A Procedure for Software Synthesis from VHDL Models

Venkatram Krishnaswamy
Coord. Science Lab.,
University of Illinois
Urbana-Champaign,
1308 W Main St.,
Urbana IL 61801
venkat@crhc.uiuc.edu

Rajesh Gupta
Dept of Information
and Computer Science,
University of California,
Irvine,
Irvine CA 92697
rgupta@ics.uci.edu

Prithviraj Banerjee
ECE Department,
Northwestern University,
4386 Tech. Institute,
2145 Sheridan Rd,
Evanston IL 60208
banerjee@ece.nwu.edu

Abstract

In this paper we address the problem of software generation from a Hardware Description Language (HDL). In particular, we examine the issues involved in translating VHDL into C or C++ for use in system simulation and cosynthesis. Because of the concurrency supported by VHDL, and a notion of timing behavior, care must be taken to ensure behavioral correctness of the generated software. The issues involved will be shown to be different in each of the application areas. The ideas set forth here have been used in an efficient VHDL simulator designed to execute on multiprocessor systems. Results are presented for simulation on uniprocessor as well as multiprocessor systems.

1 Introduction

It is necessary to translate a given VHDL model into a C/C++ program in either of the two following applications. One is in efficient simulation of VHDL models. This is accomplished by translating it into an equivalent C/C++ program, which is then executed on a general purpose processor [1]. The second application is the cosynthesis [2] of systems modeled using VHDL. In cosynthesis, the input VHDL description is synthesized into two portions, one of which is implemented as application specific hardware, and the other as a software program that runs on a general purpose processor. The goal is to automatically generate the C/C++ code for the functionality identified by the partitioner to be in software.

Since most popular HDL's, including VHDL, are based on procedural programming languages, the problem may appear to be a simple translation from one procedural language to another. However, there are two important differences between HDL's and software programming languages that makes this translation difficult. First, in an HDL, the effect of a statement is not guaranteed to take place immediately upon its execution. Using a nonzero delay in an assignment in a VHDL description can cause the effect of the execution of this statement to occur at some arbitrary time in the future. Further the resulting effect of a statement may depend upon the execution of other statements in the description. Since there can be multiple statements which assign values to the same signal with different delays, it is necessary to resolve the ordering of statement execution in order to determine the correct assignment of signal values over time. This process of ordering multiple assignments is commonly known as "signal resolution". Secondly, HDL's commonly support some form of concurrency to model the parallelism inherent in hardware. It is possible to exploit some parallelism in the descriptions for model execution but some portions may need to be executed sequentially in order to observe dependences.

In this paper we will systematically describe the delay models inherent in VHDL, and the difficulties which they pose. Full support of delay models require considerable work at runtime for scheduling events. We will isolate subsets of the delay models which require minimal work at runtime. We apply these findings to our domains of application i.e. simulation and cosynthesis. We briefly mention our simulation algorithms, and give results to demonstrate the effectiveness of our work.

The remainder of the paper is organized in the following manner. Section 2 briefly reviews related work in the area. Section 3 presents a model of the delayed assignments in VHDL that is used in our software synthesis procedure. We describe a compile time technique for resolving signal assignments in Section 4. We discuss our implementation for translation of single and multiple process VHDL in Section 5.

2 Related Work

Our work is based on analysis of delay assignments and signal resolution in VHDL. The semantics of delay assignments in event driven HDL and logic simulators has been

ASP-DAC '97
0-89791-851-7$5.00

addressed by Abramovici *et al* [3] and Augustin *et al* [4]. In particular we concern ourselves with *inertial* and *transport* delays, which are supported by the VHDL language. Augustin [5] describes the semantics associated with these delay models. The VHDL Language Reference Manual [6] contains an algorithm to implement the *preemptive* semantics associated with signal assignments. Augustin [5] and Allen [7] provide formal frameworks within which it is possible to analyze the behavior of the waveforms associated with signals in VHDL. The former provides a *waveform algebra* within which it is possible to define and manipulate waveforms and their interactions with one another. Allen describes a timing model using a set of time intervals and relations between these intervals are expressed in terms of *predicates*. This model is used by Wilsey [8] to describe signal assignments in VHDL.

