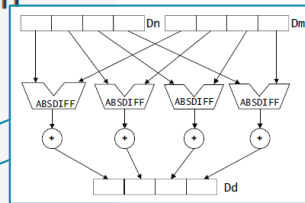


NEON Advanced SIMD Instructions

References

- NEON Programmer's Guide DEN0018 (NPG) – read this first!

📖	NEON Programmer's Guide
📖	Contents
+ 📖	Preface
+ 📖	1: Introduction
+ 📖	2: Compiling NEON Instructions
+ 📖	3: NEON Instruction Set Architecture
+ 📖	4: NEON Intrinsics
+ 📖	5: Optimizing NEON Code
+ 📖	6: NEON Code Examples with Intrinsics
+ 📖	7: NEON Code Examples with Mixed Operations
+ 📖	8: NEON Code Examples with Optimization
+ 📖	A: NEON Microarchitecture
+ 📖	B: Operating System Support
+ 📖	C: NEON and VFP Instruction Summary
+ 📖	D: NEON Intrinsics Reference

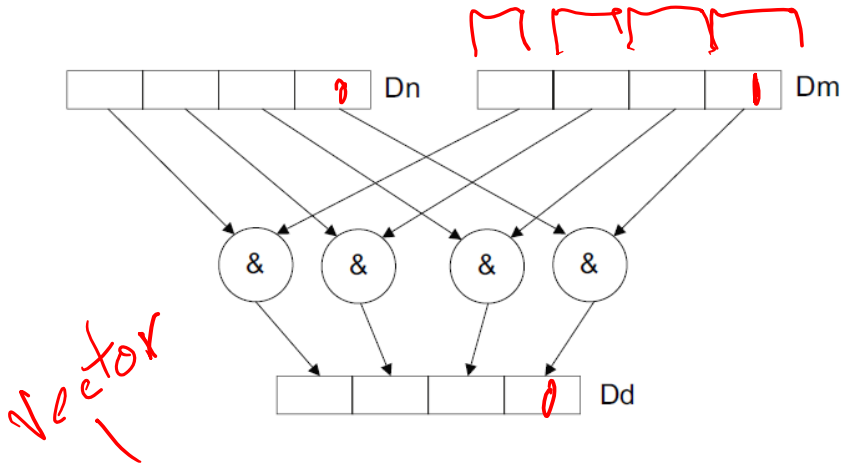


- Instr. Functionality: ARM Arch. Ref. Manual
 - Load/Store: 4.11
 - Register Transfer: 4.12
 - Data Processing: 4.13, 4.14
- ARM C Language Extensions IHI0053 (ACLE)
- ARM NEON Intrinsics Reference IHI0073 (NIR)
- Performance: Cortex-A72 Software Optimization Guide UAN0016

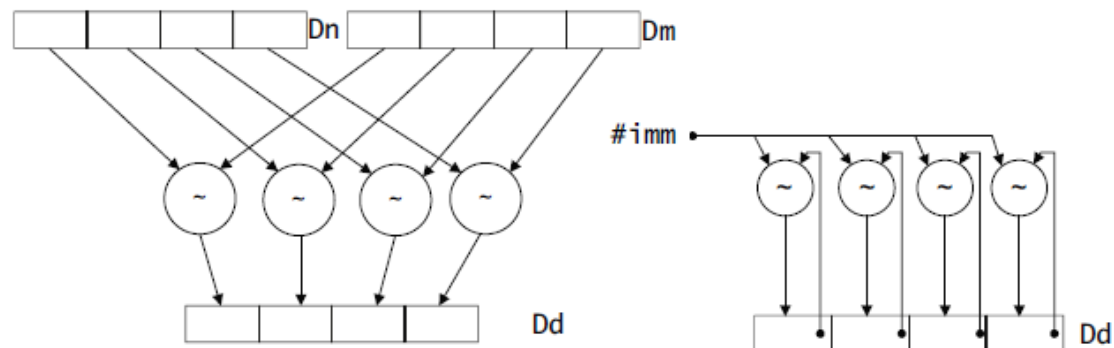
BASIC ASIMD INSTRUCTIONS: INDEPENDENT LANES

Bitwise Logic

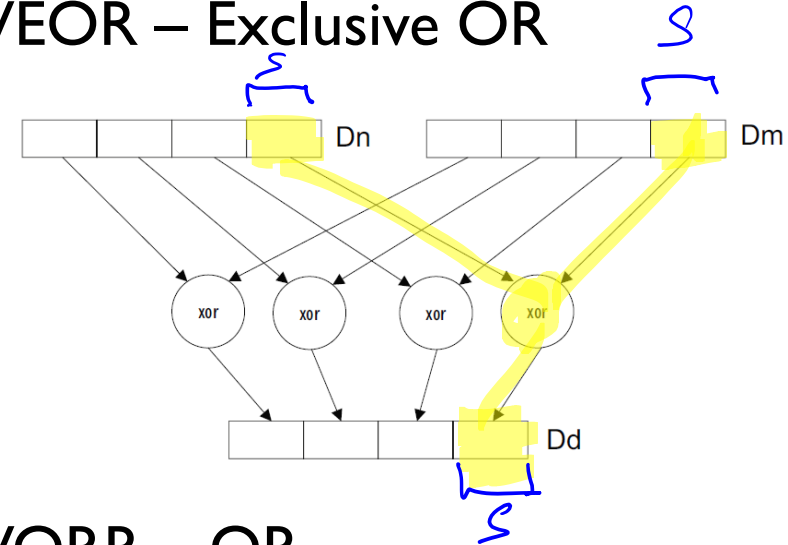
■ VAND – AND



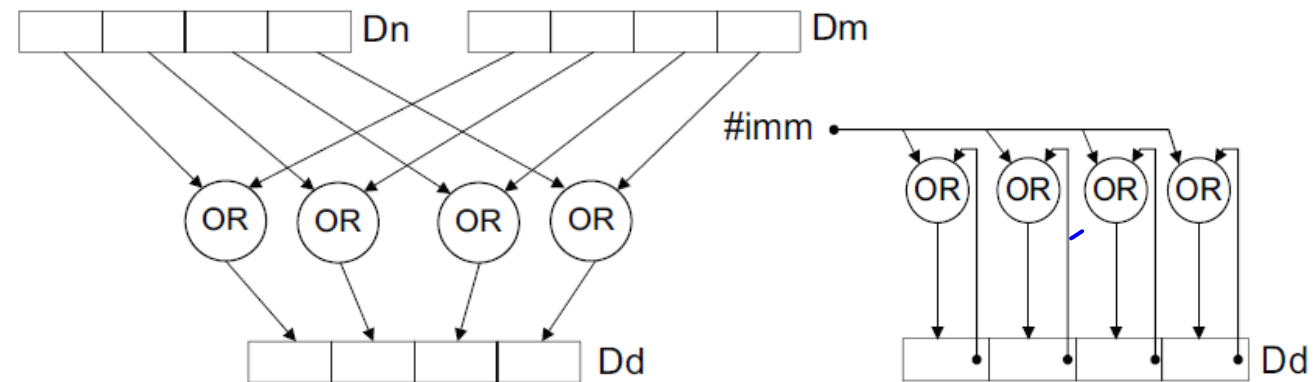
■ VBIC – Bit Clear ($\text{arg1} \& \sim \text{arg2}$)



