ARC® GDB-Insight
GNU Debugger with Insight GUI

Getting Started

ARC® GDB-Insight Getting Started

Virage Logic Corporation
47100 Bayside Parkway
Fremont, California 94538, USA
Tel: +1-510-360-8000
Toll-free: 877-360-6690

www.viragelogic.com

6009-003 March 2010

Contents

Chapter 1 — Overview

This manual provides an overview of GDB for the ARC 700 and ARC 600 processors.

This document is intended to be used in conjunction with the online documentation available for GDB. Please refer to Appendix A for more information on using the GDB online documents.

Versions of tools provided in the release:

- arc-elf32-gdb: GDB host debugger serving arc elf32 bare-metal targets such as:
  - CGEN ARC simulator.
  - xISS (ARC proprietary fast execution instruction set simulator).
  - ARCangel 4 hardware emulator via parallel port JTAG interface.
- arc-linux-uclibc-gdb: GDB host debugger serving Linux ARC700 targets
- gdbserver: ARC700 Linux target gdb server application serving host gdb (arc-linux-uclibc-gdb) via TCP/IP remote connection.
- Insight: Integrated GUI (graphical user interface) to GDB.

Chapter 2 — Building GDB

This chapter covers the details of building GDB and gdbserver for the ARC platform.

Configurations

- arc-elf32: This configuration allows debugging of ARC 600 and ARC 700 bare metal GCC applications on a variety of targets, including:
  - — CGEN ARC simulator.
  - — xISS (Fast Instruction Set Simulator).
  - — ARCangel 4 hardware emulator.
- arc-linux-uclibc: This configuration allows for remote debugging of GNU/Linux applications on the ARC 700 Linux platform.

Compiling and Installing

GDB can be built from source on most Linux based workstations. The configuration requires that a number of packages aiding the build system are present on the machine. In most cases the configuration can work around missing packages; however in certain cases this will not be possible and the build system will report a build failure. In this case correction of the problem will be possible by installing the required package on your system.

Software Required

The main software required to build GDB is as follows:

- Host GNU C Compiler.
- Autotools.
- C Development Environment on the Host where GDB is being built.
- arc-linux-uclibc-gcc tool chain for ARC 700 (required only for building gdbserver).

Instructions for Building

This section tells how to build the different configurations.

arc-linux-uclibc-gdb (ARC 700 Only)

From the top source install directory run the following scripts:

```
$ ./configure  --target=arc-linux-uclibc  --prefix=/install/path
$ make
$ make install
```

Where */install/path* is a directory where you want to install GDB.

gdbserver

From the gdbserver subdirectory run the following scripts:

```
$ source ./build_gdbserver.sh
$ ./configure  --host=i386-redhat-linux-gnu --target=arc-linux-uclibc  --
prefix=/install/path
$ make
$ make install
```

gdbserver is created in your install directory. To use gdbserver, the executable must be made available to the target Linux machine, for example an ARCangel hardware emulator. To do so, either place the executable on the Linux RAMDISK image, on the Linux IDE drive, or on a mountable network drive.

arc-elf32-gdb

1. Ensure that the expat library (available from http://www.libexpat.org) is installed at `/expat/install/path`.
2. From the top source install directory, run the following scripts:

```
$ ./configure  --target=arc-elf32   --prefix=/install/path  \
                      --with-expat  --with-libexpat-prefix=/expat/install/path \
                  [   --with-xiss     --with-xiss-prefix=/xiss/install/path    ]
$ make
$ make install
```

Where */install/path* is a directory where you want to install GDB.

If you have the xISS installed on your machine, you might have to build gdb to be able to use it.

Chapter 3 — Using GDB

GDB for ARC is used either standalone or in conjunction with the integrated graphical user interface (GUI) debugger 'insight'. GDB can also be used as an "inferior debugger" to GUI front-ends such as DDD, Eclipse CDT, etc.

Debug targets supported by GDB are:

- CGEN ARC simulator (standard integrated GDB ARC core simulator)
- gdbserver (remote target via TCP/IP socket)
- ARCangel 4 Hardware Emulator (JTAG target)
- xISS ARC Simulator (remote target via TCP/IP local and remote sockets).
- xISS ARC Simulator (dynamically-loaded target)

NOTE	ARCangel 4 and xISS targets are ARC proprietary debug components not included with the GDB release.

A description on how to use the different debug targets supported by GDB is given below:

CGEN ARC Simulator

The CGEN simulator is built into the arc-elf32-gdb debugger. It is intended to simulate ARC600 and ARC700 core instruction execution. The simulator is employed to debug applications compiled for the ARC architecture. The current CGEN simulator does not provide processor peripheral simulation support.

