Distributed, virtual and real debugging of a MIPS SoC

Martin Strubel
section5.ch

02/2013

Flight plan

1. Debugging a complex FPGA design (in theory)
    - A SoC (System on Chip) example
    - *MAIS*: A portable MIPS soft core by René Doss
    - The Test Access Port (TAP): A generic debug interface
2. Virtualizing the hardware
    - 'Model in the loop' techniques
    - Making real software speak to virtual hardware
3. Demos
    - Debugging the virtual chip
    - Debugging the real hardware

The challenge

Debug this:

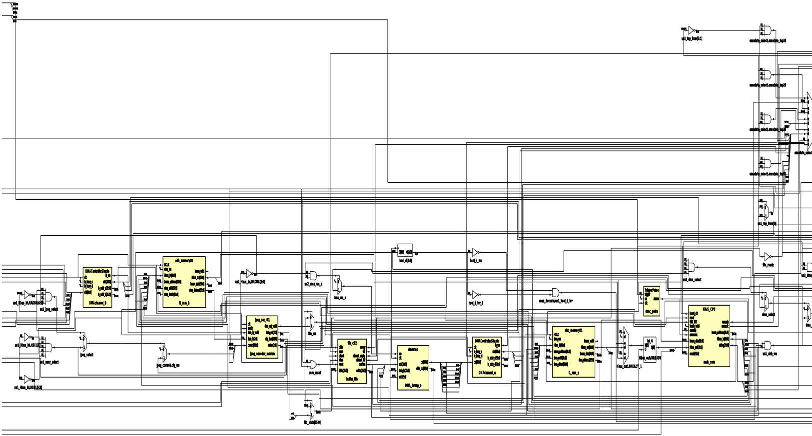


Figure: Somewhat unreadable schematic

Divide et impera

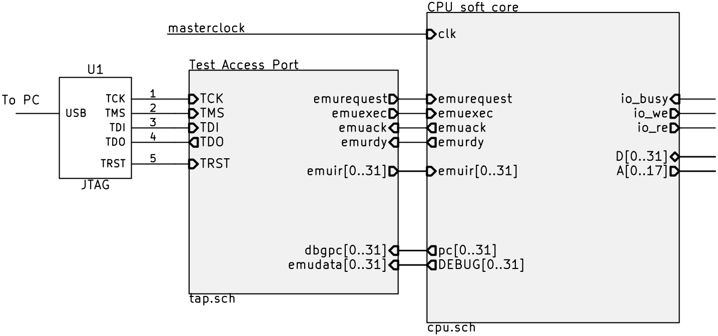


Figure: Simplified SoC schematic with Debug port

Existing solutions

Proprietary solutions from various FPGA vendors:

Signal inspection tool	Soft CPU core	Vendor
ChipScope	microblaze	Xilinx
Reveal	mico32	Lattice
SignalTap	NiosII	Altera

Table: Tool examples

- Virtualization capabilities depend on second party simulation tools ($$$-$$$$$)
- Debug port itself can sometimes not be simulated

Existing solutions

Proprietary solutions from various FPGA vendors:

Signal inspection tool	**Soft CPU core**	**Vendor**
ChipScope	microblaze	Xilinx
Reveal	mico32	Lattice
SignalTap	NiosII	Altera

Table: Tool examples

- Virtualization capabilities depend on second party simulation tools ($$$-$$$$$)
- Debug port itself can sometimes not be simulated
- No easy DIY virtualization of the hardware due to proprietary and closed libraries.

The MIPS-compatible MAIS CPU

Introducing a soft cpu core **may** speed up prototyping/debugging.

(exercised previously with ZPU soft core)

Why MIPS?

- Well-established architecture with many derivatives (Loongson SoC, Router chipsets)
- Fast, easy to implement, resource saving
- Actively maintained tool chain and emulators

The MIPS-compatible MAIS CPU

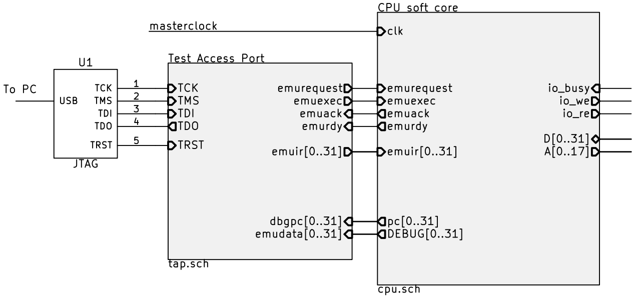
Introducing a soft cpu core **may** speed up prototyping/debugging.

