John A. Darringer
William H. Joyner, Jr.
C. Leonard Berman
Louise Trevillyan

Logic Synthesis Through Local Transformations

A logic designer today faces a growing number of design requirements and technology restrictions, brought about by increases in circuit density and processor complexity. At the same time, the cost of engineering changes has made the correctness of chip implementations more important, and minimization of circuit count less so. These factors underscore the need for increased automation of logic design. This paper describes an experimental system for synthesizing synchronous combinational logic. It allows a designer to start with a naive implementation produced automatically from a functional specification, evaluate it with respect to these many factors, and incrementally improve this implementation by applying local transformations until it is acceptable for manufacture. The use of simple local transformations in this system ensures correct implementations, isolates technology-specific data, and will allow the total process to be applied to larger, VLSI designs. The system has been used to synthesize masterslice chip implementations from functional specifications, and to remap implemented masterslice chips from one technology to another while preserving their functional behavior.

Introduction

The goal of generating an acceptable, technology-specific hardware implementation from a functional specification is not a new one, and it has received much attention in the past. The nature of this problem depends on the level of the functional description, the set of implementation primitives, and the criteria of acceptability. Early work centered on developing algorithms for translating a boolean function into a minimum two-level network of boolean primitives. Extensions were developed for handling limited circuit fan-in and alternative cost functions [1, 2]. But because these algorithms search for minimal implementations they require time exponential in the number of circuits and thus cannot be used on most actual designs.

Other efforts have attempted to raise the level of specification. The DDL work at Wisconsin [2-4], APDL at Carnegie-Mellon University [5], and ALERT at IBM [6] all began with behavioral specifications and produced technology-independent implementations at the level of boolean equations. The results were usually more expensive than manual implementations and did not take advantage of the target technology. For example, the ALERT system was validated on an existing design, the IBM 1800, and the implementation produced required 160% more gates than the manual design [7].

Attempts have been made to produce more efficient logic and to give the designer more control over the implementation [8-10]. This control has resulted in specification language constraints, so that the specification is at a fairly low level and in closer correspondence with the implementation. This necessarily decreases the advantage of an automated approach, bringing it closer to a system for logic entry than for logic synthesis.

Several tools have been developed at Carnegie-Mellon University to support the early part of the design cycle [11-14]. In one experiment [15] the CMU-DA (Carnegie-Mellon University-Design Automation) system was used to implement the data path portion of a Digital Equipment Corporation (DEC) PDP-8/E. It began with a functional description of the machine and produced an implementation in two technologies of the registers, register operators, and their interconnections, but not the control logic to sequence the register transfers. When the target technology was TTL series modules the implementation required 30% more modules than the DEC implementation. With CMOS standard cells it required 150% more area than an existing Intersil chip.

There has also been recent work in logic remapping, transforming existing implementations from one technology to another. A group in Japan has described a system to help a designer translate an existing small- or medium-scale integration implementation into large-scale integration [16].

Our approach focuses on the control portion of synchronous machines, since that design is more error-prone than data path design. Thus we assume that all memory elements of the final implementation are identified in the specification; the goal is to generate the combinational logic that computes, on each clock cycle, new values of outputs and memory elements from inputs and the old values of the memories. Also we are focusing on producing random logic implementations, initially for masterslice chip implementations, instead of generating microcode for a control processor or using a programmable logic array. Our initial experiments have been with logic for single chips, so that chip interface information (inputs, outputs, polarities, sender/receiver requirements) was assumed to be specified. The implementations produced by our system are composed of primitives selected from a specified set, connected to satisfy given performance requirements and technology restrictions, and ready to be placed on a masterslice chip.

In a previous paper [17] we described our approach to this form of synthesis; the present paper is an expansion on work reported in [18]. We are not proposing a completely automatic replacement for the manual design process. Instead, we envision an interactive system in which the user operates on a logic design at three levels of abstraction. He begins with an initial implementation generated in a straightforward manner from the specification. He can simplify the implementation at this level, and, when satisfied, can move to the next level. He does this by applying transformations, either locally or globally, to achieve the simplification or refinement. By being able to operate on the implementation at several levels, the user can often make a small change at one level that will cause a larger simplification at a lower level. By limiting the user to directing function-preserving transformations, we can ensure that in all cases the implementation produced will be functionally equivalent to the specified behavior.

