Implementing JTAG debugging solutions for custom hardware

Martin Strubel <hackfin@section5.ch>

March 10, 2012

Revision:
`1.01 / embedded world 2012`

1 Introduction

JTAG based test access ports (TAPs) have found widespread application on silicon chips and for electric testing purposes, ranging from simple flash programming solutions up to heavy weight *In Circuit Testers*. Full JTAG access to a running system has turned out to be one of the most powerful methods for debugging as well as for reverse engineering. A strategy shall be described in this article, how to easily implement a TAP for own VHDL designs, using a simple resource saving CPU soft core. Moreover, a software/hardware co-simulation and rapid prototyping method shall be explored that allows using the same software for the simulation as well as for the final hardware. In other words, the software talks to a **virtual chip** running in software on the host computer.
The presented hardware vendor independent solution is based on a free and open source tool chain, the most prominent being the GHDL simulator and the GNU Debugger (gdb) which is featured by many common CPU architectures. Nonetheless, the involved tools can handle fairly complex hardware designs and are therefore competitive with commercial simulation frameworks.

1.1 Typical development scenarios

One important standard saying that the author used to hear and say often is:

You don't want to keep debugging your debug port itself

For this reason, many firmware developers keep it simple: debugging typically occurs by using a LED, pin scoping, or usage of the `printf()` function to write out debugging messages to a serial port. Sounds familiar? However, the reader may have been confronted with scenarios where the printf method becomes ineffective or simply does not work anymore, because all serial interfaces are used up and the system does not have a display.
Another aspect is, that many debugging methods are intrusive: Like in quantum mechanics, the system may behave differently when you measure it. An effect of the 'printf probe' can be: Your program just burns a different number of cycles and has another stack memory layout when a printf is present, so for example, it might crash **without**, but not crash **with** printf, when you have a timing or stack memory bug buried in your code.
Last but not least, a scenario could be, that you develop a solution in HDL (Hardware Description Language) from scratch, like a special coprocessor for DSP operations. Before you bake this into hardware, you will have to go through extensive debugging cycles and make your development tools (software) operate smoothly with the hardware.
Let us ponder once again what we – as a developer community – typically expect from a microcontroller debugging solution. From the software side, we might want to:

1. Stop a CPU, resume program execution
2. Single step through program code
3. Read and write program counter(PC), stack pointer(SP)
4. Access I/O memory and memory mapped registers
5. Raise software exceptions/breaks using a dedicated "breakpoint" instruction

Closer to the hardware side, we would probably like to:

1. Trigger a breakpoint on a certain condition: Hardware Breakpoints
2. Count certain events
3. Read out debug registers or trace buffers

For simple to implement and least intrusive debugging, a method has been established since many years that is referred to as **In Circuit Emulation** (ICE). A CPU that is in **Emulation Mode** can accept its instructions through the TAP instead of fetching them from memory. That way, a CPU can be remote controlled from the outside. It is up to the debugger software, to save and restore the program context during a debug session so that debugging remains least intrusive. This article will be mostly dealing with the In Circuit Emulation method.

One thing that we can never completely cover is: time. When halting the CPU, other peripherals may still be running, so we are still intrusive in one or another way. This aspect is discussed later in Section 2.3.

1.2 JTAG basics

The JTAG protocol is a serial protocol like the known synchronous SPI protocol with an extra mode selection pin:

- TCK: Clock signal, rising edge sensitive
- TDI: Serial data input
- TDO: Serial data output
- TMS: Mode select

A JTAG controller has an internal state machine with 16 possible states. The JTAG states will not be discussed here, but it can be simplified, that the following basic operations can be carried out by appropriate bit sequences fed into TMS:

- Reset state machine
- Select instruction register (IR)
- Select data register (DR)
- Serially clock data into either IR or DR
- Execute an action (RunTestIdle state)

The JTAG **instruction register** has nothing to do with the native CPU instructions. It is fully TAP specific, thus independent of the attached CPU architecture.

Another property of the JTAG standard is, that devices with voltage compatible JTAG ports can be chained together by feeding their TDO into the TDI of the next chip in the chain. The internal shift registers thus concatenate into one large shift register, such that each device can be adressed by clocking in a sequence of specific instruction slices.
From the mid level software layer, a typical JTAG library allows to write an instruction into an IR and to set the appropriate DR value. An instruction written into the IR register can also be

