

UiO : Department of Informatics
University of Oslo

Introduction to ARMv8 Neon SIMD on the Tegra Xavier

Kristoffer Robin Stokke, PhD
Huddly



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



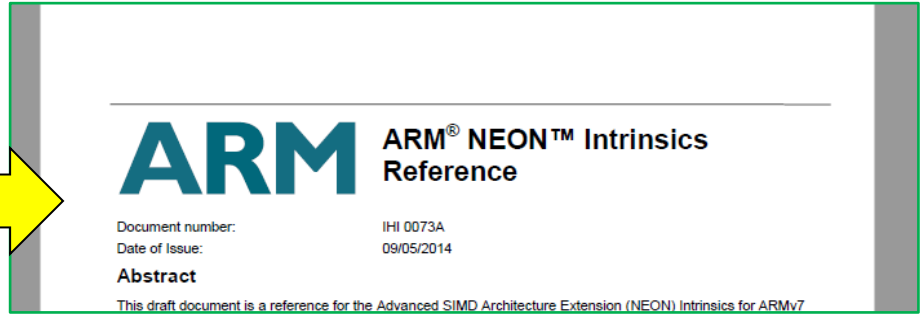
- 1 MB cartridge
- This slide: **11 MB**

Goals of Lecture

- To give you something concrete to start on
- Intro to **NEON** SIMD using 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
- Non-mandatory assignment for Friday discussion!