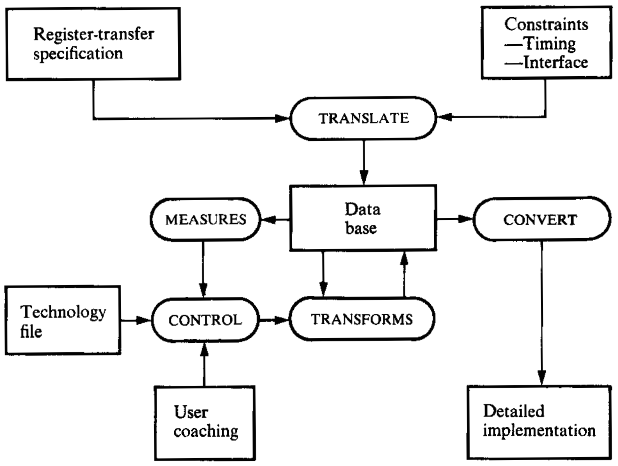

Figure 1 The logic synthesis system.

The use of transformations and levels of abstraction allows a modified form of this scenario to be used in remapping designs from one technology to another. "Remapping" usually refers to the one-to-one substitution of new technology primitives for old technology primitives. Our approach is different: We first transform technology-specific primitives to ones at a higher technology-independent level. To this intermediate-level representation we can apply the synthesis transformations to produce an implementation in a different target technology with the benefit of simplification at several levels.

Both logic synthesis and remapping are problems of finding feasible (not optimal) implementations: networks of primitive boxes that satisfy a large number of constraints. In addition to gate and I/O pin limitations, there are timing constraints, a restricted library of primitives, driver requirements, clock distribution rules, fan-in and fan-out constraints, and rules for testability. Since we hope to apply our techniques to VLSI chips, we are attempting to limit our transformations to local changes that do not require time or space exponential in the number of circuits.

An experimental system for logic synthesis and remapping

The organization of the logic synthesis system is shown in Fig. 1. Its inputs are the register transfer specification, the interface constraints, and a technology file which characterizes the target technology. The output is a detailed implementation in terms of the primitives of the target technology, which is submitted to placement and wiring programs for physical design. Some timing or other physical problems may not be detectable before placement and wiring. In this case the synthesis process

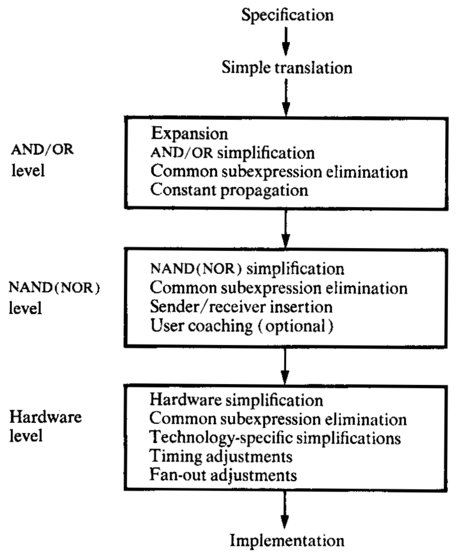

Figure 2 The scenario of synthesis.

is repeated with a revised specification or modified constraints until an acceptable implementation is achieved.

An important requirement of our approach is that the data base be capable of representing the implementation at different levels of abstraction. Our system to support logic synthesis makes use of a graph-like internal data structure for storing the implementation as it progresses from the higher-level description to its final form, and all transformations operate on this graph. There is a single organizational component: the "box." A box has input and output terminals which are connected by wires to other boxes. Each box also has a type, which may be a primitive or may reference a definition in terms of other boxes. Thus a hierarchy of boxes can be used, and an instance of a high-level box such as a parity box can be treated as a single box or expanded into its next-level implementation when that is desirable.

The logic synthesis data base is implemented using a system originally developed for use in an experimental compiler project within IBM Research [19]. It is made up of two groups of tables. The first group describes the technology being used; it is created from a technology file containing for each box type information such as name, function, and number and names of input and output pins. These data are created in batch mode and read during initialization of the interactive system.

The second group of tables contains the representation of the logic created by the interactive system. This group consists of a box table, a signal table, and a set of auxiliary tables which describe the relationship between the boxes and the signals. There is some intentional redundancy in the data; each box has a complete list of input and output signals, and each signal has a source and a list of sinks. Every box table entry contains type information which provides a link to the technology group. This allows programs to get technology information about a specific box.

Transformations communicate with the data base through a layer of functions which perform all data addition, retrieval, and deletion. These functions provide the transformations with the ability to traverse a chip by following signal paths, or by visiting each box. They make it easy to remove boxes and reconnect their input/output signals, to move connections from one box to another, to insert boxes on signal paths, etc. The functions provide a conceptual view of the data base which remains stable even when the data base implementation is altered. The table structure representing this view can be significantly changed with a minimal impact on the processing programs.

