MVE Introduction

Learning objectives

At the end of this topic you will be able to:

- Describe why Arm introduced MVE to the Armv8.1-M architecture
- Summarize the main features of MVE
- Describe the software and tools support for MVE

Agenda

- **MVE Overview**
- MVE Features
- MVE Software Support

What is MVE ?

SIMD data processing engine

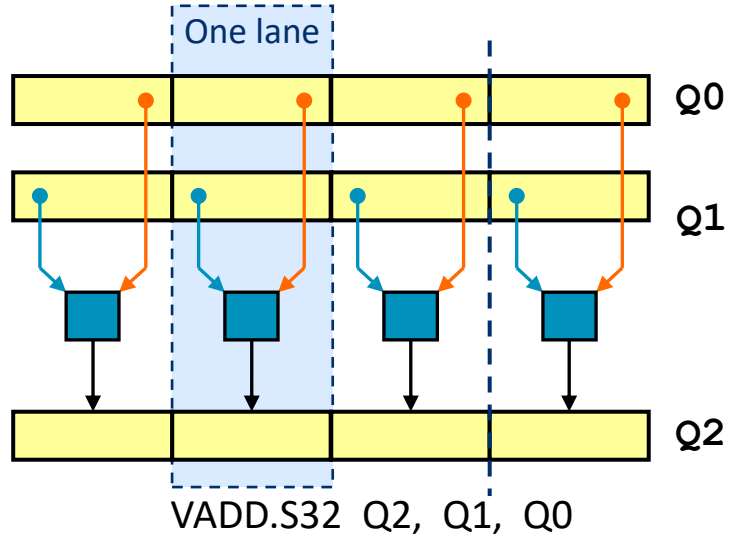
- A **S**ingle **I**nstruction operates on **M**ultiple **D**ata
- For example, a 128-bit register can be used to hold 16 independent bytes

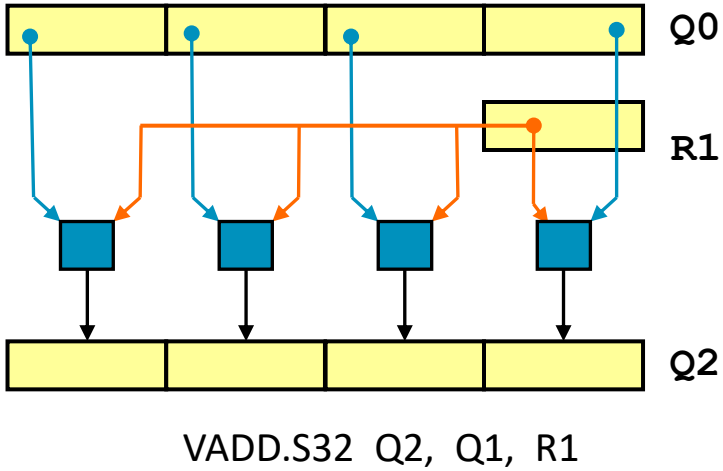
The M-Profile Vector Extension (MVE) is

- advanced SIMD technology on M-profile
- an optional architectural extension that enables higher signal processing and machine learning capabilities.

Instructions operate on vectors of elements of the same data type

- Most instructions perform the same operation in all **lanes**

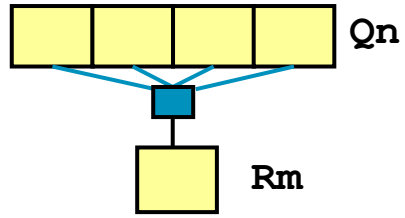




What is MVE ?

MVE supports various data types

- 8-bit, 16-bit, 32-bit, 64-bit integer
- Single/half precision floating point



VADDV.S32 R1, Q0

Some MVE instructions can operate on vector horizontally

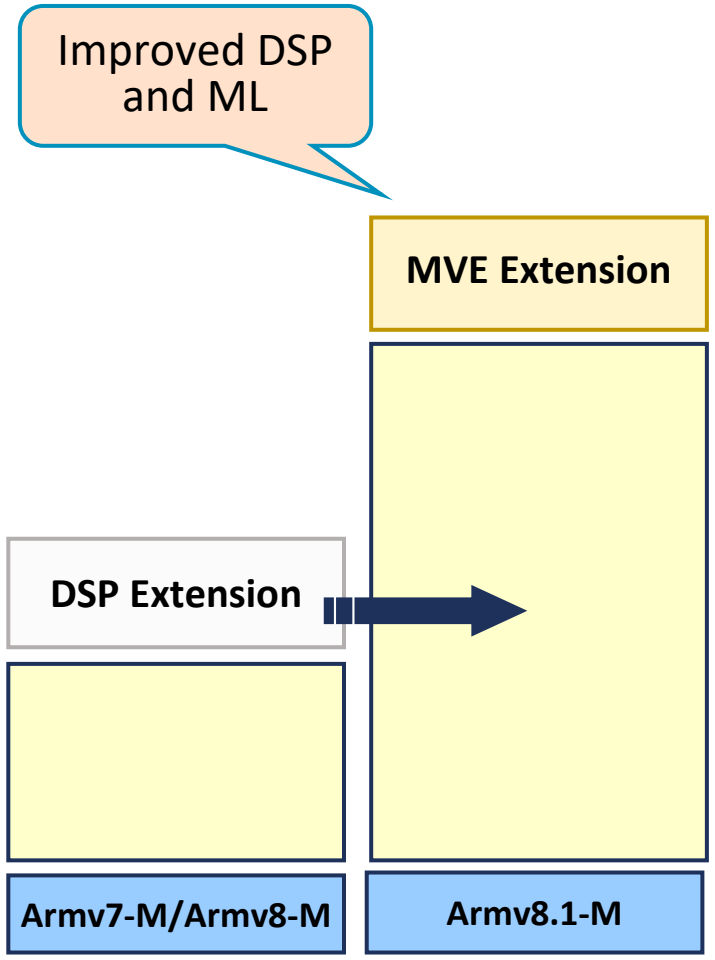
MVE key features

Advanced SIMD technology
- 128-bit vector size
- Supports 8/16/32b INT and 16/32 IEEE FP data types

Efficient loop and easier for vectorization
- Low overhead loops
- Lane predication
- Loop-tail predication

Efficient memory access
- Vector gather load and vector scatter store
- Interleaving and de-interleaving load and store

MVE compared to DSP extension



Armv7-M/Armv8-M DSP Extension	Armv8.1-M MVE Extension
Operates on standard 32-bit general purpose Arm registers	Separate register bank 8 x 128-bit wide registers
Up to 4 operations per instruction	Up to 16 operations per instruction
Specific operations targeted at particular algorithms	Cover most application requirement
Integer operations only	Supports integer and single/half precision floating point
	Good compiler target

Benefit of MVE

Reduce silicon design complexity

- Only requires a single memory system, hence enabling faster system-on-chip(SOC) design cycle and reducing costs

