Acid your ARM

Gorka Guardiola Múzquiz
Laboratorio de Sistemas
Universidad Rey Juan Carlos
`paurea@lsub.org`
9/1/2011

ABSTRACT

We have developed `jtagfs`, a protocol stack and filesystem which enables live debugging of an ARM machine using acid. It accesses the hardware through a JTAG interface, providing new ways of debugging which dissolve the boundaries between the kernel, user space and the loader and gives direct access to the hardware. At the same time acid provides high level abstractions to interpret the results and automate the debugging process.

Introduction

JTAG is a standard for boundary scanning, a method for testing digital circuits by means of a shift register (boundary scan shift register or BSR). The BSR is used to drive the inputs and outputs of different parts of the circuit. On each subcircuit the BSR is controlled by a TAP (Test Access Port), used to transverse the states of the BSR on each tick of the clock, load it and set its connections to the pins, input and output values of the circuit. On complex digital circuits, like a microprocessor, the BSR can be used to control separately and test different subcircuits by feeding it different chains (a sequence of bits).

While debugging some drivers in the Sheevaplug, it came to our attention that the ARM cores have a well documented JTAG interface [10]. Furthermore, the chains to control all of the models are very similar, with only small differences between them. The microprocessor includes a TAP controller and some extra debugging hardware as part of the macrocell called Embedded ICE [6] or Embedded ICE-RT [7] depending on the particular model. The JTAG interface provides access from an external external machine to different parts of the microprocessor. Through JTAG the debugger can make the processor enter debug mode, write directly to the processor registers, inject instructions with full access to the hardware and restart the processor no matter its state. Furthermore, some machines like the Sheevaplug contain a chip which in addition to providing access to the serial port on the machine has a subcircuit able to interact with the TAP controllers on the board and the microprocessor, making them accessible through USB. This chip, called AN2232C-01 [4] , is a command processor which can drive any kind of serial interface (act as an MPSSE or Multi-Protocol Synchronous Serial Engine). It can also work as an MCU host bus emulator. We will be using it as an MPSSE, so we will just call it MPSSE from now on.

All this capabilities mean that with the appropriate software it is possible to debug the kernel having facilities akin to that available on special development boards while at the same time running regular production kernels. It is even possible to debug simultaneously the kernel, user space and the loader, vanishing the frontiers and providing full access to the hardware. The problem is that the appropriate software did not exist. The software we have had access to has some limited debugging capabilities through gdb or direct access to the hardware. Porting existing software to Plan 9, while being more complicated than writing it from scratch would have been not enough because of the dependencies with gdb and its lack of generality in the interfaces it offered. Leveraging on [14] and with an approach similar to `rdbfs`(4) [11] but with a twist, we have developed a general purpose a complete programmable debugging interface for the ARM machines, providing full access to the hardware.

JTAG basics

Each JTAG capable device has a number of TAPs connected in series or in parallel. Each TAP normally has four inputs, TCLK, TMS, TDI and TDO, connected as is shown in Figure 1.

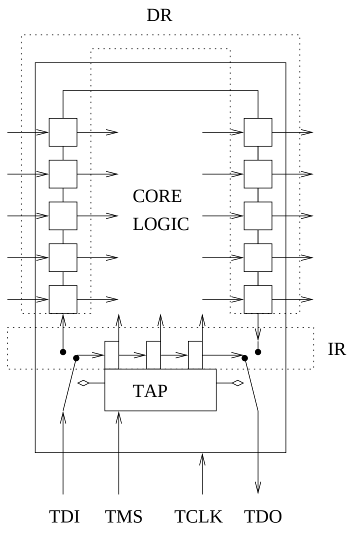

Figure 1: *Jtag enabled device*

On each TCLK down edge the system may shift the value in TDI and shift out the value to TDO depending on the state of the state machine of the TAP, depicted on Figure 2, with the transitions controlled by the value in the TMS input. There are two shift registers normally connected in parallel, the data register (DR) and the instruction register (IR). The DR is the BSR, and the IR controls what happens. Which of them is connected to TDI and TDO depends on the state of the TAP controller which also sets when the instructions or the data start being active and connected to the chips or the output pins. TAP controllers can be chained in series, with the data registers and the instruction registers concatenated. There is an instruction to disable a controller which can effectively turn off one controller so that data can be shifted in separately.

