Debugging of Behavioral VHDL Specifications by Source Level Emulation

Gernot Koch¹, Udo Kebschull¹, Wolfgang Rosenstiel²

¹Forschungszentrum Informatik (FZI), Haid-und-Neu-Straße 10-14, D 76131 Karlsruhe, Germany
²FZI and University of Tübingen, Sand 13, D 72076 Tübingen, Germany

Abstract

In this paper we present an approach to accelerate the validation speed of behavioral VHDL system specifications through the use of hardware emulation*.

1: Introduction

Validation in hardware/software codesign mainly has to solve two problems. First, appropriate means for software and hardware validation are needed and second, these means must be combined for integrated system validation. Since validation methods for software are well known the main effort is spent on hardware validation methods and the integration of both techniques.

The main approaches for hardware validation are simulation, emulation [1], [2], [3] and formal verification [4] today. Simulation provides all abstraction levels from the layout level up to the behavioral level. The highest level at which hardware emulation together with hardware debugging is currently available, is the RT level. Quickturn’s HDL-ICE system allows source level debugging of Verilog specifications at the RT level with emulator hardware support [5]. Formal verification of behavioral circuit specifications has up to now only limited relevance, since the specifications which can be processed are only small and can only use a very simple subset of VHDL.

All these validation/verification methods have substantial disadvantages with respect to their applicability in hardware/software codesign: The low level validation methods are not really useful, since hardware/software codesign aims at automatic hardware and software design starting from behavioral specifications. This would be like writing programs in C++ code and debugging the generated assembly code. This even holds for debugging at the RT level since hw/sw codesign usually implies high level synthesis for the hardware parts. Thus, the RT description of the hardware is produced automatically and may therefore be very difficult to interpret for a designer.

Though simulation can operate at the algorithmic level there are some problems with it, too. First, simulation is very time consuming and thus it may be impossible to simulate bigger systems like systems consisting of hardware and software parts. Second, simulation is too slow to bring simulated components to work with other existing components. Thus it is necessary to model the outside world within the simulator which requires at least an enormous effort.

In this paper we propose a method to overcome these disadvantages by combining behavioral simulation with hardware emulation. In the next chapter we highlight the features of our SLE, and discuss the principal problems and their solution related to the realization of SLE. In chapter 3 we introduce the hardware related aspects of our method including a short overview on our underlying emulation architecture called Weaver. The paper is concluded by a small example showing the applicability of our approach, and a summary.

2: Source level emulation

2.1: Highlights of SLE

When we speak of source level debugging (SLD), we mainly think of software SLD which is a common technique. For hardware specifications, this is only possible as a purely software based behavioral simulation today. We introduce a method which allows a similar symbolic debugging of a running hardware. This includes the examination of variables, the setting of breakpoints, and single step operation. All this is possible with the application running as a real hardware implementation on a hardware emulator. This offers some possibilities not given by debugging methods operating just in software.

First of all we do not need to capture the environment of the application in a simulator. We just connect the emu-

* Work supported by the ESPRIT 8135 project (COBRA) and Siemens AG Munich
lator to the environment of the application. There is no need to write (often enormous) simulation environments in VHDL which is even better since these simulation environments are at least as fault-prone as the application itself.

The ability to set breakpoints and to single step the hardware allows to trace reactions to external events. We just set a breakpoint at the beginning of the reaction and if it occurs, we step through.

In this context we can also do an execution time analysis by counting clock cycles which are needed by the reaction. This is only possible through the fact that we have a synthesized hardware underlying the behavioral SLE. No behavioral simulation can be used for this.

2.2: High level synthesis (HLS)

The SLE of behavioral VHDL specifications is related a lot to the HLS since HLS is the technique used to synthesize the specifications. Therefore, it is appropriate to summarize the basics of HLS. The HLS is performed upon a flow graph (fg), to which the VHDL specification is translated. This flowgraph mainly contains operations like +, *, assignment etc. The connection of these operations corresponds to the sequential character of the behavioral specification. The first step performed by the HLS is the transformation of the fg into a data flow graph (dfg) where the connection of operations corresponds to the given dataflow. Then an allocation[6] is performed which chooses a list of component types available for the circuit. Scheduling[7] assigns a component type and a timestamp to each operation of the dfg. Therefore, the controller graph (cfg) is constructed after scheduling. The last task of HLS is the assignment[8] task. This task assigns components to each operation according to the selected component types. Thereby it tries to minimize the size of the circuit by sharing components for different operations if these are not active in the same controller state. The same is done with nets assigned to registers.

