A Procedure for Obtaining a Behavioral Description for the Control Logic of a Non-Linear Pipeline

Hashem Hashemi Najaf-Abadi

Department of Computer Engineering
Sharif University of Technology
Tehran, Iran, P.O. Box: 11365-9363
Tel: +21-737-6547
e-mail: h_hashemi@ce.sharif.edu

Abstract-- Much attention has been directed to different aspects of the design of pipelines [1,2,3,4]. Design of the control logic of non-linear pipelines has however, been considered as a subsidiary issue in that an RTL description for such logic can easily be obtained from a behavioral description, with the use of widely available synthesis tools. But, as the complexity of a non-linear pipeline increases, so does the complexity of the control logic. The complexity may be to an extent that obtaining even a behavioral description for the control logic is rendered difficult.
This paper focuses on further automating the development of systems consisting of non-linear or multi-function pipelines by proposing an algorithmic technique for obtaining a behavioral description for the control logic of such pipelines. A simplified C++ implementation that produces a VHDL description of the control logic is then presented to clarify the algorithm.
Experimental results that reveal connections between the nature of pipeline functions and the complexity of the control logic, obtained by utilization of the algorithm, are also presented. Consideration of these results can reduce the design space exploration of such pipelines to a more practically feasible subspace.

I. Introduction

A pipeline can be visualized as a collection of processing segments through which binary information flows. Each segment performs partial processing dictated by the way the task is partitioned [5]. Each segment consists of an input latch followed by a combinational circuit. In linear pipelines the output of the combinational circuit in a given segment is directly applied to the input latch of another segment. Such pipelines need no control logic and all that is needed for the pipeline to function is a clock pulse to trigger its latches. In non-linear pipelines, on the other hand, the output of the combinational circuit in a given segment may need to have a datapath to the input latch of a number of different segments (bringing about what is known as a fork in the datapath) and the input to a segment may need to be selected from a number of different possible inputs (bringing about what is known as a join). Joins are implemented by switching logic such as multiplexers and it is at this point that control logic is needed. It is up to the control logic to activate the selection signals of the switching logic such that data is guided through the segments of the pipeline appropriately.
When a number of functions with equivalent sections need to be implemented within one system, their equivalent sections can be shared, resulting in a multifunction pipeline. The interconnection between segments of a multifunction pipeline also consists of forks and joins. Thus these pipelines also need control logic to function properly.
In what follows preliminary definitions are presented first. An abstract model of the control logic is described in section three. The algorithmic procedure for obtaining a behavioral description of the control logic is presented in sections four through six. Based on this algorithm, a demonstrative program is presented in section seven that, given the reservation table(s) of the function(s), produces a behavioral description of the control logic in VHDL language. Finally, in chapter eight, connections between the complexity of the control logic and the function(s) of the pipeline are outlined.

II. Preliminaries

Reservation tables: The combinational circuits within each segment are the fundamental building blocks of the pipeline, the definition of which specifies a unique pipeline when the pipeline is linear. For the definition of a non-linear pipeline to be complete, the order of pipeline segment utilization must also be specified. Usually, reservation tables are used to specify the successive segments used by a function. The rows of the reservation table denote the segments of the pipeline and the columns are representative of the latency (clock cycles following initial insertion of data) [6].

Design space exploration: High-level synthesis (HLS) is the process of translating a behavioral description to a register transfer level (RTL) structural description [7, 8, 9]. In such a process, the design space can be explored in three dimensions, namely area, latency and clock period [10].
The larger the design space, the more time consuming it is to obtain an optimal implementation of the system. Hence, it is justified to be in search of approaches to constrain the design space to more practically feasible implementations. Guidelines to such constrains, in the design of non-linear pipelines, are obtained through experimental results when the relationship between the complexity of the control logic and the order of segment utilization is studied.

