Easily Testable Data Path Allocation Using Input/Output Registers

Li-Ren Huang, *Jing-Yang Jou, Sy-Yen Kuo, and Wen-Bin Liao

Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan, R.O.C.

Abstract

Most existing behavioral synthesis systems concentrate on area and performance optimization, while ignoring other design qualities such as testability. In this paper‡, we present three algorithms for register, module, and interconnection allocation of behavioral synthesis respectively to improve testability in data path allocation without assuming any specific test strategy. By using primary input/output registers effectively, the proposed algorithms produce RTL designs with better testability, while incur low or even no hardware overhead. Four benchmarks are synthesized using the proposed approaches and the results are compared with the best results of similar works in the literature. It shows that our approaches give both higher fault coverage and lower hardware overhead.

1 Introduction

Behavioral synthesis is a translation and optimization process to generate register-transfer-level(RTL) data path circuits and its controllers from a given abstract behavioral description of the system to be synthesized. Most of the researches on behavioral synthesis focus on area or performance optimization [1-5], while ignoring that testing is also an important issue during the VLSI design and fabrication process. Lack of testability consideration in the VLSI synthesis process would easily produce totally untestable circuits. As a result, design-for-testability(DFT) circuits are usually added after the synthesis to make the synthesized circuits testable. However, adding the DFT circuits late in the design process would impose considerable hardware overheads and performance penalties. In the worst case, neither area nor delay constraints would be satisfied.

Behavioral synthesis for testability can be roughly classified into two strategies. The first one assumes that *some DFT structures*, such as scan design, BIST, or test point insertion will be added on the synthesized circuits. Mujumdar [7] proposed two unit-binding methods to improve the testability of the synthesized circuit by reducing the number of self loops. Avra [8] and Papachristou [9] described several cost-guided allocation algorithms assuming that the BIST strategy is employed. Gebotys [6] proposed a VLSI synthesis system in which testability, silicon area and delay are used as the VLSI design criteria. However, the design quality measure of this system is pretty crude.

The other strategy assumes that there is *no test strategy* at all. In other words, no DFT structures are to be added on the synthesized structures. Chen [12] proposed a testability enhancement method by modifying the input HDL descriptions rather than modifying the circuit structures. Lee [10, 11] presented a graph-based analysis algorithm for data path allocation to produce testable data paths. They used heuristics to increase the controllability and observability of registers and reduce sequential depth between registers.

In this paper, we focus on the *data path allocation* task in the behavioral synthesis. Three algorithms which perform register, module, and interconnection allocation respectively for testability are proposed. The testability of the control part is not addressed in this paper. The goal of the proposed data path allocation algorithms is to realize easily testable data paths, while keeping hardware overheads as low as possible. No specific test strategy is assumed in these algorithms.

Four benchmarks are synthesized using the proposed approaches and the results are compared with the best synthesized results in the literature. The final design quality is measured in terms of the fault coverage of the data path, the CPU time taken by ATPG, and the number of test patterns. The measurements are obtained with a commercial ATPG tool under fixed conditions.

2 Register Allocation

Register allocation assigns the variables whose lifetime span across the boundary of control steps to registers. One register can hold the values of several vari-

‡This research was supported by the National Science Council, Taiwan, R.O.C., under Grants NSC84-2215-E-002-032

*Jing-Yang Jou is with the Department of Electronic Engineering, National Chiao Tung University, Hsin-Chu, Taiwan, R.O.C.

1081-7735/96 $5.00 © 1996 IEEE
Proceedings of ATS '96

ables as long as their lifetimes do not conflict. Furthermore, the registers can be merged into a multiport register file where the number of accessed registers does not exceed the number of access ports.

2.1 Methodology

In our circuit model, external signals can only be transmitted into the circuit through the input variables. In other words, input lines corresponding to the input variables are independently controllable nodes in the circuit. Similarly, internal signals can only be transmitted out of the circuit through the output variables, and output lines corresponding to the output variable are independently observable nodes. The input and output variables have their own corresponding primary input and primary output registers to store their values.

