

Behavioral VHDL: Focal Point for Hardware Design

**John Willis, Quentin Schmierer,
Russ Hoover, Nancy Perugini,
Wayne Nation, Scott Hinkel,
Dick Booth**

**Application Business Systems (IBM)
Rochester, MN 55901**

1.0 Abstract

This paper describes our experiences using VHDL to develop a large behavioral model of Application Business System's next-generation hardware system architecture. The model addresses the need for a precise, operational definition of the system, analysis of design trade-offs, verification of design integrity, and potential for future hardware synthesis. These diverse needs lead to trade-offs in the level of abstraction, partitioning, and timing. Through this paper we attempt to convey the lessons which would have facilitated our effort and may yet benefit other architects.

2.0 Modeling Objectives

A behavioral model is driven by four primary needs:

- Operational documentation
- Increased probability of logical correctness
- Quantitative feedback on design trade-offs
- Starting point for hardware implementation

Contemporary architectures are often specified in descriptive terms as a covenant between the hardware implementors and software designers. By intent, such specifications impose the minimal set of constraints needed to insure compatible hardware and software.

In contrast, an operational specification defines an architecture as a set of behaviors operating on architecturally-defined resources. The operational specification is an architectural embodiment. On one extreme, it avoids imposing constraints beyond the descriptive specification. However, in a complex system design, we have found that designers appreciate an operational specification which embodies additional constraints compatible with their high-level design. Such an operational specification becomes a readable document addressing the needs of those not directly involved in the design of a specific chip.

In order to design chips, implementors must impose many constraints and invent mechanisms which go beyond the descriptive architectural specification. How will precise exceptions be maintained? How will caches be kept coherent? How will the required store ordering (program, weak...) be maintained? An operational specification which includes significant implementation mechanisms provides a means to study the compatibility with architectural assertions (logical correctness) and the relative merits of different design trade-offs.

Finally, an operational specification provides a more efficient starting point for hardware realization than the descriptive architectural specification. Today, experts

in implementation manually refine operational definitions into realizable hardware. Within a few years, synthesis from operational specifications will be a reality in much the same way compilers for programming languages have largely displaced assembly language programmers.

2.1 Rationale for VHDL

VHDL [IEEE88, PER91] is the best hardware description language we have found for meeting the four needs required of a behavioral model.

VHDL is capable of representing the entire spectrum of processor design activity:

- Analog (RLC, transistor and switch)
- Gates (OR, NOR, XOR)
- Register Transfer (Latches and combinational logic)
- Behavioral (Algorithmic)
- Queuing (Resource utilization and contention)

In an effort to bridge this tremendous representational range, VHDL emphasizes constructive mechanisms rather than a small set of design primitives. These constructive mechanisms include:

- Multi-dimensional array types
- Record types
- Enumerated types
- Access types (and dynamic object allocation)
- Text I/O
- Physical types
- Overloadable subprograms and procedures
- Resolution functions
- Alternative architecture to entity bindings
- Strong typing and subtyping (implied checks) helps to insure that constraints are consistent across interfaces.

With a few minor exceptions on the two extremes (analog and queuing), ABS has standardized on VHDL among architects, implementors, and (to some degree) chip technologists. This common language greatly facilitates use of VHDL for opera-

tional documentation and as an auditable embodiment of implementations.

By choosing the most widely accepted hardware description language in the world, ABS is able to take advantage of tools and expertise developed by thousands of people. As of late 1993 there are more than a dozen commercial VHDL simulators, nearly a dozen verification tools, more than six hardware synthesis products and a growing body of leading research oriented toward VHDL.

The remainder of this paper concentrates on our use of VHDL in the ongoing behavioral model of a new multiprocessor system for the AS/400 family.

3.0 Abstraction Level

The single greatest challenge in a VHDL behavioral model is to set and maintain the right level of abstraction. If the model is too abstract, it will not meet documentation, correctness or analysis requirements. If the model gets too far into the hardware implementation, it will take so long to complete that the result will be of diminished value. Furthermore the model will lack the plasticity required for trade-off analysis, and will remain sensitive to an evolving hardware design.

We chose a modeling level which would try to produce identical signals on each subsystem boundary at the end of each clock cycle when observed from the behavioral model or real hardware. We specifically did not try to model behavior within a subsystem or timing within a single cycle. As those familiar with behavioral modeling will attest, this approach to modeling will never correlate exactly between behavioral model and hardware, however it is a readily understandable modeling goal.

The second challenge in behavioral modeling is to make the best possible use of more abstract types than bit and bit vector. Integers, reals, enumerated tags, and records provide constructive mechanisms

which raise the level of modeling abstraction.

