Advanced Debug Interface

Author: Nathan Yawn
nathan.yawn@opencores.org

Rev. 2.1

March 30, 2010

History

Rev.	Date	Author	Description
1.0	27/06/08	Nathan Yawn	First Draft
2.0	17/01/10	NAY	Added hi-speed mode documentation
2.1	30/3/10	NAY	Added JTAG Serial Port documentation

Table of Contents

1

Introduction

The Advanced Debug Interface (ADI) is a hardware module which creates an interface between a JTAG Test Access Port (TAP) and the system bus and CPU debug interface(s) of a System on Chip (SoC). It is part of the system which allows software running on an SoC to be controlled and debugged by a software debugger such as GDB, running on a separate host PC. This debugging system allows the SoC to be debugged via direct hardware connection, and does not require the "GDB stub" software running on the SoC. A block diagram of this system is shown in Figure 1.

The external interface to the Advanced Debug Interface is based on IEEE Std. 1149.1, Standard Test Access Port and Boundary Scan Architecture. A JTAG TAP is required to link the ADI to an external JTAG cable. This TAP may be a stand-alone TAP[1], or it may be a specialized unit such as an Altera Virtual JTAG or a Xilinx Internal BSCAN unit. The ADI appears as a data register within the TAP.

Internally, the ADI has connections to both a WishBone bus and to one or more CPU debug interfaces. The WishBone interface does not use any of the burst features of the bus; it should therefore be compatible with any version of the bus (B.1 or B.3 as of this writing). The CPU interface is designed to connect to an OR1200 processor, or any other CPU which uses the same debug ports. Note that there are two versions of the OR1200 debug interface; the ADI is designed to use the newer version, which includes the strobe and acknowledge (dbg_stb and dbg_ack) signals.

The ADI also includes a JTAG Serial Port (JSP) module, to assist in software debugging. The JSP is a low-speed, unreliable communication channel between software running on the target SoC and the debugging host PC. The JSP appears as a (mostly) 16550 UART-compatible target on the SoC WishBone bus. Data can be transferred bi-directionally between this WishBone target and the debug host PC via JTAG, eliminating the need for a dedicated RS232 channel for program debug logging, etc.

1 Note that the ADI is incompatible with the stand-alone TAP written by Igor Mohor. Please use the version modified by Nathan Yawn instead.

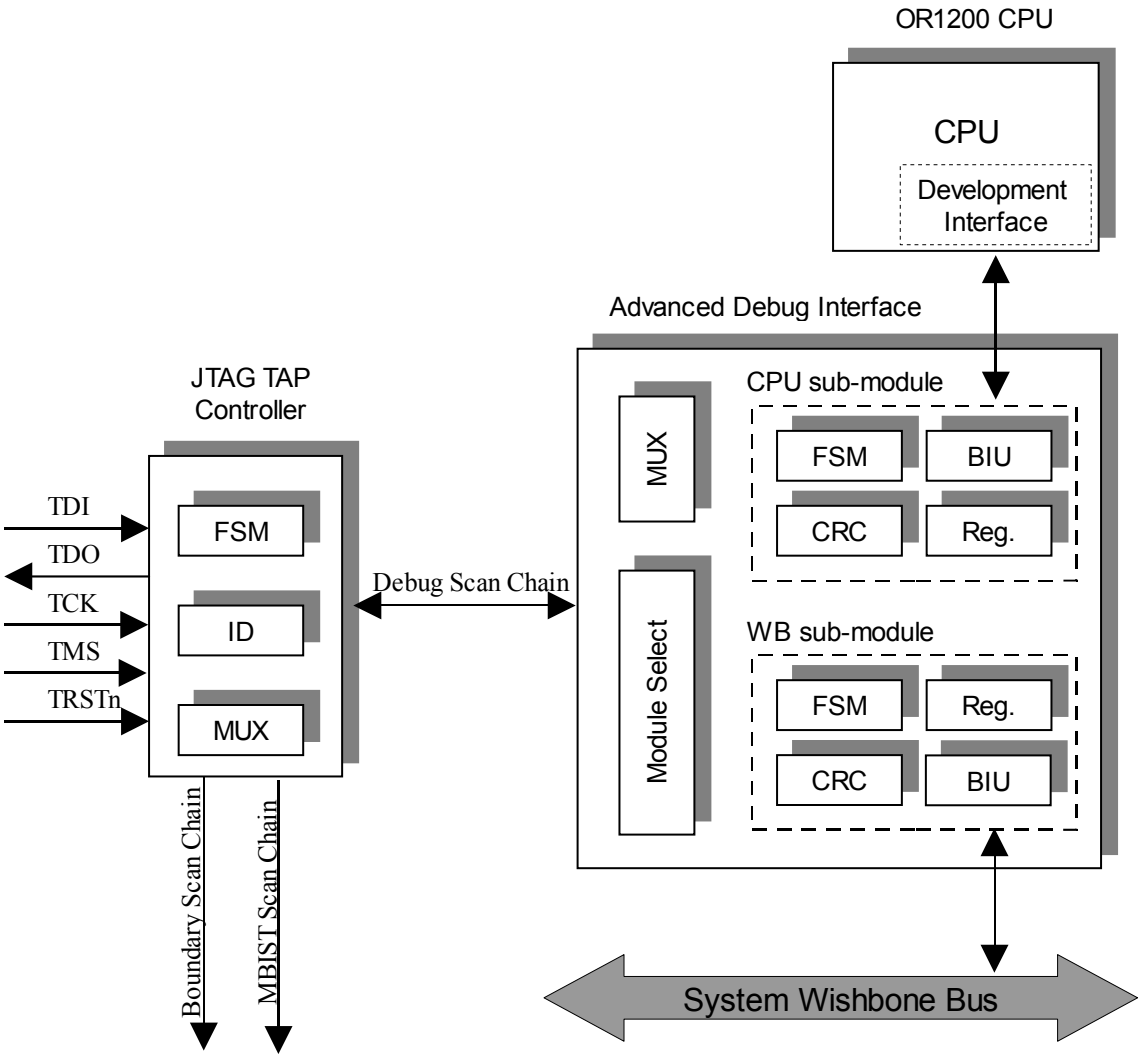


Figure 1: Block diagram of the complete debug system hardware

2

Architecture

The Advanced Debug Interface is built with a modular architecture, for flexibility and expandability. It consists of a top-level module, and several sub-modules designed to interface with individual SoC subsystems. The sub-modules currently include the WishBone module, the OR1200 module, and the JSP module.

The top-level module contains the sub-modules, and a register to set the active sub-module. In order to send a command to a sub-module, it must first be made active by setting this top level register; only one sub-module may be active at a time. Zero or more instances of any type of sub-module are valid (the default is one WishBone sub-module, one OR1200 CPU sub-module, and one JSP sub-module). The top-level module also contains the input shift register, which holds incoming serial data from the TAP. The value in the input shift register is available to all sub-modules. Note that as per the JTAG specification, all serial transfers are LSB-first.

Sub-modules generally consist of two parts: internal registers, and a bus interface. Internal module registers may contain information about the status of the module (such as the error register in the WishBone module), or they may control external I/O lines (such as the reset and stall lines from the OR1200 module). Each sub-module using one or more registers contains an index register, which enables one internal register at a time for reading or writing. Internal registers are selected, read, and written by sending commands to a sub-module through the TAP.

The bus interface of most sub-modules is designed to allow the TAP to read or write data from or to a bus as quickly as possible. Note that 'bus' in this case does not necessarily mean a WishBone bus; the bus interface of the OR1200 module connects to the processor's SPR bus. All bus transactions are 'burst' transactions from the external / TAP side: a setup command is first sent to a sub-module, then the entire block of data is streamed into or out of the sub-module without further control action. Different sub-modules may provide burst transactions using various word lengths. Burst data is CRC-protected. A block diagram of the general module structure is shown in Figure 2. Note that the JSP module is an exception to this, with a very different bus interface.

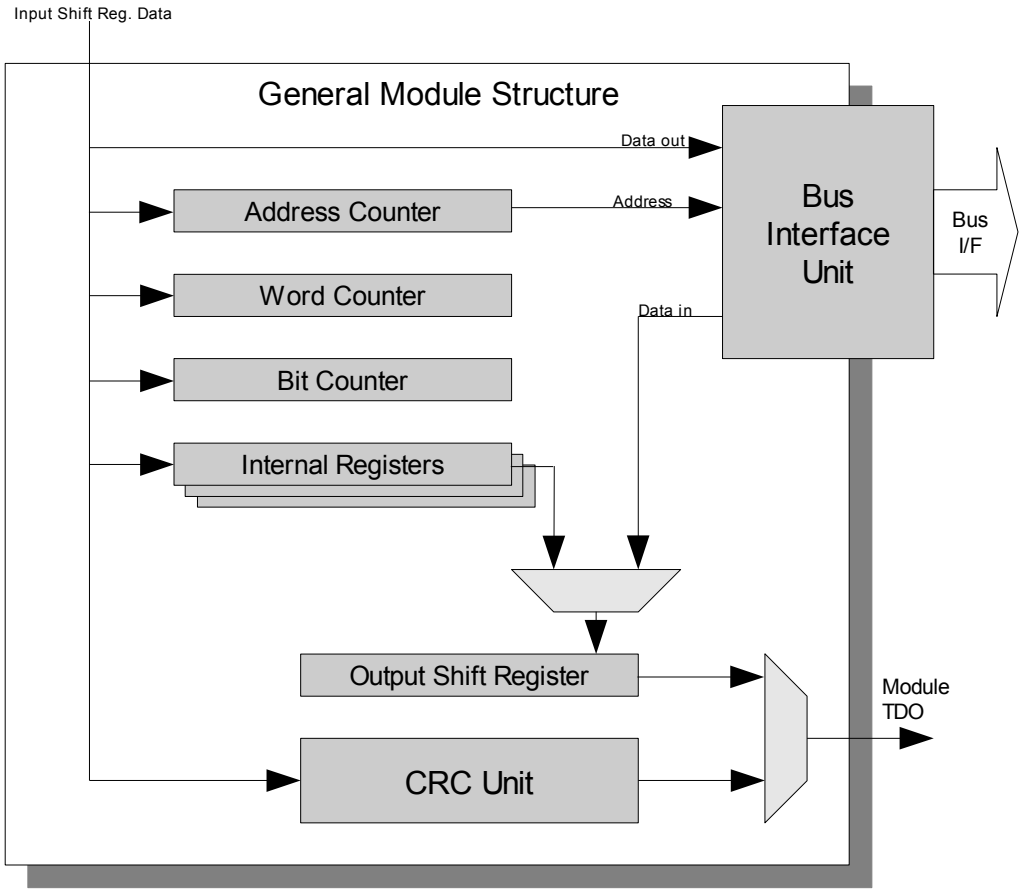


Figure 2: General module structure

In order to support JTAG scan chains with more than one device, commands are usually executed by the sub-modules when the TAP moves through the UPDATE_DR state. This allows a software driver to add the necessary bits to the end of a serial bitstream to position the command at the correct place in the scan chain.

