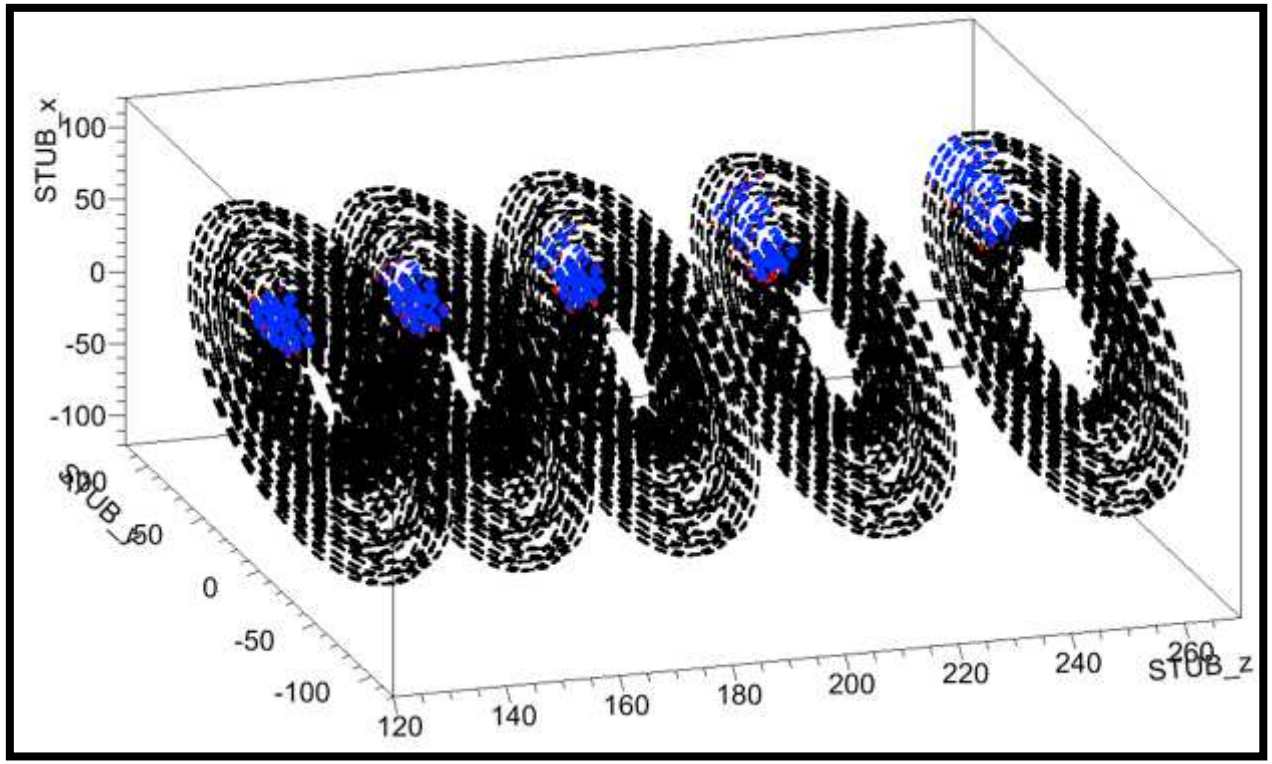
Fast track candidates 3D clustering



Algorithm,
Hardware implementation
And results

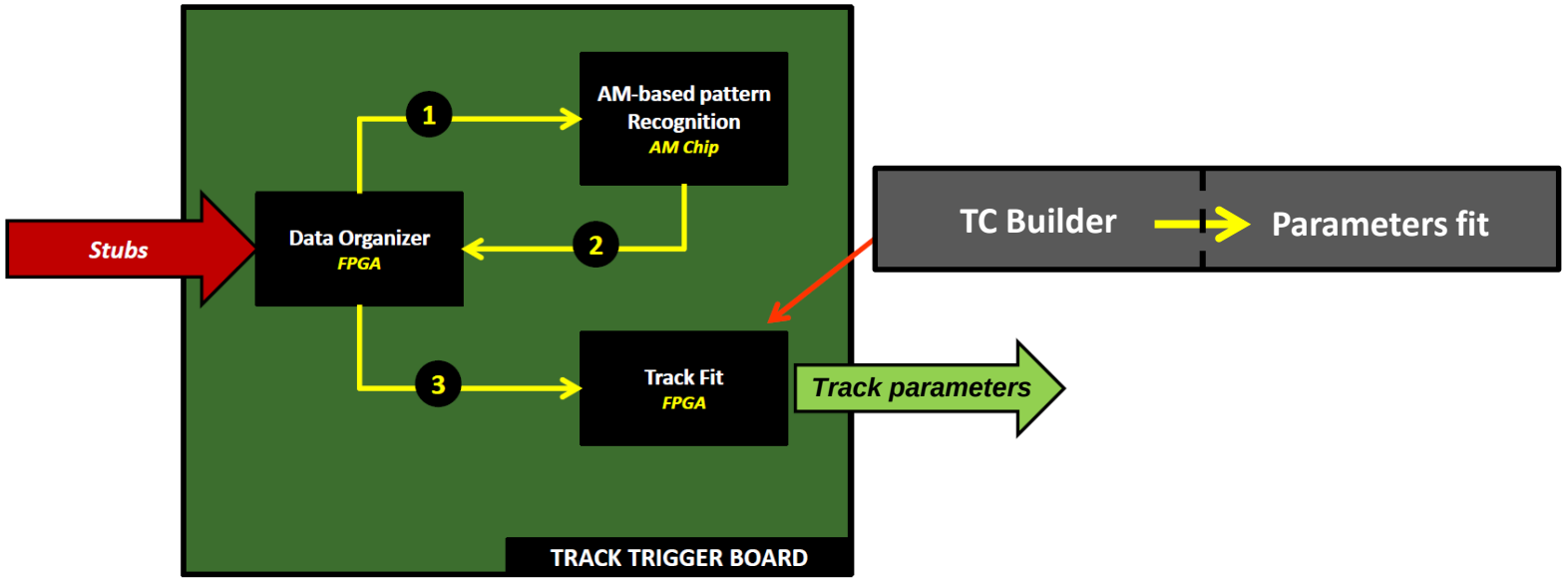
Geoffrey GALBIT
IPN Lyon

Introduction

→ **Context :**

→ This track filtering system takes place between the Associative Memories and the fit

→ The goal of this module is to provide clean Track Candidates (5 or 6 stubs) with a low fake rate

