# $M_3$ Reliable Signal Processing Datapaths Design Using Control Techniques Based on Difference Equation Models (RESPECT-DEM) [final review]

Oana Boncalo

University Politehnica Timisoara

*oana.boncalo@cs.upt.ro*

November 25, 2022





## Activity overview & General objectives

- Activity name: Reliable Signal Processing Datapaths Design Using Control Techniques Based on Difference Equation Models
- Main objective: Iterative Correction loops using control theory & optimizations
- Duration: 18 months
- Target TRL: 2
- Total project budget:101,200 € ($M_3$ - 34 K)

# Deliverable list summary for $M_3$

| Doc ID | Title | Milestone | Description of documents |
|---|---|---|---|
| **D**$_3$ | Report on process automation and design optimizations and evaluations of the complex DSP pipeline with correction. | MS$_3$ | It presents the cost and reliability assessment for the FPGA implementation of the 512 point FFT use-case. Furthermore, aspects related to as the partial automation of the process will be discussed. |
| **SW**$_3$ | Optimized Matlab and HDL models for a fault tolerant complex DSP use-case. | MS$_3$. | It will include the optimized Matlab and Verilog HDL models |





# Meetings and coordination

- GIT repository
- Weekly meetings: Friday 10:00 AM - HW & Verification, and on-demand.
- other: email & Wattsup
- frequency of unexpected events increased – personnel changes due to job opportunities (Stein Daian, master graduate left the team in September)

## Key personnel & qualifications

- Dr. Oana Boncalo (lead) - digital design, reliability, information theory, ECC, optimizations
- Dr Mircea Bogdan Radac (senior researcher) - control theory, optimization, neural network
  left the team, end of January, 2022.
- Dr. Alexandru Amaricai (senior researcher) - digital design and verification, fault tolerant digital design
  left the team
- Daian Stein (junior researcher) - main focus verification and integration activities left the team, August, 2022





## Summary of contributions

- Optimizations & TestBench:
  - refactor TestBench:
    1. Tasks and functions re-factoring, timeout counter threads for avoiding the simulation "hanging" in case DUT synchronization event does not arrive;
    2. Error injection before encoding;
  - Bug fix;
  - Checkpointing and execution for datapath buterfly outputs missmatch at the end of stage execution;
- Partial automation HLS modeling:
  - FT-memory block (syndrome/encoding) Matlab HLS modeling
  - Gradient descent iterative symbol update design and verification
  - Cost analysis based on synthesis estimates;

## Committed resources

- Resource spending budgeted to project is in line
- Overall project spending exceeds budget
  - + Effort on HDL optimization and Testbench refactoring;
  - + Effort for HLS modeling and LDPC and FFT parameter propagation throughout the model.
    1. parameters are constants
    2. automatic fixed point precision tweaking
    3. Matlab hardware aware modeling (FSM, memory design, bit operations, architecture)
  - + Most of the HDL model optimizations have already been presented during the second review meeting
- The additional effort makes possible the reaching $M_3$ – May, 2022 – without major deviation





## Meeting agenda

- 9:30-9:45 Introduction & Objectives M3 brief review
- 9:45-10:15 Optimizations of the HDL model
- 10:15-10:30 Demonstrator - TB & HDL model
- 10:30-10:45 Break
- 10:45-11:15 Automatic Verilog HDL model generation using Matlab HLS
- 11:15-11:30 Demonstrator - Matlab HLS - GDSU
- 11:30-11:50 RIDs (check ESA comments on the deliverable/s)(PO)
- 11:50-12:00 Meeting Closing

# Optimizations

Oana Boncalo

University Politehnica Timisoara

*oana.boncalo@cs.upt.ro*

November 25, 2022



## Overview

## Milestone 2 – solved problems

Fault tolerant architecture & implementation & optimizations:

- The problem of computing the auxiliary symbols efficiently;
- Memory organization of redundant information, as well as accesses non-trivial – target –minimum storage overhead for correction loop and redundant symbol data;
- The computation of the redundant data has been selected to match that of the FFT data, since the latter is well studied and optimized in literature;
- Integration of the correction loop in the iterative memory based FFT architecture – modular, concurrent execution the syndrome computation and redundant data processing;
- The Encoder vulnerability;
- Testbench refactoring;



## Milestone 3 - ToDo list

Achievements presented:

- 512 point FT FFT architecture with optimizations evaluated & demonstrator;
- optimizations: redundant data storage, code selection, on-the-fly syndrome computation, and iterative gradient correction loop implementation that peforms all-zero codeword decoding for LDPC real codes;

The research work planned for the next period requires the following actions:

- Optimization 1: precision tweaking using Matlab.This is an important source of cost saving.
- Optimization 2: improving fault tolerance by addressing the vulnerability of an error manifesting between the redundant part PE output and the next stage encoder block.
- Explore modeling in Matlab/C for high level synthesis code generation of the correction loop and/or fault-tolerant computation block using HLS.

## HW Architecture Characteristics

HW Architecture characteristics:

- memory based (registers, FFs, latches, SRAM);
- computation consists of several iterations;
- execution within an iteration is not parallel (i.e. a sub-set of the input data is processed at a time);
- memory dominates the overall architecture cost;
- memory data is overwritten at the end of a clock-cycle;



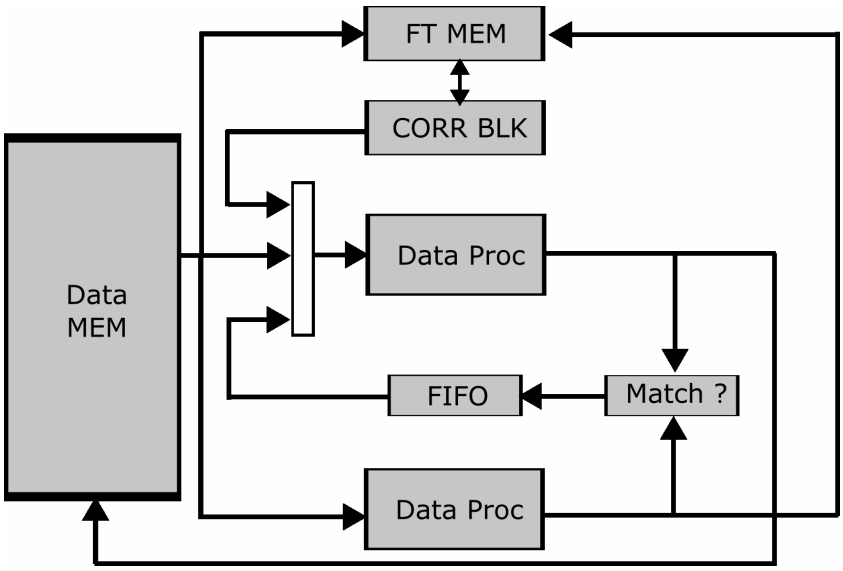
## HW Architecture Assumptions

In order to design an error correction mechanism, we have considered several aspects:

