GDB Configuration Guide

This document will describe the steps necessay to use gdb to debug an ADL model.

Author: Brian Kahne, Lu Hao
Contact: bkahne@freescale.com, lu.hao@freescale.com

Table of Contents

1 Overview

1.1 Basic Operation

The primary option needed to run the ADL model so that it can be connected to GDB is the `-g` option. This places the ADL model into the debug mode; the model will attempt to attach to a port and wait for GDB commands. The default port is 0, which means that an unused port will be selected. The model will then print the port that was selected. If the model is started with the `--jit` option, and the model was built with dynamic-binary-translation support, then during the debugging session, the high-speed translation mechanism will be used whenever the *continue* command is issued.

If the `--port-file=<path>` or `--pf=<path>` option is used, the model will create a file of the specified name and write a GDB target command to that file, containing the port number it is using. This file can then be sourced by GDB using the `source` command. Alternatively, an explicit port may be specified by using the `--port=<num>` or `--p=<num>` option.

GDB may also communicate with an ADL model using standard input/output. This is often the preferred choice, since it results in higher performance. However, it is not usable if the workload running on the ADL model is using system calls and displaying to standard out. To use this feature, use the `--gdb-stdio` option.

UDP packets may also be used by specifying `-gdb-udp`. Use of UDP packets is technically unreliable over a network but works fine if the host and simulator are running on the same machine. Performance is also quite a bit better than for TCP packets.

1.2 Multi-Processor Support

The preferred method for handling multi-core simulations is to connect a single instance of GDB to the simulator. Each core will appear as a thread to GDB. Using a GUI interface, such as Emac's Gud mode, you can then easily switch between threads, set thread-specific breakpoints, etc. This method works for decode-cache models and the multi-threaded simulation kernel.

Is is also possible to connect multiple debuggers, each to their own core. This is only supported using TCP or UDP protocols for non-MT interpreted models. For an MP situation, the `port` and `port-file` options take a comma-separated list, where each element in the list is of the form `<path>=<value>`. For `port`, the value is a port number, or 0 to select an available port. For `port-file`, the value is the name of a file to which to write connection data. Each path is a colon-delimited path identifying a core in the simulation. Cores and systems are valid targets; context-based threads may not be directly targeted. In the case of a system, all constituent cores and context-based threads will be registered as threads to the debugger. In the case of cores, any context-based threads will be registered as threads.

For example, given an ADL simulator with a top-level system named `MP` which contains two cores `p0` and `p1`, the command-line options to use the UDP protocol would be:

```
... --gdb-udp --port-file=MP:p0=port0,MP:p1=port1
```

Since no `port` option was specified, two available ports would be allocated. The connection data would be written to the files `port0` and `port1`.

1.3 Hardware/Software Debug Support

Hardware/Software debug is a feature under development. It allows user to do post-processing Hardware/Software debug using a trace file. Currently it supports TARMAC trace for Cortex-M0+. In the future it will support for more architectures including PowerPC.

2 Installing GDB

The standard installation procedures for GDB works with the ADL interface. To generate a version of GDB to work with a 32 or 64-bit PowerPC, configure GDB with `--target=ppc-elf`. For example:

```
./configure --target=ppc-elf --prefix=<install_path>
```

You can also configure GDB for other supported targets. All you need is changing the `--target=<target>` argument. All supported targets can be found in:

```
gdb-x.x/bfd/config.bfd
```

(file location may vary for different gdb versions).

A more convenient way is to find the already compiled binary file for your target, and just download and use it.

3 Setup GDB configuration File

An ISS's configuration file (specified using the `--confg=<file>` option) specifies a register ordering that maps the model's registers to the register model used by GDB. It is written in TCL. For example, the following code fragment creates a 32-bit Power register mapping:

```
set regs ""
for {set i 0} {$i < 32} {incr i} {
      lappend regs "GPR$i:32";
}
for {set i 0} {$i < 32} {incr i} {
      lappend regs 64;
}
lappend regs NIA:32 MSR:32 CR:32 LR:32 CTR:32 XER:32 32

eval setregmap powerpc $regs
```

The register names such as `NIA`, `MSR` are register names specified in ADL model. the number 32 or 64 following the register names (and sometimes the colon) are the register width accepted in GDB. Following the same format, users can create register mappings for other architectures.

