# Front-End RDMA Over Converged Ethernet, lightweight RoCE endpoint

Gabriele Bortolato[a,b,c], Antonio Bergnoli[a,b], Damiano Bortolato[d], Daniele Mengoni[a,b], Matteo Migliorini[c], Fabio Montecassiano[a], Jacopo Pazzini[a,b,e,f], Andrea Triossi[a,b], Sandro Ventura[a], Marco Zanetti[a,b]

[a]INFN sez. Padova, [b]DFA Padova University, [c]CERN, [d]INFN LNL, [e]DEI Padova University, [f]DII Padova University

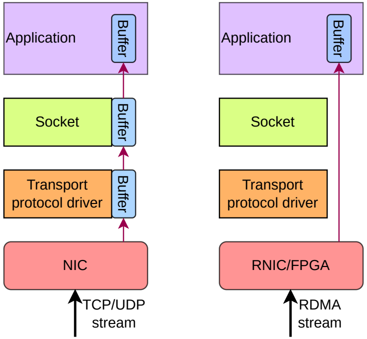
Finanziato dall'Unione europea
NextGenerationEU

Italiadomani
PIANO NAZIONALE DI RIPRESA E RESILIENZA

ICSC
Centro Nazionale di Ricerca in HPC, Big Data and Quantum Computing

# Introduction on RDMA and RoCE

In a DAQ system a large fraction of CPU resources is engaged in networking rather than in data processing; common network stacks that take care of network traffic usually manipulate data through several copies.



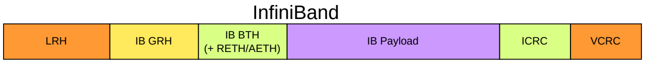
Remote Direct Memory Access (RDMA), as the name suggests, allows read and write operations directly in the target machine(s). This implies no OS involvement allowing high-throughput and low-latency applications.

This requires RDMA enabled NICs on both ends (RNIC) that perform the DMA, reducing the CPU load.

## RDMA protocols

Many RDMA flavours are available:

- InfiniBand, it requires IB capable switches
- RoCEv1, it introduces the Ethernet framing, enable use of commodity switches
- RoCEv2, it adds the UDP/IP transport protocol

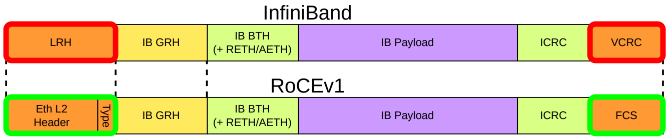


- Local and Global Route Headers (L2 and L3 respectively)
- Base and Extended Transport Headers (L4)

## RDMA protocols

Many RDMA flavours are available:

- InfiniBand, it requires IB capable switches
- RoCEv1, it introduces the Ethernet framing, enable use of commodity switches
- RoCEv2, it adds the UDP/IP transport protocol

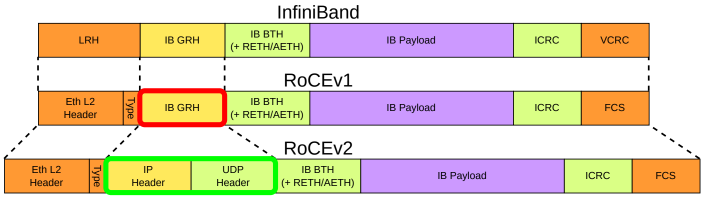


- Eth L2 Header instead of LRH

## RDMA protocols

Many RDMA flavours are available:

- InfiniBand, it requires IB capable switches
- RoCEv1, it introduces the Ethernet framing, enable use of commodity switches
- RoCEv2, it adds the UDP/IP transport protocol

