Advanced Watchpoint Control

Nathan Yawn

nyawn@opencores.org

August 20, 2010

History

Rev	Date	Author	Comments
1.0	20/8/2010	Nathan Yawn	Initial version

Contents

1. Overview

AdvancedWatchpointControl (AWC) is a Java program designed to allow a user to configure and monitor the hardware watchpoint functionality of the OpenRISC 1000 processor. The program provides a graphical interface which allows a user to individually control each hardware watchpoint function. At present, there is a 1-to-1 mapping of UI elements to hardware functions – a user must manually set up the breakpoint hardware, there is no simple way to, for example, "break when x has value 42".

AdvancedWatchpointControl is designed to be used with the Advanced Debug System. Specifically, AWC is written to interface to the adv_jtag_bridge program's HWP server. This allows AWC to be used in parallel with the standard GDB debugger. Note that because the network protocol used is a slightly modified version of RSP, it may be possible to connect AWC to a standard RSP server. However, run/stop detection will not work, and a server other than adv_jtag_bridge may explicitly disable hardware breakpoints, regardless of AWC's commands.

AdvancedWatchpointControl is intended to be used in parallel with GDB. During debugging, AWC will be connected to adv_jtag_bridge's HWP server at the same time GDB is connected to adv_jtag_bridge's RSP server. GDB is still used to start and stop the processor, and examine registers and memory. AdvancedWatchpointControl is used only to modify and examine the OR1000's hardware watchpoint registers.

In order to use hardware watchpoints, they must be implemented and enabled in the OR1000 hardware. AdvancedWatchpointControl is designed to work with OR1000 watchpoint hardware as specified in the OR1000 architecture manual. As of this writing, the OR1200v1 implementation's debug unit is not implemented to the spec, and the existing implementation is badly broken. The OR1200v3 has not yet corrected the problem. It is therefore recommended that the user implement the OR1200v1, and add the debug unit patch distributed in the `Patches/` directory of the Advanced Debug System.

2. User Interface

2.1. Overview

The AdvancedWatchpointControl GUI is separated into four logical groups: The Server Connection group, the Watchpoint group, the Counters group, and the Control group. A screenshot of the complete GUI is shown in Figure 1. A description of each UI group follows.
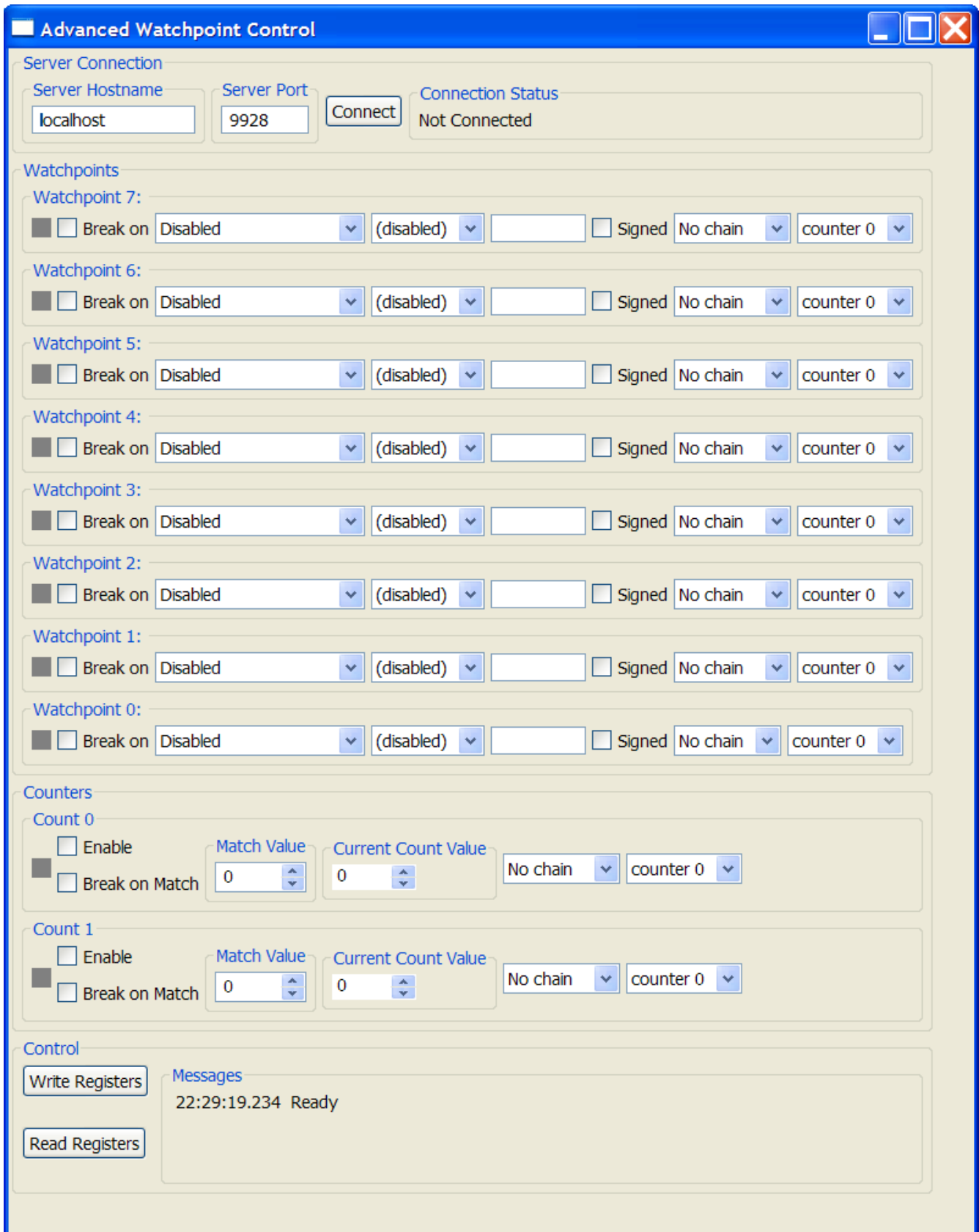
Advanced Watchpoint Control

