GDB for OR1k

Author: Marko Mlinar
marko.mlinar@opencores.org

Rev. 0.3 Preliminary

June 6, 2001

Revision History

Rev.	Date	Author	Description
0.1	27/4/01	Marko Mlinar	Initial document
0.2	22/5/01	MM	Added more descriptions to software operations
0.3	6/6/01	MM	Added special command descriptions

Contents

Contents ii
Introduction 1
1.1 Framework 1
1.2 Simple GDB Session 1
Supported Features 2
2.1 `hwatch` Command 3
2.2 `htrace` Commands 4
2.2.1 `htrace info` 4
2.2.2 `htrace trigger` 4
2.2.3 `htrace qualifier` 4
2.2.4 `htrace stop` 5
2.2.5 `htrace record` 5
2.2.6 `htrace clear records` 6
2.2.7 `htrace enable` 6
2.2.8 `htrace disable` 6
2.2.9 `htrace mode` 6
2.2.10 `htrace rewind` 7
2.2.11 `htrace print` 7
2.3 Accessing spr Registers 7
2.3.1 `info spr` 7
2.3.2 `spr` 8
2.4 Extended Simulator Support 8
Protocols 9
2.1 JP1 Protocol 9
2.2 JP3 Protocol 10
Software Operation 11
4.1 Reset and Initialization of Remote Target 11
4.2 Communication with Target 12
4.3 Hardware Supported Breakpoints and Watchpoints 12
4.4 Ending Communication 12
Expression BNF 13

1 Introduction

This document describes special support for OR1k in GNU gdb and communication protocols between GDB (GNU Debugger) and JTAG Test-Access-Port

1.1 Framework

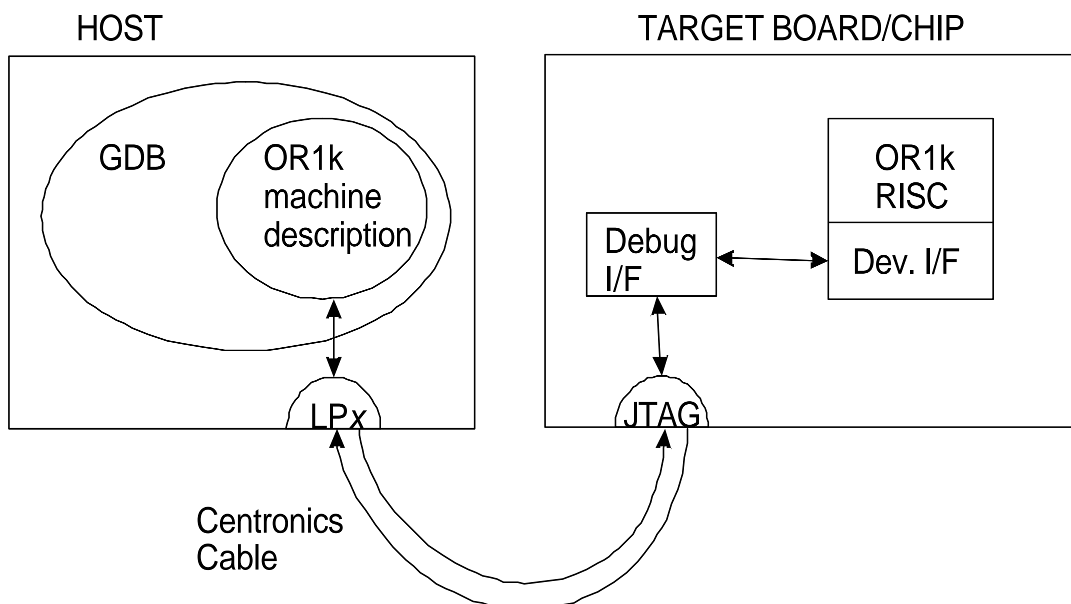

Figure 1: Connection Framework

1.2 Simple GDB Session

Following command sequence start debugging the proc.or32 program, using the architecture simulator. Simulator stops at function main, then next few instructions are shown. See gdb user manual for more examples and additional information.

```
file test.or32
target sim
load test.or32
breakpoint main
run
list
```

2

Supported Features

This section covers debugging features supported by gdb, and describes special commands in detail. This section assumes C/C++ source language. Other language should use equivalent operators.

