LSS: A system for production logic synthesis

by John A. Darringer
Daniel Brand
John V. Gerbi
William H. Joyner, Jr.
Louise Trevillyan

For some time we have been exploring methods of transforming functional specifications into hardware implementations that are suitable for production. The complexity of this task and the potential value have continued to grow with the increasing complexity of processor design and the mounting pressure to shorten machine design times. This paper describes the evolution of the Logic Synthesis System from an experimental tool to a production system for the synthesis of masterslice chip implementations. The system was used by one project in IBM Poughkeepsie to produce 90 percent of its more than one hundred chip parts. The primary reasons for this success are the use of local transformations to simplify logic representations at several levels of abstraction, and a highly cooperative effort between logic designers and synthesis system designers to understand the logic design process practiced in Poughkeepsie and to incorporate this knowledge into the synthesis system.

Introduction

As processor complexity increases and as denser, higher-performance technologies are developed, the job of the logic designer becomes more difficult. It is becoming increasingly important to provide automated tools to improve designer productivity. Successes to date have been in the area of physical design, where layout, wiring, and timing analysis have been largely automated. The next important area of design automation to improve designer productivity is to provide a tool which can generate logic implementations in the desired technology from a designer's functional specification.

There has been much research on this problem of logic synthesis, and the strategies employed fall into three broad classifications. One approach is to exploit methods of minimizing logic in a two-level representation [1, 2]. Unfortunately, since these algorithms search for minimal implementations, they require time exponential in the number of circuits and cannot be used on most actual designs. In addition, it is not always clear how to transform a minimum but idealized two-level implementation into an implementation n-level version that is realizable in an actual technology, although work continues on this problem [3].

A second approach is to view the problem as one of assembling large macros. In these design systems, the data flow of the machine is generated in terms of predesigned or generated macros, such as multiplexors and ALUs. The control logic is usually implemented by PLAs, Weinberger arrays, or ROMs with microcode. Most of the current silicon compiler work falls into this category [4, 5].

A third strategy is to use transformations to convert a high-level specification into technology-specific hardware in a series of small steps. In these systems, the logic is viewed as a graph of functional components, and transformations are applied to this graph in an attempt to improve the implementation.

These categories are not, of course, mutually exclusive. For example, a system could use the macro method for data flow and the minimization or transformation approach for control flow. A more detailed discussion of these approaches appears in [6, 7].

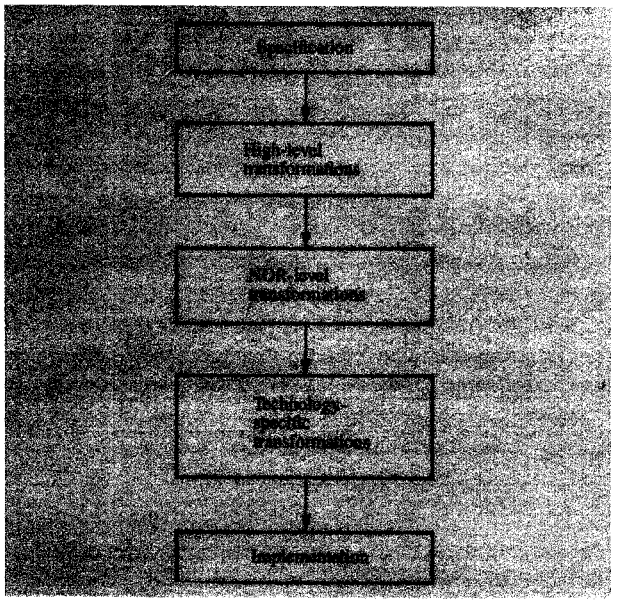

Figure 1

Three levels of transformation.

An important characteristic of synthesis systems is the level of the input specification. This ranges from low-level specifications that are close to the hardware to be implemented, through register-transfer-level input that assumes a synchronous model with all memory elements defined, to very-high-level descriptions that make no structural decisions about the implementation. The level of the language has a direct effect on the difficulty of implementing a synthesis system and on the productivity gains that can be expected of such a system. If a low-level language is chosen, it is much easier to generate the logic, but the system becomes more like a logic entry system. At the other end of the continuum, the very-high-level description yields the greatest flexibility and productivity improvement, but the job of actually generating production-quality implementations becomes extremely difficult.

For the logic synthesis system discussed in this paper, LSS, we accept input at the register-transfer level and improve the logic using local transformations which are applied at several levels of abstraction. We view the task of synthesis as one of producing feasible, not necessarily optimal, networks of technology-specific boxes that satisfy a large number of constraints. In this context, we must meet specified requirements of a technology, take advantage of features provided by the technology, and produce logic with satisfactory gate counts and path lengths.

