Automatic Generation of Breakpoint Hardware for Silicon Debug

Bart Vermeulen
Philips Research Laboratories
Eindhoven, The Netherlands
bart.vermeulen@philips.com

Mohammad Z. Urfianto
Royal Institute of Technology
Kista, Sweden

Sandeep K. Goel
Philips Research Laboratories
Eindhoven, The Netherlands

ABSTRACT

Scan-based silicon debug is a technique that can be used to help find design errors in prototype silicon more quickly. One part of this technique involves the inclusion of breakpoint modules during the design stage of the chip. This paper focuses on an innovative approach to automatically generate breakpoint modules by means of a breakpoint description language. This language is illustrated using an example, and experimental results are presented that show the efficiency and effectiveness of this new method for generating breakpoint hardware.

Categories and Subject Descriptors

B.8 [**Hardware**]: Performance and Reliability—*Performance Analysis and Design Aids*

General Terms

Design

1. SCAN-BASED SILICON DEBUG

With the integration of complete systems on a single chip, it becomes increasingly difficult to find all design errors prior to first tape-out. The industry average for first-time-right silicon is well below 50%[6]. If we can no longer guarantee that all design errors are detected prior to first tape-out, it becomes increasingly important to be able to quickly analyze and find design errors in *actual* silicon[3].

In the mid-nineties scan-based debug was introduced as a method to speed up silicon debug by increasing the internal observability [2][3][4][7][8]. During a scan-based silicon debug session, a chip is run functionally in its intended application environment. At a certain point, an on-chip breakpoint module detects the occurrence of an important internal event and stops the chip by gating all functional clocks. Using software tools, the debug engineer then reads out the entire state of the chip. This state is compared to the expected correct state, obtained from a simulation or emulation model. Repeatedly applying this technique, stopping at different points during the chip's execution, can lead to a quick localization of a design error in the actual silicon, which can then be corrected in a redesign.

Widespread adoption of this scan-based debug methodology by the design community relies heavily on good tool support to efficiently add debug IP modules to a given design. The difficulty with automating design-for-debug lies in the fact that the function of the required breakpoint module is dependent on the function of the design. Often a custom breakpoint module is required that cannot be easily shared among different types of cores and different SoCs.

This paper focuses on a novel approach to automatically generate breakpoint modules by means of a concise Breakpoint Description Language (BDL). Using this language a software tool can automatically generate the required implementation code for the corresponding breakpoint module. Although there is prior work on the *use* of *design-specific* breakpoint modules to facilitate software debug, to the best of our knowledge there is no prior work on the definition of a *generic* breakpoint description language and the subsequent *automatic generation* of the corresponding debug module.

2. PRIOR WORK

In order to derive a list of breakpoint requirements, we investigated existing implementations of breakpoint IP modules. Because many breakpoint module implementations exist, we considered a representative subset covering the breakpoint support offered in the ARM[1] and MIPS[5] CPU cores, and the Philips CPA[7], PNX8525[8] and Sorcery SoCs. From this prior work we obtained a list of in total five abstract breakpoint requirements.

1. **Signal Observation**
   Monitors are often used at various locations in a design to observe important internal signals and busses. When the value of a signal or a bus matches a predefined value, the monitor activates its output. Our BDL allows six types of unsigned and signed comparisons ('==', '<', '<=', '>', '>=', and '!='). and also allows to detect signal changes using its posedge and negedge functions.

2. **Event Combination**
   Event combination is used when multiple events need to occur simultaneously in a certain Boolean combination to signal an important point in the chip's execution.

DAC 2004 June 7–11,2004,San Diego,California,USA.

3. **Event Sequencing**
   Event sequencing is used when multiple events need to occur in a pre-defined order to signal an important point in the chip's execution.

4. **Event Counting**
   Event counting is used to determine whether a certain event has occurred a pre-defined number of times. Count intervals are allowed to be: single-valued, double-valued (e.g. [2,5]), and open-ended (e.g. [2,-] and [-,2]).

5. **Programmability**
   We often require flexibility and programmability of the breakpoint to allow a single breakpoint module to generate a breakpoint at different points during the chip's execution. This enables us to create state dumps both before and after an error is observed in a design. We use the serial IEEE 1149.1 TAP interface as our debug interface. During a debug session, the debug engineer can program reference values via the JTAG interface to generate a breakpoint at a specific point in time.

From this examination we learned that there was not one *common* breakpoint module implementation that fits the needs of all possible cores and SoCs. Too many variables in the design influence the actual breakpoint module implementation. As a result, we developed a new Breakpoint Hardware Generation Flow, which we discuss in the next section.

