Techniques of knowledge-based systems are now being applied in VLSI synthesis and design. While still experimental, they promise more capable design aids.

Automatic Data Path Synthesis

Donald E. Thomas, Charles Y. Hitchcock III, Thaddeus J. Kowalski, Jayanth V. Rajan, and Robert A. Walker

Carnegie-Mellon University

Computer design aids for digital systems began as programs performing the routine tasks of bookkeeping and manipulating large sets of data under the control of a human operator. As designs grew, reliable analysis and optimization programs evolved to aid further in integrated circuit design. Although design synthesis was formerly considered to be the realm of the creative designer, automatic and computer-aided programs are now being developed for many of the levels of IC design. As we move into the VLSI era, the demand for more capable system ICs requires even greater productivity at all levels of the design process. Thus, development of synthesis tools for the creative design process has become an important research area.

Synthesis is the creation of a detailed design from an abstract specification. Digital system design actually consists of many synthesis steps, each adding detail. The quality of designs produced by automatic synthesis programs are not yet adequate for production use. However, their use as a computer aid permitting designer interaction is becoming a reality,[1-4] and promises further benefits:

- More design alternatives. Designers can specify parts of a design and have the synthesis program fill in details quickly, or they can change constraint specifications so the synthesis aid specifies a different design.
- Correct designs. Synthesis programs make decisions which add new detail to the design. If the added detail correctly implements the abstract specification, many design errors can be eliminated.
- Multilevel representations. Synthesis programs can maintain correlations between abstract specification and detailed design in the form of a representation with multiple levels of abstraction. The representation supports the use of highly capable design aids such as mixed level simulators and timing verifiers.

The research described here is the portion of the CMU-DA system[1] that operates between the behavioral level and the functional-block level of IC design as shown in Figure 1. At the behavioral level, the abstract specification of the system to be designed resembles a program written with a high-level language such as Pascal or C. A behavioral statement such as "$A = B+C$" states only that two values B and C must be combined to form a new value A. The functional-block level is characterized both by the interconnection of abstract components such as registers, arithmetic-logic units and programmed logic arrays as well as a control sequence which specifies the order of register transfers.

While many design philosophies can use this addition statement to describe a functional-block adder module behaviorally, the CMU-DA system uses the behavioral statement to suggest functional block components such as registers to hold the three values, a logic module capable of performing an add, and data paths capable of routing information from registers B and C through the add module to register A. Any single behavioral statement can specify any number of functional-block modules. Further, this add module can perform additions specified by other behavioral statements. By considering the behavioral description as a whole, the design programs shown in Figure 1 can specify alternate interconnections of functional block modules that will implement the specified behavior.

Input to the CMU-DA system is an ISPS[5] behavioral description, which is compiled and translated into an internal data flow representation called the value trace. It is then made available to the design programs at this level, which offer the following functions to human designers and external design programs:

- graphics to plot and display the value trace,
- metrics to calculate and display the analysis based on the value trace,

0018-9162/83/1200-0059$1.00

- tranformations to the value trace data flow representation to improve design quality,
- partitioner that suggests which portions of the value trace should be implemented with common hardware,
- control step allocation for sequencing events in the data path and determining design parallelism/serialism ratios, and
- data path allocation in terms of functional block components, including the generation of a multilevel representation that specifies both behavioral and functional block information.

This article describes the development of a computer-aided design environment for studying automatic synthesis between the behavioral and functional block levels of design. The enviroment is based on the value trace, which can be used to derive alternate system implementations. The DAA[6] and EMUCS[7] synthesis programs shown in Figure 1 will also be described and compared.

The value trace

ISPS has been used widely to describe computer architectures and compare different architectural families.[8]

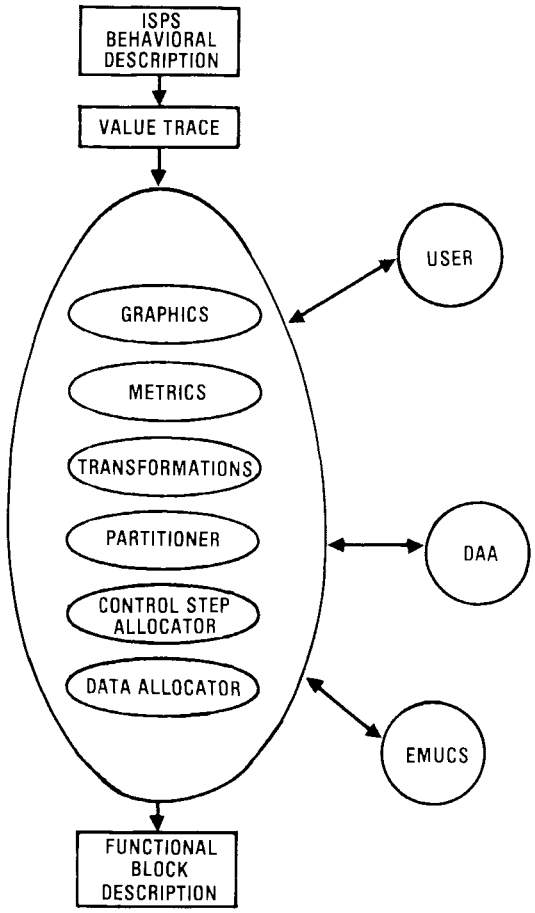


Figure 1. Behavioral to functional block level of the CMU-DA system.

Its use has been extended to the description of general digital controllers. And in the context of describing computer architectures, synthesis programs are used to generate alternate implementations of the architecture.

Synthesis programs require a representation internal to the design programs to permit easy recognition and implementation of design features. Snow proposed a data-flow representation—called the value trace—for this purpose.[9-11] The value trace is a directed acyclic graph (DAG) similar to those used in optimizing compilers and generated directly from an ISPS description. An examination of the value trace provides an understanding of the synthesis capabilities needed in behavioral-level IC design.

The nodes of the value trace graph, called activities or operators, represent operations to be performed. The arcs of the graph, called values, represent the data flow from one operator to another. To the basic DAG used in optimizing compilers, Snow added control constructs to allow conditionals and subroutines. He also identified value-trace transformations that optimize the implementation without altering behavior. Before considering the effects of the transformations, it is necessary to understand the structures of the value trace.

At the highest level, the value trace is divided into individual groups of operations (or subgraphs) called VT bodies. This division is determined from the ISPS description and occurs at procedural boundaries and at labeled blocks. Each VT body is further divided into operators or activities of three types: arithmetic and logic operators, control operators, and value-trace-specific operators. Arithmetic and logical operators are defined in ISPS and perform such functions as addition and equivalence testing. Control operators can invoke other VT bodies, restart the current VT body, and control the conditional choice associated with data multiplexing or control branching. Some operators are specific to the value trace and its needs as a representation for hardware synthesis. Examples of VT-specific operators are those required for reading a value subfield, or performing sign extension when reading from a smaller into a larger carrier.