In this paper, we give an overview of LSS and describe our experience in applying this approach to an actual technology in a production environment.

An overview of the Logic Synthesis System

Our approach is to transform logic from the high-level specification into a production-quality implementation through a sequence of local transformations. **Figure 1** shows the different levels of abstraction at which the transformations are applied. The register-transfer language is translated into a network of boxes connected by signals. Initially, this network consists of ANDs, ORs, NOTs, memory elements, and larger constructs such as decoders and adders, which are replaced eventually by more primitive implementations. The primitives change from only ANDs, ORs, and NOTs, to NANDs or NORs, and finally to technology-specific elements, by the application of sequences of simplification transformations at each level. While the transformations at the higher level are relatively independent of the eventual technology, it is the job of the technology-specific transformations to take advantage of hardware features such as "dual-rail" primitives with both output phases available, and to ensure that the implementation obeys all technology and methodology constraints.

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. Limiting the approach to simple local transformations both assures us that we will be able to handle large designs without paying a large penalty in computer time, and enhances our ability to maintain, expand, and understand the functions of the system.

As an example of how local transformations work, consider the logic network in **Figure 2(a)**. First, a transformation would examine boxes Y and Z and notice that the connection of signal e at box Y is not necessary and delete it, changing the network to that shown in **Figure 2(b)**. Another transformation would then find the double inverters X and Y. It would remove them and reroute signal a to box Z, leaving the final logic shown in **Figure 2(c)**.

In addition to these pattern-matching transformations, there are programs that propagate constants 0 and 1 through boxes, and that merge boxes performing equivalent functions. All transformations use a set of access functions to manipulate a graph composed of boxes interconnected by signals. These access functions hide the details of the underlying data structure from the transformation writer and allow powerful conversions to be developed very quickly. The technology-independent portions of LSS were implemented as part of IBM's Engineering Design System [8]. More detail about the structure of LSS and the specific transformations has been given in [7].

Initial synthesis experiments

The first experiments with the logic synthesis system were attempts to produce implementations for chips from the IBM 3081 processor. The chips were 704-gate TTL

masterslices, and the existence of the engineers' implementations permitted comparison of designs and a study of the differences between manual designs and those produced automatically. The first experiment resulted in a synthesized implementation that was very similar to the manual one, and with one percent fewer cells and connections. In the second experiment the same sequence of transformations was applied to a more complex chip, resulting in an implementation with fifteen percent more cells and twenty percent more connections than the manual implementation.

Experiments with ECL logic

The experiments with the existing 3081 chips were successful enough that there was interest in attempting synthesis for other technologies such as an ECL masterslice then in use. Here the basic primitive was a single-cell, four-input NOR with positive and negative outputs. Other functions in the technology included a two-cell, eight-input NOR, several types of latches, drivers, register-driver combinations, and a few complex functions. In addition, there is the ability to wire together or "DOT" emitter outputs to get a zero-cell OR function. The synthesis transformations and their application sequence were adapted to this technology. The three levels of implementation were maintained. The AND/OR transformations remained unchanged; certain transformations were used more heavily than in previous experiments.

Because ECL is a NOR-based technology, the NAND level became the NOR level. This required a new transformation to translate the AND/OR primitives into NORs, and a set of NOR simplification transformations. The original NOR transformations were identical to the NAND programs, with the NANDs converted to NORs. We later realized that savings at the NOR level did not necessarily translate into savings at the hardware level. As an example of this, saving a one-input NOR at the cost of increasing connections or fan-in does not generally produce a better hardware implementation when the target technology is dual-rail and distinct inverters are not required. The technology-independent transformations had to be re-evaluated and their constraints adjusted based on the improved ideas of "goodness." New technology-specific transformations also had to be implemented to take advantage of the features of the ECL technology, such as dual-rail outputs.

Once the transformations and a specific application sequence, or scenario, were developed, several experiments were run. First, five chips that had been manually designed were synthesized from their register-transfer-level specifications. In each case, the synthesized chip met all requirements and was within ten percent of the manually designed chip in cell count and logic levels.

In additional experiments we transformed six low-level, technology-specific designs from the 3081 TTL technology into the ECL technology. In this case, the input was expressed in terms of the 3081 technology primitives rather than as register-transfer specifications. There were no manual designs in ECL, but the results were evaluated by logic designers and were pronounced to be acceptable. More details on this remapping work are described in [9]. Technology remapping presents many problems that are interestingly different from those of pure synthesis and the paper on a technology mapping system [10] describes these in detail.

