

VHDL Performance: The Native Code Approach

Victor Berman
Cadence Design Systems

Abstract

Existing VHDL tools in the marketplace adopt one of two implementation strategies. These implementation approaches are:

1. Pseudo Code: A VHDL model is translated into a stream of data which is interpreted by a body of code, usually written in C, as a sequence of synthetic instructions. A single common simulator program usually executes all models.
2. C Code: A VHDL model is translated into a C program, which is compiled to produce a module which can be linked with other modules, as well as a support kernel. The resulting program is the simulation. Each separate simulation may imply a unique executable program.

Both of these approaches introduce extra layers of representation and indirection which simplify the task of building and porting the simulator, but at the expense of inhibiting VHDL performance from reaching its theoretical maximum.

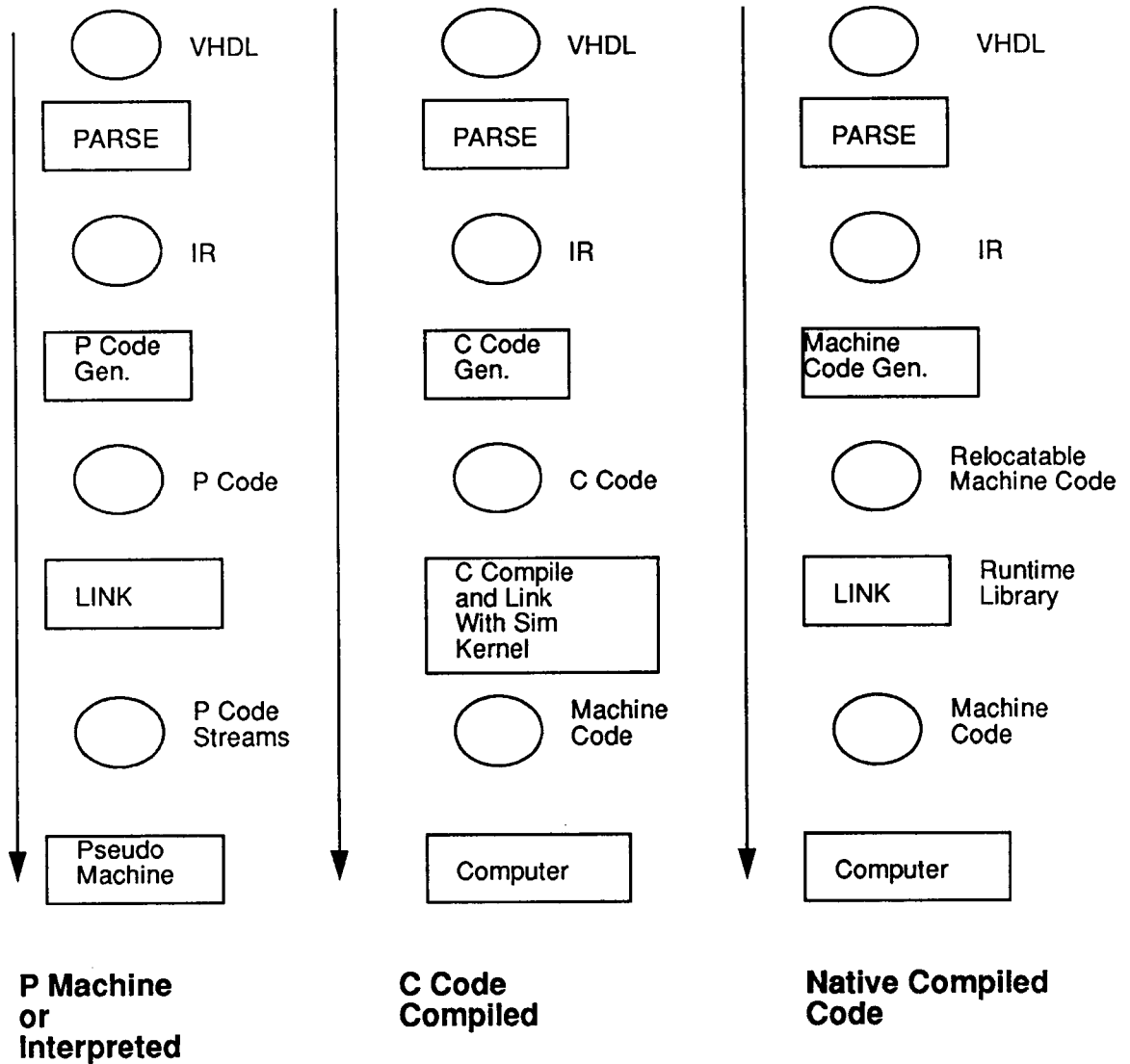
This paper describes a VHDL simulation system based on direct generation of native machine code, which is directly executable by the workstation hardware.

This new approach alone immediately and significantly improves simulation performance, especially at the behavioral modeling level. The paper describes the code generation and optimization strategy, which is similar to techniques used on other RISC-targeted compiler/code generators.

