KALRAY
THE POWER OF MORE

# RISC AND VLIW ARCHITECTURES

ARCHI 2025

March 10 – 14 2025

Benoît Dupont de Dinechin,
Chief Technology Officer

# AGENDA

1. Kalray MPPA Platform
2. RISC Architectures
3. VLIW Architectures
4. RISC-V Accelerators
5. Converging to RISC-V
6. Conclusions

# ACCELERATING DATA-INTENSIVE APPLICATIONS

## An Easy-to-Use Complete Open Environment

### AccessCore®

Open Software
& Tools
Multi-configuration

Customer Code (Processing/Application)

Open & Standard APIs (C/C++, SPDK, OpenCL, …)

Compute Framework | Storage Framework | Networking Framework | Linux

AccessCore® Middleware

Single Software Architecture & Unified Toolchain

### MPPA® DPU

Manycore Processors


MPPA®3 DPU - COOLIDGE™

### PCIe Card

K300™

TC4™

**2 modes**

**Stand-alone**
no Host

**Acceleration**
(x86 or ARM as a host)

### Systems

**MPPA®-DEV5**
Development platform

**Your own system**
Workstation or server

**Kalray FlashBox™**
Unleash the Full Potential of SSD Storage

# KALRAY MPPA® SCALABLE MANY-CORE ACCELERATORS

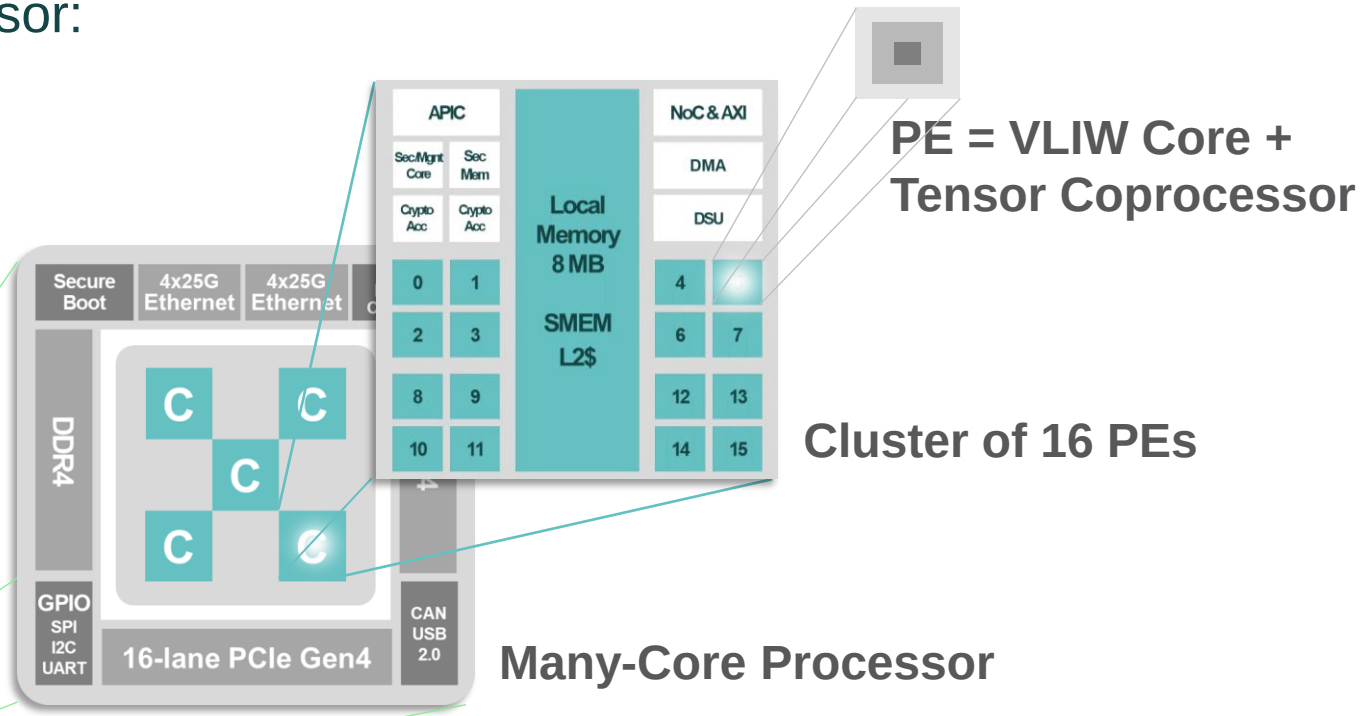
3rd-gen MPPA® processor manufactured in TSMC 16nm technology, running at up to 1.2 GHz

MPPA3 ‘Coolidge’ v2 processor back from TSMC mid-2023

4× MPPA processors with 80 PEs per processor:

- 4× 49 TOPS INT8.32
- 4× 24.5 TFLOPS FP16.32
- 4× 1.5 TFLOPS FP32


Multiple Processors per Card




PE = VLIW Core + Tensor Coprocessor

Cluster of 16 PEs

Many-Core Processor

# ACCESSCORE® COMPUTE SDK

## High-Performance Programming Models

STANDARD PROGRAMMING ENVIRONMENTS

### OPENCL 1.2 Programming

Standard accelerator programming model

- POSIX host CPU accelerated by MPPA device (OpenAMP interface)
- OpenCL 1.2 conformance based on POCL and LLVM for OpenCL-C

OpenCL offloading modes:

- Linearized Work Items on a PE (LWI)
- Single Program Multiple Data (SPMD)
- Native functions called from kernels

### C/C++ POSIX Programming

Standard multicore programming model

- MPPA Linux and ClusterOS
- Standard C/C++ programming
- POSIX threads interface
- GCC and LLVM OpenMP support

Exposed MPPA® communications

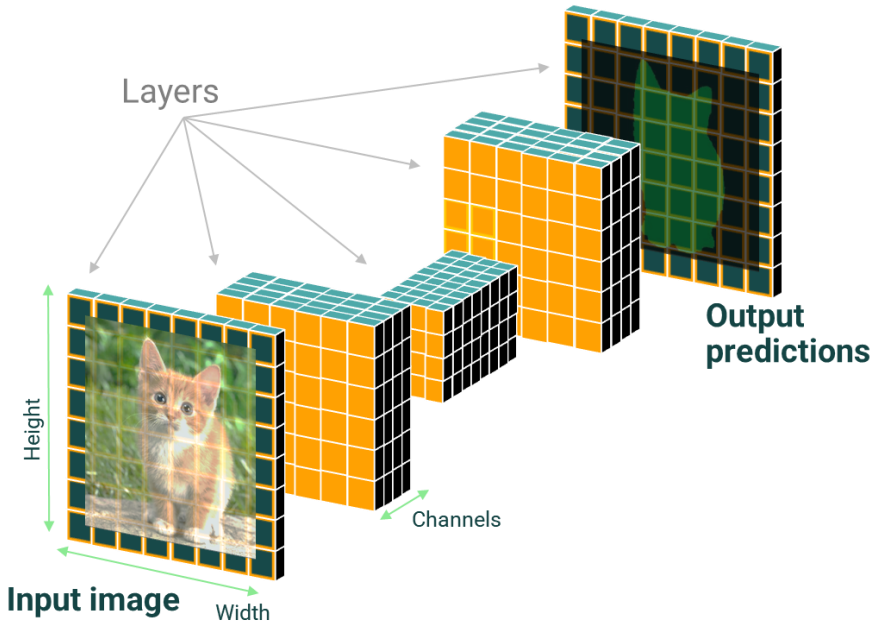
- RDMA using the MPPA Asynchronous Communication library (mppa_async)
- OpenAMP Open Standard and APIs for Asymmetric Multiprocessing Systems

# KANN™ (KALRAY NEURAL NETWORK)

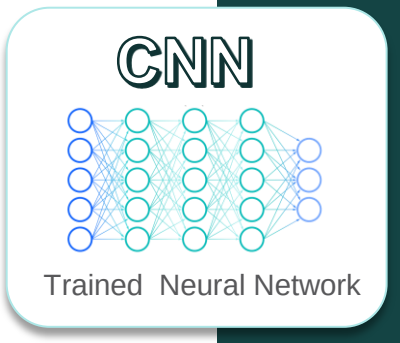
Inference compiler for convolutional neural networks

Leverage the OpenCL partition into sub-devices

On MPPA® processors, execution of each layer is parallelized and distributed on all available clusters for a low latency inference




TensorFlow TensorFlow Lite ONNX




## KaNN™

**Graph Optim.**

**Code Generator**
Scheduling and mapping of optimized Neural Network

MPPA® Platform

Video Sources

**Input data**
- image,
- camera,
- Lidar, …

Output, Display

**Results**
- Classification,
- Object detection,
- Segmentation mask, …

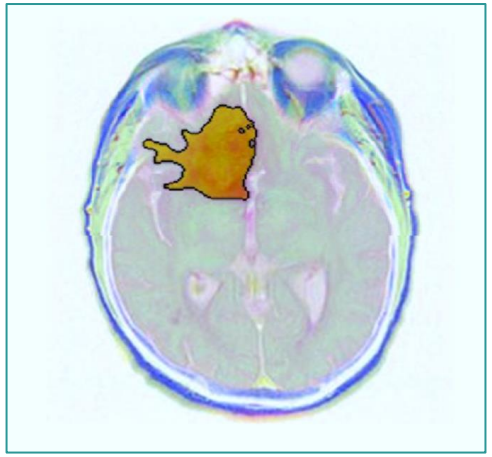
# AI INFERENCE AND DSP PROCESSING FOR EDGE COMPUTING

## Customer measured x3.8 better compute efficiency than the market leader

### Kalray TC4™ vs competition compute efficiency on UNet* model inference

| | TC4™ Board (4 x MPPA® processors) | | | High-end GPU | | |
|---|---|---|---|---|---|---|
| | Peak capabilities | Performance | Efficiency | Peak capabilities | Performance | Efficiency |
| **FP16** | 107 TFLOPs | 6 500 FPS | **42 %** | 312 TFLOPs | 5200 FPS | **11.5 %** |

*UNet complexity = 6.93 GFLOPs / frame



**UNet** predictions example on MRI brain tumor dataset

| | NN Data type | FPS TC4™ \| PCIe | FPS (for 1 Watt) | FPS (for 1💰) |
|---|---|---|---|---|
| MPPA® Coolidge V2 | FP16 | 5,600 | 18.7 | 1.4 |
| | INT8 | 8,400 | 30.5 | 2.1 |
| **High-end GPU** | FP16 | 5,200 | 17.3 | 0.27 |
| | INT8 | 5,950 | 19.8 | 0.31 |

*based on 300W consumption / PCIe card

**x3 to x6**
**Perf/$**
**vs Comp.**

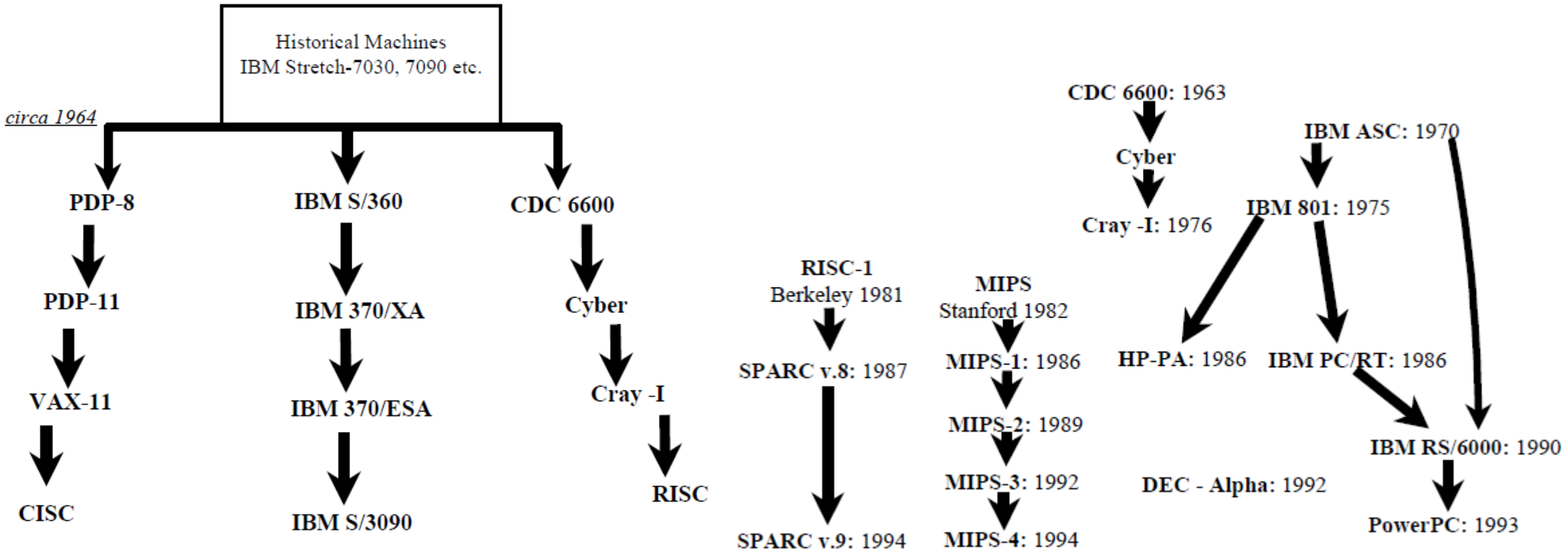
**x3.8**
**Efficiency**
**vs Comp.**

# AGENDA

1. Kalray MPPA Platform
2. RISC Architectures
3. VLIW Architectures
4. RISC-V Accelerators
5. Converging to RISC-V
6. Conclusions

# EARLY COMPUTER ARCHITECTURE EVOLUTION

Load/Store ISA on the CDC6600, extended to vector registers on the Cray-1



# ORIGINS OF RISC: THE IBM 801 MINICOMPUTER

IBM project started in 1974 to design a telephone switching machine capable of 12 MIPS

“Imposing microcode between a computer and its users imposes an expensive overhead in performing the most frequently executed instructions.” [Cocke & Markstein 1990 IBM J. Research & Development]

The most important features which contributed to its low cost/performance ratio were:

1. Separate instruction and data caches, allowing a much higher bandwidth between memory and CPU;
2. No arithmetic operations to storage, which greatly simplified the pipeline;
3. Uniform instruction length and simplicity of design, making possible a very short cycle time: ten levels of logic. (For example, all register-to-register operations executed in one cycle.)

A major advance was its ability to branch based on the state of any bit in any general purpose register.

A second form of branch, which is commonly called "delayed branch," caused the CPU to unconditionally execute the instruction immediately following the branch, whether or not the branch was successful.

Compilers were expected to play a central role in the 801.

- The antithesis of the "semantic gap“ idea, in that instructions were specifically designed for efficient use by a compiler.
- Approach to register allocation, which was deemed to be central to the proper use of the 801, was "graph coloring." This approach had been mentioned in the literature, but was implemented for the first time in the PL.8 compiler.

