UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Institute ITACA Information and Communication Technologies

On Improving the Robustness Of Convolutional Neural Networks Using In-Parameter Zero-Space Error Correction Codes

Juan-Carlos Ruiz-García

ITACA-UPV (**Spain)**
jcruizg@disca.upv.es

2ND WORKSHOP VERDI @ DSN2024
24th JUNE 2024, BRISBANE (AUSTRALIA)

Keynote Speaker Profile

- Permanent Professor at Universitat Politècnica de València, Spain
- Lecturing at the School of Computer Science (**ETSINF**) and the Department of Computer Engineering and Networks (**DISCA**)
- Research at Inst. of Information and Communication Technology (**ITACA**)
  - Computer Science
    - **Resilient computing**
  - Environment, transportation and energy
    - **Intelligent transportation**
  - Health and wellbeing
  - Manufacturing technologies and materials
    - **Electronic systems**
  - Telecommunications

$\rightarrow$ **Fault-Tolerant Systems Group (GSTF)**

- More information about me at https://shorturl.at/3I5yB

Acknowledgements

- DEFADAS project: Dependable-enough FPGA-Accelerated Deep neural networks for Automotive Systems
  - Spanish research project funded by grant **PID2020-120271RB-100**
  - Duration 4 years (2021-2025)
  - Project leaders
    - Juan Carlos Ruiz García
    - David de Andrés
  - Topics of research:
    - FPGA-based accelerated convolutional neural networks
    - Dependability assessment through fault injection
    - Fault tolerance using error correction codes

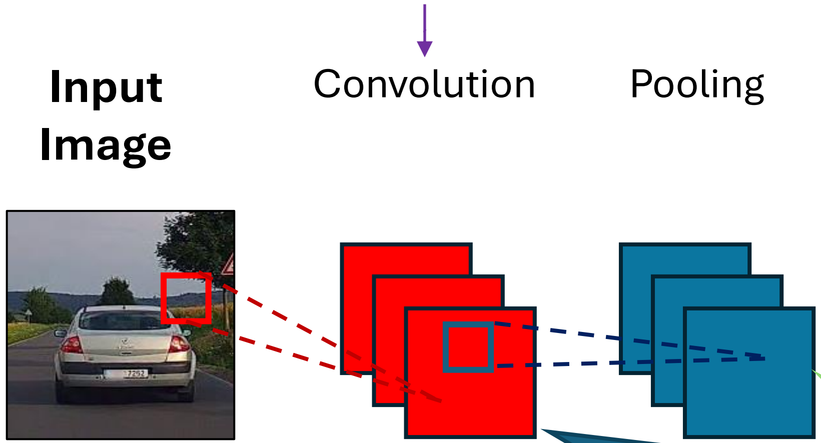
GOBIERNO DE ESPAÑA MINISTERIO DE CIENCIA E INNOVACIÓN AGENCIA ESTATAL DE INVESTIGACIÓN

CNNs in a nutshell

Input Image

CNNs in a nutshell

Training phase => **Parameters (weigths, biases, etc...)**



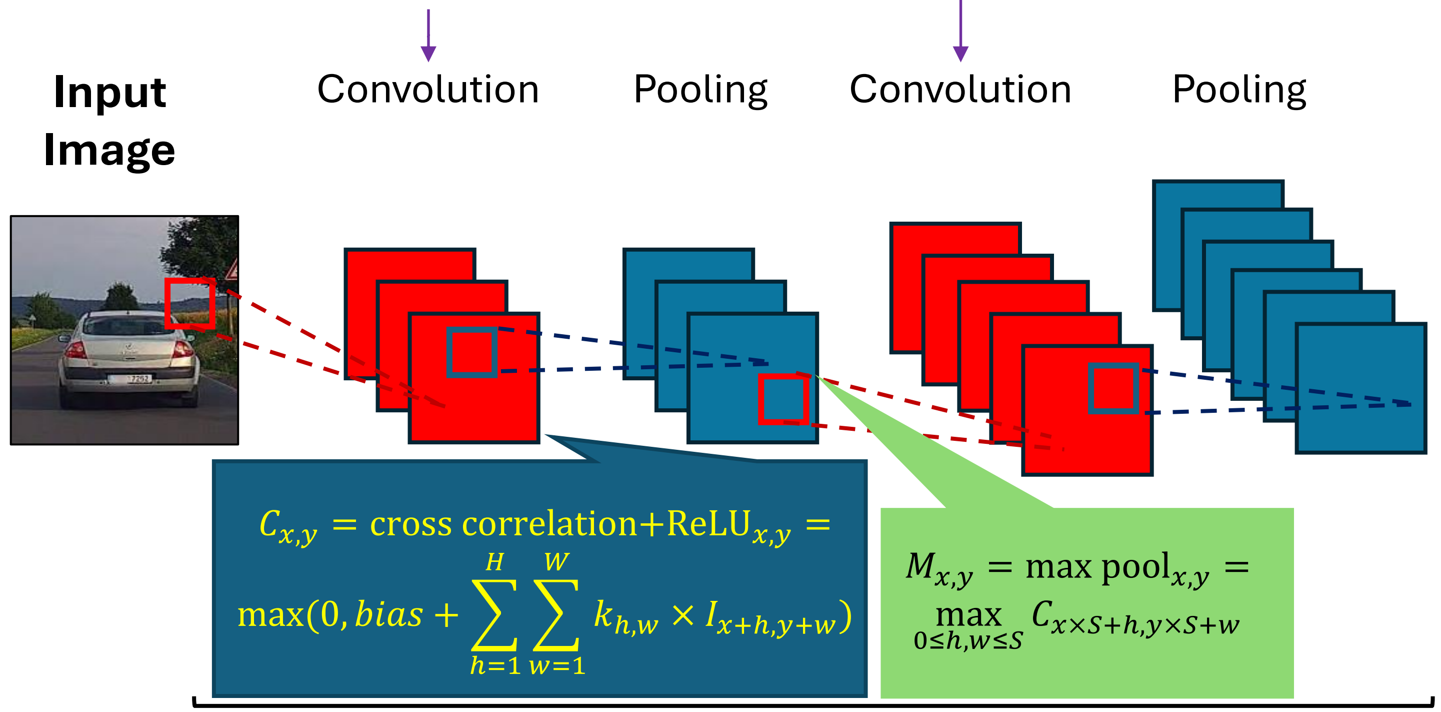
$$C_{x,y} = \text{cross correlation+ReLU}_{x,y} = \max(0, bias + \sum_{h=1}^{H}\sum_{w=1}^{W} k_{h,w} \times I_{x+h,y+w})$$

$$M_{x,y} = \text{max pool}_{x,y} = \max_{0\leq h,w\leq S} C_{x\times S+h,y\times S+w}$$

Feature extraction

CNNs in a nutshell

Training phase => **Parameters (weigths, biases, etc...)**

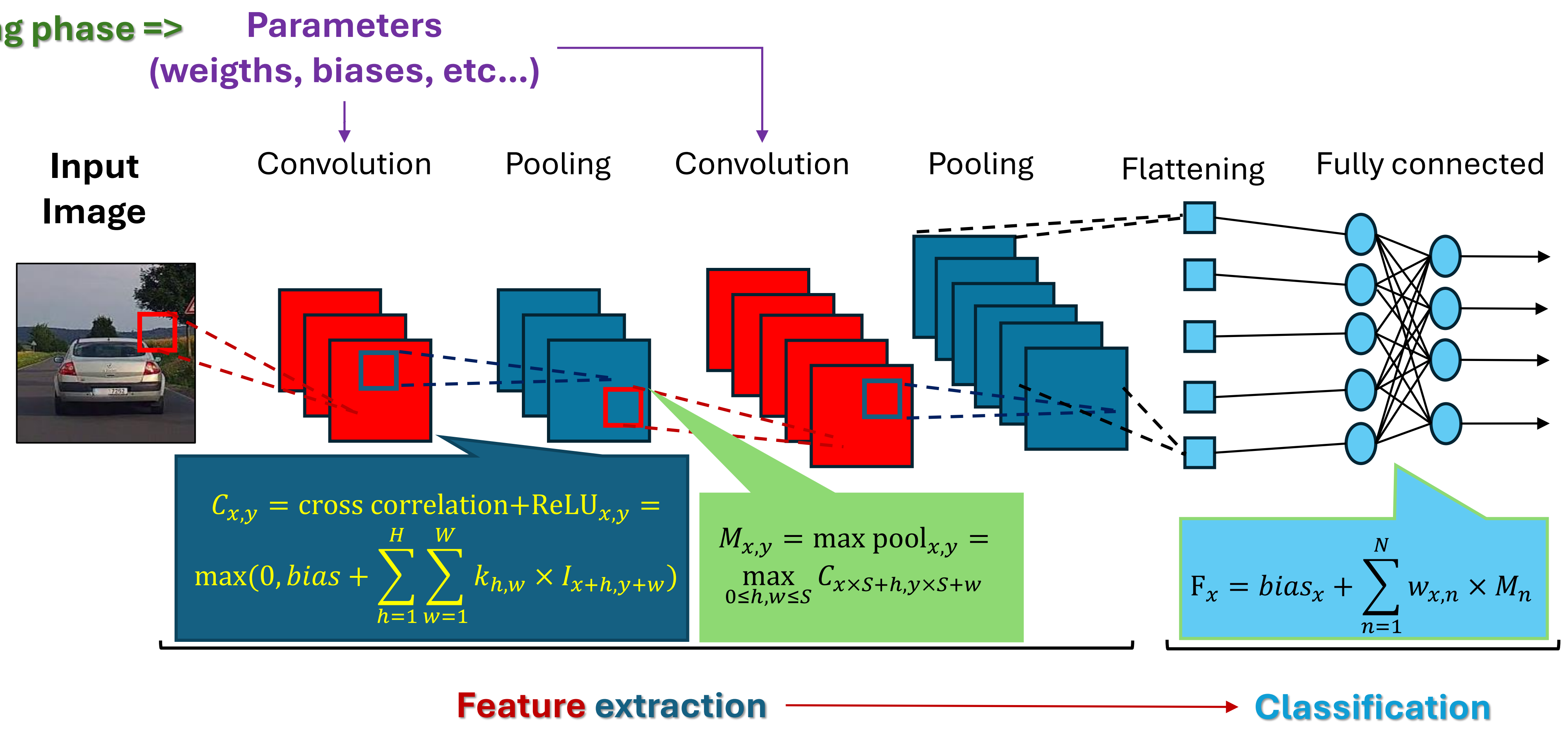


$$C_{x,y} = \text{cross correlation+ReLU}_{x,y} = \max(0, bias + \sum_{h=1}^{H}\sum_{w=1}^{W} k_{h,w} \times I_{x+h,y+w})$$

$$M_{x,y} = \text{max pool}_{x,y} = \max_{0\leq h,w\leq S} C_{x\times S+h,y\times S+w}$$

Feature extraction

CNNs in a nutshell



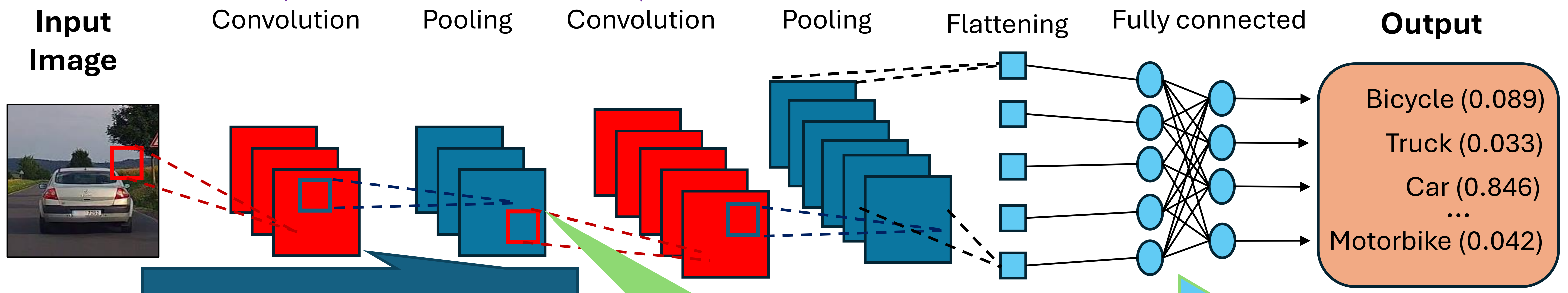
$$C_{x,y} = \text{cross correlation+ReLU}_{x,y} = \max(0, bias + \sum_{h=1}^{H}\sum_{w=1}^{W} k_{h,w} \times I_{x+h,y+w})$$

$$M_{x,y} = \text{max pool}_{x,y} = \max_{0\leq h,w \leq S} C_{x\times S+h,y\times S+w}$$

$$\mathrm{F}_x = bias_x + \sum_{n=1}^{N} w_{x,n} \times M_n$$

CNNs in a nutshell

Training phase => **Parameters (weigths, biases, etc...)**



$$C_{x,y} = \text{cross correlation+ReLU}_{x,y} = \max(0, bias + \sum_{h=1}^{H}\sum_{w=1}^{W} k_{h,w} \times I_{x+h,y+w})$$

$$M_{x,y} = \text{max pool}_{x,y} = \max_{0\leq h,w\leq S} C_{x\times S+h,y\times S+w}$$

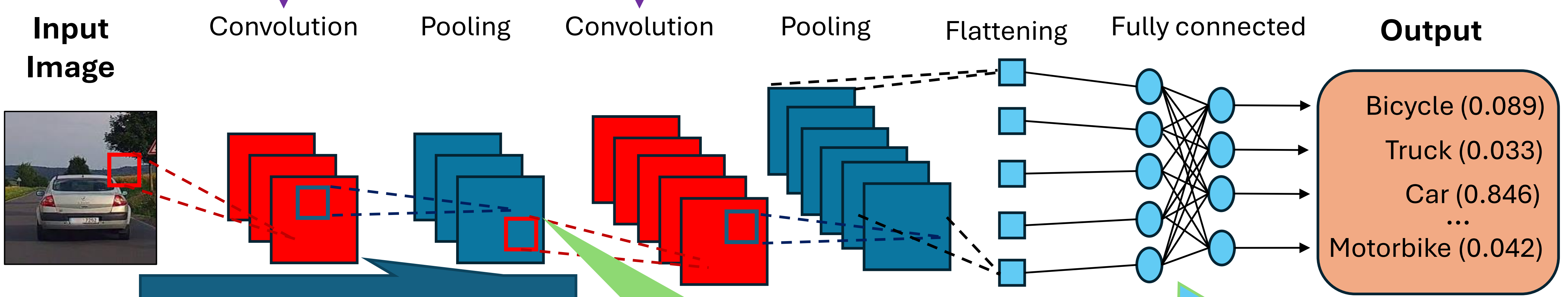
$$\mathrm{F}_x = bias_x + \sum_{n=1}^{N} w_{x,n} \times M_n$$

$$\text{softmax}_x = \frac{e^{F_x}}{\sum_{n=1}^{N} e^{F_n}}$$

Feature extraction $\longrightarrow$ **Classification** $\longrightarrow$ **Probabilistic Distribution**

CNNs in a nutshell

Training phase => Parameters (weigths, biases, etc...)



$$C_{x,y} = \text{cross correlation+ReLU}_{x,y} = \max\left(0, bias + \sum_{h=1}^{H}\sum_{w=1}^{W} k_{h,w} \times I_{x+h,y+w}\right)$$

$$M_{x,y} = \text{max pool}_{x,y} = \max_{0\le h,w\le S} C_{x\times S+h,\, y\times S+w}$$

$$\mathrm{F}_x = bias_x + \sum_{n=1}^{N} w_{x,n} \times M_n$$

$$\text{softmax}_x = \frac{e^{F_x}}{\sum_{n=1}^{N} e^{F_n}}$$

Inference phase => Feature extraction → Classification → Probabilistic Distribution

Use of CNNs in safety-critical systems

- Convolutional neuronal networks (CNN) enable object identification in images, something of great interest for embedded systems

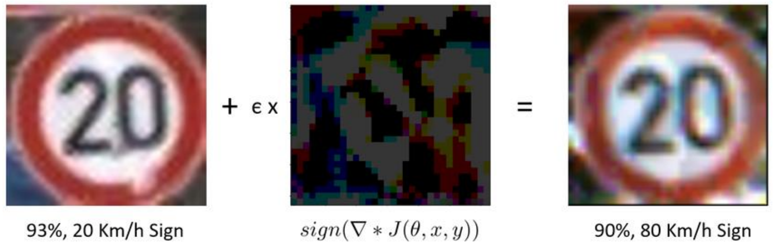
Transportation

Space exploration

- Real-time constrainsts in decisions → need of local inference
  - Use of HW accelerators implementing the CNN or supporting its execution
  - Models are adapted attending to the available computation power, memory and energy

Impairments for the adoption of CNNs

- The lack of transparency (mainly explainability and traceability), and the data-dependent and stochastic nature of CNNs clash against the solutions for critical AI-based systems
  - What do CNNs learn, and what do they miss up, during training?
  - How do operational conditions affect the CNN behavior?



- CNN assessment is in its infancy → Lack of certification standards
  - ISO/IEC Joint Technical Committee for IT (JTC 1), Subcommitee 42 (AI), Working group 3 (Truswothiness) → https://jtc1info.org/sd-2-history/jtc1-subcommittees/sc-42
    - Assessment of the robustness of neural networks (ISO/IEC 24029-1:2021, ISO/IEC 24029-2:2023)

=> Need for experimental dependability assessment of CNNs

Impairments , means and goals of experimental dependability assessment

- Study the considered target system in the presence of faults that may affect its nominal behaviour → representativeness is a must
  - Understanding how the target system works and how it is implemented
  - Activation using representative workloads and faultloads
- Fault injection is a prioritized mean for dependability assessment
  - Expected properties: low intrusiveness and repeatability
  - Useful to detect dependability bottlenecks
- Propose suitable mitigation techniques for identified bottlenecks
  - In the case of embedded systems, consider also the impact of fault tolerance on performance, power consumption and area

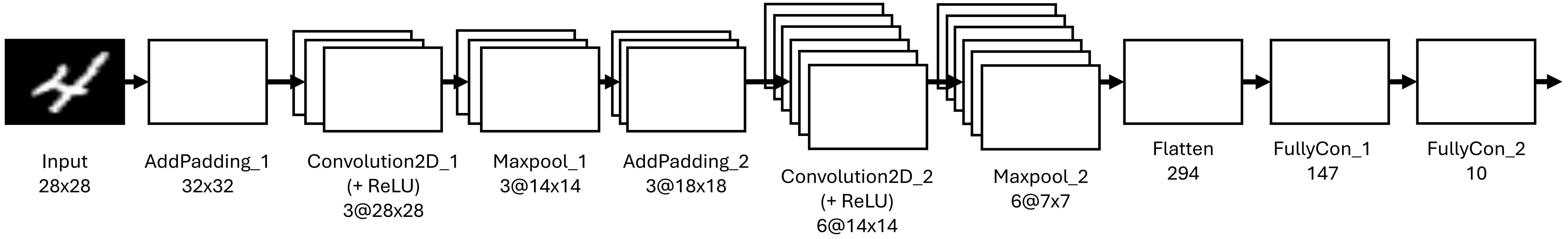
In this talk

- Understanding HW accelerators for CNN:
  Prototyping a FP32 /INT8 CNN on a FPGA: Lenet-5 as a case study
- Robustness evaluation of CNNs using fault injection:
  methodology and lessons learnt
- In-Memory Zero-Space Protection of FP-based CNNs using ECCs:
  methodology and lessons learnt
- Conclusions

In this talk

- **Understanding HW accelerators for CNNs: Prototyping a FP32 /INT8 CNN on a FPGA: Lenet-5 as a case study**
- Robustness evaluation of CNNs using fault injection : methodology and lessons learnt
- In-Memory Zero-Space Protection of FP-based CNNs using ECCs: methodology and lessons learnt
- Conclusions

Lenet-5: A simple CNN



- Identification of manuscript numbers (10 categories)
  - Depth of 2 layers
  - Parameters: 45539 (weights + bias)
  - Dataset: MNIST (10.000 monochrome test images of 28x28 pixels)
  - Accuracy: 98,23% (117 incorrect matches out of 10.000 test images)

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Institute ITACA Information and Communication Technologies

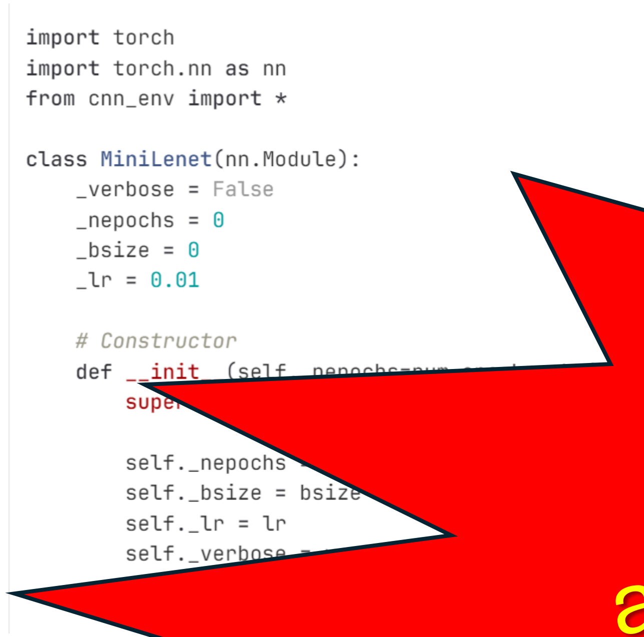
Lenet-5: A simple Python-based CNN

```python
import torch
import torch.nn as nn
from cnn_env import *

class MiniLenet(nn.Module):
    _verbose = False
    _nepochs = 0
    _bsize = 0
    _lr = 0.01

    # Constructor
    def __init__(self, nepochs=num_epochs, bsize=batch_size, lr=learning_rate, verbose = False):
        super(MiniLenet, self).__init__()

        self._nepochs = nepochs
        self._bsize = bsize
        self._lr = lr
        self._verbose = verbose

        self.createLayers()
```

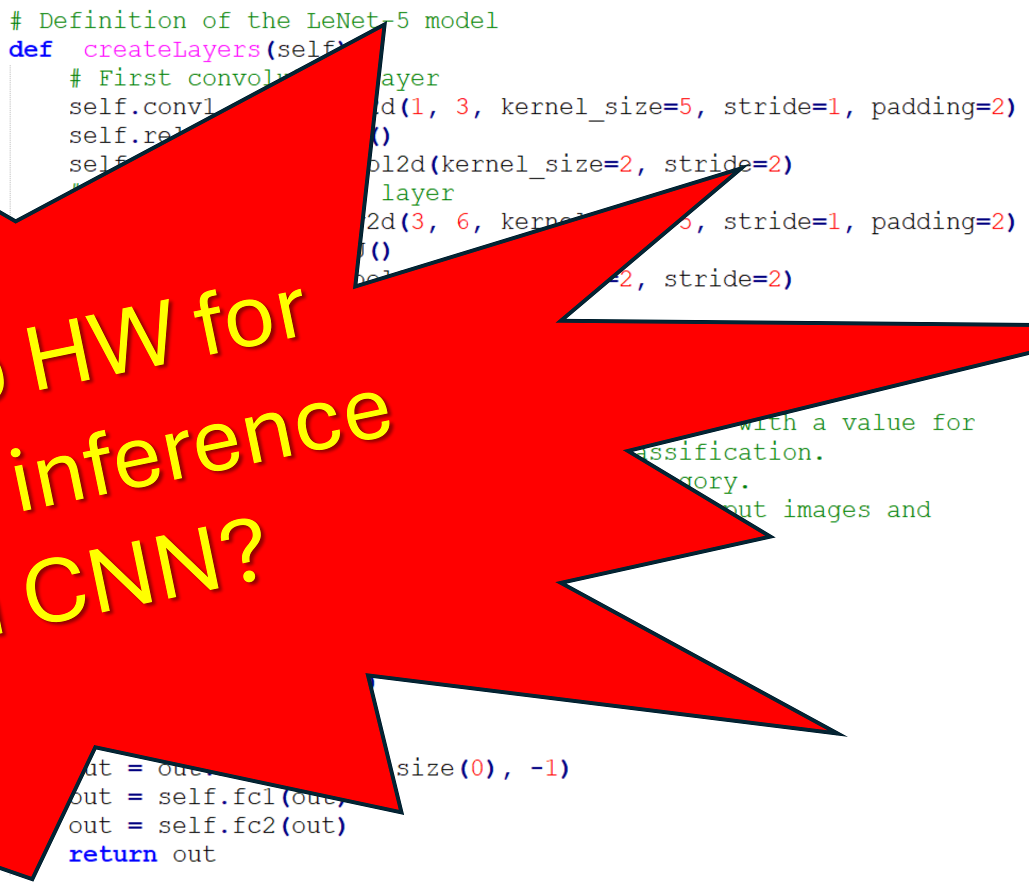
```python
# Definition of the LeNet-5 model
def  createLayers(self):
    # First convolution layer
    self.conv1 = nn.Conv2d(1, 3, kernel_size=5, stride=1, padding=2)
    self.relu1 = nn.ReLU()
    self.max1 = nn.MaxPool2d(kernel_size=2, stride=2)
    # Second convolution layer
    self.conv2 = nn.Conv2d(3, 6, kernel_size=5, stride=1, padding=2)
    self.relu2 = nn.ReLU()
    self.max2 = nn.MaxPool2d(kernel_size=2, stride=2)
    # Two fully connected layers
    self.fc1 = nn.Linear(7 * 7 * 6, 147)
    self.fc2 = nn.Linear(147, 10)

# Processes the input image and returns a tensor with a value for
# each of the considered categories for classification.
# The highest value denotes the selected category.
# In verbose mode it generates files with the input images and
# the output of each layers.
def forward(self, x):
    out = self.conv1(x)
    out = self.relu1(out)
    out = self.max1(out)
    out = self.conv2(out)
    out = self.relu2(out)
    out = self.max2(out)
    out = out.reshape(out.size(0), -1)
    out = self.fc1(out)
    out = self.fc2(out)
    return out
```

Implementation available at:
https://git.upv.es/defadas/MiniLenetPython

Lenet-5: A simple Python-based CNN



```
import torch
import torch.nn as nn
from cnn_env import *

class MiniLenet(nn.Module):
    _verbose = False
    _nepochs = 0
    _bsize = 0
    _lr = 0.01

    # Constructor
    def __init__(self, nepochs=num
        supe

        self._nepochs
        self._bsize = bsize
        self._lr = lr
        self._verbose
```



```
# Definition of the LeNet-5 model
def  createLayers(self)
    # First convol
                                ayer
    self.conv1
                                d(1, 3, kernel_size=5, stride=1, padding=2)
    self.re
                                ()
    sel
                                ol2d(kernel_size=2, stride=2)
                                 layer
                                2d(3, 6, kern
                                               , stride=1, padding=2)
                                ()
                                               2, stride=2)

                                                 with a value for
                                                ssification.
                                                gory.
                                                ut images and

    ut = ou
                                size(0), -1)
    out = self.fc1(ou
    out = self.fc2(out)
    return out
```

How to develop HW for accelerating the inference process of a CNN?

Implementation avail
https://git.upv.es/defa
iLenetPython

HW-based CNN acceleration in embedded systems

Graphics Processing (**GPU**)

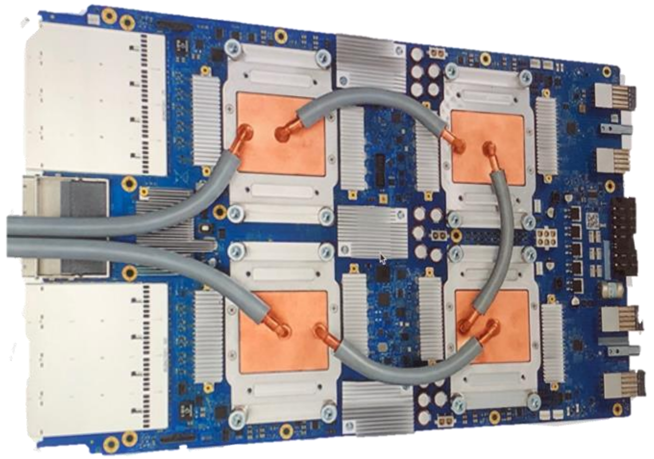
- ✓ Performance
- ✗ Energy consumption

Field-Programmable Gate Array (**FPGA**)

- ✓ Performance per watt
- ✓ Flexibility and adaptability
- ✗ Design

Tensor processing (**TPU**)

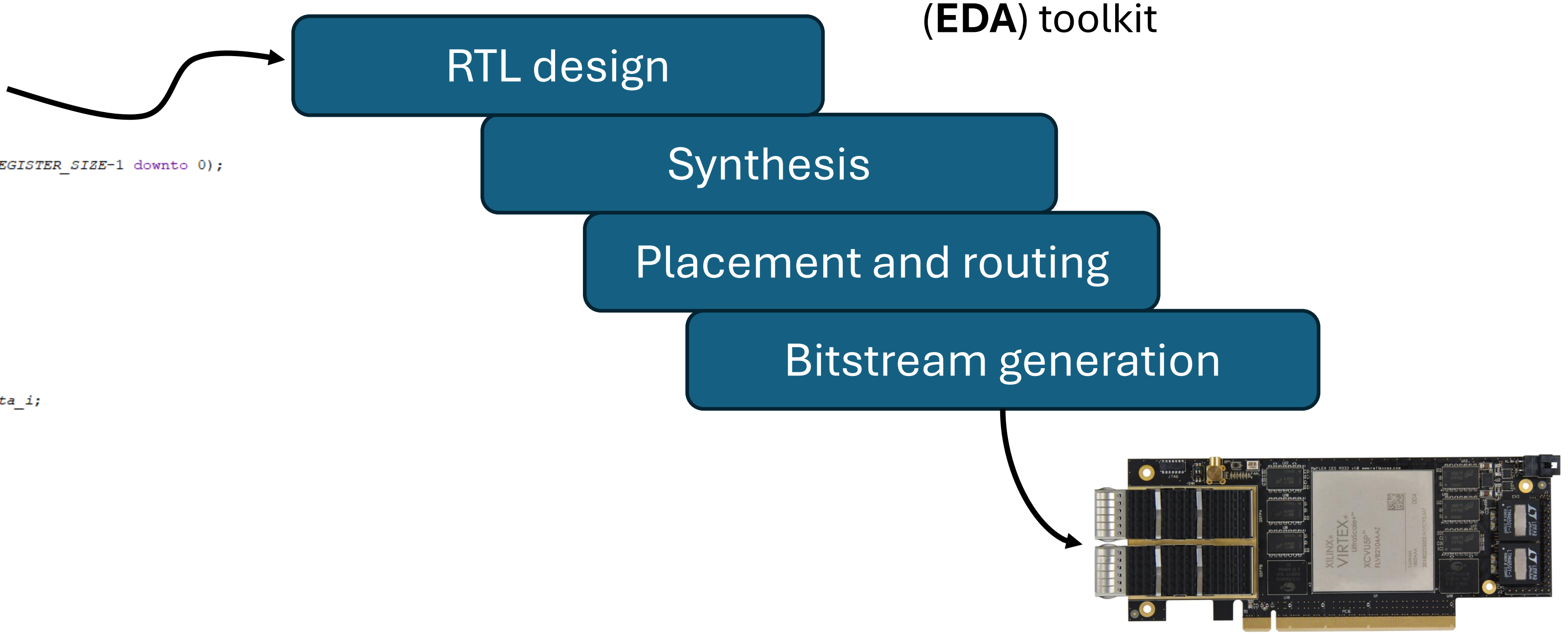
- ✓ Performance
- ✓ Energy consumption
- ✗ Flexibility



FPGA-based designs ...