Do the following to invoke the CGEN ARC simulator with GDB:

`$ arc-elf32-gdb`	Invoke the bare-metal GDB debugger for ARC.
`(gdb) file a.out`	Inform GDB of the application to be debugged (a.out is an ELF binary normally generated using the arc-elf32-gcc compiler). The ELF binary contains information to configure a number of GDB internal target parameters to the correct values.
`(gdb) target sim`	Connect to the CGEN ARC simulator.
`(gdb) load`	Load the code and data of the a.out binary into the CGEN simulator.
`(gdb) start`	Start the application.

All GDB commands are available to aid the debugging session.

xISS ARC Simulator

The xISS (ARC Fast Instruction Set Simulator) simulator can be connected with GDB to support debugging of applications that require peripheral support for ARC 600 and ARC 700 processors. xISS can be configured to support a number of ARC processor modules that can

match the exact target hardware specifications. In this way applications can be developed initially to run on xISS without the need for expensive emulation hardware.

NOTE	xISS software is not part of the GDB release. For detailed information on how to obtain/use this product contact ARC support.

Do the following to invoke the xISS ARC simulator with GDB:

Open a separate terminal window to start the xISS GDB server in.

`$ xiss_gdb :1234 –sys=linux700.xis vmlinux_700`	Start the xISS server, enabling an xISS communication channel with GDB on port number 1234. The xISS processor configuration is supplied by file linux700.xis whilst the executable binary is vmlinux_700 (in this example, an ELF binary, normally generated using the arc-elf32-gcc compiler, containing a specific version of the ARC700 Linux kernel).
`$ arc-elf32-gdb`	Invoke the bare metal GDB debugger for ARC.
`(gdb) file vmlinux_700`	Inform GDB of the application to be debugged. The ELF binary contains information to configure a number of GDB internal parameters to the correct values.
`(gdb) target remote localhost:1234`	Connect GDB to the xISS server using port 1234.
`(gdb) load`	Load the code and data of the Linux kernel binary into the xISS simulator (this is not necessary if the vmlinux_700 binary was specified as the executable to the xISS server when it was started, as was the case in this example).
`(gdb) break main`	Set a breakpoint on the ‘main’ function of the application.
`(gdb) continue`	Start the application.

All GDB commands are available to aid the debugging session except **start** and **run**.

Using the xISS Directly

It is also possible to use the xISS directly, if arc-elf32-gdb is built to support this by building as follows:

`$ arc-elf32-gdb`	Invoke the bare metal GDB debugger for ARC.
`(gdb) file a.out`	Inform GDB of the application to be debugged (a.out is an ELF binary normally generated using the arc-elf32-gcc compiler). The ELF binary contains information to configure a number of GDB internal target parameters to the correct values.
`(gdb) target arcxiss linux700.xis`	Connect to the xISS ARC simulator, supplying the xISS processor configuration in file linux700.xis.
`(gdb) load`	Load the code and data of the a.out binary into the xISS

simulator.

```
(gdb) start
```

Start the application.

All GDB commands are available to aid the debugging session.

ARCangel 4

GDB provides support for debugging ARC-based applications on the ARCangel 4 hardware emulator.

Note: ARCangel 4 hardware emulator is not part of GDB release. For detailed information on how to obtain/use this product contact ARC support.

To use the ARCangel 4 target with GDB you will need to do the following steps.

- Install the gpio Linux module. The module provides parallel port driver support to the host Linux machine. Contact ARC support to obtain a compliant gpio driver.
- Load a valid CPU emulation binary (XBF) on the ARCangel 4 (make sure that the CPU emulation binary is built with JTAG module support). Alternatively, the binary may be loaded by the GDB *target arcjtag* command after GDB has been invoked.
- Once the ARCangel 4 CPU emulation binary is loaded, GDB can be connected to the target as follows:

$ arc-elf32-gdb – invoke the bare metal GDB debugger for ARC.

(gdb) file a.out – inform GDB of the application to be debugged (a.out is an ELF binary normally generated using the arc-elf32-gcc compiler). The ELF binary contains information to configure a number of GDB internal target parameters to the correct values.

(gdb) target arcjtag – connect to the ARCangel 4 target using the gpio parallel port driver that communicates with the JTAG module on the ARCangel 4 hardware.

(gdb) load – load the code and data of the a.out binary onto the ARCangel 4 target.

(gdb) start – start the application.

All GDB commands are available to aid the debugging session.

Linux Host

You must install the gpio Linux module on your host machine in order for the arc-elf32-gdb debugger to be able to communicate with the JTAG module of an ARCangel 4 target.