The control logic: It is the responsibility of the control logic to activate the selection signals of the switching logic at the joins (multiplexers) in such a way that will cause data to pass through the segments determined by a specific function.
The number of data inputs to a join is equal to the number of segments that need their output to propagate through the combinational logic of the segment following the join, by at least one function. The total number of output signals that the pipeline control logic must possess is equal to the total of the number of selection inputs to all the joins. For the control logic to be able to delay insertions to a permissible latency, one more output signal may be required as a strobe to external logic to signify when data has been accepted by the control logic and inserted into the pipeline.
Input signals to the control logic signify when a request for the insertion of data is pending and what function must be

applied on that data. Thus, the number of possible input-signal combinations to the control logic must be considered equal to the number of different functions of the pipeline plus one, the extra combination signifying when no data is being inserted. For instance the control logic of a non-linear pipeline with only one function needs only one input signal to notify when data is and is not being inserted.

III. An Abastract Model of The Control Logic

The control logic can be considered as being a number of sequence counters, their output signals combined to form the overall output. Each counter, when triggered, produces a special sequence that sets the join (multiplexer) selection signals to values that will guide data through the segments of the pipeline in the order determined by a specific function. When data is inserted into the pipeline, one of the counters corresponding to the function to be applied on the data must be triggered. When the sequence is finished, the counter remains inactive until the next appropriate insertion activates it again.

Since data inserted into the pipeline will not need to enter all the segments of the pipeline in each clock cycle, the only output signals that need to have a specific value during a specific clock-cycle are those that correspond to the selection signals of the join preceding the segment that data must enter during that clock cycle and the other output signals are of no consequence. Therefore, combining the output signals of a number of such counters with a Boolean Or/And operator while setting the inconsequential output signals to Boolean Zero/One will produce the necessary control signals capable of guiding data through the pipeline in an overlapped manner (generation of the counting sequence for a number of insertions can concurrently be in progress).

Such control logic, though easy to design, will not be able to detect conflicts (when two instances of a function need to allocate a segment at the same time) independently. Our goal is to design a single control unit, capable of delaying insertions until a permissible latency while consuming the minimum logic circuitry.

IV. Designing the Control Logic

It is based upon the abstract model of the control logic that a procedure for obtaining a behavioral description for such logic can be devised.

The sequence of output signal combinations that the control logic must produce in consecutive cycles, for each of the functions to be implemented, are primarily determined. These sequences can be referred to as the *generator-sequences* of the functions. Each component of the *generator-sequence* corresponds to a specific latency and consists of a number of bits. Each bit corresponding to one of the control logic output signals. Some of these bits have determinate values and the values of the others are of no consequence to the implementation of the function during the corresponding latency.

Construction of the state diagram (as a graphical form of description) of the control logic is commenced with a single state that has a no-insertion loop on it (no-insertion is signified by one of the possible input combinations). All the output signals in this state have inconsequential values. When the pipeline is in this state and data is inserted to the pipeline with a function X specified to be applied on it (signified by one of the possible input combinations), the generator-sequence of function X must be produced at the control logic's output signals during the clock cycles following the insertion. Thus, there must be a transition from the initial state (for the input combination that signifies function X) to a state that has an output value equal to the first component of the generator-sequence of function X. That state must have a no-insertion transition to another state that has an output value equal to the second component of the generator-sequence of X and so on. The final state, with an output value equal to the final component in the generator-sequence of X must have a no-insertion transition back to the initial state of the state diagram. In this way, the control logic will begin producing the necessary output signals to guide data through the pipeline segments as specified by a function only if the insertion occurs when the control logic is in the initial state.

In order for the control logic to be able to guide data through the pipeline for insertions that occur before the generator-sequence of a previous insertion has finished, generator-sequences must be combined. From hear on, the combining operator in the abstract model is considered to be the Or operator and all inconsequential values are considered as being set to Zero.

The insertion of new data to a pipeline is permissible only when the generator-sequence of the function to be applied on the data and the remainder of the generator-sequence of a function to be applied on previously inserted data do not have different definite values for an output signal at the same clock cycle.

Two definitions are made to simplify descriptions. The "remaining-sequence" of a state in the state diagram of the control logic is defined as the sequence of output combinations that is produced with only no-insertion transitions, from that state. Two sequences are said to be "non-colliding" if none of the same indexed components of the two sequences have different determinate values for the same bit position.