The exception to this is burst data, due to its unknown (and potentially very large) size. To do a burst transaction, a burst command is first sent to a module, and executed by moving the TAP through the UPDATE_DR state. The next time the TAP goes into the SHIFT_DR state, 'burst mode' is active. In burst mode, bus data is immediately clocked into or out of the module, and the next bus transaction (or the end of the transaction) is determined by internal counters. In order to support multi-device chains, a "start bit" feature was added to burst mode. During burst writes, the module will not start its counters or collect write data until after the first '1' (a "start bit") is encountered in the bitstream. Since TAP devices in BYPASS mode will initially shift out a '0', this means that these extra bits from other devices will be ignored by the ADI module. Once the

module sees the start bit, it begins to capture write data (the start bit is discarded). Burst reads require no such added feature, as a software cable driver may simply discard the appropriate number of bits before beginning to capture read data. However, the first status bit of a burst read may be used as a start bit during burst reads, see the API sections on burst reads for details.

There are two more things to consider when using the ADI in a scan chain with multiple devices. First, all other devices should be in BYPASS mode when using the ADI – otherwise, a false start bit could get sent to the ADI during a burst write, corrupting the data. Second, the ADI data register is not a through shift register – that is, the serial TDI input of the ADI is not directly connected to the TDO output. This means that data shifted into the ADI will never appear at the output. As such, the ADI should never be active when other devices on the chain are in use.

Four modules are currently implemented. The following sections give details for each type.

2.1 Top Module

The top-level module is the simplest of the modules. It does not have a bus interface, and has only a single register. This register is called the "module select register", and is used to select the active sub-module. The top module does not use command opcodes the way the sub-modules do. Instead, a single bit in the input shift register (the MSB) indicates whether the command is a write to the select register, or a command to a sub-module (in which case the command is ignored by the top-level). The value in the select register cannot be read back.

The top-level module provides enable signals to all sub-modules, based on the value in the module select register. The value of the input shift register is also provided to all modules. Finally, the serial TDO output of the ADI is selected from the sub-module TDO outputs, based on the module select register. A block diagram of the top-level module is shown in Figure 3.

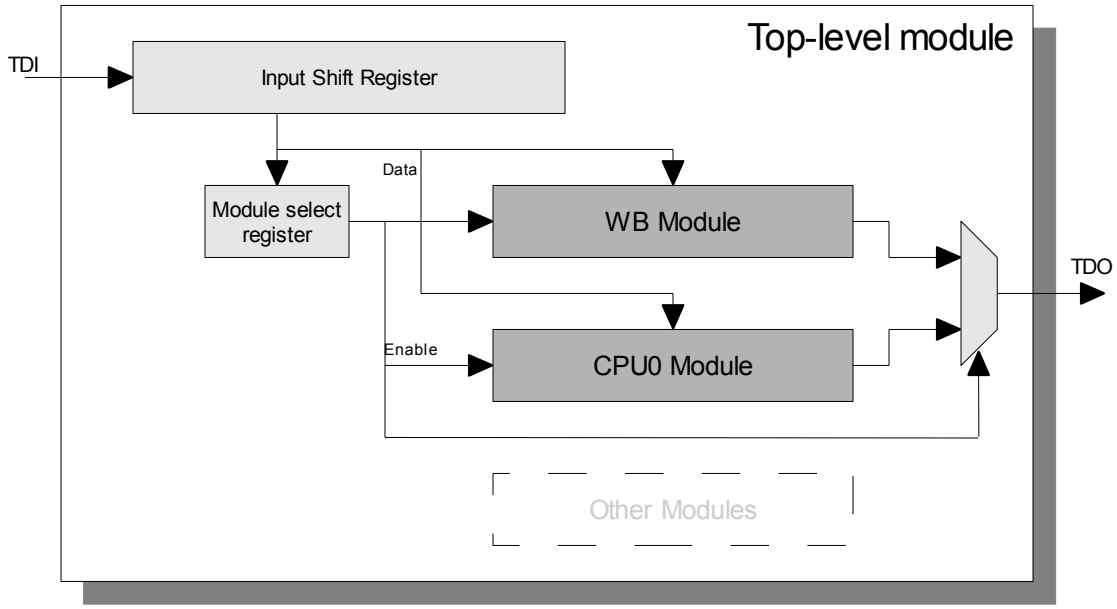


Figure 3: Block diagram of top-level module

2.2 WishBone Module

The purpose of the WishBone module is to provide a software debugger access to the SoC's memory system, allowing it to load code, examine and change program data, and set software breakpoints. The WishBone module has a bus interface which follows the WishBone standard. Because the JTAG interface is relatively slow, burst accesses on the WishBone bus interface are not used, therefore the interface should be compatible with all versions of the WishBone standard (through B.3 as of this writing). The WishBone module allows 8-, 16-, and 32-bit reads and writes over the WishBone bus interface. The WishBone Module uses a 32-bit address and a 32-bit data bus, and allows burst transfers of up to 65535 words (262140 bytes).

Since the JTAG clock is asynchronous to the WishBone bus clock, transactions are synchronized between clock domains. This clock differential may cause other problems, however, if the JTAG clock is much faster than the WishBone clock. For efficient operation, it must be possible to complete a WishBone write (plus 4 JTAG-domain clock cycles for control and synchronization) in less time than it takes to shift the next word in via the TAP. The problem is magnified when using 8-bit words. If the bus is not ready by the time the next word has arrived, then the attempt to write the next word will fail.

The ADI may be synthesized in one of two modes: Legacy (or “careful”) mode, and Hi-Speed mode. Which mode is used affects how overflow conditions are detected. In legacy mode, an overflow condition during a burst write can be detected by reading

back a status bit after writing each word. If the status bit is true, then the transaction has succeeded (the WishBone bus was ready to accept the word) and the burst should continue. If the status bit is false, then an overflow has occurred, and the software driver should retry part of the burst, starting with last word written.

In hi-speed mode, there is no status bit between data bytes, and overflows cannot be detected until after the burst transaction is finished. In hi-speed mode, an overflow is detected and treated in the same way as a WishBone bus error, and is captured by the module's "error register," described below.

In legacy mode, underflow conditions during burst reads are avoided by use of a "ready" bit, which is read by the software driver before each word is shifted out of the ADI. When data is not yet available, the WishBone module shifts out zeros. As soon as a data word is read from the WishBone bus, the module shifts out a one, indicating that data is ready (the driver should discard this '1' and all preceding ready bits). Bus data follows immediately after a true ready bit is sent.

Hi-speed mode eliminates all ready bits in a burst read except the first one – driver software must only wait for a single valid ready bit at the start of a read transaction before transferring all of the data. An underflow condition in a hi-speed mode burst read is treated as a WishBone bus error, and captured by the "error register," described below.

Which mode you select (legacy or hi-speed) depends on the relative clock speeds of your system's WishBone bus and JTAG interface. Systems with WishBone clock slower than the JTAG clock may require legacy mode. Most systems should select hi-speed mode, especially if a USB-based JTAG cable is used. A system in hi-speed mode using a USB JTAG cable is generally able to transfer WishBone data an order of magnitude faster than a system with a legacy-mode ADI.

The WishBone module contains a single 33-bit internal register, called the "error register," which is used to detect errors on the bus interface. During a burst read or write, the WishBone error (wb_err) bit is sampled at the end of each bus transaction. If the error bit is ever true, then the error bit (bit 0) in the error register is set, and the address of the failed transaction is captured into the other 32 bits of the error register. Once the error bit has been set, it can only be cleared by performing an internal register write to the WishBone module - the addresses of subsequent errors will not be captured. Thus, when the error bit is set, the error register contains the address of the first bus error encountered since the bit was last reset. Note that when the ADI is synthesized in hi-speed mode, the error bit will also be set on an underflow or overflow condition.

The error register should be reset before each bus transaction (or after each time the error bit is found set), then checked after each burst transaction. Because the error bit is the least-significant bit, a software driver may read only 1 bit in order to determine whether or not an error condition exists. If true, then the driver may read out the 32-bit error address, and retry the part of the burst starting from that address.

