eBug: Debugging Extensions for the eMIPS Dynamically Extensible Processor

Giovanni Busonera, Alessandro Forin, Richard Neil Pittman
Microsoft Research

November 2007

Technical Report
MSR-TR-2007-155

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

eBug: Debugging Extensions for the eMIPS Dynamically Extensible Processor

Giovanni Busonera, Alessandro Forin, Richard Neil Pittman
Microsoft Research

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 P2V 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.

1 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 document we introduce eBug, a flexible, low overhead, security aware and easily extensible debugging support realized for the eMIPS processor [11]. 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 actual debugger client and the eMIPS processor itself.

eBug creates little if any overhead. It is entirely optional, it uses very little area resources in one 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. In this paper we show how 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 remainder of this document is structured as follows. Section 2 summarizes the related work. Section 3 introduces the eMIPS processor. Section 4 gives an overview of eBug, and the eBug software and hardware components are then described in detail in Section 5 and Section 6. Section 7 shows how to use eBug in a practical setting. Section 8 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 9. Future work and conclusions are presented in Section 10.

2 Related Work

On-chip support for software debugging can be found in the Leon Processor [8], an open source 32-bit RISC CPU jointly designed by Gaisler Research and the European Space Agency. Leon is a Sparc V8 [14] instruction set compliant microprocessor. 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 [22] and on the PPC hardcore [13]. The XMD (Xilinx Microprocessor Debugger) [20] 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 [21]. 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 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.

3 The eMIPS Processor

eMIPS [11] is a dynamically extensible microprocessor developed by the Microsoft Research Embedded Systems group. Using the extensibility features, 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.

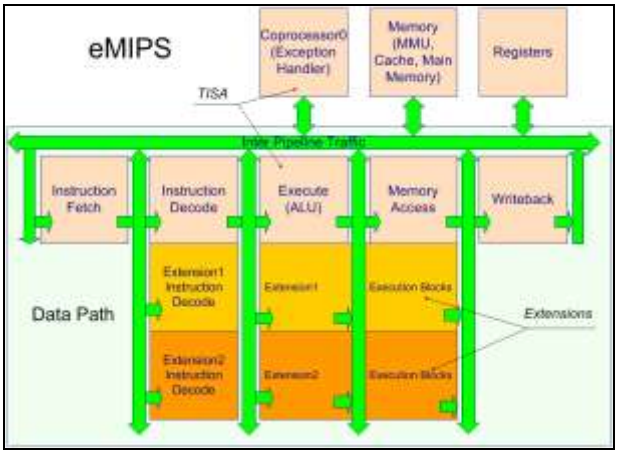


Figure 1: eMIPS Block Diagram

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 affecting 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 additions 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 also possible. Extensions can be used to provide any kind of service, for instance to realize dynamically-loaded on-chip peripherals [17]. In this work, we used the Extension interface to create an Extension that adds remote, JTAG-like debugging support to the processor.

4 eBug Overview

The debugging support provided by eBug is realized by two 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 strived to minimize the size of the hardware component, moving much 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 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 5, 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 [10, 2].

The hardware component is implemented as a Verilog module that can be synthesized either 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 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 any other process running on the system cannot be affected by the debugger. 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 lower level channel like JTAG. This gives full control to the target operating system and prevents unwanted accesses to processor 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, depending 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 8 shows some possible enhancements to the base eBug hardware support.

Figure 2: Connection to Hardware

Figure 3: Connection to Simulation

5 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 GDB 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 [18] interface that simulates the transmit and receive pins of the UART inside a C-model. A typical eMIPS debug session starts with first 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 executed 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.

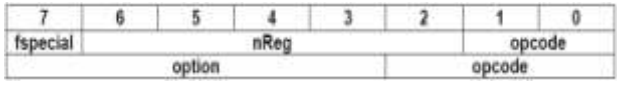


Figure 4: Command byte formats

Emips2gbd currently supports GDB as the client debugger but other debuggers, like WinDbg [19], can be supported quite simply by adding a class implementation that translates the new debugger's remote protocol into the serial protocol used by the extension.

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 8.1 expands on this notion.

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	-----

Table 1: Basic eBug commands

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 1. 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.

5.1 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.

5.2 Register Operations.

A register operation is indicated by bit one of the command byte being zero. In such a 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 as specified in Table 2.

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 in big-endian order to the emips2gdb application over the serial line.

fSpecial	nReg	Register
0	0-31	GPR file register number
1	0	PC
1	1	hi
1	2	lo
1	3	sr
1	4	bad
1	5	cause
1	6	fsr
1	7	fir

Table 2: Register file encoding

If the command byte specifies a register *Write* operation, the eBug extension waits for the register value to be written. Emips2gdb sends the expected four bytes in big-endian order. Once the value is received and stored to the requested eMIPS register, an acknowledge byte (0xFF) is sent back to emips2gdb to notify that the eMIPS state has changed.

Currently it is possible to perform both read and write operations on the general purpose registers and the PC whereas lo, hi and cp0 registers are read only.

5.3 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 (big-endian ordered) 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, no more bytes are needed.

The four subsequent bytes (big-endian ordered), 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 byte to conclude the transaction.

6 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 either case, eBug stalls the TISA execution and takes control of the processor. This list could change if/when other features are added, for example with hardware breakpoint/watchpoint support. Currently eBug only stalls the TISA before any trap occurs. If required, the extension interface has provision 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. Figure 9 details the internal structure of eBug and the relationships between the various modules. The top-level module (which must be called *extension0)*, is a wrapper that exposes all the available TISA signals to the extension main module (*debug_extension)*. This module is used in two different ways in synthesis and in simulation. In synthesis, it is the hard interface of the Extension and connects to the bus-macros that are the physical interface of the extension slot. In behavioral simulation, it is loaded along with the other TISA modules and directly interfaces with them. Notice that even though only the input signals actually needed are connected, all the output signals must be driven to their correct idle logical values.

The *debug_extension* module instantiates three modules. The first is the *reset_manager* module that deals with global reset management issues. The debug extension modules use an active high reset, whereas the TISA uses an active low reset; therefore, this module is a simple inverter of the TISA reset signal. The two other modules (*ext_debug_control* and *Top_debug*) deal with the TISA pipeline, with the registers and the memory interfaces and are depicted in Figure 5.

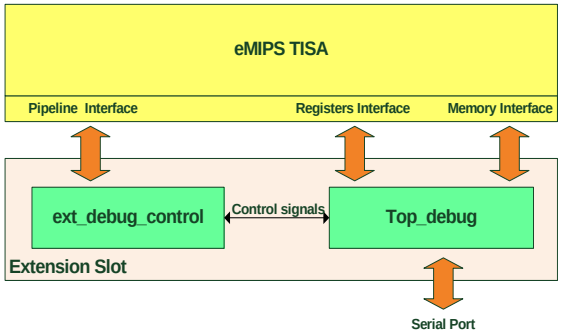


Figure 5: eBug External Interfaces

6.1 Interface to the Pipeline Arbiter

The eMIPS processor can execute both standard MIPS instructions [7] and extended instructions (see Section 3). In the former case the execution is normally delegated to the base datapath, in the latter case the extensions are responsible for the extended instructions. When an instruction is in the ID stage, both the TISA and the extensions can actually 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.

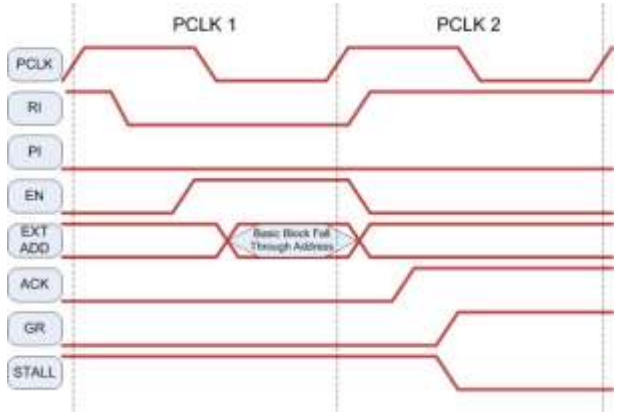


Figure 6: Taking Control of the Pipeline

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. For the meaning of the other signals please refer to eMIPS documentation [12].

All the tasks described so far in this section 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*.

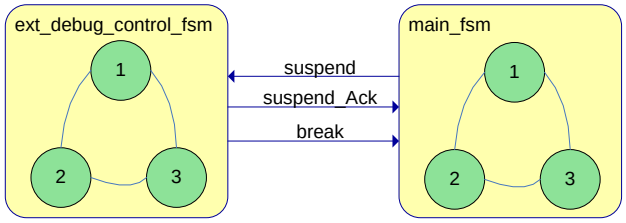


Figure 7: Suspension Protocol

The module *ext_debug_control* is composed of three sub modules:

- The finite state machine, implemented in *ext_debug_control_fsm*. A simplified diagram for this state machine is shown in Figure 11.
- The instruction decode module, which is used to recognize *break*, conditional branch and jump instructions.
- The nACK generator module.

The second module is responsible for the correct behavior of the system when a debugging session starts. The difficult case is when the ID stage holds an instruction located in a branch delay slot. The FSM of the *ext_debug_control_fsm* module lowers RI for the next ID stage instruction. Thus if a branch instruction is in the ID stage and a suspend signal is asserted, the processor is actually stalled when the delay slot instruction in the ID stage. When eBug releases the processor the execution would therefore normally restart from this instruction. Unfortunately, if the branch was taken the destination of the branch is now lost and the program control flow is altered. The instruction decode module is used to prevent this incorrect behavior. It generates a signal that delays the assertion of suspend if the instruction in ID is a conditional branch or jump instruction.

The third module is used to generate the nACK byte, which is sent to emips2gdb as a reply to an unsupported command and when a *break* instruction is encountered. Different nACK codes are used to indicate different kind of *break* instructions. Table 3 shows the nACK codes currently used by eBug.

