

System Design

Building Up Chips Using VHDL and Synthesis

One key to using VHDL to build chips is to remember what the lines of code mean in terms of hardware.

by Doug Warmke

Designers just starting with VHDL are often worried about using the language effectively. They are afraid of writing unsynthesizable code, or code that will generate too many gates, or a design that is less efficient than they could generate by hand.

Five hardware designers at IKOS Systems Inc. (Cupertino, CA) recently completed the design of the company's next-generation hardware accelerator, called Nsim. A pure VHDL methodology was used to build four complex ASICs in less than 18 months. None of the designers had VHDL or synthesis experience when starting the project. In the end, the project was a great success. There were no chip turns, and the first beta unit shipped less than 2 months after the last chip was back from the foundry.

The Nsim project taught us how to write VHDL code effectively for high-performance ASICs. This article will outline different strategies and offer design tips gained from our experience. It assumes the reader has some familiarity with VHDL syntax and characteristics. After reading this article, you should feel more comfortable writing VHDL code that will synthesize well enough to meet and even exceed your design goals.

Design and methodology overview The Nsim project is a new design based on an existing family of hardware simulators. The project consists of designing four large ASICs (one 100, one 70, one 50, and one 40 kgate array) and integrating them into a complete system. All ASICs are packaged in 476-pin ceramic PGAs. The process is American Microsystems Inc.'s (AMI, Pocatello, ID) 1 μ m CMOS with 3-layer metal (TLM). There are numerous embedded RAMs in 2 of the chips. The operating frequency is 20 MHz. A 50 ns clock period is needed to accomplish SRAM accesses every cycle without the complication of wait-states. The design is all-synchronous, and with the exception of the I/O pad rings, all of the chips are implemented 100 percent in RTL source (so there is no hand-drawn circuitry). ASIC test is done with a full-scan methodology. AMI was contracted to provide the test vectors and to build the clock trees (grid implementation).

The project design philosophy is pragmatic and conservative. There is only one clock phase. There are no tri-state drivers or asynchronous logic anywhere in the chip cores. This conservative design philosophy was shaped by time-to-market pressure, the complexity of the project, and the fact that the designers were first time VHDL/synthesis users. Note, however, that conservative design rules don't exclude designing for the highest possible system performance.

The following is a simplified 10-step overview of the design methodology used:

1. Specify system requirements in FrameMaker documents.
2. Design top-down using block-capture tools (ECS).
3. Fill out and simulate the "leaf" VHDL entities using RTL code.
4. Start validating larger and larger chunks, bottom-up style.
5. Synthesize in parallel with validation
6. Use synthesizer's timing and gate count reports to judge synthesis results.
7. If needed, re-code VHDL; work on synthesis constraint files, go back to step 3 or 5.
8. Simulate chips and system (Voyager 1.30a).
9. Run static timing analyzer on post-layout netlist (Motive 3.5).
10. Run system simulation regressions on post-layout netlist (Voyager 1.30a).

The system has a parallel processing architecture in the form of a cube (see Figure 1). The basis of the system is the evaluator unit, which is the fundamental event processor. An event processor is fairly similar in complexity to a pipelined microprocessor. The user's network under test supplies the processor with its "instructions." The network is stored in RAM. Its "instructions" consist of primitive opcodes, delays, etc. for each gate in the simulation.

As simulation progresses, the event processor updates each primitive's state as necessary. The primitive pin value memories may be thought of as "data" for the processor. During the simulation, the user may trace the state of any primitive's pins to determine if the network is functioning as desired.

The simulator architecture also includes an I/O unit acting as a two-way interface with the host workstation. It is used to source stimulus into the gate-level simulation, and it collects the gate responses and sends them to host processing software.

In such a processor architecture, there is heavy pipelining and heavy memory access. The ASIC design therefore consists mostly of control and datapath elements (i.e. state machines, muxes, adders, etc.). There is some numerical calculation in one of the chips, but it is not as complex as that of a graphics or DSP processor.

