Instruction Set Simulators and gdb Debugging

Simon Southwell

December 2024

Preface

This document is from an article written in December 2024 and published on LinkedIn, covering how an Instruction Set Simulator (ISS) can have its running code debugged with gdb.

Simon Southwell
Cambridge, UK
December 2024

Contents

Introduction

I have previously discussed instruction set simulators (ISS) in various contexts, including a discussion of how they might be constructed as part of modelling SoC systems in C/C++ and also integrated into a co-simulation environments, such as that available with OSVVM. The advantage of using ISS models of processor cores is that they can run very much faster than modelling this in, say, HDL, particularly if in a stand-alone C/C++ model, but also when integrated with a logic simulator via co-simulation elements, such as the *VProc* virtual processor, to drive RTL modules as design-under-test components.

However, running software on an ISS, whether test code or application code, is only really useful if there is a means to debug it. So in this article I want to discuss how we can connect the *GNU Debugger* (gdb) to an ISS model using its remote target and remote serial protocol features. This will require setting up some TCP/IP server code with the ISS which gdb can attach to, decoding the required received commands, generating the responses and accessing and controlling the ISS as appropriate. We shall look all at these aspects in detail. As working examples I will be referencing the rv32 RISC-V ISS model's and (to a lesser extent) the mico32 LatticeMico32 ISS's debug features and see how they work. This also maps to real world debugging of code on hardware platforms, and so is useful to know in a wider scope of SoC system development.

The GDB Remote Debug Interface

If you've ever used gdb to debug an executable program that you've written for your computer, such as a Linux workstation or PC, you'll know that there is a command line interface and various commands are typed in to do things like set breakpoints, run the program being debugged, inspect variable state, step through code, etc. When used like this, gdb runs the program from within itself and takes control without the need for any formal communication link between the program and gdb. When running a program with gdb this way, it has access to all the process space that is used by the executing program being debugged and can peek and poke directly.

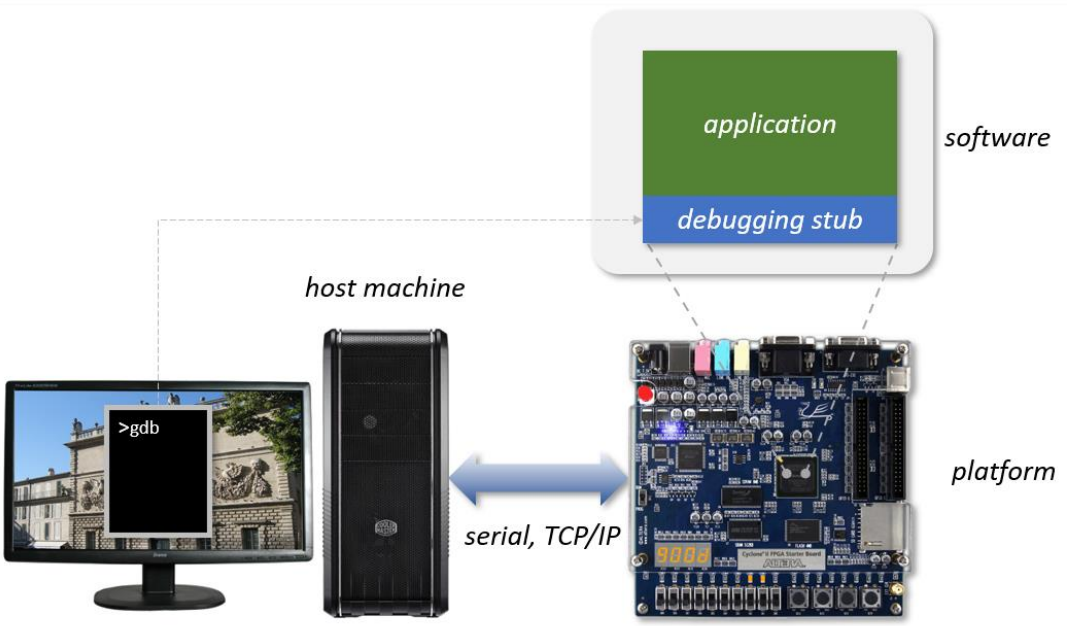
In a separate system, such as SoC on an embedded platform board, running gdb may not be an option. (Some embedded systems may have, for example, a Linux environment and enough memory to make running gdb locally possible, but many don't.) In this case gdb is run on a host machine and a communication link is setup between this host and the *target* system. This can be in the form of a serial

