SIMD Assembly Tutorial: ARM NEON



Motivation

- SIMD critical for video performance
 - It's cheap for CPUs to add wider ALUs
 - It's cheap parallelism (no locking/synchronization)
- Even if you won't write the asm, we need to design code that can be vectorized
 - Need to understand what's possible
- Why NEON?
 - Slowest architecture that's likely to be viable
 - Much nicer instruction set than x86



Intrinsics, Inline, or External?

Intrinsics

- C compiler does register allocation, manages stack, manages instruction scheduling etc.
- Not all instructions available! (e.g., 4-register loads)
- Compilers are bad at registers

Inline

- C compiler manages stack
- Limited portability (basically gcc/clang)

External

Good portability: ARM has a well-defined ABI



General ARM Assembly

Basics

- Three-address machine
 - add r1, r2, r3; r1 = r2 + r3
- 16 general-purpose registers
 - r0-r3: function parameters (not saved)
 - r4-r11: general (callee-saved)
 - r12 (ip): "intra-procedure call scratch" (not preserved)
 - r13 (sp): stack pointer
 - r14 (Ir): link register (return address, not saved)
 - r15 (pc): program counter



Instruction Format

- General ARM instructions all 32 bits
 - "Thumb" mode support 16-bit instructions, but we won't discuss them here
- Many ARM instructions take a "flexible operand2"
 - #<imm8m> (immediate)
 - Any constant that can be formed by right-rotating an 8-bit value by an even number of bits
 - In MOV, can also use bitwise complement (MVN)
 - r0 (plain register)
 - r0, LSL #0 (register shifted by a constant)
 - LSL, LSR, ASR, ROR available
 - r0, LSL r1 (register shifted by a register)



Procedure Call ABI

- Function entry: STMFD sp!, {r4-r6, lr}
 - "Store Multiple Full-Descending"
 - Equivalent to STMDB (decrement before)
 - *sp*: store to stack
 - !: update stack pointer
 - {r4-r6, lr}: Stores r4, r5, r6, r14 (lr)
 - Always save even # of registers to preserve 8-byte stack alignment: use r12 if you need an extra one
 - Order of list not preserved!
 - Always stored in r0...r15 order
- Extra parameters: LDR r4, [sp, #16]
 - Must add offset for registers we saved



Procedure Call ABI

- Function exit: LDMFD sp!, {r4-r6,pc}
 - "Load Multiple Full-Descending"
 - Equivalent to LDMIA (increment after)
 - sp: load from stack
 - !: update stack pointer
 - {r4-r6,pc}: Load r4, r6, r6, r15 (pc)
 - Loading to PC is a branch
 - Restores stack and returns in one instruction



Register Allocation

- You can always modify r0-r3
 - Even if you don't take 4 arguments
- You can use r12 for free
 - First choice if r0-r3 are not enough
- If you must save one register, pick r14 (Ir)
 - Gets you restore & return for free
- Then start using r4-r11

Flow Control

- Instructions only set flags if requested
 - sub r1, r2, r3; no flags are updated
 - subs r1, r2, r3; flags are set
 - {cmp,tst,teq} r2, r3; => {subs,ands,eors} w/o dst
- Most ARM instructions can be made conditional
 - ADDLE r1, r2, r3; Execute add if CMP was <=
- This includes branching: BLE <label>
 - CMP and B can issue in the same cycle
 - Mis-prediction is 13 cycles on a Cortex A8
- Or function returns: MOVLE pc, Ir
- But not NEON instructions!