The main objective of the synthesis for testability system for sequential machines is to improve the controllability and observability of registers. Thus, we must assign as many intermediate variables as possible to the primary input registers, which could be controlled by the primary input lines through the input multiplexer. Also we must assign as many intermediate variables as possible to the primary output registers, which could be observed by the primary output lines. That is, the lines associated with the variables allocated to primary input(primary output) registers are all easily controllable(observable).

In information theory, the entropy(randomness) of a signal decreases as the signal transmits through a channel(circuit) [15]. The amount of entropy reduction depends on the characteristics of the channel. This means that the controllability of a node depends on the structure of the circuit. Therefore, if we assign too many variables to the primary input registers, while keeping the same size of primary input/output lines passing through the multiplexer, the controllability at the output of the multiplexer will decrease. The same is true for the observability measure of primary outputs.

Therefore, we should distribute non-primary input or non-primary output variables more evenly to those primary input or primary output registers in order to avoid assigning too many intermediate variables to one primary input or one primary output register. Thus, we have the first register allocation rule as following :

Rule 1 : Assign appropriate number of variables to each primary input or primary output register.

When the lifetime span of one register does not overlap with that of a second register, the two registers can be merged into one register to reduce the hardware cost. But the testability of the resultant circuit could decrease. In order to make a path easily controllable, special values must be assigned to some registers in the path. Similarly, to make a path easily observable, special values must also be assigned to those same registers. As a consequence, easily controllable and observablable paths may need some registers to be set to two different values simultaneously. This situation can complicate the ATPG process and result in not being able to achieve the target fault coverage. Thus, we derive the second rule for register allocation :

Rule 2 : Do not assign a primary input variable and a primary output variable to the same register, if possible.

2.2 Allocation Algorithm

The following pseudocode RALLOC() show the main routine of the register allocation algorithm. The *maximum module density* NO_{tk} of operator type tk is the maximum number of module types required. It is calculated in lines 1-7, where OP_{tk} denotes the number of module types of tk. Then a list-based selection algorithm $LIST_SELECT$ is applied to primary output variables. According to the priorities of the intermediate variables, the list-based selection algorithm assigns intermediate variables V_M to the primary output registers which have been allocated to the primary output variables. This step increases the observability of the data path(line 8). The same list-based selection algorithm assigns the remaining intermediate variables V_{remain} to the primary input registers(line 9) which intends to increase the controllability of the data path. If there remain unallocated intermediate variables, we use the **left edge algorithm**[16] to allocate them(lines 10-11).

```
RALLOC()
1  forall operator type tk do
2      NO_tk = 0;
3      forall control step C_step do
4          num = NUM(OP_tk, C_step);
5          NO_tk = MAX(NO_tk, num);
6      endfor.
7  endfor.
8  (Ψ_reg, V_remain) = LIST_SELECT(V_O, V_M)
9  (Ψ~_reg, V~_remain) = LIST_SELECT(V_I, V_remain)
10 if (V~_remain ≠ { }) then
11     Ψ_reg ≡ Ψ_reg ⋃ Ψ~_reg ⋃ RALLOC_LEFT(V~_remain);
12 else
13     Ψ_reg ≡ Ψ_reg ⋃ Ψ~_reg;
14 endif.
15 return(Ψ_reg);
```

The reason we perform observability enhancement before controllability enhancement is that the number of primary output variables is usually less that of primary input variables, and one functional module has one group of output nodes but has two groups of input nodes. Therefore, enhancing observability earlier tends to improve the testability more than doing later.

The $LIST_SELECT$ first allocates primary input/output variables. Then for every partially allocated primary input/output register, $LIST_SELECT$

tries to assign more intermediate variables to it. A set of variables called the *variable search set* V_S is found, whose death time is closest to the birth time of the variable which has just been assigned to the primary input/output register. If no such variable is found, then the primary input variables are forced to be the last candidates. Thus, Rule 2 is applied in this heuristic. Given a search set V_S, $LIST_SELECT$ selects one variable from the variable search set based on the lifetime span and the operator type which produces the variables. The shorter the lifetime interval, the higher the priority it has in the ready list. As for the operator type which produces the variable, the priority order of those operators is $DIV > MUL > SUB > ADD > COMPARE > AND, OR$. Therefore, it uses **two-level priority**– the first level is the *lifetime span* and the second level is the *operator type*. Hence, we name the register allocation algorithm the **list-based two-level priority allocation**. This algorithm is similar to the ALAP scheduling proposed by Jain [4].