Let us consider the cube architecture. Each of the cube's vertices is called a *cluster*. Clusters contain 8 processors and a message router. The processors may be either event processors (evaluators) or I/O processors. There may be up to 256 clusters per cube. The initial product has 8 clusters and allows up to 8 simultaneous users.

There are 64 processors in the cube shown in Figure 1. All processors communicate to each other via "messages" that flow between processors on 45-bit wide busses. There are two bus specifications: CLINK and EBUS.

ASICs are partitioned across the design. The 8 evaluators on each cluster are connected to each other via the EBUS. Each cluster also has a router chip, which routes EBUS messages in and out of the cluster in coordination with other neighbor clusters. The routers communicate with each other via bidirectional CLINK busses.

Partitioning an ASIC for synthesis A top-down chip design begins with constructing a block diagram or model of the ASIC. Chip modularity is usually based on functionality, e.g. a bus controller and all its associated logic exist in one module. For success, there are two things a designer must keep in mind when blocking out a design for synthesis. The leaf modules should be reasonably sized and effort should be made to limit the number of separate entities a timing path can traverse.

By limiting module size, a designer gains improvements in both the synthesizer runtimes and the quality of the

synthesis results. The average entity size in the project was around 2 kgates. The largest entity was around 12 kgates. In general there was a broad range of sizes, including 140 gates, 2 kgates, 6 kgates, and 10 kgates. We needed a Sun Microsystems SPARC 690 server with around 500 Mbytes of swap space and 256 Mbytes of core memory to synthesize the largest blocks. We started out partitioning the chips on a purely functional basis. If we hadn't had such powerful computing resources available, we would have been forced to pay more attention to up-front chip partitioning.

By limiting the number of separate entities that a timing path can traverse, a designer can improve synthesis results in both timing and area. The synthesizer does much better on logic that is contained within one synthesis run. If a certain timing path travels through four separate modules, the synthesizer will make its best effort within each of the modules, but the overall results won't be as good as the case where the whole path is in a single module.

Often, bad timing paths that span multiple modules aren't obvious at the start of a design. Later on, after most of the design is complete, the designer will have to revisit the design's hierarchy structure. He will then modify the module connectivity and functionality in order to work down the long timing paths.

There is a trade-off between these two factors. The synthesizer always does better on smaller-sized entities. However, it also does better when all of a path's logic is contained within one entity. In the case of three of our chips, the final partitioning was affected by these synthesis concerns.

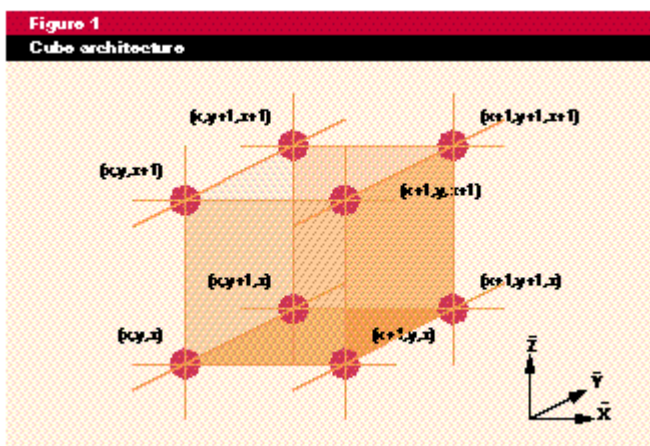


Figure 1. The cube architecture provides extremely high band-width for inter-processor event messaging.

Thinking in hardware It turns out that writing effective VHDL RTL is easy. The key is to realize that the code describes hardware, not software. The main idea of "thinking in hardware" is to always have a good idea of what the lines of code really mean in terms of hardware. You must know it when you lay down an adder. You must be aware of the cost of a magnitude comparator. You must be conscious of the relative timing of different signals with respect to the clock edge. After writing a sequence of RTL, you should be able to sketch out a quick block diagram of what the code physically represents.