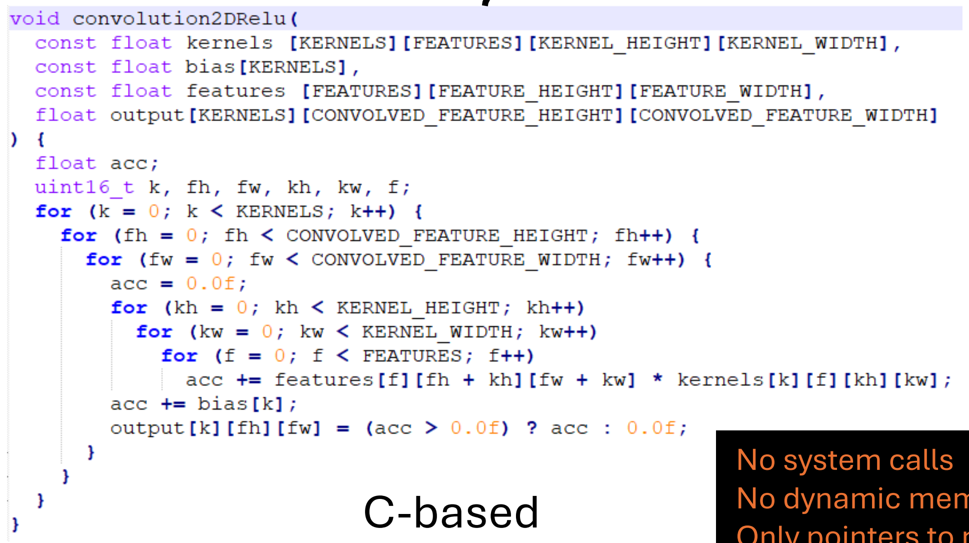
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity DualPortRegisterFile is
    Generic (ADDRESS_SIZE : POSITIVE;
             REGISTER_SIZE : POSITIVE);
    Port ( rst_i : in STD_LOGIC;
           clk_i : in STD_LOGIC;
           en_i : in STD_LOGIC;
           write_en_i : in STD_LOGIC;
           readReg1_i : in STD_LOGIC_VECTOR (ADDRESS_SIZE-1 downto 0);
           readReg2_i : in STD_LOGIC_VECTOR (ADDRESS_SIZE-1 downto 0);
           writeReg_i : in STD_LOGIC_VECTOR (ADDRESS_SIZE-1 downto 0);
           writeData_i : in STD_LOGIC_VECTOR (REGISTER_SIZE-1 downto 0);
           readData1_o : out STD_LOGIC_VECTOR (REGISTER_SIZE-1 downto 0);
           readData2_o : out STD_LOGIC_VECTOR (REGISTER_SIZE-1 downto 0));
end DualPortRegisterFile;

architecture Behavioral of DualPortRegisterFile is
    type RegFile is array (0 to (2**ADDRESS_SIZE)-1) of STD_LOGIC_VECTOR(REGISTER_SIZE-1 downto 0);
    signal registers : RegFile := (others => (others => '0'));
begin

    process(rst_i, clk_i)
    begin
        if rst_i = '1' then
            registers <= (others => (others => '0'));
        elsif rising_edge(clk_i) then
            if en_i = '1' then
                if write_en_i = '1' then
                    registers(to_integer(unsigned(writeReg_i))) <= writeData_i;
                end if;
            end if;
        end if;
    end process;

    readData1_o <= registers(to_integer(unsigned(readReg1_i)));
    readData2_o <= registers(to_integer(unsigned(readReg2_i)));

end Behavioral;
```

Electronic Design Automation (**EDA**) toolkit

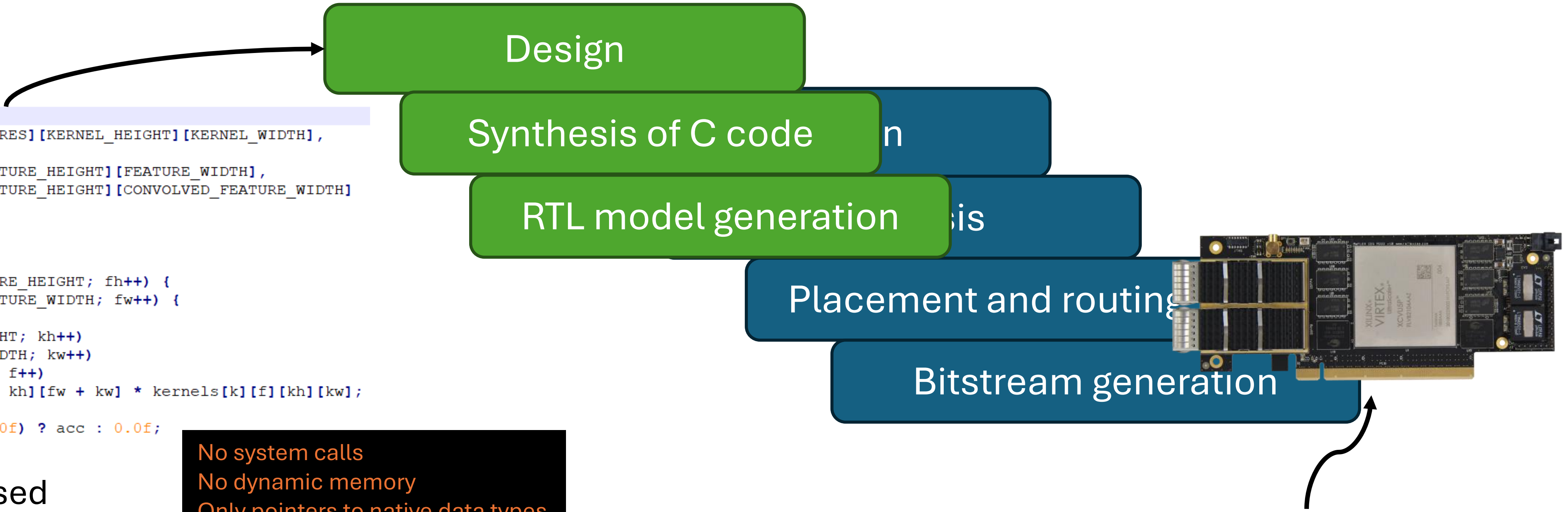


FPGA-based designs … using HLS

- Use of High-Level Synthesis (**HLS**) tools to prototype CNNs on FPGAs that have been designed using high-level programming languages



```
void convolution2DRelu(
  const float kernels [KERNELS][FEATURES][KERNEL_HEIGHT][KERNEL_WIDTH],
  const float bias[KERNELS],
  const float features [FEATURES][FEATURE_HEIGHT][FEATURE_WIDTH],
  float output[KERNELS][CONVOLVED_FEATURE_HEIGHT][CONVOLVED_FEATURE_WIDTH]
) {
  float acc;
  uint16_t k, fh, fw, kh, kw, f;
  for (k = 0; k < KERNELS; k++) {
    for (fh = 0; fh < CONVOLVED_FEATURE_HEIGHT; fh++) {
      for (fw = 0; fw < CONVOLVED_FEATURE_WIDTH; fw++) {
        acc = 0.0f;
        for (kh = 0; kh < KERNEL_HEIGHT; kh++)
          for (kw = 0; kw < KERNEL_WIDTH; kw++)
            for (f = 0; f < FEATURES; f++)
              acc += features[f][fh + kh][fw + kw] * kernels[k][f][kh][kw];
        acc += bias[k];
        output[k][fh][fw] = (acc > 0.0f) ? acc : 0.0f;
      }
    }
  }
}
```

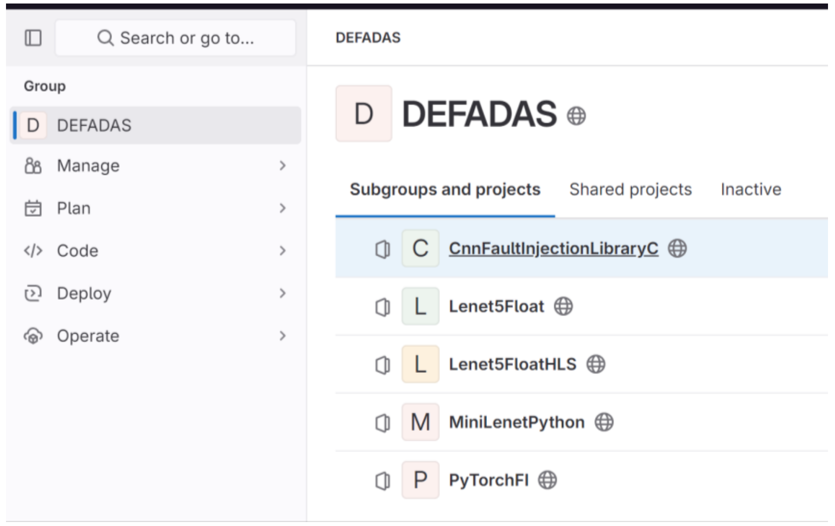


C-based Implementation with restrictions

No system calls
No dynamic memory
Only pointers to native data types
No arrays of pointers to pointers
No pointers to functions
No recursive nested calls

C-based Lenet-5 implementation

- Lenet-5 training is carried out using the Python-based model of the CNN
- The C-based implementation will use the parameters issued from the training phase
- Code publically available at https://git.upv.es/defadas
  - Lenet5FloatHLS is the version including pragmas to guide the generation of the RTL model



DEFADAS

D DEFADAS

Subgroups and projects | Shared projects | Inactive

- C CnnFaultInjectionLibraryC
- L Lenet5Float
- L Lenet5FloatHLS
- M MiniLenetPython
- P PyTorchFI

Convolution2D+ReLU in C

$$C_{x,y} = \text{cross correlation+ReLU}_{x,y} = \max(0, bias + \sum_{h=0}^{H-1}\sum_{w=0}^{W-1} kernel_{h,w} \times Input_{x+h,y+w})$$

Cross-correlation

ReLU

```
// For each kernel
conv1_K: for (k = 0; k < CONV1_KERNELS; k++) {

        // Go through features rows and columns
        conv1_FH: for (fh = 0; fh < CONV1_CONVOLVED_FEATURE_HEIGHT; fh++) {
                conv1_FW: for (fw = 0; fw < CONV1_CONVOLVED_FEATURE_WIDTH; fw++) {

                        // Reset accumulated value
                        accumulated = 0.0f;
                        // Go through the kernel rows and columns
                        conv1_KH: for (kh = 0; kh < CONV1_KERNEL_HEIGHT; kh++) {
                                conv1_KW: for (kw = 0; kw < CONV1_KERNEL_WIDTH; kw++) {

                                        // Convolve each feature with the corresponding kernel and add the result
                                        conv1_F: for (f = 0; f < CONV1_FEATURES; f++) {
                                                accumulated += input_features[f][fh + kh][fw + kw]
                                                                * input_kernels[k][f][kh][kw];
                                        }

                                }

                        }
                        // Add bias and assign result
                        accumulated += bias[k];
                        output_features[k][fh][fw] = (accumulated > 0.0f) ? accumulated : 0.0f;

                }
```

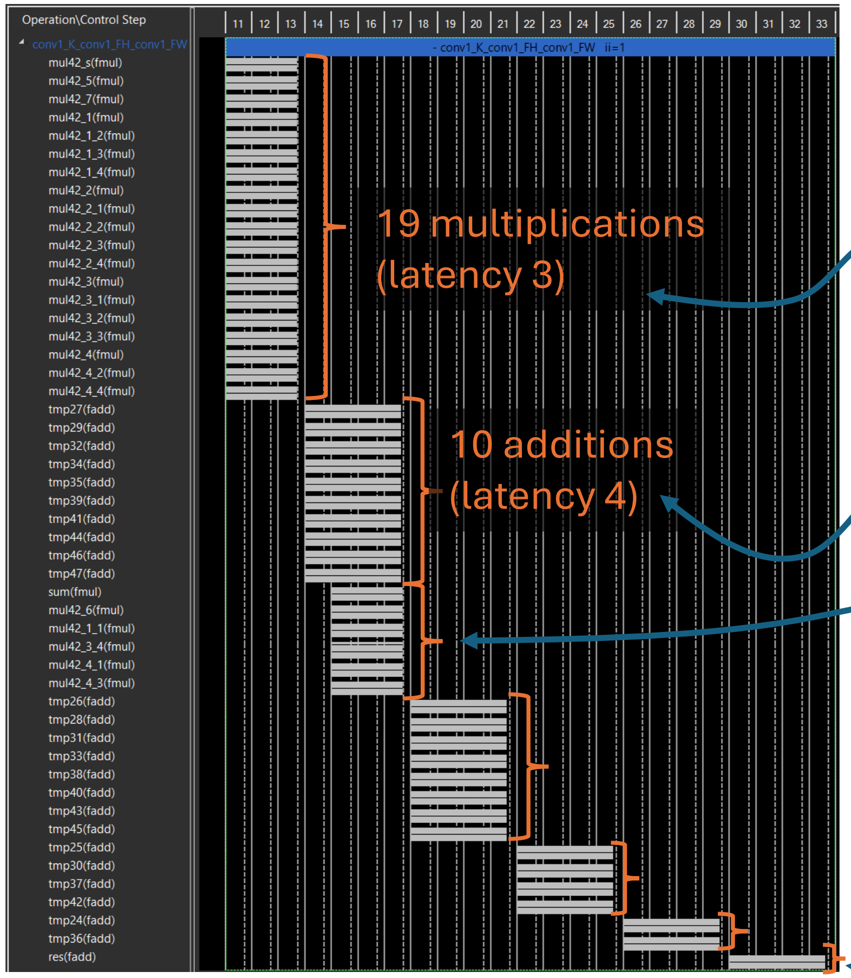
Convolution2D + ReLU layer: Synthesis

```
void convolution2DRelu(
  const float kernels [KERNELS][FEATURES][KERNEL_HEIGHT][KERNEL_WIDTH],
  const float bias[KERNELS],
  const float features [FEATURES][FEATURE_HEIGHT][FEATURE_WIDTH],
  float output[KERNELS][CONVOLVED_FEATURE_HEIGHT][CONVOLVED_FEATURE_WIDTH]
) {
#pragma HLS ARRAY_PARTITION variable=features type=cyclic factor=5 dim=3
#pragma HLS ARRAY_PARTITION variable=features type=cyclic factor=5 dim=2
#pragma HLS ARRAY_PARTITION variable=kernels type=complete dim=4
#pragma HLS ARRAY_PARTITION variable=kernels type=complete dim=3
float acc;
  uint16_t k, fh, fw, kh, kw, f;
  for (k = 0; k < KERNELS; k++) {
    for (fh = 0; fh < CONVOLVED_FEATURE_HEIGHT; fh++) {
      for (fw = 0; fw < CONVOLVED_FEATURE_WIDTH; fw++) {
#pragma HLS PIPELINE II=1
        acc = 0.0f;
        for (kh = 0; kh < KERNEL_HEIGHT; kh++)
          for (kw = 0; kw < KERNEL_WIDTH; kw++)
            for (f = 0; f < FEATURES; f++)
               acc += features[f][fh + kh][fw + kw] * kernels[k][f][kh][kw];
        acc += bias[k];
        output[k][fh][fw] = (acc > 0.0f) ? acc : 0.0f;
      }
    }
  }
}
```

rearrange data in arrays
to enable simultaneous access
to kernels and features data during
execution

Set **I**nitiation **I**nterval to 1 cycle
(all possbile multiplications and additions in parallel)

Convolution2D + ReLU layer: Synthesis



5x5 kernel
- **25 multiplications (19+6)**
- **25 additions (10+8+4+2+1)**

A kernel-based convolution takes 23 cycles

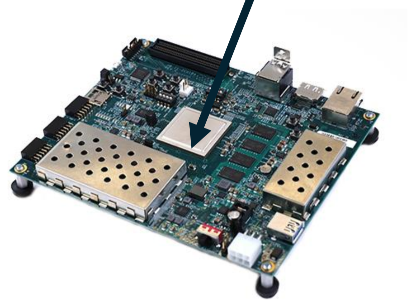
Prototyping a HW-based CNN accelerator using an FPGA

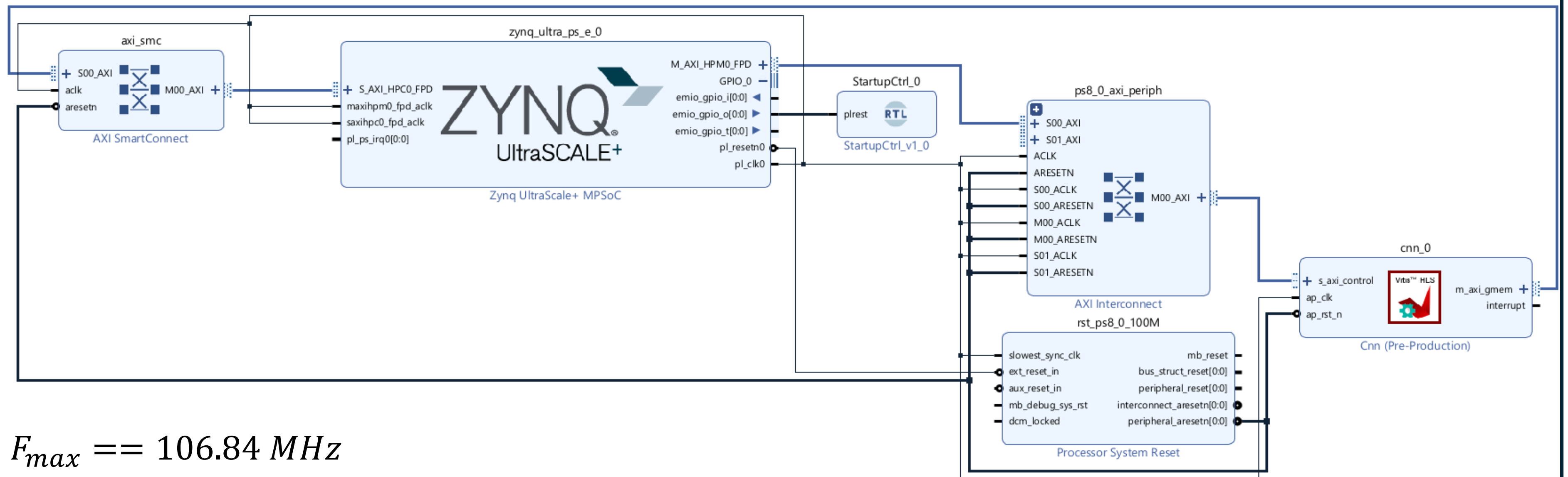
- Workflow

C-based Lenet-5 (Lenet5FloatHLS) → RTL Lenet-5 model → Zynq Ultrascale+ BitStream

AMD Vitis: Simulation, synthesis, co-simulation and RTL generation

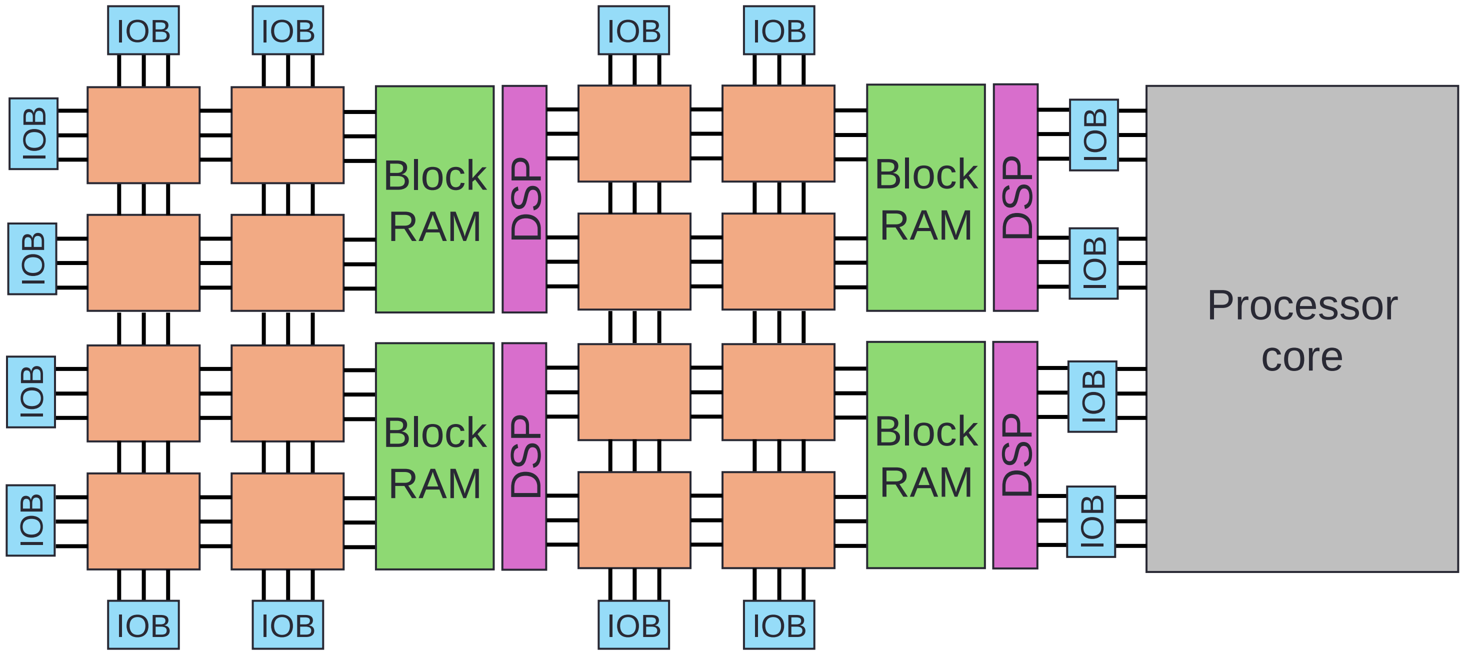
AMD Vivado Design Suite: Synthesis, placement and routing


ZCU104 prototyping board

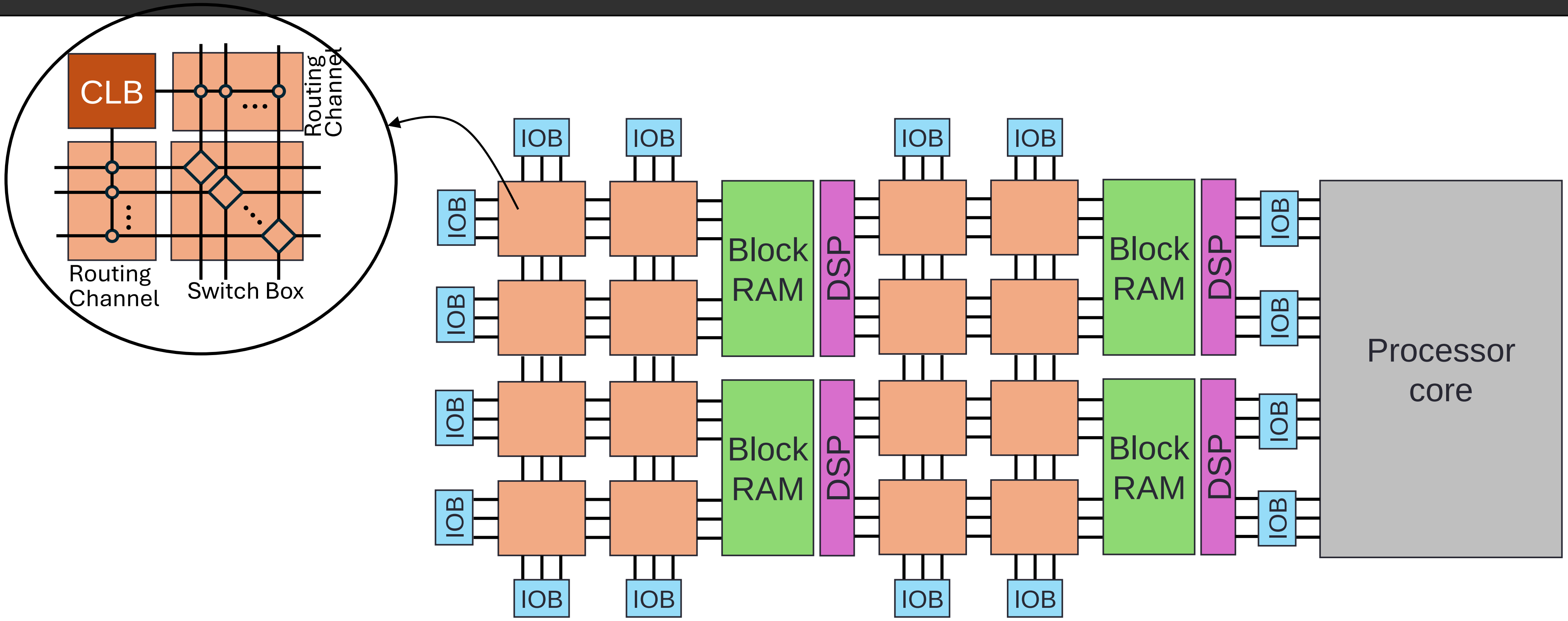


$F_{max} == 106.84\ MHz$

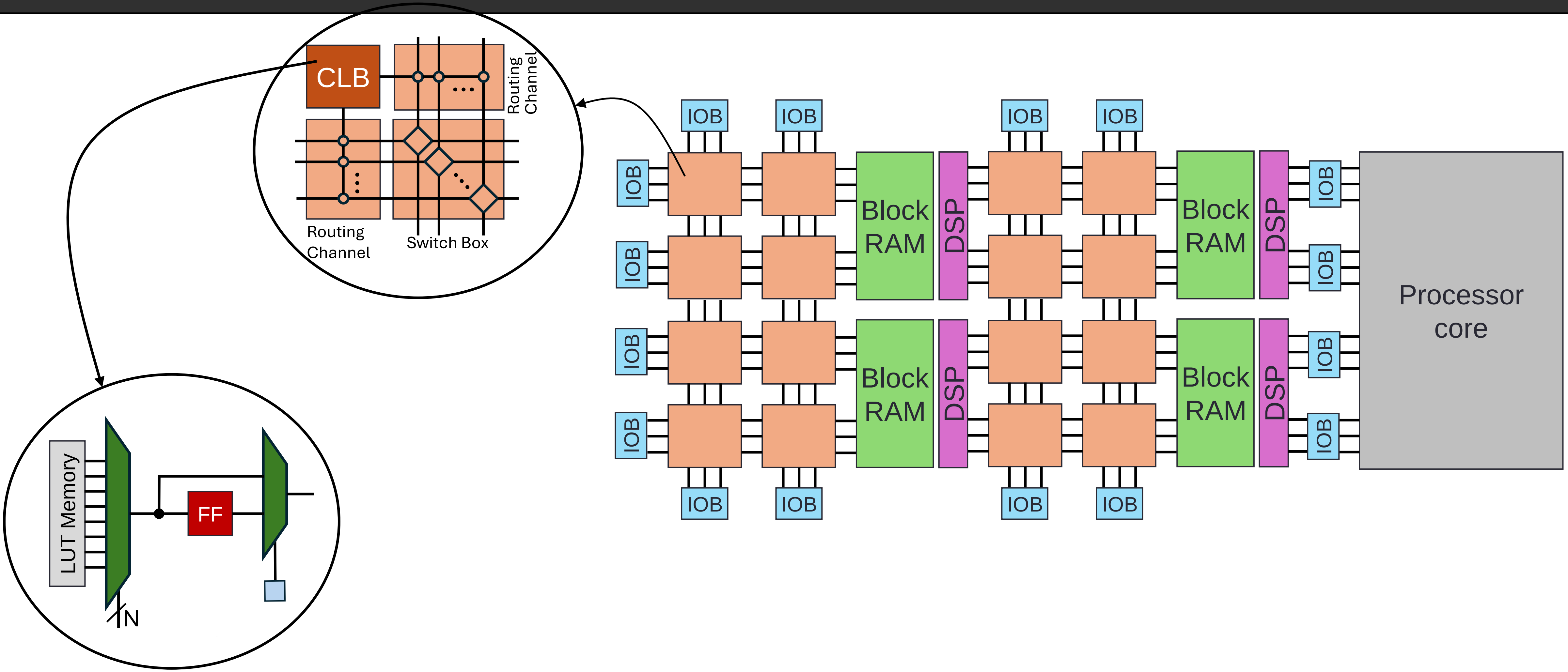
Architecture of FPGAs



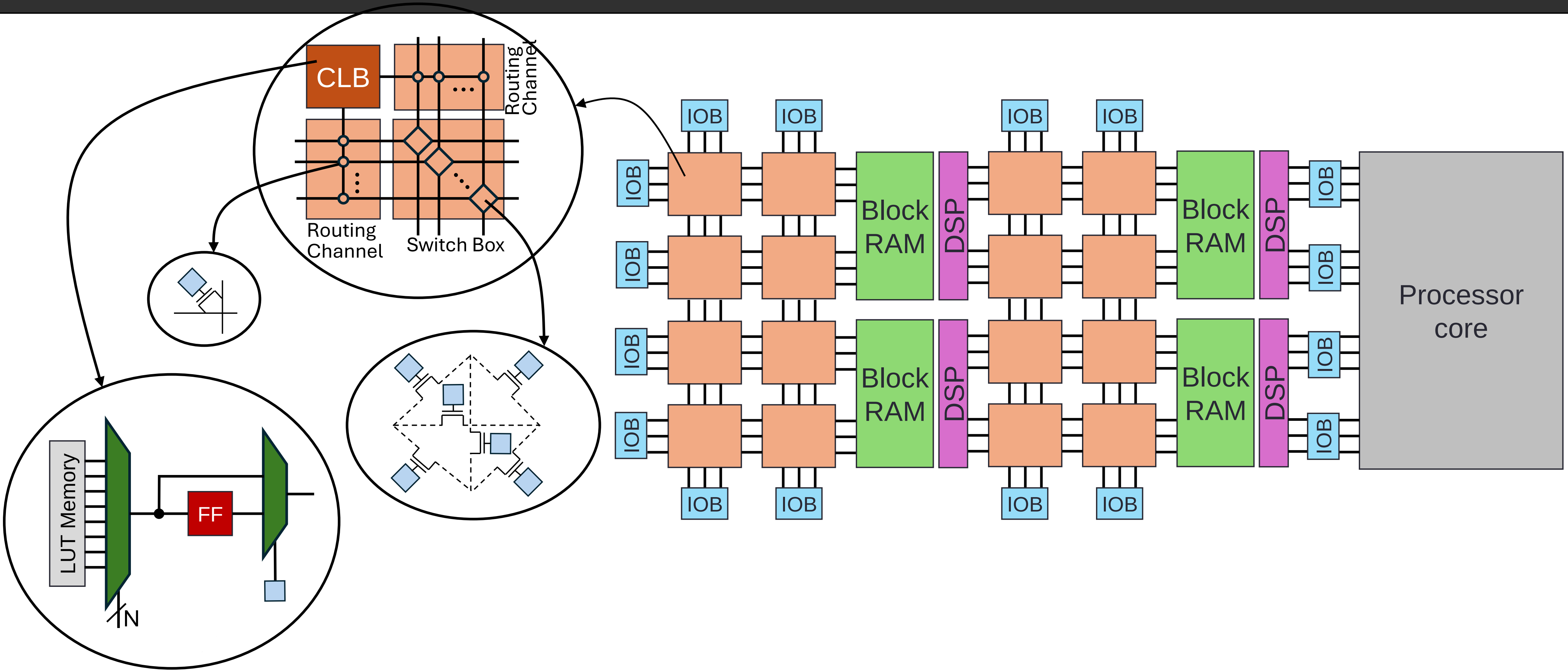
Architecture of FPGAs



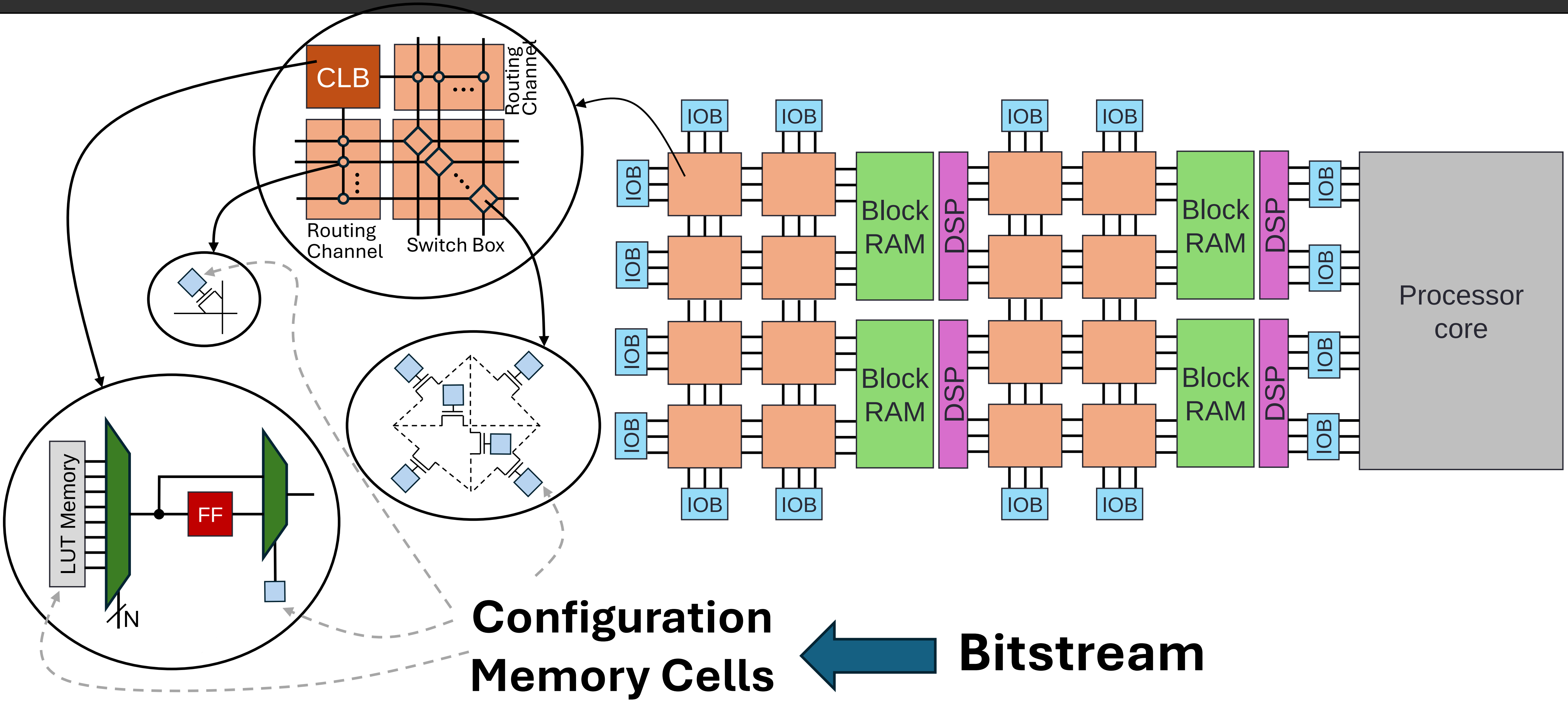
Architecture of FPGAs



Architecture of FPGAs



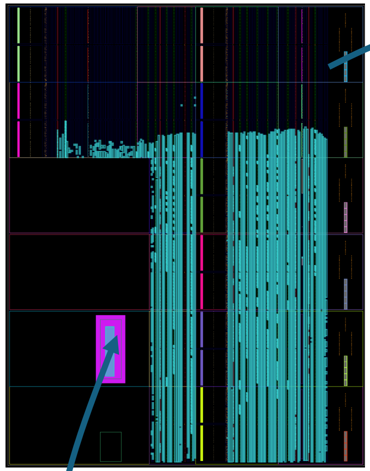
Architecture of FPGAs



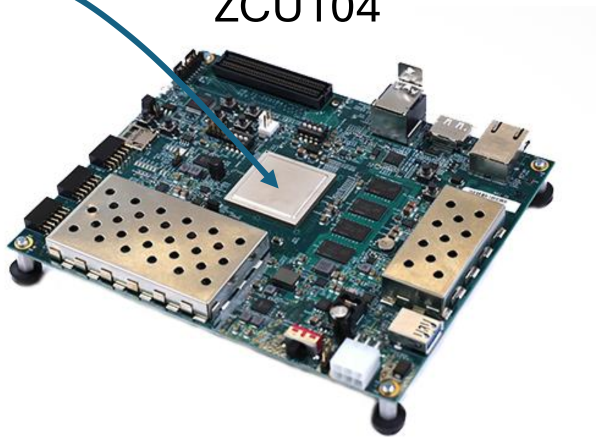
Configuration Memory Cells ← **Bitstream**

Results: Performance evaluation

Xilinx XCZU7EV-2FFVC1156



ZCU104



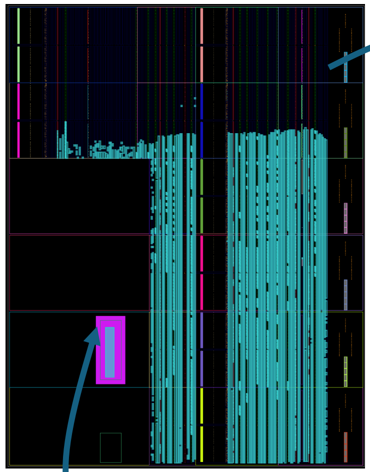
ARM Cortex-A53

Intel i7-4690

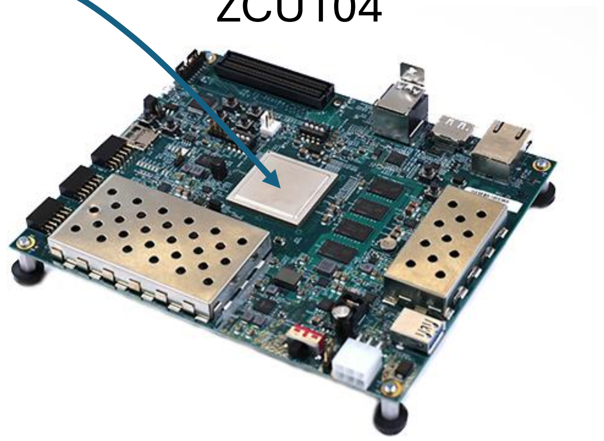
Lenet-5 implemented in C

Results: Performance evaluation

Xilinx XCZU7EV-2FFVC1156



ZCU104



ARM Cortex-A53

Intel i7-4690

Lenet-5 implemented in C

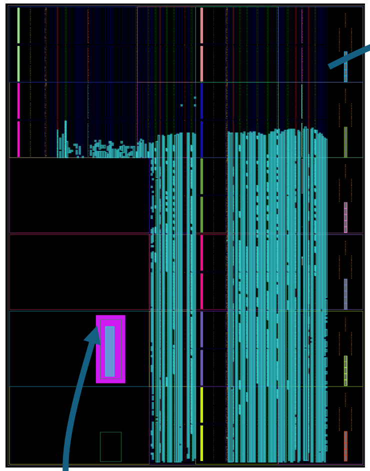
Component	Clock frequency	Execution time (100 images)	Power consumption (estimated)
ZCU104	106 MHz	62.99 ms	6.16 W
Intel i7-4790	3800 MHz	93.75 ms	47.50 W
ARM Cortex-A53	1200 MHz	1271.74 ms	2.74 W

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

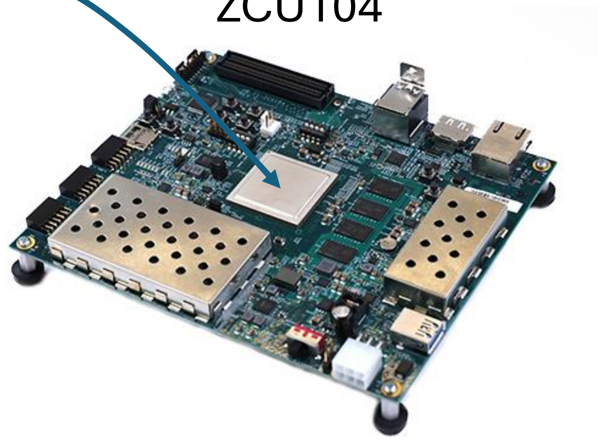
Institute ITACA Information and Communication Technologies

Results: Performance evaluation

Xilinx XCZU7EV-2FFVC1156



ZCU104



ARM Cortex-A53

Intel i7-4690

Lenet-5 implemented in C

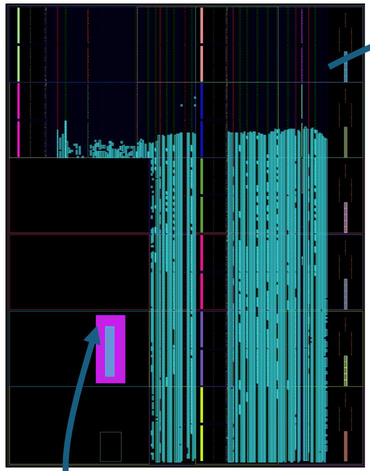
Component	Clock frequency	Execution time (100 images)	Power consumption (estimated)
ZCU104	106 MHz	62.99 ms	6.16 W
Intel i7-4790	3800 MHz	93.75 ms	47.50 W
ARM Cortex-A53	1200 MHz	1271.74 ms	2.74 W

LeNet-5 → 5.43 ms → 3.42 W

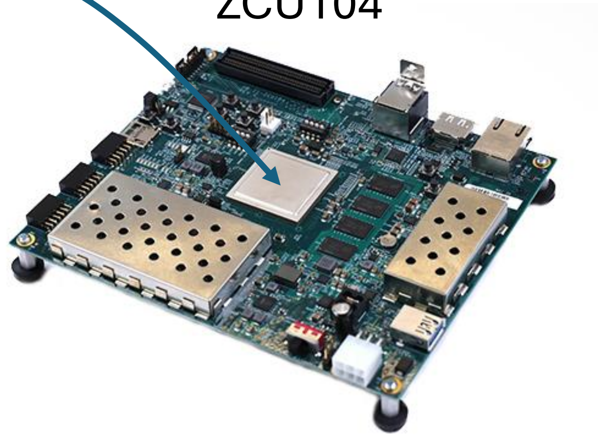
ARM Cortex-A53 → 57.56 ms → 2.74 W

Results: Performance evaluation

Xilinx XCZU7EV-2FFVC1156



ZCU104



ARM Cortex-A53

Intel i7-4690

Lenet-5 implemented in C

Component	Clock frequency	Execution time (100 images)	Power consumption (estimated)
ZCU104	106 MHz	62.99 ms	6.16 W
Intel i7-4790	3800 MHz	93.75 ms	47.50 W
ARM Cortex-A53	1200 MHz	1271.74 ms	2.74 W

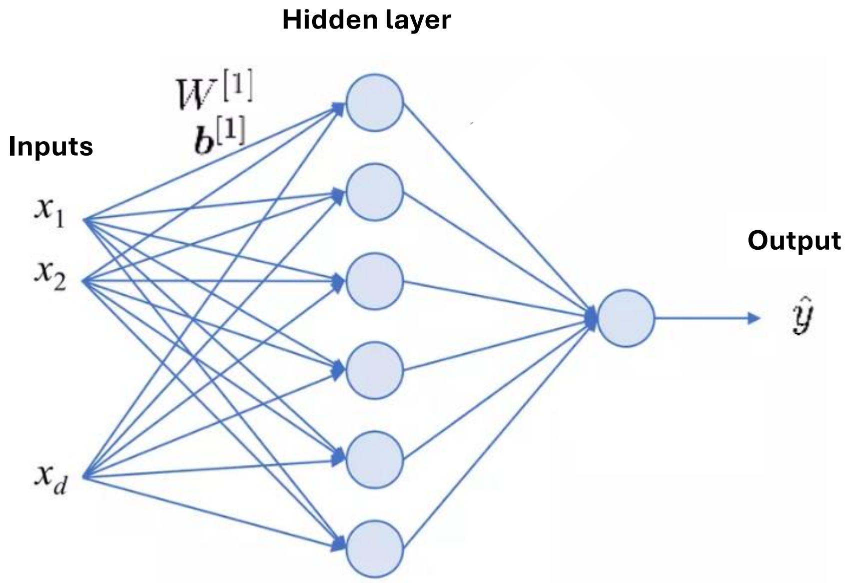
ZCU	Speed-up	Power consumption (estimated)
vs Intel i7-4790	x1.48	/ 7.71
vs ARM Cortex-A53	x20.19	x2.24

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Institute ITACA Information and Communication Technologies

And what about quantization?

- Representation for weights and activations to a lower precision data type
  - (+) Reduce memory footprint
  - (+) Speed-up computation and reduce power consumption
  - (-) Possible accuracy reduction
- Common data types
  - BF16 (currently the replacement of FP32)
  - INT16/INT8 (interesting for edge computing)
- Types of quantization
  - Dynamic (on-the-fly quantization) vs **Static** (pre-computed) quantization
  - Quantization-aware training (more accurate, but need of access to training dataset and platform) vs **Post-training quantization** (not resource intensive, but may affect accuracy)
  - **Affine** (better use of INT8 range) vs Symmetric Quantization (better performance but higher induced errors in dequantization)

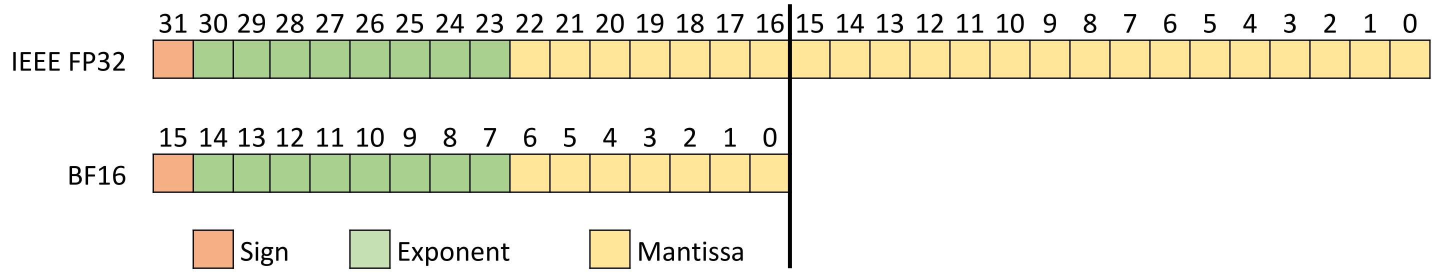


$$y(j) = \underbrace{B(j)}_{\text{bias}} + \sum_{i=0}^{I} \underbrace{x(i)}_{\text{input}} \times \underbrace{w(i,j)}_{\text{weights}}$$

Quantize

Using BF16

- Post-training Static quantization (FP32→BF16)

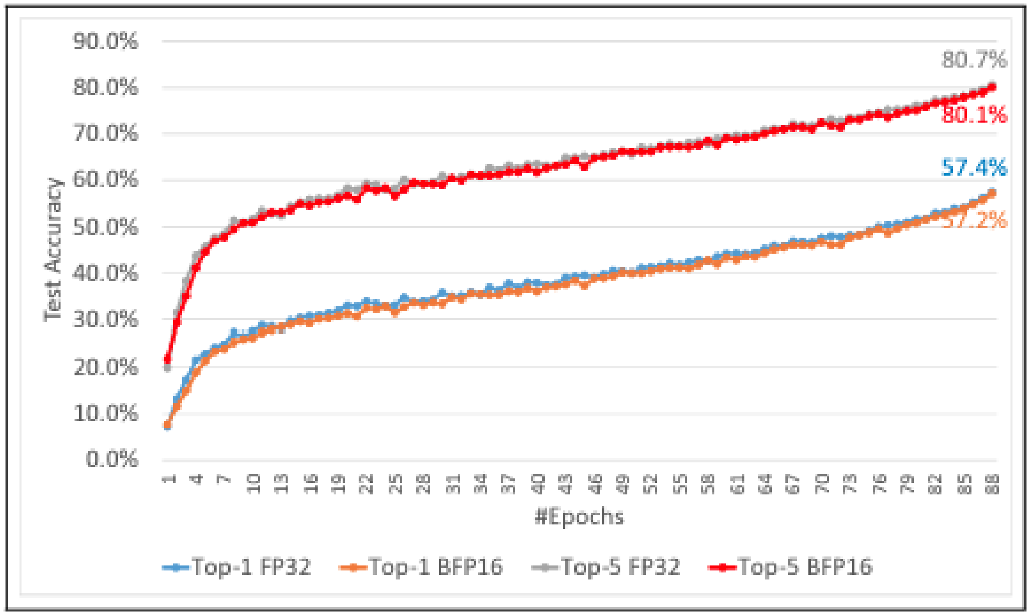


$$value = (-1)^{sign} \, x \, 2^{(E-127)} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

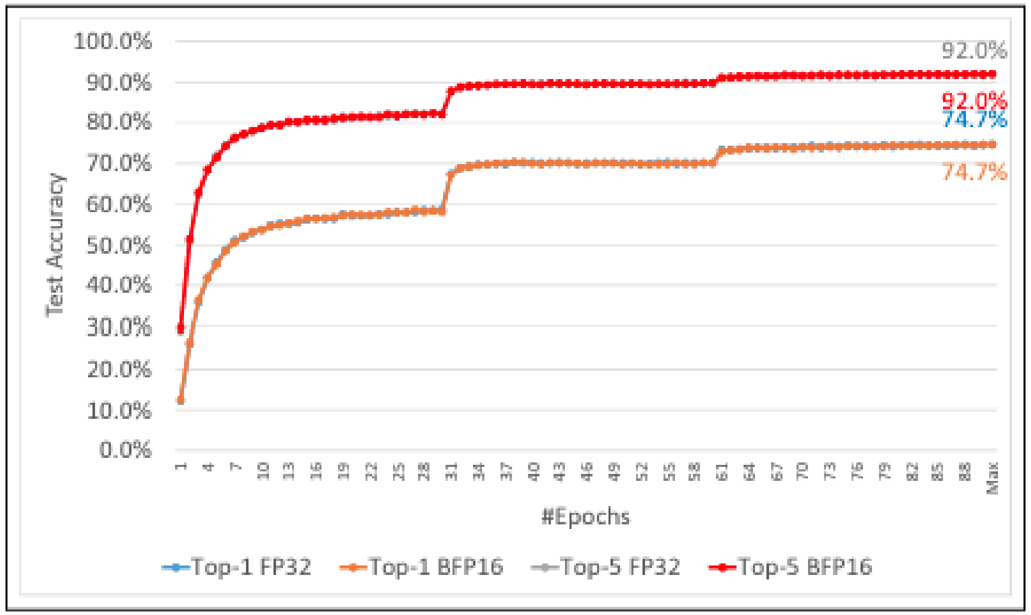
$$value = (-1)^{sign} \, x \, 2^{(E-127)} \times \left(1 + \sum_{i=1}^{7} b_{7-i} 2^{-i}\right)$$

  - Halves the amount of memory required by parameters
  - Improves performance of memory-bandwidth-bound FP operations
  - Still computing in FP, so does not significantly affect CNN accuracy

FP32 vs BF16



(a) AlexNet



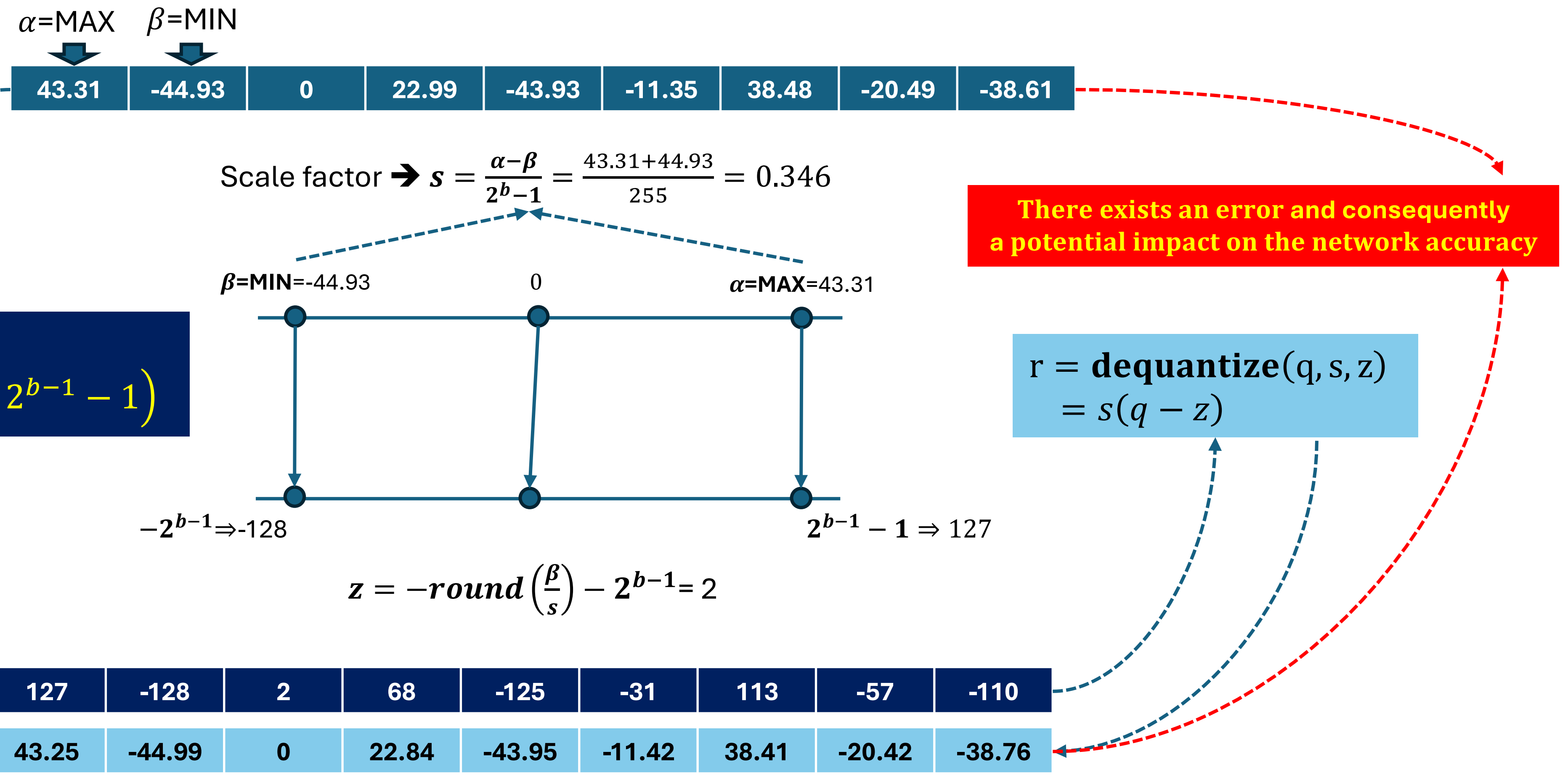
(b) ResNet-50

Source:

D. D. Kalamkar *et al.*, "A study of BFLOAT16 for deep learning training," *arXiv (Cornell University)*, May 2019, [Online]. Available: https://arxiv.org/pdf/1905.12322.pdf

Using Integers

- Example using affine (asymmetric) quantization and FP32/BF16→INT8



43.31	-44.93	0	22.99	-43.93	-11.35	38.48	-20.49	-38.61

Scale factor ➔ $s = \frac{\alpha - \beta}{2^b - 1} = \frac{43.31+44.93}{255} = 0.346$

$$z = -round\left(\frac{\beta}{s}\right) - 2^{b-1} = 2$$

$$q = \mathbf{quantize}(r, b, s, z) = \\ = \mathrm{clamp}\left(\mathrm{round}\left(\frac{r}{s}\right) + z; -2^{b-1}; 2^{b-1} - 1\right)$$

$$r = \mathbf{dequantize}(q, s, z) \\ = s(q - z)$$

There exists an error and consequently a potential impact on the network accuracy

127	-128	2	68	-125	-31	113	-57	-110
43.25	-44.99	0	22.84	-43.95	-11.42	38.41	-20.42	-38.76

Operations with quantized numbers*

$$y(j) = B(j) + \sum_{i=0}^{I} x(i) \times w(i,j)$$

biases are adjusted so that → $z_b = 0$ and $S_b = S_x \text{ x } S_w$
and remember the dequantization formula → $r_? = s_?(q_? \text{-} z_?)$

$$s_y(q_y\text{-}z_y) = S_b \text{ x } q_b + \sum_{i=0}^{I} S_x(q_x - zx) \times S_w(q_w - zw)$$

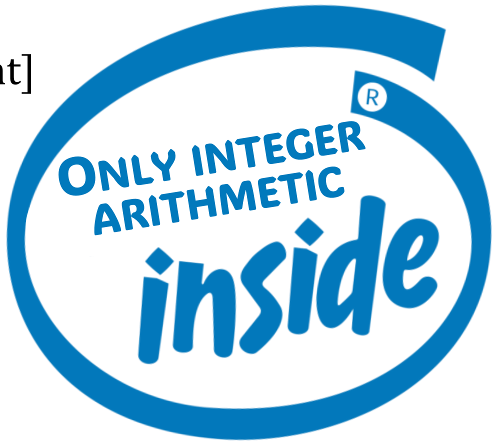
$$q_y = \frac{S_x \text{ x } S_w}{S_y} q_b + \frac{S_x \text{ x } S_w}{S_y}\left[\sum_{i=0}^{I} (q_x - zx) \times (q_w - zw)\right] + z_y, \text{ where } M_0 = \frac{S_x \text{ x } S_w}{S_y} \in [0.1[$$

Tip: $M_0 = 0.111 \rightarrow M'_0 = 2^3 \text{x } M_0$ [shift left] $= 111 \rightarrow M_0 = M'_0 / 2^3$ [shift right]

So $M'_0 = 2^{32} M_0$

Quantized output computation

$$\boxed{q_y = \frac{M'_0}{2^{32}} \left(q_b + \left[\sum_{i=0}^{I} (q_x - zx) \times (q_w - zw)\right]\right) + z_y}$$



* [CoRR 2021] Markus Nagel et al. "A White Paper on Neural Network Quantization", CoRR abs/2106.08295 (2021)

C-based implementation

$$q_y = \frac{M'0}{2^{32}} \left(q_b + \left[\sum_{i=0}^{I} (q_x - zx) \times (q_w - zw) \right] \right) + z_y$$

