IMMS

Institut für Mikroelektronik- und Mechatronik-Systeme gemeinnützige GmbH

FPGA-based Sensor Signal Processing

Georg Gläser

georg.glaeser@imms.de

We connect the digital to the analog world

We transfer results from basic research into applications

We …

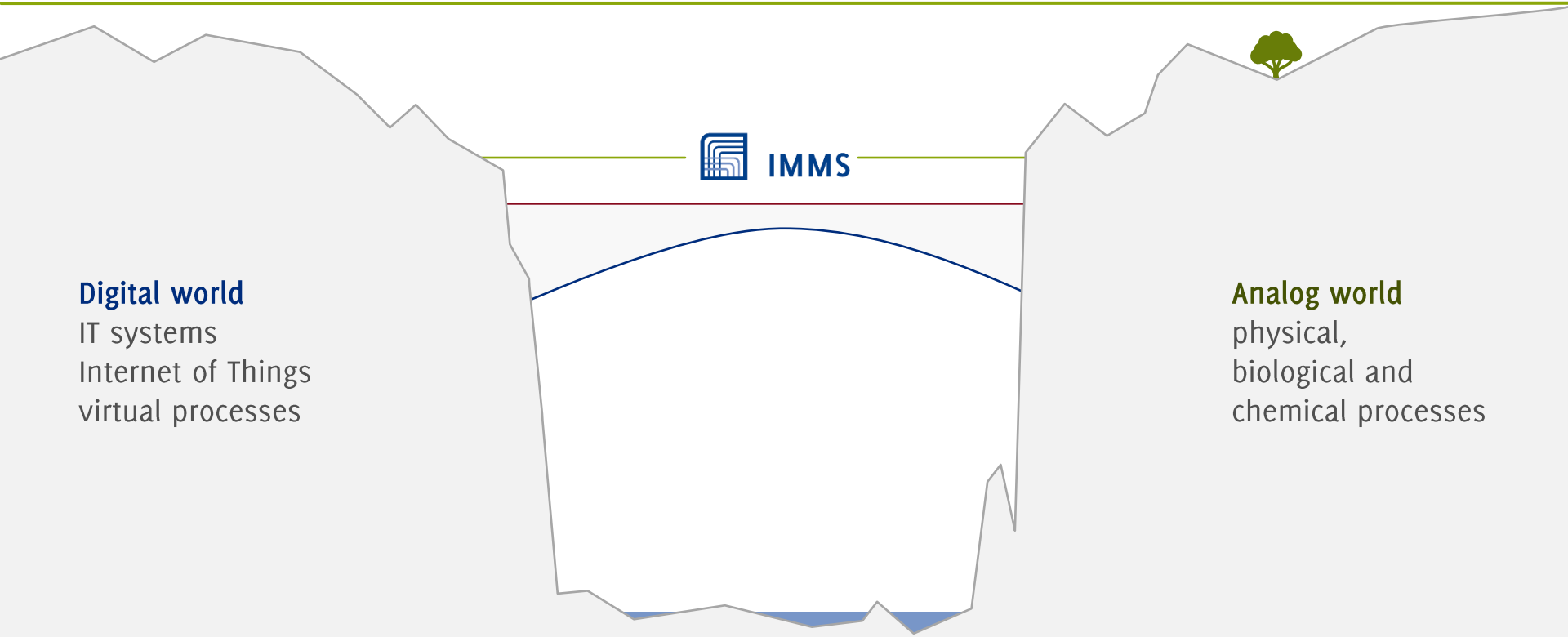
- are a land-owned **research institute** of the German land of **Thüringen** and an affiliated institute and **transfer partner** of Ilmenau TU,
- strengthen **SMEs** in particular,
- have been offering application-oriented R&D in **microelectronics, systems engineering and mechatronics since 1995,**
- support companies in launching internationally successful innovations for **health**, the **environment** and **industry** and provide solutions from the **feasibility study** to **series production.**

Sites in Thüringen, Germany

Institute headquarters
next to Technische Universität **Ilmenau**

Branch office in **Erfurt**
in the centre of Thüringen's microelectronics industry

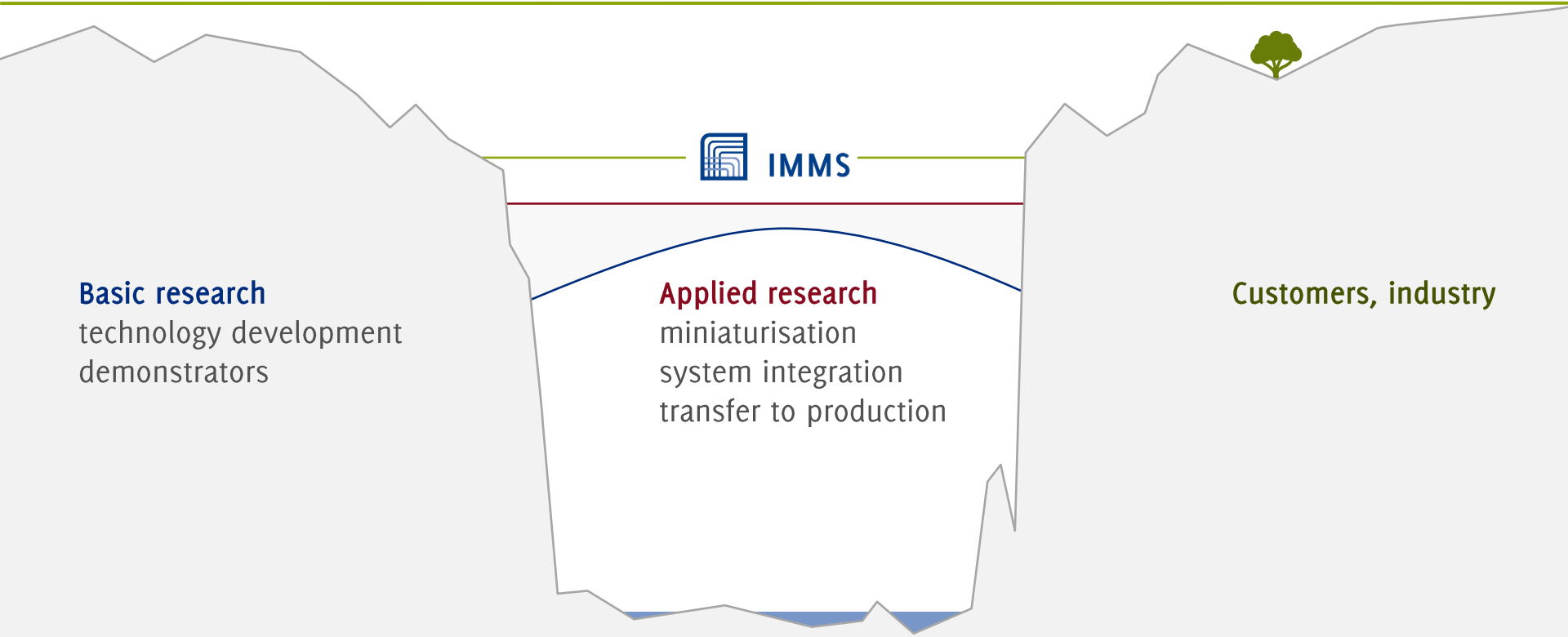
We connect the digital to the analog world



Digital world
IT systems
Internet of Things
virtual processes

Analog world
physical,
biological and
chemical processes

We bring research results into application: IMMS as R&D and transfer partner



We bring research results into application: IMMS as R&D and transfer partner

RESEARCH FIELDS

- Integrated **sensor systems**
- Smart distributed **measurement and test systems**
- Magnetic **6D direct drives** with nm precision

LEAD APPLICATIONS

- **Sensor systems for in-vitro diagnostics**
- **RFID** sensor technology
- Adaptive **edge AI** systems for industrial application
- **IoT** systems for cooperative environmental monitoring
- **nm measurement and structuring of objects**

TARGET MARKETS

- **Life sciences**
- **Automation technology and Industry 4.0**
- **Environmental monitoring and smart city applications**
- **Research institutions and ultra-precision mechanical engineering**

IMMS in figures (*31/12/2022)

Employees*
(especially engineers)

PhD candidates*

Students p.a.