The problem of HDL simulation has been addressed by various authors. Broadly, there are two different approaches to making the simulation more efficient. The first is in increasing the amount of analysis performed at compile time. Sakallah and Shriver use waveform analysis in [9] to perform Verilog simulation for synchronous systems with assigned delays. French *et al* [10] describe compile time analysis for a more general class of Verilog models, and are able to handle variable delays even in an asynchronous circuit. On the other hand, other groups have attempted to reduce the amount of runtime work. Devadas *et al* [11] have described an analysis procedure to perform *event suppression* to reduce the number of events occurring in the simulation of synchronous digital circuits. In this work, we attempt to combine the advantages of compile time analysis and reducing runtime overhead as far as possible.

3 Delay Models supported by VHDL

A VHDL description consists of a set of concurrent processes. Each `process` consists of a set of statements which execute sequentially. Processes communicate by writing to and reading from data structures known as `signals`. Values may be assigned to signals, and a temporal delay may be associated with these assignments. During simulation, the execution of a signal assignment results in creation of a *transaction*, a data structure comprising the value assigned and the time at which it is scheduled to appear on the signal. This transaction is then added to a list of transactions for that signal. A signal transaction list reflects the values which will eventually appear on the signal. In VHDL, there are two types of delays associated with signal assignments. These are known as *inertial* and *transport* delays. Transport delays model delays on wires or transmission lines which have an infinite frequency response. On the other hand, inertial delays model systems having a frequency response

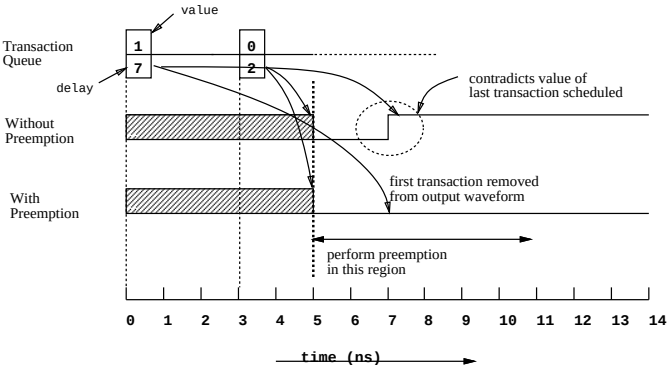

Figure 1: Preemption removes invalid transactions from the output waveform. An event of value 1 scheduled for time 7 is posted at time 0. Then at time 0, a transaction with value 0 is posted at time 3. If the first transaction is not preempted, the second transaction is contradicted at time 7 ns..

equal to the assignment delay.[1]

Since arbitrary delays can be assigned to a signal irrespective of how real life hardware components behave, *preemption* is necessary to ensure temporal causality of events. In other words, transactions posted by one assignment statement may be invalidated by another assignment to the same signal. To understand this consider the example of events in Figure 1 from [5]. Transactions that cause events earlier than already scheduled events result in invalidation of existing events. It is clear that the resolution of multiple assignments and delayed assignments are interlinked.

VHDL signal assignment has *preemptive* semantics [5]. As soon as a transaction is posted by the execution of the signal assignment statement, other transactions on the transaction list are examined for possible preemption as follows. Signal assignments with transport delays require only *forward* preemption, (see Figure 1). Inertial delays require both *backward* as well as forward preemption. Backward preemption entails removal of all transactions scheduled to appear before the most recently scheduled transaction, except those having the same signal value. Figure 2 shows assignment with inertial delays.

3.1 A Model of Delayed Assignment

A model for delayed assignment and preemption conditions has been presented in [8]. In this section, we briefly introduce the conditions for preemption and extend this model to

[1] The latest revision of the VHDL standard, VHDL-93 allows specification of a *pulse rejection* time with an inertial signal assignment. This sets a time limit before which it is guaranteed that the signal value will not change. We do not address this delay model explicitly in this paper. However, extension of the methods described herein to account for this is straightforward.

