OP2P Interface IP Core Datasheet

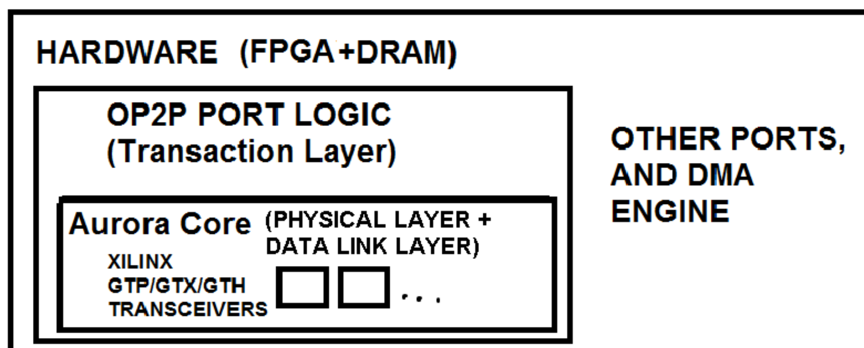
Istvan Nagy 2011
www.opencores.org
Buenos@opencores.org

Description:

Open Peer to Peer Interface, Wishbone to Aurora Bridge (OP2P).

This IP core is only one port. It implements a higher (transaction) layer of the communication stack, while the lower (physical) layer is implemented inside the Xilinx Aurora interface IP (using various types of the Xilinx multi-gigabit serial transceivers) generated in the Xilinx CoreGenerator program. The OP2P interface was developed to provide a low latency, low software-overhead board-to-board communication interface. It is basically a “Buffer-Copy” interface; it copies data from a DRAM memory buffer on one board to a memory buffer on another board, initiated by a command which specifies the address locations within both the source and the target buffers. The buffers should be memory mapped within the system address spaces of the boards independently (PCI/PCIe devices). It is based on PCI-express, with certain modifications: all ports are non-transparent and peer-to-peer supports packet forwarding in indirect mesh connections without the on-board system processor’s (usually X86 high performance processor like Intel Core-x, Xeon…) intervention. This is called distributed switching; no switch cards are needed in the system/network. The system or network can be backplane-based or cable-based, or a mixture of them. There are similarities with PCI-express in the way of handling the packets, but without the limitation of the master-slave relationships. There are also similarities with Ethernet, without the excessive software overhead and the limitations of the link-width and speed inflexibility. This interface cannot be used to replace a master-peripheral type PCI system, since it requires more intelligence in a peripheral card, and it is not compatible with the PCI Plug&Play BIOS/software, also all ports are non-transparent. The host (x86) processor does not read/write data directly from/to the OP2P port, but instead it provides a command (fill up 5 FIFOs with transaction parameters) to allow the OP2P port logic to take the data from/to the local DRAM buffer. A complete bridge/switch (FPGA chip logic) would consist of multiple OP2P ports with a local DRAM buffer, and the host (X86 processor) will have to read/write that DRAM buffer directly instead of reading/writing the appropriate ports directly.

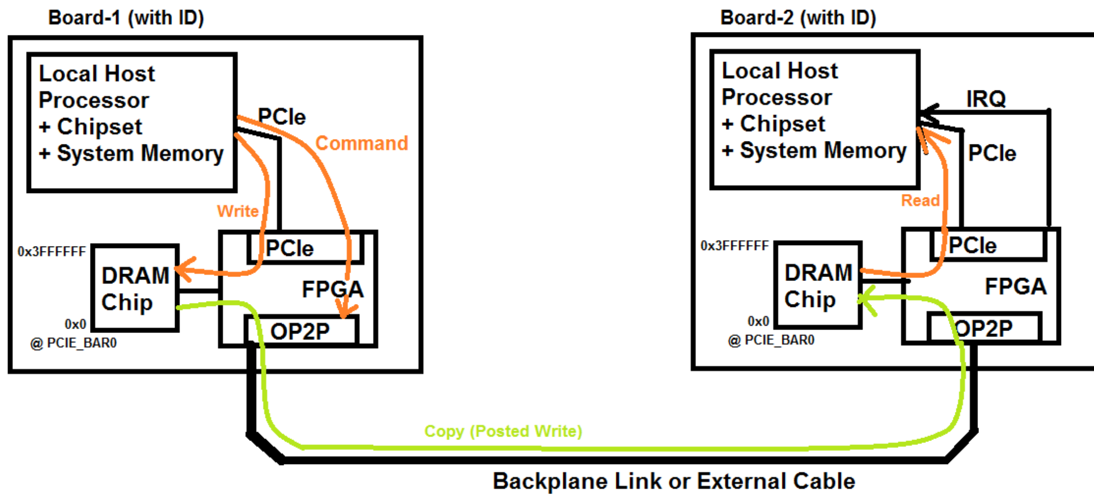
Interrupt signalling: The PCIe NT-port Doorbell interrupts also have a replacement in OP2P. Simply generate a write transaction into another device's buffer memory and then it's OP2P port will generate a local interrupt and will tell the local processor that a write data has been copied into the memory buffer (address location is stored in a readable FIFO). In the write we can have one double word payload data telling who (16-bit ID) sent the interrupting packet, and maybe telling some other parameters on the remaining 16 bits (e.g. interrupt number, or reason-code). This is all down to the software implementation. We can have a location in the local memory buffers which will only be used for interrupt signalling, so when someone writes into it then it is indeed an interrupt, and the payload data contains the sender and other codes.