The use of data abstraction, of a data base system which allows one to easily define a data base, and of modular implementation of data structures made it possible for us to quickly bring up a usable support system for the transformations. As we learned more about the requirements of the transformations, we were able to change the data base completely, to add and remove data fields, to change individual data structures, and to concentrate efforts in performance improvement in areas where experience indicated that better performance was required. In all cases, only modifications to the data management programs were required to accomplish these changes; the programs which use the data manager were completely unaffected.

The interactive design of the logic synthesis system not only allows the user to control the transform application, but also permits him to invoke programs that aid in his decisions. A BACKTRACE facility displays the cone of influence of a signal or box, showing graphically the logic producing a signal from registers or chip inputs. MEASURE lists, for a design, the number of boxes, signals, connections, inputs, outputs, cells, number of boxes of each type. SEGMENT lists, for each chip output and register, the number and names of the chip inputs and other registers influencing it, the depth of the tree with those leaves, and the number of boxes in it. PRINTBOX lists all boxes of a design and their inputs and outputs, and PRINTREF lists all signals of a design with their sources and sinks. Individual boxes and signals can also be listed in this way. Facilities also exist for producing logic diagrams from a design in the data base.

Expansion and compression commands allow the user to expand a box by replacing it with its more primitive components from a box type definition, and to identify a group of boxes and form a new type of them, replacing the group with a single box. Expansion permits hierarchical development, and compression can be used to partition a design into smaller parts.

The system will accept input in two languages. All of the examples were described in a flowchart-like language, similar to that in [20], allowing GOTOs, assignments to registers and signals, decisions based on the values of registers, computed GOTOs based on values of a group of signals, etc. Parallelism is described in this language by multiple GOTO statements which branch to several actions at the same time. We are also experimenting with a language, similar to CDL (Computer Design Language) [21], that more closely models the internal form of the data base. In addition, it allows convenient description of hardware hierarchy. This aids in the input of box type descriptions which are later to be expanded a hierarchical way, such as a parity function or a decoder.

The synthesis scenario

Though there has been some variation in the synthesis process as the system has been developed and has been applied to more examples, a fairly standard sequence of steps has emerged. Figure 2 shows the three levels of description common to our experiments: the initial AND/OR/NOT level, a NAND or NOR level (depending on the target technology), and a hardware level in which the types of the boxes are books or primitives of the target technology. At every level the implementation is a network of boxes connected by signals. Our objective in devising this scenario was to find a set of transformations and a sequence for applying them such that the original functional specification could be transformed by a sequence of small steps into an acceptable implementation. The transformations at the AND/OR level are local, textbook simplifications of boolean expressions; most of them reduce the number of boxes, but they do not produce a normal form. The NAND and NOR transforms are similar, and required more work because there was less of a foundation on which to build. The hardware transformations were developed after considerable time was spent with chip designers to understand the technologies and the motivation for the many design decisions. Transformations are used not only to simplify the implementation at each level according to appropriate measures but also to move the implementation from one level to the next. The transformations are local in that they replace a small subgraph of the network (usually five or fewer boxes) with another subgraph which is functionally equivalent but simpler according to some measure.

The initial implementation at the AND/OR level is produced by merely replacing specification language constructs with their equivalent AND/OR implementations. Methods for this translation have been described in [3, 5]. At this first level the boxes are of types such as AND, OR, NOT, PARITY, EQ, XOR, DECODE, or REGISTER. Simple local transformations are applied to reduce the number of boxes. Some of the particular transformations used are listed as follows:

$\mathrm{NOT}(\mathrm{NOT}(a)) \Rightarrow a$

$\mathrm{AND}(a, \mathrm{NOT}(a)) \Rightarrow 0$

$\mathrm{OR}(a, \mathrm{NOT}(a)) \Rightarrow 1$

$\mathrm{OR}(a, \mathrm{AND}(\mathrm{NOT}(a), b)) \Rightarrow \mathrm{OR}(a, b)$

$\mathrm{XOR}(\mathrm{PARITY}(a_1, \cdots, a_n), b) \Rightarrow \mathrm{PARITY}(a_1, \cdots, a_n, b)$

$\mathrm{AND}(a, 1) \Rightarrow a$

$\mathrm{OR}(a, 1) \Rightarrow 1$

The last two simplifications are examples of a more general constant propagation that is performed. These transformations may leave fragments of logic disconnected. We clean up this disconnected logic in a manner similar to the way compilers perform dead-code elimination. Another technique from optimizing compilers, common subexpression elimination, is also applied here and at other points in the synthesis process to further reduce the size of the implementation. The expansion of "high-level" boxes such as parity and decoders was done here in some of the experiments and was postponed to the hardware level in others. The interactive nature of the system allows this flexibility, which is useful if technology rules require that certain constructs be used for these functions. However, in most cases our simplification rules at the AND/OR and the NOR or NAND levels were powerful enough so that textbook expansions of DECODE, XOR, etc. in terms of AND/OR gates reduced to efficient technology-specific logic.

