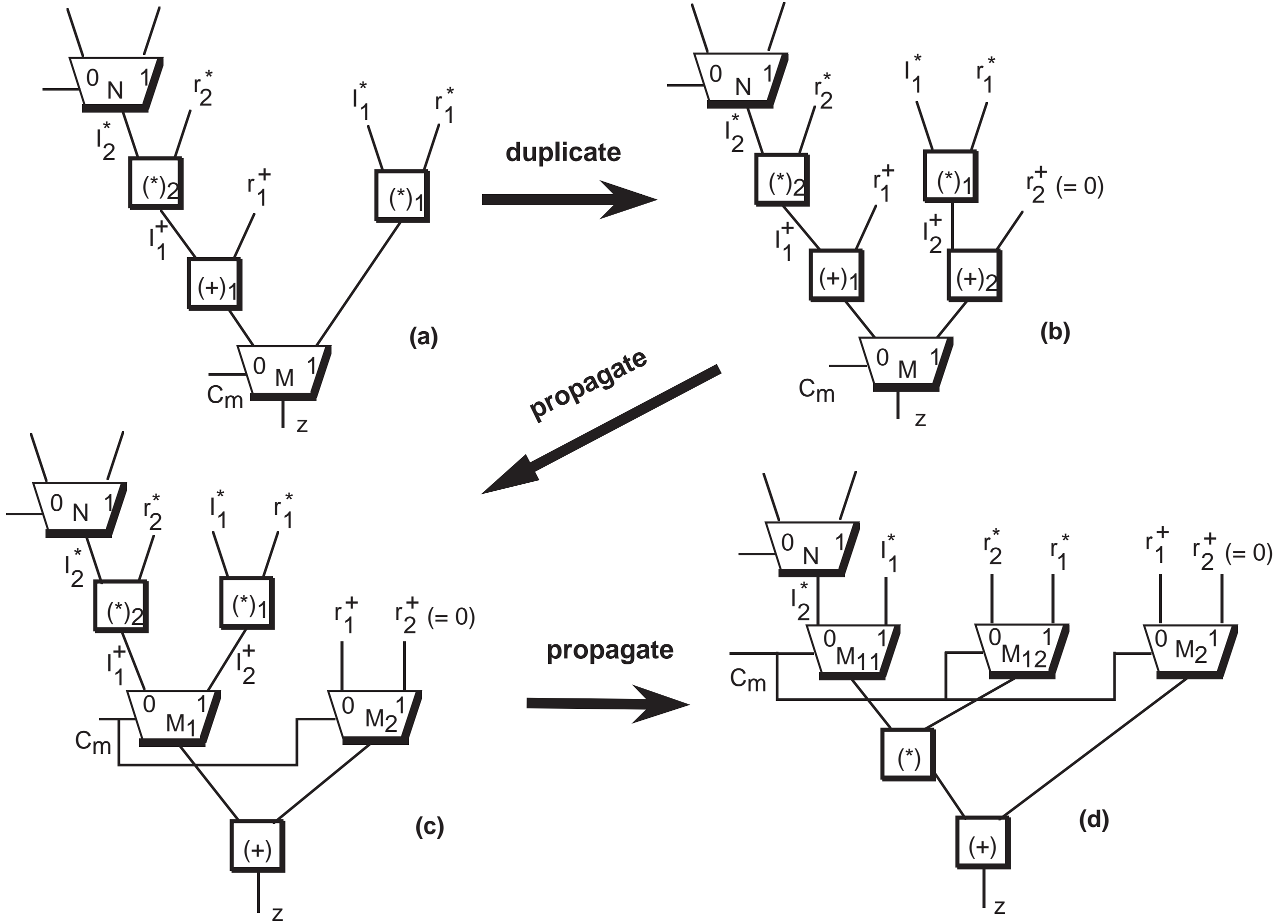

Figure 6: Data-Path Transformation with Heteregenous Functional Units.

path, it can be shown that the transformation will always decrease the number of FUs, thus improving resource sharing. However, in a more general case, the number of FUs may increase on propagation as for the QRS benchmark. In this situation, we do not propagate the FUs. Similarly, the delay of the RTL design may be affected while propagating FUs. For example, propagating two FUs of different types to the output of the multiplexor may increase the length of the critical path. Any increase in number of FUs or delay can be viewed as penalties for improved testability. We are currently investigating the trade-off between resource sharing, delay and testability.

In the next section, we show how we can transform a multiple output multiplexor network as derived by Algorithm 1 into a canonical form. The canonical form can be resynthesized, as discussed in Section 5, to result in a circuit which is not only fully testable, but also free from false paths.

4 Transforming a Multiplexor Network into a Canonical Form

After the data path has been transformed into a Mux-FU cascade, the mux network is transformed into a canonical form. The canonical network consists of disjoint mux subnetworks, one for each output of the Mux-FU cascade. These subnetworks can be realized as a chain of muxes or as partially or fully balanced mux trees. Transforming the mux network into a canonical form makes the following tasks simple: removal of *false paths* in the data path and extracting *don't cares* from the data path to minimize the control.

Figure 3(a) shows the Mux-FU cascade generated from the data path shown in Figure 2. Given the mux network in Figure 3(a), the equivalent canonical mux networks corresponding to output F_1 and F_2 are shown in Figure 3(b). In the canonical mux network corresponding to output F_1, the control signals to the muxes are the observabilities of the data inputs at the output F_1 in the initial mux network of Figure 3(a). The same is true for the canonical mux network corresponding to F_2. In this section, we first show how the observabilities of the data inputs of a Mux-FU cascade can be easily calculated. Next, we show how the data observabilities can be used to derive the desired canonical mux networks.

4.1 Computing Data Observabilities

Let the mux network in the Mux-FU cascade have inputs $d_1, d_2, \ldots, d_q$ and outputs $F_1, F_2, \ldots, F_p$. We define the observability of the data inputs as follows.

Definition 1 (Observability Function)
The observability function, $O^k_{d_i}$, for a data input d_i with respect to the output F_k in a multiplexor network is a function such that the output F_k is equal to the input d_i, if and only if $O^k_{d_i}$ is true. In other words, if a stuck at 0/1 fault were present at the input d_i, the minterms of $O^k_{d_i}$ represent the set of all test vectors that would propagate a fault on d_i to the output F_k.

Algorithm 2 uses the following properties to calculate the *exact* observabilities of the data inputs at the outputs of the mux network.

P1. For any node x, the observability of x at the output F_k, $O^k_x = O^k_{b_1} + O^k_{b_2} + \ldots + O^k_{b_n}$, where $b_1, b_2, \ldots, b_n$ are the fanout branches of node x.

P2. If a node x is the fanin of the "0" input of a mux M_i, the observability of the fanout branch from$x$ to M_i is $(O^k_{m_i}.\overline{c_i})$, where c_i is the control input of mux M_i. Similarly, if x is the fanin of the "1" input of M_i, then the observability of the fanout branch is $(O^k_{m_i}.c_i)$.

Algorithm 2 (Data Input Observabilities at Outputs of Mux Network)

1. *For each output F_k, $k = 1, \ldots, p$ do*
   - *(a) Do a depth first search from F_k to extract the subnetwork which has all the nodes in the transitive fanin of F_k. Call it G_k.*
   - *(b) Set $O^k_{m_k} = 1$, where m_k is the output of M_k. For all other nodes "x" in G_k, set $O^k_{\text{x}} = 0$.*
   - *(c) Do a breadth first traversal from M_k to the input nodes in G_k. For every node x visited, let $b_1, b_2, \ldots, b_n$ be the fanout branches of x and $O^k_{b_1}$, $O^k_{b_2}, \ldots, O^k_{b_n}$ be their corresponding observabilities. Then the observability of x at F_k is given by*
   $$O^k_{\text{x}} = O^k_{b_1} + O^k_{b_2} + \ldots + O^k_{b_n}$$
   If b_q is the fanout branch from x to the "0" input of mux M_i, then $O^k_{b_q} = O^k_{m_i}.\overline{c_i}$ else if it is the fanout branch to the "1" input of M_i, then $O^k_{b_q} = O^k_{m_i}.c_i$

Algorithm 2 computes the *exact* observabilities of the data inputs at the outputs of the mux network, with p outputs and $|E|$ edges, in $O(p.|E|)$ time. In contrast, computing the exact observabilities at the logic level, without knowledge of the RTL structure, can be computationally very expensive.

When deriving an expression for the output of a mux network it should be noted that all the data inputs to a network may not have paths to all the outputs of a multiplexor network. However, for simplicity of notation, we consider all the data inputs to all the mux networks when deriving expressions for the output of the kth mux-chain, F_k. If there are inputs which do not have any path to the particular output in question, its observability function is identically zero. In general, the observability functions have a support set which is a subset of the set of control signals to the muxes $c_1, c_2, \ldots, c_n$.

