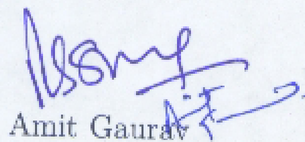
Acknowledgment

I would like to express my deep sense of gratitude to **Prof. Rajat Moona**, for his invaluable support and guidance during the course of project. I am highly oblized to him for constantly encouraging me by giving his critics on my work. I am also very much thankful to Mr. Nitin Dahra, M. Tech Student, Department of Computer Science and Engineering, for his constant support throughout the project.



Amit Gaurav
May 2007
Indian Institute of Technology, Kanpur

Implementation of Breakpoints in GDB for Sim-nML based Architectures

CS499 Report

by

Amit Gaurav
Y3036

under the guidance of

Prof. Rajat Moona

Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur
May 2007

Acknowledgment

I would like to express my deep sense of gratitude to **Prof. Rajat Moona**, for his invaluable support and guidance during the course of project. I am highly oblized to him for constantly encouraging me by giving his critics on my work. I am also very much thankful to Mr. Nitin Dahra, M. Tech Student, Department of Computer Science and Engineering, for his constant support throughout the project.

Amit Gaurav
May 2007
Indian Institute of Technology, Kanpur

Abstract

In every processor model there is a need for generic debugging environment. For this reason almost all the known processor architectures are interfaced with GDB (GNU Debugger) to facilitate the debugging part. Sim-nML is a retargetable processor description language used to develop processor modeling tools. In this project the GDB is modified to debug for Sim-nML based architectures also.

This Project is aimed at including the support of Sim-nML generated processor models in GDB. Currently GDB supports only known processor models. This debugger can be used to support the architecture models generated by Sim-nML language also. This requires porting of GDB as well as BFD (Binary File Descriptor) library to Sim-nML generated processor models.

Porting GDB requires making changes in GDB for target architecture. In this part of work we have implemented breakpoint handling in GDB and Simulator which is one part of the porting. In this case we needed to implement functions according to the Sim-nML based architecture model.

Introduction

In this Project, we have implemented breakpoints in GDB to the Sim-nML generated processor models. The porting done is not a complete porting. It is done as to make the GDB work with the Sim-nML support. This required porting of BFD libraries also. This porting lets the user to select the processor models generated by Sim-nML to be debugged by GDB with constrained functionalities. Earlier the Simulator was only interfaced with GDB with the target architecture same as that of powerpc. The main work includes implementing breakpoints in the GDB structures for the Sim-nML based architectures.

Overview of Sim-nML

Sim-nML is a language for describing an arbitrary processor architecture. It provides processor description at an abstraction level of the instruction set. Sim-nML is flexible, easy to use and is based on attribute grammar. It can be used to describe processor architecture for various processor-centric tools, such as instruction-set simulator, assembler, disassembler in a retargetable manner. Sim-nML has been used as a specification language for generation of various processor modeling tools.

Overview of GDB

GDB is a portable debugger and it supports a large number of target architectures. GDB consists of three major parts: user interface, symbol handling (the symbol side), and target system handling (the target side). The user interface consists of several actual interfaces. The symbol side consists of object file readers, debugging info interpreters, symbol table management, source language expression parsing, type and value printing. The target side consists of execution control, stack frame analysis, and physical target manipulation. An overview of GDB can be seen in the following figure.

