

Timing Analysis in High-Level Synthesis

Andreas Kuehlmann

Reinaldo A. Bergamaschi

IBM Research Division

Thomas J. Watson Research Center

Yorktown Heights, N.Y., U.S.A.

Abstract

This paper presents a comprehensive timing model for behavioral-level specifications and algorithms for timing analysis in high-level synthesis. It is based on a timing network which models the data flow as well as the control flow in the behavioral input specification. The delay values for the network modules are created by invoking the same logic synthesis procedure applied after behavioral synthesis. The timing network is built only once for a given behavioral description. Several parameters are used to explore different scheduling possibilities as well as different optimization modes (area, delay), without changing the network. The use of the timing model in conjunction with a path-based scheduling algorithm is presented. Results for several benchmarks attested the accuracy of this approach.

1 Introduction

Timing analysis during synthesis is necessary to drive the various optimization and transformation tasks according to performance requirements. Timing estimation during or after logic synthesis is typically performed at the gate or even at the transistor level [1], [2], [3] because, at these levels, accurate timing models are available. The same is not true for high-level synthesis applications. Firstly, at the behavioral level, there is no fixed control and data-path. Registers, functional units, multiplexers, and control are only defined after scheduling and allocation. Secondly, operations in a high-level description have no predefined implementations.

Timing models, at the high-level, have been used for various purposes. Some approaches are limited to interface synthesis. In [4], for example, a timing event graph is used to model the interface behavior and to generate a logic frame around the block to be synthesized. In [5], the modelling of external timing constraints is extended to the internal behavior, but the approach is limited to scheduling after allocation and also, it does not include the controller in the timing

model.

Other high-level synthesis systems use simplistic timing models to guide the scheduling before allocation. In [6] a simple area-time estimation approach is presented which assigns a single delay and area value to each operation. The HAL system [7] also uses single propagation delay values for the scheduling to determine whether data operations can be chained. The usage of single delay values for complex data operations may result in an overestimation of the propagation delay of chained operations. For example, the delay of two chained ripple carry adders is smaller than the sum of the longest delays since the critical path includes a complete carry chain only once. Furthermore, the influence of the controller on the timing of the design is not considered in all these approaches.

The Chippe System [8] performs timing estimations after scheduling, using simple delay values for data path operations and a PLA model for the controller. If the timing requirements are not met, new schedules are generated by placing critical operations in earlier states. In [9] a more detailed timing model which considers individual delays between different pairs of inputs and outputs is used for module selection.

This paper presents a comprehensive timing model for high-level synthesis, which is based solely on the behavioral specification, thus before scheduling and allocation. Global control and data-flow analysis are used to build a timing network which considers different schedules and their implications in terms of registers, functional units, multiplexers, interconnections, and control. The timing models for the basic high-level operations are generated based on the technology and on the logic synthesis system used at the back-end of the synthesis process. This effectively gives a timing estimator which is directly adjusted to the optimization algorithms used after high-level synthesis. Individual and/or clustered delays for each pair of inputs and outputs are considered for each operation.

2 Basic Timing Model

The basic timing model consists of a directed acyclic graph with weighted delay values on the edges. Propagation delays through this graph are computed based on the longest path, which can be done in linear time [3]. The graph represents a behavioral timing network and is derived from the high-level specification (control and data-flow graphs). It uses predefined *timing modules* for high-level operations and value assignments. Each timing module contains several parameters which control the edge weights in the network. These parameters allow the timing network to be reconfigurable, which is needed in order to encompass multiple different schedules and module implementations (different optimization criteria, power levels).

2.1 Library of Timing Modules

The library of predefined timing modules includes models for all high-level data and control operations, for various bit-widths and optimization levels (e.g., slow, fast). The set of timing modules includes combinatorial modules (e.g. logic and arithmetic operators) and sequential modules (for multicycle operations, value assignments, and registers). Expansion routines are provided for generating on-the-fly modules with bit-widths not present in the library.

In order to characterize accurately the delay for each timing module, one must know its final implementation. This problem is simplified if a predefined set of modules and/or module generators [9] is used, due to the limited number of possible implementations and due to the lack of optimizations across chained modules.

