EFCL Winter School – Track 2
Customizing RISC-V Based Microcontrollers

Lecture 3 – Integrating cooperative HW Processing Engines / HWPEs

Yvan Tortorella - yvan.tortorella@chips.it

Solution for XIFU Exercise

Chips-IT

- FIR XIFU RTL: https://github.com/pulp-platform/fir-xifu/tree/main
- FIR XIFU SW: https://github.com/FondazioneChipsIT/regression_tests/tree/EfclSolutions -> fir_xifu_minimal & fir_xifu_complete

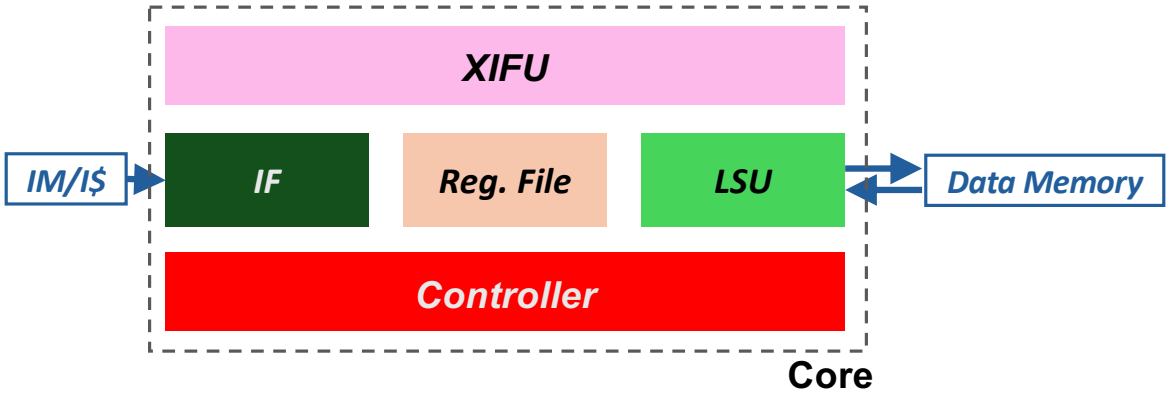
Overview of the course

Chips-IT

- **PULP platform & PULPissimo microcontroller architecture**
- **Extending RISC-V cores with dedicated custom instructions**
- **Integrating cooperative HW Processing Engines / HWPEs**
  - *Lecture*: Why should an ISA extension not be enough? PULP cooperative HWPEs and loosely-coupled accelerators
  - *Hands-on*: Guided design of a FIR filter HWPE
  - *Hands-on*: Integration of the FIR HWPE in the PULPissimo architecture
  - *Hands-on*: Evaluating acceleration in the PULPissimo environment
- **Testing extended PULPissimo on FPGA**

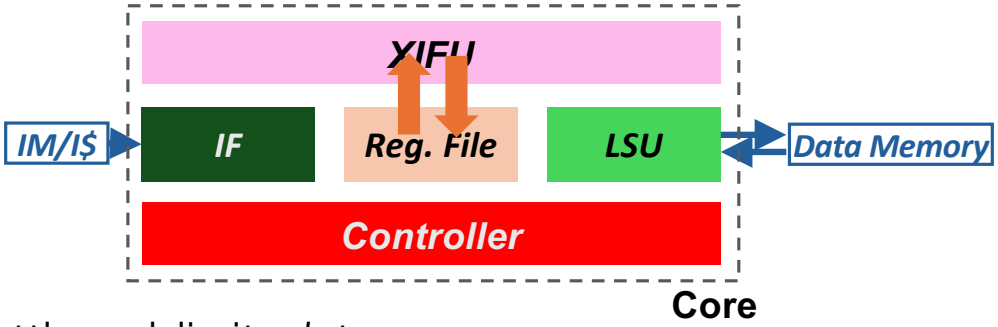
Yesterday we design a cool ISA extension …

- **… should the course be over yet?**
  - Well, that is a rhetorical question ;)
  - Let us consider a core + ISA extension, e.g., realized with CV32E40X + a XIFU like yesterday's
  - We will consider a few bottlenecks and the related workarounds



1) The register file bottleneck

- **Arithmetic instructions are primarily Reg-Reg**
  - Reg-Reg instructions are bottlenecked by the **architectural register file**
    - limited **size**, i.e., the number of architecturally exposed registers is small (=32 in RISC-V)
    - limited **bandwidth**, generally 2 or 3 read ports + 1 write port

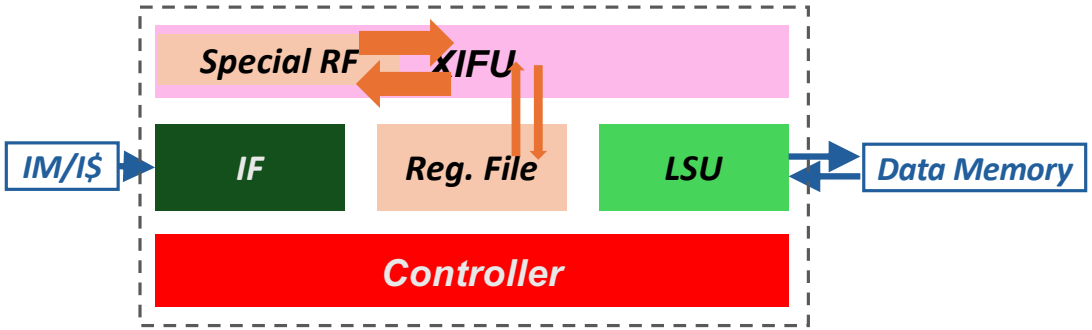


  - The register file bottleneck limits *data reuse*...
    - Circulate often data between mem and RF, killing performance and requiring much L1 bandwidth
    - Register File is a shared resource with all other instructions! It is a scarce resource

1) The register file bottleneck – a workaround

Chips-IT

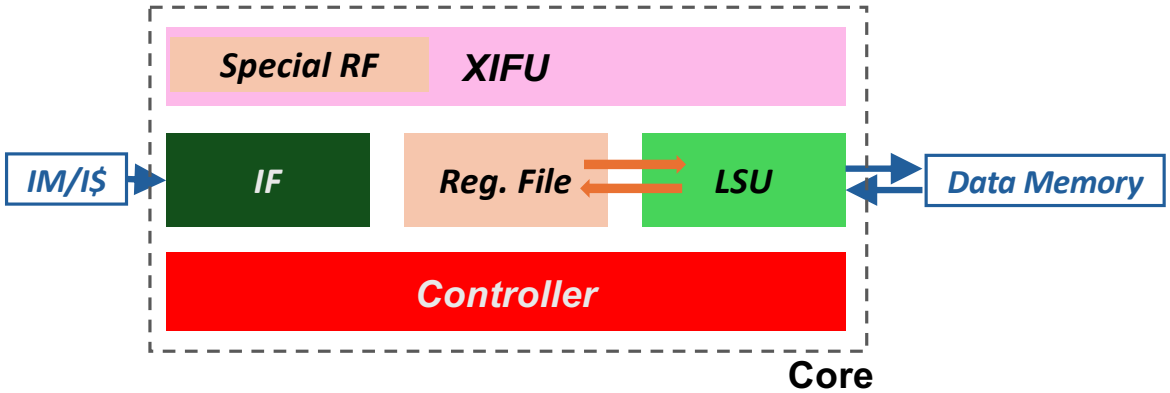
- **Internalizing the register file alleviates the bottleneck**
  - Internal register file is not shared with other SW, and it can be arbitrarily large
    - Data reuse is limited only by the overall size, which can be made much > 32 elements
    - RF can be organized in different buffers, delivering higher bandwidth
  - We actually used this trick in yesterday's XIFU!



  - Is this the **silver bullet**? Not really...
    - Less flexibility: Special RF is by definition less architecturally visible than architectural RF 😐
    - All data ultimately will be bottlenecked by LSU and memory access! ☹

2) The LSU bottleneck

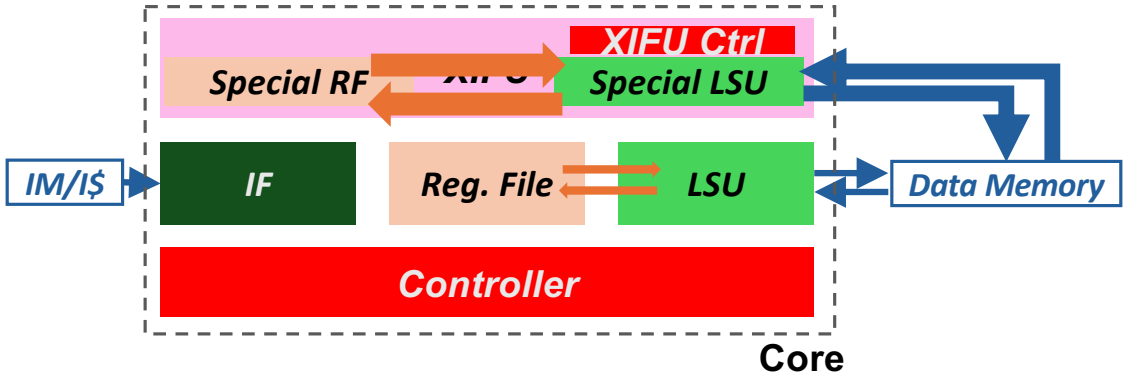
- **The core's LSU has limited bandwidth and it is shared**
  - Maximum 32bit/cycle on a RV32IM core
    - This is assuming we issue 1 load/store instruction per cycle



  - Limited L1 memory bandwidth disables any scheme providing large acceleration
    - Check *roofline plot* – we are often in L1 memory-bound conditions

2) The LSU bottleneck – a workaround

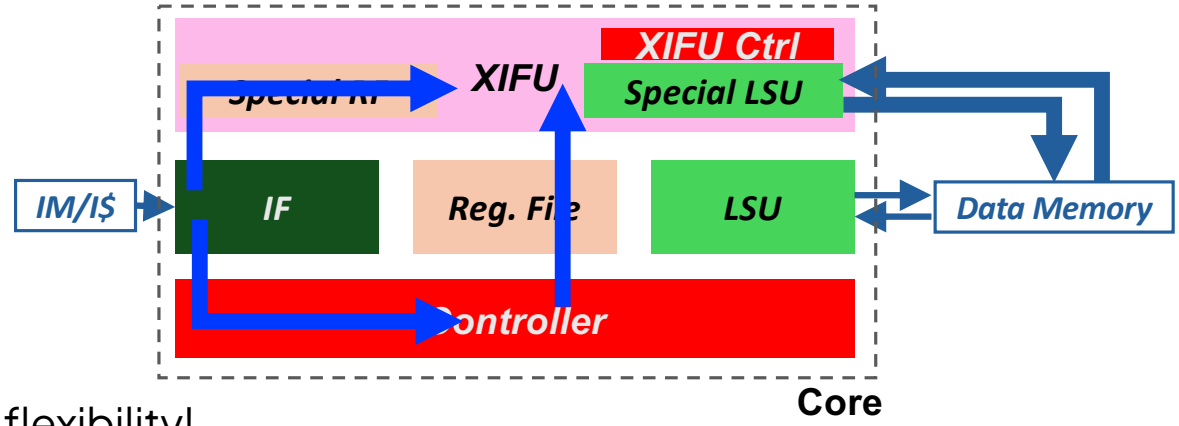
- **Provide the co-processor or ISA extension with its own LSU**
  - Can provide the required bandwidth (at a cost)
  - Does not occupy the core LSU
    - LD/ST on extension & main core can proceed in parallel
    - We need a small controller to control the extension LSU



  - Can create memory consistency problems if used in a very "tightly coupled" way
    - Especially if extension operation lasts more cycles than the core pipeline stages

3) The Von Neumann bottleneck

- **An ISA extension still executes *instructions*, leading to a control bottleneck**
  - It needs to **fetch**/**decode**/**execute** ("Von Neumann bottleneck")
    - Keeps the core occupied with "menial" control tasks
    - Either relax "RISC" assumption or accept very fine-grain computation (more difficult to accelerate)

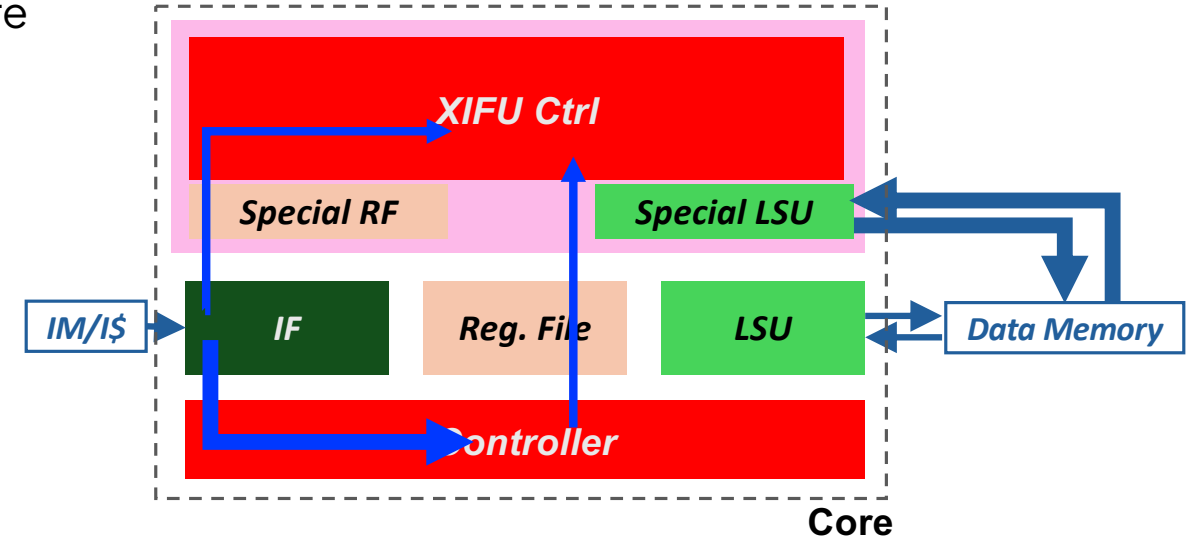


  - The price of flexibility!

3) The Von Neumann bottleneck – a workaround?

Chips-IT

- **Delegate more control to the extension itself**
  - Coarser grain, less RISC-like "megainstructions" managed almost independently from the main core

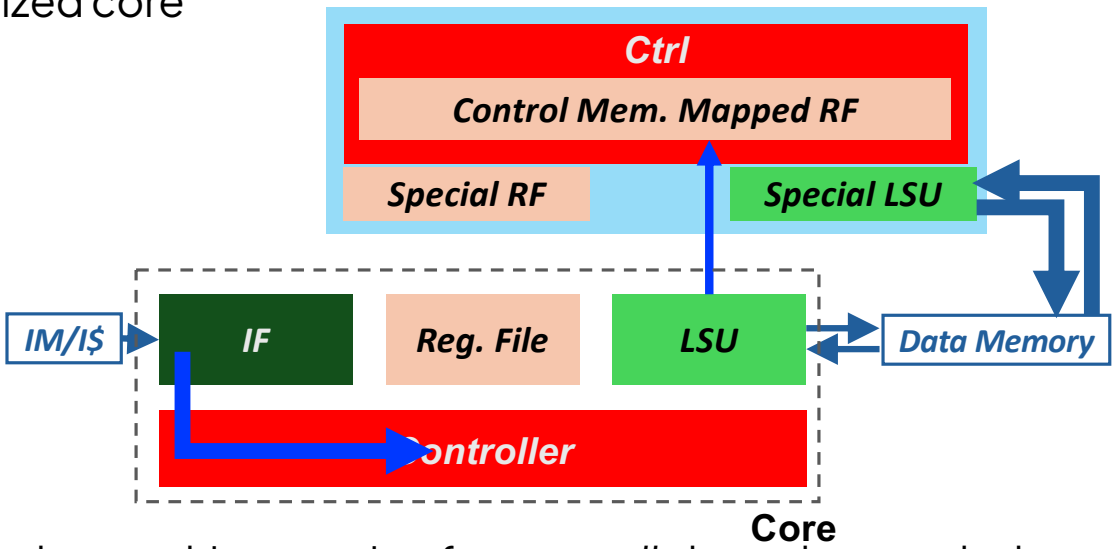


- **Privatized local register file, LSU, controller:**
  - Is this still an ISA extension at all?

