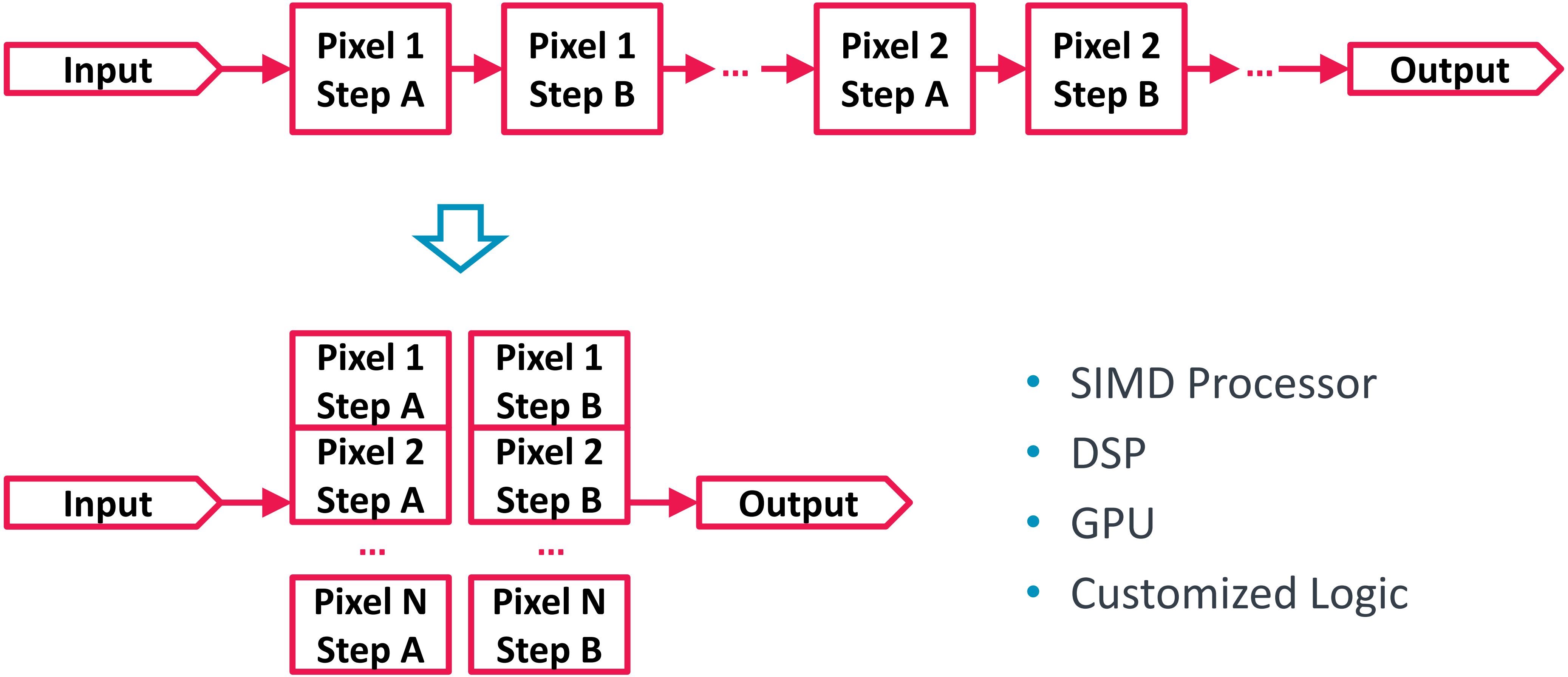
Accelerate Image Processing Using SIMD Engine

Learning Outcomes

At the end of this module, you will be able to:

- Explain the purpose of SIMD and give examples of SIMD implementation.
- Explain what Arm Neon technology does and how to use it.
- Outline the usage of Neon technology in C language.
- Give an example on using Neon in C language.
- Compare and contrast the benefits and limitations of using Neon for accelerating image processing.

