

Timur Kelin

# simSIMD

Development and Simulation  
Framework for Application Specific  
Vector Processor

Cambridge 2019



# Contents

- [Overview](#)
- [System Component: EU](#)
- [System Component: DM](#)
- [System Component: XBAR](#)
- [System Component: Streaming Unit](#)
- [System Component: Scalar Infrastructure](#)
- [Program Execution](#)



# Contents

- [“Design by Simulation” Strategy](#)
- [Hardware Considerations](#)
- [Structure of the Simulator Software](#)
- [Tests and Examples](#)
- [ToDo’s and Plans](#)
- [References](#)



# OVERVIEW



# Objectives

- Develop a framework and building blocks for the application specific vector processor, make simulation software.
  - Simplify the development of the hardware architecture which performs operations on the data structures of finite length (vectors)
    - signal processing - OFDM symbols, code blocks
    - cryptography - cipher blocks
    - networking - data packets



# Objectives

- Framework should provide unified approach for the development of the functional components: Execution Units (EU) and Data Memories (DM) as well as for their interoperation
  - Specific set and configuration of the functional units are identified by the target application



# Objectives

- Elaborate development and simulation methodology of the hardware architecture from system specifications (algorithms) and throughput requirements
- Follow top-down development and optimization strategy
  - Area: improvement of the utilization rate of the processing blocks and storage elements (RAMs)
  - Throughput: datapath is elaborated at the stage of the architecture development
  - Power
  - Non-recurring engineering (NRE) resources and risks



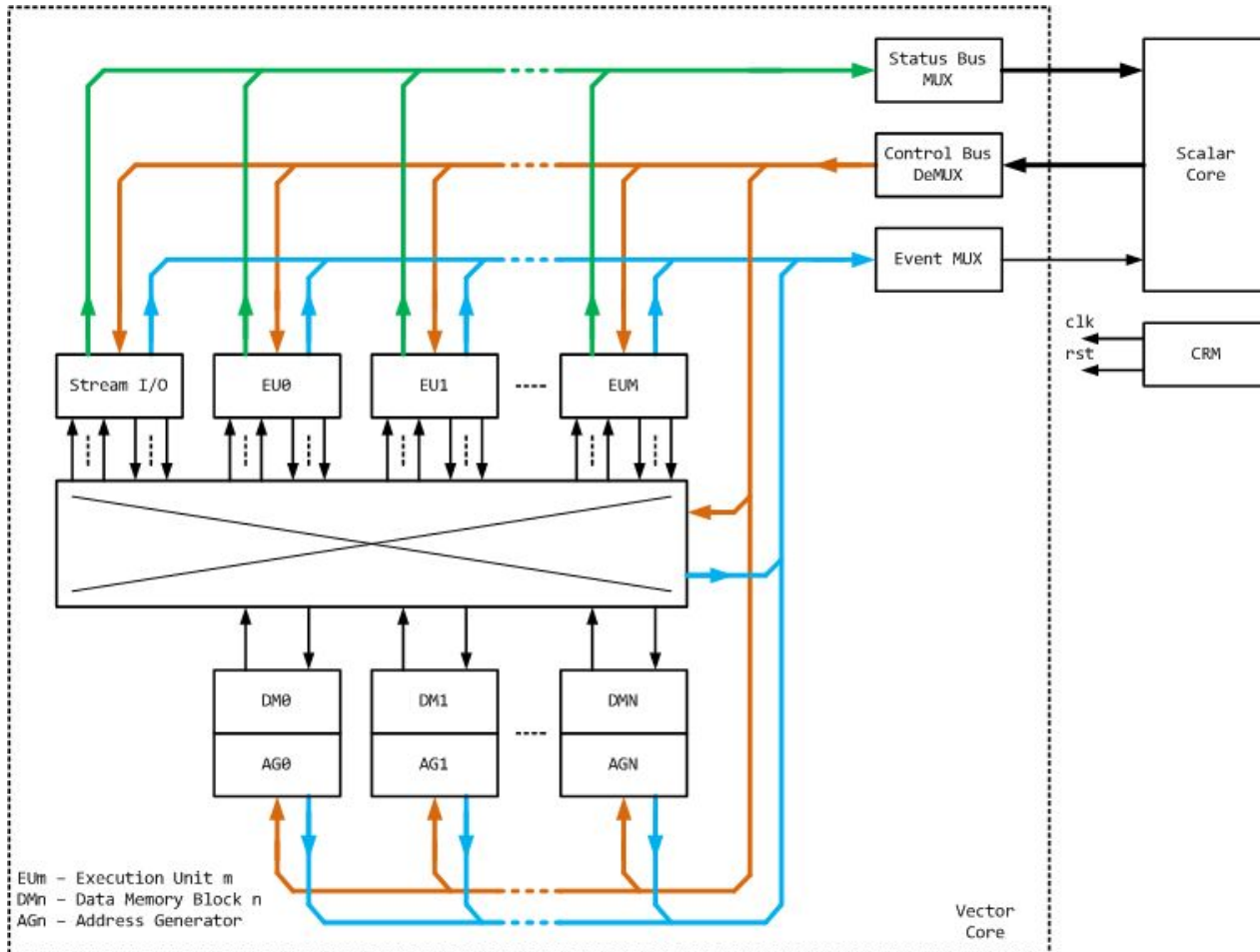
# Objectives

- Employ SystemC
  - Short simulation-analysis-update cycle which allows for simulation driven development and optimization
  - Cycle accurate simulation for the vector core to obtain realistic timing and throughput estimates at the earlier stages
  - High level of abstraction for Vector Core preferences, runtime configuration and status to stay focused on the architectural tasks
  - Large ecosystem of C/C++ libraries for data manipulation and processing allows for top-down development approach: from high level processing functions down to elementary arithmetic operations
  - Allows for integration into the existing simulation workflows
  - Open source





