TM-0672

# Redesign Mechanism for Logic Design Support System

by
T. Kakuda, F. Maruyama, Y. Matsunaga
& N. Kawato (Fujitsu)

January, 1989



ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

# Redesign Mechanism for Logic Design Support System

*Taeko KAKUDA, Fumihiro MARUYAMA, Yusuke MATSUNAGA, Nobuaki KAWATO*

Fujitsu LTD.
1015 Kamikodanaka, Nakahara-ku
Kawasaki 211, Japan
Tel: +81-44-777-1111

*abstract*

We have been developing a logic design support system which inputs a behavioral specification for hardware and generates a collection of CMOS standard cells under given constraints on area and time[1]. This system refines, evaluates against the constraints, and redesigns automatically when a constraint violation occurs. Preliminary results on this work were reported in [1]. In this paper, we describe implementation details focusing on a redesign mechanism and give results of the evaluation of the system. Furthermore, we discuss remaining problems to be solved in order to improve the ability of the system.

Datapath components such as adders and counters usually have several alternatives which have different sizes and delay times. The optimal alternative varies depending on given constraints and other parts of the whole circuit. The design does not progress until a decision is made, so the most plausible alternative at the time is usually selected. However, later evaluation against constraints may show that the decision was incorrect and must be retracted. Redesign is thus necessary to satisfy all the constraints.

Such a design process, which inevitably involves tentative decisions, has some analogy with assumption-based reasoning that uses both facts and assumptions which can be retracted. We treat design decisions as assumptions and constraint violations as contradictions, and consider redesign to be contradiction resolution. We implement the redesign mechanism based on justifications for constraint violations.

Justification was originally introduced for truth maintenance to manipulate information containing assumptions. Our system stores justifications for constraint violations and uses them to redesign. We call such justifications "Nogood Justifications (NJs)". NJs are stored in a hierar-

chy which represents design objects, and a redesign algorithm is defined on the hierarchy.

**Hierarchy of Design Objects**

Design objects are represented in a hierarchy, because design is done hierarchically. This is shown in Fig.1. The hierarchy consists of component nodes and alternative nodes. A component node, which corresponds to a component at each level, associates alternative nodes as possibilities of implementation. An alternative node, which corresponds to a design alternative, contains information about the connection between subcomponents and has the subcomponent nodes as children.

Each design alternative is produced by a rule-based approach in our system, and the hierarchy is generated as the design proceeds. An alternative is called *in* if it is adapted and *out* if it is discarded. *Out* alternatives are also preserved to be recalled later when necessary for redesign.

**Nogood Justifications (NJs)**

An NJ is a logical expression that consists of inequalities concerning constraints on area and time. Each NJ is put at one of the alternative nodes in the hierarchy and represents a condition to inhibit the alternative to be selected. For example, when a datapath consists of components A, B, and C, and A has an alternative A1 with 150 cells, A1 is inhibited if the following NJ is satisfied:

150 + #B + #C > Constr(datapath),

where #B and #C mean the number of cells of B and C respectively, and Constr(datapath) means the area constraint of datapath in terms of the basic cell count.

NJs are generated in three ways. First, given constraints are transformed in advance into default NJs, equivalent to the original constraints in that any design violating the constraints satisfies them. During the redesign, NJ expansion and NJ synthesis generate new NJs to inhibit the same alternative to be selected under similar or worse conditions. NJ expansion generates a refined NJ for one of subcomponents that the original NJ refers to and stores it at the

alternative node one level down. NJ synthesis generates a generalized NJ at the alternative node one level up.

**Redesign Algorithm**

Redesign is invoked when a default NJ turns out to be satisfied. The redesign algorithm begins with the NJ, from the alternative node where the NJ is put. It selects a subcomponent to be changed and expands the NJ for it. It repeats this process going down the hierarchy until the generated NJ does not refer to a subcomponent. Then it discards the alternative with the NJ, and adopts an *out* alternative or a new alternative produced using rules. It checks to see if any NJs at the ancestor nodes, including the alternative node itself, are satisfied. If one is found, the algorithm resumes with the NJ.

If every alternative of a component is inhibited by NJs, the logical product of the NJs corresponding to each alternative is produced by NJ synthesis at the alternative node one level up and the algorithm resumes with it. This NJ does not refer to the inhibited component; another component is thus selected.