First ISA with 2-operand, 24-bit instructions. Second ISA with 3-operand, 32-bit instructions

# CLASSIC RISC ARCHITECTURE FEATURES

## Successors of IBM 801, Berkeley RISC-I, Stanford MIPS

### Berkeley RISC-1 (1981)

- Read as zero register zero, inherited from the CDC 6600
- Six register windows containing 14 registers. Of those 14 registers, 4 were overlapped from the prior window
- All other features similar to those of the IBM 801, including delayed branches

### Stanford MIPS (1982)

- Similar to RISC-1, except for not including the overlapped register windows
- No pipeline stall on register dependencies « Microprocessor without Interlocked Pipeline Stages »

### IBM RS/6000 (1990)

- Floating-point Fused Multiply-Add (FMA) instructions
- Base register + index register memory addressing
- Conditional branch with “decrement and test CTR” option
- FPU register remapping allows independent sequences using the same architected register to be processed concurrently

# EVOLUTION OF RISC FEATURES

## RISC extension by CPU vendors

### MIPS-III / R4000 (1991)

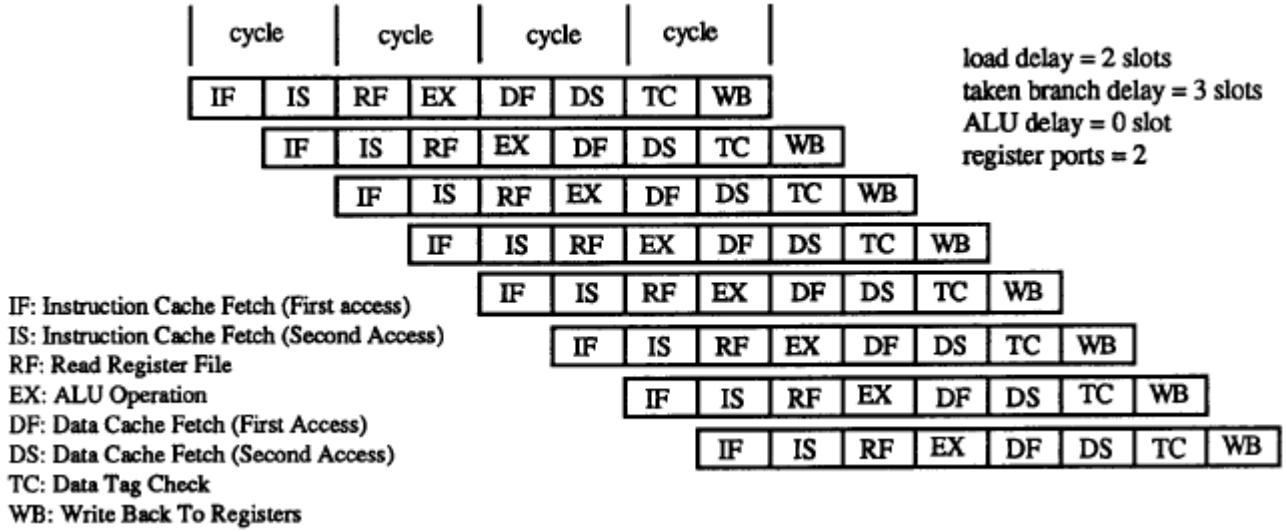
- Superpipelined implementation

### DEC Alpha 21064 (1992)

- Superscalar in-order (dual issue)
- Weakest memory consistency model

### ARM ARM7 TDMI (1994)

- Thumb ISA encoded in 16-bit words




### HP PA-RISC 1.1 (1994)

- MAX (Multimedia Acceleration eXtensions) to accelerate MPEG decoding (Pentium P5 introduced MMX in 1997)

### MIPS-IV / R10000 (1995)

- Superscalar out-of-order execution (quad issue)

### DEC Alpha 21464 (Planned 2004)

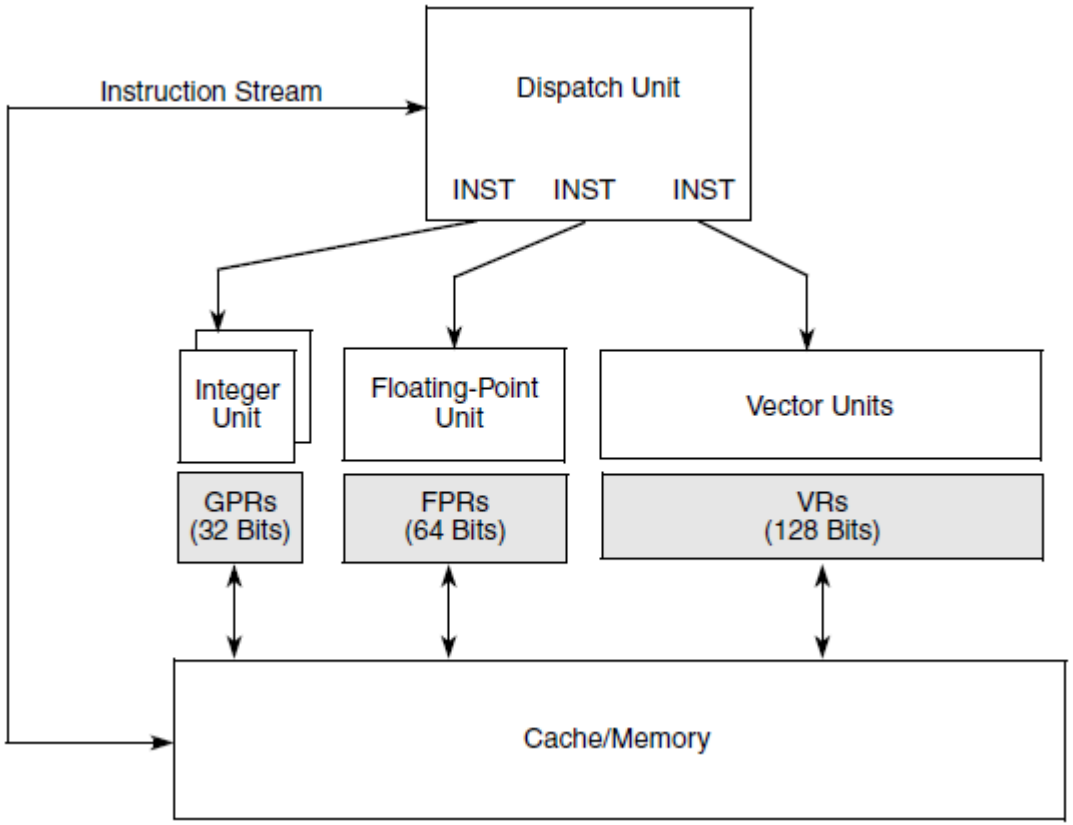
- Out-of-order execution with 4x simultaneous multi-threading

# IBM FROM ALTIVEC TO THE VECTOR SCALAR ARCHITECTURE

Integrated vector and floating-point architecture first introduced in the POWER7 processor

Restructure the core microarchitecture by integrating and sharing register files and execution units of the previously separate scalar floating-point and AltiVec units [Gschwind 2016 IBM JRD]

AltiVec is a floating point and integer 128-bit SIMD instruction set designed and owned by Apple, IBM and Freescale Semiconductor.




Code generation for the Cell SPE demonstrated the advantages of a unified register file:

- The VS register file with 64 vector-scalar registers (VSR) is created by extending the 32 floating-point registers (FPR) to 128 bit and combining them with 32 AltiVec registers (VR)
- Many applications use either scalar floating-point or vector types, but not both. These applications experience a 2× increase of usable registers by being able to allocate all registers to their dominant data types.
- Vectorized code with a unified register file is more efficient when vectorized code reads or writes scalar operands, compared to designs that require long latency transfers between distinct register files and processing units.

# RISC-V BACKGROUND

## RISC-V ISA and Extensions

RISC-V Label

- Implement one of RV* Base ISA
- No requirements for Privileged ISA
- Scarce open-source software for extensions beyond MAFDCB

RISC-V and AI

- Most AI compilers are proprietary
- IME and AME proposed extensions

CVA6 ISA

- RV64I Base ISA
- MAFDCB Extensions

KV4 Functional Equivalence

- RV64I Base ISA
- MAFDB + Zfh Extensions (no C)

| Name | Description | Version | Status | Instruction Count |
|---|---|---|---|---|
| RV32I | Base Integer Instruction Set - 32-bit | 2.1 | Frozen | 49 |
| RV32E | Base Integer Instruction Set (embedded) - 32-bit, 16 registers | 1.9 | Open | Same as RV32I |
| RV64I | Base Integer Instruction Set - 64-bit | 2.0 | Frozen | 14 |
| RV128I | Base Integer Instruction Set - 128-bit | 1.7 | Open | 14 |
| **Extension** | | | | |
| M | Standard Extension for Integer Multiplication and Division | 2.0 | Frozen | 8 |
| A | Standard Extension for Atomic Instructions | 2.0 | Frozen | 11 |
| F | Standard Extension for Single-Precision Floating-Point | 2.0 | Frozen | 25 |
| D | Standard Extension for Double-Precision Floating-Point | 2.0 | Frozen | 25 |
| G | Shorthand for the base and above extensions | n/a | n/a | n/a |
| Q | Standard Extension for Quad-Precision Floating-Point | 2.0 | Frozen | 27 |
| L | Standard Extension for Decimal Floating-Point | 0.0 | Open | Undefined Yet |
| C | Standard Extension for Compressed Instructions | 2.0 | Frozen | 36 |
| B | Standard Extension for Bit Manipulation | 0.90 | Open | 42 |
| J | Standard Extension for Dynamically Translated Languages | 0.0 | Open | Undefined Yet |
| T | Standard Extension for Transactional Memory | 0.0 | Open | Undefined Yet |
| P | Standard Extension for Packed-SIMD Instructions | 0.1 | Open | Undefined Yet |
| V | Standard Extension for Vector Operations | 1.0 | Frozen | 186 |

# RISC-V “SIMD INSTRUCTIONS CONSIDERED HARMFUL”

An older and, in our opinion, more elegant alternative to exploit data-level parallelism is vector architectures [Patterson & Waterman 2017]

- SIMD starts off innocently enough. An architect partitions the existing 64-bit registers and ALU into many 8-, 16-, or 32-bit pieces and then computes on them in parallel.
- To accelerate SIMD, architects subsequently double the width of the registers to compute more partitions concurrently.
- Because ISAs traditionally embrace backwards binary compatibility, and the opcode specifies the data width, expanding the SIMD registers also expands the SIMD instruction set.
- Each subsequent step of doubling the width of SIMD registers and the number of SIMD instructions leads ISAs down the path of escalating complexity.
- The widest data for RISC-V is 64 bits, and today’s vector processors typically execute two, four, or eight 64-bit elements per clock cycle.

```
void daxpy(size_t n, double a, const double x[], double y[])
{
 for (size_t i = 0; i < n; i++) {
   y[i] = a*x[i] + y[i];
 }
}
```

| *ISA* | *MIPS-32 MSA* | *IA-32 AVX2* | *RV32V* |
|---|---|---|---|
| Instructions (static) | 22 | 29 | 13 |
| Instructions per Main Loop | 7 | 6 | 10 |
| Bookkeeping Instructions | 15 | 23 | 3 |
| Results per Main Loop | 2 | 4 | 64 |
| Instructions (dynamic n=1000) | 3511 | 1517 | 163 |

# RISC-V VECTOR EXTENSION (RVV)

Gather objects from main memory and put them into long, sequential vector registers

Element operations are independent by definition, and so a processor could theoretically compute all of them simultaneously

A vector processor with N 64-bit elements per register also computes on vectors with 2N 32-bit, 4N 16-bit, and 8N 8-bit elements

- RVV defines 32 vector registers of size **VLEN** bits (**VLEN** is vector register bit size, e.g. 512) with elements of maximum bit size **ELEN** ≥ 32 or 64
- Vector registers are divided in elements whose bit size is a power of two ≥ 8 and ≤ **ELEN**
- The **vtype** register contains fields **sew** (standard element width) and **lmul** (length multiplier)

  8 ≤ **sew** ≤ **ELEN** and **lmul** ∈ { 1/8, ¼, ½, 1, 2, 4, 8 }

  Setting **vl** > 1 groups vector registers as single operands of vector operations

  **vlmax** = **(VLEN / sew)** × **lmul** is the maximum number of elements operated per vector instruction
- The **vl** register specifies the count of vector elements operated by subsequent vector instructions

  0 ≤ **vl** ≤ **vlmax**
- Vector operations with masking use the **v0** register as a vector of predicates (one bit per element)

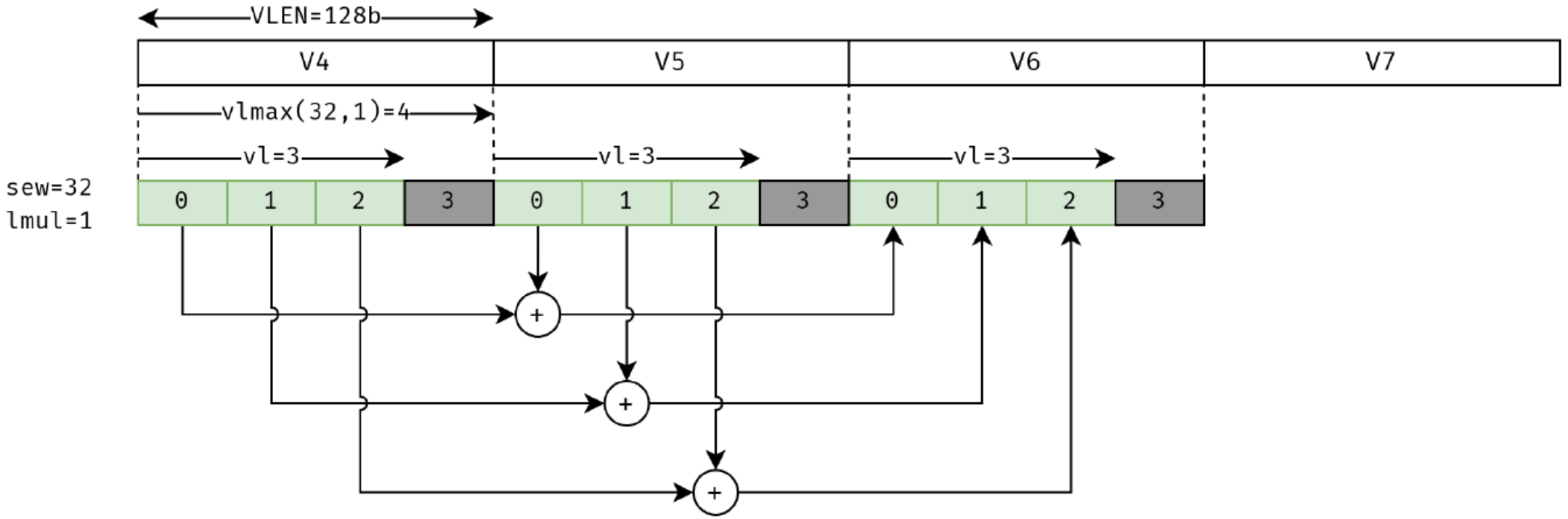
# RISC-V VECTOR INSTRUCTIONS IN PRACTICE [FERRER 2022 BSC]

RISC-V Vector operation arguments and parameters cannot be encoded in 32-bit instructions