# Block Diagram



# Components in Brief

- Vector Core functions
  - Performing vector computations in accordance to the configuration supplied from the Scalar Core
  - Generation of the events at the different stages of the execution of the vector operations
    - Synchronization of the processing threads
  - Update of the statuses of the vector operations
    - Data dependent processing
  - Sequencing of the vector operations



# Components in Brief

- Vector Core components
  - Data Memories (DM): temporary storage of the vectors.
    - A set of Address Generators (AG) associated with each DM allows for flexible addressing/fetching of the elements of the vector.
  - Execution Units (EU): perform successive processing of the elements of the vectors.
    - This part of the Vector Core is specific for target application



# Components in Brief

- Streaming Devices: interfacing Vector Core with the external devices
  - ADC or DAC
  - Preliminary or subsequent processing blocks
  - Interface to the external storage (DMA)
- XBAR: Network-On-Chip (NOC) intended for routing of the vector streams between Execution Units (EU), Data Memories (DM) and/or Streaming Devices
  - DMs, EUs and Streaming Devices have a unified interface for connection to XBAR



# Components in Brief

- Functions of the Scalar Infrastructure
  - Respond to the events and statuses from the Vector Core components
  - Control the execution flow inside the Vector Core
  - Synchronization of the vector processing threads
  - Generate and deliver configuration data to the Vector Core components



# Components in Brief

- Scalar Infrastructure components
  - Scalar Core processes the events and statuses from Vector Core, and generates configurations for the components of the Vector Core.
    - It can be implemented as a programmable general purpose CPU subsystem or an FSM depending on the complexity of the control procedures
  - Event MUX: delivers events which were generated by the Vector Core components to the Scalar Core



# Components in Brief

- Scalar Infrastructure components (cont'd)
  - Config De-multiplexer: distributes commands and data supplied from to the Scalar Core to the Vector Core components.
    - Broadcasting commands are supported for the execution control
  - Status MUX: delivers the status data, which was sent by the Vector Core components in response to the request commands from the Scalar Core, to the Scalar Core



# Outline of the development strategy

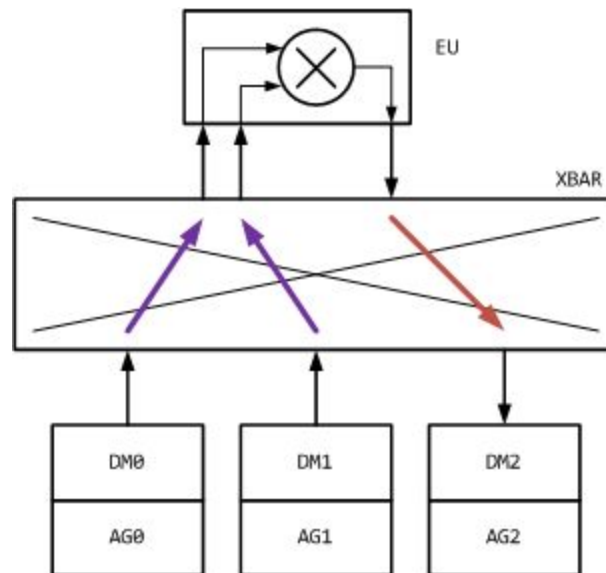
- Decompose the processing algorithm down to the level of functional blocks and storage elements
- Develop the functional blocks having the specified interface with XBAR and Scalar Infrastructure
- Assemble the functional blocks and storage elements to the framework
- Develop control FSM and/or CPU FW





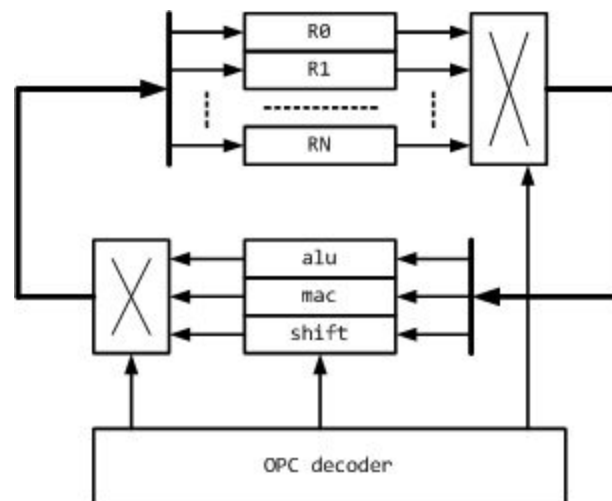
# Exemplar Data Flow

- Compute Hadamard product (element-wise multiplication) of 2 vectors  $\bar{C} = \bar{A} \otimes \bar{B}$ 
  - Elements of A reside in DM0
  - Elements of B reside in DM1
  - Elements of the product C are placed into DM2



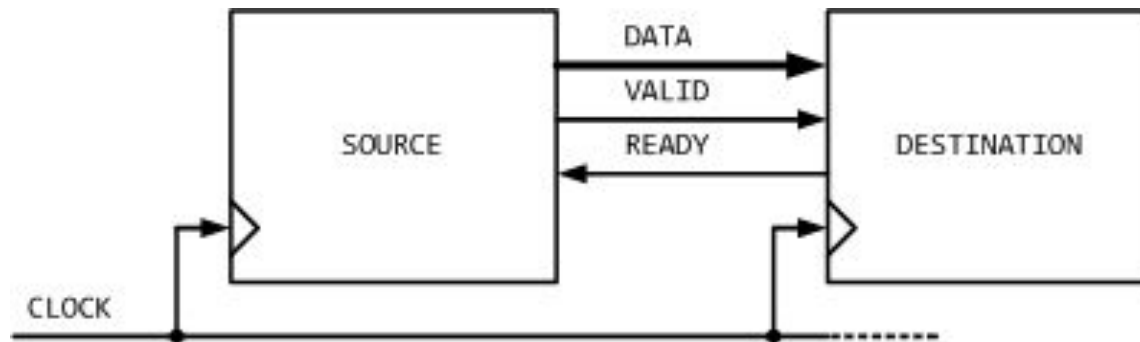
# Similarity to General Purpose CPUs

- Registers R0..RN in the register file (RF) are scalars
- Controls from OPCode Decoder
  - RF Output MUX
  - Processing Unit
  - RF Input MUX
- Parallel operation of the computational blocks [4]