Nations*

Patents*
(conferring priority
of which 12 granted)

Million € third-party funding*
for application-oriented research

Research projects*
(of which 57 industry projects)

Microelectronics department

- services / specialists for:
  - analogue IC design, digital IC design
  - mixed-signal verification
  - IC layout
  - design methodology, EDA support
- degrees in:
  - electrical engineering, IT,
  - microelectronics, engineering informatics,
  - biomedical engineering, solid-state physics
- approx. 10 ASICs per year for R&D
- approx. 15 students

IMMS

Institut für Mikroelektronik- und Mechatronik-Systeme gemeinnützige GmbH

FPGA-based Sensor Signal Processing

Georg Gläser

IMMS

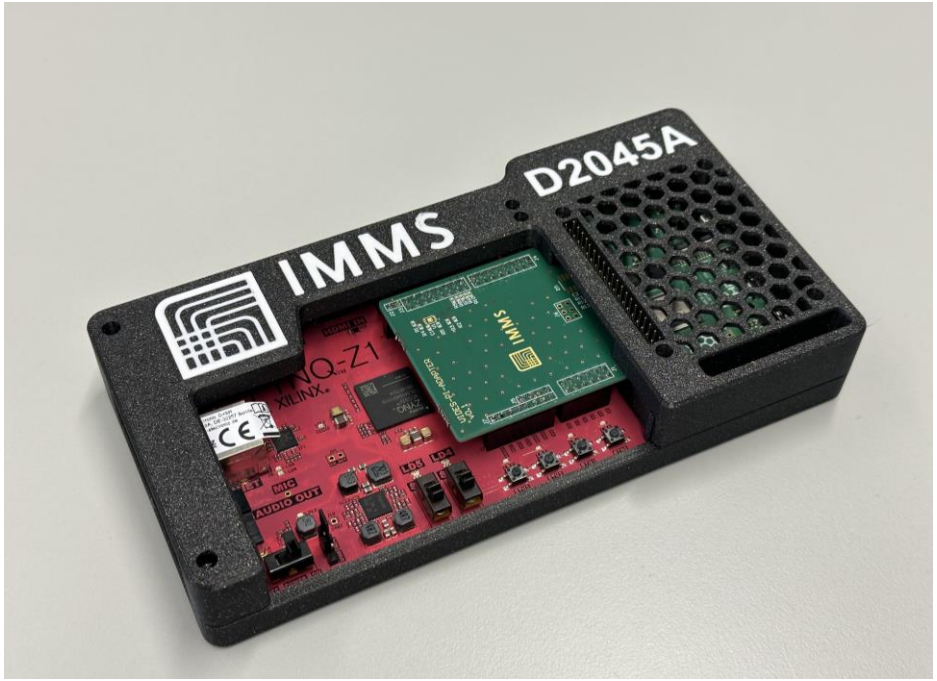
Institut für Mikroelektronik- und Mechatronik-Systeme gemeinnützige GmbH

Part 1: Introduction & Basics

Georg Gläser

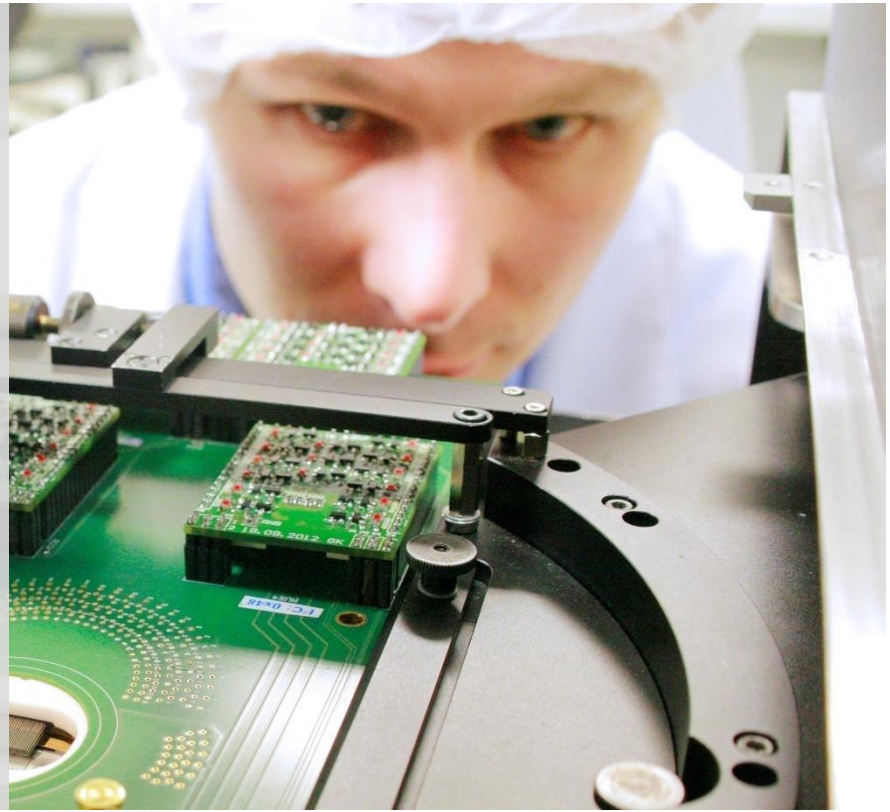
What this lecture is (not) about

- We will have a look at
  - What is an FPGA?
  - Structures for implementation
  - Verification
  - Best practice for Verilog Design
  - Common Pitfalls
- The following items are excluded
  - How to write basic verilog?
  - HLS (High-Level Synthesis)
  - Synthesis for FPGA or ASIC



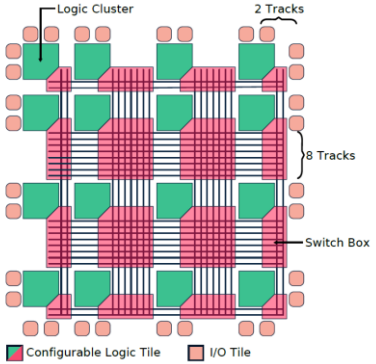
Short overview...

- Who played with Arduinos?
- Computer scientists?
- Electronic engineers?
- Who has experience with VHDL / Verilog?



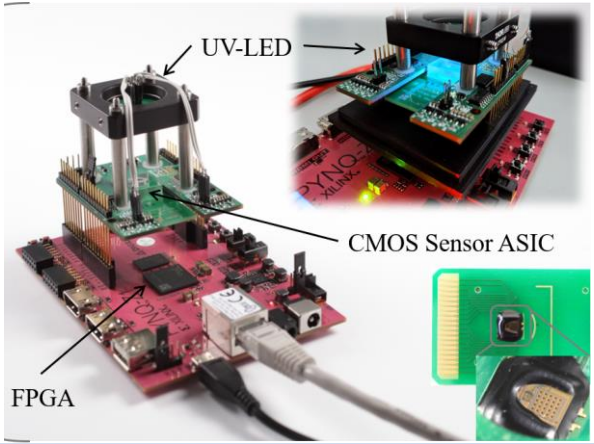
What is an FPGA?