In order to represent bytes, we defined a type which could either hold an integer subtype ranging from 0 to 255 (integer subtype) or a record with additional tagging information (for consistency checking). Addresses utilize a constrained array of such bytes in order to represent integers with range exceeding the four bytes typically provided by VHDL simulators (and the VHDL standard).

Enumerated tags provide an easily readable mechanism for representing the states of a finite state machine, the encodings on an interconnect, execution unit operators or other “bit” encodings. Through use of a named tag, the behavioral model avoids placing encoding constraints on hardware implementors. Avoid the tendency to choose terse, 3-4 character tags. Descriptive tags are important for documentation needs. Modern VHDL simulators should easily support 32 or more characters in a name.

Record types allow the modeler to collect a variety of objects with diverse types into a single aggregate object. For example, the collection of signals comprising an interface (address, read, write, strobe, etc.) can be packed into a single record at the interface of design entities. As signals are added or subtracted during the design process, only the record definition needs to be updated (not the entities or components). Cache entries, translation lookaside buffer entries or status words are other natural uses for records. Unfortunately, VHDL does not yet support variant records (known as unions in C), thus some communication interfaces are a little more complex than need be (see the byte vector techniques above).

VHDL ultimately compiles a model into a network of distinct processes communicating through signals. Choice of the right process boundaries is essential for the success of a behavioral model. Each process should advance to a new state on a globally defined clock edge. Avoid the

temptation to pass signals between processes on anything other than the leading or trailing edge of a clock; repartition instead.

In some cases, it may be known well in advance that a particular subsystem will be operating at a significantly different clock rate. To capture this aspect in the behavioral model (e.g., for performance trade-offs), some sort of speed-matching interface is needed. In this instance, an asynchronous process (triggered by push/pop signals with empty/full status signals in both clock domains) can efficiently operate as a pass-through buffer. By changing only the name of the record type of the data being passed through the process, this simple process can be reused many times in the same model.

VHDL distinguishes between signals and processes. Signals have greater than zero delay between assignment and visibility of the new value. Variables are local to a single process, however the result of an assignment is visible at the beginning of the next sequential instruction. Signals generally have greater overhead than variables during simulation, so choose process boundaries carefully.

VHDL’s concept of time consists of delta delays and finite time intervals. As long as there are new events scheduled for evaluation, VHDL permits an infinite number of delta cycles within each femtosecond. Signals propagate from driver to receiver during each delta cycle (as well as on advances of “finite” simulation time). In order to catch unintentional loops, most simulators allow the user to place a bound on the number of delta cycles. For efficiency and increased temporal range, simulators also allow the designer to define a minimum time step greater than a femtosecond.

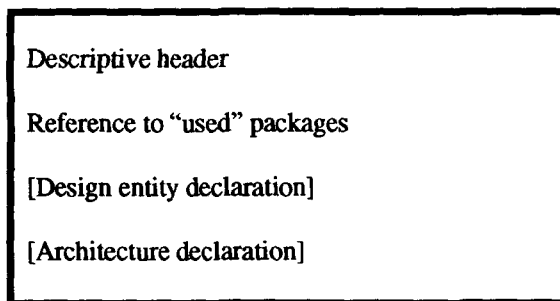
If observation (audit) points are carefully defined in time (clock edges) and space (subsystem boundaries), then abstraction need not mean inaccuracy. A collection of bits abstracted into a constrained integer retains accuracy. A set of bus lines

abstracted into a convenient record retains architecturally meaningful information.

4.0 Modeling Style

The modeling style we evolved places a package and body, an entity or an architecture into a single file. Entity and architecture files have the form shown in Figure 1.

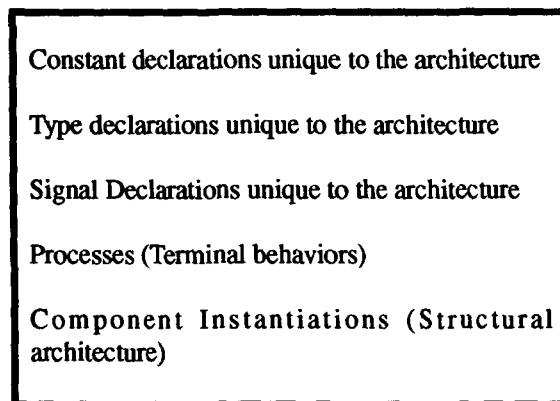
FIGURE 1. Structure of source file



We found that the revision control system (RCS) is very useful in maintaining fields within the header and insuring that a single file update is in progress at a time. Earlier versions can be reconstructed automatically by RCS.