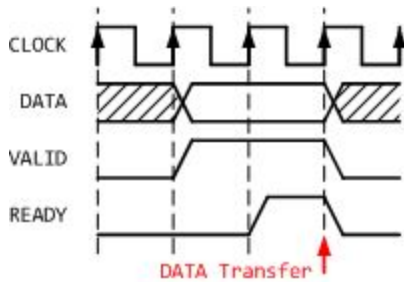
# VALID-READY Channel

- Decentralized coordination of the data transfers through the stages of the processing pipeline

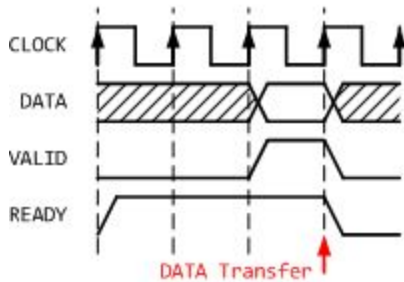


- Main rule: Data is transferred at the active clock edge when both VALID and READY are asserted
- SOURCE and DESTINATION can be stages of the processing pipeline or DM/EU modules

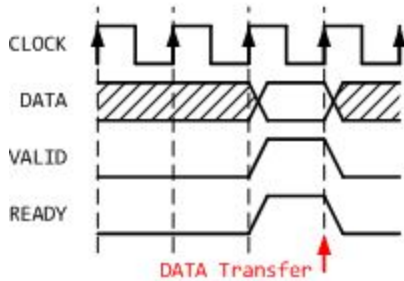
# VALID-READY Channel



VALID before READY handshake



READY before VALID handshake



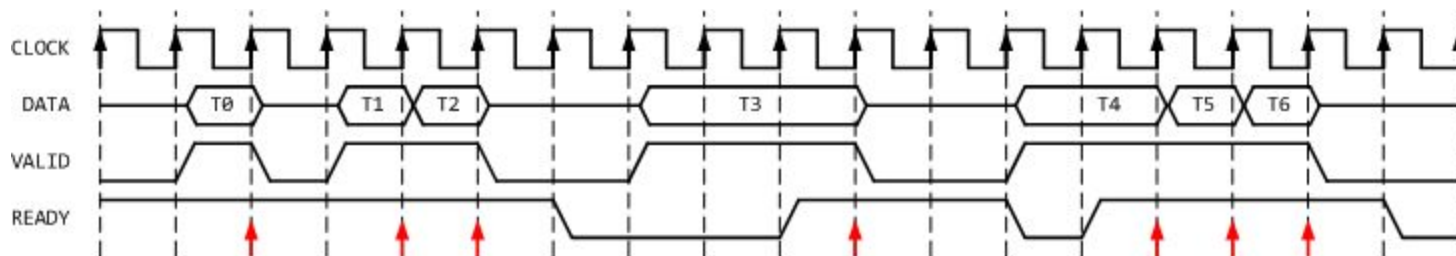
VALID with READY handshake

\*Refer to [2],[3] for more details



# VALID-READY Channel

- SOURCE retains the state of VALID and DATA signals until the transaction is acknowledged by the DESTINATION with its READY output asserted
- READY signal may change its state at any time. DESTINATION may implement a multi-cycle processing or use a shared resource which availability is changed from cycle to cycle



# VALID-READY Channel

- At the DM/EU interface DATA signal which is transferred in 1 clock cycle contains 4 data slots
- Contents of 1 slot
  - Flag which identifies that the data field is valid
  - Data field of type “complex double” representing 1 element of the vector or 1 sample
- Other application-specific data structures are possible e.g. fixed-point types, pixel colours, etc.



# VALID-READY Channel

- VALID signal is extended to 4 states (2 bits) to support of vectors transfers of finite length
- IDLE – Inactive state
- HEAD – first data transaction if the number of data transactions is more than 1
- BODY – intermediate data transaction if the number of data transactions is more than 2
- TAIL – last data transaction or data transaction with a single element
- Extended application-specific set of states is possible



# VALID-READY Channel

- In the simulator DM/EU modules are connected to the XBAR via VALID-READY sc\_channel.
- It monitors if both SOURCE and DESTINATION respect VALID-READY protocol.





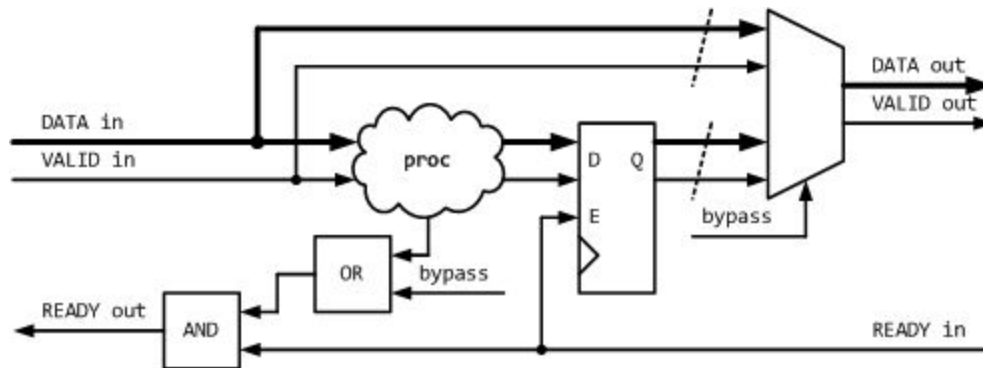
# VALID-READY Channel

- VALID output of the SOURCE must have no combinatorial dependency from its READY input. Otherwise a combinatorial loop is created
- READY output of the DESTINATION can be generated with combinatorial logic from its VALID and/or DATA inputs.



