Silicon Compilation of Very High Level Language

Mark Kahrs, *Member, IEEE*

Abstract—**This paper concerns the design and implementation of a compiler for two very high level languages. The first language is a set language similar to VERS and SETL. The second language is a novel signal processing language. The compiler uses data flow and type information to constrain possible choices before choosing a possible implementation. Heuristic search is then used to choose from competing concrete implementations of abstract data types. Constraint propagation is used at every selection step to remove incompatible configurations from the search, thereby reducing the search space considerably. A microprogram control store is automatically generated. The output of the compiler is a parts list, a net list of module interconnections, and the fields of the control store.**

I. Introduction

A. Goals

THE TERM "silicon compilation" was coined by Johannsen [1] in 1979, and has come to mean almost any design system that aims to produce silicon from anything other than mask data. Gajski, Dutt, and Pangrle [2] further divide silicon compilation into three categories: structural, behavioral, and intelligent silicon compilation. Structural compilation (sometimes called "silicon assembly" [3]) refers generally to layout description given basic modules. Silicon assembly makes the placement and routing easier but does not hide behavior from the user. Behavioral compilation aims to produce a circuit that has the same behavior (mapping of inputs to outputs) as the input program. Past work on behavioral compilers has concentrated on compiling "algorithmic" input languages like Pascal [4], [5] or hardware description languages like ISPS [6], [7]. While specifying algorithms at this level level is comfortable for some, it can expose many details of machine design. For a unknowledgable user designing an application specific integrated circuit (ASIC), this is unreasonable. This is due in part to the description level of the input language. Very high level languages can hide all of this complexity in the system by forcing the user to specify the algorithm at a very high level via abstractions. The last category of Gajski *et al.*, the intelligent silicon compiler, represents the end goal of silicon compilation research. Intelligent compilers will use multiple design styles (instead of specializing in one style), evaluate designs in progress, and perform refinements.

Manuscript received August 23, 1988; revised September 5, 1991. This work was supported in part by the National Science Foundation under Grant IST-8012418 and under Grant MCS-81004008, and by DARPA under Grant N00014-78-C-0164. This paper was recommended by former Editor M. Lightner.

The author is with the Department of Electrical and Computer Engineering, Rutgers University, Piscataway, NJ 08855-0909.

IEEE Log Number 9200549.

This paper describes a system that is halfway between a behavioral compiler and an intelligent compiler. It compiles programs written in a very high level language into a description of module interconnections. The modules are chosen from a user supplied library. They are presumed to have been designed using lower level design tools such as lower level silicon compilers, assemblers, and graphic editors. The final output from the compiler is a net list of module interconnections as well as a parts list and a listing of the microcode fields for the machine.

B. Very High Level Languages

As stated above, an unknowledgable silicon compiler user should be able to create a circuit without specifying all the details of algorithmic implementation. Earley [18] identified three criteria for the design of (very) high level programming languages. They were as follows:

1) ability to write a program in a clear and concise manner,
2) ability to ignore the implementation issues and concentate on the semantics and correctness of the algorithm,
3) ability to postpone design decisions on seemingly unrelated portions of the program until needed.

Of these three points, the second is of critical interest because the user of a high level silicon compiler *should* ignore the implementation issues of allocation, scheduling, and optimization and concentrate instead on the system design issues.

Past work in very high level languages principally has been done in languages with abstract types such as sets, tuples, and relations (an example of such a language is SETL [9]). The use of high level abstract types intentionally obscures common programming details such as pointer chasing, memory allocation, structure formation, and implementation selection.

Two very high level languages were defined: "YASL" (Yet Another Set Language) and "CLASP", a very high level signal processing language [10]. YASL resembles SETL: it includes high level types and the associated iterators, while CLASP describes signal processing algorithms using filters and transforms.

C. Compilation Strategy

The compilation process can be divided into four basic stages, shown in Fig. 1. The first stage is analysis, in-

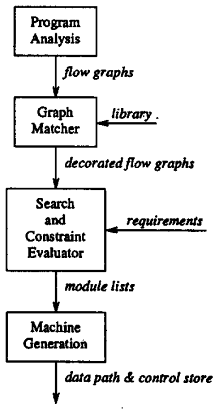

Fig. 1. Compilation process.

cluding control and data flow analysis as well as type determination. The second phase is the matching of the data flow graph with the data flow graphs in the library description. The third phase searches through the possible matches for a covering of the data flow graph while constantly evaluating constraints at all levels. The fourth and final stage generates a machine by generating a control store as well as the data paths.

The compiler analyzes the input program for data flow (data transfer between storage and computational elements) and control flow (transfer of control from one statement to another). The decomposition of an input program into data and control flow has its roots in the analysis of programming languages (see [11]). It has also been used by many in silicon compilation, beginning with Snow [12] and including the CADDY/DSL [13] group and the DDS [14] system. The program is also examined for types and other properties. These properties are attached to the data flow nodes of the program. Given the data flow graph, some sort of optimization would be useful (for example, performing algebraic simplification). However, this step can be omitted with the subsequent loss of efficiency. The data flow graph is "matched" with the data flow graphs of the parts in the parts library. This differs from other approaches, including greedy allocation [15], clique partitioning [16], critical path allocation [17], and direct compilation [13]. For example, consider the following CLASP program.

```
module TouchToneDecoder

-- The now classic touch tone decoder, as done originally in 1963 by
-- a group in Bell, then done again in 1968 by Jackson et al. and
-- done again by Lyon.

declare tuple of integer : lowerBand, upperBand;
declare tuple of integer : lowerBandCenterFrequencies;
declare tuple of integer : upperBandCenterFrequencies;
declare tuple of integer : detection;
declare integer : result; -- output from hum filter (and iteration variable)
declare integer : bandLimit; -- bandpass (and lowpass) band limit
declare integer : input; -- input from the A/D
declare integer : output; -- output from the module
declare filter from 180 to INFINITY : noHum; -- Line hum filter

lowerBandCenterFrequencies := [ 697, 770, 852, 941 ] ;
upperBandCenterFrequencies := [ 1209, 1336, 1447, 1663 ] ;

result := noHum(input);
lowerBand := filter result from DC to 1070 : lowerGroupFilter;
upperBand := filter result from 1070 to INFINITY : upperGroupFilter;

detection := phi;
foreach centerFrequency in lowerBandCenterFrequencies do
        detection := detection plus
                filter HalfWaveRectifier
                    filter lowerBand
                        from centerFrequency-bandLimit
                        to centerFrequency+bandLimit
                            with stopband attenuation to 16 db down
                              : lowerBandPass)
```

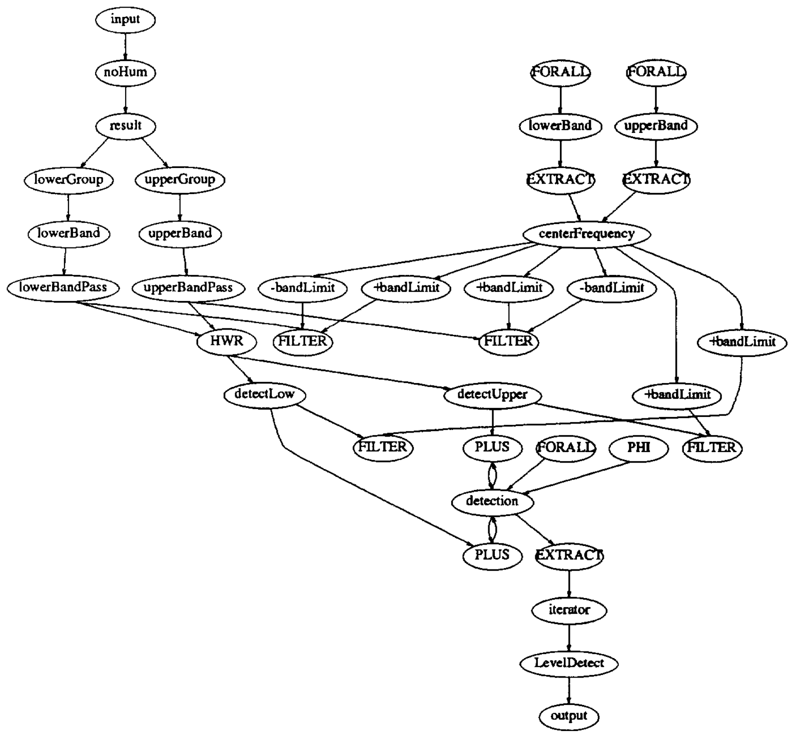

Fig. 2. CLASP data flow graph.

```
                    from DC
                    to centerFrequency+bandLimit
                : detectLowGroup;

foreach centerFrequency in upperBandCenterFrequencies do
        detection := detection plus
                filter HalfWaveRectifier(
                    filter upperBand
                        from centerFrequency-bandLimit
                        to centerFrequency+bandLimit
                            with stopband attenuation of 16 db down
                              : upperBandPass)
                    from DC
                    to centerFrequency+bandLimit
                : detectUpperGroup;

foreach result in detection do
        output := LevelDetect(result)

end.
```

This program is a touch tone receiver written in the language "CLASP". The input library for CLASP contains descriptions of various signal processing primitives, including filters and miscellaneous functions like rectifiers, limiters, and detectors.