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] + i

Identify relevant NEON instructions

-Vector addition



Lookup instruction mnemonic



-VADD

Find intrinsic

-vaddq_u8



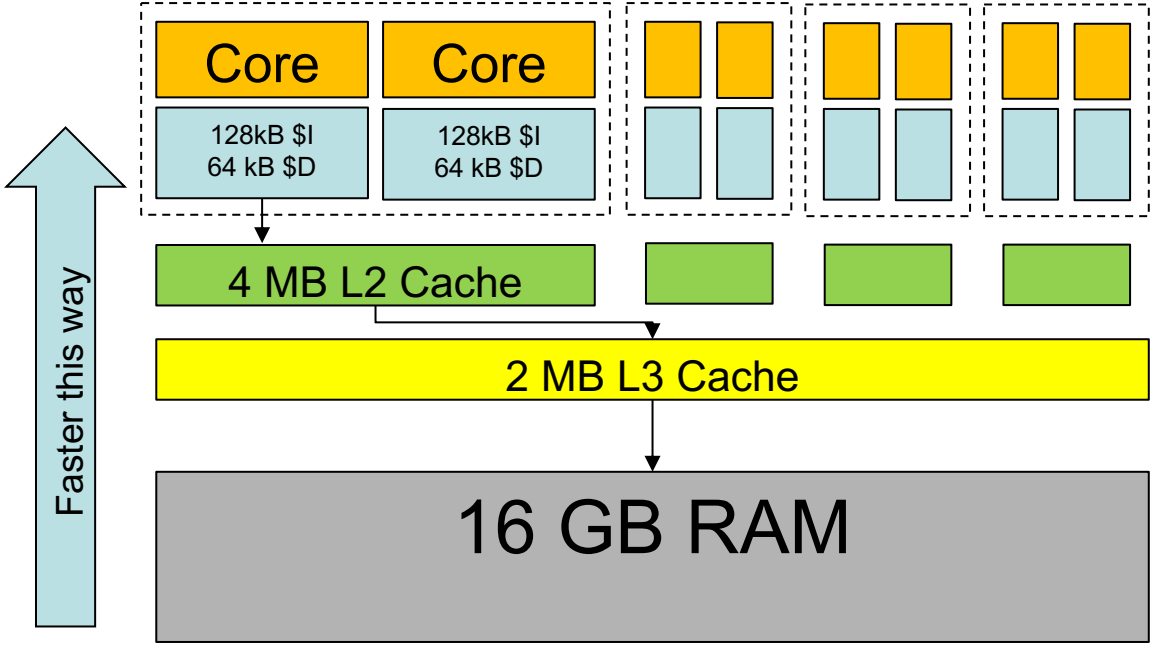
Implement



Debug



Tegra Xavier CPU Cache Hierarchy



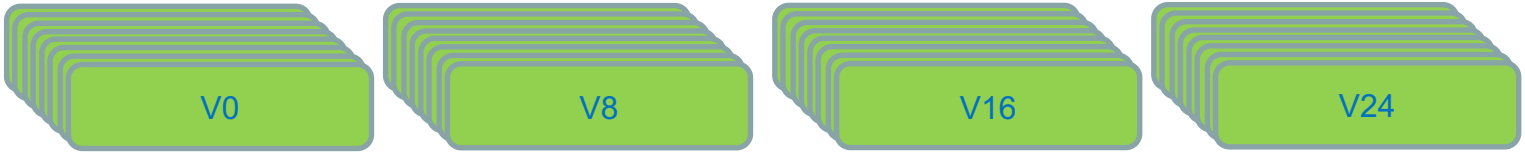
Registers: The Fastest Storage, but Size Limited

