International Journal of Current Research and Modern Education (IJCRME)
ISSN (Online): 2455 - 5428
(www.rdmodernresearch.com) Volume I, Issue II, 2016

A CASE STUDY OF IMPLEMENTING A GDB INTERFACE BETWEEN AN ARM BASED IC SIMULATOR AND GNU DEBUGGER

H. S. Sachin Kumar* & Trisila Devi Nagavi**
Department of Computer Science & Engineering, Sri Jaya Chamarajendra College of Engineering, Mysore, Karnataka

Abstract:

Simulators are basically used in pre-silicon software development, for architectural exploration etc. As the software is becoming increasingly complex, it requires a debugging support to debug the software being run on the core. So the program running on the simulator must be debugged, Keil is the debugger used for this purpose earlier. Because of some drawbacks of Keil debugger and to overcome those drawbacks we are introducing another familiar debugger called as GNU Debugger, popularly known as GDB. This paper covers the replacement of GNU Debugger (GDB) in place of Keil Debugger for the ARM Based IC simulator.

Key Words: GNU Debugger, Debugging & Multi Core Debug

1. Introduction:

In the past, embedded software development and verification was typically performed by running code on a prototype of the hardware platform until the project team was satisfied that a working system had been achieved. This solution is time-consuming, unreliable in terms of quality and hard to use, making it impractical for next generation embedded software development. Similarly to hardware verification 15 years ago, new thinking must be applied if high quality embedded software is to be produced in a timely fashion.

Virtual platforms offer an alternative to hardware prototypes. Software models of the key components in a processor platform are combined to form an executable sub-system. The models must have enough functionality to execute the code correctly, but retain a level of abstraction that provides the performance necessary for rigorous testing.

Typical Virtual Platforms make use of "Instruction Accurate," or IA, processor models together with abstractions of memory blocks and key peripherals. The virtual platforms need to be accurate enough that the software cannot tell it is not running on real hardware, and production binaries of the embedded software should be able to run unmodified. The virtual platforms may be connected to embedded software development tools to enable a comprehensive environment for the verification and analysis of code.

However Keil debugger with AGDI has some disadvantages in terms of cost. Whenever the user wants to use the Keil debugger one has to pay the licence fees. In this paper we proposed an open source GNU Debugger to debug the software being run on the simulator. GNU Debugger popularly known as GDB is a part of GNU compiler suit consortium, which we used here to debug the software program being run on the simulator. Limited features are targeted for this experiment such as Break point handling, Register access, Memory access, etc. This can be achieved by building an interface between GNU Debugger and the Simulator. This paper covers a case of such an implementation.

2. Virtual Platform:

The SoC contained in modern electronic products has evolved dramatically in recent years along three dimensions:

- ✓ **Scale:** The use of embedded software operating on standardized hardware platforms has become increasingly common as the cost of IC hardware production increases. This has driven up the sheer volume of code required for each project, and the effort to produce it.
- ✓ **Complexity:** Multi-core processor architectures have been continually improved, providing the necessary performance and capability to meet modern product requirements. However, coding these new processors has become exponentially more complex than previous generations.
- ✓ **Quality:** Modern electronic product functionality and quality requirements continue to increase dramatically, suggesting a zero tolerance for post-production bugs. In addition, embedded software has become harder to change as a product moves into production.

The tool used for debugging is Keil which is also an IDE for the software development. The SDK is integrated with the Keil in a manner where Keil acts as a front end to the end users and simulator as a virtual target acting as the back-end. The Keil tool works with the target (simulator or hardware) using a debug interface called AGDI. Hence, to support the integration with Keil, the SDK implements the standard AGDI debug interface in order to act as a virtual target to Keil. At the end, as the virtual simulator acts as a target driver to Keil, this enables and provides many advantages for the software development.

As and when the debugger is used from the start until the end of debug session, Keil generates the AGDI interface method calls which finally land-up in the target driver interfaced to the Keil. As in our case, it is the simulator which is acting as a target for Keil. The AGDI interface method calls are received by the simulator are forwarded to a debug module in the virtual system which handles all the debugger related functionality and acts as a central place for debug handling throughout the system.