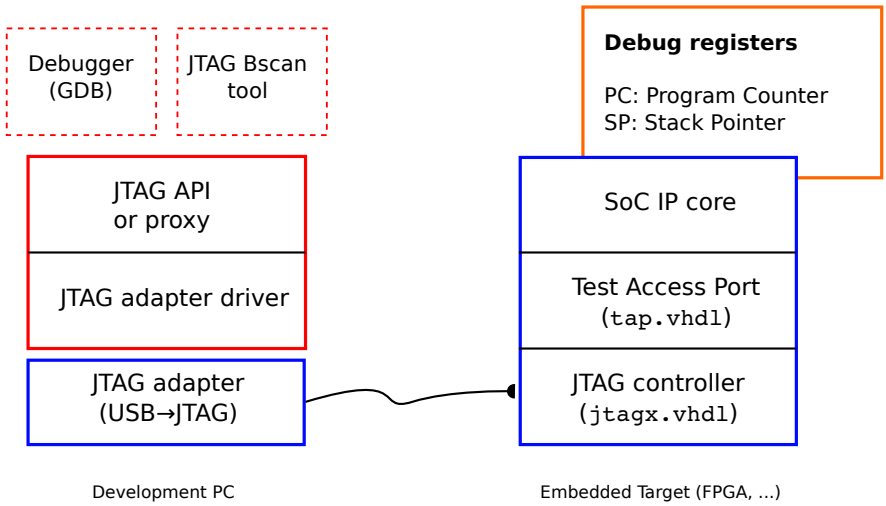

Figure 1.1: Debug TAP hardware (blue) and software (red) layers

regarded as address of a specific DR. Note that the length of the DR can vary, depending on internal TAP configuration bits or the instruction/address, for example, a CPU instruction register (EMUIR) may have a length of 8 bit whereas the program counter (PC) has 32.
Our conclusion so far is, that JTAG can basically cover full remote control of embedded logic by an appropriately designed TAP.

1.3 Existing commercial solutions

To monitor via JTAG what is happening inside an FPGA, there are existing debugging tools that are useful for debugging signals on a target FPGA, such as:

- ChipScope (Xilinx)
- SignalTap (Altera)

However, they are very vendor specific and may have restricted capabilities, due to different JTAG TAP implementations that can not be altered. There are several reasons to deploy a different solution (which will be referred to as a Soft TAP):

- Academic purposes (learning), virtual ASICs.
- SoC (System on Chip) debugging independent from vendor/chip specific libraries and TAPs for own ASIC designs
- Flexibility to enhance the TAP by complex testing features
- Software- and Hardware Co-Simulation for verification and debugging (as mainly dealt with in this article)

The drawback of a Soft TAP on real FPGA hardware is, that a few I/Os will have to be sacrificed for a ‘user defined’ JTAG port. Also, a certain complexity is added by the free choice of an appropriate JTAG tool, when it gets to real hardware debugging. However, as shown below (Section 3.2), the magic of JTAG has disappeared by now due to a few popular standard chips that have made home-grown JTAG communication convenient and simple.
Bringing a soft core into the TAP game may appear as introduction of unnecessary complexity, if you don’t require a Soft-CPU. On the other side, a few examples will later demonstrate, that using an established and resource-saving soft core pays off due to an existing and ‘approved’ debugger tool chain.

1.4 The Soft-TAP

As seen in Fig. 1.1, our proposed Soft-TAP consists of two modules:

1. `jtagx.vhdl`: The JTAG state machine implementation and controller
2. `tap.vhdl`: A generic test access port implementation

Concerning the JTAG controller, compliance with the IEEE 1149.1 standard is a must. However, there is some freedom of implementation on the logical TAP layer. To remain as generic and CPU architecture independent as possible, only a few pins and registers that are specific to a CPU are implemented. Before we list those, it makes sense to recapitulate, what is expected from a simple CPU:

- It has at least a program sequencer, i.e. can read instructions from a memory or pipeline and execute them
- It has a register or memory storage, like a stack, thus it has at least a stack pointer

In order to implement the debug primitives listed in Section 1.1, a CPU TAP needs a few registers/signals, as shown in the block schematic Fig. 1.2:

1. `emurequest` (input, level sensitive): TAP requests from CPU to go into emulation mode
2. `emuack`/`emurdy`: Acknowledge and Ready signals. Represented by status bits in `emustat`
3. `emuexec` (input, one clock wide pulse): Execute emulated instruction in emuir register
4. `emuir` (input, width is the CPU's maximal instruction width): Emulation Instruction register
5. `emuctl` (input): Emulation control register
6. `emustat` (output): Emulation state register, CPU signals TAP (i.e. user) if it is in emulation state, etc.
7. `emudata` (in/out): Data register for exchange of data between core and TAP
8. `dbgpc` (output): Current Program counter which can be read out while CPU is operating

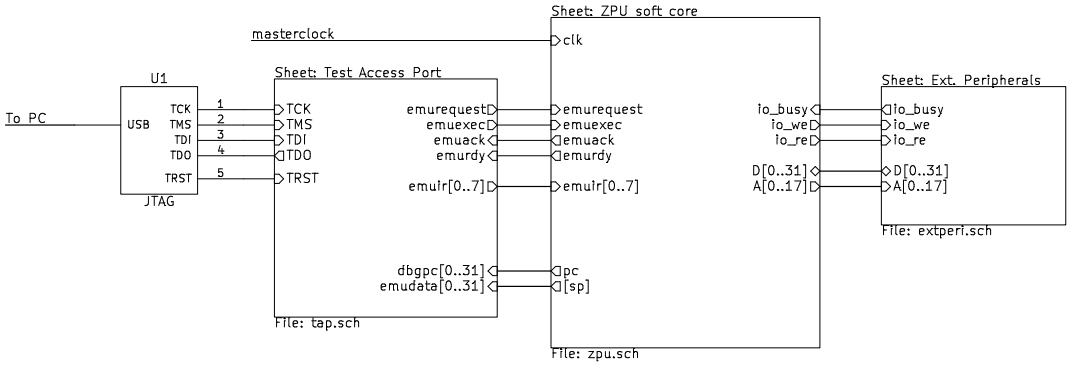

Figure 1.2: Block schematic of signals between TAP and core

The control and status signals are directly mapped to JTAG data register bits according to `tap.vhdl`. So they are modified directly via JTAG. This configuration might appear complex for debugging, but even if a CPU is not desired, it is flexible enough to control and monitor a simple state machine as well.