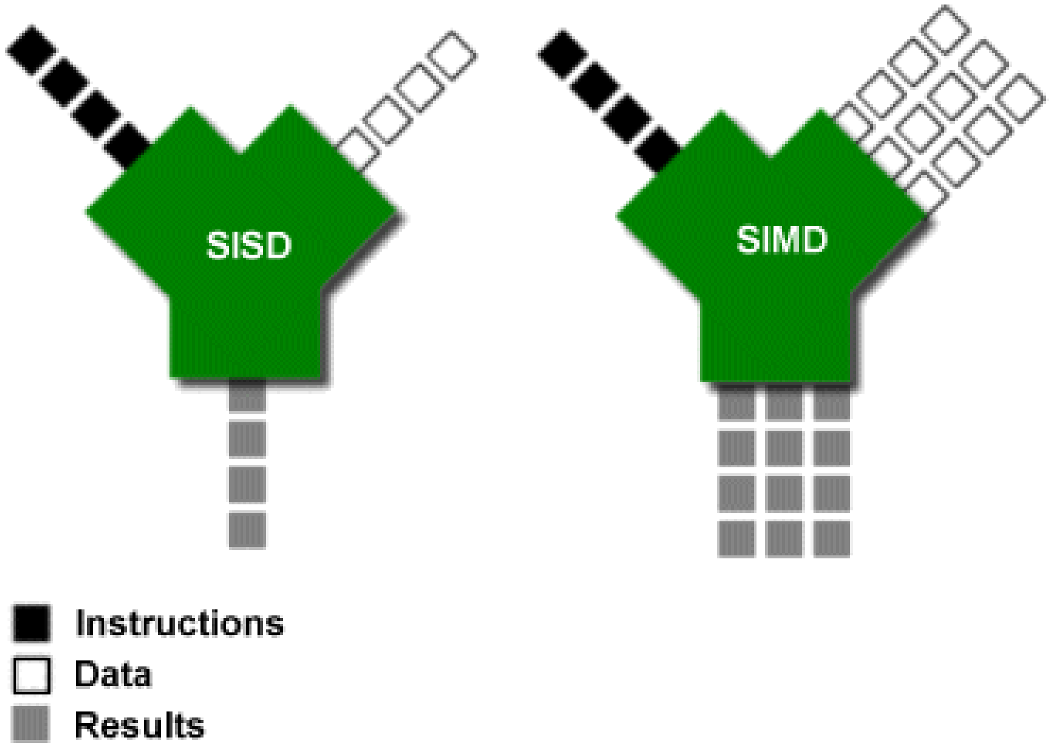
Parallel Image Processing



- SIMD Processor
- DSP
- GPU
- Customized Logic

SIMD - Single Instruction, Multiple Data

- Processors with SIMD, comparing to SISD (single-instruction-single-data, ordinary CPUs) can perform the same operation on multiple data simultaneously.



SIMD - Single Instruction, Multiple Data

- Processors with SIMD, comparing to SISD (single-instruction-single-data, ordinary CPUs) can perform the same operation on multiple data simultaneously.
- For example, grayscale computing for every pixel is the same: reading values of three color channels, multiplying by same coefficients, and adding together.
- In ordinary CPUs, pixels have to be computed one-by-one.
- SIMD processors allow to compute several pixels (multiple data) using the algorithm above (single instruction) simultaneously with less time consumption.

SIMD Implementations

- Arm Neon technology
  - Introduced from Arm Cortex-A8/A9
- Intel MMX/SSE and later versions
  - Widely used in modern x86 based processors
- SPE (Signal Processing Engine) for PowerPC
- AMD 3DNow!

- Beyond those, modern GPUs are often SIMD implementations.

