CE515
Advanced Software Architecture

Nicolas Barbot

nicolas.barbot@esisar.grenoble-inp.fr

2020-2021

Goals

- Increase the performance of embedded systems
- Decrease the power consumed by the system

Parallelization

- Hardware acceleration
- Software acceleration

Prerequisites

- CE311 : Processors architecture
- SC351 : Algorithmic and C programming

Flynn's taxonomy

- Single Instruction Single Data (SISD)
  A single instruction is executed at a time and on a single data flow
- Single Instruction Multiple Data (SIMD)
  The same instruction is executed on several data flows
- Multiple Instruction Single Data (MISD)
  A single data flow is sent to different compute units
- Multiple Instruction Multiple Data (MIMD)
  Each data flow is processed by a separate compute unit

Why parallelization?

- Processors' size: 32 bits and 64 bits
- Audio (ADC): samples coded with 12 bits (or 16 bits)
- Image (RGB pixel): samples coded with 8 bits

Results:

- Processor performance is limited during data processing
- Increased consumption
- High latency

Solutions:

DSP/FPGA

It is possible to realize some processing on a separate processing unit (DSP/FPGA) to decrease the load on the main processor

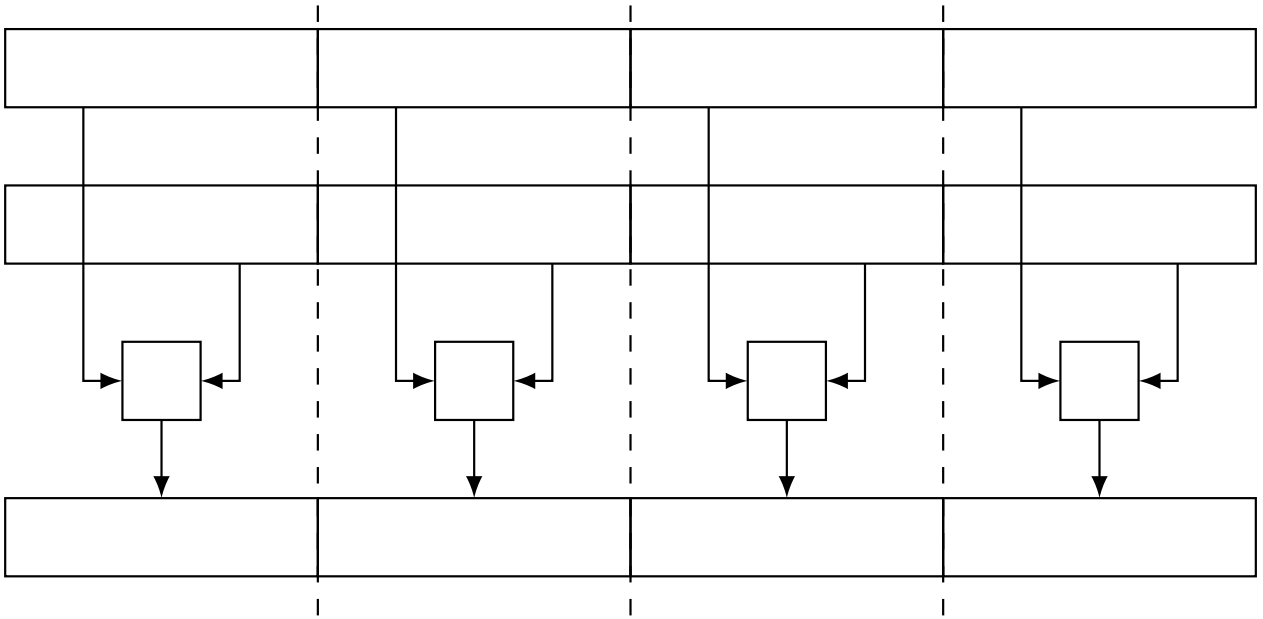
- Dada transfer management
- Hardware modification
- Increased complexity

SIMD

Process the same instruction on several data...

SIMD

- Registers are vectors that can hold several identical elements
- The same instruction is executed on all the elements



SIMD Instruction set

Most of modern processor family implement a SIMD instruction set:

- Intel x86: MMX/SSE/AVX
- PowerPC: Altivec
- ARM: NEON (Advanced SIMD)

Avantages

- No hardware modification
- Easy development
- Limited acceleration (compared to DSP/FPGA)

ARM Presentation

ARM processors dominate the embedded market:

- mobile phones
- tablets
- video game consoles
- cameras

Low Power Processors
Processors available inside System on Chip (SoC)
ARM architecture is licensed to other companies

ARM Architectures and cores

Architecture	Cores
ARMv1	ARM1
ARMv2	ARM2
ARMv2aS	ARM3 ARM 250
ARMv3	ARM6, ARM7, Amulet1
ARMv4	ARM8, StrongARM, ARM9, ARM9TDMI, ARM940T
ARMv5	ARM10, Intel XScale
ARMv6	ARM11
ARMv6-M	Cortex-M0, Cortex-M0+, Cortex-M1
ARMv7-A	Cortex-A8, Cortex-A9 MPCore, Cortex-A5 MPCore
	Cortex-A7 MPCore, Cortex-A12 MPCore, Cortex-A15 MP
ARMv7-M	Cortex-M3, Cortex-M4 et Cortex-M7
ARMv7-R	Cortex-R4, Cortex-R5, Cortex-R7
ARMv8	Cortex-A53, Cortex-A57

Instruction Set

ARM instructions classical instructions (coded over 32 bits, processed by the ARM core).

Thumb 16 bits-instructions providing a subset of the ARM instructions (ARMv4).

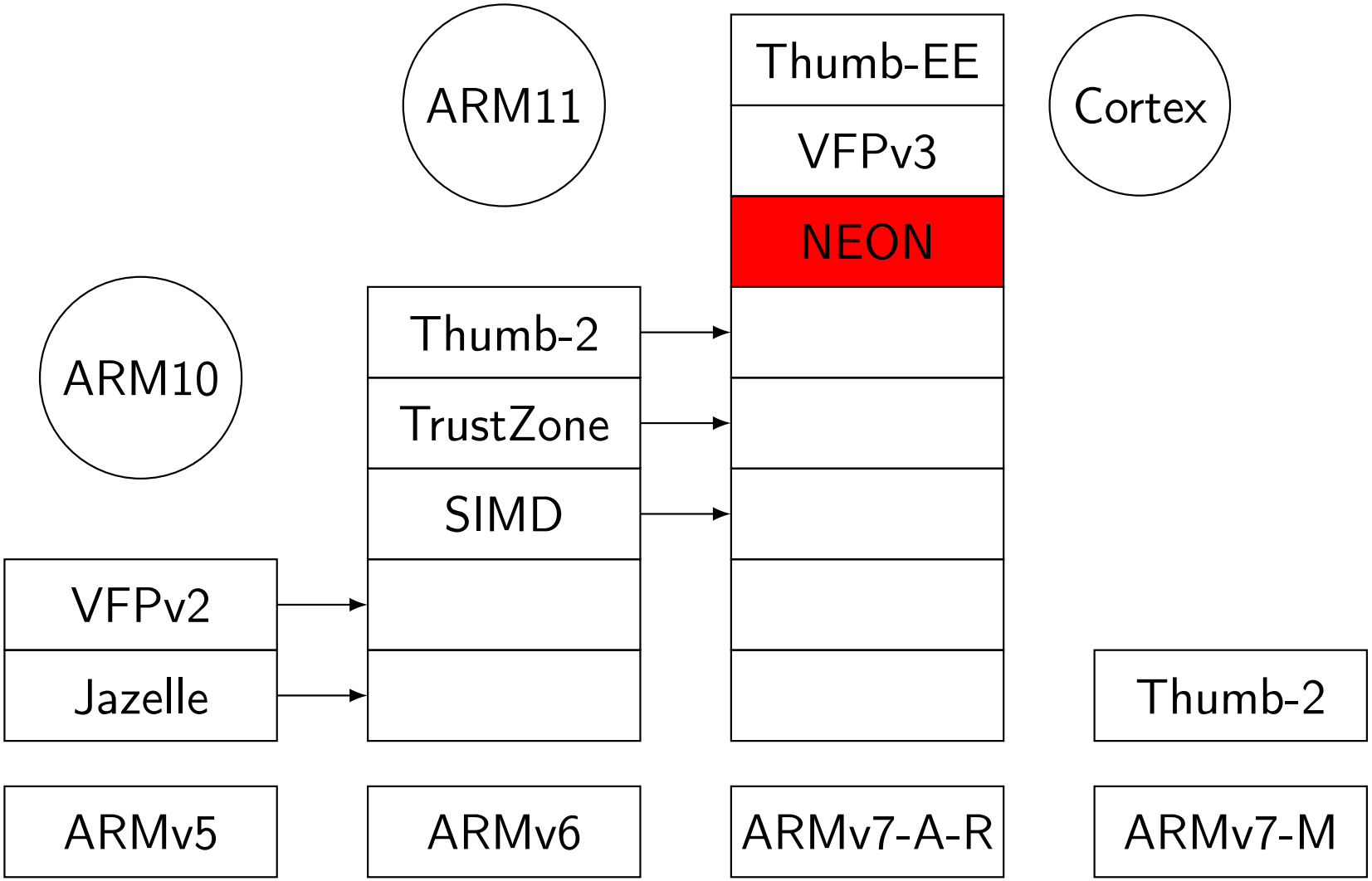
Jazelle optimized instructions to process Java bytecode. This extension has been added since ARMv5.

Vector Floating Point instructions for 32 and 64 bit floating point format (ARMv5)

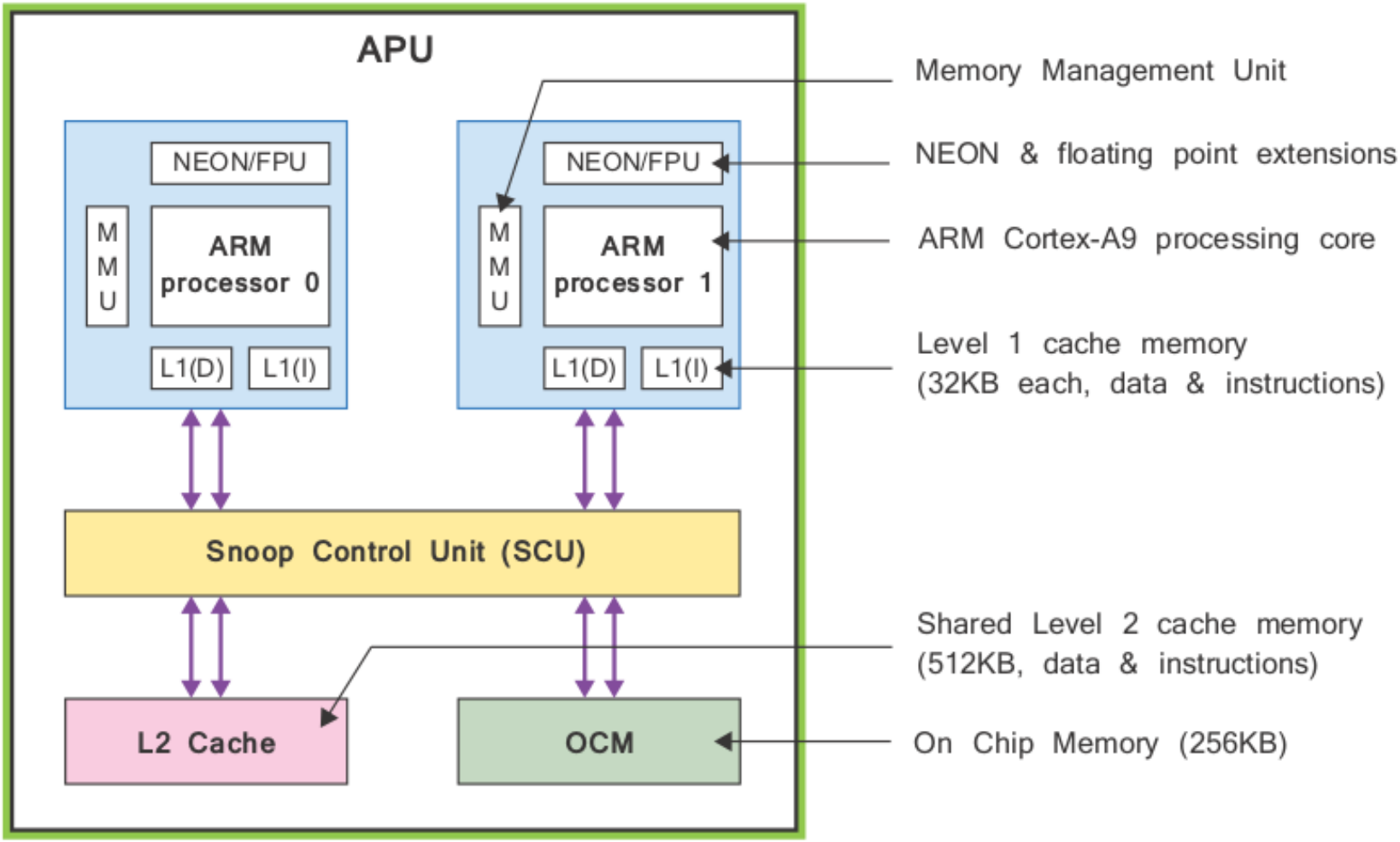
Thumb-2 instructions coded on 16 and 32 bits

