Debugging System for OpenRisc 1000-based Systems

Nathan Yawn
nathan.yawn@opencores.org
05/12/09

History

Rev	Date	Author	Comments
1.0	20/7/2008	Nathan Yawn	Initial version

Contents

1. Introduction

1.1. Overview

This document describes the debugging system for computing systems based on OpenRISC 1000-compatible processors; in particular, the OpenRISC 1200. This system allows a programmer to perform source-level debugging of software running on the target hardware, without use of supervisor code or GDB "stub" code. This system includes both hardware and software components; a block diagram of the system is shown in Figure 1.

The source-level debug functionality depends on GDB, the GNU debugger. For ease of use, a graphical front-end such as DDD can be used. GDB sends its command to a "bridge" program, which translates the commands to a format understood by the debug hardware modules in the system being debugged. The bridge program uses hardware drivers to send the translated commands to the target system via an external JTAG cable.

The JTAG cable sends the signal to a JTAG TAP, a hardware module included in the target system. The TAP enables the hardware debug module, and passes commands and data to it. The hardware debug module acts as the interface to the rest of the system, communicating directly to the CPU and the system WishBone bus. By accessing the CPU, the debug module can start and stop the processor, and read and write CPU internal registers. By accessing the system WishBone bus, the debug module can read and write program variables, upload program code, and set "soft" breakpoints (by temporarily replacing a microinstruction with a CPU trap instruction).

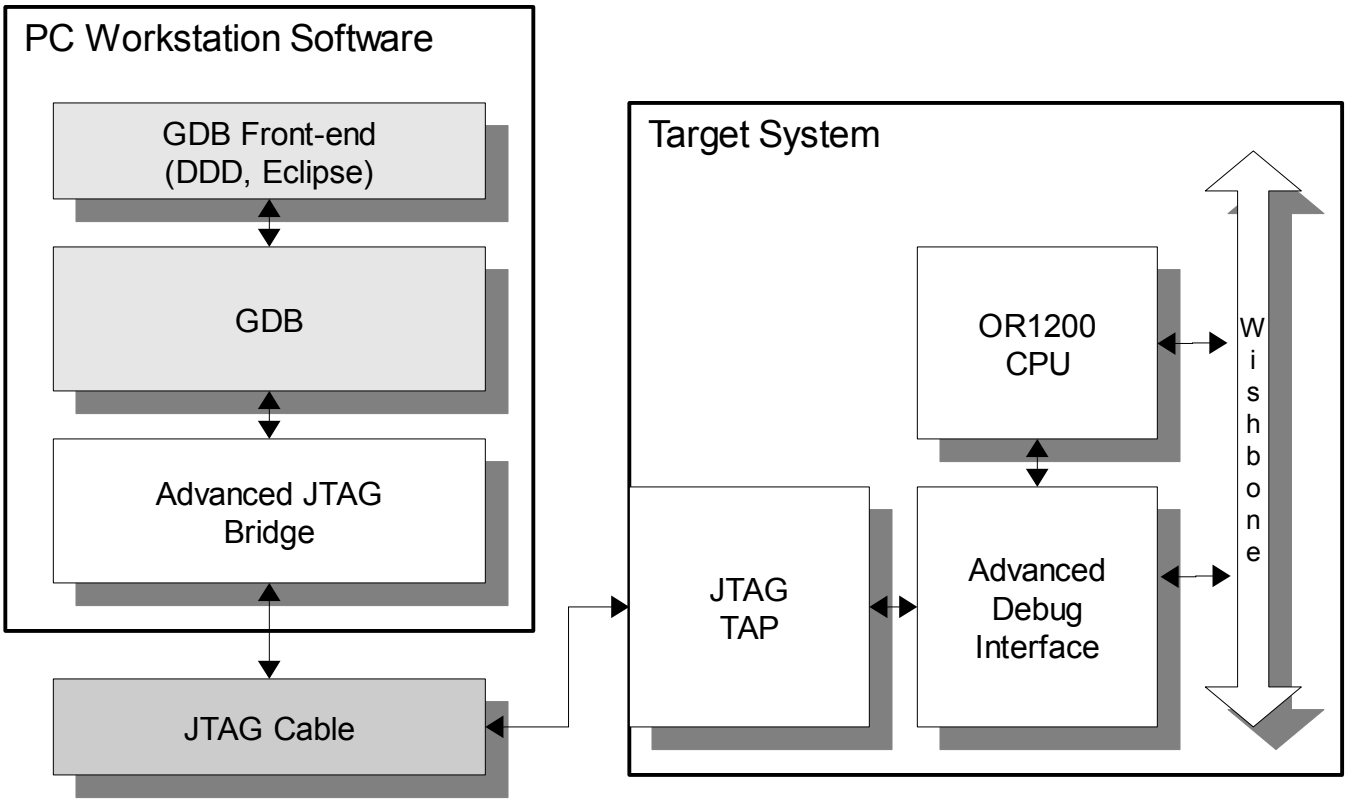

Figure 1: Debugging system block diagram

1.2. Versions

Several versions of the various hardware and software modules are available. The oldest versions are available from the OpenCores website; the hardware debug module is called "dbg_interface," and requires the JTAG TAP module "jtag" from the same source. The compatible bridge program is called "jp2". This combination (jp2 in particular) was written for a specific hardware combination, and may need to be modified for systems with a different configuration.