The fundamental unit of an RTL description is the VHDL "process." When thinking in hardware, there are two kinds of processes: a combinational cloud process and a register-instantiating process. As shown in Figure 2, the combinational processes have sensitivity lists that include all signals used in right-hand-side operations,

while register-instantiating processes only have the clock and/or reset signal in their sensitivity lists.

Note that the register processes may include some combinational logic, but all effects of that logic must end up in the D-input of a register instantiated in the process. The inputs should not feed non-registered output logic. If they did, the output logic wouldn't simulate correctly, since its inputs are not in the process's sensitivity list.

Combinational processes make up the bulk of the gates of the design, and all complexity is described in them. One must structure combinational processes intelligently to avoid gate explosion and poor timing paths. Processes in a given design should all have the same form. Use variables to structure datapath elements such as adders and comparators. Always group such resources together at the top of the process and assign their outputs to intermediate variables. Later in the meat of the process these variables will be used when the result of the datapath resource is required. This will always reduce resource use to a minimum, and the designer has a quick reference to all gate-hogging components inside the process.

Timing: still everything Timing is still supremely important. Perhaps that is the most important part of "thinking in hardware." Writing RTL that does not take a signal's timing into consideration is not useful. Signals may stabilize at any point in the clock cycle. Register-driven signals always have the earliest stability time, and different signals may have different maturity times depending on how much combinational logic precedes them in the timing path.

Writing an RTL process that uses four 32-bit adders in series is usually impractical. The designer must recognize that a clock stage is needed to break up the timing path. A register should be called out to hold an intermediate

term, and the final addition(s) should be done in the next clock cycle. The concept of relative signal timing is nothing new in hardware design, but it is easy to have the perception that VHDL will somehow magically take care of the issue.

VHDL takes care of gate-level complexity. It does not take care of gate-level timing. It is easy to forget this simple fact. However, after the first few synthesizer runs that result in a minimum clock period 180 percent longer than the target, relative signal timing becomes very clear.

In the design, it was especially easy for us to make mistakes with signals arriving late from other VHDL entities. Late-arriving signals should be run into a minimum of logic before a register stage. Miscalculations of this kind are usually only evident after the first full-chip timing report is made.

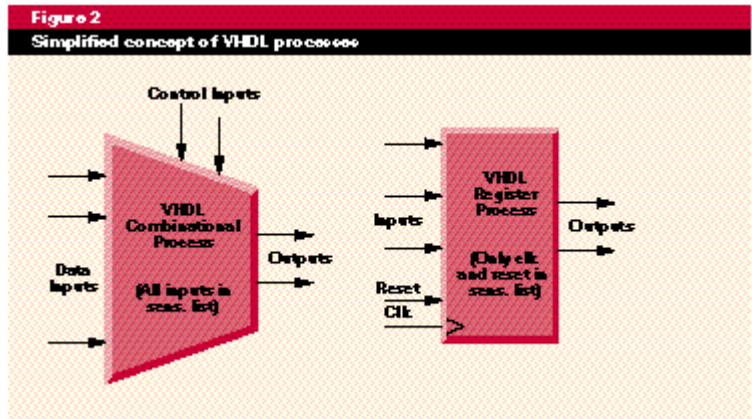


Figure 2. The HDL processes can be thought of as blocks of logic with outputs, control inputs, data inputs, and clock inputs.

Building the actual chips Each chip consists of several dozen entities and at least 20,000 lines of RTL code, including comments. To synthesize an entire chip, a certain methodology must be decided upon and used consistently. Synopsys provides two fundamental means to achieve complete-chip synthesis: iterative entity compile and hierarchical compile.

The project used the iterative entity compile approach. The idea of this approach is as follows:

1. Synthesize each RTL entity independently.
2. "Link" the design hierarchy. This is not "synthesis"; rather, it is a means for the synthesizer to gather the chip's high level structural and timing information into one database. A complete-chip timing report is available here.
3. "Characterize" each entity. This synthesizer process calculates all I/O information pertinent to the entity being characterized (output signal loads, input signal drive strengths and arrival times, and delay constraints).
4. Create a synthesizer script file that contains all the characterize data.
5. Go back to step 1 and use the characterized data as input to the synthesizer.

