

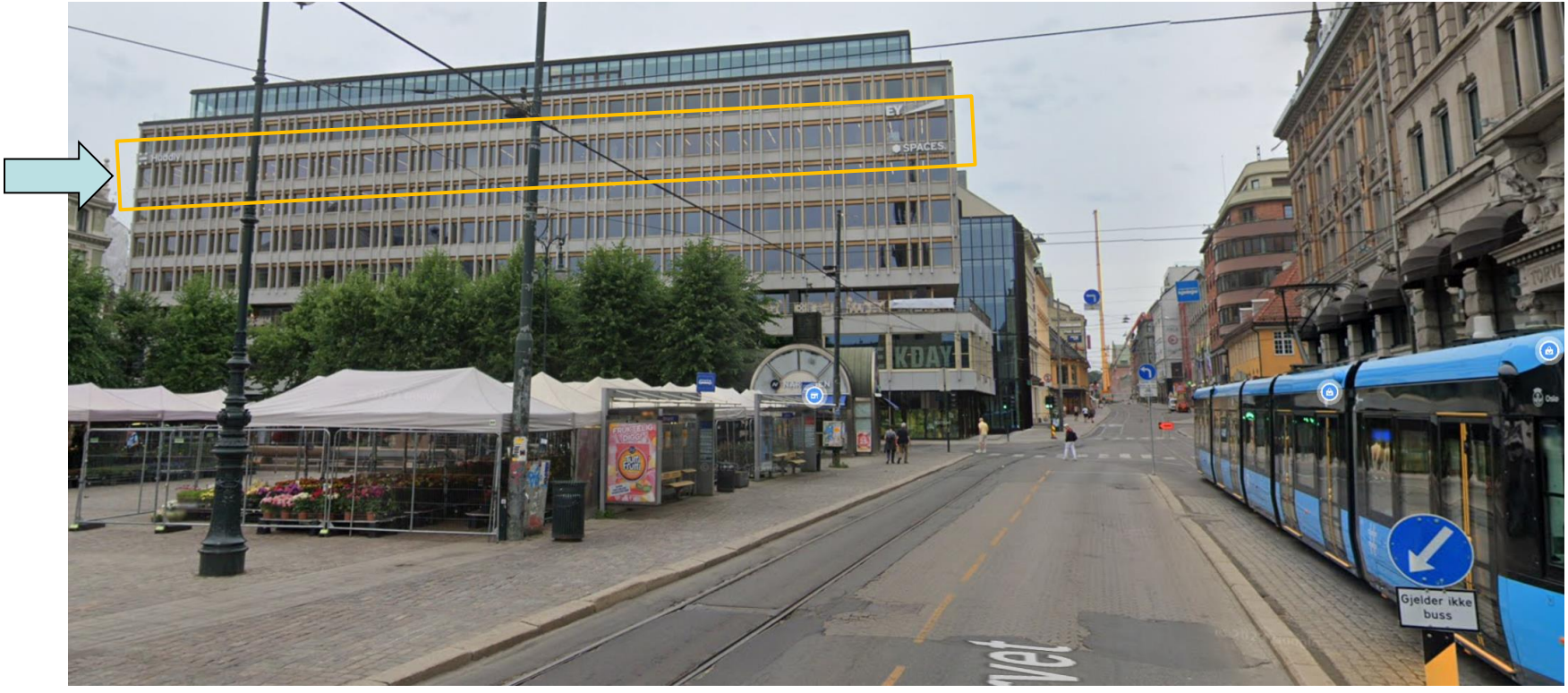
UiO : Department of Informatics
University of Oslo

Introduction to ARMv8 SIMD on the Tegra Xavier

Kristoffer Robin Stokke, PhD
Huddly



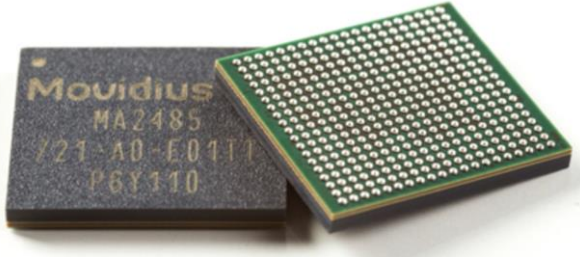
Huddly



Huddly






Movidius
MA2485
P6Y110


NXP


Ambarella
S6LM-A0-RH
NZV3X
S11121970
S6L55M

Huddly



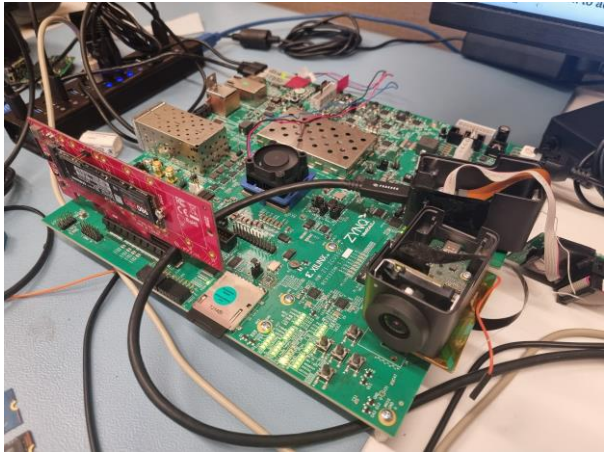


- What are these chips, really?
  - Computers!
- What do they have in common?
  - Low power
  - Inexpensive
  - Lots of IO (video/audio/network++)
  - **Accelerators**
- All these chips contain programmable processors, or **accelerators**
  - Digital signal processors (DSP)
  - Graphics processors (GPU, OpenCL)
  - **ARM cores (Neon, SIMD)**

