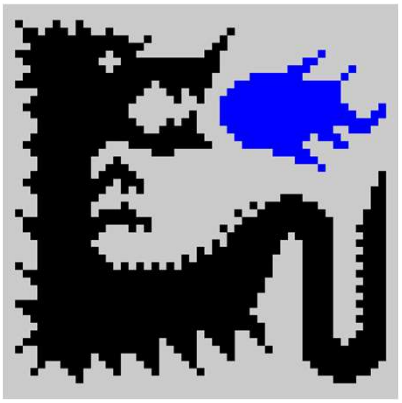


EMBER: A Cycle-based Framework for Early-Stage Reliability Assessment in Parametric RTL Designs

TWIN-RELEC Training School on Fault Tolerant AI

Alessandro Veronesi

Leibniz Institute for high performance microelectronics

14.01.2026

Downloading the Demo

Requirements:

- **Docker + Microsoft's Dev Containers extension**
- **GIT**

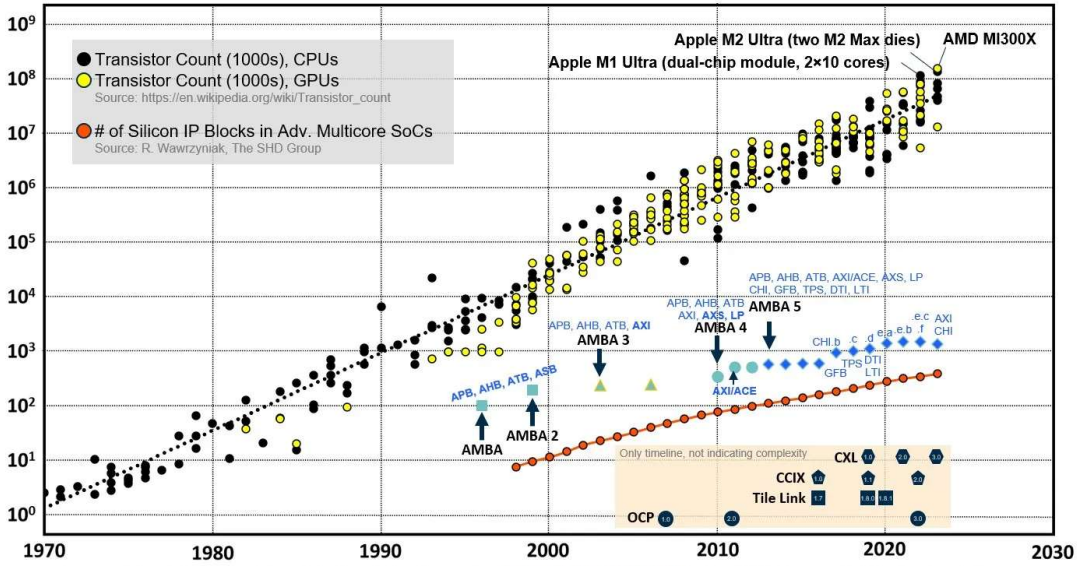
```
# Download the Demo repository (for VS Code UI)
# Otherwise visit my GitHub page (username: AlessandroVeronesi)
$ git clone git@github.com:AlessandroVeronesi/EMBER-Demo.git

# Download the Docker image
$ docker pull averonesi/ember-demo:latest
```

The Need for Early-Stage Reliability Analysis

Modern microchips integrate **complex microarchitectures with scaled technologies**, thus **increasing** the threat caused by **on-field faults** during the product's lifecycle.

- **Strict constraints** require designers to account for system metrics **since the initial design stage** (**Design-for-X**)
- **Anticipating the fix** of a design issue **means reducing the cost and the ease** of the fix itself.



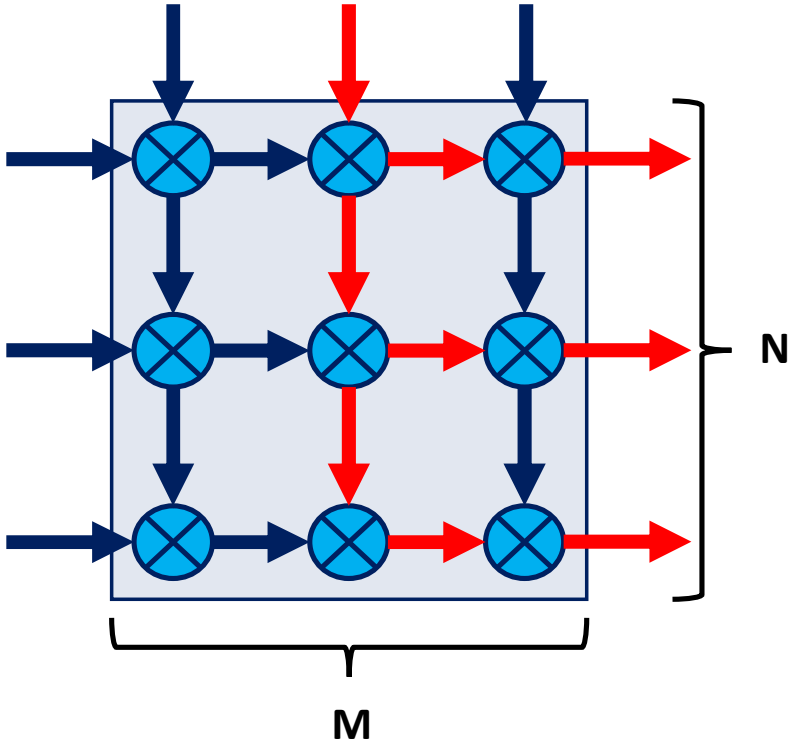
Source: Frank Schirrmeister, Arteris, plotted based on https://bit.ly/3P5y9OK, The SHD Group, Public Protocol Specs

There is a **growing need for anticipating reliability analysis** to early design stages

Reliability-driven Co-design

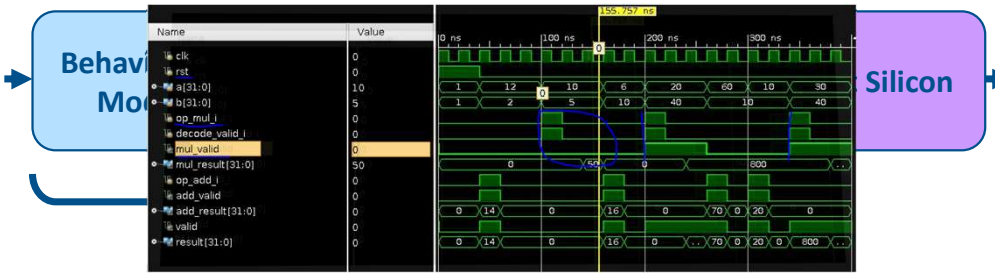
This trend is **especially visible in parallel computing architecture**, as their **data reuse policies can propagate corrupted data**, thus both **increasing** the **error propagation probability and** their **criticality** w.r.t. the running application.

- A **single corrupted data may be reused throughout all the computation** and **broadcasted to different outputs** (higher criticality), but also it increases the probability to be computed with sensitive data (lower masking probability)
- This phenomenon is **strongly correlated with the hardware configuration parameters**: the number of parallel hardware components both affect the fault propagation and the registers lifetime.



