How do debuggers (really) work?

Paweł Moll
<pawel.moll@arm.com>

Debugger

ARM

Debugger

A debugger or debugging tool is a computer program that is used to test and debug other programs (the "target" program).

— https://en.wikipedia.org/wiki/Debugger

ARM

Debugger

I don't like debuggers. Never have, probably never will.

ARM

Debugger

I don't like debuggers. Never have, probably never will.

— Linus Torvalds <torvalds@transmeta.com> (2000)

ARM

The plan

- Ptrace Me If You Can
- Everything You Always Wanted To Know About Breakpoints*
- The Hardware Strikes Back
- The Meaning of Debugging Symbols
- Interview With The JTAG

*) But Were Afraid To Ask

ARM

ptrace

- ```
  long ptrace(enum __ptrace_request request, pid_t pid,
                     void *addr, void *data);
  ```

  The ptrace() system call provides a means by which one process (the ”tracer”) may observe and control the execution of another process (the ”tracee”), and examine and change the tracee’s memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.

  — *man 2 ptrace*

- Used by gdb, strace, DynInst...

ARM
  ```

Attaching to a process

- `(gdb) start` PTRACE_TRACEME – makes parent a tracer (called by a tracee)
- `(gdb) attach PID` PTRACE_ATTACH – attach to a running process
- Watch out for Yama security module:

```
Could not attach to process. If your uid matches the uid of the target
process, check the setting of /proc/sys/kernel/yama/ptrace_scope, or try
again as the root user. For more details, see /etc/sysctl.d/10-ptrace.conf
ptrace: Operation not permitted.
```

- `prctl(PR_SET_PTRACER, pid, ...)`

ARM

Basic control

- `(gdb) stop` `kill(child_pid, SIGSTOP)` (or PTRACE_INTERRUPT)
- `(gdb) continue` PTRACE_CONT
- `(gdb) info registers` PTRACE_GET(FP)REGS(ET) and PTRACE_SET(FP)REGS(ET)
- `(gdb) x` PTRACE_PEEKTEXT and PTRACE_POKETEXT

ARM

Setting a breakpoint

- `(gdb) br *ADDRESS`
- Instruction at the given address is read, saved and replaced with a breakpoint:
  - either a special instruction,
  - or an undefined encoding.

ARM

Hitting a breakpoint

- Executing breakpoint instruction causes:
  - SIGTRAP when using special instruction,
  - SIGILL for undefined instructions.
- Any signal destined for the tracee stops its execution.
- Tracer is notified about it via `waitpid(PID)` result.
- Breakpoint-related signals are suppressed (otherwise would be delivered to the tracee after continuing).

ARM

Software breakpoints

- This kind of breakpoints is known as software breakpoints (sometimes as memory breakpoints, not to be confused with watchpoints).
- There is no limitation on its number being simultaneously active (expect for the memory available for the tracer).
- Requires modification of the program code
  - Applies to all threads of execution
  - Can be dangerous if done wrong (leftovers).
  - May requires cache maintenance.
  - Requires write access to the program memory.

Replaying the instruction

- To continue, the instruction replaced with a breakpoint has to be executed.
- It may be either temporarily restored and single stepped...

```
instr1 ← PC        instr1             instr1             instr1
break              break ← PC         instr2 ← PC        break
instr3             instr3             instr3             instr3 ← PC
```

- Or executed outside of the normal program text segment, in a scratch pad allocated and managed by the debugger.

```
instr1 ← PC        instr1             instr1             instr1
break              break ← PC         break              break
instr3             instr3             instr3             instr3 ← PC

                                      (...)