Changing the ordering of the data inputs to the MUX chain changes the order of the *don't care* sets generated for the control signals and hence changes the control signals. This can affect the final delay or area of the circuit but not the testability. Further research needs to be done in ordering algorithms for the muxes to synthesize faster and smaller circuits.

4.2 Transforming into Canonical Form

An acyclic multiplexor network with outputs F_k, inputs $d_1, d_2, \ldots, d_p$ and corresponding data observabilities $O^k_{d_1}$, $O^k_{d_2}, \ldots O^k_{d_p}$ satisfy the following properties.

Property 1 *Given an acyclic multiplexor network with outputs F_k, inputs $d_1, d_2, \ldots, d_p$ and corresponding data observabilities $O^k_{d_1}, O^k_{d_2}, \ldots O^k_{d_p}$ we state the following properties.*

1. $F_k = (d_1.O^k_{d_1}) + (d_2.O^k_{d_2}) + \ldots + (d_q.O^k_{d_q})$
2. $O^k_{d_j}.O^k_{d_i} = 0, i \neq j$
3. $O^k_{d_1} + O^k_{d_2} + \ldots + O^k_{d_p} = 1$
4. $O^k_{d_j}.\overline{O^k_{d_i}} = O^k_{d_j}, i \neq j$
5. $(O^k_{d_p}.\overline{O^k_{d_1}}.\overline{O^k_{d_2}}.\ldots.\overline{O^k_{d_{p-1}}})$
   $= (\overline{O^k_{d_1}}.\overline{O^k_{d_2}}.\ldots.\overline{O^k_{d_{p-2}}}.\overline{O^k_{d_{p-1}}})$

Property 1.1 expresses each output F_k of the mux network in the Mux-FU cascade in terms of the data inputs and their observabilities. Properties 1.1 and 1.2 indicate that for a given set of control input conditions, the output cannot be a function of more than one data input. Property 1.3 states that for any value of the control signals, the output of the multiplexor network has to be equal to one of the data inputs. Properties 1.4 and 1.5 can be derived from Properties 1.2 and 1.3.

The canonical network for any output F_k of the mux network is derived using the following lemma.

Lemma 1 *Each output F_k of the mux network can be re-expressed as*

$$\begin{aligned} F_k^{canonical} &= F_k \\ &= (d_1.O_{d_1}^k) + (d_2.O_{d_2}^k.\overline{O_{d_1}^k}) + \ldots + \\ &\quad (d_{q-1}.O_{d_{q-1}}^k.\overline{O_{d_1}^k}.\overline{O_{d_2}^k}.\ldots.\overline{O_{d_{q-2}}^k}) + \\ &\quad (d_q.\overline{O_{d_1}^k}.\overline{O_{d_2}^k}\ldots\overline{O_{d_{q-2}}^k}.\overline{O_{d_{q-1}}^k}) \end{aligned}$$

Lemma 1 can be proved by applying Properties 1.4 and 1.5 repeatedly.

The transformation to the canonical form is done in the following steps. The data inputs which have a path through the mux network to the output F_k are identified. The observabilities for each of these data inputs with respect to F_k are computed. The canonical chain of muxes is constructed, as shown in Figure 3(b). The "1" input to each mux is one of the data inputs which had a path to the output and the control signal is the corresponding data observability. The transformation has the effect of re-expressing each output F_k in the equivalent canonical form $F_k^{canonical}$. The control signals to the data path are now the observabilities of data inputs. Hence, the DL block of the control logic is changed.

We give an outline of Algorithm 3 which uses Lemma 1 to transform the mux network with outputs F_k into a functionally equivalent canonical mux network with outputs $F_k^{canonical}$. We designate the "0" and "1" inputs of the multiplexor M_i as l_i and r_i respectively , the output as m_i and the control input as c_i.

Algorithm 3 (Deriving Canonical Mux-Chains)

1. *Derive the observability function for each data input d_i at the output of the kth multiplexor network output F_k (this output either goes to a FU or a register). Call it $O_{d_i}^k$. If the data input is a constant, call the jth constant cd_j and its observability function, $O_{cd_j}^k$. Let the number of controllable data path inputs be* p *and the number of constant data path inputs be* q.
2. *For multiplexors M_i, $1 \leq i \leq (p-2)$ do* $l_i = m_{i+1}$; $r_i = d_i$; $c_i = O_{d_i}^k$;
3. $l_{p-1} = d_p$; $r_{p-1} = d_{p-1}$; $c_{p-1} = O_{d_{p-1}}^k$;
4. *For multiplexors M_i, $p \leq i \leq (p+q-2)$ do* $l_i = m_{i+1}$; $r_i = cd_{i+1-p}$; $c_i = O_{cd_{i+1-p}}^k$;
5. $l_{p+q-1} = cd_q$; $r_{p+q-1} = cd_{q-1}$; $c_{p+q-1} = O_{cd_{q-1}}$;
6. *Create a new multiplexor M_0 with $l_0 = m_1$; $r_0 = m_p$;* $c_0 = O_{cd_1}^k + \ldots + O_{cd_q}^k$;

Lemma 2 *The canonical multiplexor network derived by Algorithm 3 has output $F_k^{canonical}$ as in Lemma 1. Hence, from Lemma 1, the canonical mux network is functionally equivalent to the original multiplexor network.*

Figure 3(b) illustrates the canonical mux network generated from the mux network in Figure 3(a). Note the change in the control signals to the data path, and hence the DL part of the control logic. The transformation into canonical mux network prepares the RTL circuit for the final resynthesis phase. In this phase, the constant inputs are pruned and the control logic minimized using *don't cares* derived from the data path, as discussed in the next section.

5 Exploiting Don't Cares Derived from the Data Path to Synthesize Irredundant Circuits

This section exploits the interaction between control logic and data path to obtain an optimized RTL structure which is fully testable at the gate level. We introduce an efficient technique to extract *don't cares* from the data path and use the *don't cares* to optimize the control logic. Resynthesis of the control logic also provides opportunities to further minimize the data path. This process eliminates all the redundancies in the control logic and all the false paths in the data path.

5.1 Minimizing Control Logic Using Data Path Don't Cares

We first address the issue of using *don't cares* derived from the data path to synthesize minimized and 100% testable control logic. In Figure 7(a), we show the control logic (DL) and the canonical multiplexor network (MC) of a typical RTL design. The DL block has been made irredundant by logic synthesis techniques. The MC block is also irredundant. However, cascading the two blocks together gives rise to redundancies in the control logic (as shown by "X" in Figure 7(a)).

Let f_i be the boolean function corresponding to the control signal c_i. In Figure 7(a), note that when c_1 is "1", the value of the control signal c_2 does not affect the output of MC, "z". Consequently, the on-set minterms of function f_1 are observability *don't cares* for function f_2. Similarly, the on-set minterms of the function $(f_1 + f_2)$ are the observability *don't cares* for f_3.

Optimizing control logic DL using the observability *don't cares* derived from the data path produces a minimized control logic as shown in Figure 7(b). An important consequence of the optimization is that the cascaded DL-MC network shown in Figure 7(b) is fully testable.

Knowledge of the RTL structure facilitates easy extraction of the data path *don't cares*. Consider the canonical mux network as derived from the initial data path using Algorithm 1 and Algorithm 3. As mentioned before, the boolean function corresponding to control signal c_i is called

