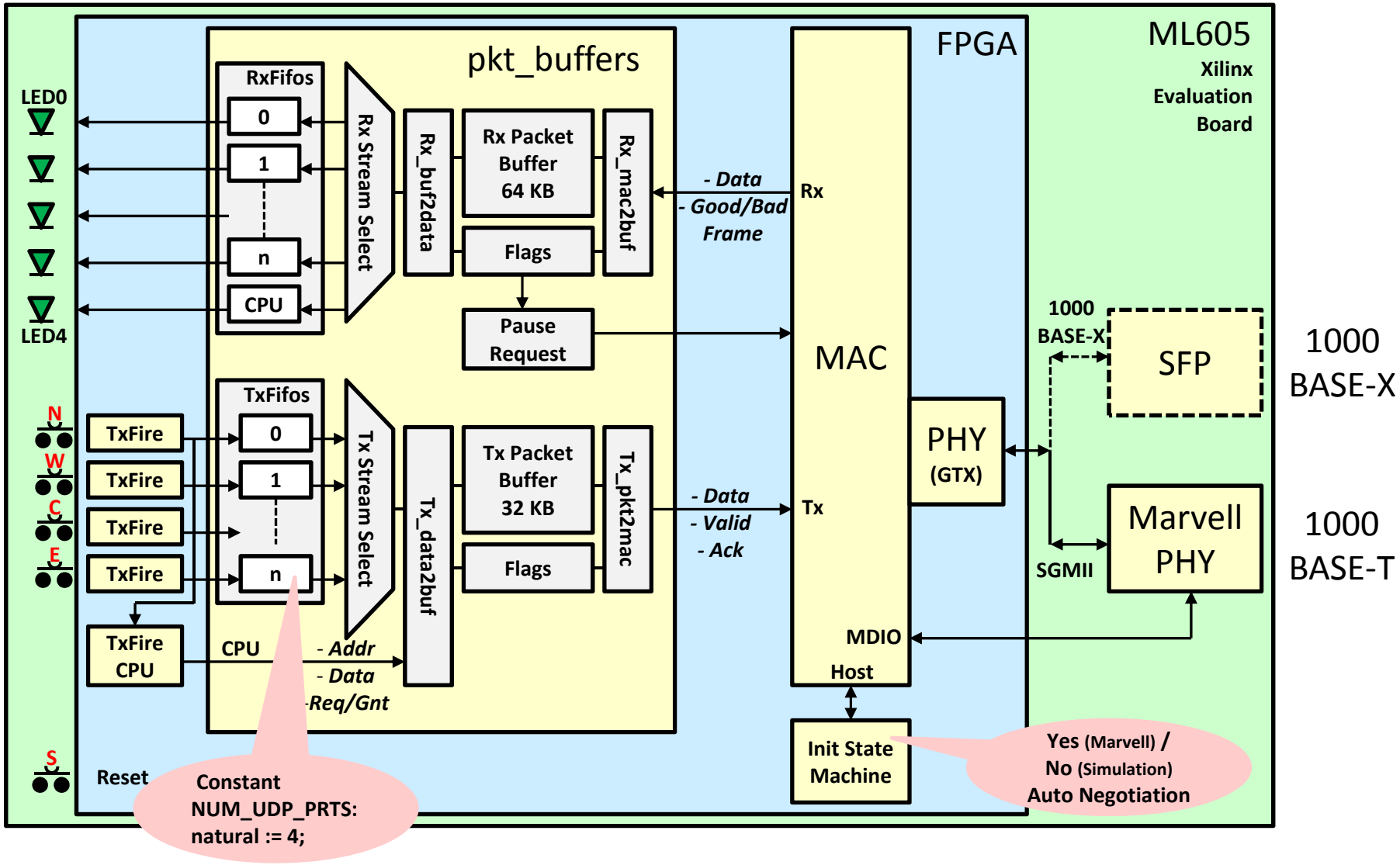
# GB ethernet UDP interface in FPGA

# Overview

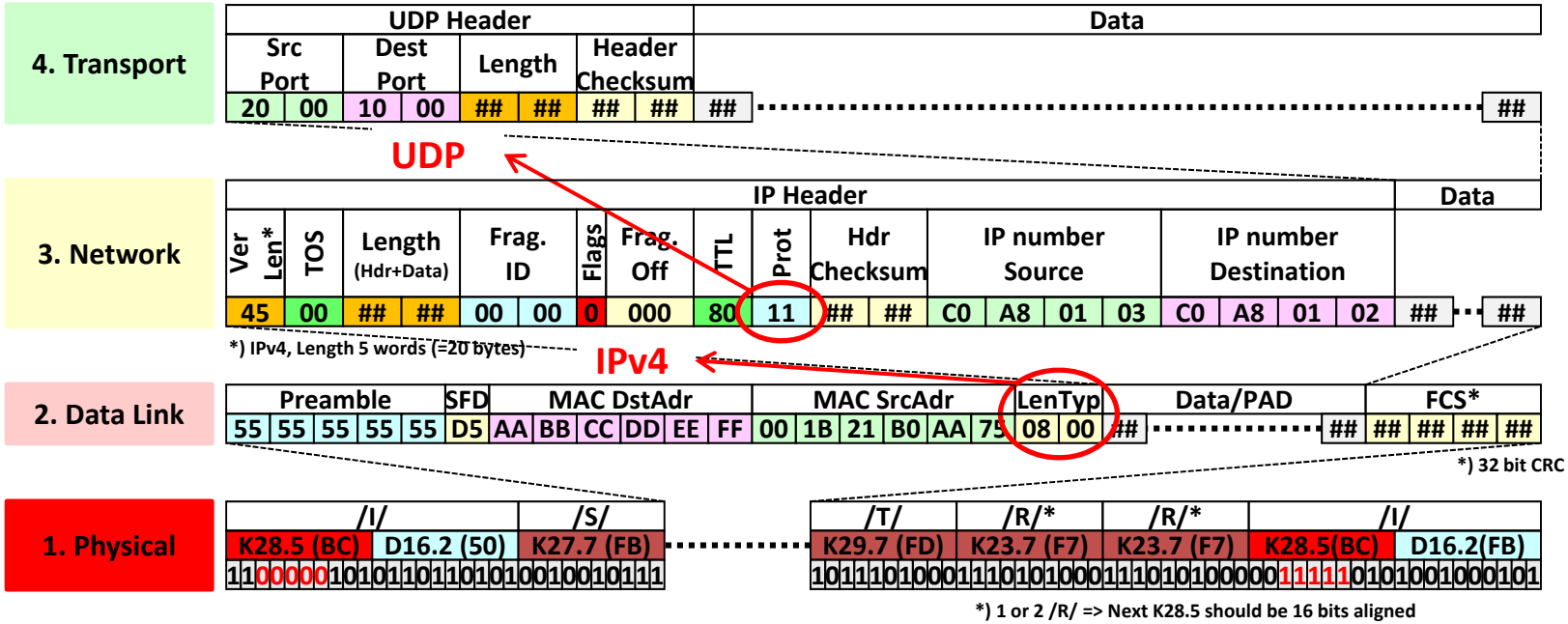

ML605
Xilinx
Evaluation
Board
FPGA
pkt_buffers
LED0
LED4
RxFifos
0
1
n
CPU
Rx Stream Select
Rx_buf2data
Rx Packet
Buffer
64 KB
Flags
Rx_mac2buf
- Data
- Good/Bad
Frame
Rx
Pause
Request
MAC
TxFifos
TxFire
TxFire
TxFire
TxFire
TxFire
CPU
Tx Stream Select
Tx_data2buf
Tx Packet
Buffer
32 KB
Flags
Tx_pkt2mac
- Data
- Valid
- Ack
Tx
- Addr
- Data
- Req/Gnt
PHY
(GTX)
1000
BASE-X
SFP
1000
BASE-X
SGMII
Marvell
PHY
1000
BASE-T
MDIO
Host
Init State
Machine
Yes (Marvell) /
No (Simulation)
Auto Negotiation
Reset
Constant
NUM_UDP_PRTS:
natural := 4;