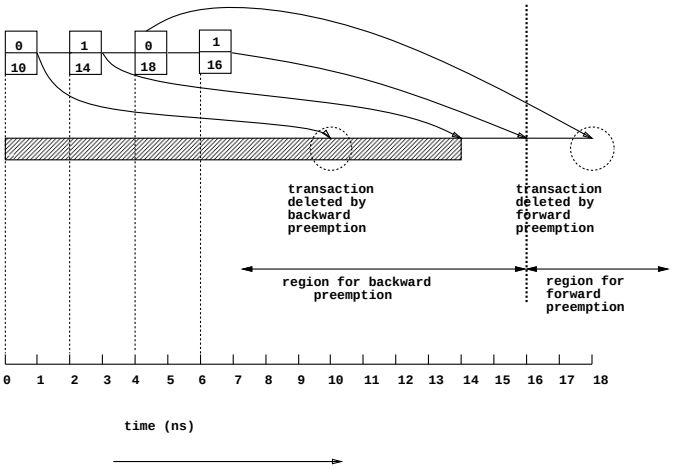

Figure 2: Forward and backward preemption in an inertial delay. The most recently scheduled transaction is located at 7 ns for 16 ns, which makes the transactions scheduled at 10 ns and 18 ns invalid. The transaction scheduled for 10 ns is rendered invalid since the value posted by it is different than the most recently posted transaction. On the other hand the transaction scheduled for 18 ns is preempted by forward preemption.

identify the runtime work needed for multiple assignment resolution in the case of multiple processes.

The timing model is based on the interval temporal logic in [7]. The basic elements are intervals of time, and the predicate **meets**. All other relations between time intervals may be expressed in terms of the **meets** predicate. A time interval t_1 **meets** t_2 if t_1 is before t_2 and there is no time interval separating them. Thus an assertion $t_1 + t_2 + t_3 = t$ means that t_1 meets t_2, t_2 meets t_3 and t is composed of all three intervals. We now consider the following three predicates shown in Figure 3.

- **during(t1, t2)** $\equiv \exists\ t_3, t_4 : t_3 + t_1 + t_4 = t_2$.
- **overlaps(t1, t2)** $\equiv \exists\ t_3, t_4, t_5 : t_3 + t_4 = t_1 \wedge t_4 + t_5 = t_2$.
- **finishes(t1, t2)** $\equiv \exists\ t_3 : t_3 + t_1 = t_2$.

Let us define a transaction by a tuple

$$t = \langle t_v, t_d \rangle$$

where t_v is the value of the assignment, and t_d is the time interval beginning with the signal assignment and ending with the time at which this value appears on the output waveform. Further, suppose that $\langle t_v, t_d \rangle$ is a transaction posted by a signal assignment statement sa. Then a transaction $t' = \langle t'_v, t'_d \rangle$ is forward preempted if

$$\mathbf{during}(t'_d, t_d) \wedge \mathbf{finishes}(t_d, t'_d) \tag{1}$$

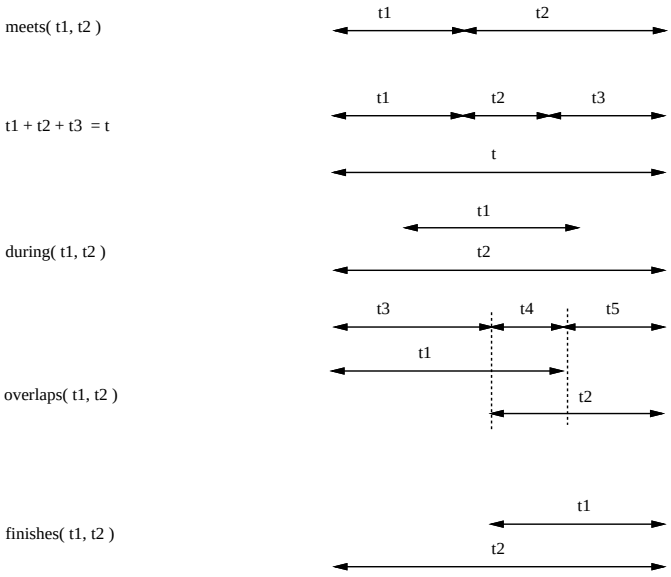

Figure 3: Relations on time intervals.

is true.