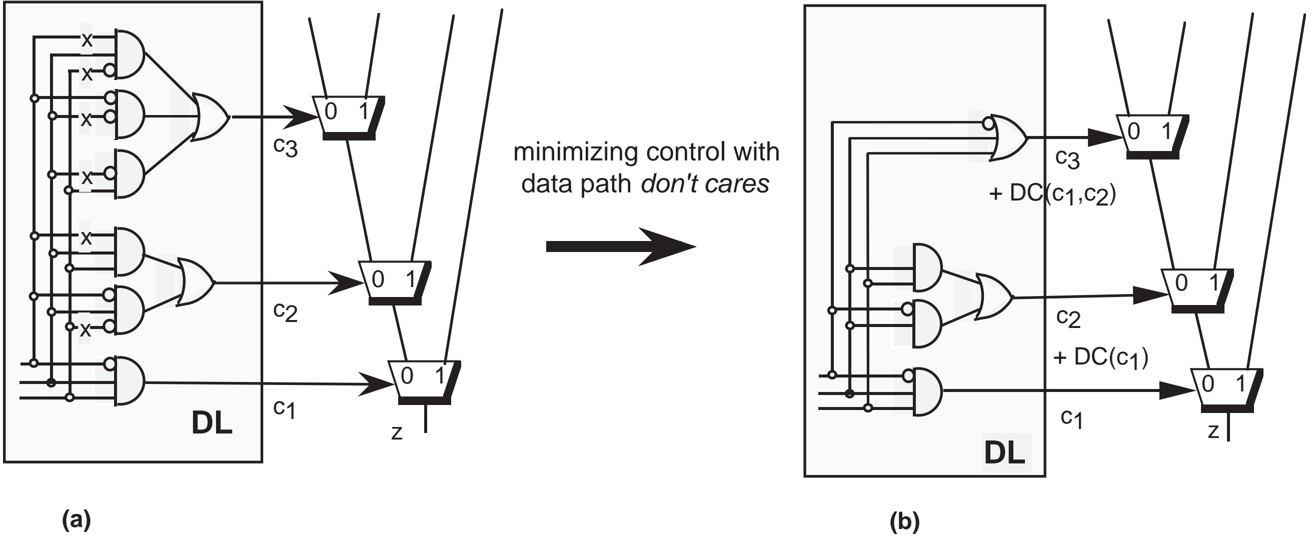

Figure 7: Synthesizing testable control using data path don't cares.

f_i. The observability *don't care* set DC_i for control signal c_i in the canonical mux network is given by:

$$DC_i = \begin{cases} \Phi & \text{if } i = 1 \\ f_1 + f_2 + \ldots + f_{i-1} & \text{if } i > 1 \end{cases}$$

Note that at the logic level, without explicit knowledge of the RTL structure, extracting the *exact* don't cares and optimizing the *specific* set of nodes c_i can be computationally very expensive.

5.2 Minimizing Data Path by Eliminating False Paths

A false path in a data path is a path from a data input to the output of the data path which is never sensitized. In an RTL network, there may be false paths in the data path but no redundancies (An illustrative example is the RTL circuit of Figure 8(a)). Deriving the canonical multiplexor network implicitly eliminates some false paths. Subsequent control minimization using data path *don't cares* produces a constant control signal for each false path that still remains in the data path. Eliminating false paths would lead to further optimization of the data path. Also, the cost of test pattern generation would be reduced.

Instead of explicitly identifying false paths and eliminating them as proposed in [18], we remove the false paths from the canonical network by identifying control signals which are constant. After *don't care* minimization of the control logic, if a control signal c_i is reduced to constant "0", the path from the data input corresponding to the "1" input of M_i is false and the data input can be removed along with mux M_i. If c_i reduces to "1", all the paths from the data inputs which pass through the "0" input of M_i are false and all muxes $M_j, j \geq i$ can be removed. This process ensures an efficient elimination of all false paths in the data path.

We give a simple example how *false paths* in the data path are eliminated by our algorithms. Though the RTL structure shown in Figure 8(a) has no redundant stuck-at-faults, it has two false paths, since control signals c_1 and c_2 are not independent.

The paths are:
$\{data\ d_1$ - $mux\ M_2$ - $mux\ M_1$ - $Op\}$, and
$\{data\ d_2$ - $mux\ M_2$ - $adder$ - $mux\ M_1$ - $Op\}$.

Figure 8(b) and 8(c) shows the effects of Algorithm 1. It creates a Mux-FU cascade by duplicating the adder so that it can be propagated. Figure 8(d) shows the effect of Algorithm 3 on the multiplexor network in Figure 8(c). The algorithm converts the multiplexor network into canonical multiplexor chains, one for each input of the FU. The controlling signals of the muxes are the data observabilities at each FU input. Pruning the constants as in step 3 of Algorithm 4 leads to the RTL circuit in Figure 8(e). At this point *don't cares* from the data path are added to minimize the control signals. We show the derivation of the optimized signals $O^1_{d_2}$ and $O^2_{d_2}$ below. DC() denotes the associated *don't cares* for the control signals.

$$\begin{aligned} O^1_{d_2}\ \text{(optimized)} &= O^1_{d_2} + DC(O^1_{``0"}) \\ &= c_1.c_2 + DC(O^1_{``0"}) \\ &= c_1.c_2 + DC(\overline{c1}) \\ &= c.\overline{c} + DC(\overline{c}) \\ &= 0 + DC(\overline{c}) \\ &= 0 \\ O^2_{d_2}\ \text{(optimized)} &= O^2_{d_2} + DC(O^2_{d_3}) \\ &= \overline{c_1}.c_2 + DC(O^2_{d_3}) \\ &= \overline{c_1}.c_2 + DC(c_1) \\ &= \overline{c}.\overline{c} + DC(c) \\ &= \overline{c} + DC(c) \\ &= \overline{c} + c \\ &= 1 \end{aligned}$$

These control signals being constant, step 5 of Algorithm 4 would remove the muxes M_{21} and M_{22} along with the redundant inputs d_2 of M_{21} and d_1 of M_{22}. This gives Figure 8(f).

It can be seen that the initial false path in Figure 8(a)

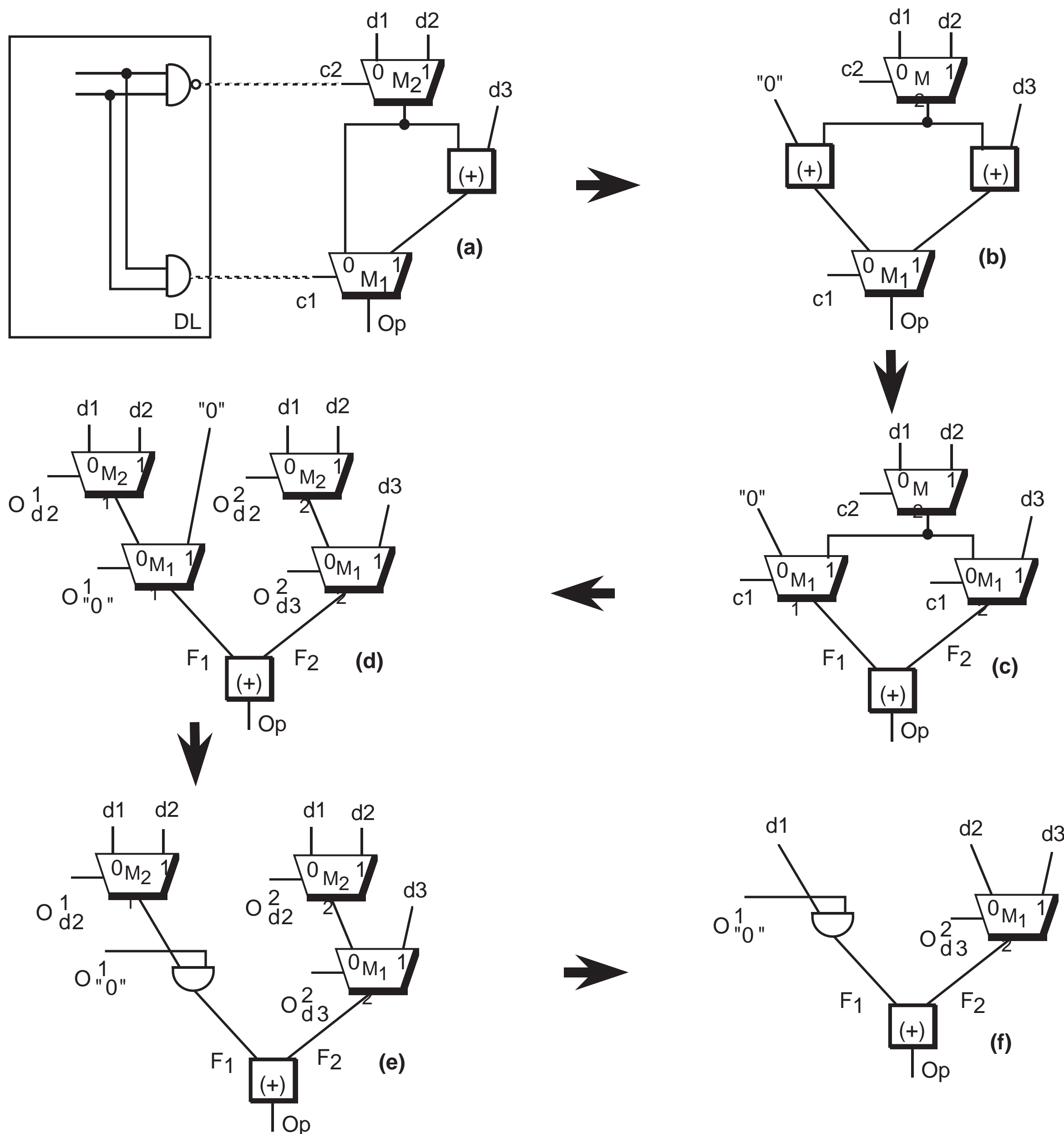

Figure 8: Removing False Paths: An Example.