During the allocation process, the maximum module density NO_{tk} is used to constrain the number of variables to be assigned to one particular input/output register, as stated in Rule 1. This heuristic tends to assign more intermediate variables to the primary input/output register.

3 Module Allocation

Module allocation assigns modules to all operations. The most popular objective of module allocation is to maximize sharing of the functional units in order to reduce the total hardware cost. Other objective is to maximize the system performance. Our objective is to *achieve maximum sharing of the functional units for hardware reduction as well as maximum testability simultaneously*. In this paper, we assume only simple schedule and use a regular module library as in [10].

3.1 Methodology

When one input variable and one output variable of a module are assigned to the same register, a self loop forms. The data path with the self loop shows that the register must serve as input and output nodes of the module when deriving the test patterns by the ATPG tool. As stated before, it is rather difficult for a node to be both controllable and observable. Therefore, avoiding the generation of self loops is very important in order to synthesize easily testable data paths.

However, it is not always necessary to prevent the generation of self loops during module allocation. In the case that register in a self loop is either a primary input or a primary output register, the controllability/observability of the node in the self loop is still rather high. The ATPG program will not have great difficulty to generate the test patterns.

In module allocation phase, forming a self loop will reduce the possibility of forming a sequential loop. In general, it is harder to generate test sequency for circuits with sequential loop than circuits with self loops. Therefore, as far as testability is concerned we prefer forming a self loop to forming a sequential loop. Furthermore, a self loop with a primary input/output register is not hard-to-test.

We can then summarize the above two observations and derive the rule for module allocation.

Rule 3 : **If the register in a self loop is a primary input/output register, keep the self loop; otherwise, perform different allocations to prevent the self loop.**

3.2 Allocation Algorithm

The module allocation algorithm MALLOC() in the following allocates operators while taking into considerations the impact of self-loop on testability. For every operator in the first control step, it is allocated with a module of its type respectively(lines 1-5). For each operator in the following control steps, it finds a *module search set* M_s which is the set of modules of the same type that have been generated(lines 6-10). For every module M_i in M_S, we must check "*Is the output variable of the operator assigned to the module M_i the same as one of the input variables of the operator?*" or "*Is there any input variable being allocated to the primary input or primary output register?*". If both conditions are satisfied, then the operator is allocated to a first module in M_s, although a self loop does form(lines 12-15). Otherwise, the self loop is not generated. The above steps intend to implement Rule 3 as stated before(lines 16-19).

If those two conditions fail and the number of modules in M_s is greater than or equal to the number of the modules required within the control step, then the last member in M_S is selected(lines 21-22). Otherwise, a new module is generated and the operator is assigned to the new module(lines 23-25).

```
MALLOC()
1   MI = { };
2   (v_i1, v_i2, OP, v_o) = GET_ENTITY(G, C_step = 1);
3   forall op ∋ OP do
4      M_i <== op;  MI = MI ∪ {M_i};
5   endfor.
6   forall C_step > 1 do
7      (v_i1, v_i2, OP, v_o) = GET_ENTITY(G, C_step);
8      forall op ∋ OP do
9         M_S = MODULE_SET(OP_op);
10        M_i = NEXT(M_S);  Not_found = true;
11        while M_i ≠ ∅ and Not_found do
12          if ((v_i1 = VO(M_i)) and
13             (PI_REG(v_i1) or PO_REG(v_i1)))
14             or
15             ((v_i2 = VO(M_i)) and
16             (PI_REG(v_i2) or PO_REG(v_i2)))   then
17             M_i <== op;
18             Not_found = false;
19          else if ((v_i1 = VO(M_i)) and
```

```
20          (!PI_REG(v_i1) or !PO_REG(v_i1)))
21          or
22          ((v_i2 = VO(M_i)) and
23          (!PI_REG(v_i2) or !PO_REG(v_i2))) then
24          M_i = NEXT(M_S);
25        endif.
26      endwhile.
27      if Not_found and #M_s >= NUM(op, C_step) then
28        M_{i-1} <== op;
29      else if Not_found then
30        M_i <== op;
31      endif.
32    endfor.
33  endfor.
34  return(MI);
```