- datapath is made of logic gates; gate fan-out spreads errors;
- SRAM is a common building block; SRAMs are more prone to errors than the underling processing logic; their protection against faults that result in errors is paramount;
- error detection is to be done in parallel with the HW datapath processing;
- Error Correction Codes are used for detection and for efficient redundant data storage;
- data used as input for the iteration processing is unavailable;
- fixed-point quantization;

# FT Mechanisms

General idea:

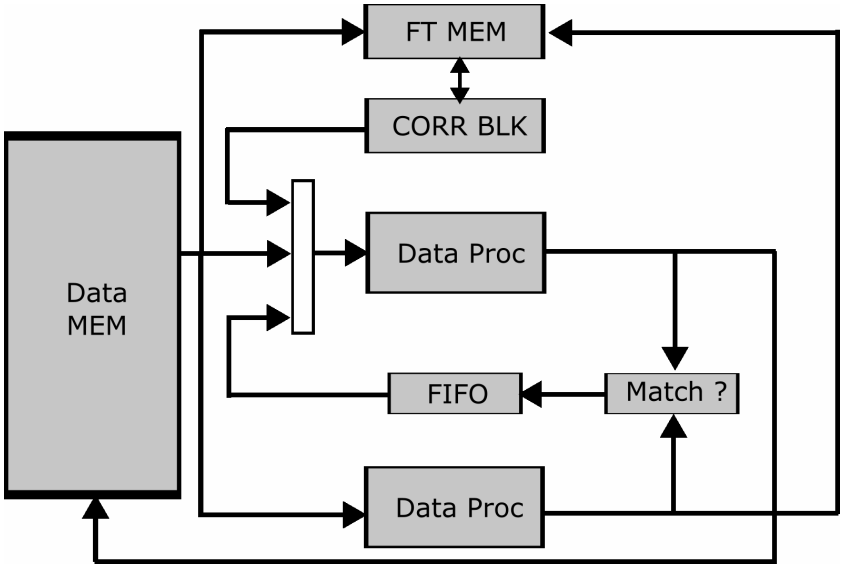



- data encode at every stage (data-path encoder data consistency needs to be ensured!)
- error detection in parallel with processing (on-the-fly syndrome)
- all zero code-word decoding (approximating the errors)



# FT Mechanisms

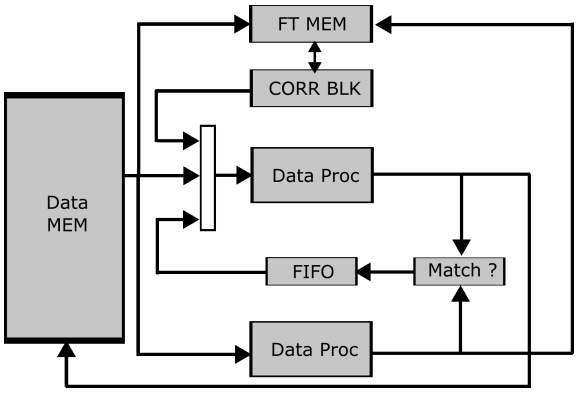
General idea:




- data encode at every stage (FIFO & low-level check-pointing)
- error detection in parallel with processing (FT MEM)
- all zero code-word decoding (Corr Block)

# FT Mechanisms

General idea:




- FIFO & low-level check-pointing;
- error detection in parallel with processing (FT MEM)
- all zero code-word decoding (Corr Block)



# Error manifestation and modeling

| | |
|---|---|
| memory | memory data + random amplitude error |
| encoder | syndrome memory data + random amplitude error |
| syndrome | syndrome memory data + random amplitude error |
| FFT butterfly | memory data + random amplitude error |
| Redundant FFT butterfly | Erroneous Encoded Syndrome! |

Last case is not covered!

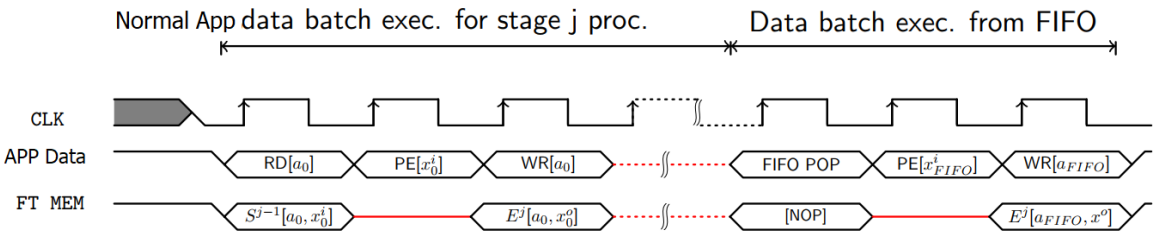
# Encoder input data vulnerability

Proposed solution: a simple checkpoint mechanism, with partial rollback:

- easily detect mismatches between the two butterfly units (xor);
- in case of missmatch:
  - save the input state for the rollback of the input execution inside a FIFO;
  - disable all memory updates;
  - continue to the next set of data;
- At the end of the FFT stage execution, all state data (PE inputs & address, SW) is rolled-back.



# FT & Data memory – revised




- Error free normal operation is done in parallel with the FT operations;
- Data encoded in previous stage is checked at the beginning of stage $j$ (*i.e.* parity checks are computed);
- Data computed during stage $j$ is encoded at the end of processing, before the write to the FFT memory;
- Encoder input data errors are solved at the end of stage processing;

# FT memory block

**Role:**

- Detection: compute syndrome for stage input data;
- Store syndrome vector accesed by the correction loop;
- Store and compute redundant encoded symbols;

**Main operations:**

- Mapping of addresses from application domain to LDPC symbol addresses domain;
  - complexity: number of $B$ rows $\times$ translation ops (part select, shift, FixtP modulo $z$ additions on $\lceil z \rceil$ bits);
- Computation logic for Syndrome, and PC storage;
  - complexity: number of $B$ rows $\times$ data size subtract ops;
- Syndrome – Zero evaluation of PCs;
  - complexity: number of $B$ rows $\times$ abs conversions followed by comparisons;



# FT memory block

**Role:**

- Detection: compute syndrome for stage input data;
- Store syndrome vector accesed by the correction loop;
- Store and compute redundant encoded symbols;

**Application/Code dependencies:**

- LDPC code parameters ($z$, $B$ matrix dimension, $B$ matrix info):
  - number of ops is – $\times$ (number of $B$ rows);
  - ROM memory for storing $B$ info
- Application & implementation choice:
  - number of data per cc processed inside the baseline architecture;
  - FixtP data representation (precision);
  - mapping of application data to LDPC symbols is influenced by application data access patterns;

