Splicer: A Heuristic Approach to Connectivity Binding

Barry M. Pangrle

Department of Electrical and Computer Engineering
University of California
Santa Barbara, California 93106

Abstract:

This paper describes a tool for constructing the connectivity between hardware components given a state-graph that the components are to be mapped into. Examples taken from previous papers in the field are used to demonstrate this connectivity binder. The important results of this paper point to heuristics that are used to generate solutions to the problem. Questions addressed include: how much of the state-graph must be considered at one time to give reasonable results and how the search space can be pruned to achieve good solutions quicker. The code for this project is written in C and runs under 4.2 BSD UNIX.

1. Introduction

As design tools improve at the floor-planning and layout levels, tool designers are pushing towards building tools that operate at higher and more abstract levels of design [Sang87]. Such high level synthesis systems would help chip designers by providing them with a tool for flushing out designs and getting performance estimates without having to go through long manual design cycles. The ability to quickly modify and update designs would allow for a greater exploration of the design space before the chip was actually produced resulting in better chips getting to market quicker.

Commercially available silicon compilers today allow designers to specify designs in terms of micro-architectural components such as RAMs, ROMs, ALUs, PLAs, etc. These components are specified by a language description or by filling out a menu or a form. Each of these micro-architectural components is then instantiated by a parameterized module generator[BuMa85]. These compilers generate the layout of the modules along with timing and logic models for the modules to aid in verifying the design. Above these module generators there are typically tools to aid in the floor-planning and the placing and routing of the modules to produce the final design.

The goal of high level synthesis tools then is to allow the designer to input a description at a higher abstraction level and let the system do most of the work. The input ideally would be a functional specification of the chip and the synthesizer would perform the necessary design exploration to produce the final chip. In moving towards this goal current research has been focused on creating tools that can synthesize designs from functional (or behavioral) specifications. A representative list of such systems includes the CMUDA project [PTSB79] [THKR83] [KoTh85] [TsSi86], Elf [GiKn84], HAL [PaKG86], MacPitts [Sout83], and MAHA [PaPM86].

Typically, the input to these systems is a functional specification written in a high level language. This input is fed into a compiler to generate a flow graph that represents the data and control dependencies of the operations in the functional specification. The next step consists of developing a set of micro-architectural components and mapping them into the flow graph to produce a data path or finite state machine that will perform a series of instructions or states that has the same behavior as that originally specified by the designer.

In the system that Splicer is one piece of, the design synthesis is performed in three separate phases:

1) component selection,
2) state synthesis, and
3) connectivity binding.

Component selection consists of developing the set of available components from which the design is synthesized. Examples are constructing ALUs and determining how many components of any one type are allowed. This phase is performed by an expert system that performs global design trade-offs [BrGa87]. These components are then passed to separate routines to perform the assignment of the operations to states (state synthesis) [PaGa87] and the connectivity binding of the components. Results from the state synthesizer and the connectivity binder are reported back to the expert, allowing it to change component selections to meet time, area, and power constraints. This paper focuses on the third phase of this process, the connectivity binding which is performed by Splicer.

The next section of th is paper describes the connectivity binding problem and how it relates to the component selection and the state synthesis phases. The third section describes some previous work. Section four describes Splicer and its approach to connectivity binding. Section five looks at three example problems used for experiments. The problems are roughly of a small, medium, and large size in an attempt to show comparisons of Splicer under varying conditions. Section six offers some conclusions for the results of the example problems.

2. Problem Description

Connectivity binding consists of mapping hardware components into a **control / data flow graph** (CDFG) with state information. The state information is added to the CDFG during state synthesis and a CDFG with the added state information is called a **state-graph**. The hardware components used by the state synthesizer (Slicer) and the connectivity binder (Splicer) are the same ones selected during the component selection phase.

