Towards an Architecture for Square Pegs

Integrating the Mingle-Mangle of Firmware, Gateware and Hardware into a Homogeneous System
The WR stack

- Hardware
- Gateware
- Bridges
- Drivers
- Libraries
- Middleware
- Software

- Being our own customers means we manage: Complete Vertical Integration

This talk!
Requirements: Gateware

• Currently 6 form factors (exploder, pexaria2+5, vetar, scu2+3)
  • +4 form factors planned (exploder5, vetar5, µTCA, PMC)
  • certainly more *unplanned* form factors to come

• Various controllers (DAC, ADC, DDR, SRAM, CFI, MIL, SCUbus, spi-flash, xcvrs, audio, displays, PLL, …)

• Various HDL features (WR, EBm, LM32-cluster, TLU, ECA, function generators, build-id, reset, …)
  • Feature subselection => gateware *flavours* (per form factor!)

• Different FPGA types (arria2, arria5, spartan, virtex, …)
Requirements: Gateware
Requirements: Gateware
Requirements: Bridges

- Support many **master** access methods (host=>FPGA)
  - PCIe (memory based)
  - VME (memory based)
  - USB (stream based)
  - Etherbone-s (packet based)
  - JTAG (register based) … many bridge shapes

- Support **slave** access methods (FPGA=>host)
  - ie: virtual software slave + WB-MSI target
  - PCIe+VME via interrupts and kernel driver
  - Etherbone-m? (transparent bridging is evil)
  - JTAB+USB cannot support this
Requirements: Bridges
Requirements: Drivers

• LM32 firmware
  • no operating system

• Kernel-mode drivers
  • need to support all access methods (PCIe, USB, …)
  • must share with other WB masters

• User-mode direct-access
  • libraries (libetherbone, libeca, libtlu, …)
  • middleware (FESA, SAFTlib, …)
  • random utilities (eb-flash, eb-console, …)
  • unknown future applications (god save us all)
Requirements: Drivers

Was nicht passt, wird passend gemacht
Bringing order to the Chaos

• GOAL: Manage these layers in a scalable way

• Good news:
  • WB works great - SoC is a highly modular+scalable concept
  • The Monster design contains the hardware zoo
  • SDB very successful - core traversal easy and effective
  • libetherbone - abstracts access methods

• Bad news:
  • rampant proliferation of bad interfaces to WB in WR
    • (libetherbone, memory mapped access, hard-coded addresses, ioctl read/writes, EPICS-WB, …)
Gateware: Pretty much solved

- Every core just provides a Wishbone interface

- If a given bridge interface is present, it is a WB master
  ➡ need optional masters
  ➡ just drive cyc=0 to cause synthesis to optimise it away

- If a given controller is present, it is a WB slave
  ➡ need optional slaves
  ➡ need to automap WB address space

- Heavy use of generics and default parameters to pick features
main : monster

generic map(
    g_family      => "Arria V",
    g_project     => "pci_control",
    g_flash_bits  => 25,
    g_gpio_out    => 8,
    g_lvds_in     => 2,
    g_lvds_inout  => 3,
    g_en_pcie     => true,
    g_en_usb      => true,
    g_en_lcd      => true)

port map(
    -- WR+clock signals
    gpio_o                 => gpio_o,
    lvds_p_i               => lvds_p_i,
    lvds_p_o               => lvds_p_o,
    lvds_oen_o             => lvds_oen,
    pcie_refclk_i          => pcie_refclk_i,
    pcie_rstn_i            => nPCI_RESET,
    pcie_rx_i              => pcie_rx_i,
    pcie_tx_o              => pcie_tx_o,
    usb_rstn_o             => ures,
    usb_ebcyc_i            => pa(3),
    usb_speed_i            => pa(0), -- more USB signals
    lcd_scp_o              => di(3),
    lcd_lp_o               => di(1),
    lcd_flm_o              => di(2),
    lcd_in_o               => di(0));
Gateware: Bridge interface

pcie_n : if not g_en_pcie generate
top_cbar_slave_i (c_topm_pcie) <= cc_dummy_master_out;
irq_cbar_master_i(c_irqs_pcie) <= cc_dummy_slave_out;
end generate;

