Exploiting Partial Reconfiguration for Flexible Software Debugging

Giovanni Busonera
University of Cagliari
Cagliari, Italy
giovanni.busonera@diee.unica.it

Alessandro Forin
Microsoft Research
Redmond, WA - USA
sandrof@microsoft.com

Richard Neil Pittman
Microsoft Research
Redmond, WA - USA
pittman@microsoft.com

Abstract*- eBug is a debugging solution for software developed on the eMIPS dynamically-extensible processor. The off-chip portion of eBug is an application that performs tasks that would be too expensive or too inflexible to perform in hardware, such as implementing the communication protocols to interface to the client debuggers. The on-chip hardware portion of eBug is realized with a new approach: rather than being built into the base pipelined data path, it is a loadable logic module that uses the standard Extension interface of the processor. This accomplishes the three goals of area minimization and reuse, security in a general purpose, multi-user environment, and open-ended extensibility. When not in use, eBug is simply not present on the chip and its area is therefore reused. eBug solves the security issues normally created by a hardware-level debug module because only the process that owns the eBug Extension can be affected by a debugging session. As an Extension, eBug is not compiled into the basic processor design and this makes it easy to add new features without affecting the core eMIPS design. Leveraging the high-visibility extension interface of eMIPS, eBug can realize arbitrarily complex features for high-level monitoring. In this paper we describe how we transparently added hardware watchpoints to the initial, simpler design. It is also possible to interface eBug with other eMIPS extensions such as those generated by the P2V checker to improve its capabilities. eBug was written in Verilog and is usable both with the Giano system simulator and on the Xilinx ML401 FPGA board.

I. INTRODUCTION

Debugging is an important but tedious part of the software development process. To be profitable, it must be supported by appropriate tools. For instance, it is desirable to present the user with the status of the application as soon as the bug occurs, even though the user does not yet have a precise idea of what the error exactly is, let alone its cause. In embedded systems, debugging is performed using a remote client debugger that connects to the embedded processor using some communication protocol. The remote debugger can access the processor's resources with one of two approaches: software based and hardware based. In the first case, a piece of software called the "debug stub" runs on the target processor itself. The stub interfaces to the remote debugger by implementing the communication protocol and responding to the debugger's requests. In case of a hardware or software exception the stub is invoked and the event is reported to the debugger. No custom hardware is needed in this case, but there is some software overhead present. Moreover, the processor status is not observed in the actual moment that a trap occurs. In the second case, a custom hardware module (such as a JTAG interface) is coupled to the processor to access its resources and communicate with the debugger. This hardware module can be more or less complex, depending on the features that it implements. In general, this approach is not very flexible because adding any new feature implies a reimplementation of the whole hardware system. Furthermore, the hardware module is designed to unconditionally access all the processor resources, without any system software control. This causes security issues and is therefore never used in a general purpose, multi-user environment. In this environment, the common approach is to provide in hardware some minimal support for single-stepping and let the (system) software handle the rest.

In this paper we introduce eBug, a flexible, low overhead, security aware and easily extensible software debugging solution realized for the eMIPS processor [10]. eMIPS is composed of a fixed basic processor module that can be dynamically augmented with custom logic modules, using the FPGA partial reconfiguration feature. These modules are termed Extensions; they can access the internal processor pipeline and resources and perform special purpose tasks, therefore adding new capabilities to the running system. The primary contribution of this paper is to show how flexible hardware debugging support can be realized as an Extension to eMIPS, without any changes to the fixed processor part. A number of debugging Extensions have been implemented, providing different levels of debugging support and therefore demonstrating the flexibility of the approach from the hardware standpoint. Software flexibility is provided by an intermediate software application that interposes between the debugger client and the eBug Extension.

eBug creates little if any overhead. It is entirely optional, it uses very little area resources in a single Extension slot, and does not affect the performance of the processor in any way. When an error occurs, eBug immediately halts the processor before a trap is generated. In addition to hardware exceptions, eBug can capture a variety of conditions at the hardware level, by passively observing the processor's execution.

A second contribution of this work is to show how hardware debugging support can safely be confined within the security envelop of a (user mode) process, but without any loss in performance, extensibility or functionality. eMIPS Extensions load, unload and access the processor resources strictly under the control of the operating system. When a process is rescheduled its extensions are disabled and can no longer observe the processor's execution and resources. When the extension is enabled, its accesses to memory are filtered by the processor's MMU.

eBug itself can be easily extended. The basic design is simple and modular and allowed us to add an advanced feature such as data watchpoints with very little effort. eBug is small enough that much more functionality can be packed even into the relatively limited area available on the first eMIPS prototype.

A third contribution of this work is a new linkage between the semi-formal debugging activities of a programmer with the more rigorous tools of temporal logic. eBug can work in concert with the P2V [6] zero-overhead, online program verification system. Temporal logic assertions are realized as program-specific Extensions that can trigger eBug whenever an assertion is violated. A programmer creates these assertions either before or after the program is compiled, possibly while debugging it, as a way to express the intended behavior of the program. Execution stops immediately once the program deviates from the expected behavior, without waiting for a hardware exception to occur.

In this paper we describe the first implementation of eBug, and analyze its security capabilities and the extensibility features. In particular, we show how to improve the basic eBug functionality by adding hardware support for watchpoints and breakpoints, without any modification to the existing eMIPS design.

The software debugging approach supported by eBug on eMIPS is usable with any other dynamically reconfigurable architecture and soft-core processor, with modest changes.