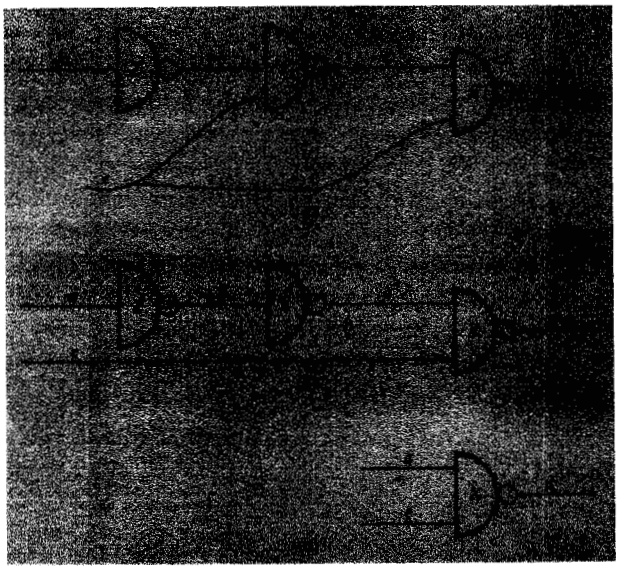

Figure 2

Example of local transformation application.

Large processor design environment

Following the successful demonstration of synthesis for ECL logic, the decision was made to use LSS in production in Poughkeepsie. This would require extensions to the prototype system and its integration into the existing tools of the Poughkeepsie design environment.

Figure 3 shows the design methodology used by the processor groups in Poughkeepsie [11]. The machine is first described at the register-transfer level in the language BDL/CS [12]. This specification declares all chip input and output signals and all on-chip memory elements. The model is assumed to be synchronous: For each implicit clock tick, outputs and new register values are defined in terms of inputs and old register values. The BDL/CS is simulated to ensure that the desired function has been specified. The machine is then implemented at the technology level in terms of hardware primitives, called *books*, that are provided by the particular masterslice technology used. This

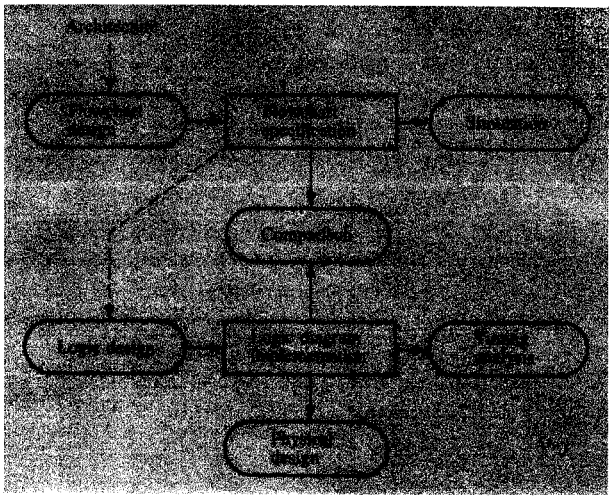

Figure 3

Large processor design methodology.

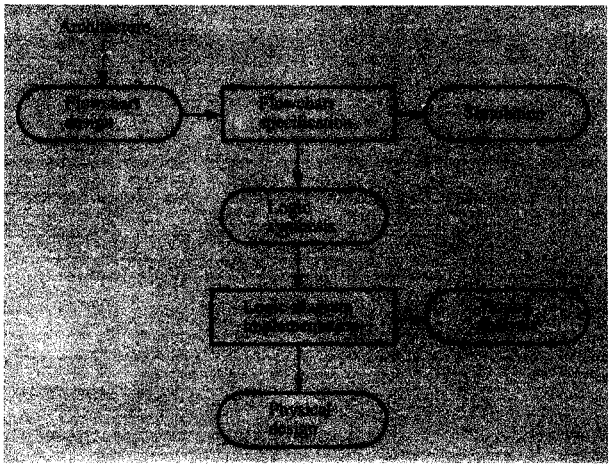

Figure 4

Automatic logic synthesis.

implementation is described in the BDL/S language, which gives interconnections among the various books in the design. The logic designer must implement the specified function, follow all technology and design rules, ensure that all path lengths meet the specified performance, ensure that the implementation is testable, and finally document the implementation with logic diagrams. To verify that the high- and low-level specifications agree, a static analysis system, SAS [13], is used to search for input patterns that will cause the two models to produce different outputs. Once the two versions of the design are equivalent, the BDL/S version is processed by tools that check technology constraints, provide information for higher levels of packaging, perform timing analysis, and finally produce the placement and wiring of the actual masterslice chip. In addition, since these are large processors with many chip part numbers, there is a chip maintenance system to keep track of the status and release level of each of the chips.

