Co-Emulation and Debugging of HW/SW-Systems

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 present a method that allows to observe and control the emulation of communicating systems consisting of hardware and software parts. The approach provides the ability to set breakpoints in either technology without breaking the protocol or loosing data. Systems may contain several software and hardware processes. Each process runs in a separate debugger while the presented approach provides the necessary synchronization between the processes that precludes loss of data at the communication interfaces due to breakpoints. Therefore it allows to continue the operation after a breakpoint which is essential for interactive debugging.

1. Introduction

Codesign of embedded systems is becoming more and more popular. While many research groups are working on new codesign methods, much less effort is spent on co-validation and co-debugging of such systems. In practice, mainly two technologies are used for co-validation, simulation based validation and emulation based validation. Simulation based validation maximizes the observability of a design, but offers a rather poor performance. The classical approach works with simulation models of processors for the software part and thus allows for a cycle true simulation of the whole system. This simulation can include other technologies than merely hardware and software [1]. Recent and more efficient approaches run the software on real processors and map the communication onto interprocess communication via the PLI of common hardware simulators. Since the simulator does not need to simulate the processors with this approach, the simulation is sped up significantly. [2] presents such an approach for C-VHDL co-simulation. Usually the performance of these approaches is up to a few hundred cycles per second, depending on the complexity of the hardware description.

Emulation based validation offers much less observability, but a much higher performance. Emulated systems can even run fast enough to be used within the real environment of a system as a prototype. Since the software parts run on their target processors for which development environments and debuggers exist, the observability issue concerns rather the hardware than the software. The Quickturn HDL-ICE [3] enables emulation based co-validation with observability of the hardware and backannotation of values to the RTL-description. [4] presented a co-emulation system for a fixed target architecture. It allows to set breakpoints by specifying patterns of the address bus. However, it does not allow to set functional breakpoints in the HDL-description of the hardware. Also it offers only very limited observability. The designer can evaluate memory contents, but not internal signals in the hardware. It is possible to continue operation after a breakpoint because the whole system clock is stopped at a breakpoint. However, in general this is impossible. Usually, processors and hardware run on different clocks and it is not possible to stop the clock of a processor. If the system includes DRAM, stopping the clock is also not possible since the memory cannot be refreshed then.

Since the hardware entry level is moving up to the behavioral level, breakpoints and observability on the gate level or on the RT-level are no longer sufficient. In [5] we have presented an approach which allows to control and to observe an emulated hardware by using the behavioral description. In [6] we have shown how we can set breakpoints in the behavioral description, and how we can use them for interactive debugging.

In this paper we present an approach based on this previous work, which allows for interactive debugging of co-emulated systems. In the next chapter we define our notion of communication. In chapter 3, we present how we can transform a design transparently into a debuggable design and in chapter 4 we show some examples to which we have applied our approach.
2. Communication Issues

This work concentrates on systems containing multiple processes. These processes may be software processes, running on a microprocessor, or hardware processes, designed as ASICs. All processes are synchronous processes and each of them is running on its own clock. Thus, all the processes run asynchronously. This is a typical constellation for embedded systems.

2.1. Problem definition

If an application with communicating processes runs on an emulator and one of the processes is stopped at a breakpoint while the other process isn’t, data may be lost or the protocol may fail. Both cases could make it impossible to continue the operation after the breakpoint. We want to prohibit data loss or protocol failure due to one communication partner being stopped in a breakpoint, allowing to continue the operation after a breakpoint. This will of course not always be possible, but there is a reasonable number of applications and communication schemes where something can be done, as the remainder of this paper shows.

2.2. Protocol classification

Before we can go further into our solution, we need to review the communication protocols used in embedded systems. In general, we can distinguish three different, largely independent criteria. Signal synchronisation describes how the IO-signals between the communicating partners interact during communication. It can be delay insensitive, e.g. a handshake protocol. It can also be delay sensitive, as any synchronous communication between a processor and a co-processor via the processor bus interface is. The processor just writes or reads according to its timing specifications and expects, that the co-processor can react fast enough.

Another criterion is the data synchronisation. Data synchronisation provides additional information about the data that is transferred, e.g. it tells the partner process, that data is waiting to be read or that it is ready to receive data. It may also be used to negotiate the direction of data transfer for bidirectional channels. Possible are external data synchronisation, internal synchronisation and no synchronisation. External synchronisation uses separate signals for synchronisation, while internal synchronisation uses the same signals as the data transmission. Internal synchronisation uses special characters or bitpatterns, which have to be escaped to be distinguishable from data.

The last criterion describes which type of communication is actually used. This can be RegisterIO, Interrupt, DMA, etc. Figure 1 shows the possibilities and some typical examples.