```c
// Applies a linear transformation
// https://pytorch.org/docs/stable/generated/torch.nn.Linear.html#torch.nn.Linear
void fullyConnected_1(
            const float input_features [FC1_INPUT_FEATURES],
            const float input_weights [FC1_FEATURES][FC1_INPUT_FEATURES],
            const float bias[FC1_FEATURES],
            float output_features[FC1_FEATURES]) {

      float accumulated;

      uint16_t f;
      uint16_t nif;
      uint16_t nv;

      // Go through all the values of that feature
      fc1_F: for (f = 0; f < FC1_FEATURES; f++) {

            accumulated = 0.0f;

            // For each feature
            fc1_NIF: for (nif = 0; nif < FC1_INPUT_FEATURES; nif++) {

                  accumulated += input_features[nif] * input_weights[f][nif];
            }

            output_features[f] = accumulated + bias[f];
      }

}
```

FP-based computation

Result of previous layer

precomputed values

Output

```c
void fullyConnected(
    const uint8_t q_x[FEATURES],
    const int8_t z_x,
    const int8_t q_w[OUTPUTS][FEATURES],
    const int8_t z_w[OUTPUTS],
    const uint32_t M[FEATURES],
    const int32_t q_b[FEATURES],
    uint8_t q_y [OUTPUTS],
    const int8_t z_y ) {

  int32_t acc;
  int64_t mXacc, y;
  uint16_t j, i;

  for (j = 0; j < OUTPUTS; j++) {
    acc = 0;
    for (i = 0; i < FEATURES; i++)
      acc += (q_x[i] - z_x) * (q_w[j][i] - z_w[j]);
    acc += q_b[j];
    mXacc = (int64_t)m[j] * acc;
    y = (mXacc>>31 & 0x1) ? (mXacc>>32) + 1 : mXacc>>32;
    y += z_y;
    if (y < 0) q_y[j] = (uint8_t)0;
    else if (y > 255) q_y[j] = (uint8_t)255;
    else q_y[j] = (uint8_t)y;
  }
}
```

INT-based computation

Outline

- ❑ Understanding HW accelerators for CNNs: Prototyping a FP32 /INT8 CNN on a FPGA: Lenet-5 as a case study
- ❑ **Robustness evaluation of FP-based CNNs using fault injection: methodology and lessons learnt**
- ❑ In-Memory Zero-Space Protection of FP-based CNNs using ECCs: methodology and lessons learnt
- ❑ Conclusions

Robutsness evalutation

- Goal:
  Estimate the impact of faults on the CNN accuracy
- Targets:
  CNN parameter bits
- Fault Injection Methodology
  - Which fault model?
  - Which fault injection process?
  - How many faults to inject?

Robutsness evalutation

- Goal:
  Estimate the impact of faults on the CNN accuracy
- Targets:
  CNN parameter bits
- Fault Injection Methodology
  - **Which fault model?**
  - Which fault injection process?
  - How many faults to inject?

Basics on fault models

- A fault model mimics the effect of events provoking a flip in a memory element or a logic gate (soft errors, wear-out, crosstalk, voltage surges, ...)
  - Permanent fault models → Stuck-at-1 / Stuck-at-0 (bits remain 1/0 and the effect remains persistent until replacing the component)
  - Transient model → Bit-flips (bit flips, but the effect can be simply fixed by rewriting the bit)
- CNN parameters are typically store in HW accelerators' internal buffers, which are not protected by memory ECCs

Already known facts

- Intrinsic robustness of CNNs to bitflips and stuck-at faults thanks to the information redundancy existing in their parameters
- INT8 CNNs are more robust than FP32/BF16 CNNs to the occurrence of single bitflips, since they may induce higher value-related effects on network parameters
  - FP32: 22,84 → 01000001101101101011100001010001
    → 01**1**00001101101101011100001010001 (**421323637458275900000!!**)
  - INT8: 68 → 01000100
    → 01**1**00100 (100)
- This may not be true in the case of multiple bitflips
  - FP32: 22,84 → 01000001101101101011100001010001
    → 011000011011011010111000**10101110** (no effect on CNN accuracy)
  - INT8: 68 → 01000100
    → **10101110** (**All bits changed!!** → potential effect on CNN accuracy

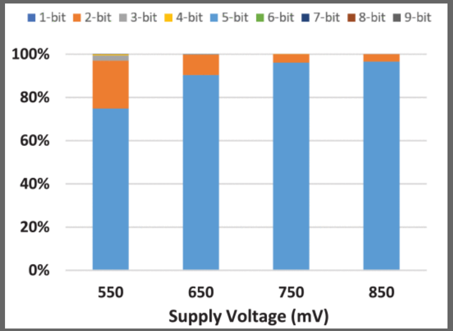
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Institute ITACA Information and Communication Technologies

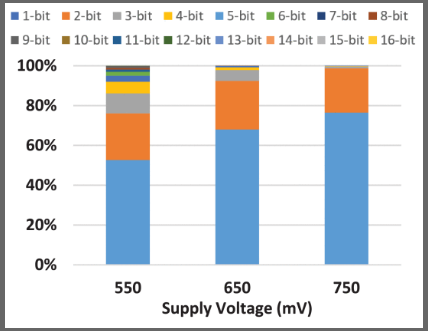
Importance of multi-bitflips

- Accidental faults: The number of bits altered by a single ionizing particle increases as CMOS integration does

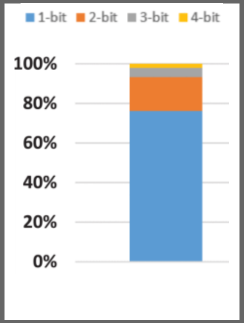
N. J. Pieper et al., "Study of Multicell Upsets in SRAM at a 5-nm Bulk FinFET Node,“ in *IEEE Transactions on Nuclear Science*, vol. 70, no. 4, pp. 401-409, April 2023



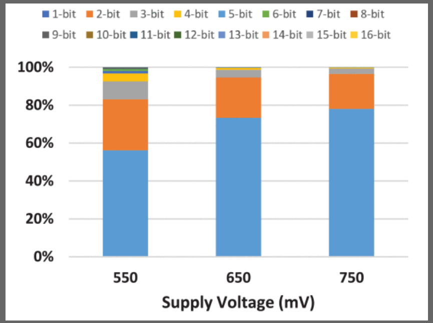
alpha particles



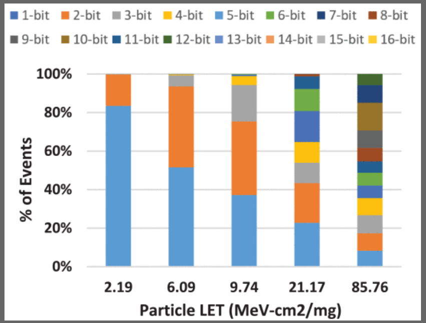
14-MeV neutrons



terrestrial neutrons



thermal neutrons



heavy ions

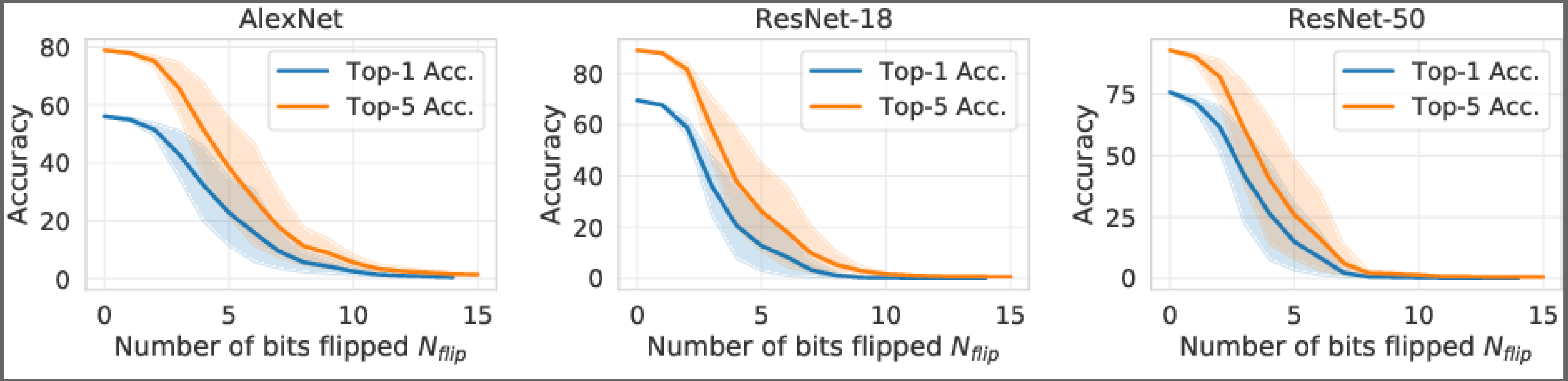
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Institute ITACA Information and Communication Technologies

Importance of multi-bitflips

- Malicious faults: A reduced number of flipped bits in parameters can lead a CNN to crush

Adnan Siraj Rakin, Zhezhi He, and Deliang Fan, "Bitflip attack: Crushing neural network with progressive bit search" in 2019 IEEE/CVF International Conference on Computer Vision (ICCV), pp. 1211–1220, 2019.

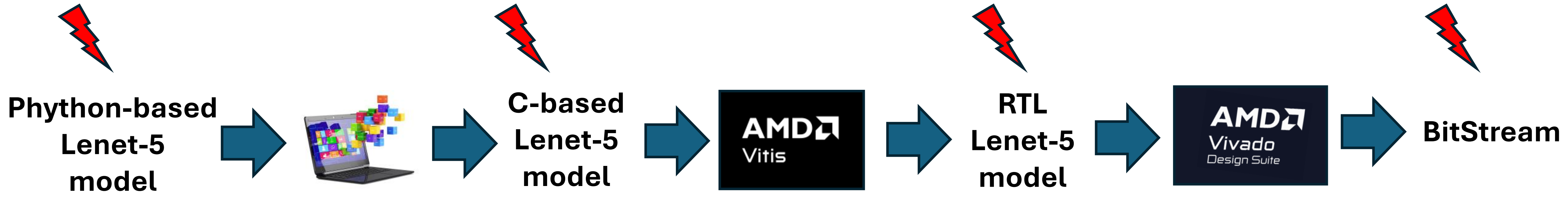


Robutsness evalutation

- Goal:
  Estimate the impact of faults on the CNN accuracy
- Targets:
  CNN parameter bits
- Fault Injection Methodology
  - Which fault model? Multiple faults
  - **Which fault injection process should be followed?**
  - How many faults to inject?

General fault injection process

1. Get access to the parameter bits
2. Alter the target bit(s) using a mask
3. Update the corresponding tensor
4. Launch the inference process and take note of the inference accuracy



Injection of HW faults

- Use of a fault injection mask where bits to alter are set to 1 or 0
  - Transient faults (bit-flips)
    - 0/1 → 1/0 : *bit* XOR 1
  - Permanent faults (stuck-at-X)
    - 0/1 → stuck-at-0 → 0: *bit* AND 0
    - 0/1 → stuck-at-1 → 1: *bit* OR 1
- Injection pattern will differentiate the type of fault
  - Locality for accidental faults
  - Potential dispersion for malicious faults

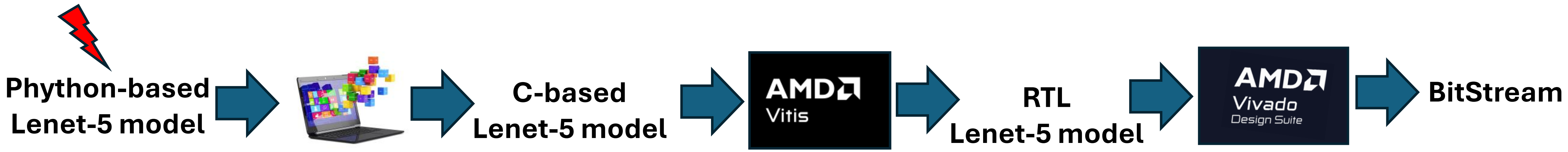
Levels of injection

- FP32
  - Python
  - C
- INT8
  - Python
  - C
- RTL
- FPGA

Results must be consistent despite the level of injection considered

Levels of injection

- FP32
  - **Python**
  - C
- INT8
  - Python
  - C
- RTL
- FPGA



Fault injection into CNNs – Python

```
model = get_model(“lenet5", weights="DEFAULT") # Obtain model
model.eval() # Set model in evaluation mode
inject_fault(model, “conv1.weight”, 0, 30, “Bit-Flip”) # Inject fault
```

Fault injection into CNNs – Python

```
def inject_fault(model, tensor, element, bit, fault):
  dict = model.state_dict()
  shape = dict[tensor].shape
  flat = dict[tensor].flatten()
  int_value = flat[element].view(torch.int)
  mask = (0x00000001 << bit)
  if fault == "Stuck-at-1":
    faulty = torch.bitwise_or(int_value, mask)
  elif fault == "Stuck-at-0":
    faulty = torch.bitwise_and(int_value, ~mask)
  elif fault == "Bit-Flip":
    faulty = torch.bitwise_xor(int_value, mask)
  else:
    pass
  flat[element] = faulty.view(torch.float)
  dict[tensor] = flat.view(shape)
```

```
model = get_model("lenet5", weights="DEFAULT") # Obtain model
model.eval() # Set model in evaluation mode
inject_fault(model, "conv1.weight", 0, 30, "Bit-Flip") # Inject fault
```

Fault injection into CNNs – Python

```
def inject_fault(model, tensor, element, bit, fault):
  dict = model.state_dict()
  shape = dict[tensor].shape
  flat = dict[tensor].flatten()
  int_value = flat[element].view(torch.int)
  mask = (0x00000001 << bit)
  if fault == "Stuck-at-1":
    faulty = torch.bitwise_or(int_value, mask)
  elif fault == "Stuck-at-0":
    faulty = torch.bitwise_and(int_value, ~mask)
  elif fault == "Bit-Flip":
    faulty = torch.bitwise_xor(int_value, mask)
  else:
    pass
  flat[element] = faulty.view(torch.float)
  dict[tensor] = flat.view(shape)
```

Get the dictionary

```
OrderedDict({
  'conv1.weight': tensor([[[[ -0.3299266695976257, -
0.2522581219673157,...],...]]]]),
  'conv1.bias': tensor([ 0.0525683201849461,  0.1234361827373505,...]),
  'conv2.weight': tensor([[[[ 0.0305016357451677, -
0.1155040338635445,...]...]]]]),
  'conv2.bias': tensor([ 0.1148738339543343, -0.0630381628870964,...]),
  'fc1.weight': tensor([[ 0.0614890158176422, 0.0941545143723488,...]...]]),
  'fc1.bias': tensor([ -0.0740724205970764, 0.0834083408117294,...]),
  'fc2.weight': tensor([[ -0.0574139244854450, 0.0175603814423084,...]...]]),
  'fc2.bias': tensor([-0.0256344508379698,  0.0859057456254959, ...])
})
```

```
model = get_model("lenet5", weights="DEFAULT") # Obtain model
model.eval() # Set model in evaluation mode
inject_fault(model, "conv1.weight", 0, 30, "BF") # Inject fault
```

Fault injection into CNNs – Python

```
def inject_fault(model, tensor, element, bit, fault):
   dict = model.state_dict()
   shape = dict[tensor].shape
   flat = dict[tensor].flatten()
   int_value = flat[element].view(torch.int)
   mask = (0x00000001 << bit)
   if fault == "Stuck-at-1":
      faulty = torch.bitwise_or(int_value, mask)
   elif fault == "Stuck-at-0":
      faulty = torch.bitwise_and(int_value, ~mask)
   elif fault == "Bit-Flip":
      faulty = torch.bitwise_xor(int_value, mask)
   else:
      pass
   flat[element] = faulty.view(torch.float)
   dict[tensor] = flat.view(shape)
```

Get the shape of the tensor to reshape it later

```
OrderedDict({
  'conv1.weight': tensor([[[[ -0.3299266695976257, -0.2522581219673157,...],...]]]]),
  'conv1.bias': tensor([ 0.0525683201849461,  0.1234361827373505,...]),
  'conv2.weight': tensor([[[[ 0.0305016357451677, -0.1155040338635445,...]...]]]]),
  'conv2.bias': tensor([ 0.1148738339543343, -0.0630381628870964,...]),
  'fc1.weight': tensor([[ 0.0614890158176422, 0.0941545143723488,...]...]]),
  'fc1.bias': tensor([ -0.0740724205970764, 0.0834083408117294,...]),
  'fc2.weight': tensor([[ -0.0574139244854450, 0.0175603814423084,...]...]]),
  'fc2.bias': tensor([-0.0256344508379698,  0.0859057456254959, ...])
})

torch.Size([3, 1, 5, 5])
```

```
model = get_model("lenet5", weights="DEFAULT") # Obtain model
model.eval() # Set model in evaluation mode
inject_fault(model, "conv1.weight", 0, 30, "Bit-Flip") # Inject fault
```

Fault injection into CNNs – Python

```
def inject_fault(model, tensor, element, bit, fault):
  dict = model.state_dict()
  shape = dict[tensor].shape
  flat = dict[tensor].flatten()
  int_value = flat[element].view(torch.int)
  mask = (0x00000001 << bit)
  if fault == "Stuck-at-1":
    faulty = torch.bitwise_or(int_value, mask)
  elif fault == "Stuck-at-0":
    faulty = torch.bitwise_and(int_value, ~mask)
  elif fault == "Bit-Flip":
    faulty = torch.bitwise_xor(int_value, mask)
  else:
    pass
  flat[element] = faulty.view(torch.float)
  dict[tensor] = flat.view(shape)
```

Flatten the tensor to access its elements

```
OrderedDict({
  'conv1.weight': tensor([[[[ -0.3299266695976257, -0.2522581219673157,...],...]]]),
  'conv1.bias': tensor([ 0.0525683201849461,  0.1234361827373505,...]),
  'conv2.weight': tensor([[[[ 0.0305016357451677, -0.1155040338635445,...]...]]]]),
  'conv2.bias': tensor([ 0.1148738339543343, -0.0630381628870964,...]),
  'fc1.weight': tensor([[ 0.0614890158176422, 0.0941545143723488,...]...]]),
  'fc1.bias': tensor([ -0.0740724205970764, 0.0834083408117294,...]),
  'fc2.weight': tensor([[ -0.0574139244854450, 0.0175603814423084,...]...]]),
  'fc2.bias': tensor([-0.0256344508379698,  0.0859057456254959, ...])
})

torch.Size([3, 1, 5, 5])

tensor([-0.3299266695976257, -0.2522581219673157,...])
```

```
model = get_model("lenet5", weights="DEFAULT") # Obtain model
model.eval() # Set model in evaluation mode
inject_fault(model, "conv1.weight", 0, 30, "Bit-Flip") # Inject fault
```

Fault injection into CNNs – Python

