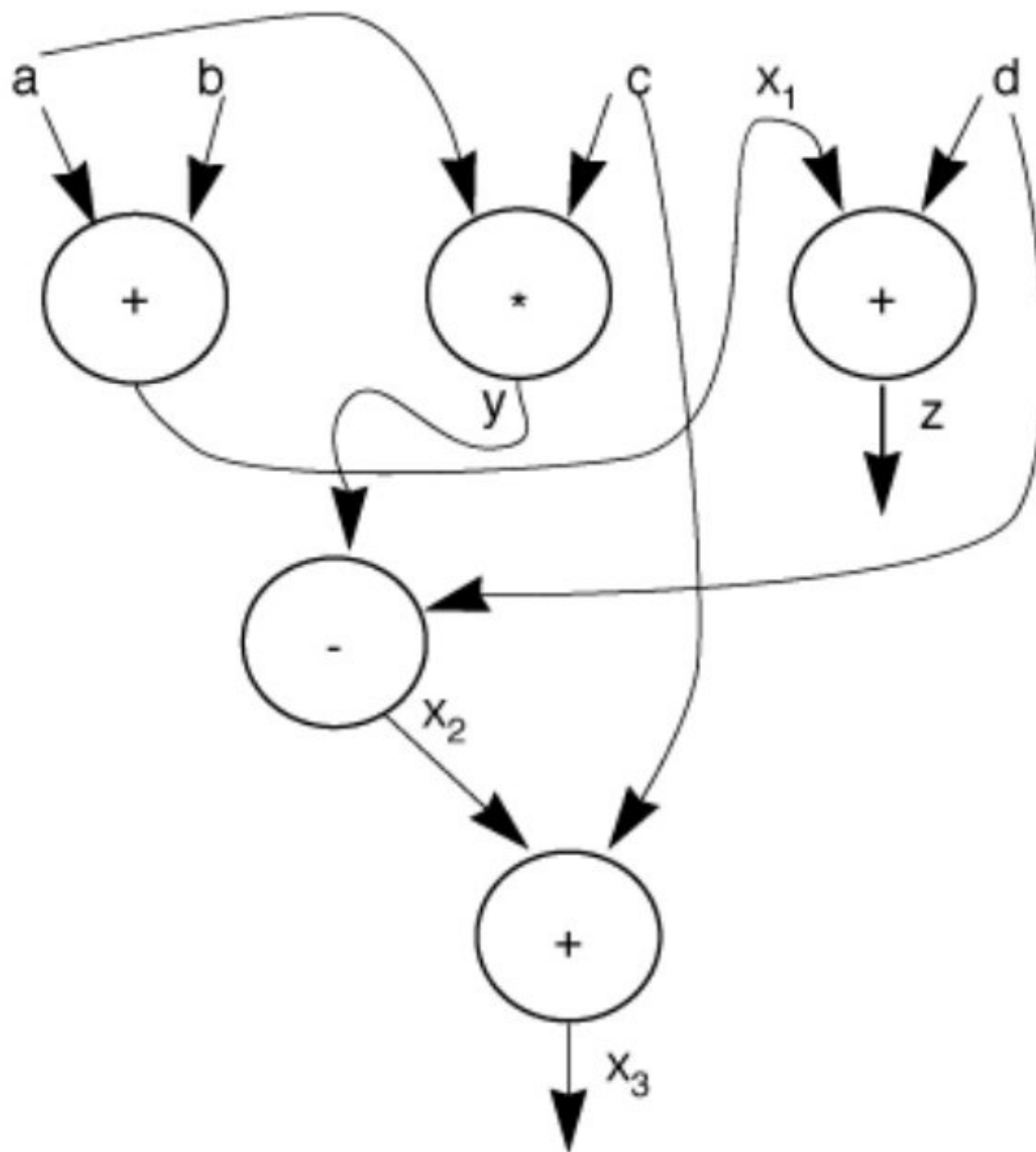




life.augmented



# High-Level Synthesis: Transforming the Design of Heterogeneous System-on-Chips

David VINCENZONI

Marcello DUSINI

Gianluca RIGANO

June 12<sup>th</sup>, 2025

# Agenda

1 Introduction

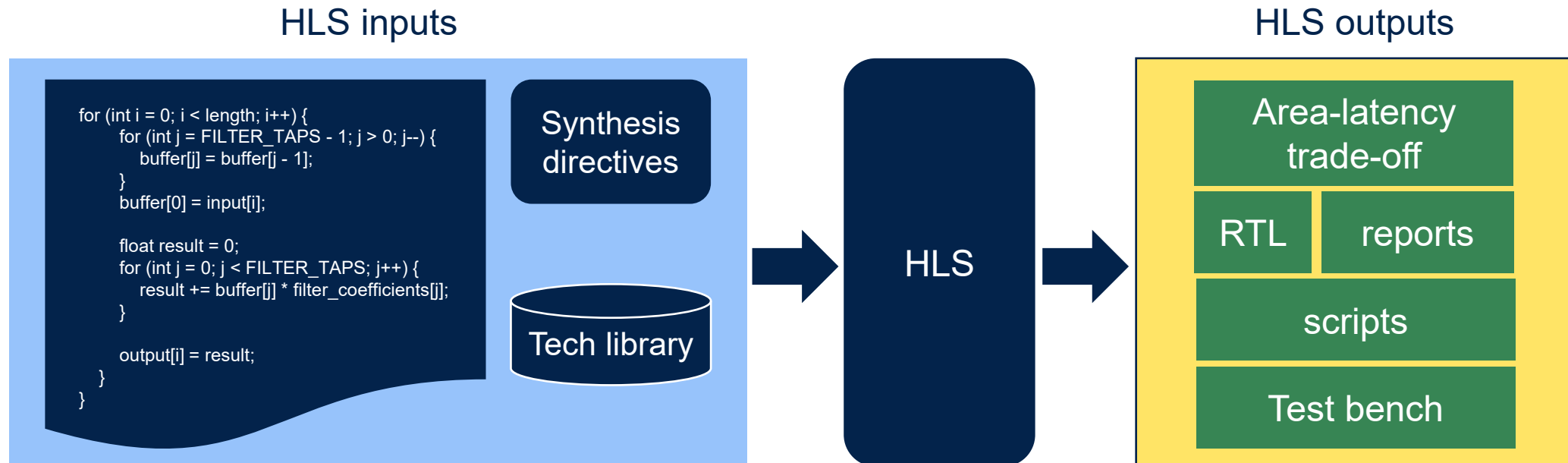
2 Catapult experience

3 Stratus HLS experience

4 Conclusions

# What is High Level Synthesis?

- The process of converting an untimed or partially timed behavioral description into efficient hardware\*



# Current flow

System Engineer

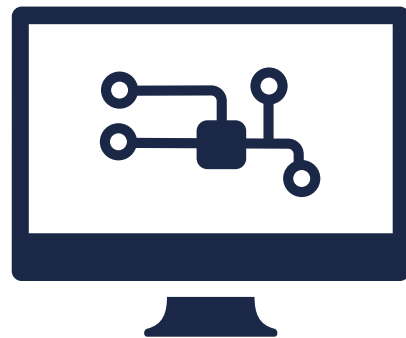


Algo  
Builder



MATLAB®/C

Designer



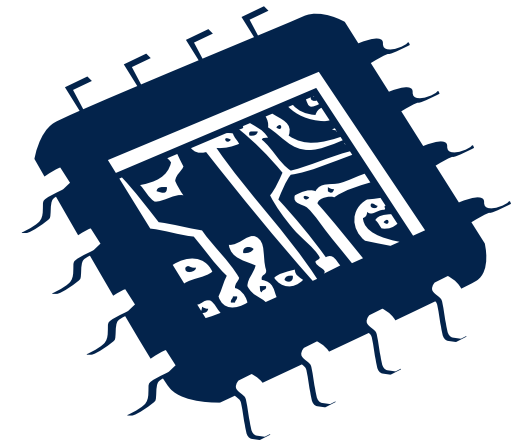
SystemVerilog

Verification

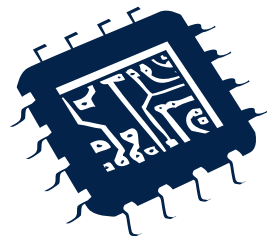
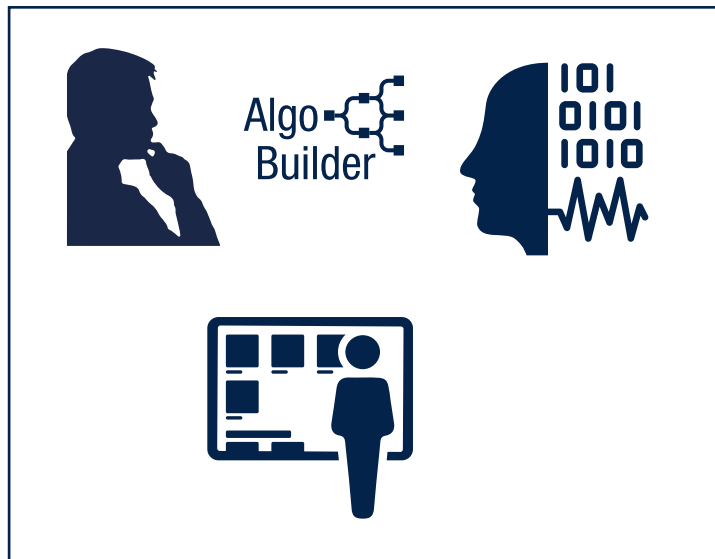


Formal/UVM

Logical Synthesis



# Current flow: What if we do not match the requirements?



Timing not met!



Combinatorial paths tool long

Latency not met!



Pipeline too long

Power not met!



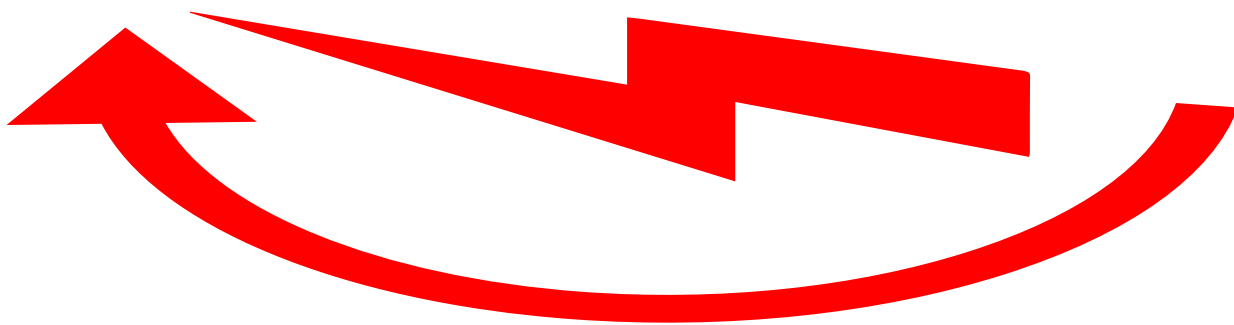
Missing clock gating cells

Area not met!



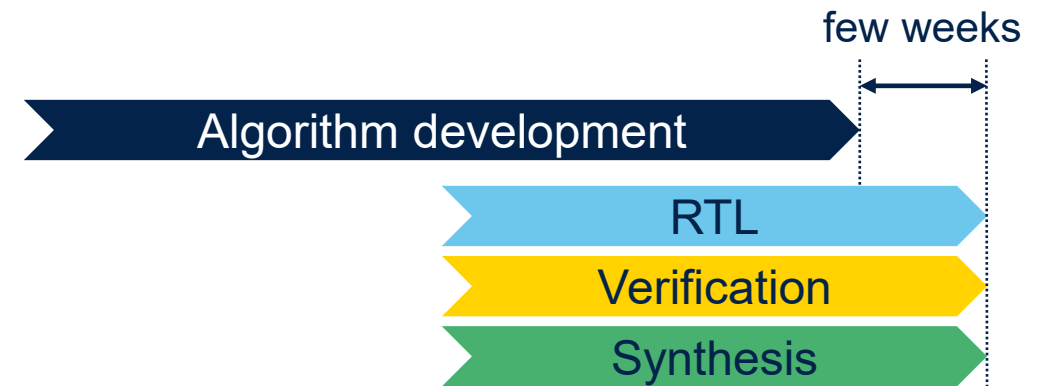
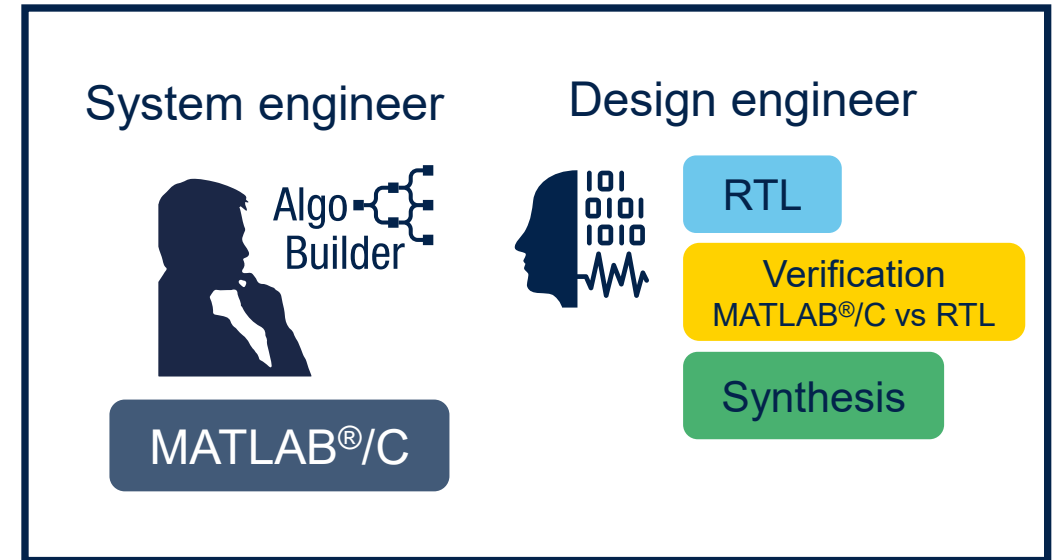
No resource sharing

Requirements met!



