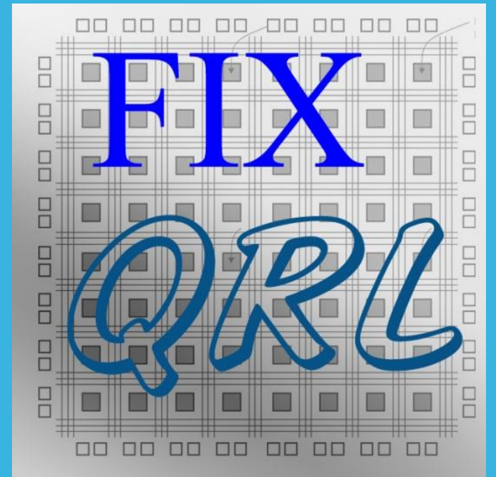
1G eth UDP IP stack

SIMPLIFIED IMPLEMENTATION
FROM THE FIX.QRL STABLES

(CONTRIBUTOR – PETER FALL)

V1.2



FEATURES

Implements UDP, IPv4, ARP protocols

Zero latency between UDP and MAC layer

- (combinatorial transfer during user data phase)
- See simulation diagram below

Allows full control of UDP src & dst ports on TX.

Provides access to UDP src & dst ports on RX (user filtering)