To do so, use the following command:

```
$ /sbin/insmod   gpio.ko   gpio_major=250
```

assuming that you have the required driver in the file `gpio.ko` in your current directory. You must root privileges to execute the command.

To check that the driver installation is successful, use following commands; their output indicates whether the gpio device exists with major number 250, and whether the module is installed.

```
$ cat   /proc/devices  |   grep  gpio
250   gpio
$ cat   /proc/modules  |  grep  gpio
```

```
gpio  <size>   0  -  Live   <hex  offset>  (U)
```

gdbserver (ARC 700 Linux Only)

gdbserver is an ARC700 Linux user mode application that provides support for GDB to debug Linux user mode applications on ARC700 Linux. The GDB host debugger that communicates with gdbserver is arc-linux-uclibc-gdb. gdbserver has been developed and tested with ARCangel 4 target hardware capable of running Linux on ARC 700 only. The ARCangel 4 target communicates with the host debugger via a TCP/IP socket connection.

The gdbserver executable must be accessible from a Linux shell on the ARCangel 4 (this could be achieved by putting the gdbserver executable on an NFS share or a Linux RAMDISK file system).

If the file to be debugged is called `file.out` (an ELF binary normally generated using the arc-linux-uclibc-gcc compiler); the first step is to start gdbserver, which must be done by commands issued on the ARC700 Linux:

```
$ telnet   <IP address of target ARCangel 4>
```

NFS mount the installation directory where "gdbserver" is installed onto the mount point /nfs.

```
$ mount –t nfs –o nolock <IP address of host computer>:<directory where gdbserver
resides>  /nfs
$ cd  /nfs
$ ./gdbserver  :1000  /nfs/<directory where file.out binary is stored>/file.out
```

The second step is to start arc-linux-uclibc-gdb on the host computer, e.g.:

```
$ arc-linux-uclibc-gdb
(gdb) file /<directory where file.out binary is stored>/file.out
(gdb) target remote <IP address of target ARCangel 4>:1000
(gdb) load
(gdb) start
```

Note that the same port number (1000 in this example) must be specified to both gdbserver and GDB.

All GDB commands are available to aid the debugging session except those related to hardware breakpoints and watchpoints, auxiliary registers, target configuration and clock settings.

Auxiliary Registers (arc-elf32-gdb only)

When using arc-elf32-gdb, you must provide it with a file that describes the auxiliary registers of the target (e.g. ARCangel 4, simulator) upon which the application is being debugged.

By default, GDB will look for a file named `arc-registers.xml`; it will look first in the current working directory, and then in the user's home directory. Alternatively, a file may be specified by means of the **arc-reg-read-file** command.

It is also possible to add additional register descriptions using the **arc-reg-read-extra-file** command, thus allowing processor variants to be described by sets of common files and variant-specific files (e.g. containing descriptions of cache or MMU-related registers).

Files containing descriptions for the ARCtangent-A5, ARC 600, ARC 700 variants of the ARC processor architecture are provided at

```
source_directory/gdb/features/arc-a5-cpu.xml
source_directory/gdb/features/arc600-cpu.xml
source_directory/gdb/features/arc700-cpu.xml
```

It is suggested that the user should copy the file appropriate to the target variant being used to his working directory or home directory, and rename it to *arc-registers.xml*.

It is simple to define a new target description; the XML schema for the description is defined in the file *source_directory*`/gdb/features/arc-registers.dtd`.

In essence, a target description consists of a set of register descriptions. Each register description has these attributes:

- name - the register name (case-insensitive); may be ‘unused’
- description - an (optional) textual description of the register's function
- number - the number of the register in the auxiliary register set (e.g. 0x0A for STATUS32)
- mask - a 32-bit mask which defines which bits of the register are valid (0xFFFFFFFF by default)
- access - the register's read/write access: R/O, R/W or W/O (R/W by default)

Each register description may also have a set of field descriptions. Each field description has these attributes:

- name - the field name (case-insensitive); may be 'reserved'
- description - an (optional) textual description of the field's function
- onwrite - the (optional) value (for a reserved field) which must be supplied on a write operation
- offset - the offset in bits of the field from the least significant bit (0) of the register
- size - the size of the field in bits
- access - the field's read/write access: R/O, R/W or W/O (that of the register by default)

Each field description may also have a set of field meanings. Each field meaning has these attributes:

- value - a value that the field may contain
- description - a textual explanation of the meaning of the field when it contains that value

There may also be a set of Build Configuration Registers (BCRs). The descriptions of these registers do not have an access attribute; and, although a BCR may have a number of fields (which do not have access or onwrite attributes), these fields do not have field meanings.