# OSI Model

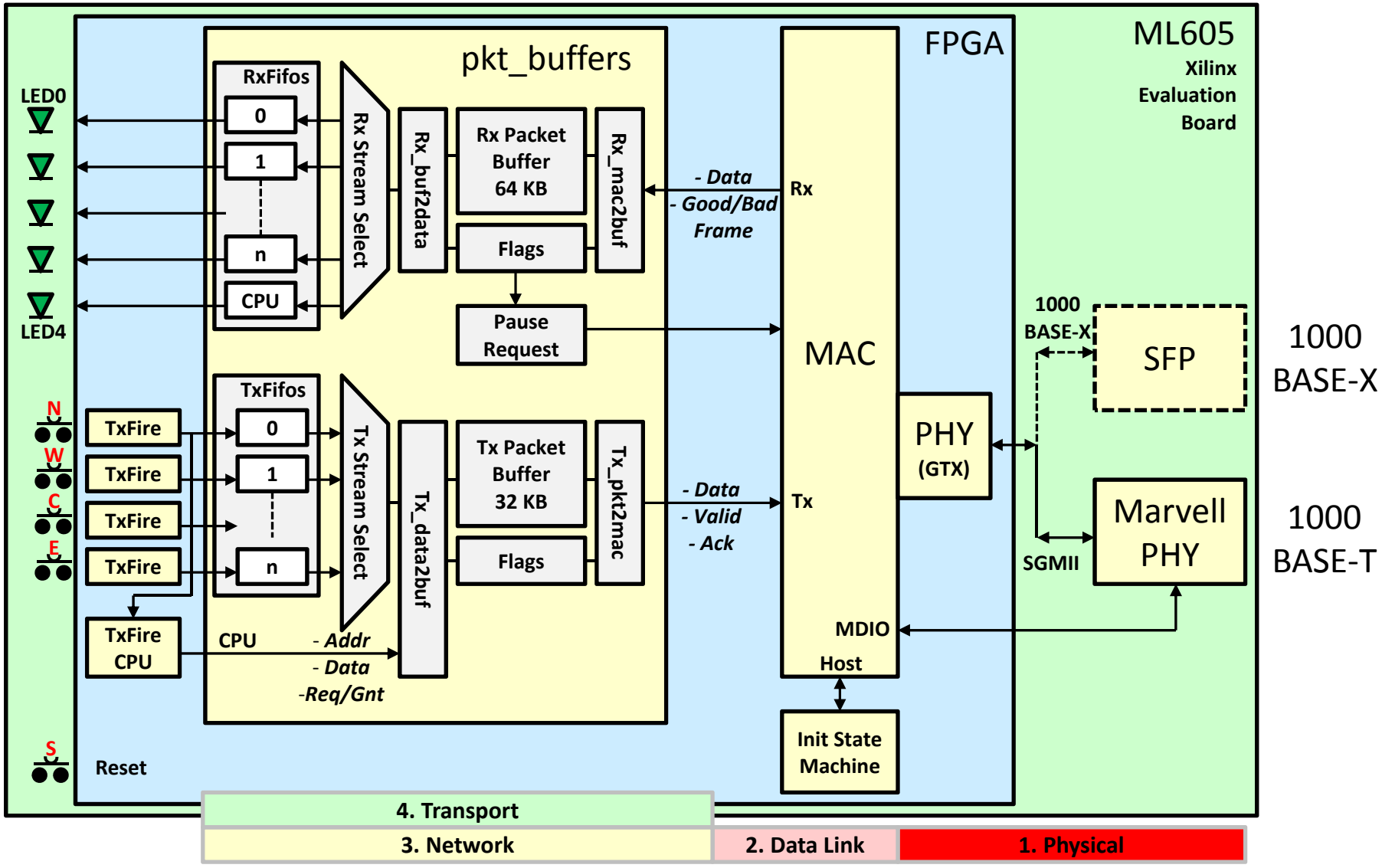
PTP over IEEE 802.3 Ethernet uses Ethertype 0x88F7

| | Data Unit | Layer | Standard | |
|---|---|---|---|---|
| Host Layers | Data | 7. Application | | |
| | | 6. Presentation | | |
| | | 5. Session | | |
| | Segment | 4. Transport | RFC-768 | UDP |
| Media Layers | Packet | 3. Network | RFC-791 | IPv4 |
| | Frame | 2. Data Link | IEEE 802.3 | MAC |
| | Bit | 1. Physical | | PHY |

