FROM UART TO PCIe and DMA: SELECTING CONNECTIVITY FOR YOUR FPGA-BASED SUBSYSTEM

Alexander Wirthmüller
aw@mpsitechnologies.com

#ossummit
Introduction

About me

• Based in Munich
• Diploma in Electrical Engineering

• R&D Engineer at Mynaric (FPGA-based error-correction algorithms for free-space optical laser communications)

• Founder and Director at MPSI Technologies
• MPSI Technologies: make Embedded Software development more fun by replacing repetitive tasks by model-based source code generation
Scenarios and interfaces

- FPGA module debugging
- Pre-processing / data reduction in FPGA
- FPGA result transfer to host
  - using DMA
- FPGA-SoC variant
  - with external peripherals
  - self-contained

Legend

transaction purpose >

command invocation

buffer transfer

low < 500kB/s  SPI, UART [over USB]

medium < 50MB/s  AXII, SPI

high > 50MB/s  AXI, PCIe

From UART to PCIe and DMA: selecting connectivity for your FPGA-based subsystem
Scenarios and interfaces

- FPGA module debugging
- Pre-processing / data reduction in FPGA
- FPGA result transfer to host
  - using DMA
- FPGA-SoC variant
  - with external peripherals
  - self-contained

Legend

<table>
<thead>
<tr>
<th>Transaction Purpose</th>
<th>Bandwidth</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>command invocation</td>
<td>low &lt; 500kB/s</td>
<td>SPI, UART [over USB]</td>
</tr>
<tr>
<td>buffer transfer</td>
<td>medium &lt; 50MB/s</td>
<td>AXIIllite, SPI</td>
</tr>
<tr>
<td></td>
<td>high &gt; 50MB/s</td>
<td>AXI, PCIe</td>
</tr>
</tbody>
</table>

From UART to PCIe and DMA: selecting connectivity for your FPGA-based subsystem
Scenarios and interfaces

- FPGA module debugging
- Pre-processing / data reduction in FPGA
- FPGA result transfer to host
  - using DMA
- FPGA-SoC variant
  - with external peripherals
  - self-contained

Legend

- command invocation
- buffer transfer

- low < 500kB/s  SPI, UART [over USB]
- medium < 50MB/s  AXIIlike, SPI
- high > 50MB/s  AXI, PCIe

From UART to PCIe and DMA: selecting connectivity for your FPGA-based subsystem
Scenarios and interfaces

- FPGA module debugging
- Pre-processing / data reduction in FPGA
- FPGA result transfer to host
  - using DMA
- FPGA-SoC variant
  - with external peripherals
  - self-contained

Legend

<table>
<thead>
<tr>
<th>transaction purpose</th>
<th>command invocation</th>
<th>buffer transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>low &lt; 500KB/s</td>
<td>SPI, UART [over USB]</td>
<td></td>
</tr>
<tr>
<td>medium &lt; 50MB/s</td>
<td>AXII1ite, SPI</td>
<td></td>
</tr>
<tr>
<td>high &gt; 50MB/s</td>
<td>AXI, PCIe</td>
<td></td>
</tr>
</tbody>
</table>
Scenarios and interfaces

- FPGA module debugging
- Pre-processing / data reduction in FPGA
- FPGA result transfer to host
  - using DMA
- FPGA-SoC variant
  - with external peripherals
  - self-contained

Legend

<table>
<thead>
<tr>
<th>Transaction Purpose</th>
<th>Bandwidth</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command Invocation</td>
<td>low &lt; 500kB/s</td>
<td>SPI, UART [over USB]</td>
</tr>
<tr>
<td>Buffer Transfer</td>
<td>medium &lt; 50MB/s</td>
<td>AXIIlite, SPI</td>
</tr>
<tr>
<td></td>
<td>high &gt; 50MB/s</td>
<td>AXI, PCIe</td>
</tr>
</tbody>
</table>

From UART to PCIe and DMA: selecting connectivity for your FPGA-based subsystem
Scenarios and interfaces

- FPGA module debugging
- Pre-processing / data reduction in FPGA
- FPGA result transfer to host
  - using DMA
- FPGA-SoC variant
  - with external peripherals
  - self-contained