Next the AND, OR, NOT, and most other operators of the initial description are replaced by their NAND or NOR implementations. The target technologies in our experiments were either NAND- or NOR-based, and this determined the primitive selected for this level. The NANDs or NORs are "idealized," however, in that they have no fan-in or fan-out restrictions. The transition to these primitives is accomplished naively by local transformations, and may introduce unnecessary double NANDs or NORs, which will be eliminated later. Also at this point, the chip interface information is used to place generic (*i.e.*, not technology-specific) senders and receivers on the chip inputs and primary outputs, and to insert inverters where necessary to ensure the correct signal polarities.

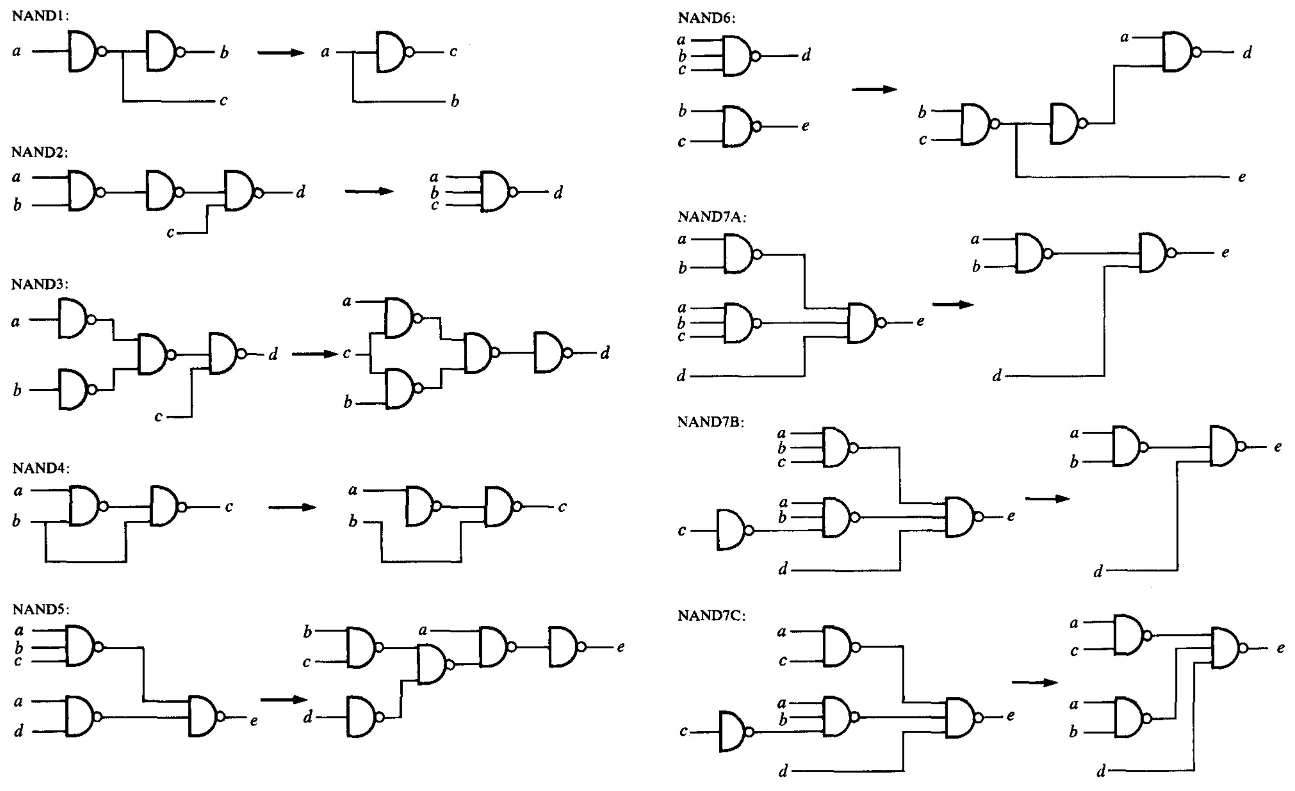

Figure 3 The NAND transformations.