The remainder of this paper is structured as follows. Section II summarizes the related work. Section III introduces the eMIPS processor. Section IV gives an overview of eBug, and the eBug software and hardware components are then described in detail in Section V and Section VI. Section VII describes how we added hardware support for watchpoints and breakpoints to the basic eBug extension. A quantitative evaluation of the design is presented in Section VIII. Future work and conclusions are presented in Section IX.

II. Related Work

On-chip support for software debugging can be found in the Leon Processor [8], an open source, Sparc V8 [13] compliant, 32-bit RISC CPU jointly designed by Gaisler Research and the European Space Agency. A debugging support unit (DSU) was introduced in the second revision (Leon2). The DSU provides a processor debug interface to the GDB debugger [3]. The DSU is available both on the real target hardware and on a simulator. In Leon2 the DSU communicates with the PC using a serial port whereas the Leon3 DSU is connected to the system bus as a slave device usable with different interfaces such as UART, JTAG, USB or Ethernet.

Xilinx provides optional hardware support for debugging software on the Microblaze soft-core [19] and on the PPC hardcore [12]. The XMD (Xilinx Microprocessor Debugger) [17] is a software tool used to interface a GDB remote session with a processor running on the real FPGA or with a cycle-accurate PPC or Microblaze instruction set simulator. The PPC hardcore includes (fixed) logic that links with XMD using a JTAG link. The Microblaze can use both a software debug stub and a hardware debug module called MDM [18]. In the latter case MDM connects the Microblaze debug interface with XMD using the JTAG interface.

Both the Leon and the Xilinx debugging support are optional features, but neither takes advantage of the FPGA reconfigurability features. Leon is an ASIC oriented design and, while FPGA implementations do exist, they do not exploit the FPGA partial reconfiguration feature to insert and remove the DSU at runtime. This is only possible at synthesis time, and only by reconfiguring the whole system. Once the DSU is included in the design, its area is wasted if debugging is not actually needed. Moreover, modifying the DSU design to implement additional features impacts the whole processor, which must therefore be re-validated.

The eBug hardware extension leverages the FPGA partial reconfiguration feature to reuse that portion of the device area when software debugging is not needed. This is a choice that is made at runtime, during execution, and not at design time. To this end, the eBug extension uses the same general purpose interface to the eMIPS datapath that is used by all the other eMIPS extensions. Using a standard interface provides additional benefits for testing and validation; only the specific extension must be re-tested and not the rest of the system or any other extension. Therefore it is possible to add new hardware-level features to eBug simply by re-implementing it, without affecting the rest of the system.

Similar considerations apply to the Xilinx’ debugging support. MDM is designed for Microblaze on FPGAs but it is not possible to remove it a run time. Moreover, MDM uses JTAG and this creates security issues. JTAG is a bus that provides low-level access to the entire system resources, not just the software under debugging. For instance, if the target processor is running a multitasking operating system there will be context-switching during a debug session. If the MDM is not properly used it can negatively affect the state of other processes and/or other parts on the system board. This is impossible with eBug because it is an extension owned exclusively by the process being debugged. When the operating system schedules another process all the extensions of the previous one are disabled and therefore they cannot affect any other software module. MDM is a proprietary system and it is not clear if it uses JTAG only to communicate with the host PC or also to access the processor resources like the register file. Compared to the processor clock, JTAG is a slow link and this can be a critical issue for remote debugging. For instance, realizing additional features such as watch-points remotely over the JTAG link would be problematic.

III. The eMIPS Processor

eMIPS [10] is a dynamically extensible microprocessor developed by the Microsoft Research Embedded Systems group. eMIPS exploits the FPGA partial reconfiguration feature to extend itself at runtime. Using this feature, a user can dynamically add custom logic to the basic processor data path at all stages of the pipeline. The additional logic, which is termed an Extension, can be used to tailor the processor for particular tasks and to improve the overall performance. Extensions can be loaded on-chip dynamically during execution by the processor itself, and only when the processor actually needs them.

Figure 1 presents a block diagram of the eMIPS processor organization. The base datapath pipeline stages, general purpose register file and memory interface match those of a 'classic' CPU [5] and are depicted in lighter color in the diagram. These pipeline stages constitute the Trusted ISA or TISA, the core portion of the architecture that is required for initial operation and to provide a level of trust in the functioning of the processor. These blocks cannot be removed or disabled and must be present at startup of the system. These blocks constitute the fixed portion of the architecture and include all resources that are of a security sensitive nature, such as the system coprocessor. The TISA also includes all the facilities for self-extension, including instructions for loading, unloading, disabling and controlling the unallocated blocks in the microprocessor. At a functional level the pipeline blocks operate similarly to a 'classic' CPU design, except their interconnections with respect to each other and other blocks differ.