Thumb-EE optimized instructions for dynamically generated code (since ARMv7)

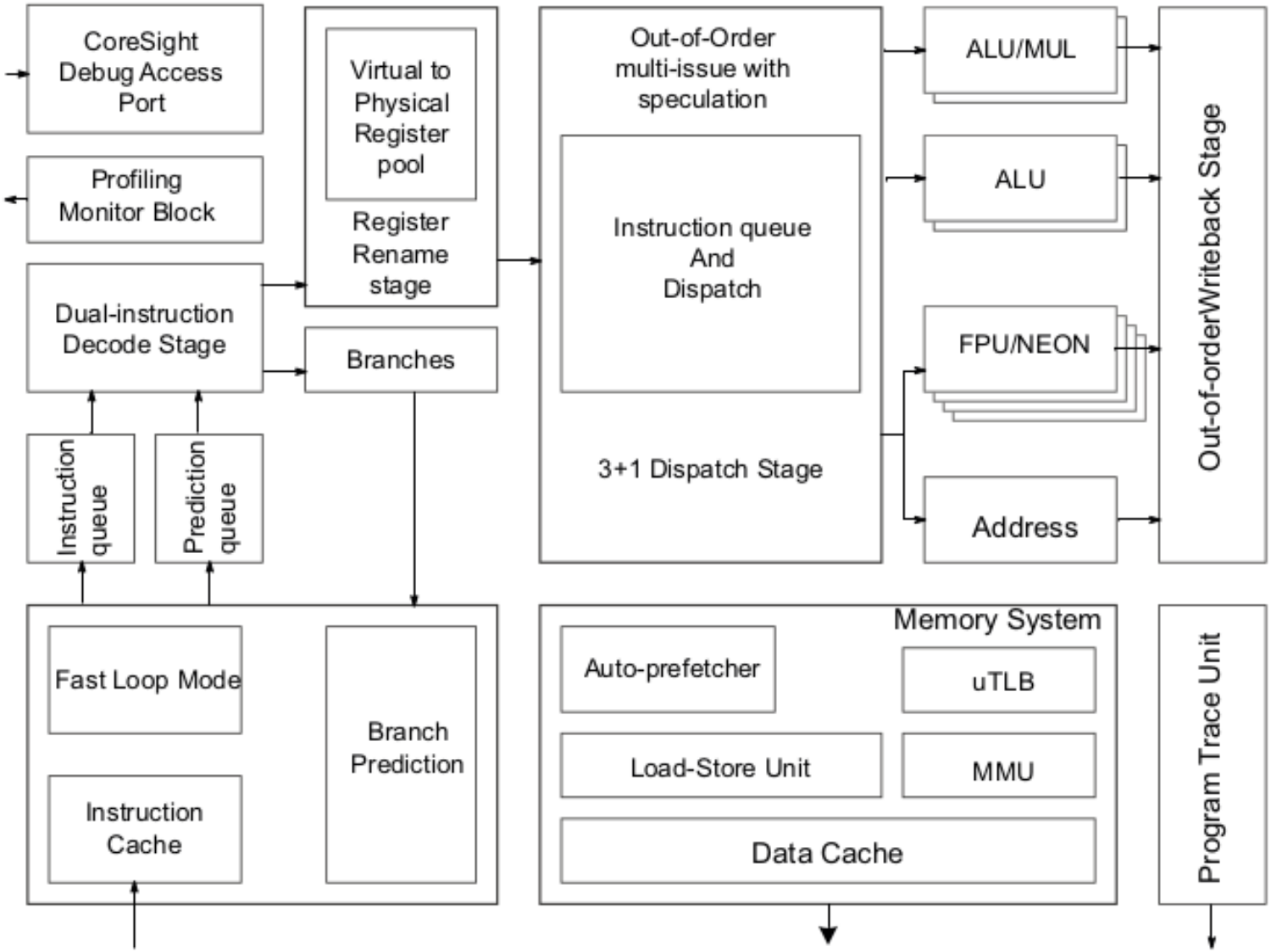
Adv. SIMD (NEON) (to be continued...)


ARM11
Thumb-EE
VFPv3
NEON
Cortex
ARM10
Thumb-2
TrustZone
SIMD
VFPv2
Jazelle
Thumb-2
ARMv5
ARMv6
ARMv7-A-R
ARMv7-M

Zync 7000 Architecture



Cortex A9 Architecture



Architecture ARM

- 16 general registers: R0 to R15
- 2 status registers: CPSR (always visible) et SPSR (copy of CPSR in exception modes) for flags, bit control
- Only `load` et `store` can memory
- limited addressing modes
- fixed length instructions (ARM instruction: 32 bits)
- 3 operands per instructions (`add r3, r2, r1` $\rightarrow$ r3 = r2 + r1)

Calling conventions

- R14: link register (lr)
- R15 program counter (pc)
- R13: stack pointer (sp)
- R11: frame pointer (fp)
- R0-R3: hold the parameters of a function
- R0 holds the returned value of a function

Examples

C Language

```
int add(int, int)
{
  return a + b;
}

int main()
{
  int a = 1, b = 2;
  int c;
  c = add(a, b);
  return 0;
}
```

ARM Assembly

```
add: str fp, [sp, #-4]!
     add fp, sp, #0
     sub sp, sp, #12
     str r0, [fp, #-8]
     str r1, [fp, #-12]
     ldr r2, [fp, #-8]
     ldr r3, [fp, #-12]
     add r3, r2, r3
     mov r0, r3
     sub sp, fp, #0
     @ sp needed
     ldr fp, [sp], #4
     bx lr
```

C Language

```
int acc = 2;
int i;

for (i = 0; i < 10; i++)
  acc = acc * acc
```

ARM Assembly

```
        b .L4
.L5:
        ldr r3, [fp, #-12]
        ldr r2, [fp, #-12]
        mul r3, r2, r3
        str r3, [fp, #-12]
        ldr r3, [fp, #-8]
        add r3, r3, #1
        str r3, [fp, #-8]
.L4:
        ldr r3, [fp, #-8]
        cmp r3, #9
        ble .L5
        ldr r3, [fp, #-12]
```

NEON

NEON

NEON is the implementation of the Advanced SIMD extension.
NEON can be included on ARMv7-A et ARMv7-R architectures.
Actually, this extension is included on all ARMv7-A architecture.

NEON can increase the execution speed of programs that realize the same process over a large quantity of data:

- Audio and signal processing
- Video codec
- Forward error correction, source coding
- Cryptography
- ...

NEON Registers

D0	Q0
D1	
D2	Q1
D3	
D4	Q2
D5	

...

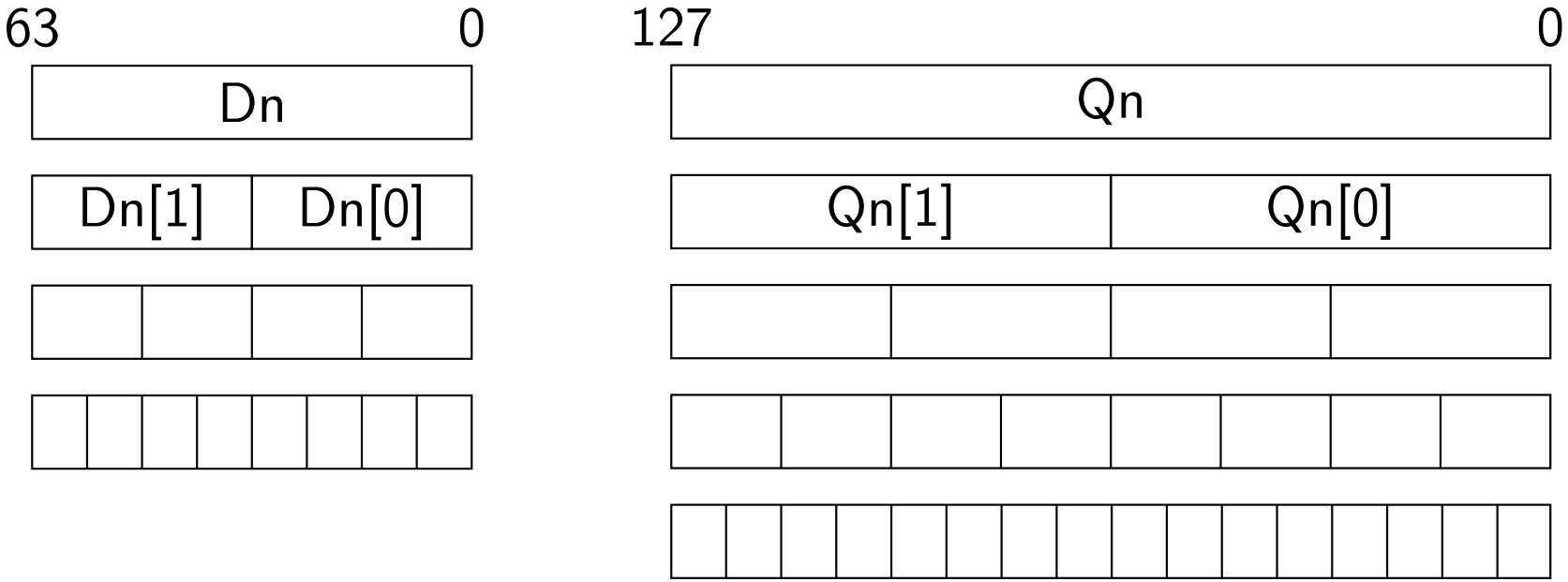
D30	Q15
D31	

- 32 of 64 bits: D0-31
- **or** 16 of 128 bits: Q0-16
- No flag for a line

NEON/VFP

Vector Floting Point (VFP) extension provides instructions to process data in floating point format (single precision and double precision).
Processors that can support the NEON extension, support also the VFP extension.
Extensions NEON and VFPv3 share the same hardware resources (the same registers). Usually, we will avoid to mix NEON instructions and VFP instructions since this imply to store and load the registers values in the stack.