```
def inject_fault(model, tensor, element, bit, fault):
   dict = model.state_dict()
   shape = dict[tensor].shape
   flat = dict[tensor].flatten()
   int_value = flat[element].view(torch.int)
   mask = (0x00000001 << bit)
   if fault == "Stuck-at-1":
      faulty = torch.bitwise_or(int_value, mask)
   elif fault == "Stuck-at-0":
      faulty = torch.bitwise_and(int_value, ~mask)
   elif fault == "Bit-Flip":
      faulty = torch.bitwise_xor(int_value, mask)
   else:
      pass
   flat[element] = faulty.view(torch.float)
   dict[tensor] = flat.view(shape)
```

Get the float element as an integer

```
model = get_model("lenet5", weights="DEFAULT") # Obtain model
model.eval() # Set model in evaluation mode
inject_fault(model, "conv1.weight", 0, 30, "Bit-Flip") # Inject fault
```

```
OrderedDict({
   'conv1.weight': tensor([[[[ -0.3299266695976257, -0.2522581219673157,...],...]]]]),
   'conv1.bias': tensor([ 0.0525683201849461,  0.1234361827373505,...]),
   'conv2.weight': tensor([[[[ 0.0305016357451677, -0.1155040338635445,...]...]]]]),
   'conv2.bias': tensor([ 0.1148738339543343, -0.0630381628870964,...]),
   'fc1.weight': tensor([[ 0.0614890158176422, 0.0941545143723488,...]...]]),
   'fc1.bias': tensor([ -0.0740724205970764, 0.0834083408117294,...]),
   'fc2.weight': tensor([[ -0.0574139244854450, 0.0175603814423084,...]...]]),
   'fc2.bias': tensor([-0.0256344508379698,  0.0859057456254959, ...])
})

torch.Size([3, 1, 5, 5])

tensor([-0.3299266695976257, -0.2522581219673157,...])

tensor(-1096225754, dtype=torch.int32)10111110101010001110110000100110
```

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Institute ITACA Information and Communication Technologies

Fault injection into CNNs – Python

```
def inject_fault(model, tensor, element, bit, fault):
   dict = model.state_dict()
   shape = dict[tensor].shape
   flat = dict[tensor].flatten()
   int_value = flat[element].view(torch.int)
   mask = (0x00000001 << bit)
   if fault == "Stuck-at-1":
      faulty = torch.bitwise_or(int_value, mask)
   elif fault == "Stuck-at-0":
      faulty = torch.bitwise_and(int_value, ~mask)
   elif fault == "Bit-Flip":
      faulty = torch.bitwise_xor(int_value, mask)
   else:
      pass
   flat[element] = faulty.view(torch.float)
   dict[tensor] = flat.view(shape)
```

Apply the generated mask to flip a bit

```
model = get_model("lenet5", weights="DEFAULT") # Obtain model
model.eval() # Set model in evaluation mode
inject_fault(model, "conv1.weight", 0, 30, "Bit-Flip") # Inject fault
```

```
OrderedDict({
   'conv1.weight': tensor([[[[ -0.3299266695976257, -0.2522581219673157,...],...]]]]),
   'conv1.bias': tensor([ 0.0525683201849461,  0.1234361827373505,...]),
   'conv2.weight': tensor([[[[ 0.0305016357451677, -0.1155040338635445,...]...]]]]),
   'conv2.bias': tensor([ 0.1148738339543343, -0.0630381628870964,...]),
   'fc1.weight': tensor([[ 0.0614890158176422, 0.0941545143723488,...]...]]),
   'fc1.bias': tensor([ -0.0740724205970764, 0.0834083408117294,...]),
   'fc2.weight': tensor([[ -0.0574139244854450, 0.0175603814423084,...]...]]),
   'fc2.bias': tensor([-0.0256344508379698,  0.0859057456254959, ...])
})

torch.Size([3, 1, 5, 5])

tensor([-0.3299266695976257, -0.2522581219673157,...])

tensor(-1096225754, dtype=torch.int32)     10111110101010001110110000100110

tensor(-22483930, dtype=torch.int32)       11111110101010001110110000100110
```

Fault injection into CNNs – Python

```
def inject_fault(model, tensor, element, bit, fault):
   dict = model.state_dict()
   shape = dict[tensor].shape
   flat = dict[tensor].flatten()
   int_value = flat[element].view(torch.int)
   mask = (0x00000001 << bit)
   if fault == "Stuck-at-1":
      faulty = torch.bitwise_or(int_value, mask)
   elif fault == "Stuck-at-0":
      faulty = torch.bitwise_and(int_value, ~mask)
   elif fault == "Bit-Flip":
      faulty = torch.bitwise_xor(int_value, mask)
   else:
      pass
   flat[element] = faulty.view(torch.float)
   dict[tensor] = flat.view(shape)
```

Set the faulty integer as a float in the flattened tensor

```
model = get_model("lenet5", weights="DEFAULT") # Obtain model
model.eval() # Set model in evaluation mode
inject_fault(model, "conv1.weight", 0, 30, "Bit-Flip") # Inject fault
```

```
OrderedDict({
  'conv1.weight': tensor([[[[ -0.3299266695976257, -0.2522581219673157,...],...]]]]),
  'conv1.bias': tensor([ 0.0525683201849461,  0.1234361827373505,...]),
  'conv2.weight': tensor([[[[ 0.0305016357451677, -0.1155040338635445,...]...]]]]),
  'conv2.bias': tensor([ 0.1148738339543343, -0.0630381628870964,...]),
  'fc1.weight': tensor([[ 0.0614890158176422, 0.0941545143723488,...]...]]),
  'fc1.bias': tensor([ -0.0740724205970764, 0.0834083408117294,...]),
  'fc2.weight': tensor([[ -0.0574139244854450, 0.0175603814423084,...]...]]),
  'fc2.bias': tensor([-0.0256344508379698,  0.0859057456254959, ...])
})

torch.Size([3, 1, 5, 5])

tensor([-0.3299266695976257, -0.2522581219673157,...])

tensor(-1096225754, dtype=torch.int32)     10111110101010001110110000100110

tensor(-22483930, dtype=torch.int32)       11111110101010001110110000100110

tensor([-112268228041022512365824446628171350016.0, -0.2522581219673157,...])
```

Fault injection into CNNs – Python

```
def inject_fault(model, tensor, element, bit, fault):
  dict = model.state_dict()
  shape = dict[tensor].shape
  flat = dict[tensor].flatten()
  int_value = flat[element].view(torch.int)
  mask = (0x00000001 << bit)
  if fault == "Stuck-at-1":
     faulty = torch.bitwise_or(int_value, mask)
  elif fault == "Stuck-at-0":
     faulty = torch.bitwise_and(int_value, ~mask)
  elif fault == "Bit-Flip":
     faulty = torch.bitwise_xor(int_value, mask)
  else:
     pass
  flat[element] = faulty.view(torch.float)
  dict[tensor] = flat.view(shape)
```

Reshape the flattened tensor and set it in the dictionary

```
model = get_model("lenet5", weights="DEFAULT") # Obtain model
model.eval() # Set model in evaluation mode
inject_fault(model, "conv1.weight", 0, 30, "Bit-Flip") # Inject fault
```

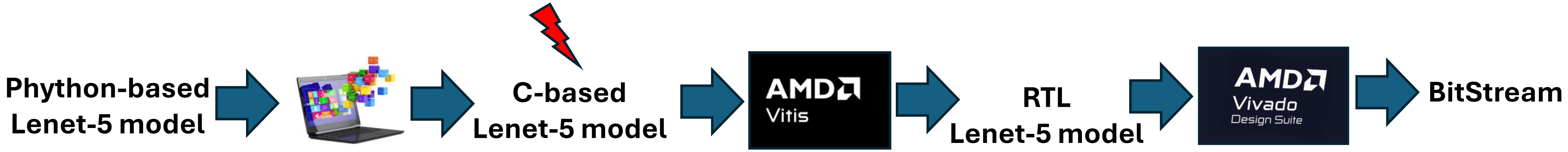
```
OrderedDict({
  'conv1.weight': tensor([[[[ -0.3299266695976257, -0.2522581219673157,...],...]]]]),
  'conv1.bias': tensor([ 0.0525683201849461,  0.1234361827373505,...]),
  'conv2.weight': tensor([[[[ 0.0305016357451677, -0.1155040338635445,...]...]]]]),
  'conv2.bias': tensor([ 0.1148738339543343, -0.0630381628870964,...]),
  'fc1.weight': tensor([[ 0.0614890158176422, 0.0941545143723488,...]...]]),
  'fc1.bias': tensor([ -0.0740724205970764, 0.0834083408117294,...]),
  'fc2.weight': tensor([[ -0.0574139244854450, 0.0175603814423084,...]...]]),
  'fc2.bias': tensor([-0.0256344508379698,  0.0859057456254959, ...])
})

torch.Size([3, 1, 5, 5])

tensor([-0.3299266695976257, -0.2522581219673157,...])

tensor(-1096225754, dtype=torch.int32)     10111110101010001110110000100110

tensor(-22483930, dtype=torch.int32)       11111110101010001110110000100110

tensor([-112268228041022512365824446628171350016.0, -0.2522581219673157,...])

OrderedDict({
  'conv1.weight': tensor([[[[-112268228041022512365824446628171350016.0,...],...]]]]),
  ...
})
```

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Institute ITACA Information and Communication Technologies

Levels of injection

- FP32
  - Python
  - **C**
- INT8
  - Python
  - C
- RTL
- FPGA



Fault injection into CNNs – C++

```
void inject_fault(float* tensor, int element, int bit, FAULT_TYPE faultType) {
   float* floatPTR = tensor + element;
   uint32_t* uiPTR = ((uint32_t*)floatPTR);
   uint32_t intValue = *uiPTR;
   uint32_t mask = (0x00000001) << bit;
   switch(faultType){
      case STUCK_AT_0:
         intValue = intValue & ~(mask);
         break;
      case STUCK_AT_1:
         intValue = intValue | (mask);
         break;
      case BITFLIP:
         intValue = intValue ^ (mask);
         break;
       case NO_FAULT:
         break;
    }
   *floatPTR = *((float*)& intValue);
}
```

```
inject_fault(getPointerToTensor(TensorID.KERNEL_C1), 0, 30, FaultType.BITFLIP)
```

Fault injection into CNNs – C++

```
void inject_fault(float* tensor, int element, int bit, FAULT_TYPE faultType) {
   float* floatPTR = tensor + element;
   uint32_t* uiPTR = ((uint32_t*)floatPTR);
   uint32_t intValue = *uiPTR;
   uint32_t mask = (0x00000001) << bit;
   switch(faultType){
      case STUCK_AT_0:
         intValue = intValue & ~(mask);
         break;
      case STUCK_AT_1:
         intValue = intValue | (mask);
         break;
      case BITFLIP:
         intValue = intValue ^ (mask);
         break;
       case NO_FAULT:
         break;
    }
   *floatPTR = *((float*)& intValue);
}
```

```
inject_fault(getPointerToTensor(TensorID.KERNEL_C1), 0, 30, FaultType.BITFLIP)
```

// Get the tensor

```
float KERNEL_C1[C1_KERNELS][C1_INPUT_FEATURES][C1_KERNEL_HEIGHT][C1_KERNEL_WIDTH] = {
  {{{-0.2016056776046753,  0.2383067756891251,...},...}}}
};

float BIAS_C1[C1_KERNELS] =
    {0.1679450124502182, 0.0189165733754635...};
float KERNEL_C2[C2_KERNELS][C1_KERNELS][C2_KERNEL_HEIGHT][C2_KERNEL_WIDTH] = {
    {{{  0.1170197501778603, 0.0326299667358398,...}...}}}
float BIAS_C2[C2_KERNELS] =
    { 0.0301316194236279, -0.0470375753939152,...};
float WEIGHTS_FC1[FC1_FEATURES][FC1_INPUT_FEATURES] = {
    {0.0561052113771439, -0.0237863268703222,...},...}
};
float BIAS_FC1[FC1_FEATURES] =
    { 0.0770544037222862, -0.0125858047977090, ...};
float WEIGHTS_FC2[LAST_LAYER_FEATURES][FC1_FEATURES] = {
    {0.1348823159933090, -0.0882048010826111,...},...}
};
float BIAS_FC2[LAST_LAYER_FEATURES] =
    {-0.0609014518558979, 0.0799584165215492, ...};
```

Fault injection into CNNs – C++

```
void inject_fault(float* tensor, int element, int bit, FAULT_TYPE faultType) {
   float* floatPTR = tensor + element;
   uint32_t* uiPTR = ((uint32_t*)floatPTR);
   uint32_t intValue = *uiPTR;
   uint32_t mask = (0x00000001) << bit;
   switch(faultType){
      case STUCK_AT_0:
         intValue = intValue & ~(mask);
         break;
      case STUCK_AT_1:
         intValue = intValue | (mask);
         break;
      case BITFLIP:
         intValue = intValue ^ (mask);
         break;
       case NO_FAULT:
         break;
    }
   *floatPTR = *((float*)& intValue);
}
```

Get a pointer to the float element from the flattened tensor

```
float KERNEL_C1[C1_KERNELS][C1_INPUT_FEATURES][C1_KERNEL_HEIGHT][C1_KERNEL_WIDTH] = {
  {{{-0.2016056776046753, 0.2383067756891251,,...},...}}}
};
```

```
-0.2016056776046753
```

```
inject_fault(getPointerToTensor(TensorID.KERNEL_C1), 0, 30, FaultType.BITFLIP)
```

Fault injection into CNNs – C++

```
void inject_fault(float* tensor, int element, int bit, FAULT_TYPE faultType) {
  float* floatPTR = tensor + element;
  uint32_t* uiPTR = ((uint32_t*)floatPTR);
  uint32_t intValue = *uiPTR;
  uint32_t mask = (0x00000001) << bit;
  switch(faultType){
    case STUCK_AT_0:
      intValue = intValue & ~(mask);
      break;
    case STUCK_AT_1:
      intValue = intValue | (mask);
      break;
    case BITFLIP:
      intValue = intValue ^ (mask);
      break;
     case NO_FAULT:
      break;
   }
  *floatPTR = *((float*)& intValue);
}
```

Get the float element as an integer (binary representation)

```
inject_fault(getPointerToTensor(TensorID.KERNEL_C1), 0, 30, FaultType.BITFLIP)
```

```
float KERNEL_C1[C1_KERNELS][C1_INPUT_FEATURES][C1_KERNEL_HEIGHT][C1_KERNEL_WIDTH] = {
  {{{-0.2016056776046753,  0.2383067756891251,,...},...}}}
};
```

-0.2016056776046753

-1102155336   10111110010011100111000110111000

Fault injection into CNNs – C++

```
void inject_fault(float* tensor, int element, int bit, FAULT_TYPE faultType) {
  float* floatPTR = tensor + element;
  uint32_t* uiPTR = ((uint32_t*)floatPTR);
  uint32_t intValue = *uiPTR;
  uint32_t mask = (0x00000001) << bit;
  switch(faultType){
    case STUCK_AT_0:
      intValue = intValue & ~(mask);
      break;
    case STUCK_AT_1:
      intValue = intValue | (mask);
      break;
    case BITFLIP:
      intValue = intValue ^ (mask);
      break;
     case NO_FAULT:
      break;
   }
  *floatPTR = *((float*)& intValue);
}
```

Apply the generated mask to flip a bit

```
inject_fault(getPointerToTensor(TensorID.KERNEL_C1), 0, 30, FaultType.BITFLIP)
```

```
float KERNEL_C1[C1_KERNELS][C1_INPUT_FEATURES][C1_KERNEL_HEIGHT][C1_KERNEL_WIDTH] = {
  {{{-0.2016056776046753,  0.2383067756891251,,...},...}}}
};
```

-0.2016056776046753

$\boxed{-1102155336}$ 10111110010011100111000110111000

$\boxed{-28413512}$ 11111110010011100111000110111000

Fault injection into CNNs – C++

```
void inject_fault(float* tensor, int element, int bit, FAULT_TYPE faultType) {
   float* floatPTR = tensor + element;
   uint32_t* uiPTR = ((uint32_t*)floatPTR);
   uint32_t intValue = *uiPTR;
   uint32_t mask = (0x00000001) << bit;
   switch(faultType){
      case STUCK_AT_0:
         intValue = intValue & ~(mask);
         break;
      case STUCK_AT_1:
         intValue = intValue | (mask);
         break;
      case BITFLIP:
         intValue = intValue ^ (mask);
         break;
       case NO_FAULT:
         break;
    }
   *floatPTR = *((float*)& intValue);
}
```

Set the faulty integer (binary representation) as a float in the tensor

```
inject_fault(getPointerToTensor(TensorID.KERNEL_C1), 0, 30, FaultType.BIT_FLIP)
```

```
float KERNEL_C1[C1_KERNELS][C1_INPUT_FEATURES][C1_KERNEL_HEIGHT][C1_KERNEL_WIDTH] = {
   {{{-0.2016056776046753,  0.2383067756891251,,...},...}}}
};
```

-0.2016056776046753

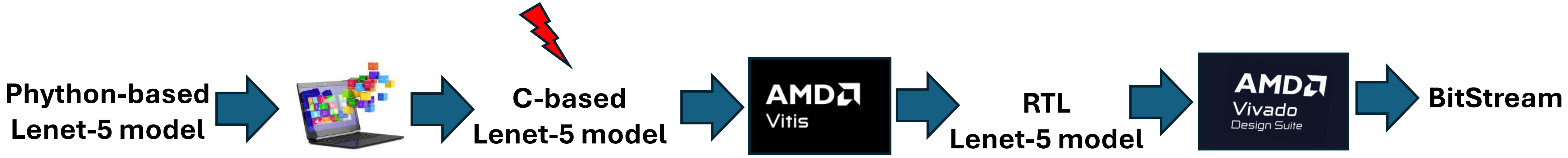
-1102155336   10111110010011100111000110111000

-28413512   11111110010011100111000110111000

-6860285716001854431189155169929737 0112.0

Levels of injection

- FP32
  - Python
  - C
- INT8
  - Python
  - **C**
- RTL
- FPGA



Fault injection into quantized C++-based CNNs

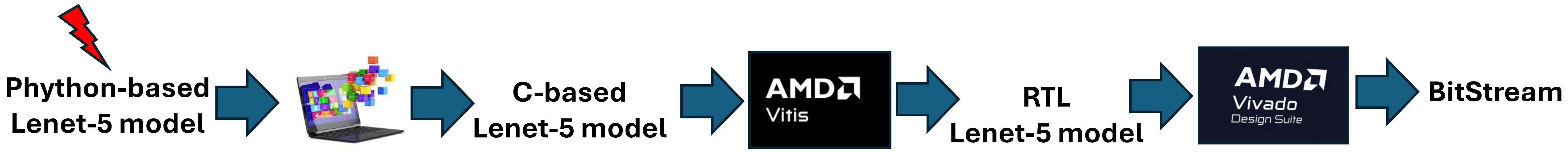
```
template<typename T>
void inject_fault(T* tensor, int element, int bit, FAULT_TYPE faultType) {
  T* dataPTR = tensor + element;
  if (std::is_same<T, int8_t>>::value || std::is_same<T, uint8_t>>::value)
    uint8_t* uiPTR = ((T*)dataPTR);
  else
    uint32_t* uiPTR = ((T*)dataPTR);
  uint32_t intValue = *uiPTR;
  uint32_t mask = (0x00000001) << bit;
  switch(faultType){
    case STUCK_AT_0:
      intValue = intValue & ~(mask);
      break;
    case STUCK_AT_1:
      intValue = intValue | (mask);
      break;
    case BITFLIP:
      intValue = intValue ^ (mask);
      break;
     case NO_FAULT:
      break;
  }
  *dataPTR = *((T*)& intValue);
}
```

Elements can be INT8, UINT8, INT32 or UINT32
No major changes!

```
inject_fault<int8_t>(getPointerToTensor(TensorID.KERNEL_C1), 0, 6, FaultType.BIT_FLIP)
```

Levels of injection

- FP32
  - Python
  - C
- INT8
  - **Python**
  - C
- RTL
- FPGA



UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Institute ITACA Information and Communication Technologies

Fault injection into quantized torch-based python CNNs

- Torch models work internally with an heterogeneus dictionary
  - Quantized values internally stored using
    - Value (float)
    - Scalar factor (affects each value and the general factor M_0)
    - Zero_point (only weights since it is 0 for biases)
- How to reproduce the effect of a bitflip in a quantized parameter by acting on its value, zero_point and scalar factor?

```
OrderedDict({
‘conv1.weight’: tensor(
  [[[[-0.0391622632741928,  0.1982589513063431,  ...],...]]]],
  size=(3, 1, 5, 5),
  dtype=torch.qint8,
  quantization_scheme=torch.per_channel_affine,
  scale=tensor(
    [0.0024476414546371, 0.0024339694064111, ...],
    dtype=torch.float64
  ),
  zero_point=tensor([0, 0,...]),
  axis=0
),
'conv1.bias': Parameter containing:
tensor(
  [-0.1437491327524185,  0.2681810557842255,  ...],
  requires_grad=True
),
'conv1.scale': tensor(0.0500611327588558),
'conv1.zero_point': tensor(0),
'fc1.scale': tensor(0.0981808006763458),
'fc1.zero_point': tensor(63),
'fc1._packed_params.dtype': torch.qint8,
'fc1._packed_params._packed_params’: (
  tensor([[ 0.1034839898347855, -0.0543729439377785, ...]]
  ....
```

Fault injection into quantized torch-based python CNNs (weights)

```
def injectFaultInConvolutionWeights(model, moduleName, outChannel,
                                    inChannel, height, width, bit):
  weights, biases = modelq._modules[moduleName]._weight_bias()
  scales = weights.q_per_channel_scales()
  zero_points = weights.q_per_channel_zero_points()
  intValue =
    round(w[outChannel][inChannel][height][width].item() / s[outChannel].item()) +
    z[outChannel].item()
  mask = 0x01 << bit;
  injectionValue = (intValue ^ mask) & 0xFF
  faultyValue = (injectionValue – zero_points[outChannel].item()) * scales[outChannel] .item()
  weights[outChannel][inChannel][height][width] = faultyValue
  modelq._modules[moduleName].set_weight_bias(weights, biases)
```

```
OrderedDict({
'conv1.weight': tensor(
   [[[[-0.0391622632741928,  0.1982589513063431,  ...],...]]]],
   size=(3, 1, 5, 5),
   dtype=torch.qint8,
   quantization_scheme=torch.per_channel_affine,
   scale=tensor(
      [0.0024476414546371, 0.0024339694064111, ...],
      dtype=torch.float64
   ),
   zero_point=tensor([0, 0,...]),
   axis=0
),
```

```
-0.03916226327419281
```

Get weights, scales and zero_points

```
modelq = get_model("quantized_lenet5", weights="DEFAULT", quantize=True)
modelq.eval()
injectFaultInConvolutionWeights(modelq, "conv1", 0, 0, 0, 0, 7)
```

Fault injection into quantized torch-based python CNNs (weights)

```
def injectFaultInConvolutionWeights(model, moduleName, outChannel,
                                    inChannel, height, width, bit):
  weights, biases = modelq._modules[moduleName]._weight_bias()
  scales = weights.q_per_channel_scales()
  zero_points = weights.q_per_channel_zero_points()
  intValue =
    round(w[outChannel][inChannel][height][width].item() / s[outChannel].item()) +
    z[outChannel].item()
  mask = 0x01 << bit;
  injectionValue = (intValue ^ mask) & 0xFF
  faultyValue = (injectionValue – zero_points[outChannel].item()) * scales[outChannel] .item()
  weights[outChannel][inChannel][height][width] = faultyValue
  modelq._modules[moduleName].set_weight_bias(weights, biases)
```

```
OrderedDict({
'conv1.weight': tensor(
   [[[[-0.0391622632741928,  0.1982589513063431,  ...],...]]]],
   size=(3, 1, 5, 5),
   dtype=torch.qint8,
   quantization_scheme=torch.per_channel_affine,
   scale=tensor(
     [0.0024476414546371, 0.0024339694064111, ...],
     dtype=torch.float64
   ),
   zero_point=tensor([0, 0,...]),
   axis=0
),
```

-0.03916226327419281

-16 **11110000**

Get the float weight as the corresponding integer value
round(-0.03916226327419281 / 0.0024476414546371) + 0 = -16

```
modelq = get_model("quantized_lenet5", weights="DEFAULT", quantize=True)
modelq.eval()
injectFaultInConvolutionWeights(modelq, "conv1", 0, 0, 0, 0, 7)
```

Fault injection into quantized torch-based python CNNs (weights)

```
def injectFaultInConvolutionWeights(model, moduleName, outChannel,
                                    inChannel, height, width, bit):
  weights, biases = modelq._modules[moduleName]._weight_bias()
  scales = weights.q_per_channel_scales()
  zero_points = weights.q_per_channel_zero_points()
  intValue =
    round(w[outChannel][inChannel][height][width].item() / s[outChannel].item()) +
    z[outChannel].item()
  mask = 0x01 << bit;
  injectionValue = (intValue ^ mask) & 0xFF
  faultyValue = (injectionValue – zero_points[outChannel].item()) * scales[outChannel] .item()
  weights[outChannel][inChannel][height][width] = faultyValue
  modelq._modules[moduleName].set_weight_bias(weights, biases)
```

<<

Apply the generated mask to flip a bit

```
OrderedDict({
 'conv1.weight': tensor(
   [[[[-0.0391622632741928,  0.1982589513063431,  ...],...]]]],
   size=(3, 1, 5, 5),
   dtype=torch.qint8,
   quantization_scheme=torch.per_channel_affine,
   scale=tensor(
     [0.0024476414546371, 0.0024339694064111, ...],
     dtype=torch.float64
   ),
   zero_point=tensor([0, 0,...]),
   axis=0
 ),
```

-0.03916226327419281

-16	11110000
112	01110000

```
modelq = get_model("quantized_lenet5", weights="DEFAULT", quantize=True)
modelq.eval()
injectFaultInConvolutionWeights(modelq, "conv1", 0, 0, 0, 0, 7)
```

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Fault injection into quantized torch-based python CNNs (weights)

Institute ITACA
Information and Communication Technologies

```
def injectFaultInConvolutionWeights(model, moduleName, outChannel,
                                    inChannel, height, width, bit):
  weights, biases = modelq._modules[moduleName]._weight_bias()
  scales = weights.q_per_channel_scales()
  zero_points = weights.q_per_channel_zero_points()
  intValue =
     round(w[outChannel][inChannel][height][width].item() / s[outChannel].item()) +
     z[outChannel].item()
  mask = 0x01 << bit;
  injectionValue = (intValue ^ mask) & 0xFF
  faultyValue = (injectionValue – zero_points[outChannel].item()) * scales[outChannel] .item()
  weights[outChannel][inChannel][height][width] = faultyValue
  modelq._modules[moduleName].set_weight_bias(weights, biases)
```

Get the faulty integer value as a float value
(112 – 0) x .0024476414546371 = 0.2741358280181885

```
modelq = get_model("quantized_lenet5", weights="DEFAULT", quantize=True)
modelq.eval()
injectFaultInConvolutionWeights(modelq, "conv1", 0, 0, 0, 0, 7)
```

```
OrderedDict({
'conv1.weight': tensor(
   [[[[-0.0391622632741928,  0.1982589513063431,  ...],...]]]],
   size=(3, 1, 5, 5),
   dtype=torch.qint8,
   quantization_scheme=torch.per_channel_affine,
   scale=tensor(
      [0.0024476414546371, 0.0024339694064111, ...],
      dtype=torch.float64
   ),
   zero_point=tensor([0, 0,...]),
   axis=0
),
```

-0.03916226327419281

-16	**11110000**
112	**01110000**

0.2741358280181885

Fault injection into quantized torch-based python CNNs (weights)

```
def injectFaultInConvolutionWeights(model, moduleName, outChannel,
                                    inChannel, height, width, bit):
  weights, biases = modelq._modules[moduleName]._weight_bias()
  scales = weights.q_per_channel_scales()
  zero_points = weights.q_per_channel_zero_points()
  intValue =
    round(w[outChannel][inChannel][height][width].item() / s[outChannel].item()) +
    z[outChannel].item()
  mask = 0x01 << bit;
  injectionValue = (intValue ^ mask) & 0xFF
  faultyValue = (injectionValue – zero_points[outChannel].item()) * scales[outChannel] .item()
  weights[outChannel][inChannel][height][width] = faultyValue
  modelq._modules[moduleName].set_weight_bias(weights, biases)
```

```
OrderedDict({
'conv1.weight': tensor(
   [[[[-0.0391622632741928,  0.1982589513063431,  ...],...]]]],
   size=(3, 1, 5, 5),
   dtype=torch.qint8,
   quantization_scheme=torch.per_channel_affine,
   scale=tensor(
     [0.0024476414546371, 0.0024339694064111, ...],
     dtype=torch.float64
   ),
   zero_point=tensor([0, 0,...]),
   axis=0
),
```

```
-0.03916226327419281
```

```
-16    11110000
112    01110000
0.2741358280181885
```

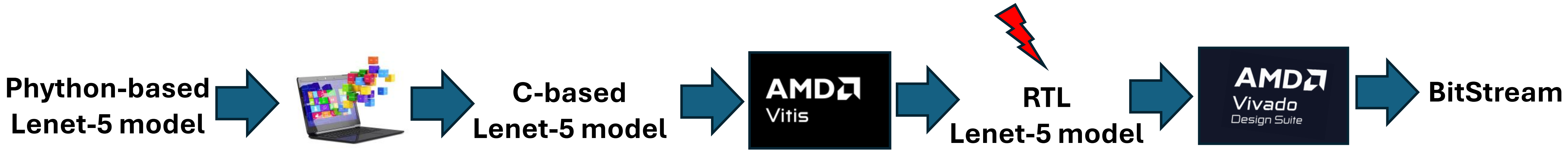
```
conv1.weight: tensor(
  [[[[0.2741358280181885,  0.1982589513063431,  ...],...]]]],
```

Set the faulty weights (and biases) by module name

```
modelq = get_model("quantized_lenet5", weights="DEFAULT", quantize=True)
modelq.eval()
injectFaultInConvolutionWeights(modelq, "conv1", 0, 0, 0, 0, 7)
```

Levels of injection

- FP32
  - Python
  - C
- INT8
  - Python
  - C
- **RTL**
- FPGA



Simulation-based fault injection – transient faults in VHDL signals

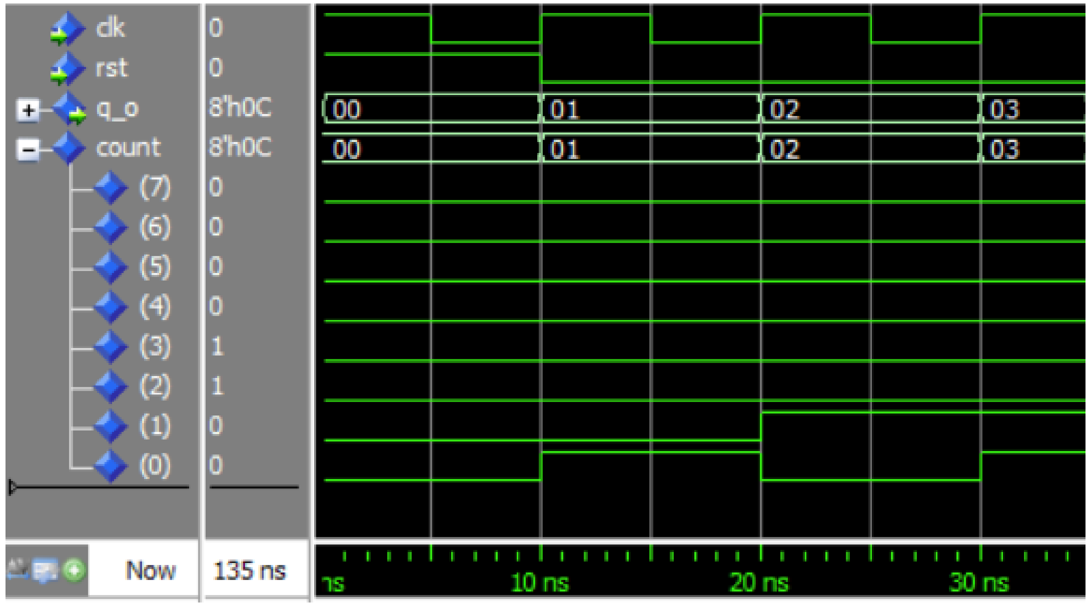
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Counter8b is
   Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           q : out STD_LOGIC_VECTOR(7 downto 0));
end Counter8b;
architecture Behavioral of Counter8b is
   constant GND : STD_LOGIC_VECTOR(7 downto 0) := x"00";
   signal count : UNSIGNED(7 downto 0) := GND;
begin
   process(clk)
   begin
      if rising_edge(clk) then
         if rst = '1' then
            count <= GND;
         else
            count <= count + 1;
         end if;
      end if;
   end process;
   q <= std_logic_vector(count);
end Behavioral;
```

Simulation-based fault injection of transient faults in VHDL signals

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Counter8b is
  Port ( clk : in STD_LOGIC;
         rst : in STD_LOGIC;
         q : out STD_LOGIC_VECTOR(7 downto 0));
end Counter8b;
architecture Behavioral of Counter8b is
  constant GND : STD_LOGIC_VECTOR(7 downto 0) := x"00";
  signal count : UNSIGNED(7 downto 0) := GND;
begin
  process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        count <= GND;
      else
        count <= count + 1;
      end if;
    end if;
  end process;
  q <= std_logic_vector(count);
end Behavioral;
```

```
run 35 ns
```

Run until the injection time is reached

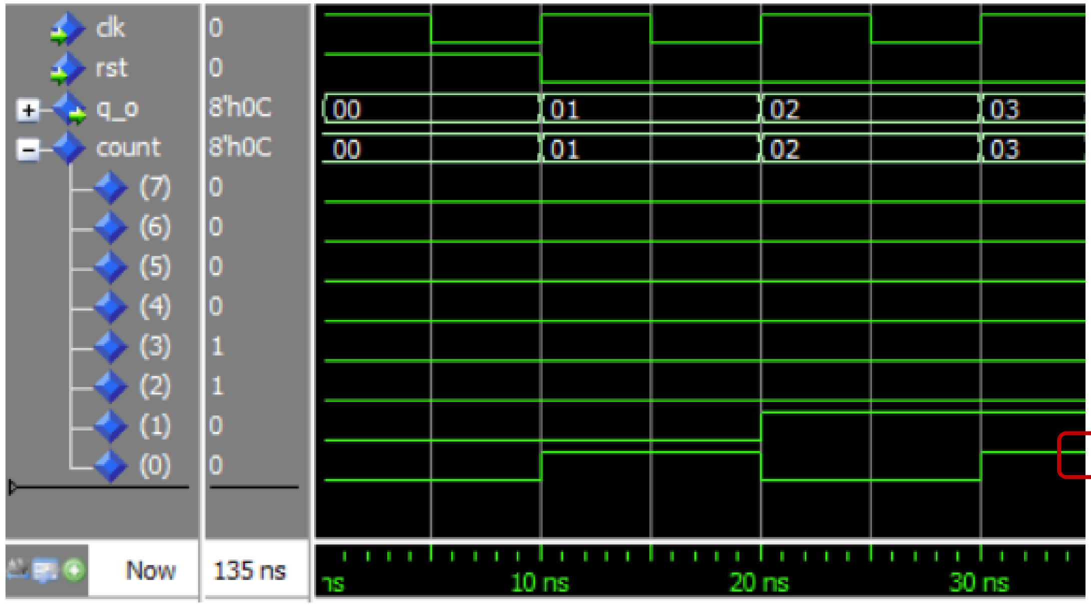


Simulation-based fault injection – transient faults in VHDL signals

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Counter8b is
  Port ( clk : in STD_LOGIC;
         rst : in STD_LOGIC;
         q : out STD_LOGIC_VECTOR(7 downto 0));
end Counter8b;
architecture Behavioral of Counter8b is
  constant GND : STD_LOGIC_VECTOR(7 downto 0) := x"00";
  signal count : UNSIGNED(7 downto 0) := GND;
begin
  process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        count <= GND;
      else
        count <= count + 1;
      end if;
    end if;
  end process;
  q <= std_logic_vector(count);
end Behavioral;
```

```
run 35 ns
set value [examine /testbench/uut/count(0)]          # value = 1
```