Couples directly to Xilinx Tri-Mode eth Mac via AXI interface

Separate building blocks to create custom stacks

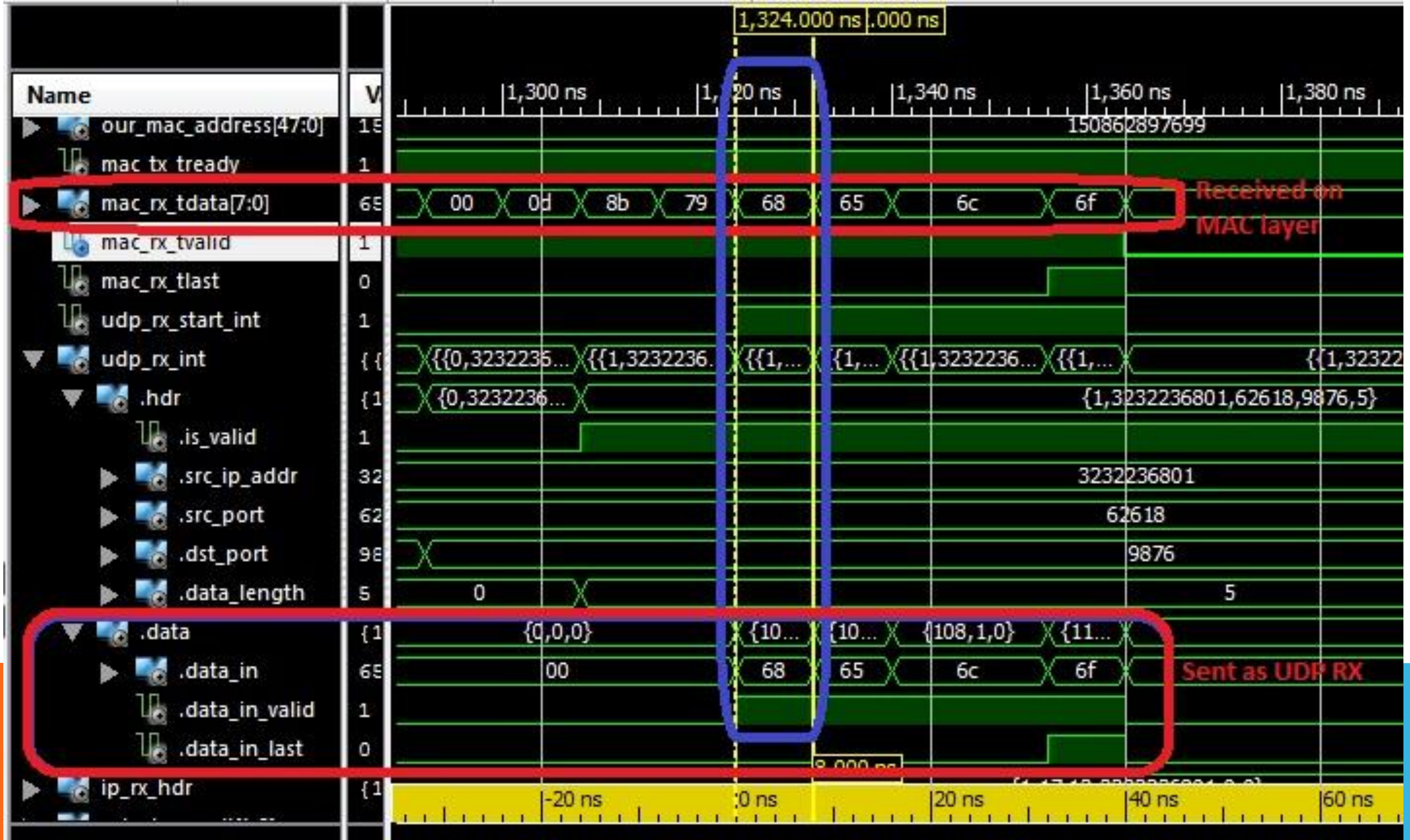
Easy to tap into the IP layer directly

Supports TX and RX with IP layer broadcast address

Separate clock domains for tx & rx paths

Tested for 1Gbit Ethernet, but applicable to 100M and 10M

SIMULATION DIAGRAM SHOWING ZERO LATENCY ON RECEIVE



LIMITATIONS

Does not handle segmentation and reassembly

- Assumes packets offerred for transmission will fit in a single ethernet frame
- Discards packets received if they require reassembly

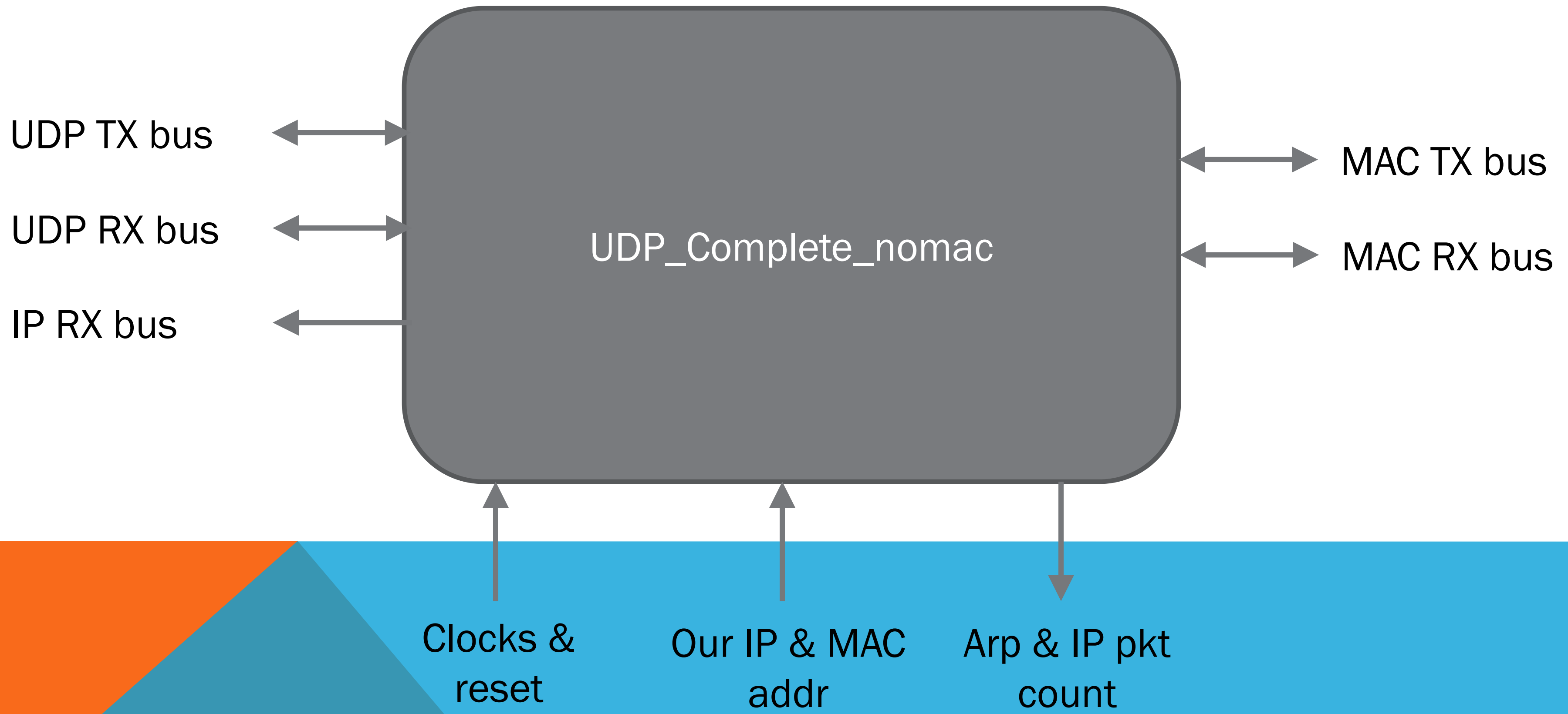
Currently implementing only one ARP resolution slot