4 Interconnection Allocation

Every data transfer from a register to a module or vice versa needs an interconnection to form a path from the source to the destination. The objective of interconnection allocation is to *maximize the sharing of interconnection units and thus minimize the interconnection cost, and at the same time maximize the testability.*

Two types of interconnection topology are commonly used. The first one is the **point-to-point topology**, which is a multiplexer-oriented interconnection. The other one is the **multi-drop topology**, which is a bus-oriented interconnection. In this paper, the multiplexer-oriented interconnection is used.

4.1 Methodology

The two operands of a commutative operator such as multiply or add could be interchanged without changing its behavior. This means that the paths from the registers to the multiplexers may be interchanged. If the interchanged paths already exist, we do not have to add another new path. Therefore, the size of the multiplexer is reduced by one.

As shown in Figure 1(a), two 3-to-1 multiplexers are required if registers are allocated for (R1,R2), (R2,R3), (R3,R4) variables to hold the two input operands of three ADD operators. Figure 1(b) shows that we only need two 2-to-1 multiplexers to realize the same functions of the three ADD operators, if we interchange the inputs operands of the ADD operator from (R2,R3) to (R3,R2). The size of the multiplexer is thus reduced. The principle to increase the testability is almost the same as that of multiplexer size reduction. Whenever an operator is commutative, we can interchange the two input operands without changing its behaviors. Having more primary input registers passing through each input multiplexer increases the testability of the corresponding module. This may be accomplished by interchanging the paths from the registers to the inputs of the multiplexers. The aim is to have at least one controllable node passing through each input multiplexer of every module.

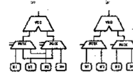

Figure 1: **(a) No interconnection allocation interchange; (b) interconnection allocation interchange.**

Based on the above observation, we thus derive the following rule for the interconnection allocation.

<u>**Rule 4**</u> **: Distribute the primary input registers among the two input multiplexers of each module as even as possible.**

4.2 Allocation Algorithm

The interconnection allocation algorithm IALLOC() attempts to equally distribute the primary input registers among the module's two input multiplexers. If the operator is not commutative then it just allocate the left multiplexer for the left variable and the right multiplexer for the right variable(lines 3-5). If the operator is of commutative type, then it first calculates the number n1 of left primary input registers and the number n2 of right primary input registers(lines 7-8). It interchanges the two input variable allocations when the difference of the two numbers n1 and n2 is greater than or equal to 1. This insures that the primary input registers have been fairly distributed among the left and right input nodes of the module(lines 9-16).

If the operator is of commutative type, and the register of the right input variable v_2 has already been connected to the left input multiplexer of the module, then it is advantageous to interchange the left and right variable allocations. The same is true for the register of the left input variable v_1 and the right input multiplexer of the module(lines 17-21). Both conditions that reduce the size of multiplexer and increase the controllability of the module may exist simultaneously without conflict.

```
IALLOC()
1  while #(G) != 0 do
2    (v1, v2, op, v_o) = NEXT(G);
3      if op != ×, +, (or/and) then
4        Left_Mux(MODULE(op)) <== v1;
5        Right_Mux(MODULE(op)) <== v2;
6      else
7        n1 = NUM_LEFT_INPUT(MODULE(op));
8        n2 = NUM_RIGHT_INPUT(MODULE(op));
9        if (n1 < n2) and PI_REG(v2) or
```

```
10          (n1 > n2) and PI_REG(v1) then
11             Left_Mux(MODULE(op)) <== v2;
12             Right_Mux(MODULE(op)) <== v1;
13       else
14          Left_Mux(MODULE(op)) <== v1;
15          Right_Mux(MODULE(op)) <== v2;
16       endif.
17       if REGISTER(v2) ∋
18          REGISTER(LEFT(MODULE(op))) or
19          REGISTER(v1) ∋
20          REGISTER(RIGHT(MODULE(op))) then
21          Left_Mux(MODULE(op)) <== v2;
22          Right_Mux(MODULE(op)) <== v1;
23       endif.
24    endif.
25 endwhile.
```

