THE VLSI DESIGN AUTOMATION ASSISTANT: A SYNTHESIS EXPERT

Thaddeus J. Kowalski *is a member of technical staff in the Computer-Aided Information Systems Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey. He is currently examining how expert VLSI designers choose a hardware architecture and whether a knowledge-based expert system, KBES, can mimic their style. His interests also include artificial intelligence, operating systems, programming and text-processing environments, and real-time systems. Mr. Kowalski, who joined the company in 1978, has a B.S.E. from the University of Michigan and both an M.S.E.E. and a Ph.D. in electrical engineering from Carnegie-Mellon University.*

This paper describes an approach to very-large-scale-integration (VLSI) design synthesis that uses knowledge-based expert systems to proceed from an algorithmic description of a VLSI system to a list of technology-dependent registers, operators, data paths, and control signals. It outlines how the Design Automation Assistant (DAA) uses large amounts of expert knowledge to design an architecture with little backtracking. This paper takes a retrospective look at that codified knowledge base, examining what has been learned about VLSI design. Finally, the paper gives an overview of our current work in using bottom-up design information in the synthesis of integrated circuits.

Perspective

Recent advances in integrated-circuit fabrication technology have allowed larger and more complex designs to form complete systems[1] on single VLSI chips. (Panel 1 defines the acronyms used in this paper.) These chips use 1-μm to 5-μm features to achieve complexities equivalent to 100,000 to 250,000 transistors. This level of design complexity has created a combinatorial explosion of details that are a major limitation in realizing cost-effective, low-volume, special-purpose VLSI systems. To overcome this limitation, design tools and methodologies that are capable of automating more of the digital synthesis process must be built.

At AT&T Bell Laboratories, we have been developing just such synthesis tools.[2] These tools help the designer develop the algorithmic description of the system and interactively add the details required to produce a finished design. Our approach aids the designer by producing data paths and control sequences that implement the algorithmic system description within supplied constraints. Thus, the designer can consider many alternatives before selecting a final design. This structured approach can decrease the time it takes to design a chip, automatically provide multilevel documentation for the finished design, and create reliable and testable designs.

Panel 1. Acronyms in this Paper

ALU	arithmetic-logic unit
CPU	central-processing unit
I/O	input/output
ISPS	instruction set processor specification
KBES	knowledge-based expert system
MOS	metal-oxide semiconductor
MUX	multiplexer
OPS5	a knowledge-based expert system writing system
RISC	reduced instruction set computer
RT	register transfer
SCF3	a structured control flow processor
TTL	transistor-transistor logic
VLSI	very-large-scale integration

From a series of acquisition interviews[3] and an initial prototype system,[4] we designed a system that generates a technology-dependent list of hardware components—such as operators, registers, data paths, and control signals—from an algorithmic description and a list of design constraints. This system, called the Design Automation Assistant or DAA,[5] has been used to design an IBM System/370 computer. An IBM System/370 designer[6] evaluated the design favorably. We are currently pursuing the use of bottom-up design information to improve estimates of physical placement and wiring.[7]

This paper focuses on the synthesis, or allocation, of the implementation-design space as it advances from an algorithmic description of a VLSI system to a list of technology-dependent registers, operators, data paths, and control signals.

This synthesis task has inspired a variety of approaches that range from the most simplistic backtracking methods through the most complicated constraint propagation methods.[8-12] Owing to the complexity of design synthesis, simplistic backtracking schemes consume large amounts of CPU (central-processing unit) time, and the constraint propagation method is too cumbersome for large designs. Because of the combinatorial explosion of details and implicit dynamic constraints involved in choosing an implementation, these algorithmic solutions are not suitable. An alternate approach to design synthesis uses a large amount of design knowledge to eliminate backtracking. Whenever possible, the focus is on specific design details and constraints. Artificial intelligence researchers have called systems developed under this heuristic approach knowledge-based expert systems (KBESs).[13]

Conception

Generally, KBESs are developed in several stages. First, one codes "book knowledge" of the problem as a set of situation-action rules, and uses interviews with experts to fill in knowledge gaps and refine current knowledge. Then, many example problems are given to the KBES, and experts closely examine and validate the results. Often, errors are found through the examples, and new rules are added to the system to correct the error situations.

Why is this iterative process necessary? Often experts are unaware of exactly how they go about designing a chip or cannot articulate the procedure. Furthermore, the knowledge base is not an exact codification of the expert's knowledge. Instead, it represents what the knowledge engineer understood.

After we gathered current book knowledge about synthesis of the architectural design space,[8-10] we interviewed four designers of varied experience. One was a novice, two were moderately experienced, and one was an expert. At the start of each interview, which lasted about an hour, we determined the designer's background, including years of experience, logic families used, and designs created. Most of the time was spent discussing the design process, with some discussion of the DAA system. Our interview method was designed to give the interviewees as much freedom as possible to generate ideas; we emphasized such questions as "What do you do next?" and "Could you elaborate?"

