ARMv8 Neon Programming

-BY KRISTOFFER ROBIN STOKKE, FLIR UAS

Goals of Lecture

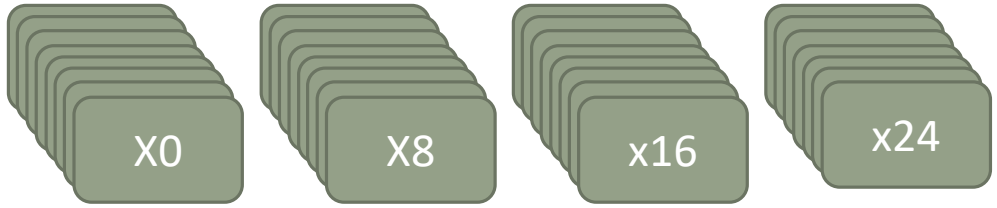
- To give you something concrete to start on
- Simple introduction to ARMv8 NEON programming environment
  - Register environment, instruction syntax
  - «Families» of instructions
  - Important for debugging, writing code and general understanding
- Programming examples
  - Intrinsics
  - Inline assembly
- Performance analysis using gprof
- Introduction to GDB debugging

Keep This Under Your Pillow

- GNU compiler intrinsics list:
  - https://gcc.gnu.org/onlinedocs/gcc-4.3.2/gcc/ARM-NEON-Intrinsics.html
- ARM Infocenter
  - infocenter.arm.com
    -> developer guides (..) -> software development -> **Cortex A series Programmer's Guide for arm8**
- This may also be useful
  - https://community.arm.com/groups/android-community/blog/2015/03/27/arm-neon-programming-quick-reference
  - https://community.arm.com/processors/b/blog/posts/coding-for-neon---part-1-load-and-stores
- Last but not least – GDB
  - You will need it

ARMv8 Registers

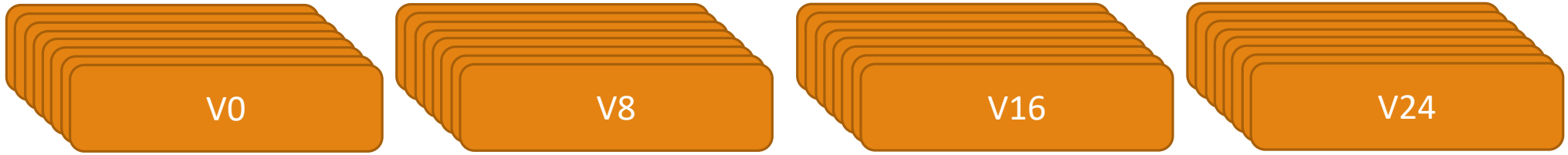
31 x 64-bit general purpose registers

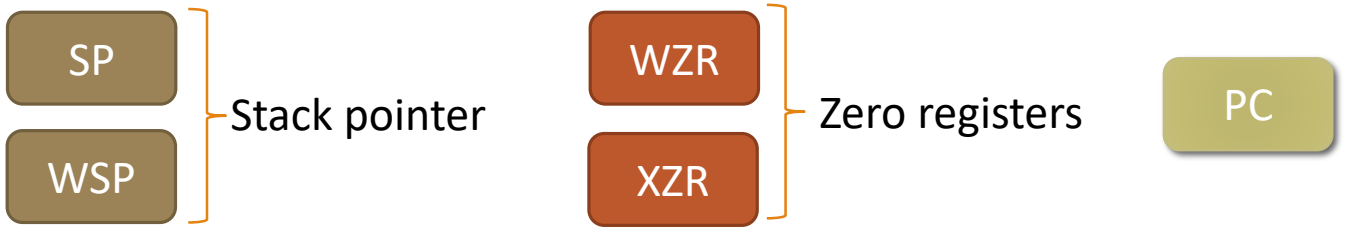


In armv7:

- Only 16 128-bit registers
- Different naming convention
  - D0-D31: 64-bit registers
  - Q0-Q15: 128-bit registers

