

Accelerating VHDL Model Execution

Bihru Tsai
Zycad Corporation
Fremont, CA
btsai@vhdl.org

Abstract

This paper takes a look at current design and verification trends and why acceleration of VHDL execution is desirable. It compares various VHDL simulation techniques and presents a new VHDL simulation environment based on a custom hardware and software. This strategy is targeted at improving VHDL performance and has proven capable of significantly increasing VHDL execution speeds, particularly for behavioral and RTL level models.

Introduction

Today's complex designs exceeding 10k gates benefit from the use of top down design methodology. VHDL with its powerful language constructs and data structures for abstract system level modeling is particularly suitable for this methodology. This VHDL design methodology facilitates evaluation, verification and correction at each stage of the design process, from behavioral to RTL to final gate level implementation.

Time to market pressures, consumer demand for high quality, reliable products, more complicated designs, more product features and tighter development schedules, make it detrimental to discover design errors in the final product. The later an error is found in the design cycle the more costly it is to fix. In fact, statistics have shown that the cost rises exponentially.

Therefore, there is an increasing demand for doing more iterations, better, more comprehensive and accurate verification of the design at the behavioral and RTL level before proceeding on to the synthesis or implementation stage. To accomplish this task, simulation speeds must be fast enough to complete verification within a satisfactory time limit.

Software VHDL simulation speeds inadequate for complete system verification

Given the speed of today's fastest pure software VHDL simulator running on the fastest generic workstation, it would not be feasible to carry out a complete pre-synthesis system verification of your design even if you wanted to. Typically, a realistic verification of a system involving long simulation runs and huge set of stimulus data would take way too long, anywhere from days to weeks. This time frame makes true system simulation prior to implementation impractical.

Some of the causes of this sluggishness can be attributed to the update and management of VHDL signals and processes. On single processor machines, multiple processes which are scheduled to execute concurrently at a certain time are in reality executed sequentially. Signal update, storage, and management adds a lot of overhead to VHDL execution and eats up a lot of memory. Process wake up scheduling and management contributes to additional overhead.

To understand what options are currently available, we need to examine the simulation strategies that have been developed so far.

A look at current VHDL simulation strategies

Current VHDL simulators can be divided into two major categories; those running on generic hardware and those running on custom hardware.

- a. Generic hardware based :
 - Interpretive

With this strategy, the VHDL simulator uses an interpreter program to interpret the pseudo machine instructions generated during the VHDL compilation phase to the actual machine instructions of the host computer.

This method offers a number of advantages. VHDL compilation time tends to be short because the generation of pseudo instructions is fast. The simulator is also very portable since the mapping of pseudo instructions to machine instructions is a direct process.

However, one significant drawback is that run time can be extremely long due to the constant process of mapping pseudo to actual machine instructions during simulation.

Thus, this environment is most appropriate for initial phases of design development where fast turnaround time is desired for models developed and tested standalone.

- C Code Compiled

This simulation strategy generates C code from the VHDL source and then uses the host C compiler to generate object code.

One major advantage this method has over the interpretive method is a much faster simulation run especially for behavioral level models containing lots of sequential statements such as case, procedural calls, etc. and concurrent statements such as conditional, selected, guarded signal assignments.

However, the VHDL compile time is typically longer than that of an interpretive system, since an extra step involving C code compilation is needed. In addition, optimization of the generated C code is limited by simulation artifacts which are hidden in the simulation kernel. The C code generated can also exceed the size limitations of the host C compiler/linker. This in turn requires the decomposition of the C code to smaller pieces, which slows down the VHDL compilation process even further and hinders C code optimization.

- Native Compiled code

With this strategy, object code of host machine is generated directly, thus bypassing the use of the C compiler.

The simulation speed achieved is even faster than the compiled C code method since the object code generated is specific to and highly optimized for VHDL simulation. The VHDL compilation time is also shorter than the C compiled method.

However, porting the simulator to different platforms is difficult since the object code generator is written for a specific machine architecture and must be re-written for each machine that it runs on. To alleviate the porting problem, the code generator should be made as generic as possible. However, this may be infeasible since a lot of the optimizations are machine architecture specific.

- Multi-threaded

This strategy uses a multi-threaded technique to access the multi-processors on the host machine for VHDL simulation activities. VHDL simulation is conducive to parallel implementation due to the inherent concurrency of the VHDL language. With this implementation, processes are removed from a queue by multiple threads for execution on the multi-processors.

The speedup in simulation depends on the design itself and the number of processors. Of course, as with any parallel application, the amount of speedup is not directly equivalent to the number of processors on the host. Speedups slightly higher than 2X have been observed for certain designs running on four processor machines. Designs with VHDL processes containing lots of sequential statements generally exhibit good speedups as well.

b. Custom hardware based :

This simulation strategy employs a combination of software and custom hardware to achieve the shortest VHDL simulation run time. For the environment under discussion, the hardware is a multi-processor system custom designed to execute and accelerate VHDL. Custom hardware offers a number of advantages over generic hardware. Since custom hardware is designed for parallelizing all aspects of the simulation flow, the process of performing time update, signal evaluation, and driver update are also parallelized. In addition, the custom hardware is not limited by popular parallel paradigms like shared memory. The software performs two major tasks; direct generation of hardware specific object code as well as partitioning and distribution of the VHDL design onto the processors.

Let's examine the hardware and software more closely to see how this combination works to accelerate VHDL simulation.

Custom Hardware

The hardware accelerator can hold 1 to 16 boards. Each board consists of 6 processor modules, 3 custom signal processor ASICs, and one communication processor ASIC, all connected together via high speed buses.