## Strategy for On-the-fly syndrome checking

- At the end of a processing stage, the PCs need to be checked implemented as:
  - circulant $z$ clock cycles delay accesses & a AND tree of $B$ number of columns;
  - AND tree of size number of redundant symbols (number of PCs), computing the syndrome in 1 cc processing delay;
- **Proposed solution:** add an addition flag bit to each $B$ matrix memory element ROM entry to mark the last update for each PC taking into account data mapping and accesses;
  - At the end of each PC update the flag read from the control ROM is evaluated and the <is codeword> condition revised as: <is codeword> flag AND ROM flag AND $zero(PC)$;
- **Remark 2:** this condition is efficiently defined at $B$ matrix level , otherwise the cost of the ROM storing the flags is non-negligible;



## Strategy for Address translation & App to LDPC data mapping

**Design decisions:**

- Number of iteration data elements versus number of code-words;

If more then 1 data element per cc is mapped to a codeword, data access patters:

- Parity checks circulate data on a row basis of $H$
- Application accesses and updates data patterns;

**Data access considerations:**

- The addressing is dictated by the underlying Application;
- Since data may occupy different banks at different stages during processing, and have different interleaving rules – paramount to identify common information for mapping consistency across iterations;
- Each Application data needs a corresponding symbol address (*i.e.* column index from the parity check matrix $H$);
- Based on the symbol address and the $B$ matrix control information, the addresses of the parity checks can be derived;

# Gradient iterative correction – revised

- Correction loop extracts the noise!
- Use the approximate gradient descent inversion function as the sum of satisfied syndromes, within a margin $\epsilon$, to which the variable node $x_k$ contributes.

$$\Delta_k^{GDRR} = \sum_{m' \in H(k)} zero(r_{m'}) \tag{1}$$

- Equivalent to all zero codeword + channel noise decoding!
- the correction offset (-error) is processed by stage $j$ and added to the FFT data memory;
- There is no error propagated from stage $j$-1!



# GDSU Algorithm

**Algorithm 4** Iterative Gradient Based Correction Loop integrated in FFT stage $j$ processing flow

**Compute FFT stage $j$, Encoding, and Syndrome acc. to Fig. 4.1**
**set** $i = 0$
**while** $(i \leq It_{max})$ and $(syndrome = false)$ **do**
  **Find symbol update position set** $F$
  $F = \{n' | n' = arg \max_{k \in [1,n]} \Delta_k^{GDRR}\}$
  **for all** $x_k \in F$ **do**
    **compute gradient reliability acc. to (4.4) and** $\delta_{n'}$ **update value acc. to (2.6)**
    **if** $\delta_{n'} \neq 0$ **then**
      Store $\delta_{n'}$ and FFT address for $n'$ symbol to the correction vector
  **for all** $\delta_{n'}$ and FFT address $a_{n'}^j$ pairs corresponding to symbol node $n'$ **do**
    **update** $r_{n'}^j = r_{n'}^j + \delta_{n'}$
    Execute FFT proc for $\delta_{n'}^o$=FFT proc$(\delta_{n'})$
    Update FFT data mem entry $x_{a_{n'}}^o = x_{a_{n'}}^o + \delta_{n'}^o$
    Map address $a_{n'}$ to stage $j+1$ symbol $n$
    Update $j+1$ stage partial encoding by adding $\delta_{n'}^o$
  **set** $i = i + 1$
**if** $(syndrome = false)$ **then**
  Exit processing with ERROR flag
Toggle partial encoding vector with syndrome vector
GoTo iteration j+1 processing

# Strategy for the GDSU implementation

Role: estimaties using gradient descent and maximum likelihood decoding rule the error in the data read from memory, as well as the Application data that is processed during the current stage; *There are three phases during the correction block processing:*

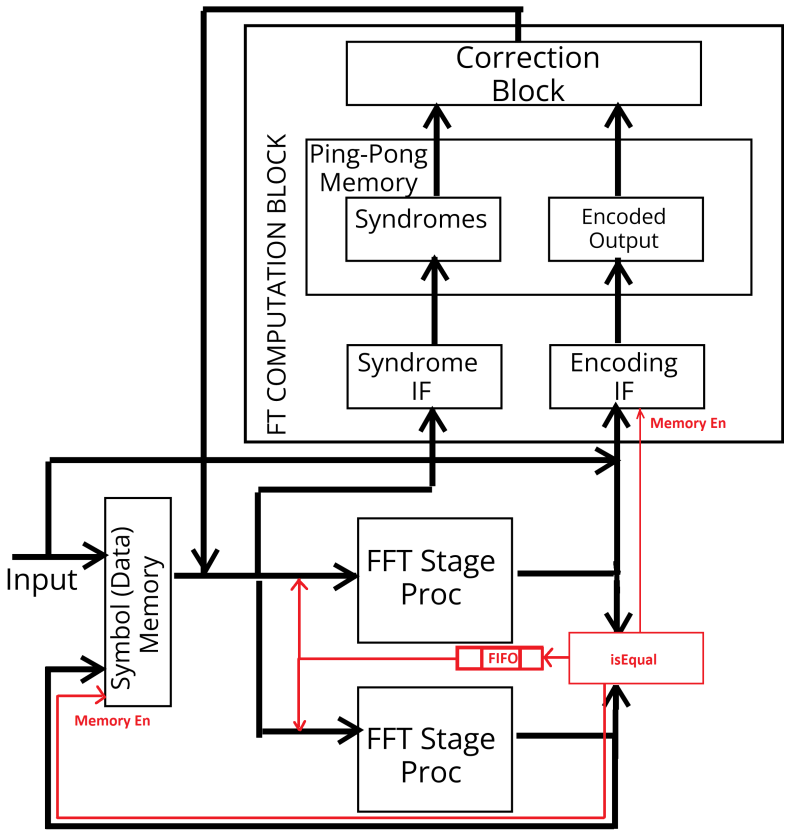
Step1 Compute the energy for each symbol (acc.1), and compute the maximum of all energies;

Step2 Select the symbols having (*i.e.* set $F$) the maximum energy, denoted by $E_{MAX}$, and compute the correction value.