3. Literature Survey:

MCD:

The MCD API is a simple yet powerful C-interface. It has been captured in a single header file and all API users have to include this header file in their source code. The MCD API is composed of two distinct parts:

- ✓ An API in order to allow tools to access debug targets in a uniform way (Tools API).
- ✓ An API in order to allow the MCD framework to access target components in a standard way (Target API).The MCD API can be classified further as following sub-APIs which have methods/functions to be implemented:
- ✓ Target Connection: MCD works on a server-client model where the target is the server and debugger is the client.
- ✓ Target System Description: These methods are used by the tool to retrieve information about the connected system.
- ✓ Target Run Control: These methods are used to control the simulation run. The simulation can be paused/resumed from the debugger.
- ✓ Trigger Support: It is used to set break-points, watch-point and complex break-points. It also allows customized ones.
- ✓ Trace Support: It is used to get the trace information from the simulation. The traces can later be processed to gather code and data coverage of the embedded software.
- ✓ Memory and Register Accesses: Unified memory and register access mechanism using transaction lists.

- ✓ Communication Channels: Various types of communication channels between debug tool and target system, e.g. for communication with applications running on the targeted core or for configuration of additional analysis resources.

4. Functional Specifications:

As aforementioned, the VP is required to implement the debug API to provide the information required by the debugger. The VP acts as the server and debugger as a client. As shown in the figure, the MCD APIs are implemented by the core model. The core model is the ideal place to implement the debug API as it has access to CPU registers, peripheral registers and memories. The following points describe the implementation in the core:

A. Simulation Control: For debugging the software running on the core, the user would need to have the feature to control the simulation – to pause or break the simulation at any-point, single step into the code and resume the simulation. The MCD API provides method to run, pause and stop the simulation. The VP has to implement the methods and perform the needed operation. The SystemC-2.3[3] library provides in-build support to pause the simulation using a function called "sc_pause". It is a non-blocking function which causes the simulation to pause after completing the tasks in the current execution phase. But it has to be used carefully. If there is no follow-up call to resume the simulation or wait on any event, the simulation will end. The situation is handled running the simulation in an OS (operating system) thread and controlling the pause/resume operations using OS events. The following diagram shows the flow for simulation control.

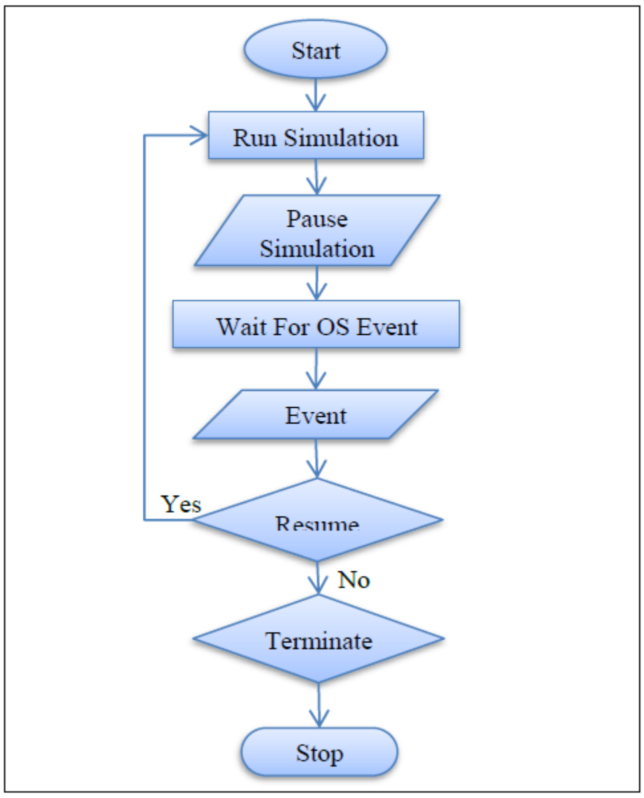

Figure 1: Flow diagram of simulation thread

As shown in the figure 3, the simulation is started in the OS thread. When the simulation receives the pause command by the debugger, the thread waits for the Resume or Terminate event. In the background, when the run command is issued, the implementation in simulator will cause the resume event to be triggered and the thread resumes the simulation. Similarly, when the stop command is issued, the thread will terminate. The same flow is used to implement the "step" command where the simulation is stopped after executing every instruction.

B. Regester Access Controls: The register and memory access methods of MCD API are also handled by the Core model. By using these methods, a debugger can access the memory regions of the simulator and show it in a window in its GUI. The addresses to be read usually depend on the addresses visible in the GUI. The following figures show a memory window and CPU register view snapshots from the Trace32 debugger.

C. Break-Point: The break-point feature allows a user to pause the simulation when it hits a particular statement in the software. It is a much needed feature for the software developers as it allows the simulation to run to the point of interest without stepping through the whole software code. When the software is compiled for a particular tool-chain, like ARM, the resulted binary is a sequence of instructions to be executed by the core. The binary is loaded in the memory (ROM) at the start of the simulation and is read and executed by the core. When a break-point is set on a particular statement in the code, the MCD API requires the corresponding instruction address to be given as input to the core model in the simulator. The core model then monitors the "program counter" and pauses the simulation when its value matches the break-point address. It supports multiple break-points to be set and cleared when needed.