32 x 128-bit vector registers

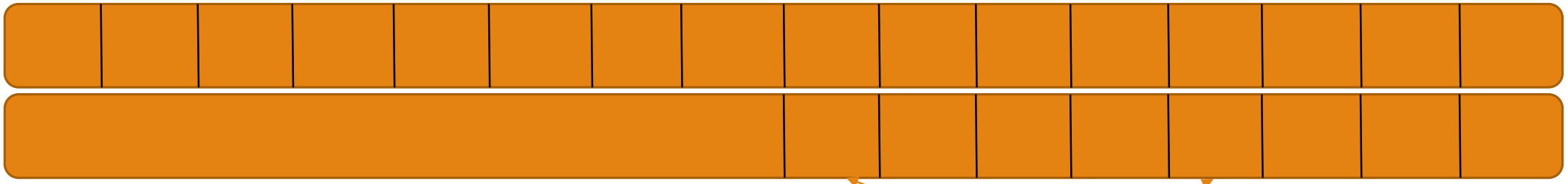




The Vector Registers V0-V31: Packing

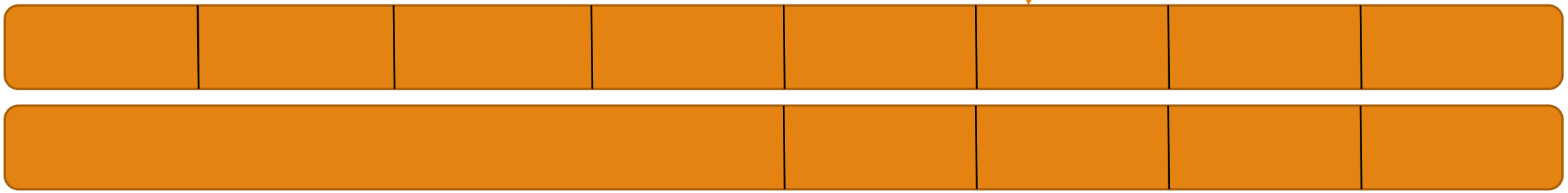
- Data in V0-V31 are **packed**, and you control **how they are packed**

Example: 16 bytes or 8 bytes



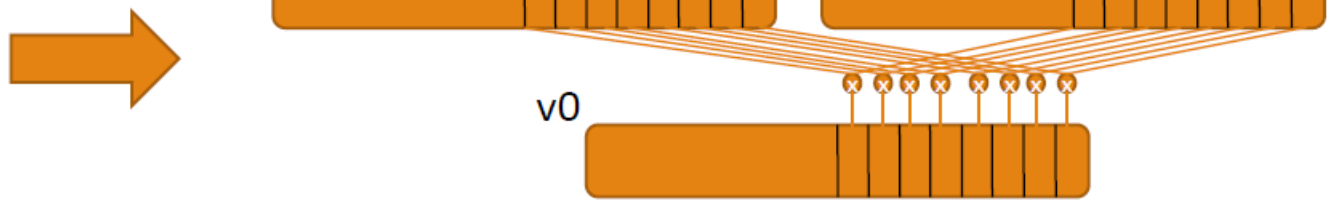
Lanes

Example: 8 half-words or 4 half-words



Example: Vector Packing

mul v0.**u8**, v0.**u8**, v1.**u8**



- v0.**8b**, v0.**16b**: 8 bytes or 16 bytes (8 bit)
- v0.**4h**, v0.**8h**: 4 half-words or 8 half-words (16 bit)
- v0.**2s**, v0.**4s**: 2 words or 4 words (32-bit)
- v0.**2d**: 2 double-words (64-bit)

Instruction Syntax

- <prefix> Represents data type (signed, unsigned, float, poly) **[S, U, F, P]**
- <op> Instruction mnemonic, for example **[mul]** or **[add]**
- <suffix> For special purpose functions, e.g. pairwise operations
- <T> Packing format. **[8B, 16B, 4H, 8H, 2S, 4S, 2D].** B=byte, H=halfword (16-bit), S=word (32-bit), D=doubledord (64-bit)

Programming With Intrinsics

- By far the most simple approach, but you *might not* be able to do everything you want
  - Some intrinsics for instructions missing
  - Assembly also needed to debug, or implement things that are not supported by intrinsics

Data Types	uint8x8_t, uint8x16_t, float32x4_t, float64x2_t
Load / Store	vld1_u8(uint8_t*), vst1_u8(uint8_t*, uint8x8_t)
Arithmetic	vadd_u8(uint8x8_t, uint8x8t), vmul_u(uint8x8_t, uint8x8t)
Conversion	vcvt_f32_u32(uint32x2_t)

++ Move Register Zip Functions Lane Functions

More in a bit!

