Breakpoints and Breakpoint Detection in Source Level Emulation

Gernot Koch[1], Udo Kebschull[1], Wolfgang Rosenstiel[2]

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

Abstract

In this paper we discuss, what breakpoints in Source Level Emulation[a] are, how we can work with them and how we have to change the cicuit generated by high level synthesis to do so. We show the details of breakpoint encoding and detection in our approach. The presented approach allows for breakpoint detection by hardware means without seriously slowing down the circuit or dramatically increasing its size.

1. Introduction

For years, high level synthesis is of increasing importance in design automation. The increasing number of commercial available tools for high level synthesis indicates that the abstraction level of the design entry will raise to the algorithmic level at least for some kinds of applications

While there is much effort spent in methods for synthesis, there are only few groups concentrating on design validation and debugging of high level specifications. In general, validation can be done by simulation and emulation. Simulation is a powerful means to detect mistakes in the specification. Some groups work on simulation tools for whole systems consisting of hardware and software and even other technologies like mechanics [1][2]. However, simulation can only be applied to the specification level, and even there, a design can only be partially validated because simulation is very time consuming.

Validating at the specification level is not sufficient, if high level synthesis is applied. There are interface components and custom components in such designs, which are not visible at the algorithmic level. So validation is also necessary on lower levels of abstraction. But there, simulation is less applicable because it is too slow.

a. Work partially supported by the Deutsche Forschungsgemeinschaft (DFG)

Emulation is a technique, which allows for validation of designs at a low abstraction level almost in real time [3]. But emulation works at the gate level. Thus, a designer cannot relate the probed signals to the specification. For this reason, debugging is only possible at the I/O signals of the system. Quickturn has addressed this problem with the HDL-ICE [4] system, which allows to relate the probed signals to an RT-specification, but still there is a big gap between the algorithmic level and the level, where validation is done.

In [10], we have proposed Source Level Emulation (SLE) as a method to close this gap by combining behavioral simulation with hardware emulation. The idea of SLE is to run the application on an emulator hardware and to keep the correlation between hardware elements and the behavioral VHDL source such that it is possible to stop the hardware by interrupting the clock and to extract values of variables in the source code by reading registers of the circuit. This correlation is mainly obtained through logging the synthesis steps of the high level synthesis.

SLE allows for symbolic debugging of a running hardware similar to software debugging. 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. By backannotating the values read from the circuit, we can do debugging at the source code level. We do not need to capture the environment of the application in a simulator. We just connect the emulator 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.

In this paper, we discuss two important issues of SLE in more detail: The hardware that we introduce additionally into the generated circuit to set and detect breakpoints, to read data path registers, and to control the circuit operation. The second issue of this paper is to define, what a breakpoint is in terms of hardware and how we can man-

age to set and detect such breakpoints without needing a tremendous hardware overhead and without extremely slowing down the circuit. In the next chapter we discuss the breakpoint issue. In chapter 3 we give a detailed description of the hardware extensions we need for SLE. We conclude the paper with some results in chapter 4 and a summary.

2. Breakpoints in SLE

2.1. Breakpoint definition

High level synthesis generates a circuit from a software program like specification. The circuit contains a data path, where the data computations are carried out, and a controller, which controls when a component in the data path is active. The controller is a finite state machine that sets control values for the data path and reacts to condition values from the data path.

Since we want to debug a running circuit at the source code level, we have to define a breakpoint as something which is visible in the source code. On the other hand, a breakpoint must also be visible in the circuit to enable us to detect it. Thus, we define a breakpoint as an operation like +, *, etc. If such an operation is executed by a component, we can detect it in the running hardware.

Definition 1. A *breakpoint* is a triple (Op, IO, T). It is interpreted as the time T, at which the input or output IO of operation Op transports data.

Definition 2. A breakpoint is called *detectable breakpoint*, if Op is implemented by a hardware component and if IO is visible in the data path at the RT level.

Figure 1 illustrates the definitions. Part a) shows all inputs and outputs of a '+' operation as visible in the data path. Part b) shows an example, where the input/output X is not visible due to an optimization during synthesis.

