Smartphone/tablet CPUs

iPad 1 (2010) was the
first popular tablet:
more than 15 million sold.

iPad 1 contains 45nm
Apple A4 system-on-chip.

Apple A4 contains
1GHz ARM Cortex-A8 CPU core
+ PowerVR SGX 535 GPU.

Cortex-A8 CPU core (2005)
supports ARMv7-A insn set,
including NEON vector insns.

Apple A4 also appeared in iPhone 4 (2010).

45nm 1GHz Samsung Exynos 3110 in Samsung Galaxy S (2010) contains Cortex-A8 CPU core.

45nm 1GHz TI OMAP3630 in Motorola Droid X (2010) contains Cortex-A8 CPU core.

65nm 800MHz Freescale i.MX50 in Amazon Kindle 4 (2011) contains Cortex-A8 CPU core.

ARM designed more cores supporting same ARMv7-A insns:
Cortex-A9 (2007),
Cortex-A5 (2009),
Cortex-A15 (2010),
Cortex-A7 (2011),
Cortex-A17 (2014), etc.

Also some larger 64-bit cores.

A9, A15, A17, and some 64-bit cores are "out of order": CPU tries to reorder instructions to compensate for dumb compilers.

A5, A7, original A8 are in-order,
fewer insns at once.

A5, A7, original A8 are in-order, fewer insns at once. $\Rightarrow$ Simpler, cheaper, more energy-efficient.

A5, A7, original A8 are in-order, fewer insns at once. $\Rightarrow$ Simpler, cheaper, more energy-efficient.

More than one billion Cortex-A7 devices have been sold.

Popular in low-cost and mid-range smartphones: Mobiistar Buddy, Mobiistar Kool, Mobiistar LAI Z1, Samsung Galaxy J1 Ace Neo, etc.

Also used in typical TV boxes, Sony SmartWatch 3, Samsung Gear S2, Raspberry Pi 2, etc.

NEON crypto

Basic ARM insn set uses
16 32-bit registers: 512 bits.

Optional NEON extension uses
16 128-bit registers: 2048 bits.

Cortex-A7 and Cortex-A8
(and Cortex-A15 and Cortex-A17
and Qualcomm Scorpion
and Qualcomm Krait)
always have NEON insns.

Cortex-A5 and Cortex-A9
sometimes have NEON insns.

2012 Bernstein–Schwabe
"NEON crypto" software:
new Cortex-A8 speed records
for various crypto primitives.

e.g. Curve25519 ECDH:
460200 cycles on Cortex-A8-fast,
498284 cycles on Cortex-A8-slow.

Compare to OpenSSL
cycles on Cortex-A8-slow
for NIST P-256 ECDH:
9 million for OpenSSL 0.9.8k.
4.8 million for OpenSSL 1.0.1c.
3.9 million for OpenSSL 1.0.2j.

NEON instructions

`4x a = b + c`

is a vector of 4 32-bit additions:

```
    a[0] = b[0] + c[0];
    a[1] = b[1] + c[1];
    a[2] = b[2] + c[2];
    a[3] = b[3] + c[3].
```

NEON instructions

`4x a = b + c`

is a vector of 4 32-bit additions:

```
    a[0] = b[0] + c[0];
    a[1] = b[1] + c[1];
    a[2] = b[2] + c[2];
    a[3] = b[3] + c[3].
```

Cortex-A8 NEON arithmetic unit can do this every cycle.

NEON instructions

`4x a = b + c`
is a vector of 4 32-bit additions:

```
    a[0] = b[0] + c[0];
    a[1] = b[1] + c[1];
    a[2] = b[2] + c[2];
    a[3] = b[3] + c[3].
```

Cortex-A8 NEON arithmetic unit
can do this every cycle.

Stage N2: reads `b` and `c`.
Stage N3: performs addition.
Stage N4: `a` is ready.

$$\begin{tikzcd}\text{ADD} \arrow[r, "2 \text{ cycles}"] & \text{ADD} \arrow[r, "2 \text{ cycles}"] & \text{ADD}\end{tikzcd}$$

`4x a = b - c`

is a vector of 4 32-bit subtractions:

```
    a[0] = b[0] - c[0];
    a[1] = b[1] - c[1];
    a[2] = b[2] - c[2];
    a[3] = b[3] - c[3].
```

Stage N1: reads c.
Stage N2: reads b, negates c.
Stage N3: performs addition.
Stage N4: a is ready.

$$\text{ADD} \xrightarrow{\text{2 or 3 cycles}} \text{SUB}$$

Also logic insns, shifts, etc.