Huddly



Sigmund Haaland



Offloading video encoding with FPGA



Sondre Bader Wang



Streaming simulated white balance test scenes to camera

Intrinsics

Why Learn to Optimize with Assembler?

- Historically: Constrained hardware
  - Entire applications written in assembler
- Still relevant today
  - but we focus on specific parts of the code



- 8 kB RAM + 8 kB VRAM
- 4.19 MHz Z80 CPU
- 10s of kBs swappable ROM

- Niche expertise: very valuable
- Challenging, rewarding and fun
  - To make non-off-the-shelf things
  - Tweak out maximum performance
- Clean, tailored C/assembler is art



- 1 MB cartridge



ARM Neon CPU core inside!



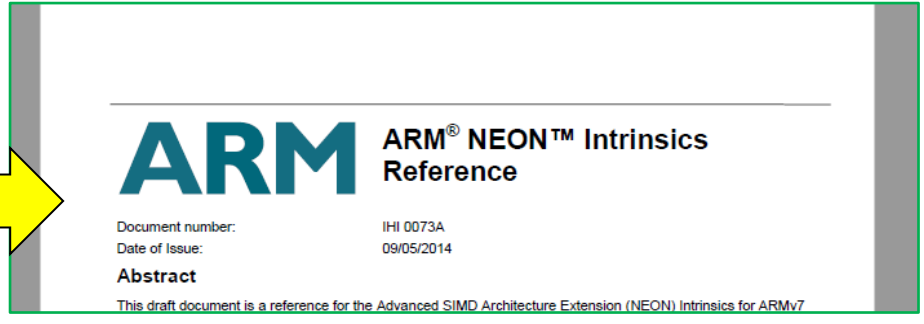
Tegra Xavier

Goals of Lecture

- To give you something concrete to start on
- Intro to **NEON** SIMD through ARM intrinsics
- Learn to step through and inspect NEON code using **gdb**
- Learn to find the proper intrinsics for a task and successfully apply them

Finding Information on Intrinsics

- Accessing compiler documentation for gcc 9.5
  - https://gcc.gnu.org/onlinedocs/
- Will find you
  - ARM C Language Extensions