# What happens when HLS is used?

- Using the HLS flow implies a change in mindset
- The system engineer works closely with the designer from the beginning
- The HLS tools allow:
  - Running verification tests
  - Exploring different implementation options
  - Performing synthesis for the target technology
  - Providing immediate feedback to the system engineer

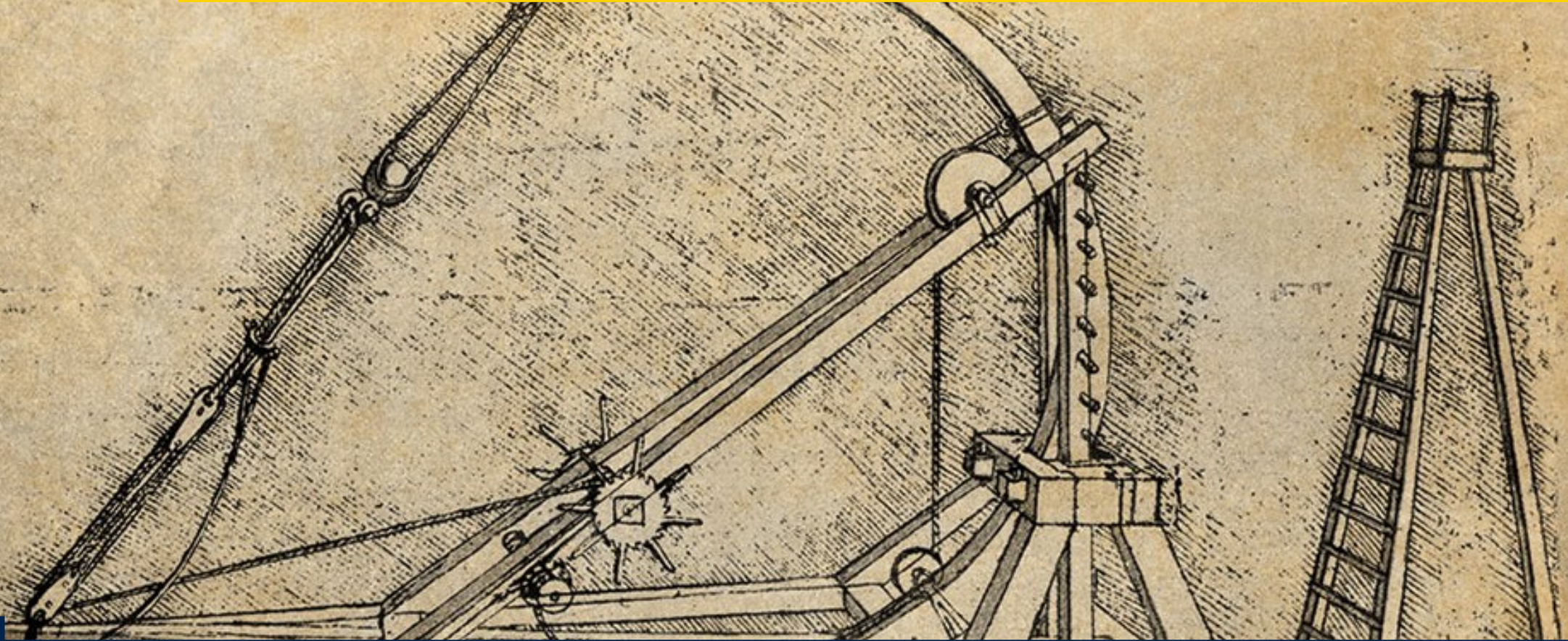


# Why and benefits of HLS

- Many SoCs are heterogeneous and contains dedicated HW accelerator such as signal processing blocks, not only a simple digital filter
- Those HW accelerators are first developed using high-level languages such as C/C++ or MATLAB®
- The HLS approach has multiple benefits
  - Much easier than the use of Verilog; less lines of code
  - Simulations are much faster (typical 10x faster)
  - Possibility to **explore** a variety of **functional equivalent HW** implementation from the **same behavioral** function

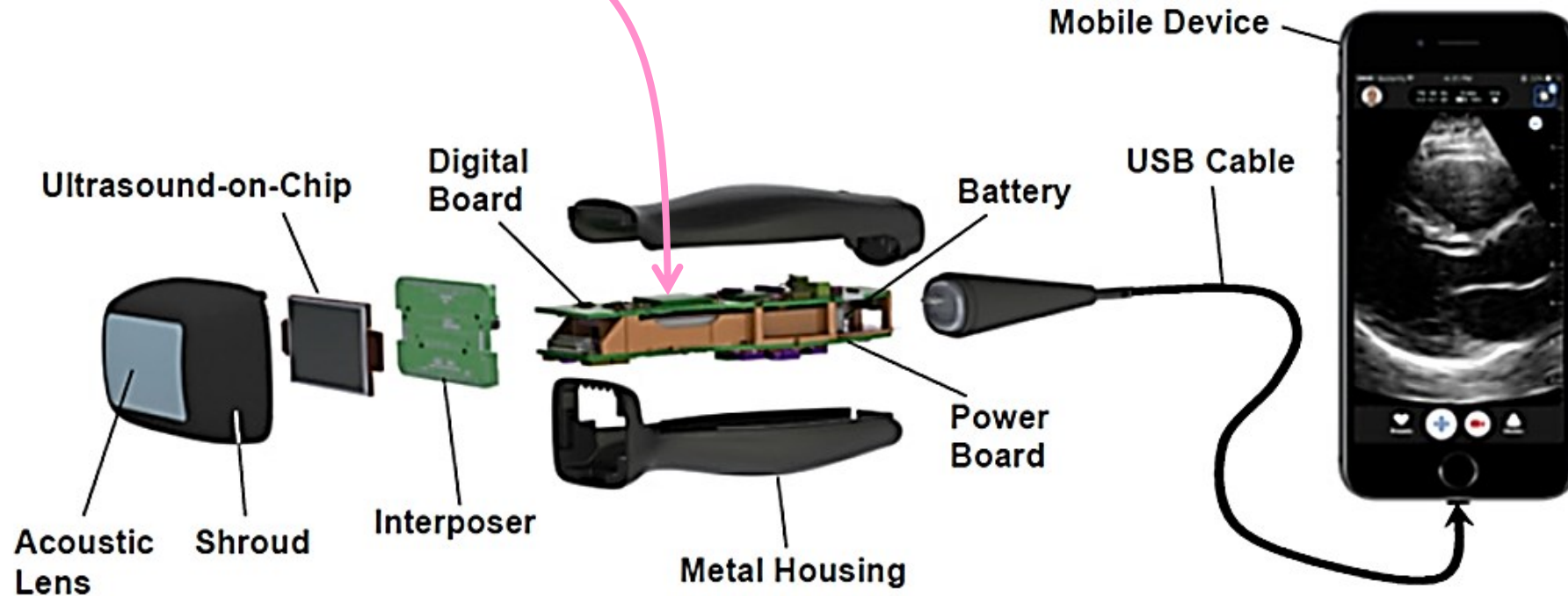


# Catapult experience



# Ultrasound portable application

**Our chip here!**  
Dedicated hw acceleration



# Timing of the Acquisition

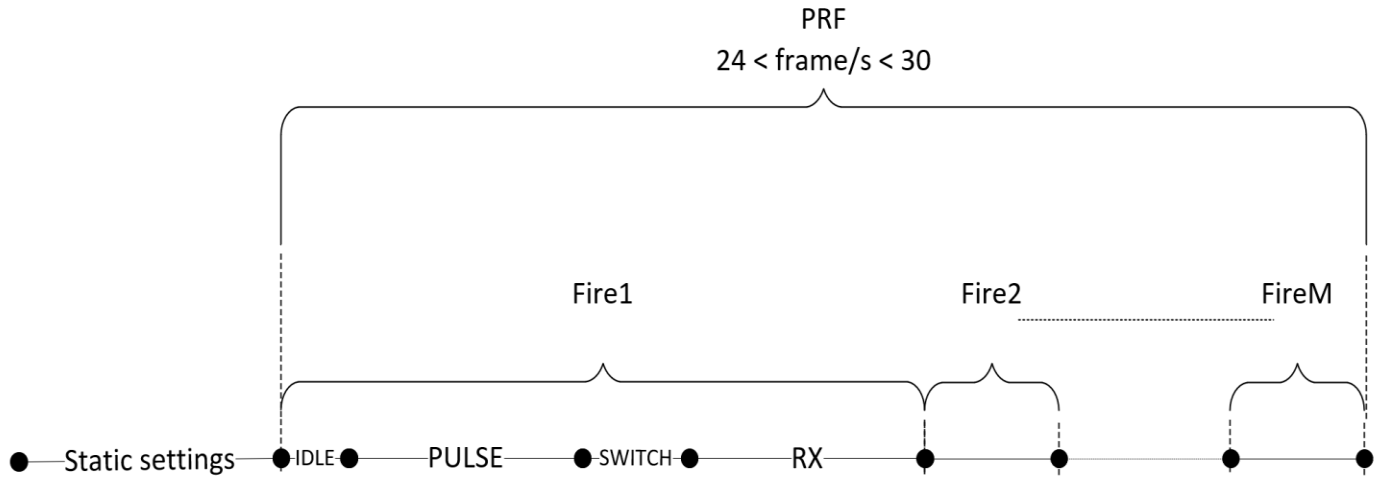
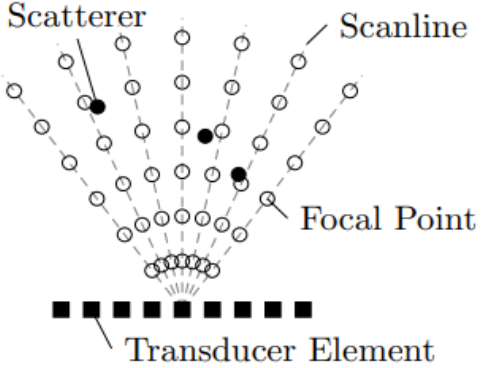
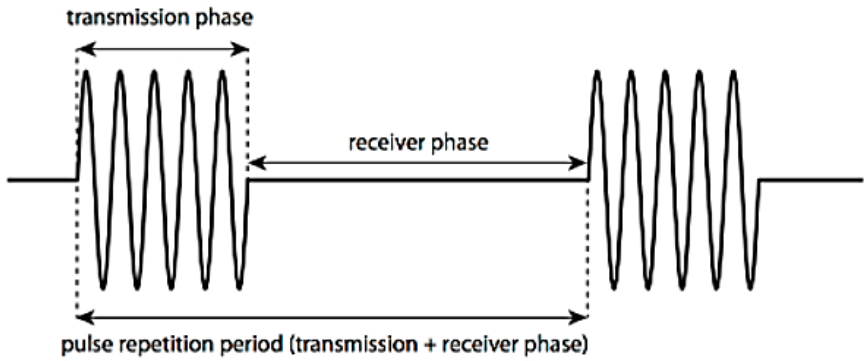
Two phases