Condition codes

- EQ: Equal
- NE: Unequal
- VS: Overflow
- VC: No overflow
- Unsigned:
 - HS: >=, HI: >, LS: <=, LO: <
- Signed
 - GE: >=, GT: >, LE: <=, LT: <, PL: >= 0, MI: < 0



NEON Assembly



Execution Pipeline

- Think of NEON like a co-processor
 - NEON instructions execute in their own 10-stage pipeline
 - ARM can dispatch 2 NEON instructions per cycle
 - 16-entry instruction queue holds NEON instructions until they can enter the pipeline
 - 12-entry data queue for ARM register values
 - Saves the value of the register at the time the instruction was dispatched



What this means

- ARM → NEON register transfer is fast
- NEON → ARM register transfer is slow
 - Minimum 20 cycles on A8, as little as 4 on A9
- The ARM side won't stall until the NEON queue fills
 - Can dispatch a bunch of NEON instructions, then go on doing other work while NEON catches up
- NEON instructions will physically execute much later than they appear to in the code
 - If one modifies a cache line the other needs, the ARM side stalls until the NEON side catches up



NEON Registers

- 32 64-bit ("doubleword") registers: d0-d31
- 16 128-bit ("quadword") registers: q0-q15
- qN is aliased to d(2N), d(2N+1)
 - e.g., q0 == d0, d1
- q4-q7 are callee-saved
 - VPUSH {q4-q7}
 - VPOP {q4-q7}

Datatypes

- Instructions specify what's in the vectors
 - VADD.I32 q1, q2, q3
 - 4x 32-bit integer add
 - VQADD.S32 q1, q2, q3
 - 4x 32-bit integer add with signed saturation
 - VQADD.U32 q1, q2, q3
 - 4x 32-bit integer add with unsigned saturation
- It's okay to re-interpret a register's contents from instruction to instruction
- 8, 16, 32, 64 bits available (not 128)
- Also F32, F64 for float, but don't need that for video



Promoting and Demoting

- VMOVL.S32 q0, d0
 - 2x 32-bit signed promotion to 64-bit
- VMOVN.132 d0, q0
 - 4x 32-bit narrow to 16-bit
- VQMOVN.U32 d0, q0
 - 4x unsigned 32-bit narrow to 16-bit with saturation
- VQMOVUN.S32 d0, q0
 - 4x 32-bit narrow signed data to 16-bit with unsigned saturation (negative values go to 0)
- Datatype always corresponds to the source
- Can't promote past 64-bit, demote to less than 8-bit



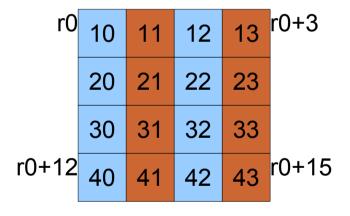
Promoting and Demoting

- Some instructions can promote/demote as part of the operation
- VADDL.S32 q0, d0, d1 (Long variant)
 - 2x signed 32-bit promotion to 64-bit and add
- VADDW.S32 q0, q0, d2 (Wide variant)
 - Promotes d2 to S64 and does 2x 64-bit adds with q0
- VADDHN.S32 d0, q0, q1
 - 4x signed 32-bit add, take the high half, and narrow to 16-bit
- VQRSHRUN.S16 d0, q0, #4
 - -d0[i] = UnsignedSat((q0[i] + 8) >> 4)



Loading and Storing Data

- "Structured" load/stores (de)interleave for free
- Syntax: VLD2.8 {d0,d1}, [r0]
 - RAM:



Registers:

42	40	32	30	22	20	12	10	d0
43	41	33	31	23	21	13	11	d1
MSB LSB						SB	•	

- Stride of 1, 2, 3(!), or 4
- 13 r₀₊₃ 8, 16, 32, or 64 bits
 - VLD<n>.64 == VLD1
 - 1, 2, 3, or 4 D registers
 - Consecutive: {d1,d2,d3,d4}
 - Or every other: {d1,d3,d5,d7}



Loading and Storing Data

- Transfer at most 128 bits per cycle
 - +1 cycle for VLD1x3, VLD3x3, VLD4x4, VST2x4, VST3x3, VST4x4
- Unaligned load/stores cost one more cycle
 - Can specify alignment: VLD2.8 {d0,d1}, [r0@128]
 - Saves one cycle if alignment large enough
 - @64 for 1-register load/stores, and 3-register stores
 - @128 for 2- or 4-register load/stores
 - Scheduling is static
 - Must specify alignment to get benefit