Multiplication insn:

```
c[0,1] = a[0] signed* b[0];
c[2,3] = a[1] signed* b[1]
```

Two cycles on Cortex-A8.

Multiply-accumulate insn:

```
c[0,1] += a[0] signed* b[0];
c[2,3] += a[1] signed* b[1]
```

Also two cycles on Cortex-A8.

Stage N1: reads b.
Stage N2: reads a.
Stage N3: reads c if accumulate.
⋮
Stage N8: c is ready.

Typical sequence of three insns:

```
c[0,1] = a[0] signed* b[0];
c[2,3] = a[1] signed* b[1]

c[0,1] += e[2] signed* f[2];
c[2,3] += e[3] signed* f[3]

c[0,1] += g[0] signed* h[2];
c[2,3] += g[1] signed* h[3]
```

Cortex-A8 recognizes this pattern.
Reads `c` in N6 instead of N3.

Time	N1	N2	N3	N4	N5	N6	N7	N8
1	b							
2		a						
3	f		$\times$					
4		e		$\times$				
5	h		$\times$		$\times$			
6		g		$\times$		$\times$		
7			$\times$		$\times$			
8				$\times$		$\times$		c
9					$\times$		$+$	
10						$\times$		c
11							$+$	
12								c

NEON also has load/store insns and permutation insns: e.g., `r = s[1] t[2] r[2,3]`

Cortex-A8 has a separate NEON load/store unit that runs in parallel with NEON arithmetic unit.

Arithmetic is typically most important bottleneck: can often schedule insns to hide loads/stores/perms.

Cortex-A7 is different: one unit handling all NEON insns.

Curve25519 on NEON

Radix $2^{25.5}$: Use small integers $(f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9)$ to represent the integer $f = f_0 + 2^{26}f_1 + 2^{51}f_2 + 2^{77}f_3 + 2^{102}f_4 + 2^{128}f_5 + 2^{153}f_6 + 2^{179}f_7 + 2^{204}f_8 + 2^{230}f_9$ modulo $2^{255} - 19$.

Unscaled polynomial view: f is value at $2^{25.5}$ of the poly $f_0t^0 + 2^{0.5}f_1t^1 + f_2t^2 + 2^{0.5}f_3t^3 + f_4t^4 + 2^{0.5}f_5t^5 + f_6t^6 + 2^{0.5}f_7t^7 + f_8t^8 + 2^{0.5}f_9t^9$.

$h \equiv fg \pmod{2^{255} - 19}$ where

$$\begin{aligned}
h_0 &= f_0g_0 + 38f_1g_9 + 19f_2g_8 + 38f_3g_7 + 19f_4g_6 + \\
h_1 &= f_0g_1 + f_1g_0 + 19f_2g_9 + 19f_3g_8 + 19f_4g_7 + \\
h_2 &= f_0g_2 + 2f_1g_1 + f_2g_0 + 38f_3g_9 + 19f_4g_8 + \\
h_3 &= f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0 + 19f_4g_9 + \\
h_4 &= f_0g_4 + 2f_1g_3 + f_2g_2 + 2f_3g_1 + f_4g_0 + \\
h_5 &= f_0g_5 + f_1g_4 + f_2g_3 + f_3g_2 + f_4g_1 + \\
h_6 &= f_0g_6 + 2f_1g_5 + f_2g_4 + 2f_3g_3 + f_4g_2 + \\
h_7 &= f_0g_7 + f_1g_6 + f_2g_5 + f_3g_4 + f_4g_3 + \\
h_8 &= f_0g_8 + 2f_1g_7 + f_2g_6 + 2f_3g_5 + f_4g_4 + \\
h_9 &= f_0g_9 + f_1g_8 + f_2g_7 + f_3g_6 + f_4g_5 +
\end{aligned}$$

Proof: multiply polys mod $t^{10} - 19$.