Loosening the core-coupling

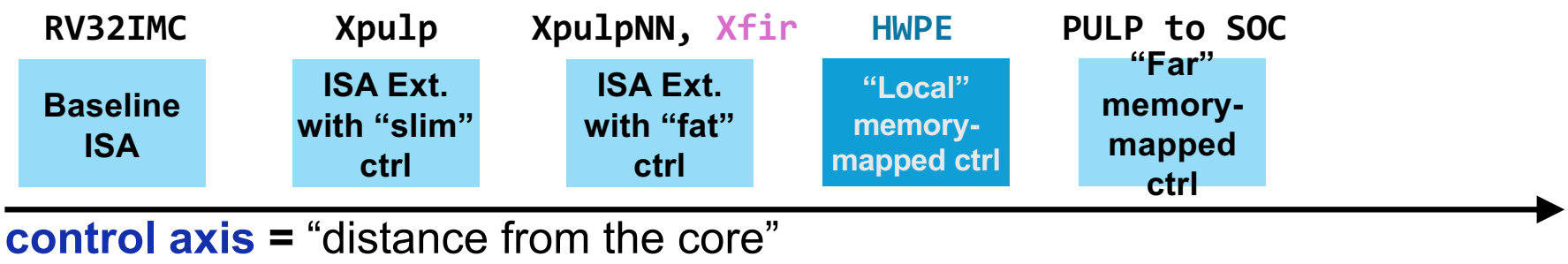
Chips-IT

- **Move the only remaining coupling (controller) to memory-mapped control**
  - Not really a coprocessor anymore, but something else entirely: an independent engine / specialized core

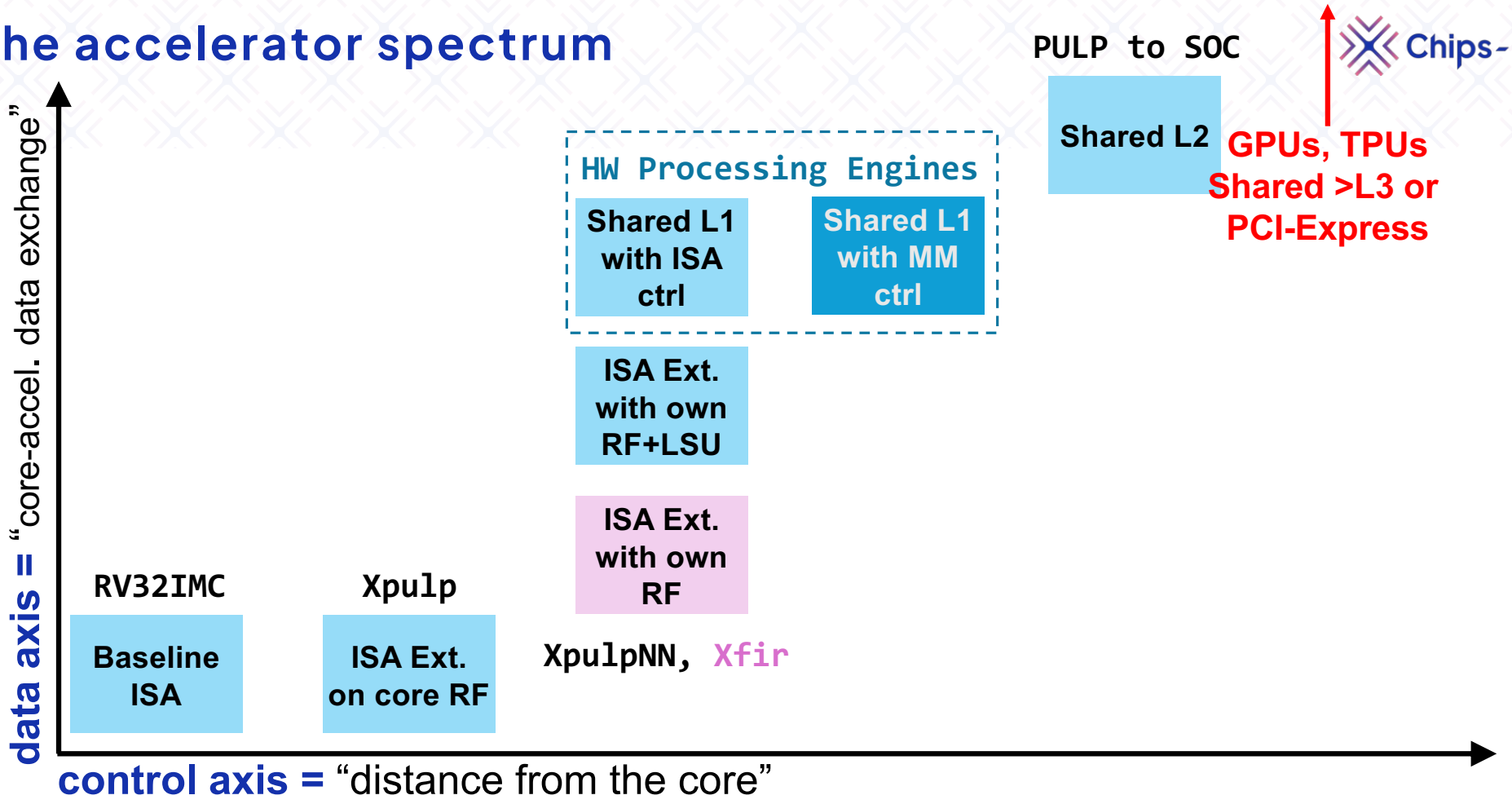


  - This can be anything ranging from a *really* loosely-coupled core (100-1000s of cycles to move data core to accelerator) to a PULP cluster-coupled engine (sharing memory at L1)