In the following, we will use breakpoint as synonym to detectable breakpoint. The use of non detectable breakpoints is not allowed since we cannot detect them by hardware means.

2.2. Breakpoint types

One possibility to detect the time T of a breakpoint in the hardware is by looking at the controller states, since the controller manages the sequential behavior of the data path. The relation between the breakpoint time, and a controller state is static and known from the synthesis process. In the data path, there are operations which are always executed in a certain controller state. We call breakpoints

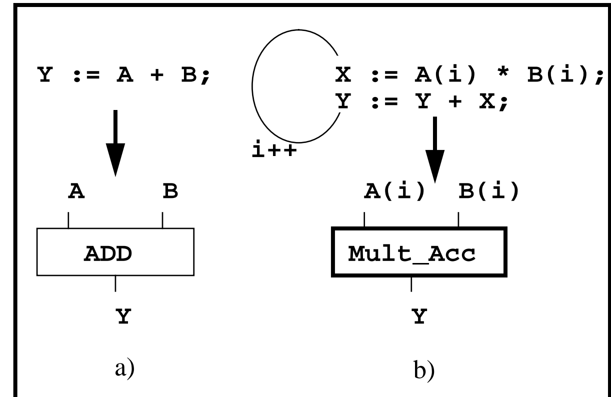

Figure 1. a) A, B, Y as detectable breakpoints and b) X as non detectable bp

at this type of operations **Moore breakpoints**. Such a breakpoint is reached, if the controller state matches the breakpoint time T.

There are also operations, that are executed at transitions of controller states. To detect breakpoints at those operations, we need to know about the current state and the next state of the controller. These breakpoints are called **Mealy breakpoints**. A Mealy breakpoint is reached, if the controller state matches the breakpoint time T and if the next state is right one for the breakpoint.

Both, Mealy and Moore breakpoints, can be extended by data dependency. This allows for setting a conditional breakpoint. For instance we could say "stop at `B` in the expression `Y := A * B;` if B = 5." More sophisticated conditions are possible, but require also more hardware overhead. Naturally, we could always stop at a breakpoint and do the evaluation of the conditional expression in software after reading the corresponding values from the circuit. This is not what we want, since it would make it impossible to run the circuit in its real environment. Thus we have to keep the conditions very simple.

2.3. Breakpoint encoding

Since we want to react, if a breakpoint is reached by interrupting the circuit, we have to detect the breakpoints by hardware means. To do that, we need to encode the breakpoints in a way that allows for very simple testing whether a controller state corresponds to a breakpoint or not. This encoding of the breakpoint (breakpoint ID) has to satisfy the following constraints:

- Different breakpoints must have different IDs.
- For each controller state that is element of a particular breakpoint set the controller must produce the corresponding breakpoint ID. This can be compared to a breakpoint register. If equal, the breakpoint is reached.

Above, we have implicitly mentioned. that a breakpoint would correspond to one controller state. Then the state

encoding would also serve as breakpoint encoding for hardware breakpoint detection. The reality is not as simple. A breakpoint can be represented by a whole set of controller states. Furthermore, these sets of states can intersect, as the example in figure 2 shows. For instance this could be caused by path based scheduling [7].

In Figure 2 we have constructed a case to show what

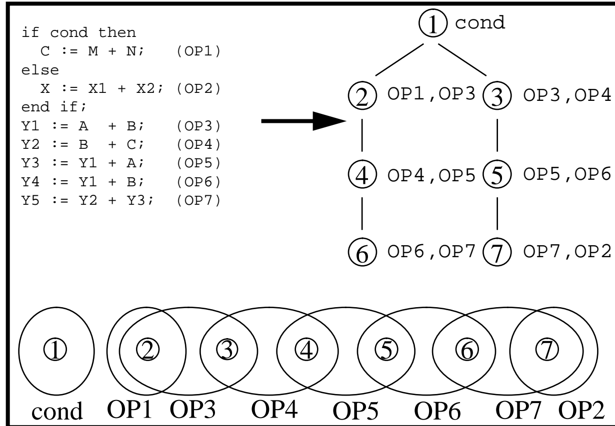

Figure 2. Breakpoints represented by an intersecting set of controller states

can happen. For simplicity, we identify the breakpoint triple (Op, IO, T) by the operation Op.The code fragment does not compute anything sensible, but a similar case may occur in a real example. If we constrain the hardware resources to two adders and a comparator, scheduling and controller construction can result in a controller as shown. The left path of the controller shows the if-path, the right one shows the else-path. There are 8 detectable breakpoints, three of them represented by one controller state, the others represented by two states each.