For every state in the state diagram, if the *remaining-sequence* of that state and the *generator-sequence* of function X are non-colliding sequences, there must be a transition for function X from that state to a new state from which no-insertion transitions will produce the sequence obtained by combining the two sequences (such as in the abstract model). In this manner the guidance of data through the pipeline, for a number of insertions, can be overlapped.

There remains something missing in the approach explained above, not allowing a backtracking algorithm to be devised from it. The condition for backtracking is incomplete and states are continuously added to the state diagram. What hasn't been considered so far is the equivalence between states.

V. State Minimization

From our knowledge of the theory of finite state machines, we know that two states are equivalent if their outputs are the same and they change to the same or equivalent next states for all input combinations [11].

In the procedure explained above, all transitions for different functions (for different input combinations) from a state are determined on the basis of that state's remaining-sequence and the generator-sequences of the functions. Thus, two states

with the same remaining-sequence will have transitions for the same function to states that have the same remaining-sequence and the same output value. In other words, two states with the same remaining-sequence will produce the same sequence of output combinations for the same sequence of input combinations and this will hold even as the length of the input sequences tend towards infinity. Hence, a pragmatic conclusion can be made that states with the same remaining-sequence are equivalent. Consequently, one of the equivalent states can be removed and transitions to that state replaced with transitions to its equivalent state. This leads to a minimum state diagram with a finite set of unique states. Consideration of the equivalence between states in the process of obtaining the state diagram leads us to a backtracking algorithm that produces the minimum state diagram.

VI. The Algorithmic Procedure

To simplify the design procedure, a list containing defined states and their corresponding *remaining-sequences* must be maintained during the design process. This list is referred to as the *created-sequence-list (abbreviated as CSL)*.

For any state in the state diagram such as S1, if the remaining-sequence of that state and the generator-sequence of a function such as X are non-colliding sequences, a transition from state S1, for function X, to a state such as S2 must be added to the state diagram. The output value of S2 and its remaining-sequence must be equal to the first component and the rest of the components of the combination of the two sequences, respectively. If no such state can be found, it must be defined and added to the CSL.

For no-insertion transitions to be considered, a function that has a remaining-sequence consisting entirely of inconsequential components, called a null sequence, must be added to the set of pipeline functions. Transitions corresponding to this function should be labeled as no-insertion transitions and the function will be referred to as the *null* function. The procedure for obtaining the state diagram of the control logic is depicted in the flowchart shown in the figure 1.

Construction of the state diagram of the control logic is commenced with a single state that has a null remaining-sequence, and the CSL is commenced with a single component consisting of a null sequence, the starting state of which is the only state in the diagram. The combination of the remaining-sequence of this state and the generator-sequence of the null function is obviously a null sequence. But such a sequence already exists in the CSL. As a result, a transition from this state to the initial state of the null sequence must be added to the state diagram. This leaves us with a single state that has a no-insertion loop on it (as explained before). The combination of the remaining-sequence of this state and the generator-sequence of a function of the pipeline will be equal to the generator-sequence of the function. Thus, there will be a transition from this state, for each function of the pipeline, to states that have output values equal to the first components of the generator-sequences of the functions of the pipeline. The remaining-sequences of these states are equal to the generator-sequence of their corresponding function, with the first component removed. Therefore, the no-insertion transitions from these states will enter states that have an output value equal to the second component of the corresponding function and so on. Finally, the output strobe