■ VEOR – Exclusive OR

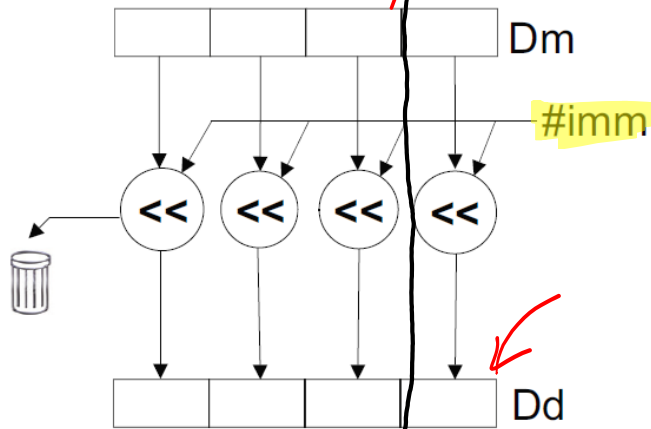


■ VORR – OR

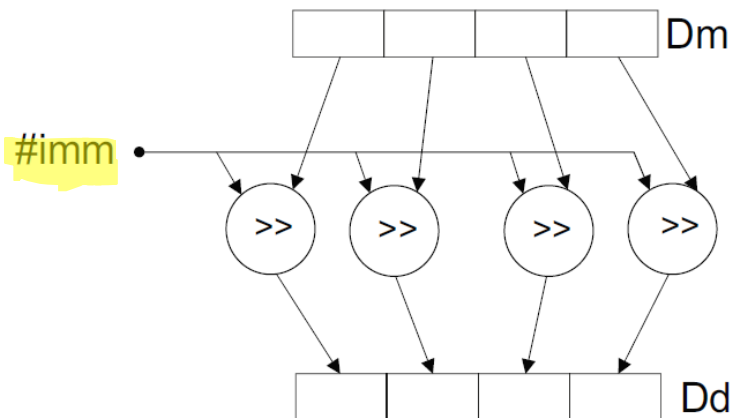


Shifts

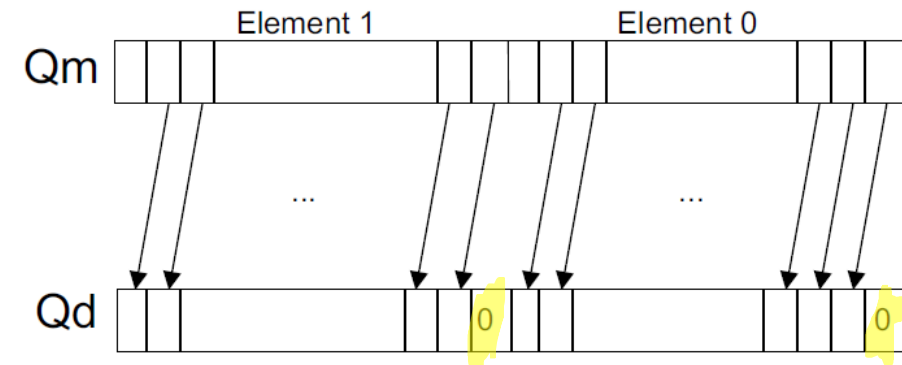
- VSHL – shift left



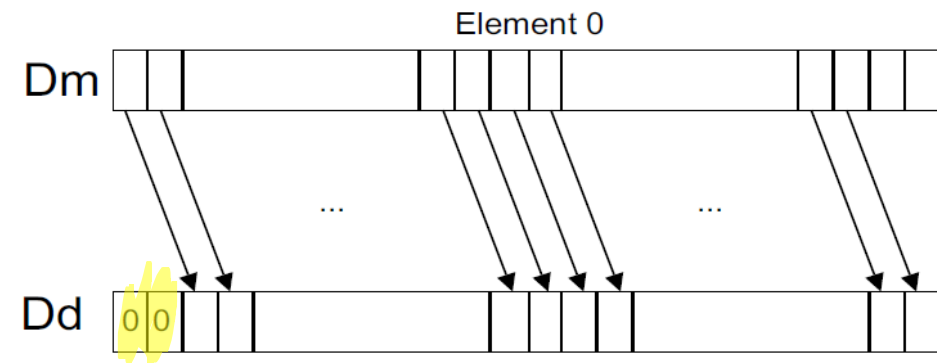
- VSHR – shift right



- VSLL – shift left and insert, leaving lower bits unchanged

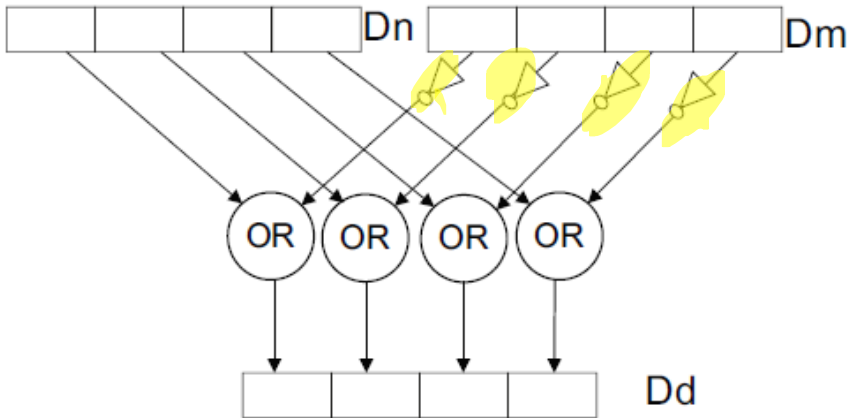


- VSRL – shift right and insert, leaving upper bits unchanged

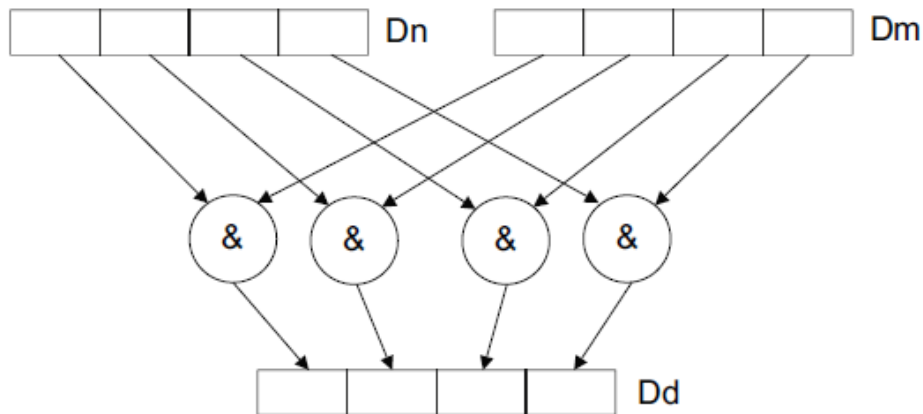


Bitwise Logic and Move

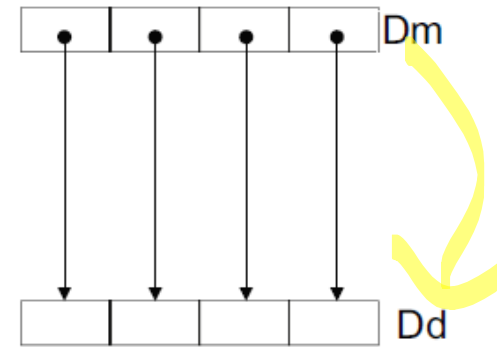
■ VORN – Bitwise OR not



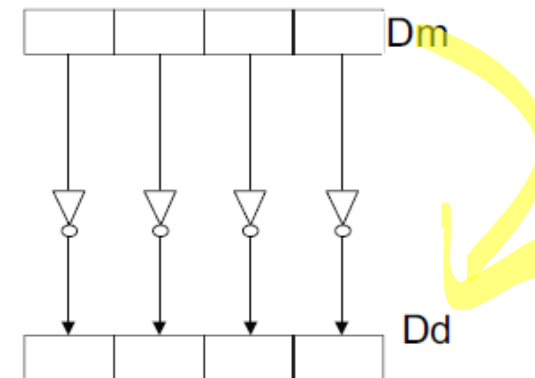
■ VTST – If element-wise AND is non-zero, set all element bits to 1



■ VMOV - Move

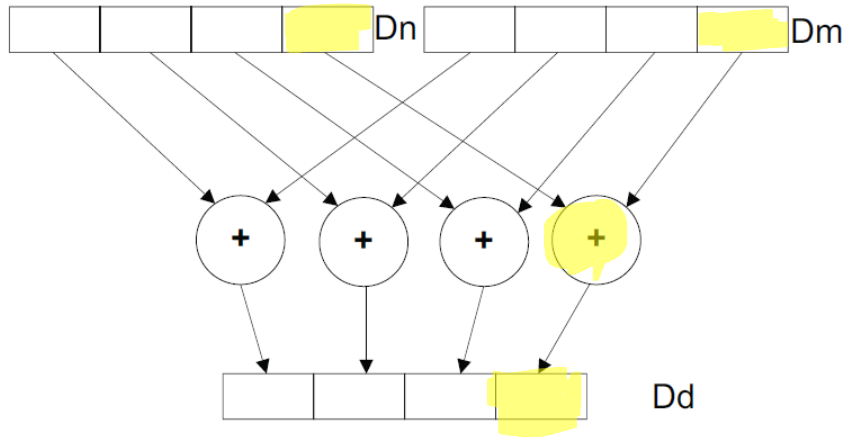


■ VMVN – Move Not. Invert all bits.

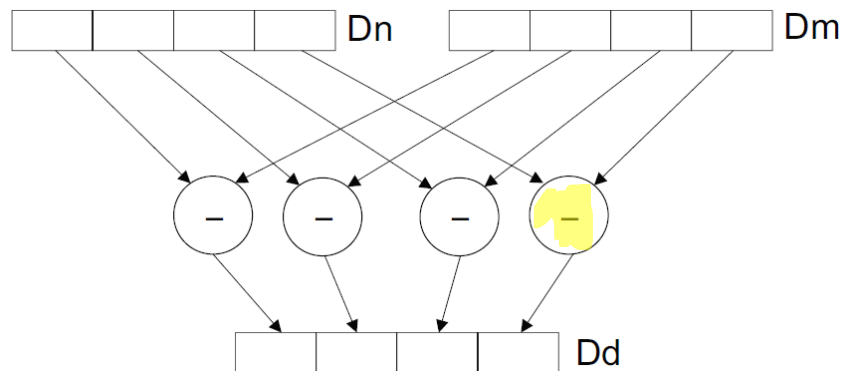


Math

VADD



VSUB

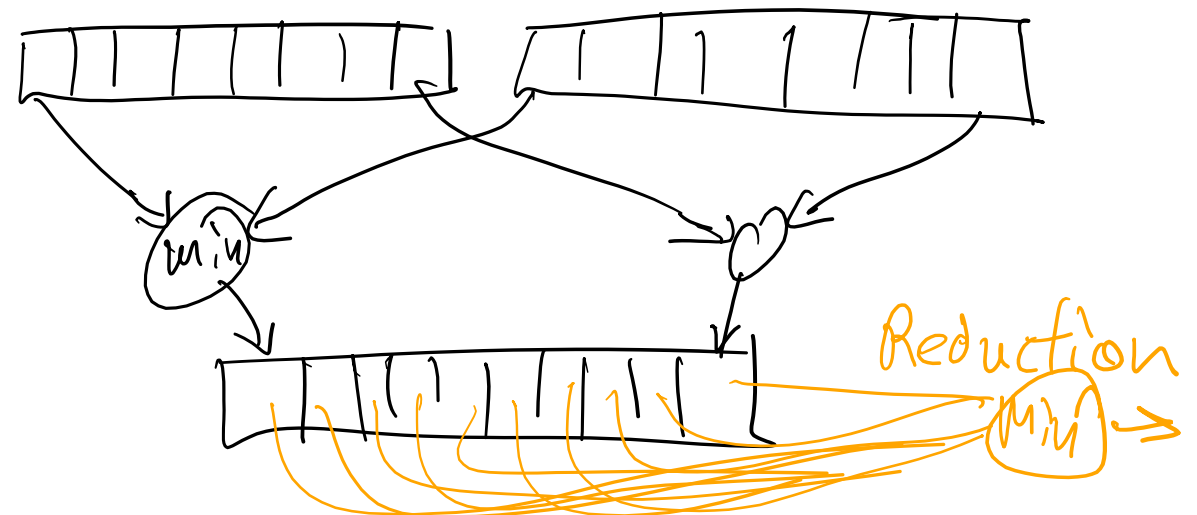


VNEG – Negate

- Destination = $-1 \times \text{source}$

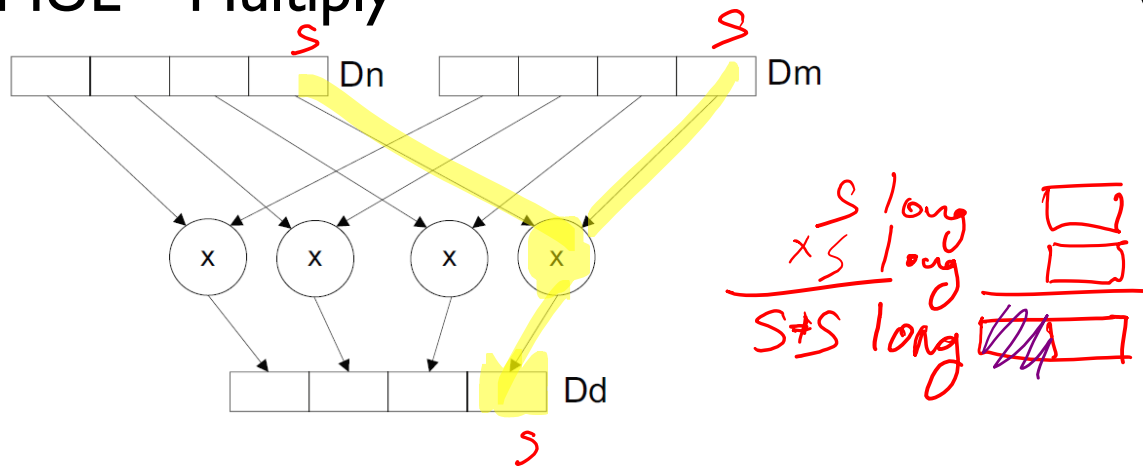
VMAX – write larger of two source lanes to destination lane