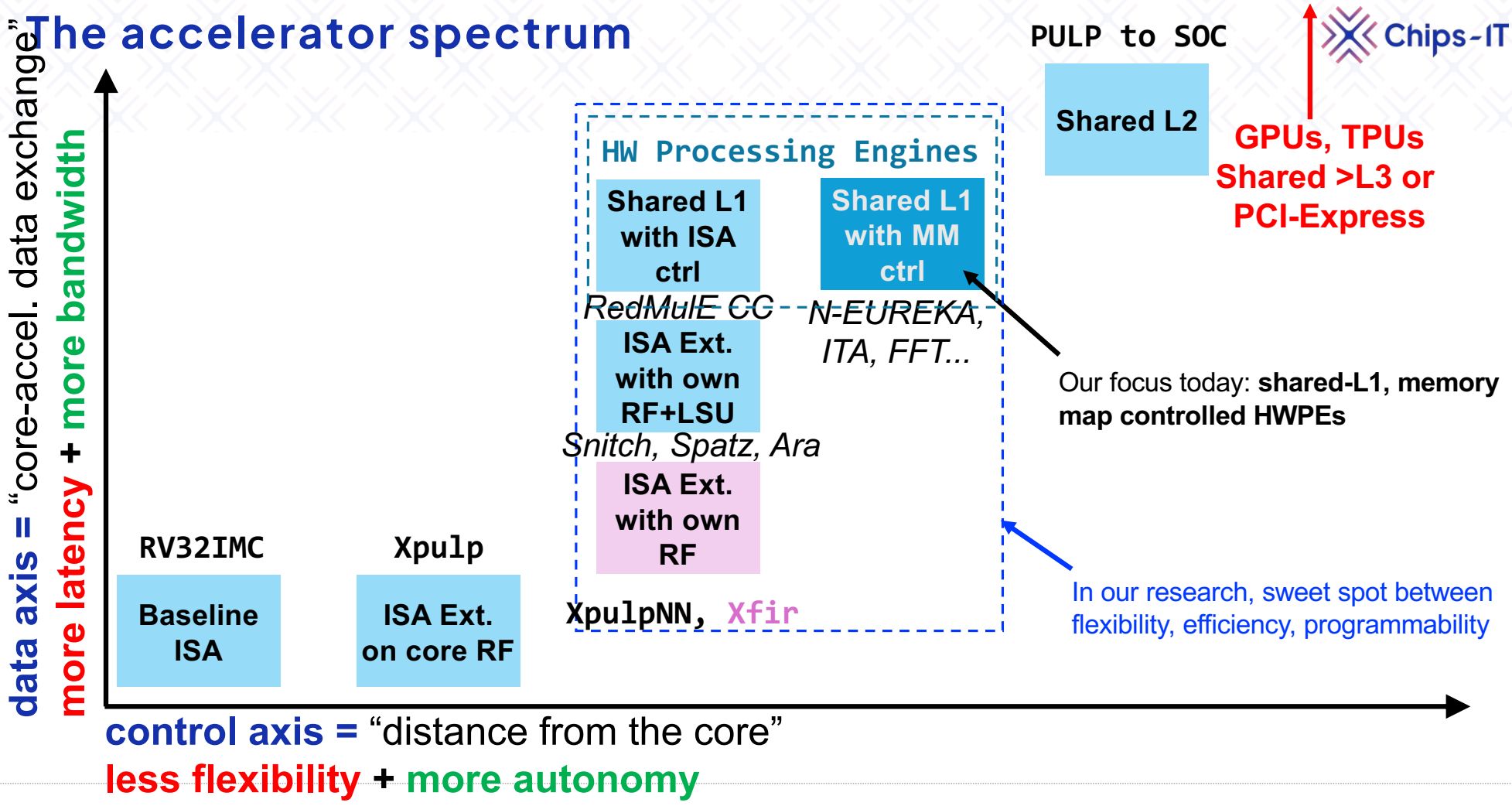
The accelerator spectrum



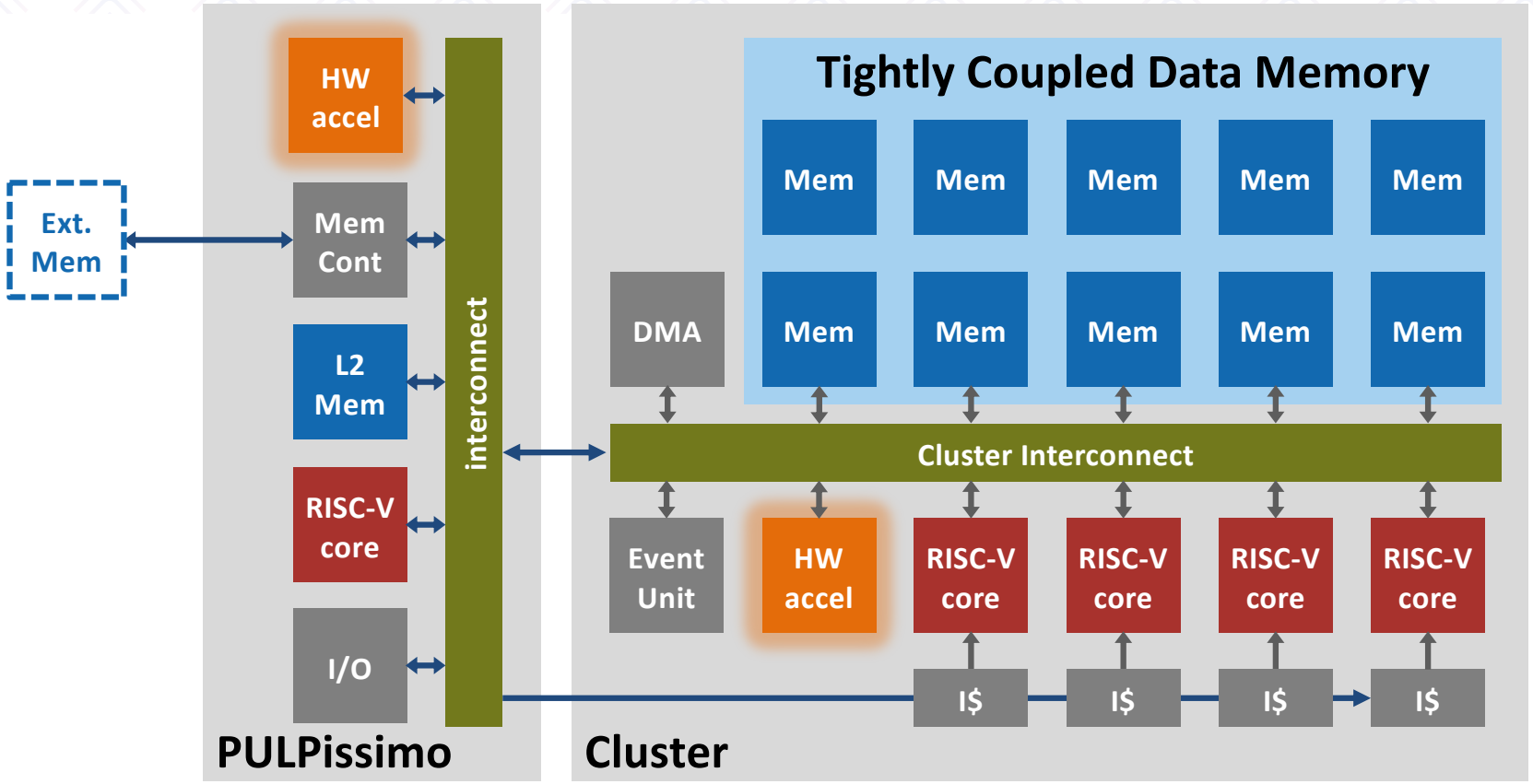
The accelerator spectrum



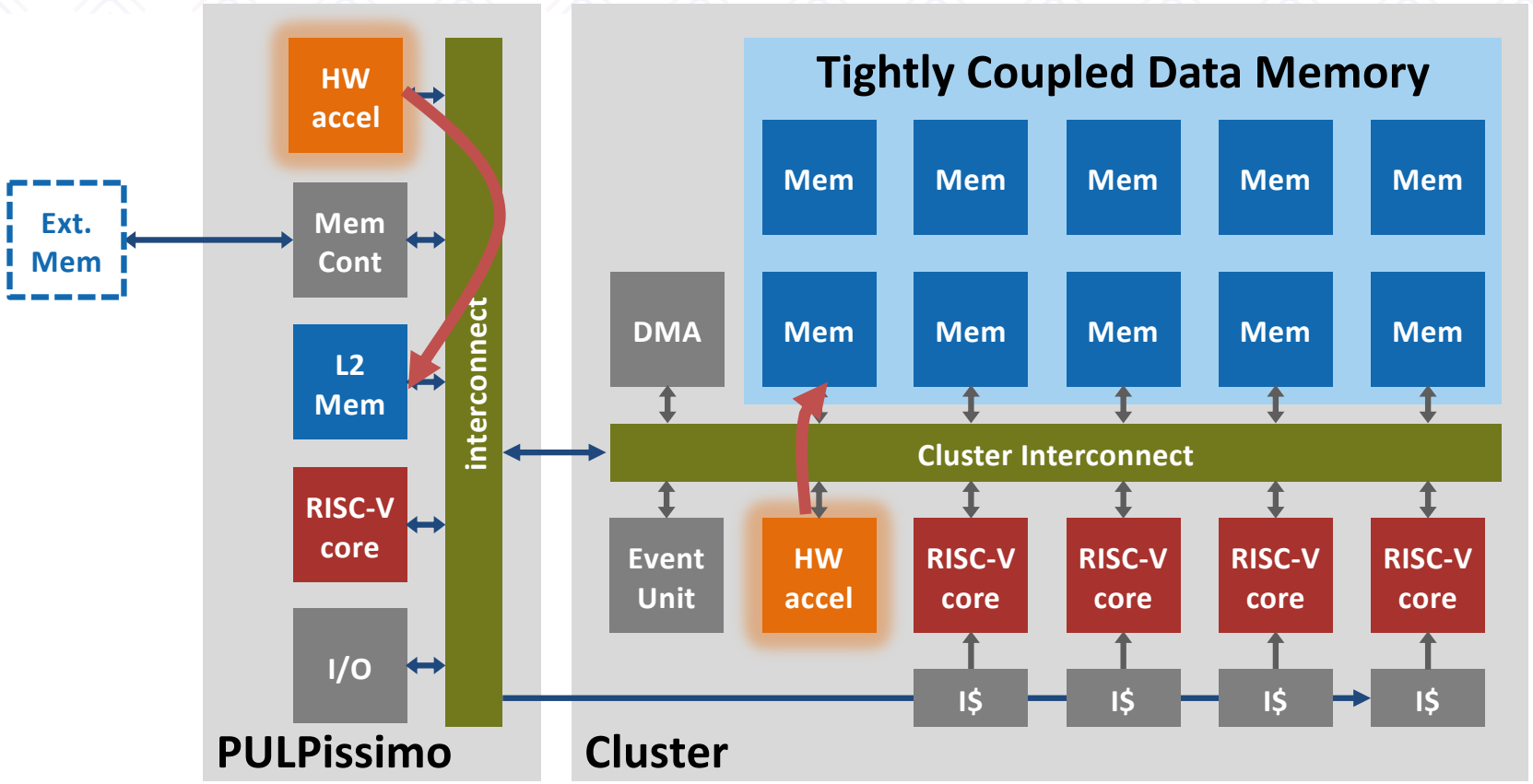
The accelerator spectrum



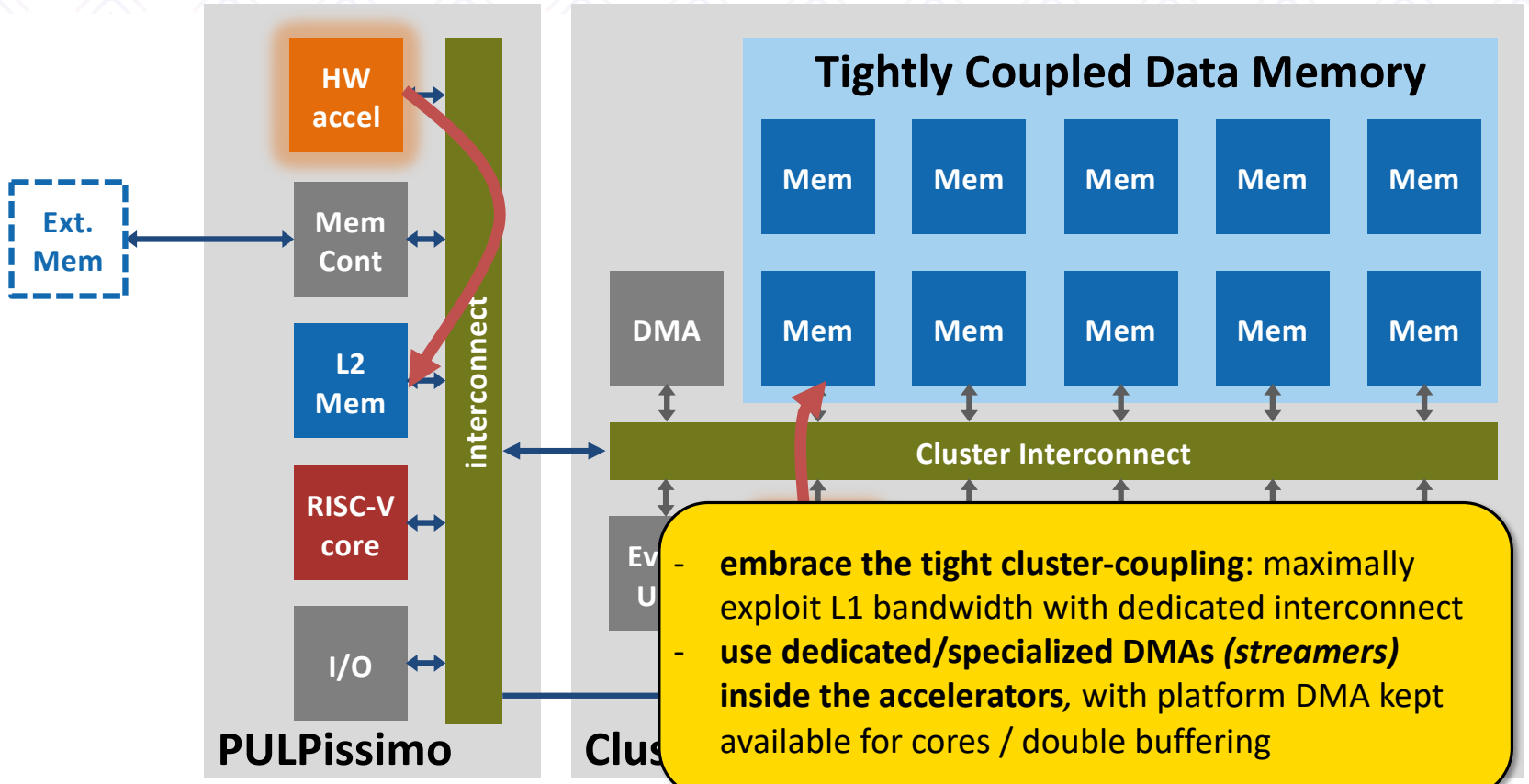
Cooperative Shared-L1 Accelerators



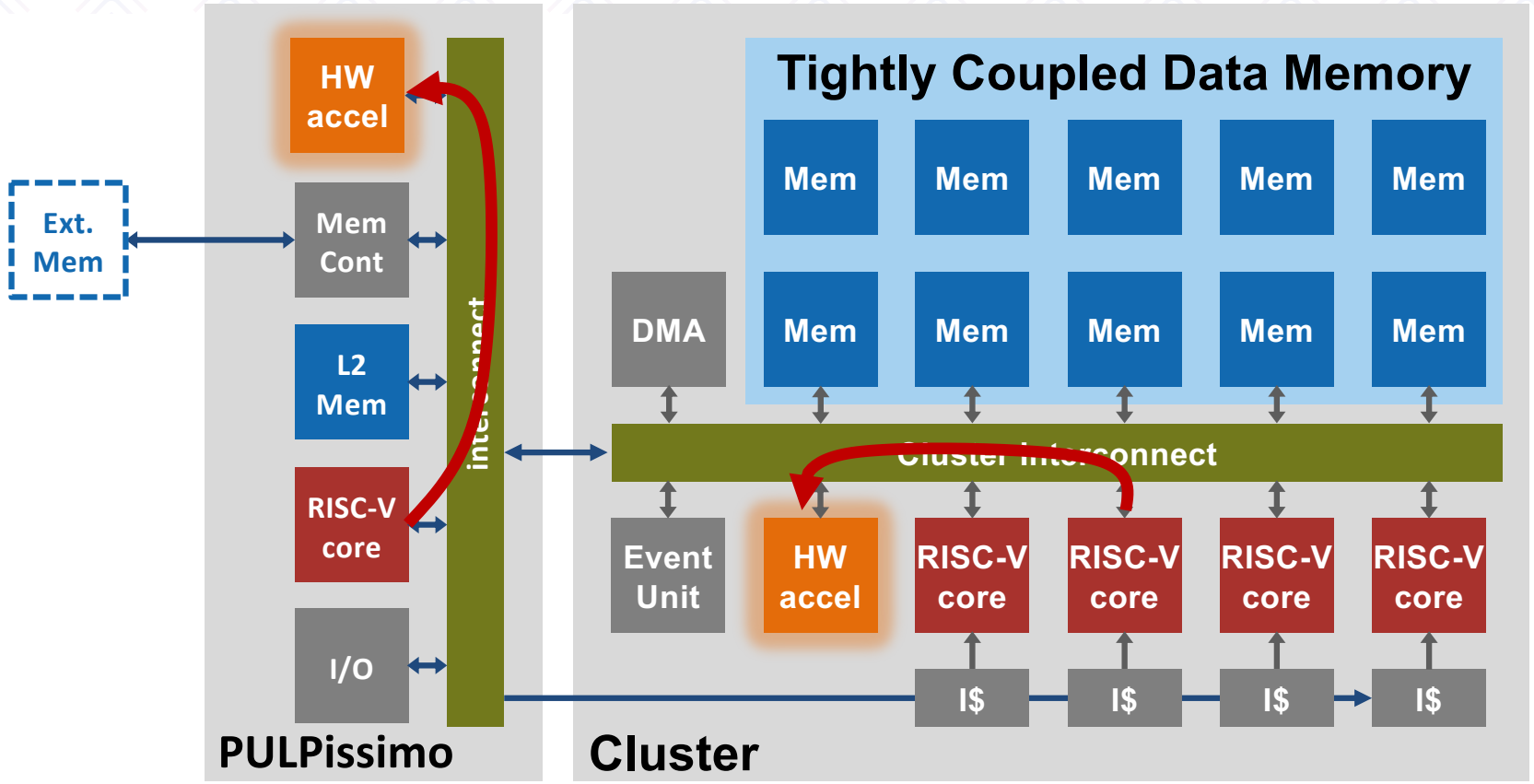
1. High-bandwidth, latency-tolerant memory access



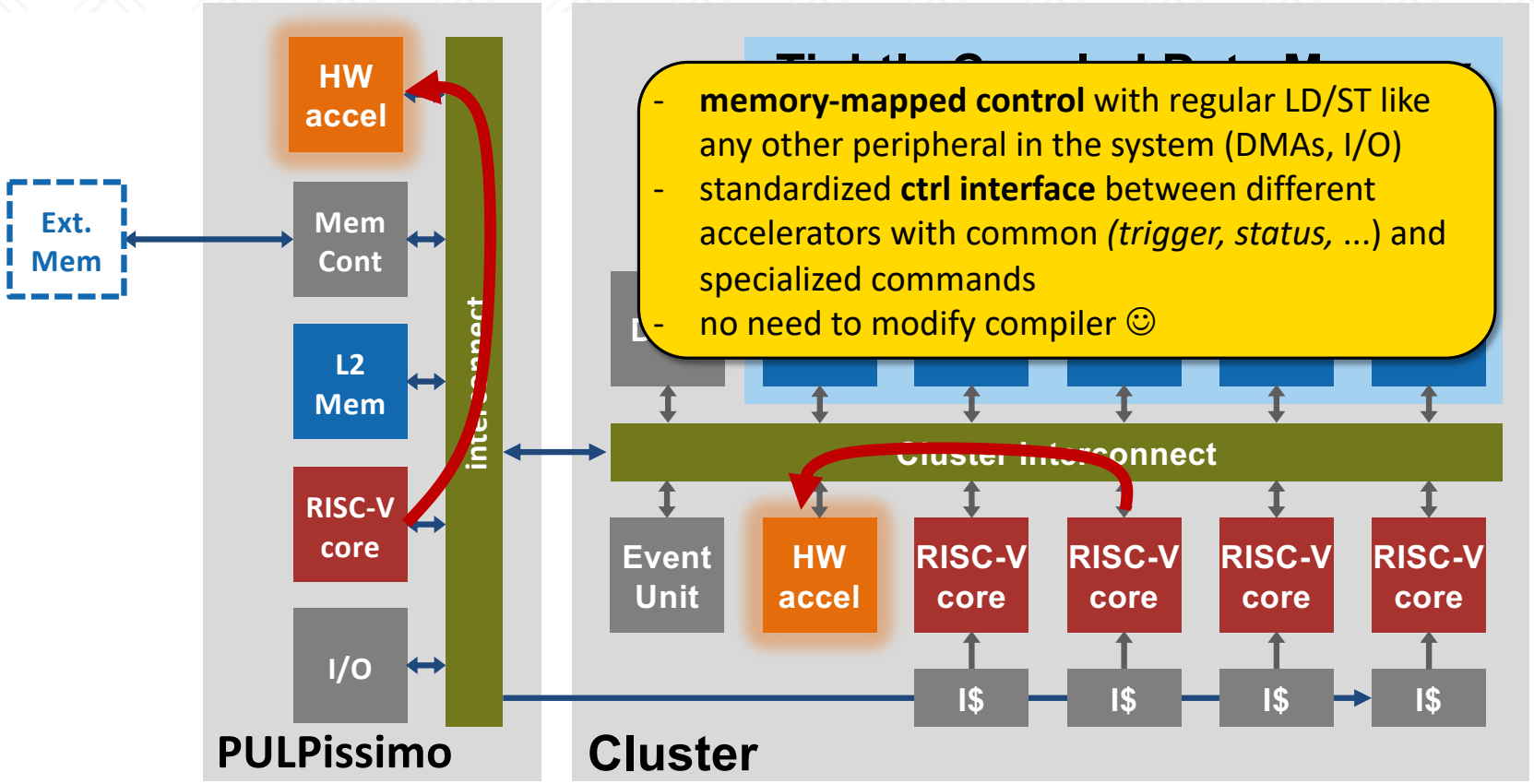
1. High-bandwidth, latency-tolerant memory access



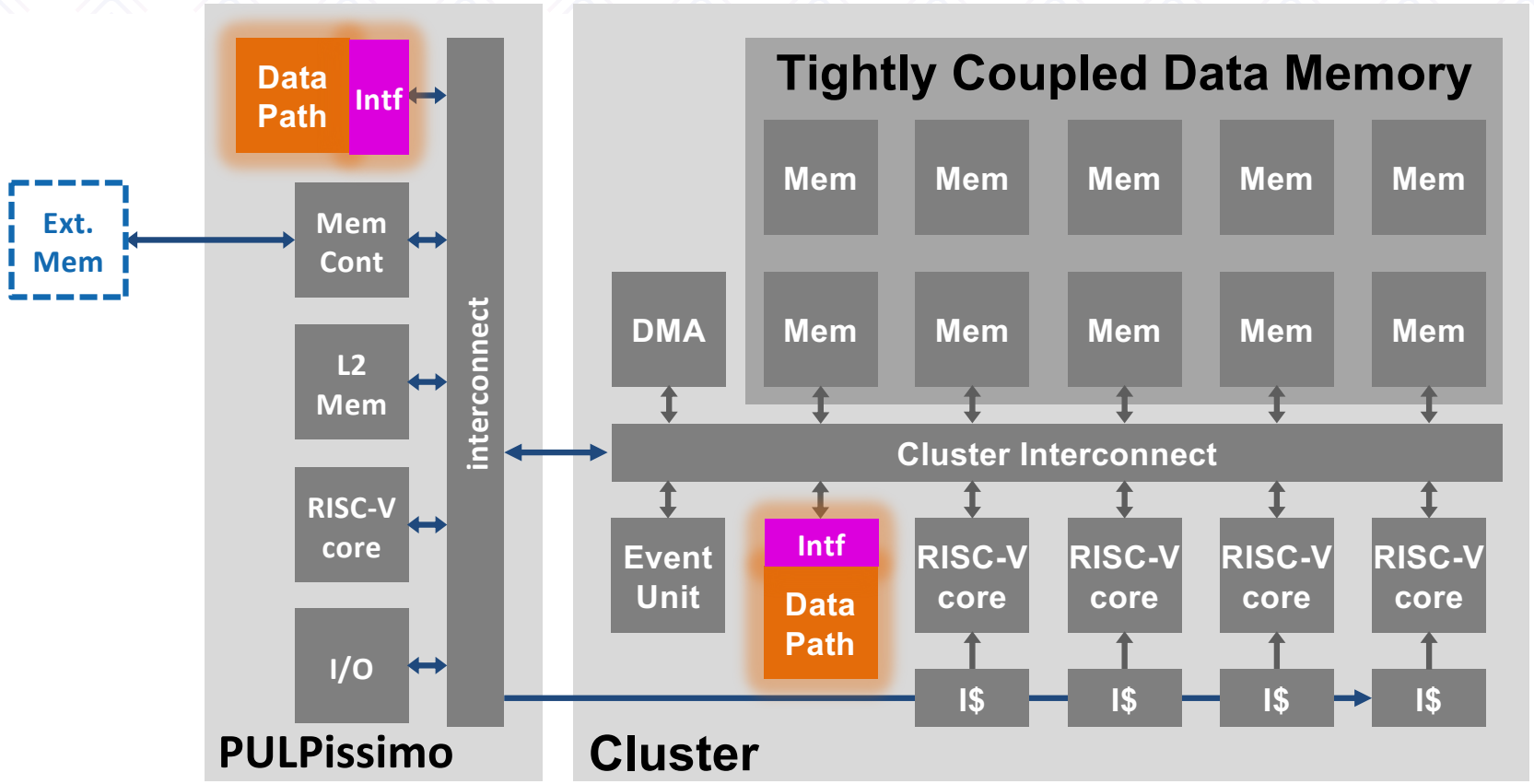
2. Low overhead control in SW



2. Low overhead control in SW

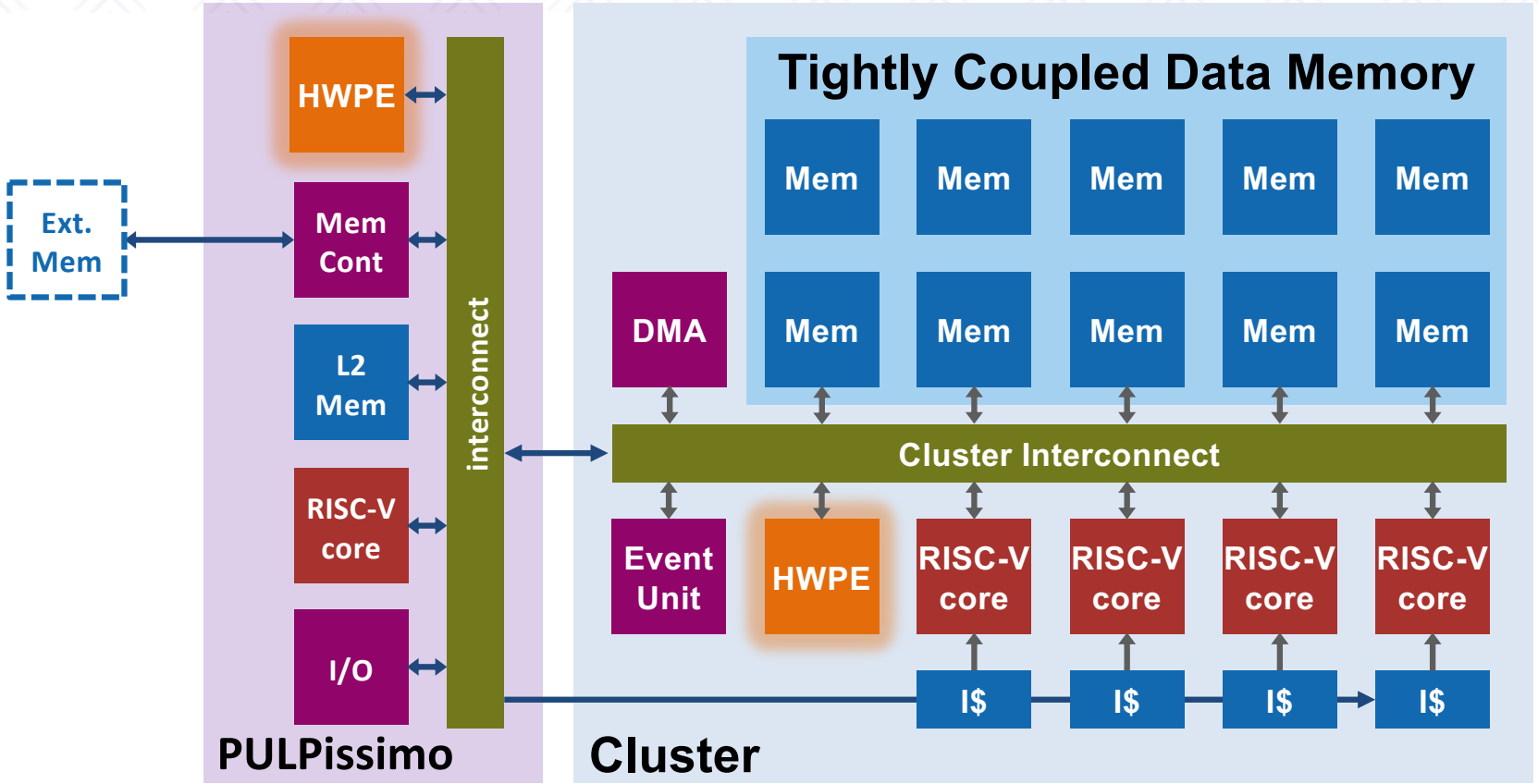


3. Standardized interface to simplify design



1+2+3 = PULP Hardware Processing Engines (HWPEs)

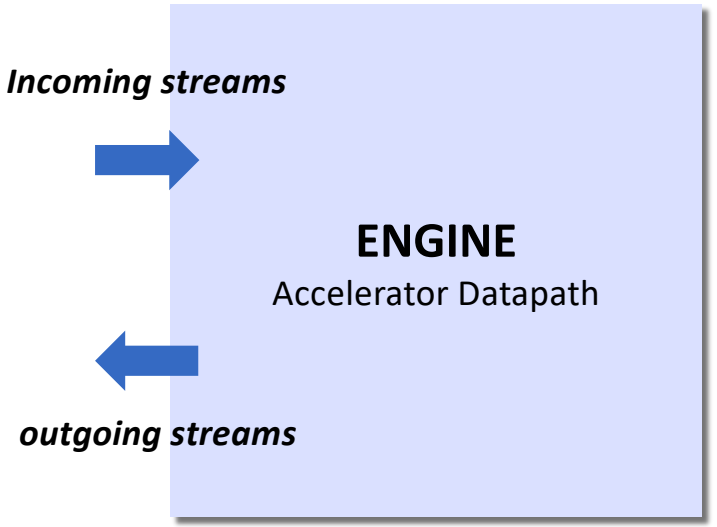
Chips-IT



Documentation: https://hwpe-doc.rtfd.io

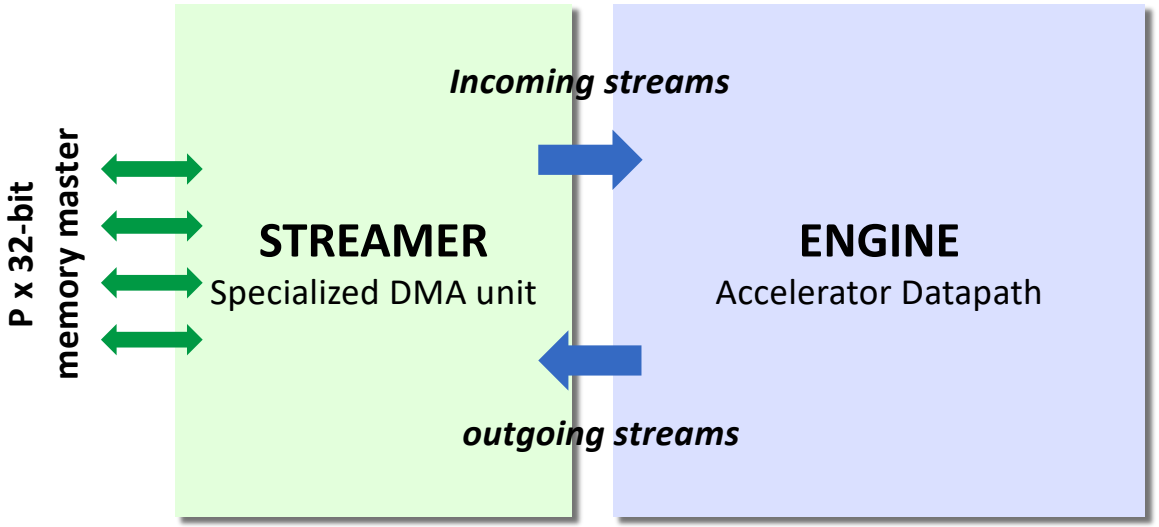
HWPE-Style to design accelerators

- **Factorize design in three main components**
  - An accelerated datapath, which we call **engine**
    - Fully custom and accelerator specific
    - Main constraint: design it to be *latency-tolerant*
    - *Example*: FP16 systolic array in RedMulE, Partially Bit-Serial Array in N-EUREKA, Radix-2 in FFT...
  - The engine exchanges data using **HWPE-Streams**
    - Not explicit memory accesses, but defined purely by *ordering*
    - Very lightweight protocol based on *ready* & *valid* handshake
    - Natively tolerant to latency!



