

**Figure 3. Incremental synthesis**

ally, or both. Thus it needs to be told how to relate primary IOs in the implementation to the primary IOs in the logic before synthesis. Similarly it needs to relate latches in the two logic specifications.

Some methodologies retain latch names throughout the synthesis process, some retain net names, some retain neither, but use other identifications. In extreme cases we need to use random pattern simulation to find the latch correspondence.

Similarly, incremental synthesis also needs to relate the old and new specifications. For that we need to relate primary IOs, which is design language dependent, but fortunately we do not need to relate latches. While the correspondence between primary IOs is the only one necessary to find the changes in the new specification, any other correspondence about internal nets helps. (It speeds up the process and can result in smaller changes to the implementation.)

How such correspondence can be inferred depends on the compiler of the design language. Therefore incremental synthesis needs to be told about the design language.

Since incremental synthesis combines two pieces of logic, old implementation and new specification, it needs to make sure that the result will be consistent with the expectations of tools following synthesis. For example, suppose that a latch A in the new specification is implemented by reusing a latch named B in the old implementation. Should it be called A or B? Sometimes the answer is A, sometimes B and sometimes it does not matter as long we indicate the correspondence by the means needed by any tool run after synthesis (e.g. verification).

Since methodology controls tend to remain constant for the duration of a whole project, they need to be specified only once. In contrast designer's expectations vary from day to day, and will be discussed next.

## 5.2 Design specific control

Designers commonly require that logic cones of primary outputs and latches unaffected by changes in the new version remain identical after incremental synthesis. We do guarantee that, but we cannot guarantee that the delay of the unmodified outputs will remain the same, because nets used by the unmodified outputs may also be used in the modified portions, thus changing their loading. Similarly faults in an unmodified cone might have been testable in the

old implementation, but not in the new one. Therefore a post-processing step may be necessary to adjust power levels or to remove untestable faults even in the logic reused from the old implementation. Whether this post-processing step is acceptable depends on why incremental synthesis is used. It may be acceptable in the earlier stages of the design process, but not later when the designer wants to minimize disruption to his chip.

Incremental synthesis can be used for different purposes. Sometimes a designer changes his specification to speed up his design, while keeping functionality the same. In that case he wants to resynthesize all the modified statements even though from a functional point of view it would be legal to reuse the old implementation. Other times the designer actually does change functionality and wants the minimal amount of resynthesis - only as much as is necessary to get the new function.

In some designs the old synthesized implementation may not be strictly speaking functionally equivalent to the old specification. This may be due to adding some logic to the implementation, say for testability. In such a case incremental synthesis preserves the unmodified cones of logic, even if their functionality does not match the new specification, so that the testability logic need not be added to every new version. However, sometimes the designer actually has an incorrect implementation and wants incremental synthesis to correct it. In that case we can reuse the old implementation only where it computes the correct function.

Besides these controls specific to incremental synthesis, there are other controls, which are similar to the ones described earlier in this paper.

## 6. Conclusions

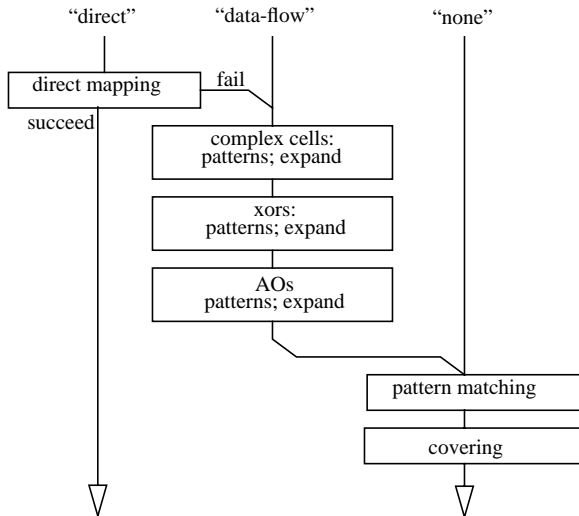
We have presented a variety of synthesis controls that the designer can use to guide synthesis to the desired solution. Most of these controls affect the way synthesis deals with the structure of the logic. In addition to being able to specify the structure completely, these controls allow a more flexible approach. The designer can control the level of restructuring, affect the mapping process and reuse old designs to generate new designs. We also described a method for controlling the run-time.