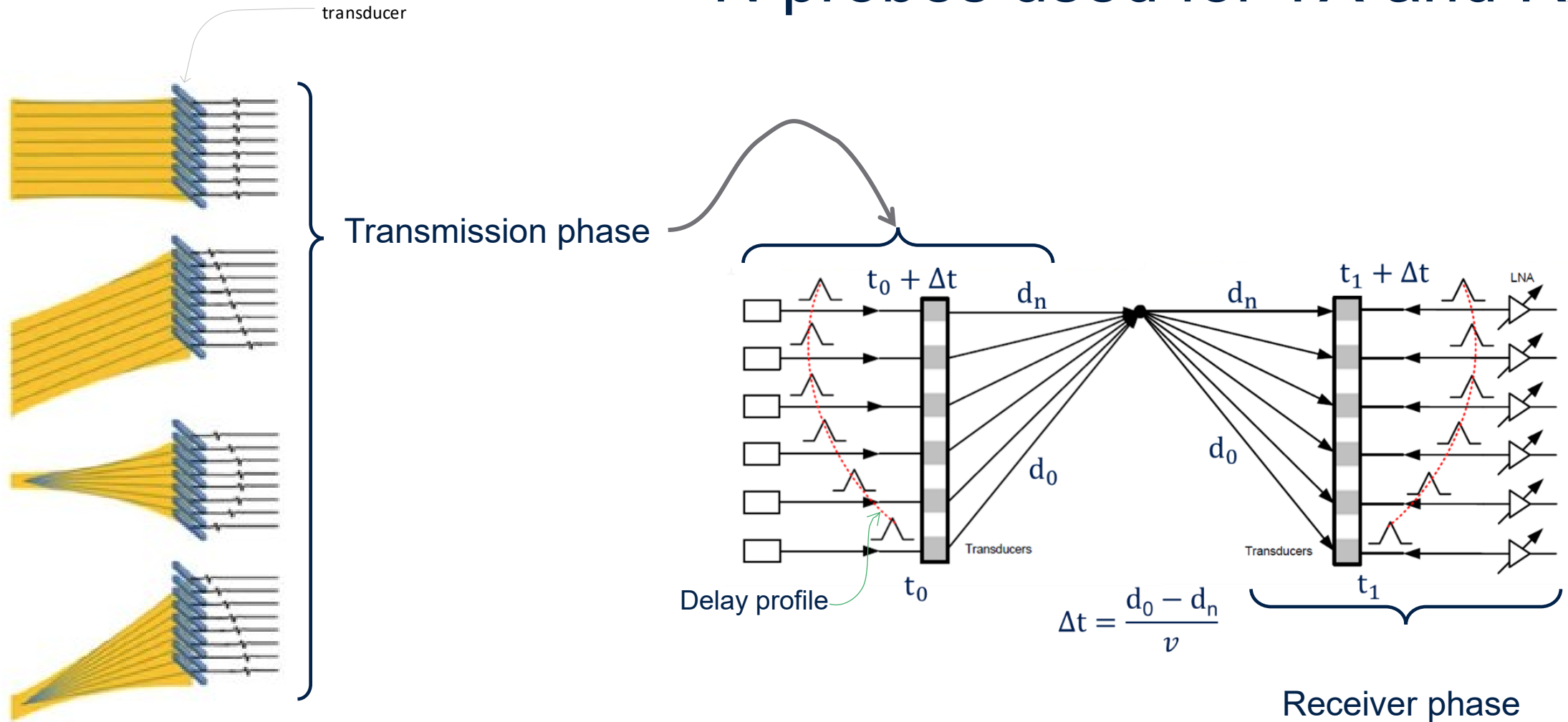
M scanlines



1 frame



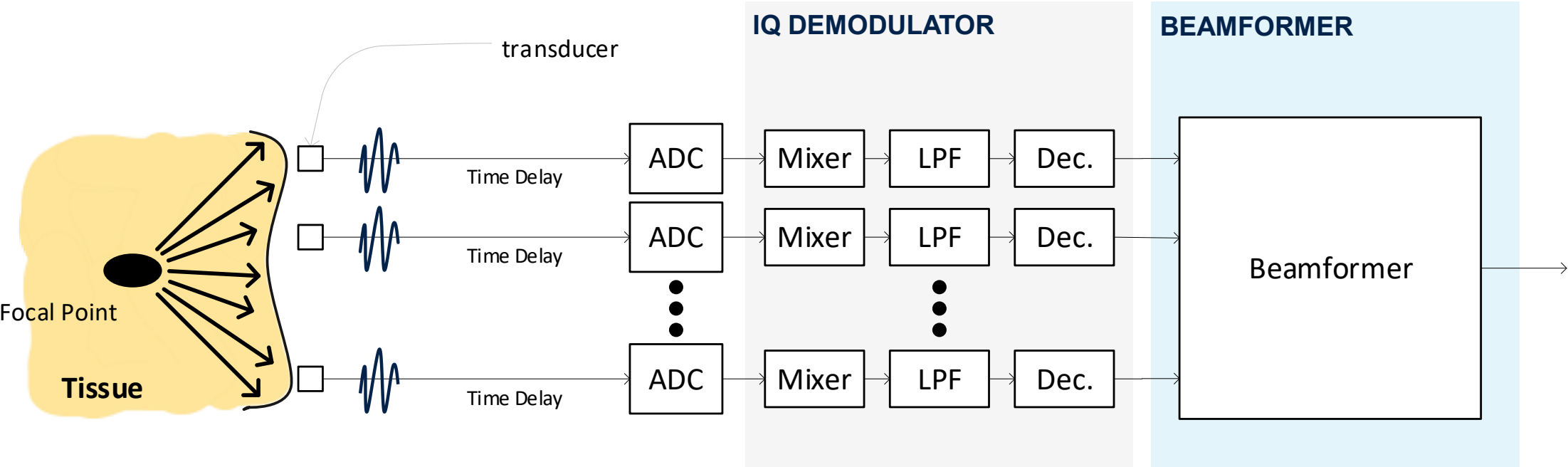
# N probes used for TX and RX



# Signal processing for the receiver

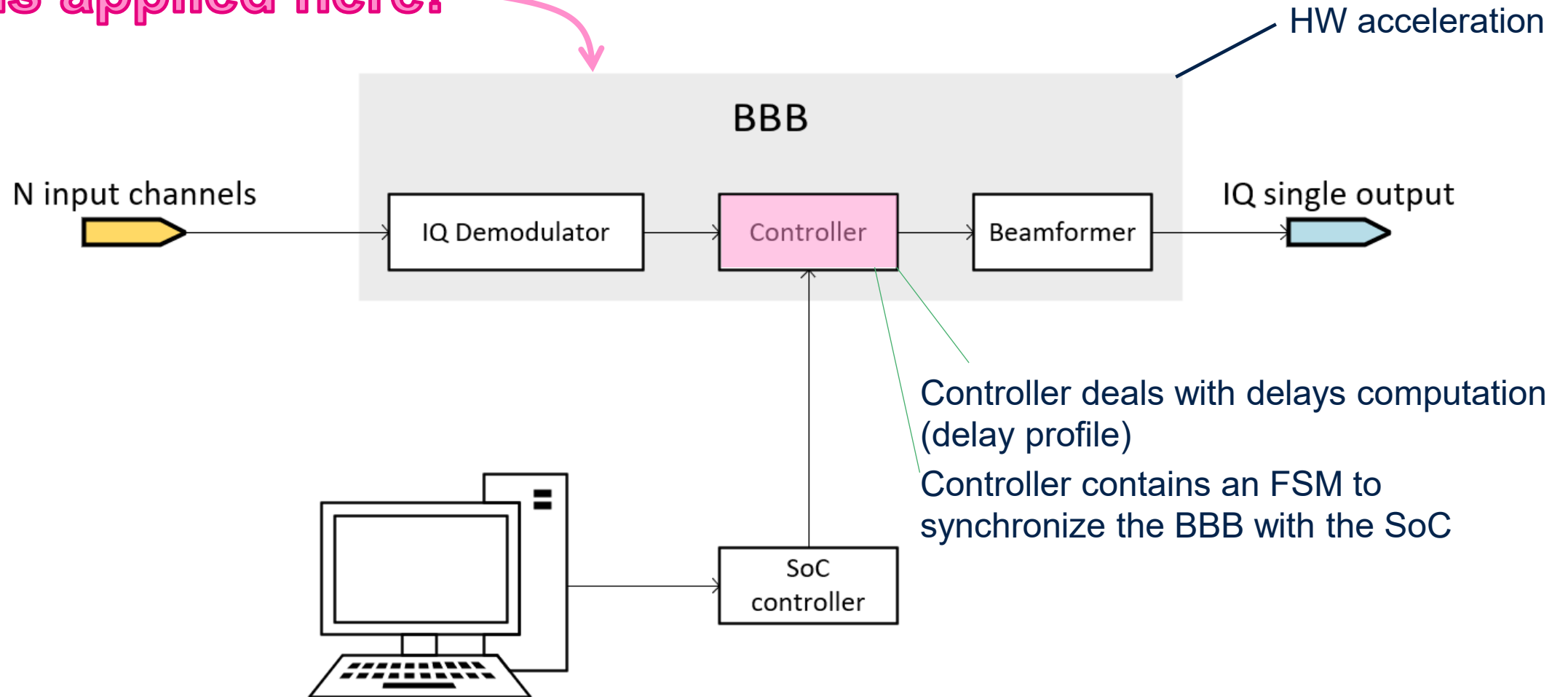


# Base Band Datapath



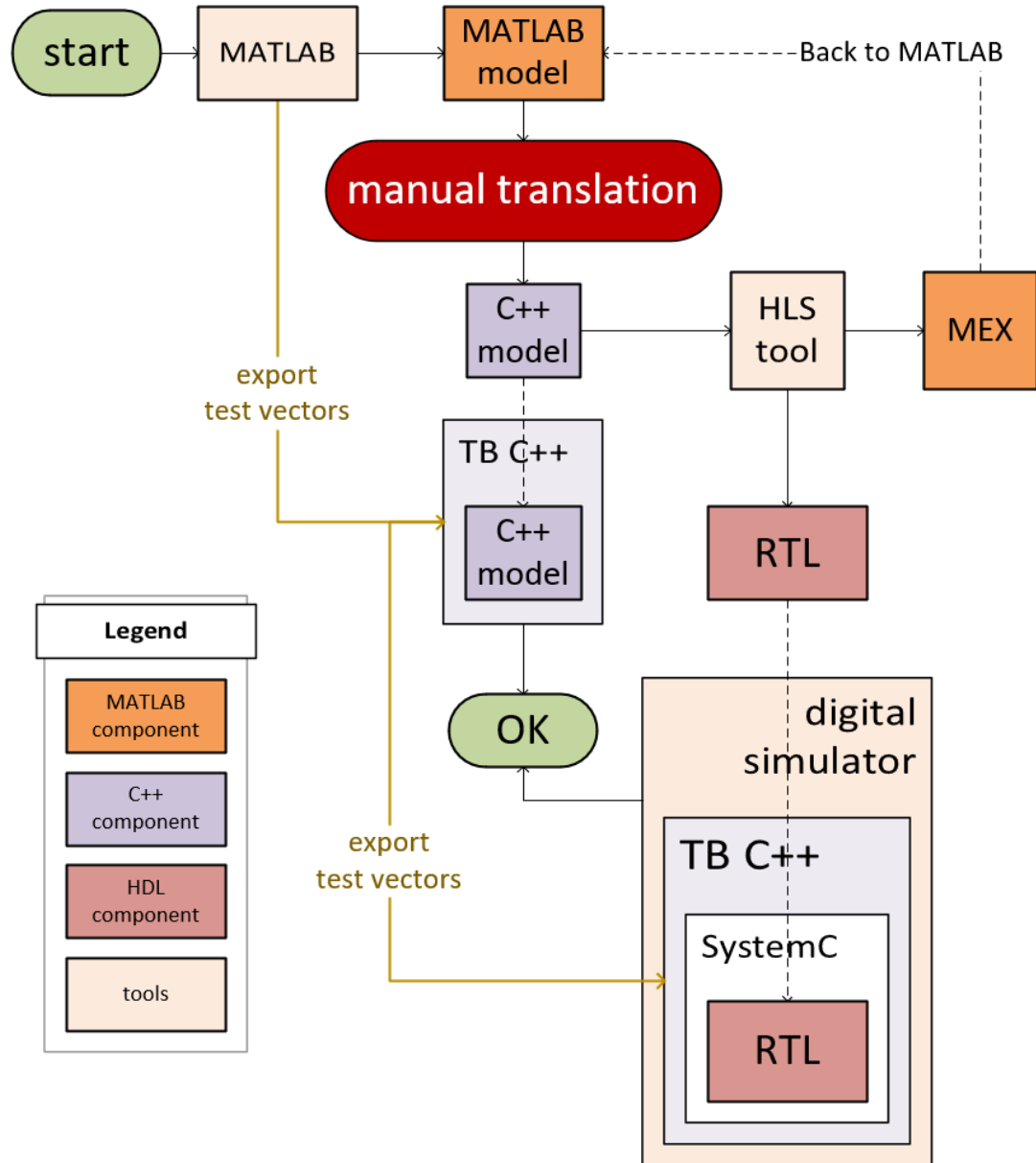
# Base Band Beamformer Architecture

HLS is applied here!



# HLS Flow

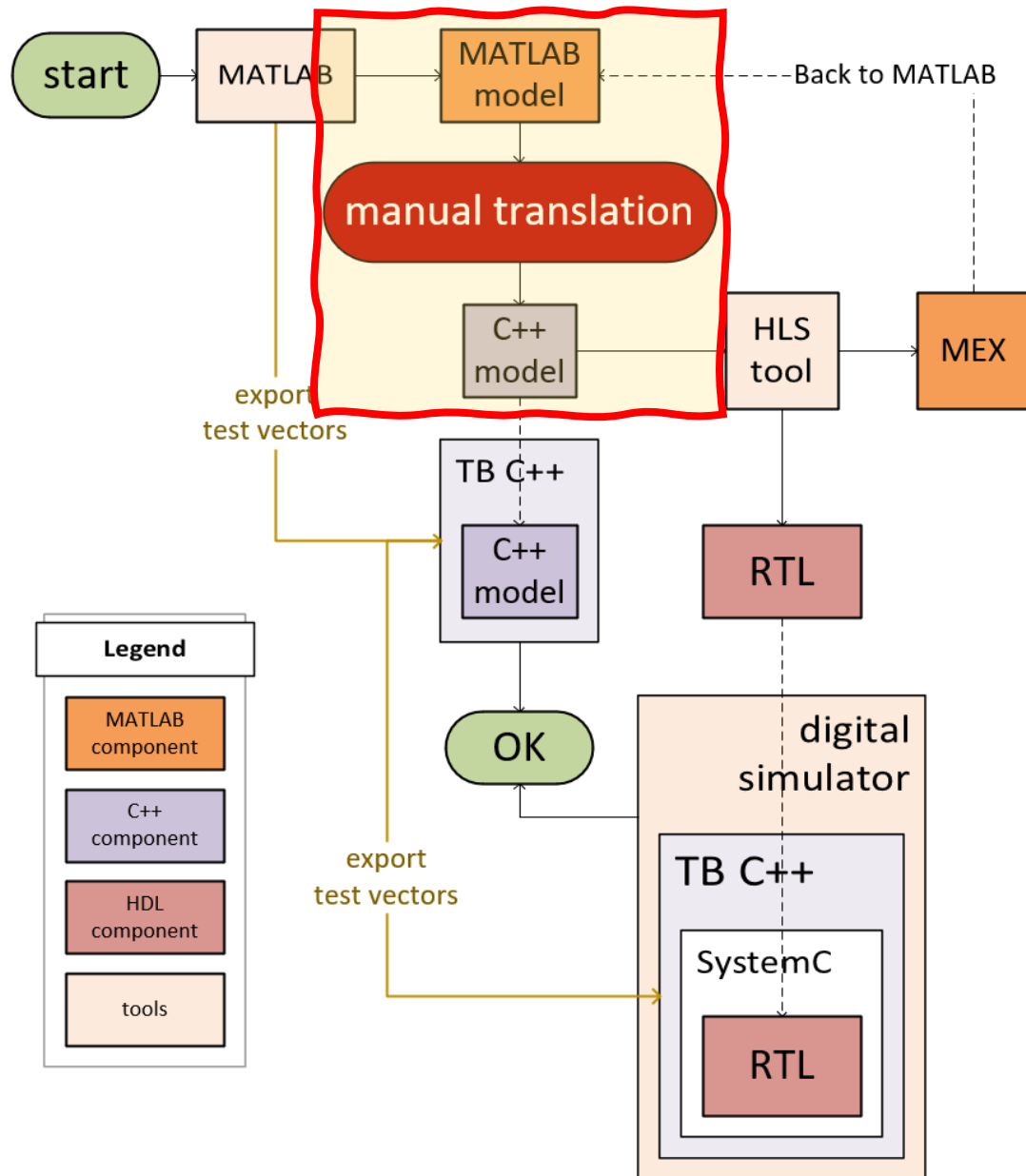
# Flow with Catapult



- manual translation
- C++ coding style
- HLS extraction techniques

C++

# Coding style



## C++ code synthesizable for RTL extraction

- Data types : fixed point
- Loops: definite number of iterations
- Interfaces
  - ✓ ac\_channel class
  - ✓ direct inputs

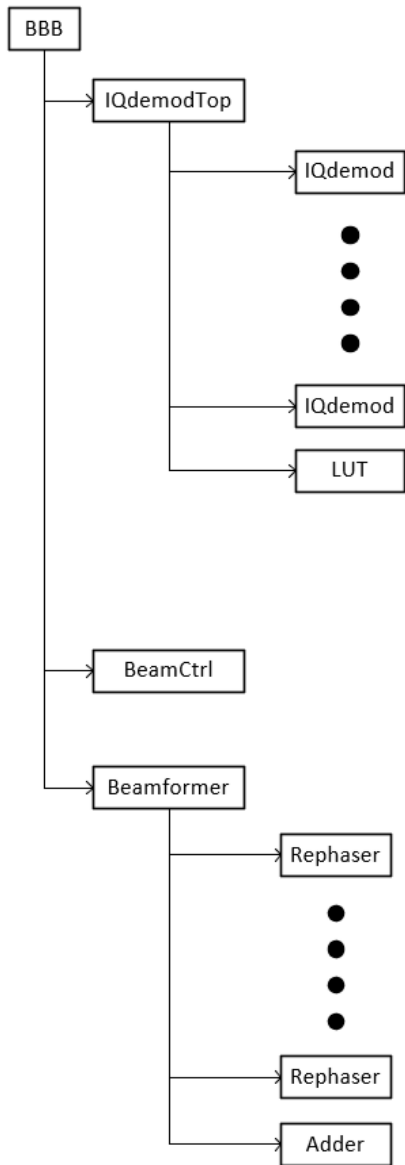
# Hierarchical style

the bigger the design → the harder the work for the tool

## Hierarchy!

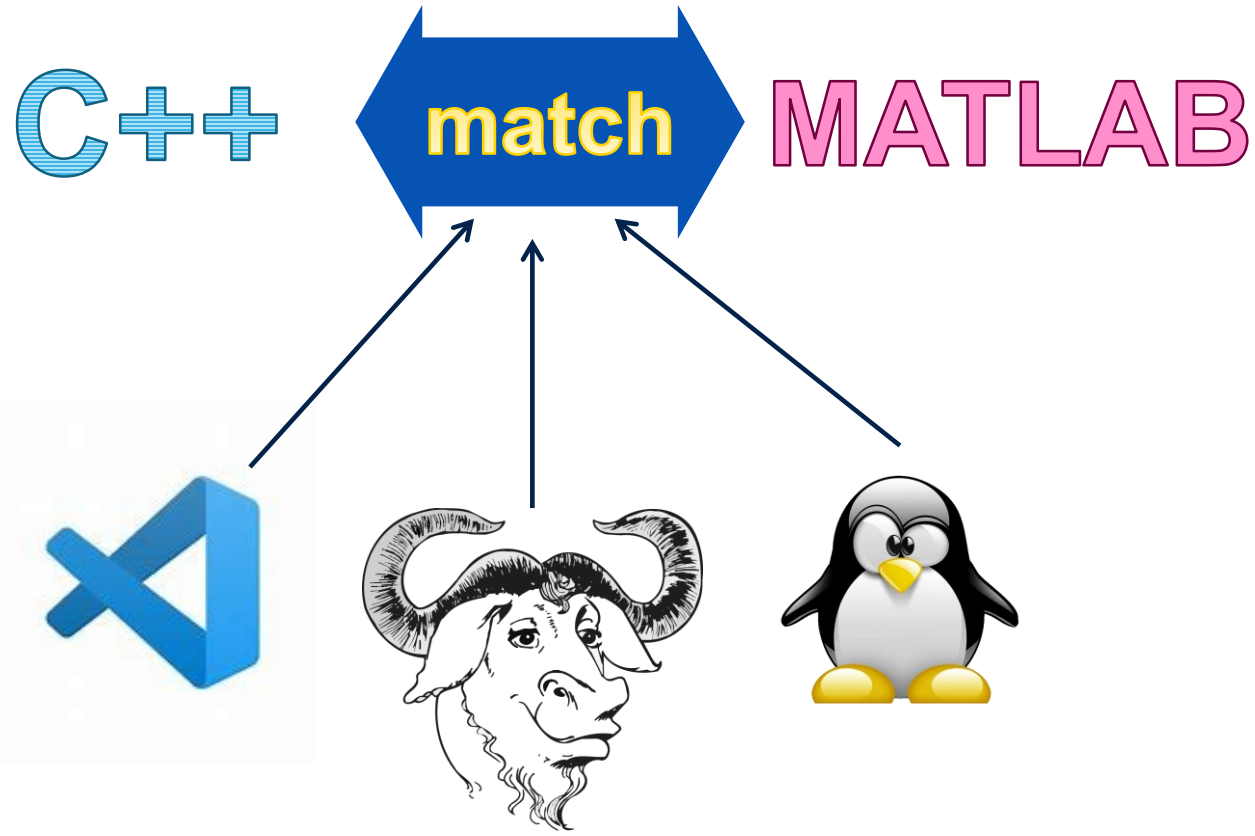
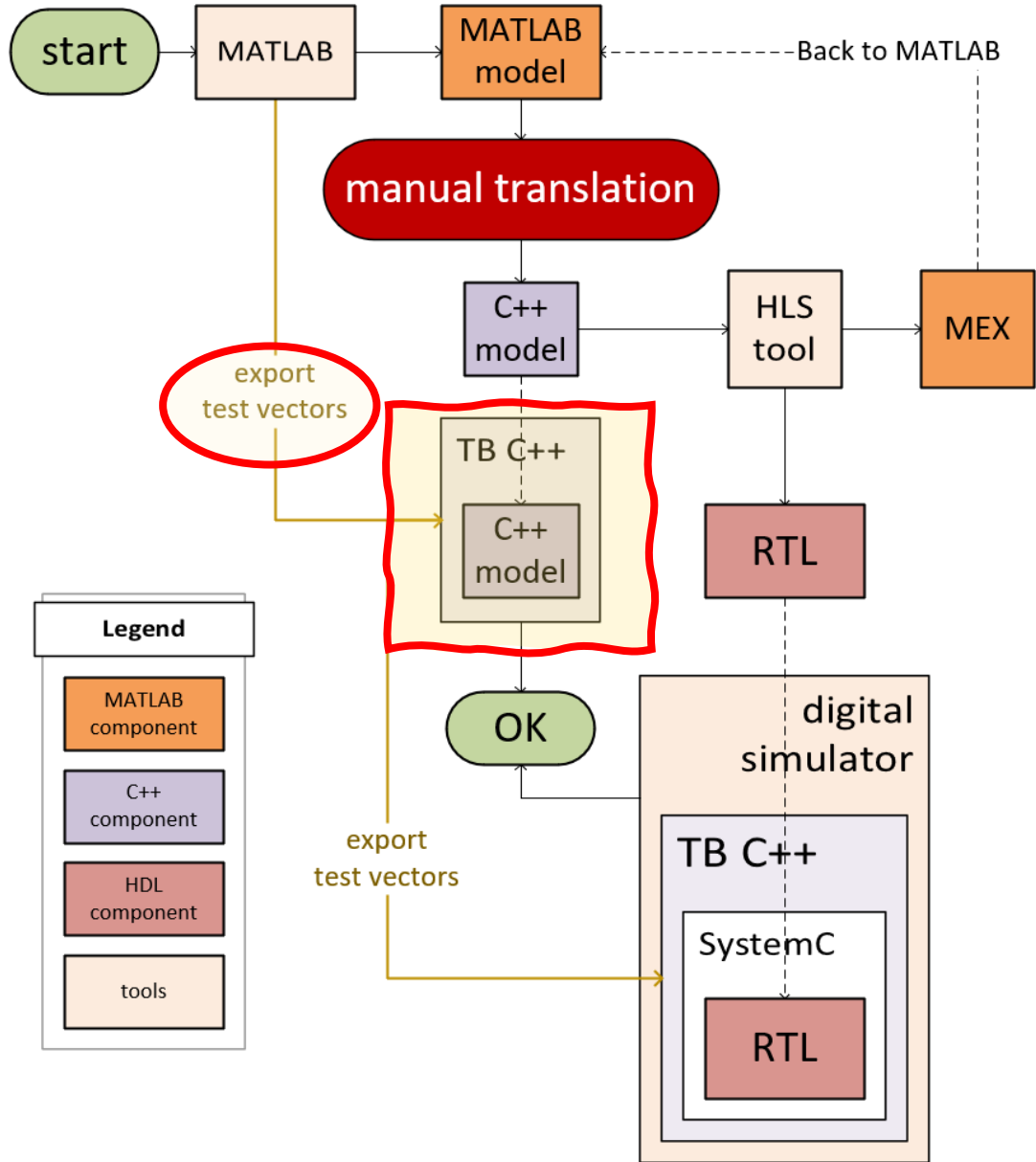
### BUT...