There may also be a set of Extension Core Registers (ECRs). The descriptions of the registers have merely the register number (which must be in the range 32 .. 59, and mask and access attributes.

ARC-Specific GDB Commands

The standard GDB commands are organized in functional groups. The specification of all new commands added to the ARC GDB is summarized below.

Breakpoint Commands

The break commands described here assume the target processor includes a debug module supporting hardware actionpoints. Up to eight actionpoints are available for an ARC processor.

arc-break-range—Set a breakpoint on a memory address range

Usage: arc-break-range <START> <LENGTH>

<START> is an address. <LENGTH> is in bytes.

Example:

```
(gdb) arc-break-range 0x10000 0x1000
Hardware assisted breakpoint 1 at 0x10000 covering 4096 bytes
```

arc-watch-range—Set a watchpoint on a memory address range

Usage: arc-watch-range <START> <LENGTH> [read | write | access]

<START> is an address. <LENGTH> is in bytes. The third parameter controls whether the action is triggered when the processor reads, writes or accesses (reads or writes) within the given memory range. If it is omitted, ‘write’ is assumed.

Example:

```
(gdb) arc-watch-range 0x10000 0x1000 read
Hardware read watchpoint 3: 0x00010000:4096
```

Auxiliary Register Commands

These commands are available only in the arc-elf32 build of the debugger.

arc-aux-read—Read and display a range of auxiliary registers

Usage: arc-aux-read <REG-FROM> [<REG-TO>]

Read and display a range of auxiliary registers. <REG-FROM> and <REG-TO> can be (case-insensitive) register names, or any expressions that evaluate to integers. <REG-TO> is optional; if it is not specified, only one register is displayed.

Examples:

```
(gdb) arc-aux-read 0x400 0x404
0x00000400 ERET                 : 00000100
0x00000401 ERBTA                : 80001ABC
0x00000402 ERSTATUS             : 00000000
0x00000403 ECR                  : 00002000

(gdb) arc-aux-read PC ICACHE_CONTROL
0x00000006 PC                   : 00000238
0x0000000a STATUS32             : 00000800
0x0000000b STATUS32_L1          : 00000000
0x0000000c STATUS32_L2          : 00000000
0x00000010 ICACHE_IVIC          : 00000000
0x00000011 ICACHE_CONTROL       : 00000000
```

Note that nothing is displayed for register numbers in the auxiliary register space that do not correspond to actual registers.

arc-aux-write—Write to an auxiliary register

Usage: arc-aux-write <REG> = <VALUE>

Write to an auxiliary register. <REG> can be a (case-insensitive) register name, or any expression that evaluate to an integer. <VALUE> can be any expression that evaluates to an integer.

Examples:

```
(gdb) arc-aux-write 6 = 0x123
(gdb) arc-aux-write COUNT0 = 84
```

arc-aux-list—List auxiliary register(s)

Usage: arc-aux-list [<REG>]

Give a description of an auxiliary register. <REG> can be a (case-insensitive) register name, or any expression that evaluates to an integer. If <REG> is omitted, all auxiliary registers are listed.

The description of a register may include descriptions of the fields of the register (if it has fields); for each field, there may be given meanings for specific values which the field may contain, as well as its position within the register, in the format <field_offset_from_register_bit0> : <bits_in_field>.

Examples:

```
(gdb) arc-aux-list STATUS32
 CONTROL0
    description: Processor Timer 1 Control Value
    number     : 0x22
    access     : read/write
    fields
    IE
        description: Interrupt Enable flag
        position   : 0:1
        access     : read/write
        field meanings
          1 ==> interrupt will be generated after timer reaches limit condition
    NH
        description: Not Halted mode flag
```

```
        position   : 1:1
        access     : read/write
        field meanings
          0 ==> timer counts every clock cycle
          1 ==> timer counts clock cycles only when processor is running
    W
        description: Watchdog mode flag
        position   : 2:1
        access     : read/write
        field meanings
          1 ==> system Reset signal will be generated after timer reaches limit
condition
    IP
        description: Interrupt Pending flag
        position   : 3:1
        access     : read/write
        field meanings
          0 ==> timer 0 interrupt line is low
          1 ==> timer 0 interrupt line is high
```

arc-aux-show—Show auxiliary register(s)

`Usage: arc-aux-show [<REG>]`

Show the contents of an auxiliary register. <REG> can be a (case-insensitive) register name, or any expression that evaluates to an integer. If <REG> is omitted, all auxiliary registers are shown.