- FPGA = Field Programmable Gate Array
- A large set of logic structures that can be programmed to
  - Realize an arbitrary logic function
  - Realize control algorithms
  - Model a processor



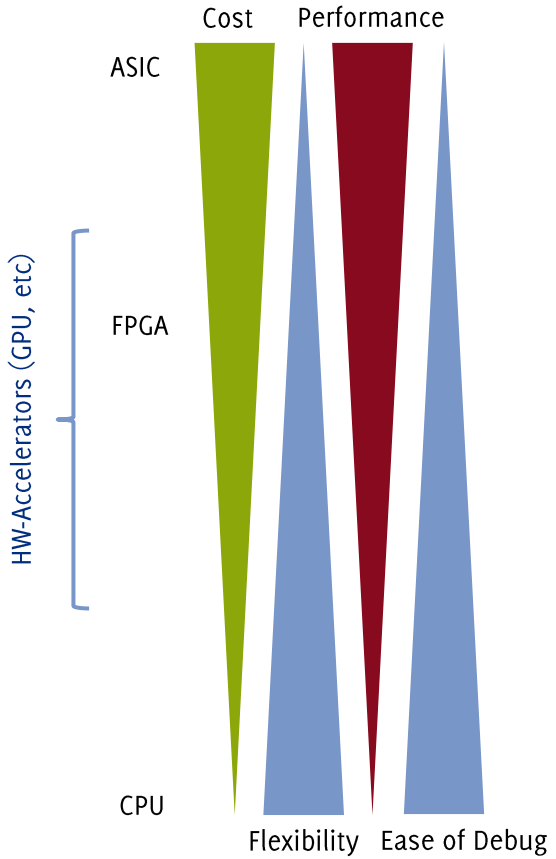
- How is it programmed?
  - Hardware description languages (Verilog / VHDL)
  - Development with vendor softare (Xilinx/AMD Vivado, Altera/Intel Quartus)

- Combination with On-Chip CPU cores possible

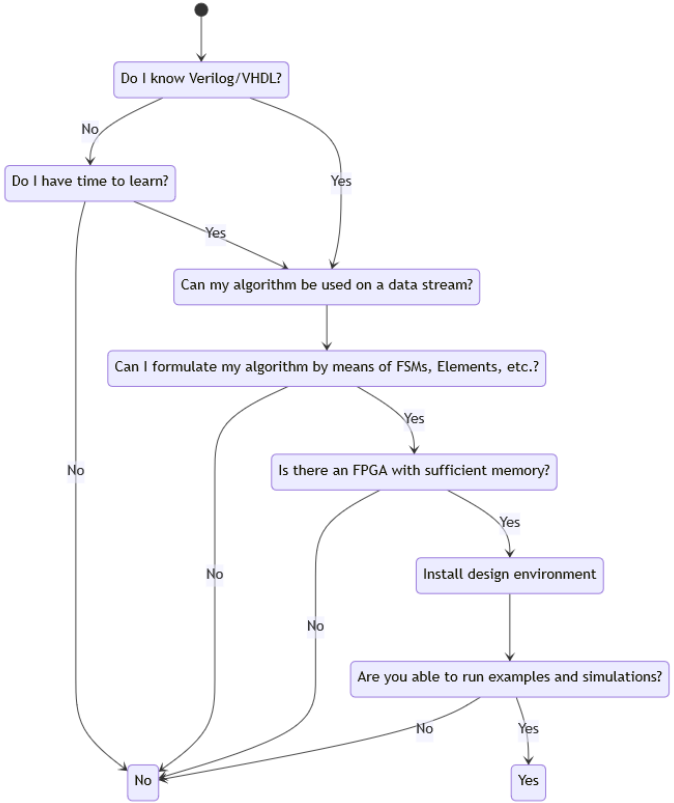


Why and when is it useful?

- Pros
  - Inherently parallel processing
  - Very fast calculations without van-Neumann Bottleneck
  - Real-Time / Constant time processing
  - Fine-Grained control of micro architecture
- Cons
  - Less available memory
  - Large design effort
  - TAT may be an issue



Should I go for an FPGA?



Where to learn good Verilog/VHDL?

- Just some links (I am not affiliated with any of them, so this represents just my personal collection)
  - https://www.fpga4fun.com/
  - “Advanced Chip Design, Practical Examples in Verilog” by Kishore Mishra (just read the first half)
  - http://www.asic-world.com/verilog/veritut.html
  - https://www.sutherland-hdl.com/papers.html

- About Architecture, Design, etc
  - „ISA System Architecture“ by Tom Shanley (old fashioned but really good, also the follow-ups are worth a read)
  - „Clean Code“ and „Clean Architecture“ by Robert Martin (not focused on hardware, but very instructive)

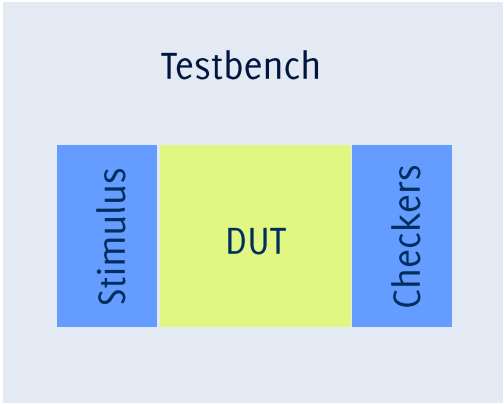
- And of cause… the documentation of your FPGA design tool

Interesting Literature

- If you have trouble with timing constraints
  https://cdrdv2-public.intel.com/653688/an433.pdf
- Verification with Python?
  https://www.cocotb.org/
- OpenSource Verilog Simulators
  https://github.com/steveicarus/iverilog
  https://www.veripool.org/verilator/
- A very clean MPS430 implementation (with good test environment)
  https://github.com/olgirard/openmsp430

Some terminology

- Module:
  An entity in Hardware
- DUT:
  Device Under Test
- Testbench:
  A module that exercises a DUT in simulation with a stimulus and checking it‘s results
- Clock:
  (Globally) available reference signal



FPGA Work Flow

Specification

- What the FPGA should do
- How the interfaces look like
- Which internal parts are needed
- Target FPGA chip

HDL coding

- Create you hardware components
- Integrate your components to a (working) system
- Define constraints

Verification

- Prove that your code actualy solves your problem
- Validate your asumptions
- Simulate your system!

Physical implementation (automated)

- Synthesis
- Place&Route
- Bitstream generation

Programming & Test in real world environment

An example algorithm

- A python class that just averages measurements
- Each measurement comprises N samples
- How do we get this thing into an FPGA?

```python
import numpy as np


class Averager():
    def __init__(self, Nsamples):
        self.mem = np.zeros(Nsamples)
        self.meas_counter = 0

    def add_measurement(self, measure):
        self.meas_counter = self.meas_counter + 1
        self.mem = self.mem + measure

    def get_result(self):
        return (self.mem / self.meas_counter)

```

An example algorithm

- How do we get this thing into an FPGA?

```python
import numpy as np


class Averager():
    def __init__(self, Nsamples):
        self.mem = np.zeros(Nsamples)
        self.meas_counter = 0

    def add_measurement(self, measure):
        self.meas_counter = self.meas_counter + 1
        self.mem = self.mem + measure

    def get_result(self):
        return (self.mem / self.meas_counter)

```

- What we need:
  - Interfaces
  - Memory
  - Processing elements

- Since we are creating hardware:
  - We cannot have infinite memory
  - We cannot dynamically increase memory size
  - Nsamples must be fixed (or at least limited to some maximum)

An example algorithm

- How do we get this thing into an FPGA?

- What we need:
  - Interfaces
  - Memory
  - Processing elements

- Since we are creating hardware:
  - We cannot have infinite memory
  - We cannot dynamically increase memory size
  - Nsamples must be fixed (or at least limited to some maximum)

```python
import numpy as np


class Averager():
    def __init__(self, Nsamples):
        self.mem = np.zeros(Nsamples)
        self.meas_counter = 0

    def add_measurement(self, measure):
        self.meas_counter = self.meas_counter + 1
        self.mem = self.mem + measure

    def get_result(self):
        return (self.mem / self.meas_counter)