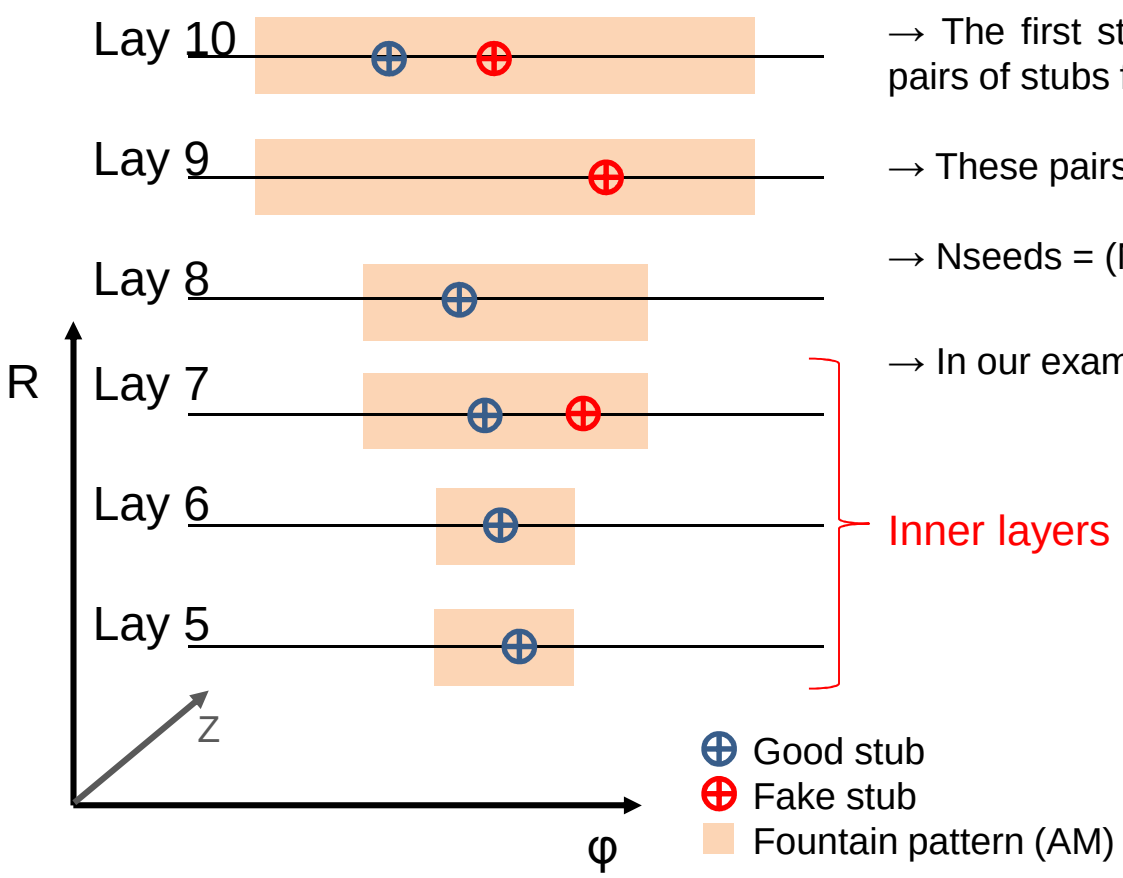


→ The TC Builder proceeds to a **3D clustering** of the stubs **pattern per pattern**

Algorithm overview

→ **Step by step algorithm :**

→ Each pattern is processed individually, treatment is operated simultaneously on (R, φ) and (R, Z) plans



→ The first step consist in generating all the possible pairs of stubs from different inner layers.

→ These pairs are the 3D seeds of the algorithm.