- Connect channels **only** point-to-point
- Direct inputs can be connected point-to-multipoint
- Array of channels are not synthesizable for Catapult
- only structural → intelligence **only in the leaves**

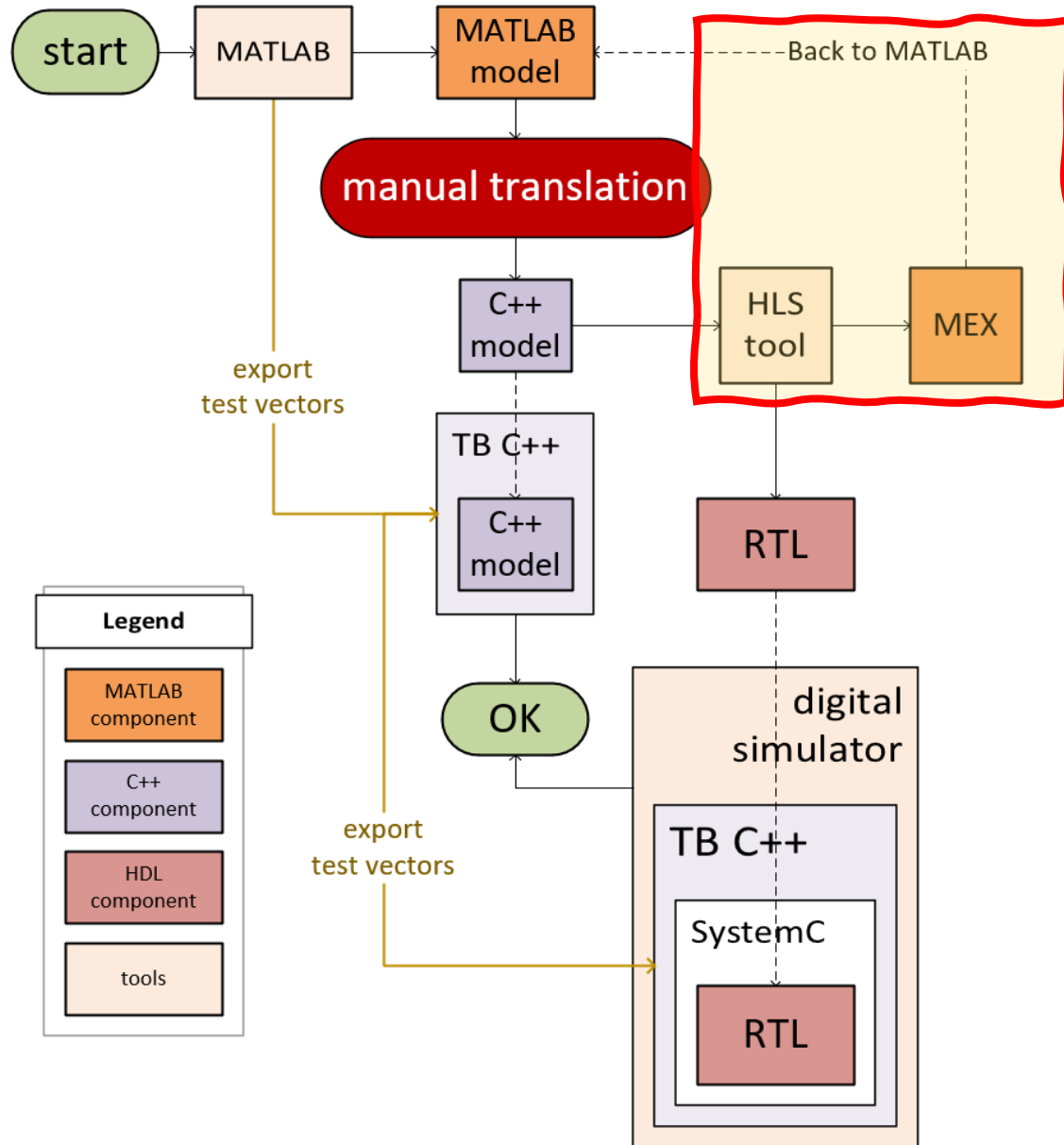


# Verification

# C++ test bench



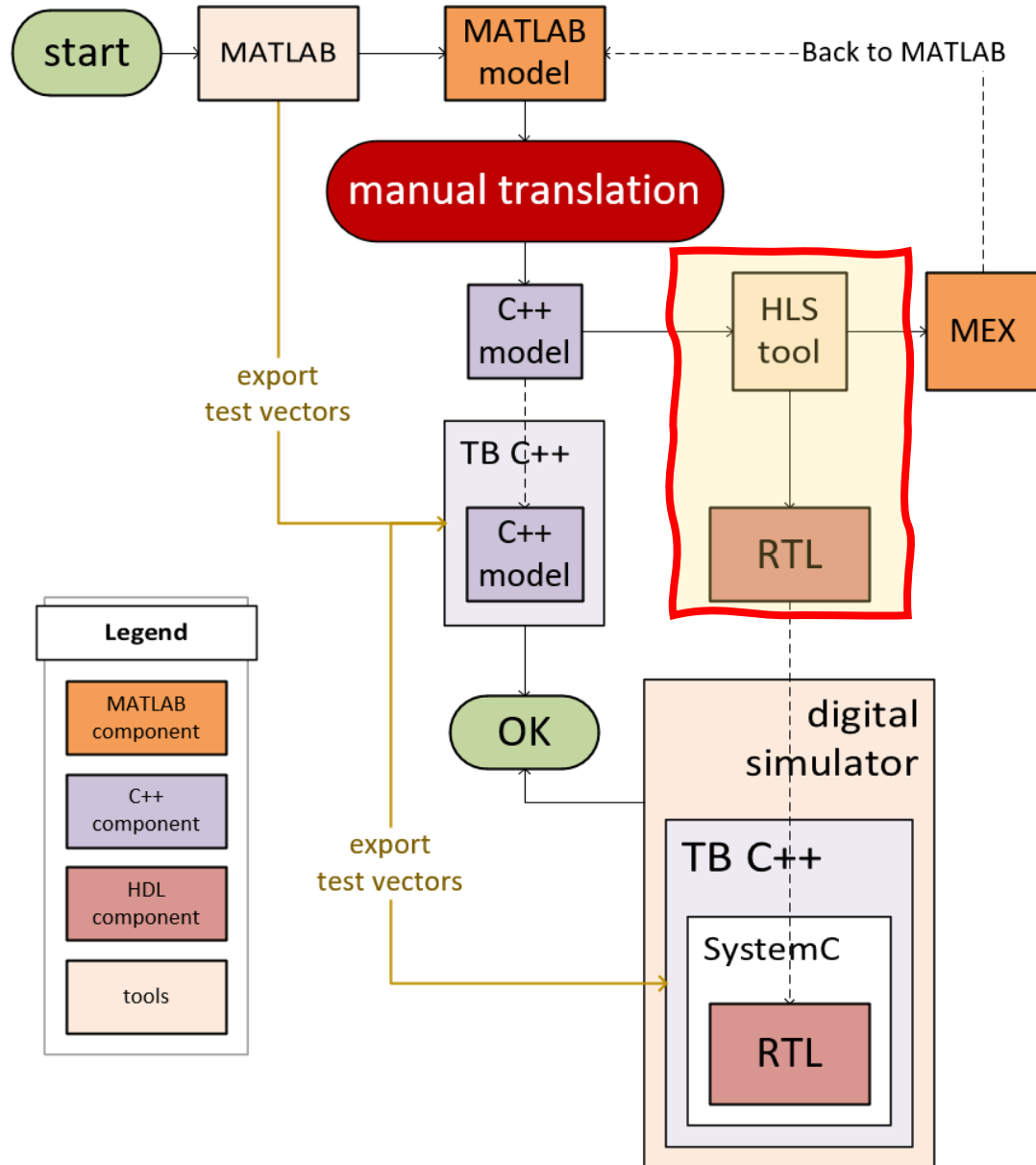
# The MEX



- Feedback **match**
- Fast creation step
- Development speed up
- Exchange

# Extraction

# The extraction



This is the main step of this flow.

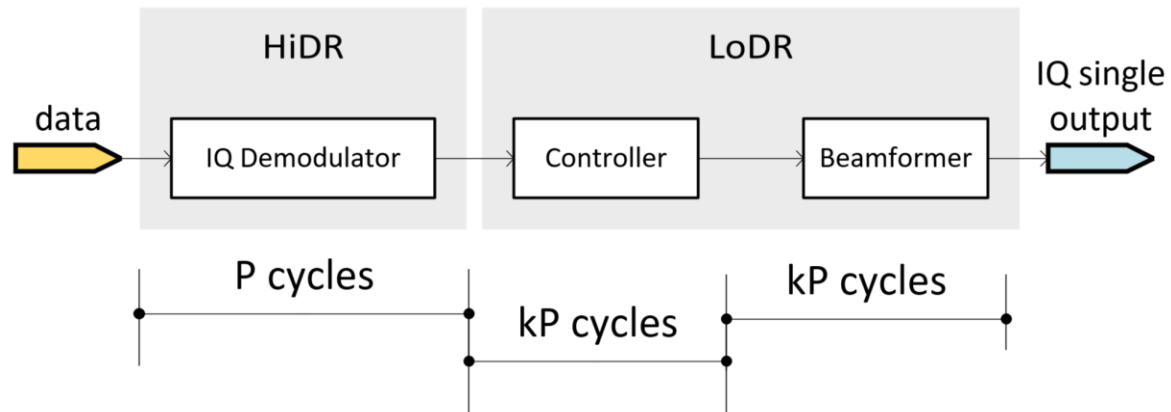
## Preparation step

- ❑ tech-libs into an HLS-library (**library builder**).

## Extraction step

- ❑ HLS tool (**catapult**)
- ❑ drive the extraction either with the GUI of the tool or with a TCL script.

# The extraction



## Constraints

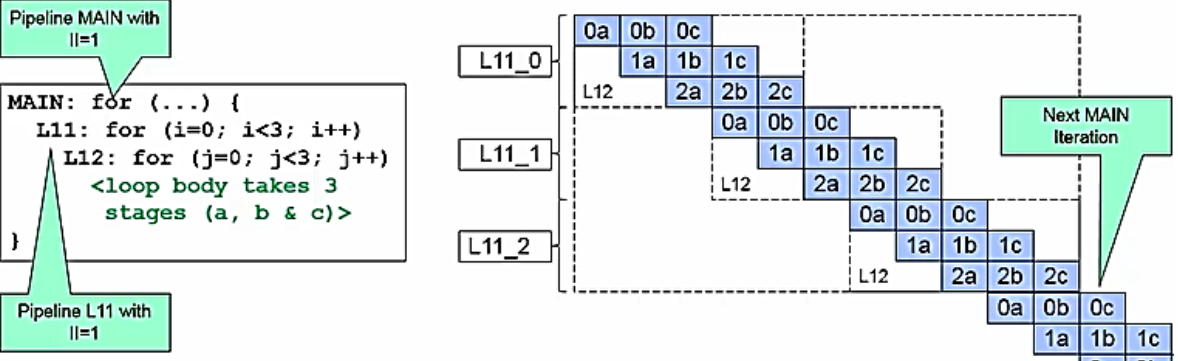
- Clock
  - frequency
  - Duty cycle
- Pipeline initiation interval (P, kP, kP)
- Max Latency

## Architectural settings

- Unrolling
- Pipelining
- Clustering

# Pipelining

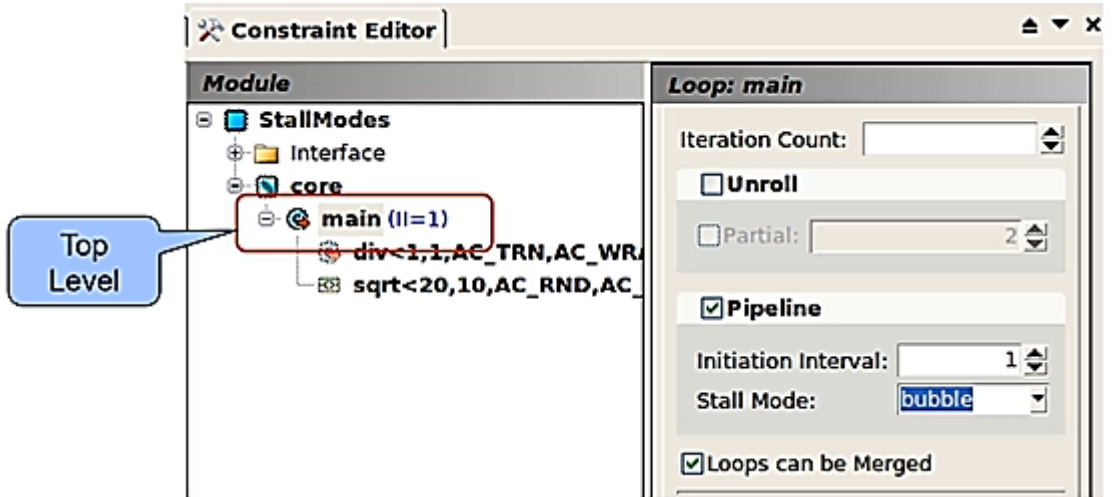
Pipelining is a technique which defines the number of clock cycles between 2 iterations of a loop. This is a way to control the I/O throughput.



Operationally:

- You can apply incrementally some pipelining and see how the extraction goes.