31 x 64-bit general purpose registers

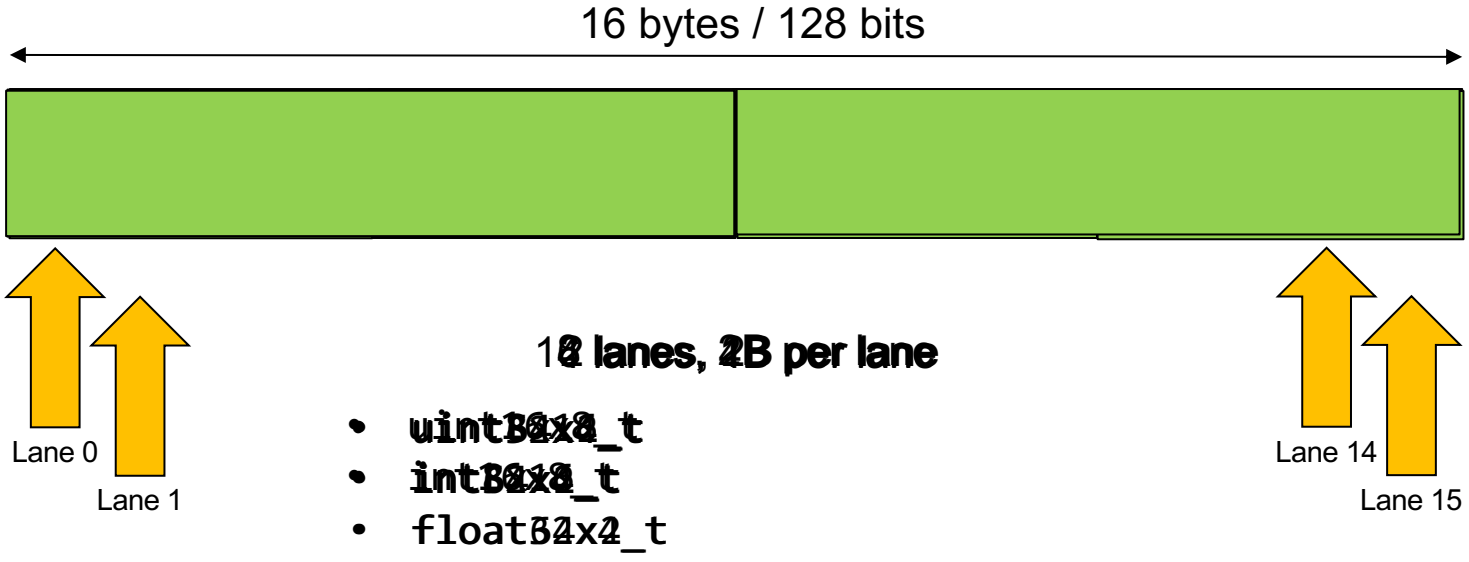


ARMv8

32 x 128-bit vector registers = 512 B of storage



The Vector Register



16 lanes, 1B per lane

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

8 lanes, 2B per lane

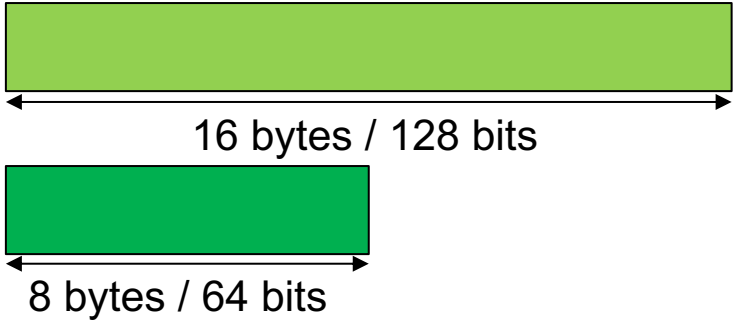
- **uint16x8_t**
- **int16x8_t**

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.
- Intrinsics exist to «convert» between them, e.g. `vcombine`

The Vector Register

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

- You must convert shorter (8, 16 bit) signed and unsigned integer primitives 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.

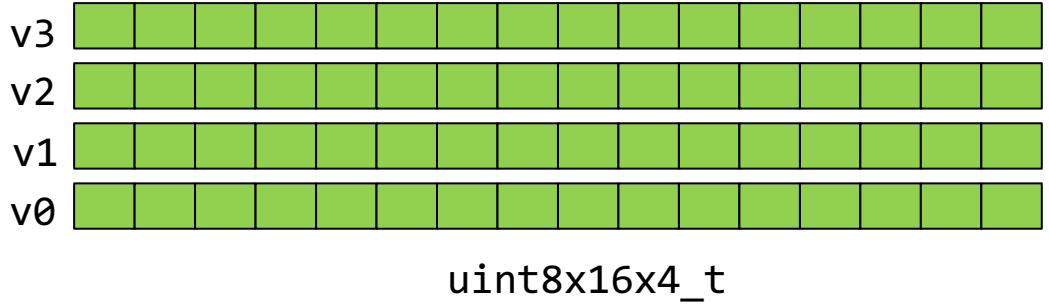
- `{v0,v1,v2,v3}` are physically consecutive.
  `{v0,v2,v1,v3}` are not physically consecutive.

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

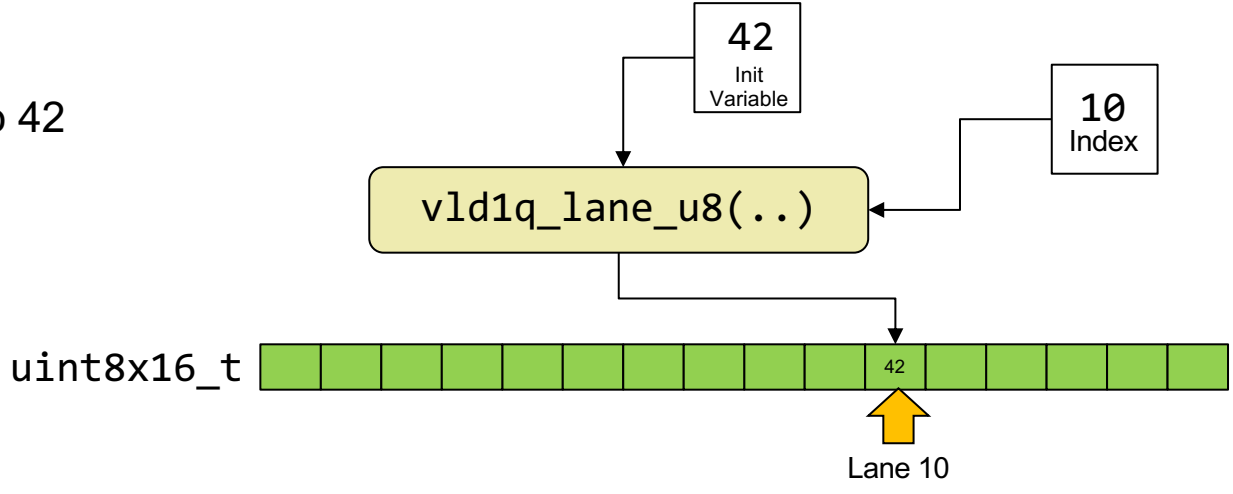
- Aggregate vector types are accessed like a structure..
  - and is kind of weird in that sense..