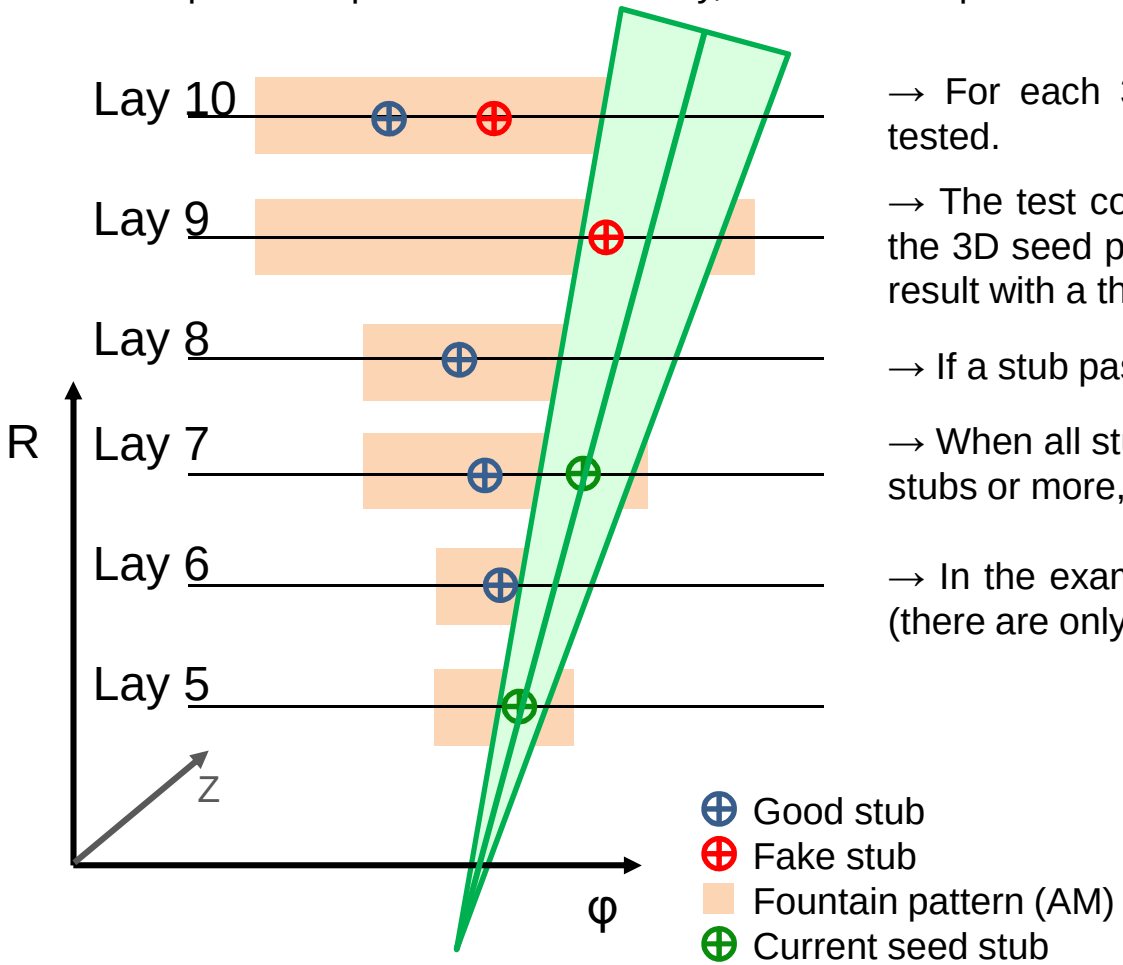
→ Nseeds = (N5 * N6) + (N6 * N7) + (N5 * N7)

→ In our example there are 5 possible different seeds.

Algorithm overview

→ **Step by step algorithm :**

→ Each pattern is processed individually, treatment is operated simultaneously on (R, φ) and (R, Z) plans



→ For each 3D seed, all the other layers stubs are tested.

→ The test consist in calculating the distance between the 3D seed projection and the stub and comparing the result with a threshold (green triangle).

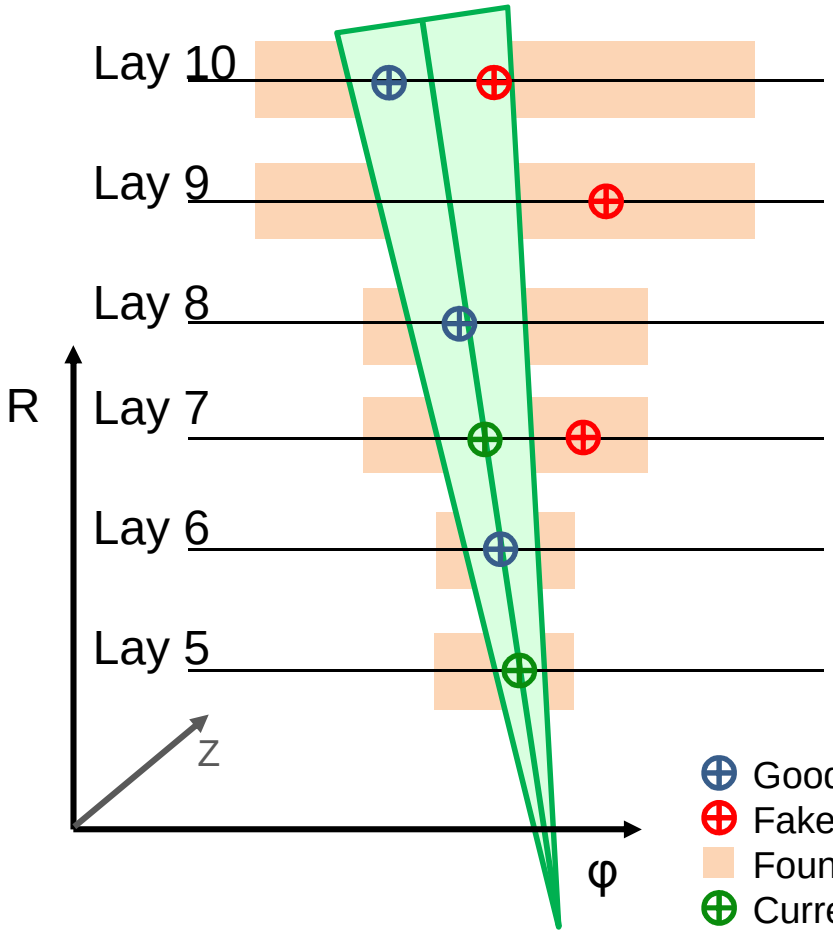
→ If a stub pass the test, it is added to the current TC.

