A VHDL Environment for ASIC Design

Vijay Nagasamy, Doron Mintz and Carlos Dangelo
LSI Logic Corporation, MS E-199
Milpitas, CA 95035, USA.

Abstract

This paper presents a rule based environment for designing ASIC's using VHDL. The environment supports all levels of design entry from High Level behavior description through RTL and structural descriptions. The design is taken through simulation and synthesis to floorplanning and layout to create a working ASIC. The environment includes a VHDL simulator, VHDL behavioral synthesizer, logic and memory block generators, logic synthesizer & optimizer, synthesis manager and a task manager. Also included are links to timing analyzers, place and route tools and backannotation of delay information to the VHDL description for simulation.

1. Introduction

Hardware systems are getting more complex and becoming increasingly difficult to design without the aid of design tools and CAD environments. More systems today use hierarchical design practices to manage high levels of complexity. Hardware description languages such as VHDL [3], enable a designer to use a unified set of semantics to specify, simulate and design complex digital systems.

VHDL, as a specification and synthesis language offers the capability of describing hardware described at various levels of abstraction [1]; for example behavior, register transfer, logic and structural levels. All these levels of abstraction may be intermixed in a single design. In this paper we use the term behavior, to represent a level of abstraction where the intended behavior of the hardware is described without specifying implementation details. Behavior level is the highest level of abstraction that can be synthesized within our design environment. The next lower level of abstraction is the Register Transfer Level (RTL), which represents descriptions where a system is viewed as a set of interconnected storage elements and functional blocks. The behavior is described as a series of data transfers and transformations between storage elements. The next level is the logic level, where the descriptions consist of a network of gates and flip-flops and the behavior is specified by logic equations. The lowest level of abstraction is the structural level or the netlist, that specifies which hardware components are to be used, and how they are to be interconnected. VHDL permits a continuum of design description styles from the behavior level down to the structural level. In the environment described in this paper, the choice of selecting the appropriate combination of abstraction levels for the design is left to the designer.

The following sections describe a design environment for specifying, simulating and synthesizing digital hardware using VHDL. Techniques used for design space exploration and synthesis at the architectural level are described. This is followed by a brief description of some RTL synthesis techniques. A methodology is presented that enables the user to use various tools in the environment to produce a working ASIC satisfying design goals and constraints. Finally, some results and conclusions are presented.

2. Overview of Design Environment

The main components of the design environment are shown in Figure 1. These include a native VHDL simulator, VHDL analyzer, design data manager, architectural exploration and high level synthesis tool, synthesis manager, logic block synthesizer, memory
compiler, RTL synthesis tool, logic optimizer, and a suite of tools (MDE) that enable a target netlist to be implemented in silicon. The environment is controlled by a task manager, not shown in the figure.

The VHDL analyzer reads in the design files and test bench files and extracts data on the entities and their associated architectures, packages, libraries and test vectors. This information is centrally managed by the design data manager which supplies the information to various tools in the environ-
The architecture exploration tool generates different implementation alternatives for behavioral descriptions and provides estimates of area and speed for each alternative. It also generates VHDL RTL descriptions for the alternate architectures after performing high level synthesis.

The logic block synthesizer (LBS) is a set of specialized tools that synthesizes regular structures such as adders, multipliers, decoders. The memory compiler (MEMCOMP) synthesizes memory blocks such as single or multi-port RAMs. The RTL synthesizer and logic optimizer provide the high level synthesis capability in the environment. The synthesis manager, guided by attributes in the VHDL design description, invokes the appropriate synthesis tools to generate a gate level implementation of the design.

The VHDL Simulator is used to debug and verify design descriptions at the behavior, RTL and structural levels at different stages of the design process.

The entire system is controlled by a Task Manager, a process driven by the design database and top down design rules. The rules include dependencies between the different stages of the design process and how to get from one stage to the next. The database holds information about the current state of the design and the dependencies between the design elements.

These major components are described in the following sections.

3. VHDL Analyzer

The entry point to the design environment is a code analyzer. This is a VHDL analyzer that is used to register the input file into the system database. The database contains information about the name and location of the file and the different design entities that it contains. The code analyzer extracts this information and performs a fully VHDL 1076 compliant syntax check.

The code analyzer is an object oriented program written in C++. It is designed to produce an intermediate form from the input VHDL code. The intermediate form is stored in a file based database. Each file in the database contains the intermediate representation of one design entity. Application programs can be written to access the files in the database. This is done through a Procedural Interface (PI). Access to a design entity in the database is easily achieved by using a unique combination of entity name, architecture name or package name.

