Data Availability Scheduling for Synthesis Beyond Basic Block Scope

Jongsoo Kim

Abstract

High-Level synthesis of digital circuits calls for automatic translation of a behavioral description to a structural design entity represented in terms of components and connections. One of the critical steps in high-level synthesis is to determine a particular scheduling algorithm that will assign behavioral operations to control states. A new scheduling algorithm called Data Availability Scheduling (DAS) for high-level synthesis is presented. It can determine an appropriate scheduling algorithm and minimize the number of states required using data availability and dependency conditions extracted from the behavioral code, taking into account resource constraint in each control state. The DAS algorithm is efficient because data availability conditions, and conditional and wait statements break the behavioral code into manageable pieces which are analyzed independently. The output is the number of states in a finite state machine and shows better results than those of previous algorithms.

I. Introduction

The goal of high-level scheduling is to optimally assign the operations specified in the behavioral description of a synchronous digital system to a sequence of control steps, which correspond to a basic machine cycle. Control steps are then collected into groups and each group is assigned to a state(control state) of an associated synchronous sequential control circuit. State transitions of the control circuits supply synchronizing trigger signals to the data paths. A good schedule will result in efficient utilization of hardware resources in the data paths and the smallest number of states in the control circuits. There are two basic types of scheduling algorithms [9, 13, 17]: time-constrained and resource-constrained. In time-constrained scheduling, the maximum time allowed to process data from the input stream is specified and the main objective of scheduling is to minimize the necessary hardware resources. The main objective of resource-constrained scheduling is to get the best performance under given hardware constraints, such as chip area or the number of functional units. In order to balance performance and area, the two approaches must be combined. However, the high-level scheduling task, which determines the area-speed trade-offs, is too complex to allow a general optimal solution. It is known that scheduling is an NP-complete problem [2,3].

This work was supported in part by the University of Ulsan Research Foundation 97.
Manuscript received August 7, 1996; accepted December 2, 1997.
The author is with the School of Electrical Engineering at the Ulsan Univ.

In order to simplify the scheduling problems, many high-level scheduling algorithms consider only specific classes of applications, such as digital filters and pipe-lined behavioral descriptions [1,7,10-12,16,18]. For example, Force-Directed Scheduling(FDS), which is a time-constrained algorithm, aims at reducing the total number of functional units by redistributing operations that use the same type of functional unit evenly into the available states through statistical method [20-23]. Integer Linear Programming(ILP) is a mathematical approach to find optimal schedule in terms of the number of functional units [15]. Besides these representative algorithms there are many other scheduling methods in the context of high-level synthesis, such as percolation-based scheduling, simulated annealing, and path-based scheduling [5,9,19,24]. Path-based scheduling is quite different from the other algorithms in which minimizing the number of functional units is the main consideration. The main objective of path-based scheduling is to minimize the number of control states needed to execute the longest paths under given timing and area constraints. The main disadvantage of the path-based scheduling is its computational complexity which is exponential in the number of conditional branch and synchronization statements, since it must expand all possible execution paths. A more detailed discussion can be found in [2].

In this paper, data availability scheduling(DAS) is presented for synchronous digital systems written in VHDL behavioral code. This scheduling can overcome the shortcomings of the path-based algorithm, while accomplishing the goal of minimizing the number of states. The central theme of the DAS is that the state minimization problem is highly constrained. Thus, general global optimization, which introduces exponential computational complexity, is unnecessary. The DAS algorithm identifies a number of operations which unequivocally require state transitions. Optimization of the assignment of operations to states can then be performed on local segments of the execution paths between the operations which require state transitions. The proposed DAS algorithm exploits the available parallelism limited by resource constraints in minimizing the number of states.

This paper is structured as follows. The next section describes the foundation of DAS scheduling algorithm. Section 3 presents how to find state transition conditions. Section 4 gives the details of the algorithm. This paper ends with a comparison of the results obtained by this algorithm with other reported short results.