The CPU state is configured with **vsetvl** / **vsetvli** / **vsetivli** instructions (“set vector length”)

First source of **vsetvl*** is application vector length, destination gets the hardware selected **vl**

```
vsetivli x10, 3, e32,m1,ta,ma   # vl ← 3, sew ← 32, lmul ← 1
vadd.vv v6, v4, v5
```



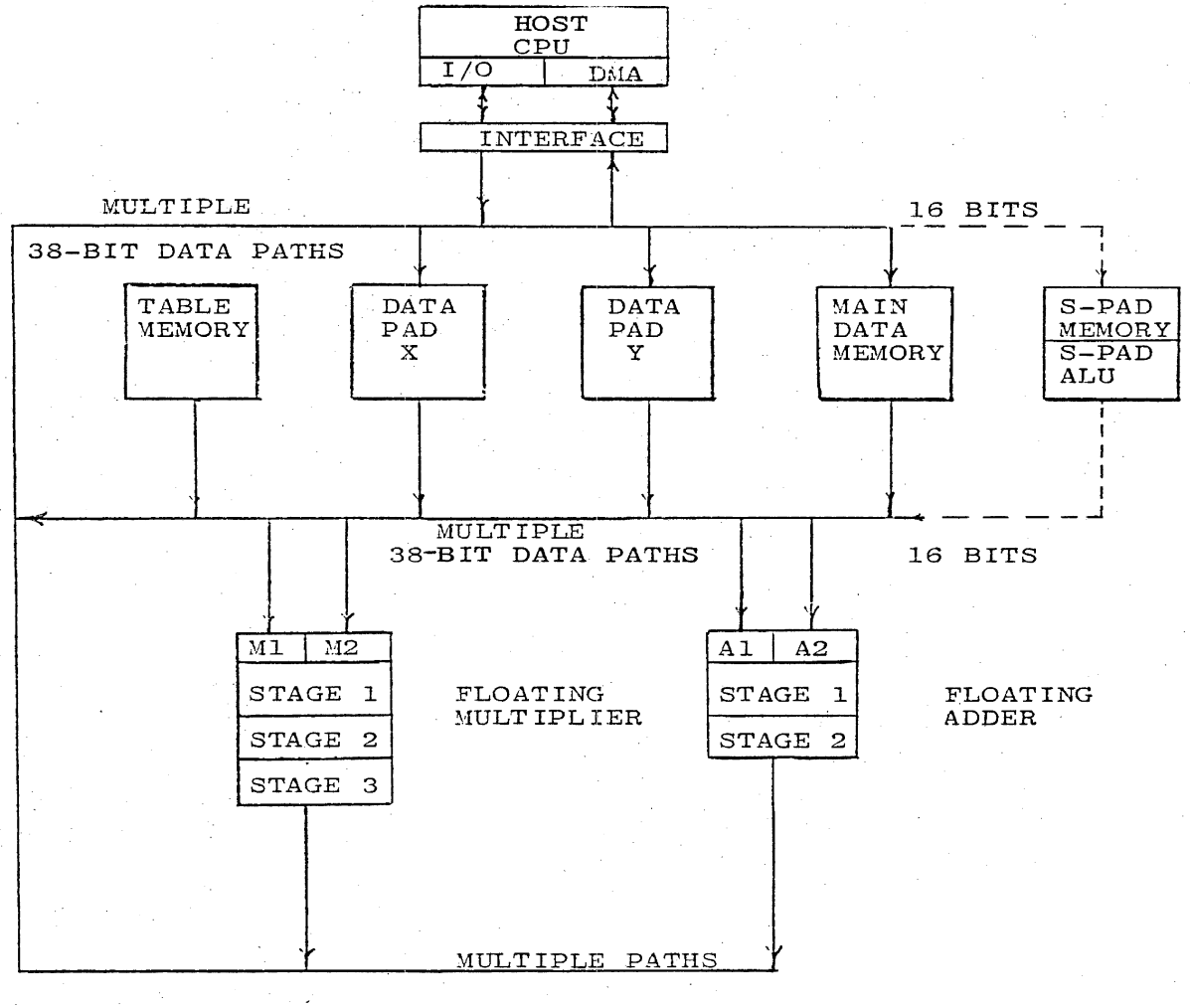
# AGENDA

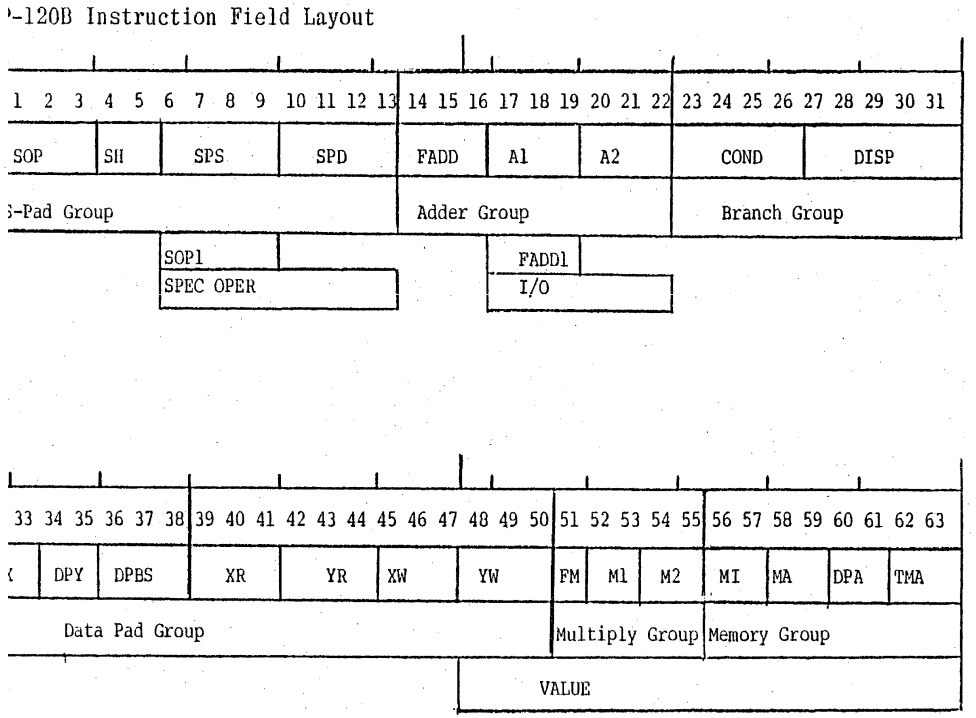
1. Kalray MPPA Platform
2. RISC Architectures
3. VLIW Architectures
4. RISC-V Accelerators
5. Converging to RISC-V
6. Conclusions

# ORIGIN OF VLIW: THE FPS AP-120B “ARRAY PROCESSOR” (1975)

Programmed with “horizontal microcode” 64-bit words to implement loop software pipelining

Data Pad X / Y accessed relative to the DPA  register are the first “rotating register files”



# SOFTWARE PIPELINING EXAMPLE ON ARM A53 (1)

‘SAXPY’ loop from BLAS

Need to use [restrict] to inform compiler there are no data dependences between the memory accesses

```
#include "math.h"

void
saxpy(int n, double a, double x[restrict],
      double y[restrict], double z[restrict])
{
  for (int i = 0; i < n; i++) {
    z[i] = a * x[i] + y[i];
  }
}
```

Compilation with GCC –O2 on a ARM workstation (cortex A53 cores)

Loop: load x[i], load y[i], FMA, store z[i], update loop counter, loop branch back

```
    .type   saxpy, %function
saxpy:
    cmp     w0, 0
    ble     .L1
    mov     x4, 0
    .p2align 3
.L3:
    ldr     d1, [x1, x4, lsl 3]
    ldr     d2, [x2, x4, lsl 3]
    fmadd     d1, d1, d0, d2
    str     d1, [x3, x4, lsl 3]
    add     x4, x4, 1
    cmp     w0, w4
    bgt     .L3
.L1:
    ret
    .size   saxpy, .-saxpy
```

# SOFTWARE PIPELINING EXAMPLE ON ARM A53 (2)

Loop schedule, no pipelining

One iteration per 11 clock cycles

| Cycle | LSU | FPU | Other |
|---|---|---|---|
| 0 | v1=[x4*0x8+x1] | | |
| 1 | v2=[x4+0x8+x2] | | |
| 2 | | | |
| 3 | | v1={v1*v0+v2} | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | [x4*0x8+x3]=v1 | | x4=x4+0x1 |
| 9 | | | cc=cmp(x0,x4) |
| 10 | | | pc={(cc>0)?Loop:pc} |

Software pipelined loop

One iteration started every 3 clock cycles

| Cycle | LSU | FPU | Other |
|---|---|---|---|
| 0 | v1=[x4*0x8+x1] | | |
| 1 | v2=[x4+0x8+x2] | | |
| 2 | | | |
| 3 | v1'=[x4'*0x8+x1] | v1={v1*v0+v2} | |
| 4 | v2'=[x4'+0x8+x2] | | |
| 5 | | | |
| 6 | v1''=[x4''*0x8+x1] | v1'={v1'*v0'+v2'} | |
| 7 | v2''=[x4''+0x8+x2] | | |
| 3*N+5 | [x4*0x8+x3]=v1 | | x4=x4+0x1 |
| 3*N+6 | v1'''=[x4'''*0x8+x1] | v1''={v1''*v0''+v2''} | cc=cmp(x0,x4) |
| 3*N+7 | v2'''=[x4'''+0x8+x2] | | pc={(cc>0)?Loop:pc} |
| … | [x4'*0x8+x3]=v1' | | x4'=x4'+0x1 |
| … | | | cc=cmp(x0,x4') |
| … | | | pc={(cc>0)?Loop:pc} |
| … | | | x4''=x4''+0x1 |
| … | | | cc=cmp(x0,x4'') |
| … | | | pc={(cc>0)?Loop:pc} |

pipelined loop prolog

pipelined loop kernel

pipelined loop epilog

In order to software pipeline, the loop temporary variables must be ‘modulo expanded’

Modulo expansion consumes registers and also requires pipelined loop kernel unrolling

Modulo expansion can be omitted on CPUs with hardware renaming or rotating registers

# MULTIFLOW TRACE 7 SERIES

Designed as a target for a trace scheduling compiler

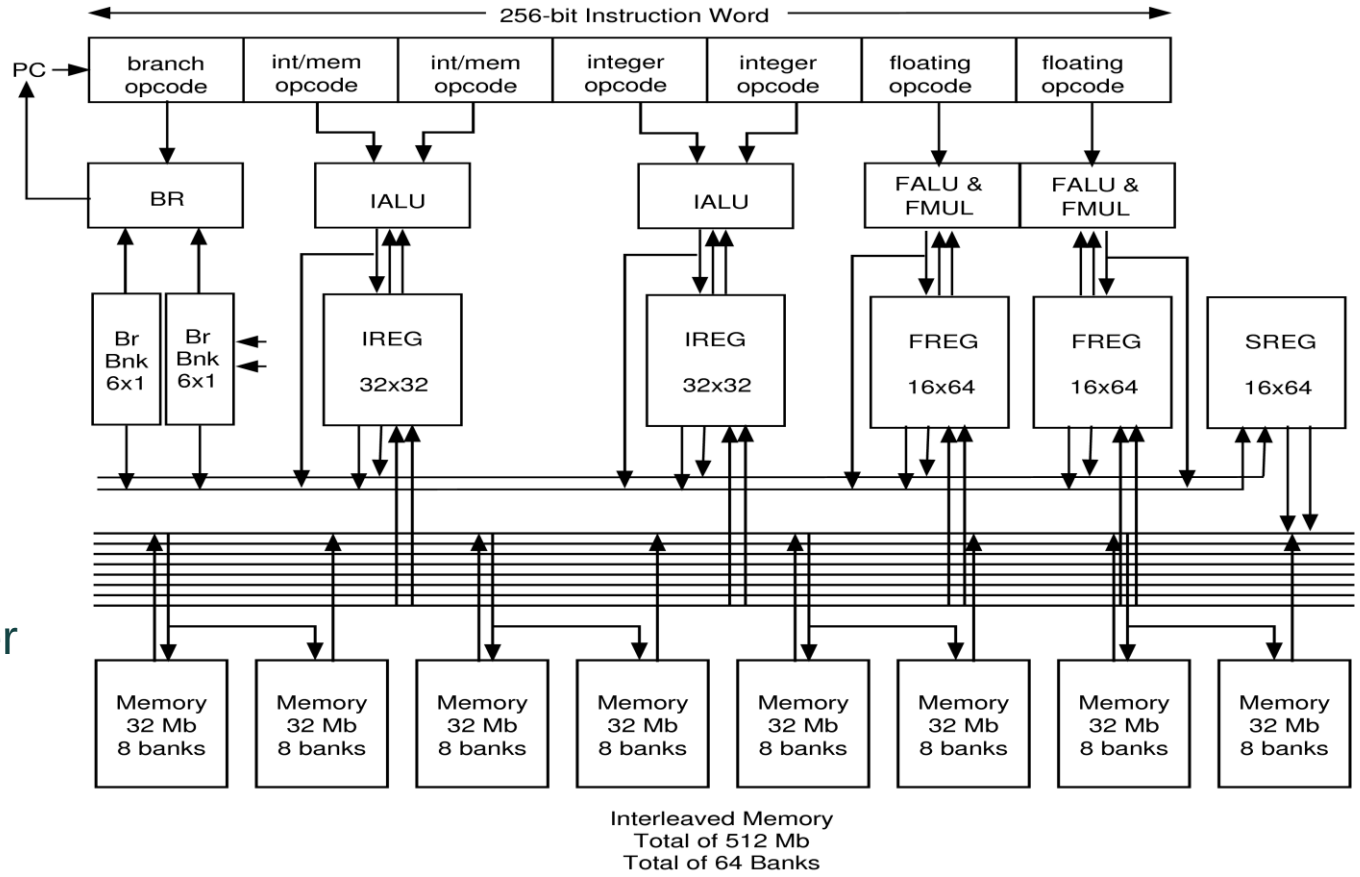
The most suitable VLIW should exhibit four basic features [Colwell et al. 1998 IEEE TC].

- One central controller issues a single long instruction word per cycle.
- Each long instruction simultaneously initiates many small independent operations.
- Each operation requires a small, statically predictable number of cycles to execute.
- Each operation can be pipelined.

In the same spirit as MIPS and the IBM 801, the microarchitecture is exposed to the compiler so that the compiler can make better decisions about resource usage

- the architecture is load/store,
- there is no microcode.