It is highly recommended to use the latest debug system (the Advanced Debug Interface), which offers the most functionality. A new hardware debug module called "adv_dbg_if" offers support for multi-device JTAG chains, as well as reduced logic requirements. This hardware module is compatible with a newer, modified "jtag" TAP core, or any version of the altera_virtual_jtag or xilinx_internal_jtag cores. This version also uses a new protocol between the bridge program and the debug hardware module, thus a new bridge program is also required. This bridge program is called "adv_jtag_bridge," and includes support for auto-enumeration of JTAG chains and BSDL file parsing. The rest of this document assumes the use of this newest combination of debug components. Note that there is *no* interoperability between old and new versions; adv_jtag_bridge will not work with the older dbg_interface core, the old jtag core is not compatible with the adv_dbg_if, etc.

1.3. Stub-based methods

This document describes hardware-based debugging, sometimes called "backdoor" debugging. An alternate method, using a GDB "stub," may be mentioned in other documents. The "stub" method uses supervisor software running on the target system to perform debugging, instead of the hardware module and bridge program. Communication with the stub is usually done serially or via ethernet. Stub debugging, however, has some drawbacks: the stub must be ported to the target architecture, the system must have a communication device dedicated to the stub, and a driver for that device must be available and stable. Hardware-based debugging places no such requirements on the target system. Stub debugging will not be discussed further in this manual; for further information, see the documentation for GDB.

2. System Components

This chapter will describe briefly all of the hardware and software components involved in the debugging system, as well as the connections between them.

2.1. Hardware

This section describes the hardware components, implemented in HDL, which form the debug system. This section will mention both hardware which performs debugging functions (such as the adv_dbg_if core), and the hardware which will be debugged (the system CPU and bus).

2.1.1. OR1200 CPU

The OpenRISC 1200 is the CPU of the target system. Software running on this CPU will be debugged. While this document assumes an OR1200, the debug system hardware may also work with other processors which use the same external interface to the debug unit. Note that if the CPU being debugged is not OR1000-compatible, a version of GDB specific to the other processor will be required.

The following signals are required from the OR1200 in order to interface to the adv_dbg_if debug hardware module:

CPU Signal name	Direction	Description
dbg_stall_i	DBG->CPU	Logic '1' causes CPU to stall (stop executing instructions)
dbg_bp_o	CPU->DBG	Indicates CPU has reached breakpoint condition
dbg_stb_i	DBG->CPU	Indicates register read or write request from debug module
dbg_ack_o	CPU->DBG	Indicates end of cycle
dbg_adr_i	DBG->CPU	Address of CPU register to be read or written
dbg_we_i	DBG->CPU	Write enable. Write cycle when true, read cycle when false
dbg_dat_i	DBG->CPU	Register write data to the CPU
dbg_dat_o	CPU->DBG	Register read data from the CPU

Note that there have been two different versions of the interface between the OR1200 and the hardware debug module. The current set of signals includes a strobe signal (dbg_stb) and an ack signal (dbg_ack). Older versions of the CPU did not include these signals. These older versions are not compatible with the Advanced Debug Interface module.

The OR1200 includes other signals marked as 'dbg'. These signals include dbg_lss_o, dbg_is_o, dbg_ewt_i, and dbg_wp_o. These signals are not used by the debug hardware module as of this writing, and the outputs may be safely left disconnected at the CPU. The dbg_ewt_i input to the CPU should be connected as a constant logic '0'.

2.1.2. WishBone

The WishBone is the system bus of the target system. For the debug system to work correctly, the WishBone must be connected to each memory which contains code or data used by the program to

be debugged, and each memory must appear to the debug unit at the same address it appears to the OR1200. The hardware debug module (adv_dbg_if) must also be connected to the WishBone bus, along with any peripherals you wish to access via the debug system. Note that the hardware debug module is a WishBone 'master', and must be connected to a WishBone master interface.

The adv_dbg_if core does not perform WishBone burst cycles. Therefore, it should be compatible with any version of the WishBone standard, (up to B.3 as of this writing). The following WishBone signals are used to interface to the hardware debug module:

WB Signal name	Direction	Description
wb_cyc_i	DBG->WB	Bus request / ongoing cycle indication
wb_stb_i	DBG->WB	Request start of cycle
wb_ack_o	WB->DBG	End of cycle acknowledge
wb_sel_i	DBG->WB	Byte lane enables
wb_we_i	DBG->WB	Write enable
wb_err_o	WB->DBG	Bus error indication
wb_adr_i	DBG->WB	Bus address
wb_dat_o	WB->DBG	Read data
wb_dat_i	DBG->WB	Write data

The adv_dbg_if WishBone interface also includes the signals wb_cab_o, wb_cti_o, and wb_bte_o. These are fixed to constant values, and are only used when the debug module is connected to a WishBone master interface which includes these signals. If the master interface does not include these signals, then they may safely be left unconnected.