This will be analyzed by the flow analysis code and will produce the following data flow and control flow graphs shown in Figs. 2 and 3, respectively.

The compiler tires to find a complete covering of the data flow graph. There can, of course, be many such coverings, since there can be different matches for the various parts. For example, in the above CLASP example, each

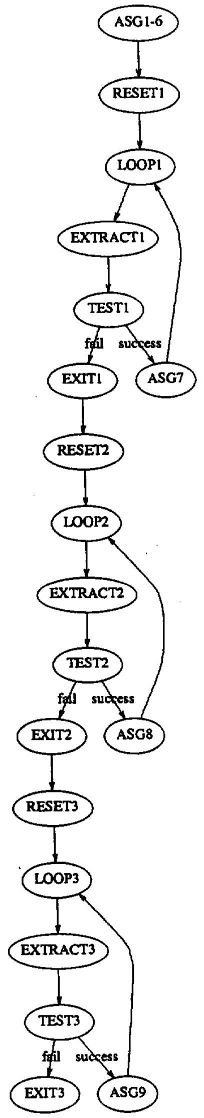

Fig. 3. CLASP control flow graph.

filter will be matched by all the filters in the library (Butterworth, Chebychev, and so forth). A separate phase, called "binding," evaluates the filter parameters and calculates the required filter order for the implementation.

A search phase is used to find optimal combinations of the subgraphs. At each stage of the search, local and global constraints are evaluated. This is reminiscent of the Emucs [15] step by step evaluation of costs. At the conclusion of the search, a microcoded machine is generated. The control store is generated from the control flow graph much as [13] did (concurrently with this paper).

D. Paper Overview

Preliminary analysis of the input program is assumed to have been performed. The matching phase is described first; here, the data flow graph of the input program is matched against the library. The next section describes how constraints will be used during search to cut the search space. The search process is described in detail, followed by a section on how to generate a machine from the resulting information. Lastly, the implementation is described, along with conclusions and directions for future work.

II. Matching

Implementation choices are made by searching through a library of template descriptions that describe the behavior (semantics) of the available modules. The object of this search is to *match* the modules with parts of the machine "generated" by the input program. The matcher described in this section matches the data flow subgraphs of the library modules against the program data flow graph. Matching uses a table of correspondences from data flow nodes to possible implementations that use that node. It tries to match the data flow subgraphs of the implementations in the library, starting from each node in the data flow graph. Since the library modules can be parameterized, a process called "binding" is used to instantiate the library templates. After binding, the library templates (now called *instances*) are ready to be evaulated for time and area bounds.

A. Matching the Library

The problem of choosing implementations begins by matching the library with the program. The data flow graph provides a reasonable representation for the matcher to work with because of the following:

1) A data flow graph is language independent, thus isolating the definition of the library from the language.
2) The data flow graph "fits" the problem of matching parts of the machine with parts from the library.
3) Data flow graphs can be easily transformed into data paths. This is in opposition to tree matching, where matched trees must still be transformed into circuits (i.e., graphs).

In its purest form, subgraph matching is NP-complete. However, here the problem is more constrained: each node in the data flow graph has a label. These labels allow the graph matcher to run in linear time. Assuming matches are found, instantiation of the successful matches follows. Note that failure to match every node in the data flow graph is a *serious* failure, since this indicates that the library fails to cover the data flow graph of the input program. Currently, such failures are reported to the user and the program continues.

The library entries are parameterized templates because the modules are often of variable size (where the size depends on the values of bound parameters). For example, the size of a bit vector representation of a set is dependent on the maximum number of items in the set. Instantiation binds properties in data flow nodes to parameters in the library specification for the matched modules. After binding, it is possible to evaluate each instance and begin searching for the set of instances that will satisfy both the design goals and semantics of the input program.

B. Graph Matching

The matcher tries to match every subgraph of each module description in the library with the program data flow graph for each node in the graph. The language used to specifiy the match graph is nearly the same as the language used to generate the data flow graphs.

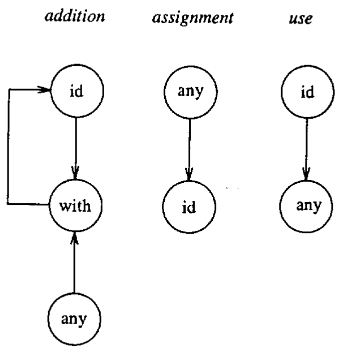

Fig. 4. Data flow subgraph of the parallel bit vector module.

1) Library Representation: A module may have more than one data flow subgraph because a module may compute more than one result. This is the case with many modules that have control signals that dictate which function is computed. For example, a set representation may have size, add, delete, and membership functions in the same module. Therefore, it follows that the library representation should reflect these different functions.

Different functions for a given module are described in terms of the control signal bindings. So for each binding of the control signals of a module, a data flow subgraph characterizes the behavior of the module, given those control signals. For example, the representation of the parallel bit vector set representation in the example has three functions: addition, assignment, and use. The data flow subgraphs of this module are shown in Fig. 4.

These are represented in the library as follows.

```
ParallelBitVector
        (variable n)
        (inputs (inputData n) (clock 2) (operations 2))
        (outputs (outputData n))
        (control (operations 2))
        (properties
                (inputData parallel integer) (outputData parallel integer)
                (clock clock) (operations control)
        )
        (area
                (width (times 20 n))
                (height 100)
        )
        (time
                (delay (lookup delay))
                (period)
        )
        (power (times 25 n))
        (parts
                (use
                    (control (operation 0))
                    (graph (attach-tail (port outputData (node IDENTIFIER SET INTEGER))
                      (node ANY)))
                    (timing (delay 1))
                    (bind (n outputData (second range) range-size-in-bits))
                )
                (assignment
                    (control (operation 1))
                    (graph (attach-tail
                              (node ANY)
                              (port inputData (node IDENTIFIER SET INTEGER))))
                    (timing (delay 1))
                    (bind (n inputData (second range) range-size-in-bits))
                )
                (insertion
                    (control (operation 2))
                    (graph
                        (loop)
                            (attach-tail
                                (join (node +)
                                        (loopnode) (port inputData (node ANY))
                                )
                                (node IDENTIFIER SET INTEGER)
                            )
                        )
                    )
                    (timing (delay 2))
                    (bind (n inputData (second range) range-size-in-bits))
                )
            )
    )
```

Briefly, the beginning of the definition defines the input and output ports of the modules. They are parameterized by the variables declared. The types of the ports are declared by the properties section. The criteria for global performance parameters such as time, area, and power are declared next; note how they can be parameterized by the variable n. The parts section declares the different sections for each different control operation. These subsections include the graph to be matched and how to calculate the value of the variables. The exact syntax and semantics of the data flow graph description are not of critical importance; the full details can be found in [18].

2) Matcher Operation: So that the matcher has an idea of where to start matching, each library subgraph has an "entry node" that the matcher uses to start traversing the data flow graph. A table gives the correspondence between node labels and the library entries with those entry node labels. In the previous example, the entry nodes would be the `ANY` node in `USE` and `IDENTIFIER` node in `assignment` and `insertion`.