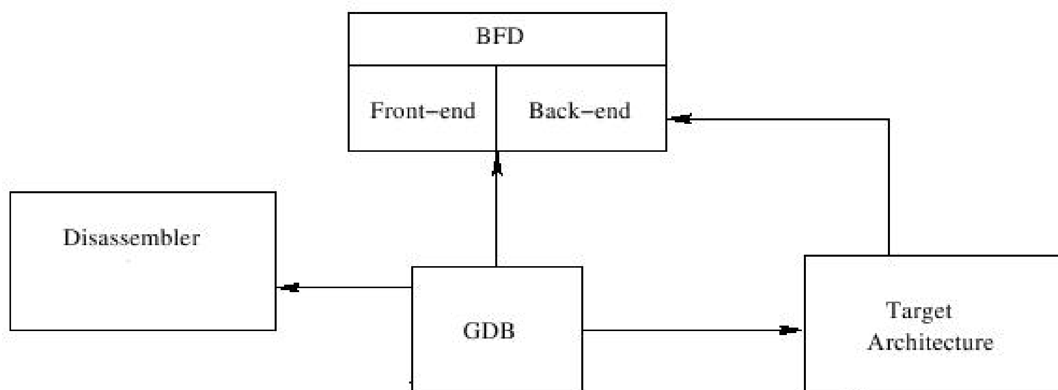


Figure: Block Diagram of GDB

GDB Configuration

GDB runs with support from the three configurations which are host, target and native. Host refers to the attributes of the system where GDB runs. Target refers to the system where the program being debugged executes. If the host and the target are the same machine then the configuration is native.
Defines and include files which are needed to build on the host system are called host support files. Whereas the files required to handle the target format are called target support files. some examples are the stack frame format, instruction set, breakpoint instruction, registers. The native configuration needs different type of files for the control of the system.

Working of GDB

GDB gains access to the target process with the help of the *ptrace* command. Whenever GDB needs to execute a target binary it creates a new process through fork() and exec the target binary. Since GDB should have full control over the child process or in other words it needs to trace the child process, So the child process tells the parent process to trace its execution through ptrace function. An elementary execution will be like:

```
int pid = fork();
if (pid == 0) {
ptrace(PTRACE_TRACEME, 0, 0, 0);
execve(target, args, NULL);
}
else {
wait(&wait_val);
// Execution of the parent process.
}
```

In this way, when executed inside GDB, the GDB will start tracing the executable program. In case the GDB wants to debug a running process, it uses the request PTRACE_ATTACH to connect to the process and after that the target process becomes the child process of GDB virtually. GDB can see andchange the data or the values in the registers using PTRACE_PEEKDATA, PRACE_POKEDATA, PTRACE_GETREGS, PTRACE_SETREGS requests. These functions are very much used in case of breakpoint handling when the data in the target process space needs to be changed.
We are concerned with adding the target architecture definition in the GDB. GDB target architecture defines what sort of machine language programs GDB can work with and the way it works. The target architecture object which is implemented with each target architecture is defined as a C structure

```
struct gdbarch *
```

The structure and its methods are generated using the script gdbarch.sh.

Registers and Memory

GDB assumes that machine has a set of registers with each register may have different size. To get an idea of which registers are used for which purpose i.e how the regiters are used by the compiler. One need to define the registers in the macros like REGISTER_NAME etc. Similar is the idea that the machine contains a block of memory. The Macros which can be used in the target machine are defines below.

`struct type *register_type (gdbarch, reg)`

If the register type is defined then return the type of register *reg* .

`BREAKPOINT`

Breakpoint is the location in the program, when reached triggers a temporary halt in the execution. It is mainly used to check the status of the program in stages. Usually trap instruction is used for a breakoint. The breakpoint should not be longer than the shortest instruction of the architecture.

`BREAKPOINT_FROM_PC`

The program counter is the one to decide the content and size of the breakpoint instruction

`frame_align (address)`

IT is needed to fulfill the alignment requirement for the start of the new stack frame. Specifically if a new frame is created both the initial stack frame pointer and the address of the return value are correctly aligned.

`CORE_ADDR unwind_pc (struct frame_info *this_frame)`

It returns the return address of the *this_frame* . Basically it returns the instruction address at which the execution which resume after *this_frame* returns.

`push_dummy_call (gdbarch, function, regcache, pc_addr, nargs, args, sp, struct_return, struct_addr)`

This function is used to push the frame's call to some other function onto the stack. In addition to pushing number of args, the code should push struct_addr (when struct_return), and the return address (bp_addr). It returns the updated top of the stack pointer.

Adding a New Target

To add a new target architecture to GDB we need to add some files into the GDB architecture.

`gdb/config/arch/arch_name.mt`

This file contains the information about the objects files needed by the target architecture. It is defined by the tag TDEPFILES and TDEPLIBS. IF there exists a header file with the object file then it is defined by TM_FILE. In addition to these variables there are some more specifications but they are deprecated.

`gdb/arch-tdep.c`

This is the main file required for the target machine. It contains functions and the macros which are other wise defined in the header file. The functions conatain the register naming, stack frame format, instruction format and other processor specific methods if needed.

`gdb/arch-tdep.h`

This file describes the basic layout of the target processor chip. The layout of the registers and the stack frames etc.