{*data* d_2 - *mux* M_2 - *adder* - *mux* M_1 - *Op*}
corresponds to the path
{*data* d_2 - *mux* M_{21} - *and_gate* - *adder*} in Figure 8(e). Optimizing DL with the *don't cares* added, implicitly removed this and the other false path.

5.3 Algorithm to Eliminate Redundant Faults and False Paths

We present an algorithm which exploits the interaction between control and data path to synthesize an optimized RTL structure using the two techniques outlined in Section 5.1 and 5.2. The multiplexors in the target architecture are in a canonical mux-chain form cascaded with the FU network as shown in Figure 3(b). The algorithm extracts *don't cares* from the canonical mux-chain to optimize the decoding logic (DL). Subsequently, it also optimizes the data path (DP) by implicitly eliminating the false paths. The EL and CL blocks are optimized independently of the other blocks. It may be observed that *don't cares* from the EL and CL blocks may be used to further minimize the design. This is the subject of ongoing research. The notation used for the mux inputs, control and output remain the same as used in Algorithm 3. The structure of the mux network produced by Algorithm 1 is shown in Figure 9. The Algorithm 4 optimizes the DP and prunes the constant inputs to the muxes to produce the mux network of Figure 10.

Algorithm 4 (Eliminating Redundant Faults and False Paths)

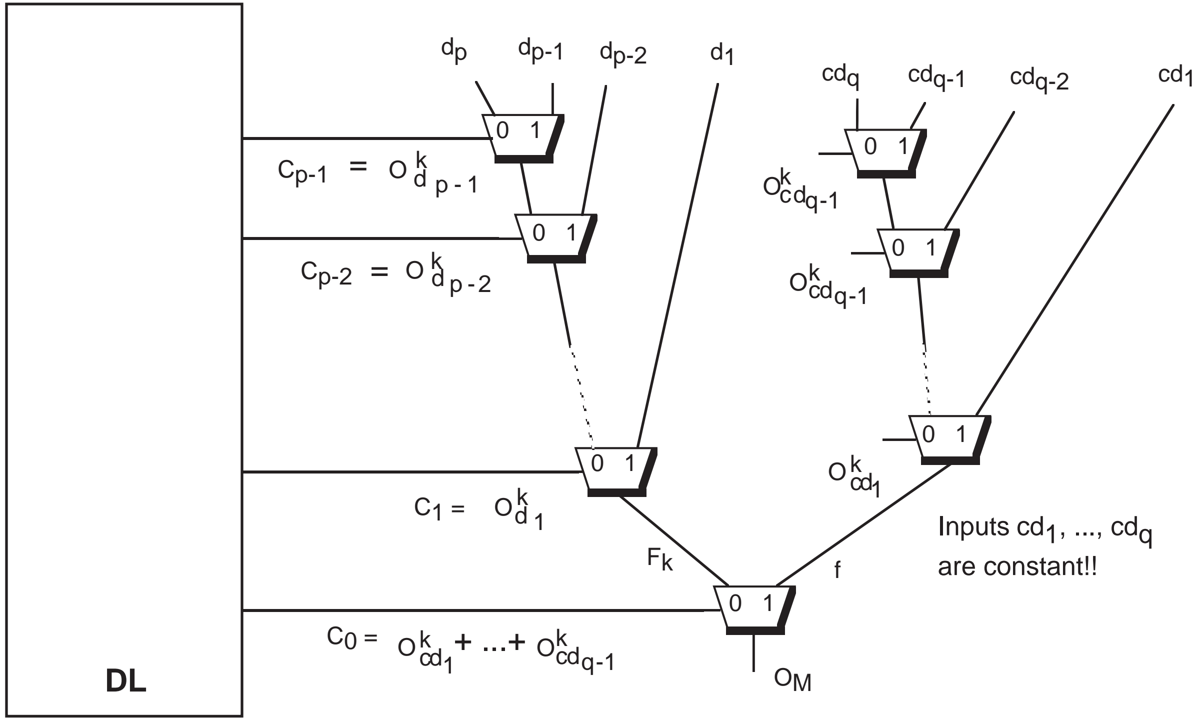

Figure 9: Initial Canonical Multiplexor Chain.

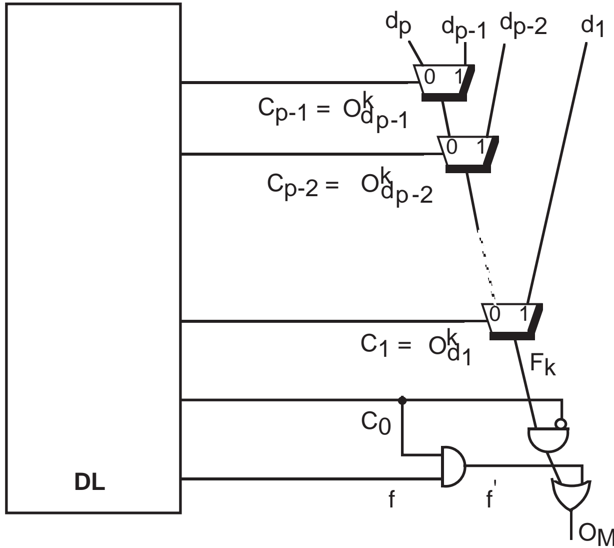

Figure 10: Final Canonical Multiplexor Chain.

1. *Prune constant inputs and redundancies from CL, EL blocks and FUs independently of the other modules.*
2. *Prune all constant inputs to the mux chain of Figure 9. This results in a mux chain as shown in Figure 10. If $f = 0$, the OR gate shown in Figure 10 is removed as well. If f is not zero, then the logic realizing $f' = f.c_0$ is considered part of the DL module when minimizing DL.*
3. *For each c_i, add to f_i its* don't care *set DC_i as in definition 3. Add $f' = f.c_0$ as a* don't care *for c_0. Make DL prime and irredundant w.r.t the* don't cares *added using two level minimization like ESPRESSO [19].*
4. *If f_i reduces to "0", remove M_i. Connect l_i to the node where m_i was connected. If f_i minimized to "1" remove muxes $M_j where j \geq i$. Connect r_i $(= d_i)$ to where m_i was connected.*

The cascade of the DL and MC blocks can be shown to be irredundant. The inputs of an FU are multi-fault testable if any multiple fault on the FU inputs can always propagate to one of the FU outputs. The next theorem shows that if the inputs of the FUs are multi-fault testable, then all faults inside the optimized DL and MC blocks can be propagated to the outputs of the FU network and hence to the inputs of the scannable registers.

Theorem 1 *After applying the RTL transformations, there are no false paths in the data path. If the inputs to the FUs are multi-fault testable and independently controllable and the FUs are irredundant, then the cascaded blocks of DL, MC and FU are 100% testable at gate-level under full-scan.*

Proof See Appendix A. □

The effect of identifying data path don't cares and minimizing the control logic DL of the benchmark `Fancy.b` is shown in Figure 4(a). Note the reduction in the size of DL after the minimization. Minimizing DL using data path *don't cares* identifies control signal O_d^2 as the constant 0. This indicates that the path from data input d is false, allowing for removal of the corresponding mux, and thereby further minimizing the data path. The mux corresponding to the constant signal e is also pruned. The optimized control (DL) and data path of the benchmark `Fancy.b` are shown in Figure 4(a).

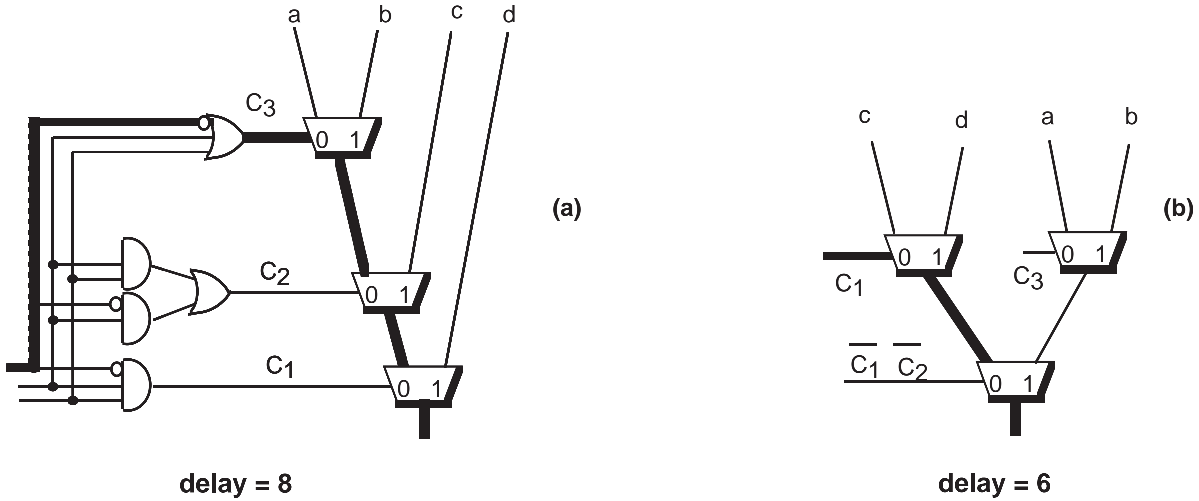

Figure 11: Example Illustrating the Effects of Mux Balancing on Delay.