PTP over IEEE 802.3 Ethernet uses Ethertype 0x88F7



# Overview

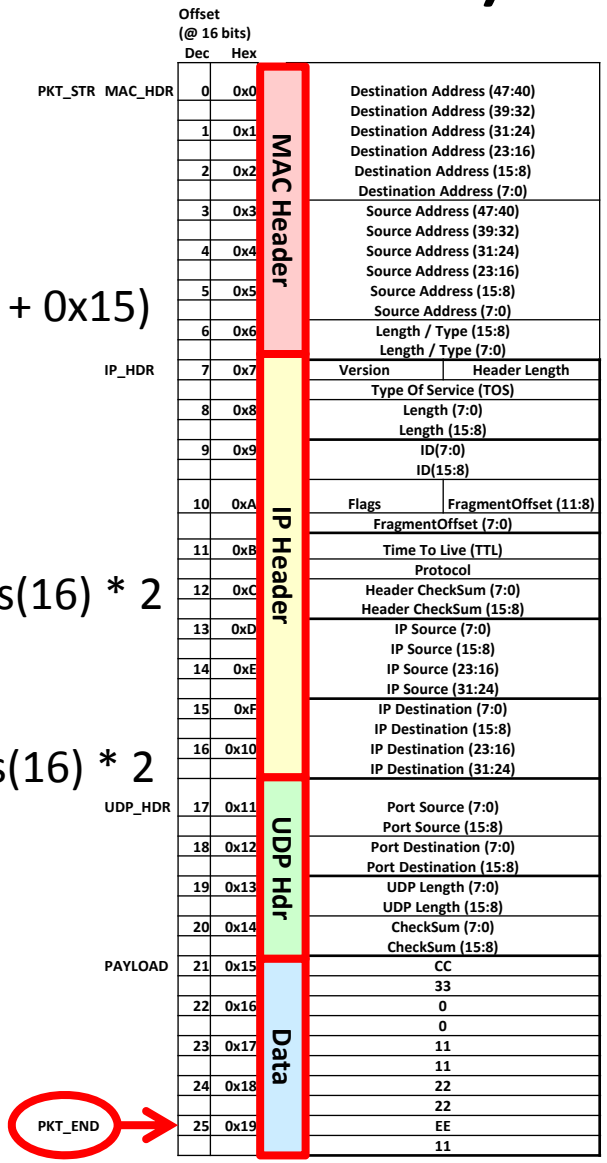

ML605
Xilinx
Evaluation
Board
FPGA
pkt_buffers
RxFifos
0
1
n
CPU
Rx Stream Select
Rx_buf2data
Rx Packet
Buffer
64 KB
Flags
Rx_mac2buf
Pause
Request
- Data
- Good/Bad
Frame
Rx
MAC
Tx
TxFifos
0
1
n
Tx Stream Select
Tx_data2buf
Tx Packet
Buffer
32 KB
Flags
Tx_pkt2mac
- Data
- Valid
- Ack
TxFire
TxFire
TxFire
TxFire
TxFire
CPU
CPU
- Addr
- Data
-Req/Gnt
MDIO
Host
Init State
Machine
PHY
(GTX)
1000
BASE-X
SFP
SGMII
Marvell
PHY
1000
BASE-X
1000
BASE-T
LED0
LED4
N
W
C
E
S
Reset
4. Transport
3. Network
2. Data Link
1. Physical

# tx_data2buf (input from FIFO)

1. Initialize fixed header values

If no CPU Access Request AND there is input data Then

2. Read input FIFO -> Start Write from PAYLOAD (offset + 0x15)
   Calculate UDP checksum (Pseudo Header)
3. Packet End := Last written offset
4. Write UDP header (offset + 0x11)
   Source/Dest port pointed by “stream”
   Length(bytes) = 8 (UDP Header) + number of data words(16) * 2
   Calculated UDP Checksum
5. Write IP header (offset + 0x07)
   Length(bytes) = 20 (IP Header) + number of data words(16) * 2
   Calculate IP Checksum
   Fragment-ID, Fragment-Offset, More-Fragments
6. Write MAC header (offset + 0x00)
   Type = EtherType x0800
7. Next Packet Pointer := Packet End + 1