5. Existing System:

As mentioned earlier, Keil has its own debug interface called AGDI through which it can communicate with different targets. Hence, it mandates that the virtual platform (or simulator) now implements the AGDI interface. As it is clear from the abstract the aim is to de-couple the debug interface and the debug implementation of simulation model. Hence, the approach is to support a tool-specific debug interface by developing a translation layer between it and the supported debugger on the SDK. Referring to the figure below, SDK implements the standard MCD interface API's and acts as a MCD client. For interfacing with Keil, the target driver implements the AGDI interface API's of Keil and acts as a client to Keil. Debug calls coming from Keil are received and processed in the AGDI Target Driver component as shown in the figure below. There is no one-to-one mapping between the AGDI interface API's and the MCD interface API's (except in some calls like reset and terminate etc.). Hence, to transfer the information between Keil and SDK there is an AGDI-MCD translator in between which takes care of translating the data structures from AGDI to MCD and vice-versa. The information received on the debug calls from Keil is in the form of AGDI interface data structures. Hence, the AGDI calls are translated using the translator. The translated calls are then passed on to the MCD driver component which is responsible for finally calling the standard MCD interface API's on the simulation model. The simulation model implements the necessary functionality in accordance with the calls received over the MCD interface. Sometimes, the information returned back from the MCD driver after the MCD calls, needs to be translated (for example in case of memory reads and register reads etc.). Hence, the translator is used again which has API's to translate the data back to AGDI from MCD. The MCD driver component is not bound to the AGDI interface (as it always sends and receives in MCD language) hence this component is re-usable to interface with other debuggers (for ex: GDB).

6. Interfacing GDB With Simulator Over MCD Interface (Proposed System):

The GNU debugger (GDB) is increasingly getting used by many embedded software developers who are familiar with Linux environment, as a tool to debug their target code. Within Infineon, several business units use GNU toolchain for embedded software development and debug. The debugger backend, by default, in such cases is GDB.

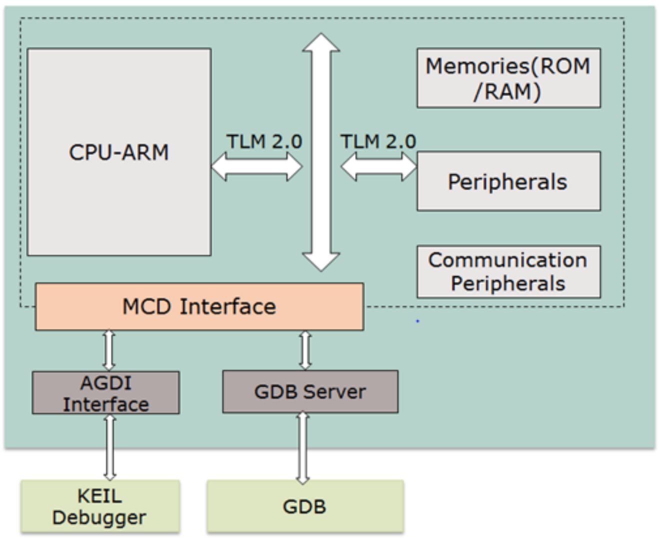

Figure 2: Interfacing with GDB using MCD

GDB provides a mechanism by which user can connect to a remote target (GDB server) over TCP socket though a protocol named Remote Serial Protocol (RSP). This mechanism can be used to connect GDB (client) to the VP through an adapter which acts as GDB server. This adapter listens on a TCP socket for commands from the GDB client. These commands are interpreted according to RSP and then propagated as MCD calls to the VP. The diagram in figure 9 shows the setup with the GDB. The MCD Driver, MCD implementation is re-used from the Keil setup which reduced the setup effort for GDB significantly.

7. Results:

By using open source debugging tool such as GNU Debugger now we can able to debug the program which is running on the simulator. Only limited features are targeted for this experiment such as simulation control, register access, memory access etc.

8. References:

1. MCD API press release Common Modeling Architecture Group project
2. GDB Tutorial a Walkthrough with Examples Simulator, CMSC 212 - Spring 2009 Last modified March 22, 2014
3. Jack Lange Department of Computer Science University of Pittsburgh, “Implementing a GDB Stub in Lightweight Kitten OS”
4. Jeremy Bennett, “How to: GDB Remote Serial Protocol Writing a RSP Server” Application Note 4, Issue 2.