Early-Stage Reliability Assessment Techniques

Simulation-based Fault Injection



- **Allows mixed-abstraction simulation** (only injected HW unit requires to be non-functional)
- **Slower evaluation** (simulated time is not as fast as real time)
- **Inexpensive** (only requires PC and simulation tool)

Emulation-based Fault Injection

Simulation-based approaches are more suited for early-stage explorations

sizable

mance are the e)

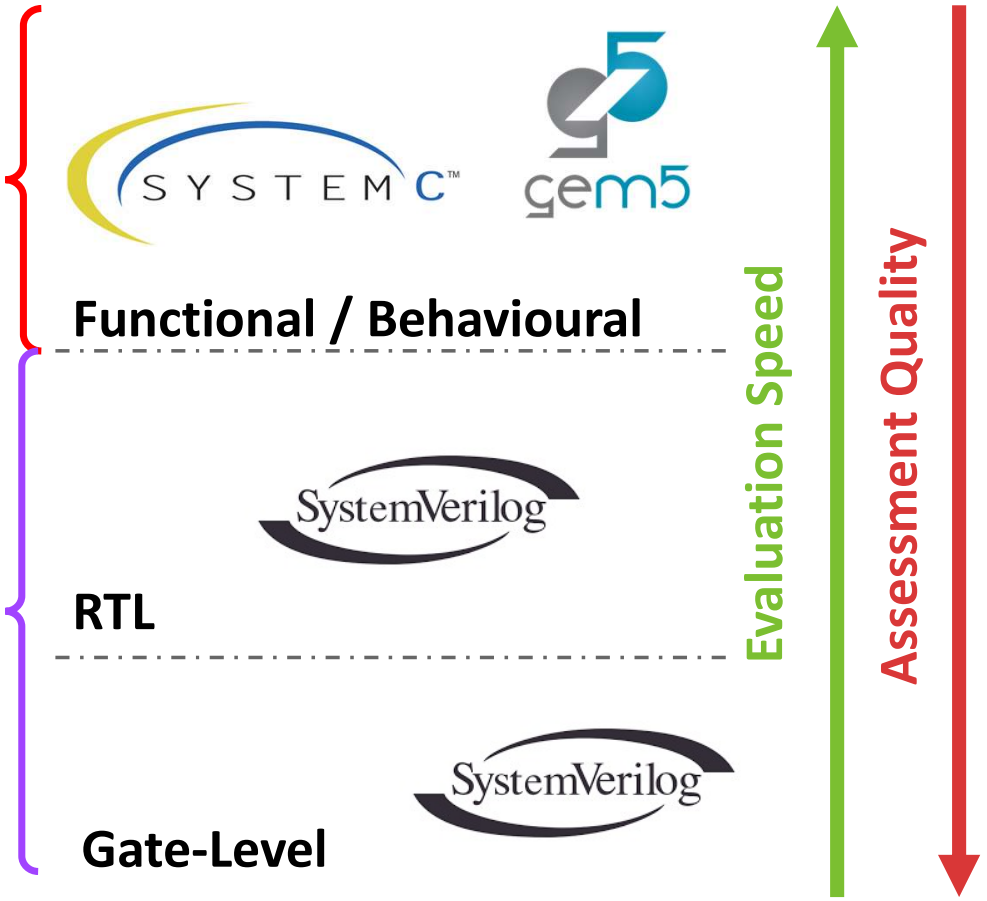
- **Expensive** (requries an FPGA prototype)

Simulation-based SOTA

The major limit of simulation-based approach is the **prohibitive simulation time**:

- Option A: **Abstract to higher abstraction levels** (complex to represent fault behaviours)
- Option B: Adopting **simulation approaches with better simulation performance** than event-driven simulation

IDEA:
Adopting State-of-the-Art Simulation Techniques

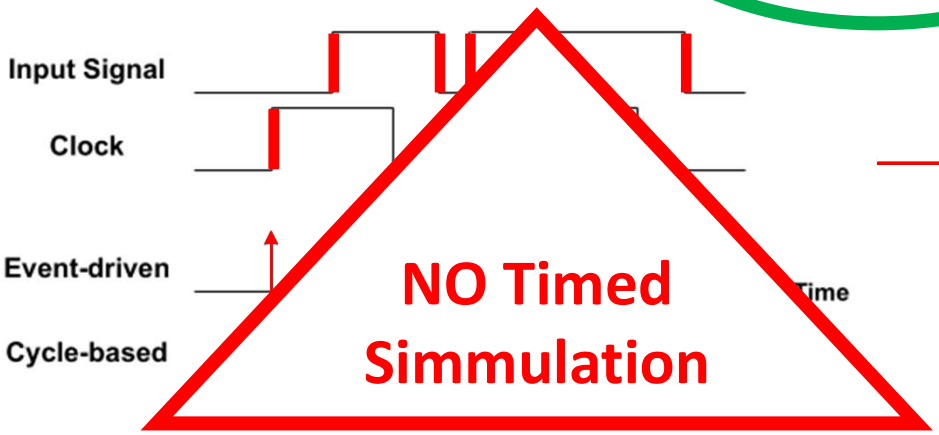


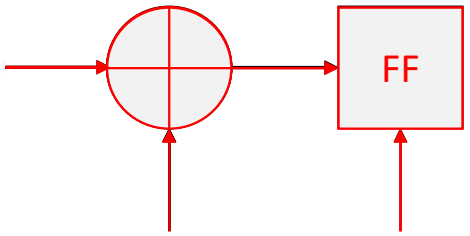
Event-Driven VS Cycle-Based: Why So Fast?

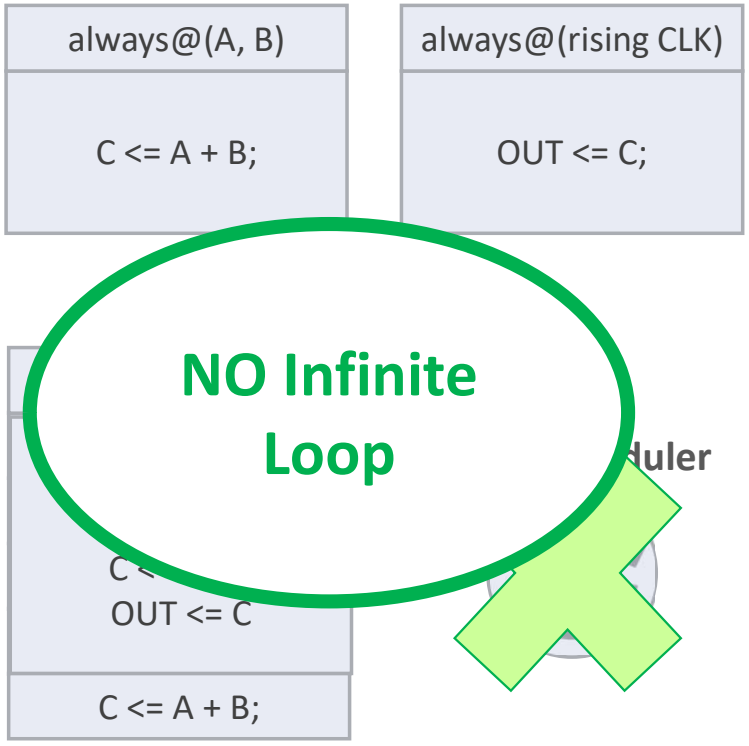
The **performance boost** cycle-based simulation expose is **caused by** the **smaller number of processes** and by the **absence** of an **event scheduler**