- Nice, human-readable NEON overview
  - NEON+VFP Programming

Developing and Learning Neon in a Nutshell



TASK

For(i = 0; i < 7; i++)
Result[i] = a + i

Find relevant NEON instruction group

-Vector addition



Lookup instruction mnemonic



-VADD

Find intrinsic

-vaddq_u8



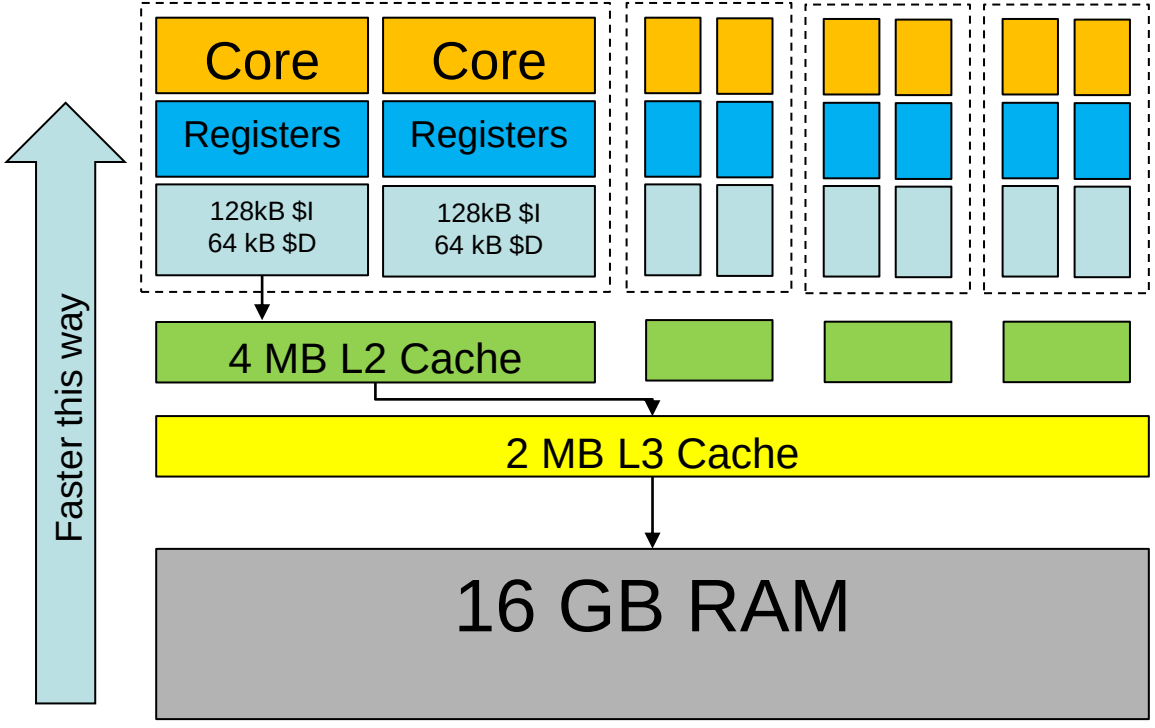
Implement



Debug



Tegra Xavier CPU Cache Hierarchy

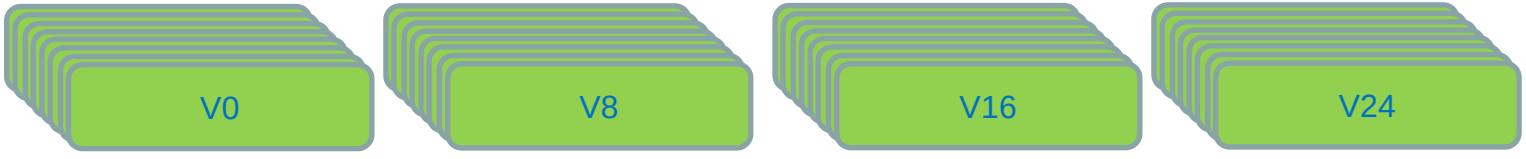


ARMv8 Register Environment

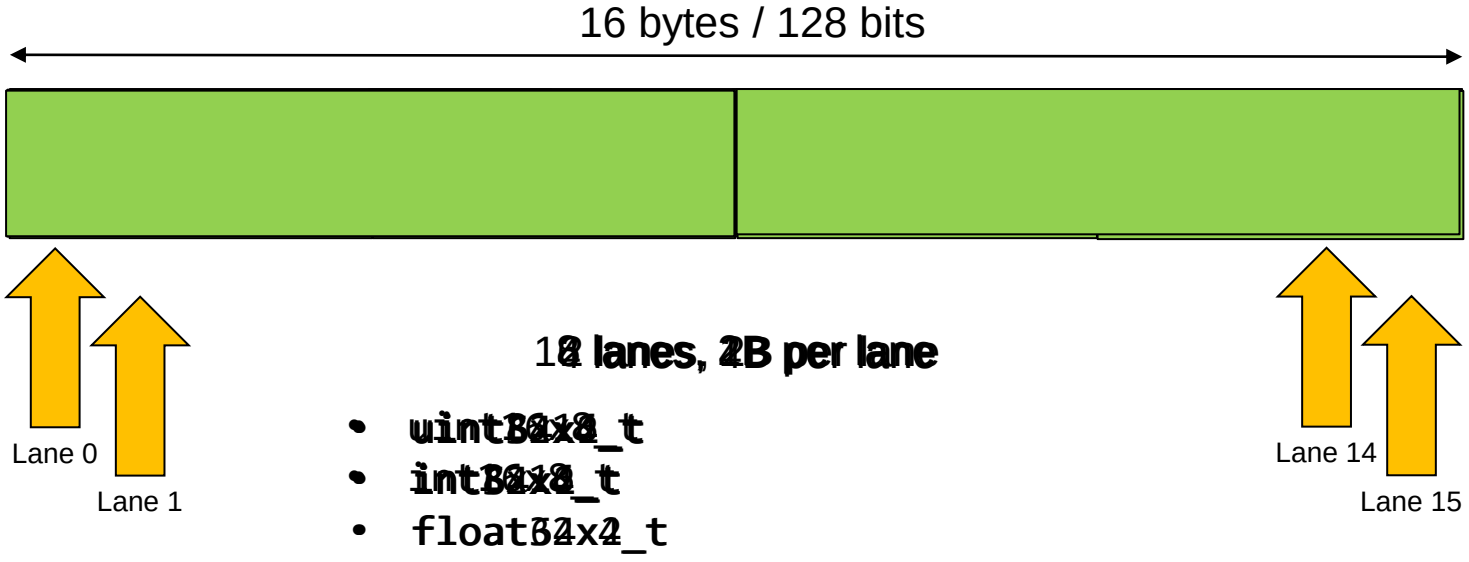
31 x 64-bit general purpose registers



32 x 128-bit vector registers



The Vector Register



16 lanes, 1B per lane

- **uint8x16_t**
- **int8x16_t**

8 lanes, 2B per lane

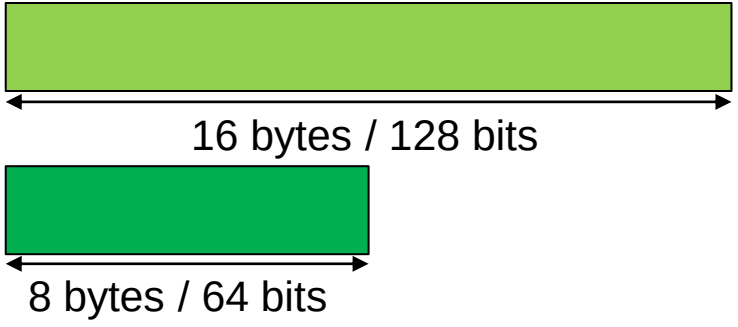
- **uint16x8_t**
- **int16x8_t**

4 lanes, 4B per lane

- **float32x4_t**

The Vector Register

- It is possible to use *half* of the vector register



- `uint8x16_t -> uint8x8_t`
- `uint32x4_t -> uint32x2_t`
- `float32x4_t -> float32x2_t`
- `float64x2_t -> float64x1_t`