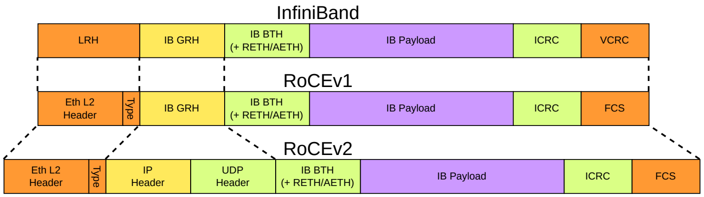


- Drop the use of Global ID (GID) in favour of IP (RoCEv2 UDP port number 4791)

# RDMA protocols

Many RDMA flavours are available:

- InfiniBand, it requires IB capable switches
- RoCEv1, it introduces the Ethernet framing, enable use of commodity switches
- RoCEv2, it adds the UDP/IP transport protocol ←



RoCEv2 is the only industry-standard Ethernet-based RDMA solution with a multi-vendor ecosystem. For this reason it has been chosen as target protocol.

Honourable mention

- iWARP, congestion-aware protocols, but higher complexity

## Front-End RDMA over Converged Ethernet

# Front-End RDMA over Converged Ethernet

Constant trend in producing larger and larger dataset in almost every experimental physics field, new requirements arise form that:

- High throughput, low latency
- Efficient data movement

## Front-End RDMA over Converged Ethernet

Constant trend in producing larger and larger dataset in almost every experimental physics field, new requirements arise form that:

- High throughput, low latency
- Efficient data movement

Such requirements lead to clever ideas and features:

- Zero-copy protocols such as InfiniBand or RoCE
- Move network protocol directly in the front-end electronics (FPGA)
- Need to be scalable 1/10/100 Gb/s to target different scenarios
- Multi-vendor ecosystem Xilinx/Microchip/Altera

## Front-End RDMA over Converged Ethernet

Constant trend in producing larger and larger dataset in almost every experimental physics field, new requirements arise form that:

- High throughput, low latency
- Efficient data movement

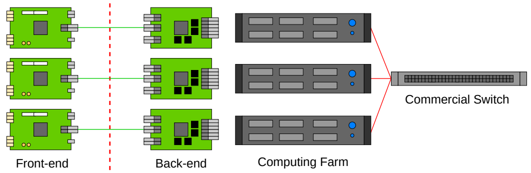
Such requirements lead to clever ideas and features:

- Zero-copy protocols such as InfiniBand or RoCE
- Move network protocol directly in the front-end electronics (FPGA)
- Need to be scalable 1/10/100 Gb/s to target different scenarios
- Multi-vendor ecosystem Xilinx/Microchip/Altera

What can we achieve?

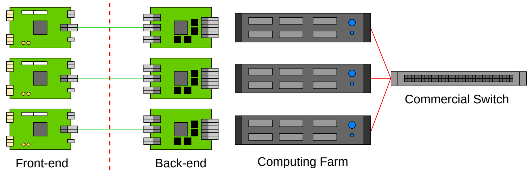
- Front-end initiates the RDMA transfer
- No point-to-point connection between front-end back-end
- Dynamical switching routing with COTS (lowering the costs and maintenance)

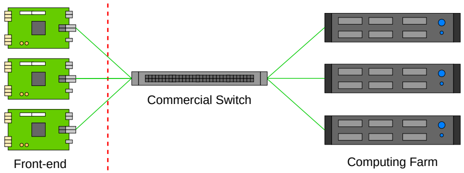
# What is FERoCE?



Back-end boards required to get the data, and send it to the computing farms. This requires multiple custom cards and custom boards

# What is FERoCE?

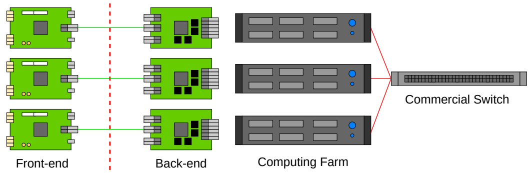


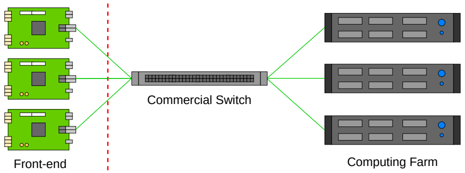


Back-end boards required to get the data, and send it to the computing farms. This requires multiple custom cards and custom boards