Step3 Update the syndrome vector, and compute the correction offsets by the FFT datapath; Update the existing stage $j$ output values with correction offset computed by the FFT processing of the estimated input error values



# Bird's eye view of modified FT-FFT architecture



# OPT: On-the-fly syndrome checking

- add an addition flag bit to each $B$ matrix memory element ROM entry to mark the last update for each PC;
- **Remark 1:** In effect two such flags are needed one for stage processing and one for offload;
  - At the end of each PC update the flag read from the control ROM is evaluated and the <is codeword> condition revised as: <is codeword> flag AND ROM flag AND $zero(PC)$;
- **Remark 2:** this condition is efficiently defined at $B$ matrix level ($8 \times 24$ bits for $B$ versus $256 \times 768$ bits for the $H$ matrix);
  - Offload has a different interleaving law and is serial one bank at a time;



# OPT: LDPC single diagonal code selection

IEEE 802.11n/ac standard staircase structure for the $B$ parity part simplifies the encoding process, and the computation of parity symbols $[p(0), p(1), \ldots, p(N-M-1)$ as follows:

- Step 1: Compute $H^s \times x = [s_0, s_1, \ldots, s_{N-M-1}]$. The $H^s$ matrix; the parity check equation $i$ is satisfied if $\sum_{m \in H(i)} x_m = 0$, which is equivalent to saying that $\sum_{m \in H^s(i)} x_m = -\sum_{m_p \in H^p(i)} x^p_{m_p}$.
- Step 2: Compute the first parity symbol group $p(0), \ldots, p_{z-1}$ as: $\sum_{i=0}^{M/z} s(i) \times (-1)^{\lfloor i/z \rfloor}$; all $z$ values can be computed in parallel given the quasi-cyclic nature of the $H$ matrix;
- Step 3: Compute the rest of the parity symbols from the parity equations given the double diagonal staircase like structure of the $B^p$ matrix.

**Design choice** – Compute only step 1! -simplified encode by changing the double diagonal of an LDPC code to a simple diagonal.

# OPT: GDSU execution and storage

The GDSU execution and data storage has been optimized as follows:

- on the fly max energy calculations (energies are nor stored explicitly), and correction values computation if value matches the current max energy, otherwise flush LIFO buffer storing corrections;
- all zero codeword decode;
- computing the max number of corrected symbols per iteration (estimated by simulation) – reduce LIFO buffer size;
- implementation opts.: tree structures for counting of zero signs, and one signs, as well as the energy, and absolute min value computation;



# GDSU principle – revised

- The term $zero(y_k - x_k)$ is removed from the inertia decoding rule.
- Correction loop extracts the error! – Equivalent to all zero codeword + channel noise decoding!
- the correction offset (-error) is processed by stage $j$ and added to the FFT data memory;
  - Only possible since the underlying application is a linear transform!
- There is no error propagated from stage $j$-1!

# OPT: FFT memory block implementation

- propose mapping rules from FFT address to LDPC symbol which are conflict free (*i.e.* )
- the on-the-fly syndrome calculation: syndromes computed in parallel with the data (useful symbols) processing
- implementation optimization: the quad port emulated by dual-port
- LDPC related information stored in ROM files, address translation – simple part select, simple modulo z=32 operation;



# Simulation performance

Rate 2/3 code selected:

$$\begin{pmatrix} 25 & 26 & 14 & -1 & 20 & -1 & 2 & -1 & 4 & -1 & -1 & 8 & -1 & 16 & -1 & 18 & 0 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ 10 & 9 & 15 & 11 & -1 & 0 & -1 & 1 & -1 & -1 & 18 & -1 & 8 & -1 & 10 & -1 & -1 & 0 & -1 & -1 & -1 & -1 & -1 & -1 \\ 16 & 2 & 20 & 26 & 21 & -1 & 6 & -1 & 1 & 26 & -1 & 7 & -1 & -1 & -1 & -1 & -1 & -1 & 0 & -1 & -1 & -1 & -1 & -1 \\ 10 & 13 & 5 & 0 & -1 & 3 & -1 & 7 & -1 & -1 & 26 & -1 & -1 & 13 & -1 & 16 & -1 & -1 & -1 & 0 & -1 & -1 & -1 & -1 \\ 23 & 14 & 24 & -1 & 12 & -1 & 19 & -1 & 17 & -1 & -1 & -1 & 20 & -1 & 21 & -1 & -1 & -1 & -1 & -1 & 0 & -1 & -1 & -1 \\ 6 & 22 & 9 & 20 & -1 & 25 & -1 & 17 & -1 & 8 & -1 & 14 & -1 & 18 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 0 & -1 & -1 \\ 14 & 23 & 21 & 11 & 20 & -1 & 24 & -1 & 18 & -1 & 19 & -1 & -1 & -1 & -1 & 22 & -1 & -1 & -1 & -1 & -1 & -1 & 0 & -1 \\ 17 & 11 & 11 & 20 & -1 & 21 & -1 & 26 & -1 & 3 & -1 & -1 & 18 & -1 & 26 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 0 \end{pmatrix} \quad (2)$$

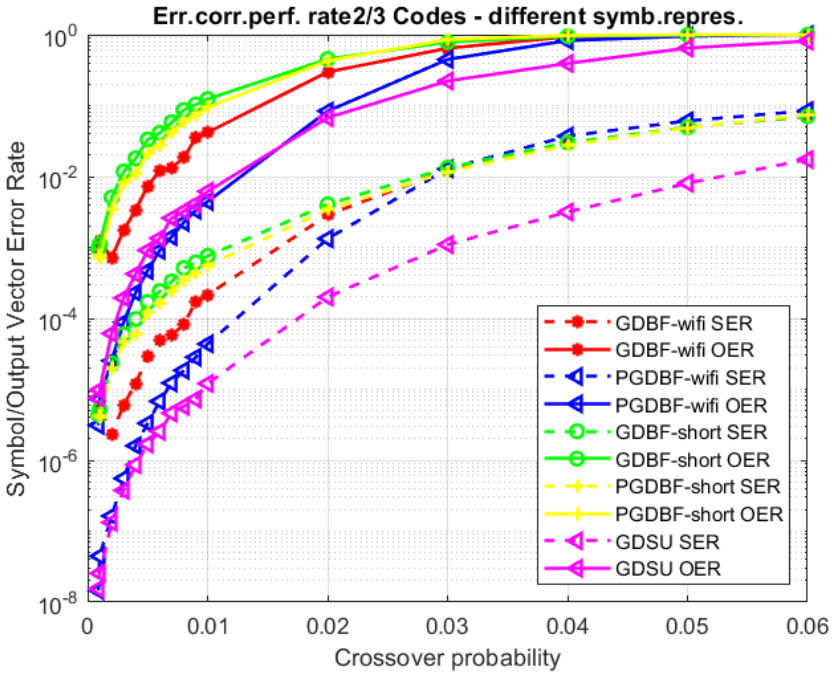
In addition to this, we have removed the channel from the local energy computation.
The effect on decoding performance needs to be evaluated!