NEON registers structure



Instruction syntax

V{mod}op{shape}{cond}{.dt} dest1, src1, src2

- dest1: Destination register D{0-31}, Q{0-16}
- src1, src2 Source register(s) D{0-31}, Q{0-16}
- mod: Q, H, D, R
- shape: L, N, W
- cond: Condition
- op: Operation (ex ADD, MULL)
- .dt: data type

mod (modifier)

mod can change the result stored in the destination register

- Q (Saturation): result is saturated in case of overflow, the value depends on the operand type and sign
- H (Halved): result is divided by 2
- D (Doubled): result is multiplied by 2
- R (Rounded): result is rounded (integer)

If mod is left blank, result is directly stored inside the destination register (without modification).

shape

By default, result and operands have the same data type. Shape can change the data type of the results

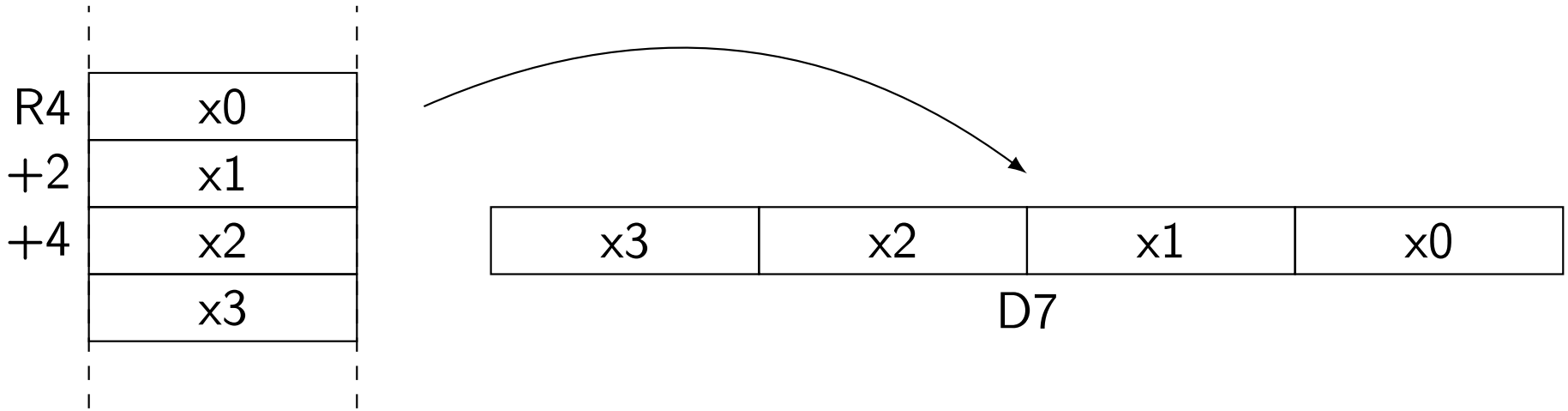
- L (Long) result's size is twice the operands' size
- N (Narrow) result's size is half the operands' size
- W (Wide) result's size and 1st operand size is twice the second operand's size

.dt (data type)

<table>
<tr><td rowspan="2">.8</td><td rowspan="2">.I8</td><td>.U8</td></tr>
<tr><td>.S8</td></tr>
<tr><td colspan="2">.P8</td></tr>
<tr><td rowspan="2">.16</td><td rowspan="2">.I16</td><td>.U16</td></tr>
<tr><td>.S16</td></tr>
<tr><td colspan="2">.P16</td></tr>
<tr><td rowspan="2">.32</td><td rowspan="2">.I32</td><td>.U32</td></tr>
<tr><td>.S32</td></tr>
<tr><td colspan="2">.F32</td></tr>
<tr><td rowspan="2">.64</td><td rowspan="2">.I64</td><td>.S64</td></tr>
<tr><td>.U64</td></tr>
</table>

VLD1: Read from memory

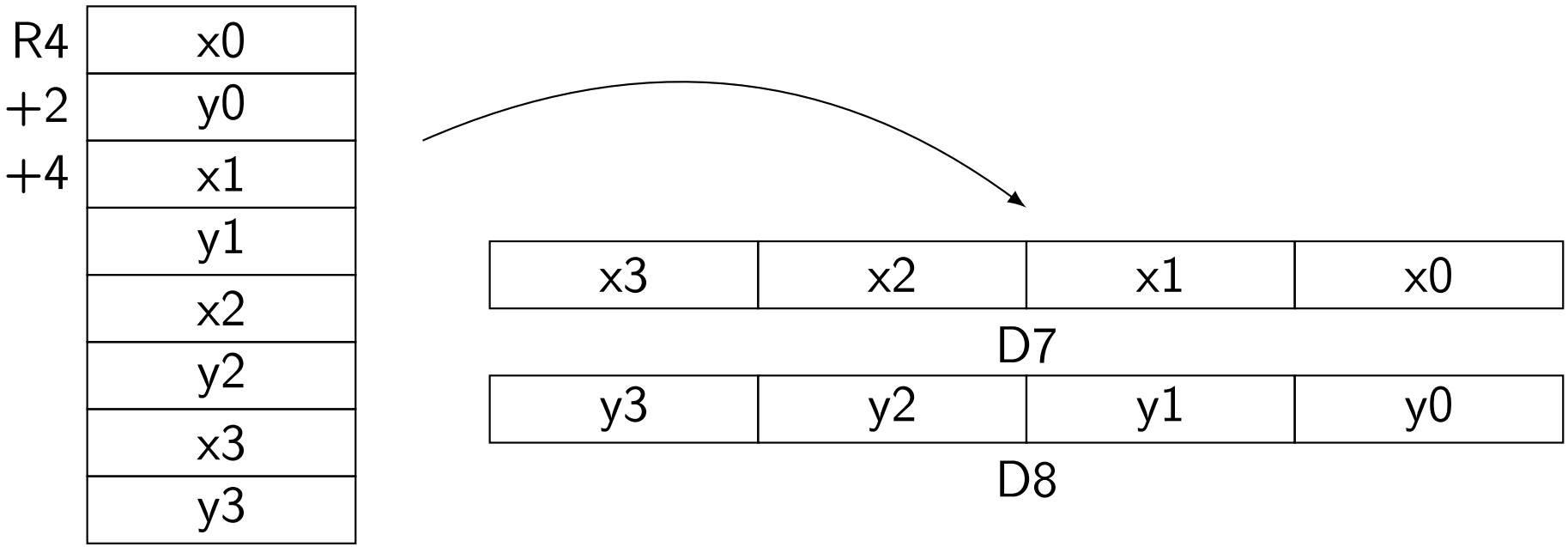
Read multiple elements from the memory to 1 or several NEON registers



VLD1.16 {D7} [R4]!
VLD1.16 {D7,D8} [R4@128] R3

VLD2: Read and de-interleave from memory

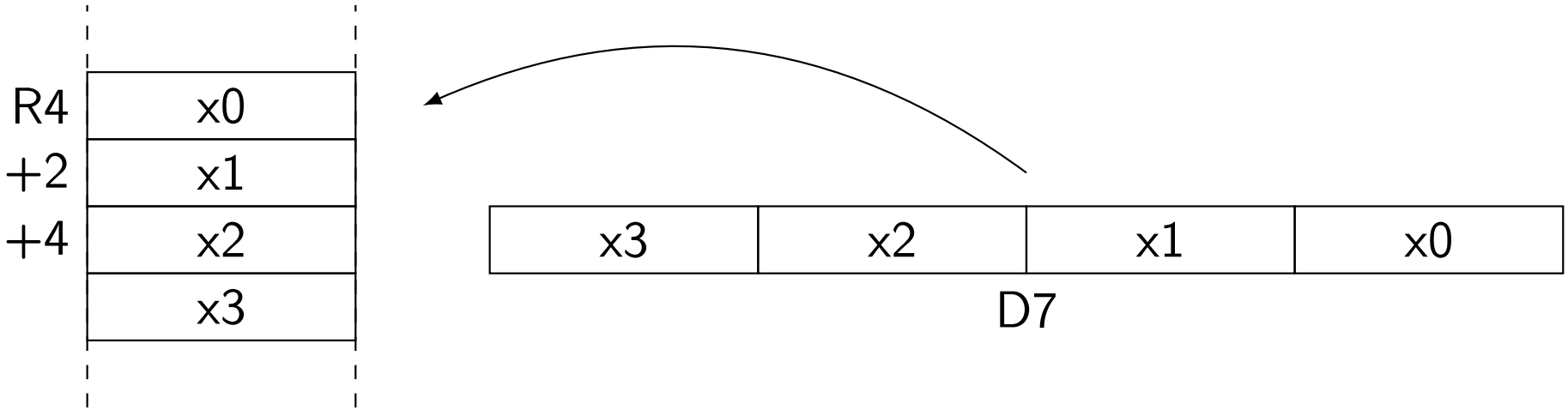
Read multiple elements from the memory with de-interleaving to 1 or several NEON registers



VLD2.16 {D7,D8} [R4]!

VST1: Write to memory

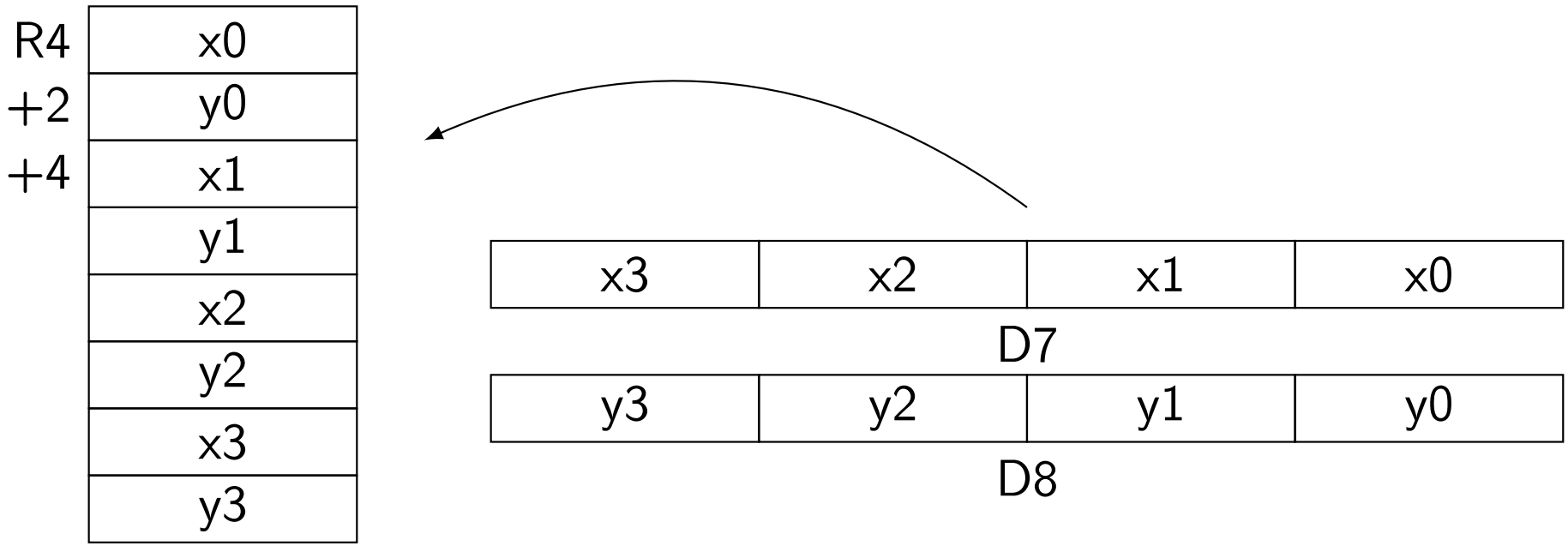
Write multiple elements from registers to the memory.