- means only realistic to use for pt-pt cxns (but can easily extend ARP layer to manage an array of address mappings

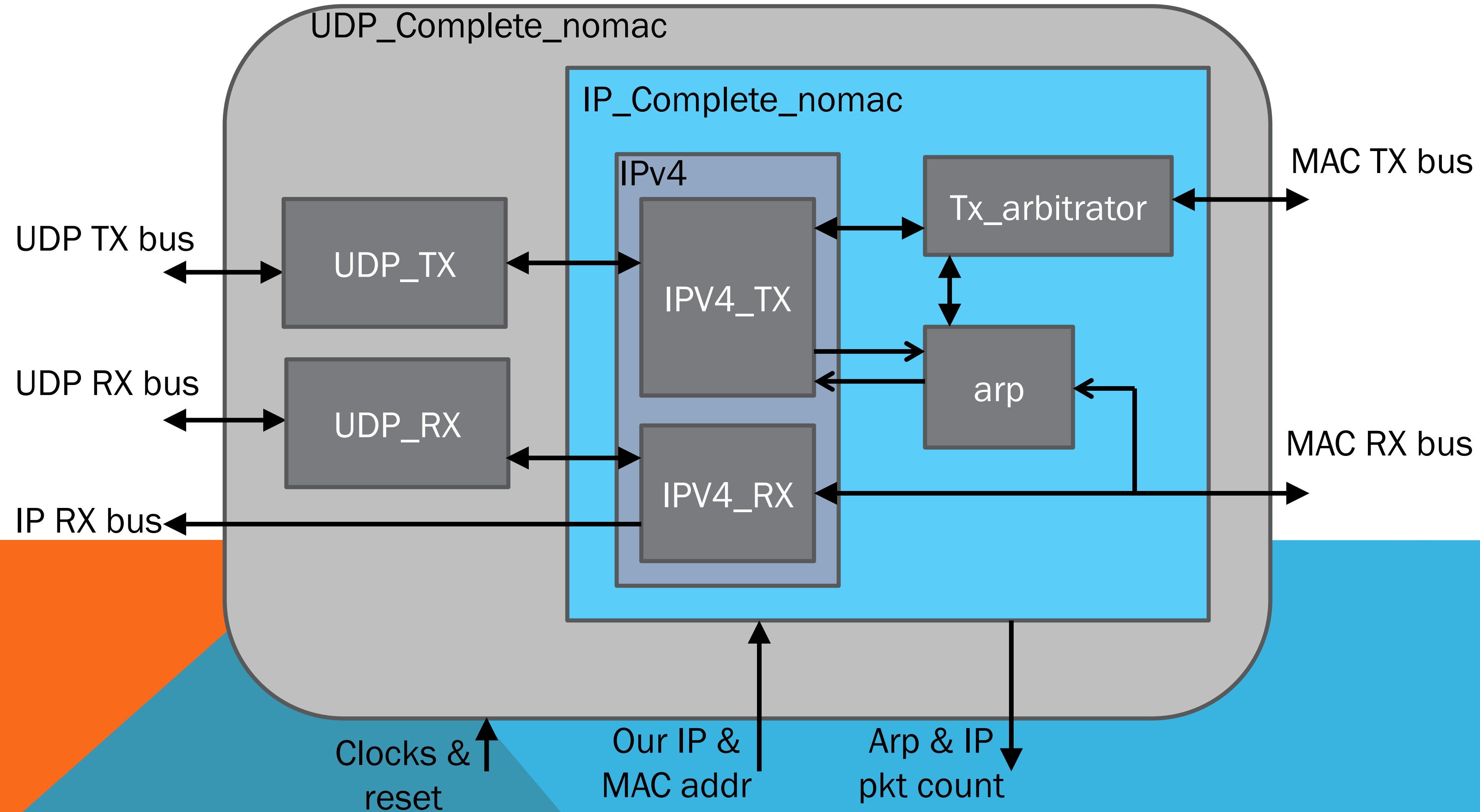
Doesnt always detect error situations (although these are flagged as TODO in the code)

