Search Space Reduction in High Level Synthesis by Use of an Initial Circuit

Atsushi Masuda Hiroshi Imai Jeffery P. Hansen Masatoshi Sekine

ULSI research center,Toshiba corporation
Kawasaki,Kanagawa,Japan

Abstract

Most existing high-level synthesis(HLS) systems attempt to generate a circuit from a behavioral description "out of the void", using the entire design space as the search domain. Because of the vastness of the search space, it is impossible to do more than a coarse grain search, often resulting in inefficient designs. This approach, ignores the designer's knowledge of the general structure of the circuit to be synthesized. In this paper, we describe the HLS system SIDER (Synthesis by Initial Design Extension and Refinement). SIDER utilizes designer knowledge about the design space in the form of an initial circuit. By limiting search to the neighborhood of this initial circuit, much finer grain search can be performed yielding a higher quality design. The effectiveness of the SIDER approach is shown by HLS of a 300 line C description of 27 instructions from a MC6502 CPU.

I. Introduction

The major goal of current HLS research is to tie the behavioral level to the intermediate RT level. Algorithms for scheduling, allocation, binding and various combined methods have been available for some time. However, even with these methods, the circuits generated by even current state-of-the-art HLS synthesizers are of too low a quality to be commercially practical.

In HAL[1, 2, 3] and SAM[4] operators associated with the respective branches of a conditional statment can only share a single functional unit, and the control and interconnect cost for sharing is not taken into account. In MAHA[5] while the number of steps along the critical path can be reduced, when the critical path goes through a conditional branch, depending on which branch is taken during actual execution, the number of control steps may not be minimal. In ILP methods such as OASIC[6], circuits with a large number of operations can not be synthesized in a reasonable amount of time.

In path-based As Fast As Possible(AFAP) scheduling[7, 8], all paths from an initial node to a final node are extracted. Then, overlapping steps can be merged and a final schedule produced. However, the execution time for this method increases exponentially with the number of operations.

An allocation method[9] for use with AFAP scheduling has also been proposed. Each path extracted during the scheduling phase is broken down into sub-paths. Each sub-path along a path is assigned to a different state.

Global optimization can be performed on the initial allocation result obtained as described above. All sub-paths are analized to find mutually exclusive conditions on registers and functional units. The problem of finding the optimal sharing of registers and functional units is solved as a graph coloring problem. Since this is an NP complete problem, heuristics are used to find a solution.

In Tree-Based Scheduling[10], a Control/Data Flow Graph (C/DFG) with conditional statments is transformed into a tree structure by replicating the nodes after the end of each conditional. In this method, sharing of operators associated with different conditional branches is taken into account, but the effect of connection cost due to the sharing is not considered. Also, as of yet, there are no allocation algorithms associated with this scheduling method.

In a real-world design, the intractably large search space makes it infeasible to explore all possible designs. Because of this, most synthesis systems do a coarse grain search of this design space, often resulting in a circuit far from an optimal point.

The basic system structure can be taken as an initial value in the search for the optimal point in the design space. By searching for an optimal point near this initial value, the demand on the HLS can be greatly reduced. Therefore, in addition to the behavioral description, this basic system structure should also be made an input to a HLS system.

In one of recent study[11] for generating a control unit from data path information. Scheduling and allocation is performed using data path information given by a designer. However, this method requires designer to give complete data path information to the HLS system.

In this paper we propose a system in which the designer supplies an initial guess in the form of an initial circuit. The initial circuit need not be complete, as additional necessary connections will be added automatically.

The number of components and RTL description resulting from incremental synthesis is much better than those produced when synthesizing a whole description at once.

II. Basic System Structure

In this section the basic structure of SIDER is described. In the following sections, each phase of the synthesis process will be discussed.

Input is taken in the form of a behavioral description and an initial circuit. Behavioral descriptions are written in C language.

The initial circuit file describes the basic structure of the data path, including the input and output terminals, the functional units, the registers and interconnect. Profit values are given to components of the circuit to guide selection in the hardware allocation phase. Components with a high profit value are more likely to be selected, than those with a lower value.