Porting the GDB

The target machine needs to be configured well to make GDB compiled on that machine. In this case we will name our target architecture as Sim-nML having an alias *simnml* and full three part configuration as simnml-none-none. To port the GDB following are the steps that are needed.

• The first thing is to make Sim-nML visible to GDB at the time of configuration. For this we need to add the Sim-nML introduction in the *config.sub* file. This can be done by adding the three part configuration to the list of supported architectures, vendors and operating systems. It can be just appended at the last of each configuration. Also adding the alias name *simnml* which map to *simnml-none-none* maps that alias to this name. We can test the configuration by typing

```
./config.sub simnml
or
./config.sub simnml-none-none
```

which will return the three-part configuration. In case of any wrong configuration it will return an error.

• To configure inside the GDB source directory first edit the *gdb/configure.tgt* file and need to set the *gdb_target* to *simnml* .

• We need to add some target specific functions in the file *simnml-tdep.c* which define the register and stack implementation. They are:

`simnml_gdbarch_init`

This function instantiates the GDB architecture along with the Sim-nML based architecture. The default GDB functions which are used while debugging are overwritten by the functions which are Sim-nML specific. The GDB architecture is initialized through *struct gdbarch *gdbarch* and the GDB target architecture is initialized through *struct gdbarch_tdep *tdep* . The main functions which are replaced for a generic target architecture are:

```
void set_gdbarch_num_regs (struct gdbarch *gdbarch, int num_regs)
```

Set num_regs that is the number of ARCH's registers to the gdbarch. The related macro is NUM_REGS.

```
void set_gdbarch_register_type (struct gdbarch *gdbarch,
gdbarch_register_type_ftype *register_type)
```

Set gdbarch_register_type_ftype type function to gdbarch. The set function uses to return the type of register that number is regnr. These types are defined in GDB/gdb/gdbtypes.c

```
void set_gdbarch_register_name (struct gdbarch *gdbarch,
gdbarch_register_name_ftype *register_name)
```

Set gdbarch_register_name_ftype type function to gdbarch. The related macro is REGISTER_NAME. The set function uses to return the name of register that number is regnr.

```
void frame_unwind_append_sniffer (struct gdbarch *gdbarch,
frame_unwind_sniffer_ftype *sniffer)
```

frame_unwind_sniffer_ftype type function will call function GDB/gdb/frame.c: frame_pc_unwind to get the address of current frame's function. If the structure frame_unwind pointer that is related with the function accords with this address, return the structure frame_unwind pointer. Return NULL if not.

```
void set_gdbarch_unwind_pc (struct gdbarch *gdbarch,
gdbarch_unwind_pc_ftype *unwind_pc)
```

This function will call function GDB/gdb/frame.c:frame_unwind_register_unsigned or function get_frame_register_signed (These functions will call function gdb/frame.c: frame_register_unwind) to get a register's value from stack. This value is the return address of the function. To choice the register is according to the frame_unwind structure's function that is called by frame_register_unwind. Such as some ARCHs return the value of PC because place of PC in stack saves the return address of the function.

```
void set_gdbarch_unwind_dummy_id (struct gdbarch *gdbarch,
gdbarch_unwind_dummy_id_ftype *unwind_dummy_id)
```

It calls function frame_id_build to get the frame_id of the current frame. The first parameter of frame_id_build is the current SP that is gotten from next_frame. The second parameter of frame_id_build is the current PC that is gotten from next_frame. The following is called process of the function dummy_frame_sniffer: the frame_unwind structure dummy_frame_unwinder that include the function dummy_frame_sniffer is registered to list frame_unwind_data by function frame_unwind_init. Function frame_unwind_init is called by _initialize_frame_unwind. So when GDB call function frame_unwind_find_by_frame to get the frame_unwind structure of a frame, the dummy_frame_unwinder will be check first, then the function dummy_frame_sniffer will be called first.
The function dummy_frame_sniffer uses to make sure if the current frame is dummy frame. In this function, it will call set function pointer unwind_dummy_id to get the

current frame_id. After that, it will compare this frame_id with each frame_id which frame is set to list dummy_frame_stack by function dummy_frame_push. If a frames frame_id is equal with current frame_id, Function frame_unwind_find_by_frame will return the dummy_frame_unwinder. The reason is only dummy_frame_unwinder can unwind the dummy frame.