Doesnt currently double register signals where they cross between tx & rx clock domain in a couple of places.

OVERALL BLOCK DIAGRAM



STRUCTURAL DECOMPOSITION



INTERFACE

```
entity UDP_Complete_nomac is
    Port (
            -- UDP TX signals
            udp_tx_start  : in std_logic;                      -- indicates req to tx UDP
            udp_txi       : in udp_tx_type;                    -- UDP tx cxns
            udp_tx_result : out std_logic_vector (1 downto 0); -- tx status (changes during tx)
            udp_tx_data_out_ready: out std_logic;              -- indicates udp_tx is ready to take data
            -- UDP RX signals
            udp_rx_start  : out std_logic;                     -- indicates receipt of udp header
            udp_rxo       : out udp_rx_type;
            -- IP RX signals
            ip_rx_hdr     : out ipv4_rx_header_type;
            -- system signals
            rx_clk        : in  STD_LOGIC;
            tx_clk        : in  STD_LOGIC;
            reset         : in  STD_LOGIC;
            our_ip_address  : in STD_LOGIC_VECTOR (31 downto 0);
            our_mac_address : in std_logic_vector (47 downto 0);
            -- status signals
            arp_pkt_count : out STD_LOGIC_VECTOR(7 downto 0);  -- count of arp pkts received
            ip_pkt_count  : out STD_LOGIC_VECTOR(7 downto 0);  -- number of IP pkts received for us
            -- MAC Transmitter
            mac_tx_tdata   : out  std_logic_vector(7 downto 0);          -- data byte to tx
            mac_tx_tvalid  : out  std_logic;                             -- tdata is valid
            mac_tx_tready  : in std_logic;                               -- mac is ready to accept data
            mac_tx_tfirst  : out  std_logic;                             -- indicates firstbyte of frame
            mac_tx_tlast   : out  std_logic;                             -- indicates last byte of frame
            -- MAC Receiver
            mac_rx_tdata   : in std_logic_vector(7 downto 0);            -- data byte received
            mac_rx_tvalid  : in std_logic;                               -- indicates tdata is valid
            mac_rx_tready  : out  std_logic;                   -- tells mac that we are ready to take data
            mac_rx_tlast   : in std_logic                      -- indicates last byte of the trame
            );
end UDP_Complete_nomac;
```

THE AXI INTERFACE

This implementation makes extensive use of the AXI interface (axi.vhd):