In addition, should the transaction t' have inertial delay associated with it, then the first transaction t is preempted if

$$\mathbf{overlaps}(t_d, t'_d) \wedge t_v \neq t'_v \tag{2}$$

is true.

Considering the case of backward preemption first, when the condition 1 holds between the time intervals resulting from two assignments to the same signal, then the values assigned must be compared. The preemption occurs if the two values are different. In cases where it is possible to determine the equivalence between two values, this can be done at compile time. However, in general the values may depend upon input and may not be known at compile time. Should the condition 2 not hold between the intervals in question, then there is no preemption to be performed, regardless of the delay model. A sufficient condition for this to happen is that each signal assignment is executed after the previous assignment has taken effect regardless of the value assigned.

For a transaction $\langle t_v, t_d \rangle$ to be preempted due to posting of a new transaction $\langle t'_v, t'_d \rangle$, it is necessary that condition 1 holds. Let t_s and t_f be the starting and finishing times of the interval t_d respectively. Similarly, let t'_s and t'_f be the starting and finishing times of the interval t'. Since the transaction t' is posted after the transaction t it follows that $t'_s > t_s$. The former transaction is only deleted when $t'_f < t_f$. Hence, no preemption is required if we know for certain, that t'_f is guaranteed to be greater than t_f. A sufficient condition for this is when only transport delays are used, and all assignments to a given signal have the same delay. Thus the time interval associated with assignments

to a given signal are always the same. However, each time interval finishes after the time interval corresponding to the preceding signal assignment i.e. the time intervals corresponding to different assignments to the same signal are disjoint. Under this assumption, no preemption needs to be performed at runtime.

3.2 Modeling of Synchronous Systems

We now consider the issues arising out of delayed assignments in the case of synchronous systems, wherein all signal assignments carried out in a particular cycle are expected to be executed before the end of the current clock cycle. As we have seen above, regardless of the delay model, the value assigned to a given signal is always the same as the most recent assignment to a signal. In a synchronous system, components sample input values only at the clock boundaries. Hence, while it is important to ensure that the correct values are placed on signals at the end of the clock cycle, the exact time at which these values are placed within a clock cycle is immaterial. We therefore observe that is sufficient to execute the last signal assignment to a signal in a process with no delay, and ignore prior assignments to the same signal. Cycle based simulators utilize this observation to produce highly efficient code.

4 Software Synthesis from VHDL

In this section we will describe the issues concerned in code generation for both simulation and cosynthesis. We will utilize the conclusions reached in the previous sections in describing solutions to these problems.

4.1 Software Synthesis for Simulation

In simulating large VHDL models such as microprocessors at the RTL level, we need not concern ourselves with detailed timing issues. This assumes that verification of these systems is broken up into logic verification (achieved by simulation) and timing verification (achieved by static timing analysis or accurate timing simulation). We implement the transport delayed model wherein all assignments to a given signal have the same delay.

As we have shown earlier, given this model, we need not perform any resolution at runtime for multiple assignments. This is because the effect of a given signal assignment appears on the signal before the next signal assignment is encountered. Hence we are able to resolve the order of execution at compile time through an analysis technique first introduced by French [10]. We have implemented this technique in a VHDL simulator. The analysis technique enables us to generate C++ code which reflects a conservative estimate of the order in which the simulation will proceed at runtime. Runtimes of this simulator running on a Sun 4 processor, and those of a commercial event driven simulator, Vantage, which implements the entire set of delay models set forth in VHDL appear in Table 1. We have used some of the ISCAS [12, 13] benchmarks for our work. Since our simulator does not have to perform the cumbersome runtime checks for resolution of signals, our simulator is 2 to 3 times faster, even though we have not spent much effort in optimizing the generated code. This is achieved since we are able to avoid the need for a centralized event queue, and therefore we do not need to dereference any pointers or traverse dynamic data structures such as queues or timing wheels.