connection, a TCP/IP link or even over JTAG using, for example, OpenOCD via USB. Whatever the connection method, the gdb running on the host will send commands over the connection and will get responses back.

There needs to be something at the target end to receive these commands and generate the responses, and to control the program that will be run on the platform. It should noted that the host end gdb program will take care of all the symbol information, source code lines, variables and their addresses, etc., so that the program to be run on the target can be compiled without all this information so that it is in the form that will be used on the platform product. There are a couple of ways that real platforms can have something to receive and process the gdb commands.

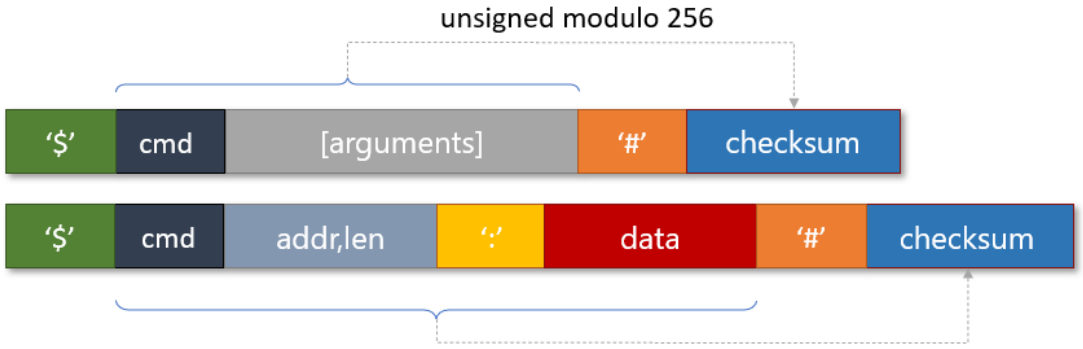
The method first is to use a "Remote Stub". This is a set of subroutines that understand the gdb commands and generate the responses whilst controlling the embedded program. The stub code (known collectively as a "debugging stub") is linked with the embedded software into a single executable. The debugging stub will be specific to the target processor architecture, and there are standard stubs available for architectures such as i368, and other hard- or softcore processor providers might make a debugging stub available with the IP.

A second method for hardware platforms is to use the `gdbserver` program. This is more useful when using an embedded system that can run Linux, as mentioned before, and `gdbserver` is run as a separate program within that operating system. It does a lot of what gdb itself can do but is a smaller program and can be controlled remotely. The diagram below summarises the situations discussed.



Serial Protocol

So, what is the protocol that gdb uses to send commands and what are these commands? In general, gdb will send commands in the form of packets of characters that start with a '$' character followed by a command type of one or more bytes, arguments of zero or more bytes and a termination 'end of packet' (EOP), which is a '#' character. Commands that have data will have a ':' character after the arguments, and then the data itself. All packets are then sent with a two digit checksum. Some typical packets are shown below:

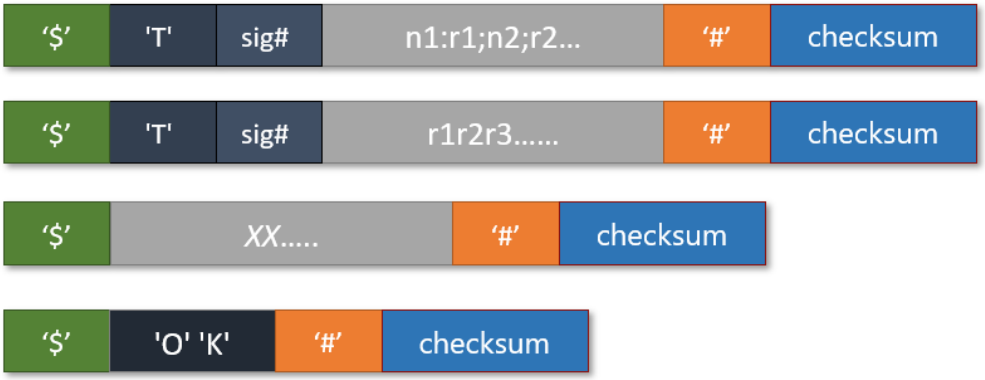


The minimum set of commands that have to be supported are surprisingly few. Each of these has a single byte character for the command, and these are listed below:

cmd	params	Description
'?'	-	Query reason for halt
'g'	-	Read general registers
'G'	*XX*...	Write general registers
'm'	addr,length	Read 'length' addressable units starting from 'addr'
'M'	addr,length:*XX*....	Write 'length' addressable units starting from 'addr'
'c'	[addr]	Continue from current address or specified 'addr'
's'	[addr]	Single step from current address or specified 'addr'

The 'm' and 'M' commands have an argument string which is an address as bytes formatted as two hexadecimal characters, a ',' character and a length formatted in the same way as the address. The 'c' and 's' commands will have their address formatted like this as well, if it is present. The memory and register write commands ('M' and 'G'), the '*XX*...' data that follows is also sent as bytes formatted as two hexadecimal characters, so will be double the specified length in size.

The packets sent from gdb must be acknowledged that they were successfully received (or not) and this is done with either a '+' character, for acknowledge, or a '-' character for not acknowledge. If the commands require a response, then these must be formatted as packets as for the received commands, with a '$' followed by a packet, a '#' EOP, and a checksum. Depending on the command, different responses are required. Some relevant ones are shown below.



The first of these responses are for commands that exit due to some sort of stop condition. From the minimum set described, these are the '?' (query), 'c' (continue) and 's' (step) commands. There is a lead 'T' character, followed by a two by hex number which is the signal number that generated the stop, as per the Linux Signals. In this context, these signals might be `SIGTRAP` (trace/breakpoint trap), `SIGHUP` (hangup/death), `SIGTERM` (termination), or `SIGILL` (illegal instruction). Following this is a list of registers and their values, with the register number as a two character hex number, a ':' character' and the value of the register as a set of two character hex numbers to make up the width of the register (4 bytes in a 32-bit processor). Between each register number/value pair is a ';' character. The data here is processor architecture specific. For example, RISC-V RV32I processors will have 32 registers (x0 to x31) of 32-bits each, and a program counter, also 32-bits wide (which has an index of 32 for this purpose), and this is the data to be sent.

The second format is similar to the first, but this is just a series of register values, it being understood for the architecture what data will be sent (values for thirty three 32-bit registers). This is used for commands that read registers, as the 'g' command does from our minimal set.

The third format is format is for memory read commands, such a 'M', and the *XX* data is a set of two character hex bytes that is the data in memory from the parameter specified start address and length.

The last format shown is for responses to commands that don't return any data (which is the rest of the commands from the minimal set in the table from before.) This is basically the string "OK".

Other Useful Commands

As well as the seven commands in the minimal implementation set, there are some other commands that are useful to have, a couple of which are applicable when debugging code in ROM, as we'll see later. The table below shows these commands that will be relevant to the discussion of debugging code on an ISS.

cmd	params	Description
'X'	addr,length:*BBB…*	Write 'length' addressable units starting from 'addr'
'p'	n	Read register n
'P'	n=r	Write register n with r
'D'	-	Detach from remote system
'k'	-	Kill request
'z'	type,addr,kind	Insert a type break-/watchpoint at addr of kind
'Z'	type,addr,kind	Insert a type break-/watchpoint at addr of kind

These additional commands extend the capabilities to be more efficient, or to take advantage of features available on the target system, The 'X' command is the same as the 'M' command, to write data to memory, but the data following the ':' is now a set of binary bytes, making the transfer more efficient. The length argument now does indicate the size of the data portion of the packet.