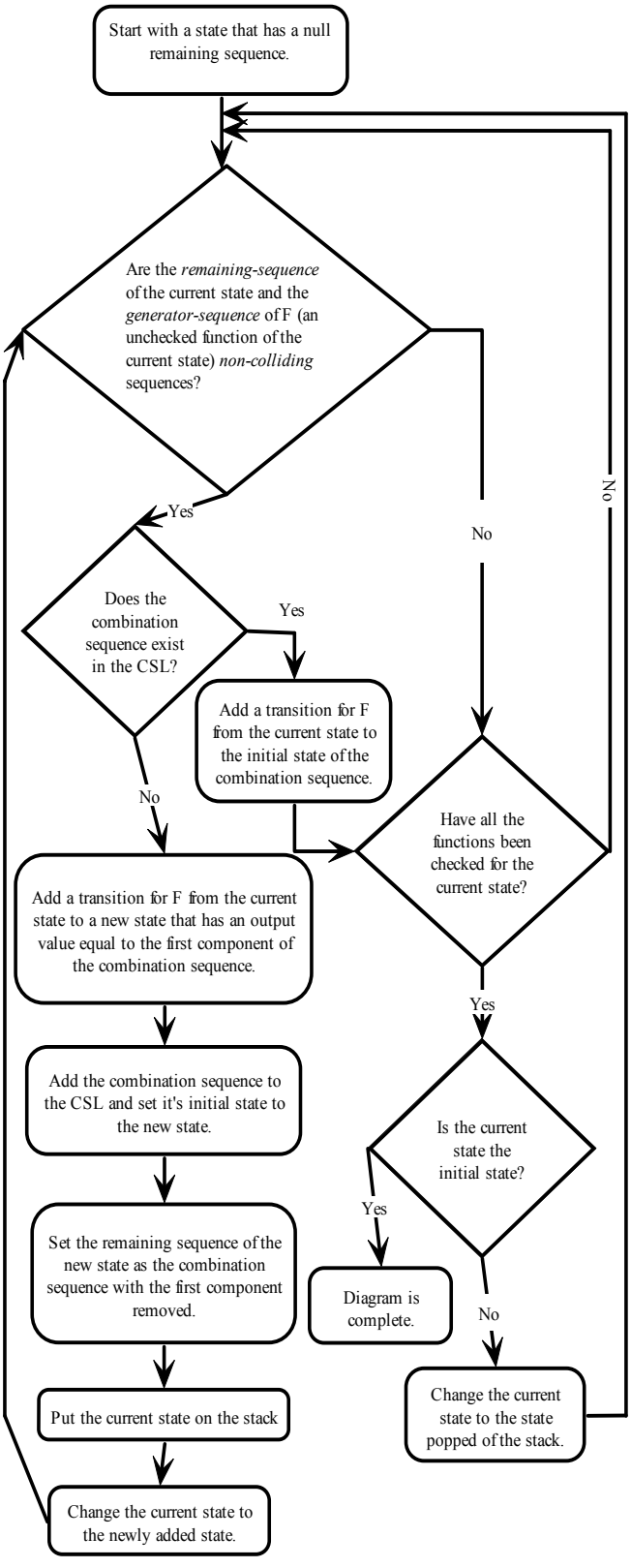
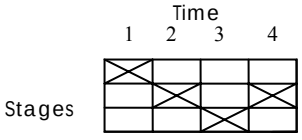


Figure1) Flowchart of algorithm for producing the description of the control logic of a non-linear pipeline.

is set to one by all transitions except no-insertion transitions to signify when data is being accepted by the pipeline.

Example: As a simple example, the design of the control logic of a representative non-linear pipeline that utilizes only one of its segments more than once is presented. In such a case, the only segment in the pipeline that may have input

from more than one segment is the reutilized segment. As a result, there is only one join in all the datapaths of the pipeline and since only two datapaths enter that join, the control logic of the pipeline need only have one output signal. Consider a pipeline with such a reservation table:



This pipeline utilizes three different pipeline segments before reutilizing the second segment. Thus, the control logic must select the data entering the second segment through the join. The status of the join is only of importance in the second and fourth latencies (when data must past through the second segment). Therefore, the generator-sequence of the only function of this pipeline can be determined as (x, 0, x, 1), in which the leftmost component corresponds to the first latency following insertion and x's correspond to "don't care" values. Figure 2 displays the state diagram of the control logic of this pipeline. State-1, the initial state of the state diagram, is a state that has a null remaining sequence. Since the remaining sequence of this state and the generator sequence of the pipeline function are non-colliding sequences, there is a transition from this state to state-2, a state whose remaining-sequence is the generator-sequence with the first component removed. The remaining-sequence of state-2 and the generator-sequence are also non-colliding sequences. Thus, there is a transition from state-2 to state-5, a state whose remaining sequence is the combination of the remaining-sequence of state-2 and the generator-sequence, with the first component removed. There is also a no-insertion transition from state-2 to state-3, a state whose remaining sequence is the remaining sequence of state-2 with the first component removed. In the same manner, states-3, 4, 5 and 6 have no-insertion transitions to states 4, 1, 6 and 4 respectively.