OR1k feature	Description	Command(s) in GDB	Comment
processor stop	immediately stops OR1k	`^C`	Fully supported.
full register and memory access		`set`	Fully supported.
l.brk	software breakpoint	`breakpoint`	Fully supported. At the same time software conditional breakpoints are supported by GDB.
matches, watchpoints, breakpoints	hardware breakpoint	`breakpoint`, `hbreak`, `watch`, `rwatch`, `awatch`, **`hwatch`**	Hardware breakpoint is fully supported with `hbreak` command. There is limited support for hardware assisted watchpoints using watch, rwatch, awatch commands. OR1k `hwatch` command allow full control over hardware watchpoints. Also HW breakpoints are set to positions where SW cannot be placed (e.g. flash).
trace	trace	`trace`, **`htrace`**	Software trace is supported with `trace` command, while `htrace` fully supports Debug interface trace.
catchpoints	special events, that cause breakpoint		Fully supported.
spr registers	spr register read/write	**`spr`**, **`info spr`**	Display/set specified spr register.
OR1k architecture simulator	Special simulator instructions	**`sim`**	Connection to OR1k architecture simulator, which allows many special diagnostic and profiling functions.

Table 1: List of supported features. Commands in bold represent added instructions.

2.1 `hwatch` Command

Warning: breakpoints, watchpoints and catchpoints have slightly different definition in OR1k architecture document and in gdb.

This command sets hardware assisted watchpoint, if there is enough matchpoint resources. See OR1k Architecture document for more info about these.

Command syntax:
`hwatch expr`

Where `expr` is expression, using logical operators `||` and `&&`. Each condition must consist of one constant (if not it is evaluated when setting watchpoint) and one special value, separated by binary operator (`==`, `!=`, `<`, `>`, `<=`, `>=` and bitwise and - `&`). See appendix A (`watch`) on more details about grammar. Each conditional requires one matchpoint resource.

Special value	Description
`$LEA`	Load effective address
`$SEA`	Store effective address
`$AEA`	like `($LEA == a || $SEA == a)`
`$IFEA`	Instruction fetch effective address
`$LDATA`	Load data
`$SDATA`	Store data
`$ADATA`	like `($LDATA == a || $SDATA == a)`

Table 2: Special Values for Watchpoints

Examples:
```
hwatch ($LEA == my_var)&&($LDATA < 50)||($SEA ==
my_var)&&($SDATA >= 50)
```
(program breaks, either when we load value, lesser than 50 from my_var, or we store value greater than 50 to it)

```
hwatch ($SEA < foo_array  || $SEA >= foo_array_end)&& ($IFEA
>= proc1 && $IFEA < proc2 )
```
(program breaks, if we write outside foo_array in function proc1)

```
hwatch ($AEA & 0x0FF0000)
```
(break occurs, when we want to access specified memory region)

2.2 `htrace` Commands

Group of command used to setup hardware trace.

2.2.1 `htrace info`

Displays info about current trace configuration.

Command syntax:
```
htrace info
```

Examples:
```
htrace info
htrace i
```

2.2.2 `htrace trigger`

Sets starting criteria for trace, if there is enough matchpoint resources. See OR1k Architecture document for more info about these.

Command syntax:
```
htrace trigger [any|breakpoint|<expr>]
```

Where `expr` is expression, using logical operators `||` and `&&`. Each condition must consist of one constant (if not it is evaluated when setting watchpoint) and one special value (Table 2: Special Values for Watchpoints), separated by binary operator (`==`, `!=`, `<`, `>`, `<=`, `>=` and bitwise and - `&`). See appendix A (`match`) on more details about grammar. Each conditional requires one matchpoint resource.