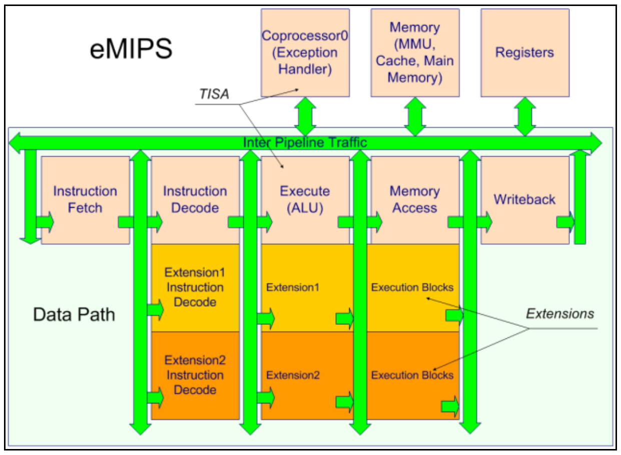

Figure 1: eMIPS Block Diagram

A simple Extension such as those depicted in darker colors in Figure 1 includes an Instruction Decode (ID) stage and an Execution stage that can span to the Memory Access stage of the datapath. This allows the extension to perform dual cycle operations without stalling the normal CPU pipeline. To perform multi-cycle operations it is possible to modify the processor control flow by stalling the TISA and maintaining ownership of all the pipeline resources. Multi-cycle operations are needed for instance to access memory. Memory accesses go through the MMU and are therefore confined within the current process' address space. Privileged-mode resources such as the system coprocessor-0 registers are not usually accessible to an Extension, unless it is owned by privileged-mode software. The extension has otherwise access to all other non-privileged resources of the executing process, such as registers and memory. The extension sees each instruction as it enters the ID phase, its program counter, the address of each memory access and the value exchanged with memory. If system software allows it, it can claim ownership of regular instructions in addition to extended instructions. The Extension can provide a non-sequential next-PC, i.e. one that differs from the following instruction, and alter the program flow.

An Extension is often a mean to add computational capabilities to the processor, but other uses are possible. For instance, Extensions can be used to realize dynamically-loaded on-chip peripherals [14]. In this work, we used the Extension interface to exploit the FPGA partial reconfiguration feature. With it we have created an Extension that adds remote, JTAG-like debugging support to the processor. The approach is usable with other dynamically extensible processors and soft-cores, and possibly to debug more than just software. We could imagine, for instance, using partial reconfiguration to dynamically load a hardware monitor such as ChipScope [23] to help debug hardware designs.

IV. eBUG OVERVIEW

The debugging support provided by eBug is actually realized by two separate and communicating components; a software component (*emips2gdb*) and a hardware component (the eBug extension). The two components cooperate in providing the necessary support for remote debugging of applications running on the eMIPS system. We minimized the size of the hardware component by moving functionality into the software component, provided the performance was not impacted. For instance, it is the software component that implements the protocol required by the client debugger, such as the *remote* protocol in the case of the GDB client. The components are somewhat independent of each other. For instance, it is possible to use another debugger client protocol simply adding a class to the software component, without changing the hardware one.

The software component is depicted as the block *emips2gdb* in Figure 2 and Figure 3. It is implemented as a single application program, running under the host PC's operating system. As further explained in Section V, it acts as an interface between a PC host running a debug client like GDB and a remote eMIPS target. The same program is used, whether the target is an actual hardware eMIPS FPGA implementation (Figure 2) or an eMIPS simulation model (Figure 3) running within the Giano simulator [9, 2].

The hardware component is implemented as a Verilog module that can either be synthesized separately as an Extension (block "eBug Extension" in Figure 2) or loaded together with the rest of the eMIPS modules and peripherals inside the Giano simulator (Figure 3). This component was developed as an eMIPS extension in order to achieve:

1) *Area reuse*: The area used by eBug is used only when an executing program is being debugged. eBug uses only one of the available Extension slots. When a debugging session is not needed the extension slot can be used for other purposes.

2) *Security*: The eMIPS processor can dynamically enable/disable individual extension slots, without reloading the Extensions in them. This feature can be used to activate the eBug extension only when the process being debugged is scheduled by the operating system. In this way, other processes running on the

system cannot be affected by the debugger. The debug client commands affect only to the state (registers, memory) of the process that owns the extension. eBug accesses registers and memory using the extension interface instead of a physical level channel like JTAG. This gives full control to the target operating system and prevents undesirable accesses to other resources by the debug client.

3) *Extensibility*: The eBug hardware component is intended as an extensible Extension. The design makes it simple to add other debugging features to the base modules. In this way, eMIPS is not limited to a fixed debug hardware support but, based on the user needs, it can evolve and provide more complex functionalities. The only constraint is the maximum area that an extension can take. Section VII shows some possible enhancements to the base eBug hardware support.

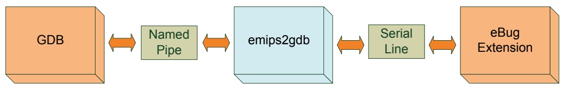

Figure 2: Connection to Hardware

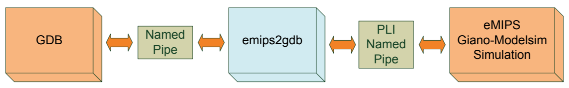

Figure 3: Connection to Simulation

V. THE EMIPS2GDB SOFTWARE COMPONENT

The eBug software component is realized in the *emips2gdb* application program. As shown in Figure 2 and Figure 3, it is possible to connect the debugger to an actual eMIPS hardware implementation as well as to a Giano simulation session, using the same *emips2gdb* program. In the first case *emips2gdb* uses an actual serial line, in the latter case it uses a PLI-based [15] interface that simulates the transmit and receive pins of the UART inside a C-model. A typical eMIPS debug session starts by running *emips2gdb* to create a server for GDB on one side and, according to the user request, a connection to a serial port or to a named pipe on the other side. The GDB debugger is then started and connected to *emips2gdb,* who acts as the remote target. Once the debug session is set, *emips2gdb* translates the GDB commands into the simpler protocol used by the eMIPS eBug Extension and performs the requested operations.

Emips2gdb currently supports GDB as the client debugger but other debuggers, like WinDbg [16], can be supported simply by adding a class implementation that translates the new debugger's remoting protocol into the protocol used by the eBug extension.