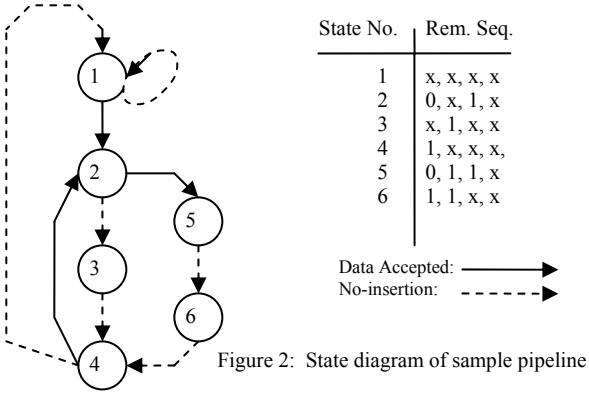


Figure 2: State diagram of sample pipeline

VII. IMPLEMENTATION

The first thing that must be determined is which output bits are to be connected to which selection bits of which multiplexer. This is an arbitrary selection, but must be determined before control logic design can begin. The *generator-sequences* of the different functions must then be determined. The following code produces the *generator-sequences* for a pipeline with n_func functions, a maximum function length of max_lngth, n_seg segments and the reservation tables stored in res_tbl (segment outputs are, for lucidity, considered to be connected to multiplexer inputs in increasing order of their index):

```
0  Int res_tbl  [n_funcs][n_segs][max_lngth] = { ... };
1  for (i = 0; i < n_funcs; i++){
2   for (k = 0; k < max_lngth; k++){
3    for (j = 0; j < n_segs; j++)
4     if (res_tbl[i][j][k] == 1){
5     int select=0;
6     int prev_seg=0;
7     for (l = 0; l < n_segs; l++)
8      if (res_tbl[i][l][k-1] == 1)
9       prev_seg = l;
10    for (n=0; n < prev_seg ; n++)
11     if (inputs_from [j][n] == 1) select++;
12    int index = seg_sbit[j];
13    gen_seq [i][k][index +  ... ] = select ;
    } } } }
```

If function i utilizes segment j of the pipeline at latency k, the segment from which data enters segment j at that latency is determined in line 7. The selection combination that must be applied to the multiplexer of segment j is determined on line 10 by counting the number of lower indexed segments that also have an output path to segment j (all segments that have an output path to segment j have been marked in a lookup table named inputs_from). On line 12, the index of the output bit corresponding to the lowest order selection bit of the multiplexer of segment j is determined from another lookup-table named seg_sbit and stored in *index*. On line 13 the binary bits representing *select* are stored in consecutive locations of the *generator-sequence* (the gen_seq array) that correspond to segment j at latency k (at the beginning, gen_seq is considered to be filled with Zeros).

Once complete, the generator-sequences can be used to obtain the description of the control logic. This can be implemented in C++ [12] by defining a class with a constructor such as:

```
1  node::node(char rem_seq[max_lngth][n_outputs], int node_num) {
2  for (func_tested =0; func_tested< n_funcs +1; func_tested++) {
     coliding = 0;
3    for (colmn_cntr= 0; colmn_cntr < n_outputs; colmn_cntr++)
4     for (row_cntr = 0; row_cntr < max_lngth -1; row_cntr++)
5      coliding |=
         d_bits [row_cntr+1][colmn_cntr] &&
                 func_dtrmint[func_tested][row_cntr][colmn_cntr];
6  if (coliding == 0) {
7    for (colmn_cntr = 0; colmn_cntr < n_outputs; colmn_cntr++)
       for (row_cntr = 0; row_cntr < max_lngth-1; row_cntr++)
        local_rem [row_cntr][colmn_cntr]
          = rem_seq [row_cntr+1][colmn_cntr]
                | gen_seq [func_tested][row_cntr][colmn_cntr];
8    for (colmn_cntr = 0; colmn_cntr < n_outputs;  colmn_cntr++)
              local_rem[max_lngth-1][colmn_cntr]
                =  rem_seq  [func_tested][max_lngth -1][colmn_cntr];
9     if (! crtd_nodes::first->InList(local_rem,next_node )){
10     created_nodes::Add2List(local_rem_seq,next_node );
11     node* adjecent = new node (local_rem_seq, next_node);
       delete adjecent;  }
12     Write2File
        ("elsif state = %d and insert_sig = %d then state := %d;
           strobe <= '1'; output <= %d",
            node_num, func_tested, next_node,
              local_rem_seq[0][...]);
13     if (func_tested == 0) no_insrtn_next = next_node; }
     else
14     Write2File
        ("elsif state = %d and insert_sig = %d then state := %d;
           strobe <= '1'; output <= %d",
            node_num, func_tested, no_insrtn_next,
              rem_seq[1][...]);
   } }
```