Examples:

`htrace trigger breakpoint`
(trace starts when breakpoint occurs)
`htrace t $SDATA == 0x0beef`
(trace starts when we are storing 0x0beef to memory)
`htrace t any`
(trace active at start)

2.2.3 `htrace qualifier`

Sets data acquire criteria for trace, if there is enough matchpoint resources. See OR1k Architecture document for more info about these. Each time qualifier condition is met and trace has been started data specified by `htrace record` is saved.

Command syntax:
```
htrace qualifier [any|breakpoint|<expr>]
```

Where `expr` is expression, using logical operators `||` and `&&`. Each condition must consist of one constant (if not it is evaluated when setting watchpoint) and one special value (Table 2: Special Values for Watchpoints), separated by binary operator (`==`, `!=`, `<`, `>`, `<=`, `>=` and bitwise and - `&`). See appendix A (`match`) on more details about grammar. Each conditional requires one matchpoint resource.

Examples:
`htrace qualifier breakpoint`
(trace records data when breakpoint occurs)
`htrace q $SDATA == 0x0beef`
(trace records data when we are storing 0x0beef to memory)
`htrace q any`
(trace records data, when active)

2.2.4 `htrace stop`

Sets stoping criteria for trace, if there is enough matchpoint resources. See OR1k Architecture document for more info about these.

Command syntax:
`htrace stop [none|breakpoint|<expr>]`

Where `expr` is expression, using logical operators `||` and `&&`. Each condition must consist of one constant (if not it is evaluated when setting watchpoint) and one special value (Table 2: Special Values for Watchpoints), separated by binary operator (`==`, `!=`, `<`, `>`, `<=`, `>=` and bitwise and - `&`). See appendix A (`match`) on more details about grammar. Each conditional requires one matchpoint resource.

Examples:
`htrace stop none`
(trace does not stop)
`htrace s $SDATA == 0x0beef`
(trace starts when we are storing 0x0beef to memory)

2.2.5 `htrace record`

Sets record data to be stored into trace buffer, when qualifier occurs. Command failes if there is not enough matchpoint resources. See OR1k Architecture document for more info about these.

Command syntax:
```
htrace record {[PC|LSEA|LDATA|SDATA|READSPR|WRITESPR|INSTR]}*
[when [breakpoint|<expr>]]
```

First data to be recorded is specified, and after `when` additional condition is set. `expr` is expression, built using logical operators `||` and `&&`. Each condition must consist of one

constant (if not it is evaluated when setting watchpoint) and one special value (Table 2: Special Values for Watchpoints), separated by binary operator (`==`, `!=`, `<`, `>`, `<=`, `>=` and bitwise and - `&`). See appendix A (`match`) on more details about grammar. Each conditional requires one matchpoint resource.

Examples:
`htrace record PC SDATA when $SEA == 100`
(saves PC and SDATA when store to location 100 occurs)
`htrace r when $SEA == 100`
(removes previously allocated record)

2.2.6 `htrace clear records`

Deallocates all matchpoint resources, allocated by `htrace record` command.

Command syntax:
`htrace clear records`

Example:
`htrace clear records`

2.2.7 `htrace enable`

Enables trace. This command has to be specified in order to start trace.

Command syntax:
`htrace enable`

Example:
`htrace enable`

2.2.8 `htrace disable`

Temporarily disables trace, execute `htrace enable` command to reenable it.

Command syntax:
`htrace disable`

Example:
`htrace disable`

2.2.9 `htrace mode`

Changes trace mode. If `continuous` parameter is specified, hardware trace buffer will be rewritten, otherwise breakpoint will occur.