VST1.16 {D7} [R4]!
VST1.16 {D7,D8} [R4@128] R3

Write and interleave to memory

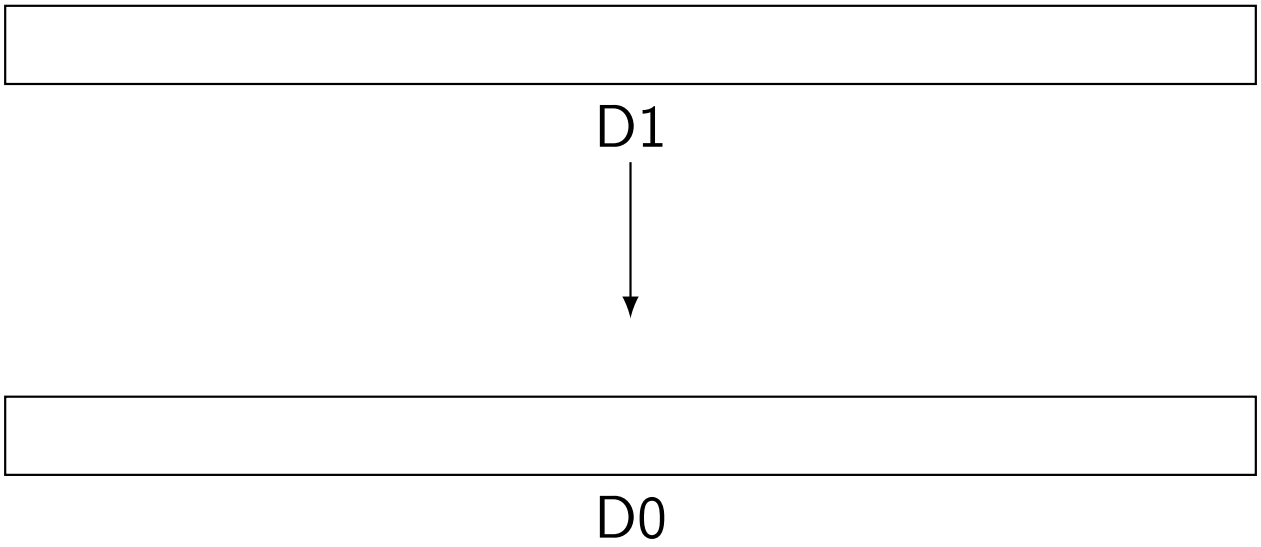
Write and interleave multiple elements from registers to the memory.



VST2.16 {D7,D8} [R4]!

VMOV: Move between registers

Copy the content of 1 NEON register to another NEON register.



VMOV D0 D1

Source register can also be replaced by an immediate value.

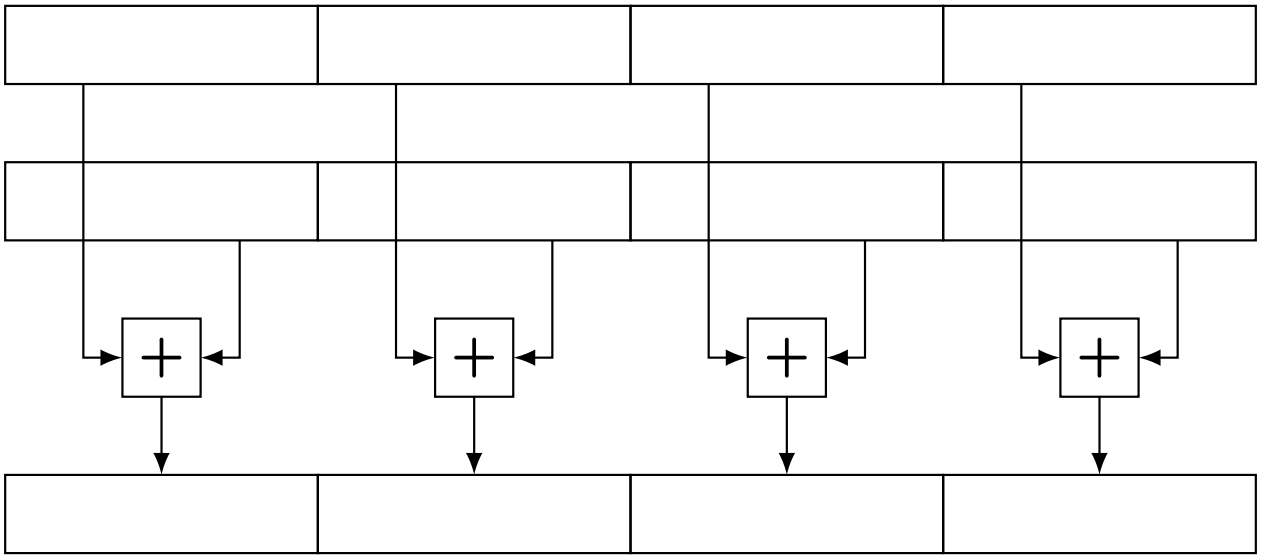
VMOV

VMOV is not limited to the transfer between NEON registers;

- VMOV d0, r0, r1 ; transfer ARM $\rightarrow$ NEON
- VMOV r0, r1, d0 ; transfer NEON $\rightarrow$ ARM
- VMOV.<8,16,32> d0[0], r0 ; single element transfer
- VMOV.<S8,S16,U8,U16,32> r0, d0[0] ; idem
- VDUP.<8,16,32> d0, d1[0] ; Broadcast
- VDUP.<8,16,32> d0, r0 ; idem

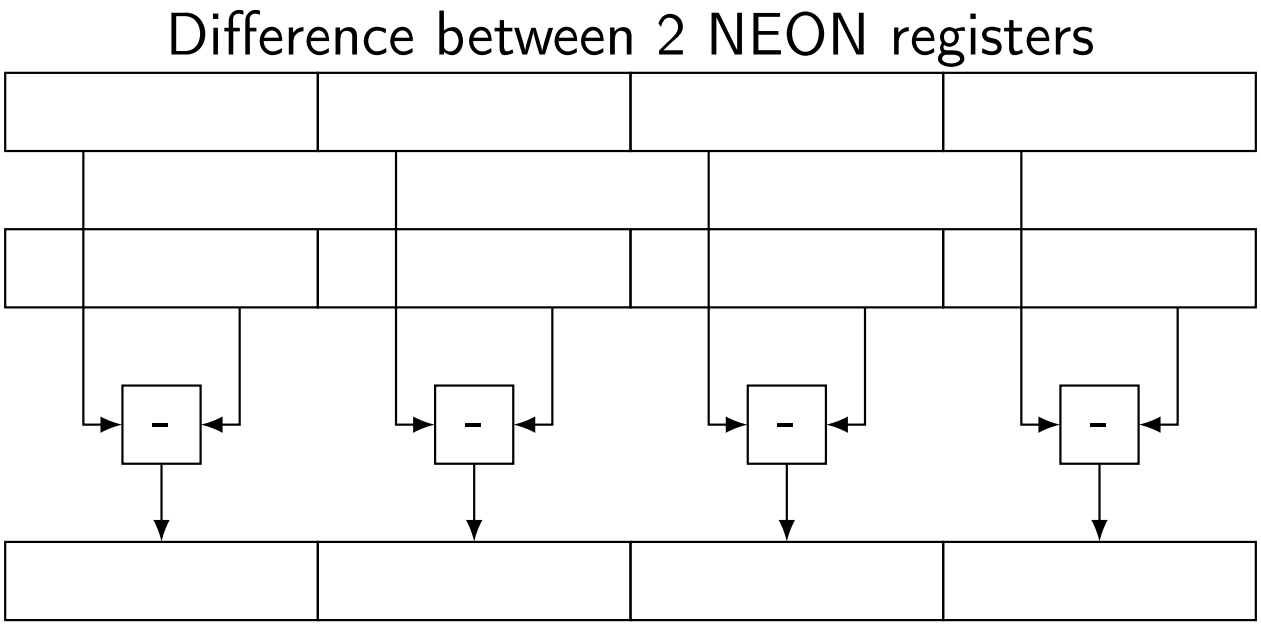
VADD: Addition

Addition between 2 NEON registers



VADD.I16 D0 D1 D2
VADDQ.F32 Q0 Q0 Q1

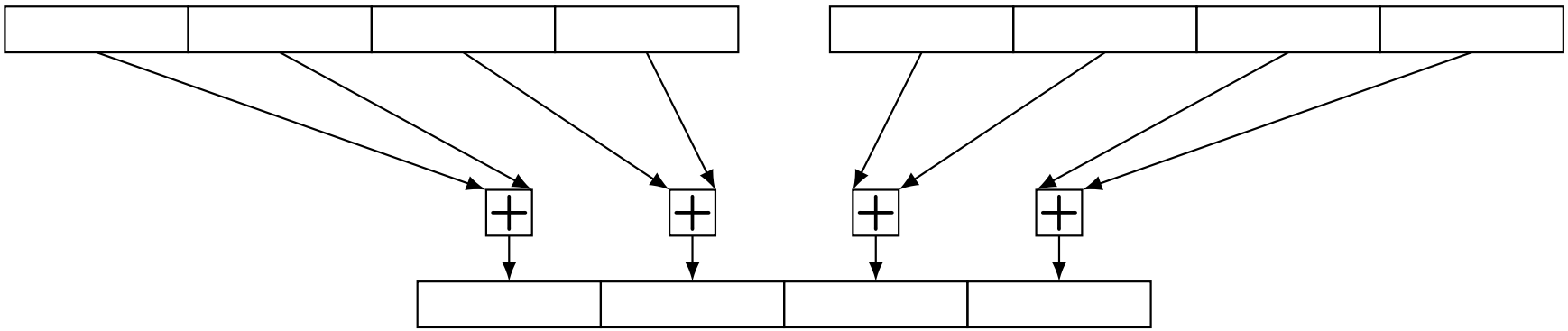
VSUB: Substraction



VSUB.I64 D0 D30 D5
VSUBL.I8 Q0 D7 D8

VPADD: Pairwise Addition

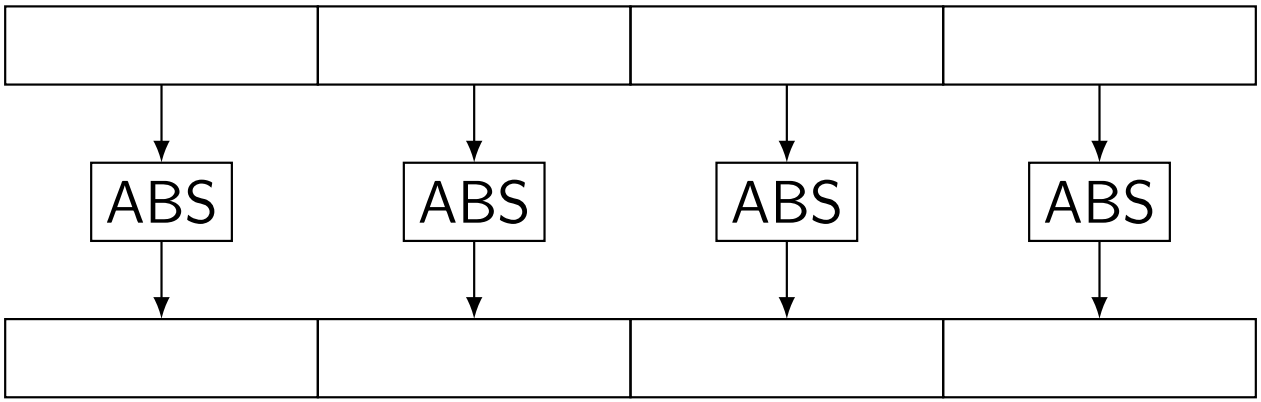
Pairwise addition between consecutive elements inside 1 or 2 registers