Our goal was to replace the manual implementation process, reading the same BDL/CS that was simulated and producing BDL/S acceptable to all of the other tools in the methodology. The new methodology is shown in **Figure 4**.

Production use of synthesis

Whereas the experiments with the prototype system had shown that we could automatically generate functionally correct logic that matched manual designs in gate count and that passed physical design checks, we knew that we were just beginning to understand the details of high-performance logic design. There were many more constraints dictated by the performance requirements, the technology, and the design methodology used.

- *Integration into design methodology*

In order to be usable at all, a completely automatic method for running synthesis had to be devised and integrated with the other tools. LSS was designed as an interactive system, to facilitate experimentation with new transformations and their sequencing. While this is useful in a research environment, it is unrealistic to expect the production user to operate the system this way. We therefore devised standard scenarios and a menu-driven system to aid the engineer in selecting a scenario suitable for his needs.

It is the job of this menu system not only to select the synthesis scenario, but to provide a link to the chip development system which keeps track of all of the interface information about chips. The menu system was used to extract the declarations of the chip I/Os and memory elements from a master list that was kept for the entire machine, to combine it with the specification of the function to be synthesized, and to manage bookkeeping information about the chip, such as date and time of the run, level of the system used, and problems encountered.

The major interface between LSS and other tools is BDL/S. Some information required by other tools was missing from the synthesized BDL/S, but it could be supplied by existing programs. However, BDL/S is more than a language to be read by other programs; it is also used to generate *automated logic diagrams*, or ALDs, multi-page graphical representations of the implementation. Although it is not important for the function of the logic to produce

good diagrams, we were expecting designers to use the diagrams and to evaluate the synthesized results, so the readability of the generated diagrams had to be improved. Since readability is a highly subjective measurement, this is a task that can never be entirely successful, and it was, in fact, a constant concern throughout the project. Nevertheless, we wrote programs and added heuristics to try to minimize page crossings, to get the logic to flow from beginning to end, and to group related items on diagram pages in ways that were familiar to the engineers.

The only existing tool that needed additional information from synthesis was the static analysis system. In order to verify the equivalence between the input to and output from synthesis, it needed to know the correspondence between the descriptive names used by the designer in the BDL/CS and the formalized names used in BDL/S. The correspondence was important not only to SAS, but to the designers who were evaluating the logic and wanted to have some explicit relationship between their inputs and the synthesized outputs. In order to maintain the BDL/CS names, the transformations were changed to be more careful about preserving useful names, and a file showing their correspondence was produced.

• *Production experience*

The first designers began creating BDL/CS specifications and synthesizing logic using the base system that had been used in the experiments with a skeletal version of the menu system described above. Engineers from several groups used the system, analyzed the generated logic, and reported unacceptable results to us. This was an arduous task which should not be undervalued, because it was only with their cooperation, hard work, and explanations that it was possible to turn a promising experimental tool into a usable production system.

Up to this point, only eleven examples in this technology had been processed by LSS. The system was now expected to cope with hundreds of chip part numbers, and its results were being evaluated with respect to more stringent constraints. As was to be expected, many problems with the system emerged and we began a period of rapid learning and improvement.

The difficulties encountered fell into the following four categories:

1. Testability and the related problems of excessive connections and cells.
2. Path lengths. Due to the clocking method, it was possible to have both paths that were too long and paths that were too short.
3. Incomplete support of the technology. The book set for the technology was very rich, and it was important for good cell count, reasonable power consumption, and acceptable path lengths to use it aggressively.
4. Inadequate chip interface information. Synthesis occurred at the chip level and no information about the external environment of the logic was available to it.

In some sense, these are not separate problems. For example, long paths could be present because of failure to use the appropriate book or because extra connections caused us to build a fan-in tree which added levels of logic. In addition, the lack of appropriate interface information made it impossible to identify the critical paths. In many cases, it was necessary to correct all of these problems simultaneously and at all levels of the synthesis scenario.