VMIN – write smaller of two source lanes to destination lane

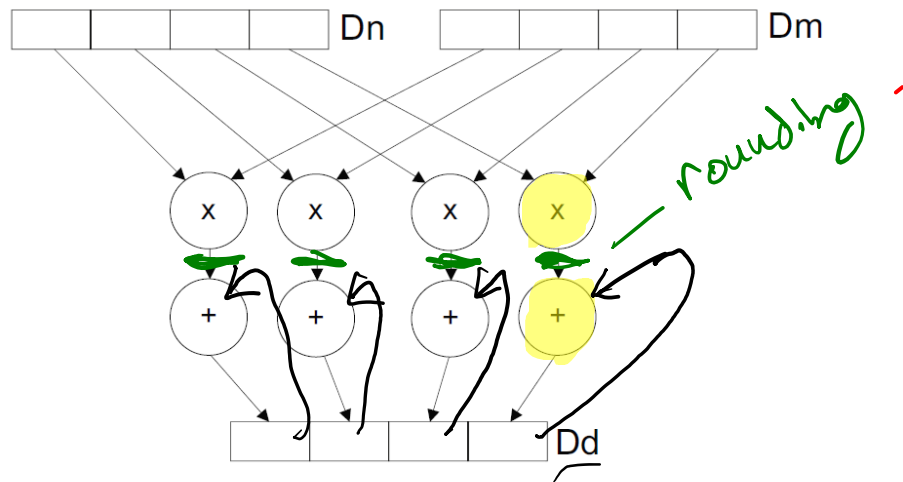


SIMD Math with Multiply

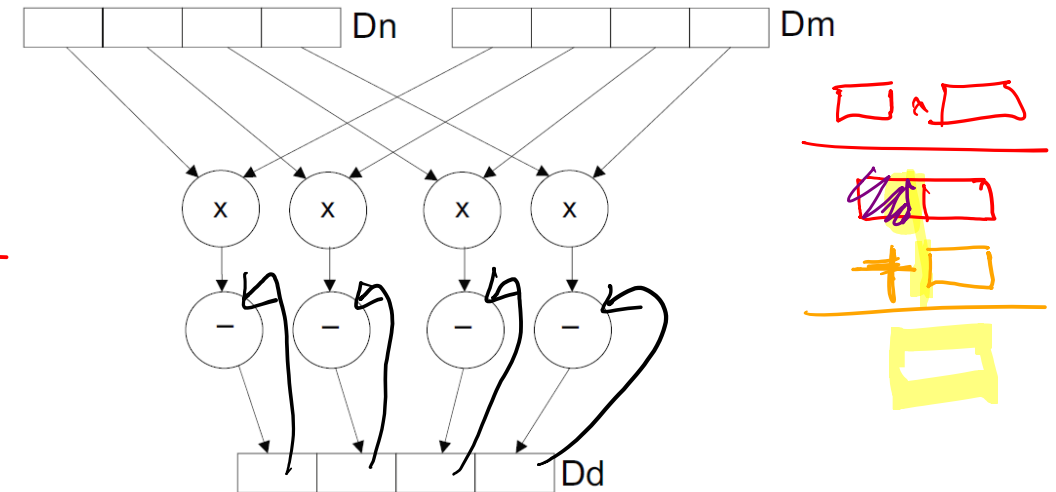
VMUL – Multiply



VMLA – Multiply Accumulate



VMSSA – Multiply Subtract



VFMA – Fused Multiply Accumulate

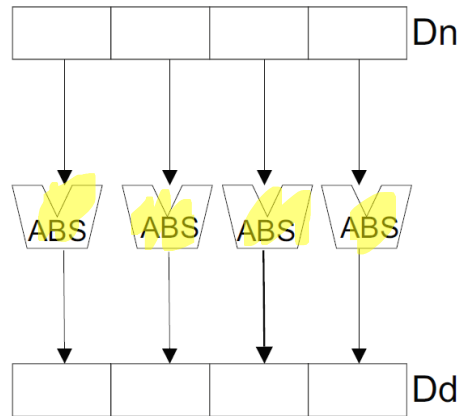
- Products not rounded before adds, so better accuracy

VFMS – Fused Multiply Subtract

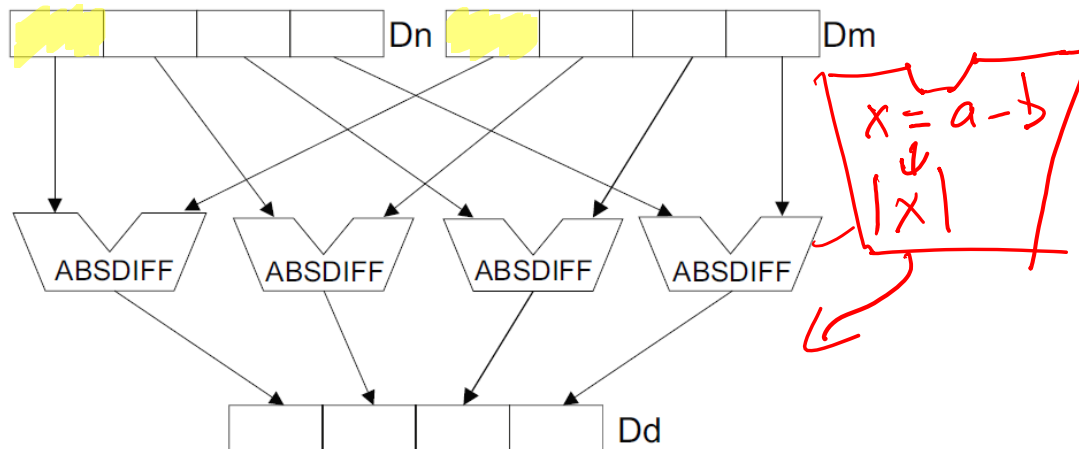
- Products not rounded before subtracts, so better accuracy

Absolute Values

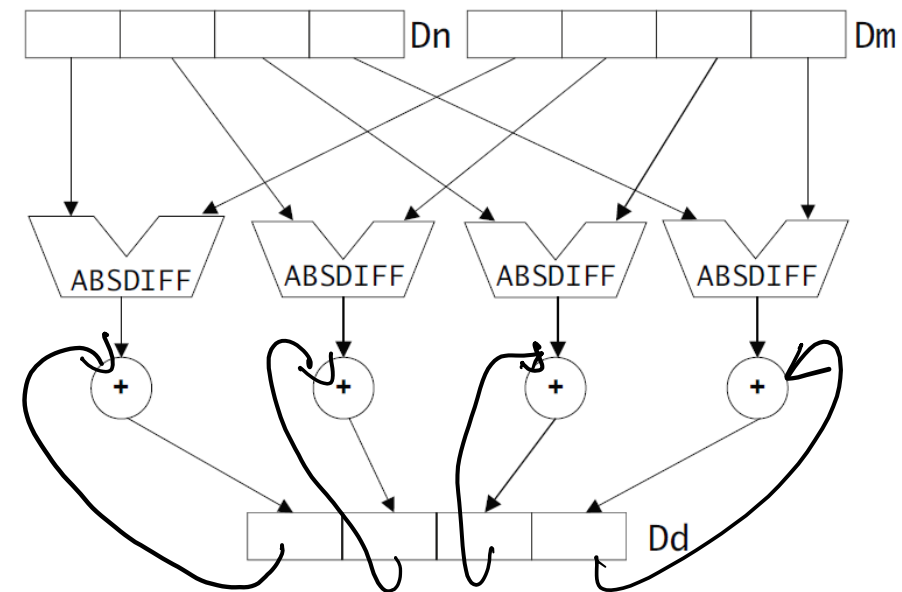
VABS – Absolute Value



VABD – Absolute Value of Difference

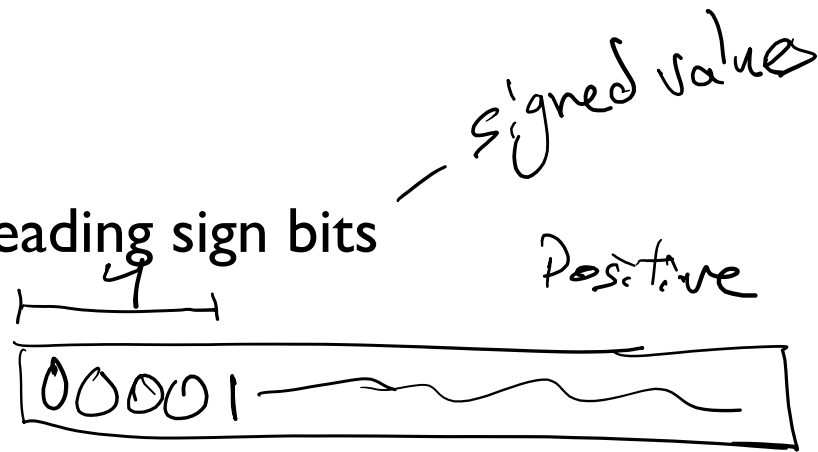


VABA – Absolute Value of Difference and Accumulate



Counting

- VCLS – Count consecutive leading sign bits



$$\text{number} < 2^{(32-4)} = 2^{28}$$

- VCLZ – Count consecutive leading zeroes

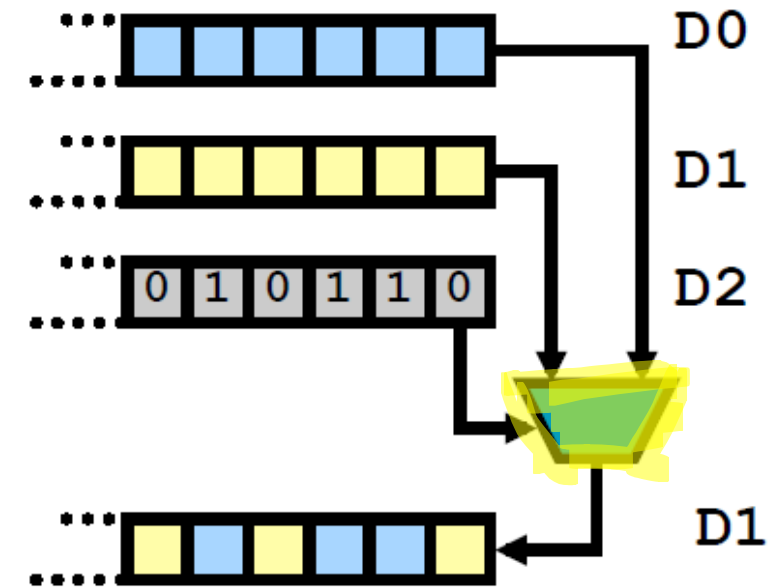
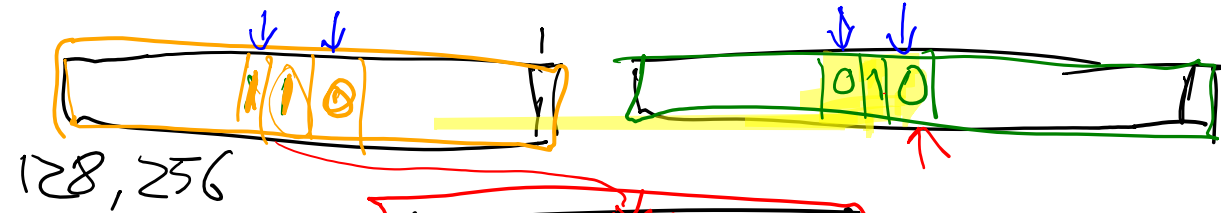