2.3: Problems of SLE

Although we speak of hardware supported SLD it is obvious that the semantics of SLE and software SLD are quite different. In software SLD, the lines of code determine the sequential order of the operations while in hardware SLE the order of the lines is relatively independent of the temporal order of the execution of operations since scheduling regards the data dependencies, not the textual order. Thus, single stepping means stepping single clock cycles, not single lines of code. Also in a circuit there are no variables, but only nets. Therefore, a variable can only be examined if it is represented by a living net at a certain time.

The HLS maps operations to components in the circuit. Thereby it can happen that an operation corresponds to several components as well as that several operations correspond to the same component.

Variables may correspond to different registers at different controller states or they may even be not represented by a register at some state. Also the content of a register may not correspond to any variable at some state.

Components may need more than one clock cycle for execution and there may be synchronous and combinatorial components. Even the inputs of a component may be read in different clock cycles while the component executes. So, if the value of an input is requested we need to find out, at which time this input is valid, and when the breakpoint must be set. This is complicated by components with an input-dependent runtime.

When the correspondence between the behavioral specification and the synthesized RT level is known, a special synthesis step is needed, which does not destroy the structure for debugging through retiming optimizations etc. Also there must be some additional hardware in a circuit that is needed for the retrieval of register values from outside and for setting breakpoints and interrupting the hardware when a breakpoint is reached.

Last but not least we need a target hardware where the circuit is mapped on, and which can be connected to an environment, and to a host. This hardware must allow the access to the requested debugging structures inside the circuit, it must be programmable, and it must provide enough capacity for large circuits and complete systems including microprocessors, RAM etc.

2.4: Solutions

In this subchapter we discuss the solutions of the problems which are related to the HLS. The more hardware related solutions are described in the next chapter.

Operation - component relationship: This correlation is used to set breakpoints at particular fg operations. Therefore, the corresponding components are identified together with the FSM state at which these components perform the requested operation. The FSM state is known from scheduling and the component is known from assignment. Now, for each component there may exist several operations, which are performed by this component. The HLS task responsible for this is the assignment task. Since we know the FSM state from scheduling, we just have to keep this information through assignment to identify for each pair of controller state and component the unique corresponding operation.

Also, through HLS a single operation may be mapped to several components which perform it. Figure 1 shows such a case. The single operation ‘+’ introduces three components. Two of these just adapt the bit size of the operands and the result of the component ‘Add’. In other cases an operation can force multiplexers into the circuit. To solve this problem, we divide the set of components into two subsets, the ‘performing components’ and the ‘put-through components’. The performing components are those which perform a logic function on one or more operands and produce an output. These components compute something like adders, comparators, etc. The put-through
components just copy selected bits of their operands to the output like transfer components or multiplexers. With this definition each set of components which maps a single operation contains exactly one performing operation, which we are interested in for debugging.

**Variable - register relationship:** Here the problems are more serious than with operation-component mapping because HLS operates on a dfg which does not have constructs like variables, but shows only the data flow between components. The assignment $x := y;$ may clarify this. The current value of $y$ is carried by a net which is driven by the component that produced this value. The future value of $x$ will be carried by a net which is an input of the component that consumes this value. Actually the output of the first component is directly connected to the input of the second component in the dfg. It does not make sense to search for a correspondence between a variable and this net. But it can be clearly stated that in this state this net carries the value of the identifier $x$ at this particular textual position. The same holds for the identifier $y$. The question must be: Given a textual position of a variable identifier, which net represents this instance of this variable at which controller state and to which register (if any) was this net assigned during assignment? The textual position of the variable identifier indicates, to which operation it belongs and whether it is an input or an output. We identify the corresponding pair (component, FSM state) to this operation and the corresponding net to the requested identifier. There are several cases for this net as Figure 2 shows. These depend on how the net is connected to the next register, and whether it is an input net or an output net of a component. The case to which a net belongs mainly determines the effort needed to retrieve its value. In case 1) the net representing the requested variable identifier (drawn with a thick line) is an input net of the component and it is connected to a register via several put through components. Only the register has to be read out to retrieve the value of this net. In case 2) it is an output net of the component which is also connected to a register via some put through components. The corresponding value can be retrieved by reading out the register in the next clock cycle. In case 3) the input net is connected to another performing component. It is impossible to retrieve the value of the net directly. In this case, the input nets of the other component have to be backtracked recursively until registers are reached at all inputs of the resulting combinational circuit. The value of the requested net must be computed from the values of these registers by the debugging software then. Therefore, the debugging software maintains a model of the circuit and the components. In case 4) where the requested output net is connected to another performing component, the input nets of the component of which the output is requested must be backtracked analogously to case 3). Then, the value of the requested output net must also be computed by software. The time overhead for the software computation does not affect the running hardware since such computations occur only at breakpoints when the hardware is interrupted. Thus, only the user has to wait on this computation. But it requires a library where a behavioral model of each component is located. The library may need a considerable amount of disk and RAM space on the host.

**Synchronous components consuming several clock cycles:** In the last section setting breakpoints has implicitly been described as determining the FSM state at which an operation is executed and interrupting the clock when this state is reached. This has to be refined for components needing several clock cycles for execution. In this case some inputs may be read at different FSM states and the output may be transferred some cycles later. Therefore, the point of interest is actually the state at which an input or output port of a component is active. This is determined from the component library, where all components are listed with their behavior and their timing specification. There to each port the clock cycle relative to the start time of the component is given, at which the port is active.

### 3: Hardware aspects

**Changes to the RT level description:** The changes to the synthesized circuit mainly come from the need to read
out the registers and to determine the current FSM state and the comparator outputs. These changes are most easily introduced at the generation of the RT level VHDL specification. All registers inserted by the HLS are exchanged with parallel loadable shift registers, to provide the ability to read out the registers. These are interconnected to a full scan path of all registers in the circuit.

State and comparator value determination is done with the insertion of additional comparators into the data path. The inputs of these comparators are the output of a register and a set of comparator outputs or the state registers of the controller. These input registers of the additional comparators are also serially connected via a scan path which allows to write values into them. As Figure 3 shows, the additional comparators are connected to an AND gate. The output of this gate indicates that a breakpoint is reached. The breakpoint indication is input to a debugging controller that interrupts the clock and propagates the break to the host. The additional hardware may result in longer combinational paths in the circuit and may thus lead to a slower clock than the original circuit would allow. Since the outputs of the comparators of the circuit must be regarded for breakpoint detection, the indication can only take place at the end of a clock cycle to guarantee that the comparator outputs are stable and valid. In a fully digital circuit this can only be done by clocking the debugging controller faster than the rest of the circuit, e.g. if the circuit clock speed is half of the debugging clock speed then the debugging controller evaluates the "breakpoint reached"-signal in the second half of the circuit clock cycle.

RT synthesis: Though the debugging structures are included at the RT level in the circuit to be independent of logic synthesis we stay on the safe side by just avoiding logic synthesis. The idea is to take the data path of the circuit as a netlist and just to insert the components as macros one by one in the Xilinx netlist format. The result is a Xilinx netlist of the data path which can be processed by our FPGA partitioning software [10], followed by the Xilinx place and route tools. This leads to a different circuit for debugging than for the final implementation since logic synthesis will be used for the final implementation. Therefore, we cannot validate the exact timing of the implementation but only functional correctness and timing in terms of clock cycles. The exact timing must be addressed by low level simulation.

Target Architecture: We use our own modular emulator hardware called Weaver [9]. It uses a hardwired regular interconnection scheme on the base modules and global busses for module interconnection. The base module of Weaver carries four Xilinx FPGAs for the configurable logic. On each side of the quadratic base module a connector with 90 pins is located. Each FPGA is connected to one of these connectors. Also every FPGA has a 75 bit link to each of two neighbors. A Control Unit is located on the base module which does the programming and readback of the particular FPGAs. An I/O module provides a con-
A connection to a host via the parallel interface of the host. With this module the host works as I/O preprocessor which offers an interface for interaction to the user.

A RAM module with 4 MB static RAM can be plugged in for the storage of global or local data. If it is in a bus module, several modules have access to it. If it is connected directly to a base module, this module has exclusive access to the memory.

The bus module makes it possible to plug modules together in a bus oriented way. With the possibility to have a bus on each side of the base module this architecture can be used to build complex systems with arbitrary structure.

Standard processors are integrated via additional modules. Figure 4 shows a possible configuration of Weaver.

With this architecture we can build a hardware which is especially suitable for our debugging method. Figure 5 shows a configuration consisting of two basic modules which are connected to a host. These basic modules carry the ASIC including the additional debugging hardware. The interaction with the debugging software is given through the I/O module. The five unused connectors may be used for communication with the target system of the circuit.

4: Example

As example we have chosen Euklid's gcd algorithm. The VHDL code is shown in Figure 6. It is a very simple algorithm but it shows the principles of our approach. Figure 7 shows the intermediate flowgraph, the generated circuit and the controller graph of the gcd algorithm. The relationship between flowgraph and VHDL code is obvious and needs no explanation. The thin lines between fg and circuit show the relating component to each operation. Both subtractions are executed by the same component, and both comparisons are performed by the same comparator. Thus the while-condition and the if-condition of the VHDL source are evaluated in parallel at the same time. The conditional transitions in the controller graph are marked with their conditions. Table 1 shows the full relationship between operations, components and controller states. The operations and controller states are numbered according to their numbers in Figure 7, and the components are also named according to Figure 7. The column Mealy/Moore is used to describe the behavior of the controller output which controls the component. For synchronous components this is of type Mealy, and for others it is of type Moore. If it is Mealy then the component belongs to a state transition rather than to a state. In this case the outputs of the comparator determine together with the controller state whether the component is executed or not.

For instance, operation No. 1 is executed by the component Read 1 in the controller states 1 and 2. But it is controlled by a mealy output of the fsm and therefore also the comparator outputs are relevant to decide whether the component is actually executed or not. In state 1 it is executed anyways, since there is no relevant condition. But in state 2 it is only executed if the following state is also 2, that is if the condition X1=X2 is true.

Table 2 shows how the multiplexers are controlled in each controller state. Some of the multiplexers are controlled by controller outputs with mealy behavior, others with moore behavior. Those with mealy behavior have a state transition after the controlling value in the table. Also these may have different values for different transitions in the same state. For instance, in state 2 the multiplexer M3 puts its first input through (control = 0) if the following state is state 3, and the second input is put through (control = 1) if the following state is state 4. In our example we request the input variable X1 of the operation No. 5. We can see in Table 1, that this operation is executed by
the component sub and that the clock must be interrupted in controller state No. 2. By evaluating the multiplexer states given in Table 2 we see that the requested input of this component is connected directly via a put-through component to register Reg 1. Thus, this register holds the requested value. We can retrieve by the same means that at the same time Reg 2 holds the value of X2 as the other input variable of operation No. 5. The values correspond also to the inputs of the operations 3 and 4. The value of Reg 3 cannot be assigned to a variable since the history of the circuit is not known.

5: Summary

In this paper we gave an overview over a system for hardware supported source level debugging of behavioral VHDL specifications. The presented approach offers a new validation technique by combining emulation with behavioral simulation. Emulation is improved via the assignment of circuit elements and the VHDL source while simulation is improved through the fact that it is not just based on a software model but on a real synthesized running hardware. The approach allows a "behavioral simulation"-like hardware debugging with the speed of emulated hardware. By connecting the emulator to an existing system the hardware can be debugged at the behavioral level without the need to write enormous simulation environments. Because we operate at the behavioral level the approach is especially suitable for hardware/software codesign.

6: References