VPADD.I16 D0 D1 D2
VPADDL.I16 D0 D0
VPSUB.I8 D0 D4 D5

VABS: Absolute Value

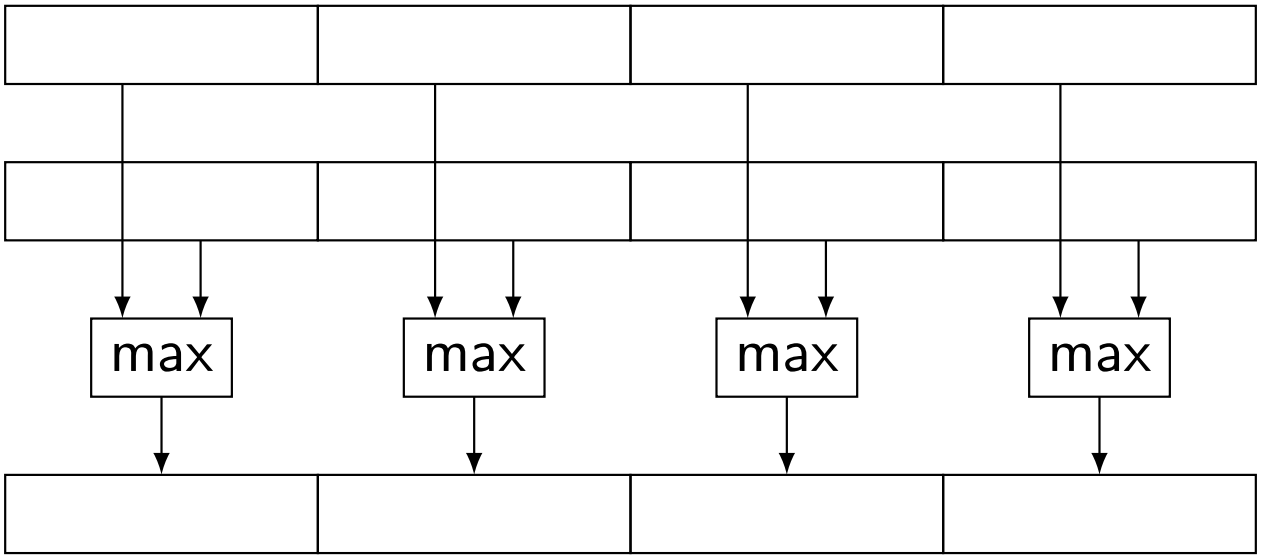
Absolute value of 1 NEON register



VABS.I64 D0 D30
VQABS.I8 Q0 Q0

VMIN and VMAX: Minimum and Maximun

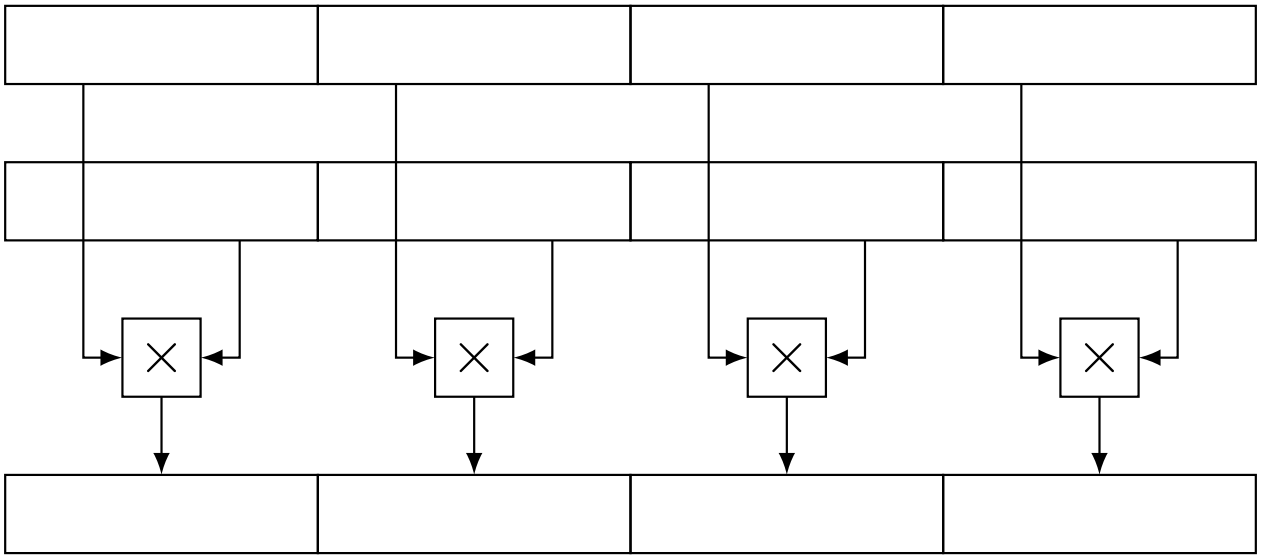
Compare elements from 2 vectors and keep the maximum (or the minimum)



VMAX.S32 D0 D30 D5
VMIN.U16 Q0 D7 D8
VPMAX and VPMIN instructions are also available.

VMULL: Multiplication

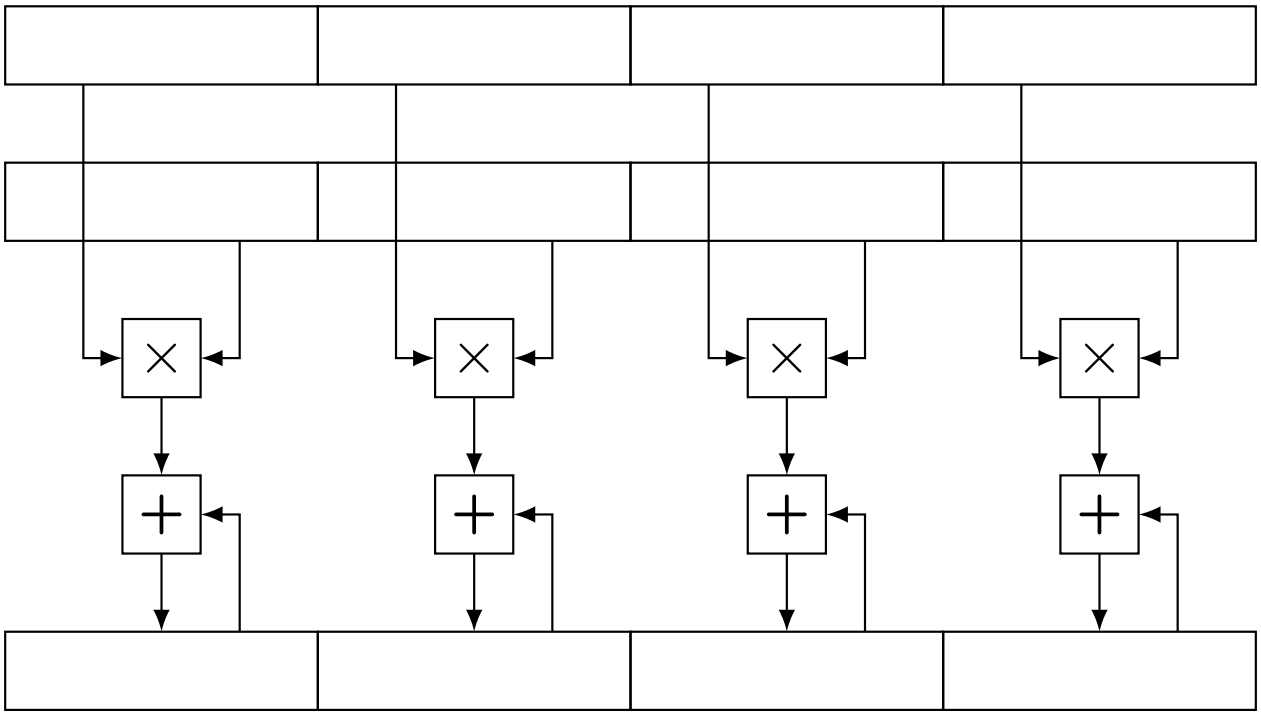
Multiplication between 2 NEON registers



VMUL.I16 D0 D1 D2
VMUL.F32 Q0 Q0 Q1

VLMA: Multiplication et accumulation

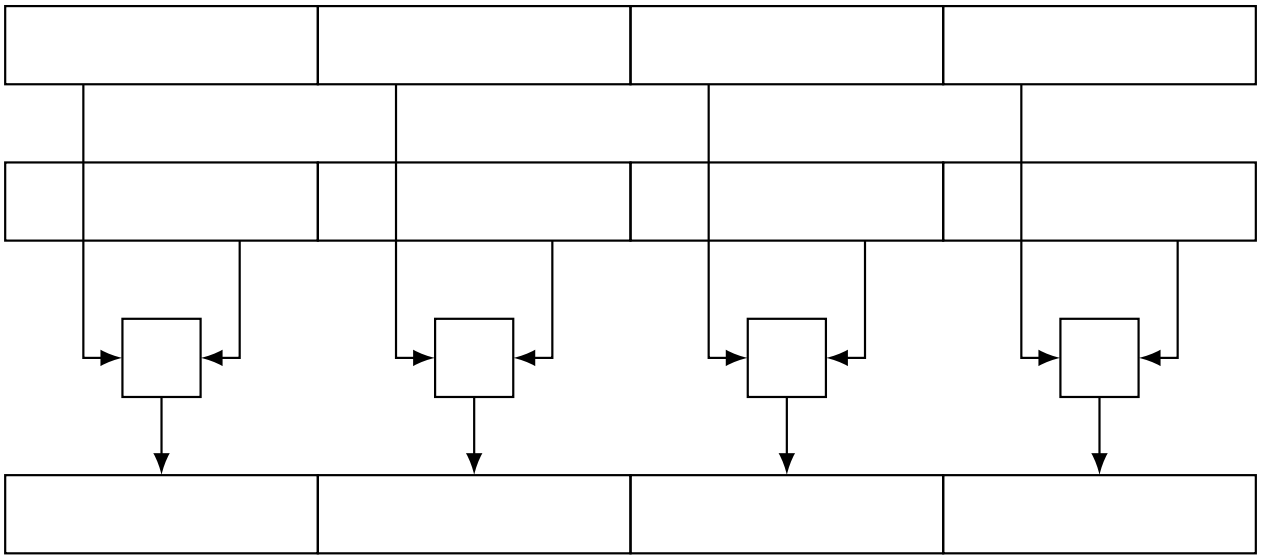
Multiply and accumulate between 3 NEON registers