The matcher starts by iterating through the symbol table. The graph matcher proceeds from node to node in a depth first manner by following both forward and backward arcs. It works backwards from the terminal nodes to the interior of the graph. Each match is recorded and associated with the library implementation and the specific control bindings that would cover the data flow subgraph. It stops traversing a branch (arc) of the graph whenever:

1) A node has been already traversed [success].
2) A node label does not agree [failure].
3) A node type does not agree [failure].

Note also that in case (3), node types are a direct result of property determination and declaration. The use of node types permits the matches to be restricted by data type.

Rather than backtrack whenever a failure is detected, the whole matcher returns a failure. This has two consequences: First, the matcher runs in linear time. Second, any part that requires more than one match must be detailed in the library. A success, on the other hand, indicates that the matcher has gone as far as it can go and that other paths should be pursued. It should also be obvious that no node will be traversed twice. Also, a special node label `ANY` will match any node (a so-called "wild card" match).

Nodes can have both labels and types. Node types are a result of property determination and declaration. Node types permit the matches to be restricted by type. This is typically used in matching typed identifiers (found in most programming languages) with specific typed implementations.

3) The Matching Algorithm: The matching algorithm assumes the existence of the following data structures:

1) A symbol table, mapping symbols to nodes in the data flow graph.
2) A data flow graph.
3) A control flow graph.
4) A table mapping symbols to possible implementations and subparts of the implementations.

The matcher starts by sequencing through the symbol table. The graph matcher works backwards from the terminal nodes to the interior of the graph. Each complete subgraph match is recorded and associated with the implementation and the control signal bindings that match the data flow subgraph. It is also recorded on each node of the match. Therefore, it is a trivial matter to calculate the number of matches on any node (of course, a node with zero matches is uncovered). The matcher algorithm follows, written in a pseudoset language.

```
;The Match procedure matches up the data flow nodes in the data flow
;graph with the data flow subgraphs in the library. Its side effect
;is to create match nodes that detail these matchings.

procedure Match( ) is
    ; symbols here include both identifiers and operators
    foreach symbol in the symbol-table do
        foreach data-flow-node bound to the symbol do
            ;Find all the possible implementations by looking them up in the
            ;symbol to implementation table. Here "implement" means module and
            ;"part" means the name of the function of the module
            ;(i.e., module=ALU,part=+)
            foreach implement, part in the
                symbol-to-implementation-table[data-flow-node] do
                    Match-from-node(data-flow-node, implement, part)
            end foreach
            ;Now match using the ANY nodes as well
            foreach implement, part in the
                symbol-to-implementation-table[ANY] do
                    Match-from-node(data-flow-node, implement, part)
            end foreach
    end for
end Match

;Match-from-node tries to establish a match starting from data-flow-node
;to the implementation "implementation" using the part name "part-name".

procedure Match-from-node(data-flow-node, implementation, part-name)
    ;Matches are specific to a given control flow node because otherwise
    ;the matcher would try combinations that weren't actually used.
    foreach control-flow-node of the data-flow-node do
        ;Each "sub part" of a implementation (library module) has a graph (sub-graph).
        ;This is matched against the program data flow graph. If successful,
         it creates a match node.
        foreach sub-graph of the implementation[part-name] do
            if Match-graph-from-node(data-flow-node, control-flow-node, sub-graph)
            then
                  Match-Check (Create-match-node (sub-graph, implementation,
                  part-name))
        end foreach sub-graph
    end foreach control-flow-node
end Match-from-node

;Match-graph-from-node tries to match the data flow graph "graph"
;starting from the node "data-flow-node".
;The matcher succeeds only if the nodes were used by "control-flow-node".
;The exact details of the graph matcher are omitted due to the dependency on the
;graph representations.

procedure Match-graph-from-node(data-flow-node, control-flow-node, graph)
    if the control-flow-node is in the control-flow-nodes of
       the data-flow-node then
            (match the graph from the description using depth first search)
end Match-graph-from-node

;Match-Check tries to determine whether the match is just a new
```

```
;part of an already existing implementation. If it is a new part,
;the it returns the instance of the old implementation, else nil

procedure Match-Check(match-node)
    ;Search all the possible matches (the union of all the matches of all the nodes
    ;in the new match-node).
    foreach other-match-node in the union of matches of the nodes in the
        data-flow-graph of the match-node do
        ;Make sure the implementations match
          if the implementation of the other-match-node =
             the implementation of the match-node AND
             ;If so, then if the nodes of the match-node overlap (intersect)
             ;with past matches
             ; (i.e., matches already made), then return the old instance
             the set of nodes in the graph of the match-node intersect
             the set of nodes of past-matches of the implementation of the match then
                 Add-match-to-instance(match-node, instance of other-node)
    end foreach other-match-node
end Match-Check
```

It should be noted that the graph matcher can find multiple operations covered by a single part in the library. Each subpart also has the relevant control signal binding described in the part declaration. This is relevant to the synthesis of the control section.

C. Binding and Instantiation

After matching, each variable in the matched subgraph must be instantiated by binding the appropriate values to the template variables. As a result of property extraction and type declaration, each identifier node in the data flow graph has a type and other properties attached to it. These properties are used to bind variables.

The binding between the template variables and the properties is specified as part of the library module definition in the library as a 3-tuple: the variable name, the property and the interpretation function. If the interpretation function is absent, then it is assumed to be the identity function, i.e., no interpretation is done. An example of a useful interpretation function is one that takes an integer range and interprets it as a size. Such an example can be found in all the `bind` definitions of the library example.

III. Constraints

A. Use of Constraints in VLSI Design

1) Introduction: Constraints are present at all levels of VLSI design. Table I illustrates four possible levels of constraints in a simplified design hierarchy. This is not meant to be all encompassing, but rather to show differing levels of constraints in the design.

At the bottom level, constraints called design rules specify the minimum spacing of lines. The next level up is the transistor level; e.g., transistor t_1 must have a ratio of 2 : 1 with transistor t_2. The next level up are cells. Typical intercell constraints are of the form "port q does not have drivers, therefore it must be close to its sink." Cells make up modules, and module constraints specify properties. For example, "module M, port input A takes parallel, two's complement integers." At the top level, modules are connected together to form systems. The constraints at the system level are global performance constraints. Examples of global performance constraints include area, critical path time, computational period, and so forth.

The lowest level of representation in this compiler is the module layer. The intermodule constraints have a local nature; they exist between ports of the modules that are connected by data paths. These will be called "port constraints."

There are also more global constraints. These constraints express the high level performance and resource bounds. These will be called "specification constraints."

Finally, there are constraints that are specific to modules being matched. If these constraints are satisfied during matching, then the module is matched; otherwise, the module is not matched. These constraints will be called "matching constraints."

The following three sections discuss the algorithms used to check port, specification, and matching constraints.

2) Port Constraints: There are two ideas behind the use of port constraints. The first purpose of the port constraints is to ensure that other modules connected to the port will be able to "talk" to the module (i.e., the ports share common typing). Note that every module in the library has semantics (or properties) associated with each port. If these semantics can be matched, then the connection is valid.

The second purpose of port constraints is to establish data paths between modules. When a module description is given to the system, there are no indications about which ports of which modules can connect to a particular port. Therefore, one method of connecting ports of modules is to match the types of the ports.

Specifically, consider the use of program properties

TABLE I
CONSTRAINT LEVELS

Level	Constraint
circuit	performance constraints
module	signal properties
cellular	intercell rules
transistor	transistor sizing
mask	design rules

TABLE II
YASL CONSTRAINT MATRIX

Type	Serial	Parallel	Integer	Float	Character
parallel	DEAD	OK	OK	OK	OK
serial	OK	DEAD	OK	OK	OK
integer	OK	OK	OK	DEAD	OK
float	OK	OK	DEAD	OK	DEAD
character	OK	OK	OK	DEAD	OK

such as types in selection. If port constraints are defined as binary relations over types, then the constraints can be expressed as the relation *equals*(t_1, t_2), where t_1 and t_2 are the types of ports. The actual type matching takes place during selection. Each type of the ports on the newly selected module is traced backward to the connecting module. If the module is not selected, then the port is deemed acceptable; the type checking will take place when the other module is selected. If the other module *has* been selected, then the type of the port on the other module is compared (using the relation "type-compare") with the type of the port on the newly selected module. If they match, then the selection is permitted to proceed. Otherwise, the selection is put into the "reject bin."