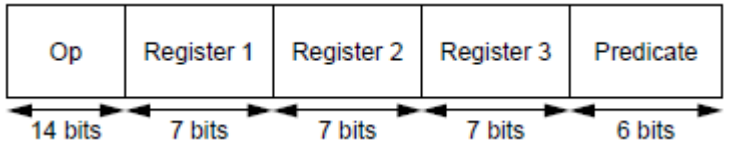
Multiflow TRACE 7/300 is the entry model with one ‘cluster’ of execution units



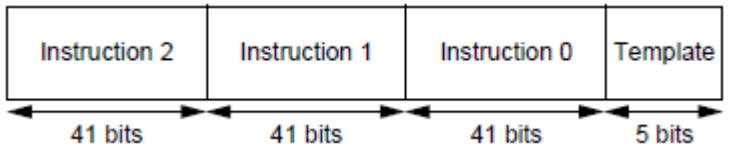
# ITANIUM ARCHITECTURE (IA64)

“Evolution of the VLIW architecture” through the Cydrome Cydra-5 then the HPL Play Doh

Encoding instructions requires 41 bits and instructions are fully predicated




Bundles of 3 instructions are encoded in 128 bits, with template bits that specify the parallel groups

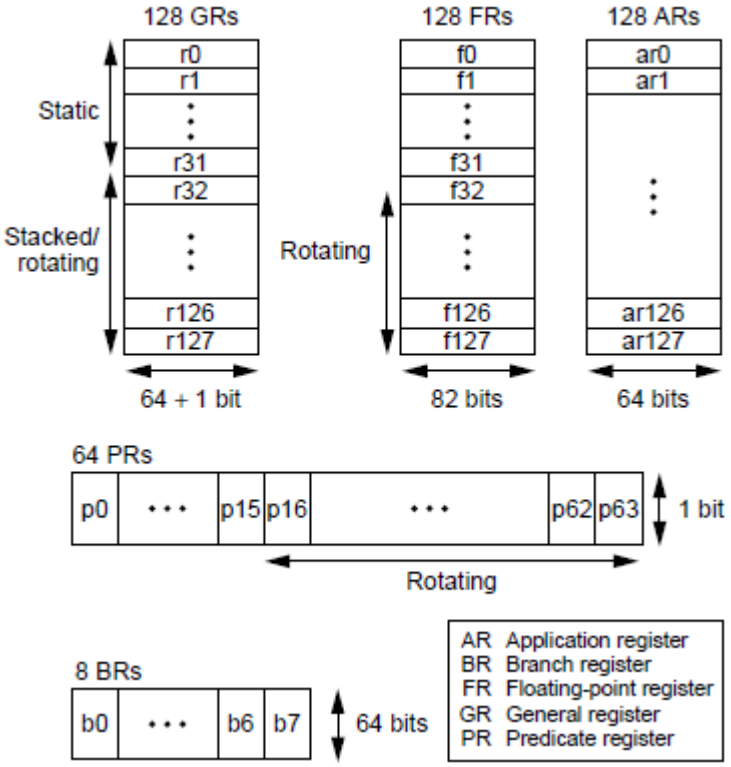



The GR registers have 64+1 bit for “Not a Thing”

Loads can be control-speculative, in case of traps the “Not a Thing” is set and checked later

Loads can be advanced before possibly interfering stores, with interference checked by the “Advanced Load Address Table” (ALAT)

Rotating register are available on GR, FR (floating-point) and PR (predicate)



# VLIW ARCHITECTURE PRINCIPLES

Promote “horizontal microcode” to bundles of RISC-like instructions

Co-design architecture, microarchitecture and compiler optimizations

## Classic VLIW architecture (J. A. Fisher)

- SELECT operation on Boolean value
- Conditional load/store/FPU operations
- Dismissible loads (non-trapping)
- Multi-way conditional branches

## Compiler techniques

- Trace scheduling (global instruction scheduling)
- Partial predication (S. Freudenberger algorithm)

## Main examples

- Multiflow TRACE (1987)
- Philips Trimedia (1998)
- HP Labs Lx / ST200 family (2000)

## EPIC VLIW architecture (B. R. Rau)

- Fully predicated ISA + predicate define instructions
- Speculative loads (control speculation)
- Advanced loads (data speculation)
- Rotating registers

## Compiler techniques

- Modulo scheduling (software pipelining)
- Full predication (R-K algorithm, J. Fang algorithm)

## Main examples

- Cydrome Cydra-5 (1987)
- TI C6X DSPs (1998)
- HP-intel IA64 (2001)

# HP LX ARCHITECTURE / STMICROELECTONICS ST200 FAMILY

VLIW architecture designed by J. Fisher et al. at HP Labs [Faraboschi et al. 2000 ISCA]
[Book “Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools”]

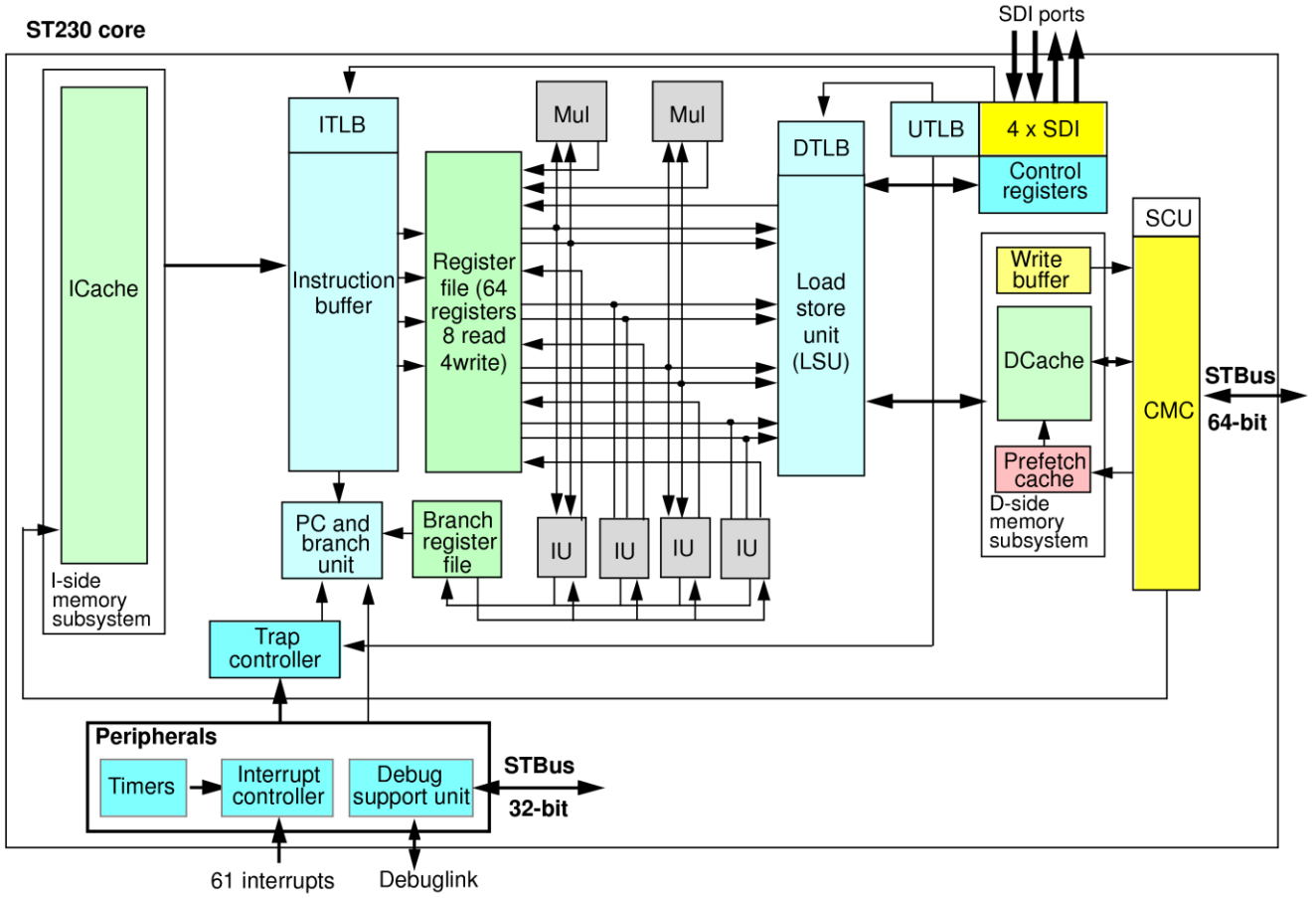
Further developed by STMicroelectronics as the ST200 family of media processors

ISA designed as a 4-issue VLIW extension of the DLX ISA (academic RISC ISA)

- Register file with R0 read as zero and R64 alias of the link register
- Fully pipelined execution units
- No multiway branch
- ‘end-of-bundle’ bit, unlike IA64 templates
- Every parallel instruction group is a valid instruction bundle

HPLabs used the Multiflow trace compiler

The STMicroelectronics st200 compiler, based on Open64 with superblock scheduling and modulo scheduling, outperformed the Multiflow trace compiler



# KALRAY KVX VLIW ARCHITECTURES

Started from the Lx/ST200 VLIW architecture and adopted PowerPC, TIC6x, ARM features

KV3 added SIMD instructions, matrix instructions and a generalization of rotating registers

KV4 is under development with RISC-V convergence requirements

## Lx/ST200 ISA features

- Fully-pipelined execution units, single-cycle reservations
- Any instruction schedule parallel group is a valid bundle

## PowerPC ISA features

- No GP register aliased to special resources (LR, zero)
- Memory addressing modes similar to those of PowerPC
- Effective floating-point support with Fused Multiply Add

## TI C6x ISA features

- Grouping of registers into pairs or quadruples
- Complex number arithmetic (integer and FP32)

## Improved data memory bandwidth

- Load/store widening to 256-bit, no alignment restrictions
- Enable large immediate values in the instruction stream
- All memory accesses may bypass the L1 data cache

## Rework if-conversion support

- Remove Boolean registers and SELECT instructions
- Use CMOV and conditional load/store instructions

## Simple hardware looping

- Counted or forever inner loops with early exits

## Tightly coupled tensor coprocessor (TCA)

# MPPA3 COOLIDGE V2 64-BIT KV3 CORE

VLIW architecture co-designed for compilers to appear as an in-order superscalar core

## Vector-scalar ISA

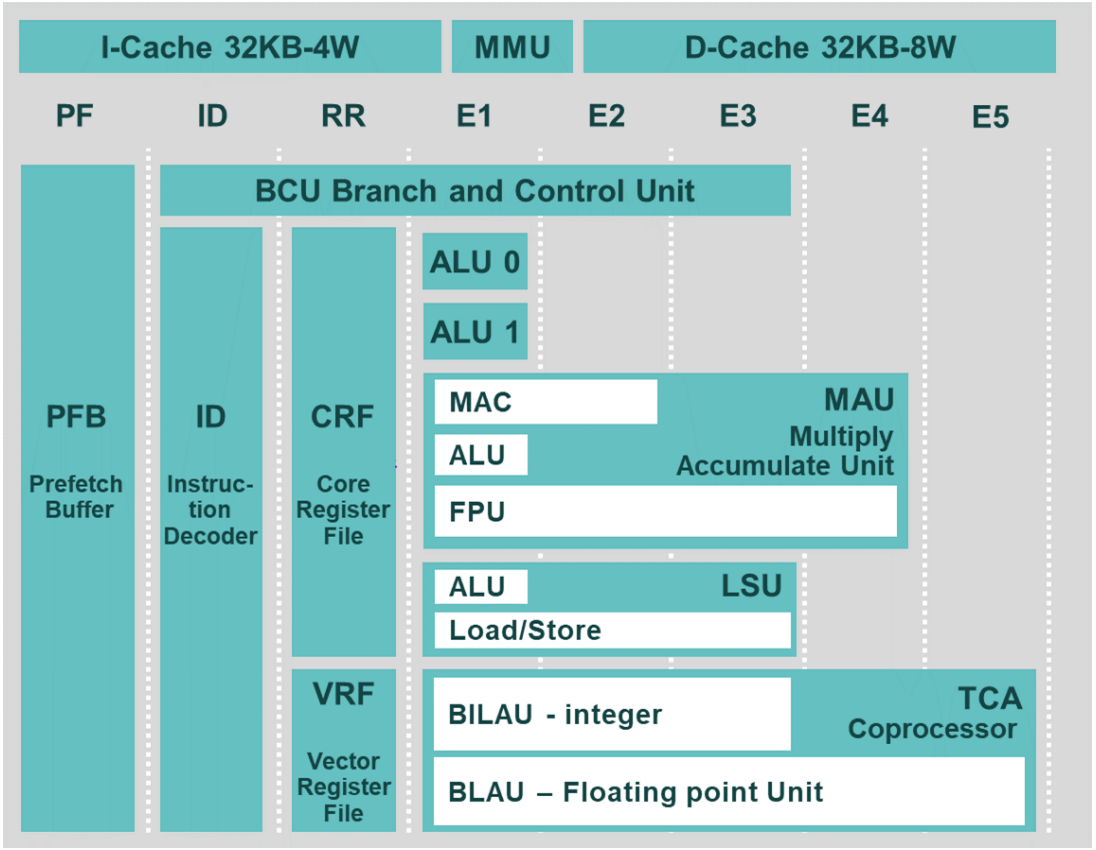
- 64x 64-bit general-purpose registers
- SIMD operands can be single registers (64-bit), register pairs (128-bit) or register quadruples (256-bit)
- 128-bit SIMD instructions by dual-issuing 64-bit on the two ALUS or by using the 128-bit FPU datapath
- FPU capable of 4x FP32 FDP2A operations / cycle
  The FDP2A operator computes $a \pm b \times c \pm d \times e$
- 256-bit load/store unit with byte masking

## DSP capabilities

- Counted or while hardware loops with early exits
- Non-temporal loads (L1 cache bypass / preload)

## CPU capabilities

- 4 privilege levels (rings), MMU (runs Linux kernel)
- Recursive ISA virtualization (Popek & Goldberg)

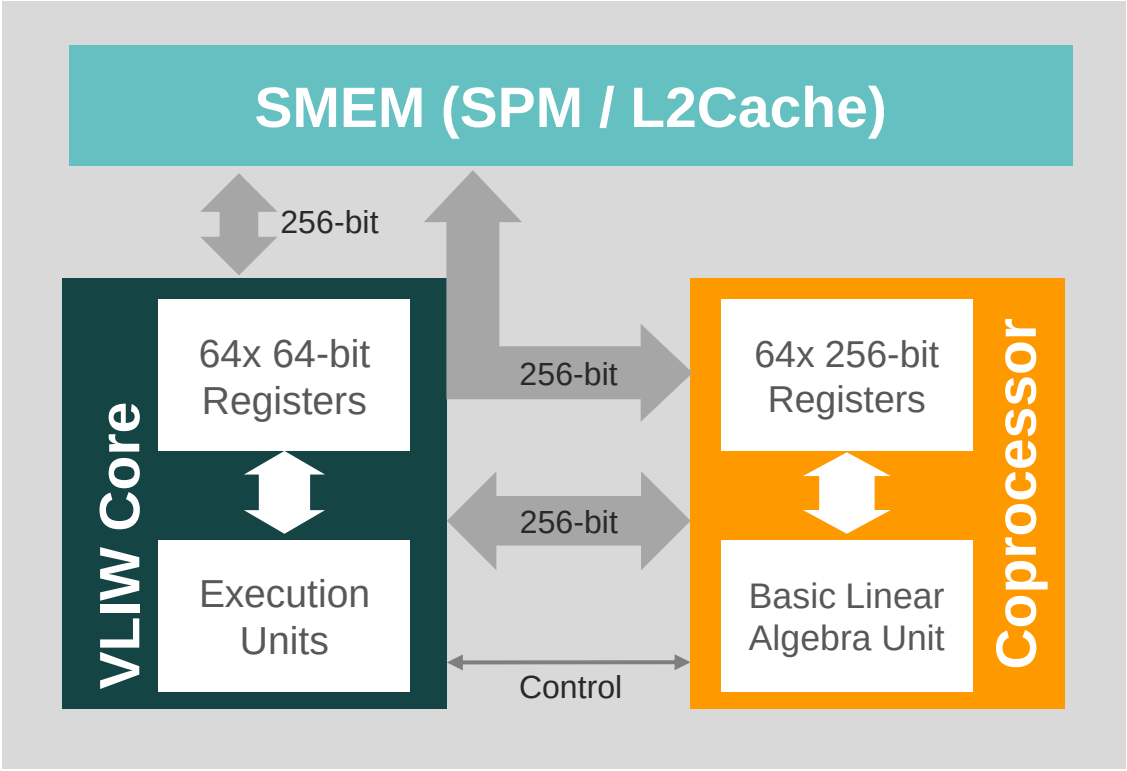



VLIW CORE PIPELINE

# MPPA3 COOLIDGE V2 PROCESSING ELEMENT (PE)

KV3 6-issue 64-bit VLIW core with a tightly-coupled tensor coprocessor

Coprocessor may use 2 issue lanes and shares the load/store unit with core




## VLIW Core

- Scalar 32-bit and 64-bit INT & FP
- 8x 8-bit, 4x 16-bit, 2x 32-bit SIMD
- 128-bit 256-bit SIMD operations by bundling 64-bit SIMD instructions
- 256-bit load/store unit with masking