Front-end boards send data already packaged within an ethernet frame allowing switching and routing. Choosing the proper protocol allows the use of COTS switches

# What is FERoCE?





Back-end boards required to get the data, and send it to the computing farms. This requires multiple custom cards and custom boards

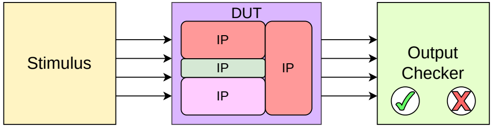
Front-end boards send data already packaged within an ethernet frame allowing switching and routing. Choosing the proper protocol allows the use of COTS switches

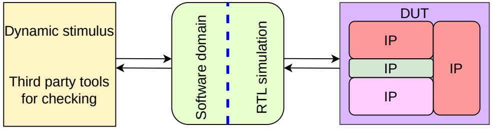
Alex Forencich Ethernet components repository has been chosen as frame. Some of its characteristics:

- Entirely written in Verilog (HW portable!)
- Handwritten MAC
- It supports 10/25G
- Multiple protocols ETH, ARP, IP and UDP

# Real-time Firmware simulation

Why a dynamic firmware simulation is needed?





- Narrow test-case, limited by the stimulus
- Difficult to evaluate the RoCE stream produced
- Easy to set-up

- Explore wider test-case phase space
- Feed/Get ethernet frames directly to/from the code
- Simulate the HDL code even if produced with HLS code
- Capture frames with third party programs (e.g. Wireshark)
- Possibility to treat it as a device and send frames to Soft-RoCE or to a physical RNIC

## Real-time Firmware Simulation

Start form Alex Forencich network stack. Functionalities and features must be understood: real-time firmware simulation with real network traffic.

## Real-time Firmware Simulation

Start form Alex Forencich network stack. Functionalities and features must be understood: real-time firmware simulation with real network traffic.

- Works on Linux machines: Tun/Tap devices

# Real-time Firmware Simulation

Start form Alex Forencich network stack. Functionalities and features must be understood: real-time firmware simulation with real network traffic.

- Works on Linux machines: Tun/Tap devices
- It makes use of DPI-C interface of SystemVerilog: C code in our testbench!

## Real-time Firmware Simulation

Start form Alex Forencich network stack. Functionalities and features must be understood: real-time firmware simulation with real network traffic.

- Works on Linux machines: Tun/Tap devices
- It makes use of DPI-C interface of SystemVerilog: C code in our testbench!
- Tap device exchanges raw ethernet frames between simulation and Linux network stack

## Real-time Firmware Simulation

Start form Alex Forencich network stack. Functionalities and features must be understood: real-time firmware simulation with real network traffic.

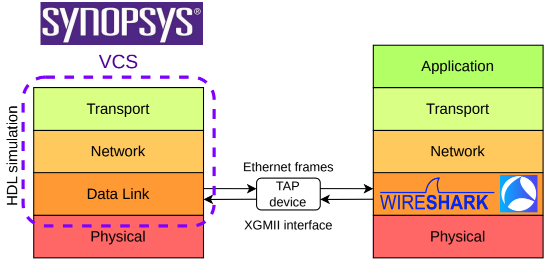
- Works on Linux machines: Tun/Tap devices
- It makes use of DPI-C interface of SystemVerilog: C code in our testbench!
- Tap device exchanges raw ethernet frames between simulation and Linux network stack
- We can capture such frames (and waveforms) and study them

# Real-time Firmware Simulation

Start form Alex Forencich network stack. Functionalities and features must be understood: real-time firmware simulation with real network traffic.