# tx_data2buf (input from CPU)

1. Initialize fixed header values

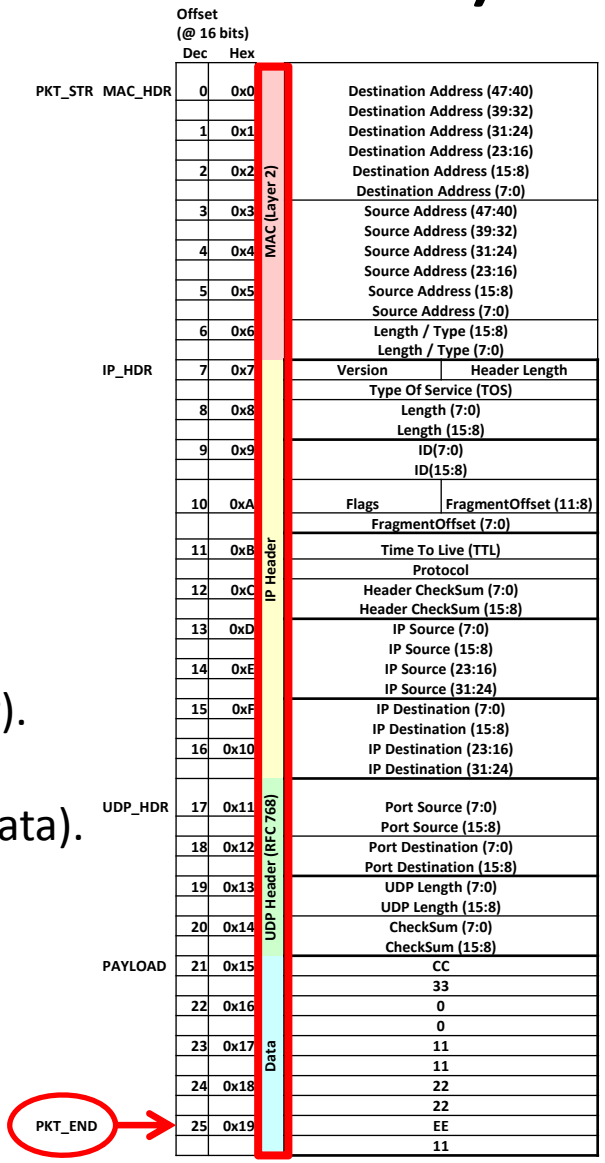
If there is a CPU Access Request Then

2. Start Write (offset + CPU_Addr)

Until CPU releases Access Request

Note: The last word of the packet must be accompanied with an End Of Packet (EOP ; Packet End := CPU_Addr) (note, since the CPU has random access to the packet buffer this doesn't necessarily mean that the last word of the packet is actually written last to the packet buffer). The CPU is responsible for generating the proper packet data format (i.e. MAC, IP etc. headers, checksums and data).

3. Next Packet Pointer := Packet End + 1

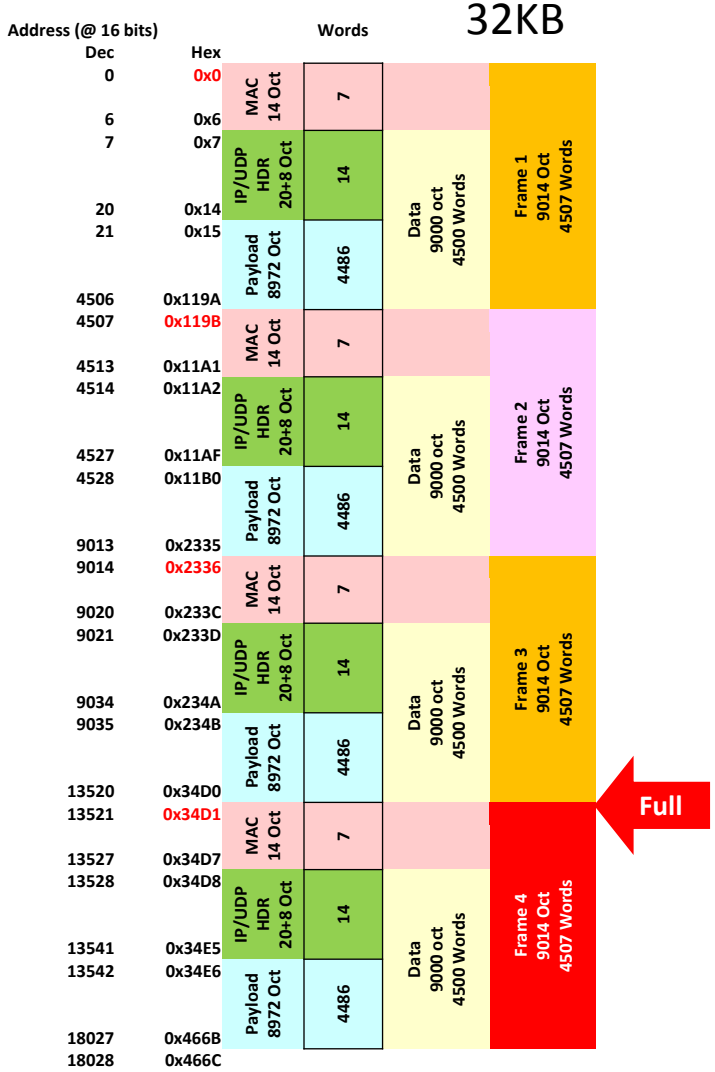


# Tx buffer memory flags

Tx Buffer of 32 KB can hold 3 complete Jumbo Frames (9000 bytes + MAC_Hdr{6 + 6 + 2} = 9014 bytes)