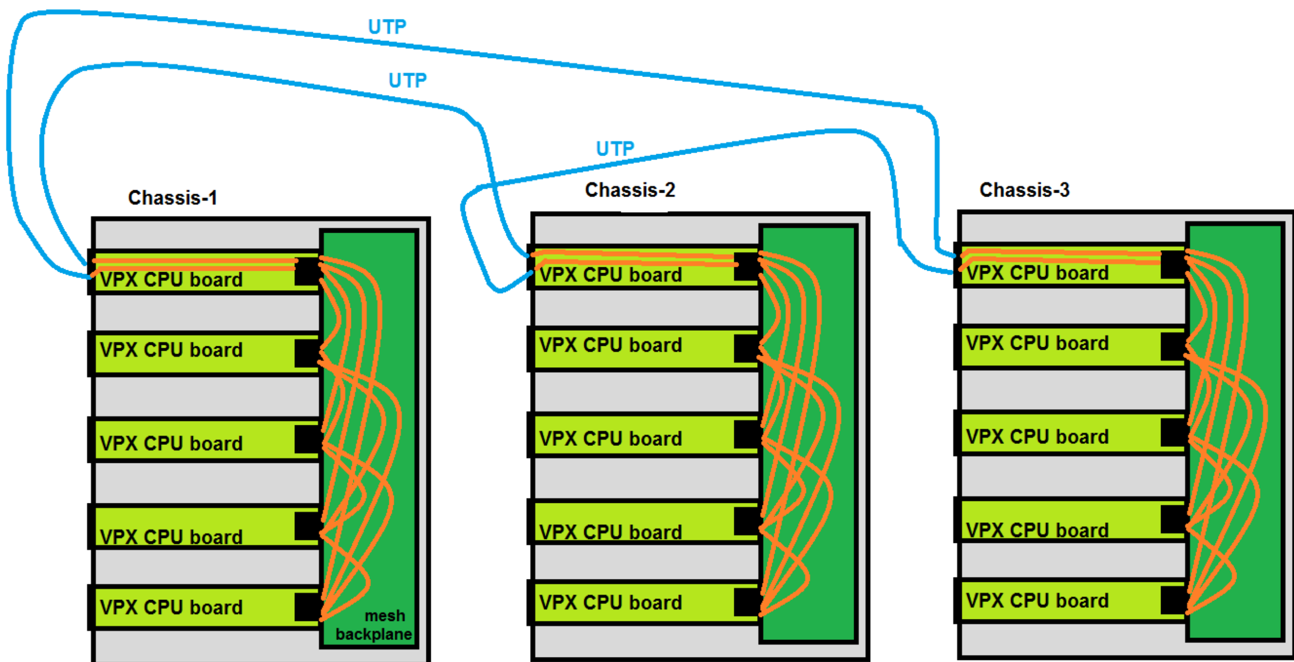
Layered Protocol

About the Xilinx Aurora Interface IP Core: This core is used as the physical layer logic for the OP2P interface. It implements packet frame generation/detection, flow control, error detection and link initialization. The clocking architecture used is pleisosynchronous, meaning that the reference clock signal does not have to be distributed to each device, instead they use clock compensation packets inserted into the data stream.

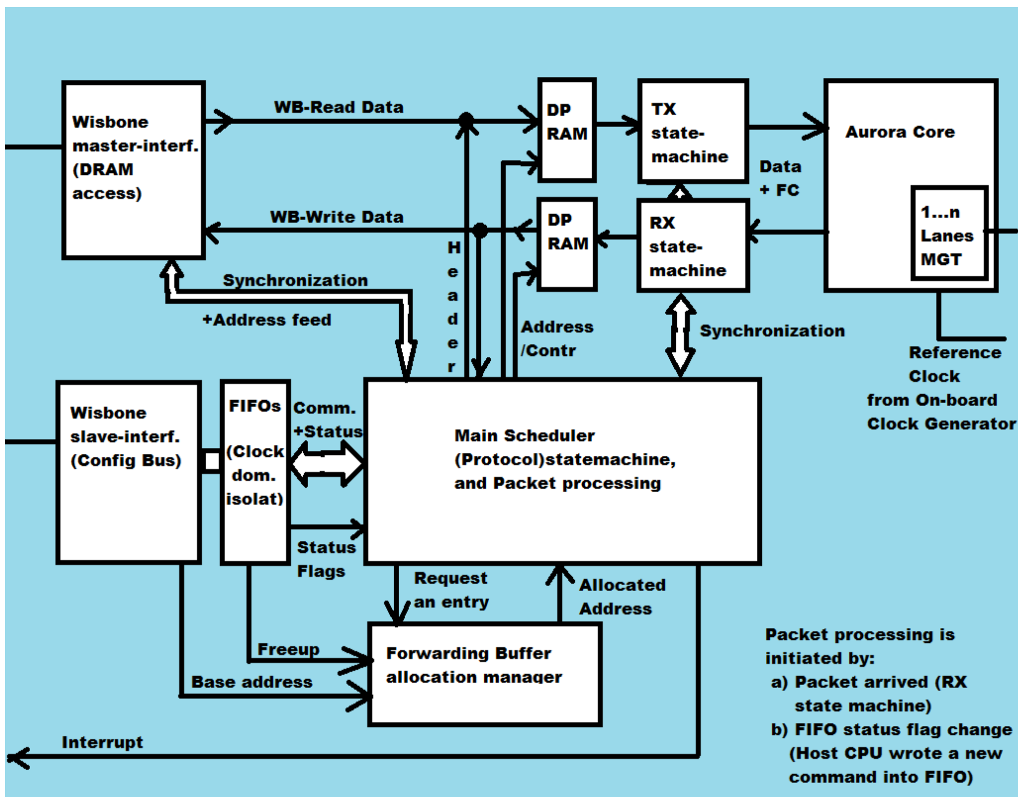
Electrical Characteristics: The actual silicon hardware electrical interface is implemented using the Xilinx FPGA's built-in multi-gigabit serial transceivers. The OP2P electrical characteristics therefore are equal to the Xilinx transceiver characteristics. These parameters are documented on the chosen FPGA device's datasheets and Characterization Report documents that are all available from the Xilinx website. Xilinx normally characterizes their transceivers against interface standards like SATA/PCIe, against electrical standards like various OIF (Optical Interface Forum) CEI (Common Electrical Interface) documents, and sometimes against mediums like CAT-5/6 UTP and other cables. The different Xilinx FPGAs have different types of transceivers built-in, for example the GTP, GTX, GTH, which all have their different maximum speed capability limits. In Q4 2011 they have FPGA built-in transceivers with maximum limits of 3.1Gbit/s to 28Gbit/sec.