• *Testability and excessive connections*

It was important to solve the problem of excessive connections and redundancies. A connection of a signal is redundant if the signal can be replaced by a constant without changing the function of the logic. The question of testability is related to redundancy. A connection is testable if changing the value of the signal at that connection causes a change in the value of at least one output or latch in the logic. Connections that are redundant are therefore not testable. Not only were some chips failing to meet the required testability coverage (98.5 percent), but the extra connections caused cell counts to exceed those expected by the designers. These problems were attacked in several ways.

Figure 5 illustrates the NAND/NOR transformations that were described in [7]. Two of these, NTR4 and NTR7, were written to remove local redundancy. They look at two-level patterns for a reconvergent signal, and remove the redundancy by disconnecting the reconvergent signal.

Also, an existing NOR transformation, NTR6, which performed local factoring to reduce connections did not seem to apply as often as it should. From Fig. 5, it appears to replace two gates with three. But in a dual-rail technology, no inverter is required and two gates are still used. When the fan-in limits of the technology are considered, the second implementation could require fewer cells than the first, since fan-in is lowered. When the constraints were adjusted at the NOR level to take this into account, the transformation applied more frequently, which resulted in fewer cells, fewer connections, and often less redundancy.

Certain chips, however, continued to have redundancy problems and it was clear that more global strategies were needed. A factoring program was written to collect global information about all high-fan-in boxes or high-fan-out signals and apply heuristics to decide on the best groups of signals to factor. This program improved our results but problems remained.

The most significant progress in solving this problem was REDUND, a nonlocal redundancy detection program [14]. REDUND uses a method similar to those used for test generation to find connections which are not testable for stuck at 0 or 1. It forms the conditions for testing the

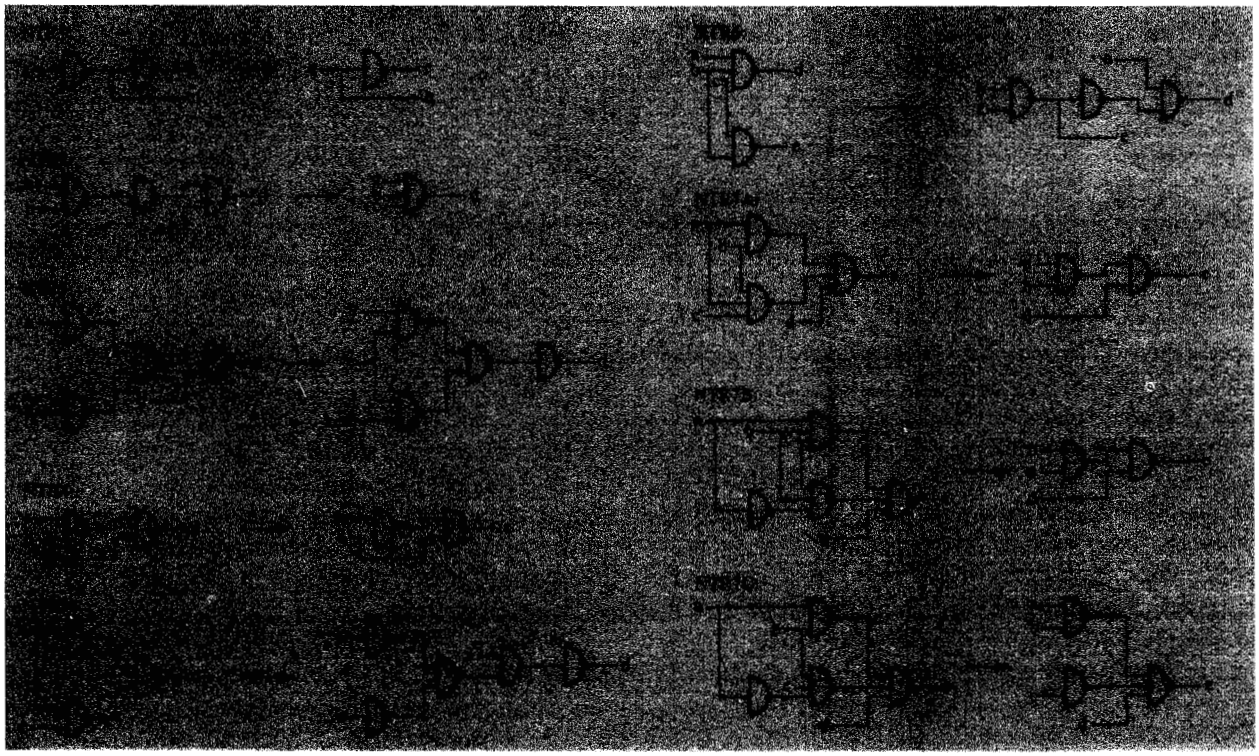

Figure 5

Example NOR/NAND transformations.

corresponding fault and checks for contradictions; when one is found, it replaces the connection with 0 or 1, as appropriate. The general constant propagation program is then used to simplify the logic. Although it is not guaranteed to detect all redundancies, REDUND has proven thorough enough to allow all chips encountered to date to meet the 98.5-percent testability requirement when allowed to run to completion. Since it may require a relatively large amount of CPU time, REDUND was initially only run on chips with known testability problems. In all but pathological cases, however, it has proved to be a reasonable performer, and it is now used on all chips.