- **Requires Zero Gate Delay** ass
- **Cannot handle events betwe**

LIGHTWEIGHT







The EMBER Framework

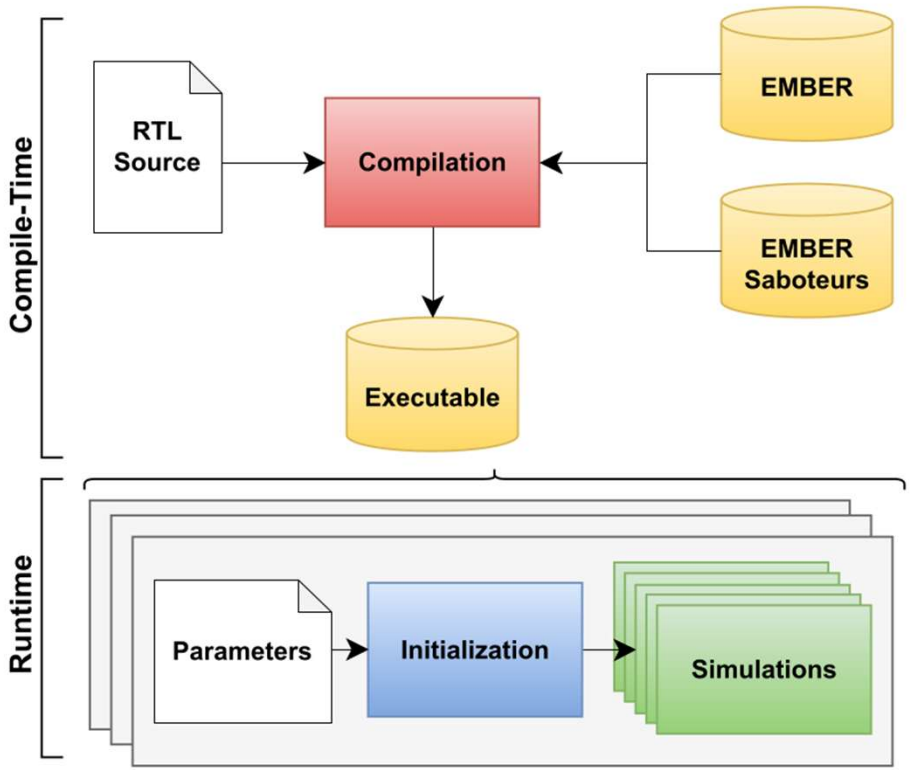
The **Extensible Microarchitectural Benchmark for Error Resilience (EMBER)** is a **C++ RTL modelling library** with **cycle-based simulation paradigm**, and **built-in fault-injection** support.

- **Cycle-Based simulation paradigm**
  - **Faster than event-driven RTL** simulation
- **Exploits C++ for agile hardware description**
  - Exploits **C++ meta-language** features **for datapath** description
  - Design‘s **configurability parameters** are **evaluated at** class‘ **initialization time**
- **Supports Fault-Injection through Software-friendly Saboteurs**
  - **No need for design customization** for saboteurs insertion
  - **Decuples RTL design** from **fault behaviour modeling**
  - **Designed to be extensible** in fault models, features, etc.

Designing Hardware with EMBER

EMBER adopts a similar workflow to other non-Verilog and non-VHDL based **hardware frameworks**:

- The **Hardware Module description** is **directly described in C++** using EMBER classes
- Also the **Saboteurs Library is described in C++** with EMBER classes
- The **Simulation Binary is compiled before** the initialization of modules‘ **parameters**, which are **initialized at runtime**

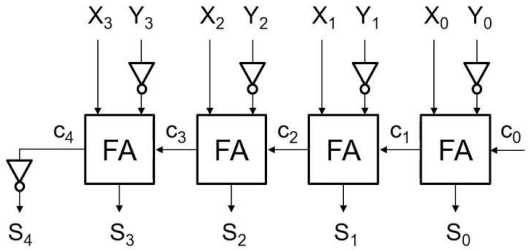


EMBER Agile Design and C++ Templates

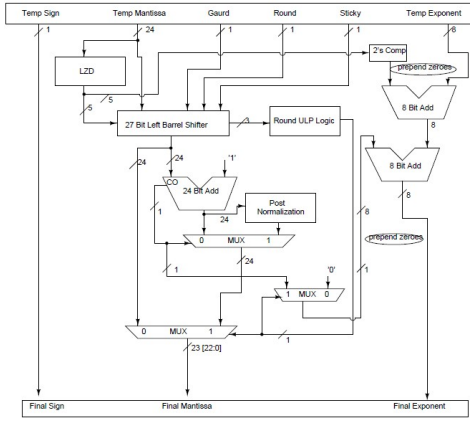
EMBER exploits **C++ meta-language** features for **quick datapath reconfiguration.**

```
1 template<class dtype>
2 void adder<dtype>::eval()
3 {
4     C.write(A.read() + B.read());
5 }
```