HLS provides the possibility to explore different implementations real time



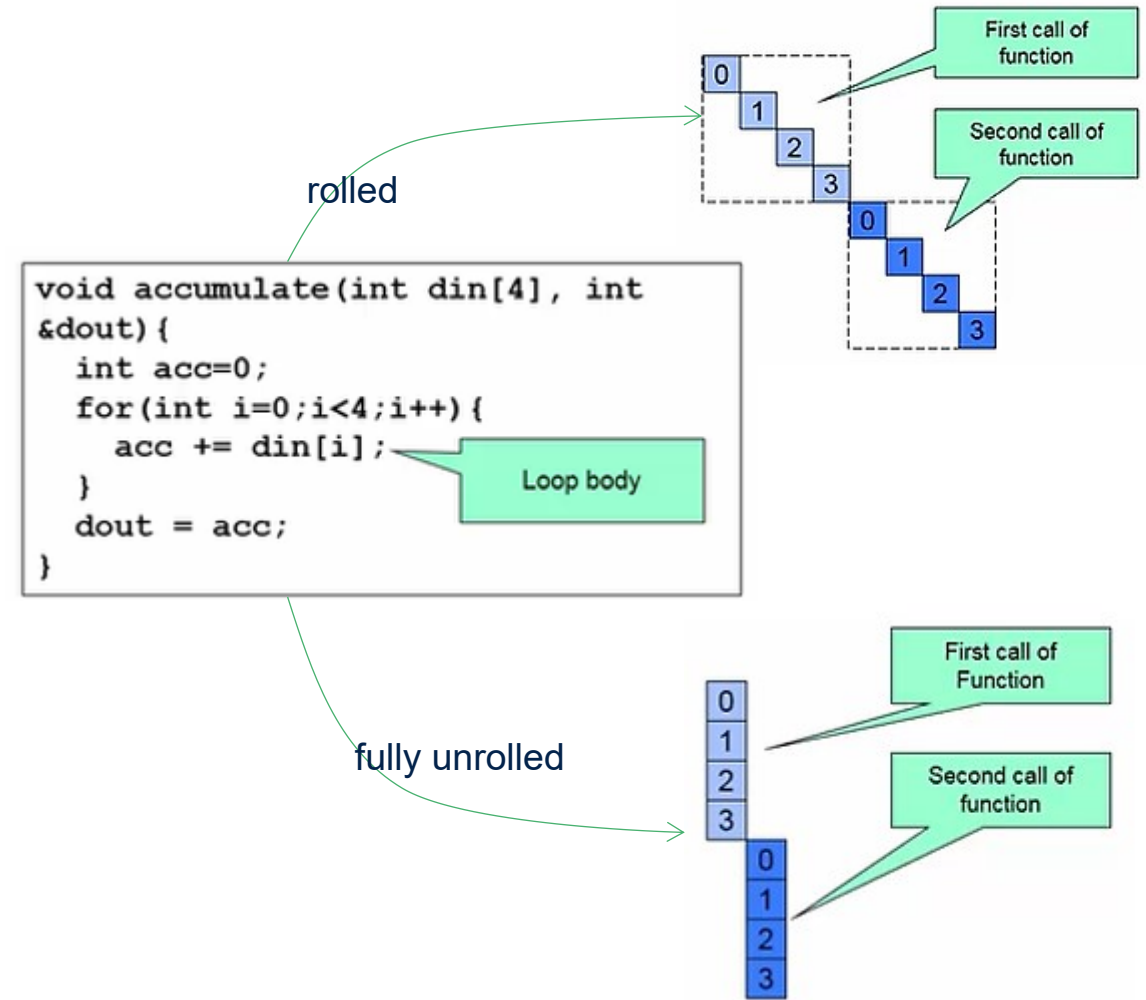
# Unrolling

Loop unrolling allows the execution of multiple loop iterations in parallel. This is a way to control the parallelism.

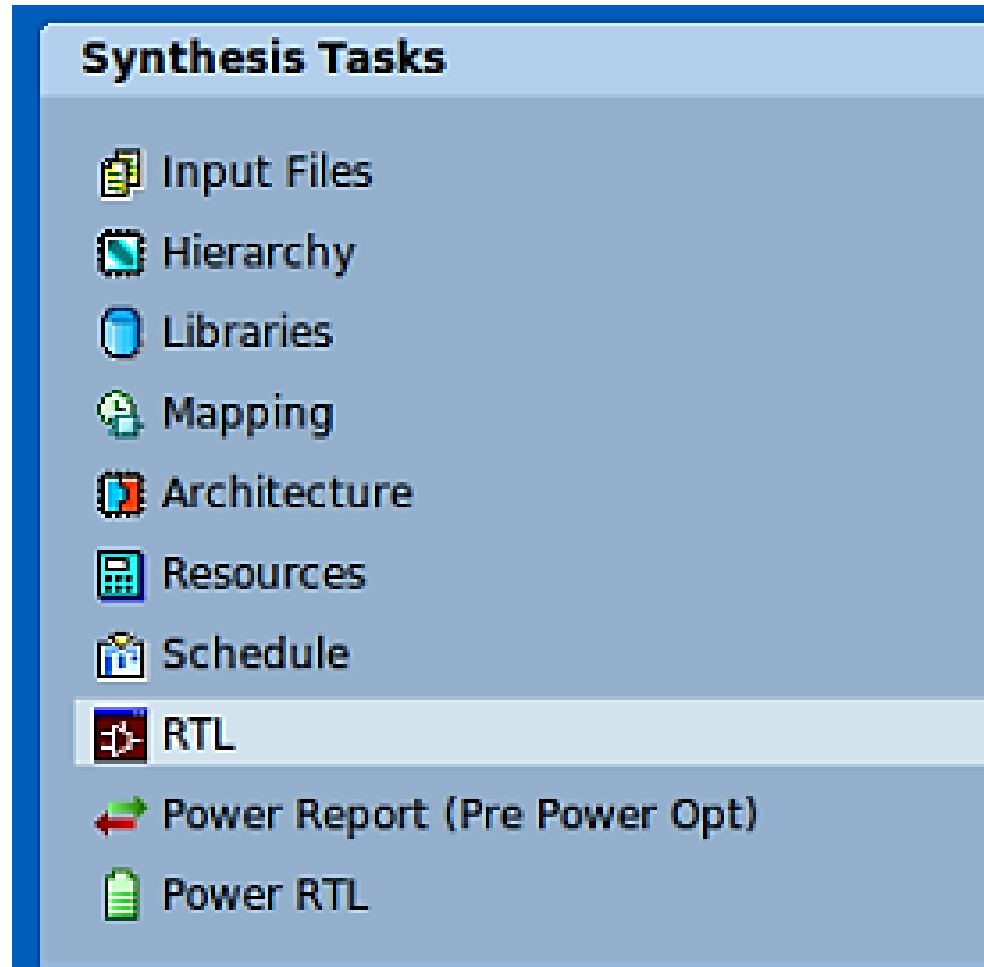
## Rules

- Unroll the inner loops
- Unroll the smallest loops first
- Unroll the independent loops first. Independent loops are, for example, those which do not access to RAM
- Do not unroll the loops which contain rolled loops.

HLS provides the possibility to explore different implementations real time



# Design exploration



Internal steps for the extraction:

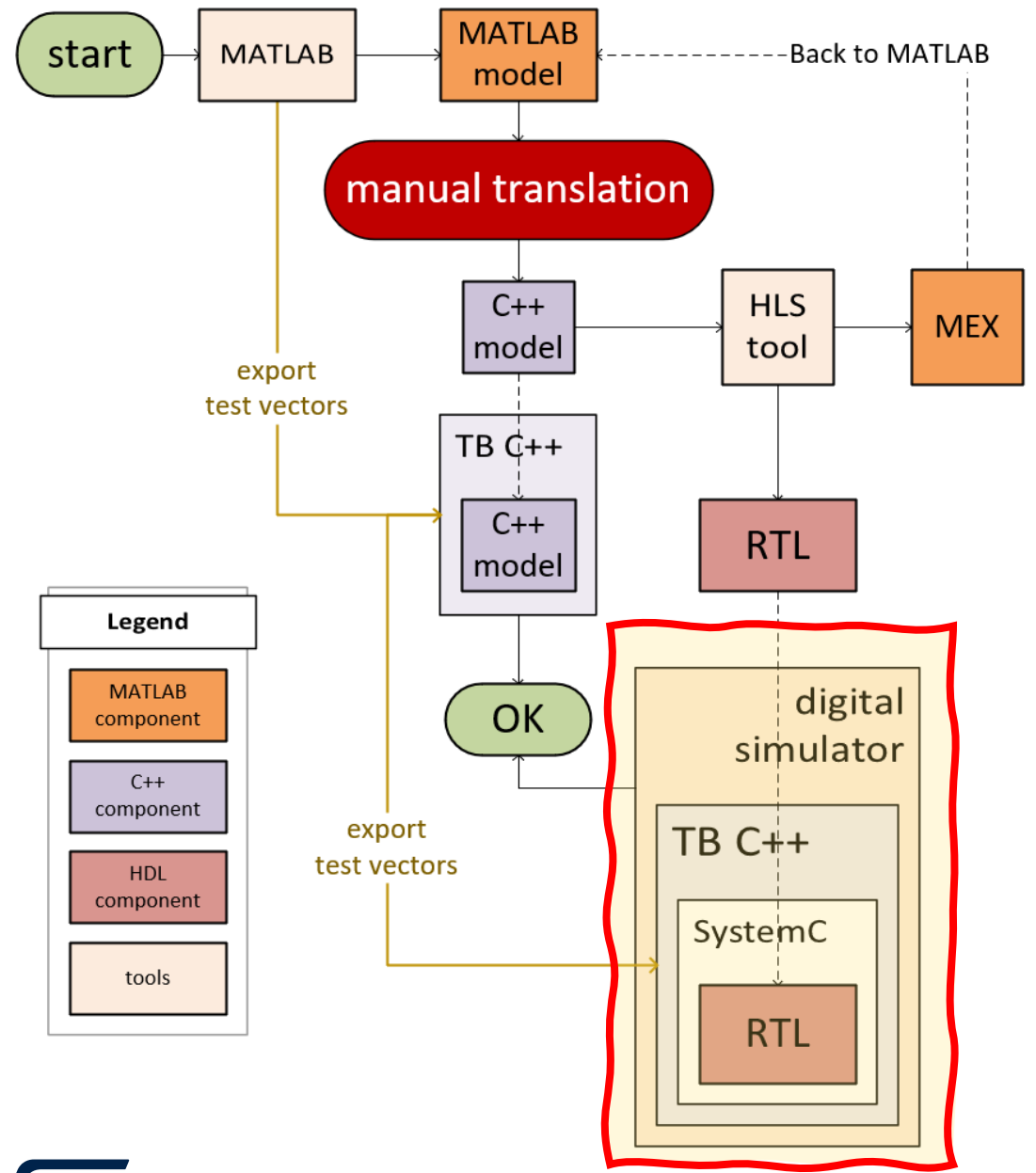
You can step back and forth seeking for the best implementation.

Real time tuning of the scripts

# Validation



Validation



# Digital simulator

RTL **match** MATLAB

(SystemC)

Legend
MATLAB component
C++ component
HDL component
tools

# Results

## Catapult results

- Frequency met
- Input pipeline Initiation interval P
- Latency (tens of cycles)
- RTL area estimation several  $mm^2$ .

## Synthesis with Design Compiler

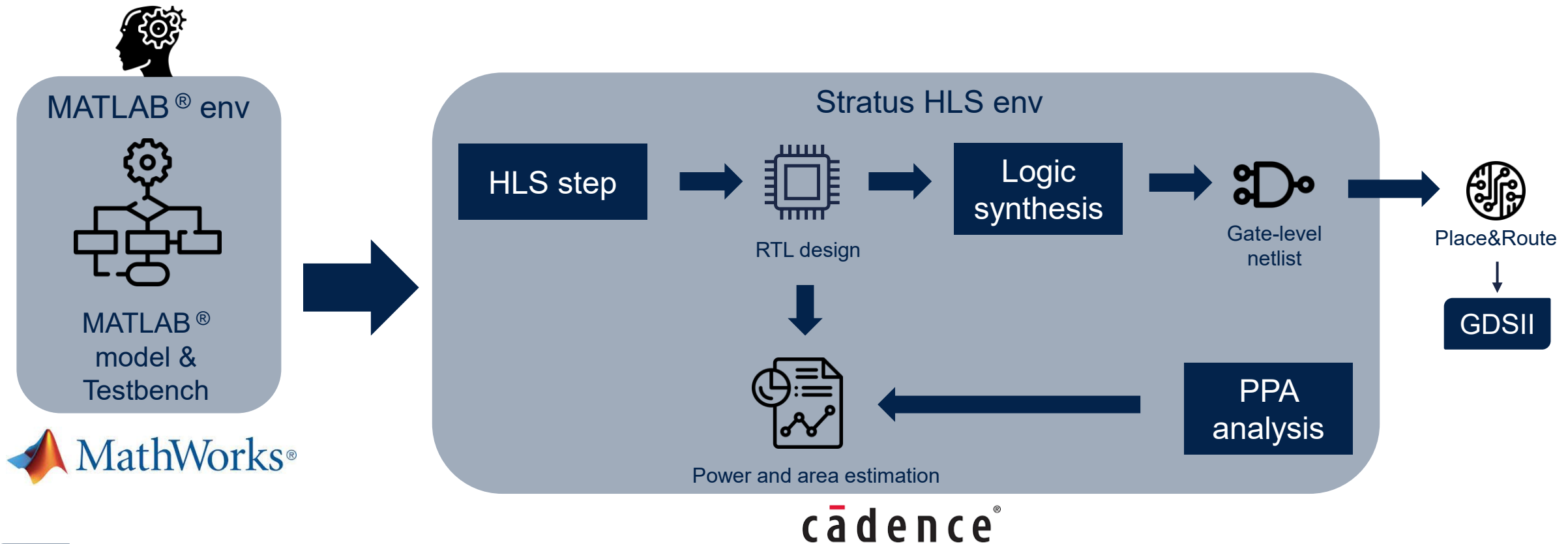
- Frequency met in target technology node
- Area: several  $mm^2$ .

Good correlation → 5% of error

# Stratus HLS experience

# Possible alternative flows?

Currently, the **only possibility** in EDA industry to use a MATLAB® model as input behavioral description for HLS implementation is to use Stratus HLS from Cadence



# Our experiences

## Speech Recognition & Machine Learning



- DNN-based Keyword Spotting System (**KWS**)