Non-Vector Load/Stores

- Single-lane: VLD1.8 {d0[0],d1[0]}, [r0]
 - Load/store one element of each vector
 - Must use same element in every register
 - Costs 1 extra cycle
- All-lane: VLD1.8 {d0[],d1[]}, [r0]
 - Load one element and copy to all elements in each register
 - VLD1 does not support 3- or 4-register versions
 - No extra cost compared to vector load



Addressing Modes

- Very limited address calculations
 - VLD1.64 (d0), [r0]
 - VLD1.64 (d0), [r0]!
 - Adds size of transfer to r0 after transfer
 - VLD1.64 (d0), [r0], r1
 - Adds r1 to r0 after transfer



Other Load/Store Instructions

VLDM/VSTM

- Only IA/DB variants supported (DB requires writeback)
- List can have at most 16 D registers
- -1 + (N+1)/2 cycles
- VLDR.64/VSTR.64
 - Loads/stores one D register in 2 cycles
 - 25% peak throughput!
 - VLDR.64 d0, [r0, #128]; +/- 1020, multiple of 4
 - VLDR.64 d0, [r0, #-8]!; => VLDMDB r0!, {d0}
 - VLDR.64 d0, [r0], #8; => VLDMIA r0!, {d0}



Constants

- Constant tables
 - Load address of table with ADR psuedoinstruction: ADR r0, <label>
 - PC-relative for Position Independent Code
 - Limited range: +/- 1020, multiple of 4 bytes
- VMOV/VMVN.
 datatype> q0, #<vimm>
 Forms for <vimm>

18	I16 I32		l64			
0xXY	0x00XY	0x000000XY	0xGGHHJJKKLLMMNNPP			
	0xXY00	0x0000XY00	(GGPP must be 0x00 or 0xFF)			
		0x00XY0000				
		0xXY000000				



Let's write some code



xcorr kernel

Motivated by actual function in CELT/Opus

```
void xcorr kernel(int32 t sum[4],
  const int16 t *x, const int16 t *y,
  int len) {
  int i;
  for (i = 0; i < 4; i++) {
    int j;
    sum[i] = 0;
    for (j = 0; j < len; j++) {
      sum[i] += x[j]*y[i + j];
```



Register Allocation

- 4 parameters
 - r0 = sum
 - r1 = x
 - r2 = y
 - -r3 = len
- We'll do all 4 iterations of outer loop simultaneously, so we only need one pass
- Can operate entirely in-place
 - Increment r1, r2 on each iteration
 - Decrement r3 as a loop counter

Prolog

```
AREA | .text|, CODE, READONLY
EXPORT xcorr kernel
xcorr kernel PROC
  ; Initialize sum[0...3]
  VMOV.I32 q0, #0
  ; Test for len <= 0 and exit early
  CMP r3, #0
  BLE xcorr kernel done
  ; Assume len > 0 and load y[0...3]
  VLD1.16 {d3}, [r2]!
  ; If len \leq 4, go to the end
  SUBS r3, r3, #4
  BLE xcorr kernel process4 done
```



Main Loop: Attempt #1

```
xcorr kernel process4
 SUBS r3, r3, #4
 ; Load y[4...7]
 VLD1.16 {d4}, [r2]!
 VLD1.16 {d2}, [r1]!
  ; Pull elements \{i...i+3\} from (d3,d4)
 VEXT.16 d5, d3, d4, #1
 VEXT.16 d6, d3, d4, #2
 VEXT.16 d7, d3, d4, #3
  ; VMLAL = Vector MuLtiply and Accumulate Long
 VMLAL.S16 q0, d3, d2[0]
 VMLAL.S16 q0, d5, d2[1]
 VMLAL.S16 q0, d6, d2[2]
 VMLAL.S16 q0, d7, d2[3]
 VMOV d3, d4
 BGE xcorr kernel process4
```



How Fast Is It?