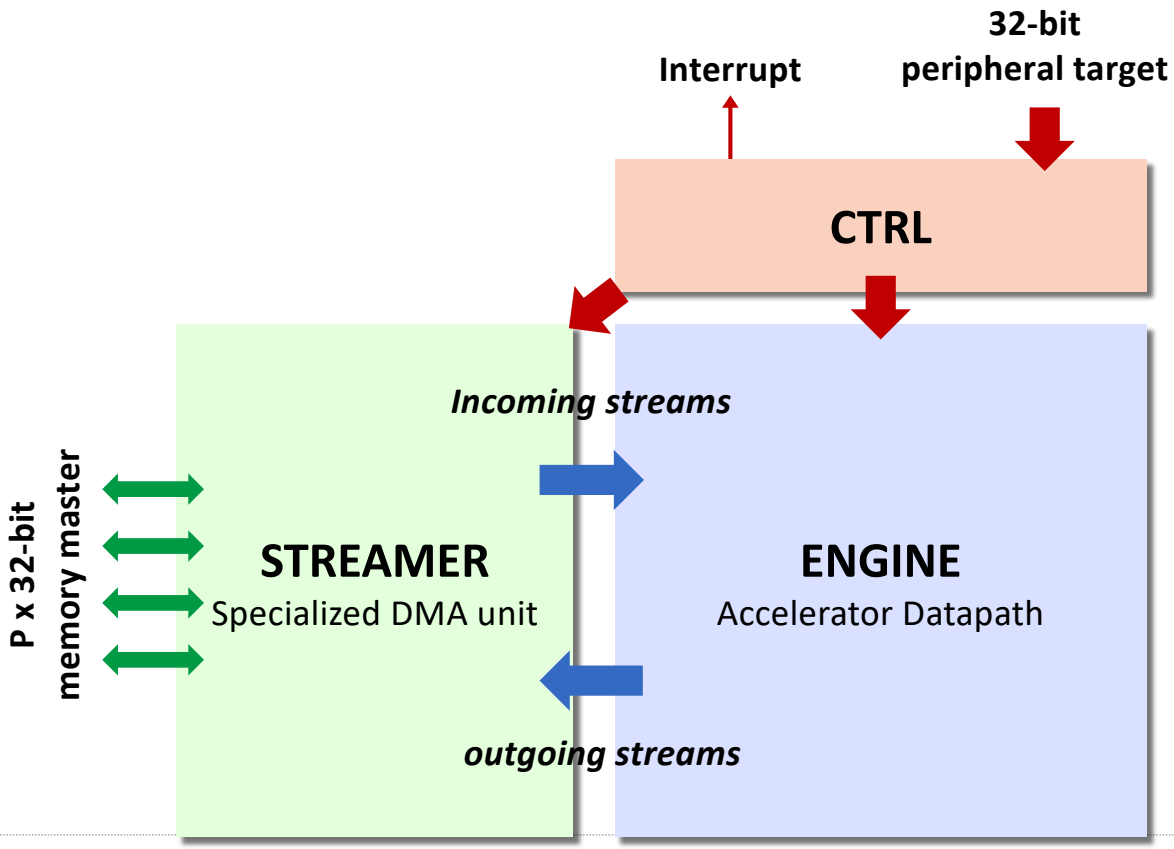
HWPE-Style to design accelerators

- **Factorize design in three main components**
  - An accelerated datapath, which we call **engine**
  - A specialized data mover called **streamer**
    - Converts from HWPE-Streams to actual memory accesses and vice versa
    - Tailored to accelerator, but composed of standard pieces (LEGO-like), e.g., FIFOs, source, sink modules



HWPE-Style to design accelerators

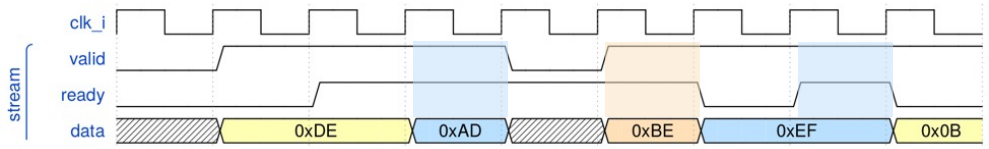
- **Factorize design in three main components**
  - An accelerated datapath, which we call **engine**
  - A specialized data mover called **streamer**
  - A local HWPE **controller**
    - Further composed of a standardized peripheral target + custom control (e.g., FSMs)
    - Several components available



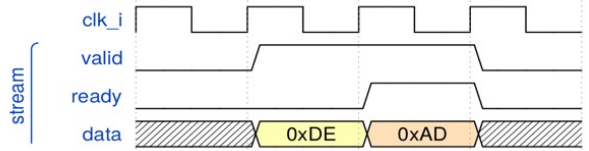
HWPE-Stream latency-insensitive protocol

Monodirectional, minimal (valid/ready handshake + data + optional strobe), no assumptions on content.
Positional information carried by order of data packets, no explicit address

1. handshake happens when valid & ready = 1

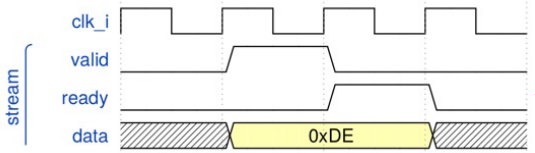


2. data/strobe can only change 1) following a handshake, 2) when valid is deasserted



3. assertion of valid (0 to 1) cannot depend combinationally on ready

4. deassertion of valid (1 to 0) can only happen in the cycle after a handshake



Documentation → https://hwpe-doc.readthedocs.io/en/latest/

Back-pressure and Latency-Insensitive scheduling

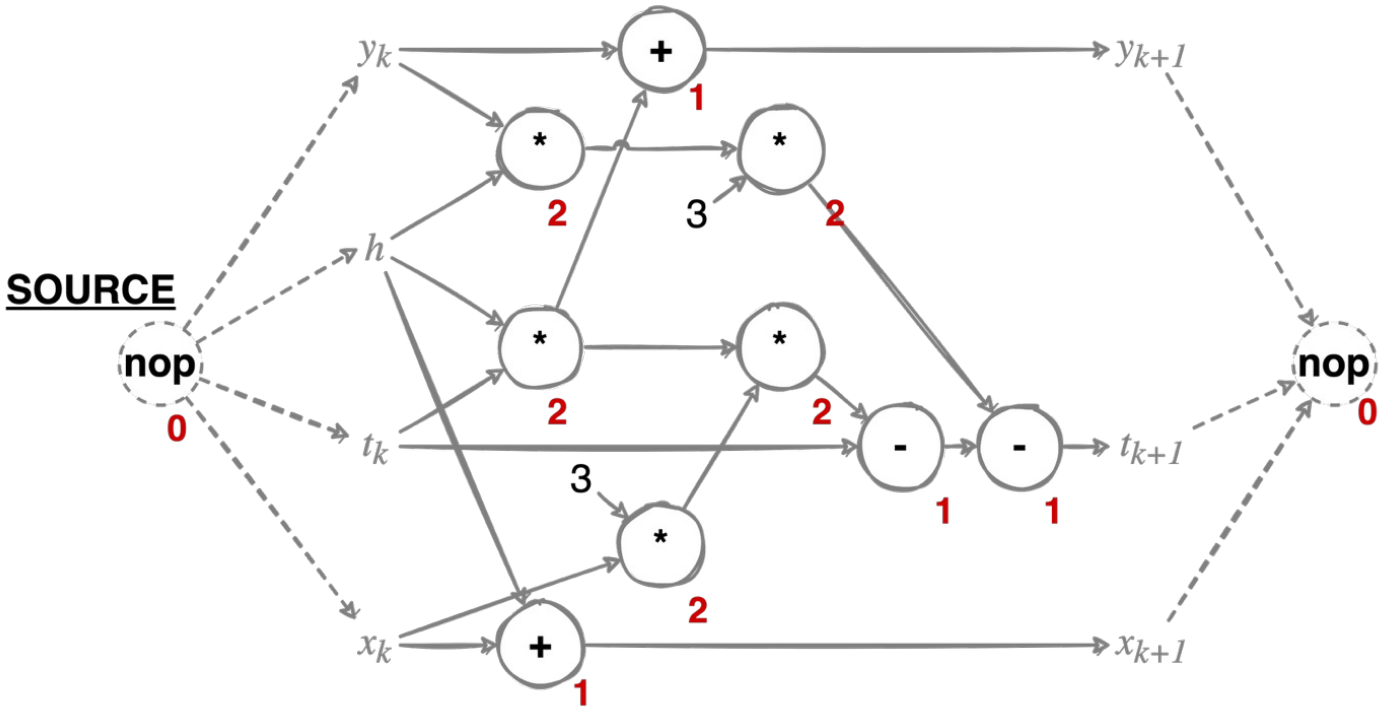
Chips-IT

- Sometimes, the **successor** nodes could also be in trouble in receiving incoming data
  - A memory or bus where we want to store *output* data from the sink
  - A node that cannot accept new inputs for 1 or more cycles after finishing its work
  - More example will naturally come out when we apply architectural transformations: **parallelism/replication**, **pipelining**, and **time-multiplexing**
- In these cases, we need a **back-pressure** mechanism so that successor nodes can stall incoming ones
- A **valid** signal is not enough alone; we need a **valid+ready** or equivalent handshaking strategy.

- We also need to make our operators ***patient***: a patient operator can be stalled from either direction (from a predecessor with **valid=0** or from a successor with **ready=0**) and must procrastinate its action appropriately to avoid incorrect conditions
  - for example: if **ready=0** do not issue the next output (even if internally available), and if necessary, back-propagate the **ready=0**
  - we propagate data when a **handshake** (i.e., an agreement between signals) happens

HWPE Engine as a latency-insensitive DFG

- Suppose all types of operations are labelled with a delay (in this example, an integer number of cycles).
- When do we run each node?
- Scheduling is the task of determining start times subject to:
  - Precedence constraints from DFG
  - Delay of each node
  - Constraints (resources, max. latency)

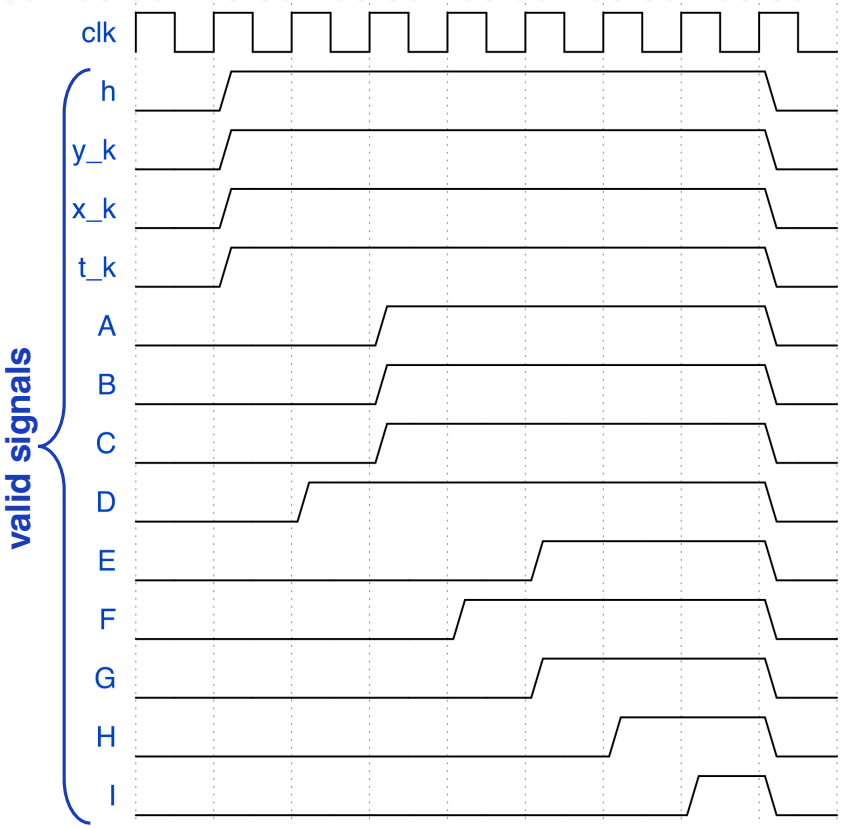


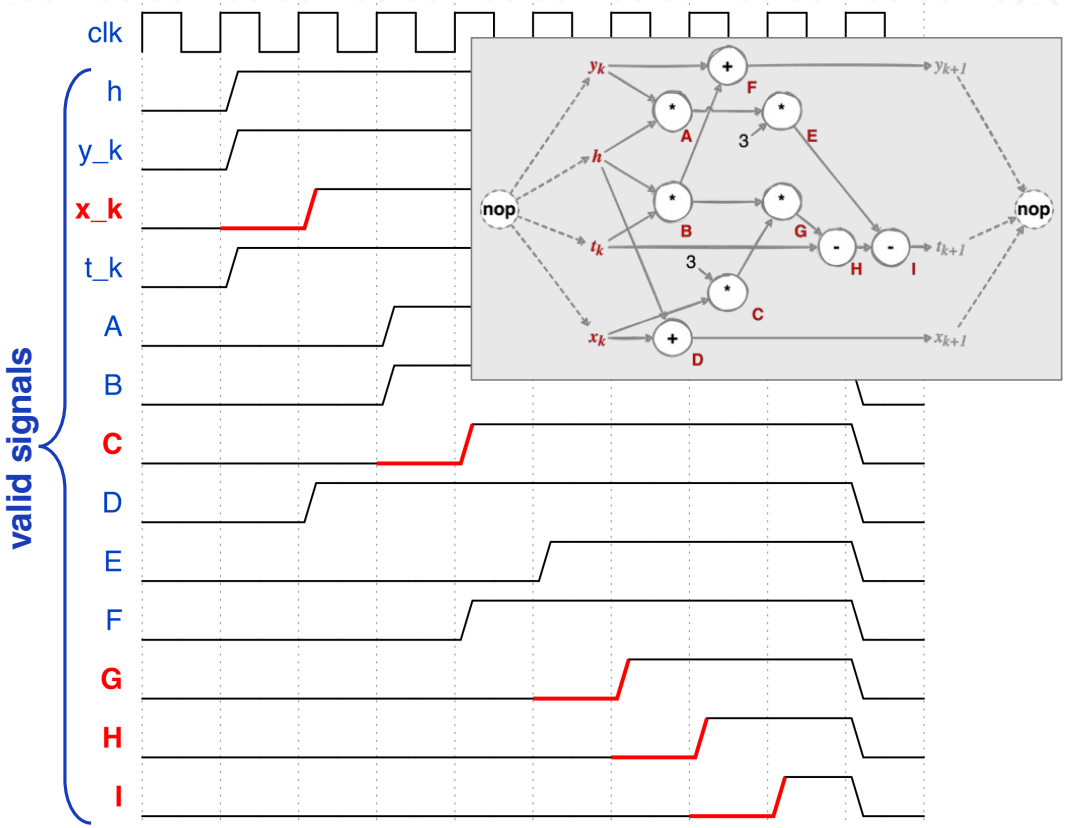
HWPE Engine as a latency-insensitive DFG

Chips-IT

- What if a node's delay is **unknown** or **unbounded**?
  - Not a theoretical problem: think of data coming from a **memory** shared (and contended) with other actors – in this case the *source* and *sink* nodes have unbounded latency!
  - Other nodes can be deterministic, but complex to assign a delay to: think of a **complex sequential circuit** as an operator node instead of the "simple" arithmetic units considered up to now
- **Adaptive** or **self-timed** strategy: start each operation not at a pre-given cycle, but whenever its inputs are **valid**
  - to do so, we couple each connection with a **valid** signal → high when the output of a node can be used by successors, low when it is not yet computed or not available for any reason
  - We can schedule the CDFG "as if" delay was known (e.g., 2 cycle * and 1 cycle +,-)
  - Each node must be stallable!

Example of adaptive scheduling



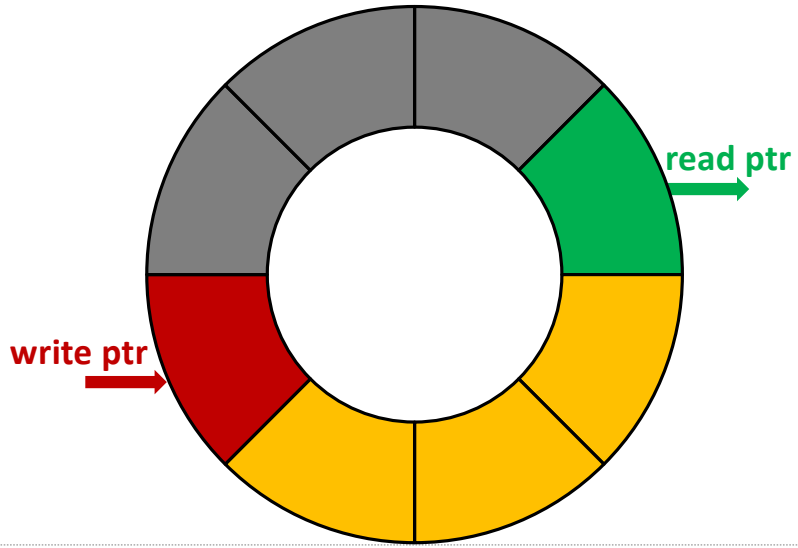


Additional delays “fall through” the CDFG without breaking its functionality

Buffering *HWPE-Streams*: FIFOs

Chips-IT

- Implemented as a **circular buffer**:
  - not moving data, but pointers (less power!)
  - requires “special FIFOs” to extract intermediate elements
  - less need for MUXes!