Figure 2 shows a sample ISPS two-part description taken from the description of the MOS Technology Incorporated MCS6502 microprocessor. The first part defines the *IR, P,* and *F* registers: *IR* and *P* are eight bits wide with the left-most bit numbered 7 and the right-most bit numbered 0, while the *F* register is one bit wide. *N* is defined to be a synonym, or mapping, for bit 7 of the *P* register, *V* a mapping for bit 6, etc. The second part of the ISPS fragment decodes the three most significant bits of the instruction register and chooses one of a set of alternative actions based on that decoding. For example, if bits 7 through 5 contain the value 2, the variable *F* is assigned the complement of the overflow bit.

The value trace fragment in Figure 3 shows this same decoding loop, with the last four branch alternatives eliminated for space reasons. Each block represents a value-trace operator. For example, the operator NOT outputs the complement of its input value. The operator <r>, called a bit read, produces the subfield of its left input, which begins with the bit specified by its right input and continues to the left for the width specified by

the output. The compound operator, consisting of the blocks SEL (SELECT) and END (END SELECT) and the items connected to them by dashed lines, is called a SELECT. It is used in the value trace to implement the DECODE and IF...THEN constructs. In Figure 3, it decodes the instruction register. Alternate actions are shown side-by-side with the value of the selector required for a particular action shown at the top of that action. The result of the chosen action is then understood to be passed to the END SELECT, where it continues to the rest of the VT body.

Besides the value trace representation, Snow defined three major groups of optimizing transformations: operator transformations, which act on individual operators or groups of operators; SELECT transformations, which act on the SELECT construct, individually or in groups; and VT body transformations, which work with individual or grouped VT-bodies. Snow has described them in detail, but summaries of some of his descriptions should be helpful here:

- Constant folding is a simple, classical optimizing transformation, replacing an operator that acts solely on constants with a new constant. For example, the PLUS operator associated with the statement 2 + 3 could be eliminated and replaced with the constant 5. Constant folding may also be applied to a SELECT, in which case the appropriate branch of the SELECT is retained and the others eliminated.
- Redundant operator elimination is classical transformation similar to common subexpression elimination in optimizing compilers. It may be used when two or more operators of the same type have the same inputs and eliminates redundant operators, referencing their outputs with outputs of the retained operator.
- SELECT motion replaces similar activities within the branches of a SELECT with a single activity outside the SELECT, and vice-versa. To be moved out of a SELECT, activities must be of the same type, have the same inputs, and occur in all branches.
- SELECT factoring removes branches from a given SELECT, then uses them to form a new SELECT cascaded with the old one. SELECT combination is the inverse of SELECT factoring.
- VT body inline expansion is analogous to the inline subroutine expansion, replacing a call to a VT body with a copy of that VT body.
- VT body formation is the inverse of VT body inline expansion; it encapsulates a group of operators into a new VT body and adds a CALL to this new VT body from the original.
- Loop unwinding consists of inline expansion of a looping VT body to permit constant folding and other transformations. It replaces a loop counter with a finite number of calls to a given subroutine.

The first benefit of optimizing transformations is the reduction of the biases inherent in the ISPS description. The designer's ISPS programming style may break up behavior into separate procedures, and the architecture description may have separate ISPS procedures for each addressing mode and machine instruction. There may also be higher level procedures that decode groups of instructions, such as the conditional branches, and finally one concise top-level procedure that describes decoding of all the instructions in terms of calls to the above procedures. While a system architect may write and understand a one-page, instruction-decoding loop that calls four pages of detailed procedures, inline expansion may be used to convert the internal representation into a five-page loop that more closely represents the final control implementation.[12]

```
IR<7:0>,                ! INSTRUCTION REGISTER
P <7:0>,                ! PROCESSOR STATUS REGISTER
    N< >  := P<7>,      ! NEGATIVE RESULT
    V< >  := P<6>,      ! OVERFLOW
    B< >  := P<4>,      ! BREAK COMMAND
    D< >  := P<3>,      ! DECIMAL MODE
    I< >  := P<2>,      ! INTERRUPT DISABLE
    Z< >  := P<1>,      ! ZERO RESULT
    C< >  := P<0>,      ! CARRY
F <0>,                  ! TEMPORARY VARIABLE

DECODE IR<7:5> →
BEGIN
    #0 := F = NOT N,
    #1 := F = N,
    #2 := F = NOT V,
    #3 := F = V
    #4 := F = NOT C,
    #5 := F = C,
    #6 := F = NOT Z,
    #7 := F = Z,
END NEXT
```

Figure 2. Sample ISPS description from the MCS6502 microprocessor.

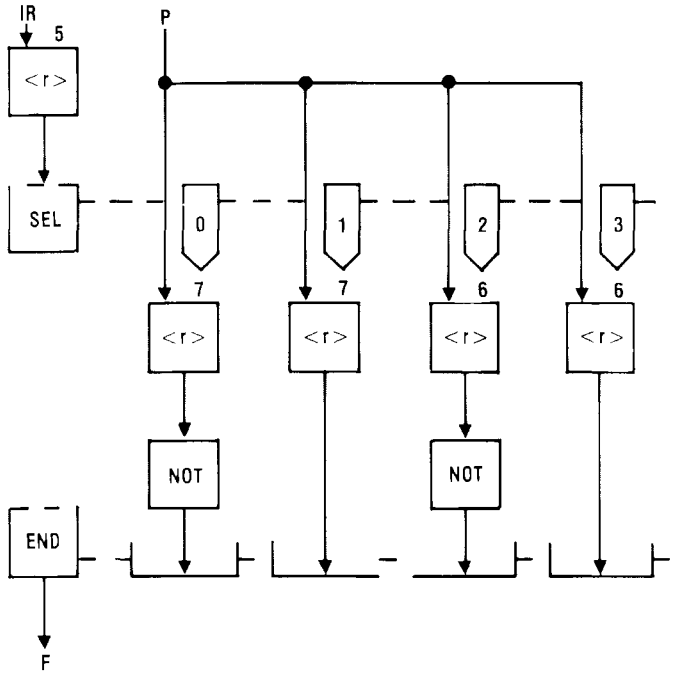


Figure 3. Value trace fragment of ISPS instruction register.

In addition, transformations eliminate duplicated operators and control steps. Successive behavior-level statements with common subexpressions can be combined. Calculations are made once, and the results are stored in a temporary variable. Besides eliminating duplicated operators and control steps, value trace transformations can eliminate redundant hardware. Thus, they function at the systems level of design as a design tool.

Finally, because of the nature of the data flow representation, the synthesis programs can change the order of the operations specified in the ISPS description—so long as data dependencies are satisfied—and can change design parallelism. This resequencing of control steps is one of the more powerful features of the value trace. In summary, the ISPS description can be used as an initial description of the behavior to be implemented in hardware. The value trace is then used as a basis for manipulating designs and making decisions about the control and data parts of the design.