adder<ember::int8_t>

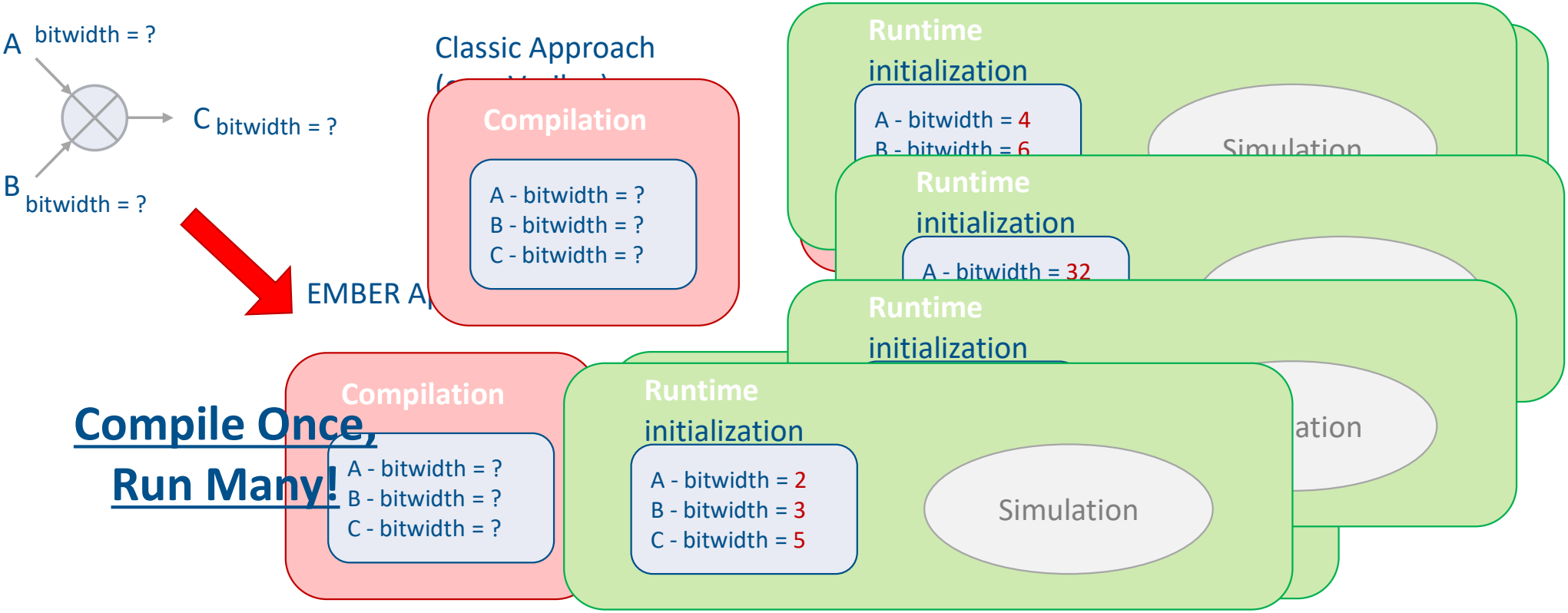


adder<ember::fp32_t>



The Configuration Initialization Step

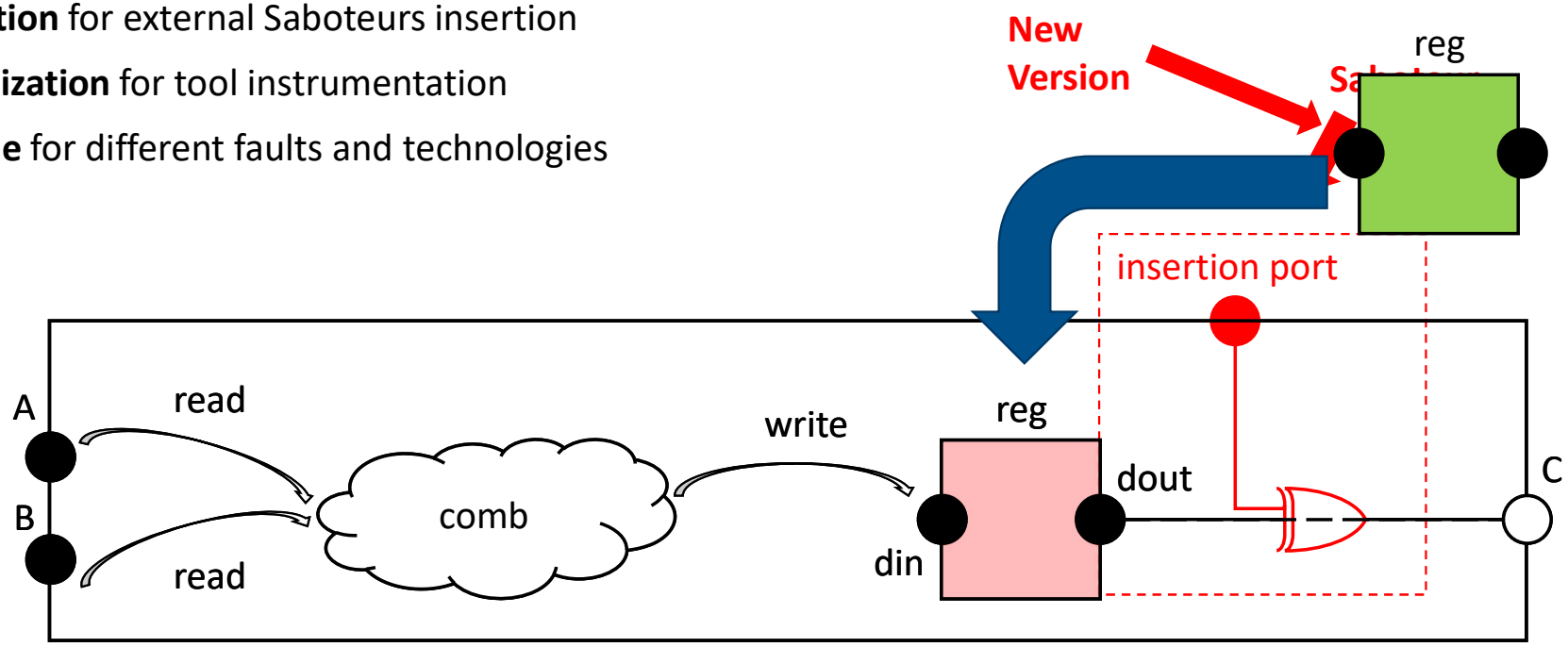
Together with **DUT Compilation** and **Simulation**, **EMBER introduces** a **DUT Initialization** step.



Fault Injection via Software Saboteurs

EMBER adopts **software saboteurs** for supporting **fault injection** capabilities

- **No RTL customization** for external Saboteurs insertion
- **No Scripts customization** for tool instrumentation
- **Easily customizable** for different faults and technologies



EMBER Modules

EMBER Hardware components are **subclasses of the abstract IModule class.** Thus, **EMBER hardware modules** have to **implement abstract methods** described in the IModule interface:

- **evaluate:** describes **combinational logic** behaviour;
- **update:** describes **clock-sensitive elements** behaviour;
- **reset:** describes sequential elements behaviour during the **reset** phase;
- **getSaboteurs:** returns a **list of saboteurs contained** in the **module** and in its **submodules**;
- **Class' Constructor:** contains all the **information to initialize the design with**, **including** subcomponent's **binding and submodules** hierarchical initialization.

```
1 class ember::IModule {
2  public:
3     virtual void          update() = 0;
4     virtual void            eval() = 0;
5     virtual std::vector<ISaboteur*>
6                         getSaboteurs() = 0;
7     virtual void           reset() = 0;
8 };
```

EMBER Saboteurs

EMBER saboteurs are **regular EMBER modules** which **additionally inherit the ISaboteur class**. **EMBER saboteurs are also EMBER modules.** Again, saboteurs description is achieved by methods overriding:

- **locations:** returns the number of physical locations in which the fault model can be injected;
- **getFaultMask:** generate a fault mask containing information about how to corrupt the signals in the EMBER module;
- **clearFaultMask:** clear the fault mask;
- **applyFault:** apply the fault mask;

```
template<ember::fault_t fmodel>
ember::ISaboteur {
 public:
    virtual const size_t
                      locations() const = 0;
    virtual void genFaultMask()       = 0;
    virtual void clearFaultMask()     = 0;
    virtual void applyFault()         = 0;
};
```

EMBER Ports

EMBER uses **ports to define components Inputs and Outputs.** In fact, EMBER ports are used to define **components connectivity**.

To use ports, there are some rules:

```
1 ember::inPort<ember::int8_t> myIPort;
2 ember::outPort<ember::int8_t> myOPort;
3
4 myIPort.bind();          // Self-Bound Port
5 myOPort.bind(myIPort); // Connection
6
7 myIPort.write(6);        // Writing to a Port
8 var = myOPort.read();   // Reading from a Port
```

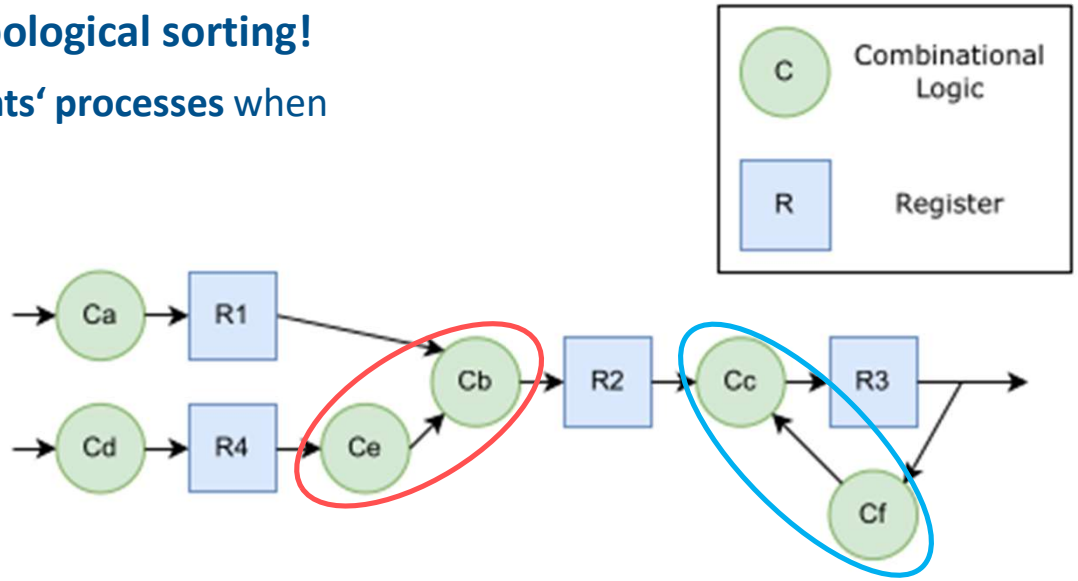
- Ports **need to be bound to each other** to connect components;
- **Ports can be self-bound** if not connected to anything else (or if their connectivity is later defined);
- **Ports can be written or read** to send/receive data;

EMBER Hierarchical Modules

Currently, **EMBER does not support automatic topological sorting!**

Designers are required to **manually schedule components' processes** when designing hierarchical modules.