The basic synthesis algorithm is outlined in Figure 1. Given a C/DFG G and an initial circuit D_0, the C/DFG is first factored by "condition pattern" into the individual DFGs G_i. Each of the G_i are then scheduled and partioned into "G-path" partionings P_i. The P_i are then mapped onto the "current design" D_{i-1} and any necessary additional circuitry is synthesized to generate the new current design D_i. Each of the steps in this process will be described in greater detail in the following sections.

$$\begin{array}{ccccccc} G & \longrightarrow & G_1 & , & G_2 & ,\ldots & G_n \\ & & \downarrow & & \downarrow & & \downarrow \\ & & P_1 & & P_2 & \cdots & P_n \\ & & \downarrow & & \downarrow & & \downarrow \\ D_0 & \longrightarrow & D_1 & \longrightarrow & D_2 & -\cdots\rightarrow & D_n \end{array}$$

Fig. 1. Basic Synthesis Algorithm

A. C/DFG and Data Path Representation

We will consider a C/DFG to be a graph G, with nodes n_i and edges e_{ij}. Each node represents an operator, and edges represent data flow. Nodes and edges are "colored" by their activating conditions as indicated by the conditional blocks in which they are contained.

In a similar mannar, the current design graph (CDG) is represented as a graph D with nodes N_i and edges E_{ij}. The nodes represent functional units and the edges represent interconnect. The CDG D_0 is generally the initial circuit, the CDGs D_i the design after refinement for each condition pattern, and the CDG D_n the final circuit which is translate into RTL.

B. Condition Analysis

The first step in condition analysis is to form a condition tree for the branching statments (e.g., `if` and `switch` statments). The condition tree is formed by writing a tautology expression $y_i + \overline{y}_i$ for each conditional statment in the description. For a nested conditional j in the `else` branch of conditional i, we write $y_i + \overline{y}_i(y_j + \overline{y}_j)$. Conditionals in series are simply the product of the tautology expressions for each of the individual conditionals. The condition patterns Φ_i are obtained by expending the tautology expression to a sum-of-products form.

Each term in the sum-of-products form corresponds to a condition pattern. After the condition patterns have been computed, the next step is to decompose the C/DFG G into DFGs for each condition pattern.

$$G = \sum_{i=1}^{n} \Phi_i \circ G_i$$

where the Φ_i terms represent the condition patterns, and the G_i represent the DFGs for each condition.

C. Scheduling

After obtaining the condition partioned G_i, the next step is to perform scheduling on each one. AFAP scheduling was selected for this prototype system because of the simplicity in implementation. In this stage, the required bit widths of the internal elements are computed from the bit widths of the input and output terminals. Based on this bit width, a hypothetical delay time for each operation is estimated. Scheduling is then performed according to the hypothetical delay time and the clock cycle time specified in the external specification. Operations that can not be processed within the system clock time are treated as multi-cycle operations.

Previous work [10, 9] has shown that substantial improvement in the design can be obtained when scheduling and allocation are performed simultaneously. Preliminary experimentation has shown that this capability could also be incorporated into the present system.

D. G-Path Partioning

For each G_i, the next step is generate the G-path partioning P_i. A G-path is a sequence of connected edges and nodes of the form $g_k = [n_{j_1}, e_{j_2 j_1}, n_{j_2}, e_{j_3 j_2}, \ldots]$. Each edge and node of a DFG falls into exactly one G-path. Each G-path must start at either an input terminal or an edge adjacent to another G-path, pass through a sequence of nodes and edges, and end at an output terminal or another edge adjacent to another G-path.

In general, there are many ways of labeling a DFG G_i to obtain a G-path partioning P_i. Currently G-paths are obtained by repeatedly extracting the longest path, until all edges in the graph have been selected. Figure 2 is an example of G-path partioning.

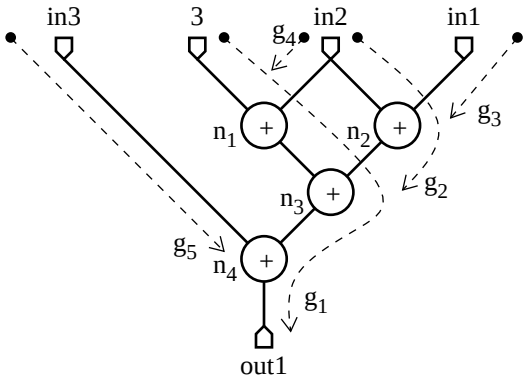

Fig. 2. An example of G-path partioning

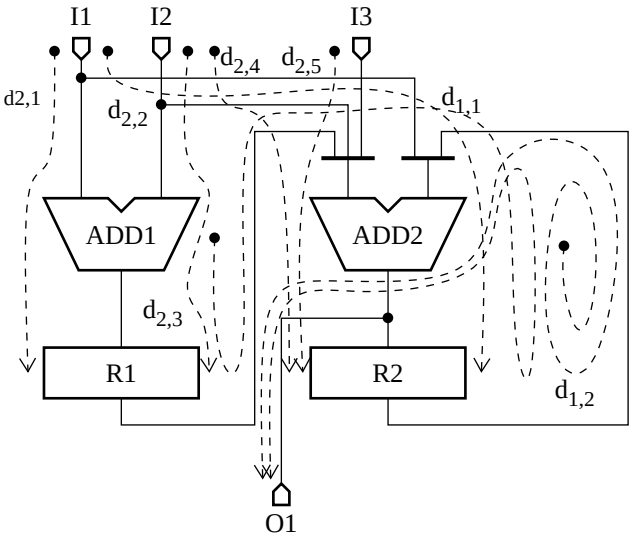

Fig. 3. An Example of Initial Circuit

E. Candidate D-Path Selection

After obtaining the G-path partionings P_i, it is necessary to find a set of candidate "D-paths" for each G-path. A D-path is a sequence of nodes and edges in the CDG of the form $d_{ki} = [N_{j_1}, E_{j_2 j_1}, N_{j_2}, E_{j_3 j_2}, \ldots]$. Unlike G-paths, nodes and edges in a D-path may be repeated as long as the sequence forms a path.

A D-path d_{ki} is "compatable" with a G-path g_k if and only if corresponding elements are compatible. Elements are compatible if they are both nodes, and the design graph nodes represent functional units that can execute the operation corresponding to the DFG nodes. Terminal nodes have the addition restriction that they must have the same label (i.e., G-path input terminal "A" must map to D-path input terminal "A"), and G-path terminals for constants are considered "free" and do not need to match a D-path element.

The candidate D-paths are generated by depth first search of the CDG. A search tree is formed with each element of the G-path as a decision node, and compatible elements from the current design as the decision branches. Each element of the current design has a profit value associated with it. The decision branches are selected randomly using the profit values to weight the selection. This process is repeated to extract as many D-paths as required. The random nature of the selection helps to provide D-path candidates that are distributed more evenly about the design space than simply taking the paths with the highest total profit values.

When no D-path candidates can be extracted from the current design, additional connections and functional units are added to the current design. The position to add the additional components is determined by the profit values of the existing components, and the profit for the new components is set lower than the surrounding components.

The number of D-path candidates must be selected is based on the size and complexity of the initial circuit. The selection of this limit has a great effect on the CPU time required for the subsequent processing, and on the quality of the final circuit. This limit is given as the synthesis parameter T_d.

In general, there are an virtually unlimited number of data path designs that can be used to implement a C/DFG. The problem of finding the optimal one from an unconstrained design space is an NP hard problem. We use the selection of an initial circuit to limit the search space, and avoid the NP-hard problem. In other words, we consider a limited number of D-paths for each G-path, and find a local optimum near the initial circuit. Examples of D-paths are shown in Figure 3 for G-paths g_1 and g_2 from the Figure 2 example. The paths $d_{1,i}$ correspond to g_1 and the paths $d_{2,i}$ correspond to path g_2 are shown. D-paths for other G-paths are not shown.

F. D-Path Optimization

After generating D-path candidates for each G-path, the next step is to choose a non-conflicting combination of bindings. A conflict occurs when two D-paths use the same functional unit or path in the same clock cycle. Three types of conflicts are considered: functional unit, interconnect and I/O port.

The optimal solution is the conflict free set of bindings that maximize the total profit. We solve this as a modified

form of the knapsack problem:

$$\max \quad z = \sum_{i=1}^{m} \sum_{j=1}^{n} p_{ij} x_{ij}$$

s.t.

$$\sum_{j=1}^{n} w_{ij} x_{ij} \leq c_i, \quad i \in M = \{1, \ldots, m\},$$

$$\sum_{i=1}^{m} x_{ij} = 1, \quad j \in N = \{1, \ldots, n\},$$

$$x_{ij} \in \{0, 1\}, i \in M, j \in N$$

where x_{ij} is 1 if node i of the DFG is bound to register or functional unit j of the CDG, p_{ij} is the profit for this binding, w_{ij} is the bit-width of element i when realized on element j of the CDG, and c_i is the total bit capacity of the register or functional unit i in the CDG. In the analogy to the knapsack problem, functional units and registers of the CDG are the knapsacks, and operators and edges in the DFG are the objects to be placed in the knapsacks.

The problem is solved using a branch-and-bound based algorithm, choosing a D-path for each G-path. When searching a D-path branch d_{ij}, for G-path g_i, if d_{ij} conflicts with any of the previously assigned bindings. If not, we then try place all of the G-path objects in the corresponding D-path knapsacks. If the capacity of any knapsack is exceeded, the path was in conflict, or the estimated maximum profit from this point is less than the current bound, we must abandon the branch and try another binding.

G. Conflict Resolution

In applying the branch-and-bound algorithm, it is possible that all bindings conflict and no solution can be found. When this occurs, we must apply the conflict resolution procedure, and add hardware to the CDG to remove the conflict.

In doing conflict resolution, we start at the search path which failed at the deepest point, and among those we take the one with the highest profit. We then record the G-path on which the conflict occurred, and continue the search without a binding for that G-path.

After obtaining a solution and a set of G-paths that caused a conflict, we then compute the minimal set of components that need to be added to the circuit to resolve the conflicts.

III. Experimental Results

To demonstrate the effectiveness of SIDER, a 300 line description was prepared. The description, written in C, encoded 27 instructions of a MC6502 processor.

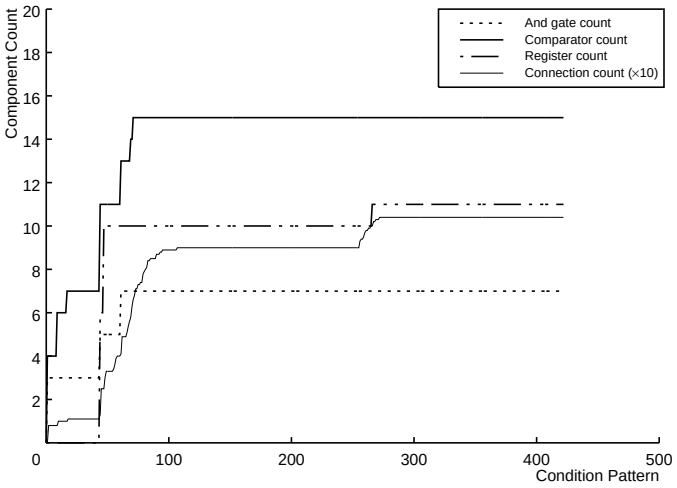

Fig. 4. Synthesis Result for $T_d = 10$

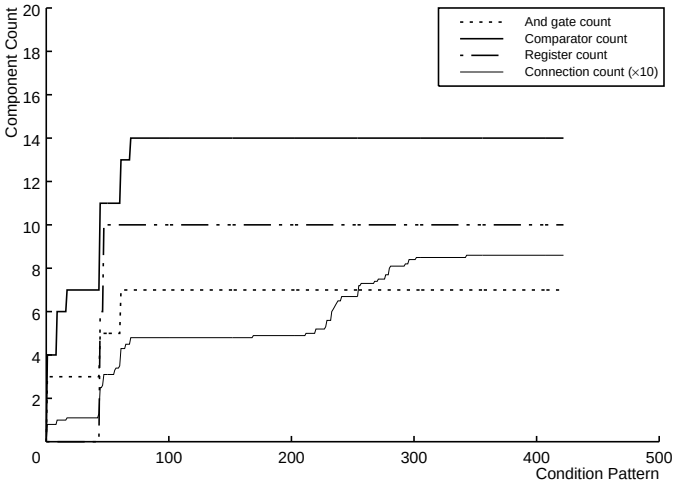

Fig. 5. Synthesis Result for $T_d = 23$

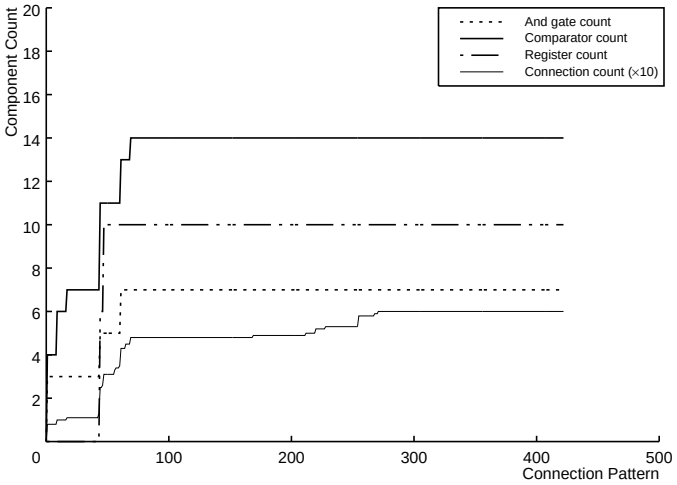

Fig. 6. Synthesis Result for $T_d = 24$

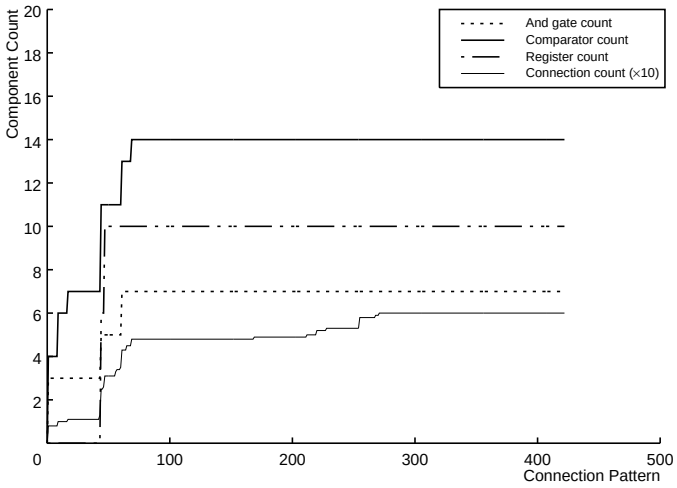

Fig. 7. Synthesis Result for $T_d = 64$

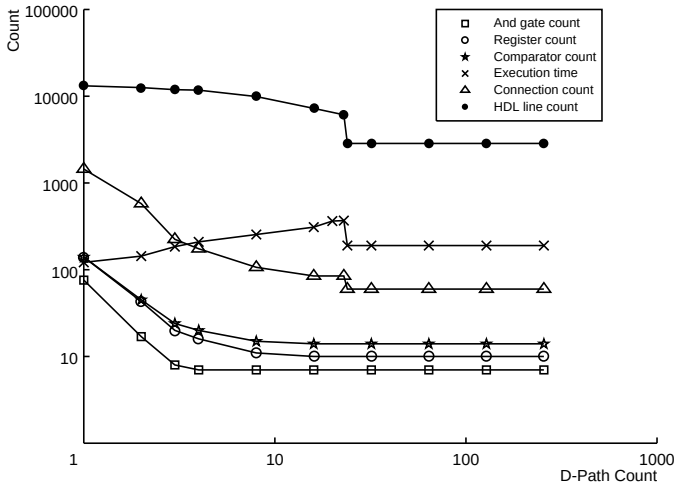

Fig. 8. Synthesis Result by T_d

There are `if` statments nested up to 12 levels deep for instruction decoding, and a total of 422 condition patterns.

The initial circuit consists of the input and output ports, four registers, and functional units for each of the operators in the descriptions. Two functional units are provided for some of the more common operations (addition, logical AND, and logical OR). Some initial connections are also given.

Figures 4 through 7 show the size of the CDG (in terms of number of components for several component types) as it is extend for each condition pattern Φ_i. The figures show the results for T_d of 10, 23, 24 and 64 respectively.

In all of the cases, most of the circuitry is added to the CDG early in the synthesis process. For DFGs G_i with a large number of G-paths, many components are added at once. After about the $i = 50$, nearly all of the necessary functional units have been added, and only the connection count changes significantly.

Consider the graphs for $T_d = 10$ and $T_d = 23$. Up to $i = 50$, the two graphs are identical. At this point, however, the $T_d = 23$ case is able to find connections that were not found in the $T_d = 10$ case. This results in a savings of one (multi-bit) AND unit and one register.

The number of interconnects is decreased from 104 to 86. When we increase T_d from 23 to 24, we find that number of interconnects drop sharply to 60, representing a local minima in the solution space. Further increase of T_d results in no additional reduction of the circuit size.

The synthesis result as a function of T_d are shown in Figure 8. The number of comparators, registers, and "AND"-gates converge to 14, 11, and 7, respectively (for $T_d = 10$). For small T_d, the generated RTL-description is over $10,000$ lines. As T_d is increased to 23, the RTL description drops to $6,200$ lines. When T_d goes from 23 to 24, there is a sudden drop to $2,856$ lines and the result does not change for further increases in T_d. Logic synthesis of the $T_d = 24$ RTL description yields a circuit with 1.2K gates.

The main cause of the circuit size reduction is the reduction in number of connections, since this simplifies the control circuitry generate by the logic synthesizer. The number of connections drops from $1,450$ at $T_d = 1$ to 86 at $T_d = 23$ and 60 at $T_d = 24$. When we examine the resulting RTL descriptions for the $T_d = 23$ and $T_d = 24$ cases we notice that in the $T_d = 23$ case there are many operations with many different conditions, but in the $T_d = 24$ case, there are many more operations controlled by the same condition.

The empirically estimated CPU time between $T_d = 1$ and $T_d = 23$ is $t = 120(T_d)^{0.36}$. The decrease in the circuit size at $T_d = 24$ reduces the search space and thus the CPU time actually drops.

IV. Incremental Design

Practical circuits in the field are often designed incrementally. To show how SIDER is useful in incremental design, we broke down the 27 instructions of the MC6502 into the four design groups.

Results from the incremental design experiment are shown in Table I. The design groups are created one at a time and merged into the design $B_n = \sum_{i=0}^{n} b_i$. Note that B_3 is the same as the full specification used in Section III. Design B_0 is synthesized with the same initial circuit (D_0) as in the previous example, and each B_i is synthesized using the result from the previous step as the initial circuit. The number of connections manually added are shown in parenthesis in the column for number of connection in the initial circuit.

In the synthesis results section of the table, the number of connections broken down into those that were in

Design			Initial Circuit		Synthesis Result						
Name	T_d	Inst.	Name	Conn.	Name	Conn.	Eq.	Reg.	AND	RTL	CPU
B_0	24	4	D_0	67	D_1	6 + 5 = 11	3	0	3	340	2.3
B_1	24	8	D_1	11	D_2	11 + 7 = 18	8	0	4	1101	19.7
B_2	24	16	D_2	20(2)	D_3	13 + 8 = 21	8	0	4	1778	126.9
B_3	24	27	D_3	47(26)	D_4	44 + 44 = 88	9	11	4	8742	411.5
B_3	24	27	D_0	67	D_5	17 + 43 = 60	14	10	7	2856	231
B_3	48	27	D_3	47(26)	D_6	30 + 27 = 57	9	11	4	2795	314.8

TABLE I
INCREMENTAL SYNTHESIS RESULTS

Initial Circuit			Synthesis Result		
Name	FUs	#Reg.	#Reg.	#Terms.	#States
count	1xINC, 1xDEC, 1x(==)	1	0	7	1
gcd	1xSUB, 1x(<), 1x(!=)	2	0	9	1
hal	3xMUL, 1xSUB, 1xADD	3	3	15	4
MAHA	1xADD, 1xSUB, 6xINPUT	3	3	6	8

TABLE II
DATA ON INITIAL CIRCUITS

the initial circuit and new ones added to the circuit, are given along with the number of components for several component types, the number of lines in the generated RTL description and the synthesis time are given.