Communication scheme: multi-step buffer-copy

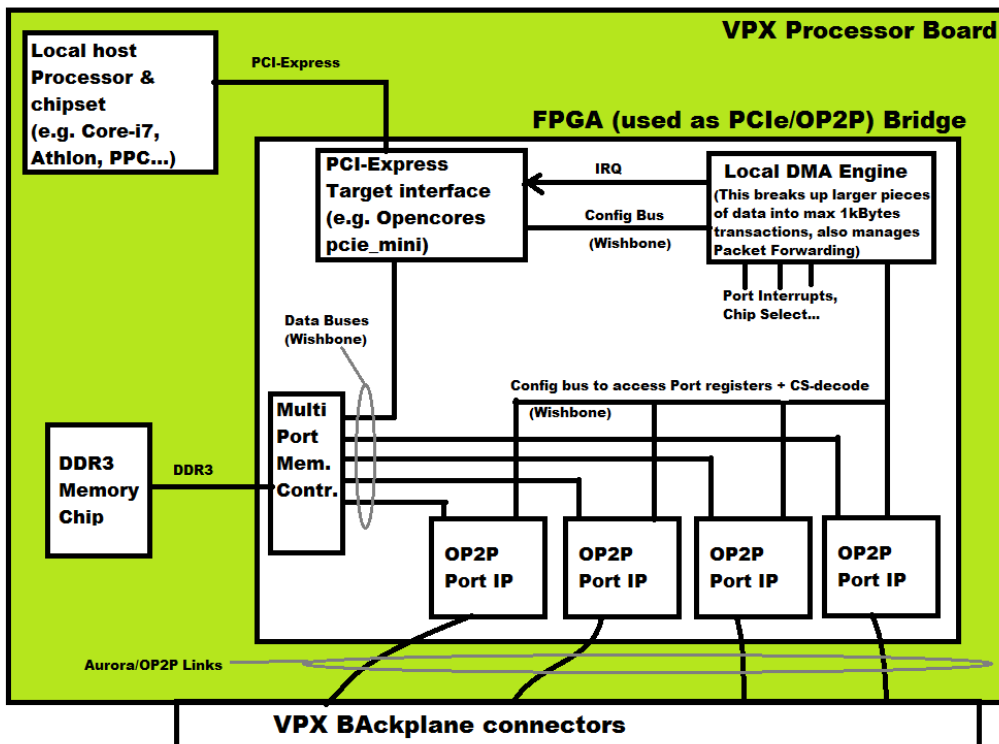


Possible OP2P connections in a server room



Port logic Block Diagram

In a plug-in board or blade server we could use an FPGA as the Backplane Bridge/switch (PCIe/OP2P Bridge). The OP2P interface is optimized for mesh backplane topologies, like the ones used on VPX form factors. The Backplane Bridge needs a separate port for each backplane link. There has to be higher level scheduler logic (DMA engine) or a soft processor to implement it inside the FPGA. To support bigger than purtnum^2 elements in a mesh, the bridge has to implement packet forwarding. This way not every node will be directly connected to every other. It may be obvious that the forwarding involves 2 ports and some interaction from a DMA or processor. The backplane bridge would normally consist of 4 OP2P ports, a host interface (PCIe), a local memory buffer interface (1 DRAM chip on-board, or BRAM on-chip) and a local-host (soft processor or state machine-type DMA logic inside the FPGA). This core can also be used in a 2-node system using a cable (Cat-6 UTP or Infiniband or SATA cables).



Possible VPX board implementation with OP2P

The maximum packet payload size is 1kBytes. This also means that the host processor has to write transfer commands for at least every Kbytes of data to be transferred. For example if we want to move 10Mbytes, then we have to issue commands 10'000 times. An on-chip (FPGA) DMA engine could handle this: the host processor would give one command to the DMA engine, while the DMA engine would give 10'000 commands to the OP2P port.

To act as a PCI-like bridge, meaning that a PCIe-to-buffer access automatically generates an OP2P access and vice versa, the a few features have to be implemented. The DRAM buffer would still be used, but one write would be followed by a read on the other side, and one read would be preceded by a write. All transactions would start in the system memory (not the FPGA buffer DRAM, but the main DIMM memory of the processor) on one board and end in the system memory on another board. When the DMA engine sees that something arrived on one side (PCIe or an OP2P port), it would initiate a transaction on the other side. With this bridge approach, the local DRAM would act as a series of windows (set up as address-ID translation) to other devices in the system/network. This however would have limitations on the network size, since we would need dedicated address range to each remote board, we cannot reuse an address to access to other devices too. It would also require larger amount of memory buffer. If we break up with the PCI-bridge-like approach and accept to break up the system memory to system memory transfer into 3 DMA transactions, then we don't have the network size – memory size limitations, and the DMA engine doesn't have to do automatic address-to-ID translations. This would work with the original above block diagram with the need for only smaller modifications (this would be the PCIe bus mastering). For reads the source data has to be already taken care of before the OP2P read arrives, for writes the arrived data has to be moved to system memory after it has arrived. Basically the software would move data (by 2 DMA transfers) from system memory address to OP2P ID with remote buffer address. Moving data from/to the system memory might not be needed at all, and then we can use the original block diagram for the PCIe-to-OP2P bridge/switch FPGA above without any modifications.
The features to implement for the full PCI-bridge-like operation:

- The OP2P port has to implement a handshaking with the DMA engine for reads. When a read request arrives it would assert a rdreq_arrived signal to the DMA engine and wait with starting the Wishbone read until the DMA engine asserts the read_data_valid signal. The OP2P interface will also have to provide an address and size information about the read request. During the waiting period the DMA engine gathers the data from the local processor's system memory.
- The PCIe interface (for example the also open source pcie_mini) has to implement the same signalling to the DMA engine, as the OP2P port has.
- The PCIe interface also has to implement bus mastering, meaning that it can initiate transactions from/to the system memory. The proposed PCIe interface, the pcie_mini currently only works as a PCI target only.
- The DMA engine has to implement System memory address to OP2P-ID translation and vice versa.

Electrical Interface:

The data line rate is set to 1.5Gbps / diffpair, with 2 lanes in each direction, so this gives a total of 6Gbps bandwidth. The reference clock is expected to be 75MHz LVDS differential, but if we want to regenerate the core for a different line rate then we need to change the ref lock frequency. The reference clock must come from an on-board external clock generator; the Spartan6's internal clock networks don't meet the GTP's jitter specs. The external clock chip frequency can be controlled through the PR0,PR1, OD0, OD1, OD2 pins.