- 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
  - **find /usr –name arm_neon.h**

Initialising Vectors

Setting a Single Lane

Set lane 10 to 42



Init example (init.c)

Robin's GDB Cheat Sheet

- Build with **–g** to generate debug symbols in executable
- Run with **gdb <executable>**
- Breakpoint (halt execution): **break <function>**
- Breakpoint (halt execution): **break <file>:<line number>**
- Watchpoint (halt execution): **watch <variable>**
- Start program: **run <executable arguments>**
- Stack trace: **bt**
- Continue execution: **cont**
- Layout: **layout asm**
- Pretty print: **set print pretty on**
- Print as hex: **print/x <symbol>**
- Display sym at each **nexti**: **display/x <symbol>**
- Default settings: **vim ~/.gdbinit**
- Get help: **help**

Assembler (GDB) Vector Register Naming

- In assembler, the 128-bit vector registers are referred to as
  - **q0, q1, …, q31:** *one 128-bit number* (typically used for load/store)
  - **v0, v1, …, v31:** *variable data type based on suffix*

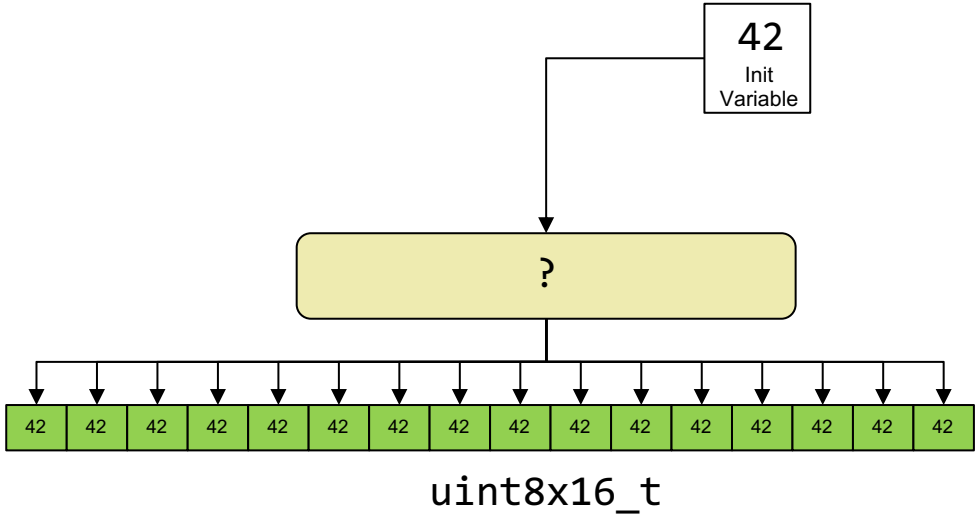
- Data width and number suffix (e.g. **print/x $v0.<SUFFIX>**):
  - **d**: two 64-bit numbers
  - **s**: four 32-bit numbers
  - **h**: eight 16-bit numbers
  - **b**: sixteen 8-bit numbers

Assembler (GDB) Vector Register Naming (cont.)