Command syntax:
`htrace mode [suspend|continuous]`

Example:
`htrace mode suspend`

2.2.10 `htrace rewind`

Clears currently recorded trace data. If filename is specified, new trace file is made and any newly collected data will be written there

Command syntax:
`htrace rewind [new_file_name]`

Examples:
`htrace rewind`
`(clears trace buffer)`
`htrace rewind`
(does not clear current trace buffer, but starts a new trace)

2.2.11 `htrace print`

Prints selection of currently collected records from hardware trace buffer.

Command syntax:
`htrace print [from [length]]`

Example:
`htrace print 0 20`
(prints first 20 records)
`htrace p 10`
(prints records starting at record 10, using previous length)
`htrace p`
(prints next records, using last length)
`htrace p -10 10`
(prints last ten records)
`htrace p 0x1000 -10`
(prints ten records before record 0x1000)

2.3 Accessing spr Registers

Group of command for handling spr registers.

2.3.1 `info spr`

Display contents of specified spr register.

Command syntax:
`info spr [register_name | group_name [register_name] | ]`

Examples:
`info spr`
`(display spr groups)`
`info spr SYS`
(display registers in group 0)
`info spr SYS UPR`
info spr UPR
info spr SYS 1
(all three prints value of UPR register)
info spr 10 0
info spr SPR10_0
(both display first register in group 10)

2.3.2 spr

Modify contents of specified spr register.

Command syntax:
`spr [register_name | group_name [register_name] | ] value`

Examples:
`spr SYS PC 0x1234`
`spr PC 0x1234`
(both sets PC to 0x1234)

2.4 Extended Simulator Support

To allow extra simulator capabilities `sim` command is available.

Command syntax:
`sim <simulator command line>`

Examples:
`sim r`
(display contents of all registers)

3

Protocols

Two simple proprietary protocols for communications between remote target and gdb are shown. Both require parallel port, and very low amount of additional hardware. Beside these two, faster protocol is in preparation, which is to allow much higher transfers, using EPP parallel port mode (bi-directional mode).

2.1 JP1 Protocol

JP1 protocol is simple JTAG compatible protocol. It does not need any extra hardware, except voltage adjustment circuitry.

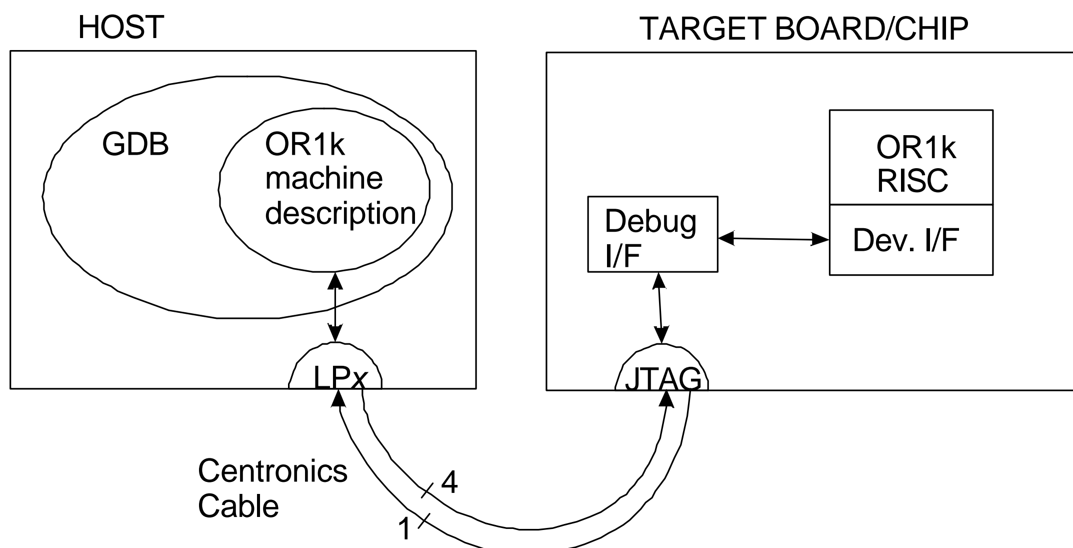

Figure 2: JP1 Protocol

Each JTAG cycle requires 2 parallel port writes and one read (if necessary) from host. First one lowers the clock and sets the data (RSTn, TMS and TDI). Second write does not modify the data, but raises the clock. See JTAG specification for more info. Then one bit is read from CENTRONICS_BUSY signal, using IOCTL.