In state 3, for instance, we have to produce the ID of breakpoint OP3 and of breakpoint OP4, but we can only have one controller output word in this state. An easy but inefficient way to solve the problem would be to code each breakpoint with three bit and to have a 6-bit word as controller output containing both IDs. The first 3 bit would then indicate OP3, while the last 3 bit would indicate OP4. To detect breakpoint OP3, we would tell the breakpoint comparator to ignore the last 3 bit. States 3 and 4 have to show the same pattern in the first 3 bit then. The same rule can be applied to each state. Then we would have to add a 6 bit word to the controller outputs just for breakpoint detection. This can cause an explosion of the controller logic in larger applications.

In general, this can be formulated as the problem to binary encode elements of an arbitrary set structure in a way, that, given a particular subset code, one can see whether or not an element is part of that subset just by ignoring some bit positions (which are fixed for a subset). As many other problems in synthesis, this is an NP-hard problem. Thus, for the general case, a heuristic has to be used.

2.4. A heuristic for breakpoint encoding

The heuristic we use is driven by the following assumptions and facts:

• Usually, if a breakpoint set A intersects with another breakpoint set B, then A is a subset of B or vice versa.

• A subset structure like the one given in Figure 2 can be constructed, but it will happen very rarely to that extend, since the data dependencies usually won‘t allow for such differences in the execution order in different if-paths.

• Breakpoint intersection can only happen in the local context of loops if these contain a hierarchy of if-branches. There cannot be an intersection of breakpoints belonging to different loops, since all loop are scheduled independently of each other. This greatly reduces the size of such intersected breakpoint clusters.

The algorithm we use is hierarchically organized. A level of hierarchy is made up of all intersecting sets, excluding supersets, which belong to the next higher level, and subsets, which belong to the next lower level. Encoding starts at the highest of hierarchy.

We define *Top* as the set of all breakpoint sets, *Top(S)* as the set of all subsets of *S*. The set of clusters in the set *S* is *C(S)*, where a cluster C_S is given by:

$$\forall N \in C_S \exists M \in C_S | \{ N \cap M \neq \emptyset \wedge N \cup M \neq N \wedge N \cup M \neq M \}$$

Init: We set the working set *Current* = *Top.*

Step 1: $\forall(C_{Current} \in C(Current))$: Assign a binary encoding (the cluster ID). The following sub-encodings of different clusters are attached to the cluster ID and can share the same bit positions.

Step 2: Within each $C_{Current}$, all intersecting sets are assigned one bit. This number of bits is added to the cluster ID.

Step 3: $\forall(C_{Current} \in C(Current)) \forall(S \in C_{Current})$: Set *Current* = *Top(S)*, and go to Step 1. The result of this sub-encoding is attached to the encoding achieved so far. For different *S*, these share the same bit positions.

In step 2, we need the assumptions we made above about the intersection of breakpoints. E.g. in a case like the one given in figure 2, the encoding according to step 2 is far away from being optimal. Here we would need 8 bits with our heuristic, while an encoding with 4 bits is possible to fulfil the constraints mentioned above.

The number of bits is given by the following recursive formula:

$$\mathrm{Bits}(Top) = \lceil \log|C(Top)| \rceil + \max(|C_{Top}|) + \max(\mathrm{Bits}(S \in C_{Top}))$$

$$\mathrm{Bits}(\emptyset) = 0 .$$

The worst case runtime of the algorithm is $O(Bp^2 * CS)$, where Bp denotes the number of possible different breakpoints and CS denotes the number of controller states. This is an equal complexity to state of the art scheduling algorithms since Bp is proportional to the number of operations. In practice, the runtime is much less, since there is no global intersection of breakpoints, thus the algorithm always works locally.