Programming Example: Intrinsics

- Remember to include **<arm_neon.h>** in sources
- gcc **–march=armv8-a** <input file> -o <output file>

Inline Assembly

- Mostly harder than using intrinsics
  - However, gives more control (and better performance?)
- Not always straightforward to figure out what mnemonics to use
  - Tips: disassemble intrinsics and look with objdump or gdb

Operand constraints
> «m» : memory address
> «r» : general purpose register
> «f» : floating point register
> «i» : immediate
++ more

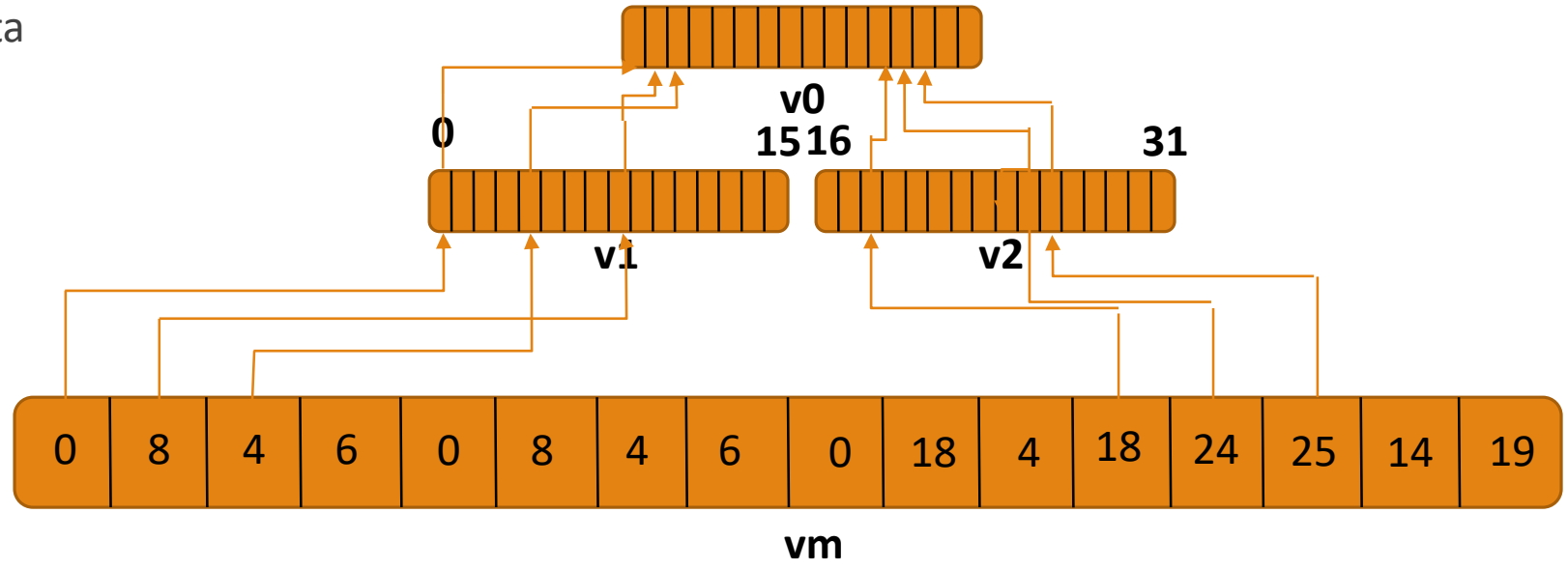
Specify dirty registers and more

```
__asm__(
        «mnemonic+operand \n\t»
        «mnemonic+operand \n\t»
        «mnemonic+operand \n\t»
        : // Output operands
        : // Input operands
        : // Dirty registers etc
)
```