The WishBone sub-module includes a CRC calculation, which is used to protect burst data. During burst transactions, an internal word counter determines when all of the data for a burst has been transferred. If a burst read has just completed, the module then begins to shift out a 32-bit CRC, which the host may compare to a locally-generated CRC. If the completed transaction is a burst write, the module accepts a 32-bit CRC from the host, then shifts out a single bit indicating whether or not the CRC received matched its internal CRC computation. Note that the CRC is done only on the burst data; the preceding burst command, and all start and ready bits, are ignored.

The CRC polynomial is 0xEDB88320. This is bitwise-reversed from many implementations; this is because we compute the CRC LSB-first (also reversed from other implementations). The LSB-first calculation was used to reduce hardware and routing requirements in the CRC module.

2.3 OR1200 CPU Module

The OR1200 CPU sub-module is designed to allow a software debugger to access the internal registers of a CPU, to stall and reset the processor, and to take control when a breakpoint occurs in software. The OR1200 module may also allow access to the CPU's hardware breakpoint and watchpoint configuration and trace buffer, if the CPU has been synthesized with these features – see the OR1000 architecture specification and the OR1200 implementation document for details.

The OR1200 sub-module is based on the WishBone module, and is therefore similar. The bus interface connects to the debug interface of the OR1200, which allows access to the processor's SPR bus. Accesses to this bus are performed by ADI burst transactions. Reads, writes, clock synchronization, ready bits, status bits, overflow, underflow, and CRC computations are all handled exactly as they are in the WishBone module. The OR1200 debug interface bus does not have an error indicator bit. Thus, the OR1200 module does not have an internal “bus error” register.

The OR1200 module also has “legacy” and “hi-speed” modes, which work in the same way as these modes in the WishBone module. The only difference is the lack of an error register in the OR1200 module. This means that in hi-speed mode, overflows and underflows during burst transactions cannot be detected. It may be possible to guess whether under- or overflows are occurring: if these errors are being reported during WishBone transactions, then it's likely they are also occurring during OR1200 transactions, and a legacy-mode ADI should be used instead. Similarly, if you notice strange behavior or unexplained data while debugging, it may be worthwhile to try legacy mode.

The OR1200 module allows the user to set and clear the CPU reset bit, to stall the CPU, and to capture breakpoints in the CPU. This is done through a module internal register, called the "CPU status register." Bit 1 of this register is the reset bit. When this bit is true, the cpu_rst_o output bit is set true, and vice-versa. This bit is set and cleared only by internal register writes via the ADI. Note that this bit is synchronized between clock domains.

Bit 0 of the CPU status register is the stall bit. When set, the cpu_stall_o output of the ADI is also set, and vice versa. This bit may be set and cleared via internal register access to the ADI. This bit may also be set by the CPU: when the cpu_bp_i input from the CPU goes high (indicating a breakpoint), the stall bit in the register is set, and the stall output set high. This effectively transfers control of the CPU to the ADI, which may perform various debugging operation before clearing the stall bit via internal register access, allowing the CPU to continue normal execution.

2.4 JTAG Serial Port Module

The JTAG Serial Port (JSP) module is designed to allow users to get debug logging from their programs running on the target SoC, and to enter simple commands, without the need for a separate, dedicated RS232 serial connection. This makes it possible to create a basic SoC on an FPGA using zero dedicated external pins (assuming a built-in TAP such as the Altera virtual JTAG core).

The JSP architecture differs significantly from the WB and OR1200 modules. A block diagram of the JSP is shown in Figure 4. The JSP does not provide direct access to an internal SoC bus. Instead, it is designed to allow bytes to be read and written to a pair of 8-byte FIFOs (one for each direction, ADI->WB "input" and WB->ADI "output"). These FIFOs are also be read and written to by software running on the SoC. The interface to the FIFOs on the SoC side is presented as a WishBone target interface, with a register format which is similar to a 16550 UART (the differences are discussed below).

The JSP module protocol was designed to be as fast as possible, with USB JTAG cables in mind. The JSP module has no burst setup command, nor a pure data burst mode. The two are instead combined into a single transaction. After a TAP state transition from CAPTURE_DR mode to SHIFT_DR mode, the first 8 bits transferred are called the "counts" byte. The counts byte transferred out of the ADI contains the amount of free space (in bytes) available in the ADI->WB FIFO in the least-significant nibble, and the number of bytes available for reading (WB->ADI) in the most-significant nibble. The counts byte transferred into the ADI includes the number of bytes the client intends to write to the ADI->WB FIFO this transaction in the most-significant nibble.

Once the "counts" byte has been transferred, the internal counts in the JSP module are initialized, and data transfer begins. Bytes to be written into the input (ADI->WB) FIFO are transferred into the ADI at the same time that bytes to be read from the output (WB->ADI) FIFO are transferred out. Data will stop being written to the input FIFO when either all bytes specified in the input counts byte have been written, or the FIFO becomes full, whichever comes first. Valid data will stop being transferred out when the WB->ADI FIFO becomes empty. Note that this means that all WB->ADI bytes available must be read each transaction; early termination will result in at least 1 byte being lost.

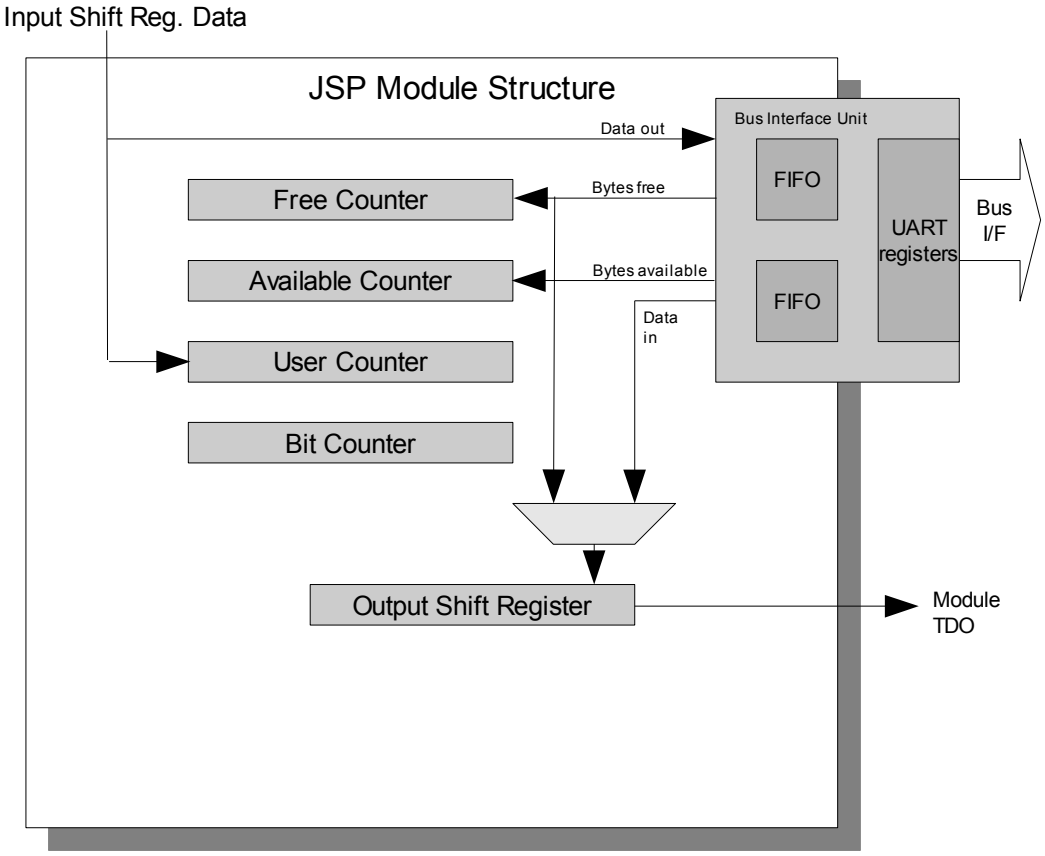


Figure 4: JSP Sub-Module Structure

The data transfer state continues until the TAP enters the UPDATE_DR state (though bits will only be transferred while the TAP is in SHIFT_DR). The JSP module can accept excess input without error, and will simply provide invalid data after all available bytes have been read from the output FIFO. Thus, a transaction may safely continue indefinitely. Practically, this means that excess data in either direction will not cause an error if the number of bytes to be transferred in and out differ.

This combined, single transaction for both counts and data allows the entire JSP communication cycle to occur in a single USB transaction. The driver software need not transfer the counts byte and the data separately – since excess data will always be ignored, it is always safe to transfer 9 bytes (1 counts byte plus 8 data bytes). The driver software can calculate the actual number of bytes written and received after the transaction. Non-USB cables, which are speed-limited by number of bits transferred rather than by number of USB transactions, may choose to transfer just the counts byte first, then calculate the minimum number of bits to transfer before beginning the data phase of the transaction.

The JSP module does not have module registers like the WishBone and OR1200 modules – there is nothing which can be selected, read, or written individually. Thus, there is no register select register, and no register select command for this module. The JSP also uses no CRC, and has no CRC module.

Because there is no setup command, there is no way for the JSP module to prevent the top-level module from interpreting a JSP transaction as a top-level command. This means that to keep the JSP module selected, each JSP transaction must end with a '0' bit. If the most-significant bit of the last byte transferred out is '0', then no additional bit needs to be added. If the last data bit is a '1', then an additional '0' bit must be added to the end of the transaction. This added bit will be ignored by the JSP module, and will prevent the top-level module from interpreting the JSP transaction as a module-select command.