The designers discussed the global picture, partitioning, selection, and allocation tasks. They began with a high-level overview of the hardware, which listed inputs and outputs to the outside world, the functions the hardware should provide, general constraints, and design feasibility with consideration of the target technology. They generally partitioned the global picture into smaller blocks and emphasized minimizing connections among

blocks, selecting blocks that operate as parallel or serial units, and grouping blocks by similarity of function. Partitions were chosen for allocation in a decreasing order of difficulty or degree of constraint. The designers reasoned that, if the most difficult part could be designed, the rest of the design was feasible.

Once selected for allocation, a partition was carried out either in parallel or in series. A parallel design made thinking of the control logic much simpler, while a serial design minimized the design area. The constraints of the parallel design were examined for size violations to determine the parts to be serialized by adding data paths, registers, and control logic to the initial parallel design. The constraints of the serial design were examined for speed violations to determine the parts to be reimplemented in parallel. If the designers saw that part of the new design was similar to part of a previous design, they used what they knew had worked in the past.

Within each partition, designers allocated clock phases, operators, registers, data paths, and control logic. The order was interesting because once registers and data paths were allocated, they were not changed. The control logic was changed because it was the hardest thing to think about and depended on a constant structure for the data path elements.

The designers described the iteration process as a step-by-step refinement. To meet a violated constraint, they looked for a technology change before making a design change. This could be as simple as finding a new chip in the TTL (transistor-transistor logic) data book, or as complicated as shrinking a design rule. Next, they would sacrifice functionality to meet a constraint. As one designer summed it up, "An engineer's training teaches him when constraints can be swept under the rug."

The relative importance of constraints depends on the application. The designers mentioned the constraints of speed, area, power, schedule, cost, drive capabilities, and bit width. Other design changes consisted of global improvements not recognized until the design neared completion. This suggests that the general choice of partitions and initial design-style selections approached optimum and that designers do not seem to use much backtracking in their designs.

Birth of the DAA System

Even though many details were missing, we had gathered enough book knowledge to put together a prototype version of the DAA system using Carnegie-Mellon University's OPS5[14] KBES writing system. While the DAA system was still far from perfect, it stimulated further sessions with expert designers to elicit more information.

Now, let us examine the flow of control in the prototype system and how the KBES approach formulates the problem.

The DAA starts with a data flow representation that it extracts from the algorithmic description. It is a good representation because it is easily manipulated by computer programs.[15] Also, the representation helps eliminate differences in the behavior descriptions from differences in writing styles.

The DAA produces a technology-dependent hardware network description composed of modules, ports, links, and symbolic microcode. The modules can be registers; operators; memories; and buses or multiplexers with input, output, and bidirectional ports. Links connect the ports, which are controlled by the symbolic microcode.

The DAA uses a set of four temporally ordered subtasks to complete the synthesis task.

1. The base-variable storage elements—constants, architectural registers, and memories with their input, output, and address registers—are allocated to hardware modules and ports.
2. A data-flow BEGIN/END block is picked, and the synthesis operation assigns minimum delay information to develop a parallel design.
3. All data-flow-operator outputs that are not bound to base-variable storage elements are mapped to register modules.

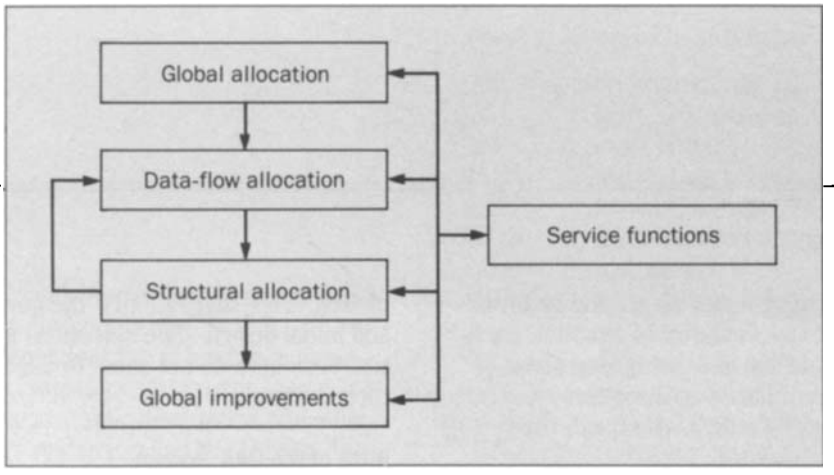

Figure 1. The DAA subtasks.

4. Each data-flow operator—with its inputs and outputs—is mapped to modules, ports, and links. In doing so, the DAA avoids multiple assignments of hardware links; it supplies multiplexers when necessary.

Steps 3 and 4 place the algorithmic description in a uniform notation for the expert analysis phase that follows.