For example, take the case of the two nodes n_1 and n_2, each with a parallel and serial implementation. Assume that n_1 and n_2 are connected, i.e., there is a data path between them. During the search phase of selection if n_1 is selected first, then attempts to propagate the port constraints will fail, as n_2 is not instantiated yet. When the search reaches n_2, then the port types of n_1 and n_2 are checked, since they will both be instantiated.

Note that matching also involves searching—each port of every selection must be matched against every other port of the connected selections. For example, suppose port A has types t_1 and t_2 and port B has types t_3 and t_4. Then type t_1 is checked with type t_3 and then type t_4. Likewise, this will happen with type t_2. The demonstration program implements this by using a depth-first search.

There is a close correspondence between satisfying local constraints, such as port constraints, and the "consistent labeling" [19] problem of classical artificial intelligence. For each choice made by the selection algorithm, the choice must agree with the choices already made. Furthermore, all of the succeeding choices must agree with the choice being made. Note that every choice restricts further choices by making the problem more constrained due to the new constraints.

Local constraints were used extremely successfully by Waltz [20]. Using a labeling scheme, he used the interaction between labels in a line drawing to drastically reduce the search space of interpretations.

The constraint matrix used by YASL is shown in Table II. OK means that the connection is fine, DEAD means that the combination is incompatible and should be thrown out. The port type constraint matrix is shown in Table III.

The algorithm for port constraint propagation follows.

```
;propagate-constraints propagates as many properties as possible in
;the data flow graph.
procedure propagate-constraints(match-nodes) is
    ;Loop through all the matches
    foreach match-node in the match-nodes
        ;Now search through the graph of the matched node looking for ports.
        foreach part of the graph of the match-node do
            if the part (of the graph) is a port then
                ;if the port is declared an INPUT port, then go backward
                ;through the graph
                ;(to the connecting connecting OUTPUT port).
                if the node is declared in the INPUT section then
                    foreach connecting-node in TraverseGraphFromNode
                    (node, BACKWARDS) do
                        propagate-properties(node, connecting-node)
                else
                ;if the port is declared an OUTPUT port,
                ;then go backward through the graph
                ;(to the connecting connecting INPUT port).
                if the node is declared in the OUTPUT section then
                    foreach connecting-node in TraverseGraphFromNode
                    (node, FORWARDS) do
                        propagate-properties(node, connecting-node)
end propagate-constraints
```

```
;propagate-properties tries to propagate all the properties of the
;port-node(which points to the library via its matches)
;to the connecting-node.

procedure propagate-properties(port-node, connecting-node):
    ;First, make sure that they talk to each other at the same time.
    if the control-node of the match-node of the port-node =
       the control-node of the match-node of the connecting-node
    then
        ;loop through all the connected matches,
        ;making sure their instance is selected.
        foreach match-node in match-nodes of connecting-node do
            if the instance of the match-node of the connecting-node
               is in the instances of the search-node then
                 PropagatePropertiesFromNodeIntoNode(port-node, connected-node);
end propagate-properties.

procedure PropagatePropertiesFromNodeIntoNode(from-node, to-node) is
    ;This is only concerned with matching ports
    if the to-node is a port then
        if property-compatibility(from-node, connected-node) then
            AddPropertiesFromNodeToNode(from-node, connected-node)
               return success
            else
               return failure
end.

;property-compatibility tests to see if the properties of two data
;flow nodes are "compatible". This is done with a simple table lookup.

procedure property-compatibility(from-node, to-node) is
    if compatibility-Table[from-node, to-node] = OK then return success
    else return failure
```

Note that the procedure `TraverseGraphFromNode` goes backward or forward one link in the data flow graph depending on "direction." `AddPropertyFromNodeToNode` adds the property list of the first node to the second.

3) Matching Constraints: After a module's data flow subgraph has been matched with the program data flow graph, there are further constraints that may require testing. Specifically, a module may have certain use requirements that must be satisfied before the module is finally selected. A good example of this is the signal processing domain where different implementations of a filter have different performance characteristics (noise, sideband suppression, Q, etc.). These performance criteria are stated as part of the module specification and are checked before being officially matched.

As an example, take the implementation of a set using linked lists. One criterion (constraint) for selection might be "use this if the number of items in the set will exceed 100." This would be specified as part of the module specification as `(constraint (> size 100))`. The actual checking of matching constraints is done during search (implementation selection).

4) Specification Constraints: Specification constraints are specified by the user of the system before selection begins. These constraints reflect the goals of performance and resource usage. For example, a designer may want a design to fit in a definite amount of area or for certain procedures to be performed in a certain amount of time. The former is an example of a resource constraint (area < area-bound); the latter is an example of a performance constraint (time-for-function < time-bound). Note that both of these constraints are taken to be musts; any design created by the system *must* satisfy these constraints.

But what happens if a selection is made that violates these constraints? There are two choices: 1) discard it and 2) try to change the design into a workable one which satisfies the constraints. The "massaging" of the designs is largely an unexplored area. Darringer [21] used very local transformations for very low level logic designs. The DSL group [13] and Trickey [4] used standard compiler transformations such as loop unrolling. Snow *et al.* [22] applied transformations to value trace graphs as a form of global optimization. A similar proposal is the use of constraint triggered "critics"[1] [23] as an "on demand" method of optimization.

[1]The term "critics" is a term from Artificial Intelligence for heuristic methods used to fix bugs in plans. See Sussman [24].

TABLE III
PORT-TYPE CONSTRAINT MATRIX

Type	Input	Output
input	DEAD	OK
output	OK	DEAD

Specification constraint checking, like local constraint propagation, is done with each implementation selection. Unlike local port constraints, specification constraints are binary relations between a resource and a fixed, measurable bound. The implementation of specification constraint checking is discussed in the section on search.

The computation of time and area bounds in the VLSI domain are complex and can be only approximated in the implemented system because of the lack of layout knowledge. Area is currently measured by simply adding the area of the selected implementations to the current total. Realistic time bounds are *much* more complex; the program just adds up module delays. This is not sufficient, since what is really required is a notion of critical path. The lack of good timing measures is discussed in greater detail in the conclusion.

IV. Search and Implementation Selection

Once the matcher has found viable implementation choices (including resolution of port and matching constraints), the next step is to somehow choose implementations that cover every data flow node in the program. By using a modified breadth first search, the search phase can return with *multiple* designs. Existing compilers return with a single "best" design; this search returns with several.

The following sections consider:

1) how to search through the implementation choices.
2) how to evaluate possible implementation choices.
3) the effects of search and evaluation on the library description.

A. Selection Using Search

1. Introduction: At this stage, the matcher has found matches between the library and the program, so every node in the program's data flow graph should have a set of possible instances attached to it. Selection is the process of choosing among the instances attached to the data flow nodes. It is also responsible for checking constraints discussed in the previous section. The selection procedure works as follows: First, the nodes in the data flow graph are sorted by the number of instances are attached to the node. The list is sorted in order to the *smallest* to the largest. This permits the selection procedure to start from the most "obvious" (most constrained) choices and continue to the most "complex" choices. Next, the search proceeds from node to node and for each instance attached to that node

1) Checks if this instance has already been selected by another node.
2) Checks port types of the new instance.
3) Checks for overlap of the new instance.
4) Evaluates the costs of the new instance.
5) Adds these costs to the costs of the already chosen instances.
6) Checks each new choice to see if it violates design constraints (and calls the appropriate critics if it does).

The first step makes certain that the choice has not already been made. This can happen if the instance involves two or more data flow nodes, and some other node has already been selected before the node being expanded. This is perfectly permissible and no further evaluation is done.

The second step checks the type compatibility of the new instance and the instances that it "talks" to. If conflicts exist, then the instance is not chosen.

The third step is necessary to ensure that the new instance does not use any of the same terminals as any of the existing instances. This prevents multiple representations for the same variable.

The fourth and fifth steps evaluate the resources now consumed by the selections made so far.

The last step, step six, ensures that the new addition does not cause the generated machine to exceed design requirements. As a side effect, a possibly inefficient machine may become optimized in order to meet the design requirements given by the user.