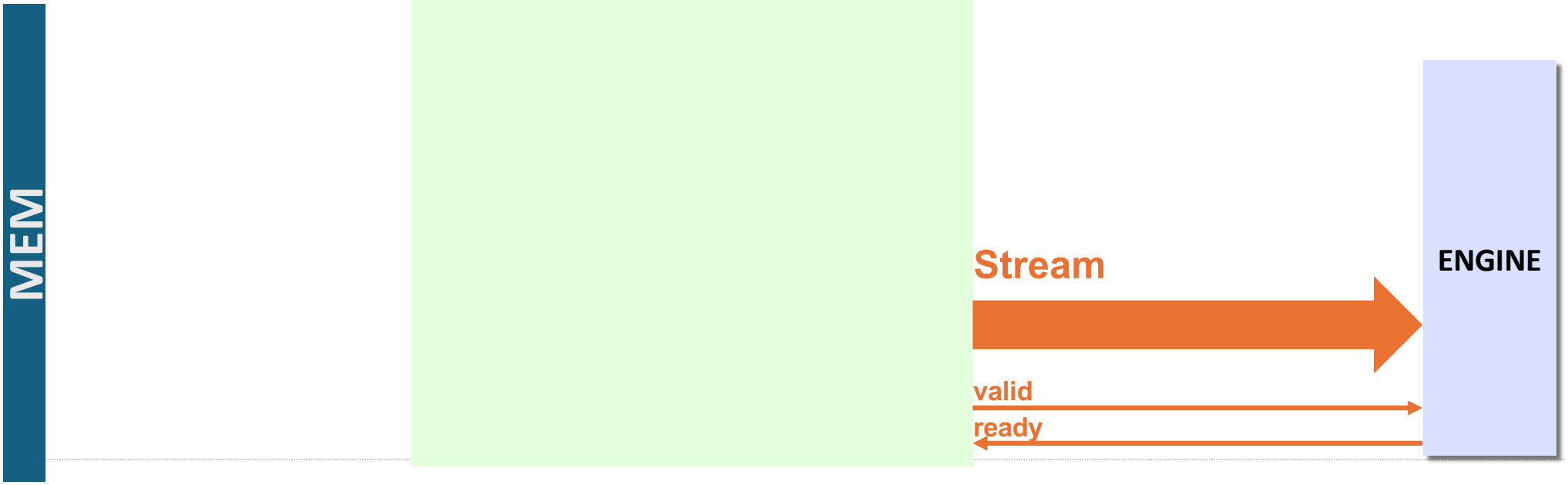

```
logic buffer[FIFO_LEN-1:0][31:0];
logic unsigned [$clog2(FIFO_LEN)-1:0] write_ptr, read_ptr;

always_ff @(clk_i) begin
  if(read_handshake) begin
    out <= buffer[read_ptr];
    read_ptr <= read_ptr + 1; // modulo FIFO length
  end
  if(write_handshake) begin
    buffer[write_ptr] <= in;
    write_ptr <= write_ptr + 1;
  end
end
```

Converting *HWPE-Streams* to memory accesses

Chips-IT

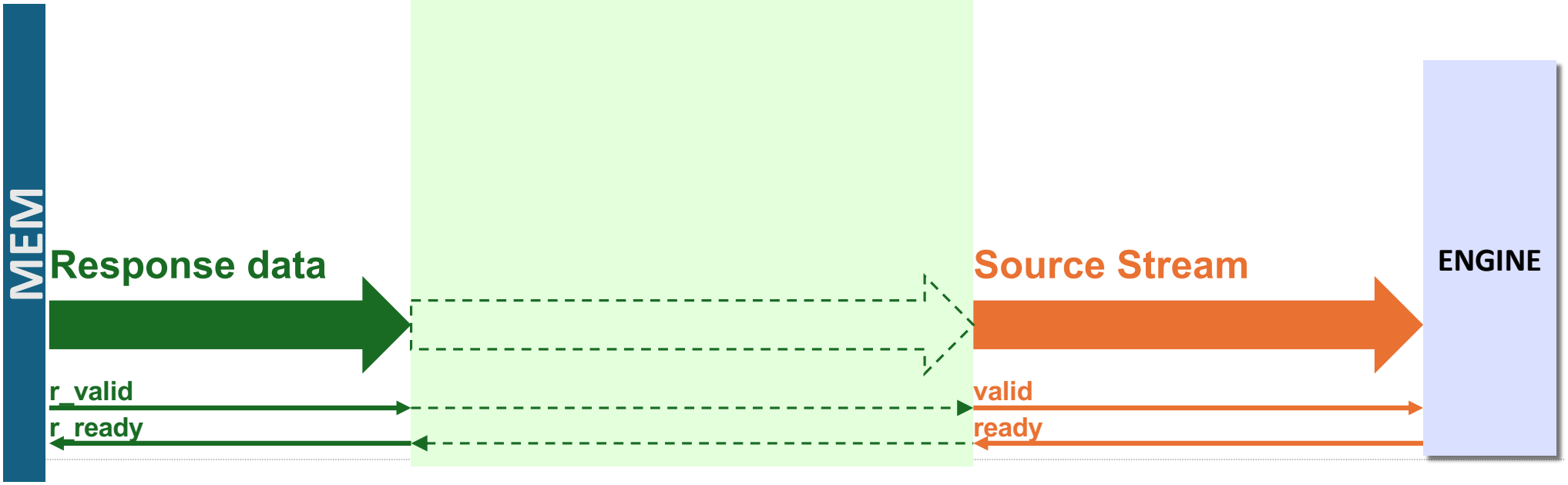
- **Source-side streams**
  - Generated from **load** operations



Converting *HWPE-Streams* to memory accesses

Chips-IT

- **Source-side streams**
  - Generated from **load** operations

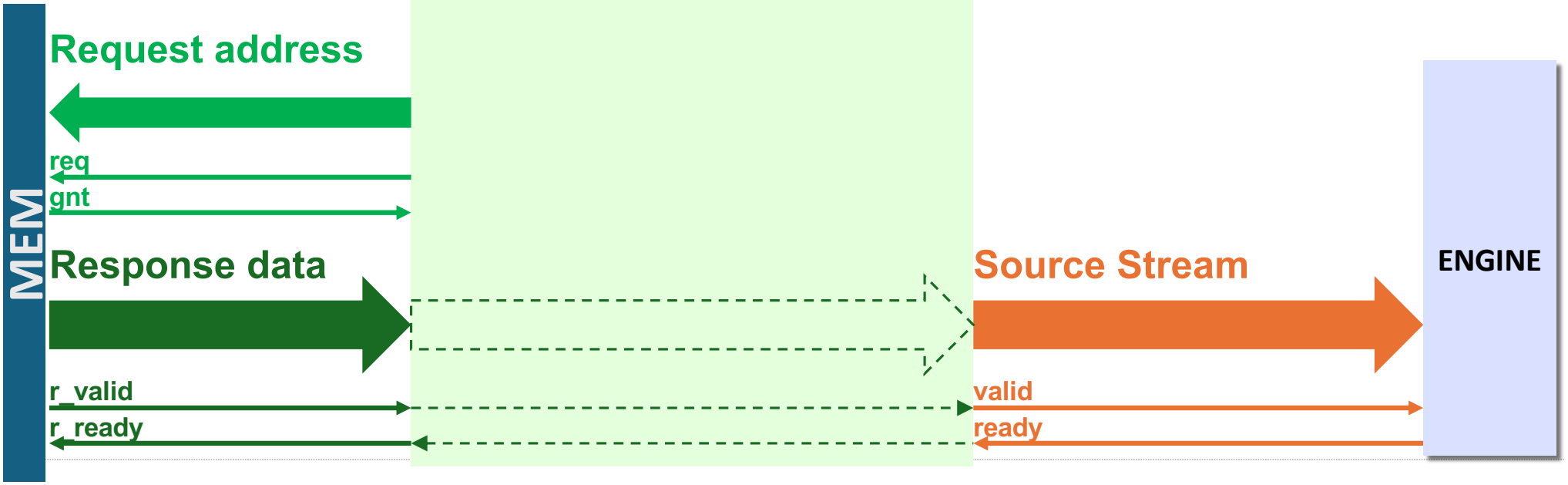


32

Converting *HWPE-Streams* to memory accesses

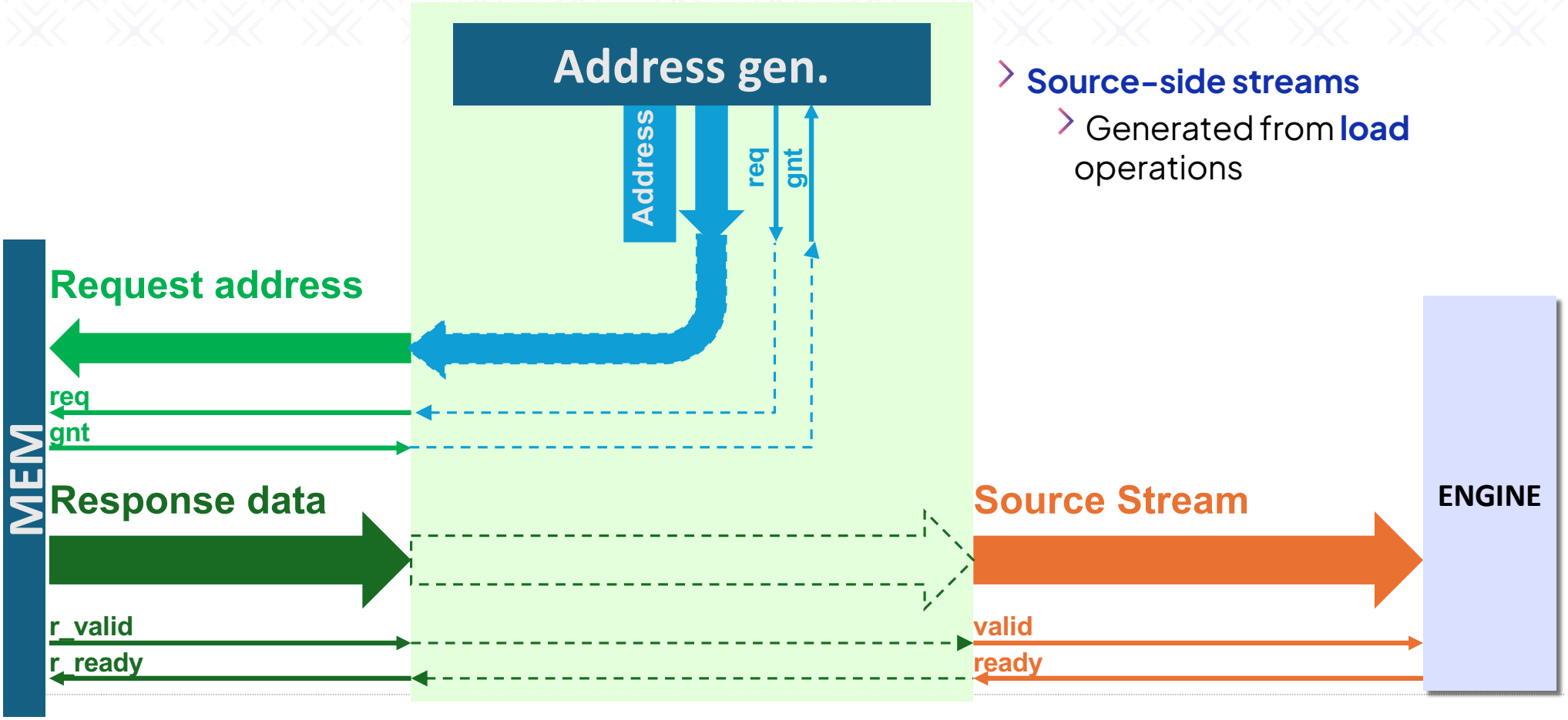
Chips-IT

- Source-side streams
  - Generated from **load** operations



Converting *HWPE-Streams* to memory accesses

Chips-IT

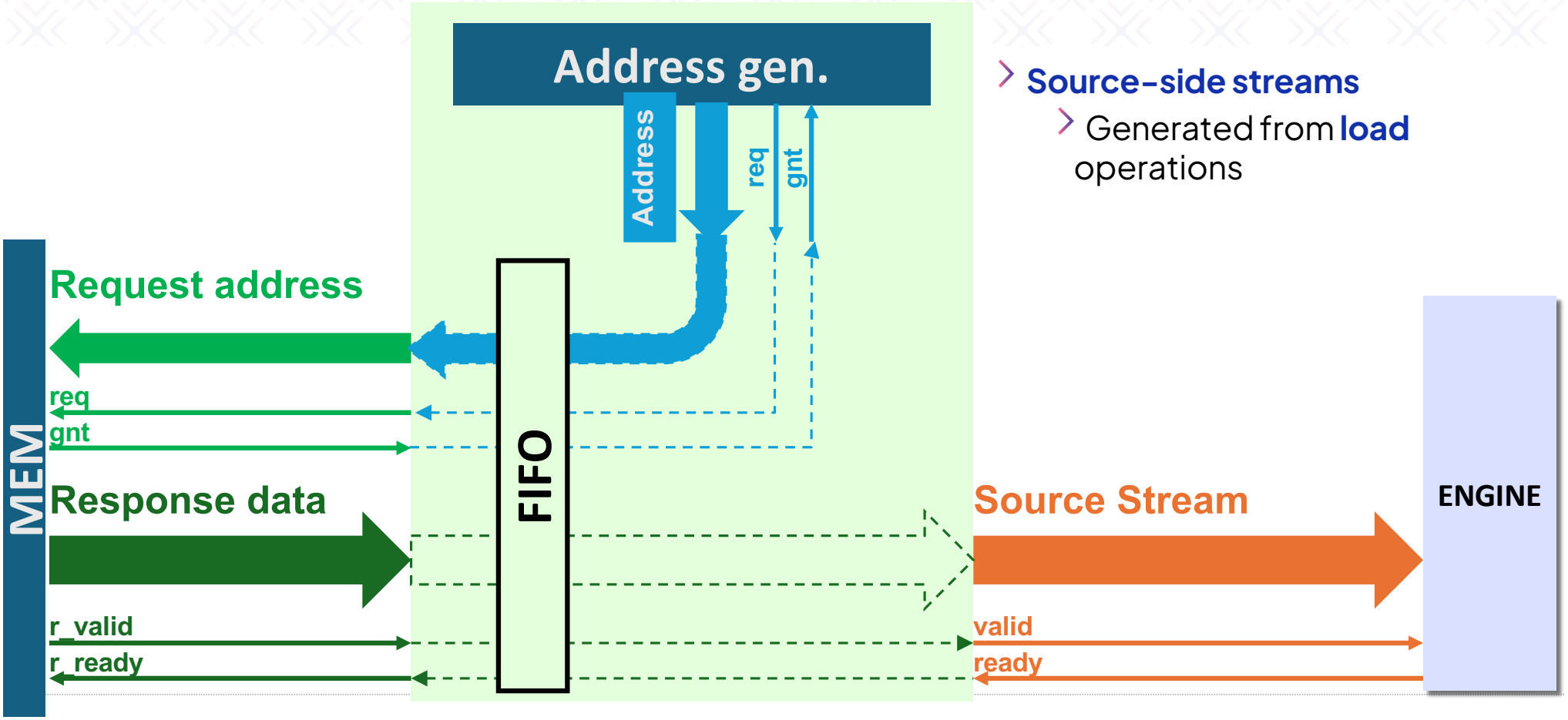


- Source-side streams
  - Generated from **load** operations

34

Converting *HWPE-Streams* to memory accesses

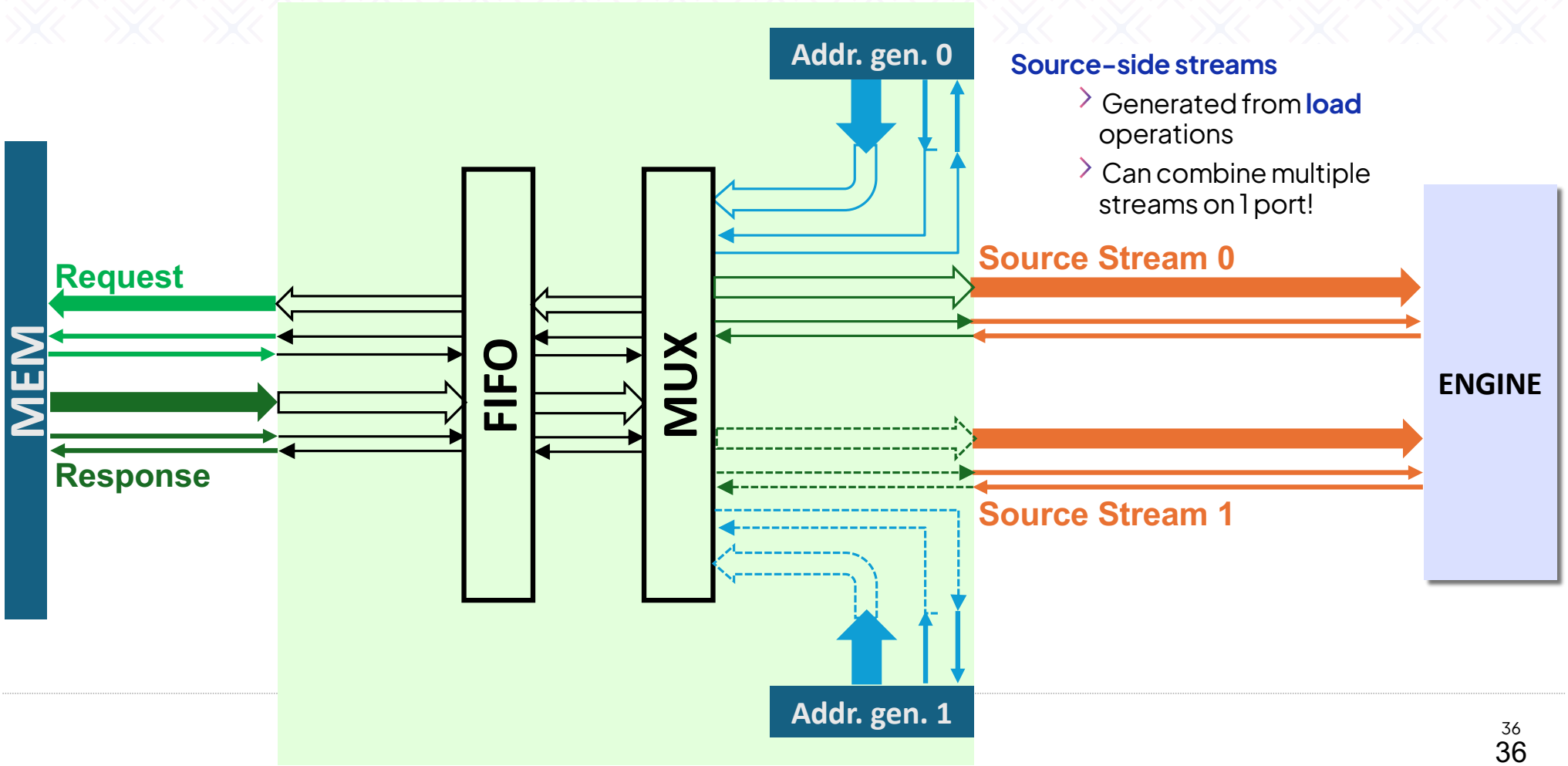
Chips-IT



- Source-side streams
  - Generated from **load** operations

35

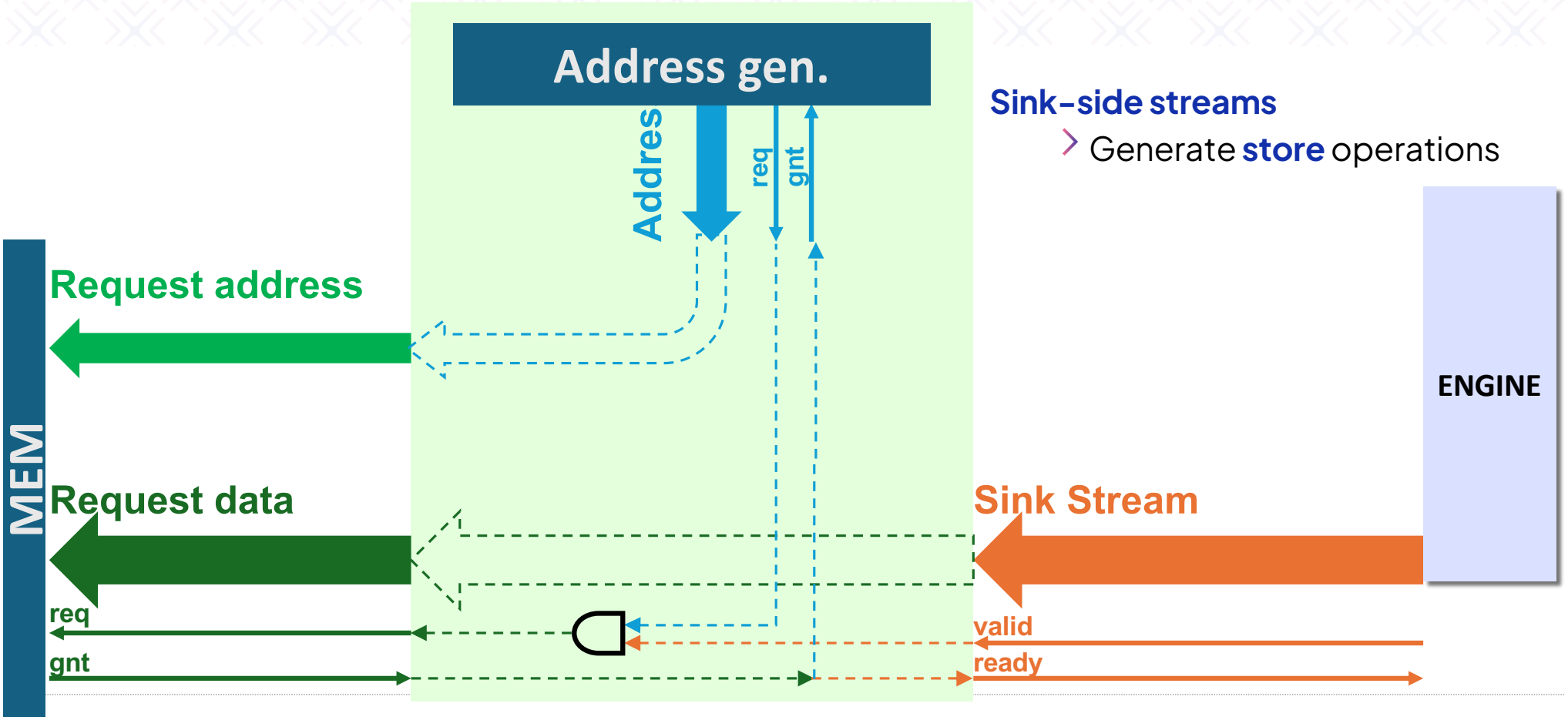
Converting *HWPE-Streams* to memory accesses