3. BREAKPOINT GENERATION FLOW

Our new flow is centered around a designer's specification of the required breakpoint(s) and is shown in Figure 1

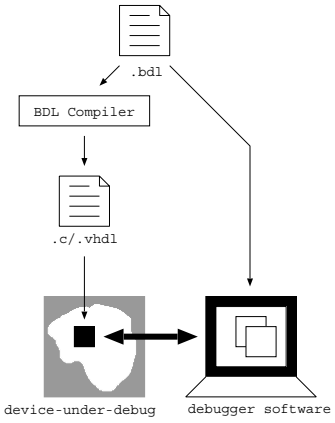

Figure 1: Breakpoint generation flow

Shown at the top is the designer's input, a concise description of the required breakpoint. This description is first used to automatically generate a breakpoint module implementation using a BDL compiler. In our first version this compiler generates VHDL RTL code. The compiler takes care of all the implementation-specific details, such as using the correct keywords to define the module's interface etc. This implementation can then be readily instantiated in a core or SOC design. When the core or SOC has to be debugged, the BDL description is used by the debugger software to automatically generate a corresponding graphical user interface, which is very easy to use by a debug engineer.

In summary, we set the following four goals for our language.

1. BDL needs to serve as a concise, high-level representation of a breakpoint. The level of this representation needs to be higher than, for example, Verilog or VHDL RTL, as we want to concentrate on describing the breakpoint itself and not how the breakpoint module for this breakpoint can be implemented.

2. BDL needs to capture the breakpoint requirements for both existing and future cores and SoCs.

3. BDL needs to allow rapid development of breakpoint modules, comprising:

   - Breakpoint resource analysis & optimization,
   - Breakpoint module generation,
   - Breakpoint module instantiation, and
   - Breakpoint module verification (at various levels, before and after instantiation).

4. BDL needs to simplify the programming of the instantiated breakpoint module during an actual silicon debug session.

4. BDL COMPILER

To automatically generate the required implementation code from a breakpoint description in BDL, we developed a BDL compiler. The steps that are executed by this compiler are explained below.

- A *scanner* is used to perform a lexical analysis of the BDL source code and to transform the stream of characters into a stream of tokens. The parser is then used to perform a syntactic analysis and generate a so-called Abstract Syntax Tree (AST), which is a data structure representing a valid sequence of tokens as defined by the BDL grammar.

- A *symbol-checking phase* performs a semantic check on the AST. It creates a symbol table of the identifiers used, and ensures that each identifier is only declared once, and is not a reserved RTL-code keyword.

- A *type-checking phase* is used to check the correctness of the assignments in the BDL description. This includes checking for the proper use of identifiers, operands, and operators in an assignment.

- *Sequence Resolving* is used to process and order the sequence event assignments.

- *Resource analysis* determines the required event generators and program registers.

- The *backend* generates the required implemtation code for the breakpoint module, based on the information gathered during the resource analysis stage. In our prototype, this backend outputs VHDL RTL code. The compiler can be easily extended to output code for other implementation languages as well, for example SystemC or Verilog RTL code.

5. BREAKPOINT EXAMPLE

In this section, we will take a realistic breakpoint scenario and show which steps need to be taken to obtain a RTL implementation using our BDL. Consider the following informal breakpoint scenario specification:

> A start_a signal normally initiates a receiver. If within the next 20 clock cycles:
>
> - It detects the data value 0x0020 then immediate flag a breakpoint condition;
> - It detects the data value 0xFFFF and this data came from an address above 0x1000, then break after 5 clock cycles;
> - Otherwise go to the first step and wait for the start_a signal to be asserted again.

The corresponding translation of this textual description in BDL is given below (line numbers are included for clarity).

```
 1  breakpointmodule dac04;
 2    input start_a;
 3    input addressbus[15:0];
 4    input databus[15:0];
 5    output stop;
 6    matchevent me0,me1,me2,me3,me4,me6;
 7    combineevent ce5;
 8    sequenceevent stop;
 9    me0 := posedge(start_a);
10    me1 := (1==1),[20];
11    me2 := (databus==0x0020);
12    me3 := (databus==0xFFFF);
13    me4 := (addressbus > 0x1000);
14    ce5 := me3 && me4;
15    me6 := (1==1),[5];
16    stop := me0 -> ( me2 | ce5 -> me6 |
17                     me1 -> stop );
18  endbreakmodule;
```

The compiler will scan, parse and check this description. The event definitions on lines 10 and 15 use expressions that are true for each clock cycle. In combination with the counter interval (specified using the square brackets), they allow the expiration of an interval, measured in a number of clock cycles, to be detected. On lines 16 and 17 we specify the *stop* event that represents the sequence the breakpoint module has to go through before signalling a breakpoint. The compiler will construct a graph for each sequence definition in the specification and merge them, if possible, in the sequence resolve step. The event definition *stop* is translated into the graph representation shown in Figure 2.