unsigned values

- VCNT – Count set (1) bits

Bitwise Multiplex Operations

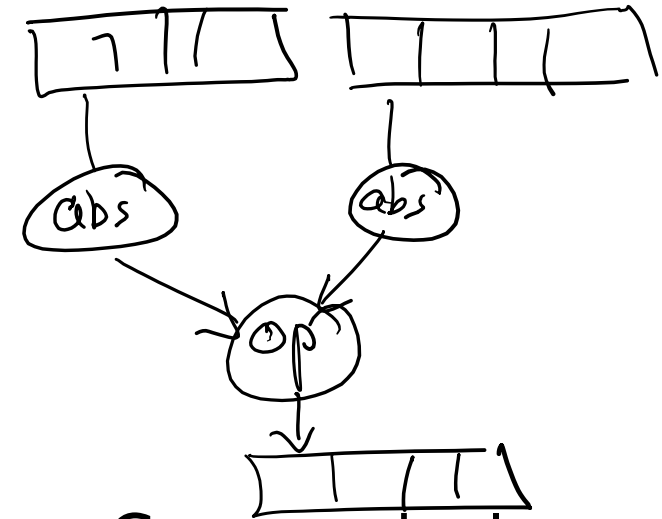
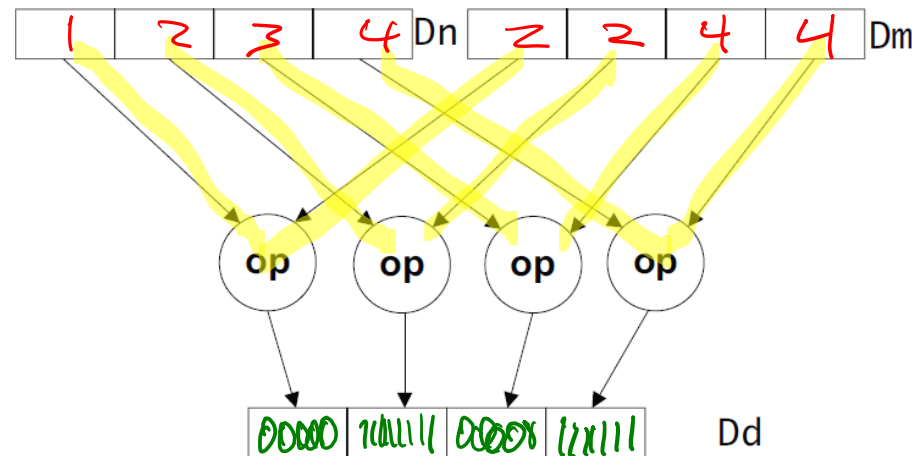
- Bitwise: each lane is one bit wide
- Copy bits specified by mask register from source register to destination register
- VBIT: Bitwise Insert if True
 - Q_m is mask register
 - VBIT Q_d, Q_n, Q_m : If $Q_m[i]$ is one, copy $Q_n[i]$ to $Q_d[i]$
- VBIF: Bitwise Insert if False
 - Q_m is mask register
 - VBIF Q_d, Q_n, Q_m : If $Q_m[i]$ is zero, copy $Q_n[i]$ to $Q_d[i]$
- VBSL: Bitwise Select
 - Q_d is mask register
 - VBSL Q_d, Q_n, Q_m : If $Q_d[i]$ is one, copy $Q_n[i]$ to $Q_d[i]$, else copy $Q_m[i]$ to $Q_d[i]$



VBIF D1, D0, D2

Compare and Absolute Compare

VCEQ



Compare: Compare elements

- $VCoP \{Qd,\} Qn, Qm$
 - Inputs $Qn[i], Qm[i]$: integer (8, 16, 32) or float (32)
 - Output $Qd[i]$ integer as wide as input
 - Compares $|Qn[i]|$ and $|Qm[i]|$
 - Is result true?
 - Yes: $Qd[i] = 11\dots11$
 - No: $Qd[i] = 00\dots00$
- Instructions: VCEQ, VCLE, VCLT, VCGE, VCGT, VCLE, VCLT

Absolute Compare: Compare absolute values of elements

- $VACoP \{Qd,\} Qn, Qm$
 - Inputs $Qn[i], Qm[i]$: must be float (F32)
 - Output $Qd[i]$ integer as wide as input (32 bits)
 - Compares $|Qn[i]|$ and $|Qm[i]|$
 - Is result true?
 - Yes: $Qd[i] = 11\dots11$
 - No: $Qd[i] = 00\dots00$
- Instructions: VACGE, VACGT, VACLE, VACL

Vector Load/Store

Table A4-13 Extension register load/store instructions

Instruction	See	Operation
Vector Load Multiple	<i>VLDM</i> on page A8-626	Load 1-16 consecutive 64-bit registers (Adv. SIMD and VFP) Load 1-16 consecutive 32-bit registers (VFP only)
Vector Load Register	<i>VLDR</i> on page A8-628	Load one 64-bit register (Adv. SIMD and VFP) Load one 32-bit register (VFP only)
Vector Store Multiple	<i>VSTM</i> on page A8-784	Store 1-16 consecutive 64-bit registers (Adv. SIMD and VFP) Store 1-16 consecutive 32-bit registers (VFP only)
Vector Store Register	<i>VSTR</i> on page A8-786	Store one 64-bit register (Adv. SIMD and VFP) Store one 32-bit register (VFP only)

Memory

- VLD
- VST
- VLDM
- VSTM
- VLDR
- VSTR
- VPOP
- VPUSH

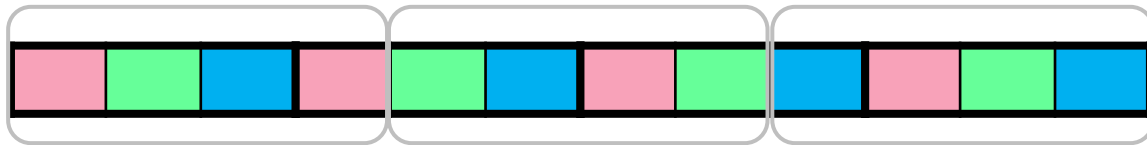
Move (See NPG, Appendix C)

- VMOV
- VDUP
- VEXT
- VMN
- VREV
- VSWP
- VTRN
- VUZP
- VZIP

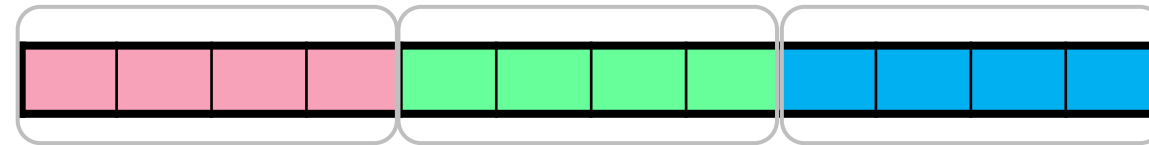
Structure Load/Store Instructions with Element De-Interleaving/Interleaving

Arrays and Structures

Memory



Registers



■ Array of structures

```
struct {
    uint8_t Red, Green, Blue;
} image[N];
```

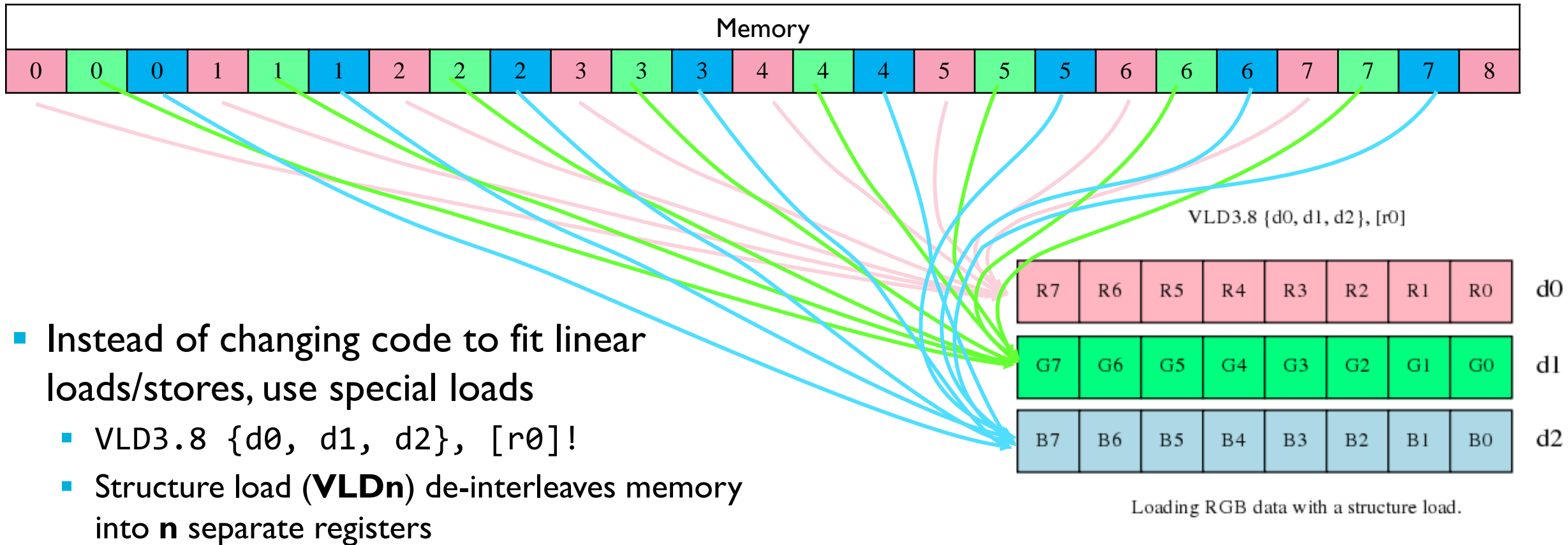
- Not a great fit for load/store register instructions

- Could rewrite code to rearrange data in memory into a **structure of arrays**:

```
struct {
    uint8_t Red[N], Green[N], Blue[N];
} image;
```