5.3.1 Need for Test Point Insertion

Instead of minimizing the whole control logic (DL, CL and EL) using the data path *don't cares*, Algorithm 4 is used to minimize the DL block only. This may be necessitated because the EL block typically contains comparators and is not amenable to two level or multilevel minimization. Consequently, there may still be redundant faults when EL and DL are cascaded.

If the complete RTL design is not fully testable after applying our optimization techniques, we insert test points by adding scannable registers at the output of EL which are used only in the test mode. As can be seen from the experimental results in the next section, we are not required to insert test points for most of the designs.

5.3.2 Transformation for Performance: Balancing the Multiplexor Chain

The RTL technique presented in the paper generates multiplexor chains before extracting and using data path *don't cares* for control logic minimization. However, balancing the mux chains may reduce the critical path and improve the performance of the final RTL design. The balanced mux tree corresponding to Figure 7 is shown in Figure 11. Assuming the muxes have a delay of two units, it can be easily seen that the balanced mux tree in Figure 11 has a smaller delay than the corresponding multiplexor chain in Figure 7.

We give a simple recursive algorithm for balancing. Given a multiplexor chain, the algorithm breaks the chain into two equal chains. It balances each chain separately and then introduces a new multiplexor with the two balanced chains as its two inputs. We denote by *mux_chain* an array of muxes of the canonical multiplexor chain, where *mux_chain[i]* refers to the ith mux. Similarly, *mux_tree* is the resultant balanced tree of muxes.

Algorithm 5 balance_mux(mux_chain: IN; num_of_muxes: IN; mux_tree: OUT)

1. *If (num_of_muxes = 1) OR (num_of_muxes = 2)*
   { mux_tree ← mux_chain;
   return; }
2. $m \leftarrow \lceil \frac{num_of_muxes}{2} \rceil$;
3. *mux_chain[m-1].leftinput*
   ← mux_chain[m].rightinput;
4. *balance_mux(mux_chain[m+1], num_of_muxes−m, right_tree);*
5. *balance_mux(mux_chain[1], m−1, left_tree);*
6. *mux_tree.control*
   $\leftarrow \prod_{1 \le i \le m} \overline{(mux_chain[i].control)}$;
7. *mux_tree.rightinput ← right_tree;*
8. *mux_tree.leftinput ← left_tree;*

The mux balancing can be done before optimizing with *don't cares*. Instead of adding *don't cares* to the mux-chain, appropriate *don't cares* can be added to the control signals of the balanced mux tree. As in Theorem 1, we can show that the resultant DL cascaded with the multiplexor tree (as opposed to the multiplexor chain) is irredundant. Transforming a mux chain into a balanced mux tree may change the control logic, introducing a new critical path through the control and data path. Consequently, the performance of the RTL design may actually degrade. We are currently investigating the issue of balancing the mux chains for improved performance.

6 Experimental Results

We have synthesized RTL descriptions from the behavioral specification of several HLSW benchmarks [20]: GCD, BarCode [3], FalsePath [21], Traffic, Fancy.b and QRS [22]. Different scheduling and resource allocation strategies produce distinct RTL specifications which map into specific technology with varying degrees of efficiency in terms of area, speed and testability. Since we want to consider general designs consisting of control as well as data opera-

tions, as opposed to data-intensive designs like DSP applications, we examined two scheduling strategies applicable to control-dominated designs in greater detail: the AFAP schedule [4] and an alternative MINC schedule [5].

The results for the benchmarks are shown in Tables 1, 2, 3, 4, 5 and 7. For GCD, Barcode and FalsePath benchmarks, we applied both scheduling strategies, AFAP and MINC, to derive two distinct initial RT-Level specifications. The columns **AFAP** and **MINC** tabulates results for the RTL specification produced by the AFAP schedule and MINC schedule respectively. The MINC-ndc refers to MINC schedule with no data chaining. Similarly, AFAP-dc and AFAP-ndc refer to AFAP schedule with and without data chaining. Note that the AFAP schedule has been obtained by us and not by using the AFAP scheduler in the IBM system. For each schedule, we report parameters of interest of the initial design. We report the same parameters after applying our RTL optimization tool WONDER. To demonstrate the effectiveness of optimization at the RT-level, we also report results using SIS, a logic level optimization tool from UC-Berkeley [23].

The RT-Level parameters reported are the number of register bits, muxes, FUs and control states. The transformations always reduce the number of muxes significantly, while the number of FUs either reduce or remain same. The number of control states always remain the same, except for the design derived from MINC schedule of FalsePath [5].

To observe the effect of the transformations at the logic-level, we further synthesized each benchmark to a standard cell layout using OASIS [24], with the scalable SCMOS 2.0 micron library supplied with the Logic Synthesis'91 Workshop benchmarks [25]. Parameters to measure the area, like number of combinational gates, literals, register bits, transistor pairs and layout area are reported. To demonstrate that optimization at RT-Level is more effective than optimization at logic level, we optimize the initial designs at the logic level using SIS. The standard script of SIS was used for logic minimization (LM) followed by the *red_removal* command to remove redundancies (RR) to the extent possible.

The experimental results show that in almost all cases the final design produced by RTL optimization has significantly less area and delay than both the initial design and the optimized design produced by SIS. A possible reason for the better performance of WONDER is that transformations involving RTL structures like FUs and muxes are possible at the RT-level, not at the logic level. Logic level tools do not recognize FUs so as to be able to propagate and merge them. However, the effectiveness of our RT-level techniques even when the number of FUs in the initial and final RTL descriptions are same is demonstrated by the results on BarCode, QRS and the AFAP version of the FalsePath benchmark. Though SIS extracts and uses *don't cares* while optimizing the entire design, WONDER produces better results because it utilizes the RT-level structure to extract and use the exact set of data path *don't cares*.

The effect of the RT-Level transformations on the logic level testability of the designs is shown in the rows *stuck-at-faults*, *untested faults*, and the *fault coverage*. The number of stuck-at-faults which are not testable are reported as *untested faults*. These include faults proven redundant or aborted after the nominal default backtrack limit of 100 backtracks, using PODEM-based test generation algorithm in OASIS. *Fault Coverage* gives the percentage of total faults which are testable in the scan mode. The results show that while none of the initially scheduled designs are fully testable, *all* final designs except for QRS produced by WONDER achieved 100% fault coverage without degradation in area or critical path delay. Increasing the test generation backtrack limit for initial designs only increased the CPU costs without significantly improving the fault coverage. Also, while RTL optimization produces circuits which are 100% testable, circuits produced by SIS are not.

It is also demonstrated in Table 6 that as the bitwidth of the data increases, the CPU time required for logic optimization increases substantially while the time for RT-Level optimization remains independent of the data bit width. In fact, for the 16 bit example, the standard script of SIS did not complete in 3 days. We used a modified script [1] to complete the optimization process. Table 6 also demonstrates that as the circuit size increases, the gap between optimization achieved at the RT-Level and logic level widens.

The experimental results further demonstrate that while a scheduling strategy does affect the initial standard cell layout area, critical path delay and logic level testability, the proposed RT-Level transformations reduce this variance significantly while consistently achieving 100% fault coverage at the logic level, regardless of the initial scheduling algorithm used.

6.1 QRS: A Chip for Biomedical Applications

We briefly discuss the RTL optimization of the QRS chip [26], which is significantly larger than the other benchmarks reported [22]. QRS is a real life design for a specific biomedical application, with a large number of control as well as arithmetic operations. The initial RTL description satisfies the constraint of three adders and two counters. The statistics of the initial description are given in Table 7 under column Init. It is comparable in size (transistors) to the circuit synthesized by CALLAS from VHDL [22].