The expert analysis subtask first removes registers from those data-flow outputs where the data-flow operator's sources are stable. The DAA combines operators, according to cost and partitioning information across the allocated design, to create ALUs (arithmetic-logic units). It also examines the possibility of sharing nonarchitectural registers. When possible, the DAA performs increment, decrement, and shift operations in existing registers. When appropriate, it places registers, memories, and ALUs on buses. Throughout this subtask, constraint violations require tradeoffs between the number of modules and the partitioning of control steps. The process is repeated for the next data-flow BEGIN/END block.

First Steps

The prototype DAA system had about 70 rules and could design an MOS Technology Incorporated MCS6502 microcomputer in about three hours of CPU time on a Digital Equipment Corporation VAX™ 11/750 minicomputer. We asked many expert designers at Intel Corporation and AT&T Bell Laboratories to critique the design by explaining what was wrong, why it was wrong, and how to fix it. At each knowledge acquisition interview, we gave the designer a drawing of the design with a sheet of clear plastic and pieces of cardboard over it. As the designer's critique proceeded, a piece of cardboard had to be lifted, the correction had to be written on plastic, and a new layer of plastic placed over the design. This provided a complete record of where the designer was focusing attention and what was corrected. The designers found this procedure compatible with their normal spatial mode of operation.

After each critique, we modified the DAA's rules, added new rules, and redesigned the MCS6502 microcomputer. As a result, the development DAA system grew to more than 300 rules and designed a much better MCS6502 microcomputer in about five hours of VAX 11/750 CPU time. In retrospect, clearly much of what we learned was common-sense design knowledge, the same things human designers learn through apprenticeship.

The DAA has undergone many improvements and produced many designs of the MCS6502 microcomputer. Although the final design was acceptable to our experts, it was not perfect. Further changes led to improvements, such as multiple buses of different widths. However, these changes did not affect the MCS6502 microcomputer, because it did not require multiple buses.

This brings up an interesting point about expert systems: They are never totally finished. Like human designers, the DAA becomes a better designer as its rule memory expands. Until all possible world knowledge about designing microprocessors has been codified in the DAA's rules, there will always be room for improvement in its designs.

The IBM System/370 Experiment

After the DAA successfully designed a MCS6502 microcomputer, we had to determine if the system had also acquired knowledge about processor design in general. So we designed an experiment to see if the DAA could design a processor that was substantially different and more com-

Table I. Rules by Function

Section	Rules	Firings
Service functions	88	4901
Global allocation	78	104
Data-flow allocation	55	1468
Structural allocation	47	1610
Global improvements	46	460
Total DAA	314	8543

plex than the MCS6502 microcomputer.

From the ISPS (instruction set processor specification) descriptions maintained at Carnegie-Mellon University, we chose one for the complete IBM System/370 processor. This description included memory management operations; channel controller I/O (input/output) instructions; and all instructions for the 370 processor, except the extended-precision floating point, characters under mask, edit and mark, and packed-decimal instructions.

The unmodified System/370 description, missing only a small percentage of the total 370, is more than ten times larger than that of the MCS6502 microcomputer, and had not been used to build the DAA. This choice offered important benefits:

- A single-chip design of the 370 had been made at IBM.
- Information was publicly available.
- Claud Davis, the design-team manager and a key designer at IBM, was willing to critique the design. During his more than 25 years at IBM, Davis has worked on designs and managed teams of designers for the higher performance 701, 702, 7074 MA 360/50, FAA, and 360/67 processors, and the μ370 microprocessor. His vast experience with these devices made his critique very valuable.

Thus, the experiment was a fair and convenient way to test the generality of the DAA's design knowledge.

Claud Davis compared his and the DAA's System/370 designs at IBM's laboratory in Poughkeepsie, New York. He summarized his comparison:[16]

"The 370 data-flow we reviewed exhibited the quality I would expect from one of our better designers. The level of detail was what we call second-level design. This encompasses all 'architected' registers, status latches, and sufficient working registers to implement the functions defined by the instruction set.

"The review included a test for 'architected' registers, data-path widths, latches for exceptional conditions, signs, and latches for temporary information in multicycle instructions.

"The assumptions for clocking and controls were examined and found to be consistent."

The complete transcript[17] is also available.

The DAA Knowledge

The DAA is implemented as a production system using the OPS5[14] KBES writing system. One merit of coding knowledge in a KBES is that we can easily quantify and qualify the knowledge. An important goal of our research has been to understand how VLSI designers choose computer implementations. The problem division that they use consists of general service functions, global implementation allocation, data-flow allocation, structural allocation, and global improvements to the implementation; see Figure 1. Table I provides a summary of these subtasks and the number of rule firings, or activations, for the complete design of the structured control flow processor,[18] SCF3. [The SCF3 is a RISC (reduced instruction set computer) architecture designed and studied at Carnegie-Mellon University.]

We now discuss these subtasks (which are implemented in the DAA) by functional sections, and describe two procedures that provide high-level floor-planning information.