2.1.3. Advanced Debug Interface

The hardware debug module (“adv_dbg_if” core) controls transactions to the CPU and the WishBone bus, and provides clock domain synchronization between the CPU, the WishBone, and the JTAG TAP. The debug module decodes the protocol sent via JTAG by the 'bridge' software program. This protocol includes CPU stall and reset commands, CPU register reads and writes, and WishBone data reads and writes. It is the primary hardware component of the debug system.

Note that the adv_dbg_if includes two CPU interfaces, one of which is disabled at synthesis time by default. While the debug hardware module and the bridge software program both support multiple CPUs, this support is currently absent from GDB. Therefore, the second CPU module is disabled in hardware.

The debug hardware module receives commands and data via a JTAG Test Access Port (TAP); the debug module appears as one of several scan chains in the TAP, and data is moved in and out via a clocked serial interface. This interface is described here:

DBG Signal name	Direction	Description
tck_i	TAP->DBG	Clock signal
tdi_i	TAP->DBG	Serial data to the debug module
tdo_o	DBG->TAP	Serial data out to the TAP
rst_i	TAP->DBG	Reset
capture_dr_i	TAP->DBG	TAP CAPTURE_DR state, load output data to shift register
shift_dr_i	TAP->DBG	TAP SHIFT_DR state, do serial in/out shift
pause_dr_i	TAP->DBG	TAP PAUSE_DR state, do nothing
update_dr_i	TAP->DBG	TAP UPDATE_DR state, capture input from shift register
debug_select_i	TAP->DBG	Enable debug module

Note that as of this writing, the pause_dr_i input is unused, and may therefore be safely connected to a constant logic '0'.

2.1.4. JTAG TAP

The JTAG TAP is the debug hardware's access off-chip. It consists of an Instruction Register (IR) and one or more scan chains, which appear as Data Registers (DR). The TAP multiplexes the serial input and output data to one DR at a time, based on the value in the IR. The hardware debug module appears as a DR.

Several different JTAG TAP implementations are available; the choice of TAP depends on the technology used to implement the system. As of this writing, three implementations are available. These are described below.

Standard JTAG

The “jtag” core represents a standard JTAG TAP, as described by standard IEEE 1149.1. It is accessed by four (or five) external pins, and includes an IR and several DRs. This TAP is suitable for use in ASICs and all FPGAs. The TAP's off-chip interface is described below.

TAP Signal name	Direction	Description
TCK	EXT->TAP	Serial clock
TMS	EXT->TAP	Mode Select signal
TDI	TAP->EXT	Serial data to TAP
TDO	EXT->TAP	Serial data from TAP
TRSTN	EXT->TAP	Optional reset line

Note that the TRSTN line is active-low. The TRSTN signal is not present on all JTAG cables. The TRSTN pin should be configured with a pull-up, or tied to a logic '1' when used with a cable without this signal.

Altera Virtual JTAG

This TAP is implemented in the “altera_virtual_jtag” core. It may be used only when the

system is implemented in an Altera FPGA which supports the “sld_virtual_jtag” megafunction. The Altera Virtual JTAG TAP allows a user to connect the debug hardware through the FPGA's TAP (the same TAP used to download a bitstream to the FPGA). This means that separate, dedicated pins for a debug system TAP are not required, and the FPGA can be configured and then the software debugged without changing the JTAG cable connection or using a second cable.

Xilinx Internal JTAG

This TAP is implemented in the “xilinx_internal_jtag” core. It may be used only when the system is implemented in a Xilinx FPGA which supports a BSCAN_* macro block (e.g. BSCAN_SPARTAN3, BSCAN_VIRTEX4, etc.). The Xilinx Internal JTAG TAP allows a user to connect the debug module through the FPGA's TAP (the same TAP used to download a bitstream to the FPGA). This means that a separate TAP for the debug system is not required, and the FPGA can be configured and the software debugged without changing the JTAG cable connection or using a second cable. This TAP module has no external pin connections.

2.1.5. JTAG Cable

This is a piece of hardware used to connect the user's PC / workstation to the system being debugged. The target-side interface consists of the JTAG signals described above (external interface to the “jtag” core). The PC interface may vary, but is usually parallel port or USB. While many different cables are made by many different manufacturers, there is no standard driver interface for them. As such, the JTAG cable must be explicitly supported by the 'bridge' software program, which must contain a driver for the cable. See the section on the Advanced JTAG Bridge for a list of cables supported by that program.

2.2. Software

This section discusses the software components of the debug system. These components run under the operating system of the user's workstation. Both components written specifically for the debug chain (adv_jtag_bridge) and components which may be used to debug any system (e.g. the DDD front-end for GDB) are mentioned here.

2.2.1. Advanced JTAG Bridge

This software program runs in the background of the user's workstation. It receives commands and data from GDB over a network socket, translates them into a JTAG bitstream formatted for the adv_dbg_if hardware core, and sends the bitstream to the target system using a JTAG cable driver. Data is read from the hardware debug module and returned to GDB by the reverse process.