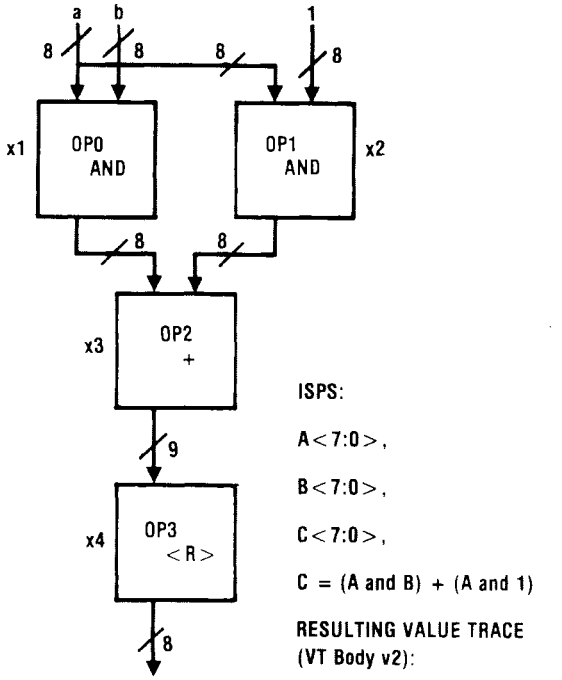


Figure 4. A short sample of a value trace body.

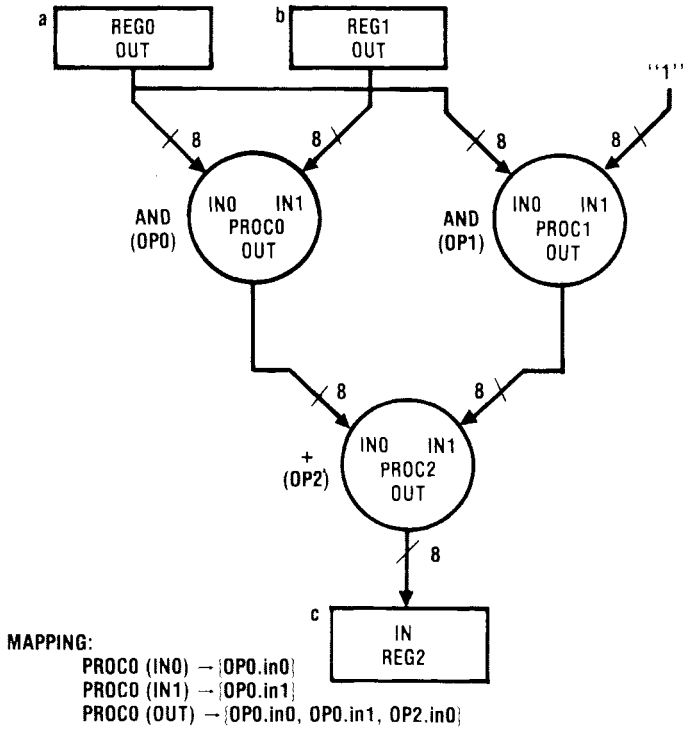


Figure 5. Hardware implementation executed by a single-control action.

Sequencing in design synthesis

Value trace descriptions of a machine's behavior can be conveniently implemented in hardware in a variety of forms. These different implementations are characterized by the design parallelism, which is set during the control step allocation of a VT body, before the functional-level synthesis that specifies the data path of the final implementation.

Designers can implement a whole VT body, such as in Figure 4, directly in hardware. Value-trace operators would be replaced by combinational circuits and arcs by wires. Figure 5 represents the resulting implementation, which executes in a single control action. Operators 0, 1, and 2 (OP0, OP1, and OP2) have been replaced by Processors 0, 1, and 2, respectively ("operator" denotes a VT-level operation; "processor" or "hardware operator," a functional level hardware module such as an adder). The values in registers 0 and 1 feed into the VT body inputs, while output goes into register 2. This implementation is as parallel as the original algorithm permits, but it requires many hardware operators. It would be more desirable to share hardware operators among abstract operators and reduce hardware requirements. Such sharing requires that a single control action be divided into a series of separate actions.

A control step is the basic control operation in a digital machine. It is often implemented as a state in the controlling state machine. When the value trace is first translated from the ISPS description, only necessary control and data dependencies are represented. Thus, value-trace-level operators that could be implemented in parallel are identified. Control-step allocation then assigns value-trace-level operations to control steps, and operators can be sequenced serially or with maximum parallelism. Therefore the control-step allocation determines the serial/parallel nature of the design and approximates cost-speed trade-offs.

Control-step allocation also specifies values to be stored for use in the next control steps and, consequently, determines data path storage needs. For example, operators 0 and 1 might be assigned to the first control step in Figure 4, and operator 2 to the second. Since the

values from operators 0 and 1 (OP0.out and OP1.out) are generated in the first control step but used in the second, they need to be stored in registers, as illustrated in Figure 6. Here, processor 0 implements operator 0 during the first control step and operator 2 during the second control step. A multiplexer is also needed to gate the value into register 1. Figure 7 shows the maximium serial implementation, a single value-trace operation per control step. In this design, a single processor is shared by all three combinational value trace operators.

A simple, automatic control-step allocator now provides a means of generating sequenced value trace descriptions for synthesis programs. And as the figures show, different hardware implementations can be synthesized from the same value trace. Each implementation requires a different number of control steps and a different amount of processing hardware, producing a cost-speed trade-off. In addition, a behavioral statement may be implemented with many functional modules, which may be shared with other behavioral statements, a feature characteristic of CMU-DA design programs.

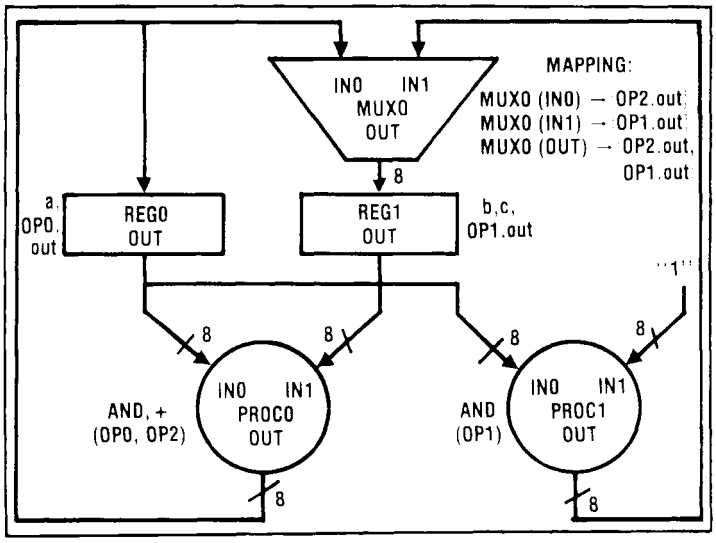


Figure 6. High serial control-step implementation.

EMUCS data path synthesis

An algorithmic approach to digital data path synthesis has been developed in a program called EMUCS. The input to the algorithm is a value trace translated from an ISPS description, while the result is a functional-block-level representation of the data path. The algorithm aims at an implementation of a VT body at minimal "cost" in terms of any quantitative parameter such as power or chip area. The algorithm maps, or binds in a step-by-step fashion, value trace elements to hardware elements—operators (which are bound to processors), stored values (which are bound to registers), and transfer paths (which are implemented as buses, multiplexers, and connections). Control steps have already been specified by the control step allocator and are not changed during the data path synthesis.