The DAA Subtasks. The first functional section, *service functions*, is a set of service rules. These 88 rules:

- Control the order of subtasks throughout the

implementation.

- Minimize connectivity between modules.
- Perform simple bookkeeping to maintain and express the design.

Thus, they both control and are controlled by the other four functional sections.

The next functional section, *global allocation*, is a set of global allocation rules. These 78 rules:

- Allocate hardware that the DAA cannot optimize. This hardware consists of architectural components such as memories, architectural registers, and controllers.
- Provide default constraints and timing information where none is supplied.
- Initialize the bookkeeping information used by the service rules.

Figure 1 shows that this set of rules is activated only for the first partition allocated.

Once the DAA has allocated the architectural hardware, the next task is to partition the whole design into smaller blocks and select a partition. Within each partition, the DAA creates clock phases, operators, registers, data paths, and control logic in two subtasks: *data-flow allocation* and *structural allocation*. This allows the DAA to gather all the information about register usage in the data-flow allocation and then create registers and modules in the structural allocation.

The 55 data-flow allocation rules:

- Assign operators to control phases.
- Determine the minimum size needed to represent an operation.
- Allocate temporary registers and ALU modules. This creation is not hardware allocation, but a mapping of the data-flow representation into the structural and control representation to gather timing, connectivity, and functionality information.
- Allocate multiplexers.

After data-flow allocation, the 47 structural-allocation rules:

- Remove registers that are not required to maintain testability.

Panel 2. Sample ISPS Description

```
Example :=
Begin

** Storage. Declaration **

cpage\current.page<0:4>,
i\instruction<0:11>,
  pb\page.0.bit<> := i<4>,
  pa\page.address<0:6> := i<5:11>

**Address.Calculation**

Global eadd\effective.address<0:11> :=
  Begin
  Decode pb =>
    Begin
    0 := eadd ='00000 @ pa,
    1 := eadd = cpage @ pa
    End
  End
End
```

- Combine other registers.
- Use fan out from existing ALU modules.
- Create ALU modules.

Figure 1 shows that the data-flow and structural allocation rules are repeated for each of the partitions that the service-function rules created.

As a design nears completion, the DAA starts the *global improvement* function. It examines the design for components that are no longer needed or could be better shared. This cleanup includes removing unused ALU modules and registers, reducing multiplexer trees, combining ALU modules by fan-out, and allocating buses in high traffic areas. This function's set of 46 rules, like the global allocation set of rules described previously, is activated only once per design.

Estimators. As the DAA evolved from the prototype system, it became clear that some things could not be handled well in rules. Primarily, these were actions that required summing, counting, or checking for set membership (i.e., looking to see if an object is among a collection of objects).

These calculations, represented in OPS5 rules, were terribly inefficient and could be much better repre-

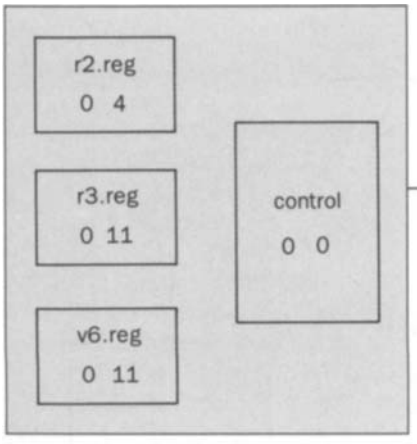

Figure 2. Design after global allocation.

Panel 3. Find Registers

IF: the most current active context is declared variable allocation
and there is a VT-body that is a carrier, or section list
and it is not an array
and it has a nonzero width
and the parent VT-body is a section list
THEN: create a register module
and create an output port
and create an input port

sented as algorithms in a language like C. So we used rules for the decision making connected with the results of these estimators, but moved the calculation into a C program. Through interviews with designers, these calculations evolved to focus attention on similarities of functions and connections in data-flow graphs and modules. They mimic the "back-of-the-envelope" floor plan on which designers base many of their decisions. These estimators—a partitioner and a cost function—have closely matched expert-designer performance in partitioning and cost-analysis experiments.

Partitioning. The first estimator is a partitioner that bridges the algorithmic to fabrication-dependent hardware-network levels. Unlike most current partitioners of digital hardware,[19-22] it does not require that the system's physical modules and their interconnections be specified. That is, in our terms, it uses only the abstract and imprecise information contained in the ISPS[23] description to predict how one may lay out a data path. Designers often refer to this layout guess as the *floor plan* of a chip. The floor-plan partitioner attempts to share hardware effectively and minimize the interconnections between partitions.[24]

Cost estimator. The second estimator is a hardware pricer that bridges the technology-independent to technology-dependent hardware-network levels. Unlike the partition estimator, it requires that the system be specified by technology-independent modules and their interconnections. The cost estimator provides information about what percentage of the required hardware for a new module already exists in another module. Thus, in our terms, it uses only the abstract and imprecise information contained in the structural and control description to predict how much it would cost to upgrade the functions and interconnects of an existing module to contain a new module. While the partition estimator gives a high-level floor plan, this estimator gives a much more local view, augmenting the high-level floor plan with more detailed information.