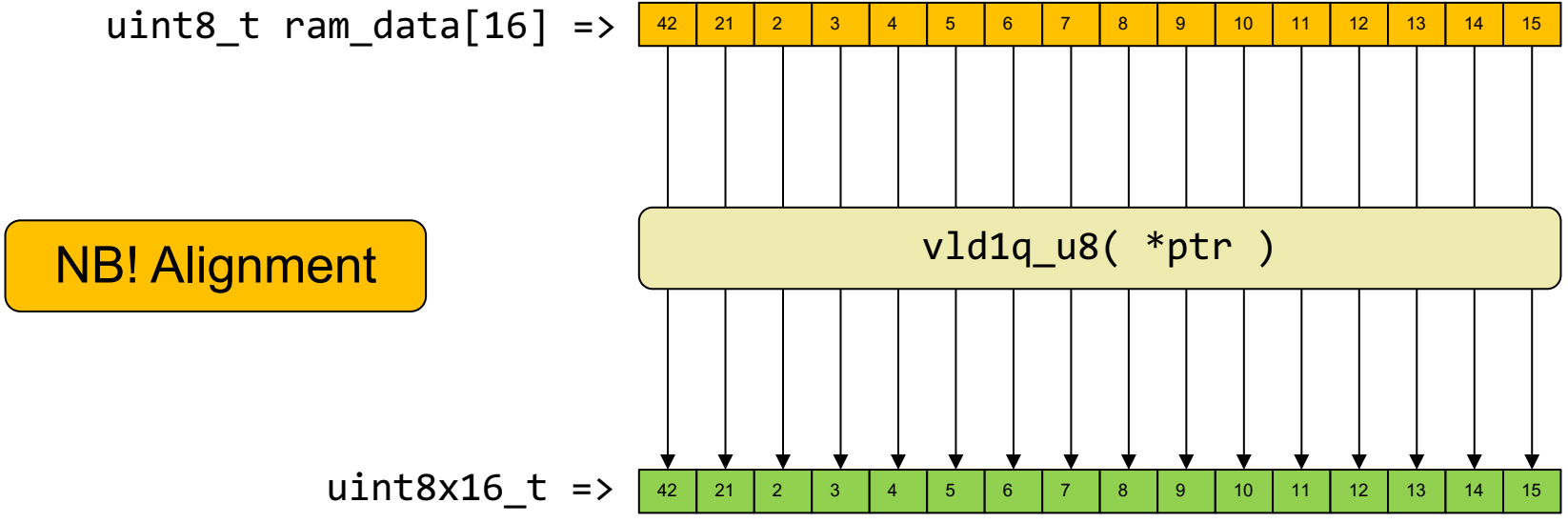
- Data type suffix (e.g. **print/x $v0.s.<SUFFIX>**)
  - **f:** floating point number
  - **u:** unsigned number
  - **s:** signed number

- Aside from this:
  - **x0, x1, …, x31:** The 64-bit general purpose registers
  - **sp:** The stack pointer

Initialising All Lanes of a Vector With a Constant



Initialising All Lanes of a Vector from RAM



NB! Alignment

Copy example (copy.c)

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.

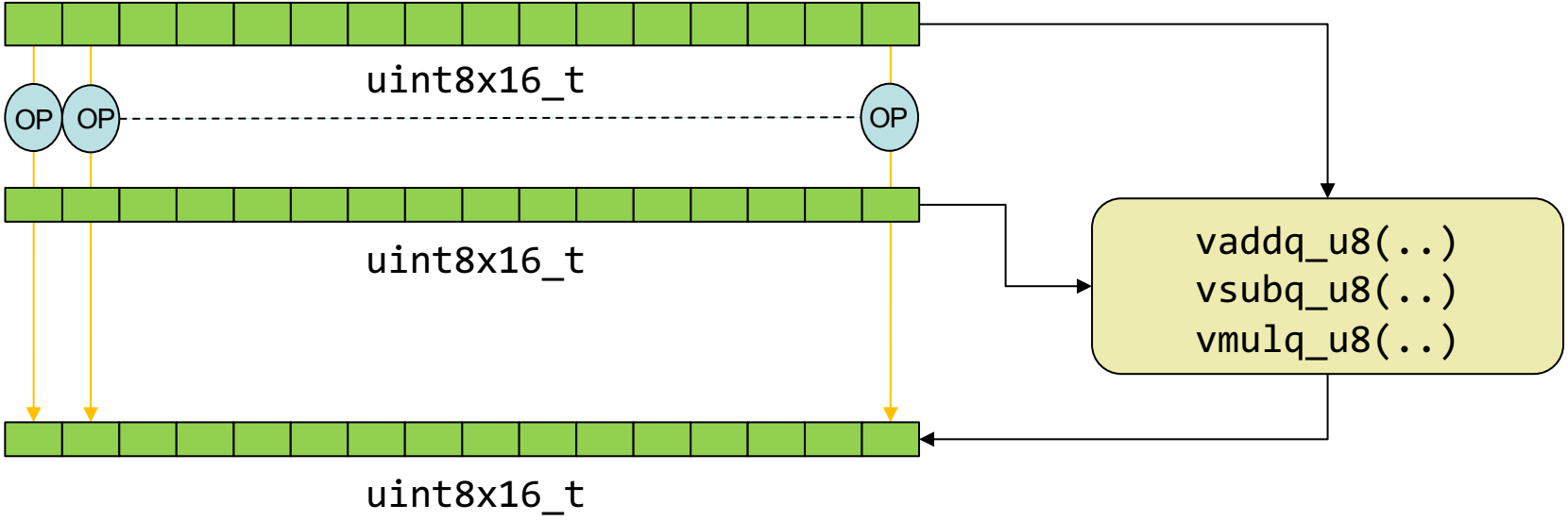
```
vst1q_u8(..)
```

```
vld1q_u8(..)
```