Server Connection

Server Hostname
localhost

Server Port
9928

Connect

Connection Status
Not Connected

Watchpoints

Watchpoint 7:
Break on | Disabled | (disabled) | Signed | No chain | counter 0

Watchpoint 6:
Break on | Disabled | (disabled) | Signed | No chain | counter 0

Watchpoint 5:
Break on | Disabled | (disabled) | Signed | No chain | counter 0

Watchpoint 4:
Break on | Disabled | (disabled) | Signed | No chain | counter 0

Watchpoint 3:
Break on | Disabled | (disabled) | Signed | No chain | counter 0

Watchpoint 2:
Break on | Disabled | (disabled) | Signed | No chain | counter 0

Watchpoint 1:
Break on | Disabled | (disabled) | Signed | No chain | counter 0

Watchpoint 0:
Break on | Disabled | (disabled) | Signed | No chain | counter 0

Counters

Count 0
Enable
Break on Match
Match Value
0
Current Count Value
0
No chain | counter 0

Count 1
Enable
Break on Match
Match Value
0
Current Count Value
0
No chain | counter 0

Control

Write Registers

Read Registers

Messages
22:29:19.234 Ready

Figure 1: Advanced Watchpoint Control User Interface

2.2. Server Connection Group

The Server Connection Group allows the user to specify the location of the HWP server that AdvancedWatchpointControl will connect to. There are four UI elements, as shown in Figure 2: **Server Hostname**, **Server Port**, the **Connect** button, and the **Connection Status** text field.
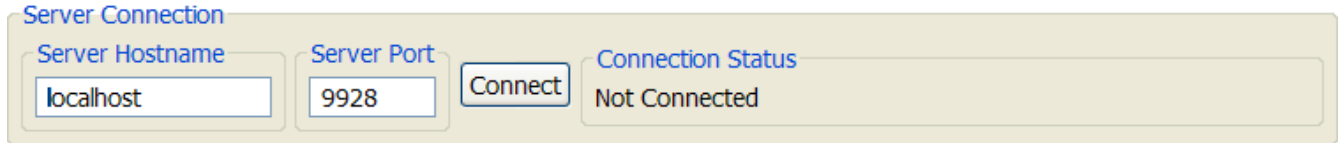

Figure 2: Server Connection Group

The **Server Hostname** field allows the user to specify the hostname or IP address of the computer where the adv_jtag_bridge HWP server is running. It is easiest to run AWC and adv_jtag_bridge on the same computer, thus "localhost" is the default value for this field. However, the AWC can connect to an HWP server on any machine to which it has a network connection.

The **Server Port** field allows the user to specify the IP port number of the HWP server. By default, adv_jtag_bridge starts the HWP server on port 9928, which is also the default for this field. If the port number is changed on the adv_jtag_bridge command line, then the same port number must be entered into this field.

The **Connect** button will cause AWC to attempt to connect to an HWP server at the specified host and port. On connection, the run/stop status of the processor will be checked. If the processor is stopped, the status of all watchpoint hardware will be read from the target CPU and displayed.

