

# *Port GDB to a new processor architecture: TI C6x*

Yao Qi

[yao@codesourcery.com](mailto:yao@codesourcery.com)

CodeSourcery/MentorGraphics

2013-07-13

- 1 *Outline*
  - Overview
- 2 *An overview of GDB*
  - Two kinds of GDB
- 3 *Bare metal or ELF*
  - breakpoint & software single step
  - inferior call
  - frame unwinding
  - prologue analyzer
  - epilogue detection
  - longjmp
- 4 *Linux or ucLinux*
  - stub frame unwinder
  - signal trampoline frame unwinding
  - next pc of syscall
- 5 *Conclusions*

# Overview

## Introduction

- TMS320 C6000 series, or TMS320C6x: VLIW-based DSPs,
- Internal development in 2Q 2011, thank Bernd Schmidt and Andrew Jenner for the various explanations,
- Upstream from July to August 2011, thank Joseph Myers and GDB maintainers for the patient review,

## Goals of this session

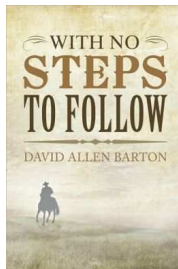
- List the **basic steps** of GDB porting. GDB has been ported to many different architectures, but the **steps** and **details** about porting is unclear.
- Make the GDB internals, **interactions** among different components, clear enough for GDB porting.

## *Typical submissions*

[RFA 0/8] New port: TI C6x  
[RFA 1/8] New port: TI C6x: Remove "gdb" from noconfigdirs in configure.ac  
[RFA 2/8] New port: TI C6x: Handle tic6x-\*-linux and tic6x-\*- in configure.tgt  
[RFA 3/8] New port: TI C6x: shared library for dsbt  
[RFA 4/8] New port: TI C6x: Read loadmap from gdbserver  
[RFA 5/8] New port: TI C6x: gdb port  
[RFA 6/8] New port: TI C6x: gdbserver  
[RFA 7/8] New port: TI C6x: test case fixes  
[RFA 8/8] New port: TI C6x: NEWS

[PATCH] Add support for Tilera TILE-Gx processor (part 1/3: gdb)  
[PATCH] Add support for Tilera TILE-Gx processor (part 2/3: gdb)  
[PATCH] Add support for Tilera TILE-Gx processor (part 3/3: gdbserver)

[patch 0/4] Altera Nios II port  
[patch 1/4] code changes for Nios II target  
[patch 2/4] Nios II target descriptions  
[patch 3/4] Nios II gdbserver support  
[patch 4/4] Nios II testsuite fix



## *Steps matter!*

In patch reader's minds:



In patch writer's minds:



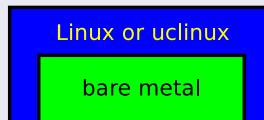
## Two kinds of GDB

### Bare metal or ELF

- breakpoint, software single step,
- inferior call, `dwarf2_frame_init_reg`,
- prologue analyzer, frame unwinding by prologue, skip prologue,
- epilogue detection,
- register type and group,
- `longjmp` target,

### on Linux or ucLinux

- stub prologue analyzer,
- signal trampoline unwinding,
- next pc of syscall (`rt_sigreturn`, `sigreturn`)
- thread local storage



## breakpoint

- A special instruction or an illegal instruction,
- The basic attribute of GDB, and the first step of porting,

```
326 static const gdb_byte *  
327 tic6x_breakpoint_from_pc (struct gdbarch *gdbarch, CORE_ADDR *bp_addr,  
328                          int *bp_size)  
329 {  
330     struct gdbarch_tdep *tdep = gdbarch_tdep (gdbarch);  
331  
332     *bp_size = 4;  
333  
334     if (tdep == NULL || tdep->breakpoint == NULL)  
335     {  
336         if (BFD_ENDIAN_BIG == gdbarch_byte_order_for_code (gdbarch))  
337             return tic6x_bkpt_illegal_opcode_be;  
338         else  
339             return tic6x_bkpt_illegal_opcode_le;  
340     }  
341     else  
342         return tdep->breakpoint;  
343 }
```