- Works on Linux machines: Tun/Tap devices
- It makes use of DPI-C interface of SystemVerilog: C code in our testbench!
- Tap device exchanges raw ethernet frames between simulation and Linux network stack
- We can capture such frames (and waveforms) and study them

Simulation with Synopsys VCS. XGMII interface directly from 10G MAC



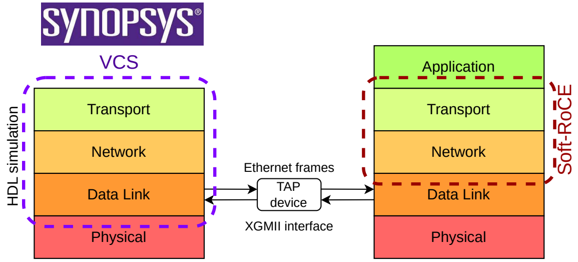
Capture and analyze packets, are they malformed? Are the RoCE parameters sent correctly?

# Real-time Firmware Simulation

Start from ETH network stack entirely developed in HLS. Functionalities and features must be understood: real-time firmware simulation with real network traffic.

- Works on Linux machines: Tun/Tap devices
- It makes use of DPI-C interface of SystemVerilog: C code in our testbench!
- Tap device exchanges raw ethernet frames between simulation and Linux network stack
- We can capture such frames and study them

Simulation with Synopsys VCS. XGMII interface directly from 10G MAC

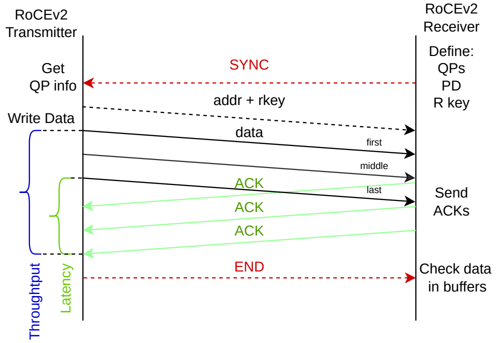


Soft-RoCE used to capture and store in memory data sent. Enable fast verification of the stack without going through sythesis/implementation every time.

Once the stack has been verified, firmware can be eventually built (Resources? Performances? Is timing closure reached?)

# RoCE

RoCEv2 is a complex protocol, but not all its features are required for this project. RoCE supports many operations such as: RDMA SEND, RDMA WRITE, RDMA READ, ATOMIC OPERATIONS.



⚠ SYNC and END* messages are outside the RoCE protocol ⚠

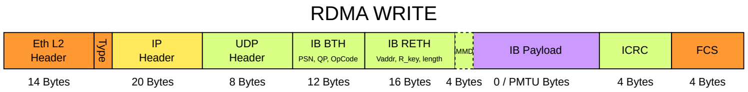
*RDMA WRITE IMMEDIATE command exists that trigger a completion message upon finishing.

The goal is only to push data and initiate the RDMA transfer, for this reason only RDMA WRITE is considered.

## What was added to the stack

Some new modules has been designed:

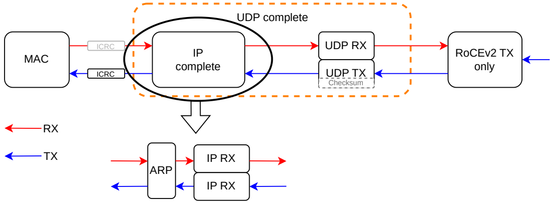
- ICRC module:
  - For 10/25G we need to compute CRC32 for 64b data words at 156.25 MHz or 390.625 MHz. Module that compute FCS was sligthly modified and used.
  - New module for 100G, parallel computation is needed here. CRC computed for 512b data words at 322.266 MHz, not that easy.
- Added RoCE TX module:
  - Supports RDMA WRITE and RDMA WRITE WITH IMMEDIATE operations
  - FSM to correctly split AXI data stream in RDMA WRITE ONLY, FIRST, MIDDLE or LAST, based on the selected PMTU



- Added RoCE RX module:
  - Only RDMA ACK packets are decoded
  - Used for latency and throughput measurements
- Added a very rough way to exchange QP info via UDP:
  - PC create a QP and sends its info via a UDP packet
  - FPGA receive this packet and sets the QP parameters in the FPGA registers