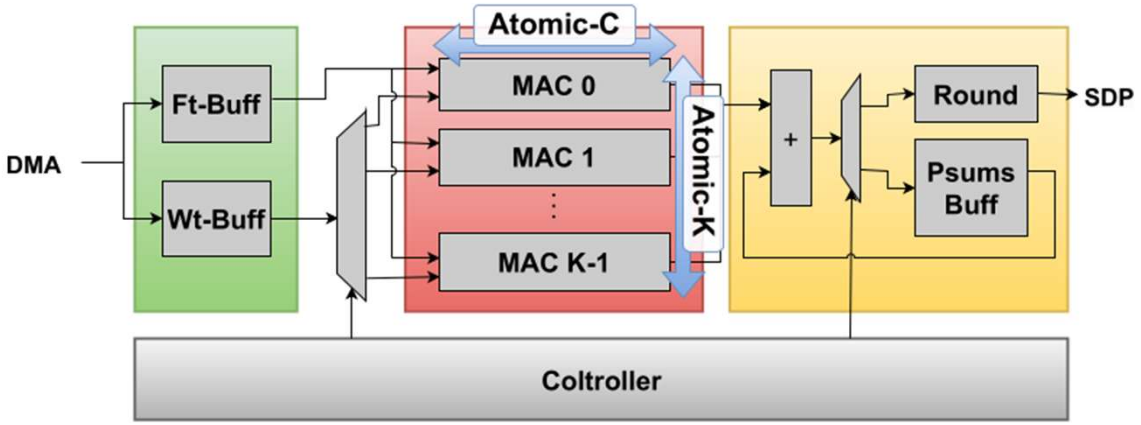
```
1 void top::eval()
2 {
3   Cf.eval();
4   Cc.eval();
5   Ca.eval();
6   Ce.eval();
7   Cd.eval();
8   Cb.eval();
9 }
```



Case Study: SEUs in NVDLA Accelerators

We deployed **EMBER** to model the **NVIDIA Deep Learning Accelerator (NVDLA)** and test its robustness agains **Single-Event Upsets (SEUs)**

- DNNs require **large fault injection campaigns** to draw statistically relevant error populations
- **SEU injection require cycle-accuracy** and **cannot be done in higher** hardware **abstractions**
- We injected SEUs in FFs for the **whole DLA pipeline**

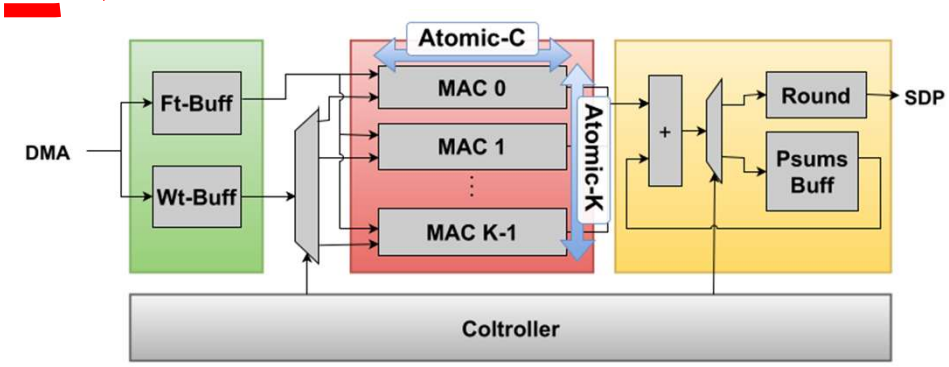


How Fast is EMBER?

HW Parameters	Value(s)
Atomic-C	8, 16, 32
Atomic-K	8, 16, 32
Wt-Buff	512 kB
Ft-Buff	64 kB
Psums-Buff	64 kB
Precision	Int8, int16

For our case study we considered 18 different DLA configurations!!!

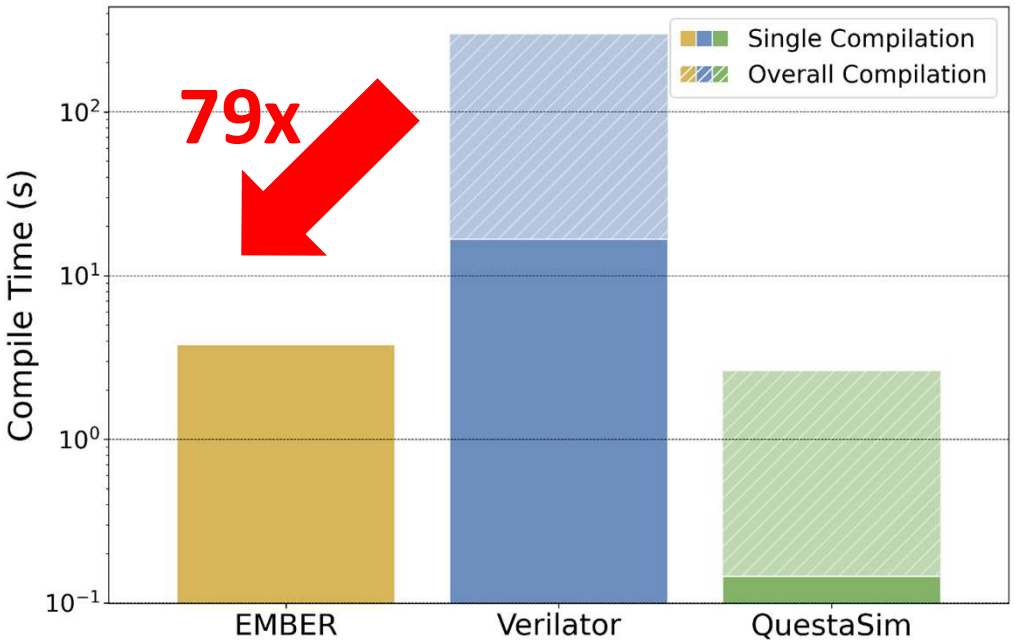
	Ft Tensor	Wt Tensor	Stride	Padding	Dilation
		1	1	2	1
		x11	1	2	1
		3x3	1	1	1
		3x3	1	1	1
Conv5	1x256x5x5	256x256x3x3	1	1	1



How Fast is EMBER? (contd.)