2

Debugging a virtual CPU

To demonstrate the power of a home made JTAG TAP, the author has chosen the free OpenSource ZPU soft core from Zylin ([zylin] Appendix A.1) and enhanced it by the above interface. The ZPU is a stack based (zero register) CPU which uses very little FPGA resources and can run at high speeds compared to other soft cores. It has a very minimal set of instructions and a simple non-pipelined CPU, therefore it is optimal for implementation of a 'lean and mean' debugging interface. The most intriguing fact however is, that it features a fully functional GNU toolchain port. For demonstration purposes, the **Zealot** variation of the ZPU is preferred, because it includes a simple trace module and a default test bench environment. Since a design never works from the first spot, simulation is a must during VHDL development.

2.1 Common strategies

The typical test bench approach is:

1. Simulate simple entities using hand coded test bench (VHDL) by signal stimulation according to the "standard scenarios"
2. Enhance your test bench by approaching the next parent hierarchy layer in your design
3. Find errors and missing test scenarios and re-iterate from (1)

This procedure turns out to be very tedious. Plus, the simulation still has a clean room character, since the software (the JTAG library) is not really interacting with the simulated hardware. Typically, data files and sequences can be output from the software into a file and read back as 'test vectors' from within VHDL, but this method still has a static character and makes full coverage complicated. Therefore, the focus is set on an interface of the real software (that would work with the hardware, likewise!) to the **simulated** hardware. This is not only interesting for academic purposes, but also saves time with industrially required verification of software-hardware interaction.

2.2 Software/Hardware Co-Simulation

A standard designed to allow software routines to interact with a hardware simulation has been defined in the Verilog Procedural Interface (VPI), or former known as PLI. Likewise, there is an interface for VHDL, called VHPI. Among a few professional and expensive tools, the free VHDL simulator GHDL offers such an interface as well.
So far, all hardware simulation aspects are covered by the above 'common strategies'. That means: up to now, we can fully simulate a virtual CPU consisting of:

1. A CPU soft core and simple program read from RAM
2. A Test access port with signals to/from the core
3. A JTAG port with 4 signals

Now as we wish to make a client software talk to it, the client hardware side (from Fig. 1.1) will have to be mimicked in VHDL as well. So there is left to develop:

1. A virtual JTAG adapter (VHDL source)

2. A driver layer (C source) sitting between generic JTAG library and virtual JTAG adapter

Since many real world JTAG adapters are plugged into USB and use a FIFO chip, it makes sense to implement a software FIFO interfacing with the VHDL simulation. The detailed technique for this simulation method will not be covered here (for more details, see [ghdlsim] Appendix A.1 or [lm] Appendix A.1 for german speaking readers), but a few important things are summarized here that need to be done from inside the simulation:

1. Do the I/O to the virtual driver via Unix pipes (i.e. a file), or
2. Start a thread from inside the VHDL simulation to listen to a TCP port while the simulation is running

Mainly, the second method was explored by using the netpp network library ([netpp] Appendix A.1), as this allows to easily create network based devices as well as a real world device hardware description. In terms of the above scheme in Fig. 1.1, the 'hard' modules become the following 'soft' modules in the virtual world of simulation:

- JTAG adapter $\rightarrow$ VHDL extension module containing FIFO, and threaded netpp server
- JTAG driver $\rightarrow$ netpp client

Due to the networked nature of netpp, virtual pins and other entities can be constructed such that a fully virtual FPGA board can work in a distributed way, i.e. the simulation can run on a powerful mainframe while the software acts on a desktop computer.
Moreover, the XML device description language allows to quickly prototype addressable registers for programmable logic without much coding effort. The software to access registers of virtual (as well as real) silicon is generic on the client side, the server side (possible embedded) part is **generated** from the XML description and potentially user-coded handler routines.

2.3 Real Time Aspects

One obvious problem was mentioned in the introduction of this article: When a real time process is interrupted or intruded another way, the timing may change such that the original problem can not be debugged. In a simulation again, the virtual logic might not run as fast as in real life, when the emulation of the logic takes much processing power. In a clean room simulation where the real time is simulated as well, this is not a problem; the simulation output will present real time values. However when asynchronously co-simulating with software, the hardware simulation may run too slow to keep up with software events. Therefore it may be necessary to introduce more real time controls, for example:

1. Introduce calls to `sleep()` in the VHPI extensions when the software is not interacting with the simulation
2. Introduce full software-based timing-control using counters
3. Use a specific (virtual) Throttle pin to change the speed of the simulation during runtime

Detailed elaboration of these techniques is again not the scope of this document, please refer to [ghdlsim] Appendix A.1 for details.