Introduction to Neon technology

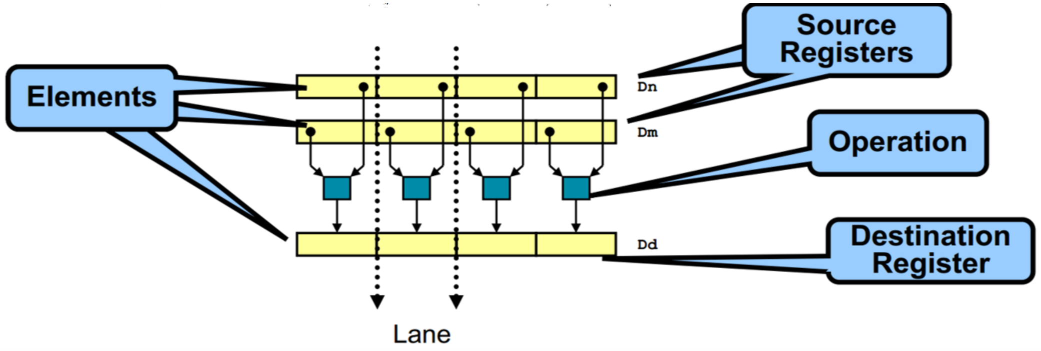
- Arm Neon technology is an Advanced SIMD (single instruction multiple data) architecture extension for the Arm Cortex-A series and Cortex-R52 processors.

	Armv7-A/R	Armv8-A/R	Armv8-A
		AArch32	AArch64
Floating-point	32-bit	16-bit*/32-bit	16-bit*/32-bit/64-bit
Integer	8-bit/16-bit/32-bit	8-bit/16-bit/32-bit/ 64-bit	8-bit/16-bit/32-bit/ 64-bit

*Only in Armv8.2-A

Introduction to Neon technology

- Neon is a wide SIMD data processing architecture
  - Extension of the Arm instruction set
  - Thirty-two registers, 64-bits wide (dual view as sixteen registers, 128-bits wide)
- Neon instructions perform “packed SIMD” processing
  - Registers are considered vectors of elements of the same data type
  - Data types can be signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit, single-precision float
  - Instructions perform the same operation in all lanes



Introduction to Neon

- General purpose SIMD processing useful for many applications
- Supports widest range of multimedia codecs used for internet applications
  - Many soft codec standards; e.g., MPEG-4, H.264, On2 VP6/7/8, Real, AVS
  - Ideal solution for normal sized “internet streaming” decoding of various formats
- Fewer cycles needed
  - Neon gives a 60-150% performance boost on complex video codecs
  - Simple DSP algorithms demonstrate a larger performance boost (4x-8x)
  - Balance of computation and memory access is required
  - Processor can sleep sooner →overall dynamic power saving

How to Use Neon

- Neon optimized open-source libraries
  - OpenMAX DL (development layer): APIs contain a comprehensive set of audio, video, and imaging functions that can be used for a wide range of accelerated codec functionality, such as MPEG-4, H.264, MP3, AAC, and JPEG.
  - Broad open-source support for Neon
- Vectorizing compilers
  - Exploits Neon SIMD automatically with existing C source code
- Neon intrinsics
  - C function call interface to Neon operations
  - Supports all data types and operations supported by Neon
- Assembler code
  - For those who want to optimize at the lowest level

Neon Vector Data Types

- Neon Support in C defines data types for vectors according to the following pattern:

```
<type><size>x<number of lanes>_t
```

- `int8x8_t int16x4_t int32x2_t int64x1_t uint8x8_t uint16x4_t uint32x2_t uint64x1_t float16x4_t float32x2_t poly8x8_t poly16x4_t int8x16_t int16x8_t int32x4_t int64x2_t uint8x16_t uint16x8_t uint32x4_t uint64x2_t float16x8_t float32x4_t poly8x16_t poly16x8_t`

- For example, `int16x4_t` is a vector containing four lanes each containing a 16-bit integer.
- There are array types defined for array lengths between 2 and 4:

```
struct int16x4x2_t
{
    int16x4_t val[2];
};
```

Neon Intrinsic

- The Neon intrinsics Arm provided to generate Neon code for ArmV7 or later processors.
- The Neon intrinsics are defined in the header file arm_neon.h.