VMLA.I16 D0 D1 D2
VMLS.F32 Q0 Q0 Q1

VAND, VEOR...: Logical operations

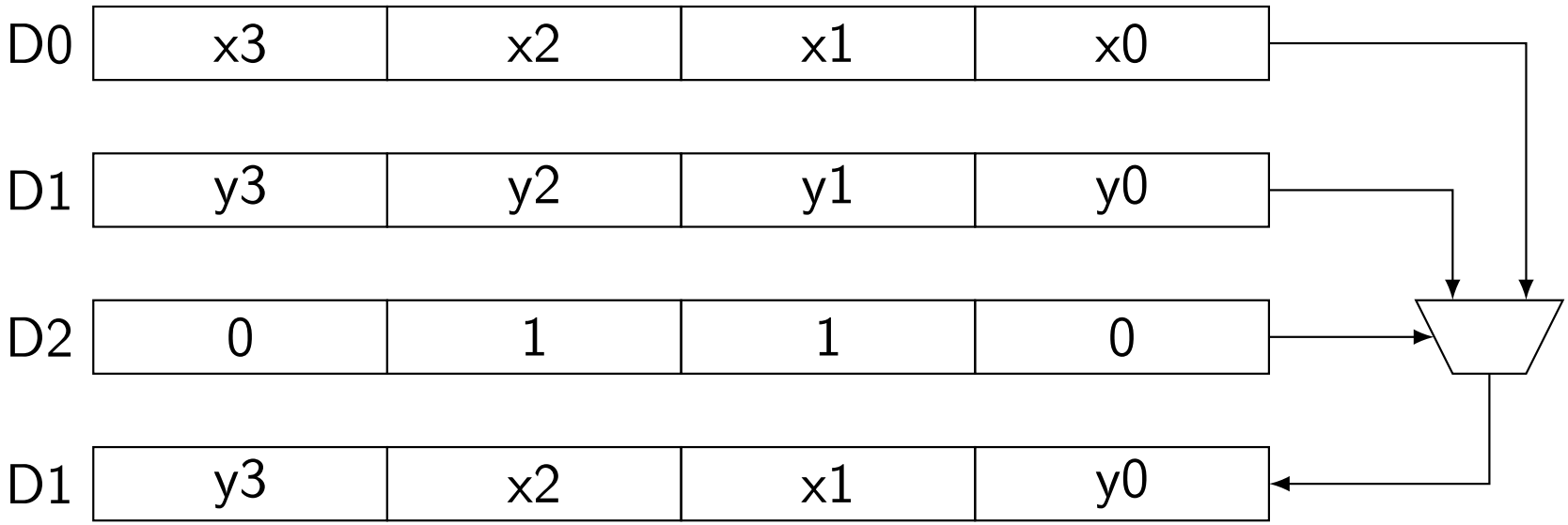
Logical operation (AND OR XOR NOT) between 2 NEON registers



```
VAND.16 D0 D1 D2
VORR Q0 Q0 Q1
VEOR.8 D0 D0 D0
VMVM D0 D0
```

VBIT, VBIF, VBSL: Multiplexing

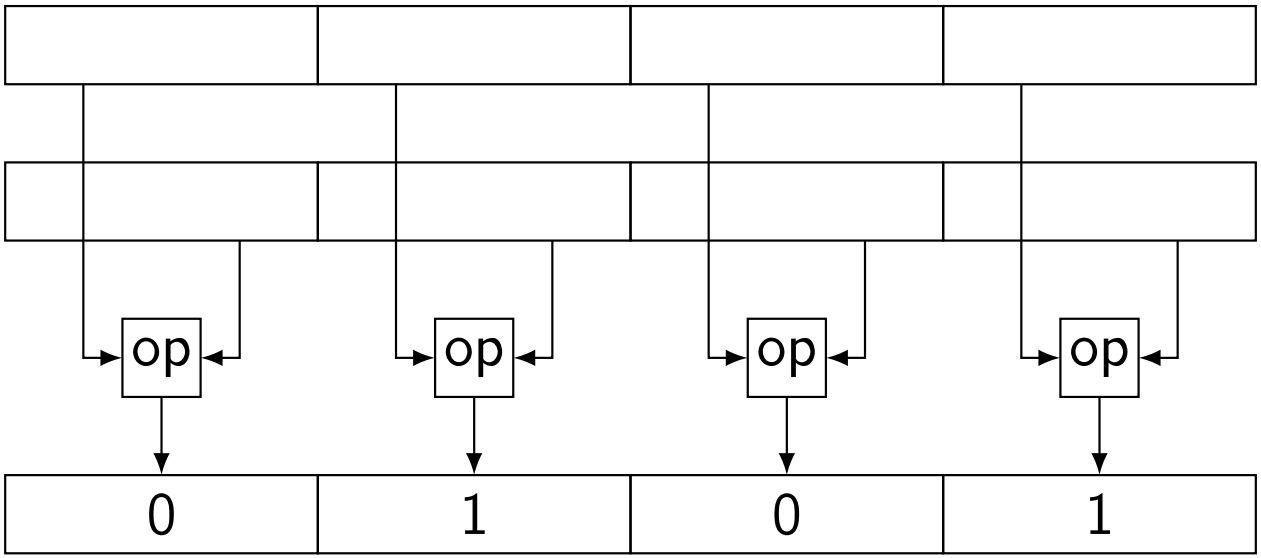
VBIT (insert if true), VBIF (insert if false), VBSL (destination)



VBIT D1, D0, D2

VEQ, VGE...: Comparison

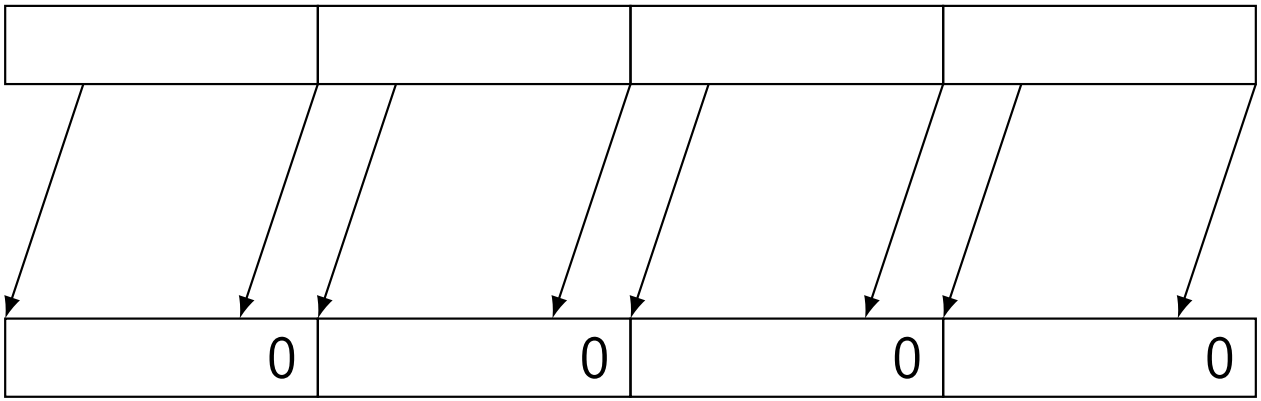
Compare 2 vectors (EQ GE GT LE LT), result is stored in the destination.



VGE D1, D0, D2 VEQ D0, #0

VSLI, VSRI: Shift (constant)

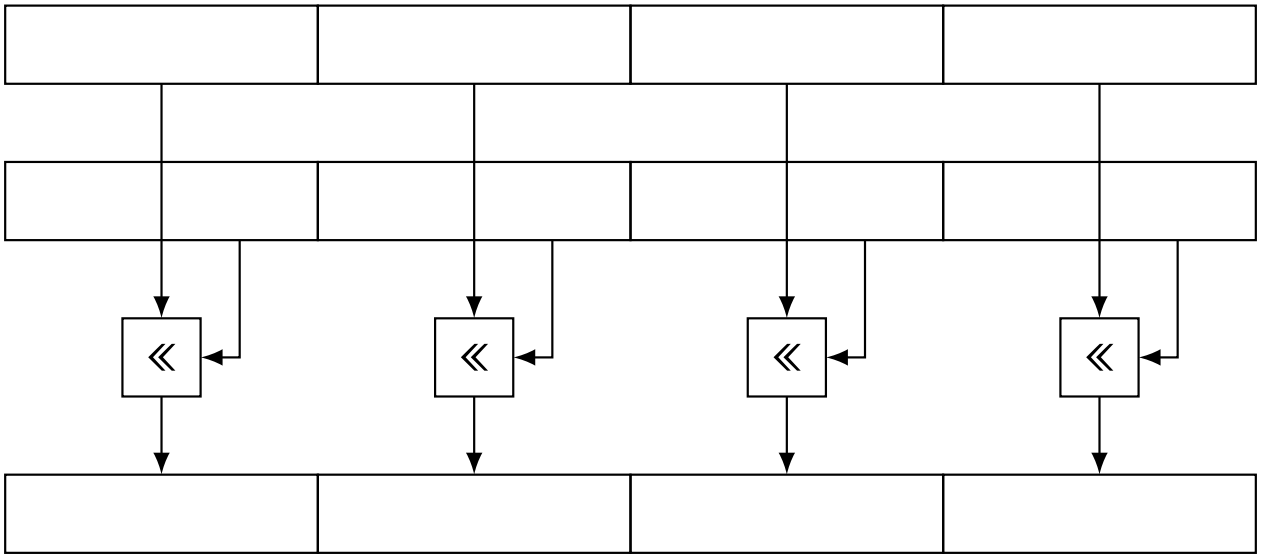
Shift left or right by a constant offset (immediate value).



VSLI.8 D1, D0, #3
VSRI.16 D0, #1

VSHL: Shift (variable)

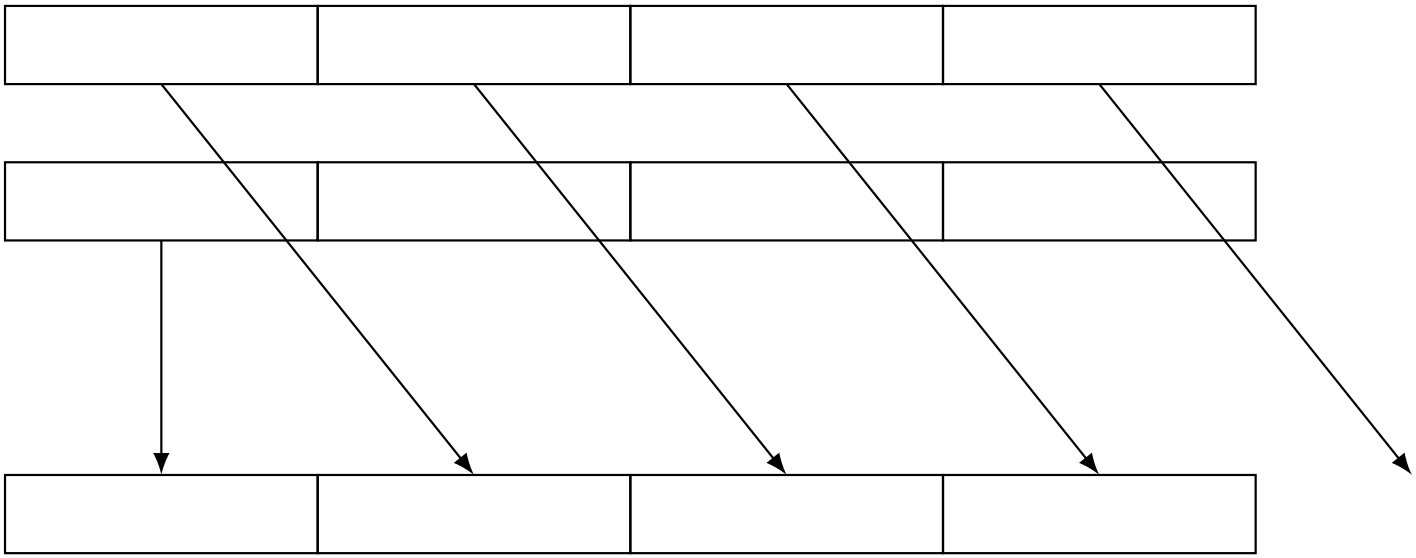
Shift left by the (signed) content of the second operand. If the offset is negative, shift right.