And the body of the class can be defined, with an array that

contains its remaining sequence, such as below:

```
class node {
public:
        node (char rem_seq[max_lngth][n_outputs],
              int node_num);
private:
        char InList(int, int, char&);
        void Add2List(int, int, char&);
        int func_tested;
        char local_rem_seq [max_lngth][n_outputs]; };
```

The constructor of each node receives a pointer to an array that is the *remaining-sequence* of that node and an integer that determines that nodes ID-number. On lines 3 through 5 the program determines whether or not the current state and the *generator-sequence* of a function are colliding sequences (the arrays d_bits and func_dtrmint assert which locations of these arrays contain determinate values). If they are not, the state diagram must have a transition from the current state to another state. The *remaining-sequence* of that state is calculated in lines 6-7 and stored in local_rem. A linked-list containing information about previously defined nodes is needed at this point. In this implementation a class named crtd_nodes has been used to implement the linked-list [13]. This class has a public static member function named InList that takes a pointer to an array as its first parameter, it explicitly returns one when it finds a state with a *remaining-sequence* identical to the array passed to it and implicitly returns the ID-number of the state it may find, through its second operand. On line 8, crtd_nodes is searched for a state with a *remaining-sequence* identical to local_rem. If no such state is found, on line 10, a state with local_rem as its *remaining-sequence* is added to crtd_nodes through one of its methods named Add2List. This method implicitly returns the ID-number of the newly added state through its second operand. On line 11, the constructor of the node corresponding to the new state begins execution. Then, in line 12, an ELSIF statement [14], in VHDL language, is written to an output file. This statement is part of the behavioral description of the control logic and determines the next state the control logic must go to and what the output must be when in a specific state and a specific function is pending. The function indexed zero is considered to be the NULL function. When the remaining-sequence of the current state and the generator-sequence of a function are colliding sequences the insertion must be delayed. This is implemented by deactivating the strobe signal. The next state that the control logic must go to is the same state that the no-insertion transition goes to. It is for this reason that, in line 13, the destination state of the no-insertion transition is saved in a local variable. Then in line 14, when an insertion must be delayed, the next state is obtained from that local variable.

Now, if an object of type *node* is defined with an array containing the generator-sequences of the different functions (the NULL function in index zero of the array) passed to it, a series of ELSIF statements will be produced and written to a file. These statements provide a complete behavioral description for the control logic when appended within a VHDL *process* [14] that is triggered by the rising or falling edge of a clock pulse.

VIII. Experimental Observations

Implementation of this procedure reveals connections worthy of note between the complexity of the control logic and the latency of initial utilization and reutilization of segments by pipeline functions. In what follows the number of states of the control logic and the ratio of the number of non-colliding transitions to the total number of transitions are considered to be representative of the complexity and through-put of the pipeline, respectively.