Simplifying transformations are now applied to each signal in the network at this level. These transformations attempt to reduce the number of boxes of the implementation without increasing the number of connections. To accomplish this, the transformations must check the fan-out of the various signals involved, since this will affect the number of boxes and signals actually removed. The transformations are applied repeatedly throughout the network until no more apply. Figure 3 illustrates the NAND transformations used in our experiments; the NOR transformations are identical except for the operator. Each transformation has an associated condition that determines if the replacement will simplify the implementation by reducing boxes or connections. These conditions depend on the fan-out of the intermediate signals and on whether the target technology is assumed to have dual-rail output. For example, NAND3 is only profitable in certain cases. It does not appear to reduce the box or connection count, but if dual-rail outputs are assumed, the single-input NAND on the right-hand side is "free" and disappears after hardware generation. NAND5 is the dual of NAND3, but the two do not cycle because of restrictions on their application. Though NAND5 and NAND6 appear to increase box count, they decrease connections and leave box count the same if dual-rail is assumed.

In the transition to the hardware level, the NAND or NOR gates and generic registers are replaced by technology-specific primitives. Single primitives or macros are selected to match the fan-in of the actual primitives with that of the "idealized" boxes. Also the number of control and data lines of the idealized registers might exceed those normally available, necessitating the generation of additional logic. At this point the implementation is in terms of primitives used by the engineers in their implementations, but because transformations have been made locally there may be some violations of timing, fan-out, and other technology restrictions.

The simplifying transformations at the hardware level are of two sorts. Some are simplifications similar to those at the previous levels, such as eliminating the equivalent of double NOTs, which may occur as a result of expanding higher-level boxes. Others attempt to take advantage of the particular technology. For example, flip-flops may provide an output and its complement, allowing some inverters to be removed at this level. Also, because of combination flip-flop-receiver books available, some receivers may be eliminated. Wired or dotted ANDs or ORs can be introduced to reduce cell count where possible. Some technologies may be dual-rail, having both phases available at every gate; this makes possible simplifications not possible with the technology-independent earlier levels. Other technology-specific transformations applied at this level distribute clock signals to flip-flops according to the technology rules, eliminate long and short paths between flip-flops (assuming a unit gate delay and technology-specific guidelines), and adjust fan-out by repowering signals.

Several of the transforms at the three levels are analogous, differing only in the types of boxes to which they apply, so that simplifications not made at one level would be caught later. This may appear redundant; however, the application of transforms as early as possible reduces the size of the implementation and helps prevent a greater explosion in size when, for example, conversion to NANDs takes place. Though the same implementation might be produced without the NAND simplifications, they are included for efficiency.

The expansion of boxes in terms of more primitive gates was first done only at the hardware level. However, in successive experiments it was found that expansions at other levels were sometimes desirable. For example, if a counter could be expanded in terms of ANDs and ORs, the same expansion could be used for various technologies. The expansion transform therefore was extended to permit selective expansion of box types at various levels.

Synthesis experiments

The synthesis system has been used to create several chip implementations in two different technologies. In some cases, an engineer had implemented the same chip, and we were able to compare the automated design with that of the engineer. In other cases no implementation had been previously attempted.

The first experiments with the logic synthesis system were attempts to produce implementations for chips from existing processors that had been specified functionally and implemented by engineers. The existence of the engineers' implementations permitted comparison of designs and a study of the differences between manual designs and those produced automatically. Each of the experiments was carried out automatically, although the particular sequence of transformations was the result of much experimentation.

Experiment 1

For our first experiment we selected a straightforward chip that had already been manually designed. The specification described seven registers totaling 24 bits, two parity operators, and the conditions for the data transfers. The target technology was a TTL masterslice that provided 96 I/O pins and 704 cells (divided between three- and four-input NAND gates) on each chip. In addition to the NAND gates, there are a number of macros such as receivers, senders, and flip-flops that are implemented with these NAND gates. Restrictions on the use of the primitives available, such as fan-in and fan-out requirements, timing constraints, clocking and powering rules, were described in the technology file or in some cases built into the transformations. In this experiment EQ, XOR, PARITY, and other high-level boxes were not expanded until the hardware level.

In examining the implementation after the NAND transformations were applied, it was noticed that further improvements could be made. In particular, a reduction in fan-out of a signal by repowering its source would allow a transformation to apply and eventually reduce the size of the implementation. The system allows repowering and some other transformations to be applied to particular signals, rather than across the whole implementation, as a form of user "coaching." In this instance coaching saved only four boxes, but resulted in an implementation slightly better than the manual design.