# VALID-READY Channel

- In RTL DATA and VALID outputs of the SOURCE stage should normally be the outputs of the registers
- It is acceptable to implement bypass mux after these registers if the DESTINATION is connected via the VALID-READY channel



# VALID-READY Channel

- Breaking combinatorial path of the READY signal is described in [2] Sections 2.1.2, 2.1.3

- SV outline:

```
always_ff @(posedge CLK or posedge RESET_IN)
  if(RESET_IN)
    READY_OUT <= 1;    // Ready on reset
  else begin
    if(CLEAR_IN) begin
      READY_OUT <= 1;  // Ready on reset
    end
    else if(READY_OUT) begin
      if(VALID_IN!=ST_IDLE & ~READY_IN)
        READY_OUT <= 0;
    end
    else begin
      if(READY_IN)
        READY_OUT <= 1;
    end
  end
end

always_ff @(posedge CLK) // no reset
  if(READY_OUT & (VALID_IN!=ST_IDLE & ~READY_IN)) begin
    DataBuffer <= DATA_IN;
    ValidBuffer <= VALID_IN;
  end
end

assign VALID_OUT = READY_OUT ? VALID_IN : ValidBuffer;
assign DATA_OUT = READY_OUT ? DATA_IN : DataBuffer;
```



# VALID-READY Channel

- When merging streams, a processing stage should rely on all the upstream SOURCES to generate VALID output and wait for an acknowledge from the DESTINATION stage
  - Time alignment buffers can be used to supply vector elements from all the SOURCES at a single clock cycle
- There are 2 ways of splitting streams:
  - DESTINATION which stalls is nominated to be a master and VALID signal to other DESTINATION s is AND'ed with the READY signal from the master DESTINATION.
    - For DMs and EUs this feature is implemented as a function of XBAR.
  - Supply VALID signal and process READY signal individually for each DESTINATION, latching the acknowledges for the given transaction.
    - This requires dedicated source ports in the DM/EU interface.



# VALID-READY Channel

- To constrain combinational paths through the XBAR, both input and output signals of VALID, DATA and READY in DM or EU should be constrained.
- Full-bandwidth elastic buffers described in [2] Section 2.1.2 can be used from both input and output sides.
  - Elastic buffer should be extended to support 4-state VALID signal
- These buffers add to the datapath a delay of 2 clock cycles per single DM or EU.



# SYSTEM COMPONENT: EU



# SYSTEM COMPONENT: DM



# SYSTEM COMPONENT: XBAR





# **SYSTEM COMPONENT: STREAMING UNIT**



# **SYSTEM COMPONENT: SCALAR INFRASTRUCTURE**



# PROGRAM EXECUTION



# General Considerations

- Efficient operation of the Vector Core from the perspective of the utilization rate and throughput:
  - Vectors are processed back to back, and
  - Multiple processing chains or threads run concurrently
- The configuration of the Vector Core should be placed inside the corresponding components before the processing of the new vector starts.
  - Pick up new configuration upon the completion of the current vector
  - Start the processing of the new vector without involving Scalar Infrastructure and thus less delay



# General Considerations

- Utilization rate and throughput of the Vector Core set the requirements on:
  - The partitioning of the target processing task
  - The set of the Vector Core components
  - The throughput and the performance of the Scalar Infrastructure



# Synchronization between Cores: Scalar Core to Vector Core

- The processing in the Vector Core is managed with the commands from the Scalar Core
  - **put** configuration to a component of the Vector Core
  - **get** status of a component of the Vector Core
  - **run** configuration with `exec_id` which is broadcasted into all the components of the Vector Core



# Synchronization between Cores: Scalar Core to Vector Core

- Configuration
  - Each Vector Core component can have a number of slots to store the configuration.
- Compulsory fields in a configuration slot
  - `exec_id`: the configuration becomes active when the block receives "run" command with the matching `exec_id`. The block picks up the configuration from the slot and starts its execution.
  - `status_slot`: the execution status is updated at the slot which is pointed by `status_slot`
    - This field is not applicable for XBAR
  - `events`: the block issues specific events during the execution of the configuration.
    - This field is not applicable for XBAR



# Synchronization between Cores: Scalar Core to Vector Core

- Compulsory fields in a configuration slot (cont'd)
  - `config_next`: when the execution of the current configuration is complete i.e.
    - DM/EU received or transmitted vector TAIL
    - XBAR identified that all the blocks have completed their processing

the block picks up the configuration from the slot which is pointed by `config_next` and starts its execution.

- This forms a processing chain
- Invalid or inexistent `config_next` completes the chain
- Looped chains are possible





# Synchronization between Cores: Vector Core to Scalar Core

- Status
  - EU components can have a number of slots to store the results of the vector processing.
  - Can reflect runtime state of the component.
    - This state is visible to Scalar Core
  - Data fields inside the status slot are specific to a particular block
- Events
  - Components of the Vector Core can notify Scalar Core on the progress of the execution of the configuration.
  - Full set of events is specific to DM/EU.



# Synchronization between Cores: Vector Core to Scalar Core

- Events (cont'd)
  - The subset of events to be issued for the vector being processed is selected in the configuration
  - XBAR can issue an event when all the connected components are finished the processing of the vector.
    - This is useful when the components are executing looped chains
  - In the simulator events are implemented with transfers through the FIFO channels with polling and readout on the side of the Scalar Core.