Because the JSP is not a true UART, many of the registers on the WishBone interface which would be required for a 16550 are unnecessary for the JSP. To save logic resources, these unnecessary registers are not implemented in the JSP. The key differences follow:

- Neither of the two divisor registers (**DLL**, **DLM**) are implemented. Writes to these registers are ignored; reads will return undefined data. The **DLAB** bit in the **LCR** is implemented, however – this prevents attempted writes to the **DLL** and **DLM** from reading or writing the data FIFOs.
- In the Interrupt Enable Register (**IER**), only the Receive Data Available and Transmit Holding Register Empty bits are implemented. Changes written to other bits will be ignored.
- Despite the presence of 8-byte FIFOs, the JSP will act as if it has no FIFOs. The two FIFO enable bits in the **IIR** will always read '0'. The **LSR** will show data ready (and the **IER** will show a receive interrupt) any time there are 1 or more bytes in the WB receive FIFO. The **LSR** will show both transmitter and transmit holding register empty any time there are less than 8 bytes in the WB write FIFO. The transmit interrupt, however, will only be activated when the WB write FIFO transitions from 1 byte to 0 bytes. As per the 16550 spec, a transmitter empty

interrupt is cleared by reading the **IER**, and is re-armed by writing 1 or more bytes to the WB write FIFO.

- The **DLAB** bit is the only read/write bit implemented in the **LCR** register. Writes to other bits will be ignored. Reads will return static data, reflecting a setup of 8 data bits, 1 stop bit, no parity.
- The **MCR** and **SCR** registers are unimplemented. Writes will be ignored, reads will return a static 0x00.
- The **MSR** is implemented statically. Writes are ignored, reads will return a static 0xB0, reflecting CD, DSR, and CTS lines true, RI false, and no delta of any bit.
- The **LSR** is incompletely implemented, and will never show an overrun error, a parity error, a framing error, a break interrupt, or an error in the receive FIFO. The 'transmitter empty' and 'transmit holding register empty' bits will always have the same value, and will be set any time there are less than 8 bytes in the WB transmit FIFO.

For a full accounting of which bits of which registers are implemented, see section 3.7, “JSP Module Registers”.

The JSP does not have a 'legacy' mode like the WB and OR1200 modules. However, there is an option to disable compatibility with multi-device JTAG chains, which causes a change in protocol. To create compatibility with multi-device JTAG chains, a '1' start bit is written to begin all JSP transactions. However, this extra bit adds multiple USB transactions for some USB JTAG cables (USB-Blaster in particular), causing a slowdown in JSP communication. As such, the ADI includes the option to disable multi-device chain compatibility in the JSP sub-module. Make sure your synthesis options match your JTAG chain configuration. Also, be sure to configure your driver program to use the right protocol to match your hardware configuration options.

3

API

This section gives information on the commands, opcodes, and data formats of the Advanced Debug Interface. There are four major sections: the top-level ADI, the WishBone module, the CPU module, and the JSP module. All commands and registers are shown with the least-significant bit to the right, and all commands are shifted out LSB-first. Data is also shifted in to and out of the ADI LSB-first. All commands are interpreted when the TAP passes through state UPDATE_DR. All commands are interpreted as MSB-aligned. This means that the minimum number of bits for a command may be shifted into the ADI, without consideration for the length of the data register.

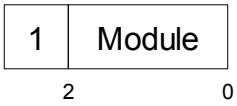
Note that the ADI must be reset before any commands will be accepted or executed. If the system TAP has an output indicating the "Test Logic / Reset" state, it is recommended that this be used as the ADI reset input. If this signal is not available, it is recommended that the ADI reset input be connected to the OR1200 or WishBone bus reset signal.

3.1 Top-level Commands

3.1.1 Module Select command

This command is used to select which one of the sub-modules is active. Only the active sub-module will process commands sent to the ADI. This command should be sent before any other command is sent to any sub-module.

Figure 5: Module Select Command format



Bit #	Access	Description
2	W	**Top-Level Select** Set to '1' to select the top level module
1:0	W	**Module** Number of the module to select. The following modules are valid: 0x0 = WishBone module 0x1 = CPU0 module 0x2 = CPU1 module (optional) 0b3 = JSP module (optional)

Table 1: Module Select command format

3.2 WishBone Commands

The WishBone sub-module uses a standardized command format. Each command must have a zero as the MSBit, to differentiate it from a module select command. In the next four most-significant bit positions is a 4-bit opcode, which indicates the operation to be performed. Following the opcode are zero or more data values, whose length and meaning are command-specific. Table 2 summarizes the opcodes supported in the WishBone module.

OPCODE	Operation
0x0	NOP
0x1	Burst Setup Write, 8-bit words
0x2	Burst Setup Write, 16-bit words
0x3	Burst Setup Write, 32-bit words
0x5	Burst Setup Read, 8-bit words
0x6	Burst Setup Read, 16-bit words
0x7	Burst Setup Read, 32-bit words
0x9	Internal register write
0xD	Internal register select

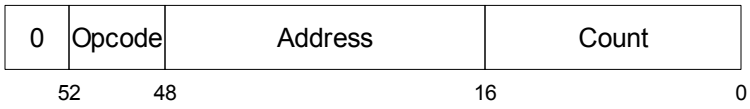
Table 2: WishBone module command opcode summary

3.2.1 Burst Setup

A burst setup command prepares the WishBone module to do either a read or write burst, in order to move data to or from a consecutive sequence of addresses on the WishBone bus. The word size of the burst is decoded from the opcode. The WishBone module can use an 8-, 16-, or 32-bit word size. After each individual word transfer during a burst, the address counter in the ADI is incremented according to the word size: it will be incremented by 1, 2, or 4, for 8-, 16-, and 32-bit words respectively.

After a burst setup command has been executed (in the UPDATE_DR state), the WishBone module will enter 'burst read' or 'burst write' mode, the next time the TAP enters SHIFT_DR mode. Whether read or write mode is used depends on the opcode sent with the burst setup command. Details on burst read and burst write modes are in the following sections.

Figure 6: Burst Setup command



Bit #	Access	Description
52	W	**Top-Level Select** Set to '0' for all sub-module commands
48:51	W	**Opcode** Operation to perform. The following are valid burst setup operations: 0x1 = Burst Write, 8-bit words 0x2 = Burst Write, 16-bit words 0x3 = Burst Write, 32-bit words 0x5 = Burst Read, 8-bit words 0x6 = Burst Read, 16-bit words 0x7 = Burst Read, 32-bit words
47:16	W	**Address** The first WishBone address which will be read from or written to
0:15	W	**Count** Total number of *words* to be transferred. Note that this means that the total number of *bits* transferred depends both on this field, and on the opcode. Must be greater than 0.

Table 3: WishBone module Burst Setup command format

3.2.2 Burst Write

The next time the ADI is accessed after sending a valid burst write command, the WishBone module will be in burst write mode. In this mode, commands and data are not interpreted or executed on transition through UPDATE_DR. Instead, counters are used to determine position in the bitstream, and a word is written to the WishBone as soon as it has been transferred in via JTAG. The total length of a burst write transfer depends on the word size (set by the opcode) and the count fields in the burst setup command; for a word size of n and a transfer of m words in hi-speed mode, the total length will be $(n * m) + 34$. In legacy mode, the total transfer size will be $((n + 1) * m) + 34$.

The first bit transferred in a burst write is a '1' start bit. This tells the WishBone module to begin counting bits, and is required due to the possibility of multiple devices on the JTAG chain. After the start bit, one word of data is transferred into the ADI. In legacy mode, this word is followed by a status bit, which is transferred *out* of the ADI, to be read by the software driver. The status bit tells the user whether the WishBone was ready to accept the word just transferred; when true, the bus was ready. When false, the

word was not written to the WishBone, and the software driver should retry the transfer, starting from the failed word. Note that timing of the actual reception of the status bit may vary, depending on the number of other devices on the JTAG chain.

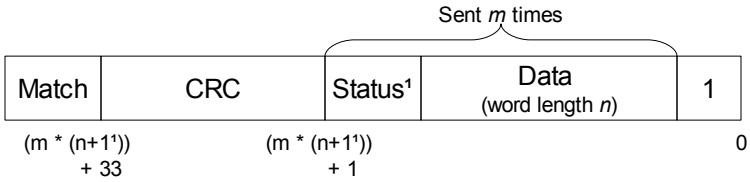
In legacy mode, data transmission continues to alternate data word and status bit until all words have been transferred. In hi-speed mode, each data word is immediately followed by another data word, until all words have been transferred. Immediately following the last status bit (in legacy mode) or data word (in hi-speed mode), a 32-bit CRC code is transferred into the WishBone module. This CRC is compared with a CRC computed internally to the WishBone module. After the CRC is transferred in, a single bit is transferred out of the ADI, indicating whether or not the CRC written matched the CRC calculated. A burst write transaction may be aborted at any time by moving the TAP through the UPDATE_DR state.

The CRC protects only the data bits in a burst transaction; commands and status bits are not included in the CRC computation. The CRC resets before each burst transaction. For more information on the CRC calculation, see chapter 6.

WishBone bus errors are captured during a burst, but the information is not transferred during the burst transaction. After a burst, the user should check the WishBone module error register to see if a bus error (or an overflow, in hi-speed mode) occurred during the burst, and if so, at what address. See the section on WishBone sub-module internal registers for details on the error register.

Note that extra bits sent at the end of a burst write are ignored; thus, the user need not worry about sending a valid or safe operation/opcode at the end of the burst transaction.

Figure 7: Burst Write format



[1] *Only in legacy mode*

Bits	Access	Description
1 bit	R	**Match** '1' if CRC sent matches internal CRC computation, '0' if not
32 bits	W	**CRC** 32-bit CRC computed on all of the data bits of the burst
1 bit	R	**Status** *(legacy mode only)* '1' if the most recently sent data word was written to the WishBone, '0' if the bus was not ready. Sent *m* times, once after each data word.
n bits	W	**Data** Data word. Length specified by the opcode in the burst setup command. Sent *m* times.
1 bit	W	**Start Bit** Set to '1' to indicate the start of a burst write.