- The 64-bit vector still occupies a full 128-bit vector.

The Vector Register

- Notice that the minimum supported floating point type occupies 4 bytes / 32 bits.

- You must perform conversions from/to other native formats to interact with floating point data types.

Aggregate Vector Types

- Some NEON instructions can operate on more than one vector register at a time.

- This is usually with the constraint that the **list of supplied vector registers** are consecutive.

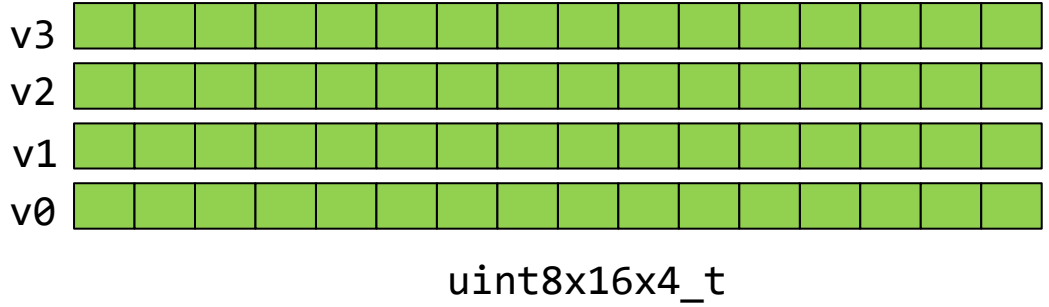
- `{v0,v1,v2,v3}` are physically consecutive.
  `{v0,v2,v1,v3}` are not physically consecutive.

Aggregate Vector Types

- The compiler fixes this for you as long as you use the aggregate vector type.

- `E.g. uint8x16x4_t`

Four consecutive vector registers of sixteen uint8



Aggregate Vector Types

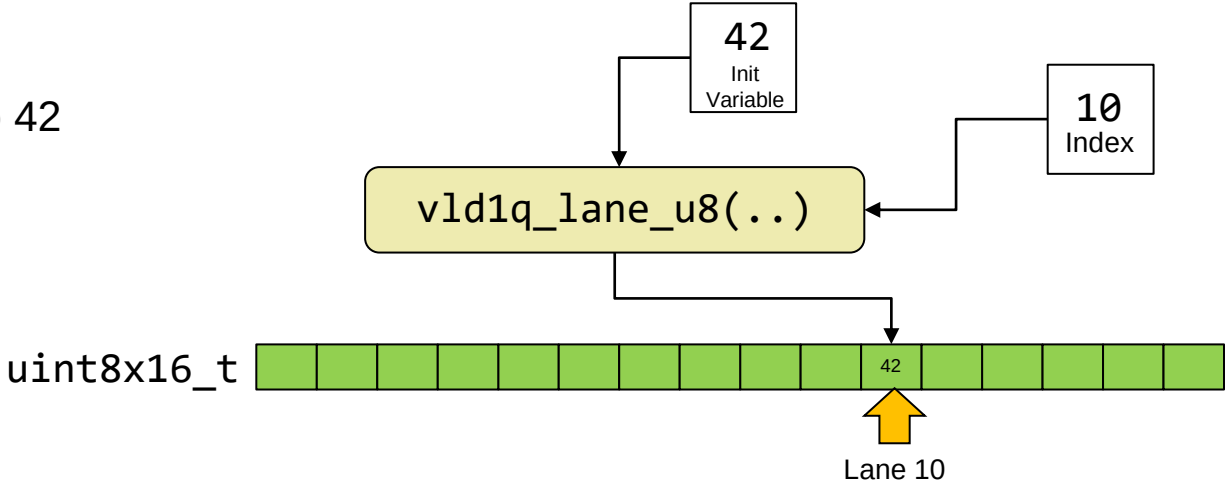
- Aggregate vector types are accessed like a structure..
  - and is kind of weird in that sense..