5 Experimental Results

The benchmarks presented in this paper were used in behavioral synthesis systems described in the literature and all input SDFGs are described in [10]. The SDFGs are allocated by the proposed allocation algorithms and the results are compared with those of PHITS-NS [11]. For the purpose of comparison, the data paths of PHITS-NS were also implemented with our basic primitive cells, and the ATPG is conducted on the two netlists under the same environment.

The allocated data paths are constructed with 4-bit wide elementary register-transfer-level primitives–adder, subtractor, multiplier, divider, comparator, multiplexer, and so on. The control signals from the control unit are all assumed to be directly controllable. The basic primitive cells are written in Tegas Description Language(TDL), and a sequential ATPG tool called ASICGEN [14] is used to evaluate the testability. The experiments are performed on the SPARC workstations.

The formula used to evaluate the fault coverage is :

Fault Coverage = HD / (Total Faults − IG)

- **Total Faults** : collapsed single stuck fault set.
- **HD** : faults that are Hard Detected, which means that the complement of the good signal value at the fault site is established and the fault effect is propagated to the circuit output.
- **IG** : untestable faults or signals tied to Vcc/GND.

Within the backtracking limit, ASICGEN runs until it either generates a test sequence for the fault or proves the fault being redundant. The backtracking limit is set to 10 CPU second per fault. Table 5 collects the testability results. It can be seen that our approach produces circuits with higher fault coverage and less CPU time to generate the test sequences and shorter test sequences on average.

The first example was presented in [1], and a modified form was presented in [11]. The results of Tseng's example in table 5 show that our algorithm produces circuits with a little higher fault coverage than that of PHITS-NS, the ATPG time to generate test sequency is less than half that of PHITS-NS, and the number of test patterns is a little larger than that of PHITS-NS.

The second one was presented in [3]. The results show that our algorithm can achieve more than 90% fault coverage for this circuit and is higher than that of PHITS-NS. The ATPG time is less than half that of PHITS-NS and test length produced is much shorter than that of PHITS-NS.

The third case is a Differential Equation Solver(DIFFEQ) and was first presented in [2]. As shown on row DIFFEQ in Table 5, our approach produces a circuit with slightly higher fault coverage while at the expense of doubling the ATPG time and longer test length than PHITS-NS.

The fourth case, the Fifth-Order Digital Elliptical Wave Filter(EWF) was first presented in [5], and has been chosen as a benchmark in 1988 High-Level Synthesis Workshop. Testability analysis results are shown in the last row of Table 5. Our approach achieves over 90% fault coverage which is much higher than that of PHITS-NS. The ATPG time incurred is less than 1/2 that of PHITS-NS. The length of test patterns is much shorter than that of PHITS-NS. The reason why we can achieve good test quality is due to the flexibilities in register and module allocations.

In summary, our approach always achieves higher fault coverage under the non-scan environment, and the ATPG time and the number of test patterns are usually much shorter than those of PHITS-NS.

6 Conclusions

In this paper, three algorithms for register, module, and interconnection allocations have been proposed respectively. The basic optimization algorithm under our approach is to utilize input/output registers effectively. The register allocation algorithm uses the Left Edge algorithm with two-level priority selection, which tries to assign appropriate number of variables to each primary input and primary output register and not to assign a primary input variable and a primary output variable to the same register. The module allocation algorithm generates self-loops conditionally. A self loop is formed only when the input/output registers are involved in the loop. And lastly, the interconnection allocation algorithm's strategy is to equally distribute the primary input registers among the left and right input nodes of the module.