```

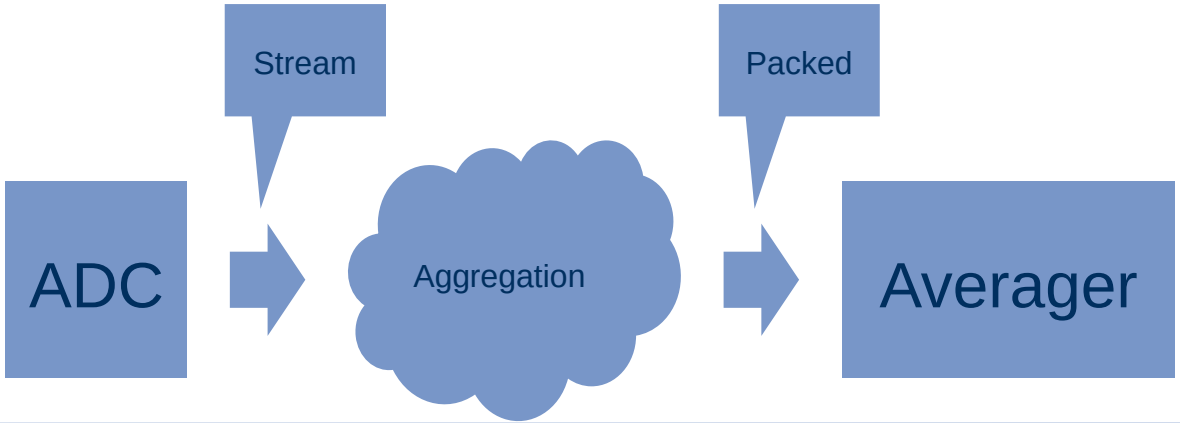
An example algorithm

- Since we are creating hardware:
  - We cannot have infinite memory
  - We cannot dynamically increase memory size
  - Nsamples must be fixed (or at least limited to some maximum)

```python
import numpy as np


class Averager():

    max_samples = 128

    def __init__(self, Nsamples):
        self.mem = np.zeros(self.max_samples)
        self.Nsamples = Nsamples
        self.meas_counter = 0

    def add_measurement(self, measure):
        self.meas_counter = self.meas_counter + 1
        self.mem[0:self.Nsamples-1] = self.mem[0:self.Nsamples-1] + measure

    def get_result(self):
        return (self.mem[0:self.Nsamples-1] / self.meas_counter)

```

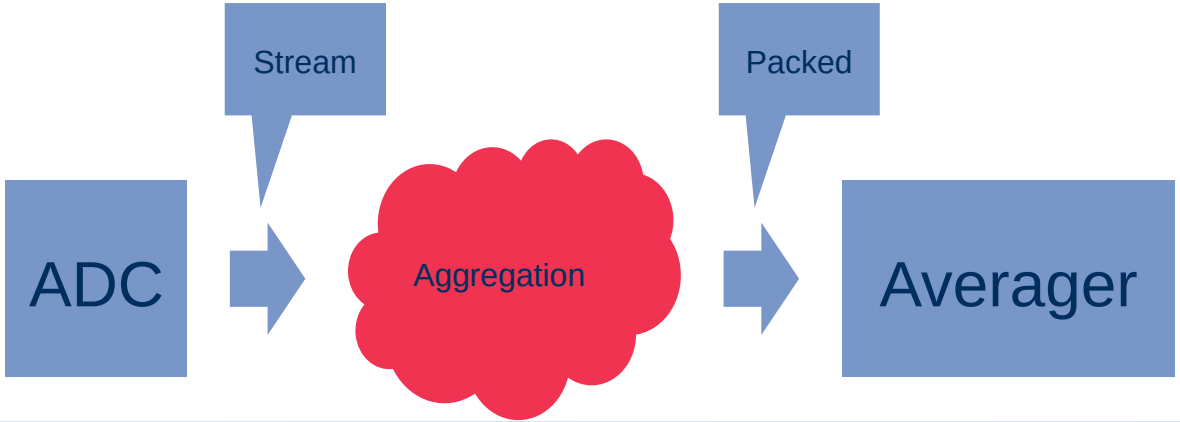
A system level perspective

- How could a system look like that works like this?



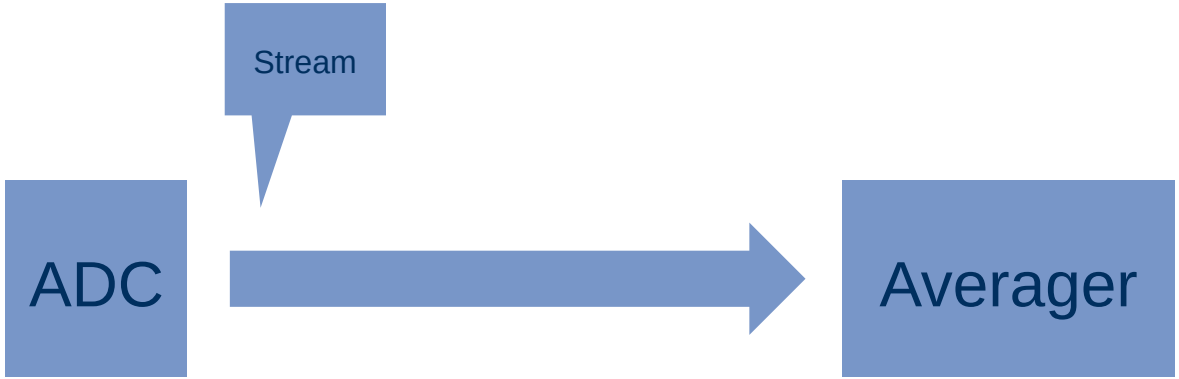
A system level perspective

- Memory-Estimation:
  - We need to aggregate N samples and pass them to the Averager
  - The Averager holds N samples for storing the data.



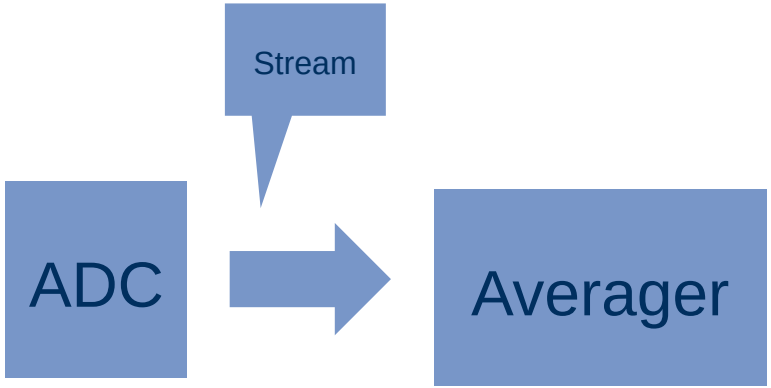
A system level perspective

- Memory-Estimation:
  - We need to aggregate N samples and pass them to the Averager
  - The Averager holds N samples for storing the data.
- We can also work directly on the data stream
  - Reduce memory demand by approx. factor 2



A system level perspective

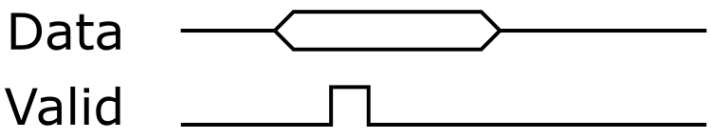
- Memory-Estimation:
  - We need to aggregate N samples and pass them to the Averager
  - The Averager holds N samples for storing the data.
- We can also work directly on the data stream
  - Reduce memory demand by approx. factor 2




```python
class Averager():

    max_samples = 128

    def __init__(self, Nsamples):
        self.mem = np.zeros(self.max_samples)
        self.Nsamples = Nsamples
        self.meas_counter = 0
        self.sample_counter = 0

    def add_sample(self, sample):
        self.sample_counter = self.sample_counter + 1

        if self.sample_counter == self.Nsamples:
            self.meas_counter = self.meas_counter + 1
            self.sample_counter = 0

        self.mem[self.sample_counter] = self.mem[self.sample_counter] + sample

    def get_result(self):
        return (self.mem[0:self.Nsamples-1] / self.meas_counter)
```