7	6	5	4	3	2	1	0
fspecial	nReg					opcode	
option				opcode			

Figure 4: Command byte formats

The GDB *remote* protocol is rather verbose and it is not area-efficient to parse it directly in hardware. To tackle this issue *emips2gdb* translates it into a more easy-to-decode protocol. Using this protocol *emips2gdb* can:

- Suspend and Resume the processor when the process that owns the debug extension is running,
- Read and write eMIPS registers,
- Fetch and Store values from and to memory.

Using these basic operations the debugger can perform more complex ones, such as single stepping, inserting software breakpoints and realizing software watchpoints. Note that, as previously explained it is also possible to add hardware support both for breakpoints and watchpoints, or other functionalities using additional basic operations. Section VII expands on this notion.

TABLE I:BASIC EBUG COMMANDS

opcode	option	Operation	Bytes returned
x00	N/A	Read from an eMIPS register	4
x01	N/A	Write to an eMIPS register	1 (Ack)
010	0x0-0x1F	Fetch byte from memory	variable
011	0x0-0x1F	Store byte to memory	1 (Ack)
110	00000	Suspend	1 (Ack)
110	00001	Continue	1 (Ack)
111	-----	Future Expansion	-----

The *emips2gdb* protocol is a stream of bytes that always begins with a command byte. As shown in Figure 4, the command byte can have two possible formats. The first format uses three fields and is used to access the eMIPS registers. The second format uses two fields and is used for memory and control operations. In both formats the opcode field alone identifies the action to be performed. The current set of legal opcode values is depicted in the first column of Table I. The second column shows the range of values for the option field, if applicable. The last column shows the number of bytes expected in the eBug response.

A. *Control Operations*

To start debugging, the first step is to connect GDB to the *emips2gdb* server. Once the connection is established, *emips2gdb* sends a Suspend byte to the debug extension to force eMIPS to idle. When eMIPS is stalled an acknowledge byte is sent back to *emips2gdb* and the eMIPS resources can be managed by GDB.

When a Continue command is issued, *emips2gdb* sends the corresponding command byte for putting eMIPS in the running state and waits for a session restart indication from eBug. This can be required, for instance, by the execution of a break instruction previously inserted by GDB.

B. *Register Operations*

A register operation is indicated by bit one of the command byte being zero. In this case, bit zero indicates whether a read or a write is desired. The remaining bits, i.e. the *fSpecial* bit and the *nReg* field in Figure 4, are used to identify an accessible eMIPS register.

Once a register Read is recognized, the eBug extension does not wait for any other bytes from the serial line. It gets the value of the desired eMIPS register from the TISA, according to the fSpecial and nReg fields. Once the value is retrieved, the four bytes are sent back to the *emips2gdb* component.

If the command byte specifies a register Write operation, the eBug extension waits to receive the register value from *Emips2gdb*. Once the value is received and stored to the requested eMIPS register, an acknowledge is sent back to *emips2gdb* to notify that the eMIPS state has changed.

C. Memory Operations

Emips2gdb sends a variable number of bytes to the eBug extension when the debugger wants to access the eMIPS memory subsystem. The first is the command byte. The number of bytes that follows depends on the value of the command byte. The command byte for Fetch and Store operations has a three bit opcode. The remaining 5 bits, i.e. the option filed, can have two sets of values:

- 0: The two bytes that follow indicate the size of the memory block that is to be read or written. A maximum block size of 64KB can be processed in a single transaction. In reality, the GDB remote protocol traces show that GDB uses a maximum block size of less than 400 bytes.
- 1-31: This is the size of the memory block. The four subsequent bytes define the starting address of the memory transaction.

In the case of a Read operation, *emips2gdb* does not send any more bytes and waits for the response from the eBug extension. After the last memory value is sent the transaction is concluded. No additional Acknowledge byte is sent.

In the case of a Write operation, *emips2gdb* sends the bytes to be written to memory, starting at the address already specified. The eBug extension stores the data to memory and then sends an acknowledge to conclude the transaction.

VI. THE EBUG HARDWARE COMPONENT

The eBug extension is not a typical eMIPS extension. It does not execute any extended instruction and does not perform any real computational task. It does take control of the processor if one of the following two conditions occurs:

1) A *break* instruction is in the ID stage, or

2) The client debugger asks to Suspend the process that owns the eBug hardware extension.

In these cases eBug stalls the TISA pipeline and takes control of the processor. This list is the minimal one implemented in the base eBug version. Other conditions are possible if/when other features are added, for example with hardware breakpoint/watchpoint support. Currently eBug only stalls the TISA before any trap occurs; it prevents the processor from ever trapping on a break instruction. If required, the extension interface has provisions for causing traps as well. eBug relinquishes control back to the TISA if one of the following two conditions occurs:

1) The operating system schedules another process, or

2) The client debugger issues a Continue command.

In all other respects, the eBug design follows the structure of any other eMIPS extension. As depicted in Figure 5, the eBug extension is composed of two modules: *ext_debug_control* that deals with the TISA pipeline interface and *Top_debug* that manages communications with the registers and the memory interfaces. These two modules are described in the following subsections.