If the register has fields, the contents of those fields will also be shown; for each field, if the field contains a specific value for which there is a known meaning, that meaning will also be shown.

`Examples:`

```
(gdb) arc-aux-show STATUS32
STATUS32 : 00000000
   fields
      H : 1    (processor is halted)
      E1: 0    (level 1 interrupts are enabled)
      E2: 0    (level 2 interrupts are enabled)
      A1: 0
      A2: 0
      AE: 0
      DE: 0
      U : 0    (processor is in Kernel mode)
      V : 0
      C : 0
      N : 0
      Z : 0
      L : 0    (zero overhead loop mechanism is enabled)
```

arc-reg-read-file—Read auxiliary or extension core register definitions from file

`Usage: arc-reg-read-file <FILEPATH>`

Read the definitions of a set of auxiliary and/or extension core registers from a file. This set will replace the set of definitions currently being used by GDB.

Examples:

```
(gdb) arc-reg-read-file  /project/arc/arc700-registers.xml
Register definitions read from file /project/arc700-registers.xml
```

arc-reg-read-extra-file—Read additional auxiliary or extension core register definitions from file

Usage: arc-reg-read-extra-file <FILEPATH>

Read the definitions of a set of auxiliary and/or extension core registers from a file. This set is added to the set of definitions currently being used by GDB.

Examples:

```
(gdb) arc-reg-read-extra-file  /project/arc/cache-aux-registers.xml
Register definitions read from file /project/cache-aux-registers.xml
```

Internal Commands

set arc-debug -- Set whether to print ARC debug messages

Usage: set arc-debug

Example:

```
(gdb) set arc-debug 1
```

set arcjtag-debug-statemachine

Usage: set arcjtag-debug-statemachine <INTEGER>

Switch on JTAG state machine debugging messages if <INTEGER> is non-zero. Switch them off if it is zero.

Example:

(gdb) set arcjtag-debug-statemachine 1

set arcjtag-retry-count -- Set the number of attempts to be made for a JTAG operation

Usage: set arcjtag-retry-count <INTEGER>

Example:

```
(gdb) set arcjtag-retry-count 10
```

Obscure Commands

arc-fill-memory—Fill a memory address range with a repeated pattern

Usage: arc-fill-memory <START> <LENGTH> [<PATTERN>]

This is useful for filling a region of memory with a known and easily recognizable value, e.g. 0xDEADBEEF. <START> is an address. <LENGTH> is in bytes. If <PATTERN> is omitted, 0 is assumed.

Example:

```
(gdb) arc-fill-memory 0x100 0x200 0xCAFEBABE
```

arc-list-actionpoints—List the hardware actionpoints

Usage: arc-list-actionpoints

Example:

```
(gdb) arc-list-actionpoints
AP 0          :: value    : 00000234
                 mask     : 00000000
                 control  : 00000020 Instruction Address, break on read in range
AP 1          :: value    : 00080100
                 mask     : 000000FF
                 control  : 00000023 Load/Store Data, break on read in range
AP 2          :: not in use
AP 3          :: not in use
AP 4          :: not in use
AP 5          :: not in use
AP 6          :: not in use
AP 7          :: not in use
```

arc-show-frame—Display the stack frame with annotation

Usage: arc-show-frame [<FRAME>]

Where <FRAME> is an expression that evaluates to an integer. A value of 0 indicates the current frame, 1 indicates the calling frame, and so forth.

Example:

```
(gdb) arc-show-frame
Frame of function: main (hello.c:7)
0x0002038C:              00000178     saved register 'blink'
0x00020388:      FP ===> 00000000     saved register 'fp'
0x00020384:              00020388     local variable 'result'
0x00020380:              00000074     parameter 'argc'
0x0002037C:              000202B0     parameter 'argv'
0x00020378:      SP ===> CA45360B
```

Target Commands

target arcjtag—Connect to a remote ARC target via a JTAG interface

Usage: target arcjtag [noreset | <XBF-FILE>]

Connect to the arcjtag target. This target expects the ARC board to be connected to the parallel port on the host. Currently debugging is supported only on GNU/Linux hosts. This target uses the gpio (General Purpose Input/Output) device to access the parallel port. You must have the gpio device driver installed and you must have read/write privileges to /dev/gpio.

By default, GDB will reset the target board upon connection. If the option *noreset* is given, this will not be done; this is useful when connecting to a target upon which a program is already being executed: the program execution is interrupted and the processor halted, and debugging may proceed from that point.

If the <XBF-FILE> parameter is supplied, the target board's FPGA will be configured ("blasted") using the contents of the given XBF file; the target is left in the reset state.