The initial circuit cannot be optimized at the logic-level using the standard script of SIS due to prohibitive memory/CPU requirements. The optimization achieved using a modified script of SIS, followed by redundancy removal, are reported in column SIS. Note that the optimized circuit has many untestable faults. Next, we applied the RT-level optimization technique to the initial circuit.

Propagating the FUs in QRS leads to a substantial increase in the number of FUs, a possible outcome of Algo-

[1] reducing argument thresh in the eliminate command from 5 to 3

Parameter	AFAP Init	AFAP SIS (LM+RR)	AFAP WONDER	MINC Init	MINC SIS (LM+RR)	MINC WONDER
register bits	49	-	53	49	-	57
muxes	464	-	116	198	-	120
func units	1	-	1	1	-	1
control states	7	-	7	4	-	4
comb. gate	831	374	369	529	370	364
literals	2511	915	1007	1434	961	984
transistors	5054	1838	2014	2868	1924	1968
layout[mm^2]	7.29	2.64	2.80	4.40	2.72	2.71
delay[ns]	111.30	70.90	87.50	89.80	71.50	89.90
stuck-at-faults	3958	1429	1687	2195	1476	1666
untested faults	52	4	0	12	10	0
fault coverage	98.69%	99.70%	100%	99.45%	99.32%	100%

Table 1: GCD (16 bit case) - results of logic-level and RT-level transformations

Parameter	AFAP Init	AFAP SIS (LM+RR)	AFAP WONDER	MINC Init	MINC SIS (LM+RR)	MINC WONDER
register bits	54	-	54	54	-	54
muxes	306	-	75	170	-	94
func units	3	-	3	3	-	3
control states	8	-	8	7	-	7
comb. gate	487	244	242	325	269	267
literals	1377	585	579	868	622	626
transistors	2754	1184	1168	1802	1252	1288
layout[mm^2]	2.98	1.28	1.23	1.95	1.48	1.39
delay[ns]	33.60	39.40	25.70	39.40	39.50	31.60
stuck-at-faults	2302	919	1055	1643	1053	1107
untested faults	100	1	0	0	7	0
fault coverage	95.66%	99.89%	100%	100%	99.33%	100%

Table 2: BarCode - results of logic-level and RT-level transformations

Parameter	AFAP Init	AFAP SIS (LM+RR)	AFAP WONDER	MINC Init	MINC SIS (LM+RR)	MINC WONDER
register bits	82	-	82	80	-	82
muxes	263	-	82	84	-	82
func units	1	-	1	4	-	1
control states	2	-	2	1	-	2
comb. gate	508	287	227	450	401	228
literals	1332	681	522	1012	968	522
transistors	2698	1360	1080	2084	1982	1076
layout[mm^2]	3.36	1.63	1.09	2.25	1.76	1.04
delay[ns]	158.10	180.10	134.80	117.60	123.60	124.40
stuck-at-faults	2356	1107	1038	1930	1567	1050
untested faults	198	47	0	43	4	0
fault coverage	91.60%	95.75%	100%	97.77%	99.74%	100%

Table 3: FalsePath (16 bit case) - results of logic-level and RT-level transformations

Parameter	MINC Init	SIS (LM+RR)	WONDER
register bits	9	-	9
muxes	81	-	12
func units	4	-	1
control states	9	-	9
comb. gate	177	96	75
literals	464	228	179
transistors	928	458	362
layout[mm^2]	.716	.351	.251
delay[ns]	36.10	27.90	19.50
stuck-at-faults	869	367	326
untested faults	49	1	0
fault coverage	94.36%	99.72%	100%

Table 4: Traffic - results of logic-level and RT-level transformations

Parameter	MINC Init	SIS (LM+RR)	WONDER
register bits	97	-	93
muxes	258	-	141
func units	3	-	1
control states	2	-	2
comb. gate	1150	1091	776
literals	2735	2750	1874
transistors	5572	5592	3778
layout[mm^2]	9.01	9.45	6.25
delay[ns]	184.30	183.80	120.10
stuck-at-faults	4064	4051	2944
untested faults	53	341	0
fault coverage	98.70%	91.58%	100%

Table 5: Fancy.b (16 bit case) - results of logic-level and RT-level transformations

Parameter	4bit WONDER	4bit SIS (LM+RR)	8bit WONDER	8bit SIS (LM+RR)	16bit WONDER	16bit SIS (LM+RR)
comb. gate	230	268	415	539	776	1091
literals	539	668	992	1347	1874	2750
transistors	1804	1350	2000	2732	3778	5592
layout[mm^2]	1.11	1.37	2.45	3.43	6.25	9.45
delay[ns]	37.00	78.10	64.20	98.80	120.10	183.80
stuck-at-faults	924	1023	1595	1980	2944	4050
untested faults	0	50	0	75	0	341
fault coverage	100%	95.11%	100%	96.21%	100%	91.58%
CPU time	5.9s	33.2s	5.9s	504.5s	5.9s	10631.4s

Table 6: Fancy.b - effect of increase in data bit width on the optimization time by WONDER and SIS

Parameter	MINC Init	SIS (LM+RR)	WONDER	WONDER+
register bits	455	-	475	571
muxes	2277	-	1294	1390
func units	3	-	3	3
control states	19	-	19	19
comb. gate	4559	3824	3667	3807
literals	12245	9388	9306	9592
transistors	24756	18616	18854	19366
delay[ns]	341.10	361.80	347.90	345.50
stuck-at-faults	18779	14117	14603	15281
untested faults	417	102	12	0
fault coverage	97.78%	99.28%	99.97%	100%

Table 7: QRS (16 bit case) - results of logic-level and RT-level transformations

rithm 1 anticipated in Section 3. Hence, we did not transform the initial design to obtain a cascade of MUX-FU network, as required by our algorithms. Instead we identified mux subnetworks which had no node with multiple fanouts. Subsequently, the mux subnetworks were used for the derivation of canonical mux chains and don't care minimization, performed by Algorithms 3 and 4 respectively.

The parameters of the resultant circuit are reported in the column WONDER in Table 7. While the number of functional units and control states remain the same as in the initial design, the number of muxes reduce significantly. The marginal increase in the number of register bits is due to the extra test points that were added to the outputs of the EL logic to improve testability. The logic-level parameters reveal a significant level of optimization achieved by our technique. For instance, the number of literals have been reduced from 12245 to 9306, a reduction of 24%. The delay of the circuit remains almost the same. The testability of the circuit also improved significantly. While the initial circuit had 417 *untested faults*, the final circuit has only 12 *untested faults*.

To make QRS 100% testable, we added extra test points at the outputs of the three adders. This effectively broke the long data chains and made most of the aborted faults testable. The inputs to the logic blocks were made controllable in the full scan mode by propagating the data path registers to their fanouts. The resultant circuit is 100% testable, and its parameters are reported in column WONDER+ of Table 7.

7 Conclusions

We successfully addressed the problem of transforming a given RTL specification into a functionally equivalent, minimized and 100% testable design. We maintained the RT-Level design hierarchy while performing RT-Level transformations of initially specified data-path, followed by resynthesis of control logic using data path *don't cares*. The transformations use information about the RT-Level structure, making several computations like observability don't cares very simple. Notably, upon resynthesis of the control, we implicitly removed *all* redundancies in the control as well as *all* false paths in the data path itself. Experiments with several RTL designs demonstrate that the transformations significantly reduce layout area and delay, while consistently achieving 100% fault coverage at the gate level. The experimental results also demonstrate that optimizing designs at the RT-level is more effective than the traditional optimization techniques performed at the logic level.

The transformations and resynthesis process described in this paper were aimed towards deriving a fully testable design. Some of the transformations, like propagating functional units and balancing multiplexor chains, may adversely affect the area and delay of the final design. We are currently investigating various RT-Level transformations for optimizing area and performance while preserving testability.

Acknowledgements

Subhrajit Bhattacharya was supported by a benchmark grant from ACM/SIGDA and a grant from C&C Research Laboratories, NEC USA. Michael Pilsl (Siemens Central Research Laboratory, Munich) generously assisted in testing VHDL descriptions of the benchmarks which we use to evaluate the experimental results. Kris Kozminski (MCNC) and Clay Gloster (NCSU/MCNC) provided valuable assistance in running the initial experiments using the OASIS system.

References

[1] R. Camposano and W. Wolf. *High-Level VLSI Synthesis*. Kluwer Academic Publishers, Norwell, Massachusetts, 1991.

[2] D. D. Gajski, N. D. Dutt, A. C-H Wu, and S. Y-L Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Norwell, Massachusetts, 1992.

[3] P. Michel, U. Lauther, and P. Duzy. *The Synthesis Approach to Digital System Design*. Kluwer Academic Publishers, Norwell, Massachusetts, 1992.