The first experiment resulted in a synthesized implementation that was remarkably similar to the manual one. In fact, it required four fewer cells, five fewer connections, and four shorter paths than the engineer's implementation. The similarity, however, was not such a surprise since we had used this example in the design of our system, and since we had worked so closely with the chip's designer.

Experiment 2

In the second experiment the same sequence of transformations was applied to a more complex chip. The chip specification contained 13 register bits, a three-bit counter, a five-bit counter, two parity operators, and more complex conditions controlling the data transfers. The target technology was the same as in the first experiment. This time there was virtually no contact with the engineer who designed the chip.

While we tried to use the same scenario, we did make two changes. There was no coaching in this experiment and counters were handled differently from the EQ and PARITY in the first experiment. We found that it is better to expand the counters at the AND/OR level than at the hardware level. This exposes the expanded counter to all subsequent simplifications and allows one definition to be used for different technologies. The expansion transformation therefore has been extended to permit expansion of a nonprimitive box at any level.

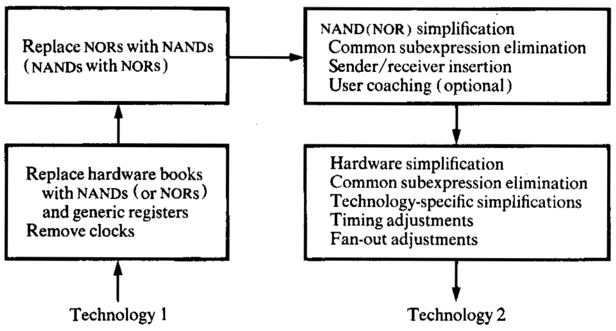
Replace NORs with NANDs (NANDs with NORs)
NAND(NOR) simplification
Common subexpression elimination
Sender/receiver insertion
User coaching (optional)
Replace hardware books with NANDs (or NORs) and generic registers
Remove clocks
Hardware simplification
Common subexpression elimination
Technology-specific simplifications
Timing adjustments
Fan-out adjustments
Technology 1
Technology 2

Figure 4 The scenario for remapping.

The synthesis of the second chip resulted in an implementation with 15% more cells and 20% more connections than the manual implementation. We are currently analyzing these results to understand why our implementation is more complex.

• *Experiment 3*
The third experiment was an attempt to synthesize another complex chip in a different technology. This third chip specification described 28 register bits, three parity operators, four decoders, seven comparators, and even more complex control logic. The target technology was an ECL masterslice. In addition to a new set of technology rules and restrictions, this meant that the basic primitive was a NOR and that each primitive had "dual-rail outputs"; that is, it provided both polarities of its output. The synthesis scenario was adapted to this technology and changed slightly, but the three levels of implementation were maintained. The decoders and comparators were expanded at the AND/OR level and the AND/OR transformations remained unchanged. Common subexpression elimination was applied more often at this level and throughout the scenario.

The NAND level became the NOR level because of the new technology. This required a new transformation to translate the AND/OR primitives into NORs, and a set of NOR simplification transformations. These were originally just the NAND transformations with the NANDs converted to NORs, but we later realized that with dual-rail outputs, an apparent box saving at the NOR level might not be a saving at the hardware level, and that the transformation might increase fan-in or number of connections. Thus different fan-out restrictions were used in the NOR transforms. The technology-specific transformations had to be rewritten for the new technology, and some new ones were added, such as the one to eliminate inverters.

This experiment resulted in an implementation with 5% more gates than the manual one. We are trying to account for this additional logic and determine if it could be eliminated through local transformations.

The remapping scenario

The logic synthesis system has been used to remap chips from one technology to another. Our approach to remapping is not to attempt a one-to-one mapping of hardware primitives, but first to abstract from the hardware level to the technology-independent NAND or NOR level, with generic registers, drivers, and receivers. The NANDs (or NORs) can be mapped to NORs (or NANDs) in a straightforward way, and the NAND/NOR and hardware parts of the synthesis scenario can be applied to produce an implementation in the target technology. This required two new transformations, one that transformed primitives at the hardware level back to the NAND level, and a second that transformed the NAND implementation into a NOR one, while preserving the chip input/output behavior. This approach is better than the straightforward replacement of old technology primitives by new ones, since it exposes the remapped implementation to the simplifications at the NOR level and at the hardware level. Figure 4 outlines the remapping scenario.