A Sample Design

To help readers better understand the relationship between the design steps, we examine a simple ISPS example description (Panel 2) throughout the implementation design process. This ISPS fragment, which is from the description of the Digital Equipment Corporation PDP®-8 computer, first defines the current page and the instruction carriers. It then labels specific fields of the instruction carrier as the page-zero bit and page-address carriers. The last part of the ISPS fragment decodes the page-zero bit of the instruction carrier and sets the effective-address carrier equal to the page-address carrier, concatenated with either zero or the current-page carrier.

To highlight various areas of knowledge in the DAA, this section also includes sample rules that are discussed and translated in the next section. Readers should realize that these translations must be taken with a grain of salt; rules do not stand by themselves, but are part of a larger network of rules connected by the working memory

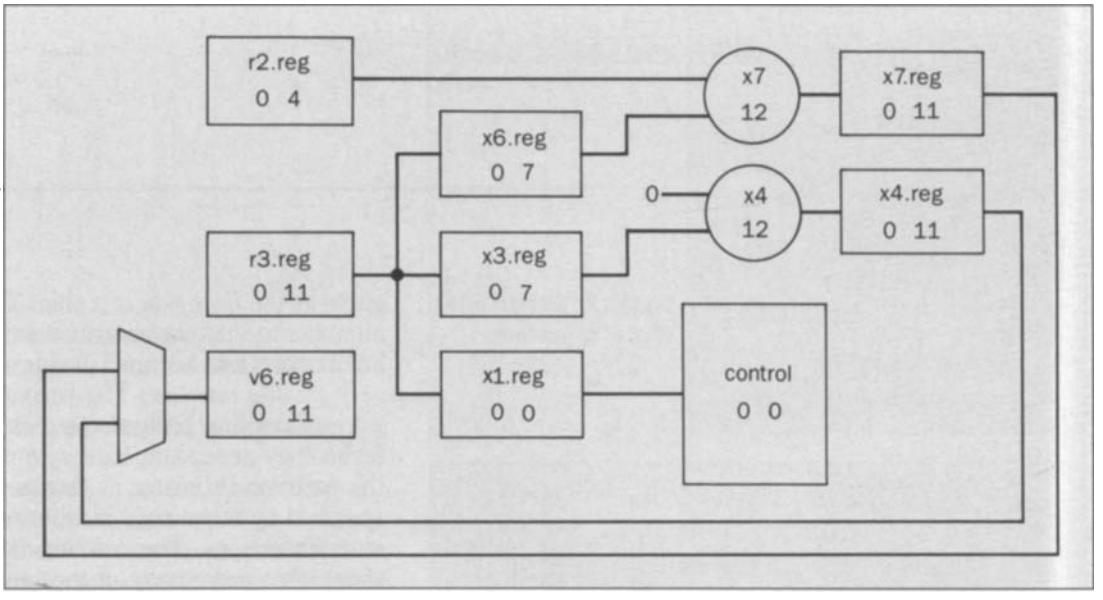

Figure 3. Design after data-flow allocation.

and the inference engine.

Sample Global Allocation. Figure 2 shows the design of the decoding loop of Panel 2 after global allocation has been done. Each symbol represent a module that has the bit width given as the bottom pair of numbers. The small rectangles are registers and the large rectangle is the controller. The registers—`r2.reg`, `r3.reg`, and `v6.reg`—are allocated because of the ISPS definitions for `cpage`, `i`, and `eadd`, respectively. The controller is allocated to provide read and write signals to the registers.

Although unseen in Figure 2, the technology database and constraints are initialized. The sample rule in Panel 3 is used during global allocation to allocate registers. It finds data-flow entities, called VT-bodies, that are not arrays and allocates a register module with input and output ports.

Sample Data-Flow Allocation. Figure 3 shows the design of the decoding loop of Panel 2 after data-flow allocation has been done. Each symbol represents a module. The circles are single-function wiring modules that bring together or concatenate two sets of signals, the trapezoid is a multiplexer that gates one of its two inputs to its output, the small rectangles are registers, the "0" (by `x4`) is a constant, the large rectangle is the controller, and each line represents a link between the modules. Where the links join with the modules, a port is defined.

The state of the design shows the creation of the temporary registers (`x1.reg`, `x3.reg`, `x4.reg`, `x6:reg`, and `x7.reg`), the concatenation modules (`x7` and `x4`), a multiplexer, and many connections. Along with each connection, the operator assignment to control steps and the data-flow references are specified, so that a control specification can be generated. The temporary registers are created to latch values for testing.

The concatenation modules are created—not assigned as register attributes—because they are wiring instructions and cost nothing to create. The multiplexer is created, because we need to store the output of either `x7` or `x4` in `v6.reg`. Finally, a constant has been shown as an input to `x4`.