The PI allows complete access to the intermediate form. Once a database file is opened the intermediate form can be traversed to examine language constructs. Since the analyzer performs semantic checks as well, all types, variable, sub program, and parameters are fully resolved. The information in the database is therefore ready for applications such as high level synthesis or language translators.

In our system one such application that uses the PI is a subset checker. This application scans the input code and reports about constructs that different applications might not support. For example, some constructs are not synthesizeable and are flagged by this application. Another task specific to our system that this application performs is to verify that names used in the code do not belong to the superset of reserved words of the languages that are supported by the system. In case a VHDL source is translated, for example, to a lower level language such as NDL, such naming conflicts can be resolved by changing the names, however, this could be unfriendly to the user that might inspect the netlist using a schematic viewing program and will find unfamiliar names.

3.1 Support for other languages

Although the primary input language to the system is VHDL, there are translators available to translate other languages to VHDL for the purpose of importing a foreign code to the system. Currently this is only done for structural level designs and we are looking into conversions at higher levels. Currently the system will import NDL and EDIF 2.0 code. A Verilog to VHDL translator is under development and is otherwise available commercially.

4. Design Database Manager

The Design Database Manager (DDBM) is a process that holds the information about the design entities that are registered in the system. The DDBM is accessed using Inter pro-
cess Communication or Unix system calls. There are two access modes. One is an add/update mode in which applications, such as the code analyzer, add or update information about design entities. The other access mode is the request mode in which applications retrieve available information about the design entities. As an example, the available information for an architecture is:

- Architecture name.
- The entity it belongs to.
- Any package that is used within the architecture.
- The file name in which it was declared and its location.
- The date of last update.

The DDBM also accepts complex requests for information such as to provide a list of all the packages that an architecture is using, or all the design entities that are affected by a change of a specific package. This information is necessary for the purpose of efficient recompilation in the case of an update of a package, for example.

An important task performed by the DDBM is to determine which synthesis techniques the user wishes to use for each module in the design. This information is specified by the user via a VHDL attribute associated with the architecture of the entity. For example, a high level synthesis tool is invoked on portions of the design described at the behavior level. Other parts described at the RTL or structural level go through the synthesis manager.

5. Architecture Exploration

The architecture exploration tool, Explorer, generates different implementation alternatives for behavior descriptions and provides estimates for area and speed for each alternative [1,2]. This enables the designer to perform "what-if" analysis at an early stage in the design cycle. Explorer also generates VHDL RTL descriptions for the alternative architectures through high level synthesis. The user can choose one or more RTL architectures for further refinement using the design environment.

High level synthesis is the process of generating an RTL description of a datapath and a control unit satisfying the intended behavior specification. A datapath is generated by assigning each operator in the VHDL description to a state, binding variables to registers, and binding operators to functional units (hardware components like adders, multipliers, alu, etc.). The process of state assignment is called scheduling, and the process of binding to hardware is called connectivity binding. The goal of the high level synthesis tool is to produce a circuit that optimizes resource usage, subject to user constraints. Among the optimizations that are performed during datapath synthesis are: sharing registers among variables, sharing functional units among operators, and sharing interconnects among functional units.

The Explorer compiler builds a control and data flow graph (CDFG) from behavior descriptions of a VHDL process that may contain nested conditional statements and data dependant loops. The data dependencies and control dependencies amongst operations in the process are recorded in the CDFG. The representation makes it easy to identify mutually exclusive operations that belong to different branches of a if or case statement.

Explorer utilizes two types of schedulers to assign operations in the CDFG to states. The first is a force-directed scheduler [9] that is used to schedule operations in the CDFG utilizing a minimum number of functional units within a minimum number of states. The second is a priority based list scheduler [2] that is used when there are known constraints on the number and type of functional units to be used in the design.

The connectivity binder uses clustering and clique partitioning techniques [10] to generate a register transfer level implementation of the architecture. In doing so, it handles appropriate resource bindings for mutually exclusive operations and operations in concurrent data dependant loops. Interconnect area due to multiplexors is also minimized during the process.

Explorer offers two different automatic techniques for exploring the design space for feasible architectures. The unconstrained exploration technique [1] starts with the CDFG and a large enough clock cycle to schedule all operations into a single state. From this starting point, a hierarchical clustering scheme is used to progressively merge operations that can share the same functional unit. The process continues until a minimal set of hard-
ware is found that can implement all the operations in the CDFG. For each set of functional units found by this scheme, the list scheduler is automatically invoked with a large enough clock cycle to get a schedule for the architecture. This exploration technique generates designs ranging from the most parallel to the most serial implementation with other architectures in between as well. The second technique called constrained exploration uses area and clock cycle constraints provided by the user. For each clock cycle a force-directed scheduler is used to obtain a schedule with a minimal number of functional units from the library that implements the operations in the CDFG.