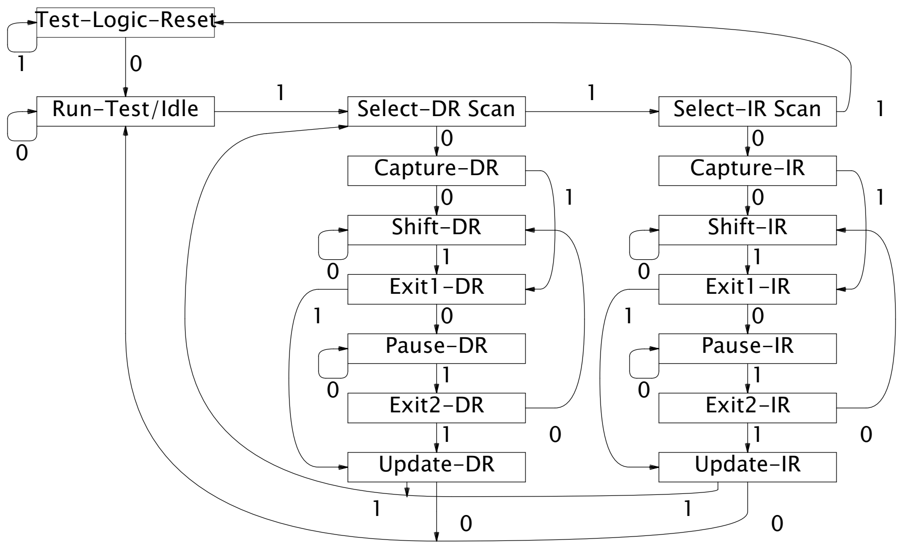

Figure 2: *JTAG TAP state machine, input is TMS*

Sometimes controllers are also chained in parallel where there are different chains and an instruction is used for chain selection. For example this approach is taken by the ARM to select between the different circuit modules. Each chain provides a length for the instruction register and the data register.

The instructions supported by the IR may vary among models with some of them being mandatory. For more details on this, see the JTAG [10], though the details of what is implemented and what instructions are supported are actually detailed in the ARM manuals.

The instructions supported by all ARM machines are (there are some minor differences in semantics):

- SCAN_N is used to input a chain number.
- INTEST is used to set the chain number input with SCAN_N.
- IDCODE is used to detect the chip and puts a special ID value in the DR.
- BYPASS disables the TAP, putting a 1 bit shift register between input and output.
- RESTART restarts the processor after it entered debug state.

Architecture of jtagfs

Figure 3 depicts the architecture for the `jtagfs`.

The first part which needed to be implemented was the access to the MPSSE in the FTDI chip. There was already partial support for the FTDI chip in `usb/serial`(4) (for more details see [12] and [1]). We completed the support for the FTDI configuration protocol, but kept the MPSSE support itself outside of it. The `usb/serial` is already complicated enough. We set the usb serial chip with the minimum possible configuration (set the interface to be MPSSE and the latency timer) and made it serve a file called jtag which serves as conduit to communicate through it. The `jtagfs` uses this file to send commands to the MPSSE.

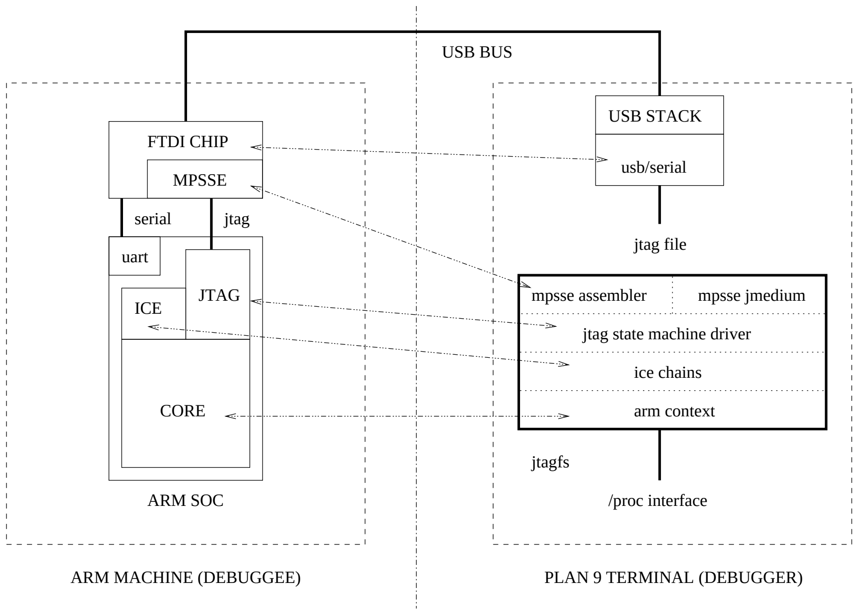

Figure 3: *Jtagfs architecture*

The next level of abstraction is the `JMedium`, a data type and a set of operations which abstracts the details of the driver for the JTAG. For now, we use an MPSSE implementation for the `JMedium`, but others may be implemented in the future. Using it, we just drive the TDI, TDO, TMS with respect to the TCLK. The interface uses buffering and takes in account there may be more than one TAP connected in series, though all but one (the current TAP being drived) are disabled.

We also wrote a state machine driver, which given an origin state and a destination state calculates the minimum distance path between them. Given the number of states, a static table could be precalculated to do this when compiling, but it was not done for simplicity, given that most of the time is spent waiting for the USB communications anyway.

The next level is the ICE chains layer, which takes care of the strange bit endianness of the JTAG chains on the ARM (some of the chains are bit order inverted and byte shuffled), and converts between that endianness and the local (debugger) machine byte order.

Built on top of the ICE chains, there is a level which interacts with the ARM processor, making it enter and exit debug mode, saving and restoring the registers and injecting instructions.

Finally, a `proc`(3) interface exports the processor registers in a manner similar to `rdbfs`(4). A small library for acid and a little modification to `attachproc` in the `mach`(2) library to make the kernel registers file, `kregs`, writable makes it possible to switch modes and manipulate the processor at will from acid.

Driving the MPSSE

The MPSSE itself is quite a complex device which can drive any kind of serial interface. It is programmed through a small machine language which can output bits in different bit endianness, driving TMS, TDI and TDO on down or up TCLK edges. We started outputting the machine commands directly, but, specially while debugging it was quite complex to keep track of all the details of the MPSSE. We ended writing a small assembler for the MPSSE instructions, which we called `ma` and assembling them on the fly. `Ma` is a good name because we will never really have `.m` object files (for which the letter m is already taken), so there will not be another assembler with this name, at least in kencc. An example of the instructions can be seen next:

```
DataIn EdgeDown LSB 3
DataIn EdgeDown LSB B3
DataOutIn EdgeDown EdgeUp LSB 3 0x42 0x34 0x56
DataOutIn EdgeDown EdgeDown LSB 3 @
DataOutIn EdgeDown EdgeUp LSB B3 0x42
DataOutIn EdgeDown EdgeDown LSB B3 @
TmsCsOut EdgeDown MSB B0x7 0x7
TmsCsOut EdgeDown LSB B7 0x7
TmsCsOutIn EdgeDown EdgeUp LSB B0x7 0x7
MCURd 0x34
WaitIOHigh
AdaptClkDisab
Div5ClkEnab
Loop
SetBitsL 0x32 0x34
```

`MSB` and `LSB` mean most and least significant bit first, BNN means a number of bits (whereas the count by itself means a number of bytes) and @ is used as a parameter when assembling on the fly as a placeholder for the data (passed as another parameter).

Assembling on the fly has proved to be a very good approach for debugging and testing, providing a low level sniffing interface. In the first prototype, we hardcoded the values using constants and some functions. Each time we found a bug in our interpretation of the MPSSE instructions, we had to fish bugs on every place where they happened. Also, the assembler itself may be useful for other applications using this chip for driving other (for example SPI) interfaces.

ICE chain support

There are several chains on each ARM machine, which provide access to different capabilities of the chip. There are minor differences among them, and some of the chains are present on some chips and not on others. We have interacted mainly with three chains.

Chain 1, is used to inject instructions to the ARM core. Special care needs to be taken with the clock. Basically, when the processor is in debug mode (which is when an instruction can be fed to the core) it runs on a slower clock. As a consequence, whenever an instruction to interact with external hardware, like RAM or a peripheral, needs to be executed, the processor must run it using the faster clock. Then it falls back to the slow clock driving the debug mode.

Chain 2, is used to access the debug registers, most of which can be accessed normally from inside the core. This registers enable hardware vector catching (enter debug on an exception, including reset), breakpoints (entering debug mode based on an address being executed) and watchpoints (entering debug mode based on an address being read or written) and instantly entering and exiting debug mode.