In our methodology, logic synthesis (LSS [10]) is used to optimize and map the register-transfer description (produced by high-level synthesis) into a technology-dependent gate-level netlist. The spectrum of possible implementations for each module is much broader, varying gradually according to optimization parameters. In order to cover this variety of implementations the delays values are characterized by a set of possible implementations generated by different logic synthesis scenarios. Each module is assigned two (or more) sets of delay values which represent the interval from slowest to fastest implementations. This technique guarantees a fine tuning between the high-level timing model and the optimization algorithms applied during logic synthesis.

Figure 1a shows the general structure of combinatorial timing modules. Basically, the delays between all pairs of inputs and outputs are specified (similarly to [9]). For sequential timing modules an additional

input pin Σ and an additional output pin Δ are generated for the source and sink of timing events respectively (figure 1b) which are used for modelling the clock. Delay edges can be omitted if no data dependency exists for the corresponding input/output pair ($w_M(e_M) = -\infty$). Clustering of input and/or output pins is applied in order to simplify the model and decrease memory and cpu usage. For instance, equivalent inputs of commutative operations often have the same timing behavior, and hence may be clustered without compromising accuracy. For non-commutative module clustering represents a trade-off between accuracy and memory/cpu usage. Figure 1c gives an example for different clusterings of a 4-bit adder module.

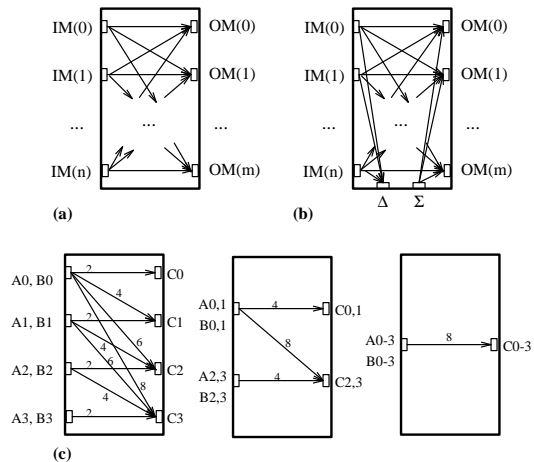


Figure 1: Structure of the timing modules: (a) combinatorial module, (b) sequential module, (c) different clusterings of a 4-bit adder timing module.

3 Behavioral Timing Network

The behavioral timing network presented here models the *control-flow* and the *data-flow* present in the high-level description. Timing estimations based on this network are used to direct the scheduling algorithm, such that the final hardware implementation meets a user-defined cycle time. The construction of the timing network is based on *state timing modules STM* and *transition timing modules TTM* for modelling the control-flow, and on *value timing modules VTM*, *operation timing modules OTM*, and *multiplexer timing modules MTM* for modelling the data-flow.

The *OTMs* model the timing of combinatorial (e.g., *AND*, *OR*, *ADD*) or sequential operations (e.g., procedure calls, multicycle operations). The *MTMs* are used for multiplexers at the inputs of operations. The

VTMs are specific implementations of sequential timing modules for value assignments from operations to variables in the data-flow domain. Depending on the final schedule of operations a value assignment may be implemented either as a register (if the value is alive over state boundaries) or as a wire (if the value is between operations which are chained in the same state). Switchable edges between the inputs I_M and the outputs O_M of the *VTMs* are used to model this without need to change the network. By setting the delay to 0, the *VTM* becomes a *wire* with no internal delay (edge switched “ON”), therefore realizing the chaining of the preceding and succeeding operations. If this delay is $-\infty$ (edge switched “OFF”), the *VTM* becomes a *register* and the delay from the I_M to Δ corresponds to the set-up time, while the delay from Σ to the O_M corresponds to clock-to-output propagation delay.

The *VTM* contains an additional set of output pins O'_M , which is necessary to avoid loops in the timing network due to control loops in the behavioral description. Only the set of edges from Σ to the outputs is applied to O'_M , whereas the edges from the inputs are omitted. Hence, the output pins O'_M are never connected to the input pins. The O'_M outputs are used to feed operations which are always scheduled in a different cycle step than the preceding operation. For example, in figure 2, the increment operation on variable x is connected to the input and output of *VTM*(x). The output O'_M is used to feed the increment operation on the next iteration of the loop while O_M is used for the successor operation $op(x)$ which may be chained with the incrementer.