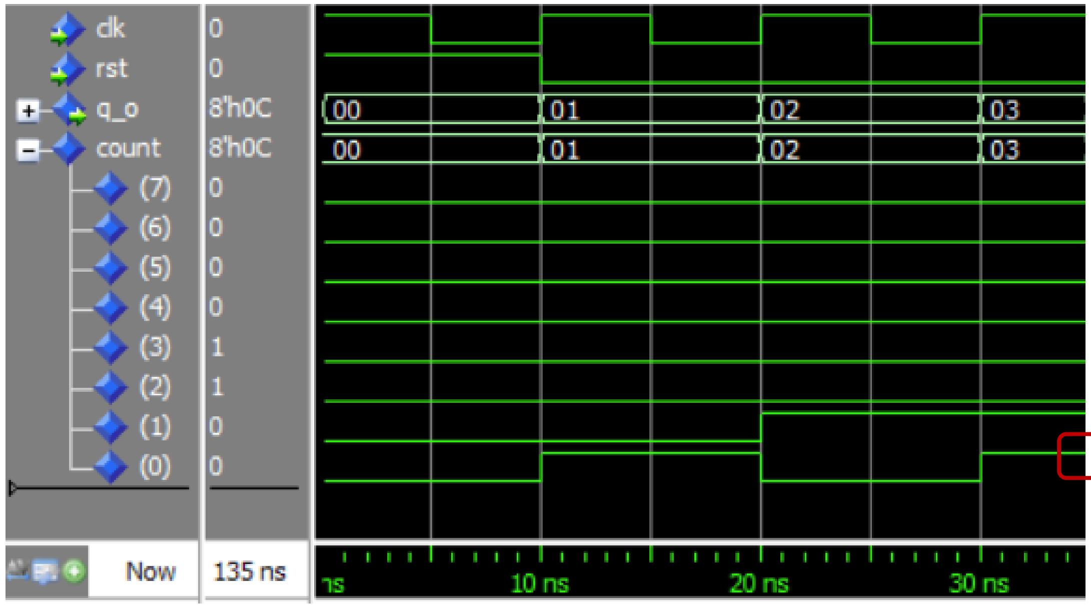
Get the current value of the target signal


```
clk     0
rst     0
q_o     8'h0C    00 | 01 | 02 | 03
count   8'h0C    00 | 01 | 02 | 03
 (7)    0
 (6)    0
 (5)    0
 (4)    0
 (3)    1
 (2)    1
 (1)    0
 (0)    0
Now     135 ns      ns   10 ns   20 ns   30 ns
```

Simulation-based fault injection – transient faults in VHDL signals

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Counter8b is
  Port ( clk : in STD_LOGIC;
         rst : in STD_LOGIC;
         q : out STD_LOGIC_VECTOR(7 downto 0));
end Counter8b;
architecture Behavioral of Counter8b is
  constant GND : STD_LOGIC_VECTOR(7 downto 0) := x"00";
  signal count : UNSIGNED(7 downto 0) := GND;
begin
  process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        count <= GND;
      else
        count <= count + 1;
      end if;
    end if;
  end process;
  q <= std_logic_vector(count);
end Behavioral;
```

```
run 35 ns
set value [examine /testbench/uut/count(0)]                # value = 1
set faultyValue [expr !$value]                             # faultyValue = 0
force -deposit /testbench/uut/count(0) $faultyValue 0      # inject fault
```

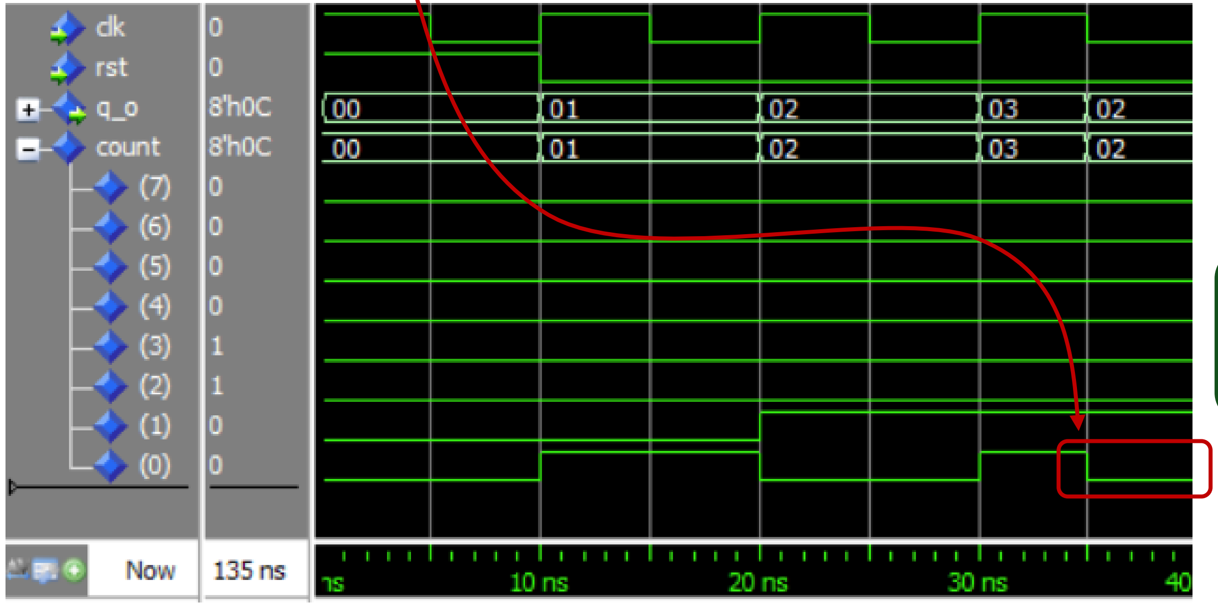


Set the signal to the faulty value until it is overridden by the system dynamics

Simulation-based fault injection – transient faults in VHDL signals

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Counter8b is
  Port ( clk : in STD_LOGIC;
         rst : in STD_LOGIC;
         q : out STD_LOGIC_VECTOR(7 downto 0));
end Counter8b;
architecture Behavioral of Counter8b is
  constant GND : STD_LOGIC_VECTOR(7 downto 0) := x"00";
  signal count : UNSIGNED(7 downto 0) := GND;
begin
  process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        count <= GND;
      else
        count <= count + 1;
      end if;
    end if;
  end process;
  q <= std_logic_vector(count);
end Behavioral;
```

```
run 35 ns
set value [examine /testbench/uut/count(0)]                  # value = 1
set faultyValue [expr !$value]                              # faultyValue = 0
force -deposit /testbench/uut/count(0) $faultyValue 0       # inject fault
run 5 ns
```



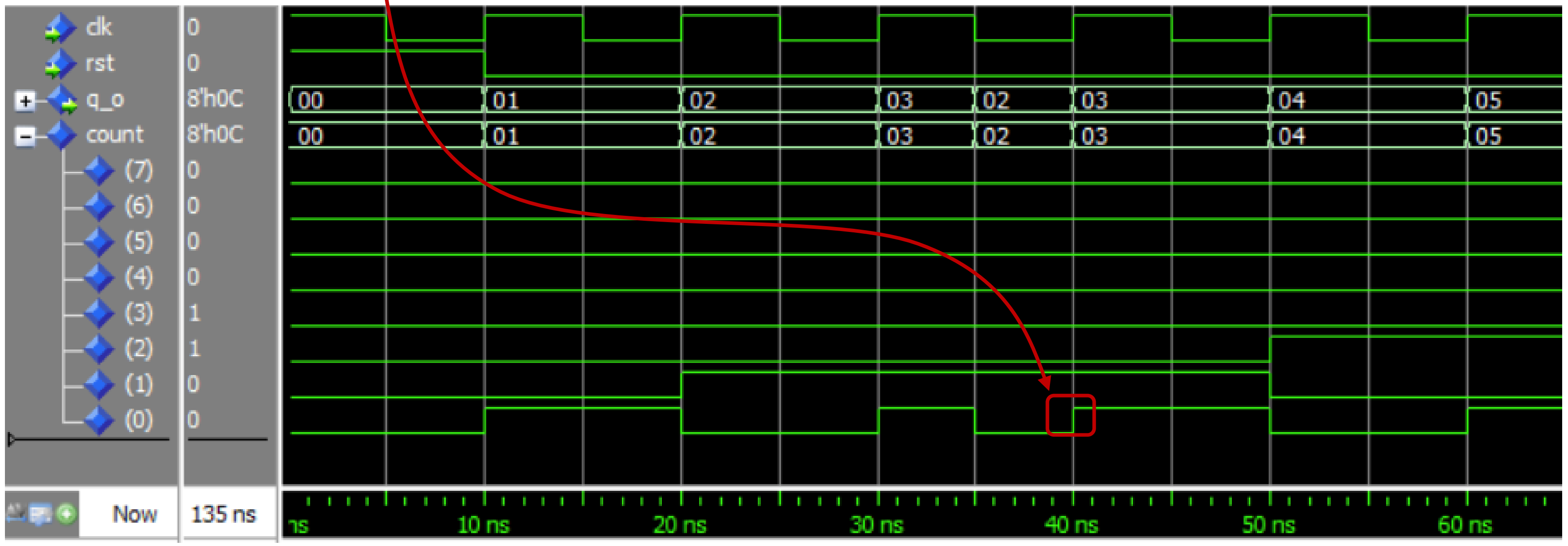
Run some more time. The target signal has a faulty value

Simulation-based fault injection – transient faults in VHDL signals

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Counter8b is
  Port ( clk : in STD_LOGIC;
         rst : in STD_LOGIC;
         q : out STD_LOGIC_VECTOR(7 downto 0));
end Counter8b;
architecture Behavioral of Counter8b is
  constant GND : STD_LOGIC_VECTOR(7 downto 0) := x"00";
  signal count : UNSIGNED(7 downto 0) := GND;
begin
  process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        count <= GND;
      else
        count <= count + 1;
      end if;
    end if;
  end process;
  q <= std_logic_vector(count);
end Behavioral;
```

```
run 35 ns
set value [examine /testbench/uut/count(0)]                # value = 1
set faultyValue [expr !$value]                             # faultyValue = 0
force -deposit /testbench/uut/count(0) $faultyValue 0      # inject fault
run 5 ns
run 100 ns
```

Run some more time.
The system dynamic has overridden the faulty value



Simulation-based fault injection – permanent faults in VHDL constants

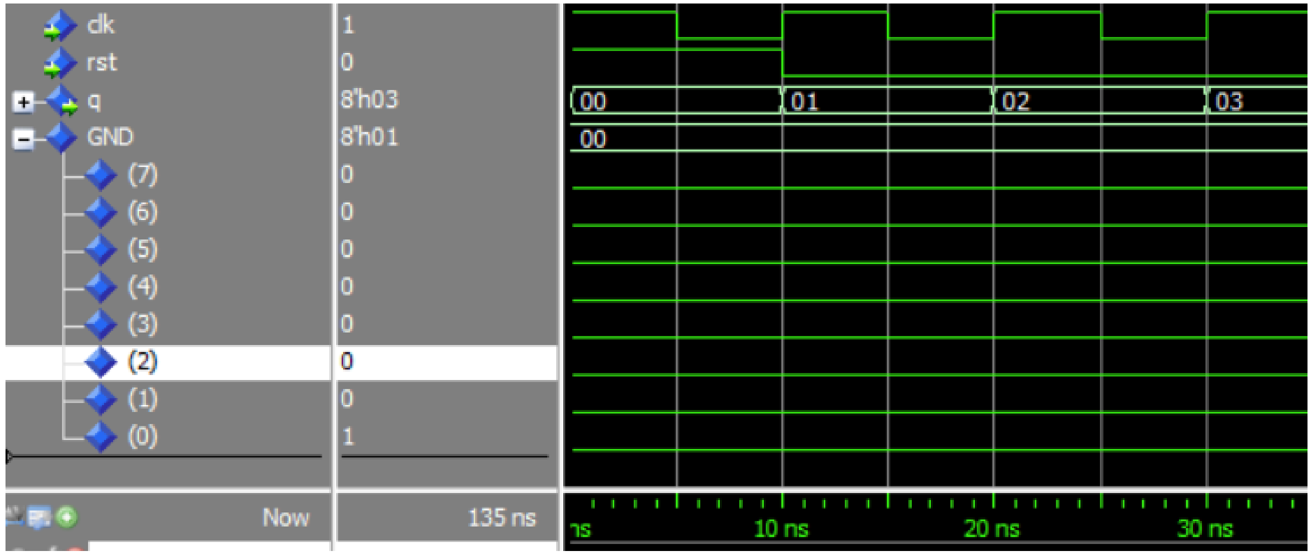
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Counter8b is
   Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           q : out STD_LOGIC_VECTOR(7 downto 0));
end Counter8b;
architecture Behavioral of Counter8b is
   constant GND : STD_LOGIC_VECTOR(7 downto 0) := x”00”;
   signal count : UNSIGNED(7 downto 0) := GND;
begin
   process(clk)
   begin
      if rising_edge(clk) then
         if rst = '1' then
            count <= GND;
         else
            count <= count + 1;
         end if;
      end if;
   end process;
   q <= std_logic_vector(count);
end Behavioral;
```

Simulation-based fault injection – permanent faults in VHDL constants

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Counter8b is
  Port ( clk : in STD_LOGIC;
         rst : in STD_LOGIC;
         q : out STD_LOGIC_VECTOR(7 downto 0));
end Counter8b;
architecture Behavioral of Counter8b is
  constant GND : STD_LOGIC_VECTOR(7 downto 0) := x"00";
  signal count : UNSIGNED(7 downto 0) := GND;
begin
  process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        count <= GND;
      else
        count <= count + 1;
      end if;
    end if;
  end process;
  q <= std_logic_vector(count);
end Behavioral;
```

```
run 35 ns
```

Run until the injection time is reached

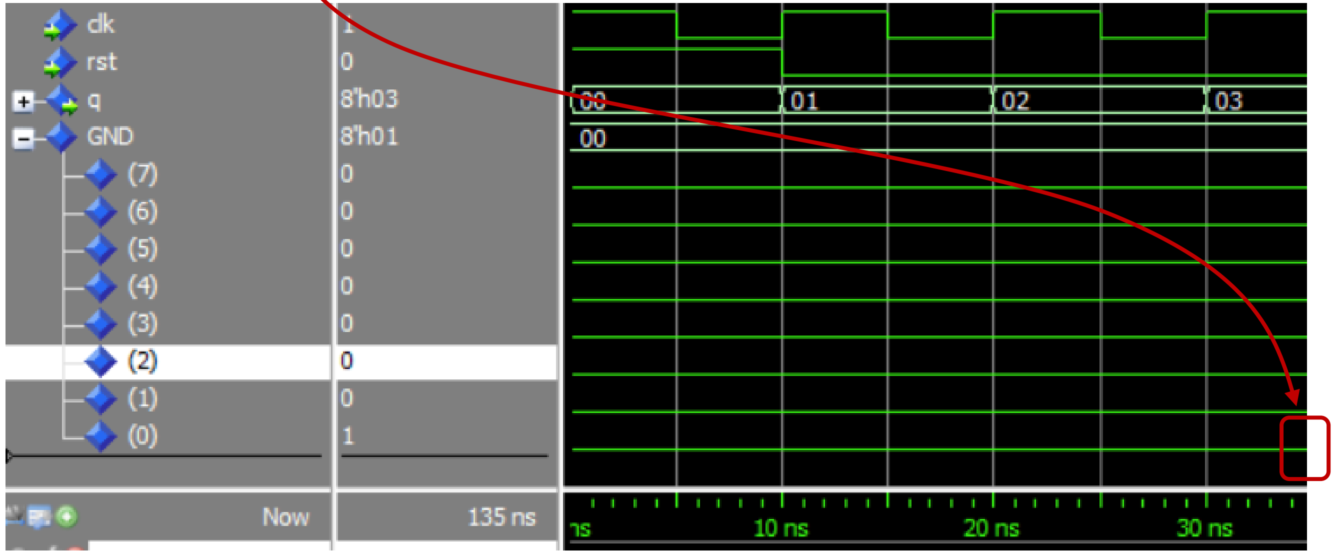


Simulation-based fault injection – permanent faults in VHDL constants

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Counter8b is
  Port ( clk : in STD_LOGIC;
         rst : in STD_LOGIC;
         q : out STD_LOGIC_VECTOR(7 downto 0));
end Counter8b;
architecture Behavioral of Counter8b is
  constant GND : STD_LOGIC_VECTOR(7 downto 0) := x"00";
  signal count : UNSIGNED(7 downto 0) := GND;
begin
  process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        count <= GND;
      else
        count <= count + 1;
      end if;
    end if;
  end process;
  q <= std_logic_vector(count);
end Behavioral;
```

```
run 35 ns
set value [examine /testbench/uut/GND(0)]      # value = 0
```

Get the current value of the target constant



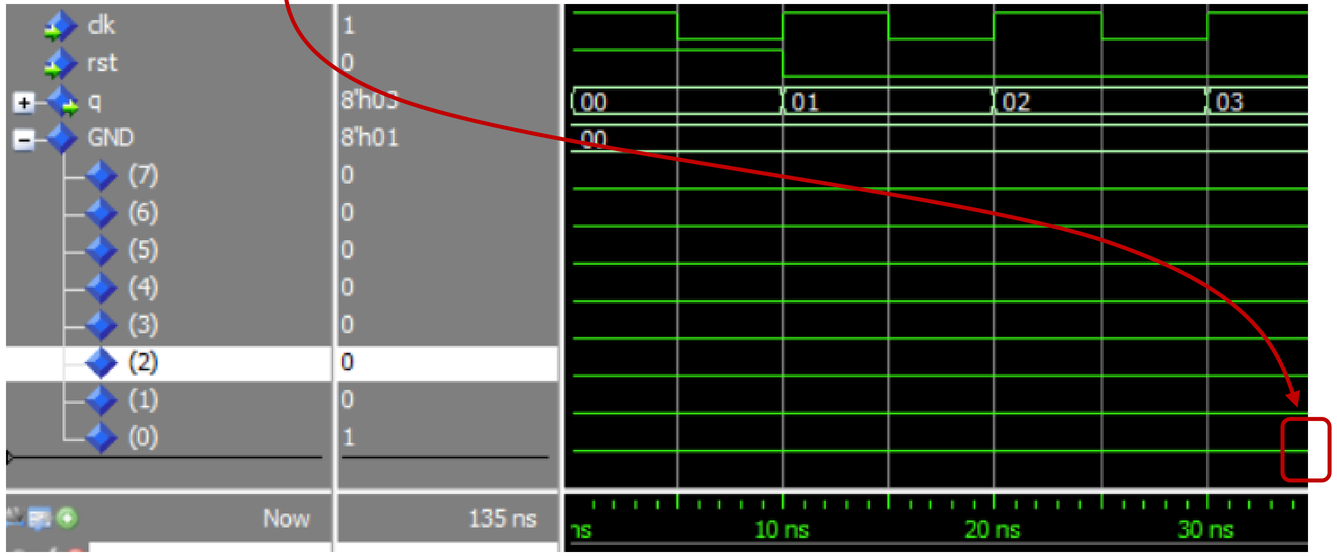
```
clk    1
rst    0
q      8'h03
GND    8'h01
 (7)   0
 (6)   0
 (5)   0
 (4)   0
 (3)   0
 (2)   0
 (1)   0
 (0)   1
Now    135 ns
```

Simulation-based fault injection – permanent faults in VHDL constants

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Counter8b is
  Port ( clk : in STD_LOGIC;
         rst : in STD_LOGIC;
         q : out STD_LOGIC_VECTOR(7 downto 0));
end Counter8b;
architecture Behavioral of Counter8b is
  constant GND : STD_LOGIC_VECTOR(7 downto 0) := x"00";
  signal count : UNSIGNED(7 downto 0) := GND;
begin
  process(clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        count <= GND;
      else
        count <= count + 1;
      end if;
    end if;
  end process;
  q <= std_logic_vector(count);
end Behavioral;
```

```
run 35 ns
set value [examine /testbench/uut/GND(0)]        # value = 0
set faultyValue [expr !$value]                   # faultyValue = 1
change /testbench/uut/GND(0) $faultyValue        # inject fault
```

Set the constant to the faulty value

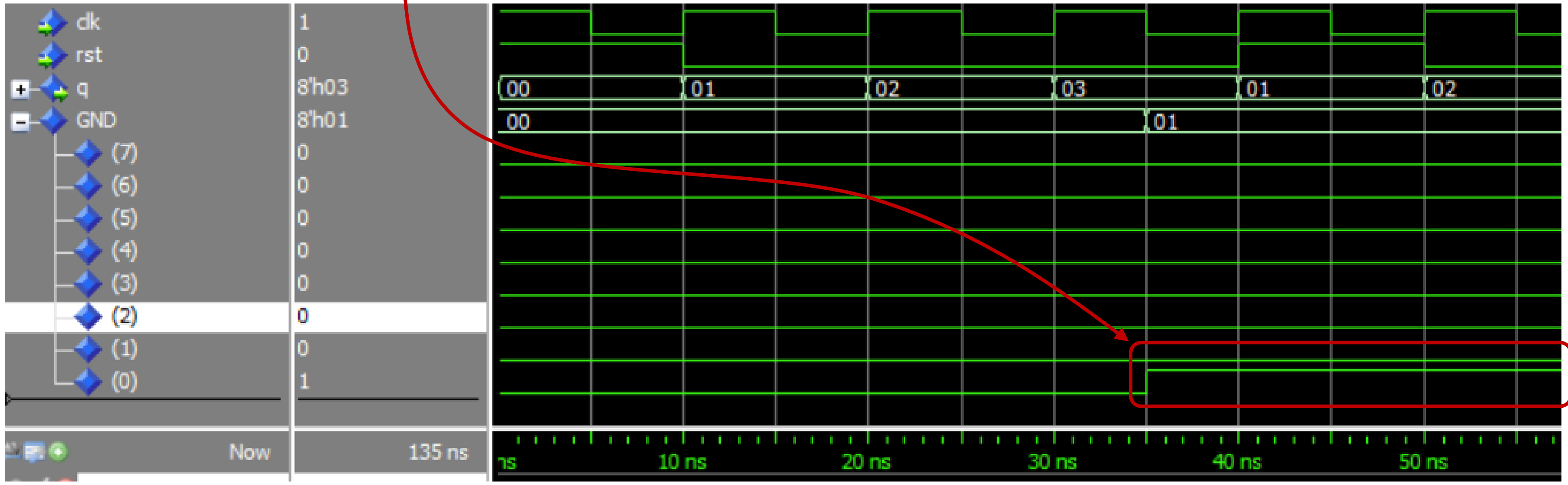


Simulation-based fault injection – permanent faults in VHDL constants

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Counter8b is
   Port ( clk : in STD_LOGIC;
          rst : in STD_LOGIC;
          q : out STD_LOGIC_VECTOR(7 downto 0));
end Counter8b;
architecture Behavioral of Counter8b is
   constant GND : STD_LOGIC_VECTOR(7 downto 0) := x"00";
   signal count : UNSIGNED(7 downto 0) := GND;
begin
   process(clk)
   begin
     if rising_edge(clk) then
       if rst = '1' then
         count <= GND;
       else
         count <= count + 1;
       end if;
     end if;
   end process;
   q <= std_logic_vector(count);
end Behavioral;
```

```tcl
run 35 ns
set value [examine /testbench/uut/GND(0)]        # value = 0
set faultyValue [expr !$value]                  # faultyValue = 1
change /testbench/uut/GND(0) $faultyValue       # inject fault
run 100 ns
```

Run some more time.
The fault is permanent (constants are not supposed to change, after all)



```
clk   1
rst   0
q     8'h03    00 | 01 | 02 | 03 | 01 | 02
GND   8'h01    00 | 01
 (7)  0
 (6)  0
 (5)  0
 (4)  0
 (3)  0
 (2)  0
 (1)  0
 (0)  1
Now   135 ns
```

RTL-based injection in the FP32 Lenet-5 (permanent faults on constants)

```
entity cnn_convolution2DRelu_1 is
port (
   ap_clk : IN STD_LOGIC;
...
end entity;

architecture behav of cnn_convolution2DRelu_1 is
   ...
   constant ap_const_lv32_BE40F773 : STD_LOGIC_VECTOR (31 downto 0) := "10111110010000001111011101110011";
   constant ap_const_lv32_3E26A585 : STD_LOGIC_VECTOR (31 downto 0) := "00111110001001101010010110000101";
   constant ap_const_lv32_3E6CFE34 : STD_LOGIC_VECTOR (31 downto 0) := "00111110011011001111111000110100";
   constant ap_const_lv32_3E58F215 : STD_LOGIC_VECTOR (31 downto 0) := "00111110010110001111001000010101";
   constant ap_const_lv32_3D150561 : STD_LOGIC_VECTOR (31 downto 0) := "00111101000101010000010101100001";
   constant ap_const_lv32_3EB152B2 : STD_LOGIC_VECTOR (31 downto 0) := "00111110101100010101001010110010";
   ....
```

First Convolution2D + Relu
Highly complex description
8535 VHDL lines vs 40 C++
lines (comments excluded)

RTL-based injection in the FP32 Lenet-5 (permanent faults on constants)

```
entity cnn_convolution2DRelu_1 is
port (
  ap_clk : IN STD_LOGIC;
...
end entity;

architecture behav of cnn_convolution2DRelu_1 is
  ...
  constant ap_const_lv32_BE40F773 : STD_LOGIC_VECTOR (31 downto 0) := "10111110010000001111011101110011";
  constant ap_const_lv32_3E26A585 : STD_LOGIC_VECTOR (31 downto 0) := "00111110001001101010010110000101";
  constant ap_const_lv32_3E6CFE34 : STD_LOGIC_VECTOR (31 downto 0) := "00111110011011001111111000110100";
  constant ap_const_lv32_3E58F215 : STD_LOGIC_VECTOR (31 downto 0) := "00111110010110001111001000010101";
  constant ap_const_lv32_3D150561 : STD_LOGIC_VECTOR (31 downto 0) := "00111101000101010000010101100001";
  constant ap_const_lv32_3EB152B2 : STD_LOGIC_VECTOR (31 downto 0) := "00111110101100010101001010110010";
  ....
```

First Convolution2D + Relu
Highly complex description
8535 VHDL lines vs 40 C++ lines (comments excluded)

3 output channels, kernel height of 5, and kernel width of 5
3 x 5 x 5 = 75 weights (float)

-0.18844394385814666748046875

RTL-based injection in the FP32 Lenet-5 (permanent faults on constants)

```
entity cnn_convolution2DRelu_1 is
port (
   ap_clk : IN STD_LOGIC;
...
end entity;

architecture behav of cnn_convolution2DRelu_1 is
   ...
   constant ap_const_lv32_BE40F773 : STD_LOGIC_VECTOR (31 downto 0) := "10111110010000001111011101110011";
   constant ap_const_lv32_3E26A585 : STD_LOGIC_VECTOR (31 downto 0) := "00111110001001101010010110000101";
   constant ap_const_lv32_3E6CFE34 : STD_LOGIC_VECTOR (31 downto 0) := "00111110011011001111111000110100";
   constant ap_const_lv32_3E58F215 : STD_LOGIC_VECTOR (31 downto 0) := "00111110010110001111001000010101";
   constant ap_const_lv32_3D150561 : STD_LOGIC_VECTOR (31 downto 0) := "00111101000101010000010101100001";
   constant ap_const_lv32_3EB152B2 : STD_LOGIC_VECTOR (31 downto 0) := "00111110101100010101001010110010";
   ....
```

First Convolution2D + Relu
Highly complex description
8535 VHDL lines vs 40 C++ lines (comments excluded)

3 output channels, kernel height of 5, and kernel width of 5
3 x 5 x 5 = 75 weights (float)

-0.188443943858146667480468 75

permanent fault must be injected using the **change** command

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

RTL-based injection in the FP32 Lenet-5 (transient faults on signals)

Institute ITACA Information and Communication Technologies

```
entity cnn_convolution2DRelu_KERNEL_CONV_2_0_0_0_ROM_AUTO_1R is
...
port (
   address0      : in std_logic_vector(AddressWidth-1 downto 0);
...
);
end entity;

architecture rtl of cnn_convolution2DRelu_KERNEL_CONV_2_0_0_0_ROM_AUTO_1R is

   type mem_array is array (0 to AddressRange-1) of std_logic_vector (DataWidth-1 downto 0);
   signal mem0 : mem_array := (
      0 => "10111100110110101010110010010111",
      1 => "00111110000111010010001000111010",
      2 => "00111101000111100010001011010110",
      3 => "10111101100001110111011100100010",
      4 => "10111110001101000010101000011011",
      5 => "10111101101001001001001001100011“
   );
   ...
```

Second Convolution2D + Relu
Highly complex description
22737 VHDL lines vs 40 C++ lines (comments and memories descriptions excluded)

RTL-based injection in the FP32 Lenet-5 (transient faults on signals)

```
entity cnn_convolution2DRelu_KERNEL_CONV_2_0_0_0_ROM_AUTO_1R is
...
port (
   address0      : in std_logic_vector(AddressWidth-1 downto 0);
...
);
end entity;

architecture rtl of cnn_convolution2DRelu_KERNEL_CONV_2_0_0_0_ROM_AUTO_1R is

   type mem_array is array (0 to AddressRange-1) of std_logic_vector (DataWidth-1 downto 0);
   signal mem0 : mem_array := (
      0 => "10111100110110101010110010010111",
      1 => "00111110000111010010001000111010",
      2 => "00111101000111100010001011010110",
      3 => "10111101100001110111011100100010",
      4 => "10111110001101000010101000011011",
      5 => "10111101101001001001001001100011“
   );
   ...
```

Second Convolution2D + Relu
Highly complex description
22737 VHDL lines vs 40 C++ lines (comments and memories descriptions excluded)

6 output channels, kernel height of 5, and kernel width of 5
6 x 5 x 5 = 150 weights (float) to process

-0.02669362537562847137451171875

RTL-based injection in the FP32 Lenet-5 (transient faults on signals)

```
entity cnn_convolution2DRelu_KERNEL_CONV_2_0_0_0_ROM_AUTO_1R is
...
port (
   address0      : in std_logic_vector(AddressWidth-1 downto 0);
...
);
end entity;

architecture rtl of cnn_convolution2DRelu_KERNEL_CONV_2_0_0_0_ROM_AUTO_1R is

   type mem_array is array (0 to AddressRange-1) of std_logic_vector (DataWidth-1
downto 0);
   signal mem0 : mem_array := (
      0 => "10111100110110101010110010010111",
      1 => "00111110000111010010001000111010",
      2 => "00111101000111100010001011010110",
      3 => "10111101100001110111011100100010",
      4 => "10111110001101000010101000011011",
      5 => "10111101101001001001001001100011“
   );
   ...
```

Second Convolution2D + Relu
Highly complex description
22737 VHDL lines vs 40 C++ lines (comments and memories descriptions excluded)

6 output channels, kernel height of 5, and kernel width of 5
6 x 5 x 5 = 150 weights (float) to process

-0.02669362537562847137451171875

transient fault must be injected using the **force** command (although it will actually be a permanent fault because the memory content is never rewritten)

RTL-based injection in the INT8 Lenet-5

```
entity cnn_convolution2DRelu_1 is
...
end entity;
architecture behav of cnn_convolution2DRelu_1 is
   ...
   constant ap_const_lv32_717 : STD_LOGIC_VECTOR (31 downto 0) := "00000000000000000000011100010
   constant ap_const_lv32_9E : STD_LOGIC_VECTOR (31 downto 0) := "00000000000000000000000010011110;
   constant ap_const_lv32_DE0 : STD_LOGIC_VECTOR (31 downto 0) := "00000000000000000000110111100000";
   constant ap_const_lv32_5F8198 : STD_LOGIC_VECTOR (31 downto 0) := "00000000010111111000000110011000";
   constant ap_const_lv32_524BD1 : STD_LOGIC_VECTOR (31 downto 0) := "00000000010100100100101111010001";
   constant ap_const_lv32_4A5C13 : STD_LOGIC_VECTOR (31 downto 0) := "00000000010010100101110000010011";    ....
```

First Convolution2D + Relu
Highly complex description
8535 VHDL lines vs 54 C++ lines (comments and memories descriptions excluded)

```
entity cnn_convolution2DRelu_1_p_ZL13KERNEL_CONV_1_0_0_0_ROM_AUTO_1R is
...
end entity;
architecture rtl of cnn_convolution2DRelu_1_p_ZL13KERNEL_CONV_1_0_0_0_ROM_AUTO_1R is
...
type mem_array is array (0 to AddressRange-1) of std_logic_vector (DataWidth-1 downto 0);
signal mem0 : mem_array := (
   0 => "11110000", 1 => "01110110", 2 => "10001011");
```

RTL-based injection in the INT8 Lenet-5 (constants)

3 output channels : 3 bias (**int32**)

First Convolution2D + Relu
Highly complex description
8535 VHDL lines vs 54 C++ lines (comments and memories descriptions excluded)

```
entity cnn_convolution2DRelu_1 is
...
end entity;
architecture behav of cnn_convolution2DRelu_1 is
   ...
   constant ap_const_lv32_717 : STD_LOGIC_VECTOR (31 downto 0) := "00000000000000000000011100010
   constant ap_const_lv32_9E : STD_LOGIC_VECTOR (31 downto 0) := "00000000000000000000000010011110";
   constant ap_const_lv32_DE0 : STD_LOGIC_VECTOR (31 downto 0) := "00000000000000000000110111100000";
   constant ap_const_lv32_5F8198 : STD_LOGIC_VECTOR (31 downto 0) := "00000000010111111000000110011000";
   constant ap_const_lv32_524BD1 : STD_LOGIC_VECTOR (31 downto 0) := "00000000010100100100101111010001";
   constant ap_const_lv32_4A5C13 : STD_LOGIC_VECTOR (31 downto 0) := "00000000010010100101110000010011";   ....
```

3 output channels : 3 M (**int32**)

```
entity cnn_convolution2DRelu_1_p_ZL13KERNEL_CONV_1_0_0_0_ROM_AUTO_1R is
...
end entity;
architecture rtl of cnn_convolution2DRelu_1_p_ZL13KERNEL_CONV_1_0_0_0_ROM_AUTO_1R is
...
type mem_array is array (0 to AddressRange-1) of std_logic_vector (DataWidth-1 downto 0);
signal mem0 : mem_array := (
   0 => "11110000", 1 => "01110110", 2 => "10001011");
```

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

RTL-based injection in the INT8 Lenet-5 (constants + signals)

Institute ITACA Information and Communication Technologies

3 output channels : 3 bias (**int32**)

First Convolution2D + Relu
Highly complex description
8535 VHDL lines vs 54 C++ lines (comments and memories descriptions excluded)

```
entity cnn_convolution2DRelu_1 is
...
end entity;
architecture behav of cnn_convolution2DRelu_1 is
  ...
  constant ap_const_lv32_717 : STD_LOGIC_VECTOR (31 downto 0) := "00000000000000000000011100010
  constant ap_const_lv32_9E : STD_LOGIC_VECTOR (31 downto 0) := "00000000000000000000000010011110";
  constant ap_const_lv32_DE0 : STD_LOGIC_VECTOR (31 downto 0) := "00000000000000000000110111100000";
  constant ap_const_lv32_5F8198 : STD_LOGIC_VECTOR (31 downto 0) := "00000000010111111000000110011000";
  constant ap_const_lv32_524BD1 : STD_LOGIC_VECTOR (31 downto 0) := "00000000010100100100101111010001";
  constant ap_const_lv32_4A5C13 : STD_LOGIC_VECTOR (31 downto 0) := "00000000010010100101110000010011";   ....
```

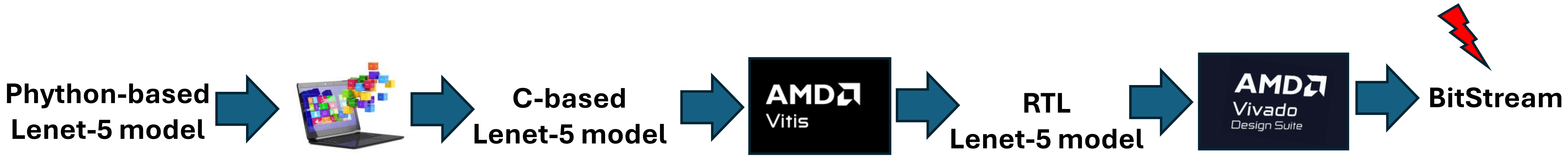
3 output channels : 3 M (**int32**)

```
entity cnn_convolution2DRelu_1_p_ZL13KERNEL_CONV_1_0_0_0_ROM_AUTO_1R is
...
end entity;
architecture rtl of cnn_convolution2DRelu_1_p_ZL13KERNEL_CONV_1_0_0_0_ROM_AUTO_1R is
...
type mem_array is array (0 to AddressRange-1) of std_logic_vector (DataWidth-1 downto 0);
signal mem0 : mem_array := (
  0 => "11110000", 1 => "01110110", 2 => "10001011");
```