Port	Description	Width	Direction (relative to host)	Assigned centronics pin
TCLK	Clock	1	Output	D0
TRSTn	Reset	1	Output	D1

TMS	Mode Select	1	Output	D2
TDI	Data Input	1	Output	D3
TDO	Data Output	1	Input	BUSY

2.2 JP3 Protocol

Unlike JP1, JP3 requires small amout of extra logic (e.g. PLD) on the board, but is six times faster.

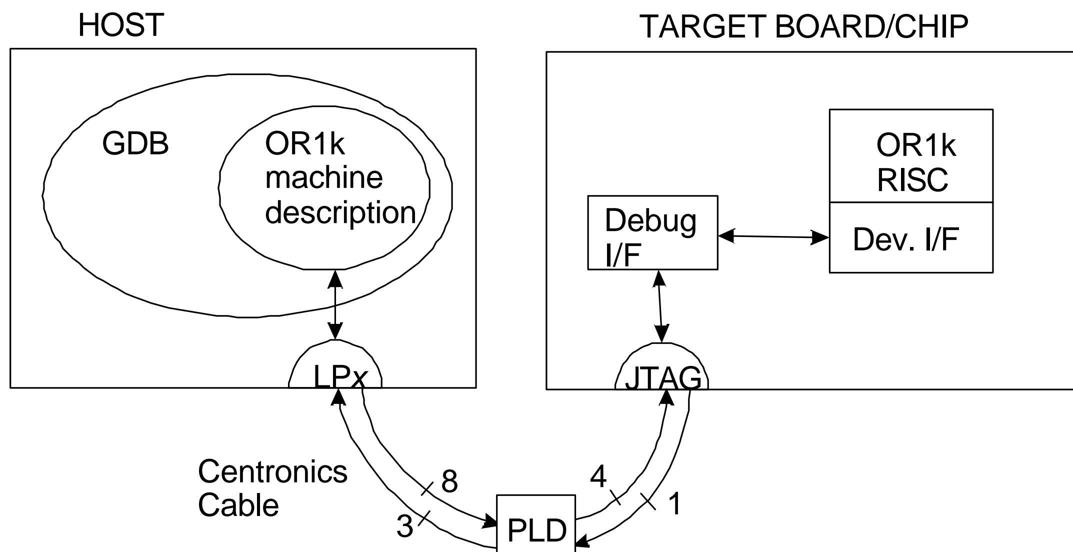

Figure 3: JP3 protocol

This protocol does not directly change signals of JTAG port, but instead sends three pairs (TMS, TDI) and receives three TDO signals. CLK signal has different meaning: both clock positive and clock negative edge represents data valid. If bitstream length is not of modulo 3, then zeros are appended to TMS, data is x. This way JTAG stays in RUN_TEST/IDLE state.
Shortly, PLD circuit should translate JP3 protocol to JP1 for each data.

Port	Description	Width	Direction (relative to host)	Assigned centronics pin(s)
TCLK	Clock	1	Output	D0
TRSTn	Reset	1	Output	D1
TMS	Mode Select	3	Output	D2, D4, D6
TDI	Data Input	3	Output	D3, D5, D7
TDO	Data Output	3	Input	BUSY, PAPER_ERR, SELECT

4

Software Operation

This section deals with the software operation.

Communication example: Setting SW Breakpoint

It all starts when setting breakpoint in gdb prompt:
`(gdb) breakpoint 0x1234`
GDB then internally searches for target specific macros, like (INSERT_BREAKPOINT, TARGET_XCHG_MEMORY, BREAKPOINT_FROM_PC, …) to replace instruction at address `0x1234` with `l.brk`. Previous instruction is backed into host buffer. When OR1k encounters `l.brk` instruction it halts. GDB meanwhile continuously polls processor status. Note that processor can be stopped using access to OR1k registers.