pcie_y : if g_en_pcie generate
pcie : pcie_wb
  generic map(
    g_family => g_family,
    sdb_addr => c_top_sdb_address)
  port map(
    clk125_i => core_clk_125m_local_i,
    cal_clk50_i => clk_reconf,
    pcie_refclk_i => pcie_refclk_i,
    pcie_rstn_i => pcie_rstn_i,
    pcie_rx_i => pcie_rx_i,
    pcie_tx_o => pcie_tx_o,
    master_clk_i => clk_sys,
    master_rstn_i => rstn_sys,
    master_o => top_cbar_slave_i (c_topm_pcie),
    master_i => top_cbar_slave_o (c_topm_pcie),
    slave_clk_i => clk_sys,
    slave_rstn_i => rstn_sys,
    slave_i => irq_cbar_master_o(c_irqs_pcie),
    slave_o => irq_cbar_master_i(c_irqs_pcie));
end generate;
Gateware: Slave mapping

```vhdl
constant c_top_layout_req : t_sdb_record_array(c_top_slaves-1 downto 0) :=
(c_tops_irq       => f_sdb_auto_bridge(c_irq_bridge_sdb, true),
c_tops_wrc       => f_sdb_auto_bridge(c_wrcore_bridge_sdb, true),
c_tops_lm32      => f_sdb_auto_bridge(c_lm32_main_bridge_sdb, true),
-- (... many slaves excluded for space)
c_tops_build_id  => f_sdb_auto_device(c_build_id_sdb, true),
c_tops_flash     => f_sdb_auto_device(c_arria_reset, true),
c_tops_ebm       => f_sdb_auto_device(c_ebm_sdb, true),
c_tops_tlu       => f_sdb_auto_device(c_tlu_sdb, true),
c_tops_eca_event => f_sdb_embed_device(c_eca_event_sdb, x"7FFFFFF0"),
c_tops_iiodir    => f_sdb_auto_device(c_iiodir_sdb, true),
c_tops_lcd       => f_sdb_auto_device(c_wb_serial_lcd_sdb, g_en_lcd),
c_tops_scubus    => f_sdb_auto_device(c_scu_bus_master, g_en_scubus),
c_tops_mil       => f_sdb_auto_device(c_xwb_gsi_mil_scu, g_en_mil),
c_tops_mil_ctrl  => f_sdb_auto_device(c_irq_master_ctrl_sdb, g_en_mil),
c_tops_ow        => f_sdb_auto_device(c_wrc_periph2_sdb, g_en_ow));

constant c_top_layout : t_sdb_record_array(c_top_slaves-1 downto 0) := f_sdb_auto_layout(c_top_layout_req);
constant c_top_sdb_address : t_wishbone_address := f_sdb_auto_sdb(c_top_layout_req);
constant c_top_bridge_sdb : t_sdb_bridge := f_xwb_bridge_layout_sdb(true, c_top_layout, c_top_sdb_address);
```

6-7 October 2014  Wesley W. Terpstra, GSI/FAIR
Gateware: Scalable?

- Easy to add controllers
- Easy to add bridges
- New form factor / feature blend?
  ➔ just instantiate the Monster!

(it's just a little big)
Bridges: Independence of access

- `eb-flash dev/wbm0 myfile.rpd` <= PCIe flashing
- `eb-flash dev/ttyUSB0 myfile.rpd` <= USB flashing
- `eb-flash udp/scuxl030t myfile.rpd` <= Network flashing
- `eb-flash tcp/scuxl030 myfile.rpd` <= Proxied flashing

- `eb-flash` does not need to care about how to access the device
  - it locates the flash controller via SDB and operates it
- Etherbone preserves full fidelity of Wishbone bus

- Problem: `libetherbone` cannot work in kernel drivers
- Problem: sharing with other masters - who has access?
- Problem: MSI-target (interrupts) claimed by only one process
struct MyHandler : public Handler {
    status_t write(address_t address, width_t width, data_t data) {
        printf("Action: 0x%"PRIx64" 0x%"PRIx64" 0x%"PRIx32" 0x%"PRIx32" %s\n",
            entry.event, entry.param, entry.tag, entry.tef, eca->date(entry.time).c_str());
        return EB_OK;
    }
};

int main(int argc, const char** argv) {
    Socket socket; Device device; MyHandler handler;

    socket.open(); /* Setup connection to FPGA via PCIe */
    socket.passive("dev/wbs0");
    device.open(socket, "dev/wbm0");

    std::vector<struct sdb_device> devs;
    device.sdb_find_by_identity(0x651, 0x8a670e73, devs); /* Find PCIe MSI-Target */
    mydevice.sdb_component.addr_first = devs[0].sdb_component.addr_first;
    mydevice.sdb_component.addr_last = devs[0].sdb_component.addr_last;
    socket.attach(&mydevice, &handler);

    std::vector<ECA> ecas;
    ECA::probe(device, ecas);
    ecas[0].channels[1].queue.front()->hook_arrival(true, devs[0].sdb_component.addr_first);
    ecas[0].channels[1].drain(false);
    ecas[0].interrupt(true); /* Enable interrupts from ECA */

    while (true) socket.run(); /* Pump MSI forever */
Drivers: The big Fail

• Existing solutions all fall short:
  • GSI drivers do not enumerate SDB and only export etherbone char devices
  • CERN drivers hard-code address, use memory mapping, cannot represent WB-ops with fidelity, ioctl user IO
  • FPGA=>Host access still rudimentary (*sharing*)
  • LM32 drivers share no code with linux drivers

• End-users have started rolling their own interfaces!
  => points to design failure of core framework
Drivers: Shape of the solution is known

- Support multiple access methods *in kernel* (GSI:userspace)
  - Provide access to *full* Wishbone on the attached bus (GSI)
  - Register bus and enumerate SDB=>sysfs (rubini)

- Support Etherbone-m *in kernel* for USB/net
- Support Etherbone-s *in kernel* for userspace tools (GSI)

- **Ban** memory mapped access! USB/net!
- Register+Enumerate+Access via common API (bare & linux)

- Crossbar extension for advisory locking of slaves
Drivers: Shape of the solution is known
Beware: Non-real-time components!

Moral:
• If you have hard requirements ➡ Leave hardware in control!
• Software control makes sense only in the >ms range