The connectivity binding problem as attacked by Splicer is more formally stated as: Given a set of possible operations $O = \{op_1,op_2,...,op_o\}$, a set of function units $F = \{fu_1,fu_2,...,fu_u\}$ with each fu_i capable of performing $P(fu_i)$ a subset of the operations in O, and a set of registers $R = \{r_1,r_2,...,r_k\}$, map the function units and registers into a directed graph G = (V,A). The graph vertices are represented by the set $V = \{v_1,v_2,...,v_x\}$, and the graph arcs by the set $A = \{a_1,a_2,...,a_y\}$, with $a_h = \{v_i,v_j\}$ representing the arc from v_i to v_j where $v_i,v_j \in V$ and $a_h \in A$. The set F corresponds to the function units chosen in the component selection phase. The vertices in the graph represent operations to be performed and the arcs represent data dependencies between the operations. The function $op(v_i) \in O$ represents the operation to be performed at vertex v_i. A set of states $S = \{s_1,s_2,...,s_n\}$ represents the temporal ordering of the operations to be performed in the graph so that the data dependencies represented by the arcs in G are preserved. Each s_i contains a subset of V such that $s_1 \cup s_2 \cup ... \cup s_n = V$. Graph G along with the information in S forms a state-graph. It is assumed that there exists some assignment of the function units in F to the vertices V in G for the given state assignment S such that letting $fu(v_i) \in F$ represent the function unit assigned to vertex v_i, then for all $v_h,v_i \in s_j$, $fu(v_h) \neq fu(v_i)$ where $h \neq i$. This implies that each function unit can be used only once during each state. If this condition cannot be satisfied no valid component mapping exists. Slicer guarantees that the state-graph meets these conditions, otherwise the expert is notified and another component set is selected. Since the connectivity binding is performed after state synthesis, a valid function unit to vertex mapping is guaranteed. So, given these hardware components (function units and registers) and the directed graph G, the problem is to find a mapping of the components to the graph that minimizes the number of connections between the function units and the registers.

In general, the problem can be thought of as minimizing the number of connections and registers. Splicer operates on the premise that every arc that crosses a state boundary requires a register, i.e. live data must be stored during state transi qtions. Because of this, the minimum number of registers that Splicer can use for any state-graph is equal to the maximum number of arcs that cross any one state boundary. Splicer is currently set to perform connectivity binding using only this minimum number of registers. So in this sense the number of registers is really predetermined in the state synthesis phase.

2.1 Function Unit Models

The function units execute operations represented by the vertices of the graph. Along with the list of operations that each unit performs P(fu), the time that it takes for a given unit to perform an operation is also specified as $t(op,fu_i)$ where $op \in P(fu_i)$ and $fu_i \in F$. This allows the modelling of two units that perform the same operation but have different execution speeds. For example an adder could be modeled as a carry-lookahead adder (fast) or as a ripple-carry adder (slow). Obviously, this extends to other operations that allow designers trade offs between area and time. Typically, this issue is more important at the state synthesis level where the design performance is determined. The importance of the operation times at the connectivity level is that when the state synthesizer assigns graph vertices to states, it also places an implied execution time limit on each vertex. When the connectivity binder assigns a function unit fu_j to a vertex v_i, the following condition must strictly hold: $t(op(v_i),fu_j) = t(v_i)$, where $t(v_i)$ = [the time requirement for vertex $v_i \in V$ assigned by the state synthesizer].

2.2 Multi-cycle and Chaining

Because function unit execution times are variable and the clock cycle (or state length) is variable, it is possible for a vertex to appear in more than one state or for a path of vertices to appear in a single state. These situations are sometimes referred to as multi-cycled operations and chained operations respectively. This implies that the model must be able to provide connections between sets of function units within one state and that the output from a unit can occur in a state after the state that the unit received its input(s).

3. Previous Work

The problem of state synthesis and connectivity binding is intertwined in a way similar to the placement and routing problem at the layout level. Because of the complexity of the problem, it is natural to perform the state synthesis first and the connectivity binding second. The CMU systems DAA, EMUCS [THKR83], and Emerald [TsSi86] are based on performing state synthesis and connectivity binding in separate phases. Emerald and HAL [PaKG86] both use clique-partitioning techniques to perform connectivity binding. Emerald can form bus based designs but HAL uses a simpler one-level mux interconnectivity model. The ELF [GiKn84] system is an exception, in that while performing the operation to state allocations a calculation of the connectivity cost of allocating an operation to a state is performed to decide if the operation should actually be allocated to that state. This proceeds on a state by state and an operation by operation basis.