                                      instr2 ← PC
```

ARM

Single stepping and reverse debugging

- Stepping through individual instructions or source codes lines is more complex than it may appear.
  - Need to decode program flow to decide what do `(gdb) next` or `(gdb) nexti` mean.
  - Can be significany simplified by hardware.
- Reverse debugging provides `(gdb) previous` command(s), allowing program history inspection (sometimes including data changes)
  - Generally relies on tracing the program execution.
  - The simplest implementation: single stepping through the code and recording context.
  - Can be also radically less expensive with hardware trace support.

ARM

Conditional breakpoints

- Breakpoint can have a script (set of conditions) attached to it.
- Each breakpoint is always taken, but gdb can automatically continue if conditions not met.
- Can introduce significant overhead, particularly when debugging remotely.
- gdb provides alternative in a form of tracepoints.
  - Can collect data and store in a buffer.
  - Can be entirely evaluated in the remote gdb stub...
  - or even in the tracee address space, using an interpreter loaded as a shared object...
  - replacing the breakpoint with a branch to a jump pad.
- Also consider DBI tools like DynInst
  - Condition evaluation or data collection performed by JITed code.

ARM

Debug hardware

- Additional logic (execution mode) in processor.
- "'Expensive" in terms of silicon area and pins.
- Can provide halting mode debugging.
- Can allow simple single stepping.
- Usually provides support for hardware breakpoints and watchpoints.
- Often provides tracing capabilities (not covered here).

ARM

Hardware breakpoints

- Comparator watching program counter value against a pre-programmed address.
- Limited number of such resources.
- Usually generates instruction fetch exception, resulting in SIGILL or SIGTRAP.
- As with software breakpoints, can be recognized based on PC value and list of active breakpoints.
- Can be implemented as a kernel perf event and interfaced with PTRACE_SET(GET)HBPREGS.

ARM

Watchpoints

- Sometimes known as data or memory breakpoints (not to be confused with software breakpoints).
- Comparator observing data address bus.
- Can trigger on loads (memory read), stores (memory write) or both.
- Usually generates data abort exception, resulting in SIGSEGV or SIGTRAP.

ARM

Debug information

- Optional sections in an ELF file, generated by a compiler.
- Defined in the DWARF standard (it's a world of fantasy).
- Debuggers live and die on its quality.
  - Trade-off between debugging accuracy and code optimisation.
- Symbols description (functions, variables, compilation units) in `.debug_info`.

```
<0><51>: Abbrev Number: 1 (DW_TAG_compile_unit)
    <52>   DW_AT_producer    : (indirect string, offset: 0x0): GNU C 4.6.3
    <56>   DW_AT_language    : 1         (ANSI C)
    <57>   DW_AT_name        : a.c
    <5b>   DW_AT_comp_dir    : (indirect string, offset: 0x13): /home/pawmol01
    <5f>   DW_AT_low_pc      : 0x836c
    <63>   DW_AT_high_pc     : 0x837e
    <67>   DW_AT_stmt_list   : 0x36
<1><6b>: Abbrev Number: 2 (DW_TAG_subprogram)
    <6c>   DW_AT_external    : 1
    <6d>   DW_AT_name        : a
    <6f>   DW_AT_decl_file   : 1
    <70>   DW_AT_decl_line   : 3
    <71>   DW_AT_prototyped  : 1
    <72>   DW_AT_type        : <0x82>
    <76>   DW_AT_low_pc      : 0x836c
    <7a>   DW_AT_high_pc     : 0x837e
    <7e>   DW_AT_frame_base  : 0x2c (location list)
```


ARM

Source line information

- `.debug_line` section.
- Mapping machine instruction address to source code line.

```
CU: a.c:
File name            Line number    Starting address
a.c                            4              0x836c
a.c                            5              0x8370
a.c                            6              0x837a
```

- Optimized code very hard to describe
  - Optimised out symbols (missing functions and variables).
  - Code reordering and folding (no instruction per source line).
  - C++ templates (more than one source line per instruction).
  - Inlining (code flow disturbances).

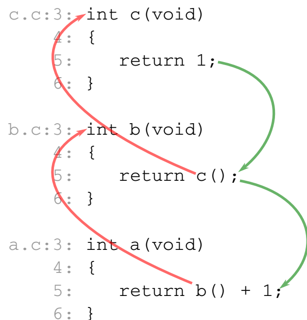
```
File name            Line number    Starting address
./dhry_1.c:[++]
dhry_1.c                     196              0x8806
/usr/include/bits/string3.h:
string3.h                    105              0x8808
./dhry_1.c:[++]
dhry_1.c                     196              0x880c
dhry_1.c                     210              0x881a
dhry_1.c                     211              0x8822
dhry_1.c                     210              0x8824
dhry_1.c                     382              0x882a
dhry_1.c                     211              0x882e
dhry_1.c                     382              0x8832
dhry_1.c                     211              0x8838
dhry_1.c                     382              0x883c
dhry_1.c                     172              0x8840
dhry_1.c                     233              0x884e
/usr/include/bits/stdio2.h:
stdio2.h                     105              0x885c
./dhry_1.c:[++]
dhry_1.c                     257              0x88dc
/usr/include/bits/stdio2.h:
stdio2.h                     105              0x88e4
./dhry_1.c:[++]
dhry_1.c                     253              0x890c
```


ARM

Call stack

- Also known as backtrace.
- Created basing on function return addresses on the program stack.
- Function `a()` calls function `b()` which calls `c()`.
- `(gdb) bt` can show the following:

```
#0 c () at c.c:5
#1 0x00008388 in b () at b.c:5
#2 0x00008374 in a () at a.c:5
```



ARM

Stack unwinding

- Frame Pointer based stack walking fast, but not always possible.
- Unwinding information (almost) accurate.
- Defined as Call Frame Information in DWARF specification.
- State machine taking Frame Description Entry instructions and processing them in a loop.
- Processing starts from a entry associated with a given instruction.
  - Unwinding tables can be generated for all instructions in the program,
  - or only for certain ones which can cause exceptions.
- The result is a pointer to the top of a stack frame, enabling return address recovery.
- Unwinding information can be big and is expensive to process.

ARM

~~Call~~ Return stack

- But `b()`'s return value is equal to the value returned by `c()`, therefore `c()` can return directly to `a()`.
  - The blue branch does not save return address on stack.
  - Tail call optimisation.
- In result `b()` was called, but `(gdb) bt` will not show it:

```
#0 c () at c.c:6
#1 0x00008366 in a () at a.c:5
```

```
c.c:3: int c(void)
    4: {
    5:     return 1;
    6: }

b.c:3: int b(void)
    4: {
    5:     return c();
    6: }

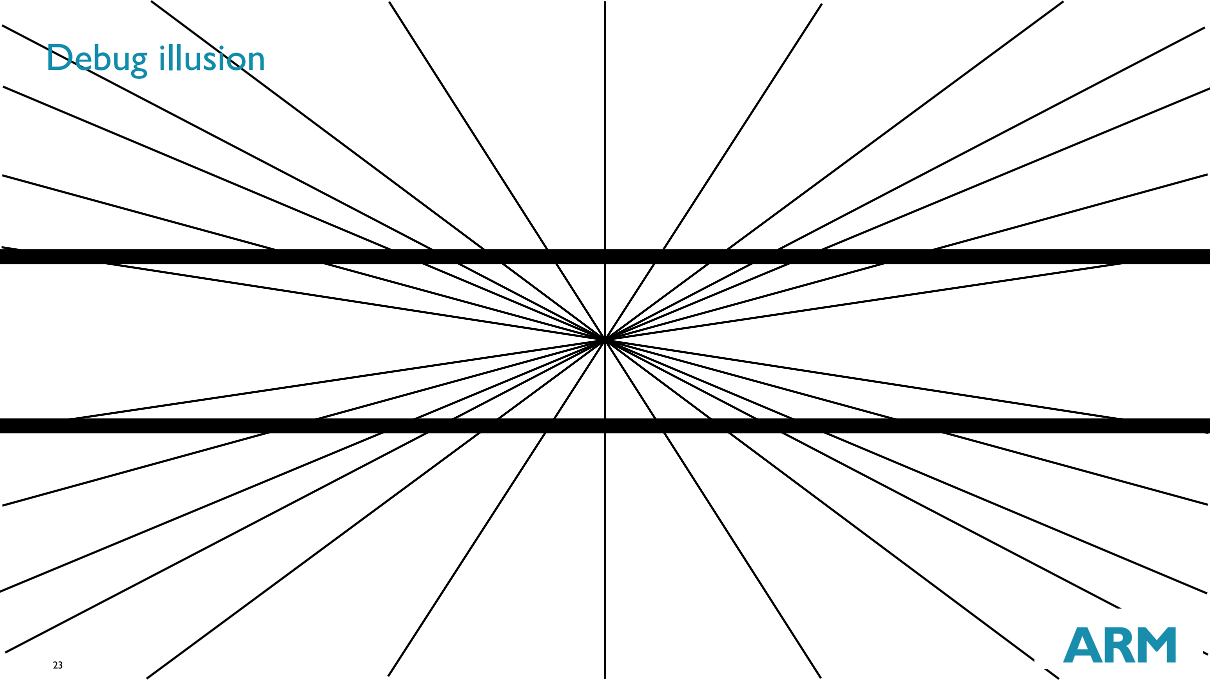
a.c:3: int a(void)
    4: {
    5:     return b() + 1;
    6: }
```

ARM

Debug illusion

ARM

Debug illusion



Remote (Serial) Protocol

- Provides cross-development environment, with tracee being traced by a “gdb stub”, with debugger UI running on a separate system.
- `(gdb) target remote TARGET` Support for serial port, TCP, UDP and custom (pipe) connections.
- ASCII based protocol (now some commands take 8-bit binary data)
- `(gdb) x/1h 0x4015bc` will send the following command packet: `$m4015bc,2#5a`.
- gdbserver could generate the following response: `+ $2f86#06`.
- Well documented and understood, used by libre (kgdb, OpenOCD) and commercial stubs and debuggers.

ARM

Halting mode debugging

- Stops normal program execution.
  - Processor clocks halted
  - But external peripherals will still run (RTC, DMA etc.)
- Provides Debug Communications Channel
  - Debugger can inject an instruction into the pipeline...
  - ... execute it...
  - ... and access a couple of special system registers.
- Target memory is accessed by executing load and store instructions (`LDR X0, [X1]`).
  - Transfers will go through MMU and caches, so accessing variables in cache will return correct value.
  - Value then transferred into Debug Data Transfer Register (`MSR DBGDTRRX_EL0, X0`).
  - Debugger must carefully manage pipe line, register bank, caches...
- Chip can also provide direct access to memory bus.
  - Usually non coherent with the processors.
  - Accessing addresses of cached variables will return wrong value.

ARM

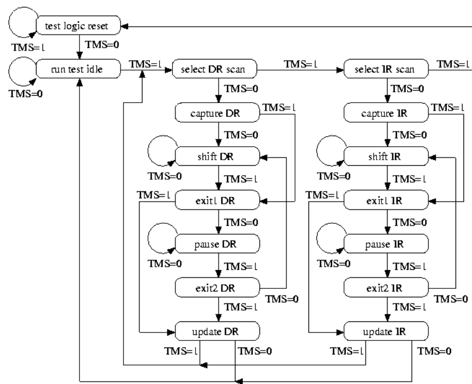
JTAG Debugger

- Also known as JTAG box, debug probe, debug adapter, debug hardware unit, protocol converter, ICE (although it's not accurate)...