(exercised previously with ZPU soft core)

Why MIPS?

- Well-established architecture with many derivatives (Loongson SoC, Router chipsets)
- Fast, easy to implement, resource saving
- Actively maintained tool chain and emulators
- *MAIS* design by René Doß:
  - Well-portable MIPS 32 bit implementation
  - Access to VHDL sources

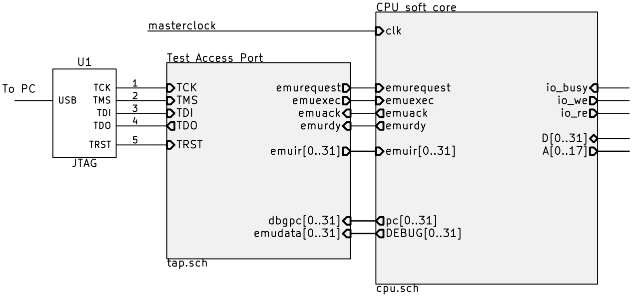
In Circuit Emulation (ICE)



In emulation mode, the CPU...

- takes opcodes from the **EMUIR** register
- executes them when it gets an **emuexec** pulse
- exchanges data with the debugger via the **EMUDATA** register

In Circuit Emulation (ICE)



In emulation mode, the CPU...

- takes opcodes from the **EMUIR** register
- executes them when it gets an **emuexec** pulse
- exchanges data with the debugger via the **EMUDATA** register

Full remote control of the CPU via a **test access port** *(TAP)!*

Debugger components

1. The developer's front end: The GNU debugger (**gdb**)

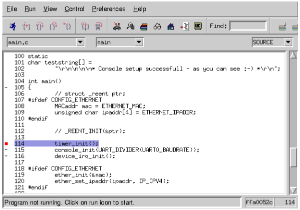


Figure: GDB

Debugger connects to back end via a TCP remote debugging protocol. Means: Distributed across networks!

Debugger components

1. The developer's front end: The GNU debugger (**gdb**)
2. The back ends:
   1. **uniproxy**: a JTAG debug server
   2. **qemu**: a MIPS CPU emulator

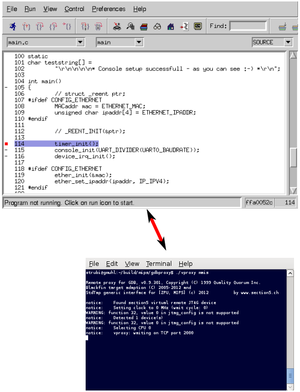


Figure: GDB and uniproxy

Debugger connects to back end via a TCP remote debugging protocol. Means: Distributed across networks!

Debugger components

1. The developer's front end: The GNU debugger (**gdb**)
2. The back ends:
    1. **uniproxy**: a JTAG debug server
    2. **qemu**: a MIPS CPU emulator
3. JTAG debugger hardware: USB JTAG adapter

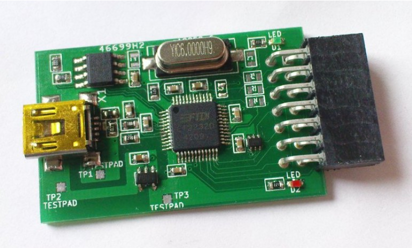


Figure: ICEbear JTAG adapter

Debugger connects to back end via a TCP remote debugging protocol. Means: Distributed across networks!

Virtualize the hardware

- qemu: software-emulated MIPS CPU – a **functional model**
  - Write C code to functionally emulate attached hardware

Virtualize the hardware

- qemu: software-emulated MIPS CPU – a **functional model**
  - Write C code to functionally emulate attached hardware
- VHDL simulation: cycle accurate – a **timing model**
  - Typically: Simulation of logical behaviour
  - Somewhat precise waveform output

Virtualize the hardware

- VHDL simulation: cycle accurate – a **timing model**
  - Typically: Simulation of logical behaviour
  - Somewhat precise waveform output