→ When all stubs are tested, if the current TC contains 5 stubs or more, it is (stored in an intermediate memory).

→ In the example the TC is not recorded for this seed (there are only 3 stubs selected).

→ **Step by step algorithm :**

→ Each pattern is processed individually, treatment is operated simultaneously on (R, φ) and (R, Z) plans



→ When a stub pass the test and is recorded in the current TC, the distance is memorized.

→ If an other stub pass the test for the same layer, the results of the stubs are compared in order to keep the best alignment with the seed (only on R, φ plan).

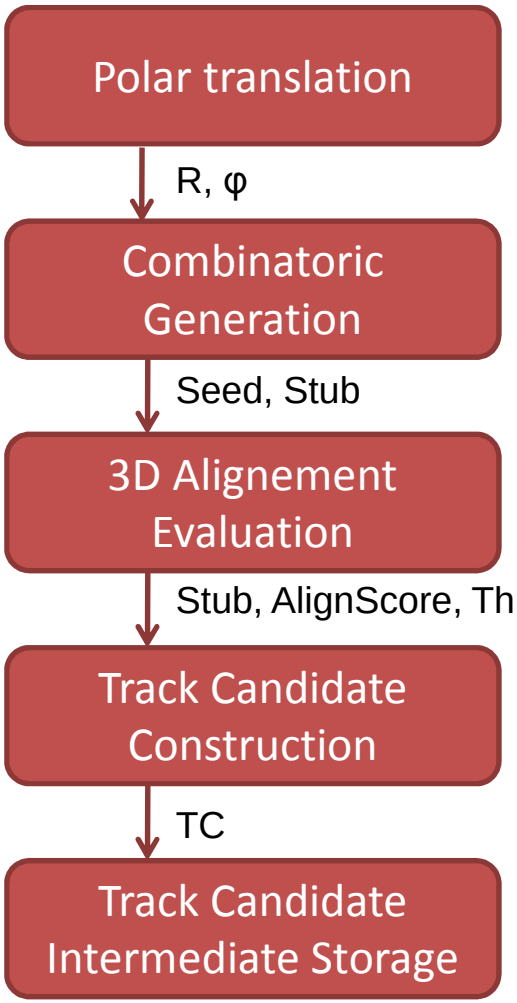
→ In our example the current TC is recorded for this seed (5 stubs selected).

→ When iterations over the seeds are finished, the intermediate memory content is read if there is a valid TC (never more than 1 TC per pattern).

→ nIter = (N5 * N6) * (Nstubs – N5 – N6)
+ (N6 * N7) * (Nstubs – N6 – N7)
+ (N5 * N7) * (Nstubs – N5 – N7)
- 2 * (N5 * N6 * N7)

→ For this example : nIter = 22

→ **Block diagram of one filtering module :**



→ Polar translation is operated by a CORDIC module for each stub of the pattern

→ For each pattern all the combinations seeds/stub are generated sequencially

→ For each iteration, stub is tested against the seed (3D alignment)

→ If the stub score is better than the threshold (or better than a previous stub of the same layer), the stub is added to the TC.

→ When iteration over stubs is finished for the current seed, if the TC have 5 stubs or more, it's memorized else it's dropped.
→ If there is already one memorized combination, priority rule is :

6 stubs > 3inLay + 2extLay > 2inLay + 3 extLay

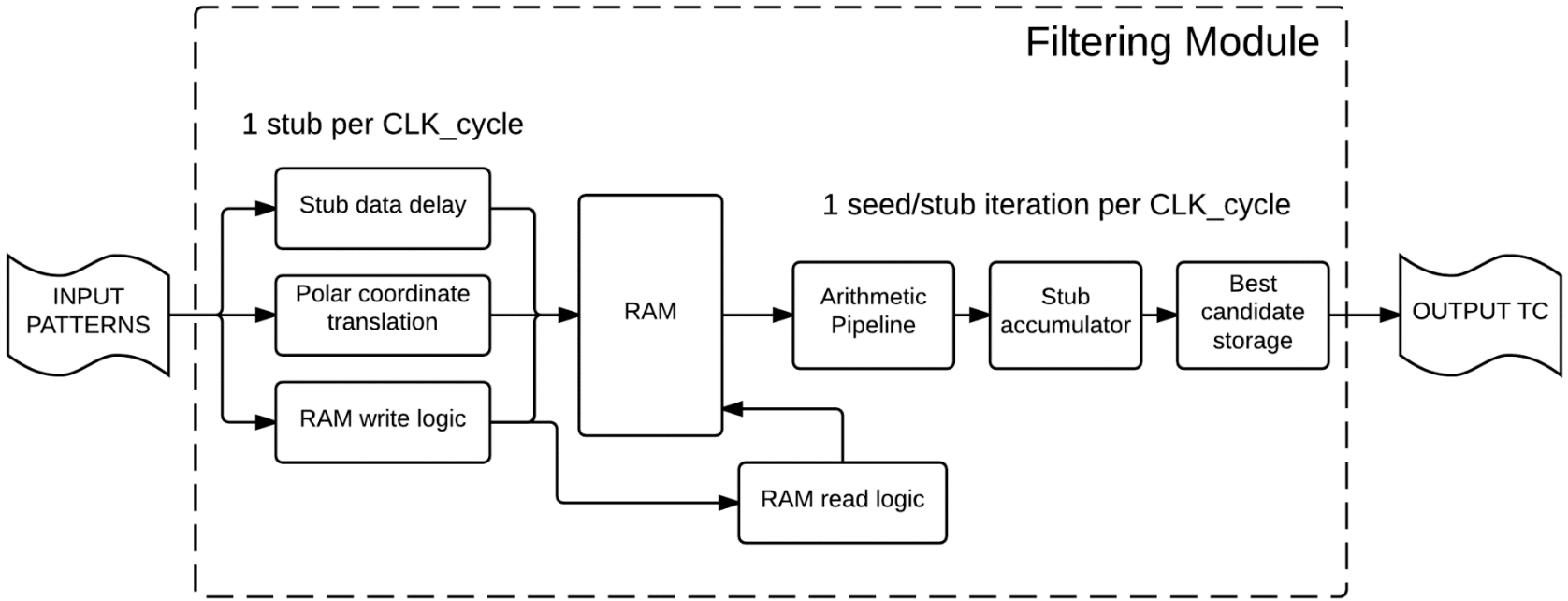
→ **Implementation of one filtering module :**