## Tensor Coprocessor

- Matrix multiply-add on 4x4 tiles
- Matrix zip/unzip & transpose
- 256-bit load/store unit with masking
- Groups of 256-bit registers used as circular buffer or as lookup table

# COPROCESSOR REGISTER GROUPS AS ROTATING REGISTERS

## XPRELOAD <reg-group>, <target-index> = <address>

- <reg-group> specifies a group of coprocessor registers whose size is a power-of-2 an starts on the same power-of-2 register specifier
- <target-index> specifies a core register containing a byte offset into <buffer> where the loaded data will be written modulo the buffer size
- <address> specifies the effective address in memory

## XALIGN <dest-regs> = <reg-group>, <source-index>

- <dest-regs> is a coprocessor register or a core register quadruple
- <reg-group> specifies a coprocessor register group as in XLOAD
- <source-index> specifies a core register containing a byte offset into <buffer> where 32-bytes will be extracted modulo buffer size

## Example of a memory copy loop

- Copy array w[] of 64-bit words (4 elements per 256-bit access)
- Assume src is aligned 8 modulo 32 (could be any byte address)
- Preloading absorbs memory latency and ensures 32-byte aligned 256-bit accesses irrespective of src data start address

```
                                ANDD $r0 = src, 31                  5 low bits of src pointer (here 8)
                                ANDD $r1 = src, -32                 32-aligned src pointer (here src-8)
                                MAKE $r2 =0                         Write index of $a0 into $a0..a3
                                MAKE $r3 = dst                      initialize to dst pointer

                                XPRELOAD $a0..a3, $r2 = 0[r1]       Load w[-1], w[0], w[1], w[2] in $a0
                                ADDD $r2 = $r2, 32                  Write index of $a1 into $a0..a3
                                ADDD $r1 = $r1, 32                  Points to src-8+32 = src+24

                                XPRELOAD $a0..a3, $r2 = 0[r1]       Load w[3], w[4], w[5], w[6] in $a1
                                ADDD $r2 = $r2, 32                  Write index of $a1 into $a0..a3
                                ADDD $r1 = $r1, 32                  Points to src-8+64 = src+56

                                XPRELOAD $a0..a3, $r2 = 0[r1]       Load w[7], w[8], w[9], w[10] in $a2
                                ADDD $r2 = $r2, 32                  Write index of $a3 into $a0..a3
                                ADDD $r1 = $r1, 32                  Points to src-8+96 = src+88

LOOP BEGIN <i>

                                                                    Load w[11+i], w[12+i], w[13+i],
                                XPRELOAD $a0..a3, $r2 = 0[r1]       w[14+i] in $a<3+i>
                                ADDD $r2 = $r2, 32                  Update write index into $a0..a3
                                                                    Points to src-8+128+32*i =
                                ADDD $r1 = $r1, 32                  src+120+32*i

                                XALIGN $r8r9r10r11 = $a0..a3,       $r8=w[0+i], $r9=w[1+i],
                                $r0                                 $r10=w[2+i], $r11=w[3+i]
                                ADDD $r0 = $r0, 32                  Update read index into $a0..a3

                                STORE 0[$r3] = $r8r9r10r11          Store w[0+i, .. 3+i] at dst+32*i
                                ADDD $r3 = $r3, 32                  Update store address


LOOP END
```

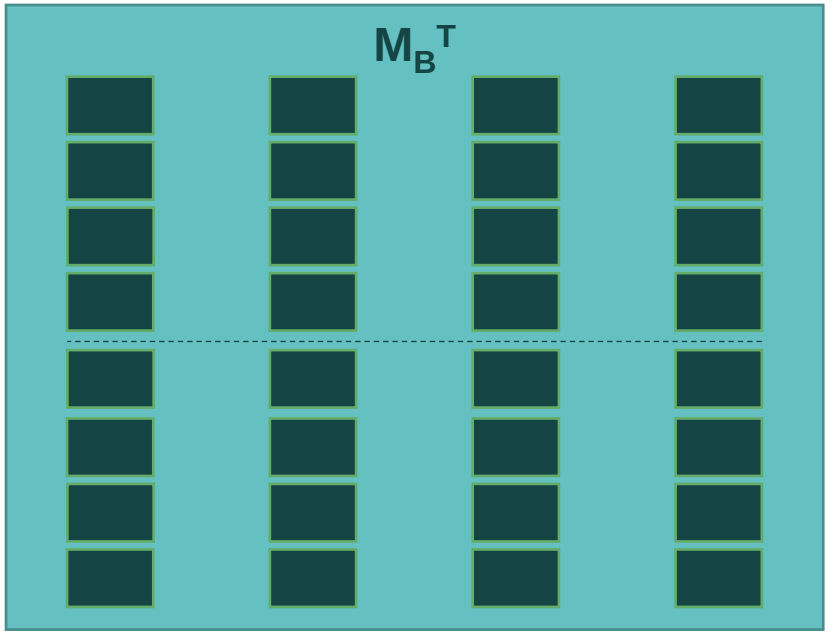
# KV3 TENSOR COPROCESSOR MATRIX MULTIPLY-ADD

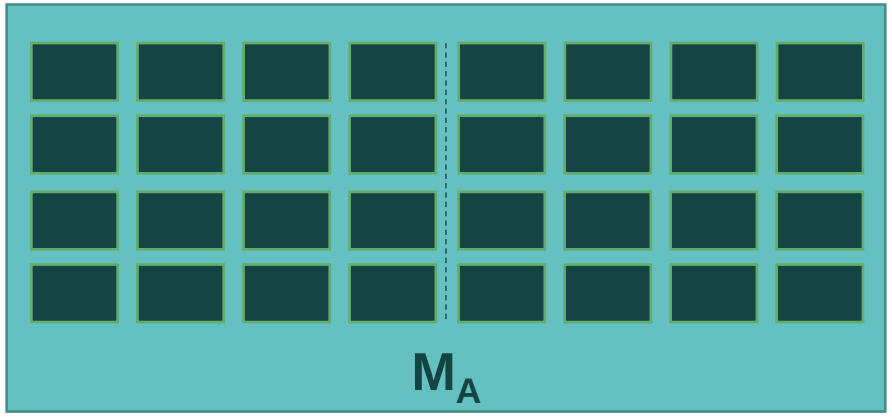
16-bit to 32-bit floating-point operation: (4x8)fp16 . (8x4)fp16 += (4x4)fp32

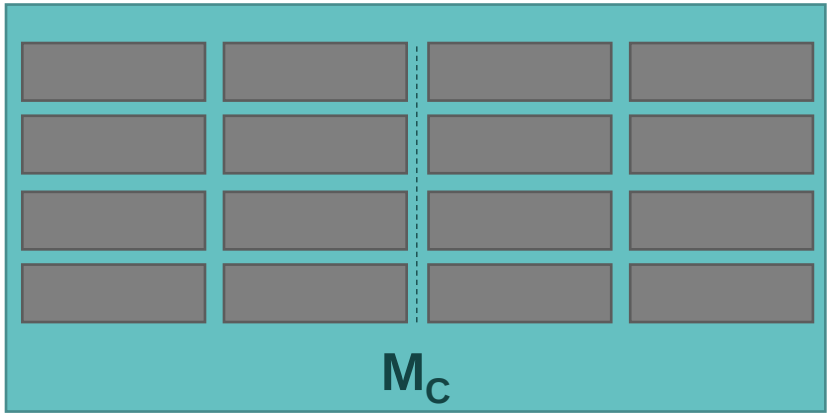
Based on exact fused Dot Product Add operator

- 512b x 512b += 512b operands
- 256-bit register pair multiplicands
- 256-bit register pair addend & accumulator
- 128 FMA equivalent per cycle, 256 flops/cycle

$$M_A x M_B^T += M_C$$

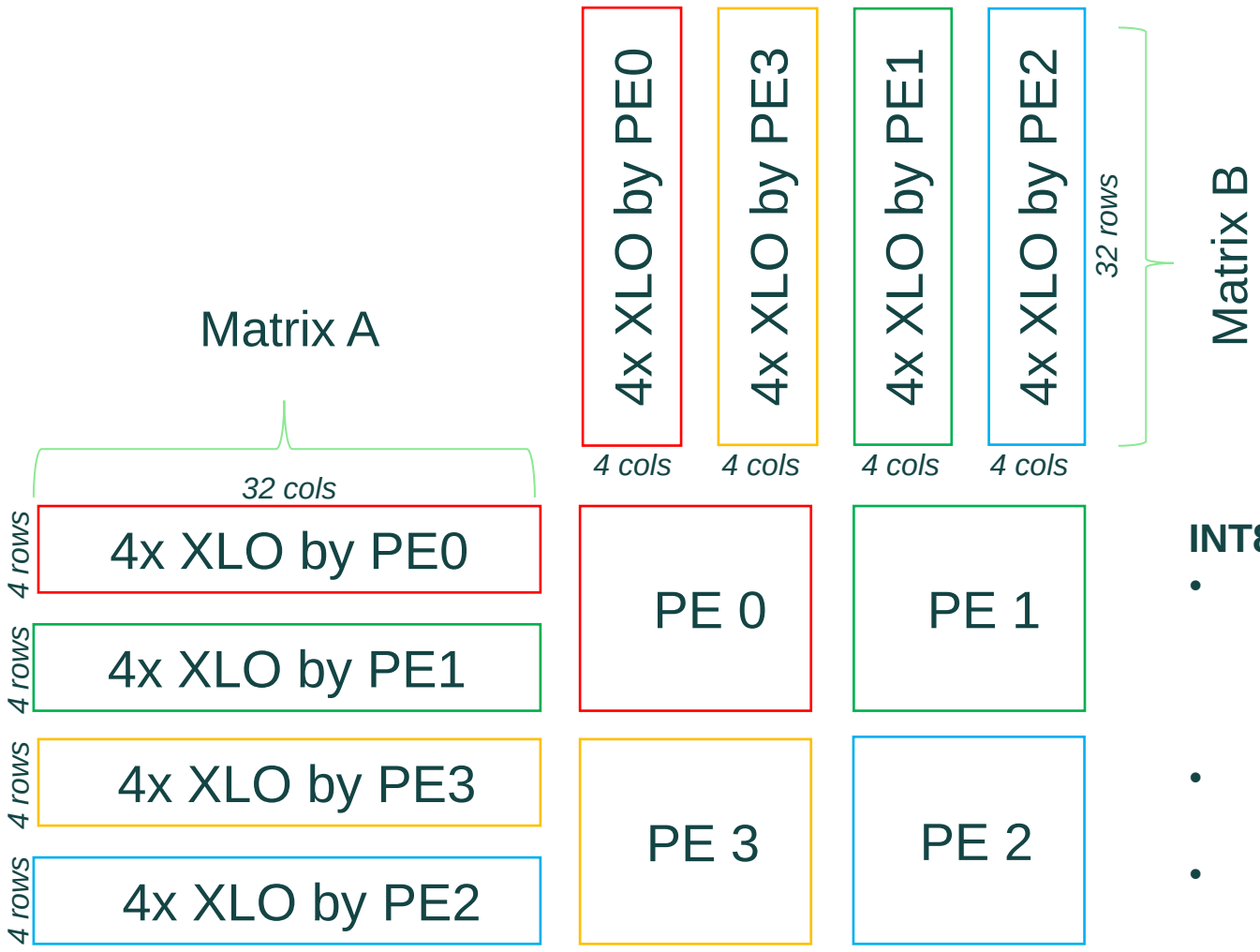


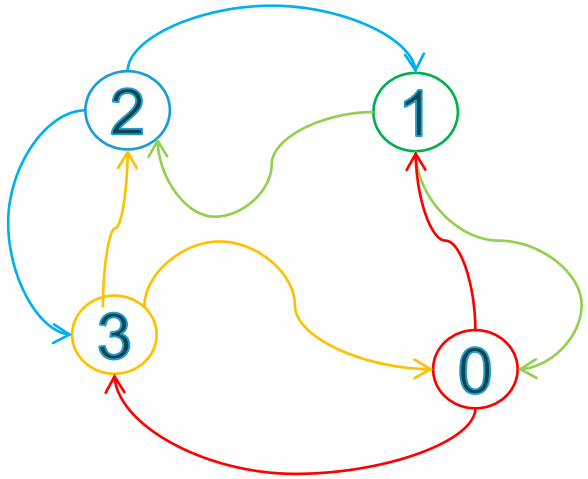



# KV3 PE TO PE 256-BIT DUPLEX COMMUNICATION RING

Distribute tensor operations across 4 PEs while avoiding redundant memory loads




**INT8.32 operation (4x16) . (16x4) += 4x4**

- Macro-scheme executed by 4 PEs
  - 8x 256-bit memory loads (XLO) per PE
  - 8x 256-bit data exchanges per PE
  - 8x matrix multiply-add per PE
- Matrix A and B are loaded by quarter by each PE which exchange one quarter with 2 different PEs
- Kernel for INT8.32: **(16 x 32) . (32 x 16) += 16 x 16**

# AGENDA

1. Kalray MPPA Platform
2. RISC Architectures
3. VLIW Architectures
4. RISC-V Accelerators
5. Converging to RISC-V
6. Conclusions

