



PixESL at MAPS 2026 Hands-on

Francesco E. Brambilla
francesco.enrico.brambilla@cern.ch

SLAC, Stanford, USA
28-4 Jul-Aug 2026

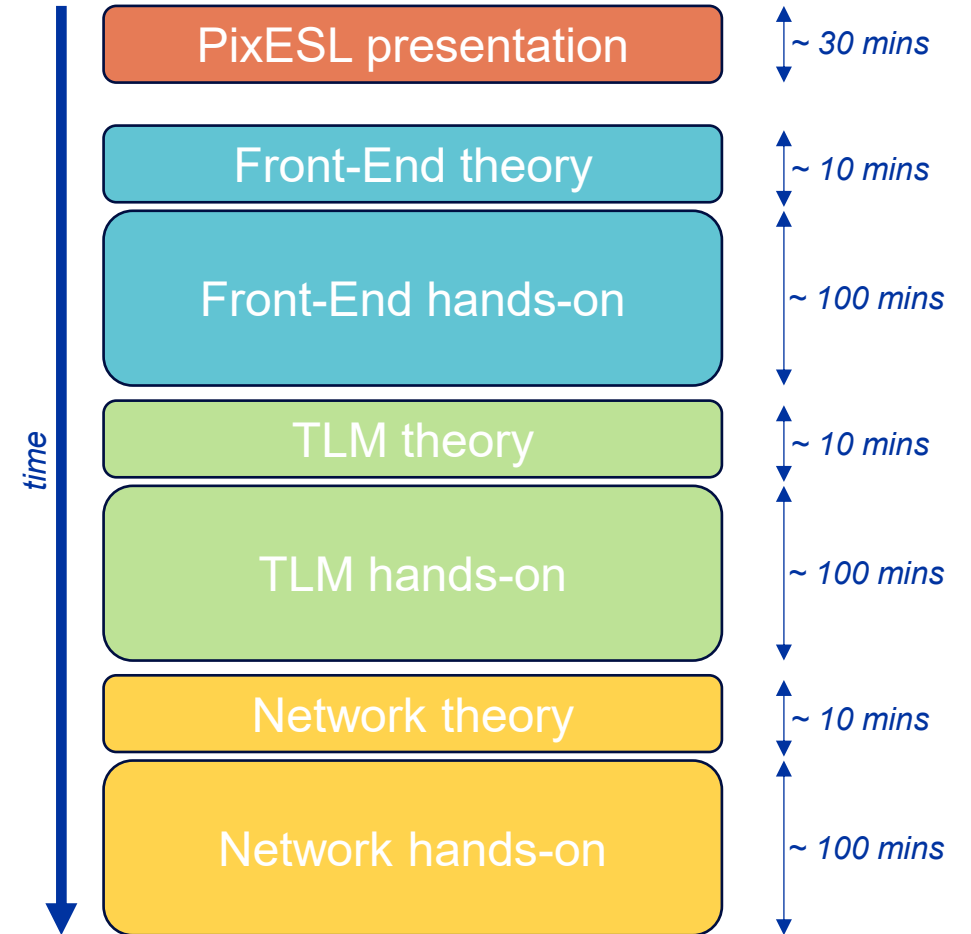
WP5 IC Tech
EP R&D

Overview

1. Goal and overall structure
2. Installation
3. Hands-on exercises

Goal and overall structure

- Understand how **detectors** are designed
- Learn how to **prototype ASIC architectures** for future experiments
- Gain experience in:
 - Modeling a **Front-End** from input charge to digitized packet
 - Data transfer with **TLM** for readout communication
 - Building a **full readout chip** and extract **metrics** for architectural studies



Installation

GitLab repository – [PixESL core](#) – [PixESL4school](#)

Follow the documentation in [pixesl-4-school/README.md](#)

1. Clone the PixESL4school

```
git clone ssh://git@gitlab.cern.ch:7999/iod/pixesl-group/pixesl-4-school.git  
or
```

```
git clone https://gitlab.cern.ch/iod/pixesl-group/pixesl-4-school.git
```

2. Navigate to the project root

```
cd pixesl-4-school/
```

3. Open the repository with an IDE (e.g. VScode)

```
code .
```

4. Install all required dependencies

```
make all -C install/setup/
```

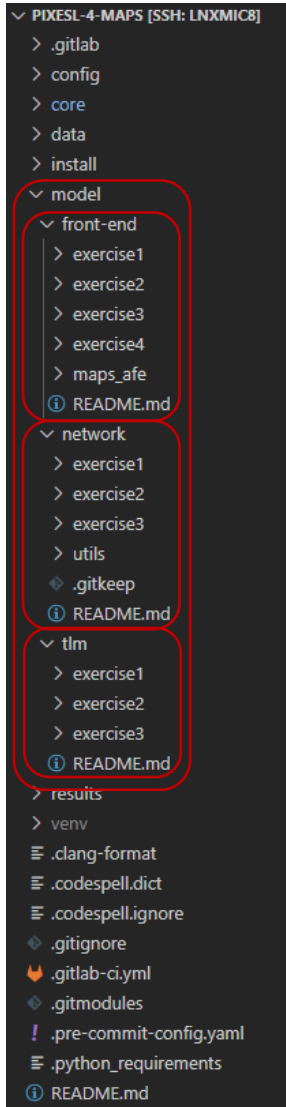
5. Use Bash shell

```
bash
```

6. Activate the virtual environment

```
source venv/bin/activate
```

Hands-on exercises



- The *model* directory contains hands-on exercises categorized into three sections:

- **Front-End**

- Exercise 1 - basics of AFE modelling
- Exercise 2 - AFE with triangular shape
- Exercise 3 - AFE from library with pile-up effect
- Exercise 4 - Front-End node

- **TLM**

- Exercise 1 - basics of TLM
- Exercise 2 - Front-End node with TLM - generate the "maps_node_front_end" package
- Exercise 3 - Front-End node and Readout node connection

- **Network**

- Exercise 1 - generate a Pixel Column
- Exercise 2 - generate a Pixel Matrix with a DataConcentrator at the periphery
- Exercise 3 - architectural exploration with Metrics Analyzer

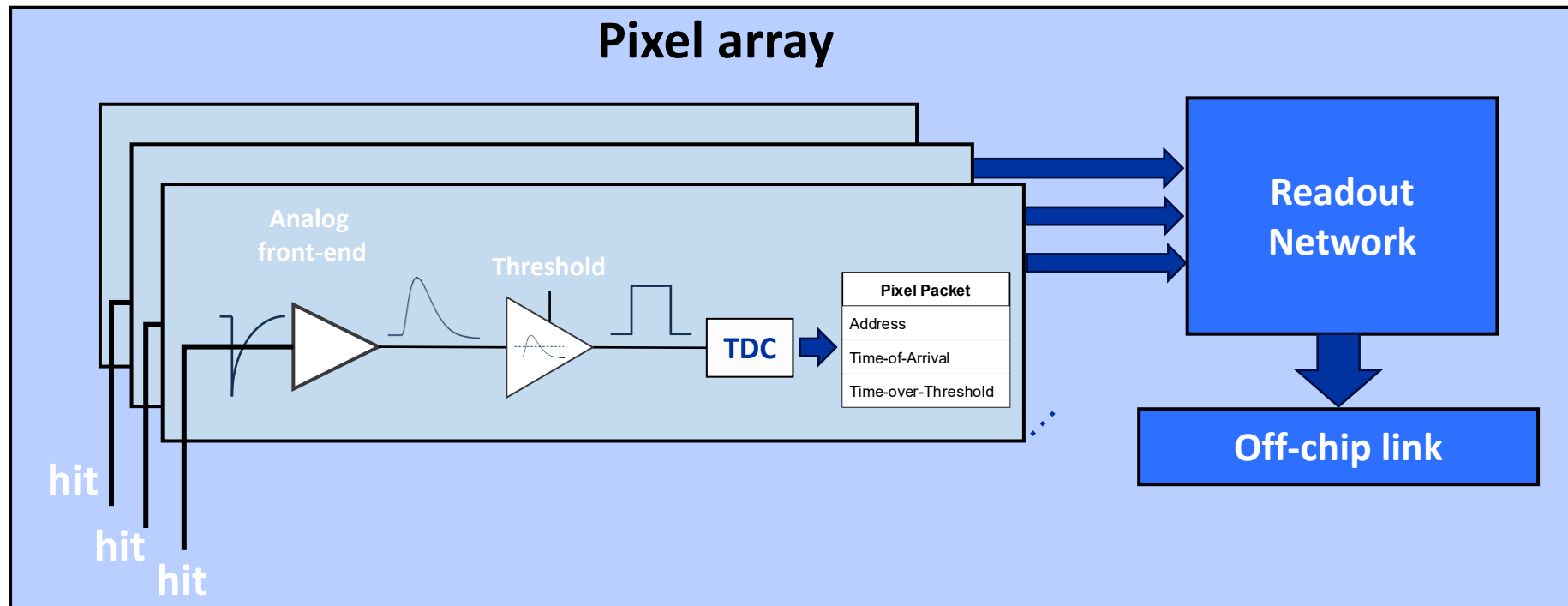
Use the **suggestions** of each exercise without shame

Front-End

1. Front-End
2. Input file
3. Analog Front-End core
4. Pile-up effect
5. AFE wrapper
6. TDC model
7. Output packet
8. Front-End node

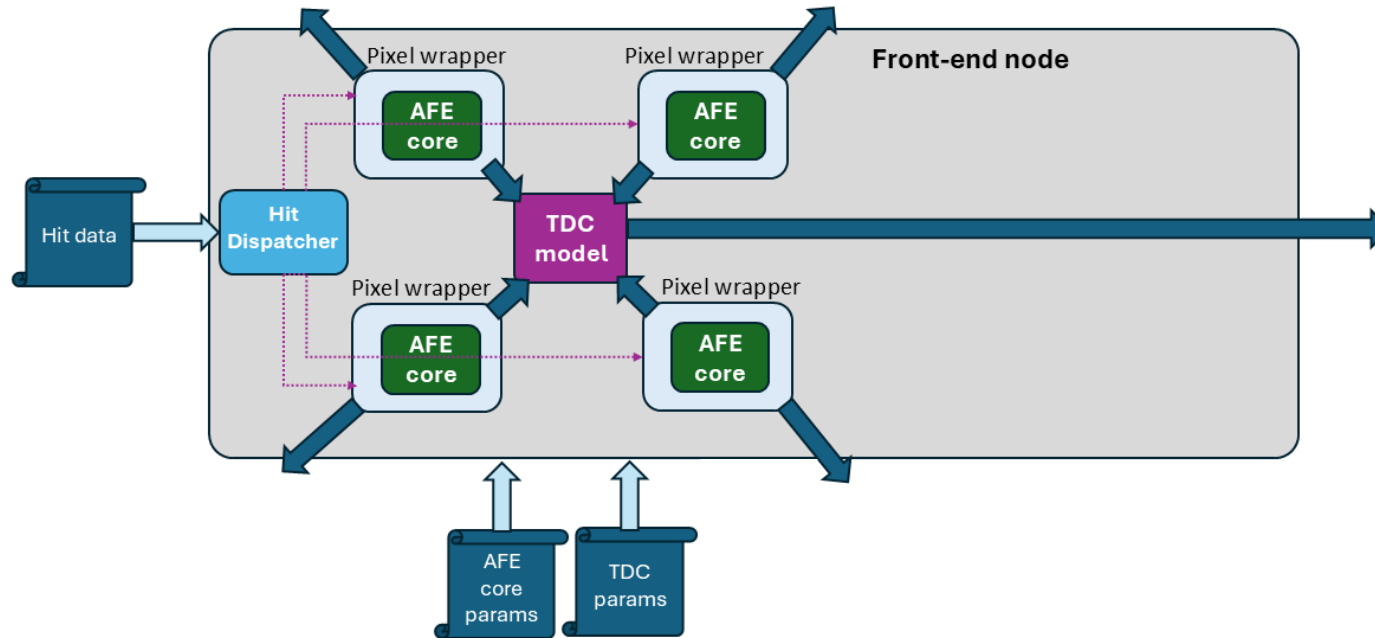
Front-End

Circuits connected to the detector that **acquire**, **amplify**, **shape**, and **digitize** the initial signal for further processing and readout



Front-End

The **Front-end (AFE)** model encapsulates the pixel detector behaviour from the **preamplifier** to the **time-to-digital converter (TDC)**.