The following is the pseudocode of the search procedure:

```
procedure Search( ) is
    old-node-list := nil;
    ;First, sort according to the number of possible implementation choices
    ;("instances")
    sort search-nodes by number of instances into node-list;
    foreach node in node-list do
        new-node-list := CrossProduct(node, old-node-list);
        ;Now, sort them by the metric so that the most promising ones are at the
        ;head of the list
        sort new-node-list by score;
        ;The user can select either full breadth first search (without pruning),
        ;a staged search with constant pruning or a contracting beam.
        switch search-type into
case CONTRACTING:
            truncate new-node-list at
                maximumBeamSize - level * beamIncrement;
            reject truncated nodes;
case STAGED:
            truncate new-node-list at maximumBeamSize;
            truncate new-node-list at maximumBeamSize;
case FULL:
        end; of switch
        old-node-list := new-node-list;
    end; of foreach
end Search.

;CrossProduct does exactly what its name implies - it returns the
;cross product of the input node and the list of nodes.

procedure CrossProduct(node, list-of-nodes) is
    ;If the list is starting out, initialize it
    if list-of-nodes = nil then return node
    else
    ;Next, check to see of the instance has already been chosen
    ;(It's possible that two data flow nodes can share an implementation)
    if the instance of the node is in
       the instances of list-of-nodes then ignore
    else
    if Overlaps(instance of the node, instances of the list-of-nodes)
       then ignore
    else
       return NewEntry(node, list-of-nodes)
end CrossProduct

;Overlaps checks to see of the data flow graphs of the instances overlap.

procedure Overlaps(instance, instance-list) is
    foreach other-instance in instance-list do
        if nodes of instance INTERSECT with
           nodes of other-instance then
             return success
        else
             return failure
end Overlaps


procedure NewEntry(new-node, past-nodes) is
    ;First, check to see if the node is already there
    if node is in past-nodes then return
    else
    ;If there aren't any other nodes, then create one for sure
    if past-nodes = NIL
    then
        create new-search-node;
        score new-search-node;
        ;Here is where properties are propagated
        if PropagateProperties(new-search-node) then
            reject new search-node;
        ;and global constraints checked (and maybe critics called)
        if ConstraintFailure(new-search-node) then
            reject new-search-node;
end NewEntry

procedure ConstraintFailure(search-node) is
    ;To check the global (performance) constraints, check each constraint against
     the design.
    ;If any constraint fails, then call all the critics associated with
     the constraint.
    foreach global-constraint in global-constraint-table do
        if Check-Constraint(search-node, global-constraint) then
           foreach critic in critics of global-constraint do
                CallCritic(critic)
    ;Now check the global constraints again; if they're still unreasonable then
     return failure...
    foreach global-constraint in global-constraint-table do
        if Check-Constraint(search-node, global-constraint) then
            return failure;
    return success
```

B. Search Techniques

1) Introduction: Search procedures can be broadly divided into backtracking and non-backtracking methods (A general overview of search techniques can be found in [25]). Both of these search methods have drawbacks: Backtracking searches (such as depth first search) can be expensive (in time costs), while breadth-first searches are exponential in space costs.

One solution, therefore, is to choose a search that can run in bounded time and bounded space. Breadth-first search can be modified to have a bounded search space by pruning the search space at every choice level. One advantage of breadth-first search is that it can return with more than one result.

2) Staged Search: Another way to surmount the expontential time and space of a breadth-first search is to expand only the most promising nodes at any stage. Lowerre [26] used this in the Harpy system and called it a "beam search." (The list of available nodes is called the "beam.") Nilsson calls it a "staged search." It was originally used by Doran and Michie [27] in a graph traverser. The problem with a staged search that it assumes that every step has the same pruning factor. However, when the search begins with the most constrained variable, there are very few choices. As the search proceeds, the number of choices expands. Therefore, the idea behind the *contracting* beam search is to permit extensive branching at first and to focus (i.e., contract) the beam as the search proceeds. The purpose of the contraction is to allow as many constraints to interact as possible during the beginning stages of the search, but as the search progresses, to count on constraint interaction to bring the search within bounds.

C. Metrics

1) Introduction: At each stage of the search, an evaluation function is called to assess the resources being used at the level of the search. These functions are called "metrics," and they guide the search by "measuring" the resources consumed by each collection of instances on the search "beam."

The design of a metric involves two factors:

1) fairness—the function should not permit unworkable solutions to achieve high scores.
2) accuracy—if possible, the metric function should return a value close to the "real world" resources consumed.

The last condition is required because the global (resource) constraints that the user provides are in terms that the user understands. Therefore, the system and the user must agree on the calculation of the metrics, otherwise the specification constraints will be either too high or too low. If the metric is overestimated, then the threshold for optimization will be exceeded too often and the circuit may be excessively optimized. It the metric underestimates, then the optimizer will not be called often enough and the resulting circuit will be inefficient.

There has been considerable theoretical work on complexity measures for VLSI. This can be used as a starting point for the derivation of real metrics.

2) VLSI Metrics: Most of the work in VLSI theory [28]–[30] uses a complexity measure of AT^2 where A is the area of the circuit and T is the time required to compute the result. This idea was extended to the digital signal processing domain by Cappello and Steiglitz [31] who used a complexity measure of ATP, where P is the period of the computation. Note that the period of a pipelined function is much less than the period for a nonpipelined function because of the higher throughput possible when the pipe is full. Therefore, this metric function favors pipelined implementations.

Wire areas are unknown until the placement and routing subsystem has been run. Therefore, it is not possible to obtain accurate figures of area consumption. As a result, the scheme behind the metrics actually used is to total the resources consumed by the non-wire portions of the machine (the modules) and estimate the wire usage. Use of a system like PLEST [32] would be extremely useful in obtaining reliable estimates of total chip area.

3) Actual Metrics: The section on binding detailed how the instantiated modules are used by the metrics to calculate the evaluation parameter. Each of these actions has an impact on the specification of the library.

As stated earlier, most library modules are parameterized so that the compiler can generate arbitrarily wide instances. The metrics also have an impact on module specification. Each module must have its height and width specified so that area can be computed. Of course, the height and width formulas can be parameterized with the module parameters and bound later. Area computation may also involve some overhead, so that must be included also.

As an example of the library specification details explained above, consider the example library. The same parallel bit vector set representation would have the following area calculation:

```
(parameter n)
(area (width (times 20 n)) (height 100)).
```

Likewise, timing parameters can be specified:

```
(timing (delay n)).
```

After the parameters have been bound, these functions can then be evaluated and used by the metric functions. The implementation used a metric function of

$$\sum_{i}^{N} A_i T_i^2$$

where A_i is the area of module and T_i is the time required to compute the result of module i. N is the number of modules. The latter is clearly a severe approximation.

V. Machine Generation

To recapitulate, the stage is now set for the actual generation of signal paths; the program has been analyzed and the selection of implementations has been made. After generation is complete, the data paths will be established and the control section will be generated. Machine generation begins by considering the control paths.

A. Control Paths

When the data flow nodes in the program were being created by the data flow analysis procedure, they were tagged with the control flow node that was "active" at that time. For example, in an assignment statement, all the data flow nodes on the right-hand side (as well as the data flow node on the left-hand side) would have the name of the assignment node in the control flow graph attached to them.

Data flow nodes also have a list of instances that "involve" the data flow node. It is therefore possible to tag each instance with the control flow node via the data flow nodes. As a result of this tagging, it is now possible to mark the control flow nodes with the instances that are active at that node.

Each control node also has a label that directs control flow. These labels are generated by the control flow analysis procedure.

Note that each match node is a specific "subsection" of a library module—in particular, these matches have bound control signals. These signals are the fields that must emanate from the control store. So control field generation is simply emitting the control bindings of every match of every instance of the implementation of a library module. The last matter in control field generation is the assignment of the microprogram counter field. Each control flow node has two pointers to other control nodes. These pointers are the "success" and "failure" pointers. Only control flow nodes generated by conditional expressions use the failure field. This field becomes the program counter field. As a default, the microcontroller assumes that the control word after the current control word will be located at the current program counter +1.

It is important to note that the current scheme of control flow generation does *not* solve the problems of precedence. For example, assignment statements involving the same variable on the left- and right-hand sides of the = generate simultaneous load and store microcommands for the implementation of the variable. This problem is easily solved; all that is required is a procedure that detects such conflicts and moves the appropriate conflicting operation down in the control store. (In this case, it would be the store).