Table 4: WishBone module burst write format

3.2.3 Burst Read

The next time the ADI is accessed after sending a valid burst read command, the WishBone module will be in burst read mode. In this mode, commands and data are not interpreted or executed on transition through UPDATE_DR. Instead, counters are used to determine position in the bitstream, and a word is read from the WishBone while the previous data word is transferred out via JTAG. The total length of a burst read transfer depends on the word size (set by the opcode) and the count fields in the burst setup command; for a word size of *n* and a transfer of *m* words in hi-speed mode, the total length will be $(n * m) + 33$. In legacy mode, the total transfer size will be $((n + 1) * m) + 32$. This assumes data is ready each time a 'ready' bit is read.

The first bit (or bits) transferred during a burst read, whether legacy or hi-speed mode, is a status bit. This bit indicates whether or not data from the WishBone is ready to be transferred out via JTAG. The WishBone module will send '0' bits until a word is ready, then send a single '1' bit. One data word will follow a '1' status bit. This bit is required even in hi-speed mode, because the clock domain synchronization to the WishBone bus interface requires several JTAG clocks, which means that data will not be ready for reading when the first status bit is sent. However, status bit(s) are only used before the first data word in hi-speed mode; the second data word follows immediately after the first, and so on.

In legacy mode, a status bit (or bits) is transferred before each data word. Data transmission continues to alternate status bits and data words until all words have been transferred.

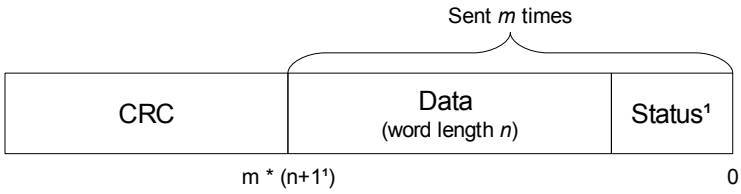
Immediately following the last data word, a 32-bit CRC code is sent from the WishBone module to the driver software. The driver software should compare the CRC received with one computed internally, to determine whether or not the complete transaction must be retried.

The CRC protects only the data bits in a burst transaction; commands and status bits are not included in the CRC computation. The CRC resets before each burst transaction. For more information on the CRC calculation, see chapter 6.

WishBone bus errors are captured during a burst, but the information is not transferred during the burst transaction. After a burst, the user should check the WishBone module error register to see if a bus error occurred during the burst, and if so, at what address. See the section on WishBone sub-module internal registers for details on the error register.

Note that extra bits sent at the end of a burst read are ignored; thus, the user need not worry about sending a valid or safe operation/opcode at the end of the burst transaction.

Figure 8: Burst Read format



[1] *Only sent once in hi-speed mode*

Bits	Access	Description
32 bits	R	**CRC** 32-bit CRC computed on all of the data bits of the burst
n bits	R	**Data** Data word. Length specified by the opcode in the burst setup command. Sent *m* times.
1 bit	R	**Status** Read '0' until a word is ready to be sent, then a single '1' bit is sent before the data word. In hi-speed mode, this is only sent before the first data word. In legacy mode, this is sent before each data word.

Table 5: WishBone module burst read format

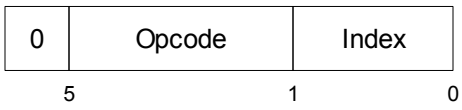
3.2.4 Register Select

A register select command will make the module-internal register with the given index active (in the currently selected sub-module). While any register in the current module can be written with a single command, only the active register can be read.

When the TAP enters CAPTURE_DR mode, the WishBone module captures the value of the active register into the output shift register, allowing the value to be read when the TAP is in SHIFT_DR mode. This will happen each time a command is sent to the WishBone module, unless the previous command was a burst setup.

The register select command uses the same top-level select bit and opcode format as the other WishBone module commands. In this case, the opcode is followed by a 1-bit value, which is the index of the register which should be made active. See the API section on registers for a complete listing of all registers in the WishBone module, and the meaning of each.

Figure 9: Register Select command format



Bit #	Access	Description
5	W	**Top-Level Select** Set to '0' for all sub-module commands
4:1	W	**Opcode** Operation to perform. 0xD = Internal Register Select
0	W	**Index** Index of the register to make active. The WishBone module uses a 1-bit index.

Table 6: WishBone module Register Select command format

3.2.5 Register Read

There is no specific command to read a module internal register; the value of the active register is shifted out every time a command is shifted in. In order to read a particular register, make that register active using the register select command, then read out the value of the register while sending a NOP command. The length of each register may vary, and the meaning of each bit is also register-specific. The register data will be LSB-aligned; that is, the LSB of the register data will be the first bit shifted out of the ADI. This allows the minimum number of JTAG bits to be transferred. It is legal to read more bits than the active register has – the additional bits will have undefined value, and should be discarded. Note that only register data will be transferred out, no command, index, opcodes, start, or status bits will be sent with it. It is legal to abort a register read at any time, provided that a valid command (probably a NOP) is in the correct position in the input shift register.

3.2.6 Register Write

A register write command contains both a register index, and data to be written to the register at that index. The register with the given index will become the active register after this command is executed. Note that the value of the *previously* active register will be shifted out as this command is shifted in. The length of the data field is variable, and depends on the particular register being written. See the API section on registers for a complete description of the registers, their lengths, and their meanings.

Figure 10: Register Write command format

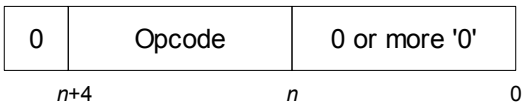
Bit #	Access	Description
5 + *n*	W	**Top-Level Select** Set to '0' for all sub-module commands
(4:1) + *n*	W	**Opcode** Operation to perform. 0x9 = Internal Register Write
n	W	**Index** Index of the register to make active. The WishBone module uses a 1-bit index.
n-1:0	W	**Data** *n* bits of data to write to the register specified by Index. *n* depends on the register being written.

Table 7: WishBone module Register Write command format

3.2.7 NOP

A NOP command will perform no operation. It is included as a "safe" command to shift into the WishBone module while shifting out internal register data. A NOP command consists of five or more zeros, making it easy to send for any length of data read.

Figure 11: NOP command format



Bit #	Access	Description
4 + *n*	W	**Top-Level Select** Set to '0' for all sub-module commands
(3:0) + *n*	W	**Opcode** Operation to perform. 0x0 = NOP
n:0	W	**Zero** Zero or more '0' bits

Table 8: WishBone module NOP command format

3.3 CPU Commands

The CPU sub-module uses the same standardized command format as the WishBone module. Each command must have a zero as the MSBit, to differentiate it from a module select command. In the next four most-significant bit positions is a 4-bit opcode, which indicates the operation to be performed. Following the opcode are zero or more data values, whose length and meaning are command-specific. Table 9 summarizes the opcodes supported in the CPU module.

OPCODE	Operation
0x0	NOP
0x3	Burst Setup Write, 32-bit words
0x7	Burst Setup Read, 32-bit words
0x9	Internal register write
0xD	Internal register select

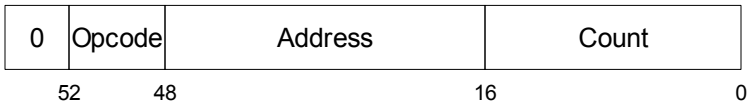
Table 9: CPU module command opcode summary

3.3.1 Burst Setup

A burst setup command prepares the CPU module to do either a read or write burst, in order to move data to or from a consecutive sequence of address on the OR1200's SPR bus. Because all SPRs are 32-bit registers, all burst transfers in the CPU module use 32-bit words. After each individual word transfer during a burst, the address counter in the CPU module is incremented by 1. Note that while the OR1000 architecture defines an SPR address as 16 bits, the OR1200 implementation uses a 32-bit address in its external debug interface. The ADI is designed to use the 32-bit OR1200 implementation.

After a burst setup command has been executed (in the UPDATE_DR state), the CPU module will enter 'burst read' or 'burst write' mode, the next time the TAP enters SHIFT_DR mode. Whether read or write mode is used depends on the opcode sent with the burst setup command. Details on burst read and burst write modes are in the following sections.

Figure 12: Burst Setup command



Bit #	Access	Description
52	W	**Top-Level Select** Set to '0' for all sub-module commands
48:51	W	**Opcode** Operation to perform. The following opcodes are valid burst setup operations for the CPU module: 0x3 = Burst Write, 32-bit words 0x7 = Burst Read, 32-bit words
47:16	W	**Address** The first OR1200 SPR address which will be read or written
0:15	W	**Count** Total number of 32-bit words to be transferred.

Table 10: CPU module Burst Setup command format

3.3.2 Burst Write

The next time the ADI is accessed after sending a valid burst write command, the CPU module will be in burst write mode. In this mode, commands and data are not interpreted or executed on transition through UPDATE_DR. Instead, counters are used to determine position in the bitstream, and a word is written to the CPU SPR bus as soon as it has been transferred in via JTAG. The total length of a burst write transfer depends on the count field in the burst setup command; for a transfer of *m* words in hi-speed mode, the total length will be (32 * *m*) + 34. In legacy mode, the total length will be (33 * *m*) + 34 bits.

The first bit transferred in a burst write is a '1' start bit. This tells the CPU module to begin counting bits, and is required due to the possibility of multiple devices on the JTAG chain. After the start bit, one word of data is transferred into the ADI.

In legacy mode, a data word is followed by a status bit, which is transferred *out* of the ADI, and should be read by the software driver. The status bit tells the user whether the OR1200 was ready to accept the word just transferred; when true, the bus was ready. When false, the word was not written to the SPR bus, and the software driver should retry the transfer, starting from the failed word. Note that timing of the actual reception of the status bit may vary, depending on the number of other devices on the JTAG chain.