The **Connection Status** is a read-only text field. It is intended to inform the user of the network connection status. It may have values Not Connected, Connected, or Error Connecting.

2.3. Watchpoints Group

The Watchpoints Group will contain from zero to eight individual watchpoint subgroups. Each watchpoint subgroup is designed to control a single watchpoint. When AWC is first connected to an HWP server, it checks to see how many watchpoints are implemented in the OR1000 hardware. AWC will display only as many watchpoint subgroups as there are watchpoints implemented in the hardware.

A single watchpoint subgroup is shown in Figure 3. All watchpoint subgroups are identical, except for the watchpoint number, and the chain function of watchpoint 0 (discussed later). A watchpoint subgroup contains 8 elements.
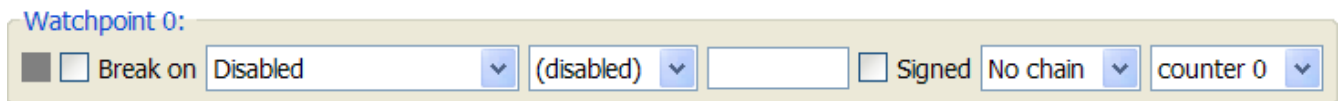

Figure 3: Watchpoint Subgroup

The **Watchpoint Caused Break** indicator shows whether a particular watchpoint caused the target CPU to break. If the watchpoint did not cause a break, the indicator will remain grey. If the watchpoint did cause a break, then the indicator will turn red. Note that after the target CPU stops, the target watchpoint registers must be read (see "Control Group," section 2.5) before the indicators will be valid. Also note that if you are using the OR1200 debug unit hardware patch, a write to the watchpoint

hardware will clear the break indicators. So, if one or more of the indicators are red and you write the hardware registers, then read them back, all indicators will revert to grey.

The **Break On** checkbox allows the user to tell the CPU to halt execution and turn control over to the debug interface if the match condition in that watchpoint occurs. If the box is checked, then the CPU will halt when the match occurs (assuming a match condition is configured). Note that this may not always be desirable – a match condition can still increment a hardware counter even if it does not cause a break.

The **Match Source** drop-down menu allows the user to select which data will be used as the comparison source. This may be Instruction Fetch Address, Data Load Address, Data Store Address, Data Load or Store Address, Load Data Value, Store Data Value, or Load or Store Data Value, as described in the OR1000 architecture manual. If Disabled is selected, then that watchpoint will not trigger a match or cause a break.

The **Comparison Type** drop-down menu allows the user to choose what sort of comparison will be performed. This may be equal (==), not equal (!=), greater than (>), greater than or equal to (>=), less than (<), or less than or equal to (<=), as described in the OR1000 architecture manual. If (disabled) is selected, that watchpoint will not trigger a match or cause a break.

The **Value** field is a text entry field into which the user enters the data value against which the **Match Source** will be compared. This is a 32-bit value, signed or unsigned (depending on the **Signed** check box), and may be entered in decimal (including negatives) or hexadecimal.

The **Signed** check box indicates whether a signed or unsigned comparison will be used. When checked, the comparison will be signed. This affects only greater than / less than comparisons. In a signed comparison, -1 (0xFFFFFFFF) is less than 0 (0x00000000). If in unsigned comparison is used, then 0xFFFFFFFF will be greater than 0x00000000.

The **Chain** drop-down menu allows watchpoints to be chained together for added functionality. For example, to detect when a particular variable is given a value of 42, a user would chain together two watchpoints; one to match the store address of the variable, and the other to match the store data of 42. Chain type may be (none), logical AND, or logical OR. Each watchpoint may only be chained to a single adjacent watchpoint, as specified in the OR1000 architecture manual. There is no limit on the number of watchpoints which may be chained. Note that chaining watchpoint 0 is a special case – this watchpoint can only be chained to the "external watchpoint" input of the OR1000. This requires special hardware to be connected to the external watchpoint input of the processor hardware. If this functionality is not implemented, then the watchpoint 0 chain function should not be used.

The **Counter Assignment** drop-down menu allows each watchpoint to be assigned to either counter 0 or counter 1. Every watchpoint must be assigned to one or the other. Each time a watchpoint generates a match, the associated counter will be incremented, if that counter is enabled.