## 7. Acknowledgments

We would like to thank Bob Kanzelman and Leon Stok for reading the paper and giving suggestions.

## 8. References

- [1] Brand, D. and A. Drumm, S. Kundu, P. Narain: "Incremental Synthesis", to appear in: Digest Int. Conf. on Computer Aided Design, Santa Clara, Nov. 1994.
- [2] Drumm, A.D. and C. P. Sweet: "Logical Synthesis", U.S. patent no. 5,029,102, Issued July 2, 1991.
- [3] Kung, D. and R. F. Damiano, T. A. Nix, D. J. Geiger: "BDDMAP: A Technology Mapper based on a New Covering Algorithm", In: Proc. Design Automation Conf., pp. 484-487, Anaheim, June 8-12, 1992
- [4] Roth, C. and T. Brodnax: "The PowerPC 604 Design Methodology", in these proceedings.



**Figure 2. Mapping flow**

expanded to these primitives to allow for logic optimization unless they are marked with “data-flow” or “direct” keywords. The marked functions are kept intact through the optimization steps and into the technology mapping step. Optimization will be limited to the logic between these functions even if set to the highest level. This allows a mixture of logic styles to coexist and lets the designer select the optimization level without regard for the portions which must not be restructured.

“Data-flow” logic and “direct” logic are treated essentially the same through the optimization step. Recall that “direct” implies stricter control: “What I wrote is what I want.” BooleDozer tries to perform a one-for-one mapping into the technology, essentially one source statement into one technology cell. So a statement describing a NAND function will become a NAND cell provided an equivalent cell exists. This is done in the first part of the mapping step. Once logic boxes are mapped, no further mapping action will affect them.

Occasionally, there will be no direct technology match for a particular function. There are a number of possible reasons for this but an important one is that the logic may have been designed with a particular technology in mind but is now being mapped into a different technology. In this case, any unmapped “direct” logic boxes are automatically added to the set of “data-flow” logic boxes. Logic marked as “data-flow” is considered next.

Mapping in BooleDozer is done by locating patterns in the logic which may be replaced by technology cells. Every logic box will be covered by one or more patterns and a tiling process [3] is used to select the best set of patterns to cover all the logic. Finding a good set of patterns in the logic which make efficient use of the technology is a complex procedure. However, “data-flow” logic is already well structured according to the designer. It follows that this information should be used to generate patterns to seed the covering algorithm. So, the mapping process for this logic is to first find one-for-one technology cell matches using the same approach as above for “direct” logic. Now, rather than performing the actual mapping, patterns representing this mapping are added to the design. Patterns may also be generated for complementary functions (e.g. both AND-OR and AND-NOR). Where there is no direct match, simple factoring may be used to find a pattern of cells which still keep the overall structure intact. This factoring exists only in the pattern; the logic itself is undisturbed.

Complex functions are then expanded into a less complex form. For example, a one bit adder is expanded into an XOR and an AND-OR. The process of finding matching technology cells and adding corresponding patterns is then repeated until all the logic is expanded to primitives. At this point, a limited set of nondestructive optimizations is applied such as redundancy removal. This increases testability while connections and wiring are decreased when compared to manual design. Finally, normal pattern generation is performed on all unmarked logic as well as the expanded “data-flow” logic. The tiling procedure is then free to choose from among all the patterns to find the best cover.

So we see that a range of structure dominance is available for the designer to exercise. Synthesis has complete freedom to optimize unmarked logic. It has the freedom to select a good mapping cover for “data-flow” logic with the implicit structure providing a basic seed for the mapping. And, it has little freedom to alter “direct” logic but it still automates the mapping process and allows the design description to remain technology independent.