In legacy mode, data transmission continues to alternate data word and status bit until all words have been transferred. In hi-speed mode, a data word is followed immediately by another data word, until all words are transferred.

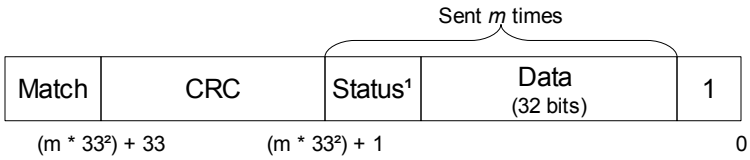
Immediately following the last data word (in hi-speed mode) or status bit (in legacy mode), a 32-bit CRC code is transferred into the CPU module. This CRC is compared with a CRC computed internally to the CPU module. After the CRC is transferred in, a single bit is transferred out of the ADI, indicating whether or not the CRC written matched the CRC calculated. A burst write transaction may be aborted at any time by moving the TAP through the UPDATE_DR state.

The CRC protects only the data bits in a burst transaction; commands and status bits are not included in the CRC computation. The CRC resets before each burst transaction. For more information on the CRC calculation, see chapter 6.

The OR1200 SPR bus does not provide any sort of error indication beyond ready / not ready. As such, there is no error register to be tested after a burst (as opposed to the WishBone module, which has such a register).

Note that extra bits sent at the end of a burst write are ignored; thus, the user need not worry about sending a valid or safe operation/opcode at the end of the burst transaction.

Figure 13: Burst Write format



[1] *Only in legacy mode*

[2] *32 in hi-speed mode*

Bits	Access	Description
1 bit	R	**Match** '1' if CRC sent matches internal CRC computation, '0' if not
32 bits	W	**CRC** 32-bit CRC computed on all of the data bits of the burst
1 bit	R	**Status** *(legacy mode only)* '1' if the most recently sent data word was written to the SPR bus, '0' if the bus was not ready. Sent *m* times, once after each data word.
n bits	W	**Data** 32-bit data word. Sent *m* times.
1 bit	W	**Start Bit** Set to '1' to indicate the start of a burst write.

Table 11: CPU module burst write format

3.3.3 Burst Read

The next time the ADI is accessed after sending a valid burst read command, the CPU module will be in burst read mode. In this mode, commands and data are not interpreted or executed on transition through UPDATE_DR. Instead, counters are used to determine position in the bitstream, and a word is read from the CPU SPR bus while the previous data word is transferred out via JTAG. The total length of a burst read

transfer depends on the count field in the burst setup command; for a transfer of *m* words, the total length will be (32 * *m*) + 33, or (33 * *m*) + 32 in legacy mode.

The first bit (or bits) transferred during a burst read is a status bit. This bit indicates whether or not data from the SPR bus is ready to be transferred out via JTAG. The CPU module will send '0' bits until a word is ready, then send a single '1' bit. A data word will follow a '1' status bit. This first status bit is required in either speed mode – due to the time domain synchronization, a data word will not be ready when the first status bit is sent.

In legacy mode, a status bit (or bits) is transferred before each data word. Data transmission continues to alternate status bits and data words until all words have been transferred. In hi-speed mode, a status bit is sent only before the first data word. After that, each data word immediately follows the previous data word until all data has been sent.

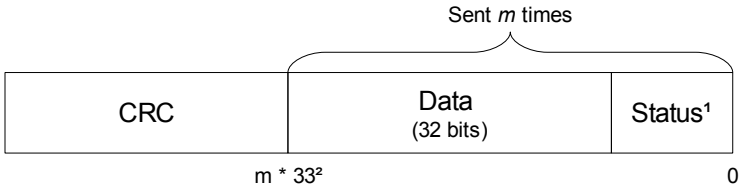
Immediately following the last data word, a 32-bit CRC code is sent from the CPU module to the driver software. The driver software should compare the CRC received with one computed internally, to determine whether or not the complete transaction must be retried.

The CRC protects only the data bits in a burst transaction; commands and status bits are not included in the CRC computation. The CRC resets before each burst transaction. For more information on the CRC calculation, see chapter 6.

The OR1200 SPR bus does not provide any error indication for failed transactions beyond ready / not ready. Thus, there is no error register to check after a burst is completed (as opposed to the WishBone module).

Note that extra bits sent at the end of a burst read are ignored; thus, the user need not worry about sending a valid or safe operation/opcode at the end of the burst transaction.

Figure 14: Burst Read format



[1] *Only sent once in hi-speed mode*

[2] *32 in hi-speed mode*

Bits	Access	Description
32 bits	R	**CRC** 32-bit CRC computed on all of the data bits of the burst
n bits	R	**Data** Data word. Length specified by the opcode in the burst setup command. Sent *m* times.
1 bit	R	**Status** Read '0' until a word is ready to be sent, then a single '1' bit is sent before the data word. In hi-speed mode, only sent before the first data word. In legacy mode, sent before each data word.

Table 12: CPU module burst read format

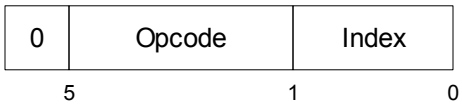
3.3.4 Register Select

A register select command will make the module-internal register with the given index active (in the currently selected sub-module). While any register in the current module can be written with a single command, only the active register can be read.

When the TAP enters CAPTURE_DR mode, the CPU module captures the value of the active register into the output shift register, allowing the value to be read when the TAP is in SHIFT_DR mode. This will happen each time a command is sent to the CPU module, unless the previous command was a burst setup.

The register select command uses the same top-level select bit and opcode format as the other module commands. In this case, the opcode is followed by a 1-bit value, which is the index of the register which should be made active. See the API section on register for a complete listing of all registers in the CPU module, and the meaning of each.

Figure 15: Register Select command format



Bit #	Access	Description
5	W	**Top-Level Select** Set to '0' for all sub-module commands
4:1	W	**Opcode** Operation to perform. 0xD = Internal Register Select
0	W	**Index** Index of the register to make active. The CPU module uses a 1-bit index.

Table 13: CPU module Register Select command format

3.3.5 Register Read

There is no specific command to read a module internal register; the value of the active register is shifted out every time a command is shifted in. In order to read a particular register, make that register active using the register select command, then read out the value of the register while sending a NOP command. The length of each register may vary, and the meaning of each bit is also register-specific. The register data will be LSB-aligned; that is, the LSB of the register data will be the first bit shifted out of the ADI. This allows the minimum number of bits to be transferred over the JTAG, and allows the user to ignore the total length of the output shift register. It is legal to read more bits than the active register has – the additional bits will have undefined values, and should be discarded. Note that only register data will be transferred out; no command, index, opcodes, start, or status bits will be sent with it. It is legal to abort a register read at any time, provided that a valid command (probably a NOP) is in the correct position in the input shift register.

3.3.6 Register Write

A register write command contains both a register index and data to be written to the register at that index. The register with the given index will become the active register after this command is executed. Note that the value of the *previously* active register will be shifted out as this command is shifted in. The length of the data field is variable, and depends on the particular register being written. See the API section on registers for a complete description of the registers, their lengths, and their meanings.

Figure 16: Register Write command format

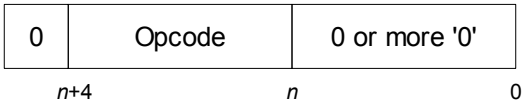
Bit #	Access	Description
5 + *n*	W	**Top-Level Select** Set to '0' for all sub-module commands
(4:1) + *n*	W	**Opcode** Operation to perform. 0x9 = Internal Register Write
n	W	**Index** Index of the register to make active. The CPU module uses a 1-bit index.
n-1:0	W	**Data** *n* bits of data to write to the register specified by Index. *n* depends on the register being written.

Table 14: CPU module Register Write command format

3.3.7 NOP

A NOP command will perform no operation. It is included as a "safe" command to shift into the CPU module while shifting out internal register data. A NOP command consists of five or more zeros, making it easy to send for any length of data read.

Figure 17: NOP command format



Bit #	Access	Description
4 + *n*	W	**Top-Level Select** Set to '0' for all sub-module commands
(3:0) + *n*	W	**Opcode** Operation to perform. 0x0 = NOP
n:0	W	**Zero** Zero or more '0' bits

Table 15: CPU module NOP command format

3.4 JSP Commands

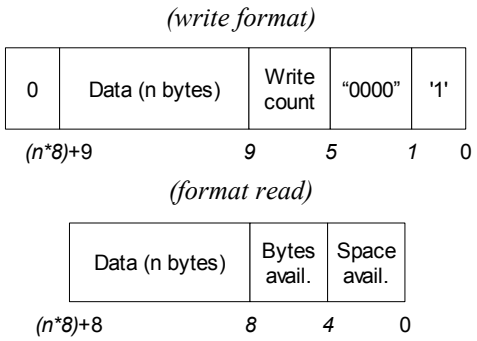
The JSP module does not use the same command set as the WishBone and OR1200 modules. Instead, it has a single command, “Transact”.

3.4.1 Transact

During the Transact command, byte counts are exchanged, then data is simultaneously read and written. When compatibility with multi-device JTAG chains is enabled (a synthesis-time option), the communication is asymmetric: the data written must have a '1' bit prefixed, whereas the data read has no such start bit. This means that the input and output data will be offset by 1 bit. The first 9 bits sent will consist of the start bit and all 8 bits of the “counts” byte, while the first 9 bits received will consist of all 8 bits of the counts byte, followed by the first bit of the first outgoing data byte. When multi-device chain compatibility is disabled, the start bit is not used for the outgoing data, and the in/out transactions are symmetric.