- Assume a Cortex A8
 - The A8 has a better NEON unit than A9
 - A9 chips generally faster anyway, though
 - A12, A15 dual-issue, fully out-of-order
 - Optimizing for A8 won't slow things down elsewhere
- Instruction timing information:
 - Cortex A8 Technical Reference Manual
 - Cortex A9 Media Processing Engine Technical Reference Manual



Step1: ARM Dispatch

- 12 total instructions
 - 2 ARM
 - 10 NEON
- Can dual-issue all of them
 - 6 cycle minimum per loop iteration



Step 2: NEON Cycle Count

```
xcorr kernel process4
  SUBS r3, r3, #1
 VLD1.16 {d4}, [r2]!
                        ; 2 cycles
 VLD1.16 {d2}, [r1]!; 2 cycles
 VEXT.16 d5, d3, d4, #1 ; 1 cycle
 VEXT.16 d6, d3, d4, #2 ; 1 cycle
 VEXT.16 d7, d3, d4, #3 ; 1 cycle
 VMLAL.S16 q0, d3, d2[0]; 1 cycle
 VMLAL.S16 q0, d5, d2[1] ; 1 cycle
 VMLAL.S16 q0, d6, d2[2]; 1 cycle
 VMLAL.S16 q0, d7, d2[3] ; 1 cycle
 VMOV d3, d4
                       ; 1 cycle
 BGE xcorr kernel process4
```



Dual-Issue

- Cortex A8 can dual-issue "load/store/byte permute" instructions with "data processing" (arithmetic)
 - First cycle can dual-issue with previous instruction
 - Last cycle can dual-issue with next instruction
- A9's NEON unit is single-issue, in-order
 - Assume we have an A8
 - But be work-efficient
 - Don't increase instruction cycles just to dual-issue



Step 2: NEON Cycle Count

```
xcorr kernel process4
  SUBS r3, r3, #1
 VLD1.16 {d4}, [r2]!
                          ; 2 cycles (LSBP)
 VLD1.16 {d2}, [r1]!
                          ; 2 cycles (LSBP)
 VEXT.16 d5, d3, d4, #1
                         ; 1 cycle (LSBP)
 VEXT.16 d6, d3, d4, #2 ; 1 cycle (LSBP)
 VEXT.16 d7, d3, d4, #3 ; 1 cycle
                                     (LSBP)
 VMLAL.S16 q0, d3, d2[0] ; 1 cycle
                                     (DP)
 VMLAL.S16 q0, d5, d2[1] ; 1 cycle
                                     (DP)
 VMLAL.S16 q0, d6, d2[2] ; 1 cycle
                                     (DP)
 VMLAL.S16 q0, d7, d2[3] ; 1 cycle
                                     (DP)
                                     (LSBP)
 VMOV d3, d4
                          ; 1 cycle
 BGE xcorr kernel process4
```



- NEON cycle count
 - 12 NEON cycles/iteration
 - -2 cycles saved by dual-issuing
 - = 10 cycles/iteration to execute (1.6 muls/cycle)
- Data-dependencies
 - Instruction latency often longer than cycle count
 - TRM has detailed diagrams of which pipeline stage each operand is required/available in



Step 3: Instruction Latency

Instruction	Register list (alignment)	Cycles		Source				Result	
			1	2	3	4	1	2	
VLD and VST m	ultiple 1-element or 2, 3, 4	-element stru	ıcture ^b :						
VLD1	1-reg	1	-	-	-	-	-	-	
	(unaligned)	2	-	-	-	-	Dd:N1	-	
VEXT	Dd,Dn,Dm,#IMM	1	Dn:N1	Dm:N1	-	-	Dd:N2	-	
VMLAa	Qd,Dn,Dm[x]	1	Dn:N2	Dm:N1	QdLo:N3	QdHi:N3	QdLo:N6	QdHi:N6	
$VMLS^a$	(.32.16 long scalar)								
VQDMLAa	Qd,Dn,Dm[x]	1	Dn:N2	Dm:N1	QdLo:N3	QdHi:N3	_	_	
VQDMLSa	(.64.32 long scalar)	2	-	-	-	-	QdLo:N6	QdHi:N6	