Fewer cycles needed than for (regular) Thumb code

- Delivering up to 15x performance uplift for machine learning and up to 5x uplift to signal processing tasks on the smallest of edge devices
- Processor can sleep sooner → overall dynamic power saving

Cost effective alternative to a separate DSP

- Easier to program than a separate DSP
- Coding and debugging use the same tools as the regular Arm core

MVE is implemented as a part of Armv8.1-M Architecture

- Arm is actively working with third parties on MVE application development

MVE uses a well-designed instruction set

- Suitable for hand coding or code generation by a vectorizing compiler

Agenda

- MVE Overview
- **MVE Features**
- MVE Software Support

Vector Extension operation

MVE-I

- Integer MVE only + optional scalar FPU (double precision support also optional)
- Operates on 32-bit, 16-bit, and 8-bit data types, including Q7, Q15, Q31 fixed-point integer values

MVE-F

- Integer + floating point MVE (support vectored single precision and half precision) + scalar FPU (double precision support also optional)
- Operates on half-precision and single-precision floating-point values

Vector instructions operate on a fixed vector width of 128 bits

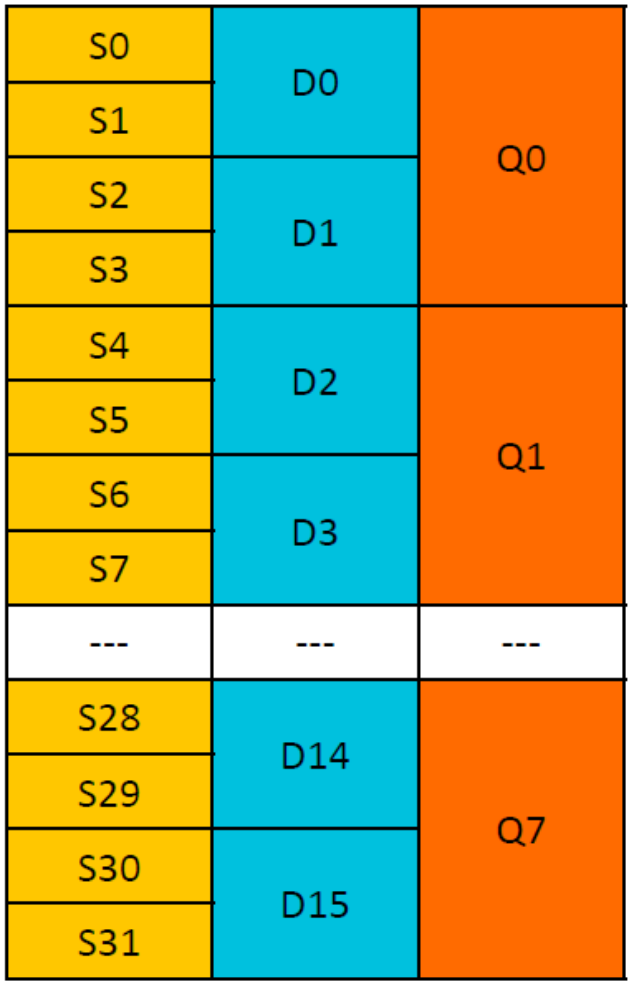
	FPU (HP,SP)	FPU (DP)	MVE integer	MVE float
Minimum	✗	✗	✗	✗
FPU	✓	✗	✗	✗
	✓	✓	✗	✗
MVE, no FPU	✗	✗	✓	✗
MVE, FPU scalar	✓	✗	✓	✗
	✓	✓	✓	✗
MVE, FPU	✓	✗	✓	✓
	✓	✓	✓	✓

Armv8.1-M implementations

Vector register file

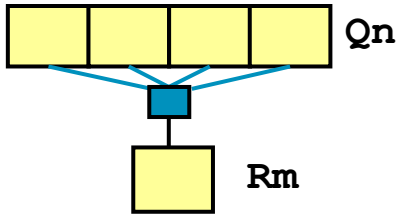
MVE defines 8 vector registers that alias onto the Floating-point Extension register file

- Provides Q, D, S register views
- Q, D, S registers are overlapped physically



If CP10 is enabled, access to vector register 0-7 is permitted.

To reduce pressure on the vector register file, many vector instructions can use scalar arguments from the general-purpose register file



VADDV.S32 R1, Q0

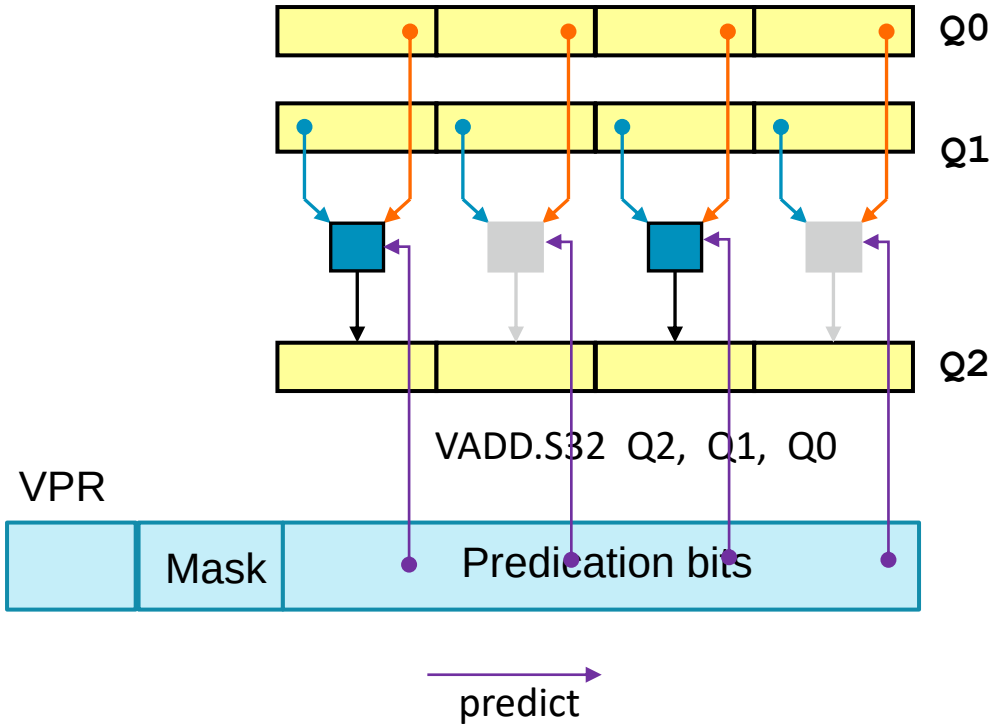
VPR register

MVE supports conditional execution for each of the lanes in the vector, which is called lane predication

- With support of a new special purpose register, Vector Predication Status and Control Register (VPR)

VPR register holds the condition for each lane

- The predication condition in this register is updated by vector instructions, such as vector compare
- The predication condition in this register is used in VPT block, see more later
- VPR can be accessed by VMSR/VMRS instruction