While chain 1 and chain 2 are well documented and seem to be the same on all the latest ARM cores, Chain 15, which provides access to the MMU, seems vary more from model to model. From what we have seen, there are two families of ARM with respect of Chain 15, the ARM 7 family and the ARM 9 family. In any case, we could not made Chain 15 work on the feroceon, so instead we have pushed MCR and MRC instructions. This approach is more portable and without any drawbacks. Using them we added the MMU state to the observable state of the processor. This state is read only at the moment, though this can change in the future.

ARM interaction

The ARM processor interaction code has two different levels. For example, there is a function, `ARMgofetch` used to inject an instruction into the core. This instruction is pushed into the pipeline and goes through the five states (fetch, decode, execute, access an writeback). At this level, one has to be careful what state the pipeline for the instruction is for. To abstract the pipeline we wrote some other functions (for example `ARMgetexec` and `ARMsetexec`) which shift in the instruction, inject NOPs and read and write the data when the pipeline is in the right state. They also make sure that after the pipeline is full until the instruction finishes.

We found that the litmus test to find whether the whole system works is if the processor is able to run again after going into debug mode. All the context for the Arm needs to be perfect. In this respect, we found two difficulties while implementing `jtagfs`. The first one is that even if the PC does not need to change, the processor will not start if the register is not written to. The second difficulty, is that interrupts need to be disabled while in debug mode. If they are not disabled, bad things will happen. An interesting consequence of this is that if while in debug mode something improper is done, like access a non mapped address, when we start the processor again, an interrupt will fire that will most probably crash the system.

Endianness

Endianness in the JTAG is tricky. There are two interfaces, the `proc`(3) on one side and the JTAG on the other side and both need to be honored. The `proc`(3) interface should be in ARM endianness (little endian on Plan 9), whereas the JTAG has special bit ordering, which is different for every chain, but at the same time the little endianness of the ARM needs to be respected. The approach we have taken is that the registers, which need to be modified by jtagfs are translated to host order at the interface (`proc`(3) and chain interface). On the other hand, other data passes through without going into host order.

Filesystem

The standard proc filesystem is used to export processes. The model was extended by `rdbfs`(4) and the `-k` flag for acid to provide access to the segments and registers of a running kernel. `Jtagfs` extends this model even further, exporting more registers (in particular exporting the MMU registers) and mapping memory outside of the segments of the binary.

The MMU registers can be accessed using the `regs` file, just after the `Ureg` and floating point registers (if there are any). To take advantage of them, the jtag acid library uses the undocumented `map()` builtin for acid to extend the register map. This approach makes it possible, when the processor is stopped, to access the memory mapped MMU registers from within acid.

The other extension implemented is that the acid library also uses `map()` to extend the memory mapped for the data segment to all the memory starting from KZERO. This approach lets us access memory outside the kernel segments, like `Mach` and the page tables.

Thanks to these extensions, with very little code, it is possible translate from virtual addresses to physical addresses by looking them up in the current page table.

Debugging

Debugging the `jtagfs` was a challenging activity. Running wireshark under linux to capture the USB dialog of OpenOCD proved invaluable in the first stages, specially to understand the finer points of the timing of the state machine and bit endiannes which is unclear in the documentation, even with the application note clarifying it [5].

Another thing we found invaluable was the verbosity flag controlled by a different character at each level of abstraction, (similar to what the compilers do in Plan 9) with the lowest printing the MPSSE assembly and the highest printing the ARM context when entering and exiting debug mode. `Jtagfs` can print any of its levels of interaction, which makes it simple to debug new devices and can be interesting for anyone willing to know more about JTAG on the ARM, which has some dark corners and rough edges (specially the bit ordering or the timing of the state machine).

Experience

While it is slow when reading or writing big amounts of data, mostly because of the roundtrip of the USB protocol, it is still quite usable for regular debugging. It could be made faster by batching together bigger chunks of data or by caching recently accessed data. Both have important drawbacks, considering that reading and writing may have ordering constraints (for example when accessing memory mapped devices), which is why we did not implement them.

`Jtagfs` has showed its power when using it several times. For example, after programming it, we found that just after stopping the core with Plan 9 on it, it would reboot no matter what we did. After some poking and probing, we learnt that it was the watchdog device rebooting the machine when the processor was stopped. Just by writing some acid code, we were able to to disable and reenable the watchdog as needed. Another interesting experience was debugging some code for traps that had failed to work for a long time and we did not understand why. It turned out that in the end we were using an instruction which was not supported in the machine, but what had stalled us for days was debugged in a couple of hours using `jtagfs`.