As long as there are nodes without an *in* alternative, the redesign algorithm selects one of these nodes and resumes with it. When every node has one *in* alternative, the design completes.

This work is done under R&D activities of Fifth Generation Computer Systems Project of Japan. We have implemented the system containing this redesign mechanism on PSI in ESP, and tested for a small example changing given constraints. Efficiency of the redesign algorithm largely depends on how to select a component to be changed. We are going to carry out evaluation on this.

**Reference**

[1]Maruyama,F. et al.,"co-Lodex: A Cooperative Expert System for Logic Design"
FGCS'88

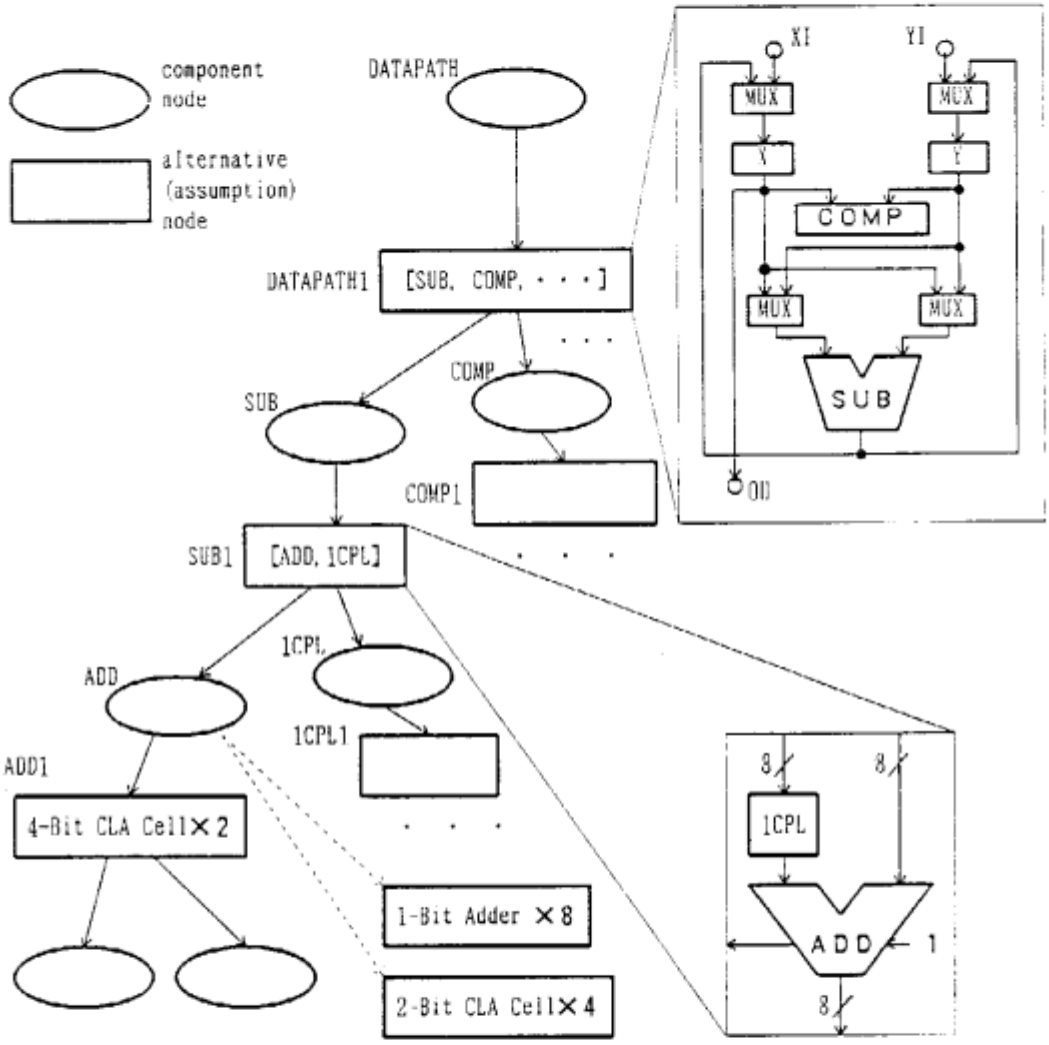


Fig. 1 Hierarchical design description