The main work involved in complete-chip synthesis is to beat down long timing paths. This process takes a lot of time and CPU cycles, especially for big chips. A designer beats down timing using a combination of two methods. The first is to provide the synthesizer with an optimum set of script files containing constraints and accurate environmental data. The second method is to restructure the hardware with the VHDL code.

At the end of the iteration process, there will be one script file for each entity in the design. These script files are generated by using the synthesizer's "characterize" and "Write_script" commands. They do not have any constraints or design rules in them. They only contain environmental data such as arrival times, loading factors, etc. The accuracy of this data is important for the synthesis process.

To start generating correct environmental data, the designer must supply arrival times and loading factors at the chip core's boundary. This information is entered into a special top-level script file. It is a good idea to add pessimistic loading factors to the chip core's ports, since the chip layout may include big delays in the paths from the chip core to the pad ring. The user-provided boundary factors work their way into the chip core with each iterative chip build. Eventually the various arrival times and output delays in the chip core reach a stable and realistic point, but it does take several iterations.

In addition to the automatically generated environmental script file, each entity needs its own set of constraints. The synthesizer needs a little help in figuring out where to best spend its effort. Only the designer can provide it such knowledge. Some of the constraints we found most effective were "max_transition" and "set_output_delay." (These commands have been renamed with newer versions of the software.) The entity-specific constraints get placed in another special script file. Once these files are generated, they don't need to be adjusted too often.

Synthesis characteristics of the Nsim chip set

The IOCHIP contains over 100 kgates with 5 embedded dual-port RAMs. It moves data in and out of the simulation over a variety of channels. The channels are all relatively independent of each other, but data is bussed throughout the chip. The main challenge in this chip was to code up all the independent entities and tie them together in a way that would fit on the base die. The final VHDL code was very similar to the pre-synthesized code. However, we did have to pull many "excess" registers out of the design in order for the Vendor to be able to route the chip. Pulling the registers cost us some monitoring features that weren't critical to the overall simulator architecture.

The QCHIP contains 70 kgates and includes 3 embedded dual-port RAMs and 1 embedded single-port. It is algorithmically very complicated. There are many sophisticated state machines in the ASIC. The largest one has 256 states and synthesizes to around 12.5 kgates. This state machine was written in 4,000 lines of RTL. Many of the state machine inputs come from off-chip SRAM read data that arrives late in the clock cycle. The QCHIP had the most difficult place-and-route results of all the chips, chiefly due to the dense, timing-constrained state machine architecture.

The QCHIP's final VHDL code was fairly similar to the pre-synthesis version, with one major exception. The 12.5-kgate state machine took over 20 hours to synthesize in its original form. This state machine has many registers that hold intermediate data terms. (Over 5 kgates worth). The designer decided to split the entity in half, placing the combinational logic and a few key registers in one sub-entity, and placing the bulk of the registers in another sub-entity. After the restructuring, the synthesizer did much better. Both run time and area/timing results were improved.

The 40-kgate ECHIP was the hardest of all the chips to synthesize. It is a very memory-cycle intensive design in which all pipe stages in the chip perform an off-chip SRAM access every clock. The SRAM data feeds back up the pipe stages of the chip. It must be run through logic and registered within setup time. The difficulty here is that critical timing paths run through many separate VHDL entities.

The original ECHIP VHDL was well-crafted and straightforward. There are 4 pipe stages in the chip. Each of the stages distributed its READY signal to all earlier stages. The earlier stages would AND together all the READYs from the later stages, and if all later stages were ready, the earlier stage shifts its data out and loads new data. Using this method on four pipe stages, the total number of READYs is: $1 + 2 + 3 + 4 = 10$.