Listing 1: breakpoint

## software single step

- Decode the instruction, and compute the **next** instruction of pc,
- Insert breakpoint at the **next** instruction of pc, resume and wait,

```
698 static int  
699 tic6x_software_single_step (struct frame_info *frame)  
700 {  
701     struct gdbarch *gdbarch = get_frame_arch (frame);  
702     struct address_space *aspace = get_frame_address_space (frame);  
703     CORE_ADDR next_pc = tic6x_get_next_pc (frame, get_frame_pc (frame));  
704  
705     insert_single_step_breakpoint (gdbarch, aspace, next_pc);  
706  
707     return 1;  
708 }
```

Listing 2: software single step

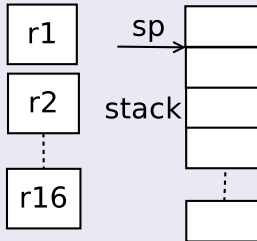


## inferior call

```
1 struct A func1 (int c, struct B b, long long d, double e, struct C f);
```

### Widely used

- p func1(1, b, 10, 0.1, f)



### GDB needs to know

- type of arguments and return,
- place to hold arguments (registers or stack),
- place of return value (registers or stack),
- pass by value or pass by reference?
- alignment of storing,
- vararg,

## *inferior call in gdb*

GDB will do the following steps:

- 1 Allocate a new piece of stack, which is called **dummy frame**, and put all arguments into the right place (registers and stack) and set the expected return address. Done by [arch\\_push\\_dummy\\_call](#),
- 2 Adjust register **sp**. [arch\\_frame\\_align](#) is needed,
- 3 Push all the info needed to restore the caller's state on **dummy frame**. [arch\\_dummy\\_id](#) is needed,
- 4 Set a breakpoint on the return address, and resume the program,
- 5 When breakpoint is hit, GDB gets the return value. Done by [arch\\_return\\_value](#).

## frame unwinding

### How GDB handles various different frames?

- Various unwinders are chained together,
- Iterate over the unwinders and associate a unwinder to the frame if the sniffer thinks this frame can be handled by the unwinder,

```
1306 /* Unwinding. */
1307 dwarf2_append_unwinders (gdbarch);
1308
1309 frame_unwind_append_unwinder (gdbarch, &tic6x_stub_unwind);
1310 frame_unwind_append_unwinder (gdbarch, &tic6x_frame_unwind);
```

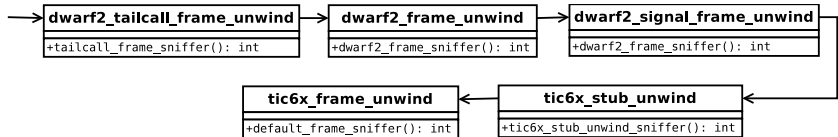


Figure: unwinders chain

## prologue analyzer

### Missions of prologue analyzer

- Where is the end of prologue? starting address of function body,
- What does prologue do? where are registers saved? sp, fp?

### Functionality of prologue analyzer

- frame unwinding by analyzing prologue,
- GDB needs to know the starting address of function body. When `break foo`, **arch\_skip\_prologue** is used to set the breakpoint at starting address of the function body.

## prologue analyzer

```
301 static CORE_ADDR
302 tic6x_skip_prologue (struct gdbarch *gdbarch, CORE_ADDR start_pc)
303 {
304     CORE_ADDR func_addr;
305     struct tic6x_unwind_cache cache;
306
307     /* See if we can determine the end of the prologue via the symbol table.
308      * If so, then return either PC, or the PC after the prologue, whichever is
309      * greater. */
310     if (find_pc_partial_function (start_pc, NULL, &func_addr, NULL))
311     {
312         CORE_ADDR post_prologue_pc
313             = skip_prologue_using_sal (gdbarch, func_addr);
314         if (post_prologue_pc != 0)
315             return max (start_pc, post_prologue_pc);
316     }
317
318     /* Can't determine prologue from the symbol table, need to examine
319      * instructions. */
320     return tic6x_analyze_prologue (gdbarch, start_pc, (CORE_ADDR) -1, &cache,
321                                     NULL);
322 }
```