3

The Real World: Hardware

Once the simulation is verified, the next step is typically taken by connecting the design to real I/O, such as data acquisition devices, sensors, etc.

3.1 Test Access Port (TAP) Implementation

As mentioned above, a typical approach is to use the predefined `jtagx.vhdl` and `tap.vhdl` implementation. However, there are various ways to do it, depending mostly on the set of software and tools you own or you wish to invest in. Also, you may wish to use an interface different from JTAG, like a UART with a simple register protocol, while still using the same TAP architecture. This section gives you a few hints about implementation options.

The TAP implementation, i.e. the above mentioned VHDL source files are found in a ZPU development branch [tap] Appendix A.1.

3.1.1 Generic 'ASIC style' TAP

The standard technique the author has explored is the raw JTAG TAP implementation, which is considered the most powerful one for development, plus probably the easiest method if you have an existing powerful JTAG tool chain. What you would basically have to do, to get your SoC test bench set up:

1. Implement TAP interface to your SoC
2. Use `tap.vhdl` and `jtagx.vhdl` to implement the JTAG controller
3. Reserve the JTAG pins on your FPGA
4. Talk to the SoC using your (favorite) JTAG tools

3.1.2 Xilinx specific primitives

When you do not have a JTAG tool chain ready, this solution may have some advantages, because you do not need to invest into an extra JTAG adapter, or worry about connections. The Xilinx BSCAN primitive allows you to use a few USER specific JTAG instructions (IR) in order to communicate with a TAP. However, the JTAG registers are not the same, you can only use the USER[1-n] instructions from the predefined JTAG interface. This would require you to:

1. Implement register Protocol between BSCAN USER IR and TAP
2. Implement a software driver for this vendor/chip specific TAP

A disadvantage is, that the BSCAN primitives vary between chip generations, so a BSCAN primitive based solution is specific to a chip family. The author has successfully implemented a TAP variant of the above for a ZPU implementation on a Spartan6 using a ICEbearPlus JTAG adapter for real hardware debugging and program downloat at runtime.

3.1.3 JTAG processor

This is basically what was implemented in the Co-Simulation paragraph (Section 2.2), but ported to real hardware. All VHDL modules that are not part of the test bench remain unchanged, a USB hardware FIFO controller chip is used for the interfacing. Ideally, such a solution could be turned into an efficient USB JTAG controller. However, for standard purposes not requiring maximum speed it is more economic to use popular solutions like the FT2232 based adapters listed below (Section 3.2).

3.2 JTAG adapters

Although the JTAG standard is well defined (IEEE 1149.1), the connector and the software functionality are not. Therefore, a large number of different implementations exist, mostly customized for specific platforms. Among many proprietary solutions, a few adapter types have become popular in the low cost area:

- Parallel port 'bit banging': ByteBlaster, Wiggler, etc.
- USB-JTAG adapters based on Cypress FX2 (Nexys eval boards) or FTDI chips (ICEbear, JTAGkey, etc.)

Parallel port adapters are somewhat disappearing off the scene, since most PCs no longer have one. Also they are slow and signal integrity is sometimes questionable. The FX2 implementations are often found on Xilinx FPGA evaluation boards and there are many free software implementations. Even more popular and simple to program are the adapters based on FTDI's FT2232 chip family. The author has developed various JTAG hardware programming and debugging solutions (in particular for Blackfin, SHARC and ARM) based on this chipset.

3.3 JTAG Software

To efficiently use JTAG technology, a large number of software and libraries is available on the market. There are many expensive solutions, here, only a few low cost or OpenSource solutions are listed:

1. urJTAG: An opensource initiative to support various popular JTAG adapters (such as listed in Section 3.2) : See [urjtag] Appendix A.1
2. xcs3prog, xilprg, cableserver: Xilinx specific open source programming tools supporting the previously mentioned USB chipsets
3. goJTAG: graphical Tool in Java, good for learning purposes (BSDL – Bounday Scan Description Language), manual hardware diagnostics and Boundary Scan featured pin manipulation: [gojtag] Appendix A.1
4. OpenOCD: Free open source debugger for various ARM platforms: [openocd] Appendix A.1
5. bfemu JTAG library, gdbproxy: powerful emulation library (partial open source) with commercial support: [bfemu] Appendix A.1

4

Example Applications

This section outlines a few debugging scenarios, based on the ZPU soft core. The reason for integrating the ZPU into the design is, that there is a GDB (Gnu Debugger) adaptation which allows full interactive remote control of ZPU-enhanced custom hardware. There are in fact three participating software modules:

1. JTAG emulation layer (virtual driver or USB JTAG driver library)
2. gdbproxy: Small server 'agent', translating between JTAG driver and GDB. The gdbproxy can therefore emulate one or more remote targets by providing several target backends for various CPU architectures or interfaces.
3. GDB: The GNU debugger, connecting to the gdbproxy via a remote protocol (typically TCP)

As mentioned before, the only difference between simulation and real hardware is the JTAG emulation layer and the virtual hardware. Instead of plugging in the JTAG adapter to the real target, the simulation executable is run to create the virtual silicon.
A note about the embedded 'soft' firmware, i.e. the program actually being executed on the soft core: The actual program consists of a bootstrap code in assembly and a main program written in C (as shown below). This code is compiled into a binary executable in ELF format and converted into bit value strings in VHDL for Block RAM initialization. This is then compiled into the simulation or the binary BIT-File that the FPGA is booted with.