The most important feature we have missed when using the device are more breakpoint and watchpoint units, which would make debugging simpler, but this is a hardware problem outside of our control. The number of units is also dependent on the core. Software breakpoints in the kernel could be implemented, but with the caches and the pipelines interactions they would probably be quite a feat to get right.

Related work

There are several programs to interface JTAG providing a backend to gdb, for example OpenOCD [2] or the Blackfin Uclinux gnu toolchain [3]. There are also developer boards and closed software like that of [9]. All of them, or at least the ones we have seen and used, provide at most batch-like capabilities (whereas acid and proc combined provide a fully programmable interface). Furthermore, the proc(3) interface is designed to be portable so it can be easily used from any other programming language and operating system, providing a simple portable interface, whereas the interfaces provided by programs like OpenOCD (OpenOCD provides a gdb commands telnet server) are designed to be used with gdb and not as general.

Future work

As it is now, jtagfs only provides Feroceon support and has only been tried in the Sheevaplug. Support has been added for the Armada, but is untried. Most of the software should work without modification on any ARM 7 or 9 as it has been written to be very portable. To support other boards the id code for the processor needs to be added and the configuration necessary to deal with the wiring of the board.

After the Sheevaplug has been running for a short period of time, the JTAG interface stops responding unless it has already been accessed, though this looks like it is a characteristic of the hardware and the same happens to OpenOCD on Linux. In any case, when the JTAG does not respond, it is detected in the identification phase and it can still be reset through the JTAG interface. As long as there is some interaction with the JTAG (it can be only to identify it) early in the boot process, the JTAG works flawlessly.

There are other capabilities of the ARM chips which can be accessed from the JTAG and which could be interesting. One of them is the Embedded Trace Macrocell or ETM [8] an instruction and data tracing interface to the processor. Another interesting capability is the DCC or Debug Communications Channel. It provides three registers to access a bidirectional serial communications channel (polled or interrupt driven) for printing and debugging using the JTAG. From inside the processor, the target sees the DCC as the coprocessor 14 using MCR and MRC. From the JTAG these registers can be accessed by means of scan chain 2.

Another interesting capability that could be implemented is to freeze the processor from within the kernel, by setting the debug registers. Then, the hardware could be accessed from the jtag port using jtagfs. We have not done this, but it should be trivial to do, as it is just setting a register. One good place to do this, for example, would be in the panic routine, so that when a kernel panics, it can be inspected.

The JTAG interface could be also used to inject a loader or a kernel as a last resort for a bricked device or to read or write the contents of the flash.

Last but not least, using /proc and acid any software running on the ARM can be debugged. It would be very interesting and not very difficult to add more support for ELF [13] symbol tables and binaries (perhaps using those of plan9ports or go) to Mach. This could enable debugging the Linux kernel or U-boot using acid.

References

1. F. J. Ballesteros, *Plan 9's Universal Serial Bus*, IWP9, 2009.
2. R. D., Open On-Chip Debugger, *Diploma, Department of Computer Science, University of Applied Sciences Augsburg*, .
3. B. U. http://blackfin.uclinux.org/gf/, Blackfin GNU Toolchain.
4. F. T. D. I. L. http://ftdichip.com, AN2232C-01 Command Processor for MPSSE and MCU Host Bus Emulation Modes.
5. http://infocenter.arm.com, Application note 205 Writing JTAG Sequences for Arm 9 Processors.

6. http://infocenter.arm.com, ARM9E-S Technical Reference Manual.
7. http://infocenter.arm.com, ARM7TDMI-S Core Technical Reference Manual.
8. http://infocenter.arm.com, Embedded Trace Macrocell Architecture Specification.
9. X. L. http://www.xjtag.com, XJTAG company.
10. IEEE, IEEE 1149.1 standard specification, *Standard Test Access Port and Boundary Scan Architecture*, .
11. B. Labs, Plan 9 man pages, *Plan 9 User's manual, Vol 1*, 1995.
12. G. G. Múzquiz, F. J. Ballesteros and E. Soriano, *Usb serial design and experience in Plan 9*, IWP9, 2010.
13. T. I. Standard, Executable and Linking Format (ELF) Specification Version 1.2, *TIS Committee*, .
14. P. Winterbottom, Acid Manual, *Plan 9 Programmer's Manual. AT&T Bell Laboratories. Murray Hill, NJ.*, 1995.