# RoCEv2 FPGA stack

Work based on Alex Forencich UDP/IP network stack with some minor modifications. (e.g. RoCEv2 requires the UDP checksum to be set to 0).



| Speed | Datapath | CLK Frequency |
|---|---|---|
| 10G | 64b | 156.250 MHz |
| 25G | 64b | 390.625 MHz |
| 100G | 512b | 322.266 MHz |

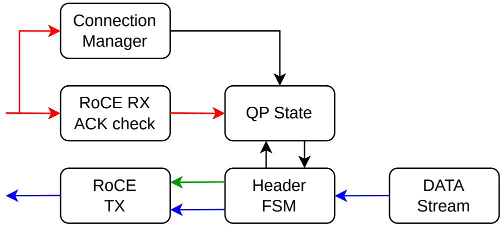
100G speed requires new ICRC module to cope with the higher bitwidth and high clock frequency.

- Data is transferred between modules using AXI4-Stream interface.
- ICRC not computed at the RX side, but we have the logic to discard packet with bad ICRC. Added to TODO list :)
- UDP checksum completely disabled, need to re-enabled it if UDP only payload is sent.

# RoCEv2 TX diagram

RDMA data transfer is not as straight forward as a UDP/IP one!

- QP created at the server side
- QP info sent via UDP to the FPGA
- FPGA can start sending data
- Notify somehow the server the end of transfer
- Latency and Throughput measured with the PSN of the sent packet and received ACK
- Without re-transmission, if a packet is not received properly the connection must be closed. Need a lossless network!



As first test, simple counters sent as data payload, easy to check them at the server's buffer.

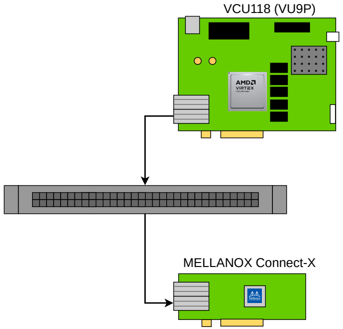
# Throughput and Latency point-to-point

Tests were performed at 10G and 25G, with a PMTU of 4096B.

- FPGA connected to the NIC through a switch, only two endpoints
- Latency and throughput evaluated thanks to the ACK packet received

| Speed | Msg.size [kB] | Latency [$\mu s$] | Tot. Throughput [Gbps] |
|---|---|---|---|
| 10G | 262 | 4.6 | 9.64 |
| 25G | 262 | 5.0 | 24.10 |
| 100G | - | - | - |

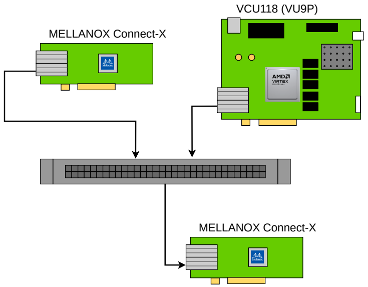
Test done where the latency and throughput plateau. Theoretical max throughput is 98.5% with PMTU=4096 or 97.1% with PMTU=2048 of the maximum speed available (headers 14+20+8+12+16+4+4 Bytes).



# Throughput and Latency with congestion

Tests were performed at 10G and 25G, with a PMTU of 4096B.