For every architecture generated via the constrained or unconstrained exploration technique, a module and connectivity binder is used to generate a register transfer level netlist incorporating a datapath and an FSM for the control unit.

Timing estimation is an integral part of Explorer. An automatic search technique allows the tool to generate designs that meet the user's clock cycle goals after taking into account the functional unit, multiplexor, control unit and interconnect delays.

A friendly graphical user interface presents the design space explored by the user in the form of a graph. This graph shows relations between various parameters of the design such as gate count, clock cycle time, total cycle time, register area, multiplexor area, and functional unit area. The total area is computed based on a fast floorplan of the design together with estimates for routing [2]. The user can edit architectures displayed on the design space graph to reschedule operations, control amount of resource sharing and explore different bindings of operations to functional units. The user can perform manual and cascaded scheduling directly on the hierarchical CDFG display. Manual scheduling allows the user to move one or more operations from one state to another, provided that the move does not violate any constraints—that is, if the move does not violate some data dependency, lack of resource, or clock cycle requirement. In cascaded scheduling, if an operation has been moved down to a later state, other operations and their resources that depend on it are automatically rescheduled below. Unlike manual scheduling, cascaded scheduling allows an operation to be moved only to a later state.

A register transfer level (RTL) schematic for a scheduled and bound design is available on screen for the user to navigate and get a better understanding of the design generated by Explorer. An integrated database maintains links between the original VHDL description, the CDFG representation and the RTL datapath during the high level synthesis process. This enables the user to point to any functional unit, multiplexor or register in the RTL schematic and obtain information such as gate count, delay etc. The corresponding elements in the CDFG and VHDL source display are simultaneously highlighted. A similar capability is also available for the CDFG display.

6. Synthesis Sub-Environment

The synthesis sub-environment consists of the synthesis manager, logic block synthesizer, memory compiler, RTL synthesizer, logic optimizer and the synthesis manager that is responsible for automatically invoking the appropriate combination of synthesis tools based on the user's VHDL design description.

6.1 RTL Synthesizer and Logic Optimizer

The RTL synthesizer generates a netlist from a VHDL RTL description [8]. Synthesis and optimization steps at this level include state minimization, state encoding, register and latch inferencing and resource sharing. Logic synthesis and optimization techniques are used to optimize the resulting netlist for area and timing in a target technology such as LSI Logic LCA300K. We have used the Synopsys VHDL Compiler and the Synopsys Design Compiler in this environment to perform RTL synthesis and logic optimization for various technologies available at LSI Logic.

6.2 Logic Block Synthesizer

The Logic Block Synthesizer (LBS) is a set of specialized tools that generates regular structures. LBS supports a variety of LSI Logic technologies. The netlists generated by LBS are based on designs crafted by circuit engineers and optimized for a target technology. The quality of the circuits in most cases is
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

LIBRARY LSI_LBS;
USE LSI_LBS.LSIMUXES.ALL; -- Package for LBS Multiplexors

LIBRARY LSI MEMORIES;
USE LSI MEMORIES.MEMORIES.ALL; -- Package for MEMCOMP generated Memories

ENTITY HASH IS
PORT (name1: IN std_logic_vector(7 downto 0);
  clk: IN std_logic;
  -- Some port declarations deleted for clarity
  hash_capacity : OUT std_logic_vector(6 downto 0));
END HASH;

ARCHITECTURE BLOCK_DIAGRAM_VIEW OF HASH IS
-- Component and signal declarations deleted for clarity
BEGIN
  opt <= nil;
  name(15 downto 8) <= name1(7 downto 0);
  name(7 downto 0) <= name0(7 downto 0);
  addr1 <= hf_addr(6 downto 0);

  MUX( opts=>opt, d1=>addr1, d0=>addr2, s=>reset1, z=>addr );

  UO: HASH_FUNC PORT MAP ( name, clk1, rst_n, start1, done1, hf_addr );

  UI: CONTR PORT MAP ( clk, done1, hash_get_n, ........ );

  RAM_LP ( wen=>wen, di=>datai, a=>addr, do=>datao, mem=>mem_content );
END BLOCK_DIAGRAM_VIEW;

Figure 2: VHDL Description containing procedures for MUX and 1 port RAM.

better than those generated by a generic logic synthesis tool. This is a primary reason for its use in the design environment. The circuits include adders, multipliers, ALUs, comparators, incrementers, decrementers, decoders, shift registers, multiplexers, FIFO's and so on. LBS is parametrized based on input and output bit widths and offers a wide variety of choices including drive strength selection and speed vs. area optimization. For example, architectures for adders include an advanced fast parallel adder, carry look ahead adder, carry select adder, carry skip adder and ripple carry adder for bit widths ranging from 1 to 64. Similarly, LBS can generate Wallace tree multipliers, multiplier accumulators and sum of products circuits for bit widths ranging from 2 to 128.