II. Foundation of the DAS Algorithm

One foundation of this new scheduling algorithm comes from the definition of sequential circuits that process operations according to a given deterministic finite automaton. Data availability is a basic characteristics of sequential circuits. Thus, the DAS algorithm suggests a new idea for how to easily find such points based on data modality. That is, the proposed algorithm can determine the exact time that data is needed. If data must be read from a variable within a behavioral process body before writing the data, it must be already be saved in a storage unit. Then, a trigger signal is required to write data to storage, and this trigger signal must be linked with state information. Therefore, the data availability condition can be used to find state separation points.

Another foundation is found in the allocation process. Allocation maps the scheduled assignment operations into registers if LHS variables need to be stored. Trigger signals for loading new data into these registers are generated by state transitions of the finite state machine's control circuits. The state transitions determined by register allocation can be anticipated during scheduling and can identify operations which must be scheduled into separate states. Furthermore, the number of registers and interconnections being determined in the allocation phase can be estimated in advance during scheduling.

III. State Transition Conditions and RBS Variables

Some state transitions are required in order to trigger the storing of data in a register. Such state transition conditions are classified into two different types by Camposano [2]: intrinsic and extrinsic. Intrinsic conditions express the fact that different data cannot be stored simultaneously into one variable. That is, two sequential assignments having the same left hand side (LHS) variable must be performed in two separate states. Extrinsic conditions are derived from explicit statements in the design description. The *wait* statement of the hardware description language is one example of extrinsic conditions. Such conditions need a register in order to synchronize with other statements. Another example of extrinsic condition is the resource constraint. That is, if a designer specifies the maximum number of available hardware functional units, the scheduling algorithm must separate the operational nodes due to violation of the given hardware constraints.

This research therefore begins to identify necessary state transition conditions by analyzing register requirements. The easiest way to find register requirement conditions is to calculate the lifetime of variables. If the lifetime of a variable must span more than one state, then a register is needed to hold previous data for the next state. If the clock cycle is predetermined and it is relatively short, then all the operations which manipulate a variable for relatively long periods cannot be executed within one clock cycle. Thus another state is necessary to execute the remaining operations and the variable has to be saved in a register. If the clock cycle is not predetermined, the clock period can be made long enough to process the series of nodes in one state. This method does not require an extra register, but it may slow down the overall processing speed. The former method can speed up the process, while requiring an extra register. Some researchers have investigated scheduling based only on such register requirement conditions [25], but the computation time and the number of total registers determined by the lifetime method were greater than those of the path-based method [2].

Another method of finding register requirement conditions is to check the access modality of variables. This is a new approach suggested in this paper as mentioned in the previous section. That is, when a variable is read before an assignment to it has taken place within the innermost loop or branch construct, it requires a register. Thus, the register requirement condition is determined from a "Read Before Set"(RBS) variable usage in contrast to the path-based scheduling algorithm which exhaustively traces to find out state separation points.

Typical behavioral descriptions do not include many intrinsic state transition conditions. Generally, only VHDL variables are candidates for RBS status. The other two VHDL objects (signal and constant) do not require registers. But all variables need not be stored in registers always. For instance, variables that do not need to hold values for more than the duration of one cycle(i. e. not across state transitions), do not need storage or state transition. However, a variable, whose value is read within a loop body before it is assigned, needs a register to hold the datum for successive iteration, even if it had been assigned a value before the loop was entered. Therefore, it is possible to identity more RBS nodes through this extended definition of RBS.

In addition to the extended RBS conditions and the extrinsic conditions derived from hardware constraints, there are two more state transition conditions. The first is a technology dependent condition. According to the previous RBS definition, a test condition that is the header of a loop or branch need not always have RBS status. Therefore, the header node can be combined with a previous control step in constructing the states. But this combination may sometimes lead to incorrect results. For instance, assume that the current input data in master-slave registers will not be available at the output port until the next clock cycle. Then, the test condition of a loop or branch can only be safely evaluated when the test value is available in the next state. A similar case occurs when the values of test variables in a loop statement are initialized first and subsequently changed inside the loop body. Thus, the loop test condition and assignment to the tested variable(s) must be separated to ensure a safe operation sequence. In both cases, the available time of test condition's output will depend on the length of propagation delays. Therefore, this condition depends on the technology mapping