The 'p' and 'P' commands are for writing individual registers, where the n argument is a two character hex number as a register index. For 'P' this is followed by '=' and a register value encoded as a set of two character hex numbers to make up the register width value. The 'D' and 'k' commands are used for ending a session, or terminating a running session, cleanly.

The 'z' and 'Z' commands are for setting or clearing, respectively, breakpoints or watchpoints in a way that isn't the standard method (more on this in a bit). The addr argument specifies which address the break- or watchpoint is for, and the kind argument is target specific, but usually specifies things like the byte size of the break-/watchpoint. For our purposes, the type values we're interested in are values '1' to '4' for commands `z1` to `z4`, and `Z1` to `Z4`. The `z1` and `Z1` commands are for setting or clearing a hardware breakpoint. That is, the processor has logic blocks that can be written with an address and monitor for instruction reads from addresses that

match its set value to then stop the processor and flag it has done so. Similarly, `z2` to `z4` and `Z2` to `Z4` are used to set and clear hardware watchpoints of various types:

- `z2`/`Z2`: break on matching data memory write access
- `z3`/`Z3`: break on matching data memory read access
- `z4`/`Z4`: break on matching data memory access of any type

The LatticeMico32 32-bit softcore has just such hardware breakpoint and watchpoint features, with four breakpoint and four watchpoint registers. The mico32 instruction set simulator models these and so its built-in debugging stub supports the `z` and `Z` commands.

How the Commands are Used

We now have a set of commands, a little bigger than the minimum required to do debug with, but what do they actually do when received? The commands fall into these rough categories:

- Read register(s): '?', 'g' and 'p'
- Write register(s): 'G' and 'P'
- Read memory: 'M'
- Write memory: 'm' and 'X'
- Execution control: 's' and 'c'
- Kill program: 'k'
- Detach from remote target: 'D'
- Set/clear h/w break- and watchpoints: 'z' and 'Z'

The read and write register commands will need access to the processor core's register state, but software (at assembly level) has just this and the debugging stub program can fetch register values to return in a response or set register values before the application program is reactivated. The reading and writing to memory just require loads or stores in the memory mapped address space. This allows the actual `gdb` program to inspect and modify variables of the running program. The debug information in the program executable (when compiled with `gcc -g`, for example) will contain information about where in memory each variable is stored, whether on the stack or in the heap. When a command at the `gdb` command prompt is used to print a variable's value (e.g., `gdb>p /x var1`) then this gets turned into a read memory command over the link with the debugging stub code, and similarly when setting a variable's name. The `gdb` program will know where each line of code is also in memory, and knowing the PC register's value, with a register read can map this to

source code to display. The application program can be restarted from a different place by updating the PC with a new value via a register write.

The 'k' kill command requests the debugging code to artificially stop the application execution, wherever it is in its program, to terminate and return. This can be done at the gdb command prompt with `gdb>kill`. Then the actual link with the system (over TCP/IP or serial) can be terminated with 'D' to detach. This will close the link so, in the case of TCP/IP, the socket will be closed. At the gdb command prompt this can be done with `gdb>detach`.

The 's' and 'c' command control the program execution. The stepping of a single instruction may use hardware support that allows the processor to execute one instruction and then halt. Continuing allows the program to execute freely until a breakpoint is encountered (see below)—or forever.