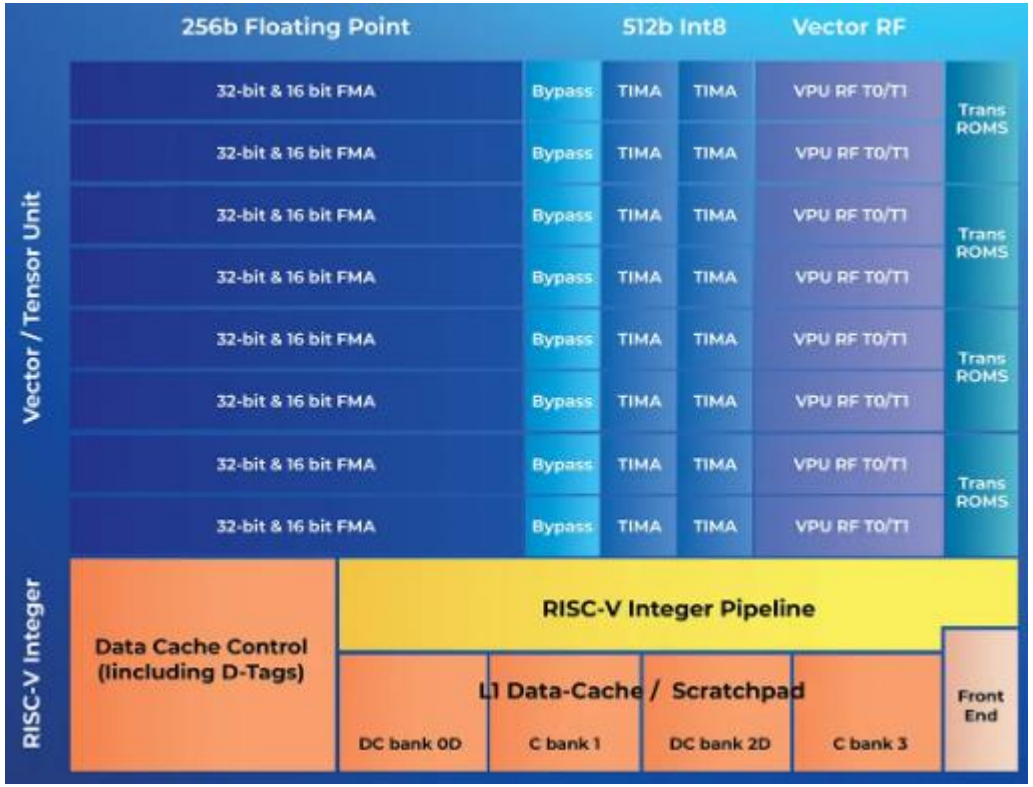
# RISC-V BASED AI ACCELERATION EXAMPLE
## ESPERANTO ET-SOC-1

“Energy efficiency across a range of AI, HPC and mixed mode workloads”

The ET-SoC-1 chip features over one thousand RISC-V processors on a single 7nm chip.

- 1088 energy-efficient ET-Minion 64-bit RISC-V in-order cores, each with a custom vector/tensor unit optimized for ML applications
- 4 high-performance ET-Maxion 64-bit RISC-V out of-order cores for running an OS in self-hosted mode
- Over 160 million bytes of on-chip SRAM

ET-Minion executes instructions in order, for maximum efficiency, while extensions support vector and tensor operations on up to 256 bits of floating-point data (using 16-bit or 32-bit operands) or 512 bits of integer data (using 8-bit operands) per clock cycle.



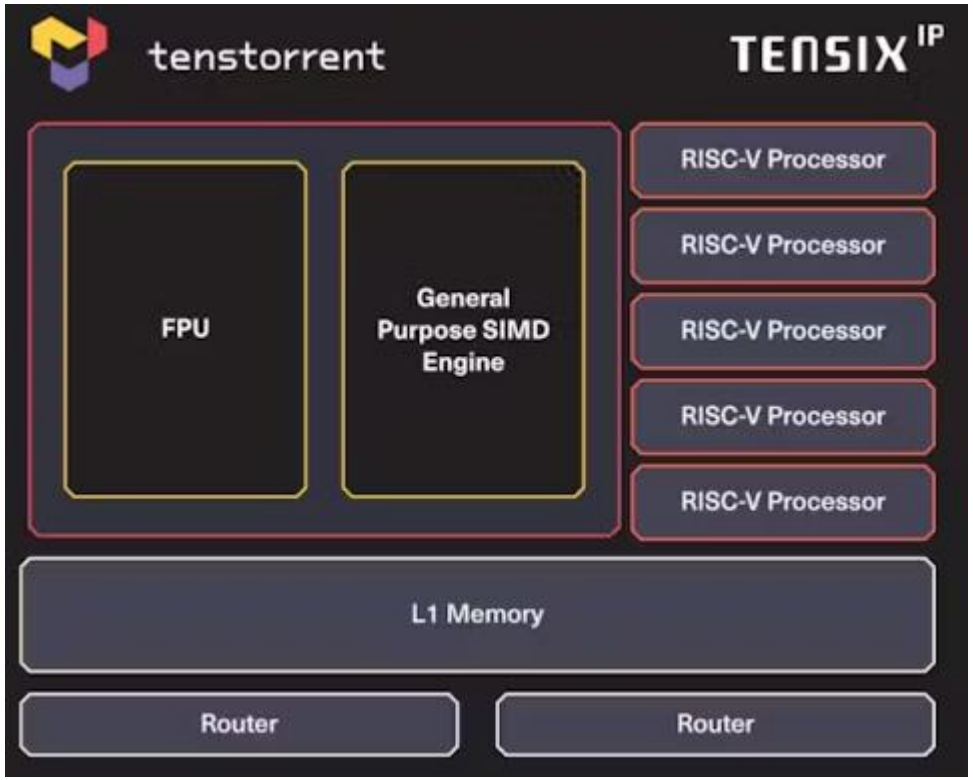
# RISC-V BASED AI ACCELERATION EXAMPLE
# TENSTORRENT WORMHOLE

"Featuring RISC-V & Phenomenal Price To Performance Value"

Tenstorrent's Wormhole n150

- The Wormhole n150 features a single processor with 72 Tensix cores. Each Tensix core features 5 RISV-C baby cores
- Tensix cores support a variety of data formats, including BF4, BF8, INT8, FP16, BF16, and even FP64
- The new Wormhole n150 chip and its single processor features a 160W TDP and pushes 262 TFLOPs of FP8 performance

Each Tensix core comprises of five RISC cores, an array math unit for tensor operations, a SIMD unit for vector operations, 1MB or 2MB of SRAM, and fixed function hardware for accelerating network packet operations and compression/decompression.



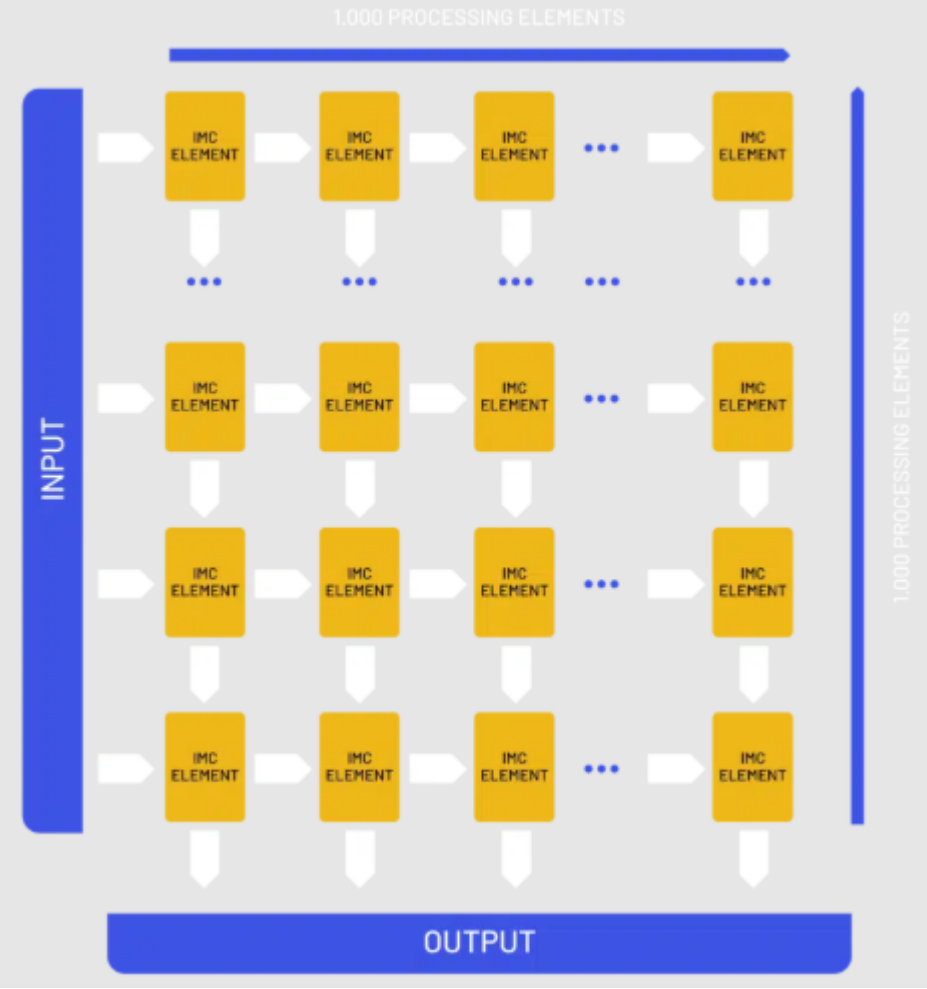
# RISC-V BASED AI ACCELERATION EXAMPLE
## AXELERA METIS

"Powerful Edge Inference requires groundbreaking and cost-effective AI acceleration"

At the heart of each AI core is a massive in-memory-computing-based matrix-vector-multiplier to accelerate matrix operations, up to 53.5 TOPS at energy efficiency of 15 TOPS/W.

At the heart of Metis AI Processing Unit, there are four AI Cores.

- Each AI core provides a 512×512 matrix-vector multiplication (MVM) in-memory compute array and a vector datapath that operates on streams of data.
- Push as much as possible into low-level driver software where we can innovate, correct, and adapt throughout the product's life cycle.
- Things like atomic handling may only be standardized for specific platforms
- Therefore, each AI Core has a dedicated RISC-V, application-class core, that has full control over the datapath unit.



# RISC-V BASED AI ACCELERATION EXAMPLE
# MIPS ACCELERATOR INTEGRATION

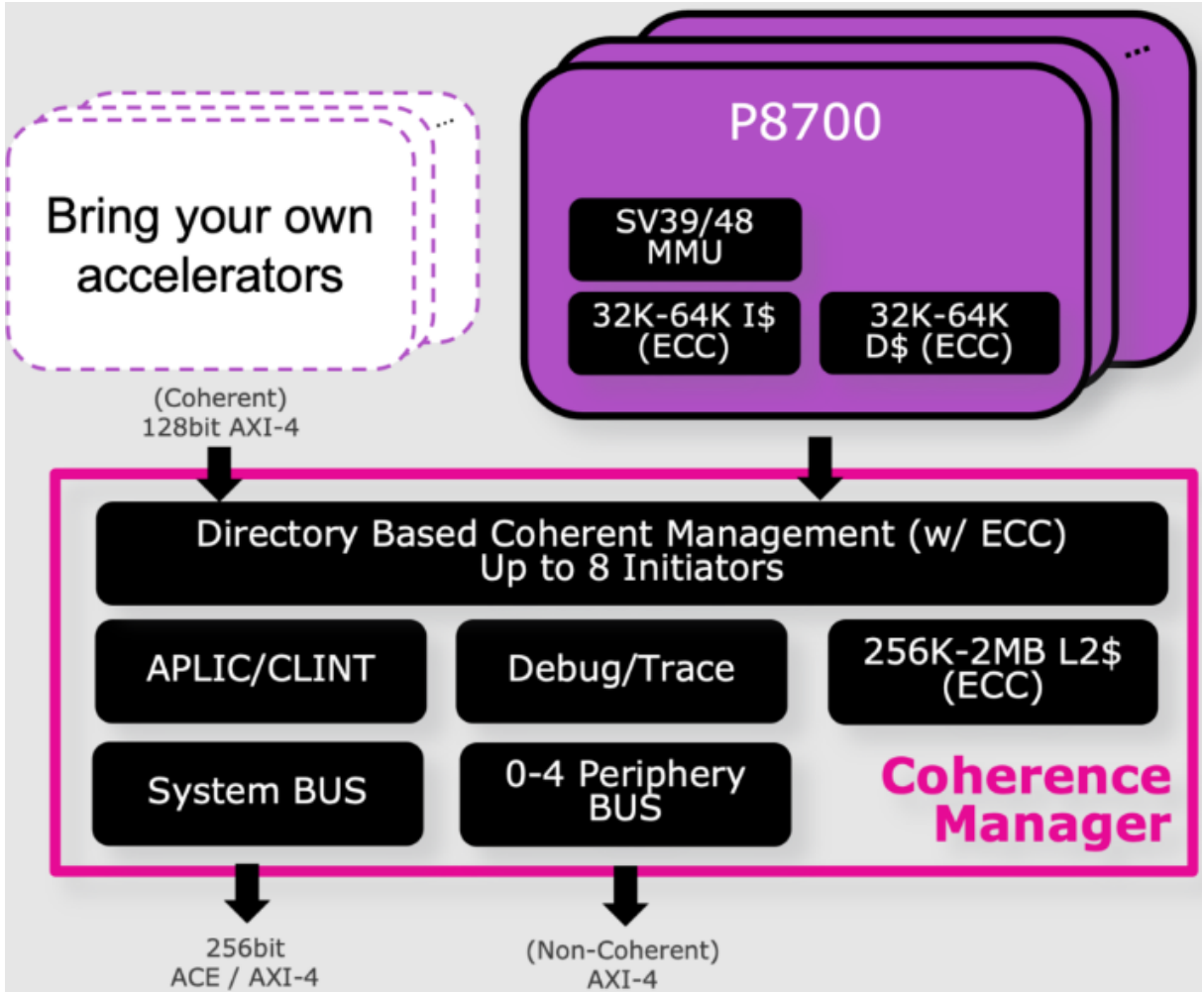
"Bring your own accelerators"

MIPS RISC-V Processors

- High-performance compute with simultaneous multi-threading (SMT)
- 4-issue, 16-stage out-of-order pipeline with 1- or 2-way SMT
- Custom instructions for improved memory operations and data movement

Coherence Manager

- Support for up to 8 Coherent initiators comprising of either MIPS RISC-V Processors or 3rd party accelerators
- Cluster Level-2 Cache L2$ up to 2MB
- HW pre-fetch, widened busses, reduced latency