Event	nACK
Command byte not supported	0
Breakpoint	0
Load software module	1
Unload software module	2
Other break codes	3

Table 3: nACK encoding

6.2 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 is depicted in Figure 10 and it is implemented in the *debug_dp* module. The upper side of the datapath communicates with the *uart* module and the lower side 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 five registers in this first 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. Bytes are expected in big-endian mode, i.e. the first is the most significant one. 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. Like *fw_reg* it is built from four bytes of big-endian ordered data.
- *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. The multiplexers are used as follows.

- *sel_addr* and *sel_m_byte*: used in memory operations. The first feeds the *mem_addr* register with the initial or with the incremented address. The second initializes *Addr_counter,* driven by the decoder *Dim_Block*. The initial value for *Addr_counter* depends on the *option* field of the command byte. If *option* is greater than 0 then this is the initial value. If it is equal to zero then the *num_byte* register is used instead.
- *sel_tisa_pc*: selects the path for updating the PC_Break register. This is either the current PC from the TISA or a new value from the debugger client.
- *sel_reg*: a decoder selects its output from the TISA register read or the PC_Break value.
- *sel_out*: feeds the *uart* transmit path with one of the registers read data, memory read data, ACK or nACK signals.
- *sel_byte*: used to serialize a 32 bit word in four bytes.

- *Mem_Addr[1:0]-1'b1*: selects the correct byte out of a 32 bit word read from memory, depending on its address. We subtract one from the last two bit of the address to simplify the finite state machine that manages the memory operations.

6.3 Control

The control part of *debug_core* is implemented by the *Debug_Control* module, 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. A simplified diagram of *main_fsm* is depicted in Figure 12. The complete diagram is shown in Figure 13.

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, an opportune nACK code is sent to emips2gdb to notify it 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 the "wait for emips2gdb commands" state. 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 (nACK) 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 ACK byte. The debug session is then correctly resumed.

The state machines *registers_fsm* and *memory_fsm* (detailed diagrams are shown in Appendix A) 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. The different phases are shown in the diagram with different colors. It is likely than in future eMIPS implementations both the register and memory access protocols will be modified to improve the performance. In that case *registers_fsm* and *memory_fsm* must be changed too. For example, in the current eMIPS implementation to read a general purpose register it takes four system clock cycles. In *registers_fsm* this value is known but it is parameterized; if it changes it is only a matter of changing the parameter declaration and to recompile eBug.

7 Structure and Usage Models

eBug is available in two slightly different versions. The first is for use with an actual hardware implementation of eMIPS whereas the second is for use with the Giano simulator framework. The main folder of the distribution is therefore divided in two subfolders: Implementation and Simulation. In the following subsections we describe the directory structure and the usage models of these two versions.

7.1 Implementation

7.1.1 Directory structure

- eMIPSv1: All TISA files (partial reconfiguration and non partial reconfiguration version) are located here
- eBug_HW: Source files of the eBug extension.
- eBug_SW: emips2gdb application folder.
- Bit Files: FPGA configuration files folder.
- Example: in this folder are located the DOWNLOAD.EXE and SERPLEXD.EXE applications, and a sample program that can be used to perform a test debug session.

7.1.2 How to use eBug

It is easier to start a debug session using the FPGA configuration files provided in the Bit Files folder. To rebuild the bitfiles, either with the partial or non partial reconfiguration flows, please refer to the eMIPS documentation [12].

The practical steps are:

1. Connect the secondary serial port to the Sparkfun RS232 Shifter board [15] using the expansion slots of the ML401 board, as shown in Figure 8. Connect one of the 3.3V power and ground pins from the J3 connector to the VCC and GND pins on the Shifter board. The red and black cables in Figure 8 are connected to the 3.3V power pair at J3.L14. Connect

the TX-O pin of the Shifter board to the J6 connector at pin 62 (green cable) and the RX-I pin to the J6 connector at pin 64 (yellow cable).

2. Configure the FPGA using the configuration file *mipspl_fpga3_base_routed_full.bit.* At the end of the configuration download both the TISA and the eBug extension are in FPGA and the boot loader is running. Make sure the option dip-switches are set to zero.
3. Open a console and go to the Example folder. Download the software application to the FPGA by typing: "*download.exe COM1: debug_extension_test.bin && serplexd.exe –n –r –s"*. This assumes that the primary serial line of the FPGA board is connected to COM1. Once the file is downloaded the program begins to run.
4. Open a new console and change the directory to eBug_SW. Then type: "*emips2gdb COMx:"* where *x* is the number associated to the PC serial port connected to the secondary serial line of the FPGA board (see point 1 above).
5. Open a new console and change directory to eBug_SW. Start the GDB debugger by typing: *"gdb.exe debug_extension_test"*. If needed, give to gdb.exe the absolute path. At the GDB prompt type: "*target remote \\.\pipe\eMips2Gdb"*. The debugging sessions should start. If the emips2gdb application is running on a different computer (say OtherPc) use the full path for the named pipe: "*target remote \\OtherPc\pipe\eMips2Gdb"*. The debugger will connect to eMIPS and stop the running program.

7.2 Simulation

7.2.1 Directory structure

- SIM: This is a blank folder used to create the Modelsim project.
- TISA_ICE: TISA files for simulation modified as follow:
    1. all assignments in sequential blocks are changed from blocking to non-blocking
    2. prefetching is disabled in the module memory_arbiter_giano.v
    3. added a file address_translation_ext.v to translate virtual address outgoing from extension (This feature should be integrated with a single address translation unit)
- eBug_HW: eBug extension source files. Files are the same of the implementation version. Only the datapath is different to be compliant with the Giano memory interface of the TISA.
- TB: The testbench file is located in this folder.
- PC_UART: The PLI based simulation model of the host PC serial line.
- PLI: in this folder are all source, include, libraries and make files to build the vpi2sl.dll.
- eBug_SW: emips2gdb.exe folder.
- Example: an example application to debug and a Giano configuration file are in this folder.

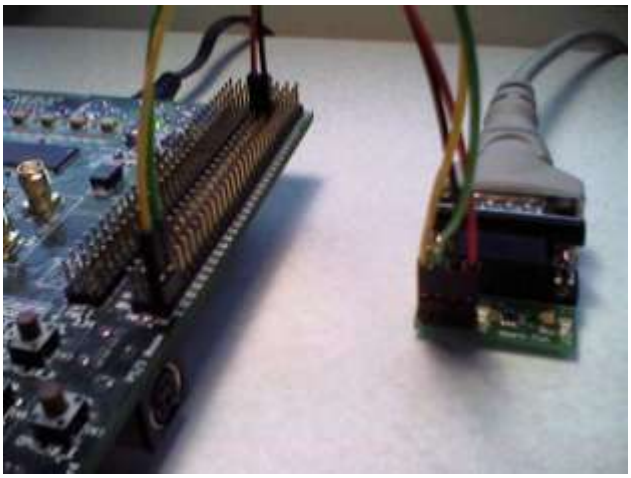


Figure 8: Cabling for the ML401 Board

7.2.2 How to use eBug

1. Create a new directory and copy all folders into it.
2. Create a Modelsim project in the SIM subfolder and add to the project all the source files present in the subfolders eBug_HW, PC_UART, TB and TISA_ICE except for the files decode.v and decoder.v (located in TISA_ICE) that must be copied into the SIM folder as well as vpi2sl.dll (file located in PLI\bin). Finally compile all the files with Modelsim. 82 files in total should compile without error.
3. Open a console and change directory to Example. Run Giano with the following command: "*giano.exe - Platform Ml401_ice2.plx GPIO::ValueAtReset 4 SRAM::PermanentStorage debug_extension_test.bin"*
4. Start a simulation in Modelsim with testMIPS as the testbench and including the vpi2g.dll and vpi2sl.dll as vsim -pli options. Then type "*run –all"* in the Modelsim console
5. Once simulation is running the named pipe EnnePiPipe66 is created. Open a new console, change directory to eBug_SW and type: "*emips2gdb \\.\pipe\EnnePiPipe66"*. A dialog box about RootBus

might appear, choose Ignore. Running emips2gdb on a different machine might slightly improve simulation performance.

6. Open a new folder and change directory to Example. Run GDB: "*gdb.exe debug_extension_test"*. Then connect to the remote target typing on GDB console: "*target remote \\.\pipe\eMips2Gdb"*.The program running on the eMIPS simulation model should be stopped and the debug session can be started.

It is possible to have a faster simulation by changing the baud rate of the PC_Uart and Debug UART modules. The default is 115,200 baud, the same used in the implementation version.

Using a different version of the simulator requires recompilation of the vpi2sl PLI. The vpi2sl.dll in the distribution is compiled for Modelsim 6.2g.

8 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. For example, in Figure 18 we show the modules that have been added (yellow boxes) and the modules that have been modified (red borders boxes) when implementing hardware breakpoints and watchpoints. The red lines connect modules in which only individual ports or instances were modified to keep them coherent with the rest of design.

In this section we present two examples of extensions to eBug. The first is an internal 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.

8.1 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. We selected the value 5'b00001, therefore the command byte used for enabling or disabling both a watchpoint and a breakpoint has the value 0x0F. 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 4.

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)

Table 4: ControlByte

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, big-endian ordered.

8.1.1 Datapath

The original eBug datapath was augmented as shown in Figure 19. The area highlighted in yellow is an instance of the *wbpoints_dp* module and provides new ports for the additional control signals. The *wbpoints_dp* module is composed of a control register (CR), a decoder and one or more *wp_bel* module instances according to the desired number of slots. The CR register is used to store the ControlByte sent by emips2gdb. The decoder selects the control signals and feeds them to the right slot, according to the slot number stored in the CR.

Wp_bel is the basic module that implements both the watchpoint and the breakpoint logic. Its diagram is depicted in Figure 20. It is composed of four registers:

- *Wp_reg* stores the address of the watchpoint or breakpoint. It is used with a comparator to assert an address match.
- *En_reg* is used as global enable. If it holds a low logical value the slot is disabled and no hit can occur.
- *Sel_addr_reg* stores the slot usage type (watchpoint or breakpoint); it is used for snooping on the address bus or the PC bus. It is also used as an enable signal for the watchpoint or the breakpoint enable logic blocks.
- *Wp_type_reg* stores the type of watchpoint (read, write and read/write).

The watchpoint and breakpoint enable logic boxes are used to enable or disable an address match. In both modules *en_reg* and *sel_addr_reg* act as enable signals. In addition, the watchpoint enable logic uses *wp_type_reg* and the snooped *write_enable* signal to consider the watchpoint type. The *bp_hit* and *wp_hit* output signals from the *wbpoints_dp* module are used to initiate the processor stalling handshake.

8.1.2 Control

The control modifications for the new feature include small changes to the *main_fsm* and the *ext_debug_control_fsm* modules and the addition of a new finite state machine. As shown in Figure 22, we added a new state to *main_fsm* to decode the new emips2gdb command. Figure 23 shows the changes in *ext_debug_control_fsm*. In the transition from the IDLE state to the RI_ASSERT state we now consider the *bp_hit* and *wp_hit* signals.

To perform the actual insertion or deletion of a watchpoint or a breakpoint we added the *wbpoints_fsm* finite state machine, shown in Figure 21. This finite-state machine manages the byte stream from the serial line to store data in the correct registers.

8.2 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.

9 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.

9.1 Synthesis Results

All designs were implemented using a Xilinx ML401 prototyping board. The board is built around the Xilinx Virtex4 device, model XC4VLX25-10ff668. 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 5 and Table 6. 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 5 details the results in area and maximum frequency for the various designs.

	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

Table 5: Synthesis results

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 6 stresses this point by comparing the percentages in area savings and frequency reduction of the first column in Table 5 against the second column. The best tradeoff is given by the speed optimization option, confirming that the design was targeted towards a small area footprint.

	% 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

Table 6: Area versus speed trade-offs

The extension slot in the first eMIPS implementation has an available area of about 1,300 slices. Extrapolating on the trend visible in Table 5, we can estimate that an eBug implementation could provide a maximum number of about 27 hardware watchpoints. When hardware watchpoints are not desired eBug uses only 21% of the available extension slot, leaving about 80% of the area for other uses. P2V assertions can fit comfortably in this area.

9.2 Response Time

We measured the time response of the debugger client in a simple test, comparing the software and hardware watchpoint implementations. The goal was to quantify the impact of the added feature from the point of view of the actual 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. Measurements were repeated five times and the average is reported in Table 7. 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 with a baud rate of 115,200 baud. 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 and the same baud rate. 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.

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

Table 7: User-perceived performance gain

The performance difference between the two machine setups appears to be due more to operating system scheduling issues (i.e. in the case of Machine1) 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 that feature extensively.

10 Conclusions and Future Work

We have introduced eBug, a hardware Extension for the eMIPS processor that provides in-process debugging support to a client debugger such as GDB. 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 simply not present on 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 and structures and C++ classes. Multiple conditions could be matched in hardware, by making one match 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 functionalites, 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] Mentor Graphics *ModelSim* at http://www.mentor.com/products/fpga_pld/simulation/index.cfm

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

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

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

[13] PowerPC processor in Xilinx FPGAs. Available at http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/capabilities/powerpc.htm

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

[15] Sparkfun Electronics RS232 Shifter SMD, SKU#PRT-00449. Available at http://www.sparkfun.com/commerce/product_info.php?products_id=449

[16] Stretch, Inc. http://www.stretchinc.com 2006.

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

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

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

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

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

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

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

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

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

Appendix A: Diagrams

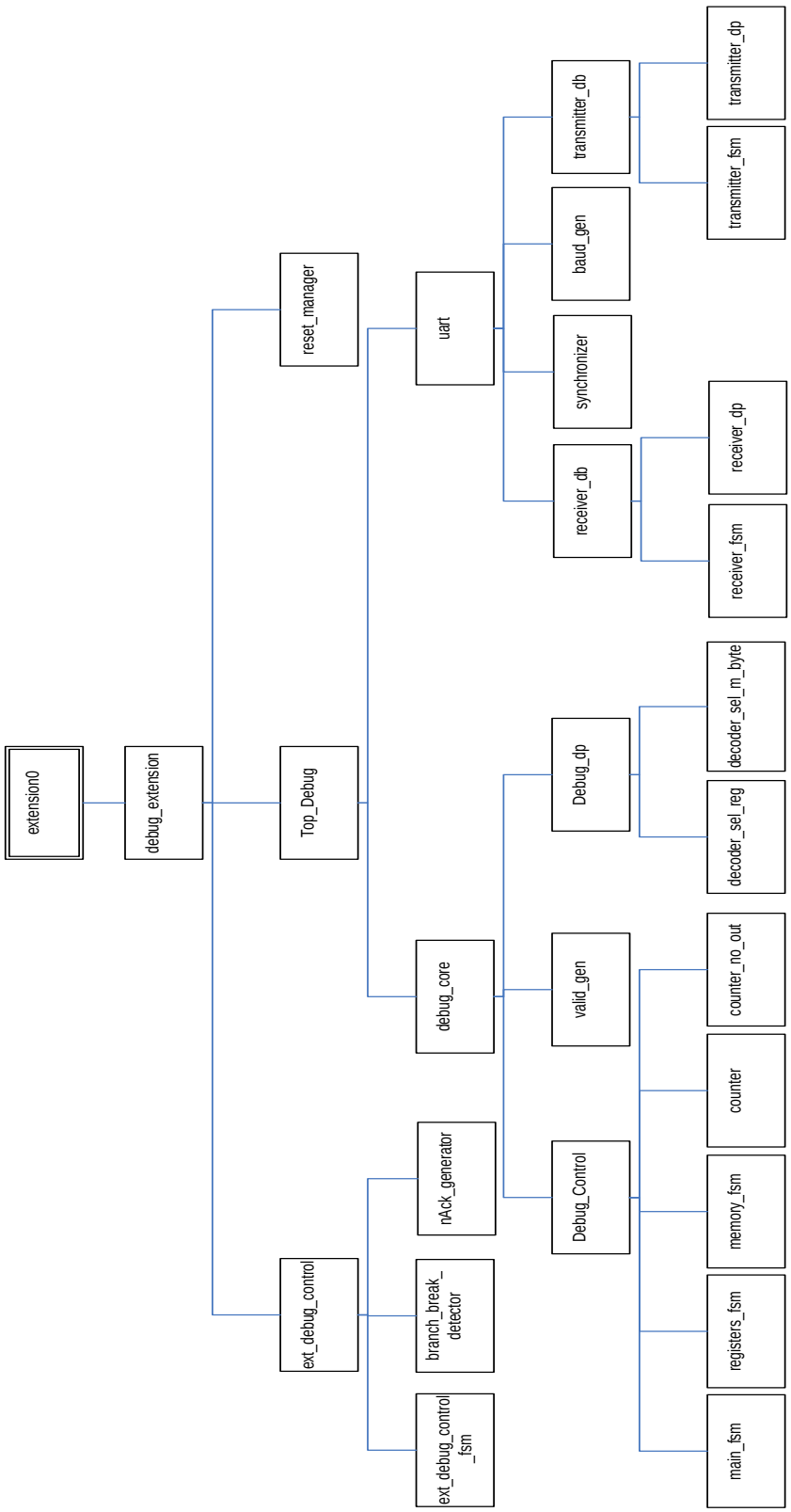


Figure 9 : Module hierarchy

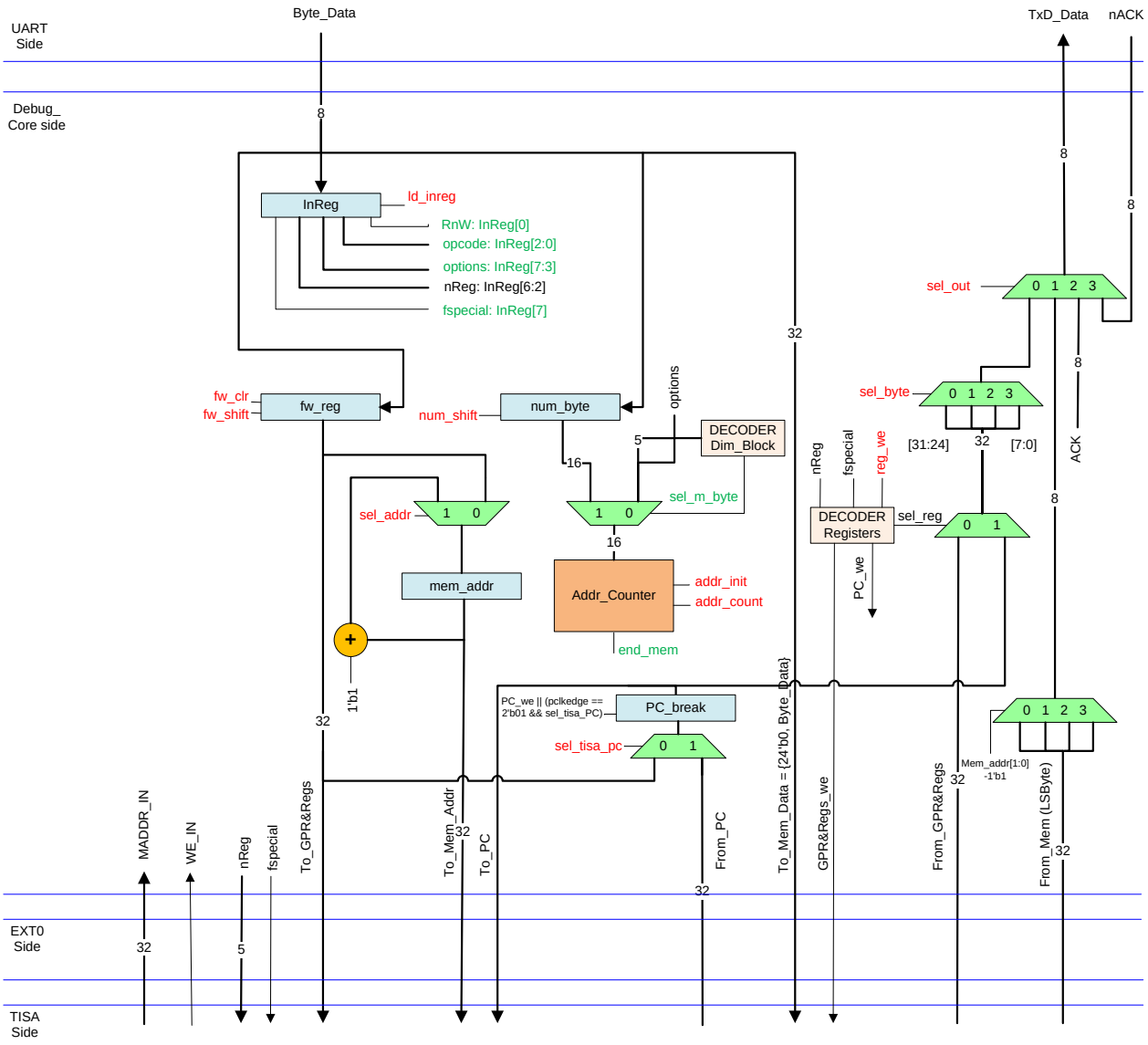


Figure 10: Debug_dp module

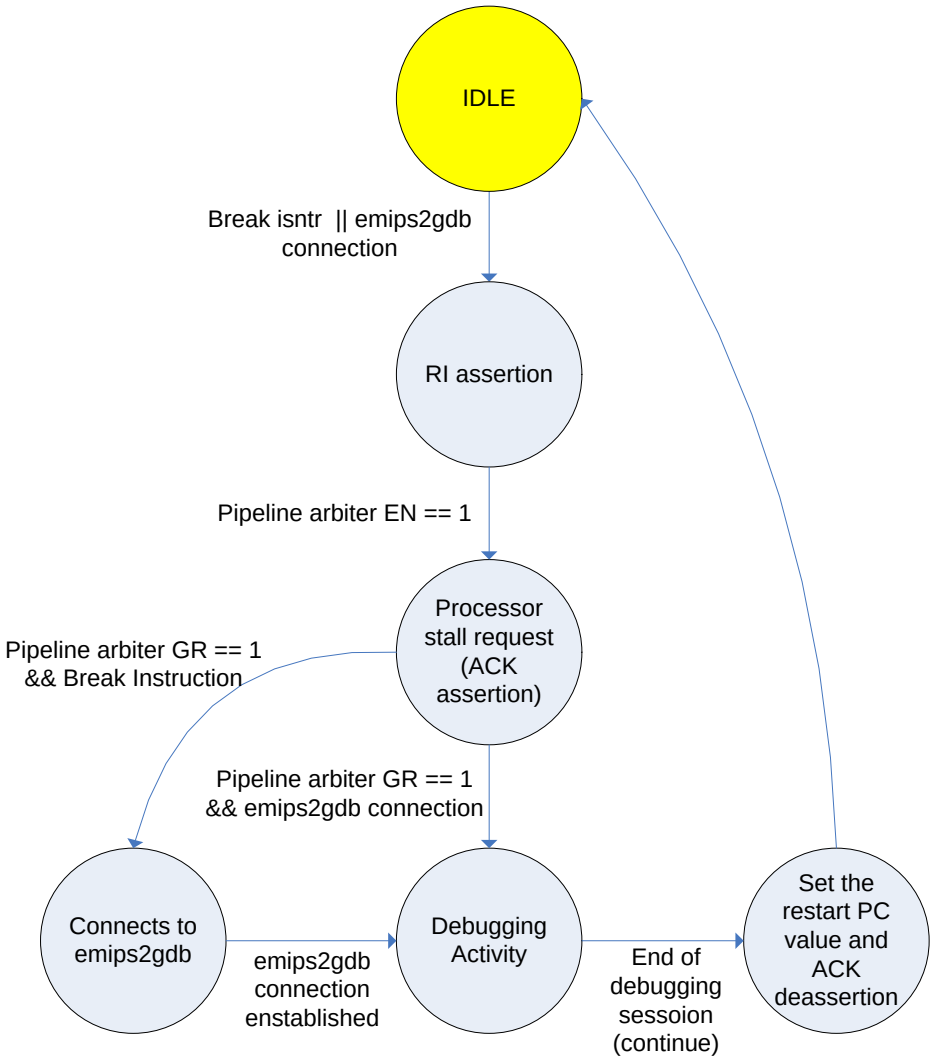


Figure 11: ext_debug_control_fsm

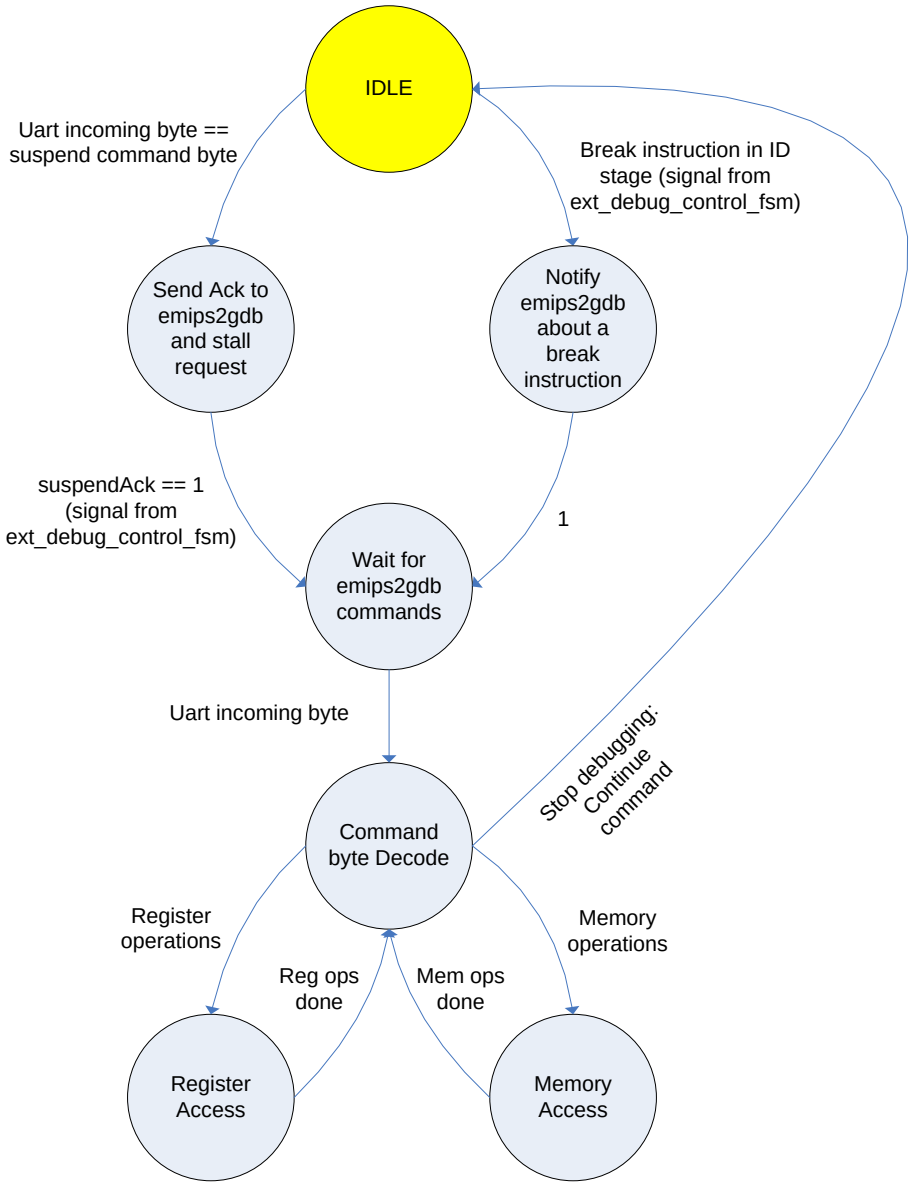
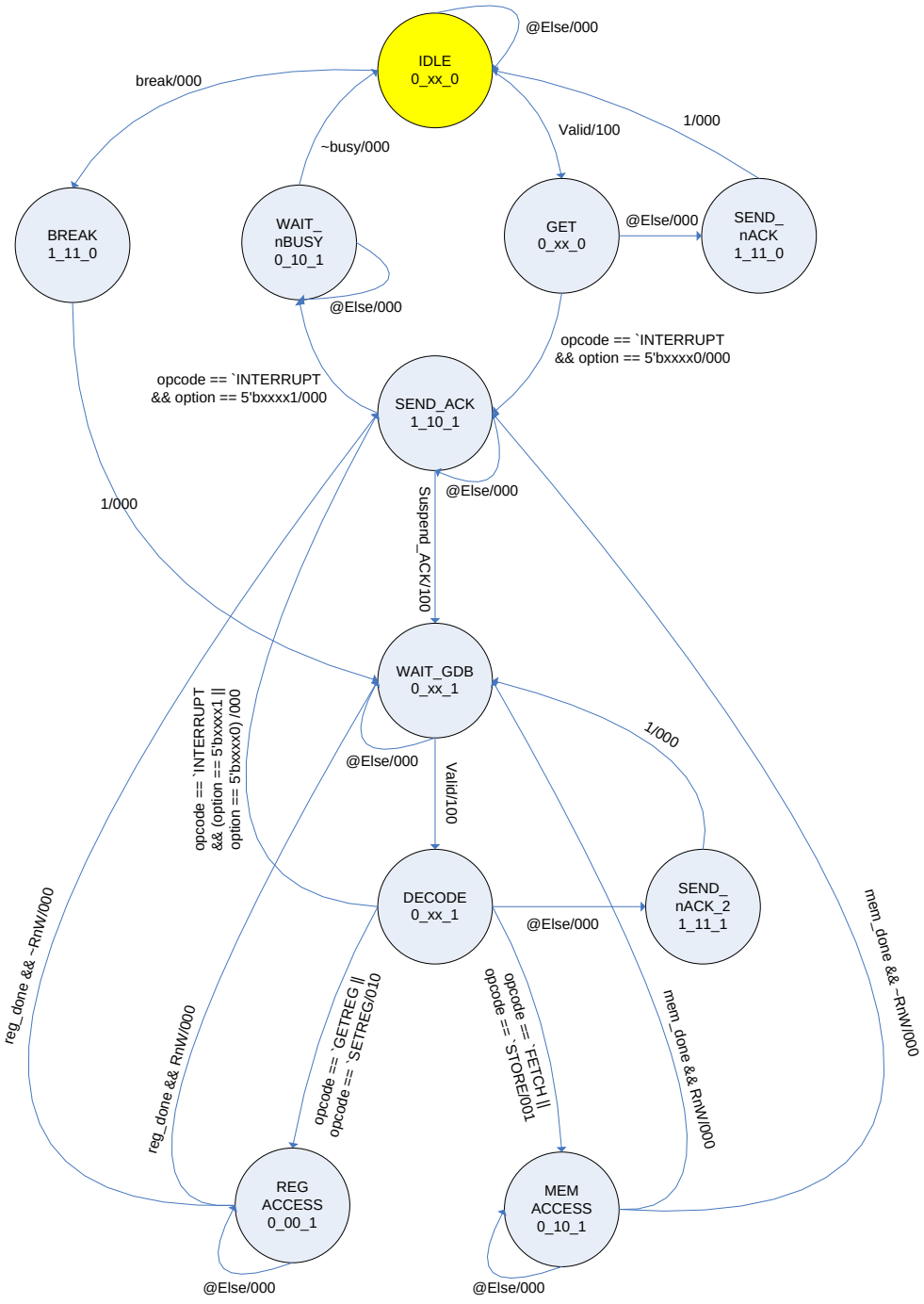
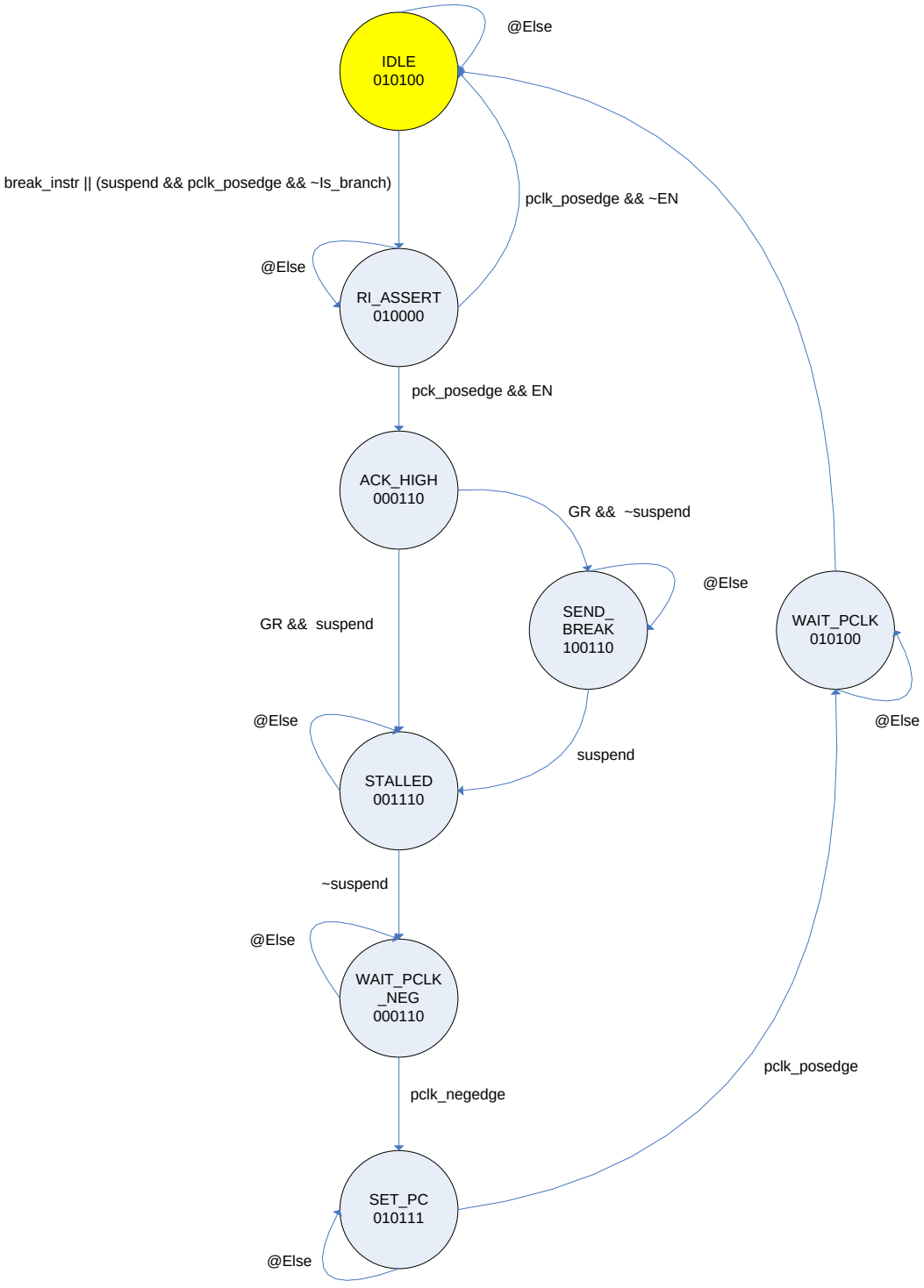


Figure 12: main_fsm



Moore Outputs = {TxD_Start_main, sel_out, suspend}
Mealy Outputs = {ld_inreg, reg_access, mem_access}

Figure 13: main_fsm detailed



Moore Outputs = {break, sel_tisa_PC, suspend_ACK, RI, ACK, PC_NEXT}

Figure 14: ext_debug_control_fsm detailed

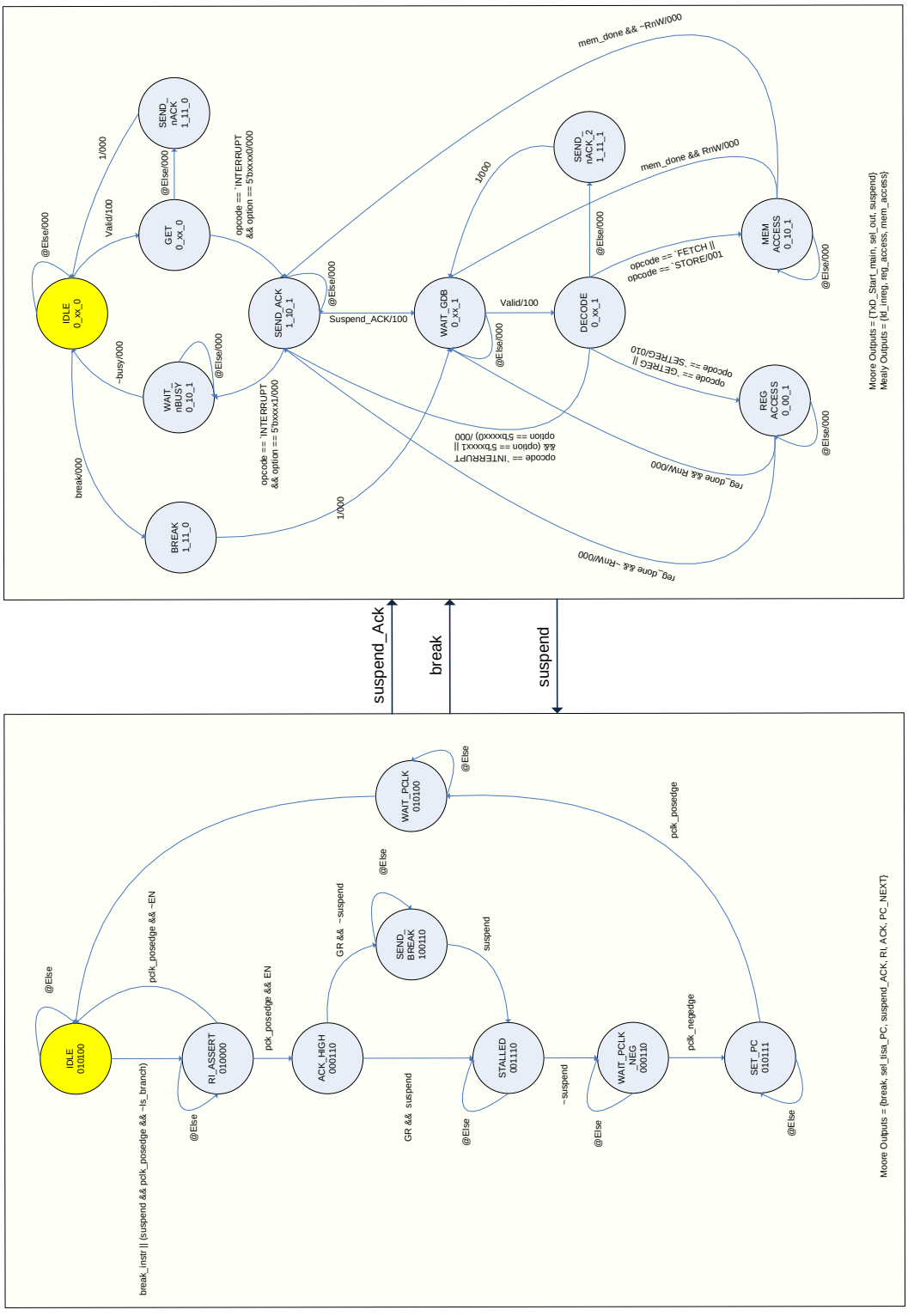
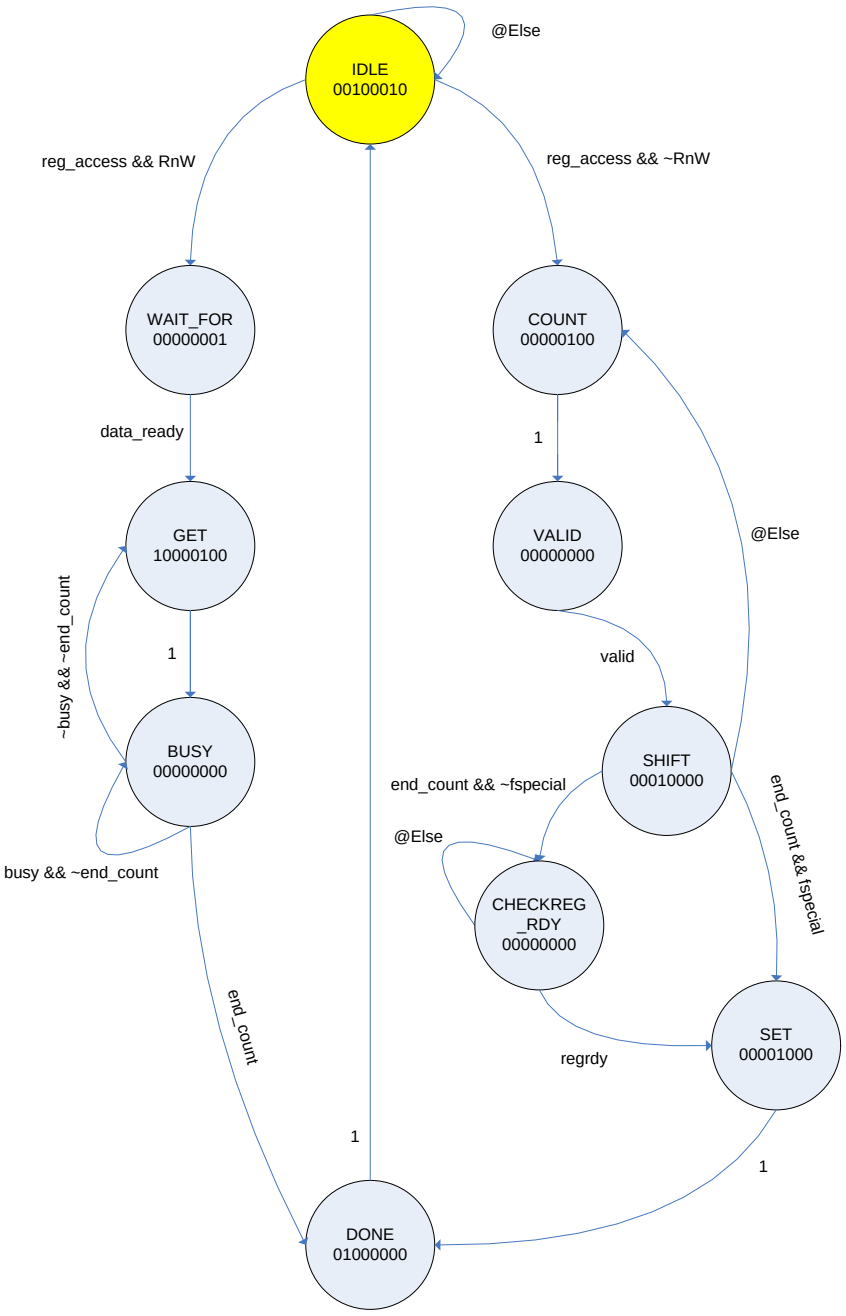


Figure 15: Interaction between main_fsm and ext_debug_control_fsm



Moore Outputs = {TxD_Start_reg, reg_done, fw_clr, fw_shift, reg_we, count_reg, init_latency, count_latency}

Figure 16: registers_fsm

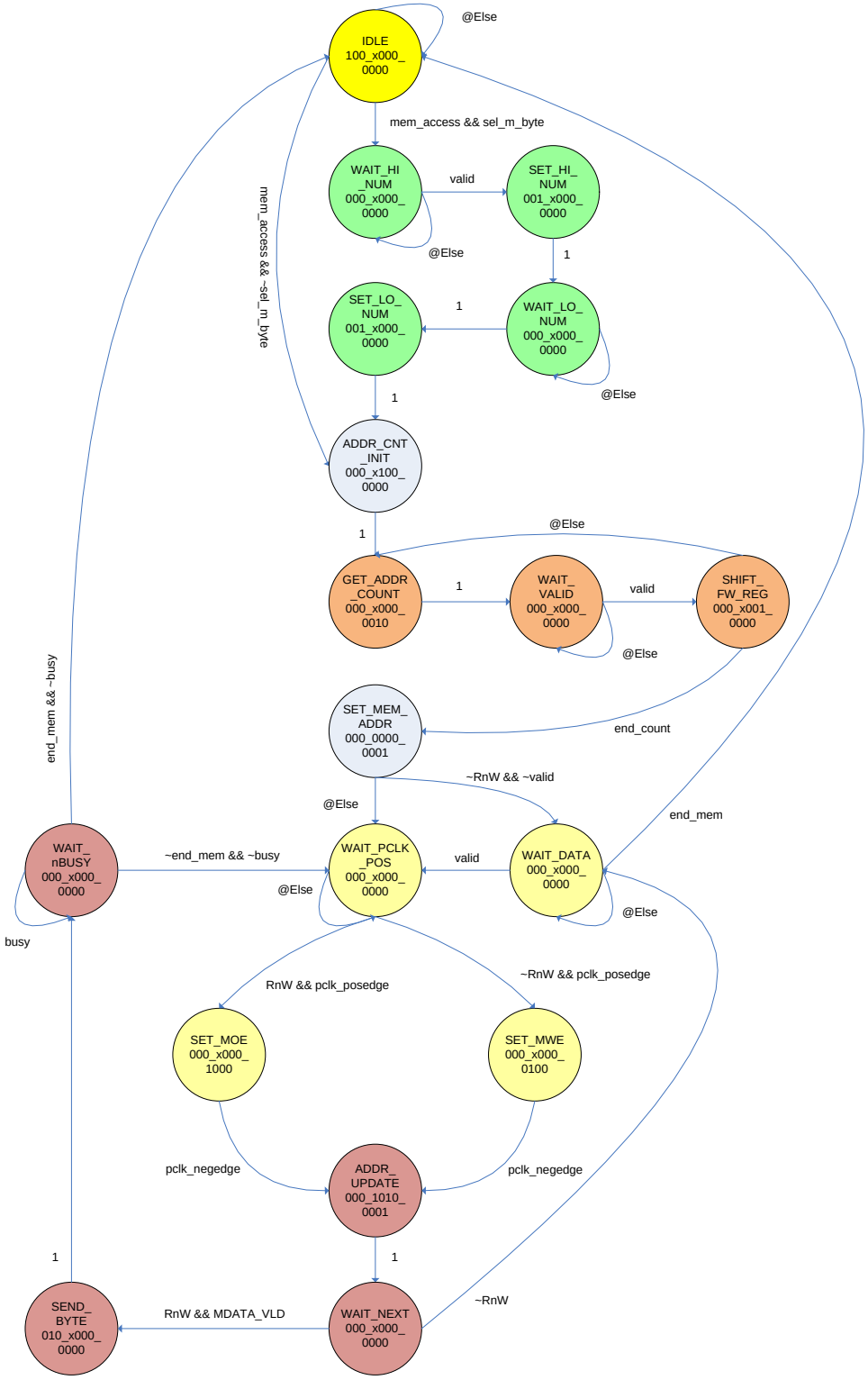


Figure 17: memory_fsm

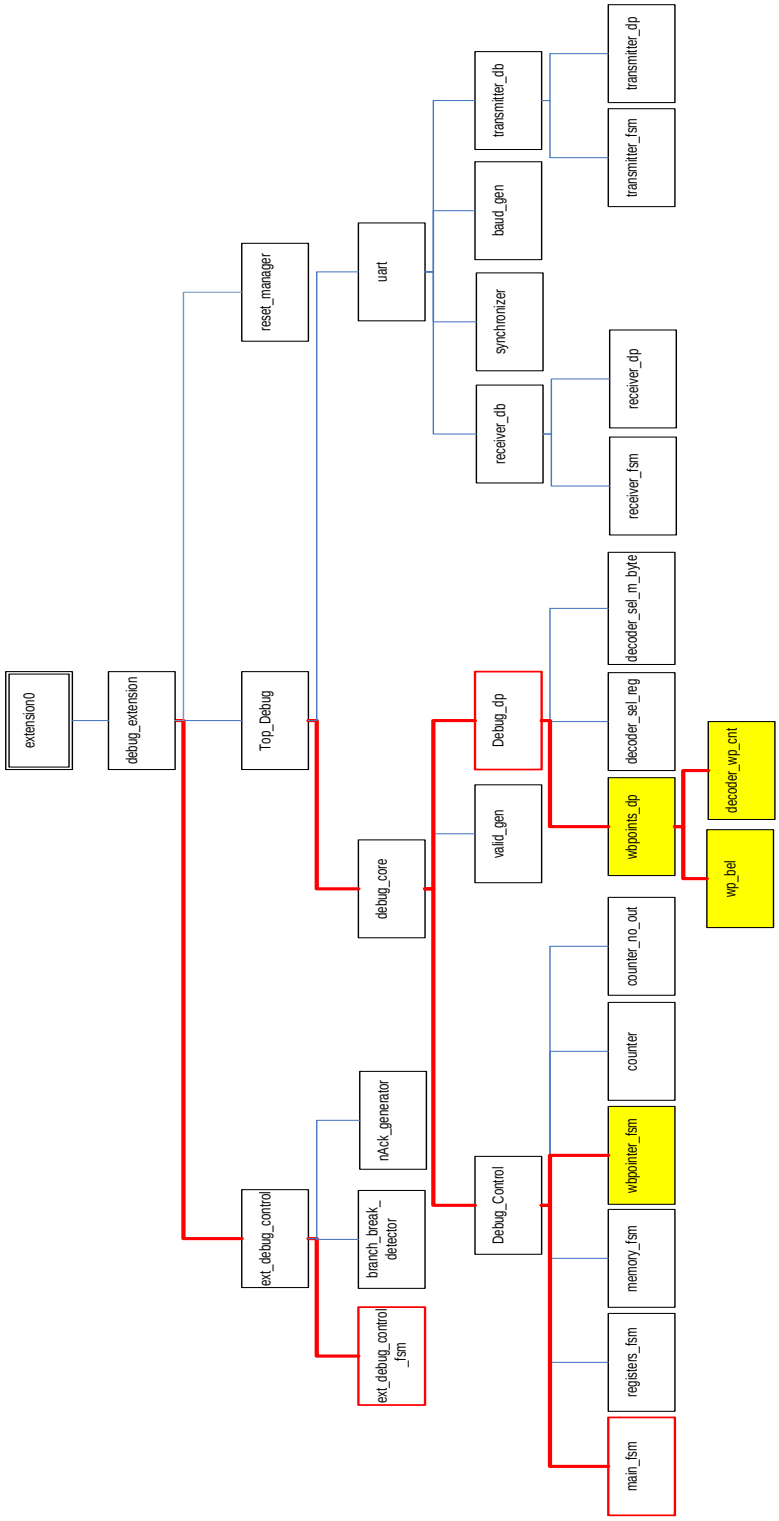


Figure 18: Module hierarchy after addition of watchpoint support

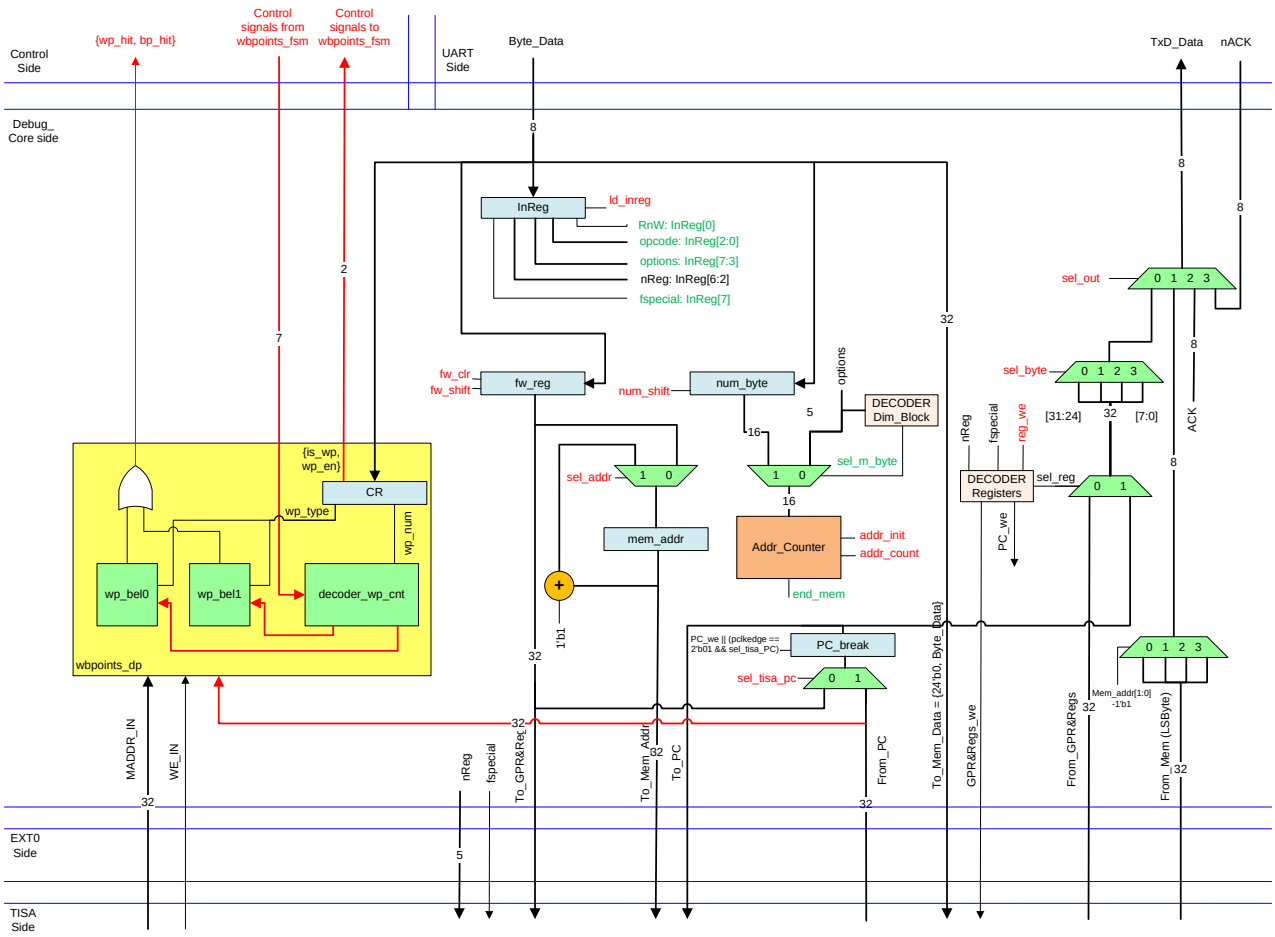


Figure 19: Debug_dp module with watchpoint support

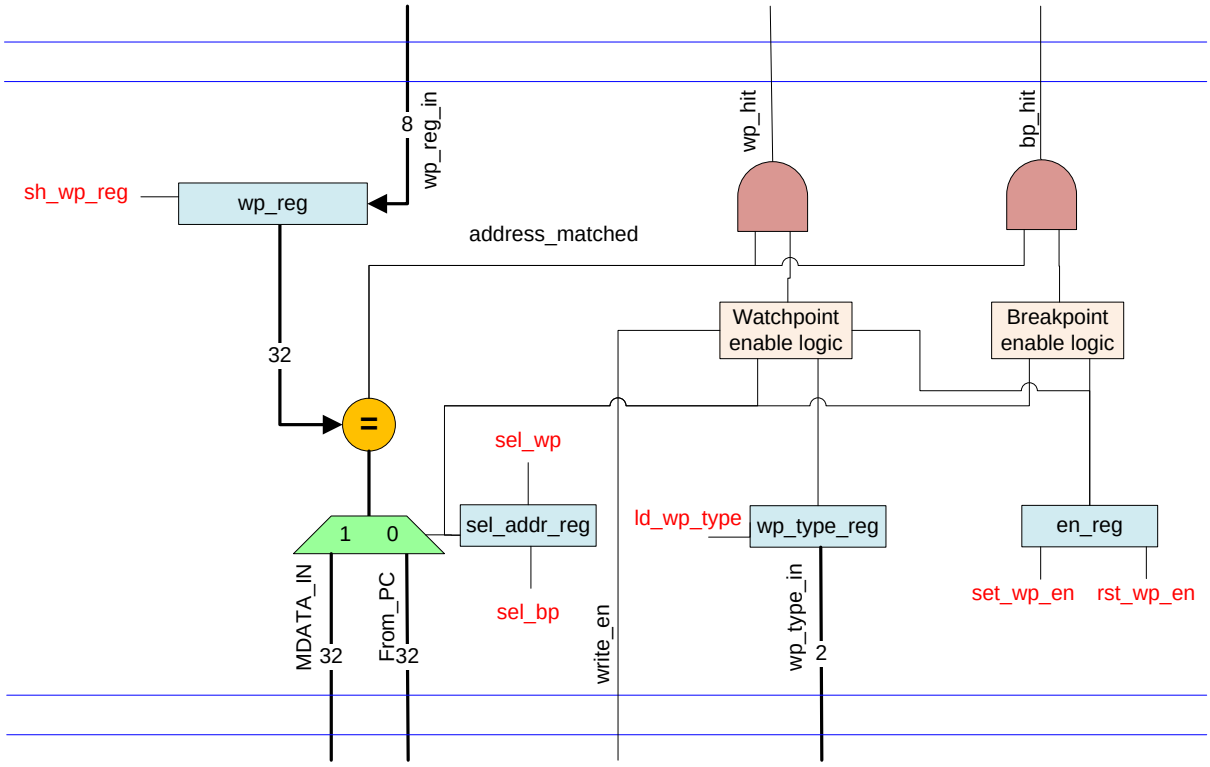
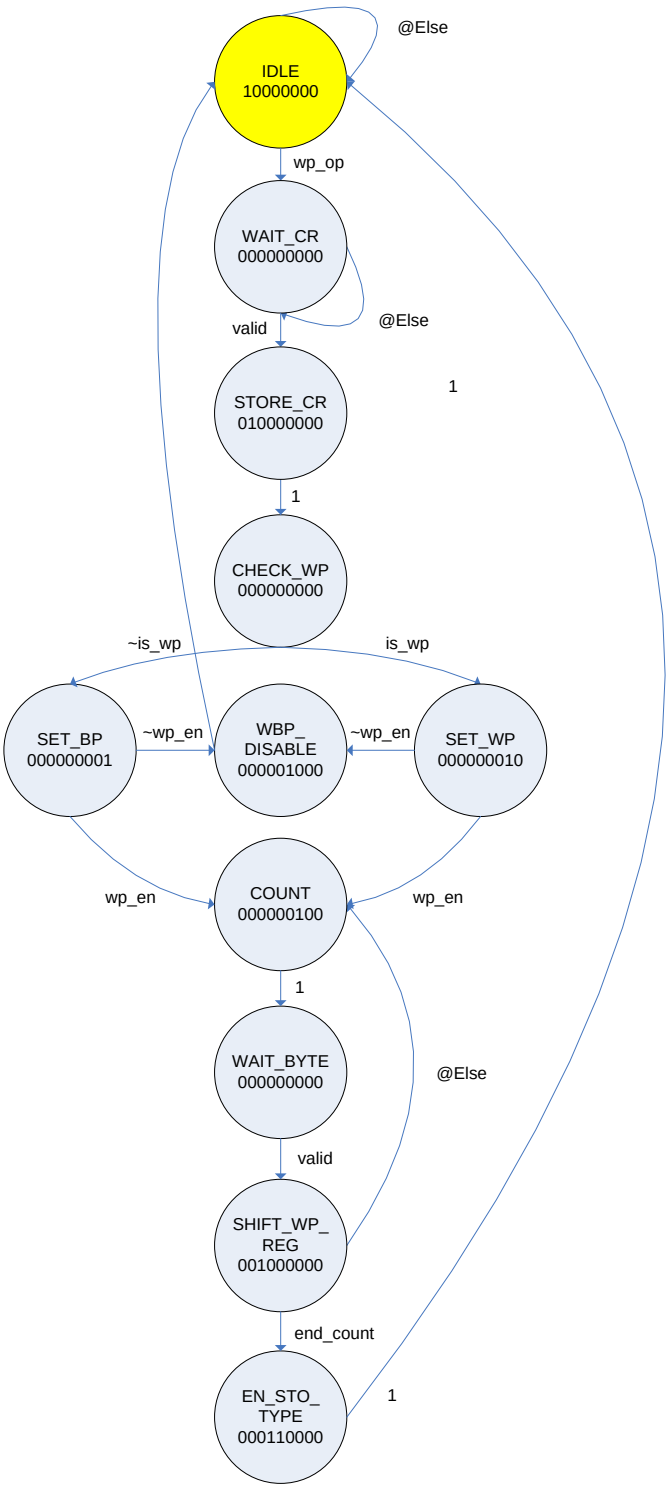