Remapping experiments

The first experiment performed was to transform a chip implementation from a TTL masterslice to an ECL masterslice. The chips were of comparable capacity and this chip-to-chip remapping was possible. Since this chip conversion had not been performed manually we could not make an objective comparison. We did check that the input/output behavior was preserved and showed the implementation to an experienced engineer, who found no serious problems.

Chip-to-chip remapping is rare. Usually a new technology will have a different density and number of pins. This could require a merging of several chips from the initial implementation and a partitioning of that remapped, larger function into the chips of the target technology.

Observations

• *Comparing implementations*
One of the problems that confronts us is the difficulty of evaluating the result of the synthesis process. In our work to date, this evaluation has meant a comparison between our generated implementation and a manually produced implementation. There are two aspects to the comparisons that we must perform. One is the problem of determining functional equivalence between the two implementations. The other is to furnish a response to the ill-posed question: "How do these implementations differ?"

Functional equivalence in its full generality is the problem of boolean equivalence and is known to be co-NP complete. This implies that at our present level of understanding it is not possible to devise a program which will efficiently, in all cases, decide equivalence between two implementations. In our case, the problem is often complicated by "don't care" conditions—certain combinations of inputs may be known not to occur. We cannot solve the functional equivalence problem, but we are exploring heuristics which may offer significant speed-up on a large class of implementations. A report on this work is in preparation [22].

Even when two implementations are functionally equivalent, we are still interested in their structural similarity. This form of comparison permits us to evaluate a stylistic difference between our implementation and that produced by an engineer. This is necessary for discovering new heuristics. For this form of comparison we are considering formalizing the notion of "distance" between two implementations, following an analogy to the spelling correction problem.

• *Completeness and coaching*
A desirable property of a set of transformations is completeness—it should be possible to reach any NAND realization of a boolean function from any other by application of the transformations. Our set of NAND transformations does not have this property. Any set of transformations complete in this sense must allow application of transformations in the reverse direction, and this would prevent an automatic application of transformations throughout a design from terminating. What seems desirable is a complete set of bidirectional transformations, with a set of preferred (*e.g.*, box-reducing) directions, yielding a set which terminates with a "good" implementation. The reverse directions would also be available, but only in a user "coaching" mode—they could be invoked on particular parts of the design.

The desire to avoid user-invoked transformations leads to the development of more complicated criteria under which a transformation is to be applied. For example, the coaching described in the first experiment invoked a transformation which would, if applied uniformly, increase the number of boxes in the design. Allowing it to be applied at a particular place by the user has the advantage of providing the (eventual) design improvement desired in the particular case while avoiding building into the transformation constraints on its application. Such constraints may sometimes be worthwhile, but they will make the transformation less local by requiring examination of a larger part of the logic.

• *Technology-specific information*
The technology file allows some generic transformations to apply at all levels of the synthesis process by testing the function of a box to which a transform is to apply, rather than its box type (which may be a hardware primitive). For instance, though it may be necessary to apply a double inverter removal at all three levels, the same transform can be used to do this for NOT, NAND, NOR, and various hardware primitives. A more ambitious use of the technology file would be in hardware generation. For example, a four-way NAND with one input receiving an off-chip signal could be translated by looking in the technology file for a primitive in the target technology implementing that function. It appears that some transformations with specific hardware information built in, such as clock distribution tree generation, will always be necessary.

Future work

Our plans include further analysis of the results of our experiments to determine what improvements should be made to our system. We will also look at more ambitious chips—chips that have required minimization or that have caused long path problems when implemented manually. We hope to arrive at a set of measures and transformations that will provide acceptable implementations for a large class of examples. In addition, we will explore the following:

- multi-chip synthesis—starting with a functional specification that requires several chips, developing additional measures and transformations that will trade resources across chip boundaries.
- engineering changes—examining how such a synthesis system could respond to engineering changes where minimum, local changes are highly desirable.
- transformation specification—looking at how transformations could be described at a high level and compiled for efficient application.
- transformation correctness—considering what properties of transformations (such as function-preservation) should be proved and demonstrating how such proofs can be accomplished.

Summary

We are in the process of exploring what we believe is a new approach to the old problem of logic synthesis and are encouraged by our initial experiments. We have built an experimental synthesis system and used it to synthesize several masterslice chips. In the cases in which we were able to compare our results with previous manual implementations, we found that the automatically produced ones required 0% to 15% more logic. The results are similar when comparing numbers of signals or numbers of connections. We have also used our system to remap implemented chips into a new technology, while preserving their input/output behavior. We plan to perform further experiments, to study the remaining differences between the automatic and manual implementations, and to improve the competence of our experimental system. Our hope is that computationally manageable techniques based on local transformations can be applied to improve naive implementations to acceptable ones. This could greatly shorten processor development and validation times.