Panel 4 provides an example of a rule to allocate temporary registers. This rule finds output values—called *outnodes*—from data-flow operators that are not associated with architectural registers, and creates temporary registers with input and output ports for them.

Sample Structural Allocation. Figure 4 shows the design of the decoding loop of Panel 2 after structural allocation has been done. The state of the design shows the removal of the temporary registers—`x1.reg`, `x3.reg`, `x4.reg`, `x6.reg`, and `x7.reg`—and many connections. For this simple design, the DAA found out that all the registers are stable and not needed to keep the design testable. Thus, they were not made permanent.

Also, the two wiring operations, `x4` and `x7`, cannot be combined because they do not have the same inputs. Had the operators not been wiring operators, they would have been examined using the partition and cost

Panel 4. Find Temporary Registers

IF: the most current active context is temporary variable allocation
and there is a produced value outnode
and the outnode is not associated with an architectural register
THEN: create a temporary register module
and create an output port
and create an input port

estimators described earlier and possibly combined.

Panel 5 provides an example of a rule to remove stable registers. If a temporary register has an input value but its output is not used anywhere, the register is found to be stable and is removed.

Sample Global Improvements. Figure 4 also shows the design of the decoding loop of Panel 2 after global improvements have been made. Because the initial ISPS was so simple, no changes were required.

Panel 6 provides an example of a rule to allocate bus structures. It finds modules that are connected to the same multiplexers and places them on a bus that is not transferring data during this time step.

Using Bottom-Up Information

Since the IBM System/370 experiment, we have been exploring ways to improve automated integrated circuit designs by using bottom-up analysis.[7] That is, to help evaluate and synthesize the design's RT-level structure, we use information about the physical characteristics of the cells from which a design will be built, along with their placement and interconnection. In this way, we hope to achieve the greater accuracy of a silicon compiler without losing the flexibility of a top-down design system.

The approach we have taken has two major elements:

- A new object-oriented database provides detailed physical and logical information about the cells available for use in the design.[25]
- The data operations that are specified in the behavior description are organized into clusters. These clusters have physical as well as logical significance, so one can infer certain properties of the layout and interconnections from the way the operations are clustered.

While the database gives size and timing information for the individual modules, clustering gives an idea of the layout and wiring so one can calculate the global properties of the chip more accurately. Each cluster represents a certain region of the chip that contains a set of modules and their interconnections. Figure 5 shows how clusters are organized hierarchically.

Clusters 1 and 2 are leaf clusters, each made up of a functional unit and storage interconnected by buses. Clusters 3 and 4 are nonterminal clusters that contain lower level clusters plus the wiring needed to tie them together.

The clusters are imprecise and do not perfectly reflect the final design, but they do make it possible to estimate the effects of layout and wiring. One can estimate the size of a leaf cluster from the size of its component functional units, registers, multiplexers, and buses. Also, one can estimate the size of a nonterminal from the sizes of its components and the number of wires interconnecting them. Finally, one can estimate the size and delay of an interconnection from the size of the clusters through which it runs. The clustering analysis can be extended to multiple-chip designs as well. At a certain level of the hierarchy, different clusters can represent different chips. Thus, the space, wiring, and delay estimators make it possible to evaluate different ways of partitioning functions onto chips.

Summary

We have shown how expert VLSI designers choose the implementation for an MOS microcomputer and how a knowledge-based expert-system, the DAA, can mimic their results. The KBES technique appears to be a promising approach to design synthesis.

Because the KBES approach provides a framework that allows incremental addition of common-sense modular design knowledge and queries about the knowledge during the design task, its use has speeded development of the DAA. The KBES framework has increased the final system's flexibility and reduced the execution time by replacing backtracking techniques with match techniques.

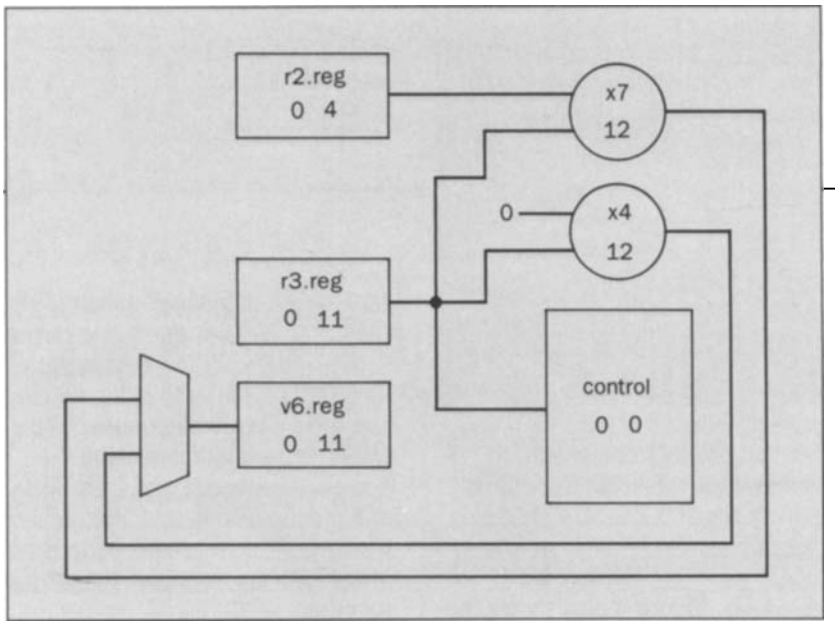

Figure 4. Design after structural allocation.