3.1. Impact of Slicing on Connectivity Binding

Because the state synthesis and connectivity binding problem are inter-related, modifications in the state assignment can directly affect the connectivity. Paulin [PaKG86] showed how two state assignments with equal execution times can have different hardware cost. His example showed how load-balancing could move operations assigned using ASAP (as soon as possible) scheduling into later states without increasing the total number of states. Balanced states tend to use less connections by reusing connection paths used in previous states. If one state is considerably wider (i.e. has more operations being performed in parallel) than all the others and can be load-balanced without increasing the number of states, the connectivity costs to provide the parallelism in this wide state in general will not be recovered in the subsequent narrower states. This is because the added connections to provide the extra parallelism in this wide state will tend to be under utilized in the rest of the design. This is especially true for bus modeled inter-connections.

3.2. Connectivity Models

Two types of models are commonly used in connectivity binders. The first is the simple point to point connectivity model and the second is the bus style model. The point to point model assigns connections from outputs to inputs. The result is that every register and function unit output has one line associated with it that may go to many inputs. This leads to designs that have multiplexers in front of the inputs of registers and function units. EMUCS [Hitc83] [THKR83] and MAHA [PaPM86] are examples of such connectivity binders. Bus style connectivity on the other hand, connects component outputs to buses and the buses to component inputs. This model can also be viewed as two levels of multiplexers where the outputs placed onto a bus are actually multiplexed and the buses are the outputs of these multiplexers. In real bus systems these "multiplexers" are distributed as tri-state drivers with the control for the drivers possibly encoded in the control word with multiplexers or buffers / latches reaching from the bus. An important advantage of bus style models is the sharing of long data lines across the chip.

3.3. Advantages of Splicer

Splicer uses a bus based interconnectivity model. The point to point (or one-level mux) model is actually a subset of the bus (or two-level mux) model. The bus based model allows for greater sharing of connections at the cost of added computational complexity to the problem. Because Splicer is cost driven, by simply modifying the under lying cost function used by Splicer, the user can select either point to point or bus-based. Splicer also properly handles multi-cycle unit connections and chained unit connections which allows the user to select the clock cycle or state length to best fit the design problem. Splicer also lets the user control the connectivity search by sending only portions of the state-graph at a time for binding. This is referred to as lookahead and is explained later in more detail.

4. Splicer

The Splicer connectivity binder is based upon branch and bound searching. Because of this underlying foundation, Splicer can find optimal solutions for some smaller sized problems. It should be noted that these solutions are optimal for the state-graph used as input and the connectivity model used. By altering the cost functions used by Splicer, the connectivity model can be changed. To experiment with Splicer, the cost functions can be modified to produce heuristics for obtaining connectivity bindings. As Splicer proceeds with any design, it keeps track of statistics for the design. The number of buses, registers, multiplexers, etc. are kept track of at each step. Cost functions can be created by using a cost formula with any or all of these statistics.

4.1. Algorithm

The Splicer algorithm is recursive in nature. Components are selected and tried and then marked internally to assure that backtracking is performed correctly over the search space. The following section of pseudo-code gives the flavor of the Splicer connectivity binder's approach.

```
procedure BIND()

for each state do {
    if (input register to be connected)
        CONNECT_REGISTER_TO_BUS()
    else if (bus to be connected to function unit)
        CONNECT_BUS_TO_UNIT()
    else if (unit to be connected to a bus)
        CONNECT_UNIT_TO_BUS()
    else if (bus to be connected to an output register)

        CONNECT_BUS_TO_REGISTER
    else if (last state)
        STORE_SOLUTION()
    else
        state ← next_state }
```

Based on the calculated cost at each step, either BIND() is called recursively or the called routine returns. BIND() clearly shows the four main steps that Splicer uses in assigning hardware components to the graph. It should also be noted that the **for each** statement refers to each state of the section of the graph passed to Splicer, i.e. a subset of sequential states of the graph can be passed to Splicer instead of the whole graph. This allows the user to make a trade-off between the design quality and the run time. If connectivity binding is performed across one state at a time and each state takes approximately time C to bind, then an n state graph can be bound in Cn time.