Differently from Verilator and QuestaSim,
EMBER compile time does not scale with the number of configurations

- **QuestaSim** has **great compilation performance** due to ist „Just-in-Time“ approach
- **Verilator & EMBER** follow the full Ahead-of-Time **compilation flow of the C++**
- The compilation time complexity of **EMBER remains constant** O(1), while **other approaches grow linearly** with O(n)

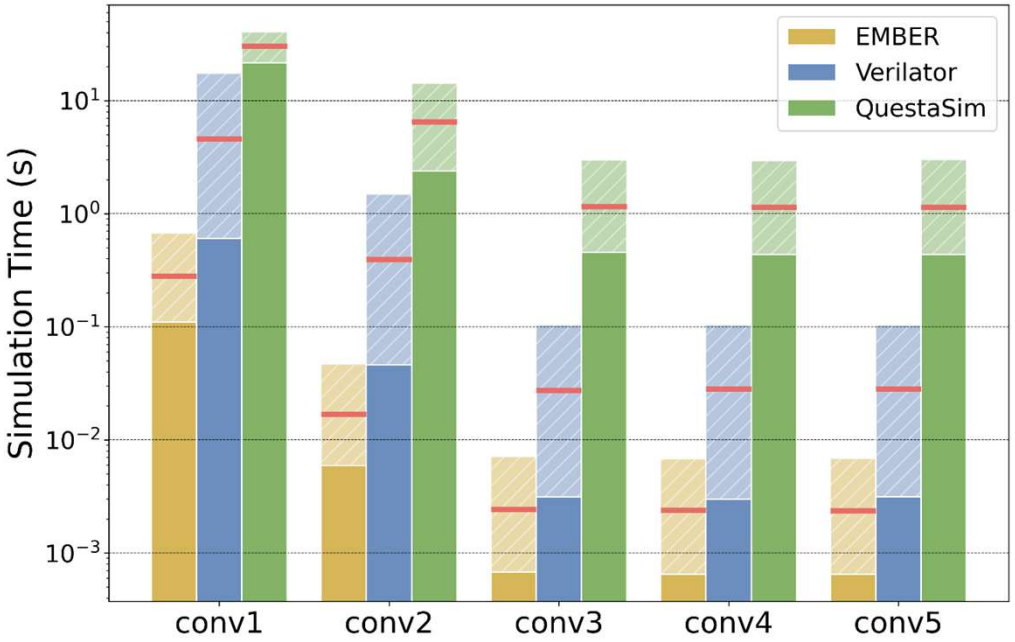


Time measurements have been performed on an Intel® Xeon W7, with 64GB DRAM

How Fast is EMBER? (contd.)

EMBER presents also **superior simulation performance** due to its **better use of CPU resources**

- **Verilator** exposes a simulation time **up to 147.8x faster than QuestaSim**
- **EMBER** exposes a simulation time **up to 37.6x faster than Verilator** and **up to 813.2x faster than QuestaSim**
- This improved performance is caused by the **better use of CPU resources**, since **Verilator is bound to Verilog's limited semantic**



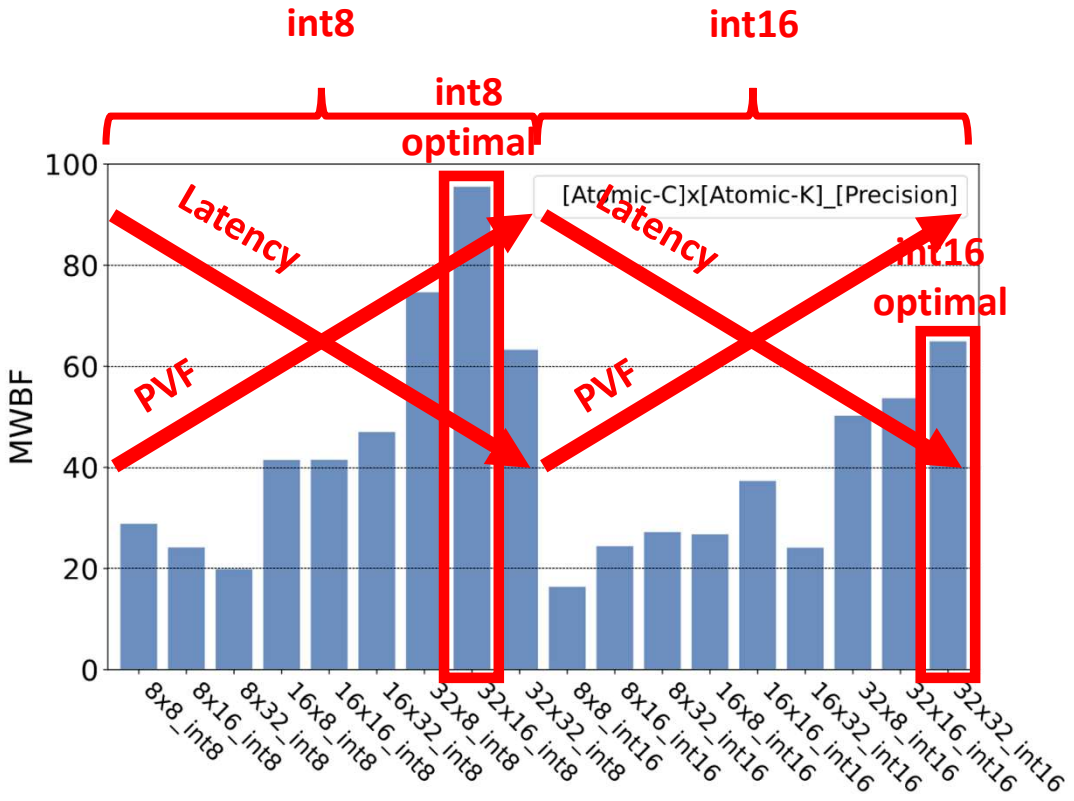
Time measurements have been performed on an Intel® Xeon W7, with 64GB DRAM

Applied Reliability-driven Co-design in NVDLA

The **effectiveness of reliability-driven co-design** is demonstrated by observing the **Mean Work Between Failures (MWBF)** of the NVDLA

- N_{FF} **is almost constant** as **the buffer size doesn't change** across configurations
- **Multiple microarchitectural effects** impact the **percormance-reliability trade-off**

$$MWBF = \frac{FPS}{BER_{FF} \times N_{FF} \times PVF}$$



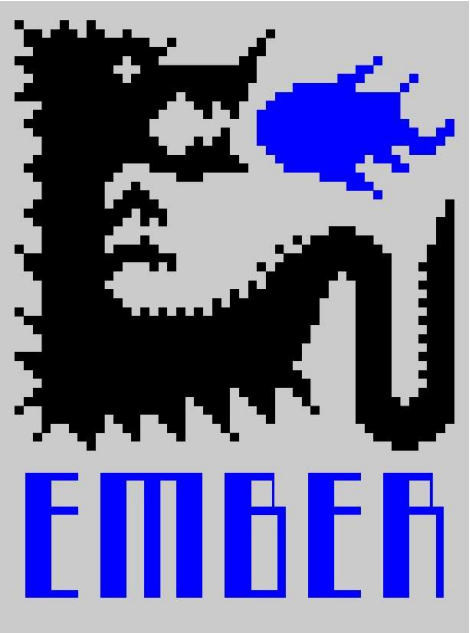
Wrap-Up and Future Challenges

EMBER offers **good performance** and **customizability,** enabling **technology-aware** microarchitectural **design-space explorations**

- **Cycle-based simulation** is a great paradigm for **simulation-based fault injection** approaches
- **Easy to integrate** with custom systems and applications (full C++ description)
- By adopting **software saboteurs** it potentially **allows** for **highly accurate fault behaviour modeling**
- **Designed to be extensible and customizable** for different teams needs