The following JTAG cables are supported by the advanced JTAG bridge:

- Xilinx Parallel Cable III (and IV in compatibility mode)
- Xilinx Platform Cable USB (DLC9)*
- Altera USB-Blaster
- XESS, Inc. direct connections

* The Xilinx DLC9 driver only supports low speed “bit-bang” mode as of this writing.

At startup, the bridge program attempts to enumerate the JTAG chain, identifying all devices in

the chain. If available, BSDL files will be used to determine the IR length of each device, and the command for the target device's TAP which will make the hardware debug module active. If BSDL files are not available, the user must enter the IR lengths on the command line. Once the chain has been enumerated, the bridge program can run an optional test on the memory and CPU on the target system. The bridge then opens a socket and waits for GDB to connect to it.

2.2.2. GDB

GDB is the GNU debugger, a program commonly used to debug software on many system types and architectures. The OR1000 debug system requires a version of GDB which has been modified for use with the OR1000 processor. In addition, the Advanced Debug system requires a version of GDB which uses the RSP protocol to communicate with remote systems. Currently, this is version 6.8.

GDB has information on all of the registers within the target system CPU; this allows it to read or write CPU registers by address. GDB also reads debug information compiled into the programs run on the target system; this allows it to find the memory addresses at which program variables are stored on the target system, and find the line of code in a source file associated with the current value of the CPU's program counter. GDB can also request that the processor be stalled or restarted. All activity in the debug system is directed by GDB.

A more thorough discussion of GDB is beyond the scope of this document. For more information, see the GDB documentation.

2.2.3. Optional GDB Front-end

GDB is a command-line program, and the user interface is text-based. It may therefore be desirable to use a graphical front-end to GDB, to aid in visualization and usability. “DDD” is one such program; no modifications need to made to DDD in order to use it with the OpenRISC debug system. DDD allows the user to start and stop the program, set breakpoints, and examine variables and data graphically. DDD must be told to use the version of GDB modified for the OR1000; add the command-line switch “--debugger or32-uclinux-gdb” when starting DDD, assuming “or32-uclinux-gdb” is the OR1000 version of GDB, and can be found in the current $PATH. You will also need to manually enter commands at the DDD/GDB console to connect to the adv_dbg_if program (“target remote :9999”), and to load your program onto the target (“load”). Consult the DDD manual for more information on debugging with DDD.

Other graphical front-ends, such as the one integrated with the Eclipse development environment, may also work, but are currently untested. In general, front-ends that have the CPU register list hard-coded will need modification, while those without may work as-is.

3. Operation

This section describes the setup and operation of the OpenRISC debugging system.

3.1. Hardware

In order to use the debugging system, the hardware modules must be included in the target system. To do this, the adv_dbg_if core and one of the JTAG TAP cores must be instantiated in the HDL for the top-level system module, and connected to each other according to the connection lists above. The debug hardware core should have the WishBone module and one CPU module enabled, and these interfaces must be connected to the system bus and OR1200, respectively. The debugging option in the OR1200 must also be enabled at synthesis time. If the standard JTAG TAP is used, the five JTAG signals must be routed to external pins; if not, no external connections must be made. The system can then be synthesized, and implemented by downloading to an FPGA (or fabricated as an ASIC, for the adventurous).

3.2. Setup

The target system must be connected to the user's workstation with a JTAG cable. For systems which use the altera_virtual_jtag or xilinx_internal_jtag cores, this connection has already been made in order to download the bitstream to the FPGA. If the standard TAP has been used, the target-side connection of the cable must be attached to the appropriate pins of the target chip, and the PC side must be attached to the user's workstation.

After the target system (and, if necessary, the JTAG cable) has been powered on, the adv_jtag_bridge program should be started on the user's workstation. Note that the program may require root privileges in order to access the parallel port. The USB cable drivers use libusb, which requires no special privileges. The bridge program will probe for the cable specified on the command line, then enumerate all devices on the JTAG chain.

In order to operate correctly, adv_jtag_bridge needs to know the length of the IR in every device on the JTAG chain, and the command for the target device's IR which will make the debug hardware module active. If a BSDL file is available for a device on the chain, the information will be taken from there. If a BSDL file is not available for a device, its IR length must be specified on the command line using the '-l' option. If a BSDL file is not available for the target device, the debug command (IR value to select the debug module) must also be specified on the command line with the '-c' command. A brief summary of the command line options for the adv_jtag_bridge is available by running the program with the '-h' option. More detailed information is available in the program's documentation.

Assuming the bridge program can find all information it needs, it will open a network socket on port 9999 (assuming this has not been overridden on the command line) and wait for a connection from GDB (optionally after a test of the target's memory and CPU, if the '-t' option is specified).