This paper further outlines a strategy for high performance gate level simulation through the application of a set of *methods* appropriate for each network determined by analysis of the network during static elaboration. These methods are executable native code sequences which compute the minimum amount of information needed to manage a signal, as a function of the particular signal usage.

Comparison of Simulation Approaches

The three approaches to building VHDL simulation executable are shown below:



P Machine or Pseudo Code Approach

This approach has been used very successfully for initial implementations of language compilers and simulators. Generally, once the language is stable, the P Code generator is replaced by a native code generator in the interest of improved runtime performance. It first gained wide acceptance with the introduction of the portable "C" compiler in the early days of Unix. These compilers have since been replaced by native code compilers in order to provide higher levels of performance.

In this method a Pseudo Machine is defined with a simple set of usually stack oriented instructions. The parser generates an Intermediate Representation (IR) of the input source text which is then processed by a P Code Generator to form instructions for the Pseudo Machine. An interpreter performs the mapping from the pseudo instructions to the actual machine instructions. The advantage of this method is that the generation of pseudo instructions can be done quickly so that setup or compile time is short. Also, the mapping of the interpreter which maps the pseudo instructions to the target machine is straight forward allowing relatively easy porting.

The disadvantage of this method is that the interpretation of the pseudo instructions is a significant bottle neck for simulation run-time performance and that the simple code generation approach makes compile time error checking difficult. This approach is best suited to designs where quick turn around times are more important than simulation speed.

C Code Compiled Approach

This approach is used in order to take advantage of the workstation C compiler and linker in building a VHDL simulation executable. The VHDL source is parsed into an Intermediate Representation (IR) which is then fed to a C Code Generator which produces the C language equivalent of the VHDL text. This code is then processed by the C compiler to produce relocatable object code. This code is linked to a fixed simulation kernel written in C to produce the executable code.

This method produces significantly higher simulation speed than possible with the interpreted, p-code approach. The disadvantage of this approach is that it is dependent on the workstation provided compilers and linkers. These tools are found to be the bottle neck in terms of setup time and design size since they generally run slower than the parser and have fixed size limitations. In order to get around the size limitations of the compiler it is necessary to break up large designs into small fragments which interferes with any attempts at optimization.

Native Compiled Code Approach

This approach provides the high speed simulation of compiled code and the fast compile/turn-around of P-code but is not dependent on workstation compilers and linkers. In this method, the IR produced by the parser is processed by a code generator which directly produces relocatable machine code. Since the code generator is specific to VHDL simulation, it can perform optimizations which are most appropriate to event driven simulation rather than the more general optimizations which are appropriate to the numerical computations commonly encountered in C programs.

Since the code generation algorithms are specific to VHDL they can produce efficient code sequences to handle queue management for signals with abstract data types and for signal valued attributes. This flexibility provides improved run time performance for these complex operations which are generally handled by a fixed simulation kernel.

Advantages of this method are that by avoiding the use of workstation software, particularly the C compile step, significant time is saved in setting up the simulation run. This also eliminates fixed size restrictions on designs processed and allows for greater optimization since the code generator can look at the whole design, and the code generator is specific to VHDL.

The disadvantage of this method is the complexity of the code generator. This approach can lead to difficulty in porting unless this problem is addressed early in the design of the simulation system.