36

Converting *HWPE-Streams* to memory accesses

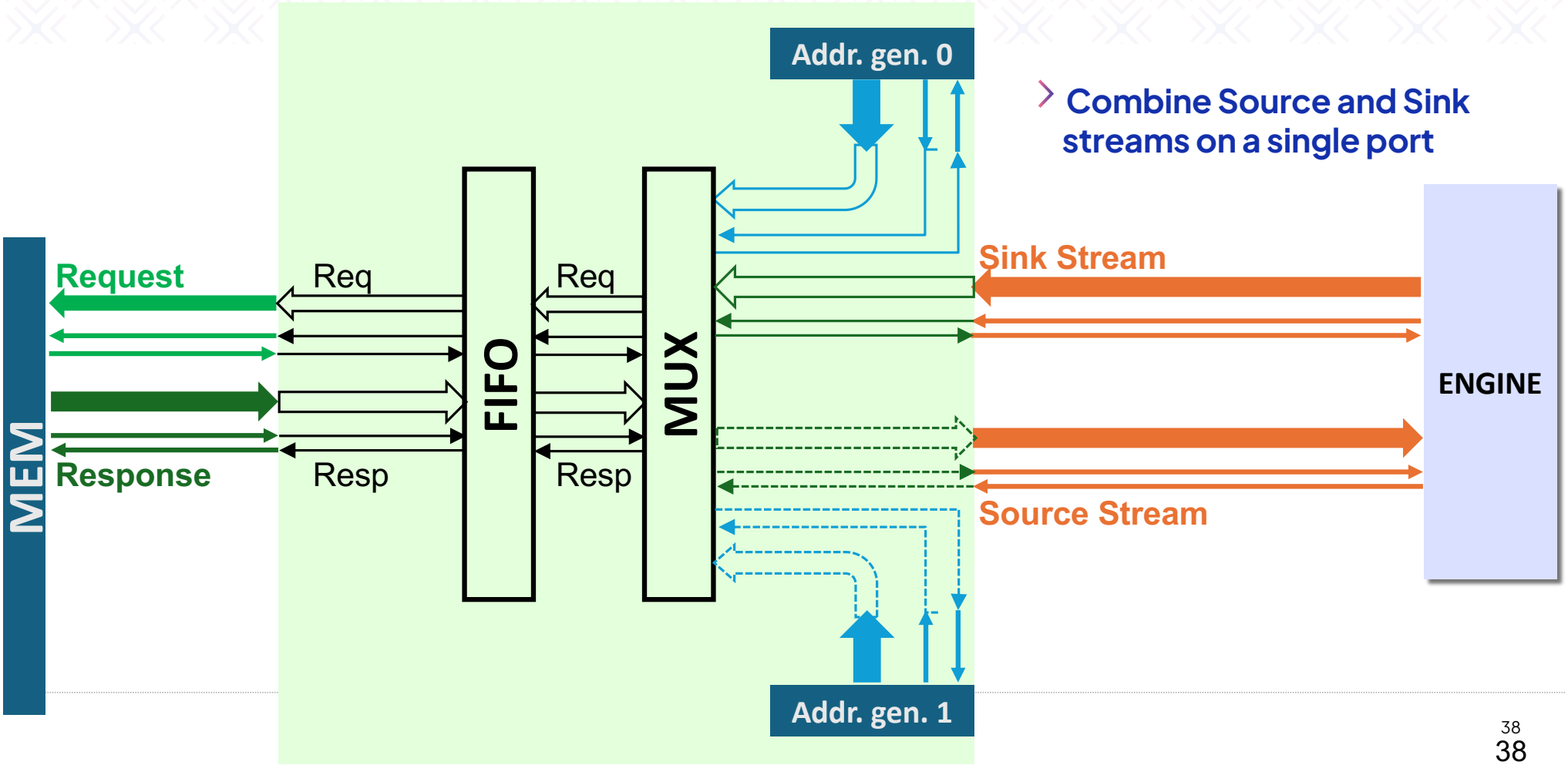
Chips-IT



Sink-side streams

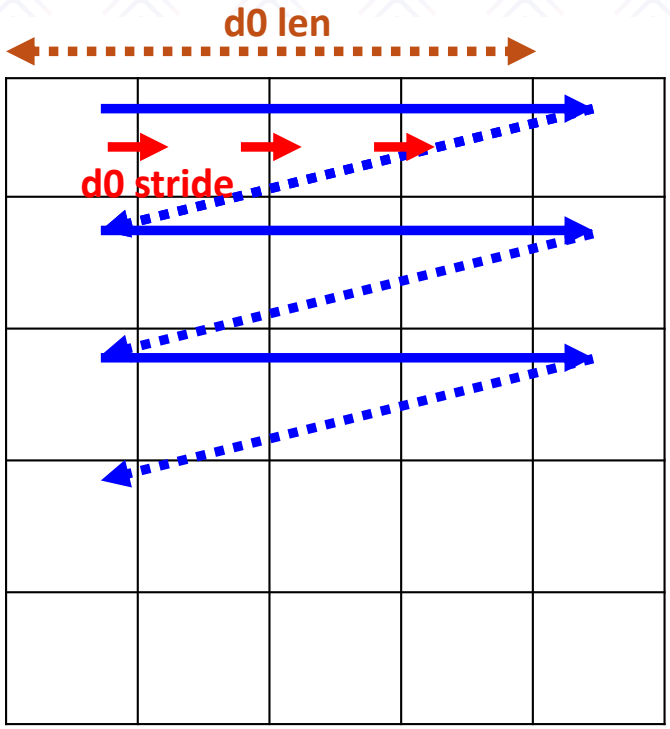
- Generate **store** operations

Converting *HWPE-Streams* to memory accesses



38

Address Generators



Streamers work on patterns

d0 stride: distance in bytes between two consecutive elements in *d0* dimension

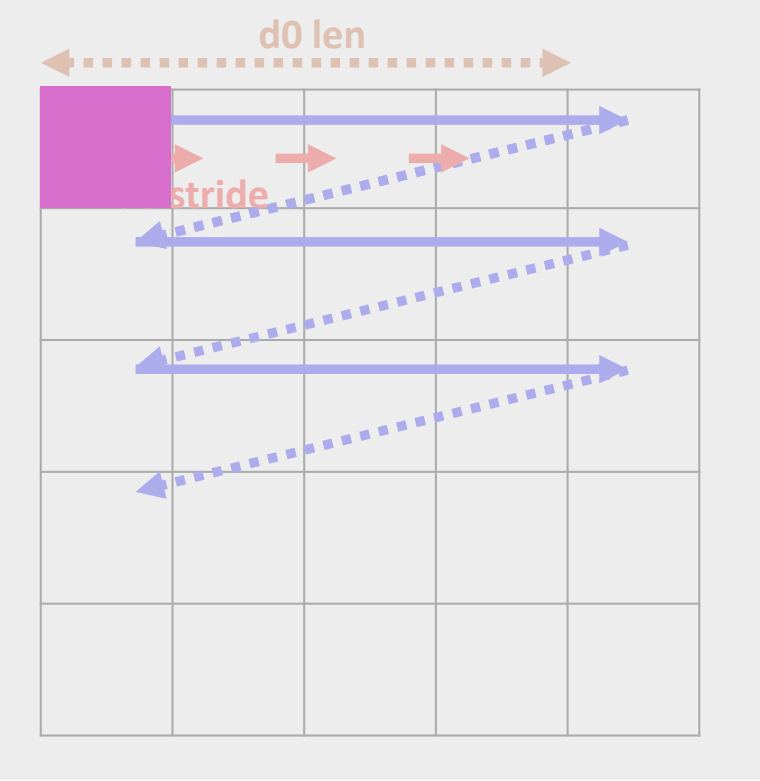
d0 len: number of consecutive elements in *d0* dimension

d1 stride: distance in bytes between two consecutive elements in *d1* dimension

HWPE Address Generators support 3D strided accesses (d0, d1, d2)

https://hwpe-doc.readthedocs.io/en/latest/modules.html#hwpe-stream-addressgen-v3

Address Generators



Streamers work on patterns

d0 stride: distance in bytes between two consecutive elements in *d0* dimension

d0 len: number of consecutive elements in *d0* dimension

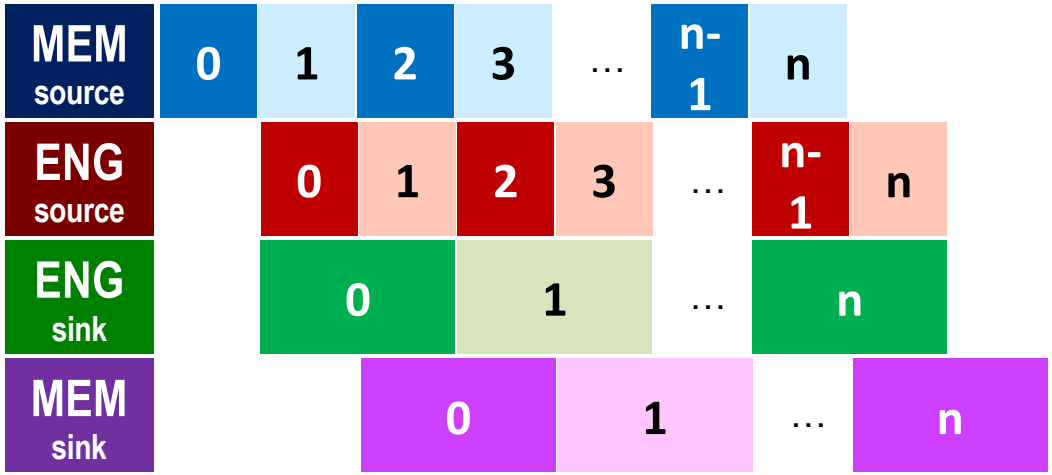
d1 stride: distance in bytes between two consecutive elements in *d1* dimension

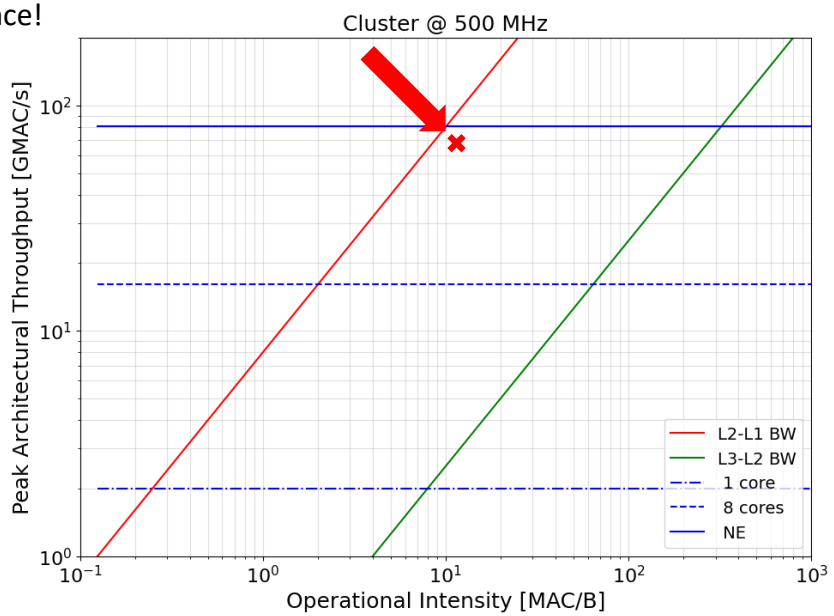
HWPE Address Generators support 3D strided accesses (d0, d1, d2)

https://hwpe-doc.readthedocs.io/en/latest/modules.html#hwpe-stream-addressgen-v3

The “cusp of perfomance”

- The *design point* for an efficient accelerator should always be at the “cusp” between the memory and compute roofs: in this point, the accelerator is optimally using the available memory bandwidth to provide performance!
- **Performance: OK ☺ Efficiency: OK ☺**



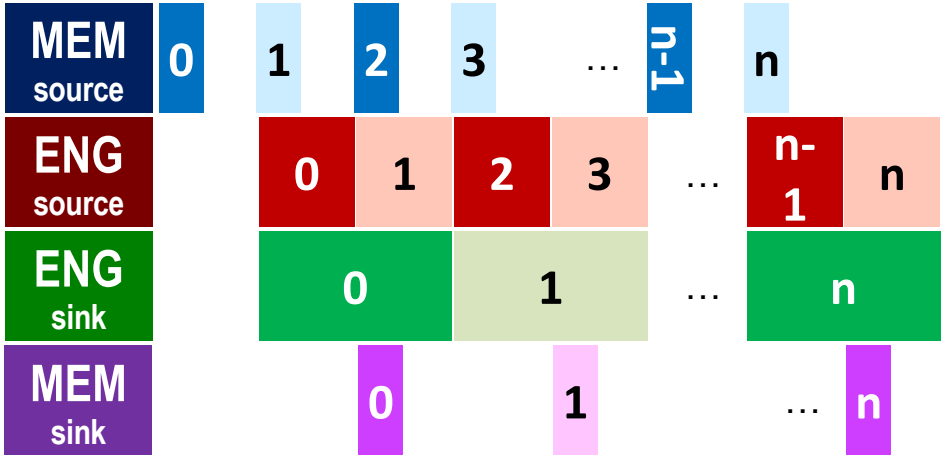


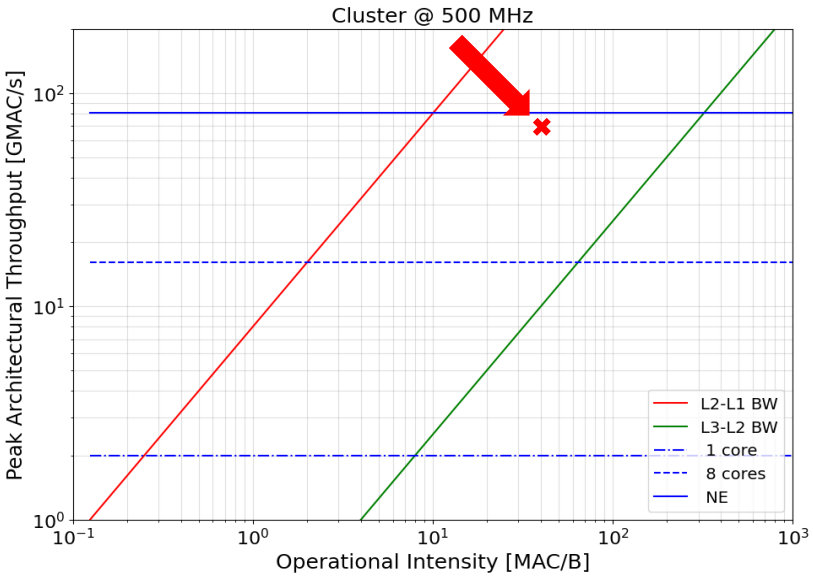
Roofline plot

- ***<u>We can not design a “better” accelerator from this point without also improving the memory system!</u>***

Compute-bound HWPE

- In **compute bound** conditions, the accelerator consumes and produces data at a higher rate than what can be furnished by the memory system
- **Performance: OK ☺ , Efficiency: not always very good 😐**



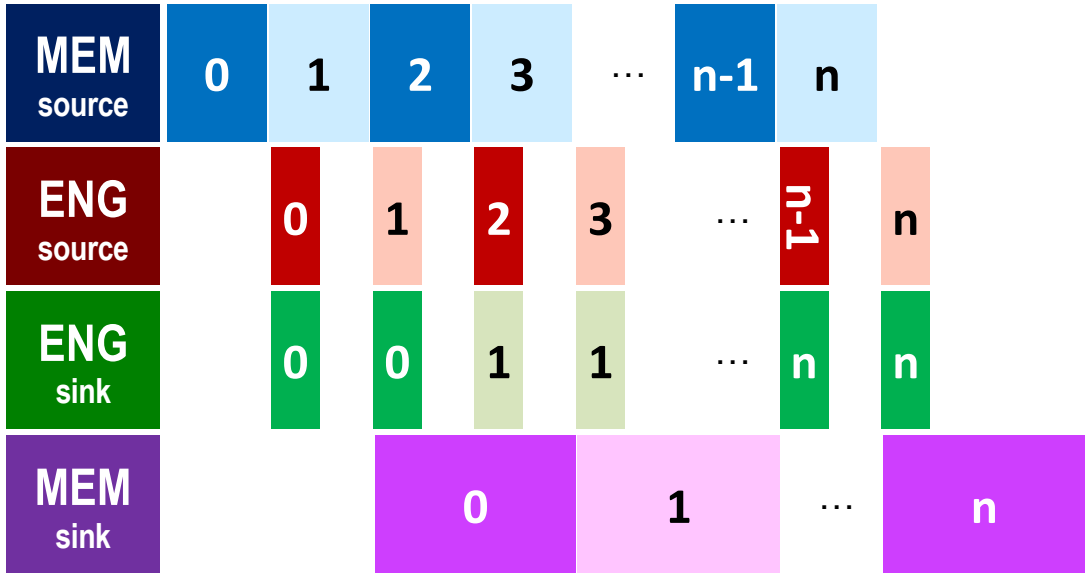


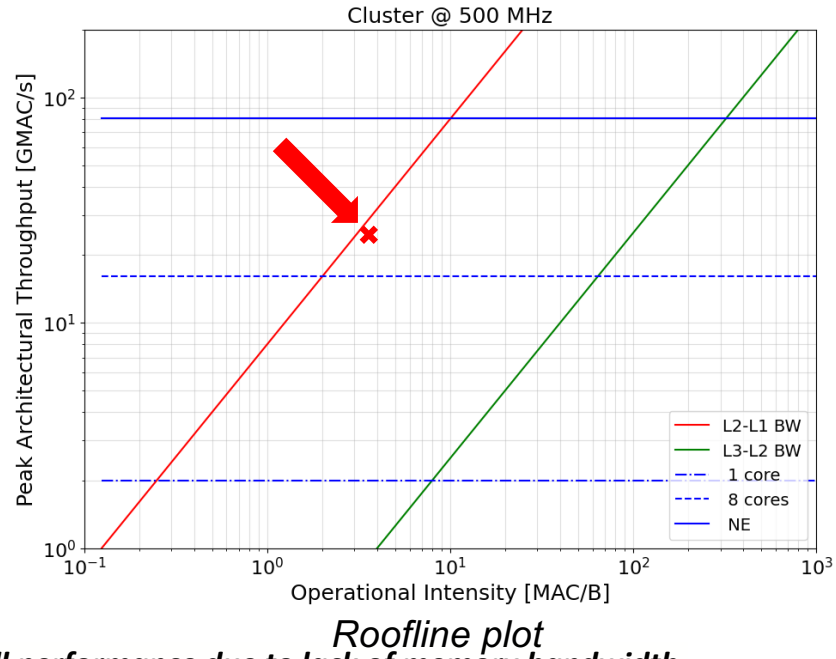
Roofline plot

- ***As we are not bound by memory, the accelerator is potentially* underdimensioned*: we can design a bigger one that will show better efficiency!***