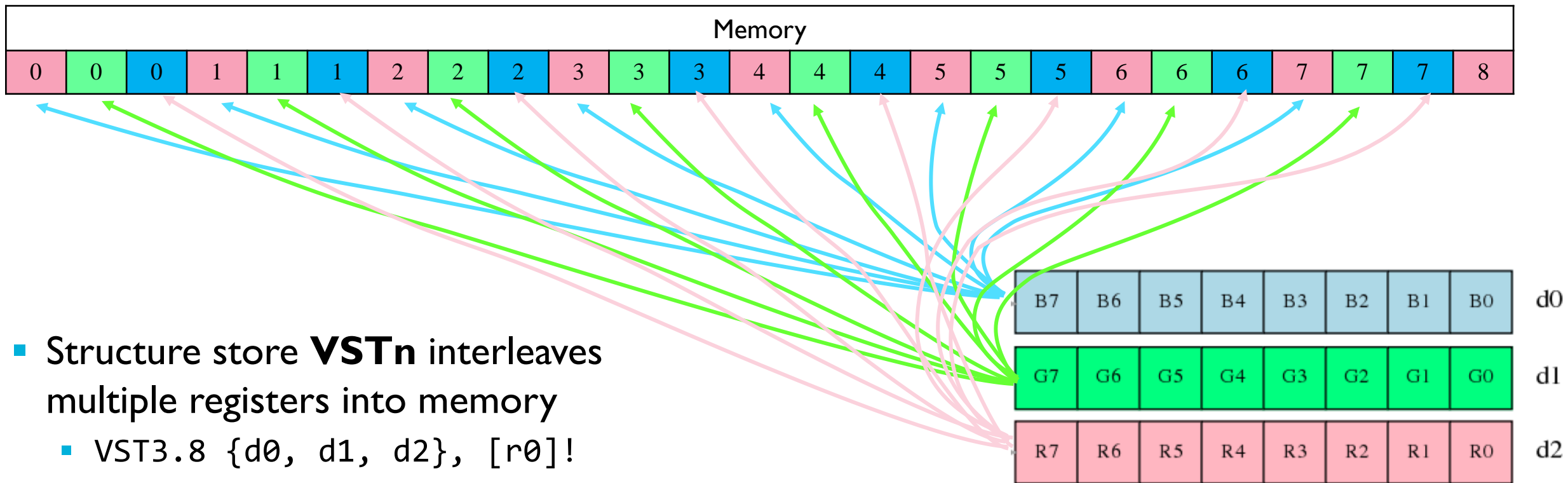
- Is better fit for load/store instructions
- Requires significant code modifications ☹️

“Structure Load” De-Interleaves From Memory Into Registers

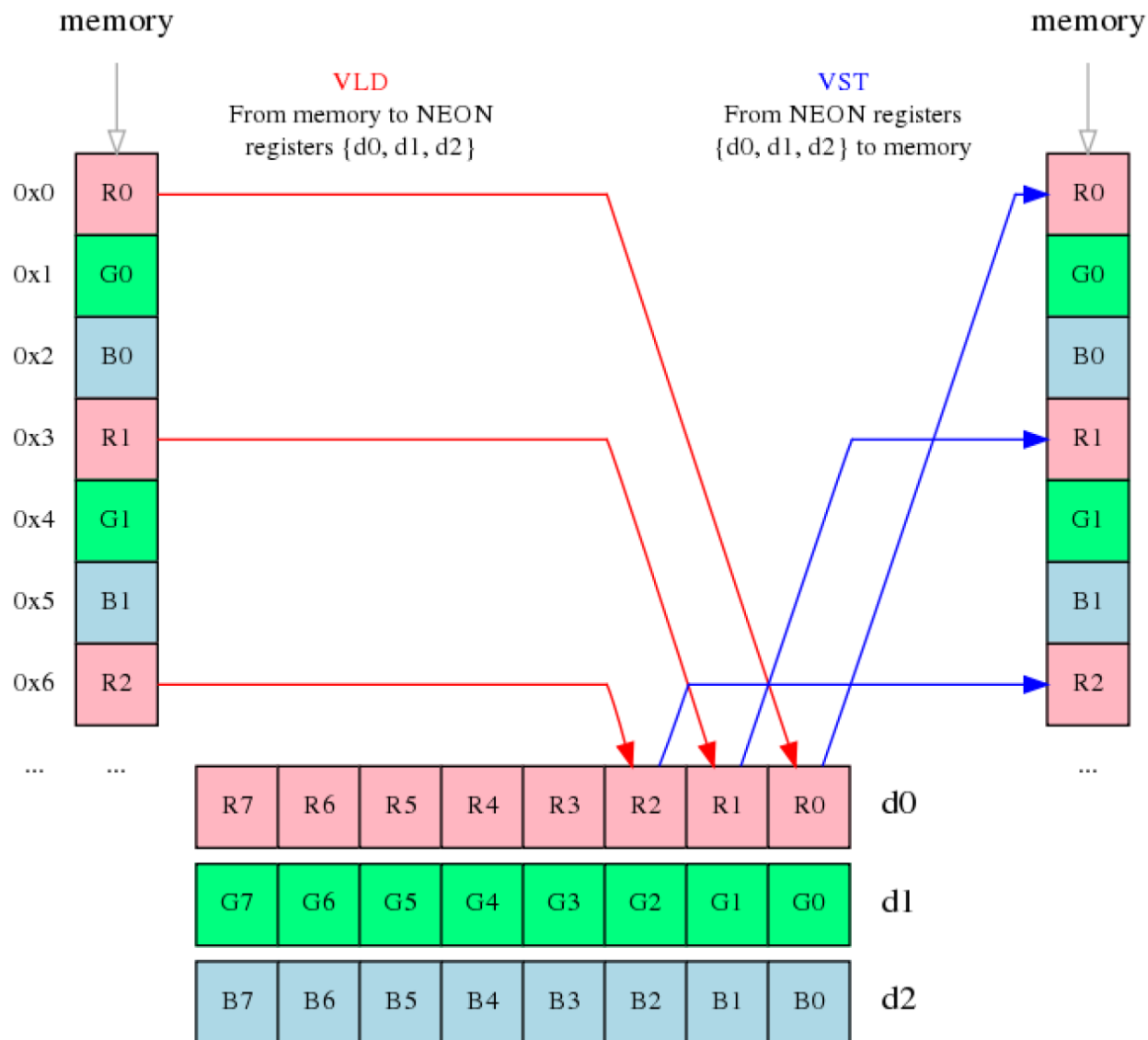


- Instead of changing code to fit linear loads/stores, use special loads
 - VLD3.8 {d0, d1, d2}, [r0]!
 - Structure load (**VLDn**) de-interleaves memory into **n** separate registers
- Instructions: NPG, page C-63

“Structure Store” Interleaves From Registers Into Memory



Another View



NEON structure loads and stores.

- Have support for 2, 3 and 4 element structures
- How well can it work?
 - Wide interfaces between NEON registers and memory
 - LI Data Cache
 - 128 bit interface

Load Structure

- Multiple Structure Access e.g. {D0, D1}
- Single Structure Access e.g. {D0[2], D1[2]}
- Single Structure Load to all lanes e.g. {D0[], D1[]}

Load single element

Multiple elements	<i>VLD1 (multiple single elements) on page A8-602</i>
To one lane	<i>VLD1 (single element to one lane) on page A8-604</i>
To all lanes	<i>VLD1 (single element to all lanes) on page A8-606</i>

Load 2-element structure

Multiple structures	<i>VLD2 (multiple 2-element structures) on page A8-608</i>
To one lane	<i>VLD2 (single 2-element structure to one lane) on page A8-610</i>
To all lanes	<i>VLD2 (single 2-element structure to all lanes) on page A8-612</i>

Load 3-element structure

Multiple structures	<i>VLD3 (multiple 3-element structures) on page A8-614</i>
To one lane	<i>VLD3 (single 3-element structure to one lane) on page A8-616</i>
To all lanes	<i>VLD3 (single 3-element structure to all lanes) on page A8-618</i>

Load 4-element structure

Multiple structures	<i>VLD4 (multiple 4-element structures) on page A8-620</i>
To one lane	<i>VLD4 (single 4-element structure to one lane) on page A8-622</i>
To all lanes	<i>VLD4 (single 4-element structure to all lanes) on page A8-624</i>

Store Structure

Store single element

Multiple elements	<i>VST1 (multiple single elements)</i> on page A8-768
-------------------	---

From one lane	<i>VST1 (single element from one lane)</i> on page A8-770
---------------	---

Store 2-element structure

Multiple structures	<i>VST2 (multiple 2-element structures)</i> on page A8-772
---------------------	--

From one lane	<i>VST2 (single 2-element structure from one lane)</i> on page A8-774
---------------	---

Store 3-element structure

Multiple structures	<i>VST3 (multiple 3-element structures)</i> on page A8-776
---------------------	--

From one lane	<i>VST3 (single 3-element structure from one lane)</i> on page A8-778
---------------	---

Store 4-element structure

Multiple structures	<i>VST4 (multiple 4-element structures)</i> on page A8-780
---------------------	--

From one lane	<i>VST4 (single 4-element structure from one lane)</i> on page A8-782
---------------	---