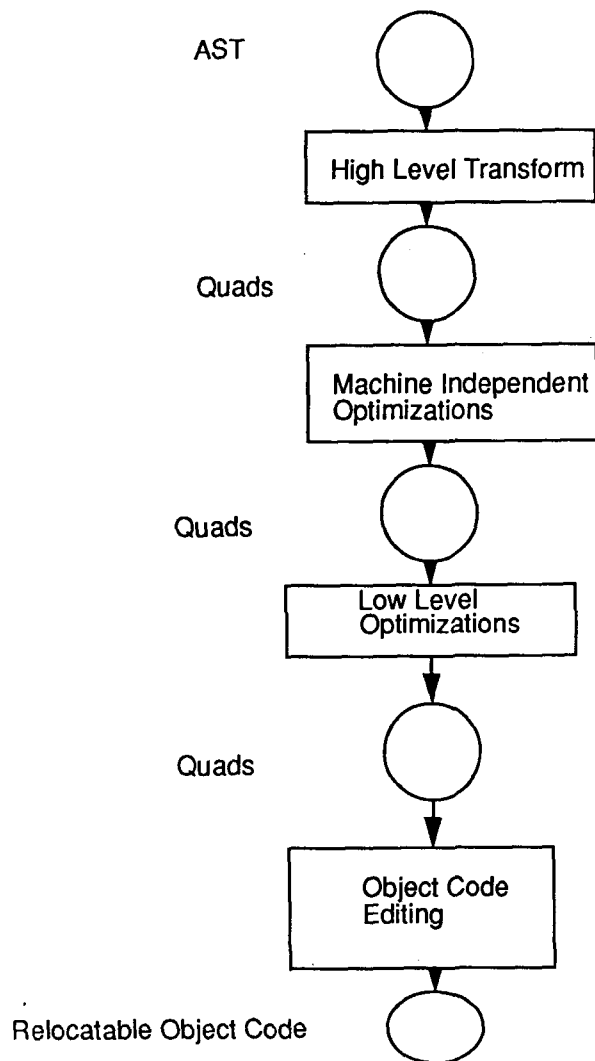
The Code Generation Process

The state of the art of compiler construction has progressed a long way from the early days in the 1950s when the first FORTRAN compilers were hand crafted for each novel computer architecture. With the advent of parser generators and the maturing of the body of knowledge about code generation and optimization techniques high quality compilers for main frame and mini-computers became common place. However these techniques have not generally been applied to logic simulators. Efficiency in logic simulation was largely based on high speed gate level algorithms which were specific to the underlying logic type of the simulator. If an HDL were to be used at all it was generally a simple extension of the logic type supported by the simulator and the efficiency of processing the language and supporting behavioral simulation was far out weighed by the gate level speed.

With the growing popularity of VHDL and top down design, the situation began to change. Processing of the behavioral aspects of VHDL and support for the simulation of signals with user defined abstract types became a significant driver in overall simulation performance. Early VHDL implementations could not deal effectively with these problems since they were only just learning to implement the language correctly. The result was that performance of VHDL tools suffered by comparison to those which implemented simpler languages.

Now that the body of knowledge about the implementation of VHDL has matured, attention can be turned to improving performance. In particular the techniques of code generation and optimization developed for compilers can be applied to this domain.

In the implementation described here, a hand crafted recursive descent parser is used to produce an abstract syntax tree (AST) representation of the VHDL source code. This technique, rather than the more conventional table driven LALR or LR(k) bottom up parser, was chosen because of the processing speed advantage it provides. This AST is the form of Intermediate Representation (IR) which is input to the code generator. The code generator goes through several phases before outputting optimized, relocatable object code.



Code Generation and Optimization Process

The first phase of the code generation process transforms the AST representation of the source into a data structure called QUADs. This format is a three address abstract instruction that represents the next level down between source code and object code. A QUAD is of the form: OP ARG1 ARG2 ARG3 where OP is any operations and the args are generally two operands and a destination address.

The statement $a := b + c$; might be represented by the QUAD

PLUS a b c

The QUADs may now be operated on by the portion of the optimizer which performs machine independent transformations. These include:

- Common subexpression elimination
- Reduction in strength
- Loop optimization
- Elimination of redundant subtype and index constraint checks

Portability Enhanced by Separation of Machine Independent Code Generation From Machine Dependent Code Generation and Object Code Generation.

At this point in the code generation process the QUAD data structures are still machine independent and have been designed for efficient mapping to a generalized RISC architecture., Thus all processing to this point is applicable to any RISC based machine.

From this point on the process becomes more specific to the target machine. First an additional set of optimizations are performed to take advantage of the target architecture. For the SPARC these include:

- Register allocation
- branch delay-slot optimization
- Leaf procedures

The final phase of this process is the object code generation which produces machine specific relocatable object code modules. These modules are processed by a dynamic linker/loader which incorporates needed run time library functions to build the executable simulation image.

Gate Level Speed Through Static Elaboration Optimization

The process of static elaboration is defined in the VHDL LRM and must be performed on the design hierarchy defining the model in order for that model to be executed. The approach taken here is that during that process a parallel analysis of the signal network takes place in order to optimize the processing of signal value and attribute propagation during simulation run time. Since the simulator does not use a conventional fixed kernel, the static elaborator can specify code sequences called *methods* to most efficiently evaluate signals depending on their usage. By analyzing the full network, only the minimal set of methods required are used and no extraneous values are computed. Thus models which do not make use of signal valued attributes are not penalized in terms of either space or run time efficiency. Resolved signals are analyzed for reflexivity (the property of the sig-

nals resolution function that when there is a single source, the value of the signal is defined to be the value of the source). The elaborator identifies reflexive signals, with a single source, and without type conversion functions, and removes the call to the resolution function. This improves simulation performance significantly, particularly when models are generated automatically and extraneous resolved signals are used for ease of generation.

Summary

This paper briefly describes the native compiled code approach to VHDL simulation. It highlights a few of the areas which enable this method to provide both high speed simulation and relatively short set up time and freedom from dependence on workstation supplied software.

This approach represents a third generation of HDL simulation techniques: interpreted, compiled, native compiled. Up until now this approach would not have been practical because of the lack of maturity of the language and implementations and also because of the great variety of computer architectures that needed to be supported. With the emergence of VHDL as a stable standard and with the wide scale adoption of the RISC workstation architecture, this efficient methodology has become cost-effective for commercial, state-of-the-art tools.