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

Arithmetic

Addition, Subtraction, Multiplication



Addition, Subtraction, Multiplication

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

- Invalid
  - vaddq_u8(uint8x16_t, float32x4_t)

- Perfectly OK
  - vaddq_u8(uint8x16_t, uint8x16_t)

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

Add/sub/mult example (sum.c, sub.c, mul.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 $\frac{a}{n+1}$
  - a << n is equal to to $a * (n + 1)$

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

Arithmetic – Finalising Notes

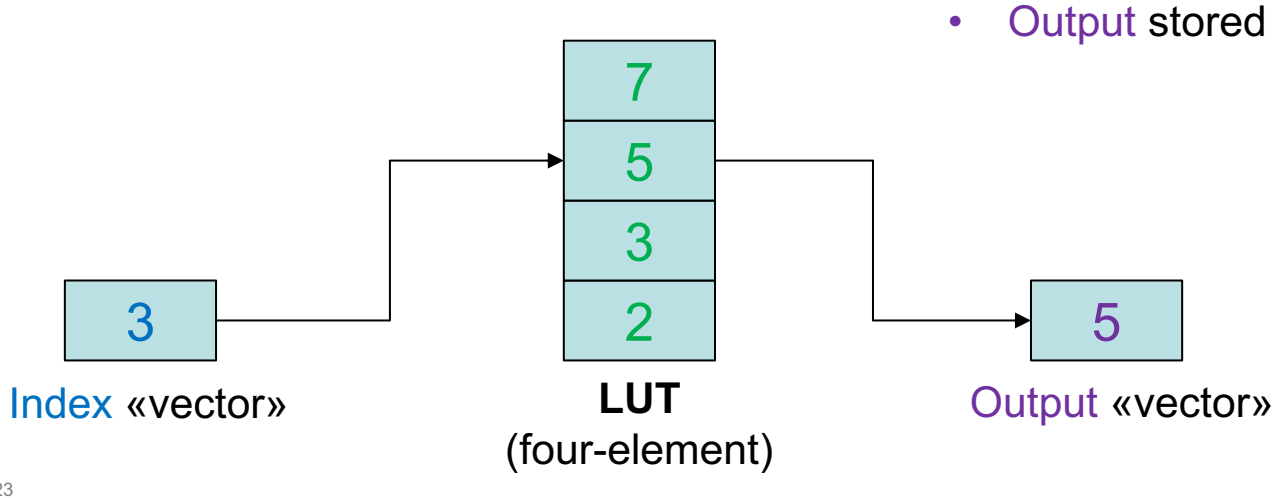
- Of course many more instructions than these very basic ones..
  - Accumulative, max/min, absolute value, square root.. and more.