A pipeline that implements a single function that utilizes all segments exactly once is a linear pipeline. Thus, a pipeline capable of implementing only a single function that utilizes only one segment twice in an execution is considered first. Table I shows the number of states of the control logic of such a pipeline for various values of *the initial utilization latency* (IUL) and *latency difference between initial utilization and reutilization* (LDIR) of the reused segment. Note that the figures in this table are independent of the number of segments of the pipeline. It is observed that the number of states of the control logic of such a pipeline grows exponentially with IUL and LDIR. In fact, the control logic of such a pipeline with IUL and LDIR equal to one (a pipeline that utilizes its first segment twice before utilizing other segments) has *three* different states and this figure roughly doubles for each increase in LDIR and is approximately multiplied by 1.5 for each increase in IUL. However, if the only reused segment of a pipeline is utilized more than twice, the latency of utilizations other than initial and final utilizations, do not appear to have very much effect on the number of states of the control logic. Evidence of this fact can be observed in Table II. This table displays the number of states of the control logic for various values of IUL and LDIR of the second utilization (the LDIR of the last utilization being considered fixed). It is observed that for a fixed IUL, different values for the LDIR of the second utilization do not cause very much change in the number of states of the control logic, (except when the LDIR of the second utilization is approximately half the LDIR of the final utilization of the segment).

The next type of pipeline considered is a pipeline with a single function that performs multiple utilizations of segments, in such a way that the IUL of one segment is less than the IUL of all other reused segments and the latency of final utilization of that segment is greater than all the other reused segments, i.e. the initial utilization of the last segment that is reutilized by the pipeline function occurs before the initial utilization of all other reutilized segments. It is observed that the number of states of the control logic of such a pipeline is absolutely independent of the IUL of the reused segments. In other words, the latency of initial utilization of the pipeline segments can be varied without any change in the complexity of the control logic. More interesting is the fact that the same holds for the through-put of the pipeline in such a scenario. Table III displays the complexity of the control logic and the through-put of a pipeline that utilizes a segment once, then utilizes another segment two times consecutively and finally utilizes the first segment again. The row numbers correspond to difference between the IUL of the two segments and the column numbers correspond to the LDIR of the segment that has a greater IUL.

Bearing in mind that the order of utilization of segments in nonlinear pipelines is usually somewhat negotiable, factors such as the ones mentioned above are of importance when considered in the early stages of designing such systems (designing the reservation tables). For instance it is evident

from Table III that, in such a scenario, initial utilization of the second segment at an earlier latency after initial utilization will result in less complex control logic.

IX. Conclusions

The presented procedure can be implemented in software to produce a behavioral description in an HDL. The control logic will however insert fetched data into the pipeline at the first permissible latency. This latency may not produce the minimum average latency (MAL) and extra logic will be needed to delay insertions until a latency that will produce the MAL. Some experimental results are presented, showing that connections do exist between the complexity of the control logic and the order of segment utilization and that consideration of these connections in the early stages of design can result in less complex control logic and a reduction in the design space exploration. But further experiments need to be conducted in order to find more appreciable connections between the definition and the complexity and through-put of non-linear and especially multi-function pipelines.

References

[1] Peter Grun, Ashok Halambi, Nikil Dutt, Alex Nicolau "An Algorithm for Automatic Generation of Reservation Tables from Architectural Descriptions", International Symposium on system synthesis, 1999.

[2] N. Park and A. C. Parker, "Sehwa: a software package for synthesis of pipelines from behavioral specifications," IEEE Trans. on CAD, vol. 7, pp. 356--370, March 1988.

[3] D. A. Lobo, B, M, Pangrle**, "**Generating pipelined datapaths using reduction techniques to shorten critical paths", Proceedings of the conference on European design automation 1992, pp 390 – 395.

[4] K. N. McNallm A. E. Casavantm, "Automatic operator configuration in the synthesis of pipelined architectures, Conference proceedings on 27th ACM/IEEE design automation conference, 1991, pp. 174 – 179.

[5] M. Morris Mano, "Computer system architecture", Prentice-Hall International, 1993, pp. 302.

[6] E.S.Davidson, "The design and control of pipelined function generators", Proc. 1971 Int. Conf. on Systems, Networks and Computers, Oaxtepec, Mexico, pp. 19-21

[7] R. Camposano, W. Wolf, "High level synthesis", Kluwer Academic, 1991.

[8] G. De Micheli, "Synthesis of digital circuits", McGraw-Hill, 1994.