→ Module is fed with patterns content (1 stub per clock cycle).

→ In the first pipeline, polar translation is processed by CORDIC while an other module compute the address where the stubs will be stored in the RAM and begins to send informations to the read logic.

→ When there is at least one complete pattern in the memory, the read logic begins to generate all the read addresses corresponding to an iteration (1 seed + 1 stub).

→ Last part of the module is a pipeline which tests alignment and memorizes stubs to construct TC.



→ **Focus on the 3D alignment evaluation module :**

→ Alignment test on R, φ :

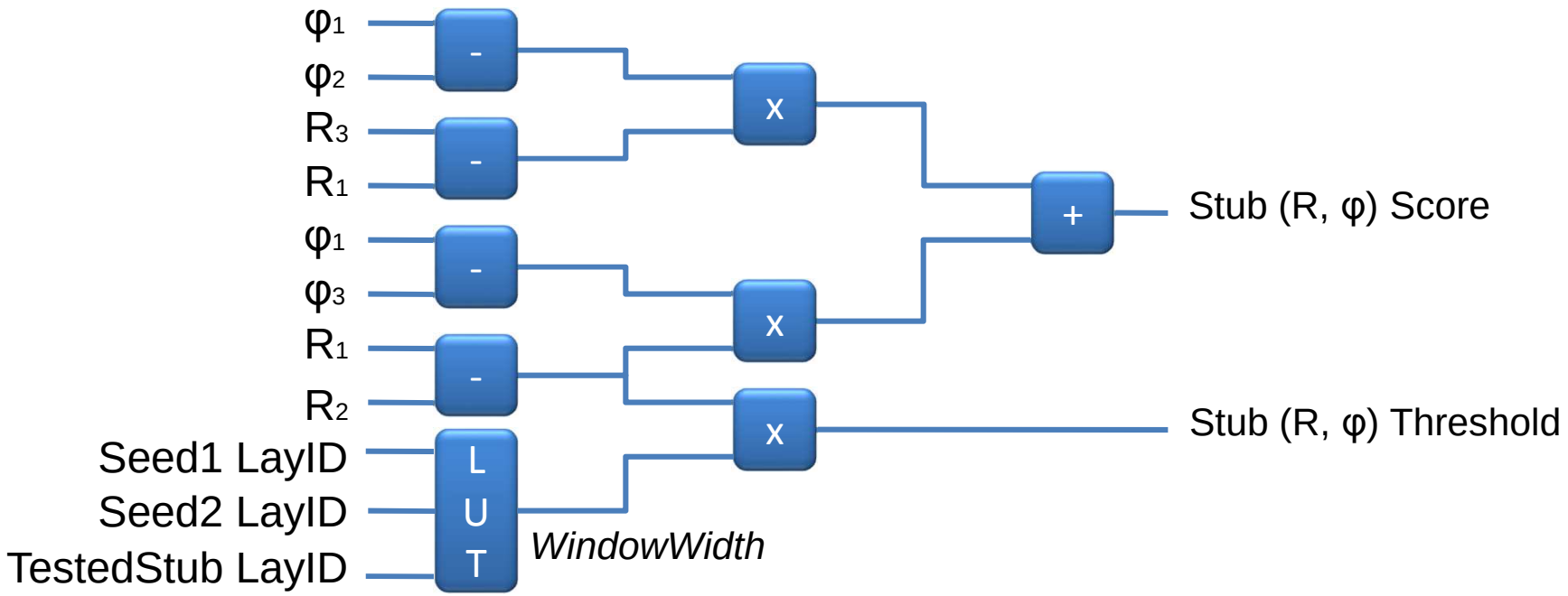
$$\underbrace{|(\varphi_1 - \varphi_2) * (R_3 - R_1) + (\varphi_1 - \varphi_3) * (R_1 - R_2)|}_{\text{Stub (R, φ) Score}} \leq \underbrace{|WindowWidth * (R_1 - R_2)|}_{\text{Stub (R, φ) Threshold}}$$

Seed1 (R_1, φ_1, Z_1)
Seed2 (R_2, φ_2, Z_2)
TestedStub (R_3, φ_3, Z_3)