# Simulation performance

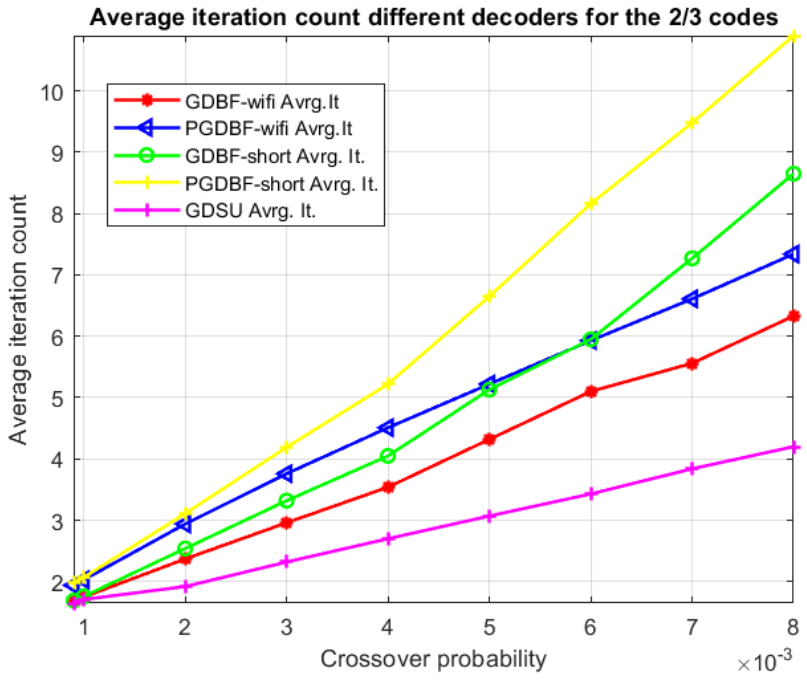
- Goal: to evaluate the performance of the $Short_2/3$ code for the GDSU, the binary PGDBF anf GDBF, as well as the binary GDBF and PGDBF versions of the WiFi code.
- A total of 100 Output/Frame Errors have been simulated for each point out of a maximum of 100 million frames, and a maximum of 100 decoding iterations.
- Objectives: evaluate against binary, and conclude whether is gives a good cost/performance trade-off!
- Errors are assumed random, of amplitude in range [-2,2]
- OER - if maximum number of iterations is reached, and <is codeword> flag is not satisfies, or if an error is detected in the output vector.



# Simulation statistics- OER/SER

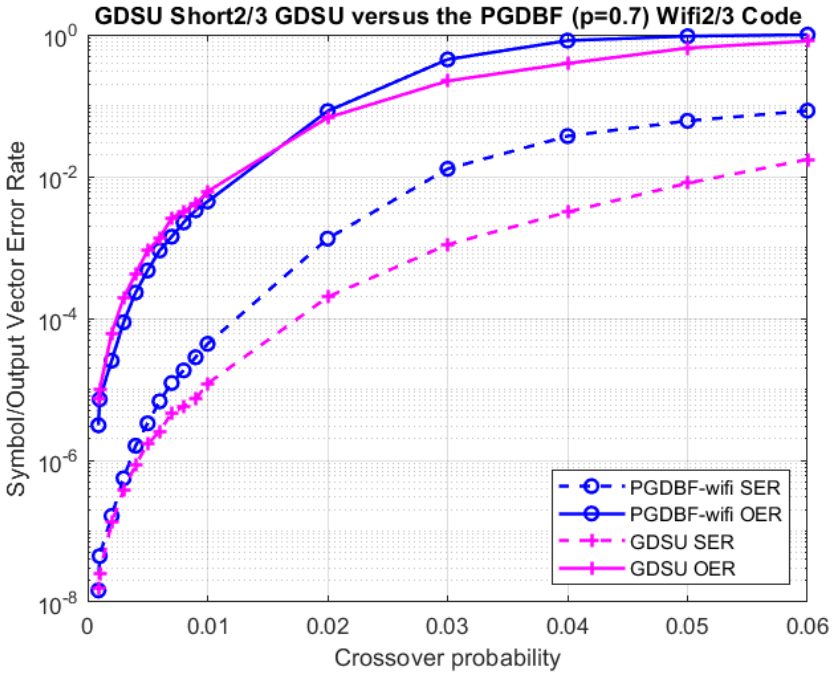


# Simulation statistics- Average iterations






# Simulation statistics- real versus binary



## Synthesis results

| Circuit | Slice LUT | Regs | DSP | BRAM (18 kb) |
|---|---|---|---|---|
| FT-FFT | 5977 | 5720 | 4 | 3 |
| GDSU | 773 | 1364 | 0 | 0 |
| FT Computation Block | 1886 | 1979 | 0 | 0 |
| Baseline FFT | 307 | 206 | 4 | 3 |

Table: Synthesis results for the FT implementation of the 512 point memory based FFT

- 1 PE datapath
- 512 points FFT
- rate 2/3 code
- OPT1: on-the-fly syndrome
- OPT2: correction loop LIFO buffer with size tuned by simulation (32);



## Synth results discussion

| Circuit | Slice LUT | Regs | DSP | BRAM (18 kb) |
|---|---|---|---|---|
| FT-FFT | 5977 | 5720 | 4 | 3 |
| GDSU | 773 | 1364 | 0 | 0 |
| FT Computation Block | 1886 | 1979 | 0 | 0 |
| Baseline FFT | 307 | 206 | 4 | 3 |

Table: Synthesis results for the FT implementation of the 512 point memory based FFT

With repect to logic and FF used:

- correction loop cost $\approx$ the cost of the 8 point fully parallel FFT design;
- the fault tolerant block $\approx 6\times$ the cost of the 8 point FFT;
- FT mechanism for an iterative stage by stage 512 point FFT $\approx 7\times$ the cost of the 8 point fully-parallel FFT implementation;

## Conclusions - Summary of optimizations

- Verilog HDL model of a FFT enchanced with LDPC redundant data storage and gradient descent correction has been implemented from SCRATCH;
- Verified FT behavior using random error simulations → Demonstrator
- OPT 1: efficient conflict free mapping;
- OPT 2: on-the-fly syndrome computation and <is codeword> condition evaluation;
- OPT 3: On-the-fly computation and efficient storage in LIFO buffer of the estimated errors during an iteration of the correction loop;
- OPT 4: two-frequency domain FT Memory implementation;
- all arithmetic units are implemented to the exact required precision for each level;
- OPT5: FSMs have tight synchronization with no dead cycles in between;
- OPT6: Mitigation of Encoder vulnerability!



## Discussion

Use-case so far – Multiple data per cc encoded in the same code-word which:

- for 1 butterfly unit – managed an optimized implementation;
- Limited scalability due to increase in the number of FT mem block number of ports;
- may lead to FT memory data access conflicts;

Proposed solution:

- One data per cc encoded in an codeword;

Single data channel encoding!