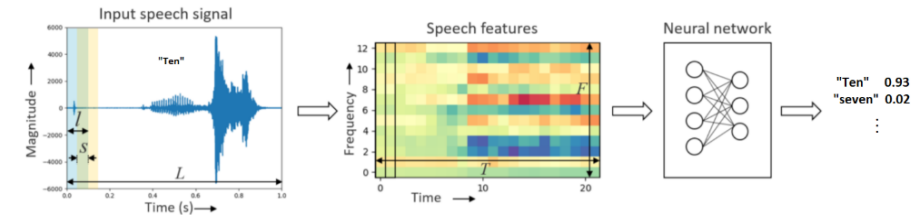
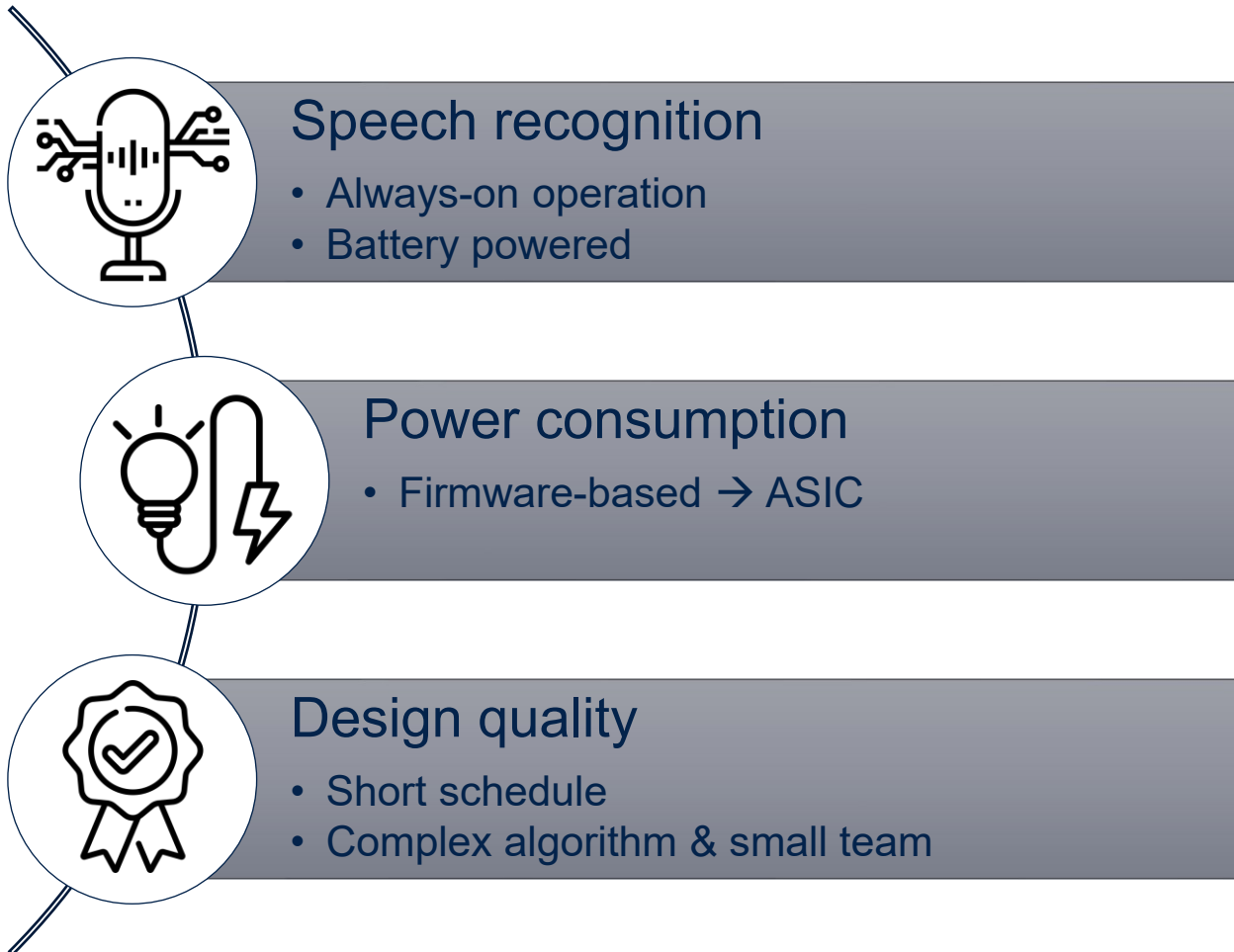
## Ultrasound portable application



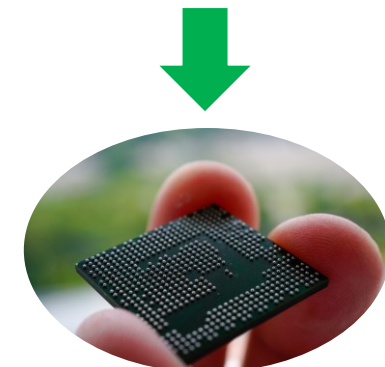
- RX BaseBand Beamformer
  - Filtering & downsampling
  - Internal controller (FSM)
  - Final beamforming

# DNN-based Keyword Spotting System

# DNN-based Keyword Spotting System (KWS)



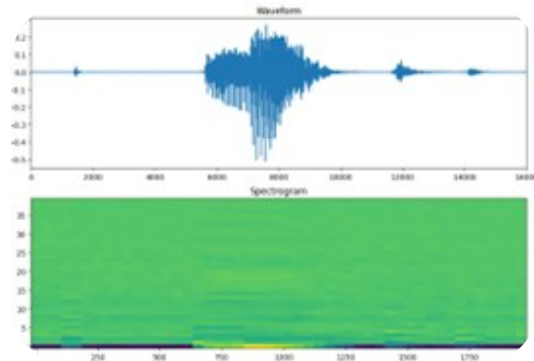
**Firmware-based solution**



ASIC solution  
**HLS design-based**

# KWS – block diagram

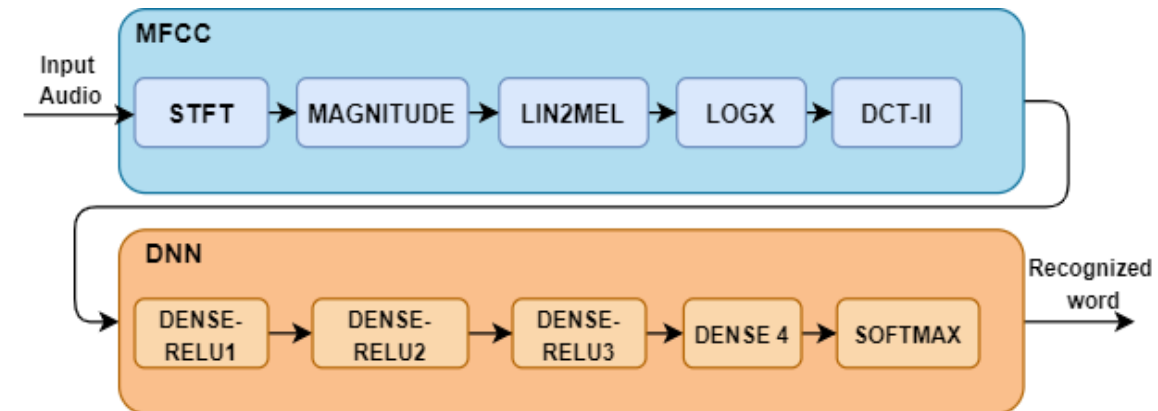
The architecture comprises two main blocks: Mel Frequency Cepstral Coefficients (**MFCC**) block and a Deep Neural Network (**DNN**).



2D Spectrogram



DNN confusion matrix



# KWS – HLS optimizations

## Optimizing loops

- *HLS\_PIPELINE\_LOOP* & *HLS\_UNROLL* directives
- Higher **throughput**
- Affects area

## Controlling scheduling

- *HLS\_CONSTRAIN\_LATENCY* directive
- Define **latency constraints** for specific parts of SystemC code (such as loop or if/else)
- Achieve shortest possible latency

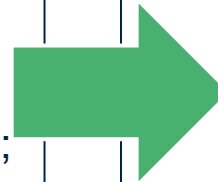
## Reducing power consumption

- Stratus HLS Low power settings
- Add **enable** to some RTL parts to insert CG cells.
- FSMs **optimization**
- Insert **disable** logic for memories when not accessed

# KWS – HLS optimizations

## STFT SystemC code

```
read_loop:
  for ( int j=0; j<stft_config::stft_out; j++ ) { //513
    for ( int k=0; k < stft_config::frame_length; k++ ) {
//640
      data_in =
ibuffer967_fft_in[k+(stft_config::frame_step*i)];
      assign_int_to_float(data_in, data_in_float);
      stft_func(j,data_in_float, k);
    }
  }
....
....
```



## STFT SystemC code + optimizations

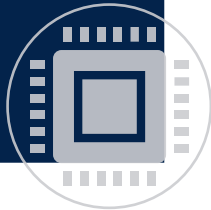
```
read_loop:
  for ( int j=0; j<stft_config::stft_out; j++ ) { //513
    for ( int k=0; k < stft_config::frame_length; k++ ) { //640
      #ifdef __PIPE_ON__
        HLS_PIPELINE_LOOP( .... );
      #endif
      #if defined(LAT)
        HLS_CONSTRAINT_LATENCY( ... );
      #endif
      data_in = ibuffer967_fft_in[k+(stft_config::frame_step*i)];
      assign_int_to_float(data_in, data_in_float);
      stft_func(j,data_in_float, k);
    }
  }
```

The **directives** can be inserted in **key** points of the design, allowing the HLS tool to optimize the generated RTL based according to the desired constraints

# KWS – HLS environment

- **Easy process** to select your target library

Technology  
library



- Create and synthesize all your design **“versions”**

Design  
exploration



- In the same environment you can:
  - Run RTL/GL logic **sims**
  - Run logic **synthesis**
  - Run **power** estimations

Integrated  
environment

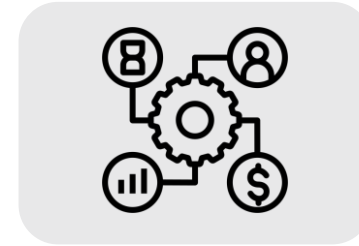


# KWS – HLS environment

**Power estimation** for each design version



- Power Configs
  - PGXG\_BASIC\_TOP\_1
  - PGXG\_BASIC\_TOP\_LP\_1
  - PGXG\_OPT\_TOP\_1
  - PGXG\_OPT\_TOP\_LAT\_BEST\_1
  - PGXG\_OPT\_TOP\_SHORT\_1
  - PGXG\_TOP\_NETLIST\_RTL\_BASIC\_1



- dct
  - BASIC\_1
  - BASIC\_LP\_1
  - BASIC\_2
  - BASIC\_LP\_2
  - BASIC\_3
  - BASIC\_LP\_3
  - BASIC\_4
  - BASIC\_LP\_4

**RTL** for each design version

**Netlist** for each design version



- Logic Synth Configs
  - G\_TOP\_opt\_1
  - G\_TOP\_basic\_1
  - G\_TOP\_basic\_LP\_1
  - G\_TOP\_opt\_2
  - G\_TOP\_basic\_2
  - G\_TOP\_basic\_LP\_2
  - G\_TOP\_opt\_3
  - G\_TOP\_basic\_3
  - G\_TOP\_basic\_LP\_3
  - G\_TOP\_opt\_4
  - G\_TOP\_basic\_4
  - G TOP basic LP 4

- Sim Configs
  - BASIC\_TOP\_V\_1
  - BASIC\_TOP\_LP\_V\_1
  - OPT\_TOP\_V\_1
  - OPT\_TOP\_BEST\_LAT\_V\_1
  - BASIC\_G\_1
  - OPT\_G\_1
  - BASIC\_TOP\_V\_2
  - BASIC\_TOP\_LP\_V\_2
  - OPT\_TOP\_V\_2
  - OPT\_TOP\_BEST\_LAT\_V\_2

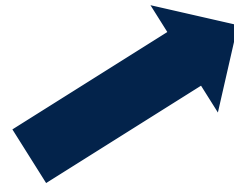
**Digital simulation** for each design version



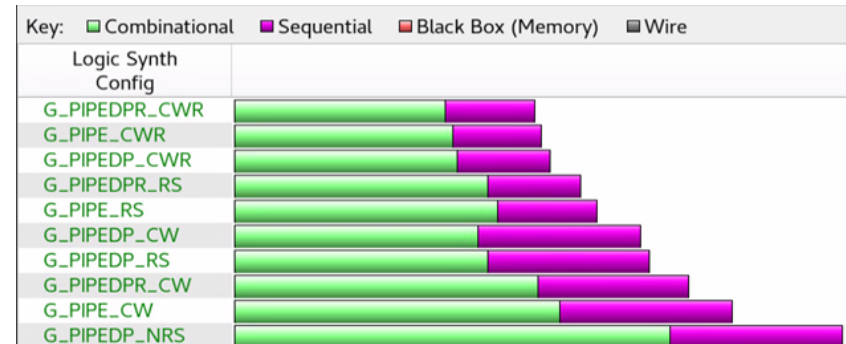
# KWS – HLS exploration

## Design exploration

Design version	Brief description
BASIC	Baseline design
BASIC_LP	Only Low Power optimization enabled
OPT	Low Power optimization enabled ; Constrained latency and pipeline/unrolling enabled on critical blocks



## Area comparison on the GUI



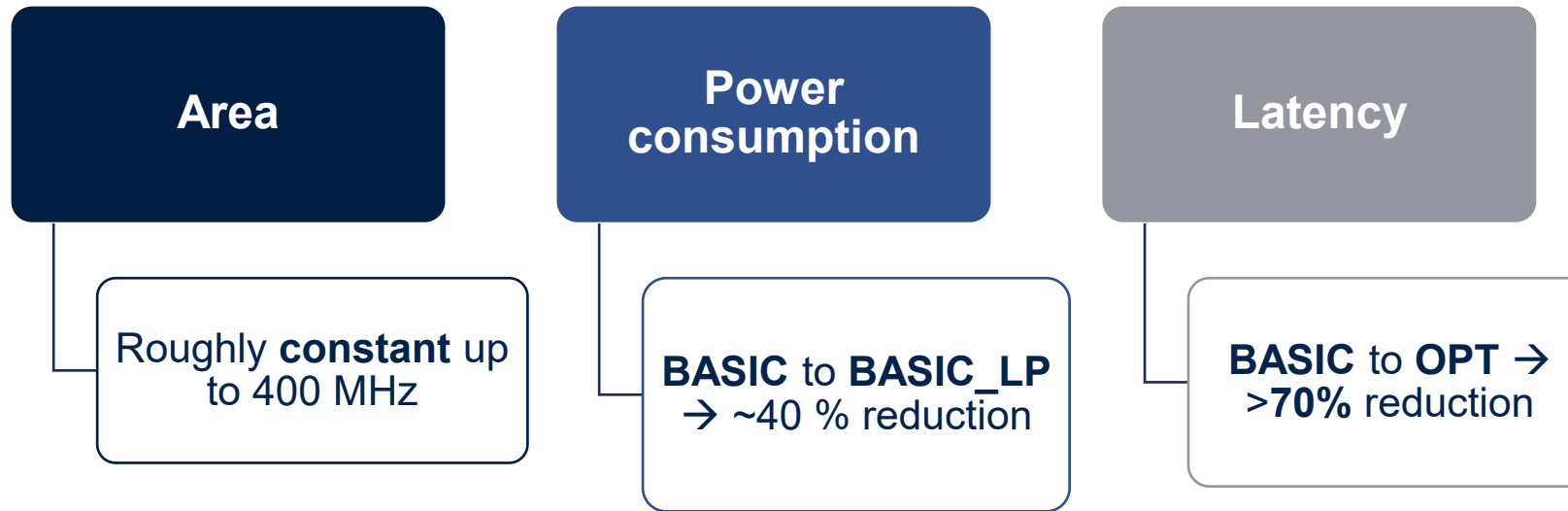
## Power consumption comparison on the GUI

Power Config	Sim Config	Total Power (mW)	Static Power (mW)	Dynamic Power (mW)
PGXG_BASIC_TOP_1	BASIC_TOP_V_1	1645.498	3.132	1642.365
PGXG_BASIC_TOP_2	BASIC_TOP_V_2			
PGXG_BASIC_TOP_3	BASIC_TOP_V_3			
PGXG_BASIC_TOP_4	BASIC_TOP_V_4			
PGXG_BASIC_TOP_LP_1	BASIC_TOP_LP_V_1	1291.116	3.064	1288.053
PGXG_BASIC_TOP_LP_2	BASIC_TOP_LP_V_2			
PGXG_BASIC_TOP_LP_3	BASIC_TOP_LP_V_3			
PGXG_BASIC_TOP_LP_4	BASIC_TOP_LP_V_4			
PGXG_OPT_TOP_1	OPT_TOP_V_1	1609.151	3.108	1606.039

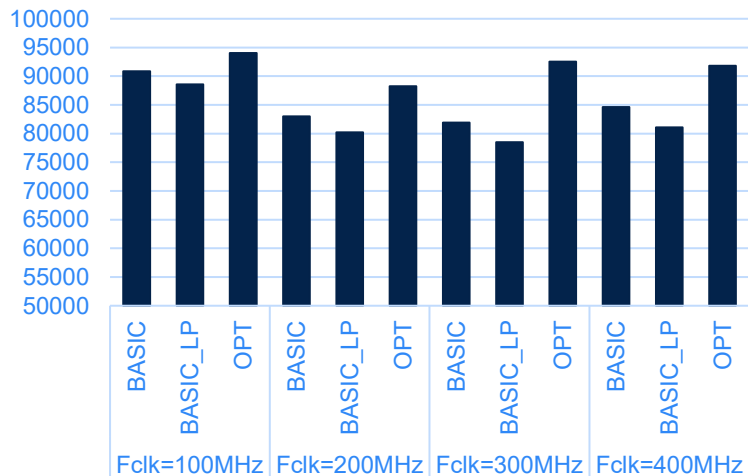
## Parts Library

Name	Part Kind	Area	Comb. Area
<b>Total of 68 items:</b>		<b>16529...</b>	<b>16529.0</b>
logx_cynw_float_mul_E8_M23_4_33	logx_cynw_float_mul_E8_M23_4	3359.5	3359.5
logx_cynw_float_mul_E8_M23_4_187	logx_cynw_float_mul_E8_M23_4	3359.5	3359.5
logx_cynw_float_mul_E8_M23_4_180	logx_cynw_float_mul_E8_M23_4	3359.5	3359.5
logx_cynw_float_add_E8_M23_4_91	logx_cynw_float_add_E8_M23_4	1552.3	1552.3
logx_cynw_float_add_E8_M23_4_31	logx_cynw_float_add_E8_M23_4	1552.3	1552.3
logx_cynw_float_add_E8_M23_4_170	logx_cynw_float_add_E8_M23_4	1552.3	1552.3
logx_cynw_float_div_bottom_4_102	logx_cynw_float_div_bottom_4	357.7	357.7
loax_int_to_cynw_float_4_88	loax_int_to_cynw_float_4	324.2	324.2