$$\begin{aligned}
38f_5g_5 &+ 19f_6g_4 + 38f_7g_3 + 19f_8g_2 + 38f_9g_1,\\
19f_5g_6 &+ 19f_6g_5 + 19f_7g_4 + 19f_8g_3 + 19f_9g_2,\\
38f_5g_7 &+ 19f_6g_6 + 38f_7g_5 + 19f_8g_4 + 38f_9g_3,\\
19f_5g_8 &+ 19f_6g_7 + 19f_7g_6 + 19f_8g_5 + 19f_9g_4,\\
38f_5g_9 &+ 19f_6g_8 + 38f_7g_7 + 19f_8g_6 + 38f_9g_5,\\
f_5g_0 &+ 19f_6g_9 + 19f_7g_8 + 19f_8g_7 + 19f_9g_6,\\
2f_5g_1 &+ f_6g_0 + 38f_7g_9 + 19f_8g_8 + 38f_9g_7,\\
f_5g_2 &+ f_6g_1 + f_7g_0 + 19f_8g_9 + 19f_9g_8,\\
2f_5g_3 &+ f_6g_2 + 2f_7g_1 + f_8g_0 + 38f_9g_9,\\
f_5g_4 &+ f_6g_3 + f_7g_2 + f_8g_1 + f_9g_0.
\end{aligned}$$

Each h_i is a sum of ten products after precomputation of $2f_1, 2f_3, 2f_5, 2f_7, 2f_9$, $19g_1, 19g_2, \ldots, 19g_9$.

Each h_i fits into 64 bits under reasonable limits on sizes of $f_1, g_1, \ldots, f_9, g_9$.

(Analyze this very carefully: bugs can slip past most tests! See 2011 Brumley–Page–Barbosa–Vercauteren and several recent OpenSSL bugs.)

$h_0, h_1, \ldots$ are too large for subsequent multiplication.

Carry $h_0 \to h_1$: i.e.,
replace (h_0, h_1) with
$(h_0 \bmod 2^{26}, h_1 + \lfloor h_0/2^{26} \rfloor)$.
This makes h_0 small.

Similarly for other h_i.
Eventually all h_i are small enough.

We actually use signed coeffs.
Slightly more expensive carries
(given details of insn set)
but more room for $ab + c^2$ etc.

Some things we haven't tried yet:

- Mix signed, unsigned carries.
- Interleave reduction, carrying.

Minor challenge: pipelining.
Result of each insn cannot be
used until a few cycles later.

Find an independent insn
for the CPU to start working on
while the first insn is in progress.

Sometimes helps to adjust
higher-level computations.

Example: carries $h_0 \to h_1 \to h_2 \to h_3 \to h_4 \to h_5 \to h_6 \to h_7 \to h_8 \to h_9 \to h_0 \to h_1$
have long chain of dependencies.

Alternative: carry

$h_0 \to h_1$ and $h_5 \to h_6$;
$h_1 \to h_2$ and $h_6 \to h_7$;
$h_2 \to h_3$ and $h_7 \to h_8$;
$h_3 \to h_4$ and $h_8 \to h_9$;
$h_4 \to h_5$ and $h_9 \to h_0$;
$h_5 \to h_6$ and $h_0 \to h_1$.

12 carries instead of 11,
but latency is much smaller.

Now much easier
to find independent insns
for CPU to handle in parallel.

Major challenge: vectorization.

e.g. `4x a = b + c`
does 4 additions at once,
but needs particular arrangement
of inputs and outputs.

On Cortex-A8,
occasional permutations
run in parallel with arithmetic,
but frequent permutations
would be a bottleneck.

On Cortex-A7,
every operation costs cycles.

Often higher-level operations do a pair of mults in parallel: $h = fg$; $h' = f'g'$.

Vectorize across those mults. Merge $f_0, f_1, \ldots, f_9$ and $f'_0, f'_1, \ldots, f'_9$ into vectors (f_i, f'_i). Similarly (g_i, g'_i). Then compute (h_i, h'_i).

Computation fits naturally into NEON insns: e.g.,

```
c[0,1] = a[0] signed* b[0];
c[2,3] = a[1] signed* b[1]
```

Example: Recall
$C = X_1 \cdot X_2$; $D = Y_1 \cdot Y_2$
inside point-addition formulas
for Edwards curves.

Example: Recall
$C = X_1 \cdot X_2$; $D = Y_1 \cdot Y_2$
inside point-addition formulas
for Edwards curves.

Example: Can compute
$2P, 3P, 4P, 5P, 6P, 7P$ as
$2P = P + P$;
$3P = 2P + P$ and $4P = 2P + 2P$;
$5P = 4P + P$ and $6P = 3P + 3P$
and $7P = 4P + 3P$.

Example: Recall
$C = X_1 \cdot X_2$; $D = Y_1 \cdot Y_2$
inside point-addition formulas
for Edwards curves.

Example: Can compute
$2P, 3P, 4P, 5P, 6P, 7P$ as
$2P = P + P$;
$3P = 2P + P$ and $4P = 2P + 2P$;
$5P = 4P + P$ and $6P = 3P + 3P$
and $7P = 4P + 3P$.