Memory-bound HWPE

- In **memory-bound** conditions, the accelerator throughput is limited by the external memory bandwidth.
- **Performance & Efficiency: bad ☹**





Roofline plot

- ***The accelerator is* overdimensioned*: we are not capable of exploiting its full performance due to lack of memory bandwidth.***

HWPE Control

Chips-IT

Control Registers

0x00	*commit & trigger*	*wo*	starts execution on HWPE, unlocks the ctrl
0x01	*acquire*	*ro*	starts a job offload on HWPE,returns a job ID handle, locks the ctrl
0x02	*finished jobs/instructions*	*ro*	returns no. of completed jobs
0x03	*status*	*ro*	each byte returns status of a job in the queue (1=working, 0=idle)
0x04	*running job*	*ro*	returns ID of the currently running job
0x05	*soft clear*	*wo*	clears the HWPE
0x06	*offloader ID*	*ro*	each byte returns ID+1 of a job in the queue
0x07	*sw synch*	*wo*	triggers an implementation-specific internal HWPE event (0 to 7 depending on written data)
Generic Registers			
Job-Dependent Registers			

Control registers used for standard HWPE control model, common to all HWPEs

Memory-mapped, typically connected to Peripheral Interconnect or Peripheral Demux.
Executes a **queue** of **jobs** (typically 2 entries).

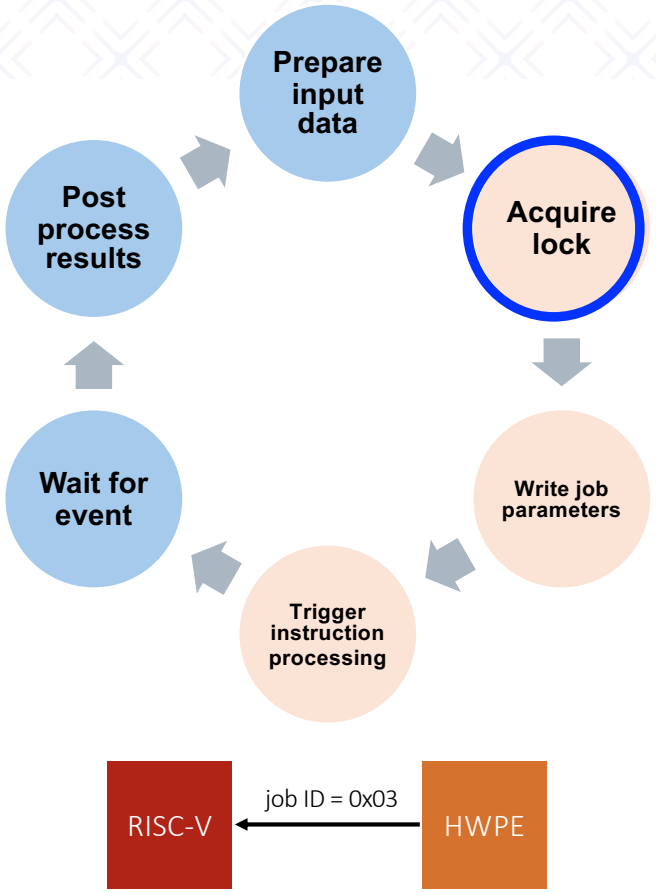
Generic registers and Job-dependent registers used for runtime parameters specific of a HW accelerator.

HWPE Control

Chips-IT

Control Registers

0x00	*commit & trigger*	*wo*	starts execution on HWPE, unlocks the ctrl
0x01	*acquire*	*ro*	starts a job offload on HWPE,returns a job ID handle, locks the ctrl
0x02	*finished jobs/instructions*	*ro*	returns no. of completed jobs
0x03	*status*	*ro*	each byte returns status of a job in the queue (1=working, 0=idle)
0x04	*running job*	*ro*	returns ID of the currently running job
0x05	*soft clear*	*wo*	clears the HWPE
0x06	*offloader ID*	*ro*	each byte returns ID+1 of a job in the queue
0x07	*sw synch*	*wo*	triggers an implementation-specific internal HWPE event (0 to 7 depending on written data)
Generic Registers			
Job-Dependent Registers			

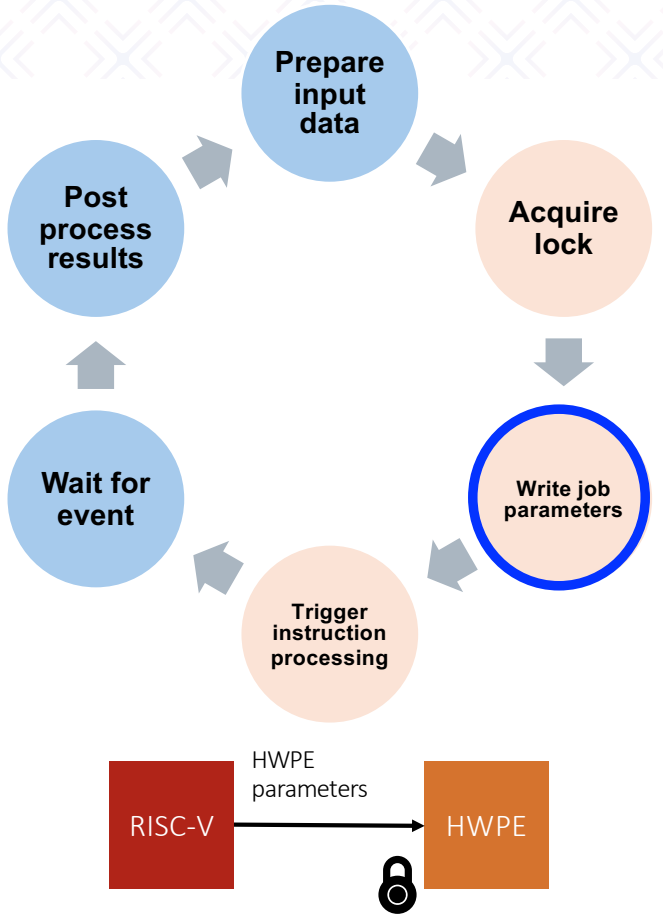


HWPE Control

Chips-IT

Control Registers

0x00	*commit & trigger*	*wo*	starts execution on HWPE, unlocks the ctrl
0x01	*acquire*	*ro*	starts a job offload on HWPE,returns a job ID handle, locks the ctrl
0x02	*finished jobs/instructions*	*ro*	returns no. of completed jobs
0x03	*status*	*ro*	each byte returns status of a job in the queue (1=working, 0=idle)
0x04	*running job*	*ro*	returns ID of the currently running job
0x05	*soft clear*	*wo*	clears the HWPE
0x06	*offloader ID*	*ro*	each byte returns ID+1 of a job in the queue
0x07	*sw synch*	*wo*	triggers an implementation-specific internal HWPE event (0 to 7 depending on written data)
Generic Registers			
Job-Dependent Registers			

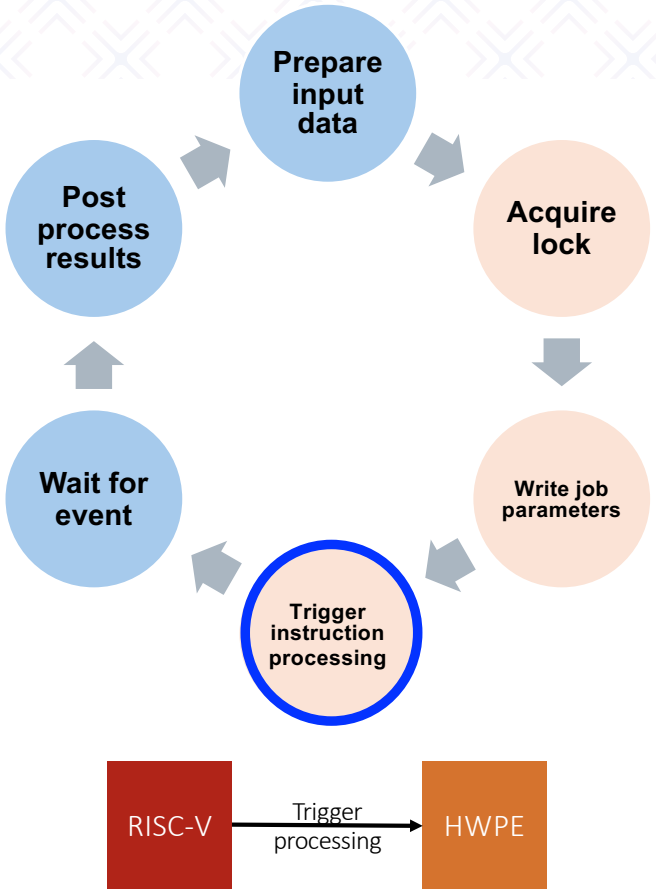


HWPE Control

Chips-IT

Control Registers

0x00	*commit & trigger*	*wo*	starts execution on HWPE, unlocks the ctrl
0x01	*acquire*	*ro*	starts a job offload on HWPE,returns a job ID handle, locks the ctrl
0x02	*finished jobs/instructions*	*ro*	returns no. of completed jobs
0x03	*status*	*ro*	each byte returns status of a job in the queue (1=working, 0=idle)
0x04	*running job*	*ro*	returns ID of the currently running job
0x05	*soft clear*	*wo*	clears the HWPE
0x06	*offloader ID*	*ro*	each byte returns ID+1 of a job in the queue
0x07	*sw synch*	*wo*	triggers an implementation-specific internal HWPE event (0 to 7 depending on written data)
Generic Registers			
Job-Dependent Registers			

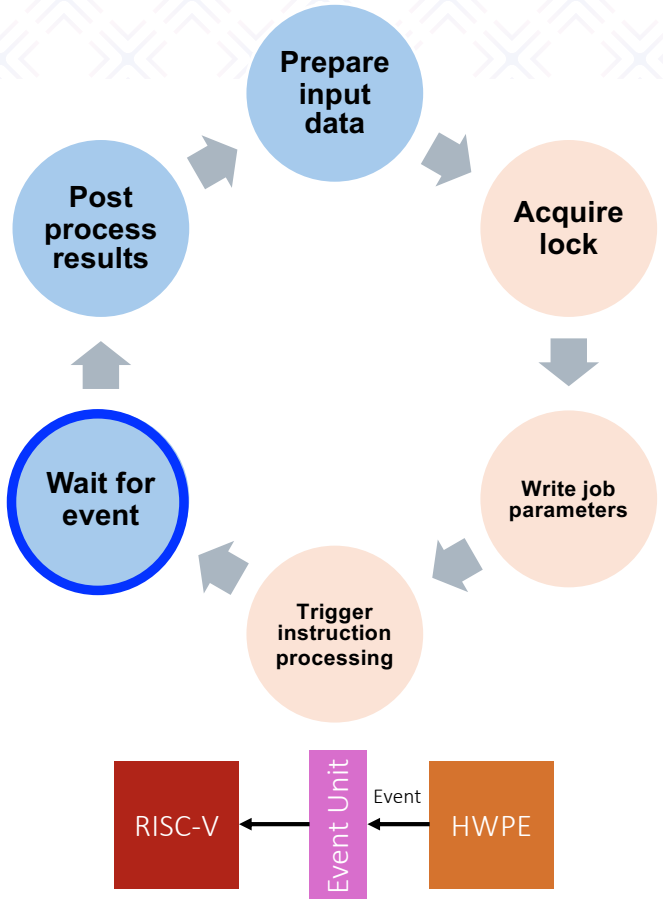


HWPE Control

Chips-IT

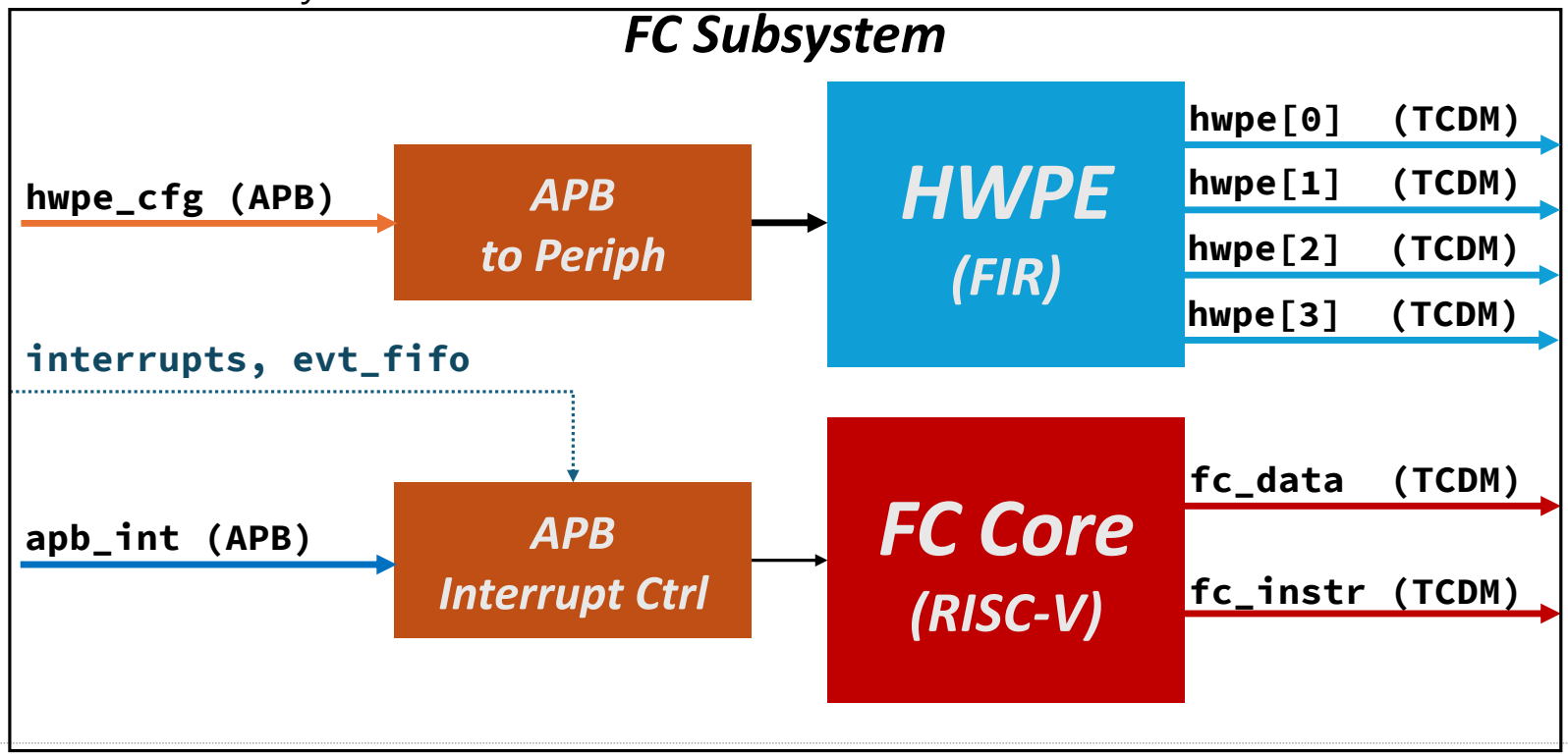
Control Registers

0x00	*commit & trigger*	*wo*	starts execution on HWPE, unlocks the ctrl
0x01	*acquire*	*ro*	starts a job offload on HWPE,returns a job ID handle, locks the ctrl
0x02	*finished jobs/instructions*	*ro*	returns no. of completed jobs
0x03	*status*	*ro*	each byte returns status of a job in the queue (1=working, 0=idle)
0x04	*running job*	*ro*	returns ID of the currently running job
0x05	*soft clear*	*wo*	clears the HWPE
0x06	*offloader ID*	*ro*	each byte returns ID+1 of a job in the queue
0x07	*sw synch*	*wo*	triggers an implementation-specific internal HWPE event (0 to 7 depending on written data)
Generic Registers			
Job-Dependent Registers			



Fabric Controller Subsystem with HWPE

- The *Fabric Controller* subsystem can contain also a HWPE



How to use a HWPE? Some pseudo-C code


```
int id = hwpe_fir_async(coeff, arr, arr_len, coeff_len); // returns immediately
// possibly do something else
hwpe_fir_wait(id); // block waiting for event
```

High-level view

How to use a HWPE? Some pseudo-C code

Chips-IT

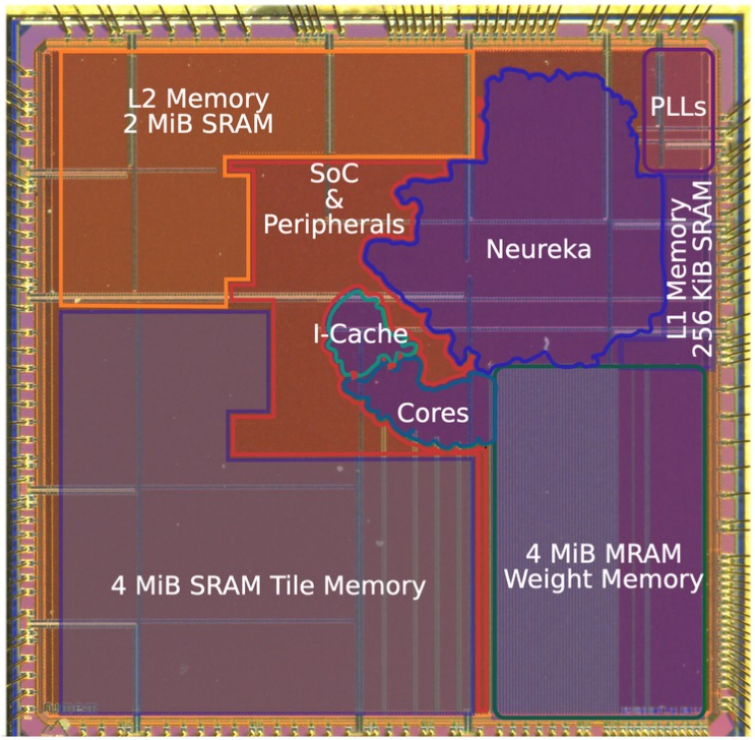
```
int id = hwpe_fir_async(coeff, arr, arr_len, coeff_len); // returns immediately
// possibly do something else
hwpe_fir_wait(id); // block waiting for event
```