- The intrinsics use a naming scheme that is similar to the Neon assembler syntax:
- `v<opname><flags>_<type>`
- An additional q flag is provided to specify that the intrinsic operates on 128-bit vectors.

- For Example:
- `uint16x8_t vmull_u8 (uint8x8_t a, uint8x8_t b)`

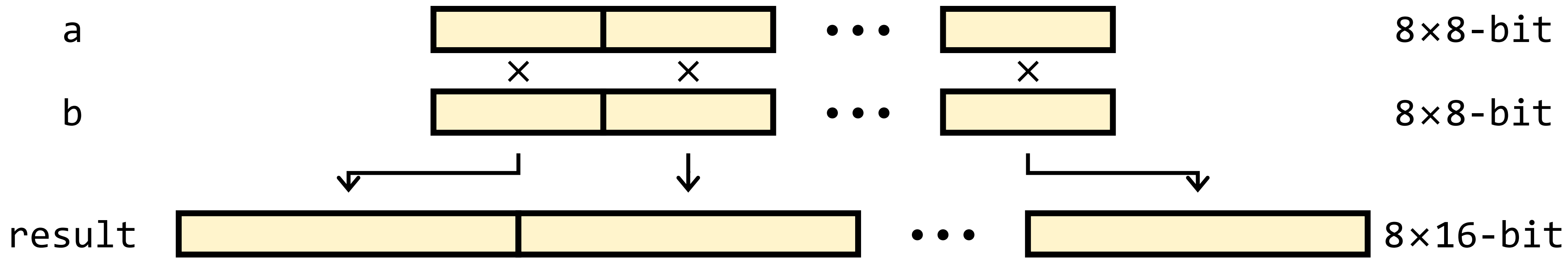
Neon Intrinsic Example

```
uint16x8_t vmull_u8 (uint8x8_t a, uint8x8_t b)
```

- It will be compiled to

```
a → Vn.8B, b → Vm.8B
UMULL Vd.8H,Vn.8B,Vm.8B
Vd.8H → result
```

- which performs multiplication of two 64-bit vectors containing unsigned 8-bit integers, resulting in a 128-bit vector of unsigned 16-bit integers.



More Intrinsics

- There are more than 1000 intrinsics available.

```
• int16x4_t vadd_s16    (int16x4_t a, int16x4_t b);   // 64-bit registers
• int16x8_t vaddq_s16   (int16x8_t a, int16x8_t b);   // 128-bit registers
• int32x4_t vaddl_s16   (int16x4_t a, int16x4_t b);   // long form
• int32x4_t vaddw_s16   (int32x4_t a, int16x4_t b);   // wide form
• int16x4_t vqadd_s16   (int16x4_t a, int16x4_t b);   // saturating form
• int16x8_t vqaddq_s16  (int16x8_t a, int16x8_t b);
• int8x8_t  vaddhn_s16  (int16x8_t a, int16x8_t b);   // narrow form
• int8x8_t  vraddhn_s16 (int16x8_t a, int16x8_t b);   //  + rounding
• int16x4_t vhadd_s16   (int16x4_t a, int16x4_t b);   // halving add
• int16x8_t vhaddq_s16  (int16x8_t a, int16x8_t b);
• int16x4_t vrhadd_s16  (int16x4_t a, int16x4_t b);   //  + rounding
• int16x8_t vrhaddq_s16 (int16x8_t a, int16x8_t b);
• int16x4_t vpadd_s16   (int16x4_t a, int16x4_t b);   // pairwise
• int32x2_t vpaddl_s16  (int16x4_t a);                // long pairwise
• int32x4_t vpaddlq_s16 (int16x8_t a);
```

Comparing between Ordinary Processor and Neon

Ordinary Processor

Loop for 8 times:

```
for (i = 0; i < 8; i++)
{
    result[i] = a[i] * b[i];
}
```

Neon Engine

Compute in one single instruction:

```
result = vmull_u8 (a, b);
```

or:

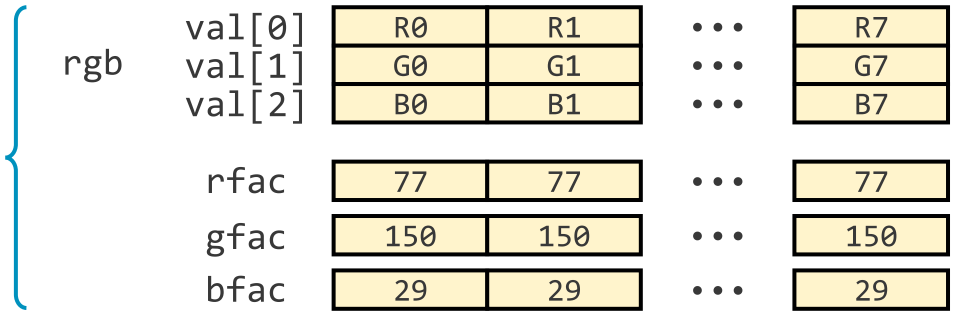
```
a → Vn.8B, b → Vm.8B
UMULL Vd.8H,Vn.8B,Vm.8B
Vd.8H → result
```

Example: Using Neon Engine for 8-Pixel Grayscale

- $A = (77R + 150G + 29B)/256$

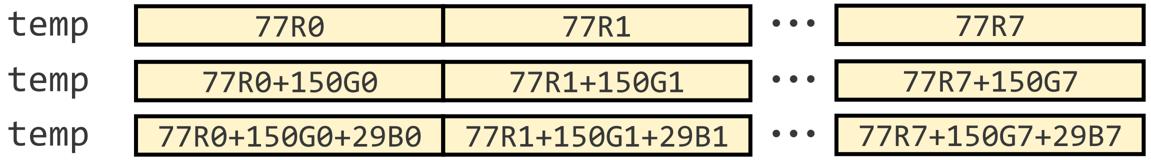
Step 1: Load data and constants

```
uint8x8x3_t rgb  = vld3_u8   (pixel);
uint8x8_t   rfac = vdup_n_u8 (77);
uint8x8_t   gfac = vdup_n_u8 (150);
uint8x8_t   bfac = vdup_n_u8 (29);
```



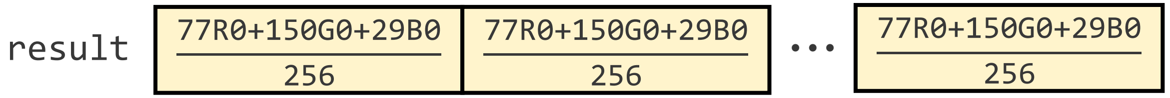
Step 2: Compute grayscale

```
uint16x8_t  temp;
temp = vmull_u8 (      rgb.val[0], rfac);
temp = vmlal_u8 (temp, rgb.val[1], gfac);
temp = vmlal_u8 (temp, rgb.val[2], bfac);
```