- All participants set at the same speed (10G or 25G), forcing congestion on the receiver Connect-X
- Pause frames sent to stall the TX stream, latency will increase
- Total throughput should not change to the point-to-point test

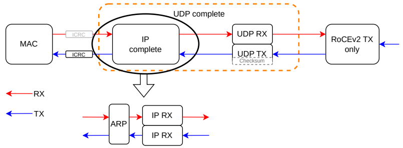


| Speed | Msg.size [kB] | Latency [$\mu s$] | Tot. Throughput [Gbps] |
|---|---|---|---|
| 10G | 262 | 13.2 | 9.63 |
| 25G | 262 | 20.0 | 24.09 |
| 100G | - | - | - |

## Summary and Outlook

### Summary

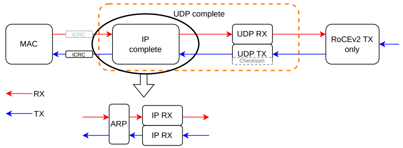
- Developed a dynamic simulation
- Written a simplified RoCE transmitter in verilog
- Used dynamic-simulation to test the new code
- Implemented and evaluated stack at 10/25G speeds



# Summary and Outlook

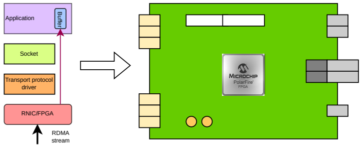
## Summary

- Developed a dynamic simulation
- Written a simplified RoCE transmitter in verilog
- Used dynamic-simulation to test the new code
- Implemented and evaluated stack at 10/25G speeds



## Outlook

- Finalize and optimize stack at 100G
- Explore other QoS feature that RoCEv2 has, e.g. ECN
- Deploy the light-RoCE in a Microchip FPGA, targeting 10G speed



## BACKUP

## RoCEv2 operations

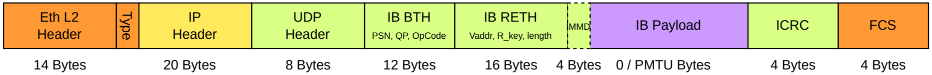
Four different connection types

| Type | ACK/NAK protocol | Private |
|---|---|---|
| RC | Yes | Yes |
| UC | No | Yes |
| RD | Yes | No |
| UD | No | No |

Multiple message types

- RDMA WRITE
- SEND
- RDMA READ
- ATOMIC

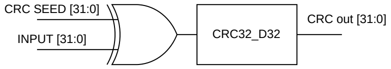
RDMA WRITE

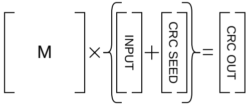


## CRC32 operation

CRC32 computation can be seen as matrix multiplication:

$$CRC = M \times (I + S)$$





Where

- M is the matrix related to the CRC computation (generated form the polynomial)
- I is the 32-bit input
- S is the CRC seed or initial value (usually set to 0xFFFFFFFF)
- $\times$ is the AND operation
- $+$ is the XOR operation

## CRC32 operation

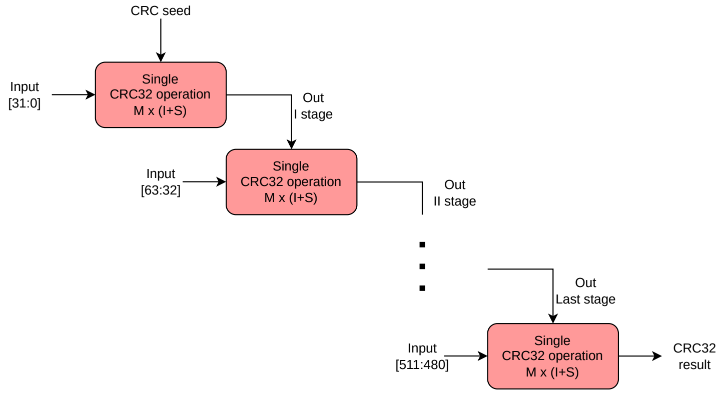
$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{031} \\ a_{10} & a_{11} & \cdots & a_{131} \\ \vdots & \vdots & \ddots & \vdots \\ a_{310} & a_{311} & \cdots & a_{3131} \end{bmatrix} \times \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{31} \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{31} \end{bmatrix}$$

$$c_i = a_{i0} \wedge b_0 \oplus a_{i1} \wedge b_1 \oplus \cdots \oplus a_{in} \wedge b_n = \sum_{k=0}^{31} a_{ik} \wedge b_k$$

Where the matrix is generated starting from the polynomyal, endianess and shift direction

## CRC32 operation

Now if we start applying the operation to subsequent 32-bit data we obtain the diagram below.



