User Manual for GDB based debugger for AJIT Processor

Titto Thomas

July 1, 2016

1 Introduction

Ajit processor is based on SPARC V8 instruction set, and currently has seperate C and micro-architecture implementations. This documents contains useful information regarding the usage of GDB to debug the programs running on any of the following AJIT processor models.

- ISA-C model
- Micro-architecture model in C
- Micro-architecture VHDL model running in simulator
- Micro-architecture model on FPGA prototype

2 Building AJIT processor models

Throughout the document we will be using repository home to indicate the Ajit project git repository. First go to that directory and run the following commands.

```
$ source exports.sh
$ ./build.sh
```

This will build the both processor and ISA-C model executables. For using GDB to debug the programs running on it, you need to use the appropriate one from the following.

ISA-C model	:	`AJIT_cpu_cache_mmu_testbench_gdbserver`
Micro-architecture model in C	:	`ajit_cpu_uarch_test_gdbserver`
Micro-architecture VHDL model running in simulator	:	`ajit_cpu_uarch_test_gdbserver_sockpipes`
Micro-architecture model on FPGA prototype	:	`ajit_cpu_uarch_test_gdbserver_riffa`

3 Creating executables for AJIT

We are using the cross compiler and gdb that comes along with the buildroot package. The installation instructions are given in `repository_home/tools/buildroot/README`. After installing them, go to the directory where your C file is located and run the following command

```
$ cToSparc.py <input file>.c
```

An `output` directory will be created in the same folder with memory map files and executable.

4 Starting a session of GDB for AJIT

Go to the directory where your C file was compiled and enter the following command.

```
$ sparc-linux-gdb <input file>.elf
```

Now the gdb will start and you can see the messages shown in Fig. 1

Figure 1: GDB debug session

Note: The reading symbols from elf file should succeed without any errors

Now, open another tab in the terminal and navigate to `repository_home`, run

```
$ source exports.sh
```

and move over to the directory where your C file was compiled. Run the appropriate AJIT processor executable depending on the model to be used.

ISA-C model

```
$ AJIT_cpu_cache_mmu_testbench_gdbserver <input file>_memmap.txt
<port number>
```

Micro-architecture model in C

```
$ ajit_cpu_uarch_test_gdbserver -m <input file>_memmap.txt
-p <port number>
```

Micro-architecture VHDL model running in simulator

```
$ ajit_cpu_uarch_test_gdbserver_sockpipes -m <input file>_memmap.txt
-p <port number>
```

In this case the simulator has to be started as well. Open another tab in the terminal, navigate to `repository_home/processor/Aa_v2/modelsim/vsim_functional_check` and run the following script.

```
$ run_modelsim vsim.do
```

Micro-architecture model on FPGA prototype

```
$ ajit_cpu_uarch_test_gdbserver_riffa -m <input file>_memmap.txt
-p <port number>
```

You can give any of the allowed free ports for `port number`, preferably 8888.

Now go back to the other tab where gdb is located, and enter the following command with the same `port number`.

```
(gdb) target remote :<port number>
```

The gdb side will also show the following messages confirming the connection establishment. Since the connection has now been established, you can control and monitor the program execution on AJIT processor in real time.

```
(gdb) target remote :8888
Remote debugging using :8888
0x00000000 in ?? ()
(gdb)
```

Basic GDB commands for use

Continue / Next

When the execution is stopped due to some reason, you can let the processor continue with these commands.

```
(gdb) continue
(gdb) c
```

Next command also lets the processor continue, but stops it after executing one more line of code.

```
(gdb) next
(gdb) n
```

Breakpoints

Breakpoints are useful for stopping the execution on reaching specific points in the source code, and probe the processor for information. They can be set using the following commands.

```
(gdb) break <function_name>
(gdb) break <line_number>
(gdb) break <filename : function_name>
(gdb) break <filename : line_number>
(gdb) break *<address>
```

Example:

```
(gdb) break 24
Breakpoint 1 at 0xf0004044: file Watchpoints.c, line 24.
```

AJIT processor will stop execution when a breakpoint is hit, and you will be notified in gdb.

```
(gdb) continue
Continuing.

Breakpoint 1, main () at Watchpoints.c:24
24              j = 25;
(gdb)
```

Breakpoints can be listed by the following command.

```
(gdb) info breakpoints
```

This command can be used to delete specific breakpoints.

```
(gdb) delete break <breakpoint_number>
```

Watchpoints

Breakpoints are useful for stopping the execution on modification of specific variables in the source code, and probe the processor for information. They can be set using the following commands. This command can be used to delete specific breakpoints.

```
(gdb) watch [-l|-location] <variable>
(gdb) break [-l|-location] <expression>
```

Ordinarily a watchpoint respects the scope of variables in expression. The `-location` argument tells gdb to instead watch the memory referred to by expression.

Example:

```
(gdb) watch t
Hardware watchpoint 2: t
```

AJIT processor will stop execution when a watchpoint is hit, and you will be notified with their values in gdb.

```
(gdb) continue
Continuing.
Hardware watchpoint 2: t

Old value = 20
New value = 5
0xf0004058 in main () at Watchpoints.c:25
25              t = j - t;
(gdb)
```

Watchpoints can be listed by the following command.

```
(gdb) info watchpoints
```

This command can be used to delete specific watchpoints.

```
(gdb) delete watch <watchpoint_number>
```

Traps

While the AJIT processor is executing a program, it will stop and inform the gdb in case of any traps. After it traps, you can use the `tbr` register to get information about the trap occurred.

Example:

```
(gdb) c
Continuing.

AJIT: Trap Occured

Program received signal SIGTRAP, Trace/breakpoint trap.
halt () at /home/titto/AJIT_Project/AjitRepoV2/os/kernels/pico/src/sparc_stdio.c:7
7       __asm__ __volatile__("ta 0\n\t"
(gdb)
```

Setting variables / memory / registers

You can modify the values of variables, memory contents and register values with the set command.

```
(gdb) set var <variable> = <value>
(gdb) set $ <register> = <value>
(gdb) set *(int)<memory_address> = <value>
```

Example:

```
(gdb) set *(int)0xfffffff8 = 0x11
(gdb) set $l1=0x55
(gdb) set var t=0x50
```

Printing variables / memory / registers

You can display the values of variables, memory contents and register values with the `print` or `p` command.

```
(gdb) print var <variable>
(gdb) print $ <register>
(gdb) print *(int)<memory_address>
```

Example:

Detach

You can stop debugging the program and let the processor continue till it finishes execution, using this command.

```
(gdb) detach
```

Example:

Other useful commands

You can get help on any commands using

```
(gdb) h[elp]
(gdb) h[elp] <command>
```

Finish current function, loop, etc. with

```
(gdb) fin[ish]
```

Show lines of code surrounding the current point

```
(gdb) l[ist]
```

Delete all breakpoints

```
(gdb) d[elete]
```

Add a list of gdb commands to execute each time a breakpoint is hit

```
(gdb) comm[ands] <breakpoint_number>
```