The setting of breakpoints will depend on whether hardware assistance is available in the processor core or not. For code that is running out of RAM, gdb sets a breakpoint by swapping the instruction at the desired address with a 'break' instruction. It will also remember the instruction that was replaced. As an example, RISC-V has an `ebreak` instruction that will cause an exception with a particular 'cause'. The debugging stub can set up the trap handling so that if this occurs it can jump back into the debugging stub code. This is what happens when a breakpoint is set in gdb. When the `ebreak` is encountered the gdb prompt becomes active again waiting for new commands. The breakpoint might be deleted, in which case the original instructions is put back. Even if the program is 'continued', the original instruction is replaced, the program stepped one instruction (with the 's' command), and the `ebreak` reinstalled before continuing.

If code is executed out of ROM, rather than RAM, then the above technique won't work as instructions can't be replaced. This is where the hardware break- and watchpoints are needed. The 'z' and 'Z' commands are then used and do the job by monitoring processor bus activity as discussed. The method is limited by how many hardware break- and watchpoints are supported (four each in the case of the LatticeMico32), but each will be used as breakpoints are set until full, when an error is returned if attempting to set another.

Debugging on an ISS

We now have a set of commands we can use on an ISS for debugging, decoding them, affecting ISS execution and generating responses. We'll need some debugging stub code, a means to connect this with the remote gdb, and a means of connecting

to the stub with the ISS that will assist in implementing the debug commands so let's deal with the ISS first.

ISS debugging features

In the normal execution of a program on an ISS, after some preliminary setup, the ISS is 'run', passing in configuration data to a `run()` method. This method then loops around doing fetch, decode and execute type operations, forever by default. However, within the configuration passed in at run time are various conditions for breaking out of the run loop under certain conditions. Relevant to this discussion are the following:

- After *n* instructions have executed
- Executing a 'break' instruction

There are other halting condition in the rv32 and mico32 ISS models, but these aren't used with gdb debugging so we can safely skip these. With just these two features we already have a means to single step (setting *n* to 1) and for setting a breakpoint (where an `ebreak` instruction is encountered in RISC-V). This last requires the external reading and writing of memory. And also, we need to be able to read and set registers. For the rv32 ISS, it provides the following public methods:

```
uint32_t read_mem  (const uint32_t byte_addr, const int type, bool &fault);

void     write_mem (const uint32_t byte_addr, const uint32_t data,
                    const int type, bool &fault);

rv32i_hart_state rv32_get_cpu_state (int hart_num = 0);

void             rv32_set_cpu_state (rv32i_hart_state &s, int hart_num = 0);
```

The mico32 ISS has almost identical methods to this as well. In addition, the mico32 setting and getting of state also includes the hardware breakpoints and watch points, so these can be updated as for any of the general-purpose registers. The `read_mem` and `write_mem` methods speak for themselves as to their usage. The state set and get methods use a structure with all the state for the processor so that fields it can be fetched, state modified and written back.

Now we have methods to do all the commands that we identified that we want to support and the stub code can decode the commands, uses these methods to fetch or set CPU state and generate the responses. We have now connected the debugging stub to the ISS, but we still need to connect the stub to gdb as a remote target.

TCP/IP Socket Code

Both the rv32 and mico32 ISS models support TCP/IP socket connection, but the mico32 also supports serial COM port connection. However, we will just discuss TCP/IP as this is the most flexible method.