## 5. Incremental Synthesis

The most common complaint about logic synthesis is the problem of consistency from one run to the next. When a designer changes a parameter in his specification, be it either the source description, or one of the constraints (such as timing constraints), the synthesis implementation may not meet his expectations. For instance, the designer may make a little change in his specification (to correct a functional mistake or for better timing) and synthesis may generate a completely different implementation. This is a consequence of synthesis having the goal of logic minimization and of the fact that two functions with only slightly different specifications may have completely different optimal representation. The problem is compounded by the fact that the optimization algorithms are heuristics, so that the new implementation may be worse than the old implementation.

BooleDozer provides a mechanism to reuse the old synthesis implementation for a new implementation[1]. It will determine the differences between the new and old specification, and then will determine which portions of the old implementation can be reused. The unusable portions are replaced by technology independent gates taken from the new specification, which implement the changes in the new specification. This combination of the old implementation and the new specification is then synthesized as usual. However, all the gates from the old implementation are protected from any transformations, hence only the gates taken from the new specification are optimized and mapped into technology. Incremental synthesis is involved with many aspects of the overall design methodology beyond synthesis itself. Therefore in addition to information about the design itself, it needs to know where the inputs into incremental synthesis are coming from, and where they go to.

### 5.1 Methodology controls

Incremental synthesis operates on three pieces of logic - the logic before any optimization for both the new and old version, (the old and new specification) and the synthesized implementation for the old version.

The logic for the new specification is the normal input into synthesis; the other two must be found in the file system. Where to find them depends on version control, which is different at different installations, and therefore has to be done by a special program tailored to each different version control convention.

Incremental synthesis makes no assumptions about how the old implementation was created; it could be done automatically, manu-

mutative (AND-OR) structures.

5. Here synthesis has complete freedom to change anything it wants. However, it may choose to retain certain structures that it (heuristically) judges to be good. Other than functional correctness, no properties are guaranteed to be maintained.
6. Destructure the network. The structure is assumed to be poor. Synthesis will attempt to remove as much structure as it can.

### 3. Budgeting of Execution time

Since it is difficult to predict in advance how hard it will be to meet the timing specifications, the run time of a synthesis run can vary widely. It is therefore important to be able to control the run time. When a limit is set on the run time it is important that the run is not cancelled without giving reasonable results. When the time expires the results should at least be mapped and written to a file. To accomplish this, the run time of the various tasks in the system can be budgeted.

The synthesis system allows three types of user specified budgets: (1) a CPU time budget, specified as the amount of CPU time that can be used, (2) a real time limit, specified as a point in time when the run has to be complete and (3) a maximum count of elementary operations. The third limit is used to ensure reproducibility.

The budget for each task is predetermined (not given by the user) as a percentage of the overall budget. The deadline for the task can be calculated from the overall budget given by the user. Because each task can consist of several sub-tasks, the budget is hierarchical and each sub-task has a budget given as a percentage of the budget of the parent task.

When a task is not completed when its deadline passes (it exceeds its budget), it is terminated at the next convenient place, and execution of the next task begins. If a task is completed before the set deadline, the remaining time can be used for the next task. In other words: the deadlines do not change because a task was completed ahead of schedule. Furthermore, it is possible to have some required tasks which are always executed, regardless of time pressure. For instance, one always wants the synthesis result written to a file, regardless of how far the deadlines are exceeded.

Because the synthesis scenario consists of a hierarchical set of procedures, the budget is done hierarchically, such that for small tasks a budget can be set, and such that some required tasks are always executed.

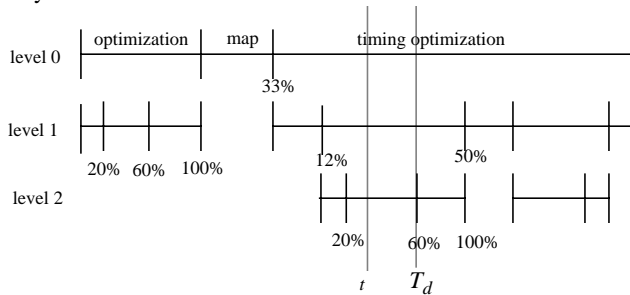


Figure 1. Hierarchical time budgets

Let  $T$  be the total budgeted time (CPU time or real time), let  $B_{i,c}$  be the  $c$ th budget (as a percentage) at level  $i$ , let  $c_i$  be the number of the current budget at level  $i$ . The next deadline  $T_d$  at level  $L$  is