## Discussion cont'

Single data channel encoding!
Advantages:

- Decouples the Application data access flow from the LDPC data access flow.
- Scales with the increase in data items processed within the Application architecture per clock-cycle.
- No address mapping is required!
- The multi-port memory blocks are no longer needed in the FT MEM block implementation;

Disadvantages:

- Computing the correction offsets is serialized! A minor execution overhead is expected since the number of updates per iteration is relatively small;



## Q&A

Thank you!

# Automation using High Level Synthesis (HLS)

Oana Boncalo

University Politehnica Timisoara

*oana.boncalo@cs.upt.ro*

November 25, 2022



## Overview

# Milestone 3 - ToDo list

Achievements presented:

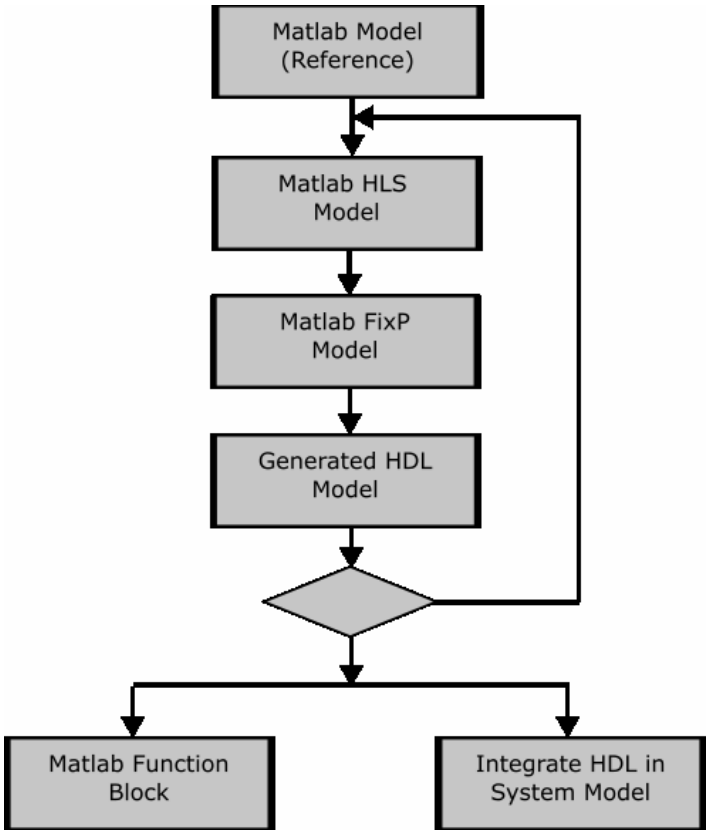
- 512 point FT FFT architecture with optimizations evaluated & demonstrator;
- optimizations: redundant data storage, code selection, on-the-fly syndrome computation, and iterative gradient correction loop implementation that peforms all-zero codeword decoding for LDPC real codes;

The research work planned for the next period requires the following actions:

- Optimization 1: precision tweaking using Matlab.This is an important source of cost saving;
- Optimization 2: improving fault tolerance by addressing the vulnerability of an error manifesting between the redundant part PE output and the next stage encoder block;
- Explore modeling in Matlab/C for HLS code generation of FT components;

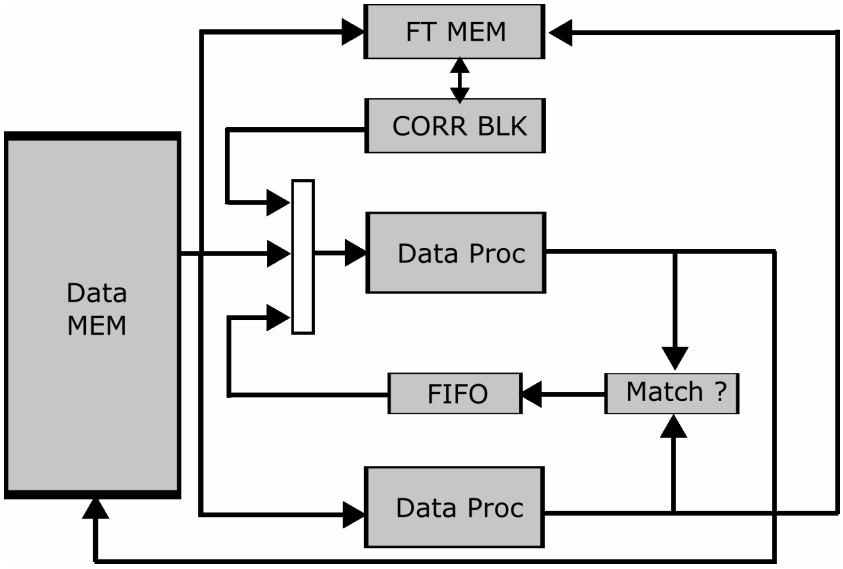


# Proposed automation Flow



# FT Mechanisms

General idea:




USE HLS Matlab model to generate the HDL design model for:

- FT memory block;
- Correction Block;



# The Correction block

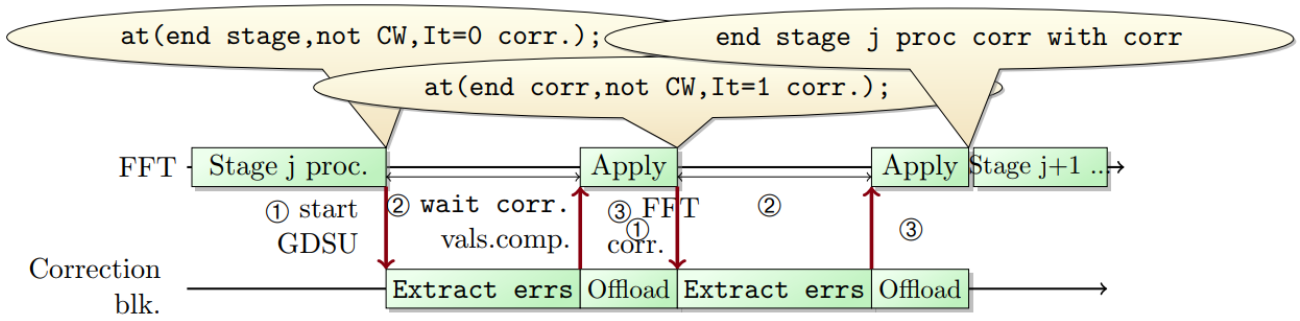
Role: estimate using gradient descent and maximum likelihood decoding rule "the error" in the data read from App. data memory. *There are three phases during the correction block processing:*

Step1 Compute the energy for each symbol, and compute the maximum of all energies, .