Examples:

```
(gdb) target arcjtag
Processor is halted.
Connected to the arcjtag target.

(gdb) target arcjtag  arc700.xbf
CPLD Revision = 1010 1001 1000 1000
********************
FPGA configured
Connected to the arcjtag target.
```

target arcxiss—Connect to a xISS ARC target

Usage: target arcxiss [<XIS-FILE>]

Connect to the arcxiss target. This target expects the xISS to be installed in a directory identified by the environment variable XISS_HOME.

If the <XIS-FILE> parameter is not supplied, the debugger will attempt to use the file 'default.xis' in the current directory.

Example:

```
(gdb) target arcxiss linux700.xis
Connected to the arcxiss target.
```

arc-blast-board—Blast the board's FPGA

Usage: arc-blast-board <XBF-FILE>

Configure the board. For this command to work, the debugger must be connected to the arcjtag target (see command *target arcjtag*).

Example:

```
(gdb) arc-blast-board  arc600_mmu.xbf
CPLD Revision = 1010 1001 1000 1000
********************
FPGA configured
```

arc-set-clock-frequency—Set the board clock frequency

Usage: arc-set-clock-frequency [<CLOCK> =] <FREQUENCY> [, <FREQUENCY>]

Set the one or both PLL clock frequencies for the board (the target board PLL actually possesses two clocks: a Memory and I/O Timing clock (MCLK) and a Video Clock (VCLK)). For this command to work, the debugger must be connected to the arcjtag target (see command *target arcjtag*).

<CLOCK> may be either *mclk* or *vclk*; if it is omitted and only one frequency is given, *mclk* is assumed. If <CLOCK> is omitted and two frequencies are given, the first frequency is for MCLK and the second for VCLK.

<FREQUENCY> is a number which is interpreted as Megahertz; because of the digital nature of the clocks, it may not be possible to set exactly the frequency requested; instead, the closest frequency which may be set is used.

Note that if only one clock frequency is set, and the frequency of the other clock has not yet been explicitly set, the debugger will set the other clock to a frequency which is not an integer multiple of the given frequency – this is in order to avoid clock signal degradation through jitter.

Examples:

```
(gdb) arc-set-clock-frequency mclk=40
PLL clock MCLK set to 40.01 MHz.
PLL clock VCLK set to 57.27 MHz.
(gdb) arc-set-clock-frequency 40, 70
PLL clock MCLK set to 40.01 MHz.
PLL clock VCLK set to 70.00 MHz.
```

arc-set-clock-source—Set the board clock sources

Usage: arc-set-clock-source gclk [<N>] = <SOURCE>

gclks = <SOURCE> , { <SOURCE> }

harvard

where <N> is in the range 0 .. 3

and <SOURCE> = crystal : use the 48 MHz crystal oscillator

dips : use the crystal divided by the DIP switch settings

highimp : set the clock input to high impedance

host : use the host strobe

mclk : use the PLL MCLK clock

vclk : use the PLL VCLK clock

<freq> : use a PLL clock set to the given frequency

Set the source of one or more of the global FPGA clock pins GCLK0 .. GCLK3. For this command to work, the debugger must be connected to the arcjtag target (see command *target arcjtag*). If *gclk* (with no number) is specified, the source of the main clock for the ARC processor (GCLK3) is set. If *gclks* is specified, the sources of some or all of the clocks may be set. If *harvard* is specified, the PLL clocks are configured to drive Harvard Ctl_Clk.

Examples:

```
(gdb) arc-set-clock-source gclk1=host
Attempting to set clocks.
GCLK0   <<   High Impedance
GCLK1   <<   Host Strobe
GCLK2   <<   Host Strobe
GCLK3   <<   Crystal With Division (+Harvard)
(gdb) arc-set-clock-source gclks=45,dips
PLL clock MCLK set to 45.00 MHz.
Attempting to set clocks.
GCLK0   <<   PLL MCLK @ 45.00 MHz
GCLK1   <<   Crystal With Division
GCLK2   <<   Host Strobe
GCLK3   <<   Crystal With Division (+Harvard)
```

arc-reset-board—Reset the board

Usage: arc-reset-board

Reset the board. This command works even if the debugger is not connected to the arcjtag target.

Example:

```
(gdb) arc-reset-board
```

xISS Tracing Commands

These commands are available only in the arc-elf32 build of the debugger, for the arcxiss target.