→ Same processing is operated in parallel on the R, Z plan (with different thresholds)

→ This processing step is fully pipelined

→ Multiplications are operated by DSP Slices (up to 18 bits per operand for 1 Xilinx DSP Slice)



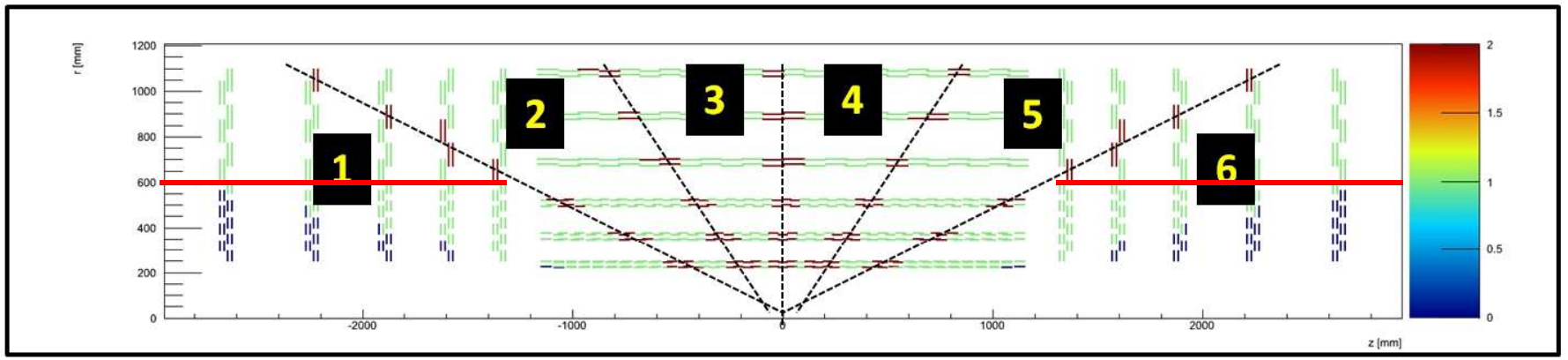
Full detector adaptation

→ **Algorithm adaptation for the other tower types (hybrid and end-cap) :**

→ Straight forward to adapt to other towers (exactly the same algorithm)

→ PS and 2S modules of the disk have to be split (they receives 2 distinct sets of acceptance window width during the calibration)

BE_5D/Eta6_Phi8 configuration



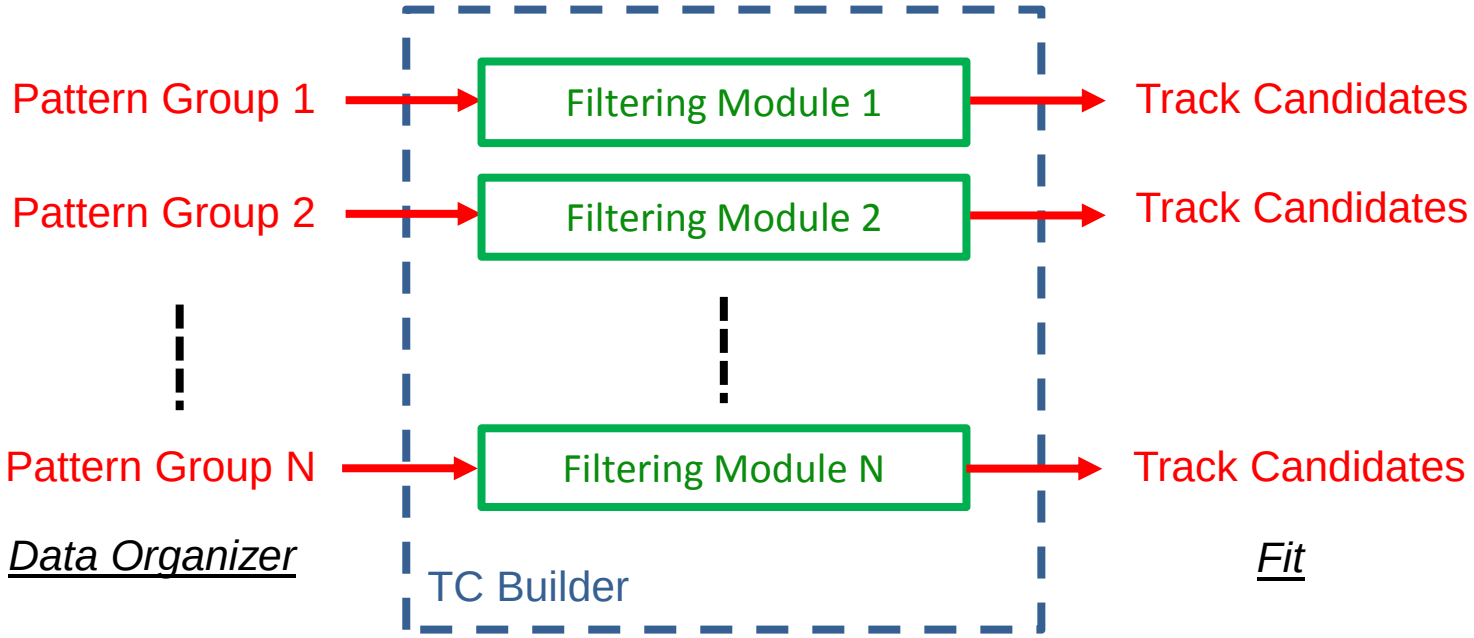
Eta range	1	2	3	4	5	6
Sector numbers	*0→7*	*8→15*	*16→23*	*24→31*	*32→39*	*40→47*
Active layers	*5 6* 18 19 20 21 *22*	5 6 7 8 *9 10 18 19*	5 6 7 8 9 10	5 6 7 8 9 10	5 6 7 8 *9 10 11 12*	*5 6* 11 12 13 14 *15*