Step2 Select the symbols having (*i.e.* set $F$) the maximum energy, denoted by $E_{MAX}$, and compute the correction value.

Step3 Update the syndrome vector, and compute the correction offsets by the Baseline Application datapath; Update the existing stage $j$ output values with correction offset computed by the FFT processing of the estimated input error values

# The Correction block






# Correction Block - implementation details

The HLS Matlab model is built of the following sub-components:

- Finite state machine that is responsible for:
  - generate the FFT access patters for reading the syndrome data in Step1;
  - manage the control signal generation of the Last-In First-Out (LIFO) buffer: reset the current maximum energy and number of LIFO data entries, enable the load of a new set of LIFO data entries, enable the offload of a set of LIFO data entries;
- Combinational logic – tree structure modeling for – energy computation, and counting positive and negative signs;
- Update of the sign and maximum energy values;
- Buffer (LIFO) for storing the FFT state (*e.g.* data, address);

# Correction Block - verification & discussion

HLS model details:

- FixtP model;
- Detailed data-path modeling:
  - count – adder trees;
  - sum (Energy) – adder tree;
  - comparison (min absolute value) – comparator tree;
  - all trees have precision preciseness set for each level;
- The Matlab Test bench uses the same high level Matlab function model used for the DPI HD Verilog HDL unit verification.



# Correction Block - cost

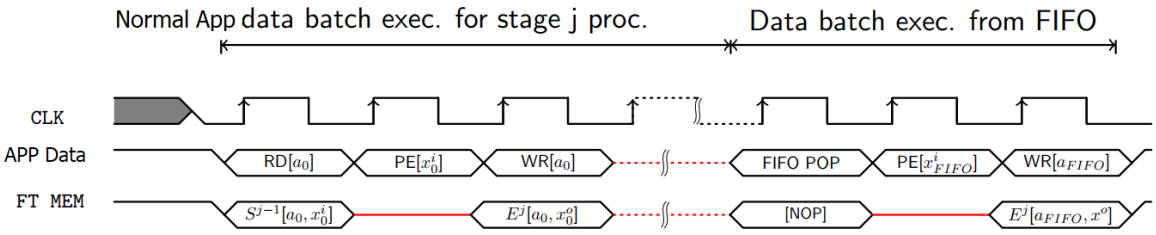
| Circuit | Slice LUT | Regs | DSP | BRAM (18 kb) | F7/F8 MUX |
|---|---|---|---|---|---|
| GDSU | 773 | 1364 | 0 | 0 | 0 |
| HLS GDSU | 1873 | 1749 | 0 | 0 | 250 |

Synthesis results for Correction block implementing the Gradient Descent Symbol Update Iterative loop (GDSU) for the code parameters of the 512 point memory based FFT use-case.

- results show an increase in logic ( 2 $\times$ Slice LUT), and a noticeable increase in MUX utilization (168 F7 Muxes, and 42 F8 Muxes respectively);
- Regs cost increase likely caused by the Matlab R2020b HLS option for pipeline input and output;

# FT memory - multiple data encoding channel (MDC)

Role: store and process the redundant information.




- Address translation for Syndrome– FFT address to symbol map for both the Syndrome interface (during FFT data memory read, and correction block read and write operations);
- Address translation for Encode – FFT address to symbol map for the Encode interface (during FFT data memory write, and FFT correction offset update operation);
- Processing – Given the diagonal structure of the parity symbols from the base ($B$) and parity check ($H$) matrices –
  - the syndrome computation is a simple substract of the data (useful symbols) read from FFT data memory from the parity checks dictated by $H$.
  - the encoding computation is a simple add of the output stage FFT



# FT MEM MDC – implementation details

The address translation rule to match the FFT read sequence for the for the read IF (syndrome) is depicted in:

$$\begin{aligned} idx_{COL} &= \lfloor addr_{FFT}/z_0 \rfloor \\ symbol_{addr0} &= idx_{COL} \times z + mod(addr_{FFT}, z_0); \end{aligned} \quad (1)$$

The memory design with optimization is challenging:

- 2 clock domains, quad port emulated by dual port working in higher frequency;
- several memory blocks;

# FT MEM MDC – Float to FixtP

The address translation rule to match the FFT read sequence for the for the read IF (syndrome) is depicted in:

$$\begin{aligned} idx_{COL} &= \lfloor addr_{FFT}/z_0 \rfloor \\ symbol_{addr0} &= idx_{COL} \times z + mod(addr_{FFT}, z_0); \end{aligned} \tag{2}$$

The memory design with optimization is challenging:

- the expansion factor $z$, as well as $z_0$ are a power of 2, in HDL the division by $z$, the modulo by $z$, and the multiplication are part select operations or shift operations;
  - easy to do in FixtP;
  - FixtP not supported for Float to FixtP conversion!
  - The solution for performing these operations was that of using masks, as well as shift operations;
  - experienced issues with shift operators, leading to failed fixed-point simulation after conversion from floating point;
    - Cause: the final precision was correct, intermediate shift left operations required a higher precision than that of the final result;
    - Error message did not give sufficient info to localize problem;
    - The solution for performing these operations was that of using masks, as well as shift operations;



# FT MEM MDC – Float to FixtP

The memory design with optimization is challenging:

- Only found solutions based on Simulink;
- clock-domain crossing, possibly by some Simulink build in block;
- considered options: dual rate dual port, separate addres and data selection logic from the actual storage;
- Multi-port register file – modeled, BUT not a cost-efficient solution!

Verification:

- A time unroll of the input signals and function calls;
- Scales poorly with the increase of interface control signals and interface commands;
- Cumbersome and error prone!

Cost estimates of individual components:

- Synthesis estimates require a Matlab test-bench of the HLS sub-component from the design;