Interfacing

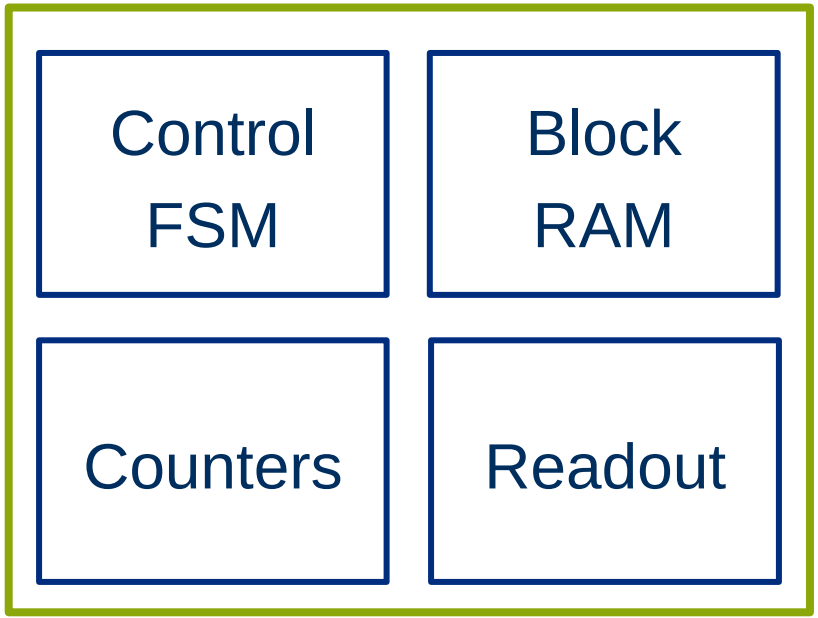
- We assume here a very simple data interface based on a handshake



- The readout is assumed to happen only when the measurement is done
  - Use a fully parallel Interface (like SRAM)
- Setting configuration registers is not shown here

Implementation idea 1 (FSM based)

- FSM = Finite State Machine
- Describe your algorithm by means of state transitions



Implementation idea 1 (FSM based)

- FSM = Finite State Machine

- Describe your algorithm by means of state transitions
  - Common pattern for hardware description
  - State is updated on each clock cycle
  - Step-by-Step description of what should be done
  - Is a sketch for implementation
  - Can be used for throughput estimation

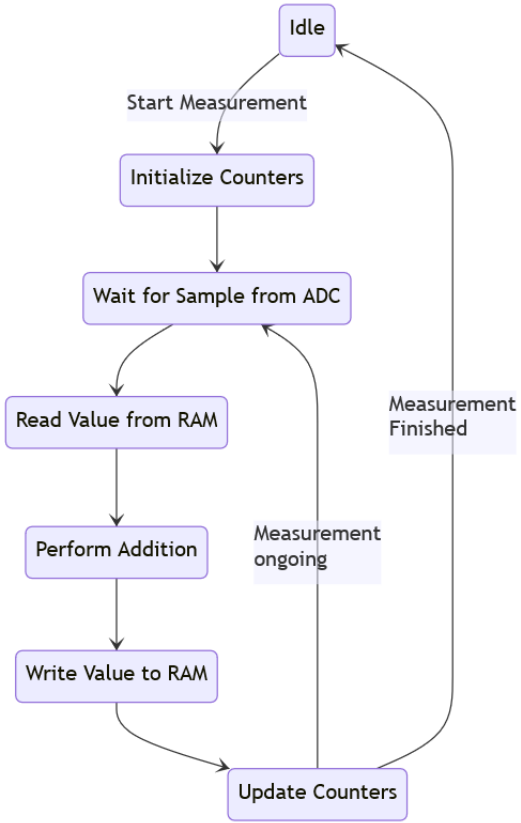


Diagram was created with mermaid.js

Implementation idea 1 (FSM based)

- Throughput estimation
  - How many data samples can I apply per clock cycle?
  - How many clock cycles are needed for processing a new data sample?
- Let‘s assume 1 cycle per state transition
  - 5 cycles per sample
  - Maximum data rate of (Fclk/5)
- Can we get faster?

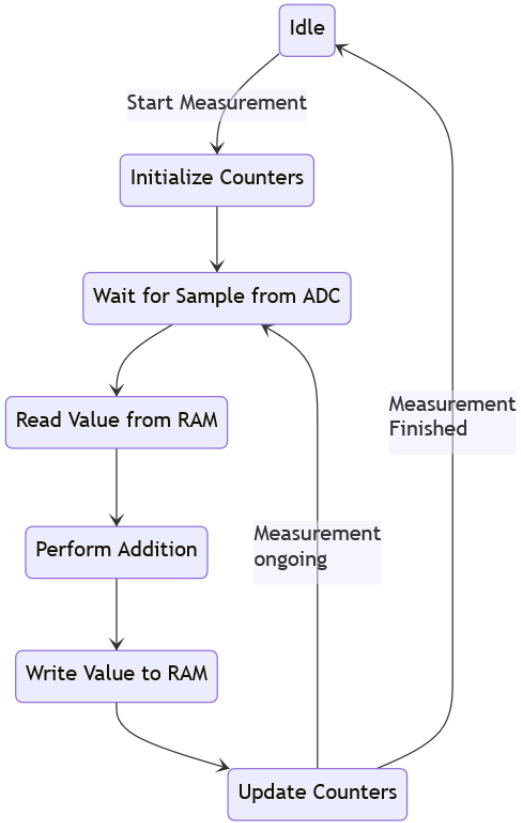
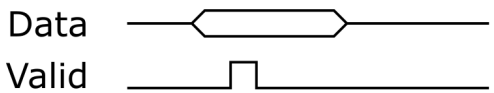


Diagram was created with mermaid.js

Implementation idea 2 (Pipeline)

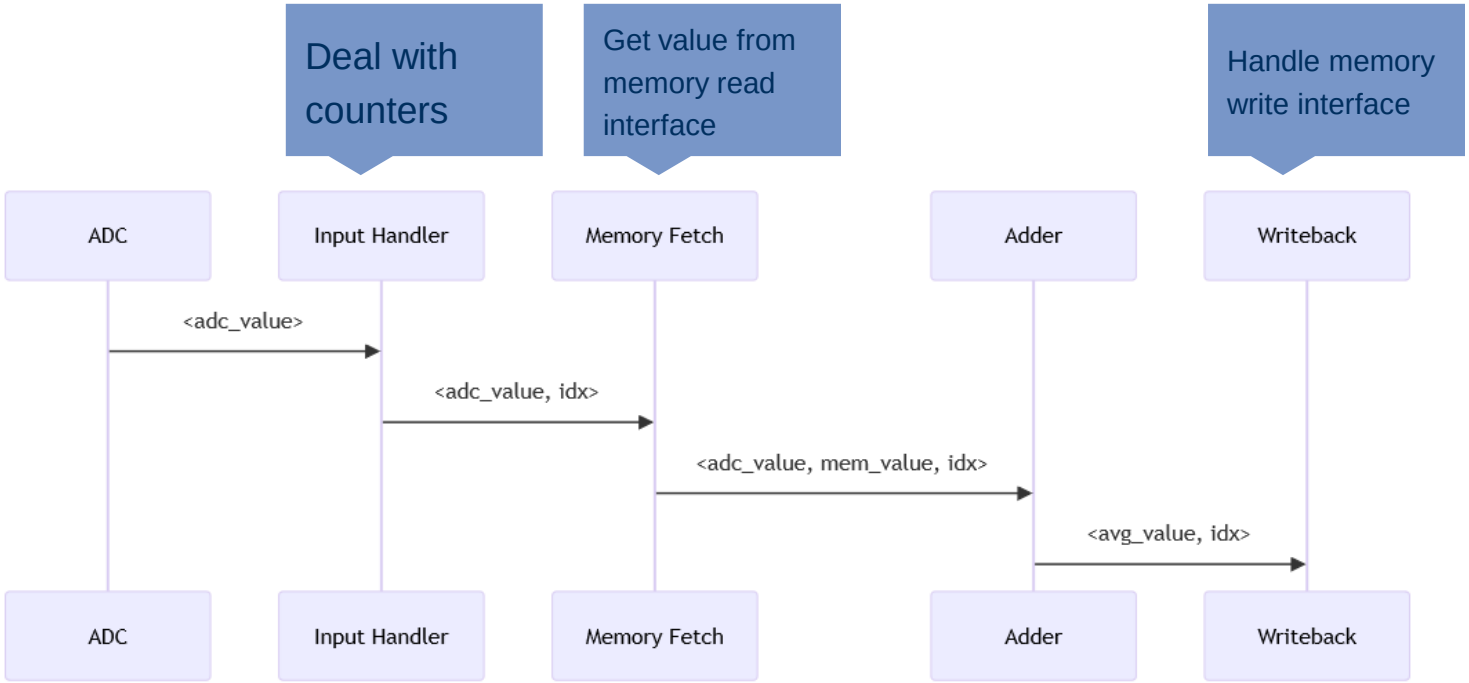
- We can go for a more parallel implementation
- Independent units for each step