Step 3: Instruction Latency

Instruction	Register list (alignment)	Cycles Source				Result		
			1	2	3	4	1	2
VLD and VST mi	ultiple 1-element or 2, 3, 4	-element str	ucture ^b :					
VLD1	1-reg	1	-	-	-	-	-	-
	(unaligned)	2	-	-	-	-	Dd:N1	-
VEXT	Dd,Dn,Dm,#IMM	1	Dn:N1	Dm:N1	-	-	Dd:N2	-
VML(Aa)	Qd,Dn,Dm[x]	1	Dn:N2	Dm:N1	QdLo:N3	QdHi:N3	QdLo:N6	QdHi:N6
VML(Sa)	(.32.16 long scalar)							
VQDML(Sa)	Qd,Dn,Dm[x]	1	Dn:N2	Dm:N1	QdLo:N3	QdHi:N3	-	-
	(.64.32 long scalar)	2	-	-	-	-	QdLo:N6	QdHi:N6

There's a special forwarding path for just this case



Step 3: Instruction Latency

Instruction	Register list (alignment)	Cycles Source					Result			
			1	2	3	4	1	2		
VLD and VST mi	ultiple 1-element or 2, 3, 4	-element	structure ^b :							
VLD1	1-reg	1	-	-	-	-	-	-		
	(unaligned)	2	-	-	-	-	Dd:N1	-		
VEXT	Dd,Dn,Dm,#IMM	1	Dn:N1	Dm:N1	-	- (Dd:N2	-		
VMLAa	Qd,Dn,Dm[x]	1	Dn:N2	Dm:N1	QdLo:N3	QdHi:N3	QdLo:N6	QdHi:N6		
VMLSa	(.32.16 long scalar)	Hmm!								
$VQDMLA^a$	Qd,Dn,Dm[x]	1	Dn:N2	Dm:N1	QdLo:N3	QdHi:N3	-	-		
VQDMLSa	(.64.32 long scalar)	2	-	-	-	-	QdLo:N6	QdHi:N6		



Main Loop: Attempt #2

```
xcorr kernel process4
  SUBS r3, r3, #4
 VLD1.16 {d2}, [r1]!
                          ; 2 cycles
                                     (LSBP)
 VLD1.16 {d4}, [r2]!
                          ; 2 cycles (LSBP)
 VMLAL.S16 q0, d3, d2[0]
                         ; 1 cycle
                                      (DP)
 VEXT.16 d5, d3, d4, #1 ; 1 cycle
                                      (LSBP)
 VMLAL.S16 q0, d5, d2[1]
                         ; 1 cycle
                                      (DP)
 VEXT.16 d5, d3, d4, #2 ; 1 cycle
                                      (LSBP)
 VMLAL.S16 q0, d5, d2[2];
                            1 cycle
                                      (DP)
 VEXT.16 d5, d3, d4, #3 ; 1 cycle
                                      (LSBP)
 VMLAL.S16 q0, d5, d2[3] ; 1 cycle
                                      (DP)
                                      (LSBP)
                          ; 1 cycle
 VMOV d3, d4
 BGE xcorr kernel process4
```



Total: Attempt #2

- 8 Load/Store/Byte Permute cycles
- 4 Data Processing cycles
- -4 dual-issue cycles
- = 8 cycles/iteration (2 muls/cycle)
- How can we do better?