High-level view

```
int hwpe_fir_async(int16_t *coeff, int16_t *arr, int arr_len, int coeff_len) {
  int id = -1;
  do {
    id = hwpe_fir_acquire();
  } while (id < 0);
  hwpe_fir_cfg_write(REG_COEFF, coeff);
  hwpe_fir_cfg_write(REG_ARR,   arr);
  hwpe_fir_cfg_write(REG_LEN,   (coeff_len << 16) | arr_len);
  hwpe_fir_trigger();
}
void hwpe_fir_wait(int id) {
  while(hwpe_fir_cfg_read(STATUS) != 0)
    WFE(); // clock-gate core and wait for event
}
```

Low-level view

Siracusa: 16nm SoC for XR



	Vega [1]	Diana [2]	Marsellus [3]	[4]	[5]	*Siracusa*
Technology	22nm FDX	22nm FDX	22nm FDX	40nm	22nm	16nm FinFET
Area	10mm²	10.24mm²	8.7mm²	25mm²	8.76mm²	16mm²
On-chip mem	1728 KB SRAM 4 MB MRAM (L3)	896 KB SRAM	1152 KB SRAM	768 KB	1428 KB	6400 KB SRAM 4 MB MRAM (L1)
Peak Perf 8b	32.2 GOPS	140 GOPS	90 GOPS	N/A	146 GOPS	**698 GOPS**
Peak Eff 8b	1.3 TOPS/W	2.07 TOPS/W	1.8 TOPS/W	0.94 TOPS/W	0.7 TOPS/W	**2.68 TOPS/W**
Peak Eff (WxAb)	1.3 TOPS/W	4.1TOPS/W (2x2b) **600 TOPS/W (analog)**	12.4 TOPS/W (2x2b)	60.6 TOPS/W (1x1b)	0.7 TOPS/W	8.84 TOPS/W (2x8b)
Area Eff	3.2 GOPS/mm²	21.2 GOPS/mm²	47.4 GOPS/mm²	N/A	58.3 GOPS/mm²	**65.2 GOPS/mm²**

[1] D. Rossi et al., JSSC'21
[2] P. Houshmand et al., JSSC'23
[3] F. Conti et al., JSSC'23
[4] M. Chang et al., ISSCC'22
[5] Q. Zhang et al., VLSI Symposium'22

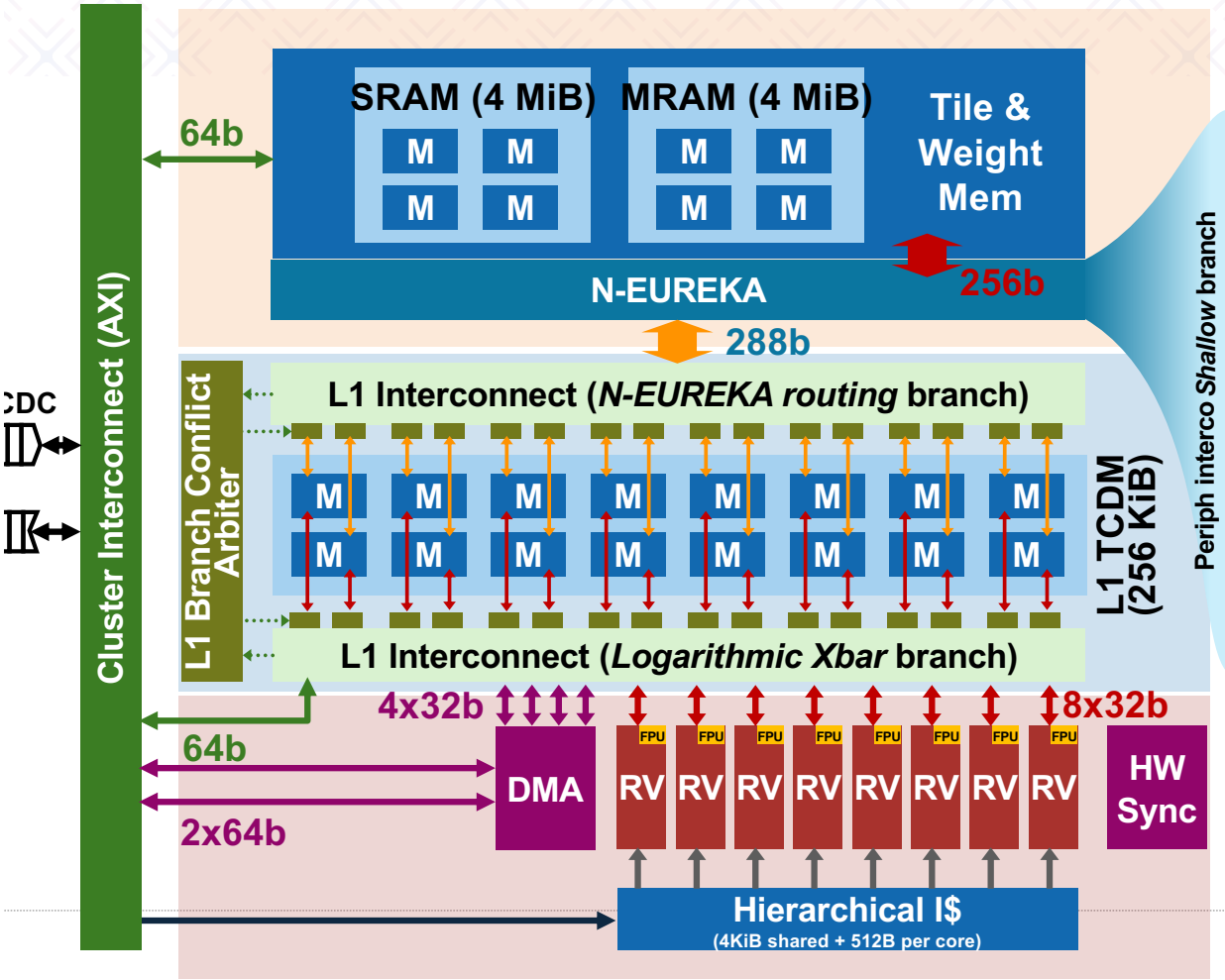
Balance efficiency, peak performance, area efficiency without compromises in precision

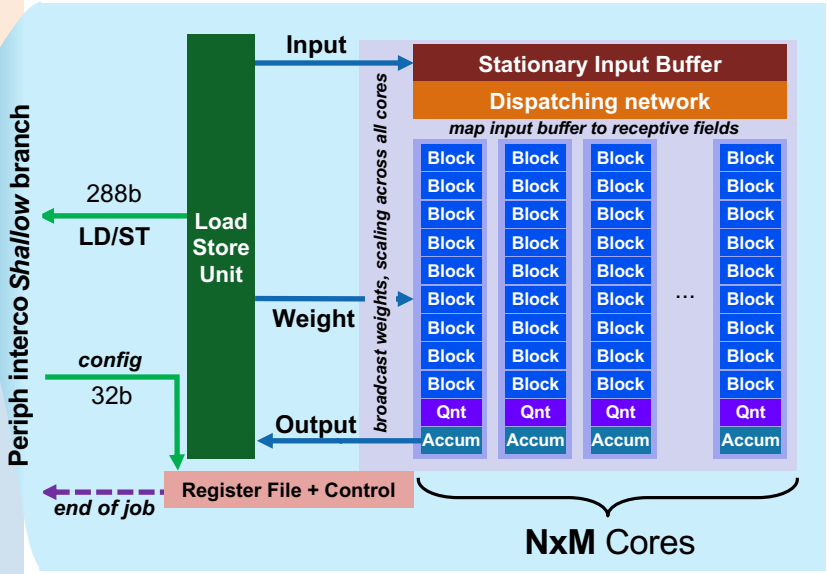
N-EUREKA 36-cores configuration

[A. Prasad et al., "Siracusa: a 16nm Heterogeneous RISC-V SoC for Extended Reality with At-MRAM Neural Engine," IEEE Journal of Solid-State Circuits (accepted)]

N-EUREKA (36-cores + MRAM)
https://arxiv.org/abs/2312.14750 ; https://github.com/pulp-platform/neureka

Siracusa compute principle





- 1 Core = receptive field of 1x1 px in output across 32 out-chans
- Output stationary, Input quasi-stationary
- Parametric number of Cores (NxM out-px)
- 8b activations, 2-8b weights

Not only academia: GAP9 with NE16

GREENWAVES TECHNOLOGIES

Chips-IT

Best-in-class in latency and energy efficiency in MLPerf Tiny 1.0!

							Benchmark Results									
							Task	Visual Wake Words		Image Classification		Keyword Spotting		Anomaly Deteciton	
							Data	Visual Wake Words Dataset		CIFAR-10		Google Speech Commands		ToyADMOS (ToyCar)	
							Model	MobileNetV1 (0.25x)		ResNet-V1		DSCNN		FC AutoEncoder	
							Accuracy	80% (top 1)		85% (top 1)		90% (top 1)		0.85 (AUC)	
Submitter	Board Name	SoC Name	Processor(s) & Number	Accelerator(s) & Number	Software	Notes	Units	Latency in ms	Energy in uJ	Latency in ms	Energy in uJ	Latency in ms	Energy in uJ	Latency in ms	Energy in uJ
Greenwaves Technologies	GAP9 EVK	GAP9	RISC-V Core (1+9)	NE16 (1)	GreenWaves GAPFlow	GAP9 (370MHZ, 0.8Vcore)		1.13	58.4	0.62	40.4	0.48	26.7	0.18	7.29
Greenwaves Technologies	GAP9 EVK	GAP9	RISC-V Core (1+9)	NE16 (1)	GreenWaves GAPFlow	GAP9 (240MHZ, 0.65Vcore)		1.73	40.8	0.95	27.7	0.73	18.6	0.27	5.25
OctoML	NRF5340DK	nRF5340	Arm® Cortex®-M33		microTVM using CMSIS-NN backend	128MHz		232.0		316.1		76.1		6.27	
OctoML	NUCLEO-L4R5ZI	STM32L4R5ZIT6U	Arm® Cortex®-M4		microTVM using CMSIS-NN backend	120MHz, 1.8Vbat		301.2	15531.4	389.5	20236.3	99.8	5230.3	8.60	443.2
OctoML	NUCLEO-L4R5ZI	STM32L4R5ZIT6U	Arm® Cortex®-M4		microTVM using native codegen	120MHz, 1.8Vbat		336.5	7131.6	389.2	21342.3	144.0	7950.5	11.7	633.7
Plumerai	B_U585I_IOT02A	STM32U585	Arm® Cortex®-M33		Plumerai Inference Engine 2022.09	160MHz		107.0		107.1		35.4		4.90	
Plumerai	CY8CPROTO-062-4343w	PSoC 62 MCU	Arm® Cortex®-M4		Plumerai Inference Engine 2022.09	150MHz		192.5		193.1		61.4		6.70	
Plumerai	DISCO-F746NG	STM32F746	Arm® Cortex®-M7		Plumerai Inference Engine 2022.09	216MHz		57.0		64.8		19.1		2.30	
Plumerai	NUCLEO-L4R5ZI	STM32L4R5ZIT6U	Arm® Cortex®-M4		Plumerai Inference Engine 2022.09	120MHz		208.6		173.2		71.7		5.60	
Silicon Labs	xG24-DK2601B	EFR32MG24	Arm® Cortex®-M33	Silicon Labs MVP(1)	TensorFlowLite for Microcontrollers, CMSIS-NN, Silicon Labs Gecko SDK			111.6	1139.2	120.9	1234.7	36.3	401.9	5.43	47.3
STMicroelectronics	NUCLEO-H7A3ZI-Q	STM32H7A3ZIT6Q	Arm® Cortex®-M7		X-CUBE-AI v7.3.0	280MHz, 3.3Vbat		50.7	7978.3	54.3	8707.3	16.8	2721.8	1.82	266.5
STMicroelectronics	NUCLEO-L4R5ZI	STM32L4R5ZIT6U	Arm® Cortex®-M4		X-CUBE-AI v7.3.0	120MHz, 1.8Vbat		230.5	10066.6	226.9	10681.6	75.1	3371.7	7.57	323.0
STMicroelectronics	NUCLEO-U575ZI-Q	STM32U575ZIT6Q	Arm® Cortex®-M33		X-CUBE-AI v7.3.0	160MHz, 1.8Vbat		133.4	3364.5	139.7	3642.0	44.2	1138.5	4.84	119.1
Syntiant	NDP9120-EVL	NDP120	M0 + HiFi	Syntiant Core 2 (98MH	Syntiant TDK	Syntiant Core 2 (98MHz, 1		4.10	97.2	5.12	139.4	1.48	43.8		
Syntiant	NDP9120-EVL	NDP120	M0 + HiFi	Syntiant Core 2 (30MH	Syntiant TDK	Syntiant Core 2 (30MHz, 0		12.7	71.7	16.0	101.8	4.37	31.5		
		Next													

NE16 (9-cores)
https://github.com/pulp-platform/ne16

Latency 1.73ms (against ~29ms on GAP8)
Energy 41uJ (against 1450uJ on GAP8)

Bibliography (accelerators)

- **NEureka:** A. S. Prasad, L. Benini, and F. Conti, “Specialization meets Flexibility: a Heterogeneous Architecture for High-Efficiency, High-flexibility AR/VR Processing,” in 2023 60th ACM/IEEE Design Automation Conference (DAC), DAC 2023. [IEEE]
- **RedMulE:** Y. Tortorella, L. Bertaccini, D. Rossi, L. Benini, and F. Conti, “RedMulE: A Compact FP16 Matrix-Multiplication Accelerator for Adaptive Deep Learning on RISC-V-Based Ultra-Low-Power SoCs,” in Design, Automation & Test in Europe Conference & Exhibition, DATE 2022. [arXiv extension]
- **IMA:** A. Garofalo, G. Ottavi, F. Conti, G. Karunaratne, I. Boybat, L. Benini, and D. Rossi, “A Heterogeneous In-Memory Computing Cluster for Flexible End-to-End Inference of Real-World Deep Neural Networks,” in IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 12, no. 2, pp. 422-435, June 2022, doi: 10.1109/JETCAS.2022.3170152. [arXiv]
- **FFT:** L. Bertaccini, L. Benini, and F. Conti, “To Buffer, or Not to Buffer? A Case Study on FFTAccelerators for Ultra-Low-Power Multicore Clusters”, in IEEE International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2021. [IEEE]
- **XNE:** F. Conti, P. D. Schiavone, and L. Benini, “XNOR Neural Engine: A Hardware Accelerator IP for 21.6-fJ/op Binary Neural Network Inference.,” IEEE Trans. Comput. Aided Des. Integr. Circuits Syst., vol. 37, no. 11, pp. 2940–2951, 2018, doi: 10.1109/TCAD.2018.2857019 (ESWEEK CODES+ISSS 2018 Best Paper Award). [arXiv]
- **HWCE:** F. Conti and L. Benini, “A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters.,” in Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015, 2015, pp. 683–688, [ACM]

Bibliography (chips)

- **Siracusa:** A. S. Prasad, M. Scherer, F. Conti, D. Rossi, A. Di Mauro, M. Eggimann, J. T. Gomez, Z. Li, S. S. Sarwar, Z. Wang, B. De Salvo, and L. Benini, "Siracusa: A 16 nm Heterogenous RISC-V SoC for Extended Reality with At-MRAM Neural Engine." [arXiv]
- **Marsellus:** F. Conti, G. Paulin, A. Garofalo, D. Rossi, A. Di Mauro, G. Rutishauser, G. Ottavi, M. Eggimann, H. Okuhara, and L. Benini, "Marsellus: A Heterogeneous RISC-V AI-IoT End-Node SoC with 2-to-8b DNN Acceleration and 30%-Boost Adaptive Body Biasing," in IEEE Journal of Solid-State Circuits, doi: 10.1109/JSSC.2023.3318301. [arXiv]
- **Darkside:** A. Garofalo, Y. Tortorella, M. Perotti, L. Valente, A. Nadalini, L. Benini, D. Rossi, and F. Conti, "DARKSIDE: A Heterogeneous RISC-V Compute Cluster for Extreme-Edge On-Chip DNN Inference and Training," in IEEE Open Journal of the Solid-State Circuits Society, vol. 2, pp. 231-243, 2022, doi: 10.1109/OJSSCS.2022.3210082. [open access]
- **Vega:** D. Rossi, F. Conti, M. Eggimann, A. Di Mauro, G. Tagliavini, S. Mach, M. Guermandi, A. Pullini, I. Loi, J. Chen, E. Flamand, and L. Benini, "Vega: A 10-Core SoC for IoT End-Nodes with DNN Acceleration and Cognitive Wake-Up from MRAM-Based State-Retentive Sleep Mode," in IEEE Journal on Solid-State Circuits. [arXiv]
- **Quentin:** A. Di Mauro, F. Conti, P. D. Schiavone, D. Rossi, and L. Benini, "Always-On 674uW @ 4GOP/s Error Resilient Binary Neural Networks With Aggressive SRAM Voltage Scaling on a 22-nm IoT End-Node.," IEEE Trans. Circuits Syst., vol. 67–I, no. 11, pp. 3905–3918, 2020, doi: 10.1109/TCSI.2020.3012576.
- **Fulmine:** F. Conti et al., "An IoT Endpoint System-on-Chip for Secure and Energy-Efficient Near-Sensor Analytics.," IEEE Trans. Circuits Syst. I Regul. Pap., vol. 64–I, no. 9, pp. 2481–2494, 2017, doi: 10.1109/TCSI.2017.2698019 (IEEE CAS Darlington Award 2020). [arXiv]

Hands-on time again!

Chips-IT

- Today and (partly) tomorrow we will dirty our hands with the guided design of a HWPE following up our FIR example

1. Design the HWPE unit in SystemVerilog
   - We provide you with a partial HWPE in its own standalone testbenches (one for the datapath alone; another for Ibex + HWPE + dummy memory)
   - You will implement and test part of the accelerator's datapath, streamer, and controller
2. Integrate the HWPE unit into PULPissimo
   - You will need to learn how to modify the PULPissimo system to insert an HWPE
3. Adapt the FIR filter application and check the speedup
   - Again, we can compare with yesterday's work and our baseline from Monday: how much faster are we (if any)?

Thank you

Chips-IT