The synthesis algorithm, as proposed by McFarland,[13] is iterative in nature. It first analyzes the existing intermediate data path to decide which value-trace element to bind. It then binds that one element, changing the data path as necessary. Then it iterates, reanalyzing and binding until all value trace elements have been bound and the final data path created.

The analysis step generates cost tables that reflect the feasibility of binding each value trace element to each hardware element. It evaluates bindings that require more hardware or more complexity as being more costly. After computing costs, the algorithm searches the cost tables to find the least expensive binding, not from the standpoint of the lowest cost during this step, but because it might minimize costs in the next iterations.

For example, the program can make one of several possible bindings during an iteration. Hardware design rules might prevent it from making a similar binding during the next iteration. To illustrate, the current binding might have used hardware element 1 during the third time interval. In the next iteration, it would be impossible to bind another value trace element to hardware element 1 during the third time interval.

To eliminate potential conflicts, the algorithm looks not only at the least costly, but also at the second least costly value-trace element. The lowest cost is guaranteed only if that element can be bound during this iteration.

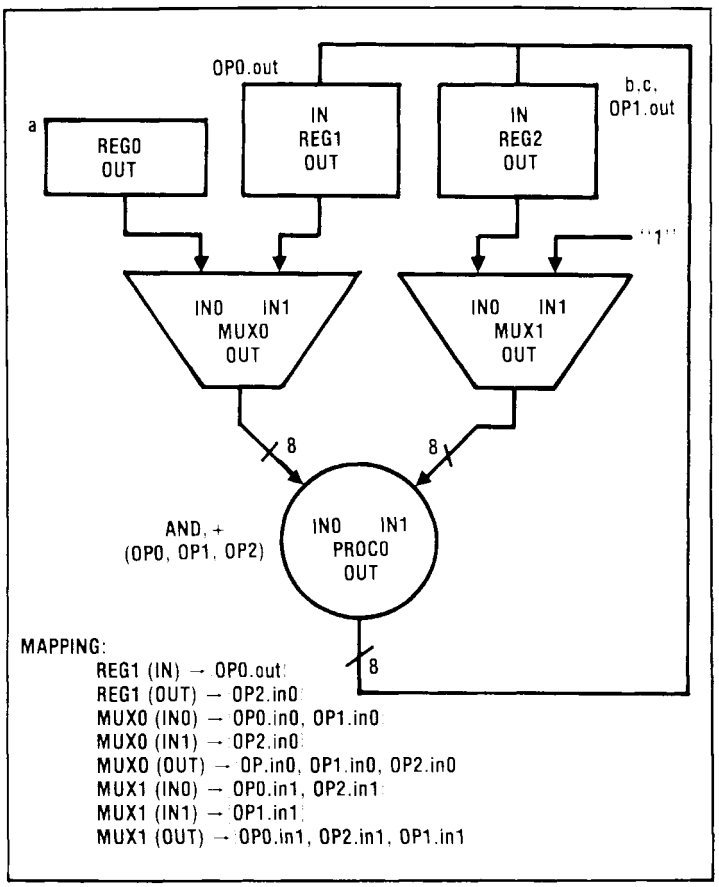


Figure 7. Maximally serial implementation with single value trace operation for each control step.

Otherwise the second lowest may be the best. The difference between the two lowest costs for each value trace element then becomes a measure of how important it is to bind that element during this iteration—of how much stands to be "saved." If the difference is great, it might cost much more to bind it later; if small, it should not cost much more to bind it later. The algorithm chooses the binding that saves "the most": this is the min-max criterion. But this cost evaluation is short-sighted, since these binding costs can change with every new binding. Nonetheless the algorithm still produces powerful results.

Cost tables. The cost-table calculations are based on cost parameters reflecting the need to use, create, and/or modify hardware to make a binding. The cost parameters form a database that determines the preferred bindings. By changing the database, the same synthesis algorithm can create different designs from the same value trace, the same control-step allocation, same registers, and same processors (with functionality possibly changed). Thus, the algorithm's primitive cost parameters are the only internal factors that affect design style.

Whenever a binding uses a piece of hardware (such as storing a value in a register), there is a use cost proportional to the size of the hardware and the number of control steps involved. Similarly, create costs are used to assign a cost to the installation of a piece of hardware, such as a multiplexer. The final costs reflect modifications to existing hardware, as in adding an input to a multiplexer. These modify costs are proportional to the size in bits of their modifications. Since hardware modifications cannot be bound to hardware operators and VT operators cannot be bound to registers, we have separate cost tables for each type.

A sample iteration. A quick pass through an iteration of the algorithm will make the process clear. Figure 8 shows the design of a partially bound VT body partitioned into two control steps. Stored values (VALx) have already been bound to registers 1 and 2. Operator 3 (OP3) has been bound to processor 1 (PROC1). Figure 9 shows the partial data path with bindings labelled by the corresponding elements (i.e., value 3 is bound to register 2). Table 1 shows the updated operator/processor cost table for this iteration of the algorithm. The value/register cost table is not shown, since all the values have already been bound.

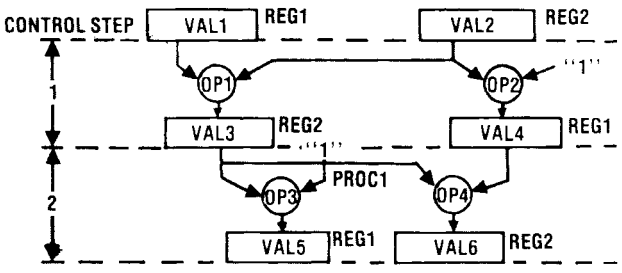


Figure 8. Value trace body fragment in two control steps.

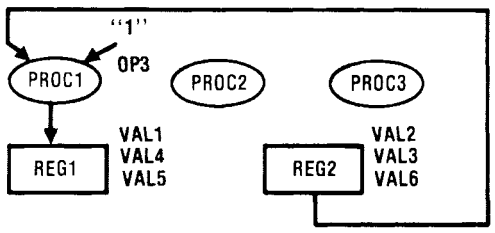


Figure 9. Intermediate data path with bound design values.

The entries for operator 3 in the operator/processor cost table are shown as X, since that operator has already been bound to processor 1. Notice in Figure 8 that operator 4 cannot be bound to processor 1 since that would prevent operator 3 from using processor 1 during control step 2. To analyze the cost of binding operator 2 to processor 3, we first check for use conflicts. Since processor 3 is not in use during control step one, we proceed to the cost of adding the functionality of operator 2 (AND, +, etc.) to processor 3. In this simplified example, we will assume that the function has no assigned cost and that all other cost elements relate to connectivity. We start at the operator's left input, which is fed by value 2. Value 2 is bound to register 2, so we need to calculate the cost of connecting the register 2 output to processor 2 left input. Since the processor 2 left input is currently unconnected, the connection could be a direct connection at the parameterized cost of 10 units. Similarly, a constant value of 1 could be directly connected to the processor 2 right input at a cost of 10 units.