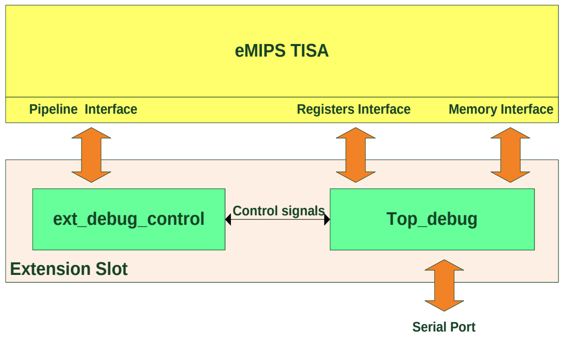

Figure 5: eBug External Interfaces

A. Interface to the Pipeline Arbiter

The eMIPS processor can execute both standard MIPS instructions [7] and extended instructions (see Section III). In the former case the execution is usually delegated to the base datapath, in the latter case the extensions are always responsible for the instruction. When an instruction is in the ID stage, both the TISA and the extensions can recognize it by lowering the recognized instruction (RI) signal. The eMIPS pipeline arbiter then decides to give the control to the TISA or to one of the extensions that claim to recognize it. When collisions occur, a priority scheme establishes the pipeline owner. Normally, the TISA has priority over the extensions but individual slots can be assigned higher priority and therefore override the TISA.

Using this mechanism it is possible for eBug to request a stall of the processor when a *break* instruction is encountered. Notice that this prevents the TISA from issuing a software trap, which would change the state of the processor and the register contents. The same mechanism is used if the debugger client sends a Suspend command, i.e. when it first tries to connect to eMIPS. In the latter case, the eBug extension unconditionally recognizes the instruction in the subsequent pipeline cycle. Notice that the instruction is therefore not executed, execution will restart from the current PC. The suspension mechanism must also deal with an issue specific to the MIPS architecture [7]. The MIPS processor uses delay-slot instructions, an instruction that immediately follows a branch but is executed as part of the branch itself. To simplify the design of eBug we implemented a mechanism that avoids stalling the processor when a delay slot instruction is in the ID stage. In this way the extension can always use the correct restart PC value.

Figure 6 depicts the handshaking signals between eBug and the pipeline arbiter. After the extension lowers the RI signal, the pipeline arbiter asserts the enable (EN) signal notifying the extension that its instruction was allowed. At the following positive edge of the pipeline synchronization clock (PCLK), the extension must release RI, setting it high. EN is also deasserted. The extension can now access the pipeline resources for multiple clock cycles (as eBug does) by asserting the acknowledge (ACK) signal. The pipeline arbiter grants control to the extension raising the GR signal. The processor is now stalled. To release the processor the extension must de-assert ACK.

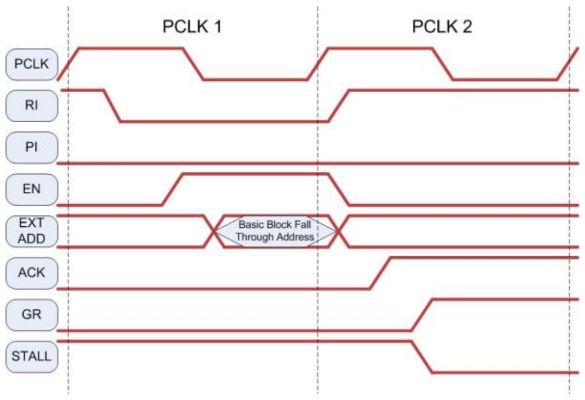

Figure 6: Taking Control of the Pipeline

All these tasks are performed by the *ext_debug_control* module. This module interfaces to the pipeline arbiter and to the *Top_debug* module. More specifically, it interfaces to *main_fsm* (see Figure 7), a sub module of *Top_debug*. As explained in more details in the next subsection, one of the tasks of *main_fsm* is to support communication with *emips2gdb*. Every time a *break* instruction is in the ID stage a signal (break signal in Figure 7) is asserted and *main_fsm* in turn communicates it to *emips2gdb*, to restore the debugging session. Similarly, when *emips2gdb* sends a Suspend command *main_fsm* sends a signal (suspend signal in Figure 7) to take control of the eMIPS resources. Once the processor is stalled, the *ext_debug_control* module finite state machine sends an acknowledge (suspend_Ack in Figure 7) back to *main_fsm*.

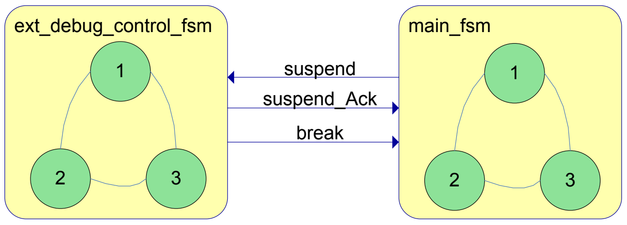

Figure 7: Suspension Protocol

B. *Datapath*

As shown in Figure 5, the module *Top_debug* is responsible for communication with the host PC over the Serial Port, for the register and memory interfaces, and it links with the *ext_debug_control* module. Internally, it is composed of two modules: the uart and the debug_core. The uart module is an implementation of the RS232 serial communication link, with a compile-time configurable baud rate. It lacks runtime configurability to simplify as much as possible the design. This leads to a very small area footprint of about 50 slices. Should a different baud rate or serial parameters be needed it is simpler and more effective to create a new eBug instance. To limit the serial line bottleneck effect, we use a default value of 115,200 baud.

The debug_core module is the main control center for the whole extension. The datapath communicates with the uart module and is interfaced with the TISA resources, namely the registers and memory subsystems. The design of the datapath strives to minimize the area utilization. Pipelined registers and other critical path reduction techniques are not used. There are only five registers in this implementation:

- *InReg* is used to store the command byte from the uart module.
- *fw_reg* is used to pack four bytes into a 32 bit word. This register is used for write operations to registers and memory.
- *PC_Break* is used to store the address of the instruction currently in the ID stage. Once a debug session starts PC_Break can be only changed by the debugger. This register is an image of the actual PC. When the program is restarted this is the value used to restart execution.
- *mem_addr* is used to store the start address for memory operations.
- *num_byte* stores the number of bytes requested for a memory operation.