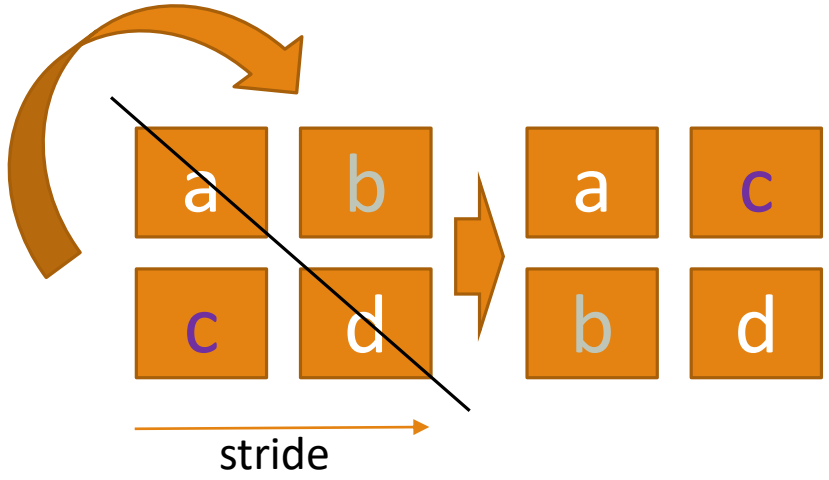
Programming Example: Inline Assembly

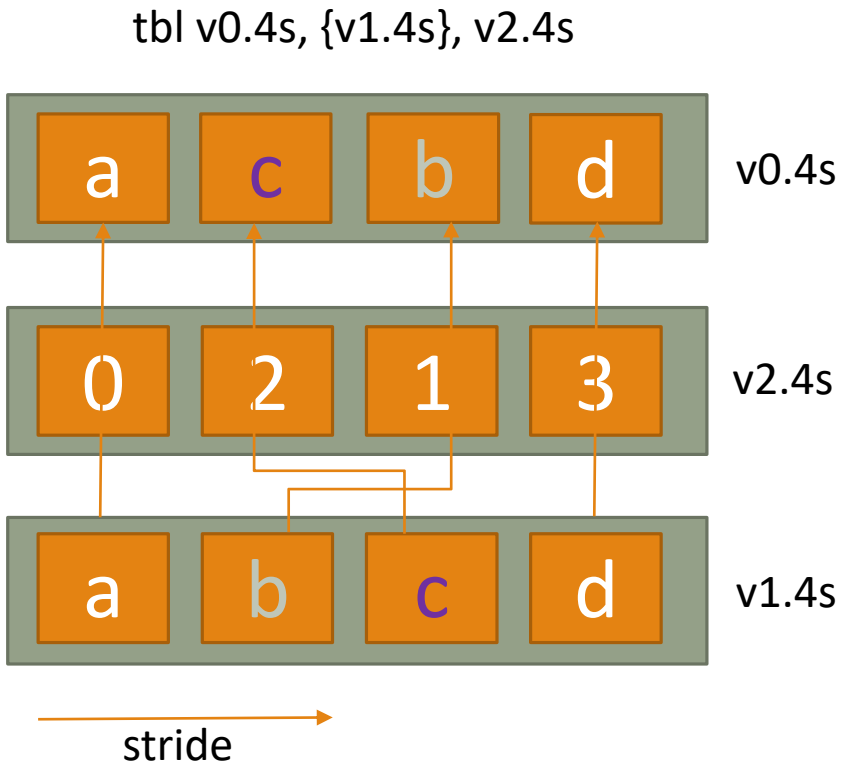
Table Lookup

- Not straightforward to use for any purpose
- Vector table lookup: **vtbl v0, {v1, v2, ..., vn}, vm**
  - **V0**: destination vector
  - **{v1, v2, ..., vn}**: source data
  - **vm**: data selector



Matrix Transpose





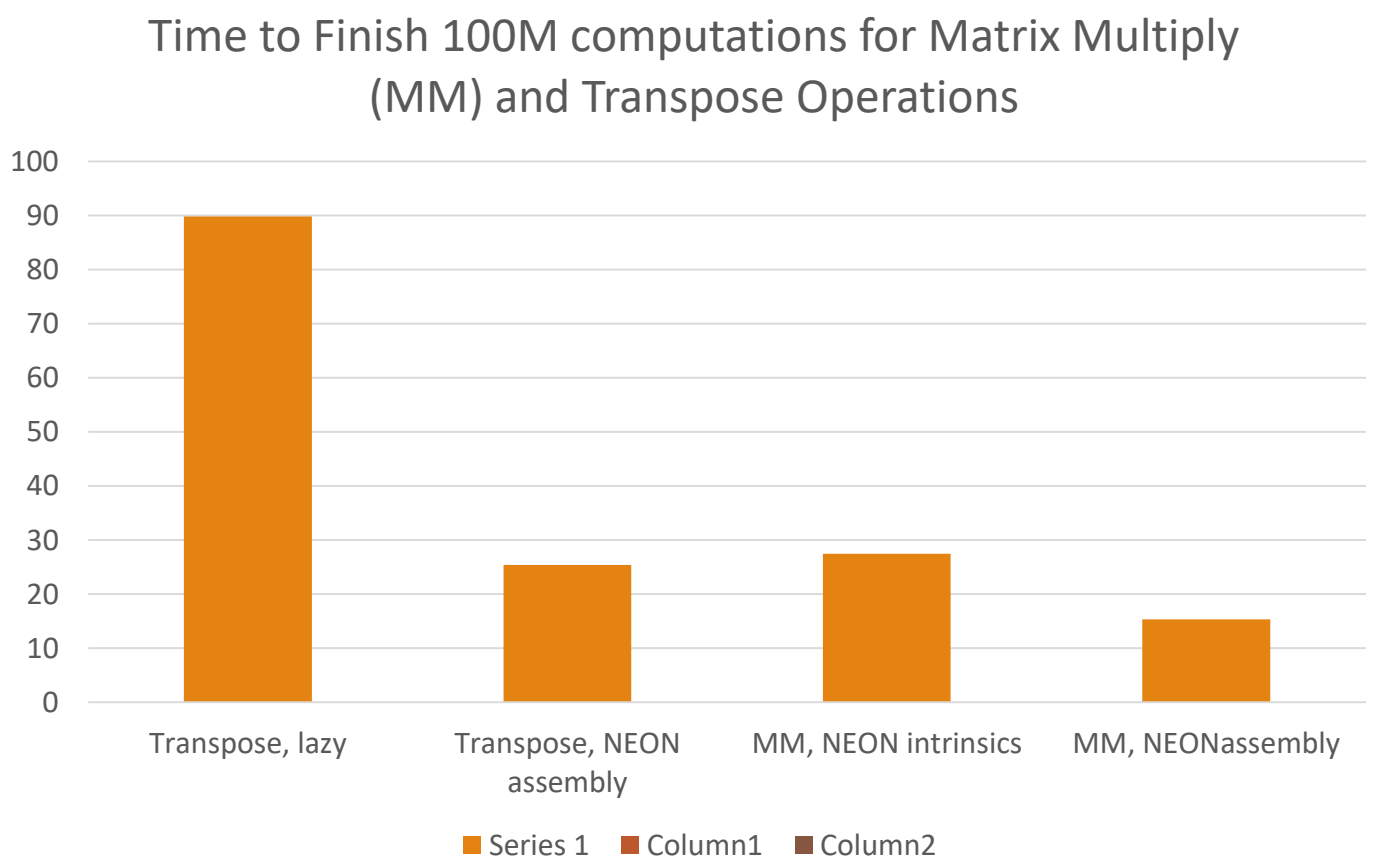
Think like this:

For each output row,

select increasing column

Code Profiling

- Compile with –pg
- Run application: **./main**
- Run **gprof ./main gmon.out**



GDB Example

- You will need to debug with GDB at times.
  - Turn on –g flag in Makefile, turn off –Ox, run make, then gdb ./<yourapplicationnamegoeshere>
- Review of useful commands
  - layout asm : get a nice-looking disassembly of current instruction location
  - b <symbol name> : breakpoint..
  - info all-registers : Complete print of processor and register state
  - p $v0 : print register v0 (also works for general purpose x0, stack pointer etc)
  - display $v0 : At every step, display the value of v0
  - si: Step instruction
  - + breakpoint on conditionals is useful when debugging loops

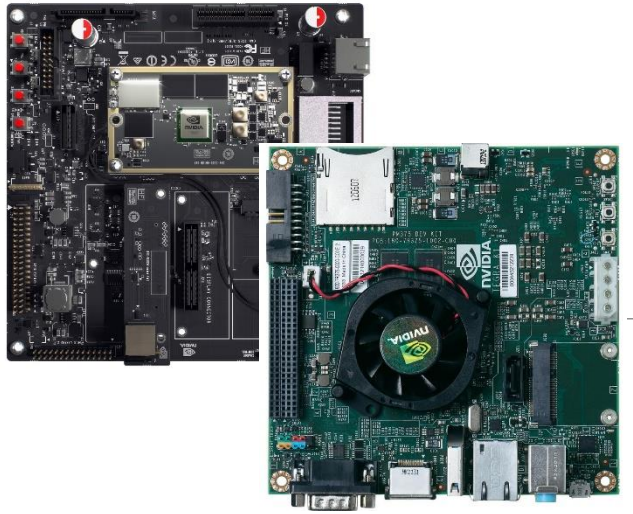
Tips

- Build functions to print out macroblocks from vector registers and memory
- Start small – test out independent parts of the code that are easy to verify
- When in trouble, step through the code, display the relevant registers, verify with output you know is working
- Many things to investigate
  - Single versus double precision?
  - Different, possibly more ways to implement e.g. transpose?
  - Re-using vector registers across different functional blocks?
  - ..but stick to what the assignment says

Good Luck!

- You’re going to need it ☺

ARMv7 vs. ARMv8



- Armv8 uses the same mnemonics as for general purpose registers
  - E.g., in ARMv7, «mul, r0, r0, r1» (**normal**) and «vmul d0, d0, d1» (**SIMD**)
  - In ARMv8: «mul x0, x0, x1» (**normal**) and «mul v0, v0, v1» (**SIMD**)
  - Simplifies life, but **take care to use correct operands**
- ARMv8 has twice as many 128-bit registers
  - 32 128-bit registers, vs 16 128-bit registers for ARMv7
- Different instruction syntax