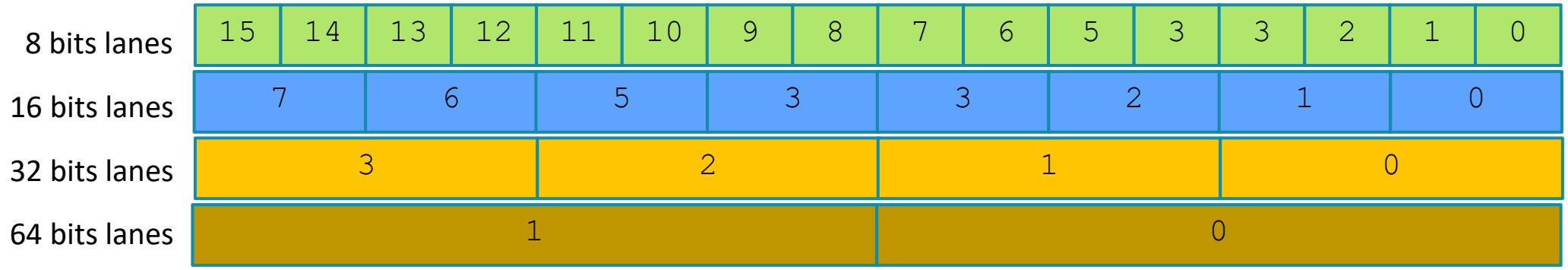
Lanes

The lane width of the operation to be performed is specified by the instruction that is being executed.

The permitted lane widths are:

- 64-bit, 32-bit, 16-bit, 8-bit

The word ‘Element’ is used to refer to the data that is put into a lane.

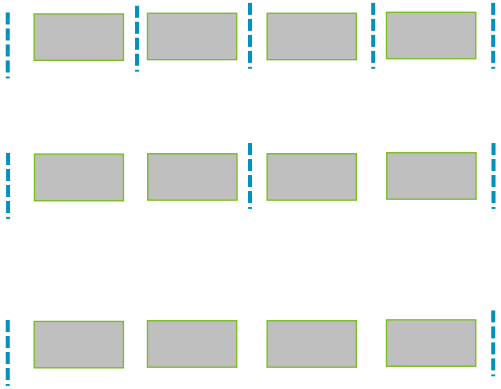


Beats

A vector instruction executes in 4 beats sequentially, from beat 0 to beat 3.

It is IMPLEMENTATION DEFINED how many beats are executed for each architecture tick.

- In a single-beat system, one beat might occur for each tick
- In a dual-beat system, two beats might occur for each tick
- In a quad-beat system, four beats might occur for each tick

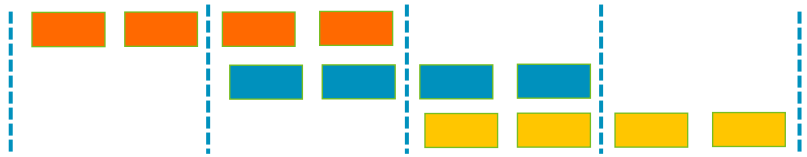


Beats overlapping

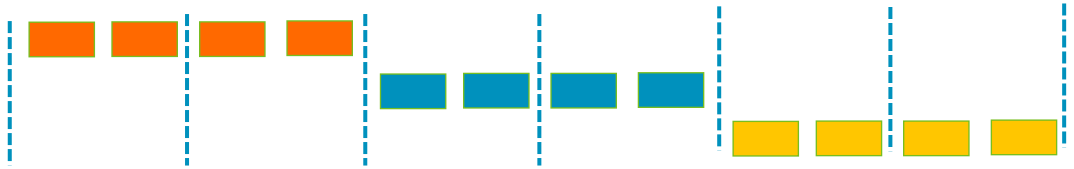
vector instructions are allowed to overlap by 2 beats