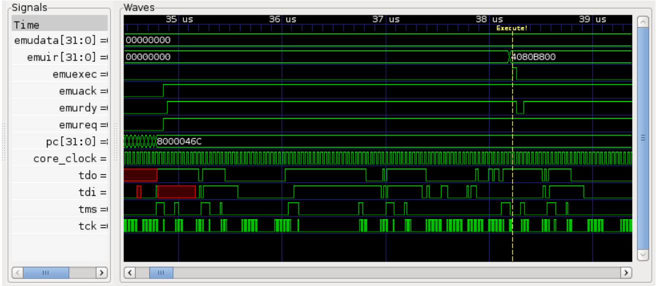
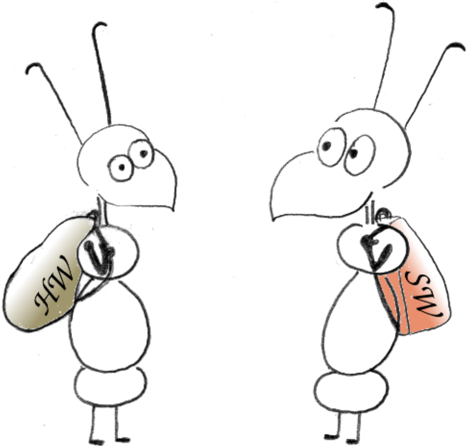


Figure: Timing accurate simulation

Make antz meet...



Drawing by Britta Schneider

Now seriously: make ends meet

Task: Make real world software speak to virtual hardware.

Result: **ghdlex** *OpenSource* simulator extension library:

- Describe virtual board in XML ⟶
- Attach virtual components in HDL design:
  - JTAG debugger
  - shared RAM
  - USB FIFO
  - I/O pins, registers

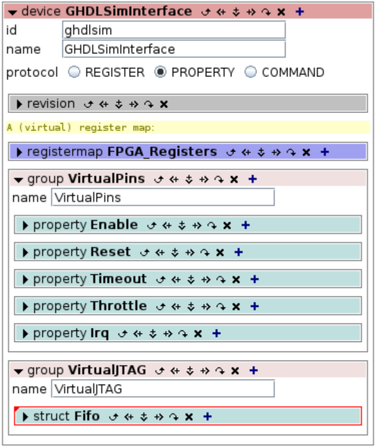
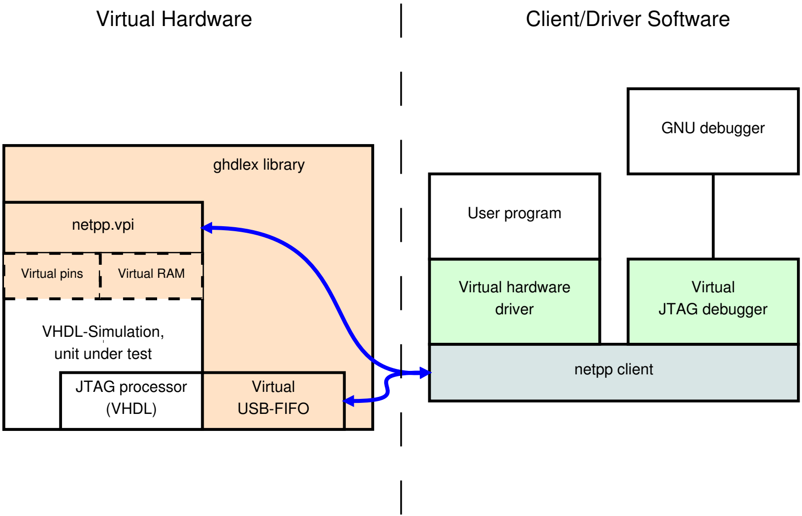
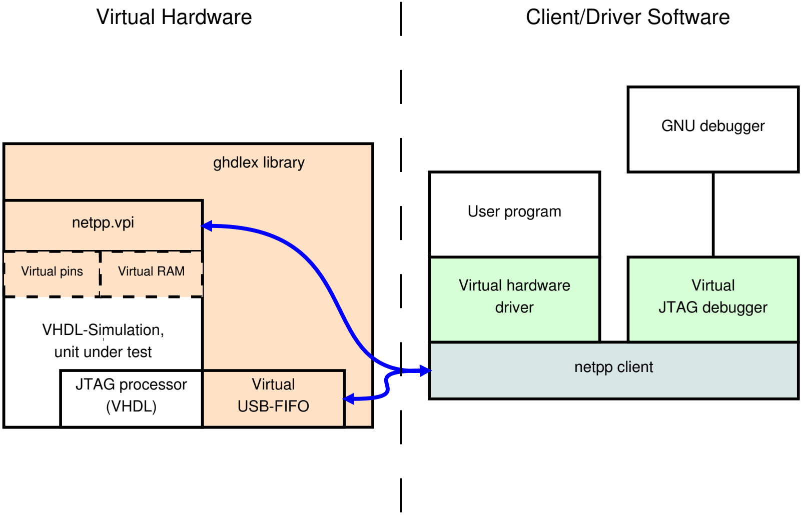