- *Path lengths*

For a chip implementation to be usable, the path lengths must be acceptable. The technology provides shift register latches (SRLs), which consist of separate latch and trigger storage elements, each with a different clock. The clocks are overlapped in such a way that the latch inputs remain enabled for a time after new data have been launched from trigger outputs. If register-to-register paths are too short, new data can be made available a cycle too soon. If the register-to-register paths are too long, data are not available when needed.

The technology provides delay books, or *pads*, that can be used to lengthen short paths. Care must be taken in using delay books since they use valuable cells and can create a long path when inserted. The first pad-inserting transformation determined path lengths in levels of logic and added pads to short paths in such a way that no long paths were created. It was sometimes necessary to split a path into a long and a short component so the short could be padded. This implementation of short-path correction was inadequate because too many pads were added. There were several attempts to implement pad-sharing and pad-optimization programs, but we were not totally successful in correcting short paths.

The first attempt to correct long-path problems was done at a technology-independent level. It was observed that some of the NOR transformations tended to save cells by increasing the depth of logic, while others tended to increase cells by decreasing depth. By manipulating the application and sequences of such transformations, three scenarios were developed: one which produced the smallest cell count, a second which produced the smallest logic depth, and a third which was a compromise between the two. The defect of this multi-scenario strategy is that a typical chip has a mixture of long and acceptable paths and there was no reasonable way

of using the scenarios to get minimum cells on acceptable paths and minimum depth on long paths. Since we were synthesizing a single chip without knowledge of its environment, it was not always possible to identify the critical paths.

This resulted in modifications to the BDL/CS declarations, to allow off-chip delays for primary inputs and outputs to be specified in terms of levels of logic. The cycle time was also given in logic levels, and a unit delay model of logic was used. Programs at the NOR level were used to decrease the levels of NORs to within the given constraints, at the expense of adding more NOR boxes to the implementation.

However, due to the presence of dual-rail outputs and the availability of complex books in the technology, it was difficult to predict the depth of logic at the hardware level from the depth at the NOR level. It was often the case that an apparent shortening of a path at the NOR level had no effect at the hardware level except to add cells. This led to path-directed transformations at the hardware level which forced DOTs by duplicating logic and which reordered fan-in and fan-out trees with path length in mind.

None of these approaches to timing solved all of the problems. In the final system, some manual editing of the synthesized logic was necessary in order to correct short paths efficiently.

- *Enhanced support for the technology*

As mentioned earlier, it was necessary to make effective use of all the primitives provided by a technology to produce acceptable logic. One of the important items was the introduction of *emitter dots*. In this technology, emitter outputs can be wired or dotted together to perform an OR function. This is faster than a book OR and uses no cells. It is not as general as the OR book, since the DOT inputs cannot have other fan-out and cannot have sources such as latches or primary inputs, which have hidden fan-out.

The basic tool for identifying possible DOTs is a program that runs at the NOR level. It is similar to the factoring program in that it gathers information about all of the logic and then picks out candidates for DOTs from combinations of signals which have identical sink boxes. After this is done, it is up to the technology-dependent transformations to decide whether the candidate should actually be made into a DOT.

Care must also be taken in the introduction of DOTs. Introduction of DOTs puts constraints on the transformations subsequently applied, because they cannot add fan-out to dotted signals, create cascaded DOTs, or replicate DOTs to fix fan-out violations. DOTs can also be used to lower fan-in of NORs, converting them from two-cell books to one-cell books.

Other technology-specific transformations look for opportunities to use special books with complex functions to save space, time, or power. The ECL masterslice technology contains many special-function books that must be effectively utilized to approach the quality of a manual design. These books produce product-of-sums, XOR, XNOR, and other complex functions.