- 4 beats/tick

- 2 beats/tick

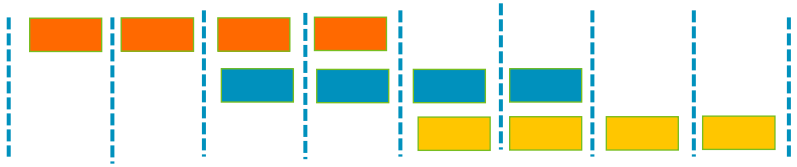


Overlapping



No overlapping

- 1 beat/tick



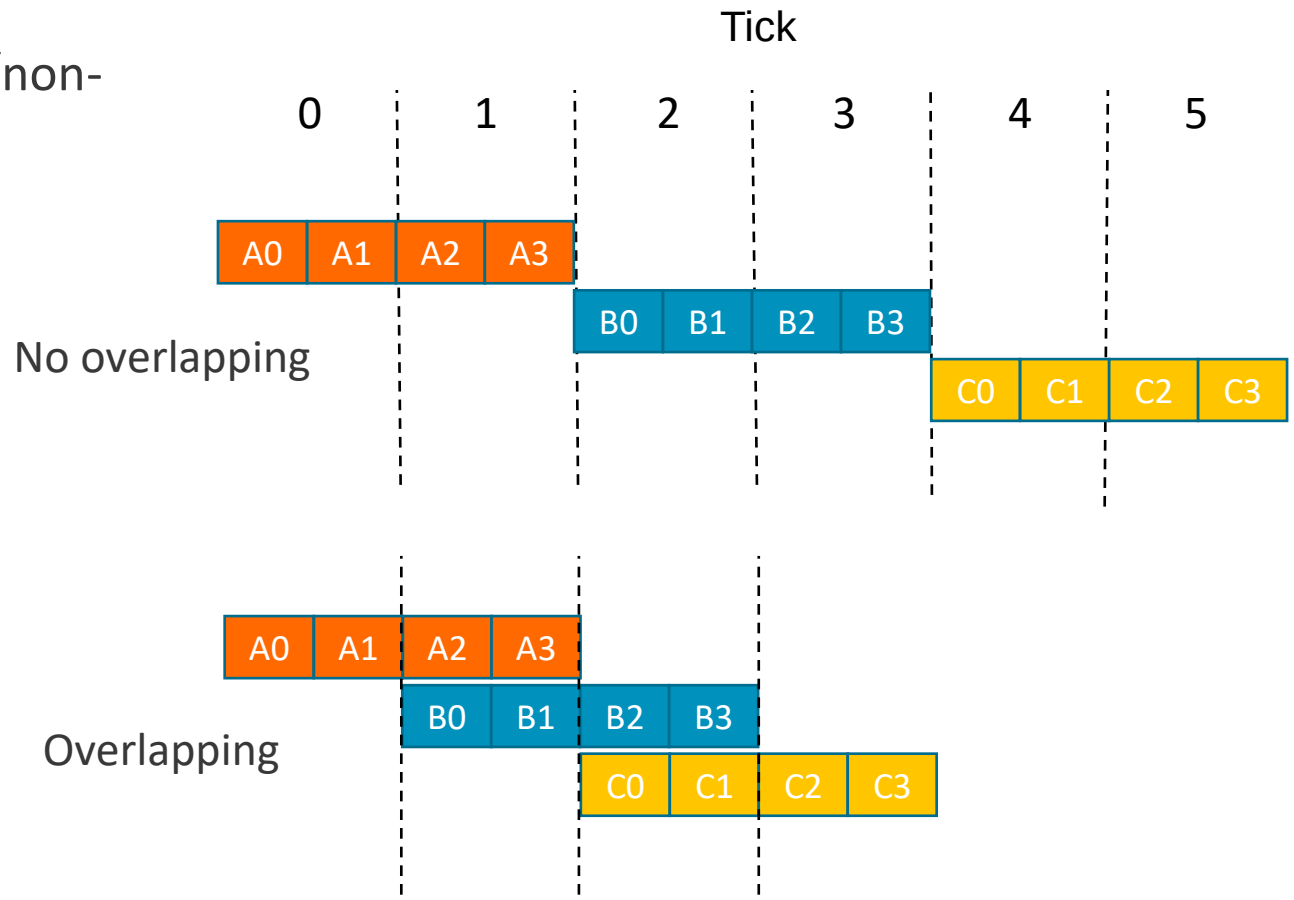
Overlapping

No overlapping

Dual-beat overlap example

Execute following instructions in overlap/non-overlap dual-beat system

```
VLDRW.U32 Q1, [R0], #16
VMUL.I32  Q0, Q1, Q2
VSHR.U32  Q0, Q0, #1
```



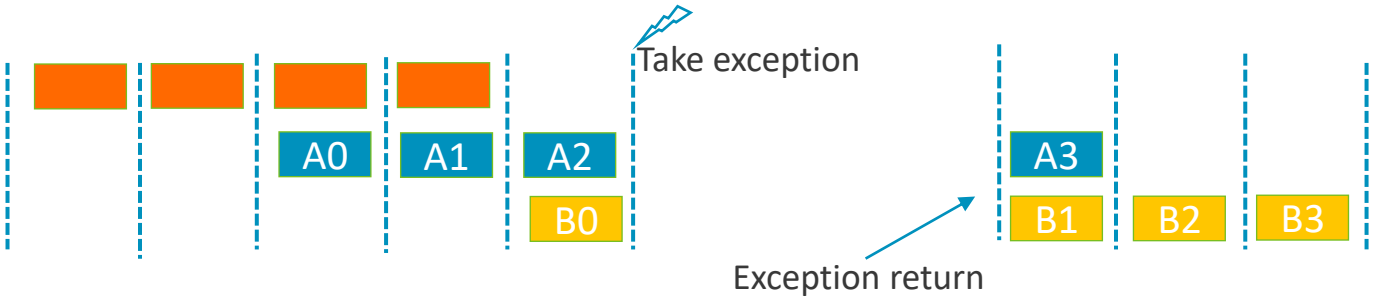
Tick: Architecture tick

- A0-A3 Beats of the VLDRW instruction
- B0-B3 Beats of the VMUL instruction
- C0-C3 Beats of the VSHR instruction

Exception state

For exceptions that occur in the middle of a beat-wise vector instruction that is executing:

- The exception return address points to the oldest incomplete instruction
- RETPSR.ECI in the exception stack frame stores information about how many beats of the instruction at the return address, and how many beats of the subsequent instruction, have already been executed



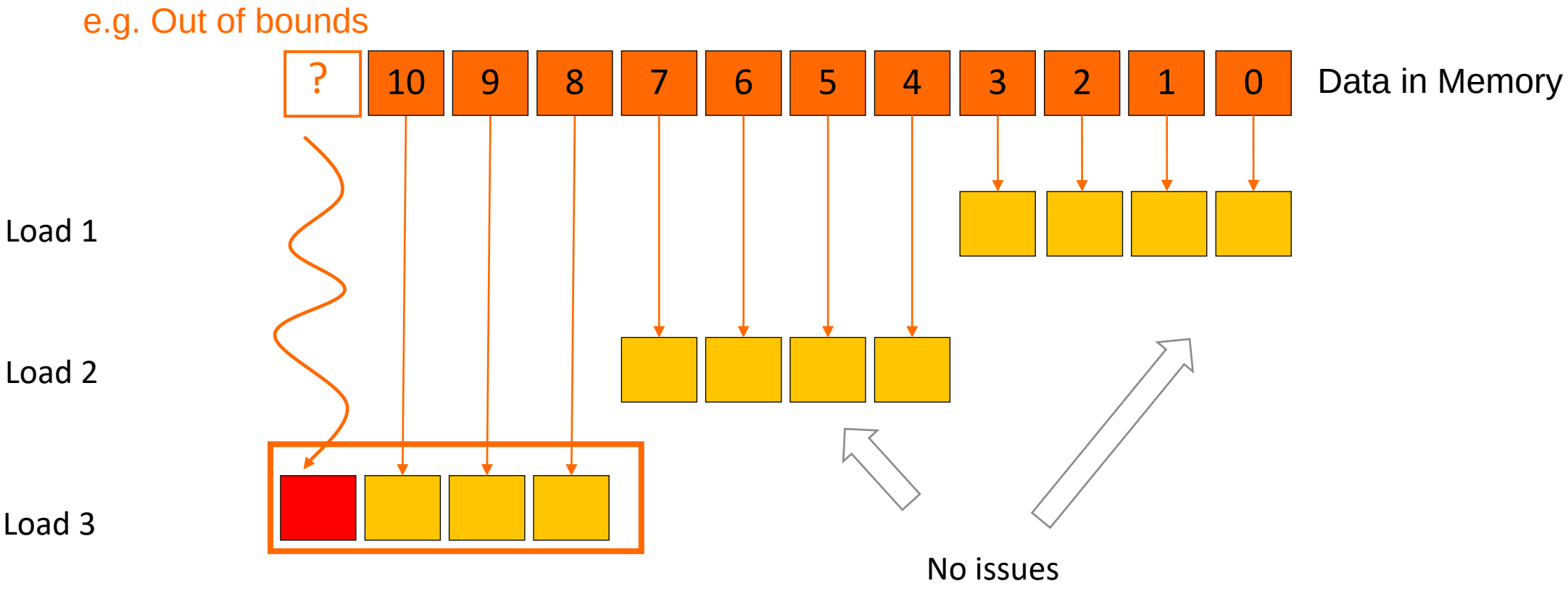
ECI	0x0	0x1	0x2	0x4	0x5
Completed beats	No beat	A0	A0, A1	A0, A1, A2	A0, A1, A2, B0

VPR is saved and restored automatically in exception events

- Using a reserved word inside the extended exception stack frame

Loop tail predication(1)

What happens when the input data is not a multiple of the vector length?