2.5. Example for breakpoint encoding

The example in figure 3 shows a number of intersecting breakpoints to which we apply our algorithm to clarify its operation.

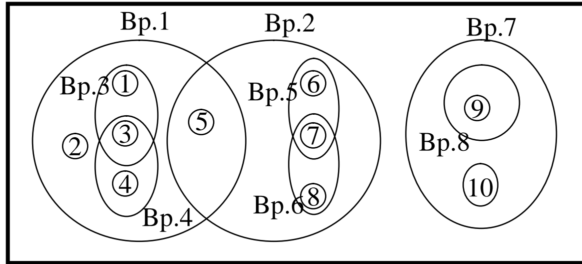

Figure 3. Example for the encoding algorithm

First, we identify two clusters at the top level of hierarchy:

C1: Bp. 1, 2, 3, 4, 5, 6

C2: Bp. 7, 8

The first bit of the encoding is then '0' for C1 and '1' for C2. The next bits depend on the clusters and may be shared for different clusters. For C2 we get one bit for Bp. 7 on the highest level and if we go down one level, we get an additional bit for Bp. 8.

For cluster 1, we get 2 bits, one for Bp.1 and one for Bp. 2 at the highest level. Going down one level, we get another 2 bit for Bp. 3 and Bp. 4, which share the positions with the 2 bit for Bp. 5 and Bp. 6. Thus, the encoding of the states is according to table1.

Bit 1 tells us the cluster, bit 2 indicates that a state belongs to Bp 1, if we are in cluster C1, or to Bp. 7, if we are in cluster C2. The meaning of the other bits is analog to that. The encoding needs 5 bit, which is optimal by chance in this case. It cannot be done with less bits and still satisfy the necessary constraints.

3. Breakpoint detection in the hardware

3.1. Breakpoint detection logic

As mentioned above, we can detect breakpoints by comparing the encoded controller state identifier with a given breakpoint ID. Depending on the breakpoint we are interested in, we need to ignore certain bits of the identifier. The identifier is a signal generated by the controller. An example of the VHDL style of the generated controller together with the breakpoint identifier is shown in figure 4.

State	Encoding
1	0 10 10
2	0 10 00
3	0 10 11
4	0 10 01
5	0 11 00
6	0 01 10
7	0 01 11
8	0 01 01
9	1 1 1 --
10	1 1 0 --

Table 1. Example State encodings

```
fsmlogic: process(current_state, input_list)
begin
  default_assignments;
  case current_state is
    WHEN state1 =>
      identifier <= "01010";
      output_assignments;
      next_state_assignment;
    WHEN state2 =>
      identifier <= "01000";
      output_assignments;
      next_state_assignment;
    ......
  end case;
end process fsmlogic;
```

Figure 4. Breakpoint IDs in the generated VHDL controller

It is a process with a case statement, in which the outputs and the next state are computed according to the current state. In each state, we generate the corresponding identifier for breakpoint detection.

The breakpoint detection logic we need to add to the generated circuit is shown in figure 5.

We need a register to write in the desired breakpoint and a mask register, where we tell the comparator, which bits of the breakpoint identifier and the breakpoint register it should ignore. The first bit of the breakpoint register tells the comparator to constantly output '0' if it is set, i.e. it represents the operation with no breakpoint set. The comparator tests the not masked bits of the breakpoint identifier and the breakpoint register for equality. If equal, the BP-reached signal is raised to '1'. The BP-scanin signal represents a scan path for setting and changing breakpoints in circuit. Thus, we do not need to synthesize the

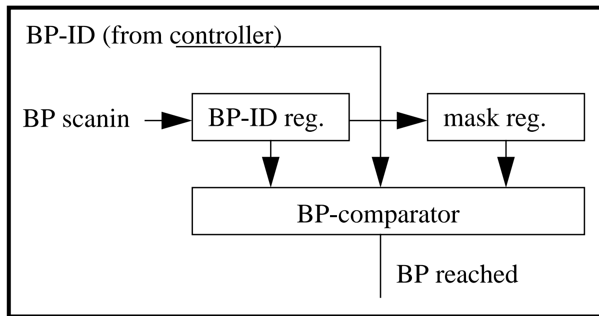

Figure 5. Breapoint detection logic

circuit each time we want to set a new breakpoint. All this is done dynamically in the implemented circuit.