Limitations:

This code is a proof of concept only. The data bandwidth is limited, since the on-chip data buses/processing is 32-bit. The Spartan-6 (-2 speed grade) device is able to run the interface with an around 100MHz on-chip parallel bus, but since the valid PLL settings don't allow for parallel bus speed in this range, the on-chip logic runs on 75MHz with serial line rate of 1.5Gbps. On a newer 7-series FPGA (Kintex-7, Virtex-7), it would run faster. To utilize the 10.3Gbps speed on a 4x4 lane 6U-VPX backplane application, we would need around 1.03GT/sec on-chip parallel buses with long burst support. At the moment the core has 32-bit buses (FIFOs, Wishbone-bus to DRAM) and no burst support on the Wishbone side. If we change the design to have 128bit on-chip parallel buses, then it would need a 250MHz parallel-bus-clock speed, which might work on a Virtex-7 or Kintex-7 device. This would mean a 4GBytes/s/port/direction data bandwidth (32GB/s/card total) on each backplane port (6U VPX at 10.3Gbps). On ISE-12.1 for Series-6 FPGAs, 128bit bus is only available for min x4. A 3U VPX system at 10.3Gbps line-rate would have 1GBytes/sec/dir data bandwidth on each port, which could still be achieved with a 32-bit on-chip bus at 250MHz. At 10.3Gbps (6U VPX), x4 link width (instead of x1) would not provide additional bandwidth when using 32-bit parallel bus at 250MHz. 250MHz is around at the device limit with this core for Xilinx series-6/7. A 64-bit 66MHz Compact-PCI interface only has a 0.528GBytes/sec aggregate bandwidth used by all cards and directions together. The serial and the Wishbone transactions overlap to save time (latency), but transferring the same amount of data on the Wishbone bus takes 3-10-times more time than

it is on the serial side. Currently at every 3-10 clock cycles there is a new Wishbone transaction when processing a single OP2P packet. A long burst support would mean one WB transfer at every clock cycle.

The burst support has to divide the incoming/outgoing packet payload data into smaller Wishbone-bus bursts up to a maximum specified size. For example a 1 Kbytes (256 Dword) packet will be loaded on the Wishbone bus in eight consecutive 32-Dword bursts (128 Bytes). This would increase the on-chip parallel bus performance to 60-80% of the theoretical bandwidth (based width and clock frequency) of the parallel bus. Without burst support, the real bandwidth is about 10-50% of the theoretical. Implementing bursts in the existing VHDL code should be simpler than the bus-width increase, although it would still require lots of changes and debugging with the ChipScopePro logic analyser.

Protocol:

The lower layers are based on the Xilinx Aurora protocol, and generated by the Xilinx CoreGenerator tool and instantiated in this file. The higher layers are implemented in this file. All transactions are peer to peer, non-transparent, and implementing buffer-copy from device/address to device/address. The specified address is within the memory buffer chip, and the memory buffer chips are memory-mapped into the local host processor's PCI-memory space. Since the buffers are independently mapped into the two end's memory spaces, the accesses are non-transparent at both ends. Transactions can be initiated by local host (it writes into local memory, then sets a register bit accessed through the op2p_config_wb bus) or by arrived read request packet from remote device. What happens during transmit: The IP reads the data from the local buffer, and sends it to the specified destination (target), then generates an interrupt. Receive: when receiving a packet, the aurora IP writes it into the memory (address is specified in the packet), then generates an interrupt to the local host. So, all transfers happen directly between the memory and the aurora link, the host cannot write the data into the aurora IP.

Trigger for (DRAM) WB transfer: From Local-Host (op2p_config_wb bus, command registers filled) or request or response packet from remote device, or arriving forwarding packet.

Main transaction types:

- Posted Write
- Read request. The destination will send back a read response packet with data when ready, and that arrives like a write from the original destination board/device.
- Resend-request. In case of an error detected in a received packet, the last sent/received packet will be re-sent with the same header, by the link-partner.

Error handling:

The core can detect if there was an 8b10b-encoding based "non-existing-code" and disparity error, then it requests the last packet to be re-sent by the link partner. The number of total errors is counted and can be read by software from a register.

Target Devices:

The Xilinx Spartan-6 LXT XC6SLX45T (for a 1.5GGbit/s 1x2 CAT-6-UTP), device was used for initial debugging. An optimal device for backplane applications could be the Xilinx Kintex XC7K355T-2FFG901 (10.3Gbit/s 4x4, for 6U VPX), or a Kintex XC7K160T-2FFG676 (10.3Gbit/s 4x1, for 3U VPX), but any Xilinx Kintex or Virtex series FPGA would be suitable.

Dependencies:

This design includes the aurora interface core which was generated by the Xilinx CoreGenerator. All the VHD files were copied here, including the ones from the "Reference Design" folder. This file is the top level source of the module, and is not generated by CoreGen. Search for all VHD files in all subfolders, then copy all.

Two files had to be modified:

- aurora_8b10b_v5_1_reset_logic.vhd
- aurora_8b10b_v5_1_example_des_modified.vhd

The last one has its filename and module name also modified, not only the internal logic.

REQUIRED MODIFICATIONS ON THE GENERATED AURORA FILES:

This design includes the aurora interface core which was generated by the Xilinx CoreGenerator. Most of the VHD files were copied from the CoreGenerator project into the OP2P project, including the ones from the "Reference Design" folder, then some were modified.

If we use different Reference clock frequency or Line-rate or parallel bus width or a different device-type than the reference design has used, then we have to regenerate the Aurora core, and re-modify the files.

Importing a new Aurora core:

1.) Use search in the CoreGenerator project folder (\aurora_8b10b_v5_1) for *.VHD files.

2.) Select all of them.

3.) De-select the following (ctrl+click):

aurora_8b10b_v5_1_example_design.vhd
demo_tb.vhd
aurora_8b10b_v5_1_frame_check.vhd
aurora_8b10b_v5_1_frame_gen.vhd

4.) Copy the selected files into the FPGA project's source folder (\User_Sources) to overwrite all files there.

5.) Modify the two files based on the instructions below.