Input file

This file contains the **property** of each **particle hitting** the pixel detector.

```
# cycle, addr_x, addr_y, h_time, qin  
1,0,0,200,100  
1,0,0,300,200
```

- **cycle**: Clock cycle at which the hit occurs. This is typically an integer and defines the simulation or hardware clock tick when the event is injected.
- **addr_x**: X address of the hit (pixel column). This identifies the horizontal position of the pixel hit.
- **addr_y**: Y address of the hit (pixel row). This identifies the vertical position of the pixel hit.
- **h_time**: Hit time or timestamp of the hit in finer resolution than cycle. This can be used for sub-cycle timing precision or time-walk correction.
- **qin**: Input charge (in electrons), representing the strength of the signal at the pixel. This value drives the AFE's response (e.g., amplification, shaping, threshold comparison).

Analog Front-End Core

The **AFE core** models the response of the Front-End to a particle, and particularly the discriminator of the pixel detector.

Input:

Charge generated by particle on the sensor

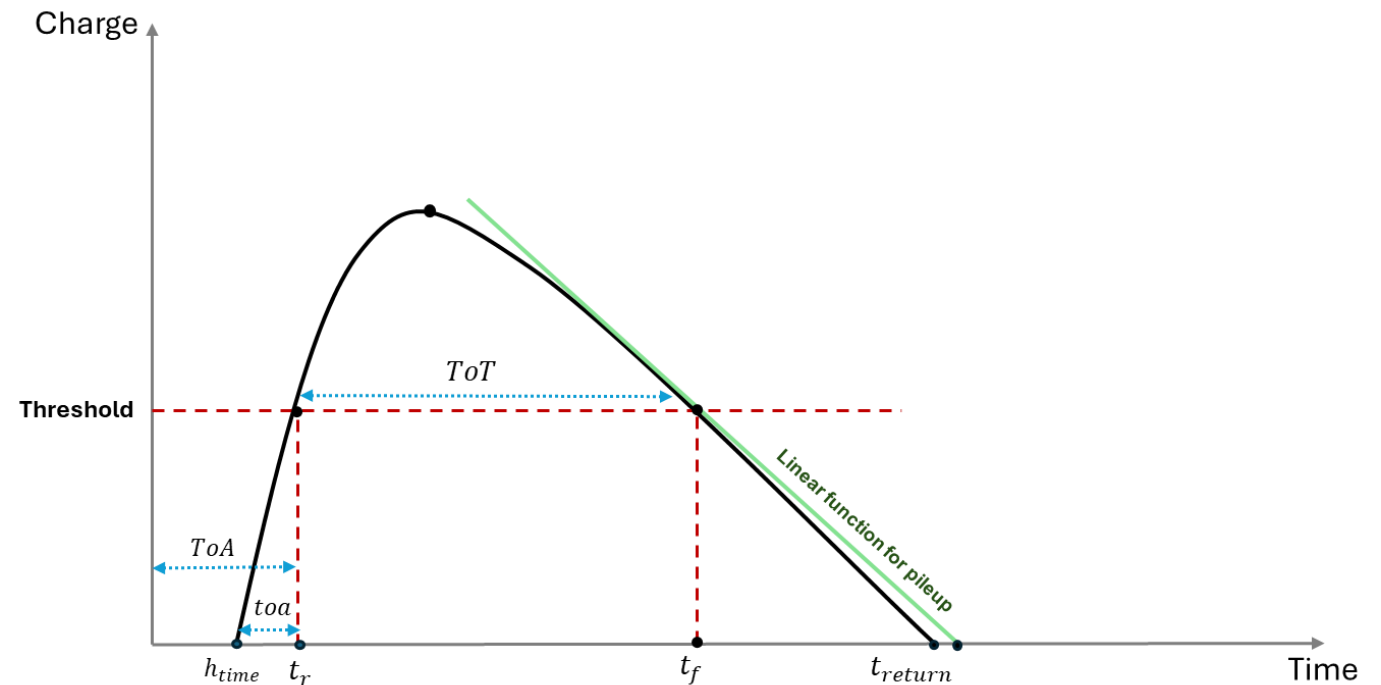
- **qin**: Incoming charge
- **h_time**: Time of pixel hit



Output:

Digital Asynchronous Pulse

- **tr**: Time of Rising edge
- **tf**: Time of Falling edge
- **ToT**: Time over Threshold



**AFE
core**

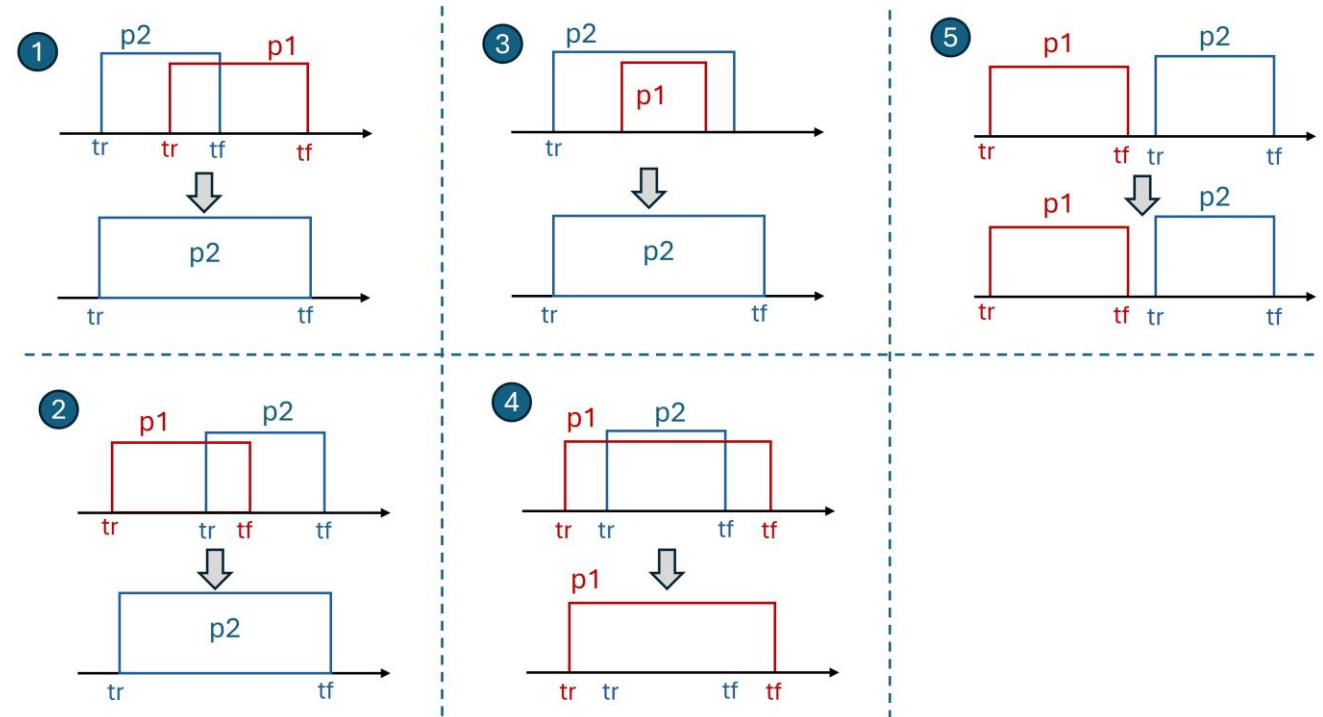
C++ only:
**Does not consume
simulation time**

Pile-up effect

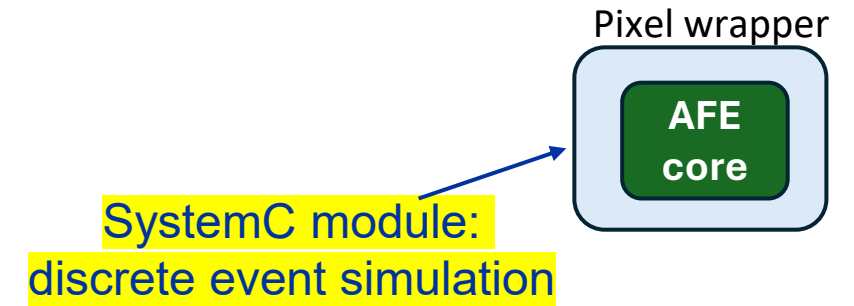
Pile-up refers to the **overlapping of signals** from multiple hits arriving **close in time** such that the detector cannot resolve them as distinct events.

Pile-up generates:

- Aliasing of nearby hits
- Distorted timing
- Altered ToT



AFE Wrapper

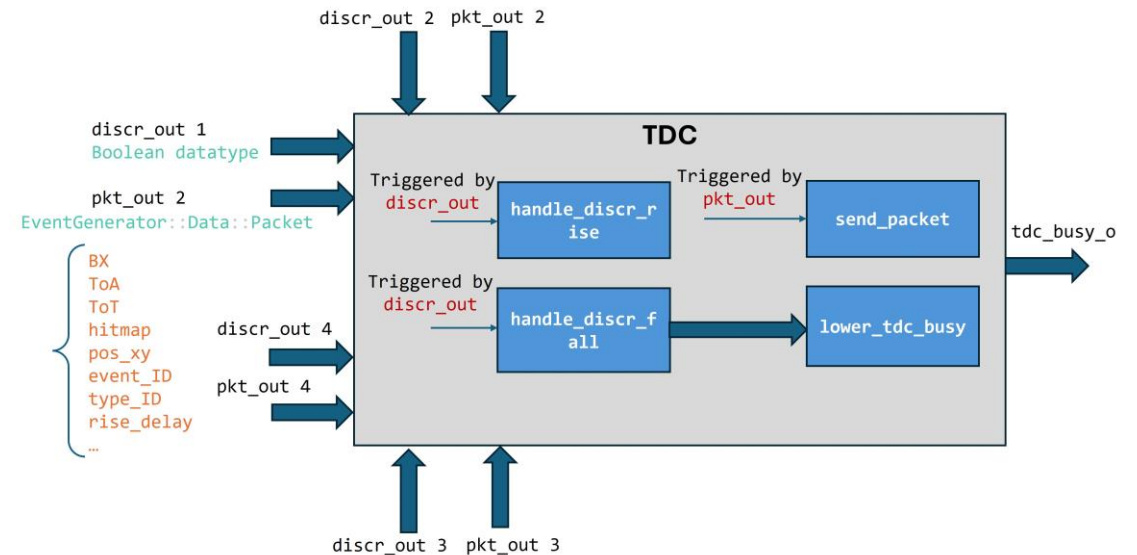
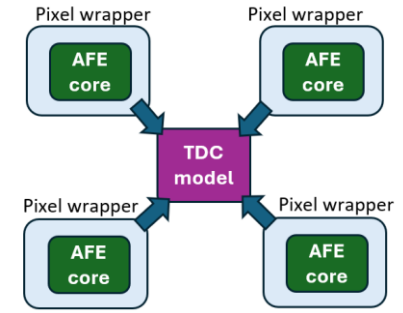


In PixESL, the **AFE wrapper** translates the AFE outputs into **SystemC events**:

- Forwards **input hit data** (e.g H_{time} and Q_{in}) to the core processing stage.
- Receives one $\{t_{\text{rise}}, t_{\text{fall}}\}$ pair for each injected pixel hit.
- **Filters** out low-charge signals and **merges** overlapping events.
- **Translates** AFE output t_{rise} and t_{fall} into the SystemC simulation:
 - **Manages** timing events by pushing **rise** and **fall times** into event queues.
 - **Triggers event creation** and updates signal outputs based on timing.
 - **Drives** the **AFE discriminator output** (digital asynch. pulse) and its rise/fall events,
 - Connects to the next module (**TDC**) in the detector hierarchy.