2.4. Counters Group

The counter group contains two counter subgroups. Each counter subgroup controls one of the two hardware watchpoint counters, as specified in the OR1000 architecture. The two subgroups are identical, with the exception of the counter number and the chaining options (discussed below). One counter subgroup is shown in Figure 4. Each counter subgroup has seven elements.
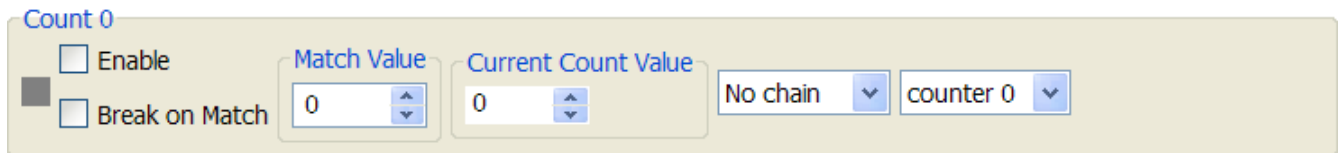
Figure 4: Counter Subgroup

The **Counter Caused Break** indicator shows whether a particular counter caused the target CPU to break. These function exactly as the **Watchpoint Caused Break** indicators do, including the caveats about the indicators being cleared on a write to the target CPU's watchpoint registers.

The **Enable** check box allows a user to enable or disable the counter. When the box is checked, then a match on any watchpoint assigned to that counter will cause the counter to increment, regardless of whether or not the watchpoint caused a break. Note that a counter will only be incremented once per CPU instruction executed, even if that instruction caused multiple watchpoint matches. When the enable box is not checked, the counter will not increment. Note that a counter can still cause a break even if not enabled, if the match value equals the count value and the **Break on Match** box is checked.

The **Break on Match** check box configures whether a counter match will cause the target CPU to break and turn control over to the debug interface. When the box is checked, then a match between the count value and the match value will cause a break. When not checked, the counter will not cause a break on a match. Note that the count value is not reset to 0 by a match or a break, it must be reset manually.

The **Match value** spinner allows the user to enter the value which will be compared to the running counter hardware. This is an unsigned 16-bit value, which may be entered as a decimal number or set using the spinner's arrows.

The **Current Count Value** spinner has two purposes. On a read of the CPU's watchpoint registers, this spinner will be set to the value read from the hardware counter. This value may also be changed by the user, then written back to the hardware. This can be used to reset the count to zero. This is a 16-bit unsigned value, which may be entered as a decimal number or set with the spinner arrows.

The **Chain** drop-down menu allows the user to set the chain options for the counters. Counters may be chained in the same way as watchpoints. Counter 0 may only be chained to watchpoint 3, and counter 1 may only be chained to watchpoint 7.

The **Counter Assignment** drop-down menu allows each counter to be assigned to either counter 0 or counter 1. Counters must be assigned to counter 0 or counter 1, just as watchpoints must. Note that this means a match on a counter may be configured to increment that same counter.

2.5. Control Group

The control group contains two buttons which cause the program to interact with the target CPU hardware, and a messages area. The control group is shown in Figure 5.

The **Write Registers** button is used to set the values of the hardware watchpoint registers on the target CPU. When the button is pressed, the values of all UI elements in the Watchpoints and Counters Groups are read, and binary values for the OR1000 debug unit's watchpoint registers are generated. All of these values are then written to the target CPU. Note that this action will reset all "Watchpoints Breakpoint Status" bits in register DMR2 on the target CPU; the next time the registers are read, these bits will return false, and the **Watchpoint Caused Break** and **Counter Caused Break** indicators in the AWC UI will turn grey.
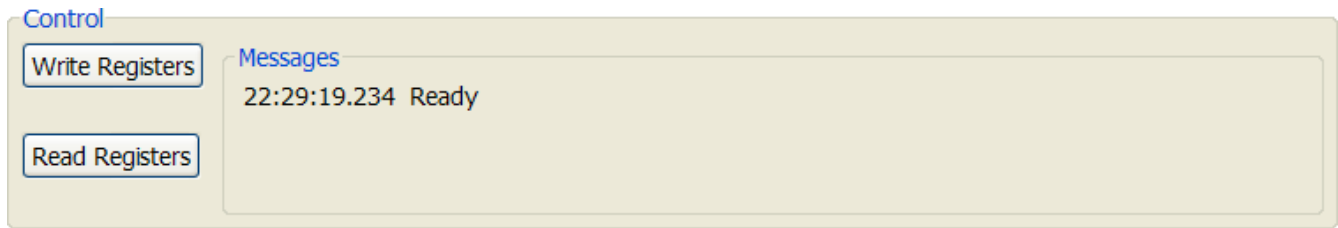
Figure 5: Control Group

The **Read Registers** button is used to get the values of the hardware watchpoint registers from the target CPU. When this button is pressed, all watchpoint and counter hardware registers are read. These values are then used to set all UI elements in the Watchpoints and Counters Groups.