GDB and Simulator

The GDB is connected to the Simulator through a set of functions which act as an interface to both GDB and the Simulator itself. The functions are:
1) *sim_open (SIM_OPEN_KIND kind, host_callback *callback, struct bfd *abfd, char **argv)*: This function starts the simulator through GDB. The IR (Intermediate Representation) structure is generated at this stage. It can be done from an image file or from nml file. Currently we are generating it from nml file.

2) *sim_load (SIM_DESC sd, char *prog, bfd *abfd, int from_tty)*: This function decodes the binary target given as argument and extracts the instructions, and other data from it and puts it in data structures. The Binary format is assumed to be ELF format and so the reading is based on ELF pattern only.

3) *sim_read (SIM_DESC sd, SIM_ADDR addr, unsigned char *buffer, int size)* : This function is used to read the data from the target memory space. The prototype of the function is almost same as the read system call.

4) *sim_write (SIM_DESC sd, SIM_ADDR addr, unsigned char *buffer, int size)* : Same as the sim_read function but it is used to write in the memory space of the target machine.

5) *sim_create_inferior (SIM_DESC sd, struct bfd *abfd, char **argv, char **env)* : It actually starts the process by calling the init_simulator() process which allocates memory space for the target. Also it initializes all the variables for the execution like program counter, stack pointer etc.

6) *sim_fetch_register (SIM_DESC sd, int rn, unsigned char *memory, int length)* : This function fetches the register value from the target memory space. The register contents are fetch one byte once and are stored in an array.

7) *sim_store_register (SIM_DESC sd, int rn, unsigned char *memory, int length)* : This stores the register value from the data structure into the registers putting one byte once.
The fetch and store register functions are for floating point numbers also with the same function as for the General purpose registers.

8) *sim_stop (SIM_DESC sd)* : This function sets the variable stop_simulator as 1. This stops the simulator.

9) *sim_create_breakpoint (unsigned long address)* : It adds an entry into the breakpoint structure where it stores the address where the breakpoint instruction has to be put and the function which is present at that address.

10) *sim_insert_breakpoint ()* : After the creation of the breakpoint, they are not inserted into the main instruction set. instead we have a separate function for inserting them. The breakpoints are inserted when we resume the execution.

11) *sim_remove_breakpoint ()* : The breakpoints are removed by this function. When the user wants to see the original instruction set, the breakpoint instructions are removed from the set and it is displayed. When again at the time of resuming the execution, breakpoints are inserted.

12) *sim_resume (int step, int signal)* : This function invokes run_simulator which actually run the instructions given in exec_tbl structure. It checks for the execution after the breakpoint stop. Whenever the simulator is stopped due to breakpoint and again it is resumed, this function executes the function in the buffer which was there on behalf of the breakpoint instruction. And then it calls run_simulator for the normal execution thereafter.

Implementing Breakpoints

The breakpoints should be handled in a very different manner because in this case of the simulator the instructions are not taken from the symbol table in the memory but rather from a file structure. Thus in order to implement a breakpoint we need to change the instructure in this structure. The GDB interface is used to facilitate the address of the instruction whenever a brakpoint is addressed. This address from the GDB is used in the simulator. The step by step procedure is :
1) When the GDB starts with the target executable as the argument, the GDB loads this file by reading its symbol table. It uses BFD library to perform this operation. Once it loads the file, it has the information about the instructions and the symbols. By default we are assuming the file to be in the ELF format.
2) For normal execution the GDB transfers the control to the Simulator which then takes the instructions from the instruction structure exec_tbl and executes the instructions one by one. But when GDB gives the breakpoint instruction at certain function or address. We need to replace exact address in the simulator instruction set by the breakpoint instruction. To do this we extract the address from the symbol table loaded in GDB and pass this address to Simulator. In Simulator there is a structure BREAK_ADDR as below:

```
struct BREAK_ADDR {
void (*func_ptr) ();
unsigned long address;
}
BREAK_ADDR[BREAK_POINTS];
```