In the control-flow domain, the *TTMs* model the computation of conditions, while the *STMs* model the state in which one or more operations are executed. The *STM* is similar to the *VTM* and its edges can be switched “ON” or “OFF” for modelling different schedules.

3.1 Timing Network for Control Flow

The control-flow graph is defined as $CFG = (N, P)$, where N is the set of operations and P a set of edges representing the control precedence relations [11]. Given the *CFG* of a high-level description the

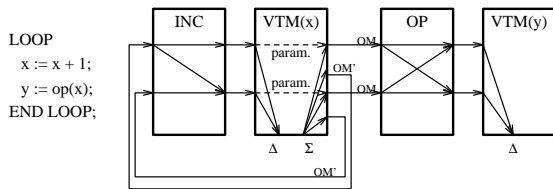


Figure 2: Usage of the value timing modules *VTM*.

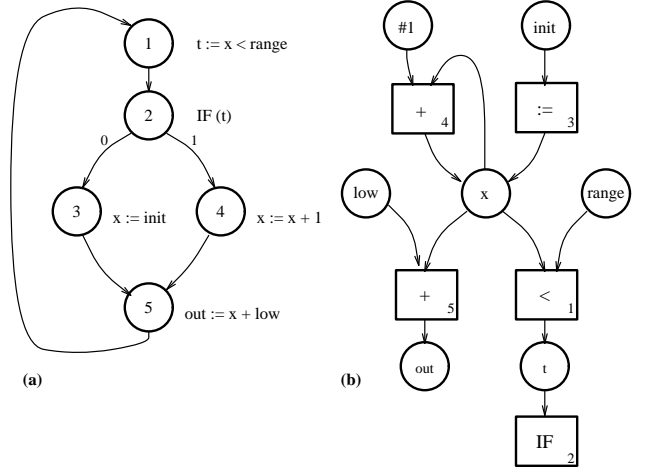


Figure 3: Example for a timing network: (a) control-flow graph, (b) data-flow graph.

timing network for the control flow is constructed as follows:

- for each operation $n \in N$ generate a state timing module $STM(n)$ and a transition timing module $TTM(n)$,
- interconnect $TTM(n)$ to the corresponding $STM(n)$,
- interconnect $STM(n)$ to $TTM(n')$, for all operations n' such that $(n, n') \in P$.

The *STMs* model the state register of the final controller. The control timing network can be regarded as a “one-hot encoding” where each state is represented by a single state register bit. A state beginning with operation n in the *CFG* is modelled by switching the corresponding $STM(n)$ “OFF” (which blocks the delay flow from the previous *TTM*). If several succeeding operations are scheduled in the same state the corresponding *STMs* are switched “ON”. This models their concurrent execution including delays for the calculation of conditions resulting from FORK/JOIN constructs.

The *TTM* models the delay for the computation of the next state. In a linear sequence of operations, it degenerates to a “wire” with zero delay. JOIN control operations are implemented by an “OR” element modelling the merging of the control flow. For all successors of FORK operations the *TTM* models “AND” elements, which combines the flow of the predecessor *STM* with the output of the conditional data operation at the FORK node.

Figure 4 shows the timing network for the *CFG* example of figure 3a. $TTM(1)$ and $TTM(2)$ are wires

with 0 delay and are not shown. Note that the O'_M output from $STM(5)$ is used to feed $STM(1)$ because it goes beyond the loop edge $(5,1)$ which implicitly generates a state boundary.

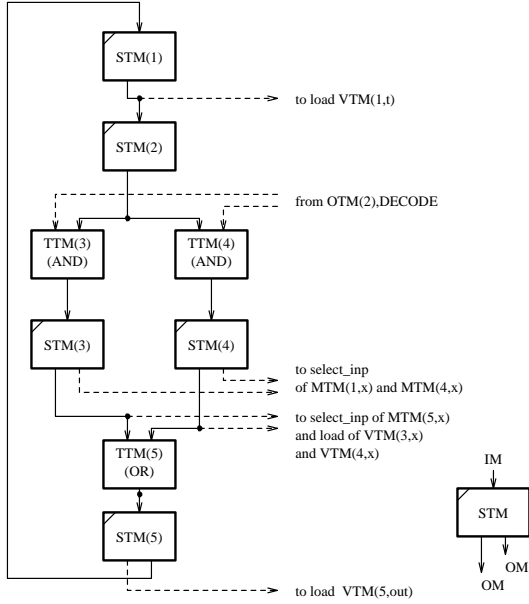


Figure 4: Timing network for the control-flow graph of figure 3a.