Future steps:

- **Automatic Verilog code generation**
- Automatic **Topological sorting** of combinational logic
- Simplified support for **additional fault models** and **technologies**



Available at:
https://github.com/IHP-Neuromorphic/EMBER

Time for a Break…

See you in few mins…

Downloading the Demo

Requirements:

- **Docker + Microsoft's Dev Containers extension**
- **GIT**

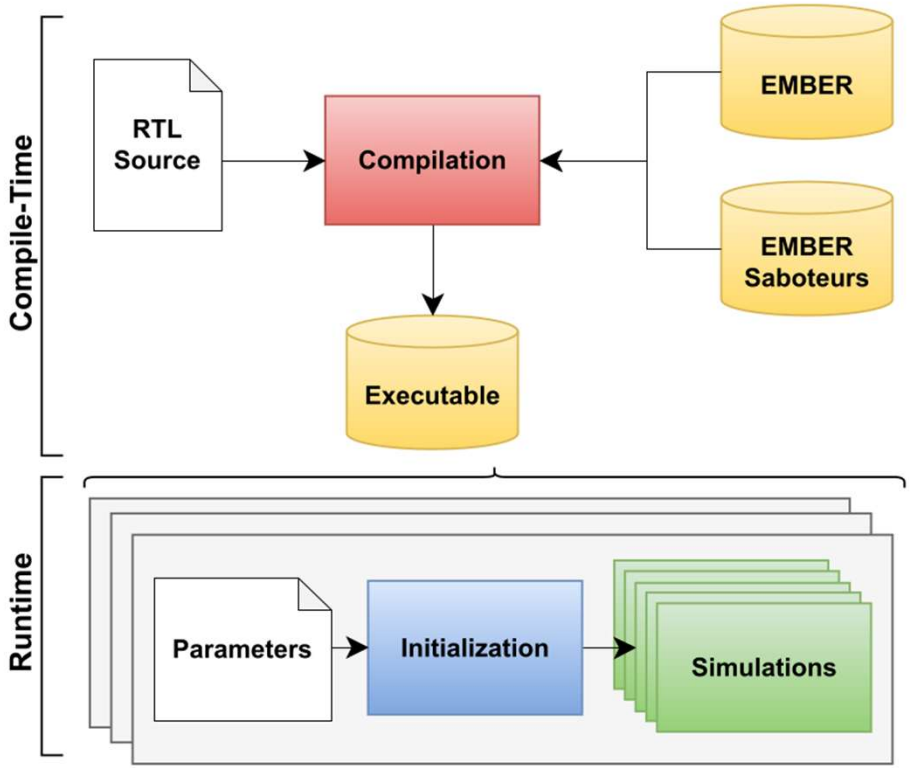
```
# Download the Demo repository (for VS Code UI)
# Otherwise visit my GitHub page (nickname: AlessandroVeronesi)
$ git clone git@github.com:AlessandroVeronesi/EMBER-Demo.git

# Download the Docker image
$ docker pull averonesi/ember-demo:latest
```

Designing Hardware with EMBER

EMBER adopts a similar workflow to other non-Verilog and non-VHDL based **hardware frameworks**:

- The **Hardware Module description** is **directly described in C++** using EMBER classes
- Also the **Saboteurs Library is described in C++** with EMBER classes
- The **Simulation Binary is compiled before** the initialization of modules' **parameters**, which are **initialized at runtime**



EMBER Modules

EMBER Hardware components are **subclasses of the abstract IModule class.** Thus, **EMBER hardware modules** have to **implement abstract methods** described in the IModule interface:

- **evaluate:** describes **combinational logic** behaviour;
- **update:** describes **clock-sensitive elements** behaviour;
- **reset:** describes sequential elements behaviour during the **reset** phase;
- **getSaboteurs:** returns a **list of saboteurs contained** in the **module** and in its **submodules**;
- **Class' Constructor:** contains all the **information to initialize the design with**, **including** subcomponent's **binding and submodules** hierarchical initialization.

```
1 class ember::IModule {
2  public:
3     virtual void          update() = 0;
4     virtual void            eval() = 0;
5     virtual std::vector<ISaboteur*>
6                         getSaboteurs() = 0;
7     virtual void           reset() = 0;
8 };
```

EMBER Saboteurs

EMBER saboteurs are **regular EMBER modules** which **additionally inherit the ISaboteur class**. **EMBER saboteurs are also EMBER modules.** Again, saboteurs description is achieved by methods overriding:

- **locations:** returns the number of physical locations in which the fault model can be injected;
- **getFaultMask:** generate a fault mask containing information about how to corrupt the signals in the EMBER module;
- **clearFaultMask:** clear the fault mask;
- **applyFault:** apply the fault mask;

```
1 template<ember::fault_t fmodel>
2 ember::ISaboteur {
3  public:
4     virtual const size_t
5                         locations() const = 0;
6     virtual void genFaultMask()         = 0;
7     virtual void clearFaultMask()       = 0;
8     virtual void applyFault()           = 0;
9 };
```

EMBER Ports

EMBER uses **ports to define components Inputs and Outputs.** In fact, EMBER ports are used to define **components connectivity**.

To use ports, there are some rules:

```
1 ember::inPort<ember::int8_t> myIPort;
2 ember::outPort<ember::int8_t> myOPort;
3
4 myIPort.bind();          // Self-Bound Port
5 myOPort.bind(myIPort); // Connection
6
7 myIPort.write(6);        // Writing to a Port
8 var = myOPort.read();   // Reading from a Port
```

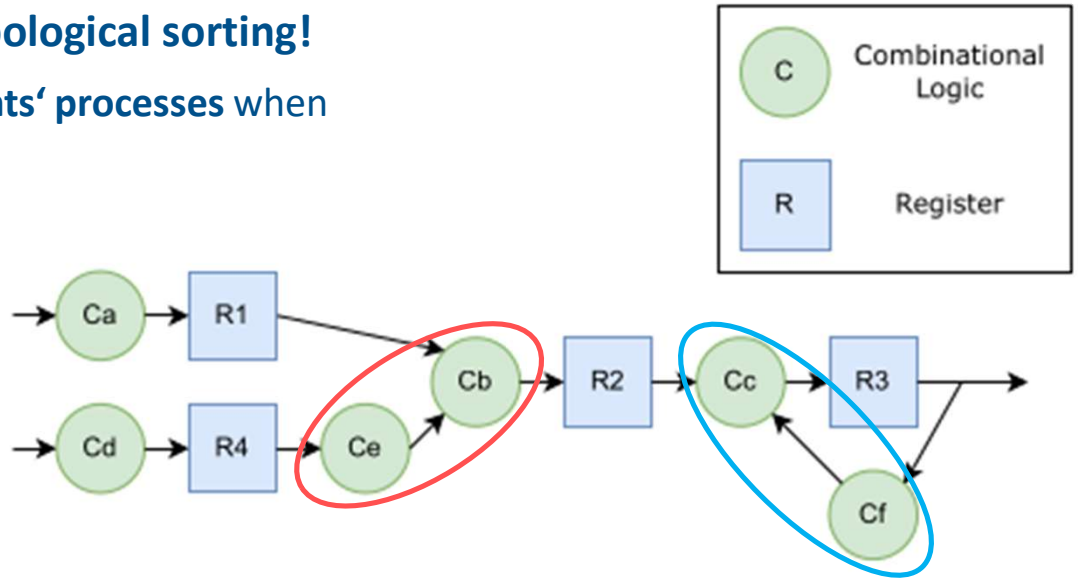
- Ports **need to be bound to each other** to connect components;
- **Ports can be self-bound** if not connected to anything else (or if their connectivity is later defined);
- **Ports can be written or read** to send/receive data;