$$T_d = T \sum_{i=0}^L B_{i,c_i} \prod_{j=0}^{i-1} (B_{j,c_j} - B_{j,c_{j-1}})$$

As an example consider figure 1. At time  $t$ ,  $c_0 = 2$ ,  $c_1 = 5$ ,  $c_2 = 1$ . The total budget is  $T.(B_{0,2} + (B_{0,3} - B_{0,2}). (B_{1,5} + (B_{1,6} - B_{1,5}). B_{2,2})) = T.(33\% + (100\% - 33\%). (12\% + (50\% - 12\%). 60\%))$

### 4. Dominant Mapping

Designers often plan out the detailed implementation of parts of their designs. These sections are usually the most critical and the most difficult to get right and tend to be dataflow logic. The size and the shape of such designs have been generated based on the designer's experience and knowledge of the overall requirements. Many designers used to find that they could not use synthesis for such designs, instead, designing them by hand. This is time consuming and error-prone and last minute technology changes can invalidate the manual design, meaning a missed deadline.

We've seen how optimization may be controlled by the designer but sometimes this is not enough. A designer may not wish to avoid all optimization yet still wishes to maintain the structure of certain paths in the design.

Another problem is the mapping of complex functions to complex cells such as multiplexors, decoders, and adders. These cells are often more efficient in size or speed or both. Logic optimization, especially factoring, can disguise these functions making them difficult to recognize. For instance, a selector needs to have all its select inputs one hot encoded for correct operation. Functionally it can be seen as a large AO. If it is expanded into primitives, then the select signals may no longer be 1 hot after optimization, and it will be difficult to recognize that a selector can be used to implement the function. Again, designers may resort to direct entry of technology cells eliminating the advantages of automation and synthesis and tying the design to a single technology cell library.

Both of these problems are addressed in BooleDozer. A designer may mark portions of the logic to indicate the level of structural dominance that should be imposed[2]. Structural dominance means the implicit logic structure in the original design description "dominates" the mapping. That is, the structure of the technology level netlist closely resembles the structure of the original design. The levels are, in order of increasing structure dominance, "none," "data-flow," and "direct." The designer marks the logic by placing attributes on statements in the design description. "None" means that synthesis may process the logic normally under the constraints of the other controls described earlier and may completely restructure the logic. "Data-flow" and "direct" are both used to maintain the logic structure implied in the design description. This provides several benefits for a logic designer. First, it allows for a more natural partitioning since dataflow and control logic do not have to be artificially split between partitions. Second, logic optimization may be used fully without concern for how it may restructure the critical parts of the logic. Third, it provides a means to define the logic at a low-level where necessary, limiting the role of synthesis while remaining technology independent. This can be essential to making a tough design work on a tight schedule. Again, a single design may contain one or all of these logic styles.

BooleDozer considers each logic statement as a unit and recognizes structural constructs. For instance, a sum-of-products expression is represented in logic as an AND-OR gate. Or, a set of comparisons between a bus and various constants may be represented as a decoder. In this way, there is a close correspondence between the structure entered by the designer and the logic structure represented within BooleDozer. Attributes on the source statements become keywords on the logic boxes in the synthesis model.

Functions more complex than AND, OR, and NOT are normally

# In the Driver's Seat of BooleDozer

Daniel Brand, Robert F. Damiano,  
Lukas P.P.P. van Ginneken  
IBM Research Division  
Yorktown Heights, NY

Anthony D. Drumm  
IBM Application Business Systems Division  
Rochester, MN

*Abstract - This paper describes some of the synthesis controls in the BooleDozer<sup>1</sup> synthesis system which are unique in concept and implementation. Rather than attempting to achieve the maximum amount of optimization in the minimum amount of run time, the designer specifies the restructuring level which allows him to specify to what extent the original structure should be preserved. We also describe controls which affect the mapping process. Finally we describe the incremental synthesis feature. The run time can be accurately controlled by a run-time budgeting mechanism.*

## 1. Introduction

Normally most controls give hard parameters to the system. For instance, a control will set the timing specifications of the logic, set rules for fanin and fanout, specify the standard cells in the library. This type of control gives the constraints under which the synthesis algorithms must work. However, just giving such constraints has proven not to be sufficient.