- Check out `arm_neon.h` if you want to see how these are accessed.
  - This can also be a nice file to search for intrinsics of interest

Initialising Vectors

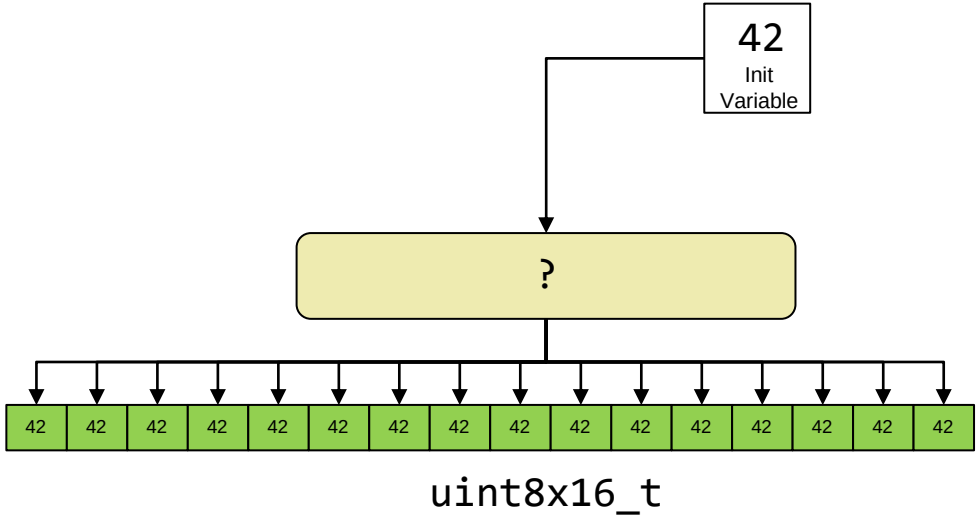
Setting a Single Lane

Set lane 10 to 42



Init example (init.c)

Initialising All Lanes of a Vector With a Constant



Vector Management

- Reverse operations exist, of course.

- E.g. **storing** a lane to RAM instead of **loading** it from RAM.

```
vst1q_lane_u8(..)
```

```
vld1q_lane_u8(..)
```

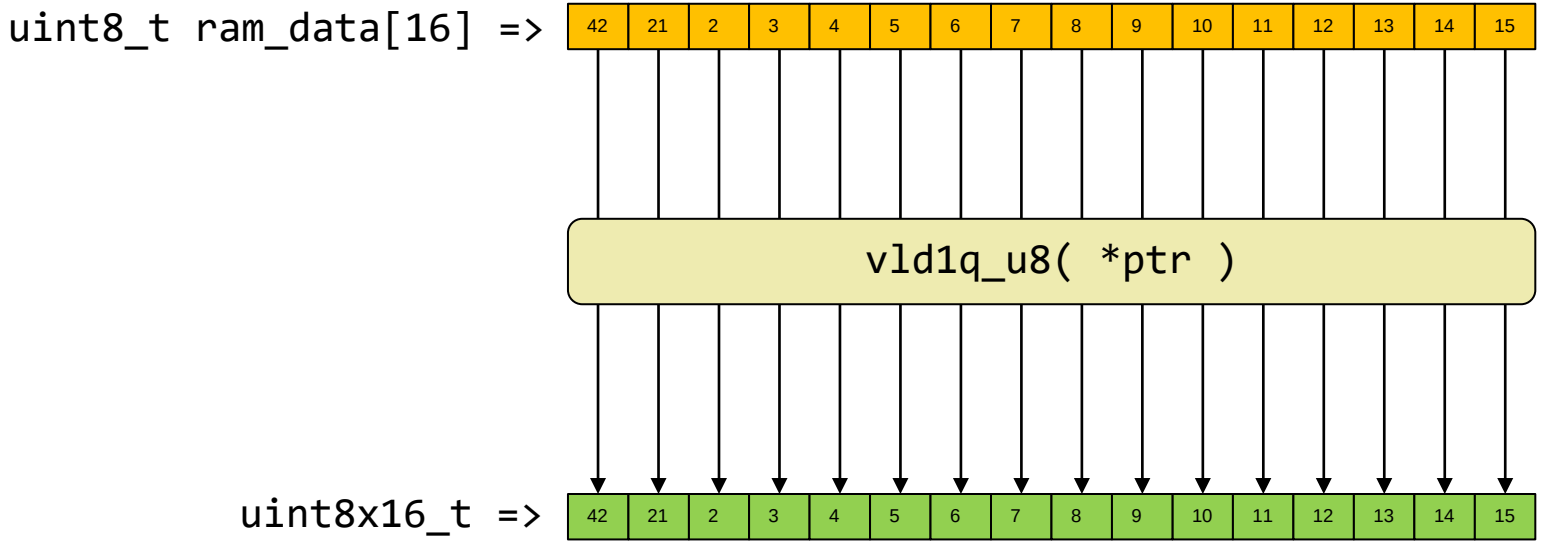
- E.g. **storing** a vector to RAM instead of **loading** it from RAM.

```
vst1q_u8(..)
```

```
vld1q_u8(..)
```

- It is also possible to store/load more than one vector to/from RAM in a single instruction!