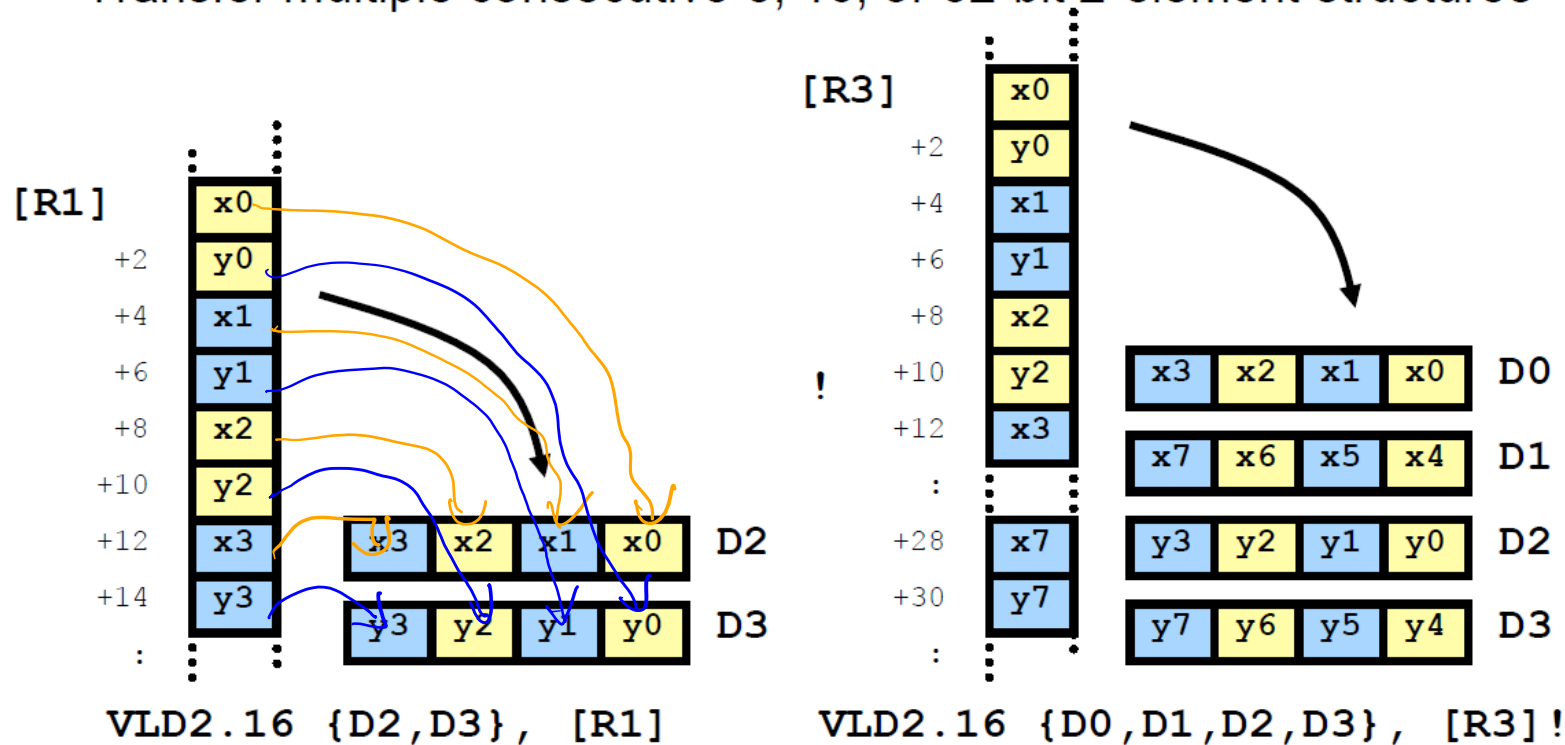
Structure VLD/VST and Operand Syntax

Elements per Structure	Load all structures to all lanes. Load every element of every structure	Load one structure to registers. Load one element into one lane in each register	Load one structure to all lanes. Load multiple copies of structure elements into multiple registers.
1 (no interleaving)	{D0, D1}	{D0[2], D1[2]}	{D0[], D1[]}
2			
Elements per Structure	Store all registers to all lanes. Store every element of every structure	Load one structure to one lane	Load one structure to all lanes
1 (no interleaving)	{D0, D1}	{D0[2], D1[2]}	{D0[], D1[]}
2			
3			
4			

- Three forms
- Forms distributed
 - Multiple
 - Single
 - Single

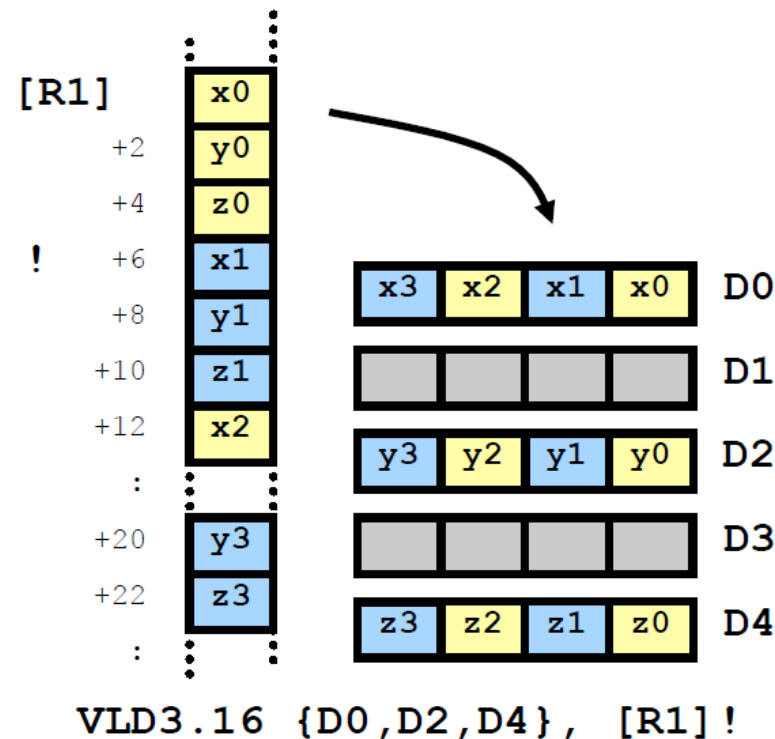
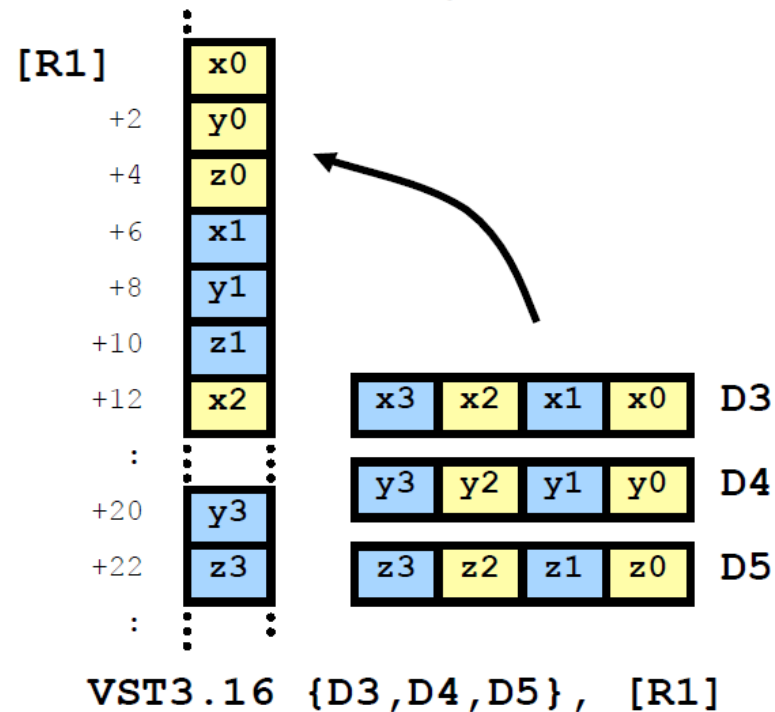
Multiple 2-Element Structure Access

- VLD2, VST2 provide access to multiple 2-element structures
 - List can contain 2 or 4 registers
 - Transfer multiple consecutive 8, 16, or 32-bit 2-element structures



Multiple 3/4-Element Structure Access

- VLD3/4, VST3/4 provide access to 3 or 4-element structures
 - Lists contain 3/4 registers; optional space for building 128-bit vectors
 - Transfer multiple consecutive 8, 16, or 32-bit 3/4-element structures

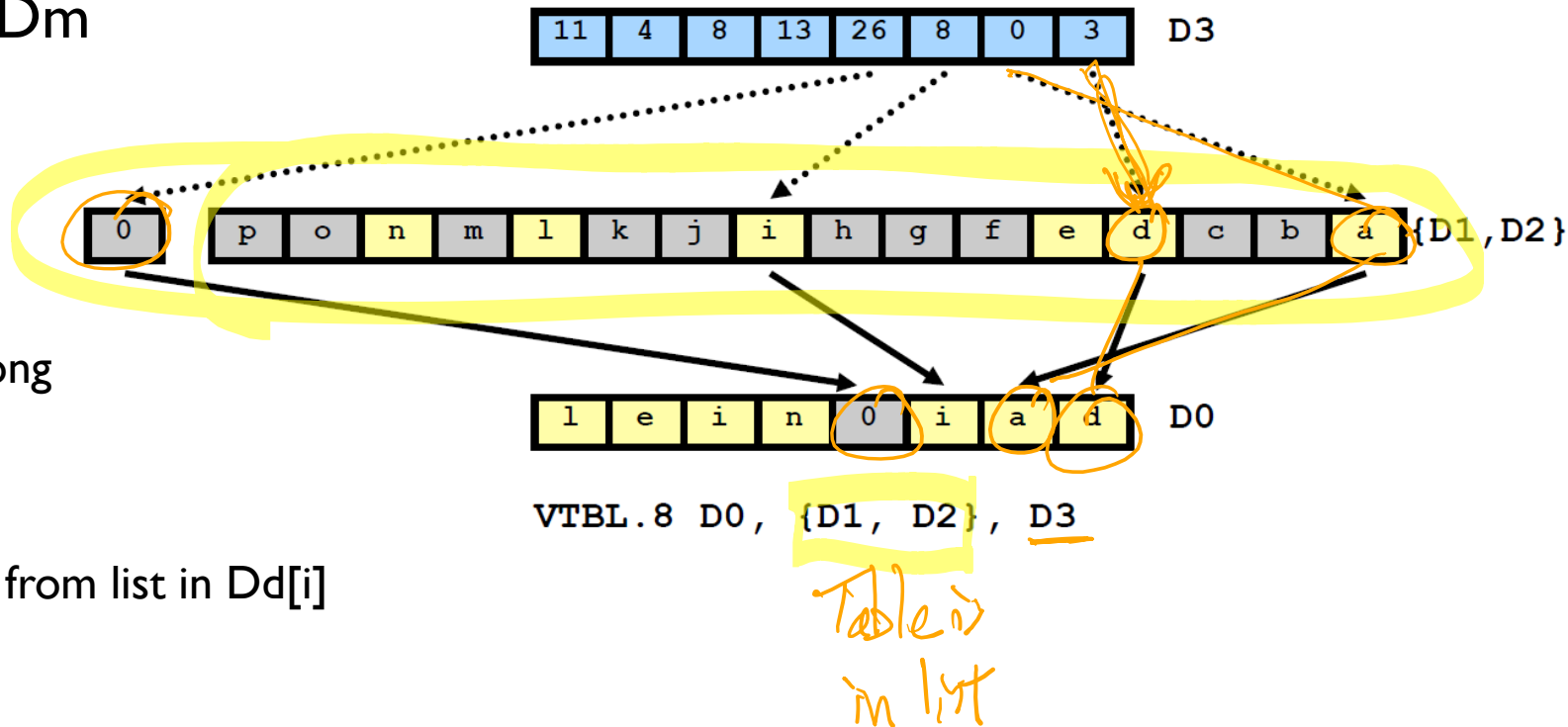


UNCOMMON ASIMD INSTRUCTIONS AND FEATURES

Table Lookup

■ Table Lookup: VTBL Dd, list, Dm

- Only works on 8-bit data
- list: holds table in one to four consecutive D registers, or two consecutive Q registers
 - Table is up to 256 bits (32 bytes) long
- Dm: vector of indices
- Is index Dm[i] in range (list)?
 - Yes: return element number Dm[i] from list in Dd[i]
 - No: else return zero in Dd[i]



■ Table extension: VTBX Dd, list, Dm

- Same as VTBL, but doesn't overwrite Dd[i] if Dm[i] is not in range

Vector Reciprocal and Reciprocal Square Root

The NEON instruction set does not include:

- division operation (use VRECPE and VRECPS instead to perform Newton-Raphson iteration)
- square root operation (use VRSQRTE and VRSQRTS and multiply instead).