The datapath additionally includes seven multiplexers, two decoders and a counter of the number of bytes read or written in a memory operation.

C. *Control*

The control part of debug_core is implemented using three finite state machines: main_fsm, registers_fsm and memory_fsm. The finite state machine implemented by the main_fsm module handles synchronization with the ext_debug_control_fsm module and communication with *emips2gdb*, as previously described. When in the idle state only two possible events can take place: a break instruction is executed, or an *emips2gdb* connection is requested. In the first case the debugger must be notified of the break instruction. With the processor already stalled, a notification is sent to *emips2gdb* to signal that the processor is waiting for debugging. In the second case, main_fsm assert the suspend signal to request a processor stall. In either case, the finite state machine then goes into a state waiting for an *emips2gdb* command. Once a command is received and recognized, for example for a "register access" operation, the state machine performs the operation and eventually comes back to this state. If the command is a Continue then main_fsm returns to the idle state, after notifying ext_debug_control_fsm to releases the TISA pipeline. If an incoming command is not recognized, main_fsm responds with a zero value byte and then comes back to waiting for another *emips2gdb* command. An interesting case is if *emips2gdb* crashes while the processor is stalled and main_fsm is waiting for a command. If *emips2gdb* subsequently reconnects it sends a new Suspend command to eBug, who then replies with an acknowledge byte. The debug session is then correctly resumed.

The state machines registers_fsm and memory_fsm implement the eMIPS compliant protocol to access the TISA registers and the memory subsystem. Memory_fsm is much more complex than registers_fsm because the *emips2gdb* protocol for memory operations is a variable-length byte stream protocol. The state machine must control the flow of data through the datapath, correctly store the initial memory address and the number of bytes involved in the memory transaction, and eventually obey the memory subsystem protocol.

VII. eBug Extensibility

The eBug design is meant to be easily extended. Adding support for new features can potentially require modifying both the hardware side and the software side. It is desirable that only a well identified subset of modules requires modification to add new features, and that the design structure can be preserved.

In this section we present two examples of extensions to eBug. The first is a set of changes made to realize breakpoints and watchpoints in hardware. Hardware support for watchpoints provides performance gains that strongly affect the user's experience. The second is a connection to the eMIPS extensions generated by P2V [6]. Using eBug in concert with P2V provides very sophisticated, high-lever debugging facilities which are especially useful in the case of embedded and real-time applications.

A. Hardware watchpoints

The debug target can dynamically declare to the GDB debugger that hardware watchpoints and/or breakpoints are supported. In this case, GDB uses different commands in its remote protocol to notify the target of the insertion or deletion of a breakpoint or watchpoint. To support these operations, the emips2gdb protocol was extended using the opcode for Extended operations (111) and choosing an appropriate value for the option field. Additional information is sent to the eBug extension following this command byte. The next byte is called the ControlByte, and the encoding is shown in Table II.

TABLE II: ControlByte

Bits	Meaning
3-0	Slot number
4	Watchpoint (1) or Breakpoint (0)
5	Enable(1) or Disable (0)
7-6	Access (00-write, 01-read, 11-all)

The least significant four bits hold the hardware slot number to be used. When GDB inserts or deletes a breakpoint or a watchpoint, it identifies it only by its address. If this information is sent directly to the hardware a complex logic would be needed to identify the corresponding hardware slot. To avoid the extra costs in area we modified *emips2gdb* instead, adding a simple data structure to the class that implements the protocol. This table keeps track of the address and all the other information related to the hardware slots, and it is used by software to translate an address in a slot number. When *emips2gdb* initially makes a new connection to eBug it synchronizes this data structure with the hardware slot information.

Bit 4 in the ControlByte is used to indicate to eBug if an insertion of a watchpoint or a breakpoint is requested. In the first case, bits 7-6 are used to indicate the watchpoint type, since eBug can selectively watch for read or write accesses (or both). Finally bit 5 is used to enable or disable a slot. When a slot is disabled only the slot field and bit 5 hold significant information, the other bits are not used. If a slot is enabled, *emips2gdb* follows the ControlByte with a 4 byte address.

B. Adding features via other extensions

eBug can also be extended by leveraging other, separately developed eMIPS extensions. One example is the extensions generated by the P2V compiler [6]. The PSL-to-Verilog (P2V) compiler can translate a set of assertions about a block-structured software program, expressed in the simple subset of the Property Specification Language PSL, into an eMIPS extension that observes the program's execution and validates the assertions. PSL is based on the LTL temporal logic, and can therefore express the complex patterns that define the behavioral correctness of the software program in a natural and compact form.

As a simple example, suppose we want to check if a program's variable is within a desired range, but without recompiling and without altering the program's temporal behavior in any way. Note that currently P2V is the only system that can do this. It does so by creating a specialized eMIPS extension that passively monitors the program execution. If the variable is assigned an illegal value, the P2V extension will signal the violation in some unspecified way. For instance, it could assert a trap and let the operating system manage it according to its own policies. There are two limitations, however, in this approach. In the first place, it is not possible to observe the state of the system at the exact moment when the assertion is violated, but only later, after the operating system's trap handler has captured it and only limited to what software can self-observe. In the second place, we lack an explanation for why the program attempted the illegal assignment.