What is left is the cost of connecting the operator 2 output. Through the value trace, we see that value 4 is bound to register 1, but the constraint of a single input prevents a direct connection. So a multiplexer with an input from processor 1 output feeding register 1 must be created. The output of processor 3 would then be connected to an input of this newly created multiplexer. The create cost for this new multiplexer might be five units, and 10 units for creating a direct connection from the output of processor 3 to an input on the new multiplexer. The accumulated cost for binding operator 2 to processor 3 is 35 units, as reflected in the operator/processor cost table in Table 1. Other binding costs shown there were calculated similarly.

The min-max criterion now decides which element should be bound next. The last column in the operator/processor cost table shows the difference between the

Table 1.
Operator/processor connection costs in units representing chip area or speed.

OP	PROC 1	PROC 2	PROC 3	MIN-MAX COST
1	40	30	30	0
2	0	35	35	35
3	X	X	X	X
4	X	30	30	0

two lowest costs in each row. To minimize the loss, we pick the row with the largest value in this column, the one for operator 2 in this case, and bind it to the minimum cost hardware element, processor 1. This choice is not surprising since the structure of operator 2 is exactly that of operator 3, which was already bound to processor 1. Note that until now no changes to the hardware structures have been made, only predicted through associated costs. And the hardware is now changed to reflect this binding decision. The next iteration of the algorithm then begins by recalculating the operator/processor cost table so that another binding decision can be made.

EMUCS MCS6502 design results

To test and measure EMUCS, the program was run on a series of samples of increasing size, the largest an implementation of the MCS6502 processor instruction set. Its ISPS description was translated into a series of VT bodies. In its current prototype state, EMUCS can only process a single VT body. Fortunately the inline expansion transformation previously described could reduce most of the MCS6502 value trace to a single VT body with 197 operators and 181 stored values.

Two implemenations of this behavior were created, the first automatically. The eight-bit data paths are shown in Figure 10. This automatic implementation presented an opportunity for good bus structure, so a second bus implementation was suggested before any elements were bound. The resulting eight-bit data paths are shown in Figure 11. Manually inserting a bus is quite easy in EMUCS, as are such design interactions as setting synthesis breakpoints, examining the value trace or hardware structures, or binding elements. EMUCS interaction was patterned after symbolic debuggers to produce a convenient, semiautomatic synthesis tool.

To compare the two implementations, bit counts of hardware categories are provided in columns 1 and 2 of Table 2. Horizontal lines show the number of bits in data path modules, such as ISPS declared registers (Dreg), temporary or other registers (Treg), specified operator, constants, multiplexer, controller, or bus found in the design. Numbers do not truly reflect the best possible EMUCS performance, since it still is restricted to mapping value trace operators to processors of the same width. Without this constraint, the number of bits for the processor and multiplexer input and output would decrease appreciably.

The two MCS6502 implementations differ very little, except for the differences caused by the added bus. Their forms are both quite distributed, with many registers fed by the multiplexer and many point-to-point interconnections. Registers tend to hold the values of several different ISPS variables at different times in the VT body's sequencing, which explains why the table lists all registers in the Treg category. Exceptions are the three registers *A*, *X*, and *Y* in Figure 10. They are associated almost exclusively with the ISPS variables *A*, *X*, and *Y*, respectively. This correspondence reflects the EMUCS ability to combine values with common requirements in the same register, but it is less strong in the bus version.

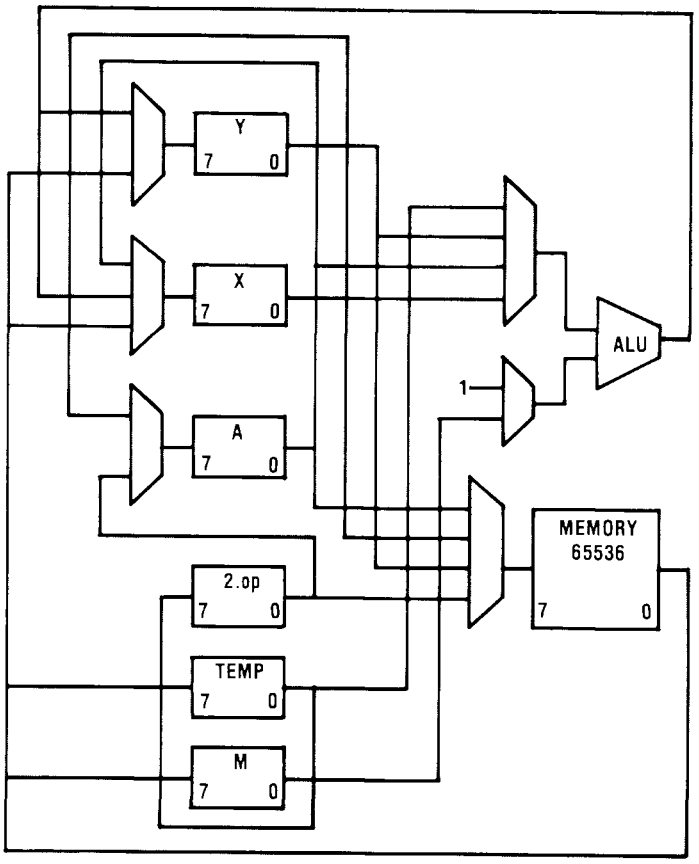


Figure 10. Eight-bit MCS6502 data path designed automatically by EMUCS.

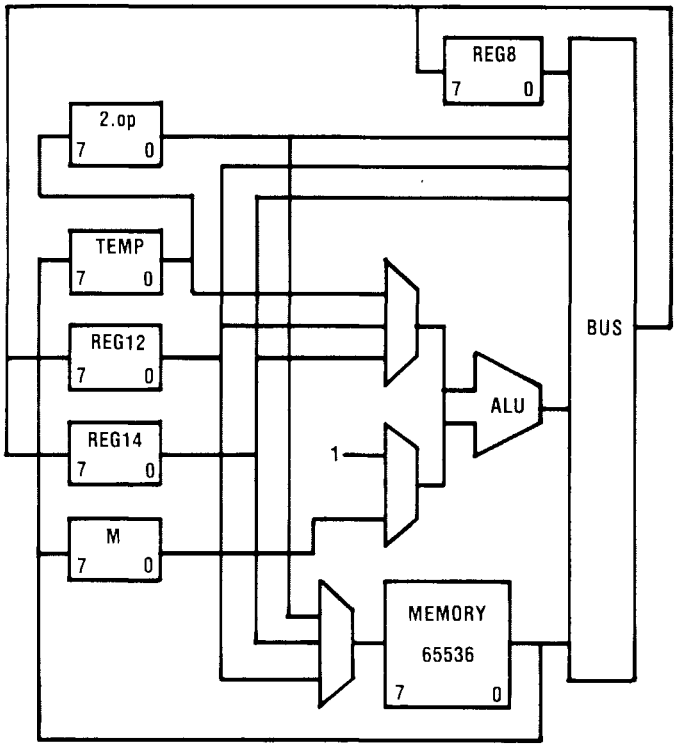


Figure 11. EMUCS automated design of MCS6502 data paths with bus.