VSHL.16 D0 D1

VEXT

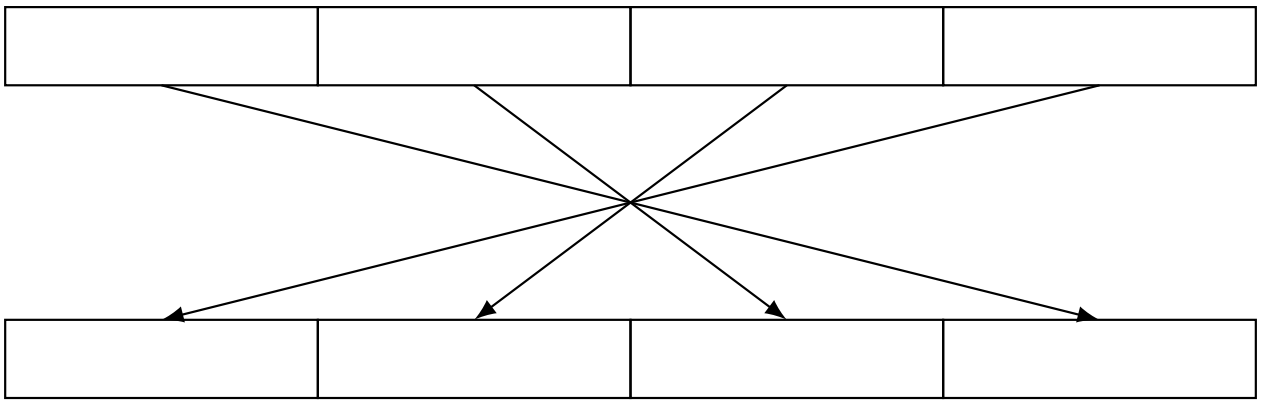
VEXT extract the n first elements from vector 2 and the other from vector 1 and place the result in the destination (circular shift if V1 = V2).



VEXT.16 d0, d0, d1, #1

VREV64, VREV32, VREV16

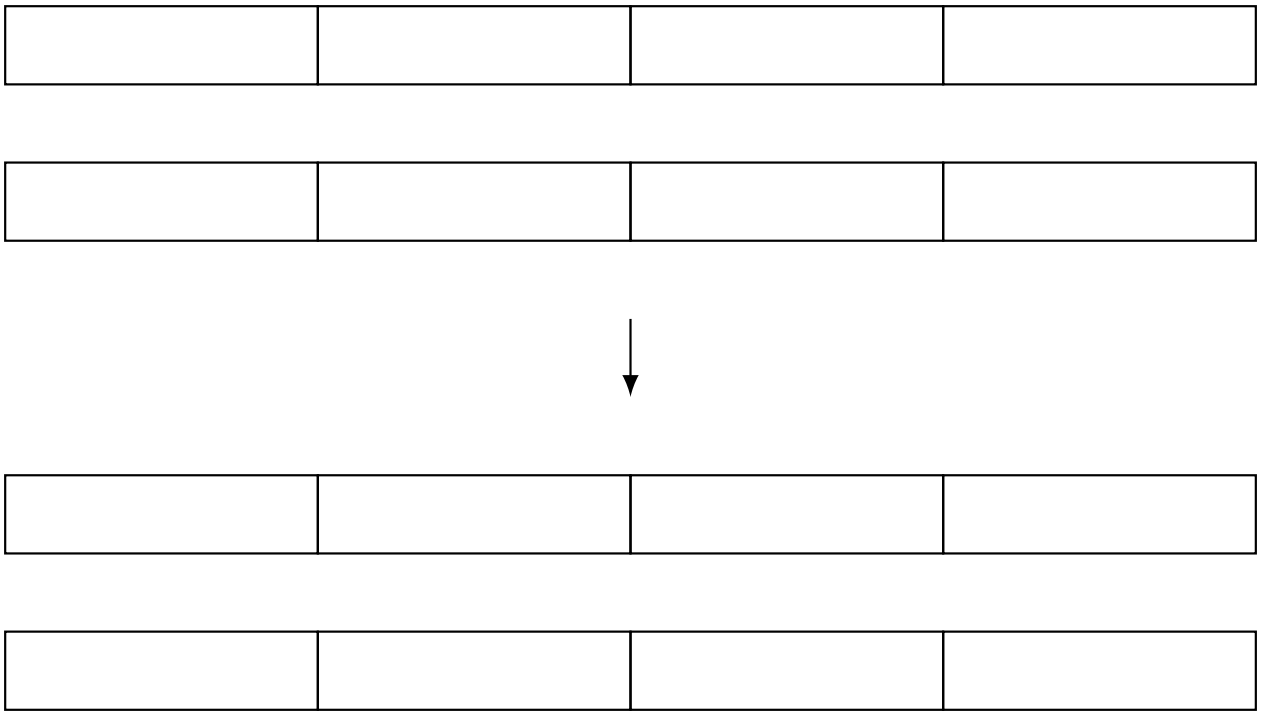
VREV inverse the order of each element in source vector and store the result in the destination vector.



VREV64.16 d0, d0

VSWP

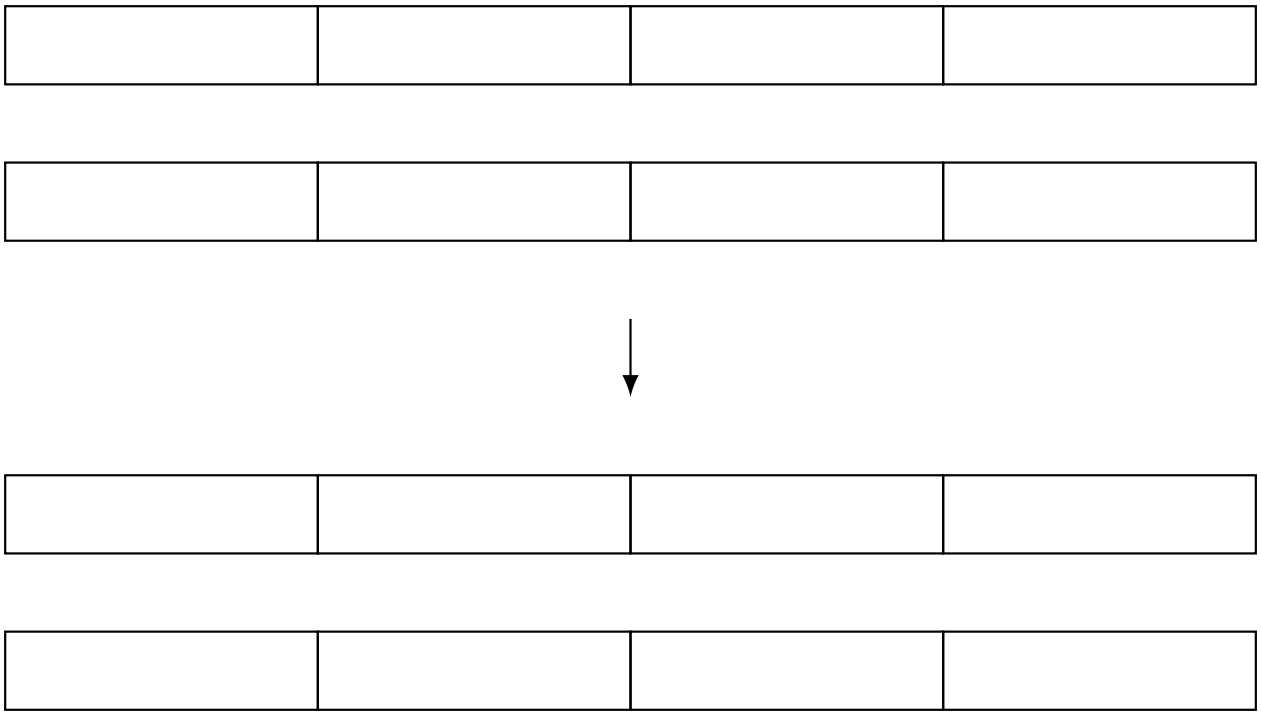
VSWP swap the content of 2 vectors.



VSWP.16 d0, d1

VTRN

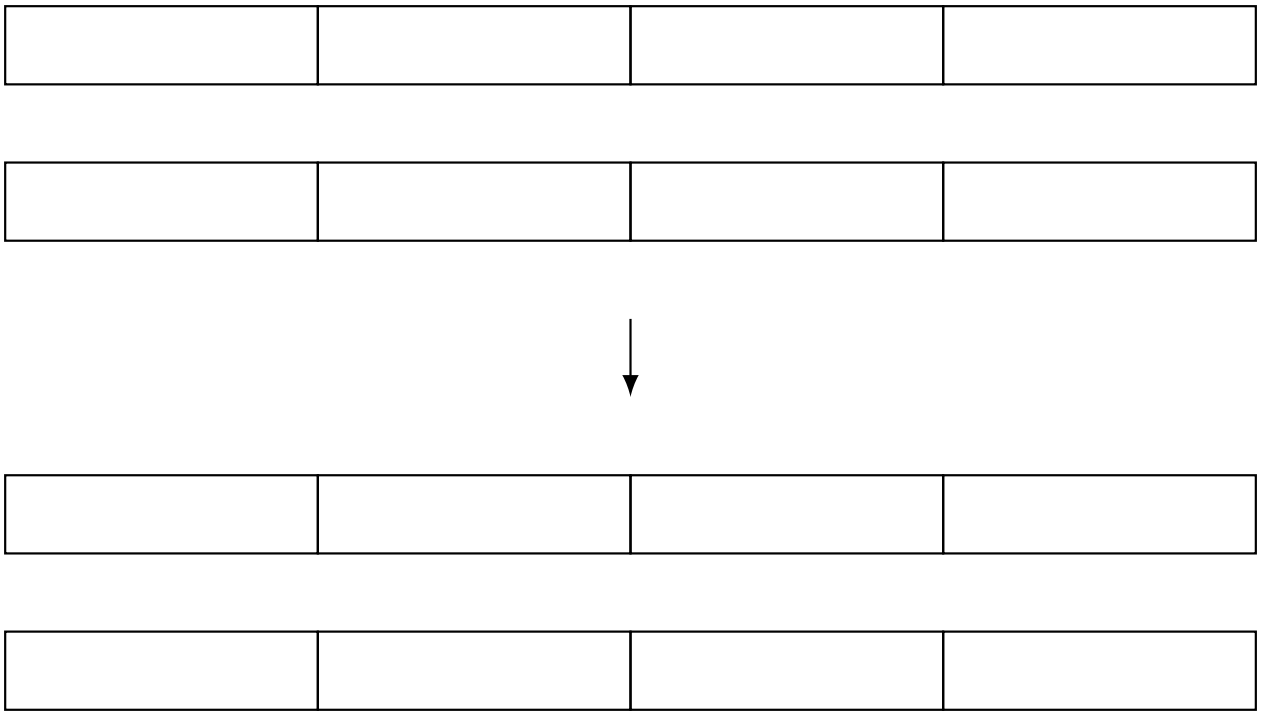
VTRN process the elements as 2x2 matrix and realize transposition.