We can easily overcome these limitations with eBug. Rather than using the trap signal, P2V can insert a *break* instruction in the ID pipeline stage. This produces exactly the same trap behavior when eBug is not present. When eBug is present, it takes control of the processor in the actual moment the failure occurs, and without otherwise affecting the state of the system. The failure is reported to the debugger and the user can explore the system's state at length and discover the reason for the erroneous behavior.

We can go further. P2V is implemented in Python, using an interpreter. We can connect the GDB command line interpreter to the Python interpreter, and generate the P2V extensions on-the-fly, while debugging the program. The user types the PSL assertions about the running program while it is suspended, a new extension is created and loaded in a separate extension slot, and execution is then resumed. An interesting side-effect of this approach is that the user can produce and test a new/additional set of formal declarations about the program's properties as a natural result of debugging it. This has the additional benefits of quantifying the extent of the testing actually performed, and of creating input data for even more sophisticated program analysis tools, such as theorem provers and symbolic execution. In the first release, eBug and P2V must be compiled together because the ML401 implementation of eMIPS only supports one extension slot.

VIII. RESULTS

In this section we show two separate measures that quantify the performance of eBug. In both cases, we analyze the effects of adding one single feature, namely hardware watchpoints. We first look at the area and frequency results in the synthesis of different implementation of the eBug extension. This quantifies the impact of the feature from a hardware point of view. We then measure the changes in response time, from the user's point of view, when adding the feature to eBug.

A. Synthesis Results

All designs were implemented using a Xilinx ML401 prototyping board. The board is built around the Xilinx Virtex4 device, model XC4VLX25. To synthesize, implement and build the configuration files we used the Xilinx ISE version 8.2.01i, with the partial reconfiguration overlay applied. The synthesis results are summarized in Table III and Table IV. The first row in the two tables corresponds to the basic design, where hardware support for watchpoints is missing and must be realized in software. Additional rows correspond to designs that support two, four and eight hardware watchpoints, respectively. Table III details the results in area and maximum frequency for the various designs.

TABLE III: SYNTHESIS RESULTS

	Area optimization		**Speed optimization**	
	Area	***f(MHz)***	***Area***	***f(MHz)***
SW WP	273	112,96	316	175,04
2 HW WP	359	88,51	381	175,00
4 HW WP	422	89,70	451	174,93
8 HW WP	568	61,13	603	174,61

When optimizing for area, the maximum frequency of the design decreases dramatically against an increasing number of watchpoints, without providing an equally significant saving in area. Table IV stresses this point by comparing the percentages in area savings and frequency reduction of the first column in Table III against the second column. The best tradeoff is given by the speed optimization option, confirming that the design was already targeted towards a small area footprint.

TABLE IV: AREA VERSUS SPEED TRADE-OFFS

	% Area Savings	**% Freq. Reduction**
SW WP	13.6	35.47
2 HW WP	5.77	49.42
4 HW WP	6.43	48.72
8 HW WP	5.80	64.99

The extension slot in the first eMIPS implementation has an available area of about 1,300 slices. Extrapolating on the trend visible in Table III, we can estimate that eBug on eMIPS can provide a maximum number of about 27 hardware watchpoints. Most users typically set one or two watchpoints and then move them around as the debugging session evolves. For very large applications it can nonetheless be useful to be able to observe many more program variables at once. When hardware watchpoints are not desired eBug uses only 21% of the available extension slot, leaving about 80% of the area for other uses. Simple P2V assertions can fit comfortably in this area.

B. Response Time

We measured the response time of the debugger client in a simple interactive test, and compared the software and hardware watchpoint implementations. The goal was to quantify the impact of the added feature from the point of view of the user. The test was performed using a simple C program that loops incrementing a variable and printing a message on the console, as follows:

```
while(1){
        i=i+20;
        Puts("Ciao!\n");
        PutWord(i);
}
```

We instructed GDB to insert a watchpoint for the variable *i* by issuing a "watch i" command while the program was suspended at some arbitrary loop iteration. We then took the time from a "continue" command to the subsequent suspension with the new variable value. Using this simple example allows software based watchpoints to work well because there are few machine instructions between two successive updates of the variable i. In this way we can evaluate the minimum speedup achievable using hardware watchpoints. Measurements were repeated five times and the average is reported in Table V. There was very little variance in the measured results. The test was repeated using two different machine configurations. The Machine1 setup is a single machine with a dual-core Intel Centrino Core2/6600 processor operating at 2.4GHz and running the Windows XP SP2 operating system. An ML401 board is connected to the machine using a serial cable. The Machine2 setup includes two separate machines, one running the GDB debugger and the other the *emips2gdb* server, connected in turn to the ML401 board using a serial cable. The first machine uses a dual Intel Xeon processor operating at 2.8GHz and running the Windows Server 2003 SP2 operating system. The second machine uses an old Intel Pentium3 processor operating at 800MHz and running the Windows 2000 SP4 operating system.

TABLE V: USER-PERCEIVED PERFORMANCE GAIN

	Software	**Hardware**	**Speedup**
Machine 1	272 sec	1,1 sec	247
Machine 2	44 sec	0,4 sec	110

The performance difference between the two machine setups is due to operating system scheduling issues (i.e. in the case of Machine1) rather than to eBug itself. In all cases, the CPU load of the GDB and *emips2gdb* processes is at most 1%.

The 100-fold speedups provided by the hardware watchpoints are impressive, but of more practical importance are the absolute values. A user is unlikely to use a feature that costs almost a minute per loop iteration, whereas a cost of less than a second makes it quite feasible to use it extensively. Note that the use of software-based watchpoints becomes prohibitively expensive when used with an embedded RTOS that is compiled-in with the application, because the debugger in this case must single-step through potentially large sections of system code. In this case, the JTAG-like feature also proves very useful since the user can now single-step through previously undebuggable sections of code, such as interrupt service routines, trap handlers and other low-level operating system code.