The detection of data dependent breakpoints is also very similar to that. We just replace each register in the data path by a component which contains a data register, a programmable register for breakpoint detection and a comparator.

3.2. Debugging controller

The programming of breakpoints and the interrupt of the circuit is done by the debugging controller. This component allows the host to control the circuit operation. An abstract scheme of its operation is shown in figure 6.

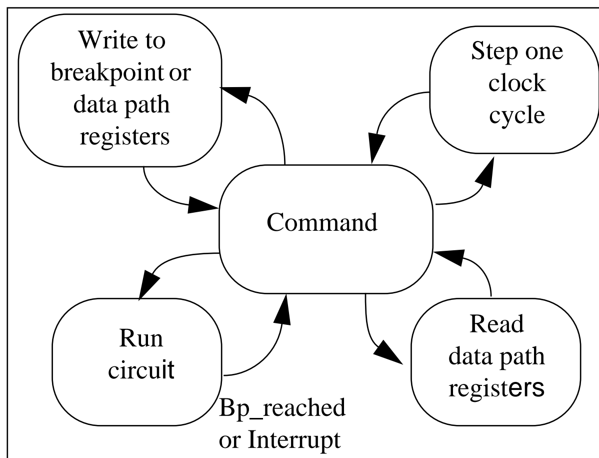

Figure 6. Simplified diagram of dbg. controller

The macro state **command** is the initial state of the controller. In this state it waits for a command and the circuit is interrupted. As commands we can have the following:

- Write a breakpoint
- Write values to the data path registers
- Read the contents of the data path registers
- Step one clock cycle
- Run the circuit

Each command is decoded, and the controller switches to the corresponding macro state, which manages the required action. In each macro state, the controller comes back to the **command** state after the action is finished. Only in **run_circuit** state it remains until a breakpoint is reached or an interrupt command is issued by the host. Interrupting the circuit is done by setting a clock_control signal to '0', which then prevents the data path registers and the circuit controller from operation. Thus, the circuit halts.

Each register in the data path is exchanged according to figure 7 by a register which allows for the required opera-

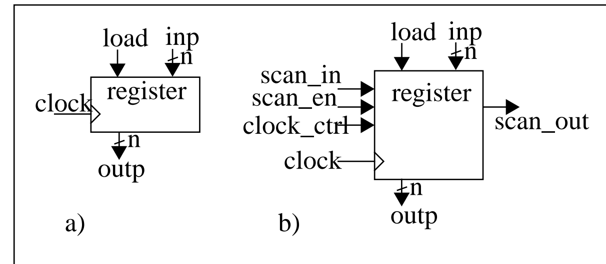

Figure 7. a) Original and b) for dbg inserted data path register

tion control. The inputs added to the original data path register are used for programming/readback of the register contents via a scan path (scan_in, scan_en, scan_out), and for enabling normal data path operation (clock_ctrl). These additional inputs are managed by the debugging controller.

The debugging controller is not application dependent. We implement it together with the application on the FPGAs for reasons of simplicity, but it may also reside outside of the programmable logic.

4. Results

The described technique is implemented within our synthesis tool named CADDY [5][6][7]. All changes to the circuit controller and to the data path are applied automatically during the generation of the VHDL code by CADDY. There is no manual work by the user required for the generation of the debug model. The debugging controller communicates with a SUN workstation via the parallel port.

The applications listed below for the results were all real synthesized, downloaded and run on our WEAVER [8] prototyping board. The following applications were done:

- GCD: A standard example for high level synthesis.
- SIRDG: A circuit which computes "single image random dot stereograms". It receives a source image with the height information via a parallel interface and writes the generated image to a host via a parallel interface.

• DCT: A circuit which computes a two-dimensional 8x8-DCT. It is implemented as a coprocessor for the Hyperstone processor [9]. The communication with the processor is done via the external processor bus.

In table 2, the area values of the different circuits are listed. All values are given in CLBs for the Xilinx XC4000 series. Please note, that the constant overhead for the debugging controller (ca. 44 CLBs) virtually can be subtracted from the overhead numbers for the examples, because it is not depending upon the application. It could easily be implemented separately from the FPGAs.