6.3 Memory Compiler

The memory compiler, MEMCOMP, generates memory blocks such as single or multiport RAMs and ROMs used in a design. The memory blocks can be targeted for either low or high power applications. Most digital systems, today, use fair amounts of memory blocks and in this respect MEMCOMP proves to be a valuable resource in the design environment.

6.4 Synthesis Manager

The synthesis manager invokes the appropriate block synthesizer (LBS or MEMCOMP) based on the designers intent expressed in the VHDL source description. Portions of the VHDL description that are targeted for such
procedure RAM_1P (SIGNAL WEN : IN STD_LOGIC;
SIGNAL DI : IN STD_LOGIC_VECTOR;
SIGNAL A : IN STD_LOGIC_VECTOR;
SIGNAL DO : OUT STD_LOGIC_VECTOR;
SIGNAL MEM : INOUT MEMCONT);
attribute lsi_module_generator of RAM_1P : procedure is "MEMCOMP";
attribute lsi_comp_designator of RAM_1P : procedure is "RR";
attribute lsi_module_type of RAM_1P : procedure is "1P_RAM";
attribute lsi_module_sub_type of RAM_1P : procedure is "1P_RAM";
attribute lsi_comp_length of RAM_1P : procedure is "DO";
attribute lsi_aux_comp_dim of RAM_1P : procedure is "MEM";
attribute lsi_comp_technology of RAM_1P : procedure is "LCA100K";

Figure 3: Attributes associated with procedure RAM_1P in package LSI_MEMORIES.

tools are described using concurrent procedure calls (see Figure 2). Such procedures are identified by using special attributes defined in accompanying packages (see Figure 3). The synthesis manager recognizes such procedures from the information put out by the VHDL analyzer and stored in the design database manager. The attributes provide adequate information to the synthesis manager, including the name of the block synthesis tool, circuit type, name and bit width that is required to generate the components. The synthesis manager processes an RTL VHDL description and does the following:

- Creates a new VHDL description by exchanging the procedures targeted for LBS or MEMCOMP with an equivalent component declaration and instantiation.
- Invokes the appropriate block synthesis tool to create a structural VHDL description at the gate level for the component.
- Binds the generated structural description to the appropriate components in the VHDL description.
- Feeds the new VHDL RTL description along with the netlists for LBS generated components to a RTL synthesis and logic optimization tool.
- Puts a "dnot_touch" attribute on LBS generated components so that the logic optimizer does not re-optimize the structure.

The user subsequently interacts and drives the logic optimizer to meet the overall area and delay constraints for the design.

7. Task Manager

The design environment is controlled by a Task Manager, a process driven by the design database and top down design rules. The rules include dependencies between different stages of the design process and how to get from one stage to the next (see Figure 4). The database holds information about the current state of the design and the dependencies
between design elements. The task manager monitors execution of various tools and will not allow the user to proceed if any of the dependency rules are violated. This is yet another check to maintain the integrity of the design within the environment.

8. Delay Backanotation

The delay estimates for synthesized designs are obtained using LSI Logic CAD tools. MDE [4] (Modular Design Environment) and more recently CMDE [5], form a suite of CAD tools that are used to analyze gate level designs before sign-off to layout. Some of the commonly used tools include:

- Gate level simulation (LSIM)
- Pre-layout delay estimation
- Post-layout delay computation
- Static timing analysis
- Floorplanning

The VHDL design environment is integrated with LSI Logic’s CAD tools in order to provide delay estimates for gate level simulation. Software available within the environment reads in the output of the delay estimator and backannotates it into gate level VHDL models using the generic construct support in VHDL. This enables the user to perform a full timing and functional simulation of the design at the gate level.

9. VHDL Simulator

The VHDL simulator assists the user in verifying the design at different stages in the design process. During the initial specification stage it is used to verify whether the input VHDL description satisfies the intended functionality as set forth in the requirements. At the RT level, it is used to simulate the design to verify the correctness of manual transformations or the output of the high level synthesis tool. Lastly it is used to help verify the gate level implementation after logic optimization and delay backannotation.