- Be careful to avoid 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



The index vector selects elements from the LU

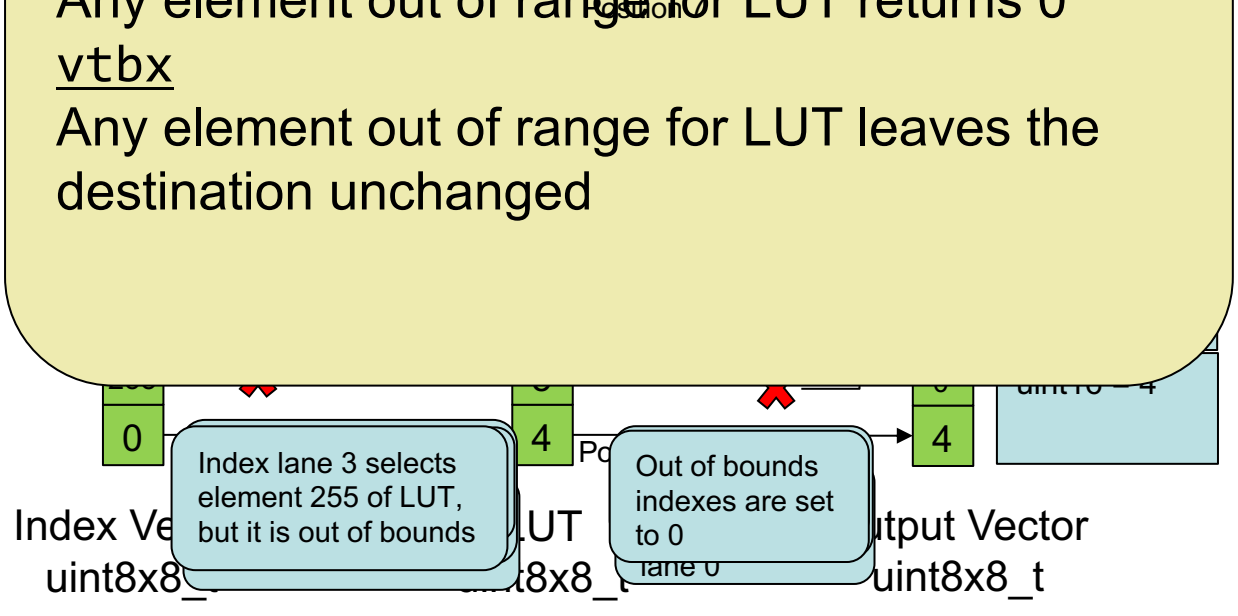
From

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`

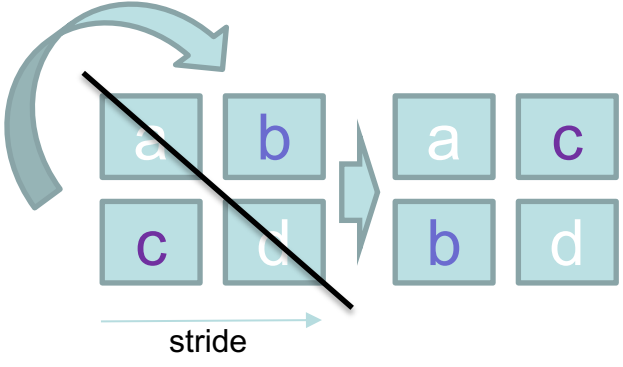


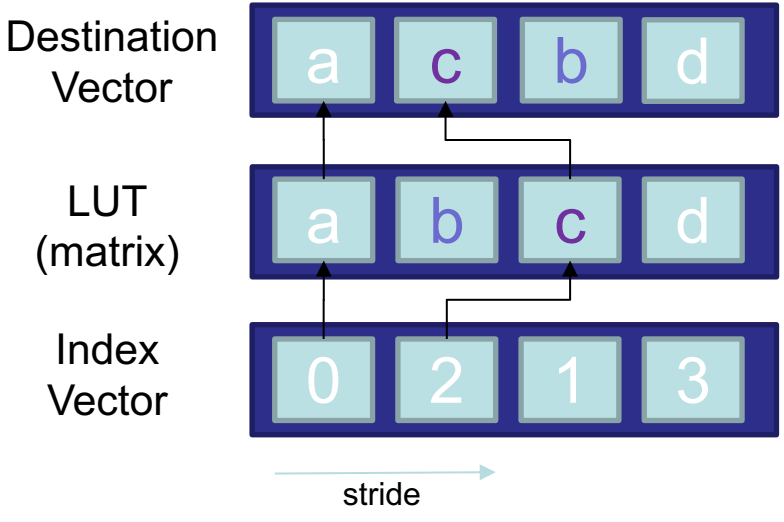
Conversion example (cnv_u8_u16.c)