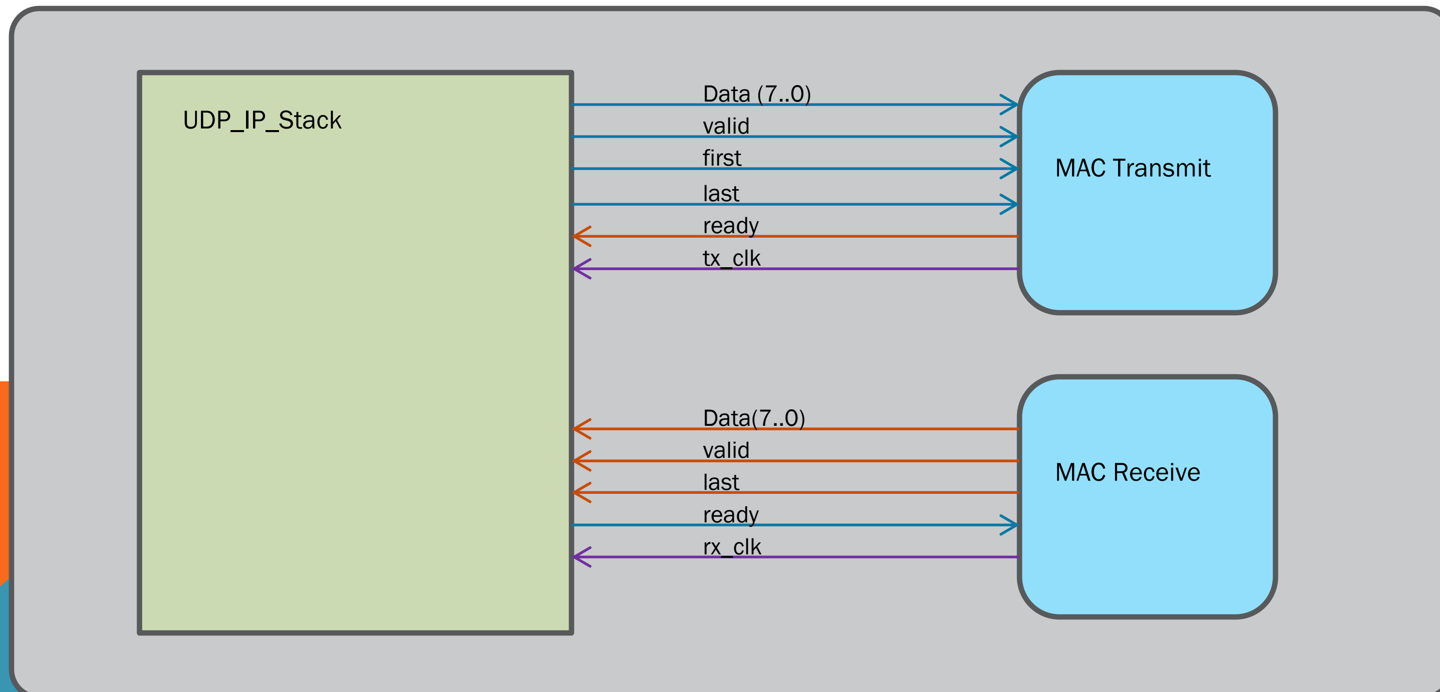
```
package axi is

    type axi_in_type is record
            data_in          : STD_LOGIC_VECTOR (7 downto 0);
            data_in_valid : STD_LOGIC;              -- indicates data_in valid on clock
            data_in_last  : STD_LOGIC;              -- indicates last data in frame
    end record;

    type axi_out_type is record
            data_out_valid   : std_logic;           -- indicates data out is valid
            data_out_last    : std_logic;           -- indicates last byte of a frame
            data_out             : std_logic_vector (7 downto 0);
    end record;

end axi;
```

MAC INTERFACE

The MAC interface is fairly simple with separate clocks for receiver and transmitter. Each interface (RX and TX) is based on the AXI interface and has an 8-bit data bus, a valid signal, a last byte signal, and a backchannel signal to indicate that the other end is ready to accept data.

The Transmit interface has an additional signal (`mac_tx_tfirst`) which can be used by MAC blocks that need something to indicate the start of frame. This signal is asserted simulaneous with the first byte to be transmitted (providing that tready is high).

On the following diagram, tx_clk and rx_clk are shown sourced from the MAC transmit and receive blocks, but can come from an independent clock generator that feeds clocks to both the MAC blocks and the UDP_IP_stack. Data is clocked on the rising edge.



SYNTHESIS STATS

504 occupied slices on Xilinx xc6vlx240t (1%)

(621 flipflops, 1243 LUTs)

Test synthesis using

- Xilinx ISE 13.2

MODULE DESCRIPTION: **UDP_COMPLETE_NOMAC**

Simply wires up the following blocks:

- UDP_TX
- UDP_RX
- IP_Complete_nomac

Propagates the IP RX header info to the UDP_complete_nomac module interface.