# RISC-V BASED AI ACCELERATION EXAMPLE
## SIFIVE ACCELERATOR MORAY

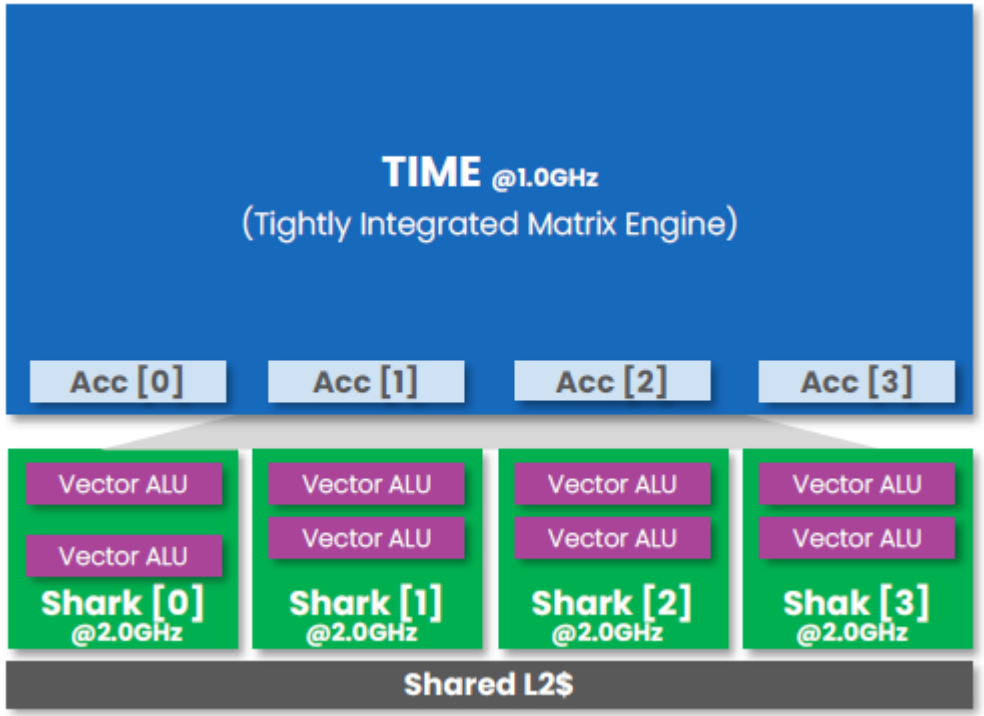
"Tightly Integrated Matrix Engine shared between four Shark processors with dual vector ALUs"

Vector Processing for AI/ML workloads

- SiFive Intelligence Extensions (custom instructions that accelerate AI/ML performance critical operations)
- VCIX interface for direct connectivity of vector accelerators
- Separate vector load/store units (full-duplex operation)
- 1024-bit VLEN (X390), 512-bit VLEN (X380)

Scalar processing architecture

- 64-bit RISC-V ISA, 8-stage dual-issue in-order
- Linux capable Applications processor




Matrix instructions accelerates GEMM routines

- Instructions fetched by CPU
- Source data comes from vector registers

Accumulator can be accessed by vector unit

- Accumulator context for each Shark core

# RISC-V BASED AI ACCELERATION EXAMPLE
# SEMIDYNAMICS ALL-IN-ONE

"Fuse CPU, GPU and NPU into a unified solution"

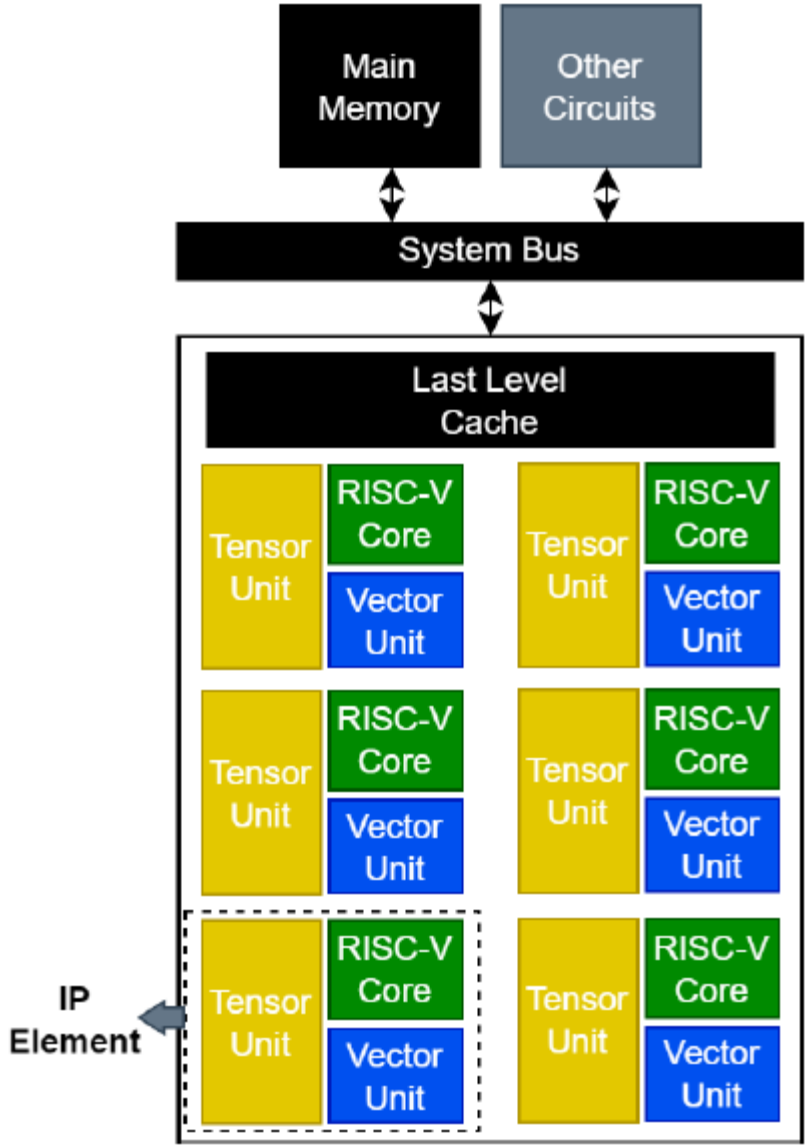
Out-of-order 64-bit core based on RISC-V ISA that includes *Gazzillion Misses*™ Technology to efficiently manage large data sets

The VU is fully compliant with the RISC-V Vector Extension 1.0, can deliver up to 2048 bits of computation per cycle

The TU implements Semidynamics' custom extension for tensor instructions achieves up to 8 TOPS (INT8) per GHz

The three components share the L1 data cache. Furthermore, the VU and TU have access to the same Vector Register

DMA-free programming: computations can be offloaded to VU or TU with zero latency. Memory copies are not required since the core, VU and TU can access the same shared data cache



# AGENDA

1. Kalray MPPA Platform
2. RISC Architectures
3. VLIW Architectures
4. RISC-V Accelerators
5. Converging to RISC-V
6. Conclusions

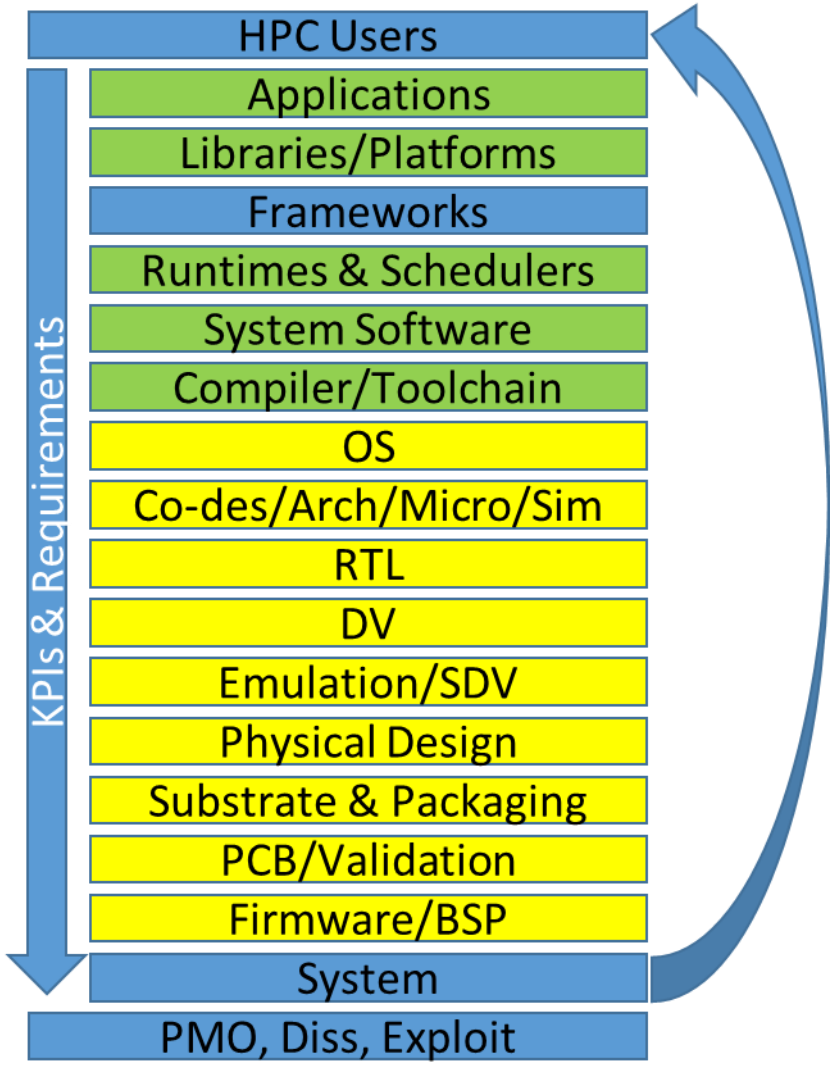
# FRAMEWORK PARTNERSHIP AGREEMENT (FPA) FOR DEVELOPING A LARGE-SCALE EUROPEAN INITIATIVE FOR HIGH PERFORMANCE COMPUTING (HPC) ECOSYSTEM

## Expected Outcome:

Framework Programme Agreement (FPA) for European hardware and software technologies, based on RISC-V in order to deliver high-end processors and/or accelerators and systems based on a strategic research roadmap, and the realisation of test-beds, pilots and/or demonstrators, integrating these processors.

## The FPA is expected to address the following outcomes:

Contribution towards European technological sovereignty, by establishing, maintaining and implementing a strategic R&I roadmap that fosters the European capabilities to design, develop and produce the IP related to high-end processors and/or accelerators based on RISC-V, driven by relevant key performance indicators.



# RISC-V CORE IMPLEMENTATION DIRECTIONS

Application cores: for multicore processors maximizing single-thread performance

Acceleration cores: for manycore processors maximizing multiple thread throughput

## RISC-V FOR APPLICATION CORES STANDARD LIBRARIES AND TOOLS

- Runs rich operating system (Linux distribution)
- Implements standard extensions, not V
- SMP programming model
- Out-of-Order superscalar micro-architecture
  - Branch prediction
  - Multiple instruction issue
  - Hardware register renaming
  - Hardware data prefetching
  - Hardware speculative execution

## RISC-V FOR ACCELERATION CORES CUSTOMIZED LIBRARIES AND TOOLS

- Runs simple run-time system or RTOS
- Compute unit (cluster) with local memory and data move engine (DMA with atomics)
- Power-efficient micro-architecture
  - In-order issue or VLIW core
  - High-throughput local memory access
  - Software data prefetching or preloading
- Possibly implements V extension as prerequisite for matrix operations (custom extension or IME & AME draft standards)

# TWO OPTIONS FOR A RISC-V KVX EXTENSION

Both options require that KV4 instructions align on RISC-V (floating-point, atomics, memory)

Both options enable to execute multiple RISC-V instruction per cycle (in-order superscalar)

Ensure RISC-V compliance by running the standard RISC-V architecture test suites

1. KVX with RISC-V execution mode (PS bit)
   - RISC-V and KVX instruction encoding may conflict
   - Execute the RV64G user instructions without privilege
   - Interrupts, Traps and SysCalls switch mode to KVX
   - Also run low-level and runtime software in KVX mode

2. KVX as a (non-standard) RISC-V Extension
   - Reuse the ‘C’ extension to encode the KVX ISA inside the RISC-V ISA opcode space
   - Freely mix the two ISAs in a single application, enable to unify the two software toolchains
   - Requires to reorganize the KVX encodings, which disconnects the KV4 tools from the KV3 tools

# OPTION1: KV4 CORE EXECUTING RISC-V BINARY CODE

Compiled application code produced with a standard RISC-V toolchain (GCC, NewLib)

Boot and initialize the KVX execution platform in KVX mode

Set the RISC-V start address in $SPC (saved PC), and Privilege Level in $SPS (saved PS)

Execute RFE so that execution starts in RISC-V mode at $SPC address

```
2432025:       0x00000000000218f8:    wfxl $sps(0x0000000000f10f71), $r1(0x0000000100000000)
2432026:       0x00000000000218fc:    set $spc(0x0000000001200120) = $r2(0x0000000001200120)
2432027:       0x0000000000021900:    rswap $r0(0x0000000001052000) = $sr(0x0000004000000000)
2432028:       0x0000000000021904:    rswap $r0(0x0000004000000000) = $sr(0x0000000001052000)
2432028:       0x0000000000021904:    sq 0[$r0(0x0000000001052000)] = $r12r13(0x0000000000106b000_0000000001053e80) [
2432029:       0x000000000002190c:    make $r0(0x0000000000000000) = 0
2432030:       0x0000000000021910:    barrier
2432031:       0x0000000000021914:    d1inval
2432032:       0x0000000000021918:    i1inval
2432033:       0x000000000002191c:    barrier
# RFE PL transition: PL2 => PL3
2432034:       0x0000000000021920:    rfe [Branch Taken]
2432035:       0x0000000001200120:    auipc x3(0x0000000001207120), 7
2432036:       0x0000000001200124:    addi x3(0x0000000001207810), x3(0x0000000001207120), 1776
2432037:       0x0000000001200128:    addi x10(0x0000000000000000), x0(0x0000000000000000), 0
2432038:       0x000000000120012c:    beq x10(0x0000000000000000), x0(0x0000000000000000), 8 [Branch Taken]
2432039:       0x0000000001200134:    auipc x10(0x0000000001208134), 8
2432040:       0x0000000001200138:    addi x10(0x00000000012079d0), x10(0x0000000001208134), -1892
2432041:       0x000000000120013c:    auipc x12(0x000000000120813c), 8
2432042:       0x0000000001200140:    addi x12(0x0000000001207fd0), x12(0x000000000120813c), -364
2432043:       0x0000000001200144:    sub x12(0x0000000000000600), x12(0x0000000001207fd0), x10(0x00000000012079d0)
2432044:       0x0000000001200148:    addi x11(0x0000000000000000), x0(0x0000000000000000), 0
2432045:       0x000000000120014c:    jal x1(0x0000000001200150), 4580
```

On RISC-V system call (ECALL), transition to higher Privilege Level which is back to KVX

# OPTION2: REUSE THE ‘C’ ENCODING SPACE FOR VLIW BUNDLES

Supporting VLIW encodings [The RISC-V Instruction Set Manual Volume I]

The base 32-bit encoding has to be supported to allow use of any standard software tools

Fixed-Size Instruction Group

Encoded-Length Groups

Fixed-Size Instruction Bundles

End-of-Group bits in Prefix

- Repurpose the two prefix bits in the fixed-width 32-bit encoding.
- One prefix bit can be used to signal "end-of-group" if set …
- The main disadvantage of this approach is that the base ISAs lack the complex predication support usually required in an aggressive VLIW system, and it is difficult to add space to specify more predicate registers in the standard 30-bit encoding space.