Up to 8 bytes may be read and written for each Transact operation, as this is the maximum capacity of the read and write FIFOs. Fewer data bytes may be transferred as well – zero data bytes is a legal transaction, provided there are no bytes in the FIFO available for reading (if there are, then at least one byte will be lost if all available bytes are not read).

Figure 18: JSP TRANSACT command format



Bit #	Access	Description
(n*8) +9	W	**Zero** One bit with value '0'. Used to insure top-level module does not interpret the transaction as a module select command. Unnecessary if the most-significant bit of data byte n is '0'.
(n*8) +9:9	W	**Data** n bytes of data. The first "write count" bytes will be written to the FIFO, if there is space. Subsequent bytes will be ignored.
9:6	W	**Write count** Number of valid data bytes which will be sent following the write count. (Note that an unlimited number of invalid data bytes may follow the valid data.) Legal values are 0-8 inclusive.
5:1	W	**Zero** Four bits with value '0'
0	W	**Start Bit** Must have value '1'. Used only when ADI is synthesized with JSP compatibility for multi-device JTAG chains.

Table 16: JSP Transact command write format

Bit #	Access	Description
(n*8) +8:8	R	**Data** n bytes of data. The first "bytes available" bytes will be valid data from the output FIFO. Subsequent bytes will have undefined values, and should be discarded.
7:4	R	**Bytes Available** Number of bytes which may be read from the output FIFO. All available bytes should be read on every transaction. May have values from 0-8 inclusive.
3:0	R	**Space Available** Bytes free in the input FIFO. This is the maximum number of bytes which will be written to the FIFO. May have values of 0-8 inclusive.

Table 17: JSP Transact command read format

3.5 WishBone Module Registers

Table 18 summarizes all of the registers contained within the WishBone sub-module. Note that the data format of a register may be different depending on whether it is read or written; this saves the user from having to shift in extra bits to fill read-only values when writing.

Index	Register name
0x0	Error register

Table 18: WishBone module register summary

3.5.1 Error Register

The error register captures WishBone bus errors during burst transactions. Each time a bus access is completed, the WishBone error indicator bit (wb_err) is tested. If an error is present, then the error bit in the error register is set to '1', and the address of the failed access is stored in the rest of the error register. Once the error bit is set, the error register will retain its value and further WishBone errors will be ignored until the error

bit is reset. The error bit may only be reset by writing a '1' to the error bit via an internal register write.

When read, the error bit is the first bit shifted out. This allows transferring the minimum number of bits when testing whether an error has occurred (5 bits must be transferred in order to send a valid NOP command while reading). If an error has occurred, then an error handling routine in the driver software can read the error register again to get the 32-bit error address – the value will not change until the error bit is reset.

When written, the error register consists of a single bit, the error bit. This should be written as '1' in order to clear the error bit and re-enable error detection.

Figure 19: Wishbone module Error Register, as read

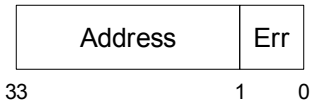
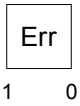


Figure 20: Wishbone module Error Register, as written



Bit #	Access	Description
33:1	R	**Address** When error bit = '1', contains the address of the failed transaction
0	R	**Err** *(when read)* Error bit. Set to '1' when a WishBone error has occurred since the last time the error bit was reset.
0	W	**Err** *(when written)* Write as '1' to reset the error bit to '0' and re-enable error detection

Table 19: WishBone module Error Register format

3.6 CPU Module Registers

Table 20 summarizes all of the registers present within the CPU sub-module.

Index	Register name
0x0	Status register

Table 20: CPU module register summary

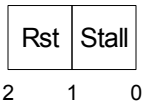
3.6.1 Status Register

The status register is used to control the reset line to the CPU, and to detect and control breakpoint conditions. The format is the same whether read or written.

Bit 1 of the register controls the reset line to the CPU. When written as a '1', reset to the CPU is active, and the CPU is put into a reset state. When written '0', the reset line is negated, and the CPU may run normally. The reset bit can only be changed by an internal register write in the CPU module.

Bit 0 of the status register detects breakpoints, and controls the stall line to the CPU. When written '1' via internal register write, the stall line to the CPU is made active, and the CPU stops executing instructions (but retains the ability to resume). When written '0', the stall line is negated, and the CPU resumes execution. The stall bit will also be set when the breakpoint output of the CPU goes active. The breakpoint output will be registered, the stall bit will be set, and the CPU will be held in the stall state by the ADI. This condition must be detected by polling in the software driver – once stalled due to a breakpoint, the CPU cannot resume execution until the stall bit is reset to '0' by an internal register access via JTAG.

Figure 21: CPU module Status register



Bit #	Access	Description
1	R/W	**Reset** Set to '1' to put the CPU in reset state. Set to '0' to allow the CPU to restart execution.
0	R/W	**Stall** Set to '1' to suspend execution in the CPU. Set to '0' to resume. Will be set to '1' automatically when a breakpoint indicator arrives from the CPU.

Table 21: CPU module Status Register format

3.7 JSP Module Registers

Table 22 summarizes all of the registers present within the JSP sub-module. Note that these are registers seen from the WishBone target interface; the JSP sub-module has no internal registers accessible from the JTAG interface. All JSP WishBone registers are 8 bits wide. For a more complete description of how these registers should work, see the 16550 UART specification.

Index	Access	Register name
0x0	R	**RDR**
0x0	W	**THR**
0x1	R/W	**IER**
0x2	R	**IIR**
0x2	W	**FCR**
0x3	R/W	**LCR**
0x4	R	**MCR**
0x5	R/W	**LSR**
0x6	R	**MSR**
0x7	R	**SCR**

Table 22: JSP WishBone register summary

3.7.1 Receive Data Register (RDR)

Reads from this register retrieve bytes from the ADI->WB data FIFO, when bytes are available. Each reads pulls one byte from the FIFO. When no bytes are available in the FIFO, undefined data is read.

Bit #	Access	Description
7:0	R	**Data** 1 byte of data from the receive FIFO

Table 23: JSP module RDR register format

3.7.2 Transmit Holding Register (THR)

Data written to this register is put into the WB->ADI FIFO, when space is available. When no space is available, the write is ignored. A write to this register will also arm the transmit interrupt, when this interrupt has been enabled in the **IER**.

Bit #	Access	Description
7:0	W	**Data** 1 byte of data for the transmit FIFO

Table 24: JSP module THR register format

3.7.3 Interrupt Enable Register (IER)

This register is used to enable interrupts (on the `int_o` port) when certain events occur. Currently, only the 'receive data available' and 'transmit register empty' interrupts are implemented. When the 'receive data available' interrupt is enabled, the interrupt output will be active any time one or more bytes is available in the receive FIFO. When the 'transmit register empty' interrupt is enabled, the interrupt output will become active when the byte count in the transmit FIFO goes from 1 to 0. The 'transmit' interrupt can be cleared by reading the **IIR**, or by writing the 8th byte to the **THR** register.

Bit #	Access	Description
0	R/W	**ERBFI** Set to '1' to enable receive data interrupt
1	R/W	**ETBEI** Set to '0' to enable transmit interrupt
3:2	R/W	**ELSI, EDSSI** Unimplemented. May be read/written, but values ignored.
7:4	R	Always read as '0'

Table 25: JSP module IER register format

3.7.4 Interrupt Identification Register (IIR)

This register is used to determine the cause of an interrupt event. Only the highest priority active interrupt will be indicated at any given time; lower-priority interrupts will be indicated only after the higher-priority interrupt has been cleared. Note that in this implementation, the **IIR** does not take the **IER** into account. Thus, even if the receive interrupt is not enabled in the **IER**, a receive interrupt may still be shown as active in the **IIR**.

Bit #	Access	Description
0	R	**IP** '0' if any interrupt is pending, '1' otherwise.
2:1	R	**Interrupt ID** 10 Receive Data Available (highest priority) 01 Transmit Holding Register Empty Note that other values are specified in the 16550 documentation, but are not used by the JSP.
7:3	R	Always read as '0'

Table 26: JSP module IIR register format

3.7.5 FIFO Control Register (FCR)

This register is used to control the FIFOs. Since the JSP WB interface does not have 16550-compatible FIFOs, only two bits of this register are implemented – the two bits which clear and reset the FIFOs. These reset bits are not persistent; they do not need to be written back to '0' after a reset before the device can be used.

Bit #	Access	Description
0	W	Unimplemented
1	W	**RCVR FIFO RESET** Write a '1' to reset the receive FIFO
2	W	**XMIT FIFO RESET** Write a '1' to reset the transmit FIFO
7:3	W	Unimplemented

Table 27: JSP module FCR register format

3.7.6 Line Control Register (LCR)

The **LCR** controls options such as number of data bits and parity for a standard UART; it is therefore mostly unused in the JSP. Only the **DLAB** bit is implemented as a read/write register. This is to prevent standard 16550 software drivers from sending or receiving incorrect data to or from the FIFOs while trying to set a baud rate divisor. Other bits are read-only, and reflect ordinary UART settings.

Bit #	Access	Description
1:0	R	**WLS** Word Length Select, always read as “11” (8 bits)
2	R	**STB** Number of stop bits, read as '0' (1 bit)
3	R	**PEN** Parity Enable, read as '0' (disabled)
4	R	**EPS** Even Parity, read as '0' (disabled)
5	R	**Stick Parity** Sticky parity bit, read as '0'
6	R	**Set Break** Read as '0'
7	R/W	**DLAB** Divisor Latch Access Bit. When '0', reads and writes to address offset 0x0 access the read and write FIFOs. When '1', access is disabled.