VTRN.16 d0, d1

VZIP

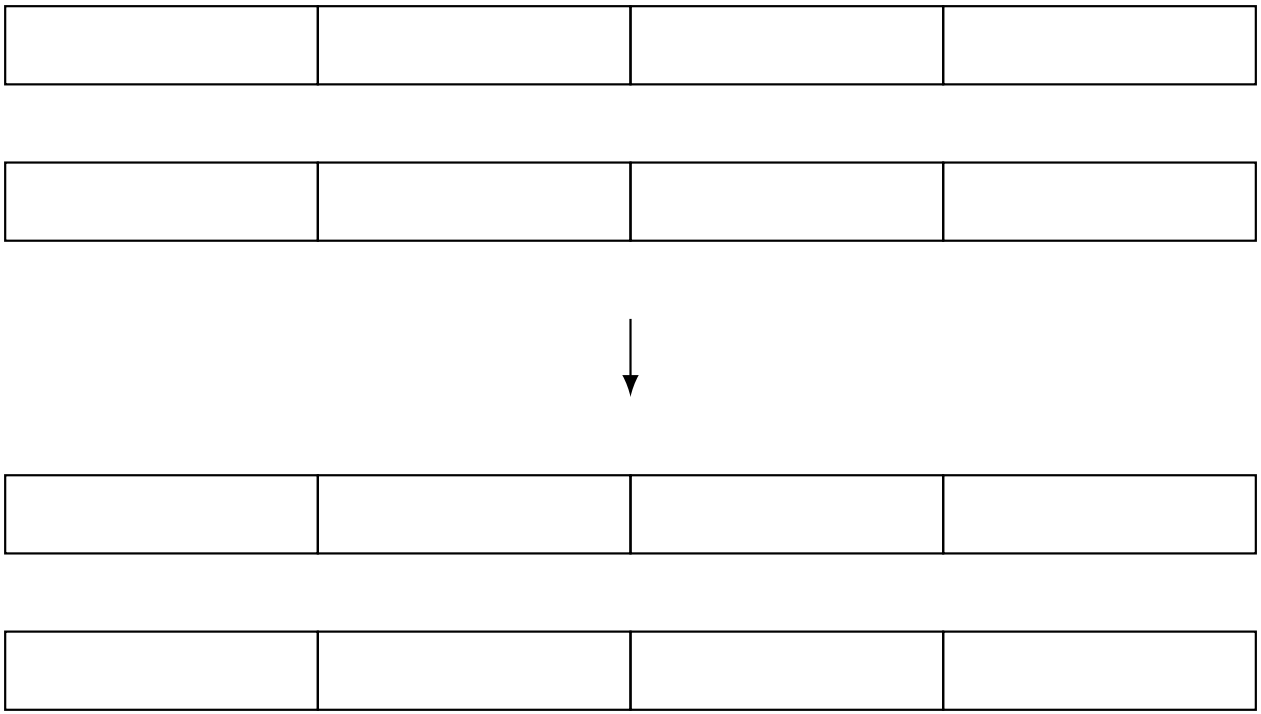
VZIP interleave the elements of 2 vectors.



VZIP.16 d0, d1

VUZP

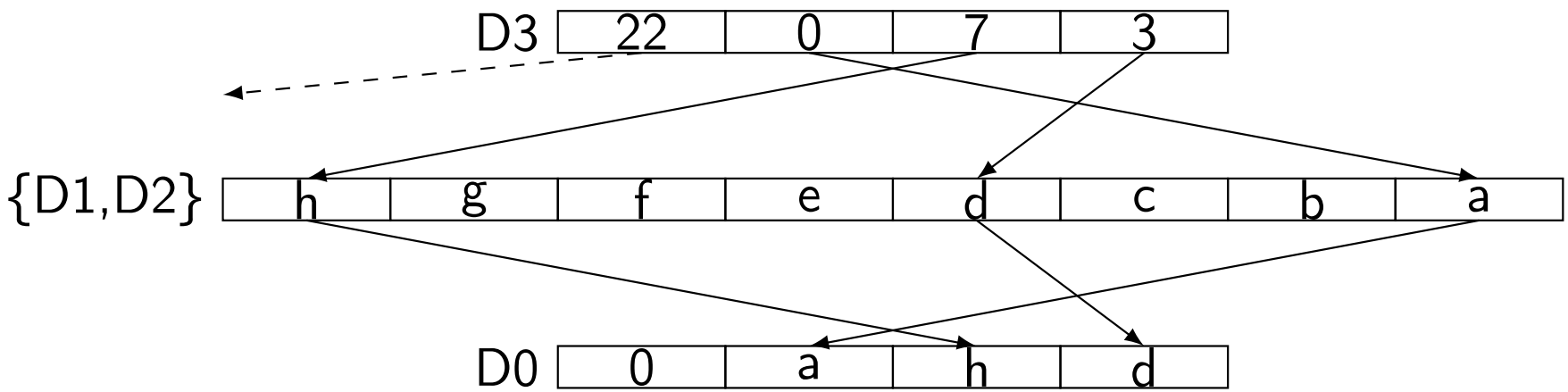
VUZP de-interleave the elements of 2 vectors.



VUZP.16 d0, d1

VTBL: Lookup table

Evaluate a complex function by using a pre-stored lookup table.
Elements outside the index range are set to 0.



VTBL.16 D0, D1, D2, D3

Activation of the NEON engine

By default, NEON is not activated.

```
__asm void EnableNEON(void)
{
  MRC p15,0,r0,c1,c0,2 // Read CP Access register
  ORR r0,r0,#0x00f00000 // Enable full access to NEON/VFP by
  // Coprocessors 10 and 11
  MCR p15,0,r0,c1,c0,2 // Write CP Access register
  ISB
  MOV r0,#0x40000000 // Switch on the VFP and NEON hardware
  MSR FPEXC,r0 // Set EN bit in FPEXC
}
```

Timing

Name	Format	Cycles	Source	Result	Writeback
VADD	Dd,Dn,Dm	1	-,2,2	3	6
VAND	Qd,Qn,Qm				
VORR					
VEOR					
VBIC					
VORN					
VSUB	Dd,Dn,Dm	1	-,2,1	3	6
	Qd,Qn,Qm				
VADDL	Qd,Dn,Dm	1	-,1,1	3	6
VADDW	Qd,Qn,Dm	1	-,2,1	3	6
VSUBW					
VHADD	Dd,Dn,Dm	1	-,2,2	4	6

Name	Format	Cycles	Source	Result	Writeback
VRHADD	Qd,Qn,Qm				6
VQADD					
VTST					
VADH	Dd,Qn,Qm	1	-,2,2	4	6
VRADH					
VSBH	Dd,Qn,Qm	1	-,2,1	4	6
VRSBH					
VMUL	.8 Dd,Dn,Dm	1	-,2,2	6	6
VMLA	.8 Dd,Dn,Dm	1	3,2,2	6	6
VMULL	.16 Qd,Dn,Dm[x]	1-,2,1	6	6	
VLD1	Dd,[]	2	-	2	7
VST1	Dd,[]	2	1	-	-

Intrinsics

NEON intrinsics are a set of function (macros) and data types that can permit using NEON instructions directly from a C source file.

- available for `armcc` et `gcc`
- strictly identical API
- compiler handle register allocation, stack, instruction scheduling

The generated instructions are sometimes different from those inside the C file and data type conversions require additional instructions.

Intrinsics

Header file:

```
#include <arm_neon.h>
```

Compiler options have to be added:

- `-mcpu=cortex-a9`
- `-mfpu=neon`
- `-mfloat-abi=hard`

Data types:

Registre D (64 bits)	Registre Q (128 bits)
int8x8_t	int8x16_t
int16x4_t	int16x8_t
int32x2_t	int32x4_t
int64x1_t	int64x2_t
uint8x8_t	uint8x16_t
uint16x4_t	uint16x8_t
uint32x2_t	uint32x4_t
uint64x1_t	uint64x2_t
float16x4_t	float16x8_t
float32x2_t	float32x4_t
poly8x8_t	poly8x16_t
poly6x4_t	poly16x8_t

Some structures are also available for load and store functions

Each NEON instructions has an equivalent using the intrinsics:

NEON assembly	NEON macro
`vld1.16 {d18-d19}, [r1]!`	`va = vld1_s16(&a[i]);`
`vadd.i16 q8, q8, q9`	`vsum = vaddq_s16(vsum, va);`
`vsub.i16 d0, d0, d1`	`v1 = vsub_s16(v1, v2);`
`vmul.i16 q9, q9, q10`	`va = vmulq_s16(va, vb);`
`vst1.16 {d0}, [r2]!`	`vst1_s16(&b[i], v1);`

Macro NEON

During compilation, each macro is replaced by the correct instruction. There is no function call.

- Initialization (per lane):

```
uint8x8 start_value = vdup_n_u8(0);
```

- Initialization (per registre):

```
int8x8 start_value = vreinterpret_u8_u64(\\
vcreate_u64(0x123456789ABCDEFULL));
```

- Transfer vecteur to C variable:

```
result = vget_lane_u32(vec64a, 0);
```

- D register access from Q register:

```
vec64a = vget_low_u32(vec128);
vec64b = vget_high_u32(vec128);
```

- Vector Casting (data type conversion):

```
uint8x8_t byteval;
uint32x2_t wordval;
byteval = vreinterpret_u8_u32(wordval);
uint8x16_t byteval2;
uint32x4_t wordval2;
byteval2 = vreinterpretq_u8_u32(wordval2);
```

Limitations

Some instructions do not have any equivalent:

- VSWP
- VLDM
- VLDR
- VMRS
- VMSR
- VPOP
- VPUSH
- VSTM
- VSTR
- VBIT
- VBIF

Example

C Language

```
int8_t a[128], b[128], c[128];
int i;

for (i = 0; i < 128; i++)
  c[i] = a[i] + b[i];
```

NEON Intrinsics

```
int8_t a[128], b[128], c[128];
int i;
int8x8_t va, vb, vc;

for (i = 0; i < 128; i+=8)
{
  va = vld1_s8(&a[i]);
  vb = vld1_s8(&b[i]);
  vc = vadd_s8(va, vb);
  vst1_s8(&c[i], vc);
}
```

```
        sub sp, sp, #384
        mov r3, #0
.L3:
        add r1, sp, #384
        add r2, r1, r3
        add r3, r3, #8
        sub r0, r2, #384
        sub r1, r2, #256
        cmp r3, #128
        sub r2, r2, #128
        vld1.8 {d17}, [r0]
        vld1.8 {d16}, [r1]
        vadd.i8 d16, d17, d16
        vst1.8 {d16}, [r2]
        bne .L3
        mov r0, #0
        add sp, sp, #384
```

Automatic Vectorization with gcc

The following options are required to enable automatic vectorization:

- `-ftree-vectorize`
- `-mfpu=neon`
- `-mcpu=cortex-a9`
- `-mfloat-abi=hard`

Moreover `-O3` imply `-ftree-vectorize`
finally `-ftree-vectorizer-verbose=1` gives useful information to understand the automatic vectorization process with `gcc`.

Example

```
int accumulate(char *c, char *d, char *restrict e, int len)
{
  int i;
  for(i=0 ; i < (len & ~3) ; i++)
  {
    e[i] = d[i] + c[i];
  }
  return i;
}
```

Additional informations

- simple and short loop are more easily optimized
- avoid break statement to exit a loop
- give indications to known the number of iteration
- any function call inside a loop should be `inline`
- use the keyword `restrict` ou `__restrict`
- avoid memory between iterations
- use table with index (no pointers)
- use the shortest data type to hold the result

Utilization of pre-compiled libraries

Some libraries have been optimized for the ARM architecture and support NEON extension:

- OpenMAX: signal, audio, image processing. Library written by the Khronos group (OpenCL, OpenGL...)
- ffmpeg: audio and video codecs (VLC, GStreamer...)
- Eigen3: mathematical library
- x264: video codec
- Bluez: Bluetooth stack
- Pixman: image processing
- FFTW: library used for FFT operation (Matlab)

References

Documentation is freely available on ARM website (and on chamilo).

- ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition
- NEON Programmer's Guide
- Cortex -A9 NEON Media Processing Engine, Technical Reference Manual