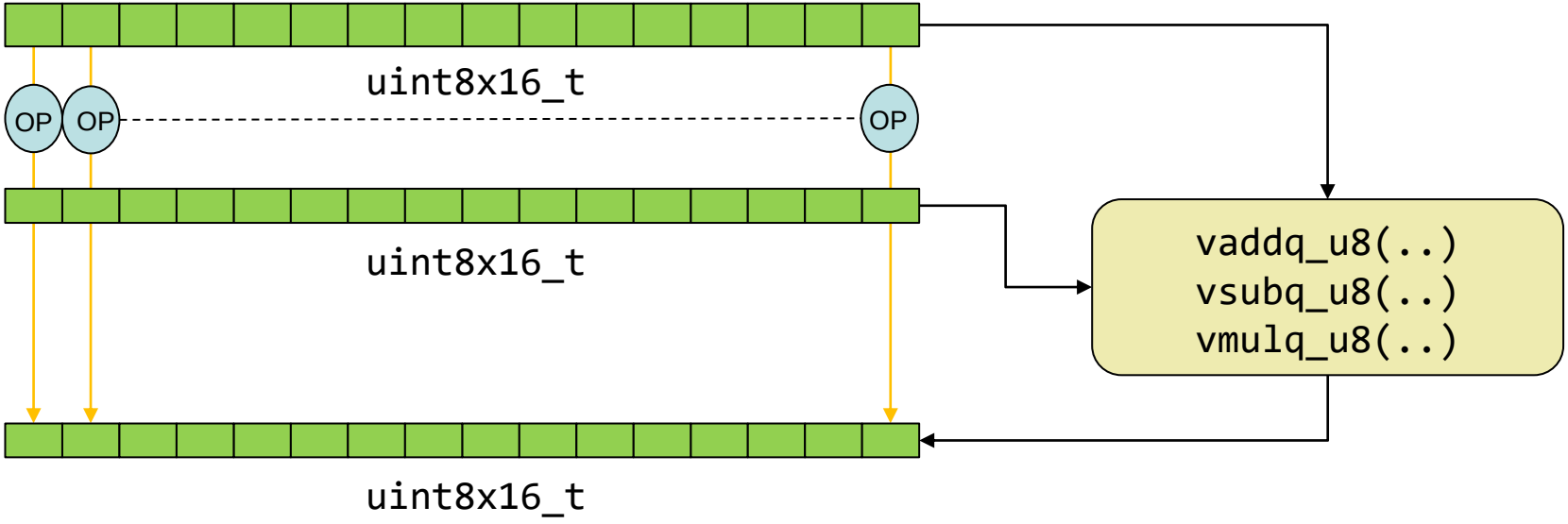
Initialising All Lanes of a Vector from RAM



Copy example (copy.c)

Arithmetic

Addition, Subtraction, Multiplication



Addition, Subtraction, Multiplication

- Input and output data types **must be the same**

- Invalid
  - vaddq_u8(uint32x4_t, float32x4_t)

- Perfectly OK
  - vaddq_u8(uint32x4_t, uint32x4_t)

```
vaddq_u8(..)
vsubq_u8(..)
vmulq_u8(..)
```

Add/sub/mult example (sub.c)

Division

- There is nothing like `vdivq_u8(..)` (!)

- `vrecpe` can find the **reciprocal** of each lane in a vector
  - Only supports floating point data types.

- **rec(x)** = $\frac{1}{x}$ such that $x * rec(x) = 1$

- Dividing by a number is the same as multiplying with the reciprocal of that number.

Other Approaches to Division

- It can be wasteful to convert to and from float32.

- Other approaches are **bitshifts**
  - a >> n is equal to $\frac{a}{2^n}$
  - a << n is equal to to $a * (2^n)$

- This will effectively **floor** your result, but there are ways around this.

Arithmetic – Finalising Notes

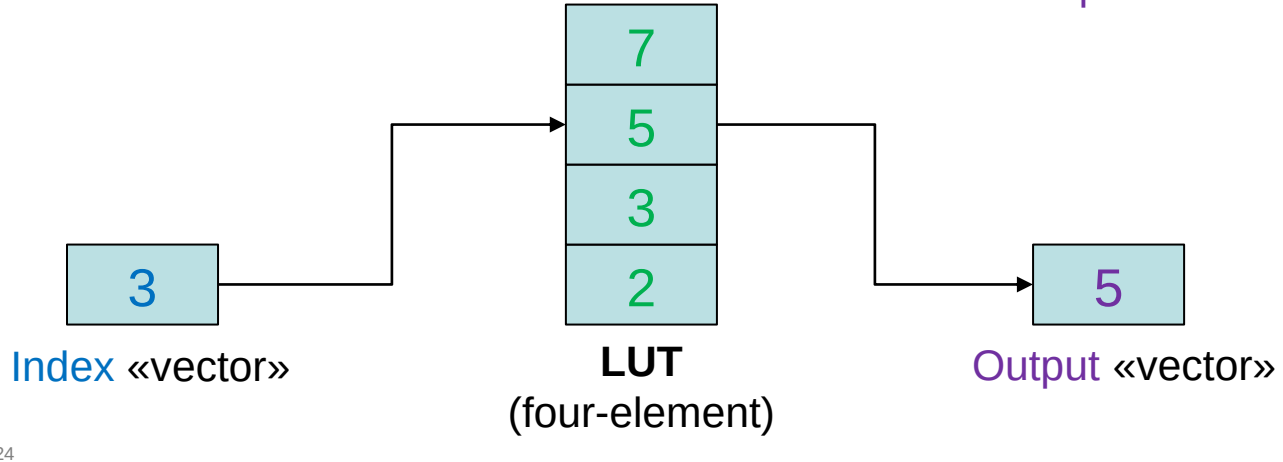
- Of course many more instructions than these very basic ones..
  - Accumulative, max/min, absolute value, square root.. and more.