GDB (remote) target uses JP1/3 protocol via parallel port driver (e.g. /dev/lp0) and JTAG I/F to access OR1k registers, as specified in Debug Interface Document and OR1k architecture document. For each 32b memory or register access we have to send 65 bits (data, R/W bit and address), 8 bit CRC and some control bits (for JTAG purposes). See the RISC Development document for details. Using JP3 protocol we don't need to send extra dummy bits (one transfer requires exactly 24 parallel port writes, and for reading extra 11 reads).

4.1 Reset and Initialization of Remote Target

In order to debug the target, program has to be transferred to a stable environment. Since after the chip reset the processor is surely in stable and well defined state, it is naturally to stall processor right after the reset. Implementation specific processor info is then read, and program data is transferred. Right after that remote debugging can start.
More accurately - following steps are taken:

1. set processor reset
2. set processor stall
3. unset processor reset
4. read implementation specific registers and configure gdb (e.g. UPR)
5. set debug specific registers to idle state
6. transfer data (when user executes `load` command)
7. unstall processor

4.2 Communication with Target

It is not smart to do complex operations while processor runs, since we can enter unpredicted state. During such complex operation (program loading, setting breakpoints, etc.) processor is stalled. Smaller operations like register or memory read can be made during normal processor operation[1].

4.3 Hardware Supported Breakpoints and Watchpoints

Since debug unit has limited number of matchpoint resources, they should be used wisely. gdb default operation is first to set HW breakpoints and then SW ones. Hardware breakpoint can be set explicitly on e.g. some ROM location, using `hbreak` command.
DVRx and DCRx pairs are programmed to set proper matchpoints. Normal breakpoints use only one matchpoint, while watchpoints at least two (e.g. data access watchpoint is set on memory address range, thus yielding conditional: addr >= 0x1000 && addr <= 0x1003). For each watchpoint chaining is set in DMR1 register to properly connect matchpoint conditionals. We always tend to use lower indexes first and sometimes mathcpoints must be reordered to find optimal fit.

4.4 Ending Communication

It is not necessary for gdb to do anything when ending remote session. However, sometimes processor is connected to viable equipment. If continuing program or unpredicted state is entered, damage can occur, thus processor stall is attempted[2].

[1] gdb user must be aware that he is using asynchronous operation.
[2] Note that it is not always possible for gdb to properly end communication, e.g. cable to the target is disconnected.

Appendix A

Expression BNF

Since expressions have limited hardware support (we have limited hardware resources), not every expression can be specified. gdb automatically translates normal expression to fit hardware resources, so user don not have to worry about it at all - it will report an error, if expression is too complex to fit into OR1k development scheme, so user can rephrase expression. gdb can translate all expressions to hardware ones, if that is possible.
Basically our expression grammatics is very similar to gramatics with logical operators `||` and `&&`, without priorities (e.g. `(a || b) && c` would be `a || b && c`). For example simple calculators accept expressions without priority - e.g. you cannot calculate `(8/5)+(7*6)`.

BNF of OR1k matchpoint grammatics is:

<watch> ::= <match> | <match> || <watch>
<match> ::= <cond> | <match_o> || <match> | <match> || <match_o> | <match_a> &&| <match> | <match> && <match_a>
<match_o> ::= <cond> | <cond> || <match_o>
<match_a> ::= <cond> | <cond> && <match_a>
<cond> ::= <ct> <cc> <const> || <const> <cc> <ct
<ct> ::= $LEA | $SEA | $AEA | $IFEA | $LDATA | $SDATA | $ADATA
<cc> ::= < | > | == | != | <= | >= | &
<const> any numeric constant specified, read from register or memory, or obtained from symbol table