Notice that the control store is not compacted in any way. A useful addition at this stage would be a microcode optimizer that would move microcode fields upward in the control store.

1) Control store generation: Generation of the control store is simple. For each node in the control flow graph, extract the control fields from the library definitions of each instance used at the node. If the node is `LOOP`, `EXIT` or `TEST`, then change the success and failure fields of the microcode word. Note that this omits the generation of the jump fields and compression of empty nodes (nodes without any control fields). The following algorithm describes the control store generation algorithm in the pseudoset language.

```
procedure ControlStoreGeneration is
    foreach node in the control-flow-graph do
        word := NewControlWord( );
        switch label of node into
```

```
case LOOP:
            word.success = node.success;
case EXIT:
            word.success = node.success;
case TEST:
            word.success = node.success;
            word.failure = node.failure;
case default: ;MUST be a ordinary node
            word.success = node.success;
            ;Now, for all the matches of all the instances for a given control node
            ;store the control signal bindings from the implementation.
            foreach instance in instances of search-node do
                foreach match in matches of the
                    ;This selects only the match nodes that effect that the
                    ;particular instance
                    instance INTERSECT matches of the node do
                        ;store field value (field name = control field of match part)
                        ;by the instance. The "implementation" is the-description
                        ;of the library module.
                        word.instance . control of match.part :=
                                        implementation of match.implements;
end ControlStoreGeneration
```

B. Data Path Generation

The previous section has shown how to construct the control paths and the control store for the machine that implements the input program. The last task is the generation of the data paths between the modules.

Data paths are informally established during selection via port constraints. As each selection is made, a data path is established between the ports of the new selection and the ports of the selections connected to the new instance via the data arcs in the data flow graph.

Net list generation is performed in two stages. First, the connections are made for every port in every instance (except for control ports). This establishes the data path section of the machine. Second, the output control ports of every instance are connected to the jump field multiplexer. Next, the control fields from the control store are finally connected to the instances they control. As a last step, the control field of the jump multiplexer is connected to the jump control field.

VI. Implementation

The implemented system (called SILI, short for SILIcon) is organized along the lines shown in Fig. 5. Compare this with Fig. 1. The program analysis is divided into control and data flow analysis; the control flow graph is not used by the matcher at all. The machine generation phase consists of control store generation and net list generation. Note that solid lines denote data flow; dotted boxes denote unimplemented sections. The labels on the arcs are the names of data formats.

Before SILI can process the input program, the various language dependent files must be read in. SILI is designed to be language independent—the syntax and semantics of the language are defined by language dependent files that represent the parsing rules (productions), the data and control flow "equations," the property propagation table, and the implementation library.

Briefly, SILI runs as follows: the input program is scanned and parsed by a recursive descent parser. The output of the parser is an ordinary parse tree. The parse tree is used as an intermediate form for several stages of analysis. The first action after parsing is control and data flow analysis. After this is completed, both the data flow and control flow graphs have been constructed. This is a good time for optimization of the data flow graph but this was not implemented in SILI. Next, the parse tree is traversed and declarations of types and other properties are attached to the terminal data flow nodes. Note that the rather baroque type declarations of both YASL and CLASP are meant as a substitute for more involved property extraction. Next, property extraction is done and properties are propagated to the non-terminal (interior) nodes of the data flow graph.

The graph matcher takes the decorated data flow nodes and the library of data flow nodes and generates matches called "instances." These instances are given to the search phase which performs the breadth-first search.

Note that critics may be called at any stage of the search, hence there is a dashed line to the "critics gallery," which was intended to be a collection of LISP code. Finally, the remaining implementations are given to the control store generator, which creates the control store and assigns the control fields. The final output is the net list generated from the implementations along with a list of "parts" from the library (with bound parameters).

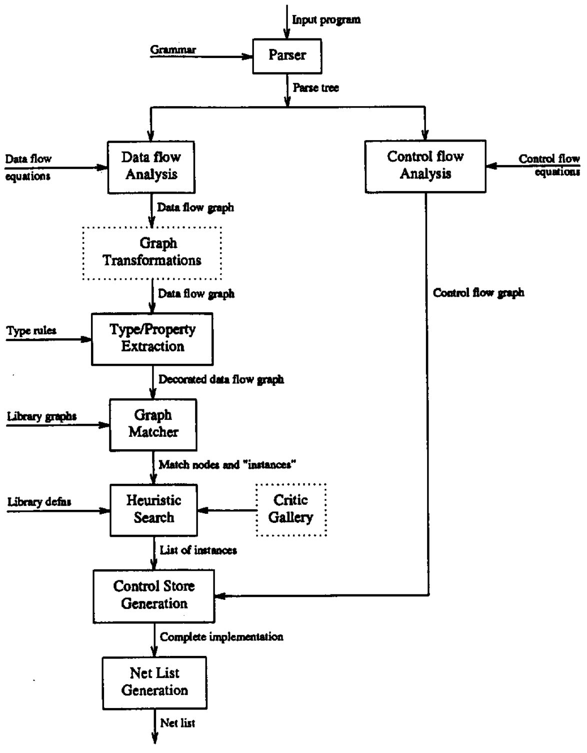

Fig. 5. Detailed block diagram of system organization.

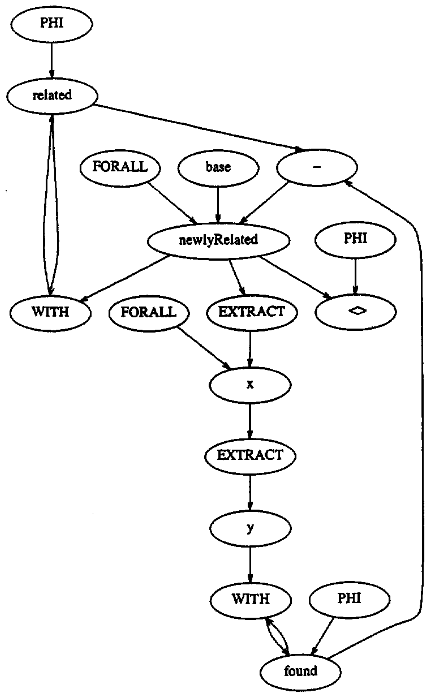

Fig. 6. Data flow graph of the transitive closure program.

The implementation was written in Franz-Lisp, a MacLisp dialect (in turn a descendant of Lisp 1.5) that runs on the VAX-11 series computers. The program occupies 475 pages of memory before compilation begins. The implemenation is meant as a "proof of concept" prototype; obviously, rewriting the code in an algorithmic language such as C would reduce the run time considerably. Also, precomputation of the constraint propagation between module ports would eliminate on-line computation of constraint violation during search.

A. An Example of System Operation

As a demonstration of the capabilities of the system, consider the following program written in YASL (this program is simpler than the CLASP program and therefore easier to explain):

```
program transitiveClosure

set of set of integer : base, related, newlyRelated, found;
set of integer : x,y;
set with size 0 of integer : phi;

related := phi;
newlyRelated := base;
while (newlyRelated <> phi) do
begin
        found := phi;
        forall x in newlyRelated do
                forall y in x do
                        found := found with y;
        related := related with newlyRelated;
        newlyRelated := found - related
end

end.
```

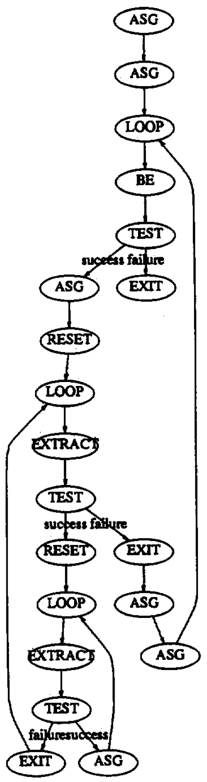

Fig. 7. Control flow graph of the transitive closure program.