→ **Full TC Builder :**

→ The patterns of an event have to be split into different groups

→ A Clustering Module is light enough to be instanciate many times

→ The required N_Modules can be defined thanks to a latency study



→ **Ressource utilization :**

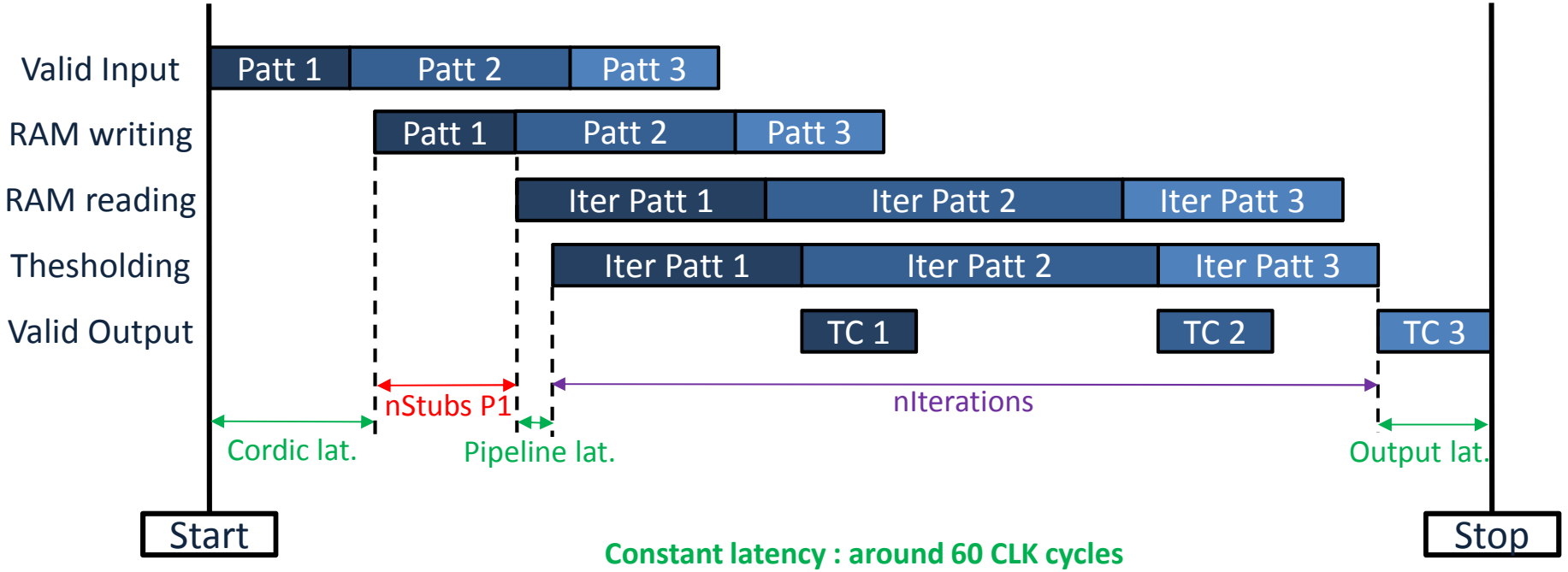
→ Vivado ressource utilization estimation for one filtering module on a Kintex 7 FPGA :

Resource	Estimation	Available	Utilization %
FF	6100	407600	1.50
LUT	3731	203800	1.83
Memory LUT	3	64000	0.01
I/O	311	500	62.20
BRAM	1	445	0.22
DSP48	6	840	0.71
BUFG	1	32	3.12

→ Note : It is not relevant to consider the I/O line (inputs and outputs of the FPGA) while the goal of this module is to be inserted between the Data Organizer and a fitter present on the FPGA.