The proposed RISC-V KVX extension adapts the “End-of-Group bits in Prefix” direction:

- “The main disadvantage” is from the implicit assumption that a VLIW is EPIC-Style and that bundles contain RISC-V instruction opcodes
- The RV64G ISA is register-constrained so the VLIW instructions have to be recoded anyway

[The C extension also recodes basic instructions]

| | | xxxxxxxxxxxxxxaa | 16-bit (aa≠11) |
|---|---|---|---|
| | xxxxxxxxxxxxxxxx | xxxxxxxxxxbbb11 | 32-bit (bbb≠111) |
| · · · xxxx | xxxxxxxxxxxxxxxx | xxxxxxxxxx011111 | 48-bit |
| · · · xxxx | xxxxxxxxxxxxxxxx | xxxxxxxxx0111111 | 64-bit |
| · · · xxxx | xxxxxxxxxxxxxxxx | xnnnxxxxx1111111 | (80+16*nnn)-bit, nnn≠111 |
| · · · xxxx | xxxxxxxxxxxxxxxx | x111xxxxx1111111 | Reserved for ≥192-bits |
| base+4 | base+2 | base | |

# SUPPORTING THE RISC-V VECTOR EXTENSION

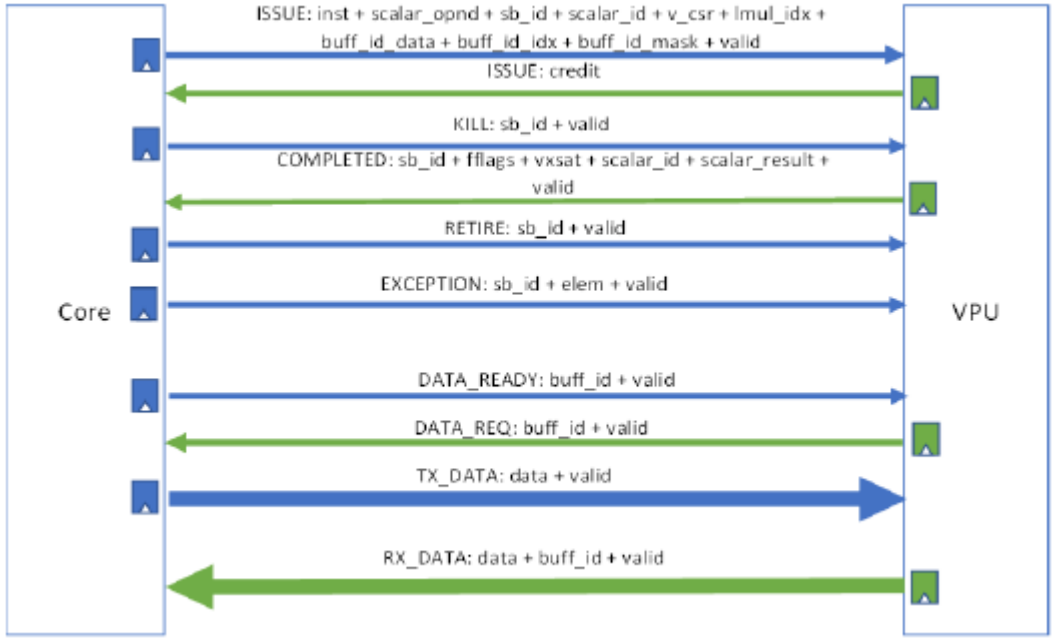
The RISC-V Vector extension is designed for classic HPC applications (FP64)

Not yet proven for Edge / DSP applications, unlike Intel AVX512 and ARM NEON

The KVX SIMD ISA is designed for efficient mapping of Intel AVX512 and ARM NEON

## OPTION 1: CONNECT A VECTOR PROCESSING UNIT (VPU) TO THE KVX USING A WELL DEFINED INTERFACE

BSC VPU Open Vector Interface (EPI VEC)




## OPTION 2: EXECUTE VECTOR INSTRUCTIONS BY RUNNING “HORIZONTAL MICROCODE” KERNELS

Each RISC-V vector instruction is dispatched to a KVX kernel that implements its functionality

Each KVX kernel could match or outperform a RISC-V Vector instruction given the same local memory system (bandwidth, latency)

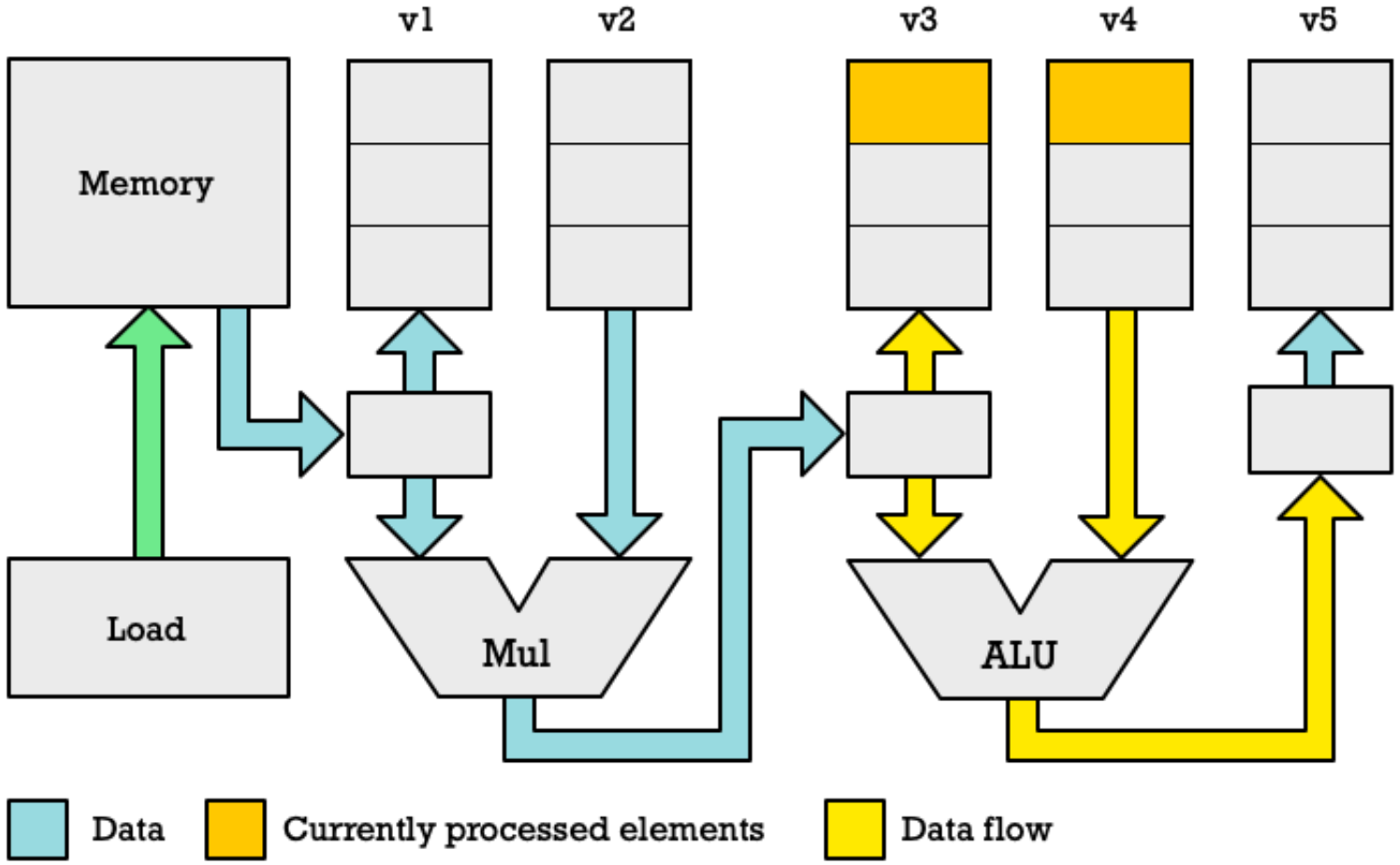
Overhead of branching to a software kernel is not significant for long vectors (BSC VPU target) and could be lowered with hardware support

However, sequential runs of optimized hardware / software KVX kernels misses the “vector chaining” opportunities

# VECTOR INSTRUCTION CHAINING

Overlapped execution in a sequence of dependent vector instructions [Cray-1, Cray XMP]

Vector instruction starts as soon as the first elements of its source vectors are available



# DIRECTIONS FOR OPTION 2

Addressing the challenges of executing one vector instruction / kernel at a time

```
void add_ref(int N, double *c, double *a, double *b) {
  for (int i = 0; i < N; i++)
    c[i] = a[i] + b[i];
}
```

Output from the BSC LLVM-based compiler in vector-length agnostic (VLA) mode [Ferrer 2022]

```
.LBB0_4: # %vector.body
  slli a7, a4, 3
  add a6, a2, a7
  sub a5, a0, a4
  vsetvli t0, a5, e64, m1, ta, mu
  vle64.v v8, (a6)
  add a5, a3, a7
  vle64.v v9, (a5)
  vfadd.vv v8, v8, v9
  add a5, a1, a7
  add a4, a4, t0
  vse64.v v8, (a5)
  bne a4, a0, .LBB0_4
```

Proposed solution: patching of binary RVV code at load time or at MMU mapping time

- Identify the sequences of same **vtype** and **vl** inside basic blocks (split at **vsetvl*** instruction)
- Select the subsequences of vector instructions that expose chaining opportunities
- Generate or instantiate from a code template the corresponding KVX kernel
- Patch all but one RVV instruction with NOP and call the kernel in the last RVV instruction

See “Software Vector Chaining” [Ertl 2018]

# AGENDA

1. Kalray MPPA Platform
2. RISC Architectures
3. VLIW Architectures
4. RISC-V Accelerators
5. Converging to RISC-V
6. Conclusions

# RISC AND VLIW ARCHITECURES

Common origin of RISC and VLIW architectures

- RISC architecture was motivated by exposing “vertical microcode” as simple register-register instructions
- VLIW architecture was motivated by exposing “horizontal microcode” as bundles of RISC-like instructions

In both cases, microcode was eliminated in favor of pipelined instructions, and architecture was tuned to match compiler machine code optimizations

RISC and VLIW architectures have been extended to support SIMD and vector instructions

- SIMD instructions have architecturally defined vector sizes and execute single-cycle
- Vector instructions abstract vector size for binary compatibility but may not execute single-cycle

Compiler compatibility trumps binary compatibility, especially for VLIW domain-specific architectures (DSA)
“Ten lessons from three generations shaped Google's TPUv4i” [Jouppi et al. 2021 ISCA]

RISC implementations have evolved through superpipeline, superscalar in-order, superscalar out-of-order

VLIW implementations cannot execute out-of-order, so must explicitly manage register renaming at compile time and mitigate the effects of variable load latencies

VLIW architectures still dominate in image processing, signal processing and AI pre/post processing

# SUCCESSFUL VLIW ACCELERATORS

Main applications in image processing, signal/telecom, AI

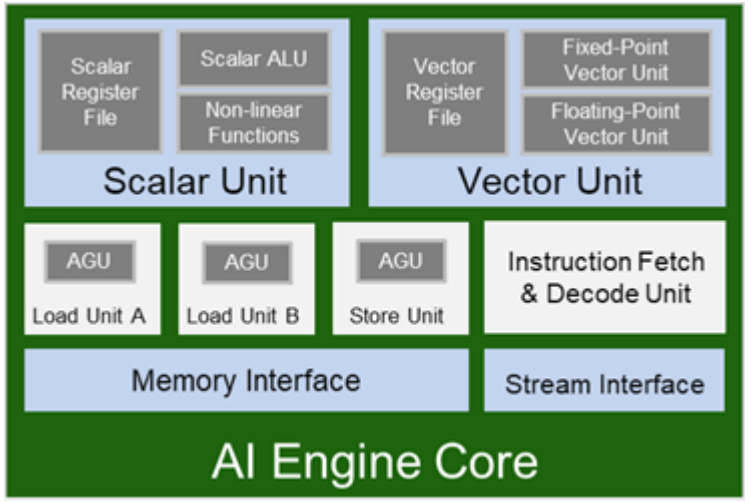
Synopsys, CEVA, Intel/Habana, Xilinx, AMD

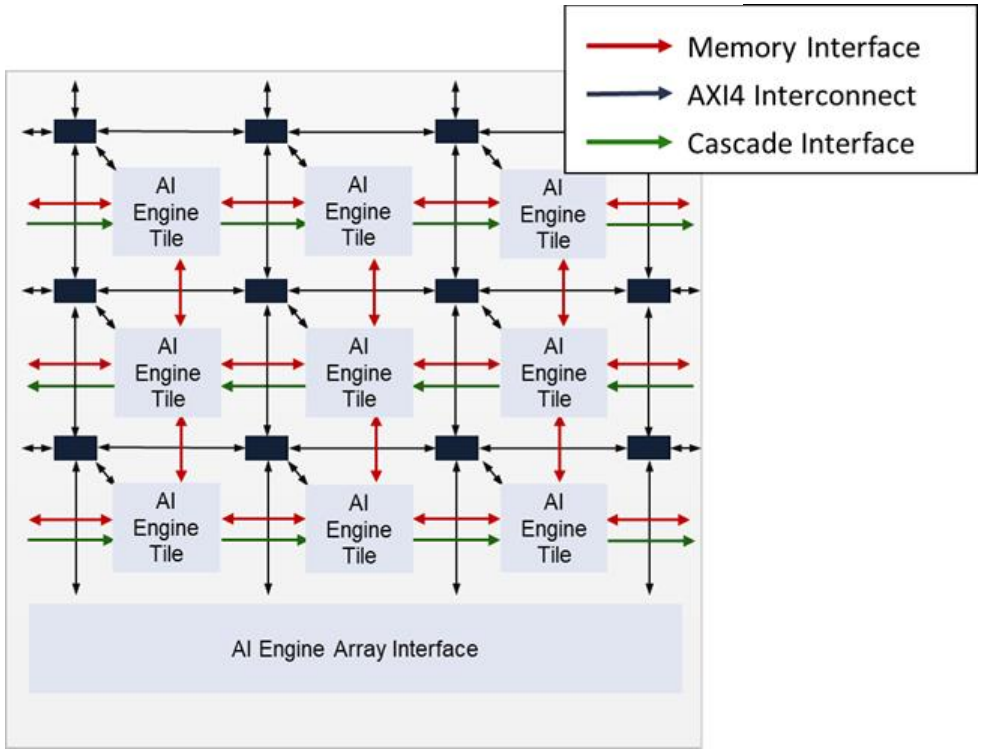
## INTEL/HABANA GAUDI AI ACCELERATORS

- Gaudi2 integrates Habana's fourth generation Tensor Processor Core.
- The TPC is a general purpose VLIW processor which is 256B SIMD wide and supports FP32, BF16, FP16 & FP8, in addition to INT32, INT16 & INT8 data types.
- the TPC exposes a DMA-free programming model which significantly eases SW development.

## XILINX VERSAL / AMD XDNA AI ENGINES

- Each AI Engine tile consists of a very long instruction word (VLIW), single instruction multiple data (SIMD) vector processor optimized for machine learning and advanced signal processing applications.
- AMD XDNA is a spatial dataflow NPU architecture consisting of a tiled array of AI Engine processors.



# THANK YOU

KALRAY

THE POWER OF MORE

www.kalrayinc.com