# Execution Model

Command sequencing under the control of the Scalar Core

- Scalar Core configures DMs and EUs, which are employed in the processing chains
  - Multiple processing chains can run concurrently
- Scalar Core configures routing through the XBAR
- Same `exec_id` is programmed for all the DMs, EUs and XBAR configuration
- Scalar Core issues **run** command with the specific `exec_id`



# Execution Model

Command sequencing under the control of the Scalar Core

- Scalar Core waits for the TAIL event from the blocks which are employed in the processing chain
- Scalar Core read and process blocks' status
- Scalar Core issues **run** command for the new chain which has been previously configured
- Repeat
- This behavior is tested in vri\_test1..3



# Execution Model

## Command sequencing under the control of the Vector Core

- Configuration Chaining
  - Cycle stationary (data independent) execution can be handled by the Vector Core without the need for the intervention from the Scalar Core
- Events from DMs/EUs/XBAR can be issued in the process of the chain execution to notify the Scalar Core.
  - The status of the blocks, which have completed the processing, can be read out.
  - Any of the configuration slots of the blocks, which completed the processing, can be reconfigured
  - The blocks, which completed the processing, can execute another configuration
- FFT with 1 EU and 2 DMs for ping-pong is a good application example of this mode
- This behavior is reflected in vri\_test4



# Execution model: Example of the Configuration Chaining

Succession of the operations in the processing chain:

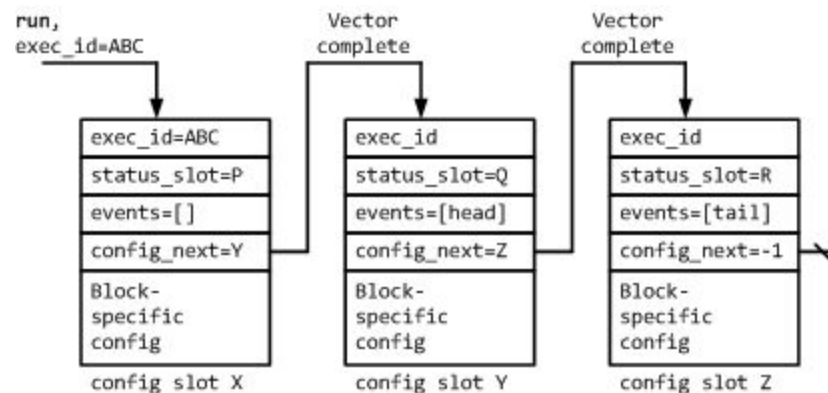
Processing chain is initiated when the block receives run command with `exec_id=ABC`

Configuration slot X becomes active. A vector is processed. Block status is updated in the slot P. No events are generated.

Configuration slot Y becomes active. A vector is processed. Block status is updated in the slot Q. Event is generated when vector HEAD marker is received.

Configuration slot Z becomes active. A vector is processed. Block status is updated in the slot R. Event is generated when vector TAIL marker is received.

The processing chain is complete.



# Execution Model

## Deferred Execution

- Deferred Execution allows writing configuration and issuing **run** command to DMs, EUs and XBAR while they are busy with the processing.
  - This relaxes performance constraints on Scalar Core
  - Configuration chaining prevails over the deferred execution
- DM or EU operation
  - If DM or EU is busy with processing a chain at the moment they receive **run** command, then the valid `exec_id` is placed into the FIFO buffer to be processed after the chain execution is complete.
  - This behavior is reflected in `vri_test6`



# Execution Model

## Deferred Execution

- XBAR
  - If XBAR receives **run** command for a configuration which connects DMs/EUs and at least one of them is busy, then it postpones the execution until all of the DMs and EUs for the requested configuration become idle.
  - This behavior is reflected in vri\_test5, 6
  - Request for changing or running a configuration which is already active is illegal.





# Execution Model

## Data Exchange and Data Dependent Execution

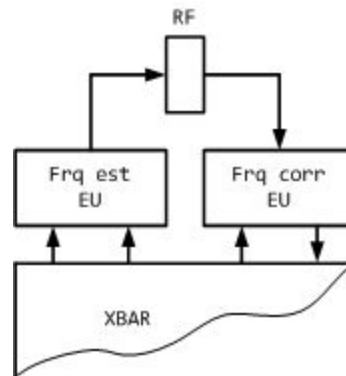
- Components of the Vector Core can exchange small amounts of data on their own without involving Scalar Core or communication through XBAR
  - This data can be used as a Processing Parameter, or
  - A configuration pointer and/or execution trigger
- Simple data transfers should not involve Scalar Core
  - Relaxes throughput constraints for Scalar Infrastructure
- This exchange can be done via common register files or FIFOs on the top of the EUs
  - HW semaphores
- Currently this feature is outside of the scope of the simulation framework but it can be implemented as a specific feature of the simulated application.



# Execution Model

## Data Exchange and Data Dependent Execution

- Exemplar task: frequency offset estimation in one chain and frequency offset correction in another
  - Estimation task processes a vector and produces a resulting scalar value
  - Estimation task upon the completion sends its result to the correction task
  - Correction task uses the scalar value in its processing



# **“DESIGN BY SIMULATION” STRATEGY**



# HARDWARE CONSIDERATIONS



# **STRUCTURE OF THE SIMULATOR SOFTWARE**



# TESTS AND EXAMPLES



# vri\_test

- Verifies system integration and overall functionality
  - Vector transfers
  - Block configuration, status and events transfers
- Verifies operation of VALID-READY interface and XBAR functionality
- Verifies execution modes and command sequencing
  - Under the control of the Scalar Core
  - Under the control of the Vector Core
- vrisrc: Streaming block which simulates VALID-READY source
  - Configurable random VALID or always VALID
  - Transmits random DATA with checksum at TAIL
- vridst: Streaming block which simulates VALID-READY destination
  - Configurable random READY or always READY
  - Receives DATA and verifies the checksum
- Assertions inside VALID-READY channel
- Inspection of the VCD trace
- Tests 01..07 run continuously in the random order for 1 ms