Figure: XML hardware description

Virtual Hardware

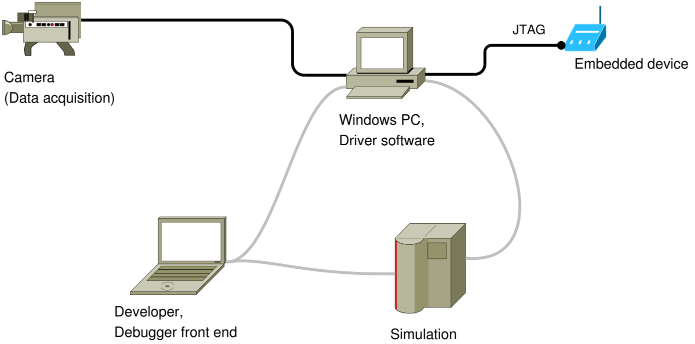


Virtual Hardware



Expose design components to the network!

Distributed processing



ghdlex speaks **netpp** (network property protocol), therefore things can run anywhere.

- HDL-Simulation on powerful main frame
- Data routing from real world software on Windows PC to simulation
- Debugger (Laptop) connecting to any of the debug servers

Now, where's the bug?

- Bug could sit:
  - .. in peripheral access (HDL design), or the CPU
  - .. in SoC firmware (Code running on CPU core)
  - .. in host (PC) software

Now, where's the bug?

- Bug could sit:
  - .. in peripheral access (HDL design), or the CPU
  - .. in SoC firmware (Code running on CPU core)
  - .. in host (PC) software
  - .. in Debugger components itself (reserve many gaelic curses)
  - .. between two human ears

Now, where's the bug?

- Bug could sit:
  - .. in peripheral access (HDL design), or the CPU
  - .. in SoC firmware (Code running on CPU core)
  - .. in host (PC) software
  - .. in Debugger components itself (reserve many gaelic curses)
  - .. between two human ears
- Avoid to introduce bugs during development:
  - Verify CPU behaviour against **qemu** (functional simulation)
  - Keep device configuration in **exactly one** XML file
  - Use Makefile rules or similar to keep source and generated files in sync ($\rightarrow$ **GNU make**)
  - Introduce detection mechanisms: ID codes or functionality descriptors (JTAG `USERCODE` register)

Hands on!

Demos:

1. Debugging the simulation
2. Debugging the hardware: HDR-60 FPGA camera kit
3. Verifying the CPU using qemu

Hardware Test Bench

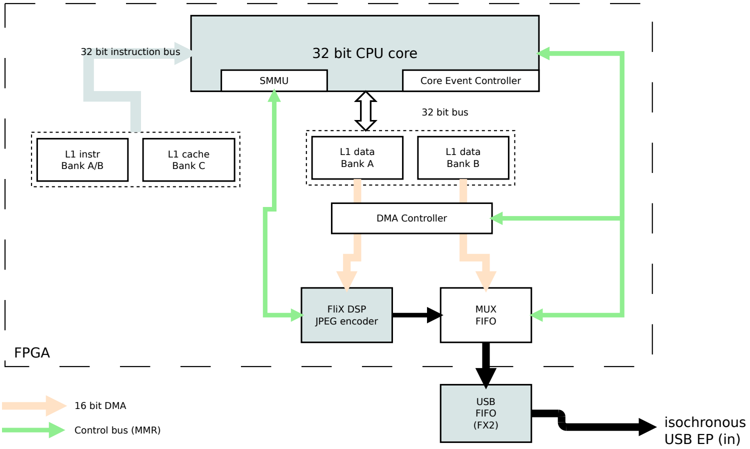


Figure: JPEG encoder test bench

Final notes

- Questions?
- More about device hardware XML description:
  → http://www.section5.ch/netpp
- Don't miss René's Introduction to his Mais MIPS core (later today in this session)

Thank you for listening!

www.section5.ch

StdTAP: The 'standard test access port'

The interface between the JTAG port and the CPU: a somewhat generic HDL library.

- Vendor independent interface ('standard' register set)
- Supports Xilinx and Lattice native JTAG components
- CPU core architecture independent
- Software support by emulation library (Python, uniproxy debug server)

TAP registers

Register	Description	Signals
EMUSTAT	ICE and CPU state	emuack, emurdy, state
EMUCTRL	ICE control	emurequest, (emuexec)
EMUIR	ICE instruction register	32 bit (to core)
EMUDATA	ICE data register	32 bit (from core)

Table: TAP registers

Actual register addressing is TAP (FPGA family) specific