Listing 3: skip prologue

## epilogue detection

- Suppose a watchpoint is used to monitor a local variable, gdb takes care of the scope of the local variable,
- When the program goes to epilogue, GDB will remove the watchpoint by mistake because it doesn't know the program is still within the scope (in epilogue of the function),

```
1128 static int  
1129 tic6x_in_function_epilogue_p (struct gdbarch *gdbarch, CORE_ADDR pc)  
1130 {  
1131     unsigned long inst = tic6x_fetch_instruction (gdbarch, pc);  
1132     /* Normally, the epilogue is composed by instruction 'b .S2 b3'. */  
1133     if ((inst & 0x0f83effc) == 0x360)  
1134     {  
1135         unsigned int src2 = tic6x_register_number ((inst >> 18) & 0x1f,  
1136                                                    INST_S_BIT (inst),  
1137                                                    INST_X_BIT (inst));  
1138         if (src2 == TIC6X_RA_REGNUM)  
1139             return 1;  
1140     }  
1141  
1142     return 0;  
1143 }
```

## epilogue detection

These fails are fixed:

FAIL: gdb.base/watch-cond.exp: watchpoint with local expression, local condition evaluates in correct frame

FAIL: gdb.mi/mi-watch.exp: sw: watchpoint trigger (unknown output after running)

FAIL: gdb.mi/mi2-watch.exp: sw: watchpoint trigger (unknown output after running)

## longjmp

- GDB needs to know the **protocol** between longjmp and setjmp, and extract the target address of longjmp,
- so program can stop after the location of setjmp,

```
1147 static int
1148 tic6x_get_longjmp_target (struct frame_info *frame, CORE_ADDR *pc)
1149 {
1150     struct gdbarch *gdbarch = get_frame_arch (frame);
1151     enum bfd_endian byte_order = gdbarch_byte_order (gdbarch);
1152     CORE_ADDR jb_addr;
1153     gdb_byte buf[4];
1154
1155     /* JMP_BUF is passed by reference in A4. */
1156     jb_addr = get_frame_register_unsigned (frame, 4);
1157
1158     /* JMP_BUF contains 13 elements of type int, and return address is stored
1159        in the last slot. */
1160     if (target_read_memory (jb_addr + 12 * 4, buf, 4))
1161         return 0;
1162
1163     *pc = extract_unsigned_integer (buf, 4, byte_order);
1164
1165     return 1;
1166 }
```



## longjmp

These fails are fixed:

```
FAIL: gdb.base/longjmp.exp: next over longjmp(1)
FAIL: gdb.base/longjmp.exp: next over call_longjmp (2)
FAIL: gdb.base/longjmp.exp: next over patt3
```

## stub frame unwinder

### Why does GDB need a stub frame unwinder

- When type command next, the program will cross the function call, and stop at the next line,
- GDB won't stop the program if it is still in the inner frame,
- When the program is in plt stub, GDB can't tell the program is still in the inner frame or not, so it stops the program by mistake,
- GDB needs a special frame unwinder for plt stub,

## stub frame unwinder

```
525 static int
526 tic6x_stub_unwind_sniffer (const struct frame_unwind *self,
527                             struct frame_info *this_frame,
528                             void **this_prologue_cache)
529 {
530     CORE_ADDR addr_in_block;
531
532     addr_in_block = get_frame_address_in_block (this_frame);
533     if (in_plt_section (addr_in_block))
534         return 1;
535
536     return 0;
537 }
538
539 static const struct frame_unwind tic6x_stub_unwind =
540 {
541     NORMAL_FRAME,
542     default_frame_unwind_stop_reason,
543     tic6x_stub_this_id,
544     tic6x_frame_prev_register,
545     NULL,
546     tic6x_stub_unwind_sniffer
547 };
```

Listing 6: stub frame unwinder

## signal trampoline frame unwinding

- GDB has a good infrastructure to do frame unwinding in signal trampoline,
- Each port has to define its instruction pattern to match its signal trampoline code sequence.

```
139 static struct tramp_frame tic6x_linux_rt_sigreturn_tramp_frame =  
140 {  
141     SIGTRAMP_FRAME,  
142     4,  
143     {  
144         {0x000045aa, 0x0fffffff}, /* mvk .S2 139,b0 */  
145         {0x10000000, -1}, /* swe */  
146         {TRAMP_SENTINEL_INSN}  
147     },  
148     tic6x_linux_rt_sigreturn_init  
149 };
```