The next section of pseudo-code outlines the routine CONNECT_REGISTER_TO_BUS() which is representative of the approach used in the other routines.

```
procedure CONNECT_REGISTER_TO_BUS()

register ← current input register up for assignment

If (register is connected to a bus and bus is not already
used or tried) {
    connect register to bus
    update current input register
    BIND()
    un_update current input register
    mark register bus connection as tried}
else (find next available bus that register is not
connected to) {
    calculate cost of connecting register to bus
    if (cost acceptable) {
        add cost of connection to current cost
        add connection to component connectivity table

        update current input register
        BIND()
        un_update current input register
        mark register bus connection as tried
        delete connection from component connectivity
table
        remove cost of connection from current cost } }

return
```

4.2. Cost Functions

Splicer works on an underlying bus based model as is shown in the previous pseudo-code segments. Statistics are kept at four separate levels of connectivity. The first two levels, 1 and 2, refer to connections **to** and **from** input buses respectively and levels 3 and 4 are for connections **to** and **from** output buses. Multiplexer connections are also modeled at four levels: 1) registers to input buses, 2) input buses and output buses to function unit inputs, 3) function unit outputs to output buses, and 4) output buses and input buses to registers. This model can be collapsed to a point to point model by allowing the outputs of registers and function units to be connected to one and only one bus. This constraint placed in the cost function corresponds to multiplexers being generated at the inputs of registers and function units only.

4.3. Lookahead

Lookahead corresponds to the size of the state graph section passed to Splicer. This allows the user to vary how much of the graph Splicer attempts to allocate at one time. A lookahead of 0 means that Splicer looks ahead 0 states into the graph to determine the connectivity assignment for the current state. A lookahead of 1 means that Splicer allocates two states simultaneously, the current state and the 1 lookahead state. Presently, Splicer discards the bindings for all lookahead states and saves only the bindings for the current state. Therefore, Splicer is called N times to bind an N state graph saving the bindings of each state sequentially one at a time.

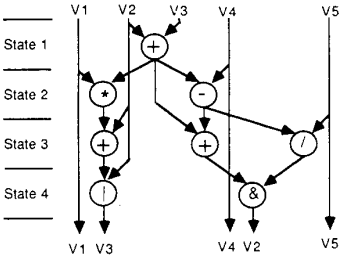
5. Experiments

This section describes some experiments performed with Splicer to determine the effects of look-ahead, cost functions, and operator commutativity on the quality of the solutions obtained. Because Splicer uses a greedy approach initially for binding and a depth-first search, a fair solution is arrived at rather early in the search and is improved upon as Splicer backtracks. This allows the user to place a bound on the number of iterations Splicer spends on any part of the search. Three important questions that these experiments help answer are: 1) how many states should be sent to Splicer at one time relative to the rest of the graph (i.e. how much look-ahead should be used), 2) what modifications can be made to the cost function to transform the search from branch and bound to a heuristic that effectively prunes the search without penalizing the result quality too heavily, and 3) how important is it to be able to exchange inputs on commutative operators.

Three example state graphs from the literature are used to illustrate some of the results. They were selected because of their previous use and because they are fairly well known. On each graph three different types of runs were made 1) a simple branch and bound run using the number of multiplexer inputs as the ctriterion, 2) a heuristic cost function that uses multiplexer inputs as its criterion but further prunes the search by performing incremental checking on the partial solutions, and 3) branch and bound again, but with the ability to swap operands on commutative operations removed. For each of the three scenarios, runs were made using several different amounts of lookahead. All the CPU times are for a SUN 3/260 with 16Meg of RAM memory.