MODULE DESCRIPTION: UDP_TX AND UDP_RX

UDP_TX:

- Very simple FSM to capture data from the supplied UDP TX header, and send out a UDP header.
- Asserts data ready when in user data phase, and copies bytes from the user supplied data.
- Assumes user will supply the CRC (specs allow CRC to be zero).

UDP_RX

- Very simple FSM to parse the UDP header from data supplied from the IP layer, and then to send user data from the IP layer to the interface (asserts udp_rxo.data.data_in_valid).
- Discards IP pkts until it gets one with protocol=x11 (UDP pkt).

MODULE DESCRIPTION: IPV4

Simply wires up the following blocks:

- IPv4
- ARP
- Tx_arbitrator

Arp reads the MAX RX data in parallel with the IPv4 RX path. ARP is looking for ARP pkts, while IPv4 is looking for IP pkts.

IPv4 interacts directly with ARP block during TX to ensure that the transmit destination MAC address is known.

TX_arbitrator, controls access to the MAC TX layer, as both ARP and IPv4 may want to transmit at the same time.

MODULE DESCRIPTION: IPV4_TX

IPv4_TX comprises two simple FSMs:

- to control transmission of the header and user data
- to calculate the header checksum

To use,

- set the TX header, and assert ip_tx_start.
- The block begins to calculate the header CRC and transmit the header
- Once in the user data stage, the block asserts ip_tx_data_out_ready and copies user data over to the MAC TX output

MODULE DESCRIPTION: IPV4_RX

Simple FSM to parse both the ethernet frame header and the IP v4 header.

Ignores packets that

- Are not v4 IP packets
- Require reassembly
- Are not for our ip address and are not for the broadcast address

Once all these checks are satisfied, the rx header data: ip_rx.hdr is valid and the module asserts ip_rx_start.

Received user data is available through the ip_rx.data record.

MODULE DESCRIPTION: ARP

Handles receipt of ARP packets

Handles transmission of ARP requests

Handles request resolution (check ARP cache and request resolution if not found)

Three FSMs, one for each of the above functions

ARP mapper cache is only 1 deep in this implementation

- which means that it is only really good for point-point comms.
- Can easily be extended though for greater depth.

Input signals to module indicate our IP and MAC addresses

MODULE DESCRIPTION: TX_ARBITRATOR

FSM to arbitrate access to the MAC TX layer by

- IP TX path
- ARP TX path

One of the sources requests access and must wait until it is granted.

Priority is given to the IP path as it is expected that that path has the highest request rate.

SIMULATION

Every vdhl module has a corresponding RTL simulation test bench.

Additionally, there are simulation test benches for various module integrations.

In this version, verification is not completely automatic. The test benches test for some things, but much is left to manual inspection via the simulator waveforms.

TESTBENCH - HW

The HW testbench is built around the Xilinx ML-605 prototyping card.

It directly uses the card's 200MHz clocks, Eth PHY (copper) and LEDs to indicate status.

A simple VHDL driver module for the stack replies with a canned response whenever it receives a UDP pkt on a particular IP addr and port number.

The Xilinx LogiCORE IP Virtex-6 FPGA Embedded Tri-Mode Ethernet MAC v2.1 is used to couple the UDP/IP stack to the board's Ethernet PHY. This is used with the standard FIFO user buffering (which adds a one-frame delay). It should be possible also to remove this FIFO to reduce latency.