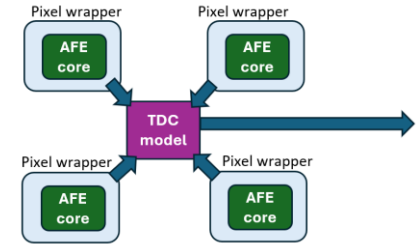
TDC model

- The **Time-to-Digital Converter (TDC)** is an electronic circuit that converts a digital pulse's length into a digital value.
- In **PixESL**, it receives packets and event information provided by lower-level C++ modules (AFE cores).
- Depending on the pixel detector design, multiple pixels may **share a single TDC**
- In a shared TDC, the **first pixel** whose **discriminator** signal **rises** gains control and can **send its packet** along with a **hitmap** of the others



TDC Output packet

```
BX, COL, ROW, TOA, TOT, HITMAP, LOG_TIME, LOG_COL, LOG_ROW
2, 1, 1, 2512, 11267, 000000000000000011000001, 63779, 0, 0
```



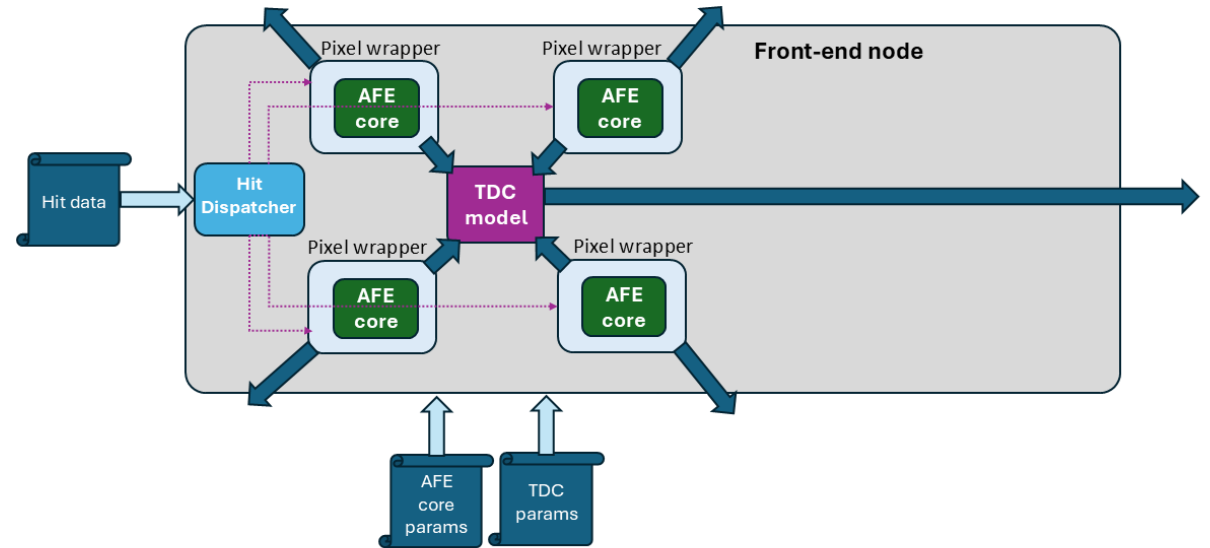
This packet format is ready to be transferred from the Front-End to the readout network:

- **BX**: Bunch crossing number; indicates the time slot or cycle in which the event occurred.
- **COL**: Column index of the pixel that registered the hit.
- **ROW**: Row index of the pixel that registered the hit.
- **TOA**: Time-of-Arrival; the timestamp (in ps) when the hit was detected.
- **TOT**: Time-over-Threshold; the duration (in ps) that the signal stayed above threshold, related to the charge deposited.
- **HITMAP**: Binary map indicating which pixels in a region registered hits during this event; each bit represents a pixel



Front-End node

- The **front-end node** in PixESL is a specialized component **derived from the *node_base* class**
 - Nodes are the basic elements forming the pixel matrix and readout network.
- It is designed to **receive hit data** and **generate packets**
- It includes:
 - AFE core
 - AFE wrapper
 - TDC



Front-End node parameters

- The readout node parameters are defined in a YAML file
- You can define:
 - Size
 - Clock period
 - Number of transactions in input
 - Number of transactions in output
 - FE shape
 - TDC parameters
 - AFE parameters
- Look at *model/front-end/exercise4/config/school_node_front_end.yaml*

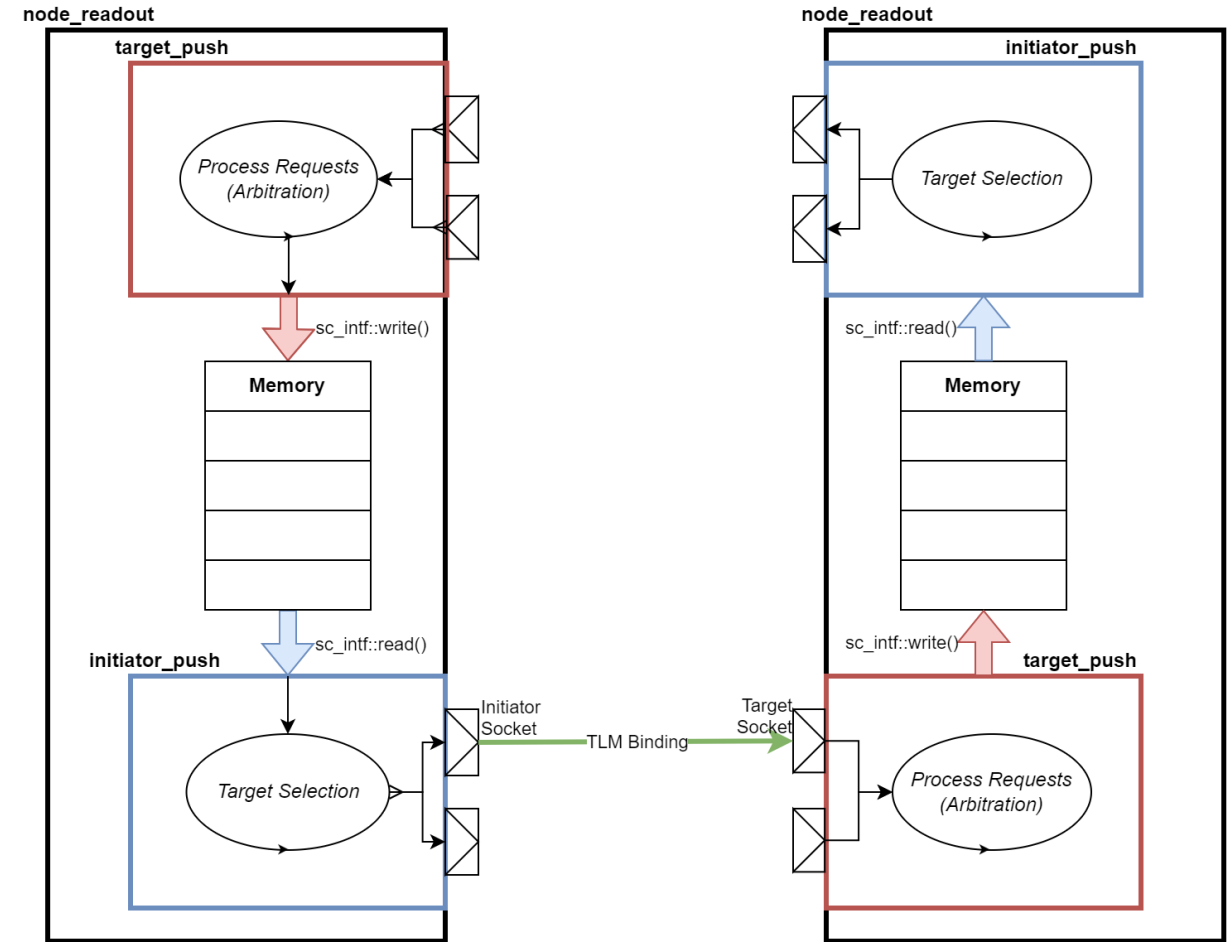
```
node_front_end:
  size: [1, 1]
  node_pars:
    clock_period: 25000 # in ps
    max_transactions_in: 1
    max_transactions_out: 1
    FE_shape: [2, 2]
    tdc_pars:
      BX_rate: 25000 # in ps
      TDC_CALIBRATION_PS: 0
      TDC_MIN_FALLRISE_DELAY_PS: 0
      discr_in_host_size: 4
      discr_in_neighbor_size: 32
    afe_pars:
      BX_rate: 25000 # in ps
      enable: 1
      vals_fit_rise: [0, 0.143076952466058, 0.26932441401332, 0.38045595908508,
                    0.478014893864497, 0.563391090519082, 0.63783617336955,
                    0.702477192643168, 0.758328951331277, 0.80630511296818,
                    0.847228211536724, 0.881838671987408, 0.910802939138167,
                    0.934720803061467, 0.954132000359441, 0.969522162882072,
                    0.981328178372991, 0.989943021155625, 0.995720105230198,
                    0.998977206977272, 1]
      vals_fit_fall: [0.808767092652401, 0.821754667436491, 0.83471482896308, 0.847616738795438,
                    0.860425643521304, 0.873102435264406, 0.885603163812446, 0.897878495052794,
                    0.909873109825037, 0.92152503665353, 0.932764911106293, 0.943515153731232,
                    0.95368905763802, 0.963189775814601, 0.971909197180541, 0.979726699173489,
                    0.986507763326871, 0.992102438812094, 0.996343637270821, 0.999045240434559, 1]
      thl_mean: 2000.0
      thl_sigma: 100.0
      tot_sigma: 10.0
      noise_sigma: 100.0
      sys_noise_thl: 10e5
      sys_noise_val: 500
      peaking_time: 3200
      preamp_dis: 3.5
      tot_mismatch: 0
      sys_noise: 0
      tot_rnd: false
      thl_rnd: false
      noise_rnd: false
      sys_noise_rnd: false
```

TLM

1. TLM definition
2. TLM ports
3. TLM transaction flow
4. Front-End node with TLM
5. Readout node
6. Readout node parameters
7. Front-End node and readout node connection

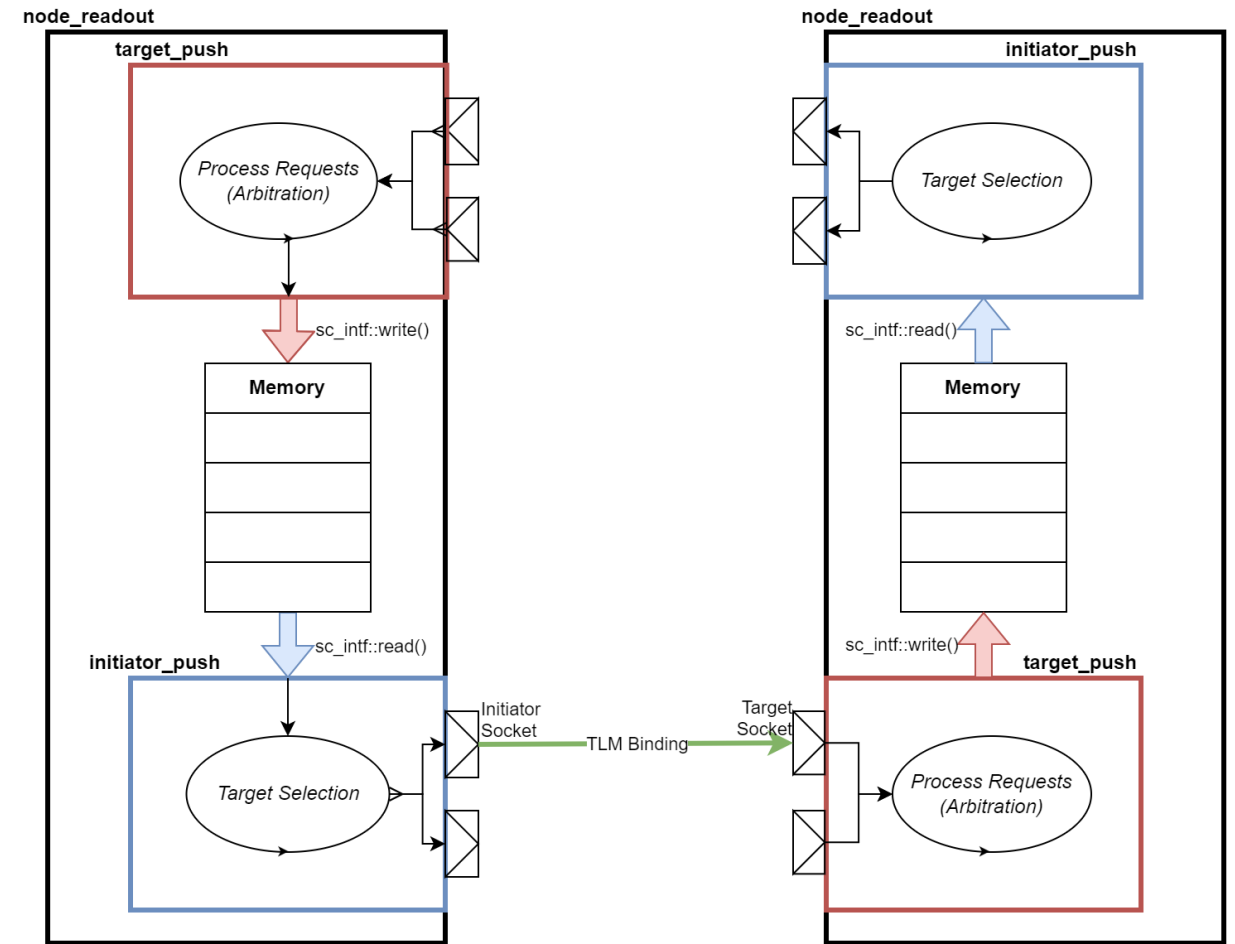
TLM definition

- Transaction-level modeling (TLM) is an approach to data communication modeling that **abstracts the transfer protocol** away from its implementation, **focusing on the functionality of the transfer** rather than the details of the protocol itself.
- PixESL implements TLM 2.0 library to create TLM ports:
 - Event-driven
 - Push-based protocol
 - Multi-passthrough sockets



TLM ports

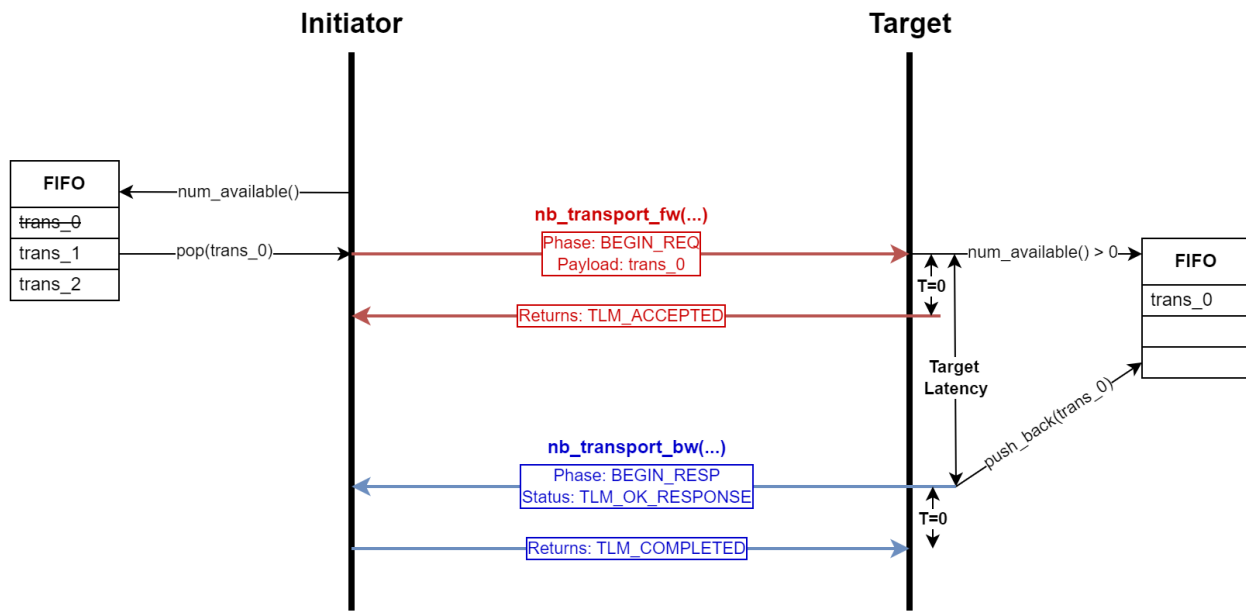
- TLM ports are modules made up by TLM sockets
- TLM ports can be either TLM initiators or TLM targets
- TLM ports need to be bound to a memory element (FIFO)



TLM transaction flow

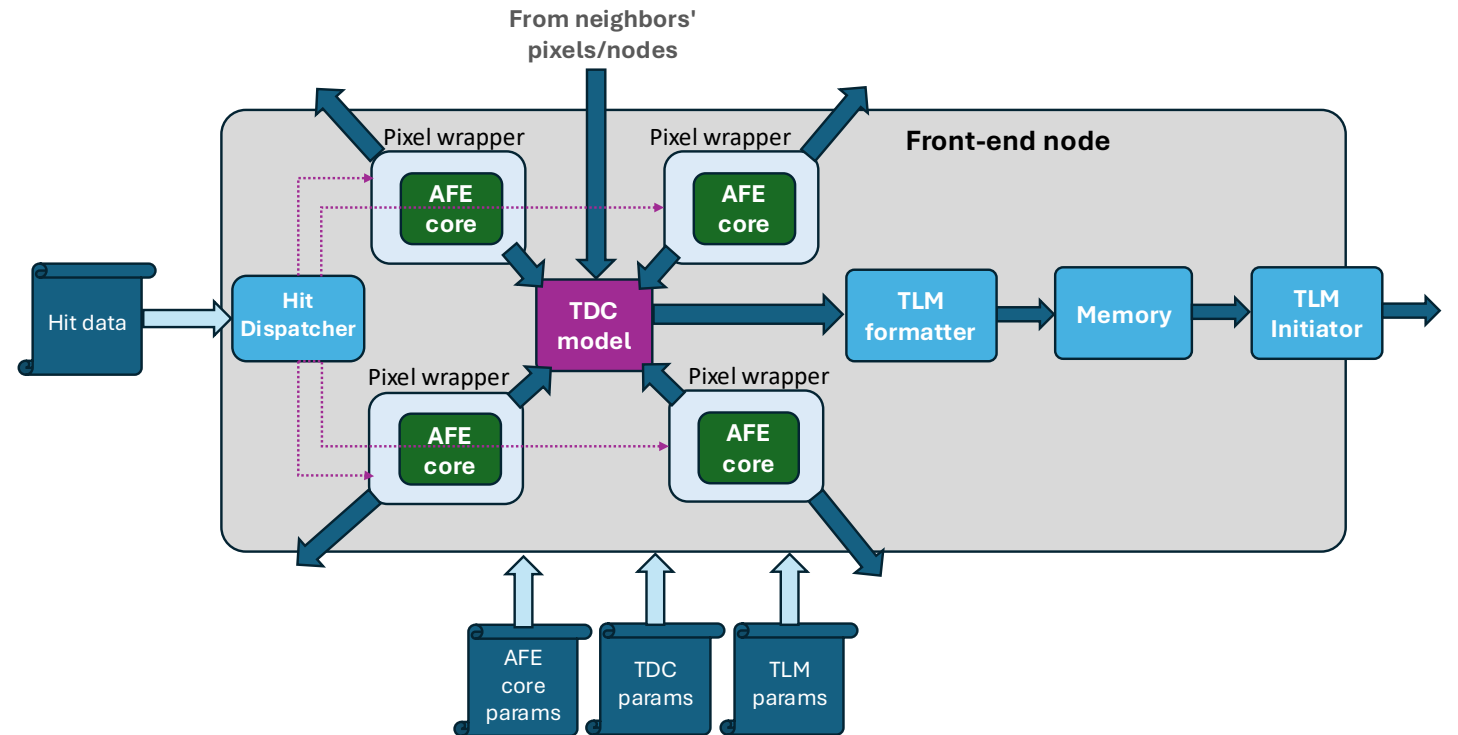
Transaction phases:

1. The *Initiator* waits until the memory notifies a write event, which means that a new packet is available
2. The *Initiator* packages a new transaction with the packet from memory, and selects one of the *Targets* bound to the socket.
3. The *Initiator* sends the transaction to the chosen *Target* (`nb_transport_fw(...)`), which could immediately refuse the incoming transaction, for example in case the *Target's* memory is full.
4. This is repeated until all packets, *Targets* are exhausted, or the maximum amount of pending transactions in the *Target* is reached.
5. On the *Target* side, all accepted transactions are sorted in a queue based on their priority token, generated by the target based on its arbitration algorithm.
6. After a certain *Target Latency*, all the transaction in the priority queue are ready to be resolved:
7. The first transactions in the queue, up to the maximum number of transactions per cycle, are accepted by calling `nb_transport_bw(...)` with status `TLM_OK_RESPONSE`, and their packet written to the *Target's* memory.
8. The other transactions are refused and written back in the original memory elements by the *Initiators*.



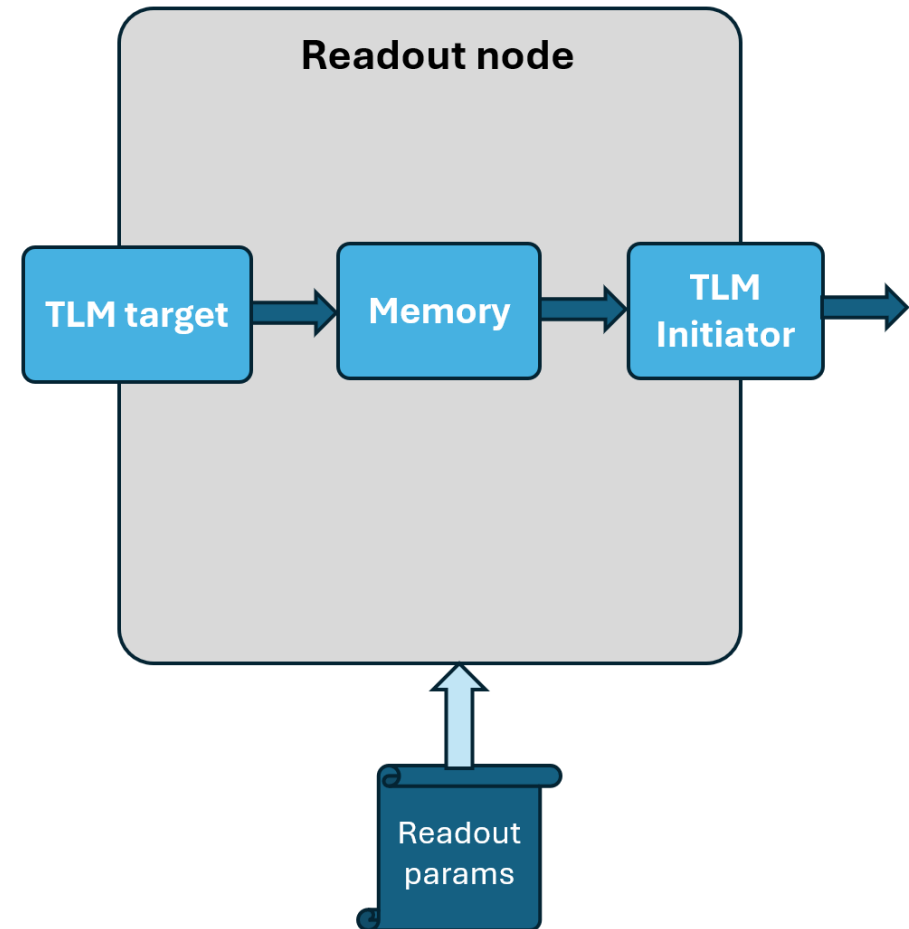
Front-End node with TLM

- TLM formatter receives the packets from TDC, converts them to TLM payload, and writes them into memory
- Memory element to store data
- TLM initiator monitors the memory; as a new data written into the memory, it will take it and push it into the network.



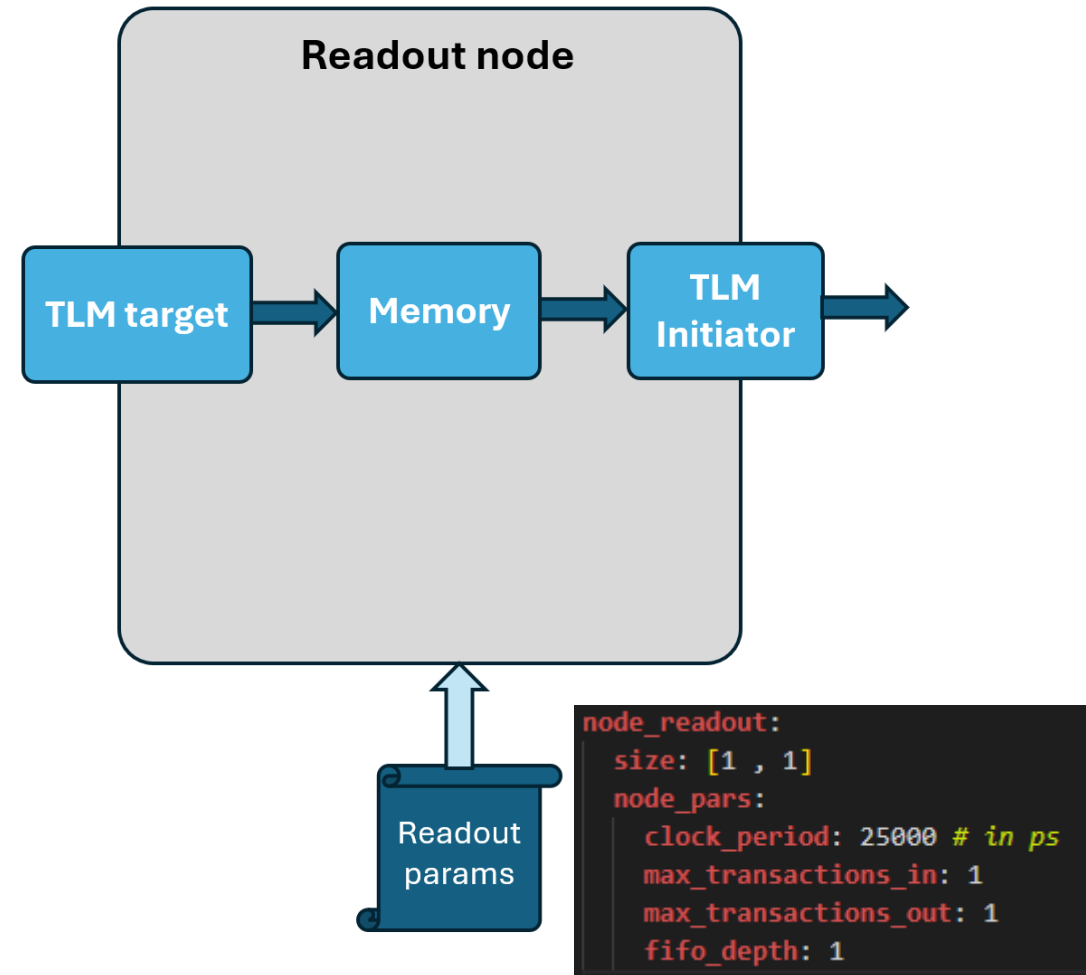
Readout node

- The **readout node** in PixESL is a specialized component **derived** from the *node_base* class
- It represents a **node** in the **network**
- It **read** from its **memory** and **transmits** to the other nodes using **TLM-based communication**
- It includes:
 - TLM target
 - Memory
 - TLM initiator



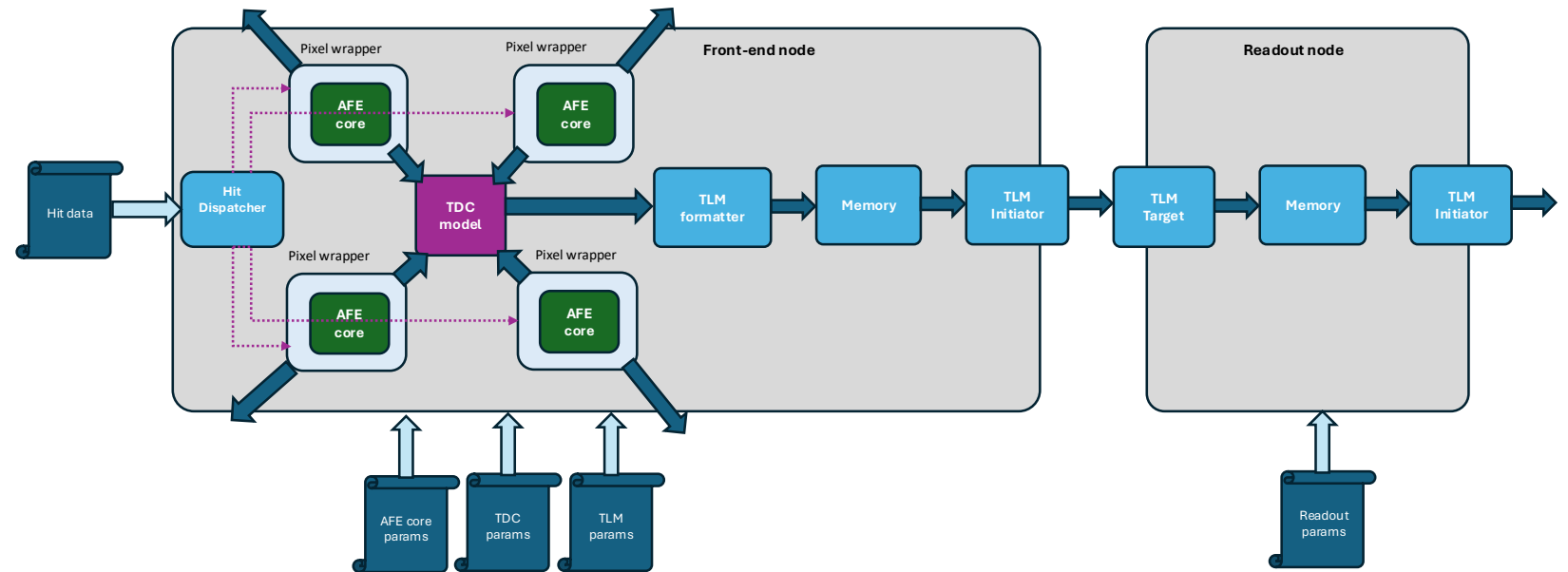
Readout node parameters

- The readout node parameters are defined in a YAML file
- You can define:
 - Size
 - Clock period
 - Number of transactions in input
 - Number of transactions in output
 - FIFO depth
- Look at *model/tlm/exercise3/config/node_readout.yaml*



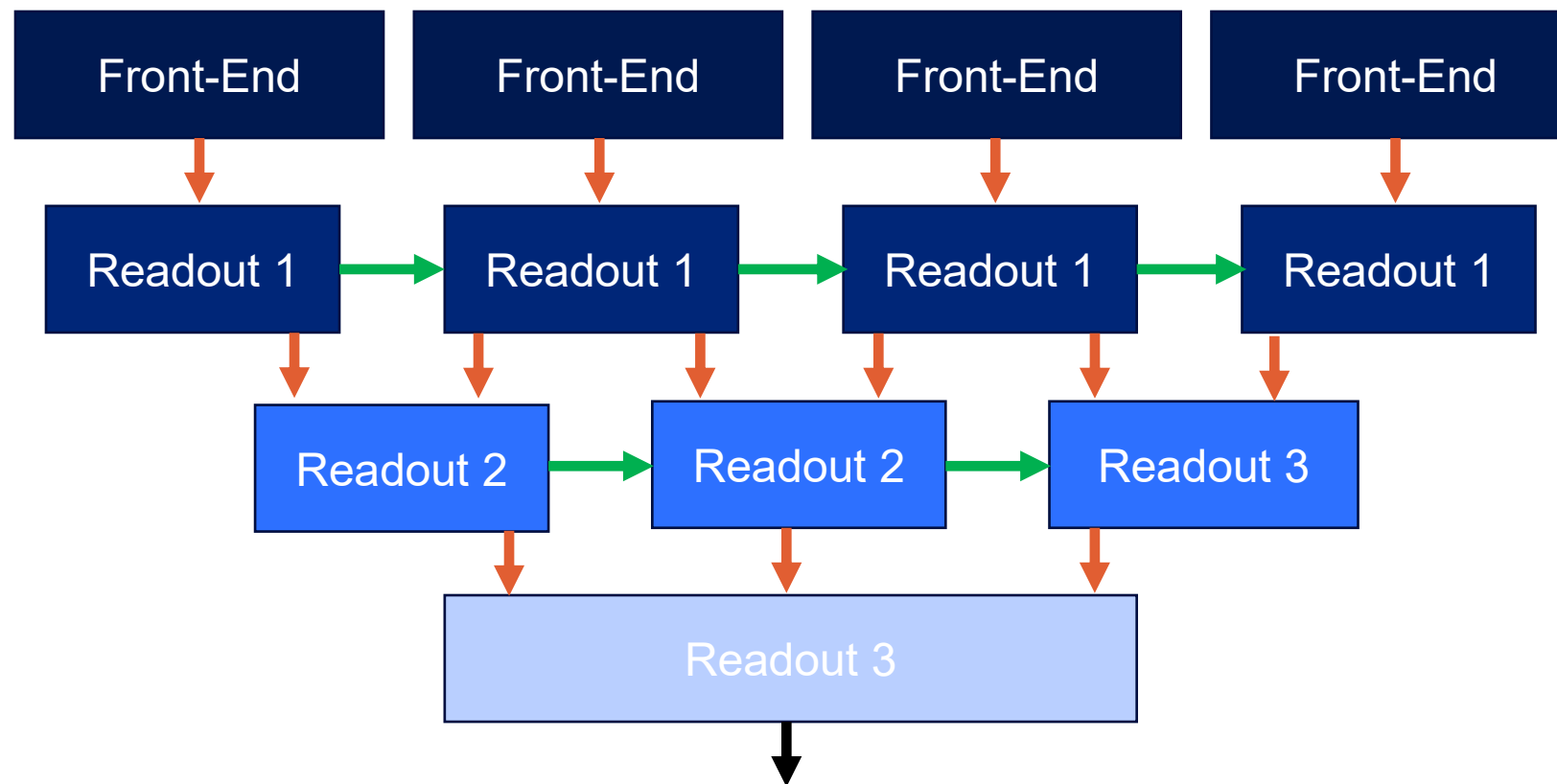
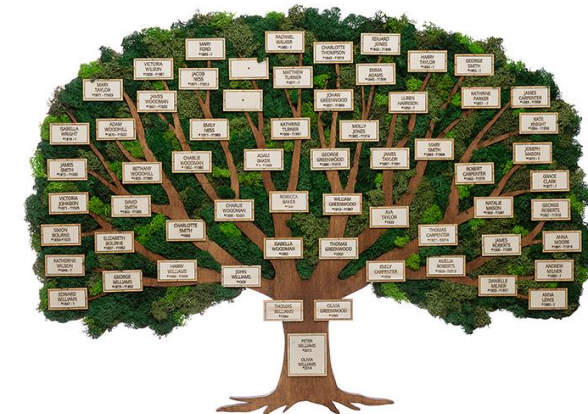
Front-End node and readout node connection

- Connect the TLM initiator of the Front-End node to the TLM target of the readout node
- `my_school_node_front_end.m_init.sockets(Direction::PARENTS).bind(my_node_readout.m_targ.sockets(Direction::CHILDREN));`
- Exercise [model/tlm/exercise3/](#)



Directions of TLM connections

- The **TLM connection** between an initiator and a target can be either:
 - CHILDREN → PARENTS**
 - SIBLING → SIBLING**

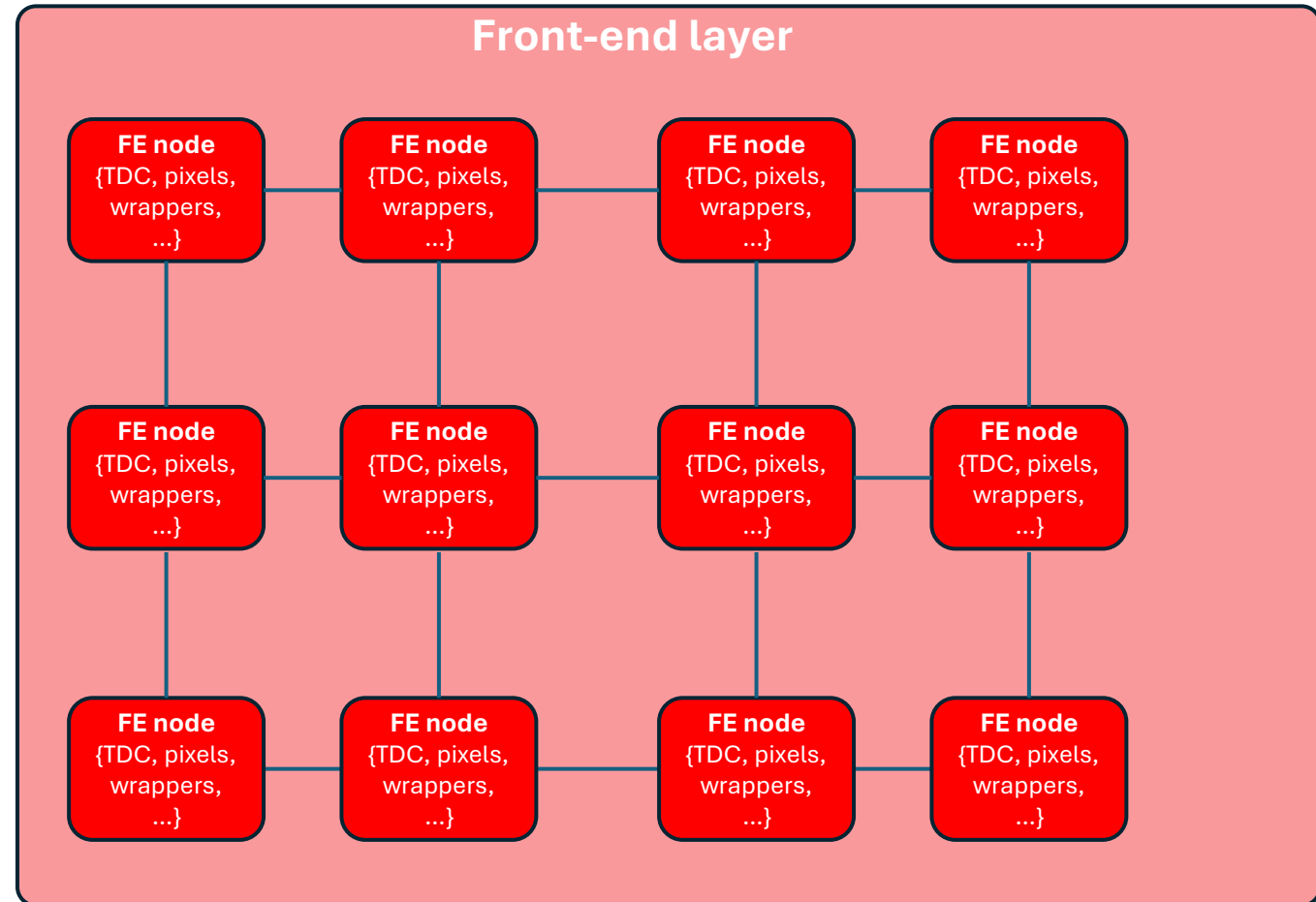


Network

1. Layers
2. Network
3. Configuration files
4. Connections
5. Model generation
6. Simulation

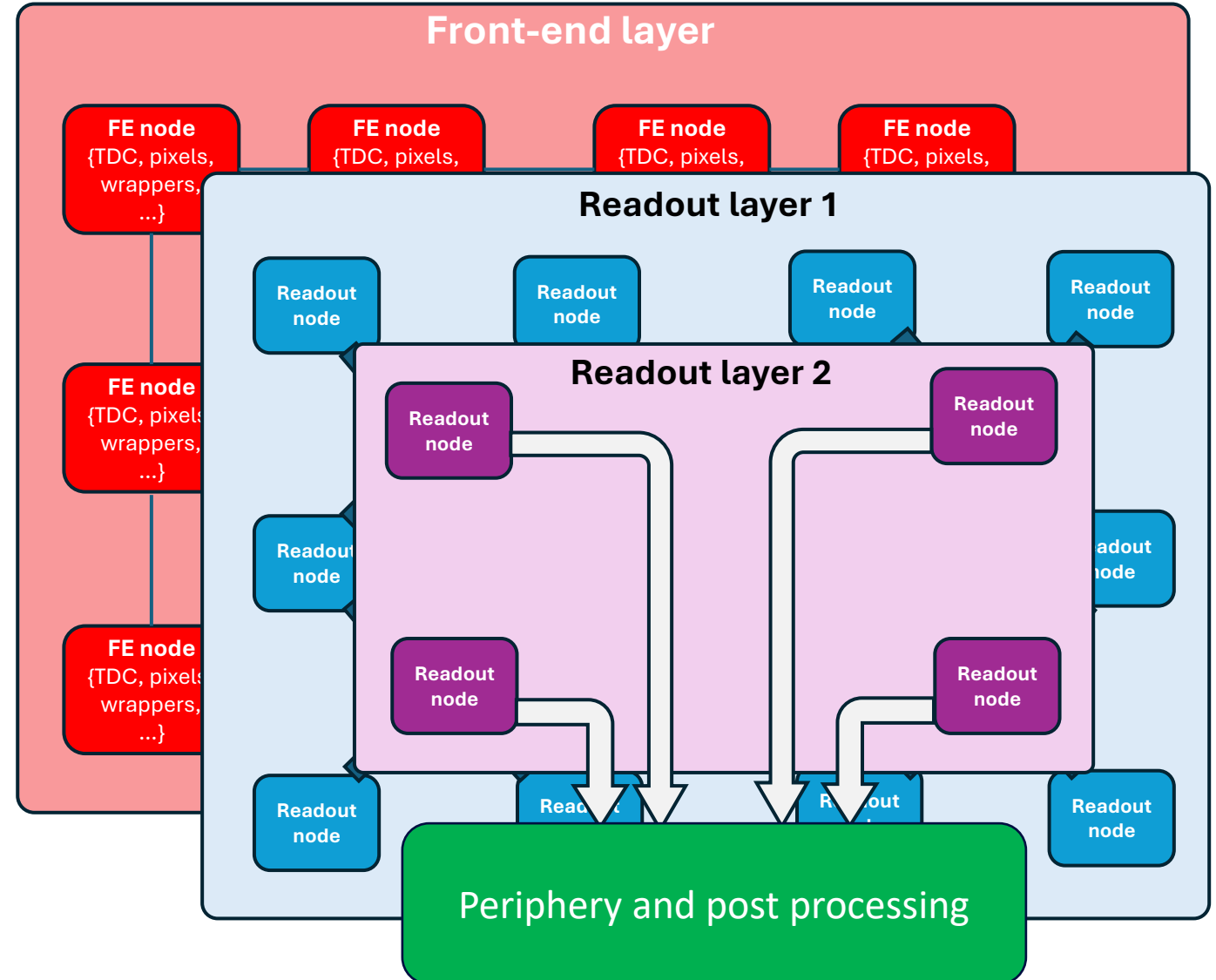
Layers

- Heterogenous nodes are grouped into **layers** to manage:
 - **Instances**: size of the node array.
 - **Parameters**: All nodes in a layer share the same values.
 - **Connections**: how each node is connected to other nodes in the same layer or an external one.
- Different classes of layers contain different nodes.
 - Some nodes require special methods to connect them.
- Layer and node type in a network can be configured through files → no need to look into layers for this tutorial.



Network

- A pixel detector consists of:
 - Front-End nodes
 - Readout network
 - Off-chip data links
- Each set of nodes is arranged in a square grid, represented in a layer.
- **PixESL** contains templates and basic models of these components.
- The user can simply re-use, overwrite, and contribute to the library.



Configuration files

- PixESL defines and runs a simulation through configuration files:
 - Layer parameters:
 - `node_frontend.yaml`
 - `node_readout.yaml`
 - `connection.yaml`:
 - defines the hierarchical and geometric connections between layers in a network.
 - `network.yaml`:
 - defines the structure and connections of layers in a network.
 - `simulation.yaml`:
 - configures simulation parameters, testbench settings, stimuli input, and monitoring sinks for data flow and occupancy tracking.

```
▼ config
  ▼ data
    ■ stimuli.csv
  ▼ layer_pars
    ! EndOfColumn.yaml
    ! FrontEnd.yaml
    ! SuperPixels.yaml
    ! connections.yaml
    ! network.yaml
    ! simulation.yaml
```

Connections

- The **connections** are defined in a **YAML file**.
- For each connection block, you can define:
 - **Layer source**: layer where the data is sent from.
 - **Layer target**: layer where the data is received in.
 - **Connection type**:
 - **SC**: used for connections with SystemC ports.
 - **TLM**: used for connections with TLM ports.
 - **Block size**: x-y size of the repeating pattern.
 - **Link mode**: how to connect each node to its target.
 - **Relative**: target is defined relative to the source.
 - **Absolute**: target address is explicitly defined.
 - **Block**: 3D array to specify row-column-link.
- Look at *model/network/exerciseX/config/connections.yaml*

```
connection_blocks:
  # FrontEnd -> SuperPixels
  # Each FrontEnd node is connected to the SuperPixel right above it (hierarchically)
  - layer_source : FrontEnd
    layer_target : SuperPixels
    connection_type: TLM
    block_size : [1, 1]
    link_mode: "rel_s"
    block :
      - [[[a]]]

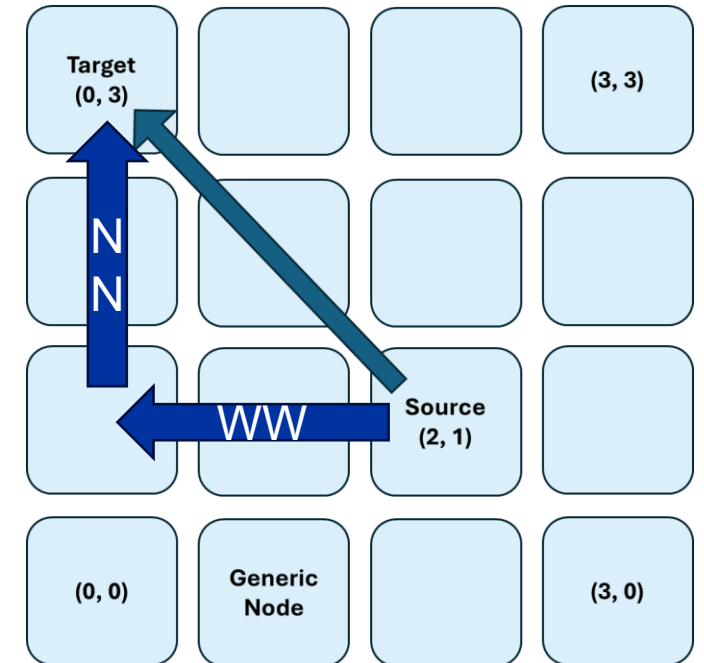
  # SuperPixels -> SuperPixels
  # Each SuperPixel node is connected to the SuperPixel right below (geometrically)
  # or the Region above it (hierarchically, connected in the next block)
  - layer_source : SuperPixels
    layer_target : SuperPixels
    connection_type: TLM
    block_size : [1, 1]
    link_mode: "rel_s"
    block :
      - [[[s]]]

  # SuperPixels -> EndOfColumn
  - layer_source : SuperPixels
    layer_target : EndOfColumn
    connection_type: TLM
    block_size : [1, 4]
    link_mode: "rel_s"
    block :
      - [[[a]]]
      - [[[x]]]
      - [[[x]]]
      - [[[x]]]

  # EndOfColumn -> Sink
  - layer_source : EndOfColumn
    layer_target : tb.sink
    connection_type: TLM
    block_size : [1, 1]
    link_mode: "abs_i"
    block :
      - [[["0 0"]]]
```

Connections – link mode

- Link mode specifies the mode of linking:
 - **rel_s** using a *string* pointing to an address relative to the source node
 - **n | s | e | w** : respectively point to the target addresses in directions north, south, east and west
 - **x**: used to leave a node unconnected
 - **a**: used to connect to the “above” node
 - Each direction can be used multiple times
 - **(nnww)** would point to a node 2 addresses above and 2 to the left
 - **rel_i** for relative connections using an integer address;
 - **(-2 2)** would point to a target node 2 addresses above and 2 to the left
 - **abs_i** for absolute connections using an integer address;
 - **“(0 3)”** connects to node with the address corresponding to that value.



Model generation

- The DUT is generated through a template DUT `network/lib/gen_network/templates/network_base.hpp.jinja`
- Look at `model/network/exerciseX/config/network.yaml`

network.yaml

```
network:
  name: pixel_column
  params_layer_dir: layer_pars/
  layers:
    - name: FrontEnd
      layer_type: layer_front_end
      layer_source: inc/impl/layer/layer_front_end.hpp
      node_type: maps_node_front_end
      node_source: inc/maps_node_front_end.hpp
      connection_list:
        - target_name: SuperPixels
          target_node: node_readout
    - name: SuperPixels
      layer_type: layer_readout
      layer_source: inc/impl/layer/layer_readout.hpp
      node_type: node_readout
      node_source: inc/impl/node/node_readout.hpp
      connection_list:
        - target_name: SuperPixels
          target_node: node_readout
        - target_name: EndOfColumn
          target_node: node_readout
    - name: EndOfColumn
      layer_type: layer_readout
      layer_source: inc/impl/layer/layer_readout.hpp
      node_type: node_readout
      node_source: inc/impl/node/node_readout.hpp
```

DUT generator

dut.hpp

```
// This code is generated, do not edit by hand.
//
// Sprague 2015
#include <system.h>

#include <inc/impl/layer/layer_front_end.hpp>
#include <inc/impl/layer/layer_readout.hpp>
#include <inc/impl/node/node_readout.hpp>
#include <inc/maps_node_front_end.hpp>

namespace PixESL::Network {
  using namespace sc_core;
  using namespace PixESL::700;

  class pixel_column : sc_module {
  public:
    pixel_column(sc_module_name name)
      : pixel_column() = default;

    void binding() {
      // Layers
      layer_front_endmaps_node_front_end FrontEnd;
      layer_readoutnode_readout SuperPixels;
      layer_readoutnode_readout EndOfColumn;

      // Configuration and parameters methods.
      void only_params(const std::string &par_dir);
      void connect_layers(const std::string &cfg_file);

      template <typename sink_t>
      void bind_sink(sink_t *sink);

      void before_and_after_elaboration() override {
        template <typename hit_t>
        void inject(hit_t *hit, const std::string &input_str);

        // Configuration file
        std::string connect_cfg_file;
      };

      // Constructor
      pixel_column(pixel_column(sc_module_name name) : FrontEnd("FrontEnd"), SuperPixels("SuperPixels"), EndOfColumn("EndOfColumn")) {
        // Parse parameters for layers and generate elems
        void pixel_column::only_params(const std::string &par_dir) {
          // Connect layers, argument is the connections.yaml file
          void pixel_column::connect_layers(const std::string &cfg_file) {
            template <typename hit_t>
            void pixel_column::inject(hit_t *hit, const std::string &input_str) {
              template <typename sink_t>
              void pixel_column::bind_sink(sink_t *sink) {
            } // namespace PixESL::Network
          }
        }
      }
    }
  };
}
```

Simulation

- The simulation parameters are defined in a **YAML** file
- You can define:
 - Clock period
 - Simulation time
 - Stimuli file
 - Latency threshold (for metrics)
 - Logging files for monitoring (for metrics)
 - Enabling occupancy logging (for metrics)
- Look at *model/network/exerciseX/config/simulation.yaml*

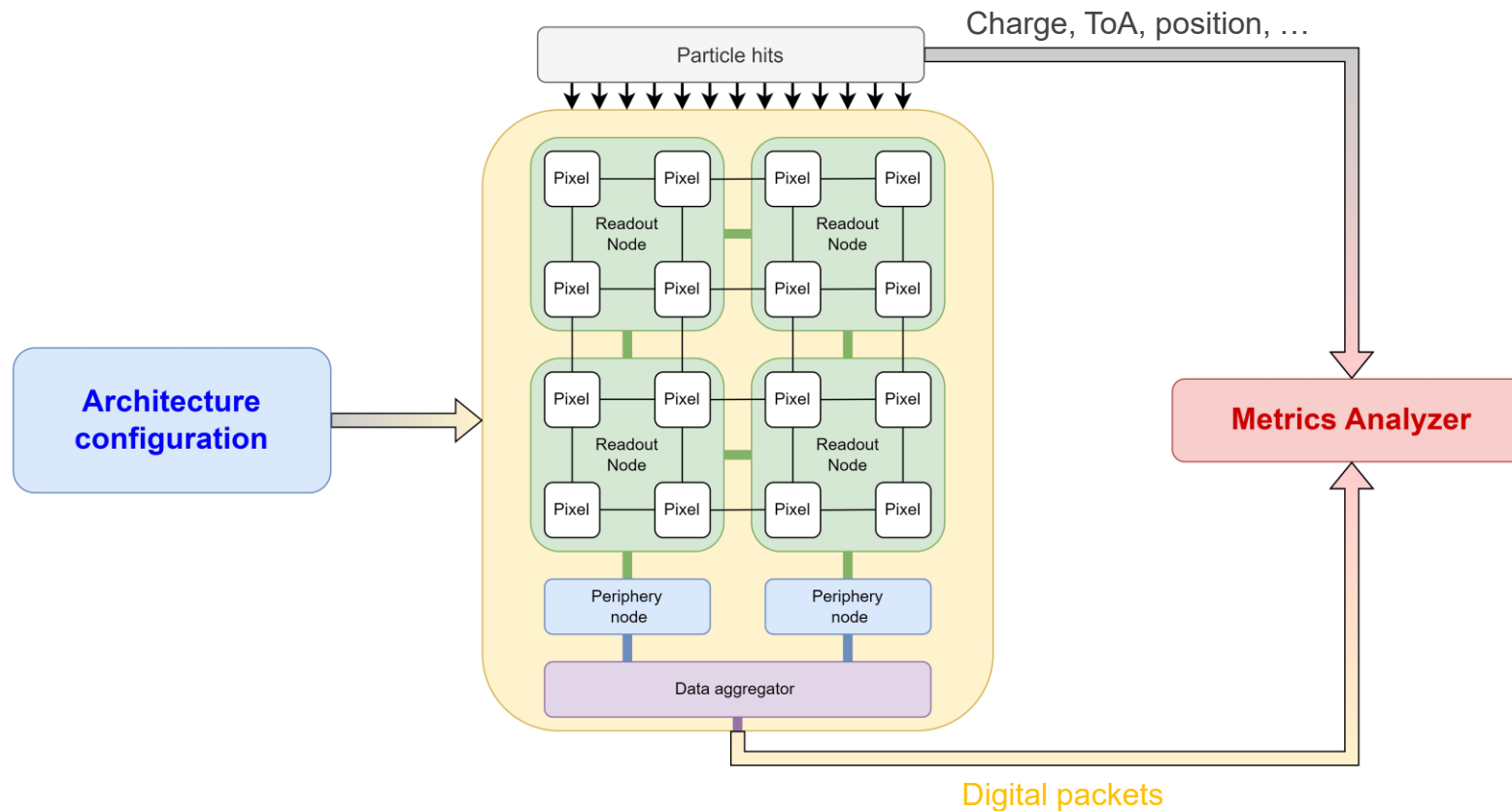
```
Simulation:
  base_testbench_pars:
    clock_period: 25000
    simulation_time: 1000000
    stimuli_file: ../config/data/stimuli.csv
    latency_threshold: 20 # cycles
  monitor_sinks:
    - file_name : packets_SuperPixels_IN_from_children.csv
      nodes : SuperPixels
      direction: CHILDREN
      io_type: INPUT
    - file_name : packets_output_EndOfColumn.csv
      nodes : EndOfColumn
      direction: CHILDREN
      io_type: INPUT
  sink_enable_global: false
  sink_enable_occupancy: true
```