Instruction Type Selection

- No good way to implement VEXT with dataprocessing instructions
 - Largest type VSHL supports is I64
 - Can't shift across 64-bit boundary
 - Other approaches require multiple instructions or multi-cycle instructions
- But VMOV...
 - Any operation that does nothing will do
 - Except VORR d3, d4, d4 (assembled to VMOV)
 - VAND d3, d4, d4 works



Instruction Type Selection

- But VMOV...
 - Any operation that does nothing will do
 - Except VORR d3, d4, d4 (assembled to VMOV)
 - VAND d3, d4, d4 works

Instruction	Register format	Cycles	Source				Result	
			1	2	3	4	1	2
VADD	Dd,Dn,Dm	1	Dn:N2	Dm:N2	-	-	Dd:N3	-
VAND VORR VEOR VBIC VORN	Qd,Qn,Qm	1	QnLo:N2	QmLo:N2	QnHi:N2	QmHi:N2	QdLo:N3	QdHi:N3



Main Loop: Attempt #3

```
Need to update prolog to
xcorr kernel process4
                       load in d4
  SUBS r3, r3, #4
  VLD1.16 {d2}, [r1]!
                           ; 2 cycles
                                      (LSBP)
 VAND d3, d4, d4
                           ; 1 cycle
                                      (DP)
                       ; 2 cycles (LSBP)
 VLD1.16 {d4}, [r2]!
 VMLAL.S16 q0, d3, d2[0]; 1 cycle
                                      (DP)
 VEXT.16 d5, d3, d4, #1 ; 1 cycle
                                      (LSBP)
 VMLAL.S16 q0, d5, d2[1] ; 1 cycle
                                      (DP)
 VEXT.16 d5, d3, d4, #2 ; 1 cycle
                                      (LSBP)
 VMLAL.S16 q0, d5, d2[2] ; 1 cycle
                                      (DP)
 VEXT.16 d5, d3, d4, #3 ; 1 cycle
                                      (LSBP)
 VMLAL.S16 q0, d5, d2[3]; 1 cycle
                                      (DP)
  BGE xcorr kernel process4
```



Total: Attempt #3

- 7 Load/Store/Byte Permute cycles
- 5 Data Processing cycles
- -5 dual-issue cycles
- = 7 cycles/iteration (2.3 muls/cycle)
- Still not keeping up with the ARM dispatch rate
 - But we're getting close!



More Unrolling

- Load instruction throughput 128 bits/cycle
 - We're only loading 64 bits
 - And paying an extra cycle for unaligned access
 - Only 25% of available throughput!
- So process 8 values at a time...



Main Loop: Attempt #4

```
xcorr kernel process8
  SUBS r3, r3, #8
 VLD1.16 {d6,d7}, [r1]! ; 2 cycles (LSBP)
 VAND d3, d5, d5 ; 1 cycle (DP)
 VLD1.16 {d4,d5}, [r2]! ; 2 cycles (LSBP)
 VMLAL.S16 q0, d3, d6[0]; 1 cycle (DP)
 VEXT.16 d16, d3, d4, #1; 1 cycle (LSBP)
 VMLAL.S16 q0, d4, d7[0]; 1 cycle (DP)
 VEXT.16 d17, d4, d5, #1; 1 cycle (LSBP)
 VMLAL.S16 q0, d16, d6[1]; 1 cycle (DP)
 VEXT.16 d16, d3, d4, #2; 1 cycle (LSBP)
 VMLAL.S16 q0, d17, d7[1]; 1 cycle
                                   (DP)
 VEXT.16 d17, d4, d5, #2; 1 cycle
                                   (LSBP)
 VMLAL.S16 q0, d16, d6[2]; 1 cycle (DP)
 VEXT.16 d16, d3, d4, #3 ; 1 cycle
                                  (LSBP)
 VMLAL.S16 q0, d17, d7[2]; 1 cycle
                                   (DP)
 VEXT.16 d17, d4, d5, #3 ; 1 cycle
                                   (LSBP)
 VMLAL.S16 q0, d16, d6[3]; 1 cycle
                                   (DP)
 VMLAL.S16 q0, d17, d7[3]; 1 cycle
                                   (DP)
 BGE xcorr kernel process8
```



Total: Attempt #4

- 10 Load/Store/Byte Permute cycles
- 9 Data Processing cycles
- -9 dual-issue cycles
- = 10 cycles/iteration (3.2 muls/cycle)
- Align x => 9 cycles (3.6 muls/cycle)
- 19 instructions => 9.5 ARM cycles to dispatch
 - NEON side no longer the bottleneck