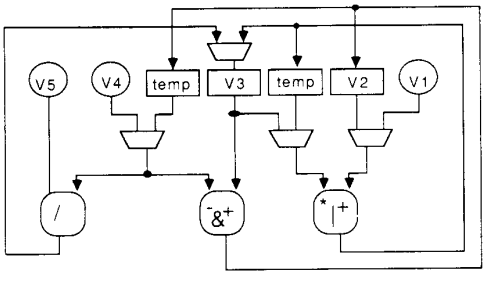
5.1. Example 1.

The state graph shown below is used for the first example. This is the same state graph that was used in [TsSi86].



The table below shows how the solution quality varies for different amounts of lookahead with the same cost function (branch and bound [B&B] or the heuristic). The best solution found for this example uses four 2-input multiplexers This example was small enough that by sending the whole state graph to Splicer at one time (lookahead = 3) it was possible to let it run until exhaustion to show that this in fact is an optimal design in terms of the number of multiplexers used. This design is also shown below the table.

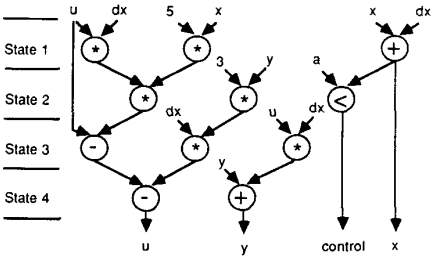
EXAMPLE #1			
Lookahead	# of Muxes	# of Mux Inputs	CPU (sec)
Branch & Bound Solutions			
0	5	10	2.0
1	4	8	17
2	4	8	290
3	4	8	230
Heuristic Solutions			
0	5	10	0.6
1	4	8	1.4
2	4	8	3.4
3	4	8	4.0
B & B w/o commutative operators			
0	5	11	1.0
1	4	9	11.5
2	4	8	55
3	4	8	55



Using the heuristic cost function, even with a lookahead of 3 it only takes 4 seconds to complete the search. With a lookahead of one for either the B&B or the heuristic the optimal solution is obtained. Eliminating the commutativity required a lookahead of 2 to obtain an optimal solution. Facet's solution for the same graph requires four 2-input muxes, one 3-input mux and 8 registers instead of the 7 registers used by Splicer.

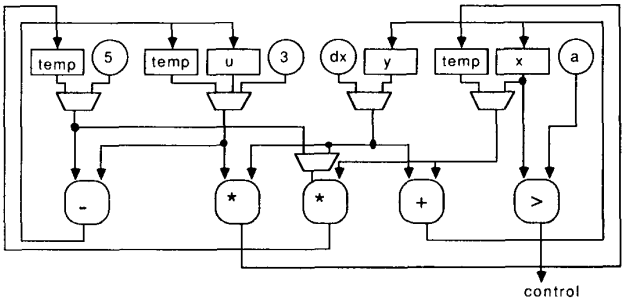
5.2. Example 2.

The second example graph is shown below and it has been used with HAL [PaKG86]. This graph contains more vertices than the first example and has fewer operation types. These two factors add to the complexity of finding good connectivity solutions.



The large number of operations performed in parallel increases the number of connections that are necessary to create the data path. The best solution found for the connectivity of this graph uses 5 multiplexers with a total of 11inputs (four 2-input multiplexers and one 3-input multiplexers). A fully exhaustive search over the graph was not performed, so it is not known if this is an optimal solution. As can be seen in the table for example 2, running even lookahead of 1 with B&B requires almost 1 hour of cpu time. The best solution was found using the heuristic cost function and lookahead of 3. It should be noted though that a lookahead of 1 produces an almost as good solution. Eliminating commutativity reduced the search time at the cost of solution quality in this case. Hal's solution to this state graph requires 6 multiplexers (four 2-input and two 3-input) thus Splicer's best solution saves one 3-input multiplexer. This design is shown below.