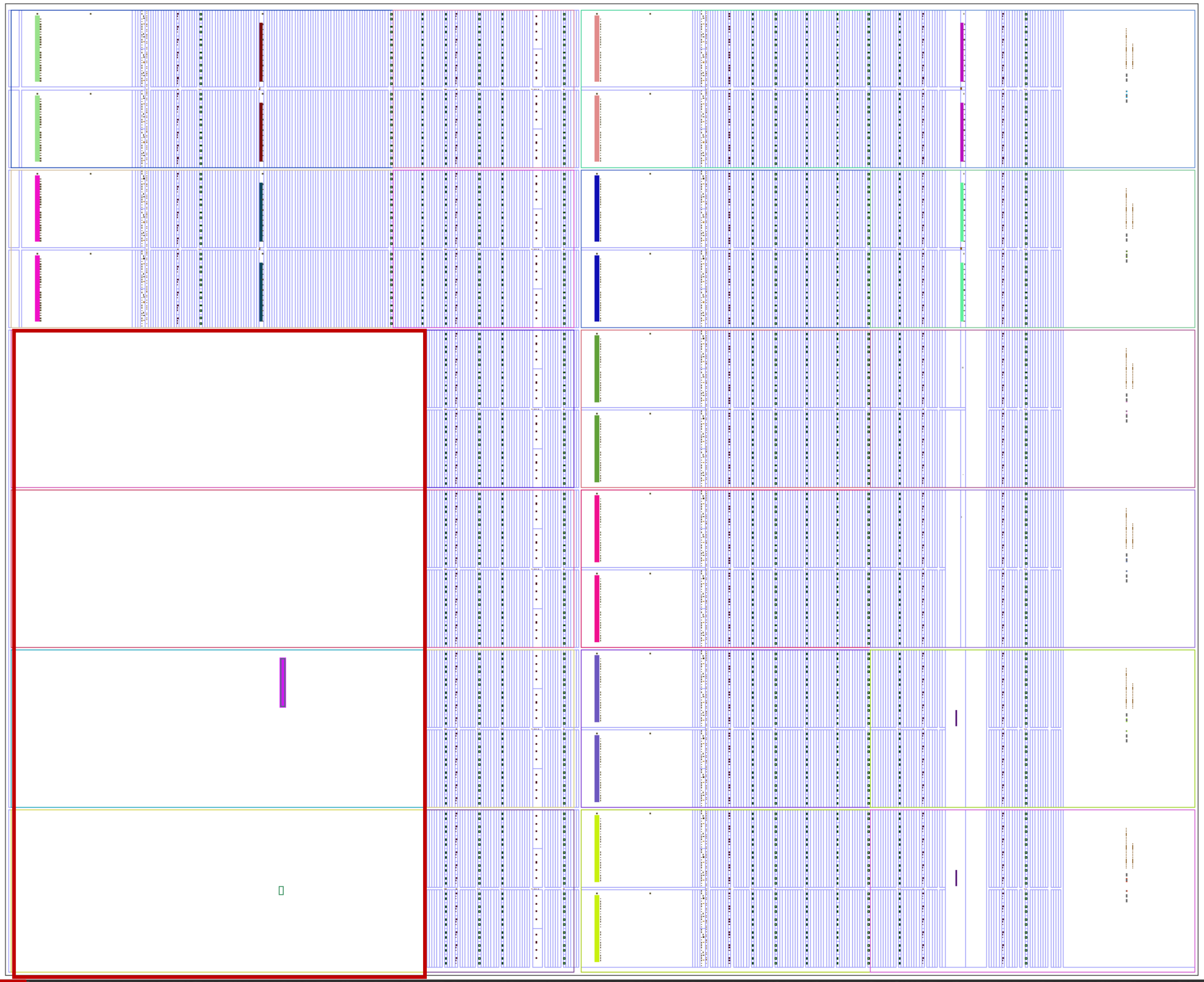
kernel height of 5, and kernel width of 5
5 x 5 = 25 memories (int8) to process

Levels of injection

- FP32
  - Python
  - C
- INT8
  - Python
  - C
- RTL
- **FPGA**



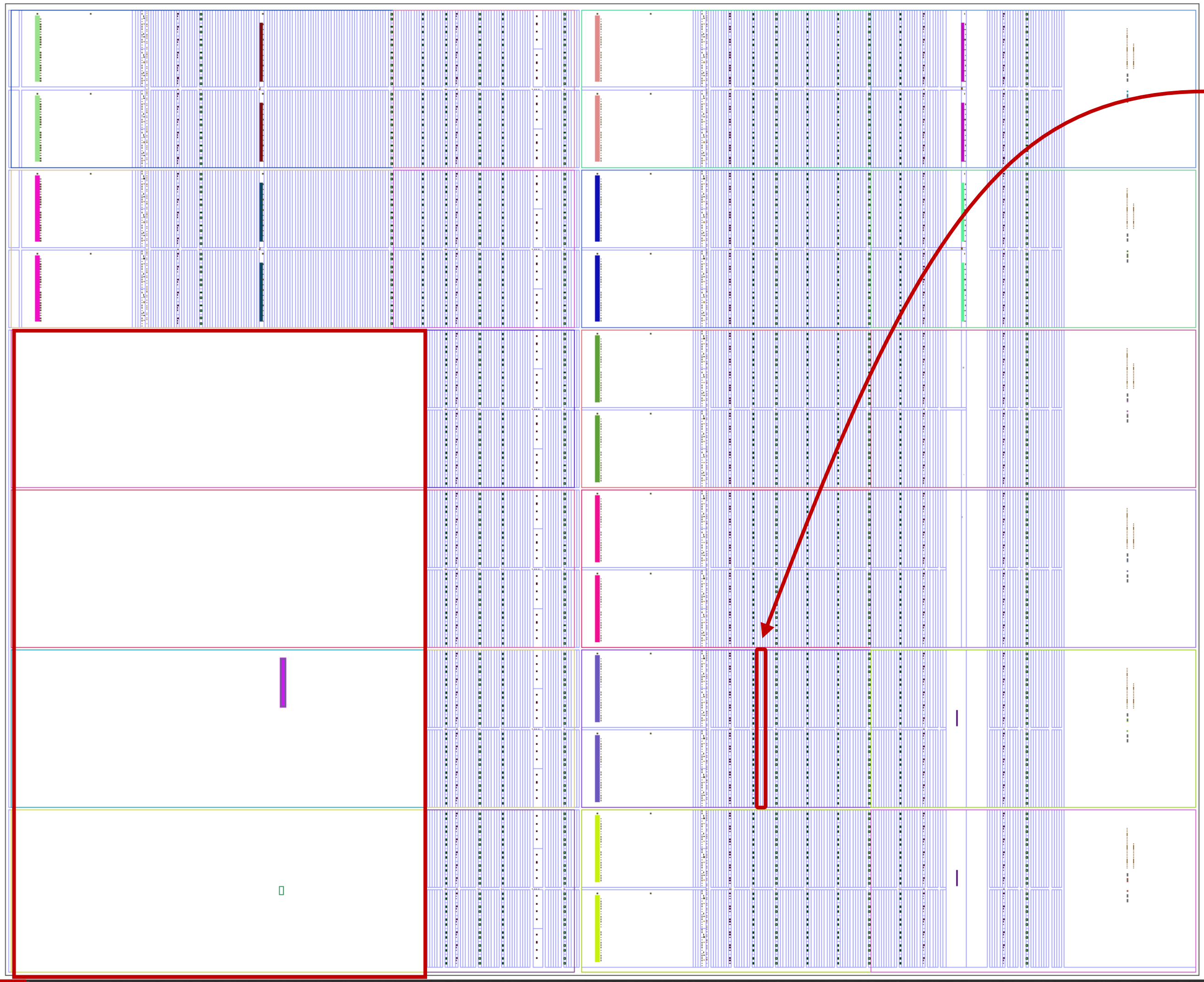
Architecture of the AMD Zynq UltraScale+ device



UNIVERSITAT POLITÈCNICA DE VALÈNCIA

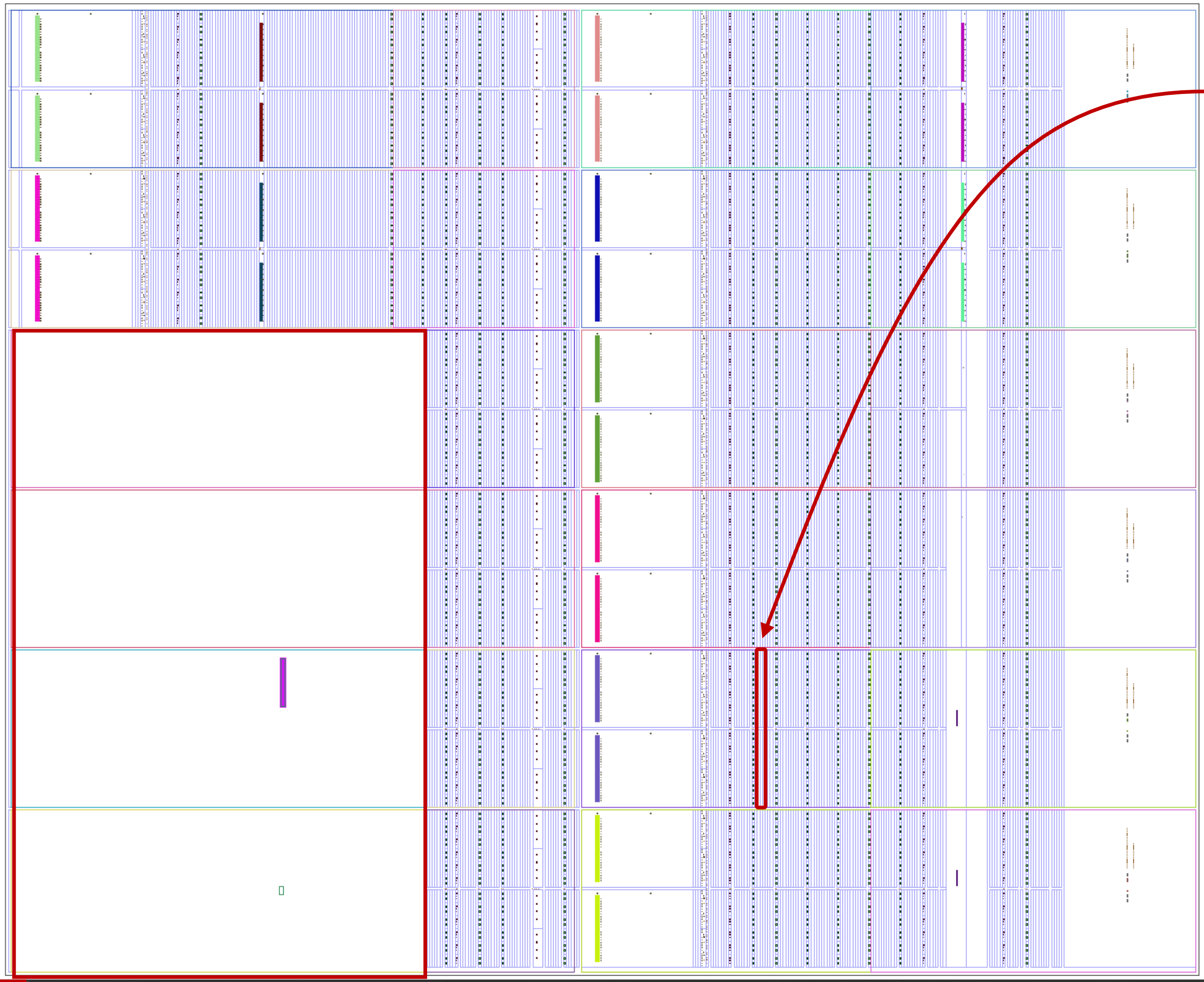
Architecture of the AMD Zynq UltraScale+ device

Institute ITACA Information and Communication Technologies



Frame
Smallest addressable unit of configuration memory
2,976 bits (93 x 32-bit words)

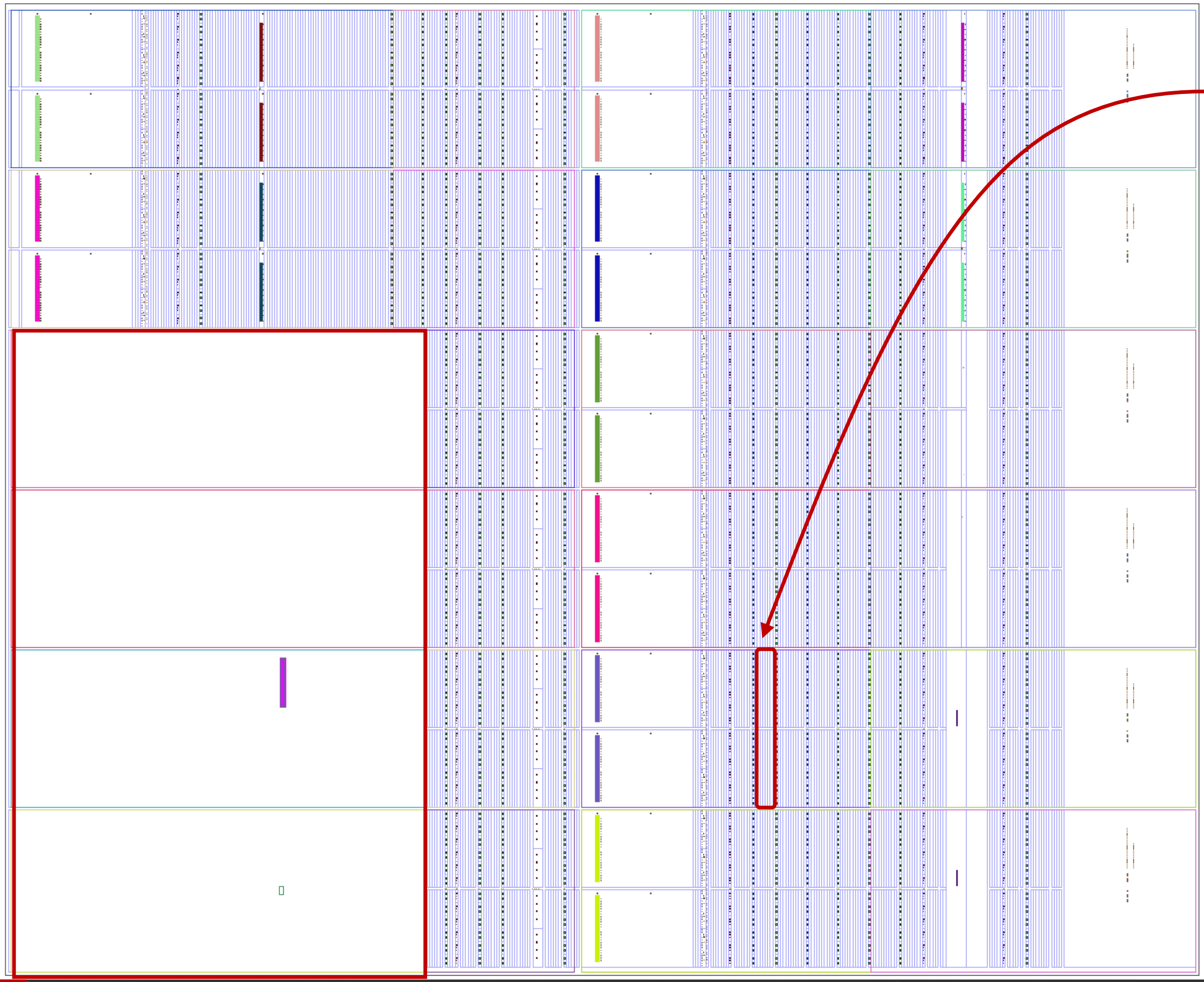
Architecture of the AMD Zynq UltraScale+ device



Frame
Smallest addressable unit of configuration memory
2,976 bits (93 x 32-bit words)

ZU7 device
Configuration bitstream length: 154,488,736 bits
20,956 frames x (93 x 32-bit words)

Architecture of the AMD Zynq UltraScale+ device



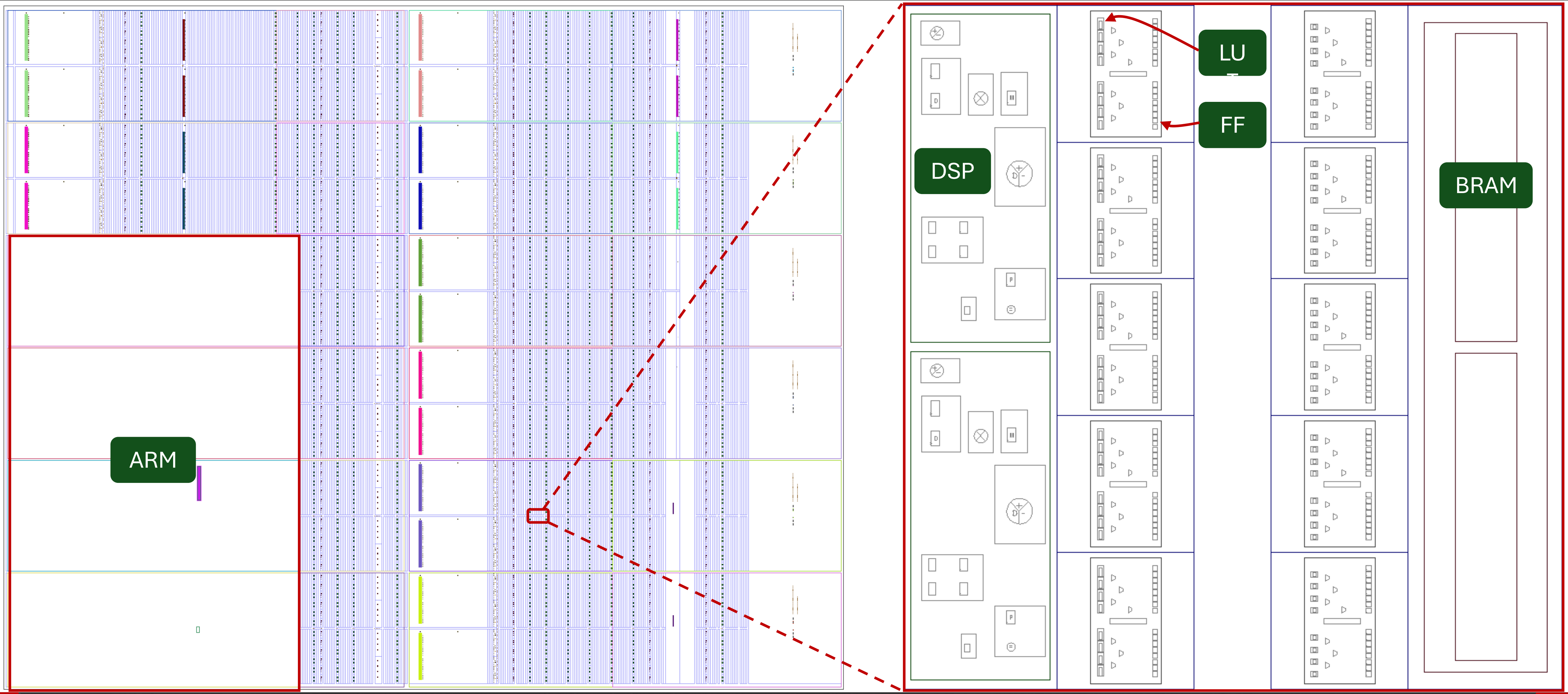
Frame
Smallest addressable unit of configuration memory
2,976 bits (93 x 32-bit words)

ZU7 device
Configuration bitstream length: 154,488,736 bits
20,956 frames x (93 x 32-bit words)

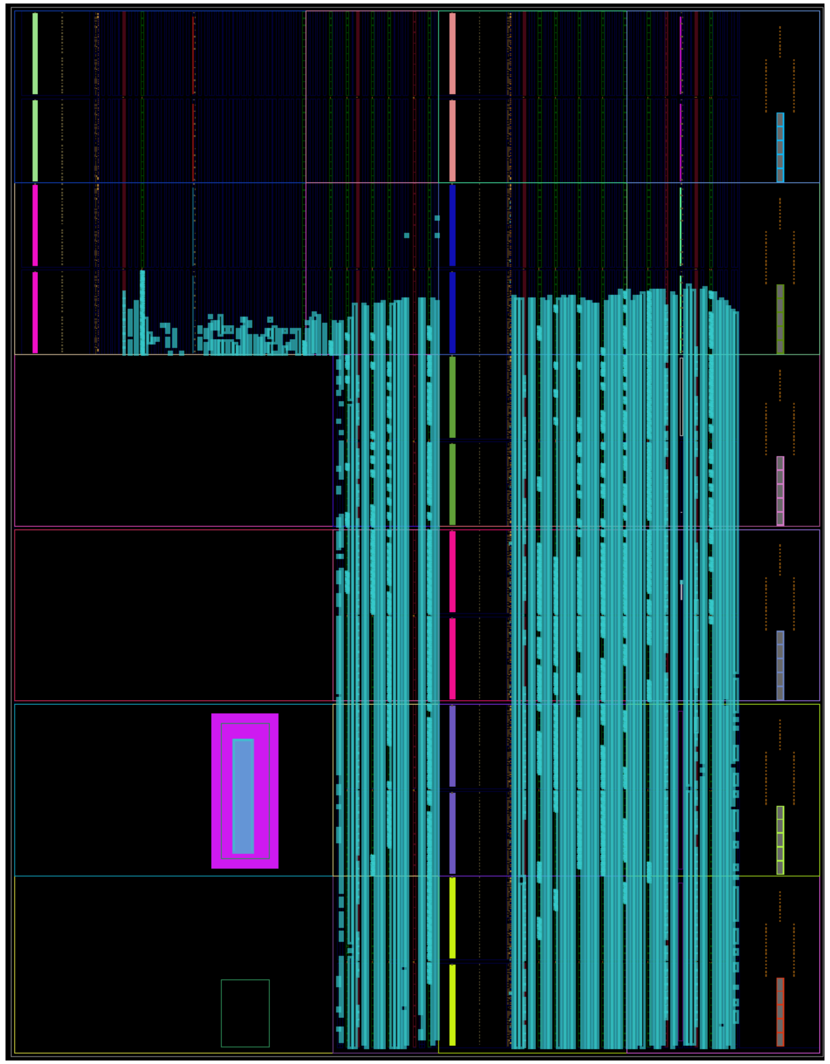
Frame Address Register (FAR)

[26:24]	[23:18]	[17:8]	[7:0]
Block Type (CLB/IO/CLK = 000, BRAM = 001)	Row Address (increments from bottom to top)	Column Address ((increments from left to right)	Minor Address (frame within a major column)

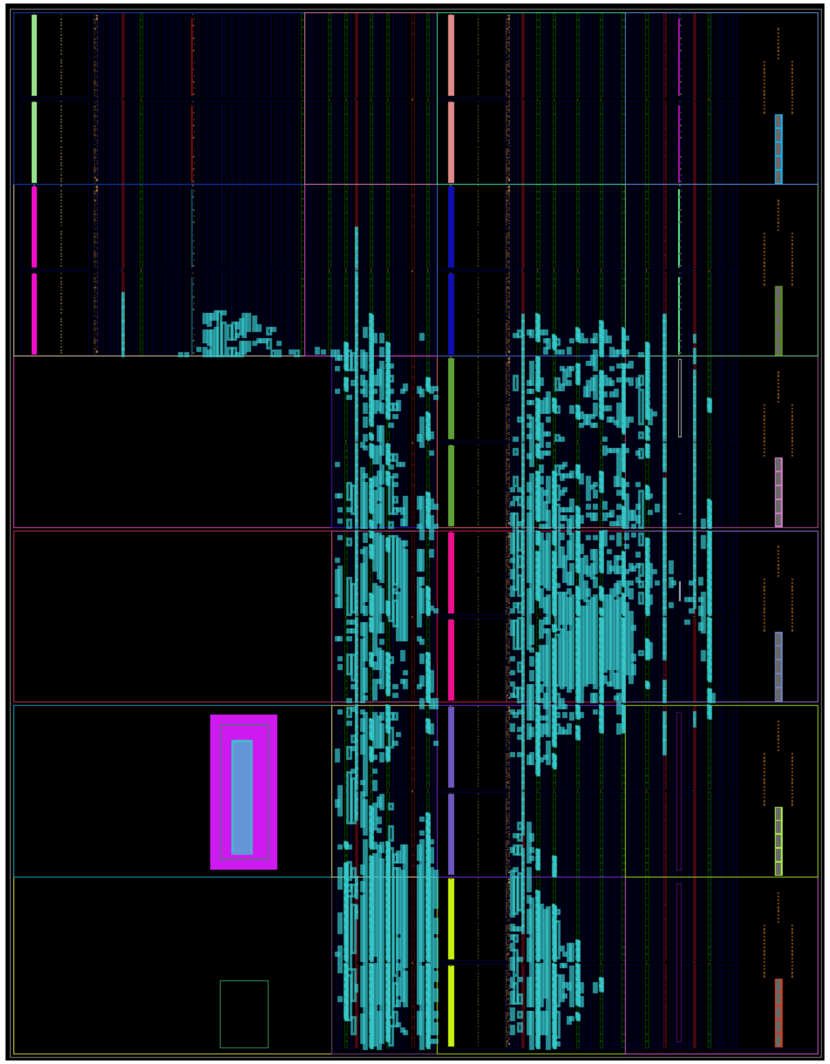
Architecture of the AMD Zynq UltraScale+ device



Implementation of Lenet-5 on an AMD Zynq UltraScale+ device: Float vs Quantized



Lenet-5 (float)



Lenet-5 (quantized
)

Implementation of a CNN on an AMD Zynq UltraScale+ device

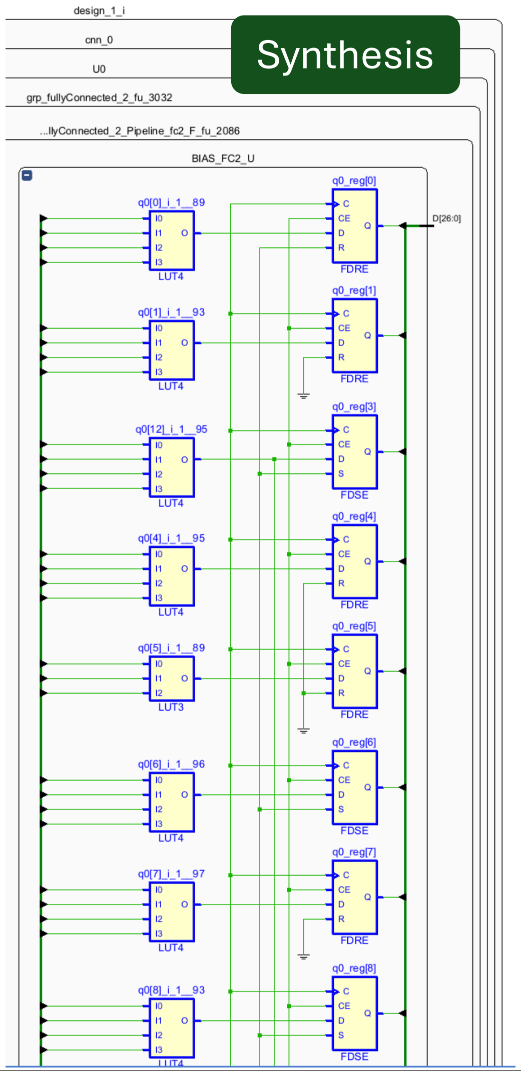
```
entity cnn_FC_2_Pipeline_fc2_F_BIAS_FC2_ROM_AUTO_1R is
...
end entity;

architecture rtl of
cnn_FC_2_Pipeline_fc2_F_BIAS_FC2_ROM_AUTO_1R is

  signal address0_tmp : std_logic_vector(AddressWidth-1 downto 0);

  type mem_array is array (0 to AddressRange-1) of
            std_logic_vector (DataWidth-1 downto 0);
  signal mem0 : mem_array := (
    0 => "00111101101111110110101001110111 0",
    1 => "00111101100010111001111011101101",
    2 => "10111101000110100101010111101100",
    3 => "00111100111101110110100110110100",
    4 => "10111011111010001001011101001001 1",
    5 => "00111100111110010111111110011010",
    6 => "10111101011110000001101000011000",
    7 => "00111100100001100110001110010011",
    8 => "10111001011001111100001101101110",
    9 => "00111100001010001110101111011000“
  );
...
```

RTL

Implementation of a CNN on an AMD Zynq UltraScale+ device

```
entity cnn_FC_2_Pipeline_fc2_F_BIAS_FC2_ROM_AUTO_1R is
...
end entity;

architecture rtl of
cnn_FC_2_Pipeline_fc2_F_BIAS_FC2_ROM_AUTO_1R is

  signal address0_tmp : std_logic_vector(AddressWidth-1 downto 0);

  type mem_array is array (0 to AddressRange-1) of
              std_logic_vector (DataWidth-1 downto 0);
  signal mem0 : mem_array := (
    0 => "00111101101111101101010011101110",
    1 => "00111101100010111001111011101101",
    2 => "10111101000110100101010111101100",
    3 => "00111100111101110110100110110100",
    4 => "101110111110100010010111010010011",
    5 => "00111100111110010111111110011010",
    6 => "10111101011110000001101000011000",
    7 => "00111100100001100110001110010011",
    8 => "101110010110011111100001101101110",
    9 => "00111100001010001110101111011000“
  );
...
```

RTL

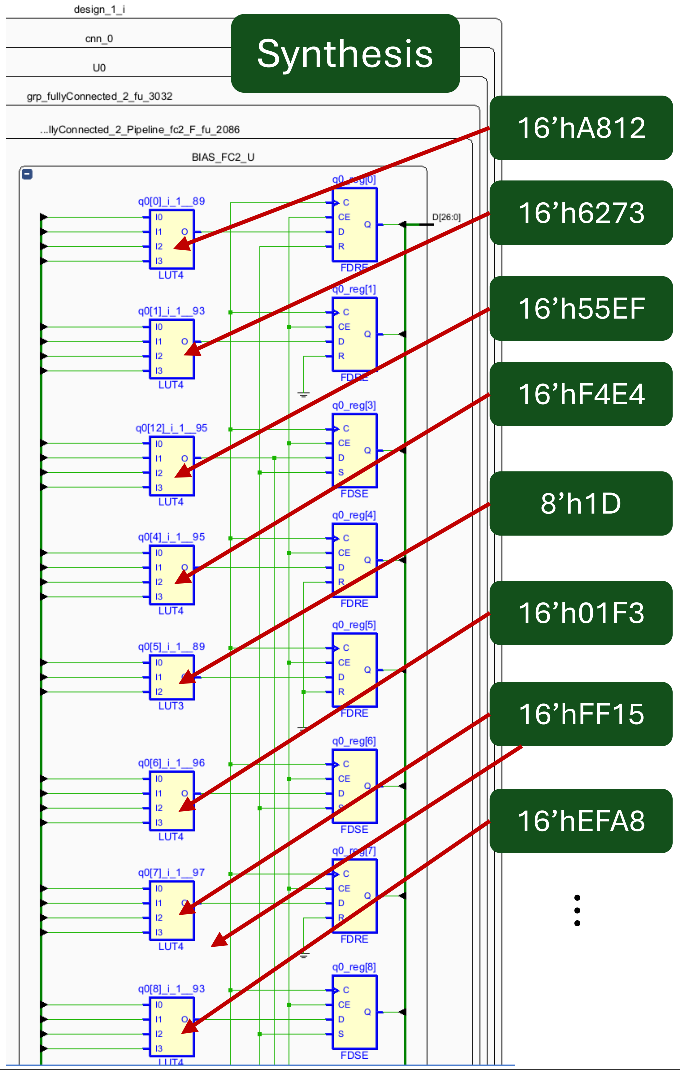


Implementation of a CNN on an AMD Zynq UltraScale+ device

```
entity cnn_FC_2_Pipeline_fc2_F_BIAS_FC2_ROM_AUTO_1R is
...
end entity;

architecture rtl of
cnn_FC_2_Pipeline_fc2_F_BIAS_FC2_ROM_AUTO_1R is

  signal address0_tmp : std_logic_vector(AddressWidth-1 downto 0);

  type mem_array is array (0 to AddressRange-1) of
             std_logic_vector (DataWidth-1 downto 0);
  signal mem0 : mem_array := (
    0 => "00111101101111101101010011101110",
    1 => "00111101100010111001111011101101",
    2 => "10111101000110100101010111101100",
    3 => "00111100111101110110100110110100",
    4 => "10111011110100010010111010010011",
    5 => "00111100111110010111111110011010",
    6 => "10111101011110000001101000011000",
    7 => "00111100100001100110001110010011",
    8 => "10111001011001111110000110110110",
    9 => "00111100001010001110101111011000“
  );
...
```

RTL

Synthesis



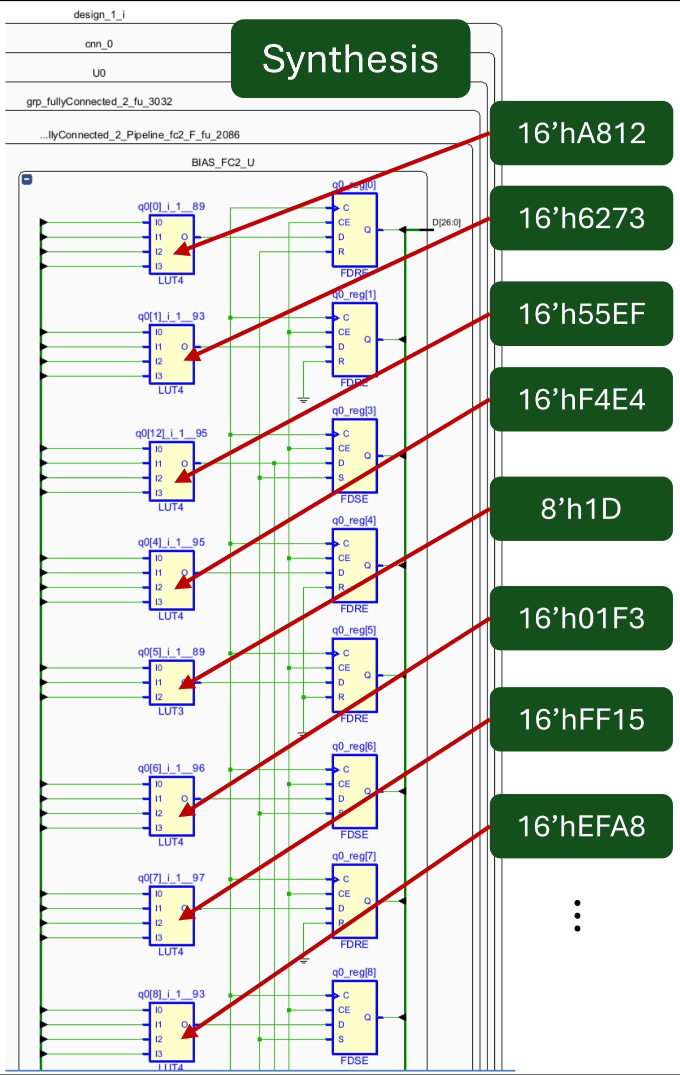
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