Two different kind of instruction tracing are supported:

1) a minimal trace which records only the address of each instruction which is executed;
2) a more detailed trace which includes the ordinal number of the instruction in the trace, condition code settings, instruction address, instruction disassembly, and the values of source and destination operands; e.g.

```
{I} <0x0000003E> Z=1,N=0,C=0,V=0,k=1  [00000110]   j_s     [blink]
Dest=0000016c,Src2=0000016c
{I} <0x0000003F> Z=1,N=0,C=0,V=0,k=1  [0000016c]   mov_s   r0,r13
Dest=00000000,Src2=00000001
{I} <0x00000040> Z=1,N=0,C=0,V=0,k=1  [0000016e]   mov_s   r1,r14
Dest=00000000,Src2=00020298
{I} <0x00000041> Z=1,N=0,C=0,V=0,k=1  [00000170]   bl.d    0xb0
Dest=00000220,
```

The minimal trace is recorded in an internal buffer whose size may be determined by the user; this buffer wraps around, so that only the addresses of the most recently executed instructions are held. This data may then be saved to a file for subsequent inspection. The trace data is written in an encoding which gives a reduction in size of approximately 80% compared to the raw data; to read this file, a decoding utility is provided (see the README.arcgdb file for instructions upon how to build this utility).

set arc-xiss-trace—Set whether to trace instruction execution

Usage: set arc-xiss-trace on | off

This controls whether the xISS outputs a detailed trace of each instruction in the program being debugged as it is executed. The trace output will by default be directed to the console; it may be interrupted at any time by the use of Ctrl-C.

Example:

```
(gdb) set arc-xiss-trace on
```

set arc-xiss-trace-file—Set the output file for instruction tracing

Usage: set arc-xiss-trace-file [<FILE>]

This will cause the xISS to write the detailed instruction trace information to the specified file. If <FILE> is different from the last file thus specified, that file will be closed; if it is the same, this command will have no effect. If <FILE> is omitted, the current trace file (if any) will be closed, and the trace output will be redirected to the console.

Example:

```
(gdb) set arc-xiss-trace-file  arc700.txt
```

set arc-xiss-trace-buffer-size—Set the size of the instruction address tracing buffer

Usage: set arc-xiss-trace-buffer-size <INTEGER>

This sets the size of the buffer used to hold the instruction address trace. If the given size is zero, the current contents of the buffer are discarded, and instruction address tracing is disabled; if the size is greater than zero, the buffer is re-sized as necessary, and instruction address tracing is enabled. If the buffer is made smaller by the re-sizing, this may result in some of the stored instruction addresses being lost; in this case, the oldest addresses in the buffer will be discarded.

Example:

```
(gdb) set arc-xiss-trace-buffer-size   100000
```

arc-xiss-empty-trace-buffer—Empty the instruction address tracing buffer

Usage: arc-xiss-empty-trace-buffer

This will discard the current contents of the instruction address trace buffer. The buffer's size is not altered.

Example:

```
(gdb) arc-xiss-empty-trace-buffer
```

arc-xiss-save-trace—Save the contents of the instruction address tracing buffer to a file

Usage: arc-xiss-save-trace <FILE>

This will write the instruction address trace information to the specified file. The buffer's contents are not altered.

Example:

```
(gdb) arc-xiss-save-trace   arc700_demo.trc
3546 instructions in trace buffer
```

arc-xiss-list-trace—List the instructions executed from an instruction address trace

Usage: arc-xiss-list-trace [FROM=<VALUE>] [TO=<VALUE>] [<FILE>]

This will disassemble and display the instructions corresponding to the address trace information. If <FILE> is specified, the instruction address trace information is read from that file; otherwise, it is read from the tracing buffer. The FROM and TO parameters, which may be any positive integer value, may be used to select a range of instructions to be displayed.

Example:

```
(gdb) arc-xiss-list-trace   from=100  to=110  arc700_demo.trc
<0x00000064>       0x00002374          c800     ld_s       r0,[gp,0]
<0x00000065>       0x00002376      1404341b     ld.ab      fp,[sp,4]
<0x00000066>       0x0000237A          c0a1     add_s      sp,sp,4
<0x00000067>       0x0000237C      02ddff0f     b          0x658 <_vfprintf_r>
<0x00000068>       0x00000658          c0f1     push_s     blink
<0x00000069>       0x0000065A      1cc8b348     st.a       r13,[sp,-56]
<0x0000006A>       0x0000065E          c641     st_s       r14,[sp,4]
<0x0000006B>       0x00000660          c742     st_s       r15,[sp,8]
<0x0000006C>       0x00000662      1c0c3400     st         r16,[sp,12]
<0x0000006D>       0x00000666      1c103440     st         r17,[sp,16]
```

```
<0x0000006E>        0x0000066A      1c143480      st          r18,[sp,20]
```