Four benchmarks are evaluated using a commercial sequential ATPG tool under the non-scan environment. Results show that the proposed algorithms achieve higher fault coverages, take less CPU time on ATPG, and require smaller number of test patterns on average compared with the best results of similar works in the literature.

Table 1: **Testability analysis.**

Circuit	test pattern		collapsed faults		abort fault		fault coverage		ATPG time	
	P-NS	Our	P-NS	Our	P-NS	Our	P-NS	Our	P-NS	Our
Tseng	1536	1570	985	955	29	9	95.37%	95.60%	06:22	02:54
Paulin	2906	1752	862	766	76	18	87.93%	91.29%	23:28	09:16
DIFFEQ	1280	1602	951	1033	33	67	89.00%	90.84%	07:19	15:18
EWF	3676	2826	1571	1571	592	55	58.18%	91.50%	22:19	09:59
Average	2350	1938	1092	1081	183	37	82.62%	92.31%	14:52	09:22

P-NS : PHITS-NS

References

[1] C.J. Tseng, D.P. Siewiorek, "Automated synthesis of data paths in digital systems," *IEEE Trans. Computer-Aided Design*, vol.CAD-5, pp.379-395, July 1986.

[2] P.G. Paulin, J.P. Knight, E.F. Girczyc, "HAL: A multi-paradigm approach to automatic data path synthesis,", *Proc. 23th Design Automation Conf.*, pp.263-270, June 1986.

[3] P.G. Paulin, J.P. Knight, "Force-directed scheduling for the behavioral synthesis for ASIC's," *IEEE Trans. Computer-Aided Design*, vol.8, pp.661-679, June 1989.

[4] R. Jain, A. Mujumdar, et al., "Empirical evaluation of some high-level synthesis scheduling heuristics," *Proc. 28th Design Automation Conf.*, pp.686-689, June 1991.

[5] S.Y. Kung, H.J. Whitehouse, T. Kailath, *VLSI and modern signal processing,* pp.256-264, Prentice-Hall Inc., 1985.

[6] C.H. Gebotys, M.I. Elmasry, "Integration of algorithmic VLSI design synthesis with testability incorporation," *IEEE J. Solid-State Circuits*, vol.24, pp.409-416, Apr. 1989.

[7] A. Mujumdar, K. Saluja, R. Jain, "Incorporating testability considerations in high-level synthesis," *Proc. FTCS-92*, pp.272-279, June 1992.

[8] L. Avra, "Allocation and assignment in high-level synthesis for self-testable data paths," *Proc. ITC-91*, pp.463-472, Oct. 1991.

[9] C.A. Papachristou, S. Chiu, H. Harmanani, "A data path synthesis method for self-testable design," *Proc. 28th Design Automation Conf.*, pp.378-384, June 1991.

[10] T.C. Lee, W.H.Wolf, N.K. Jha, J.M. Acken, "Behavioral synthesis for easy testability in data path allocation," *Proc. ICCD-92*, pp.29-32, Oct. 1992.

[11] T.C. Lee, N.K. Jha, W.H.Wolf, "Behavioral synthesis for highly testable data paths under the non-scan and partial scan environments," *Proc. 30th Design Automation Conf.*, pp.292-297, June 1993.

[12] C.H. Chen, D.G. Saab, "Behavioral synthesis for testability," *Proc. ICCAD-92*, pp.612-615, Nov. 1992.

[13] M. Abramovici, M.A. Breuer, A.D. Friedman, *Digital system testing and testable design,* Computer Science Press, 1990.

[14] "Picasso User's Manual, V1.0.8" SynTest Technologies, Inc., 1994.

[15] K. Thearling, J. Abraham, "An easily computed functional level testability measure," *Proc. ITC-89*, pp.381-390, Oct. 1989.

[16] F.J. Kurdahi, A.C. Parker, "REAL: A program for register allocation," *Proc. 24th Design Automation Conf.*, pp.210-215, June 1987.