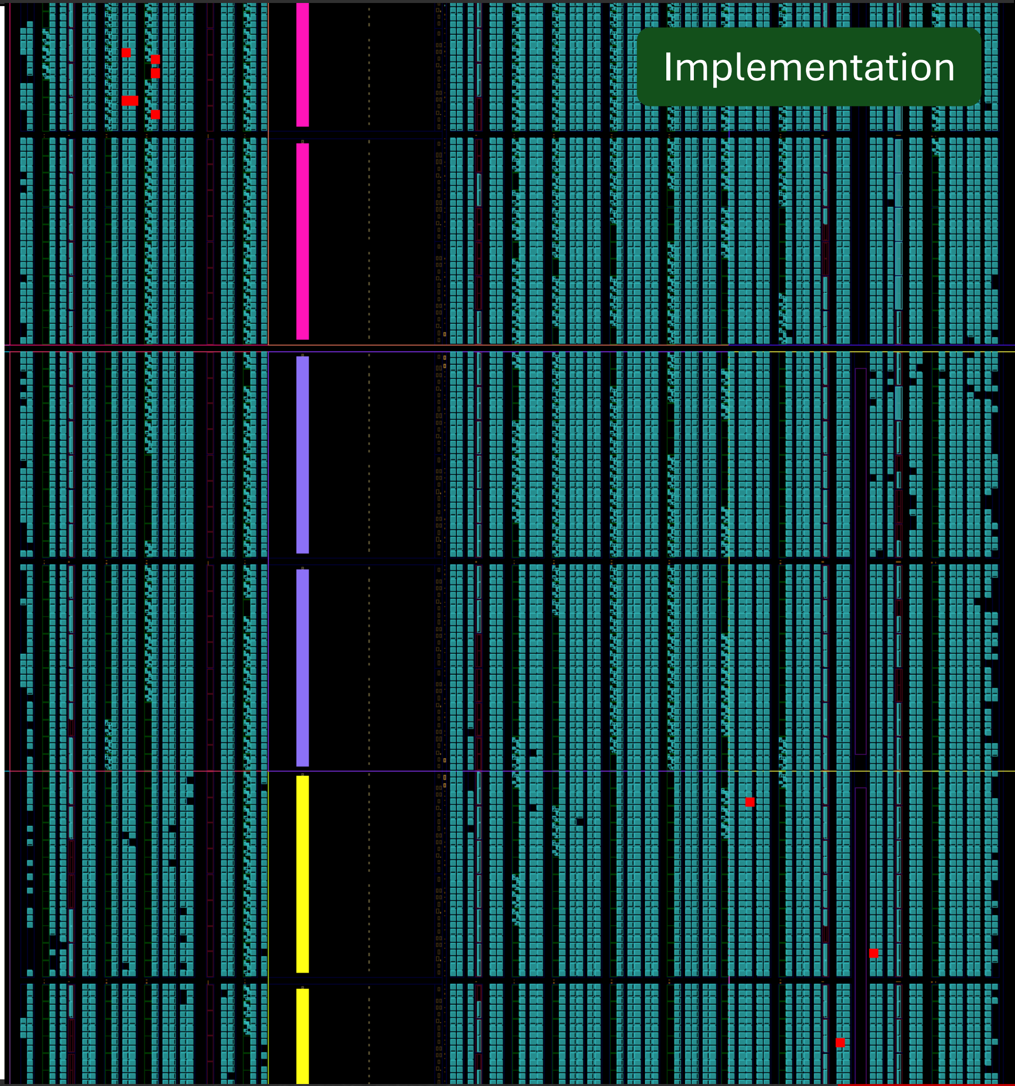
Institute ITACA Information and Communication Technologies

Implementation of a CNN on an AMD Zynq UltraScale+ device

```
entity cnn_FC_2_Pipeline_fc2_F_BIAS_FC2_ROM_AUTO_1R is
...
end entity;

architecture rtl of
cnn_FC_2_Pipeline_fc2_F_BIAS_FC2_ROM_AUTO_1R is

  signal address0_tmp : std_logic_vector(AddressWidth-1 downto 0);

  type mem_array is array (0 to AddressRange-1) of
        std_logic_vector (DataWidth-1 downto 0);
  signal mem0 : mem_array := (
    0 => "001111011011111101101010011101110",
    1 => "00111101100010111001111011101101",
    2 => "10111101000110100101010111101100",
    3 => "00111100111101110110100110110100",
    4 => "10111011110100010010111010010011",
    5 => "00111100111110010111111110011010",
    6 => "10111101011110000001101000011000",
    7 => "00111100100001100110001110010011",
    8 => "10111001011001111110000110110110",
    9 => "00111100001010001110101111011000“
  );
...
```

RTL

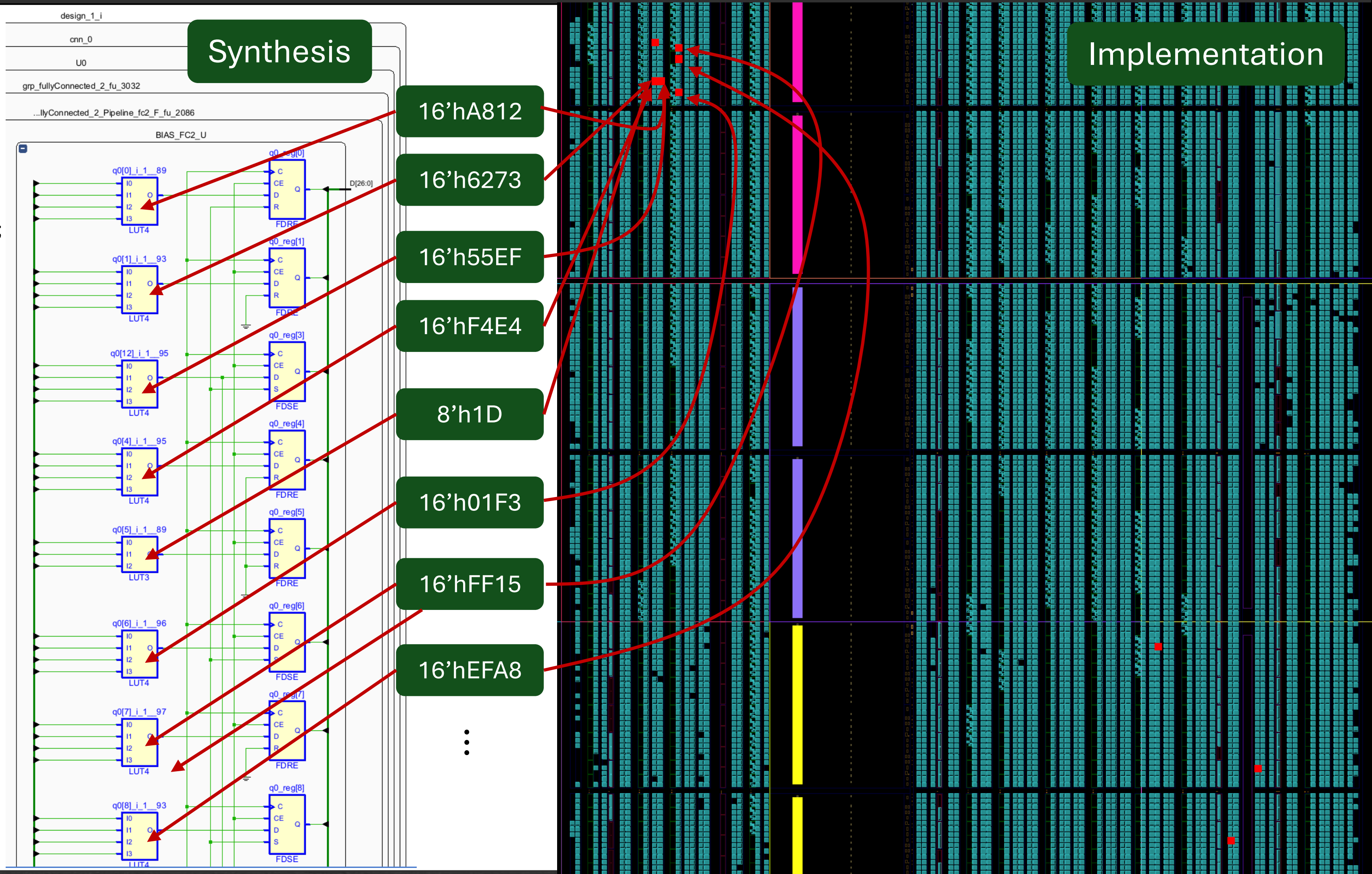




Implementation of a CNN on an AMD Zynq UltraScale+ device

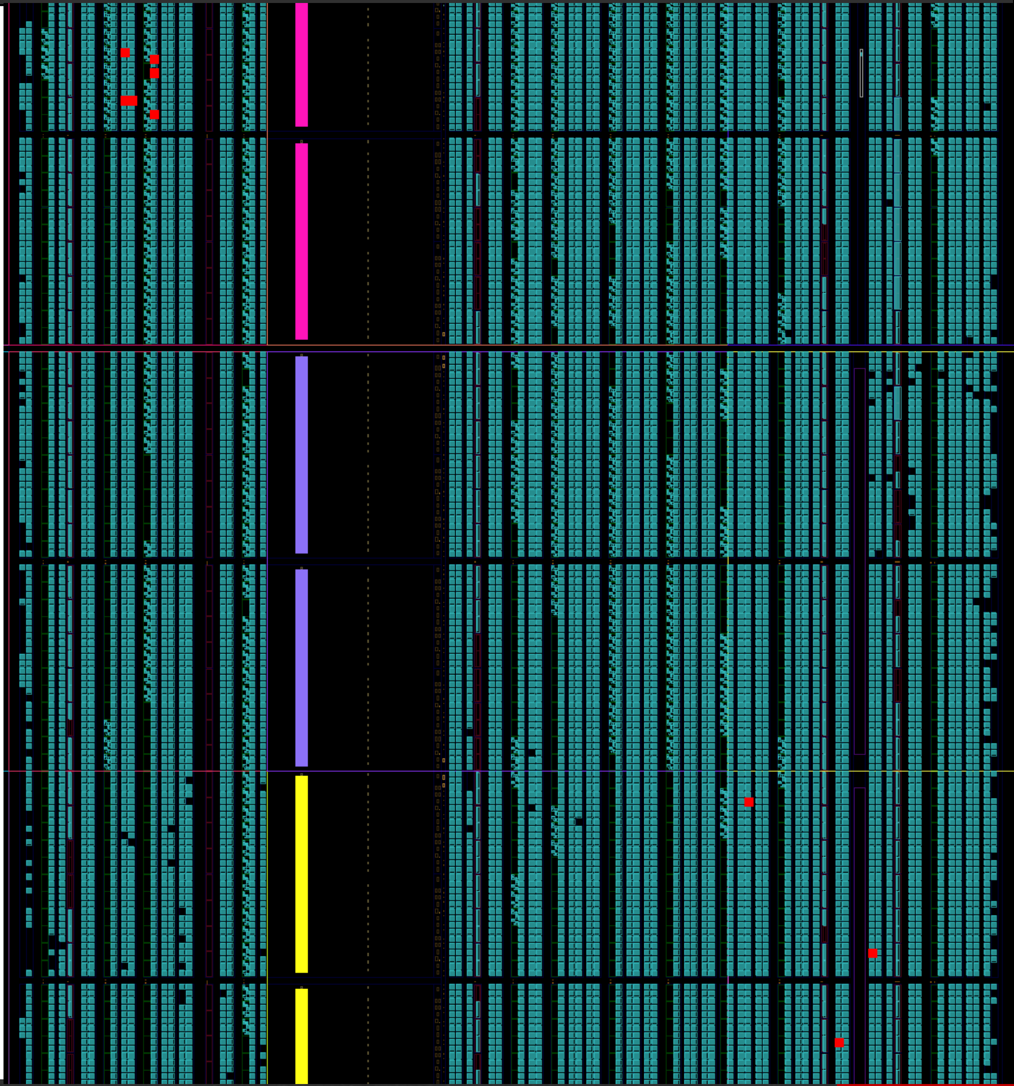
```
entity cnn_FC_2_Pipeline_fc2_F_BIAS_FC2_ROM_AUTO_1R is
...
end entity;

architecture rtl of
cnn_FC_2_Pipeline_fc2_F_BIAS_FC2_ROM_AUTO_1R is

  signal address0_tmp : std_logic_vector(AddressWidth-1 downto 0);

  type mem_array is array (0 to AddressRange-1) of
        std_logic_vector (DataWidth-1 downto 0);
  signal mem0 : mem_array := (
    0 => "00111101101111110110101001110110",
    1 => "00111101100010111001111011101101",
    2 => "10111101000110100101010111101100",
    3 => "00111100111101110110100110110100",
    4 => "10111011110100010010111010010011",
    5 => "00111100111110010111111110011010",
    6 => "10111101011110000001101000011000",
    7 => "00111100100001100110001110010011",
    8 => "10111001011001111110000110110110",
    9 => "00111100001010001110101111011000“
  );
...
```

RTL



Fault injection in an AMD Zynq UltraScale+ device

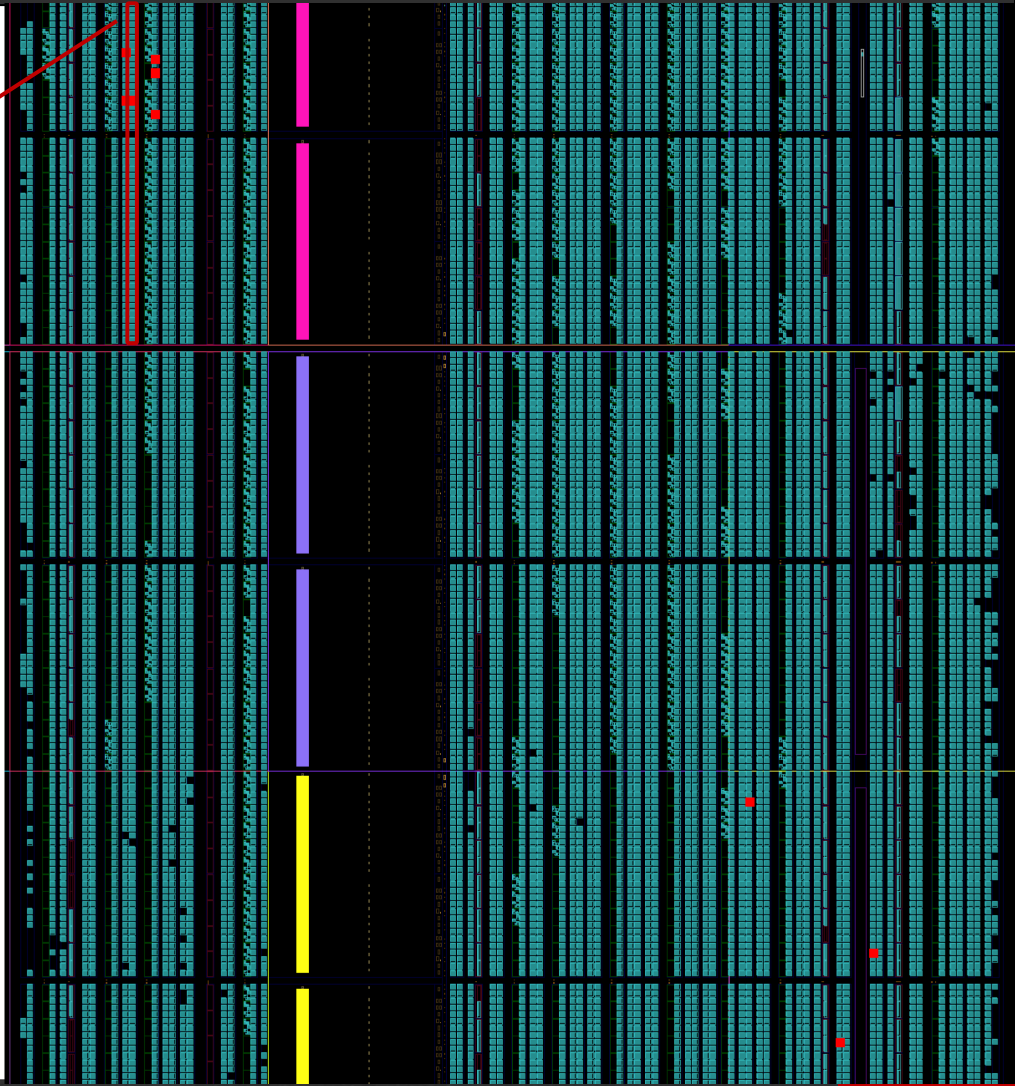
```
# Injection controlled by the integrated ARM core
enable_clock_throttle()
run_clock(FOR_INJECTION_TIME_CYCLES)
frameAddressRegister = getFAR(targetLut)
```



Fault injection in an AMD Zynq UltraScale+ device

```
# Injection controlled by the integrated ARM core
enable_clock_throttle()
run_clock(FOR_INJECTION_TIME_CYCLES)
frameAddressRegister = getFAR(targetLut)
frame = read_frame(frameAddressRegister) # 2976 bits
```

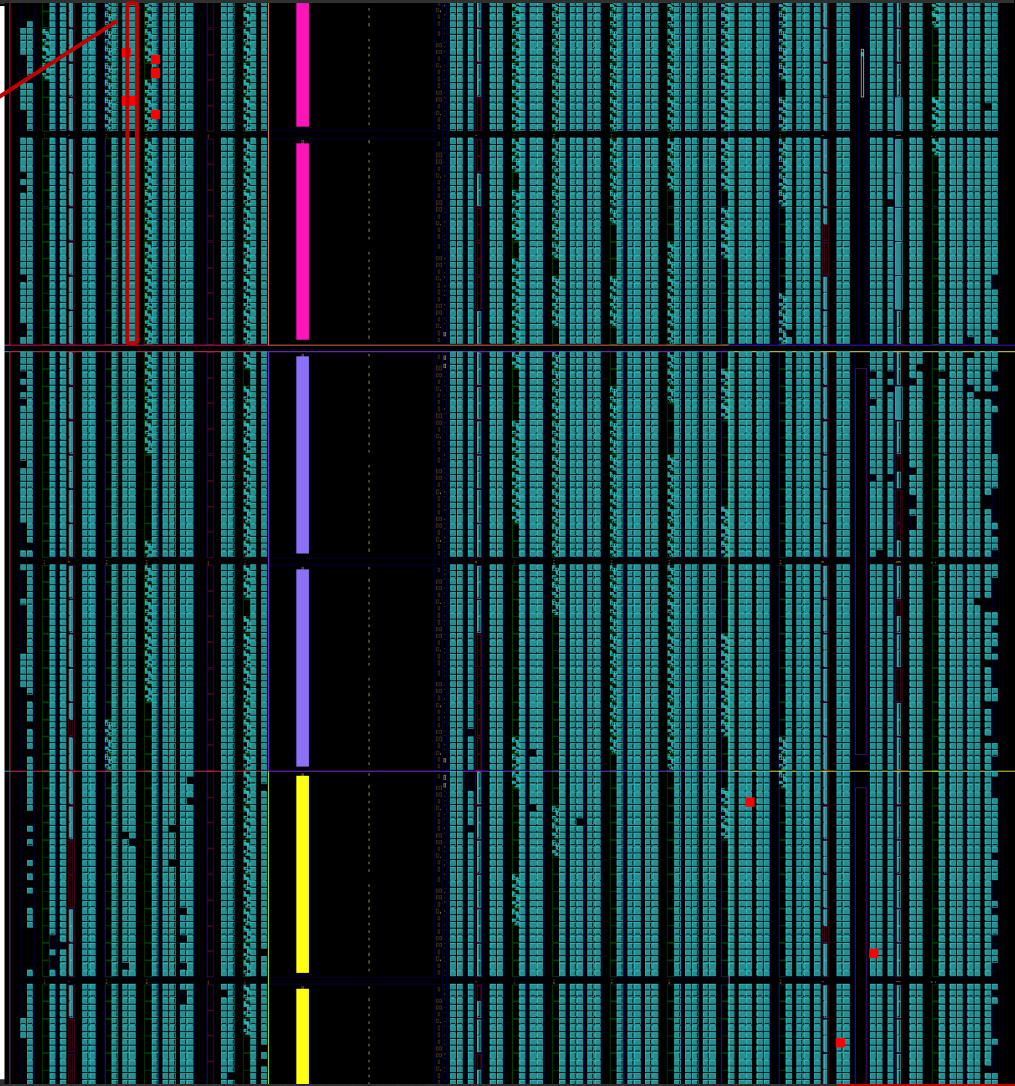
```
Address   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
0106a5d0 0c 01 65 22 40 b6 10 00 00 49 12 91 00 c8 48 90
0106a5e0 c8 b0 5b 20 a0 05 80 d2 64 50 6d 14 99 10 9b 69
0106a5f0 14 00 00 08 21 8a 24 14 c0 30 93 61 b2 28 26 0a
0106a600 00 5b 25 22 0b b4 90 68 06 69 24 c0 00 00 21 34
0106a610 81 92 c0 00 00 48 92 52 24 49 29 96 88 12 18 4d

                         ...
```



Fault injection in an AMD Zynq UltraScale+ device

```
# Injection controlled by the integrated ARM core
enable_clock_throttle()
run_clock(FOR_INJECTION_TIME_CYCLES)
frameAddressRegister = getFAR(targetLut)
frame = read_frame(frameAddressRegister) # 2976 bits
lutContent = get_lut_content(frame) # 64 bits
```

```
Address   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
0106a5d0 0c 01 65 22 40 b6 10 00 00 49 12 91 00 c8 48 90
0106a5e0 c8 b0 5b 20 a0 05 80 d2 64 50 6d 14 99 10 9b 69
0106a5f0 14 00 00 08 21 8a 24 14 c0 30 93 61 b2 28 26 0a
0106a600 00 5b 25 22 0b b4 90 68 06 69 24 c0 00 00 21 34
0106a610 81 92 c0 00 00 48 92 52 24 49 29 96 88 12 18 4d
                          ...
```



UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Institute ITACA Information and Communication Technologies

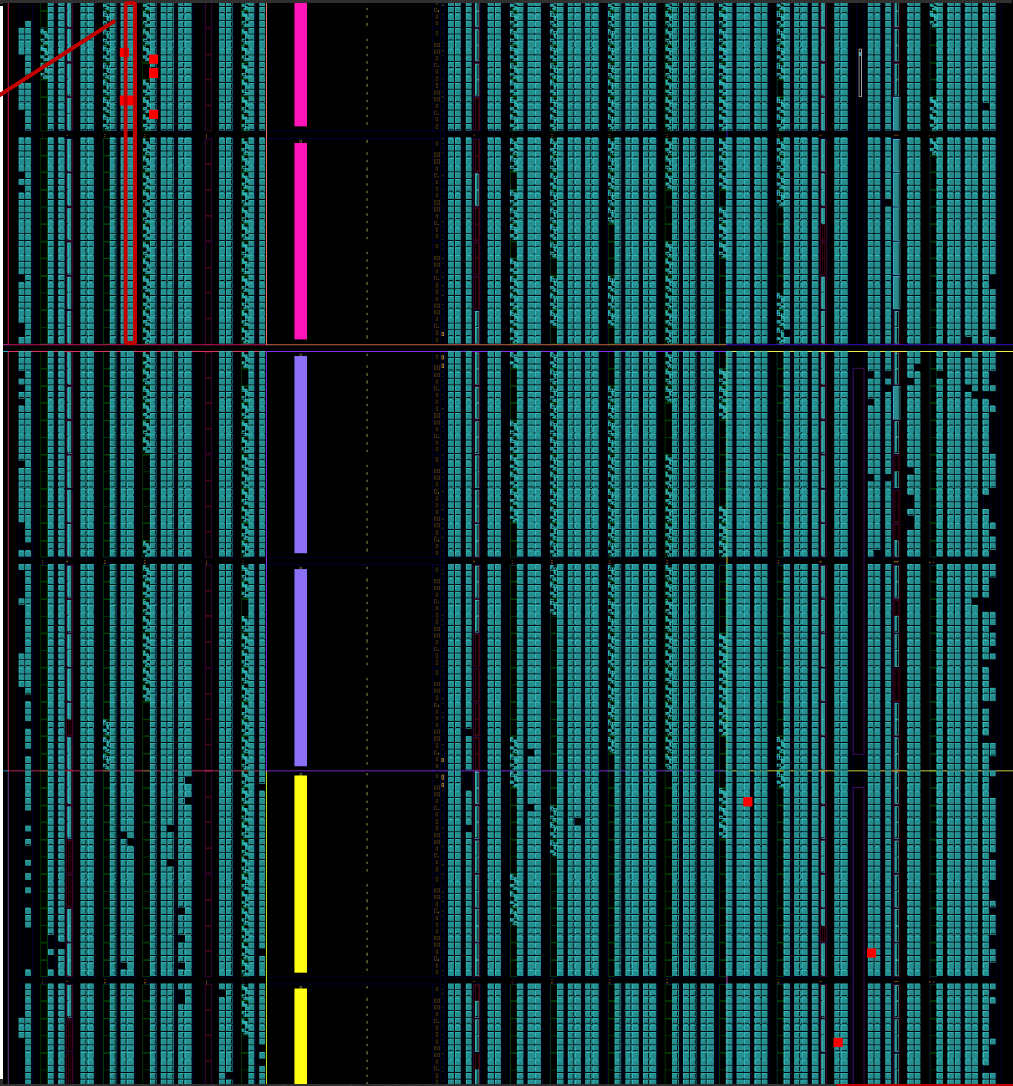
Fault injection in an AMD Zynq UltraScale+ device

```
# Injection controlled by the integrated ARM core
enable_clock_throttle()
run_clock(FOR_INJECTION_TIME_CYCLES)
frameAddressRegister = getFAR(targetLut)
frame = read_frame(frameAddressRegister) # 2976 bits
lutContent = get_lut_content(frame) # 64 bits
faultyLutContent = inject_fault(lutContent, bit) # invert the target bit
update_frame(faultyLutContent, frame)
```

```
Address   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
0106a5d0 0c 01 65 22 40 b6 10 00 00 49 12 91 00 c8 48 90
0106a5e0 c8 b0 5b 20 a0 05 80 d2 64 50 6d 14 99 10 9b 69
0106a5f0 14 00 00 08 21 8a 24 14 c0 30 93 61 b2 28 26 0a
0106a600 00 5b 25 22 0b b4 90 68 06 69 24 c0 00 00 21 34
0106a610 81 92 c0 00 00 48 92 52 24 49 29 96 88 12 18 4d
```

...

```
Address   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
0106a5d0 0c 01 65 22 40 b6 10 00 00 49 12 91 00 c8 48 90
0106a5e0 c8 b0 5b 20 a0 05 80 d2 64 50 6d 14 99 10 9b 69
0106a5f0 14 00 00 08 31 8a 24 14 c0 30 93 61 b2 28 26 0a
0106a600 00 5b 25 22 0b b4 90 68 06 69 24 c0 00 00 21 34
0106a610 81 92 c0 00 00 48 92 52 24 49 29 96 88 12 18 4d
```



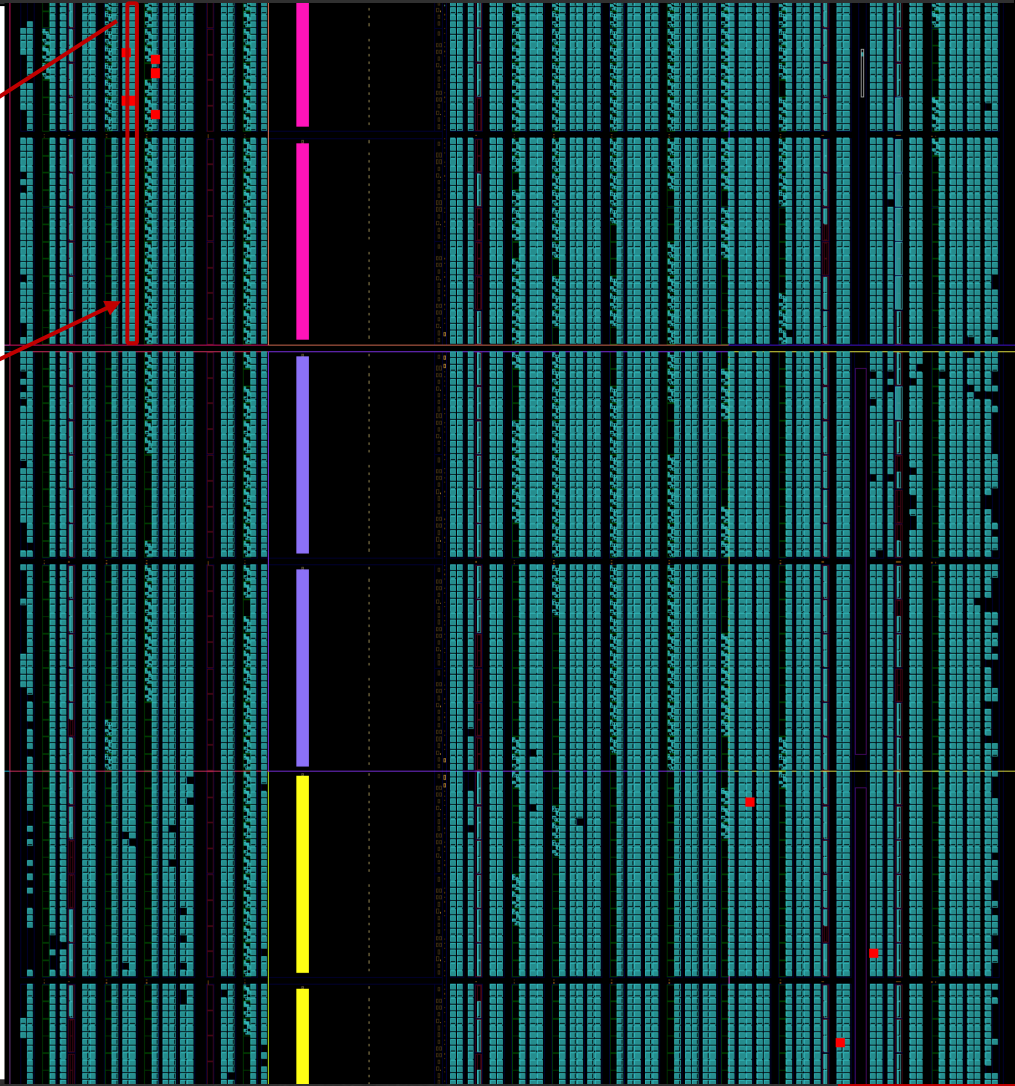
Fault injection in an AMD Zynq UltraScale+ device

```
# Injection controlled by the integrated ARM core
enable_clock_throttle()
run_clock(FOR_INJECTION_TIME_CYCLES)
frameAddressRegister = getFAR(targetLut)
frame = read_frame(frameAddressRegister) # 2976 bits
lutContent = get_lut_content(frame) # 64 bits
faultyLutContent = inject_fault(lutContent, bit) # invert the target bit
update_frame(faultyLutContent, frame)
write_frame(frame, frameAddresRegister)
run_clock(UNTIL_EXPERIMENT_ENDS)
```

```
Address   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
0106a5d0 0c 01 65 22 40 b6 10 00 00 49 12 91 00 c8 48 90
0106a5e0 c8 b0 5b 20 a0 05 80 d2 64 50 6d 14 99 10 9b 69
0106a5f0 14 00 00 08 21 8a 24 14 c0 30 93 61 b2 28 26 0a
0106a600 00 5b 25 22 0b b4 90 68 06 69 24 c0 00 00 21 34
0106a610 81 92 c0 00 00 48 92 52 24 49 29 96 88 12 18 4d
```

…

```
Address   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
0106a5d0 0c 01 65 22 40 b6 10 00 00 49 12 91 00 c8 48 90
0106a5e0 c8 b0 5b 20 a0 05 80 d2 64 50 6d 14 99 10 9b 69
0106a5f0 14 00 00 08 31 8a 24 14 c0 30 93 61 b2 28 26 0a
0106a600 00 5b 25 22 0b b4 90 68 06 69 24 c0 00 00 21 34
0106a610 81 92 c0 00 00 48 92 52 24 49 29 96 88 12 18 4d
```



UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Institute ITACA Information and Communication Technologies

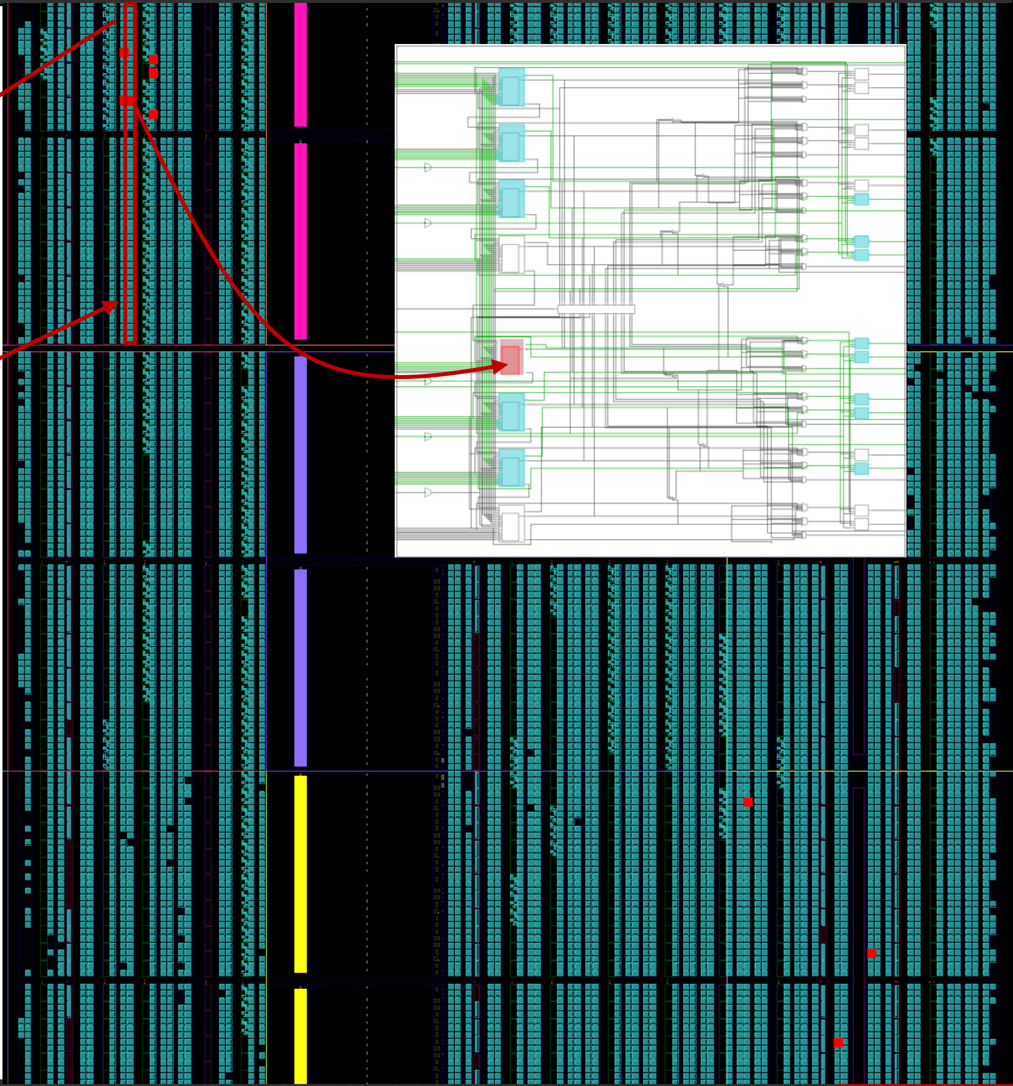
Fault injection in an AMD Zynq UltraScale+ device

```
# Injection controlled by the integrated ARM core
enable_clock_throttle()
run_clock(FOR_INJECTION_TIME_CYCLES)
frameAddressRegister = getFAR(targetLut)
frame = read_frame(frameAddressRegister) # 2976 bits
lutContent = get_lut_content(frame) # 64 bits
faultyLutContent = inject_fault(lutContent, bit) # invert the target bit
update_frame(faultyLutContent, frame)
write_frame(frame, frameAddresRegister)
run_clock(UNTIL_EXPERIMENT_ENDS)
```

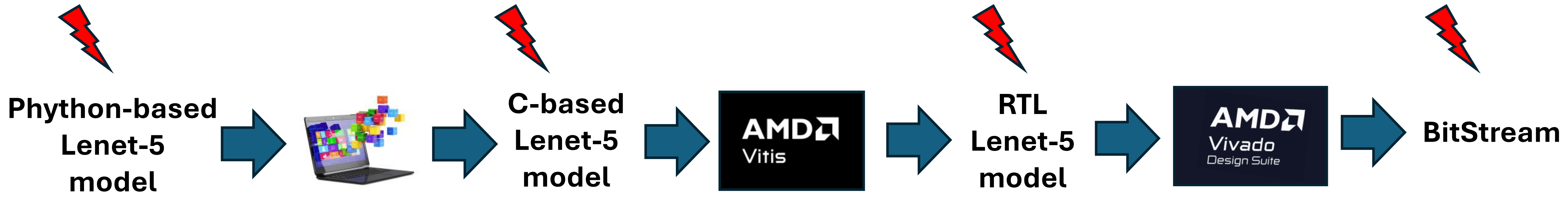
```
Address   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
0106a5d0  0c 01 65 22 40 b6 10 00 00 49 12 91 00 c8 48 90
0106a5e0  c8 b0 5b 20 a0 05 80 d2 64 50 6d 14 99 10 9b 69
0106a5f0  14 00 00 08 21 8a 24 14 c0 30 93 61 b2 28 26 0a
0106a600  00 5b 25 22 0b b4 90 68 06 69 24 c0 00 00 21 34
0106a610  81 92 c0 00 00 48 92 52 24 49 29 96 88 12 18 4d
```

…

```
Address   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
0106a5d0  0c 01 65 22 40 b6 10 00 00 49 12 91 00 c8 48 90
0106a5e0  c8 b0 5b 20 a0 05 80 d2 64 50 6d 14 99 10 9b 69
0106a5f0  14 00 00 08 31 8a 24 14 c0 30 93 61 b2 28 26 0a
0106a600  00 5b 25 22 0b b4 90 68 06 69 24 c0 00 00 21 34
0106a610  81 92 c0 00 00 48 92 52 24 49 29 96 88 12 18 4d
```



Wrap-up on Fault Injection



- The closer to the implementation, the higher the representativity?
  No, as far as injection into CNN weights is considered
- Flipping or stucking a bit in a CNN weight is very easy, doesn´t it?
  It is not very complex, but not as easy as it may seem at a first sight
  - Python-based fault injection can be privileged, but not easy for pytorch-based quantized CNNs. C-based fault injection mitigate such problems
  - RTL-based fault injection easier than injecting at the FPGA level, but slower. In FPGA, the challenge is to stablish a precise mapping between RTL components and the FPGA resources under use
- And what about injecting faults into CNN processing elements? Out from the scope of this talk

Robustness evalutation

- Goal:
  Estimate the impact of faults on the CNN accuracy
- Targets:
  CNN parameter bits
- Fault Injection Methodology
  - Which fault model? Multiple faults
  - Which fault injection process should be followed?
  - **How many faults to inject?**

How many faults to inject?