Each processor is a MCM module containing primary as well as secondary cache. The processor does not run UNIX but directly runs a custom kernel optimized for VHDL execution. In addition, each individual processor has memory dedicated to it for storing memory models, large data arrays and VHDL process code.

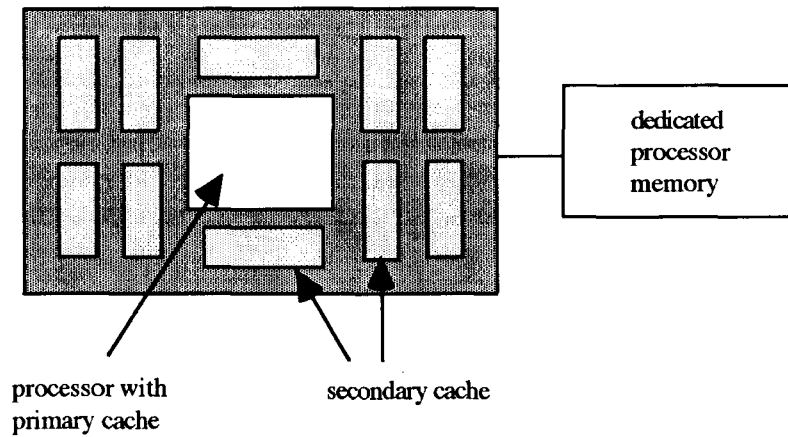


Figure 1 Processor MCM and its dedicated memory

The Signal Processor ASICs are designed to update and manage signals, as well as handle process scheduling and wake up management. They take these simulation overhead tasks away from the processor so that it can concentrate on VHDL process execution. When a new signal value results from executing a VHDL process, the signal processor is informed. The signal processor then figures out which processes are effected by this event and schedules them for wake up.

To facilitate the communication of data between the various boards and with the host computer, a Communications Processor ASIC resides on each board. Primary functions of the Communication Processor include loading the VHDL design onto the boards, interfacing with host computer for text I/O and design data tracing activities.

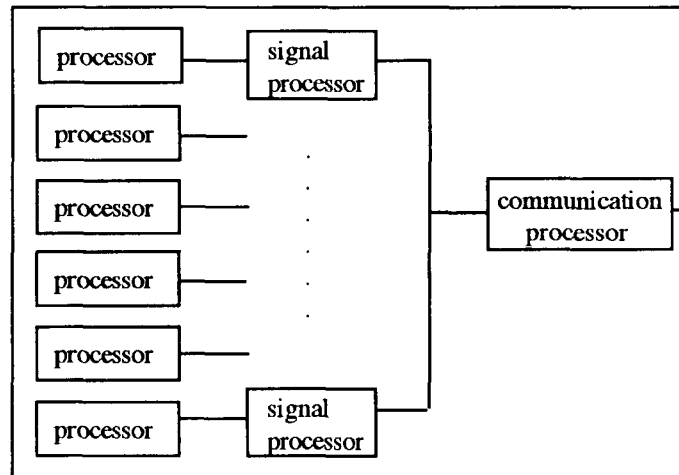


Fig. 2 Accelerator Board Architecture

As seen from the above architecture, the accelerator is custom designed for VHDL simulation and provides functionalities not found in generic multi-processor systems.

Native compiled code and multiprocessor partitioning software

The software analyzes the VHDL source and generates accelerator specific object code that is optimized and runs directly on the processors. The design partitioning software analyzes the design and uses a built-in algorithm to determine the best use of the available processors in the system that will result in the highest simulation performance for that design.

For instance, if a design contains three concurrent VHDL processes that will wake up at the same time, the partitioning algorithm will ensure that they reside and are run on three different processors.

Designs which benefit most from this simulation strategy

The amount of simulation speedup will depend on the concurrency of the design. Designs that have a lot of parallelism already built into them eg. designs with a large number of VHDL processes and plenty of concurrent activities benefit most from this multi-processor partitioning technique. Usually, RTL level designs written for synthesis exhibit high levels of concurrency and therefore are excellent candidates for speedup.

Huge designs such as system level designs that need to be simulated after integration of its sub-systems will definitely see big performance advantages. Even small designs with long simulation run times will show significant run time improvements.

Example designs that fall nicely into the above criteria are video image chips, telecommunication and networking products, computer systems, radars, HDTV, etc.

Conclusion

As multi-processor systems become more popular, applications will be developed to take advantage of this new multi-processing capability. The inherent parallelism of the VHDL language and the VHDL simulation cycle makes it ideal for multi-processing. In fact, a multi-threaded implementation of VHDL simulation has already shown speedup of at least 2X on certain designs running on a four processor system.

The extent of the speedup in a multi-processing environment will depend greatly on how much of the VHDL simulation cycle is parallelized, how efficiently data is transmitted to the various processors and how fast the processors are able to access data needed during processing. On generic multi-processor systems, since a common memory is shared by all processors, memory contention can become a primary bottleneck for high performance. On the contrary, the custom designed VHDL simulation hardware does not have to deal with memory contention problems since each processor has its own dedicated memory. The custom hardware is also designed so that all aspects of VHDL simulation, not just the process execution phase is parallelized. Thus, the custom hardware will be able to achieve higher levels of speedup unattainable with a generic multi-processing hardware.

References

1. Dave Wharton, "Benchmarks test a few simulators" EE Times June 6, 1994, pp.50
2. Victor Berman, "VHDL Performance: The Native Code Approach" Proceedings of the VHDL International User's Forum, Spring 1993 Conference
3. William Paulsen, "Simulation Strategies:Multi-Threaded VHDL Simulation" Silicon Valley VHDL Local Users Group, March 1994
4. John P. Huber, Mark W. Rosneck, "Successful ASIC Design the First Time Through", Van Nostrand Reinhold, New York 1991