Each architecture declaration has a form similar to shown in Figure 2. Constant and type declarations preface the architecture's declarative region. Process behaviors precede structural behaviors,

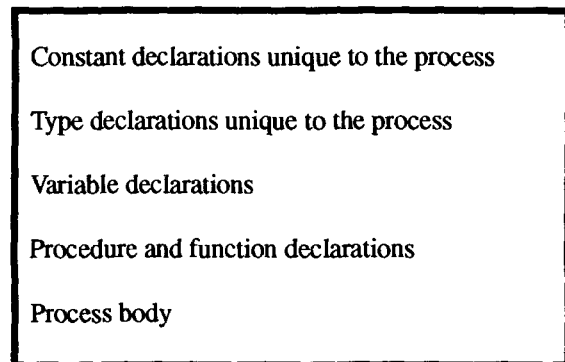
FIGURE 2. Structure of architecture declaration



Each process has the form shown in Figure 3. The process body is generally less

than a page in length so as to make clear the flow of control within the process. Procedures define abstract operations on the process's variables and visible signals. Functions define operations in terms of call parameters without side effects.

FIGURE 3. Structure of process



For example, consider a process modeling a cache. This might be the only process defined within a cache architecture. Either generic parameters defined within the architecture declaration or visible constants might define the cache line width, number of congruence entries and number of associative entries. A set of type declarations might define a cache entry, then the two dimensional array of cache entries defining the cache. This type would be associated with a variable. Procedures might handle cache lookup, misses, and coherence operations. Functions provide utility operations such as the computation of an address hash or extraction of the byte offset within a cache line.

The cache example might have a process body such as Figure 4. The `handle_reset` procedure waits on the deassertion of reset, however the only other wait statement appears at the end of the process body, waiting on the rising edge of the clock. Each time the clock signal goes to '1', the process executes again.

Other than the reset handler, the presence of a single wait statement at the end of the process body leads to relatively simple timing interaction between processes. (every process executes exactly once on each clock cycle).

FIGURE 4. Example process

```
process example is
<constant, type, subprogram declarations>
begin
if (reset = '1') then handle_reset() end if;
handle_cache_op();
handle_memory_return();
handle_coh_op();
wait on clock until (clock = '1');
end;
```

VHDL's strong type and subtyping capabilities provide an important consistency test. Early in our design, a length specification was set to run from 1 to 256. One designer encoded this as 0 to 255, another from 1 to 256. When the two operational specifications were integrated, the compiler determined that there was a subtype mismatch.

VHDL also requires that all choices in a case statement are covered by some action. While the language provides "others" as a valid choice, trying to explicitly cover all cases within state machines forces consideration of each reachable state. For us, the process of defining obscure states was an important part of the early design process.

5.0 Enhancing the System Specification Process with Behavioral VHDL

The use of behavioral VHDL offers unique opportunities to enhance both the efficiency and quality of the system specification process. Analysis of various performance and complexity trade-offs are much more feasible at a behavioral level than at more detailed design levels.

5.1 Inline and Procedural Instrumentation

While assertion statements and processes do not correspond to actual hardware, they are an important part of any operational specification. Within our processor building block, we used more than 140 assertions statements to verify claims made in the descriptive architecture specification. Simple assertions insure that alignment rules are always met or that a sequence number is not reused too early. In our model, much more complex assertion processes insure that consistency predicates are met. By using in-line and procedural processes tasked to verify either simple or complex assertions as the model is executing, the laborious and expensive task of creating and post-processing an arbitrary number of traces (e.g., instruction execution traces, memory accesses, bus cycles) is eliminated. Collecting and storing such traces for a substantial system size will tax most any file system.

5.2 Stimuli

The value added by having a behavioral VHDL model of a system architecture lies in the ability to execute massive amounts of test cases early in the system design cycle. Simulating a system via a behavioral model is much cheaper than simulating the same system at the gate level. While simulation of the actual hardware is still necessary, behavioral simulation can be accomplished much earlier in the design cycle and will trap architectural problems very early. It is of great value to single out and resolve architectural problems before they are manifest at the gate level.

With a new instruction set architecture, acquiring the required stimulus early in the design cycle is a major challenge. We are pursuing two approaches: trace driven and fully functional instruction execution.

Trace driven execution replaces each processor with a processor shell. This shell fetches trace entries from a file. Each

entry includes an address (real or virtual), read/write attribute, transfer length, and related attributes. The trace files we are using were acquired and adapted from a prior generation machine. A synthetic workload generator has been developed that is capable of generating "inorganic workloads" from statistical parameters.

Actual fetching and execution of instructions requires a fully-functional processor model, a major undertaking in itself. Such a model is under development. When this model is delivered we hope to execute instruction sequences from a new random test pattern generator [Mal93] with extensions for multiprocessors. As compilers and key operating system code becomes available, we would like to take the model through IPL into execution of benchmark applications.