Example: Typical algorithms
for fixed-base scalarmult
have many parallel point adds.

Example: A busy server
with a backlog of scalarmults
can vectorize across them.

Example: A busy server
with a backlog of scalarmults
can vectorize across them.

Beware a disadvantage of
vectorizing across two mults:
256-bit f, f', g, g', h, h'
occupy at least 1536 bits,
leaving very little room
for temporary registers.

We use some loads and stores
inside vectorized `mulmul`.
Mostly invisible on Cortex-A8,
but bigger issue on Cortex-A7.

Some field ops are hard to pair inside a single scalarmult.

Example: At end of ECDH, convert fraction $(X : Z)$ into $Z^{-1}X \in \{0, 1, \ldots, p-1\}$.

Easy, constant time: $Z^{-1} = Z^{p-2}$. $11\mathbf{M} + 254\mathbf{S}$ for $p = 2^{255} - 19$:

```
z2 = z1^2^1

z8 = z2^2^2

z9 = z1*z8

z11 = z2*z9

z22 = z11^2^1

z_5_0 = z9*z22

z_10_5 = z_5_0^2^5
```

```
z_10_0 = z_10_5*z_5_0

z_20_10 = z_10_0^2^10

z_20_0 = z_20_10*z_10_0

z_40_20 = z_20_0^2^20

z_40_0 = z_40_20*z_20_0

z_50_10 = z_40_0^2^10

z_50_0 = z_50_10*z_10_0

z_100_50 = z_50_0^2^50

z_100_0 = z_100_50*z_50_0

z_200_100 = z_100_0^2^100

z_200_0 = z_200_100*z_100_0

z_250_50 = z_200_0^2^50

z_250_0 = z_250_50*z_50_0

z_255_5 = z_250_0^2^5

z_255_21 = z_255_5*z11
```

Can still vectorize
inside a single field op.

Strategy in our software:

50 mul insns starting from

$(f_0,2f_1),(f_2,2f_3),(f_4,2f_5),(f_6,2f_7),(f_8,2f_9)$;

$(f_1,f_8),(f_3,f_0),(f_5,f_2),(f_7,f_4),(f_9,f_6)$;

$(g_0,g_1),(g_2,g_3),(g_4,g_5),(g_6,g_7)$;

$(g_0,19g_1),(g_2,19g_3),(g_4,19g_5),(g_6,19g_7),(g_8,19g_9)$;

$(19g_2,19g_3),(19g_4,19g_5),(19g_6,19g_7),(19g_8,19g_9)$;

$(19g_2,g_3),(19g_4,g_5),(19g_6,g_7),(19g_8,g_9)$.

Change carry pattern to vectorize,
e.g., $(h_0,h_4) \to (h_1,h_5)$.

Core arithmetic: 100 cycles
on mul insns for each field mul.
Squarings are somewhat faster.

Some loss for carries etc.

ECDH: $\approx$10 field muls $\cdot$ 255 bits.

More detailed analysis:
356019 cycles on arithmetic;
$\approx$78% of software's total
Cortex-A8-fast cycles for ECDH.
Still some room for improvement.

Core arithmetic: 100 cycles
on mul insns for each field mul.
Squarings are somewhat faster.

Some loss for carries etc.

ECDH: $\approx$10 field muls $\cdot$ 255 bits.

More detailed analysis:
356019 cycles on arithmetic;
$\approx$78% of software's total
Cortex-A8-fast cycles for ECDH.
Still some room for improvement.

Each CPU is a new adventure.
e.g. Could it be better to use
Cortex-A7 FPU with radix $2^{21.25}$?

Much more work to do

https://bench.cr.yp.to:
benchmarks for (currently)
2137 public implementations of
hundreds of crypto primitives—
39 DH primitives,
56 signature primitives,
304 authenticated ciphers, etc.

Much more work to do

https://bench.cr.yp.to:
benchmarks for (currently)
2137 public implementations of
hundreds of crypto primitives—
39 DH primitives,
56 signature primitives,
304 authenticated ciphers, etc.

Many interesting primitives
are far slower than necessary
on many important CPUs.

Much more work to do

https://bench.cr.yp.to:
benchmarks for (currently)
2137 public implementations of
hundreds of crypto primitives—
39 DH primitives,
56 signature primitives,
304 authenticated ciphers, etc.

Many interesting primitives
are far slower than necessary
on many important CPUs.

Exercise: Make them faster!