A technology-dependent transformation was written to find multi-level patterns of NORs that could be replaced with these special books. This program, initially fairly small when supporting only the product-of-sum books, required special extensions for each additional type and eventually became quite large. The XOR function was particularly difficult to detect, as a great number of structural permutations produce these $\neg AB$ and $A \neg B$ functions. Designers also had divergent opinions on when particular types should be used, and on the usage of some output pins. The conclusion was that this approach was moderately successful in its results, but cumbersome to implement.

Occasionally we over-used a feature provided by the technology. For example, it is possible to use the output of an off-chip driver within the chip. We at first assumed that these outputs were free and used them freely. Later we learned that the loading of these pins slowed the external outputs somewhat, made physical design more difficult, and could introduce timing problems because these outputs were slower than equivalent internal NORs. It was necessary to take these disadvantages into consideration and reformulate the constraints on the program that used such outputs.

Technology considerations also forced a reexamination of the order of application of transformations. For example, common-term elimination, unless restricted, cannot be done after fan-out correction, because it can introduce fan-out violations. Other interactions between transformations turned out to be more subtle. For example, one technology-specific transformation links the registers together in a scan ring. This appears to be an operation separable from the rest of the synthesis, but care must be taken to avoid creating fan-out violations. Similarly, adding clock signals to the multiple inputs of a register can cause a fan-in violation.

One area where synthesis could not fully use the technology was in the area of off-chip drivers and receivers. The technology provides several varieties of these, but the one to be chosen depends on information not available to synthesis, such as the amount and type of fan-out or wire lengths on the higher-level packaging. The declaration of primary inputs and outputs was expanded to allow the designer to provide this information. The technology-dependent transformations then used this information to generate the correct book types.

- *Chip interface information*

In addition to the cases mentioned earlier, there were two other situations that required information about the external environment of the chip being synthesized. When resynthesizing a chip after its BDL/S has been used in the

design of a larger module, it is necessary to ensure that the I/Os correspond to the same physical pins as they did in the first run. Since synthesis had no memory of what it had done before, it was necessary to feed back this information for subsequent runs. Also, information about register clocking has to be associated with the latch declarations. This is necessary in order to distribute the clocks correctly and to compute accurately required times in path correction transformations. Such information is kept in the chip library system, and it is extracted by the menu system for each run.

Production synthesis results

Fortunately we were able to solve most of the problems that arose in the refinement of LSS for production synthesis. The system was useful enough that many designers tried it and shared their experiences with us. Most users found that they could complete a first-pass design much more quickly than with manual design methods. In addition, they could perform timing analysis earlier, find their major problems, modify their specifications, and quickly generate a new implementation. This freedom to make sweeping changes allows a new approach to solving timing problems: They can be corrected by high-level changes rather than low-level path tuning. One project was able to use LSS for 90 percent of their more than 100 chip designs. This cut their initial design time in half.

Other projects were not as successful. They had more difficulty meeting their timing constraints and cell count limits. Interestingly, many of the specifications that LSS finds difficult are for highly regular designs, such as shifters and arithmetic units. There are chips that designers handle very well and seem to require global planning. In other cases the designer was able to produce a more efficient implementation because he could take advantage of information not available to the synthesis system, such as "don't-care" conditions. We are optimistic that progress will continue to be made and that eventually LSS will be able to produce acceptable implementations for all but the most pathological specifications.

Summary and future work

This paper has described the development of a system that is capable of automatically transforming functional specifications into production-quality masterslice implementations. The system has been used to produce 90 percent of the parts for a large (over 100 chips) high-performance project in Poughkeepsie. The primary reasons for this success are the use of local transformations to simplify logic representations at several levels of abstraction, and a highly cooperative effort between logic designers and synthesis system designers to understand the logic design process practiced in Poughkeepsie and to incorporate this knowledge into the transformations.

The fact that almost all of the transformations are local and have linear run times allows us to apply these techniques to much larger chips. Further, the structuring of the synthesis process as a sequence of simple transformations provides a great deal of flexibility in dealing with new and different technologies.

The experiences discussed above have pointed out some deficiencies and limitations with the current LSS. We are continuing to explore new methods for improving the ability to meet path length constraints and exploit complex books. Further, now that a path has been demonstrated from a register-transfer-level description to production-quality hardware, there is much more interest in higher-level synthesis transformations.

Acknowledgments

Throughout this paper we have stressed the importance of the participation of many experienced designers in the formulation of synthesis techniques. In addition, we would like to acknowledge the valuable contributions of Len Berman and Al Kashner.