3.2 Timing Network for Data Flow

The construction of the timing network for the data-flow is based on the results of data-flow analysis. Global data-flow analysis is performed in the beginning of high-level synthesis in order to determine the *lifetimes* of all assignments of values to variables and primary outputs [11]. It considers conditional operations, loops, and is not restricted to basic blocks.

The data-flow graph is defined as $DFG = (N \cup V, D)$ [11], where N and V are the sets of operations and variables respectively, and D is the set of edges representing data dependencies. Furthermore, the following notations are assumed: $A = \{(n, v) \in D \mid n \in N \wedge v \in V\}$ is the set of values assigned by all operations, and $B_n \subseteq A$ is the set of values assigned by operation $n \subseteq N$. $L(a) \subseteq N$ is the lifetime (set of nodes where a is alive) of a value $a \subseteq A$.

Let $V_n \in V; n \in N$ denote the set of input variables of an operation. Thus, the set $A_n(v)$ of values alive at input $v \in V_n$ of operation n is determined by: $A_n(v) = \{a = (n', v) \in A \mid n' \in N \wedge n \in L(a) \wedge v \in V_n\}$.

Based on these definitions the construction of the timing network is done as follows:

- generate a value timing module $VTM(a)$ for each

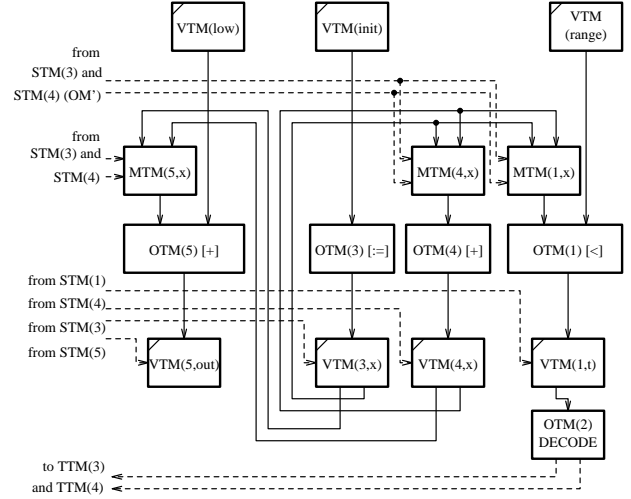


Figure 5: Timing network for the data-flow graph of figure 3b.

value $a = (n, v) \in A$,

- generate an operation timing module $OTM(n)$ for each data operation $n \in N$,
- generate a multiplexer timing module $MTM(n, v)$ with $|A_n(v)|$ inputs, for each input $v \in V_n$ of each operation $n \in N$,
- interconnect the $VTM(a)$ to all appropriate $MTM(n, v)$, where $n \in L(a) \wedge a = (n', v)$,
- interconnect the $MTM(n, v)$ to the corresponding $OTM(n)$,
- interconnect the $OTM(n)$ to the values $VTM(a)$, where $a \in B_n$.

Figure 5 shows the timing network for the data-flow graph in figure 3b. Variable x has two values $VTM(3, x)$ and $VTM(4, x)$ from operations 3 and 4 respectively. Both values are alive and used at operations 1, 4, and 5. Therefore, multiplexers are needed for these three operations to select the appropriate value of x ; the selection is controlled by the control-flow.

3.3 Interconnection between the Data and Control Timing Networks

An important part of the timing model is the interactions between the control-flow and the data-flow. Although in many cases the performance of the design is mainly determined by the timing of the data operations, critical delay paths often go through the controller. In the example in figure 3 the timing of output operation 5 is dependent on the comparison operation 1 although there is no direct data dependency. In

the described timing model, the delay path is modelled by the chain: $OTM(1) \rightarrow VTM(1, t) \rightarrow OTM(2) \rightarrow TTM(3/4) \rightarrow STM(3/4) \rightarrow MTM(5, x) \rightarrow OTM(5) \rightarrow VTM(5, out)$.