We have seen how the DAA, like a human designer, has become a better designer as its rule memory expands. The extraction, codification, and testing of the expert designers' knowledge has made possible a better understanding of VLSI design synthesis, while providing another KBES system for computer scientists and knowledge engineers to examine.

To explore the generality of design knowledge in the DAA, we have compared and contrasted an IBM System/370 processor—designed by an expert human designer, Claud Davis—against the design that the DAA produced. This is the first large, automatically generated design that has exhibited the quality expected from one of IBM's better designers. Furthermore, the design required 47 hours of CPU time, which—with some work—can be reduced by a factor of 12 to about 4 hours of CPU time. This clearly shows the dramatic improvement in CPU time for large designs obtained using methods that replace backtracking by match techniques.

We have also shown how designers partition the design task into four major subtasks. These subtasks involve global allocation of architectural modules and registers; local allocation of control steps, operators, registers, data paths, and control; global allocation of operators and registers; and global improvements to the design. Within these subtasks, a dominant goal is to partition the design into smaller, more manageable pieces.

While making the design more manageable, it is also important to retain a global view of the hardware. To accomplish this, designers use high-level floor plans that are filled in as the design proceeds. The floor-planning mechanism is composed of estimators for partitioning and pricing hardware. These estimators deal with connectivity and functionality of the hardware networks.

At the same time, the designers feel it is important to design testable synchronous designs. This constraint is even more important than making the least expensive connections or optimally sharing hardware. After testability, the most important consideration is how the design will lay out or how to minimize connectivity. Thus, creating a good design is not just minimizing components, but paying careful attention to testability and connectivity.

Panel 5. Fold No Output Register

IF: the most current active context is fold allocation
and there is a temporary register
and there is a link to the temporary register
and there is not a link from the temporary register
THEN: remove the temporary register
and remove its link

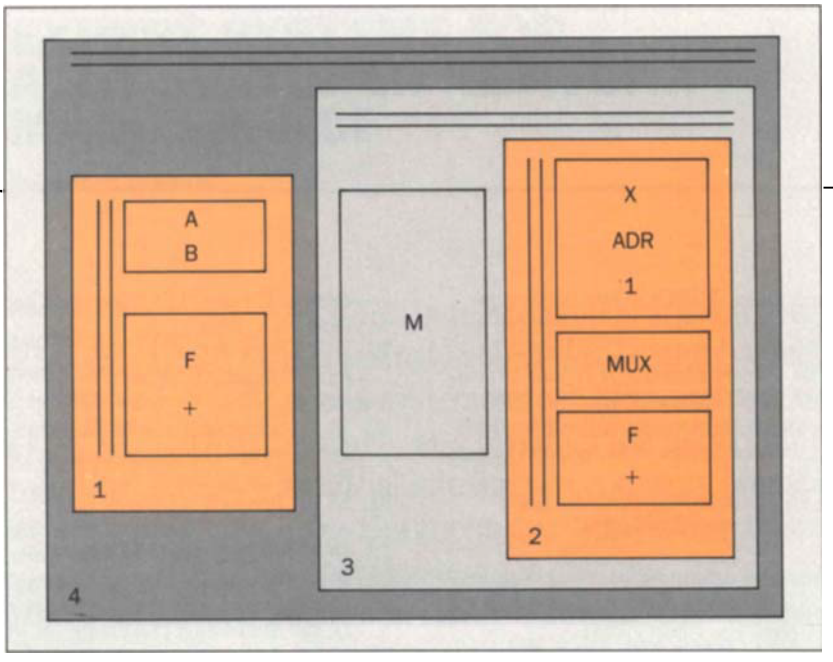

Figure 5. An example of clusters.

In retrospect, much of what we learned was common-sense design knowledge, the same things human designers learn through apprenticeship. This learning process is not complete, nor will it ever be complete. Like human designers, the DAA becomes a better designer as its knowledge base expands. Until all possible world knowledge about designing microprocessors has been coded in the DAA's knowledge base, there will always be room for improvement in its designs.

This KBES approach has opened the door to a whole new class of intelligent computer-aided design tools.