Legend

- transaction purpose >
  - command invocation
  - buffer transfer

- bandwidth >
  - low < 500kB/s: SPI, UART [over USB]
  - medium < 50MB/s: AXII lite, SPI
  - high > 50MB/s: AXI, PCIe

From UART to PCIe and DMA: selecting connectivity for your FPGA-based subsystem
Demo project: Tabletop 3D laser scanner

Hardware variants | Key software functionality

- Lattice CrossLinkNX (volatile FPGA) evaluation board
  a) UART-over-USB connected to laptop
  b) PCIe connected to NXP i.MX6 (quad ARM) system
- Microchip PolarFire SoC Icicle kit (quad RISC-V + antifuse FPGA)
- Xilinx Zynq-7020 (dual ARM + volatile FPGA) developer board
- FPGA-less designs available, more FPGA-SoC designs under development

Features

- Turntable with stepper motor
- Tripod with camera/laser holder
- IMX335 MIPI CSI-2 camera (5MP) max. data rate 150MB/s @30fps
- Two adjustable red line lasers
Demo project: Tabletop 3D laser scanner

Hardware variants | Key software functionality

- Preview image acquisition

2560 x 1920 → 160 x 120 (color)

2048 x 1536 → 512 x 384 (grayscale)
Demo project: Tabletop 3D laser scanner

Hardware variants | Key software functionality

- Preview image acquisition
- Checkerboard corner detection for orientation
Demo project: Tabletop 3D laser scanner

Hardware variants | Key software functionality

- Preview image acquisition
- Checkerboard corner detection for orientation
- On/off identification of line laser traces in frames
Demo project: Tabletop 3D laser scanner

Hardware variants | Key software functionality

- Preview image acquisition
- Checkerboard corner detection for orientation
- On/off identification of line laser traces in frames

Each algorithm can be performed either on the Linux host or on the FPGA, with varying load on the interconnect.
## Outline

**Bottom-up then full-circle**

<table>
<thead>
<tr>
<th>Layer</th>
<th>Linux host</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>application layer</td>
<td>C++ data processing</td>
<td>RTL algorithms, state machines, etc.</td>
</tr>
<tr>
<td>application layer handoff</td>
<td>target-specific</td>
<td>target module</td>
</tr>
<tr>
<td></td>
<td>C++ API library</td>
<td>RTL handshake</td>
</tr>
<tr>
<td>protocol layer</td>
<td>encode/decode</td>
<td>host interface</td>
</tr>
<tr>
<td></td>
<td>C++ code</td>
<td>RTL module</td>
</tr>
<tr>
<td>hardware abstraction layer</td>
<td>device driver</td>
<td>soft IP</td>
</tr>
<tr>
<td>physical layer</td>
<td></td>
<td>silicon IP and copper wires</td>
</tr>
<tr>
<td></td>
<td></td>
<td>standard-compliant</td>
</tr>
</tbody>
</table>
## Physical layer

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Net bandwidth</th>
<th>Support</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>UART</td>
<td>400 kB/s</td>
<td>i.MX6 (all 32/64bit SoC’s)</td>
<td>4 Mbps on-PCB routing</td>
</tr>
<tr>
<td>UART over USB</td>
<td>417 kB/s</td>
<td>FTDI</td>
<td>x64 host USB2.0 hi-speed, FT232R</td>
</tr>
<tr>
<td>SPI</td>
<td>4.8 MB/s</td>
<td>OMAP3xxx (all 32/64bit SoC’s)</td>
<td>40 MHz on-PCB routing</td>
</tr>
<tr>
<td>AXI4</td>
<td>50 MB/s</td>
<td>Zynq (all FPGA-SoC’s)</td>
<td>32 bit words, 100 MHz clock</td>
</tr>
<tr>
<td>PCIe</td>
<td>250 MB/s</td>
<td>CrosslinkNX (all mid-range FPGA’s)</td>
<td>one lane PCIe 1.x, 2.5 Gbps</td>
</tr>
<tr>
<td>AXI4</td>
<td>776 MB/s</td>
<td>PolarFire SoC (all FPGA-SoC’s)</td>
<td>64 bit x 256 bursts, 100 MHz clock</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Linux host</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>application layer</td>
<td>target module</td>
</tr>
<tr>
<td>application layer handoff</td>
<td>target-specific</td>
</tr>
<tr>
<td>protocol layer</td>
<td>encode/decode</td>
</tr>
<tr>
<td>hardware abstraction layer</td>
<td>device driver</td>
</tr>
<tr>
<td>physical layer</td>
<td>silicon IP and copper wires</td>
</tr>
<tr>
<td></td>
<td>standard-compliant</td>
</tr>
</tbody>
</table>

