GDB Server for Soclib

The GdbServer class implements the soclib software debugger.

Overview

The GdbServer is able to manage all processors in a soclib platform. Once added to the platform netlist, it listens for TCP connection from ?Gnu GDB clients. Once connected, clients can be used to freeze, run, step every processor in the platform, add breakpoints, catch exceptions and dump registers and memory content.

Implementation

The GdbServer contains no processor specific code and can be used to manage any Soclib processor model using the generic Iss interface. It is implemented as an Iss wrapper class. When the GdbServer is in use, it intercepts all events between the processor Iss model and the Soclib platform. This enables the GdbServer to access platform ressources as viewed from the processor without modifing platform components or processor model source code. The GdbServer is able to freeze the nested processor model while the platform is still running.

Usage

Adding GdbServer support to your platform

Adding the GdbServer to your topcell is easy. First include the header:

```
#include "gdbserver.h"
```

Then replace processor instantiation:

```
   // Without GdbServer
// soclib::caba::IssWrapper<soclib::common::MipsElIss> cpu0("cpu0", 0);
   // With GdbServer
   soclib::caba::IssWrapper<soclib::common::GdbServer<soclib::common::MipsElIss> > cpu0("cpu0",
```

Do not forget to update the platform description file:

```
Uses('iss_wrapper', iss_t = 'common:gdb_iss', gdb_iss_t = 'common:mipsel'),
```

Connecting with a GDB client

When the simulation is running, the GDB Server listen for client connections on TCP port 2346.

```
$ ./system.x mutekh/kernel-soclib-mips.out
```

Its easy to connect to the simulation with a suitable gdb client:

```
$ mipsel-unknown-elf-gdb mutekh/kernel-soclib-mips.out
GNU gdb 6.7
Copyright (C) 2007 Free Software Foundation, Inc.
(gdb) target remote localhost:2346
Remote debugging using localhost:2346
0xe010cef4 in cpu_atomic_bit_waitset (a=0x602002cc, n=<error type>) at /home/diaxen/projets/mute
99      {
```

The processors are now frozen. Each processor is seen as a thread by the GDB client:

```
(gdb) info threads
  4 Thread 4 (Processor mips_iss3)  0xe010ceec in cpu_atomic_bit_waitset (a=0x602002cc, n=<error
    at /home/diaxen/projets/mutekh/cpu/mips/include/cpu/hexo/atomic.h:99
  3 Thread 3 (Processor mips_iss2)  0xe010ce64 in lock_spin (lock=0x602002cc) at /home/diaxen/pr
  2 Thread 2 (Processor mips_iss1)  0xe010d110 in gpct_lock_HEXO_SPIN_unlock (lock=0x602061e8) a
* 1 Thread 1 (Processor mips_iss0)  0xe010cef4 in cpu_atomic_bit_waitset (a=0x602002cc, n=<error
    at /home/diaxen/projets/mutekh/cpu/mips/include/cpu/hexo/atomic.h:99
```

Classical GDB debugging session takes place. Here is a register dump of the processor 0 (thread 1):

```
(gdb) info registers
          zero       at       v0       v1       a0       a1       a2       a3
 R0   00000000 0000ff00 00000001 00000000 60200338 00000001 00000000 e010e74c
            t0       t1       t2       t3       t4       t5       t6       t7
 R8   e010ef54 00000000 00000000 00000000 00000000 00000000 00000000 602021dc
            s0       s1       s2       s3       s4       s5       s6       s7
 R16  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
            t8       t9       k0       k1       gp       sp       s8       ra
 R24  00000000 00000000 00000000 602007fc 60207ff0 60205ce8 60205ce8 e0101134
            sr       lo       hi      bad    cause       pc
      0000ff00 00000000 00000000 00000000 00000000 e010117c
           fsr      fir
      00000000 00000000
```

More informations on using the GDB client can be found on the ?The GNU Project Debugger home page.