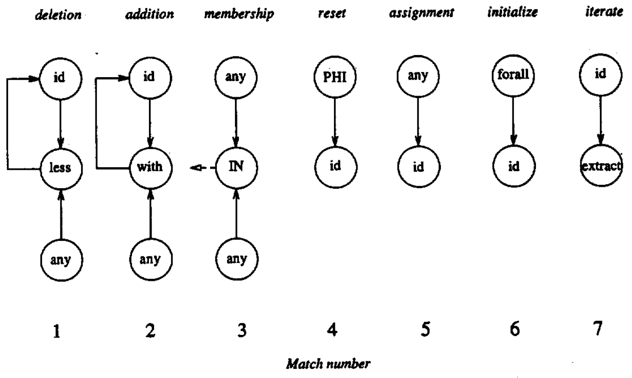

Fig. 8. Dataflow subgraphs of the binary tree module.

This program is a slightly reworked example from Low's dissertation [33, p. 14], written in the language YASL. The input library for YASL contains descriptions of sets inplemented as both parallel and serial registers. The data flow graph for the input program is shown in Fig. 6. The control flow graph of the example program is shown in Fig. 7.

As an example of matching, consider the binary tree set implementation (in the library) which has the data flow subgraphs shown in Fig. 8.

After calling the matcher with the subgraphs of the binary tree implementation, the data flow graph of the sample program will be matched as shown in Fig. 9. Note that the numbers next to the nodes in the program data flow graph denote the match number of the data flow subgraphs of the binary tree implementation shown in the previous figure.

Now, consider the following subgraph of the data flow graph in Fig. 9 (the port names are shown in *italics*). As each node is picked during search, the properties attached to each port are propagated to their connecting port. Consider the subgraph shown in Figure 10.

There are two cases to be considered. In the first case, all the terminals (`related`, `newlyRelated` and `found`)

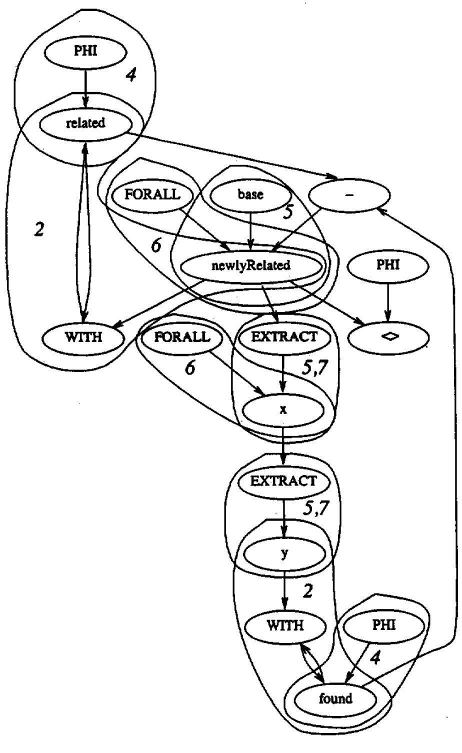

Fig. 9. Match of the data flow graph and binary tree module.

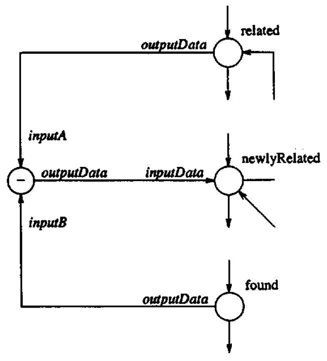

Fig. 10. Data flow subgraph.

are selected. In the second case, the nonterminals (in the subgraph this is only −) have been selected first. The two tables below illustrate the actions of the propagation algorithm. The notation "check" means that the properties of the ports will be checked whereas the notation "*no action*" means that no action will be taken since the node on the "other end of the wire" is not instantiated. The first case when the terminals are selected first is shown in Table IV. The second case when nonterminals are selected first is shown in Table V. Finally, the control section for the sample program (for the all parallel implementation) is found in Fig. 11.

The generated control store fields for one of the all parallel

TABLE IV
Constraint Propagation Example: Terminals Selected First

from node	to node	action taken
related:outputData	-:inputA	no action
newlyRelated:inputData	-:inputB	no action
found:outputData	-inputB	no action
-:inputA	related:outputData	check
-:inputB	newlyRelated:inputData	check
-:outputData	found:outputData	check

TABLE V
Constraint Propagation Example: Nonterminals Selected First

from node	to node	action taken
-:inputA	related:outputData	no action
-:inputB	found:outputData	no action
-:outputData	newlyRelated:inputData	no action
related:outputData	-:inputA	check
newlyRelated:inputData	-:outputData	check
found:outputData	-:inputB	check

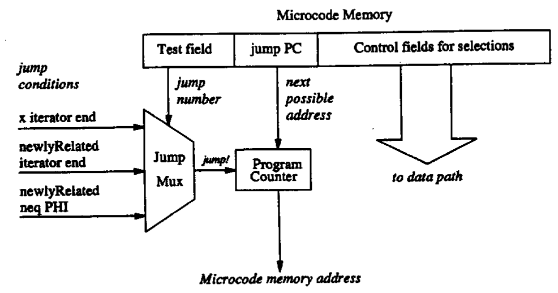

Fig. 11. Generated microcontroller.

solutions are shown in Table VI. The final step is the conversion of the machine graph to a "net list" (a connection list) suitable for a placement and routing system. The example YASL program compiles into the data path shown in Fig. 12.

The example program generated two solutions (the fully serial and fully parallel solution) using a library with parallel and serial implementations of the same modules (a total of 26 modules). The search makes *no* wrong selections at any point due to the use of the local constraint propagation algorithm.

VII. Results

The ultimate goal of this work, as elucidated in the introduction, was to enable an unsophisticated user to generate a VLSI circuit that executed the user's program and also met the user's established design requirements.

SILI meets these goals through its exploitation of various constraint based methods and heuristic search. By using an external library, the low level constraints can be met via other lower level design tools. However, there are still problems to be solved.

TABLE VI
MICROCODE FIELDS FOR SAMPLE PROGRAM AND LIBRARY

PC	base	newlyRelated	related	found	*x*	*y*	test	next PC
0			reset					
1	load	store						
2							newlyRelated =PHI	12
3				reset				
4		reset						
5		test						
6		iterate			store		iterator end	10
7					reset			
8					test			
9					iterate	store	iterator end	12
10		load	with					
11		store	load	load				
12								

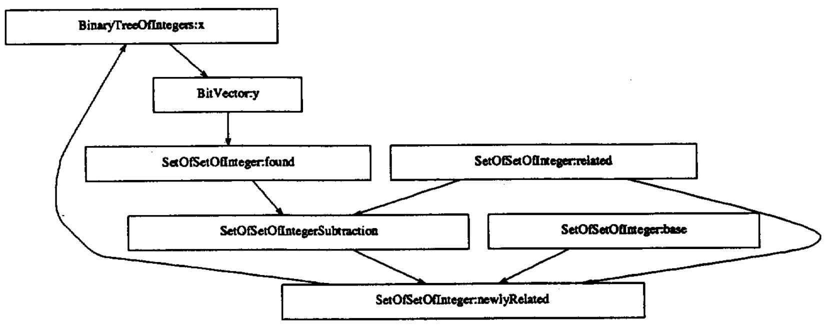
BinaryTreeOfIntegers:x
BitVector:y
SetOfSetOfInteger:found
SetOfSetOfInteger:related
SetOfSetOfIntegerSubtraction
SetOfSetOfInteger:base
SetOfSetOfInteger:newlyRelated

Fig. 12. Data path.

A. Timing Measurements

Unfortunately, SILI lacks a good timing measurement subsystem. This was due strictly to the amount of effort spent in describing and analyzing the timing of the generated circuits. As it stands now, SILI adds up the "delay" times that are specified as part of each implementation specification. This should be replaced with a timing analysis subsystem that uses such measurements as the delay from statement to statement or the delay of a loop. Systems like those described by Cohen and Zuckerman [34] or Ramshaw [35] could be extended to cover such timing calculations. The CADDY system used a very simple technique for estimating time using control flow graphs. Such a technique could easily be used here. A complex timing analysis subsystem should be part of any silicon compiler.

B. Optimization

The optimization step was not fully completed because of the implementation complexity; changes to the data flow graph at any point in the search cannot be passed on to other designs in progress. A reimplementation taking care to avoid this problem would ease the implementation of critics. Without optimization operators, the circuits created by SILI are unrealizably large. However, even with such optimization, there is still a problem with the level of module specification and the direct reduction of data flow graphs to machines; this is discussed next.