| Jumbo | Number of Bytes | % of ADRSIZE 14 (32678 Bytes) | % of ADRSIZE 15 (65536 Bytes) |
|---|---|---|---|
| 1 | 9014 | 27,51% | 13,75% |
| 2 | 18028 | 55,02% | 27,51% |
| 3 | 27042 | 82,53% | 41,26% |
| 4 | 36056 | 110,03% | 55,02% |
| 5 | 45070 | | 68,77% |
| 6 | 54084 | | 82,53% |
| 7 | 63098 | | 96,28% |
| 8 | 72112 | | 110,03% |
| 9 | 81126 | | |

Example: 32KB




Signal Full, as soon as there is no room left for a complete (Jumbo) packet

# rx_mac2buf

1. When Rx buffer is *not* Full then there is at least space for one Jumbo Frame
   (If Rx buffer is Full but MAC starts writing data then signal this with “Overflow”)
2. Assemble 8-bit MAC output bytes into 16-bit words and store them
3. As soon as the MAC signals
   “**Good Frame**” then increment the Packet Pointer
   “**Bad Frame**” then leave the Packet Pointer thus overwriting the bad frame with the next frame

# Rx buffer memory flags

Rx Buffer of 64 KB can hold 7 complete Jumbo Frames (9000 bytes + MAC_Hdr{6 + 6 + 2} = 9014 bytes)

| Jumbo | Number of Bytes | % of ADRSIZE 14 (32678 Bytes) | % of ADRSIZE 15 (65536 Bytes) |
|---|---|---|---|
| 1 | 9014 | 27,51% | 13,75% |
| 2 | 18028 | 55,02% | 27,51% |
| 3 | 27042 | 82,53% | 41,26% |
| 4 | 36056 | 110,03% | 55,02% |
| 5 | 45070 | | 68,77% |
| 6 | 54084 | | 82,53% |
| 7 | 63098 | | 96,28% |
| 8 | 72112 | | 110,03% |
| 9 | 81126 | | |


When “High Water Mark” is reached (i.e. buffer fills up after reception of 4 Jumbo Frames; Note that the Packet Pointer is only updated after the complete reception of the Packet), then a Pause Request (0xFFFF) is issued.

When “Low Water Mark” is reached (i.e. buffer empties and there is still 1 Jumbo Frame available), then a Pause Request (0x0000) is issued which means “Cancel the Pause Request”

# Pause Request

- Defined in See IEEE802.3-2005 Annex 31B
- Operates on the MAC-Control level:
  - Uses globally assigned 48-bit multicast address 01-80-C2-00-00-01
- One bit time = 0,8 ns @ 1Gb-ethernet
- One Pause Quanta = 512 bit times (= 512 * 0,8 ns = 409,6 ns)
- 0xFFFF Pause Quanta's = 26,8 ms.

# Pause Request test

- Hold Rx buffer readout while sending Jumbo Frames from PC to ML605 Evaluation kit
- Implemented a “EmptyRxBuffer” push button

./ipsar udp_tx_jumbo recv.bin
(Command sends 4 Jumbo frames)