- Prerequisites:
  - Dual-Port RAM (Independent Read/Write)
  - Augment each sample with meta-data (here: counter value)
  - Clean interfaces for each unit,
    we use the handshake from before



Implementation idea 2 (Pipeline)

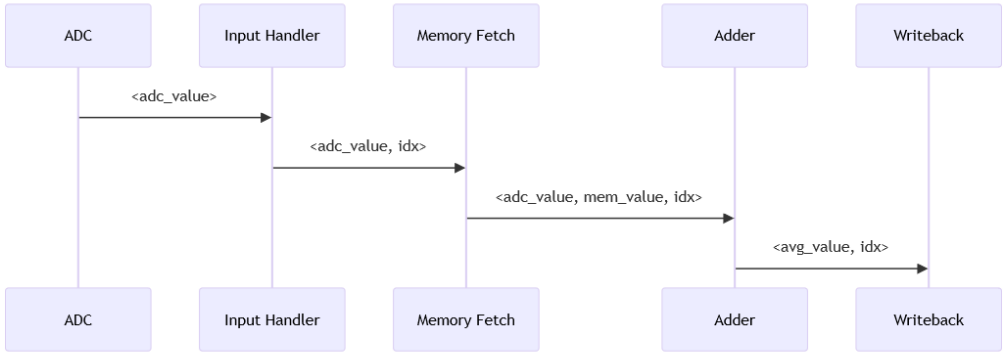
- We can go for a more parallel implementation
-



Implementation idea 2 (Pipeline)

- Throughput:
  - If each step needs 1 clock cycle:

    Maximum data rate: Fclk
- Latency:
  - How long does it take to process an item
  - Still 5 cycles



Comparison

FSM

- Maximum data rate: Fclk / 5
- RAM type: Single Port
- Structure: Monolithic

- Needs additional state variables
- Verification only in a single block
- Easily readable code

Pipeline

- Maximum data rate: Fclk
- RAM type: Dual Port
- Structure: Individual blocks

- Overhead for inter-block signalling
- Verification of individual blocks
  ➔ Easier to debug if you have a full set of TBs
- Code is harder to understand

Things I did not cover

- Actual realization of the components
- Data formats
  - From unsigned integer to arbitrary precision floats
- Timing constraints and clocking schemes
- Vivado or Quartus HowTo
- Processor or Accelerator design
  - There are special patterns for e.g. matrix multiplications
- Bus systems like Whishbone, AHB, AXI,...

General Hints for implementation

- Choose the pattern with the lowest implementation complexity
- Do not blindly implement but test your code in simulation
  - Yes, this seems like extra effort, but will help you debugging
- Choose your data formats wisely
  - Do you really need full-double-precision floats?
- Compare your hardware to your python code!
- Do benchmarks ;)

Where to start?

- Now that you have a brief overview
  - What FPGAs are
  - How to use them
  - Some ideas about implementation patterns

- Get an FPGA board, install the software and start coding!

Where to start?

- Now that you have a brief overview
  - What FPGAs are
  - How to use them
  - Some ideas about implementation patterns

- Get an FPGA board, install the software and start coding!

- After the Q&A and the break:
  - Deep-Dive into implementation practices and pitfalls

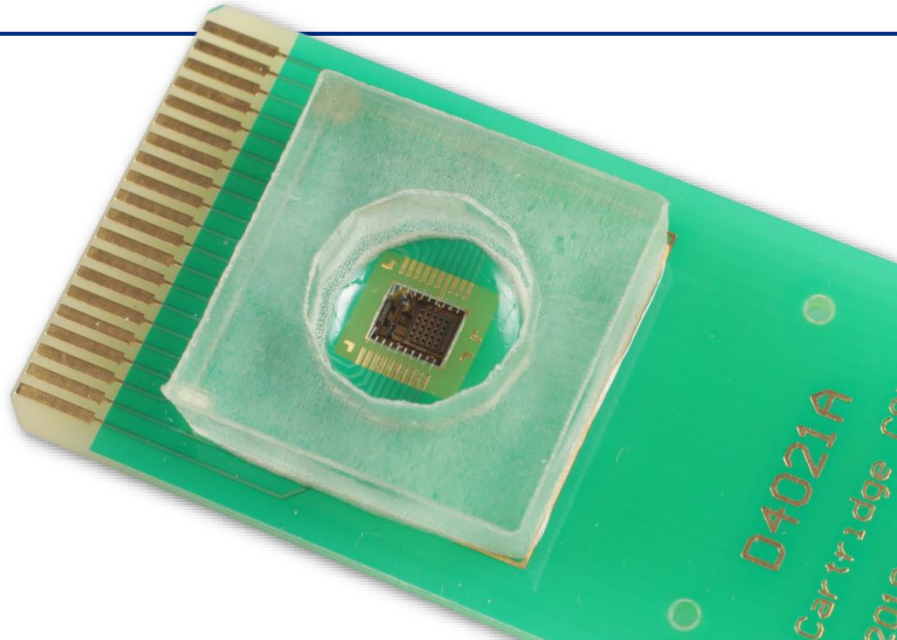
Time for coffee…

IMMS Institut für Mikroelektronik- und Mechatronik-Systeme gemeinnützige GmbH (IMMS GmbH)
Ehrenbergstr. 27, 98693 Ilmenau, Germany, Tel: +49 3677-8749300, Fax: +49 3677-8749315, **www.imms.de**
Dr. Georg Gläser, georg.glaeser@imms.de, phone: +49 361 663 25 33

IMMS

Institut für Mikroelektronik- und Mechatronik-Systeme gemeinnützige GmbH

Part 2: Practical Hints, Good and Bad Ideas



Motivation

- Actual coding is just a part of the work
- Structure and experience are important

- Not only experienced digital designers write code – so be aware of the basics
  - How to do things right?
  - Where are the pitfalls?

- Use-Cases collected from real designs
- Verilog-centric

- Goal: A slideset that you can review before starting your first design ;)

Good Design Practices

- Things that are known to work…

- Basic schemes that should be **always** followed!
  - If not, you'll have a hard time coding and debugging
  - If not, debugging will cause you to miss deadlines
  - If not, it may possibly introduce new bugs

- Most of the rules seem obvious
- Some of them seem to produce effort
  - That's effort you'll save later!

- Basic rule:

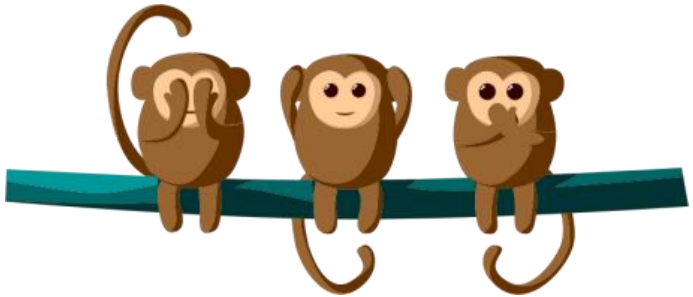
  Your code should be easy to read & understand!

www.openclipart.org CCo

Good Design Practices: Communication

- **Talk to an experienced digital designer!**

- Be prepared if he or she asks you:
  - What does the code do?
  - How does the reset work?
  - What is the clock speed?
    (maximum, max. trimming, corner, etc)
  - Are there internal clocks?



https://openclipart.org/detail/294188/see-hear-speak-monkeys

- If he or she misses out something:

  Talk to your digital designer!

Good Design Practices: Modularity / Architecture

- Ok, that would be another training (and material for a good book)

- Basics:
  - Seperation of concerns should be considered:
    Single-Responsibility Principle (SRP)
    (Each block has exactly one specific purpose)
  - **Cohesion** of components should be low
    (Functionality should not depend on many additional blocks)
  - **Testability** / Verificability
    (There is a simple possibility to verify a block – unit testing)
  - Clean **interfaces**
    (Use standards where possible, be always synchronous!)
  - **Do not repeat yourself**
    (Need for repetition is usually a sign for bad architecture)

Good Design Practice: Keep the implementation in mind!