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. R. Brayton and C. McMullen, "The Decomposition and Factorization of Boolean Functions," *Proceedings, International Symposium on Circuits and Systems*, April 1982, pp. 58–63.
4. D. Johannsen, "Bristle Blocks: A Silicon Compiler," *Proceedings of the 16th Design Automation Conference*, June 1979, pp. 310–313.
5. J. R. Southard, "MacPitts: An Approach to Silicon Compilation," *COMPUTER* **16,** No. 12, 74–82 (December 1983).
6. D. E. Thomas, C. Y. Hitchcock III, T. J. Kowalski, J. V. Rajan, and R. Walker, "Automatic Data Path Synthesis," *COMPUTER* **16,** No. 12, 59–73 (December 1983).
7. John A. Darringer, William H. Joyner, Jr., C. Leonard Berman, and Louise Trevillyan, "Logic Synthesis Through Local Transformations," *IBM J. Res. Develop.* **25,** No. 4, 272–280 (July 1981).
8. J. B. Bendas, "Design through Transformation," *Proceedings of the 20th Design Automation Conference*, June 1983, pp. 253–256.
9. 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*, Port Chester, NY, 1980, pp. 234–237.
10. J. L. Gilkinson, S. D. Lewis, B. B. Winter, and A. Hekmatpour, "Automated Technology Mapping," *IBM J. Res. Develop.* **28,** No. 5, 546–556 (1984, this issue).
11. R. N. Gustafson and F. J. Sparacio, "IBM 3081 Processor Unit: Design Considerations and Design Process," *IBM J. Res. Develop.* **26,** No. 1, 12–21 (January 1982).
12. 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.
13. G. L. Smith, R. J. Bahnsen, and H. Halliwell, "Boolean Comparison of Hardware and Flowcharts," *IBM J. Res. Develop.* **26,** No. 1, 106–116 (January 1982).
14. Daniel Brand, "Redundancy and Don't Cares in Logic Synthesis," *IEEE Trans. Computers* **C-32,** No. 10, 947–952 (October 1983).

Received March 10, 1984; revised April 30, 1984

Daniel Brand *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Brand has been a research staff member at the IBM Thomas J. Watson Research Center since 1976. He is currently working on a project which is investigating a system for automatically synthesizing technology-specific chip implementations from register-transfer-level descriptions. His previous work was in the areas of program verification and automatic theorem proving. Dr. Brand received the Ph.D. from the University of Toronto, Canada, and is a member of the Association for Computing Machinery.

John A. Darringer *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Darringer received his Ph.D. from Carnegie-Mellon University, Pittsburgh, Pennsylvania, in 1969. From 1969 to 1972, he was a design automation consultant to the large-computer division of Philips in Holland. In 1972 he joined IBM at the Thomas J. Watson Research Center, where he worked on program specification and verification. He started the Logic Synthesis project in 1979 to take a new look at the problem of automatic hardware generation. Dr. Darringer is a member of the Association for Computing Machinery, Eta Kappa Nu, the Institute of Electrical and Electronics Engineers, and Tau Beta Pi.

John V. Gerbi *IBM Data Systems Division, P.O. Box 390, Poughkeepsie, New York 12602.* Mr. Gerbi is a senior programmer in design tools development in the Poughkeepsie laboratory. He received the B.S. in electrical engineering from Union College, Schenectady, New York, and is pursuing graduate studies at Syracuse University in computer and information science. He has, since joining IBM in 1965, been involved with communications and database systems and, more recently, with mechanical and logical design tools. Mr. Gerbi has received DSD Division and IBM Outstanding Technical Achievement Awards for his work in automatic logic synthesis.

William H. Joyner, Jr. *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Joyner has been a research staff member at the IBM Thomas J. Watson Research Center since 1973. He is currently manager of a project which is investigating a system for automatically synthesizing technology-specific chip implementations from register-transfer-level descriptions. His previous work was in the areas of program verification and automatic theorem proving. He received the B.S. from the University of Virginia, Charlottesville, and the Ph.D. in applied mathematics from Harvard University, Cambridge, Massachusetts. Dr. Joyner is a member of the Association for Computing Machinery and Tau Beta Pi.

Louise Trevillyan *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Ms. Trevillyan has been a research staff member at the IBM Thomas J. Watson Research Center since 1978. Her current work involves the investigation of a system for automatically synthesizing technology-specific chip implementations from register-transfer-level descriptions. She joined IBM at Data Processing Services, San Francisco, California, in 1974, moving to the IBM Thomas J. Watson Research Center in 1977. Ms. Trevillyan received a B.A. in mathematics in 1968, and M.A. degrees in mathematics and computer sciences in 1970, all from the University of Michigan, Ann Arbor.