- Approximate with estimate instruction, then refine step instruction(s)
- <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka14282.html>
- <https://github.com/thenifty/neon-guide>

- Estimate reciprocal

```
float32x4_t v = { 1.0, 2.0, 3.0, 4.0 };
float32x4_t reciprocal = vrecpeq_f32(v);
// => reciprocal = { 0.998046875, 0.499023438, 0.333007813, 0.249511719 }
```

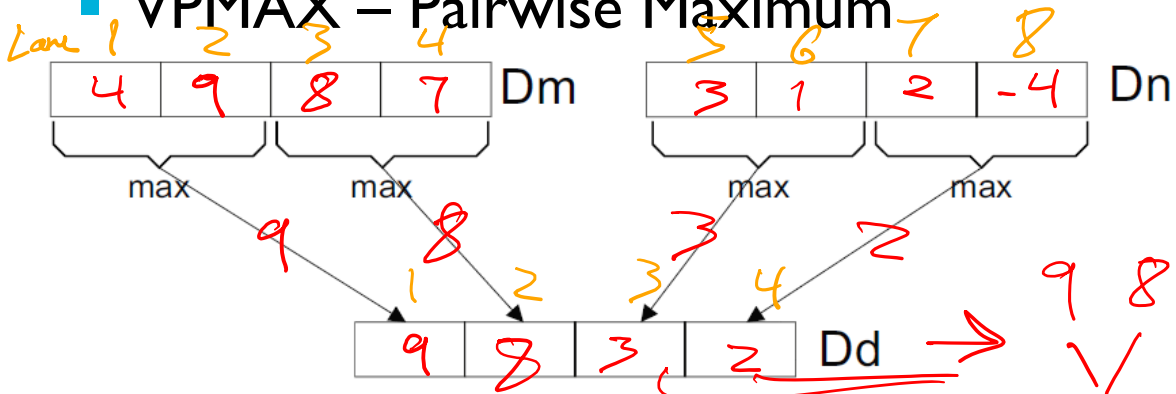
- More accurate (estimate plus one refinement step)

```
float32x4_t v = { 1.0, 2.0, 3.0, 4.0 };
float32x4_t reciprocal1 = vrecpeq_f32(v);
float32x4_t reciprocal2 = vmulq_f32(vrecpsq_f32(v, reciprocal1), reciprocal1);
// => inverse = { 0.999996185, 0.499998093, 0.333333015, 0.249999046 }
```

Lane Changes

Merge Lanes with Pairwise Reduction Operations

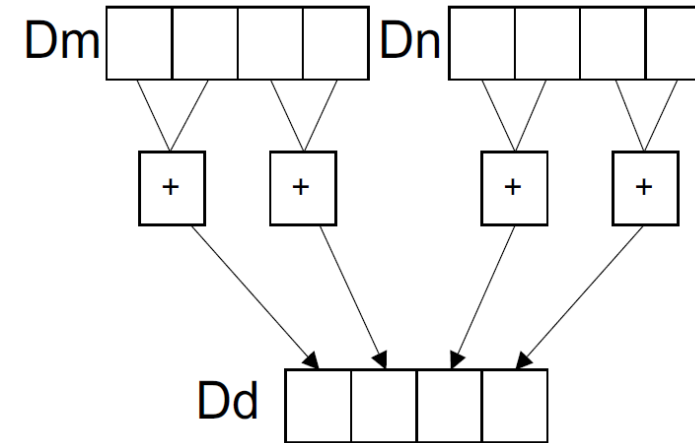
VPMAX – Pairwise Maximum



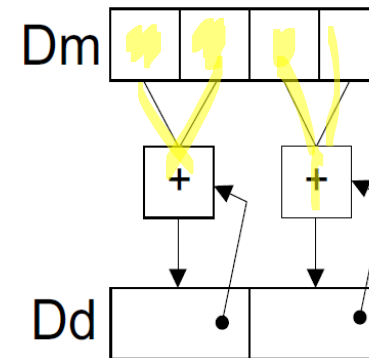
VPMIN – Pairwise Minimum



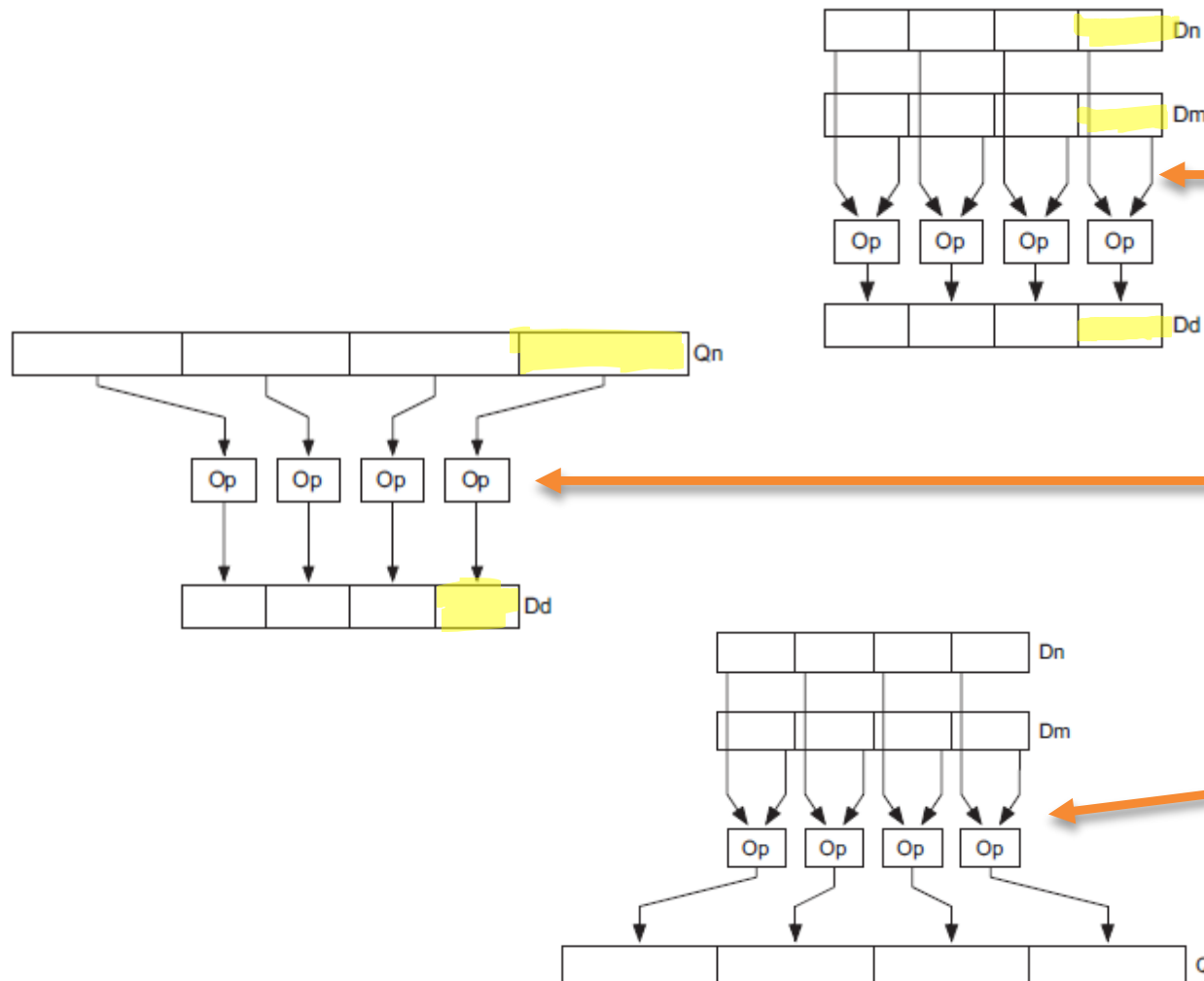
VPADD – Pairwise Add



VPADAL – Pairwise Add and Accumulate Long



Change Lane Width with Instruction “Shape” Modifiers



None specified

Both operands and results are the same width

Example:

VADD.I16 Q0, Q1, Q2

Narrow – N

Operands are the same width. Number of bits in each result element is half the number of bits in each operand element.

Example:

VADDHN.I16 D0, Q1, Q2

Long – L

Operands are the same width. Number of bits in each result element is double the number of bits in each operand element.

Example:

VADDL.S16 Q0, D2, D3


Wide – W

Result and operand are twice the width of the second operand.

Example:

VADDW.I16 Q0, Q1, D4

Modifiers for Instruction Operation

Modifier	Action	Example	Description
None	Basic operation	VADD.I16 Q0, Q1, Q2	The result is not modified
Q	Saturation 	VQADD.S16 D0, D2, D3	Each element in the result vector is set to either the maximum or minimum if it exceeds the representable range. The range depends on the type (number of bits and sign) of the elements. The sticky QC bit in the FPSCR is set if saturation occurs in any lane.
H	Halved	VHADD.S16 Q0, Q1, Q4	Each element shifted right by one place (effectively a divide by two with truncation). VHADD can be used to calculate the mean of two inputs.
D	Doubled before saturation	VQDMULL.S16 Q0, D1, D3	This is commonly required when multiplying numbers in Q15 format, where an additional doubling is needed to get the result into the correct form.
R	Rounded	VRSUBHN.I16 D0, Q1, Q3	The instruction rounds the result to correct for the bias caused by truncation. This is equivalent to adding 0.5 to the result before truncating.

Example – adding all lanes

- Input in Q0 (D0 and D1)
- u16 input values
- Now Q0 contains 4x u32 values (with 15 headroom bits)
- Reducing/folding operation needs 1 bit of headroom