- Just by example:

  Which Flip-Flop type is this?

```
always @ (posedge CLK or posedge RST)
begin
        if(RST)
                a <= 0;
        else
                a <= b;
 end
```

That‘s the D Flip-Flop with async reset!

Good Design Practice: Keep the implementation in mind!

- Just by example:

  Which Flip-Flop type is this?

```
always @ (posedge CLK or posedge RST or negedge EN)
begin
        if(CLK)
                a <= 0;
        else if (RST)
                a <= 1;
        else
                a <= b;
end
```

That‘s the impossible Flip-Flop!

Good Design Practice: Keep the implementation in mind!

- Just by (real) example:

  Which Flip-Flop type is this?

Impossible Flip-Flop with self-generated clock

```
reg CTRL = 1'b0;
initial begin
        #(1);
        CTRL = 1'b0;
end
always begin
        @(negedge RSTN or BIN, THERM0) begin
                if (!RSTN) begin
                        CTRL = 1'b1;
                        #(100);
                        CTRL = 1'b0;
                end
                else begin
                        CTRL = 1'b1;
                        #(100);
                        CTRL = 1'b0;
                end
        end
end
endmodule
```

Good Design Practice: Keep the implementation in mind!

- Just by (real) example:
  What's actually driving?

- Hint: May work in simulation

```
module mux4bit ( IN0, IN1, OUT, SEL );

input  [3:0] IN0;
output  [3:0] OUT;
input SEL;
input  [3:0] IN1;
reg [3:0] OUT;

always @(*)       begin
                  OUT=4'h0;   //Initialization
End
assign OUT= SEL ? IN1: IN0;

endmodule  //end module mux4bit
```

Asynchronous driver in always block combined with continuous driver in assign

Unpredictable in the best case, worst case: Conditional Short

Good Design Practice: Code Style

- Format your Code!
- Use comments (during development, not after)
  - Describe purpose on high level
- Keep your comments up to date
- Do not comment code!
- Keep your code clean
  - If you change something,
    always make your code better in quality
- Name your signals and wires clearly
- Name your states
- Do not mix functionality in always-Blocks!
  SRP!
- Be consistent!

Quick test for code quality

- Metric:

  #(WTF)/Line-of-Code

- How many screen pages does your code need to show ist functionality?
  - A testbench with ›500 lines is hard to debug even for simple bugs
  - A if/else-Statement › 2 screen pages demands for a mode elegant solution
  - …
- Develop a „feeling“ for code complexity that fits your collegues heads!
  - The complexity of undestandable code is limited by the reader (not by the designer)
  - Be kind to your verification engineer and future generations of designers!

Good Design Practice: Code Style

- Format your Code!

```
always@(posedge CLK_I or posedge RST_I)
begin
if(RST_I) begin
a <= 0;
end
else
begin
for (i=0;i<19;i=i+1)
begin
G[i] <= IN;
end
end
end
```

Good Design Practice: Code Style

- Format your Code!

```
always @(posedge CLK_I or posedge RST_I)
begin
        if(RST_I)
        begin
                a <= 0;
        end
        else
        begin
                for (i = 0; i < 19; i = i + 1)
                begin
                        G[i] <= IN;
                end
        end
end
```

Good Design Practice: Code Style

- Use comments (during development, not after)
  - Describe purpose on high level

```
// set EN to 1
EN <= 1;
```

Good Design Practice: Code Style

- Use comments (during development, not after)
  - Describe purpose on high level
  - Be specific in WHY you do something, not how.

```
// set EN to start conversion
// of CDC, will be reset in state X
EN <= 1;
```

Good Design Practice: Code Style

- Keep your comments up to date
- Do not comment code!
  - Use version control!

WTF?!

```
// set EN to start conversion
// of CDC, will be reset in state X
// EN <= 1;
// new version
EN <= 0;
```

Good Design Practice: Code Style

- Keep your code clean
  - If you change something,
    always make your code better in quality

```
// End conversion cycle
EN <= 0;
```

Good Design Practice: Code Style

- Do not mix functionality in always-Blocks!
  SRP!

```
always@(posedge CLK_I or posedge RST_I)
begin
        // Cares about I2C and CDC FSM
        …
        //I2C Part 1
        …
        //CDC FSM
        …
        //I2C synchronizer FFs
        …
end
```

Good Design Practice: Code Style

- Be consistent

```
always@(posedge CLK_I or posedge RST_I)
begin
        if(RST_I)
                a = 0;
        else
        begin
                for (i=0;i<19;i=i+1)
                        G[i] <= IN;
        end
end
```

WTF?!

Good Design Practice: Code Style

- Be consistent!

```
always@(posedge CLK_I or posedge RST_I)
begin
        if(RST_I) begin
                a <= 0;
        end
        else
        begin
                for (i=0;i<19;i=i+1)
                begin
                        G[i] <= IN;
                end
        end
end
```

Good Design Practice: Code Style

- Format your Code!
- Use comments (during development, not after)
  - Describe purpose on high level
- Keep your comments up to date
- Do not comment code!
- Keep your code clean
  - If you change something,
    always make your code better in quality
- Name your signals and wires clearly
- Name your states
- Do not mix functionality in always-Blocks!
  SRP!
- Be consistent!

Good Design Practice: Testbenches

- … should be **always** self-checking!
  - If not: You have to manually re-validate your simulation every time you change sth!
  - Be sure: Eventually you forget it (and beware of Murphy's Law for TapeOut)
- Write a simple testbench **before** you start coding!
  - Ensures good test-coverage
  - Fall-Back plan, if something changes fundamentally
- Keep your testbenches for later
  - If there is a need for re-verification
- Integrate your testbenches in a CI environment
  - Keep track of your design and discover errors early!

Test Driven Development

- Why test?
  - Verification
  - Test, if system still works after small (virtually simple) changes
  - Monitor the design state

Test Driven Development

- Kinds of tests
  - Unit tests
    - Test behaviour of single design units without context (Just with inputs and outputs)
  - Integration tests
    - Test integration of several units together ➔ Test interaction of units
  - Acceptance tests
    - Test, if the built construct fulfils requirements

Test Driven Design Methods and Rules (Uncle Bob, „Clean Code“)

- You are not allowed to write any production code unless it is to make a failing unit test pass.
- You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
- You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

- Test everything!!!
- Automate your tests!!!

Architecture rules also apply to tests!

- Seperation of concerns should be considered:

  Single-Responsibility Principle (SRP)

  (Each testbench tests exactly one specific feature/issue)

- **Cohesion** of components should be low

  (Functionality should not depend on many additional blocks around the DUT)

- **Testability** / Verificability

  (Debugging does not require going through hundreds of waveforms)

- Clean **interfaces**

  (Use standards where possible, be always synchronous! Use interface packages)

- **Do not repeat yourself**

  (Need for repetition is usually a sign for bad architecture, copied test code over tbs violates SRP!)

Things to adopt

- Registered Outputs
- Clean sensitivity lists
- State Machines
- Synchronous Design! (Don‘t mess with async!)
- Consistent Blocking and Non-Blocking Assigns
- Code Documentation

Registered Outputs

- Why put all outputs to Flip-Flops?
  - Glitch-Free output signals
  - Avoid asynchronous paths crossing several modules
    (eliminates most timing issues)

- How?
  - Assign outputs only in synchronous processes!
    Always@(Clock[…])
  - Do not assign outputs with asynchronous logic, e.g.

```
wire a = (CNT1==1)?1:0;
wire b = (CNT2==2)?0:1;
assign OUT = EN?a:b;
```

Clean Sensitivity Lists

- Why?
  - Reduce simulation / synthesis mismatches
  - Clarifies misunderstandings for sync and async processes
- How?
  - Synchronous process:
    - Always @(‹pos/neg›edge CLK or ‹pos/neg›edge RST)
  - Async processes:
    - Always @(*)

Clean Sensitivity Lists

- Bad async example
  - Change on IN3 does not change value of OUT in simulation

    … but in synthesis

```
always @(IN1, IN2)
begin
        OUT = IN1?IN2:IN3;
end
```

State Machines

- The best way to describe synchronous behavior
- Why
  - supported by all synthesis tools
- How?
  - Either one-process or two-process notation
  - Choose **only** one notation
- Name your states!
- (If you did not understand what I'm talking about, take a look in first-semester digital design lectures!)

Synchronous Design!

- Do everything synchronously

- Why
  - Complete design flow relies on synchronisity and clocking constraints
  - Verilog is just not powerful enough to handle e.g. synthesizeable async state machines without additional tricks
  - Asynchronous logic might cause glitches!

- Be synchronous and be happy with it!

How to design glitch-free blocks

- Asynchronous logic might glitch
  - Due to different path delays
  - Due to unstable inputs
  - …
- Destroys your power budget and disturbs control signals for analog components

https://openmoji.org/library/#search=cry&emoji=1F62D CC-BY-SA 4.0

How to design glitch-free blocks

- Place Flip-Flops at the all inputs and outputs of a block
  - Keeps asynchronous paths short
  - Prevents asynchronous paths to spread along several modules
  - Makes the output glitch-free

Code Documentation

- Code Documentation is like working in the kitchen:
  Cooking and eating is more fun than doing the dishes

- A lifehack from youtube:
  If cleaning is easy, the probability of doing it is higher.

- Hence: The effort for code documenation has to be as low as possible

- In general:
  Document what you implement, why and whatfor? and especially: Why like this?
  - Non-obvious design descisions are of special interest

Code Documentation

- Integrate Documentation in your design repository!
  - Write your doc in parallel to your code
  - Hence, the doc will always match your recent code version
  - Ok, it‘s about discipline

- Tools
  - Markdown can be versioned and is very powerful
  - Most editors have good plugins for Markdown, e.g. Markdown Preview Enhanced in VSCode supporting
    State charts, Flow charts, waveforms, etc.
  - Your markdown can be converted to different format for getting your documentation into a datasheet or similar

Things to avoid

- No combinational loops
- No unintended Latches
- No Delay-based Design
- No Multi-Assigns
- Unconnected Inputs

No combinational loops

- Endless-loops in zero-time
  - Causes the simulator to hang forever
  - Hard to debug
  - Leads to (unstable) oscillating digital systems after synthesis
  - Just don‘t do it.

Example:

```
wire a,b;

Assign a = ~b;

Assign b = a;
```

No unintendet latches

- Only instantiate latches if needed, be aware of latch descriptions!
- Unintended latches may cause simulation/synthesis problems and destroy timing!

```
//synthesizes to latch
Always @(*)
Begin
        if (enable1)
                a=IN;
end
```

```
//synthesizes to combinational logic
Always @(*)
Begin
        a = 0;
        if (enable1)
                a=IN;
end
```

No delay based design

- Delay elements are subject to process variations
  - Destroys timing analysis
- Just don‘t do it.

```
// A very bad edge detector
//   pulsewidth on c is completely unpredictable and not included
//   in timing analysis
Assign a= #1 b;
Assign c = (a!=b);
Always @ (posedge c)
[…]
```

No Multi-Assigns

- Never ever assign a signal from two processes
  - Best case: Result is unpredictable
  - Worst case: [Conditional] Short (multiple drivers on one line)

```
Always @ (posedge clk)
begin
[…]
                a <= 1;
End

Always @(posedge clk2)
                a <= 0;
```

Unconnected Inputs

- Best case: 0
- Worst case: transient random

- Connect everything to defined levels!

Version Control, Bug Tracking, Code distribution,...

- Use Cases
- Basics of git
- Use model for Verilog development
- Gitlab
  - Issue Tracking
  - Continuous Integration

Use cases

- Version control
  - Code reviews
  - Blaming someone with a bug ;D
  - **Tracking your progress** (or: Why did it actually work yesterday?)
- Code distribution
  - Supplying every developer with the recent code version
    (without manual copy stuff)
- Releases and CI
  - Execute checks on every commit

Basics of git

- How to initialize a git repository?

```
git init .
```

- How to commit a new revision of a file?

```
git add <yourfile.v>
git commit
```

  - Several files can be commited together
  - Please use meaningful commit messages!

- Cheat sheet:

  https://about.gitlab.com/images/press/git-cheat-sheet.pdf

Basics of git: Working with remotes

- A remote is a centralized git repo (e.g. at our Gitlab server)
- Cloning a remote from the server

  `git clone <your url>`
- How to push your changes to the server (after commit)

  `git push`
- How to get recent changes from the server?

  `git pull`

- Left out issues: Merges, Branches,...
  - Hard to teach on slide, but easy to use
  - Solves the problem of working on a project with a team

Working with Gitlab – Issue Tracking

- Make your digital guy‘s life easier by describing your requirements or his bugs
  - Write notes that he gets onto his todo-list

- Hints:
  - Give meaningful names
  - Extensive descriptions
  - Track progress

- Just a tool for organization, not for communication

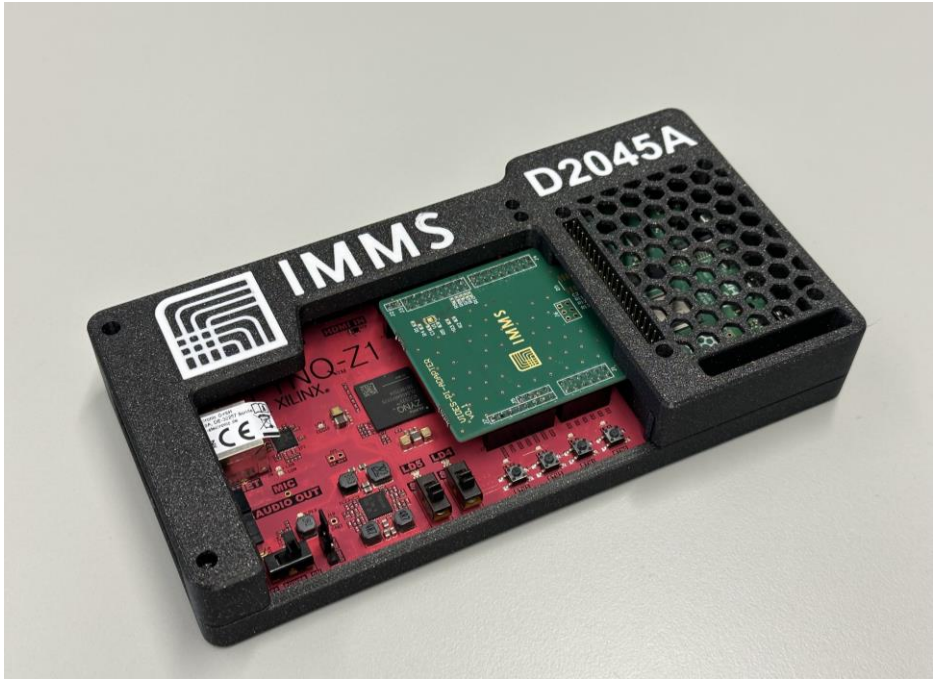
  ➔ You need to talk to people

Working with Gitlab – Continuous Integration

- What for?
  - Monitor your design‘s state by running the tests automatically
  - Run tests in a fixed schedule and saves results
  - Recognize errors early and fixed imidiately
- How?
  - Define scripts in gitlab_ci.yml
  - Apply a run schedule
  - Observe the results!!!
- Prerequisites
  - Self-checking testbenches
  - Automated run-script (avaliable)

Summary & Take Home Messages

- FPGA Development exhibits potential for
  - Fast execution of parallelized algorithms
  - Real time processing
  - Direct stream processing
- But… you have to invest additional effort
  - Design flow
  - Simulation & Testing
  - Bring-up of a hardware platforms
- Common challenges shown here… there is more ;)



Together with you, we would like to work on the next batch of upcoming ideas!

IMMS Institut für Mikroelektronik- und Mechatronik-Systeme gemeinnützige GmbH (IMMS GmbH)
Ehrenbergstr. 27, 98693 Ilmenau, Germany, Tel: +49 3677-8749300, Fax: +49 3677-8749315, **www.imms.de**
Dr. Georg Gläser, georg.glaeser@imms.de, phone: +49 361 663 25 33