From UART to PCIe and DMA: selecting connectivity for your FPGA-based subsystem
Hardware abstraction layer
Device driver [host] | Soft IP [FPGA]

- Character device driver category
- transfer control by host
- from user space: `open()`, `ioctl()`, `read()`, `write()`, `close()`

UART:
  - typ. drivers: `bus/platform/drivers/imx-uart`
  - typ. device files: `/dev/ttyS*`, `/dev/ttymxc*`

UART over USB:
  - typ. drivers: `bus/usb-serial/drivers/ftdi_sio`, `bus/usb-serial/drivers/cp210x`
  - typ. device files: `/dev/ttyUSB*`, `/dev/ttyACM*`

SPI:
  - typ. drivers: `bus/spi/drivers/spidev`
  - typ. device files: `/dev/spidev*.*`
Hardware abstraction layer
Device driver [host] | Soft IP [FPGA]

• Character device driver category (cont’d)
  • a simple cross-platform AXIlite driver, to appear at /dev/<pick_your_favorite>

```c
static ssize_t device_read(
    struct file* file
    , char __user* buffer
    , size_t length
    , loff_t* offset
) {
    ssize_t retval = 0; // number of bytes read
    size_t len;
    u32 rdy;

    size_t len32 = length / 4; // align bytes to words
    if ((length & 0x3) > 0) len32++;
    void* buf = kmalloc(4 * len32, 0); // allocate memory

    if (buf) {
        for (i = 0; i <= 20; i++) { // wait for max. 210us
            if (i != 0) usleep({});
            iowrite32(0xAAAAAAAA, mmAxilite + OFS_AXILITE_SETREAD); // REQ to read control address
            rdy = ioread32(mmAxilite + OFS_AXILITE_SETREAD); // data from actual read address
            if (rdy == 0xAAAAAAAA) break; // ACK from read control address
        }
        if (rdy == 0xAAAAAAAA) {
            for (i = 0; i < len32; i++) // ACK to read control
                ((u32*) buf)[i] = cpu_to_be32(ioread32(mmAxilite + OFS_AXILITE_READ));
            iowrite32(0x55555555, mmAxilite + OFS_AXILITE_SETREAD); // ACK to control address
            retval = length - copy_to_user((void __user*) buffer, buf, length);
            kfree(buf);
        }
    }
    return retval;
}
```
Hardware abstraction layer

Device driver [host] | Soft IP [FPGA]

- Advanced category: PCIe and AXI4(-Burst)
- typically, only makes sense combined with DMA and interrupts
- the kernel’s Userspace I/O has corresponding features, already integrated with PCIe
- helpful article by Oleg Kutkov regarding PCIe
  https://olegkutkov.me/2021/01/07/writing-a-pci-device-driver-for-linux/
Hardware abstraction layer

Device driver [host] | Soft IP [FPGA]