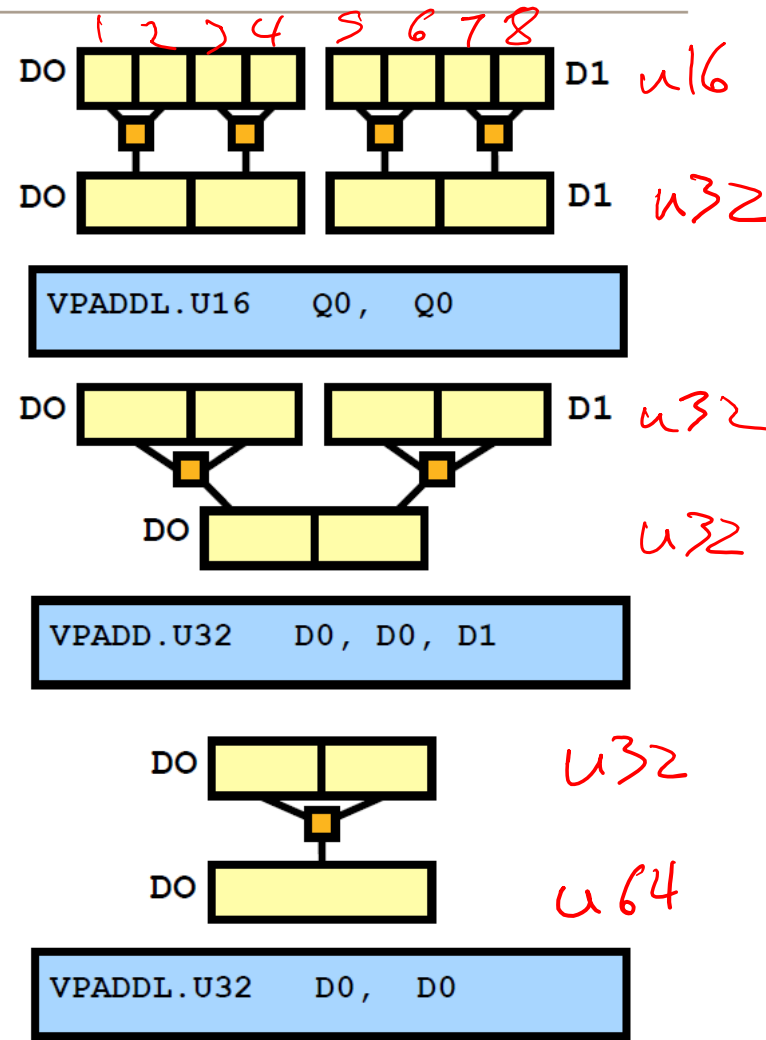


Image Format Information

A Digression: YUV Color Space



Full-Color
Image



Y Component:
Luminance



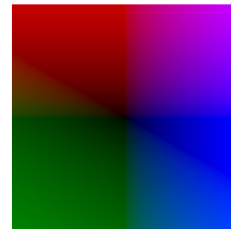
U Component:
Blue projection



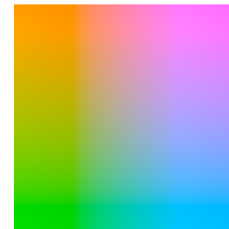
V Component:
Red Projection

- Y: Luminance (brightness)
 - Y components alone give gray-scale image (no color)
- U,V: Chrominance (color)
 - U: Blue projection
 - V: Red projection

- Slices of color space with fixed Y (luminance)



Y = 0



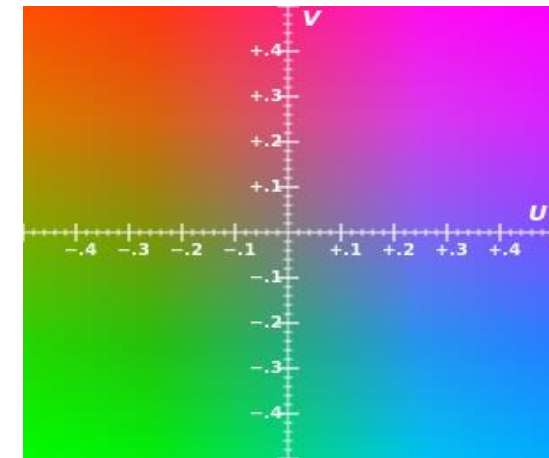
Y = 0.5



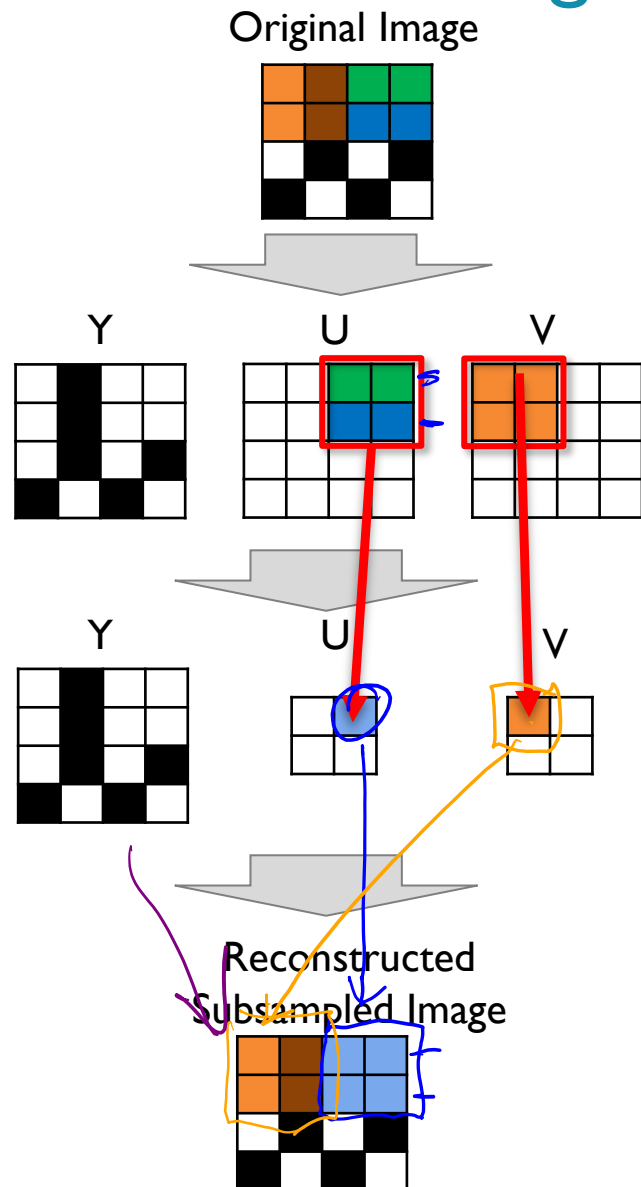
Y = 1.0

- References

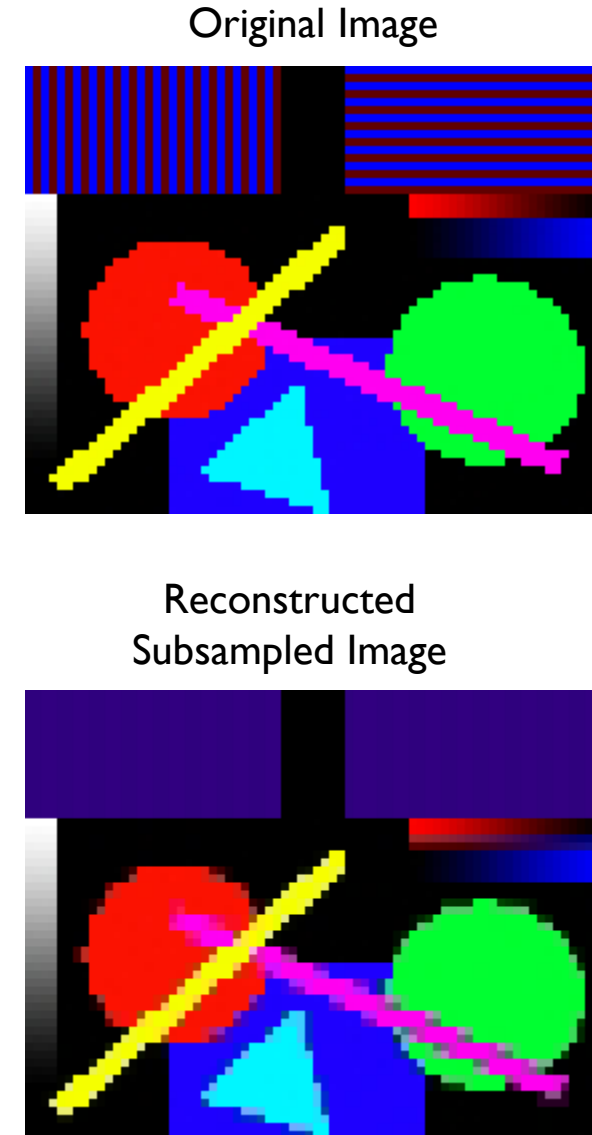
- <https://softpixel.com/~cwright/programming/colorspace/yuv/>
- <https://en.wikipedia.org/wiki/YUV>



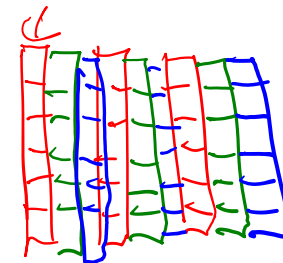
A Further Digression: YUV Chrominance Subsampling



- Retina in human eye has far more brightness sensors (rods) than color sensors (cones)
 - Worse spatial resolution for color than brightness (except in very center of vision)
- Color in digital images is often spatially sub-sampled
 - Removes information we can't see, saving time and space
 - Good explanation: <https://www.impulseadventure.com/photo/chroma-subsampling.html>
- 4:2:0 (aka 2x2) subsampling
 - Average together chroma values of 4 adjacent pixels
 - Reduces chrominance resolution by half horizontally and half vertically compared with luminance resolution
- Example:
 - 1 MPixel image needs to represent 3 million elements: 1MY, 1MU, 1MV
 - Subsampling reduces it to 1.5 million elements: 1MY, 0.25M U, 0.25M V



-



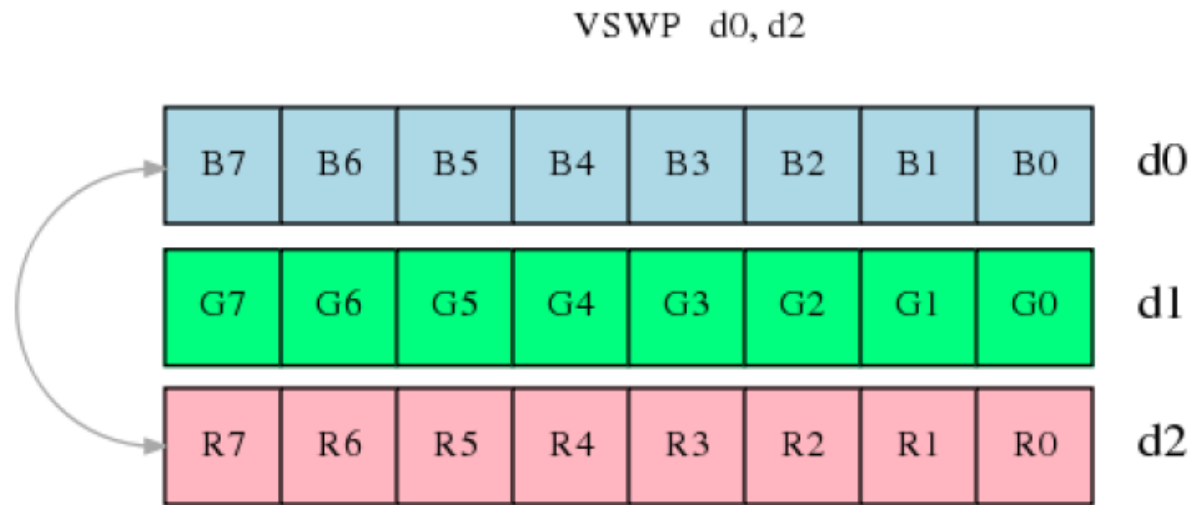
- VLD QO $p \times n + 1$ $p \times n$
- D G R B G R
- R ————— R

- 37

Swap Registers

- Now can swap red and blue easily

VSWP d0, d2



Swapping the contents of registers d0 and d2.