FILE-1 (aurora_8b10b_v5_1_reset_logic.vhd):

Change the following:

Comment this out:

```
  --Assign an IBUFG to INIT_CLK
init_clk_ibufg_i : IBUFG
port map
   (
        I    =>  INIT_CLK,
        O    =>  init_clk_i
  );
```

Copy this in instead:

```
init_clk_i <= INIT_CLK;
```

FILE-2 (aurora_8b10b_v5_1_example_des_modified.vhd):

Do not overwrite my old file with a newly generated one. modify my old file with some data from the new file (\example_design\aurora_8b10b_v5_1_example_design.vhd):

Replace this:

```
attribute core_generation_info           : string;
attribute core_generation_info of MAPPED : architecture is "aurora_8b10b_v5_1,aurora_8b10b_ ...
Parameter Declarations
        constant DLY : time := 1 ns;
External Register Declarations
```

To the same part from the new file, but replace the architecture name from: “mapped” to: “behavioural”

If necessary, replace the module names from “aurora_8b10b_v5_1” to the new name coming from the core generator. it might use a new version of the core.

Change the gtp and refclock ports if needed, in the top port list and also internally, to the used port widths or names, based on the board design.

For example from this:

```
GTPD1_P   : in  std_logic;
GTPD1_N   : in  std_logic;
RXP                 : in std_logic_vector(0 to 1);
RXN                 : in std_logic_vector(0 to 1);
TXP                 : out std_logic_vector(0 to 1);
TXN                 : out std_logic_vector(0 to 1)
```

Change to this:

```
GTPD0_P   : in  std_logic;
GTPD0_N   : in  std_logic;
RXP                 : in std_logic_vector(0 to 3);
RXN                 : in std_logic_vector(0 to 3);
TXP                 : out std_logic_vector(0 to 3);
TXN                 : out std_logic_vector(0 to 3)
```

Packet format:

Header + payload data. Max 1kBytes in one packet is allowed. The header is always 4 double words (128bit total).

Header:

1st DW: source ID (16bit 31:16), destination ID (16bit 15:0)
2nd DW: destination address (32bit)
3rd DW: source address (32bit)
4th DW: byte count(16bit: 1-64k 31:16), packet-type (4bit 15:12), status (4bit 11:8),
first byte enable (4bit 7:4), RFU (4bit 3:0)
Packet type: 0000=wr_req, 0001=rd_request, 0010=rd_completion, 1010=retransmit_req, others: RFU
Status: 0000=succesful_transaction, 0001=no_further_hop, 0010=unknown_error

Addressing: device identification is based on ID. Every packet copies a specified amount of data from source_address in source device to destination_address in destination device (wr) or requests a read in the opposite way. The Aurora bridge normally has its own 4G or less address space, which is a DRAM chip connected to the bridge (FPGA), typically 64MBytes. Every board has its own 0...MAX address space, which is independent from the other board's address spaces. The local host is a processor, which can be the board's main x86/PPC/MIPS processor or a dedicated small processor inside or attached-to the aurora bridge (FPGA). The completion swaps the source/destination ID/address fields! ID: 10-bit chassis address and 6-bit slot address. The slot address can be figured out from the backplane's geographical addressing pins. The chassis address have to be specified by the user in a software or stored on the backplane in an EEPROM. The ID is programmed into the aurora interface registers after power up by the local host. There is no plug and play device discovery by hardware, although software can initiate discovery by pinging all possible device numbers in the chassis or in the server room. ID=0 means that the destination of the transaction is the device immediately found at the other end of the physical link, which also means that chassis addresses start from 1 till 1023, slot addresses from 0 till 63. ID0 can be used for discovery in backplanes, or for communication in pint-to-point logical/physical connections in dedicated cable-links (system with 2 devices only). Discovery: in the backplane, the port/link connections are known based on the geographical addressing and the fixed topology. Only slot-0 has access to other chassis. At power up, every slot pings the other slots to see if they are there, by a dummy 1-byte read. This has to be initiated by software on the host processor or by a DMA engine.

Multi-hop transactions FORWARDING) in multi-mesh topologies: if a device receives a packet which does not match its ID, then it has to store the packet in local DRAM, and the local host has to retransmit in the appropriate port (another aurora block), based on the discovery or map. A packet that arrived without an ID-match is considered as a forwarding packet. Packets to other chassis have to go through slot-0, to simplify the packet routing protocol, but this is a software consideration. When a port forwards a packet, the host writes the original source ID into the source ID FIFO, not its own local ID. Sending forwarding (posted-write or read-request) request packets is done like sending normal request through the same command FIFOs, with source-ID/=local_id written into the FIFO. Receiving a forwarding (not matching ID) packet: store the whole packet in DRAM at an automatically allocated location. The host can see in the siso_status_reg if there is anything, and read the latest packet's address from the op2p_forwreq_pointer_reg FIFO. Later when the local host processor initiates a re-transmit, it should tell the port logic to free-up the memory buffer portion used by the packet. This is done by using the op2p_forw_freeupaddr_reg command FIFO. The buffer in use starts at a pre-programmed point set by op2p_forwarding_bufferbase_reg. The buffer can have max 512 entries, each 2kByte (to accommodate a 1kBytes data + 4*32bit header), 1MByte total. If a device is the actual destination for a forwarding packet, then it will see it as a normal packet, since the destination ID will match its local_id, and it will act as it would to a normal request packet.

Software requirements:

Each board's local host processor has to have a complete map of the system. This can come from a discovery protocol (not detailed here) or from a user manual entry (non- plug and play, requires an administrator to maintain the system, instead of "end users" like in case of commercial products). This map tells which port we find the node that we want to talk to, and which port we send the forwarding packets to. The local host interaction to the system can be done two ways: a) simple system where one processor handles all transactions at a register-level, b) a local DMA controller or soft processor inside the FPGA handles all packets at the register level but the local host processor only handles the user data and port mapping. In a possible discovery protocol we do iterations, where node requests information from their neighbours in every iteration. The data requested/provided contains data gathered in the previous iterations. In the first iteration we detect only the direct neighbours (link partners) and their slot/chassis addresses. The number of iterations needed equal to the number of hops needed plus one. In a room of four 8-slot chassis we need 4 iterations. Handling this should be done by a software layer.

Registers (op2p_config_wb bus):

WRITE REQ (local host writes): source address (local), destination address, source ID, destination ID, byte-count.
WRITE_COMPLETION (this IP writes it): source address (remote), destination address (local), source ID, destination ID (check for match or forwarding), byte-count.
READ REQ (local host writes): source address (local), destination address, source ID (set up once), destination ID, byte-count.
READ completion status: completed_localaddress
Forwarding REQUEST received: buffer start address pointer and buffer size (local host writes it, not a FIFO), Packet pointer (FIFO, the aurora IP writes it). The complete packet with header is stored in memory. the packet size is in the header
Forwarding buffer freeup: free up buffer area already forwarded. address and size.
Control registers: FIFO status for every FIFO.
All of these registers are FIFOs, one transaction involves reading or writing 5 FIFOs (a COMMAND). Source ID is local ID for most of the transactions, except for the forwarded packets. Initiate outgoing transactions: Write all required registers in a set (wr, rdreq). Where more than one FIFO has to be written, the last written one will initiate the OP2P transaction. (all regs in a set must be written once). If a read request arrives, the core will send the completion back with data, and will mark the source ID from the localid_reg, that has to be written after system initialization once.

Link health: The host software should read the op2p_link_status_reg before initiating any op2p transactions, because if the link is not alive and trying to send packets then (due to broken connection or unpowered link partner) the system might hang. The logic will not initiate any transactions if the link is not alive, this can cause command buffer overflow.

Register Addresses (BYTE ADDRESSES):

00h - op2p_fifostatus_reg
Tells the status flags of the command FIFOs
04h - op2p_wr_sourceaddress_reg (write command)
08h - op2p_wr_destinationaddress_reg (write command)
0Ch - op2p_wr_sourceid_reg (write command)
10h - op2p_wr_destinationid_reg (write command)
14h - op2p_wr_bytecount_reg (write command)
18h - op2p_rdreq_localaddress_reg (read request command)
1Ch - op2p_rdreq_destinationaddress_reg (read request command)
20h - op2p_rdreq_sourceid_reg (read request command)
24h - op2p_rdreq_destinationid_reg (read request command)
28h - op2p_rdreq_bytecount_reg (read request command)
2Ch - op2p_rdcompl_localaddress_reg
If a read is completed, this FIFO will show the address where the data has been stored.
This can be used to poll the status of the read operation.
30h - op2p_forwarding_bufferbase_reg (forwarding buffer setup)
At startup write here the base address of the 1MBytes forwarding buffer in the DRAM buffer
38h - op2p_forw_freeupaddr_reg (forw buf runtime management)
After retransmitting a forwarding packet, the local host proc/DMA should write the
value from the op2p_forwreq_pointer_reg back into this FIFO to free-up the buffer entry.
40h - op2p_forwreq_pointer_reg (forw buf runtime management)
If a packed arrived without ID-match, then it got stored in the local DRAM buffer
for retransmitting. The starting addresses are stored in this FIFO for the CPU to read.
44h - op2p_arrivedwrite_address_reg
If a write has arrived, this FIFO will show the address where the data has been stored.
This can be used to to see if anyone wrote data to us.
48h - op2p_localid_reg
The local ID of this device, written by local host. Receiving completion
or addressed (dest ID/=0) packet without setting this is not possible. Set after start-up.
4Ch - op2p_link_status_reg
Tells if the link is alive, and also which lanes
50h - op2p_port_reset_reg
We can initiate a soft reset to this OP2P port by register write
54h - link_error_count_reg

This counts errors found in incoming packets. Counts forever from reset.
8b10b encoding-based "non-existing-code" and disparity errors get detected.

Register bits:

op2p_link_status_reg:
bit-0=CHANNEL_UP,
bit-0=CHANNEL_UP,
bit-n:1=LANE_UP(0:n),
other bits are zero
bit-29: fc_haltlinkpartner
bit-30: fc_halted_bylinkpartner
bit-31: HARD_ERROR (needs port-reset)
op2p_fifostatus_reg
(0) <= op2p_forwreq_pointer_regempty;   empty
(1) <= op2p_rdcompl_localaddress_regempty;
(2) <= op2p_arrivedwrite_address_regempty;
(3) <= op2p_wr_sourceaddress_regfull;   full
(4) <= op2p_wr_destinationaddress_regfull;
(5) <= op2p_wr_sourceid_regfull;
(6) <= op2p_wr_destinationid_regfull;
(7) <= op2p_wr_bytecount_regfull;
(8) <= op2p_rdreq_localaddress_regfull;
(9) <= op2p_rdreq_destinationaddress_regfull;
(10) <= op2p_rdreq_sourceid_regfull;
(11) <= op2p_rdreq_destinationid_regfull;
(12) <= op2p_rdreq_bytecount_regfull;
(13) <= op2p_forw_freeupaddr_regfull;
(14) <= RFU
(31 downto 15) <= (OTHERS => '0');
op2p_port_reset_reg
bit-0: set 1 to hold reset, set 0 to release from reset. After startup the port is not in reset.
Address registers:
all 32 bits are used. 4GB address space in the local DRAM buffer. (not x86 host memory space address)
ID registers:
bit [15:0] are used for the 16-bit IDs.
Byte count registers
The number of bytes to be transferred. This must be 32bit aligned, 2 LSBs will be ignored.

Interrupts to local host:

- read completion (fifo was written)
- forwarding request
- write arrived

if any of the readable FIFOs is not empty, the interrupt stays asserted.

Flow control:

When the core is still busy with processing the previous packets, it signals to the link partner through the Flow control interface (which embeds FC packets into the gigabit external stream) to pause sending packets, then when it becomes not-busy again it will signal to the link partner to re-enable arriving packets.

Clocking architecture:

Every chip has its own reference clock, to refclk is wired between boards, only the data is connected. Clock Compensation: The Aurora 8B/10B protocol specifies a clock compensation mechanism that allows up to +/- 100 ppm difference between reference

clocks on each side of the link. This is a part of the CoreGenerator-Aurora IP and it automatically inserts CC messages into the communication channel, using up around 1-2% of the data bandwidth. Reference clock: From external pin, on-board (on the PCB) 75MHz low jitter clock generator/PLL/oscillator. For the prototype board, a Texas Instruments CDCM61001 was used. The reference clock source must be at least +/-50ppm accurate. The DRAM/buffer interface is in the same clock domain as the rest of the OP2P/Aurora logic, while the host interface is separate and isolated by command FIFOs.

Device Type Migration:

This core should work on any Xilinx Series-5/6/7 FPGAs, but at now it runs on XC6SLX45T. For a new device (not an XC6SLX45T) we have to regenerate the CoreGenerator cores, replace all BUFIO2/MGT/BUFG/BRAM (and other) to the chosen device's appropriate resources, in both the VHDL and the UCF sources. Also in the UCF the BUFIO2 and MGT placements will have to be re-specified with the appropriate resources/locations. The CoreGenerator will have to be set up to generate cores with the same parameters and ports as they are used here (to be useable as a drop-in replacement). Some resources are instantiated as part of the CoreGenerator cores, so they will be chosen by CoreGenerator appropriately, we just need to adjust their LOC placement constraints in the UCF file.

CoreGenerator parameters:

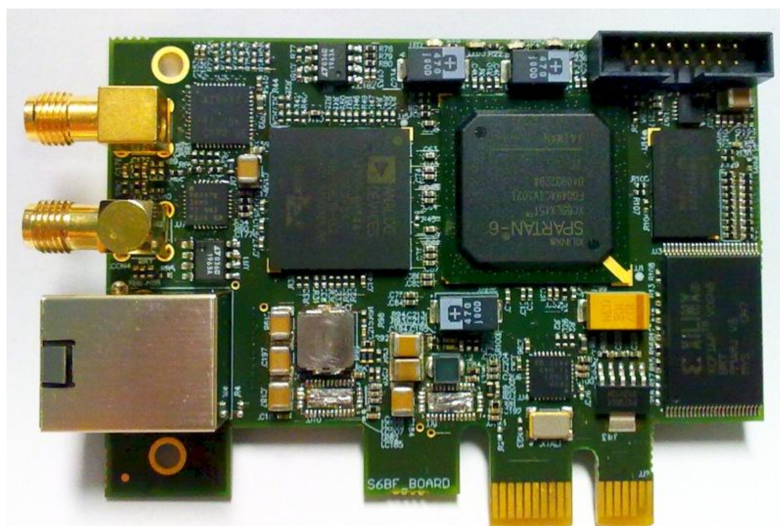
Aurora8b10b: Name=aurora_8b10b_v5_1, Lanes=2, LaneWidth=2 (bytes on parallel bus from one lane),
LineRate=1.5, GT_REFCLK=75MHz. These first few parameters have to be set accordingly,
to achieve a 32bit databus (lanes x LaneW =4), valid PLL settings (from datasheet:
a valid SerialRate-parallelclock-REFCLK combination), STA result on the parallel bus
clock (check max possible CLK speed in STA report).
DataFlowMode=Duplex, Interface=Framing, FlowControl=CompletionNFC, GTP-LOC:X1Y0-GT0=1/GT1=2,
Clock source=GTPD1. The LOC has to work with the board schematics and other cores (PCIe).
Parallel interface is TRN-interface.

Blockram: Name=blk_mem_gen_v4_1, Type=SimpleDpRAM, WriteEn=off, Algor=MinArea,
WriteWidth=32, WriteDepth=512, Ena=AlwaysEnabled, ReadWidth=32, RegisterPorttB=off
LoadInitFile=off, Fill=off, UseRSTB=off.

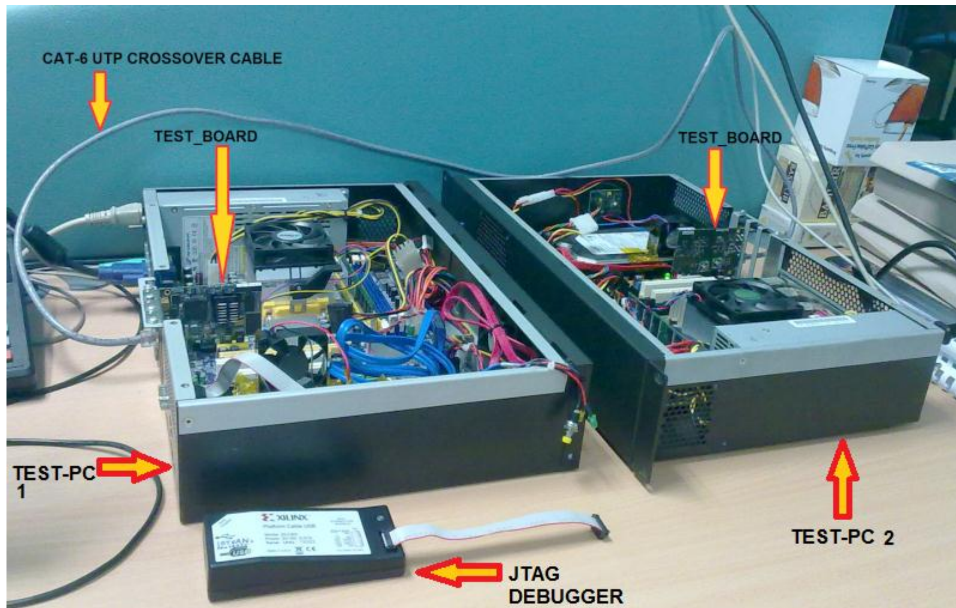
FIFOs: Name=fifo_generator_v6_1, R/W-ClockDomains=IndependentClocks/DistrRAM, ReadMode=StandardFIFO,
WriteWidth=32, WriteDepth=16, AlmostFull/Empty=off, WriteACK=off, WrOverflow=on, RdValid=off,
RdEnderflow=on, ResetPin=on, EnableResSync=on, FullFlagResVal=0, UseDoutReset=on, DoutResVal=0,
ProgrammableFlags(full/emp)=No, DataCount=AllOff,

Test board, proof of concept

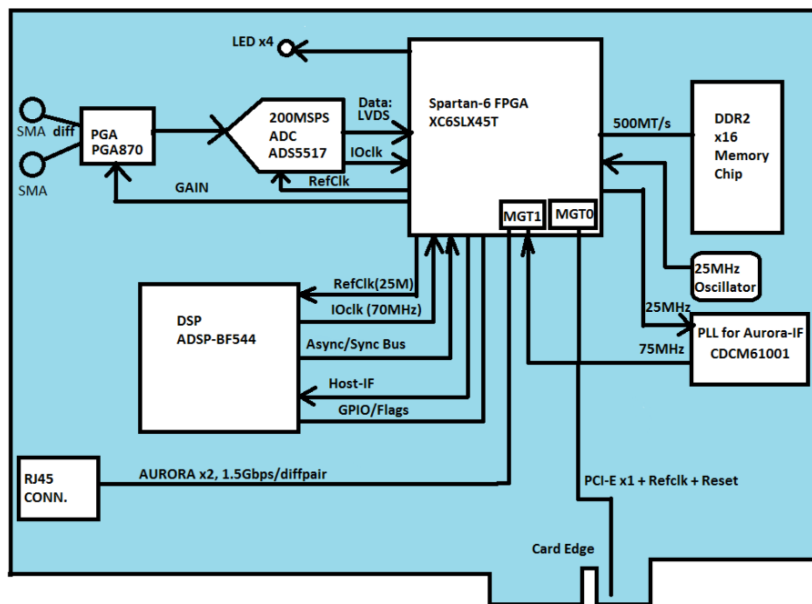
The OP2P port logic IP core was tested on a small PCI-express card (S6BF_Board) with a Xilinx Spartan-6 LX45T FPGA on it. The test card had a single OP2P port routed to an RJ45 connector with two lanes (two sets of RX/TX signals). The connection was not backplane based, but a Cat-6 UTP crosslink cable was used. The test setup consisted of two PCs with an S6BF_Board plugged into each. This card had other purposes as well (data acquisition and processing), not only the testing of the OP2P interface.