EXAMPLE #2			
Lookahead	# of Muxes	# of Mux Inputs	CPU (sec)
Branch & Bound Solutions			
0	6	16	16.4
1	5	12	3209
Heuristic Solutions			
0	6	16	6.4
1	5	12	291
2	5	12	524
3	5	11	1245
B & B w/o commutative operators			
0	7	17	13
1	5	14	338



5.3. Example 3.

The third and last example is a fifth order elliptic filter [KuWK85]. This example was used by Paulin [PaKn87]. The input language description is shown below. The state graph consists of 21 states (control steps) and the components consist of 2 adders that execute in one control step and 1 multiplier that executes in 2 control steps.

```
program ellip(input,output);  /* fifth order elliptic filter */

type    integer = {0..15};

reg     t2, t13, t18, t33, t39, t26, t38, m21, m24, m9,  m30,
        m40, m36, m16, m6 : integer;

port    In, Out: integer;

var     a, b, c, d, e, f, g, h, i, j, k, o : integer;

begin           /* block automatically solves loop boundaries */
        i := In;
        a := i + t2;
        b := a + t13;
        g := t33 + t39;
        e := g + t26 + b;
        d := (m21*e) + b;
        f := (m24*e) + g;
        t26 := f + d + e;
        c := m9 * (b + d) + a;
        h := m30 * (f + g) + t39;
        j := t18 + c + d;
        k := t38 + f + h;
        t39 := o + h;
        t38 := t38 + (m36*k);
        t33 := t38 + k;
        t18 := t18 + (m16*j);
        t13 := t18 + j;
        t2 := c = i + m6*(a + c);
        Out := o;
end.
```

The table for example 3 below shows how quickly the problem search space grows as the lookahead is increased. It is interesting that the best solution found was with the lookahead set to one. Maybe more surprising is that the heuristic found a better solution than the B&B with equal lookahead. Because the solutions found by B&B over a subset of the state-graph are only locally optimal, they do not guarantee that a better solution couldn't be found using a heuristic with an equal amount of lookahead. What's rather pleasing about this instance is that the best solution found was obtained in only 55 seconds of cpu time using the heuristic cost function.

EXAMPLE #3			
Lookahead	# of Muxes	# of Mux Inputs	CPU (sec)
Branch & Bound Solutions			
0	10	49	48
1	9	44	20257
Heuristic Solutions			
0	13	48	19
1	9	43	55
2	9	45	300
3	12	46	875
4	10	45	10238
B & B w/o commutative operators			
0	14	55	401
1	10	51	18190+

Another noticable point about this example is that it appears that the removal of the commutativity of operations doesn't help speed up the search. In fact the lookahead of 1 non-commutative run has a plus sign after it because of a time bound set on searching individual states was exceeded. If the bound hadn't stopped the search , the time would have been longer and the solution may have gotten better or might have gotten worse. The time bound is setable by the user and this was the only run that hit such a bound.

The difference between the heuristic cost functionand the branch and bound cost function is that the heuristic compares partial solution costs of the solution being constructed to partial solutions of the *best solution found so far* (BSFSF). This differs from true branch and bound which compares partial solution costs to the total cost of the BSFSF. By comparing partial solutions to partial solutions of the BSFSF, branches can be pruned before their costs exceed the solution costs, thus reducing the search space. This also removes the guarantee of an exhaustive run finding the optimal solution with the heuristic cost function.

The most natural place to store partial solutions of the BSFSF for comparison to new partial solutions is at the boundaries between states (this is useful only when using lookahead). Sometimes adding extra connections in previous states can result in savings later in the binding. At an extreme, if the partial solution costs at the boundaries of states are compared and the search only continues when the partial cost is less than the partial cost of the BSFSF at the end of the same state, then the final connectivity solution is equivalent to using a lookahead of 0. If the search is allowed to continue when the cost is less than or equal to the BSFSF partial solution cost, then it is possible that a better connectivity arrangement with the same cost may be found which reduces the global cost.