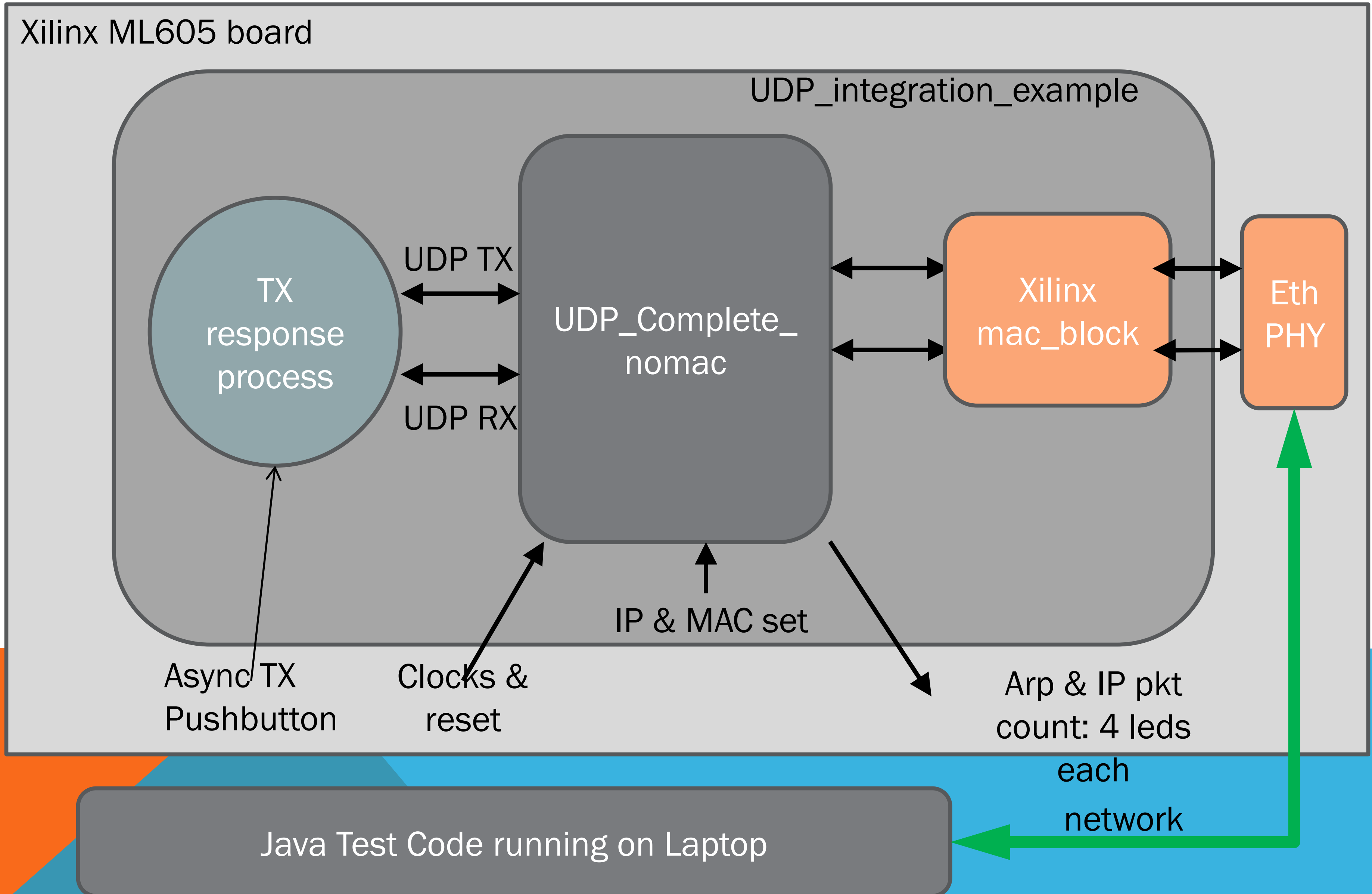
A laptop provides stimulus by way of one of two Java programs:

- UDPTest.java – writes one UDP pkt and waits for a response then prints it
- UDPTestStream.java – writes a number of UDP pkts and prints responses

The test network is a single twisted CAT-6 cable between the laptop and the ML-605 board.

Wireshark (on the laptop) is used to capture the traffic on the wire (sample pcap files are included)

TEST SETUP



TESTBENCH HW - ML605 MODULES

- UDP_Complete – integration of UDP with a mac layer
- IP Complete – integration of IP layer only with a mac layer
- UDP_Integration_Example – test example with vhdl process to reply to received UDP packets

TEST RESULTS

The xilinx MAC layer used contains a FIFO which therefore introduces a 1 frame delay.

- For tightly coupled low latency requirements, this can be removed.

Output from UDPTest:

- Sending packet: 1=45~34=201~18=23~ on port 2000
  Got [@ABC]

Output from UDPTestStream:

- …
  Sending price tick 205
  Sending price tick 204
  Sending price tick 203
  Sending price tick 202
  Got [@ABC]
  Got [@ABC]
  Got [@ABC]
  Got [@ABC]
  …