- Be careful to avoid issues with sign, overflows and underflows when working with any datatype

Conversion

Lookup Tables (LUT)

- This function is useful to rearrange vectors.
- Some “index” points into a **LUT** offset that contains precomputed values



- Output stored in a vector

The index vector selects elements from the LU

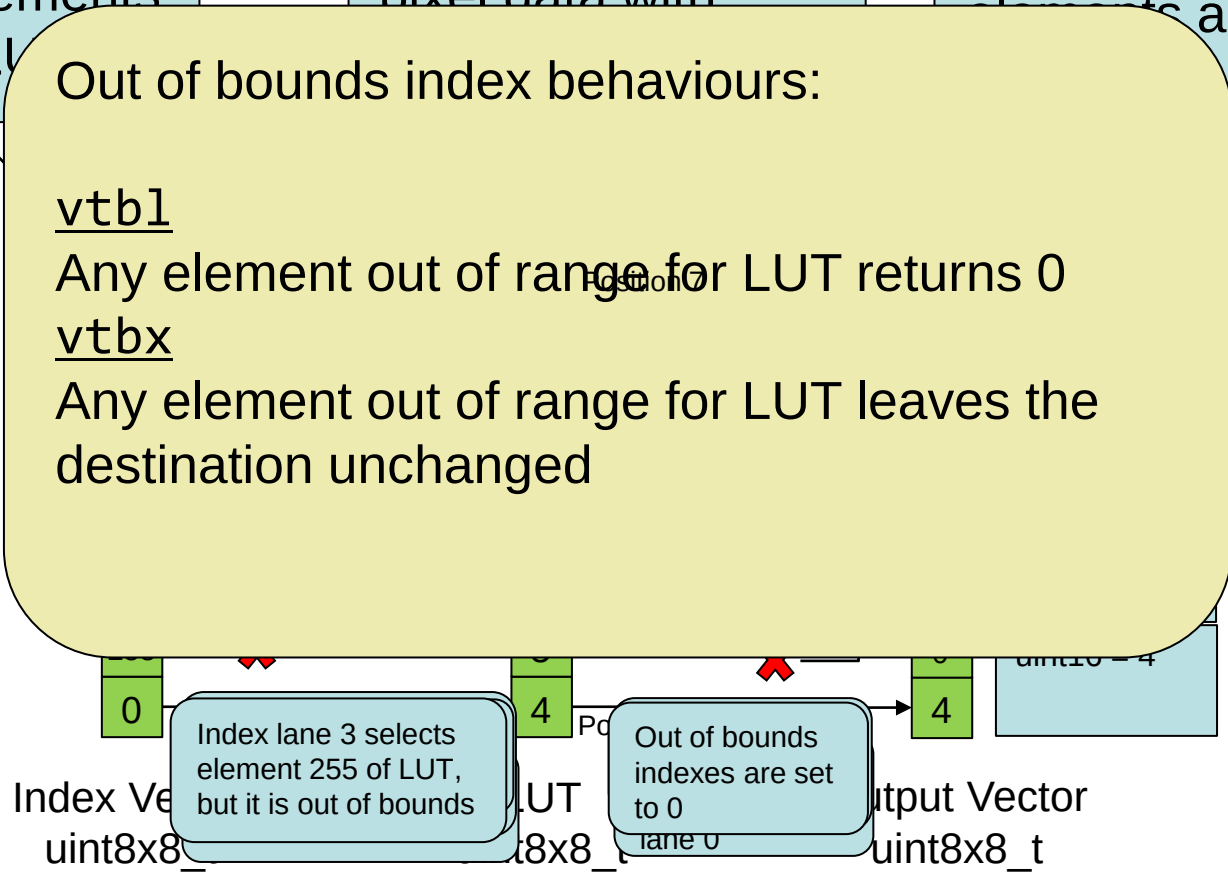
From

Imagine a vector of pixel data with

..and the selected elements are stored in vector.