Status Commands

info arc-bcr-registers —Show Build Configuration Registers in the ARC processor variant (arc-elf32 build only)

Usage: info arc-bcr-registers

Example:

```
(gdb) info arc-bcr-registers
[61] DCCM_BASE_BUILD  : 0x00
[62] CRC_BASE_BUILD   : 0x00
[63] BTA_LINK_BUILD   : 0x00
[64] DVBF_BUILD       : 0x00
[65] TEL_INSTR_BUILD  : 0x02
[67] MEMSUBSYS        : 0x01
[68] VECBASE_AC_BUILD : 0x01
[69] P_BASE_ADDRESS   : 0xfc0001
[6f] MMU_BUILD        : 0x00
[70] ARCANGEL_BUILD   : 0x00
[72] D_CACHE_BUILD    : 0x12201
[73] MADI_BUILD       : 0x00
[74] DCCM_BUILD       : 0x00
[75] TIMER_BUILD      : 0x303
[76] AP_BUILD         : 0x00
[77] ICACHE_BUILD     : 0x22401
[78] ICCM_BUILD       : 0x00
[79] DSPRAM_BUILD     : 0x00
[7a] MAC_BUILD        : 0x00
[7b] MULTIPLY_BUILD   : 0x00
[7c] SWAP_BUILD       : 0x00
[7d] NORM_BUILD       : 0x00
[7e] MINMAX_BUILD     : 0x00
[7f] BARREL_BUILD     : 0x02
```

info arc-configuration—Show ARC configuration information (arcjtag target only)

Usage: info arc-configuration

Example:

```
(gdb) info arc-configuration
ARC600
```

info arc-fpga—Check whether the board's FPGA has been configured

Usage: info arc-fpga

Check whether the board's FPGA has been configured (by "blasting" it). This command works even if the debugger is not connected to the arcjtag target.

Example:

```
(gdb) info arc-fpga
FPGA is not configured.
```

info arc-clock-settings—Show ARC clock settings (arcjtag target only)

Usage: info arc-clock-settings

Example:

```
(gdb) info arc-clock-settings
PLL clock MCLK : 85.00 MHz.
PLL clock VCLK : 45.00 MHz.
GCLK0   <<   High Impedance
GCLK1   <<   Crystal With Division
GCLK2   <<   Host Strobe
GCLK3   <<   PLL MCLK @ 85.00 MHz
```

show arc-debug—Show whether to print ARC debug messages

Usage: show arc-debug

Example:

```
(gdb) show arc-debug
Whether to print ARC debug messages is off.
```

show arcjtag-debug-statemachine—Show whether to print JTAG state machine debug messages

Usage: show arcjtag-debug-statemachine

Example:

```
(gdb) show arcjtag-debug-statemachine
Whether to print JTAG state machine debug messages is off.
```

show arcjtag-retry-count—Show the number of attempts to be made for a JTAG operation

Usage: show arcjtag-retry-count

Example:

```
(gdb) show arcjtag-retry-count
The number of attempts to be made for a JTAG operation is 50.
```

GUI debug Interfaces and Front-Ends

ARC GDB includes the integrated GUI interface “insight”. The GUI consists of several separate windows, which use standard elements like buttons, scrollbars, entry boxes and such to create a fairly easy-to-use interface. Each window has a distinct content and purpose, and can be enabled or disabled individually. The windows contain things like the current source file, a disassembly of the current function, text commands (for things that are not accessible via a button), and so forth.

GDB supports a number of industry standard front-ends like the Eclipse CDT plug-in (C Development Toolkit) and DDD (the Data Display Debugger) front-ends.

arc-linux-uclibc-gdb can be used with the Eclipse CDT plug-in. Please refer to the Using Eclipse CDT doc for more details. The DDD debugger front-end can be used with arc-linux-uclibc-gdb as well as arc-elf32-gdb.

In addition, arc-linux-uclibc-gdbtui is a standard TCL-based full screen command line front-end available for GDB. This tool is built by default during gdb build (using the –enable-tui option).

Chapter 4 — References

The following information is available.

GDB online documentation: a complete GDB reference manual. It contains documentation for all GDB commands.

- http://sourceware.org/gdb/current/onlinedocs/gdb_toc.html
- http://sourceware.org/gdb/documentation/

DDD front-end to the GDB debugger online documentation

http://www.gnu.org/manual/ddd/

Eclipse online documentation (includes CDT etc.)

http://help.eclipse.org/

Insight GDB GUI interface online documentation.

http://sourceware.org/insight/faq.php