• Generic RTL modules for UART and SPI

```vhdl
entity Uartrx_v1_1 is
  generic(
    fMclk: natural range 1 to 1000000;
    fSclk: natural range 100 to 5000000
  );
  port(
    reset: in std_logic;
    mclk: in std_logic;
    req: in std_logic;
    ack: out std_logic;
    dne: out std_logic;
    len: in std_logic_vector(16 downto 0);
    d: out std_logic_vector(7 downto 0);
    strbD: out std_logic;
    rxd: in std_logic;
    burst: in std_logic
  );
end Uartrx_v1_1;
```

```vhdl
entity Spislave_v1_0 is
  generic(
    cpol: std_logic := '0';
    cpha: std_logic := '0';
    nssByteNotXfer: std_logic := '0';
    nssPrecphaNotCpha: std_logic := '0'
  );
  port(
    reset: in std_logic;
    mclk: in std_logic;
    req: in std_logic;
    ack: out std_logic;
    dne: out std_logic;
    len: in std_logic_vector(16 downto 0);
    send: in std_logic_vector(7 downto 0);
    strbSend: out std_logic;
    recv: out std_logic_vector(7 downto 0);
    strbRecv: out std_logic;
    nss: in std_logic;
    sclk: in std_logic;
    mosi: in std_logic;
    miso: inout std_logic
  );
end Spislave_v1_0;
```
Hardware abstraction layer
Device driver [host] | Soft IP [FPGA]

- Generic RTL code for AXI-lite in two parts

```vhln
to
```
Hardware abstraction layer

Device driver [host] | Soft IP [FPGA]

- Vendor-specific IP for PCIe required
- Available from major vendors free of charge
- Survey mid-June 2022
- intel PSG Cyclone-V: Avalon interfaces
  “Cyclone V Avalon {Memory Mapped/Streaming} Interface for PCIe Solutions” (UG-01110_av{mm/st})
- Lattice CrossLinkNX
  “PCIe X1 IP Core” (FPGA-IPUG-02091-1.4)
- Microchip PolarFire SoC
  “PolarFire FPGA and PolarFire SoC FPGA PCI Express” (user guide)
- Xilinx Zynq: AXI4-Stream interface
  “7 Series FPGAs Integrated for PCI Express v3.3” (PG054)
From UART to PCIe and DMA: selecting connectivity for your FPGA-based subsystem

Application and protocol layers
Host to FPGA | FPGA to host

- Host: C++ API library forms byte code and initiates transfers guarded by CRC

<table>
<thead>
<tr>
<th>Terminal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection state: <strong>not connected</strong></td>
</tr>
<tr>
<td>Data in/out</td>
</tr>
<tr>
<td>Command execution</td>
</tr>
<tr>
<td>Command sequence</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>Append</th>
</tr>
</thead>
<tbody>
<tr>
<td>step.moveto()</td>
<td></td>
</tr>
<tr>
<td>step.moveto(angle=uint16, Tstep=uint8)</td>
<td></td>
</tr>
</tbody>
</table>

| Submit |

<table>
<thead>
<tr>
<th>Linux host</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>application layer</td>
<td>C++ data processing</td>
</tr>
<tr>
<td>application layer handoff</td>
<td>RTL algorithms, state machines, etc.</td>
</tr>
<tr>
<td>protocol layer</td>
<td>host interface</td>
</tr>
<tr>
<td>hardware abstraction layer</td>
<td>device driver</td>
</tr>
</tbody>
</table>

**Step Examples:**

- **tx:** 0x02 06 01 0005 7FPA
  - 1. hostToCmdInv
  - 2. controller: step
  - 3. command: moveto
  - 4. length: 5
  - 5. CRC

- **rx:** 0xAAAA ACK
  - 1. state: idle
  - 2. angle: 340 (uint16)
  - 3. CRC

- **tx:** 0x0106000005 7FPA
  - 1. controller: moveto
  - 2. length: 5

- **rx:** 0x000154 AD52
  - 1. state: moving
  - 2. angle: 226 (uint16)
  - 3. CRC

- **tx:** 0x0000 0005 7FPA
  - 1. controller: getinfo
  - 2. length: 5

- **rx:** 0x0000 0005 7FPA
  - 1. controller: getinfo
  - 2. length: 5

- **tx:** 0x0106000005 7FPA
  - 1. controller: moveto
  - 2. length: 5
FPGA: “host interface” module decodes the byte string and triggers a handshake with the ”step” target module.

```
entity Step is
  generic (
    fMclk: natural range 1 to 1000000 := 50000 in kHz
  );
  port (
    reset: in std_logic;
    mclk: in std_logic;
    tkclk: in std_logic;
    ...
    reqInvMoveto: in std_logic;
    ackInvMoveto: out std_logic;
    movetoAngle: in std_logic_vector(15 downto 0);
    movetoTstep: in std_logic_vector(7 downto 0);
    ...
    nslp: out std_logic;
    m0: inout std_logic;
    dir: out std_logic;
    step0: out std_logic
  );
end Step;
```
**Application and protocol layers**

**Host to FPGA | FPGA to host**

- FPGA: reduce 2560x1920 YUV images @30fps (150MB/s) to 160x120 RGB images (1.73MB/s), then provide to host in A/B buffer

- **Host:** poll the buffer status, initiate buffer transfer and display

```c
while (true) {
    if (shrdat.cancelPvw) break;

    shrdat.mPvw.lock("JobWzskAcqFpgapvw", "runPvw[2]");
    srv>xrcarty->camacq_getPvwinfo(tixVPvwbufstate, tkst);

    if (((tixVPvwbufstate == VecVWskdArtyCamacqPvwbufstate::ABUF) || (tixVPvwbufstate == VecVWskdArtyCamacqPvwbufstate::BBUF)) ||
        if (tixVPvwbufstate == VecVWskdArtyCamacqPvwbufstate::ABUF) srv>xrcarty->shrdat_hw.readPvwabufFromCamacq(sizeBuf, buf, datalen);
        else if (tixVPvwbufstate == VecVWskdArtyCamacqPvwbufstate::BBUF) srv>xrcarty->shrdat_hw.readPvwbbufFromCamacq(sizeBuf, buf, datalen);
    shrdat.mPvw.unlock("JobWzskAcqFpgapvw", "runPvw[2]");
} else {
    shrdat.mPvw.unlock("JobWzskAcqFpgapvw", "runPvw[3]");
    nanosleep(&deltat, NULL);
};
```

**From UART to PCIe and DMA: selecting connectivity for your FPGA-based subsystem**

Embedded Linux Conference North America 2022
Model-based design
Less repetitive coding and consistent results

• “Single source of truth” generation of code for host and FPGA sides
• One model file for modular structure, specifying “controllers” aka. RTL modules with commands / buffer transfers attributed
• Another model file to specify those commands and buffer transfers

From UART to PCIe and DMA: selecting connectivity for your FPGA-based subsystem

Embedded Linux Conference North America 2022
## Three FPGA API concepts

**Simple, callback-based, register-based**

<table>
<thead>
<tr>
<th></th>
<th>Simple (cf. slides before)</th>
<th>Callback-based</th>
<th>Register-based</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Blocking Access</strong></td>
<td>blocking access to FPGA resource for all command / buffer transfer requests</td>
<td>commands / buffer transfers can be launched (non-blocking), with notification on reply</td>
<td>no commands / buffer transfers exist; targets and their {set/get}table parameters are written / polled</td>
</tr>
<tr>
<td><strong>Processing</strong></td>
<td>one-at-a-time processing in host interface</td>
<td>command invocation / return buffer FIFO’s</td>
<td>simple mapping</td>
</tr>
<tr>
<td><strong>Pros</strong></td>
<td>• low FPGA footprint</td>
<td>• delayed and multiple command returns possible</td>
<td>• industry standard, good also for bare-metal</td>
</tr>
<tr>
<td><strong>Cons</strong></td>
<td>• host may have to poll status from multiple threads</td>
<td>• larger FPGA footprint</td>
<td>• lack of comfort, prone to error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• undefined behavior as update order not guaranteed</td>
</tr>
</tbody>
</table>
Conclusion

• Linux host to FPGA connectivity is not rocket science

• What helps
  • Few, established hardware interfaces are used throughout the industry
  • The corresponding FPGA IP is free, ideally Open Source
  • On the Linux side, there is convenient character device and UIO driver support

• Model-based generation of source code for both sides simplifies life further
  • Designs become interface-agnostic
  • Vendor lock-in can be avoided
Also, feel free to connect.

- [https://www.linkedin.com/in/wirthmua](https://www.linkedin.com/in/wirthmua)
- [https://github.com/mpsitech](https://github.com/mpsitech)