Out of bounds index behaviours:

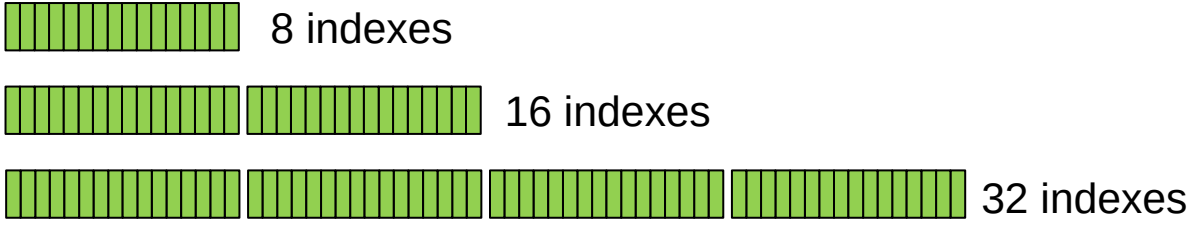
`vtbl`
Any element out of range for LUT returns 0
`vtbx`
Any element out of range for LUT leaves the destination unchanged



The LUT can be quite large.

- The LUT can be one, two or four vectors using the aggregate vector type

- `uint8x16_t`
- `uint8x16x2_t`
- `uint8x16x4_t`

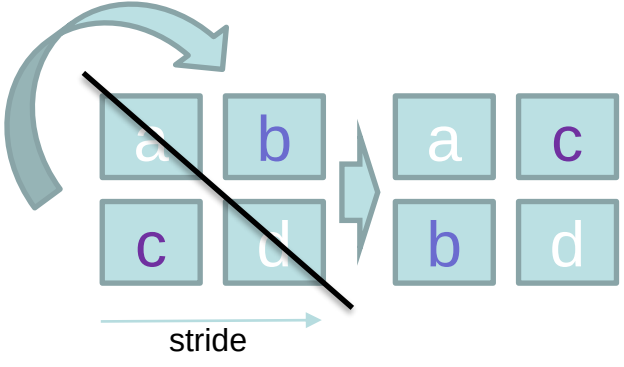


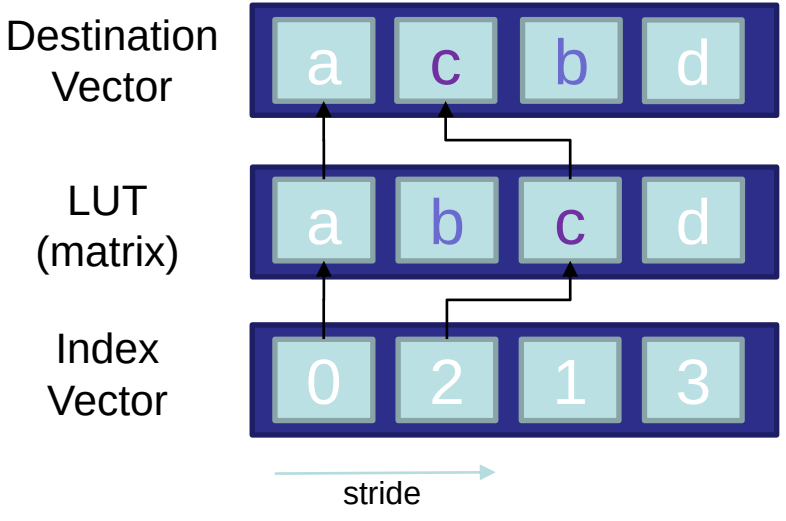
Conversion example (cnv_u8_u16.c)

Ending Notes on Conversions

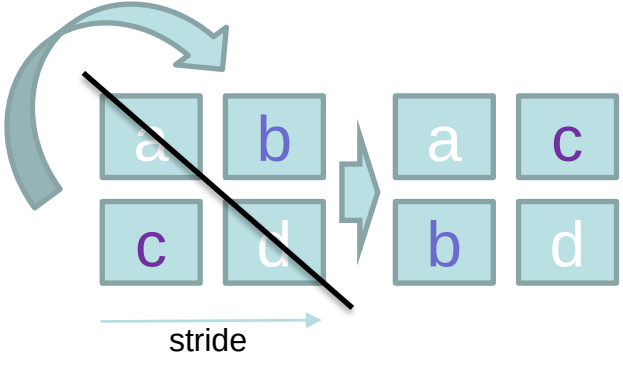
- Separate instructions exist to convert between unsigned/signed integer and floating point formats.

2x2 matrix, stride = 2

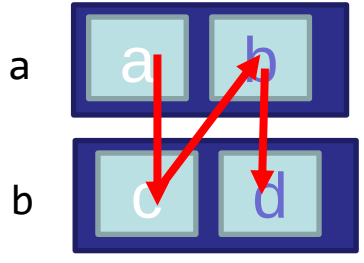




2x2 matrix with ZIP function








```
c = zip( a, b  )
```

How to do this for larger matrices, like 8x8?

Intrinsics, Inline Assembly or Assembly?

Intrinsics

```
int32x8 vector;

// Do stuff on vector

Vector = vaddq_s32(vector, vector)
```

Goes inside C functions

Inline Assembly

```
int32x8 vector;

// Do stuff on vector

__asm__(
    “vadd [v].s4, [v].s4, [v].s4”
     : : [vector] “q” (vector) : )
```

Goes inside C functions

Assembly

```
.text
.arm
.global double_elements
double_elements:
vadd.i32 v0,v0,v0
bx lr
.end
```

Goes in .s file