Loop tail predication(2)

MVE loop tail prediction can process loop that iteration number is not multiple of the vector length

- Supported by **WLSTP**, **DLSTP** & **LETP** instructions
- **FPSCR.LTPSIZE** register bits are used to indicate tail predication state and element size
  - tail predication is disabled by setting FPSCR.LTPSIZE to 0b100
- There is implicit lane predication in tail predication state
  - if the number of remaining elements is less than the vector length, the corresponding elements of the vector are disabled

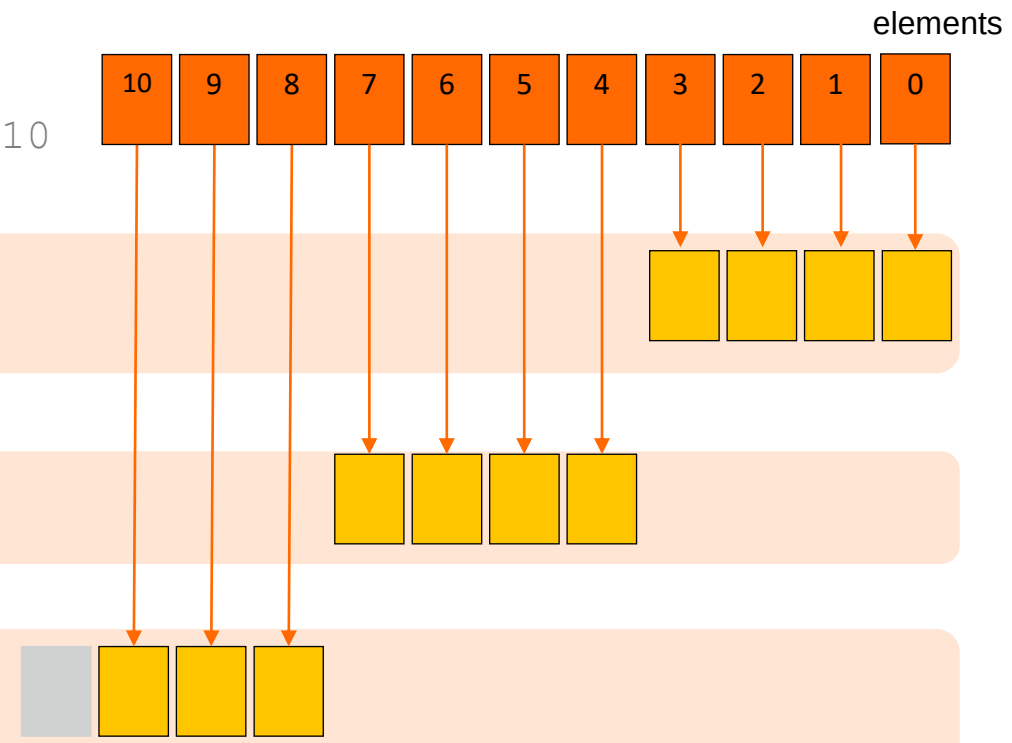
```
      WLSTP.32   LR, R2, end    ; Enable word predication(.32), "LR" set to number of
                                ; elements, FPSCR.LTPSIZE is set to element size (not 0b100)
start:
      VLDRW.U32 Q0, [R1], #16   ; If "LR" is smaller than "elements per vector",
      VMLA.U32   Q1,  Q0,  R0   ; LDR/MAC only on the remaining elements

      LETP       LR, start      ; Decrement "LR" by "number of active elements per vector
                                ; for the last loop, if (number of remaining elements) <=
                                ; (number of elements per vector), sets FPSCR.LTPSIZE=0b100
end:
```

Loop tail predication(3)

For example, a loop with 11 32-bit elements

```
    WLSTP.32   LR, R2, end
start:
    VLDRW.U32 Q0, [R1], #16
    VMLA.U32   Q1,  Q0,  R0

    LETP       LR, start
end:
```

```
    ;R2=11, number of elements
    WLSTP.32   LR, R2,  end   ; LR=11,
                              ;FPSCR.LTPSIZE=0b010
start:
    VLDRW.U32 Q0, [R1], #16
    VMLA.U32   Q1,  Q0,  R0

    LETP       LR,   start   ;LR=11-4
    VLDRW.U32 Q0, [R1], #16
    VMLA.U32   Q1,  Q0,  R0
    LETP       LR,   start ;LR=7-4

    VLDRW.U32 Q0, [R1], #16
    VMLA.U32   Q1,  Q0,  R0

    LETP       LR,   start   ;LR=3-3, FPSCR.LTPSIZE=0b100
end:
```



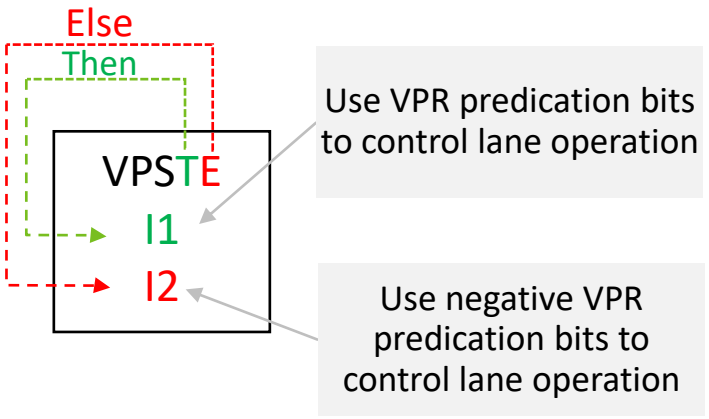
Execution trace

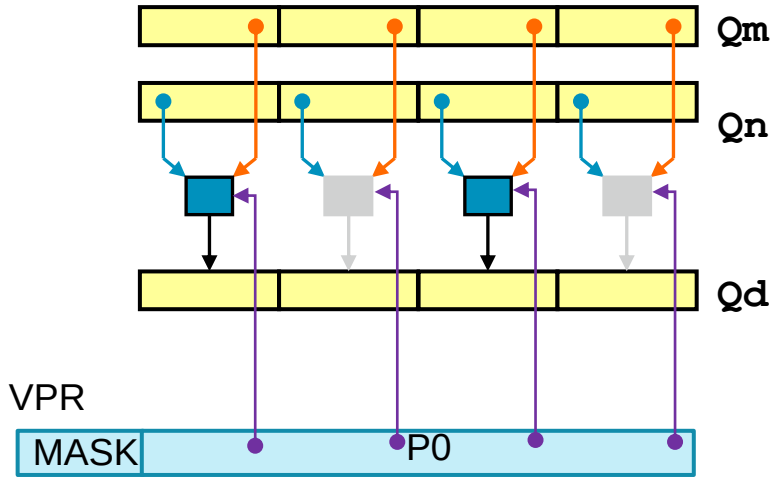
VPT predication(1)

Comparison based predication is supported by vector predication blocks – VPT blocks