4.1 ZPU Simulation Examples

4.1.1 A simple, fully simulated debugging session

First, the virtual hardware and the virtual JTAG driver needs to be compiled into a simulation executable. This is based on the GHDL powered HW/SW co-simulation technique described shortly in Section 2.2.
Then, the simulation executable is launched. In our example design, the ZPU soft core is taken immediately out of reset and begins to boot the built-in main program shortly after the simulation starts. To halt the CPU automatically inside the `main()` routine without having to trigger an external emulation request, we insert a breakpoint assembly instruction as shown below.

```
int main(void)
{
    uint32_t v;
    asm("breakpoint"); // Stop right here
    lcd_init();
    ...
}
```

If we now launch the JTAG debugger tool chain by starting gdbproxy and gdb, we will see that the ZPUs program counter is stuck right after the breakpoint command and the ZPU is actually in emulation mode (this is triggered by the breakpoint instruction).
When gdbproxy starts, it will detect the virtual JTAG chain and output some information:

```
<...>
notice:    Setting clock to 10 MHz (wait cycle: 2)
notice:    Detected 1 device(s)
notice:    Selecting CPU 0
notice:    gdbproxy: waiting on TCP port 2000
notice:    gdbproxy: connected
notice:    chain[0] Found device 'Zealot/section5 variant'
```

The verbose output reports that it is listening on the local port 2000 for incoming debugger connections. Once the connection is made, the device is being detected and reported.
The ZPU Gnu Debugger is started by providing the main executable as argument:

```
user@devpc:~/src/vhdl/soc/exec$ zpu-elf-gdb main
```

GDB will read all the debugging information from **main**, but has not been told yet how to access the target. Therefore we will have to connect to gdbproxy using the following command:

```
(gdb) target remote :2000
```

If successfully connected, GDB will print out:

```
Remote debugging using :2000
main () at main.c:42
42      lcd_init();
```

This is right one command after the breakpoint instruction. You can now continue the program or start debugging variables as usual.
The question might arise how the switching between simulation and hardware occurs. This is currently done by changing the search path to the driver DLL (Dynamic Link Library, one for the simulation, one for the hardware). However, runtime switches can be implemented using gdbproxy driver options.

4.1.2 Debugging and visualizing external memory access

Let us assume, we have a simple I/O device controller mapped to an external asynchronous 8 bit memory bus, like for example an LCD controller. This asynchronous memory is mapped according to the table Table 4.1 below. Note that even though our external bus may have a width below 32 bits, we use 32 bit boundaries, i.e. our addresses are integer divideable by four. For memory mapped I/O, this layout and 32 bit access is advised for the ZPU. The MSB in the chosen address space switches between internal and external addressing mode, in this case, we have an 18 bit wide address bus.

I/O Address	Name	Description
0x20000	LCD_Data	Data I/O
0x20004	LCD_Cmd	I/O command register
0x20008	LCD_Control0	Control register 0
0x2000c	LCD_Control1	Control register 1

Table 4.1: Example memory map for I/O device

For complex register maps, it is recommended to write a device description using the XML device description language being part of netpp ([netpp] Appendix A.1).
From the software side, access to this I/O space is typically implemented as follows:

```
uint32_t *LCD_Data     = (uint32_t *) 0x20000;
uint32_t *LCD_Cmd      = (uint32_t *) 0x20004;
uint32_t *LCD_Control0 = (uint32_t *) 0x20008;
// Reset Controller via reset pin:
*LCD_Control0 |= RESET;
usleep(10);
*LCD_Control0 &= ~RESET;
// Write value into command register:
*LCD_Cmd = 0x3f;
// Write values into data register:
*LCD_Data = 0x01;
*LCD_Data = 0x04;
*LCD_Data = 0xff;
```

Note again that the above code is run on the soft core, i.e. the ZPU and can as such be single stepped through via JTAG. However, the same memory access sequence can be run interactively through the GNU debugger. This leads to the two typical strategies for debugging:

Code stepping using Breakpoint and SingleStep method

Assuming, the above code is put into a function `lcd_init()`, we can set a breakpoint in this function using the GDB command

```
(gdb) break lcd_init()
```

When the program is run using the **continue** command, it will halt once it has entered the `lcd_init()` function. We can then step through the above code using the **next** command. For all details in how to examine data and debug executables with gdb, please refer to the many GNU debugging tutorials found on the internet or the built-in GDB help.