![Figure 1. Protocol classification](image)

In this figure, the shaded block shows a bidirectional communication via a standard parallel interface. The signal synchronisation is implemented as handshake, while the data synchronisation and negotiation of the transfer direction is done as internal synchronisation. Typically, the parallel interface works as RegisterIO, since at both ends, simple registers read or write the data. The shaded block shows a DMA. Data synchronisation is done via memory address which is not different from the data addresses, thus it is an internal synchronisation. Reading from or writing to the RAM is usually delay sensitive. The figure shows only the data communication, not the negotiation with the DMA controller. That part is usually a handshake protocol. Accordingly, shows a typical processor - co-processor communication, while is typical for a pure hardware communication. Often, the fact that data is written or read is used as data synchronisation, making explicit data synchronisation unnecessary.

2.3. Specifying communication protocols

The ability to debug communicating processes is largely depending on the specification style of communication protocols. There are basically two ways to specify protocols at the behavioral level. The protocol can be embedded in the algorithm and it can be encapsulated in a component. Figure 2 shows an example for both alternatives together with the resulting parts of the controller state machine.

Part a) shows how communication can be embedded into the algorithmic specification. The protocol in this example is a parallel interface protocol. The timing relative to the clock and the interaction of the signals is explicitly described. This has the advantage that the resulting design is quite small, but on the other hand the protocol cannot be easily exchanged and the design is only capable of receiving data, when the controller is waiting in the corresponding states. There is no computation going on in
parallel to the communication (rendevous schema). Both communicating processes meet at that time. Thus, only protocols with a handshake signal synchronisation can be implemented as shown in part a).

Part b) shows how communication protocols can be encapsulated and hidden from the algorithm. The data path controller communicates via a standard procedure with an interface component. The component is triggered with a start signal and answers back with a done signal and, if available, with data. The value of the done signal determines whether or not the data is valid. A start trigger like the start signal here is necessary for all components with an internal controller. In the HDL code this is simply represented as a procedure call.

Encapsulating the communication protocol has three advantages. First, the protocol can easily be exchanged by another protocol by simply calling another procedure, without messing up the whole algorithm. It could even be exchanged by replacing the interface component in the data path after high level synthesis is done, without changing the algorithm at all. As we will show in the next chapter, this is exactly the feature that we use to create a debuggable circuit.

Second, the interface is always ready to communicate, no matter what the data path is doing at that time. So communication and data path operation is somewhat decoupled and running in parallel. This can speed up the whole operation. Third, the designer of the algorithm does not have to be a communication expert. The interface components are pre-defined and placed into a library. There is no need to re-write them each time from scratch. As a little penalty, the resulting design size will increase somewhat. The data path controller does not change a lot (depending on the protocol of course), while the interface component needs an internal controller in addition.

3. Creating a Design-for-Debugging

3.1. Synthesis -g

As to produce a debuggable version of a design, we employ a special synthesis as shown in earlier publications. This synthesis instruments the design in a way that allows to control and observe its operation. As figure 3 shows, there is an additional component library with debug models of all sequential components like pipelined components or components with internal controllers, e.g. interface components. During normal synthesis, all components are taken from the standard library, while during synthesis for debug, called synthesis -g in the figure, all sequential components are taken from the debug library where the debug models reside. This happens automatically and is not visible to the designer.

There are two types of debug models for sequential components. The ones for purely internal components, which do only interact with the data path, simply do not change their internal state in a breakpoint. The debug models for interface components show a more sophisticated behavior. On one hand, their interface to the data path remains constant during a breakpoint, while on the other hand the communication protocol with the outside world continues to work as to fulfil the protocol.

The idea is that for each sequential component in the component library the designer also provides a debug model of the component in the debug library. In most cases doing this is trivial. However, it is not trivial for interface components

3.2. Debug models for interface components

Unfortunately we cannot provide a general rule how to derive a debug model from an arbitrary interface component. But we can provide a rule for a certain type of interface component and we can show that all common communication schemes can be implemented with components of that type. A typical protocol for reading from a standard parallel port as shown in figure 4 may serve as an example.