- The block is marked by a **VPT** or **VPST** instruction
  - N instructions following a `VPT` or `VPST`, N<=4
  - `VPT` and `VPST` instructions are similar to If-Then (`IT`) instructions
    - Could be used in situation that there is 'if-else' in a loop

```
for(i = 0; i<N; i++)
 {
      if (data[i] > m)
      {
        …
      }
      else
      {
        …
      }
   }
```





VPT predication(2)

Comparison and Predication instructions

- VPST
  - It predicates the following, up to maximum of four instructions. Each of these instructions is subject to the current predication condition held in the VPR register
  - For example, VPST, VPSTE, VPSTTE, VPSTTEE
- VPT
  - Performs a lane-wise comparison and updates the predication flag VPR.P0
  - The VPR is used to predicate the following, up to maximum of four instructions. Each of these instructions is subject to the current predication condition held in the VPR register
  - For example,

```
VPTE.S32 GT, Q0, R0     ; Compare each lane with R0, set VPR.P0 and VPR.MASK to  predict following 2 instructions
```

- VCMP
  - Performs a lane-wise comparison and updates the predication flag VPR.P0, so can affect the following instructions in the VPT block
  - For example,

```
VCMP.S32 LE, Q0, Q1      ; set VPR.P0 on Q0[i] <= Q1[i] (i=0..3)
```

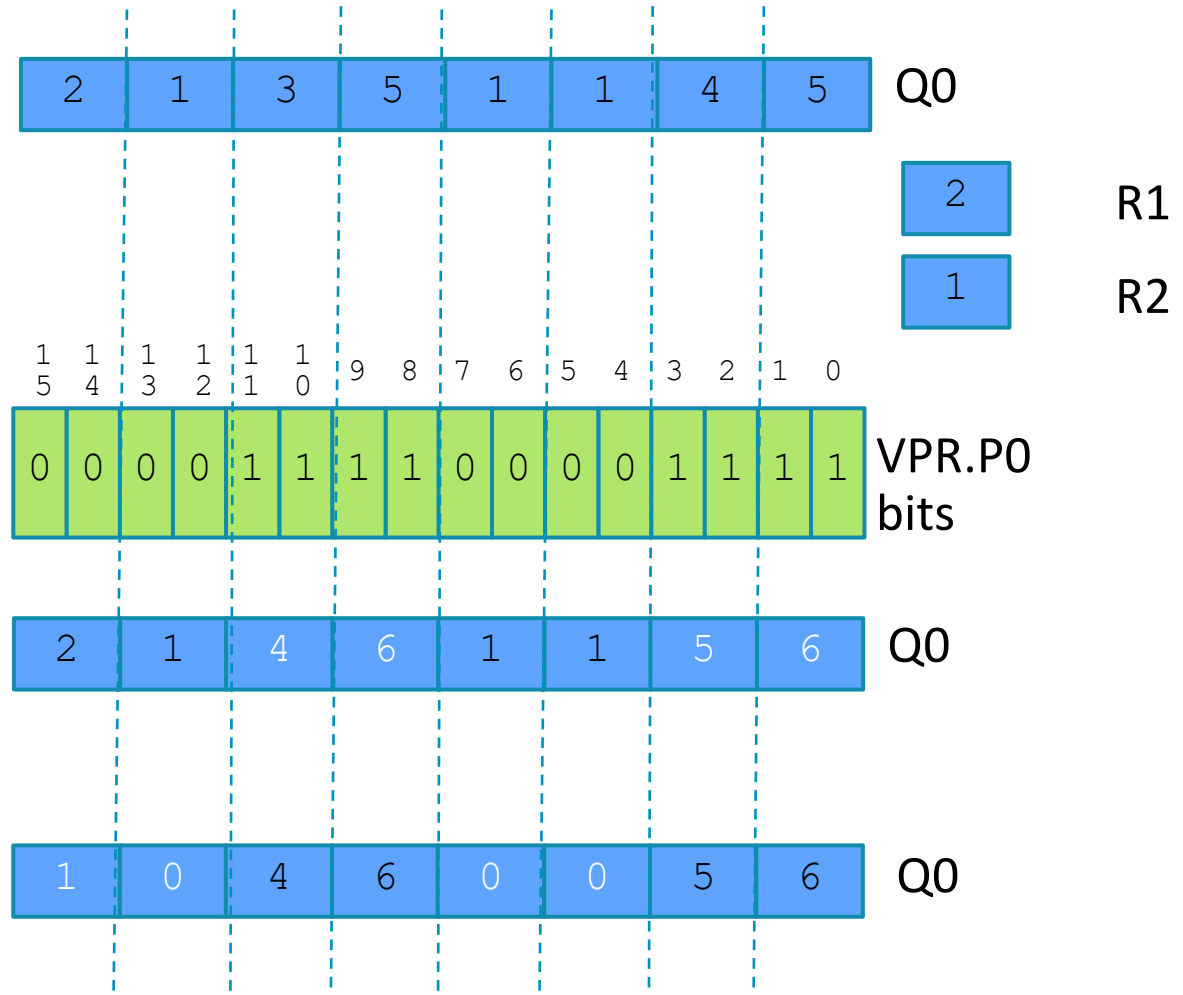
VPT predication(3)

Comparison based predication is supported by vector predication blocks – VPT blocks

An example in quad-beat system

```
; R1=2, R2=1
VPTE.S16  GT, Q0, R1
VADDT.I16 Q0, Q0, R2
VSUBE.I16 Q0, Q0, R2
```

```
for(i = 0; i<N; i++)
{
  if (data[i] > 2)
  {
    data[i]+=1;
  }
  else
  {
    data[i] -=1;
  }
}
```