[4] R. Camposano and R. A. Bergamaschi. Synthesis using Path-Based Scheduling: Algorithms and Exercises. In *Proceedings of the 27th Design Automation Conference*, June 1990.

[5] S. Bhattacharya and F. Brglez. Synthesis for Testability from Behavioral Specifications: Benchmarking the Scheduling Strategies. MCNC Technical Report, December 1992.

[6] J. Rajski and J. Vasudevamurthy. The Testability-Preserving Decomposition and Factorization of Boolean Expressions. *IEEE Transactions on Computer-Aided Design*, June 1992.

[7] S. Devadas and K. Keutzer. A Unified Approach to the Synthesis of Fully Testable Sequential Machines. *IEEE Transactions on Computer-Aided Design*, Jan 1991.

[8] S. S. K. Chiu and C. Papachristou. A Built-In Self-Testing Approach For Minimizing Hardware Overhead. In *Proceedings of the International Conference on Computer Design*, 1991.

[9] T. C. Lee, W. Wolf, N. K. Jha, and J. M. Acken. Behavioral Synthesis for Easy Testability in Data Path Allocation. In *Proceedings of the International Conference on Computer Design*, 1992.

[10] L. Avra. Allocation and Assignment in High-Level Synthesis for Self-Testable Data Paths. In *Proceedings of the International Test Conference*, 1991.

[11] A. Mujumdar, K. Saluja, and R. Jain. Incorporating Testability Considerations in High-Level Synthesis. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, 1992.

[12] F. Brglez, D. Bryan, J. Calhoun, G. Kedem, and

R. Lisanke. Automated Synthesis for Testability. *IEEE Transactions on Industrial Electronics*, 36(2):263–277, May 1989.

[13] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A Multiple-Level Logic Optimization System. *IEEE Transactions on Computer-Aided Design*, cad-6(6):1062 – 1081, November 1987.

[14] H. Savoj and R. K. Brayton. Observability Relations and Observability Don't Cares. In *IEEE International Conference on Computer-Aided Design*, Nov. 1991.

[15] S. Dey, F. Brglez, and G. Kedem. Circuit Partitioning for Logic Synthesis. *IEEE Journal of Solid-State Circuits*, 26(3):350 – 363, March 1991.

[16] K. De and P. Banerjee. Logic Partitioning and Resynthesis for Testability. In *IEEE International Test Conference*, Oct. 1991.

[17] N. Wehn, J. Biesenack, and M. Pilsl. A New Approach to Multiplexer Minimization in the CALLAS Synthesis Environment. In VLSI 91, A. Halaas and P.B. Denyer, editors, Elsevier Science Publishers B.V. (North-Holland), August 1991.

[18] A. Saldanha, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Circuit Structure Relations to Redundancy and Delay: The KMS Algorithm Revisited. In *29th ACM/IEEE Design Automation Conference*, June 1992.

[19] R.L. Rudell. *Logic Synthesis for VLSI Design.* PhD thesis, ECE Dept. University of California at Berkeley, April 1989.

[20] 1992 High-Level Synthesis Workshop. Benchmarks available in the HLSW92 directory via ftp: send mail to benchmarks@mcnc.org for details.

[21] R. A. Bergamaschi. The Effects of False Paths in High-Level Synthesis. In *IEEE International Conference on Computer-Aided Design*, Nov 1991.

[22] M. Pilsl, S. Bhattacharya, and F. Brglez. Synthesizing Behavioural Benchmarks in VHDL into Standard Cell Layout. In *Sixth International Workshop on High Level Synthesis* , November 1992.

[23] E.M. Sentovich, K.J. Singh, C. Moon, H. Savoj, R.K. Brayton, and A. Sangiovanni-Vincentelli. Sequential Circuit Design using Synthesis and Optimization. In *Proceedings of the International Conference on Computer Design*, October 1992.

[24] K. Kozminski (ed.). *OASIS Users Guide.* MCNC, Research Triangle Park, N.C. 27709, 1991.

[25] Saeyang Yang. Logic Synthesis and Optimization Benchmarks, User Guide Version 3.0. In *International Workshop on Logic Synthesis*, MCNC, Research Triangle Park, NC, May 1991.

[26] M.G. McNamer S.C. Roy, H.T. Nagle and W.T. Krakow. QRSBIST: A Reliable Cardiac Arrhythmia Monitor ASIC. In *Proceedings 3rd Annual IEEE ASIC Seminar and Exhibit*, September 1990.

Subhrajit Bhattacharya graduated with a B.Tech. degree in Computer Science and Engineering from Indian Institute of Technology, Kharagpur, in May 1989. He joined the PhD programme at Duke University, North Carolina, in the fa ll of 1989 and is currently continuing his PhD there.

His present interests are in the area of testability and optimization for performance in high-level and register-level synthesis. He is also interested in scheduling and its relationship to testability and performance.

He has previously worked on visualization in parallel coordinates for the linear programming problem and continues to be interested in randomized and parallel algorithms and numerical computation.

Franc Brglez (S'68-M'70) graduated with a Dipl.-Ing. degree in electrical engineering from the University of Ljubljana, Slovenia, and received a Ph.D. degree in electrical engineering from the University of Colorado, Boulder.

He held several positions with BNR and Northern Telecom in Ottawa, Canada, and joined MCNC in Research Triangle Park, N.C., in 1986 as a resident professional and later-on as the director of Design Synthesis Research. He is also an adjunct Professor at the Department of Electrical and Computer Engineering at NCSU (North Carolina State University) in Raleigh and at the Department of Computer Science, Duke University in Duhram.

At MCNC, he organized a team to develop, in collaboration with Participating Universities, a digital design system (OASIS) with emphasis on compiling testable standard cell IC layouts, applying advanced synthesis and test generation techniques. While his current research is in the area of automated design of digital logic, with a major focus on partitioning, synthesis, test generation and verification, his earlier publications span the areas of CAD and optimization of analog/digital filters and equalizers, large-scale nodal analysis of multi-phase periodically switched-capacitor circuits, and digital signal processing applications to testing of mixed analog/digital communication circuits.

In 1985, he co-authored and distributed the widely accepted ISCAS'85 benchmark set. Since 1986, he has been organizing a yearly series of international workshops at MCNC that alternate between layout and logic synthesis, introducing new generations of benchmarks and their distribution, with the support from ACM/SIGDA.

Sujit Dey (S'90-M'92) received the B. Tech. degree in Computer Science and Engineering from the Indian Institute of Technology, Kharagpur, in 1985, the M. S. degree in Computer Science from Southern Illinois University, Carbondale, in 1987, and the Ph. D. degree in Computer Science from Duke University in 1991.

In 1991, he joined the C&C Research Laboratories, NEC USA, where he is presently a member of the Research Staff. His research interests include several aspects of computer-aided design of integrated circuits and systems, parallel algorithms and architectures, and distributed systems. His current research focuses on partitioning and synthesis for performance and testability, at several levels of abstraction, including the system, register-transfer and logic levels.

8 Appendix A

Theorem 1 *After applying the RTL transformations, there are no false paths in the data path. If the inputs to the FUs are multi-fault testable and independently controllable and the FUs are irredundant, then the cascaded blocks of DL, MC and FU are 100% testable at gate-level under full-scan.*

Proof Follows from Lemma 3 and 4. □

For Lemma 3 and Lemma 4, refer to Figure 9 and Figure 10 for notation.

Lemma 3 *After applying Algorithm 4, there are no false paths from the data inputs in the data path through the multiplexor chains.*

Proof If there is any false path through the multiplexor chain from the data input $d_i, i \neq p$, we show that c_i is stuck-at-0 and hence is removed by step 5 of Algorithm 4. There can be two possibilities. It is possible that c_i is stuck-at-0 even before adding *don't cares* in which case it would be removed in step 5 of Algorithm 4. Otherwise it would mean that whenever f_i was true, some f_j, $j < i$ was true. However, $f_i = O^k_{d_i}$ and $f_j = O^k_{d_j}$. But from Property 1.2, we see if $i \neq j$, f_i and f_j cannot be "1" at the same time. Hence we have a contradiction.