Note that both the read and write operations will first check that the target CPU is stopped before proceeding. If the target CPU is running, then an error message will be displayed, and no further interaction with the target will occur.

The **Messages** text area is used to display status and error messages. Network, target, and other errors may be displayed here, along with network status. Messages are displayed with a time stamp.

3. Usage

3.1. Startup

On startup, AdvancedWatchpointControl will be disconnected from the server. All watchpoint subgroups will be visible, and all UI elements will be set to the processor's default values (no active breakpoints, all watchpoints and counters disabled, all values zero).

AWC must be connected to adv_jtag_bridge's HWP server before it can be used. For this connection to be made, the adv_jtag_bridge program must be running and connected to the target CPU. Additionally, GDB should be running and connected to adv_jtag_bridge. GDB should be used to stop the target CPU before AWC is connected.

Once the above requirements have been met, enter the hostname or IP address and the IP port number of the adv_jtag_bridge HWP server into AWC's Server Hostname group. If adv_jtag_bridge is running on the same machine as AWC, and the default port is used, then AWC's defaults will work. Once the correct address information is entered, press the **Connect** button. The **Connection Status** should change to Connected. If the **Connection Status** indicates an error, check the **Messages** area in the Control Group for more details about the error.

Once AWC is connected, it will check that the target processor is stopped. If it is, AWC will read the values of all watchpoint and counter hardware registers from the CPU, and update the AWC UI to reflect these values. AWC can now be used to set up watchpoints and counters.

3.2. Simple Example

With the target CPU stopped and AWC ready, we can set up a simple watchpoint to cause the CPU to break when it reaches a particular instruction. First, we must determine where we want the processor to break. This can be done by using GDB to find function addresses, examining code disassembly, looking at system.map files, or by other means. Once you have found the address, enter it as a hexadecimal number into the **Value** field for watchpoint 7. Then, check the **Break on** checkbox, select Instruction Fetch Addr from the **Match Source** dropdown, and select (==) from the **Comparison**

Type dropdown. The rest of the watchpoint 7 elements can be left as default – unsigned comparison, no chain, assign to counter 0.

Once you have the watchpoint configured, click on the ***Write Registers*** button in the Control Group. This will write your configuration to the CPU hardware.

Once your configuration is written, use GDB to start the target processor. The CPU should run until the specified instruction address is reached, then break. GDB will indicate when the CPU has stopped.

After the target CPU has stopped, click on ***Read Registers*** in AWC's Control group. The ***Watchpoint Caused Break*** indicator in the watchpoint 7 subgroup should turn red, indicating that watchpoint 7 caused the break. From here, you may reconfigure the watchpoint hardware to stop at a different address, or you may use GDB to restart the CPU – the processor will break again if it reaches the same instruction.

Note that when changing the values in the AWC UI, there is no "undo" feature. However, changes to the AWC UI are not committed to the CPU registers until the ***Write Registers*** button is clicked. So, if you wish to revert the UI elements to their previous values, simply click on the ***Read Registers*** button, and the UI elements will revert to the state stored in the CPU's registers.

3.3. Chaining Example

We can use chaining to trigger a break when a particular value is written to a particular variable. First, we must find the address in system memory where the variable is stored. If the variable is a global variable, then its address will be constant throughout the execution of the program on the target platform. In this case, the variable's address can be found in the system.map file (which can be created using the nm utility). For this example, we will assume this is the case.

We assume the steps in the Startup section (3.1) have been completed. To configure the watchpoint, first click the ***Break On*** checkbox to enable breakpoints for watchpoint 7. Next, select Store Addr in the ***Match Source*** dropdown, and (==) in the ***Comparison Type*** dropdown. Enter the variable's address in the ***Value*** field, leave the ***Signed*** checkbox un-checked, and leave the ***Counter Assignment*** at the default. In the ***Chain*** drop-down, select AND WP6.

In the watchpoint 6 subgroup, leave the ***Break On*** checkbox un-checked. Select Store Data Value in the ***Match Source*** dropdown, and (==) as the ***Comparison Type.*** Enter the data value you want to match in the ***Value*** field. Since the compare type is (==), the ***Signed*** checkbox has no effect, it can be left as default, along with ***Chain*** and ***Counter Select***.

Click on the ***Write Registers*** button to commit your configuration to the target processor, then use GDB to start the target. When the data value you entered as the ***Value*** in the watchpoint 6 subgroup is written to the variable at the address you entered as the ***Value*** of watchpoint 7, the target CPU will stop, and return control to GDB. When this happens, click on the ***Read Registers*** button in the Control Group. The ***Watchpoint Caused Break*** indicator in the watchpoint 7 subgroup should turn red. Note that the indicator in the watchpoint 6 group will remain grey.