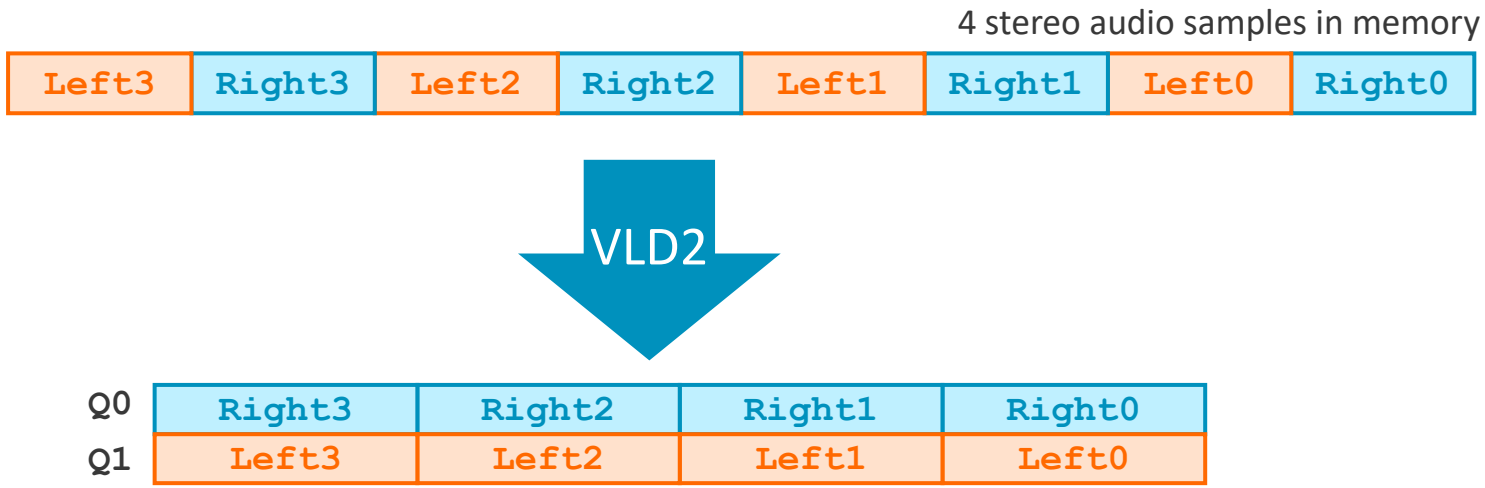


Interleaving and de-interleaving load and store (1)

Data streams can be interleaved and de-interleaved with strides of 2 and 4, using **VLD2**/**VLD4** and **VST2**/**VST4**

This is useful for processing signals that have 2 or 4 sets of values for each sample, for example:

- 2 sets of values for a stereo audio sample
- 4 sets of values for an RGB+Alpha image pixel



Pattern Variants (2)

Interleaving and de-interleaving load and store instructions have multiple “pattern variants”

- VLD20, VLD21 , VST20, VST21
- VLD40, VLD41, VLD42, VLD43, VST40, VST41, VST42, VST43

They are designed to only be used in a full sequence of each variant

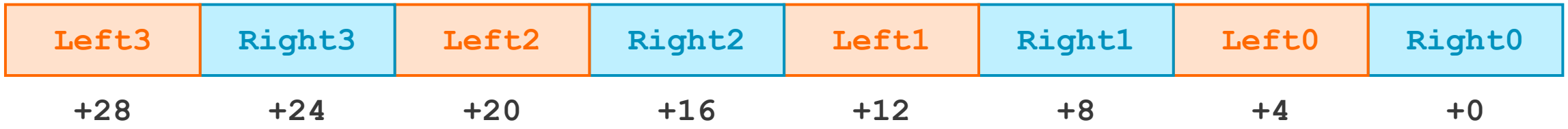
- Effectively, VLD20 + VLD21 = VLD2, …

Pattern Variants are introduced to address hardware resource limitation on M-profile CPUs for MVE Interleaving and de-interleaving load and store

- Better fit for beatwise execution
- Friendly memory access by avoiding non-contiguous bytes
- Efficient register files access

Interleaving and de-interleaving load and store (3)

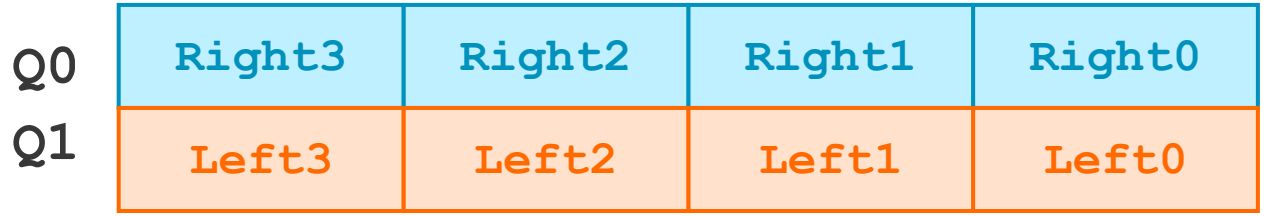
4 stereo audio samples in memory



How do we apply different calculations to each channel?

1. De-interleave using set of **VLD2** instructions, copying into **Q0** and **Q1**, assuming base pointer is register **R0**:

```
VLD20.F32 {q0, q1}, [r0]
VLD21.F32 {q0, q1}, [r0]
```



Interleaving and de-interleaving load and store (4)

2. Process the two channels separately as needed.
   For example, amplify each channel using a different factor:

```
VMUL.F32 q0, q0, r2
VMUL.F32 q1, q1, r3
```

Assuming R2 and R3 are the amplification factors for the right and left channels respectively.

Q0	Right3	Right2	Right1	Right0
Q1	Left3	Left2	Left1	Left0

3. Re-interleave using set of **VST2** instructions, copying from **Q0** and **Q1**:

```
VST20.F32 {q0, q1}, [r0]
VST21.F32 {q0, q1}, [r0]
```

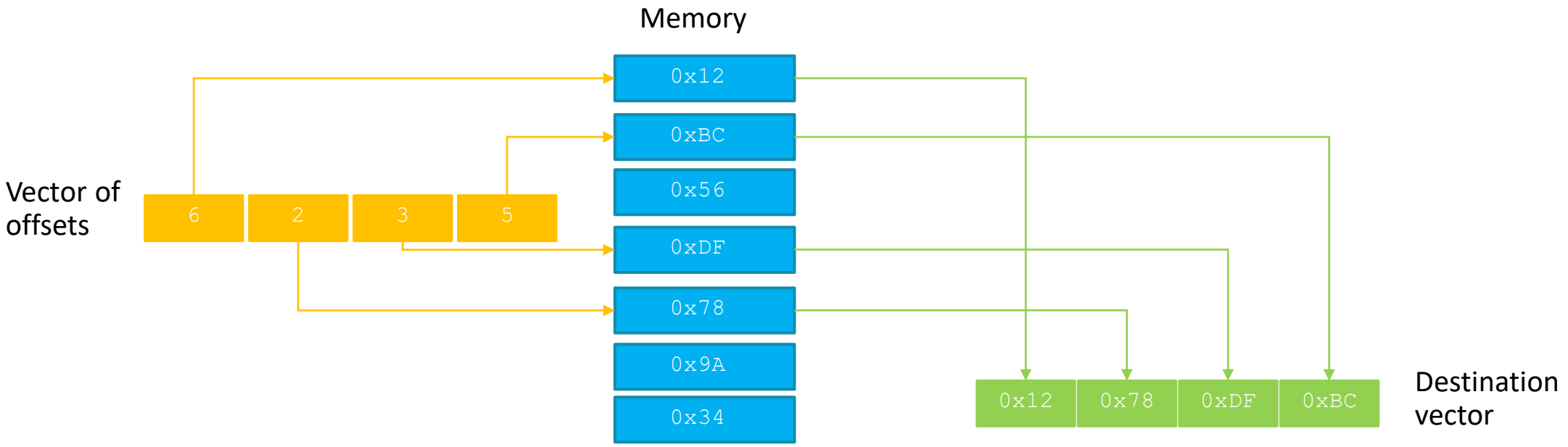
Left3	Right3	Left2	Right2	Left1	Right1	Left0	Right0
+28	+24	+20	+16	+12	+8	+4	+0

Vector gather load and vector scatter store

Sometimes, data stored in memory cannot be structured in a convenient way that makes contiguous accesses possible

MVE solves this problem with scatter-gather operations

These point a vector of offsets into memory so that multiple non-contiguous addresses can be accessed with a single instruction



Agenda

- MVE Overview
- MVE Features
- **MVE Software Support**

How to use MVE

CMSIS DSP and NN library

- Set of common functions optimized for Cortex-M and MVE
- Targeted to Microcontroller market
- Source code available from https://github.com/ARM-software/CMSIS_5

Vectorizing Compilers

- Exploit MVE automatically with existing source code
- May require source code changes to get best results

C Intrinsics

- C function call interface to MVE operations
- Models the precise data types and operations supported in hardware while allowing the compiler to deal with scheduling and register allocation

Assembler

- For those who really want to optimize at the lowest level
- Optimizing hand-coded MVE assembly can be **very** difficult (pipeline issues, memory issues, etc.)