The second condition stems from using a programming oriented design description. In a programming language, the flow of execution cannot jump directly into a loop body. This rule must also be enforced in scheduling to prevent unwanted side effects. It is sufficient to consider the header node of the loop statement because it must always be executed first. If the header node was combined with nodes from outside the loop in one state, nodes outside the loop would be executed repeatedly and create unwanted side effects. Therefore, the starting node of a loop structure cannot be merged with any previous node. In summary, state transition conditions resulted from RBS(collectively called RBS conditions) and others are summarized as follows:

(1) Variable can be assigned only once in one state.
(2) A *wait until* statement must be scheduled separately from previous nodes.
(3) An output port can be written only once in one state.
(4) The variable assignment and variable test statements must be separated.
(5) Functional units can be used only once in one state.
(6) RBS variables, which need registers, are defined by limiting the scope of the RBS test to within a loop body.
(7) The header statement of a loop cannot be in the same state with previous statements as a side effect occurs outside the loop.

IV. The Data Availability Scheduling Algorithm

In this section, the DAS algorithm is defined. The goal of the DAS algorithm is not only to reduce the computation time, but also to minimize the number of control states. To accomplish both goals, the control and dataflow graph(CDFG) is the most suitable intermediate representation, because it exhibits the available parallelism through combined dependency and control flow. In addition, the CDFG can provide critical path information, which is essential to the DAS algorithm. The critical path is the longest path from the first to the last control step and indicates the maximum number of control steps. Hardware constraints may force the number of control steps to be increased from what had been determined from the purely behavioral VHDL code. In case of multiple equidistant paths, the path that traverses more RBS nodes is selected as the critical path. RBS nodes correspond to operations which satisfy a RBS condition. The RBS nodes in the critical path are called *reference* nodes.

```
Call ALAP(N) and ASAP(N);              --unbalanced schedule.
Calculate_Mobiltity(N);                --mobility calculation.
Mark_Critical_Path(N);                 --check RBS conditions.
Find_Critical_Path(N);                 --longest path with most RBS nodes.
i = 1;
start = 1;
while i <= last do                     --loops until the last control step.
   if (RBS(Ncp,i) = true) then         --between pairs of adjacent RBS nodes.
        Apply_Constraint(start, I-1);  --resource constraints to set RBS.
        Move_Node(start, I-1);         --move RBS nodes to subsequent
        start = I;                     --reference node.
    endif
    i++;
endwhile
Apply_Constraint(start, last);         --take care of nodes between last RBS
Move_Node(start, last);                --node and bottom node.
Group_Node(N);                         --collect nodes between RBS nodes.
Merge_State(M);                        --apply chaining constraint.
Find_State_Transition(L);              --determine transition conditions.
```

Fig. 1. Data Availability Scheduling Algorithm.

Figure 1 shows the pseudo code of the DAS algorithm. The call to ASAP and ALAP is necessary to calculate node mobility, where N is the set of nodes to be scheduled. The function "Mark_RBS(N)" uses conditions (2), (4), and (6) listed in Section 3 to find RBS nodes and set an RBS flag. Next the function "Find_Critical_Path(N)" uses the RBS flag to determine the critical path and returns the critical path that has the most RBS nodes. When there is more than one critical path with the same number of RBS nodes, one critical path is arbitrarily chosen. In the *while* loop, N*cp,i* are the nodes in the critical path, where the loop iterates *i* times over the control steps until *i* reaches the value of last. The parameter *last* is the value of the last control step number. The *while* loop processes nodes between pairs of adjacent RBS nodes. It consists of two functions; "Apply_ Constraint(*start, i-1*)" and "Move_Node(*start, i-1*)". Both functions use the control steps of a RBS pair of RBS nodes as parameters. The first function "Apply_Constraint(*start, i-1*)" applies conditions (1), (3) and (5) to set the RBS bit of nodes in all paths between control steps *start* and *i-1*. This function also applies the designer's constraints, such as the maximum number of functional units. The second function "Move_Down(*start, i-1*)" moves the nodes marked by the function "Apply_Constraint(*start, i-1*)" to the control step *i* of reference node N*cp,i,* if it is within the node's mobility ranges. This reduces the number of state transitions required, because all nodes in one control step can be triggered by the same clock edge. If the node cannot be moved due to mobility, then the node will require an additional unconditional state transition. Notice that nodes are moved only if their RBS flag is set.

After the *while* loop, a set of nodes between control steps of the last RBS reference node in the critical path and the last control step will not have been processed unless the last control step contains an RBS nodes. Thus, the two functions "Apply_Constraint() and Move_Node()" outside the *while* loop are to process the remaining nodes. The next function "Group_Node(N)" simply collects nodes separated by RBS nodes into M groups. After that, the function "Merge_State(M)" is called to combine compatible groups associated with new RBS nodes created by the function "Apply_Constraint()" which cannot be merged with the preceding groups. Thus, the function "Merge_State(M)" investigates small numbers of merging candidates instead of processing the total number of groups M. Finally, the function "Find_State_transition(L)" builds the state diagram of the control circuit based on the transition conditions required by the nodes within the groups. L is the number of state transitions. In general, these conditions are derived only from branch, loop or wait statements; Therefore, this function processes a small number of groups only. Further details of state transition formulation can be found in [2].

To illustrate the DAS algorithm, we use the prefetch example shown in [2]. Figure 2 is the behavioral code of prefetch written in VHDL. The path-based scheduling expanded 3 paths and built a finite state machine with 2 control states after applying the clique algorithm. See more detail in [2]. Figure 3 is the CDFG where the asterisks indicate the RBS nodes. All nodes except 1, 2, and 3 have 0 mobility in this graph. There are 2 critical paths: one is P1 = {4, 5, 6, 7, 8, 10} and the other is P2 = {4, 5, 6, 7, 9, 10}. We choose P1 arbitrary. Thus the reference nodes are {5, 7, 8}. The functions, Apply_Constraint(*start, i-1*) and Move_Node(*start, i-1*), are called with the control step parameters in the sequence of (1, 1), (2, 3), and (4, 4). The above two functions cannot find any movable RBS nodes that must be aligned to reference node. The function Group_Node(N) groups the nodes 1, 2, 3, 4, 5, and 6 into one group, and nodes 7, 8, 9, and 10 into a second group. Thus, there are two states. Finally the state transitions between these states are determined by the condition "ire='1'". Therefore the DAS algorithm also yields the same number of states as the path-based algorithm.

```
entity prefetch is
  port (branchpc, ibus    : in bit32;
        branch, ire       : in bit;
        ppc, popc, obus   : out bit32);
end prefetch;

architecture behavior of prefetch is
begin
  process
    variable pc, oldpc : bit32 :=0;
  begin
    ppc <= pc;        -- 1
    popc <= oldpc; -- 2
    obus <= ibus + 4;-- 3
    if ( branch = '1')-- 4
      then
        pc := branchpc;-- 5
    end if ;          -- 6
    wait until (ire = '1')-- 7
    oldpc := pc;      -- 8
    pc := pc + 4;    -- 9, 10
  end process;
end behavior;
```

Fig. 2. Behavioral Description Example.

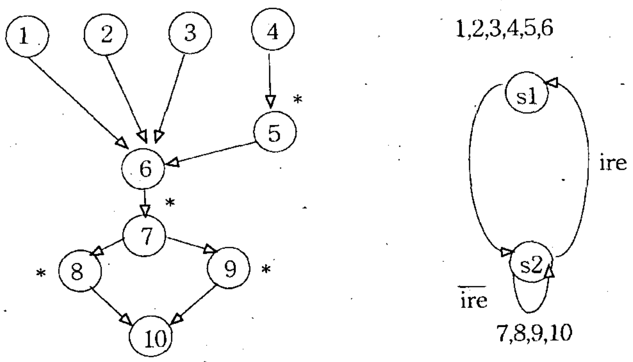


Fig. 3. CDFG and Final States.

The computational complexity of the DAS algorithm depends on the total number of nodes, the shape of the CDFG, and the number and position of the RBS nodes in the CDFG. In extreme cases, there are N RBS nodes in the critical path of length N. Then there is nothing for the scheduler to do. But it is rare to find this kind of description. In general, the number of RBS nodes will be smaller than the length of the critical path. Therefore, the order of computation is given by $O(N \times B)$, where N is the number of RBS nodes in the critical path, and B is the number of nodes that must be scanned with each RBS node. However, when the number N increases for a given CDFG, the number B gets smaller. Therefore, the total computation overhead will saturate at a certain value. More precisely, the computation time will be given by $\sum_{i=1}^{N} Bi$,where Bi is the number of nodes being scheduled at the *i*th RBS node. This analysis shows that the algorithm operates most efficiently when there are many RBS nodes in the critical path.

V. Results and Comparisons

Only two papers have reported results on the benchmark program for the general finite state machine [2,25], and these two reports have used different design environments. The bridge synthesis system has used a behavioral description language called FDL2, while the path-based system has used the description of *the 1989 Workshop on High-Level Synthesis* for the Intel 8251 USART[27]. The same benchmark program written in VHDL [28] is used in this work.

Table 1. Comparison with other algorithms.

Model	Method	Add	Sub	Mul	state	CPU
COUNTER	path	1	1	-	1	<1
	DAS	1	1	-	1	<1
GCD	path	-	1	-	2	<1
	DAS	-	1	-	2	<1
PREFETCH	path	1	-	-	4	<1
	DAS	1	-	-	4	<1
TLC	path	-	-	-	8	<1
	DAS	-	-	-	6	<1
Diffeq	path	1	1	2	4/1	<1
	force	1	1	2	4	<1
	DAS	1	1	2	4/1	<1
8251 xmtr	path	-	1	-	22	357.6
	bridge	-	1	-	37	300.2
	DAS	-	1	-	20/16	<1

Table 1 lists the number of states generated by different algorithms under the stated constraints on the number of functional units, and the reported synthesis CPU times for three different benchmark problems. COUNTER, GCD, and PREFETCH are written in VHDL [27]. COUNTER is a 4-bit counter. GCD is a greatest common divisor example. PREFETCH is an instruction fetch description different from Figure 2. The TLC example is the traffic light controller from [28]. The Diffeq example is the differential equation solver from [23, 28]. The 8251 xmtr indicates the transmitter part of the Intel 8251 USART. In the TLC example, path-based scheduling created eight states, while the DAS algorithm needs only six states. In the Diffeq example, force-directed and path-based scheduling yield the same result. The path-based algorithm specifies only one state, if the exit condition of the loop is assumed true at start[2]. As the path-based scheduling, the DAS algorithm also yields one state, because there are no RBS nodes in the behavioral description.

The total number of nodes of the 8251 transmitter part in the VHDL benchmark program is 97. These nodes are distributed along 42 control steps in the control flow graph. Initially, there are 28 RBS nodes in the control flow graph. The DAS algorithm generates only 16 states, if a series of test conditions of "if statements" can be processed in one state. If test condition must consume one state each, then 20 states are needed. In general, it is plausible that a number of test conditions can be processed within one cycle, since the test conditions can be realized by simple combinational circuits. The bridge system yielded 37 states for the same design problem (Intel 8251), while the path-based algorithm gave 22 states under more computation time.

The execution times of these algorithms couldn't easily be compared, since all scheduling algorithms were run on different machines. The path-based algorithm was run on am IBM 3090/200 machine under CMS VM/SP 4.2 with less than 7 Mbytes of memory. For the DAS algorithm, the computation time is almost negligible on a PC even with an Intel CPU 486/33 and 8 Mbytes of RAM. The indicated computation times of DAS algorithm include only the scheduling overhead from CDFG and exclude the remaining logic synthesis.

VI. Conclusions

There are two main contributions of the DAS algorithm. First, it can detect whether the behavioral input description may be mapped to a FSM or not. None of the available scheduling algorithms perform equally well in diverse domains. Some algorithm, such as percolation-based scheduling, is suitable for pipeline architectures, while others, such as DAS or path-based scheduling, are suitable for control-oriented descriptions. Thus, a particular scheduling algorithm may require more functional units and more complicated control circuits than another algorithm on a particular design problem. Therefore, it is necessary to know which algorithm is suitable for a given design. Unfortunately, previous algorithms were tested only on small examples in limited domains. No discussion was provided as to which domains the algorithms are suitable for. The new scheduling algorithm suggested in this paper can be used to determine an appropriate scheduling algorithm depending on the input description, even though its ultimate goal is to schedule the operations in the input description using a finite state machine. Since our algorithm locates the registers that need trigger signals to hold information and the wait until statements that need state transition information, it is possible to detect whether a behavioral description belongs to the class of control or popeline-oriented problems. If there are many RBS nodes in the input description, finding an optimal control circuit is important, and DAS is appropriate. If there are no RBS nodes, a behavioral input description can be synthesized either as a pipeline or as a purely combinational logic circuit by the ILP algorithm, for example.

Second, the new scheduling algorithm can reduce not only the final number of states but computation overhead as well. The path-based system uses clique partitioning, which is very costly in computation time, to find a set of minimum, non-overlapping intervals. It also uses a control flow graph modified by another scheduling algorithm, such as list scheduling, to generate the execution paths. The bridge system has focused on detecting the variables and signals through the lifetime scheme. Although the bridge algorithm has used a local slicing technique, it still has limitations on branching constructs. These approaches have also resulted in additional computation overhead because they operate globally. The DAS algorithm uses the CDFG to exploit the parallelism and the RBS concept to make local slices along the critical path. These two factors can contribute to reducing the number of states and the computation time compared with other scheduling algorithms.

As future research, the concept of RBS conditions can also be applied in path-based scheduling. The RBS nodes in the control flow graph can localize processing when constructing the interval graph. Therefore, the computational overhead can be lessened. The register requirement conditions uncovered in the DAS algorithm will help to more tightly integrate scheduling and allocation which should improve the efficiency of high-level scheduling and make its results more accurate. Finally, more study is needed to address the lack of pipeline scheduling capability, and to make scheduling sensitive to technology dependency.

References

[1] A. Aiken and A. Nicolau, "Perfect Pipelining: A New Loop Parallelization Technique," European Symposium on Programming, 1988, pp. 221-235.

[2] Raul Camposano, "Path-Based Scheduling for Synthesis," IEEE Transactions on Computer-Aided Design, vol. 10, no. 1,January 1991, pp.85-93.

[3] Raul Camposano and W. Wolf, "High-Level Synthesis Systems," Kluwer Academic Publishers, Boston, 1991.

[4] Raul Camposano, "From Behavioral to Structure: High-Level Synthesis," IEEE Design & Test of Computers, vol 7, Issue. 5, Oct. 1990, pp.8-19.

[5] Raul Camposano and R. A. Bergamaschi, "Synthesis Using Path-Based Scheduling: Algorithms and Exercises," Proc. 27th ACM/IEEE Design Automation Conference, June 1990, pp. 450-455.

[6] Raul Camposano and Wolfgang Rosenstiel, "Synthesizing Circuits from Behavioral Descriptions," IEEE transactions on Computer-Aided Design, vol. 8, no. 2, Feb. 1989, pp.171-180.

[7] A.E. Casavant, et al., "A Synthesis Environment for Designing DSP Systems," IEEE Design & Test of Computers, April 1989, pp.35-44.

[8] A. E. Casavant, D. D. Gajski, and D. J. Kuck, "Automatic Design With Dependence Graphs," 17th Design Automation Conference, 1980 pp.506-515.

[9] D. Gajski, N. Dutt, A. Wu, and S. Lin, "HIGH-LEVEL SYNTHESIS Introduction to Chip and System Design," Kluwer Academic Publishers, 1992.

[10] Cheng-Tsung Hwang, Yu-Chin Hsu, and Youn-Long Lin, "Scheduling for Functional Pipelining and Loop Winding," Proc.28th Design Automation Conference, 1991, pp.764-769.

[11] Ki Soo Hwang, Albert E. Casavant, Ching-Tang chang, and Manuel A. d'Abreu, "Scheduling and Hardware Sharing in Pipelined Data Paths," Proc. IEEE Int. Conf. Computer-Aided Design, Nov. 1989, pp.24-27.

[12] Ki Soo Hwang, et al., "Constrained Conditional resource Sharing in Pipeline Synthesis," Proceedings of International Conference on Computer Aided Design, 1988, pp.52-55.

[13] Van E. Kelly, M. C. McFarland, Alice C. Parker, and R. Camposano, "The High-Level Synthesis of Digital Systems," Proc. IEEE, vol. 78, no. 2, Feb. 1990, pp. 301-317.

[14] D. J. Kuck, R. Khun, D. Pauda, B. Leasure, and M. Wolfe, "Dependency Graphs and Compiler Optmizations," Proc. of 8th ACM Symp. Principles Programming Languages, 1981.

[15] Jiahn-Hung Lee, Yu-Chin Hsu, and Youn-Long Lin, "A New Integer Linear Programming Formulation For The

Scheduling Problem In data Path Synthesis", ICCAD, November 1989, pp. 20-23.

[16] Tsing-Fa Lee, Allen C-H. Wu, D. D. Gajski, and Youn-Long Lin, "An Effective Methodology for Functional Pipelining," ICCAD, 1992, pp.230-233.

[17] M. C. McFarland, Alice C. Parker, and R. Camposano, "Tutorial on High-Level Synthesis," Proc. 25th Design Automation Conference, 1988, pp. 330-336.

[18] A. Nicolau and R. Potasman, "Realistic Scheduling: Compaction for Pipelined Architectures," Proc. 23rd Annual Workshop and Symposium Micro 23 Mircoprogramming and Microarchitecture, Nov. 1990, pp. 67-79.

[19] A. Nicolau, "Loop Quantization: Unwinding for Fine-Grain Parallelism Exploitation," Technical Report TR85-709, Department of Computer Science, Cornell University, 1985.

[20] Nohbyung Park and Alice C. Parker, "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications," IEEE Transactions on Computer-Aided Design, vol. 7, no. 3, March 1988, pp. 356-370.

[21] P. G. Paulin and J. P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," IEEE Transactions on Computer-Aided Design, vol. 8, no. 6, June 1989, pp. 661-679.

[22] P. G. Paulin and J. P. Knight, "Force-Directed Scheduling in Automatic Data Path Synthesis," 24th Design Automation Conference, 1987, pp. 195-202.

[23] P. G. Paulin, J. P. Knight, and Girczyc, "HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis," 23rd Design Automation Conference, 1986, pp. 263-270.

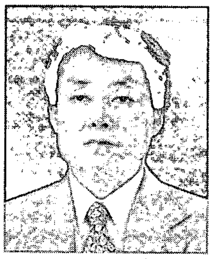
[24] Roni Potasman, et al., "Percolation Based Synthesis," Proc. 27th Design Automation Conference, 1990, pp. 444-449.

[25] Chia-Jeng Tseng, et al., "Bridge: A Versatile Behavioral Synthesis System," Proc. 25th Design Automation Conference, 1988, pp. 415-420.

[26] Kazutoshi Wakabayashi and Takeshi Yoshimura, "A Resource Sharing and Control Synthesis Method for Conditional Branches," Proc. ICCAD, Nov. 1989, pp. 62-65.

[27] Benchmarks for the Fourth International Workshop on High-Level Synthesis, 1989.

[28] Benchmarks on High-Level Synthesis, 1992.



Jongsoo Kim received the B.E. degree from Yonsei University, Seoul, Korea in 1976, the M.S. degree from the University of Arizona, Tucson, in 1989, and the Ph. D. degree from the University of Alabama in Huntsville, Huntsville, in 1994, all in electrical and computer engineering. Currently, he is an Associate Professor of the School of Electrical Engineering at the University of Ulsan.