[9] Youn-Long Lin, "Recent developments in high-level synthesis", ACM Tras. Design Automation of electronic systems, 2(1):2-21, Jan, 1997.

[10] Stephan A. Blythe and Robert A.Walker. "Towards a practical methodology for completely characterizing the optimal design space", Proceeding of the 19th International Symposium on system synthesis, pages 8-13, La Jolla, CA, Nov 1996, ACM-IEEE.

[11] P.Linz, "An Introduction to Formal Languages and Automata", D.C. Heath and company, 1990.

[12] Nicolai M.Josuttis, "The C++ Standard Library", Addison-Wesley Reading, 1999.

[13] J. Tremblay, P. G. Sorenson, "An introduction to data structures with applications", McGraw-hill, 1984, pages 254-294

[14] Z.Navabi, "VHDL, Analysis and modeling of digital systems", McGraw-Hill, Newyork, 1998

TABLE I - The number of states of the control logic of a pipeline with a single function that utilizes only one segment twice. The row numbers denote the IUL and the column numbers denote the LDIR of the reused segment. (N.C.: not calculated)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	6	12	24	48	96	192	384	768	1536	3072	6144	12288	24576
2	5	9	18	36	72	144	288	576	1152	2304	4608	9216	18432	36864
3	8	15	27	54	108	216	432	864	1728	3456	6912	13824	27648	N.C.
4	13	25	45	81	162	324	648	1296	2592	5184	10368	20736	N.C.	N.C.
5	21	40	75	135	243	486	972	1944	3888	7776	15552	N.C.	N.C.	N.C.
6	34	64	125	225	405	729	1458	2916	5832	11664	N.C.	N.C.	N.C.	N.C.
7	55	104	200	375	675	1215	2187	4374	8748	N.C.	N.C.	N.C.	N.C.	N.C.
8	89	169	320	625	1125	2025	3645	6561	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.
9	144	273	512	1000	1875	3375	6075	10935	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.
10	233	441	832	1600	3125	5625	10125	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.
11	377	714	1352	2560	5000	9375	16875	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.
12	610	1156	2197	4096	8000	15625	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.
13	987	1870	3549	6656	12800	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.
14	1597	3025	5733	10816	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.
15	2584	4895	9261	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.	N.C.

TABLE II - The number of states of the control logic of a pipeline with a single function that utilizes only one segment three times. The row numbers denote the IUL and the column numbers denote the LDIR of the second utilization of the reused segment. The LDIR of the last utilization is equal to 16.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2817	2820	2817	3087	2817	2820	2817	8748	2817	2820	2817	3087	2817	2820
2	3804	3600	3592	3969	3571	3600	3595	11664	3595	3600	3571	3969	3592	3600
3	5024	4860	4582	5103	4529	4560	4670	15552	4670	4560	4529	5103	4582	4860
4	6676	6561	6180	6561	5743	5776	6060	20736	6060	5776	5743	6561	6180	6561
5	8862	8667	8372	8748	7301	7448	7888	N.C.	7888	7448	7301	8748	8372	8667
6	11763	11449	11341	11664	9783	9604	10192	N.C.	10192	9604	9783	11664	11341	11449
7	15616	15194	15019	15552	13506	13230	13176	N.C.	13176	13230	13506	15552	15019	15194

TABLE III - The number of states and the through-put of the control logic of a pipeline with a single function that utilizes a segment before and after utilizing a second segment twice. The row numbers denote the difference between the IUL of the two segments and the column numbers denote the LDIR of the second reutilized segment. In all cases, the IUL and LDIR of the first segment are 1 and 12, respectively.

	1	2	3	4	5
1	521~0.182745	609~0.236364	704~0.238095	875~0.230769	945~0.230769
2	521~0.182745	609~0.236364	704~0.238095	875~0.230769	945~0.230769
3	521~0.182745	609~0.236364	704~0.238095	875~0.230769	945~0.230769
4	521~0.182745	609~0.236364	704~0.238095	875~0.230769	945~0.230769
5	521~0.182745	609~0.236364	704~0.238095	875~0.230769	945~0.230769