You must ensure that the register model in the configuration script matches GDB's register model exactly. One way to figure out GDB's register model is to go to:

```
gdb-x.x/gdb/features/
```

first open `Makefile`, find in `WHICH` field the one best describes the target you have, then find the corresponding xml files and see its register list with name and size.

Another way is to start GDB with the machine interpreter, launch a simulation, then query for the register names and values:

```
./gdb --interpreter=mi
(gdb) source iss-macros.gdb
(gdb) startsimarch  <model_path> <elf_file_name> <architecture>
(gdb) -data-list-register-names
(gdb) -data-list-register-values r
```

The register values are printed in raw hex format, with one hex digit per nibble.

GDB may recognize different register size from the one in ADL model, this may require size adjustments to the registers. For example, if register F00 is a 64-bit register in ADL model, but GDB thinks that it is a 32-bit register, then the corresponding entry in the register map should be:

```
FOO:32
```

This will cause the ADL model to only send the least-significant 32 bits of F00 to GDB.

Please refer to the ADL CLI reference manual for details on the *.ttc* file format.

4 Connecting GDB To ADL

4.1 Socket-based Approach

Launch the simulator from the command-line:

```
<model> <elf_file> --config=<mapping script> --port-file=.gdbport
```

Start GDB from the command-line in another window:

```
gdb <elf_file>
```

Set the architecture:

```
(gdb) set archi <architecture>
```

The above command is required for 64-bit PowerPC because GDB contains a bug which erroneously sets the architecture to `e500` for any BookE ELF file. In such a case, you would issue the following command:

```
(gdb) set archi powerpc:common64
```

Source the port file in order to connect to the simulator:

```
(gdb) so .gdbport
```

Run your program:

```
(gdb) break main
(gdb) continue
```

The use of `extended-remote` means that the simulation can be reset. In other words, issuing the `run` command will reset the model and re-run the program from the beginning.

4.2 Standard I/O Approach (currently disabled)

Start GDB from the command-line:

```
gdb <elf_file>
```

Set the architecture:

```
(gdb) set archi <architecture>
```

The above command is required for 64-bit PowerPC because GDB contains a bug which erroneously sets the architecture to `e500` for any BookE ELF file. In such a case, you would issue the following command:

```
(gdb) set archi powerpc:common64
```

Launch the simulator:

```
(gdb) target extended-remote | <model> <elf_file> -g -gdb-stdio --config=<mapping file>
```

Run your program:

```
(gdb) break main
(gdb) continue
```

4.3 Hardware/Software debug mode

Launch the simulator from the command-line:

```
<model> -trace-run -if=tarmac <tarmac_file> -o=<output_dat_file> -g --fep --sce --config=<mapping script
```

Start GDB from the command-line in another window (currently only support arm-none-eabi-gdb):

```
arm-none-eabi-gdb <elf_file>
```

Source the port file in order to connect to the simulator:

```
(gdb) so .gdbport
```

5 Helper Macros

The ADL distribution contains a few helper macros for automating the process of launching a simulator within GDB. The macros are contained in the file `share/iss-macros.gdb`:

- *startsim model elf_file*: Launch a simulator. For example, assuming that the user has the line `so <path-to-adl>/share/iss-macros.gdb` in a `.gdbinit` file in the current directory:

```
gdb
(gdb) startsim ./altair_r1_ut sieve.elf
(gdb) break main
(gdb) continue
```

  This will launch the ADL model named `altair_r1_ut` with the ELF file `sieve.elf`.
- *startsimarch model elf_file architecture*: Launch a simulator and specify the target architecture. For example:

```
gdb
(gdb) startsimarch ./altair_r1_ut sieve.elf powerpc:common64
(gdb) break main
(gdb) continue
```

6 Remote Serial Protocol Notes

The current implementation supports the extended run-time control extensions, so the model may be reset so that a program can be restarted.

By default, the addresses of memory accesses (the `m` and `M` RSP commands) are considered to be effective addresses. The address is translated using the MMU and memory is accessed from the top of the memory hierarchy so that any dirty data in a cache will be seen.

Two additional RSP query commands are implemented so that a client application may access memory directly without involving the MMU or any caches. The query for reading memory is `adl.m` and has the following form:

`qadl.m` **addr**, **length**

To write memory:

`qadl.M` **addr**, **length**

Generated on: 2018/02/28 15:14:26 MST.