Since both designs were initialized identically, the number of control steps and state (registers) and processor (ALUs) are the same. The only structural difference is in the transfer path hardware and routing. The bus implementation uses fewer multiplexers, but at the cost of an added bus. Although this bus design might require more area, the regularity imposed by the bus simplifies fabrication.

VLSI design automation through DAA

During the past decade, knowledge-based-expert systems have been developed by researchers in artificial intelligence to help solve design problems when structures do not lend themselves to recipe solutions. A KBES is based on the premise that humans solve problems by recognizing patterns and associating previously effective solutions with them. Pattern recognition may not always be based on the complete, current situation; nor is it recognized with absolute certainty. Still, the presence or absence of patterns can suggest possible solutions.

Based on this human model, DAA manages design synthesis from the behavioral to the functional block level in the CMU-DA system. Its design function resembles the EMUCS program, but its approach is completely different. The initial knowledge in DAA was encoded from the algorithms of a previous CMU-DA allocator[14] and from designer interviews.[6] Then the prototype system was refined by expert designers. It has now been implemented as a production system using the OPS5 KBES writing system.[15] It formulates the problem using three major components: a working memory, a rule memory, and a rule interpreter.

The working memory that describes the current situation is a collection of elements similar to the data structures in conventional programming languages:

```
literalize module
  id:adder.0
  type: operator
  atype: two's complement
  bit-left: 17
  bit-right: 0
  attibute: +
```

The above element describes an 18-bit, two's complement adder module. Additional information might include its connection to other components, such as a register, and a tag referencing an operator in the value trace. The working memory elements are pattern-matched by the rule interpreter against the rule memory to determine what rules are applicable in a given situation.

The rule memory is a collection of conditional statements similar to those of conventional programming languages, which operate on elements stored in the working memory. The following sample rule (translated into English) describes a replacement link to the input of a module:

```
If the most current active context is to create a link
  and the link should go from a source port to a
    destination port
  and the module of the source port is not a
    multiplexer or a bus
  and there is a link from another module to
    the same destination port
  and this other module is also not a multiplexer
    or a bus,
Then create a multiplexer module
  and connect the multiplexer to the destination port
  and connect the source port and destination port
    link to the multiplexer
  and move the other link from the destination port to
    the multiplexer.
```

Rule selection is data driven. The rule interpreter looks through the rule memory for a rule whose conditions are all true. If more than one rule is applicable, the rule dealing with the working memory element most recently modified is selected first. If there are still multiple rules applicable, the most specific rule is selected. In this case, the most specific rule is the one with the most conditionals true, since each conditional gives more detailed knowledge. The process resembles a human train of thought and the use of special-case before general-purpose knowledge as far as possible. It is repeated until no more rules are applicable or until a rule explicitly stops the process. Separating expert knowledge from reasoning simplifies the incremental addition of new rules and the refinement of old ones, since rules have minimal interaction with one another.

Table 2.
Tabulation of data path components for EMUCS and DAA.

MODULE TYPE	EMUCS-1	EMUCS-2	DAA-1	DAA-2
TREG	182	182	32	58
DREG	—	—	173	196
ALU	68	68	17	17
PLUS	0	0	17	17
CMP	0	0	0	1
NOT	0	0	4	4
XOR	0	0	8	9
OR	0	0	9	1
AND	0	0	9	9
CONSTANTS	143	143	140	177
MUX IN	673	613	80	92
MUX OUT	186	161	32	34
BUS IN	0	81	569	849
BUS OUT	0	51	146	241
CONTROLLER IN	42	41	32	36
CONTROLLER OUT	80	83	78	95

Synthesis subtasks. DAA rules are grouped into a set of temporally ordered subtasks. Synthesis begins by assigning hardware storage modules such as registers and memories to all VT values declared in the ISPS description. This step corresponds to assigning storage modules for the major state registers of a machine. Then a VT body is selected, and control steps are allocated to develop a parallel design. Next the synthesis task maps all VT operator outputs not previously assigned to storage modules. Finally it maps each VT operator to processor modules, connecting links and supplying multiplexers where necessary. The two mapping steps regularize the

behavioral description for the following expert-analysis phase.

The expert-analysis subtask removes registers from processor outputs, where sources are stable. Results from combinational logic modules (processors) are latched in a temporary register only if the values in the register sourcing the processor change before the subtask needs them. Single function operators are combined, where appropriate, into ALUs.

DAA also examines the possibility of sharing temporary registers. Where possible, it increments, decrements, and shifts operations in existing registers. It also places registers, memories and ALUs on appropriate buses. Throughout this subtask, constraints cause tradeoffs between the number of modules and the allocation of control steps. Once a subtask is complete, the process is repeated for other VT bodies.

Associated with each DAA subtask is a set of rules for designing VLSI systems—about 300 for the whole system—most of which define extensions for partial designs. By determining at each step whether extending the design in some way is consistent with constraints, they enable DAA to synthesize an acceptable design.

DAA uses a search called Match[16] to explore possible designs. Following the same design process for each subtask, it extends partial design without backtracking in any problem. Rule selection order within each subtask is highly variable, depending on the particular combination of data flow components in a design and the selection order for VT bodies.

DAA MCS6502 design results

The prototype DAA system had almost 70 rules and could design a MOS Technology Incorporated MCS6502 microcomputer in about three hours of VAX 11/750 CPU time. We asked expert designers at Intel and Bell Laboratories to evaluate the design, explain what was wrong, why it was wrong, and how to fix it. Rules were modified, new rules were added, and the MCS6502 was re-designed. DAA recently designed an acceptable MCS-6502 microcomputer in four hours. Table 2 summarizes the design characteristics. The third column summarizes DAA-created data paths of the partial MCS6502 design used by EMUCS. The fourth column gives the complete design.

Figure 12 shows the eight-bit-wide data paths for the MCS6502 microcomputer. The 16-bit and one-bit data paths have been excluded for clarity but are included in Table 2.

The experts found the complete design satisfactory, but recommended point-to-point connections and multiplexers to reduce contention on the bus and save control steps. They also suggested using a second eight-bit bus. These and other recommendations are being incorporated in the system.