Ending Notes on Conversions

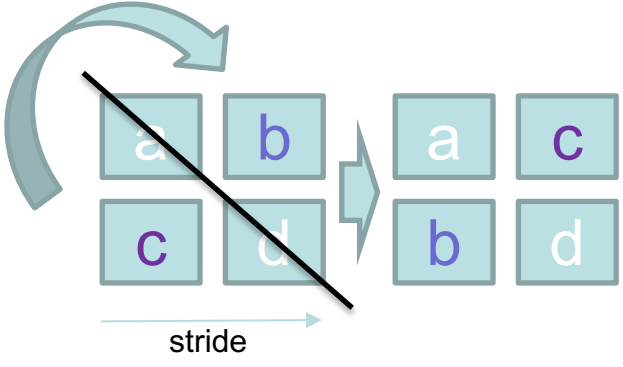
- Separate instructions exist to convert between unsigned/signed integer and floating point formats.

2x2 matrix, stride = 2

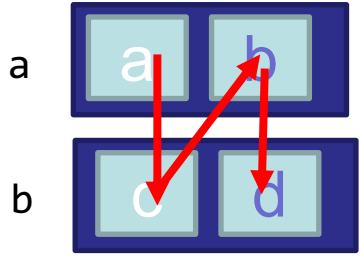




2x2 matrix with ZIP function








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

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



Intrinsics Register Spilling

- When using intrinsics, you have no control of *which vector registers you are using*

- Consequently, if you use more registers *than available* then *register spilling occurs*, where some vector registers must be evicted to some location (hopefully in the L1 cache)

- **gcc** reserves some NEON registers for internal use. This behaviour can be avoided with the **–mgeneral-regs-only** compiler switch

- Do you want to see some inline assembly just because?
- :D

Non-Mandatory Assignment

- Simple executable with some simple NEON snippets
  - Init vector to constant
  - Copy memory
  - Add, subtract, multiply
  - Convert between uint8 and uint16
  - Transpose 2x2 matrix

- We will go through how this works on friday.

- The tasks are hopefully relatively small, but some may take more effort (part 5 & maybe part 4)
- The purpose is just to get you started with NEON.
- It is more important that you have a look at things, step through the code with GDB, and look at the list of intrinsics etc than getting it right.
- Maybe you can team up and collaborate on your progress?

Part 1 – Initialisation (init.c)

- Vector initialisation calls a lane insertion 16 times just to initialise the same value to all lanes of a vector

- Find and use a single intrinsic to initialise all lanes of a vector.

Part 2 – Memory Copy (copy.c)

- Use the aggregate vector type `uint8x16x4_t` to copy `memory`, instead of four individual calls to load a single `uint8x16_t`

Part 3 – Subtraction (sub.c)

- I have deliberately broken the subtraction example.
- Step through the code with GDB, inspect the vector registers, and see if you can find the root cause!
- Can you propose any alternative that will help solve the problem for the add, sub and mult examples?

Part 4 – Conversion (cnv_u16_u8.c)

- (Harder assignment)
- We have provided sample code to convert from uint8 to uint16 using LUT table indexing
- Attempt the reverse – go from a vector of eight uint16 to eight uint8 using LUT table indexing

Part 5 – Matrix Transpose (transpose.c)

- Transpose the 2x2 matrix of uint32_t with the zip intrinsic.

Good Luck!

- You'll be fine.
- I'll be happy if you have a go at the assignment but don't spend too much time on it. If you're stuck and really want to finish, please come to me and I will try to help you.
- I'll try to hang out with you on Mattermost if you want to discuss something or otherwise ☺