Cleanup

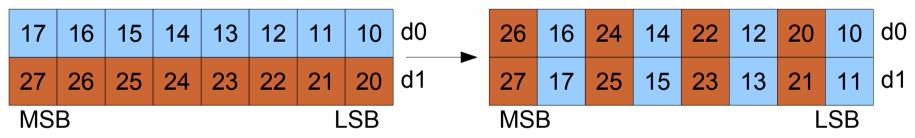
- Need to handle last few iterations
- Omitted for brevity, but a couple of points
- 2-way parallelism
 - Load two x values into d6 and d7:
 - VLD2.16 {d6[], d7[]}, [r1]!
 - Why? Non-scalar VMLAL needs 2nd input 1 cycle later
 - Load two y values into d5:
 - VLD1.32 {d5[]}, [r2]!
 - Prepare d5 for next iteration with DP instruction
 - VSRI.64 d5, d4, #32



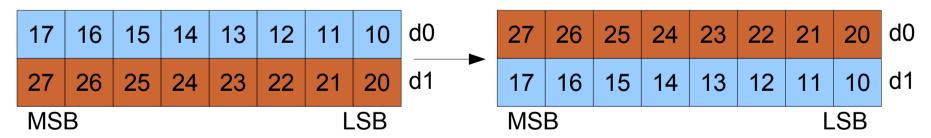
Other Available Instructions



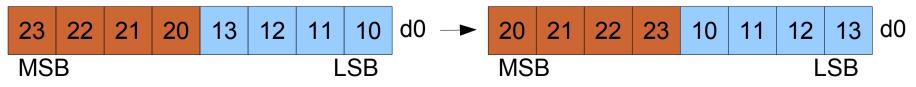
VTRN.<8,16,32> d0, d1: Transpose



VSWP d0, d1: Swap

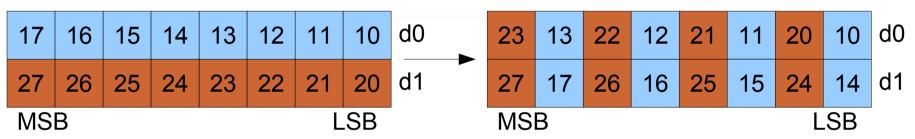


VREV<16,32,64>.<8,16,32> d0, d0: Reverse



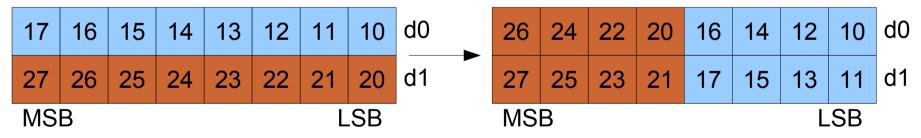


• VZIP.<8,16,32> d0, d1: Zip

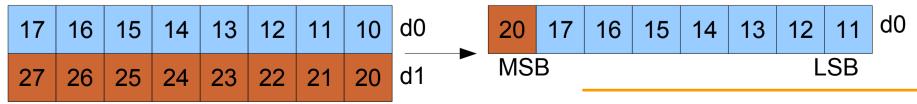


• VUZP.<8,16,32> d0, d1: Unzip

LSB



• VEXT.<8,16,32,64> d0, d0, d1, #1: Extract



51



- There are lots of MOVs
 - VMOV d0, r0, r1; ARM → NEON transfer
 - VMOV r0, r1, d0; NEON → ARM transfer
 - VMOV.<8,16,32> d0[0], r0; Single-element
 - VMOV.<S8,S16,U8,U16,32> r0, d0[0]
 - VDUP.<8,16,32> d0, d1[0]; Broadcast
 - VDUP.<8,16,32> d0, r0; ARM → NEON broadcast



- VTBL.8, VTBX.8 d0, {d1, d2, d3, d4}, d5
 - Table lookup! Incredibly powerful
 - But only from a small, consecutive register list
 - And only 64 bits of output at a time (no Q version)
 - -1+(N+1)/2 cycles (N = table size in registers)
 - 4x4 16-bit de-zig-zag in 10 cycles (8 can dual-issue)
 - Out-of-bounds indices
 - VTBL: Insert #0
 - VTBX: Leave destination unchanged



Conditional Instructions

- NEON instructions can't be conditional
 - Really want per-element flags, anyway
- VC<op>.<datatype> d0, d1, {d2 or #0}
 - Sets each element to all-0's (false) or all-1's (true)
 - op = EQ, GE, GT, LE, LT
 - datatype
 - 18, 116, 132 for EQ
 - S8, S16, S32, U8, U16, U32 for GE, GT, LE, LT
 S8, S16, S32 only with #0
- VTST.<8,16,32> d0, d1, d2: element-wise TST



Bitwise Instructions

- VAND, VBIC ("and not") d0, d1, d2
 - VAND: zeros out elements where condition false
 - VBIC: zeros out elements where condition true
- VBIT, VBIF, VBSL d0, d1, d2
 - VBIT: Bit Insert if True: Copy d1[i] to d0[i] if d2[i]
 - VBIF: Bit Insert if False: Copy d1[i] to d0[i] if !d2[i]
 - VBSL: d0[i] = d0[i] ? d1[i] : d2[i] (destroys mask)



Horizontal Arithmetic

- VPADD.I<8,16,32> d0, d1, d2
 - Concatenate d1, d2, add adjacent pairs
- VPADDL.I<8,16,32> d0, d1
 - Long variant (only one source register)
- VPADAL.I<8,16,32> d0, d1
 - Long, accumulate variant
- VPMAX.<S,U><8,16,32>,
 VPMIN.<S,U><8,16,32> d0, d1, d2
 - Pairwise max/min (no long variant needed)

Multiplies

- Normal and Long variants (no wide)
 - Second argument can be scalar in all cases
- Multiply-add, and multiply-subtract variants
- "Doubled" multiplies
 - VQDMULL.<S16,S32> q0, d2, d3
 - q0 = Saturate(2*d2*d3) (MLA, MLS versions also)
 - VQDMULH.<S16,S32> d0, d1, d2
 - Takes high half: Q15 or Q31 multiply (no MLA, MLS)
 - VQRDMULH.<S16,S32> d0, d1, d2
 - Adds rounding offset first



Other Specialized Arithmetic

- VABD.<S,U><8,16,32> d0, d1, d2
 - Absolute difference: d0 = |d1 d2|
 - VABDL (long), VABA (accumulate), VABAL (long accumulate) variants
- V{R}HADD, V{R}HSUB.<S,U><8,16,32> d0, d1, d2
 - Add/sub and halve, with optional rounding
 - Not to be confused with V{R}ADDHN,
 V{R}SUBHN!
- VQABS, VQNEG: INT_MIN → INT_MAX



Conspicuously missing

- PMOVMSKB (SSE)
 - Coverts vector mask to scalar bitmask
 - Understandable, given NEON→ARM transfer latency
 - 16-bit version emulatable in 7 instructions

```
VNEG.8 q0, q0
VZIP.8 d0, d1
VSLI.8 d0, d1, #4
VMOV r0, r1, d0
ORR r0, r0, r1, LSL #2
ORR r0, r0, r0, LSR #15
UXTH r0, r0
```



Resources

- ARM Quick Reference
 - http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001I/QRC0001_UAL.pdf
- Cortex A8 Technical Reference Manual
 - http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344k/index.html
- Cortex A8 Cycle Counter
 - http://pulsar.webshaker.net/ccc/index.php
- Coding for NEON Part 2: Dealing with Leftovers
 - http://blogs.arm.com/software-enablement/196-coding-for-neon-part-2-dealing-with-leftovers/
- My notes
 - https://people.xiph.org/~tterribe/notes/neon_instructions.txt
 - Maybe incomprehensible, but better than ARM's site