Step 3: Normalize results

```
result = vshrn_n_u16 (temp, 8);
```

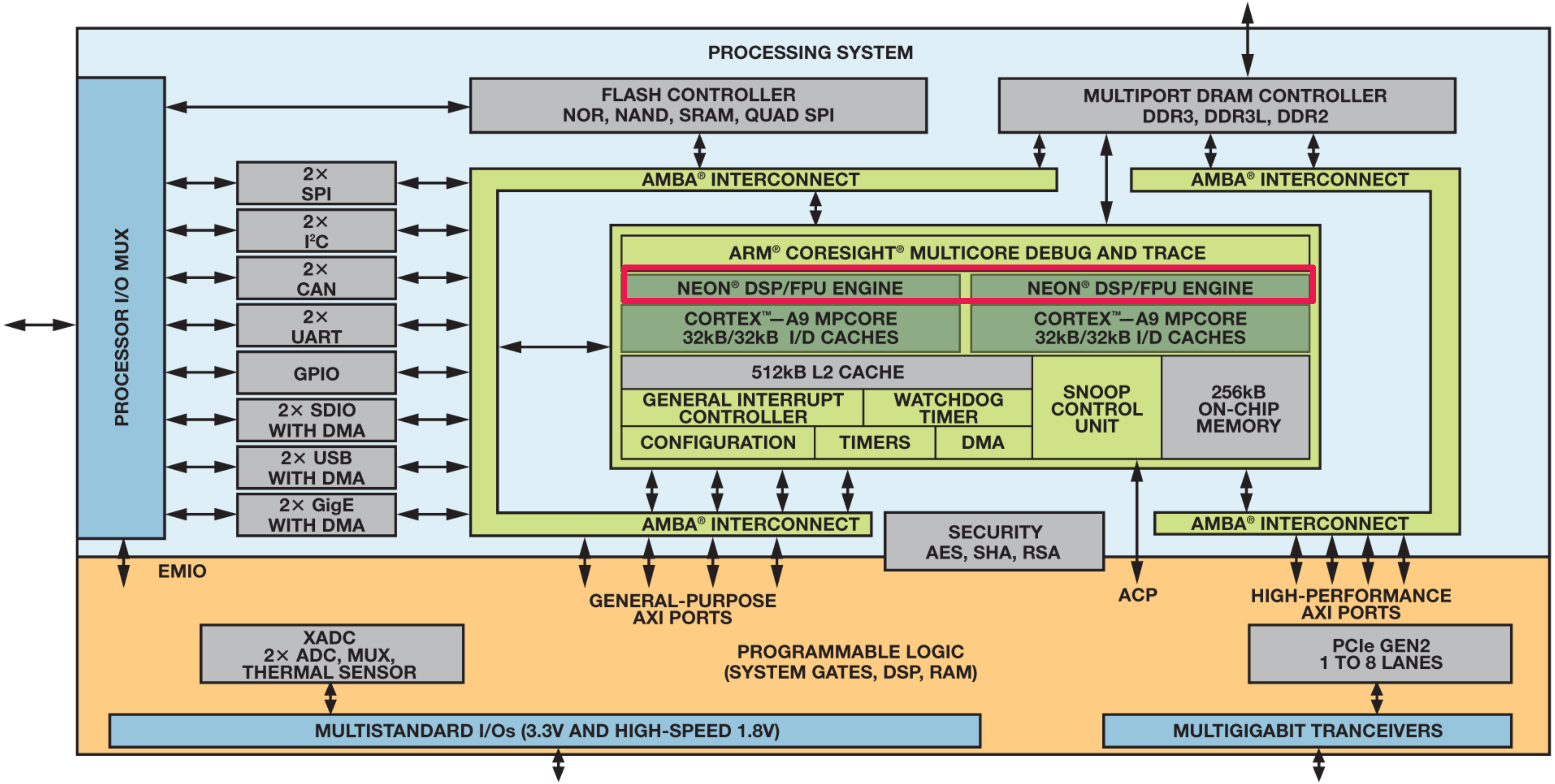


Neon Ecosystem

- Arm Compute Library
  - The Compute Library contains a comprehensive collection of software functions, from basic mathematical operator to machine learning support like Convolutional Neural Networks, implemented for the Arm Cortex-A family of CPU processors (and the Arm Mali family of GPUs).
  - It is a convenient repository of low-level optimized functions that developers can source individually or use as part of complex pipelines in order to accelerate their algorithms and applications.

- Partner’s Modules
  - A wide range of codecs and DSP modules are available from several partners.
  - Video codecs, audio codecs, computer vision, machine learning, etc..

Neon Engine on Zynq-7000 Platform



Limitations and Tips

- Loading Neon registers costs delay.
  - Make Neon codes together.
  - Consider prefetch.

- Optimize your codes using libraries or assembler.
  - You may get higher efficiency by writing assembler Neon codes by yourself, or using highly optimized libraries.