To facilitate simulation, models at different levels of abstraction are provided within the environment. For functional simulation at the behavior and RT levels, parameterized models for blocks generated by LBS and MEM-MCOMP are provided to the user in the form of VHDL packages. For gate level simulation, VHDL models for the macrocells in the technology library are available to the user.

The environment also facilitates the creation of test benches for entities in the design. This is done in the following way:

- Automatically creating and managing VHDL files associated with the test benches for entities in the design.
- Generating templates for entities where the user can specify the test patterns in VHDL.

The same test vectors can be used, if desired, to simulate and verify the design at the behavior, RTL and gate levels. Our environment uses the Vantage VHDL simulator.

10. ASIC Design Methodology

Although we do advocate a top-down design methodology to manage complexity, for the sake of flexibility, the environment does not impose any particular approach on the user [6]. It can be configured to allow either a top-down, bottom-up or middle-out design approach depending on the project and what the designers are comfortable with.

Figure 5 illustrates a typical flow chart for designing an ASIC using a top down approach. The designer starts with a set of VHDL files, each file containing an entity/architecture pair for a module in the design. Each entity can in turn use other entities as components, thereby creating a design hierarchy. One of these VHDL files contains the description of the top-level entity for the design. Registering all the VHDL files with the environment, subsequently frees the user from file management.

The user invokes the code analyzer on the top-level entity to make sure that all the design files are fully VHDL 1076 compliant. At this stage the code analyzer also performs subset checking to ensure that the entities to be synthesized conform to the synthesizable VHDL subset. This saves the designer valuable time and effort at the synthesis stage.

After all the relevant design files have been analyzed, the user sets the current entity to the top-level module in the design and invokes the VHDL simulator. The test bench files and templates are automatically created for this entity and the user enters the test vectors for simulation. Since the design database contains information on the design hierarchy, the task manager prepares the design for sim-
ulation by traversing through all the entities, and packages in the design in the correct order. After debugging and verifying simulation results the user starts the synthesis cycle. At this stage, if the design contains any behavior descriptions, the architecture exploration and high level synthesis tools are automatically invoked. For each entity with a behavior description, the user chooses the appropriate RTL implementation after doing
a “what-if” analysis and examining the area-speed trade-off curve generated by the tool. The user is allowed to manually fine tune the VHDL RTL description in case the implementation generated by the high level synthesis tool is not competitive enough with respect to manual design.

The synthesis manager is automatically invoked when the user starts synthesis. At this point all the synthesizeable entities are described at the RT level or the structural level. The user interacts with the RTL synthesizer and logic optimizer to create a netlist in a target technology that meets the desired area and speed constraints. If not, the source description is modified and the design process is repeated until the goals are met. The design environment incrementally re-analyzes and re-synthesizes entities that are affected by changes made by the user to ensure that all synthesized results are up to date with the source descriptions.

At the end of this synthesis cycle, the user once again simulates the design at the gate level and invokes a sign-off procedure if the simulation results are satisfactory. The design environment is tightly integrated with a proprietary set of CAD tools (MDE) that allows the user to further analyze the design, simulate if necessary using LSI Logic’s gate level simulator, and sign it off for layout.

11. Results and Conclusions

This design environment has been successfully used both internally and by LSI Logic customers to implement ASIC’s starting with VHDL descriptions.

An early implementation of this environment was used by one of LSI Logic’s customers. The environment enabled a single designer to implement a 50K gate DSP ASIC within a couple of months. Another successful use of this environment was the design of a cache controller megafunction. The design effort consisted of writing and debugging VHDL RTL descriptions and synthesizing the circuit to meet timing constraints. The resulting 30K gate synthesized design (excluding memory) was completed in 2 months, fabricated, and it worked right the first time. The design activity did however leverage test vectors developed during a parallel manual design effort. The synthesized design was within 15% of the manually designed circuit.

More recently, the environment has been used internally to design a part of the CCITT H.261 DSP chip set and the MPEG video decoder chip. Details about the H.261 design and the methodology can be found in [7].

ACKNOWLEDGMENT

We thank Ahsan Bootesaz, Seeranga Rajan, Allen Yeung, Gary Chan, Jeffrey Vo, Julius Miao, Vijayanand Ponukumati, Richard Deeley, Neerav Berry, Rajiv Dutta and Pravin Madhani for their invaluable contribution towards developing the design environment.

REFERENCES