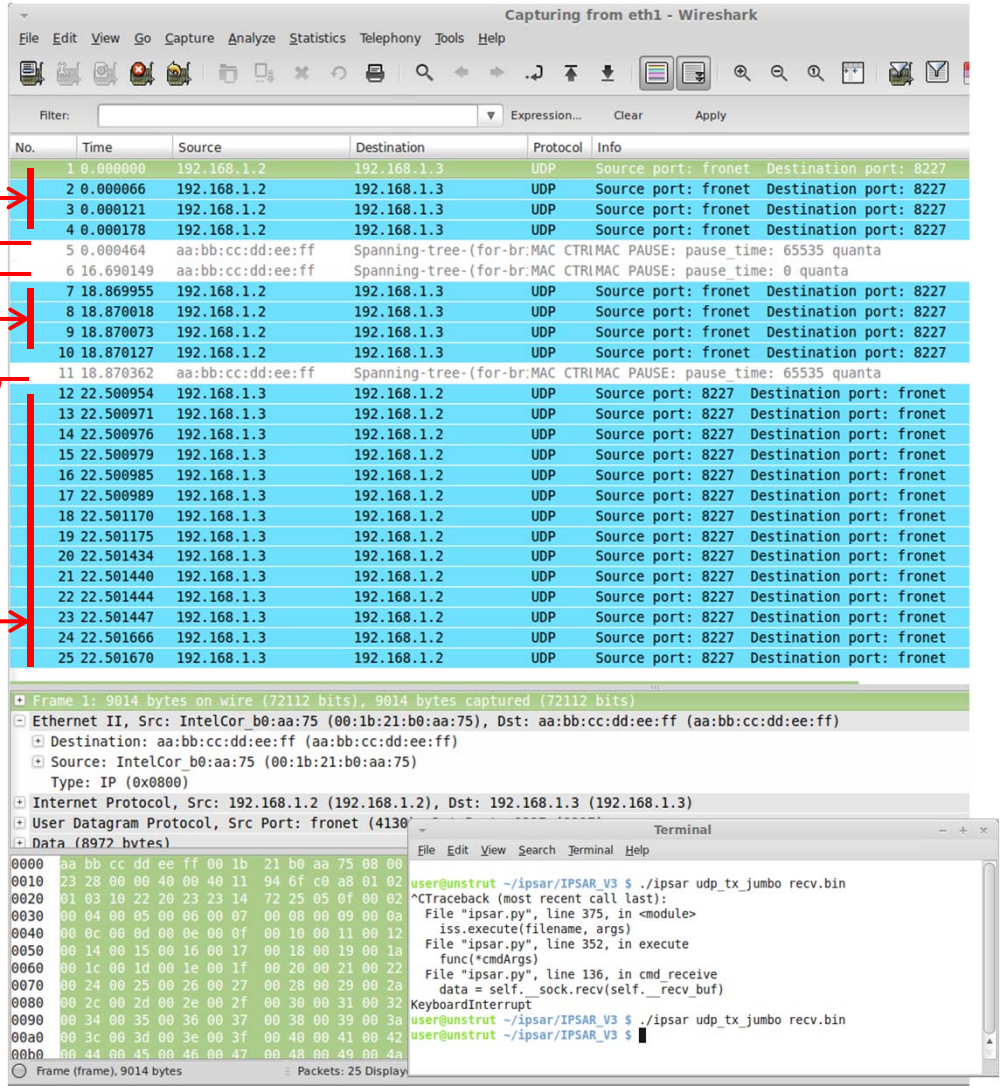
ML605 responds with Pause Request 65535

Push “EmptyRxBuffer” button (SW_North)
ML605 sends Pause Request 0

Send 4 Jumbo frames

ML605 responds with Pause Request 65535

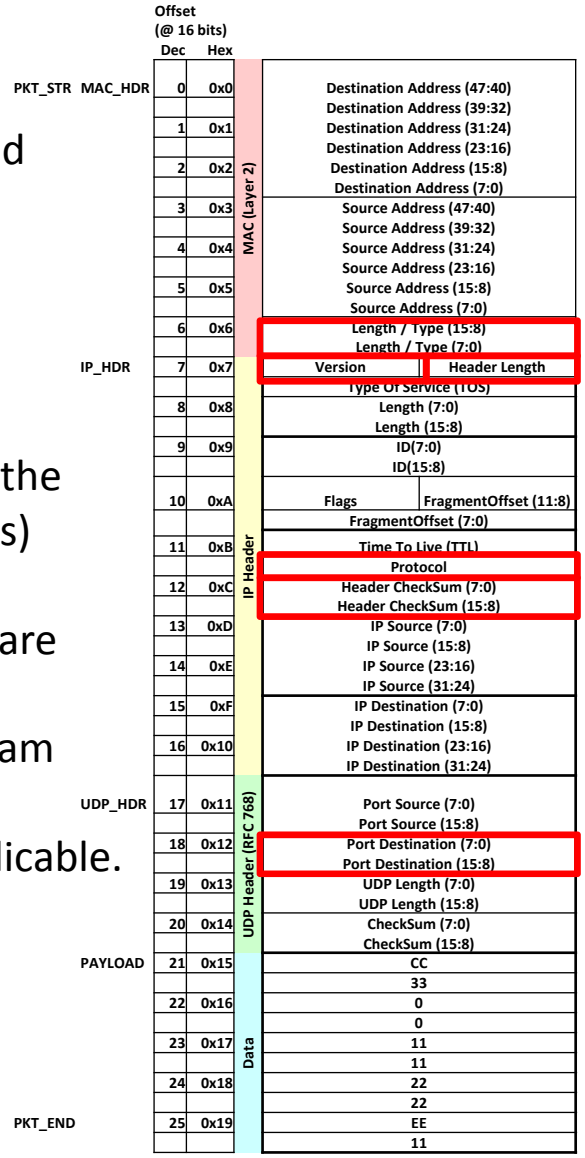
Receive 14 jumbo frames from M605 (pushed TxFire button (SW_Centre)