EMUCS and DAA provide two examples of design aids that synthesize a functional block design from a behavioral description. Alternate designs produced by these synthesis programs have similar characteristics and present reasonable design trade-offs. The following list presents a general comparison of the two synthesis programs:

- EMUCS allocated values to registers more effectively (182 versus 205) than DAA. DAA strictly preserved the architectural registers declared in the ISPS description, yielding a more comprehensible final design.
- Both algorithms specify hardware operators, but DAA tends to leave some simple logic operations in separate modules whereas EMUCS tries to combine all such operations into ALU's.
- Multiplexer and bus inputs/ouputs must be considered together. EMUCS mixes point-to-piont connections and multiplixers, but uses buses only when they are manually introduced. DAA uses a multiplexer strategy, introducing buses in a limited number of situations.
- DAA can change the control-step allocation during the design process. When processor delays do not fit into a control step, it changes the control step assignment and continues.
- EMUCS is written in C programming language, which uses much less CPU time and memory than OPS5, a LISP-based system used for DAA. However, a newer version of OPS5, based on BLISS, is expected to increase DAA speed 10 times.
- DAA has a rich set of domain knowledge (or information directly pertinent to IC design) in its rules. EMUCS has little domain knowledge but applies a

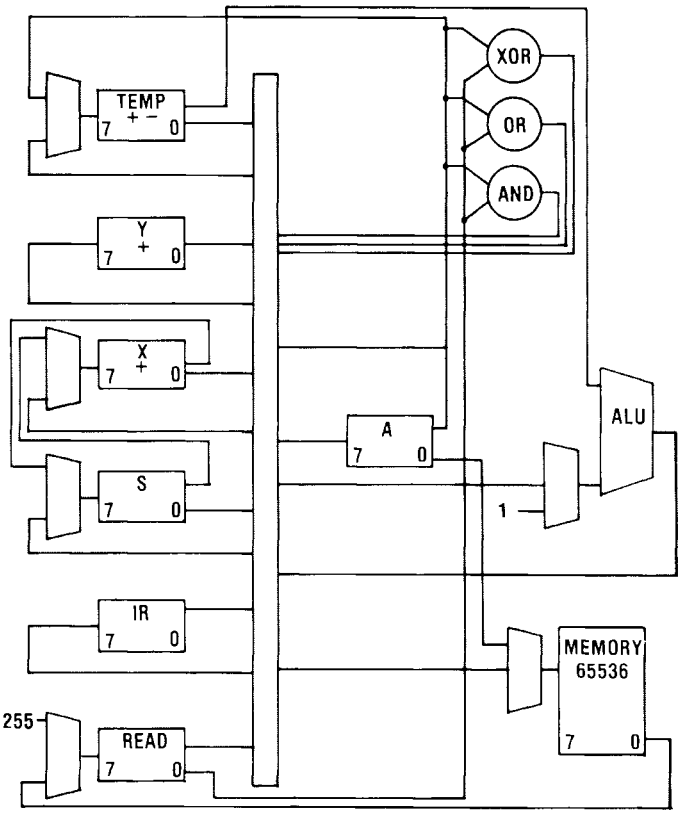


Figure 12. MCS6502 eight-bit data path designed by DAA.

simple, general, parameterized costing algorithm to the synthesis problem.

In short, EMUCS is highly interactive and may make a good computer-aided synthesis program, but a KBES approach, such as DAA offers, should capture more of the creative design process and thus produce better designs.

Multilevel representation

The CMU-DA multilevel representation maps structural elements at the functional-block-level map into nodes and arcs in the value trace. It provides design trade-offs between more detailed modeling and model simplification, particularly at the behavioral level. Multilevel simulation of a digital system selects subsystems at the functional-block level, retaining the bulk of the system at the behavioral level. It is possible to verify partial designs, substantially reducing turnaround time and design cost.

Multilevel representation also supports formal verification of transformations from one representation to another (i.e., the synthesis decisions). It can also be used to feed back information extracted from the lower levels and thereby control high-level synthesis.

Examples of multilevel representation. To define a multilevel representation, it is necessary to isolate and identify structural and behavioral features at each level and relate them to one another. At the functional block level, the primitive structural elements are of three major types: storage elements, such as registers and memories; transfer paths, such as multiplexers, buses, and demultiplexers; and processors, which are combinational circuits that multiply, add, etc. Although an unallocated value trace has no structural features, we can identify some structural elements in an allocated value trace because values and operations are bound respectively to storage elements and hardware operators by synthesis programs.

```
controller: EXAMPLE;
  COMPUTE (v2) :
    cstep [1:1] {
      mux MUX0 (IN0) {VT_references: v2.x1};
      mux MUX1 (IN0) {VT_references: v2.x1};
      fwrite REG1 {VT_references: v2.x1};
      and PROC0 {VT_references: v2.x1};
    }
    cstep [2:2] {
      mux MUX0 (IN0) {VT_references: v2.x2};
      mux MUX1 (IN1) {VT_references: v2.x2};
      and PROC0 {VT_references: v2.x2};
      fwrite REG2 {VT_references: v2.x2};
    }
    cstep [3:3] {
      mux MUX0 (IN1) {VT_references: v2.x3};
      mux MUX1 (IN0) {VT_references: v2.x3};
      plus PROC0 {VT_references: v2.x3};
      fwrite REG2 {VT_references: v2.x3};
    }
end;
```

Figure 13. DAA mapping of control specifications for single value trace operations. References correlate implementations of Figures 4 and 7.

Storage elements are bound to VT-body inputs and outputs and inputs of value-trace operations. A register has two ports— an input port and an output port—while a memory has three, an address port, a data-in port, and a data-out port. Operations consume values that are either produced by other operations or are inputs to VT bodies in what corresponds to a read operation. If the value is bound to a storage element, the input that consumes the value is associated with the output port of the storage element. A value produced by an operation is stored in an element bound to the output, such that it maps into an input port of the storage element.

Figure 7 shows an example of how relationships between registers and values are maintained at the functional-block level. There, under the heading "Mapping," is a partial list of the relationships between between hardware elements and value-trace values. REG1 is used to hold the output of OP0. According to the previous definition, the input port of REG1, IN, maps into the output value of OP0, OP0.out. Also, the output port of REG1, OUT, maps into the first input of operation OP2, OP2.in0.

Transfer paths route values from one operation to another. Values enter and leave through their input and output ports, to which values are bound. For example, if multiplexer input ports are bound to the outputs of different operations, they map into the values produced by these operations. The output port of this multiplexer maps into the union of all the values at the input ports. Figure 6 shows such a multiplexer fed by the outputs of two different processors, OP1.out and OP2.out. The partial multilevel mapping given in Figure 7 also shows how the multiplexer output port maps into the union of the input values.

Processors may be bound to more than one operation if the operations have been assigned different control steps. The mapping rule for processors is simple: the input ports of a processor map into the input values of the operations bound to it, while the output ports map into the outputs of the corresponding operations and the inputs of the operations that they feed. Figure 5 shows a processor, PROC0, bound to operation OP0. Port IN0 of PROC0 maps into the left input of OP0 (OP0.in0), while port IN1 of PROC0 maps into the right input of OP0 (OP0.in1). Port OUT of PROC0 maps directly into these inputs and the left input of OP2.

Behavioral features at the functional block level are modeled by control steps that correspond to register and memory read/write operations; to port-selection operations for multiplexers, demultiplexers, and buses; conditional constructs that represent multiway branches, procedure calls, and returns; and loop instantiations. At the value trace level, behavioral features are represented by control flow constructs such as SELECTS, VT-body instantiations, transfer of control operations, and array reads/writes. Each VT body maps into a control step block at the functional block level. All operations explicitly represented at the value-trace level map into the corresponding micro-operations at the functional-block level. Port selection for transfer paths does not have ex-

plicit representation in the value trace. They map into the operation that is active when the port is selected. To illustrate, Figure 13 shows the control specification for the allocation shown in Figure 7. The control step block, COMPUTE, maps into the VT body, v2, of Figure 4. In control step [1:1], input IN0 of MUX0 and input IN0 of MUX1 (values *a* and *b* respectively) are directed through to PROC0, which ANDs the values. The result is written into REG1. Control step [2:2] ANDs the value *a* with 1 writing the result into REG2, and control step [3:3] adds the values in REG1 and REG2, leaving the result in REG2. Thus the behavioral references correlate the specific implementation proposed in Figure 4 and illustrated in Figure 7 and in Figure 13.