The results indicate that while the number of components resulting from the incrimentaly produced result is comparable to those produced when synthesizing the whole description at once, the RTL descriptions size is much larger (8742 lines versus 2856 lines). If we resynthesize using a large T_d, however, the number of lines generated in the incrmental case drops to 2795, actually better than in the non-incrmental design case. These results compare favorably to our experinces in the case of human designers.

Four HLS benchmarks, rewritten in C , where chosen for comparison with other methods, one benchmark (HAL) without conditional statements, and three benchmarks (count,GCD and MAHA) with conditional statements. For each of these examples, an initial circuit was supplied. Data on the initial circuits are shown in Table II.

In the HAL example, since there are no conditional statments, the synthesis reduces to a simple AFAP scheduling result. In the MAHA initial circuit there are two functional blocks (ADD and SUB), 6 input terminals, and some simple interconnect. The shortest path is three steps, and the longest is eight.

V. Conclusion

In this paper we have proposed a new HLS approach which reduces the design space by use of an initial circuit. It is shown that by using an initial circuit to limit the search space we can synthesize large-scale circuits in reasonable time with results comparable to human designers.

References

[1] P.G. Paulin, J.P. Knight and E.F. Girczyc "HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis", From *23th ACM/IEEE Design Automation Conference*, pp. 263-270 (1986)

[2] P.G. Paulin and J.P. Knight "Force-Directed Scheduling in Automatic Data Path Synthesis", From *24th ACM/IEEE Design Automation Conference*, pp. 195-202 (1987)

[3] P.G. Paulin and J.P. Knight "Force-Directed Scheduling for the Behavioral Synthsis of ASICs", From *IEEE Transactions on Computer-Aided Design*, vol. 8, pp. 661-679 (1989)

[4] R.J. Cloutier and D.E. Thomas, "The Combination fo Scheduling, Allocation, and Mapping in a Single Algorithm", From *Proc. 27th Design Automation Conference*, pp. 71-76 (1990)

[5] Alice C. Parker, Jorge "T" Pizarro and Mitch Mlinar, "MAHA: A Program for Datapath Syntethesis", From *23rd ACM/IEEE Design Automation Conference*, pp. 461-6 (1986)

[6] C.H. Gebotys and M.I. Elmasry, "Simultaneous Scheduling and Allocation for Cost Constrained Optimal Architectural Synthesis", From *Proc. 28th Design Automation Conference*, pp. 2-7 (1991)

[7] R. Camposano and R.A. Bergamaschi, "Synthesis Using Path-Based Scheduling: Algorithms and Exercises", From *Proc. 27th Design Automation Conference*, pp. 450-5 (1990)

[8] R. Camposano, "Path-Based Schedulig for Synthesis", From *IEEE Transactions on CAD*, vol. 10, pp. 85-93 (1991)

[9] R. A. Bergamaschi, R. Camposano, and M. Payer, "Data-Path Synthesis Using Path Analysis", From *Proc. 28th Design Automation Conference*, pp. 591-6 (1991)

[10] S.H. Huang, Y.L. Jeang, C.T. Hwang, Y.C. Hsu, and J.F. Wang, "A Tree-Based Scheduling Algorithm For Control-Dominated Circuits", From *Proc. 30th Design Automation Conference*, pp. 578-82 (1993)

[11] T.Miyazaki,M Ikeda, "High-Level Synthesis Using Given Datapath Information", From *IEICE Transactions fundamentals,vol.E76-A*,pp.1617-25 (1993)