Metrics

1. Metrics in PixESL
2. Metrics generation flow
3. Metrics Analyzer

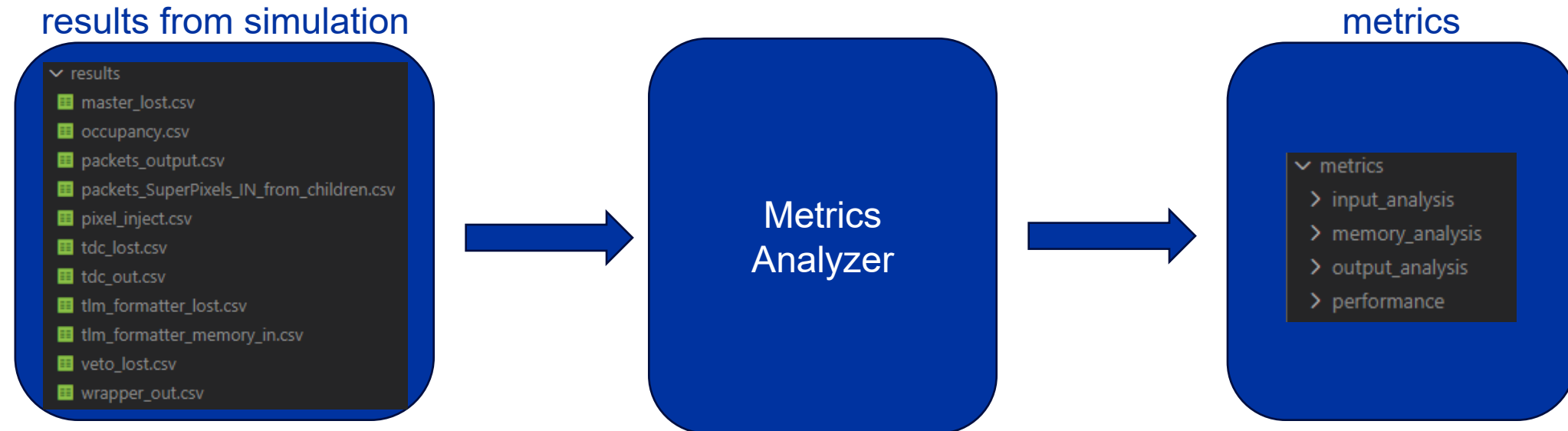
Metrics in PixESL

PixESL includes a Python-based **Metrics Analyzer** to examine the metrics information logged by the SystemC model



Metrics generation flow

PixESL includes a Python-base [Metrics Analyzer](#) to examine the metrics information logged by the SystemC model



Metrics Analyzer

Metrics Analyzer generates metrics for:

- *Performance*
- *Input analysis*
- *Output analysis*
- *Memory occupancy*

Metrics Analyzer – performance

Metrics Analyzer generates metrics for:

- *Performance*
 - Logs performance metrics, efficiency, latency, and packet statistics
- *Input analysis*
- *Output analysis*
- *Memory occupancy*

```
### SUMMARY OF GROUPS (avg & std dev. of hits/packets per BX) ###

Group                Average                Std Deviation
-----
pixel_inject.csv     2.17                    2.32
master_lost.csv      0.52                    0.77
wrapper_out.csv      1.87                    2.00
veto_lost.csv        0.00                    0.00
tdc_lost.csv         1.00                    0.00
tdc_out.csv          1.33                    1.38
t1m_formatter_memory_in.csv 1.33                    1.38
t1m_formatter_lost.csv 0.00                    0.00
packets_output.csv   1.33                    1.38

#### Efficiency of end-to-end PixESL model ####

- efficiency:    61 / 61    (100.0%)

#### PERFORMANCE ####

Average latency (overall): 7 cy.
Under th. latency: 7 cy.
Over th. latency: 0 cy.
Max latency: 10 cy.

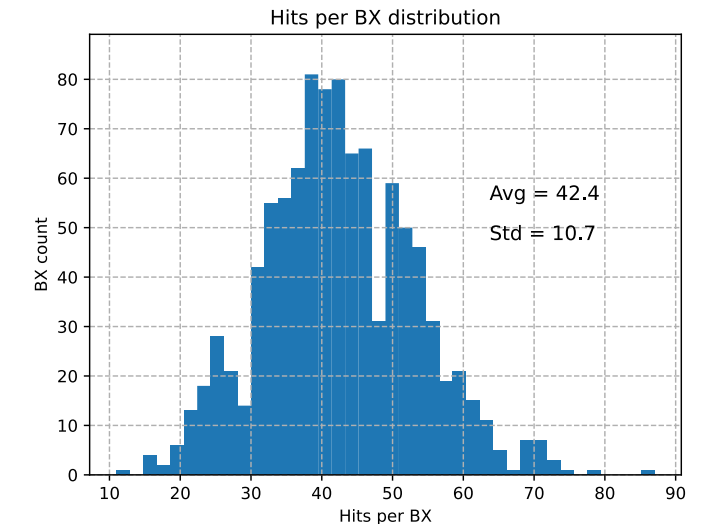
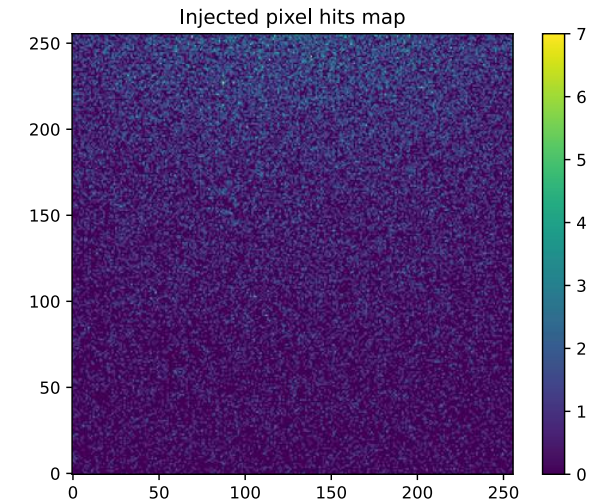
Readout efficiency (input_ro -> output_ro): 100.0 %
Under th. efficiency: 100.0 %
Over th. efficiency: 0.0 %

Missing packets (RO): 0
Lost in model packets (RO): 0
```

Metrics Analyzer – input analysis

Metrics Analyzer generates metrics for:

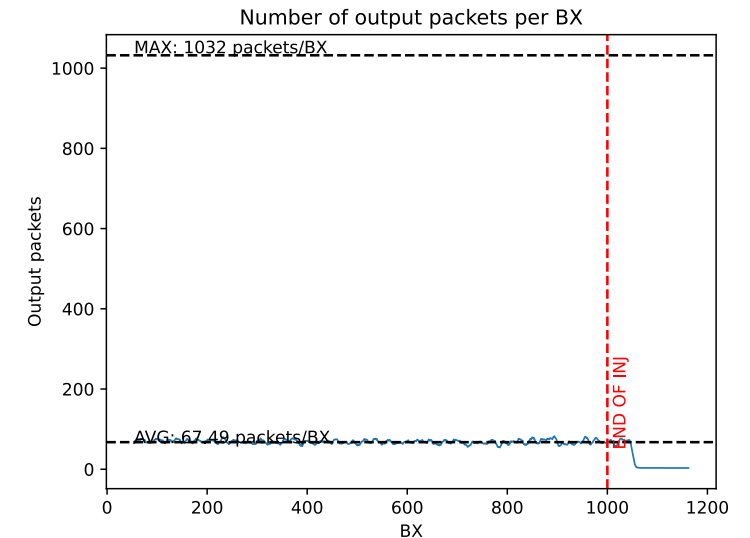
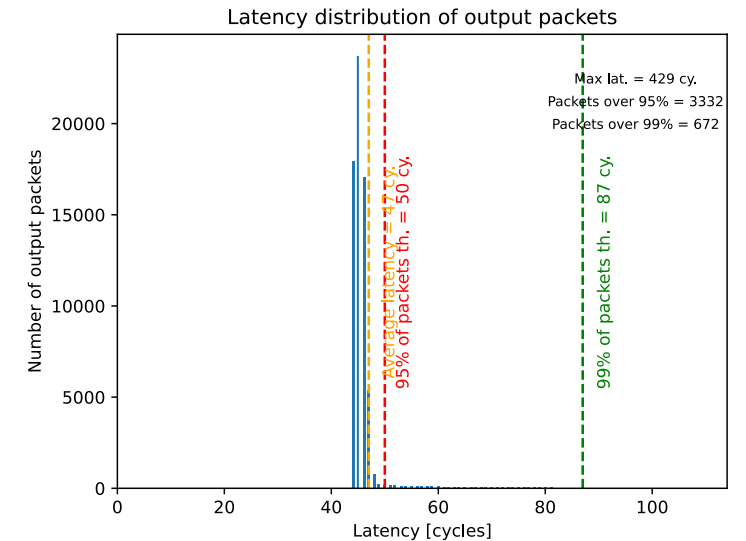
- *Performance*
- *Input analysis*
 - Compute and plot the *pixel hits VS BX* histogram of injected hits
 - Overall map of the injected pixel hits (it sums all the frames)
 - Overall map of the probability of injected pixel hits (hits/cycle) in log10
 - Cross-section cumulative plots along rows and columns of each layer
 - Cross-section cumulative plot (for the pixel column with MAX hits) along rows of each layer
- *Output analysis*
- *Memory occupancy*



Metrics Analyzer – output analysis

Metrics Analyzer generates metrics for:

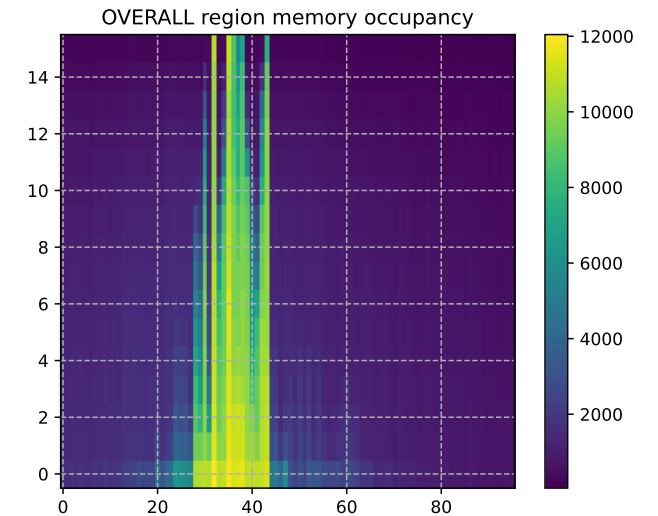
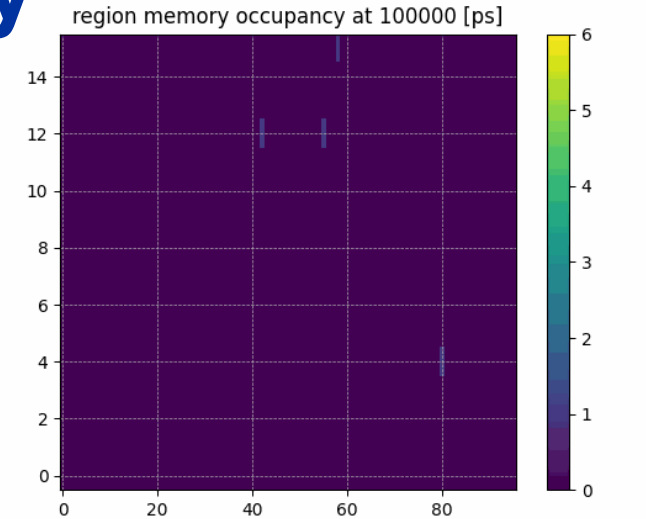
- *Performance*
- *Input analysis*
- *Output analysis*
 - Overall map of the missing pixel hits
 - Plot of output packets per cycle (output channels saturation)
 - Plot of latency distribution of output packets
 - Map of over-latency-threshold packets
- *Memory occupancy*



Metrics Analyzer – memory occupancy

Metrics Analyzer generates metrics for:

- *Performance*
- *Input analysis*
- *Output analysis*
- *Memory occupancy*
 - GIF of memory occupancy of each layer
 - Overall map of memory occupancy of each layer



Thank you!



home.cern