Panel 6. Convert Multiplexer (MUX) Inputs to Bus

IF: the most current active context is bus allocation
and there is a module that is a multiplexer
and there is another module that is also a multiplexer
and there is a link from a non-bus module to the first multiplexer
and there is a link from that module to the second multiplexer
and there is a link from another non-bus module to the first multiplexer
and there is a link from that module to the second multiplexer
THEN: place these connections on an idle bus

Acknowledgments

We would like to thank M. C. McFarland for his contributions in bottom-up design.

References

1. C. Mead and L. Conway, *Introduction to VLSI systems*, Addison-Wesley, Reading, Massachusetts, 1980.
2. T. J. Kowalski et al., "The VLSI Design Automation Assistant: From Algorithms To Silicon," *Design and Test of Computers*, Vol. 2, No. 4, August 1985, pp. 33-43.
3. T. J. Kowalski and D. E. Thomas, "The VLSI Design Automation Assistant: First Steps," *Twenty-sixth IEEE Computer Society International Conference*, February 28, 1983, pp. 126-130.
4. T. J. Kowalski and D. E. Thomas, "The VLSI Design Automation Assistant: A Prototype System," *Proceedings of the Twentieth Design Automation Conference*, IEEE, June 27, 1983, pp. 479-483.
5. T. J. Kowalski, *An Artificial Intelligence Approach to VLSI Design*, Kluwer, Boston, Massachusetts, 1985.
6. T. J. Kowalski and D. E. Thomas, "The VLSI Design Automation Assistant: An IBM System/370 Design," *Design and Test of Computers*, Vol. 1, No. 1, February 1984, pp. 60-69.
7. M. C. McFarland S.J. and T. J. Kowalski, "Assisting DAA: The Use of Global Analysis in an Expert System," *Proceedings of the IEEE International Conference on Computer Design*, October 6, 1986, pp. 482-485.

8. P. Marwedel and G. Zimmermann, *MIMOLA Software System User Manual*, Vol. 1, Institut Fur Informatik und Praktische Mathematik, Christian-Albrechts-Universitat Kiel, May 1979.
9. L. J. Hafer, *Automated data-memory synthesis: A Format Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic*, Ph.D. thesis, Department of Electrical Engineering, Carnegie-Mellon University, June 1981. (Also in Design Research Center DRC-02-05-81.)
10. L. Hafer, *Data—Memory Allocation in the Distributed Logic Design Style*, Master's thesis, Carnegie-Mellon University, December 21, 1977.
11. C. Y. Hitchcock III, *Automated Synthesis of Data Paths*, Report No. CMUCAD-83-4, SRC-CMU Center for Computer-Aided Design, Carnegie-Mellon University, January 1983.
12. C. J. Tseng and D. P. Siewiorek, "Facet: A Procedure for the Automated Synthesis of Digital Systems," *Proceedings of the Twentieth Design Automation Conference*, IEEE, June 27, 1983, pp. 490-496.
13. E. A. Feigenbaum, *Knowledge Engineering: The Applied Side of Artificial Intelligence*, Computer Science Department, Stanford University, 1980.
14. C. L. Forgy, *OPS5 User's Manual*, Department of Computer Science, Carnegie-Mellon University, July 1981.
15. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Massachusetts, April 1979.
16. C. Davis, personal letter to D. E. Thomas, August 12, 1983.
17. T. J. Kowalski, "The VLSI Design Automation Assistant: The IBM 370 Critique," Department of Electrical and Computing Engineering, Carnegie-Mellon University, September 1983.
18. M. A. Rose, *Structured Control Flow: An Architectural Technique for Improving Control Flow Performance*, Master's thesis, Department of Electrical Engineering, Carnegie-Mellon University, November 1983.
19. B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell System Technical Journal*, Vol. 49, No. 2, February 1970, pp. 291-308.
20. D. G. Schweikert and B. W. Kernighan, "A Proper Model for the Partitioning of Electrical Circuits," *Proceedings of the 9th Design Automation Workshop*, ACM IEEE, 1972, pp. 57-62.
21. M. A. Breuer, "A Class of Min-Cut Placement Algorithms," *Proceedings of the 13th Design Automation Workshop*, ACM IEEE, 1976, pp. 284-290.
22. T. S. Payne and W. M. vanCleemput, "Automated Partitioning of Hierarchically Specified Digital Systems," *Proceedings of the 19th Design Automation Conference*, ACM IEEE, 1982, pp. 182-192.
23. M. R. Barbacci et al., *The ISPS Computer Description Language*, Department of Computer Science, Carnegie-Mellon University, August 16, 1979.
24. M. C. McFarland, "Computer-Aided Partitioning of Behavioral Hardware," *Proceedings of the 20th Design Automation Conference*, ACM IEEE, June 1983, pp. 472-478.
25. W. Wolf, "An Object-Oriented, Procedural Database for VLSI Chip Planning," *Proceedings, 23rd Design Automation Conference*, ACM IEEE, June 1986.

(Manuscript received September 30, 1987)