EMBER Hierarchical Modules

Currently, **EMBER does not support automatic topological sorting!**

Designers are required to **manually schedule components' processes** when designing hierarchical modules.

```
1 void top::eval()
2 {
3   Cf.eval();
4   Cc.eval();
5   Ca.eval();
6   Ce.eval();
7   Cd.eval();
8   Cb.eval();
9 }
```



Starting the Demo

Start the Docker Container from VS Code:

1) **Open VS Code in the Repository‘s Top Level Directory**
2) **Open the Command Palette (shortcut: CTRL + SHIFT + p)**
3) Type **„Dev Containers: Reopen in Container“** in the Command Palette

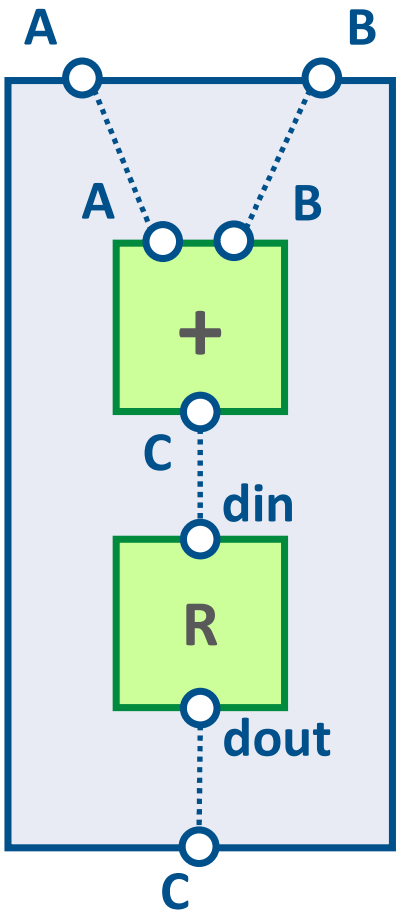
For Windows Users, remind to disable Wayland‘s Socket:

1) **Open Settings (shortcut: CTRL + ,)**
2) **Search and disable „Mount Wayland Socket“**

Exercise 1: Adder with a Register

Part 1: Implementing an Adder in EMBER

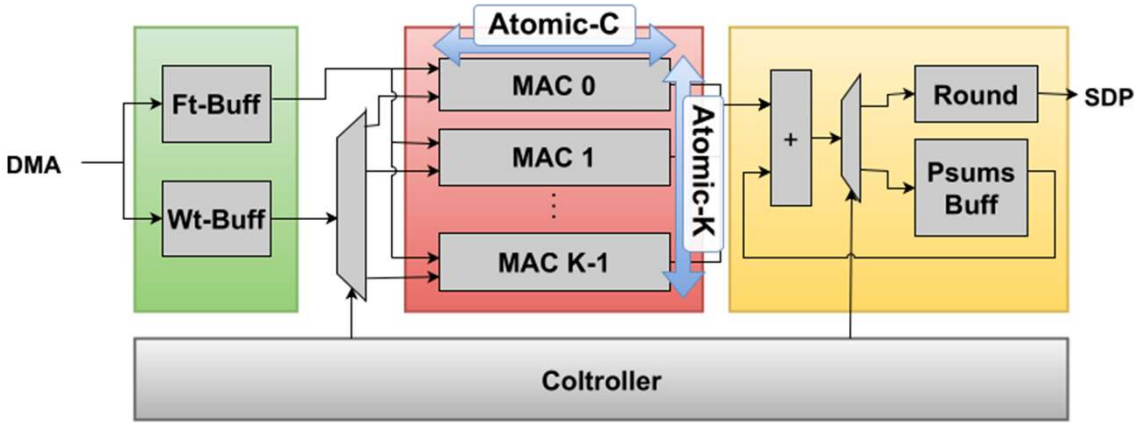
Part 2: Injecting SEUs in the Adder



Exercise 2: NVDLA Accelerators

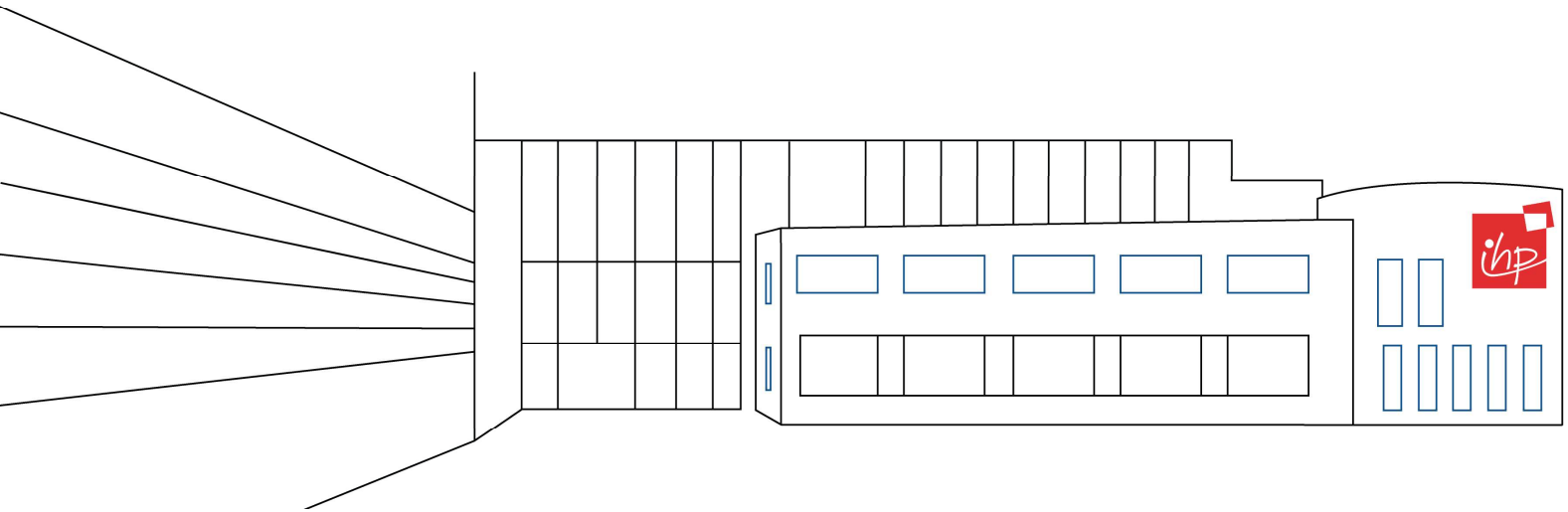
We deployed **EMBER** to model the **NVIDIA Deep Learning Accelerator (NVDLA)** and test its robustness agains **Single-Event Upsets (SEUs)**

- DNNs require **large fault injection campaigns** to draw statistically relevant error populations
- **SEU injection require cycle-accuracy** and **cannot be done in higher** hardware **abstractions**
- We injected SEUs in FFs for the **whole DLA pipeline**



Thank you for your attention!

IHP – Innovations for High Performance Microelectronics
Im Technologiepark 25
15236 Frankfurt (Oder)
Tel.: +49 (0) 335 5625 459
E-Mail: **veronesi@ihp-microelectronics.com**



Leibniz
Association