The interconnection between the data and control-flow timing networks is done as follows:

- interconnect the $STM(n)$ to the load inputs of the $VTM(a)$, where $a = (n, v) \in B_n$,
- interconnect the $STM(n)$ to the select inputs of the $MTM(n', v)$; $\forall a$, where $a = (n, v) \in B_n \wedge a \in A_{n'}$,
- interconnect the $OTM(n)$ of all conditional operations to the $TTM(n')$, where $(n, n') \in P$.

4 Timing Driven Scheduling

The presented timing model is used to direct the path-based scheduling algorithm implemented in the HIS system [11]. This algorithm schedules each execution path (sequence of operations) in the control-flow graph in the minimum number of control steps, given input constraints and a fixed ordering of operations in the CFG. Constraints are derived by the system according to user directives such as area and delay. Given a path $PA = (n_1, n_2, \dots, n_p)$, where $n_i \in N$ and $(n_i, n_{i+1}) \in P$, it exists a constraint $CO = \{n_i, n_j\}$, $n_i, \dots, n_j \in PA$, iff nodes n_i and n_j must be scheduled in different control steps [12, 13].

Timing analysis is performed on each path separately. Given a sequence of operations in a path, timing analysis starts with the first operation and computes its delay. The algorithm figures out which values would have to be stored and switches the corresponding $VTMs$ “OFF”. If the worst delay does not exceed the cycle time, the next operations in the path are added one after another. Since the whole sequence would be scheduled in the same state, the first STM in the sequence is switched “OFF” and the others are switched “ON”. If the worst delay exceeds the cycle time, then a constraint is generated between the first and last operations in the sequence. This procedure is repeated for all sequences of operations in a path. Once all timing constraints are derived, the scheduler finds a schedule which satisfies all constraints.

Delay dependencies from all other paths are blocked by switching “OFF” the $VTMs$ corresponding to values not used in the current path, and switching “OFF” the $STMs$ of operations not in the current path. This makes the delay calculation dependent only on the operations in the path being considered. False control paths are deleted [14], which helps to eliminate false delay dependencies [15].

The following procedure returns the estimated worst delay through a sequence of operations $(n_i, n_{i+1}, \dots, n_j)$ in path PA , assuming that they are scheduled in the same control cycle.

```

Estimate_Delay ( $PA, i, j$ )
Begin
  switch OFF all  $VTM(a); a \in A$ ;
  switch OFF all  $STM(n); n \in N$ ;
  For  $k = i$  To  $j - 1$  Do
    switch ON all  $VTM(a); a \in B_{n_k}$ ;
    switch ON all  $STM(n_{k+1})$ ;
  End For;
  For  $k = i$  To  $j$  Do
    Compute arrival times at  $\Delta$  of any  $VTM(a); a \in B_{n_k}$ ;
    Compute arrival times at  $\Delta$  of  $OTM(n_k)$ 
  End For;
  Compute arrival times at  $\Delta$  of any  $STM(n_s); \forall n_s : (n_j, n_s) \in P$ ;

  Return worst computed delay;
End;
```

A set of timing constraints is generated for each path by repeating timing analysis for each subsequence of operations in the path, as formalized in the following procedure.

```

Generate_Constraints ( $PA, cycle\_time$ )
Begin
  For  $i = 1$  To  $p - 1$  Do
    For  $j = i + 1$  To  $p$  Do
      If ( $Estimate\_Delay(PA, i, j) > cycle\_time$ ) Then
        Generate Constraint  $(n_i, n_j)$ ;
      End For;
    End For;
  End For;
End;
```

5 Results

Two types of experiments were done in order to evaluate the presented algorithms: experiments for validating the timing library, and experiments for validating the high-level modelling.

The timing library was validated by running the delay estimation algorithms through several fixed combinations of hardware modules, such as chains of adders and subtractors. The same circuits were submitted to LSS for logic synthesis, using both area and delay optimization modes. Table 1 shows the delay results for different chains of adders and subtractors, in normalized time units. In area optimization mode, LSS performs less timing-driven optimizations and consequently, the final delay tends to be similar to the chained delay. In delay optimization mode, LSS tries to reduce delay through the chain by reorganizing the logic across different modules and choosing faster cells. Consequently, the final delay is usually smaller than the chained delay.

The high-level timing modelling was validated by running scheduling using timing analysis for several target cycle times, for several high-level synthesis benchmarks. The output of high-level synthesis (after scheduling and allocation) was submitted to LSS

for logic synthesis. The final gate-level delays were compared to the high-level estimated delays.

Table 2 shows the scheduling results for two designs with different targets for the cycle time. Example SUM consists of a chain of 10 additions, while MAHA1 (taken from [16]) contains 12 control paths, 8 additions, 8 subtractions and 6 conditional operations. The columns under “scheduling” list the target cycle time used for driving the scheduler, the estimated cycle time, and the number of cycles in the longest and shortest paths in the candidate schedule. Also listed are: the final cycle time (after scheduling, allocation and LSS-logic synthesis), the difference between the estimation delay and the final delay, and the CPU time for high-level timing analysis. In order to measure the effects of timing analysis in scheduling, sharing of operators in allocation was not performed.

From the delay differences, it can be seen that the high-level timing analysis performed is reasonably accurate and can effectively be used for guiding the scheduler. The average delay difference in MAHA1 was 9.2%.

As expected, increasing the target cycle time led to a decrease in the number of cycles per path. This indicates clearly the trade-off between cycle time and number of cycles in performance optimization. Performance optimization has to consider the product between the cycle time and number of cycle steps needed for the execution of all paths, which is subject of further research.

If resource sharing is applied a trade-off between performance and area becomes also evident. Decreasing the target cycle time led to more states, and fewer operations per state, therefore increasing the chances for operator sharing. In MAHA1, the schedule with the shortest cycle time needed 1 ALU, while the schedule with longest cycle time needed 8 ALUs.

Chained Operations	Optimized for Area			Optimized for Delay		
	LSS	Est.	Δ %	LSS	Est.	Δ %
2 x ADD8	100.0	99.2	0.8	52.6	58.3	-10.8
3 x ADD8	130.6	127.5	2.4	71.2	80.8	-13.5
4 x ADD8	152.6	152.8	-0.2	84.5	102.8	-21.8
2 x SUB8	78.0	75.4	3.4	56.5	56.5	0.0
3 x SUB8	99.0	93.5	5.5	63.5	69.7	-9.8
4 x SUB8	108.5	109.6	-1.0	87.6	82.9	5.3
2 x ADD16	142.5	143.0	-0.4	82.6	85.8	-3.8
3 x ADD16	189.1	189.1	0.0	110.6	140.9	-27.4
4 x ADD16	241.5	230.3	4.6	125.4	142.7	-13.8

Table 1: Comparison between the delays resulting from logic synthesis and from estimation. (obs.: delays were normalized to the delay of 2xADD8.)

Design	Scheduling				LSS	Δ %	CPU time (sec.)
	Targ. CT	Est. CT	# CS long. p.	# CS short. p.	final CT		
SUM	100	96.6	10	10	113.4	17.5	1.0
	170	159.7	5	5	162.0	1.4	1.2
	210	205.4	4	4	226.0	10.0	1.5
	255	250.0	3	3	266.6	6.6	1.5
	310	292.6	2	2	361.1	23.4	1.5
	510	489.4	1	1	532.3	8.8	0.2
MAHA1	100	85.0	8	3	82.7	-2.7	5.7
	130	125.0	8	2	127.0	1.6	6.0
	150	142.0	4	2	165.7	16.7	5.6
	165	160.7	4	2	160.7	0.0	6.3
	200	186.0	4	1	203.0	9.1	6.6
	215	205.7	3	1	213.7	3.9	5.5
	230	225.7	2	1	275.7	22.2	4.7
	300	297.3	2	1	305.7	2.8	3.3
	365	333.7	2	1	359.7	7.8	2.3
	415	402.0	1	1	301.7	-25.0	0.5

Table 2: Cycle time (CT) and number of cycle steps (CS) for the longest and shortest path with different targets. (obs.: delays were normalized to the shortest possible target cycle time for each design.)

6 Conclusions

This paper presented a comprehensive timing model for high-level synthesis. This model is based solely on the control and data-flow graphs derived from the high-level specification prior to scheduling and allocation. It uses control and data-flow analysis in order to build a timing network which considers different schedules and their effects on registers, functional units, multiplexers, interconnections, and control. The network is built only once and is reconfigured on-the-fly, when considering different scheduling possibilities, by means of parameters applied to the basic timing modules. Delay estimations on this network are used for generating scheduling constraints according to a given cycle time. These constraints are used by the scheduling algorithm, which generates an implementation that meets the cycle time.

The timing modules used in the network are stored in a technology timing library. The delays of the modules are obtained by using the same logic synthesis system used at the back end of synthesis. This guarantees a fine tuning between the high-level timing model and the optimization algorithms applied during logic synthesis.

Results showed that these algorithms can estimate delays at the high-level which are reasonably accurate when compared to the final gate-level delays. Scheduling using timing analysis produced implementations with cycle times within 20% of the estimated cycle time in most cases.

The high-level timing model is general enough to be used in other high-level synthesis tasks such timing driven allocation, and performance optimization. Fur-

thermore, it can also be used for timing estimations of RTL descriptions, prior to logic synthesis.

Acknowledgements

The authors would like to thank Daniel Brand and Louise Trevillyan for the invaluable help in understanding LSS and logic synthesis.

References

- [1] T. M. McWilliams, "Verification of timing constraints on large digital systems," in *Proceedings of the 17th ACM/IEEE Design Automation Conference*, (Minneapolis), pp. 139–147, ACM/IEEE, June 1980.
- [2] T. I. Kirkpatrick and N. R. Clark, "PERT as an aid to logic design," *IBM Journal of Research and Development*, vol. 10, pp. 135–141, March 1966.
- [3] R. B. Hitchcock, "Timing verification and the timing analysis program," in *ACM IEEE Nineteenth Design Automation Conference Proceedings*, (Las Vegas, Nevada), ACM/IEEE, June 1982.
- [4] G. Borriello and R. H. Katz, "Synthesis and optimization of interface transducer logic," in *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pp. 274–277, IEEE, 1987.
- [5] R. Zahir and W. Fichtner, "Specification of timing constraints for controller synthesis," in *International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, ACM/IEEE, August 1990.
- [6] R. Jain, M. J. Mlinar, and A. C. Parker, "Area-time model for synthesis of non-pipelined designs," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, IEEE, November 1988.
- [7] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design*, vol. CAD-8, pp. 661–679, June 1989.
- [8] F. Brewer and D. Gajski, "Chippe: A system for constraint driven behavioral synthesis," *IEEE Transactions on Computer-Aided Design*, vol. 9, pp. 681–695, July 1990.
- [9] S. Note, F. Catthoor, G. Goossens, and H. J. D. Man, "Combined hardware selection and pipelining in high-performance data-path design," *IEEE Transactions on Computer-Aided Design*, vol. 11, pp. 413–423, April 1992.
- [10] J. Darringer, D. Brand, J. V. Gerbi, W. Joyner, and L. Trevillyan, "LSS: A system for production logic synthesis," *IBM Journal of Research and Development*, vol. 28, September 1984.
- [11] R. Camposano, R. A. Bergamaschi, C. Haynes, M. Payer, and S. M. Wu, "The IBM high-level synthesis system," in *High-Level VLSI Synthesis* (R. Camposano and W. Wolf, eds.), pp. 79–104, Kluwer Academic Publishers, 1991.
- [12] R. A. Bergamaschi, R. Camposano, and M. Payer, "Scheduling under resource constraints and module assignment," *INTEGRATION, the VLSI Journal*, vol. 12, pp. 1–19, December 1991.
- [13] R. Camposano, "Path-based scheduling for synthesis," *IEEE Transactions on Computer-Aided Design*, vol. CAD-10, pp. 85–93, January 1991.
- [14] R. A. Bergamaschi, "The effects of false paths in high-level synthesis," in *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, (Santa Clara, California), pp. 80–83, IEEE, November 1991.
- [15] D. Brand and V. S. Iyengar, "Timing analysis using functional relationships," in *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pp. 126–129, IEEE, November 1986.
- [16] A. C. Parker, J. T. Pizarro, and M. Mlinar, "MAHA: A program for datapath synthesis," in *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pp. 461–466, ACM/IEEE, June 1986.