Chaining may be used to further refine your search / stop parameters. For example, if you only wanted to stop when a variable was given a certain value by a certain function, you could use a disassembly to find the addresses of the first and last instruction of the desired function. You could then set up watchpoint 5 to trigger when the Instruction Fetch Address was greater than the first instruction, set up watchpoint 4 to trigger when the Instruction Fetch Address was less than the last instruction of the function, and use AND chaining from 6 to 5, and from 5 to 4.

Note that if the variable you wish to observe is local to a function, then it will be stored on the

stack, which means that its address may be different each time the function is called. In this case, you may use a disassembly (or binary debug information) to find the offset from the stack pointer where the variable is stored in that function. A breakpoint can then be set for the first instruction of the function proper (after the function prolog has adjusted the stack pointer), the CPU stack pointer register can be read using GDB, and the address of the target variable computed. The chained watchpoint can then be set up to look for writes of a particular value to that variable's address. It is suggested that a breakpoint be added at the end of the function, so that the variable watchpoint can be removed (preventing incorrect triggering by other writes in other functions).

3.4. Counters Example

We can use a hardware counter to break on a particular loop iteration. If a loop uses a local variable, it may be difficult to set a watchpoint to look for a particular value in the loop counter (see above). However, with hardware counters it is easy to track the number of times a particular instruction is executed.

Using a disassembly of your code, find the address of an instruction in the target loop where you'd like to stop after a set number of iterations. Enter this address into the *Value* field of watchpoint 7. Leave the *Break On* checkbox un-checked, select Instruction Fetch Addr from the *Match Source* dropdown, and (==) as the *Compare Type.* Leave the *Signed* box unchecked, disable chaining, and make sure watchpoint 7 is assigned to counter 0.

In the Counter 0 subgroup, check the *Enable* and *Break On Match* boxes. Set the *Match Value* to the iteration at which you would like to break (the first time through the loop will be iteration 1), and make sure the *Count Value* is set to 0. Disable chaining for counter 0, and assign it to counter 1 (so that it does not increment itself on a match). Once this is done, write the registers to the target CPU and use GDB to start the target. The CPU will break when it reaches the specified loop iteration. Read the registers back after the CPU stops to show the *Counter Caused Break* indicator in red. The *Count Value* will also be updated, and should equal the *Match Value*. Remember that the count value in the CPU hardware will not reset itself to 0 after a match, it must be reset manually.

4. Future Work

AdvancedWatchpointControl makes it possible to use the watchpoint hardware in an OR1000 CPU. Using counters and chaining, the processor can be stopped on almost any condition imaginable. However, it can be difficult to find the addresses needed to create useful breakpoints – the user must search disassembly and map files.

It is intended that future versions include a "Watchpoint Wizard" feature, which will be able to parse the debug information in ELF binary executables, and automatically find addressing information for the the user. Using the Wizard, a user could simply select a variable and a value, and the Wizard would set up the watchpoints necessary to break when the variable was written to that value.

It may also be possible to write a Wizard to watch a function-local variable; it could start by setting a breakpoint on entry to a function. When this break occurred, AWC could then configure the desired breakpoint and restart the CPU, without user interaction. However, this would likely require changes to adv_jtag_bridge, to coordinate possible negative interactions with GDB.

Appendix A: Code Structure

The code structure for AdvancedWatchpointControl is loosely based on the Model-View-Controller pattern. Here, the View consists of four GUI classes which create and manage all of the user interface elements. The class names all begin with "gui": guiControlGroup, guiContRegsGroup, guiDCRGroup, and guiServerGroup. The Controller is the class mainControl. The Model here is the set of classes used to read and write data to the target system CPU. A diagram of the main classes is shown in Figure 6.

The mainControl class is the core of the system. On instantiation, it creates an object to hold a cache of the register states from the target CPU, of class targetDebugRegisterSet. GUI objects get a reference to the cache from mainControl, but wrap it in a facade object (registerInterpreter) to allow them to individually get and set bitfields in the registers. GUI objects may also register with mainControl as Observers of the register cache; mainControl will notify each registered observer any time new register data has been read from the target CPU into the cache, and any time that data in the UI elements should be written into the cache in preparation for writing to the target CPU.

Two other types of observers may register with mainControl. LogMessageObservers will be notified each time a new log message should be displayed to the user. NetworkStatusObservers will be notified each time the network connection status changes. The actual log message or network status can then be pulled by the observer directly from mainControl.