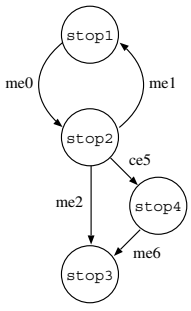

Figure 2: *stop* sequence graph

This graph corresponds to the state transition diagram of the FSM inside the breakpoint module implementation, as generated by the compiler. For this example, the internal state machine starts upon reset, in the state 'stop1'. If it detects the event 'me0', it will transition to state 'stop2'. In that state, several possibilities exist. If event 'me2' is detected, it will directly go to state 'stop3'. State 'stop3' is the final state of this state machine (because it has no outgoing edges). When this state is reached the output of the breakpoint module, the signal 'stop', is asserted. However the state machine goes back to state 'stop1', if in state 'stop2' event 'me1' is detected. There it will once again wait for event 'me0' to occur. If the combination of events 'me3' and 'me4' (combined in event 'ce5') is detected while in state 'stop2', the state machine is taken to state 'stop4'. There it will wait for event 'me6' to occur, after which it will move to state 'stop3' and assert the breakpoint module output 'stop'.

For each state in the diagram the labels of the outgoing edges indicate the breakpoint resources that are required to inform the state machine whether to remain in that state or move to another state. The compiler will construct an event detector for each outgoing edge. For example the event detector that is used by the state 'stop0' is capable of detecting event 'me0' (which is the label on the outgoing edge of that state). In each state, the FSM will only enable those event detectors corresponding to its outgoing edges. This means that the event generator of event 'me6' is not enabled (i.e. in reset), until the state machine is in state 'stop4'.

Based on the information gathered from the state diagrams, and the event definitions, the compiler backend generates the appropriate implementation code. For the example presented, the compiler's implementation is shown in Figure 3. Here the label 'ED' stands for 'Event Detector'. Although we specified all values in the BDL as fixed values, the program register from the generic breakpoint module template is still included to provide the option to enable the

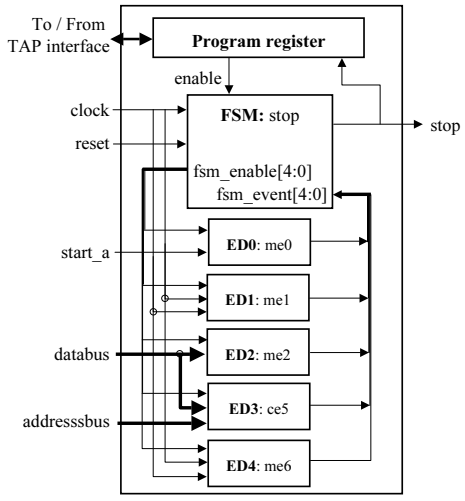

Figure 3: Compiler implementation of BDL example.

6. EXPERIMENTAL RESULTS

To evaluate the effectiveness of BDL and our compiler, we re-implemented the breakpoint modules mentioned in Section 2. We first expressed the required breakpoint behavior

in BDL, and used the BDL compiler to generate the corresponding VHDL RTL code for each breakpoint module. The specification in BDL took between 2 to 30 minutes, while for all cases the execution time of the BDL compiler was well below 1 second.

We then compared the area size of the synthesized original with that of the synthesized BDL version. Unfortunately we do not have access to the original RTL code of the ARM and MIPS breakpoint components, so we are unable to compare against these originals. The results we obtained are shown in Figure 4.

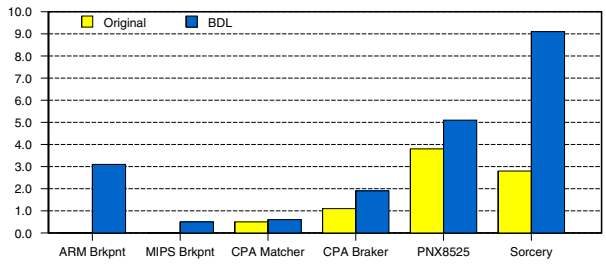

Figure 4: Breakpoint hardware area comparison.
(Area numbers in 10.000 μm^2)

The numerical results of the last four designs are listed in Table 1. Please note that both the original and BDL versions were synthesized using the same 0.18 μm CMOS technology libraries and the same synthesis scripts.

Table 1: Silicon area for original and BDL versions.

Design	Original	BDL	% DfD area increase	% SoC area increase
CPA matcher	5865	7344	+25.20%	+0.0033%
CPA brake	12345	17895	+44.96%	+0.0124%
PNX8525	37548	52346	+39.41%	+0.0164%
Sorcery	27631	91918	+232.66%	+0.0583%

The area of the generated breakpoint modules is - on average - roughly 40% larger than their original implementation. The exception to this is the Sorcery SoC, which is more than 3 times as large. After analyzing the four cases, we discovered that the breakpoint modules that are generated do not share common resources, i.e. comparators, counters, etc. This can also be observed for the example in Figure 3, where the interval detector for 'me1' and 'me6' are not combined into one detector, even though these two detectors will never be active in the same state of the FSM. Likewise, the generated breakpoint module for the Sorcery SoC contains no less than 42 counters, where in the original design 2 counters sufficed. We are investigating a resource optimization step in the compiler to automatically detect the possibility to use a single detector for multiple events, thereby reducing the required area. Regardless of the resource optimization step, to properly appreciate the area numbers reported in Table 1, we need to consider the total area of the SoCs these breakpoint modules are used in, as is shown in the fifth column of Table 1. For all cases, the actual breakpoint module itself occupied far less than 0.1% of total chip area, and therefore this area increase can, for all intent and purposes, still be considered negligible.

7. CONCLUSION AND FUTURE WORK

In this paper we have presented a method to automatically generate the breakpoint modules for silicon debug by means of a Breakpoint Description Language (BDL). The main emphasis of the use of BDL in this paper has been on the rapid development of breakpoint modules; that is to capture the specification of a breakpoint module, and then by means of a BDL compiler generate its hardware implementation. We conclude that BDL can indeed serve as a concise representation of our design-specific breakpoints. Writing down breakpoint requirements in BDL takes very little time. An added advantage is that BDL allows different back-ends to be used (e.g. VHDL RTL or SystemC). The fast compiler allows the debug engineer to evaluate different breakpoint module implementations, and to trade-off the breakpoint functionality with the required area.

We proved that BDL succeeds in capturing the breakpoint requirements for existing SoCs, and are confident that for the next generation of SoCs, BDL can be readily used to reduce the design effort for custom breakpoint module implementations.

Meanwhile we continue this work to automatically verify a breakpoint module implementation and instantiation in a core or SoC. We are also extending our debugger software tools to read in BDL descriptions and configure breakpoint options in a user-friendly graphical user interface. We want to encourage other companies to participate with us in a discussion on how to best automate and standardize design-for-debug and silicon debug for complex digital SoCs.

8. ACKNOWLEDGMENTS

The authors thank Erik-Jan Marinissen and Harald Vranken for their review comments on draft versions of this paper, and Greg Ehmann for his input on the Sorcery SoC.

9. REFERENCES

[1] ARM Limited, http://www.arm.com. *ARM920T (Rev1) Technical Reference Manual*, 2001.

[2] H. Hao and R. Avra. Structured design-for-debug - the SuperSPARC-II methodology and implementation. In *Proceedings IEEE International Test Conference (ITC)*, pages 175–183, Washington, DC, USA, Oct. 1995.

[3] K. Holdbrook, S. Joshi, S. Mitra, J. Petolino, and R. Raman. microSPARC: A Case Study of Scan Based Debug. In *Proceedings IEEE International Test Conference (ITC)*, pages 70–75, Washington, DC, USA, Oct. 1994.

[4] D. Josephson, S. Poehlmann, and V. Govan. Debug Methodology for the McKinley Processor. In *Proceedings IEEE International Test Conference (ITC)*, pages 451–460, Baltimore, MD, USA, Oct. 2001.

[5] MIPS Technologies, http://www.mips.com. *EJTAG Specification, Document Number: MD00047, Revision 02.53*, Jan. 2001.

[6] B. Roberts. The verities of verification. *Electronic Business*, Jan. 2003.

[7] G.-J. v. Rootselaar and B. Vermeulen. Silicon Debug: Scan Chains Alone Are Not Enough. In *Proceedings IEEE International Test Conference (ITC)*, pages 892–902, Atlantic City, NJ, USA, Sept. 1999.

[8] B. Vermeulen, S. Oostdijk, and F. Bouwman. Test and Debug Strategy of the PNX8525 Nexperia Digital Video Platform Chip. In *Proceedings IEEE International Test Conference (ITC)*, pages 121–130, Baltimore, MD, USA, Oct. 2001.