Once adv_jtag_bridge is running and ready, GDB must be started. If no front-end is used, GDB may be started directly on the command line by typing “or32-uclinux-gdb <program name>”. Once GDB is started and the GDB console is ready, it must be pointed at the bridge program. Use the command “target remote :9999” to make GDB connect to the bridge program. Note that it is also not required for GDB to run on the same machine as the bridge program; if a different machine is used, add the name or IP address of the machine on which adv_jtag_bridge is running before “:9999”, with no space between.

After the 'target' command, type “load” at the GDB console. This make GDB send the program to the target system's memory via the debug system. After the upload is complete, the CPU's program counter must be set to the start address of the program. Use the GDB command “set $pc=<hex address>” to do this. Because the memory system may be differently configured, this address may change depending on your system and linker script. At this point, GDB is ready to begin running or single-stepping the program, set breakpoints, or any other debugging operation. See the GDB manual for details on debugging with GDB.

If DDD is used, “--debugger or32-uclinux-gdb” should be added on its command line. GDB will be started automatically. However, the 'target,' 'load,' and 'set $pc' commands must still be entered at the DDD/GDB console before any of the debugging operations may be used.

3.3. Control and Data Flow

To illustrate information flow in the debugging system, we will use the example of the user interrupting the program running on the target system, then reading the current value of the program counter. We assume that the system has already been set up, and the code has been downloaded and started on the target CPU.

The process begins with the user pressing the 'break' button in DDD, or typing 'control-C' in the GDB console. GDB sends a message via network socket to adv_jtag_bridge, indicating that it should stop running the program.

Upon receipt of the message, adv_jtag_bridge formats a JTAG bitstream for the TAP, which will make the debug hardware module the active DR in the TAP (if it has not done so already). This bitstream is sent over the JTAG cable via a cable driver in adv_jtag_bridge. Once the hardware debug module is active, a second JTAG bitstream is sent which makes the CPU sub-module active inside the debug module (this is also only done if it has not been done already). Finally, the bridge program forms and sends a message which will set the stall bit inside the CPU sub-module. This will cause the dbg_stall_o line of the debug module to go high, which will cause the CPU to stop executing instructions and freeze its pipeline. Once this is done, adv_jtag_bridge will send a response to GDB via network socket, indicating the CPU is stopped.

Once this message is received, GDB will attempt to read the program counter (PC) to determine where in the program the CPU is executing. GDB will send another request via network socket indicating a read from the CPU, at the register address of the PC.

The bridge program will again receive this request, and insure that the debug module is active in the TAP and that the CPU sub-module is active in the debug module. The bridge will then form a JTAG bitstream requesting a read from the CPU's internal register bus, starting at the PC's address and with length 1. The read must be performed in two separate JTAG transactions; the first transaction sends the address to the CPU sub-module and begins the read operation on the CPU register bus. The second transaction gets the read data from the CPU sub-module and brings it back into the bridge program. The bridge program then returns the read data (the value of the OR1000's program counter) to GDB via the network socket.

4. Simulation

Sometimes it may be desirable to use a software debugger on a simulation of the target system. This may be because the target hardware is not yet available, or to verify functionality before committing the design to silicon.

4.1. Or1ksim

The program "or1ksim" is the OpenRISC 1000 architectural simulator. It is a stand-alone C program which emulates the instruction set and behavior of an OR1000 CPU. The simulator has also been expanded to included simulations of many OpenCores peripherals, including serial ports, memory controllers, etc. The simulator may be used to develop software for the target platform before the hardware becomes available or fully verified. This will allow the user to separate debugging of the hardware and software, rather than having to run untested software on uncertain hardware.

The or1ksim simulator includes a built-in GDB server. This means that GDB can connect directly to the simulator, and the bridge program and JTAG cable are not required. A block diagram of this system is shown in Figure 2.

Note that if you want to simulate any peripherals, the simulator must be configured to match your target hardware system. This configuration is beyond the scope of this document, see the or1ksim documentation for details.