5.3 Parametric Modeling and Code Reuse

Gate level designs typically do not allow the flexibility of doubling the width of a data bus, increasing the size of cache, or doubling the number of processors. However quantifying the impact of such alternatives provides valuable information to guide design trade-offs. By approaching the construction of the behavioral model with this intent, models can quite easily be written at the behavioral level to allow parametric instantiation of components with a variety of performance "personalities."

With enough foresight, even functionally different components in a system can be represented with a single piece of parameterized code. For example, a cache memory code component can be written to allow such things as set size, replacement algorithm, and write-through properties to vary. This suggests the possibility of using the same piece of code to be parametrically instantiated as an instruction cache, data cache, and/or L2 cache memory. The same approach applies to components such as cache/bus interfaces and main memory/bus interfaces.

6.0 Modeling Experience

Writing a behavioral model which embodies architecture and select implementation mechanisms is a time-consuming effort and is frequently underestimated. A substantial part of the time goes into learning the language, tools, and behavioral modeling style. We initially did not account for this 2-3 month learning curve when new designers joined the behavioral modeling effort. In addition to design expertise, experience with a programming language such as Pascal or Ada shortens the learning curve.

In our multiprocessor model, each subsystem model roughly corresponded to a chip. Examples include the processor and memory storage control unit (address translation and cache). Each model runs about 8K to 14K lines of VHDL source code (measured as the number of semicolons).

Within each subsystem, it is tempting to partition the behavioral model along the same lines as the actual hardware design. This generally leads to more complex timing than a behavioral model requires. Knowledge of the actual subsystem implementation can be a substantial impediment to behavioral modeling.

As one might expect, a behavioral model is most efficiently done with a small group (1-3 people) who are familiar with the architecture and general implementation approach and have done several VHDL models before. If a large team of implementors is engaged to write the behavioral model, then ready access to simulators, VHDL consultants, and a style auditor is essential. An auditor needs to insure that compatible timing and abstraction styles are evolving. Without a predefined timing and abstraction style followed-up by periodic audits, the result is likely to be unsuitable for integration.

Behavioral modeling works best when treated as an artist's sketch pad, recording what is known about the design without getting bogged down in the fluid

portions. The behavioral model can be a real learning experience, identifying what is not well understood.

After trying several different approaches, ranging from a teams of one to fifteen, we found that the most effective initial approach was to have one person familiar with the subsystem design and behavioral modeling write the code followed by designer audits. Through the audits, designers begin to see a new VHDL style with minimal impact to their primary chip delivery. In the future we hope that these implementation teams will learn from what they've seen in the audits to do their own behavioral modeling.

Once models are written and tested, simulation capacity and throughput determine modeling effectiveness. We found that several vendor simulators (MTI and Intermetrics) were capable of simulating behavioral models consisting of a few processors, however model builds of 30 to 90 minutes are not unusual. In the future we hope such simulators will provide an incremental or expedited model build.

Design trade-off analysis and exhaustive tests require simulation rates (100s or 1000s of simulated instructions per second) which are beyond the capability of today's vendor simulators. IBM Rochester is developing its own parallel VHDL simulator (MINSIM [Wil92, Wil93]) in an effort to meet current needs for higher capacity and higher performance.

7.0 Summary

From what we've seen so far, we believe that behavioral modeling is key to better understanding our design, increasing the overall quality of the system specification, and reducing development cycle time.

We've found that behavioral modeling is a non-trivial skill embracing facets of architecture, hardware design, and programming. Tools for behavioral VHDL design are improving each week, and a polished pathway is expected to evolve. Through

this effort, we hope to accelerate the development of such tools by making appropriate test cases available.

8.0 Acknowledgments

The appreciation of the authors go to the to the Rochester and Yorktown management teams for having the vision to try behavioral modeling despite a tight development schedule. Many thanks to Rochester's design teams for taking time to explore behavioral VHDL and making this model happen. The designers who wrote or audited this model included Dave Krolak, Lyle Grosbach, Fred Ziegler, John Irish, Jim Marcella, Russ Hoover, Don Baldus, Eric Rotenberg, Nancy Duffield, Dan Young and Robert Burton.

9.0 References

- [IEE88] "IEEE Standard VHDL Language Reference Manual," IEEE Std. 1076-1987, IEEE., 1988.
- [Mal93] Yossi Malka, et al., "Model-Based Test Generation for Processor Design Verification," Technical Report 88.337, IBM Haifa, Israel, 1993.
- [Per91] D.L. Perry, "VHDL," McGraw-Hill, 1991.
- [Wil92] J.C. Willis and D.P. Siewiorek, "Optimizing VHDL Compilation for Parallel Simulation," IEEE Design & Test, Sept. 1992, pp. 42-53.
- [Wil93] J.C. Willis et al., "Optimizing VHDL Compilation for Parallel Simulation," VHDL International User's Forum, 1993.