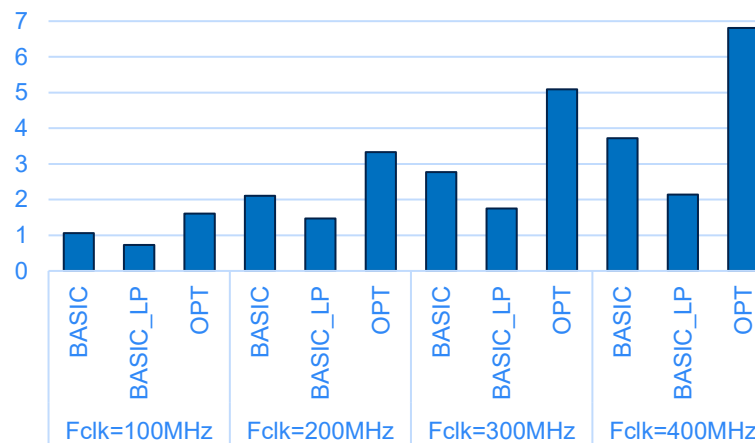
# KWS – HLS results



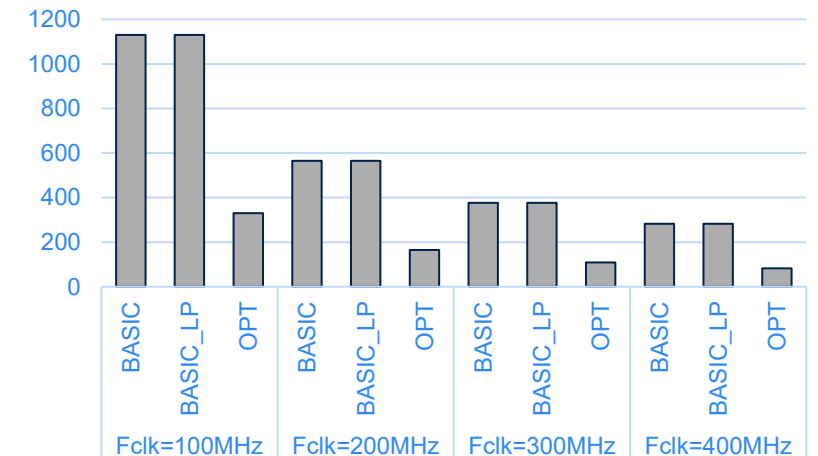
RTL Total Area Estimation [ $\mu\text{m}^2$ ]



RTL Power Estimation [mW]



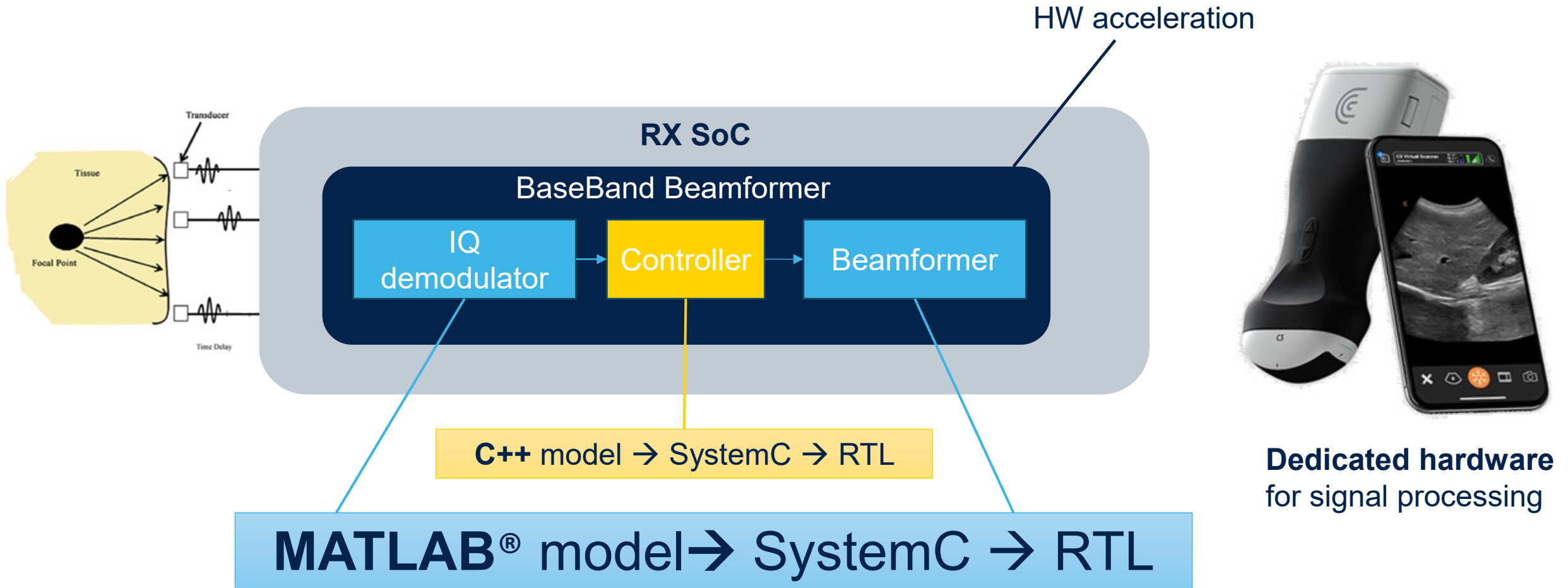
Latency performance [ms]



# Ultrasound portable application

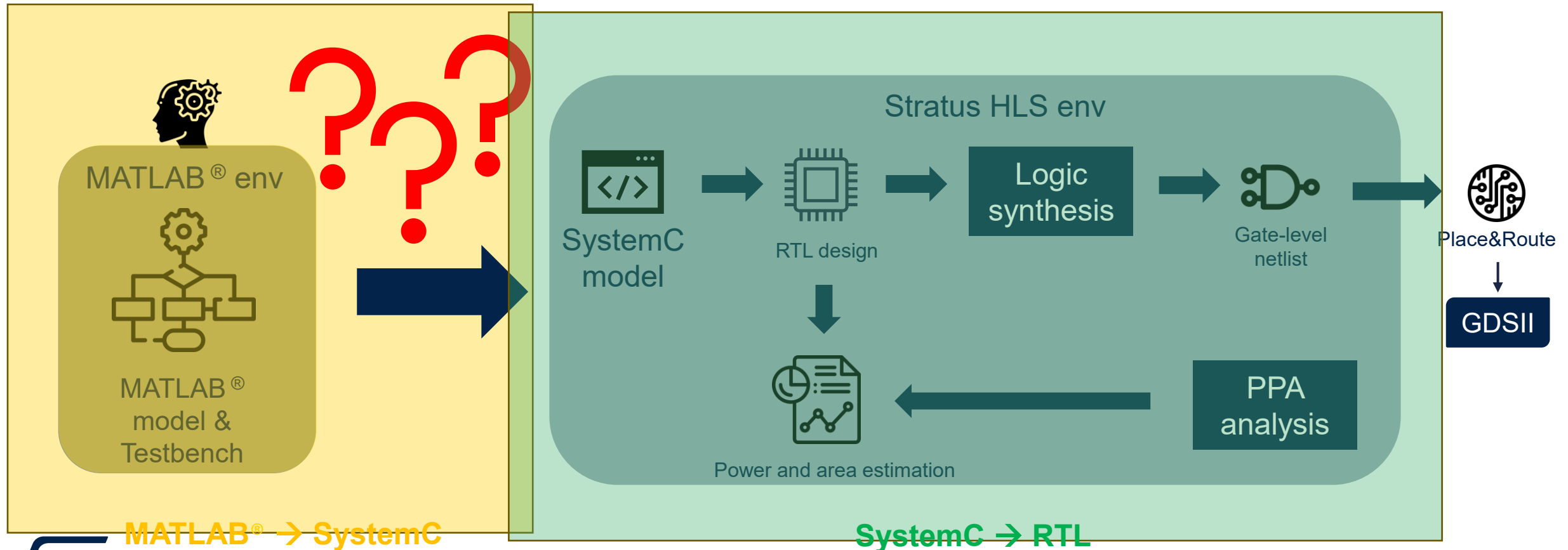


# Ultrasound portable application



# MATLAB® to SystemC: How to proceed ?

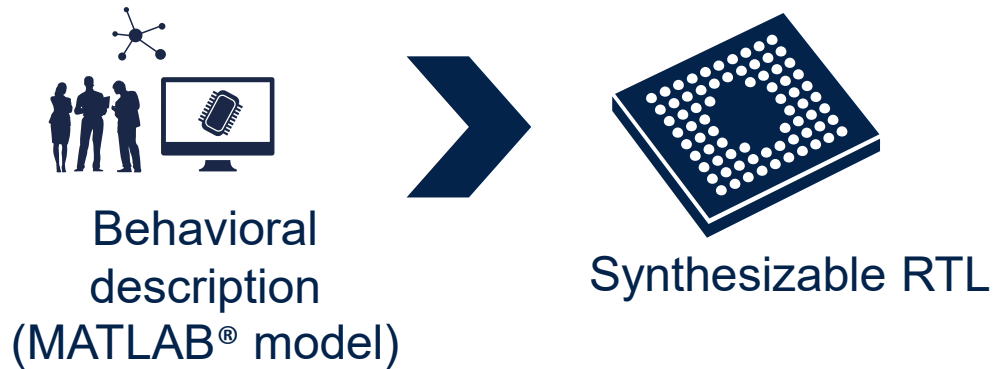
Currently, the **only possibility** in EDA industry to use a MATLAB® model as input behavioral description for HLS implementation is to use Stratus HLS from Cadence



# Intermediate SystemC step

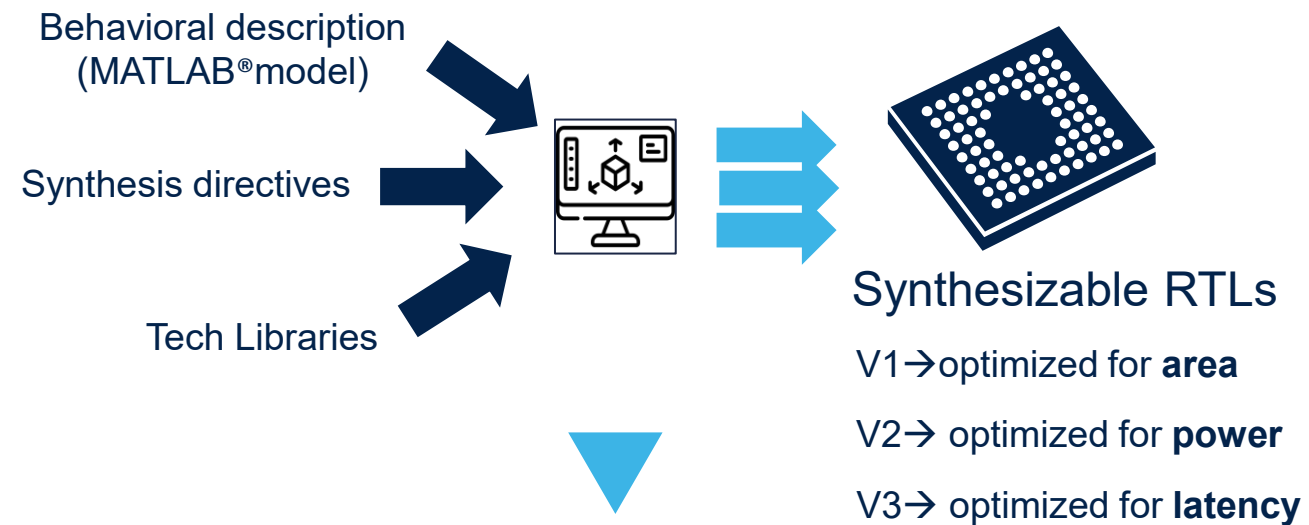
**Question:** MATLAB® had already the possibility to generate RTL ! Why don't we exploit it?

## MATLAB®+ HDL Coder



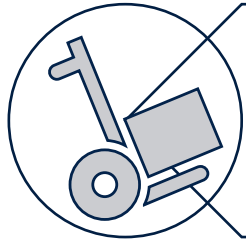
**No traditional HLS flow!**

## MATLAB®+ HDL Coder + Stratus HLS



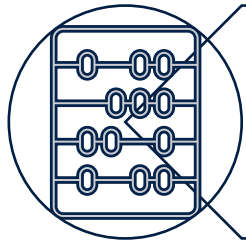
**Classical HLS flow!**

# IQ demodulator



## Very large IP

- Slow RTL generation (several hours)