# vri\_test

- Clean

```
make EXAMPLE=basic/vri_test clean
```

- Build

```
make EXAMPLE=basic/vri_test all
```

- Run

```
./build/Release/out/simsimd
```

- Inspect the result:

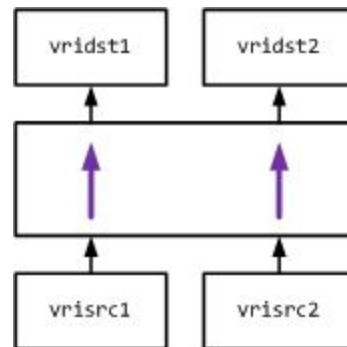
```
In gtkwave File->Open New Window->trace.vcd
```





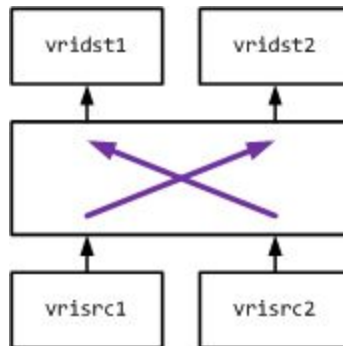
# vri\_test / test01

- Verifies basic operation
- Scalar core configures vector core components
  - src1->dst1, src2->dst2
- Scalar core initiates 2 concurrent vector transfers
- Wait for transfer completion by polling the events from XBAR and EUs



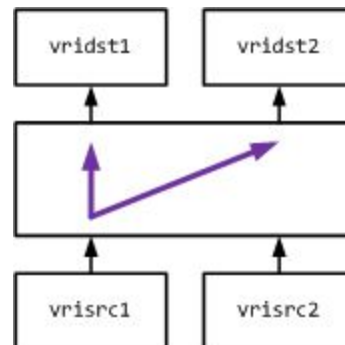
# vri\_test / test02

- Verifies operation of the “always ready” destination
- Scalar core configures vector core components
  - src1->dst2 with READY=1, src2->dst1
- Scalar core initiates 2 concurrent vector transfers.
- Wait for transfer completion by polling the events from XBAR and EUs



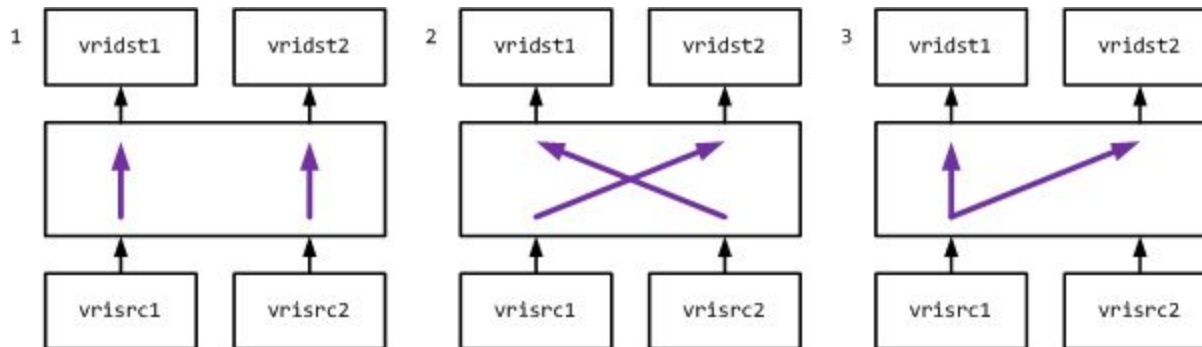
# vri\_test / test03

- Verifies data multicasting
- Scalar core configures vector core components
  - src1->dst1, dst2. dst1 set as master, dst2 has READY=1
- Scalar core initiates 2 concurrent vector transfers.
- Wait for transfer completion by polling the events from XBAR and EUs



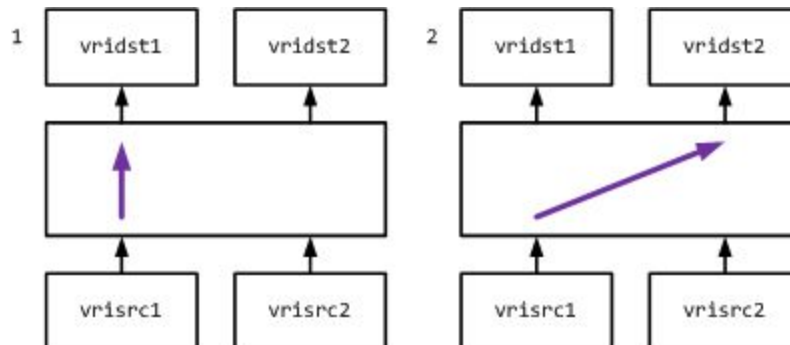
# vri\_test / test04

- Verifies automatic stepping through the EU and XBAR configuration slots
- Scalar core configures vector core components to execute 3 configuration slots in a succession.
  1. src1->dst1, src2->dst2
  2. src1->dst2, src2->dst1
  3. src1->dst1, dst2. dst1 set as master, dst2 has READY=1
- Scalar core initiates vector transfers which correspond to slot 1.
- Scalar core waits for the completion of the transfers which correspond to slot 3 by polling the corresponding events