Table 1: Runtimes (in seconds) of Vantage vs our sequential simulator on Sun4 machine

.

circuit	vantage	our simulator	# vectors
c432	95.4	37.3	10000
c499	121.7	44.8	10000
c1908	40.5	21.9	10000
c3540	216.3	148.3	10000

Our simulation algorithm can be refered to as "compiled event driven" since it relies on compilation of efficient C code from the source VHDL program, but at runtime provides event driven behavior using branching constructs. There is no event queue involved in this approach. We then parallelize this simulation to execute on shared memory workstations. This is done by effectively partitioning the graph obtained from the static analysis techniques mentioned above. Our partitioner minimizes the communication across processors. Results for our parallel simulator executing on a SparcServer 1000 appear in Table 2. Note that the single process times are not the same as those reported in Table 1 because each processor on the SparcServer is faster than that on which we ran Vantage. In addition to the ISCAS benchmarks, we have used two additional benchmarks. The first, XCVR32 is a 32 bit transceiver which is available from the RASSP web site [14]. Its aberrant behavior is due to poor load balancing achieved by our current algorithm. The second description, MAC, a floating point multiplier and accumulator from Ashenden's book [15] scales poorly owing to the limited parallelism in the circuit. However, speedups of upto 4 on 8 processors are achieved for some of the other benchmarks. In order to achieve these numbers for all benchmarks, we are refining our partitioning algorithms. Detailed descriptions of our implementation of the static analysis, code generation and parallelization of the simulator are omitted owing to lack of space [16].

Table 2: Speedup using multiprocessor simulation on an 8 processor Sparcserver 1000 using Solaris threads. Each processor is a Super Sparc clocked at 50MHz. There are 512 MB of physical memory. Runtimes are reported in seconds.

circuit	1 proc	2 procs	4 procs	8 procs
c432	12.05	8.80	7.07	4.37
c499	15.80	11.39	7.99	6.34
c1908	10.90	6.89	4.30	2.75
c3540	126.1	98.5	79.9	48.8
XCVR32	10.68	8.90	25.85	7.15
MAC	3.65	2.97	3.77	4.77

4.2 Software Synthesis for Embedded Systems

In software synthesis for embedded systems, we assume that we are dealing with strictly synchronous systems. Furthermore, we assume that all assignments to signals within a given clock cycle take effect before that cycle terminates. As has been shown earlier, this enables us to consider only the last signal assignment to a given signal in a cycle, thus simplifying the problem of multiple assignments to a signal. We assume that we are generating code for partitions of a system which have already been assigned for implementation in software [2].

These assumptions enable us to remove unnecessary signal assignments as an optimization as shown in the VHDL segment below.

```
P : process
  begin
   a <= 1 after 1 ns;
   a <= 3 after 10 ns;
   a <= 0 after 2 ns;
   a <= 0 after 4 ns;
   wait;
end process;
```

At the end of the clock cycle in which the above process is activated, it will have the value of 0, and any values resulting from signal assignments earlier than the last assignment will not be sampled by other processes waiting on the signal `a`. Hence, we can simply remove the other signal assignments, keeping only the last one in the process. Assignments to signals within blocks which are executed conditionally (e.g loops or if-then constructs) cannot be blindly removed as mentioned above. In order to determine which signal assignments can be removed from the generated code, we build and analyse control flow graph (CFG) for each process statement. We perform *reaching definition* analysis, and any definitions which do not reach the end of the CFG are removed. The delay values in these are immaterial, since we are only interested in the value at the end of the clock cycle.

In the presence of multiple processes and I/O interactions the generated code consists of multiple lightweight threads generated for each process assigned to implementation in software. A shared memory model of computation is assumed. In other words, each thread sees the same data space. The runtime thread scheduler is ised to multiple process interactions in VHDL.