## Modular structured

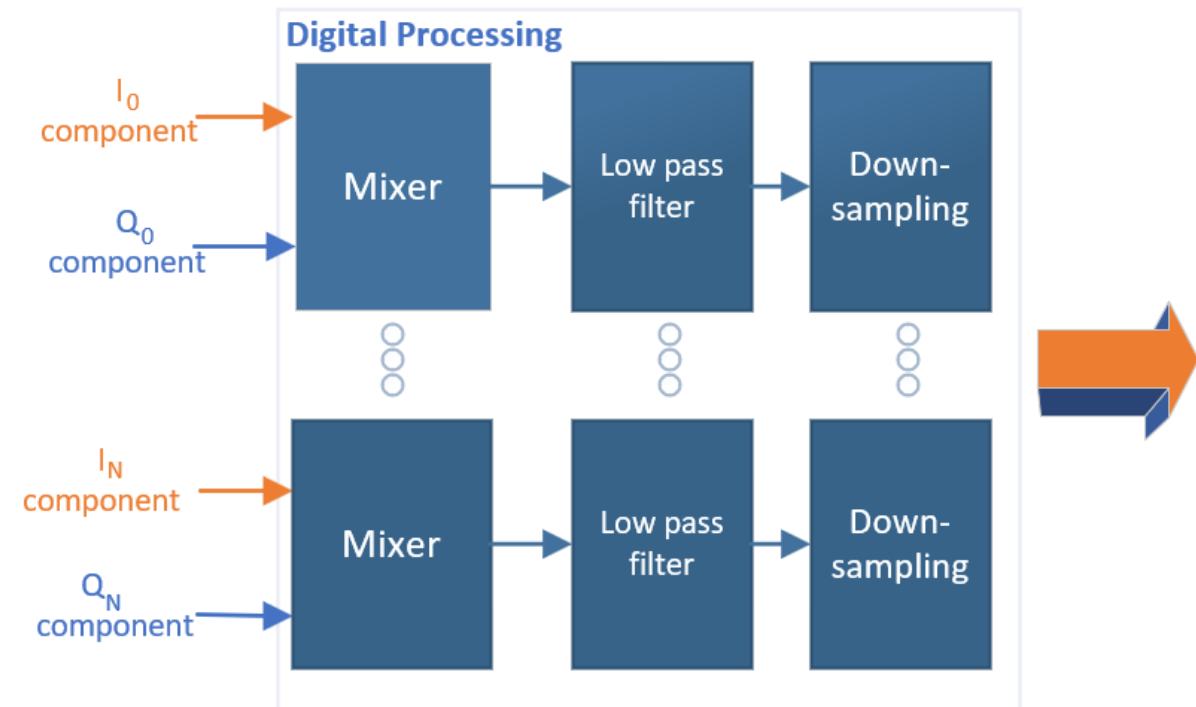
- N-channels
- Synthesized all-in-one



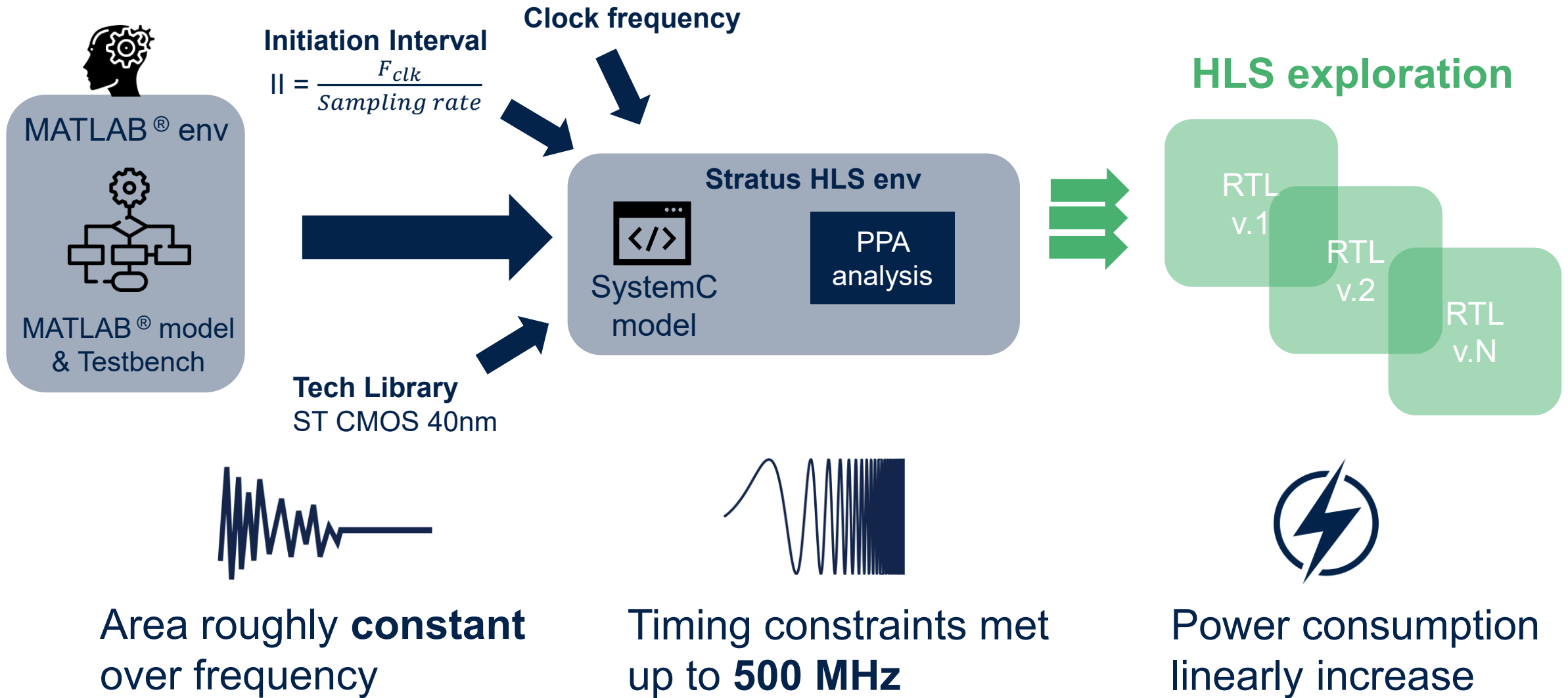
## Simple operators

- No impact on original MATLAB model

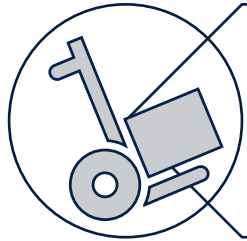
## IQ Demodulator



# IQ demodulator – HLS result

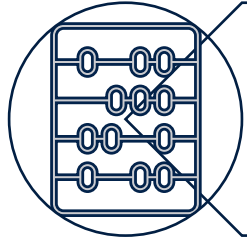


# Beamformer



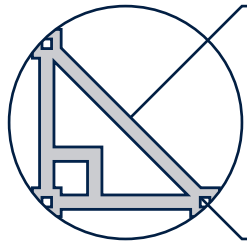
## Very large IP

- Slow RTL generation (several hours)



## Modular structured

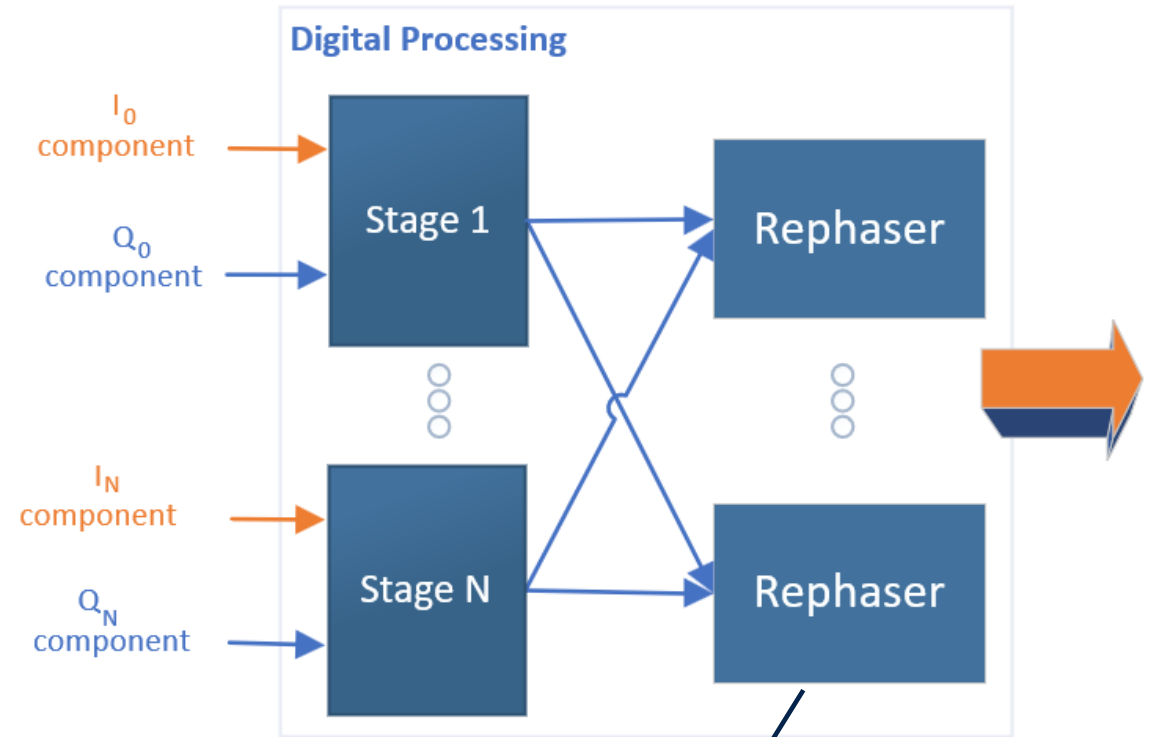
- N-channels
- Synthesized all-in-one



## Trigonometric operators

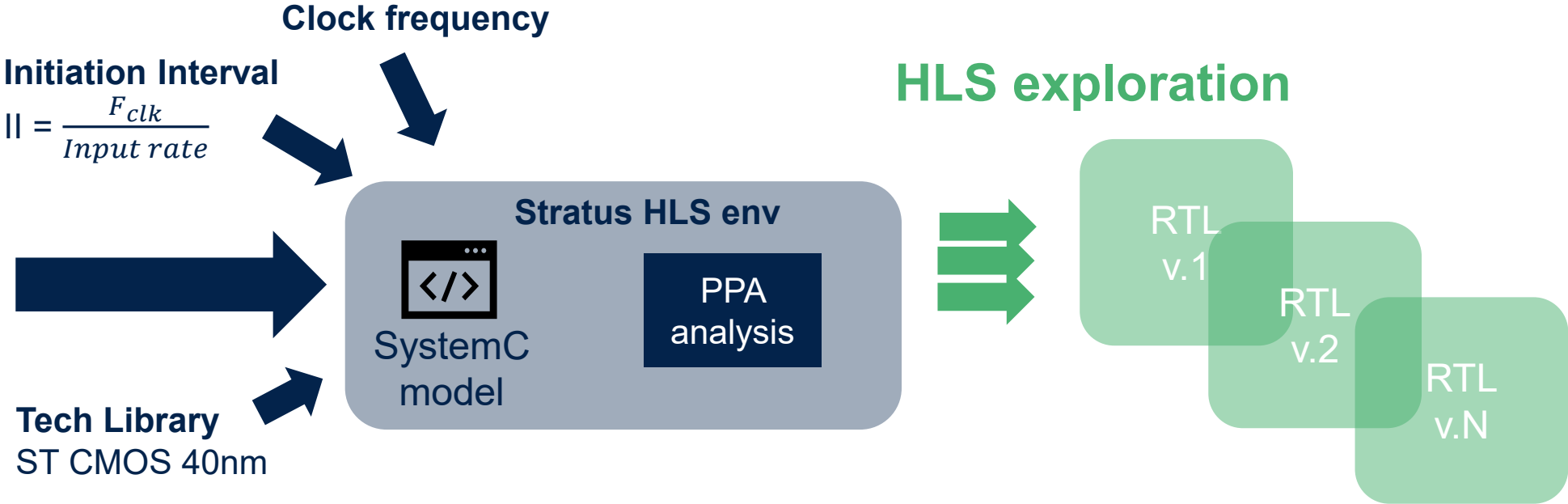
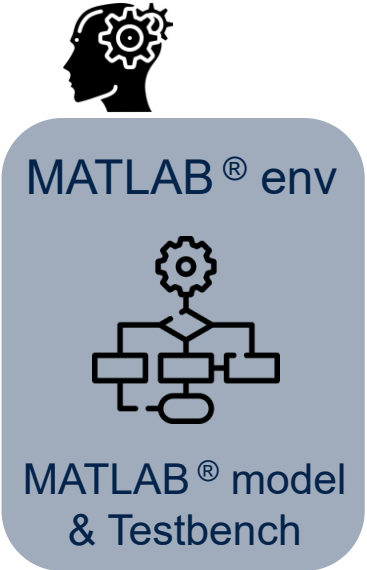
- They shall be described within MATLAB®

You **must** provide an “hardware” model for **sin** and **cos** in your MATLAB® code!



$$x \cdot (\cos(\alpha) \cdot \sin(\alpha))$$

# Beamformer – HLS result



Area even **decreasing** over frequency



Timing constraints met up to **300 MHz**

# “Synthesizable” MATLAB model

1

It shall be cycle-accurate

2

Loops (if any) must include a fixed number of iterations

3

Be very careful about dimension of array/matrix used

4

Use left/right shift operators to describe multiplication/divisions by power of two

5

Any non-standard operator must be explicitly modeled in your MATLAB code

6

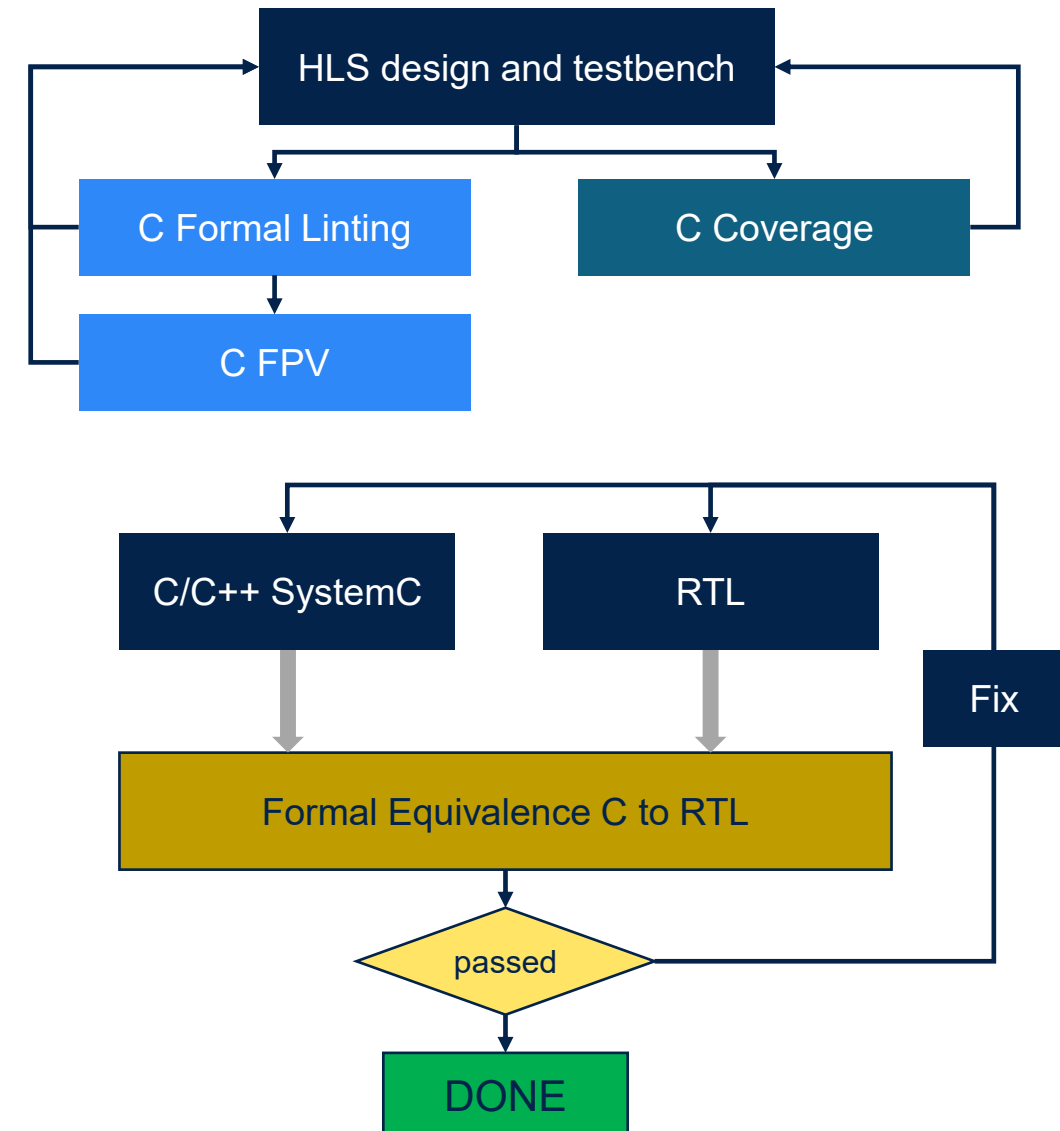
Quantization step is crucial. Do not trust only the suggestions within HDL Coder quantization step.

More in general....**thinks** like a **hardware engineer** during **MATLAB modeling!**

# Conclusions

# What is coming?

- The source code is the C/C++ and SystemC
  - The RTL is the output, not the entry point!
- The source code must be verified deeply
  - The major EDA vendor is going to develop tools for:
    - C/C++ SystemC coverage
    - C/C++ SystemC Formal linting
    - C/C++ SystemC Formal Property Verification
- One of the most need tool is the Logical Equivalence Check between C/C++ SystemC and RTL

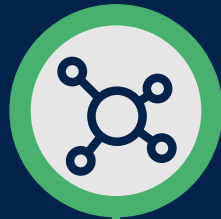


# Conclusions

**Drastic time reduction**  
compared to manual flow;  
Integrates with **MATLAB®**



**Power-  
Performance-Area  
exploration easy**



**Easy reuse of  
MATLAB®/C  
testbench at RTL  
level**



**Learning curve;  
requires training  
and new design  
perspective**



**Poor or missing  
formal tools**



# Contacts



**David Vincenzoni**

**STMicroelectronics**

Design Manager  
Sr. Member of Technical Staff

[david.vincenzoni@st.com](mailto:david.vincenzoni@st.com) ✉



**Gianluca Rigano**

**STMicroelectronics**

Sr. Design Engineer

[gianluca.rigano@st.com](mailto:gianluca.rigano@st.com) ✉



**Marcello Dusini**

**STMicroelectronics**

Sr. IC Designer

[marcello.dusini@st.com](mailto:marcello.dusini@st.com) ✉

# Questions & Answers

Q&A