- Usually expensive...
- ... but can be an FTDI dongle driven by OpenOCD!
- JTAG stands for Joint Test Action Group.
- Created IEEE 1149.1-1990 "Standard Test Access Port and Boundary-Scan Architecture".
- "Low cost" port and infrastructure for silicon testing.

ARM

JTAG Interface

- Bi-directional serial interface with data input and output, single control bit and a clock input.
- Defines two registers (DR and IR) of different lengths.
- Meaning (and length) of DR depends on current IR content.
- Modern chips contain many separate scan chains, often not connected to the “JTAG port”.



JTAG Test Access Port (TAP) controller state transition diagram

picture courtesy of openocd.org

ARM

Debug Access Port

- Gateway between re-purposed JTAG bit protocol and debug logic
- Debug hardware often visible in a special memory address space
- E.g. `(gdb) stop` requires writing 0x1 (Halt Request) to address 0x090 (Debugger Run Control Register) of the CPU debug unit.
  - Shift 4 bits into IR
  - Shift 34 bits into DR
  - Shift 4 bits into IR
  - Shift 34 bits into DR
  - Shift 34 bits into DR

ARM

Debug Access Port

- Gateway between re-purposed JTAG bit protocol and debug logic
- Debug hardware often visible in a special memory address space
- E.g. `(gdb) stop` requires writing 0x1 (Halt Request) to address 0x090 (Debugger Run Control Register) of the CPU debug unit.
  - Shift 4 bits into IR
  - Shift 34 bits into DR
  - Shift 4 bits into IR
  - Shift 34 bits into DR
  - Shift 34 bits into DR

I really wish this guy was an UART expert!

ARM

Debug Access Port

- Gateway between re-purposed JTAG bit protocol and debug logic
- Debug hardware often visible in a special memory address space
- E.g. `(gdb) stop` requires writing 0x1 (Halt Request) to address 0x090 (Debugger Run Control Register) of the CPU debug unit.
  - Shift 4 bits into IR
  - Shift 34 bits into DR
  - Shift 4 bits into IR
  - Shift 34 bits into DR
  - Shift 34 bits into DR

I really wish this guy was an UART expert!

- There are (slightly) better alternatives like ~~Single~~Serial Wire Debug port (*two* wires plus ground)

ARM

Summary

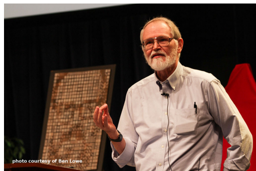
- Debugging needs support from operating system and/or hardware.
- Debug hardware is expensive therefore limited.
- Debugger can only show what it knows about.
- Describing code generated by a modern compiler is hard.
- Backtraces must be treated with limited confidence.

ARM

Closing remark

The most effective debugging tool is still careful thought,

— Brian W. Kernighan, Unix for Beginners (1979)

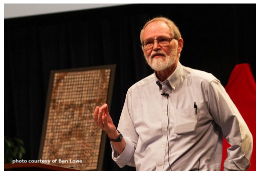


ARM

Closing remark

The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

— Brian W. Kernighan, Unix for Beginners (1979)



ARM

Thank You

The trademarks featured in this presentation are registered and/or unregistered trademarks of ARM limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.