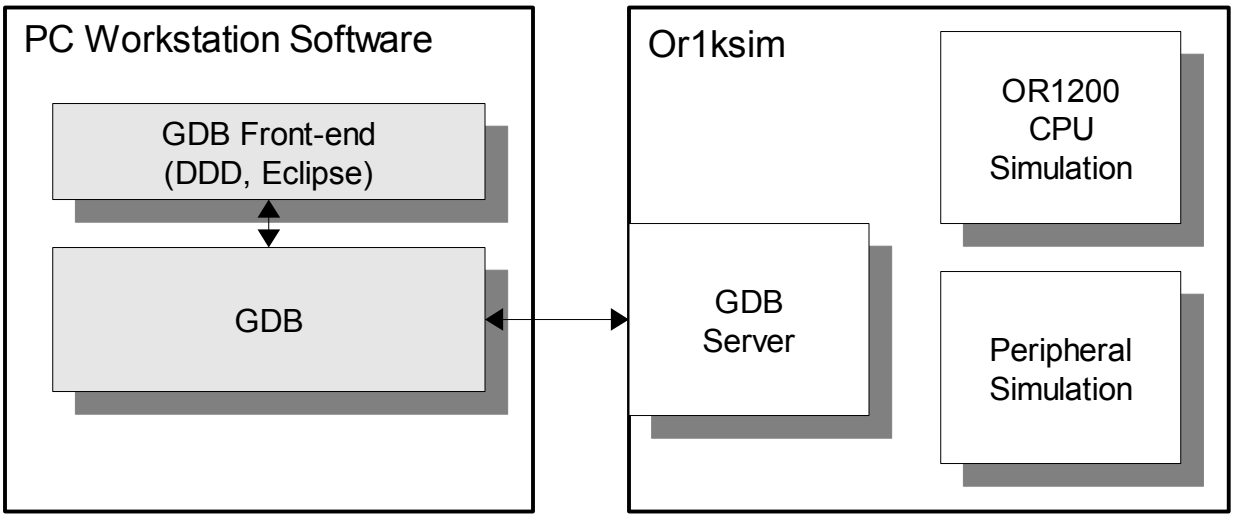

Figure 2: Block diagram of debugging system using or1ksim

4.2. RTL simulation

It is possible to connect GDB to an HDL simulation of an OR1200 / WishBone system and debug it as if it were real hardware. The HDL simulation may be run using ModelSim, or another simulator with the necessary features (described below). Most of the same debug system components are used; code which connects the bridge program to the simulator takes the place of the hardware

JTAG cable. There are two different methods available to connect the bridge program to an HDL simulation, the method you choose will depend upon the capabilities of your simulation program.

4.2.1. File IO

This method uses the workstation's file system to pass data between the adv_jtag_bridge program and an HDL simulation. As such, the HDL simulator must support verilog file IO in order for this method to work.

A verilog module is added to the hardware system which uses data read from a file to set the states of the JTAG lines. The state of the serial data output is written to another file. These files are written and read, respectively, by the bridge program. A block diagram of the file IO simulation debug system is shown in Figure 3.

The verilog module which performs the file IO and controls the JTAG lines (dbg_comm.v) should be included with the adv_jtag_bridge program, in the rtl_sim/ subdirectory. To use it, a top-level HDL module must be created (by hand) which instantiates both the regular hardware system and the dbg_comm entity, and connects them together. Note the dbg_comm module also provides system clock and reset signals, for ease of use. The standard JTAG TAP must be used when simulating: the FPGA-specific TAP cores do not have external inputs for the JTAG signals.

dbg_comm Signal name	Direction	Description
SYS_CLK	COMM->TAP	Clock for WishBone and OR1200 CPU
SYS_RSTN	COMM->TAP	Reset for WishBone and OR1200 CPU; active low
P_TCK	COMM->TAP	Clock signal for JTAG TAP
P_TMS	COMM->TAP	Mode Select signal for JTAG TAP
P_TDI	COMM->TAP	Data In to JTAG TAP
P_TDO	TAP->COMM	Data Out from JTAG TAP
P_TRST	COMM->TAP	Reset signal for JTAG TAP; active low

In order to use a file IO simulator connection, select the cable “rtl_sim” when starting adv_jtag_bridge. Be sure to specify the same directory for the communication files on the command line that is specified in the dbg_comm.v verilog file; the verilog file must be changed to match user preferences before compilation. All other setup steps are the same as described above for hardware debugging. Remember that your simulation must be actually running in order to communicate with the bridge program and GDB.

The adv_jtag_bridge program waits for an acknowledgment each time it writes a signal to the simulator. As such, the bridge program and the simulation may be started in any order. If the simulation is started first, it will run without changing the state of the JTAG lines. If the bridge program is started first, it will attempt to write the first bit to the simulator, then wait for the simulator to acknowledge. The simulation will wait until reset is complete before reading the shared files and reading the bit from the bridge program. Note however that the files are actually created by the bridge program; if they do not exist when the simulator is running, warnings may occur in the simulator.

It should be noted that debugging a simulation using file IO can be very slow. Depending on the capabilities of your workstation and the complexity of the simulated system, the optional self-test

may take an hour or more to simulate. You may wish to use a RAMdisk in order to reduce file IO latency.

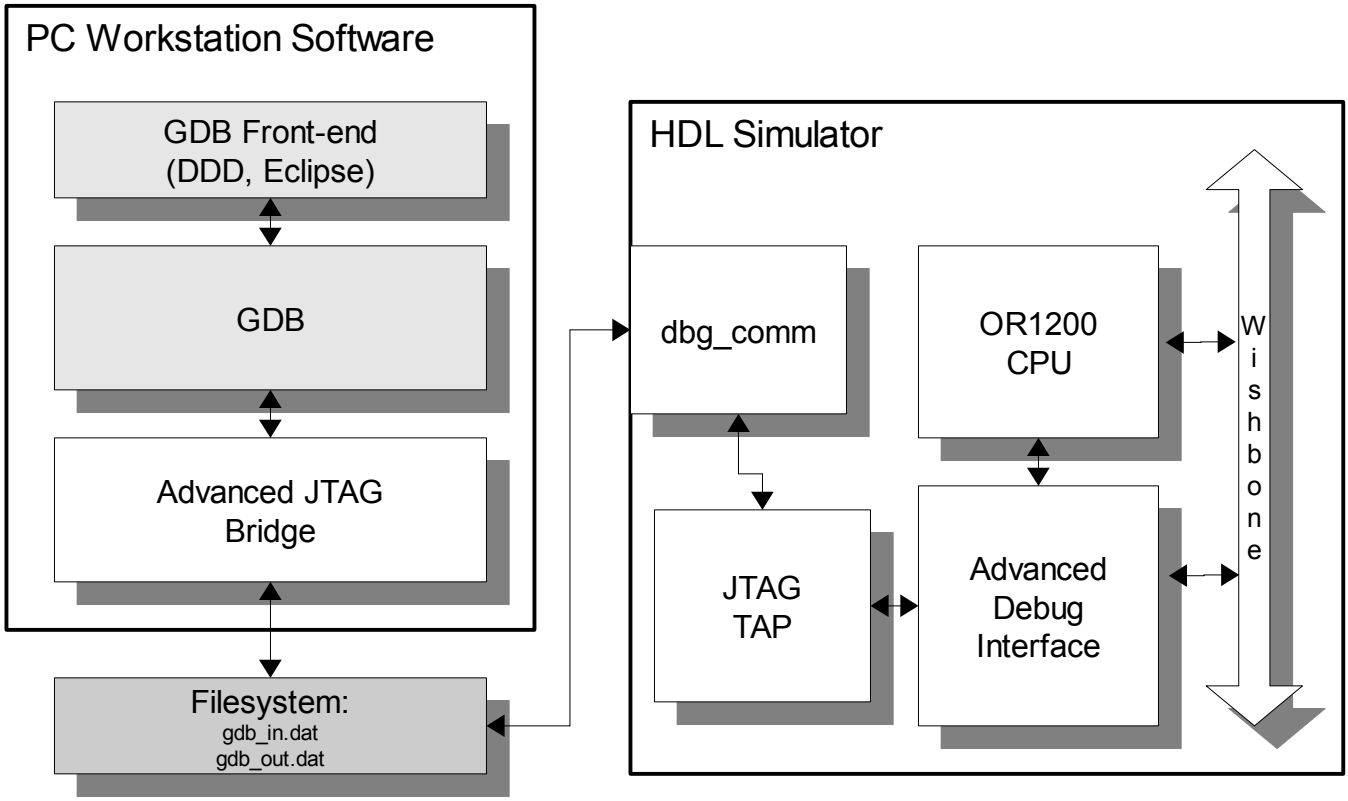

Figure 3: Debugging system block diagram using file IO simulator connection

4.2.2. VPI IO

This method uses the Verilog Program Interface (VPI) to connect a verilog simulation to the bridge program. VPI is an interface which allows arbitrary verilog system tasks to be written by a user in C. The code is compiled into a shared library, which is linked to the simulator at run-time. This allows the newly defined system tasks to be called by verilog code during simulation.

A C library has been implemented which performs communication to the bridge program. The library uses network sockets for communication instead of filesystem IO, and may be faster than file IO. However, this method is more complex to use: your HDL simulator must support UDI / VPI in order for this method to work. Modelsim, NCsim, and Icarus are all known to support VPI. The source code for the C library (jp-io-vpi.c) should be included along with the source for the adv_jtag_bridge program, in the rtl_lib/src/ subdirectory. Because it may be used with several different simulators and operating systems, the method for building the library may vary. Makefiles for some combinations are included in subdirectories of the rtl_lib/ directory, and pre-built binaries may be included as well. If a Makefile is not included for your simulator / operating system, see the documentation for your simulator for instructions on how to build a VPI library for your system.

You will also need to find how to connect the library to your simulator. This step also differs for each simulator. For Modelsim, it is sufficient to place the compiled library in the base directory of

the simulator project, and to indicate the library to be used by setting the simulator's PLIOBJS environment variable (you may also specify VPI libraries in the modelsim.ini file, and on the vsim command line using the "-pliobjs" argument). For other simulators, different library locations and indications may be required. Check your simulator documentation for details.

Similar to the file IO method, a verilog module is added to the hardware system which interfaces to the C library. This module receives commands from the bridge program, sets the JTAG outputs accordingly, and returns the state of the TDO line to the bridge program via the C library. A block diagram of the VPI simulation debug system is shown in Figure 4.

The verilog module which connects to the library and controls the JTAG lines (dbg_comm_vpi.v) should be included with the adv_jtag_bridge program, in the rtl_sim/ subdirectory. To use it, a top-level HDL module must be created (by hand) which instantiates both the regular hardware system and the dbg_comm_vpi entity, and connects them together. Note the dbg_comm_vpi module also provides system clock and reset signals, for ease of use. The standard JTAG TAP must be used when simulating: the FPGA-specific TAP cores do not have external inputs for the JTAG signals.

dbg_comm_vpi Signal name	Direction	Description
SYS_CLK	COMM->TAP	Clock for WishBone and OR1200 CPU
SYS_RSTN	COMM->TAP	Reset for WishBone and OR1200 CPU; active low
P_TCK	COMM->TAP	Clock signal for JTAG TAP
P_TMS	COMM->TAP	Mode Select signal for JTAG TAP
P_TDI	COMM->TAP	Data In to JTAG TAP
P_TDO	TAP->COMM	Data Out from JTAG TAP
P_TRST	COMM->TAP	Reset signal for JTAG TAP; active low

In order to use a VPI connection, select the cable "vpi" when starting adv_jtag_bridge. You may optionally specify the host the VPI library is running on (default is "localhost"), and the port that the VPI library listens for a connection on (default is 4567; this is set for the VPI module in the dbg_comm_vpi.v file at compile time). All other setup steps are the same as described above for hardware debugging. Remember that your simulation must be actually running in order to communicate with the bridge program and GDB.