In order to be a remote target for gdb we need to set up some code in the debugging stub as a TCP/IP server, and gdb will then be a client that can connect to this server. The rv32 ISS debugging stub has a function `rv32gdb_connect_skt()`, which creates and opens a TCP/IP socket with a specified port number. Some simplified code is shown below as to how this is done. It has had all error checking removed and removes some initialisation and specifics for Windows.

```
#include <sys/socket.h>

rv32gdb_skt_t rv32gdb_connect_skt (const int portno) {

    // Create an IPv4 socket byte stream
    rv32gdb_skt_t svrskt = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);

    // Create and zeroed a server address structure
    struct sockaddr_in serv_addr;
    bzero ((char *) &serv_addr, sizeof(serv_addr));

    // Configure the server address structure
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port        = htons(portno);

    // Bind the socket to the address
    bind(svrskt, (struct sockaddr *) &serv_addr, sizeof(serv_addr));

    // Advertise the port number
    fprintf(stderr, "RV32GDB: Using TCP port number: %d\n", portno);
    fflush(stderr);

    // Listen for connections (blocking)
    int status = listen(svrskt, MAXBACKLOG);

    // Get a client address structure and length as has to be passed as a pointer to accept()
    struct sockaddr_in cli_addr;
    socklen_t clilen = sizeof(cli_addr);

    // Accept a connection, and get returned handle
    rv32gdb_skt_t cliskt = accept(svrskt, (struct sockaddr *) &cli_addr,  &clilen);

    // No longer need the server side (listening) socket
    close (svrskt);

    // Return the handle to the connected socket. With this handle can
    // use recv()/send() to read and write (or, Linux only, read()/write()).
    return cliskt;
}
```

I won't go into fine detail here, but the function creates an IP socket of an 'internet' type (TCP, UDP…) and a server address structure. The structure is configured to be for 'internet' and the socket is bound to this server address. At this port the socket is

open and the port number used is printed out so the user knows which to use for the gdb target remote command. The server can now be started to listen for a connection. This blocks until a client has attached and the connection is accepted, returning the client socket handle. Then the listening socket can be closed and connection is fully established. The client socket is returned to the calling function and can be used like a file descriptor to read and write bytes for processing incoming commands and generate acknowledge and responses.

And there we have it. The ISS is now capable to be debugged as a remote target. By default, the rv32 and mico32 ISS models run without going through the debugging stub code but they can be configured to enter debug mode, where the top level program is now the debugging stub code instead of directly calling the ISS' run methods.

Both the rv32 and mico32 ISS models have top level wrappers (e.g., `cpurv32i.cpp` for rv32) that allow the models to be run as standalone executables and have various command lines options to configure and control the ISS, using some of the methods already described (as well as some others). The rv32 model is compiled as a stand-alone executable called `rv32` (or `rv32.exe` on Windows) and typing rv32 -h gives the following output:

```
rv32 version 1.1.4. Copyright (c) 2021-2024 Simon Southwell.

Usage: /home/simon/git/riscV/iss/rv32 [-hHeEbdragxcCT][-t <test executable>][-n <num instructions>]
      [-S <start addr>][-A <brk addr>][-D <debug o/p filename>][-p <port num>]
      [-m <num words>][-M <addr>][-u <uart addr>]

   -t specify test executable (default test.exe)
   -n specify number of instructions to run (default 0, i.e. run until unimp)
   -d Enable disassemble mode (default off)
   -r Enable run-time disassemble mode (default off. Overridden by -d)
   -a display ABI register names when disassembling (default x names)
   -C Use cycle count for internal mtime timer (default real-time)
   -T Use external memory mapped timer model (default internal)
   -H Halt on unimplemented instructions (default trap)
   -e Halt on ecall instruction (default trap)
   -E Halt on ebreak instruction (default trap)
   -b Halt at a specific address (default off)
   -A Specify halt address if -b active (default 0x00000040)
   -D Specify file for debug output (default stdout)
   -x Dump x0 to x31 on exit (default no dump)
   -c Dump CSR registers on exit (default no dump)
   -m Dump specified number of 32 bit words from data memory on exit (default 0)
   -M Start byte address of memory dump (default 0x1000)
   -g Enable remote gdb mode (default disabled)
   -p Specify remote GDB port number (default 49152)
   -S Specify start address (default 0)
   -u Specify UART base address (default 0x80000000)
   -h display this help message
```

Relevant to this article are the `-g` and `-P` options which enable gdb debugging and specify a connection port number. The rv32 model has built in memory and real-time

timer, as well as a UART TX port for printing output. A RISC-V compiled executable (in ELF format) can be specified and loaded at the command line, or with gdb mode enabled alternative loaded by the gdb session with a `load` command. A typical session might look like the following:

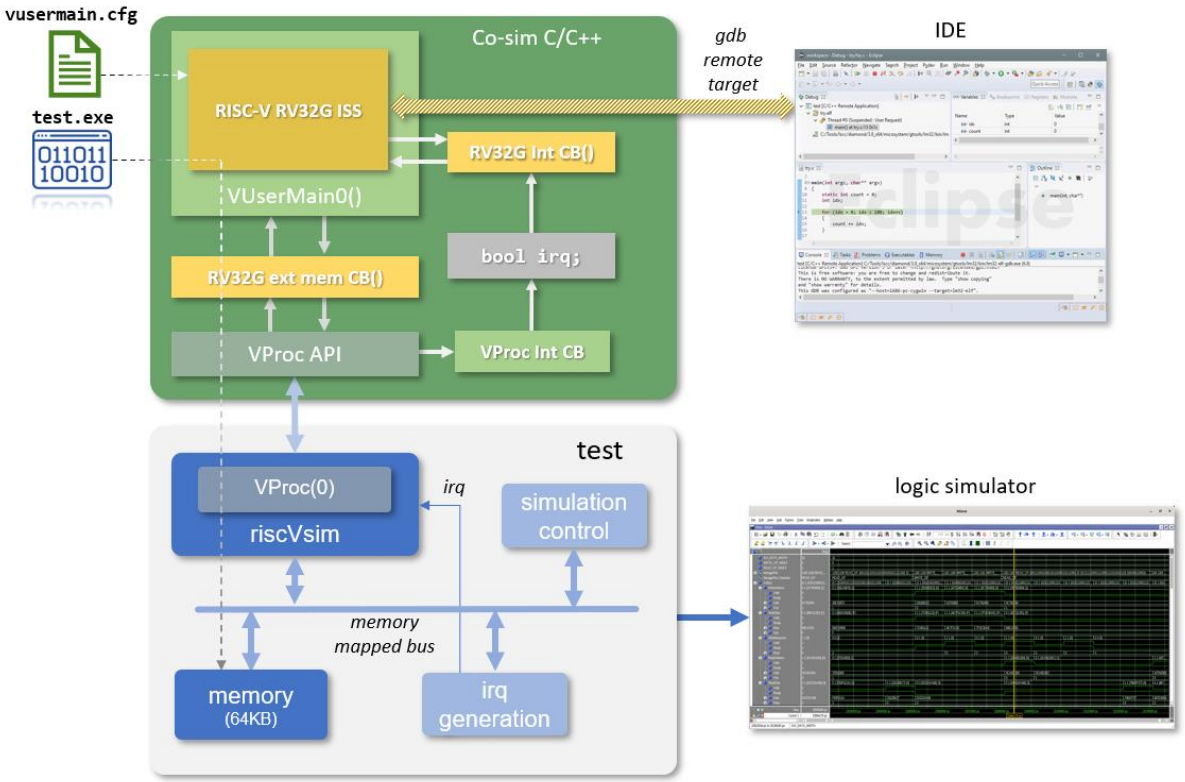
```
GNU gdb (GDB) 9.1
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=i686-w64-mingw32 --target=riscv64-unknown-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from simple.exe...
(gdb) target remote :49152
Remote debugging using :49152
_start () at c:\git\riscv-tests\isa\rv64ui\simple.s:16
16      RVTEST_CODE_BEGIN
(gdb) load
Loading section .text, size 0x1c4 lma 0x0
Loading section .tohost, size 0x48 lma 0x1200
Loading section .got, size 0x28 lma 0x1248
Start address 0x00000000, load size 564
Transfer rate: 23 KB/sec, 141 bytes/write.
(gdb) b *0x100
Breakpoint 1 at 0x100: file c:\git\riscv-tests\isa\rv64ui\simple.s, line 16.
(gdb) c
Continuing.

Breakpoint 1, 0x00000100 in reset_vector () at c:\git\riscv-tests\isa\rv64ui\simple.s:16
16      RVTEST_CODE_BEGIN
(gdb) stepi
0x00000104      16      RVTEST_CODE_BEGIN
(gdb)
0x00000108      16      RVTEST_CODE_BEGIN
(gdb) c
Continuing.

Program received signal SIGTERM, Terminated.
0x00000040 in write_tohost () at c:\git\riscv-tests\isa\rv64ui\simple.s:16
16      RVTEST_CODE_BEGIN
(gdb) detach
Detaching from program: c:\git\riscV\iss\test\simple.exe, Remote target
Ending remote debugging.
[Inferior 1 (Remote target) detached]
(gdb) q
```