→ **Latency of the filtering module:**

→ During one event, a module can manage multiple patterns



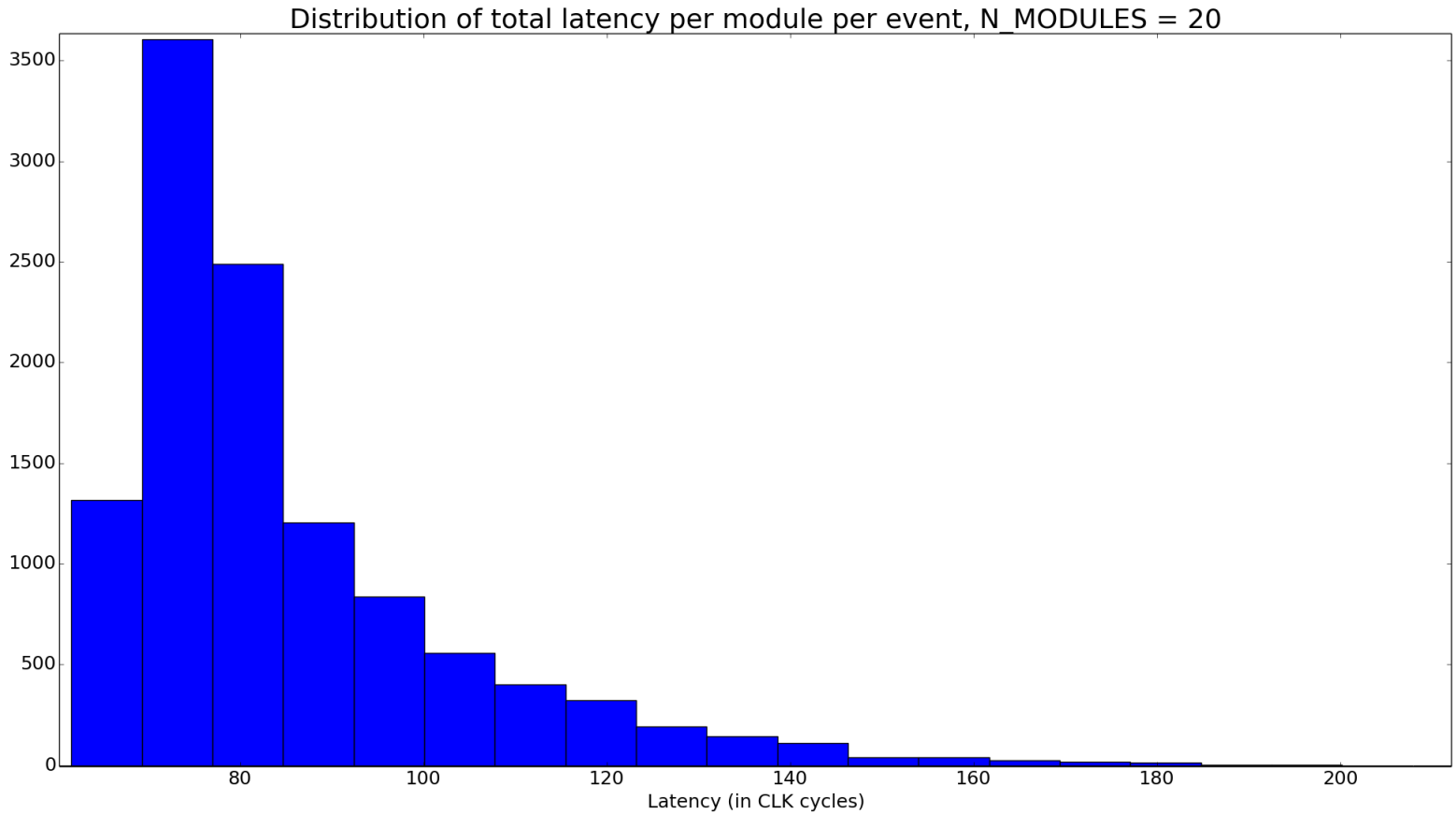
Total latency = Constant latency + size of the 1st pattern + sum of nIterations for all the patterns of the module

Latency Study

→ **Number of clock cycles required (per filtering module per event) :**

→ Round robin algorithm (each module take alternatively a pattern until there is no more)

→ This latency can be reduced thanks to a better algorithm (better load repartition between modules)



Efficiency

→ **Current status:**

	Muons		Electrons		PU140		PU200	
	$p_T \geq 3$	$p_T \geq 10$	$p_T \geq 3$	$p_T \geq 10$	$p_T \geq 3$	$p_T \geq 10$	$p_T \geq 3$	$p_T \geq 10$
ϵ^{SR}	$100.0_{-0.05}$	$100.0_{-0.05}$	97.0 ± 0.1	98.4 ± 0.1	98.0 ± 0.1	98.4 ± 0.2	98.1 ± 0.1	98.5 ± 0.2
ϵ^{TW}	99.9 ± 0.05	99.9 ± 0.05	99.8 ± 0.05	99.8 ± 0.05	$100.0_{-0.05}$	$100.0_{-0.05}$	$100.0_{-0.01}$	$100.0_{-0.05}$
ϵ^{AM}	99.0 ± 0.05	99.1 ± 0.05	94.9 ± 0.1	95.8 ± 0.1	97.8 ± 0.1	99.5 ± 0.1	97.8 ± 0.1	99.3 ± 0.2
ϵ^{CB}	99.9 ± 0.05	99.9 ± 0.05	98.4 ± 0.1	98.5 ± 0.1	99.3 ± 0.05	99.4 ± 0.1	99.1 ± 0.1	99.3 ± 0.2
$\epsilon^{SR \to CB}$	98.8 ± 0.05	98.9 ± 0.05	91.4 ± 0.2	92.8 ± 0.2	95.2 ± 0.1	97.3 ± 0.2	95.0 ± 0.1	97.2 ± 0.3

Table 6: Efficiencies up to the combination builder, for $p_T \geq 3$GeV/c

→ > 98% in all the case for the TC Builder stage (not affecting the overall efficiency)

→ Same effect in all the towers

→ fixed point 18 bits give the same results as the floating point software simulation

Conclusion

→ **Conclusion :**

→ 3D thresholding allows a good fake rejection.

→ TC are perfectly formated for the fit step (5 or 6 stubs per TC).

→ An unique solution for barrel, hybrid and endcap towers.

→ **Future work :**

→ Add some optional features to the C++ (and future CMSSW) code, like the ability to emulate the HW binning and to estimate the latency needed by the TC Builder to process an event.

→ Increase the filtering frequency (goal = 400 MHz).

→ Continue the global optimization of the filtering modules.

→ Implement a coarse track parameters estimator.

→ Insert the TC builder implementation between the Data Organizer and a Fitter