C. One Step Selection

One step selection (direct selection of high level modules from a library) hides much of the complexity from the user. Unfortunately, it also hides hierarchy from the system as well. In order to generate truly optimized designs, each library module should be decomposable. Without such a multilevel view, it is impossible to perform optimizations within a library module. Such optimizations are mandatory if the generated designs are to approach the area of human engineered designs. This is one of the reasons that SILI's circuits are unreasonably large.

D. Conclusion

This work is one step toward the ultimate goal of a system that compiles a program to a description of an integrated circuit. A start has been made by using techniques from classical Artificial Intelligence and conventional compiler theory and practice. This potent combination of techniques has resulted in the reduction of search space and an ease of compilation. The work reported in this paper has shown that:

- Very high level languages can be used to hide the implementation complexity of VLSI design.
- Local constraints can cut heuristic search time considerably.
- Heuristic search and constraints can be successfully used to choose implementations with differing costs.

- The use of graph representations for modules eases circuit construction and description of module semantics.

Acknowledgment

Diane Litman and Howard Trickey have generously read this paper and offered detailed comments. The referees have helped the author refine and clarify this paper considerably. The former Editor, Michael Lightner, also helped facilitate the final version of this paper.

References

[1] D. Johannsen, "Bristle blocks: A silicon compiler," in *Proc. 16th Design Automation Conf.*, San Diego, CA, 1971, pp. 310-313.

[2] D. D. Gajski, N. D. Dutt, and B. M. Pangrle, "Silicon compilation (tutorial), in *Proc. 1986 Custom Integrated Circuits Conf.*, 1986, pp. 102-109.

[3] L. Monier and J. Vuillemin, "Using Silicon Assemblers," in *Proc. VLSI-85*, Tokyo, Japan, Aug. 1985, pp. 309-318.

[4] H. W. Tickey, "Flamel: A high level hardware compiler," *IEEE Trans. Computer-Aided Design*, vol. 6, pp. 259-269, Mar. 1987.

[5] B. M. Pangrle and D. J. Gajski, "Design tools for intelligent silicon compilation," *IEEE Trans. Computer-Aided Design*, vol. 6, pp. 1098-1112, Nov. 1987.

[6] A. Parker, D. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Lieve, and J. Kim, "The CMU design automation system," in *Proc. 16th Design Automation Conf.*, San Diego, CA, 1971, pp. 73-80.

[7] M. R. Barbacci and D. P. Siewiorek, *The Design and Analysis of Instruction Set Processors.* New York: McGraw-Hill, 1982.

[8] J. Earley, "Relational data structures for programming languages," *Acta Inf.*, vol. 2, pp. 293-309, 1973.

[9] J. H. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg, *Programming with SETL: An Introduction to SETL.* New York: Springer-Verlag, 1986.

[10] M. Kahrs, "Silicon compilation of a very high level signal processing specification language," in *VLSI Signal Processing*, P. R. Capello, Editor. New York: IEEE Press, 1984, pp. 228-238.

[11] M. Hecht, *Flow Analysis of Computer Programs.* New York: Elsevier, 1977.

[12] E. A. Snow, "Automation of module set independent register transfer level design," Ph.D. dissertation, Dept. of Electrical Engineering, Carnegie Mellon Univ., Pittsburgh, PA, April 1978.

[13] R. Camposano and W. Rosenstiel, "Synthesizing circuits from behavioral descriptions," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 171-180, Feb. 1989.

[14] D. Knapp and A. Parker, "A unified representation for design information," in *Proc. CHDL-85*, Tokyo, Japan, Aug. 1985, pp. 337-353.

[15] D. E. Thomas, C. Y. Hitchcock, T. J. Kowalksu, J. V. Rajan, and R. A. Walker, "Automatic data path synthesis," *IEEE Computer*, vol. 6, pp. 59-70, Dec. 1983.

[16] C. Tseng and D. P. Siewiorek, "Automatic synthesis of data paths in digital systems," *IEEE Trans. Computer-Aided Design*, vol. 5, pp. 379-395, July 1986.

[17] A. J. Parker, J. Pizarro, and M. Milnar, "MAHA: A program for datapath synthesis," in *Proc. 23rd Design Automation Conf.* Las Vegas, NV, June 1986, pp. 461-466.

[18] M. W. Kahrs, "Silicon compilation of very high level language," Ph.D. dissertation, Univ. Rochester, NY, Oct. 1983.

[19] A. Mackworth, "Consistency in Networks in relations," *Artif. Intell.*, vol. 6, pp. 99-118, 1977.

[20] D. Waltz, "Using constraints in computer scene understanding," in *Psycology of Computer Vision*, H. Winston, Editor. New York: McGraw-Hill, 1975, pp. 19-92.

[21] J. A. Darringer and W. H. Joyner, Jr., "A new look at logical synthesis," IBM, Yorktown Heights, NY, RC 8268, May 1980.

[22] E. A. Snow, D. P. Siewiorek, and D. E. Thomas, "A technology-relative computer-aided design system: abstract representations, transformations, and design tradeoffs," in *Proc. 15th Design Automation Conf.*, Las Vegas, NV, 1978, pp. 220-226.

[23] M. Kahrs, "Critics at optimization elevators in a silicon compiler," in *Knowledge Engineering in Computer-Aided Design*, J. Gero, Editor. New York: Elsevier-North-Holland, 1985, pp. 313-325.

[24] G. J. Sussman, *A Computer Model of Skill Acquisition.* New York: Elsevier, 1975.

[25] N. J. Nilsson, *Principles of Artificial Intelligence.* Palo Alto, CA: Tioga Press, 1980.

[26] B. T. Lowerre, "The HARPY speech recognition system," Ph.D. dissertation, Carnegie Mellon Univ., Pittsburgh, PA, 1976, also Tech. Rep., Carnegie Mellon Univ.

[27] J. Doran, "An approach to automatic problem solving," in *Machine Intelligence*, D. Mitchie and N. L. Collins, Editors. Edinburgh, U.K.: Edinburgh University Press, 1967, pp. 105-123.

[28] C. D. Thompson, "A complexity theory for VLSI," Ph.D. dissertation, Carnegie Mellon Univ., Pittsburgh, PA, Aug. 1980.

[29] G. Baudet, "On the area required for VLSI circuits," in *CMU Conference on VLSI Systems and Computations*, H. T. Kung, B. Sproull, and G. Steele, Editors. Rockville, MD: Computer Science Press, 1981, pp. 100-107.

[30] B. Chazelle and L. Monier, "A model of computation for VLSI with related complexity results," Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep. 81-107, Feb. 1981.

[31] P. Capello and K. Steiglitz, "Digital signal processing applications of systolic algorithms," in *CMU Conference on VLSI Systems and Computations*, H. T. Kung, B. Sproull, and G. Steele, Editors. Rockville, MD: Computer Science Press, 1981.

[32] F. J. Kurdahi and A. C. Parker, "Techniques for area estimation of VLSI layouts," *IEEE Trans. on Computer-Aided Design*, vol. 8, pp. 81-82, Jan. 1989.

[33] J. R. Low, "Automatic coding: Choice of data structures, CS-74-452/AIM-242," Ph.D. dissertation, Stanford Univ., Stanford, CA, Aug. 1984. (Also Cambridge, MA: Birkhauser.)

[34] J. Cohen and C. Zuckerman, "Two languages for estimating program efficiency," *Comm. ACM*, vol. 17, no. 6, pp. 301-307.

[35] L. H. Ramshaw, "Formalizing the analysis of algorithms," Xerox, PSRC, CSL-79-5, June 1979. (Also Stanford Univ., Stanford, CA, STAN-CS-79-741.)

Mark Kahrs (S'78-M'82) received the Ph.D. degree in computer science from the University of Rochester, NY.

He is an Assistant Professor of electrical and computer engineering at Rutgers University, Piscataway, NJ, where he teaches courses in digital signal processing, audio engineering, and computer systems architecture. Formerly, he was with Xerox PARC, the Center for Computer Research in Music and Acoustics (CCRMA), Stanford University, Stanford, CA, the Institute de Rechereche et Coordination Acoustique Musique (IRCAM), Paris, and AT&T Bell Laboratories.