Most of the READY signals depend on late-arriving SRAM read data, so there is very little time for logic to decode the READY signals. The designer had to modify the design in such a way that a minimum of logic was fed with the late-arriving data. After all such timing "surgery" was performed on the design, the net count of READY-related signals increased from 10 to several dozen. The new implementation was much less intuitive to a reader. But it met the tough timing requirements, so it replaced the old straightforward design and went to silicon.

The ROUTER is a 45-kgate, 7-way buffered programmable crosspoint chip with difficult timing constraints. It must register and pass CLINK messages at a rate of one message per clock. There were difficulties similar to the ECHIP's in that critical timing paths were distributed across the chip. However, the solution to the ROUTER's problems was different than the ECHIP's solution.

The ROUTER's original structure includes 7 major blocks, one at each of the 7 crosspoint switch ports (CLINKs). Each of the blocks has a FIFO that buffers incoming messages. At the start of the clock cycle, the source block decodes the message at the output of its FIFO and sends it to the appropriate destination block. The destination block selects one of its 6 input messages and sends it off-chip to the next router. The timing path includes logic in both the source and destination blocks, as well as the off-chip flight path.

The original structure of the ROUTER did not yield adequate performance. The chip had to be substantially

restructured to achieve the desired clock period. The final structure still includes the 7 major blocks, but almost all critical path logic is removed from them. The designer moved the critical logic of all 7 ports into a new block in the middle of the chip. The new block is called the "Router Cloud," and contains over 7 k gates. The synthesizer could then synthesize all the chip's critical paths within one entity, yielding much better timing results. The results improved even further when the designer applied the QCHIP trick of placing the cloud's registers into sub-entities that were synthesized separately.

A final interesting point about the ROUTER's synthesis involves the seven register-based FIFOs at the chip's ports. Less than 10 days before sign-off, our vendor indicated that the chip's clock tree had grown too large. The designer thought over his options, then decided to remove one word from each of the 6 CLINK FIFOs. This procedure took less than one day and resulted in a savings of 270 clock loads. The make-controlled synthesizer produced a netlist in a few hours, and our gate-level system simulation regressions showed the change working perfectly on the first run. VHDL and synthesis allowed this large-scale change to be made at the last minute with practically no risk to the chip's integrity.

The manipulations made to synthesize all four Nsim chips within our target clock period involved several man-months of effort. No manipulations were made at any level lower than the VHDL source code. We considered it bad form to modify the synthesizer's output netlists, and this was never done.

The next step in building a chip is to set up a build process. We use Unix "make" to build chips. Makefiles clearly spell out hierarchical dependencies, and they give a concise summary of how each chip is built from its source. We write all our makefiles by hand, adding comments to explain what is going on. There is one makefile per chip. Once the chip is building and all the synthesizer script files are in place, a designer needs to improve timing by modifying the VHDL source. There are two parts to modifying the VHDL. The first is to concentrate on improving delay within each entity. The second is to redistribute logic among the entities in the hierarchy.