If the false path was from data input d_p, it can be shown that either c_{p-1} would be stuck-at-1, or some c_j, $j < (p-1)$, would be stuck-at-1. In both cases, d_p and the corresponding mux would be removed in step 5 of Algorithm 4. We discuss the proof when none of the c_j, $j < (p-1)$ reduces to "1" after step 4 of Algorithm 4. In this case, whenever f_{p-1} is "0", some $f_j, j < (p-1)$ is "1" because the path from d_p to the output of the mux chain is false. Hence, $\overline{f_{p-1}}$ is a *don't care* for f_{p-1}. Thus after *don't care* minimization in step 4, f_{p-1} reduces to "1" i.e. c_{p-1} is stuck-at-1. □

Lemma 4 *After applying Algorithm 4, there are no redundancies in the combined decode logic (DL) and multiplexor chain (MC) if they are isolated from the other modules. Moreover, if the inputs to the FUs are multi-fault testable and independently controllable and the FUs are irredundant, then the cascaded blocks of DL, MC and FU are irredundant.*

Proof (1) We first show that the combined DL and MC block is irredundant. In part (a), we show that the DL block has no redundant fault. We next show in part (b) that the MC block has no redundant fault. We show that if there is a redundant fault in DL when cascaded with the mux chain (MC), then DL had not been made prime and irredundant with respect to the *don't cares* as claimed in step 4 of Algorithm 4.

(a) Suppose there is some redundant fault inside DL after connecting it back to the MC. Let the fault be at node X. *We discuss the case X stuck-at-1 first.* The argument for X stuck-at-0 follows similarly. Let c_i be the control signal with the smallest index in the transitive fanout of the node X. We assume for the moment that neither f_0 or $f^{'}$ is in the transitive fanout of X, where f_0 is the boolean function corresponding to c_0 in Figure 10. The fault $\overline{D}$ at X can propagate to c_i because it has been claimed in step 4 of Algorithm 4 that DL is irredundant. However, $\overline{D}$ cannot propagate to the output F_k of the multiplexor chain. This implies that each minterm that sets X to "0" and propagates $\overline{D}$ to c_i, also sets some f_j, $j < i$ to "1". Hence these minterms are in the onset of some control signal c_j, $j < i$. Consequently from section 5.1 it can be seen that these minterms are *don't cares* for c_i and hence for the corresponding boolean function f_i. Let f_X be the function corresponding to node X. The function f_i can be written as

$f_i = f_X . f_i|_{f_X=1} + \overline{f_X} . f_i|_{f_X=0}$

and the corresponding observability function for X at c_i as

$(f_i|_{f_X=1} . \overline{f_i|_{f_X=0}} + \overline{f_i|_{f_X=1}} . f_i|_{f_X=0})$

The minterms that set X to "0" and also propagate it to c_i are given by

$(f_i|_{f_X=1} . \overline{f_i|_{f_X=0}} + \overline{f_i|_{f_X=1}} . f_i|_{f_X=0}) . \overline{f_X}$

We have found that these minterms are *don't cares* for f_i. We draw the Karnaugh-map of f_i along with its *don't care* set in Figure 12.

From the Karnaugh map, we see that $f_i|_{f_X=1}$ is a minimal cover for f_i with the *don't cares* added. If X were to be present in DL after step 4 of Algorithm 4, then the cover derived for c_i and hence for DL was not prime and irredundant. This is a contradiction. Hence, the assumption that there was a fault X which was redundant is false.

$f = x f_X + \overline{x}\, \overline{f_{\overline{X}}}$

$DC(f) = (f_X \overline{f_{\overline{X}}} + \overline{f_X} f_{\overline{X}}).\overline{x}$

cover of $(f + DC(f)) = f_X$

x \ $f_X f_{\overline{X}}$	00	01	10	11
0		1/DC	1	DC
1			1	1

Figure 12: Karnaugh Map Used to Prove Lemma 4.

Carrying the proof for faults in the cone of f_0 and $f^{'}$ is similar and is hence not discussed.

(b) To prove there are no redundant faults inside the multiplexor chain (MC), we refer to Figure 13 which shows a typical multiplexor in the multiplexor chain Figure 10 and produced by Algorithm 4. The nodes I, c_j and d_j are independent and controllable where I is the output m_{j+1} of multiplexor M_{j+1} or is a data input. It is not possible that f_j is a constant, because step 5 of Algorithm 4 removes constant outputs of the DL block. If node α_1 is redundant, it can be shown that f_j will be reduced to a constant after step 4 of Algorithm 4. Let us assume that α_1 is stuck-at-1.

Hence all the minterms that propagate a "0" to c_j make some $f_i, i < j$ "1". Hence $\overline{f_j}$ is a *don't care* for f_j i.e. f_j can be reduced to "1". The corresponding c_j must have been removed in step 5 of Algorithm 4. Hence redundant nodes like α_1 could not exist after application of Algorithm 4. Similarly it can be shown that the faults $\alpha_2, \alpha_3, \alpha_4$ and α_5 are not redundant.

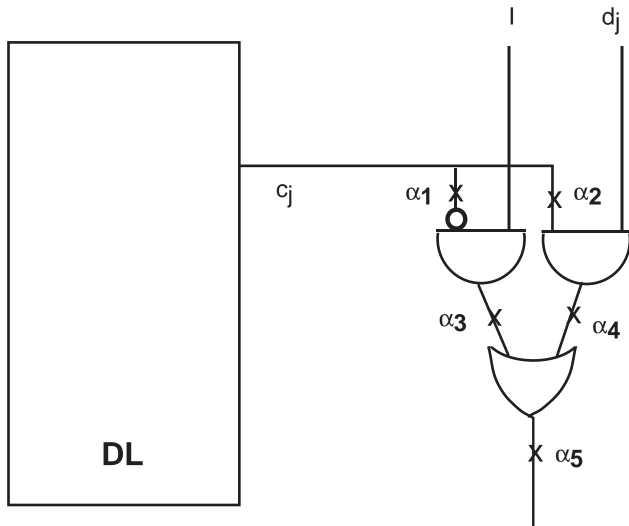

Figure 13: Faults Inside a Multiplexor.

(2) From (1), all faults inside the DL and MC blocks can propagate to the outputs of the MC block, and hence, to the inputs of the FUs. Since the inputs of the FUs are multi fault testable, any fault on the inputs of the FUs can always propagate to the outputs of the FU network. Consequently, all faults in DL and MC blocks are testable in the DL, MC, FU cascade. Also, because the FUs are irredundant and the inputs of the FUs are independently controllable, all faults in the FUs are testable in the DL, MC, FU cascade.

9 Appendix B

```
PACKAGE types IS
   SUBTYPE nat8 is integer RANGE 255 DOWNTO 0;
END types;

LIBRARY bench;
USE bench.callas.ALL, work.types.all;

ENTITY fancy_b IS
PORT(reset : IN bit; -- Global reset
      clk : IN bit; -- Global clock
      start : IN boolean;
      ainp : IN nat8;
      binp : IN nat8;
      cinp : IN nat8;
      eoc : OUT boolean;
      f : OUT nat8);
END fancy_b;

ARCHITECTURE algorithm OF fancy_b IS
BEGIN
   fancy_b: PROCESS
      VARIABLE temp1a, temp1b : nat8;
      VARIABLE temp3, temp4, temp6a : nat8;
      VARIABLE a, b, c, counter : nat8;
   BEGIN

      eoc <= false;
      f <= 0;

-- CYCL statements are for simulation purposes.
-- They maybe ignored otherwise.

      CYCL(reset,clk,1);IF (reset = '1') THEN
      ELSE outest_loop:LOOP

      a := ainp;
      b := binp;
      c := cinp;
      temp1a := 0;
      counter := 0;
      eoc <= false;
      WHILE (counter < b) LOOP
         IF (a <= counter) THEN
            IF (a <= counter) THEN
               temp6a := b;
            ELSE
               temp6a := temp1a;
            END IF;
         ELSE
            temp6a := a;
         END IF;

         IF (temp1a = 0) XOR start THEN
            temp1b := c;
         ELSE
            IF (a > b) THEN
               temp1b := a;
            ELSE
               temp1b := b;
            END IF;
         END IF;

         IF start THEN
            temp4 := temp1b;
         ELSE
            temp4 := temp6a;
         END IF;

         IF (start XOR (a > b)) THEN
            IF (b > c) THEN
               temp3 := c;
            ELSE
               temp3 := b;
            END IF;
         ELSE
            temp3 := temp1b;
         END IF;

         IF (temp1a = 0) THEN
            temp1a := temp4 + temp3;
         ELSE
            IF ((a > b) XOR (b > c)) THEN
               temp1a := temp4 + temp1a;
            ELSE
               temp1a := temp4 + a;
            END IF;
         END IF;
         counter := counter + 1;

         CYCL(reset,clk,1);EXIT outest_loop WHEN (reset = '1');
      END LOOP;

      f <= temp1a;
      eoc <= true;

      CYCL(reset,clk,1);EXIT outest_loop WHEN (reset = '1');
      END LOOP outest_loop;END IF;

   END PROCESS;
END algorithm;
```