Circuit	Orig.	Dbg 1	Dbg 2	Mealy	Moore
GCD	60	128/84	141/97	3	3
SIRDG	99	190/146	213/179	2	5
DCT	387	474/430	476/432	1	6

Table 2. Area values

• **Orig.** refers to the original circuit without debug overhead.

• **Dbg 1** includes the overhead for debugging without the possibility of data dependent breakpoints. It shows the value with/without debugging controller.

• **Dbg 2** represents the full version with data dependent breakpoints.

• **Mealy** denotes the number of bits needed for the Mealy identifiers, **Moore** denotes the number of bits of the Moore identifiers. The sum of Mealy and Moore is added to the output of the circuit controller.

• **#Bp** denotes the number of different detectable breakpoints in the specification.

• **#States** lists the number of controller states of the circuit controller.

Table 3 shows the delay that is added by the additional logic for debugging.

Circuit	Orig.	Dbg 1	Dbg 2
GCD	5.7 Mhz	4.7 Mhz	4.5 Mhz
SIRDG	7.7 Mhz	4.8 Mhz	4.9 Mhz
DCT	3.5 Mhz	3.5 Mhz	3.5 Mhz

Table 3. Clock frequencies

The values are obtained by the Xilinx tool 'xdelay'. This tool provides a quite pessimistic estimation. All designs run at a significantly higher clock speed on our Weaver board. Nevertheless, it shows the relation between the different implementations. There is not much difference between the two versions with debugging overhead, since the evaluation of conditional breakpoints does not slow down the circuit additionally. Our approach mainly adds controller delay. Therefore we do not add any delay to the dct, which is mainly determined by the combinatorial multiplier. To this path, we do no add any delay. Therefore, the debug versions can be clocked with equal frequency.

5. Summary

In this paper, we have presented the details of how we handle breakpoints in our new SLE approach. We have discussed, how we can relate breakpoints set in an algorithmic specification to an implemented circuit. We have shown, how we encode the breakpoints and how we detect them by hardware means. We have demonstrated the applicability of our approach by a set of implemented circuits.

6. References

[1] Y. Tanurhan, S. Schmerler, K. Mueller-Glaser, *A Backplane Approach for Cosimulation in High-Level System Specification Environments*, European Design Automation Conference EURODAC‘95, Brighton, 1995

[2] J. Soininen, T. Huttunen, K. Tiensyrjae, H. Heusala, *Cosimulation of Real-Time Control Systems*, European Design Automation Conference EURODAC‘95, Brighton, 1995

[3] H. Owen, U. Kahn, J. Hughes, *FPGA based ASIC Hardware Emulator Architectures*, School of Electrical and Computer Engineering, Georgia Institute of Technology, 1993

[4] *ASIC-Emulation auf RTL-Level*, Markt&Technik - Wochenzeitung für Elektronik Nr. 42, 1994

[5] R. Camposano, W. Rosenstiel, *Synthesizing Circuits from Behavioral Descriptions*, IEEE Transactions on CAD, Vol. 8, 2-1989

[6] P. Gutberlet, J. Müller, H. Krämer, W. Rosenstiel, *Automatic Module Allocation in High Level Synthesis,* European Design Automation Conference EURODAC‘92, Hamburg, 1992

[7] P. Gutberlet, W. Rosenstiel, *Scheduling Between Basic Blocks in the CADDY Synthesis System,* European Conference on Design Automation EDAC‘92, Brussels, 1992

[8] G. Koch, U. Kebschull, W. Rosenstiel, *A Prototyping Architecture for Hardware/Software Codesign in the COBRA Project,* Proceedings of 3rd international Workshop on Hardware/Software Codesign Codes/CASHE‘94, Grenoble 1994

[9] Hyperstone electronics, *Hyperstone E1 32-Bit-Microprocessor User‘s Manual*, 1990

[10] G. Koch, U. Kebschull, W. Rosenstiel, *Debugging of Behavioral VHDL Specifications by Source Level Emulation,* European Design Automation Conference EURODAC‘95, Brighton, 1995