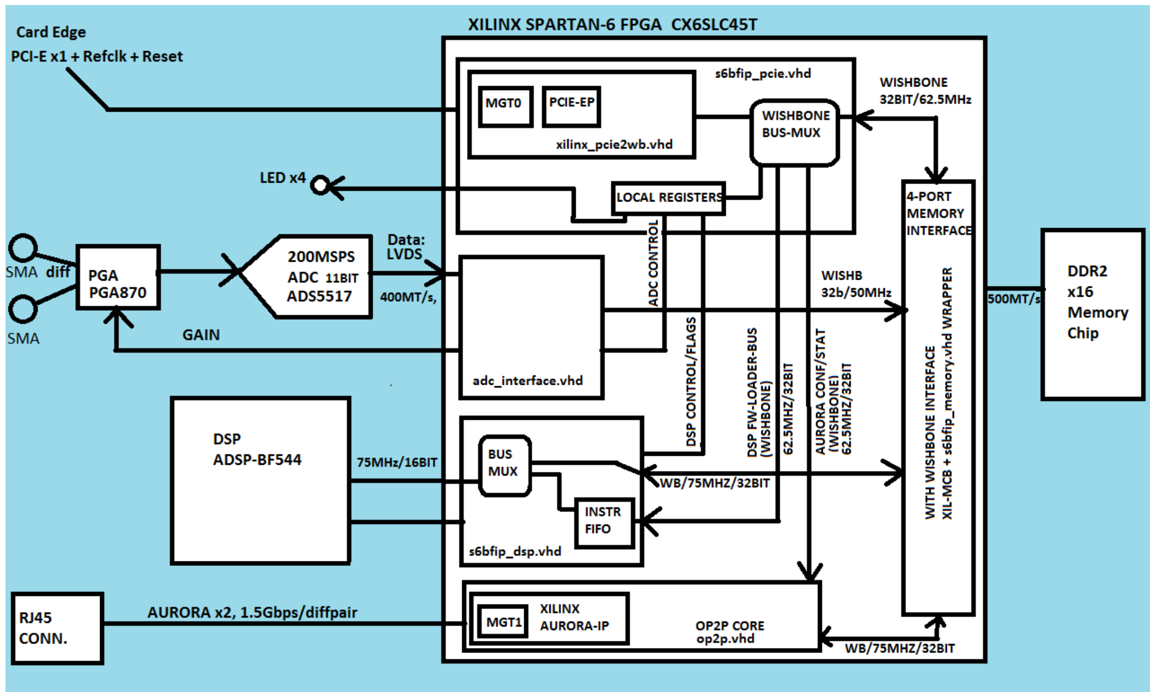
Prototype test board (S6BF_BOARD without the FPGA-heat sink)



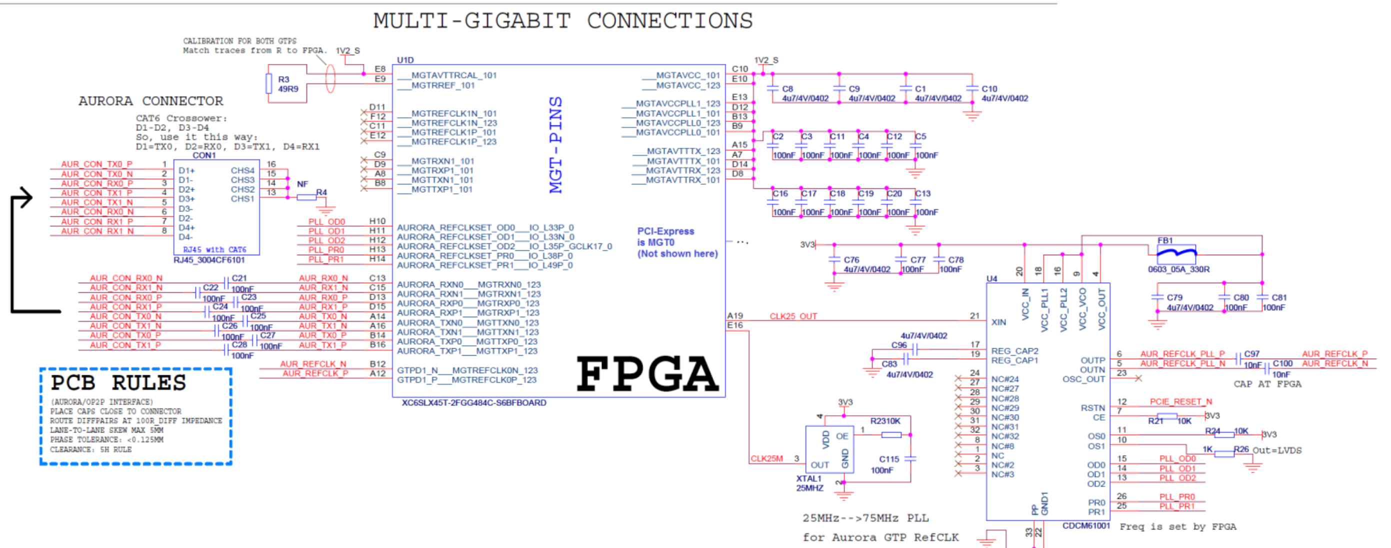
Test setup



Prototype test board block diagram



Prototype test board FPGA logic block diagram



Prototype test board schematics FPGA connections