The Model portion of the code sends and receives RSP packets via network. It consists of three main functional classes: targetTransactor, rspCoder, and networkSystem. The targetTransactor class is aware of the different types of RSP transactions which may occur. A targetTransactor method is called for a particular type of transaction (register read, register write, check if target CPU is running). This method creates an object specific to this type of transaction (ReadRegisterTransaction, WriteRegisterTransaction, or TargetRunningTransaction, respectively), which implements the TargetTransaction interface. This object formats the outgoing data packet upon construction. The targetTransactor class then passes the TargetTransaction object to the rspCoder.

The rspCoder retrieves the outgoing data packet from the TargetTransaction object, adds the checksum, escaping, and delimiters, then hands the fully-formed packet to networkSystem to be sent over the network. The rspCoder gets an RSP ACK (or a NAK, then retries), then reads a response packet from the network. The rspCoder un-escapes the received packet, tests the checksum, and removes the delimiters before handing the received packet back to the TargetTransaction object and returning to the targetTransactor.

The TargetTransaction object knows how to extract packet-type-specific data from the received packet. The targetTransactor can then retrieve this data using class-specific methods, and return the data to mainControl.

Note that the RSP protocol is designed to have an initiator / target architecture; the server will never send data to the client (AWC) unless AWC initiates the transaction. As such, the network architecture of AWC is single-threaded and blocking; a more complex architecture here would serve no purpose. For clarity, step-by-step examples of register reads and writes will be described below.

To write all registers:

- The user clicks the "Write Registers" button
- guiControlGroup calls method doWriteAllRegisters() in mainControl

- mainControl notifies all RegisterObservers that they must write all data into the register cache
  - Each RegisterObserver in the GUI writes its data to the cache
- For each register in the cache:
  - mainControl calls method writeRegister() in the targetTransactor
  - targetTransactor constructs a WriteRegisterTransaction object, which creates and formats an outgoing data packet
  - targetTransactor hands the WriteRegisterTransaction object to rspCoder.Transact() as a TargetTransaction
  - rspCoder extracts the outgoing data from the TargetTransaction object, escapes it, and adds the checksum
  - rspCoder hands the fully-formed packet to networkSystem.sendData()
  - rspCoder gets characters from networkSystem.getChar() until it has a complete packet
  - rspCoder un-escapes the packet, tests the checksum, and hands it to the WriteRegisterTransaction object via the receivePacket() method
  - receivePacket() parses the packet, and returns true on success
  - rspCoder returns control to the targetTransactor
  - Since no data is returned by a register write, targetTransactor returns to mainControl, which moves on to the next register

To read all registers:

- The user clicks the "Read Registers" button
- guiControlGroup calls method doReadAllRegisters() in mainControl
- For each register in the cache:
  - mainControl calls method readRegister() in the targetTransactor
  - targetTransactor constructs a ReadRegisterTransaction object, which creates and formats an outgoing data packet
  - targetTransactor hands the ReadRegisterTransaction object to rspCoder.Transact() as a TargetTransaction
  - rspCoder extracts the outgoing data from the TargetTransaction object, escapes it, and adds the checksum
  - rspCoder hands the fully-formed packet to networkSystem.sendData()
  - rspCoder gets characters from networkSystem.getChar() until it has a complete packet
  - rspCoder un-escapes the packet, tests the checksum, and hands it to the TargetTransaction object via the receivePacket() method
  - receivePacket() parses the packet, extracts the read register data, and returns true on success
  - rspCoder returns control to the targetTransactor
  - targetTransactor retrieves the register data from the ReadRegisterTransaction object, and returns it to mainControl
  - mainControl writes the data to the cache and moves on to the next registered
- When all registers have been read, mainControl notifies all register observers that the data in the