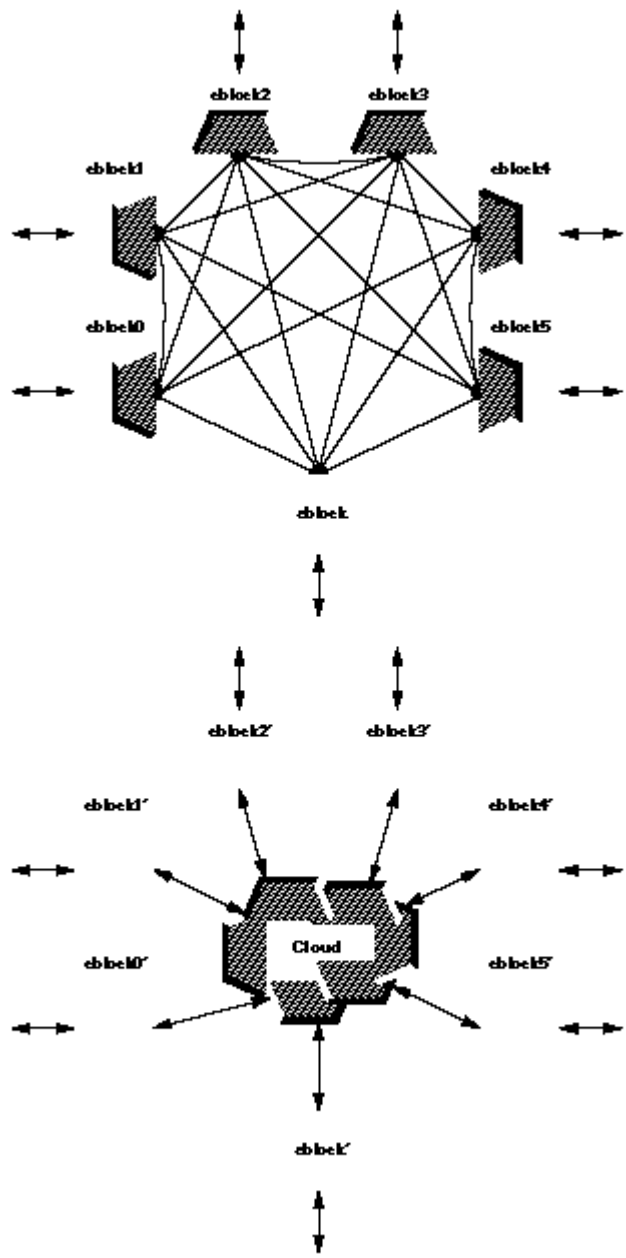
Deciding what changes to make in the code is a design-specific challenge. Here is a partial summary of things to try:

1. Re-structure RTL for better synthesis.
2. Move logic into an earlier pipe stage.
3. Duplicate logic in different entities of the chip to avoid long paths.
4. Add extra pipe stage(s).
5. Move logic into a later pipe stage.
6. Reduce entity sizes by adding sub-hierarchy or new entities.
7. Re-implement or re-design algorithms.

Refer to the sidebar for concrete examples of how these ideas were applied to the Nsim chips.

Notice the methodology implications of this chip building process. What is happening is that engineers design in a top-down fashion, but building the chip is a bottom-up process. The necessities of making timing at the bottom levels of the design reflect back up into the top levels of the hierarchy. In some cases the top levels are profoundly affected. The results are that the final design may not be as readable or structured as simply as a true top-down design would be. This is not always the case. For example, the IOCHIP needed a lot of work to get its timing in shape, but the original design hierarchy was scarcely affected.

Figure 3
Synthesis effects on top-down design



The design group dealt with this issue via the extensive use of regression simulations. After the first cut at a chip, when the design is structured in the simplest fashion, we establish a known-working regression suite. Subsequent design iterations will gradually move the design away from its simple and intuitive structure. The design is periodically run through the regression simulations to ensure that the restructuring does not break the functionality.

Don't forget the hardware The abstract nature of VHDL and the power of synthesis allowed the design team to implement algorithms in hardware that were previously considered not possible.

The key to using VHDL to build chips is to remember that it describes real hardware and to establish a methodology based on this fact. VHDL allows designers to think at a more algorithmic level, but it does not allow them to forget they are designing hardware.

In the future, synthesis technology will certainly make many advances. Engineers will build chips from high-level descriptions without being required to significantly manipulate source code, or perhaps even understand the hardware they are creating. Currently, "donkeying" around with the source code is a key part of chip-building methodology. However, this is a small price to pay for the gains that VHDL and synthesis provide.

Doug Warmke is the manager of hardware development and integration at IKOS Systems Inc. (Cupertino, CA), where he has been for 7 years. He was responsible for the design of the Nsim simulation engine.

Figure 3. The initial design of the router followed the designer's top-down intuition. Synthesis required him to fundamentally redistribute the chip's logic blocks.