## CRC32 operation

For example let's consider the computation for a 64-bit data:

$$CRC_{tot} = M \times (M \times (I_0 + S) + I_1)$$

Where

- $M$ is the CRC32 matrix
- $I_0$ and $I_1$ are the 32-bit data slices
- $S$ is the CRC seed, usually set to 0xFFFFFFFF

Expanding the product:

$$CRC_{tot} = M \times (M \times (I_0 + S) + I_1) = M^{(2)} \times I_0 + M^{(1)} \times I_1 + M^{(2)} \times S$$

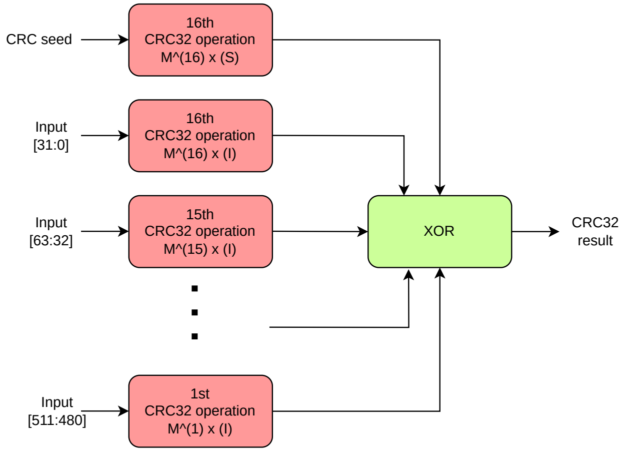
Now for a 512-bit data vector

$$CRC_{tot} = M \times I_{15} + M^{(2)} \times I_{14} + M^{(3)} \times I_{13} + \cdots + M^{(16)} \times I_0 + M^{(16)} \times S$$

Or written in a shorter manner:

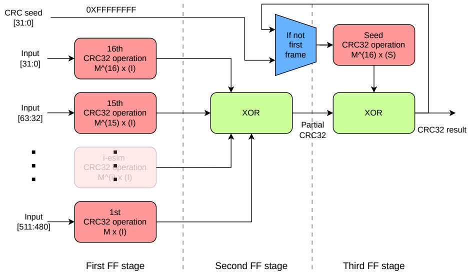
$$CRC_{tot} = \left( \sum_{(i=0)}^{15} M^{(i+1)} \times I_{15-i} \right) + M^{(16)} \times S$$

# CRC32 operation parallel



# CRC implementation

With the last equation we can pre-compute the various matrices and apply them to the various slices.
The final result will be simply the XOR of the results.
Such computation can be pipelined to achieve the $\sim$ 322 MHz target frequency.
In the design 3 stages were used [1], possibility to stream with different keep values (still need to be multiple of 32).


CRC seed [31:0]
0XFFFFFFFF
Input [31:0]
16th CRC32 operation M^(16) x (I)
Input [63:32]
15th CRC32 operation M^(15) x (I)
i-esim CRC32 operation M^(i) x (I)
Input [511:480]
1st CRC32 operation M x (I)
XOR
If not first frame
Seed CRC32 operation M^(16) x (S)
XOR
Partial CRC32
CRC32 result
First FF stage
Second FF stage
Third FF stage

[1] Matrix multiplications with data slices, XOR, XOR with CRC SEED result

## Matrix-Matrix multiplication

The generating matrix to the n$^{th}$ power is obtained with the matrix-matrix multiplication applied n times.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{bmatrix}$$

$$c_{ij} = a_{i1} \wedge b_{1j} \oplus a_{i2} \wedge b_{2j} \oplus \cdots \oplus a_{in} \wedge b_{nj} = \sum_{k=1}^{n} a_{ik} \wedge b_{kj}$$

This computation is done in VHDL, where the matrices are computed at compile time.