6. Conclusions

A look-ahead of one is often sufficient to generate good connectivity solutions. Pruning the search space at different connectivity levels is a worthwhile method. For small graphs, Splicer is sufficiently powerful enough to be able to find optimal solutions. In examples 1 and 2 splicer produced designs that used one 3-input mux less than the designs produced by the systems that originally ran the examples. In each case this was over a 20% decrease in the number of mux inputs. As a tool, Splicer allows the user to trade off run time for result quality making it a useful tool for rough estimations or for finding very good final solutions. In example 3 it appeared that removing commutativity, decreased the possible solution quality such that the loss of being able to prune against a better found solution resulted in longer run times. This implies that sometimes it may be better to increase the possible search space if better solutions are arrived at quicker. There is also a random element of how the input was specified as to how good or bad a solution might be based on eliminating commutativity. Future use of Splicer could involve the use of an expert to match input graphs to cost functions from a library of cost functions to efficiently produce designs based on different connectivity styles.

Acknowledgement

I would like to thank Professor Daniel Gajski for his helpful suggestions on how to improve this paper and Forrest Brewer and Nikil Dutt for supplying me with more examples to run.

7. References

[BrGa87] F. D. Brewer and D. D. Gajski, "Knowledge based control in micro-architecture design," *24th Design Automation Conference*, June 1987.

[BuMa85] M. R. Buric and T. G. Matheson, "Silicon compilation environments," *Custom Integrated Circuits Conference*, May 1985.

[GiKn84] E. F. Girczyc and J. P. Knight, "An ADA to standard cell hardware compiler based on graph grammars and scheduling," *International Conference on Computer Design*, October 1984

[Hitc83] C. Y. Hitchcock III, "Automated synthesis of data paths," Master's Thesis, Dept. of Electrical and Computer Engineering, Carnegie-Mellon University, January 1983.

[KoTh85] T. J. Kowalski and D. E. Thomas, "The VLSI design automation assistant: What's in a knowledge base," *22nd Design Automation Conference*, June 1985.

[KuWK85] S. Y. Kung, H. J. Whitehouse, T. Kailath, "VLSI and modern signal processing," Prentice Hall, 1985.

[PaGa87a] B. M. Pangrle and D. D. Gajski, "Slicer: a state synthesizer for intelligent compilation," *International Conference on Computer Design*, October 1987.

[PaGa87b] B. M. Pangrle and D. D. Gajski, "Design tools for intelligent silicon compilation," *IEEE Transactions on Computer-Aided Design*, vol. CAD-6 no. 6, November 1987.

[PaKG86] P. G. Paulin, J. P. Knight, and E. F. Girczyc, "HAL: A multi-paradigm approach to automatic data path synthesis," *23rd Design Automation Conference*, June 1986.

[PaKn87] P. G. Paulin and J. P. Knight, "Force-directed scheduling in automatic data path synthesis," *24th Design Automation Conference*, June 1987.

[PaPM86] A. C. Parker, J. Pizarro, and M. Mlinar, "MAHA: A program for data path synthesis," *23rd Design Automation Conference*, June 1986.

[PTSB79] A. C. Parker, D. E. Thomas, D. Sieworek, M. Barbacci, L. Hafer, G. Lieve, and J. Kim, "The CMU design automation system: An example of automated data path design," 16th Design Automation Conference, June 1979.

[Sang87] A. Sangiovanni-Vincentelli, "Synthesis of LSI circuits," *Design Systems for VLSI Circuits*, ed. G. D. De Micheli, A. Sangiovanni-Vincentelli, and P. Antognetti, Martinus Nijhoff Publishers, 1987.

[Sout83] J. R. Southard, "MacPitts: An approach to silicon compilation," *IEEE Computer*, vol. 16, no. 12, December 1983.

[THKR83] D. E. Thomas, C. Y. Hitchcock III, T. J. Kowalski, J. V. Rajan, and R. J. Walker, "Methods of automatic data path synthesis," *IEEE Computer*, vol. 16, no. 12, December 1983.

[TsSi86] C. J. Tseng and D. P. Sieworek, "Automated synthesis of data paths in digital systems," *IEEE Transactions on Computer-Aided Design*, vol. CAD-5 no. 3, July 1986.