The scheduler enforces a dynamic order of execution on the processes. It accepts interrupts from hardware components and schedules processes which are sensitive to these signals. Furthermore, changes in values of values modified by software processes implemented as threads are noted by the scheduler and sensitive threads are scheduled. The scheduler is also responsible for taking care of resolution of signals assigned to by multiple processes. This is done quite easily, since VHDL requires a bus resolution function for each such signal. Hence the scheduler invokes the bus resolution function when it is notified that a process wishes to write to such a signal.

5 Summary and Conclusions

We have identified problems in generating C/C++ code from VHDL for the purpose of simulation and cosynthesis. A formal model is used to understand the extent of runtime support required and restrictions on the compile time analysis possible for each delay model supported by VHDL. We have briefly described our simulation algorithms, and the algorithms for parallelizing them. We have included runtimes showing the effectiveness of this algorithm. We have shown how to avoid unnecessary signal assignments from the VHDL in the context of code generation for synchronous embedded systems. We explain how to embed this scheme within a multithreaded paradigm which includes runtime scheduling of processes.

We intend to concentrate on simulation on multiprocessors. Our future work includes relaxing some of the constraints on the delay model, optimizing the generated code and improving the parallelization algorithms to obtain better scaling.

Acknowledgements

This research was supported in part by the National Science Foundation under NSF Career Award No. 95-01615, the Semiconductor Research Corporation under grant SRC 95-DP-109, and the Advanced Research Projects Agency under contract DAA-H04-94-G-0273 administered by the Army Research Office. We would also like to thank Intel Corporation for the donation of an Intel Paragon to the University of Illinois. We appreciate the helpful comments of

the reviewers.

References

[1] V. Krishnaswamy and P. Banerjee, "Actor based parallel VHDL simulation using Time Warp," in *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, May 1996. To appear.

[2] R. K. Gupta and G. DeMicheli, "A cosynthesis approach to embedded systems design automation," *Design Automation for Embedded Systems*, vol. 1, no. 1-2, pp. 69–120, 1996.

[3] M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design*. Computer Science Press, 1990.

[4] L. Augustin, B. Gennart, Y. Huh, D. Luckham, and A. Stanculescu, "Verification of VHDL designs using VAL," in *Proceedings of 25th ACM/IEEE Design Automation Conference*, pp. 48 – 53, 1988.

[5] L. Augustin, "Timing models in VAL/VHDL," in *Digest of Papers, International Conference on Computer Aided Design*, pp. 122 – 125, 1989.

[6] IEEE, New York, NY, *IEEE Standard VHDL Language Reference Manual*, 1988.

[7] J. F. Allen, "Maintaining knowledge about temporal intervals," *Communications of the ACM*, pp. 832–843, Nov. 1983.

[8] P. A. Wilsey, D. M. Benz, and S. L. Pandey, "A model of VHDL for the analysis, transformation, and optimization of digital system designs," *Proc. of the International Symposium on Computer Hardware Description Languages*, pp. 611–616, August 1995.

[9] E. Shriver and K. Sakallah, "Ravel: Assigned delay compiled code logic simulation," in *A Digest of Papers, ICCAD*, pp. 364 – 368, 1992.

[10] R. French, M. Lam, J. Levitt, and K. Olukotun, "A general method for compiling event driven simulations," in *Proceedings of 1995 Design Automation Conference*, pp. 151 – 156, 1995.

[11] S. Devadas, K. Keutzer, S. Malik, and A. Wang, "Event suppression: Improving the efficiency of timing simulation for synchronous digital circuits," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 16, pp. 814 – 822, June 1994.

[12] F. Brglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran," *IEEE Intl. Symp. on Circuits and Systems*, vol. 3, June 1985.

[13] F. Brglez, D. Bryan, and K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits," *IEEE Intl. Symp. on Circuits and Systems*, pp. 1929–1934, May 1989.

[14] "Rassp vhdl models." http://rassp.scra.org/information/public-vhdl/models/models.html.

[15] P. Ashenden, *The Designer's Guide to VHDL*. Morgan Kaufmann, 1995.

[16] V. Krishnaswamy and P. Banerjee, "Parallel compiled event driven simulation," tech. rep., University of Illinois, 1996. under preparation.