Acknowledgments

We would like to thank William van Loo and James Zeigler for many helpful discussions on masterslice chip design, and James Gilkinson for the benefit of his experience in remapping. Also, John Gerbi, Thomas Wanuga, and Alan Stern have made valuable contributions to the design and implementation of the experimental synthesis system.

References

1. M. A. Breuer, Ed., *Design Automation of Digital Systems*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972.
2. D. L. Dietmeyer, *Logic Design of Digital Systems*, Allyn and Bacon, Boston, 1978.
3. J. R. Duley, "DDL—A Digital Design Language," Ph.D. Thesis, University of Wisconsin, Madison, WI, 1968.
4. J. R. Duley and D. L. Dietmeyer, "Translation of a DDL Digital System Specification to Boolean Equations," *IEEE Trans. Computers* **C-18,** 305-320 (1969).
5. J. A. Darringer, "The Description, Simulation, and Automatic Implementation of Digital Computer Processors," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, PA, 1969.
6. T. D. Friedman and S. C. Yang, "Methods used in an Automatic Logic Design Generator (ALERT)," *IEEE Trans. Computers* **C-18,** 593-614 (1969).
7. T. D. Friedman and S. C. Yang, "Quality of Designs from an Automatic Logic Generator (ALERT)," *Proceedings of the Seventh Design Automation Conference*, San Francisco, CA, 1970, pp. 71-89.
8. H. Schorr, "Toward the Automatic Analysis and Synthesis of Digital Systems," Ph.D. Thesis, Princeton University, Princeton, NJ, 1962.
9. C. K. Mesztenyi, "Computer Design Language Simulation and Boolean Translation," *Technical Report 68-72*, Computer Science Department, University of Maryland, College Park, MD, 1968.
10. F. J. Hill and G. R. Peterson, *Digital Systems: Hardware Organization and Control*, John Wiley & Sons, Inc., New York, 1973.
11. M. Barbacci, "Automated Exploration of the Design Space for Register Transfer Systems," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, PA, 1973.
12. D. E. Thomas, "The Design and Analysis of an Automated Design Style Selector," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, PA, 1977.
13. E. A. Snow, "Automation of Module Set Independent Register-Transfer Level Design," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, PA, 1978.
14. L. J. Hafer and A. C. Parker, "Register-Transfer Level Digital Design Automation: The Allocation Process," *Proceedings of the Fifteenth Design Automation Conference*, Las Vegas, NV, 1978, pp. 213-219.
15. A. Parker, D. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Leive, and J. Kim, "The CMU Design Automation System—An Example of Automated Data Path Design," *Proceedings of the Sixteenth Design Automation Conference*, San Diego, CA, 1979, pp. 73-80.
16. S. Nakamura, S. Murai, C. Tanaka, M. Terai, H. Fujiwara, and K. Kinoshita, "LORES—Logic Reorganization System," *Proceedings of the Fifteenth Design Automation Conference*, Las Vegas, NV, 1978, pp. 250-260.
17. J. A. Darringer and W. H. Joyner, "A New Approach to Logic Synthesis," *Proceedings of the Seventeenth Design Automation Conference*, Minneapolis, MN, 1980, pp. 543-549.
18. J. A. Darringer, W. H. Joyner, L. Berman, and L. Trevillyan, "Experiments in Logic Synthesis," *Proceedings of the IEEE International Conference on Circuits and Computers ICCC80*, Port Chester, NY, 1980, pp. 234-237A.
19. F. E. Allen, J. L. Carter, J. Fabri, J. Ferrante, W. H. Harrison, P. G. Loewner, and L. H. Trevillyan, "The Experimental Compiling System," *IBM J. Res. Develop.* **24,** 695-715 (1980).
20. G. L. Parasch and R. L. Price, "Development and Application of a Designer Oriented Cyclic Simulator," *Proceedings of the Thirteenth Design Automation Conference*, San Francisco, CA, 1976, pp. 48-53.
21. Y. Chu, "An ALGOL-like Computer Design Language," *Commun. ACM* **8,** 607-615 (1965).
22. C. L. Berman, "On Logic Comparison," *Proceedings of the Eighteenth Design Automation Conference*, Nashville, TN, 1981 (to appear). Also *Research Report RC5342*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1980.

Received August 22, 1980; revised January 15, 1981

The authors are located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.