Interactive access using GDB commands/scripts

To interactively manipulate data within gdb, and visualize what is happening on the lowest signal level, we start the simulation with the wave output option. Since the simulation is continously generating wave signals, we can meanwhile start gtkwave and keep pressing the **Reload** key combination to update the current wave display. A typical debug session then looks like in Fig. 4.1, note particularely the `emuack_o` signal which asserts high when we are in emulation. When we connect to the (virtual) target and switch into Emulation mode, a few values are read by GDB from the target, therefore you see some activity on the `emurdy_o` signal which becomes high when emulation is ready to accept the next instruction over JTAG.
To generate the waveform in Fig. 4.1, the following gdb script is used (comments should be omitted when copying this script):

```
# GDB script to demonstrate interactive I/O access
target remote :2000                  # connect to (virtual) target, enter emulation
set $a = (unsigned long *) 0x20000   # Set I/O base address
set $a[2] = 0x01                     # Set RESET pin
set $a[2] = 0x00                     # Clear RESET
set $a[1] = 0x3f                     # Write command
set $a[0] = 0x01                     # Write data
set $a[0] = 0x04
set $a[0] = 0xff
```

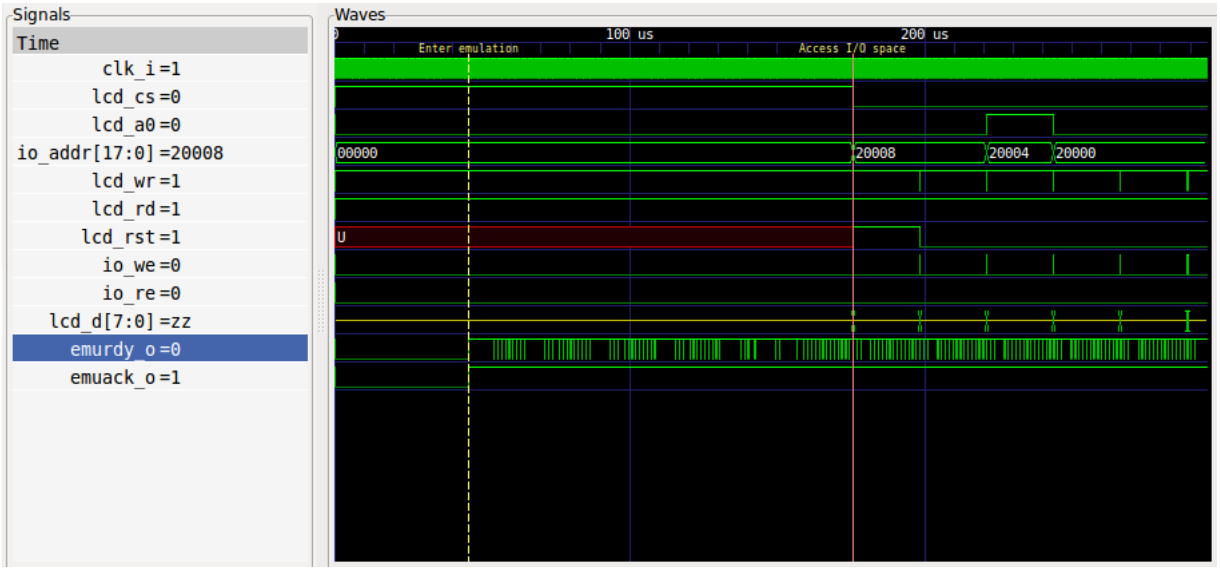

Figure 4.1: Visualization of debug session

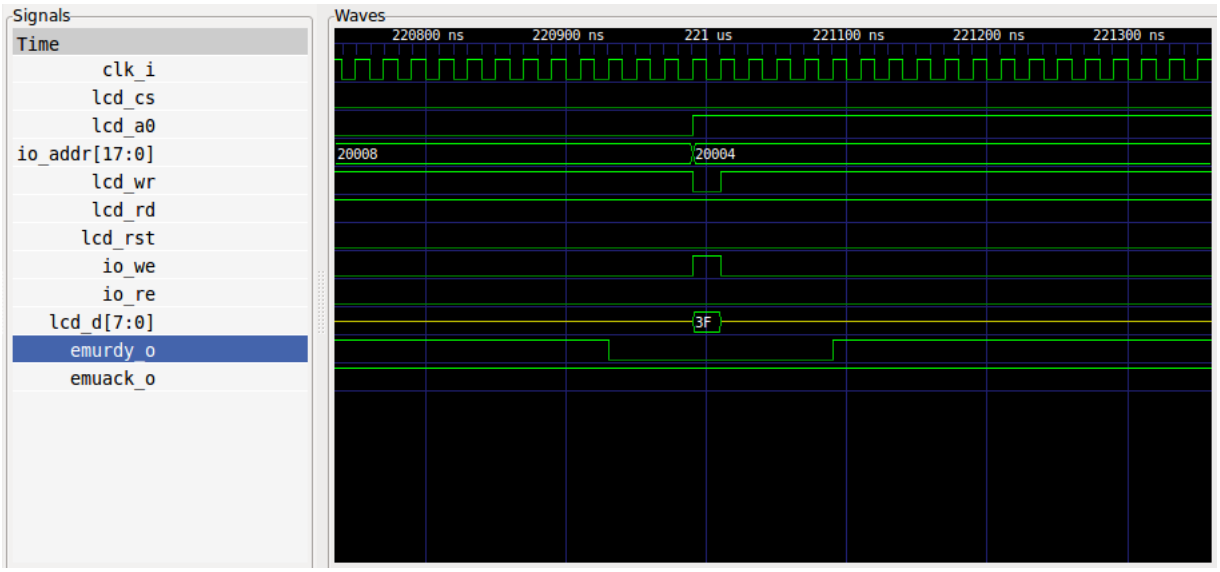

Figure 4.2: I/O access verification

The above "Write command" emulation turns out in the waveform sequence as shown in Fig. 4.2.
We can also group these command scripts into user defined functions inside GDB to trigger complex actions using one command macro. With this technique, we can verify complex software behaviour down to the signals on the simulated chip – using free OpenSource technology!
On a side note: We can see that emulation isn't much cycle optimized, in general, the 'small' Zealot ZPU variant takes a few cycles per instruction. However, finding a better optimized soft core was not the primary goal for this debugging strategy.

4.1.3 Manipulating virtual pins

The netpp server part of the simulation (which is running as separate thread parallel to the simulation) implements a few debug pins or signals, as shown in the Property table (Table 4.2) below. Those pins can be manipulated externally using various netpp clients.
For example, the following shell command (Linux or Windows command line and netpp installation) disables the Throttle signal on the virtual hardware:

```
netpp localhost Throttle 0
```

Alternatively, a python script can create an interrupt pulse using the code below:

```
import netpp                              # Import netpp module