CMSIS-DSP and CMSIS-NN libraries

Benefits for embedded developers

- Consistent software interfaces for silicon and middleware vendors
- Simplifies re-use across Cortex-M processor-based devices
- Reduces learning curve, development costs, and time-to-market

Benefits for silicon vendors

- Well-defined APIs and libraries, familiar to developers
- Common components to prevent reinventing the wheel
- Access to a vast middleware ecosystem

Benefits for tool and middleware vendors

- Well-defined deliverables from silicon vendors
- Simplified porting across Cortex-M microcontrollers

Automatic vectorization

Automatic vectorization can generate code targeted for MVE from ordinary C source code

- Low effort to produce vectorized code
- Portable - no compiler-specific source code features need to be used

To enable automatic vectorization with the Arm Compiler 6, use:

- `-fvectorize -O1`, or
- `-O2` and higher

To enable automatic vectorization of floating-point types with Arm Compiler 6:

- compile with `-ffp-mode=fast`, and
- for half-precision floating-point, use the `_Float16` data type instead of `__fp16` or `float16_t`

Note: When using and linking libraries, both the compiler and linker can still introduce MVE instructions with vectorization disabled

Arm Compiler 6 MVE automatic vectorization

Simple Precision Float Vector Multiply Case

CMSIS DSP (non ARM_MATH_DSP)

```
void arm_mult_f32(
       float32_t * pSrcA,
       float32_t * pSrcB,
       float32_t * pDst,
       uint32_t blockSize)
{
  while (blockSize > 0U)
  {
    *pDst++ = (*pSrcA++) * (*pSrcB++);
    blockSize--;
  }
}
```

Compiled with

```
--target=arm-arm-none-eabi
-mcpu=cortex-m55
-mfloat-abi=hard -O3
```

VLDRW, VSTRW, VMUL, VSTRW: MVE vector operations

DLSTP/LETP: Do Loop Start/End: Low Overhead loop structure

Assembly (inner loop)

```
.LBB0_4:
        dlstp.32        lr, r3
.LBB0_5:                                @ =>This Inner Loop H
eader: Depth=1
        vldrw.u32       q0, [r0], #16
        vldrw.u32       q1, [r1], #16
        vmul.f32        q0, q0, q1
        vstrw.32        q0, [r2], #16
        letp    lr, .LBB0_5
```

Intrinsics

Intrinsics are pseudo-function calls that the compiler replaces with an appropriate instruction or sequence of instructions

- Architecture specific (unlike C/C++)
- You still have to optimize your code to take advantage of MVE

MVE intrinsics are defined in a special Arm Compiler Toolchain header file **`arm_mve.h`**

This also defines a set of vector data types of different sizes, for example:

`int16x8_t`	vector of eight 16-bit short int values
`float32x4_t`	vector of four 32-bit float values

Example:

```
int32x4_t result, a, b;
result = vaddq_s32(a,b)
```

```
VADD.I32 q0, q1, q2
```

See the online reference documentation here for a complete list of intrinsics which are part of the Arm ABI, and so are portable between the Arm Compiler and GNU toolchains:

https://developer.arm.com/architectures/instruction-sets/simd-isas/helium/mve-intrinsics

MVE Intrinsics

Modified C code

```
void arm_mult_f32_intr_2(
  float32_t * pSrcA,  float32_t * pSrcB,
  float32_t * pDst,  uint32_t blockSize)
{
  uint32_t      blkCnt;
  float32x4_t   vecA, vecB, vecDst; /* vectors of 4 x SP floats */

  blkCnt = blockSize / 4;
  while (blkCnt > 0U)
  {
      vecA = vldrwq_f32(pSrcA); /* 0 offset */
      vecB = vldrwq_f32(pSrcB);
      vecDst = vmulq_f32(vecA, vecB);
      vstrwq_f32(pDst, vecDst);

      blkCnt--;
      pSrcA += 4;
      pSrcB += 4;
      pDst += 4;
  }
}
```

MVE Intrinsics for VLDRW, VSTRW, VMUL:

```
float32x4_t vldrwq_f32(float32_t const *base, const int offset)
float32x4_t vmulq_f32(float32x4_t a, float32x4_t b)
void vstrwq_f32(float32_t * base, const int offset, float32x4_t value)
```

Assembly code

```
arm_mult_f32_intr_2:
        push    {r7, lr}
        lsr.w   lr, r3, #2
        wls     lr, lr, .LBB0_2
.LBB0_1:                                @
 =>This Inner Loop Header: Depth=1
        vldrw.u32       q0, [r0], #16
        vldrw.u32       q1, [r1], #16
        vmul.f32        q0, q0, q1
        vstrb.8         q0, [r2], #16
        le      lr, .LBB0_1
.LBB0_2:
        pop     {r7, pc}
```

Notes:
No Low overhead loop structure
No post-incremented access - will be added in future Arm Compiler 6 releases
VLDRW.x32 means load 32-bit memory Words into 32-bit vector elements (no widening)
VSTRW.x32 means store 32-bit vector elements into 32-bit memory Words (no narrowing)

References

Arm Architecture Reference Manual Armv8-M edition

- https://developer.arm.com/products/architecture/cpu-architecture/m-profile/docs/ddi0553/latest

Arm Helium Technology: M-Profile Vector Extension (MVE) for Arm Cortex-M Processors

- https://www.arm.com/resources/ebook/helium-mve-reference-book

Helium Programmer's Guide

- https://developer.arm.com/architectures/instruction-sets/simd-isas/helium/helium-programmers-guide/coding-for-helium

"Making Helium" series blogs:

- https://community.arm.com/developer/research/b/articles/posts/making-helium-why-not-just-add-neon
- https://community.arm.com/developer/research/b/articles/posts/making-helium-sudoku-registers-and-rabbits
- https://community.arm.com/developer/research/b/articles/posts/making-helium-going-around-in-circles
- https://community.arm.com/developer/research/b/articles/posts/making-helium-bringing-amdahl-s-law-to-heel

Thank You
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה