ARM NEON SIMD

Andrey Kamaev

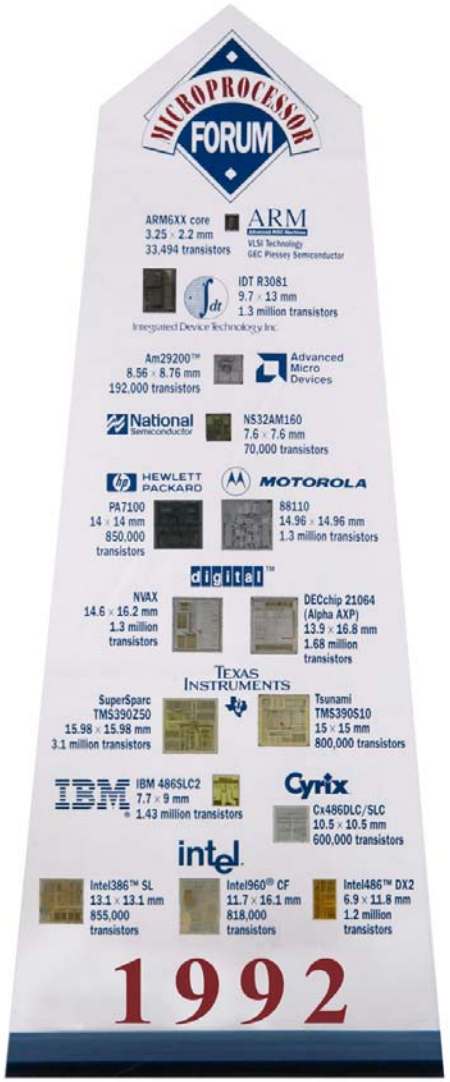
13.07.2012

Agenda

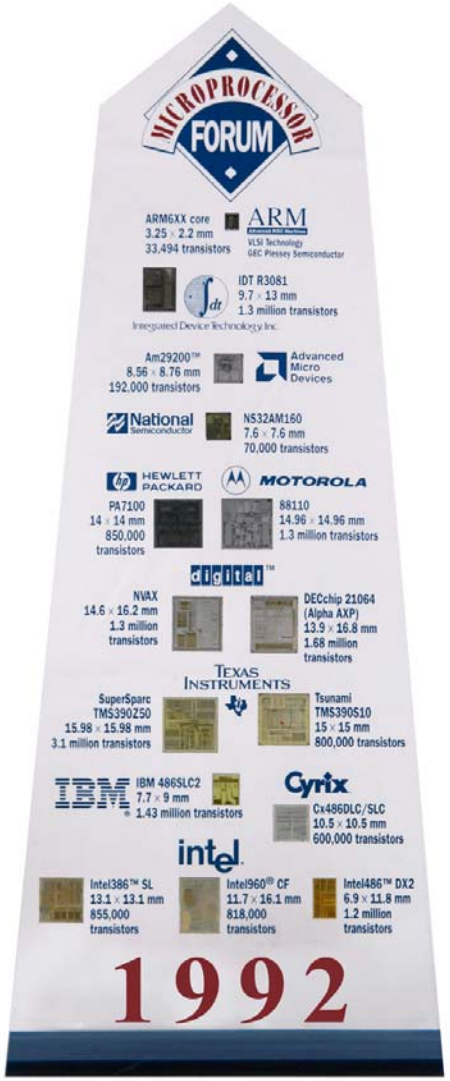
- ARM
- NEON
- Examples
- Tips & Tricks

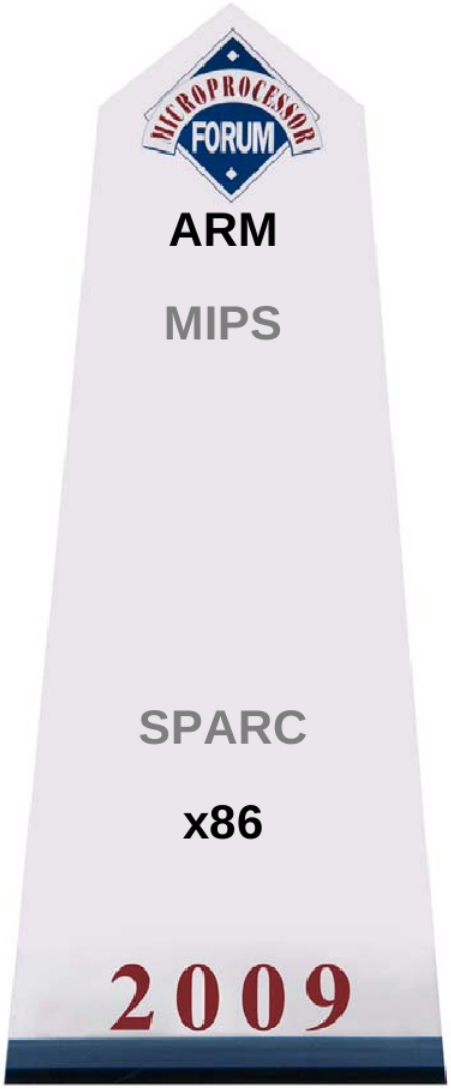
ARM

History: Microprocessor Forum 1992



History: The Survivors





ARM architectures (27 years of history)

- **ARMv1**: ARM1
- **ARMv2**: ARM2, ARM3
- **ARMv3**: ARM6, ARM7
- **ARMv4**: ARM7TDMI, ARM8, StrongARM, ARM9TDMI

- **ARMv5**: ARM7JE, ARM9E, ARM10E, XScale (1997)

- **ARMv6**: ARM11 (2002)
- **ARMv7**: Cortex-A, Cortex-R, Cortex-M (2006)

- **ARMv8**: 64-bit ARM - will come in 2014

Complete list: http://www.enotes.com/topic/List_of_ARM_microprocessor_cores

ARM Cortex-A family

- Cortex-A5 - NEON is optional
- Cortex-A8 - NEON, single core only
- Cortex-A9 - NEON is optional

- Cortex-A7 - NEON
- Cortex-A15 - NEON

Cortex-A9

- Introduces out-of-order instruction issue and completion
- Register renaming to enable execution speculation
- Non-blocking memory system with load-store forwarding
- Fast loop mode in instruction prefetch to lower power consumption

However this is not necessary applicable for NEON unit :-(

Cortex-A15

- 1 TB physical addressing (40-bit)
- Full hardware virtualization
- Scaling to 8,16-core, and beyond (AMBA4)
- ECC (Error Correction Control) on L1 and L2

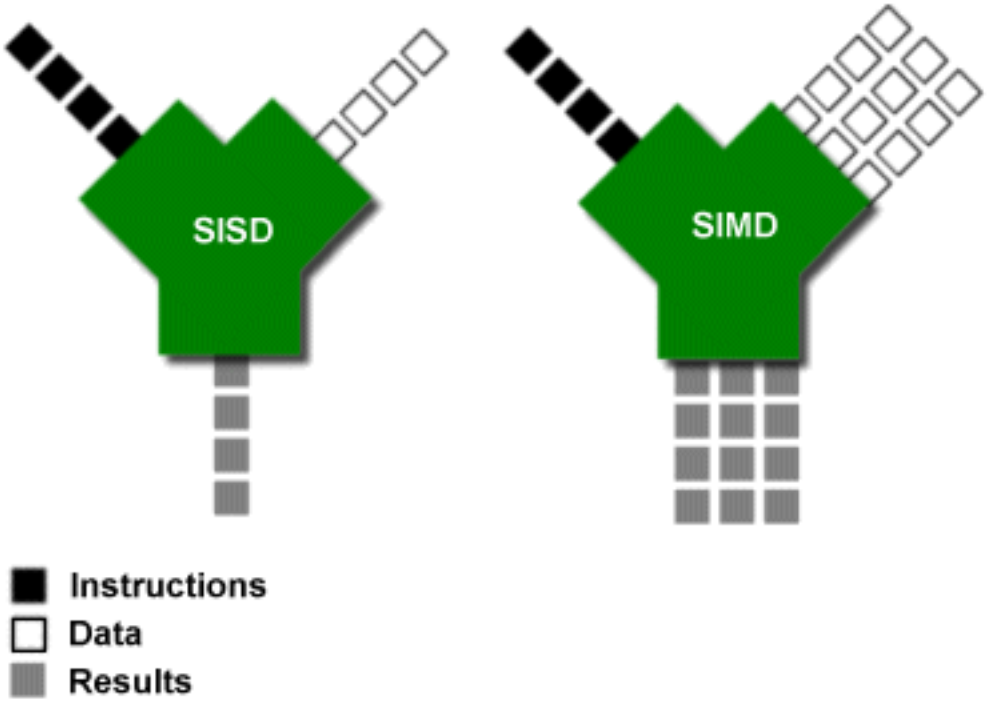
- Integer Divide
- Fused multiply-accumulate

ARM optimization

- Try to not pass more than 4 args to small functions
- Cortex-A9 has a dual-issue pipeline. Unroll loops once or twice to interleave operations, so that each operation does not need the very last few results.
- Try to just do 32-bit operations on CPU (int, float). Anything else is slower.
  - signed char is extremely slow!
- Excellent guide for pure ARM techniques: http://www.davespace.co.uk/arm/efficient-c-for-arm/index.html

NEON

NEON == Advanced SIMD



was announced in 2004

NEON often gives 8–20x boost on Cortex-A8 but only 2–5x on Cortex-A9

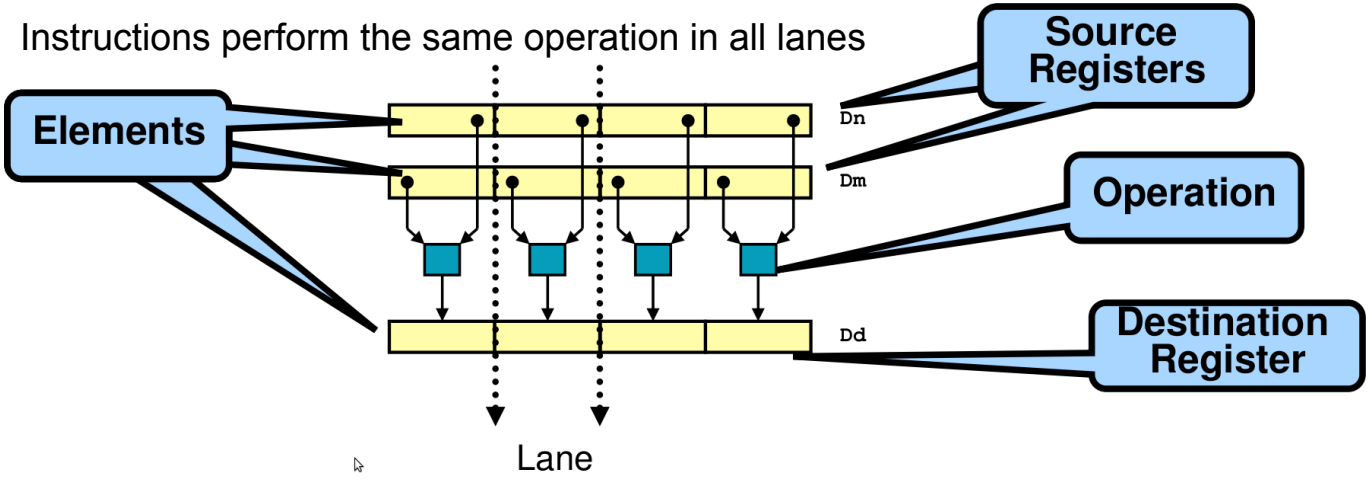
NEON or VFP?

VFP (Vector Floating Point) is ARM version of FPU. It is IEEE compliant.

NEON is opcode-compatible with VFPv3-D32 and capable for both NEON and VFP instruction sets.

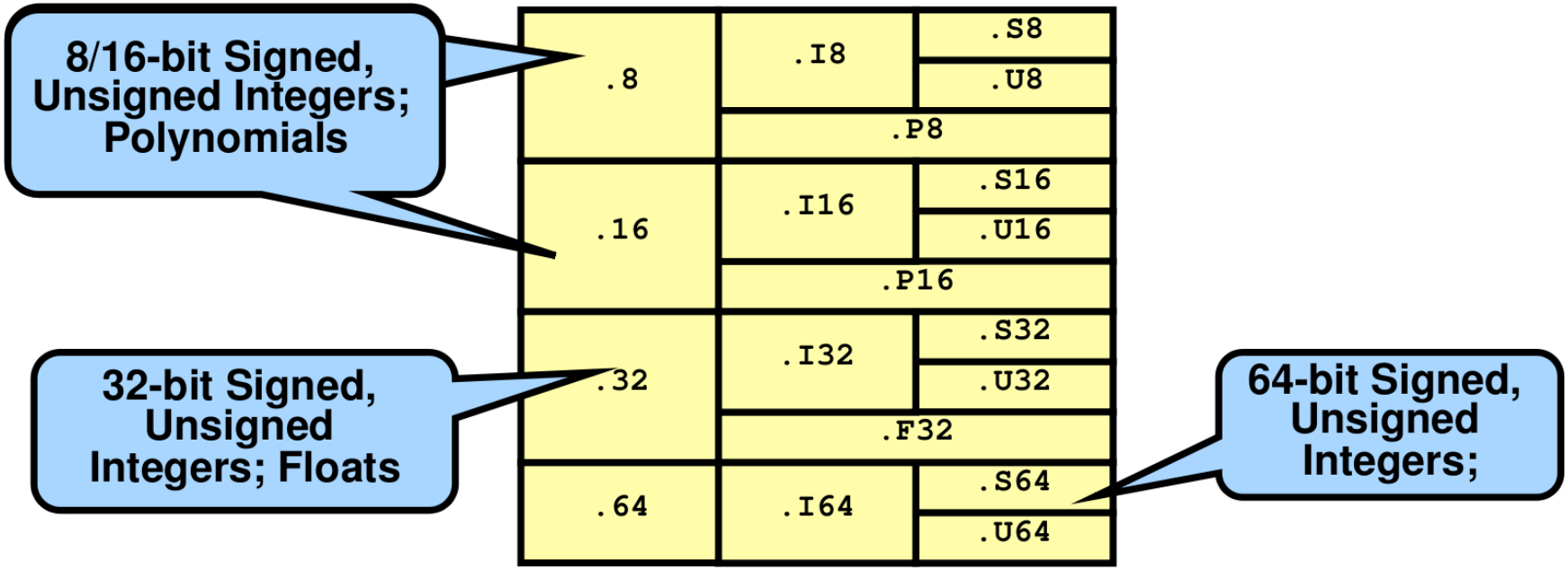
What is NEON?

- NEON is a wide SIMD data processing architecture
  - Extension of the ARM instruction set
  - 32 registers, 64-bits wide (dual view as 16 registers, 128-bits wide)
- NEON Instructions perform “Packed SIMD” processing
  - Registers are considered as vectors of elements of the same data type
  - Data types can be: signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit, single prec. float
  - Instructions perform the same operation in all lanes



Data Types

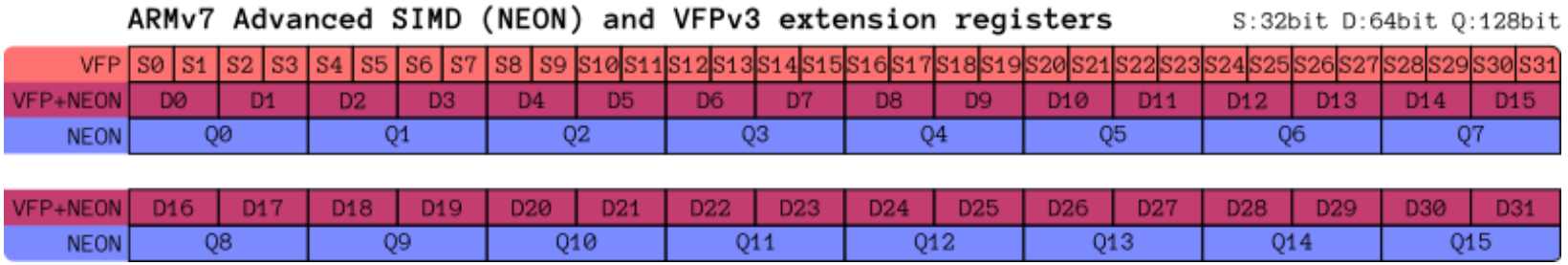
- NEON natively supports a set of common data types
  - Integer and Fixed-Point; 8-bit, 16-bit, 32-bit and 64-bit
  - 32-bit **Single-precision** Floating-point



- Data types are represented using a bit-size and format letter

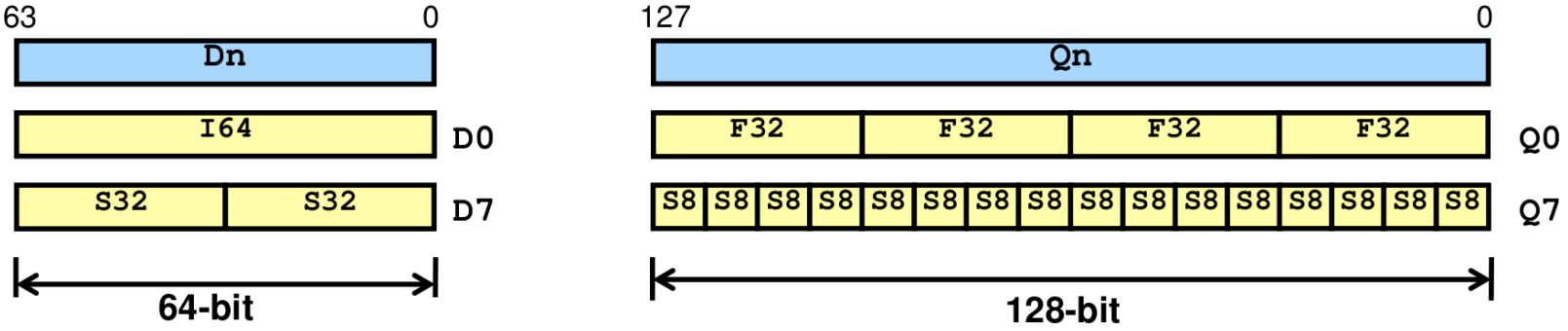
Registers

- NEON provides a 256-byte register file
  - **NEON has its own execution pipelines and a register bank that is distinct from the ARM register bank**
  - shares the same set of registers with VFP that is a floating point hardware accelerator, not parallel like NEON
- NEON Dual view
  - 32 registers, 64-bits wide (Dx)
  - 16 registers, 128-bits wide (Qx)
- VFP Dual view
  - 32 registers, 64-bits wide (Dx)
  - 32 registers, 32-bits wide (Sx)

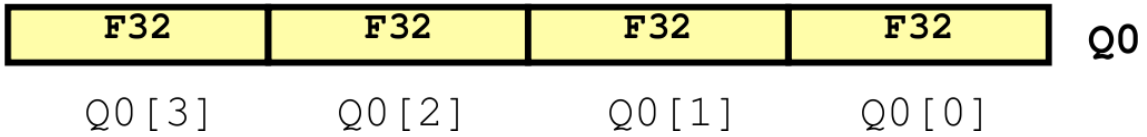


Vectors and Scalars

- Registers hold one or more elements of the same data type
  - Vn can be used to reference either a 64-bit Dn or 128-bit Qn register
  - A register, data type combination describes a vector of elements



- Some instructions can reference individual scalar elements
  - Scalar elements are referenced using the array notation Vn[x]



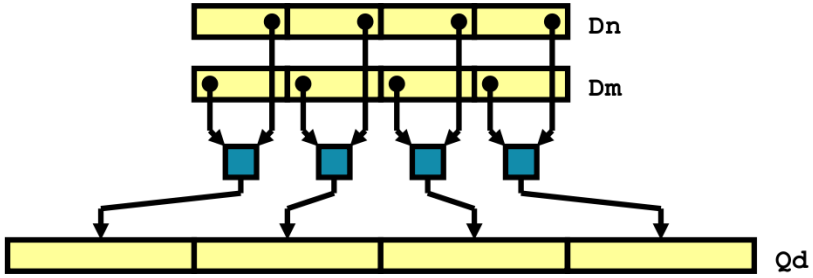
- Array ordering is always from the least significant bit

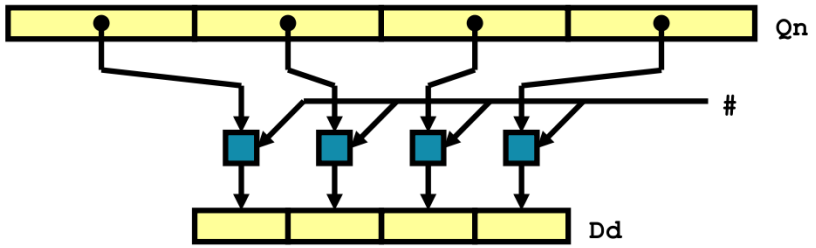
NEON operations

- Arithmetic
  - VABA, VABD, VABS, VNEG, VADD, VSUB, VADDHN, VSUBHN, VHADD, VHSUB, VPADD, VPADAL, VMAX, VMIN, VPMAX, VPMIN, VCLS, VCLZ, VCNT
- Multiplication
  - VMUL, VMLA, VMLS, VQDMULL, VQDMLAL, VQDMLSL, VQDMULH
- Shifts
  - VSHL, VSHR, VSRA, VSLI, VSRI
- Comparison and Selection
  - VCEQ, VCGE, VCGT, VCLE, VCLT, VTST, VBIF, VBIT, VBSL
- Logical
  - VAND, VBIC, VEOR, VORN, VORR, VMVN
- Reciprocal Estimate/Step, Reciprocal Square Root Estimate/Step
  - VRECPE, VRSQRTE, VRECPS, VRSQRTS
- Miscellaneous
  - VMOV, VDUP, VCVT, VEXT, VREV, VSWP, VTBL, VTBX, VTRN, VUZP, VZIP
- Load/Store
  - VLD1, VLD2, VLD3, VLD4, VST1, VST2, VST3, VST4

Long, Narrow and Wide Operations

- NEON can utilise both register views in the same instruction
  - Enables instructions to promote or demote elements within operation



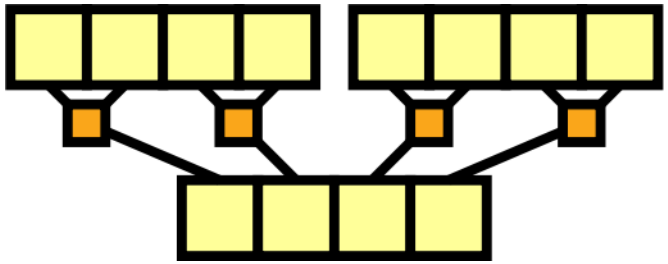


- Long operations promote elements to double the precision
  - Multiply Long (16 x 16 -> 32), Add/Sub Long, Shift Long
- Narrow operations demote data type to half the precision
  - Shift Right and Narrowing Add/Sub, Move
- Wide operations promote the elements of the second operand
  - Add/Sub Wide (16 + 32 -> 32)

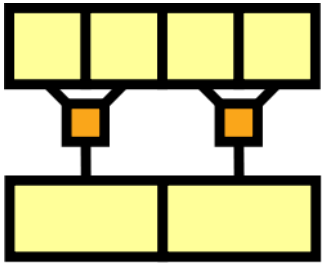
Pairwise Operations

- NEON also supports pairwise instructions to add across registers
  - ADD, MIN, MAX

- Normal

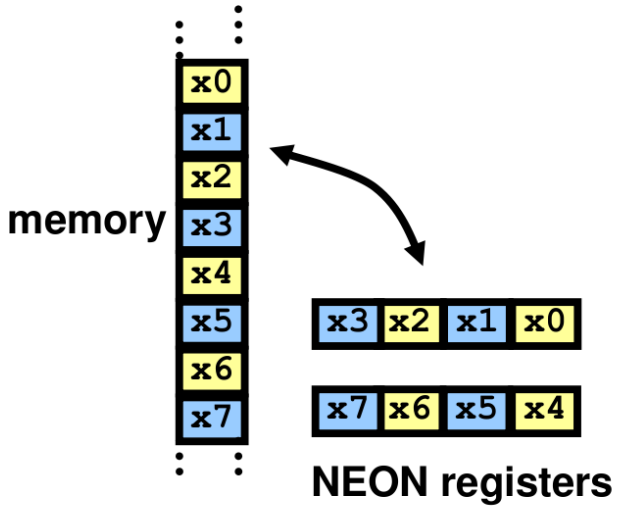


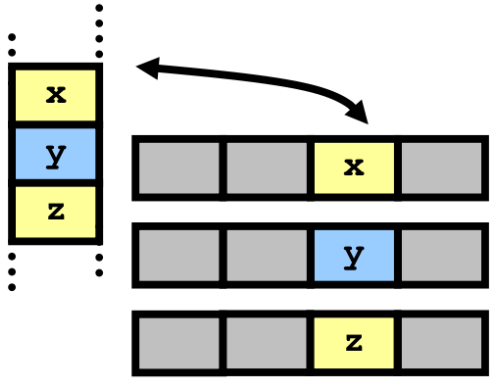
- Long

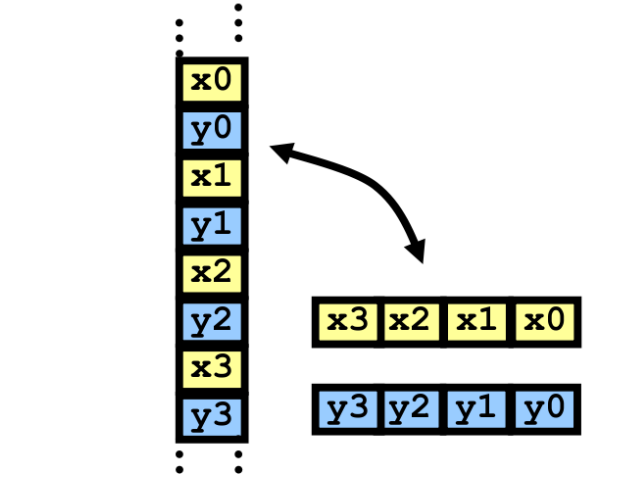


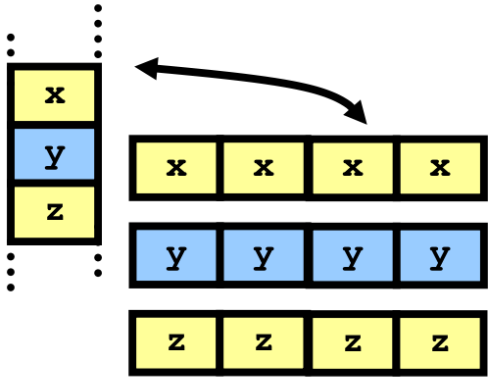
Load/Store Instructions

Several memory access patterns are possible with single instructions









NEON Intrinsics

```
#ifdef __ARM_NEON__
#include <arm_neon.h>
#endif
```

gcc: -march=armv7-a **-mfloat-abi=softfp -mfpu=neon**

1870 intrinsics are available!

NEON Intrinsics

Provide an intermediate step for SIMD code generation between a vectorizing compiler and assembler code

- The intrinsics use a naming scheme that is similar to the NEON unified assembler syntax: **`v<opname><flags>_<type>`**
- An additional **q** flag specifies operation on 128-bit vectors
- **Strongly typed!** (Use *`vreinterpret_<type>_<type>()`* to change the type)

For example:

`vmul_s16`, multiplies two 64-bit vectors containing signed 16-bit values.

This compiles to *`VMUL.I16 d2, d0, d1`*

`vaddq_u16`, is an add of two 128-bit vectors containing unsigned 16-bit values, resulting in a 128-bit vector of unsigned 16-bit values

This compiles to *`VADD.I16 q2, q0, q1`*

Vector data types

NEON vector data types are named like this:

<type><size>x<number of lanes>_t

```
int8x8_t   uint8x8_t   poly8x8_t    float32x2_t
int8x16_t  uint8x16_t  poly8x16_t   float32x4_t
int16x4_t  uint16x4_t  poly16x4_t
int16x8_t  uint16x8_t  poly16x8_t
int32x2_t  uint32x2_t
int32x4_t  uint32x4_t
int64x1_t  uint64x1_t
int64x2_t  uint64x2_t
```

Array of vector types:

<type><size>x<number of lanes>x<length of array>_t

These are ordinary C structures, like:

```
struct int16x4x2_t {
    int16x4_t val[2];
};
```

NEON Intrinsic example

```
uint8x8_t vqrshrun_n_s16(int16x8_t a, const int b);
```

Signed rounding shift right by immediate value, narrow and saturate to unsigned.

```
void vqrshrun_n_s16(short a[8], const int b, uchar dst[8])
{
  for (int i = 0; i < 8; ++i)
    dst[i] = cv::saturate_cast<uchar>((int(a[i]) + (1 << (b-1))) >> b);
}
```

15 More Examples

```
int16x4_t vadd_s16   (int16x4_t a, int16x4_t b);  //64-bit registers
int16x8_t vaddq_s16  (int16x8_t a, int16x8_t b);  //128-bit registers
int32x4_t vaddl_s16  (int16x4_t a, int16x4_t b);  //long form
int32x4_t vaddw_s16  (int32x4_t a, int16x4_t b);  //wide form
int16x4_t vqadd_s16  (int16x4_t a, int16x4_t b);  //saturating form
int16x8_t vqaddq_s16 (int16x8_t a, int16x8_t b);
int8x8_t  vaddhn_s16 (int16x8_t a, int16x8_t b);  //narrow form
int8x8_t  vraddhn_s16(int16x8_t a, int16x8_t b);  // + rounding
int16x4_t vhadd_s16  (int16x4_t a, int16x4_t b);  //halving add
int16x8_t vhaddq_s16 (int16x8_t a, int16x8_t b);
int16x4_t vrhadd_s16 (int16x4_t a, int16x4_t b);  // + rounding
int16x8_t vrhaddq_s16(int16x8_t a, int16x8_t b);
int16x4_t vpadd_s16  (int16x4_t a, int16x4_t b);  //pairwise
int32x2_t vpaddl_s16 (int16x4_t a);               //long pairwise
int32x4_t vpaddlq_s16(int16x8_t a);
```

Code Examples

Example: Binary Threshold

```
void threshold(uchar* src, uchar* dst, int length,
               uchar thresh, uchar maxval)
{
    for(int i = 0; i < length; ++i)
        dst[i] = src[i] <= thresh ? 0 : maxval;
}
```

Binary Threshold: NEON

```
void threshold_NEON(uchar* src, uchar* dst, int length, uchar thresh, uchar maxval)
{
     uint8x16_t vthreshold = vdupq_n_u8(thresh);
     uint8x16_t vvalue = vdupq_n_u8(maxval);
     for(int i = 0; i <= length - 32; i += 32 )
     {
          __builtin_prefetch(src + i + 320);
          uint8x16_t v0 = vld1q_u8(src + i);
          uint8x16_t v1 = vld1q_u8(src + i + 16);
          uint8x16_t r0 = vcgtq_u8(v0, vthreshold);
          uint8x16_t r1 = vcgtq_u8(v1, vthreshold);
          uint8x16_t r0a = vandq_u8(r0, vvalue);
          uint8x16_t r1a = vandq_u8(r1, vvalue);
          vst1q_u8(dst + i, r0a);
          vst1q_u8(dst + i + 16, r1a);
     }
}
```

Binary Threshold: SSE

```
void threshold_SSE(uchar* src, uchar* dst, int length, uchar thresh, uchar maxval)
{
     __m128i _x80 = _mm_set1_epi8('\x80');
     __m128i thresh_s = _mm_set1_epi8(thresh ^ 0x80);
     __m128i maxval_ = _mm_set1_epi8(maxval);
     for(int i = 0; i <= length - 32; i += 32 )
     {
          __m128i v0 = _mm_loadu_si128( (const __m128i*)(src + i) );
          __m128i v1 = _mm_loadu_si128( (const __m128i*)(src + i + 16) );
          v0 = _mm_cmpgt_epi8( _mm_xor_si128(v0, _x80), thresh_s );
          v1 = _mm_cmpgt_epi8( _mm_xor_si128(v1, _x80), thresh_s );
          v0 = _mm_and_si128( v0, maxval_ );
          v1 = _mm_and_si128( v1, maxval_ );
          _mm_storeu_si128( (__m128i*)(dst + i), v0 );
          _mm_storeu_si128( (__m128i*)(dst + i + 16), v1 );
     }
}
```

Example: BGRA unpack

Input:

```
BGRA BGRA BGRA BGRA
BGRA BGRA BGRA BGRA
BGRA BGRA BGRA BGRA
BGRA BGRA BGRA BGRA
```

Output:

```
BBBB BBBB BBBB BBBB
GGGG GGGG GGGG GGGG
RRRR RRRR RRRR RRRR
AAAA AAAA AAAA AAAA
```

BGRA unpack: SSE2 (20 instructions)

```
__m128i v0 = _mm_load_si128((const __m128i*)(bgra));
__m128i v1 = _mm_load_si128((const __m128i*)(bgra + 8));
__m128i v2 = _mm_load_si128((const __m128i*)(bgra + 16));
__m128i v3 = _mm_load_si128((const __m128i*)(bgra + 24));
__m128i t0 = _mm_unpacklo_epi8(v0, v1);
__m128i t1 = _mm_unpackhi_epi8(v0, v1);
__m128i t2 = _mm_unpacklo_epi8(v2, v3);
__m128i t3 = _mm_unpackhi_epi8(v2, v3);
v0 = _mm_unpacklo_epi8(t0, t1);
v1 = _mm_unpackhi_epi8(t0, t1);
v3 = _mm_unpacklo_epi8(t2, t3);
v4 = _mm_unpackhi_epi8(t2, t3);
t0 = _mm_unpacklo_epi32(v0, v1);
t1 = _mm_unpackhi_epi32(v0, v1);
t3 = _mm_unpacklo_epi32(v2, v3);
t4 = _mm_unpackhi_epi32(v2, v3);
__m128i B = _mm_unpacklo_epi64(t0, t2);
__m128i G = _mm_unpackhi_epi64(t0, t2);
__m128i R = _mm_unpacklo_epi64(t1, t3);
__m128i A = _mm_unpackhi_epi64(t1, t3);
```

BGRA unpack: NEON (8 instructions)

```
uint8x16_t v0 = vld1q_u8(bgra);
uint8x16_t v1 = vld1q_u8(bgra+8);
uint8x16_t v2 = vld1q_u8(bgra+16);
uint8x16_t v3 = vld1q_u8(bgra+24);
uint8x16x2_t v01 = vtrnq_u8(v0, v1);
uint8x16x2_t v23 = vtrnq_u8(v2, v3);
uint16x8x2_t BR = vtrnq_u16(vreinterpretq_u16_u8(v01.val[0]),
                            vreinterpretq_u16_u8(v23.val[0]));
uint16x8x2_t GA = vtrnq_u16(vreinterpretq_u16_u8(v01.val[1]),
                            vreinterpretq_u16_u8(v23.val[1]));
uint8x16_t B = vreinterpretq_u8_u16(BR.val[0]);
uint8x16_t G = vreinterpretq_u8_u16(GA.val[0]);
uint8x16_t R = vreinterpretq_u8_u16(BR.val[1]);
uint8x16_t A = vreinterpretq_u8_u16(GA.val[1]);
```

BGRA unpack: NEON 2

```
uint8x16x4_t vbgra = vld4q_u8(bgra);
```

(surprisingly, 2 instructions)

Example: BGR to RGB conversion

```
void convert(uchar* bgr, uchar* rgb, int length)
{
    for(int i = 0; i <= length - 16; i += 16)
    {
        uint8x16x3_t v = vld3q_u8(bgr + i*3);
        uint8x16_t tmp = v.val[0];
        v.val[0] = v.val[2];
        v.val[2] = tmp;
        vst3q_u8(rgb + i*3);
    }
}
```

Example: BGR to RGB conversion 2

```
void convert(uchar* bgr, uchar* rgb, int length)
{
     for(int i = 0; i <= length - 16; i += 16)
     {
          __asm__ (
               "vld3.8 {d0, d2, d4}, [%[in0]]          \n\t"
               "vld3.8 {d1, d3, d5}, [%[in1]]          \n\t"
               "vswp q0, q2                            \n\t"
               "vst3.8 {d0, d2, d4}, [%[out0]]         \n\t"
               "vst3.8 {d1, d3, d5}, [%[out1]]         \n\t"
               : /*no output*/
               : [out0] "r" (dst + 3 * i),
                 [out1] "r" (dst + 3 * (i + 8)),
                 [in0]  "r" (src + 3 * i),
                 [in1]  "r" (src + 3 * (i + 8))
               : "d0","d1","d2","d3","d4","d5"
          );
     }
} // see http://hardwarebug.org/2010/07/06/arm-inline-asm-secrets/for magic registry codes
```

NEON vs SSE

- Separate HW unit
- Strongly typed intrinsics
- 256 byte register bank

- No double precision
- No way to specify alignment for intrinsics
- Need software prefetch

- Same HW unit
- Type-aware intrinsics
- 128(x86)/256(x64) byte register bank
- Has double precision
- Special instructions for aligned load/store
- Good hardware prefetch

Tips & Tricks

and other nuances

Don't mix NEON & non-NEON code

Copying from a NEON register to a CPU register causes ~20 cycle delay!

- avoid branches on results of floating-point compare
- don't use vget_lane_<type>() /vset_lane_<type> for CPU<->NEON transfers
- process leftovers with NEON

Data Alignment

- Intrinsics do not support alignment
- Assembler does:

```
vst1.16 {d6-d7}, [%[out0],:128]
```

- Aligned loads/stores are bit faster
  - but it is hard to get any speed improvement from the alignment on Cortex-A9

Software prefetch

- Hardware prefetch unit is weak
  - often it is completely turned off by OS

- GCC intrinsic:

```
void __builtin_prefetch (const void *addr, ...)
```

- Assembler command:

```
PLD [Rn {, #offset}]
```

There is also PLDW instruction (prefetch with intent to write) but it is not supported by GCC.

Software prefetch

- Safe to access unallocated memory
- Single prefetch command fetches only one cache line (32 bytes)

- Prefetch allows to minimize slowdown if input data excess the size of L2
- Experiments show that the best strategy of prefetching is looking 4-12 cache lines after the currently accessed address
- Can give up to 2x performance boost (but 15-20% in average case)

Non-IEEE floating point

NEON floating point operations are not fully-compliant with IEEE standard. So explicit cast to **float32_t** is required:

```
void copy(const float *src, float*dst)
{
    float32x4_t v = vld1q_f32((const float32_t*)src);
    vst1q_f32((float32_t*)dst, v);
}
```

No hardware divide

But there are *reciprocal estimate* and *reciprocal step* instructions:

```
float32x4_t vx = vld1q_f32((const float32_t*)(X+i)); //x
float32x4_t vxinv_rep = vrecpeq_f32(vx);
float32x4_t vxinv_st1 = vrecpsq_f32(vxinv_rep, vx);
float32x4_t vxinv_st2 = vmulq_f32(vxinv_rep, vxinv_st1);
float32x4_t vxinv_st3 = vrecpsq_f32(vxinv_st2, vx);
float32x4_t vxinv = vmulq_f32(vxinv_st2, vxinv_st3); //1/x
```

Same situation with sqrt.

Prefer 128-bit registers

The instructions operating on long 128-bit registers can make your code twice faster than instructions operating on 64-bit registers just because they process twice more data within the same processor clocks.

Even if it is not true for current NEON implementations it will be in future ARM SOCs.

Don't combine many intrinsics in one line

ARM compilers are not as smart as x86 compilers we are usually using

Do not:

```
vst1q_u8(dst + i, vandq_u8(vcgtq_u8(vld1q_u8(src + i), vthreshold), vvalue));
```

Do:

```
uint8x16_t v = vld1q_u8(src + i);
uint8x16_t m = vcgtq_u8(v, vthreshold);
uint8x16_t t = vandq_u8(m, vvalue);
vst1q_u8(dst + i, t);
```

Do not reuse variables in adjacent commands

Unroll your loops 2 more times if inner parallelism in not enough:

```
for(int i = 0; i <= length - 32; i += 32 )
{
      uint8x16_t v0 = vld1q_u8(src + i);
      uint8x16_t v1 = vld1q_u8(src + i + 16);
      uint8x16_t r0 = vcgtq_u8(v0, vthreshold);
      uint8x16_t r1 = vcgtq_u8(v1, vthreshold);
      uint8x16_t r0a = vandq_u8(r0, vvalue);
      uint8x16_t r1a = vandq_u8(r1, vvalue);
      vst1q_u8(dst + i, r0a);
      vst1q_u8(dst + i + 16, r1a);
}
```

Spill & Fill
(when 32 registers is not enough)

Compiler (GCC) can't do it right

1. Optimize registers usage
2. Use twice smaller unroll step
3. Do spill/fill manually

- Don't mix NEON intrinsics/assembler with floating point arithmetic
  - it can work if you use only few registers

Compiler lies to you

There is a bug in GCC breaking vldN and vstN intrinsics (N=2,3,4):

vst3q_u8(dst, vec16)

Fixed only in GCC 4.6

Intrinsics lie to you

```
int32x4_t vmlaq_n_s32(int32x4_t a, int32x4_t b, int32_t c)
```

Never do:

```
inline int32x4_t addWeighted(int32x4_t a, int32x4_t b)
{
    return vmlaq_n_s32(a, b, 3);
}
```

Do it right:

```
inline int32x4_t addWeighted(int32x4_t a, int32x4_t b, int32x4_t c)
{
    return vmlaq_n_s32(a, b, vgetq_lane_s32(c, 0));
}
```

Assembler lies to you

There are two different instructions with the same name:

`vcvt.s32.f32`

The first one belongs to VFPv3 instruction set and rounds to nearest even ("banker's rounding")

Second is NEON vector round towards zero

Advanced: Be aware of typical NEON implementation

- 2 64-bit simple integer ALUs, but only one of them is capable of operations like bit selects, variable shifts, and horizontal operations
- 1 64/128-bit permute unit
- 1 128-bit integer multiplier w/accumulate
- 1 128-bit load/store unit
- 2 single precision floating point multipliers and 2 single precision floating point add/sub/cmp/etc

So integer multiply-accumulate is always fast.
But zip/unzip/ext - are usually much slower.

Q&A

Links

- Coding for NEON - Part 1: Load and Stores
- Coding for NEON - Part 2: Dealing With Leftovers
- Coding for NEON - Part 3: Matrix Multiplication
- Coding for NEON - Part 4: Shifting Left and Right
- Coding for NEON - Part 5: Rearranging Vectors

- Condition Codes 1: Condition Flags and Codes
- Condition Codes 2: Conditional Execution
- Condition Codes 3: Conditional Execution in Thumb-2
- Condition Codes 4: Floating-Point Comparisons Using VFP

- NEON and VFP Programming - instructions reference
- gcc builtins - prefetch, etc

Links 2

- List of ARM microprocessor cores
- ARM inline assembler codes
- Efficient C for ARM
- Choosing the next generation application processor Cortex-A15.pdf
- NEON support in the arm compiler.pdf
- Lection from EITF20 course: Lessons from the ARM Architecture
- Lections from Stanford EE282 spring 2010 (##7,9,10)
- ARM NEON technology
- NVIDIA Tegra2

ARM assembly in nutshell

RISC processor

Almost orthogonal instruction set

- easier to learn than Intel assembly

Free per-instruction conditionals

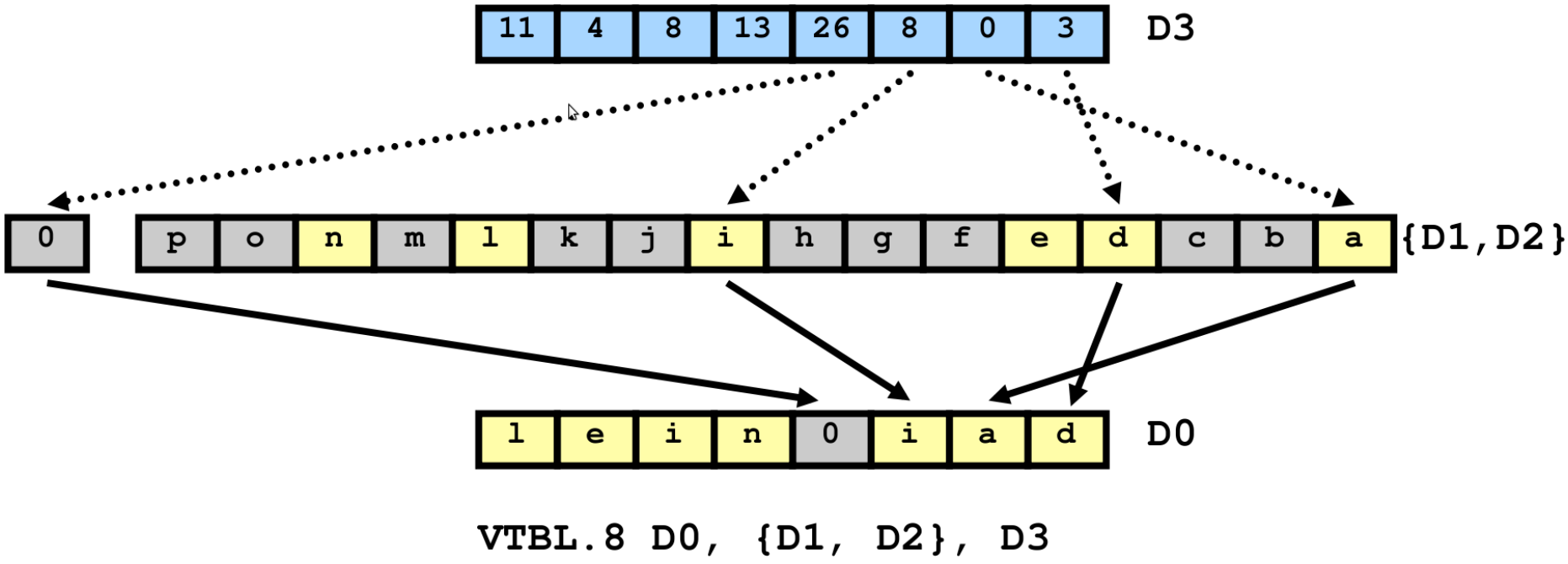
16 registers (R0 – R15), all 32-bit

- R15 PC (program counter)

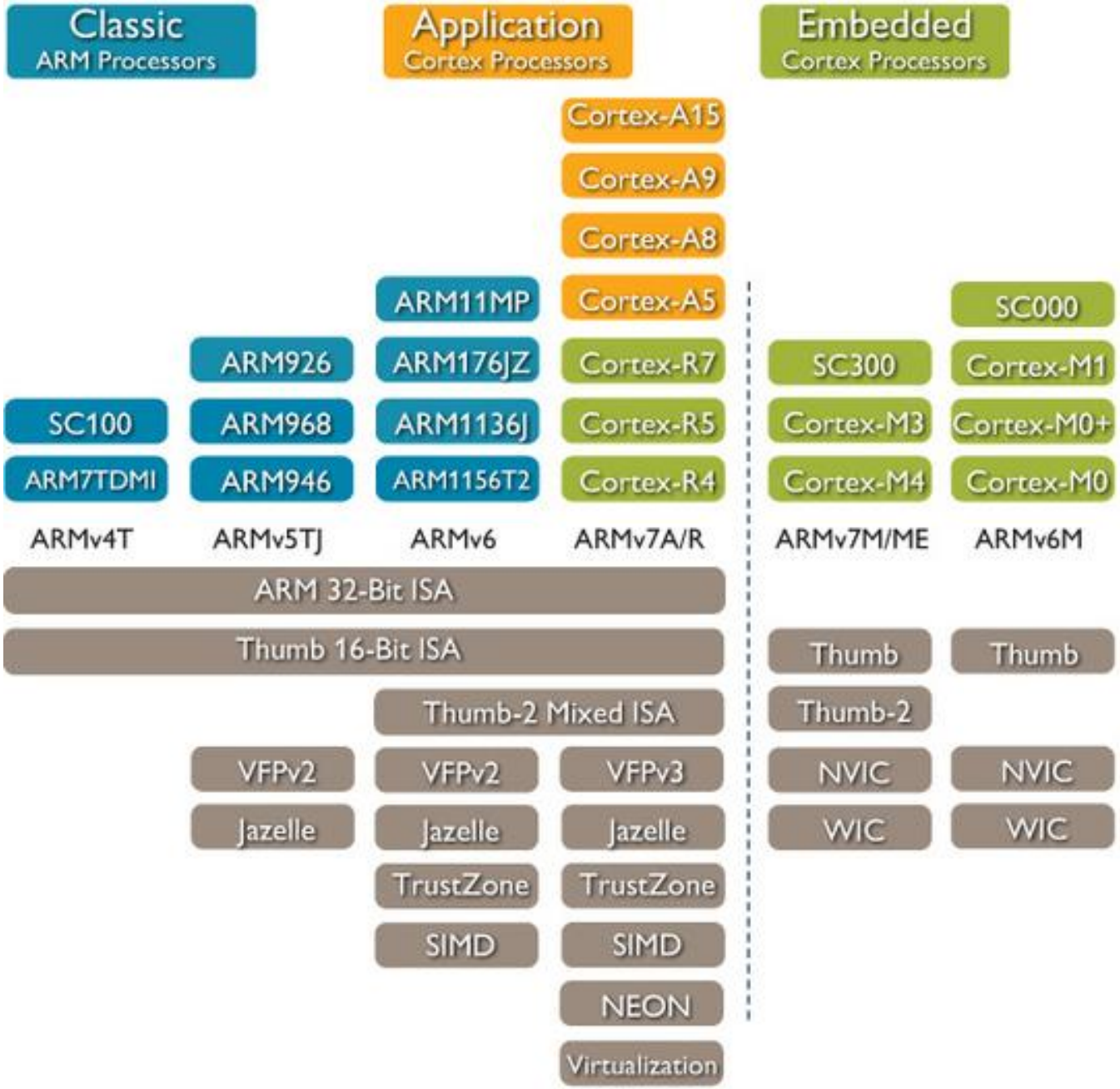
Calling Convention registers:

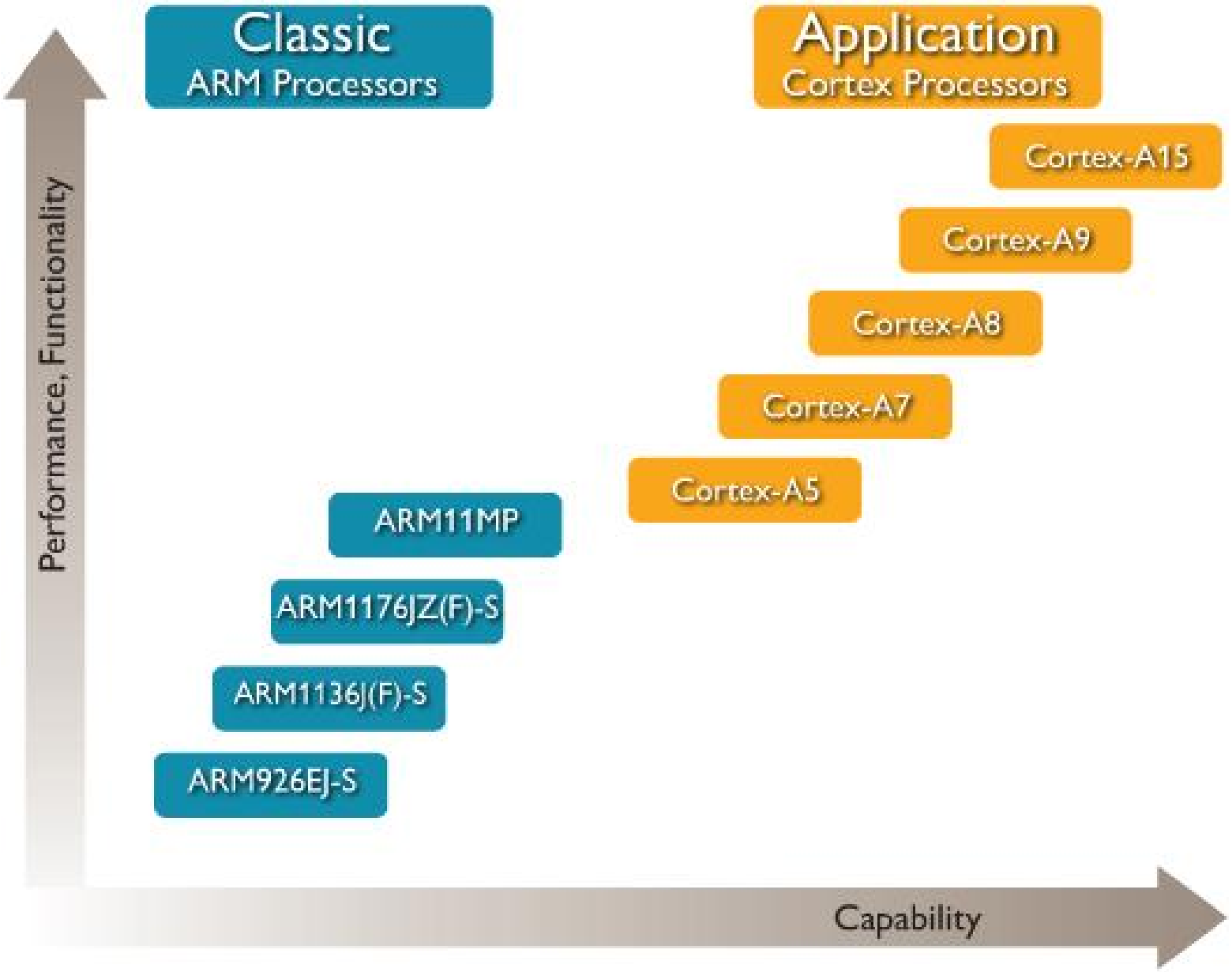
- R13 SP (stack pointer)
- R14 LR (link register, function return address)
  (can be used in functions by storing to stack)
- R0-R3 first 4 parameters to a function (scratch registers)
- R0 return value from a function

NEON VTBL: Table Lookup



- VTBL : out of range indexes generate 0 result
- VTBX : out of range indexes leave destination unchanged


Classic
ARM Processors
Application
Cortex Processors
Embedded
Cortex Processors
Cortex-A15
Cortex-A9
Cortex-A8
ARM11MP
Cortex-A5
SC000
ARM926
ARM176JZ
Cortex-R7
SC300
Cortex-M1
SC100
ARM968
ARM1136J
Cortex-R5
Cortex-M3
Cortex-M0+
ARM7TDMI
ARM946
ARM1156T2
Cortex-R4
Cortex-M4
Cortex-M0
ARMv4T
ARMv5TJ
ARMv6
ARMv7A/R
ARMv7M/ME
ARMv6M
ARM 32-Bit ISA
Thumb 16-Bit ISA
Thumb
Thumb
Thumb-2 Mixed ISA
Thumb-2
VFPv2
VFPv2
VFPv3
NVIC
NVIC
Jazelle
Jazelle
Jazelle
WIC
WIC
TrustZone
TrustZone
SIMD
SIMD
NEON
Virtualization


Classic
ARM Processors
Application
Cortex Processors
Cortex-A15
Cortex-A9
Cortex-A8
Cortex-A7
Cortex-A5
ARM11MP
ARM1176JZ(F)-S
ARM1136J(F)-S
ARM926EJ-S
Performance, Functionality
Capability