device = netpp.connect("TCP:localhost")   # Connect to simulation server
r = device.sync()                         # Synchronize with device (get root node)
r.Irq.set(1)                              # Set IRQ line
r.Irq.set(0)
```

Property	Type	Flags	Description
`Enable`	BOOL	RW	Enable bit, High active. Not used by all targets.
`Reset`	BOOL	RW	External reset pin. High active.
`Timeout`	INT	RW	Timeout value of FIFO in real microseconds
`Throttle`	BOOL	RW	Throttle bit. If set, simulation will sleep (therefore run slower) while there is no FIFO activity
`Irq`	BOOL	RW	Simulates an IRQ pin. High active. Might not be implemented in all cores.
`Fifo`	BUFFER	RW	A FIFO buffer for communication between VHDL simulation and external software.

Table 4.2: Simulation Interface Property documentation (generated from ghdlsim.xml)

The XML source file for the pin description and a netpp server example is found in the ghdlex distribution mentioned in [ghdlsim] Appendix A.1.

Most chip designs allow pin manipulation and debugging via a specific Boundary Scan Register (BSR). The approach described here does not make use of a BSR, it directly manipulates the virtual signals within the simulated test bench.

4.2 Hardware-debugging a ZPU powered SoC

To boot, debug and remote control a ZPU SoC design running on a Spartan3 200k gate evaluation board, we have used the approach from Section 3.1.1 and attached an ICEbear JTAG adapter to custom FPGA I/Os. The behaviour of this debugging environment is the same as in the two previous examples, except that the I/O visualization is a little harder to achieve (by using a logic analyzer).
The advantage of this setup is, that the program to be debugged can be loaded into the FPGA at runtime – via emulation. This is reasonably fast, although the emulation method to write to the on-chip block RAM was not coded to be very effective. On the simulated virtual target however, the same "load" command in GDB would take much longer to complete.
More details and pictures about this setup can be found in the article from [tap] Appendix A.1.

4.3 A virtual multi core debugger

When dealing with more complex hardware consisting of several independently working logic cores, it should remembered that JTAG devices can be daisy chained by just connecting the TDO output of one unit to the TDI of the next unit. Their instruction registers are therefore logically concatenated to one long instruction register as shown in Fig. 4.3, the same applies to the data registers. To address one device in the chain, a JTAG master shifts in ones (HIGH) into the IR slices of the devices that are to be bypassed. The 'all ones' instruction is generally reserved for this BYPASS instruction, according to the standard.