cache has changed

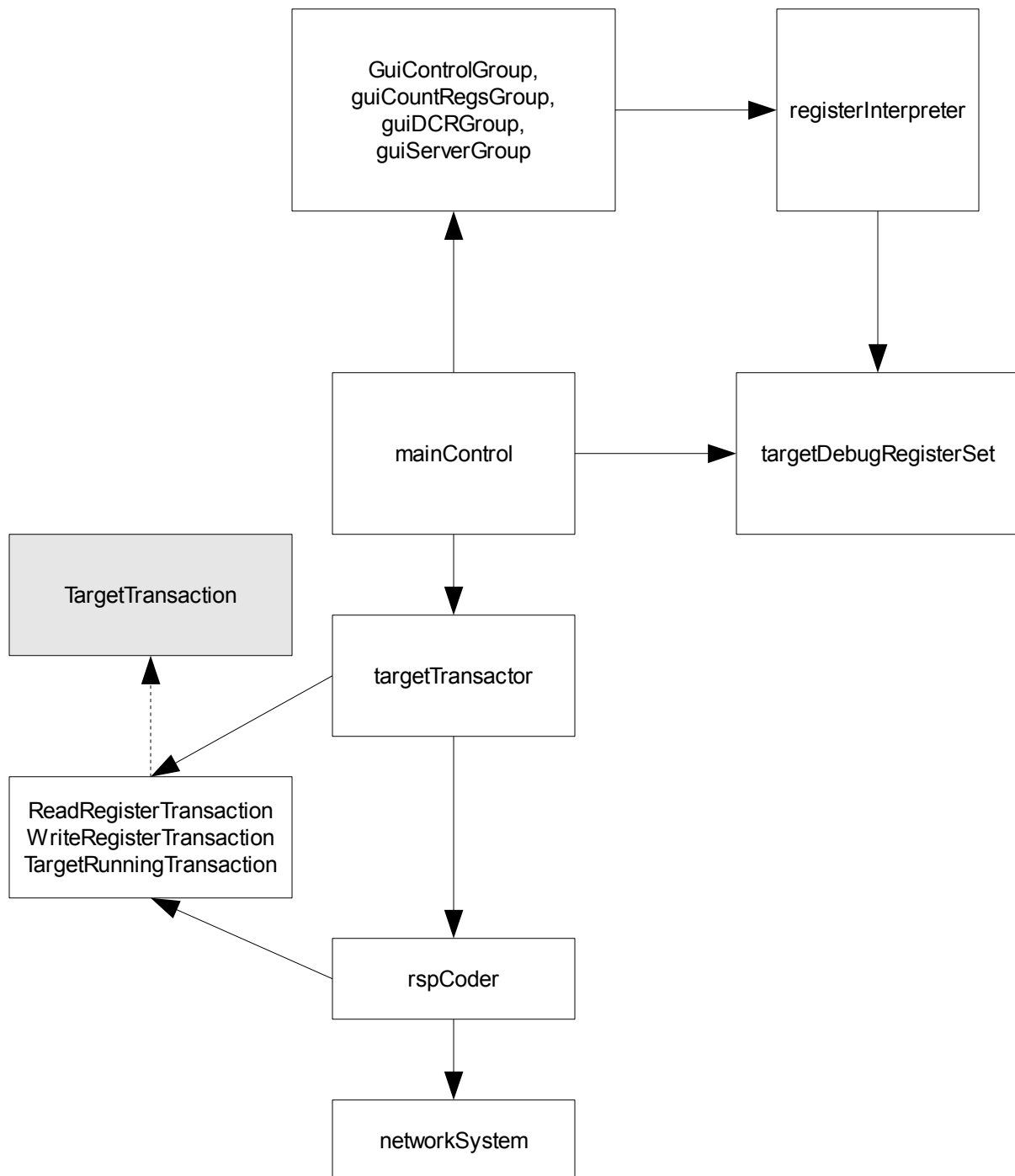o All register observers update their GUI displays with the new data from the cache

```
┌─────────────────────┐           ┌─────────────────────┐
│   GuiControlGroup,   │           │                     │
│  guiCountRegsGroup,  │──────────▶│ registerInterpreter │
│     guiDCRGroup,     │           │                     │
│    guiServerGroup    │           │                     │
└─────────────────────┘           └─────────────────────┘
            ▲                                 │
            │                                 │
            │                                 ▼
┌─────────────────────┐           ┌─────────────────────┐
│                     │           │                     │
│     mainControl     │──────────▶│ targetDebugRegisterSet │
│                     │           │                     │
└─────────────────────┘           └─────────────────────┘
            │
            │
            ▼
┌─────────────────┐     ┌─────────────────────┐
│                 │     │                     │
│ TargetTransaction│◀┄┄ │   targetTransactor  │
│                 │     │                     │
└─────────────────┘     └─────────────────────┘
            ▲                       │
            ┊                       │
┌─────────────────────┐            ▼
│ ReadRegisterTransaction│   ┌─────────────────────┐
│ WriteRegisterTransaction│◀─│      rspCoder       │
│ TargetRunningTransaction│  │                     │
└─────────────────────┘     └─────────────────────┘
                                    │
                                    ▼
                            ┌─────────────────────┐
                            │    networkSystem    │
                            └─────────────────────┘
```

Figure 6: Code Structure