These interlevel behavioral and structural references maintain the correlation between a value trace and functional block implementations. The multilevel representation generated by the synthesis programs can then be used by analysis aids, such as multilevel simulators, and subsequently by verification programs. It can also provide feedback to system level architects about the results of automatic or computer-aided synthesis programs.

Although there is much research still left to be done, design aids for the creative IC design process are becoming a reality. The portions of the CMU-DA system demonstrate promising design approaches and will serve in the future for further research. The main areas to be explored include the interaction of KBES with more algorithmic design methods, the development of better system-level description languages, and the use of multilevel representations, which includes multilevel analysis aids. These synthesis and analysis design aids will eventually form an effective environment for the system-level designer. ■

Acknowledgments

The authors express their gratitude to S. W. Director, D. P. Siewiorek, and M. J. McFarland for their advice on the project; also to J. A. Nestor, who contributed extensively to our early work on multilevel representations. This research was funded in part by grants from the Digital Equipment Corporation, IBM Corporation, and the Intel Corporation, the Semiconductor Research Corporation, the National Science Foundation (grant ENG 78-25755), and the Army Research Office (grant DAAG/29/79/C/0213).

References

1. S. W. Director et al., "A Design Methodology and Computer Aids for Digital VLSI Systems," *IEEE Trans. Circuits Systems*, Vol. CAS-28, No. 7, July 1981, pp. 634-645.
2. D. E. Thomas, "The Automatic Synthesis of Digital Systems," *Proc. IEEE*, Vol. 69, No. 10, Oct. 1981, pp.1200-1211.
3. L. J. Hafer, "Automated Synthesis of Digital Hardware," *Proc. IEEE Trans. Computers*, Vol. C-31, No. 1, Jan. 1982, pp. 93-109.

4. G. Zimmermann, "The MIMOLA Design System: A Computer-Aided Digital Processor Design Method," *Proc. IEEE 16th Design Automation Conf.*, 1979, pp. 53-58.

5. M. R. Barbacci, "Instruction Set Specifications (ISPS): The Notation and its Applications," *IEEE Trans. Computers*, Vol. C-30, No. 1, Jan. 1981.

6. T. J. Kowalski and D. E. Thomas, "The VLSI Design Automation Assistant; Prototype System," *Proc. 20th Design Automation Conf.*, June 1983.

7. C. Y. Hitchcock III and D. E. Thomas, "A Method of Automatic Data Path Synthesis," *Proc. 20th Design Automation Conf.*, June 1983.

8. M. R. Barbacci and D. P. Siewiorek, "Evaluation of the CFA Test Programs via Formal Computer Descriptions," *Computer*, Vol. 10, No. 10, Oct. 1977, pp. 36-43.

9. E. A. Snow, "Automation of Module Set Independent Register-Transfer Level Design," PhD dissertation. Carnegie-Mellon University, April. 1978.

10. M. C. McFarland, "The Value Trace: A Data Base for Automated Digital Design," master's thesis, Carnegie-Mellon University, Dec. 1978.

11. D. A. Gatenby, "Digital Design From an Abstract Algorithmic Representation: Design and Implementation of a Framework for Interactive Design," master's thesis, Carnegie-Mellon University, Oct. 1981.

12. R. A. Walker and D. E. Thomas, "Behavioral Level Transformation in the CMU-DA System," *Proc. 20th Design Automation Conf.*, June 1983.

13. M. C. McFarland, S. J., "Allocating Registers, Processors and Connections", Internal Carnegie-Mellon University Report, Pittsburgh, Pa., Aug. 1983.

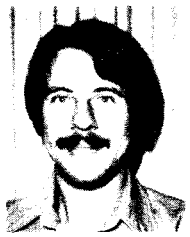
14. L. J. Hafer, "Data-Memory Allocation in the Distributed Logic Design Style," master's thesis, Carnegie-Mellon University, Dec. 1977.

15. C. L. Forgy, "OPS5 User's Manual," Department of Computer Science, Carnegie-Mellon University, July 1981.

16. A. Newell, *Heuristic Programming: Ill-Structued Problems*, John Wiley & Sons, New York, 1969.

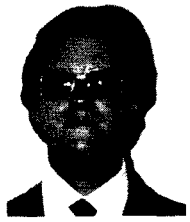
Donald E. Thomas is an associate professor of electrical and computer engineering at Carnegie-Mellon University. He received a PhD from CMU in 1977. His research interests include the automatic synthesis of digital systems, system-level description languages, and multilevel simulation. He currently serves on the IEEE program committee for the Design Automation Conference, as vice chairman of the Design Automation Technical Committee, and as an editor of *IEEE Design and Test of Computers*. He is also active in the design and use of high-speed networks for computer-aided instruction.

Charles Y. Hitchcock III is pursuing research in computer architectures as a doctoral candidate in the Carnegie-Mellon University's Department of Electrical and Computer Engineering. He graduated with honors in 1981 from Princeton University with a BSE in electrical engineering and computer science and received an MSEE from CMU in 1983. He is a member of the IEEE and ACM.



Thaddeus J. Kowalski is a doctoral candidate and IBM fellow in the Carnegie-Mellon University Department of Electrical and Computer Engineering. He is currently researching the use of knowledge-based expert systems in computer architecture, but his interests also include artificial intelligence, operating systems, real-time systems, and computer architecture. He graduated from the University of Michigan with a BSE in computer engineering in 1977 and joined the Unix Support Group at Bell Laboratories, Murray Hill, NJ. He received an MSEE in computer engineering at CMU in 1978. He is a member of the Vaulcans engineering-honor-service society, Eta Kappa Nu, and Phi Beta Kappa.

Jayanth V. Rajan received a bachelor of technology degree in electrical engineering from the Indian Institute of Technology, Madras, in 1980 and an MSEE from Carnegie-Mellon University in 1982. He is currently a doctoral candidate in CMU's Department of Electrical and Computer Engineering. His research interests include VLSI chip planning and synthesis.



Robert A. Walker is a doctoral candidate in the Department of Electrical and Computer Engineering at Carnegie-Mellon University, where he is researching hardware languages as behavioral descriptions for logic synthesis. His research interests include optimizing transformations, computer architecture, and programming language issues. In 1981, he graduated from Tennessee Technological University with a BS and received an MS degree in computer engineering from Carnegie-Mellon in 1982. He is a student member of the IEEE, the IEEE Computer Society, ACM, Eta Kappa Nu, and Tau Beta Pi.

Questions about this article can be addressed to the above authors at the Electrical and Computer Engineering Department, Carnegie-Mellon University, Pittsburgh, PA 15213.