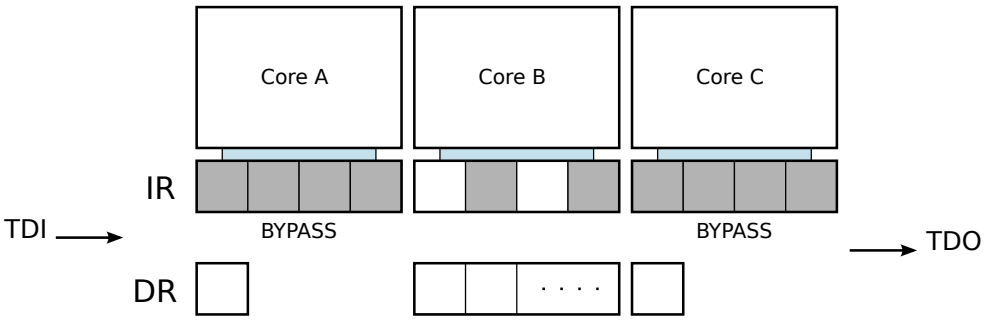

Figure 4.3: Multi core chain with selected core B

The above SoC design, once verified on the hardware, can be used to improve and debug a multi core debugging front end. So, on the hardware programmer's side, a VHDL **generate loop** statement instances several cores and daisy chains them as described above. From the JTAG front end side, the number of devices in the chain are typically detected. The devices do not necessarily have to be homogeneous, but it is eventually up to the debugger front end, what architectures it supports, plus it may need a configuration file describing the chain architecture. Typically, unknown devices are just put into BYPASS mode and are ignored. A device in BYPASS mode has a DR length of one bit, so dummy bits will have to be inserted accordingly when accessing a specific device's DR.
The VHDL code below demonstrates, how a multi core JTAG chain is instanced:

```
loop_taps: for i in 0 to NUM_UNITS-1 generate
    tapn: McpuTap
        generic map (IDCODE => x"deadbeef")
        port map (
            tck => tck, trst => '1', tms => tms,
            tdi => td(i),
            tdo => td(i+1)
        );
    end generate;
```

What is left to do is to link `td(0)` to the TDI pin, `td(NUM_UNITS)` to TDO, respectively.
This strategy is again useful to verify and debug software for multi core functionality without having to use expensive hardware.
To learn more about the low level JTAG functionality, the free goJTAG software can be used to import BSDL files, chain devices together and select instructions for each CPU by a graphical user interface ([gojtag] Appendix A.1).

goJTAG is an interesting alternative when not using a soft core or a GNU debugger. However, a BSDL file will have to be written to describe the (virtual) chip's TAP architecture.

5

Conclusion and summary

The author has presented a few possible solutions to debug VHDL SoC designs, demonstrated by means of a simple CPU soft core that uses little resources (approx 10% logic elements, 50% block RAM) on a Spartan3 200k gate FPGA.
Using interactive, JTAG powered Co-Simulation techniques based on free tools, software and hardware developers are able to verify behaviour of complex systems by open source implementations of a Test Access Port and JTAG controller. This enables a 'rapid prototyping' style implementation of a system on chip (SoC) while being independent of FPGA vendor specific tools, thus no initial investments are required.
GHDL plus its extensions (featured by the netpp library) hence allow to design a **virtual system on chip** that can be debugged using existing GDB implementations for various architectures (like the ZPU soft core). Developers may save much time by designing registers, the simulation, and the according control software in one go via **XML device descriptions** – first for the simulation, later for the final hardware, while the software will work with both.
At a later stage, a JTAG interface always turns out to be a good investment for debugging difficult hardware scenarios under real conditions.
Compared to the classic approach, the simultaneous development of hardware and software converges much faster to a stable solution, as depicted in the semi-serious doodle Fig. 5.1.

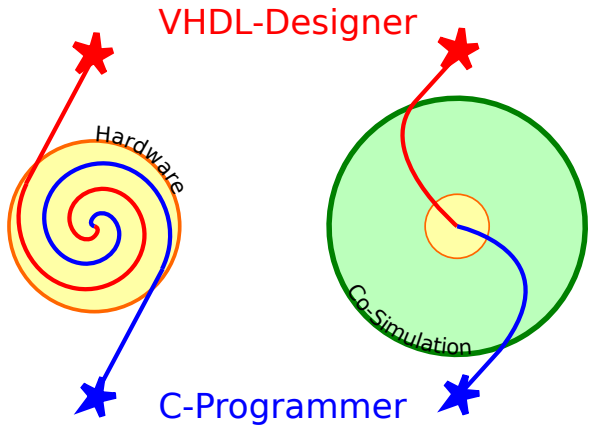

Figure 5.1: Schematic development paths of classic (left) and co-simulation (right) approach

A

References

A.1 Bibliography

[bfemu] **The bfemu library**
5/2005, section5::ms <hackfin@section5.ch>
A JTAG emulation library for the Blackfin CPU
URL: http://section5.ch/blackfin

[ghdlsim] **Using GHDL for interactive simulation (under Linux)**
10/2011, section5::ms <hackfin@section5.ch>
An introduction to Software- and Hardware co-simulation using OpenSource software
URL: http://www.fpgarelated.com/showarticle/20.php

[gojtag] **goJTAG**
Testonica
An open source JTAG test bench software written in JAVA
URL: http://www.gojtag.com/

[lm] **So tun als ob Chip** (german)
Hardware- und Software-Simulation, Linux Magazin, Ausgabe 2/2012
M. Strubel

[netpp] **The netpp library**
09/2009, section5::ms <hackfin@section5.ch>
An open source library and description language for device remote control
URL: http://www.section5.ch/netpp

[openocd] **The OpenOCD debugger**
An open source debugger for ARM targets
URL: http://openocd.sourceforge.net/

[tap] **In Circuit Emulation for the ZPU**
08/2011, Martin Strubel
A TAP implementation for the ZPU soft core
URL: http://tech.section5.ch/news/

[urjtag] **urJTAG – a free OpenSource JTAG tool**
URL: http://urjtag.org

[zylin] **The Zylin ZPU**
Øyvind Harboe
URL: http://opensource.zylin.com/zpu.htm

The author can be reached under the following address:
Martin Strubel, Im Dörfli 21, CH-8700 Küsnacht, Switzerland
email: strubel –ät– section5.ch