Table 28: JSP module LCR register format

3.7.7 Modem Control Register (MCR)

In an actual UART, the bits in this register are used to set and clear various modem control lines (RTS, DTR). Since the JSP has none of these, this register is unimplemented. Writes will be ignored, and reads will always return 0x00.

Bit #	Access	Description
7:0	R	**Unimplemented** Always read as 0x00

Table 29: JSP module MCR register format

3.7.8 Line Status Register (LSR)

The **LSR** is used to detect conditions in the UART that require attention. Since the JSP does not implement breaks or error detection, only three bits are implemented (and two have the same value).

Bit #	Access	Description
0	R	**Data Ready** Read as '1' when there is at least 1 byte in the receive FIFO
4:1	R	**Unimplemented** Always read as '0'
5	R	**THRE** Transmit Holding Register Empty. Read as '1' when the number of bytes in the transmit FIFO is 0.
6	R	**TEMT** Transmitter Empty. Read as '1' when the number of bytes in the transmit FIFO is 0.
7	R	**Unimplemented** Always read as '0'

Table 30: JSP module LSR register format

3.7.9 Modem Status Register (MSR)

The **MSR** is used to detect the states of incoming modem control lines in a standard UART. Since the JSP does not implement these lines, the **MSR** is read-only, with static values.

Bit #	Access	Description
0	R	**DCTS** Delta Clear To Send. Read as '0'.
1	R	**DDSR** Delta Data Set Ready. Read as '0'.
2	R	**TERI** Trailing Edge Ring Indicator. Read as '0'.
3	R	**DDCD** Delta Data Carrier Detect. Read as '0'.
4	R	**CTS** Clear To Send. Read as '1'.
5	R	**DSR** Data Set Ready. Read as '1'.
6	R	**RI** Ring Indicator. Read as '0'.
7	R	**DCD** Data Carrier Detect. Read as '1'.

Table 31: JSP module MSR register format

3.7.10 Scratch Register (SCR)

The scratch register has no effect on the workings of a standard UART. To conserve logic, it is unimplemented in the JSP. Writes to the **SCR** are ignored, reads return a static 0x00.

Bit #	Access	Description
7:0	R	**Scratch** Unimplemented. Always read as 0x00.

Table 32: JSP module SCR register format

4

IO Ports

4.1 TAP Ports

The Advanced Debug Interface connects to the TAP controller with the signals shown in Table 33.

Port	Width	Direction	Description
tck_i	1	input	Test clock input
tdi_i	1	input	Test data input
tdo_o	1	output	Test data output
shift_dr_i	1	input	TAP controller state “Shift DR”
pause_dr_i	1	input	TAP controller state “Pause DR”
update_dr_i	1	input	TAP controller state “Update DR”
capture_dr_i	1	Input	TAP controller state “Capture DR”
rst_i	1	input	Reset signal.
debug_select_i	1	input	Instruction DEBUG is activated

Table 33: TAP Ports

4.2 CPU Ports

For each CPU module included, one set of I/O lines for that module will be present. The '*n*' in the port name will be replaced with the CPU module number, always starting from 0. These lines are shown in Table 34.

Port	Width	Direction	Description
cpu*n*_clk_i	1	input	CPU clock signal.
cpu*n*_addr_o	32	output	CPU address
cpu*n*_data_i	32	input	CPU data input (data from CPU)
cpu*n*_data_o	32	output	CPU data output (data to CPU)
cpu*n*_bp_i	1	input	CPU breakpoint
cpu*n*_stall_o	1	output	CPU stall (selected CPU is stalled)
cpu*n*_stb_o	1	output	CPU strobe
cpu*n*_we_o	1	output	CPU write enable signal indicates a write cycle when asserted high (read cycle when low).
cpu*n*_ack_i	1	input	CPU acknowledge (signals end of cycle)
cpu*n*_rst_o	1	output	CPU reset output (resets CPU)

Table 34: CPU Ports

4.3 WISHBONE Ports

The WishBone module, if included, will add a set of WishBone initiator interface signals to the top-level IO, as described in Table 35.

Port	Width	Direction	Description
wb_clk_i	1	input	WISHBONE clock
wb_rst_i	1	Input	WISHBONE reset signal

Port	Width	Direction	Description
wb_ack_i	1	input	WISHBONE acknowledge indicates a normal cycle termination
wb_adr_o	32	output	WISHBONE address output
wb_cyc_o	1	output	WISHBONE cycle encapsulates a valid transfer cycle.
wb_dat_i	32	input	WISHBONE data input (data from WISHBONE)
wb_dat_o	32	output	WISHBONE data output (data to WISHBONE)
wb_err_i	1	input	WISHBONE error acknowledge indicates an abnormal cycle termination
wb_sel_o	4	output	WISHBONE select indicates which bytes are valid on the data bus.
wb_stb_o	1	output	WISHBONE strobe indicates a valid transfer.
wb_we_o	1	output	WISHBONE write enable indicates a write cycle when asserted high (read cycle when low).
wb_cab_o	1	output	WISHBONE consecutive address burst indicates a burst cycle. (always false)
wb_cti_o	3	output	WISHBONE cycle type identifier indicates type of cycle (single, burst, end of burst) (always single)
wb_bte_o	2	output	WISHBONE burst type extension (always 0)

Table 35: WISHBONE Ports

4.4 JSP Ports

The JSP module, if included, will add a set of WishBone target interface signals to the top-level IO, as described in Table 36.

Port	Width	Direction	Description
wb_clk_i	1	input	WISHBONE clock (same as WB module)

Port	Width	Direction	Description
wb_rst_i	1	Input	WISHBONE reset (same as WB module)
wb_jsp_ack_o	1	output	WISHBONE acknowledge indicates a normal cycle termination
wb_jsp_adr_i	32	input	WISHBONE address input
wb_jsp_cyc_i	1	input	WISHBONE cycle encapsulates a valid transfer cycle.
wb_jsp_dat_i	32	input	WISHBONE data input (data from WISHBONE)
wb_jsp_dat_o	32	output	WISHBONE data output (data to WISHBONE)
wb_jsp_err_o	1	output	WISHBONE error acknowledge indicates an abnormal cycle termination
wb_jsp_sel_i	4	input	WISHBONE select indicates which bytes are valid on the data bus.
wb_jsp_stb_i	1	input	WISHBONE strobe indicates a valid transfer.
wb_jsp_we_i	1	input	WISHBONE write enable indicates a write cycle when asserted high (read cycle when low).
wb_jsp_cab_i	1	input	WISHBONE consecutive address burst indicates a burst cycle. (always false)
wb_jsp_cti_i	3	input	WISHBONE cycle type identifier indicates type of cycle (single, burst, end of burst) (always single)
wb_jsp_bte_i	2	input	WISHBONE burst type extension (always 0)
int_o	1	output	Interrupt indicator (active high)

Table 36: JSP Ports

5

Module Configuration

The Advanced Debug Interface supports three options, which allow the user to configure which sub-modules will be included when the design is synthesized.

Option: `DBG_WISHBONE_SUPPORTED`

Use: Define this option if you want to include a WishBone module in the ADI. Default is defined.

Option: `DBG_CPU0_SUPPORTED`

Use: Define this option if you want to include one or more OR1200 CPU debug modules. Default is defined.

Option: `DBG_CPU1_SUPPORTED`

Use: Define this option if you want to include a second OR1200 CPU debug module. Default is undefined.

Option: `ADBG_USE_HISPEED`

Use: Define this option to synthesize the WishBone and OR1200 modules in hi-speed mode. This mode allows an order-of-magnitude improvement in data transfer speed when using USB JTAG cables, at the cost of some error detection. Default is defined.

Option: `DBG_JSP_SUPPORTED`

Use: Define this option if you want to include the JTAG Serial Port (JSP). Default is defined.

Option: `ADBG_JSP_SUPPORT_MULTI`

Use: Define this option when you are using the JSP module, and your system has more than one device on the JTAG chain. This will cause a protocol change that will allow the JSP to be used on systems with multi-device chains.

One other WishBone-specific option can be found in `adbg_wb_defines.v`:

Option: `DBG_WB_LITTLE_ENDIAN`

Use: Define this option when the system CPU uses big-endian byte ordering. When left undefined, little-endian byte ordering (the OR1200 default) will be used. Default is undefined.

6

CRC Module

A CRC calculation module is contained within each WishBone and CPU sub-module. The CRC module is active only during burst read and write transactions. It calculates a 32-bit CRC on only the data bits of the transaction, starting with the LSB of the first word transferred, and ending with the MSB of the last word transferred.

In order to simplify the hardware, it was desirable to have the shift performed during the CRC calculation be in the same direction as the shift required to read out the CRC serially, LSB-first. This is the reverse of previous CRC implementations. As such, the CRC polynomial used is also the bitwise-reverse of the standard Ethernet CRC-32 polynomial: `0xEDB88320`. A C-language routine for computing a compatible CRC, up to 32 bits at a time, is reproduced below. When computing a CRC for multiple words, the output of the last call to `compute_crc()` should be passed to the next call as `crc_in`. When calling `compute_crc()` for the first word of a burst, `crc_in` should be set to `0xFFFFFFFF`.

```
#define CRC_POLY 0xEDB88320


uint32_t compute_crc(uint32_t crc_in, uint32_t data_in,
                        int length_bits)
{
uint32_t crc_out, c, d;
int i;

      crc_out = crc_in;
      for(i = 0; i < length_bits; i++) {
            d = ((data_in >> i) & 0x1) ? 0xffffffff : 0x0;
            c = (crc_out & 0x1) ? 0xffffffff : 0x0;
            crc_out = crc_out >> 1;
            crc_out = crc_out ^ ((d ^ c) & CRC_POLY);
      }

return crc_out;
}
```