\[
\begin{align*}
\text{ack} & \leftarrow '0'; \\
\text{bsy} & \leftarrow '0'; \\
\text{wait until} & \text{ clk'event and clk} = '1'; \\
\text{while} & \text{ strb} = '0' \text{ loop} \\
\text{wait until} & \text{ clk'event and clk} = '1'; \\
\text{end loop; } \\
\text{data} & \leftarrow \text{idata; } \\
\text{bsy} & \leftarrow '1'; \\
\text{ack} & \leftarrow '1'; \\
\end{align*}
\]

\[
\begin{align*}
L: & \text{loop} \\
\text{done} & \leftarrow \text{read(data); } \\
\text{wait until} & \text{ clk'event and clk='1'; } \\
\text{exit} & \text{L when} \text{ done} = '1'; \\
\text{end loop;} \\
\end{align*}
\]
It uses external data synchronisation via the busy signal. If busy is '1', the partner cannot send data because the component is not ready to receive it. The rest of the protocol is a 4-way handshake involving the nstrobe and nack signals. The behavior in a breakpoint is that the states IDLE and READ are processed as in normal operation, allowing the component to receive data even in a breakpoint, if it is ready to receive. In the state DATA, the transition triggered by \( \text{start} = '1' \) is delayed until the next active clock edge after the breakpoint. Since the value \( \text{"busy"-signal} \) is constant '1' in that state, the sender cannot send any data and has to wait until the end of the breakpoint. Thus no data is overridden or lost.

We can identify three important elements in that example. First, there is a data synchronisation through the \( \text{"busy"-signal} \). This allows the component to prevent the sender from sending data that cannot be received during a breakpoint. Second, there is a state, where the component is waiting for the data path to act via the \( \text{"start"-signal} \). In that state, the communication partner is held back by the data synchronisation. Third, there is a set of states, in which the component runs through the protocol. In these states, requests from the data path (\( \text{"start} = '1' \)) are answered with \( \text{"done} = '0' \), meaning that there is no data waiting. The general cases for sending and receiving data are depicted in figure 5. In both directions there is a state, where the component is waiting for the data path to give the start trigger. In this state, data synchronisation prevents the partner process from becoming active at the interface. Only in this state, state transitions are triggered by the data path.

A component of this type can easily be transformed into a debug model by performing two steps. Step 1 delays the state transition from states like EMPTY or HAVE_DATA, which are triggered by the data path, until the next active clock edge after a breakpoint. In a breakpoint, the component can work on the protocol until it runs into a state like EMPTY or HAVE_DATA and then it stops until the operation is continued. During the stop, data synchronisation ensures, that nothing gets lost at the interface.

**Figure 4. Simple protocol**

<table>
<thead>
<tr>
<th>State</th>
<th>busy &lt;= '0'</th>
<th>nack &lt;= '1'</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE</td>
<td></td>
<td></td>
<td>Ready to receive data</td>
</tr>
<tr>
<td>READ</td>
<td>busy &lt;= '1'</td>
<td>nack &lt;= '0'</td>
<td>Reading data</td>
</tr>
<tr>
<td>DATA</td>
<td>busy &lt;= '1'</td>
<td>nack &lt;= '1'</td>
<td>Waiting for data path to take over the data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>start = '1'</th>
</tr>
</thead>
<tbody>
<tr>
<td>nstrobe = '0'</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>nstrobe = '1'</td>
</tr>
</tbody>
</table>

**Figure 5. Behavior of interface components**

Step 2 adds some hardware which holds all outputs to the data path constant during a breakpoint. This cannot be guaranteed without additional hardware, because of the possible operation that is going on during a breakpoint in the component.

Bidirectional interface components can easily be implemented by combining the state machines for sending and receiving as two independent state machines in one component. These components still follow the same rules and are transformed in the same way.

The protocol part of the state machines can implement any type of protocol. It is easily possible to implement interrupts or DMA protocols. In case of an interrupt protocol the state machine would first set an interrupt signal and then wait for the interrupt procedure to handle the data transfer.

If a DMA is implemented, the component would only handle the protocol with the DMA controller of the processor, not the data communication itself. This would be implemented via a multiport RAM or an addressable register file.

### 3.3. Limitations

It is obvious that there are limitations to the presented approach. In general, it cannot be applied to two types of systems. First, if the environment of the design imposes hard realtime constraints on the design, which cannot be softened for the prototype. E.g. an ethernet controller which has to listen at the bus with a certain speed cannot be interrupted at a breakpoint and still continue to work after a while. Second, if the design communicates with external devices without any data synchronisation and without a handshake protocol. In these cases, the external device cannot be manipulated by a special synthesis for debugging and thus, the presented method is not applicable. However, if two processes communicate with a protocol that uses neither data synchronisation nor handshake,
but both processes are part of the synthesized design, the protocol can be transparently exchanged by a suitable protocol during synthesis as to make the prototype debuggable.

If the approach is not applicable, then there are two possible debugging strategies. One can operate with a logic analyser as it is done today, and one can run until a breakpoint occurred, evaluate the state of the system and restart from the beginning with the same parameters, but stop at another breakpoint.