Figure 20: wp_bel module



Moore Outputs = {wp_done, ld_CR, sh_wp_reg, ld_wp_type, set_wp_type, rst_wp_type, count_up, is_wp_on, is_wp_off}

Figure 21: wbpoints_fsm

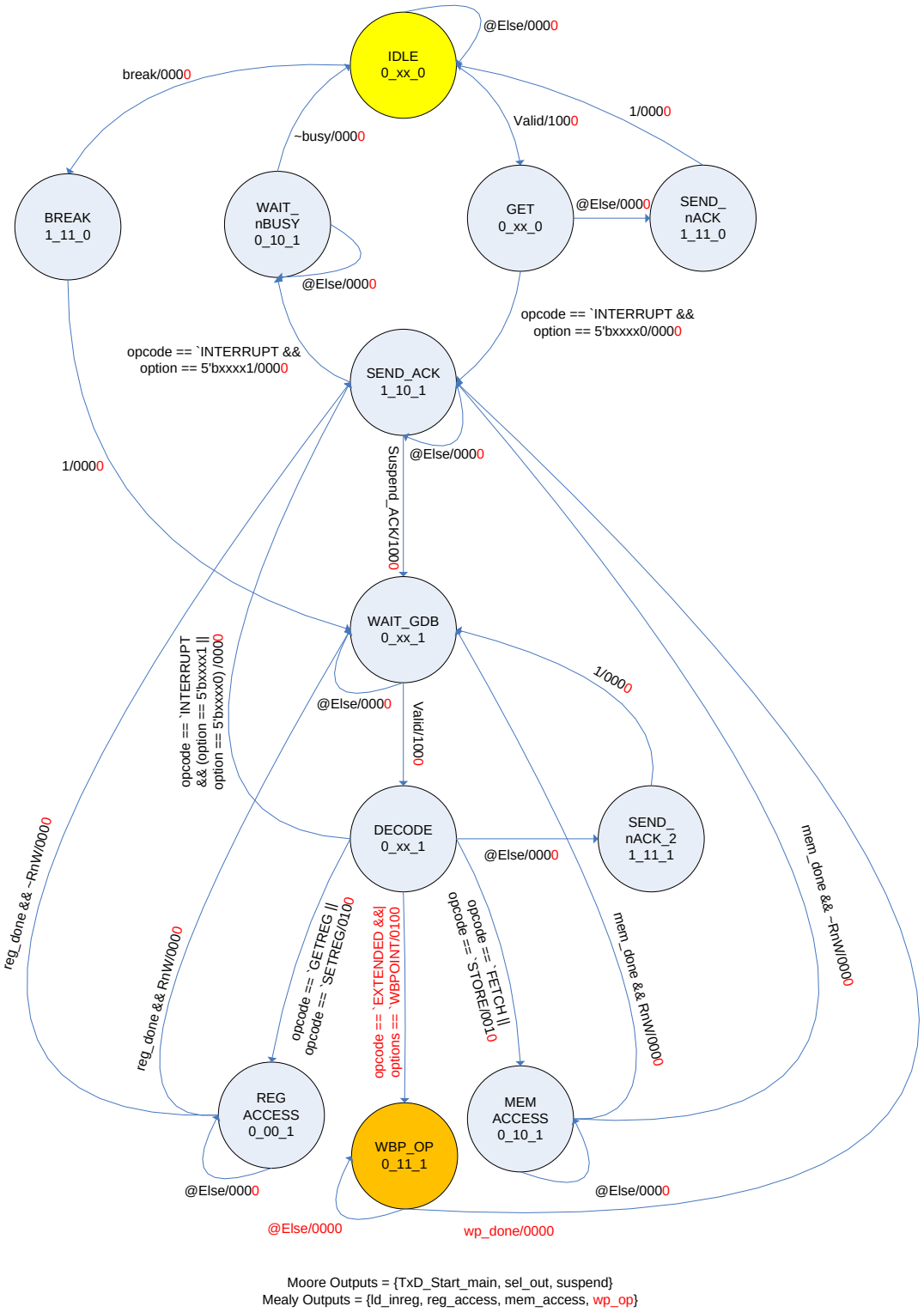


Figure 22: main_fsm modified for watchpoint support

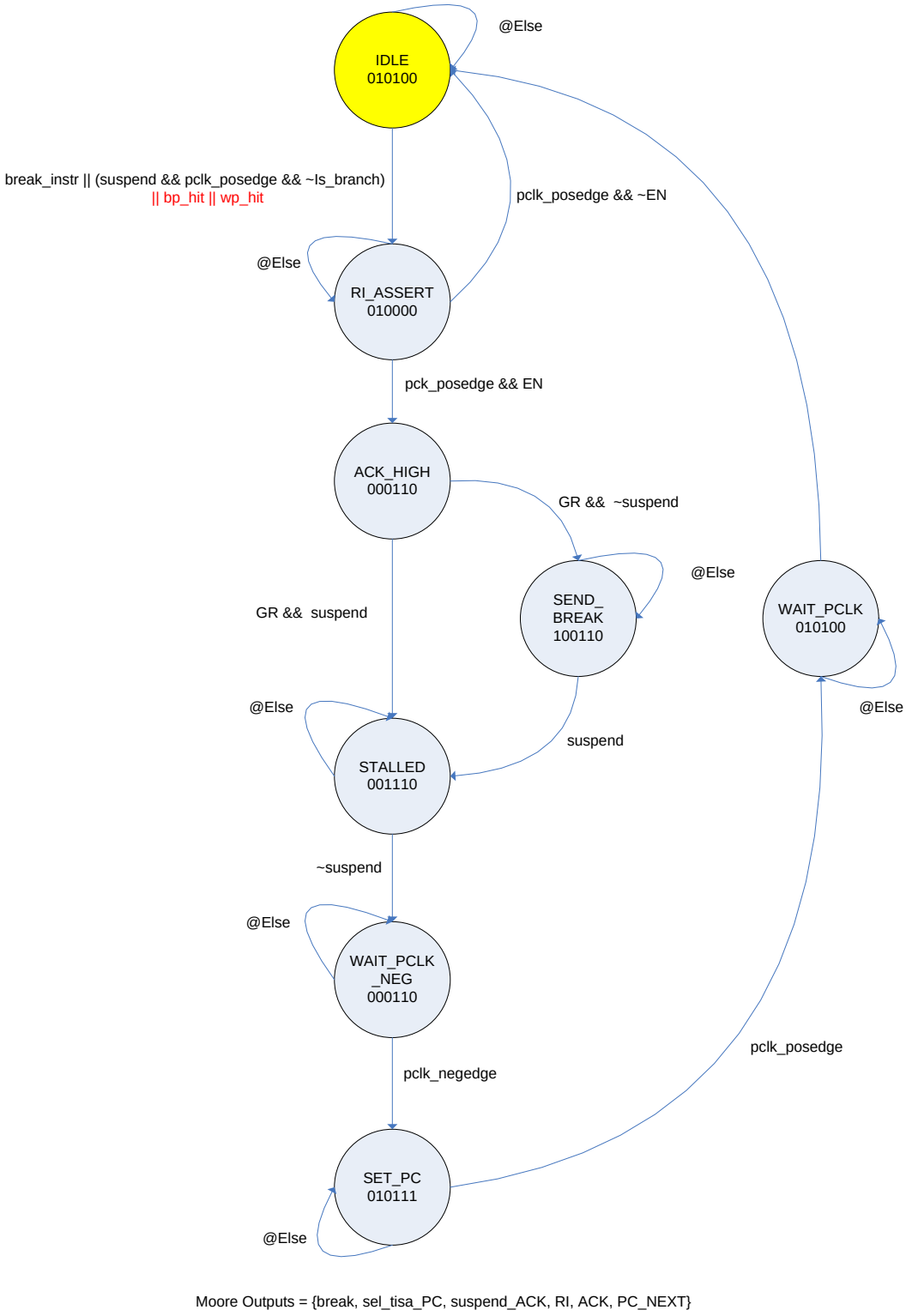


Figure 23: ext_debug_control_fsm modified for watchpoints support