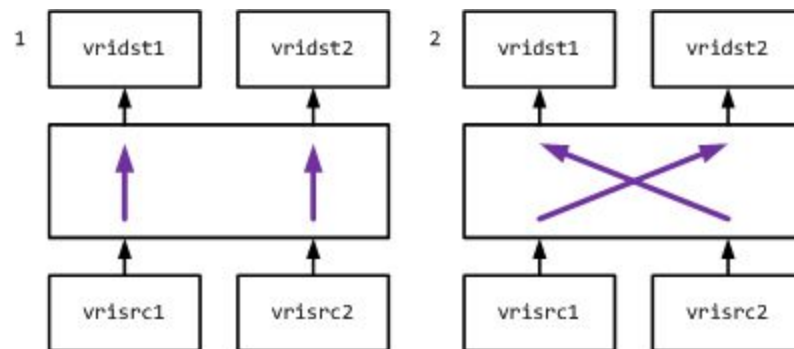
# vri\_test / test05

- Verifies deferred execution in XBAR
- Scalar core configures vector core components in slot 1
  - src1->dst1
- Scalar core initiates vector transfers which correspond to slot1.
- Without waiting for completion Scalar core configures vector core components in slot 2
  - src1->dst2
- Scalar core initiates vector transfers which correspond to slot 2. Vector core defers the execution of slot 2 until slot 1 is complete and src1 becomes available.
- Scalar core waits for the completion of the transfers which correspond to slot 2.



# vri\_test / test06

- Verifies deferred execution in XBAR and EUs
- Scalar core configures vector core components in slot 1
  - src1->dst1
  - src2->dst2
- Scalar core initiates vector transfers which correspond to slot1.
- Without waiting for completion Scalar core configures vector core components in slot 2
  - src1->dst2
  - src2->dst1
- Scalar core initiates vector transfers which correspond to slot 2. Vector core defers the execution of slot 2 until slot 1 is complete and EUs become available.
- Scalar core waits for the completion of the transfers which correspond to slot 2.



# vri\_test / test07

- Verifies execution priorities
- Scalar core configures vector core components to execute configuration slots 1 and 2 in a succession.
  1. src1->dst1
  2. src1->dst2
- Scalar core initiates vector transfers which correspond to slot 1.
- Without waiting for completion Scalar core configures vector core components in slot 3
  - src1->dst1
- Scalar core initiates vector transfers which correspond to slot 3. Vector core defers the execution of slot 3 until slots 1 and 2 are complete and EUs become available.
- The resulting succession of transfers:
  1. src1->dst1
  2. src1->dst2
  3. src1->dst1



# dm2dm

- Verifies operation of the DM RAM blocks
  - dm\_ram\_1rw – Single port RAM with non-simultaneous read and write operations from a single address
  - dm\_ram\_1r1w – simple dual-port RAM with simultaneous one read and one write operations to different locations
- The test checks
  - Integration into the vector core structure
  - Operational modes of address generator
  - AG register and configuration
- Test runs continuously with the random vector sizes and VRI parameters for 1 ms





# dm2dm

- Clean

```
make EXAMPLE=basic/dm2dm clean
```

- Build

```
make EXAMPLE=basic/dm2dm all
```

- Run

```
./build/Release/out/simsimd
```

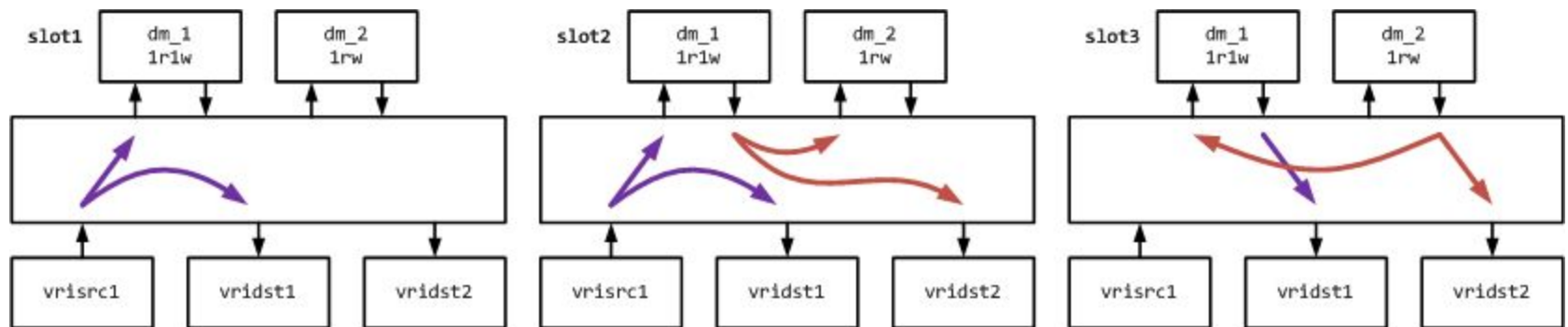
- Inspect the result:

```
In gtkwave File->Open New Window->trace.vcd
```



# dm2dm

- Scalar core configures vector core components to execute configuration slots 1..3 in a succession.
  1. src1->dm1 (lower half), dst1
  2. src1->dm1 (upper half), dst1; dm1 (lower half)->dm2, dst2
  3. dm1 (upper half)->dst1; dm2-> dm1 (lower half),dm2
- Scalar core initiates vector transfers which correspond to slot1.



# dm\_init

- Verifies initialization of the DM block from .mat file
  - .mat file is generated with Matlab
- The test checks
  - Integration of matIO library
  - DM initialization functionality
- Test runs continuously with the random VRI parameters for 100 us



# dm\_init

- Clean

`make EXAMPLE=basic/dm_init clean`

- Build

`make EXAMPLE=basic/dm_init all`

- Generate initialization .mat file

Execute `./examples/basic/dm/init/mat/dm_init.m`

`dm_init.mat` should be created in the same directory

- Run

`./build/Release/out/simsimd`

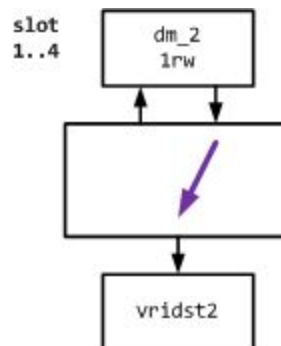
- Inspect the result:

In `gtkwave` `File->Open New Window->trace.vcd`



# dm\_init

- 4 regions of dm1 block are initialized from the file at before the simulation starts
  - Initialize with the data accepted by vridst
- Scalar core configures vector core components to configuration slots 1..3 in a succession.
  1. dm1-> dst1, Initialized region 1
  2. dm1-> dst1, Initialized region 2
  3. dm1-> dst1, Initialized region 3
  4. dm1-> dst1, Initialized region 4
- Scalar core initiates vector transfers which correspond to slot1.