IX. Conclusions and Future Work

We have introduced eBug, a secure, extensible and efficient software debugging tool for dynamically reconfigurable architectures. A first release of eBug has been implemented as a hardware Extension for the eMIPS processor and provides hardware-level, in-process debugging support to a software debugger such as GDB. The approach is usable with all processors and soft-cores that expose the registers and memory subsystems to a FPGA reconfigurable slot. The system is split into a hardware and software component, with much of the functionality realized in software to minimize area and maximize flexibility. eBug can work in concert with the P2V zero-overhead assertion checker to realize a more advanced software debugging environment, especially for embedded and real time systems.

eBug was conceived as an Extension rather than a fixed hardware module to achieve three main goals: area reuse, security and extensibility. eBug uses the area already devoted to an Extension slot on eMIPS, without changes to the base processor pipeline. When not in use eBug is not present on the chip and its area is therefore reused, e.g. in the final product. eBug is security-aware because it can only access and modify the status of the process that owns it, privileged or not that it might be. eBug is extensible because it makes it easy to add new features without changing the whole design or the interface to the processor. When a new feature is added only the eBug extension must be regenerated. We proved this point by adding hardware support for watchpoints and breakpoints to the basic design, and measuring the difference in terms of area occupation, speed performance and improved debugging capabilities.

Because of the extensibility feature, adding new features to eBug is straightforward. For instance, it is easy to implement a value-based watchpoint that observes the actual data written to a program variable, rather than just the address. Adding hardware support for variable size watchpoints can be achieved by changing the watchpoint logic to use two watchpoint slots and look at an address range rather than a single address mask. This allows monitoring more complex data types like C arrays, structures and C++ classes. Multiple conditions could be matched in hardware; one match can be the enabler for subsequent ones. Possible additional features are not limited to the debugging aspects. Ethernet or USB interfaces could replace the simple but slow serial line currently used. Other communication protocols could be added to the software component of eBug. Additional functionalities, such as tracing and performance profiling, could be added by modifying both the hardware and the software components.

References

[1] Dean, J., et al. *ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors*. MICRO, 1997.

[2] Forin, A., Neekzad, B., Lynch, N., L. Giano: *The Two-Headed Simulator*. Microsoft Research Technical Report MSR-TR-2006-130, September 2006.

[3] GDB: *The GNU Project Debugger.* Available at http://www.gnu.org/software/gdb/

[4] Graham, S.L., P.B. Kessler and M.K. McKusick. *gprof: a Call Graph Execution Profiler.* SIGPLAN Symp. on Compiler Construction, pp. 120-126, 1982.

[5] Hennessy, J. L., Patterson, D.A. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann Publishers, San Francisco, CA. 1998.

[6] Hong Lu, Alessandro Forin, *P2V: An Architecture for Zero-Overhead Online Verification of Software Programs*, Workshop on Application Specific Processors, WASP 2007

[7] Kane, G., Heinrich, J. *MIPS RISC Architecture.* Prentice Hall, Upper Saddle River, NJ. 1992.

[8] Leon Processor user manual. Available at http://www.gaisler.com/cms/

[9] *Microsoft Giano* at http://research.microsoft.com/downloads/ and http://www.ece.umd.edu/~behnam/giano.html

[10] Pittman, R., N., Lynch, N., L, Forin, A. *eMIPS, A Dynamically Extensible Processor* Microsoft Research Technical Report MSR-TR-2006-143, October 2006.

[11] Pittman, R., N., Forin, A. *Microsoft eMIPS Release v1.0* Microsoft Research, Fall 2007.

[12] PowerPC processor in Xilinx FPGAs. Available at http://www.xilinx.com/

[13] Sparc processor architecture. Available at http://www.sparc.org/

[14] Sukhwani, B., Forin, A., Pittman, R. N. *Extensible On-Chip Peripherals* Microsoft Research Technical Report MSR-TR-2007-120, September 2007.

[15] Sutherland, S. *The Verilog PLI Handbook*, 2nd ed. Kluwer Academic Publishers, Norwell, MA. 2002.

[16] WinDbg multipurpose debugger. Available at http://www.microsoft.com/whdc/devtools/debugging/default.mspx

[17] Xilinx Embedded System Tools reference. Available at http://www.xilinx.com/ise/embedded/edk91i_docs/est_rm.pdf

[18] Xilinx Microblaze Debug Module MDM. Available at http://www.xilinx.com/bvdocs/ipcenter/data_sheet/opb_mdm.pdf

[19] Xilinx Microblaze soft processor core. Available at http://www.xilinx.com/products/ipcenter/micro_blaze.htm

[20] Zagha, M., B. Larson, S. Turner, and M. Itzkowitz. *Performance Analysis Using the MIPS R10000 Performance Counters.* Supercomputing, Nov. 1996.

[21] Zhang, X., et al. *System Support for automatic Profiling and Optimization.* Proceedings of the 16th Symposium on Operating Systems Principles, 1997.

[22] Zilles, C.B. and G.S. Sohi. *A Programmable Co-processor for Profiling.* International Symposium on High-Performance Computer Architectures, 2001.

[23] Xilinx: *ChipScope Pro Software and Cores User Guide*. Xilinx Inc., October 2005, Available at http://www.xilinx.com/ise/verification/chipscope_pro_sw_cores_8_2i_ug029.pdf