This is the final step in connecting gdb to an ISS and debugging software that is running on it. But this also opens up a couple of other possibilities.

Co-simulation

An ISS model is a useful model as part of modelling an SoC system in C/C++, and I have discussed this before in previous articles (see part1 and part2), and the ISS debugging capabilities are, of course, available in such models. However, I also have a means to run a program that's natively compiled for a host machine (a PC or Linux workstation, for example) on a 'virtual processor' that is instantiated in a logic simulation as a processor core with a bus output that can drive read and write transactions on a memory mapped bus from the code that is 'running' on that processor. This is the *VProc* virtual processor, but similar capabilities, based on *VProc* principles, are also available in the OSVVM VHDL verification environment (see this document for details). This delivers a means to co-simulate software with a logic simulation, with all that this brings in terms of writing test code to drive a DUT IP block or system, or running application code to co-design with the logic IP.

The ISS models we have been talking about are just such programs compiled as native code and, as such, can be 'run' on the Virtual Processor. This is documented in the *VProc* manual, but the diagram below summarises this situation.



Here we have the rv32 ISS model which, using its callback methods, can access the *VProc* API to generate memory access transactions and to receive interrupts. The *VProc* HDL component is instantiated in the logic, wrapped in some kind of bus

functional model wrapper to drive the bus protocols required in the test bench. The gdb capabilities (and IDE capabilities, see next section) of the ISS are still available in this environment and so the code running on the ISS can be debugged, *but* with joint debug capability with the IP running in the simulation. The test bench could run an HDL model of the processor, but the ISS model will run much faster and a means would still be needed to debug software running on that HDL model.

Using an IDE

Now that we have the means for gdb to be connected to our ISS we also have a path to using an integrated development environment (IDE), such as Eclipse, that understands gdb and can sit on top of it and drive its command interface. The *VProc* manual documents in detail how to set up Eclipse to drive gdb and hence debug in a co-simulation environment. This setup would be similar for the stand-alone ISS programs as well. Thus we can use an IDE for any of the applications of the ISS models.

Conclusions

We have looked at gdb and how it can connect to a remote system to enable debugging by sending commands to debugging stub code over a communications link such as a TCP/IP socket. The set of minimal commands is quite small, and with a smattering of additional commands a basis for a powerful debug capability was possible.

With this knowledge we saw how we could generate a debugging stub for an instruction set simulator to decode the gdb remote target commands and generate responses, whilst controlling and inspecting state of the ISS. The ISS itself provided a simple set of capabilities to control program flow, set and clear breakpoints, inspect and set register state, and read and write memory. The debugging stub finally had capability to create a server TCP/IP socket that the gdb remote target client was capable of connecting to, and the system was complete to be able to run a program on the ISS and debug it.

The ISS models can be compiled as stand-alone executable in a C/C++ SoC modelling environment, but it was also shown how the ISS can be part of a co-simulation and co-design logic simulation environment with joint software and logic debug capabilities.

The aim here has been to show the (hopefully) simple steps at each stage to build up quite sophisticated verification and development environments, culminating in a co-simulation environment and with IDE debugging capability.