4. Results

We have applied the approach to some example systems. For hardware synthesis, we have used the CADDY High Level Synthesis tool [7][8][9], which was developed at FZI. The mechanism for transparent component exchange for debugging purposes has been implemented as a "-g" switch in this tool.

The first example is a circuit that computes a 2-dimensional DCT. Data is transferred from and to a software process via DMA accesses. Figure 6 shows the structure of the example. It uses two DMA channels of the PPC403GA [10] embedded processor, one for receiving the source data and the other for sending the transformed data. A DMA component which interacts with the DMA controller of the processor, and an internal RAM which holds the data are associated with each channel.

Figure 7 shows an excerpt from the behavioral VHDL code of the design. The first inner loop initiates the DMA request and waits until this has been acknowledged by the component. The second inner loop waits until the DMA has been finished and the data is available. Then the DCT Algorithm can start to work on the data. Writing the result back to the processor is done analogously. Both, the DMA[0,1]()-procedures and the sync[0,1]()-procedure are mapped to the DMA[0,1]-component.

The behavior of the debug model of the DMA0 component is shown in figure 8. In the IDLE state, the component waits for the data path to initiate the DMA request via a call of the procedure DMA0(). It then works through the handshake protocol with the DMA controller and ends up in the SYNC state. Now the whole DMA is finished, the data is transferred and the component waits for the next transition to the DMA Req or Ack state. The DMA1 component is treated exactly in the same way.

The version of the design that uses an interrupt to trigger the data transfer has exactly the same structure. The only difference is in the state machine of the DMA[0,1]() component. So, from a hardware point of view, an interrupt protocol doesn’t infer any further problems.

Two other protocols have been implemented and tested. One is a bidirectional RegisterIO communication with the Hyperstone [11] processor, the other is a communication via a standard parallel interface. The RegisterIO protocol uses a status register for data synchronisation and two 32bit registers for data transfer, one for each direction. Each direction of the communication is guarded by a separate state machine, which handles the data synchronisation. Both state machines are of the type that was required in chapter 3, thus transformation from the component to the debug model of the component was done in the way that was described there.

The parallel interface protocol was implemented for both directions, each direction in a separate component. On the software side, the communication is associated
with a timeout. This timeout is implemented in the operating system. When the hardware is stopped due to a breakpoint and the software wants to transfer data, then the timeout will occur. To allow debugging, the software must catch the timeout and retry to initiate the transfer in a loop. A similar protocol encapsulation in software as it is proposed for hardware in this paper allows to do that in a simple way.

We can run these designs in our debugger and do single stepping, set breakpoints and evaluate the contents of the data path registers without losing any data at the communication channel or causing a failure of the protocol.

In table 1, the area values of the different designs, which use the different protocols, are listed. All values are given in CLBs for the Xilinx XC4000 series.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Orig.</th>
<th>Debug</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCD</td>
<td>60</td>
<td>84</td>
<td>Hyperstone RegIO</td>
</tr>
<tr>
<td>SIRDG</td>
<td>99</td>
<td>146</td>
<td>Sparc10 parallel Port</td>
</tr>
<tr>
<td>DCT</td>
<td>387</td>
<td>430</td>
<td>PPC403GA DMA</td>
</tr>
</tbody>
</table>

Table 1. Area values

- **Orig.** refers to the original circuit without debug overhead.
- **Debug** includes the overhead for debugging without the possibility of data dependent breakpoints.
- **Protocol** denotes the communication protocol that was used to communicate with a software process.

Table 2 shows the delay that is added by the additional logic for debugging.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Orig.</th>
<th>Debug</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCD</td>
<td>5.7 Mhz</td>
<td>4.7 Mhz</td>
</tr>
<tr>
<td>SIRDG</td>
<td>7.7 Mhz</td>
<td>4.8 Mhz</td>
</tr>
<tr>
<td>DCT</td>
<td>3.5 Mhz</td>
<td>3.5 Mhz</td>
</tr>
</tbody>
</table>

Table 2. 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 [12] board. Nevertheless, it shows the relation between the different implementations. Our approach mainly adds controller delay. Therefore we do not add any delay to the DCT, which is mainly determined by the combinational multiplier. Thus, the debug version can be clocked as fast as the original version.

5. Summary

In this paper we have presented an approach for a combined debugging of systems containing hardware and software parts. The approach is applicable to systems which use communication protocols with data synchronisation, and to systems which use handshake protocols. Systems using neither type of protocol are debuggable only, if it is possible to exchange the protocol via protocol encapsulation, when the prototype is built. We have applied the approach to some example systems to demonstrate its feasibility.

6. References