- Potential impact of faults on the CNN inference process
  - Accuracy (hit rate) in image classification rarely reaches 100%
  - The misclassification of certain images is normal (not a failure)

Original CNN	Injected CNN	Failure Mode
Hit/Miss	Hit/Miss	No failure
Miss	Hit	Unexpected Hit
Hit	Miss	Unexpected Miss

- Exhaustive fault injection is only feasible with toy CNNs → Proposal: use of statistical fault injection

Statistical Fault Injection

$$n = \frac{N}{1 + e^2 \times \frac{N-1}{t^2 \times p \times (1-p)}} \xrightarrow[\text{(more than 10000 individuals)}]{\text{assuming an infinite population}} n = \frac{z^2 p(1-p)}{e^2}$$

Sampling	Fault injection
Population (N)	{fault, location}
Sample size (n)	Fault injection experiments to be carried out
Characteristic (p)	probability for a population individual to have a characteristic (No failure, unexpected Hit or Unexpected Miss in our case). When no knowledge of the population under study is available, p should be 50% (0,5)
Margin of error (e)	Margin of error (typical values smaller than 5%)
Confidence level (z-score)	Confidence level (typical value 95% → z-score=1.96)

[Tuzov et al. 2018]* Ilya Tuzov et al. "**Accurate Robustness Assessment of HDL Models Through Iterative Statistical Fault Injection**", 14th European Dependable Computing Conference (EDCC 2018), DOI: 10.1109/EDCC.2018.00013

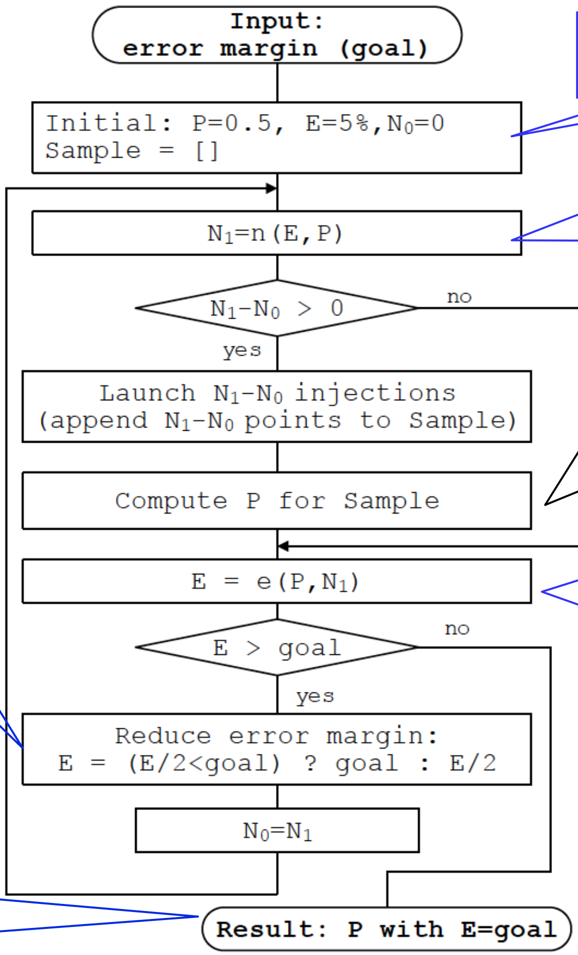
Interesting results

- With **only 384 experiments** per type of considered fault one can estimate failure modes of a system with a confidence level of 95% and an error margin of 5% !!!
- This error maybe too high if the percentage of cases when the considered failure mode occurs is very low

In all cases e≤ 0.1 Confidence level = 95%	Conservative sample	
	P	Size
Unexpected Miss	50%	784447
Unexpeted Hit	50%	784447

[Tuzov et al. 2018]* Ilya Tuzov et al. "**Accurate Robustness Assessment of HDL Models Through Iterative Statistical Fault Injection**", 14th European Dependable Computing Conference (EDCC 2018), DOI: 10.1109/EDCC.2018.00013

Improvement: Statistical Iterative Fault Injection



For Confidence level of 95% → Z = 1.96

$$N_1 = \frac{Z^2 P(1-P)}{E^2}$$

The actual rate (P) for the considered failure mode is computed attending to the results issued from the exper. carried out so far

$$E = \sqrt{\frac{Z^2 P(1-P)}{N_1}}$$

Each iteration divides the error margin by 2. Fast error convergence may imply the realization of many experiments.

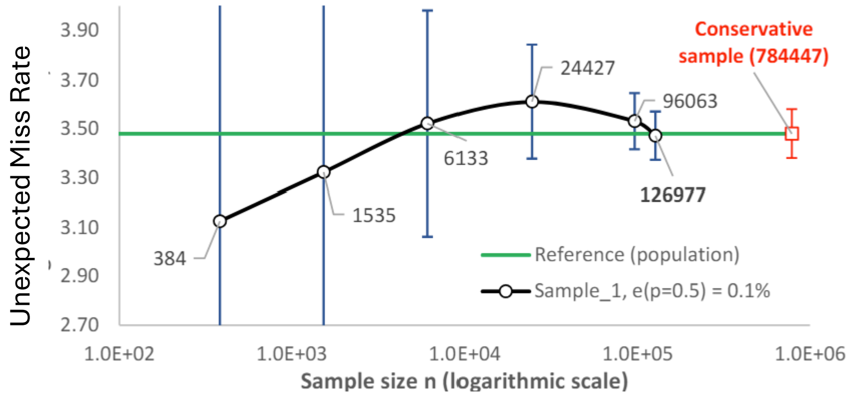
Final estimations can be always reported with their associated error margins

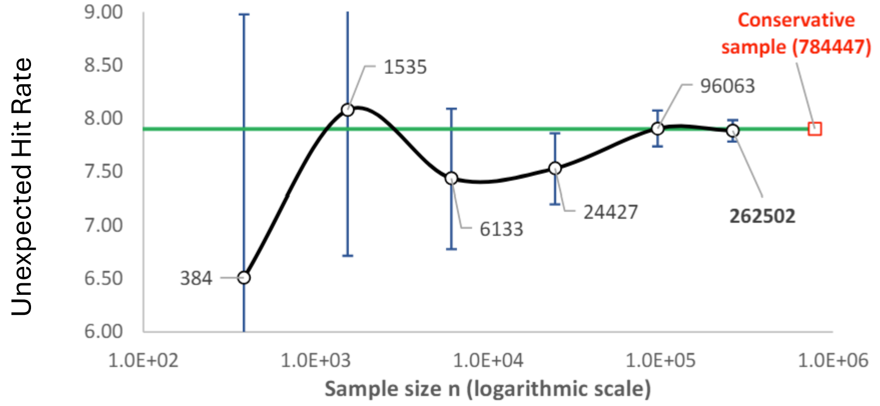
Driven by error margin

[Tuzov et al. 2018][*] Ilya Tuzov et al. "**Accurate Robustness Assessment of HDL Models Through Iterative Statistical Fault Injection**", 14th European Dependable Computing Conference (EDCC 2018), DOI: 10.1109/EDCC.2018.00013

Improvement: Statistical Iterative Fault Injection

- The approach in action for the example CNN





In all cases e≤ 0.1 Confidence level = 95%	Conservative sample		Required sample	
	P	Size	P	Size
Unexpected Miss	50%	784447	7.90%	262502
Unexpeted Hit	50%	784447	3.48%	126977

[Tuzov et al. 2018][*] Ilya Tuzov et al. "**Accurate Robustness Assessment of HDL Models Through Iterative Statistical Fault Injection**", 14th European Dependable Computing Conference (EDCC 2018), DOI: 10.1109/EDCC.2018.00013

Results for Lenet-5

- Failures (unexpected misses and hits) provoked by stuck-at-faults estimated with confidence Interval 95% and Error 0,1%:

Faults per injection	Fault model	FP32-based Lenet 5	INT8-based Lenet 5
Single faults	Stuck-at-0	0,0010%	0,0047%
	Stuck-at-1	1,6658%	0,0620%
Double adjacent faults	Stuck-at-0	0,0013%	0,0062%
	Stuck-at-0	3,4391%	0,0715%
Triple adjacent faults	Stuck-at-0	0,0013%	0,0079%
	Stuck-at-1	3,5630%	0,0835%

$>$

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Institute ITACA Information and Communication Technologies

Results for Lenet-5

- Failures (unexpected misses and hits) per layer provoked by stuck-at-1 faults Confidence Interval 95%, Error 0,1%, :

Faults	Type of layer	FP32-Lenet5	INT8-lenet5
Simple faults	Convolution	2,1675%	3,8782%
	Fully connected	1,6632%	0,0365%
Double adjacent faults	Convolution	3,5688%	4,4326%
	Fully connected	3,4385%	0,0417%
Triple adjacent faults	Convolution	4,1134%	5,0245%
	Fully connected	3,5601%	0,0488%

<

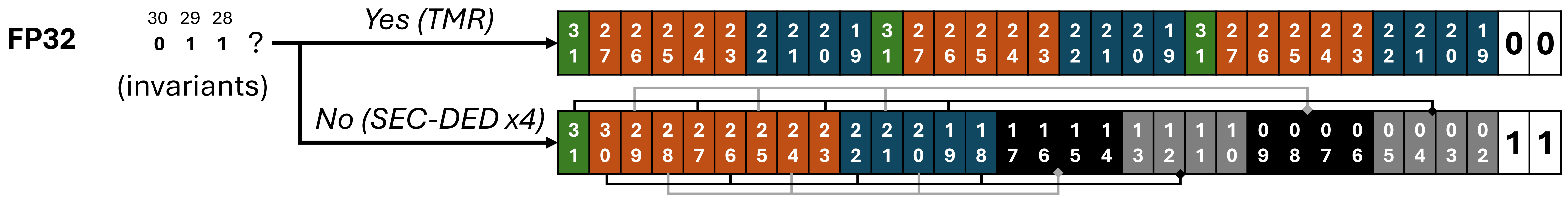
Outline

- ❑ Understanding HW accelerators for CNNs: Prototyping a FP32 /INT8 CNN on a FPGA: Lenet-5 as a case study
- ❑ Robustness evaluation of CNNs using fault injection: methodology and lessons learnt
- ❑ **In-Memory Zero-Space Protection of FP-based CNNs using ECCs: methodology and lessons learnt**
- ❑ Conclusions

Existing approaches

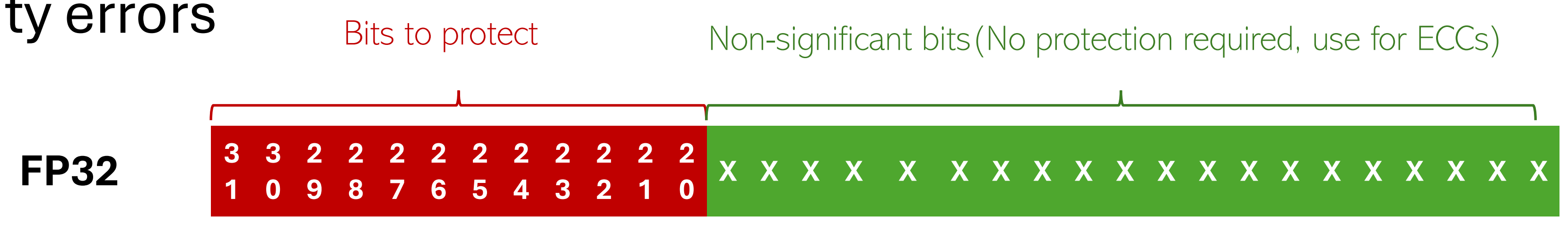
- CNN retraining required
  - Normalize weights to compensate weight criticity
  - Fault injection during training to learn fault tolerance Retrain the CNN to
  - Retrain the CNN to ensure a certain weight bit distribution including no significant bits that can be used for ECC deployment → useful for quantized CNNs (SEC-DED max)
- No CNN retraining required

  - MATE: Memory And retraining-free Error correction for CNN weights



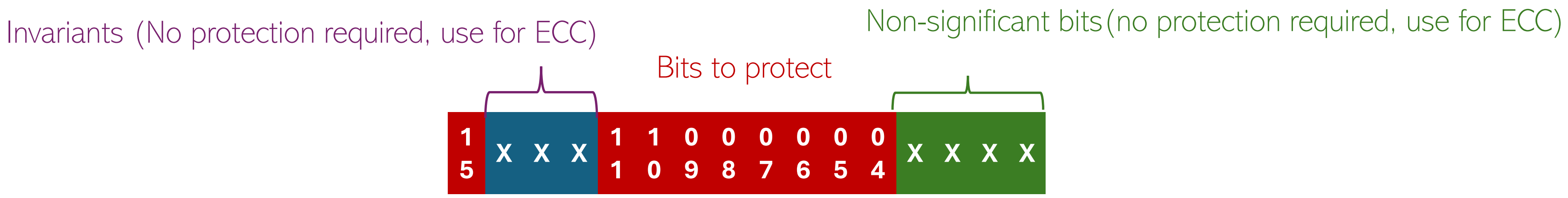
Parameter protection without retraining

- **[EDCC 2024]**[1] Idenfication of non-significant bits + use of those bits to hold ECC parity errors


Bits to protect
Non-significant bits (No protection required, use for ECCs)
FP32
31 30 29 28 27 26 25 24 23 22 21 20
X X X X X X X X X X X X X X X X X X X X

* Note: The concrete división between red and green bits will vary from one CNN to another

- **[SAFECOMP 2024]**[2] Use of non-significant and invariant bits for BF16 CNN protection with ECCs


Invariants (No protection required, use for ECC)
Non-significant bits (no protection required, use for ECC)
Bits to protect
15
X X X
11 10 9 8 7 6 5 4
X X X X

* Note: The concrete división between blue, red and green bits will vary from one CNN to another

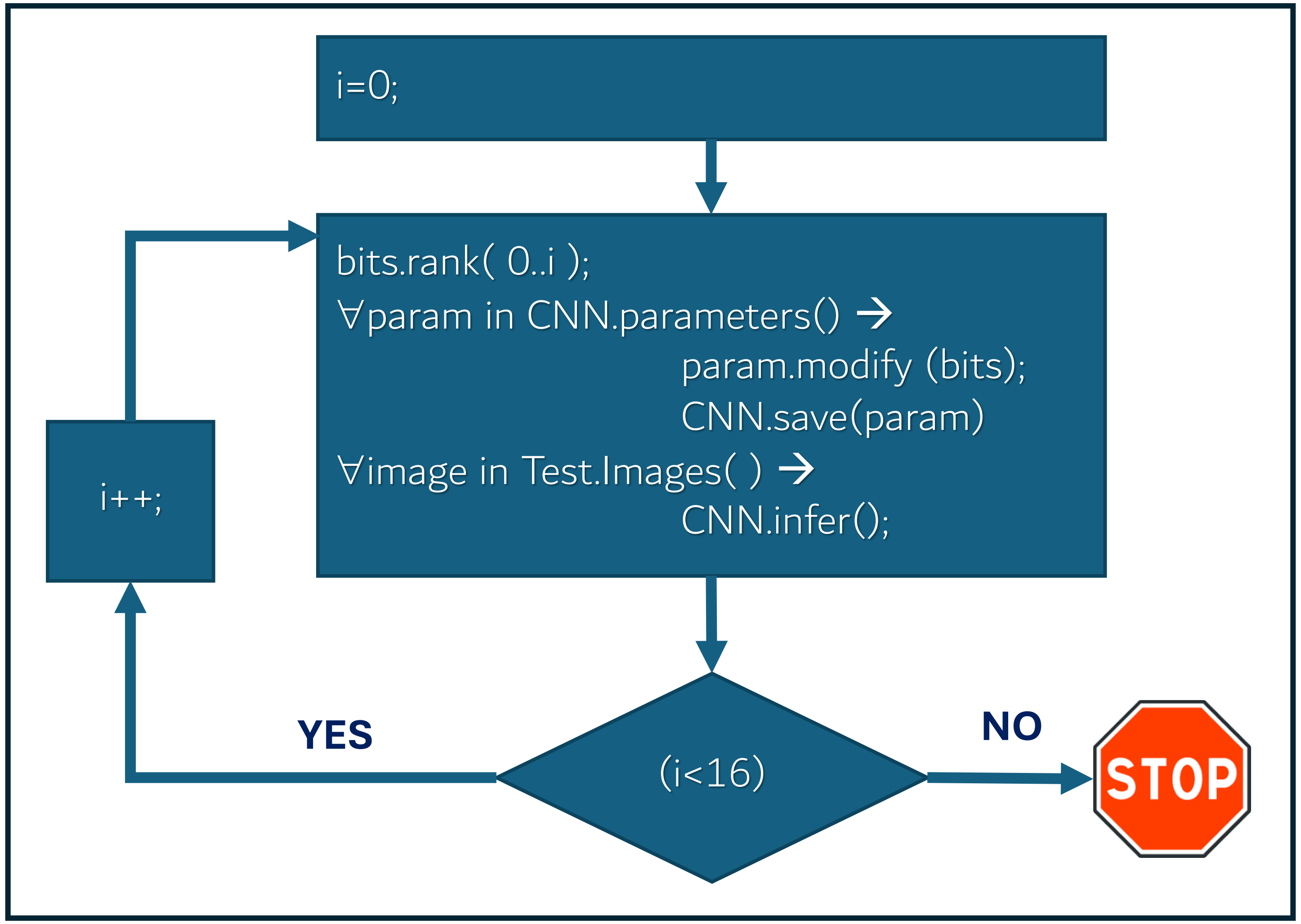
[1] **[EDCC 2024]** Juan Carlos Ruiz, David de Andrés, Luis J. Saiz-Adalid, Joaquin Gracia-Moran: Zero-Space In-Weight and In-Bias Protection for Floating-Point-based CNNs. EDCC 2024: 89-96, Lovaina (Bélgica), Abril 2024.
[2] **[SAFECOMP 2024]** Juan Carlos Ruiz, David de Andrés, Luis J. Saiz-Adalid, Joaquin Gracia-Moran: In-Memory Zero-Space Floating-Point-based CNN Protection Using Non-Signicant and Invariant Bits, SAFECOMP 2024, Florencia (Italia), Septiembre 2024.

Location of non-significant bits

Why is this location necessary?

Non-significant bits do not require protection → use them to hold parity ECC bits

How to locate such non-significant bits?

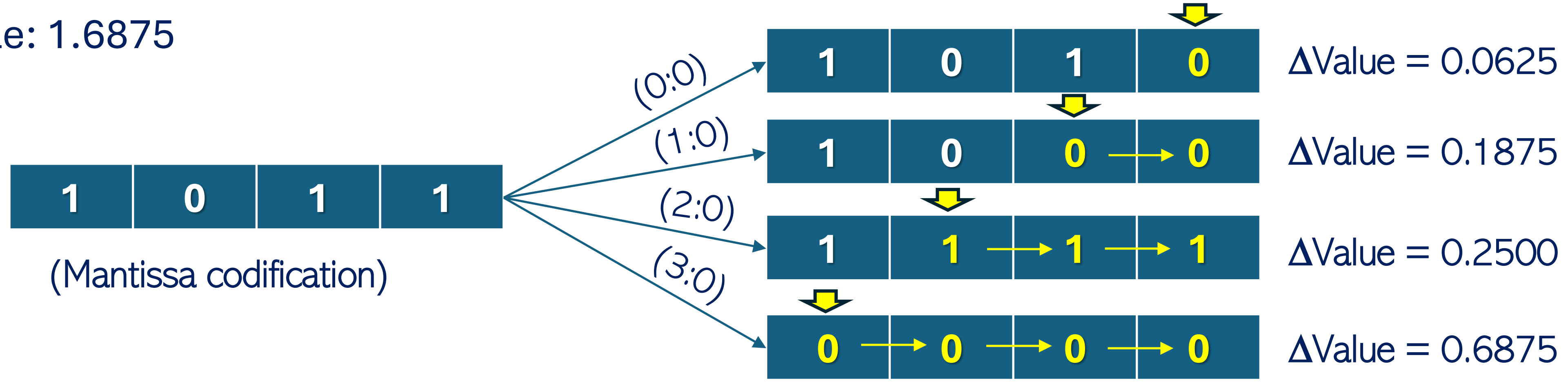


```
i=0;

bits.rank( 0..i );
∀param in CNN.parameters() →
                        param.modify (bits);
                        CNN.save(param)
∀image in Test.Images( ) →
                        CNN.infer();

(i<16)
YES → i++;
NO → STOP
```

Target parameter modification

- Goal is to maximize the difference between the original and the injected value
  - If a ‘0’/‘1’ is injected many bits will remain unaltered
  - If a bitflip is injected the effect can be very small (011 → 100: ΔValue = 1!!)
- **Mixed injection process**
  1. Flip the most signicant bit (msb) in the considered rank of bits
  2. The rest of bits in the rank adopt the value of the msb

Example: 1.6875



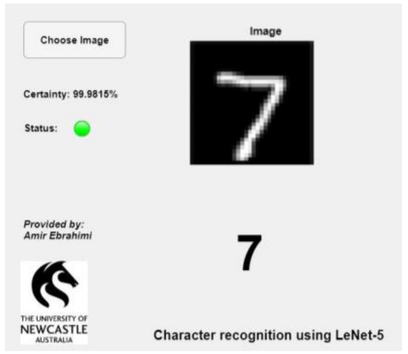
Invariant bit identification

- **Invariant bits** keep the same value in all CNN parameters
- Will you always find invariants in weights?
  - Parameter exponents are rarely bigger tan 0
    - BF16 exponent are encoded using excess 127: 0 → 011111111
    - Small values will be 011… this is why invariants are likely to exist

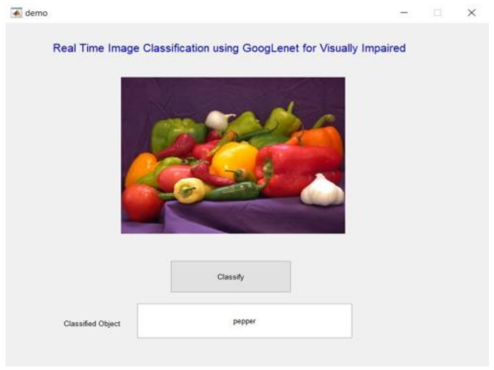
- And what about those bits that are "nearly" invariants?
  - It is possible to set them as **forced invariants**
  - Study the impact of this decition on the CNN accuracy

Case studies

- Lenet-5 (BF16 version): Identification of manuscript numbers (10 categories)
  - Depth of 2 layers
  - Parameters: 45539 (weights + bias)
  - Dataset: MNIST (10.000 monochrome test images of 28x28 pixels)
  - Accuracy: 98,23% (117 incorrect matches out of 10.000 test images)


Choose Image
Image
Certainty: 99.9815%
Status:
Provided by:
Amir Ebrahimi
7
THE UNIVERSITY OF NEWCASTLE AUSTRALIA
Character recognition using LeNet-5

- Googlenet (BF16 version): Object identification (up to 1000 categories)
  - Depth of 22 layers
  - Parameters : 6624904 (weights + bias)
  - Dataset: ImageNet (50.000 objet and animal RGB test images of 256x256 pixels)
  - Accuracy: 69,772% (15.114 incorrect matches out of 50.000 images)


demo
Real Time Image Classification using GoogLenet for Visually Impaired
Classify
Classified Object
pepper

Non-significant bits

- Their simultaneous modification in all the CNN parameter will not significantly affect the network accuracy
- Maximum allowed variation of **1 porcentual point** over the original network accuracy
  - Lenet-5: $\text{Accuracy}_{\text{original}}$=98,23%
  - GoogleNet: $\text{Accuracy}_{\text{original}}$=69,772%

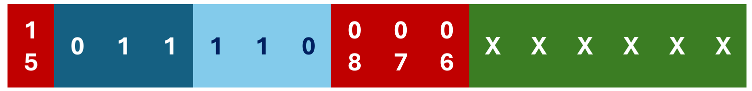
“nearly” significant bits

	Lenet-5		GoogLeNet	
Bits	**Accuracy**	**Difference (in p.p.)**	**Accuracy**	**Difference (in p.p.)**
(0:0)	98,21%	0,02	69,826%	-0,054
(1:0)	98,24%	-0,01	69,290%	0,482
(2:0)	98,17%	0,06	68,908%	0,864
(3:0)	98,09%	0,14	64,974%	4,798
(4:0)	97,88%	0,35	50,952%	18,82
(5:0)	98,03%	0,2	1,800%	67,972
(6:0)	88,20%	10,03	0,136%	69,636
(7:0)	66,94%	31,29	0,088%	69,684
(8:0)	64,16%	34,07	0,106%	69,666
(9:0)	23,63%	74,6	0,108%	69,664
(10:0)	8,17%	90,06	0,104%	69,668
(11:0)	9,80%	88,43	0,100%	69,672
(12:0)	9,80%	88,43	0,100%	69,672
(13:0)	9,80%	88,43	0,100%	69,672
(14:0)	9,80%	88,43	0,100%	69,672
(15:0)	9,80%	88,43	0,100%	69,672

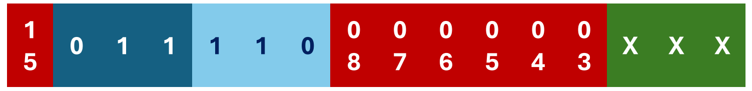
Invariant bits

	LeNet-5		GoogLeNet	
Bit	0	1	0	0
0	50,02%	49,98%	50,11%	49,89%
1	50,20%	49,80%	50,30%	49,70%
2	50,54%	49,46%	50,54%	49,46%
3	51,56%	48,44%	51,13%	48,87%
4	52,12%	47,88%	52,27%	47,73%
5	54,38%	45,62%	54,41%	45,59%
6	58,63%	41,37%	58,50%	41,50%
7	48,16%	51,84%	50,14%	49,86%
8	35,91%	64,09%	49,75%	50,25%
9	84,49%	15,51%	79,38%	20,62%
10	9,49%	90,51%	20,31%	79,69%
11	0,02%	99,98%	0,08%	99,92%
12	0,00%	100,00%	0,00%	100,00%
13	0,00%	100,00%	0,00%	100,00%
14	100,00%	0,00%	100,00%	0,00%
15	50,20%	49,80%	45,19%	54,81%

At a first sight



LeNet-5 (12 available bits)

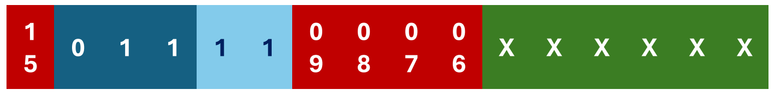


GoogLeNet (9 available bits)

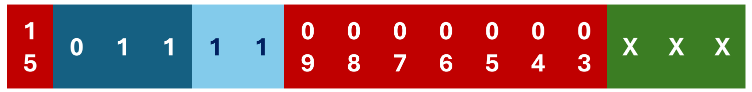
Impact of considering the 3 non-invariant bits as invariants

		Lenet-5		GoogLenet	
Bits	Invariants	Accuracy	Difference (p.p.)	Accuracy	Difference (p.p.)
(14:11)	0111	98,22%	0,01	69,85%	0,07
(14:10)	01111	98,20%	0,02	69,62%	0,15
(14:9)	011110	95,84%	2,39	0,10%	69,67

Finalmente



LeNet-5 (11 available bits)



GoogLeNet (8 available bits)

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Institute ITACA Information and Communication Technologies

Considered ECCs

Invariants vs Non-significant bits

		S	Exponent								Mantissa						
ECC	**Fault coverage**	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
(16, 10)+3	SEC-DAEC	🟥	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟩	🟩
(16, 10)+4	SEC-DAEC	🟥	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟩
(16, 10)+5	SEC-DAEC	🟥	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩

🟥 Protected bit	🟩 Non-significant bit holding a parity bit
🟦 Invariant bit holding a parity bit	⬜ Unprotected non-significant bit

ECCs considerados

Tuning the number of protected bits

		S	Exponent								Mantissa						
ECC	**Fault coverage**	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
(15, 9)+4	SEC-DAEC	🟥	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	⬜	🟩	🟩
(14, 8)+5	SEC-DAEC	🟥	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟥	⬜	⬜	🟩

🟥	**Protected bit**	🟩	**Non-significant bit holding a parity bit**
🟦	**Invariant bit holding a parity bit**	⬜	**Unprotected non-significant bit**

ECCs considerados

Use of more bits than required to deploy the considered ECCs

		S	Exponent								Mantissa						
ECC	**Fault coverage**	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
(16, 9)+4	SEC-DAEC	🟥	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟩	🟩
(16, 9)+5	SEC-DAEC	🟥	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟩
(16, 8)+5	SEC-DAEC	🟥	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟩	🟩

🟥	**Protected bit**	🟩	**Non-significant bit holding a parity bit**
🟦	**Invariant bit holding a parity bit**	⬜	**Unprotected non-significant bit**

ECCs considerados

			S	Exponent								Mantissa						
	ECC	Fault coverage	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Deployment of higher levels of protection	(16, 10)+5	SEC-DAEC-TAEC	🟥	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩
	(16, 9)+5	SEC-DAEC-3bBEC-4AEC	🟥	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟩
	(16, 8)+5	SEC-DEC-TAEC-4AEC	🟥	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟩	🟩

🟥	Protected bit	🟩	Non-significant bit holding a parity bit
🟦	Invariant bit holding a parity bit	⬜	Unprotected non-significant bit

ECC impact on CNN accuracy

		Lenet-5		GoogLeNet	
ECC	Fault coverage	Accuracy	Difference (en p.p)	Accuracy	Difference (en p.p)
None	None	98,23%	--	69,77%	--
(16, 10)+3	SEC-DAEC	98,20%	0,03	69,39%	0,38
(16, 10)+4	SEC-DAEC	98,24%	-0,01	69,67%	0,10
(16, 10)+5	SEC-DAEC	98,25%	-0,02	69,59%	0,18
(15, 9)+4	SEC-DAEC	98,20%	0,03	69,39%	0,38
(14, 8)+5	SEC-DAEC	98,21%	0,02	69,13%	0,64
(16, 9)+4	SEC-DAEC	98,20%	0,03	69,39%	0,38
(16, 9)+5	SEC-DAEC	98,27%	-0,04	69,49%	0,28
(16, 8)+5	SEC-DAEC	98,21%	0,02	69,13%	0,64
(16, 10)+5	SEC-DAEC-TAEC	98,24%	-0,01	69,67%	0,10
(16, 9)+5	SEC-DAEC-3bBEC-4AEC	98,27%	-0,04	69,49%	0,28
(16, 8)+5	SEC-DEC-TAEC-4AEC	98,21%	0,02	69,13%	0,64

Impact of combining forced invariants and "nearly" significant bits

Deployment of ECC decoders on Lenet-5

- Decoders implemented in C for HLS
- Deployment on a FPGA AMD Zynq UltraScale+ XCZU7EV-2FFVC1156 MPSoC

Better using invariant than non-significant bits

Leave unprotected those bits that do not matter

Possitive effect of using more bits than necessary for ECC deployment

CNN	Fault coverage	LUT	FF	BRAM	DSP	Decos	Deco size (LUTs)	Latency (clock cyclesj)	Power (mW)	Energy (mW/imagen)
LeNet	None	85655	107478	156	500	0	0	7443	3,34	0,231
(16, 10)+3	SEC-DAEC	13,02%	5,07%	-78,53%	0,00%	237	269	0,04%	-11,47%	-10,82%
(16, 10)+4	SEC-DAEC	15,15%	5,93%	-78,53%	0,00%	237	240	0,04%	-2,13%	0,97%
(16, 10)+5	SEC-DAEC	13,60%	5,99%	-78,53%	0,00%	237	256	0,04%	-4,94%	-6,16%
(15, 9)+4	SEC-DAEC	13,60%	4,76%	-78,53%	0,00%	237	263	0,04%	-3,47%	-0,37%
(14, 8)+5	SEC-DAEC	9,43%	3,38%	-78,53%	0,00%	237	181	0,04%	-6,32%	-10,57%
(16, 9)+4	SEC-DAEC	11,58%	4,20%	-78,53%	0,00%	237	241	0,04%	-3,38%	-8,36%
(16, 9)+5	SEC-DAEC	10,69%	4,45%	-78,53%	0,00%	237	193	0,04%	-2,69%	-6,10%
(16, 8)+5	SEC-DAEC	7,08%	4,36%	-78,53%	0,00%	237	155	0,04%	-3,41%	-19,87%
(16, 10)+5	SEC-DAEC-TAEC	15,14%	5,88%	-78,53%	0,00%	237	535	0,04%	1,35%	-3,22%
(16, 9)+5	SEC-DAEC-3bBEC-4AEC	13,25%	5,48%	-78,53%	0,00%	237	725	0,04%	-12,40%	-20,53%
(16, 8)+5	SEC-DEC-TAEC-4AEC	19,93%	4,84%	-78,53%	0,00%	237	638	0,04%	6,44%	-2,90%

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Institute ITACA Information and Communication Technologies

Outline

- Understanding HW accelerators for CNN:
  Prototyping a FP32 /INT8 CNN on a FPGA: Lenet-5 as a case study
- Robustness evaluation of FP-based CNNs using fault injection: methodology and lessons learnt
- In-Memory Zero-Space Protection of FP-based CNNs using ECCs: methodology and lessons learnt
- **Conclusions**

Conclusions

- HW acceleration is a must for the use of CNNs in CPS
- Without protection, the accuracy of these accelerators may be drastically altered
- Dependability assessment can be carried out in a consistent way at high-levels of abstraction as FI inparameters is considered
- Statistical fault injection is a MUST for keeping result representativity
- CNN parameters can be protected using ECCs without requiring further memory and with a negligible overhead for FP32 and BF16 HW accelerators, but the approach losses most of its benefits for INT8-based CNNs

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Institute ITACA
Information and Communication Technologies

On Improving the Robustness Of Convolutional Neural Networks Using In-Parameter Zero-Space Error Correction Codes

Juan-Carlos Ruiz-García

ITACA-UPV (**Spain**)
jcruizg@disca.upv.es

2ND WORKSHOP VERDI @ DSN2024
24th JUNE 2024, BRISBANE (AUSTRALIA)