# FT MEM MDC – cost

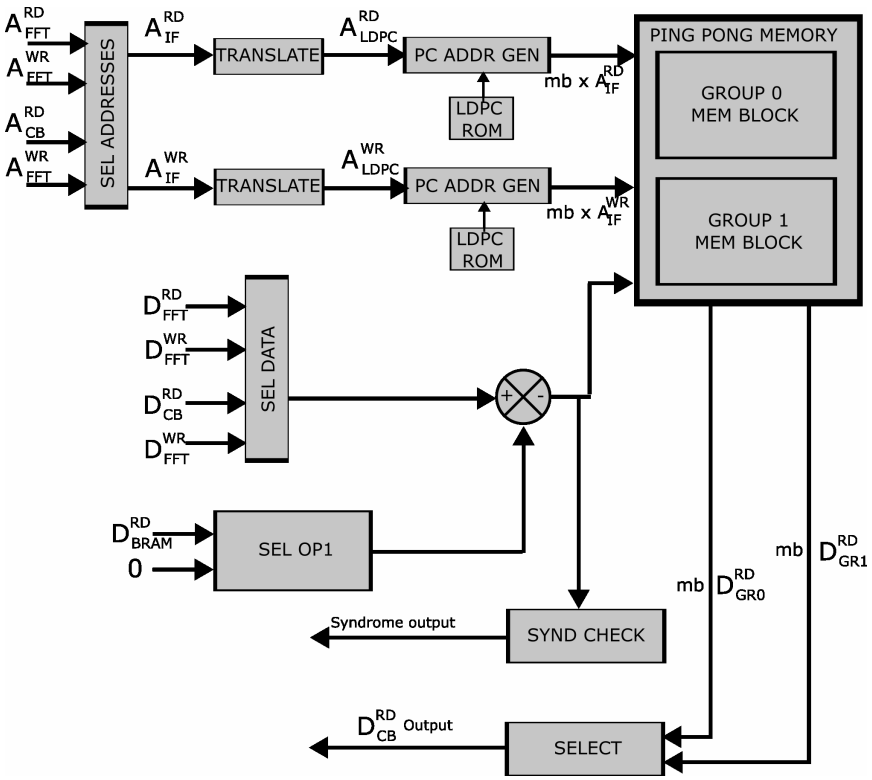
| Circuit | Slice LUT | Regs | DSP | BRAM(18 kb) | F7/F8MUX |
|---|---|---|---|---|---|
| Mem update(HDL) | 1534 | 2049 | 0 | 0 | 0 |
| Syndrome IF(HDL) | 192 | 0 | 0 | 0 | 0 |
| Encode IF(HDL) | 171 | 0 | 0 | 0 | 0 |
| FT memory(HDL) | 1886 | 1979 | 0 | 0 | 0 |
| Mem update(HLS) | 5086 | 353 | 0 | 0 | 0 |
| Syndrome IF(HLS) | 103 | 40 | 0 | 4 | 0 |
| Encode IF(HLS) | 119 | 36 | 0 | 4 | 0 |
| FT MEM MDC(HLS) | 34590 | 2971 | 0 | 8 | 7223 |

Synthesis results – Vivado tool and targeted Xilinx Virtex-7 FPGA, device xc7vx485t, package ffg1761, and speed grade -2.

- 10 $\times$ higher cost then its Verilog HLD model alternative;
- the IFs storage uses Distributed RAM;
- Cause: the quad-port memory model!
- It is "easy" to generate an architecture with sub-optimal memory!



# FT memory - single data encoding channel (SDC)




- "Cleaned up" model! Better than the Verilig HDL organization.

# FT MEM SDC – implementation & conversion details

- Read modify write not so trivial to model in Matlab;
- Delays for Sync purposes a hurdle;
- hdl.RAM – nice way to have multiple bank memory (vector addr in, vector data, etc.);
- the address translation rule not needed;
- loop unroll, float to fixed point automatic conversion and model generation;
- annoying issue with Verilog HDL model for FixtP model – hierarchy flattening;
- Matlab Verilog HDL generation extremely sensitive to Test bench quality!



# FT MEM SDC – cost

| Circuit | Slice LUT | Regs | DSP | BRAM(18 kb) | F7/F8 MUX |
|---|---|---|---|---|---|
| Mem update (HDL) | 1534 | 2049 | 0 | 0 | 0 |
| Syndrome IF (HDL) | 192 | 0 | 0 | 0 | 0 |
| Encode IF (HDL) | 171 | 0 | 0 | 0 | 0 |
| FT memory (HDL) | 1886 | 1979 | 0 | 0 | 0 |
| FT MEM SDC (HLS) | 1521 | 1692 | 0 | 16 | 0 |
| FT MEM MDC (HLS) | 34590 | 2971 | 0 | 8 | 7223 |

Synthesis results – Vivado tool and targeted Xilinx Virtex-7 FPGA, device xc7vx485t, package ffg1761, and speed grade 2.

- 0.6 $\times$ higher Slice LUT cost then its Verilog HLD model alternative;
- 0.7 $\times$ higher Regs cost then its Verilog HLD model alternative;
- 16 extra BRAM!
- 260 MHz versus 70 MHz!

# Pitfalls of HLS modeling

- multi-clock design, and clock domain domain crossing synchronization;
- optimized memory design;
- Functions that have persistent data and need to be called more than once in the Matlab design need to be replicated (*i.e.* copy and paste and function rename).
- Bitwise operations with data-types that are not fixed point are problematic.
- Generic modeling require model parameters. Furthermore, similar to their HDL counterparts the HLS models require static parameter definition. There are two ways to propagate these parameters inside a model:
  - add the parameters as function input;
  - design a configuration script/function;



# Pitfalls of HLS modeling

- Matlab is a loosely or weakly-typed, as well as a dynamically typed language (*i.e.* variables don't have to be explicitly declared; type is not explicitly specified). A lot of landmines:
  - the data casts from one type to another are handled automatically;
  - Always cast, initialize in the model and Test-bench!
  - Code coverage paramount!

  ,
- Test-bench quality and input data and scenarios coverage!!!
- Test-bench support missing for scenarios coverage!!!
- No concurrent execution and "time" concept;
- Some HDL component modeling require more lines of Matlab compared with Verilog HDL: *e.g.* Finite State Machines, functions that need to be replicated due to having persistent variables, and the model needing multiple function calls.

# Pitfalls of HLS modeling

Test-benches require a time unroll of the stimuli generation of a Verilog HDL testbench. Each clock-cycle execution of the testbench requires:

- the assignment of the input data (*a.k.* stimuli);
- the function call of the HLS model;
- the checking of the HLS model outputs – typically either a comparison against the reference, or some output flag state checking;

This does not look "pretty" when:

- there are several operations and input combinations and configurations for the model checked;
- quick-fix refactor using scripts and functions; forseen issues
  - maintenance;
  - model interface changes;
  - better, but still messy on the long run;



# Strong points of HLS modeling

The advantages of HLS modeling are:

- completeness due to automatic code coverage metrics;
- productivity: once the model is in place it is easy to change the code parameters;
- single language modeling;
- functional correctness by automatic testbench generation.

# Conclusions and Future work

- Managed to obtain fairly good results for an HLS generated Verilg HDL model;
- Matlab community for HLS is still small;
  - Forum responses;
  - Number of non-trivial examples;
- Error and Warning messaging can be improved;
- The bias is towards Simulink modeling;

Future work:

- Find a Matlab set of "good practices" for designing reusable and easy to read and maintain Test benches!
- Validate FT MEM SC in the HDL;
- Aggregating Function Blocks in Simulink models programmatically;
- Investigate the scallability of the SDC approach with the number of PE;
- C+Vitis+possibly SystemC for testbench generation!



# Q&A

Thank you!