# rx_data2buf

1. If Rx buffer is *not* Empty then
2. Assumption is made that an IPv4 / UDP header is received
3. Check:
   EtherType = IPv4 (=0x0800)?
   IP header checksum?
   IP Version = 4 and IP Header Length = 5?
   Protocol = UDP (0x11)?
4. Check if the UDP destination port corresponds to one of the predefined (originating from either constants or registers) output streams
5. Fragment-ID, Fragment-Offset and More-Fragments flag are updated.
6. Only the data is routed to the corresponding output Stream FIFO
7. After routing, UDP checksum error is asserted when applicable.

All packets that fail the above checks are routed *integral* (i.e. MAC, IP, UDP header and Data) to the CPU Stream FIFO



# Checksum calculation

- IP and UDP Checksums
  - Checksum is a sum, not a CRC!
  - Add all and finally add the overflow (i.e. bits 31:16) to the lower 16 bits (15:0)
  - Again this may lead to an overflow in bit 16 (thanks to Frans Schreuder for debugging). For example 0xffff + 0xffff = 0x1fffe. This overflow bit 16 also needs to be added to (15 downto 0).
  - Take the ones complement of the lower 16 bits => this is the checksum
- IP Checksum
  - Mandatory
  - Calculated over the IP header (Not including IP header checksum field which is added as 0x0000)
- UDP Checksum
  - Optional
  - Calculated using a UDP *Pseudo-Header* that contains:
    - IP Source address
    - IP Destination address
    - x"00" & Protocol
    - UDP Length
  - Further calculation includes UDP Header:
    - UDP Source Port
    - UDP Destination Port
    - UDP Length (Note that UDP Length is present twice (in the Pseudo header and in the UDP header)
  - UDP Checksum is the sum over the Pseudo Header, UDP header and all data words.

# Resources

Entity “pkt_buffers”

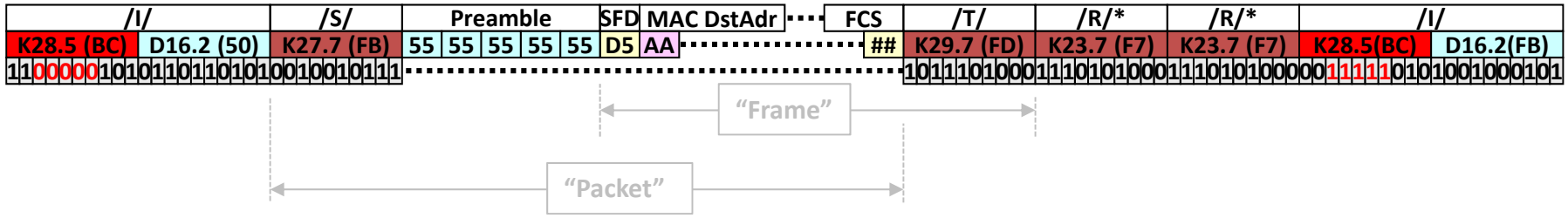
implementing 4 Tx/Rx Streams:

- Dffs or Latches 644
- Block RAMs
  - 32 KB + 64 KB, Packet buffers = 24 * RAMB36E1
  - 5 Rx FIFOs, 4 Tx FIFOs = 9 * RAMB18E1

Virtex-6 “xc6vlx240t” => 5% RAM, < 1% Slices

# Frame and Packet Defenitions

IEEE802.3-2008
Clause 30 (see also 46.2.5)



# Frame and Packet Defenitions

| | Data Unit | Layer | Standard | |
|---|---|---|---|---|
| Host Layers | Data | 7. Application | | |
| | | 6. Presentation | | |
| | | 5. Session | | |
| | Segment | 4. Transport | RFC-768 | UDP |
| Media Layers | Packet | 3. Network | RFC-791 | IPv4 |
| | Frame | 2. Data Link | IEEE 802.3 | MAC |
| | Bit | 1. Physical | | PHY |