Listing 7: tic6x\_linux\_rt\_sigreturn\_tramp\_frame

## signal trampoline frame unwinding

- Check the kernel source and find the location of struct `rt_sigframe` on stack. Usually, it is an offset based on `sp`.

```
83 static void
84 tic6x_linux_rt_sigreturn_init (const struct tramp_frame *self,
85                               struct frame_info *this_frame,
86                               struct trad_frame_cache *this_cache,
87                               CORE_ADDR func)
88 {
89     struct gdbarch *gdbarch = get_frame_arch (this_frame);
90     CORE_ADDR sp = get_frame_register_unsigned (this_frame, TIC6X_SP_REGNUM);
91     /* The base of struct sigcontext is computed by examining the definition of
92        struct rt_sigframe in linux kernel source arch/c6x/kernel/signal.c. */
93     CORE_ADDR base = (sp + TIC6X_SP_RT_SIGFRAME
94                      /* Pointer type *pinfo and *puc in struct rt_sigframe. */
95                      + 4 + 4
96                      + TIC6X_SIGINFO_SIZE
97                      + 4 + 4 /* uc_flags and *uc_link in struct ucontext. */
98                      + TIC6X_STACK_T_SIZE);
```

Listing 8: `tic6x_linux_rt__sigreturn_init`

## *next pc of syscall*

### *Next pc*

- GDB backend on software single step computes the next pc of a given instruction,
- except for syscall, such as sigreturn or rt\_sigreturn.

### *Implementation in GDB*

- There are similar implementations in MIPS, ARM, C6x and NIOS 2,
- When computing the next pc, take syscall into account. Get the syscall number, if it is sigreturn or rt\_sigreturn, get the address from a certain register specified by the specification.

## next pc of syscall

```
154 static CORE_ADDR
155 tic6x_linux_syscall_next_pc (struct frame_info *frame)
156 {
157     ULONGEST syscall_number = get_frame_register_unsigned (frame,
158                                                             TIC6X_B0_REGNUM);
159     CORE_ADDR pc = get_frame_pc (frame);
160
161     if (syscall_number == 139 /* rt_sigreturn */)
162         return frame_unwind_caller_pc (frame);
163
164     return pc + 4;
165 }
166
167 static void
168 tic6x_uclinux_init_abi (struct gdbarch_info info, struct gdbarch *gdbarch)
169 {
170     struct gdbarch_tdep *tdep = gdbarch_tdep (gdbarch);
171
172     linux_init_abi (info, gdbarch);
173
174     /* Shared library handling. */
175     set_solib_ops (gdbarch, &dsbt_so_ops);
176     tdep->syscall_next_pc = tic6x_linux_syscall_next_pc;
177 }
```

Listing 9: next pc of syscall

## next pc of syscall

```
602 static CORE_ADDR
603 tic6x_get_next_pc (struct frame_info *frame, CORE_ADDR pc)
604 {
605     struct gdbarch *gdbarch = get_frame_arch (frame);
606     unsigned long inst;
607     int register_number;
608     int last = 0;
609
610     do
611     {
612         inst = tic6x_fetch_instruction (gdbarch, pc);
613
614         last = !(inst & 1);
615
616         if (inst == TIC6X_INST_SWE)
617         {
618             struct gdbarch_tdep *tdep = gdbarch_tdep (gdbarch);
619             if (tdep->syscall_next_pc != NULL)
620                 return tdep->syscall_next_pc (frame);
621         }
622     }
```

Listing 10: get next pc



## Conclusions

### Conclusions

- Follow the steps during the porting,
- GDB porting is a process:  
FAIL -> component ->  
patch -> PASS
- Track the changes of test results after a feature is ported,
- Be familiar with GDB testsuite,
- Have a look at other ports,
- Fix as many fails as you can,

Questions?

## What do we have to modify?