The process of logic synthesis is very complex, and even very small suboptimizations have been proven NP-complete or worse. Despite intensive research, and great progress in logic synthesis algorithms, most algorithms that are used are greedy heuristics, which cannot be guaranteed to deliver optimal results. In cases where the algorithms fail to give the expected results, logic synthesis needs to be guided towards a good solution. In BooleDozer, we developed several means of controlling synthesis.

The most primitive form of such a control disables synthesis for a particular piece of the circuit, the way the "dont\_touch"<sup>2</sup> attribute works. Often, the designer wants a more flexible control to take advantage of certain capabilities of the synthesis system, while performing other tasks manually, either because the synthesis results are not satisfactory or the run time is too large. For instance, the designer may want to specify the overall structure of a design, but wants synthesis to perform the mapping.

In the remainder of this paper we will present 4 types of controls: restructuring controls, (which control the level of optimization), mapping controls, incremental synthesis and execution time budgeting.

## 2. Restructuring Controls

A common need in a synthesis system is to have control over the level optimization. Usually this control will trade-off the amount of effort (CPU time) against the quality of the result. For instance, at a low level it runs the most effective optimizations, attempting to get a lot of optimization with little run time. At a high level all available transforms are run, perhaps even several times.

The problem with such a control is that it is extremely unpredictable - the run time is very difficult to predict from the setting of the control. The run time depends on the size of the circuit and above all on how difficult the timing constraints are to achieve. Our alternative to control CPU-time will be explained in section 3.

Another undesirable effect is that it is usually not guaranteed that more CPU time will yield a better result. Because of the heuristic nature of some of the algorithms involved, it is possible that a longer CPU time will make results worse. This is primarily caused by the difficulty of accurately predicting area and timing in an early stage of synthesis.

To avoid this, the usual level of optimization control is replaced by a restructuring control. The restructuring control is a flexible way of controlling the amount of optimization. A higher level of restructuring gives synthesis more freedom to change the network, but also increases the level of unpredictability. This control affects primarily the early stages of synthesis, as these stages change the structure of the network most, while the area and timing estimates at this stage are most inaccurate. Each level of restructuring preserves some important property of the network. The levels are ordered such that each level also preserves the properties that are preserved at a higher level. The levels that we distinguish are:

0. At the lowest level, nothing is changed so the entire structure of the network is preserved.
1. At this level the fanin and fanout of no net in the logic is increased. This is to avoid any changes that might increase the delay - assuming that the delay cannot be estimated accurately at the early stages of synthesis. Optimizations that are done at this level include making the logic testable by removing redundant connections; constants are propagated and dangling logic is removed.
2. At this level, the number of logic levels on any path from any primary input to any primary output will not be increased. This again is to avoid changes that increase delay. At this level we assume that delay is mostly influenced by the number of levels in a path. Optimization at this level include the sharing of common expressions (but not factoring sub-expressions) and bringing signals forward in the logic by forward and backward global flow analysis.
3. Here only transformations which decrease the number of connections are allowed. Due to the heuristic nature of synthesis, we cannot otherwise guarantee with certainty that the final logic is not larger than the initial logic. However, common subexpressions may be factored, increasing the number of levels.
4. At this level the number of connections may temporarily increase by collapsing cubic factors, however the OR-AND-OR structures are not multiplied out. This assumes that commutative structures are much easier to rearrange for timing than non-com-

1. BooleDozer<sup>TM</sup> is a trademark of IBM Corp.

2. dont\_touch<sup>TM</sup> is a trademark of Synopsys Inc.