The VPI library acts as the network server, adv_jtag_bridge acts as a client. As such, the simulation must be started and some simulator time must have elapsed before adv_jtag_bridge can be started; starting adv_jtag_bridge first should result in a network connection error. Also note that the server socket is closed after a connection is made – this means that if adv_jtag_bridge is killed, the simulation must be restarted before it will accept another network connection from adv_jtag_bridge.

It should be noted that debugging a simulation can be slow. Depending on the capabilities of your workstation and the complexity of the simulated system, the optional self-test may take 20 minutes or more to simulate.

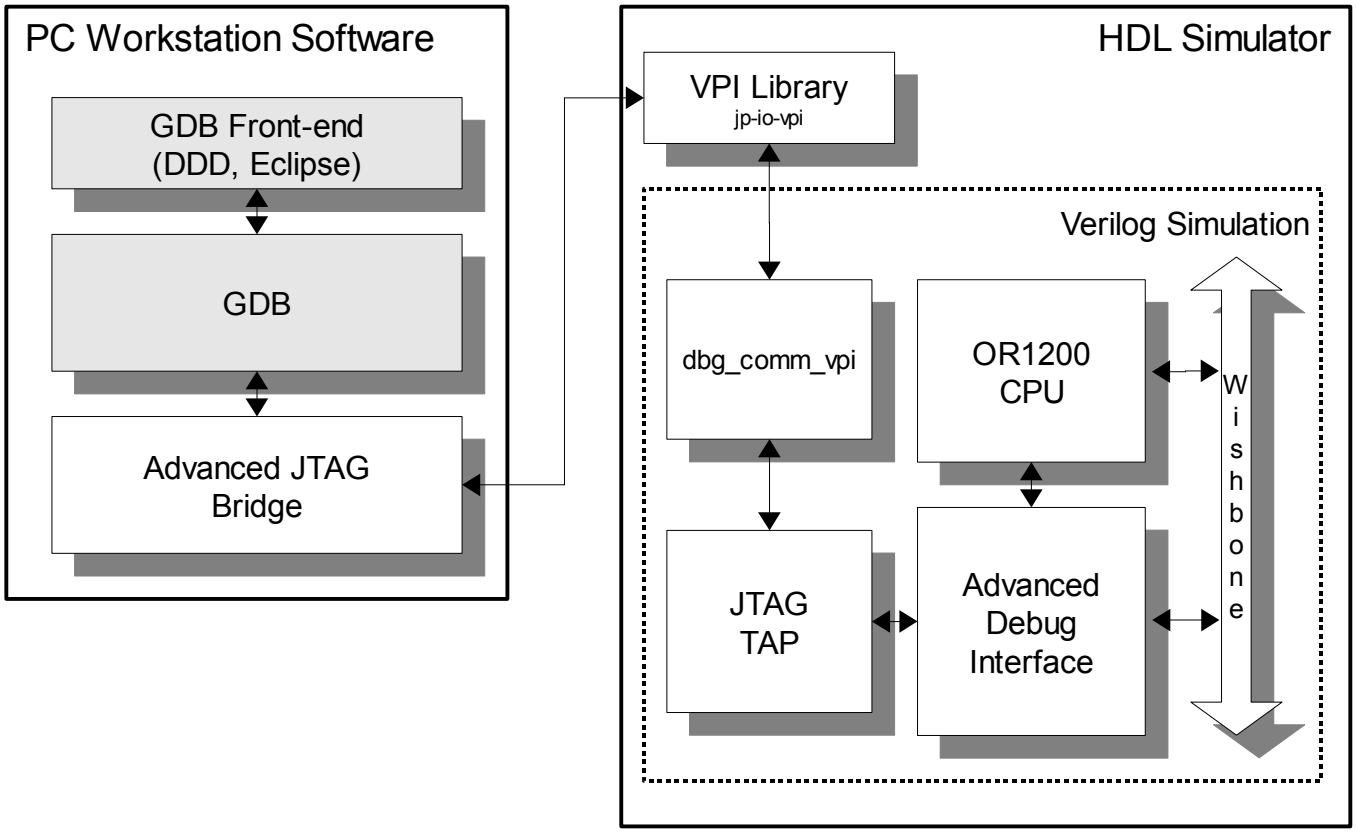

Figure 4: Debugging system block diagram with VPI simulator connection