This structure keeps an record of the breakpoints that have been assigned by the user and the breakpoints are consecutively added into this array. It stores:
- The function pointer which points to the actual function which exists in the instruction set of the executable file.
- The address where the breakpoint has to be inserted.
After this, we have the list of breakpoints which are given by the user.
3) Whenever we resume or run the executable file all the breakpoints which are in the structures are inserted into the actual instruction set and the functions there are replaced with the breakpoint functions. A breakpoint function changes SIM_STAT to STOP which is RUN before. Removes breakpoints every where from the instruction set. Buffers the function where the execution has stopped due to the breakpoint.
4) When we again continue the function after the stop due to the breakpoint, the Simulator inserts the breakpoints again and calls the resume function. The resume function then changes the Simulator status to RUN and executes the buffer function stored before. Then it calls the run_simulator function which then performs the normal execution.
5) In case of next and step instruction, the function information is necesary. For next command, for a function the address of the return value of the function is taken and is given to the simulator. Then the executable has a normal execution until that address is reached. If it reaches then Simulator is stopped and the control is given back to GDB. For a whole line, just the program counter which corresponds to the next instruction is noted and execution is done until that address.

Porting the BFD

We are not completely porting BFD but we are just porting it to make the GDB structure work. To port BFD first thing is to edit the *config.bfd* file.

In the config.sub file we need to add the target architecture as *bfd_simnml_arch* in the list of all supported architectures. Also we need to define the default vector for the simnml generated target. It is defined by the keyword *targ_defvec* . Also in addition to it we also define whether underscores are used or not.

The second file that needs editing is *archures.c* . BFD keeps one atom in BFD describing the architecture of the data attached to the BFD. So we need to define Sim-nML architecture the same way. It is define in the enum *bfd_architecture* . Within the BFD the information about the architectures is also contained in a structure called *bfd_arch_info* . We need to initialize the Sim-nML based architecture here also. For various architecture information about them is defined in this file like that of mips, sparc etc. But since Sim-nML is not certain architecture specific so we left the addition at the time of execution only.

The *targets.c* lists all the vector for the target architectures. For running purpose we define the default vector for the Sim-nML based architecture to be *elf32* . Thus we need to have an entry of simnml here also.

We need to create a file *cpu-simnml.c* which contains the information about the target architecture. To make this porting work we have included the default generic information about the Sim-nMl generated architecture models. Every information is put up in the structure *bfd_arch_info_type* which is already instantiated in the archures.c file. It contains various parameters some of them are:

No. of bits in a word - In this case we kept it 32.
No. of bits in an address - It is 32 in case of Sim-nML based architecture.
No. of bits in a byte - Keeping it generic it is 8.
Name of the BFD architecture of the target architecture - bfd_arch_simnml.
No. of Machines associated with this architecture - By machines it specifies whether its a 32-bit machine or a 64-bit machine etc. it is 1 in this case which is 32-bit.

The other file which we need to create is *elf32-simnml.c* , the vector file for the target architecture. In case of the Sim-nML based architecture we have kept it as 32-bit elf target. In this we define some macros which are:

```
ELF_ARCH
ELF_MACHINE_CODE
ELF_MAX_PAGESIZE
bfd_elf32_bfd_reloc_type_lookup
bfd_elf32_bfd_get_relocated_section_contents
```

Lastly we need to add the Sim-nML description in the *configure.in* file so that the architecture can be added during the compilation of the source. In *configure* file it basically mentions the files required to produce target binary in each case listed alphabetically.

Result

The GDB architecture is now recognizing the Sim-nML generated architectures models with some default functionality. The compilation with target as simnml is done successfully. We used only static library of the GDB with no dynamic linking. The execution of the binary targets with Sim-nML as target architecture definition can be run with the GDB support. The debugging of the Simulator with the GDB support for the executables of certain architectures is done with breakpoint handling support.

Conclusion

In this section of work, we have implemented breakpoint handling in the GDB to support the Sim-nML based architecture models. This was needed as to provide a decent debugging environment. We have also ported the BFD configuration to support the architecture aforesaid. The porting of the GDB is not complete as it needs other information like registers etc. In our case we used only one binary file format which is *elf32* .

Bibliography

[1] BFD Library URL: http://www.gnu.org/software/binutils/manual/bfd-2.9.1/bfd.html.

[2] Surendra Kumar Vishnoi. Functional Simulation Using Sim-nML. Master's Thesis Department of Computer Science and Engineering, IIT Kanpur, May 2006.

[3] GDB Internals URL: http://sources.redhat.com/gdb/current/onlinedocs/gdbint.html

[4] Sim-nML Specifications URL: http://www.cse.iitk.ac.in/users/simnml/docs/simnml.pdf

[5] The GNU Project Debugger URL: http://www.gnu.org/software/gdb/