# transp

- Verifies operation of the Transparent EU blocks
  - Synchronous input-to-output transfer: via the register
  - Asynchronous input-to-output transfer: wires
- The test checks
  - Integration into the vector core structure
  - Operational of the basic EU block
  - Asynchronous operation inside EU
- Test runs continuously with the random vector sizes and VRI parameters for 100 us



# transp

- Clean

```
make EXAMPLE=basic/transp clean
```

- Build

```
make EXAMPLE=basic/transp all
```

- Run

```
./build/Release/out/simsimd
```

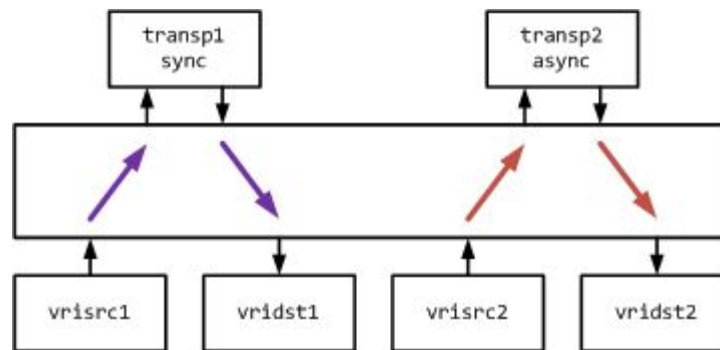
- Inspect the result:

```
In gtkwave File->Open New Window->trace.vcd
```



# transp

- Scalar core configures vector core components to execute single configuration.
  - src1->transp1->dst1 (synchronously transparent EU)
  - src2->transp2->dst2 (asynchronously transparent EU)
- Scalar core initiates vector transfers which correspond to slot1.





# TODO'S AND PLANS



# Runtime Statistics

- For the specific application collect runtime usage data for the resources which were allocated in the preferences
  - Access to DMs and EUs. Are they actually used?
  - Which blocks need supporting command sequencing under the control of the Vector Core
  - Range of the execution indexes
  - Internals of the DMs and EUs:
    - Configuration and status slots , bitwidth of the fields
    - Execution modes
    - Depth of the FIFO for the execution indexes
    - EU operational modes
    - DM AG modes
    - Events which were issued and processed



# Runtime Statistics

- Logging of the of the XBAR
  - Switching matrix
    - For the particular application not all of the switching routes are used.
    - Matrixes for data/valid and for ready can be different if data transfers to multiple destinations are used
  - Configuration slots, bitwidth of the fields
  - Execution modes
  - Events which were issued and processed



# RTL Code and Testbench Generator

- Generate RTL code on the basis of the initial preferences and runtime statistics
  - Generate vector core and parameters for XBAR, DMs and EUs
- Use VRI interfaces or modules
  - Data injection
  - Data pickup
  - VRI protocol assertions
- Generate testbench
  - Verifies system integration of the vector core



# Common Pool of Address Generators for DMs

- Additional cross-bar switch (AG → 4way DMs) can make the area savings negligible



# Debug Interface for the Vector Core

- Debug Interface for the Vector Core



# Data Dump

- Data Dump



# FXP

- FXP





# REFERENCES



# Source Code and Documentation

<https://github.com/timurkelin/simsimd>



# Books, Papers and Presentations

- [1] Bridging dream and reality: Programmable baseband processors for software-defined radio, D.Liu, A.Nilsson, E.Tell, D.Wu, J.Eilert  
IEEE Communications Magazine 47 (9), 134-140
- [2] Microarchitecture of Network-on-Chip Routers, A Designer's Perspective  
Dimitrakopoulos G.; Psarras A.; Seitanidis I.  
2015, 175p., Springer



# Standards and Datasheets

[3] AMBA 4 AXI4-Stream Protocol Specification,  
Version: 1.0, (c) 2010 ARM

[4] SHARC Processor Programming Reference,  
Revision 2.2, (c) 2013 Analog Devices, Inc.



# Patents (for reference only)

- [EP2751670B1: Digital signal processor](#)
- [EP2751671B1: Digital signal processor and baseband communication device](#)
- [US20060271764A1: Programmable digital signal processor including a clustered SIMD microarchitecture configured to execute complex vector instructions](#)
- [US20060271765A1: Digital signal processor including a programmable network](#)
- [US20070198815A1: Programmable digital signal processor having a clustered SIMD microarchitecture including a complex short multiplier and an independent vector load unit](#)
- [US20140244970A1: Digital signal processor and baseband communication device](#)
- [US20140281373A1: Digital signal processor and baseband communication device](#)
- [US20140344549A1: Digital signal processor and baseband communication device](#)
- [US20140351555A1: Digital signal processor and method for addressing a memory in a digital signal processor](#)
- [US20140359252A1: Digital signal processor](#)
- [US20140372728A1: Vector execution unit for digital signal processor](#)
- [US7299342B2: Complex vector executing clustered SIMD micro-architecture DSP with accelerator coupled complex ALU paths each further including short multiplier/accumulator using two's complement](#)
- [US7415595B2: Data processing without processor core intervention by chain of accelerators selectively coupled by programmable interconnect network and to memory](#)
- [US8874968B1: Method and system for testing a processor designed by a configurator](#)
- [US9557996B2: Digital signal processor and method for addressing a memory in a digital signal processor](#)

